On Development, Feasibility, and Limits of Highly Efficient CPU and GPU Programs in Several Fields

Fast Parallel SIMDized GPU-accelerated Reed-Solomon Encoding, Heterogeneous Linpack Benchmark, and Event Reconstruction for the ALICE Experiment

Dissertation

for attaining the PhD degree

of Natural Sciences

submitted to the Faculty for Computer Science and Mathematics

of the Johann Wolfgang Goethe University

in Frankfurt am Main

by

David Rohr

from Mannheim

Frankfurt 2013

(D 30)

accepted by the Faculty for Computer Science and Mathematics of the

Johann Wolfgang Goethe University as a dissertation.

Dean:	Prof. Dr. Thorsten Theobald
Expert assessor:	Prof. Dr. Volker Lindenstruth Prof. Dr. Udo Kebschull
Date of the disputation:	23.6.2014

Abstract

On Development, Feasibility, and Limits of Highly Efficient CPU and GPU Programs in Several Fields

Fast Parallel SIMDized GPU-accelerated Reed-Solomon Encoding, Heterogeneous Linpack Benchmark, and Event Reconstruction for the ALICE Experiment

With processor clock speeds having stagnated, parallel computing architectures like GPUs have achieved a breakthrough in recent years. Despite of their shortcomings concerning efficient execution of serial tasks, their sheer parallel processing power makes them predestined for parallel applications while the simple construction of their cores makes them unbeatably power efficient. Unfortunately, old programs cannot profit through simple recompilation, and adaptation usually requires rethinking and modifying algorithms. Modern clusters are often designed heterogeneously and offer different processors for different applications. In order not to waste the available compute power, highly efficient programs are mandatory. This thesis is about the development of fast algorithms and their implementations on modern CPUs and GPUs, about the maximum achievable efficiency with respect to peak performance and to power consumption respectively, and about feasibility and limits of programs for CPUs, GPUs, and heterogeneous systems. Three totally different applications from distinct fields, which were developed in the course of this thesis, are presented.

The ALICE experiment at the LHC studies heavy-ion collisions at high rates of several hundred Hz, while every collision produces thousands of particles, whose trajectories must be reconstructed. For this purpose, ALICE HLT TPC track reconstruction and track merging have been adapted for GPUs outperforming the fastest available CPUs by about a factor three. Since the beginning of 2012, the tracker has been running in nonstop operation on 64 nodes of the ALICE HLT providing full real-time track reconstruction.

The Linpack benchmark employs matrix multiplication (DGEMM) to solve a dense system of linear equations and is the standard tool for ranking compute clusters. Heterogeneous multi-GPU-enabled versions of DGEMM and Linpack have been developed supporting CAL, CUDA, and OpenCL as backend. An elaborate lookahead algorithm hides the serial CPU-bound tasks of Linpack behind DGEMM execution on the GPU reaching the highest efficiency on GPU-accelerated clusters. Employing this implementation, the LOEWE-CSC cluster ranked place 22 in the November 2010 Top500 list of the fastest supercomputers, and the Sanam cluster achieved the second place in the November 2012 Green500 list of the most power efficient supercomputers.

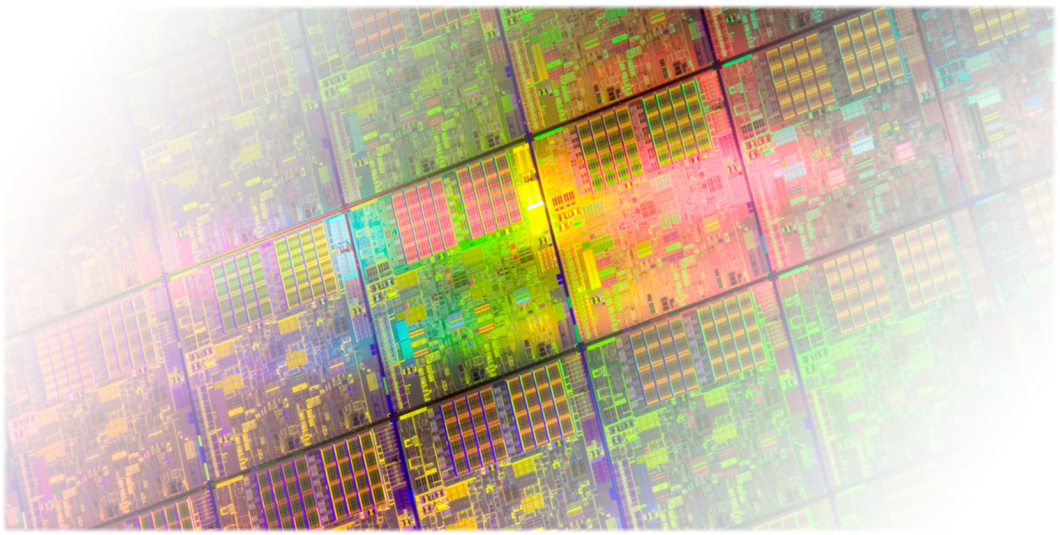
Failure erasure coding enables failure tolerant data storage and is an absolute necessity for present-day computer infrastructure. The mathematical theory behind the codes involves matrix-computations in finite fields, which are not natively supported by modern processors and hence computationally very expensive. This thesis presents a novel scheme for fast encoding matrix generation and demonstrates fast implementations for the encoding itself, which use exclusively either integer or logical vector instructions. Depending on certain parameters, they always hit different hard limits of the hardware: either the maximum attainable memory bandwidth, or the peak instruction throughput, or the PCI Express bandwidth limit when GPUs or FPGAs are employed.

The thesis demonstrates that in most cases GPU implementations can be as efficient as their CPU counterparts with respect to the available peak performance. With respect to costs and power consumption, they are much more efficient. For this purpose, complex tasks must be split in serial as well as parallel parts such that multithreaded pipelines and asynchronous DMA transfers can hide CPU bound tasks ensuring continuous GPU kernel execution. Few cases are identified where this is not possible due to PCI Express limitations or not reasonable because practical GPU languages are missing.

Table of Contents

Part I

Introduction



Source of Image: Intel Press Room Website.

Chapter 1

Motivation & Outline

During the last decades, compute performance of microchips has been rapidly and steadily increasing. Moore's law states that diminution of manufacturing processes permits a doubling of transistor count and performance every 18 months. It is physically evident that this exponential growth must hit a limit in the near future. Accordingly, the increase of processor clock speeds, which had followed Moore's prediction for quite some time, has already begun to stagnate in recent years because the heat density has reached an unmanageable level. Hardware manufacturers have accomplished to find other ways of improving their chips and keep on track with Moore's law: Advanced designs significantly increase the work performed per clock cycle but – obviously – cannot go beyond several orders of magnitude. Therefore, to overcome the serial computing limit, broader chip designs perform various tasks in parallel.

There exist two traditional concepts for parallel computation: vectorization and parallelization. Vectorization means the processor executes the same instruction not on a single data element but on a vector (SIMD – Single Instruction Multiple Data). It was popular in the age of early supercomputers, has been almost completely replaced by parallelization in the meantime, and has experienced a revival recently. Parallelization stands for parallel execution of multiple instruction streams on different datasets (MIMD – Multiple Instruction Multiple Data). Conventionally, independent compute nodes connected by a network or multiple processors in one node perform parallel computation. This has changed in the way that nowadays processors contain multiple cores. Each core resembles a traditional processor and processes an independent instruction stream. While it is often relatively easy and needs little effort to adapt a program to use multiple cores, utilization of vector capabilities usually requires refactoring large parts of the code and enhancing the implementation of algorithms or even algorithms themselves.

In order to provide competitive compute power, every state-of-the-art processor, irrespective of whether the system is part of a compute cluster or an autonomous computer, must consist of multiple cores, which all support vector extensions. Besides the usual processors (CPUs), there are other hardware designs which excel under this perspective, among them special accelerators like the Cell processor, the ClearSpeed chip used e. g. in the Tsubame cluster, but also off-the-shelf graphics cards (GPUs). Graphics processing, where the same task must be performed independently for each pixel, is a prime example for parallel computing and GPUs have been optimized particularly for this purpose. Today, the former fixed function units of GPUs have been replaced by flexible programmable shaders, which can be used for general-purpose programming. This thesis covers GPUs and CPUs (and presents one FPGA implementation), where CPUs are necessary for data transfer as well as pre- and postprocessing, provide non-negligible compute performance themselves, and are present in an accelerator-enabled system anyway.

There are several APIs for programming GPUs, all providing similar interfaces, mostly supporting C, C++, or even GPU assembler code. For different tasks in this thesis, different APIs are employed, whichever is suited best. However, the subject of this thesis is not a comparison of GPU programming models. Instead, the focus is put on exploiting the potential of GPUs and

modern CPUs for different problems. While it is comparatively easy to distribute a task among the different multiprocessors on a GPU (which correspond to the CPU cores), usage of the vector capabilities can require as much or even more effort as for the CPU. It depends greatly on the field of application how much time and effort are actually necessary but also how well the GPU can be utilized at all. The problems analyzed in this thesis can be categorized as follows:

- **Synthetic Benchmarks**: Measurements of a quantity directly related to a processor characteristic, e. g. memory bandwidth. Certain low-level problems which can be mapped perfectly to the hardware fall in this category as well. One example is matrix-matrix multiplication, which requires exclusively alternating additions and multiplications perfectly matching the processor capabilities. Such benchmarks are supposed to achieve close to peak performance and indicate problems in the system if not. In general, they can verify whether a processor delivers its specified performance under optimal conditions.
- **Semi-Synthetic Benchmarks**: Tasks which may be complex as a whole but whose computational hot spot belongs to the above category. An example is matrix factorization, which is based on matrix-matrix multiplication. Such benchmarks consist of a synthetic task achieving peak performance and making up most of the runtime and other tasks, which cannot be processed that efficiently. The main objective is to hide the execution time of those tasks behind the synthetic task, for instance by pipelining. Problems in this category require more optimization effort but can achieve almost the same performance as synthetic problems.
- **Application Benchmarks**: The most general case. A problem in this category can consist of various subtasks, all of which contribute to the runtime differently; perhaps some of them cannot be implemented efficiently, and each of them requires optimizations on its own, which does not mean that they are independent, though.

Loosely speaking, this thesis presents several optimized GPU and CPU implementations for different problems that cover all of the above categories and analyzes their efficiency. The problems belong to the following fields of applications:

- **Experiments in high energy physics** pose large challenges for many sciences, among them physics, electronics, engineering, and also computer science. The latter is responsible for the analysis of the massive amount of data generated by the detectors. Part II of this thesis presents a GPU program for reconstructing particle trajectories in real-time. The developed implementation is used in ALICE, which is one of the four major experiments of the Large Hadron Collider at CERN in Geneva.
- **Benchmarking modern heterogeneous compute clusters** is a complex task. The available performance for real applications cannot simply be calculated by accumulating the peak performances of all available processors because many other factors such as memory or network are important. Linpack is the most popular benchmark for ranking supercomputers. To obtain good results, it is usually optimized for each particular cluster. With the emergence of heterogeneous clusters employing GPUs or other hardware accelerators, the tuning effort has even increased. Part III is about an optimized Linpack implementation for AMD GPUs, especially for the LOEWE-CSC cluster and the Sanam cluster.
- **Failure erasure coding** is a necessity for reliable data storage. Encoding that tolerates the loss of multiple hard disks or servers cannot be realized with simple parity and is computationally very expensive. Part IV presents various algorithms and optimized implementations, especially a new approach for creating codes over residue class rings, and a new general implementation of codes based on matrix multiplication, which is only limited by either peak memory bandwidth or maximum instruction throughput of the hardware.

The rest of this introductory part of the thesis, complemented by Appendices A, C, E, and H, introduces the hardware, explains the boundary conditions for the benchmarks, and provides the basis to comprehend all optimizations. The following parts deal with the above fields of application one by one. In the end, Part V compares the achieved results with each other, between CPU and GPU, and in relation to the theoretical peak performance arising from the hardware specifications.

Chapter 2

CPUs

For an in-depth understanding of the optimizations applied later, fundamental knowledge about the hardware is required. This chapter gives a brief introduction to the employed CPUs and their features while the next chapter covers the GPUs.

2.1 Intel

2.1.1 Nehalem

Intel introduced the quad-core Nehalem architecture in November 2008. It is the first Intel CPU with integrated memory controller.[1] A triple-channel memory interface provides an improved memory bandwidth compared to traditional dual-channel interfaces.

In contrast to the former Intel Core2 quad-core CPUs, which are built by internally connecting two dual-core CPU dies[2] in one package, the Nehalem processors are native quad-cores. The **Hyperthreading** feature makes all physical CPU cores available as two virtual CPU cores to the operating system. Two virtual cores share a common ALU[3] while registers and control logic are present twice. When one virtual core waits for data from memory, the other one can use the ALU. This can help to reduce latencies, especially for memory-bound applications. Since one virtual core can already use the ALU to the full extent, the CPU's peak performance is not increased. Indeed, scheduling an additional virtual core can even have a negative effect. Hence, the benefit of Hyperthreading depends greatly on the application. For instance, the Linpack benchmark presented in Part III achieves close to peak performance and is faster with Hyperthreading deactivated. The ALICE TPC tracker in Part II performs mostly random memory access and benefits from Hyperthreading. In [Roh 10 I, 9.1.2], it is shown that the tracker performance scales almost linearly with the number of Nehalem cores used, up to exactly half the number of virtual cores.[4] Using all virtual cores improves the performance, but not by a factor of two.

The Nehalem generation supports the SSE[5] vector instructions[6] in revision 4.2. SSE provides integer and floating point instructions for vectors of four 32-bit or two 64-bit values. It does not specify an FMA[7] instruction. However, as all other CPUs presented here, the Nehalem can process additions and multiplications in parallel, as long as they are independent. Thus, it can do at maximum four double precision or eight single precision calculations per cycle.

---

1 AMD integrated the memory controller into the CPU with the introduction of the Opteron CPU years before.
2 A die is a raw unpackaged semiconductor chip.
3 **A**rithmetical **L**ogical **U**nit.
4 The OS scheduler ensures that only one virtual core of each physical core is used in this situation.
5 **S**treaming **SIMD** **E**xtensions (SIMD stands for **S**ingle **I**nstruction **M**ultiple **D**ata).
6 See [Kre 09] for details on vectorization.
7 **F**used-**M**ultiply-**A**dd is widely used in matrix operations.

2.1.2 Westmere & Sandy Bridge

Westmere is a die-shrink of Nehalem. It comes in quad- and hexa-core variants.[8] Some new special instructions primarily for cryptographic applications are supported. However, they are not relevant for this thesis.

Sandy Bridge is the successor of Nehalem and Westmere. The most important change is the increase of vector size by a factor of two. The new vector extension is called AVX[9] instead of SSE. FMA3 and FMA4 are extensions upon AVX using the VEX instruction coding introduced with AVX. The consumer grade Sandy Bridge is restricted to two memory channels offering less bandwidth than Nehalem, the server grade CPU has four memory channels. The server-grade versions of Intel CPUs are called Xeon.

2.2 AMD

2.2.1 Magny-Cours

The Magny-Cours processor by AMD is available in an eight-core and a twelve-core variant. Almost exclusively the twelve-core model is considered throughout this thesis. Internally, the twelve-core Magny-Cours consists of two six-core dies connected via **HyperTransport** (HT).[10] Each die provides a dual-channel memory interface. Systems with two Magny-Cours CPUs thus provide 24 cores on four dies with eight memory channels (Fig. 2.1 visualizes the situation). This is the most complicated **NUMA**[11] architecture used in this thesis.

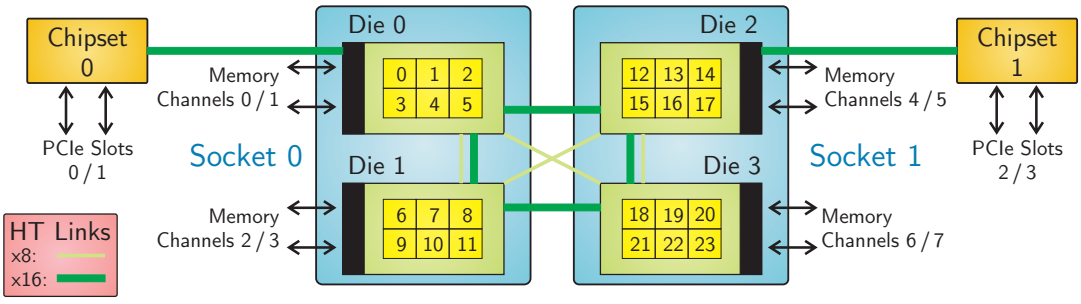


Figure 2.1: Block Diagram of Dual Socket Magny-Cours CPU with NUMA

For a program running on one core, there are three different areas in the memory hierarchy:[12]

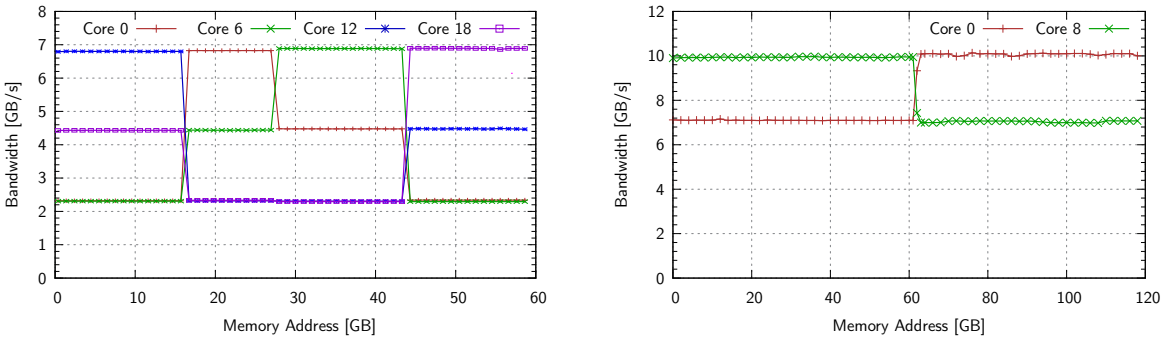
- **Same Die:** The memory connected to the memory controller of the die where a program is executed.
- **Same Socket:** The memory connected to the second die on the same Magny-Cours CPU, thus connected via HyperTransport.
- **Other Socket**[12]**:** The memory connected to one of the two dies of the second Magny-Cours CPU on the other socket (connected via HyperTransport as well, but farther away).

[8] There is an octo-core version of Westmere for multi-socket servers.
[9] **A**dvanced **V**ector **I**nstructions.
[10] AMD puts two six-core dies in one package – in the same way as Intel did for the quad-core Core2 processors.
[11] **N**on **U**niform **M**emory **A**rchitecture.
[12] In fact, there are four categories in the memory hierarchy since within the "Other Socket" category one can distinguish the two dies, which are connected by different links. However, Fig. 2.2a reveals that the width of the HyperTransport link is not important – at least not for a single thread.

2.2.2 Interlagos

Interlagos belongs to the Bulldozer family, which is the successor of Magny-Cours. It has up to sixteen cores. Two cores are grouped to a module each. The AVX extension and FMA4 are supported, but only one core per module can access them at a time and execute only one AVX instruction per cycle. Hence, the peak performance per core is identical to Magny-Cours.[13]

2.3 Summary



(a) AMD Magny Cours [IX] (b) Intel Sandy Bridge [VI]

Figure 2.2: Single Threaded NUMA Memory Performance
(Memory Read Bandwidth versus Memory Address)

The actual location of data in memory has a great influence on memory bandwidth – for both CPUs and hardware accelerators connected via PCI Express (PCIe) [Dre 07].[15] Intel CPUs show a smaller dependency than AMD CPUs.[16] Synthetic measurements in Fig. 2.2 show that the AMD system has three performance levels while the Intel system has only two. They demonstrate Intel's superior per-core memory bandwidth due to direct access to three memory channels. It is clearly visible which memory address is connected by which CPU die.

As a summary, Table 2.3 gives an overview of the characteristics of the CPUs mentioned.

CPU	Cores	Frequency [GHz]	Hyper-threading	Multi-Die Package	Flops / Cycle[17] (Single / Double)	Memory Channels
Nehalem	4 – 6	2.2 – 4.0	Yes	No	4 / 8	3
Sandy Bridge	4 – 8	3.3 – 3.7	Yes	No	8 / 16	2 / 4
Magny-Cours	12	2.1 – 2.3	No	Yes	4 / 8	4
Interlagos	16	2.1 – 2.6	Special	Yes	4 / 8[13]	4

Table 2.3: Overview of the CPUs used throughout this Thesis

[13] Interlagos has an FMA4 peak performance in double precision of 8 $^{\text{Flop/Cycle}}$ per core as long as only half of the cores (one per module) are used, otherwise it is only 4 $^{\text{Flop/Cycle}}$.

[14] The author would like to thank Matthias Kretz for providing the synthetic benchmark results. The plots show the single-thread memory read throughput. The bandwidth test is run on different cores corresponding to different dies. On the AMD system, CPU 0 belongs to die 0, CPU 6 to die 1, and so on. The Intel system has only two dies. (For reference: the bandwidth of all cores on one die is identical.)

[15] On Magny-Cours, Interlagos, and Nehalem, the PCI Express interface is implemented in the chipset which is connected via HyperTransport (or the Intel pendant QPI – Quick Path Interconnect) to only one of the CPUs. On Sandy Bridge the PCI Express interface is integrated in the processor. In both cases, a PCI Express transfer to memory on the other CPU has to pass through the CPU which connects the PCI Express hardware.

[16] Fig. 11.4 in Section 11.2.3.1 shows the impact of NUMA on the performance of an application on an AMD system. Fig. 12.13 shows that the effect is weaker on Intel CPUs.

[17] Only one CPU core is considered here. Two numbers for double and single precision are given.

Chapter 3

GPUs

Some years ago, the increase in CPU clock speed began to stagnate while new processors started to excel at SIMD and multi-core capabilities. A different, originally non-general-purpose, and very parallel chip design had been used in **G**raphics **P**rocessing **U**nits (GPUs) for quite some time. Graphics processor manufacturers recognized the potential of GPUs for HPC[1] applications and have been steadily adapting their chips for usage in compute clusters. They started with support for the C programming language and have recently begun to offer ECC[2] memory and C++ support. This makes GPUs suitable for general-purpose programming (GPGPU).

There are several frameworks available for GPU programming. CUDA[3], OpenCL[4], and DirectCompute are the most popular general-purpose frameworks. The CAL[5] framework is still used for low-level programming while AMD has discontinued its Stream framework based on Brook+ [Adv 09, §2]. New approaches like C++ AMP [MS], OpenACC, and HMPP have potential to simplify GPU programming and Intel presents a different paradigm with the MIC[6] [Int 10].

Typically, a GPU has some fixed function units, e. g. for rasterization, and many programmable shaders (ALUs). Recently, the shaders have become more and more important as well as highly flexible. Many tasks formerly performed by fixed function units can be processed by shaders in software today.[7] Fixed function units do not play a significant role in HPC but shaders with their highly parallel architecture can bring optimized code to yet unknown performance levels.

This section gives a brief introduction to a generalized GPU chip and all features available on today's hardware. Of course, not every feature is supported by each of the real GPUs. The GPUs used for this thesis are described in Appendix A by specializing the generalized design presented here. Since only the shaders (ALUs) are relevant for HPC, fixed function units are omitted.

3.1 General GPU Architecture

Fig. 3.1 shows a scheme of an abstract GPU. A typical GPU combines a graphics processor and a certain amount of global memory on one board. (There are GPUs employing two graphics processors, but then both processors are equipped with their own global memory. Hence, these so called **dual-GPUs** can be considered just as two distinct GPU boards.) The graphics processor

[1] **H**igh **P**erformance **C**omputing.
[2] **E**rror **C**orrection **C**ode.
[3] **C**ompute **U**nified **D**evice **A**rchitecture was the first framework for general-purpose programming. It is currently restricted to NVIDIA as single vendor and allows programming in C and C++.
[4] OpenCL is an open industry standard adopted for a huge variety of compute devices. This flexibility makes low-level optimization more difficult than with CUDA, which is closer to the hardware. Still, OpenCL resembles CUDA very much. At the moment, it supports only plain C, but recently AMD started to offer a C++ extension.
[5] **C**ompute **A**bstraction **L**ayer allows writing applications in GPU assembler on AMD hardware.
[6] **M**any **I**ntegrated **C**ores emerged from the Larrabee, which was discontinued. It can be programmed in C++ with vector-intrinsics. A different name for the MIC is Xeon Phi.
[7] In fact, Intel even tried to perform almost all tasks in software for the canceled Larrabee GPU [Int 08].

consists of m independent **multiprocessors**.[8,9] Each multiprocessor has n arithmetical logical units but only one instruction decoder. This means that all ALUs must execute the same common instruction.[10] Each multiprocessor can execute a large number of threads (also called **work-items**), which may exceed the number of ALUs allowing the scheduler to hide memory latencies. Such a group of threads running on one multiprocessor is called a **block** (or a **work-group**). The blocks are divided into groups of w threads each, which are called **warps** (or **wavefronts**). These warps build the basis for the GPU scheduler. Each cycle the scheduler takes threads out of one warp that are ready to execute and dispatches them to the ALUs or the memory fetch units. Typical warp sizes are $w = 32$ for NVIDIA and $w = 64$ for AMD.

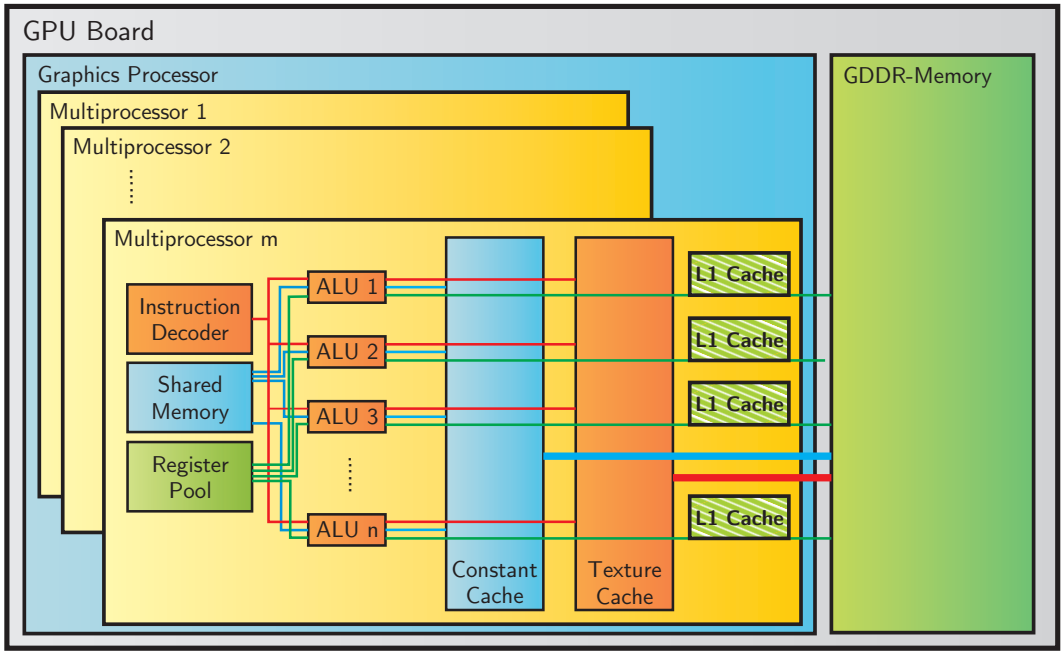


Figure 3.1: A generalized GPU Design[11]

It must be noted that all threads within a warp must execute one identical instruction. If different threads in one warp take different branches in the code, thus decoding different instructions, the execution is serialized. This is called **warp-serialization**. On some GPUs there is one exclusion to this: **Very-Long-Instruction-Words** (VLIW): k shaders are grouped into a k-dimensional (k-D) VLIW shader and k different (small) instructions are encapsulated into one (large) VLIW for the k-D shader. These k instructions are executed in parallel, even though they are not identical.[12] In other words, all VLIW shaders must execute the same common k instructions. They are inefficient if there are not k instructions executable in parallel.

A program executed on a GPU (or the sources for the GPU programm respectively) is called a **kernel**. It is executed in a **grid** of blocks. Each thread in the grid executes the same kernel with threads in one block running on one multiprocessor. The number of blocks may exceed the multiprocessor count. Different blocks are executed on different or on the same multiprocessor –

[8] Other names for multiprocessors are **CU**s (AMD Graphics Core Next), **CUDA-Core**s (NVIDIA Fermi), and **SIMD**s (AMD Cypress, Cayman, and Tahiti).

[9] Naturally, there are one or multiple memory controllers on the chip, each responsible for a group of multiprocessors. However, for the programmer it does not make a difference for two reasons: first, the multiprocessors address the memory transparently; second, there is no control over which part of the program runs on which multiprocessor.

[10] Due to the single instruction decoder, the multiprocessor of a GPU resembles very much a vector processor. However, it does not necessarily operate on vectors, but the data can be scattered in memory.

[11] DDR stands for **D**ouble **D**ata **R**ate and GDDR is optimized for graphics cards.

[12] VLIWs are included in the OpenCL standard. If not supported by the hardware, a VLIW can easily be simulated by k sequential scalar instructions.

sequentially or even concurrently if enough resources are available. Each multiprocessor offers a **register pool**[13] and an amount of **shared memory** (or **LDS**)[14], which is shared among all threads within one block. All threads have a compile-time determined register requirement and each block has predefined shared memory requirements. This restricts the number of blocks a multiprocessor can execute simultaneously and it restricts the number of threads per block as well. Newer GPUs can partition their set of multiprocessors for executing multiple kernels in parallel.

The kernels are started by the host but executed asynchronously. GPUs usually have DMA[15] engines that can handle DMA transfers independent of kernel execution. Thus, execution of host code, of GPU code, and DMA transfers can overlap. The host memory for the DMA transfer must be page locked.[16] Direct access to this page locked host memory can be given to the GPU kernel. This feature is called **Zero-Copy**. In the other way around, GPU memory can be mapped into the virtual address space of host applications (also called Zero-Copy).

Each GPU has at least two different caches: A constant data cache[17] and a texture cache, which is read-only, optimized for two- or three-dimensional access patterns, and can perform some simple on-the-fly data format conversions. Some GPUs offer general-purpose L1 and L2 caches. Each thread maintains a local set of variables, which are stored either in registers or in **private memory**.[18] As private memory is not on chip, in contrast to shared memory, access to it is very slow. Therefore, if no L1 cache is present, usage of private memory should be avoided wherever possible. For fast access to both shared and global memory, several vendor specific coalescing rules must be obeyed, as presented e. g. in [Roh 10 I, 3.7] or [NVI 11, F.4.2 / F.4.3] for NVIDIA hardware.

Most GPUs are available in a consumer-grade and a professional-grade version, where the latter one is particularly optimized for the HPC market, e. g. by offering more memory, ECC support, better double precision support, or special cooling that is optimized for rack-mount servers.

Appendix A lists the GPU architectures used within this thesis in detail. Fig. 3.2 gives an overview how GPU and CPU performance have evolved.

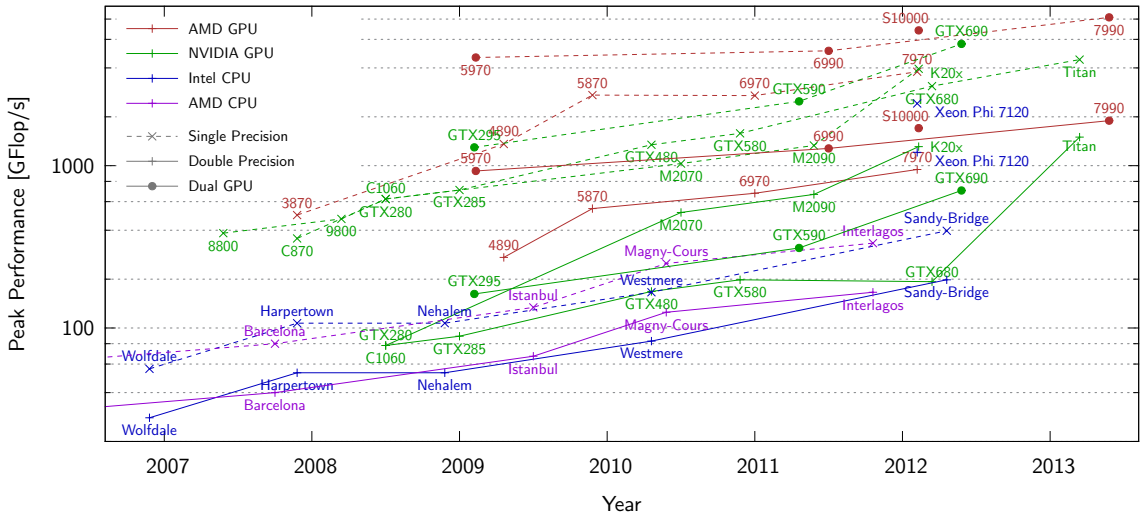


Figure 3.2: GPU and CPU Performance Evolution

[13] Registers are allocated dynamically to all the threads currently executed on the multiprocessor.
[14] In OpenCL, shared memory is called local memory, which conflicts with the NVIDIA notation. Therefore, in the following it is denoted exclusively by shared memory. LDS stands for **L**ocal **D**ata **S**torage.
[15] **D**irect **M**emory **A**ccess.
[16] For optimal performance, features as (speculative) write-combining, uncacheable memory, or GPU-registered memory can be used.
[17] A segment of usually up to 64 KB of constant global memory can be cached in constant memory.
[18] It is a distinct part of global memory for every thread. Private memory is called local memory in CUDA terminology, which conflicts with the OpenCL notation.

Chapter 4

Benchmark Proceeding & Statistics

4.1 Conventions & Statistics

In literature and vendor specifications depending on the field of application, SI prefixes such as "kilo", "mega", and "giga" are used inconsistently, e. g. giga can mean either 10^9 or 2^{30}, which differs by a factor $1.024^3 \approx 1.0737$. In this thesis, all prefixes denote exclusively powers of ten, i. e. kilo means 10^3, mega means 10^6, giga means 10^9 and terra means 10^{12}. There is one single exception: For memory sizes (not memory bandwidths) it means powers of two.

Appendix H gives an overview of all nodes used for benchmarks throughout this thesis. The nodes are indexed by **Latin numbers**. Latin numbers in square brackets at the end of performance plot captions refer to the test-node the benchmarks were taken on (e. g. Fig. 4.1a is based on measurements taken on node [I]). It should be kept in mind that even benchmarks taken on the same node at different times are not necessarily comparable, e. g. due to different software versions. Thus, if not explicitly stated differently, only the values inside one plot but not between plots can be compared. Figures showing a comparison of different hardware refer to all participating nodes in the figure caption. It will be clear from the context which dataset refers to which node.

The following sections demonstrate that usually the deviation of GPU kernel execution time is so small that with relatively few runs it is possible to force the statistical error to an insignificant magnitude. Throughout the thesis, the number of benchmark repetitions is chosen such that the collected statistics suffice for a relative statistical error (relative **R**oot **M**ean **S**quare (**RMS**)) below **0.2** %. In most cases, the error is negligible and error bars are omitted for an easier inspection. Since the reference clocks set by hardware vendors on different mainboards usually differ by this order of magnitude, even though the installed hardware components are identical in construction, a higher accuracy is unnecessary.

4.2 Benchmark Conditions for NVIDIA

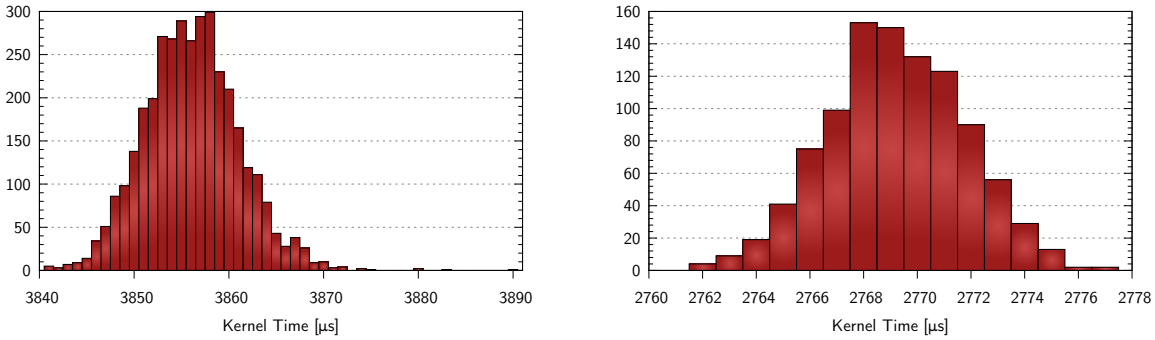
In [Roh 10 I, 5.6], the statistical distributions of the execution times of NVIDIA GTX285 GPU kernels and multi-threaded CPU programs are analyzed. They are more or less Gaussian distributed and the Root Mean Square is a good model for estimating the deviation from the average of multiple measurements with reasonable effort. The exact conclusions are the following:

- The kernel execution time follows almost entirely a Gaussian distribution with the error so small that usually already five runs give sufficient statistics to reach the 0.2 % relative RMS limit (e. g. for the kernel employed in Fig. 4.1).
- The total execution time of a mainly GPU-based program with many kernel invocations is Gaussian distributed, as long as the CPU code is pinned to the CPU core closest to the

GPU and the FIFO scheduler is used. The RMS is generally a bit bigger than for the kernel only (see Fig. 4.2), but still few runs are sufficient.

- Execution times of single-threaded CPU-bound programs are also Gaussian distributed, as long as the thread is pinned to one core and the real-time scheduler is used.
- Without the FIFO scheduler or CPU pinning, dirt-effects appear in the distribution. In this case, there is one large Gaussian distributed peak and multiple other (non-Gaussian) peaks, some order of magnitudes smaller and possibly far away from the primary peak.
- The RMS for multi-threaded CPU tasks is generally larger compared to single-threaded or GPU programs. Scheduling plays an important role. It requires 50 or more runs to gain enough statistics.

These conclusions are based on a GTX285 GPU. Fig. 4.1 reveals that the time distributions of kernels on Fermi and on GTX285 are alike. Thus, the above conclusions are valid for Fermi GPUs as well. All following benchmarks are taken with proper CPU pinning and the real-time scheduler, each measurement repeated often enough to get the stated accuracy of 0.2 %.



(a) GTX285 [I]

(b) Fermi [III]

Figure 4.1: Exemplary Kernel Time Distribution (ALICE GPU Tracklet Constructor)

4.3 Benchmark Conditions for AMD

Interestingly, the Cypress GPU (see Fig. 4.3) does not show a Gaussian distribution. Still, the deviation is of the same order of magnitude as for Fermi. Benchmarks are taken under the same conditions as for NVIDIA.

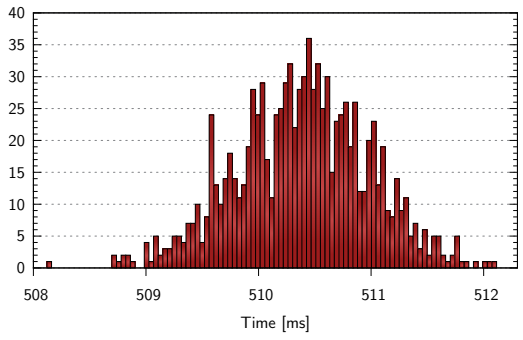


Figure 4.2: Time Distribution of a Full Run of the ALICE GPU Tracker on Fermi [III]

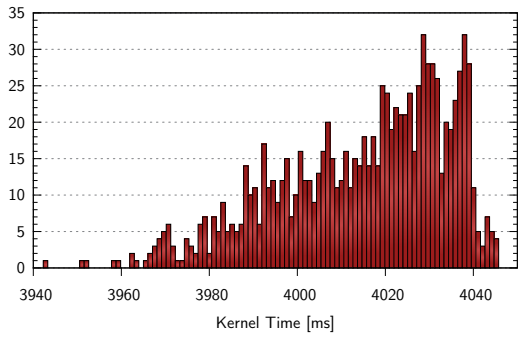
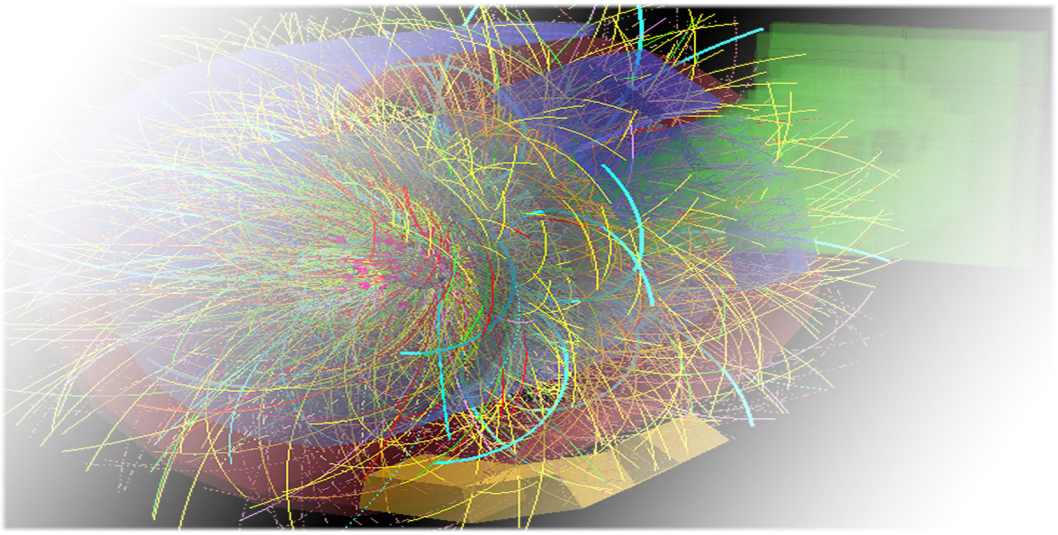


Figure 4.3: Cypress Kernel Time Distribution (DGEMM Kernel) [V]

Part II

Event Reconstruction for the ALICE Experiment



Source of Image: CERN [CER 11].

Chapter 5

Introduction

5.1 The ALICE Detector of the LHC Experiment

The **L**arge **H**adron **C**ollider (LHC) is the most powerful particle accelerator today. It is located at CERN (**C**onseil **E**uropéenne pour la **R**echerche **N**ucléaire) in Geneva (see [Brü+ 04]). **A** **L**arge **I**on **C**ollider **E**xperiment (ALICE) is one of the four major experiments (see [ALI 95]). The LHC can collide protons as well as heavy ions (primarily lead). In contrast to the general purpose detectors ATLAS and CMS, ALICE is specifically designed for the heavy ion case. Of particular interest for ALICE is the so called **Q**uark **G**luon **P**lasma, a state where hadronic matter dissolves into its constituents. The LHC commenced operation in November 2009. Yet, the LHC has been running with reduced energy and luminosity. It is going to reach its design-values after an upgrade of the magnet interconnects scheduled to finish in the end of 2014. In November and December 2010, the first heavy ion collisions took place. Figures 5.1 and 5.2 show an overview of LHC and ALICE.

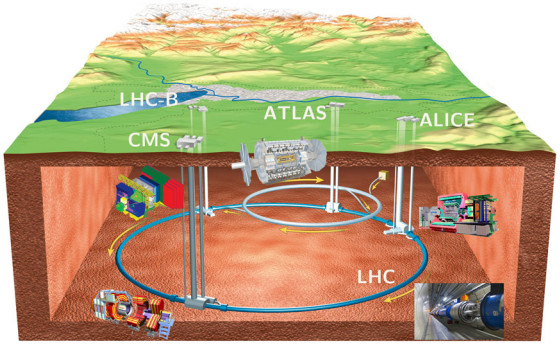


Figure 5.1: LHC with four major Experiments [CER 06]

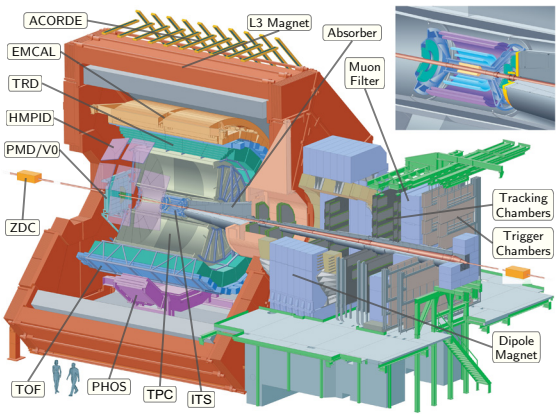


Figure 5.2: The ALICE Detector [ALI II]

An event comprises all data obtained from the detectors for one collision.[1] Heavy ion events result in many more particles inside the experiment compared to proton events. Reconstructing their trajectories is a very complex task, which is called **tracking**. ALICE' primary detector for this purpose is the TPC[2], which is a cylindrical chamber filled with gas. It is split in two halves, which are further subdivided into 18 trapezoidal readout chambers each. These, in total 36 chambers, are called **slices** (or sectors). (Fig. 5.3 shows an illustration of the TPC and the splitting in slices.)

[1] One collision can contain multiple interactions. Due to the high collision rate, events also contain leftovers from the previous collisions. This is called pileup.

[2] **T**ime **P**rojection **C**hamber: See [ALI III] for more information.

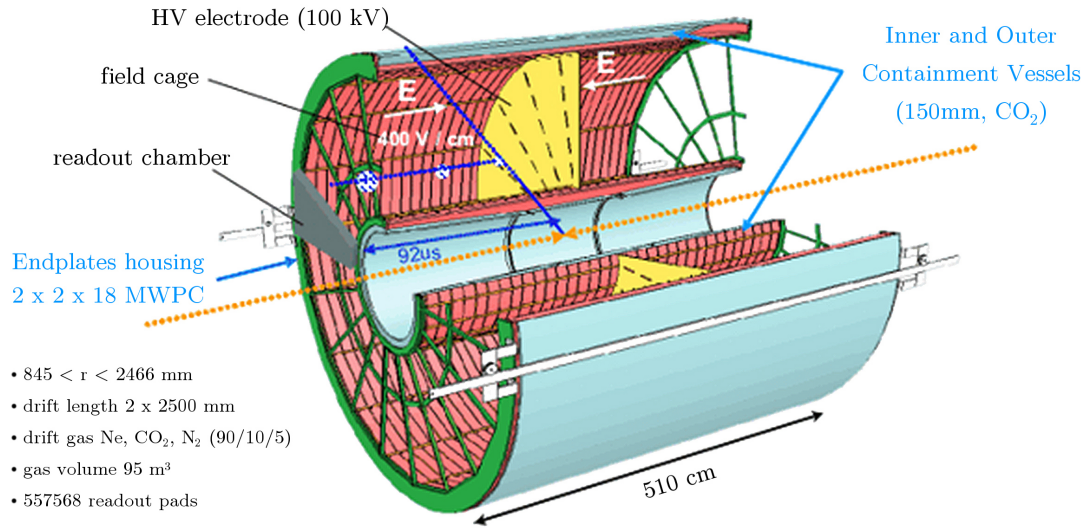


Figure 5.3: ALICE Time Projection Chamber [ALI III]

5.2 The High Level Trigger

Most events contain no new physics (such as new, yet unknown particles) but only reproduce effects which have already been studied sufficiently such that no more data are needed. Interesting events are quite rare. By filtering for relevant events, the raw data obtained from the detectors can be compressed by multiple orders of magnitude – without losing scientifically meaningful data. This data rate decrease reduces storage costs dramatically and can, if the data rate exceeds the storage capacity, ensure collecting more relevant data.

Triggers observe data from a certain subset of detectors and do a fast search for indications of relevant events like jets or high momentum electrons. If they find something, they trigger the readout of other (often slower) detectors and / or the storage of the event for later analysis. A hierarchy of triggers is applied in ALICE, searching for physically relevant events – or parts thereof. The last step in this trigger hierarchy is the **H**igh **L**evel **T**rigger (HLT) [ALI 04], which, in contrast to the lower level triggers, is a software trigger. The HLT is capable of a full real-time event reconstruction and performs an on-the-fly data compression. For processing, a compute-farm of about 250 nodes is employed, which can process up to 30 $^{\mathrm{GB}}/_{\mathrm{s}}$ of data coming from the detectors. Hereby, TPC tracking is the most compute-intensive task, especially for lead-lead collisions. A more detailed introduction to ALICE, the TPC, and the HLT, in particular with respect to TPC tracking, can be found in [Roh 10 I, 1.1].

5.3 The ALICE HLT TPC Tracker

Since a comprehension of the algorithm is required and because the terminology is used later, a short description of the tracking algorithm is given here. Detailed introductions can be found in [Gor 12, 2.3/5.3] and [Roh 10 I, 2.3].

The tracking algorithm consists of five steps. The first three are combinatorial and produce an assortment of track candidates, which are called **seeds**. The latter two steps use the Kalman filter [Kal 60] to create the final tracks. Appendix D gives a very brief formulation, how the Kalman filter is used for tracking. More elaborate descriptions can be found in [Gor 12, §1/§2], [Frü$^{+}$ 00], [Man 04], and [Roh 10 I, 2.1]. Fig. 5.4 shows the reconstructed tracks of an exemplary event.

5.3.1 Geometry

As stated in Section 5.1, the TPC is divided in 36 slices. Tracking for each slice is done independently by the so called **slice tracker**. The different track-segments within each slice are then combined to complete tracks by the **track merger**. Slice-tracking is the time-critical task.

Within each slice the local coordinate system is chosen such that in the middle of the slice the x-axis points in the radial direction. The z-axis is oriented parallel to the beam (see Fig. 5.5). The TPC measures coordinates where particles which pass through the chamber ionize gas-molecules. The data are digitized and processed by an FPGA[3] cluster finder. The input for the slice tracker is an assortment of three-dimensional space points, called **clusters** or **hits** hereafter.

The TPC measures the x-coordinate of clusters discretely in 159 possible distinct values, which are called **rows**. The y- and z-coordinates are measured continuously. As particles produced by the collision in the center of the TPC, and in particular the important high momentum particles, can be assumed to pass the TPC slice in x-direction, the ALICE tracker searches for seeds in this direction. It fits a set of track parameters to the clusters in the seed, then follows the extrapolated trajectory using these parameters, collects more clusters positioned close to the trajectory, and improves the fit with them.

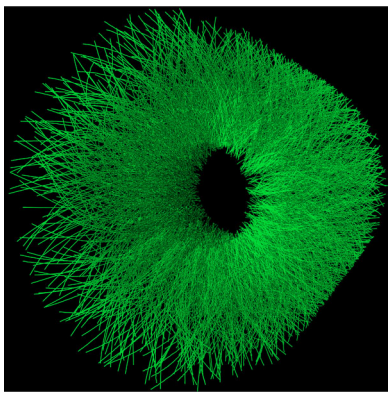


Figure 5.4: Tracks found by the Tracker in a simulated Heavy Ion Event

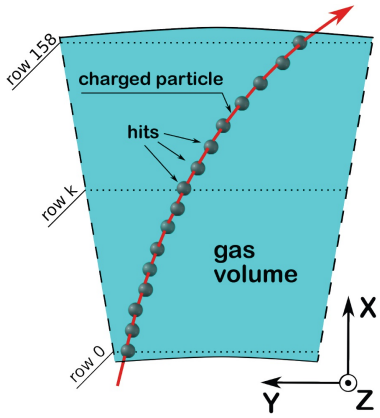


Figure 5.5: Geometry of a Single TPC Slice [Gor 12]

5.3.2 Creating Track Seeds

The seeds are created in three steps. Figures 5.6 to 5.8 illustrate the tasks.

Neighbors Finder [I] In the first step, for each cluster C in each row i, the neighbors finder searches for the pair of clusters in the adjacent rows[4] $(i \pm 2)$, such that these three clusters compose the best straight line. Such a connection is referred to as **link**.

Neighbors Cleaner [II] In the second stage, for each upward[5] link from cluster C to cluster D it is checked whether the downward[5] link of cluster D points back to cluster C. The same is done in the opposite direction. Links not meeting this criterion are removed.

Start-Hits Finder [III] A cluster with an upward but no downward link is a **start-hit**. A set of at least three clusters connected by upward links from a start-hit is called a **seed**.

[3] A **F**ield **P**rogrammable **G**ate **A**rray is an integrated circuit whose behavior can be configured after manufacturing (see Appendix C.6).

[4] The neighbors finder skips one row in order to lessen the relative impact of measurement errors as explained in [Roh 10 I, 2.3.1].

[5] Upward links point to the next row, downward links to the previous row.

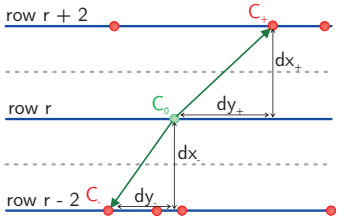


Figure 5.6: Links found by Neighbors Finder for C_0

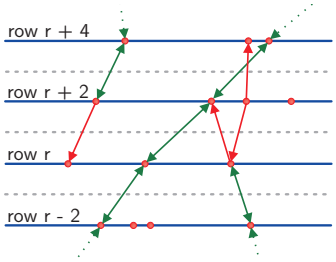


Figure 5.7: Links removed by Neighbors Cleaner[6]

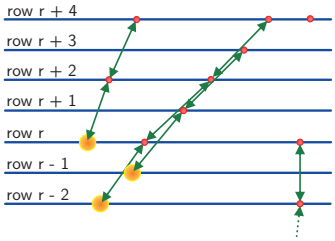


Figure 5.8: Illustration of Start-Hits and Seeds[7]

5.3.3 Fitting Tracks with the Kalman Filter

Tracklet Constructor [IV] Track parameters are fitted to the seeds (stage a). If a χ^2 check is passed, the seed, together with the parameters, forms a **tracklet**. Using the track parameters, the tracklet is extrapolated upward (stage b) and downward (stage c) one after another to all remaining rows. After each extrapolation step, the cluster in the respective row closest to the extrapolated trajectory is determined (Fig. 5.9). The new cluster is incorporated in the tracklet fit. If the χ^2 condition is met, this cluster is finally added to the set of clusters in the tracklet. The Kalman filter [Kal 60] is used for the extrapolation and the fit (Appendix D lists the formulas).

Tracklet Selector [V] This last step performs the final cluster to track assignment. (During tracklet construction, the same cluster can be added to multiple tracklets.) Each cluster is assigned to the longest tracklet it belongs to (see Fig. 5.10). To ensure a unique assignment, out of multiple tracklets of the same length the first one is chosen. For handling multihit-clusters, there is a heuristic to share clusters between tracks [Roh 10 I, 2.3.5].

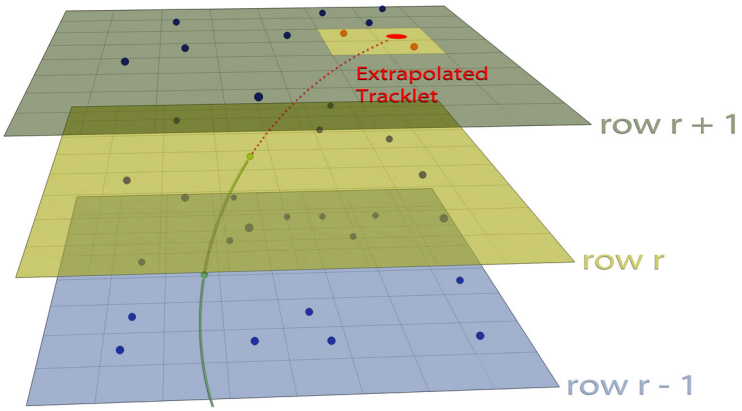


Figure 5.9: Tracklet Constructor Extrapolation Step

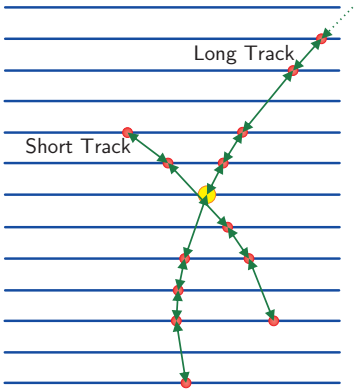


Figure 5.10: Cluster Assignment in Tracklet Selector

5.3.4 Initialization & Output

In addition to these five algorithmic steps, two more are implemented.

- **Initialization:** Creation of special data-structures such as the **grid**[8] for fast cluster search.
- **Track Output:** The data are written to a consecutive memory segment in a new format.

[6] Green links are kept, red ones are removed by the neighbors cleaner. The lower arrow on the left track is removed since by construction there can never be a reciprocal link at the end of the track. (That would need three hits.)
[7] The seeds are green, the start-hits orange.
[8] The grid splits the space in cells for a fast spatial search (see [Roh 10 I, 2.2.3]).

Chapter 6

TPC Slice Tracking on GPU

6.1 The ALICE HLT TPC GPU Tracker

As the TPC-tracking is the most critical task, the HLT TPC tracker has been adapted to run on GPUs as the author's diploma thesis [Roh 10 I]. The first GPU implementation could not outperform the CPU. Many optimizations were required to reach good GPU performance. The CPU could profit from these optimizations as well. In contrast to similar studies [Sch+ 11], the ALICE GPU tracker can stick to single precision, speeding up GPU processing tremendously.[1] Results of the initial GPU tracker have been published in [Gor+ 11]. Fig. 6.1 shows the GPU tracker performance in comparison to the CPU as a synopsis of the diploma thesis. This and the following plots show measurements based on a central heavy ion events with more than 20000 tracks. For enabling a fair comparison, an optimized multi-threaded CPU tracker is used [Roh 10 I, 9.2.1]. It can be seen that the GTX285 outperforms a Nehalem CPU by more than a factor of two while it costs only a fraction. The implementation is such that CPU and GPU version share a common source code. This ensures good maintainability and all advancements introduced later are realized in this way. For this thesis, the tracker has been improved and modified to run on modern Fermi GPUs. The Fermi tracker and its new features have been published in [Roh 12 I] and [Roh+ 12 II].

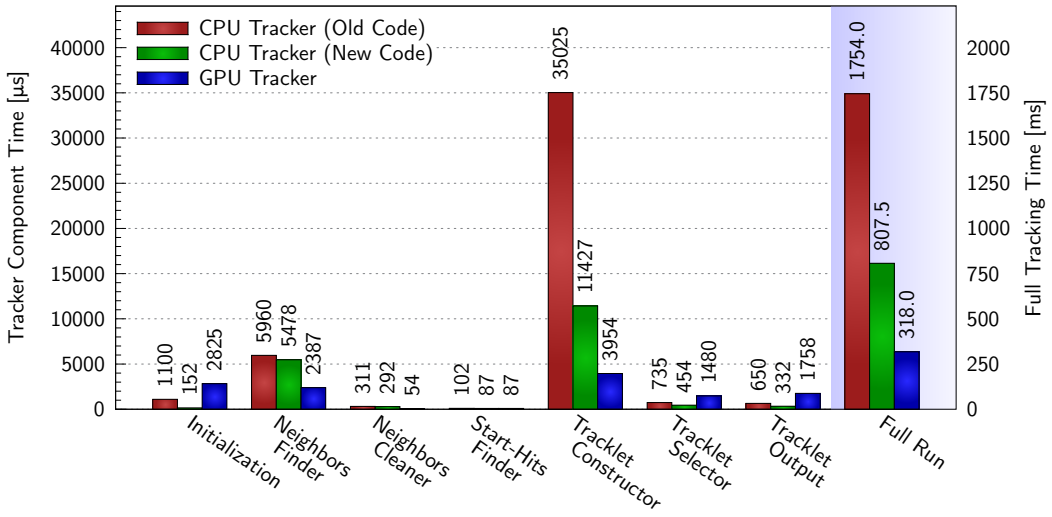


Figure 6.1: GPU Tracker Performance on GTX285 [I] [Roh 10 I, 9.2.2]

One important aspect of the plot is not visible immediately: Component times are measured for a single slice while the time for a full run refers to a full event with all 36 slices. Thus, one would

[1] The Kalman filter was improved by Sergey Gorbunov to ensure numerical stability in single precision [Gor 12, 4.2.3].

expect that the time of the full run is the accumulated component time times 36. However, this is not the case. On the GPU-side, the full run is 30 % faster because of the pipeline [Roh 10 I, 6.3.3], which executes certain steps concurrently. On the CPU side, the full run is 19 % slower because of scheduling overhead and different track counts in different slices. The latter phenomenon depends much on the event size and is discussed in Appendix E.

Originally, it was planned to employ the tracker on the GTX295: the dual-GPU version of the GTX285 based on the same GT200b chip. As the tracker itself can only use a single GPU, multiple tracker instances are needed to use both GPU chips.[2]

6.2 Porting the Tracker to the Fermi Architecture

The GPU tracker was developed primarily for lead-lead collisions.[3] Heavy ion collisions at the LHC started in late 2010. In fall 2010, the HLT was upgraded with GPU nodes for the upcoming lead-lead runs. However, at this point the GTX295 GPU was already discontinued. It would have been possible to still equip the nodes with the GT200b chip but it was not desired to use outdated hardware.

The Fermi chip had been available for some months at that time. Unfortunately, the original tracker code did not immediately work on the new architecture. Up to that point, it was seen as a rather academic attempt to port the tracker to the Fermi. With the decision to employ the Fermi chips, the efforts were concentrated on the Fermi tracker. Due to the limited time, a comprehensive performance tuning for the Fermi chip was obviously not possible. The main goal was to obtain a stable tracker for the run in November. The existent parameters were tuned as far as possible. Major changes to the tracker were not feasible before November 2010 and delayed until after the heavy ion run.

The Fermi port is presented in three steps. This section describes the steps that brought the GPU tracker to the Fermi GPU and the results of parameter tuning. The next one analyzes the performance of the GPU tracker in the November 2010 heavy ion run. The following section presents further improvements implemented after 2010 in preparation for the next heavy ion phase in 2011.

6.2.1 Fermi Support & Compiler Bugs

The original GPU tracker implementation was recompiled for the Fermi architecture. Unfortunately, the old code caused invalid memory accesses on the Fermi GPU (corresponding to segmentation faults on the host). Most of these problems were related to a change in the compiler, which is related to shared memory. Shared memory locations accessed by multiple threads must be declared as *volatile*. In principle, this rule had applied to all prior CUDA versions, too. However, the access violation is only triggered by new optimizations the previous compiler versions did not implement. This is a well-known issue with the Fermi compiler. In fact, the same problem occurs even in many NVIDIA SDK examples.

Another problem was a compiler bug which exported *inline static* host functions to the object file. This led to collisions during linking. Changes in the code structure circumvented this problem.

In addition, some kernel compiler bugs were revealed. With the help of NVIDIA, most bugs could be circumvented by modifying the code: inserting extra variables, changing the order of statements, etc. One crucial bug that could not be bypassed shall be described exemplarily here. Listings 6.2 and 6.3 show test-cases used to track down the bug in CUDA 3.0. The *if* statement accessing a member function may not alter the atomic instruction below in any way.

[2] Multiple instances of the GPU tracker can be used to hide latencies in the framework, too.

[3] The GPU tracker is inferior to the (multi-threaded) CPU version for PP collisions [Roh 10 I, 7.1]. A special version was created for PP, but since the CPU tracker is by far fast enough for PP, it was only an academic topic.

```
__global__ void example_kernel()
{
    example_cls obj;
    __shared__ int tmp;
    atomicAdd( &tmp, 1 );
}
```

Listing 6.2: Working Kernel Example

```
__global__ void example_kernel()
{
    example_cls obj;
    __shared__ int tmp;
    if (obj.example(0) < 0)
        atomicAdd( &tmp, 1 );
}
```

Listing 6.3: Miscompiling Kernel Example

Analyzing the PTX assembler code reveals the compiler bug.[4] Listings 6.4 and 6.5 show the compiled PTX codes for both cases. In the first case, the correct "atom.shared.add.s32" instruction is used, while the second case compiles to "atom.add.s32" accessing global instead of shared memory.

```
mov.u64          %rd1, __cuda_tmp0;
mov.s32          %r1, 1;
atom.shared.add.s32 %r2, [%rd1],%r1;
```

Listing 6.4: Working PTX Code Example

```
mov.s64          %rd1, %rd2;
ld.s16           %r1, [%rd1+0];
mov.u32          %r2, 0;
setp.ge.s32      %p1, %r1, %r2;
@%p1 bra $Lt_0_1026;
cvta.shared.u64 %rd3, __cuda_tmp0;
mov.s32          %r3, 1;
atom.add.s32     %r4, [%rd3], %r3;
$Lt_0_1026:
```

Listing 6.5: Miscompiled PTX Code Example

With this and other bugs fixed by NVIDIA in the CUDA 3.1 release – and in combination with the above mentioned workarounds for other bugs – the tracker accomplishes to run on the Fermi card.

6.2.2 First Comparison

Compared to the reference performance of the original GTX285 tracker (Fig. 6.1), the neighbors finder time decreased from 2387 μs to 1801 μs, the tracklet constructor time from 3954 μs to 3045 μs, and the tracklet selector time from 1480 μs to 1311 μs. In contrast, the initialization time increased from 2825 μs to 3814 μs, the tracklet output time from 1758 μs to 5032 μs, and the total tracking time from 318 ms to 513 s. In total, the CPU tasks took more and the GPU tasks took less time than before. Changes in the data format made the initialization and the output computationally more expensive. It is obvious that for the multi-threaded CPU tracker, the contribution of these pre- and postprocessing steps to the overall computation time is almost irrelevant. However, the GPU tracker uses only one single CPU thread for these tasks. This made it an urgent issue further amplified by the following two facts.

- **CPU Change:** All prior benchmarks were taken on Nehalem CPUs with at least 3 GHz. The cluster's GPU nodes are equipped with 2.1 GHz AMD Magny-Cours CPUs. Even though the AMD CPU has a much higher total performance due to its many cores, its single-threaded performance is far behind the Nehalem's. Unfortunately, this is exactly what is important for the GPU tracker. (Fig. 6.25, in the chapter after the next, shows a tremendous difference in the total tracking time for the same GPU implementations on the two CPU platforms.)

[4] For an introduction to PTX, see [Roh 10 I, Appendix A] or [NVI 12].

- **Pipeline:** The GPU tracker employs a pipeline to run the initialization and the neighbors finder as well as tracklet selection and tracklet output on different slices in parallel. In the Nehalem / GTX285 combination, the compute times for these two tasks are alike (Fig. 6.6[5]). However, the neighbors finder on the Fermi is much faster while the initialization on the Magny-Cours is slower. Therefore, the pipelined initialization is rendered inefficient (Fig. 6.7). In the output phase, the GPU is idling more than 75 % of the time.

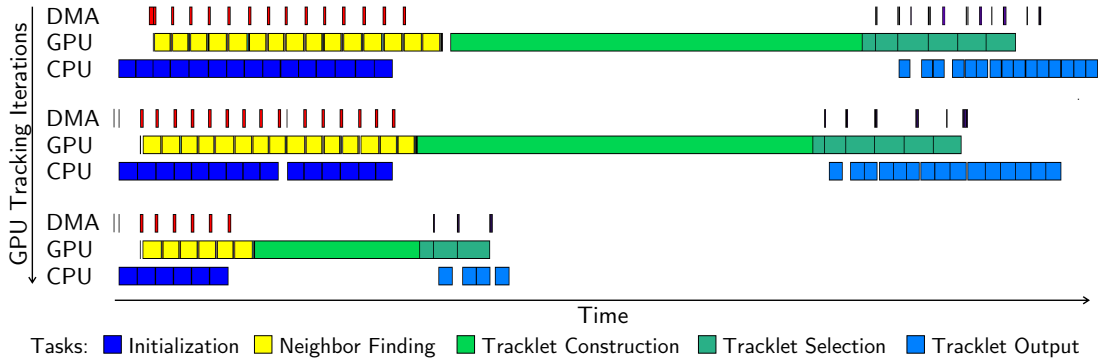


Figure 6.6: Workflow for a Pipeline on the GTX285 with 15 Slices and Asynchronous Transfer *(The x-axis shows the time within one GPU tracker invocation. Multiple tracking runs are shown one below the other. In each run three rows represent DMA, GPU, and CPU steps.)*[5] [I]

Figure 6.7: Pipeline of the first Fermi Tracker Implementation (Compare Fig. 6.6) [II]

The pipeline problems are approached in Section 6.4.3. The solution involves a redesign of the pipeline, which was not possible in the short time frame available. The remainder of this section presents optimizations which could be applied fast and easily prior to the 2010 run.

6.2.3 Tuning Parameters

Several parameters of the GPU tracker can easily be tuned for the Fermi cards. A detailed description of the parameters can be found in [Roh 10 I, §6].

Integer Multiplication One example where the Fermi architecture differs from the previous family is the support for 32-bit integer calculation. The GTX285 offers a fast 24-bit integer multiplication instruction for address calculation but has to emulate 32-bit multiplications. Fermi offers real support for 32-bit multiplication while the 24-bit instruction is dropped and has to be emulated. Therefore, now the standard 32-bit instruction is used.

Texture Fetches The Fermi chip has an L1 cache and does not depend solely on the texture cache. The original GTX285 GPU tracker reads the clusters for the neighbors finder and the tracklet constructor through the texture cache. Fig. 6.8 clearly demonstrates that global memory access with the general-purpose L1 cache of the Fermi is superior to texture fetches for the given purpose.

[5] Fig. 6.6 illustrates a pipelined run for a full event with asynchronous memory transfer and 15 simultaneous slices. As 15 does not divide the slice count of 36, the third and last run contains a reduced number of 6 slices. The NVIDIA CUDA profiler is used to obtain kernel and DMA start times and durations. Each step is assigned a different color. Consecutive kernels for the individual slices are shown separately. The tracklet constructer processes all slices and the tracklet selector some slices in parallel. Therefore, there are less kernel invocations for these routines.

Shared Memory Size The Fermi chip has 64 KB of on-chip memory per multiprocessor. This memory can be configured in two ways:

- 16 KB Shared Memory / 48 KB L1 Cache
- 48 KB Shared Memory / 16 KB L1 Cache

Some GPU tracker optimizations in [Roh 10 I, 6.2] suffered from limited shared memory, e. g. the number of hits the neighbors finder can keep in shared memory is limited [Roh 10 I, 6.2.1]. The cache size can be increased with the Fermi. However, performance is better with 48 KB of L1 cache available (Fig. 6.9). The same holds for shared memory in the tracklet selector.

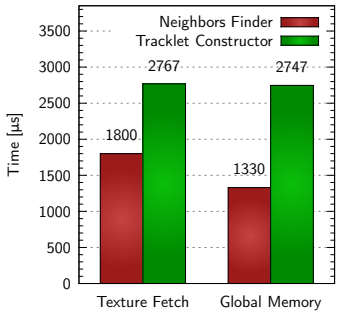


Figure 6.8: Texture Fetches versus Global Memory Loads with L1 Cache [III]

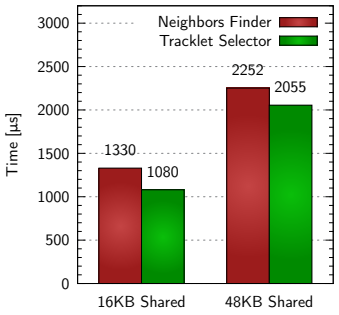


Figure 6.9: Comparison of 16 KB versus 48 KB Shared Memory [III]

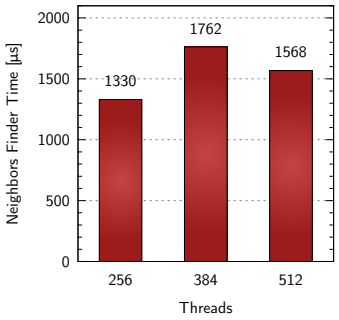


Figure 6.10: Neighbors Finder Performance for multiple Thread Counts [III]

GPU Thread Count & Block Count The optimal execution configuration is determined individually for each kernel by parameter range scan. For instance, Fig. 6.10 shows that 256 threads are optimal for the neighbors finder.

The oldest Fermi cards are called GTX480. They have 15 multiprocessors in comparison to the 30 multiprocessors of the GTX285. On the GTX285 the internal GPU scheduler turned out to work not optimally for the tracklet constructor and tracklet selector kernels. Thus, the original GPU tracker executed these kernels in a grid of 30 blocks (matching 30 multiprocessors on the GTX285) while the scheduling is governed manually by atomic instructions on global memory. For the GTX285, 256 threads per block were optimal.

It seems natural to reduce the block count to 15 for Fermi, which has 15 multiprocessors. However, it turns out that a reduced block count results in decreased performance. A detailed analysis reveals the following cause: as the Fermi has twice the number of registers available per multiprocessor, it can execute multiple blocks on one multiprocessor, and thus more concurrent threads in parallel.[6] (Be aware that from the ALU perspective 256 threads are enough for full GPU utilization on both GPUs. Still, more threads can better hide latencies.)

There are in fact two possibilities to increase the concurrent number of threads: an increase of the block count or an increase of the number of threads per block. It turns out that compared to the version with 15 blocks of 256 threads, both settings improve the performance. The greater benefit is with a larger block count. Increasing both parameters at the same time has a negative effect. It was attempted to decrease the number of threads per block in order to increase the block count even further, but this slows down the tracking.

The question remains why it is better to increase the block count. Naively, one would assume to get optimal performance when the block count equals the number of multiprocessors. However,

[6] In addition to the increased register count, the presence of the L1 cache reduces the urgency to keep all variables in register. The penalty for accessing local variables in the global memory, which do not fit in the register file, has decreased significantly. Therefore, the compiler creates code that requires less registers.

the scheduling integrated in the tracklet constructor kernels always schedules an entire block at a time. More but smaller blocks can be scheduled more fine-grained. Of course, the block count should remain a multiple of the multiprocessor count.

Fig. 6.11 gives an overview. The tracklet constructor achieves its best result with 30 concurrent blocks of 256 threads, while for the tracklet selector 45 blocks turn out to be optimal.

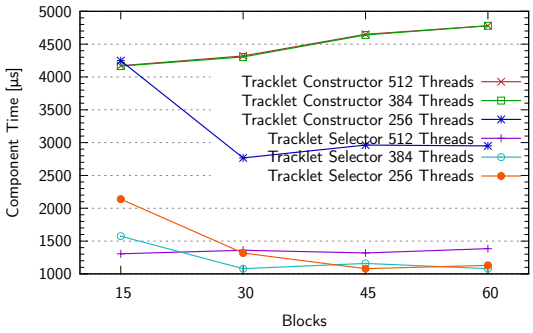


Figure 6.11: Tracklet Constructor / Selector Performance for multiple Numbers of Threads and Blocks [III]

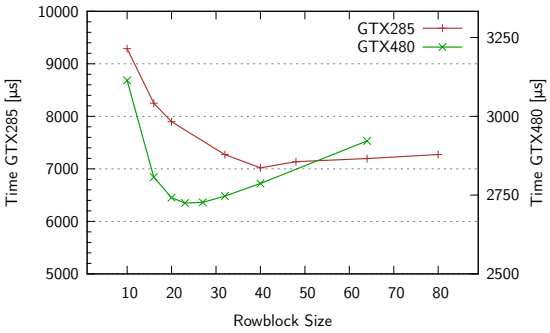


Figure 6.12: Tracklet Constructor Performance for different Rowblock Sizes [III]

Tracklet Constructor Scheduling Parameters The tracklet constructor uses a custom integrated scheduler (see [Roh 10 I, 6.2.2.5]). Informally, the algorithm works the following way:

- The rows are grouped in **rowblocks** of r adjacent rows each.
- A **tracklet pool** contains all tracklets left for processing, grouped by the rowblock to which the tracklet will be extrapolated next.
- A block of n threads processes n tracklets from the pool for one rowblock and does the extrapolation step to all r rows in this rowblock. Each thread processes one tracklet.
- After the processing of one rowblock is finished, tracklets that need to be extrapolated to further rowblocks are stored in the corresponding tracklet pools again whereas tracklets whose processing is completely finished are stored to global memory.
- The scheduler repeats this procedure until all tracklets are fully processed.

Threads that have finished processing their tracklet idle as long as their block is still processing its current rowblock. In this case, tracklet and thread are denoted **inactive**; otherwise **active**. Thus, smaller rowblocks yield a better efficiency but introduce additional scheduling overhead. As each multiprocessor on the GTX480 has 32 ALUs (the GTX285 only has 8), the efficiency is more critical. Fig. 6.12 shows that in fact, the optimum rowblock size for the GTX480 is smaller.

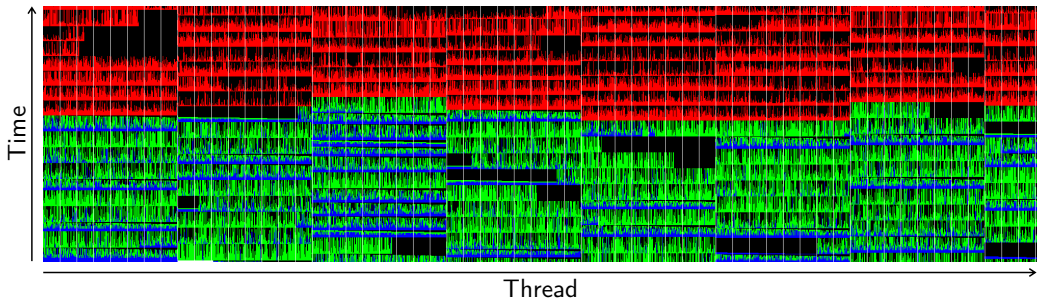


Figure 6.13: GPU Utilization during Tracklet Construction[7]

[7] The colors stand for: black = inactive thread (idling), blue = track fit of the seed [IV (a)], green = forward extrapolation [IV (b)], red = backward extrapolation [IV (c)]. More information can be found in [Roh 10 I, 6.2.2.1].

As the custom scheduler for the tracklet constructor is not altered, the scheduling efficiencies for GTX285 and Fermi should be identical. Fig. 6.13 shows a scheduling plot on the GTX480. The scheduling efficiency is the fraction of ALUs with active threads.[8] The GTX285 reached 62 % efficiency [Roh 10 I, 6.2.2.11] while the Fermi reaches 66 % utilization. The Fermi provides more memory than the GTX285 and thus gives the scheduler more freedom by processing more slices in parallel. This, in combination with the tuned parameters, leads to increased utilization. Running the Fermi with the original parameters and slice count yields 62 % efficiency as for the GTX285.

Results Fig. 6.14 shows the gain by the tuned parameters. Besides the GPU-side improvements, a patch by Sergey Gorbunov made the tracklet output with the new format as fast as with the old one. This update was not available during the first Fermi tests. Since the new version is incompatible with the GTX285, results for the GTX285 obtained with the original tracker version are included. Even though most components are faster or equally fast, the GTX480 is only marginally faster than the GTX285 for two reasons: due to the slow CPU parts in the pipeline, the improvements in neighbors finder and tracklet selector are hidden; the slower initialization compensates the gain in the tracklet construction. Solutions for this are presented later.

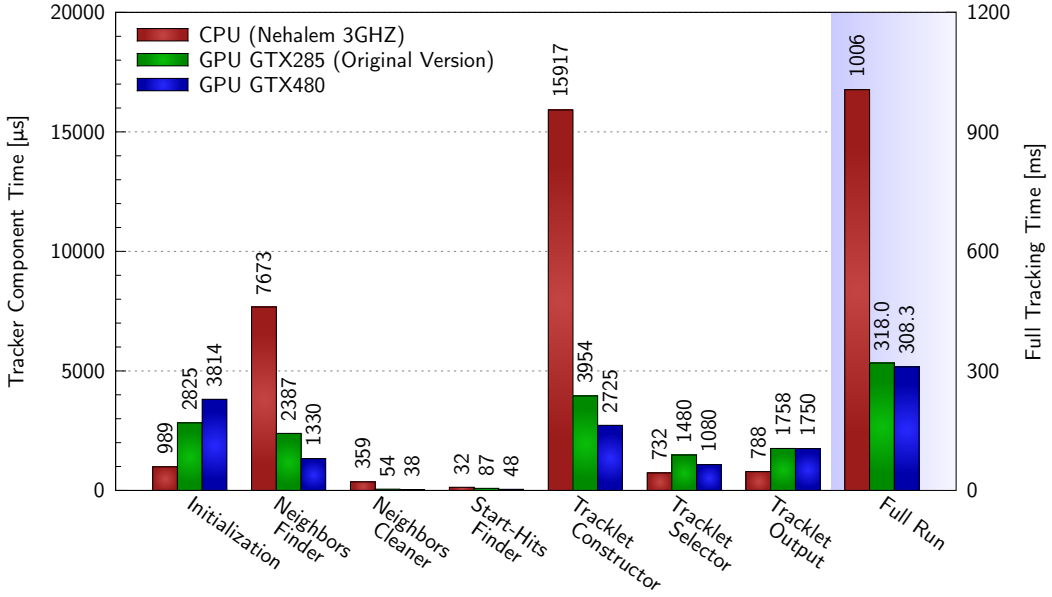


Figure 6.14: Fermi GPU Tracker Performance with tuned Parameters [III]

6.2.4 Integration in the HLT Framework

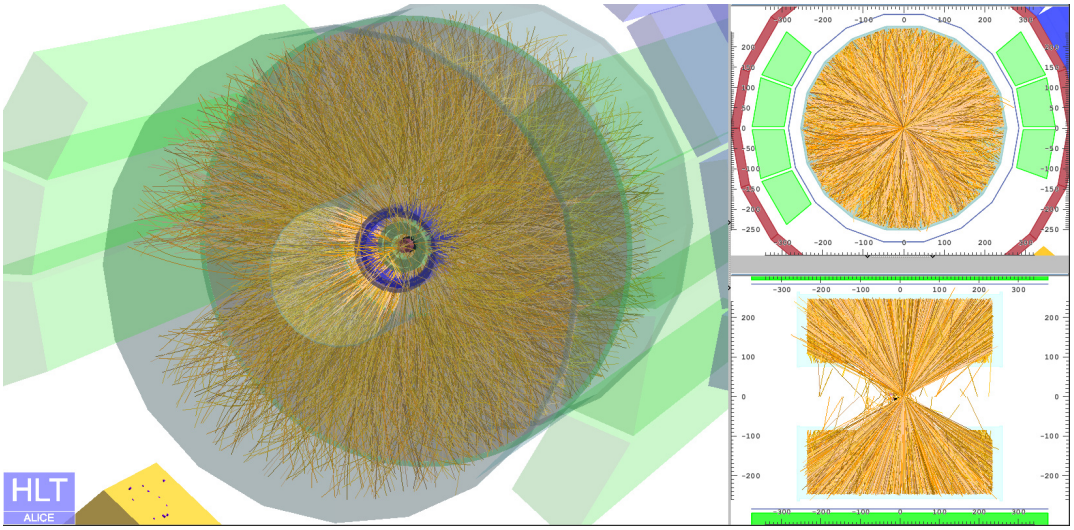
The GPU tracker is available as a standalone version for tests and as a library that cooperates with the HLT-TPC analysis library of AliRoot. AliRoot [ALI I] is the standard event reconstruction, simulation, and analysis framework for ALICE. It is based on ROOT [CER]. The HLT chain is operated by the PubSub framework [Ste 04]. A common tracker interface can run the CPU as well as the GPU tracker in an HLT mode, an AliRoot mode, and a standalone mode.[9] See [Roh 10 I, §8] for details on the integration of the tracker in AliRoot and PubSub.

[8] To be precise, efficiency is the ratio of threads whose tracklet is in phases [IV (a)] to [IV (c)] to the total number of threads. Warps without active threads are excluded in both counts as they are not executed. This number measures the raw scheduling efficiency, i. e. the percentage of threads processing a track. It does not consider threads which are active but are waiting for data from memory, etc.

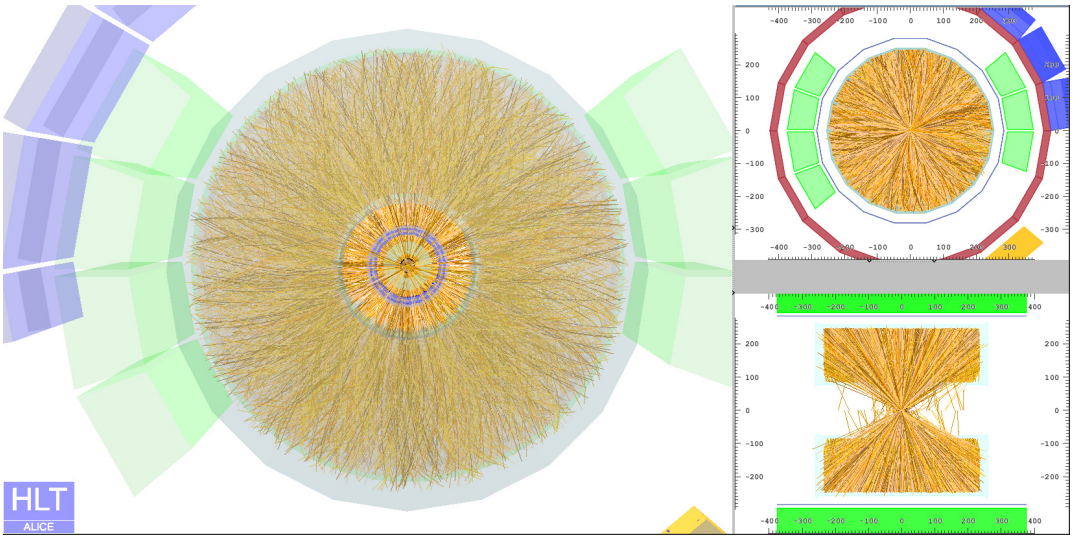
[9] AliRoot and HLT mode are the common tracker operation modes for data analysis and differ only in the interface. The plain standalone mode provides integrated replacements for ROOT functions. This simplifies the debugging without having to deal with AliRoot and ROOT.

6.3 Online Tracking during the November 2010 Heavy Ion Run

In November 2010, a stable GPU implementation for Fermi was available. Due to the low LHC luminosity during the first heavy ion run, even the CPU tracker was fast enough to perform the tracking. Hence, the usage of the GPU tracker was not really vital. Because of high memory consumption of the framework (see below), it was not possible to track many slices in parallel on one node. In order to test the GPU tracker under real conditions with only a minimal risk to disturb the HLT operation, a configuration was created that tracked only four slices on the GPU. The rest was processed by CPU trackers.



(a) Perspective View



(b) Front View

Figure 6.15: Screenshots of Online Event Display during first Run with active GPU Tracking

In December, during the second phase of the 2010 heavy ion run, the GPU tracker was activated. Fig. 6.15 shows two snapshots of the Online Event Display during the first physics run with GPU tracking enabled. The tracker ran stable in the HLT chain. Two problems were discovered and solved during the tests in the online framework:

- **GPU Initialization:** The CUDA runtime API[10] context is thread local. This posed a problem because the HLT operation framework uses different threads for initialization and actual usage of components. Originally, this was solved by skipping the framework initialization phase and initializing the GPU tracker just before the first event is processed. Unlike the GTX285, the Fermi initialization takes more than a second adding a large delay to the first event. In contrast to the CUDA runtime API, the CUDA driver API offers the possibility to migrate contexts between threads. An API change thus solves the problem.
- **Memory Usage:** The buffer sizes of the framework depend on the input data size. Since the GPU tracker processes multiple slices in parallel, its input data size is much bigger than for its CPU counterpart. Initially, due to very conservative settings, the framework allocated many gigabytes and exhausted the system memory.

6.3.1 Evaluation & Quality Assurance for the Tracking Results

Two methods are used to check the functionality of the GPU tracker. Before the first heavy ion run, the GPU results were checked with Monte-Carlo simulations. During the run, CPU and GPU tracking results of real events were compared.

6.3.1.1 Verification of Simulated Data

Monte-Carlo simulations can be used to create particle trajectories and clusters as expected in heavy ion collisions. The simulated clusters are processed by the GPU tracker, and the reobtained tracks are then compared to the original tracks from the simulation. Verification is performed with respect to different criteria:

- **Efficiency:** Percentage of simulated tracks that are found by the tracker.
- **Fake Rate:** Fraction of tracks found by the tracker that do not correspond to tracks from the simulation.
- **Clone Rate:** Fraction of tracks found twice, e. g. two different segments of one track that are not merged.
- **Resolution:** The resolution of the track parameters after the fit.

A comprehensive analysis of the GTX285 GPU tracker based on simulated data is presented in [Roh 10 I, 9.1]. As the GTX285 already matches the CPU tracker in terms of efficiency and resolution, no problems were expected with the Fermi card. Since the tracker and the evaluation algorithms have evolved and the new version is incompatible with old GPUs, no direct comparison can be made between the Fermi and the GTX285. Table 6.16 gives an overview over the tracking quality of the tracker version used in the 2010 run. Fig. 6.17 reveals that CPU and Fermi GPU tracker efficiency are even at any momentum. Detailed and more recent results are presented later.

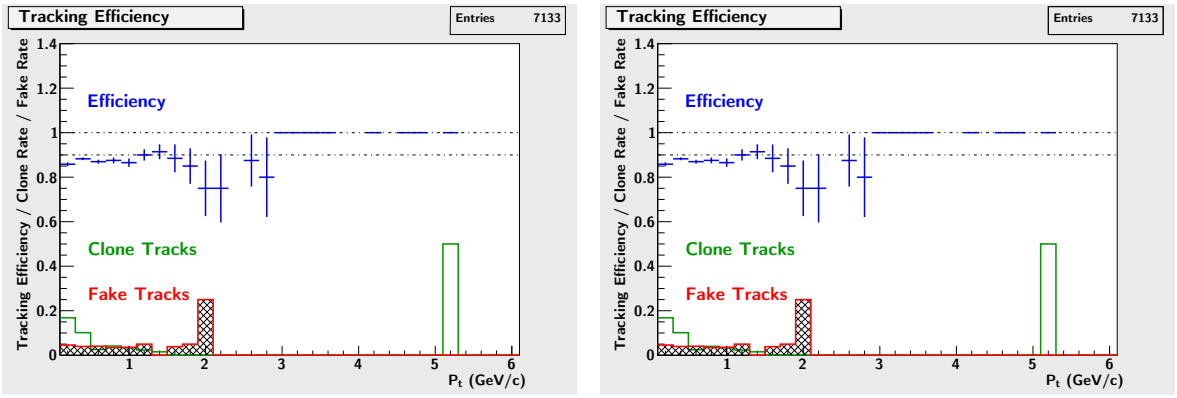
Processor	Efficiency	Clone Rate	Fake Rate	Z-Resolution
CPU	87.33 %	8.852 %	4.055 %	0.1434 mm
GPU	87.33 %	8.817 %	4.057 %	0.1430 mm

Table 6.16: Tracker Efficiency for Pb-Pb-Simulations with maximum Multiplicity

6.3.1.2 Verification of Physics Runs

The tracks reconstructed in physics runs cannot be analyzed so easily since no Monte-Carlo reference exists. In order to ensure that the GPU tracker works correctly, statistics of GPU

[10] **A**pplication **P**rogramming **I**nterface.



(a) CPU (b) GPU (Fermi)

Figure 6.17: Tracker Efficiency, Clone, and Fake Rate

and CPU runs are compared. Figures 6.18 and 6.19 show some QA[11] plots of similar runs in December 2010, created with the GPU and the CPU tracker. The usual cuts are not applied since the raw tracker output shall be compared.

The statistic for the number of clusters per track (Fig. 6.18) shows a slight variation for longer tracks. This is caused by an indeterministic cluster to track assignment, which was improved for the next heavy ion run in 2011 (see Section 6.4.1). Tracking efficiency and resolution are not affected since the tracker finds the same tracks with almost identical parameters. Only in some rather rare cases a cluster close to the intersection point of two tracks can be assigned to the one or the other track. Fig. 6.19 shows that indeed the statistics for physical observables (pseudorapidity, polar angle distribution, and momentum) overlap clearly. No difference between GPU and CPU tracker could be observed in terms of tracking quality yet. Still, the inconsistency makes a direct comparison of GPU and CPU tracks rather complicated. This poses two problems: even though no differences have been found yet, in principle, every new algorithm that runs on top of the tracking result should be verified with both versions. On top of that, full reevaluation of both trackers is needed after every code update.

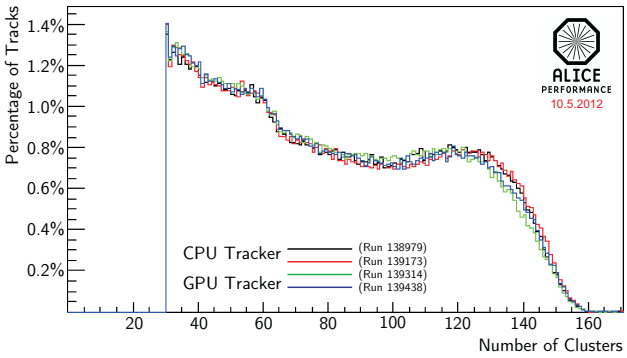


Figure 6.18: Clusters per Track Statistic Comparison of two GPU and CPU Runs[12]

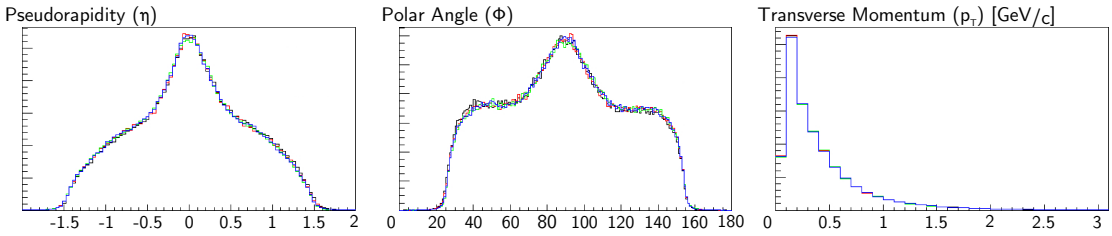


Figure 6.19: Comparison of Physical Key Observables of two GPU and two CPU Runs[12]

[11] **Q**uality **A**ssurance.
[12] The author would like to thank Camilla Stokkevåg for providing the QA plots.

6.4 Further Optimizations & the Heavy Ion Runs in 2011 and 2012

The GPU tracker succeeded in its first test under real conditions in 2010. Since the next lead-lead run was scheduled to start in late 2011, there was plenty of time to better tune the tracker for the Fermi hardware and resolve the issues observed during the 2010 period. This section describes all optimizations in between of the two heavy ion runs.

6.4.1 Improving the Cluster Assignment

The results of the original GPU and CPU trackers could not be compared bitwisely due to an indeterministic cluster to track assignment. The initial rule for the assignment was to assign the clusters to the longest tracklet possible. In case two tracklets were equally long, the first one was used. The problem is that the multi-threaded start-hits finder creates the start-hits in an undefined order. Thus, the order of the track seeds is undefined as well. For this reason, especially for central lead-lead events, the GPU tracker did not deliver completely reproducible results. Even the number of obtained tracks fluctuated slightly because some tracks were rejected or not depending on the cluster assignment. Fig. 6.20 shows that the number of found tracks has a Gaussian distribution. The number of tracks found by the CPU tracker is reproducible and is just one of the possible numbers the GPU tracker can find. In the example, it lies slightly outside the 1σ-bound. Clearly, the deviation is small and has no implication on the physics performance. In particular, tracks which are rejected or not are very close to the χ^2 cut anyway, hence it is not relevant whether they are reconstructed.

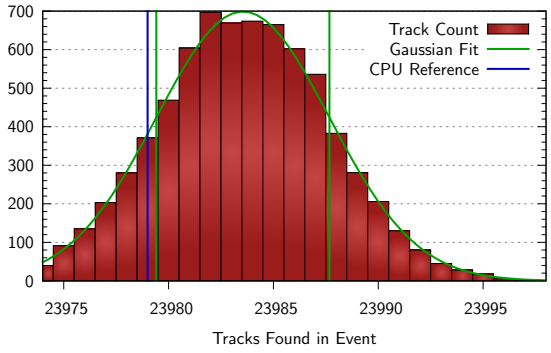


Figure 6.20: Track Output Statistics of the original GPU Tracker in an exemplary Event[12]

A better criterion for the cluster assignment has been searched for. The idea is to replace the physically meaningless and random start-hit order by a more expedient value. A continuous parameter is chosen to avoid the ambiguity that comes along with the few possible discrete track lengths.

6.4.1.1 Incorporating the χ^2 Value

The cluster assignment is performed by giving each tracklet a **hit weight**. Clusters are then assigned to the tracklet with the highest possible weight. Yet, the tracklet length has been used as hit weight. The χ^2 value / residual (deviation between the clusters and the trajectory) is a continuous floating point value and can be used in combination with the tracklet length as a criterion for the cluster assignment. Two χ^2 values exist: the deviation of the tracklet to one cluster (for each cluster / tracklet combination) and the χ^2 of the entire tracklet, which is the average of the deviations to all its clusters. Multiple possibilities exist to use the χ^2 value to determine the hit weight.

- The reciprocal of the χ^2 for the cluster / tracklet combination.
- The reciprocal of the χ^2 of the entire tracklet.
- A combination of one of the above and the tracklet length.

The first possibility sounds self-suggesting. However, the results are merely bad. Usually, at least two tracklets for one track are found. This track is then reconstructed twice resulting in two tracklets with almost identical clusters. It is desired to keep the better tracklet, assign all clusters to this tracklet, and remove the other instance of the track as a clone. However, using the residual

between the cluster and the tracklet, about half of the clusters get assigned to one instance of the track while the rest is considered to belong to the other one. If one of the tracklets is removed and not merged correctly, half of the clusters are lost. In addition to this flaw, implementing the first version is computationally the most expensive option. Thus, it is not used.

The second option suffers from another drawback. It is relatively easy to fit a helix to a small number of clusters. In addition, the fewer the clusters the higher is the probability that no cluster has a high residual. Thus, the χ^2 value for short tracks might be inappropriately small. In particular, one track might be found twice: once as a short tracklet and once as a long tracklet. The shorter tracklet might well have the better χ^2. Thus, this option is ruled out as well.

These considerations leave only the third option (with the χ^2 of the entire tracklet) left. The hit weight w is implemented as $w = n \cdot (\alpha - \chi^2/\beta)$. The χ^2-cut is chosen as the value of β such that χ^2/β is of order 1. This is only for normalization reasons. The factor α is called the χ^2-suppression factor and n is the tracklet length. Thus, $\alpha \gg 1$ results in the old behavior, where only the tracklet length is decisive. In general, the bigger α the lower is the influence of χ^2.

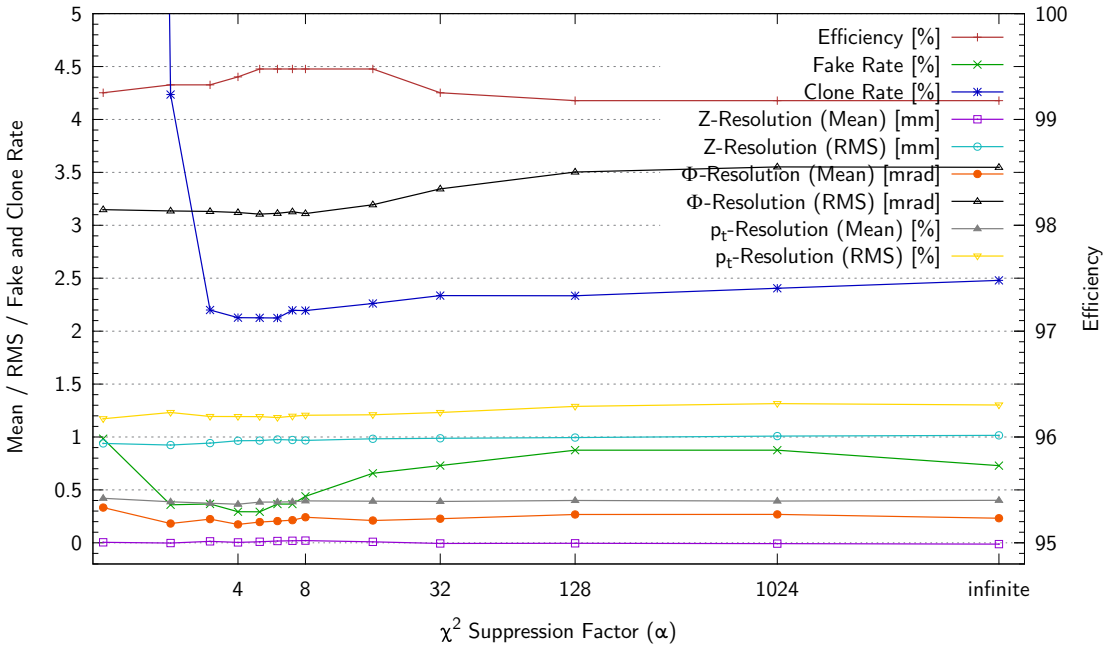


Figure 6.21: Efficiency and Resolution using different χ^2 Suppression Factors

Fig. 6.21 shows the variation of the tracking quality with respect to α. It turns out that incorporating the residual even improves the tracking efficiencies, reduces clone and fake rates, and either improves or maintains the resolutions. Below about $\alpha = 4$ the situation changes and the tracking becomes unstable. (At low α only the χ^2 value is dominant and the problem explained above sets in.) Finally, the tracker employs a value of $\alpha = 6$ as it is considered the best trade-off between efficiency and stability.

6.4.1.2 Track Order

The χ^2 usage negates the concurrency effects for the cluster assignment. Still, the sequential arrangement of the tracks returned by the GPU tracker is arbitrary. The track merger output depends on the order of the tracks in the same way the cluster assignment did. In certain situations where two tracks are equally good, it takes the first one. To obtain reproducible merger results, the slice tracks are sorted before they are passed to the merger.

During the tracklet output, the tracks are copied anyway. The sorting is performed by sorting an array of track indices. The subsequent copy step respects the resulting order. In principle, an arbitrary sorting criterion can be used. In order to avoid ambiguities, a combination of the tracklet length, the Z-, and the Y-coordinate of the track parameters is used. Depending on the emphasis on each of these parameters during the sort sequence, the merger output changes marginally. Using a parameter range scan, the weighting is chosen such that tracking efficiency is optimal. The time required for sorting lies within the measuring inaccuracy.

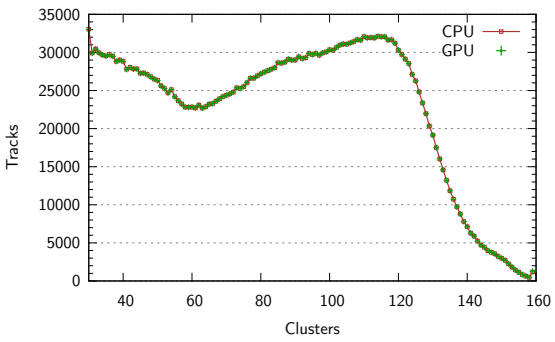


Figure 6.22: Clusters per Track Statistics with improved Cluster Assignment

Fig. 6.22 shows the accumulated clusters per track statistics of the advanced GPU tracker for a constellation of many proton-proton and heavy ion events of different energies and centralities. In contrast to Fig. 6.19, the curves show a perfect match proving that the observed inconsistency has been eliminated. A detailed comparison of the physics performance is presented in Chapter 9.

6.4.1.3 Binary Comparison

The improved algorithm allows for a binary comparison of CPU and GPU tracks. The slice tracker only outputs the tracks in terms of sets of clusters. The final track parameters are completely refitted by the merger. Therefore, as long as both CPU and GPU tracker find the same clusters, the output is bitwisely identical. Comparing the clusters of the sorted tracks row by row reveals that in Pb-Pb-events in average 0.00024 % of the clusters and 0.012 % of the tracks differ.

The difference is caused by different rounding due to non-associative floating point arithmetic, which is a direct consequence of the "-ffast-math" compiler option. The same behavior is observed when switching the compiler or even when selecting different compiler options for the CPU tracker. In three situations the rounding affects the output:

- Two adjacent clusters in a row can have a similar distance to the extrapolated trajectory. Depending on the rounding, either one of the two is chosen.
- It might depend on the rounding whether a cluster is rejected or accepted with respect to the χ^2 cut.
- Both above aspects change the track parameters slightly. This variation carries on in the extrapolated trajectory. Thus, if a CPU and a GPU track differ at all, it is likely that they differ by multiple clusters.

6.4.2 Using the GTX580

In December 2010, the GTX580 GPU was released. All following benchmarks are created with the new GPU. As the chip is identical except for clock frequencies and multiprocessor count, the optimal version for the GTX480 is considered optimal for the GTX580 as well.

6.4.2.1 Variable Block Size

The old GTX285 (and similar cards) have very tight register restrictions, which are especially important in the tracklet constructor. Many constants, such as the number of blocks and threads were thus defined at compile-time. Thus, derived constants were precalculated saving registers. Due to the newly introduced general-purpose cache, this is no longer necessary. Performance does

not suffer measurably when replacing the constants with variables initialized at runtime. The latter improves the usability, especially on development systems where different GPUs are present. So far, multiple binaries had to be maintained for each GPU family. Now, one binary suffices.

6.4.3 Multi-Threading the CPU Parts

Fig. 6.23 shows the pipeline of the new tracker including all previous optimizations. Compared to Fig. 6.7, the faster output improved the pipeline performance significantly. Still, quite a lot of GPU time is wasted during the output. Also during initialization on the CPU, the GPU idles regularly because of the faster neighbors finder.

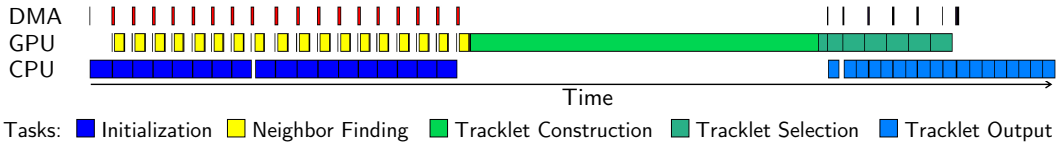


Figure 6.23: Pipeline of the Fermi Tracker with improved Output Routine [II]

Both bottlenecks are approached by multi-threading the CPU parts. A thread-server based on the pthread library has been integrated, which distributes the work. Initialization and output are processed in a round-robin scheme on multiple CPU cores. Fig. 6.24 determines the optimal number of additional threads to be two. Fig. 6.26 shows the pipeline plot with multi-threading. Obviously, the GPU idling phases have been reduced to a minimum. Only during initialization and transfer of the first slice as well as during the postprocessing of the last slices, the GPU idles.

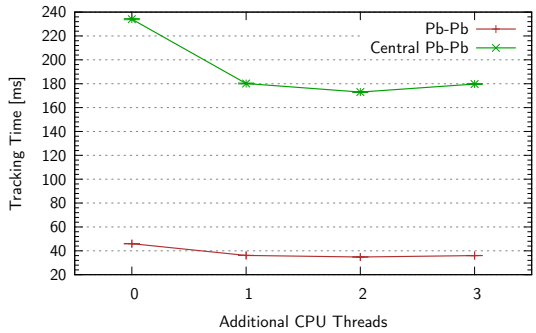


Figure 6.24: Multi-Threaded GPU Tracker Performance [II]

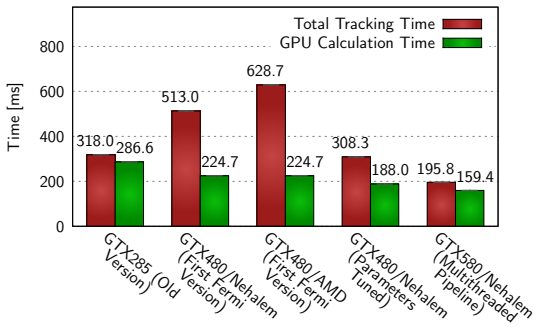


Figure 6.25: Total Tracking Time and GPU Time for all Implementations [I,II,III]

Figure 6.26: Pipeline of the Fermi Tracker with Multi-Threading [II]

Fig. 6.25 shows the total tracking time and the time the GPU actually processes data. The overhead for the Fermi is still slightly larger than for the first GTX285 implementation. However, this is unavoidable for two reasons: increased GPU performance and increased complexity because of multi-threading. Compared to all GTX480 versions without multi-threading, especially the one with the Magny-Cours CPUs, the pipeline is working well again. It will be shown later (in Section 9) that the GPU tracker performance no longer depends on the employed CPU.

6.4.4 Improved Scheduling

For two reasons it was desired to implement a new scheduling algorithm:

- In [Roh 10 I, 6.2.2.13], a different scheduling algorithm was attempted, which competes against the one used for the GTX285. It was discarded because of two bottlenecks, which slow down this alternative algorithm on the old GPU but do no longer exist on the Fermi:
  - The absence of an L1 cache is a disadvantage for algorithms where the threads access data from different rows.
  - Without the L1 cache, respecting the register limitations is extremely critical. Thus, some improvements to the alternative scheme cannot be realized on the GTX285.
- In preparation for the 2011 heavy ion run, it was desired to have the GPU tracker participate in proton-proton runs. First tests with proton-proton data showed two flaws:
  - The dynamic rescheduling after each rowblock is completely unsuited for proton-proton events. It leads to an enormous number of scheduling collisions (see [Roh 10 I, 6.2.2.7]) and slows down the tracking dramatically. The Fermi cards, being the faster cards, suffer from this problem even more than their predecessors.
  - A third type of scheduling collision appears, which was never detected on the old GPU generation. Due to incoherent memory, the tracklet parameters can be corrupted when passed from a thread of one multiprocessor to a different multiprocessor. The possibility of this type of collision was clear from the beginning and is a direct consequence of the design of the scheduler. On the GTX285, the occurrence of this theoretical problem was excluded statistically. On the Fermi, proton events can trigger this flaw.

  The problem for proton-proton can be solved by disabling the dynamic scheduler. However, this eliminates the possibility of using the proton-proton run as a test for heavy ion.

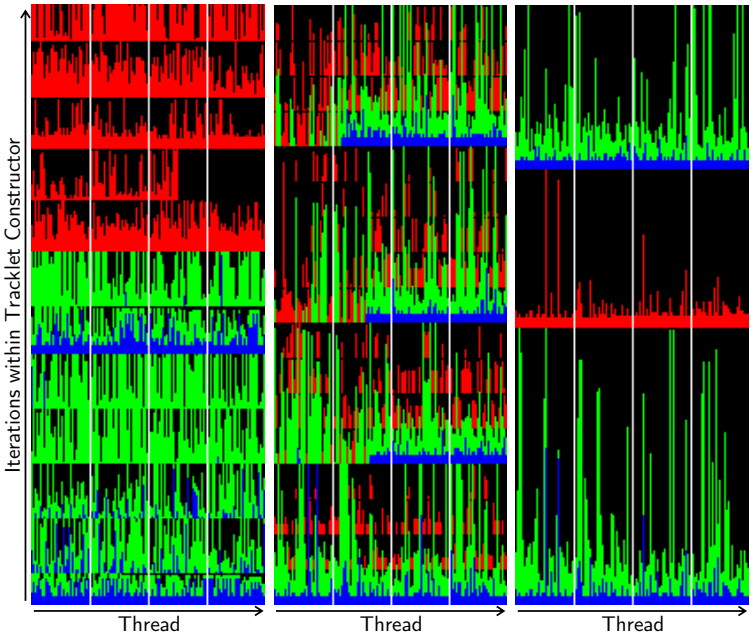
Thus, a scheduler similar to that in [Roh 10 I, 6.2.2.13] is implemented, which does never transfer data between multiprocessors. This way, it completely rules out scheduling collisions. The scheduler implements two parameters $n \geqslant 1$ and $0 \leqslant \alpha \leqslant 1$ and works the following way (from here on out, the dynamic scheduler is denoted old scheduler):

1. Each block picks a slice to start with[13] and each thread is assigned a tracklet within this slice. The initial assignment is predetermined statically to speed up the scheduling. Start rows are arbitrary.
2. The tracklets are processed for n rows or until all of them are completely processed, whichever comes first.
3. The scheduler checks whether the percentage of inactive threads[14] is greater than α. If this is not the case, it resumes with step 2.
4. Finished tracks are stored.
5. Via shared memory the tracklets are redistributed among the threads of the multiprocessor in such a way that the lower numbered threads hold all active tracklets.
   - In contrast to the old scheduler, redistribution within one multiprocessor does not pose coherency problems.
   - This reorganization is technically not necessary but can reduce the number of warp-serializations since the active tracklets are likely to be in phase [IV b)] or [IV c)] while new tracklets begin with phase [IV a)] (see Section 5.3.3).

[13] In order to improve the load balancing, each multiprocessor starts with a different slice if possible.
[14] The number of inactive tracklets divided by the number of total tracklets. For a definition see Section 6.2.3.

6. Organized via shared memory, new tracklets are fetched from the pool of unprocessed tracklets such that afterward all threads process active tracklets again – if present.
7. If any thread has an unfinished tracklet, it is resumed with step 2.
8. The block switches to the next slice and proceeds with step 6 until all slices have been processed.

Setting $n = 1$ and $\alpha = 0$ enforces a rescheduling after each row such that in every moment each thread is active. However, the scheduling overhead is enormous. Setting $n = 160$ and $\alpha = 1$ no rescheduling is performed at all but all tracklets are completely processed at once. The best parameters are determined experimentally. For comparison, a simple scheduler implements the behavior for $n = 160$ and $\alpha = 1$ directly. This makes the above checks (points 3 and 5) obsolete and speeds up the process slightly. The simple scheduler very much resembles the initial scheduler on the GTX285, which achieved only a very low GPU utilization and hence little performance. Naturally, the prefiltering introduced in [Roh 10 I, 6.2.2.9], which removes track candidates with three clusters or less, remains active in all variants as it comes at zero overhead. By construction, the simple scheduler profits most from it.



(a) Old Scheduler (b) New Scheduler (c) Simple Scheduler

Figure 6.27: Comparison of Scheduling Algorithms – Detailed View[15]

Fig. 6.27 shows the GPU utilization of four warps inside one multiprocessor for each scheduling algorithm. The displacement of the active tracklets to the lower numbered threads can be seen clearly in Fig. 6.27b. In contrast to the old scheduler, tracklets in phases [IV (b)] and [IV (c)] are processed simultaneously. The simple scheduler naturally offers the worst utilization.

6.4.4.1 Improved Scheduling Performance

All schedulers have been tested on central lead-lead data. Table 6.28 presents the results. Against all expectations and experience from the GTX285 boards, the simple scheduling algorithm is clearly the fastest on the Fermi. The new scheduler is ahead of the old algorithm but the advantage is almost negligible. Its main improvement is the ensured coherency.

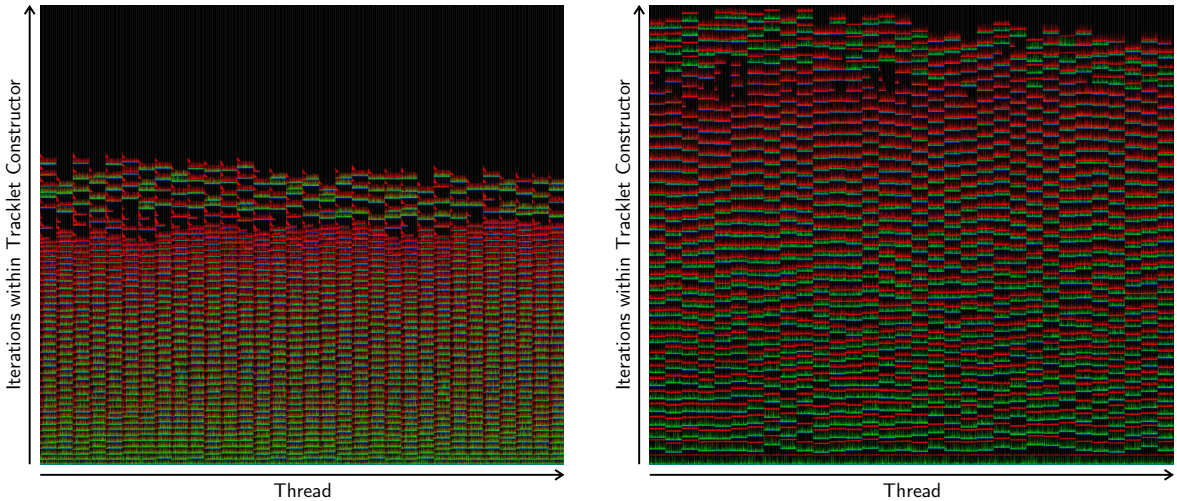
The question arises why the simple scheduler performs so well in comparison to the new scheduler. Fig. 6.29 shows the full plots for the entire tracking of all 36 slices. The y-axis in both plots is scaled linear in terms of iterations of the Kalman filter. It is obvious that the new scheduler requires significantly fewer iterations, so the iterations themselves must be slower. One reason directly visible from the plot lies in the number of blocks (32), which is no divisor of the number of slices (36). After the first 32 slices have been processed, the GPU utilization during the processing

[15] The colors in the scheduling plots have the same meaning as in Fig. 6.13 and are explained there.

Algorithm	Tracking Time [μs]
Old Scheduler	2403
New Scheduler	2376
Simple Scheduler	2148

Table 6.28: Performance of Scheduling Algorithms [II]

of the remaining four slices drops. The efficiency drop is larger for the new scheduler. Another and probably the main cause is the L1 cache. The shared memory based synchronization of the new scheduler leaves only 16 KB of L1 cache while the simple scheduler version can use 48 KB. Since, in contrast to the simple scheduler, the new scheduler extrapolates tracklets in arbitrary rows in both directions, cache size has become even more critical. To estimate the effects, the simple scheduler was modified to allocate as much shared memory as the new scheduler yielding a 62 % longer execution time – much longer than the new scheduler. On top of that, the simple scheduler benefits most from the above-mentioned prefiltering for short seeds. Finally, on the one hand, it is a bit disappointing that the effort put into the scheduling, which achieved good results on the old GPUs, does not yield anything for Fermi GPUs, and it is unsatisfactory to have no proper solution and to miss a complete and detailed insight where the bottlenecks are. Fig. 6.27c shows that the GPU utilization of the simple scheduler definitely leaves room for improvement. On the other hand, the Fermi is still much faster than the GTX285, overall performance increase would probably be in the range of 15 % only, leaving the advanced schedulers out reduces the complexity significantly, two elaborate scheduler implementations already failed on the Fermi, and development of new schedulers is time-consuming without any guarantee that they work well. In the end, with the current code the GPUs in the HLT are already fast enough for real time tracking at maximum TPC readout rate. Hence, in the following, only the simple scheduler is used. Since GTX285 and Fermi GPUs show very different behaviors with different schedulers, a reexamination of the schedulers should be performed when the next GPU generation becomes available.



(a) New Scheduler (b) Simple Scheduler

Figure 6.29: Comparison of Simple and New Scheduling Algorithms –Total Overview[15]

6.4.5 Combined GPU / CPU Tracking

The availability of the GPU tracker makes lots of CPU resources available for other tasks. Initially, these resources were fully used by the merger but lately, optimizations to the merger (see

Section 7.1) have reduced the merger workload dramatically. The free CPU resources can be used for tracking – in addition to the GPU.

This could be realized by using instances of both the GPU and the CPU tracker within the HLT framework. However, the configuration would become very inflexible since the number of GPU and CPU trackers is hardcoded. A transparent approach is wanted that enables one to change the number of CPU cores via a simple software switch.

Thus, the GPU tracker itself has been improved and now offers the possibility to offload a certain number of slices to the CPU. The thread server (see Section 6.4.3) has been extended to maintain CPU tracker threads which are used for this purpose. Assume n CPU threads process m slices each. As long as the single-threaded CPU tracker finishes the tracking of m slices earlier than the GPU finishes its $36 - mn$ slices, a speedup of $\frac{36}{36-mn}$ is possible.

The GTX580 GPU, for instance, has 16 multiprocessors to process 36 slices. Every multiprocessor processes two slices and those that finish first jointly process the remaining four. If synchronization slowed down the tracking of the last four slices, offloading them to the CPU would result in an even larger speedup.

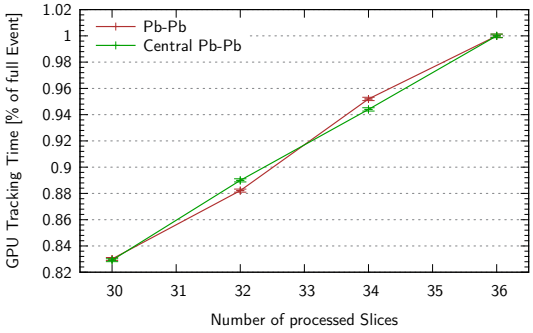


Figure 6.30: Tracking Performance Dependency on Slice Count [II]

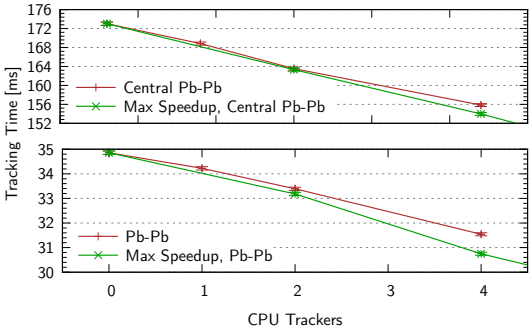


Figure 6.31: Performance of Combined GPU / CPU Tracker [II]

Fig. 6.30 shows that for central heavy ion events, the tracking time depends linearly on the number of slices processed. Therefore, the achieved speedup depends only linearly on the number of slices processed by the CPU. The smaller non-central event shows a small effect at 32 slices (a multiple of the multiprocessor count), where the speedup is more than linear. So the proclaimed positive effect exists but is very small. Finally, Fig. 6.31 compares the achieved results with the **maximum possible speedup**, which is defined as the tracking time the GPU takes for processing its part of $36 - mn$ slices. It reveals that for large events the implementation is almost optimal.

The combined GPU / CPU tracker complicates the QA. Hence, and due to the small benefit, it is not used in the HLT for the time being. If tracking becomes a bottleneck in the future, it can be accelerated with a software switch without having to wait for delivery and deployment of new nodes.

6.4.6 Final Performance Analysis

Fig. 6.32 shows a summary plot of this chapter. The new GPU tracker with all optimizations is about three times faster than the CPU version on the highly clocked hexa-core Westmere. Interestingly, the speedup is of the same order of magnitude as for the previous generation of GPUs and CPUs (GTX285 and Nehalem). The GTX285 result lacks the newer optimizations from this chapter since they have not been backported. Initialization and tracklet output are fastest on the CPU (using twelve threads) and slowest for the GTX285 (single-threaded). The Fermi cards lie in between using three CPU threads, which is sufficient for the pipelined processing. More CPU cores do not speed up the overall process. The performance of joined GPU / CPU tracking is shown for comparison. Four slices are offloaded to four CPU threads in this case.

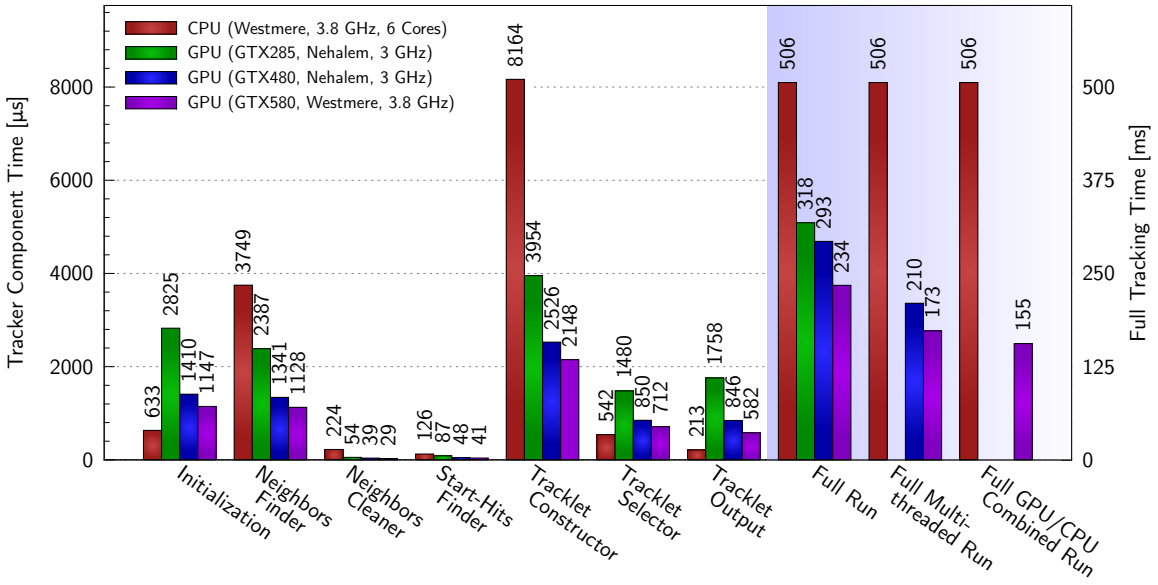


Figure 6.32: GTX580 Tracker Performance with and without Multi-Threading[17] [II,III]

6.4.7 The 2011 Heavy Ion & 2012 Proton-Lead Runs

For the 2011 run, the HLT was upgraded with more GPUs to be able to process central events at a rate of 200 Hz. Due to data compression performed by the HLT, ALICE DAQ[16] was able to store heavy ion events at the full TPC readout rate. Triggering and thus also tracking were not critical for operation. In addition, the transformation component that applies the calibration corrections to the TPC clusters was not fast enough to run at 200 Hz. The GPU tracker processed a subset of the events in real-time. After the TPC had changed the process for trip recovery in the middle of November, it was possible that partially black events were passed to the GPU tracker, where some TPC slices were filled with noise. This can exceed certain buffer sizes of the GPU tracker causing it to skip the event and reinitialize. Depending on which buffer's limit was reached first, a race condition in the synchronization of the GPU tracker threads could trigger a deadlock. This bug has been fixed. Other than that, the tracker ran stable.

The GPU tracker was operated during the entire proton run in 2012. After the cluster-transformation improvements by Sergey Gorbunov enabling higher rates, the GPU tracker accomplished to provide incident-free full online tracking during the proton-lead run at the beginning of 2013.

6.4.8 GPU Tracking on non-CUDA Hardware

From a strategic perspective, it is desirable to use a vendor-independent API instead of CUDA for the tracker. Initially, there was no alternative but in the meantime OpenCL has matured and has been reevaluated. Since OpenCL does not have full C++ support, two strategies have been followed. First, under the supervision of the author, Jens Börger has ported the corresponding code to plain C for his master thesis [Bör 11]. Second, AMD has added C++ kernel language extensions to their OpenCL SDK. Based on this, a wrapper is being developed that uses a common C++ source code in NVIDIA CUDA and AMD OpenCL granting vendor-independence. Following on one of these approaches, an OpenCL tracker is likely to be realized in the future.

[16] **D**ata **A**cquisition - The DAQ stores all data which are not rejected by the HLT to tapes.

[17] For comparison, two twelve-core Magny-Cours CPUs with 2.1 GHz [V] perform the tracking in 495 ms and are thus equally fast as the six-core Westmere [II]. The unequal per-thread performance arises from three facts: more efficient scheduling on the Westmere since the slice count is a multiple of its core count, different frequencies, and different single thread performance in general.

Chapter 7

TPC Track Merging on GPU

7.1 Review of the Situation

After the first well-performing version of the GPU tracker had been implemented, the track merger was left as the most computationally intensive part of event reconstruction. The switch from the single-threaded CPU tracker to the GPU tracker made the slice tracking time almost negligible. Thus, in [Roh 10 I, 10.7] it was concluded mandatory to speed up the merger, possibly using a GPU. However, meanwhile, Sergey Gorbunov has dramatically accelerated the merger by algorithm optimizations. Fig. 7.1 gives an overview.

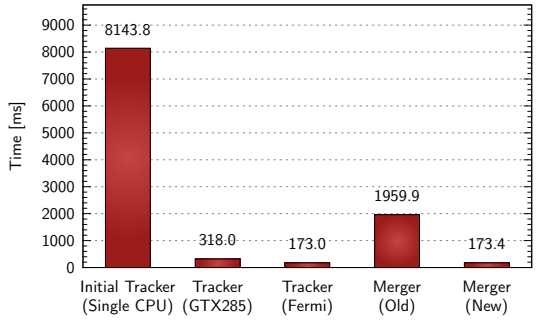


Figure 7.1: Initial Performance of Track Merger and Slice Tracker [I,II][1]

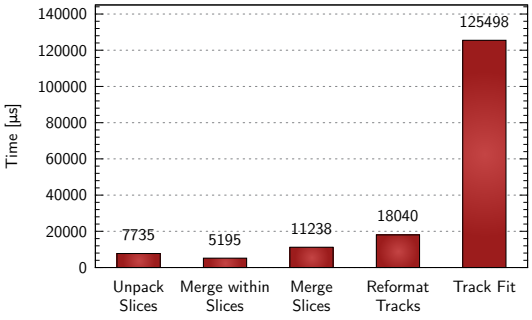


Figure 7.2: Duration of Merger Steps [II]

The new merger and the Fermi tracker require comparable compute time. However, the merger requires only a single CPU core while the tracker occupies the GPU and multiple CPU cores, thus much more resources are spent for tracking. For this reason, less effort has been invested into the GPU merger than into the GPU tracker.

The merger itself involves five steps: Unpacking the input data from the tracker, merging tracks within slices, merging tracks between adjacent slices, reformatting the output, and the final track fit. The final track fit involves the Kalman filter and a polynomial approximation to the magnetic field. To be essentially correct, the fit in the merger uses the same formulas as the slice tracker, which are presented in Appendix D. In contrast to the tracker, the merger does not apply simplifications to speed up the fit. In particular, the slice tracker assumes a constant magnetic field, it assumes that the errors of x- and y-coordinates are uncorrelated, and it ignores the energy loss in the material during the propagation. This makes the fit in the merger much more complex. Fig. 7.2 lists the duration of all merger steps measured with the new single-threaded CPU version.

[1] The benchmarks of the initial CPU tracking and for the GTX285 are taken on [I].

7.2 GPU-based Track Fit

The track fit is the most compute intensive task of the merger and all tracks can be fitted in parallel. It is thus very well suited for parallelization using a GPU. The remaining tasks are partially combinatorial and involve many substeps such as sorting and cannot be ported to GPU easily. In addition, these steps require additional data. If they were offloaded to the GPU, this data would have to be transferred to the GPU as well. Thus, only the track fit is ported to GPU.

Similarly to the tracker (see [Roh 10 I, 4.2]), the code has been modified to allow for a common source code for GPU and CPU. For enabling a fair comparison, the CPU track fit has been multi-threaded with OpenMP [OMP]. The GPU version transfers the data to the GPU, fits the tracks, and transfers the result back to the host. To allow for a fast PCI Express (PCIe) DMA transfer, the memory allocation routine of the merger has been altered such that it uses GPU registered memory. Table 7.3 shows the duration of the GPU merger steps. The DMA transfer uses the PCIe bandwidth (generation 2) almost to the full extent, thus the transfer cannot be accelerated much more. Unfortunately, the transfer time still exceeds the computation time.

Task	DMA	Time [ms]	Speed [GB/s]
Transfer to GPU	No	10.25	3.51
Transfer to GPU	Yes	6.25	5.77
Transfer to Host	No	9.40	3.83
Transfer to Host	Yes	5.61	6.43
Track Fit Kernel	-	6.80	-

Table 7.3: Steps of GPU Track Fit [II]

Fig. 7.4 shows the achieved speedup. Although the GPU kernel itself is faster than the CPU counterpart, the required PCIe transfer makes the GPU track fit fall behind the multi-threaded CPU version. One interesting observation is that the difference between the track fit duration and the total merger duration of the measurements is not equal. The reason is that the other CPU tasks besides the track fit show small performance variations. They are faster if only one CPU thread is used and if the GPU transfer is done via the DMA engine; they are slower with multi-threading and without DMA. The differences arise most probably from OpenMP overhead and cache effects. (A transfer by the DMA engine maintains the CPU cache while a CPU supervised transfer does not.) In the end, this makes the GPU merger slightly faster than the multi-threaded CPU version.

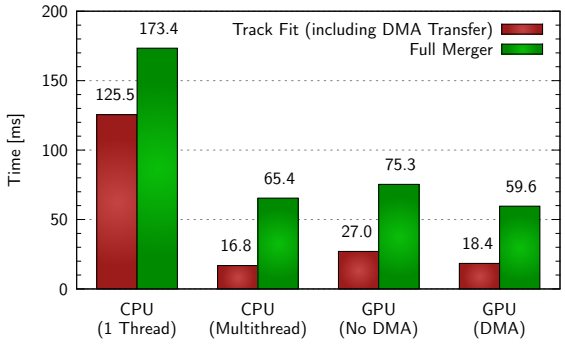


Figure 7.4: Speedup of GPU Merger [II]

Overall, the GPU merger speedup is almost negligible and using GPUs for tracking is much more profitable. For this reason, the GPU merger is not used in the HLT at the moment. At long last, it shall be mentioned that a very fast CPU is used for the test, which is much more expensive than the GPU. Clearly, compared to a slower and cheaper CPU, GPU speedup is accordingly greater.

Chapter 8

Global Tracking across Slice Borders

8.1 Limits of the Slice Tracking Approach

A particle trajectory that crosses the slice boundaries is reconstructed by first finding the track segments within each slice independently (slice tracker) and then merging these segments to the final track (merger). The slice tracker applies a cut for at least thirty clusters per track segment. Thus, a track with an extension of less than thirty clusters into an adjacent slice cannot be reconstructed completely. This flaw is coped with by a new feature called **global tracking**. The track parameters of slice tracks which end in the inner or outer part of the TPC close to the slice boundary are rotated into the local coordinate system of the neighboring slice and used as seeds for an additional iteration of the tracklet constructor. The additional track segments obtained this way are called global tracks. Fig. 8.1 illustrates the situation.

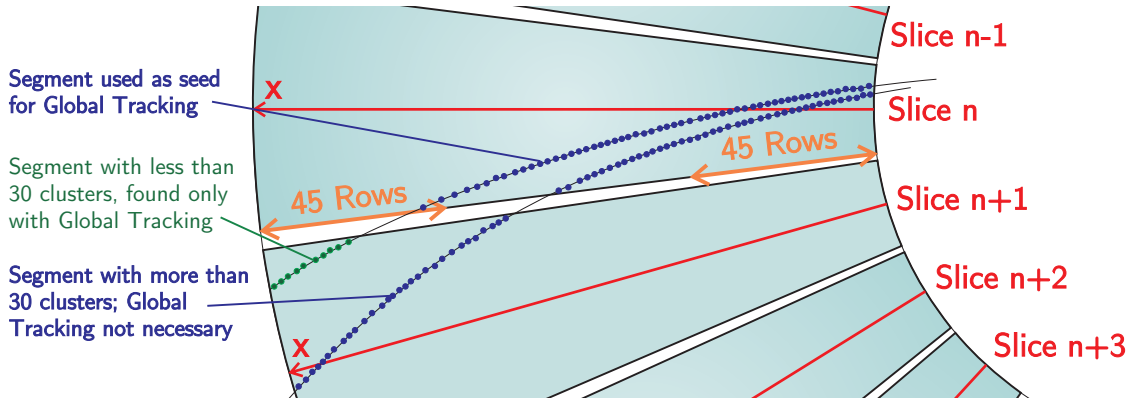


Figure 8.1: Illustration of Global Tracking Principle

8.2 Implementation

The search for global tracks is performed in the 45 highest and the 45 lowest rows of the TPC. The area is larger than thirty rows since a track does usually not possess a cluster in each and every row. Thus, the short track segments which are not found in the first place can range over more than thirty rows. This poses the problem that a track segment which has been found in the initial slice-local tracking phase can be found again as global track segment. There are two ways to handle this:

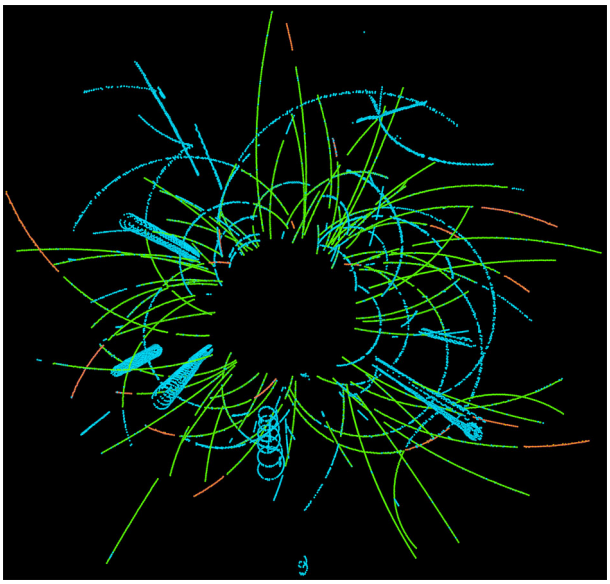


Figure 8.2: Global Track Segments (orange) found in Proton-Proton Event [II]

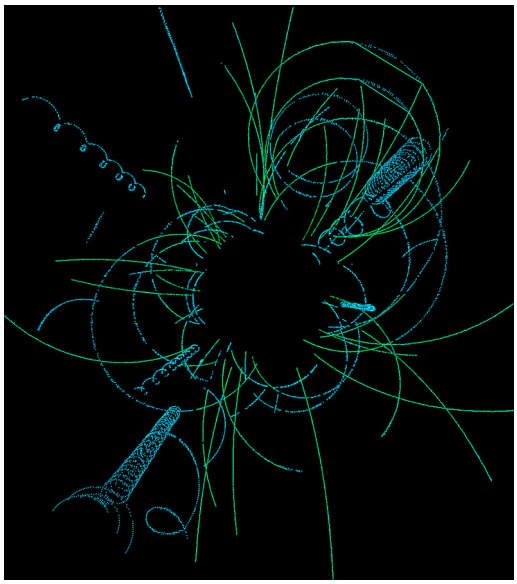


Figure 8.3: Reconstruction of full Helix (Upper right Image Section) [II]

- The clusters assigned to initial slice tracks can be ignored during the global tracking. This is technically complicated and lengthens the execution time.
- Since the tracks are merged in the track merger afterward anyway, a subsequent filter can remove duplicate clusters, i. e. it checks the IDs of all clusters assigned to a final merged track and it removes one instance of the cluster if the ID is present twice.

Measurements show that only 0.1 % of the clusters of global track segments already belong to a slice track from the first phase rendering the first variant immoderate. The second variant is faster and is hence applied in the HLT. It has the additional benefit that it also filters for identical clusters within the original slice tracks.[1] Global tracking is available for both the CPU and the GPU tracker. The GPU version performs the second tracklet construction phase on the CPU since the tracklet count is much smaller than during the first phase. This additional task is inserted into the pipeline in between of tracklet selection and tracklet output. The synchronization must respect various boundary conditions, e. g. global tracking for a slice can begin only after the tracklet selection for the neighboring slices have finished (since both write to the same data structure). Fig. 8.2 shows the slice track segments (light green) and the additional global track segments (orange) in a proton-proton event. (Unassigned clusters are shown in blue.)

The original combination of slice tracker and merger is unable to reconstruct a full helix. Due to a cut on $\sin(\varphi)$, with φ being the angle between the slice-local radial x-coordinate and the particle momentum vector, the slice tracker cannot extrapolate tangential to the primary vertex and thus cannot follow the entire helix. The merger only compares track segments of neighboring slices. Due to a momentum cut, the maximum curvature of the helix is limited and without the global track segments the range where the trajectory cannot be followed ranges over at least one full slice. Thus, the track segments of the helix never reside in adjacent slices and cannot be merged.

Fig. 8.3 shows an example and reveals that the track segments are merged if global tracking is used. The plot does not visualize the fitted trajectory but simply connects the clusters with lines. The long straight part omits exactly the clusters where the extrapolation does not work due to the cut on $\sin(\varphi)$. Unfortunately, the ALICE offline framework is incapable of handling or even

[1] The cluster to track assignment of the slice tracker is not completely unique but in certain situations a cluster is shared between multiple tracks – in a deterministic way [Roh 10 I, 2.3.5]. In rare cases, a cluster can be shared between two tracklets which are merged to one track containing the cluster twice.

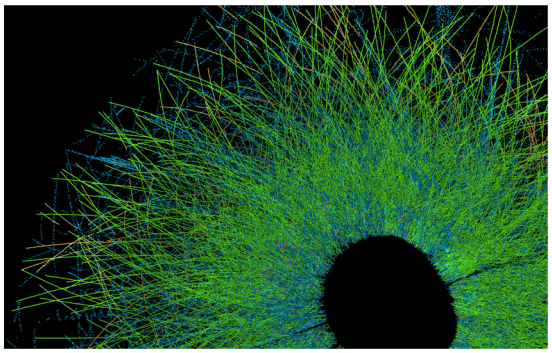


Figure 8.4: Global Track Segments (orange) found in Heavy Ion Event [II]

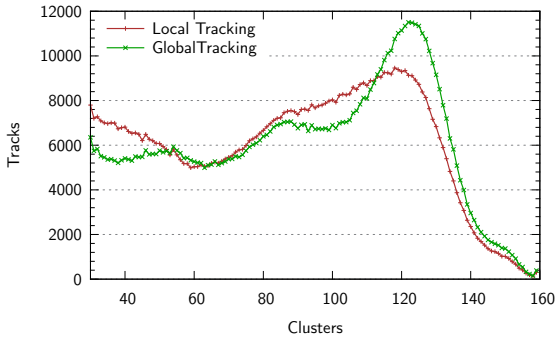


Figure 8.5: Cluster per Track Statistics for Global Tracking [II]

storing track segments where the sign of $\cos(\varphi)$ changes. Thus, such tracks are currently split artificially in two parts with positive and negative $\cos(\varphi)$.

8.3 Results

Fig. 8.4 visualizes global tracking in a heavy ion event while Fig. 8.5 presents the clusters per track statistics with and without global tracking. It is obvious that with global tracking by trend more clusters per track are found.

Global tracking only prolongs tracks but does not find new ones. Hence, it is unlikely that the efficiency changes. The resolution, however, should improve since more clusters are available for the fit. Fig. 9.1[2] shows that these assumptions are correct. It further proves that CPU and GPU results match perfectly. (Curves in the efficiency plot for CPU, GPU, and global tracking lie exactly on top of each other. So do curves for CPU and GPU in the resolution plots.) Global tracking influence on the resolution is rather small because in the end not so many tracks are affected.

Finally, Table 8.6 shows that the global tracking feature increases the tracking time only slightly. Due to its much shorter overall execution time, the relative increase is higher for the GPU tracker. The time increase of the GPU tracker can be reduced to only 4.6 % by using one additional CPU thread for the global tracking part. The very low overhead of the CPU global tracker is due to its single thread. An exemplary multi-threaded global CPU tracker results in a tracking time increase of 12.6 % compared to the multi-threaded slice tracker. Obviously, the number of threads which must be synchronized plays an important role (three for the GPU version, twelve for the multi-threaded CPU version).

Component	Slice-Only Tracking	Global Tracking	Time Increase
CPU Tracker	553.3 ms	565.4 ms	2.19 %
GPU Tracker	35.8 ms	39.0 ms	9.03 %
Track Merger	30.3 ms	31.9 ms	5.23 %

Table 8.6: Global Tracking Time [II]

[2] Figures 9.1 and 9.4 use the following notation: z is the beam axis, x and y are the radial and tangential axes respectively, Φ is the polar angle ($\tan\Phi = p_y/p_x$), p_t the transversal momentum (the projection orthogonal to the beam), $\tan\lambda = p_z/p_\mathrm{t}$, $\eta = 1/2 \ln\left(\frac{|\mathbf{p}|+p_z}{|\mathbf{p}|-p_z}\right)$ the pseudorapidity, and the index "mc" denotes the Monte-Carlo truth. Primary particles are those produced in the primary vertex. This must be understood geometrically not physically, i. e. decay products of short-lived particles like J/Ψ are considered primary particles as well. Secondary particles originate from secondary vertices accordingly. A track is declared findable if a check defined by the ALICE offline software is met, which selects relevant tracks and discards tracks which are unlikely to be found – such as very short tracks. Applied cuts where not stated differently are: $p_\mathrm{t} \geqslant 200\,\mathrm{MeV}$, $|\eta| \leqslant 0.9$, and at least 30 clusters per track.

Chapter 9

Comparison to Offline & Conclusions

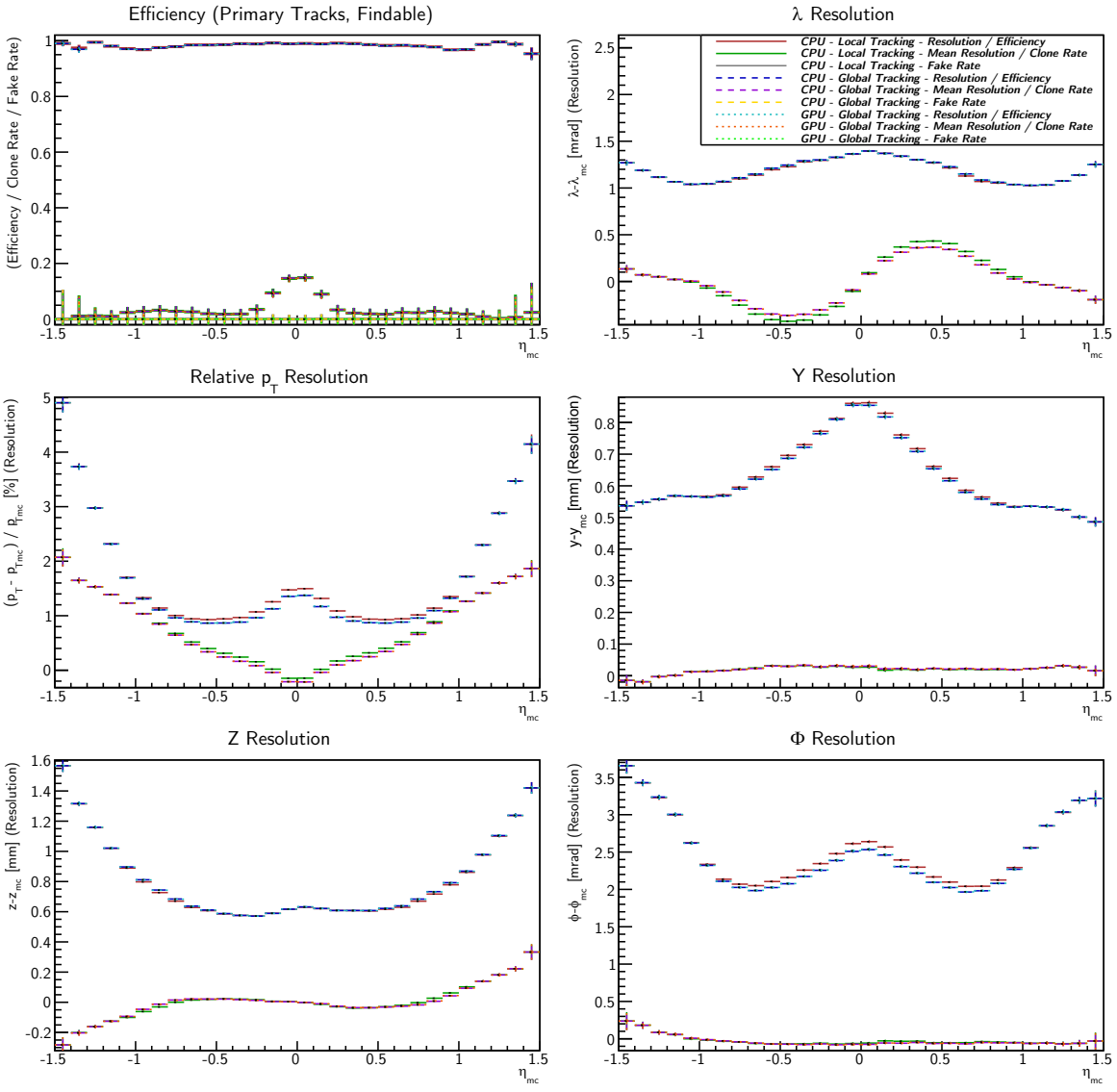


Figure 9.1: Efficiency and Resolution of GPU and CPU Tracker with and without Global Tracking visualized versus Pseudorapidity[2]

Online tracking for the ALICE TPC is split in two tasks: the slice tracker, which searches track segments in all slices independently, and the track merger, which merges associated track segments. GPU-enabled versions for both tasks have been implemented, which are in no way inferior to their CPU counterparts. Fig. 9.1 shows a perfect match in terms of resolution and tracking efficiency. Since GPU and CPU versions share a common source code, only little additional maintenance effort is needed. The tracking time depends linearly on the input data size (see Fig. 9.2), the GPU tracker having a small offset for initialization etc.

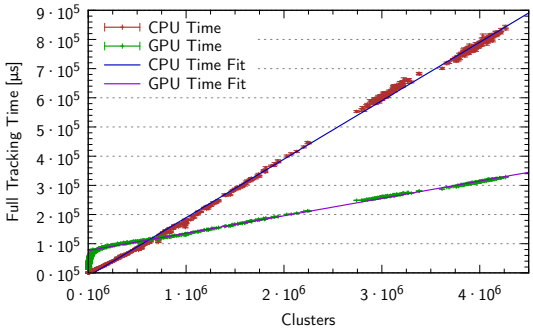


Figure 9.2: Tracking Dime Dependency on Input Data Size [I]

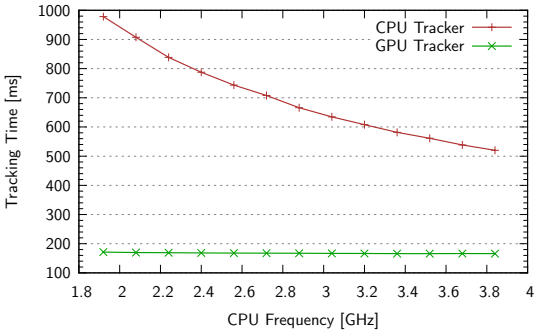


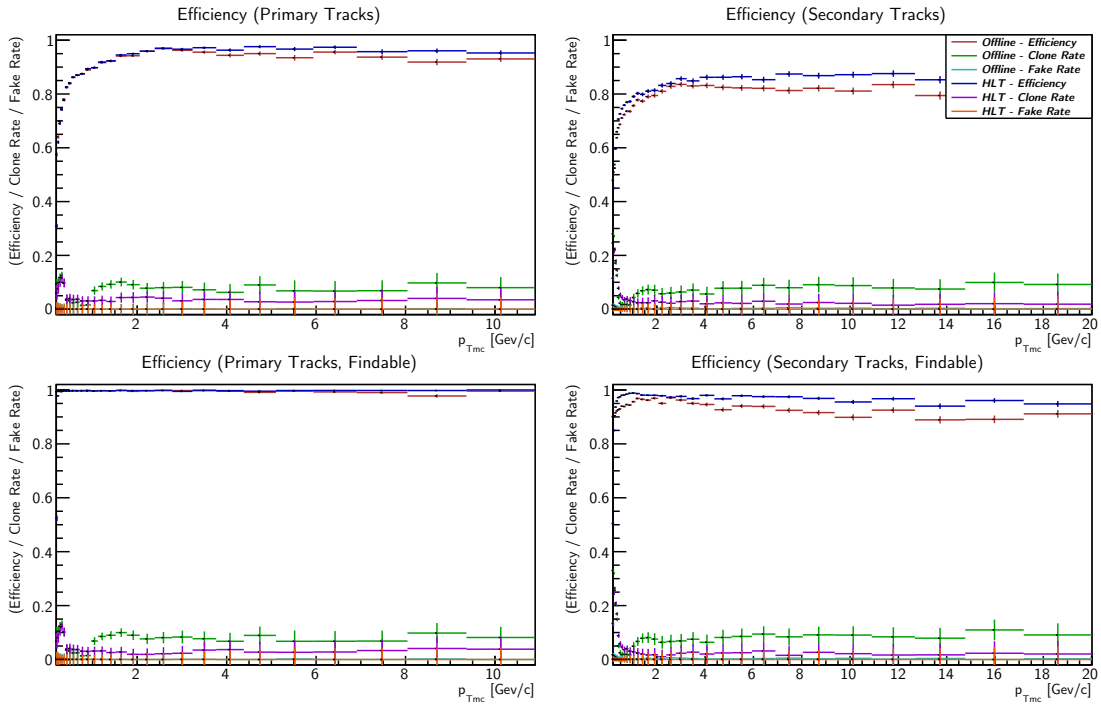
Figure 9.3: Performance of the GPU Tracker depending on the CPU Frequency [II]

The track fit uses about 75 % of the total merger time and is the only relevant part for GPU adoption. It is shown that the necessary data transfer for the GPU track fit already takes 71 % of the time the CPU takes for the whole track fit (in the multi-threaded version). Hence, irrespective of the GPU track fit performance, the GPU merger as a whole cannot outperform the CPU version by much. For these reasons, the GPU merger is neither utilized nor developed further. Instead, the GPUs are used for slice tracking where they deliver a significant benefit.

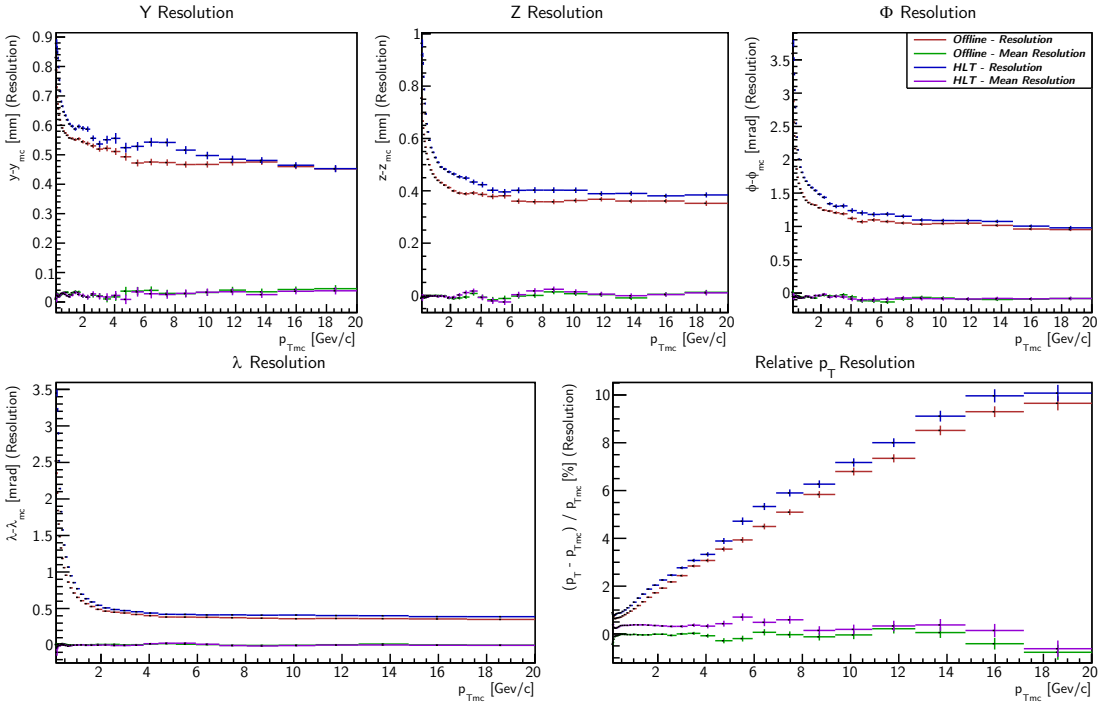
The original GPU slice tracker [Roh 10 I], which had been developed for the GTX280 series, has been ported to the newer Fermi generation and many improvements have been applied. Optimizations to the internals of the tracker improve the performance on the Fermi. In order for the initialization task on the CPU to keep in step, the pipeline has been multi-threaded. Consideration of the χ^2 value together with fast sorting ensures a deterministic cluster assignment and thus consistency between CPU and GPU tracker. On top of that, it improves both efficiency and resolution. Finally, the global tracking feature collects missing track segments, which the antecedent slice tracking cannot find by construction.

Overall, the GPU tracker speedup is about factor three compared to a high-end CPU, quite exactly the same factor as measured with the old version on previous hardware [Roh 10 I, 9.2]. Taking into account the hardware prices, the cost benefit of the GPU tracker is even higher. Fig. 9.3 shows that with the improved pipeline the GPU tracking time does no longer depend on the CPU performance at all. As long as the CPU can feed the pipeline, it is irrelevant which CPU is employed. With most of the compute performance in the GPU anyway, cheap CPUs can be used.

A final interesting analysis compares the HLT tracker to the offline tracker, which is eventually used for data analysis. Consider that the HLT tracker is optimized for fast processing in order to enable real-time analysis while the offline version is tuned for optimal fit results. Ramifications on the execution time are discussed later. Fig. 9.4 compares resolution and efficiency of both tracker implementations broken down in p_t intervals. Subject of the analysis are final global TPC tracks, i. e. HLT slice tracker and merger are compared to TPC offline tracker without ITS (**I**nner **T**racking **S**ystem) fit. In order to facilitate this comparison, the official offline QA scripts were modified and complemented with the possibilities to use TPC-only tracks and to exclude ITS data. In addition, HLT tracks are stored with slice-local coordinates and must be rotated accordingly.



(a) Efficiency



(b) Resolution

Figure 9.4: Comparison of HLT and Offline Tracker[1]

Obviously, efficiencies are equal or the HLT efficiency is even better while offline resolutions are superior under all perspectives. However, it is clear from the design that the offline tracker yields better resolution result. Excellent efficiency and only a moderate difference in the resolution qual-

[1] See footnote 2 on Page 48 for a description of the parameters.

ifies the HLT tracker for a fast data analysis. Selected events can be reexamined with the offline version later. The comparison to offline serves as a tool to investigate possible improvements. There are currently three apparent points with an unfavorable effect on HLT tracking:

- Due to the non-linearity in the propagation formula, the Kalman filter must be iterated such that the result converges to the optimal estimator (see Appendix D). The offline tracker performs three repetitions; the HLT tracker skips the repetitions for performance reasons. This costs between one and two percent of the resolution.

- Previously, the offline QA macro used to generate Fig. 9.4 has not been compatible with HLT tracks. The HLT tracker was tuned based on a similar script, which applies a different set of cuts. Hence, there is a chance that tuning the HLT tracker with respect to the new cuts can improve the results in the future.

- Fig. 9.1 shows a systematic error in the HLT p_t resolution that depends on η. There are indications that this is because the HLT tracker for performance reasons does currently not respect the y- and z-components of the magnetic field during the propagation. At the moment, there is an ongoing investigation how to implement a proper extrapolation without slowing down the reconstruction.

Fig. 9.5 relates the number of clusters found by HLT and by offline for identical tracks. Identification is performed by comparing the assigned Monte-Carlo labels. Most data points are close to the green line which indicates equality. By trend, HLT finds even more clusters. However, this is actually no deficiency of the offline tracker because in certain situations the offline implementation discards clusters on purpose when this improves the final track resolution.

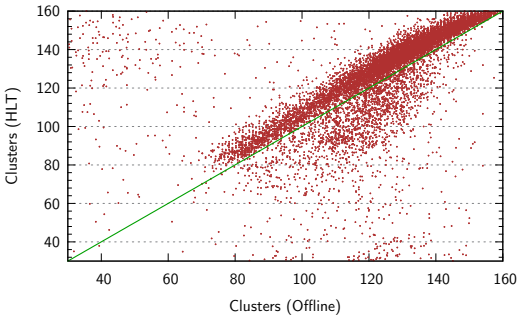


Figure 9.5: Clusters per Track Comparison

The HLT tracker is wrapped by a **component** of the HLT framework which collects and unpacks incoming data for all slices. Previously, sorting the incoming data fragments by slice number was implemented inefficiently. This did not play a role for the single-threaded CPU tracker but had to be completely rewritten to reduce the delay to a minimum for the GPU tracker. The improved version is about ten times faster. Fig. 9.6 shows the remaining overhead. The merger component requires quite exactly the same amount of time as the tracker component. Fig. 9.7 demonstrates an improvement of an order of magnitude compared to the CPU tracker (single-threaded) and of multiple orders of magnitude compared to the offline tracker.

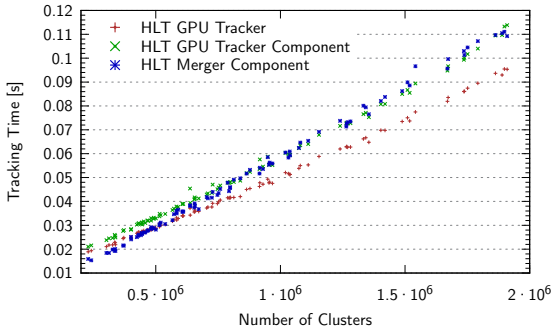


Figure 9.6: Processing Time of HLT Components [II]

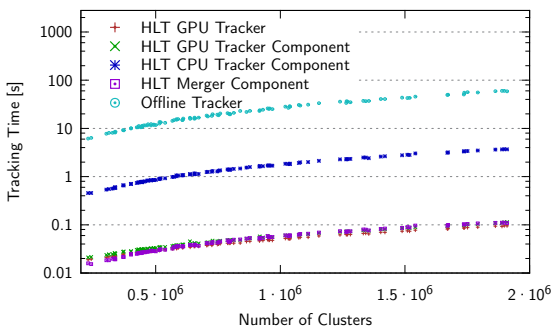


Figure 9.7: HLT & Offline Processing Time [II]

The HLT compute nodes are not used exclusively for tracking but the processors are also used for other tasks such as cluster transformation and vertexing. Therefore, the final speedup in the HLT is analyzed in the following way. The GPU tracking component uses three CPU threads, one merger component needs one more CPU core. Since the HLT automatically pipelines the events and since GPU tracker and merger times are alike, GPU tracking and track merging together occupy about four CPU cores.

In the speedup shown in Fig. 9.8, the time for GPU tracking is thus taken as the maximum of GPU tracker time and merger time times four – for the four CPU cores. The speedup is then calculated as the ratio compared to offline and CPU tracking time. Overall, the plot shows the speedup of the HLT CPU and GPU tracker if four CPU cores are used for tracking leaving all other cores available for other tasks. The HLT tracker is already fifteen times faster than the offline version, and the GPU tracker outperforms the CPU version by another factor of ten.

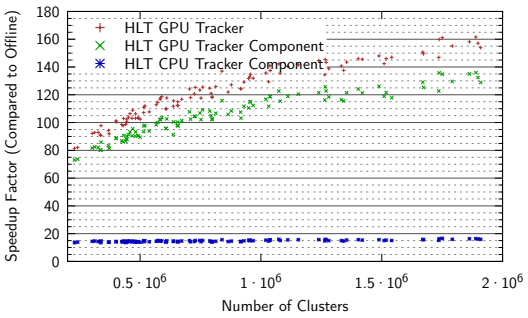


Figure 9.8: Speedup of HLT [II]

In order to get an overview of the cost benefit, consider that a single GPU performs the tracking of a whole event about as fast as a full compute node with two Magny-Cours processors and 24 cores. Since the GPU tracker occupies only three cores, it is no unreasonable statement that the CPU is still almost fully available during GPU tracking. CPU resources are required for various tasks anyway. Hence, plugging a GPU in a compute node essentially saves the costs for one full additional node and for the extra infrastructure required for more nodes. With GPU costs being a fraction of the entire HLT facility costs, the GPU tracker allows TPC online tracking with almost negligible investment in extra hardware. Assuming $ 5000 for a 24-core compute node and $ 300 for a GPU, considering the 64 GPUs currently deployed in the HLT, and neglecting the infrastructure, the net saving is about $ 300000.

Since the HLT tracker has competitive efficiency, an imaginable step for the future is to combine the fast HLT track finder with its excellent efficiency and the better but slower offline track fit. Currently, there is an ongoing joint investigation by the Offline group and Sergey Gorbunov to use the HLT tracks as seeds for the offline tracker. In a later step, one could think about discarding clusters not associated to HLT tracks in order to reduce the storage demands further.

With the modifications to the offline QA macros, they can be used to evaluate online tracking quality automatically – and even more important – under exactly the same constraints as the offline QA.

ALICE is the only LHC experiment that employs GPUs for online analysis. Other high energy physics experiments like CMS and ATLAS employ conventional processors but have started evaluating GPUs for their update plans [Sch$^+$ 11, Eme$^+$ 12, Fun$^+$ 13]. In the current configuration, the ALICE HLT can process central heavy ion events at a rate of more than 200 Hz. The GPU tracking was active and ran stable during the heavy ion phases in 2010 and 2011, during the proton run in 2012 and during the proton-lead phase in 2013.[2]

[2] In 2010 and 2011, only a subset of the incoming events were processed due to a memory limitation in the framework and due to a bottleneck in the preceding cluster transformation component, which was unable to pass transformed clusters to the tracker at full rate. Starting with the proton run in 2012, the HLT GPU tracker was steadily enabled, tracking all events during data-taking.

Part III

Heterogeneous High Performance Linpack Benchmark

Chapter 10

Introduction to Linpack, DGEMM, and LOEWE-CSC

The second part of this thesis is about the Linpack benchmark: the standard tool for ranking HPC systems. After the necessary terms and definitions are introduced, Chapter 11 will describe a new HPL implementation for the LOEWE-CSC compute cluster. It is called **HPL-GPU** and based on a new DGEMM (Section 10.3.2) library called **CALDGEMM** [Roh+ 10 II]. Both are available as open source (see Appendix I). The results achieved with this implementation on the LOEWE-CSC have been published in [Bac+ 11 I]. The following Chapter 12 extends HPL-GPU to a wider set of architectures. In particular, it focuses on energy efficient clusters culminating in the Sanam cluster. Results of this chapter have been published in [Roh+ 11].

10.1 Heterogeneous Compute Clusters

In the last decades the size, the compute performance, and the power consumption of compute clusters have steadily been increasing. A point has been reached where the power becomes the dominant cost factor and where an enormous engineering effort is required for power supply and heat dissipation. This makes it prohibitively expensive to buy more performance with more electrical power [Fen 05]. The outdated pure speed metric must be replaced by a new measure that considers power consumption as well [Kam+ 08, Sha+ 06]. Nonetheless, users expect the compute performance to grow in the way they are used to. Evidently, conventional clusters cannot achieve this. Heterogeneous clusters employing special energy efficient accelerators are one way out.

Some heterogeneous clusters use special hardware such as CELL processors or ClearSpeed accelerators [End+ 10]. Recently, the trend has been moving to GPUs. In the last five Top500 lists [Meu+], always two to three of the top ten clusters have employed GPUs. The design varies from multi-GPU [Mat+ 10] to low-power GPU [Cop 10, Sco+ 10] approaches. When this work was started, power efficiency of GPU clusters was below 1 $^{\mathrm{GFlop}}/_{\mathrm{J}}$ and their achieved Linpack score was only about half the theoretical peak performance.[1] The following chapter introduces an optimized Linpack version that can reach above 70 % of the theoretical performance on the LOEWE-CSC.

10.2 The LOEWE-CSC Compute-Cluster

The LOEWE-CSC [Bac+ 13, Goe] is a heterogeneous compute cluster built by the University of Frankfurt in 2010. It consists of 34 racks (Fig. 10.1) of 12 boxes (Fig. 10.2) with two nodes

[1] The Tsubame cluster reached 53 % with ClearSpeed accelerators [End+ 10], competitive clusters with NVIDIA GPUs reach (in the June 2011 Top500 list) between 42.6 % (Nebulae) and 54.6 % (Tianhe-1A). The former Tianhe cluster [Wan+ 11] reached quite good 70 %, but on a single and slow node which boosts the efficiency.

each. AMD Magny-Cours twelve-core CPUs and AMD 5870 Cypress GPUs are employed. Most compute nodes are equipped with two CPUs and one GPU, resulting in 24 CPU cores. Some special nodes for applications with high CPU and memory demands provide four CPUs for a total of 48 cores. These nodes are called "**quads**" hereafter. The GPU nodes are equipped with 64 GB of main memory, the quads have twice that much. In total, 768 GPU-nodes and 40 quad-nodes are present – in addition to separate nodes for login, infrastructure, mass storage, etc. Two separate networks are installed: a gigabit Ethernet network for management and a high-speed 40 $^{\text{GBit}}/_{\text{s}}$ QDR InfiniBand interconnect providing half-bisectional bandwidth and low-latency RDMA[2].

Figure 10.1: The LOEWE-CSC Supercomputer [Goe]

Figure 10.2: One LOEWE-CSC Rack [Goe]

A new cooling concept is used in the LOEWE-CSC. All racks are equipped with a heat exchanger for water cooling in their back doors. The air pressure produced by the servers creates sufficient airflow making additional fans obsolete. The water is cooled outside the building with one or two evaporative coolers, depending on the environmental temperature. For these reasons, the extra power required for cooling is very low compared to other compute-clusters. The LOEWE-CSC has a PUE[3] below 1.1 [Bac$^+$ 13].

Also the compute nodes are very power efficient. The double precision peak performance of a Magny-Cours CPU is 100.8 $^{\text{GFlop}}/_{\text{s}}$. In combination with 544 $^{\text{GFlop}}/_{\text{s}}$ provided by the Cypress GPU, each node has a theoretical performance of 745.6 $^{\text{GFlop}}/_{\text{s}}$ while consuming about 750 watts.

10.3 Linpack

Linpack is a widely used benchmark for supercomputers and builds the basis for the semiannual Top500 supercomputer list [Meu$^+$]. It solves a dense system of linear equations $\widetilde{A} \cdot x = y$. The Top500 rules demand that the factorization algorithm must be of complexity $\mathcal{O}(N^3)$ where N is the matrix size. This ensures a measurement of the compute cluster's performance and not of the algorithm. Still, the implementation of the benchmark may be optimized individually for each supercomputer. In order to guarantee comparability of HPL results, which depend on the dimension of the matrix, all single-GPU results in this thesis, if not stated differently, are based

[2] **R**emote **DMA** – **R**emote **D**irect **M**emory **A**ccess.
[3] **P**ower **U**sage **E**ffectiveness is defined as the ratio "Total Facility Power" divided by "IT Equipment Power".

on a 64 GB matrix while multi-GPU tests use a 128 GB matrix. Systems which are included for comparison but do not have enough memory (see Appendix H) use the maximum possible matrix size.

It can be shown that solving a system of linear equations can be reduced (complexity-wise) to matrix multiplication. As a remark: the Strassen-Algorithm [Str 69] (see Section 16.5.2) can perform matrix multiplication in $\mathcal{O}(N^{\log_2 7})$. This can even be improved to $\mathcal{O}(N^{2.375477})$ as shown in [Cop$^+$ 87]. However, the latter one has no application on HPC as the appearing constants are far too large. In the following, the naive $\mathcal{O}(N^3)$ matrix multiplication algorithm is used exclusively and only its implementation improved.

10.3.1 High Performance Linpack

High **P**erformance **L**inpack (HPL) is an implementation of the Linpack benchmark provided by the University of Tennessee [Don$^+$ 03, UoT] and the University of Colorado. It employs the "**B**asic **L**inear **A**lgebra **S**ubprograms" (**BLAS**) and the LAPACK libraries [Don$^+$ 90], which are common interfaces for matrix / vector operations. HPL implements the benchmark by performing an iterative LU-decomposition ($\widetilde{A} = \widetilde{P}\widetilde{L}\widetilde{U}$ with an upper triangular matrix $\widetilde{U}$, a lower triangular matrix $\widetilde{L}$, and a permutation matrix $\widetilde{P}$) with row-partial pivoting and by solving the triangular system $\widetilde{U}x = (\widetilde{L}^{-1}\widetilde{P}y)$ afterward [Pre$^+$ 92, 2.3]. (See Appendix C.4 for a short introduction.) The LU-decomposition is calculated via Gaussian elimination, which computes $\widetilde{L}^{-1}\widetilde{P}y$ as a side-effect. Let $A = (\widetilde{A}|y) \in M_{N,N+1}$ be an $N \times (N+1)$ matrix. Fig. 10.3 shows all relevant submatrices.

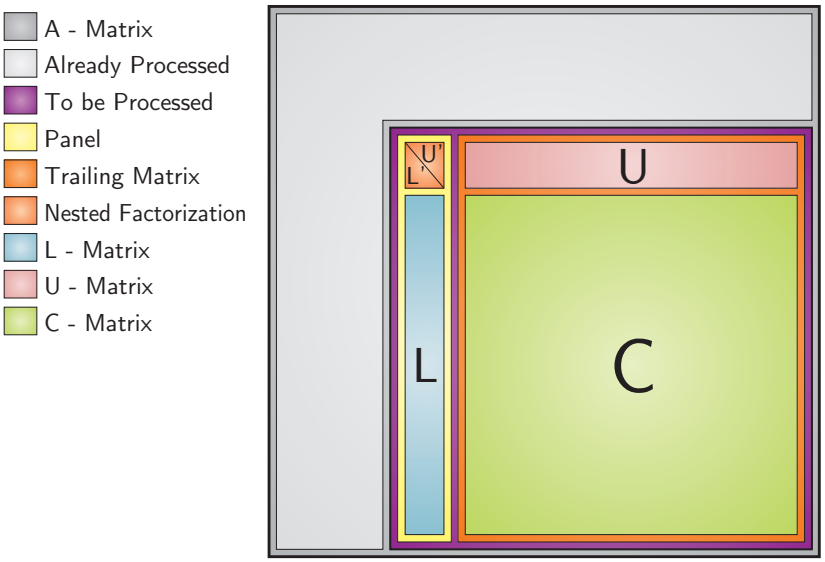


Figure 10.3: Submatrices in HPL

Each iteration of the outermost loop of HPL implements the following steps:

- **Recursive panel factorization** (or only **factorization**): a panel of N_b columns is factorized through LU-decomposition of the top-left $N_b \times N_b$ submatrix into L' and U' using an arbitrary solver as a black box. For numerical stability, a row pivoting algorithm swaps lines of the panel (which includes L, but neither U nor C) such that the diagonal matrix element, which occurs in the denominator during the Gaussian elimination, is maximal.
- **Panel broadcast** (or only **broadcast**): After the panel has been factorized, it is broadcasted to all nodes that take part in the computation.

- **Replication of row pivotization** (or only **pivoting**) consists of two steps: An **LASWP** function applies the row swaps carried out during the factorization to the trailing matrix as well. In preparation for the next steps, the ***U*-broadcast** distributes the current U-matrix to all nodes.[4]
- ***U*-matrix update** (or **DTRSM**): The row-operations originating from the Gaussian elimination for calculating L' and U' during the first step are replicated on the U-matrix. Mathematically, this corresponds to a multiplication by U'^{-1} from the left or to solving the triangular system $U' \cdot U_{\text{new}} = U_{\text{old}}$, which is accomplished by the BLAS routine DTRSM.
- ***C*-matrix update** (or **DGEMM**): The row operations for zeroing out the former L-matrix must be performed on C as well. The BLAS function DGEMM calculates $C_{\text{new}} = C_{\text{old}} - L \cdot U$.

The next iteration continues on the matrix C. After the last iteration, the matrix A has been transformed as $(\widetilde{A}|y) \Longrightarrow (\widetilde{L} \setminus \widetilde{U}|\widetilde{L}^{-1}\widetilde{P}y)$ (see Appendix C.4). It is important to note that the panel factorization step involves lots of BLAS calls, among them many calls to DGEMM and DTRSM. In the following, if not stated differently, DTRSM and DGEMM refer to the above steps of the iterative factorization and not to the nested calls inside the recursive panel factorization.[5]

In multi-node runs on HPC-clusters, the matrix is distributed among many compute nodes. These nodes are arranged in a **grid**. The distribution is further described in Section 12.4.

All steps except for the final matrix multiplication (DGEMM) can be performed in $\mathcal{O}(N^2)$.[6] The $\mathcal{O}(N^3)$ matrix multiplication is the time critical part of HPL, at least considering the computational effort. Memory or IO-bound operations can still have a significant contribution to execution time. Table 10.4 compares the contribution of all relevant functions to the overall HPL execution time in an exemplary configuration of sixteen nodes (without distinguishing DGEMM calls inside of and outside of the factorization). It becomes clear that the DGEMM, which does the matrix multiplication, is the most important part and is thus analyzed in more detail.

Function	DGEMM	DTRSM	DSCAL	DGEMV	DCOPY	DAXPY
Time [%]	96.59	2.57	0.35	0.21	0.20	0.08

Table 10.4: Total Contribution of BLAS Functions to HPL Runtime

10.3.2 Double Precision General Matrix Multiplication

DGEMM performs a "Double-Precision General Matrix Multiplication". It computes the generalized matrix-product $C' = \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C$ where C' is overwritten onto C, α and β are scalars, A is an $m \times k$-matrix, B is a $k \times n$ matrix, and $\text{op}(A)$ can be A or A^t. For the time being, the transposition parameters shall be ignored. Both A and B are assumed transposed or not transposed, whichever can be processed better. Section 11.2.5.1 explains how transposed matrices are treated. Both row- and column-major format are supported (see Appendix C.2).

HPL can use different BLAS libraries. For the Linpack on the LOEWE-CSC, the GotoBLAS2 library [TAC] is used. Its DGEMM has an outstanding performance. It employs multiple blocking levels[7] and is specifically optimized to respect page boundaries and to minimize TLB[8] misses.

The ACML-GPU[9] library offloads BLAS routines to AMD GPUs but could not utilize the GPU to the desired extent. Hence, it was decided to start a new approach from scratch.

[4] The broadcast is performed by a spread and roll algorithm as described in [UoT].
[5] The recursive factorization of the panel employs essentially the same algorithm with smaller and smaller block sizes recursively. Finally, at very small block sizes, the above algorithm produces too much overhead and the matrix is factorized directly instead.
[6] The matrix multiplication is at least $\mathcal{O}(N^2)$ as it must calculate at least N^2 distinct values. All known algorithms have a higher complexity. Hence, matrix multiplication is the dominant part of HPL.
[7] Blocking improves cache utilization in matrix multiplication. See Section 11.2.4.1 for details.
[8] **T**ranslation **L**ookaside **B**uffer.
[9] **A**MD **C**ore **M**ath **L**ibrary for GPU [Adv I].

Chapter 11

An Optimized HPL Variant for the LOEWE-CSC

11.1 Target Architectures

During this thesis a version of HPL has been developed, which is specifically designed for good GPU utilization. The main objective was to maximize the Linpack performance for the LOEWE-CSC architecture. Therefore, in the first phase, only one particular architecture is considered: Two AMD Magny-Cours CPUs with one Cypress GPU per node. Optimizations for this specific hardware were pushed to their very limit until the submission for the Top500 list in late 2010.

In a second stage, the Linpack and especially the DGEMM library were optimized for good utilization of other architectures as well. Of course, not each and every hardware constellation could receive as much devotion as the LOEWE cluster. Special focus is laid on the AMD Cayman and Tahiti GPU series, on multi-GPU configurations, on Intel platforms, on other GPU programming APIs, and in particular on the Sanam cluster. See Appendix G for platform-related features.

11.2 CALDGEMM

The CALDGEMM library provides a GPU-based DGEMM. It has been developed at the University of Frankfurt for this thesis and is available as open source (see Appendix I). Initially, it ran on AMD Cypress hardware only. It was later improved to run on a wider variety of hardware, too (see Chapter 13). It consists of two parts: The kernel, which performs the actual matrix multiplication on the GPU, and the framework, which handles DMA transfers as well as data pre- and postprocessing. In the first version, the kernel from the "*double_ matmult*" example of the AMD Stream SDK [Adv 09] was adopted. The framework itself is written from scratch.

11.2.1 GPU-based DGEMM

At first, only the matrix-product $C = A \cdot B$ is considered. Scalars and addition can easily be included later and are ignored for the time being. When not explicitly stated otherwise, $k = 1024$ is assumed. The justification for this will come up later in this chapter. The BLAS interface supports both column-major and row-major matrices.[1,2] Switching from column- to row-major and vice versa corresponds to the transition $A \cdot B \Rightarrow (A^t \cdot B^t)^t = B \cdot A$ so only the order of the operands needs to be changed. Therefore, in the following all matrices can be assumed row-major.

[1] Refer to Appendix C.2 for the definition of row- and column-major.

[2] Column-major matrices are primarily used by Fortran programs, while C code usually implements the row-major version.

The 5870 AMD GPU has 1 GB of memory, the LOEWE-CSC nodes offer 64 GB of main memory. Typically, the HPL matrix size is chosen as large as possible, i. e. as large as the host memory. Therefore, it is not possible (or even desirable) to process the whole DGEMM calculation in a single step on the GPU. Instead, a pipelined streaming approach is implemented, which operates on tiles. The matrices A, B, and C are divided into submatrices A_i, B_j, and $C_{i,j}$ as shown in Fig. 11.1. Without loss of generality, k is assumed small ($k \approx 1024$) because a DGEMM with large k parameter can be emulated by multiple DGEMMs with smaller k. All dimensions are powers of two or multiples of higher powers of two for all but the last A_i and B_j respectively (and the corresponding $C_{i,j}$). The $C_{i,j}$ are called **tiles**. The matrix-product $C = A \cdot B$ can thus be calculated as $C_{i,j} = A_i \cdot B_j$. Each LOEWE-CSC node is equipped with two twelve-core Magny-Cours processors. These 24 cores provide a non-negligible amount of compute power. They can be easily used for the DGEMM as well by calculating some tiles on the CPU (see Section 11.2.3 for details).

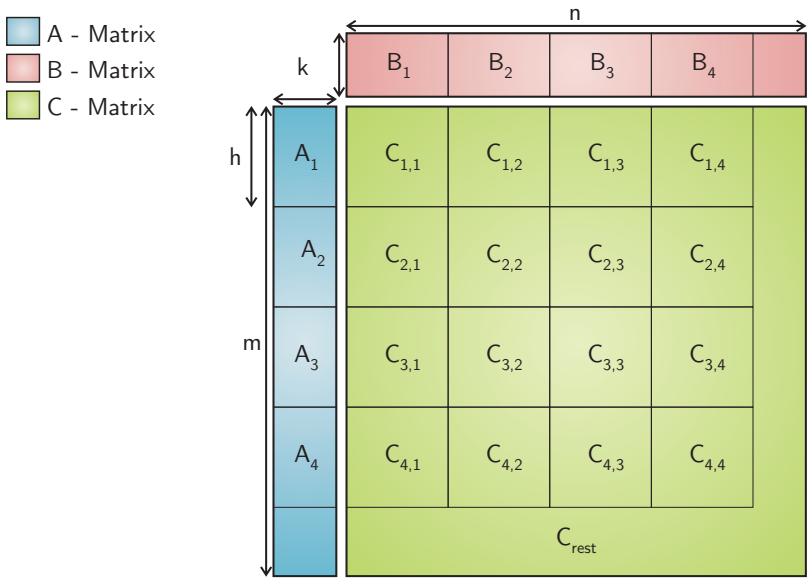


Figure 11.1: Splitting of Matrices in Tiles for Streaming DGEMM

If the CPU treats the remainder part C_{Rest} (corresponding to the last A_i and B_j), as a further simplification, all GPU tiles $C_{i,j}$ can be assumed as square matrices of dimension h. The parameter h is later set to a value that is optimal for the GPU.

The best data arrangement for GPU memory fetches and stores is not necessarily the format of the plain input and output matrices A_i, B_j, and $C_{i,j}$, e. g. it might be good to store multiple rows in an interleaved format.[3] CPU based pre- and postprocessing reformats the data accordingly. These steps are called "***DivideBuffer***" and "***MergeBuffer***" respectively.[4] The *DivideBuffer* function can also be used to transpose the input matrices, as required by the BLAS specifications.

The first naive CALDGEMM implementation works in the following way. The output matrix C is divided in two parts: one is processed by the CPU and the other by the GPU. This is done in a way, such that the GPU part fulfills the above size constraints. GotoBLAS processes the CPU part. The GPU part is streamed through the GPU. First, the input matrices A_1 and B_1 are preprocessed and then transferred to the GPU. Thereafter, the GPU kernel is executed to calculate the matrix-product, which is transferred back to the host. Finally, the postprocessing is performed. Then the steps are repeated with the next tile. Fig. 11.2 visualizes the process.

[3] The storage format is described in Section 11.2.4.2.

[4] The names originate from the AMD Stream SDK sample *double_ matmult*, which CALDGEMM was originally based on.

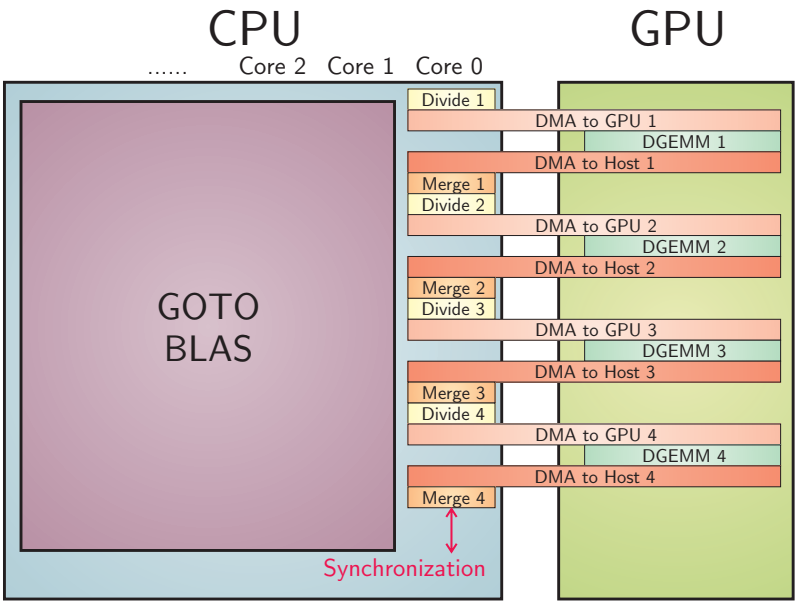


Figure 11.2: Process-Flow of first CALDGEMM Implementation

There are two apparent issues. Workloads for GPU and CPU must be chosen such that none of them idles toward the end. Hence, either the required computation time must be known in advance or they must be scheduled dynamically. In addition, the pre- and post-processing as well as the transfer to and from the GPU should overlap with GPU calculation. Otherwise, good GPU utilization cannot be achieved.

11.2.2 Implementation Details

For certain reasons explained later, CALDGEMM spreads both each input and each output tile over multiple buffers (in interleaved or more complicated formats). The usage of multiple input buffers can reduce the register requirements (see Section 11.2.4.1), alter the cache access pattern, and is a general prerequisite for Color Buffer output (see Section 11.2.4.1).

Theoretically, Zero-Copy allows for locating both the input and the output buffers on the host or the GPU. In practice, CALDGEMM always locates the input memory on the GPU as the input throughput of the kernel exceeds the PCI Express bandwidth by far. For the output memory, both variants are implemented. Input data are preprocessed by *DivideBuffer* into page locked buffers on the host of exactly the same size as the input buffers on the GPU. Then, these buffers are transferred via one large DMA transfer. The output is either directly written to the host or it is first written to GPU memory and transferred to the host via a single huge DMA transfer. The original C-matrix is never sent to the GPU as it is read only once. The GPU calculates only $X = \alpha \cdot AB$ while the *MergeBuffer* routine calculates $C' = X + \beta \cdot C$. Section 11.2.4.1 explains how transposed matrices are processed. In this way, a full DGEMM is implemented. The parameters m, n, k, and α are stored in constant buffers on the GPU.

11.2.3 Combined GPU/CPU DGEMM

This section describes in more detail how the combined GPU/CPU DGEMM is implemented. The matrix is split in m-direction, where the upper part is processed by the GPU.[5] To be

[5] The reason to split the matrix in m direction is to have GPU and CPU both process consecutive memory segments.

precise, the GPU part ranges on the m-scale from 0 to $u \cdot h$ with u minimal such that $u \cdot h \geqslant m \cdot r$ where $r \in [0, 1]$ is the GPU-ratio. The first naive implementation calculates the ratio by

$$r = \frac{p_{\text{GPU}}}{p_{\text{GPU}} + p_{\text{CPU}}},$$

with p_{GPU} and p_{CPU} the projected performances of the CPU and the GPU DGEMM for the employed matrix size. Section 11.2.4.4 explains the estimation of these parameters.

As Fig. 11.2 shows, one processing component is likely to idle toward the end of the DGEMM. It is desired that rather the CPU idles because it is the slower processor. Hence, the GPU performance is overestimated by requiring $u \cdot h \geqslant m \cdot r$ instead of taking the closest multiple of h. The GPU tiles are calculated in a loop over the vertical direction and an inner loop over the horizontal direction (see Fig. 11.3). The CPU computes also the rightmost n mod h columns due to size constraints.

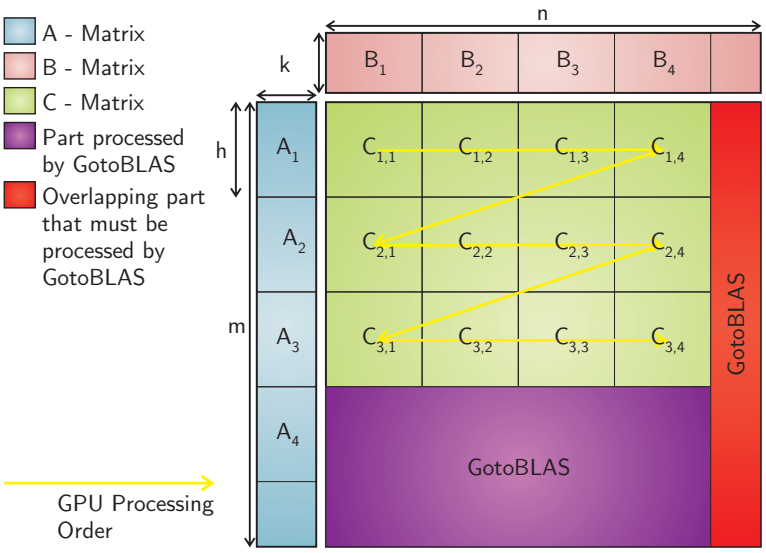


Figure 11.3: Distribution of DGEMM Workload on GPU / CPU

Splitting the matrix in an upper and a lower part does not work well with flat matrices ($n \gg m$), especially when $m \approx h$. Therefore, flat matrices are split in a left and a right part rather than upper and lower parts. In that case, every step that depends on the splitting simply works the other way around. In principle, everything remains the same. To keep the notation simple, assume $n \leqslant m$ for the rest of this chapter.

11.2.3.1 CPU Affinity

CALDGEMM requires CPU resources, even for the GPU part of the matrix, namely the *DivideBuffer* and *MergeBuffer* functions. For this purpose, it includes a thread-server implemented with pthreads synchronized by POSIX semaphores, which handles all multi-threading requirements. Each dual-CPU LOEWE-CSC node has four independent memory controllers: one per CPU die.[6] The available PCI Express bandwidth depends on which memory controller connects the host memory for the transfer (see Fig. 11.4). Only one die, on the LOEWE-CSC it is the first one (die 0), is connected directly via HyperTransport to the chipset which connects the GPU via PCI Express. All GPU related memory (i. e. input and output buffers) is thus allocated on DRAM connected to die 0.[7] To allow for fast access by the *DivideBuffer* and *MergeBuffer* func-

[6] Each Magny-Cours CPU internally consists of two dies with one memory controller each (see Section 2.2.1).
[7] Measurements of 5870 and 6970 DMA performance in a multi-GPU environment are presented in Fig. 12.33 in Section 12.8.2. They reveal that one should use different memory controllers for multi-GPU.

tions, the threads executing those are pinned to cores on die 0 as well. Later, a multi-threaded CALDGEMM (with parameter t) is introduced that runs $t + 1$ GPU related threads. These threads are pinned to cores 0 to t. For the serial version presented before, assume $t = 0$.

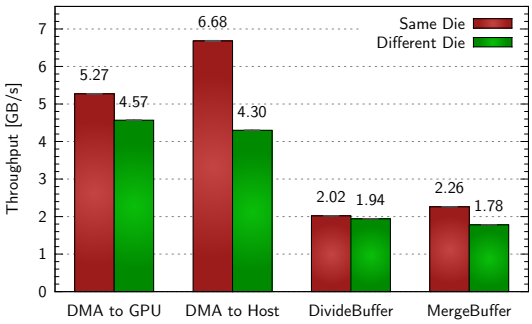


Figure 11.4: Dependency of PCI Express Bandwidth on CPU Die (AMD) [V]

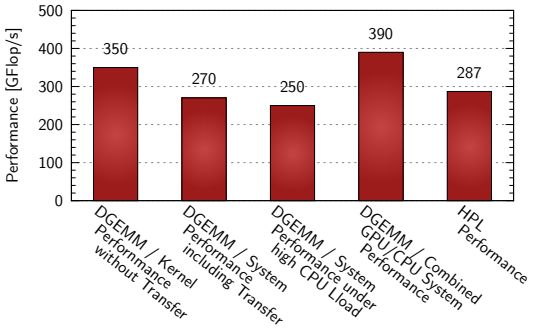


Figure 11.5: Performance of the very first CALDGEMM / HPL Implementation [V]

Utilizing $t + 1$ threads for the GPU part of CALDGEMM leaves $23 - t$ of the 24 cores available. To avoid congestion on the CPU cores, GotoBLAS should use only cores $t + 1$ to 23. Since GotoBLAS comes with its own thread pinning policy, a patch has been implemented that allows for excluding CPU cores from its pinning mechanism.[8] Fig. 11.5 shows the performance achieved by the first implementation in the very first successful HPL run (with the CAL SDK kernel).

11.2.4 DGEMM Optimizations

CALDGEMM has been optimized in two steps. To ensure the maximum possible GPU performance in the final version, the kernel performance is optimized independently of the framework. Afterward, the framework is optimized with respect to the boundary conditions set from the optimal kernel. The idea is that a good framework should be able to sustain almost the full kernel performance in the application whereas, as a matter of fact, any performance lost in the kernel is irretrievably lost in the application as well.

11.2.4.1 Kernel Optimization

Older GPU DGEMM implementations, especially for NVIDIA GPUs, usually used shared memory / **L**ocal **D**ata **S**torage as described in [Vol+ 08]. However, the AMD LDS is not capable of delivering sufficient bandwidth to obtain full ALU utilization on Cypress GPUs. Therefore, N. Nakasato [Nak 10] concludes that it is better to read the input data through the texture cache.[9]

This subsection only handles the processing of one tile $C_{i,j}$ of size $h \times h$, which will be called C hereafter. The tile is subdivided into $C^{\mu,\nu}$ again (with upper indices μ, ν – the lower indices i, j are omitted). The same also holds for A and B. However, this is absolutely unrelated to the previously described tiling. Lower indices determine the tile processed on the GPU at a time, upper indices determine the block of the tile processed by one thread on the GPU (see below). The reader should keep this in mind so that no ambiguity occurs.

The CALDGEMM kernel is written in the AMD Intermediate Assembler Language (IL) for achieving optimal performance [Adv 10 II]. Preliminary tests with OpenCL yielded only unsatisfactory results and other high level languages were not yet available by then.

[8] GotoBLAS implements two routines for this: one for NUMA aware operating systems and for the rest. Although only the NUMA variant is used on the cluster, both routines are patched to allow for the greatest possible compatibility.

[9] N. Nakasato [Nak 10] also provides a faster kernel implementation than the one from the AMD Stream SDK. However, this was not available when the work for CALDGEMM started.

Blocking Since blocking is a common technique that can operate on non-square matrices in general, this paragraph on blocking assumes a matrix tile $C(= C_{i,j})$ of size $\widetilde{m} \times \widetilde{n}$. (So C is an $\widetilde{m} \times \widetilde{n}$ matrix, A an $\widetilde{m} \times k$ matrix, and B a $k \times \widetilde{n}$ matrix.) This notation is used exclusively in this paragraph and afterward, square tiles of size $h = \widetilde{m} = \widetilde{n}$ are considered again.

Calculating the DGEMM result needs $\widetilde{m} \cdot \widetilde{n} \cdot (2k + 2)$ floating point operations, out of which the GPU computes $\widetilde{m} \cdot \widetilde{n} \cdot 2k$ operations. If all destination entries are processed independently and no input data are reused, $\widetilde{m} \cdot \widetilde{n} \cdot 2k$ memory fetches are required, i. e. exactly one memory fetch per floating point operation. A peak performance implementation with this instruction to memory fetch ratio exceeds the GPU memory bandwidth by far. Even the cache bandwidth is insufficient.

The number of fetches can be reduced with a technique called **blocking**, which is already known from CPU based matrix multiplication. In the literature, blocking is also called tiling. Since the labeling "tiling" has already been used in another context throughout this thesis, it is not used in this meaning. Each C-matrix tile is further subdivided into blocks $C^{\mu,\nu}$ of dimension $a \times b$ where a is the vertical blocking size and b the horizontal, and each thread processes a different $C^{\mu,\nu}$. The A- and B-tiles are also subdivided into $A^{\mu,l}$ and $B^{l,\nu}$ of dimension $a \times 1$ and $1 \times b$ respectively. Then, C can be calculated by $C^{\mu,\nu} = \sum_{l=1}^{k} A^{\mu,l} B^{l,\nu}$. The blocking optimization works in the following way: in each addition step, the matrices $A^{\mu,l}$ and $B^{l,\nu}$ are loaded into registers. Then, each element of the one is multiplied with each element of the other and the result is added to the corresponding entry in $C^{\mu,\nu}$ (see Fig. 11.6).

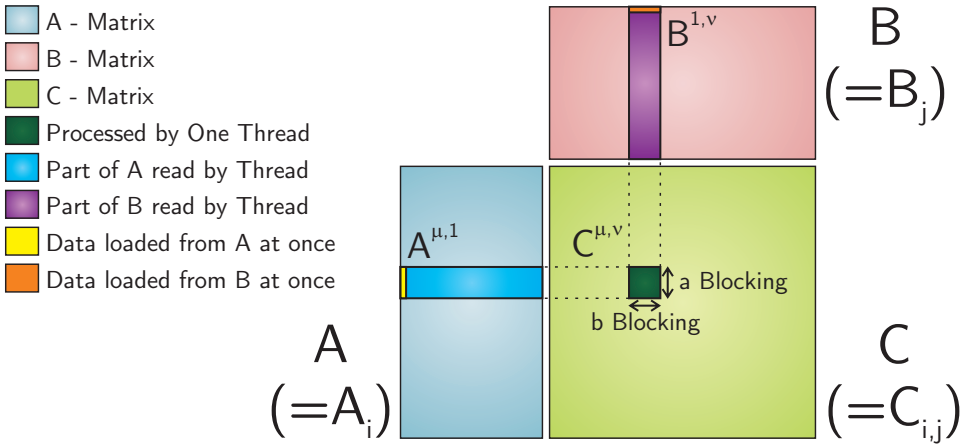


Figure 11.6: Blocking inside the DGEMM Kernel

This requires only $k \cdot (a + b)$ memory loads to calculate $a \cdot b$ entries of C, for a total of

$$\frac{\widetilde{m}}{a} \cdot \frac{\widetilde{n}}{b} \cdot k \cdot (a + b) = \widetilde{m}\widetilde{n}k \cdot \frac{a + b}{ab}$$

memory fetches. The number of memory accesses is thus reduced by a factor $\frac{2ab}{a+b}$. Obviously, a symmetric blocking $(a = b)$ is optimal.

If x memory fetches of s bytes each are required for a calculation involving y floating point operations, a memory bandwidth of $^{xsg}/_{y}$ is needed for reaching a performance of g. The peak performance of the employed GPUs is 544 $^{\text{GFlop}}/_{\text{s}}$ in double precision which translates to 4352 $^{\text{GB}}/_{\text{s}}$ for $a = b = 1$ and 1088 $^{\text{GB}}/_{\text{s}}$ for $a = b = 4$. The 5870 has 20 multiprocessors each offering a texture cache bandwidth of 54.4 $^{\text{GB}}/_{\text{s}}$, i. e. 1088 $^{\text{GB}}/_{\text{s}}$ in total. Thus, a 4×4 blocking is exactly sufficient to achieve peak performance, given that the cache hit efficiency is 100 %. Of course, this is unrealistic and suggests analyzing bigger blockings.

The following variants are implemented: 8×2, 4×4, 8×4, 4×8, and 8×8 blocking.

Register Usage Since each GPU thread calculates one of the $C^{\mu,\nu}$ submatrices of C, the thread needs at least $a \cdot b$ registers for $C^{\mu,\nu}$, a registers for $A^{\mu,l}$, and b registers for $B^{l,\nu}$, plus 3 registers for μ, ν, and l, making a total of 3 integer registers and $(a+1)\cdot(b+1)-1$ double precision registers. As the GPU offers common 128-bit registers for integer and floating point, $\frac{(a+1)\cdot(b+1)+1}{2}$ registers are needed. All kernels benchmarked later are implemented such that they really hit this lower register boundary, which is e. g. 41 for the 8×8 kernel, the biggest one implemented. A simple trick to reduce the number of registers is to interleave multiple input data buffers. This way the same relative pointer can be used within multiple buffers to fetch multiple entries, one from each buffer, e. g. by four-fold row-based interleaving the pointer to the buffer position (x, y) can be used to fetch the matrix entries $(x, 4y)$ to $(x, 4y + 3)$. Obviously, the highest possible number of threads should be running concurrently to hide memory latencies. However, due to the limited register file, thread-count and blocking size cannot both be maximized at the same time. This shows that a larger blocking is not always preferable and that the trade-off point has to be found.

For large blockings the register usage is critical while for the 4×4 kernel it is not: the unrolled kernel in the next paragraph caches some entries in registers and does not reach the lower limit. Nonetheless, it is the faster version.

Unrolling & Hardcoding Constants The kernel source code explicitly contains the multiplication of $A^{\mu,l}$ with $B^{l,\nu}$, i. e. there is one line of code for each scalar multiply-add. The entry $C^{\mu,\nu}$ is calculated with a single loop over l. This loop can be unrolled to achieve optimal performance. Fig. 11.7 shows the performance of the 4×4 kernel and different unrolling factors. The optimal factor depends on the blocking size. (Bigger blocking sizes need less unrolling.) For each kernel individually, the optimal parameter is determined experimentally.

Furthermore, the k constant can be hardcoded in the kernel instead of being read from a constant buffer. A special kernel with hardcoded $k = N_b = 1024$ is in stock. The correct kernel is chosen at runtime depending on the actual k. Fig. 11.8 shows the result. As can be seen, a hardcoded k does not always improve kernel performance.[10] However, for the B transposed kernel variant, which is used for HPL in the end, the performance increases. Analogously, special kernels for $\alpha = 1$ and $\alpha = -1$ have been, which spare oneself the scalar multiplication.

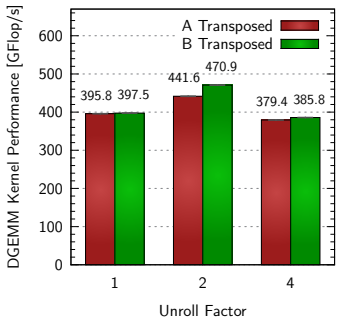


Figure 11.7: Performance of unrolled Kernels [V]

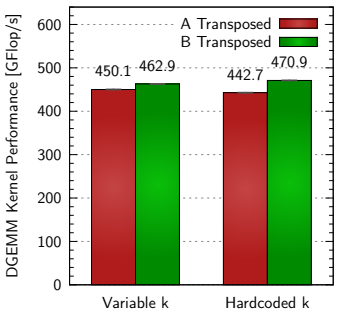


Figure 11.8: Improvements with hardcoded k [V]

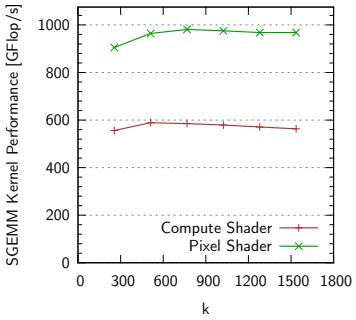


Figure 11.9: Comparison of SGEMM Shader Types [V]

Cache organization The texture cache can be configured in two ways: linear or tiled.[11] The tiled mode is optimized for two-dimensional access patterns such as for matrices. It is by far the faster mode for CALDGEMM (470.8 $^{\text{GFlop/s}}$ *v.s.* 98.9 $^{\text{GFlop/s}}$).[12]

[10] The loop optimizer of the current compiler does not always work well and should be improved.

[11] See [Adv 09, 1.11] for a description of the tiled mode.

[12] It has to be admitted that the kernel used for the comparison has been tuned for the tiled mode. It should be possible to improve the kernel for linear mode quite a bit. However, tuning for linear access is much more complicated and was thus not attempted. On top of that, tiled mode is optimized for two-dimensional patterns and should be faster for matrix operations anyway.

Output Memory The Cypress chip offers two ways to write data to the output memory:

- **Color Buffers**: Each thread can output to a maximum of eight buffers containing *float4* (a vector of four floats) entries. The position of the thread in the execution configuration automatically defines the storage position of each thread in the buffer, so address calculation is neither necessary nor even possible. As a thread calculates adjacent entries of the destination matrix but the output is written to eight distinct buffers, the output format cannot be the native memory format of the C-matrix. This already shows that the *MergeBuffer* functionality is mandatory for Color Buffers. Eight *float4* Color Buffers allow for storing sixteen double values and are therefore unsuited for large blockings.[13]
- **MemExport**: The newer AMD chips starting with the Cypress support the MemExport function. With MemExport threads can directly address and access a linear destination buffer (the **global buffer**). However, only one Global Buffer can be used at a time. This leads to some problems which are discussed in Section 11.2.5. All kernels with large blocking use the MemExport function in lack of alternatives. Additionally, the smaller kernels have been implemented with the MemExport function, too.

Besides the output method, different output locations can be chosen. The output can be written to the global GPU memory or to page locked host memory. In the first case, the data are transferred to the host after the kernel execution in one big DMA transfer. In the second case, the output is directly written to the host memory via DMA by the kernel (Zero-Copy). So either an extra DMA transfer is required or the kernel execution might be slowed down due to latencies of the host memory access. For each kernel variant the better implementation is determined experimentally.[14]

Shader Type In addition to Pixel Shaders, the Cypress chip supports Compute Shaders. Compute Shaders are meant not for graphics but allow for more complex GPGPU programs. They do not support Color Buffers but only the MemExport output. It turns out that the Compute Shader version is inferior to the Pixel Shader version with MemExport, although the kernel is identical except for the Shader Type definition. (Fig. 11.9 shows the SGEMM performance measured with both shader types at different k.[15]) The Compute Shader approach was thus not followed.

Matrix Size The matrix size affects the memory access pattern and thus the cache hit ratio. Hence, it is relevant for the performance. The tiling enables the usage of almost arbitrary matrix sizes. So they are chosen in a way that maximizes the performance. Fig. 11.10 shows the performance for multiple h and k values.[16] In theory, non square matrices enable more degrees of freedom for the matrix size, but measurements demonstrated that the individual parameters $\widetilde{m}$ and $\widetilde{n}$ (from the blocking paragraph) affect performance very similarly such that setting $\widetilde{m} = \widetilde{n} = h$ is no real restriction. Using such plots, for each kernel the matrix sizes suited best are determined.

The performance decreases significantly when h drops below 1024 while it increases very slightly above 1024. CALDGEMM uses large h and drops below 1024 only if enforced by matrix size restrictions. The total peak is observed at $k = 512$. However, small k lead to a higher synchronization overhead and thus a larger value is needed. Above $k = 1024$, the performance decreases more rapidly. Therefore, even as the kernel is not the absolutely fastest one, $k = 1024$ remains the chosen value for HPL. Figures 11.11 and 11.12 show more detailed plots for some chosen k and h.

The final values used for h are 512, 1024, 2048, 3072, and 4096 depending on the matrix size. Section 11.2.5.1 has more information. The value 512 is used only when absolutely necessary, namely in the case when one of the total matrix dimensions is very small, i. e. $m < 1024$ or $n < 1024$.

[13] The 4×4 and 8×2 blockings can use Color Buffers. All other blockings are restricted to MemExport.
[14] The performance of the output types is shown in Fig. 11.13 in the overall kernel comparison at the end of this section.
[15] The SGEMM (Single-Precision General Matrix Multiplication) performance in the plot is obtained from the AMD SDK example and not from the SGEMM implementation introduced in Section 12.13.
[16] The kernel used for the figure is the best-performing kernel that is determined only at the end of this chapter. Showing all plots for all kernels would by far exceed the scope of this document.

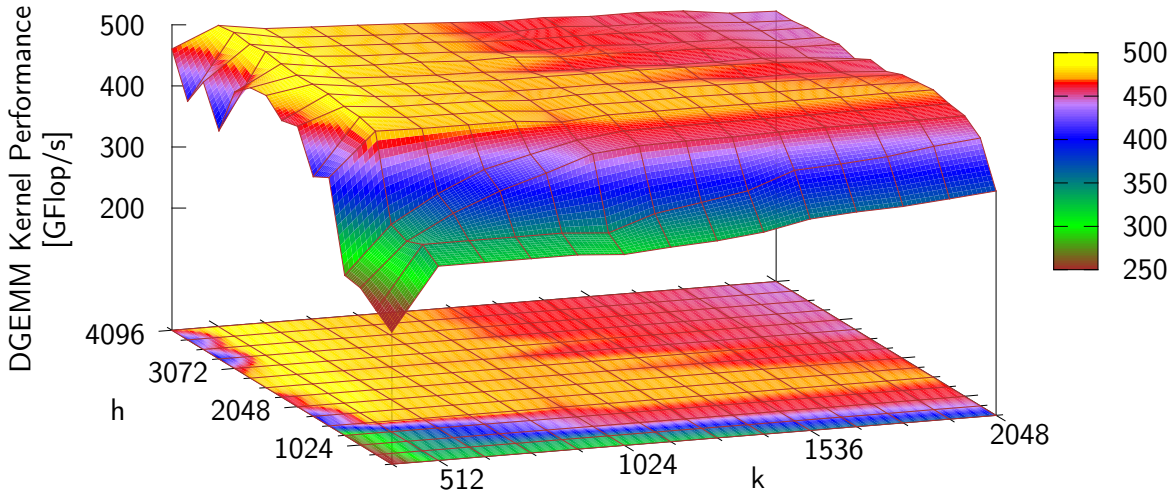


Figure 11.10: DGEMM Kernel Performance for different Matrix Sizes [V]

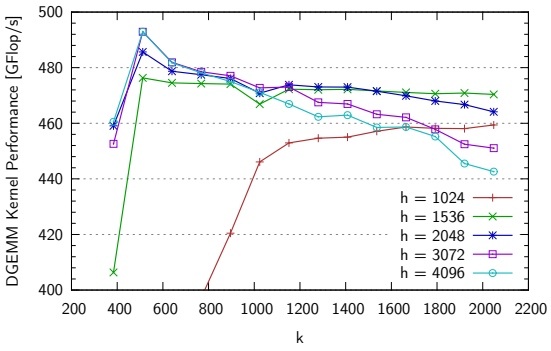


Figure 11.11: Kernel Performance at different k [V]

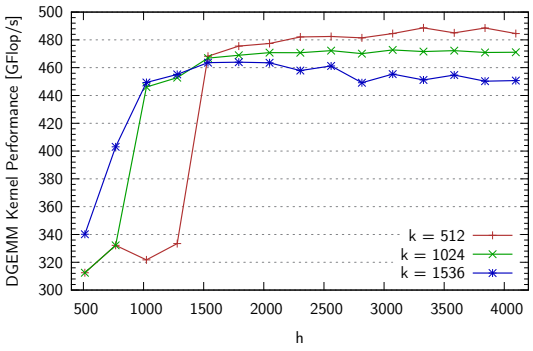


Figure 11.12: Kernel Performance at different h [V]

Transposed Matrices For complying with the DGEMM specification, either the kernel or the *DivideBuffer* routine must be capable of transposing input matrices. For the moment, only the influence on kernel performance shall be considered.

The schemes how the matrices A and B are read in the innermost loop of matrix multiplication with blocking are different: A is read column-wise and B is read row-wise. Consider a symmetric blocking, which will turn out to be the most effective one: if A is transposed, it requires exactly the row-wise scheme of B (untransposed) and vice versa. Either the row-wise or the column-wise scheme is faster and thus should be used exclusively – for both A and B. For this reason, many of the following sections only treat the case where exactly one matrix is transposed.

The GPU always reads a *double2* value at a time. For a transposed B-matrix, this means that two columns are read at once. This is unsuited regarding the blocking approach, where only entries of one column are required at the same time. To avoid this, the data structure of the input matrices can be altered in a way that two consecutive doubles in memory correspond to the same column. (See Section 11.2.4.2 for an illustration of the storage format.) The conversion is done by *DivideBuffer*. The same issue exists with non-transposed A-matrices and is solved analogously. Unrolled kernels always need at least two columns and are unaffected. Kernels without transposition, A transposed, B transposed, as well as both A and B transposed are implemented with all blocking methods. The correct version of *DivideBuffer* is chosen depending on how the kernel expects its input and how the data are actually formatted. For symmetric blocking, the case with both matrices transposed is identical to the one with none transposed so

it is omitted. For the large 8×8 blocking, the B transposed part has not been implemented after the A transposed version turned out not to perform well and because the effort to introduce the above-mentioned special data structure for reading single columns is considered inappropriate.

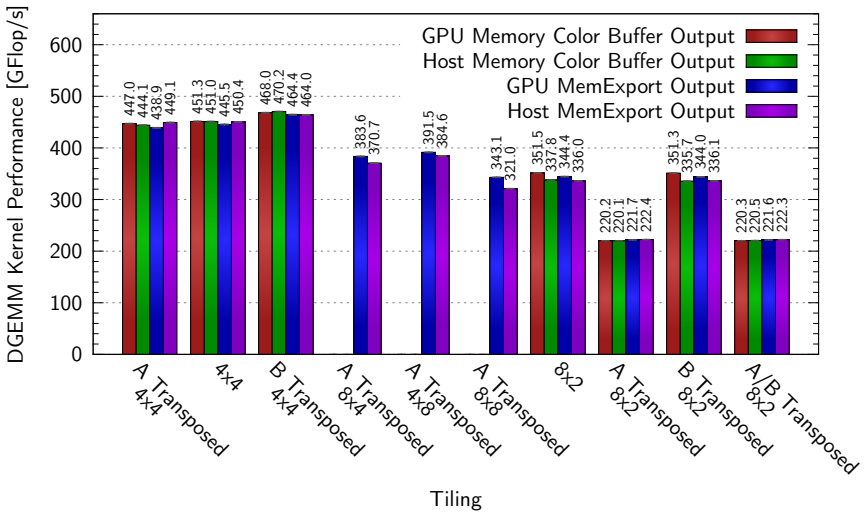


Figure 11.13: DGEMM Kernel Performance Overview [V][17]

Fig. 11.13 shows the results using all kernels with all transposition, blocking, and output settings. All presented results are tuned individually using the best unrolling factor, matrix blocking and tiling sizes, as well as hardcoded constants.

In contrast to previous DGEMM implementations for Cypress GPUs [Nak 10, Vol+ 08, NVI], the 4×4 kernel reaches at least 450 $\mathrm{GFlop/s}$ independent of whether the input matrices are transposed or not. It is obvious that the 4×4 kernel with transposed B-matrix is the fastest one. It turns out that the MemExport output is generally a bit slower than Color Buffers. This is a fortunate coincidence as the MemExport output causes a problem which is described in Section 11.2.5.

Comparing the output location unveils an interesting fact. For most kernels, the GPU memory is faster – as expected due to the PCI Express delay. However, it is the other way around for the fastest kernel: 4×4 with B transposed. The most probable explanation is that the input data rate already exhausts global GPU memory bandwidth. The PCI Express bandwidth is available in addition, thus the output does not slow down the input. Since writing to host memory directly is also the best case from the synchronization perspective, this version is used in CALDGEMM.

Best Parameters In summary, the kernel used in all later benchmarks on the LOEWE-CSC has the following characteristics:

- 4×4 blocking.
- B-matrix transposed.
- Pixel Shader kernel.
- Output via Color Buffers.
- Output buffer located in host memory and accessed by the kernel directly via DMA (Zero-Copy).
- Loop unrolled with an unrolling factor of two.
- $k = 1024$.
- $h \in \{512, 1024, 2048, 3072, 4096\}$ (chosen at runtime as described in Section 11.2.5.1).
- Texture cache set to tiled mode.

[17] The performance of the host memory output depends to a large extent on the system platform (see Section 12.5.1).

11.2.4.2 Data Buffer Format

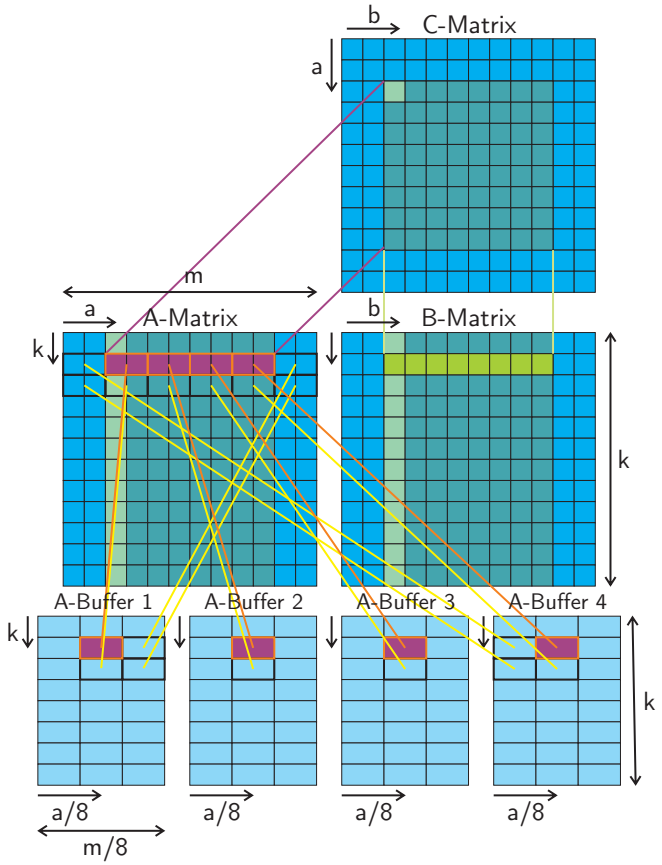


Figure 11.14: Storage Format of Input Buffers for 8×8 Kernel with A transposed

Figures 11.14 and 11.15 show the storage format of the input data for the 8×8 A transposed kernel and the 4×4 B transposed kernel without unrolling. In every iteration, the 8×8 kernel reads four values from four distinct buffers, all storing a part of A. Analogously, four values of the B-matrix are read. As A is stored in transposed form, A and B are read in exactly the same way.

The format for the B Transposed kernels is slightly more complicated. In Fig. 11.14 eight values within one row could be read by fetching four *double2*-values from the four distinct buffers. However, it is not possible to read four values within one column using two *double2* fetches because each *double2* fetch automatically reads two values from one row. To cope with this behavior, data must be stored interleaved, such that the two components of the *double2*-vector belong to the same column. Fig. 11.15 visualizes the data format. Clearly, this format prohibits the usage of simple streaming *double2*-copy instructions in *DivideBuffer*.

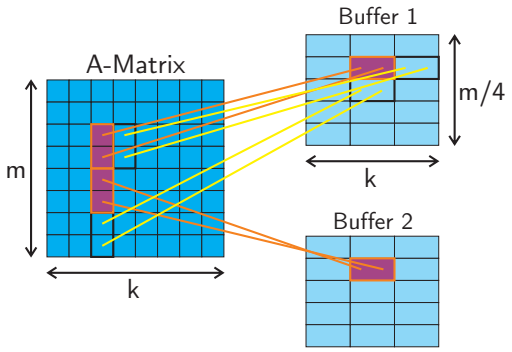


Figure 11.15: Storage Format of Input Buffers for 4×4 Kernel with B transposed and without Unrolling

The situation is different for unrolled kernels because there, two columns are needed within each iteration. Theoretically, registers could be saved by reading the first column first, then performing the multiplication, and only then reading the

second column. However, the negative effect of the context switch between ALU instructions and texture fetches outweighs the register benefit. (Storing four extra values for the A and the B-matrix requires only four additional (128-bit) registers.)

11.2.4.3 Exemplary 8 × 8 Kernel

The 8×8 kernel with A transposed, although not the fastest kernel, can be considered as the reference implementation. Hitting the lower register boundary is most critical in this case. Every other kernel (with A transposed) can be derived directly from this one by reducing the number of input buffers. This section presents the original IL (**I**ntermediate Assembler **L**anguage) code and the compiled ISA (**I**nstruction **S**et **A**rchitecture) code of the kernel. Appendix B gives a short introduction to the assembler languages used. Listing 11.16 shows the IL DGEMM kernel with hardcoded $k = 1024$ and $\alpha = 1$ (α being the factor for the scalar multiplication).

```
1   il_ps_2_0
2   dcl_cb cb0[4]
3   dcl_input_position_interp(linear_noperspective) vWinCoord0.xy__
4   dcl_resource_id(0)_type(2d,unnorm)_fmtx(unknown)..._fmtw(unknown)
    ...
11  dcl_resource_id(7)_type(2d,unnorm)_fmtx(unknown)..._fmtw(unknown)
12  dcl_literal l0, 0x00000000, 0x00000000, 0x00000000, 0x00000000
13  dcl_literal l1, 1024.0, 1.0, -0.5, 2.0
14  mov r1, l0
    ...
45  mov r32, l0
46  mov r0.xy__, vWinCoord0.xy00
47  mov r0.w, l1.z
48  whileloop
49      add r0.w, r0.w, l1.y
50      ge r33.z, r0.w, l1.x
51      break_logicalnz r33.z
52      sample_resource(0)_sampler(0) r33, r0.yw
        ...
59      sample_resource(7)_sampler(7) r40, r0.xw
60      dmad r1.xy, r33.xy, r37.xy, r1.xy
61      dmad r1.zw, r33.xy, r37.zw, r1.zw
        ...
124     dmad r32.xy, r36.zw, r40.xy, r32.xy
125     dmad r32.zw, r36.zw, r40.zw, r32.zw
126 endloop
127 flr r0.xy, vWinCoord0.xy
128 ftoi r33.xy, r0.xy
129 imul r33.x, r33.x, cb0[2].z
130 imad r0, r33.yyyy, cb0[2].yyyy, r1.xxxx
131 iadd r33, r0, cb0[3]
132 iadd r34, r33, cb0[4]
133 mov g[r33.x], r1
    ...
140 mov g[r34.w], r8
    ...
193 end
```

Listing 11.16: DGEMM IL Kernel (8×8 Tiling, A transposed)

Eight input buffers are defined in lines 4 to 11: four buffers for the A-matrix and four buffers for the B-matrix. The target registers for the C-matrix are initialized to zero in lines 14 to 45. l1.x

is the hardcoded k value. Lines 50 and 51 represent the loop exit condition.[18] Assume the thread is calculating rows a to $a+7$ and columns b to $b+7$ of the target matrix in the k^{th} iteration. The registers r0.x and r0.y (abbreviated by r0.xy) store the target location in the C-matrix divided by eight (due to 8×8 blocking). Lines 52 to 59 read eight values from the A and the B-matrix each. Every instruction reads two doubles. The location inside the A input buffers, row a and column k, is stored in r0.yw. (Fig. 11.14 visualizes how the input data are stored in memory.) They are swapped as A is transposed and the row is divided by eight because each iteration reads two doubles from four buffers each (r0.y $= a/8$, r0.w $= k$). The same index can be used to access all four buffers. In the same way, eight doubles are read from the B-matrix. Lines 60 to 125 multiply each element read from A with each element read from B and add the results to the target registers. Lines 127 to 132 calculate the destination address for eight stores to the global buffer. Lines 133 to 140 store eight of the target registers.[19] This is repeated four times to store all 32 target registers.

Register r0 stores the loop iteration counter and the two-dimensional index of the C-matrix entry that is calculated. Registers r1 to r32 are used to accumulate the results. (64 entries require 32 *double2* registers.) Register r33 is used temporarily to store the result of the comparison in line 50. Registers r33 to r40 cache the data read from A and B. Finally, registers r33 and r34 are reused during index calculations for storing the results. At no time more than 41 registers are required.

Some tricks are needed to make the compiler stick with this lower bound of 41 registers, e. g. the copying of the vWinCoord0 register to r0 in line 46. The "right" code for 41 registers was found experimentally. Inspecting the corresponding ISA code shown in Listing 11.17 shows that the kernel in fact uses only 41 registers (R0 – R40).[20]

```
1   00 ALU: ADDR(64) CNT(125)
2         0  x: MOV          R9.x,   0.0f
3            y: MOV          R9.y,   0.0f
4            z: MOV          R9.z,   0.0f
5            w: MOV          R9.w,   0.0f
6            t: MOV          R0.w,  -0.5
7         1  x: MOV          R10.x,  0.0f
8            y: MOV          R10.y,  0.0f
9            z: MOV          R10.z,  0.0f
10           w: MOV          R10.w,  0.0f
11           t: MOV          R11.x,  0.0f
     ...............
132      25  x: MOV          R40.x,  0.0f
133          y: MOV          R40.y,  0.0f
134          z: MOV          R40.z,  0.0f
135          w: MOV          R40.w,  0.0f
136 02 LOOP_DX10 i0 FAIL_JUMP_ADDR(9)
137     03 ALU_BREAK: ADDR(193) CNT(2) KCACHE0(CB0:0-15)
138           26  w: ADD          R0.w,   R0.w,   1.0f
139           27  x: PREDGT       ____,   KC0[1].x,   R0.w
140     04 TEX: ADDR(560) CNT(8) VALID_PIX
141           28  SAMPLE R3, R0.yw0y, t0, s0  UNNORM(XYZW)
142           29  SAMPLE R4, R0.yw0y, t1, s1  UNNORM(XYZW)
     ...............
147           34  SAMPLE R6, R0.xw0x, t6, s6  UNNORM(XYZW)
148           35  SAMPLE R8, R0.xw0x, t7, s7  UNNORM(XYZW)
149     05 ALU: ADDR(195) CNT(124)
150           36  x: FMA_64       R12.x,  R3.y,   R1.y,   R12.y
```

[18] Line 50 can be replaced by "ge r33.z, r0.w, cb0[0].x" for dynamic k.
[19] In the example, α is fixed to 1.0. For dynamic α the lines must be preceded by "dmul r0.xy, r0.xy, cb0[0].zw", etc.
[20] The r123 register is used only as a fake register for the 64-bit FMA.

```
151                y: FMA_64       R12.y,   R3.y,   R1.y,   R12.y
152                z: FMA_64       R123.z,  R3.y,   R1.y,   R12.y
153                w: FMA_64       R123.w,  R3.x,   R1.x,   R12.x
154           37   x: FMA_64       R123.x,  R3.y,   R1.w,   R12.w
155                y: FMA_64       R123.y,  R3.y,   R1.w,   R12.w
156                z: FMA_64       R12.z,   R3.y,   R1.w,   R12.w
157                w: FMA_64       R12.w,   R3.x,   R1.z,   R12.z
    ...............
402           99   x: FMA_64       R123.x,  R7.w,   R8.w,   R37.w
403                y: FMA_64       R123.y,  R7.w,   R8.w,   R37.w
404                z: FMA_64       R37.z,   R7.w,   R8.w,   R37.w
405                w: FMA_64       R37.w,   R7.z,   R8.z,   R37.z
406 08 ENDLOOP i0 PASS_JUMP_ADDR(3)
407 09 ALU: ADDR(451) CNT(31) KCACHE0(CB0:0-15)
    ...............
418 10 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R0.x], R12, ELEM_SIZE(3)  VPM
419 11 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R1.x], R20, ELEM_SIZE(3)  VPM
420 12 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R2.x], R17, ELEM_SIZE(3)  VPM
421 13 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R3.x], R25, ELEM_SIZE(3)  VPM
422 14 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R4.x], R14, ELEM_SIZE(3)  VPM
    ...............
450 END_OF_PROGRAM
```

Listing 11.17: DGEMM ISA Kernel (Corresponding to IL Kernel in Listing 11.16)

11.2.4.4 Scheduling & GPU / CPU Performance Ratio

The goal of the CALDGEMM scheduler is to utilize both GPU and CPU to the full extent and hide all latencies. On the one hand, scheduling based on small tiles simplifies load balancing, while on the other hand, both the CPU and the GPU DGEMM work better on larger matrices. An optimum has to be found. First, recapitulate the situation: the matrix product $C = A \cdot B$ is calculated, where C is an $m \times n$, A an $m \times k$, and B a $k \times n$ matrix. The matrix C is divided in a GPU and a CPU part. The GPU part is further split in tiles of size $h \times h$. The dimensions of the GPU part can be assumed multiples of h as the CPU processes the borders.

As described in Section 11.2.3, the matrix is split in two parts, where one is processed by the GPU and one by the CPU. Obviously, the optimal splitting ratio is not a constant but can depend on the input parameters m, n, and k. A plausible assumption is that it does depend on the size of C but not on the shape, so it depends on $m \cdot n$ but not on m and n individually. Naturally, because of the tiling, this assumption holds for the GPU part of CALDGEMM. GPU DGEMM performance depends mostly on $m \cdot n$, a small dependency on $m + n$ is caused by the DMA transfer. It is now analyzed whether the same holds true for GotoBLAS.

Fig. 11.18 visualizes the ratio of GPU and CPU performance as a function of $m \cdot n$ using an assortment of runs with various m and n. Optimally, the plot should show a curve that goes asymptotically toward a constant for large $m \cdot n$ with only tiny fluctuations on the y-axis. Unfortunately, this is not the case. A deeper analysis unveils that this is related to a slowdown in GotoBLAS for certain input parameters. The green points in the figure correspond to runs with $n = 2048$, red points to runs with n close to 0 (mod 4096), and blue points to all other parameters. (The term (mod 4096) in brackets means the full (in)equation is read modulo 4096.)

In order to understand the behavior at $n \equiv 0 \pmod{4096}$, Fig. 11.19 shows the performance in a range of 2048 around $n = 40960$ (which is congruent 0 (mod 4096)). It clearly demonstrates that the slowdown only occurs in a small area around $n = 40960$.

Fig. 11.20 shows the region near $n = 40960$ in a higher resolution. In order to exclude the possibility that the slowdown is related to the thread count, two variants are shown; with similar

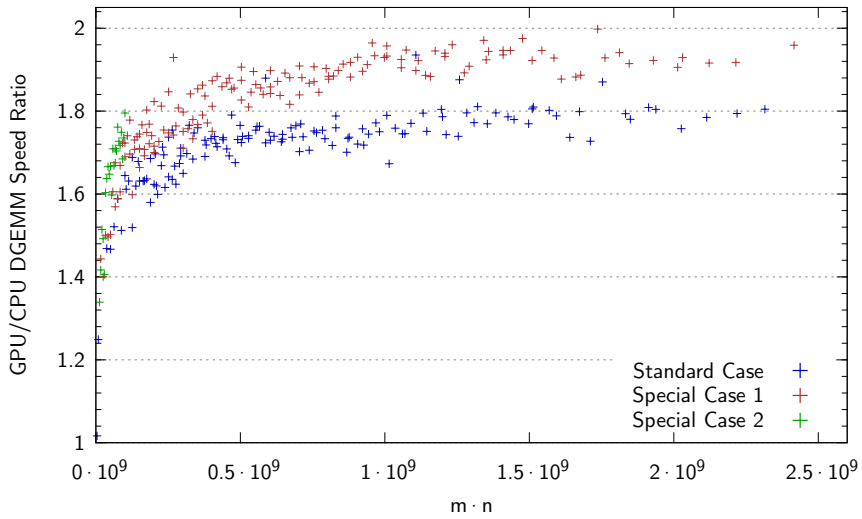


Figure 11.18: GPU / CPU DGEMM Performance Ratio [V]

behavior. Fig. 11.21 shows that there is no such dependence on the m parameter. As a conclusion there are two special cases, $n = 2048$ and $n \equiv 0 \pmod{4096}$, which have to be treated independently. Otherwise it can be assumed that the ratio depends only on $m \cdot n$.

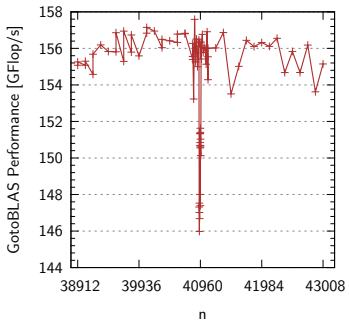


Figure 11.19: Performance of GotoBLAS depending on n (21 Threads) [V]

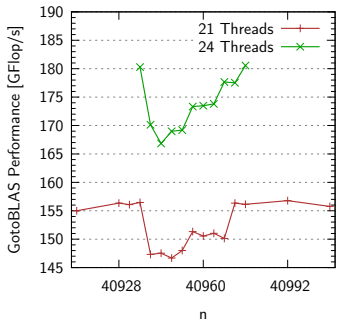


Figure 11.20: Performance of GotoBLAS near $n = 40960$ [V]

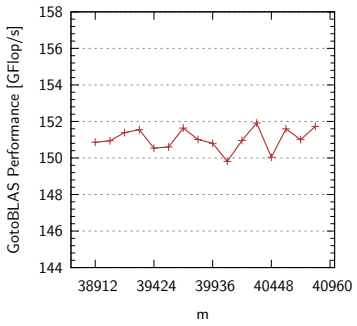


Figure 11.21: Performance of GotoBLAS depending on the Value of m [V]

Fig. 11.22 shows two fits of rational functions to the experimental ratio results. The special cases are excluded. Naturally, the data do not compose a smooth curve, but the fit is a simple method to estimate the ratio, and the analysis in Section 11.3.4.3 demonstrates that the approach works perfectly. In the data for the "separated GPU / CPU DGEMM" ratio, independent DGEMMs are executed on GPU and CPU respectively. However, finally they should run concurrently and are likely to influence each other. Therefore, a new data-set is created, executing a "combined GPU / CPU DGEMM" and measuring the GPU and CPU performance contribution to the combined DGEMM independently. In this second combined run, the scheduling is based on the data obtained by the previous separated runs. The ratio is slightly larger because the CPU DGEMM is slowed down due to GPU pre- and postprocessing while GPU performance is hardly affected. A new fit is done for the new data-set. The resulting curve is used to determine the ratio for CALDGEMM.

The two special cases ignored for the fit are handled the following way: If n lies in the problematic region, the ratio is first determined normally. Then, a correction for a 5 % slower CPU accounts for the slowdown. This is not entirely accurate, but these cases are rare anyway, rather GPU than CPU performance is overestimated (motivated in Section 11.2.3), and finally, advanced scheduling introduced in the next subsection corrects for small inaccuracies.

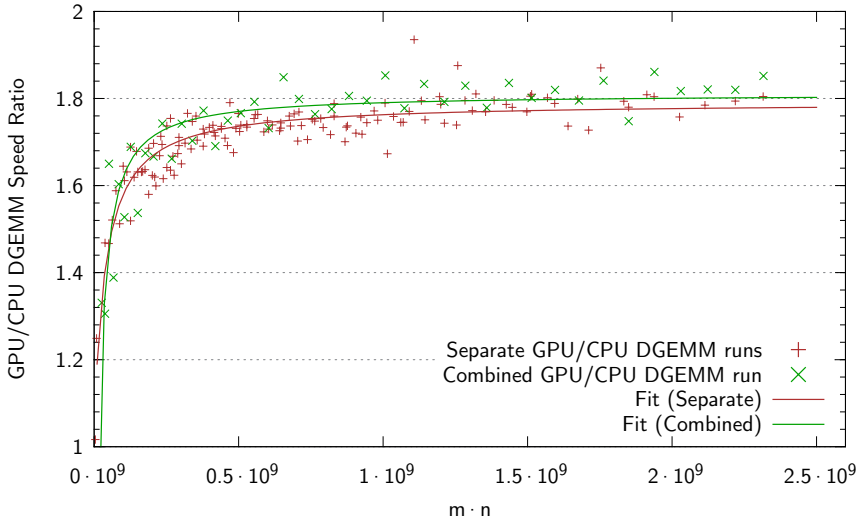


Figure 11.22: Fitted GPU / CPU DGEMM Performance Ratio [V]

One additional fact must be considered. The CPU also has to process the rightmost part of the matrix, as long as $n \not\equiv 0 \pmod{h}$. The size of the overlapping region depends on m and n. This is handled the following way. First, the CPU part is calculated as normal. It is then downsized to correct for the overlapping region and only then adjusted to be a multiple of h.

11.2.4.5 Second & Third Phase

Clearly, CPU and GPU DGEMM performance cannot be predicted well enough for the static scheduling to result in optimal performance. Additionally, due to tiling size constraints, the splitting position cannot be chosen arbitrarily but only in steps of h, which is 4096 for large matrices. Therefore, a more fine-granular scheduling is needed. It can be argued that GotoBLAS could process tiles the same way the GPU does. However, GotoBLAS shows an acceleration of 7.25 % when processing large blocks compared to the largest tile size.[21] Therefore, it is desirable to schedule the largest possible submatrices.

For the above reasons, the scheduler contains a second and a third phase. (The first two CPU DGEMMs, for the CPU part and the remainder of the C-matrix, are both considered phase one.) When the CPU has finished the first phase run, it checks how many GPU tiles are still unprocessed. It uses the same ratio as for the first phase to determine how many of these it should process. Then, it takes a rectangular unprocessed section all at once, which is as large as possible but contains at maximum this number of tiles. This is called **second phase run**.

Since CPU performance is regularly underestimated, rounding is always performed in favor of the GPU, and the rectangular CPU part during the second phase once again poses some restrictions, even with the two-phase scheduling the CPU sometimes idles for a short period at the end of the run. Therefore, in a **third phase** a work-stealing CPU scheduler takes tiles from the GPU part, as long as there are at least s tiles unprocessed. The optimal value of s on the LOEWE-CSC turns out to be three. This can be explained as follows: for processing a tile, the CPU requires about three times the time the GPU does; when the check is performed, the GPU still has to process 50 % of its current tile in average; and after the GPU has finished its last tile, the output must still be postprocessed by *MergeBuffer*, which takes roughly as long as the kernel execution. Fig. 11.23 shows an exemplary matrix distributed between the GPU and different CPU phases. (The second phase is not necessarily restricted to one row of tiles.)

[21] The improvement of 7.25 % is measured between square matrices with $m = n = 40960$ and $m = n = 4096$.

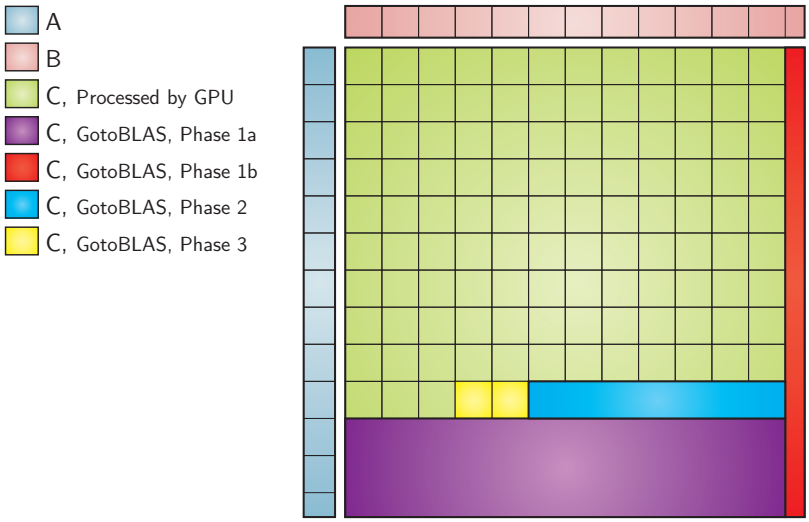


Figure 11.23: GPU / CPU Distribution of C-Matrix with three CALDGEMM Phases

11.2.4.6 Transfer Optimizations

Asynchronous Transfer Up until now, pre- and postprocessing, DMA transfer to the GPU, and kernels have been executed serially. Only the DMA transfer to the host is done in parallel to the computation using Zero-Copy and thus completely hidden (as described in Section 11.2.4.1). It is desired to parallelize these tasks as far as possible. In principle, the GPU DGEMM simply divides the part of the matrix to be calculated into several jobs. Each of these jobs consists of *DivideBuffer*, data transfer to the GPU, DGEMM kernel execution together with data transfer back to host memory, and *MergeBuffer*. The jobs are completely independent (except for GPU resource usage) and can be arranged in a pipeline. In the n^{th} iteration, the CPU preprocesses tile n. In the meantime, tile $n-1$ can be transferred to the GPU via DMA. Concurrently, the GPU can already process tile $n-2$. And again at the same time, another CPU thread can postprocess tile $n-3$. Principally, this can reduce the overall execution time to the pure GPU calculation time – together with some overhead. This overhead consists of the accumulated time for *DivideBuffer*, DMA transfer to GPU, and *MergeBuffer* – however, only for a single tile.

According to the CAL specifications, kernel execution and DMA transfer are completely unrelated and asynchronous. Table 11.24 shows measurements of a CAL kernel and a DMA transfer that take approximately the same time. Apparently, the execution time depends on the execution order when both are called in parallel. Starting the kernel first gives the sum of the single execution times while starting the DMA transfer first results in the maximum of the execution times. This means that with the current drivers, a DMA transfer cannot be started while a kernel is being executed. Still, it works the other way around.

Operation	Time [s]
Kernel Execution	0.060
DMA Transfer	0.050
Combined (Kernel started first)	0.110
Combined (DMA started first)	0.060

Table 11.24: Asynchronous CAL DMA Transfer [V]

Due to this behavior, the CPU must be two steps ahead of the GPU in the pipeline. DMA transfer for tile $n+1$ must start prior to the kernel call for tile n. Hence, the pipeline overhead

consists of the time for *DivideBuffer*, *MergeBuffer*, and twice the DMA transfer. The CPU first prepares only one tile, transfers it to the GPU, and calls the first kernel. While the first kernel is running, the CPU preprocesses tiles two and three. Transfer of tile two is postponed until the first kernel finishes. This is the point where the overhead of the second DMA transfer comes into play. An alternative is to prepare two tiles in the very beginning, but this is counterproductive since DMA transfer time is shorter than the *DivideBuffer* time (see Fig. 11.27).

As the runtime of the *MergeBuffer* routine exceeds the kernel execution time, or is at least of the same order of magnitude, two *MergeBuffer* threads are required to ensure continuous kernel execution. The threads postprocess the output tiles alternately. Let t be the number of output threads. Together with one thread for *DivideBuffer* this makes $t + 1$ threads to handle the GPU DGEMM.[22] Since the GPU writes directly to one output buffer in host memory as well, in total $t + 1$ output buffers are needed. They are used in a cyclic fashion.

BBuffers Recapitulate the way tiles are processed by the GPU: an inner horizontal loop and an outer vertical loop. First, the matrix A_i is transferred for processing $C_{i,1}$. The following tiles in the stream, $C_{i,2}$ to $C_{i,n/h}$ also require A_i, which should thus not be retransferred but simply remain on the GPU. Afterward, A_i is never used again and A_{i+1} is transferred. Now consider the B-matrix: when calculating $C_{1,1}$ to $C_{1,n/h}$, each B_j is used exactly once. As long as it is possible to store all the B_j-matrices in GPU memory, they never need to be retransferred again. The B_j-matrices are cached in so-called **BBuffers** on the GPU. CALDGEMM allocates two buffers for the A-matrix (to allow for concurrent DMA transfer and kernel execution of different tiles) and as many BBuffers as possible. (For instance, 21 BBuffers fit on the 5870.)

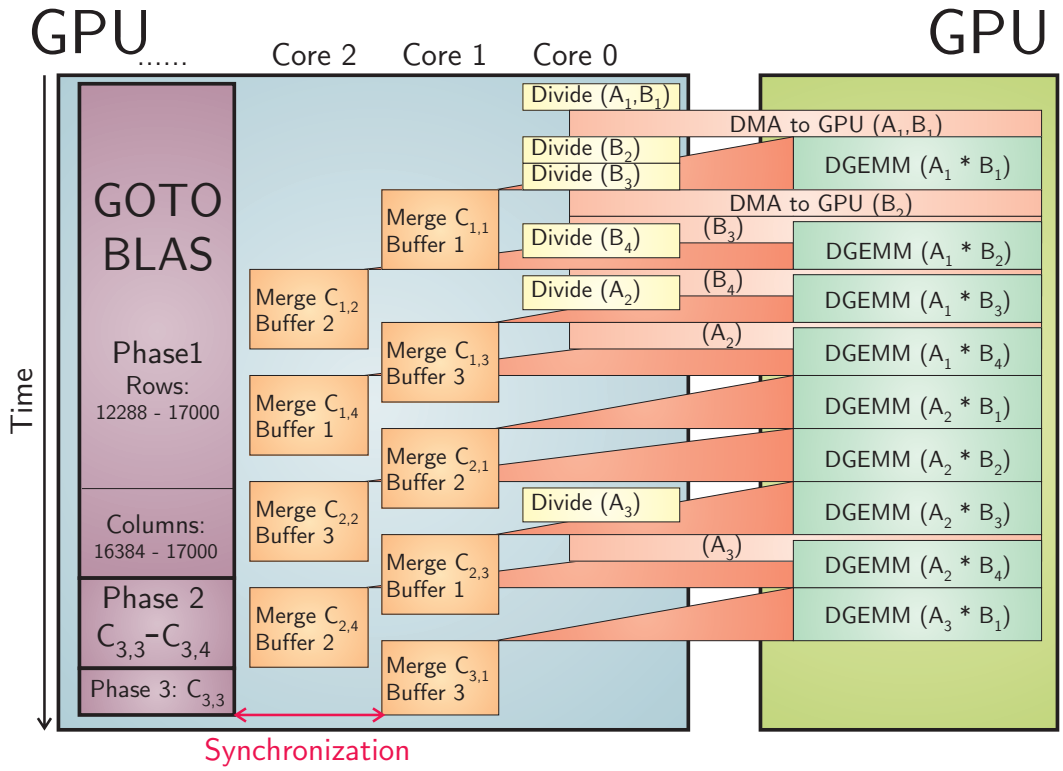


Figure 11.25: Process-Flow of improved CALDGEMM Implementation with Pipeline ($h = 4096$, $m = n = 17000$)

[22] Compare to Section 11.2.3.1, which introduces the t variable. All GPU threads are pinned to cores on die 0.

If n does not equal m, the matrices A and B require a different amount of memory. Swapping the inner and the outer loop of the tiling process swaps the roles of A and B in the above consideration. So the buffers in the GPU memory only need to store the smaller matrix. Hence, the BBuffers are sufficient as long as $\min(m, n) \leqslant 21 \cdot 4096$. It has to be noted that the CPU also processes a part of the matrix. This can be chosen in a way that lowers the smaller dimension – if otherwise the buffer size was insufficient. Finally, C only exceeds the buffer size on the LOEWE-CSC if it is at least of dimension 129024×129025 (which requires 124 GB of main memory).

Altogether, the BBuffers ensure that on the LOEWE-CSC both A- and B-matrix are transferred and preprocessed exactly once.

Fig. 11.25 visualizes the implementation while Fig. 11.26 shows the effect of both optimizations – individually and in combination. Since the overall performance dependencies on m and n are alike, this and the following plots' measurements are based on square matrices. It can be seen that both attempts provide a considerable performance benefit. The combination, however, does not result in another huge leap in performance because both optimizations approach the same inefficiency (the GPU waiting for input data from the host). Still, especially for small matrices, the combined implementation delivers definitely the best performance.

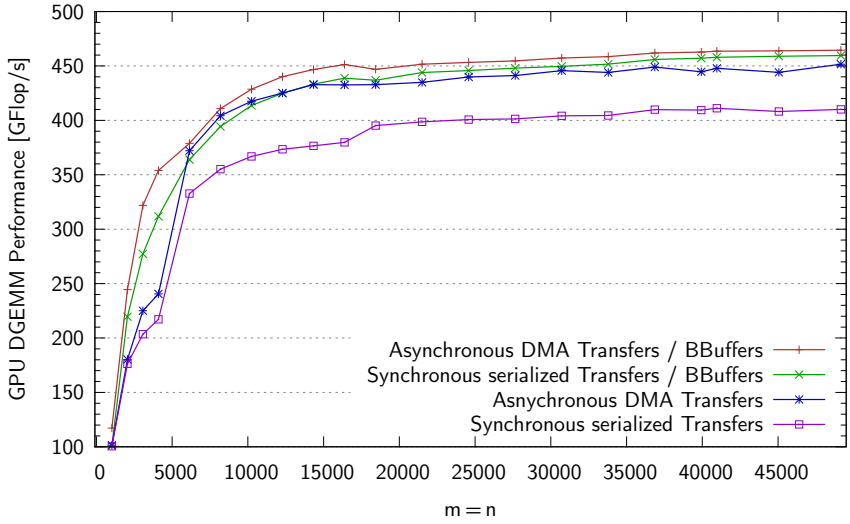


Figure 11.26: GPU DGEMM Performance for Asynchronous Transfer and BBuffers[23] [V]

***DivideBuffer* & DMA** The GPU kernel can write its result directly to host memory via Zero-Copy. In the same way, the *DivideBuffer* routine could write its output directly to the GPU memory via Zero-Copy DMA. This would make the subsequent large DMA transfer unnecessary. However, Fig. 11.27 shows that a *DivideBuffer* version writing directly to GPU memory takes more time than the usual *DivideBuffer* and the DMA transfer together. The reason lies in the increased latency of the PCI Express DMA access compared to the host memory. This problem is amplified since *DivideBuffer* does not write one but multiple streams in parallel, one for each output buffer. The approach was therefore discarded.

Up until now, different schemes for buffer types and locations as well as DMA transfer scenarios have been discussed. Certain extensions will follow throughout this thesis. A detailed overview of all DMA paths which finally turn out to be relevant is given in Section 12.12. The discussion is postponed until then in order to not repeat the extensive diagrams.

[23] Since *MergeBuffer* execution takes even longer than kernel execution, all versions use two *MergeBuffer* threads.

11.2.5 Vectorization & Patched AMD Driver

Fig. 11.27 shows the throughput of the DMA transfer and the pre- and postprocessing routines.[24] Consider the DMA bandwidth at first. Benchmarks show a huge dependence on the employed system BIOS version. With earlier versions, the throughput was only about 3 GB/s. With the new version, the transfer to the GPU is still inferior to the transfer to the host. This is unfortunate since only the transfer to the GPU is used in practice (as the kernel writes directly to the host). However, considering the matrix sizes appearing during HPL and remembering that all matrices A, B, and C are transferred exactly once, the transfer to the GPU appears less relevant. For instance, for $k = 1024$, $m = n = 81920$ the data transferred to the host is 40 times the data transferred to the GPU (50 GB versus 1.25 GB).

All versions of the *DivideBuffer* and *MergeBuffer* routines are vectorized. The author would like to thank Matthias Kretz for his help with *DivideBuffer*. Prefetches[26] are used and also streaming stores[27] are employed where they offer a benefit. The routines optionally use the *prefetchw* instruction, which is available on AMD CPUs.

It is obvious that the *DivideBuffer* routine is much slower when it has to transpose the input matrix. Section 11.2.5.1 will answer the question whether this is acceptable.

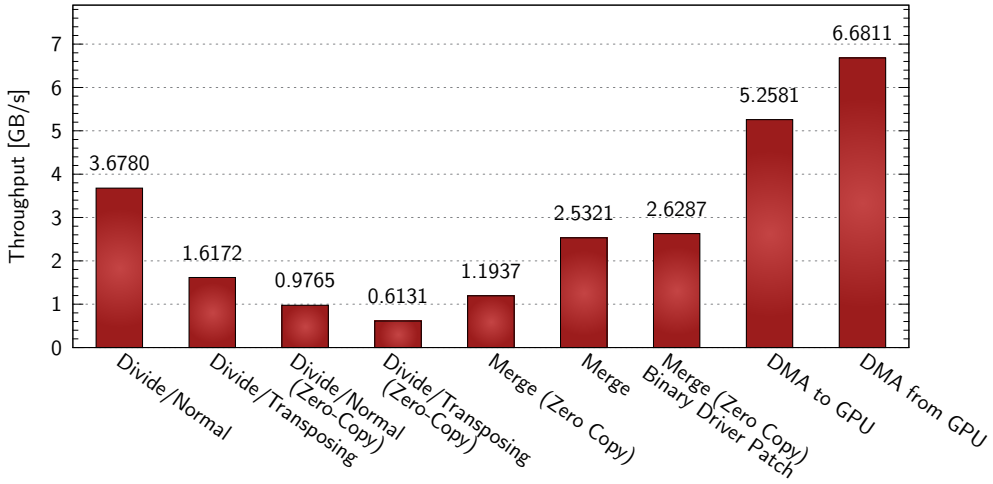


Figure 11.27: Performance of Pre-/Postprocessing and DMA Transfer[24,25] [V]

The *MergeBuffer* results need some explanation. At first, consider only the results without the binary driver patch (introduced below). It turns out that the *MergeBuffer* routine is faster when the data is transferred in one large DMA transfer than when the kernel writes directly to the host memory. The reason is the following: In order to enable Zero-Copy access for the GPU kernel, the memory region must be unmapped such that it has no virtual address. Naturally, the memory must be mapped again in order to provide access for user space routines such as *MergeBuffer*. Thereafter, *MergeBuffer* encounters a page fault for every page it accesses. To make things even

[24] The best performing kernel writes directly to host memory via Zero-Copy. Thus, the calculation time and the DMA transfer time cannot be distinguished. This makes the DMA throughput impossible to measure. Hence, for this measurement a kernel is used that stores its output in GPU memory. The throughput of a subsequent DMA transfer of this output to the host is shown.

[25] For *DivideBuffer* results marked by Zero-Copy, the routine writes directly to GPU memory instead of CPU memory. In contrast, *MergeBuffer* always reads from host memory. *MergeBuffer* results marked as Zero-Copy refer to situations where the result is transferred to the host by the kernel via Zero-Copy, otherwise the transfer is performed by the DMA engine.

[26] Prefetches explicitly load data from memory into the cache prior to their actual usage to avoid memory latencies later.

[27] A streaming store bypasses the cache and writes directly to the memory. Another scenario where streaming stores bring a huge benefit is presented in Section 17.3.8.

worse, every page is read exactly once. (Issuing a second *MergeBuffer* call on the same buffer, without un- and remapping, yields the same performance as in the non Zero-Copy case.)

Unmapping memory regions prior to kernel execution is technically not necessary but it is enforced by the AMD driver. A **binary patch** to the AMD driver has been created which removes this check making the Zero-Copy version work perfectly.[28] In addition to the higher throughput, the system load of the CPU is lowered significantly without the page faults. This is analyzed in more detail in Section 11.3.4.3.

Performing the Binary Driver Patch Disassembling the driver's shared object files reveals that the *calCtxRunProgram* API function calls a function, which checks the state of the buffer. Its return value stored in the *RAX* register is checked. If the correct value is not found, a conditional jump (*JE* – jump if equal) is not performed and the kernel is not executed. Replacing the *JE* instruction (with opcode 0x74) by a *JS* (short jump) instruction (with opcode 0xEB) of the same opcode size eliminates the check.

Global Buffer Review Recall that the global buffer is required for MemExport. At the moment, MemExport is not required in CALDGEMM but it might become relevant in the future. The current kernel with 4×4 blocking is coming to its very limit (due to the texture cache bandwidth limit discussed in Section 11.2.4.1). For any further performance increase, a bigger blocking is required. However, in that case MemExport is mandatory and thus also the global buffer.

The problem is: only a single global buffer is usable at a time. Currently, CALDGEMM requires $t + 1$ output buffers for t output threads. As only one single global buffer is available, the output buffers must all reside in this single big global buffer. *MergeBuffer* requires the buffer to be mapped whereas a kernel, without the binary driver patch, would not start if the buffer is mapped. This makes the binary driver patch mandatory for using both MemExport and the asynchronous output processing at the same time.

11.2.5.1 Miscellaneous Optimizations

Tiling Size Section 11.2.4.1 analyzed the influence of the matrix size on kernel performance. It was concluded that $h = 512$ can be used for matrices with $n < 1024$ or $m < 1024$. Otherwise, all 1024, 2048, 3072 and 4096 yield good results. However, up until now, only kernel performance has been taken into account. Now the effect on overall performance shall be discussed. Naturally, on the one hand, a larger h reduces the synchronization overhead. On the other hand, it increases the overhead before the first and after the last kernel call and it leaves less freedom to choose the splitting ratio. However, these negative effects should become less significant for large m and n. Therefore, the best tiling size is expected to depend on m and n. It can be shown that it depends only on $m \cdot n$ so again only $m = n$ is considered. Fig. 11.28 shows the performance relation between h and $m = n$. As expected, bigger matrices favor bigger tilings. The tiling size is thus auto-adjusted to be optimal for each input matrix by heuristics deduced from the figure.

One issue with the automatic tiling size selection shall be discussed. The CAL API does not support transferring only a part of a buffer to the GPU. As the allocation takes comparably long, the buffers are created during library initialization. This predetermines the buffer sizes. Since the BBuffers multiply the number of input buffers, it is not possible to allocate distinct buffers for different tiling sizes. Instead, only buffers for the maximum tiling size are allocated. For smaller h, parts of these buffers are used. Still, the entire buffer must be transferred. This is a huge overhead, especially for a small tiling. However, there is no other possibility to solve the issue and benchmarks show that the performance increases regardless of this inefficiency.

[28] AMD is aware of the problem, and for OpenCL kernels the corresponding check is already removed in the driver.

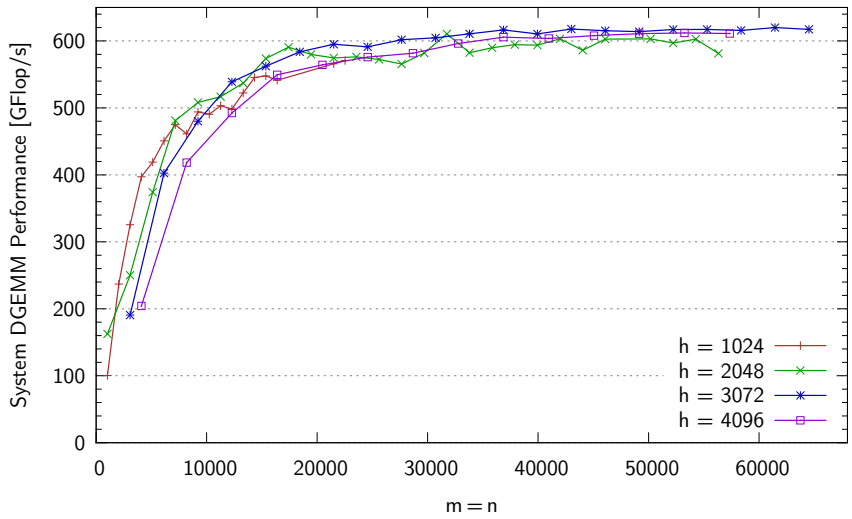


Figure 11.28: Performance for different CALDGEMM Tiling Sizes [V]

Transposed Matrices Up until now, the correct input format has not been respected. Both the possibility to transpose a matrix in the kernel and to perform the transposition during *DivideBuffer* have been discussed. It was concluded that for the kernel a transposed B-matrix is optimal. Consider the usage of the transposed B kernel for a transposed A-matrix. Both matrices have to be transposed by *DivideBuffer*, which is the worst case for this function. Alternatively, the kernel for the transposed A-matrix can be used, which is the best case for *DivideBuffer*. Table 11.29 compares the 4 possible combinations of kernel and input type.

Kernel	A Input Transposed	B Input Transposed
Kernel for A Transposed	600.1 GFlop/s	597.7 GFlop/s
Kernel for B Transposed	619.3 GFlop/s	618.0 GFlop/s

Table 11.29: Combined CPU / GPU DGEMM Performance for Transposed Input Matrices [V]

It turns out that the faster kernel when B is transposed outweighs the penalty for the transposition during *DivideBuffer*: the B transposed kernel is faster in both cases and is thus used exclusively. Interestingly, the transposed A-matrix is processed even faster than the transposed B-matrix. The explanation lies in the GotoBLAS library, which is slightly faster when A is transposed.

For very small matrices, the *DivideBuffer* function consumes a more significant amount of time. In this case, the overhead for the transposition weights more and the A transposed kernel becomes the faster one. However, the data format of the buffers must be declared at allocation time (compare to the previous paragraph) but it differs for both kernel variants. Thus, it is impossible to allocate buffers for both kernels at the same time. Hence, the B transposed kernel is used regardless of input format and matrix size.[29]

Huge Pages CALDGEMM has been tested with huge pages (see Appendix C.3 for details). Unfortunately, they do not always bring a benefit. It turns out that it is related to the employed hardware and the operating system whether the performance gains or suffers. On the LOEWE-CSC nodes (and similar systems) no improvement could be observed. However, on certain systems dealt with in the next chapter, performance does improve indeed. For instance, the non-NUMA SDS system in Section 12.7.3 shows good results with huge pages. Although NUMA might have an influence on huge page performance, it is not true that the AMD Magny-Cours NUMA systems

[29] CALDGEMM can still switch to the A transposed variant, but not at runtime.

generally show worse performance with huge pages (as is shown in Section 12.8.4). In the following, huge pages are used only where stated explicitly. On the LOEWE-CSC the feature is disabled.

11.2.6 Summary & Results

In summary, CALDGEMM is implemented the following way:

- The B transposed kernel is always used, regardless of whether a matrix (or which one) must be transposed or not.
- The optimal tiling size is selected automatically depending on the input matrix size. Possible values are 512, 1024, 2048, 3072, and 4096.
- The splitting ratio between GPU and CPU is chosen automatically based on the matrix size.
- A pipeline, asynchronous DMA transfers, and BBuffers on the GPU ensure continuous DGEMM kernel execution and minimize data transfer.
- GotoBLAS is patched to allow for core reservation. Second and third phase GotoBLAS runs ensure that both GPU and CPU are always fully utilized and do not idle.
- A patch to the AMD driver minimizes page faults and is needed to achieve full performance.
- Since the CPU can handle the borders, GPU matrix size can be subject to restrictions.

Finally, CALDGEMM can deliver a pure DGEMM performance of 625 $^{\mathrm{GFlop}}/_{\mathrm{s}}$ on a LOEWE-CSC node. DGEMM performance depends only marginally on whether matrices are transposed or not. It depends slightly on the input parameters but remains high as long as the matrices do not get too small. Fig. 11.30 gives an overview of the final CALDGEMM performance. CPU and GPU utilizations during a run are analyzed in detail in Section 12.6.1.1. Appendix G lists the CALDGEMM features and recommended settings for various hardware platforms.

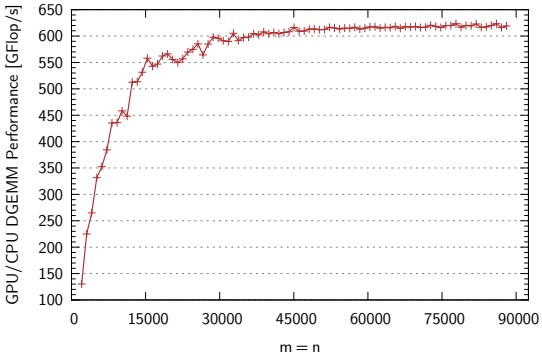


Figure 11.30: Overview of CALDGEMM Performance [V]

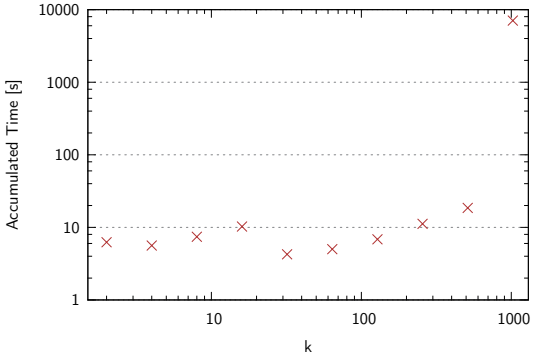


Figure 11.31: Time Consumption of DGEMM Runs with varying k during HPL (16 Nodes)

11.3 GPU-based HPL

11.3.1 Integrating CALDGEMM

The previous sections described the development of a fast GPU aware DGEMM implementation. The HPL benchmark has been modified to make use of the CALDGEMM library. The adapted HPL version called HPL-GPU is available as open source (see Appendix I). In the following, the term HPL always refers to HPL-GPU if not stated differently. CALDGEMM provides only

DGEMM but no full BLAS capability and it is based on GotoBLAS itself. Therefore, the combination of HPL, CALDGEMM, and GotoBLAS is used. CALDGEMM itself is multi-threaded and involves multi-threaded GotoBLAS calls. Thus, the common HPL approach of multiple MPI[30] processes per node, one per core, is not suited. Lots of problems would arise, e. g. which process shall use the GPU, how to handle processes with different DGEMM performances, etc.

The solution is a single MPI process per node, which is multi-threaded itself. HPL uses pthreads and TBB[31]. As a consequence, other tasks apart from BLAS have to be parallelized. DLACPY[32] is accelerated by trivial parallelization, SSE loads and stores. Additionally, Matthias Kretz implemented parallelized and vectorized versions of DLATCPY[32] and all relevant LASWPs.[32]

Some initial benchmarks revealed that $N_b = 1024$ is an appropriate value for HPL-GPU. Lowering N_b reduces the GPU DGEMM speed. A larger N_b dramatically increases the time required for factorization (the complexity is $\mathcal{O}(N \cdot N_b^2)$) while the DGEMM performance suffers as well.

Throughout an HPL run there are lots of DGEMM calls in the factorization (with $k < N_b = 1024$) and one huge DGEMM call with $k = N_b = 1024$ outside the factorization. At first, some statistics about the DGEMM parameters and the time distribution of the DGEMMs are collected. As expected, Fig. 11.31 shows that the calls during the factorization do not have a significant contribution to the overall time. Hence, only the large DGEMM for the C-matrix update is offloaded to the GPU. Later, execution time of all other HPL tasks is hidden as far as possible to keep the GPU executing DGEMM kernels all the time.

The first HPL-GPU version is presented in Fig. 11.32. Factorization, LASWP and DTRSM are multi-threaded. The DGEMM is multi-threaded as well and in addition uses the GPU. The broadcasts of the panel and the U-matrix are not multi-threaded. All these tasks are executed one after another. Hence, the performance is limited according to Amdahl's law [Amd 67].

11.3.2 Optimizing HPL

Naturally, the first implementation leaves much room for improvement. At first, the given HPL parameters, affecting mostly broadcast and recursive factorization, are tuned to their best. The factorization parameters (see [UoT]) were determined by parameter range scanning to:

- NBMin = 64
- NBDiv = 2
- Panel factorization: Crout oriented
- Recursive factorization: Left oriented

11.3.2.1 Alignment

HPL usually aligns data to eight doubles, which is the cache line size. However, entries in the same column but different rows (or vice versa in column-major representation) are likely to have the same cache tag, so when accessing entries within a single column, only a tiny fraction of the cache is used. The stride between the rows is thus altered such that consecutive rows start in addresses with a cache tag difference coprime to the cache tag index range. This change in the leading dimension[33] of the HPL-matrix guarantees that the full cache is used and in particular speeds up LASWP. In single-node runs, the U-matrix is contained within the matrix that is factorized. In multi-node runs, not every node possesses the part of the large matrix A where U is stored. In this case, the U-matrix is stored externally and the same alignment correction is applied to it.

[30] HPL uses the **M**essage **P**assing **I**nterface (MPI) library for data transfer between processes / nodes.
[31] Intel **T**hreading **B**uilding **B**locks [Int].
[32] DLACPY, DLATCPY and LASWP are functions of the BLAS and LAPACK libraries. They copy and transpose matrices or swap lines or columns. HPL comes with its own single-threaded version of these functions.
[33] Refer to Appendix C.2 for an explanation of leading dimension.

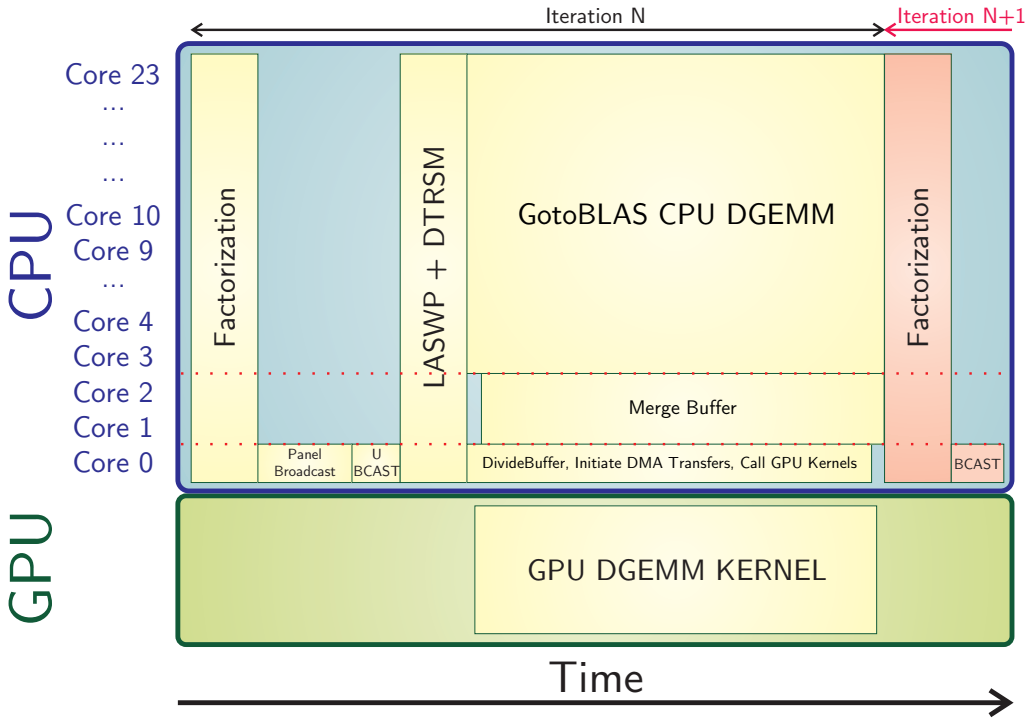


Figure 11.32: Process-Flow of GPU-based HPL[34]

11.3.3 Multi-Node HPL

The way how CALDGEMM is integrated allows for multi-node HPL-GPU runs without further modifications. In the same way as for the original HPL, the matrix is distributed among a grid of $p \cdot q$ processes. (Details can be found in Section 12.4.) Still, plenty of tuning can be applied. Again, the available HPL parameters [UoT] are tuned first. HPL offers a parameter called **lookahead** which is intended to hide communication latencies and shorten the critical factorization path. In addition, HPL offers multiple network transfer modes. Fig. 11.35 shows that the long transfer mode is superior to the ring transfer mode (see [UoT]).[35,36] It also shows that the original lookahead with depth one as implemented in HPL has a negative effect. (Still, with the lookahead the performance of the ring and the long transfer mode are equal. This shows that the lookahead successfully hides the communication time. The bottle-neck is rather that lookahead is not optimized for the chosen approach with large N_b and only one MPI process per node. J. Kurzak [Kur+ 12] shows that in principle, lookahead works for GPU-accelerated HPL with small N_b and the standard MPI process approach.)

The RDMA feature of InfiniBand accelerates HPL by 2.3 %.[37] It further turns out that copying the L-matrix to a consecutive memory segment before the transfer brings a performance boost.

Another important parameter is the matrix-size. Obviously, for best performance the matrix should be as large as possible. However, for stability reasons or short test, it might be interesting to reduce the matrix size. Fig. 11.33 shows the performance in relation to the matrix size per

[34] The time axis is not to scale.
[35] The long mode is usually used for fast nodes and the ring mode for a fast network. At the LOEWE-CSC both the nodes and the network are very fast so it is not immediately clear which algorithm to use.
[36] With an increased number of nodes, the modified long mode performs even better.
[37] Unfortunately, at the moment there are unsolved problems that lead to errors when using RDMA and PCI Express DMA transfer to the GPU at the same time. These instabilities occur only when multiple hundreds of nodes participate. The following exemplary benchmarks are mostly done on four nodes ([V]) with RDMA enabled. However, for large runs on the cluster, RDMA has to be disabled for the time being.

node in a $p \times q = 2 \times 2$ grid-configuration. It demonstrates that down to 45 GB the performance hardly suffers.[38] The shape of the process grid (see Section 12.4 and [UoT] for details) also plays an important role (Fig. 11.34). The number of process rows and columns should have a three to two ratio or vice versa. If such a ratio is not possible due to a fixed finite number of nodes, every grid shape similar to a square is acceptable, too. In contrast to HPL-GPU, the unmodified HPL ([Don$^+$ 03, UoT]) favors a flat grid ($p < q$) because its lookahead hides the panel-broadcast time but not the U-broadcast time.

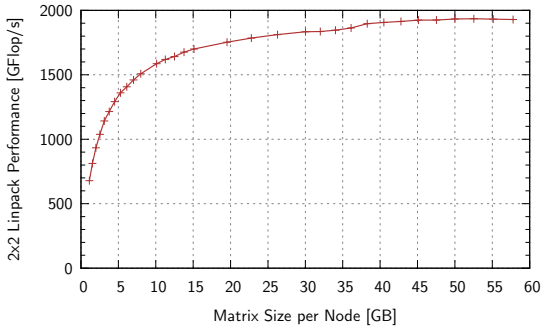


Figure 11.33: Linpack Performance Dependency on Matrix Size[38] [IX]

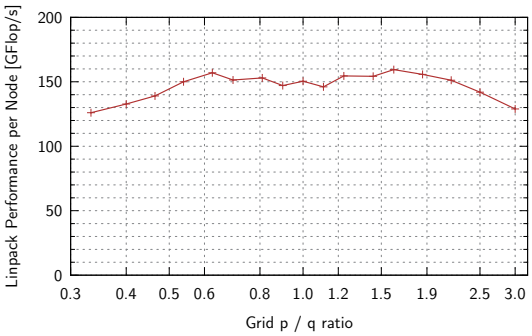


Figure 11.34: Linpack Performance Dependency on Process Grid Shape[38] [IX]

11.3.4 Lookahead

The initial HPL benchmarks without GPU revealed that the DGEMM contributes 96.59 % to the overall HPL time (see Table 10.4). The adoption of CALDGEMM decreases this portion significantly. Thus, optimizing other steps, such as broadcast, factorization, or pivoting becomes more and more important. Up until now, these steps are executed one after the other, resulting in a high GPU idle time. Fig. 11.36 shows the contribution of all relevant steps of HPL-GPU. According to Amdahl's law ([Amd 67]), the HPL performance is limited to about 0.79 times the DGEMM performance. This encourages trying to optimize the sequential parts.

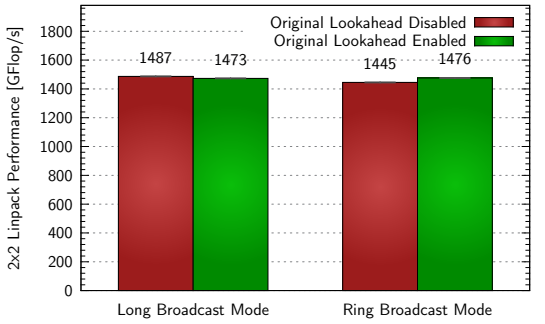


Figure 11.35: Influence of HPL Parameters on Performance [V]

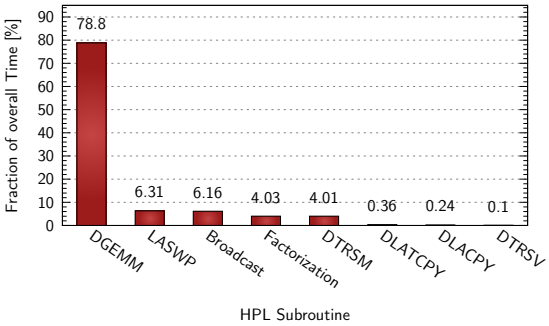


Figure 11.36: Time Consumption of HPL Subroutines [V]

As the original lookahead algorithm seemed useless and since its design does not take into account the GPU, a completely new lookahead has been developed from scratch. The aim is to hide the factorization and the broadcast, and later even the swaps and DTRSM, to keep the GPU executing DGEMM kernels 100 % of the time. In the meantime, Intel has assimilated the new lookahead into their HPL for the Xeon Phi [Hei$^+$ 13]. The lookahead is implemented in multiple steps.

[38] The plots are created with results from the final HPL-GPU version presented at the end of this chapter. However, thematically they belong here. Fig. 11.34 shows results for small matrices to amplify the influence of communication.

11.3.4.1 Lookahead 1

In a first step, lookahead hides the panel broadcast and the factorization, on which the panel broadcast depends. The U-broadcast, DTRSM, and LASWP are ignored for the time being.

Looking at the algorithm reveals that the factorization requires the DGEMM of the previous iteration to have finished only the first N_b columns. Thus, it is possible to start the next factorization iteration while the DGEMM is still running. After the factorization, the panel can be broadcasted in parallel to the DGEMM computation, too. The dimension $N_b = 1024$ is a rather small size for a DGEMM call: tile size would be 1024 only and there would be only one column of tiles. Thus, the first N_b columns are excluded from the GPU DGEMM and handled by the CPU before the CPU even starts its actual part of the large DGEMM (according to the GPU / CPU splitting). Between the two DGEMMs, the CPU performs panel factorization and broadcast. This also makes the synchronization easier compared to the scenario where the GPU processes the first N_b columns. (Section 14.2.2 reevaluates this strategy for multi-GPU systems.)

A "Linpack-Mode" is introduced in CALDGEMM, in which it supports callback functions for the factorization and the broadcast. The first lookahead implementation is presented in Fig. 11.37.

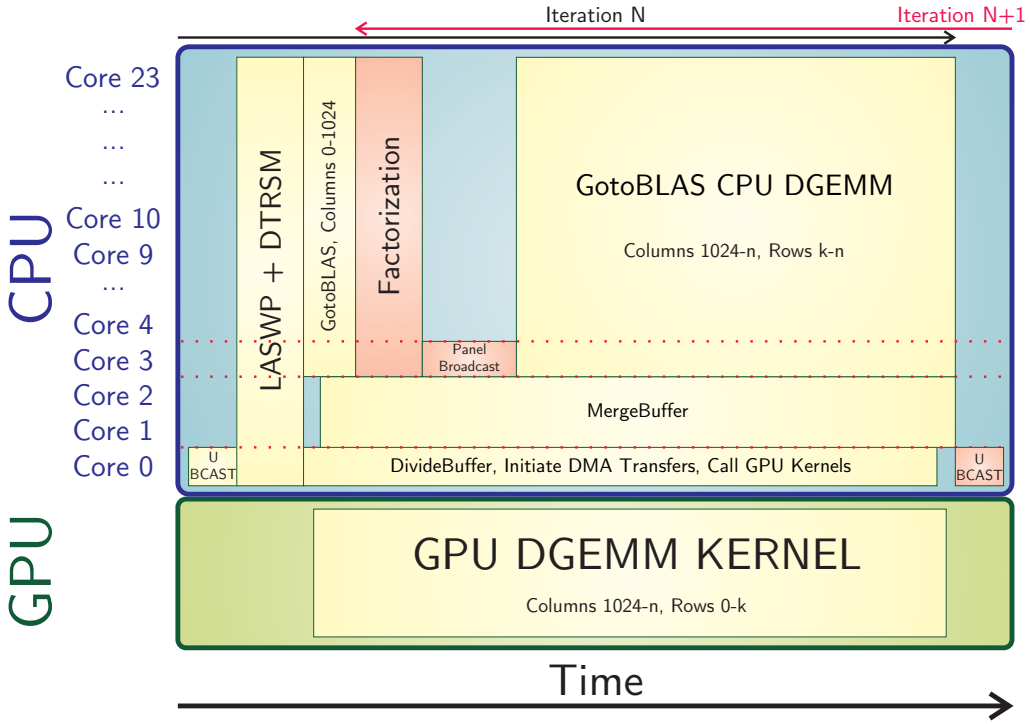


Figure 11.37: Process-Flow of GPU-based HPL with Lookahead 1 (Initial Version)

The additional tasks of varying duration make a static load distribution between CPU and GPU impossible. Therefore, CALDGEMM continuously measures CPU and GPU performance as well as the time required for transfer and factorization. Based on this, it continuously adjusts the GPU / CPU splitting ratio. With r the old ratio, g the GPU performance, c the CPU performance, t_g the GPU DGEMM time, and t_c the pure CPU DGEMM time (not including factorization and transfer), the new ratio r' is calculated as $r' = \frac{1}{2} \cdot (r + \frac{g}{c'+g})$ with $c' = {}^{ct_c}/_{t_g}$. In a multi-node run, not every node is involved in every factorization iteration. This depends on its position inside the grid. CALDGEMM respects the grid position and maintains ratio parameters for usage with and without factorization. It continuously rechecks whether the applied ratio has been appropriate. If that is not the case, it reinitializes the ratio using the static curve originally

used in CALDGEMM. For the (re)initialization, an empirically chosen penalty is applied to the CPU performance to correct for factorization and broadcast time.

Maximizing CPU Utilization During the broadcast many CPU cores are inactive. This can be improved by starting an additional DGEMM that utilizes the remaining cores. There are two ways to implement this:

- One CPU core is reserved exclusively for the broadcast. It is idling afterward. The DGEMM can start immediately and processes the entire matrix.
- Two DGEMMs are executed serially. The first one uses all but the communicating core and the GPU cores. Its matrix size is chosen such that according to the performance estimations, it should finish shortly after the broadcast. Afterward, the second DGEMM uses all available cores.

HPL-GPU now uses the second variant, which results in 2090 $^{\text{GFlop}}/_{\text{s}}$ versus 2075 $^{\text{GFlop}}/_{\text{s}}$ for the first version (measured with four nodes in a 2×2 grid).

The estimation for the matrix size of the first DGEMM is based on continuous measurements of broadcast time and CPU DGEMM performance. The reduced DGEMM thread count is respected. The matrix size is chosen slightly bigger than necessary to ensure that in any case the broadcast finishes first. If the estimation concludes that the first call would already process most of the matrix, CALDGEMM does not split the matrix and uses the first variant.

In case the predicted broadcast time has been too short nonetheless, the first DGEMM finishes before the broadcast. As it makes no sense to wait for the broadcast to finish, the second DGEMM is then started without the additional core.

Maximizing CALDGEMM GPU Performance Unfortunately, benchmarks reveal that the lookahead mode – in the version just explained – is inferior to the version without lookahead. It turned out that this is related to a decreased GPU DGEMM performance. The GPU contribution to the DGEMM drops from about 450 $^{\text{GFlop}}/_{\text{s}}$ to only 400 $^{\text{GFlop}}/_{\text{s}}$. However, this is the effect on the overall GPU DGEMM performance. The factorization and the broadcast do not last for the entire DGEMM. As long as they run, they reduce the GPU DGEMM performance even further: to about 300 $^{\text{GFlop}}/_{\text{s}}$. (After they have finished, performance comes back to 450 $^{\text{GFlop}}/_{\text{s}}$ resulting in an average of 400 $^{\text{GFlop}}/_{\text{s}}$.)

The cause is a strongly reduced performance of the *MergeBuffer* routine due to memory congestion. It was attempted to cope with this with an additional merge thread, i. e. by incrementing t from 2 to 3 during factorization and broadcast. The additional thread is only active during factorization and broadcast. Fig. 11.38 shows a process-flow diagram.

Sadly, this approach does not improve the overall performance at all. As expected, the GPU DGEMM performance returns to normal. However, the combined factorization and broadcast time increases by an order of magnitude and even exceeds the DGEMM time. The problem just moves to another spot.

Lowering the Memory Load with the AMD Driver Patch It shall be noted here that the above tests had been made before the binary driver patch was performed. Now, the binary driver patch allows for a reduction of the output thread count to a single thread while still achieving about 620 $^{\text{GFlop}}/_{\text{s}}$ of combined CPU / GPU DGEMM performance. (See Section 11.3.4.3 for an analysis of the CPU utilization with and without the driver patch.) The driver patch lowers, in particular, the memory load. Reverting to the previous lookahead version without the additional merge thread and using the driver patch with only one single output thread ($t = 1$) helps, but does not solve the problem completely.

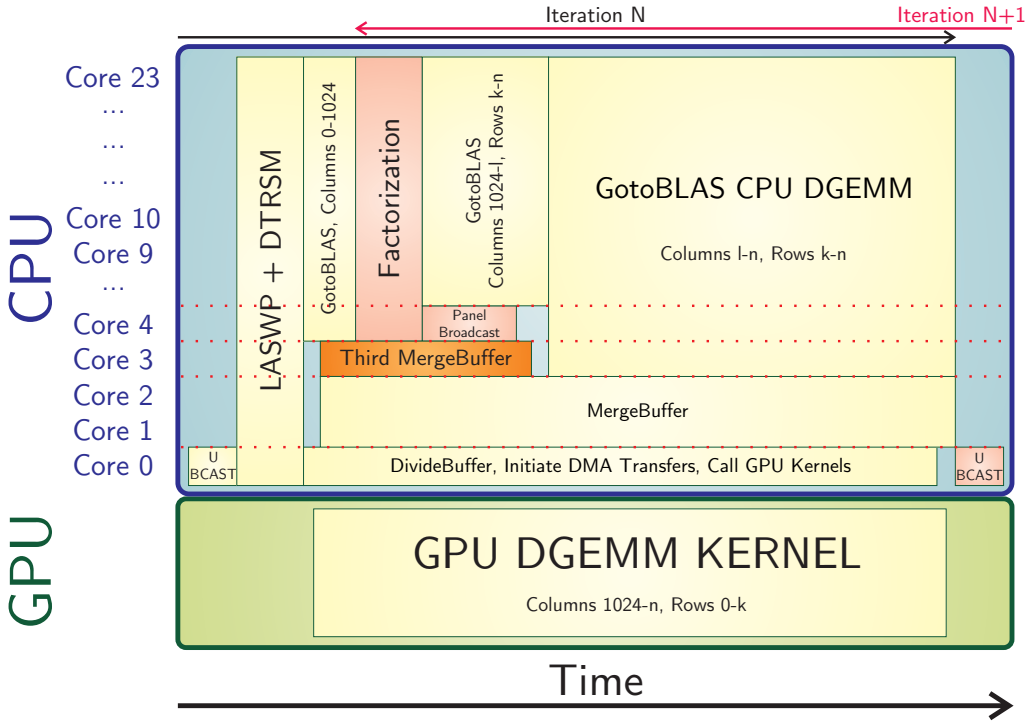


Figure 11.38: Process-Flow of GPU-based HPL with Lookahead 1 (Three Output Threads)

Further Reduction of the Memory Congestion Full GPU performance is finally achieved by, in addition to the binary driver patch, reducing the factorization thread count to eight. Actually, the factorization performance does not even suffer much because several memory-bound tasks do not profit from more than eight cores. The very first factorization iteration, which does not run concurrently to a DGEMM, still uses all available cores.

The lookahead performance with reduced factorization thread count is analyzed in more detail. Since the binary driver patch is not generally available (e. g. for new driver versions), the implications of a reduced factorization thread count are evaluated with and without the patch.

Fig. 11.39 shows the performance with and without lookahead throughout an HPL run. The x-axis shows the iteration number in HPL, so on the right side of the diagram, the matrix size decreases. The axis is not proportional to the time as the first iterations need much more time than the later ones. Runs with the binary driver patch require only a single output thread. In contrast, versions without patch and two or three output threads are shown.

It can be seen that with the binary driver patch and only a single thread, both versions (with and without lookahead) are equally fast or even faster than the yet fastest version without the patch but also without lookahead. This means the GPU DGEMM performance no longer suffers when enabling lookahead. Without the driver patch but with lookahead enabled, the GPU DGEMM performance is significantly lower.

A confusing observation is that with the patch, the GPU DGEMM performance close to the end is even higher with lookahead than without. This is due to the fact that only the raw GPU contribution to the DGEMM performance is shown here. With lookahead, the CPU performs the factorization and the broadcast during the GPU DGEMM. At the end of the run, the factorization takes up a significant amount of the iteration time. This blocks CPU recourses, reduces the part of the DGEMM processed by the CPU, and enlarges the GPU part. A bigger GPU matrix can increase the GPU DGEMM performance as the pipeline can better hide the latencies. Accordingly, the combined GPU / CPU DGEMM performance is always higher without lookahead (see Fig. 11.40).

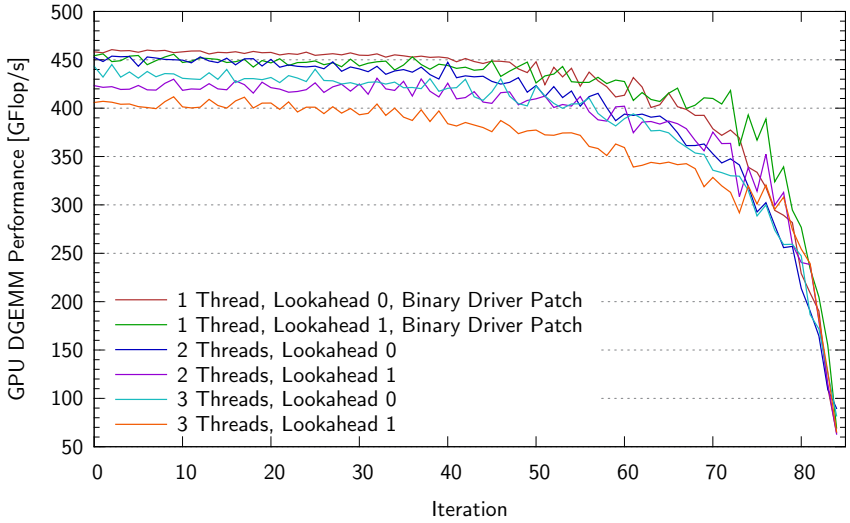


Figure 11.39: GPU-only DGEMM Performance for Lookahead during Linpack [V]

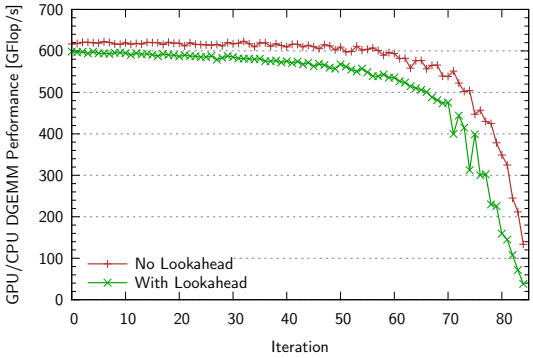


Figure 11.40: Total GPU / CPU DGEMM Performance during Linpack with Lookahead (With Binary Driver Patch) [V]

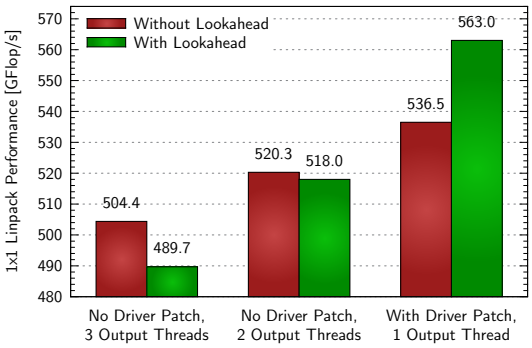


Figure 11.41: Lookahead Performance with Binary Driver Patch [V]

In combination with binary driver patch and reduced factorization thread count, lookahead is finally working well. Overall performance improves during the whole runtime. Fig. 11.41 shows the performance summary and Fig. 11.42 demonstrates the process-flow.

11.3.4.2 Lookahead 2

In a second step, lookahead attempts to hide DTRSM and LASWP. Since these tasks create the U-matrix used by the DGEMM, they must have finished before the DGEMM starts. However, it is possible to swap only the first x columns and run DTRSM only on the first x columns of the U-matrix. Then, the DGEMM can already process the first x columns of the C-matrix. So the DTRSM, LASWP, and the C-matrix update by the DGEMM can be pipelined.

At first, $x = N_b + h_{\max} = 1024 + 4096 = 5120$ columns are processed. Afterward, the GPU can immediately start processing columns $1024 - 5120$, regardless of the tiling size. (The first N_b columns are skipped as they are completely computed by the CPU DGEMM for lookahead 1 afterward.) Concurrently to the DGEMM, LASWP and DTRSM are iterated over the entire matrix. Before CALDGEMM processes a tile, it checks whether the swaps and the DTRSM for the corresponding columns have already finished. The number of columns processed in each LASWP / DTRSM

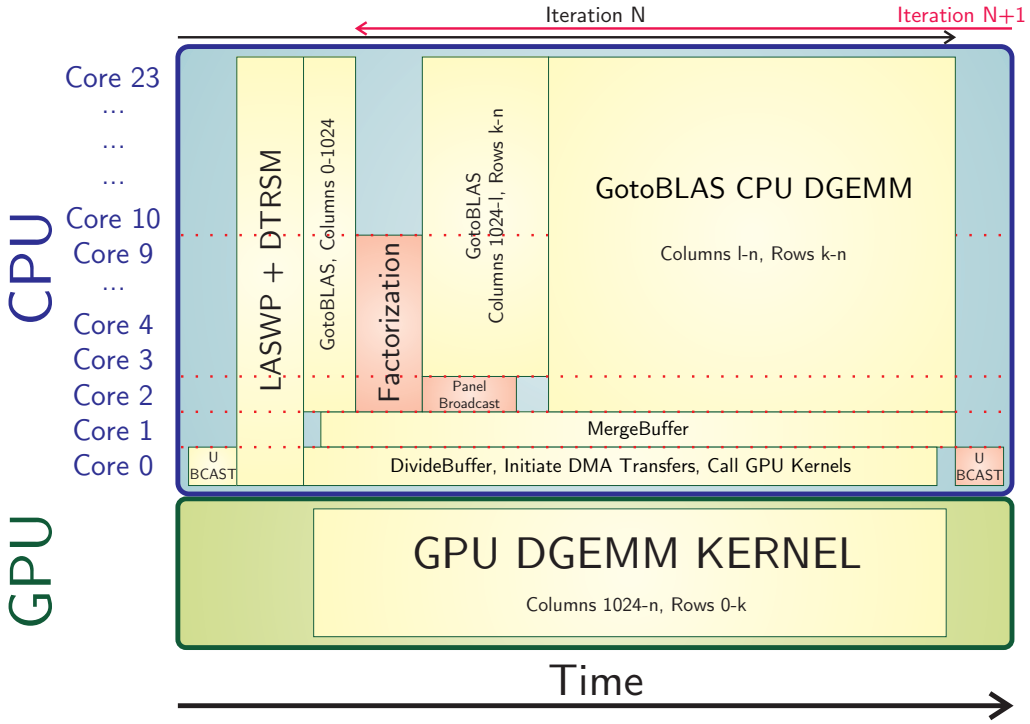


Figure 11.42: Process-Flow of GPU-based HPL with Lookahead 1 (Final Version)

pipeline step is continuously increased by a factor of two since LASWP and DTRSM operate faster on a bigger submatrix. The U-broadcast is still not included in the pipeline for its small time contribution and since HPL has no functionality to transfer submatrices of U (see Section 14.2.4.2). Since LASWP is split in two phases prior to and after the U-broadcast, the multi-node version of HPL-GPU can only pipeline the second phase of LASWP.

The first lookahead 2 implementation suffered from the same problem as the initial lookahead 1 version. In order to avoid memory congestion, the number of threads processing LASWPs and DTRSM is reduced as for the factorization. As the LASWPs are memory-bound, it seems reasonable to employ all available memory controllers. This is done by running only on even numbered cores, i. e. on half of the cores of each die. (The version is called improved lookahead 2.)

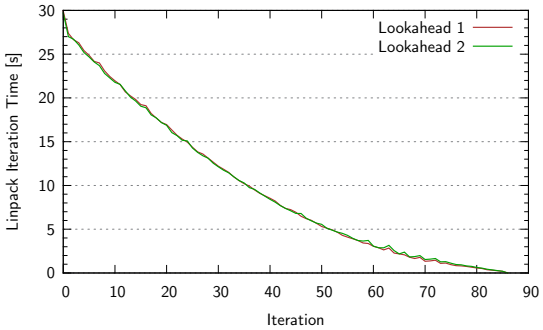


Figure 11.43: Iteration Times during Linpack with Lookahead 1 / 2 [V]

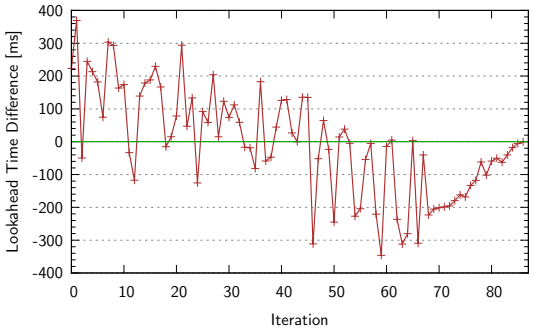


Figure 11.44: Iteration Time Difference [V]

Fig. 11.43 shows the time required for each HPL iteration using lookahead 1 and lookahead 2. They are very much alike. Fig. 11.44 shows the time difference. It reveals that lookahead 2 is faster

at the beginning of the Linpack run, but gets slower toward the end. Therefore, a mixed lookahead 2 is implemented, which switches back to lookahead 1 when the new variant gets slower.[39]

Fig. 11.45 shows the performances for the different lookahead 2 variants. Only the mixed implementation can slightly outperform the lookahead 1 code. From now on, lookahead 2 always refers to this mixed / adaptive[39] version. A comparison of single-node and multi-node runs without lookahead and with lookahead 1 and 2 is shown in Fig. 11.46. The single-node performance improves, too. This is due to the fact that not only broadcast time but also the factorization and the LASWP times are hidden. In the end, lookahead 2 does not bring about the significant improvement that has been expected. One reason, at least for the multi-node version, is that LASWP is only partially pipelined. More reasons are discussed in the next subsection. Fig. 11.47 shows the final process diagram with lookahead 2.

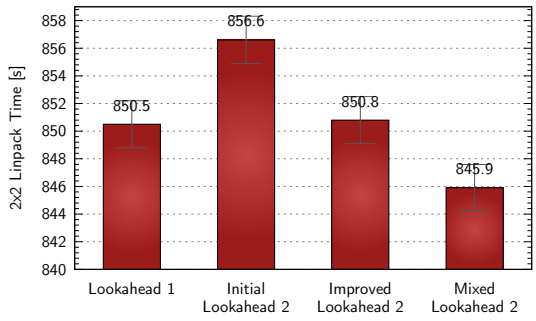


Figure 11.45: Performance of different Lookahead 2 Implementations [V]

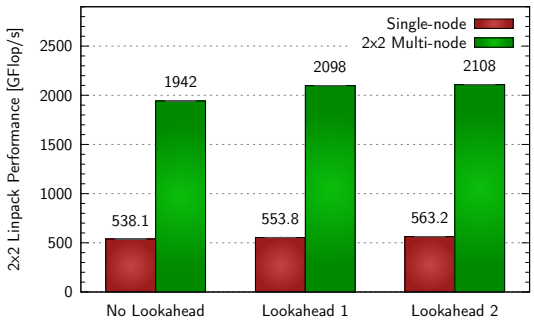


Figure 11.46: Performance of different Lookahead Modes [V]

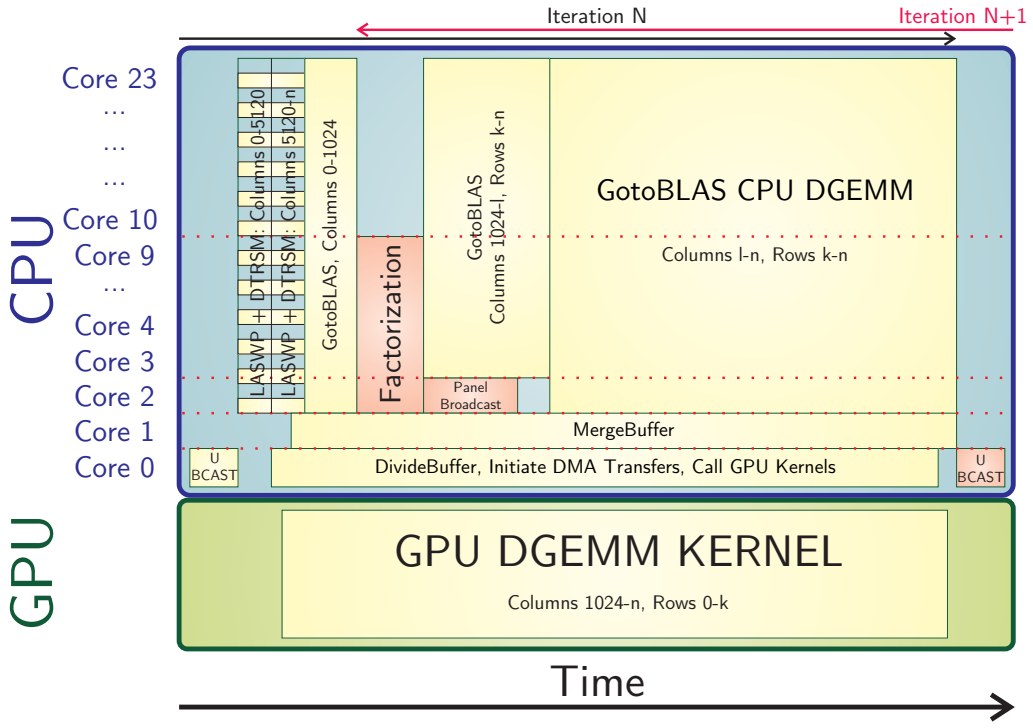


Figure 11.47: Process-Flow of GPU-based HPL with Lookahead 2

[39] The mixed mode was a quick optimization for the LOEWE-CSC. Later, an adaptive lookahead has been implemented which offers configurable turnoff points, where both lookahead 1 and 2 can be deactivated.

11.3.4.3 Performance Analysis

This section analyzes how well the CALDGEMM features work within HPL. Also the effects of the lookahead optimizations are discussed further.

CALDGEMM Performance in HPL First, an overview over the CALDGEMM scheduling efficiency shall be given. The splitting position for the first phase GotoBLAS run cannot be chosen arbitrarily. The optimal value from the performance perspective is usually unavailable due to the tiling. It lies within one row of tiles. Therefore, it is very probable that when the first phase is finished, there are still tiles unprocessed. Applying the same ratio again for the second phase should assign tiles within that row of tiles to the CPU. (If the second phase processes multiple rows of tiles, the first phase ratio was suboptimal.) Obviously, no third phase run is needed if all ratios are optimal. Fig. 11.48 visualizes the scheduling in an HPL run without lookahead. It shows the number of rows of tiles processed in the second phase and the total number of tiles processed in the third phase. The scheduling is considered optimal when the second phase value is one[40] and the third phase value is zero. Applying these criteria, the figure reveals an almost optimal scheduling.

In contrast, Fig. 11.49 shows the same plot using lookahead 2. It can be seen that the second phase value is still quite optimal. The third phase value, however, has a broader distribution. The reason is that the parallel factorization and the swapping introduce additional inherent inaccuracy into the ratio calculation. Anyway, the third phase value still drops to zero for quite a few iterations. Therefore, the CPU part must not be chosen any larger.

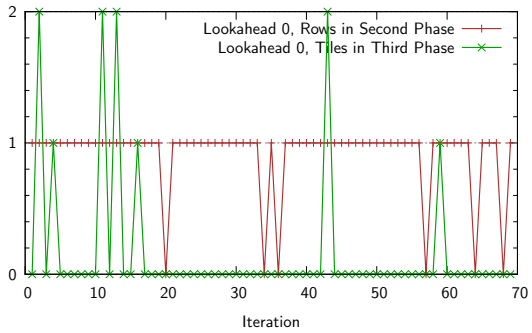


Figure 11.48: Analysis of Scheduling Efficiency without Lookahead [V]

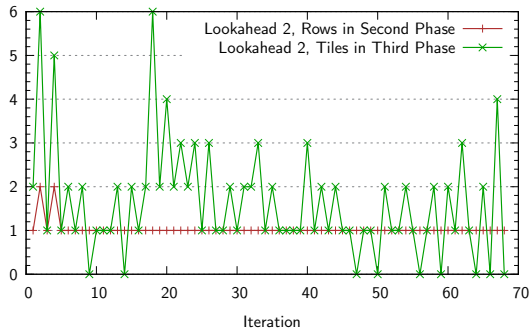


Figure 11.49: Analysis of Scheduling Efficiency with Lookahead 2 [V]

Fig. 11.50 shows the total GPU and CPU utilization time as well as the CPU DGEMM time measured by CALDGEMM. The total HPL iteration time is included for comparison. At iteration 30 CALDGEMM switches from lookahead 2 to lookahead 1. With lookahead 2, CALDGEMM GPU time and HPL iteration time are the same, with lookahead 1 they differ by the time for LASWP and DTRSM, which are no longer processed by CALDGEMM. Furthermore, with lookahead 2 the difference between the CALDGEMM CPU DGEMM time and the CALDGEMM GPU DGEMM time is the sum of factorization, LASWP and DTRSM time. Later, with lookahead 1 it equals the factorization time. One can see a jump in the CPU DGEMM time, precisely at the position where the mixed lookahead switches to mode 1. (Since DTRSM and LASWP are serialized, the CPU can process a larger part of the DGEMM in parallel to the GPU DGEMM.)

Obviously, the CALDGEMM total CPU and the CALDGEMM GPU time are almost the same. Fig. 11.51 shows the difference of the two. For processing one tile (assuming $h = 4096$), the GPU requires about 75 ms, the CPU about 230 ms. Thus, the scheduling can be improved as soon as the difference is above 230 ms or below -75 ms. It can be seen that the latter is never the case

[40] It is possible that the tiling allows for an optimal (or almost optimal) splitting position by coincidence. In these cases, a second phase value of zero is optimal.

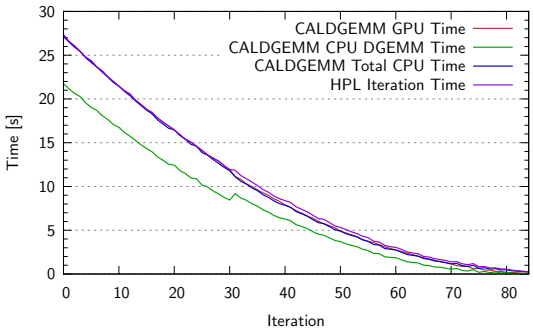


Figure 11.50: Analysis of CALDGEMM Ratio Calculation with Lookahead 2 [V]

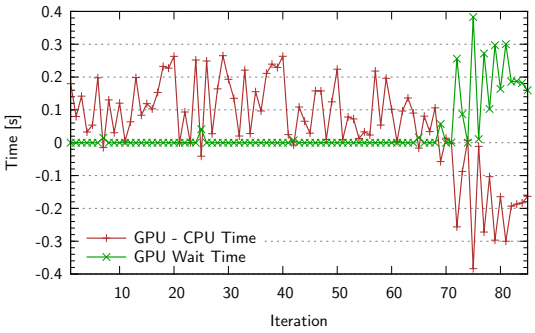


Figure 11.51: Difference in GPU / CPU Time during Linpack with Lookahead 2 [V]

except for the very end of the run. This, however, is unavoidable since here the CPU processes only the absolutely necessary tasks (pivotization, factorization, and the remainder of the matrix due to the tiling) but still takes longer for this than the GPU for the full DGEMM. In total, the difference reaches up to 250 ms, which, however, does not allow for much optimization. It should also be noted that at many points the GPU performance is overestimated, which is reflected here: The CPU idle time is longer than the GPU idle time.

In addition to the time difference, the GPU wait time is shown. This is the duration the GPU thread is waiting for the CPU thread's semaphore to unlock. In an optimal situation, this should be zero if the time difference is positive, and this should equal the absolute value of the difference otherwise. As this is the case, the synchronization works well.

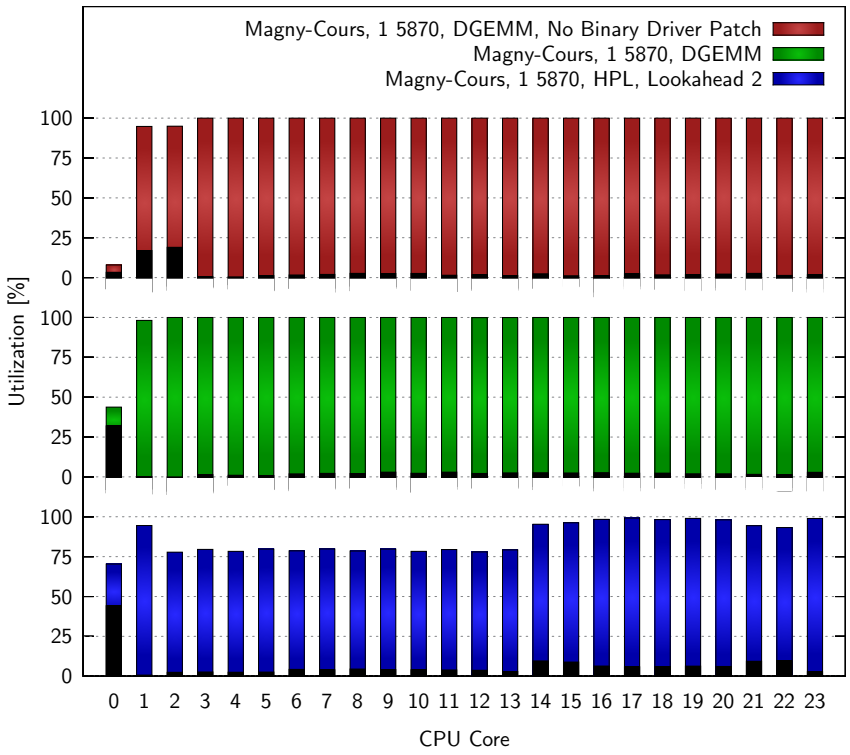


Figure 11.52: CPU Utilization during Single-GPU DGEMM and HPL[41] [V]

[41] The fraction of system load is displayed in black. The system load corresponds to the time required by the operating system while the general load accumulates the times of operating system and application. The high system load on core 0 stems from the fact that at many points synchronization is performed by active waiting.

CPU Utilization during CALDGEMM & HPL An indication for the quality of the multi-threading can be given by analyzing the CPU load of all cores. For this purpose, Fig. 11.52 gives an overview of the CPU utilization of all cores during HPL and DGEMM runs. Without the binary driver patch, the two *MergeBuffer* threads on cores 1 and 2 produce a high system load due to page faults. With the driver patch, the system load vanishes and one single CPU core (1) is sufficient, although it is heavily loaded. All CPU cores used for GotoBLAS continuously operate at 100 % load with a very small fraction of system load. The load of core 0 remains far below 100 %. However, as a matter of fact, this is essential to ensure good responsiveness to interrupts. So this tiny waste of computational power is a necessary evil.

In contrast to pure DGEMM, the average CPU load during HPL is below 100 %. First, the load is reduced toward the end of the run. Second, the processor is fully loaded only during DGEMM phases while pivotization and factorization are memory-bound and cannot utilize the CPU to its full extent. Third, in HPL with lookahead, the GotoBLAS threads (2 to 23) are unequally loaded due to the reduced factorization thread count (see Section 11.3.4).

In summary, it can be concluded that the CPU utilization during CALDGEMM is almost optimal. During HPL, the utilization is very good but especially the reduced factorization thread count results in a measurable drop of utilization.[42]

Linpack Performance during the Run The crucial task for the Linpack benchmark is to maintain the DGEMM performance in HPL. The DGEMM-portion of the total workload of each HPL iteration decreases during the run. Thus, at the beginning of the run, it is easier to achieve close to full DGEMM performance. Fig. 11.39 in Section 11.3.4.1 showed the DGEMM performance achieved in each HPL iteration. Its x-axis is not proportional to the progress because the later iterations are shorter. Fig. 11.53 compares the HPL performance during the run to the maximum possible DGEMM performance with the x-axis rescaled such that it is proportional to the progress. The single-node version with lookahead 2 achieves 93 % of the DGEMM performance at the beginning of HPL and maintains this for almost three quarters of the run. The loss of the multi-node version is bigger.

HPL Overall Performance The question remains why the speedup by lookahead 2 is so much less than by lookahead 1. A reason is given by Fig. 11.54. The LASWP and DTRSM times increase significantly when these tasks are executed in parallel to the DGEMM, leaving less time for the CPU DGEMM. For comparison: factorization time (of the first iteration) increases less when enabling lookahead 1, namely from 1.608 s to 3.216 s. In addition, factorization and broadcast require more time than DTRSM and LASWP anyway – hence serializing them costs more time. In multi-GPU configurations, lookahead 2 becomes more important (see Section 14.2.4.2).

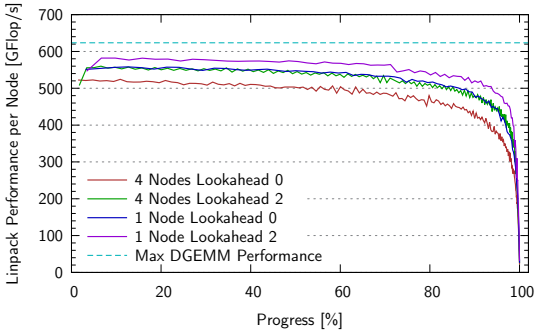


Figure 11.53: Linpack Performance during a Run (Rescaled) [IX]

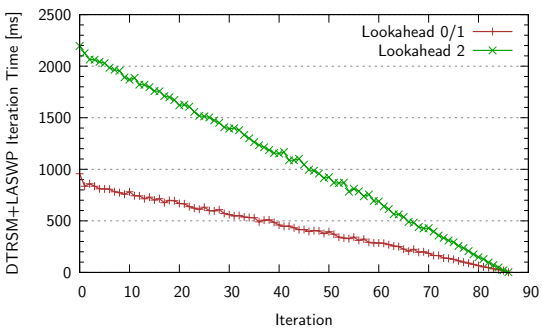


Figure 11.54: Sum of DTRSM and LASWP Time during Linpack [IX]

[42] Possible improvements to the CPU utilization during factorization are discussed in Sections 12.6.4.3 and 15.2.

Finally, Fig. 11.55 shows how the multi-node Linpack performance evolved over time with more and more improvements.

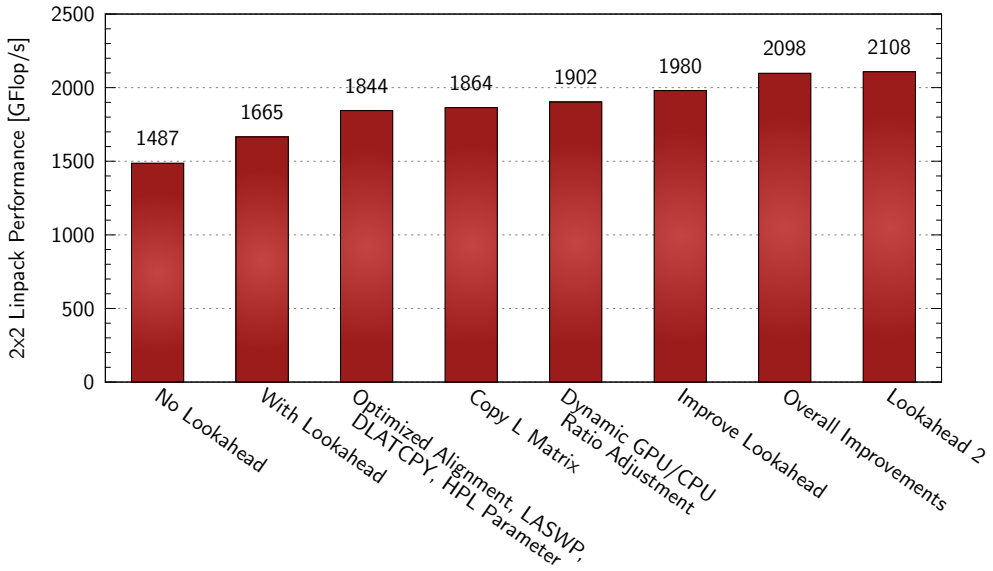


Figure 11.55: Linpack Performance Evolution Summary [V]

11.3.5 Miscellaneous

11.3.5.1 Rescheduling Workload

One possibility to boost the DGEMM performance toward the end of a Linpack run would be to redistribute the trailing matrix to fewer nodes so that the remaining matrix size per node does not shrink too much. This would reduce the number of nodes but increase the per-node performance. Fig. 11.56 shows the Linpack performance measured during the run normalized to (divided by) the remaining trailing-matrix size. If, at any point, a reduction of the number of nodes increased the overall performance, this would be visible as a falling slope. Obviously, this is never the case. Still, this approach could boost the power efficiency in the future.

11.3.5.2 MPI Threading Support

Due to stability reasons, the mvapich MPI-library is used for tests on the cluster. This MPI implementation is not reentrant[43] and supports only funneled threading (see Appendix C.1), which means that only the main thread which invoked *MPI_Init* may perform MPI calls. The first lookahead implementation invoked its MPI calls from the broadcast thread running on core 2 in Fig. 11.47 but not from the main thread running on core 0, which invokes all other MPI calls of HPL. This violates the MPI threading rules of mvapich. (Still, at no time two MPI routines are called simultaneously but sequential MPI calls are issued from different threads. In terms of MPI specifications, the first lookahead implementation requires MPI serialized threading support.)

This is solved by creating an extra dedicated CALDGEMM-main-thread. The HPL-main-thread remains in a wait state. The broadcast and factorization threads started by CALDGEMM do not issue MPI-calls themselves but delegate this work to the HPL-main-thread. The overhead in total execution time for this workaround is 0.14 %.

[43] A (library) function is called reentrant if it can be called concurrently by different threads.

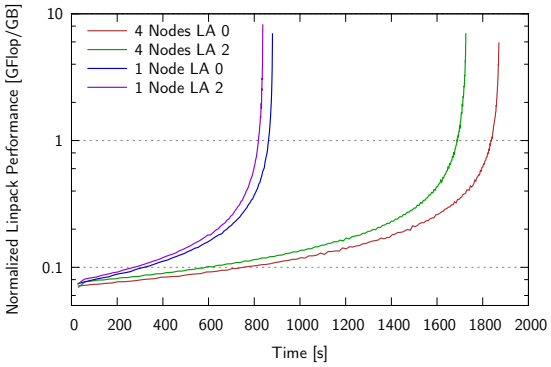


Figure 11.56: Linpack Performance normalized to Matrix Size (Lookahead 0 and 2) [IX]

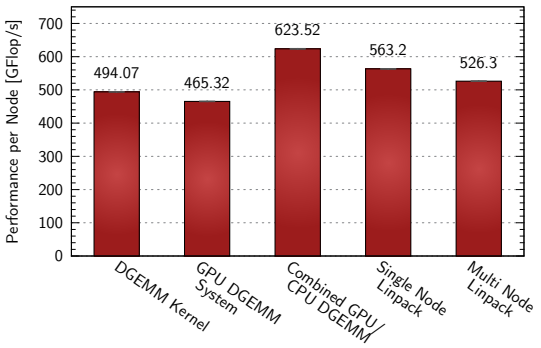


Figure 11.57: Peak Performances achieved with CALDGEMM / Linpack [V]

11.4 DGEMM & Linpack Performance

In summary, Fig. 11.57 displays the peak performance per node reached in various disciplines. Table 11.58 shows the numbers in relation to the theoretical peak performance. The kernel itself can utilize above 90 % of the GPU peak performance, DGEMM 80 % of the combined GPU / CPU peak performance, and single-node HPL still more than 75 %. The most direct competitor, the GPU implementation for the Tianhe cluster with AMD GPUs [Wan+ 11], is clearly beaten even though the higher peak performance of the LOEWE nodes makes it harder to achieve a high efficiency.

Discipline	Performance per Node	Peak Performance	Efficiency
DGEMM Kernel	494.07 GFlop/s	544.00 GFlop/s	90.93 %
GPU DGEMM[44]	465.32 GFlop/s	544.00 GFlop/s	85.54 %
GPU / CPU DGEMM	623.52 GFlop/s	745.60 GFlop/s	83.63 %
Single-node HPL	563.20 GFlop/s	745.60 GFlop/s	75.54 %
Multi-node HPL (2 × 2)	527 GFlop/s	745.60 GFlop/s	70.68 %

Table 11.58: Peak Performance and Efficiency per Node [V]

Discipline	Performance Reached	Peak Performance	Efficiency	Network Efficiency
Single-node HPL	563.2 GFlop/s	745.6 GFlop/s	75.54 %	
Multi-node HPL (2 × 2)	2108.0 GFlop/s	2982.4 GFlop/s	70.68 %	93.57 %
Many-node HPL[45]	**285200 GFlop/s**	409248 GFlop/s	69.69 %	92.26 %

Table 11.59: Multi-Node HPL Performance and Network Efficiency [V,IX]

The performance per node is used to analyze multi-node efficiency. Table 11.59 shows that the lookahead is able to conserve almost the full single-node performance. (The four-node version loses less than 7 %.) Additionally, it shows that HPL scales well to many-node systems. The number reported to the Top500 list is **285.2 TFlop/s**. It was achieved with 630 nodes and positioned

[44] The fastest kernel achieves 494 GFlop/s at $k = 448$, which causes too much overhead. The kernel used for all other measurements runs with $k = 1024$ and achieves 472 GFlop/s. The kernel performance itself is worse but the system performance is better. Thus, the system performance of the GPU DGEMM is 98.6 % of the employed kernel performance and thus almost optimal.

[45] The many-node HPL run was performed with different GPU clock speeds and the peak performance is calculated accordingly and thus no even multiple of 745.6 GFlop/s.

the LOEWE-CSC on place **22** in the Top500 and on place **8** in the Green500 [Sha+ 06, Fen+ 10] in November 2010. The efficiency of 70 % greatly exceeds the results from competing GPU clusters or other GPU-accelerated HPL implementations, which usually only achieve results slightly above 50 % [End+ 10, Kur+ 12]. The number was later improved to 299 TFlop/s using more nodes and a newer software version. Refer to Section 14.2.5 for results of the newer Sanam cluster and a comparison to more up-to-date implementations developed after the November 2010 list. Appendix G compares the ramifications of the HPL-GPU features on different hardware platforms.

11.5 Torture Tests

Measuring the GPU temperature reveals that kernels which come close to peak performance can easily overheat GPUs. This opens the possibility to use CALDGEMM as a torture test. For all tests, both system and GPU fans are pinned to 100 %.

CALDGEMM should be superior to single-node HPL in this discipline because it can keep executing GPU kernels without interruption, whereas HPL is suspending GPU execution from time to time. The CALDGEMM benchmark / torture test was originally supplied with two initialization methods: a fast method that initializes all matrices with zero (used only for performance tests) and a uniformly distributed linear congruence random number generator in the range $(0 : 1)$. Fig. 11.60 unveils an interesting fact: single-node HPL causes much more heat on the GPU than CALDGEMM using the original random number generator. This is in fact another indication how well HPL uses the GPUs. CALDGEMM with the zero initialization is way behind. This is due to the fact that no bits flip in the GPU (i. e. most flip-flops keep their state).

The reason why CALDGEMM remains cooler than single-node HPL turns out to be the random number generator. Changing the CALDGEMM random number generator range to $(-0.5, 0.5)$ or $(-10, 10)$ produces more heat but does still not reach the level of HPL. As HPL multiplies matrices that are already altered in the factorization process, the HPL matrix entries are not in the range $(-0.5, 0.5)$ – even though the initial random numbers are. Over time, the factorization makes the entries in the HPL matrix rather Gaussian distributed around zero.

A random number generator with a Gaussian distribution, estimation value zero, and variance 25 has been implemented for CALDGEMM, which finally results in the same temperature as single-node HPL. No possibility has been found to optimize the torture test any further. The increased heat generation stems from the larger variations in the exponent of the floating point representation, which lead to more shifts for the additions and more bit flips in the exponents in general.

Fig. 11.61 shows the torture test with CALDGEMM initialized by the Gaussian random number generator and single-node HPL on different LOEWE-CSC nodes. It can be seen that the temperature of both benchmarks is the same. The temperatures between the nodes, however, differ tremendously. Some nodes overheat very soon, while others remain below 85°C.

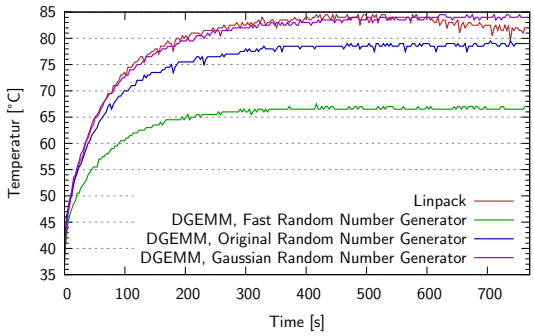


Figure 11.60: Heat produced by Linpack and different Torture Tests

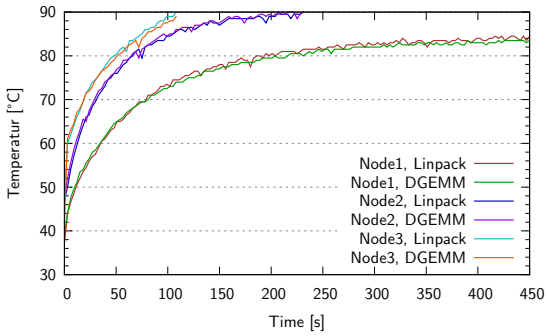


Figure 11.61: Temperature of Linpack and Torture Test for different Nodes

Chapter 12

Optimizations for other architectures

The previous chapter showed that CALDGEMM and HPL perform very well on one specific architecture: AMD Magny-Cours twelve-core dual CPU and 5870 Cypress GPU. This chapter is about achieving good performance in general configurations and on new hardware generations, while completely maintaining the performance on the reference architecture. Appendix G lists the features developed for various platforms and their impact on performance.

12.1 CPU-only HPL

One of the major changes in HPL-GPU is the restriction to one MPI process per node. The original HPL supports this configuration as well, but it does not show good performance with it. This is because only the BLAS library supports internal multi-threading but all other tasks, such as the LASWPs, do not. HPL-GPU uses parallelized versions of these tasks. The question arises how HPL-GPU performs compared to the original HPL with multiple MPI processes in CPU-only runs, i. e. how well the internal multi-threading works compared to multiple MPI processes. Table 12.1 shows the results on Intel and AMD platforms.

System	HPL, 1 MPI Process	HPL, Multiple MPI Processes	HPL-GPU	Theoretical Peak
AMD $24 \times 2.2\,\text{GHz}$	$155.3\,\text{GFlop/s}$	$174.6\,\text{GFlop/s}$	$173.6\,\text{GFlop/s}$	$211.2\,\text{GFlop/s}$
Intel $12 \times 3.3\,\text{GHz}$	-	$120.4\,\text{GFlop/s}$	$120.9\,\text{GFlop/s}$	$128.2\,\text{GFlop/s}$

Table 12.1: Performance of CPU-only HPL [VI,XVI,XVII]

It is obvious that the original version does not perform well with a single MPI process. However, the original version with multiple MPI-threads and HPL-GPU show almost equal CPU-only performance. Hence, on the one hand, HPL-GPU brings no benefit for CPU-only runs. On the other hand, since the original HPL is optimized well for CPUs, multi-threading in HPL-GPU works very well. Comparing the results with the theoretical peak performance shows that the AMD system has the better total performance but the Intel system comes closer to its peak performance. The next section shows that also the operating system has an influence.

12.2 Real-Time Operating Systems

There exist special real-time operating systems offering better pinning and scheduling capabilities as well as improved responsiveness to interrupts. Two systems are investigated.

12.2.1 The Chaos Operating System

The chaos operating system was based on Linux kernel 2.6.18 when the tests were performed. Unfortunately, the multi-threaded GotoBLAS shows bad performance in combination with the old kernel on Magny-Cours CPUs leading to only 135 $^{\text{GFlop}}/_{\text{s}}$ in Table 12.2. Either upgrading the kernel version (verified with old and new Vanilla Kernels) or switching from the AMD to an Intel platform solves this. In contrast, the original HPL running with multiple MPI processes per node and employing a single-threaded GotoBLAS performs well. Hence, the problem must lie in the GotoBLAS threading. Since multiple MPI processes are no suitable solution for HPL-GPU, the Chaos operating system can not be employed to test the GPU-based HPL. (Since Chaos consists in particular of a large assortment of kernel patches, the kernel cannot be updated to a recent version with better threading support.)

Still, the performance of CPU-only HPL runs can be compared. All measurements are taken on two 6174 Magny-Cours CPUs and 64 GB of RAM. Table 12.2 gives an overview. In the original HPL configuration, Chaos outperforms a standard Linux distribution measurably – with HPL-GPU and a single MPI process, the Chaos results are underwhelming due to the old kernel.

Operating System	Node	HPL Version	Performance
Chaos	[XVI]	Original HPL with multiple MPI Processes	174.6 $^{\text{GFlop}}/_{\text{s}}$
Chaos	[XVI]	HPL-GPU with a single MPI Process	135.0 $^{\text{GFlop}}/_{\text{s}}$
openSUSE 11.3	[VI]	Original HPL with multiple MPI Processes	168.7 $^{\text{GFlop}}/_{\text{s}}$
openSUSE 11.3	[VI]	HPL-GPU with a single MPI Process	173.6 $^{\text{GFlop}}/_{\text{s}}$

Table 12.2: Chaos HPL Performance (AMD Magny-Cours)

12.2.2 SUSE Linux Enterprise Server with Real-Time Extensions

Since SUSE Linux is available with a recent kernel version, it is not subject to the above threading issue and it supports HPL-GPU. In order to measure the interrupt and scheduling capabilities, the tests are done with the 5970 dual-GPU because this configuration puts more stress on the scheduler. (See Section 12.6 for details on the dual-GPU implementation.) The combined GPU / CPU DGEMM performance is measured.

The SUSE real-time extensions can shield CPU cores such that they are not used by the operating system anymore. In the measurement, all GPU related cores are shielded. Table 12.3 shows the results.

Configuration	Performance
No Real-Time Extensions	777.6 $^{\text{GFlop}}/_{\text{s}}$
With Real-Time Extensions	774.6 $^{\text{GFlop}}/_{\text{s}}$
With Shielding	777.7 $^{\text{GFlop}}/_{\text{s}}$

Table 12.3: SUSE Real-Time DGEMM Performance [IV]

Enabling the real-time extensions decreases the performance slightly. The shielding feature only brings the performance back to the level without real-time extensions – but it brings no additional benefit. Obviously, the thread affinities set by CALDGEMM make the shielding obsolete. This renders the SUSE real-time extensions useless for DGEMM and HPL. On top of that, the shielding feature only supports up to sixteen CPU cores.

Finally, neither Chaos nor the SUSE real-time extensions bring a benefit and are not analyzed further.

12.3 CPU Scaling

The GPU/CPU splitting is based on experimental data provided in Section 11.2.4.4 on a dual Magny-Cours system with 24 CPU cores. These data can be used as a good approximation for other CPU architectures or core counts as well. The splitting ratio is rescaled respecting the CPU frequency and the number of cores. A test on a 16-core Magny-Cours system results in about the same efficiency with respect to the theoretical peak performance (compared to the 24-core CPU).

12.4 Heterogeneous Nodes

HPL arranges all MPI nodes in a grid. In this section p denotes the number of process-rows and q the number of process-columns. A special challenge when maximizing HPL performance is a heterogeneous configuration. For instance, some LOEWE-CSC nodes are equipped with GPUs whereas other nodes offer more CPU cores but no GPU. Thus, the HPL performance of the nodes is unequal. Even the GPU nodes' performance is not necessarily identical since GPU clocks are reduced on some nodes for temperature and stability reasons. The original HPL matrix-distribution among the nodes is visualized in Fig. 12.4. The example composes six nodes – two process-rows, and three process-columns – in a column-major arrangement. The HPL-matrix is divided into blocks of size $N_b \times N_b$. The process in process-row $i \in \{0, \ldots, p-1\}$ and process-column $j \in \{0, \ldots, q-1\}$ is assigned all blocks of the matrix in block-row a and block-column b with $a \equiv i \pmod{p}$ and $b \equiv j \pmod{q}$. In this configuration, each node is assigned a submatrix of more or less equal size. This is obviously not suited for heterogeneous configurations.

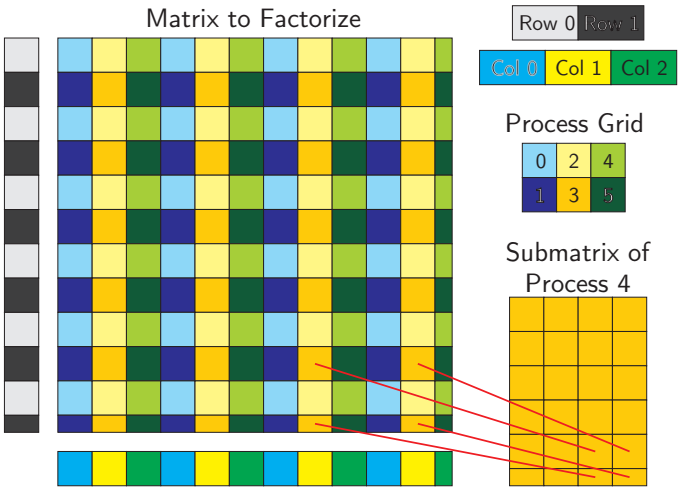


Figure 12.4: Distribution of the Matrix in the Original HPL

HPL-GPU supports a heterogeneous mode where, compared to the original HPL, the matrix size can differ among process-columns. Nodes of equal performance can be grouped in such columns. (Process-columns are preferred over rows because the implementation is easier.) A configuration file sets relative node-performances for MPI-ranks or hostnames. If no value is set, it defaults to 1.0. Let $\gamma_{i,j}$ be the setting for the process in row i and column j. The performance for each process row γ_j is the minimum of the performances of all contained nodes ($\gamma_j = \min_k(\gamma_{k,j})$). (Hence, if nodes of different performance are assigned to the same column, the matrix size is chosen based on the slowest node.) The distribution is done such that the fraction δ_j of the matrix assigned to a process-column j corresponds to the ratio of this column's performance and the sum of the column-performances ($\delta_j = \gamma_j / \sum_k \gamma_k$).

To be precise, the matrix-columns are distributed in a round-robin fashion where some process columns are skipped in such a way that each connected submatrix is distributed among the process-columns with the same ratios as the full matrix.[1] This means that a process storing e. g. 30 % of the full matrix also stores 30 % of each connected submatrix. This ensures that in each HPL iteration the distribution of the trailing matrix respects the performance ratios of the process-columns. Clearly, due to finite matrix dimensions, the ratios cannot be chosen completely arbitrarily. The matrix-row distribution-scheme is unchanged. A heterogeneous LOEWE-CSC configuration contains three types of nodes: GPU-nodes at stock clock rates, GPU-nodes with reduced clock rates, and 48-core CPU-nodes without GPU (quads). Fig. 12.5 shows the distribution for a heterogeneous HPL run with six nodes, two of each category.

HPL contains various functions for finding the process-column storing matrix entries and translating between local and global matrix coordinates. These indexing functions are altered to work with the new heterogeneous distribution scheme.

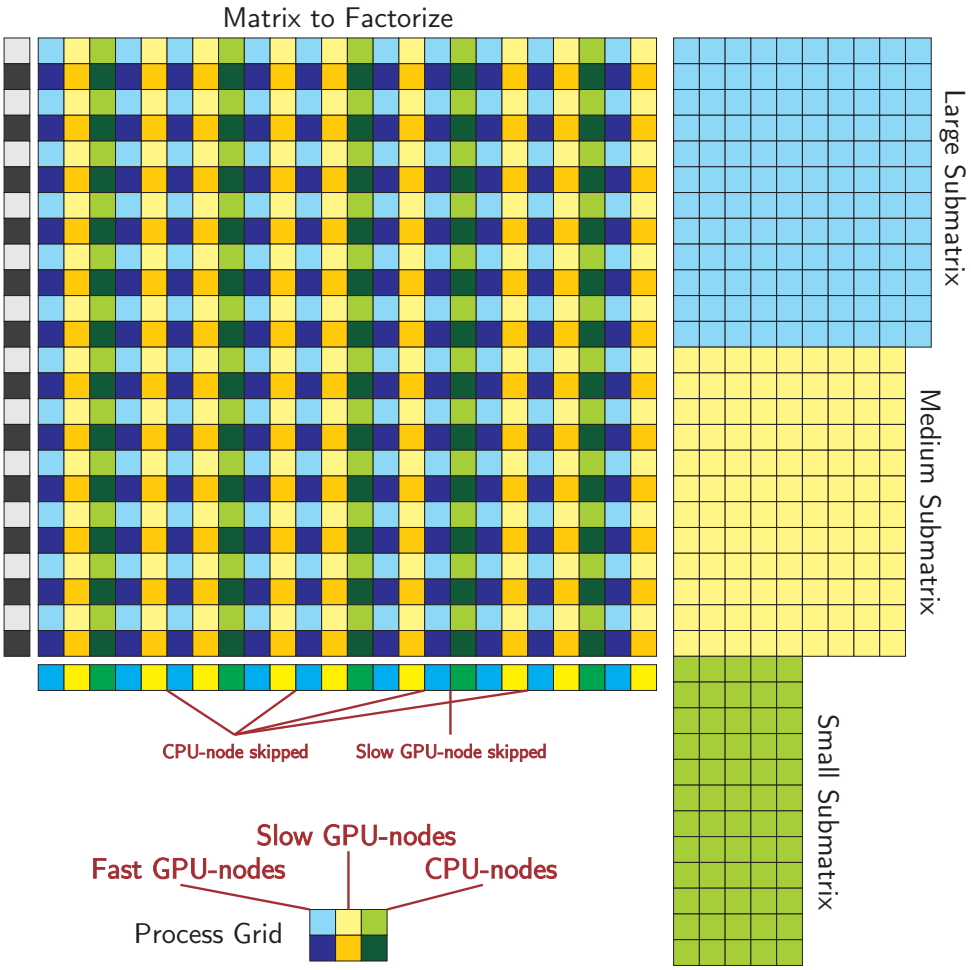


Figure 12.5: Distribution of the Matrix in the Heterogeneous HPL

12.4.1 Heterogeneous Solver for Triangular Matrices

Unfortunately, the distributed triangular matrix solver of HPL, **PDTRSV**, relies implicitly on the homogeneous round-robin distribution scheme of the original HPL. Therefore, changing only

[1] By and large, column j is skipped if the matrix-fraction already assigned to column j is larger than the total yet assigned faction times δ_j. Some heuristics treat inefficiencies arising from discrete distribution and finite dimensions.

the indexing functions without adjusting internals of the triangular solver is insufficient and breaks the functionality of PDTRSV. The original PDTRSV implements a lookahead algorithm to shorten the critical path – just like the lookahead for the original LU-factorization. Maintaining this feature turned out to be the primary challenge. In the following, as a simplification it is assumed that the process-grid contains solely a single process-row ($p = 1$), i. e. process-column j consists exactly of process j. Furthermore, it is assumed that the matrix-size is $r \cdot N_b$, i. e. a multiple of N_b. The generalization is done in the obvious manner. Each process-row handles the matrix-rows it stores and the lowermost and rightmost matrix blocks are accordingly smaller.

The original PDTRSV-solver for triangular matrices (without lookahead) works the following way: Each process has a temporary storage for the **RHS** (**R**ight **H**and **S**ide of linear equation system) which is called **swapspace**. The process storing the last diagonal $N_b \times N_b$ matrix block initializes its swapspace with the RHS, solves the triangular subsystem of the last diagonal block, and stores the solution subvector to the appropriate position of its swapspace. (For $p > 1$, the solution vector has to be replicated to all processes in the process-column.) Next, the matrix entries above the solved block are zeroed out. Being no longer needed these entries themselves are not updated at all but only the remaining swapspace (the RHS above the solution subvector just stored) is updated. Afterward, the swapspace is sent to the process storing the previous diagonal block, where the iteration starts from the beginning. After all diagonal blocks have been solved, the swapspace (of process 0) contains the solution vector.

The original lookahead introduces the following change: without loss of generality, assume that the last process $(q-1)$ stores the RHS. Note that with lookahead the swapspaces do not directly store the current RHS at any point in time but the RHS equals the sum of certain swapspaces as is seen below. Processes 0 to $q-2$ initialize their swapspace to zero. Altogether, these $q-1$ processes need only $N_b \cdot (q-1)$ rows of the RHS for the first block each of them solves. After solving the diagonal block, process $q-1$ does not update its entire RHS. Instead, only the entries of the lowermost $q-1$ blocks above the diagonal block are updated. The updated part of the swapspace is sent to the previous process, which adds this update to its swapspace. As this swapspace was originally initialized with zero, the previous process now has the part of the RHS required for the next $q-1$ triangular solve steps. While the remaining processes repeat these steps, process $q-1$ can update the rest of its RHS in order to zero out the remaining matrix entries above the diagonal block it solved. After q steps the next diagonal block to be solved is stored by process $q-1$ again. It receives the update for the current RHS belonging to the diagonal block from process 0, namely blocks $r-2q+1$ to $r-q-1$. (The block index starts with 0. Recall that in the first step only $q-1$ blocks of the RHS were transferred, namely blocks $r-q$ to $r-2$.) Hence, the update will not contain data from the original RHS (the other swapspaces were initialized with 0) but only the updates to the RHS required for zeroing out the matrix entries above the diagonal blocks processed by processes 0 to $q-2$. After solving its next diagonal block, process $q-1$ will again update $q-1$ blocks of the RHS and send this update to the previous column. With this update the previous column also gets the remaining part of the updated RHS for the very first iteration, which process $q-1$ has calculated in the meantime.

Fig. 12.6 visualizes the solution process by showing how the swapspaces develop over time. Here, $Solve(M, b)$ denotes the solution x of $M \cdot x = b$ where M is an $N_b \times N_b$ submatrix block, R_i the $r - i^{\text{th}}$ block of the initial RHS, $M_{i,j}$ the $(r-i^{\text{th}}, r-j^{\text{th}})$ $N_b \times N_b$ submatrix block of the HPL matrix, $M_i = M_{i,i}$ the diagonal block,

$$R_{i,j} = \begin{cases} R_i & \text{if } j = 0 \\ R_{i,j-1} + U_{i,j} & \text{if } 0 < j < i \\ Solve(M_i, R_{i,i-1}) & \text{if } i = j \\ R_{i,j-1} & \text{if } i < j \end{cases}$$

the $r - i^{\text{th}}$ block of the RHS (not the swapspace) after iteration j, and

$$U_{i,j} = -M_{i,j} \cdot R_{j,j-1}$$

the update to the $r-i^{\text{th}}$ block of the RHS in iteration j for $0 < j \leqslant i$. The $r-i^{\text{th}}$ block of the solution vector x is then given by $x_i = R_{i,r} = R_{i,i} = Solve(M_i, R_i + \sum_{j=1}^{i-1} U_{i,j})$. The clue of the lookahead is that after iteration j block $r-i$ of the swapspace does not necessarily have to contain $R_{i,j}$. All the updates can be delayed until this block is used as second argument of *Solve*. On top of that, the order of the accumulation $(\sum_{j=1}^{i-1} U_{i,j})$ of the update steps is not important. After the final iteration, all x_i have been calculated and each is stored by exactly one process column only.

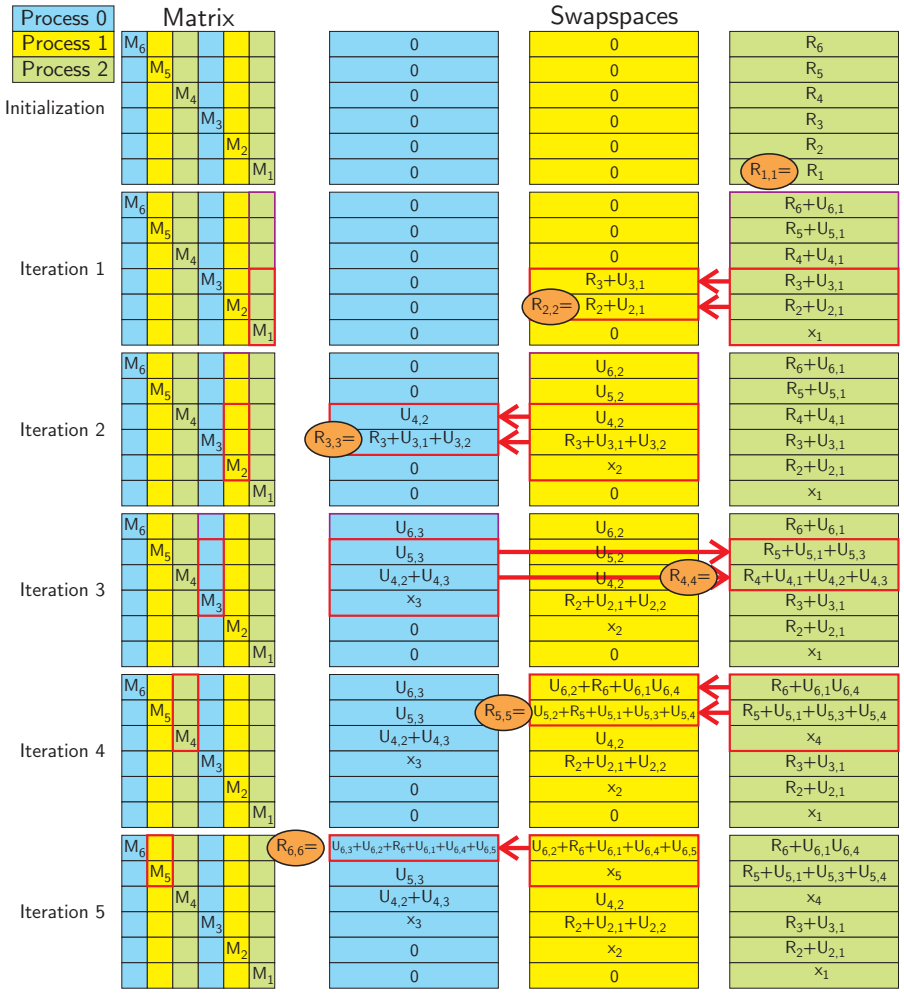


Figure 12.6: Right Hand Side Update in Original PDTRSV (With Lookahead)

The above scheme does no longer work with a heterogeneous matrix-column assignment. If a process is skipped, another process will contain two diagonal blocks which are less than $q-1$ blocks away from each other. Without loss of generality, assume this is the case for process $q-1$. In the first iteration, it sends $q-1$ blocks of its swapspace, which contain a part of the initial RHS. After less than $q-1$ iterations it will receive an update containing at least one of the blocks it sent in the first iteration. The problem is that this update will contain a part of the original RHS. Accumulating this update to the swapspace will multiply a part of the initial RHS by 2 leading to a wrong result. Fig. 12.7 demonstrates the problem.

The solution (Fig. 12.8) is to make a process send an update for only so many blocks that the first block which is not sent will be either processed by itself or the process will be skipped in the round robin distribution between the last block updated and this next block. *(The first criterion addresses the above problem and the second criterion a similar issue for the process-column next to the skipped one. Phenomenologically, updates are sent up to where the process is next considered*

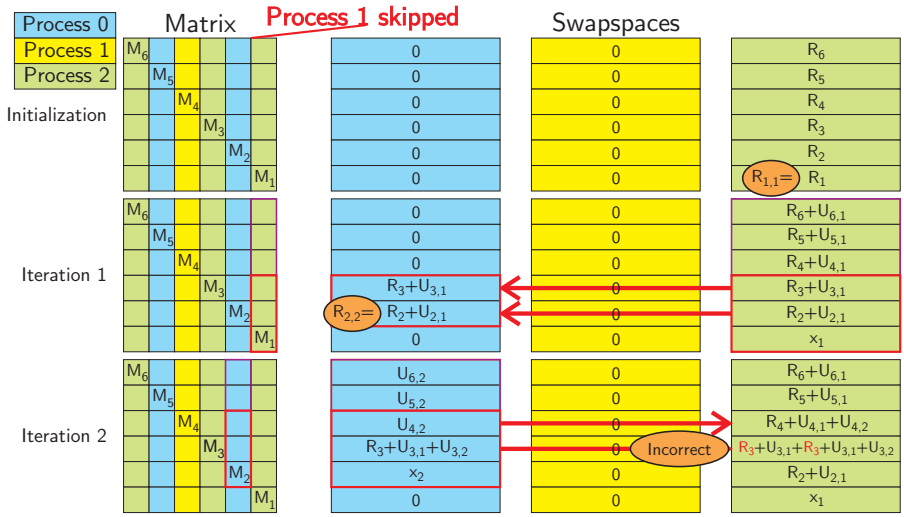


Figure 12.7: False Right Hand Side Update in incorrect PDTRSV of Heterogeneous HPL

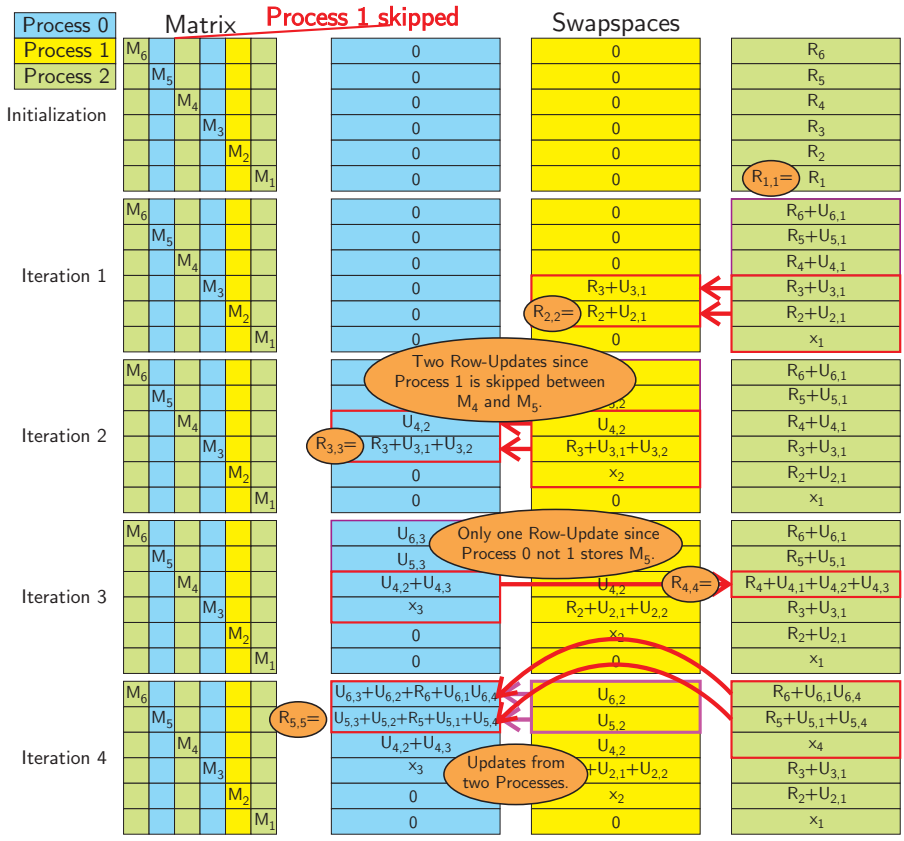


Figure 12.8: Correct Right Hand Side Update in fixed PDTRSV of Heterogeneous HPL[1]

in the round robin distribution. Mathematically, after process j solves block b it sends an update for blocks a to b − 1 with a minimal such that process j neither stores a block in the range a to b − 1 nor is skipped when round-robin distributing blocks a to b − 1.) Additionally, not only the process storing the next diagonal block must send an update but also each process skipped in the distribution since the last diagonal block. (These updates can be but are not necessarily empty.)

[1] The distribution ratios are altered in comparison to Fig. 12.7 to demonstrate all effects.

12.4.2 Heterogeneous HPL Performance

In order to measure the efficiency of the heterogeneous HPL, a test-setup with six nodes is used: two quad nodes without GPU, two GPU nodes clocked at 700 MHz, and two GPU nodes at 750 MHz. The GPU clock discrepancy is chosen small to check whether the implementation can cope with small performance differences. Each node category is first tested in a $p \times q = 2 \times 1$ grid-configuration separately. This way the performance ratios are determined. The best-case performance is defined as the sum of the $^{\mathrm{GFlop}}/_{\mathrm{s}}$ of the three 2×1 runs. The matrix sizes are chosen such that the average matrix size per node is equal in all tests. Due to the nature of the heterogeneous HPL, only the average size can be compared. Afterward, the nodes are tested in a 2×3 configuration with equally fast nodes grouped in process-columns. (It has to be noted that the heterogeneous run suffers from the disadvantage that the 2×3 configuration is more communication intensive than the 2×1 configuration used as reference.) A reference test is done without the heterogeneous feature (run I), a second test is performed with a reduced matrix size for the quad nodes (run II), and a third test with fine-tuned matrix sizes also for differently clocked GPU-nodes (run III). Table 12.9 shows the relative performance settings used for the tests. Fig. 12.10 shows the results of the 2×1 runs while Fig. 12.11 compares the reference performance, the best-case scenario, and the heterogeneous benchmarks.

Run	GPU @ 750 MHz	GPU @ 700 MHz	Quad
I	1.0	1.0	1.0
II	1.0	1.0	0.63
III	1.0	0.975	0.60

Table 12.9: Performance Ratios used in Configuration of Heterogeneous HPL Benchmark [IX,X]

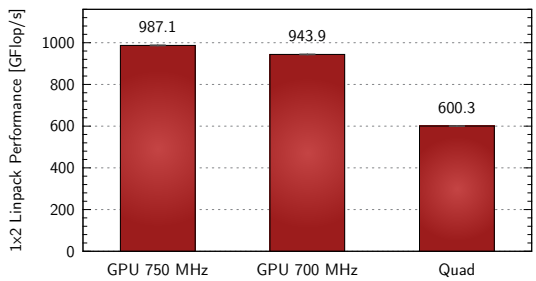


Figure 12.10: Reference Performance of Node Categories of Heterogeneous HPL [IX,X]

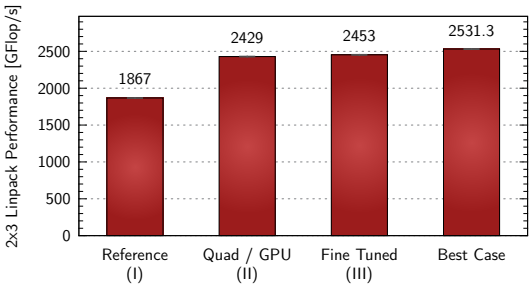


Figure 12.11: Heterogeneous HPL Performance [IX,X]

The plot shows that it is possible to fine-tune the ratios in order to account for small performance variations. The implementation achieves 96.9 % of the best-case scenario. As there are inevitable losses due to finite matrix dimensions and thus restrictions to the applied ratios, it is probably hardly possible to further improve these results.

12.5 Zero-Copy DMA Transfer on Intel CPUs

The Magny-Cours shows a very significant dependency of PCI Express throughput on the memory controller which connects the DRAM (Fig. 11.4). If it is located on a foreign die connected via HyperTransport, the bandwidth drops considerably. Fig. 12.13 shows that the effect is still measurable but far less relevant on an Intel system.

12.5.1 Kernel DMA Performance

In preparation for creating a multi-GPU DGEMM, the original CALDGEMM implementation is tested on a Nehalem configuration with a 5970 GPU. For the moment, only one GPU chip of the 5970 is active. Fig. 11.13 showed that writing the kernel output directly to the host using Zero-Copy does not slow down the kernel on the Magny-Cours system. To the contrary, both the kernel performance and the overall performance increase. This was very helpful and simplified the scheduling on the Magny-Cours because no extra control logic is needed for retrieving data from the GPU. If Zero-Copy is used on systems with Intel chipset, Fig. 12.14 shows that kernel performance decreases by about 50 % (even though the DMA performance is comparable).[2] Thus, the transfer approach chosen for the Magny-Cours is infeasible for the Nehalem. The following sections introduce a new strategy which enables CALDGEMM on Intel architectures. In addition, although not encountering a breakdown of this magnitude, also some AMD systems[3] show better kernel performance without Zero-Copy.

12.5.2 Alternative DMA Transfer Approach

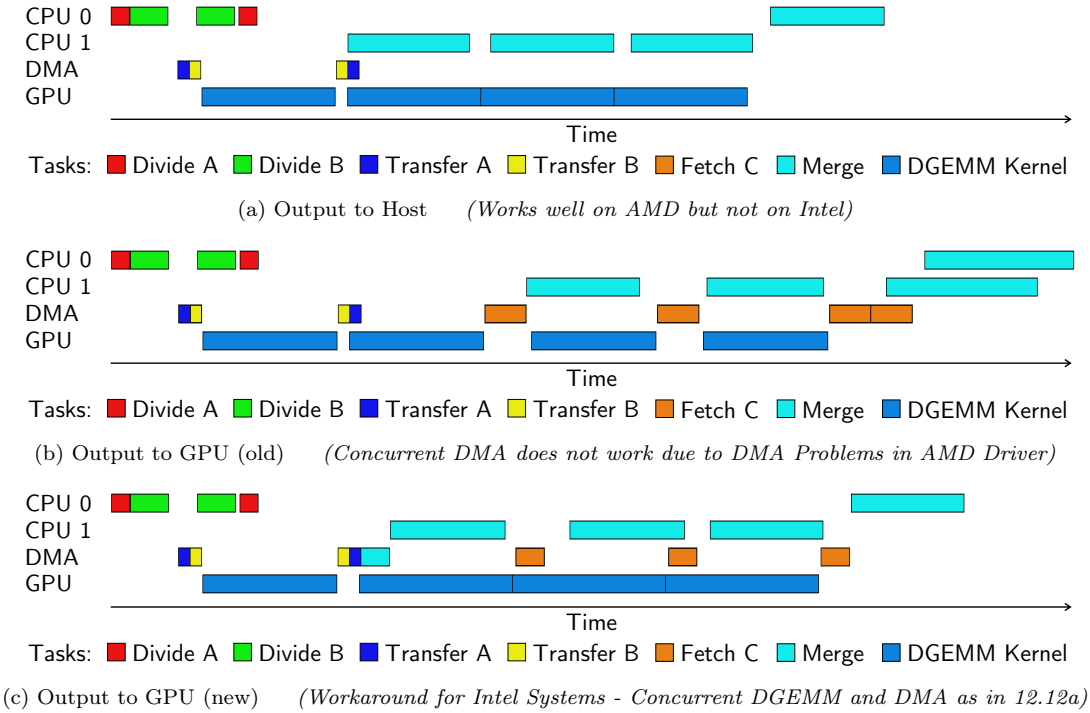
For Nehalem based systems, CALDGEMM offers the option to keep the kernel output in GPU memory and copy data to host memory in an extra step because such transfers perform well on Intel chipsets. The scheme is the following: up until now, the CALDGEMM main thread on CPU core 0 handles only the *DivideBuffer* function as well as control logic for DMA transfer to the GPU and kernel execution. The thread has a great amount of idle time and can thus also handle the DMA transfer back to the host. Instead of directly delegating the postprocessing to the *MergeBuffer* thread on CPU core 1, the main thread issues the DMA transfer back to host, waits for the transfer to finish, and only then starts the postprocessing on core 1. Clearly, also the *MergeBuffer* thread on core 1 could handle the DMA transfer. However, the load on core 1 is much higher than on core 0, thus delegating more tasks to core 1 turns out to perform worse.

Unfortunately, with the current AMD driver version, the above-described concept does not work well. The driver is incapable of starting a DMA transfer while a kernel is being executed (see Section 11.2.4.6). The kernel calls for tiles are queued. So, at the moment when the host recognizes that the kernel for tile i has finished, the kernel for tile $i + 1$ is already running and thus preventing any new DMA transfer. Unfortunately, there is no designated possibility to queue kernels and DMA transfers automatically. Comparing Figures 12.12a and 12.12b reveals how the DMA transfer is serialized. Principally, the pipeline could be extended by an additional stage such that postprocessing was two steps behind GPU kernel execution in the same way as preprocessing must be two steps ahead (see Section 11.2.4.6). However, this would require additional buffers and complicate things and thus a different approach is favored.

To make a virtue out of necessity, the DMA transfer is issued directly after the kernel invocation before any other command is sent to the GPU. According to the AMD CAL specifications, the DMA transfer would start while the kernel is still running and the data would be corrupted. However, since the driver implicitly serializes the kernel and the DMA transfer, the transfer automatically waits for the kernel to finish. It shall be emphasized that the specification does not provide the possibility to enforce synchronization in this way. The behavior of the driver gives this possibility to enqueue kernels and DMA transfers, which is not even envisioned in the specifications. Fig. 12.12c visualizes the fix.

This implicit synchronization improves the performance with the current driver, but it would fail if a new driver allowed for starting a DMA transfer during kernel execution. However, in that case the original version performs well. Since the new DMA approach does not comply with the CAL specifications, there is the possibility that a new driver leads to unspecified behavior, but is still subject to the DMA restriction. For this case, a third solution was developed. It waits

[2] The bottleneck vanishes on an AMD chipset hence it cannot originate from the 5970 GPU's PCI Express switch.
[3] Disabling Zero-Copy improves the DGEMM kernel performance from 460.8 GFlop/s to 468.8 GFlop/s on [VI].



(a) Output to Host *(Works well on AMD but not on Intel)*

(b) Output to GPU (old) *(Concurrent DMA does not work due to DMA Problems in AMD Driver)*

(c) Output to GPU (new) *(Workaround for Intel Systems - Concurrent DGEMM and DMA as in 12.12a)*

Figure 12.12: Workflow of CALDGEMM with four Tiles [V]

until a kernel finishes, then it starts the DMA transfer and only then enqueues the next kernel. It shows a performance decrease of 2.2 % and is only a fallback if both other versions fail. In summary, good CALDGEMM performance is ensured in any case.

Another aspect is important. Using the GPU's DMA engine instead of the kernel for DMA transfers does not require the host side buffers to be unmapped. Thus, the problem that led to the binary driver patch is null and void. If the driver patch is not available, the modified DMA scheme is a useful option on AMD systems as well.

A comparison of the new DMA path, the original one, and some more which are introduced later in this thesis is given in Section 12.12.

12.5.3 DMA Performance Comparison

Fig. 12.14 gives an overview of the performance of the new and the old output schemes on AMD and Intel hardware. On Intel platforms, the original version (with Zero-Copy output) does not even come close to the kernel performance. Comparing the new GPU output with the original Zero-Copy output on the AMD systems shows that the new version is less than 1 % slower. This demonstrates that the synchronization works quite well. In the end, also the Intel system reaches outstanding DGEMM system performance with the new DMA approach.

12.6 Dual-GPU & Multi-GPU

12.6.1 Dual-GPU DGEMM Implementation

In dual-GPU (multi-GPU) CALDGEMM, each device side buffer is allocated on each of the GPUs separately. Host side buffers must be bound to a specific GPU and rebinding the buffers

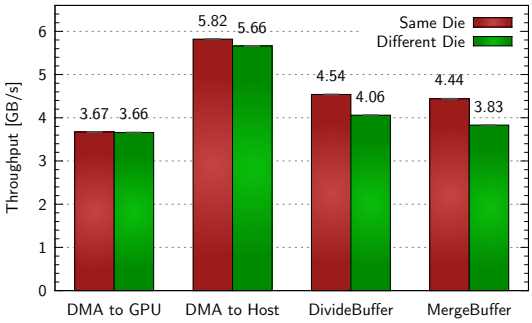


Figure 12.13: Dependency of PCI Express Bandwidth on CPU Die (Intel) [IV]

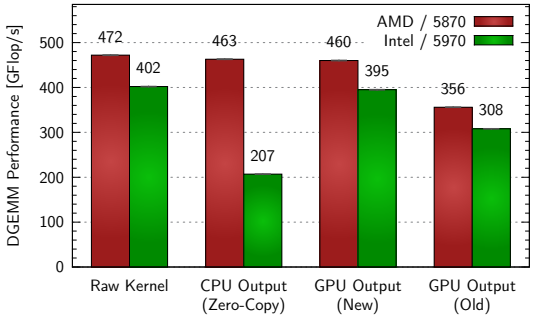


Figure 12.14: Comparison of CALDGEMM Kernel Output Schemes *(New GPU Output enables CALDGEMM on Intel CPUs)* [IV,V]

requires a considerable amount of time. On top of that, it triggers page faults similar to the buffer remapping in Section 11.2.5. Therefore, host side buffers are replicated for each device, too. As the *MergeBuffer* threads have turned out to be a critical part, each GPU gets its own dedicated *MergeBuffer* threads running on different CPU cores.

The GPU tiles are distributed among the available GPUs. The input buffers are transferred to the GPUs on demand just before the data are needed for computation (always one iteration ahead due to the DMA problem). This minimizes the data transfer if one of the GPUs does not require the whole input matrix for the tiles it processes. As described in Section 11.2.4.6, asynchronous DMA transfer ensures permanent GPU kernel execution. To analyze the simplest case first, only two GPUs are considered for the moment. In the first implementation, the tiles are distributed by a round-robin scheduler. The following results are based on this scheduling. The modification required for more than two GPUs will be small because the number of GPUs is only a parameter.

The GPU / CPU splitting ratio calculation has been adapted to respect the number of GPUs. To support different GPU types, the number of shaders and the frequency is also taken into account. In addition, the criterion for starting third phase DGEMM runs is changed and now respects the performance of CPU and GPU. Still, for a completely different architecture, this is only a rough approximation. For optimal tuning to a particular target system, the splitting ratio can be set manually. In particular, for the comparatively slow Nehalem system with only eight cores, the splitting ratio is extremely critical. Overestimating the CPU performance only slightly can lead to a huge performance decrease. To account for this, a special filter compares the aggregated CPU and GPU performances. If they differ much, a GPU-ratio increase shortens first and second phase CPU runs and provides a safety-margin. The CPU is then driven by third phase DGEMMs much more. The best value of h as obtained in Section 11.2.5.1 depends on the number of GPUs. It turns out that the optimal h for a single GPU and a problem size N scaled down by a factor of n yields a good approximation to the optimal h for n GPUs.

12.6.1.1 CPU & GPU Utilization

Fig. 12.15 visualizes the timing of a dual-GPU run.[4] The close to 100 % GPU utilization is obvious. The test was performed on an Intel based system, where the *MergeBuffer* routine is very fast.[5] Still, the load on cores 1 and 2 is above 50 % which shows that a single thread is insufficient. In any case, offloading merge tasks onto CPU core 0 is counterproductive as this reduces its responsiveness to hardware interrupts.

[4] The time axis continues in a second row.

[5] Section 12.6.2.2 shows that, compared to an Intel system, the merge thread on the AMD system causes almost full CPU load.

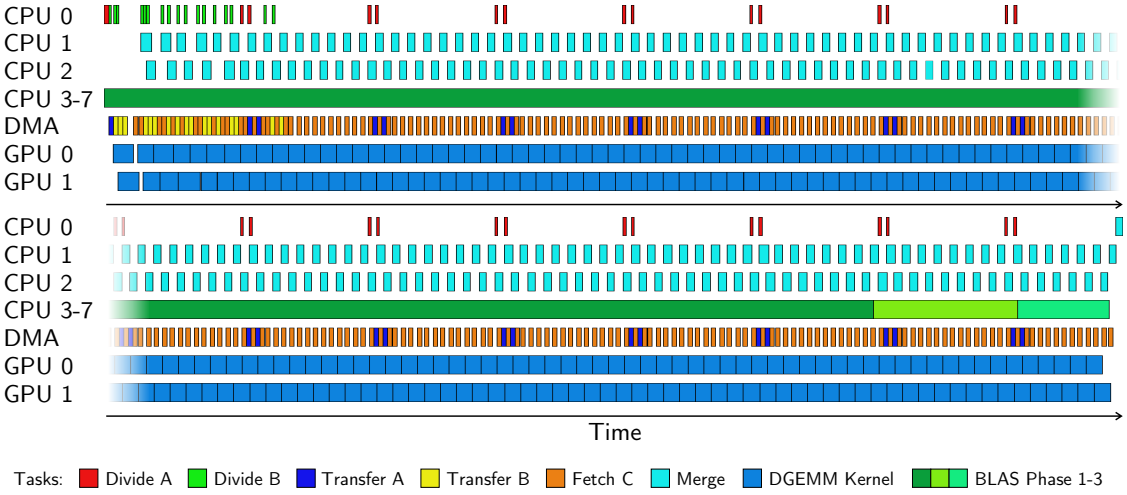


Figure 12.15: Workflow of CALDGEMM with eight Nehalem Cores and two GPUs [IV]

12.6.1.2 Performance

Fig. 12.16 shows the performance achieved on the Nehalem system with the 5970 GPU.

Two criteria are used to evaluate the quality of the implementation. First, the usual efficiency defined as the ratio of the achieved performance divided by the theoretical peak performance is of interest. This yields 85.0 % and is similar to the single-GPU version. A second approach measures the actual scalability of the multi-GPU implementation. The efficiency definition in the previous form includes inefficiencies arising from the original single-GPU version (e. g. the kernel efficiency). Thus, even an optimal multi-GPU framework is not capable of reaching 100 % efficiency.

Therefore, in order to analyze the scalability of the multi-GPU implementation, it is most instructive to examine the performance that is lost compared to the best case scenario. The accumulated maximum performance is the sum of the individual DGEMM performances of all participating components. Since every GPU requires CPU cores for pre- and postprocessing, these cores cannot contribute to the CPU DGEMM anymore. Thus, in the best case, the corrected accumulated performance is the sum of the CPU DGEMM performance with a reduced number of cores and the product of the number of GPUs with the performance of a single GPU. The **scalability** is then calculated as the achieved performance divided by the best case corrected accumulated performance. This value is not related to the theoretical peak performance but measures the quality of the multi-GPU implementation in relation to single-GPU performance.

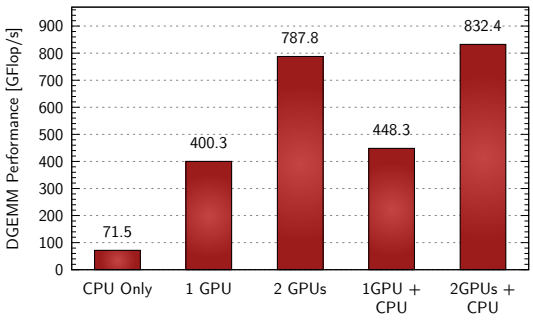


Figure 12.16: Performance of Dual-GPU Implementation [IV]

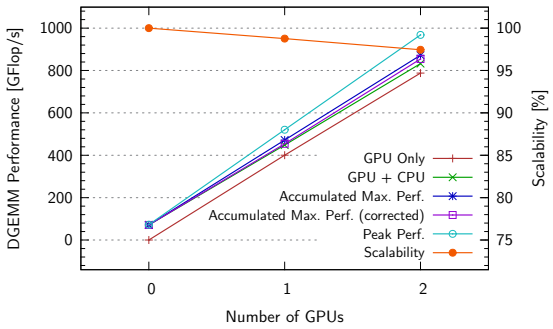


Figure 12.17: Scalability of Dual-GPU Implementation [IV]

Fig. 12.17 visualizes the scalability. On a Nehalem with two GPUs, CALDGEMM manages to achieve 832 GFlop/s out of 846 GFlop/s corrected accumulated performance corresponding to a scalability of 98.5 %, which is so close to 100 % that further optimization is unnecessary. The round-robin scheduler is thus sufficient. A more complex scheduler is discussed for multi-GPU later.

12.6.2 Scaling to Multi-GPU DGEMM

The last section showed that the dual-GPU implementation performs almost optimally. Hence, from the tile-scheduling perspective, there is no reason why this should be different for a multi-GPU version. However, several other factors have an impact on multi-GPU performance.

12.6.2.1 Memory & PCI Express Throughput

With g the GPU DGEMM performance and s the size of a matrix entry in bytes, the required PCI Express (p) and host memory (m) throughput can be calculated as:[6]

$$p \approx \frac{g \cdot s}{2k} \qquad m \approx 4 \cdot \frac{g \cdot s}{2k} = 4 \cdot p.$$

In contrast to the PCIe transfer, each matrix entry needs four memory transactions: the result is stored by the GPU in the page-locked buffer via DMA; it is then read by the *MergeBuffer* thread on the CPU, together with the old C-matrix entry; and finally, the updated C-matrix entry is written back. An increase of k shifts the calculation-to-transfer ratio toward the calculation and can thus hide transfer bottlenecks. Considering the 5870 top DGEMM system performance of 465 GFlop/s measured in Section 11.4, this yields for n GPUs and $k = 1024$ or $k = 2048$ respectively:

$$\begin{aligned} p_{1024} &= 1.82 \cdot n \quad [\text{GB/s}] \\ p_{2048} &= 0.91 \cdot n \quad [\text{GB/s}] \\ m_{1024} &= 7.27 \cdot n \quad [\text{GB/s}] \\ m_{2048} &= 3.63 \cdot n \quad [\text{GB/s}] \end{aligned}$$

The measured PCIe bandwidth on the LOEWE-CSC is 5.4 GB/s host to GPU and 6.7 GB/s GPU to host. This is still sufficient for two GPUs connected via a PCIe switch to one second generation x16 PCIe link, even at $k = 1024$. More GPUs are connected via independent PCIe links and thus do not exceed the PCIe capacity. Hence, PCIe bandwidth does not pose a problem.

System	open64 [GB/s]	ICC [GB/s]	GCC [GB/s]
Westmere, 12 Cores	39.13	39.34	-
Magny-Cours, 24 Cores	48.11	-	35.31

Table 12.18: Memory Bandwidth of Stream Benchmark [McC 95] (Copy Task) on Westmere and Magny-Cours [XVI,XVII]

The situation is different for the memory bandwidth, e. g. running three 5870 GPUs in parallel would require 21.8 GB/s only for the GPU DGEMM. Table 12.18 shows the peak memory transfer rates measured on an Intel and an AMD system using different compilers. Apparently, the AMD system has a superior peak bandwidth. However, the achievable bandwidth depends to a large extent on memory access patterns and code optimizations. The Intel system has a lower peak rate but is faster for non-specialized code.[7] It must be noted that these peak rates are achieved

[6] The factor $2k$ comes from the fact that approximately $2k$ floating point operations are required for each matrix entry.

[7] A phenomenological explanation is that the dual-CPU AMD system offers eight memory channels for four CPU dies whereas the Intel system offers six channels for only two dies. Therefore, the aggregate bandwidth is higher on AMD whereas the Intel CPU excels at achievable bandwidth of single cores.

using all CPU cores (i. e. 24 cores on the AMD system) and that, in addition, each CPU core accesses only local memory directly connected to its own memory controller. In the example with three 5870 GPUs, a throughput of almost 22 GB/s must be achieved by only three CPU cores and the DMA transfer by the GPU. This code is not specialized for memory throughput as it is constrained by the required task and it has to access partially arbitrary memory locations.[8] The parallel execution of the CPU-DGEMM even increases this demand for memory bandwidth. Therefore, the available memory bandwidth sets a limit for the DGEMM performance, which can only be raised by increasing the parameter k. Unfortunately, the kernel performance is optimal for $k = 1024$, it remains close to optimal up to $k = 2048$, and it drops for higher k. Thus, a larger k is not always better. Fig. 12.21 visualizes the k-dependency. A dual-CPU AMD Opteron at 2.3 GHz with 128 GB RAM and three AMD 5870 GPUs is used as a **reference system** in the rest of this section. The initial CALDGEMM multi-GPU implementation described above reaches 1284 GFlop/s at $k = 2048$ and has its peak of 1333 GFlop/s at $k = 2304$.

12.6.2.2 CPU Utilization

Besides the memory throughput, the CPU load on the dedicated GPU pre- and postprocessing cores can limit the achievable performance. Fig. 12.19 gives an overview of the CPU utilization of all cores during a DGEMM run. For comparison, the optimized single-GPU run from Section 11.3.4.3 is included.[9] It must be considered that synchronization by active waiting is performed in some situations, which artificially increases the system CPU load on core 0. The dual-GPU run (one 5970) performs the preprocessing for both GPUs on core 0. The load is high (96 %) but the single core is sufficient. The postprocessing uses a dedicated core for each GPU. The measurement is based on an Intel Nehalem CPU with the 5970 GPU. Due to its superior memory interface for general-purpose situations (refer to Table 12.18), the load is much lower than on AMD. Still, it turns out that a single CPU core is insufficient.

The runs with three GPUs cannot be directly compared to those with fewer GPUs because k is set to 2048. Postprocessing on cores 1 – 3 runs on medium load, but the preprocessing on core 0 saturates at 100 % load and limits the performance. This suggests a multi-threaded *DivideBuffer* implementation. The next paragraphs describe how the multi-GPU implementation has been improved based on these conclusions.

CPU Pinning The original single-GPU HPL pins the pre- and postprocessing cores to the CPU closest to the GPU and allocates the GPU buffers there. There is no need to change this for a dual-GPU like the 5970. However, the three-GPU system has the GPUs connected to different CPU dies (see Fig. 12.33 in the next chapter). The pinning routine is altered such that a CPU die can be set for each GPU. Multiple GPUs can be assigned to the same die. CALDGEMM automatically runs the *MergeBuffer* threads on cores on that die and allocates the memory accordingly. This requires a reservation of a non-continuous set of cores from GotoBLAS. The GotoBLAS patch is thus extended to provide this functionality. This update provides more memory bandwidth for CALDGEMM and the optimal k decreases from 2304 to 2048. The performance increases from 1333 to 1366 GFlop/s.

Multi-Threaded *DivideBuffer* Fig. 12.19 reveals a very high CPU load for the *DivideBuffer* core in a three-GPU configuration. Thus, in an improved version one thread is created per CPU die that has at least one GPU connected to it. Such a thread then performs the preprocessing and DMA transfer for all GPUs connected to this die (usually not more than two GPUs). The scheduling tasks are processed by the first *DivideBuffer* thread. In the three-GPU reference configuration,

[8] The page-locked buffer is allocated on memory local to the CPU core but the C-matrix is stored in interleaved form, i. e. it is spread among all available memory controllers in a round robin fashion.

[9] The single-GPU run (one 5870) shows a medium load for the core doing preprocessing (43 % on core 0) and high load for the postprocessing core (97 % on core 1). Still, the single postprocessing core is sufficient.

one GPU is connected to die 0 (cores $0-5$) and two GPUs are connected to die 2 (cores $12-17$) (Compare the PCI Express performance in Fig. 12.33). Thus, two *DivideBuffer* threads are started, one on core 0 (for GPU 0) and one on core 12 (for GPUs 1 and 2). In addition, three *MergeBuffer* threads are running on cores 1, 13, and 14. The figure shows that the load on core 0 does not decrease significantly but is very close to 100 %, probably due to the additional synchronization overhead (core 0 does all the scheduling and synchronization). The second *DivideBuffer* thread on core 12 has a considerably lower load. In the future, the scheduling could be improved to not rely on a single thread. The reference performance improves from 1366 to 1410 GFlop/s.

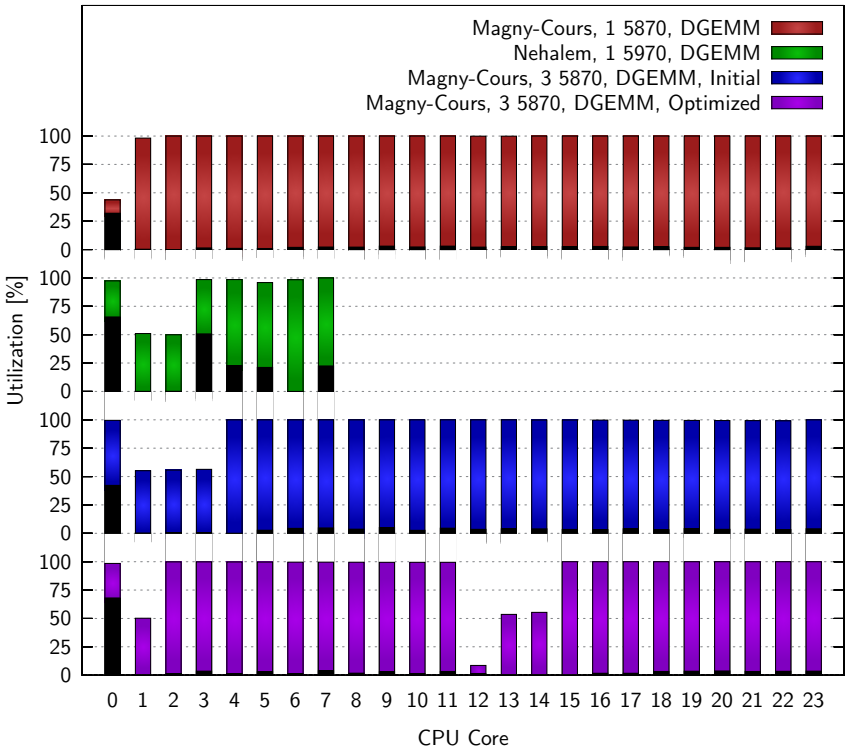


Figure 12.19: CPU Utilization during Multi-GPU DGEMM[10] [IV,V,VI]

12.6.2.3 Other Multi-GPU Improvements

Partial *B*-Matrix Caching Originally, BBuffers could be used as long as the smaller matrix side was below $\frac{b \cdot h}{r}$, with r the splitting ratio, h the tiling size, and $b \sim \frac{1}{k}$ the number of BBuffers. Switching from $k = 1024$, $r \approx 0.7$ to $k = 2048$, $r \approx 0.93$ the maximum allowed matrix size (to be precise its smaller dimension) drops from 129024 to 48128. This makes the multi-GPU DGEMM infeasible for almost-square matrices as in HPL. In a first step to mitigate this problem, a partial B-matrix caching is implemented.[11] The highest possible number b of BBuffers is allocated in each GPU's memory. The first $b-2$ blocks of the B-matrix required on a GPU are cached on this GPU in dedicated buffers. Retransfer of these blocks is not necessary. All other blocks, however, have to be retransmitted every time (stored in the remaining two buffers alternately). This simple cache scheme is in fact optimal. If a more sophisticated approach (e. g. an LRU) substituted block i by block j, block i will have been retransferred before block j will be needed again. Partial caching also improves the single-GPU version for very large matrices.

[10] The fraction of system load is displayed in black. The optimized GPU version includes all improvements of this section whereas the initial version does not.

[11] In the following, it is assumed (as in Section 11.2.4.6) that the B-matrix is smaller than the A-matrix ($n < m$). If this is not the case, inner and outer loop of the tiling are permuted such that all findings remain the same.

***B*-Matrix Splitting** The next step bases on the partial BBuffers above and finally solves the matrix size restrictions by replacing the round-robin scheduler with an improved version. The B-matrix is split in as many parts as there are GPUs. The tiles corresponding to each part of B are processed solely on one of the GPUs.[12] Only a fraction of the B-matrix must be transferred onto each GPU. While possibly the GPU cannot cache the entire B-matrix, its memory might very well suffice for this fraction. In this case, no tile is ever retransferred. This change in the scheduling policy comes at zero overhead since one CPU core is dedicated for preprocessing and DMA management anyway. As soon as a GPU has processed its part of B, it falls back to the round-robin scheduler and can steal tiles from other GPUs. This ensures continuous GPU utilization in any case. The corresponding blocks of B can still (in the round-robin phase at the end) be cached if sufficient BBuffers are available. Fig. 12.20 shows how the matrix is distributed. This enables one to run on square matrices spanning the full 128 GB of memory the system offers. The maximum achieved performance is **1435** $\mathbf{GFlop/s}$. The multi-GPU scalability increases to 93.8 %.

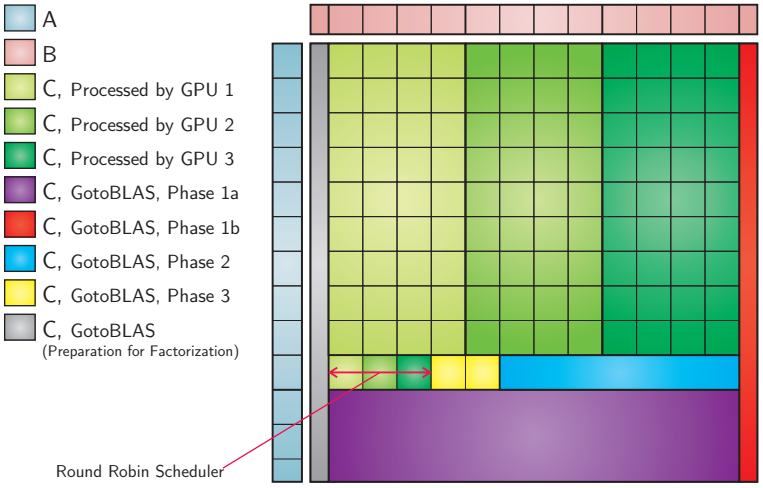


Figure 12.20: Multi-GPU Distribution of C-Matrix[12]

12.6.3 Multi-GPU DGEMM Results

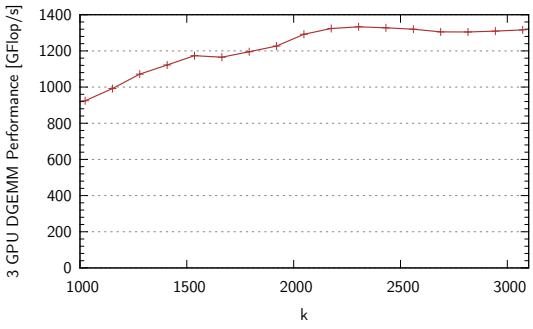


Figure 12.21: Dependency on k Parameter using three GPUs [VI]

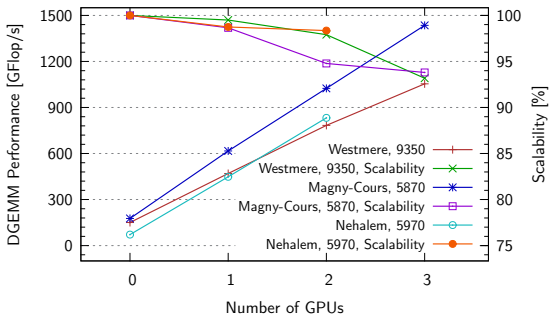


Figure 12.22: Performance of multiple GPUs on different Architectures [IV,VI,XV]

Fig. 12.22 visualizes performance and scalability while Table 12.23 lists various multi-GPU DGEMM results on different platforms in more detail. Table 12.24 shows the corrected accu-

[12] The actual implementation is a bit more sophisticated than indicated in this section. In case the B-matrix cannot be distributed uniformly among the GPUs due to tiling size restrictions, tiles within one column of tiles are spread onto two GPUs such that in the end each GPU computes the same number of tiles.

mulated performances and multi-GPU scalabilities (as for the initial dual-GPU implementation in Section 12.6.1.2). Out of the two-GPU measurements, the Intel system with the 5970 GPU shows the best scalability. The reason is that its memory interface allows for running two GPUs with $k = 1024$ while the AMD system has to use $k = 2048$, as soon as two GPUs are active. For three GPUs or faster GPUs, all systems use $k = 2048$ and the scalabilities are comparable.[13]

System		GPUs	Mode	Performance [GFlop/s] $k = 1024$	$k = 2048$
Nehalem, 8 Cores, 5970	[IV]	0	GPU / CPU	71.5	
Nehalem, 8 Cores, 5970	[IV]	1	GPU / CPU	448.3	
Nehalem, 8 Cores, 5970	[IV]	2	GPU / CPU	832.4	
Nehalem, 8 Cores, 9350	[XV]	1	GPU	345.9	343.7
Nehalem, 8 Cores, 9350	[XV]	2	GPU	669.0	680.4
Nehalem, 8 Cores, 9350	[XV]	3	GPU	858.5	1003.7
Nehalem, 8 Cores, 9350	[XV]	1	GPU / CPU	469.7	468.0
Nehalem, 8 Cores, 9350	[XV]	2	GPU / CPU	722.6	783.2
Nehalem, 8 Cores, 9350	[XV]	3	GPU / CPU	770.0	1053.9
Magny-Cours 24 Cores, 5870	[VI]	1	GPU	463.1	445.6
Magny-Cours 24 Cores, 5870	[VI]	2	GPU	699.7	891.1
Magny-Cours 24 Cores, 5870	[VI]	3	GPU	887.7	1332.7
Magny-Cours 24 Cores, 5870	[VI]	1	GPU / CPU	616.9	594.8
Magny-Cours 24 Cores, 5870	[VI]	2	GPU / CPU	705.1	1024.3
Magny-Cours 24 Cores, 5870	[VI]	3	GPU / CPU	923.0	1435.3

Table 12.23: Multi-GPUs DGEMM Results on different Architectures

System		GPUs	k	Performance	Best Case	Scalability
Nehalem, 8 Cores, 5970	[IV]	2	1024	832	846	98.35 %
Nehalem, 8 Cores, 9350	[XV]	3	2048	1054	1131	93.18 %
Magny-Cours 24 Cores, 5870	[VI]	3	2048	1435	1530	93.78 %

Table 12.24: Multi-GPU DGEMM Scalabilities on different Architectures

12.6.4 Multi-GPU HPL

The previous section optimized the CALDGEMM multi-GPU performance as far as possible. The main bottleneck for multi-GPU HPL now lies in the factorization. Multiple reasons exist for this: (Consider one HPL iteration)

- Increasing $N_b = k$ to 2048 doubles the DGEMM time (per iteration) for the trailing matrix update whereas the factorization time goes with N_b^3. Thus, the factorization, which cannot be implemented as efficiently as the DGEMM, contributes much more to the overall execution time.
- The multi-GPU CALDGEMM does speed up the DGEMM but not the factorization. Thus, the factorization contributes even more to the overall time.

Obviously, there is no other solution except for speeding up the factorization.

[13] In fact, the triple-GPU measurement on the Magny-Cours reveals a slightly higher scalability than on Intel CPUs. The reason is that the Intel node has less main memory than all other test systems. At equal matrix sizes, due to Intel's better memory interface, it is the other way around. Still, the differences are marginal.

12.6.4.1 GPU-based Factorization

An obvious approach is to do the factorization on the GPU. In principle, there are two realizations:

- The entire factorization is performed by the GPU. The full matrix is transferred to and from the GPU before and after the GPU factorization.
- Only time consuming BLAS calls during the factorization are offloaded to the GPU. This requires many more PCIe transfers in between, namely two or more transfers per BLAS call.

Both possibilities are analyzed in [Bac 09, 5.3]. For obvious reasons, the first version is shown to perform better. However, it only works well as long as the grid is configured as $1 \times q$. As soon as the process-grid contains at least two rows, the pivotization requires one communication step for each column processed, i. e. $N_b = 2048$ transfers to and from the GPU in the given case. For running tens to hundreds of nodes, this approach is thus not suited.

For the second approach, a case-study is conducted. The AMD ACML-GPU library [Adv I] is linked to HPL and its GPU-based BLAS functions are used for the factorization. Clearly, these GPU implementations are not as specialized for the hardware as CALDGEMM is for large matrices, but the library allows one to easily examine the capabilities of the GPU. The BLAS calls during the factorization have quite small parameters, the largest is an $m = n = k = 1024$ DGEMM. Such small calls cannot benefit from multiple GPUs at all, thus only a single GPU is used. Unfortunately, the overhead for DMA transfer, etc. greatly exceeds the speed gain of the GPU BLAS calls such that the overall factorization time even increases. Naturally, computers with less potent CPUs might still benefit (see Section 12.11). In the following, only optimizations to the CPU factorization are discussed. From this perspective the deficient tuning of ACML-GPU and the lack of an AMD counterpart to GPU Direct take their toll. Modern HPL implementations for NVIDIA support GPU-based factorization with promising results [Kur$^+$ 12, Shi 13].

12.6.4.2 GotoBLAS Tuning

The recursive factorization is based on GotoBLAS calls, however, with a wide input parameter range. Obviously, using 24 cores is not always optimal, e. g. for multiplying 4×4 matrices. Table 12.25 lists all GotoBLAS calls during the factorization with nonzero time consumption.

BLAS Call	DTRSV	DTRSM	DSCAL	DGEMV	DCOPY	DAXPY	DGEMM
Time	0.22 %	2.38 %	18.60 %	11.07 %	10.71 %	4.34 %	52.91 %

Table 12.25: Time Distribution of GotoBLAS Routines during Factorization [VI]

These calls are analyzed deeper. The performance for multiple input parameters resulting in various complexities is measured and related to the thread count. In this way, the optimal number of threads for each level of complexity / input parameter range is determined and GotoBLAS has been patched to use this number automatically. As an example, Fig. 12.26 shows the results for the DGEMM, which is the most important call – also inside the factorization. Clearly, the obtained thread count is only a vague approximation to the real optimum since different parameters can lead to the same overall complexity. It turns out that the DGEMM performance does not depend heavily on m, n, and k individually (compare to Section 11.2.4.4) except for special cases such as $m, k \gg 1$, $n \approx 1$. Other calls, especially the memory-bound tasks, show a huge dependency indeed. Thus, the thread count is chosen based on the matrix shape and the complexity.

12.6.4.3 Enabling Lookahead

Next, lookahead is enabled. In Section 11.3.4, it was concluded that the factorization thread count has to be reduced to avoid memory congestion. A deeper analysis of this reveals that this

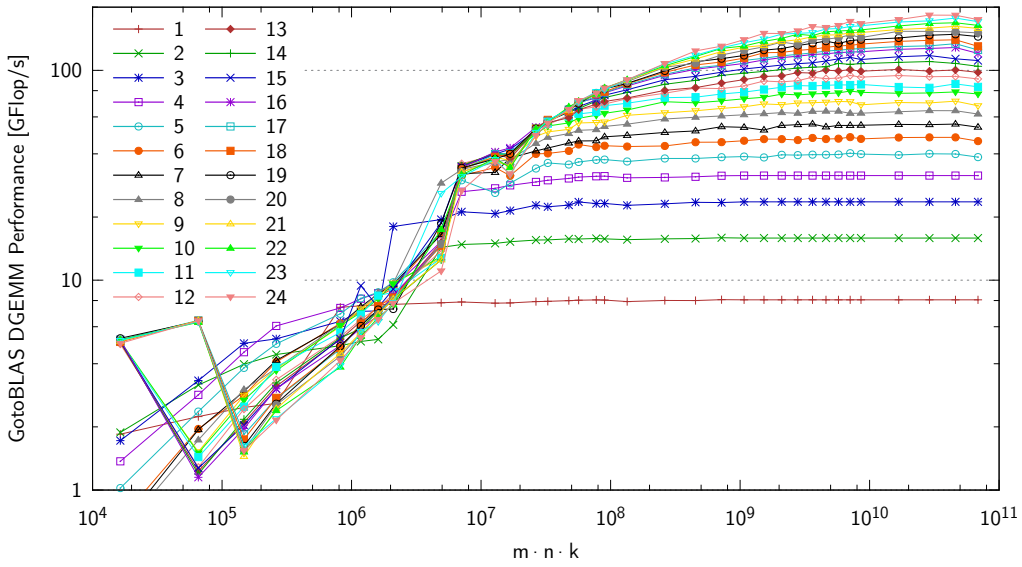


Figure 12.26: GotoBLAS DGEMM Performance in Relation to Thread Count and computational Complexity [VI]

is only relevant for the memory-bound BLAS tasks. Thus, the cut for a maximum of eight threads is applied only for them (in addition to the thread count reduction for small input parameters). CPU-bound tasks, such as the DGEMM and in certain cases the DTRSM, can utilize all available cores. Fig. 12.27 shows a scheme of multi-GPU HPL with lookahead.

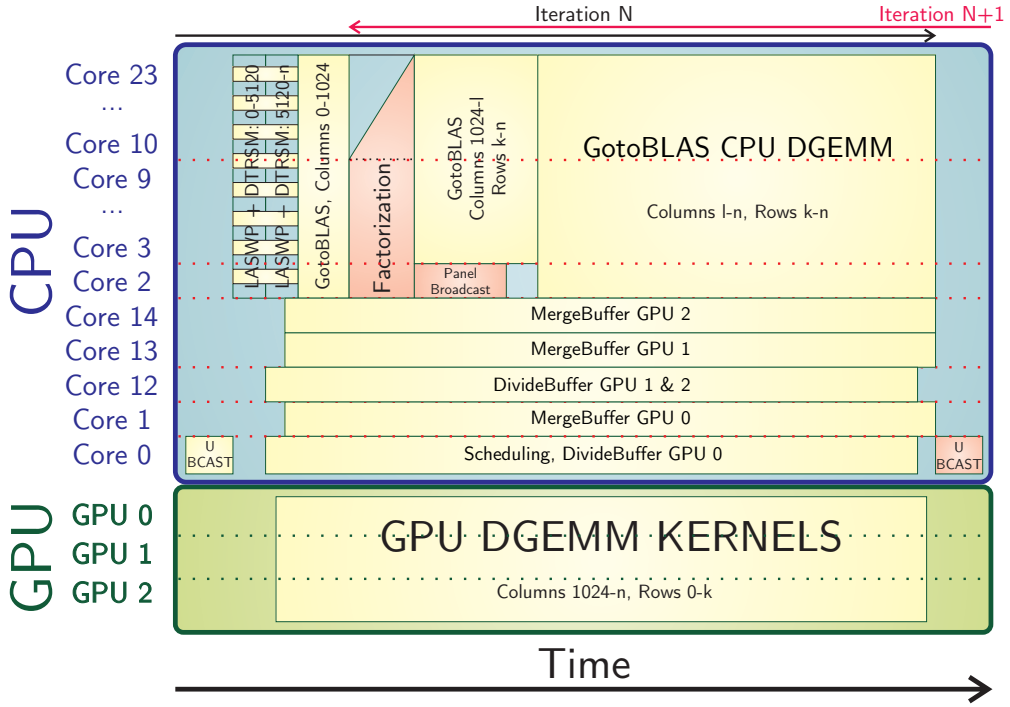


Figure 12.27: Process-Flow of Multi-GPU HPL

Despite the GotoBLAS optimizations, the factorization time with lookahead is much higher than without (9.5 s versus 5.8 s on the test system [VI]). Lookahead can only be used as long as the nec-

essary CPU tasks do not take longer than the GPU DGEMM. These tasks are: the factorization itself, the processing of the first N_b columns to prepare the factorization, and the processing of the borders of the matrix to ensure a matrix size divisible by h. For small matrices, these tasks exceed the DGEMM time and only for large matrices above 64 GB lookahead can be used. In comparison to the single-GPU version, the large value of k increases the factorization time enormously. The adaptive lookahead (originally switching from mode 2 to 1), is complemented with the capability to turn off lookahead completely as soon as the trailing matrix size shrinks below a certain configurable limit. The original pipeline implementation of lookahead 2 has been incompatible with the initial multi-GPU implementation and is not used for the following benchmarks. (This has been solved for the Cayman GPU family in Section 12.8.4.) The maximum HPL performance measured with three GPUs, 128 GB of memory, and dynamic lookahead deactivation is **1114 GFlop/s**.

12.7 Energy Efficiency

12.7.1 Multi-GPU Considerations

Naively, with s_g the single-GPU-DGEMM speed, s_c the CPU-DGEMM speed, p_g and p_s the GPU and system power consumption, α the multi-node HPL performance compared to the DGEMM performance, and β the power-supply efficiency, the power efficiency (performance per watt) relevant for the Green500 of an n-GPU system can be calculated as:

$$\frac{n \cdot s_g + s_c}{n \cdot p_g + p_s} \cdot \alpha \cdot \beta.$$

(In reality, this ratio has to be multiplied by a factor $\gamma(n)$ which corrects for the reduced efficiency when running multiple GPUs, e. g. caused by memory bandwidth limitations and the pre- and postprocessing for the additional GPUs. In the following, it is assumed that the implementation can achieve that $\gamma(n)$ remains of order one, at least for $1 \leqslant n \leqslant 4$.)

The GPU performance per watt ($\frac{s_g}{p_g}$) greatly exceeds the CPU's ($\frac{s_c}{p_s}$). Thus, the overall power-efficiency increases with more GPUs.

12.7.2 First Results

To measure the achievable power efficiency, a test-setup with three 5870 GPUs, two 6174 Magny-Cours CPUs at 2.2 GHz, and 128 GB RAM [VI] is used. Two measurements are taken:

- A reference benchmark utilizing the full 128 GB of memory.
- A benchmark with only 64 GB of physical memory installed in the system. (HPL performance generally increases with more memory but power consumption does as well. Thus, the best configuration has to be determined experimentally.)

Fig. 12.28 shows the results. The power consumption jumps between two values: one high value during the DGEMM and one low value during the pivotization when the GPU is inactive (because lookahead 2 is not used due to incompatibilities with multiple GPUs). The power consumption decreases toward the end, where the factorization becomes more important. To enable a fair comparison, the average power through the entire run is considered. In average, the performance divided by the power consumption during the tests is:

- **1167.6 GFlop/J** using a 128 GB Matrix
- **1020.6 GFlop/J** using a 64 GB Matrix

The curve for the 64 GB test lies slightly below the 128 GB curve. Still, the 128 GB run is more efficient due to higher HPL performance.

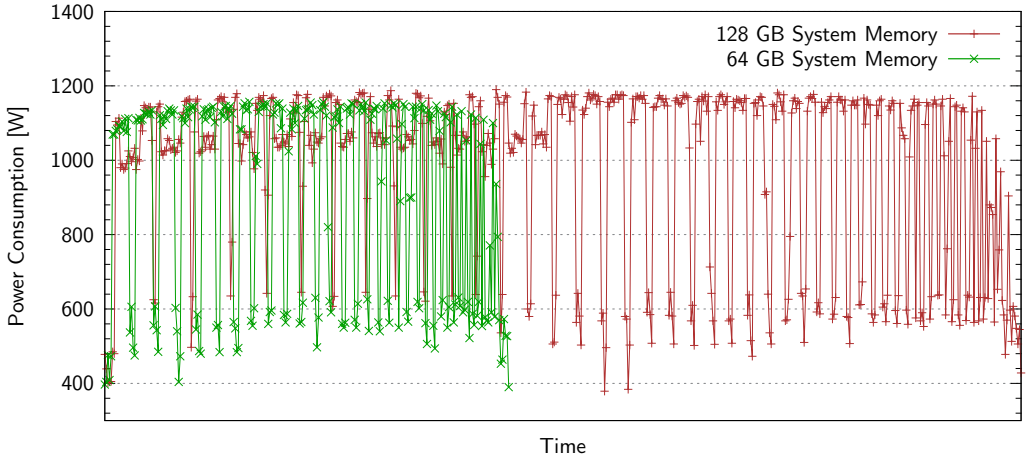


Figure 12.28: Power Consumption during Multi-GPU HPL Run [VI]

12.7.3 Improvements by more efficient Hardware

Highest performing enthusiast hardware is usually not the most power efficient one. More power efficient hardware increases the performance per watt while decreasing the overall performance of the system. Recall that multi-GPU usage increases the power efficiency as the relative CPU contribution to the overall power consumption decreases with more GPUs. Clearly, the usage of more efficient (but slower) GPUs weakens this effect. Thus, very low-power GPUs do not help, at least as long as high performance CPUs are employed. In addition, the gain in power efficiency is larger when switching from high-end to middle-class than when switching from middle-class to low-end / low-power. This opens two ways to design energy-efficient systems:

Using both low-power GPUs and CPUs **S**plitted **D**esktop **S**ystems (SDS) has designed a system according to this approach, which remains within a total power envelope of below 200 W. Using the special HPL versions for slow CPUs (see Section 12.11) and special optimizations for the system, which were developed in cooperation with SDS, an efficiency of **1.12 GFlop/J** was achieved [SDS+ 11, Fra 11]. In contrast to the LOEWE-CSC nodes, huge pages (see Section 11.2.5.1) have a positive effect.

Using multiple middle-class GPUs For the second approach, the three 5870 GPUs in the reference system are substituted by lower clocked V7800 GPUs. In addition, the low-voltage Hynix memory running at 1.35 V is exchanged by special low-power Samsung memory. The Samsung memory is fabricated using a 30 nm process and saves about 10 W in total, even though running at the same voltage. There is one more possibility to reduce the power consumption: The CPU contribution to the DGEMM is very small while the CPUs drain much more power when fully loaded than when idling. In a second attempt the work is thus offloaded from the CPU as far as possible, i. e. the CPU processes only the remainder from the GPU DGEMM tiling but all other CPU DGEMM phases are skipped. This lets several CPU cores idle for about 50 % of the time. Still, a fast CPU is required to ensure a fast factorization. Table 12.29 lists all results.

Configuration	HPL Result	Average Power	Efficiency
5870 / Hynix Memory	1114 GFlop/s	954.1 W	1.168 GFlop/J
*V*7800 / Samsung Memory	983 GFlop/s	820.6 W	1.198 GFlop/J
*V*7800 / Low CPU Load	949 GFlop/s	766.8 W	**1.238 GFlop/J**

Table 12.29: Power Efficiency reached with more efficient Hardware [VI]

These values can be extrapolated to multi-node runs relevant for the Green500. It is required to add the power consumption of the InfiniBand network adapter and to adjust the HPL performance for the multi-node efficiency. On the LOEWE-CSC, the multi-node network-efficiency is reduced to 93.2 % compared to single-node results. Assuming 12.2 W for the InfiniBand interconnect as specified by Mellanox [Mel 10], this yields an estimation for the multi-node power efficiency of 1.136 $^{\text{GFlop}}/_{\text{J}}$. This would correspond to a second place in the November 2010 Green500 list, which was the most recent list at the time the experiment was conducted.

12.8 AMD 6000 Series GPU

The kernel performance for the Cayman GPU (6000 series) has been optimized in the same way as for Cypress (5000 series, see Section 11.2.4.1). For each blocking scheme, the best parameters such as unrolling factor, matrix size, etc. are determined experimentally. As for the 5000 series, the 4×4 B transposed kernel turns out to be the fastest. Fig. 12.30 shows the performance depending on the matrix size on an AMD Radeon HD 6970 GPU with Cayman chip.

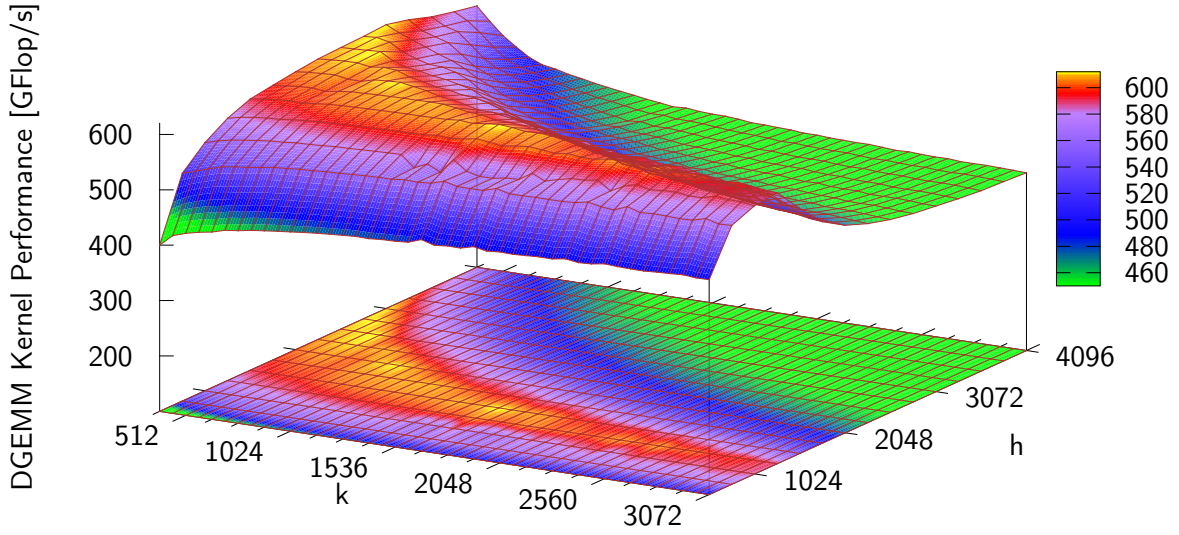


Figure 12.30: DGEMM Kernel Performance for different Matrix Sizes on 6970 [VI]

The GPU reaches a very good kernel performance of 617 $^{\text{GFlop}}/_{\text{s}}$ at $k = 1024$ and $h = 2304$. For both larger h and larger k, the performance decreases rapidly. The Cayman GPU suffers from a DMA problem, which is explained and solved in Section 12.8.2. However, even with this DMA problem fixed, the system performance was limited to 412 $^{\text{GFlop}}/_{\text{s}}$ at first, which is not consistent with the kernel performance of about 600 $^{\text{GFlop}}/_{\text{s}}$.

12.8.1 Temperature & Power

The Cayman GPU throttles itself under two conditions: high temperature and high power consumption that exceeds its **T**hermal **D**esign **P**ower (TDP). While throttling due to temperature can be monitored and excluded easily, it is hard to monitor the actual power consumption. Even further, the GPU management tools do not even report that the GPU throttles due to power consumption. The throttling starts after a short delay. This explains the performance drop in Fig. 12.30 for large h or k: if kernel execution time exceeds the throttling delay, the kernel is slowed down. It also explains the low system performance: system performance measurements involve various kernel executions organized in a pipeline. While the first kernel (with medium h and k) achieves the full performance, all following executions are slow, and so is the system performance.

The AMD ADL library [Adv II] can set voltage and clock rates for all AMD GPU operating modes. Having raised the clocks of the throttled mode to the stock clocks, the kernel works perfectly. Fig. 12.31 shows the kernel performance again, this time without any throttling.

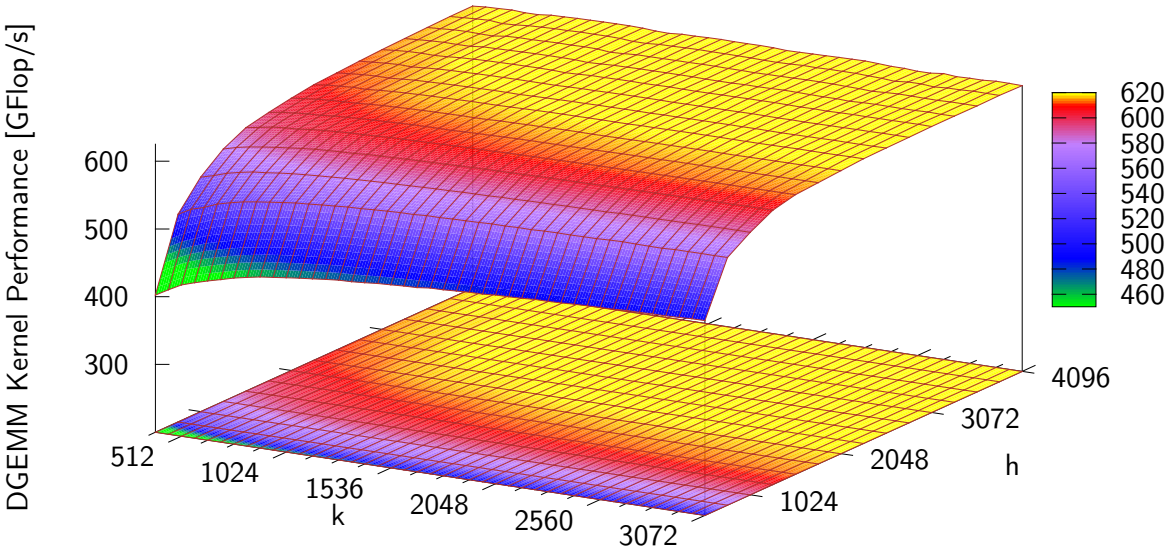


Figure 12.31: DGEMM Kernel Performance for different Matrix Sizes on 6970 with fixed Clocks [VI]

The Cayman GPU shows an absolutely flat performance distribution as soon as the matrix reaches a certain minimal size. The optimum of 624 $^{\text{GFlop}}/_{\text{s}}$, which is 92.3 % of the peak performance, is found at matrix sizes which are suitable for HPL. In contrast, the fastest kernel on the 5000 series uses $k = 512$ where the overhead is too big. Unlike the 5000 series, the best performing Cayman kernel writes to GPU device memory. The kernel writing directly to host memory achieves only 610 $^{\text{GFlop}}/_{\text{s}}$.

12.8.2 DMA Performance

Unfortunately, the 6970's initial system performance lacked far behind its kernel performance. The first implementation achieved 316.4 $^{\text{GFlop}}/_{\text{s}}$, 50.7 % of its kernel performance – even after the power-throttling had been deactivated. (In contrast, Table 11.58 shows that the 5870 reaches 98.6 % of its kernel performance.) Table 12.32 demonstrates that asynchronous kernel execution and DMA transfer take as long as the sum of kernel and DMA transfer time, regardless of the transfer direction and the order of execution. Apparently, in contrast to Cypress, asynchronous DMA does not work. (For measurements on Cypress, see Table 11.24.)

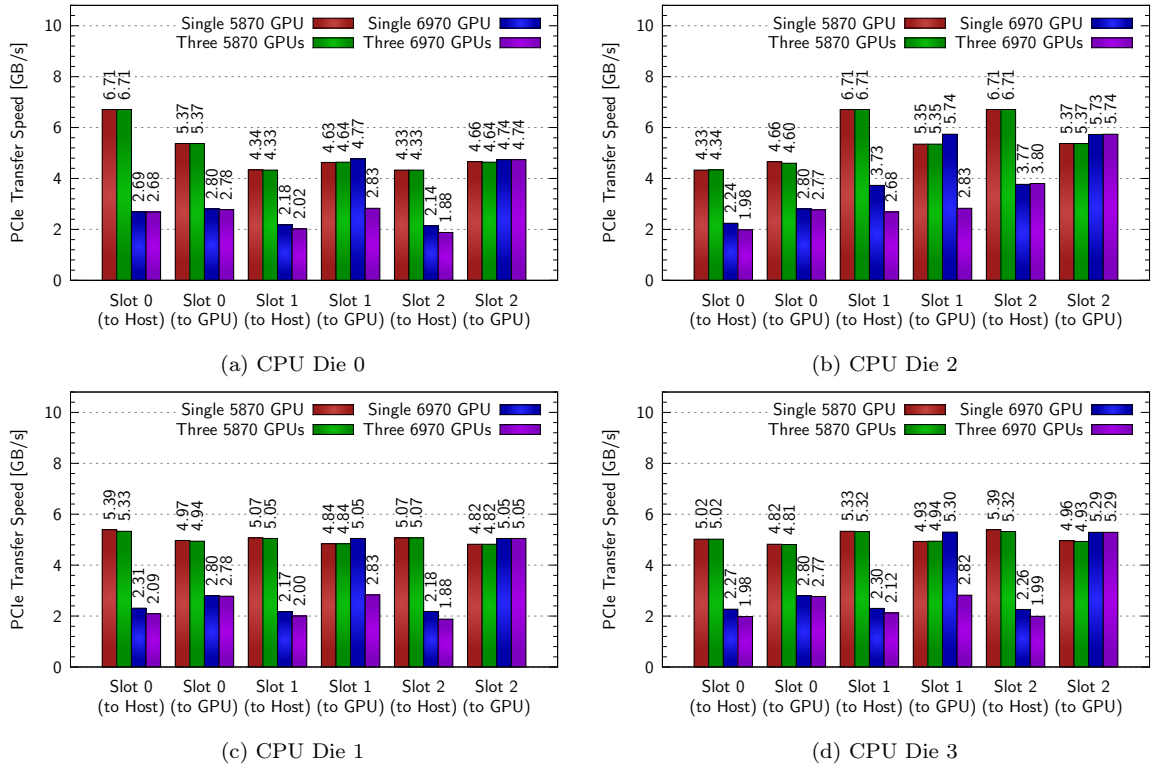
Operation	Time (GPU to Host) [s]	Time (Host to GPU) [s]
Kernel Execution	0.103	0.103
DMA Transfer	0.116	0.097
Combined (Kernel started first)	0.208	0.185
Combined (DMA started first)	0.207	0.185

Table 12.32: Asynchronous CAL DMA Transfer on 6970 [VI]

This lack of DMA functionality leads to very poor transfer speeds as shown in Fig. 12.33, especially when multiple GPUs are installed in one host. The PCI Express bandwidth of a single GPU is measured for every PCI Express slot. The triple-GPU measurements differ from the single-GPU

measurements in the way that all slots are populated. In both cases, the bandwidth to one GPU is measured. The 5000 series DMA performance fulfills the expectations and does not depend on whether one or multiple GPUs are present. In contrast to Cypress, the 6000 series shows irregular effects and poor performance except for the transfer to the GPU in slot two.

In order to achieve good performance, one should allocate the page-locked memory on the CPU die which is closest to the GPU. In multi-GPU environments, this die can vary with the GPU. One can see that slot 0 is connected to CPU die 0 while slots 1 and 2 are connected to CPU die 2.[14]



(a) CPU Die 0

(b) CPU Die 2

(c) CPU Die 1

(d) CPU Die 3

Figure 12.33: 5870 and 6970 Multi-GPU DMA Performance[14] [VI]

12.8.3 Workaround for the DMA Issue

With the help of AMD, the behavior could be understood: the DMA engine of the Cayman chip is unable to handle the type of DMA transfer CALDGEMM performs and the driver handles the transfer either via the CPU or via an artificial DMA transfer kernel. Workarounds exist for the transfer to both the GPU and the host. The approaches for the two cases are totally different.

12.8.3.1 Improving GPU to Host Transfer

Since only the DMA engines do not work properly, the DGEMM kernel version writing directly to host memory, although not the fastest on the 6970, is tested. For the moment, the transfer to the GPU remains an explicit DMA engine transfer and thus synchronous. Table 12.34 shows that this improves the 6970 system speed enormously. The 14 $^{\text{GFlop}}/_{\text{s}}$[15] the employed kernel lacks behind the fastest version is a price one has to pay.

[14] The relation which GPU is connected by which CPU die becomes clear when one searches for the CPU die and PCI Express slot combinations that yield the highest bandwidths.

[15] See Section 12.8 for a comparison of the kernels.

GPU	Transfer of Output	System Performance	Kernel Performance
6970	Explicit Transfer after Kernel	316 GFlop/s	624 GFlop/s
6970	Transfer via DMA by Kernel	599 GFlop/s	610 GFlop/s
5870	Explicit Transfer after Kernel	460 GFlop/s	465 GFlop/s
5870	Transfer via DMA by Kernel	465 GFlop/s	472 GFlop/s

Table 12.34: Workaround for 6970 GPU to Host DMA Issue [VI]

Comparing the workaround for the 6000 series with the workaround for the Intel DMA issue (Section 12.5.2) reveals that they are mutually exclusive to each other. As both workarounds are mandatory on the respective platforms, it is currently not possible to reach good performance using Cayman GPUs in combination with an Intel chipset.

12.8.3.2 Improving Host to GPU Transfer

The transfer from the GPU back to the host has been improved by making the DGEMM kernel perform remote memory access via Zero-Copy. Of course, in the other way around, the kernel could read its input from host memory via Zero-Copy, too. However, this limits the memory bandwidth to the PCI Express speed, not to mention the latency. Thus, a different approach is needed.

64-bit Kernels It turns out that the problem with the DMA engine affects only 128-bit (*double2*) transfers to tiled buffers. Both linear buffers and 64-bit (*double*) transfers work well. In tiled mode, the data in the buffer are stored in a non-linear format. The pattern for buffers storing *double*-values and *double2*-values is different. Hence, a 64 bit transfer cannot be used to fill a 128 bit buffer. Since in Section 11.2.4.1 the linear buffer was discarded for its poor performance, as a last resort, the CALDGEMM kernel is modified to read from 64-bit buffers. Two approaches are implemented:

1. The buffers are scaled by a factor of two in width (in terms of elements). The linear data format in the host is unchanged. This is possible since a buffer storing n *double2* elements can be interpreted as a buffer storing $2n$ *double* elements. The 128-bit texture fetch for the *double2* element at coordinate (x, y) is replaced by two 64-bit fetches for *double* elements at coordinates $(2x, y)$ and $(2x + 1, y)$.

2. The buffer size (in terms of elements) remains the same but the number of buffers is doubled. The 128-bit fetch for the (x, y) element of buffer i is replaced by two fetches for the (x, y) elements of buffers $2i$ and $2i + 1$. This approach requires a change to the *DivideBuffer* function as the linear storage is changed. In exchange, the access pattern to the 64-bit buffer is identical to the old pattern, which is known to perform well.

Table 12.35 compares the kernel performance of both approaches with the original kernel performance using 128-bit buffers. To neglect side effects by the DMA transfer, the output is stored on the GPU. Both approaches are underwhelming.

Input Format	Kernel	Performance
128-bit	Reference Kernel	610 GFlop/s
64-bit	Approach 1	335 GFlop/s
64-bit	Approach 2	334 GFlop/s

Table 12.35: Workaround for 6970 Host to GPU DMA Issue [VI]

A measurement of the raw cache bandwidth helps to understand why the 64-bit kernels perform so badly. This can be done by continuously fetching data from the same address ensuring a cache hit ratio of 100 %. In order to suppress compiler optimizations like dead code elimination, a constant offset of 0 stored in the constant buffer is added after each texture fetch. This offset is not known at compile time. To allow for coalescing (combining multiple memory accesses [Roh 10 I, 3.7]), the test is repeated fetching 1×1, 1×2, 2×1, and 2×2 areas of the texture. Every time the resulting bandwidth corresponds precisely to the bandwidth required for 335 $^{\text{GFlop}}/_{\text{s}}$ in the DGEMM.

As a conclusion, it is not possible to achieve a good DGEMM performance with 4×4 blocking and 64-bit texture fetches. For Intel processors it is known that the actual number of fetches to the L1 cache is limited, no matter whether 64 or 128 bits are fetched. In fact, the processor's L1 cache is limited by the throughput in fetches not the throughput in bytes. Most probably the same holds for the Cayman GPU which means that 64-bit accesses inevitably halve the available texture cache bandwidth explaining the above results. The 64-bit kernels were thus dropped.

64-bit to 128-bit Conversion Kernels According to the last paragraph, either the blocking size must be increased or one must combine 128-bit kernels with 64-bit DMA transfers. The latter can be reached by introducing an additional conversion kernel in the pipeline, which copies the data from a 64-bit DMA buffer to a 128-bit buffer, which is then used by the DGEMM kernel. After all, the conversion time is almost negligible. As an example: assuming $k = 1024$ and $h = 4096$ the buffer size is 64 MB. Considering the GPU memory bandwidth of more than 100 $^{\text{GB}}/_{\text{s}}$, this can be done in few milliseconds,[16] while the DGEMM kernel execution takes orders of magnitude longer. On top of that, the conversion is not necessarily repeated prior to each kernel invocation since the kernel input data are cached on the GPU anyway (see Section 11.2.4.6). Clearly, it makes sense to cache the converted input data. The implementation is straight forward but involves rather complex scheduling since the buffer transfer must happen two iterations prior to the kernel execution (see Section 11.2.4.6). Section 12.12 contains more details and an in-depth comparison of all DMA data paths.

Results In order to verify that the additional overhead has only a small impact on the performance, the results with and without the conversion kernel are compared on the Cypress hardware. There, the performance loss is below 1 %. The optimization of the host to GPU transfer is mostly relevant for small matrices. (For large matrices the transfer time is almost negligible due to the caching.) Table 12.36 shows that the highest achievable performance increases from 599 $^{\text{GFlop}}/_{\text{s}}$ to 604 $^{\text{GFlop}}/_{\text{s}}$. The system performance comes very close to the kernel performance and the efficiency is even higher than for the Cypress GPU (compare Table 11.58).

Measurement	Performance [$^{\text{GFlop}}/_{\text{s}}$]	Efficiency [%]
Theoretical GPU Peak Performance	675.8	
DGEMM Kernel Performance (Best Kernel)	623.5	92.3
DGEMM Kernel Performance (DMA Workaround)	610.0	90.3
DGEMM System Performance	603.5	89.3

Table 12.36: Final AMD 6970 single-GPU DGEMM Performance [VI]

12.8.4 6000 Series Multi-GPU DGEMM & HPL Performance

The tests in this section are performed on a 2.3 GHz 24-core Magny-Cours system with two AMD 6990 dual-GPUs, i. e. four GPU chips. The following optimizations and adjustments have been made on top of those required for three 5870 GPUs in Sections 12.6.2 and 12.6.4.

[16] Measurements show varying results between 2 ms and 6 ms for the conversion time.

- Even more than in Section 12.6.2.1, memory bandwidth poses a problem. For achieving the best DGEMM performance, k has to be increased to 2560. For HPL, such a large block size is infeasible. Thus, $N_b = k = 2048$ is used despite the reduced DGEMM performance.
- Especially for single-GPU DGEMM, optimal performance is achieved by allocating GPU related memory on and pinning pre- and postprocessing threads to a CPU die close to the GPU. For four GPUs this paradigm changes entirely. The two GPU boards are connected to two CPU dies, but these two dies cannot deliver enough memory performance. Hence, it is essential to accumulate as much aggregate bandwidth as possible. CALDGEMM is modified to allow for a more fine-granular pinning, which allows for explicitly and individually setting the CPU core for all GPU related threads and for memory allocation. The best-performing option is then determined experimentally. The author would like to thank Petr Borodkin who contributed a lot for finding the best pinning. The optimal parameters are: memory is allocated on dies 0 and 2, closest to the GPU. Preprocessing is performed on dies 0 and 2 as well, but postprocessing is performed on dies 1 and 3, thus accessing the GPU memory via HyperTransport. For standalone DGEMM, four preprocessing threads (one per GPU) deliver additional $10\,\mathrm{GFlop/s}$ compared to two threads. Still, only two threads (one per dual-GPU) are better in HPL since they block less CPU resources.
- The pinning of the CPU threads is modified such, that during the phases with reduced thread counts (factorization and LASWP during lookahead), the loaded threads are distributed equally among the NUMA nodes maximizing the available memory bandwidth.
- Lookahead 2 is now available in multi-GPU runs after the pipeline has been modified to be compatible with multiple GPUs. As before, lookahead is running in adaptive mode, i. e. both lookahead 1 and 2 are disabled as soon as this shortens the iteration time.
- DGEMM on four GPUs is so fast that the CPU processing time for the remainder part of the matrix becomes a bottleneck. Especially due to the availability of lookahead 2, it is essential to have enough free CPU resources. CALDGEMM has been modified such that it can process arbitrary non-square tiles, still with the restriction that tile dimensions are a multiple of 128. This downsizes the remainder part of the matrix.
- In contrast to the LOEWE-CSC nodes, the test system gains additional $67\,\mathrm{GFlop/s}$ if the huge pages option introduced in Section 11.2.5.1 is enabled.
- At the very end of the run, the performance is dominated by the factorization. During this phase, the block size N_b is halved to 1024. (This is impossible for multi-node tests.)
- The system is running very close to its memory bandwidth limit. Parallel execution of GotoBLAS on the processor costs more GPU performance (because of memory load) than the processor can contribute and thus overcompensates the advantage. Thus, the CPU load is reduced as much as possible. The matrix is not split in a CPU and a GPU part, no second and third phase runs are started. The processor computes only the absolutely necessary remainder part of the matrix (which can be chosen very small due to the non-square tiles) and, of course, it runs LASWP, DTRSM, and factorization for the lookahead.
- The HPL factorization parameters (see Section 11.3.2) can be tuned for lower memory bandwidth requirements instead of best factorization performance. Since the CPU does not contribute to the DGEMM, increased factorization time does not affect the total execution time as long as it is hidden by lookahead. In contrast, the GPU DGEMM can benefit from the reduced memory load, i. e. HPL overall performance increases even though factorization performance decreases. With each HPL-iteration factorization time becomes more and more important. As soon as the lookahead implementation can no longer hide the factorization time, instead of turning off lookahead earlier, the parameters are modified gradually to take the factorization performance to the highest possible level ensuring that lookahead can remain active as long as possible before it is deactivated.

Fig. 12.37 shows the total iteration time for all lookahead modes (the adaptive mode time equals the minimum and is not shown for simplicity) and the duration when GPU and CPU respectively are active. Starting from the left side of the diagram, GPU DGEMM time exceeds the CPU time and the total iteration time equals the GPU time. Factorization parameters are optimized for fastest GPU DGEMM. At iteration 26 they are changed to speed up the factorization (slightly slowing down the DGEMM), to keep the GPU the dominant part as long as possible. Afterward, the total iteration time follows the CPU time.[17] Lookahead 1 is never the fastest mode. In iteration 49 CALDGEMM switches directly from mode 2 to mode 0, speeding up both GPU and CPU processing but serializing them.

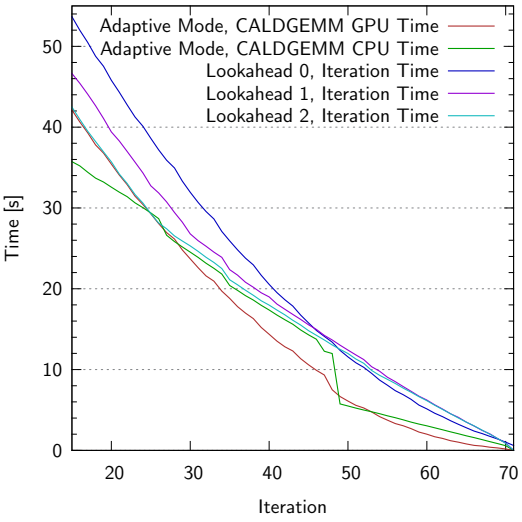


Figure 12.37: Multi-GPU Lookahead and Factorization Parameter Analysis[17] [XIII]

Table 12.38 summarizes the achieved performance. The 6990 has slightly lower clocks than the 6970. Hence, single-GPU results are included for comparison with Table 12.36. The four GPUs, compared to the peak performance, achieve an outstanding efficiency in DGEMM, which exceeds the efficiency of the triple-GPU measurements by far (compare to Section 12.6.3). This has three reasons: the k-parameter is bigger, the CPU is not running DGEMM in parallel and thus also not considered when calculating the DGEMM peak performance, and the employed processors have a slightly higher clock rate than the ones in Section 12.6.2.3. The scalability (which for a fair comparison includes the inactive thus wasted CPU cores) calculates to 94.8 %. With the HPL parameter $k = 2048$, the DGEMM achieves 2240.7 $^{\text{GFlop}}/_{\text{s}}$. HPL can maintain 80.1 % of the DGEMM performance – which is almost identical to the percentage of the triple-GPU HPL measurement.[18]

Measurement	Performance [$^{\text{GFlop}}/_{\text{s}}$]	Efficiency [%]
Theoretical GPU Peak Performance	637.5	
Single-GPU DGEMM System Performance	573.7	90.0
Single-GPU / CPU DGEMM System Performance	729.4	85.0
Quad-GPU DGEMM System Performance	**2291.9**	89.9
Quad-GPU HPL Performance	**1794.0**	64.7

Table 12.38: AMD 6990 Multi-GPU DGEMM and HPL Performance [XIII]

As a final resort to squeeze out the highest possible performance, the memory in the test-node is doubled to 256 GB using 16 GB DIMMs. Naturally, the following results cannot be compared to any of the previous measurements and, in addition, the now available 10.67 GB of RAM per CPU core exceed the application demands by far and are thus no viable option for a compute cluster. Previously, the feasible block size range has been bound by 2048 due to the N_b^3 time dependency of the factorization. Now, the huge memory allows to increase the block size to $N_b = 2560$ because the CPU dominated period is less significant. Table 12.39 shows that in this extreme situation the lookahead implementation works very well, with the adaptive lookahead coming out fastest again achieving 2007 $^{\text{GFlop}}/_{\text{s}}$. Appendix G lists the new features and recommended settings.

[17] Between around iteration 30 to 40, iteration time and CPU time do not match completely. Because of the lookahead, the CPU time includes the factorization of the next iteration which is faster than the factorization of the current iteration.

[18] For comparison: single-GPU HPL achieves 90.3 % of single-GPU DGEMM performance (see Table 11.58).

Lookahead	Performance [GFlop/s]	Improvement [%]
No Lookahead	1610	
Lookahead 1	1792	11.3
Lookahead 2	1948	21.0
Adaptive Lookahead	**2007**	24.7

Table 12.39: AMD 6990 HPL Performance with 256 GB RAM [XIII]

12.9 CALDGEMM for Interlagos / Sandy Bridge and without GotoBLAS

To run GPU DGEMM, CPU DGEMM, and HPL tasks such as factorization, pivoting and broadcast in parallel, a patch to the GotoBLAS library enables the reservation of CPU cores from GotoBLAS in order to use them dynamically for multiple tasks. Without this feature, it is likely that e. g. GPU postprocessing and a GotoBLAS DGEMM thread would run on the same core leading to miserable performance. In order to make other BLAS libraries compatible with CALDGEMM, the patch must be ported. This, however, is hardly possible if the source code of the BLAS library is not publicly available.

Meanwhile, development of GotoBLAS has been discontinued and the latest version cannot use the AVX vector extensions of recent processors like Interlagos or Sandy Bridge. While the Nehalem version of GotoBLAS still achieves about half the peak performance on Sandy Bridge using SSE, the Magny-Cours version is incompatible with Interlagos because AMD dropped the 3DNow! [Adv 10 I] extensions for the Bulldozer CPU family.[19] Both vendors offer proprietary BLAS libraries, Intel MKL and AMD ACML [Adv I], specifically tuned for these new processors. Unfortunately, since the sources are unavailable, the patch cannot be adapted easily for these BLAS implementations.

In contrast to GotoBLAS, most BLAS libraries do not come with their own threading implementation, but rely on available threading APIs, e. g. ACML employs OpenMP [OMP]. When entering the first OpenMP parallel section, OpenMP spawns as many threads as are required. The following parallel sections reuse these threads, or spawn new ones if more threads than available are requested. If a parallel section is distributed on less threads than present, it uses the lower numbered ones, e. g. if 24 threads are started and a section requires 18 threads, it runs on threads 0 – 17.[20,21] This provides an opportunity for reserving threads from OpenMP based binary libraries. With respect to the core reservation, there are three phases in an HPL run:

I No parallel execution of tasks, e. g. during factorization without lookahead.
II Parallel execution of GPU DGEMM and BLAS, e. g. during factorization with lookahead, or simply during combined GPU / CPU DGEMM.
III Parallel execution of GPU DGEMM, CPU DGEMM and MPI broadcast, during broadcast with lookahead.

Phase I does not require core reservation, in phase II all GPU related cores must be reserved, and in phase III an additional broadcast core must be put aside. For this purpose, at first an OpenMP parallel section with as many threads as CPU cores is started ensuring that all threads are spawned. Assume n CPU cores, g GPU related threads, and one broadcast thread. OpenMP threads $n - g$ to $n - 1$ are pinned to the same cores as the GPU pre- and postprocessing threads, thread $n - g - 1$ is pinned to the same core as the broadcast thread. Every

[19] It is possible to run the Nehalem version of GotoBLAS on Bulldozer CPUs, like the Interlagos, but since the timing of the SSE instructions and the cache size is different on the AMD processor, this does not yield good performance.
[20] The main thread which starts the OpenMP parallel section is thread 0.
[21] Although the OpenMP specification does not explicitly enforce the described thread selection, all tested OpenMP implementations behave this way. Unfortunately, it cannot be ensured for new implementations.

time HPL enters a different phase, CALDGEMM sets the number of threads that shall be used using the *omp_set_num_threads* API function. The above pinning ensures that at no time two threads are executed on the same core. For instance, during phase II, BLAS runs with $n - g$ threads (0 to $n - g - 1$), thus using the broadcast core (thread $n - g - 1$) but not the GPU related cores (since threads $n - g$ to $n - 1$ are inactive). The workaround is not as sophisticated as the GotoBLAS patch because in every additional phase only more and more but not less cores can be reserved. (For instance, the broadcast core could not be reserved on its own.) Still, it works for HPL and has been implemented for the OpenMP [OMP] and TBB [Int] threading APIs.

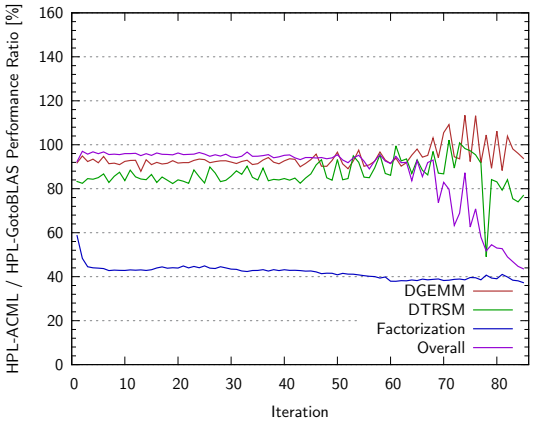


Figure 12.40: Relative ACML Performance (compared to GotoBLAS) of multiple Tasks during HPL on Magny-Cours [VI]

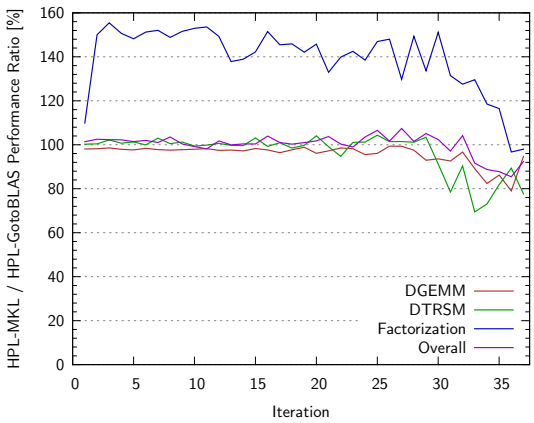


Figure 12.41: Relative MKL Performance (compared to GotoBLAS) of multiple Tasks during HPL on Nehalem [II]

For directly comparing the performance of ACML and GotoBLAS, a Magny-Cours system is used, which is supported by both libraries. Fig. 12.40 shows the performance ratios of HPL tasks employing ACML and GotoBLAS. The trailing matrix size goes with $1/\text{iteration}^2$, thus the rightmost part involves only tiny matrices where the matrix size can have a large effect on the BLAS performance, especially on many-core processors, explaining the large variations in the plot. Though, this does not affect HPL performance much since it applies only to a very small fraction of the overall execution time. Since ACML produces more memory load than GotoBLAS, one more postprocessing thread per GPU must be used. This reduces the available compute resources for ACML by 4.5 %. In DGEMM, ACML is almost as fast as GotoBLAS (in the majority of the time it achieves about 93 % of the GotoBLAS performance.); in DTRSM, the difference already becomes significantly larger; and during the factorization, ACML hardly achieves half the GotoBLAS performance. One reason is that the factorization includes various BLAS calls on tiny datasets, where the OpenMP overhead for the threading weights much more than the overhead of GotoBLAS, which has its own fine-tuned threading implementation. Another reason is that the GotoBLAS optimizations in Section 12.6.4.2 are missing. For the overall HPL performance, which is dominated by GPU DGEMM, the factorization does not play an important role as long as it is hidden by the lookahead. Finally, the ACML run achieves 524.5 GFlop/s HPL performance while the GotoBLAS version reaches 574.4 GFlop/s.

Fig. 12.41 shows that in contrast to ACML, MKL is only minimally slower than GotoBLAS in DGEMM, but faster in any other discipline.[22] During the very first factorization iteration, the GPU DGEMM is not active. In this iteration, the performances of ACML, MKL, and GotoBLAS do not differ so much. As soon as the GPU DGEMM causes heavy memory load, ACML becomes much slower compared to GotoBLAS while MKL gets much faster. This shows that MKL is least and ACML is most affected by memory load. Single-GPU HPL performance of GotoBLAS and

[22] In this case, a Nehalem system is used as common basis.

MKL is alike. Since factorization performance is the bottleneck for multi-GPU systems, the MKL library, especially in combination with future AVX processors, forms an auspicious solution.

12.10 Performance Limits & Exceeding Peak Performance

Although DGEMM reaches almost peak compute performance, the main obstacle during its development is limited memory bandwidth not limited instruction throughput. The optimization task lies mainly in ensuring good cache utilization. The achieved kernel performance with optimal cache utilization can be measured by altering the addressing such that always the same matrix entry is fetched. This results in 100 % cache hit ratio. Obviously, this corrupts the DGEMM result and can be used only for theoretical considerations. In order to force the compiler to create actual texture fetches, an offset of 0 stored in the constant buffer, which is unknown at compile-time, is added to the address in the texture (in the following called **Constant Texture Fetch**). With the constant texture fetch, the L1 cache bandwidth still limits the performance. The raw computation performance can be measured by omitting the memory access completely (**No Texture Fetch**).

The 5000 series offers 5D-VLIW-shaders and combines four of the single-precision ALUs of one 5D-shader to a double-precision ALU. Therefore, each shader can process a single and a double precision instruction simultaneously. It is well known [Hid$^+$ 01] that a double precision FMA-instruction can be simulated using eight single precision instructions. Hence, the $5D$-shader can perform 9 double precision instructions in 8 cycles. (Unfortunately but obviously, the result is not compliant with the IEEE double precision floating point specifications (IEEE 754) and thus such an optimization is not allowed for HPL.) Still, this trick boosts the theoretical peak performance of the 5870 by one eighth to 612 $^{\text{GFlop}}/_{\text{s}}$ exceeding the official double precision peak performance of 544 $^{\text{GFlop}}/_{\text{s}}$. Since the cache bandwidth limits the 4×4 kernel to 544 $^{\text{GFlop}}/_{\text{s}}$ (Section 11.2.4.1), only the test without texture fetch – if any – has the possibility to go beyond this. Table 12.42 shows the results. All measurements use 4×4 blocking at $k = 1024$.

Cache	Mode	Performance [$^{\text{GFlop}}/_{\text{s}}$]
Regular	Double Precision	469
Constant Texture Fetch	Double Precision	484
No Texture Fetch	Double Precision	533
Regular	Double Precision & Emulation	459
No Texture Fetch	Double Precision & Emulation	545

Table 12.42: Synthetic DGEMM Peak Performance Analysis [VI]

Comparing the regular results with the results for constant texture fetches demonstrates that the loss due to cache misses is only 3.1 %. Omitting the texture fetches entirely and using the double-precision emulation technique exceeds the theoretical peak performance by 1 $^{\text{GFlop}}/_{\text{s}}$. Unfortunately, with regular texture fetches the emulation technique is even counterproductive. The cause is the compiler, which reorders the instructions such that the 5D-shaders are not used efficiently. Due to these compiler problems, the limit by the L1 cache bandwidth, and the disappearance of the 5D-shaders in the new generations, the idea was discontinued.

12.11 Systems with a slow CPU

The most critical task on systems with a GPU but a slow CPU is offloading work to the GPU. CALDGEMM accomplishes this by omitting the second and third phase CPU DGEMM runs and by utilizing non-square tiles (Section 12.8.4), such that the overlap processed by the CPU is minimal. In addition, the CPU pinning is adjusted. If there are insufficiently many CPU

cores available, the tasks without a dedicated CPU core are assigned to the CPU core running *DivideBuffer* since this routine in general requires fewer resources than *MergeBuffer*. With these changes, e. g. a dual core AMD CPU clocked at 2.2 GHz in combination with a 5870 GPU[23] and 32 GB RAM reaches 460 GFlop/s of DGEMM performance.

Achieving good HPL performance on such systems is a bigger challenge. In contrast to DGEMM, the work cannot easily be offloaded completely to the GPU. Especially the factorization, the pivotization, and the DTRSM pose a problem. Since small systems usually do not offer a large amount of main memory, the following tests are restricted to $N = 24576$ corresponding to a 4.8 GB matrix.

The above described optimizations for CALDGEMM increase the HPL performance of the dual core AMD system with a 5870 GPU from 46.25 GFlop/s to 118.3 GFlop/s. The DTRSM can be replaced by a DTRTRI (inversion of triangular matrix) and a subsequent DGEMM call where the DGEMM can easily run on GPU. The DTRTRI remains on the CPU, but its complexity is $\mathcal{O}(N_b^2)$ in comparison to $\mathcal{O}(N \cdot N_b)$ for DTRSM. This update further increases the performance to 138 GFlop/s. The high energy efficiency of the SDS system in Section 12.7.3 is reached by, in addition, offloading medium DGEMM calls during the factorization to the GPU.

12.12 Overview of CALDGEMM DMA Paths

This section gives an overview over relevant DMA paths which have been introduced throughout this thesis. Only the scenarios with practical applications are presented. Ideas which have been dropped are not repeated. The C-matrix is split in tiles, which are distributed among the GPUs. Fig. 12.43 visualizes the situation. Roughly, the tiling scheme is such that the B-matrix is split and each GPU processes a part of the B-matrix. (The detailed multi-GPU scheme is presented in Section 12.6.2.3.) All buffers are replicated for each GPU. The only difference is that each GPU stores a different part of the B-matrix. Therefore, only one GPU is treated in the following.

In general, there exist two DMA paths for the input and two paths for the output. As they are unrelated, they are discussed one after the other. In any case, the framework circumvents the DMA driver issue by initiating DMA engine transfers always two iterations prior to data usage.

Input DMA Paths Each submatrix (part of A and B respectively needed for the current tile) is split in two halves and, for 4×4-tiling, stored in two buffers. This reduces the register requirement and optimizes the cache access patterns. These two buffers always belong together and are shown on top of each other in Fig. 12.43. To simplify the notation, the word **buffer** in the following refers to such a pair. For the A-matrix, two buffers (buffer-pairs) exist on both the GPU and the CPU side. They are used in a round robin fashion such that the next buffer can already be filled while the previous buffer is transferred to the GPU. The full A-matrix is not stored on the GPU. Instead, one tile of the A-matrix is replaced as soon as it is no longer needed. The situation for the B-matrix is different. While on the host side only two buffers exist, too, on the GPU side as many buffers as possible are allocated. This allows for storing the entire B-matrix (or at least a big fraction of it) on the GPU. When iterating all B-tiles over a fixed A-tile, no (or fewer) retransfers are needed. In case the matrix does not fit entirely in GPU memory, the first two buffers are used in a round robin fashion for the remaining tiles.

DMA Path 1a This path involves solely 128-bit buffers. The data are prepared by the *DivideBuffer* function and then asynchronously transferred to the GPU by the DMA engine. If supported by the hardware, this is the best solution.

DMA Path 1b This path stores the output of *DivideBuffer* in a 64-bit buffer and performs a 64-bit DMA transfer to a temporary buffer on the GPU. Afterward, the conversion

[23] Slow CPU benchmarks are performed on [VI] by restricting Linux to two CPU cores using a kernel setting.

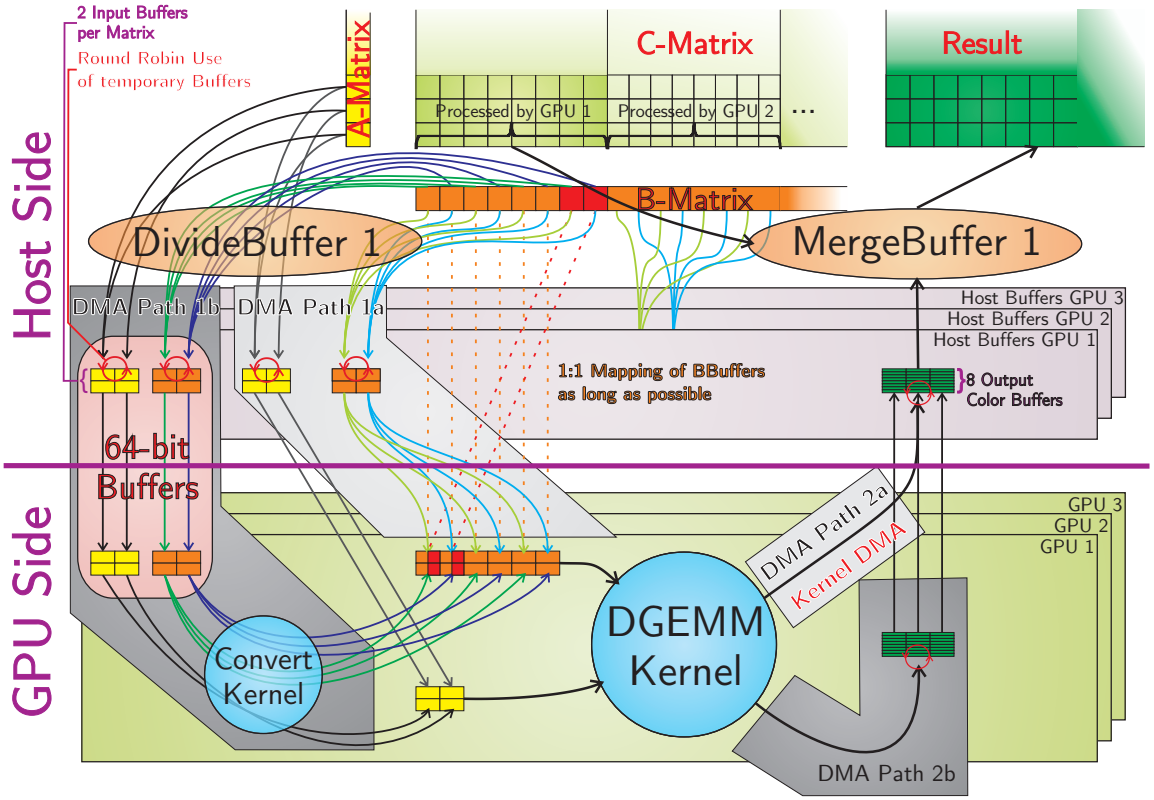


Figure 12.43: DMA Paths, Buffers, and Workflow of CALDGEMM

kernel converts the data to the 128-bit format. As for path 1a, data are then stored in 128-bit format on the GPU until they are no longer needed. Therefore, as long as sufficient BBuffers exist, neither retransfer nor repeated conversion are needed. This allows the kernel to read *double2* entries while enabling asynchronous DMA transfer on devices such as the 6000 series. If 128-bit DMA is supported by the hardware, this is clearly inferior to path 1a due to the additional overhead.[24]

Output DMA Paths The output buffers are grouped in eight buffers per submatrix shown beneath each other again. As for the input, such a group is simply denoted as buffer. The situation is less conclusive than for the input path. In DMA path 2a, the kernel stores the results directly to host memory using Zero-Copy. In path 2b, the output is first stored to a temporary buffer on the GPU and then transferred to the host by the DMA engine. A trick described in Section 12.5.2 automatically queues the DMA transfer. Path 2a provides better total memory bandwidth as the GPU memory bandwidth and the PCI Express bandwidth accumulate. In addition, it requires less GPU memory bandwidth. In contrast, path 2b offers better latency for stores by the kernel. The faster path varies with the employed GPU, and especially the performance of path 2a depends on the chipset as well. On AMD, Cypress kernels favor path 2a (0.6 % faster), Cayman and Tahiti kernels are faster with path 2b (2.3 % / 1.2 % faster). On Intel systems only path 2b works properly. In the end, the differences are measurable but not so significant.

When one path is not well supported by the hardware, choosing the correct path becomes much more important. The Intel chipsets employed on Nehalem motherboards show a flaw with path 2a whereas path 2b should be avoided if the DMA engine is restricted to 64-bit DMA transfers (like for Cayman). Hence, at the moment Cayman cannot be used in combination with Intel CPUs. As

[24] In addition to the conversion overhead, depending on the hardware, the 64-bit transfer speed in path 1b can be lower than in path 1a.

a remark: a third output path, with 64-bit transfers as for input path 1b, is no feasible workaround for the Cayman DMA problem since the output rates are much higher than the input rates.

12.13 Single Precision General Matrix Multiplication

Besides DGEMM, also **SGEMM** (Single Precision General Matrix Multiplication) is a relevant BLAS function for scientific applications. It is thus analyzed how the work that has been put into DGEMM can be used for obtaining an SGEMM implementation. The Cayman architecture offers 4D VLIW shaders, which offers an easy way to move from DGEMM to SGEMM:

- The Cayman GPU can execute four single precision instructions instead of one double precision instruction.
  - This does not depend on whether the instruction is addition, multiplication, or fused-multiply-add.
- For input matrices of equal size (in bytes) and identical k parameter, the SGEMM requires almost four times the instructions DGEMM does.
  - This is due to the C-matrix having exactly four times the number of entries while the loop has equally many iterations.
  - Even more, a *double* value used in the DGEMM can be interpreted as a *float2* vector in the SGEMM. Instead of multiplying two double values in the DGEMM, the components of the *float2* vectors must be multiplied with each other. (This requires the data format of the input matrices to be chosen appropriately, i. e. each *float2* fetched from the A-matrix must contain two elements of the same column while each entry loaded from B must hold two elements of the same row.)

According to the above considerations, a DGEMM kernel can be converted to an SGEMM kernel, by just exchanging the double precision fused-multiply-add VLIWs for the VLIWs representing four single precision fused-multiply-add instructions on the same source registers. Such an SGEMM kernel achieves exactly four times the DGEMM performance, at least as long as the GPU employs 4D shaders. Thus, on Cayman it is as efficient as the DGEMM kernel. On the Cypress architecture the situation is different because of the 5D shaders. With a 4×4 tiling, the cache bandwidth is exactly sufficient to achieve peak performance in the DGEMM (see Section 11.2.4.1). As the 5000 series has five times the performance in single precision it has in double precision, the above approach, per definition, limits the achievable Cypress performance to 80 % of the peak, if the cache hit ratio is 100 %. For a better performance, the tiling size must be increased. However, since the Cypress architecture is outdated, this drawback is considered acceptable.

Unfortunately, reality is not exactly as simple as described above. Obviously, the result must be stored, first in a temporary register, and later to memory. As the SGEMM thread processes four times the number of entries (of the C-matrix) in parallel, it requires twice the number of registers to store and accumulate intermediate results. In addition, it must write twice the amount of data to memory. This poses two problems:

- Color Buffers can no longer be used for the output. (DGEMM with 4×4 tiling already generates the maximum output size that can be written to Color Buffers.) Thus, the output must be done via MemExport. Section 11.2.5 explained that when combining asynchronous transfer and MemExport, the binary driver patch is mandatory. This is a major but unavoidable drawback.
- The doubled output data size can pose bandwidth problems. As stated in Section 12.6.2.1, these can be overcome by increasing the k parameter. The following results are achieved

after a parameter range scan, which revealed that for the Cypress in fact an increased k parameter of 2048 is ideal. For the Cayman $k = 1024$ remains optimal.

Having an SGEMM implementation, it is also interesting what the maximum achievable performance for integer matrix multiplication is. Additionally, for reasons given in Section 17.2, also a GEMM implementation with the multiplication replaced by logical *AND* and the addition exchanged with a logical *XOR* operation are analyzed. In the following, these variants are called **IGEMM** (Integer) and **BGEMM** (Binary) respectively. All variants are tested on Cypress and Cayman hardware. Table 12.44 shows the results.

GPU		SGEMM	IGEMM	BGEMM	*DGEMM*
5870	Achieved	1432 $^{\text{GFlop}}/_{\text{s}}$	472 $^{\text{GOp}}/_{\text{s}}$	736 $^{\text{GOp}}/_{\text{s}}$	*494* $^{GFlop}/_{s}$
6970	Peak	2703 $^{\text{GFlop}}/_{\text{s}}$	676 $^{\text{GOp}}/_{\text{s}}$	1352 $^{\text{GOp}}/_{\text{s}}$	*676* $^{GFlop}/_{s}$
6970	Achieved	1844 $^{\text{GFlop}}/_{\text{s}}$	492 $^{\text{GOp}}/_{\text{s}}$	1024 $^{\text{GOp}}/_{\text{s}}$	*624* $^{GFlop}/_{s}$
6970	Efficiency	68.2 %	72.8 %	75.7 %	*92.3* %

Table 12.44: SGEMM (and Variants) Kernel Performance [VI]

For understanding the results, it is essential that an n-D-VLIW shader can process n single precision floating point or logical instructions but only one double precision or integer instruction. This yields different theoretical peak performances for the investigated disciplines even though all of them are based on 32-bit data-types.

SGEMM On the Cayman, the SGEMM achieves close to 70 % of the peak performance, on the Cypress only slightly above 50 %. The difference of 20 % is likely to be related to the proclaimed 20 % loss caused by the 5D shaders of Cypress. In general, the reduced SGEMM performance, compared to the DGEMM, is most probably based on the increased register requirement and less effective MemExport output. Clearly, deducing the SGEMM kernel from the DGEMM kernel is not necessarily optimal in terms of cache access pattern, tiling, register usage, etc. N. Nakasato [Nak 10] achieves a performance of approximately 2 $^{\text{TFlop}}/_{\text{s}}$ on a 5870, which is a bit higher than the result presented here for the 6970.[25]

As a comparison: the LOEWE-CSC nodes achieve 360 $^{\text{GFlop}}/_{\text{s}}$ in CPU-only SGEMM, which is exactly twice their DGEMM performance of 180 $^{\text{GFlop}}/_{\text{s}}$.

IGEMM Naively, one would expect an identical IGEMM performance compared to SGEMM as the GPU supports both floating point fused-multiply-adds and integer fused-multiply-adds. Unfortunately, as for double precision, a VLIW instruction can only perform one 32-bit integer operation per cycle. Thus, the IGEMM performance should be compared to the DGEMM rather than the SGEMM. Obviously, on the Cypress the IGEMM and the DGEMM match up quite good, on the Cayman the IGEMM falls behind. The most probable reason is that the IGEMM uses the SGEMM framework, which relies on MemExport.

BGEMM Naturally, there is no "Fused-*AND*-*XOR*" operation on GPUs. This decreases the achievable performance by a factor of two. Since the GPU with n-D shaders can perform n logical operations per cycle (identical to single precision floating point), BGEMM performance is expected to be half the SGEMM performance. For Cypress this is almost exactly the case. On Cayman the BGEMM is faster than expected. Most probably the performance loss due to the usage of MemExport weights heavier for the SGEMM as its required output bandwidth is higher.

[25] As the 5870 and 6970 have almost identical theoretical single-precision performances (see Table A.2), it is not unreasonable to compare them.

Chapter 13

CALDGEMM Support for Arbitrary GPU Frameworks

13.1 Motivation

The foundation of the initial CALDGEMM implementation on the CAL API makes it little portable. CAL was chosen for squeezing the last bit of performance out of the kernel. For multi-GPU versions, this last bit is not that important since kernel performance is not the bottleneck anymore. The performance is rather limited by the host, often by PCI Express or memory bandwidth. Besides, it is desirable to use CALDGEMM with other processing devices apart from AMD GPUs. For this purpose, the CALDGEMM library has been split in three parts: an abstract interface class that encapsulates all access to the hardware accelerator like the GPU, an implementation of this interface for CAL, and the actual CALDGEMM library class that utilizes the interface. This makes CALDGEMM easily adaptable to different architectures like OpenCL, CUDA on the NVIDIA Fermi, or the new Intel Xeon Phi [Int 10] accelerators. Adoption of new hardware only requires the creation of a proper interface implementation.

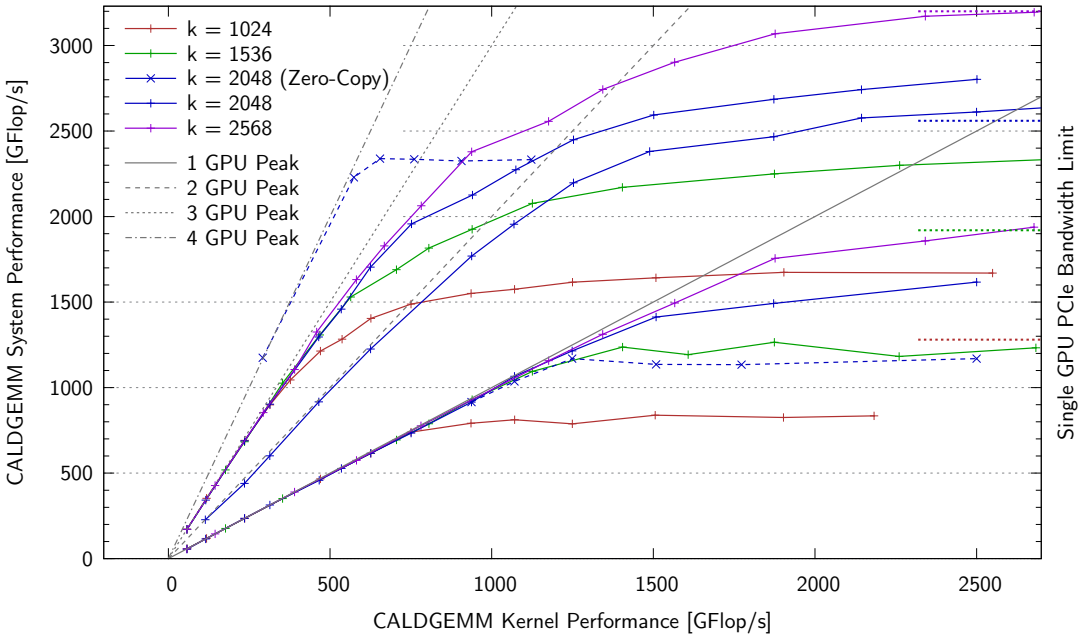


Figure 13.1: Scalability of CALDGEMM System Performance[1,2] [VI]

[1] The four GPU measurement is performed on [XIII] with two dual-GPU boards.

Before new implementations are discussed, the bottlenecks of the CAL implementation for multi-GPU and for upcoming very fast GPUs are analyzed. In order to simulate faster and slower GPUs respectively, the DGEMM kernel is modified to process only a partial or multiple k-loops respectively (whereas the framework uses the correct k). Obviously, running a partial loop speeds up the kernel while running multiple loops slows it down. Naturally, the results are wrong but the kernel can simulate an arbitrary performance. The advantage of using the real DGEMM kernel over an arbitrary kernel is that it uses the real DGEMM memory access patterns. Fig. 13.1 gives an overview of the achieved DGEMM system performance in relation to kernel performance, number of GPUs, and value of k. All remaining options are individually tuned for the best performance at every measuring point. Thus, all curves are themselves the maximum of multiple curves for different parameters, which explains the kinks in the curves where the optimal parameters change. The Zero-Copy performance for one GPU and $k = 2048$ is shown exemplarily for comparison. All other kernels write to GPU memory. In general, the Zero-Copy version is faster than or equally fast as the GPU output variant for slow kernels but the saturation sets in earlier. Since no Cypress based system with four GPUs has been available, quad-GPU benchmarks are performed with Cayman GPUs. Due to the DMA issue, only the Zero-Copy variant can be used in this case.

The figure includes the single-GPU performance limits posed by PCI Express bandwidth at different k.[2] Obviously, this limit is never reached. System performance scales almost linearly with kernel performance until it saturates due to the memory. The saturation point shifts to the right with larger k and to the left with more GPUs. Section 12.8.4 concluded that $k > 2048$ is impracticable for HPL due to the N_b^3 dependency of the factorization time while $k < 2048$ does not perform well with three or more GPUs. Hence, the following discussion focuses on $k = 2048$. The figure shows that the current framework can well utilize single GPUs up to 1500 $^{\mathrm{GFlop}}/_{\mathrm{s}}$ kernel performance, two GPUs up to 1000 $^{\mathrm{GFlop}}/_{\mathrm{s}}$, three GPUs up to about 750 $^{\mathrm{GFlop}}/_{\mathrm{s}}$, and four GPUs up to 600 $^{\mathrm{GFlop}}/_{\mathrm{s}}$.

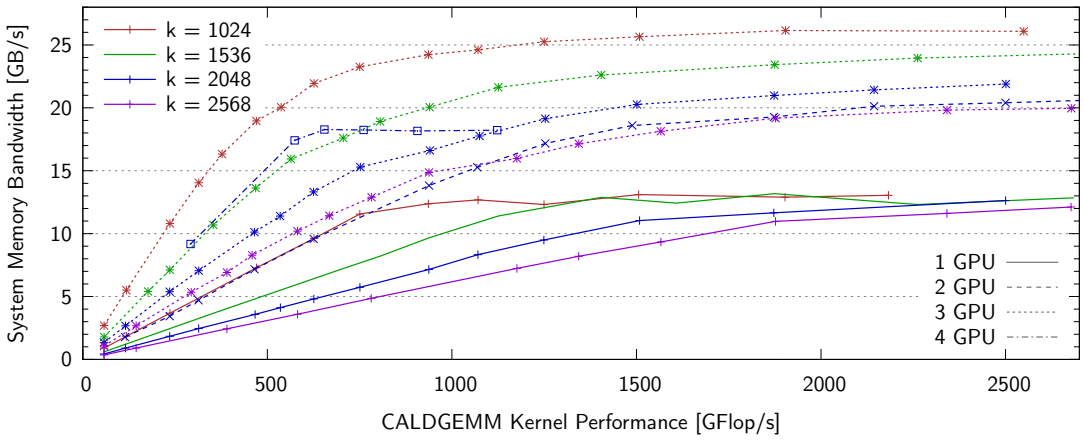


Figure 13.2: Memory Bandwidth required for Multi-GPU CALDGEMM [VI]

Fig. 13.2 shows the host memory bandwidth achieved during the runs: one GPU already saturates at 13 $^{\mathrm{GB}}/_{\mathrm{s}}$ while multiple GPUs reach $20-26$ $^{\mathrm{GB}}/_{\mathrm{s}}$. Two postprocessing cores are used per GPU. In the single-GPU case, this results in 4.875 $^{\mathrm{GB}}/_{\mathrm{s}}$ memory bandwidth per CPU core and 3.25 $^{\mathrm{GB}}/_{\mathrm{s}}$ by the DMA transfer. Employing more cores does not increase the overall bandwidth. Since GPUs are connected to particular CPU dies each, while all dies contribute to pre- and postprocessing, it is improbable that the memory bandwidth can be increased much further. A quantitative analysis is hardly possible. Due to the complicated dependencies between thread pinning and memory location, it is not meaningful to compare the achieved bandwidth to the theoretical peak bandwidth.

[2] The lines in the rightmost part of the diagram indicate the performance limit derived from the PCI Express bandwidth for a single GPU at different k. For n GPUs the limit is n times that high – if all GPUs have a separate connection. The grey line a curve approaches asymptotically on the left, indicates the number of GPUs in the measurement.

13.2 A DMA Framework with better Scalability

It was a design decision of the original CALDGEMM implementation to calculate only $X = \alpha \cdot A \cdot B$ on the GPU and then calculate $C' = X + \beta \cdot C$ on the CPU. On the one hand, this saves oneself the transfer of the original C matrix to the GPU. On the other hand, this results in four memory accesses per matrix element (see Section 12.6.2.1). To cope with multiple fast GPUs in the future, a scalable framework can perform the entire DGEMM calculation on the GPU. For this purpose, the original C matrix must be transferred to the graphics card. Only two host memory transactions per entry are required: one when sending the initial C matrix-entry per DMA and one when receiving the new entry. This reduces the system memory load to one half while it doubles the required PCI Express bandwidth. To be precise, this raises the required host to GPU bandwidth, which has been negligible before, to roughly the same level as the GPU to host bandwidth, which is unchanged. Since PCI Express is specified as full duplex and the maximum unidirectional bandwidth does not change significantly, this should not pose a problem.

With the above approach, the host memory is read and written directly by DMA. Hence, neither the host side DMA buffers nor pre- nor postprocessing threads are required. In addition, it simplifies the host-scheduling significantly. In contrast, it poses high demands on the DMA engine. First, the DMA transfer is not restricted to a small dedicated DMA buffer on the host but the DMA engine must access the entire host memory. Second, instead of a consecutive memory segment, the DMA transfer must work on submatrices of the C matrix, where each line of the submatrix is a separate memory segment. Finally, it requires full duplex operation. Both OpenCL and CUDA offer a DMA API that is capable of autonomically transferring submatrices from host memory to continuous GPU memory and vice versa. Such a transfer is called a **strided** transfer – in contrast to a **linear** transfer of a connected memory segment.

DMA transfers can be performed by the GPU DMA engine or by either a GPU kernel or a CPU thread via Zero-Copy. This offers the following DMA transfer schemes:

I Both, transfer to GPU and back to host are performed by the GPU DMA engines prior to and after kernel execution respectively.

II The GPU kernel directly reads and stores the entries from and to host memory via Zero-Copy. The DMA engines are not used.[3]

III The processor copies the matrix to the GPU via Zero-Copy. The transfer back is done via the DMA engine. This option is particularly useful for GPUs with only a single DMA engine.[4]

In theory, method II is sovereign since it does never store the C matrix in global GPU memory. Both other methods do this and so, as the data are read and written by the kernel as well, cause global GPU memory load equal to twice the PCI Express throughput.[5] The host memory can be allocated either via *malloc* as usual or via special API functions provided by CUDA / OpenCL, which register the memory for fast GPU access. Table 13.3 lists uni- and bidirectional DMA throughput measurements employing all methods and the maximum DGEMM performance possible by reason of the available bandwidth (at $k = 2048$ – Section 12.6.2.1 explains the calculation). Results with linear transfers are shown for comparison although they are useless for the task. The S10000 measurement shows that PCI Express generation three is likely to double the performance in near future. However, the only available PCI Express 3.0 system employs AMD GPUs on an Intel chipset and therefore it does not properly support the required DMA transfer modes (as explained below). For this reason, and because no NVIDIA system with PCI Express 3.0 is available for comparison anyway, the following discussion is limited to generation two.

[3] Naturally, this could be combined with method I, e. g. by sending via I and receiving via II. However, measurements show that either of the two methods is faster for both sending and receiving and thus should be used exclusively.

[4] The processor can write to the GPU quite fast with DMA write combining. Reading from the GPU is significantly slower, thus this method cannot be used the other way around. Naturally, it could be used in combination with method II instead of I, but if the kernel DMA is faster than the DMA engine, it can be used exclusively in any case.

[5] For instance, on the LOEWE-CSC with $k = 1024$, methods I and III require 7.3 GB/s of GPU memory bandwidth.

GPU	Method	Memory	To GPU GB/s	To Host GB/s	Bidirectional GB/s	Max Flops GFlop/s
GTX580	I (linear)	*malloc*	3.725	3.973	3.908	1000.5
GTX580	I (linear)	runtime	5.976	6.653	6.330	1620.5
GTX580	I (strided)	*malloc*	3.764	4.024	3.919	1003.2
GTX580	I (strided)	runtime	6.043	6.624	6.241	1597.7
GTX580	II (strided)	runtime			7.547	1932.0
M2070	I (strided)	runtime	5.959	6.591	8.717	2231.7
6970	I (linear)	*malloc*	4.949	5.214	5.739	1469.1
6970	I (linear)	runtime	6.293	6.679	6.479	1658.7
6970	I (strided)	*malloc*	1.529	0.492	0.859	219.8
6970	I (strided)	runtime	4.964	0.624	1.112	284.8
6970	II (strided)	runtime			8.935	2287.4
6970	III (strided)	runtime	4.500	0.624	1.112	284.8
S10000	I *(linear)*	*runtime*	8.798	12.168	14.684	3759.1

Table 13.3: OpenCL / CUDA DMA Throughput [II,XIV,XVIII]

These measurements lead to the following considerations:

- The DMA engines of the Fermi Tesla (M2070) can do full duplex transfers, the engines of the consumer-grade Fermis and of the AMD GPUs cannot.[6]
- The NVIDIA GPUs show good DMA performance as long as the memory is allocated by the NVIDIA runtime. With more than 6 GB/s, the GTX580 can transfer data corresponding to about 1.6 TFlop/s via method I, which is absolutely sufficient for the next GPU generation. The Fermi excels with 2.2 TFlop/s.
- The GTX580 is faster with method II, which unfortunately turns out to be incompatible with certain tested AMD chipsets and is thus no panacea.
- On runtime allocated memory, AMD's DMA engine shows good host-to-GPU but poor GPU-to-host strided transfer speed rendering method I unfeasible.
- Method III has acceptable host to GPU performance on AMD platforms but suffers from the poor DMA transfer in the opposite direction, too.
- On AMD hardware both method I and III require the strided DMA transfer to the host to be sped up in order to be practicable. If a future driver provided these necessary improvements, and at the same time was still unable to access both DMA engines simultaneously[6], method III would probably be the faster one.
- Still, the above scenario depends on too many uncertain factors. On top of that, method III requires three additional CPU cores creating extra scheduling overhead. For these reasons, the idea was dropped.
- Until AMD improves the strided GPU to host bandwidth, the only feasible option for AMD GPUs is method II. However, this works solely when Zero-Copy is well supported by the chipset[7] currently disqualifying Intel platforms.

In summary, method II is currently the only viable option for AMD and at the same time the faster version for all non-Tesla NVIDIA GPUs, but it is not compatible with every chipset. In contrast, method I is faster on the Tesla (although the competition is on a very high level), ensures good performance on NVIDIA in every case, and shows no incompatibilities; but it is slow on AMD. In order to ensure the greatest flexibility for upcoming GPUs and driver improvements, both method I and II have been realized for two APIs each: CUDA and OpenCL. For this purpose, two implementations of the abstract CALDGEMM interface have been created, one for each API.

[6] In fact, all DMA engines themselves are half-duplex but every listed GPU possesses two engines for full duplex transfers. NVIDIA deactivates one engine on the GTX580 to distinguish it from the professional market segment whereas the current AMD driver seems to be incapable of driving both engines simultaneously.

[7] Section 12.5.1 provides information on Zero-Copy support and discusses problems with Intel chipsets.

The original CAL implementation uses preprocessing for transposing input matrices and providing the input data in an optimal format for the kernel. The new implementations relocate these preprocessing tasks to conversion kernels on the GPU, which were introduced in Section 12.8.3.2 to cope with the Cayman DMA issues. In the same way as for the DMA issue, since usually the input matrices A and B are relatively small compared to C, the performance impact is negligible. Since no pre- or postprocessing is needed, the new framework is of incredible simplicity. All DMA transfers and DGEMM kernels are queued into the CUDA or OpenCL command queues automatizing the scheduling. Only the check for the availability of matrices in the BBuffers remains.

To analyze the scalability of the framework, the same technique as for the initial CAL implementation is employed: a fake kernel simulates a particular kernel performance. Currently, there persists one problem with OpenCL drivers. For good performance method I requires the memory to be allocated by the runtime, method II enforces this restriction at all times. Both the AMD and the NVIDIA driver only support a very limited amount of runtime allocated OpenCL memory insufficient for large matrices. In contrast, the CUDA runtime has no such restriction.

Fig. 13.4 compares the scalability of the CUDA and the CAL framework. For the above reasons, OpenCL is not included. Measurements with Zero-Copy and with three or more NVIDIA GPUs are taken on a Sandy-Bridge platform since the reference AMD system does not support Zero-Copy properly and since no AMD system with more than two NVIDIA GPUs was available. A comparison with Table 13.3 reveals that the single-GPU CUDA performance with method I saturates at about 90 % of the limit posed by PCI Express bandwidth, method II reaches 95 % even with four GPUs. With up to two GPUs, the CAL framework can compete with the half-duplex DMA CUDA-version on the GTX580. As soon as CUDA with full-duplex DMA, the Zero-Copy method, or more than two GPUs are employed, the new framework easily outperforms the original implementation. The figure also shows clearly, that the old framework is not capable of running more than two high performance GPUs of the next generation. It must be noted that even in situations where CAL and CUDA performance is even, the new framework causes only half the memory load and utilizes less CPU cores leaving significantly more resources for concurrent tasks on the CPU. Even at low kernel performance, the Fermi Tesla outperforms the GTX580, which encounters some delays due to half-duplex transfers. The new framework scales to a kernel performance of nearly 2 TFlop/s for up to four GPUs, reaching 7.27 GB/s PCI Express bandwidth per GPU, 29.1 GB/s memory bandwidth on the host, and about the threefold total performance of the CAL implementation.

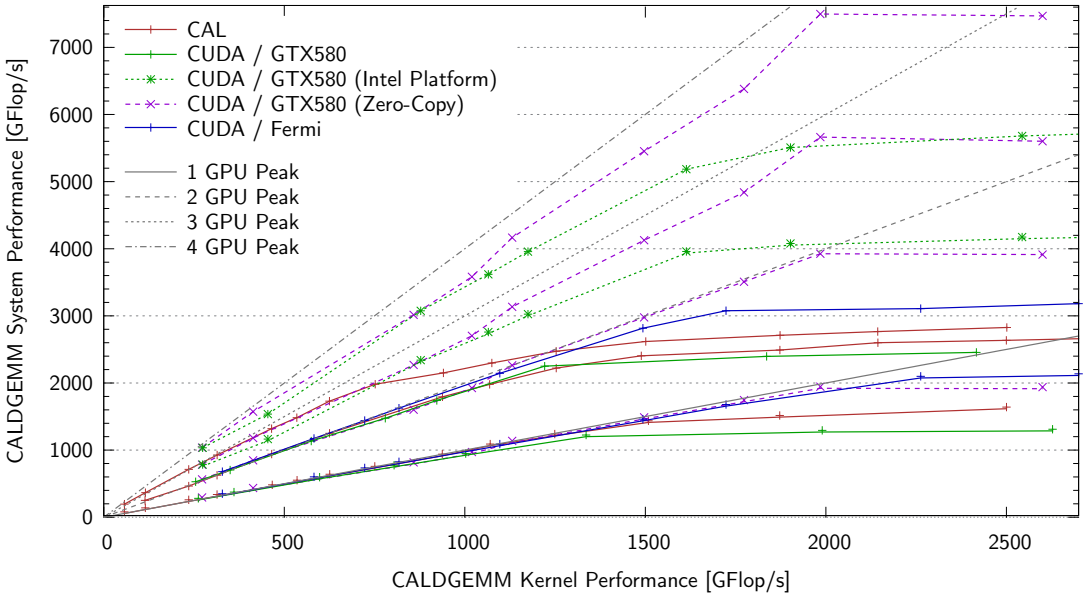


Figure 13.4: Scalability of CALDGEMM CUDA Backend [VI,VIII,XVIII]

Chapter 14

The Sanam Cluster & the Lattice-QCD Cluster at GSI

This final chapter on Linpack optimizations introduces the current AMD GPU generation called Tahiti and a compute cluster architecture basing on it. The architecture boosts power efficiency by using multiple GPUs as primary compute devices while the CPU is merely used for data movement and management instead of computation. The Sanam cluster [Adv 12] (Fig. 14.1) of the King Abdulaziz City for Science and Technology (KACST [KAC]) employs this design and the scheduled Lattice-QCD cluster, which will be built at GSI in the course of the year, will use it as well. Evaluation of performance, power efficiency, and interaction of available hardware components, which finally led to the decision on the node configuration, was completely performed in the course of this thesis. Amongst others, the Linpack benchmark has been used to examine hardware constellations. Due to missing driver support for certain fast OpenCL DMA operations, the improved DMA framework introduced in the previous chapter could not be employed yet.

To provide an OpenCL version despite the current driver problems, a second OpenCL implementation with the old DMA scheme of the CAL version has been implemented. It shows similar performance as with CAL ($\approx 3\,\%$ slower). Therefore, and since Tahiti still supports CAL, and since the CAL framework has been tested more extensively, the CAL version is still used in this chapter.

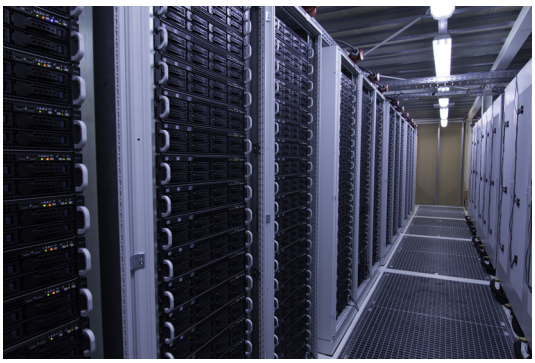


Figure 14.1: The Sanam Cluster at GSI

The proceeding for hardware selection is the following: The setup for cooling and racks is inherited from the LOEWE-CSC (see Section 10.2). Having optimized the GPU kernels, the next step is tuning the available system platforms to the very end with respect to single-node performance with the given GPU. Multi-node tests and acquisition of additional prototypes are postponed until after the decision on the system platform because this is a considerable investment. Potential platforms are Intel Sandy Bridge and AMD Interlagos. With the platform fixed, a cost-benefit analysis with respect to the utilizable multi-node performance (including GPUs) points out the exact CPU model to be used. Then, the software is optimized for multi-node tests and fine-tuned for the particular node design.

14.1 AMD 7000 Series (Tahiti)

Since the CALDGEMM framework design cleanly separates the GPU kernel from the host code, adoption of a new GPU generation requires only a customized kernel. Analogously to the param-

eter range scan in Section 11.2.4.1, the Tahiti kernel was created. In contrast to its predecessors, Tahiti favors a transposed A-matrix. Other kernel characteristics remain unchanged. Since Tahiti does not show the DMA issue that Cayman has, all DMA schemes are available with kernel output to a temporary GPU buffer being the fastest one. With one additional Tahiti specific optimization, which shifts even numbered rows of the input matrices by one column in order to reduce memory bank conflicts, the 7970 GPU achieves up to 805 GFlop/s in DGEMM at $k = 1024$ (85 % of the peak performance). Unfortunately, performance is not flat with respect to matrix sizes (see Fig. 14.2), and due to memory bandwidth limitations for multi-GPU systems (see Section 12.6.2.1), only k-settings of 1920 or 2048 are feasible still yielding about 790 GFlop/s.

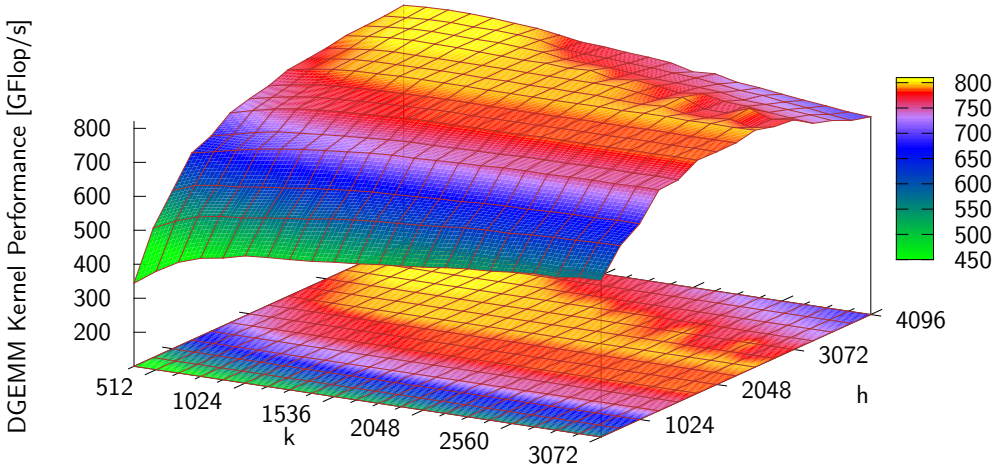


Figure 14.2: DGEMM Kernel Performance for different Matrix Sizes on 7970 [VII]

14.2 Putting the Pieces together

This section combines many – not to say most – of the advancements in the previous chapters, among them: the CALDGEMM and HPL-GPU implementation with lookahead together with the multi-node improvements for the LOEWE-CSC from Chapter 11, the DMA workaround for Intel platforms in Section 12.5.2, the multi-GPU implementation from Section 12.6 together with the memory bandwidth considerations in Section 12.6.2.1, improvements for the factorization in Section 12.6.4.2, modifications to boost power efficiency in Section 12.7, quad-GPU DGEMM and HPL optimizations from Section 12.8.4, changes in order to support other BLAS libraries as MKL or ACML in Section 12.9, and finally, the optimized kernel for the Tahiti GPU.

The compute node design to be investigated is subject to the following boundary conditions: two socket servers are usually the best compromise of price, performance, power efficiency, and hardware support, i. e. number of available PCI Express lanes, memory channels and capacity, etc. Feasible processors are six- and eight-core Intel Sandy Bridge or 16-core AMD Interlagos. Graphics cards by AMD are more than competitive and still reasonably priced. Cards with two GPU chips on one board are an effective option to assemble compact nodes with several GPUs. The AMD S10000 is such a dual-GPU of the recent Tahiti family, which offers improved reliability being a server grade card. Assembling two such cards in one chassis provides four GPUs, which is a good tradeoff of performance and usability and a balanced approach with one CPU socket handling two GPUs. Since 64 GB is insufficient to achieve high load on four GPUs (compare Fig. 12.28) whereas 256 GB is completely out of scale for almost all applications, the nodes have 128 GB of memory. Usage of eight modules of 16 GB each yields the best power efficiency while still populating all available channels. InfiniBand is used as high performance interconnect. Unused hardware such as VGA port, additional LAN ports, and especially USB is disabled or set to auto-suspend, the latter saving about 15 W. The main remaining question is which processor to choose.

14.2.1 Preliminary Improvements

This section lists necessary adaptations and improvements for the Intel and AMD platforms as well as generic optimizations.

AMD Platform The AMD platform is subject to the same restrictions as the Cayman system in Section 12.8.4. Hence, lookahead 1 and 2 are turned off one after another, the optimal times being determined experimentally. The processor only processes the absolutely necessary part of the DGEMM since the negative effect on the GPU DGEMM overcompensates the gain of the additional compute capacity. Since AMD CPUs react very sensitively to memory congestion while the ACML causes even heavier load than GotoBLAS (see Section 12.9), a reduction of the number of concurrent OpenMP threads similar to the GotoBLAS optimization in Section 12.6.4.2 is a logical step. This is realized in two ways: the most recent ACML has an experimental switch to automatically determine the number of concurrent threads based on the problem size; alternatively, a wrapper library that encapsulates all relevant BLAS calls manually applies the heuristics that were developed for GotoBLAS by running *omp_set_num_threads*. Unfortunately, both attempts result in the same problem. The OpenMP specification does not specify when threads are created and terminated or how many threads to keep ready for subsequent parallel sections. The GCC compiler implements a heuristics that terminates certain threads if they have been unused during multiple parallel sections. When it respawns such a thread for a larger parallel section afterward, the new thread inherits the CPU affinity from the spawning thread, which does in general not coincide with the affinity of the terminated thread, rendering the core reservation technique developed in Section 12.9 for ACML ineffective. There are workarounds for this, which can reset the thread affinity but still, since the factorization issues countless BLAS calls with arbitrary problem sizes, this leads to the creation and termination of millions of threads during an HPL run deteriorating the performance. Since this is triggered by GCC internals which comply with the OpenMP specification and are not configurable, the only way to tackle this is modifying the OpenMP sources of GCC. An according patch has been developed and is used in the following. Both methods yield similar results and the optimal N_b for AMD CPUs is 2048.

Intel Platform The HPL-GPU benchmark shows one big flaw on Intel platforms: Querying GPU events from the CPU socket that connects a GPU is very fast, but querying from the other socket introduces a tremendous latency, which costs about 30 % of HPL performance. It is not known yet whether this is caused by the hardware, by the AMD driver, or by something else. Since CALDGEMM uses a single thread for DMA management, the only feasible workaround is repinning this thread to another core on the appropriate socket if necessary. This procedure must be repeated prior to each GPU event query. The performance impact of this approach can be checked on an AMD system, where the repinning is not needed. It is below 1 %. This approach permits another strategy in lieu of the multiple *DivideBuffer* threads suggested in Section 12.6.2.2. If the main thread that supervises DMA transfers is repinned to the correct cores anyway, it can preprocess all tiles itself for all GPUs, always writing to locally connected memory. This frees up certain CPU resources. However, it turns out that during preparation of the BBuffers at the beginning of the run, four Tahiti GPUs temporarily require more CPU power than a single thread can deliver since at this point in time all GPUs wait for new B-tiles simultaneously. The overall performance penalty is in the range of 2 % so it is not used.

Apart from that, HPL-GPU runs literally out of the box on Sandy Bridge without necessity for customization or tuning. In contrast to the AMD processors, factorization and pivotization with the MKL library are much faster and, even further, the interdependency with the GPU DGEMM is so weak that there is no need to turn off lookahead. On top of that, after the factorization the CPU can contribute to the DGEMM. Fig. 14.3 shows HPL runs with combined GPU / CPU DGEMM using no, two-phase, and three-phase dynamic scheduling. (A GPU-only run achieves 2282 GFlop/s, early lookahead is a related improvement introduced below.) Even

though the three-phase run can reduce the CPU idle duration slightly further, it is still slower than the two-phase run due to scheduling overhead. The core reservation technique introduced in Section 12.9 works fine without further modifications. More changes to the thread count such as in Section 12.6.4.2 are not necessary. Intel achieves its maximum DGEMM performance already with $N_b = 1920$ further speeding up the factorization compared to AMD with $N_b = 2048$.

Generic Improvements All benchmarks are performed with transparent huge pages (see Appendix C.3). In many cases, it turns out that the optimum implementation from a performance perspective differs from the most power efficient code. In order to position the system as good as possible in both the Top500 and the Green500 list, two code versions are maintained for both purposes. (For instance, the power efficient version reduces CPU load as far as possible whereas the fast version utilizes the CPU to the full extent if that speeds up processing (see Section 12.7.3).) With the support for non-square tiles (see Section 12.8.4) the tile size of the outermost GPU tiles can be adjusted such that GPU tile size restrictions for the first phase CPU part become negligible. This **adaptive tile size** renders the dynamic scheduler obsolete for very fast multi-GPU configurations. Measurements reveal that the row shifts in the Tahiti kernel optimized for highest performance speed up the processing but cause much higher power consumption thus reducing the power efficiency. Hence, an alternate kernel is optimized for power efficiency.

14.2.2 Early Lookahead

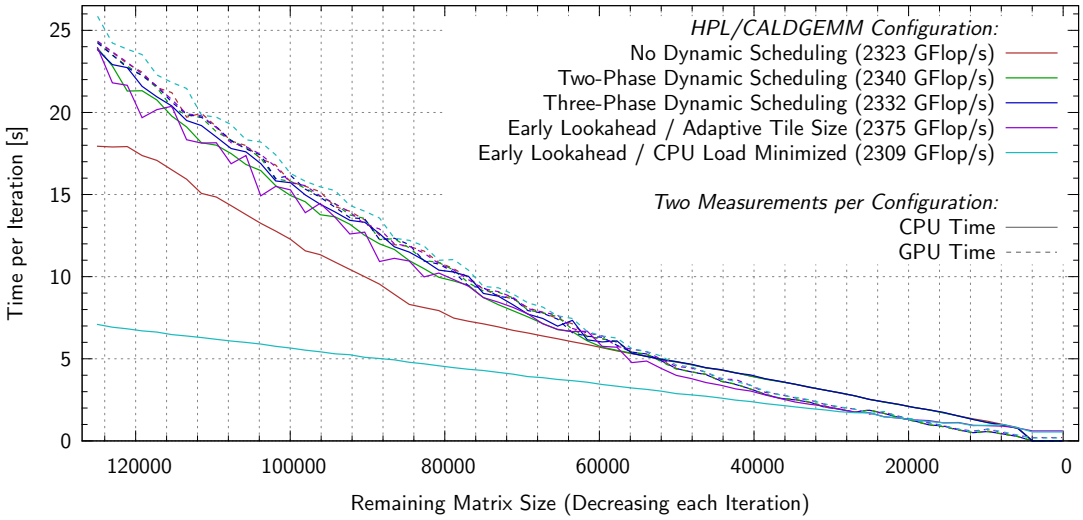


Figure 14.3: Dynamic Scheduling and Early Lookahead Performance[1] [VIII]

Processing is slowed down especially toward the end of the run when CPU time exceeds GPU time and lookahead can no longer hide the CPU tasks. Lookahead is implemented such, that it takes the first N_b columns of the matrix and processes them on the CPU to avoid unnecessary synchronization. This is not optimal at the end of the run, where the GPU should take care of this part. Therefore, lookahead is complemented by an **early lookahead** mode: the CPU processes only the matrix remainders from the GPU matrix size restrictions, and then waits until the GPU has finished all tiles containing entries of the leftmost N_b columns. The GPU tile scheduler ensures that these tiles are processed first. As a side effect, this boosts the power efficiency for the setup with minimized CPU load (since it reduces the minimum CPU problem size). So, the Top500 run switches to early lookahead as soon as it is faster whereas the Green500

[1] In Figures 14.3 to 14.6 dashed lines represent GPU time and DGEMM performance respectively for every configuration while continuous lines represent CPU time and HPL performance respectively.

run uses early lookahead right from the beginning. Fig. 14.3 shows that both CPU load for the power efficient run as well as GPU idle period in general are reduced significantly. Energy efficiency improves from 1935 $^{\mathrm{MFlop}}/_{\mathrm{J}}$ with the old lookahead to 2028 $^{\mathrm{MFlop}}/_{\mathrm{J}}$ with early lookahead and reaches 2066 $^{\mathrm{MFlop}}/_{\mathrm{J}}$ in the minimal CPU load configuration.

14.2.3 Choosing a Platform

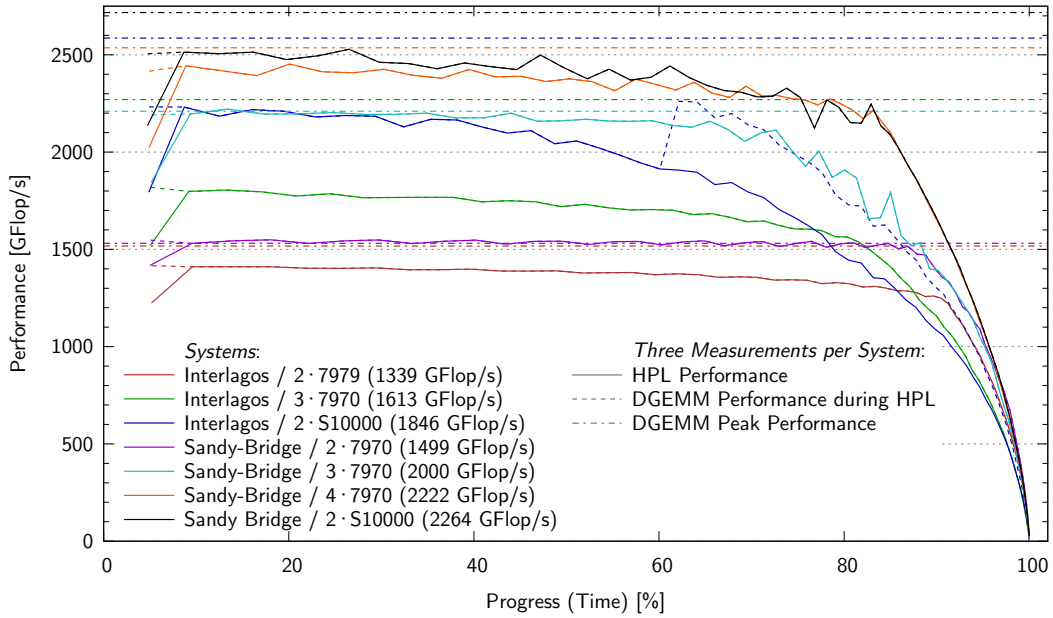


Figure 14.4: Multi-GPU HPL Performance on Intel and AMD Systems[1] [VII,VIII]

The primary objective of the processor is to support and to load the GPU at its best but not necessarily to work as a number cruncher. Hence, even though the Interlagos has a superior theoretical peak performance at an even lower price, the AMD platform does not automatically yield better overall performance. In order to measure the influence of the CPU on HPL performance, plots like Figures 14.4 and 14.5 are used. Fig. 14.4 presents the peak GPU-only DGEMM performance and sets this in relation to the DGEMM performance during an HPL run and to the HPL overall performance. It shows preliminary measurements with 7970 GPUs and measurements with the final S10000. The preliminary measurements with the 7970 measure the pure GPU scaling, hence CPU processing is restricted to the necessary matrix borders. The final S10000 configuration employs a full combined GPU / CPU DGEMM where applicable. With two GPUs, peak performance of AMD and Intel platforms are equal, but only the Intel system can sustain this level during HPL. Sometimes it even exceeds the GPU DGEMM performance since the processor contributes some $^{\mathrm{GFlop}}/_{\mathrm{s}}$ for the matrix borders. With three GPUs, the Intel DGEMM peak performance is lower than AMD's due to the loss caused by the above-mentioned repinning workaround. In HPL, the Intel system maintains the full performance for quite a while whereas the loss on AMD is tremendous. With a fourth 7970 GPU, the Intel system reaches its limits and can no longer achieve full DGEMM performance during HPL. The AMD system cannot house four single-GPUs. In the four-GPU measurement (with two dual-GPUs), the AMD system has to turn off lookahead after 60 % of the time, and the Intel system can play the strength of its processor, which can well contribute to the GPU DGEMM. Besides the achieved performance during the fast part of the run at the beginning, also the point where the CPU starts to limit the performance is an important indication. Fig. 14.4 demonstrates that the Intel platform can maintain good performance for a longer time in either configuration. In summary, the Intel platform is superior in multi-GPU configurations justifying the higher processor price.

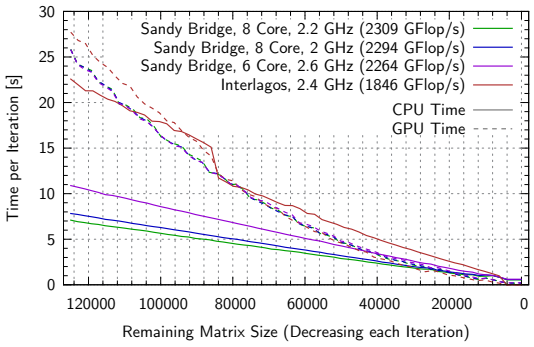


Figure 14.5: HPL Iteration Times on Intel and AMD Systems [VII,XIV]

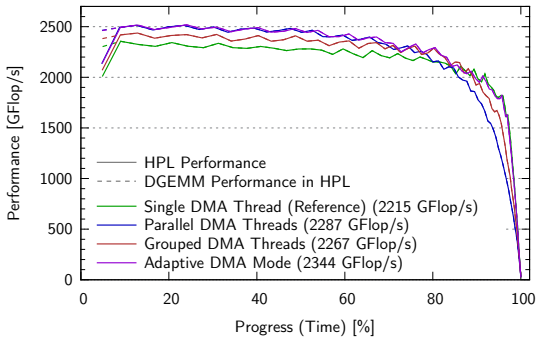


Figure 14.6: Grouped DMA Mode Performance [XIV]

Fig. 14.5 analyzes the above results in more detail. It relates the GPU DGEMM time to the duration of the tasks necessarily performed by the processor. It becomes clear that the AMD processors suffer from the poor performance during factorization and pivotization. This is not necessarily a hardware flaw but the Intel MKL library is probably optimized better. Looking ahead to multi-node tests where the broadcast increases the necessary CPU time since it cannot run in parallel to the other CPU tasks, it is clear that the Intel platform provides by far more margin while the AMD CPU is already at its very limit. All these reasons, together with the fact that adaptation and optimization for the Intel platform are much easier because the processor is less sensitive to memory load, lead to the selection of the Sandy Bridge Processor. A cross-check with other benchmarks and PCIe 3.0 support confirm this choice. The question remains, which Sandy-Bridge model to choose. The six-core version is significantly cheaper delivering almost competitive performance through higher clock rates. However, CALDGEMM requires six CPU cores for pre- and postprocessing of four GPUs, leaving six cores available in the six-core configuration and ten cores with the eight-core CPUs. For this reason, the eight-core variants in Fig. 14.5 show significantly shorter CPU times, the frequency having only a minor influence. Finally, the 2.0 GHz Sandy Bridge E5-2650 is chosen being the best compromise as cheapest eight-core CPU.

In order to predict the estimated multi-node performance, a network mockup was created, emulating the influence of the network (panel broadcast, U-broadcast, etc.) by introducing latencies and producing memory load. The reference values for this simulation are estimated by extrapolating the observations from the LOEWE-CSC to faster nodes and a faster network. This led to an expected network loss of less than 10 %, which is in accordance with the measurements below. The estimated performance penalty on the AMD system is – as expected – significantly larger.

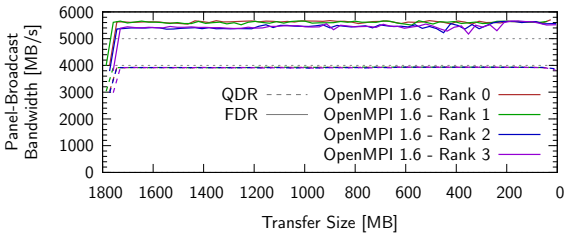


Figure 14.7: InfiniBand Throughput in HPL (Transfer Size decreases each Iteration) [XIV]

With the platform decision made, the ASUS ESC4000/FDR G2 server, which provides onboard FDR InfiniBand, turned out to be well capable of housing two dual-GPUs and Adtech was selected as reseller. The nodes are comprised of: the ASUS ESC4000/FDR G2 chassis with 2U, an ASUS Z9PH-D16/FDR mainboard with onboard 56 $^{\mathrm{GBit}}/_{\mathrm{s}}$ FDR InfiniBand, 128 GB of DDR3-1600 memory at 1.35 V in form of eight 16 GB DIMMs, two Intel Xeon E5-2650 processors, and two AMD S10000 dual-server-GPUs. SUSE Linux Enterprise 11 SP2 is used as operating system since Mellanox supports it for their ConnectX-3 HCA and since it comes with a recent 3.0 Linux kernel with transparent huge pages support (see Appendix C.3). Four prototype nodes were installed and tested with a QDR and an FDR switch. Fig. 14.7 demonstrates that HPL-GPU achieves peak InfiniBand bandwidth except for the very first and very last iterations.

14.2.4 Multi-Node, Fine-Tuning, and Results

With the exact hardware configuration known, final optimizations can be applied.

14.2.4.1 Grouped DMA Thread Mode

One bottleneck, which has persisted for a long time and which was already addressed in Section 12.6.2.2, is the reliance on a single thread for scheduling, management, and supervision of DMA transfers. With four Tahiti GPUs this gains priority. Since thanks to adaptive tile sizes the dynamic scheduler is not used anyway, a simpler and faster management scheme is feasible: the matrix is split in fixed parts for the CPU and each GPU at the very beginning. **Parallel DMA threads** drive the GPUs independently, perform the preprocessing for their GPU, handle the DMA transfer, and synchronize only with the *MergeBuffer* thread for this GPU. This also makes the additional *DivideBuffer* threads obsolete. **Grouped DMA threads** are an advanced version, where GPUs are divided in groups each group managed by one DMA thread. The latter frees some CPU resources and is especially useful when multiple GPUs are connected to the same CPU socket. In the given configuration, one S10000 is connected to one socket and the other S10000 to the other socket requiring two grouped DMA threads instead of four parallel DMA threads. Fig. 14.6 shows that each mode is fastest at a certain time, hence the **adaptive DMA mode** switches from parallel to grouped to a single DMA thread at the optimal points in time. The power efficient code uses only the grouped DMA threads because the two additional loaded CPU cores during the parallel DMA phase lead to a worse net power efficiency.

14.2.4.2 Lookahead 2b

With all of the above optimizations, CALDGEMM accomplishes to achieve a peak performance of **2923 GFlop/s** in combined GPU / CPU DGEMM. Fig. 14.8 relates GPU and CPU time of the best performing single-node run, pointing out the time contribution of all CPU tasks in different colors. The GPU idle time corresponds exactly to the integral over the colored area above the green line for the GPU time. It consists of the lookahead initialization time in the first iteration, of the negligible pipeline initialization time every iteration, and of the CPU dominated part at the end. The figure shows that lookahead hides almost all CPU tasks and how well the scheduling distributes the DGEMM workload. The GPU is kept under full load and CPU idle time is minimum. The final single-node HPL result is **2679 GFlop/s**.

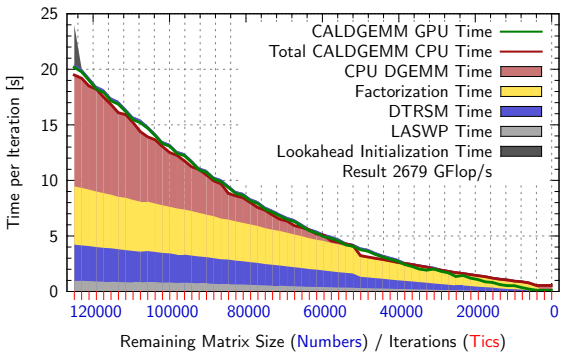


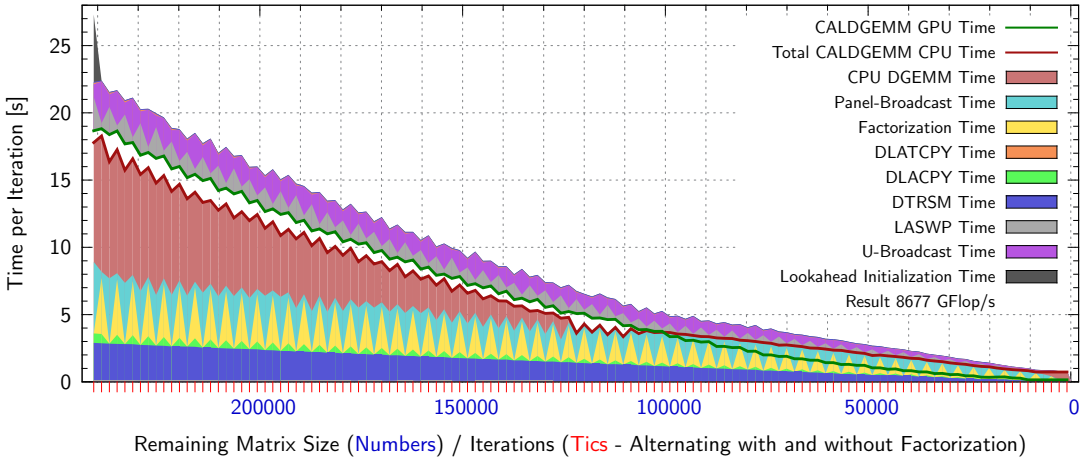
Figure 14.8: Single-Node Lookahead Efficiency and Duration of HPL Steps [XIV]

The first lookahead 2 implementation (**lookahead 2a**) pipelined DTRSM and a part of LASWP[2] but not the U-matrix broadcast achieving good single-node efficiency but leaving some room for multi-node improvements. On the LOEWE-CSC, the U-broadcast has not been that relevant for its minor time consumption but on much faster multi-GPU systems, it becomes significant. The DLACPY[3] deliberately remained single-threaded on the LOEWE since it can run in parallel to the DGEMM, but with CPU execution time becoming critical with faster and faster GPUs, this is no longer ideal. Fig. 14.9a visualizes the time which is lost (again as the integral over the

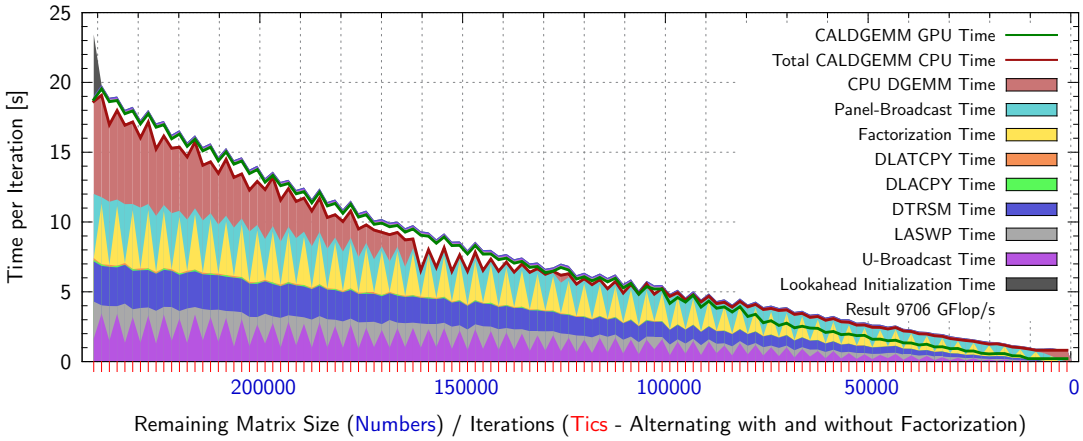
[2] The LASWP consists of two parts, one prior to and one after the U-broadcast. Without the U-broadcast pipelined, lookahead 2a could only pipeline the latter one by construction.

[3] DLACPY and DLATCPY are BLAS [Don+ 90] routines for submatrix copies with and without transposition. There is one large call to both of them in each HPL iteration.

colored area above the GPU time) in a four-node setup achieving 8677 GFlop/s or 2169 GFlop/s per node. Consider that panel broadcast and CPU DGEMM overlap.



(a) Mode 2a



(b) Mode 2b

Figure 14.9: Quad-Node Lookahead Efficiency and Duration of HPL Steps [XIV]

An advanced new **lookahead 2b** mode pipelines the full LASWP and the U-broadcast. It also includes the DLATCPY into the pipeline and employs multiple threads for DLACPY. In order to pipeline the U-broadcast, submatrices of the U-matrix must be transferred which are not stored in continuous memory. This is realized with custom MPI data-types.[4] Fig. 14.9 demonstrates the improvement. GPU idle time is reduced to a minimum. At the beginning, the pipeline even works equally well as in the single-node case but due to the additional CPU workload, the processor dominated period is larger. The final four-node performance is **9706 GFlop/s** or **2427 GFlop/s** per node. To give some more insight, Fig. 14.10 presents an MPI trace of the run. All hidden, non-GPU-related threads of each node contribute to factorization, GPU DGEMM, LASWP, etc. For a similar plot visualizing the GPU related threads (which are omitted here), refer to Fig. 12.15.

14.2.4.3 Power Efficiency

Having demonstrated the potential of the high performance code, this section works out the characteristics of the energy efficient version. The GPU voltage has a significant influence on the system power consumption since power drain goes with the square of the voltage. Also the GPU

[4] At the moment, there is a bottleneck with non-continuous MPI data-types, FDR, and interleaved memory, which is currently investigated by Mellanox. Thus, the following measurements with lookahead 2b use QDR only.

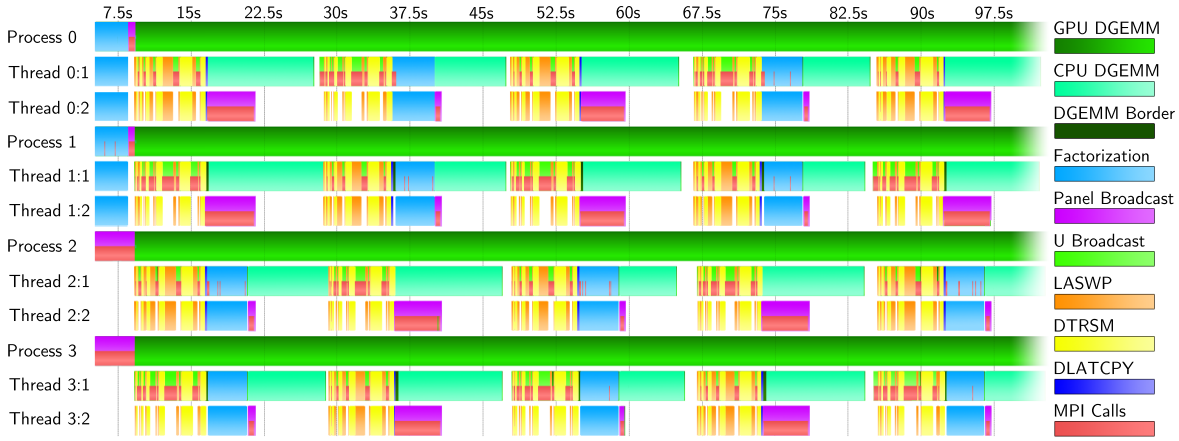
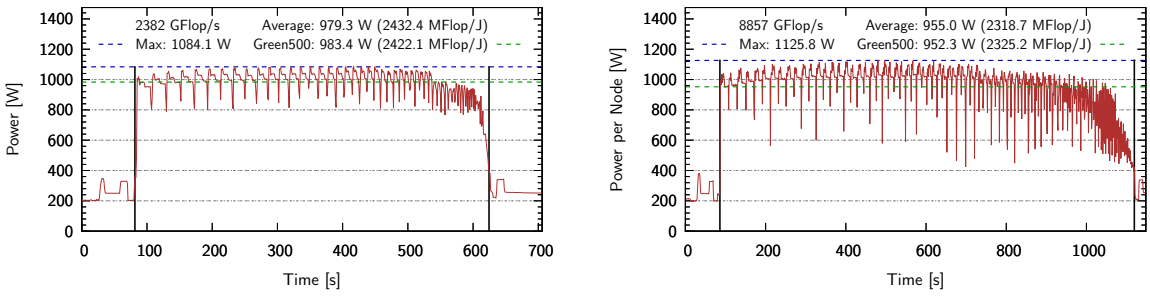


Figure 14.10: Timeline Trace of Four-Node HPL [XIV]

frequency has an impact. However, even though a lower frequency reduces the power consumption, it does not automatically improve the efficiency for reasons explained in Section 12.7.3. The minimum voltage required for stable operation also depends on the frequency and because of fluctuations in the manufacturing process, it also differs from GPU to GPU. Hence, the optimal voltage / frequency combination with respect to power efficiency was determined experimentally using four prototype nodes. The optimum in this case is 1.005 V at 900 MHz compared to stock values of 1.03 V at 950 MHz. To give an impression on the voltage dependency: average system power consumption with 1.005 V, 1.03 V, and 1.08 V is 967.5 W, 1002.7 W, and 1084.7 W respectively. Besides, using fewer BLAS threads in the GPU dominated period improves the efficiency.

Using an LMG95 power meter, the average power consumption during the HPL run is measured. Fig. 14.11 shows the power measurements for a single- and a four-node run. The factorization increases the power consumption measurably while it does not contribute significantly to the compute performance. The fact that a computer participating in a multi-node run does not have to factorize during each HPL iteration compensates the network losses to some extent. The decrease in power consumption toward the end of the run when the GPUs idle a lot makes the average power consumption much smaller than the maximum power consumption. The Green500 run rules demand a measurement over at least 20 % of the runtime which excludes the first 10 %, in order to ensure the hardware components have reached operating temperature. Appropriate Green500 measurements are listed in all power efficiency plots. It is questionable whether this procedure makes sense or not because it overrates the low-power period at the end. On the other hand, the figure shows only a tiny difference between the average and the Green500 measurement. There are two reasons: first, power consumption at the very beginning is slightly lower since the hardware is cooler; second, the distribution of the matrix entries gradually changes from uniform to Gaussian and Section 11.5 argues that a Gaussian distribution results in higher power consumption.



(a) Single-Node (b) Four-Node

Figure 14.11: HPL Power Efficiency with Intel E2650 and AMD S10000 [XIV]

The run rules allow for the unfair trick to measure exactly the last 20 % which leads to ridiculous results of about 3000 MFlop/J for Sanam. The Green500 authors are aware of this and have announced to modify the rules for the next list. All Green500 measurements here measure the average consumption between 10 % and 100 % of the runtime. Compared to the high performance version of HPL-GPU with about 2100 MFlop/J, the power efficient code loses several hundreds of GFlop/s of performance, but it reaches a power efficiency of **2422 MFlop/J** for one node and more than **2300 MFlop/J** for all tested multi-node configurations. To be precise, the energy-optimized HPL-GPU run (single-node) yields a 23.2 % decreased power consumption at a 11.1 % reduced performance resulting in **15.8 %** improved power efficiency.

14.2.5 The November 2012 Top500 & Green500 Lists

The Sanam supercomputer consists of 314 compute nodes employing the architecture described above. Since the last shipment of nodes arrived only less than a week before the submission deadline, only 210 nodes took part in the subsequent runs. In order to reach a stable system in a very short time frame, the GPU frequency was reduced further to 825 MHz. Of course, these two points have a large impact on the Top500 result while the Green500 result is almost unchanged. The large scale test on Sanam unveiled a statistical GPU instability, which is so rare that it had not been observed on the prototypes before but which prohibits any large run over the whole cluster. To investigate this, a full AMD CAL API emulation was written, which does a complete internal event and resource accounting. It can pass all calls through to the real library checking for proper API utilization by performing a bunch of sanity checks such as ensuring that a kernel-finish event is queried before output memory is read. In the end, the cause was tracked down to be inside the GPU driver and AMD could provide a new version within few working days facilitating the measurement presented in Fig. 14.13, which was finally submitted to the list. The cluster achieved rank 52 in the Top500 [Meu+] with **421.2 TFlop/s**, 2000 GFlop/s per node, and secured and outstanding second place (see Fig. 14.12) in the Green500 [Fen+ 10] with **2351.1 MFlop/J** – only second to a very special cluster that employs the most expensive Intel eight-core processors, four single-GPUs, and 256 GB of memory per node. Unfortunately, there was no time for an improved Top500 run with the high performance code. A run after the deadline achieved 532.6 TFlop/s with 256 nodes, 2.08 TFlop/s per node (still with reduced clocks for temperature reasons). Table 14.14 summarizes the results.

Figure 14.12: Green500 Award for Sanam [Fen+ 10]

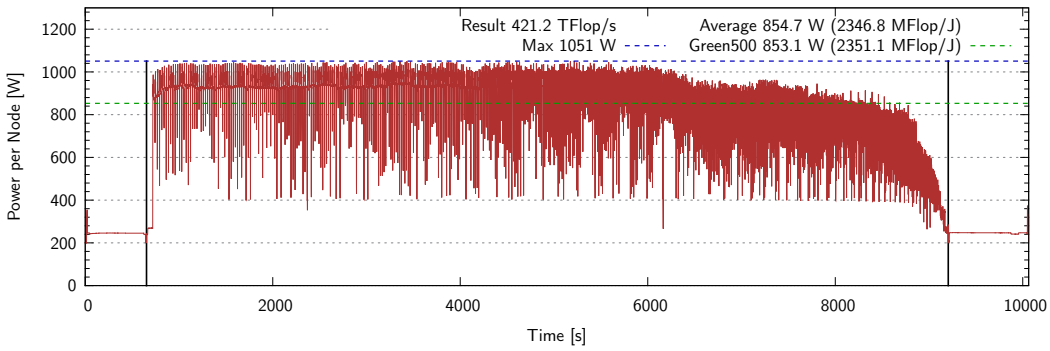


Figure 14.13: HPL Power Efficiency of Sanam Cluster reported to Green500 [XIV]

The efficiency with respect to peak performance of the four-node run is about 4 % below the corresponding LOEWE-CSC result (Table 11.58), although the new benchmark version is much

Discipline	GPU	Perf. per Node	Peak Performance	Efficiency
DGEMM Kernel	7970	805 GFlop/s	947 GFlop/s	85.0 %
DGEMM (4GPU + 2CPU)	S10000	2923 GFlop/s	3661 GFlop/s	79.8 %
Single-node HPL	S10000	2679 GFlop/s	3661 GFlop/s	73.2 %
Multi-node HPL (2 × 2)	S10000	2427 GFlop/s	3661 GFlop/s	66.3 %
256-node HPL	S10000	2080 GFlop/s	3212 GFlop/s	64.8 %

Table 14.14: Peak Performances and Efficiencies achieved on Tahiti [XIV]

more advanced. The reason is that with much faster nodes housing multiple GPUs, exploiting the full potential becomes harder and harder. Meanwhile, Intel uses most HPL-GPU features such as lookahead in their HPL with similar results [Hei+ 13]. Other GPU-enabled HPL implementations take different approaches. J. Kurzak [Kur+ 12] achieves good performance without the modifications of HPL-GPU (one MPI process per node and large N_b). It implements a tiling similar to HPL-GPU but with much smaller tile sizes and it performs more tasks on the GPU not only the DGEMM. The NVIDIA HPL implementation for the Titan supercomputer [Bla 12, Shi 13] performs the entire processing on the GPU using the processor exclusively for management. On top of that it uses a relatively small matrix that fits in GPU memory. With CPU and main memory idling, they drain much less power resulting in excellent power efficiency. With this approach, Titan achieves 64.9 % of its peak performance and demonstrates a power efficiency of 2142.8 MFlop/J.

Such approaches can boost energy efficiency as they allow to offload more work from the GPU; however, in the author's opinion, they are generally not optimal with respect to performance. GPUs excel at simple tasks such as matrix multiplication, where they can access a large fraction of their theoretical peak performance. Other tasks can benefit from GPU acceleration as well but it is very unlikely they experience an acceleration of the same order of magnitude. It is thus the best strategy to try to fully load the GPUs with DGEMM kernels all the time, process other tasks on the processor, and hide the CPU time behind GPU calculation. This has two advantages: first, the CPU can usually reach a higher fraction of its peak performance than the GPU for general tasks; second, this makes the full GPU performance available. Admittedly, during the phase at the end of the run where the CPU tasks become dominant, the other approaches are superior and would be a valuable extension for HPL-GPU in the future. Since for efficient operation they require a feature analogues to GPU Direct, which is not offered by AMD yet, they are not realized within HPL-GPU. This chapter concludes with Fig. 14.15, which shows how HPL-GPU performance has evolved on various platforms culminating in the above result for Sanam.

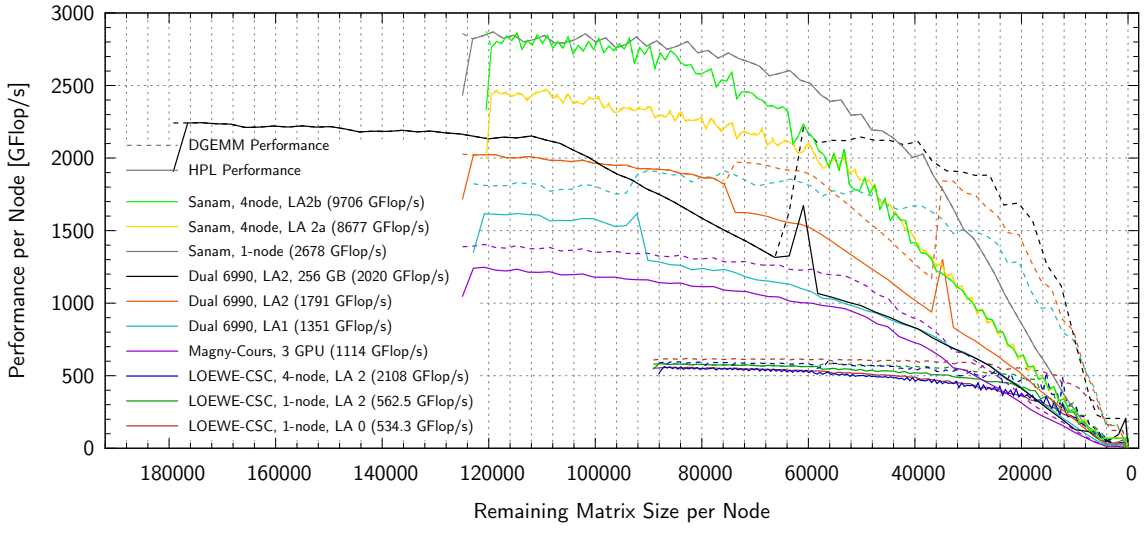


Figure 14.15: Evolution of HPL-GPU Performance – From LOEWE-CSC to Sanam

Chapter 15

Summary & Perspective for the Future

15.1 Summary

A GPU-based DGEMM has been created, integrated into HPL, and tuned specifically for the LOEWE-CSC architecture. The GPU DGEMM kernel achieves 494 GFlop/s on a 5870 AMD GPU and 624 GFlop/s on a 6970 AMD GPU, in both cases more than 90 % of the peak performance. Almost the entire DGEMM kernel performance is made available in the system. This requires well-known techniques like pipelining, asynchronous DMA transfer, and setting CPU affinities. In addition, special optimizations are applied, such as a binary patch to the AMD driver that reduces page faults. A dynamic scheduler distributes the workload among multiple GPUs and CPUs.

Meanwhile, other DGEMM implementations [Nak 10] show comparable kernel performance on Cypress GPUs (5000 series), however, only for certain transposition parameters. On top of that, they do not offer a sophisticated framework that hides PCI Express latencies and schedules CPU and GPU dynamically. With CALDGEMM the LOEWE-CSC nodes reach 623.5 GFlop/s DGEMM performance (83.6 % of peak). Multiple GPUs can be used at the same time. The multi-GPU scalability[1] is 98.35 % for two, 93.6 % for three, and 95.3 % for four GPUs.[2] In addition to GotoBLAS, AMD ACML and Intel MKL can be used as BLAS library. Patches alter the thread pinning policies such that collisions with the GPU threads do not occur. The OpenMP code in GCC has been modified to avoid unnecessary termination and recreation of threads. Besides the CAL framework, also CUDA and OpenCL are supported. The CUDA and OpenCL implementations use an advanced DMA transfer scheme that occupies significantly less system resources. As the new scheme is not generally supported by all OpenCL implementations, a fallback to the original scheme is foreseen. The CUDA DMA framework was shown to scale to at least 8 TFlop/s per node with multiple next-generation GPUs. All device code is encapsulated in the implementation of an abstract interface class making CALDGEMM easily adaptable to new hardware.

HPL has been parallelized, vectorized, and its lookahead rewritten from scratch. Lookahead involves pipelining the pivotization and hiding the factorization and broadcast times. The LOEWE-CSC nodes reach 563 GFlop/s (75.5 % of peak) and 527 GFlop/s (70.7 %) in single and multi-node configurations. The performance scales linearly to hundreds of nodes. Due to the heterogeneous nature of the LOEWE-CSC, HPL has been adapted to allow for an inhomogeneous matrix distribution among the nodes. A test setup with six nodes of three speed-grades demonstrated only 3.1 % granularity loss. The per-node performance becomes very close to the average node performance instead of being limited to the minimum node performance – which is the case for other Linpack implementations. In the November 2010 Top500 list employing HPL-GPU, the LOEWE-

[1] The scalability is defined in Section 12.6.1.2 and measures the efficiency of the multi-GPU implementation in relation to the single-GPU version.

[2] Four GPUs scale better than three because the load is distributed more evenly among the CPU sockets.

CSC ranked place 22 and in the Green500 list it ranked place eight. The efficiency of 70 % with respect to peak performance significantly exceeds the results for all competitive GPU clusters in that list, and even up until the June 2012 list no other GPU cluster had reached this benchmark. The high efficiency is facilitated by the lookahead algorithm. To the best knowledge of the author, no equally advanced implementation for the Linpack benchmark exists. An experiment with three Cypress GPUs achieved an HPL performance of 1114 GFlop/s and an average power efficiency of 1.238 GFlop/J, which would correspond to place two in the November 2010 Green500 list. This result, and an efficiency of 1.12 GFlop/J demonstrated by a low-power SDS system employing HPL-GPU, were among the first results breaking the 1 GFlop/J barrier with a GPU. A DMA problem with the Cayman GPU series was identified and a solution presented. (Appendix G lists all HPL-GPU features.) With four Cayman GPUs, an HPL performance of 2007 GFlop/s was reached.

Intel and AMD platforms have been evaluated as basis for a quad-GPU system. The Sanam cluster, each of its nodes equipped with four GPUs of the new AMD Tahiti GPU generation, was deployed at GSI in cooperation with Sebastian Kalcher and five students from the KACST institute in Riyadh. Head node, cluster operating system, and benchmark setup was jointly installed with Sebastian Kalcher. To facilitate easy benchmarking with fast deployment of new software versions to all compute nodes, a network boot setup with read-only NFS root directory was created, that can optionally mount its root directory via InfiniBand. CALDGEMM and HPL-GPU have been fine-tuned for this particular architecture, reaching a GPU kernel efficiency of 85 % with 805 GFlop/s on a 7970. Combined GPU / CPU DGEMM achieves 2923 GFlop/s. HPL reaches 2679 GFlop/s (73.2 %) in single-node and 9706 GFlop/s (66.4 %) with four nodes. This is less efficient with respect to the peak performance than on the LOEWE-CSC but it must be noted that node performance increased dramatically making it more difficult to reach high efficiency. A special HPL version optimized for energy efficiency has been developed, which enabled the Sanam cluster to achieve the second place in the November 2012 Green500 list with 2351.1 MFlop/J. With 532 TFlop/s reached by 256 nodes, which cost about $ 2.5 million including the infrastructure, Sanam facilitates a cost / performance ratio of outstanding 213 MFlops/$.

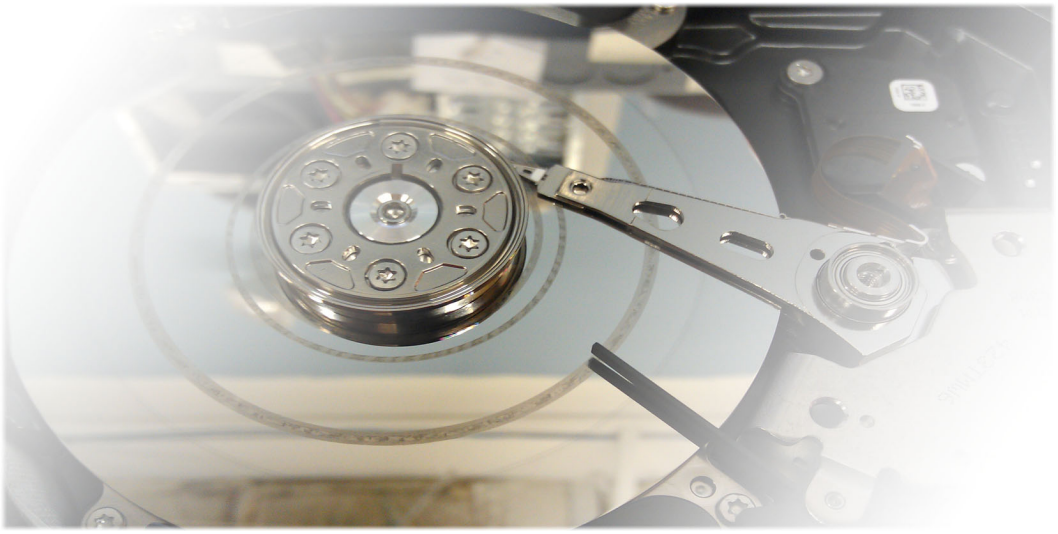
15.2 Perspective for the Future

Some possibilities for optimizations have not yet been approached or could be followed further. Instead of dynamically using CPU cores for the factorization and letting the rest idle, the free CPU cores during factorization and LASWP could be used to contribute to the large CPU DGEMM. This should be possible as it is observed that in most cases CPU DGEMM does not slow down the GPU. It is unclear, however, how to run the factorization and the DGEMM in parallel: Currently, the GotoBLAS library is not capable of running multiple independent multi-threaded tasks simultaneously. For proprietary BLAS libraries, this is even more complicated since the source code cannot be modified. Finally, with node performance moving further and further from the processors to accelerators, wasting a small fraction of the CPU performance has little impact on the overall results. The possibilities to offload the factorization or parts of it to the GPU could be analyzed further. However, this will have to be tuned for each particular configuration. It is highly improbable that a fast generic version can be implemented. For this improvement a method to transfer data directly from GPU memory via InfiniBand to remote GPU memory on a different node is essential. NVIDIA provides such a functionality with GPU Direct in CUDA 5 as does Intel for its Xeon Phi. AMD is still lacking support for this. Besides, a dynamic block size during the HPL run could decrease the CPU load and shift some work from the factorization to the update DGEMM at the end of the run when the CPU becomes the limiting factor. However, this would make the multi-node distribution extremely complex.

Recently, Intel has adopted the lookahead algorithm of CALDGEMM into their HPL implementation for the Xeon Phi inheriting many CALDGEMM features [Hei+ 13]. Currently, the work is focused on a cooperation with AMD to create an end-user Linpack version for AMD GPUs employing OpenCL based on CALDGEMM and HPL-GPU.

Part IV

Optimized High Performance Redundant Data Storage



Source of Image: [ECO].

Chapter 16

Theory

16.1 Coding Theory

In data centers, but also in every production environment, redundant data storage is of paramount importance. In case of a hard disk failure, a server crash, or a broken network link – just to mention some common examples – it must be ensured that no data are lost, and even further, that the system as a whole keeps operational. This part of the thesis is about encoding for redundant data storage and has mostly **RAID**[1] arrays of hard disks and distributed network storage on multiple servers in mind. However, the principles can also be used for failure-tolerant network communication, etc. This work is focused on the redundant encoding itself but not on its applications.

Besides linear algebra, this chapter assumes familiarity with Galois theory and p-adic numbers [Lan 05]. At first, some notation on **coding theory** is introduced. In the following, consider a fixed ring R and natural numbers $n > 0$, $k > 0$.

Definition 1: *Let R be a commutative ring with unity, $n, k \in \mathbb{N}$. An $(\boldsymbol{n}, \boldsymbol{k})$-**code** (on the ring R) is a map*

$$\widetilde{M} \colon R^n \to R^{n+k}$$

*that is injective. The code is called linear if the map is linear. In that case, $\widetilde{M}$ is called the **encoding-matrix**. If the parameters n and k are clear from the context, they are omitted. In the literature, also the alternative notation $(N', k') = (n + k, k)$ is used often.*

Denotation: *Let $M \in R^{(n+k)\times n}$ be a matrix, $I \subseteq \{1, \ldots, n+k\} \subset \mathbb{N}$ a finite set of natural numbers. Then M^I denotes the $(n + k - \#I) \times n$ submatrices of M with all rows removed whose indices lie in I.*

Definition 2: *A linear code $\widetilde{M} \colon R^n \to R^{n+k}$ is called an **MDS**-code (**M**aximum **D**istance **S**eparable) if $\widetilde{M}^I$ is regular for all $I \subset \{1, \ldots, n+k\}$ with $\#I = k$. Obviously, this means every $n \times n$ square submatrix[2] of $\widetilde{M}$ is regular. The inverses of these $n \times n$ square submatrices are called **decoding-matrices**. If only the term "decoding-matrix" is used, it will be clear from the context which of the decoding-matrices is meant. The standard denotation is $\widetilde{C}$ for code words and D for data words, s. t. $\widetilde{C} = \widetilde{M} \cdot D$.*

Definition 3: *A linear code is called **systematic** if $\widetilde{C}_i = D_i$ for $1 \leqslant i \leqslant n$ and $\widetilde{C} = \widetilde{M} \cdot D$. Two linear codes $\widetilde{M}$ and $\widetilde{M}'$ are called equivalent if there is a regular matrix A such that $\widetilde{M} = \widetilde{M}' \cdot A$.*

Lemma 4: *Every linear MDS-code is equivalent to a systematic code.*

Proof: For an MDS-code $\widetilde{M}$, the square submatrix $A := \widetilde{M}^{[n+1\colon n+k]}$ is regular with inverse A^{-1}. Then $\widetilde{M} \cdot A^{-1}$ is an equivalent systematic code. □

[1] **R**edundant **A**rray of **I**nexpensive **D**isks – or – **R**edundant **A**rray of **I**ndependent **D**isks.
[2] A square submatrix of a matrix M is a square matrix obtained by removing lines and columns of M.

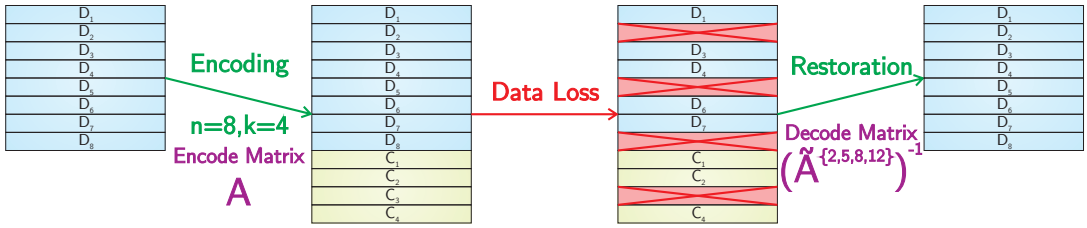


Figure 16.1: Failure Tolerant Encoding / Decoding Scheme[3]

From now on, every code is assumed to be a linear MDS-code. Due to Lemma 4, without loss of generality, the code can be assumed systematic. In applications, there are n data words $D_i \in R$ with $1 \leqslant i \leqslant n$ to be stored. There are $n + k$ data storages available. Each of them stores one of the code words $\widetilde{C}_i$ $(1 \leqslant i \leqslant n + k)$ that are defined by $\widetilde{C} = \widetilde{M} \cdot D$. Due to the postulation in Definition 2, the data words can be recovered by any set of n code words, i. e. the system survives the failure of up to k storages without any data loss. Consider that the lost data words' indices must be known since erasure codes do not have to detect data corruption by definition.[3] Fig. 16.1 visualizes the process. The encoding-matrix of a systematic MDS-code $\widetilde{M}$ has the form

$$\widetilde{M} = \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \\ \hdashline & M & \end{pmatrix} = \begin{pmatrix} \mathbb{1} \\ M \end{pmatrix}$$

and the encoding yields

$$\widetilde{M} \cdot D = \begin{pmatrix} \mathbb{1} \\ M \end{pmatrix} \cdot D = \begin{pmatrix} D \\ C \end{pmatrix} = \widetilde{C}.$$

The code is determined completely by the $k \times n$ matrix M. Despite the ambiguity, M is called encoding-matrix as well. The encoding equation simplifies to $C = M \cdot D$.

Definition 5: *A matrix whose every square submatrix is regular is called* ***locally regular*** *(or locally invertible). In particular, every matrix entry is a* 1×1 *submatrix itself. Thus, all entries of a locally regular matrix are units and especially nonzero.*

Lemma 6: *A matrix* M *yields an MDS-code* $\widetilde{M} = \left(\begin{smallmatrix} \mathbb{1} \\ M \end{smallmatrix}\right)$ *if and only if it is locally regular.*

Proof: It must be verified whether every $n \times n$ square submatrix of $\widetilde{M}$ is regular. (Definition 2 poses no requirement on smaller square submatrices.) Each such $n \times n$ square submatrix has the form

$$\begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & & & & & 0 & 1 \\ & & & & & & & & \ddots \\ \hdashline & & & & & M^* & & & \end{pmatrix}$$

with a submatrix M^* of M. By elementary row and column operations the form

$$\begin{pmatrix} \mathbb{1} & 0 \\ 0 & M^{**} \end{pmatrix}$$

can be obtained. This is invertible if and only if the square submatrix M^{**} of M is regular. Assume $I \subseteq \{1, \ldots, n+k\}$ with $\#I = k$. Define $I_1 = I \cap [1\colon n]$ and $I_2 = I \cap [n+1\colon n+k]$. Verifying whether the $n \times n$ square submatrix $\widetilde{M}^I$ of $\widetilde{M}$ is regular leads to the square submatrix M^{**} of M with the rows removed whose index shifted by n lies in I_2 and with the columns removed whose index lies in I_1. Thus, there is a one to one correspondence between the square submatrices M^{**} of M and the sets I. A square submatrix M^{**} is regular if and only if the corresponding matrix $\widetilde{M}^I$ is regular. All square submatrices can be obtained by removing rows and columns hence all square submatrix M^{**} must be regular, which completes the proof. □

16.2 Reed-Solomon Code

Reed-Solomon Codes [Ree$^+$ 60] are examples of linear MDS-codes. It is known that for each prime power $q = p^l$ there is (up to isomorphism) a unique finite field $\mathbb{F}_q$ with q elements. It is an algebraic Galois extension of $\mathbb{F}_p \cong \mathbb{Z}/p\mathbb{Z}$, i. e. $\mathbb{F}_q \cong \mathbb{F}_p[X]/f(x)$ with $f(x) \in \mathbb{F}_p[X]$ an irreducible polynomial of degree l.

Definition 7: *Let $\lambda_i \in R$ for $0 \leqslant i \leqslant n$. The **Vandermonde matrix***

$$V = \begin{pmatrix} 1 & \lambda_0 & \lambda_0^2 & \cdots & \lambda_0^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \lambda_n & \lambda_n^2 & \cdots & \lambda_n^n \end{pmatrix}$$

is defined by $V_{ij} = \lambda_i^j$.

There is the following well known

Lemma 8: *The determinant of the Vandermonde matrix V is*

$$\det(V) = \prod_{0 \leqslant i < j \leqslant n} (\lambda_j - \lambda_i).$$

It is invertible if and only if all $\lambda_j - \lambda_i$ are units. Therefore, if R is a field, the matrix V is regular if and only if the λ_i are pairwise distinct.

Also the inverse of the Vandermonde matrix is known explicitly [Tur 66]. Now, the Reed-Solomon code can be introduced.

Definition & Proposition 9: *Assume $q \geqslant n+k$ and consider the matrix $\widetilde{M}$ defined by*

$$\widetilde{M} = \begin{pmatrix} \lambda_1^0 & \cdots & \lambda_1^n \\ \vdots & \ddots & \vdots \\ \lambda_{n+k}^0 & \cdots & \lambda_{n+k}^n \end{pmatrix}$$

with pairwise distinct $\lambda_i \in \mathbb{F}_q$. This defines an MDS-code: the Reed-Solomon code.

Proof: By construction, every $n \times n$ square submatrix is a Vandermonde matrix with pairwise distinct λ_i and thus regular. Therefore, $\widetilde{M}$ is a linear MDS-code. □

[3] The theory of erasure codes can be extended to error-correction-codes which can detect a corruption of up to a certain number of data words and can also identify the corrupted ones. Hamming codes are one example. Another possibility is to store checksums for each chunk, such that corrupted storages can be identified easily [Kal 13, 5.9].

16.3 Integer Calculations & Codes on finite Rings

From the implementation perspective, the Reed-Solomon code suffers from the fact that computers do not natively support calculations in the field $\mathbb{F}_q$. Usually, q is chosen to be 2^{2^α} which maps nicely to the CPU registers and the calculation in $\mathbb{F}_{2^{2^\alpha}}$ is emulated. In this case, the addition in $\mathbb{F}_{2^{2^\alpha}}$ maps directly to a logical *XOR* operation. However, the multiplication involves a polynomial division and is thus quite expensive. It is usually implemented using logarithm tables [Pla 97], which hardly fit into CPU caches. S. Kalcher [Kal$^+$ 11] demonstrates how to overcome the memory issue with a vectorized direct approach, however, the emulated operation is still much slower than native integer or floating point operations.

It is thus desired to have an encoding scheme that only performs integer operations the CPU has native support for.[4] Consider a processor that operates on b-bit registers, e. g. 32 bits. In fact, such a CPU does not really support full integer calculation but it encounters overflows when exceeding 2^b. This is equivalent to operations in the residue class ring $\mathbb{Z}/2^b\mathbb{Z}$. Nevertheless, such calculations are called integer calculations in the following.

The following two subsections present two constructions of integer codes. The first one was developed during this thesis. It uses integral $\mathbb{Z}$-algebras and algebraic number fields in order to derive codes over finite rings. This is the approach the implementation in Chapter 17 follows. Codes can be constructed and implemented explicitly this way. The second construction is more direct and in fact more general as it can handle rare cases which cannot be treated by the first one. In the end, both methods lead to the same encoding matrices. The codes are also described e. g. in [And$^+$ 05], where the derivation is similar but not identical to the second approach presented here. The main advantage of the first approach is a fast inversion method for ring elements and thus a faster encoding-matrix generation than the other approaches provide.

16.3.1 Deriving Codes from Algebraic Number Fields

This section derives codes from commutative $\mathbb{Z}$-algebras with unity and without zero divisors (i. e. the algebra is an integral domain), which are finitely generated as $\mathbb{Z}$-modules. It is clear that all of the above postulations are natural, e. g. computation in algebras which are not finitely generated is possible on computers only in special cases. At first, only rings of integers of algebraic number fields are treated. It is proven later (see Section 16.3.1.4) that this is no restriction.

16.3.1.1 Integrality

In the following, K always denotes a number field, i. e. a finite algebraic extension of $\mathbb{Q}$, of degree l.

Definition 10: *The set of all elements*

$$\mathcal{O}_K = \Big\{ z \in K \quad s.\,t. \quad \exists\ f(X) \in \mathbb{Z}[X] \ \textit{normalized}^5 \colon \ f(z) = 0 \Big\},$$

is called the ***ring of algebraic integers*** *of the number field K.*

It is a well-known fact [Neu 99, I §2] that this set is a finitely generated $\mathbb{Z}$-module of rank l and carries the structure of a ring. A basis of this module is called an **integral basis**. Clearly, it is an integral domain and thus lies in the above described category. Since $\mathbb{Z}$ is a principal ideal domain and $\mathcal{O}_K$ is torsion-free, $\mathcal{O}_K$ is a free $\mathbb{Z}$-module, i. e. $\mathcal{O}_K \cong \mathbb{Z}^l$ as a $\mathbb{Z}$-module. It follows that for $p \in \mathbb{Z}$ prime

$$\mathcal{O}_K/p^b\mathcal{O}_K \cong \left(\mathbb{Z}/p^b\mathbb{Z}\right)^l \quad \text{(as an abelian group)}.$$

[4] Floating point computation is by definition unsuited for encoding since it is not associative.
[5] A polynomial is normalized if its highest coefficient is one.

In the following, ${}^{K}/_{\mathbb{Q}}$ is assumed a Galois extension with Galois group $G = \mathrm{Gal}({}^{K}/_{\mathbb{Q}})$ and $\mathfrak{P} = p\mathcal{O}_K$ is the ideal generated by p in $\mathcal{O}_K$. It shall be analyzed when a Vandermonde matrix V (respectively its residue class) with entries in $\mathcal{O}_K$ is invertible in the matrix ring over ${}^{\mathcal{O}_K}/_{p^b\mathcal{O}_K}$. According to Cramer's rule, this is the case if and only if $\det(V)$ is invertible (in ${}^{\mathcal{O}_K}/_{p^b\mathcal{O}_K}$).

In general, $\mathcal{O}_K$ is not a unique factorization domain[6], but it can be shown that it is a Dedekind domain, and thus there is a unique prime ideal factorization instead [Neu 99, I 3.2 / 3.3].

Proposition 11: *Assume $\mathfrak{P}$ is a prime ideal in $\mathcal{O}_K$ and V a Vandermonde matrix, such that the λ_i are pairwise distinct modulo $\mathfrak{P}$. Then V is invertible in ${}^{\mathcal{O}_K}/_{\mathfrak{P}^b}$.*

Proof: By construction the differences $\lambda_j - \lambda_i$ are not divisible by $\mathfrak{P}$ thus neither is

$$\mathcal{N}(\det(V)) = \prod_{\sigma\in G} \sigma(\det(V)) = \prod_{\sigma\in G} \prod_{1\leqslant i<j\leqslant n} \sigma(\lambda_j - \lambda_i)$$

since $\mathfrak{P}$ is prime. Using the extended Euclidean algorithm there exist integers $x, y \in \mathbb{Z}$ such that $x \cdot \mathcal{N}(\det(V)) + y \cdot p^b = 1$. This yields that $\det(V)$ is invertible in ${}^{\mathcal{O}_K}/_{\mathfrak{P}^b}$ with inverse

$$x \cdot \prod_{1\neq\sigma\in G} \sigma(\det(V)). \qquad \square$$

Before a code can be generated from this, a little more theory is needed. There is the following [Neu 99, I §3] well-known

Definition & Proposition 12: *A prime number $p \in \mathbb{Z}$ factors uniquely in $\mathcal{O}_K$ as*

$$p\mathcal{O}_K = \prod_{i=1}^{r} \mathfrak{P}_i^{\,e}$$

with $\mathfrak{P}_i \lhd \mathcal{O}_K$ prime where the index calculates to

$$f = \dim_{\mathbb{F}_p} {}^{\mathcal{O}_K}/_{\mathfrak{P}_i\mathcal{O}_K} \quad \forall i$$

with $e \cdot f \cdot r = l$. It is said that p splits in r prime ideals of ramification index e and inertia degree f. The prime p is called unramified if $e = 1$, it does not split if $r = 1$, and it is inert if $f = l$. (If ${}^{K}/_{\mathbb{Q}}$ is not Galois, a similar proposition holds.)

By definition a prime number $p \in \mathbb{Z}$ generates a prime ideal $\mathfrak{P} = p\mathcal{O}_K \lhd \mathcal{O}_K$ if and only if it is inert.

16.3.1.2 An MDS-Code on Residue Class Rings

Proposition 13: *Assume p is inert in $\mathcal{O}_K$ and $p^l \geqslant n + k$. Then there is a linear MDS-code on ${}^{\mathcal{O}_K}/_{\mathfrak{P}^b\mathcal{O}_K}$ for all $b \geqslant 1$.*

Proof: Since p is inert, the equation $l = f = \dim_{\mathbb{F}_p} {}^{\mathcal{O}_K}/_{\mathfrak{P}\mathcal{O}_K}$ holds. Thus, there are $p^l \geqslant n + k$ elements λ_i pairwise distinct modulo p. Using the encoding-matrix $\widetilde{M}_{ij} = \lambda_i^j$ the claim follows from Proposition 11 in the same way as for the Reed-Solomon code. $\square$

Remark 14: *Assume either $p^l < n+k$ or both $p^l \geqslant n+k$ with l minimal and p not inert in $\mathcal{O}_K$. In both cases, there are less than $n + k$ pairwise disjoint elements available and thus the above construction cannot lead to an MDS-code.*

[6] A counter-example is $K = \mathbb{Q}(\sqrt{-5})$ where $6 = 2\cdot 3 = (1+\sqrt{-5})\cdot(1-\sqrt{-5})$ has no unique prime number factorization.

In order to minimize the computational complexity, it is desirable to choose l as small as possible. The natural number l is called admissible if there exists a Galois extension of degree l such that p is inert. This requires p not to split and to be unramified. The question arises which numbers l are admissible.

Lemma 15: *Every factor of an admissible number l is admissible itself.*

Proof: Since p does not split, the local extension ${}^{K_\mathfrak{P}}/_{\mathbb{Q}_p}$ has the same degree l with isomorphic Galois group. Since the extension is unramified, the Galois group must be abelian and even cyclic. Therefore, every subextension $\mathbb{Q} \subseteq L \subseteq K$ is Galois as well. Clearly, if p is inert in K, the same holds in L. Since for every factor l' of l there exists a subextension of degree l', if l is admissible, every factor is. □

The question which l are admissible is answered (with few exceptions) by the following

Theorem 16 (Grunwald Wang)**:** *Let S be a finite set of primes. Let G be a finite abelian group. For all $p \in S$ let ${}^{K_\mathfrak{P}}/_{\mathbb{Q}_p}$ be local Galois extensions with Galois group G_p such that $G_p \subseteq G$ are subgroups. Assume either $2 \notin S$ or $\#G_2 \notin 8\mathbb{Z}$. Then there exists a Galois extension ${}^{K}/_{\mathbb{Q}}$ with Galois group G such that for all p the local extensions coincide with the predetermined $K_\mathfrak{P}$.*

The theorem in fact applies to an even more general case. The general formulation and a proof can be found in [NSW 08, 9.2.8]. With the theorem of Grunwald Wang, the existence of admissible l can be proven in almost every case.

Proposition 17: *Let $p \neq 2$ or $l \notin 8\mathbb{Z}$. Furthermore, assume $p^l \geqslant n + k$. Then l is admissible and an MDS-code exists.*

Proof: Apply Theorem 16 with $S = \{p\}$ and $G = G_p = {}^{\mathbb{Z}}/_{l\mathbb{Z}}$ with ${}^{K_\mathfrak{P}}/_{\mathbb{Q}_p}$ unramified.[7] In the resulting extension K, the prime p does not split since $G = G_p$ and it is unramified by construction. Hence p is inert. With Proposition 13 the claim follows. □

In the following, only the case $p = 2$ is regarded. First, this is the special case in the theorem of Grunwald Wang and second, it is the case relevant for the implementation on a computer. The Grunwald Wang theorem does not make any assertion for the special case so the question arises whether $l = 8$ could still be admissible for $p = 2$. Unfortunately, this is not the case as is shown in [AT 67, §10 Theorem 1]. Due to Lemma 15, every multiple of 8 is not admissible, either. Hence in summary, l is admissible if and only if it is no multiple of 8.

As a remark: from Section 16.3.2 it follows that every l is admissible if the assumption is dropped that ${}^{K}/_{\mathbb{Q}}$ is Galois. In contrast, the above theory can only cover the cases where l is not divisible by eight. Appendix F.1 lists examples how codes from Proposition 17 can be generated for many l.

16.3.1.3 Codes on ${}^{\mathbb{Z}}/_{p^b\mathbb{Z}}$ (Integral Codes)

Reflecting the last section reveals that the implementation of the above algorithm requires calculations in the ring $\mathcal{O}_K$ which usually involves polynomial division as for the Reed-Solomon code. However, there is a simple approach to overcoming this. As an additive group, $\mathcal{O}_K$ is isomorphic to $\mathbb{Z}^l$ and every $\mathcal{O}_K$-linear map φ is clearly $\mathbb{Z}$-linear. The data words D_i and the redundancy data words C_i are considered elements of $\mathbb{Z}^l$. The $k \times n$ encoding-matrix M with entries in ${}^{\mathcal{O}_K}/_{2^b\mathcal{O}_K}$ thus yields a $kl \times nl$ matrix with entries in ${}^{\mathbb{Z}}/_{2^b\mathbb{Z}}$. The new matrix is called encoding-matrix and is denoted by M, too. Such codes are also called **integral codes**.

Later, the implementation operates on 2^b-bit values interpreted as elements of ${}^{\mathbb{Z}}/_{2^b\mathbb{Z}}$. Consecutive l elements are considered an element of ${}^{R}/_{2^bR} \cong ({}^{\mathbb{Z}}/_{2^b\mathbb{Z}})^l$ and are stored on the same storage each. Such a set of l elements makes up a data word D_i or a redundancy word C_i. These words

[7] From class field theory it is known that an unramified extension of $\mathbb{Q}_p$ of degree l exists [Neu 99, IV §5].

are stored on mutually exclusive storages. This means, losing one storage results in losing one of the data words D_i or one of the redundancy words C_i, i. e. after losing arbitrary k storages the original data can in any case be reconstructed. The matrix multiplication for the encoding can be performed by every processor by simple integer multiplications and additions. It shall be noted here that the occurrence of integer overflows is essential. Naturally, the above considerations hold for the decoding-matrix as well. In formula, the encoding is done via:

$$C = \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (C_1)_1 \\ \vdots \\ (C_1)_l \end{pmatrix} \\ \vdots \\ \begin{pmatrix} (C_k)_1 \\ \vdots \\ (C_k)_l \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (M_{11})_{11} \cdots (M_{11})_{1l} \\ \vdots \quad \ddots \quad \vdots \\ (M_{11})_{l1} \cdots (M_{11})_{ll} \end{pmatrix} \cdots \begin{pmatrix} (M_{1n})_{11} \cdots (M_{1n})_{1l} \\ \vdots \quad \ddots \quad \vdots \\ (M_{1n})_{l1} \cdots (M_{1n})_{ll} \end{pmatrix} \\ \vdots \qquad \ddots \qquad \vdots \\ \begin{pmatrix} (M_{k1})_{11} \cdots (M_{k1})_{1l} \\ \vdots \quad \ddots \quad \vdots \\ (M_{k1})_{l1} \cdots (M_{k1})_{ll} \end{pmatrix} \cdots \begin{pmatrix} (M_{kn})_{11} \cdots (M_{kn})_{1l} \\ \vdots \quad \ddots \quad \vdots \\ (M_{kn})_{l1} \cdots (M_{kn})_{ll} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \begin{pmatrix} (D_1)_1 \\ \vdots \\ (D_1)_l \end{pmatrix} \\ \vdots \\ \begin{pmatrix} (D_n)_1 \\ \vdots \\ (D_n)_l \end{pmatrix} \end{pmatrix} = MD$$

where $C_i, D_j \in (\mathbb{Z}/2^b\mathbb{Z})^l$ are tuples of l associated 2^b-bit words and the entries $M_{ij} \in (\mathbb{Z}/2^b\mathbb{Z})^{l \times l}$ are $l \times l$ matrices themselves with 2^b-bit entries.

The above-obtained code does not exactly fit the definition of an MDS-code on $\mathbb{Z}/2^b\mathbb{Z}$ but it is quite similar. Instead of tolerating the loss of k single words, the loss of k tuples of l consecutive words is tolerated. In the following, codes of this kind are called **vector-MDS**-codes (over the base ring $\mathbb{Z}/2^b\mathbb{Z}$).

16.3.1.4 The general Case

Proposition 18: *Every code obtained using an integral domain R which is finitely generated as a $\mathbb{Z}$-module can be obtained from the ring of algebraic integers $\mathcal{O}_K$ of a number field K.*

Proof: The integral domain R is torsion-free and thus free over the principal ideal domain $\mathbb{Z}$. Therefore, assume its dimension is r and let $\{e_i\}_{1 \leqslant i \leqslant r}$ be a basis. For fixed i consider the powers e_i^j. The module created by all these powers is a submodule of R and thus finitely generated since $\mathbb{Z}$ is a principal ideal domain. Hence, there exists m such that e_i^0, ...,e_i^m are linearly dependent and thus e_i is integral over $\mathbb{Z}$, i. e. $\exists_{\alpha_j \in \mathbb{Z}}$ s. t. $\sum_{j=0}^{m} \alpha_j e_i^j = 0$ and at least one of the α_j is nonzero. Without loss of generality, assume $\alpha_0 \neq 0$. (Consider that R is free and factor out e_i.) The quotient field of R is hence a finite algebraic extension K of $\mathbb{Q}$ and has a ring of algebraic integers $\mathcal{O}_K$. It is clear that R is a subring of $\mathcal{O}_K$ but it is not necessarily identical.

Since R is a free $\mathbb{Z}$ module generated by $\{e_i\}_{1 \leqslant i \leqslant r}$, also $R \otimes_{\mathbb{Z}} \mathbb{Q}$ is a free $\mathbb{Q}$ module generated by $\{e_i \otimes 1\}_{1 \leqslant i \leqslant r}$ and thus a $\mathbb{Q}$-vector space of dimension r. In $R \otimes_{\mathbb{Z}} \mathbb{Q}$, this yields:

$$(e_i \otimes 1) \cdot \frac{-1}{\alpha_0} \sum_{j=1}^{m} \alpha_j (e_i \otimes 1)^{j-1} = 1.$$

An analogous equation holds for every element $0 \neq x \in R$.[8] Hence $R \otimes_{\mathbb{Z}} \mathbb{Q}$ is already a field and thus $R \otimes_{\mathbb{Z}} \mathbb{Q} \cong K$. In total, this yields: $r = \dim_{\mathbb{Z}}(R) = \dim_{\mathbb{Q}}(R \otimes_{\mathbb{Z}} \mathbb{Q}) = \deg({}^{K}/_{\mathbb{Q}}) = \operatorname{rank}_{\mathbb{Z}}(\mathcal{O}_K)$. Therefore, R and $\mathcal{O}_K$ have the same rank and are isomorphic as $\mathbb{Z}$-modules but not necessarily as algebras. Still, there is a natural inclusion $\iota \colon R \hookrightarrow \mathcal{O}_K$, which can be lifted (componentwise) to the matrix algebras $\iota^* \colon R^{n \times n} \hookrightarrow \mathcal{O}_K^{n \times n}$ and an endomorphism φ of R^n with matrix representation $A \in R^{n \times n} \subseteq \mathcal{O}_K^{n \times n}$ extends to an endomorphism of $\mathcal{O}_K^n$ with representation $\iota^*(A)$.

The above paragraphs proved that R is a subalgebra of $\mathcal{O}_K$ of the same rank. According to the structure theorem for finitely generated modules over principle ideal domains [Lan 05, 7.8], there

[8] The module generated by the powers of x is a finitely generated submodule of R since $\mathbb{Z}$ is a principal ideal domain.

are bases $\{e_i\}_{1\leqslant i\leqslant r}$ of R and $\{e'_i\}_{1\leqslant i\leqslant r}$ of $\mathcal{O}_K$ such that $e_i = \beta_i \cdot e'_i$ with $\beta_i \in \mathbb{Z}$. These bases yield $\mathbb{Z}$-module isomorphisms ψ_R and $\psi_{\mathcal{O}_K}$ and the following commutative diagram.

$$\begin{array}{ccc} R^{n\times n} & \xrightarrow{\quad\iota^*\quad} & \mathcal{O}_k^{n\times n} \\ {\scriptstyle\psi_R}\downarrow\wr & & {\scriptstyle\psi_{\mathcal{O}_K}}\downarrow\wr \\ \mathbb{Z}^{nr\times nr} & \xrightarrow{\quad\iota^{**}\quad} & \mathbb{Z}^{nr\times nr} \end{array}$$

A simple calculation reveals[9] $\iota^{**} = \mathrm{Id}$, i. e. every encoding-matrix obtained through R can be obtained through $\mathcal{O}_K$ as well. □

16.3.2 Deriving Codes from Finite Field MDS-Codes

The above derivation generates codes explicitly (Appendix F.1). However, the parameter l is in a certain sense restricted. The goal is to create a code for words in $\mathbb{Z}/p^b\mathbb{Z}$ for arbitrary l. Consider the fixed finite field $\mathbb{F}_q = \mathbb{F}_{p^l}$ and an MDS-code (e. g. Reed-Solomon) $\widetilde{M}$ on $\mathbb{F}_q$.

Proposition 19: *Every MDS-code on $\mathbb{F}_q$ ($q = p^l$) can be lifted to an MDS-code on a $\mathbb{Z}/p^b\mathbb{Z}$-algebra of dimension l.*

Proof: There is an irreducible polynomial $f(x) \in \mathbb{F}_p[X]$ such that $\mathbb{F}_q \cong \mathbb{F}_p[X]/f(x)$. It is clear that $\mathbb{F}_q$ is a vector space of dimension l over $\mathbb{F}_p$. Consider a normalized lift $\overline{f}(x) \in \mathbb{Z}/p^b\mathbb{Z}[X]$ of $f(x)$. Define $R = (\mathbb{Z}/p^b\mathbb{Z})/\overline{f}(x)$. Clearly, R is a free $\mathbb{Z}/p^b\mathbb{Z}$-module of dimension l. By construction, $R/pR \cong \mathbb{F}_q$ is a field and p is maximal. Conversely, for any maximal ideal $\mathfrak{p}$, the quotient $R/\mathfrak{p}$ is a field whose characteristic can only be p. Therefore, $\mathfrak{p} = pR$ and R is a local ring. Take an arbitrary lift $\overline{M} \in R^{(n+k)\times n}$ of $\widetilde{M} \in \mathbb{F}_q^{(n+k)\times n}$. The projection of the determinant $\det(\overline{M}^I)$ of an $n \times n$ submatrix of $\overline{M}$ equals the determinant of the corresponding submatrix of $\widetilde{M}$ and is thus nonzero. It follows that $\det(\overline{M}^I)$ does not lie in the maximal ideal and is thus invertible in R. Thus, $\overline{M}$ is locally regular and yields an MDS-code on R. □

Remark 20: *There are the following properties of vector-MDS-codes derived from finite fields:*

(i) In the same way as in Section 16.3.1.3, a vector-MDS code can be obtained ($\mathbb{F}_{p^l}$ is regarded as vector space over $\mathbb{F}_p$). Even further, analogously to the proof of Proposition 19, each vector-MDS-code on $\mathbb{F}_p$ can be directly lifted to $\mathbb{Z}/p^b\mathbb{Z}$ by lifting the encoding-matrix $\widetilde{M}$.

(ii) As a warning it should be noted that in general, an arbitrary lift of the decoding-matrix does not lead to the decoding-matrix for a lifted MDS-code.

(iii) Consider a vector-MDS-code with $p = 2$. As the lift of the encoding-matrix can be chosen arbitrarily, it is possible to choose the lift of the encoding-matrix from $\mathbb{F}_2^{n+k\times k}$ such that all entries are either 0 or 1.

For actually generating both encoding and decoding matrices, it is necessary to invert units of R. Since R is a finite ring, elements can be inverted in finite time using brute force. However, for large rings this is not possible or practical respectively in practice. In Proposition 11, the units are inverted by multiplication with Galois conjugates. In the case of a non-Galois extension, this is naturally not possible. Still, there is an algorithm for inverting elements in acceptable runtime.

[9] Naturally, ι^{**} depends on the choice of the basis$\{e_i\}_{1\leqslant i\leqslant r}$, but every change of basis matrix applied to the basis $\{e_i\}_{1\leqslant i\leqslant r}$ can be applied to $\{e'_i\}_{1\leqslant i\leqslant r}$ as well yielding the same ι^{**}.

Lemma 21: *Consider the situation from Proposition 19.*

$$R = \left(\mathbb{Z}/p^b\mathbb{Z}\right)/\overline{f}(x) \quad \mathbb{F}_q \cong \mathbb{F}_p[X]/f(x) \cong \mathbb{Z}[X]/(\ p\mathbb{Z}, \overline{f}(x)\ ) \cong R/pR$$

Take a unit $x \in R^*$. The units of R are all elements of $R \backslash pR$. By the theorem of Lagrange it follows that $x^\gamma = 1$ for $\gamma = \#R^* = p^{bl} - p^{(b-1)l}$. The inverse can thus be calculated by $x^{-1} = x^{\gamma-1}$.

16.3.3 Summary

Yet, two different approaches have been presented for creating codes on finite rings. The algebraic number field approach generates a subset of the encoding matrices obtained from the second approach. Still, it covers all cases with $l \notin 8\mathbb{Z}$, which is sufficient in almost every case. During generation of encoding and decoding matrices, inversion of ring elements is a compute intensive task. In both approaches, this task is put down to multiplication in the ring, which is compute intensive itself. (Ring multiplication is explained in Appendix F.2. It is performed distributively requiring $(l-1)^2$ integer multiplications and needs a subsequent reduction step of at least the same complexity.) Hence, the number of ring multiplications is a suitable measure for the performance. The following lemma gives a partial answer which approach is faster.

Lemma 22: *In terms of ring multiplications, the method involving algebraic number fields (Proposition 17) is in average at least $\frac{3b}{2} \ln_2 p$ times faster then the method in Lemma 21.*

Proof: The number of ring multiplications for the algebraic number field approach can be counted easily in Equation F.2 in Appendix F.2.1: it is $l-2$. The finite field method requires computing the $(p^{bl} - p^{(b-1)l} - 1)^{\text{th}}$ power, which can be done using binary exponentiation (see [Knu 97, 4.6.3]) in $\alpha \cdot \ln_2(p^{bl} - p^{(b-1)l} - 1)$ steps with $1 \leqslant \alpha \leqslant 2$. For $p \geqslant 2$, $l \geqslant 3$, $b \geqslant 2$ the quotient calculates to:

$$\frac{\alpha \cdot \ln_2(p^{bl} - p^{(b-1)l} - 1)}{l-2} = \alpha \frac{\ln_2(p^{bl}) + \ln_2(1 - p^{-l} - p^{-bl})}{l-2} \quad \geqslant$$

$$\alpha b \ln_2(p) \cdot \frac{1 + \ln_2(1 - p^{-l} - p^{-bl})}{1 - 2/l} \geqslant \alpha b \ln_2(p) \cdot \frac{1 + \ln_2(1 - 2^{-3} - 2^{-6})}{1 - 2/l} \geqslant \alpha b \ln_2(p). \quad \square$$

The lemma does not take the costs of the operation of the Galois group into account, which depend on the particular field. However, since the Galois group acts linearly, the operation can be performed via matrix multiplication (on the ring as a $\mathbb{Z}$-module). Therefore, calculating the Galois conjugates is less than or equally expensive as multiplication in the ring. The lemma also does not consider the extended Euclidean algorithm at all, which is calculated in $\mathbb{Z}$ though, and thus very fast.

In the cyclotomic example in Appendix F.1, calculation of the Galois conjugates takes zero time since the Galois group only permutes the basis vectors $\zeta_m^j = e^{2\pi i \cdot j/m}$. In addition, the relation between the powers of ζ_m makes most of the reduction steps for ring multiplication obsolete. Thus, the speed advantage for cyclotomic fields is much higher than stated in Lemma 22.

The following proposition concludes the section, taking into account the explicit example in Appendix F.1 and the calculations in Appendix F.2:

Proposition 23: *Let $b \in \mathbb{N}$, p prime, and $n, k, l \in \mathbb{N}$ such that $p^l \geqslant n + k$:*

- *There exists an (n,k)-vector-MDS-code on the ring $\mathbb{Z}/p^b\mathbb{Z}$.*
- *The code can be derived from the ring of integers of an algebraic number field if and only if l is no multiple of 8 (i. e. $l \notin 8\mathbb{Z}$) or $p \neq 2$.*
- *Every code that can be obtained from an integral domain which is finitely generated as a $\mathbb{Z}$-module can already be obtained from a ring of algebraic integers.*

- *Cyclotomic fields yield codes for $l \in \{1, 2, 4, 6, 10, 12, 18, 20, 28\}$. Their totally real subfields yield additional codes for $l \in \{3, 5, 9, 14\}$.*
- *If the code can be derived from an algebraic number field, ring elements can be inverted easily using multiplication with Galois conjugates and the extended Euclidean algorithm as described in Appendix F.2.1. If the code cannot be derived from an algebraic number field, inverses can still be calculated using the theorem of Lagrange as described in Lemma 21.*
- *Using algebraic number fields results in an about $\frac{3b}{2} \ln_2 p$ times faster inversion algorithm. For cyclotomic fields, the factor is even higher.*

16.4 Cauchy-Reed-Solomon Code

Definition & Proposition 24: *Let $a_i, b_j \in R$ $(1 \leqslant i \leqslant k, 1 \leqslant j \leqslant n)$ be pairwise distinct elements. The matrix*

$$M = \begin{pmatrix} \frac{1}{a_1 - b_1} & \cdots & \frac{1}{a_n - b_1} \\ \vdots & \ddots & \vdots \\ \frac{1}{a_1 - b_k} & \cdots & \frac{1}{a_n - b_k} \end{pmatrix}$$

*defined by $M_{ij} = \frac{1}{a_i - b_j}$ is called a **Cauchy matrix**. In the case that $n = k$ and that R is a field, the determinant calculates to*

$$\det(M) = \frac{\prod_{i=2}^{n} \prod_{j=1}^{i-1} (a_i - a_j)(b_i - b_j)}{\prod_{i=1}^{n} (a_i - b_i)} \tag{16.1}$$

and is nonzero for pairwise distinct a_i and b_j. Clearly, every submatrix of a Cauchy matrix is a Cauchy matrix itself and a Cauchy matrix is locally regular. With Lemma 6 it follows that each Cauchy matrix yields an MDS-code.

A proof for the determinant formula can be found in [Sch 59]. By specializing the principle of a vector-MDS-code based on a Cauchy matrix to $b = 1$ (or alternatively by applying the vector-MDS principle to the original Reed-Solomon code with a Cauchy matrix instead of the Vandermonde matrix), the **Cauchy-Reed-Solomon code (CRS)** [Blö$^+$ 95] is obtained. The integral matrix multiplication from Section 16.3.1.3 becomes a matrix multiplication with binary entries in $\mathbb{F}_2$. In the following, vector-MDS-codes over $\mathbb{F}_2$ are called CRS codes, regardless of which matrix is used. The implementation in the next chapter still uses the Vandermonde matrix.

16.4.1 *XOR*-only Codes

The binary matrix multiplication of the Cauchy-Reed-Solomon code has a simplified form. Clearly, multiplication in $\mathbb{F}_2$ is equivalent to a logical *AND* while addition is equivalent to a logical *XOR*. However, elements multiplied by 0 can be omitted while elements multiplied by 1 are untouched. Thus, the multiplication can be eliminated. This is called an ***XOR*-only code**. To simplify the equations, this section uses C-syntax for logical operations, i. e. & stands for a logical *AND*, | stands for a logical *OR*, and ^ stands for a logical *XOR*.

Example 25: *Consider the simple example of $M \cdot D = C$:*

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix}$$

Writing this in terms of logical AND and XOR operations yields:

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} (1 \;\&\; D_1) \;\hat{}\; (0 \;\&\; D_2) \;\hat{}\; (0 \;\&\; D_3) \;\hat{}\; (1 \;\&\; D_4) \\ (0 \;\&\; D_1) \;\hat{}\; (0 \;\&\; D_2) \;\hat{}\; (1 \;\&\; D_3) \;\hat{}\; (1 \;\&\; D_4) \\ (0 \;\&\; D_1) \;\hat{}\; (1 \;\&\; D_2) \;\hat{}\; (0 \;\&\; D_3) \;\hat{}\; (1 \;\&\; D_4) \\ (1 \;\&\; D_1) \;\hat{}\; (1 \;\&\; D_2) \;\hat{}\; (0 \;\&\; D_3) \;\hat{}\; (0 \;\&\; D_4) \end{pmatrix} = \begin{pmatrix} D_1 \;\hat{}\; D_4 \\ D_3 \;\hat{}\; D_4 \\ D_2 \;\hat{}\; D_4 \\ D_1 \;\hat{}\; D_2 \end{pmatrix} \tag{16.2}$$

The binary matrix multiplication originally involving sixteen logical AND and twelve logical XOR operations can thus be evaluated by only four operations.

Under the assumption that for large encoding-matrices equally many zeroes and ones are present, half of the *XOR* operations can be eliminated (in addition to all *AND* operations). This reduces the calculation effort by a factor of four. Obviously, the same result holds for the decoding-matrix.

16.4.2 *Add*-only Codes

Proposition 19 showed that an arbitrary lift of an encoding-matrix with entries in $\mathbb{F}_2$ yields an encoding-matrix with entries in $\mathbb{Z}/2^b\mathbb{Z}$. Clearly, this lift can be chosen such that the encoding-matrix contains only the elements 1 and 0 (Remark 20 (iii)). In the same way as in Example 25, the multiplication can be eliminated in this case. Encoding is then performed only by addition while the calculation effort is reduced by a factor of four, as for the *XOR*-only code. This variant is called an ***Add*-only code**.

In contrast to *XOR*-only codes on the field $\mathbb{F}_2$, only the encoding-matrix can be restricted to the elements zero and one. Since the encoding-matrix uniquely defines the decoding-matrix, which has entries in $\mathbb{Z}/2^b\mathbb{Z}$, the full matrix multiplication must be performed for decoding.

16.5 Variants

16.5.1 Encoding by Matrix-Matrix Multiplication

In practice, the data that are stored on redundant storage systems greatly exceed the size of the data word vector D. The encoding is then performed block by block. The input data are split in blocks $D^{(s)} = (D_1^{(s)}, \cdots, D_l^{(s)})$ $(1 \leqslant s \leqslant S)$, which are encoded independently:

$$C^{(s)} = M \cdot D^{(s)}.$$

All code words $C_j^{(s)}$ for fixed j are stored on the same storage device.

Proposition 26: *Combining the data and code words in matrices*

$$C^T = \begin{pmatrix} C_1^{(1)} & \cdots & C_1^{(S)} \\ \vdots & \ddots & \vdots \\ C_k^{(1)} & \cdots & C_k^{(S)} \end{pmatrix} \qquad D^T = \begin{pmatrix} D_1^{(1)} & \cdots & D_1^{(S)} \\ \vdots & \ddots & \vdots \\ D_n^{(1)} & \cdots & D_n^{(S)} \end{pmatrix}$$

the encoding of the entire dataset can be performed by a matrix-matrix multiplication:

$$C^T = M \cdot D^T. \tag{16.3}$$

This can be rewritten to

$$C = D \cdot M^T. \tag{16.4}$$

Equation 16.4 is more practical than 16.3 since the input and output matrices are stored in row-major format (see Appendix C.2). The previous single-block case is the special case with a matrix consisting of a single column. Mathematically, nothing new has been obtained. However, from the computational side this is a great advantage. While matrix-vector multiplication is usually bandwidth-bound, matrix-matrix multiplication can achieve the computational peak performance. All the common techniques such as blocking (see Section 11.2.4.1) can be applied to speed up the encoding.

16.5.2 Strassen Matrix-Matrix Multiplication

According to the previous section, the encoding can be performed by matrix multiplication. Thus, for fast encoding a fast matrix multiplication is required. In contrast to Part III, the rules for the Linpack benchmark demanding an $\mathcal{O}(n^3)$ algorithm do not apply. Hence, the fast Strassen matrix-matrix multiplication algorithm [Str 69] can be used.[10] In the following, the algorithm is presented briefly.

Theorem 27 (Strassen)**:** *Consider square $n \times n$ matrices A and B. The matrix product $A \cdot B$ can be calculated with $\mathcal{O}(n^{\log_2(7)})$ operations.*

Proof: Write the matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

(The naive algorithm calculates $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$.) After calculating the intermediate matrices

$$\begin{aligned} I &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ II &= (A_{21} + A_{22}) \cdot B_{11} \\ III &= A_{11} \cdot (B_{12} - B_{22}) \\ IV &= A_{22} \cdot (B_{21} + B_{11}) \\ V &= (A_{11} + A_{12}) \cdot B_{22} \\ VI &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ VII &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{aligned}$$

the matrix C can be obtained (with seven instead of eight multiplication steps) by

$$\begin{aligned} C_{11} &= I + IV - V + VII \\ C_{21} &= II + IV \\ C_{12} &= III + V \\ C_{22} &= I + III - II + VI. \end{aligned}$$

Applying this scheme recursively until only scalars are multiplied, the claim follows by induction. (The addition steps in each recursion iteration do not contribute to the asymptotic complexity since matrix addition is only of complexity $\mathcal{O}(n^2)$.) □

The Strassen algorithm can still be applied for non-square matrices. The maximum number of Strassen iterations is limited by the lowest occurring matrix dimension. Each iteration reduces the complexity by $^7/_8$ compared to the naive approach.

[10] There are even faster algorithms for matrix multiplication like the Coppersmith-Winograd algorithm [Cop+ 87], whose time constants, however, make them infeasible.

16.5.3 Parallel Codes

Definition 28: *Let R be a ring, A an R-module, and $M\colon R^n \to R^k$ the encoding-matrix of an MDS-code (or a vector-MDS-code over a ring S). There is a canonical linear operation $M\colon A^n \to A^k$, which has all the properties of a vector-MDS-code over R (or S). A special case is $A = R^t$ for $t \in \mathbb{N}$. Such a code is called a* ***parallel (vector) MDS-code*** *of width t. The encoding-matrix $\hat{M}\colon (R^t)^n \to (R^t)^k$ is composed of blocks. The $(i,j)^{th}$ block is defined by $\hat{M}_{ij} = M_{ij} \cdot \mathbb{1}_{t\times t}$.*

Example 29: *The parallel code version of Example 25 reads:*

$$\hat{M} = \begin{pmatrix} \mathbb{1} & 0 & 0 & \mathbb{1} \\ 0 & 0 & \mathbb{1} & \mathbb{1} \\ 0 & \mathbb{1} & 0 & \mathbb{1} \\ \mathbb{1} & \mathbb{1} & 0 & 0 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} & 0 & 0 & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \\ 0 & 0 & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \\ 0 & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} & 0 & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} & \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} & 0 & 0 \end{pmatrix}$$

By defining a component-wise AND-operation (i. e. $0 \,\&\, D = 0$, $1 \,\&\, D = D$ for $D \in \mathbb{F}_2^t$), Equation 16.2 from Example 25 holds exactly.

16.6 Code Overview

All presented codes can encode via matrix multiplication $C^T = M \cdot D^T$, in long[11]:

$$\begin{pmatrix} \begin{pmatrix} (M_{11})_{11} & \cdots & (M_{11})_{1l} \\ \vdots & \ddots & \vdots \\ (M_{11})_{l1} & \cdots & (M_{11})_{ll} \end{pmatrix} & \cdots & \begin{pmatrix} (M_{1n})_{11} & \cdots & (M_{1n})_{1l} \\ \vdots & \ddots & \vdots \\ (M_{1n})_{l1} & \cdots & (M_{1n})_{ll} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ \begin{pmatrix} (M_{k1})_{11} & \cdots & (M_{k1})_{1l} \\ \vdots & \ddots & \vdots \\ (M_{k1})_{l1} & \cdots & (M_{k1})_{ll} \end{pmatrix} & \cdots & \begin{pmatrix} (M_{kn})_{11} & \cdots & (M_{kn})_{1l} \\ \vdots & \ddots & \vdots \\ (M_{kn})_{l1} & \cdots & (M_{kn})_{ll} \end{pmatrix} \end{pmatrix} \cdot \begin{pmatrix} \begin{pmatrix} (D_1)_1^{(1)} \\ \vdots \\ (D_1)_l^{(1)} \end{pmatrix} & \cdots & \begin{pmatrix} (D_1)_1^{(S)} \\ \vdots \\ (D_1)_l^{(S)} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ \begin{pmatrix} (D_n)_1^{(1)} \\ \vdots \\ (D_n)_l^{(1)} \end{pmatrix} & \cdots & \begin{pmatrix} (D_n)_1^{(S)} \\ \vdots \\ (D_n)_l^{(S)} \end{pmatrix} \end{pmatrix}^T$$

For a Reed-Solomon code (i), the elements $(M_{ij})_{kl}, (D_i)_k^{(s)}$ are in $\mathbb{F}_{2^\alpha}$, for codes on finite rings (ii) in R, for Cauchy-Reed-Solomon (iii) in $\mathbb{F}_2$, for vector codes on finite rings (iv) in $\mathbb{Z}/2^b\mathbb{Z}$ and for parallel Cauchy-Reed-Solomon codes of width t (v) in $(\mathbb{F}_2)^t$. Computers cannot compute (i) and (ii) natively but have to emulate the operations. The CRS-code (iii) is computed via logical *AND* and *XOR* operations on single bits. With SSE, single-bit operations can be vectorized 128-fold. However, the reduction during the matrix multiplication poses a problem for such a broad vectorization. Assume $t = b = 32$. Versions (iv) and (v) perform native integer operations and component-wise logical operations on 32-bit values. Hence, for these codes, similar to single precision floating point, the matrix multiplication can achieve the computational peak performance. Thus, they are suited best for implementation on a computer.

[11] For codes (i) and (ii) there are single scalar entries M_{ij} instead of the $l \times l$ submatrices, which, however, makes no difference for a computer.

16.7 Computational Complexity

For comparability, in this entire part an **AOp** is defined as a 32-bit **generalized arithmetical operation**. This can be single precision floating point addition / multiplication, 32-bit integer addition / multiplication or a 32-bit logical operation, i. e. *AND*, *OR* or *XOR*.

Assume fixed parameters n and k with a $k \times n$ encoding-matrix. Consider either the integral code on $\mathbb{Z}/2^{32}\mathbb{Z}$ or the parallel Cauchy-Reed-Solomon code with $t = 32$. This results in integral or binary encoding matrices of dimension $lk \times ln$. Assume a dataset of $S \cdot nl$ data words of 32 bits each. The encoding is performed by multiplying an $S \times ln$ matrix with an $ln \times lk$ matrix and requires $S \cdot lk \cdot (2ln - 1)$ operations in both cases. Introducing the input size in bytes: $I = 4 \cdot S \cdot nl$, this yields asymptotically $\frac{1}{2} Ilk$ AOps for the matrix multiplication.

The *XOR*-only variant of the parallel Cauchy-Reed-Solomon and the *Add*-only variant of the integral code do the matrix multiplication faster, namely in $\frac{r}{4} Ilk$ operations where r is the **matrix fill-ratio**.[12] In any case, x Strassen iterations reduce the complexity by a factor $(8/7)^x$. For both the Vandermonde and the Cauchy matrix, the dimension l must not be chosen smaller than $l \geqslant \log_2(n+k)$. Thus, all encoding schemes have asymptotical complexity $\mathcal{O}(I \cdot k \cdot \log_2(n+k))$. The constants, however, are very different. The *XOR*-only and *Add*-only variants are faster by a factor of $2/r \approx 4$ (which is shown later).

16.8 Lower Bound for l

Both the Vandermonde and the Cauchy matrix demand that $p^l \geqslant n + k$. Since a larger l leads to a higher computational complexity, the question arises whether a code can be created with a smaller l. A simple lower bound for l is given by the following

Lemma 30: *An (n, k)-MDS-code on a local ring R with residue field $\kappa = R/m$ and $k \geqslant 2$ is subject to the inequality*

$$\#\kappa > max(n, k) \qquad \textit{(for the above codes consider } \kappa = \mathbb{F}_{p^l},\ \#\kappa = p^l\textit{)}.$$

Proof: Without restriction, the code can be assumed to operate on a field by calculating everything modulo the maximal ideal. Assume the MDS-code $\widetilde{M}$ in reduced form. (Since M is locally regular all its entries are nonzero.)

$$\widetilde{M} = \begin{pmatrix} \mathbb{1} \\ M \end{pmatrix}$$

Multiplying columns or rows with nonzero scalars does not affect the local regularity of the encoding-matrix. By multiplying each column with a scalar, the elements of the $(n+1)^{\text{th}}$ row can be changed to one. Multiplying each row with scalars yields the form:

$$\widetilde{M} = \begin{pmatrix} 1 & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & 1 \\ 1 & \cdots & \cdots & 1 \\ 1 & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 1 & * & \cdots & * \end{pmatrix}$$

Since $\widetilde{M}$ is locally regular, all entries of the $n + 2^{\text{th}}$ row must be nonzero. In addition, they must be pairwise distinct for the following reason: assume $\widetilde{M}_{n+2,i} = \widetilde{M}_{n+2,j}$. Then the 2×2 submatrix

[12] The matrix fill-ratio is defined as quotient of the number of nonzero entries divided by the total number of entries.

of $\widetilde{M}$ consisting of columns i and j and of rows $n+1$ and $n+2$ is not regular. ↯
Likewise, the elements of the first and second column below row n must be nonzero and pairwise distinct. But then, the field must at least contain $n+1$ or $k+1$ elements, whichever is higher. □

One idea to overcome this limitation (while losing the MDS property) is presented in the following

Example 31: *Let $p = 2$, $\mathbb{F}_q \cong \mathbb{F}_{2^l} \cong \mathbb{F}_2[X]/f$, $R = \mathbb{Z}[X]/2^b,f$, and $K = \mathbb{Q}[X]/f$. Consider the Cauchy-like matrix $M = 2N$ defined by*

$$M = \begin{pmatrix} \frac{2}{a_1+2+a_1} & \cdots & \frac{2}{a_n+2+a_1} \\ \vdots & \ddots & \vdots \\ \frac{2}{a_1+2+a_k} & \cdots & \frac{2}{a_n+2+a_k} \end{pmatrix}$$

for a_i pairwise distinct modulo 2. (This is actually twice the Cauchy matrix N with $b_i = a_i + 2$.) Its entries lie in R and according to Proposition 24 it is locally invertible in K. Consider a square submatrix M' of M and the corresponding submatrix N' of N. With the adjoint matrix $M^{\#}$ of M the invert has the form $M'^{-1} = \frac{1}{2} \cdot \frac{1}{\det(N')} N'^{\#}$. According to the determinant formula for Cauchy matrices (Equation 16.1), every factor of 2 in a denominator of any entry of N' occurs also in the denominator of $\det(N')$. Therefore, the matrix $\frac{1}{\det(N')} N'^{\#} = X$ has entries in R. Now both M and X have entries in R with $M \cdot X = 2 \cdot \mathbb{1}$.

In summary, only $2^l \geqslant max(n,k)$ distinct elements are required instead of $2^l \geqslant n+k$ for the previous methods. When encoding and decoding a dataset

$$C = M \cdot D \qquad D_{restored} = X \cdot C = 2D,$$

the original data are not restored but multiplied by two, i. e. the binary representation is shifted left by one bit and the most significant bit is lost. To account for this, this bit must not be used to store data. Assume for instance $b = 64$ and $n = k = 32$. In this case, the storage requirement increases by 1/64. The dimension l can be reduced from six to five while the encoding has to operate on 64 bits to encode only 63 bits of data. The computational complexity (considering single bit operations) decreases by factor

$$\frac{6 \cdot 63}{5 \cdot 64} = \mathbf{1.18125}.$$

Naturally, for larger l or smaller b the improvement is less.

Finally, due to the small benefit, the increased storage requirement, and the complex data handling[13] this approach is not used.

16.9 Partial Update-Codes (Differential Codes)

Assume that the redundancy information (C_j), $(1 \leqslant j \leqslant k)$ has already been generated for the data words (D_i), $(1 \leqslant i \leqslant n)$. Assume that one data word D_{i_0} is changed. The question is whether the redundancy information has to be completely recalculated from scratch or whether there is a simpler possibility for an update. A scenario where this applies to is a redundant network storage, where only a part of the data is updated. On one storage at most one of the redundancy words (C_j) is available. The following lemma shows how this redundancy word can be updated. (This is called **differential code** or **update-code**.)

Lemma 32: *Let M be the encoding-matrix of an MDS-code and $C = M \cdot D$ the redundancy information for the data D. Let D' be a different set of data words such that $D_i = D'_i$ for all $i \neq i_0$. In this case, $C' = M \cdot D'$ can be calculated by*

$$C'_j = C_j + M_{ji_0} \cdot (D'_{i_0} - D_{i_0}).$$

[13] The input data in Example 31 still consist of 64 bits. Thus, the most significant bit of each quad-word must be treated independently.

Chapter 17

Implementation

While the last chapter handled the theoretical part of failure erasure coding, this chapter introduces implementations of the algorithms. As codes (iv) and (v) from Section 16.6 are best suited for computers, these are implemented. Both variants encode via matrix multiplication. Asymptotically, all algorithms have the same complexity, namely the complexity of matrix multiplication. Still, optimized implementations can vary strongly in performance. This chapter starts with simple pure matrix multiplication based codes and then proceeds to more advanced versions.

17.1 Metrics

For the following analysis, only the case $k = n$ is considered. Except for very flat matrices, the performance depends rather on the total matrix size but not on its shape. The dependency on k will be analyzed independently in Section 17.6. Assume that the encoding process takes t seconds for encoding I bytes. In order to analyze the performance of the codes, multiple metrics are used (compare to the computational complexity derived in Section 16.7):

1. **Bandwidth**: The code bandwidth B is defined as the maximum incoming data rate which can be processed. It is defined by $B = {}^{I}/_{t}$. In the case $n = k$, the total required memory bandwidth is twice the code bandwidth as the data must be both fetched from and stored to memory. The bandwidth is the metric which is finally relevant for the user.
2. **Update Bandwidth**: The simple bandwidth definition above does not reflect the increased computational complexity for larger k. The metric should reflect that creating k code words has the k-fold complexity of creating one code word. This is described by the update bandwidth[1] B_u defined by $B_u = k \cdot B = \frac{I \cdot k}{t}$. This metric is a transparent measure of the performance of the implementation which is finally available to the user.
3. **Matrix-Multiply (MM) $^{\text{AOp}}/_{\text{s}}$**: The parameter l is irrelevant for the user as a higher l brings no benefit. Instead, it is entailed by internals of the implementation with larger l resulting in higher complexity. Hence, the update bandwidth is insufficient to compare the performance in terms of operations per second for different matrix sizes. For this purpose, the raw matrix multiplication performance $G = \frac{I \cdot l \cdot k}{2t}$ is used. In short, this measures the performance of the matrix multiplication assuming the naive algorithm is employed.
4. **Throughput**: The *XOR*-only and *Add*-only codes can do the matrix multiplication with fewer operations than the naive version. So the MM $^{\text{AOp}}/_{\text{s}}$ metric G can exceed the theoretical peak performance of the processor. Thus, the raw assembler instruction throughput[2], as

[1] The background of the name is that the encoding can be interpreted as a series of partial updates (see Section 16.9). The update bandwidth is the bandwidth of these updates.

[2] For the matrix multiplication based codes, which are derived from GotoBLAS, this throughput is set to the MM $^{\text{AOp}}/_{\text{s}}$ performance achieved. In fact, this is not absolutely correct as slightly more instructions are needed for loads, stores, and shifts. However, the effort to calculate the real assembler instruction count for GotoBLAS runs is impracticable.

a fourth metric, analyzes how much potential is left. For better taking into account vector-processor architectures, the metric is scaled by the vector width w (in terms of 32-bit values). For instance, the throughput of an SSE processor is the number of instructions actually executed per second times four for the vector width. This takes into account all instructions (not only SSE) since they share the same pipelines. With a the total number of assembler instructions executed, the throughput is thus defined as $T = \frac{a \cdot w}{t}$.

17.2 Matrix Multiplication based Codes

Section 16.3.1.3 introduced a matrix-vector multiplication based code, which is generalized to matrix-matrix multiplication in Section 16.5.1. In the following, matrix multiplication codes refer to a full naive matrix multiplication without the shortcuts used for *XOR*-only and *Add*-only codes. Matrix multiplication in the HPC sector usually employs SGEMM, DGEMM, or their complex-valued versions. These variants operate on floating point data, while for the vector MDS-code over $\mathbb{Z}/2^b\mathbb{Z}$ (iv) an integer and for the parallel CRS code (v) a binary valued implementation is required. All following matrix multiplication codes are implemented using 128-bit SSE instructions. The AVX instruction set could theoretically double the encoding speed in the future. In the first phase, only the matrices from Example 36 in Appendix F.1 have been implemented. As described before (Proposition 23), this poses some restrictions on the dimension l.

17.2.1 IGEMM

In the following, the term IGEMM (in allusion to SGEMM) denotes integer-valued matrix-matrix multiplication. It is desired to utilize the common expertise for floating-point matrix multiplication which is available from BLAS libraries. In Part III, the GotoBLAS library demonstrated outstanding SGEMM and DGEMM performances on recent CPUs. Both SGEMM and IGEMM process data-types of the same size. Thus, an IGEMM implementation can be obtained from the GotoBLAS SGEMM by exchanging the floating point additions and multiplications with their integer counterparts. In practice, this is not so simple since some SSE floating point instructions do not have an integer equivalent. This enforces some minor changes in the code. Shift and move instructions of the GotoBLAS SGEMM are unchanged.

To optimize the IGEMM for the special case of encoding, the operation is simplified from the standard SGEMM formula $C' = \alpha AB + \beta C$ to $C' = AB$. This saves the multiplication by the scalar factors and especially the loading of the former C-matrix. As the current version of the 256-bit AVX vector extension does not support integer computation, only SSE is used.

17.2.2 BGEMM

In contrast to IGEMM, a binary matrix multiplication (BGEMM) for the standard CRS code cannot be obtained directly from the GotoBLAS SGEMM. CRS operates on 1-bit values. This has no counterpart in GotoBLAS. In general, the implementation of a vectorized version of the single-bit BGEMM is a straight-forward task. The BGEMM involves solely binary *XOR* and *AND* operations. The SSE instructions for *XOR* and *AND* can thus be used and the 128-bit SSE registers are used as vector registers with 128 components. Unfortunately, there is one drawback: the vectorized binary matrix multiplication requires the reduction of a 128-bit vector and bit shifting of the results to the correct position for the output. This makes the 1-bit BGEMM inferior to the IGEMM, where the reduction and the scatters can be handled better.

This problem and the fact that the BGEMM would have to be written from scratch led to a different approach. Instead of the plain CRS code, a parallel CRS code with $t = 32$ is used. The

encoding-matrix is formed such that each bit is replicated 32 times, e. g. Example 29 looks like:[3]

$$\begin{pmatrix} 1111\cdots1111 & 0000\cdots0000 & 0000\cdots0000 & 1111\cdots1111 \\ 0000\cdots0000 & 0000\cdots0000 & 1111\cdots1111 & 1111\cdots1111 \\ 0000\cdots0000 & 1111\cdots1111 & 0000\cdots0000 & 1111\cdots1111 \\ 1111\cdots1111 & 1111\cdots1111 & 0000\cdots0000 & 0000\cdots0000 \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix}$$

In this representation, the 1-bit logical operations of plain CRS are exchanged by 32-bit logical operations. This approach allows for using a modified GotoBLAS SGEMM again. Single precision floating point addition and multiplication are replaced by 32-bit logical *AND* and *XOR* operations respectively. As for the IGEMM, neither shift nor load / store instructions have to be modified.

Fig. 17.1 shows IGEMM and BGEMM performance on Westmere and Sandy Bridge. A matrix size of at least $n = k = 48$ is required until performance saturates. On the Westmere, IGEMM shows quite exactly half the performance of BGEMM while on Sandy Bridge both are equally fast. Westmere IGEMM performance is poor because the Westmere can process only one and not two integer SSE operations per cycle (in contrast to floating point operations or logical operations; see Table 17.3). The BGEMM performance equals quite exactly the SGEMM performance on both architectures. Theoretically, the BGEMM could be even faster since both architectures can execute three logical SSE operations per cycle. However, GotoBLAS is designed and optimized for two operations per cycle (which is the floating point limit). The adapted GotoBLAS does not exceed this. An improved BGEMM version could theoretically speed up the encoding by 50 %. It was not realized because a different approach below performs better, as shown in the following.

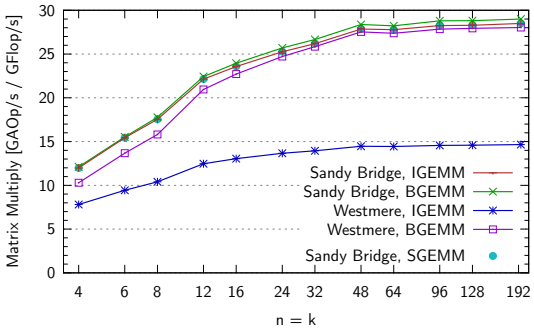


Figure 17.1: IGEMM / BGEMM Performance [II,XII]

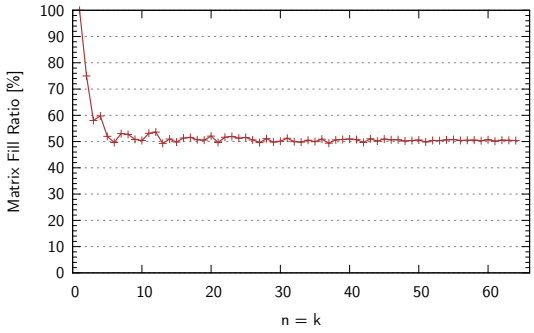


Figure 17.2: Matrix Fill-Ratios of Vandermonde Matrix employed by QEnc [XII]

17.3 Automorphic Assembly Codes

Section 16.7 deduced that the encoding time complexity constant of *XOR*-only (or *Add*-only) codes is smaller than the one for naive matrix multiplication based codes. The constant depends on the matrix fill-ratio r. Fig. 17.2 shows fill-ratios for Vandermonde encoding matrices at different $n = k$. The fill-ratio is high for very small matrices but goes asymptotically to 0.5 very fast. Thus, in the following $r = 0.5$ is assumed.

An optimal encoding could reach three logical or move SSE-instructions per cycle (Table 17.3), a 50 % improvement over the BGEMM above. Since the CPU can still execute a scalar operation in addition, address calculation can be hidden and the peak performance is achievable. On top of that, according to Section 16.7 assuming $r = 0.5$, an optimally implemented *XOR*-only code (or *Add*-only code) can outperform the matrix multiplication based version by a factor of:

$$150\,\% \cdot \frac{(\text{Complexity}_{XOR\text{-only}})^{-1}}{(\text{Complexity}_{\text{BGEMM}})^{-1}} = 1.5 \cdot \frac{(\frac{r}{4}Ilk)^{-1}}{(\frac{1}{2}Ilk)^{-1}} = \mathbf{6}.$$

[3] In the BGEMM, the multiplication is a component-wise *AND* while the addition is a component-wise*XOR*.

Recall that the encoding is performed by $C = D \cdot M^T$. Since M contains exclusively zeros and ones, the C-matrix entries are obtained by taking the *XOR* (or *Add*) of certain entries of D. The entries to use are determined by M, namely where the corresponding entry in M is nonzero.

Instruction Type	Westmere	Sandy Bridge
Floating Point (SSE)	2 / cycle	2 / cycle
Integer (SSE)	1 / cycle	2 / cycle
Logical / Move (SSE)	3 / cycle	3 / cycle
Floating Point (AVX)	-	2 / cycle
Integer (AVX)	-	-
Logical (AVX)	-	1 / cycle
Move (AVX)	-	2 / cycle

Table 17.3: SSE Instruction Throughput on Intel Architectures[4]

The question is how to store which entries of the matrix are nonzero. With a fill-ratio of $r = 0.5$, a binary matrix storage maximizes the entropy [Sha 48] and is thus the densest form. Sadly, decoding the data requires either a logical *AND* with the matrix entry or a conditional jump/ assignment. All of this makes the advantage of the *XOR*-only code null and void. Alternatively, a list can be maintained for each entry of C specifying which entries of D to use. This, however, requires many loads from the list itself. In addition, it is difficult to reuse entries of D for the calculation of different elements of C, which is of eminent importance for reducing the required memory bandwidth.

The only possible way to not waste instructions for decoding the M-matrix – in whatever format – is to encapsulate the M-matrix into the source code itself. The matrix multiplication is written down explicitly in the sources. Listings 17.4 and 17.5 demonstrate how this is meant. In practice, 128-bit SSE datatypes are used, i. e. parallel codes with $t = 128$ for the *XOR* and $t = 4$ for the *Add* variant, eliminating all shift instructions that are used in GotoBLAS for 32-bit data-types.

```
int M[3][3] = {
  {0, 1, 1},
  {1, 0, 0},
  {1, 1, 0}
};

void enc(int D[3][3], int C[3][3])
{
  for (int i = 0;i < 3;i++)
  {
    for (int j = 0;j < 3;j++)
    {
      C[i][j] = 0;
      for (int k = 0;k < 3;k++)
      {
        C[i][j] ^= D[i][k] & M[j][k];
      }
    }
  }
}
```

Listing 17.4: Standard Matrix Multiplication Code

```
void enc(int D[3][3], int C[3][3])
{
  C[0][0] = D[0][1] ^ D[0][2];
  C[0][1] = D[0][0];
  C[0][2] = D[0][0] ^ D[0][1];
  C[1][0] = D[1][1] ^ D[1][2];
  C[1][1] = D[1][0];
  C[1][2] = D[1][0] ^ D[1][1];
  C[2][0] = D[2][1] ^ D[2][2];
  C[2][1] = D[2][0];
  C[2][2] = D[2][0] ^ D[2][1];
}
```

Listing 17.5: Encoding-Matrix encapsulated in Instruction Stream

[4] Under certain conditions [Int 11, 2.1.4], the CPU can execute one scalar instruction in parallel, too.

Naturally, this blows up the code size for larger matrices tremendously. An even bigger problem is that the matrix is hardcoded in the source code. It is definitely impossible to include each and every encoding-matrix for all n and k parameters, not to mention the decoding matrices. The solution to the latter problem is an **automorphic code**. After the encoding-matrix is generated, binary assembler instructions for this matrix are created and written to memory which is assigned the executable flag. This assembler-code can then be executed as if it were contained in the regular source code. Unfortunately, all the compiler optimizations are not available automatically but they must be implemented by hand. Generating and compiling C code does not work well for two reasons (as seen in Section 18.2): the compiler can barely handle the huge source codes for large matrices and evaluation trees get so complex that the compiler can hardly apply its optimizations.

17.3.1 *XOR*-only Encoding

An *XOR*-only code based on the above principle has been implemented. It is available as open source library called **QEnc** (see Appendix I for the sources). The first version bases on 128-bit SSE instructions. A new frontend for the YASM [Yas] assembler has been written, which provides the possibility to compile the generated source code directly within QEnc. Sufficient executable memory is allocated using *VirtualAlloc* (on Windows) or *mmap* (for Linux). Having generated the assembler code once, the code can be executed to encode data multiple times. Changing the n or k parameters requires a new matrix and a new assembler code. So does a decoding-matrix as well.

Fortunately, a matrix multiplication in assembler is not so complicated. Thus, the lack of compiler optimizations can be handled. The most important optimizations are related to cache usage where a compiler cannot help that much anyway. The following sections present cache and compiler optimizations applied to the assembler code. Some examples are shown later. Trivial multi-threading over the input data is possible. In order to simplify the evaluation, only one CPU core is used in the following. Development is performed on a Sandy Bridge system [XII]. Sometimes, results on a Westmere [II] are presented for comparison. Since the code is single-threaded, both processors can utilize their turbo boost functionality, which can overclock a single CPU core as long as it remains within the **T**hermal **D**esign **P**ower (TDP). With turbo boost, the Westmere runs at 3.8 GHz while the Sandy Bridge is clocked with 3.7 GHz. This makes the results quite comparable.

17.3.2 Blocking & Cache Usage

The fact that the assembler code performs a matrix multiplication ($C = D \cdot M^T$) suggests itself to use the blocking technique (Section 11.2.4.1). In fact, a whole hierarchy of blocking levels is implemented in the QEnc automorphic code. In the following, these levels are introduced step by step. (See Appendix F.3 for a C++ example illustrating all blocking levels.)

17.3.2.1 Register Blocking

The lowest possible blocking level is on a register basis. Assume r elements of C are calculated in parallel, which requires r registers to store and accumulate the intermediate results. These registers are called **accumulation registers**. Entries of D are fetched one after another. Each of them is used to update all r registers corresponding to r entries of C. In average, 50 % (the matrix fill-ratio) are affected, i. e. the data can be reused $r/2$ times. The SSE instruction set offers sixteen 128-bit registers (*xmm0* – *xmm15*). (AVX offers sixteen *ymm* 256-bit registers.) Some of them are required to store the elements of D (**scratch registers**). Due to the read-after-write latency, it is inefficient to use a value directly after it has been stored to a register. Thus, at least two registers are used to store the entries of D in a round robin fashion. The next entry is fetched before the calculation for the current entry starts. Listing 17.6 demonstrates the register blocking for the example in Listing 17.5 with two scratch registers *xmm14* and *xmm15*. The values *C[0][0]* to *C[0][2]* are stored in registers *xmm0* to *xmm2* (accumulation registers), *rbx* points to *D[1][0]*.

```
movdqa xmm14, [rbx]          //Fetch D[1][0]
...                          //Previous Iteration
movdqa xmm15, [rbx + 16]     //Fetch D[1][1]
movdqa xmm1,  xmm14          //Process D[1][0] (movdqa for the initialization)
movdqa xmm2,  xmm14          //...
movdqa xmm14, [rbx + 32]     //Fetch D[1][2]
movdqa xmm0,  xmm15          //Process D[1][1]
pxor   xmm2,  xmm15          //...              (pxor for all further operations)
...                          //Fetch value for next Iteration
pxor   xmm0,  xmm14          //Process D[1][2]
```

Listing 17.6: QEnc Assembler Code, Register Blocking

The optimal number of scratch registers depends on the latency for the load operations. From Fig. 17.7 it is obvious to use as many registers for blocking and as few for scratch as possible.

17.3.2.2 L1 Blocking

An additional blocking is applied to the loop calculating a C-matrix entry $C_{ji} = \sum_{q=1}^{n} M_{iq} D_{jq}$. The computation is split in partial loops starting with $C_{ji}^{(0)} = \sum_{q=1}^{c} M_{iq} D_{jq}$. This loop is calculated for the first set of registers $0 < i \leqslant r$. The entries D_{j1} to D_{jc} reside in the CPU L1 cache afterward. Next the partial loops $C_{ji}^{(0)} = \sum_{q=1}^{c} M_{iq} D_{jq}$ are calculated for $r < i \leqslant 2r$ and so forth. The cached entries of D are reused. Having finished the first partial loop for all i, the next part of C_{ji} is computed ($C_{ji}^{(1)} = C_{ji}^{(0)} + \sum_{q=c+1}^{2c} M_{iq} D_{jq}$). To fully understand the effect of the L1 blocking, a comprehension of the full blocking hierarchy is required. Thus, at first consider Fig. 17.8 in the range $1 \leqslant n \leqslant 16$. The performance increases with a larger blocking and saturates at $c = 96$. The rightmost part of the diagram will be analyzed in Section 17.3.2.5 later.

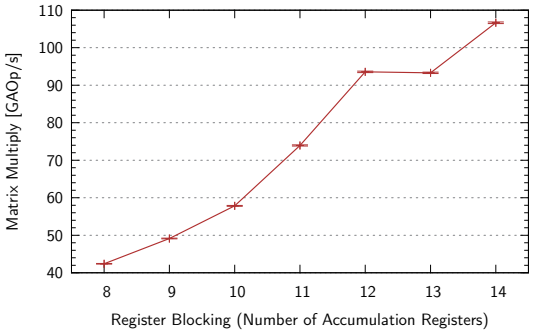


Figure 17.7: XOR128 Register Blocking Performance [XII]

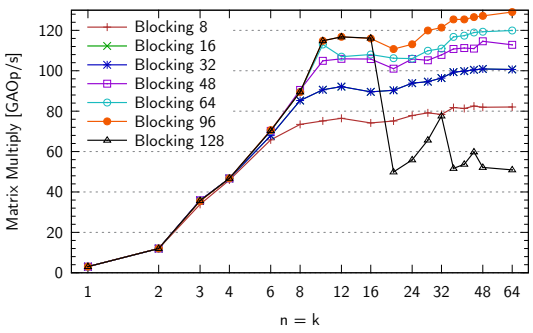


Figure 17.8: XOR128 L1 Data Blocking Performance [XII]

17.3.2.3 L1 Instruction Cache Blocking

The size of the generated binary code (after assembly) scales linearly with the size of the matrix (i. e. with $nl \cdot kl$). Besides the L1 data cache, the processor employs a dedicated L1 instruction cache sized 32 KB. Fig. 17.9 correlates the performance with the size of the instruction stream. As soon as the code size exceeds the L1 cache size of 32 KB, the performance drops tremendously. To cope with this, an L1 instruction cache blocking is applied. The instruction stream is split in blocks of less than 32 KB. Since the entries of M do not depend on j, the following is possible: instruction stream blocks of less than 32 KB are first iterated over j in the above loop ($C_{ji} = \sum_{q=1}^{n} M_{iq} D_{jq}$) reusing the code from the $L1$ instruction cache. Having finished this j-loop, the next 32 KB part of the instruction stream is executed.

17.3.2.4 L2 Blocking

The L1 instruction blocking does not completely solve the problem. The reason is that after the loop over j, the higher L2 and L3 CPU caches contain the entries of M for a high j index whereas the next instruction stream will require low j values again. Thus, an L2 data blocking is applied such that the loop over j is only executed so far that the relevant source data for the next instruction stream block are still in the cache. It shall be noted that the L2 blocking is a second level blocking, not necessarily related to the L2-cache. It can span over the L3 cache as well. Fig. 17.10 shows the performance with instruction cache blocking without and with different L2 blocking levels. It can be concluded that almost any L2 blocking size is good while 64 comes out best – but with only little advance. (See Appendix F.3 for a C++ example of all blocking levels.)

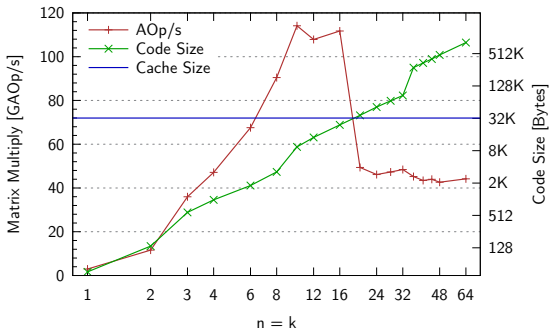


Figure 17.9: XOR128 Performance depending on Code Size [XII]

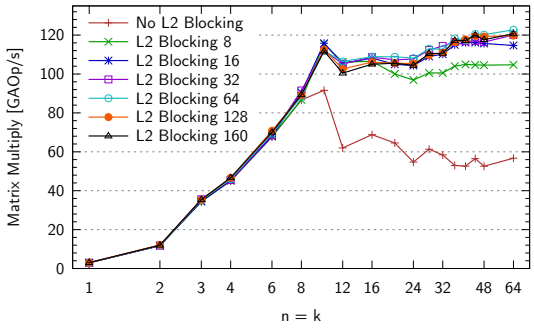


Figure 17.10: Performance with Instruction Cache Blocking and L2 Data Cache Blocking[5] [XII]

17.3.2.5 L1 Blocking, Second View

Now the entire L1 blocking behavior becomes clear. Fig. 17.8 already includes L1 instruction blocking and L2 data blocking. Both only set in for $n > 16$. With larger $L1$ blocking constant c, fewer iterations over i can be executed before the instruction stream blocks hit the L1 instruction cache size.[6] In summary: the larger c the more data are cached in L1, the smaller c the more often data from L1 are reused. An L1 blocking size of 96 turns out to be the best trade off.

Using the four blocking levels yet introduced (registers, L1 data, L1 instruction, L2 data), performance remains high up to medium sized matrices. (Large matrices are treated in Section 17.3.6 later.) The matrix-multiply performance reaches more than 120 $^{\mathrm{GAOp}}/_{\mathrm{s}}$, which is about four times the performance achieved by the BGEMM implementation visualized in Fig. 17.1.

17.3.3 Code Optimizations

17.3.3.1 Prefetching

Since QEnc performs a matrix multiplication, the memory access pattern is known in advance. It can thus benefit from prefetching. The *prefetcht0* instruction is used exclusively, which prefetches a cache line of 64 bytes to the L1 cache. With L1 instruction blocking, QEnc stores and reloads results of the partial calculations in buffers in memory. The reload can benefit from prefetching the buffer as well. It is also tested whether it is better to prefetch data one or multiple iterations prior to their usage. Fig. 17.11 demonstrates the results.

[5] All presented curves include L1 instruction blocking. L2 blocking size is measured in terms of iterations of one instruction stream block.

[6] When the L1 instruction cache blocking is hit, the following loop over j evicts the L1 data cache.

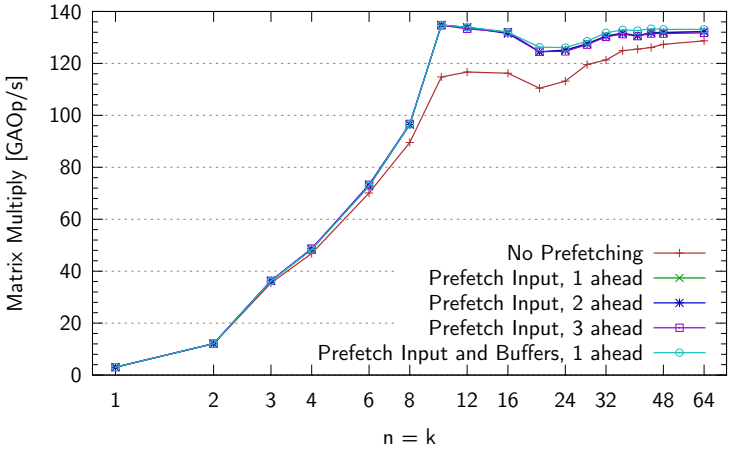


Figure 17.11: Prefetching XOR128 Input Data and Buffers [XII]

It turns out that the exact delay between prefetch and read does not play a great role, the speedup is always significant. Gathering enough statistics shows that a delay of one iteration is the fastest option. Prefetching the buffer for L1 instruction blocking slightly improves the performance.

17.3.3.2 Ternary Instructions

SSE instructions support only two operands enforcing one of the source registers to work as destination register, too. In contrast to that, AVX supports three-operand instructions, also called **ternary instructions**. These ternary instructions can also operate on 128-bit SSE registers. For now, only these 128-bit AVX instructions are considered. Full AVX usage is discussed in Section 17.3.11.2 later. Without macro-op fusion, the Sandy Bridge instruction decoder can decode up to sixteen bytes containing four instructions per cycle [Int 11]. The *pxor* and *movdqa* SSE instructions employed by QEnc have a four-byte opcode if both register indices are **low**, i. e. below eight ($xmm0 - xmm7$). Otherwise the encoding is five bytes long. For optimal QEnc performance, the CPU should decode and execute three SSE / AVX instructions per cycle and one scalar instruction for address calculation if necessary. Three five-byte instructions leave only one byte in the instruction decoder window left, not enough for the scalar instruction. Hence, address calculation can only be hidden behind the calculation if instructions with small encoding are used. In addition, smaller encodings improve the cache utilization.

In contrast to the SSE instruction *pxor*, the ternary AVX counterpart *vpxor* has a five-byte encoding only if its third operand is a **high** SSE register ($xmm8 - xmm15$). Thus, in many cases it has a smaller encoding than *pxor*. One approach to improving the performance is to replace all *pxor* by *vpxor* instructions.

In addition to *movdqa* and *pxor* which are meant for integers, there are the floating point instructions *movaps* and *xorps* which do basically the same. The floating point variants have a smaller encoding of three to four bytes. Unfortunately, they do not reach the throughput that can be achieved with the integer instructions. Most probably this is related to domain crossing of registers between the floating point and the integer domain [Int 11], which can cause a latency.

17.3.3.3 Register Selection

According to the last section, the opcode size depends on the registers used. Registers with low index usually lead to a smaller encoding. To account for this, QEnc uses low numbered registers as scratch registers because they are used more frequently than accumulation registers. Since the

third operand of a ternary instruction is always a scratch register, this enforces permanent four byte encoding of all instructions with 128-bit AVX operations.

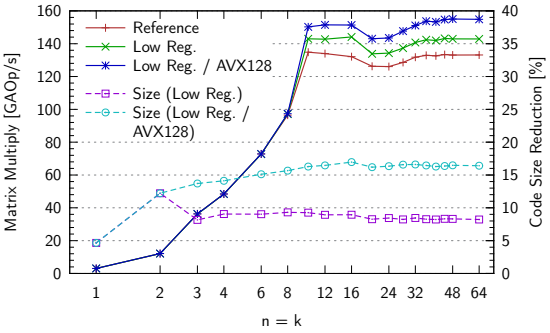


Figure 17.12: XOR128 Performance in Relation to Code Size [XII]

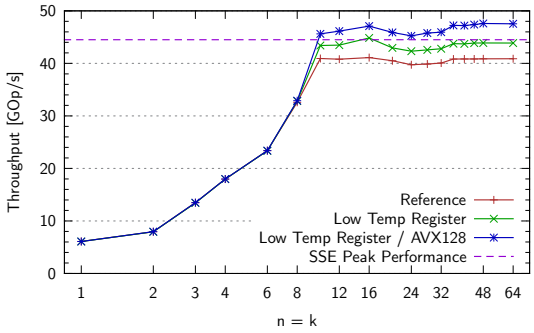


Figure 17.13: Instruction Throughput using all Optimizations [XII]

Fig. 17.12 demonstrates how the performance increas1es with the above optimizations. Besides, it reveals that the total size of the instruction stream can be reduced by up to 16 %. The matrix-multiply performance reaches $G = 155\,\mathrm{GAOp/s}$ which is about five times the BGEMM performance. Clearly, the processor is getting close to its peak performance. The peak throughput for the Sandy Bridge with turbo boost clocks at 3.7 GHz and at maximum three SSE instructions per cycle calculates to $3 \cdot 4 \cdot 3.7\,\mathrm{GHz} = 44.4\,\mathrm{GOp/s}$. Fig. 17.13 reveals that the processor actually achieves its theoretical peak performance and in some cases even slightly exceeds it, with a highest throughput of $T = 47.540\,\mathrm{GOp/s}$ (3.21 instructions per cycle). The latter can have three reasons:

- The processor can process three SSE instructions but in some cases can process a fourth scalar instruction in parallel, e. g. for address calculation.[7]
- The processor supports macro-op fusion and micro-op (μ-op) fusion, mainly for scalar and control flow instructions, which can raise the throughput.[7]
- Mainboard vendors normally set the reference clock rather slightly above the specification than below which usually raises the CPU clock rate in the range of 0.5 % to 1 %.

The number of assembler instructions per actual *XOR* operation for the encoding calculates to $\frac{T}{4G} = \mathbf{1.23}$. This shows that, in fact, the assembler code contains only little overhead.

17.3.4 Reducing Computational Complexity

The previous section shows that QEnc hits the theoretical peak performance of the processor with only little overhead left in the code. This does not leave much room for improvements on the assembler side. Instead, this section introduces ways to speed up the matrix multiplication itself.

The computational complexity scales linearly with the matrix fill-ratio r. The ratio was shown to go asymptotically to 0.5 (Fig. 17.2). J. S. Plank [Pla 05] introduces ways to create Cauchy matrices with a lower matrix fill-ratio – called **Good Cauchy Matrices (GC)** – by choosing good a_i and b_j values. It is even shown that the minimal possible dimension l is not always the best choice since a higher l leads to larger matrices but with possibly lower fill-ratio.

The QEnc implementation uses the Vandermonde construction. Thus, the Cauchy matrix optimizations cannot be implemented out of the box. Therefore, a different method was invented, which applies to all types of matrices, not only to Cauchy type matrices.

[7] Alternatively, either the peak throughput could be calculated with four instructions per cycle not three (a) or the throughput metric could count only SSE instructions (b). However, (a) overestimates the peak throughput of real SSE instructions while (b) ignores the shared pipeline of SSE and scalar instructions, which are only executed in parallel in certain cases. Micro- and macro-op fusion happen at runtime and cannot be considered anyway.

In retrospect, the new method presented below proves to be superior to the GC matrices for the purpose of QEnc. The GC matrices show the best performance for low n and k. However, it will be demonstrated that QEnc is entirely memory-bound up to $n = k = 12$ in Section 17.3.10 and thus cannot benefit from a lower fill ratio anyway. This makes the GC matrices less attractive. For medium n and k, e. g. $12 \leqslant n \leqslant 64$ and $k \geqslant \frac{n}{2}$, the average benefit of GC matrices is less than 13 %, which is significantly lower than the results of the new method below.

Still, the GC matrices constitute an opportunity for an additional reduction of the computational complexity. In combination with the new technique, they might increase the performance further and may be a valuable improvement for the future.

17.3.4.1 Local Matrix Optimizations

During matrix multiplication, intermediate results can be reused to save instructions. This optimization is best explained by means of an example. (For the sake of simplicity, the data matrix consists of a single row.) Let $C_i^j = \sum_{k=1}^{j} D_k \cdot M_{i,k}$ be defined as the intermediate result of C_i with the partial loop only going till j. Consider:

$$C = D \cdot M^T = \begin{pmatrix} D_1 & D_2 & D_3 & D_4 & D_5 & D_6 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}^T$$

Copying Rows The first and the second row of M resemble each other except for the last two columns. Thus, C_2 can be calculated as $C_1^4 \hat{} D_6$ instead of $D_1 \hat{} D_3 \hat{} D_6$. The intermediate result of the first row after four columns is copied to the second row saving one operation.

Similar Rows Row three equals row one except for the first column. Since the first columns differ, the above copying scheme cannot be used. Instead, row three can be replaced by the column-wise *XOR* of both rows: (1, 0, 0, 0, 0, 0). After the calculation, the result with the (replaced) row three is taken *XOR* with the result for row one. This means $C_3 = C_3' \hat{} C_1$ $(= C_3 \hat{} C_1 \hat{} C_1 = C_3)$ with C_3' the temporary result for the replaced row $(C_3' = D_1)$. This saves one instruction compared to the original naive $C_3 = D_3 \hat{} D_5 \hat{} D_6$.

Complex Row Similarity The row similarity optimization could be applied for rows three and four as well. However, in this case a combination of the row copy and row similarities is better: Not necessarily the full result C_3 has to be used, but it is better to use the intermediate result C_3^5 as this does not yet incorporate D_6, which is not needed for C_4 after all. Therefore, only the first five columns of the fourth row are replaced by the *XOR* of row three and four leading to ((1, 0, 0, 0, 0), 0). The intermediate result can be calculated as $C_4^5 = C_4^{5\prime} \hat{} C_3^5$ (with $C_4^{5\prime} = D_1$).

Multi Row Similarity Finally, if one row does not resemble any other row, it might still resemble the *XOR* of two rows. For instance, the last row in the example can be calculated as $C_6 = C_4 \hat{} C_5$. This can also be exploited for intermediate results as in the paragraph above.

Since all intermediate results must be available, the improvements can only be applied to entries processed at the same time, namely $r \times c$ submatrix blocks (register blocking × L1 blocking). All blocks are optimized independently – hence the name local optimization. Each combination of submatrix-rows and the above optimizations is checked and the one saving the most instructions is applied. Afterward, the search is repeated until no further optimization is possible. The check is performed using exhaustive search. The register blocking factor of $c = 14$ limits the number of

rows checked at a time making the exhaustive search feasible. Fig. 17.14 shows the reduction of the *XOR*-operation count (in the following denoted fill-ratio reduction). With increasing n the possible reduction factor first rises rapidly to 45 % but then falls again and remains at about 30 %. The reason is that the exploited similarities arise by coincidence. Unfortunately, the probability for similar rows decreases with the number of columns, i. e. with the matrix size. Since at maximum an $r \times c$ block is considered at a time, larger matrix sizes do not affect the row similarity probability anymore. Besides, fill-ratios of small matrices are above 50 % leading to more similarities.

As a remark: the row similarity optimization is only available at runtime. It is not possible to do the row operations on the encoding-matrix itself because changing the encoding-matrix might destroy its MDS-code property. Instead, the local optimizations maintain exactly the result of the encoding, however, with fewer operations.

17.3.4.2 Global Matrix Optimizations

The above optimizations are applied to local parts of the matrix only. The question is whether the matrix fill-ratio can be globally optimized by elementary operations on the matrix' rows and columns. In general, such operations would destroy the MDS property of the matrix. Still, some special cases are allowed. For an MDS-code, the matrix must be locally regular. After applying elementary row operations, this property is in general lost. After adding row i to row j and removing row i afterward, the matrix might not have full rank. In vector-MDS-codes, instead of single rows or columns, always a block of l rows or columns is removed. Thus, elementary operations within such a block do not destroy the MDS property. Since the above row similarity optimizations operate in blocks of 14 rows (for the register blocking of $c = 14$), the global matrix optimization has another degree of freedom as it operates on blocks of l rows (at least as long as l and c are coprime). Clearly, global elementary column operations (operating on columns not rows) introduce additional degrees of freedom for the optimization, regardless of local optimizations. Of course, the approaches are similar and probably not independent.

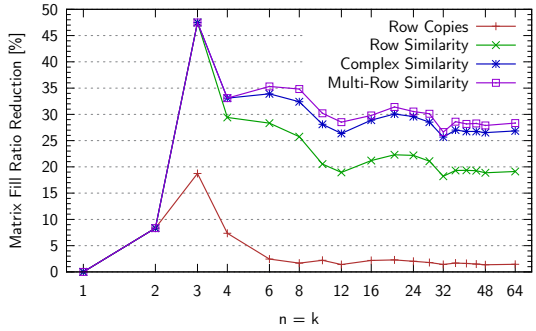


Figure 17.14: Matrix Fill-Ratio Reduction by Local Optimizations[8] [XII]

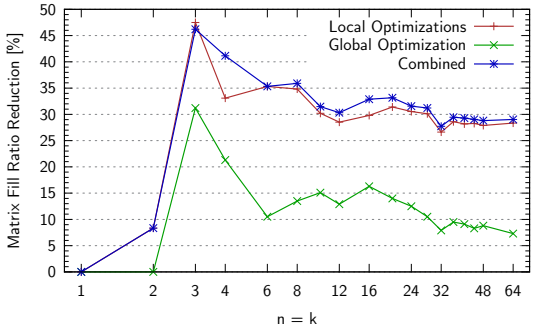


Figure 17.15: Total Matrix Fill-Ratio Reduction[8] [XII]

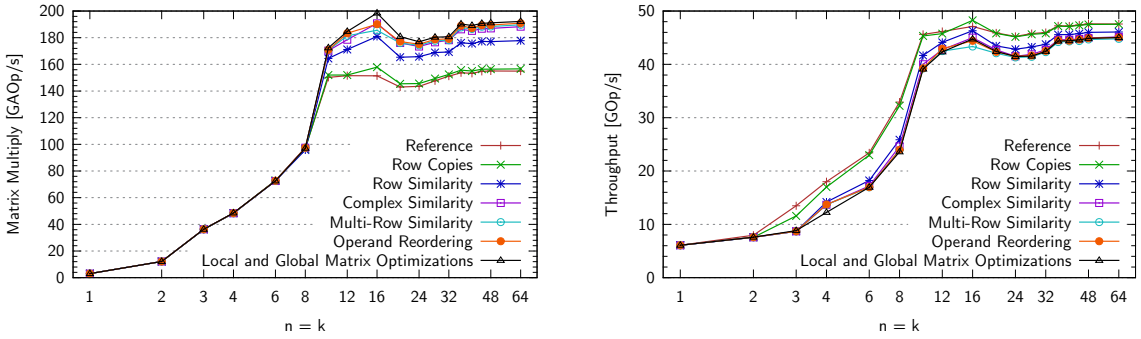
Fig. 17.15 shows the fill-ratio reduction by global and by local matrix optimizations. While the reduction factor for local optimizations becomes asymptotically constant, the potential for global optimizations decreases with the matrix size. The reason is that the possibility for global optimizations occurs by coincidence, too. Thus, the probability falls with more columns and rows respectively. In contrast to local optimizations, there is no upper limit on the submatrix size considered at a time.

The figure shows that both local and global optimizations can reduce the fill-ratio. The local version brings greater benefit. Applying both optimizations together brings some improvement

[8] Possible optimizations arise by coincidence and depend on the employed matrix, i. e. on the chosen λ_i. This explains the large variations especially for small matrices. (In large matrices they are averaged out.)

but definitely not the sum of the individual reduction factors. This is due to both optimizations utilizing more or less the same degrees of freedom. Considering this result, it is likely that the GC method for fill-ratio reduction uses similar degrees of freedom. It is very well possible that the combination of GC matrices as well as local and global optimizations will bring no or only little speedup. The final optimized fill-ratio itself goes asymptotically to 35.5 %.

A statistical analysis reveals that the fill-ratio of the decoding-matrix is similar to the one of the encoding-matrix: about 50 %. Local optimizations are applied at runtime and perform equally well during decoding. Since global optimizations are applied during matrix creation and since the encoding-matrix predetermines the decoding matrices, global optimizations cannot be used for decoding. Still, statistics shows that they alter the encoding matrix in a way which does not change the fill-ratio of the decoding-matrix and thus they have no negative effect.



(a) Matrix Multiply Performance

(b) Instruction Throughput

Figure 17.16: Improvements by Optimized Matrices width reduced Fill-Ratio [XII]

Eventually, Fig. 17.16 shows matrix-multiply performance and throughput for all above described optimizations. While the matrix-multiply performance increases, the throughput had already reached the peak and can not increase at all. Instead, it even falls slightly because the optimizations introduce additional more complicated register dependency chains [Int 11, 2.1.3]. The same work is performed faster with fewer instructions in total but also fewer instructions per cycle.

One more optimization is tacitly applied: inspecting Fig. 17.16 in detail reveals that in average, especially for medium to large matrices, all the matrix optimizations improve the performance. For small matrices, however, the encoding can get even slower. Due to the local optimizations, high *xmm* registers appear as third operand in ternary instructions leading to a larger encoding (see Section 17.3.3.2). For larger matrices, this effect also occurs but is simply averaged out. This has been coped with by **operand reordering**. The instruction $pxor\ r_a,\ r_b,\ r_c$ is equivalent to $pxor\ r_a,\ r_c,\ r_b$ but the encoding size can vary depending on the value of b and c. In an optimized version, the optimal order of the operands is chosen. This ensures a speedup for all matrices.

17.3.4.3 Eliminating Instructions

Besides matrix optimizations, improvements to the assembler code itself can also save some instructions or cycles respectively (which is the work the compiler usually does). This section lists the applied optimizations.

Direct Loads Due to the matrix optimizations, there are rare cases where a data word of D is used only once in a register blocking phase. In this case, there is no sense in loading it to a scratch register first but it can be used directly as a source operand for the XOR instruction.

Page Alignment In very rare cases, a slowdown is observed when an instruction starts one or two bytes before a page boundary. If so, the instruction is aligned to the page size of 4096 bytes

by padding with *nop* (no operation) instructions. In most cases, this does not change anything at all; it negates the slowdown and never has a negative effect.

Register Renaming In general, an accumulation register r_a is not initialized with zero. Instead, the first nonzero element M_{i_0,j_0} (stored in the scratch register r_s) of M affecting the value of r_a initializes the register via $r_a = r_s$. In this case, the scratch register r_s is actually not needed. The register r_a can be directly initialized from memory and then used as the source for further move or *XOR* instructions instead of the scratch register. This saves the instruction for copying r_s to r_a.

Load / Store Interleaving The processor can issue two SSE loads and one SSE store to the L1 cache per cycle. When the intermediate results for the instruction level blocking are stored, the instructions can be interleaved such that one store is combined with two loads – as far as possible. This ensures that the pipeline is not stalled because of too many successive stores.

It was attempted to extend that to a store / calculation interleaving but this slows down the *XOR* instructions in some cases. The attempt was thus dropped.

Ternary Optimizations In rare cases, *movdqa* r_a, r_b is followed by *vpxor* r_a, r_a, r_c. The move can be eliminated by writing *vpxor* r_a, r_b, r_c.

Small Pointer Offsets Loads and stores are performed with a base address in a register and offsets corresponding to the index i of D_i. The offset x is encoded in the opcode either as one byte if $-128 \leqslant x < 128$ or as four bytes else. Regularly incrementing the base register by 256 results in a smaller code size. Unfortunately, most probably due to register dependency chains, this slows down the encoding and was discarded. Still, a simpler optimization is possible. Shifting all base pointers by +128 bytes at least doubles the period with $-128 \leqslant x < 128$, where the small opcode is used.

Results Figures 17.17 and 17.18 show how many instructions can be saved and how the performance develops. The influence on the performance is rather small. It must, however, be considered that the code is getting close to the computational peak performance. This makes improvements harder and harder. In fact, many assembler optimizations were developed prior to the matrix optimizations presented before. Without the matrix optimizations, the speedup is higher.

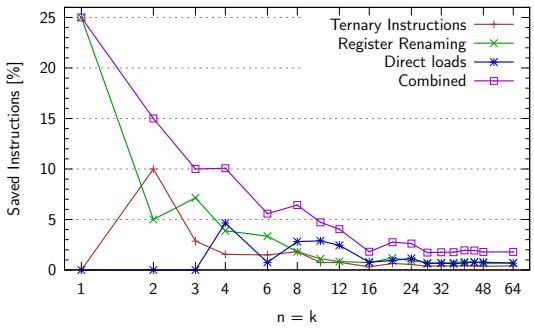


Figure 17.17: Low Level Assembler Instruction Optimizations [XII]

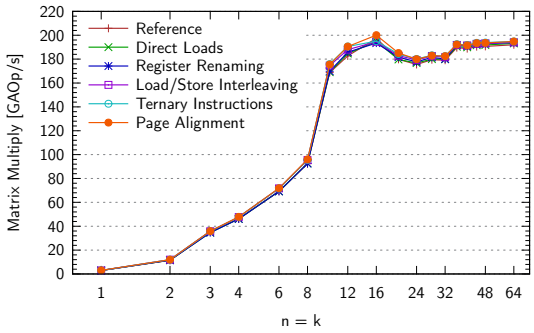


Figure 17.18: Performance Gain by Instruction Optimizations [XII]

17.3.5 Improved Matrix Size (Smaller l Dimension)

Since with all above improvements it became very hard to optimize the implementation of the automorphic *XOR*-only code any further, a bottleneck from which all codes suffered since the beginning has been eliminated. Up until now, only the matrices from Example 36 in Appendix F.1

have been used. Thus, in many cases the dimension l was not optimal. Matrices for arbitrary l can be obtained according to Section 16.3.2. The proof is not constructive. Still, in the CRS case relevant for BGEMM and *XOR*-only codes, the matrices can be easily calculated (see Appendix F.2). They are not included for IGEMM and the *Add*-only variant since this is more complicated and the binary codes are faster anyway.

Figures 17.19 and 17.20 show the results. Measurements classified as "polynomial list" use arbitrary l while the others do not. Similarly to Section 17.3.4.2, the achieved update bandwidth rises while the matrix-multiply performance even falls. The reason lies in the simple fact that fewer operations are needed for the same task. New possible values of l are: $3, 5, 7, 8, 9$. Especially the filled gap between six and ten (corresponding to $32 < n = k < 512$) results in a significant performance gain. For values of l that have been possible before, everything stays the same.

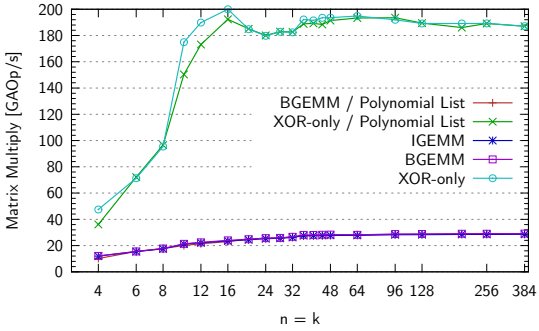


Figure 17.19: Performance Gain by Optimized Matrix Dimension [II,XII]

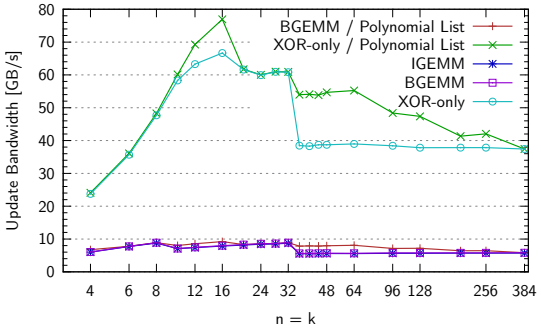


Figure 17.20: Instruction Throughput for Optimized Matrix Dimension [II,XII]

17.3.6 Large Matrices

Up until now, only small to medium size matrices have been considered ($n \leqslant 64$). With larger matrices ($n > 64$) new aspects become relevant.

17.3.6.1 Assembling Large Codes

Both assembler source code and binary code grow linearly with the matrix size, i. e. with the product $n \cdot k \cdot l^2$. For huge matrices this can become several hundreds of megabytes up to a gigabyte or even more. It turns out that the memory size required by the YASM assembler for internal buffers is multiple times the size of the source code. This makes it hard, if not impossible, to assemble the binary code for large matrices ($n \geqslant 512$).

To cope with this problem for good and all, a new integrated assembler has been written from scratch, optimized for speed and memory usage. The assembler does not support the full x86 instruction set. It is sufficient to implement support for x86, SSE, and AVX instructions required. Measurements show that it outperforms YASM by a factor of three to four in compilation time for large source files. Even further, it can compile input files of almost arbitrary size with less than a hundred megabytes of internal buffers.[9]

For even larger matrices, the source code itself becomes too big to fit in memory. The binary code is usually one fifth to one fourth of the source code in size. There are situations in which the binary code fits in memory while the source code does not. The assembler is written such that it can assemble a source stream in a single pass, instruction by instruction.[10] This way, the source code is never stored in memory completely but compiled directly.

[9] Internal buffers are used primarily for jump addresses, which are not frequently needed by QEnc.
[10] Assemblers usually perform a multi-pass assembly to handle jump addresses.

17.3.6.2 L2 Instruction Blocking

At about $n = k = 128$, the code size comes in the range of the L3 cache size of the Sandy Bridge. As the L3 cache is used for the D-matrix entries as well (L2 blocking), not the entire cache is available for the instruction stream. This leads to a non-dramatic but significant decrease in performance. The following optimizations mitigate the problem.

Adapted L2 Blocking The optimal L2 blocking is redetermined after all the above optimizations have been applied. The optimal value for the new code is 96. It is used from now on.

L2 Instruction Blocking Analogously to L1 instruction blocking, an L2 instruction blocking option is implemented.

Huge Pages The standard page size of 4 KB is not very well suited for instruction streams of such size. QEnc can utilize huge pages (see Appendix C.3 for details) for storing its automorphic code. The benchmark suit optionally uses huge pages for the input and output data, too.

Fig. 17.21 shows the improvements. Increasing the L2 blocking brings a general speedup. L2 instruction blocking becomes only relevant for huge matrices ($n > 256$) and is not significantly faster. Huge pages for code and especially for data speed up the encoding, in particular for large matrices. With the combination of huge pages and L2 instruction blocking, performance remains good up until $n = 512$. (QEnc automatically disables L2 instruction blocking for small matrices.)

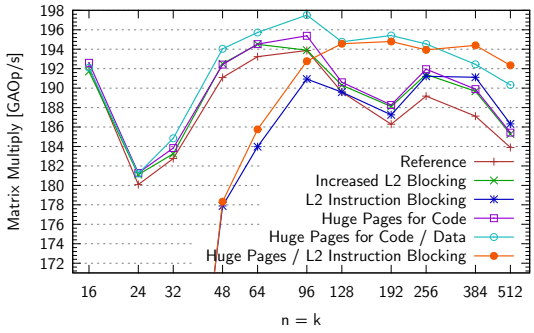


Figure 17.21: XOR128 Performance for Large Matrices [XII]

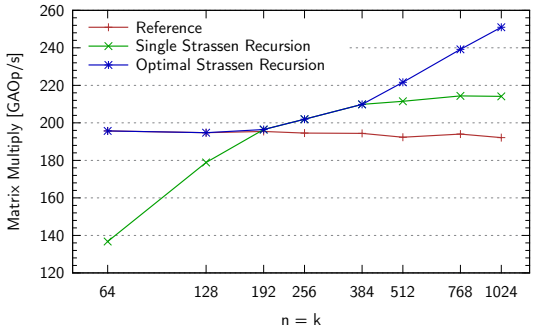


Figure 17.22: Performance Gain by Strassen Algorithm [XII]

17.3.7 Exploiting the Strassen Algorithm

The Strassen algorithm performs a recursive matrix multiplication. Each recursion step can bring a speedup of up to $^8/_7$ while the matrix dimensions for the next iteration are halved (see Section 16.5.2). However, the Strassen method introduces overhead, which is significant for small matrices. Therefore, at a predefined size the Strassen recursion is usually not continued but naive matrix multiplication is used. In order to determine this break-even point, a run with one Strassen iteration is compared to the naive matrix multiplication version. The break-even point sits where the Strassen algorithm becomes faster. For twice that matrix dimension, a second Strassen iteration can be used, for four times that size a third iteration, and so forth.

Fig. 17.22 determines the minimal dimension to $n = k = 192$. The optimal number of Strassen iterations is thus calculated by $\log_2(\frac{n}{96})$ rounded down. With this number of recursions, at $n = 1024$ a matrix-multiply performance of **251 GAOp/s** is achieved (by only one single CPU core). Compared to the single-threaded peak SGEMM performance of 27.5 GFlop/s (and the derived and equal BGEMM performance), the matrix multiplication is accelerated by a factor of 9.1.

17.3.8 Small Matrices

In contrast to large matrices, their small counterparts are limited by memory bandwidth not by instruction throughput. To determine an upper bound for the possible performance, the maximum achievable memory bandwidth is measured on the test systems. Two benchmarks are used: Stream [McC 95] and a simple self-written benchmark in assembler which simply performs SSE loads to consecutive memory addresses over one gigabyte of memory (in the following called Maximum Memory Throughput). The results are given by Table 17.23.[11]

System	Stream Benchmark (Copy)	Maximum Memory Throughput
Westmere, 1 Cores	11.74 GB/s	16.62 GB/s
Westmere, 6 Cores	15.85 GB/s	28.62 GB/s
Sandy Bridge, 1 Cores	12.12 GB/s	19.81 GB/s
Sandy Bridge, 4 Cores	12.18 GB/s	20.00 GB/s

Table 17.23: Maximum Memory Bandwidth on Westmere and Sandy Bridge [II,XII]

Multiple observations can be derived from the table:

- Single-thread throughput is better on the Sandy Bridge due to architectural optimizations.
- The multi-thread memory throughput achieves the maximum bandwidth which can be delivered by the memory controller. In this discipline, the Westmere is superior since it offers a triple-channel interface in comparison to two channels for Sandy Bridge.
- One single thread on the Westmere cannot achieve peak bandwidth, one thread on the Sandy Bridge can. So, multi-threading on the Westmere is likely to speed up the encoding while on Sandy Bridge from this perspective one single thread suffices – theoretically.
- The Stream benchmark (copy task) does not achieve the peak bandwidth.

The QEnc implementation described before achieves an encoding bandwidth of 5 GB/s on Westmere and 6 GB/s on Sandy Bridge. Since input and output are of equal size, the corresponding memory bandwidth is twice that high. These values are very similar to the single-thread Steam benchmark copy results on both architectures. It turns out that both Stream and QEnc lack behind the peak throughput for the same reason.

From the memory perspective, QEnc is very similar to the copy in the Stream benchmark. It reads one data stream and writes another (the calculation in between does not affect the memory). Both streams are of the same size. The Stream / QEnc performance is pretty much two third of the peak bandwidth. This is due to stores to memory locations which are not already contained in the cache. They lead to a memory access fetching the remaining entries of the cache line the store is performed to. Thus, the output data stream is both read and written doubling the memory load while the input stream is only read explaining exactly the factor of two third.

This (general problem) is resolved by streaming stores which bypass the cache [Dre 07, 6.1]. The improvement is demonstrated in Fig. 17.24. Both architectures show a nice speedup and come very close to the peak bandwidth. The Sandy Bridge achieves 9.03 GB/s .

17.3.9 Complex Code Example

Listing 17.25 shows an example of the generated assembler code for $n = k = 64$ demonstrating all optimizations. An L1-blocking size of 48, L2-blocking size of 96, and an instruction stream

[11] According to the specification, the theoretical bandwidth is 38.4 GB/s for Westmere and 25.6 GB/s for Sandy Bridge, which is significantly higher than the maximum bandwidths measured. Still, the maximum bandwidth actually measured is the better reference for examining whether QEnc achieves the highest attainable bandwidth or not since the theoretical peak bandwidth may be unreachable.

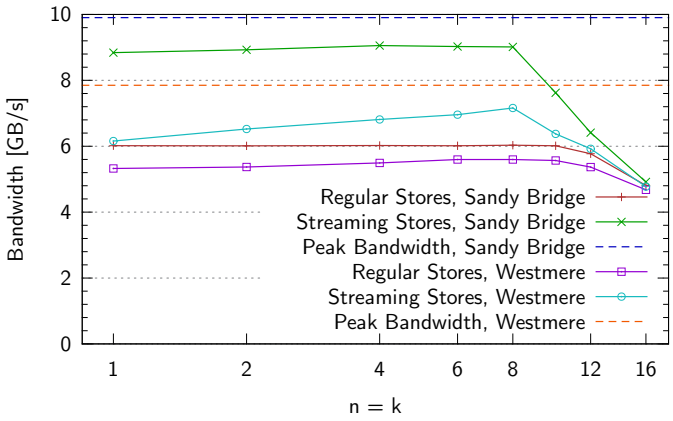


Figure 17.24: Performance Gain using Streaming Stores [XII]

blocking in 40 blocks is used. The first instruction stream block is presented in some detail, the following blocks are skipped as they are similar. (See Appendix F.3 for a C++ example.)

```
push rbx                              //Save non-volatile registers
...                                   //Input parameters: rdi: source address,
add rdi, 128                          //rsi: destination address, rdx: block count
add rsi, 128                          //SMALL_POINTER_OFFSETS: Shift base pointer
asmouterloop:                         //Outermost loop over L2 blocks
sub rdx, 96                           //L2 Data Blocking over 96 blocks
mov rcx, 96                           //Initialize register for inner loop
mov rax, rsi
mov rbx, rdi
asminnerloop_0:                       //Inner loop over first instruction stream

//L1 data blocking (Columns 1 - 48, Rows 1 - 14)
//Column 1
prefetcht0 [rbx + 7168 - 128]         //Prefetch next input data block
movdqa xmm2, [rbx + 0 - 128]          //REGISTER_RENAMING: Use xmm2 as scratch
movdqa xmm6, xmm2                     //DIRECT_LOAD: [rbx+16] not loaded to scratch
movdqa xmm15, xmm2
//Column 2
movdqa xmm3, [rbx + 32 - 128]         //Load column 3 while processing column 2
movdqa xmm13, [rbx + 16 - 128]        //DIRECT_LOAD: [rbx+16] directly used
//Column 3
vpxor xmm6, xmm6, xmm3                //DIRECT_LOAD: [rbx+48] not loaded to scratch
movdqa xmm12, xmm3
//Column 4
movdqa xmm8, [rbx + 64 - 128]
vpxor xmm2, xmm8, [rbx + 48 - 128]//DIRECT_LOAD + TERNARY_INSTRUCTION
//Column 5
prefetcht0 [rbx + 7232]               //Prefetch for next iteration
movdqa xmm1, [rbx + 80 - 128]
vpxor xmm3, xmm8, xmm3                //OPERAND_REORDERING: xmm8 not third operand
//Column 6
movdqa xmm4, [rbx + 96 - 128]
...
//ROW_SIMILARITY updates
vpxor xmm9, xmm9, xmm3
vpxor xmm14, xmm14, xmm6
vpxor xmm8, xmm8, xmm4                //MULTI-ROW_SIMILARITY
vpxor xmm8, xmm8, xmm6
```

```
//Store intermediate results for L1 data blocking
movdqa [rax + 0 - 128], xmm2
...
movdqa [rax + 208 - 128], xmm15
//L1 data blocking (Columns 1 - 48, Rows 15 - 28)
...
//L1 data blocking (Columns 49 - 96, Rows 1 - 14)
...
//Include intermediate results from previous step of L1 data blocking
vpxor xmm2, xmm2, [rax + 0 - 128]
vpxor xmm3, xmm3, [rax + 16 - 128]
//Store updated intermediate results for L1 data blocking
movdqa [rax + 0 - 128], xmm2      //STORE_INTERLEAVING: 1 store per 2 loads
vpxor xmm4, xmm4, [rax + 32 - 128]
vpxor xmm5, xmm5, [rax + 48 - 128]
movdqa [rax + 16 - 128], xmm3
...
movdqa [rax + 144 - 128], xmm11
//L1 data blocking (Columns 49 - 96, Rows 15 - 28)
...
add rax, 7168                     //Load address for next L1 block
add rbx, 7168                     //Same increment for rax and rbx since n = k
dec rcx
jnz asminnerloop_0                //End of first instruction stream
mov rcx, 96                       //Reinitialize register for inner loop
mov rax, rsi
mov rbx, rdi
asminnerloop_1:                   //Inner loop over second instruction stream
...
jnz asminnerloop_39               //End of last instruction stream
add rdi, 688128                   //Load addresses for next L2 block
add rsi, 688128
cmp rdx, 0
jne asmouterloop
pop rsi                           //Restore non-volatile registers
...
ret
```

Listing 17.25: QEnc Elaborate Assembler Code Example for $n = k = 64$

17.3.10 Analysis

The final performance of the *XOR*-only code with 128-bit SSE registers over the full range of matrix dimensions is visualized in Fig. 17.26. It is clearly visible that either the bandwidth is limited by the peak bandwidth or the instruction throughput is limited by the peak throughput of the hardware. (For the same reasons as for Fig. 17.13, the peak throughput can be outmatched in some cases.) Using the peak bandwidth, the peak throughput, the fill-ratio of the employed matrices, and the average number of assembler instructions per *XOR* operation, the highest possible update bandwidth can be calculated. Comparing this to the achieved update bandwidth reveals how close the implementation comes to the hardware limits.

Analyzing the metrics in detail reveals:

- The bandwidth is close to the peak bandwidth until the instruction throughput becomes the limiting factor. At all time, the performance hits hard architectural limits.
- The update bandwidth increases strongly at the beginning reaching a peak where the matrix-multiply performance has its first plateau. Afterward, it falls slowly due to the increasing

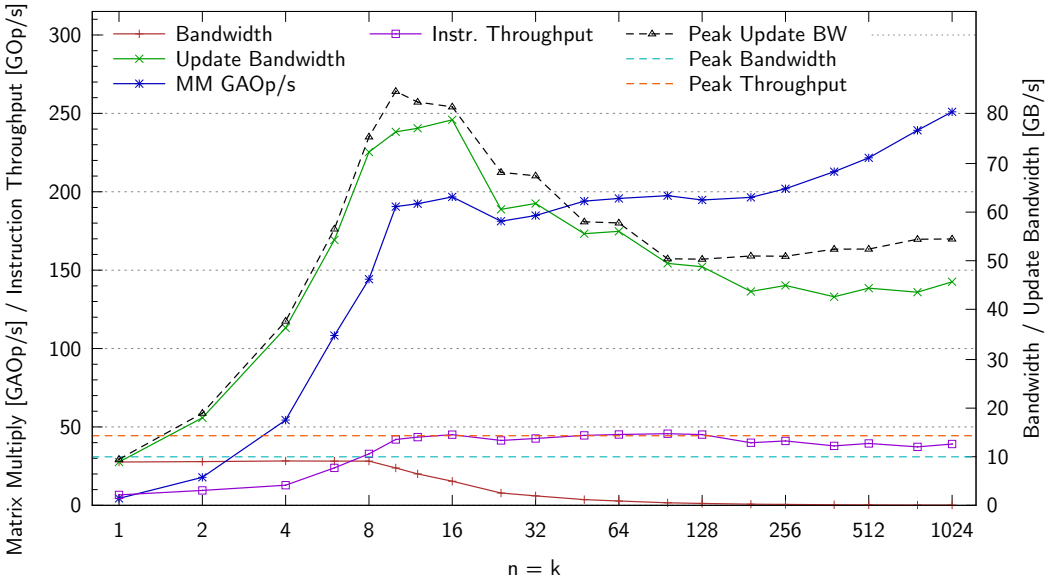


Figure 17.26: Final QEnc XOR128 Performance [XII]

dimension l which causes a greater computational complexity at constant matrix-multiply performance. Toward large matrices the update bandwidth becomes more or less constant. The Strassen algorithm compensates the growth of l and for even larger matrix sizes, it actually increases the bandwidth again since its performance increases much faster than l.

- Matrix-multiply performance falls slightly at two points: at $n \approx 20$ and at $n \approx 128$. At these matrix sizes, the code size exceeds the L1 and L3 cache size of the processor.
- The instruction throughput increases until it saturates closely below the theoretical peak. When the code size hits the L1 cache size, it falls slightly in the same way the matrix-multiply performance does. Above $n = 192$ it falls again due to the Strassen overhead.

Fig. 17.27 shows how the size of the code increases with the matrix dimension. The mostly polynomial dependency yields a linear curve in the bilogarithmic plot with two exceptions. For small matrices the slope is smaller since the matrix fill-ratio is lower. At three matrix sizes a small step is observed: namely at $n = 192$, $n = 384$, and $n = 768$. These numbers correspond to where the Strassen recursion gets one step deeper, which increases the code size by seven fourth each time.

17.3.11 Variants

17.3.11.1 *Add*-only Encoding

An *Add*-only code (see Section 16.4.2) can be implemented in the same way as the *XOR*-only code. The encoding-matrix is encapsulated in the instruction stream. In fact, *pxor* and *vpxor* instructions are simply exchanged by *paddd* and *vpaddd*. In contrast to logical operations, the processor can perform only two integer SSE operations per cycle (see Table 17.3). Still, one move instruction can be executed in parallel. Thus, the *Add*-only code reaches an instruction throughput between two and three SSE instructions per cycle – less than the *XOR*-only code.

17.3.11.2 A 256-bit *XOR*-only Code with AVX

Full AVX support can be added easily, especially since ternary AVX operations are already used: *ymm* registers are used instead of *xmm* registers. Unfortunately, since the Sandy Bridge

processor can execute only one logical AVX instruction and one AVX move in parallel, only slightly above one AVX instruction is executed per cycle. This corresponds to two to three SSE instructions and thus results in similar performance as for the *Add*-only SSE code. An *Add*-only AVX code can not be implemented as Sandy Bridge lacks AVX support for integer calculation. This is foreseen for the next AVX version (AVX2), which will be introduced with the new Haswell CPU family.

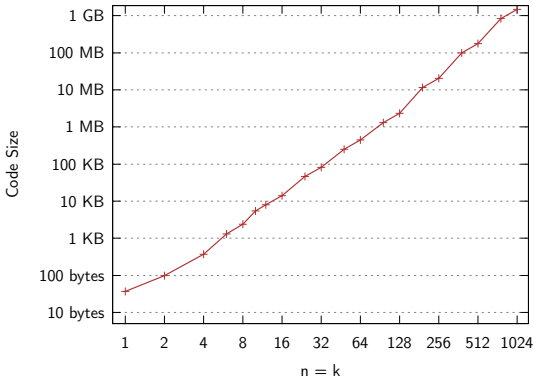


Figure 17.27: Binary Code Size of XOR128 Implementation [XII]

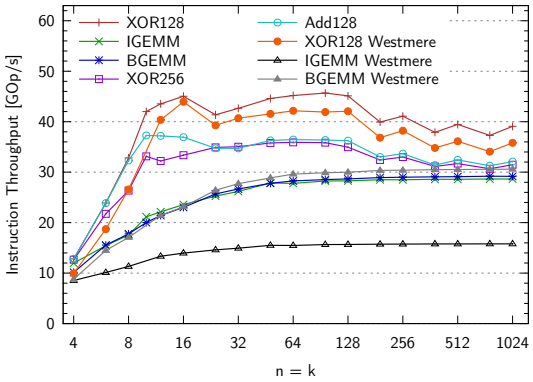


Figure 17.28: Final Instruction Throughput of all Encoding Implementations [II,XII]

17.3.12 Comparison

The performance of IGEMM and BGEMM as well as *XOR*-only and *Add*-only codes using SSE and AVX are presented in Figures 17.28 to 17.31. Clearly, the *XOR*-only code with SSE emerges as winner. The AVX variant suffers heavily from the fact that only one single simultaneous logical operation is possible on Sandy Bridge. This is very likely to improve with the next processor generations. Then, most probably, AVX will achieve twice the performance of the SSE implementation. The *Add*-only code shows similar performance as the *XOR*-only AVX code. Certain matrix dimensions show significantly lower performance. The reason is that the *Add*-only version suffers from the same implementation flaw as the IGEMM version, which cannot encode with arbitrary l. However, implementing the missing matrices for $\mathbb{Z}/2^b\mathbb{Z}$ codes was considered a waste of time as the *XOR* code is faster anyway. All automorphic codes outperform the IGEMM and the BGEMM codes by an order of magnitude.

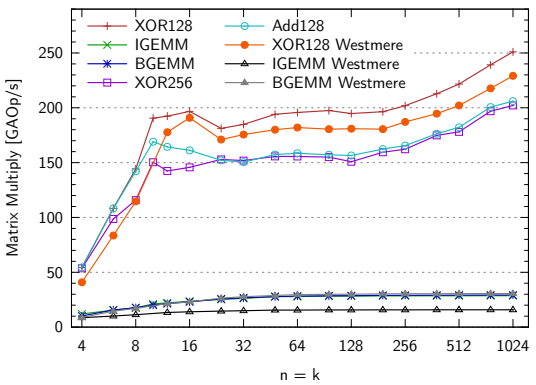


Figure 17.29: Final MM Performance of all Encoding Implementations [II,XII]

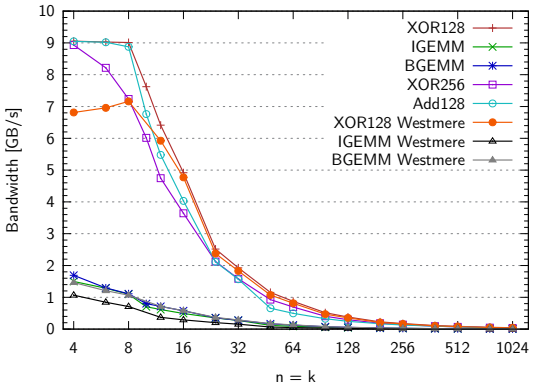


Figure 17.30: Final Bandwidth of all Encoding Implementations [II,XII]

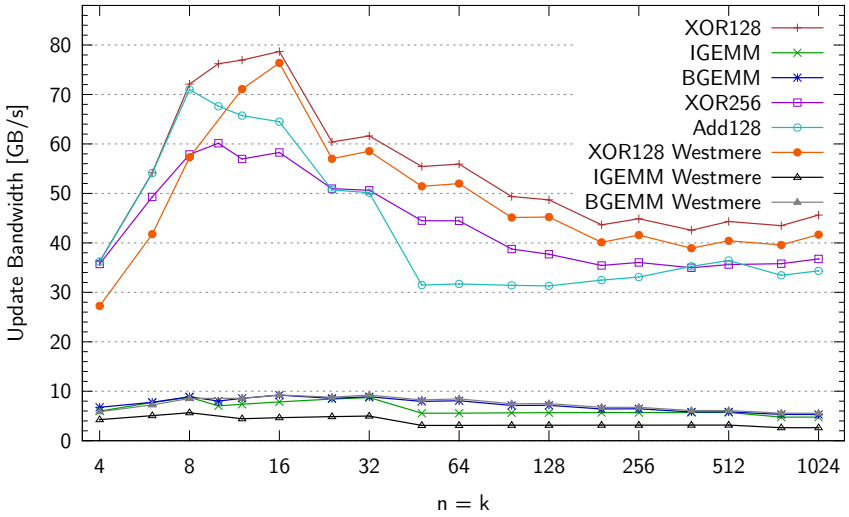


Figure 17.31: Final Update Bandwidth of all Encoding Implementations [II,XII]

17.4 Multi-Threading

Up until now, only a single processor core has been used. The encoding process can be multi-threaded trivially by making each thread encode a different portion of the source data. This is implemented using OpenMP [OMP]. The speedup is shown in Fig. 17.32 while the multi-threaded bandwidth is visualized in Fig. 17.33.

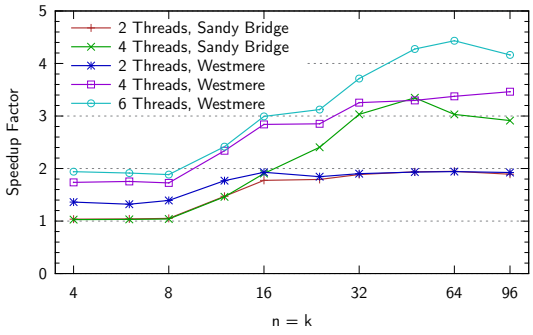


Figure 17.32: Speedup of the *XOR*-Code by Multi-Threading [XII]

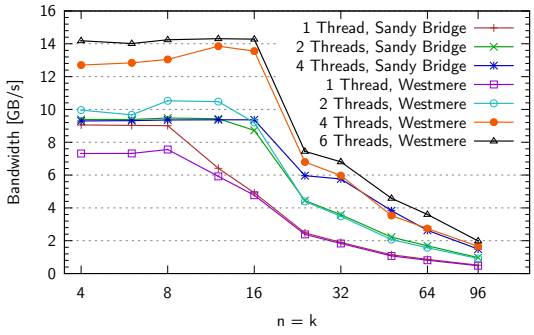


Figure 17.33: Memory Bandwidth achieved by Multi-Threaded *XOR*-Code [XII]

On Sandy Bridge, there is no significant speedup for small matrices simply as a single thread can already reach peak memory bandwidth (see Section 17.3.8). Increasing the number of cores shifts the tradeoff point where the limitation switches from bandwidth-bound to compute-bound to the right. For medium to large matrices, the speedup factor for four CPUs is at least three. Apparently, the memory controller is unable to sustain the full load of all four CPU cores and the shared CPU caches take their toll.

On the Westmere, also the small matrix performance improves on multiple cores (since one core cannot reach the full memory bandwidth), although not to the same extent as the performance for large matrices. Using all six Westmere cores, the processor can play the strength of its three memory channels and reach an encoding bandwidth of 14.3 GB/s. As for Sandy Bridge, the performance does not scale completely linearly with the number of cores.

Naturally, the OpenMP approach is not very optimized leaving room for improvement.

17.5 Update-Codes

Partial updates according to Lemma 32 are a critical task for all redundant storage systems. Recall that an update computes $C_i^{(\text{new})} = C_i^{(\text{old})} + M_{ij} \cdot (D_j^{(\text{new})} - D_j^{(\text{old})})$. For this purpose, a special update implementation is made available. The update is performed simply by a multiplication with one matrix element of the MDS-code which corresponds to a binary or integral $l \times l$ matrix multiplication for vector-MDS-codes. Since the matrix sizes are tiny, the update process is entirely memory-bound. In contrast to full encoding, QEnc does not only write to the output stream but also reads the old redundancy data from it. Thus, the streaming stores cannot be used and the highest achievable encoding bandwidth is one third of the memory bandwidth instead of one half. Prefetching the output stream guarantees maximum bandwidth in this case. Fig. 17.34 shows that the peak bandwidth measured by the SSE sequential read benchmark for Table 17.23 is actually achieved for all matrix sizes.

A question is what is the maximum matrix size for which the update implementation reaches the peak bandwidth. The matrix dimension $n = k = 64$ corresponds to $l = \log_2 128 = 7$ (due to $2^l \geqslant n + k$) and requires about 20 GAOp/s matrix-multiply performance to achieve peak bandwidth. The required matrix-multiply performance (in order to achieve peak bandwidth) is linear in l. Extrapolating to $l = 56$ predicts a required performance of 160 GAOp/s. The peak performance achieved on Sandy Bridge is more than 190 GAOp/s leaving a safety margin of 30 GAOp/s. Naturally, $l = 56$ is high enough to ensure maximum memory bandwidth in the update process for arbitrary n and k (to be precise $n + k \leqslant 2^{56}$). Hence, QEnc achieves peak bandwidth for update-codes in any case – also in the future.

17.6 Dependency on k

Fig. 17.35 shows the dependency of all metrics on the k parameter for $n = 64$ fixed. In this case, the peak encoding bandwidth is not constant but depends on k because input and output streams are of different size. Up to $k = 4$, 75 % or more of the peak bandwidth are achieved. For $k = 6$ the instruction level blocking is activated which explains the slight decrease in the matrix-multiply performance. For larger k, the matrix-multiply performance steadily increases from 165 to 190 GAOp/s. For comparing arbitrary k at fixed n to the well-known case $n' = k'$, a square encoding-matrix for $n' = k'$ of about the same total size is chosen as a reference, i. e. $nl \cdot kl \approx n'l' \cdot k'l'$. The figure reveals that for every k the matrix-multiply performance is very similar to the reference performance of such a square matrix, as long as the bandwidth is no limiting factor. This analysis confirms that the QEnc performance depends mostly on the matrix size but not on the matrix shape and thus not on k.

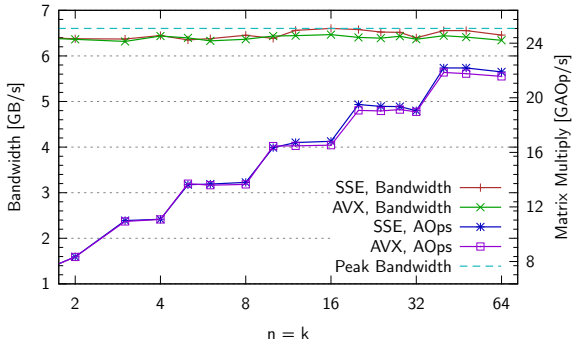


Figure 17.34: Update Performance of the *XOR*-only Code [XII]

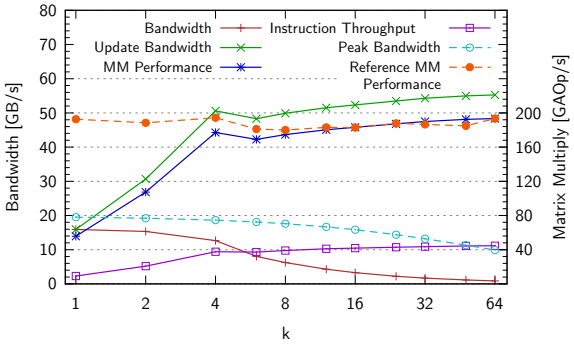


Figure 17.35: Encoding Performance Dependency on k [II,XII]

Chapter 18

Encoding with GPU & FPGA Accelerators

Both, matrix multiplication based codes and automorphic codes are ported to GPUs. In addition, a proof of concept FPGA (see Appendix C.6) implementation exists. The bottleneck for small matrices is the PCI Express bandwidth while for large matrices the accelerator can play its strength. In the following, only GPU kernel / FPGA performance is analyzed. To achieve good performance on the host, a streaming framework as in Section 11.2 is required. As long as the PCI Express bandwidth does not pose a limitation, the ratio of GPU kernel performance to host performance should be the same as in the CALDGEMM case (94.1 % – 98.5 % depending on the platform).

18.1 Matrix Multiplication based Codes for GPUs

Section 12.13 introduced IGEMM and BGEMM implementations for AMD GPUs. Good performance is achieved for matrix sizes $h, k \geqslant 2048$ which translates to $n = k \geqslant 256, l \geqslant 9$. The encoding performance of the kernel is exactly the GPU IGEMM / BGEMM kernel performance shown in Table 12.44, i. e. IGEMM encoding achieves 492 $\mathrm{GAOp/s}$ and BGEMM can reach 1024 $\mathrm{GAOp/s}$.

18.2 *XOR*-only Encoding with OpenCL

The automorphic approach has been ported to GPUs by generating OpenCL source code rather than binary x86 assembler code. The blocking factors and register counts are optimized for the GPU architecture by parameter range scanning in the same way as for the CPU. As a reference, the OpenCL version on the CPU is compared to the assembler version as well. Three OpenCL compilers are used: namely from the Intel OpenCL SDK 1.1, from the NVIDIA CUDA SDK 4.0.1, and from the AMD APP SDK 2.4 for both CPU and GPU.

It turns out that the AMD and the Intel compilers cannot compile the code for $n \geqslant 16$.[1] The NVIDIA compiler can compile the code up to $n = 128$. For larger matrices, it runs out of memory.[2] Fig. 18.1 shows the compilation time for all compilers. For comparison, also YASM and the QEnc integrated assembler are included. The compilation time depends greatly on the compiler. The NVIDIA compiler can process the biggest code and comes out fastest.[3] Fig. 18.2 shows that the OpenCL performance on the CPU lacks far behind the assembler performance – as expected. The Intel compiler produces much faster code than the AMD compiler. The multi-threaded OpenCL code scales well to the six Westmere cores but is much slower than the assembler code.

[1] The AMD compiler causes a segmentation fault during compilation while the Intel compiler produces incorrect code.
[2] Since the compiler is a 32-bit executable, it is limited to 2 GB.
[3] In fact, also the compiler version is very relevant. The previous NVIDIA compiler of the SDK version 3.2 required about 5000 seconds for the $n = 16$ code.

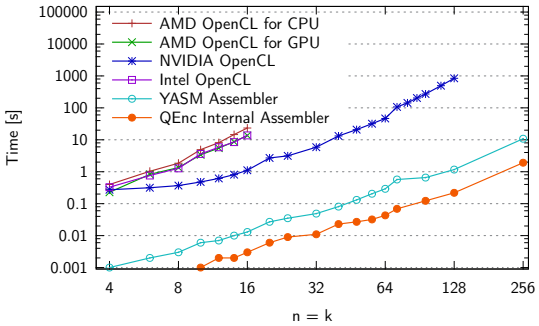


Figure 18.1: Compilation / Assembly Time of *XOR*-only Code [II]

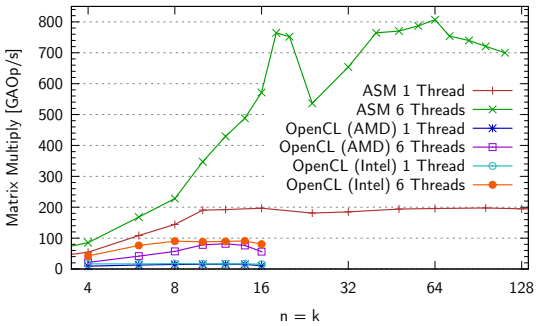


Figure 18.2: QEnc OpenCL Performance on CPU [II]

18.3 An FPGA Implementation

While GPUs are designed specifically for the highest calculation throughput, FPGAs are more flexible and excel in bit manipulation. The *XOR*-only code is the natural choice for the FPGA implementation. Instead of assembler and OpenCL code, QEnc can generate VHDL[4] FPGA code as well. The following tests are performed with a Xilinx Virtex 6 XC6VLX240T-2 FPGA, which is of medium size and costs about as much as the most expensive CPUs or GPUs used in this thesis. The current implementation is not as seamless as the assembler or OpenCL versions, but in principle, compilation, place and route, and image deployment to the FPGA can be automatized with scripts making the FPGA encoding as transparent as the other versions to the user. The blocking for the matrix multiplication with intermediate results stored in memory is exchanged for a pipeline, which is very similar to a one-level blocking: the innermost loop corresponds to all calculations the FPGA performs in one clock cycle. The pipeline length depends on the number of operations per clock cycle and on the employed matrix size. The number of operations per clock cycle defines the maximum possible clock rate. Since encoding is a streaming process by definition, a large pipeline is no problem at all. Hence, the pipeline steps are designed rather small to achieve high frequencies. As the assembler code, the VHDL code grows with the matrix size and is limited by the number of FPGA slices (which contain LUTs and registers).

Every computed codeword C_j is computed independently and has its own independent, parallel logic path on the FPGA. Hence, disregarding the latency, the FPGA computes a full set of codewords each cycle. The encoding bandwidth is thus $B = f \cdot n \cdot l \cdot t/8$ [Bytes/s], with f the FPGA frequency and t the width of the parallel code. Unlike the CPU, there is no general restriction on t. The value of t can be chosen small to reduce the usage of slices; it can be chosen large to speed up the encoding. The user can flexibly select the best option depending on the task. It seems irritating, that more complex codes with bigger n are faster than those with small n, but this comes from the fact that the complex code with large n requires significantly more logic, which operates in parallel. In the same way, a small matrix with larger t requires more cells but becomes faster, too. Still, because the complexity goes with $n \cdot l^2 \sim n \cdot (\log n)^2$ but it depends linearly on t, codes with small n and large t are faster – if they occupy the same number of slices.

Interestingly, the FPGA can benefit from many of the assembler optimizations since lots of them reduce the required number of operations in the computation. Still, compared to the memory capacity of contemporary systems, the amount of slices is certainly limited. Hence, large codes that require several gigabytes of assembler code can never run on the FPGA. Actually, the code size limitation is much smaller. The Xilinx compiler has enormous problems with the $n = k = 64$ code. Synthesis takes about ten times as long as for $n = k = 48$ and the achieved frequencies are much worse than with smaller parameters. A workaround for the future might be emulating the large matrix multiplication with multiple smaller ones reducing the complexity of the entities.

[4] **V**HSIC **H**ardware **D**escription **L**anguage – VHSIC standing for **V**ery-**H**igh-**S**peed **I**ntegrated **C**ircuits.

18.4 Performance

Figures 18.3 and 18.4 show the bandwidth and the matrix-multiply performance of the OpenCL GPU code. The NVIDIA GPU achieves 75 % of the peak memory bandwidth for very small matrices. The AMD GPU remains slower. For larger matrices the NVIDIA GPU reaches about 900 GAOp/s, the AMD GPU reaches about 780 GAOp/s. The comparison of GPU to CPU performance is postponed to the next chapter.

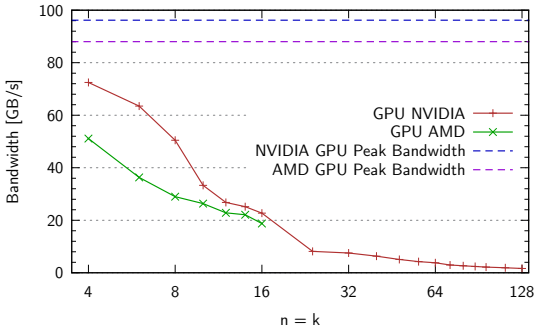


Figure 18.3: QEnc Bandwidth on GPU [II]

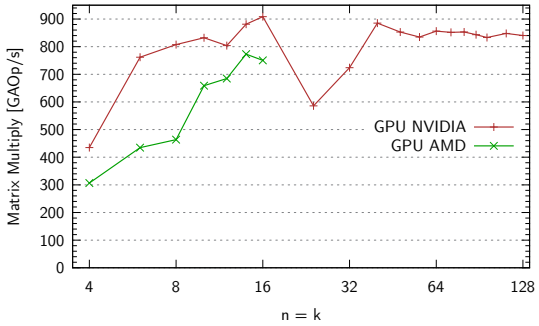


Figure 18.4: QEnc Performance on GPU[5] [II]

The performance of the FPGA implementation is difficult to compare to the other versions, because one can trade more slices for more performance more or less linearly. To compare FPGA versions with different n against each other and against CPU / GPU encoding, it is most instructive to scale the FPGA results to 100 % FPGA utilization. Hence, Table 18.5 contains the **extrapolated maximum bandwidth**, which is defined as the bandwidth (with $t = 1$) divided by slice occupancy. The advertised maximum design frequency of 600 MHz is reached or exceeded except for the large matrices, whose code the compiler can hardly handle. For the latter case, the previously suggested splitting of the multiplication into multiple entities might help.

$n = k$	f [MHz]	Bandwidth [GB/s]	Pipeline	Occupancy [%]	Max. Bandwidth [GB/s]
8	682	2.73	4 stages	0.12	2187
12	685	5.14	6 stages	0.38	1344
16	638	6.38	8 stages	0.68	939
24	614	11.07	15 stages	2.14	518
32	621	14.90	20 stages	3.78	394
48	556	23.35	34 stages	9.43	248
64	321	17.98	45 stages	14.48	124

Table 18.5: FPGA Encoding Performance

The extrapolated FPGA results are outstanding, of course; but there are two major drawbacks: FPGAs are impractical for small to medium n due to PCI Express bandwidth limitations; they are unqualified for large n due to finite slices and compiler problems. The most promising field of application for FPGA encoding is a design which requires encoding capabilities on the chip with reasonable performance and low slice utilization. Since FPGA results are off scale compared to traditional processors and graphic cards, they are not included in the next chapter's comparison.

[5] There is a dip in the performance of both the multi-threaded CPU version and the GPU version exactly at the point where L1 instruction level blocking and L2 blocking set in. These are also the measuring points with the greatest demands on the caches because the caches are used, but they are refilled extremely frequently. For larger matrices, data can remain in the caches for longer and performance returns to the normal level. Smaller matrices simply do not use the L2/L3 caches and never refill the instruction cache. For this reason, a cache bottleneck is the most plausible explanation. This assumption is fortified by the fact that the single-threaded version, with smaller stress on the shared caches by design, does not show this behavior.

Chapter 19

Results

19.1 Achieved Results

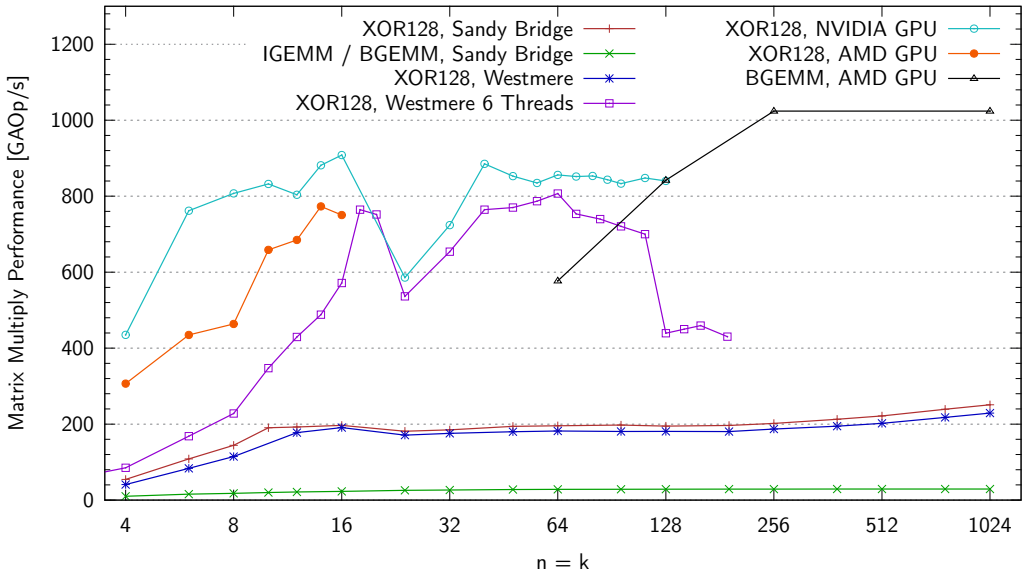


Figure 19.1: Final Encoding Performance Comparison (Performance)[5] [II,XII]

The QEnc library, which enables failure erasure coding via BGEMM and IGEMM as well as via automorphic assembler and OpenCL *XOR* code has been presented. Figures 19.1 and 19.2 gives an overview. Clearly, the assembler code is the fastest on a CPU. BGEMM (and IGEMM) lack far behind. On a single core, Sandy Bridge is slightly faster than Westmere, due to smaller opcodes. The single-threaded code is bandwidth-limited up to $n = 10$, above it is compute-limited. The multi-threaded code can maintain this (Sandy Bridge) or a higher (Westmere) bandwidth up to about $n = 20$. Unfortunately, the speedup does not grow linearly with the number of cores. Still, the Westmere's six-core matrix-multiply performance reaches 800 $^{\mathrm{GAOp}}/_{\mathrm{s}}$ which is (at different matrix sizes) as fast as the AMD GPU and only slightly slower than the NVIDIA GPU.

On the GPU, the automorphic OpenCL code performs well but does not even come close to the theoretical peak performance. In contrast to the CPU, the BGEMM code is faster but works only for large matrices. However, the BGEMM code is optimized CAL code. Most probably an automorphic GPU IL or PTX code can raise the automorphic GPU encoding to higher performance levels. Such an implementation has not been attempted for multiple reasons: GPU compilers encounter problems with huge source codes, no general framework for writing GPU assembler exists, and GPU architectures differ more than CPU ones complicating universal optimizations.

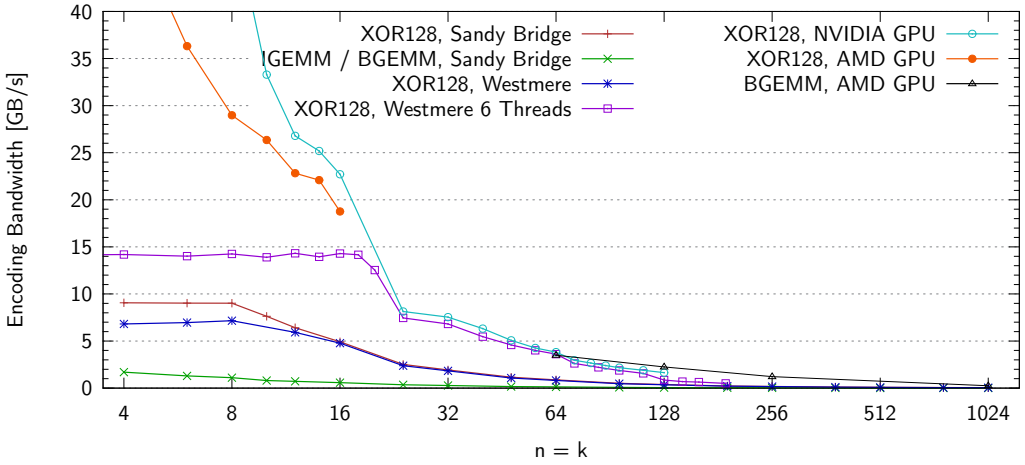


Figure 19.2: Final Encoding Performance Comparison (Bandwidth)[5] [II,XII]

GPU kernel performance for small matrices is unmatched due to the immense memory bandwidth. Unfortunately, there is literally no gain in practice because the PCI Express interface poses a bandwidth limit of about 6 GB/s. (The theoretical peak throughput of PCI Express 2.0 is 8 GB/s but this is not reachable in practice. Compare Table 7.3 as well as Figures 11.4 and 12.13.) At $n = 40$ the GPU kernel's encoding bandwidth drops to 6.3 GB/s. For smaller matrices, GPU encoding is inevitably PCI Express bandwidth-bound to 6 GB/s (with PCI Express 2.0). By coincidence, the CPU bandwidth drops below 6 GB/s at quite exactly $n = 40$, too (see Fig. 17.33). Thus, below $n = 40$ the multi-threaded CPU encoder is fastest, for larger matrices the GPU encoder performs better – and is not inherently PCI-Express-bandwidth-limited. Above $n = 128$ the OpenCL compiler is unable to compile the sources. In this region, only the matrix-multiplication version works on the GPU, which achieves peak performance at $n = 256$ and above.

Related publicly available failure erasure coding implementations can be categorized as follows:

- Most implementations of the original Reed-Solomon-Code use logarithm lookup tables and are inevitably memory-bound [Pla 97]. (Lookup table access is the bottleneck hitting the bandwidth limit. This is totally unrelated to the bandwidth metric measuring the raw encoding rate, which poses a limit for QEnc performance at small matrix sizes.) Optimized direct Reed-Solomon approaches [Kal+ 11] can overcome this problem, but are still limited by the slow emulation of the ring operation. In general, codes of this category often put tight restrictions on the possible l (e. g. $l = 2^{16}$), unnecessarily loosing performance (compare Section 17.3.5). Eventually, Reed-Solomon implementations can hardly show competitive performance compared to QEnc by construction. Implementations of the update-procedure reach bandwidths between 100 MB/s and 1.6 GB/s on a CPU or up to 2 GB/s on a GPU while QEnc reaches peak bandwidth, which is about 6 GB/s on Sandy Bridge.
- Available Cauchy-Reed-Solomon implementations are either suboptimally vectorized or restricted to predefined matrix sizes. For instance, for a fixed matrix with $n = 5$ ad $k = 3$, they reach about 50 % of the theoretical peak bandwidth [Ste+ 10]. QEnc, in contrast, reaches the full attainable memory bandwidth for all matrix sizes with $n \leqslant 12$, $k \leqslant 12$.

In the end, QEnc demonstrates unrivaled performance at arbitrary matrix sizes, only bound by hard limits of the employed hardware. It offers the greatest possible flexibility by not restricting the matrix size in any way.

A simple consideration reveals some natural limits for encoding performance. Since every code word depends on every data word, it seems plausible that per data word, at least k instructions are

required for creating k code words. In other words, the minimum number of b-bit instructions α necessary per b-bit data word and code word is one. In fact, this assumption is false. It arises from the same misconception which made people think that matrix multiplication has cubic complexity for a long time. Consequently, the Strassen algorithm proves that α is smaller than one, at least for large matrices. For the moment, this shall be ignored; assume $\alpha \approx 1$, and consider the automorphic code without the Strassen optimization. This is actually reasonable and fair since the Strassen improvement speeds up the encoding exactly to the same extent as it reduces α.

With the optimizations from Section 17.3.4, the matrix fill ratio is 0.3. Per b-bit data word, QEnc needs $0.3 \cdot lk$ b-bit instructions for creating k code words (Section 16.7). With $l \geqslant \log_2(n+k)$ this yields $\alpha_{Qenc} \leqslant 0.3 \cdot \log_2(n+k)$. Assuming $n \leqslant 512$, $k \leqslant 512$ this results in $\alpha_{Qenc} \leqslant 3$. Obviously, emulation of ring operations can by no means reach this efficiency. The emulation with a modified Russian peasant algorithm [Kal+ 11] needs up to $2l$ shift and $2l$ *XOR* operations, together with more complex conditional statements than used in the automorphic code. In addition, only certain powers of two are possible values for l. The plain matrix multiplication, for comparison, needs about $2l$ instructions. Furthermore, the instruction throughput is an important aspect. QEnc reaches up to 3.21 instructions per cycle (Section 17.3.3.3) requiring 1.3 – 1.4 assembler instructions per *XOR* operation (in certain cases even down to 1.23 – see Section 17.3.3.3). It executes up to 2.61 *XOR* operations per clock cycle, 3 being the hard limit defined by the CPU architecture. Since the implementation also needs instructions for loads, stores, logic, prefetches, and address calculation, there is almost no margin for further improvements. In total, QEnc needs about 1 cycle per 128-bit data and code word.

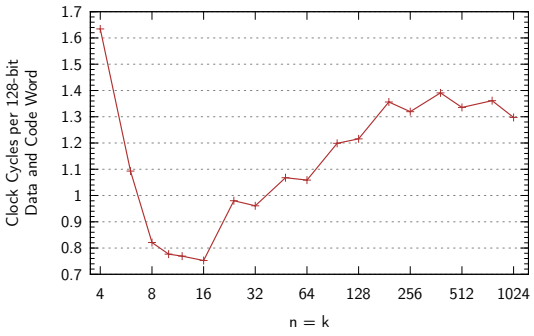


Figure 19.3: Number of Clock Cycles required by QEnc per 128-bit Data and Code Word [XII]

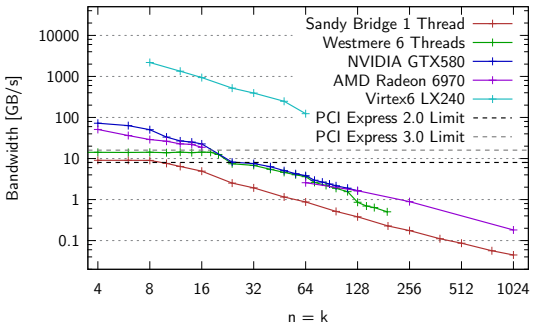


Figure 19.4: Overview of QEnc Encoding Bandwidth on CPU, GPU, and FPGA [II,XII]

The required clock cycles per data and code word from the measurement presented in Fig. 19.3 are in good accordance with the above considerations. Values for small $n = k$ are larger because there the memory bandwidth is the limiting factor and the processor cannot reach its peak instruction throughput. The results are excellent around $n = 12$ where the number of assembler instructions per *XOR* operation is minimal. For larger matrices the computational effort increases as the l dimension becomes larger while for even larger matrices the Strassen algorithm sets in reducing the complexity again. This demonstrates that QEnc hits limits under two perspectives. From the mathematical side, $\alpha = 3$ operations per data and per code word are extremely efficient and to the best knowledge of the author there is no better algorithm. (For large n this becomes even better when the Strassen algorithm sets in.) From the implementation side, with up to 2.61 calculations (not instructions) per cycle, QEnc hits the utmost limitations of the hardware. Taking into account that each data word affects each code word, there is very little room to achieve a factor significantly below 1 clock cycle per data and code word.

This section concludes with Fig. 19.4, which demonstrates the peak encoding bandwidth measured on all hardware platforms: single-threaded CPU, multi-threaded CPU, GPU and FPGA.

The indicated PCI Express limits reveal where the implementations are bandwidth bound and where they can reach their full performance. The figure illustrates that for most situations CPU encoding is sufficient and points out the areas where GPU encoding or the FPGA can deliver a real benefit.

19.2 Conclusions

The original Reed-Solomon code suffers from the fact that computers cannot calculate natively in finite fields and have to employ a slow emulation. The problem can be overcome with two strategies: the parallel Cauchy-Reed-Solomon code encodes with logical operations on bit-vectors; certain codes on finite rings can encode via integer operations. Both operations are well supported on all hardware today. In addition to codes on finite rings from the literature, a new approach is presented in this thesis, which results in the same encoding matrices but generates them faster.

Both codes are implemented via matrix multiplication and achieve peak performance on all tested CPUs and a high fraction of peak performance on GPUs. The matrix multiplication can be sped up by assimilating the encoding-matrix into the source code. Usually, this entrains the problem that only one fixed erasure code is included in the source code. To ensure the greatest possible flexibility, an automorphic encoder is presented, which modifies itself to incorporate every particular encoding-matrix as needed. In addition to the *XOR*-only encoding based on the parallel Cauchy-Reed-Solomon code, an *Add*-only variant based on integral codes is implemented.

The two automorphic variants are realized with low-level assembler optimizations and achieve peak performance on the tested processors. At any time, the implementation is either limited by memory bandwidth or by instruction throughput, which are hard limits set by the architecture. The plain matrix multiplication based code is outperformed by about a factor of 6.5. On a single Sandy Bridge core, the maximum measured encoding bandwidth is above $9\,\mathrm{GB/s}$ and the maximum compute throughput is 3.21 instructions per clock cycle. The algorithm needs less than three operations per code and data word and the implementation is very efficient with an overhead of 1.23 instructions per operation. Finally, QEnc creates about one code word per clock cycle and per data word, which leaves very little room for improvements.

Multi-threaded encoding on the processor achieves the maximum possible aggregate memory bandwidth. Unfortunately, for large input parameters, where the single-thread version is entirely instruction-throughput-bound, the multi-thread variant does not scale completely linearly. Cache and memory effects are two reasons. The four-thread version reaches a speedup of up to 3.3.

Both pure matrix multiplication and automorphic codes are ported to GPUs. The plain matrix multiplication achieves close to peak performance on both GPU and CPU, i. e. its GPU implementation achieves the maximum possible speedup. The automorphic code on the GPU is implemented in OpenCL and is subject to less sophisticated optimizations than the assembler code on the CPU. This explains that the code does not reach the theoretical GPU peak performance. In the end, CPU encoding is memory-bandwidth-limited for a great input parameter range. Since CPU memory bandwidth exceeds PCI Express bandwidth by far, GPU encoding is throttled by the DMA transfer in many applications. For this reason, and due to the inhomogeneous GPU architectures of different vendors, no optimized GPU assembler implementation of the automorphic encoding was created. GPU encoding gets interesting for large datasets and very large input parameters, where PCI Express does no longer pose a bottleneck. One imaginable field of application is rebuilding a full large RAID-array with many disks after multiple drives failed. In contrast, GPUs bring little or no benefit for normal RAID operation as the CPU is already fast enough. The FPGA version provides unrivaled but unfortunately inaccessible performance orders of magnitude faster than even the GPUs. PCI Express limits the FPGA even stronger than the GPU making the FPGA encoder always bandwidth-limited.

The performance of QEnc does not depend on the input parameters n and k but only on available memory bandwidth and maximum instruction throughput, at least for nowadays hardware. In addition to encoding and decoding, also update-codes (differential encoding) are implemented. They achieve the peak bandwidth attainable from memory in any case.

Encoding is possible using either SSE or AVX vector extensions. At the moment, SSE is faster because the instruction level parallelism for logical vector operations with SSE is much higher and overcompensates the smaller vector width. If a new processor was able to process logical AVX instructions at the same rate as SSE, performance would most likely double.

The newest Intel specifications for the Haswell CPU even include a fourth port for logical operations. This opens the possibility for an acceleration by another 33 %, if four 256-bit XOR operations can be executed in parallel – but this remains to be seen.

Part V

Synthetic Benchmarks & Real World Applications

Sources of Images: Anandtech - NVIDIA Launches Tesla K20 & K20X:GK110 Arrives At Last; Tom's Hardware - Intel Xeon E5-2600: Doing Damage With Two Eight-Core CPUs.

Chapter 20

Achievable CPU & GPU Performance

20.1 Overview of Synthetic and Application Benchmarks

Comparing GPUs and CPUs from a synthetic perspective is relatively easy as compute performance ($^{\text{GFlop}}/_{\text{s}}$) or bandwidth ($^{\text{GB}}/_{\text{s}}$) can be measured. Using the correct access patterns, such benchmarks usually come very close to the theoretical peak performance. An absolute measure of this kind is hardly available in application benchmarks. Instead, the application runtime allows for calculating the GPU speedup.

However, the pure GPU speedup itself has little validity. The GPU is the faster processor by design, which has to be taken into account. Usually, a GPU implementation achieves a smaller fraction of the GPU's peak performance than its CPU counterpart. To set the GPU speedup in relation to the highest possible speedup considering the peak performance ratio, a **speedup-index** γ is defined. With a_g and a_c the achieved GPU and CPU performance and p_g and p_c the respective peak performance, the index is defined as:

$$\gamma = \frac{a_g/p_g}{a_c/p_c} \tag{20.1}$$

$$= \frac{a_g/a_c}{p_g/p_c}. \tag{20.2}$$

Equation 20.2 allows for using different measures for the achieved and for the peak performance, e. g. achieved performance is often measured in one over seconds which is obviously impossible for the peak performance. Depending on the bottleneck of the application, the peak performance is either defined as peak memory bandwidth or peak compute performance.

An index significantly above 100 % is highly unlikely and implies that the GPU version scales even better than the CPU version. Such a result mostly indicates a poor CPU implementation. Depending on the boundary conditions, close to 100 % may be unattainable and a lower index is optimal. One simply tries to come as close as possible to 100 %. In the majority of the cases, a value between 60 % and 100 % is optimal.

Table 20.1 lists the results and the speedup-indexes on a representative selection of architectures for all problems examined in this thesis. Grouped results are taken on comparable GPUs and CPUs available and state-of-the-art at the same time. By and large, most measurements reveal a speedup of about three for a single GPU, for multiple GPUs it is larger. However, this may not lead to the conclusion that GPUs scale equally well on both synthetic benchmarks and applications. The synthetic measurements mostly base on double precision calculations while many applications in the table stick to single precision, where often the possible GPU speedup is

Benchmark	Type	Hardware	Performance / Time	% of peak	Speed-up	γ [%]
(old) HLT Tracker	Single	Nehalem 4C 3 GHz	1122 ms			
		GTX285 + CPU	312 ms		3.60	53
(new) HLT Tracker	Single	2×Magny-Cours 2.2 GHz	495 ms			
		GTX580 + CPU	155 ms		3.19	85
Track Merger	Single	Westmere 6C 4 GHz	65 ms			
		GTX580 + CPU	60 ms		1.10	13
Track Fit	Single	Westmere 6C 4 GHz	16.8 ms			
		GTX580	6.8 ms		2.47	29
SGEMM (Kernel)	Single	2×Magny-Cours 2.1 GHz	360 GFlop/s	89.3		
		5870	2014 GFlop/s[1]	74.0	5.59	83
		6970	1844 GFlop/s	68.2	5.12	76
DGEMM (Kernel)	Double	2×Magny-Cours 2.1 GHz	180 GFlop/s	89.3		
		5870	494 GFlop/s	90.8	2.74	102
		6970	624 GFlop/s	92.3	3.47	103
		7970	805 GFlop/s	84.4	4.47	95
DGEMM (System)	Double	2×Magny-Cours 2.1 GHz	180 GFlop/s	89.3		
		5870 + CPU	623.5 GFlop/s	83.6	3.46	94
		3×5870 + CPU	1435 GFlop/s	78.3	7.98	87
		2×6990	2292 GFlop/s	89.9	12.73	104
		2×S10000	2923 GFlop/s	79.8	16.24	89
One-Node HPL	Double	2×Magny-Cours 2.1 GHz	174.6 GFlop/s	86.6		
		5870 + CPU	563.2 GFlop/s	75.5	3.23	87
		3×5870 + CPU	1114 GFlop/s	60.7	6.38	70
		2×6990 + CPU	2007 GFlop/s	72.4	11.49	84
		2×S10000 + CPU	2679 GFlop/s	73.1	15.34	84
Erasure Codes (small n)	32-bit logical	Westmere 6 · 3.8 GHz	14.3 GB/s	74.7		
		GTX580	72.5 GB/s	75.3	5.32	102
		6970	51.1 GB/s	58.0	4.10	78
		Virtex 6 LX240 FPGA	2187.0 GB/s		152.94	
Erasure Codes (large n)	32-bit logical	Sandy Bridge 1 · 3.7 GHz	251.0 GAOp/s[2]			
		Westmere 6 · 3.8 GHz	807.0 GAOp/s[2]			
		GTX580	908.4 GAOp/s		1.13	19
		6970	1024 GAOp/s		1.27	26

Table 20.1: GPU / CPU Performance Summary and Speedup-Indexes

greater. Harking back to the classifications of benchmarks from the very beginning (Chapter 1), the examined problems are handled one after another.

SGEMM and DGEMM belong to the synthetic benchmarks for compute performance; failure erasure coding for small n is a synthetic benchmark as well but it measures memory bandwidth. All these benchmarks reach very close to peak performance. In particular, the erasure code efficiency looks worse than it actually is: The achieved bandwidth on the CPU is 75.3 % of the specified peak bandwidth, but even pure memory bandwidth benchmarks cannot attain higher

[1] The performance stated in the table for the 5870 was measured by N. Nakasato [Nak 10].
[2] The matrix-multiply performance as defined in Section 17.1 is the number of instructions a plain matrix multiplication would require. Optimized implementations can thus exceed the specified peak performance.

results either. A comparison with Table 17.23 reveals that the encoding utilizes the accessible bandwidth to quite exactly 100 %. The GPU encoding is limited by the DMA transfer anyway. Therefore, not much effort was invested in order to increase the utilized GPU memory bandwidth (which is already quite acceptable). Standalone DGEMM comes closest to peak performance everywhere, hence $\gamma \approx 100\,\%$. SGEMM on GPU is slightly less efficient due to less tuning and since it requires more memory bandwidth.

HPL is based primarily on DGEMM and hence a semi-synthetic benchmark. It has been shown that processing time for all other HPL tasks can be hidden behind the DGEMM time. For this reason, HPL performance is close to DGEMM performance. In summary, all synthetic and semi-synthetic benchmark results come close to peak performance, both on GPU and on CPU, regardless of whether double or single precision is used. Hence, for all these benchmarks, the speedup-index is quite good, to be precise between 70 % and 104 %.

HLT tracker and merger belong to the application benchmark category. For the GPU merger, the DMA transfer takes almost as long as the CPU needs for the entire processing. Thus, the actual performance of the GPU track fit is irrelevant; the GPU merger can never be much faster than the CPU version. This explains the small speedup and speedup-index. Currently, the GPU track fit itself is more than twice as fast as the CPU one. For the above mentioned reason, very little effort has been spent for the GPU track fit – it was considered a waste of time. It is highly probable that the GPU version (of the fit only) can be sped up, at least by the same order of magnitude as the tracker.

In contrast to the track merger, the slice tracker is not limited by PCI Express. It shows a nice speedup and its speedup-index is in the same order of magnitude as for synthetic benchmarks. Table 20.1 lists that the speedup-index for the new tracker version has even increased and is now almost optimal. However, the index increase is misleading. The new version is compared to AMD compute nodes, which have been installed in the HLT recently. The old version is compared to an Intel CPU, which reaches a higher efficiency in slice tracking even though the absolute performance is better on AMD. So, the speedup-index depends on the CPU which is used for comparison. Still, as stated before, applications have a hard time achieving 100 % speedup-index and 53 % – 85 % are very good. This is an interesting observation in combination with the analysis from Section 6.4.4, which concludes that the tracker triggers many warp-serializations and its GPU-utilization is by far not optimal. Even though the tracking algorithm is really fast compared to other approaches, its implementation cannot attain the full potential neither of GPUs nor of CPUs. This conclusion is in accordance with investigations about tracking accelerations by explicit SSE programming [Kre 09]. Hence, the tracker is a counter-example to the naive thought that in general, complicated algorithms can be realized better for CPUs than for GPUs. Of course, it does not proof the opposite as well.

Failure erasure coding is implemented in two versions. The first one is based on naive matrix multiplication. It is compute-bound for large n and all conclusions made for SGEMM are valid without restrictions. Since the second version (*XOR*-only / *Add*-only code) is superior, only this one is discussed further. A GPU speedup exists but it is much smaller than for all other benchmarks. The reason is that the CPU version employs manually optimized assembler code. A competitive GPU implementation would have to be written in GPU assembler, too. Such a version has not yet been realized. The necessary effort is tremendous, and every GPU architecture would require its own implementation. (The CPU version solely requires an x86 instruction set. Only the parameters for the cache sizes need to be modified.) In contrast to HPL, where the relatively small DGEMM kernel is the critical task, encoding requires much more assembler code. On top of that, many GPU compilers encounter problems when compiling source code of that size. At small matrix sizes, where the GPU is significantly faster, PCI Express poses an insuperable limit. Eventually, in most cases the CPU is fast enough for the encoding anyway.

The FPGA encoder easily outperforms the GPUs but all just-mentioned restrictions hold for it as well – and they deteriorate the performance even stronger. Considering the FPGA on its own and ignoring that it is simply impossible to get the data in and out the FPGA at that speed, it peaks at few terabytes per second.

20.2 Summary

In summary, it is possible to utilize a great fraction of the theoretical peak performance of GPUs in synthetic benchmarks. It is much harder and often not possible to obtain a speedup in the order of the theoretical value for real world-applications. Still, for most applications the improvement is significant and a similar fraction of the peak performance can be used on CPU and GPU. In most but the special cases (merger and encoding), a speedup-index of about 75 % or more is achieved. The PCI Express transfer can pose limits, such as for the merger. It must be noted that usually at least some CPU cores are required for GPU pre- and postprocessing or scheduling. These tasks can usually be pipelined, regardless of whether the problem is synthetic like HPL or an application like the tracker. In all examined programs, pipelines and asynchronous DMA transfer are absolutely necessary to achieve good performance. Thus, speaking of a GPU speedup is a bit unfair as the processor contributes its part as well. However, this contribution is mostly restricted to few cores so that a big fraction of the CPU is still available for other tasks.

Concluding on the analyzed problems individually, the GPU tracker demonstrates that it is possible to employ GPUs for complex tasks in a production environment. The tracking in the HLT is sped up by about a factor three compared to a CPU, the cost benefit is even larger. The GPU tracker proved its stability in the heavy ion phases from 2010 to 2013. In contrast to the tracker, the example of the merger shows that problems exist which by definition cannot be solved well on a GPU. The reason lies in the data transfer. Vectorization of the Kalman filter, which makes up for most of the merger execution time, is very well possible [Gor$^+$ 08] – as well as is a GPU based Kalman filter [Bac 09].

CALDGEMM is among the fastest realizations of matrix multiplication. In contrast to other implementations, it achieves its outstanding performance irrespective of the transposition parameters and makes close to 100 % of the kernel performance available in the system. The improved HPL implementation demonstrates that under certain conditions it is possible to reach the GPU peak performance also for non-synthetic benchmarks. It further proves that limited GPU memory is no general deficiency but can be overcome e. g. by the tiling approach. Up to the June 2012 Top500 list, no other GPU cluster could reach a comparable efficiency with respect to theoretical peak performance. With HPL-GPU, the LOEWE-CSC cluster ranked place 22 in the November 2010 Top500 list and the Sanam cluster ranked second in the November 2012 Green500 list.

Last but not least, the automorphic failure erasure codes, using either logical or integer operations, show the potential of CPU vector extensions. They are an example of a problem which can benefit from GPUs in some cases, but where mostly an optimized CPU implementation is sufficient, or even faster. This problem might benefit much from new architectures like fused CPU / GPU chips such as the AMD Fusion [Bra$^+$ 12], where the GPU can access the host memory directly and PCI Express is no bottleneck anymore. The presented QEnc library reaches the maximum attainable bandwidth of up to 15 $^{\mathrm{GB}}/_{\mathrm{s}}$ for small matrices while other CRS implementations [Ste$^+$ 10] are limited to around 50 % and other approaches for optimized Reed-Solomon codes achieve even less.

20.3 Conclusions & Comments

GPUs have proven to be excellent accelerators for all examined problems that are not bandwidth-bound. Even further, GPUs excel in many bandwidth-bound problems not mentioned here such

as lattice QCD [Bac+ 11 II], where the problem, or an isolated part of it, fits in GPU memory. In that case, the superior GPU memory bandwidth can play its strength. An insolvable problem appears only when the PCI Express bandwidth limit cannot be overcome because data transferred via PCI Express are only accessed few times and then replaced by new data for the next task (like the HLT merger). In some cases, theoretically doable optimizations for the GPU are hard to realize in a universal and practical way due to missing software infrastructure, rapid hardware development, and large differences in GPU architectures (automorphic GPU encoding is an example). Then, implementations can be so laborious and non-portable that they do not pay off in the long run.

In general, the question arises whether the time invested for GPU optimizations is worth it. The answer, if there is one, depends to a large extent on the particular problem and on the employed computers. Naturally, if an algorithm runs for a macroscopic time on large-scale clusters for several millions of dollars, even small optimizations are worthwhile. In this case, besides the financial perspective, it is even possible that the available compute time is insufficient for unoptimized code. On smaller systems the duration of the development is more important. It is cheaper to buy a second computer than to employ a programmer for half a year. Consider the GPU Linpack benchmark as one example: the development time required for the actual GPU kernel was much less than the total development time. Of course, the CPU part must be modified as well to make use of the GPU. The modifications involve two tasks: DMA transfer management and scheduling. The DMA management is purely GPU related overhead. On top of that, GPU-based programs complicate the scheduling since processors with different capabilities must be synchronized; but most multi-threaded high-performance codes need some kind of scheduling anyway – irrespective of the GPU. In summary, the effort for the actual GPU part of HPL-GPU is significant but not the single dominant contribution to total development time. By and large, GPU optimizations are usually rewarding if algorithms have isolated computational hot spots which can be well processed by GPUs. In such cases, good performance gains can be achieved with only few small and focused modifications. Speaking generally, there are many applications with tens of thousands of lines of code, but less then 1 % of them are executed during 95 % of the runtime. Moving this part to a GPU can yield significant gains at reasonable effort.

An example that does not possess an isolated hot spot is the ALICE HLT tracker. Still, GPU adaptation is possible. In this case, a large fraction of the tracker code runs on the GPU. Naturally, the individual tracking steps have not been tuned as comprehensively as the DGEMM kernel for HPL. Still, development time of the GPU parts for both problems are comparable because the tracker simply contains much more code. However, it must be noted that the tracker is a full scientific application while HPL merely solves a system of linear equations. While in general, fast solvers for linear equations are required in plenty of applications, the matrix sizes typically used in HPL have grown beyond any reasonable scale; the implication of the Linpack score on the processing performance for small matrices is diminishing. On top of that, the benefit of GPU tracking cannot be quantified simply by the achieved speedup. The ALICE HLT GPU tracker currently uses two to three CPU cores out of 24 cores in a compute node. It is therefore not unreasonable to claim that the processor is still available during tracking. Various HLT tasks such as track merging, cluster transformation, vertexing, and so forth require these CPU resources anyway. In terms of tracking performance, a single GPU is equivalent or even superior to both Magny-Cours processors. Hence, plugging a GPU in a compute node actually saves the cost of an additional full node and of the additional infrastructure required for more nodes. Since the GPUs cost only a tiny fraction of the entire HLT facility, the extra hardware costs for tracking are negligible. In summary, HPL-GPU achieves a larger speedup due to more sophisticated optimizations to a distinguished computational hot spot. The tracker acceleration is a bit smaller even though development time has been similar. Still, its speedup factor of 3.19 is enormous and certainly justifies the invested effort. The HLT approach with tracking on GPUs and other tasks on CPUs is probably a universal model because most clusters do not exclusively run a single but various programs – all of them more or less suited for GPUs. Hardware can

be used best by porting those tasks to GPUs which can benefit from parallel architectures and whose optimization does not require changing millions of lines of code. Problems unsuited for GPUs can occupy the processors at the same time. Complex applications can make use of GPUs by internally splitting the problem in GPU and CPU tasks.

For the future it is evident that scientific and other HPC applications cannot ignore technological hardware changes. With more and more CPU cores in the system, trivial parallelization gets more and more inefficient. In addition, auto-vectorization features are limited to certain types of loops [Kre+ 11]. In order not to neglect a large fraction of the available performance, proper vectorization is unavoidable. This is a fact, and it is true for all modern processors. Hence, programming paradigms must comply with these features irrespective of the target platform. In the end, CPU and GPU architectures are converging: the AMD Fusion chip [Bra+ 12] is one real example, NVIDIA has plans to include "latency cores" [Dal 10] in their chips, and Intel had plans for a GPU based on a CPU instruction set [Int 08], which was later modified to an HPC-only product [Int 10]. All these developments aim at a heterogeneous processor with fast scalar cores and broad vector cores accessing a unified host memory. The only such chip available today, the AMD Fusion, shows good compute performance but suffers from the fact that host memory is about one order of magnitude slower than onboard GPU memory. This is a price one has to pay for in exchange having access to a large memory; and it is currently a general deficiency of such approaches. It is not yet clear how this bottleneck can be overcome; an explicit big L4 DRAM cache is a possible solution. No matter what, the single-threaded scalar programming concept is outdated, not to say antiquated, and programmers must adapt in order to programm CPUs, GPUs, and what else the future holds in an efficient way.

New programming languages appear that encapsulate GPU kernels in high-level code and take care of the data transfer internally. Microsoft C++ AMP [MS], OpenACC, and HMPP are some examples. The future question is not whether to use GPUs or not to use GPUs but rather to which extent low-level optimizations are desired – or required. This determines programming language and concept. GPU assembler is the lowest possible level; current programming models like CUDA or OpenCL are still close to the hardware. Their high-level counterparts are yet under development with first pragma based approaches like OpenACC currently hitting the market. Many people proclaimed a soon death to C++ suggesting it will be replaced by managed languages like Java. They were proven wrong. In the same way, it is very probable that multiple programming models will coexist, the current ones becoming the lower-level ones possibly allowing for better and more focused optimization. GPUs might be an intermediate step but the programming concepts may survive even for fused processors and new graphics cards, e. g. the AMD Fusion is programmed in OpenCL. Therefore, code written in current languages (except for GPU assembler) will likely work on future hardware as well (with moderate modifications).

The final question that comes up is how well code optimized for a particular processor performs on future hardware. There is no universal answer to this question. In general, the closer optimizations are to the hardware the more likely it is that the code must be optimized again in the future. However, it is often possible to abstract hardware characteristics and make implementations offer many parameters that can be tuned for new architectures with little effort. The ALICE tracker is one such example. Most optimizations for the Fermi could be realized by changing existing parameters. There is one modification that required larger code modifications: the multi-threaded pipeline. However, in that case the necessity arises simply because the GPUs have become faster but not because of changes in the architecture.

In the end, all problems analyzed in this thesis could benefit greatly from modern CPU vector extensions or from GPUs; often a combination of both together with asynchronous processing is required. It has been shown that not every problem is suited for every type of processor or optimization approach respectively. Instead, the right combination must be found for each particular task. Having accomplished this, the improvements are usually well worth the effort.

Appendix A

GPU Architectures in Detail

A.1 NVIDIA

A.1.1 GeForce

The GeForce GT200b chip belongs to a previous NVIDIA generation and is used in the GTX285 and GTX295 cards, of which the latter one is a dual-GPU board with slightly reduced clocks. Neither an L1-cache nor VLIW-shaders are employed. It offers eight single precision ALUs per multiprocessor and its warp size is 32. The chip has 30 multiprocessors resulting in 240 shaders. Double precision calculations can be done by combining all eight ALUs to one double precision ALU reducing the performance by a factor of eight.

A.1.2 Fermi

The Fermi is the successor to the GT200b offering approximately twice as many shaders. For a long time it has been the only GPU offering a general-purpose L1 cache. The consumer-grade version suffers from the same penalty of factor eight for double precision. The GTX480 was the first consumer variant, which was later replaced by the improved GTX580. The professional-grade (M2070) corresponds to the GTX480. It has special double precision ALUs, which reduce the penalty to a factor of two and supports ECC memory as well. It was replaced by the M2090 corresponding to the GTX580. Compared to the GTX285, the number of multiprocessors is roughly halved but the ALU count rose to 32 per multiprocessor. The warp size remains at 32. This reduces the number of concurrent threads required for optimal performance, in comparison to the old design.

The successor to the Fermi is the Kepler generation, with the consumer card GTX680 and the professional card K20x. They have been released only recently and are not used in this thesis. Table A.1 lists all the NVIDIA GPU types that are used.

A.2 AMD

A.2.1 Cypress

The Cypress chip (5000 series) is used in the 5870, V7800, and 5970 cards, the latter one being a dual-GPU board. As the old NVIDIA family, Cypress does not offer a general-purpose L1 cache. Cypress consists of up to 20 multiprocessors with 80 shaders each. The shaders are organized as 5-D VLIW shaders. Within each VLIW shader, four ALUs can be combined to one double precision ALU resulting in a double precision penalty of five.

GPU	GTX285	GTX295	GTX480	GTX580	GTX680	M2070
GPU Chips	1	2	1	1	1	1
Multiprocessors	30	30	15	16	8	14
Shaders / MP	8	8	32	32	192	32
Total Shaders	240	480	480	512	1536	448
Memory	1 GB	2 · 896 MB	1.5 GB	1.5 GB	2 GB	6 GB
Core Clock [MHz]	648	550	700	772	1006	700
Shader Clock [MHz]	1476	1350	1401	1544	1006	1150
Memory Clock [MHz]	1242[1]	1000[1]	924	1002	1502	750
Peak (Single) [GFlop/s]	713.8	1296	1344	1581	3090	1030
Peak (Double) [GFlop/s]	88.6	162	168	197.6	128.75	515.2
Mem. Interface [bits]	512	448	384	384	256	384
Mem. Bandwidth [GB/s]	159	223.8	177.4	192.4	192.3	144
Prof.-Grade	No	No	No	No	No	Yes
Cooling	Active	Active	Active	Active	Active	Passive

Table A.1: Overview of the NVIDIA GPUs used throughout this Thesis

A.2.2 Cayman (Northern Islands)

The Cayman chip of the 6000 series is the successor to Cypress. The 5D-shader design is changed to 4D reducing the double precision penalty to four. It has 24 multiprocessors with 64 shaders each.

A.2.3 Tahiti (Southern Islands / Graphics Core Next)

With the Tahiti (7000 series) AMD abandoned the VLIW principle. Like the Fermi, the Tahiti offers general-purpose caches. The double precision penalty remains at four.

Table A.2 lists all the AMD GPU types that are used.

GPU	5870	5970	6970	6990	7970	V7800	S10000
GPU Chips	1	2	1	2	1	1	2
Multiprocessors	20	2 · 20	24	2 · 24	32	18	2 · 28
Shaders / MP	80	80	64	64	64	80	64
Total Shaders	1600	3200	1536	3072	2048	1440	3584
Memory	1 GB	2 · 1 GB	2 GB	2 · 2 GB	3 GB	2 GB	2 · 3 GB
Clock [MHz]	850	725	880	830	925	725	950
Memory Clock [MHz]	1200	1000	1375	1250	1375	1000	1250
Peak (Single) [GFlop/s]	2720	4640	2703	5099	3789	2088	6810
Peak (Double) [GFlop/s]	544	928	675.8	1275	947.2	417.6	1702.4
Mem. Interface [bits]	256	256	256	256	384	256	384
Mem. Bandwidth [GB/s]	153.6	256	176	320	264	128	480
Prof.-Grade	No	No	No	No	No	Yes	Yes
Cooling	Active	Active	Active	Active	Active	Active	Active

Table A.2: Overview of the AMD GPUs used throughout this Thesis

[1] The GPU uses GDDR3, which transfers only half the amount of data per clock cycle compared to GDDR5, which is used by other GPUs.

Appendix B

AMD Intermediate Language/ISA Assembler

AMD provides two low-level programming languages: the **I**ntermediate **L**anguage (IL) [Adv 10 II] and the **I**nstruction **S**et **A**rchitecture (ISA) [Adv 10 III] language. High level code, such as for OpenCL [Khr] or Brook+ [Adv 09], is compiled to the architecture independent intermediate assembler language (IL). This IL code is then compiled to ISA assembler code respecting the target architecture. Contrary to NVIDIA, which only enables the user to view the intermediate PTX code produced when compiling CUDA files, the AMD SDK allows for writing explicit IL or ISA code. The 7000 series (also called Tahiti or Graphics Core Next) introduced a new intermediate language called FSAIL. This series is still compatible to the old IL, which is marked as deprecated today, however.

To keep the code portable, only IL assembler code is used in this thesis. However, for optimizing the IL code, a deep inspection of the derived ISA code is required. It is a common practice to have a particular ISA code in mind and alter the IL code such that the compiler produces the desired ISA code. This appendix does not provide a full documentation of the assembler languages but is meant for a reader already familiar with assembler programming. It only points out the aspects of the GPU languages needed to understand the kernels shown in this thesis. Only *float4* and *double2* vectors are used. Other types are handled analogously.

B.1 IL

All AMD stream GPUs (5000 and 6000 series) use 128-bit registers. No distinction is made between the data-types stored in the registers. A register can store four single precision floats, two double precision floats, or four 32-bit integers. The single components of the registers can be accessed using the "**.x**", "**.y**", "**.z**", and "**.w**" suffixes, for 64-bit types "**.xy**" and "**.zw**" are used. All operations can be performed on a single component of the register or on the entire vector. During a read, the vector-components can be shuffled using an "**.abcd**" suffix, where a, b, c, and d can be:

- **x**/**y**/**z**/**w** The respective component of the vector.
- **0**/**1** The constant 0 or 1.
- **_** Omit this component in the target register.

As an example: "a.xy_w = b.yx01" stands for "a.x = b.y", "a.y = b.x", "a.w = 1".

IL uses instructions with up to four operands. The first operand is the target. Registers are denoted by ri where i is the index. Constants (except for 0 and 1) must be declared as 128-bit constants explicitly and are accessed the same way the registers are. Constants are usually identified by li. The following other definitions are important:

- **dcl_input_position_interp**(**mode**) register: Declare a register containing the position of the thread in the grid.[1]
- **dcl_resource_id**(i)_type(**dim**, **normalization**)_fmtx(**format**)_..._fmtw(**format**): Declare an input resource buffer; usually each entry consists of 128 bits.[2]
- **dcl_cb cb**$i[j]$: Declare the constant buffer i of size j.

While-loops are enclosed by "**whileloop**" and "**endloop**" clauses. A "**break_logicalnz** register" statement exits the loop depending on the value in the register. The input buffers are accessed by the "**sample_resource**(i)**_sampler**(i) r1, r0.xy" instruction, which reads from location r0.x, r0.y in buffer i to register r1. The global buffer can be accessed with "**g**[addr]". Table B.1 shows some examples.

Instruction					Explanation
whileloop					Start of while loop
mad	r1.xy_w,	r3.1yzw,	r4.zwxy,	r2.xy00	Single Precision Multiply-Add: r1.x = r2.x + r4.z, r1.y = r2.y + r3.y * r4.w, r1.w = r3.w * r4.y
add	r5.x,	r5.x,	r5.1		Increment r5.x by 1
ge	r5.y,	r5.x,	cb0[0].x		Compare r5.x to element 0 in constant buffer 0
break_logicalnz	r5.y				Exit loop if above comparison (stored in r5.y) was true
endloop					End of while loop

Table B.1: IL Assembler Instruction Examples

B.2 ISA

Consider the VLIW shaders introduced in Section 3.1. (For instance, the Cypress chip of the 5870 has 5D-shaders). Each VLIW instruction for a 5D-shader encapsulates five single instructions for the five X, Y, Z, W, and T shaders. Five sequential instructions of the IL code are combined in one VLIW instruction as long as they are independent. Clearly, the use of vector instructions for the four .x to .w components of the 128-bit registers simplifies this work for the compiler. The T shader can execute an additional instruction, e. g. in the example in Table B.1 the counter increment can be performed by the T shader in parallel to the fused-multiply-add.

Double precision operations are performed by the X to W shaders which leaves the T shader available. Thus, the Cypress can perform a single and a double precision instruction per clock cycle. In the 6000 series chip, the T shader is no longer present. Hence, the 128-bit registers map directly to the 4D-shaders and, in addition, during double precision calculation no shader idles anymore.

The ISA code is separated into clauses. There are clauses for different execution units. A clause is prefixed with "ALU:" if it is executed by the ALUs or with "TEX:" for the texture engine, etc. On the one hand, the context needs to be changed between the clauses[3], thus, intermixing calculation

[1] The parameters define the interpolation mode (e. g. linear) and whether the coordinates are normalized to the unity interval.

[2] Dim is the dimension of the buffer. Values read from the buffer can be interpolated according to the normalization parameter. The format parameters define the type for each 32-bit subset.

[3] A clause switch causes up to 40 cycles of latency.

and texture fetches can decrease performance. On the other hand, ALU and TEX clauses can be executed in parallel.

Table B.2 shows an example for a VLIW ISA instruction. The corresponding IL code is:

- mad r0, r1, r2, r0
- add r3.x, r3.x, r3.1

Shader	Instruction				
X:	MULADD	R0.x,	R1.x,	R2.x,	R0.x
Y:	MULADD	R0.y,	R1.y,	R2.y,	R0.y
Z:	MULADD	R0.z,	R1.z,	R2.z,	R0.z
W:	MULADD	R0.w,	R1.w,	R2.w,	R0.w
T:	ADD	R3.x,	R3.x,	1.0f	

Table B.2: ISA Assembler Example VLIW Instruction

Table B.3 shows a similar example with a double precision FMA instruction executed by four shaders jointly. (The register R123 is an internal dummy register inserted by the compiler.) The IL code is:

- dmad r0.xy, r1.xy, r2.xy, r0.xy
- add r3.x, r3.x, r3.1

Shader	Instruction				
X:	FMA_64	R0.x,	R1.y,	R2.y,	R0.y
Y:	FMA_64	R0.y,	R1.y,	R2.y,	R0.y
Z:	FMA_64	R123.z,	R1.y,	R2.y,	R0.y
W:	FMA_64	R123.w,	R1.x,	R2.x,	R0.x
T:	ADD	R3.x,	R3.x,	1.0f	

Table B.3: Double Precision FMA ISA Assembler Example

Appendix C

Specifications & Definitions

This appendix is meant as a summary of specifications and definitions which the reader is not automatically assumed to be familiar with. Still, the information is very brief and more details can be found in the references given.

C.1 MPI Threading

The **M**essage **P**assing **I**nterface (MPI) [MPI] is a common library for handling communication in cluster-computation. The specification defines multiple levels of support for threaded applications:

- **MPI No Threading:** The application using the library must not be threaded at all.
- **MPI Funneled Threading:** The application may be threaded but only the thread that invokes *MPI_Init* is allowed to call MPI routines.
- **MPI Serialized Threading:** The application may be threaded. However, the MPI library is not reentrant. Thus, the application must ensure that only one single MPI routine is running at a time.
- **MPI Multiple Threading:** The MPI library is completely thread-safe.

C.2 Matrix Representations

Consider an $m \times n$ matrix M. Let p be a pointer to the matrix. Besides special cases, there are two common ways to represent the matrix in memory:

1. **row-major**: Rows are stored in consecutive memory. The matrix entries are accessed via:
$$M_{ij} = p[i * n + j].$$
2. **column-major**: Columns are store in consecutive memory. Access is carried out via:
$$M_{ij} = p[i + j * m].$$

Usually, a generalization of the above is employed. A **leading dimension** LD_M of the matrix is introduced ($LD_M \geqslant n$ or $LD_M \geqslant m$ respectively). Entries are accessed via

$$M_{ij} = p[i * LD_M + j] \quad \text{and} \quad M_{ij} = p[i + j * LD_M]$$

respectively. This allows for easy access to submatrices.

C.3 Huge Pages

The size of memory pages for the virtual memory of x86 processors has traditionally been 4 KB. This size was chosen when computers had only a system memory capacity of several megabytes. Still today, it works well for managing lots of small memory allocation requests usually used in object oriented programming. However, for scientific applications that work on a single huge dataset and do not need to allocate and free memory in between, 4 KB pages introduce a tremendous overhead. (For instance, a dataset of 100 GB consists of 25 million pages and takes about a minute to allocate.) The processor TLB is not designed for accessing such an enormous number of pages and can easily become a bottleneck.

Modern processors and operating systems offer support for huge pages (Linux) (or large pages (Windows) respectively). In both cases, the increased page size is 2 MB or more. Huge pages cannot be allocated via a simple *malloc* command but instead, there is special support inside the operating system. On Linux a huge page pool must be reserved beforehand, on Windows administrator privileges are needed. With Linux kernel 2.6.38 and later, the operating system automatically stores large memory chunks in huge pages. (This is called transparent or anonymous huge pages.)

Huge pages bring no benefit for most applications. Programs that require very little memory would allocate at least 2 MB each if they used huge pages. In contrast, applications that allocate a huge amount of memory in the beginning and then work on this memory for quite some time can greatly benefit. At several points throughout this thesis, huge pages are used. A special remark points this out each time.

C.4 *LU*-Factorization

Every regular square matrix M can be written as a product

$$M = PLU \qquad U = \begin{pmatrix} * & \cdots & * \\ 0 & \ddots & \vdots \\ 0 & 0 & * \end{pmatrix} \quad L = \begin{pmatrix} 1 & 0 & 0 \\ * & \ddots & 0 \\ * & * & 1 \end{pmatrix}$$

with an upper triangular matrix U, a lower triangular matrix L with diagonal entries 1, and a permutation matrix P. When solving a system of linear equations $Mx = y$, the permutation is usually applied to the right hand side directly: $LUx = Py$. The solution vector x is obtained by solving $Lb = Py$ and $Ux = b$ via backward substitution. The factorization algorithm is usually implemented by applying Gaussian elimination to the matrix $A = (M|y)$. Mathematically, Gaussian elimination multiplies the system with $L^{-1}P$ from the left yielding $L^{-1}PA = (L^{-1}PM|L^{-1}Py) = (L^{-1}PPLU|L^{-1}Py) = (U|b)$ since $PP = \mathbb{1}$ and b is defined by $Lb = Py$. During the process, the lower left part of the former A is zeroed out. This creates space to store the L matrix (without the ones on the diagonal and the zeroes above). The LU-factorization thus carries out the transformation:

$$M = (A|x) \Longrightarrow \left(L \diagdown^{U} \middle| L^{-1}Py\right) = \left(L \diagdown^{U} \middle| b\right).$$

See [Pre+ 92] for more details.

C.5 Interleaved Memory

On NUMA systems, applications running on one CPU core usually allocate memory local to that core. If all CPU cores in the system jointly work on one memory segment, the NUMA

architecture cannot provide equal memory bandwidth for all cores. In that case, an interleaved memory allocation can be better, where the memory pages are distributed among all NUMA nodes in a round-robin fashion.

C.6 Field Programmable Gate Arrays

Field **P**rogrammable **G**ate **A**rrays (FPGAs) are integrated circuits whose behavior can be configured by the user. To put it in a nutshell, in a circuit design driven by a clock, the design describes how the signal on the output pins depends on the input signal and on the internal state as well as how the internal state is modified every clock cycle. The FPGA is internally build by many fixed function units (like FIFOs and shift registers) and it offers a huge set of registers and LUTs (lookup tables). Registers store the internal state. LUTs are small cells with few in- and outputs whose input to output mapping is configurable. In addition, the FPGA has a large route grid, which allows to connect inputs and outputs of arbitrary components. The behavior of the FPGA is determined by the LUT mappings and by the routes. Nowadays, FPGAs have become more complex. They have block RAMs, PCI Express interfaces, DSPs, floating point dividers, and even integrated PowerPC cores.

Hardware description languages like VHDL or Verilog formally specify structure and operation of electrical circuits and are used to program FPGAs. The compiler compiles the source code and creates the firmware, which is then flashed to the FPGA. This task consists of synthesis (comprehending the code and mapping it to generalized units like registers, LUTs, and logic gates), placement (physically placing the units on the FPGA, i. e. deciding which FPGA register stores which signal, etc.), and routing (configuring the route network to provide the required connections). For simplifying the programming for the user, enabling code reuse, and for supporting the compiler in interpreting, designs are usually composed of multiple entities encapsulating subcomponents (in a way similar to classes in C++).

The frequency the FPGA operates at is not fixed but it depends on the configuration. The compiler computes the maximum operation frequency out of known setup and hold times of registers, delay of logic gates, length of routes and electrical signal speed, etc. It is possible to pass timing constraints to placer and router, which they will try to meet while mapping the design to the device. This allows one to specify a desired design frequency.

FPGA units such as registers and LUTs are grouped in slices. Larger and more expensive FPGAs offer more slices. This means that they can carry a larger design but generally cannot run faster. FPGAs of the same size are available in multiple speed grades declaring how fast the FPGA can perform certain tasks – just like there are CPUs identical in design, which can operate at different frequencies. All measurements in Section 18.4 are taken with an FPGA of speed grade 2, which is the medium version. As an example, the $n = k = 48$ design which is listed with 556 MHz in Table 18.5 can be run at 618 MHz by the faster FPGA version with speed grade 3.

The functionality of an FPGA design can be verified by a behavioral simulation. Fig. C.1 presents such a simulation for the FPGA failure erasure coding introduced in Section 18.3. The simulation design connects an encoding entity with a decoding entity and checks whether the final output matches the input. After a certain time, the input data has propagated through the pipeline and the simulation shows that the design operates correctly.

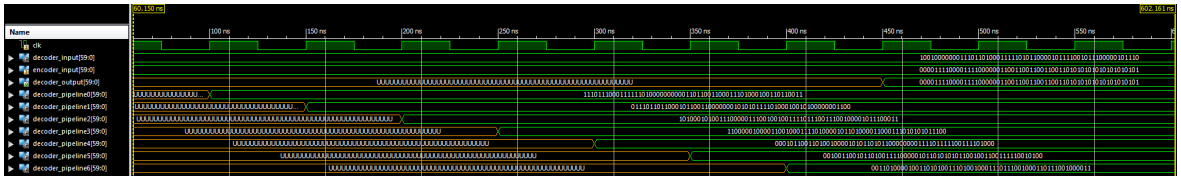


Figure C.1: Simulation of FPGA-based Reed-Solomon Encoding and Decoding

Appendix D

TPC Tracking Model

Let measurements $\mathbf{m}_k$ depend linearly on state vectors $\mathbf{r}_k^t$ such that $\mathbf{r}_{k+1}^t$ depends linearly on the previous state $\mathbf{r}_k^t$ ([Roh 10 I, 2.1] presents the model in detail). Let $\mathbf{r}_k$ be an estimator:

$$\mathbf{r}_k^t = A_k \mathbf{r}_{k-1}^t + \nu_k \qquad \mathbf{m}_k = H_k \mathbf{r}_k^t + \eta_k \qquad \epsilon_k = \mathbf{r}_k^t - \mathbf{r}_k \qquad C_k = \mathrm{cov}(\epsilon_k)$$

The Kalman filter determines the best linear unbiased ($\langle \epsilon_k \rangle = 0$) estimator, which minimizes $\sigma_k^2 = \langle ||\epsilon_k||^2 \rangle$, if the following conditions are met: (See [Kal 60] for a proof.)

$$\langle \eta_k \eta_l^T \rangle = \langle \nu_k \nu_l^T \rangle = \langle \eta_k \rangle = \langle \nu_k b \rangle = 0 \qquad \mathrm{cov}(\eta_k) = V_k \qquad \mathrm{cov}(\nu_k) = Q_k$$

It initializes $\mathbf{r}_0$ arbitrarily and C_0 to infinity, then alternately performs an extrapolation (D.1)

$$\tilde{\mathbf{r}}_k = A_k \mathbf{r}_{k-1} \qquad \tilde{C}_k = A_k C_{k-1} A_k^T + Q_k \tag{D.1}$$

$$C_k = \tilde{C}_k - K_k H_k \tilde{C}_k \qquad \chi_k^2 = \chi_{k-1}^2 + \zeta_k^T (V_k + H_k \tilde{C}_k H_k^T)^{-1} \zeta_k \tag{D.2}$$

$$K_k = \tilde{C}_k H_k^T (V_k + H_k \tilde{C}_k H_k^T)^{-1} \qquad \zeta_k = (\mathbf{m}_k - H_k \tilde{\mathbf{r}}_k) \qquad \mathbf{r}_k = \tilde{\mathbf{r}}_k + K_k \zeta_k$$

and a filter step (D.2). A particle of charge q in a magnetic field follows a trajectory[1]:

$$\mathbf{r}(t) = \mathbf{r}_0 + \big(R \cdot \cos\omega(t - t_0) + \vartheta_0,\ R \cdot \sin\omega(t - t_0) + \vartheta_0,\ \lambda(t - t_0)\big)^T$$

The employed state vector $\mathbf{r}_k^t = \mathbf{r}^t(x_k)$ is modeled after X not t. It has transformed coordinates:

$$Y = r_{0,y} + R \cdot sin(\omega(t - t_0) + \vartheta_0) \qquad Z = r_{0,z} + \lambda(t - t_0) \qquad \sin(\varphi) = \cos(\vartheta)$$

$$\lambda = \frac{dz}{ds} = \frac{p_z}{|\mathbf{p}|} \qquad \kappa = \frac{q}{p_\mathrm{t}} = \frac{B_z}{R} \qquad (ds = |d\mathbf{r}|)$$

The extrapolation from x_0 to the state vector $\tilde{\mathbf{r}}^t = (\widetilde{Y}, \widetilde{Z}, \sin(\widetilde{\varphi}), \widetilde{\lambda}, \widetilde{\kappa})^T$ at $x = x_0 + \triangle x$

$$\widetilde{Z} = Z + \lambda \cdot 2 \underbrace{\Big(\kappa B_z\Big)^{-1} \arcsin\Big(\frac{1}{2}\kappa B_z \frac{\triangle x}{\cos(\varphi + \widetilde{\varphi})}\Big)}_{\mathrm{ds}} \qquad \sin(\widetilde{\varphi}) = \sin(\varphi) + \triangle x \cdot B_z \cdot \kappa$$

$$\widetilde{Y} = Y + \cos(\widetilde{\varphi}) - \cos(\varphi) = Y + \triangle x \cdot \tan\Big(\frac{\varphi + \widetilde{\varphi}}{2}\Big) = Y + \triangle x \cdot \frac{\sin(\varphi) + \sin(\widetilde{\varphi})}{\cos(\varphi) + \cos(\widetilde{\varphi})}$$

is non-linear. In this case, one can linearize $\tilde{\mathbf{r}} \approx F_{lin}(\mathbf{r}) = F_x(\mathbf{r}_0) + \partial F_x|_{\mathbf{r}_0}(\mathbf{r} - \mathbf{r}_0)$ and repeat all the Kalman filter iterations, using each result as linearization point in the next repeat to approximate the optimal estimator [Gor 12, 1.3]. In contrast to the merger, the slice tracker assumes $\mathrm{cov}(Y, Z) = 0$ for performance reasons. Its filter step (D.2) reads:

$$Y_k = \widetilde{Y}_k + \frac{C_k^{0,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k) \quad Z_k = \widetilde{Z}_k + \frac{C_k^{1,1}}{\sigma_z^2 + C_k^{1,1}}(z - \widetilde{Z}_k) \quad \lambda_k = \widetilde{\lambda_k} + \frac{C_k^{3,1}}{\sigma_z^2 + C_k^{1,1}}(z - \widetilde{Z}_k)$$

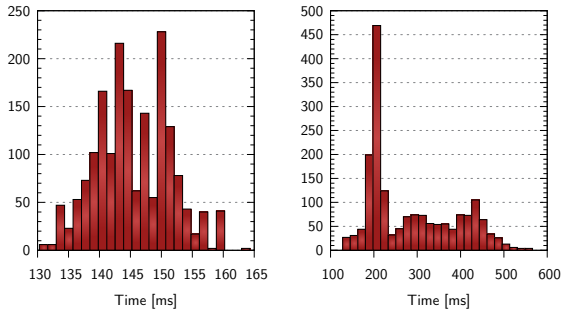
$$\kappa_k = \widetilde{\kappa_k} + \frac{C_k^{4,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k) \quad \sin(\varphi_k) = \widetilde{\sin}(\varphi_k) + \frac{C_k^{2,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k)$$

[1] The transversal momentum p_t is the projection of the momentum orthogonal to the beam.

Appendix E

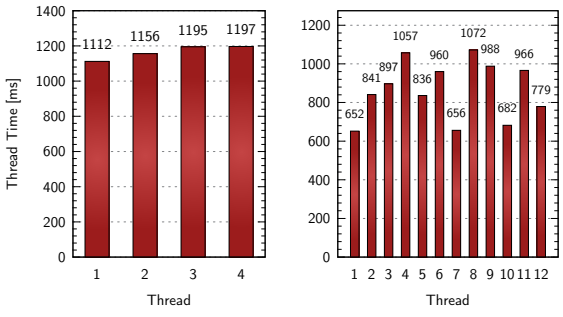
CPU Tracker Performance Evaluation

The CPU tracker uses trivial parallelization in which each thread processes a different slice. This turned out to be both the simplest and most efficient way. For many-core processors, this leads to a problem. Each event consists of 36 slices. The compute nodes at CERN, dual socket systems with either twelve-core Magny-Cours or quad-core Nehalems with Hyperthreading, offer 16 or 24 (virtual) cores, both no divisor of 36. Hence, no balanced scheduling is possible. Processing multiple events concurrently would overcome this problem. The GPU tracker could benefit from this approach, too. However, the current HLT framework does not support processing multiple events in one component simultaneously. For this reason, in order to enable a fair comparison, in this entire thesis both GPU and CPU tracker always process only one event at a time. The benchmarks are performed mostly on a single quad-core Nehalem. Since the tracker can benefit from Hyperthreading (see [Roh 10 I, 9.2.1]), but the thread count eight does not divide 36, the benchmarks use twelve concurrent slices. Fig. E.1 demonstrates that the scheduler introduces huge variations, which are high for four threads already and increase enormously with twelve threads because twelve is no multiple of the core count eight. In order to reach the 0.2 % statistical error limit, several hundred runs are needed.



(a) Four Threads (b) Twelve Threads

Figure E.1: Processing Time of a Single Slice [III]



(a) Four Threads (b) Twelve Threads

Figure E.2: Active Tracking Time of each Thread [III]

Fig. E.2 shows the total execution time of each thread for a single event in a four- and a twelve-thread configuration. The average thread execution time of twelve threads is about 19 % lower than the maximum thread execution time. In this thesis, performance plots like Fig. 6.1 always show average values for a single slice as the time for independent tracking steps. But since the tracker always processes one full event at a time, the full tracking time is defined as the maximum of the thread times (even though the average time is lower). Hence, the full tracking time is longer than the sum of the individual times.

Appendix F

Explicit Encoding Examples

F.1 Code Examples

From the theorem of Grunwald Wang, only the existence of MDS-codes on ${}^{R}/_{2^b R}$ can be deduced. Naturally, an explicit description of a code is desired. For many l, such codes can be gained through cyclotomic fields. This section presents the example of these fields in detail.

Definition 33: *The m^{th} roots of unity are the complex numbers $\zeta_m^j = e^{\frac{2\pi ij}{m}}$. A root is called primitive if j and m are coprime. The field $\mathbb{Q}(\zeta_m)$ is called the field of m^{th} roots of unity. In the following, it is denoted by K_m. All these fields are called cyclotomic fields.*

The Galois theory of cyclotomic fields is pretty well understood. The following proposition summarizes the (well-known) properties.

Proposition 34: *Consider the field K_m. Then:*

(i) $[K_m : \mathbb{Q}] = \varphi(m)$ *with φ being the Euler-φ function, which is the count of numbers smaller than and coprime to m.*

(ii) Both the field and the ring of algebraic integers are generated as an algebra by a single primitive m^{th} root of unity.

(iii) The Galois group $G = Gal({}^{K_m}/_{\mathbb{Q}})$ is isomorphic to $({}^{\mathbb{Z}}/_{m\mathbb{Z}})^*$.

(iv) The minimal polynomial $\Phi_m(x)$ can be calculated recursively by

$$\Phi_m(x) = \frac{X^m - 1}{\prod\limits_{\substack{d|m \\ d<m}} \Phi_d(x)}.$$

(v) An integral basis of $\mathcal{O}_{K_m}$ is given by $\{1, \zeta_m, \zeta_m^2, ..., \zeta_m^{\varphi(m)-1}\}$.

The integral basis is of particular importance for generating encoding matrices. A proof can be found in [Neu 99, 10.2]. Fortunately, also the ramification of primes in ${}^{K_m}/_{\mathbb{Q}}$ is well known and given by the following

Proposition 35: *Write $m = p^\nu \cdot x$ with $p \nmid x$. Let f be minimal with $p^f \equiv 1$ mod x. Then the ramification index is $e = \varphi(p^\nu)$ and the inertia degree equals f.*

For a proof see [Neu 99, 10.3]. From Proposition 35 it follows immediately that the prime 2 is inert if and only if $2 \nmid m$ and $f = \varphi(m)$ is minimal such that $2^f \equiv 1 \bmod m$. The latter condition is equivalent to 2 generating the multiplicative group $({}^{\mathbb{Z}}/_{m\mathbb{Z}})^*$. This further translates to $2^{\frac{\varphi(m)}{2}} \not\equiv 1 \bmod m \Leftrightarrow 2^{\frac{\varphi(m)}{2}} \equiv -1 \bmod m \Leftrightarrow m \mid 2^{\frac{\varphi(m)}{2}} + 1$. Furthermore, the proof of Lemma 15 showed that the extension must be cyclic, i. e. m must be a prime power. This gives some examples for MDS-codes on $\mathcal{O}_{K_m}$.

Example 36: *Assume $p = 2$, $m = p'^s$ a prime power with $p' \neq 2$, $\varphi(m) = (p'-1)p'^{s-1} = l \geqslant n+k$ and $m \mid 2^{\frac{l}{2}}+1$. Consider $n+k$ pairwise disjoints elements λ_i of the form (out of 2^l combinations)*

$$\lambda_i = \begin{cases} 1 \\ 0 \end{cases} + \begin{cases} \zeta_m \\ 0 \end{cases} + \begin{cases} \zeta_m^2 \\ 0 \end{cases} + \cdots + \begin{cases} \zeta_m^{\varphi(m)-1} \\ 0 \end{cases}.$$

Then the matrix defined by $M_{ij} = \lambda_i^j$ represents an MDS-code. Clearly, an analogous result holds for an arbitrary prime instead of 2.

Each of the l automorphisms of the Galois group maps ζ_m to a different primitive root of unity ζ_m^j. This gives an explicit rule for inverting elements using the algorithm from the proof of Proposition 11. Using the explicitly known minimal polynomial, elements of K_m can be multiplied. A simple bound for the complexity of creating the encoding and decoding-matrix is $\mathcal{O}((n+k)^3 \cdot (\log(n+k))^2)$.

A simple calculation reveals that cyclotomic fields deliver the following admissible l:
$1, 2, 4, 6, 10, 12, 18, 20, 28$.

Since all of them are imaginary extensions, the corresponding totally real subextensions extend this list to:
$1, 2, 3, 4, 5, 6, 9, 10, 12, 14, 18, 20, 28$.

F.2 Generation Encoding Matrices

The following appendix explicitly formulates how matrices for codes on $\mathbb{Z}/p^b\mathbb{Z}$ are generated using cyclotomic extensions. It is assumed that $l \in \{2, 4, 6, 10, 12\}$. In the second part, a code that operates on $\mathbb{F}_2$ is created for arbitrary l.

F.2.1 Codes on $\mathbb{Z}/p^b\mathbb{Z}$

In the following, R denotes the ring $\mathbb{Z}/2^b\mathbb{Z}$ and S denotes the extension $R[X]/f$ with a normalized irreducible polynomial $f = \sum_{i=0}^{l} \alpha_i X^i$ of degree l. For computations on computers, the algorithm is expressed only in terms of operations within R, i. e. setting $b = 32$ usual integer calculation with 32-bit integers can be used. All following operations are integer calculations with overflow where every division is rounded down.

At first, by exhaustive search an admissible l is determined that satisfies the conditions (see Example 36):

$$m = p^s, \quad \varphi(m) = (p-1)p^{s-1} = l \geqslant n+k, \quad p \text{ prime}.$$

For cyclotomic extensions, the roots of the irreducible polynomial (of a root of unity) are the primitive roots of unity. Thus, the polynomial Φ_m can be recursively calculated by

$$\Phi_m(x) = \frac{x^m - 1}{\prod\limits_{\substack{d \mid m \\ d \neq m}} \Phi_d(x)}.$$

Consider the basis $1, X, X^2, \cdots, X^{l-1}$ of S over R. In the following, every element x of S is represented by a tuple of elements $x^{(i)} \in R$, $(0 \leqslant i < l)$. Addition $(z = x + y)$ is performed component-wise $(z^{(i)} = x^{(i)} + y^{(i)})$. Multiplication $(z = x \cdot y)$ is performed in two steps: first a distributive multiplication is performed:

$$w^{(i)} = \sum_{j+k=i} x^{(j)} \cdot y^{(k)} \quad (0 \leqslant i \leqslant (l-1)^2).$$

Second, high power coefficients are reduced recursively using the minimal polynomial. For $k \geqslant l$, starting with the highest k (in this case $k = (l-1)^2$), the $w^{(i)}$ are updated via

$$w_{\text{new}}^{(i)} = \begin{cases} 0 & i = k \\ w^{(i)} - w^{(k)} \cdot \alpha_{i+l-k} & k-l \leqslant i < k \\ w^{(i)} & i < k-l \end{cases} \tag{F.1}$$

until all elements $w^{(i)}$, $i \geqslant l$ are zero. These $w^{(i)}$ form the result $z^{(i)} = w^{(i)}$.

For creating the Vandermonde matrix, $n+k$ values λ_j, $0 \leqslant j < n+k$ are chosen which are pairwise distinct modulo 2. A possible choice is $\lambda_j^{(i)} = \frac{j}{2^i} \bmod 2$. (Calculated in $\mathbb{Z}$ and rounded down.) An $(n+k) \times n$ Vandermonde matrix V is generated by $V_{ji} = \lambda_j^i$.

The inverse I of the upper $n \times n$ submatrix of V is calculated by Gaussian elimination. For this, the inverse of a scalar element $x \in S$ is calculated in the following way: first the Galois conjugates of x are multiplied:

$$w = \prod_{\substack{\sigma \in \mathrm{Gal}(K_m/\mathbb{Q}) \\ \sigma \neq 1}} \sigma(x) = \prod_{\substack{\gcd(d,m)=1 \\ 1<d<m}} \sigma_d(x) \quad \text{with} \quad (\sigma_d(x))^{(i)} = \sum_{\substack{d \cdot j = i \\ j<l}} x^{(j)}. \tag{F.2}$$

Next, w is reduced in the same way as for the multiplication and the product $y = w \cdot x$ is calculated (using the reduced w). Since y is the norm of x, it is ensured that $y \in R$, i. e. $y^{(i)} = 0$ for $i \neq 0$ and that $\gcd(y, 2) = \gcd(y^{(0)}, 2) = 1$. Using the extended Euclidean algorithm two integers β, γ can be calculated such that $\beta y^{(0)} + 2^b \gamma = 1$. The inverse of x is obtained by calculating $x^{-1} = \beta w$.

The lower $k \times n$ submatrix M of $V \cdot I$ is the encoding-matrix of the MDS-code. The multidimensional $lk \times ln$ encoding-matrix is created by substituting the scalars $M_{i'j'} \in S$ of M by $l \times l$ submatrices with entries $(M_{i'j'})_{ij}$. The matrix A corresponding to a scalar a is constituted such that its j^{th} column consists of the components of the tuple obtained by the product $a \cdot e_j$. Here, e_j is the j^{th} basis vector defined by $e_j^{(i)} = \delta_{ij}$.

The resulting matrix can be used for IGEMM encoding. For BGEMM encoding, the least significant bit of each matrix entry is replicated to all bits of that entry ($x \to (x \bmod 2) \cdot 0\text{xFFFFFFFF}$). The decoding-matrix is created by Gaussian elimination. (A faster version could use the explicitly known inverses of Vandermonde matrices.)

F.2.2 *XOR*-only Codes for arbitrary l

The above described algorithm requires a cyclotomic extension with a particular m. This is not possible for arbitrary l. A simple change to the algorithm allows one to use arbitrary l but only for BGEMM and *XOR*-only encoding.[1]

The minimal polynomial f must be replaced by an arbitrary irreducible polynomial of degree l which remains irreducible in $\mathbb{F}_2$. QEnc uses the polynomials from [Ser 98]. In the inversion step of scalars, w is calculated by:

$$w = \prod_{0<d<l} \sigma_d(x) \quad \text{with} \quad (\sigma_d(x))^{(i)} = \sum_{\substack{2^d \cdot j = i \\ j<l}} x^{(j)}.$$

Also the IGEMM and the *Add*-only code can in principle operate with arbitrary l. However, as described before, in certain cases the construction with cyclotomic fields employed by QEnc does not work. It is a straight forward task to use the alternative construction from Section 16.3.2, but that is a major effort which turned out to be unnecessary since the binary codes perform better.

[1] A more elaborate modification (see Lemma 21) could provide support for the IGEMM with arbitrary l as well, but this was not implemented in QEnc since BGEMM showed better performance anyway.

F.3 C Example Code for QEnc Blocking Levels

Listing F.1 illustrates all the blocking levels of QEnc in form of a C++ example. It includes the temporary variables for register blocking, but in order to keep it simple, it neither includes the scratch registers nor the Strassen algorithm nor any of the optimizations applied to the assembler code in Chapter 17. The assembler code is in principle the completely unrolled version (except for the loops over k and $k0$ of course) of this C++ example, with all the tweaks applied. (See Listing 17.25 for an exemplary assembler code.)

```
void encode(int* pC, int* pD, int* pM, int nRows, int nCols, int nBlocks)
{
 const int bRegister = 14, bL1Data = 96, bL2Code = 2048, bL2Data = 96;
 //The L1 Instruction Blocking Constants are chosen for 32 KB Instruction Blocks
 const int bL1CodeX = getCodeBlockingNBlocksX(bL1Data, bRegister, nRows, nCols);
 const int bL1CodeY = getCodeBlockingNBlocksY(bL1Data, bRegister, nRows, nCols);
 memset(pC, 0, nRows * nBlocks * sizeof(pC[0]));

 for (int i0 = 0;i0 < nCols;i0 += bL2Code)
 { //Outermost loop over Columns of M
  for (int j0 = 0;j0 < nRows;j0 += bL2Code)
  { //Outermost loop over Rows of M
   const int nCols0 = min(nCols, i0 + bL2Code), nRows0 = min(nRows, j0 + bL2Code);
   for (int k0 = 0;i0 < nBlocks; k0 += bL2Data)
   { //Outermost Loop over Input Data (Inside the Loop over M for L2 Code Blocking)
    for (int i1 = i0;i1 < nCols0;i1 += bL1CodeX * bL1Data)
    { //X Part of L2 Instruction Blocking
     for (int j1 = j0;j1 < nRows0;j1 += bL1CodeY * bRegister)
     { //Y Part of L2 Instruction Blocking
      const int nCols1 = min(i1 + bL1CodeX * bL1Data, nCols0);
      const int nRows1 = min(j1 + bL1CodeY * bRegister, nRows0);
      for (int k = k0;k < k0 + bL2Data && k < nBlocks;k++)
      { //L2 Data Blocking
       for (int i2 = i1;i2 < nCols1;i2 += bL1Data)
       { //X Part of L1 Instruction Blocking
        for (int j2 = j1;j2 < nRows1;j2 += bRegister)
        { //Y Part of L1 Instruction Blocking
         register int tmp[bRegister]; //Registers for Register Blocking
         for (int j = 0;j < bRegister;j++) tmp[j] = 0;

         for (int i = i1;i < i1 + bL1Data && i < nCols1;i++)
         { //L1 Data Blocking
          for (int j = j1;j < j1 + bRegister && j < nRows1;j++)
          { //Register Blocking
           if (pM[j * nCols + i]) tmp[j] ^= pD[k * nCols + i];
          }
         }

         //Save Data from Register Blocking
         for (int j = 0;j < bRegister;j++) pC[k * nRows + j] ^= tmp[j];
} } } } } } } } }
```

Listing F.1: All Blocking Levels in QEnc

In the code, *nRows* is the number k of rows of M, *nCols* the number n of columns of M, *nBlocks* the number S of data blocks, pC points to the encoded data C, pD points to the source data D, and pM points to M. (Consider that the multiplication is $C = D \cdot M^T$.) All blocking levels are visual as one loop, some requiring two loops due to the two-dimensional nature of matrices.

The modifications for other data types or the *Add*-only variant are marginal. The latter can be realized simply by exchanging the *XOR* operations for additions. The 32-bit *int* data type can be replaced by *__mm128* for SSE (and by *__mm256* for AVX) – in that case also SSE loads, stores, and logical operations must be used. The matrix M contains only single-bit entries storable with a *char* or in bit-arrays.

Appendix G

CALDGEMM & HPL-GPU Settings

Table G.1 lists most of the parameters CALDGEMM and HPL-GPU offer for tuning. The table lists suggested values and whether a feature should be used respectively for different exemplary platforms: a system with a slow CPU (e. g. one dual core processor), a system where both CPU and GPU provide significant processing capabilities (e. g. the LOEWE-CSC), and multi-GPU systems like Sanam based on an Intel and an AMD platform. The section of this thesis which discusses a feature is listed below the feature name. In cases where sufficient benchmarks have been performed, the relative performance gain on the exemplary systems is shown. It must be noted that these values are only based on the servers used for benchmarks in this thesis. Other but similar systems might show different results. Certain features have constrains, have been developed for a particular hardware, or improve energy efficiency rather than performance (and might even slow down HPL slightly). Such relevant dependencies are printed in boldface within the comments below each feature. The suggestions of features for the exemplary systems are categorized as follows:

- **No**: Does not apply to this platform / Should not be used.
- **Optional**: Might yield small improvements but is not that relevant.
- **Recommended**: Should be used if there are no issues.
- **Mandatory**: Required on this platform to reach good performance or to enable other features.
- **Depends**: Results depend on other factors mentioned in the respective comment below.

Feature	Slow CPU	Single-GPU	Multi-GPU (Intel)	Multi-GPU (AMD)
Blocking Size $N_b = k$ (*12.6.2.1*)	512 – 1024	≈ 1024	≈ 1920	≈ 2048
	Larger N_b shifts workload from the DGEMM to the less efficient factorization, smaller N_b needs more memory and PCIe bandwidth. Large and small N_b can slow down GPU DGEMM.			
DGEMM Tile Size (h) (*11.2.5.1*)	⩽ 4096	⩽ 4096	⩽ 3072	⩽ 3072
	Usually automatically adjusted, large h for large matrices and small h for small matrices. Larger blockings (N_b) favor smaller tiles. Certain values might lead to bad DGEMM kernel performance.			
Kernel Output via Zero-Copy to Host (*12.5.2*)	Depends	Depends	No (≈ 91 % slower)	Depends
	*Usually faster on **Cypress** and slower on **Tahiti** but depends on the system. On **multi-GPU systems** this can slow down the GPU DGEMM due to memory latencies. Mandatory on **Cayman** due to DMA problem (Section 12.8.3.1). Not compatible with **Intel chipsets** (Section 12.5.2).*			

Continued on next page

Table G.1: CALDGEMM & HPL-GPU Settings

Feature	Slow CPU	Single-GPU	Multi-GPU (Intel)	Multi-GPU (AMD)
		Table G.1 – *Continued from previous page*		
Implicit DMA Synchronization (*12.5.2*)	Depends	Depends	Mandatory (≈ 28 %)	Depends
	Should be used if supported by the driver if the Zero-Copy output is not used. Hence it is strongly recommended on Intel platforms. If the driver does not support this, there is a fallback for Intel platforms.			
MergeBuffer Threads per GPU (*11.3.4.1*)	1	1 – 2	1	2
	If Zero-Copy is used without the binary driver patch, two such threads are required on AMD systems. Interlagos can run two threads on the two cores of one module but should not mix DGEMM and MergeBuffer threads on one module.			
Asynchronous DMA Transfer (*11.2.4.6*)	Recommended	Mandatory (≈ 16 %)	Mandatory	Mandatory
	The asynchronous transfer is required for the pipeline and is the more important the faster the system is.			
Lookahead 1 (*11.3.4*)	No	Recommended (≈ 8 %)	Recommended (≈ 22 %)	Recommended (≈ 11 %)
	Hides factorization and broadcast time behind GPU DGEMM. Systems with slow CPU should rather offload more tasks to GPU.			
Lookahead 2a (*11.3.4.2*)	No	Optional (≈ 2 %)	Recommended (≈ 14 %)	Recommended (≈ 12 %)
	Pipelines DTRSM and LASWP with GPU DGEMM.			
Lookahead 2b (*14.2.4.2*)	No	Optional	Recommended (≈ 12 %)	Recommended
	Pipelines Z-Broadcast with GPU DGEMM.			
Early Lookahead (*14.2.2*)	No	Optional	Recommended (≈ 2 %)	Recommended
	Shifts more DGEMM workload to GPU with lookahead enabled. Important if CPU is the limiting factor (multi-GPU AMD systems) and for ***power efficiency****.*			
Dynamic Lookahead Turnoff (*11.3.4.2*)	No	Depends	No	Mandatory
	Mostly on ***AMD*** *systems it is better to turn of lookahead at a certain time, serializing and speeding up the individual tasks.*			
Parallel DMA Threads (*14.2.4.1*)	No	No	Recommended (≈ 8 %)	Recommended (≈ 16 %)
	Uses one management thread per GPU. Grouped DMA threads should be used instead if ***power efficiency*** *is more important than performance.*			
Grouped DMA Threads (*14.2.4.1*)	No	No	Recommended	Recommended
	Group management threads for GPUs.			
Repinning DMA Threads (*14.2.1*)	No	No	Mandatory (≈ 30 %)	No
	This is mandatory on Intel platforms when GPUs are connected to different CPU dies and when the grouped DMA mode is not used.			
Combined CPU / GPU DGEMM (*11.2.3*)	Optional	Mandatory (≈ 34 %)	Recommended	No
	Use both CPU and GPU for DGEMM. Does not always work for multi-GPU since it causes plenty of memory load and can slow down the GPU DGEMM.			
Second Phase CPU DGEMM (*11.2.4.5*)	Optional	Recommended	Optional	No
	Dynamically schedule a second large CPU DGEMM run in order to achieve higher CPU utilization.			
		Continued on next page		

Table G.1: CALDGEMM & HPL-GPU Settings

Feature	Slow CPU	Single-GPU	Multi-GPU (Intel)	Multi-GPU (AMD)
				Table G.1 – *Continued from previous page*
Third Phase CPU DGEMM (*11.2.4.5*)	Optional	Optional	No	No
	Uses a work-stealing scheduler for full CPU and GPU utilization after the initial two large CPU DGEMM runs. Causes too much overhead for multiple GPUs.			
Non Square Tile Size (*12.8.4*)	Mandatory	No	Recommended	Mandatory
	*Use non square tiles in DGEMM. This provides more freedom to choose the CPU / GPU splitting ratio. It can be used to shift more DGEMM workload to the GPU (by reducing the minimum CPU part) or to improve the CPU utilization without the dynamic scheduler. This feature is important when the CPU is a bottleneck and for **power efficiency**. Can slightly slow down GPU DGEMM.*			
Balanced CPU Affinities (*12.8.4*)	Mandatory (≈ 156 %)	No	No	Recommended
	Usually, GPU-related threads should be pinned to CPU cores close to the GPU. When memory bandwidth is the bottleneck, they should be distributed on all CPU dies to aggregate more bandwidth. Systems with few CPU cores should use all cores anyway.			
NUMA-aware Thread Affinities (*12.8.4*)	No	Optional	Recommended	Recommended
	Distributes all CPU-related workloads equally on all NUMA nodes as far as possible.			
Factorization Parameters (NBMin / NBDiv) (*12.8.4*)	64 / 2	64 / 2	32 / 2	16 / 4 – 32 / 2
	*The parameters can be optimized for faster factorization and for less resource requirements on **multi-GPU AMD platforms**. They can be adjusted during the HPL run speeding up the DGEMM during the GPU dominated phase and the factorization during the CPU dominated phase.*			
Reduced Factorization Thread Count (*11.3.4.1*)	No	Depends	Depends	Mandatory
	*Important for **AMD** systems with lookahead. Improves **power efficiency** in general.*			
Dynamic BLAS Thread Count (*12.6.4.2*)	No	Depends	No	Recommended
	*This improves the performance on **AMD** systems.* CALDGEMM provides a patch for **GotoBLAS** and a wrapper for **ACML**.			
Offload Factorization to GPU (*12.6.4.1*)	Recommended	No	No	No
	*In general, all systems could profit from GPU-based factorization at the end of the run. However, the current implementation does not support this. This would also improve the **power efficiency**.*			
Split DTRSM in DTRTRI and DGEMM (*12.11*)	Recommended (≈ 16 %)	No	No	No
	This offloads a large fraction of the DTRSM workload to the GPU.			
Dynamic N_b Reduction (*12.8.4*)	Recommended	No	No	Optional
	*Speeds up the last iterations of HPL by shifting workload to the GPU. Incompatible with **multi-node runs**.*			
Huge Pages (*11.2.5.1*)	Recommended	Depends	Recommended	Recommended (≈ 3 %)
	Does not work well on certain systems. Must be evaluated for each server individually.			
Interleaved Memory (*C.5*)	No	Depends	Recommended	Mandatory
	*Interleaved memory should be used on all **NUMA** systems. The influence is stronger on AMD CPUs than on Intel CPUs.*			
Alternate Tile-Scheduler (*12.6.2.3*)	No	No	Recommended	Recommended
	*The alternate scheduler (about 2 % faster) becomes mandatory on **multi-GPU systems** if the GPU memory does not suffice for enough BBuffers.*			
				Continued on next page

Table G.1: CALDGEMM & HPL-GPU Settings

Feature	Slow CPU	Single-GPU	Multi-GPU (Intel)	Multi-GPU (AMD)
		Table G.1 – *Continued from previous page*		
Multithreaded Divide Buffer (*12.6.2.2*)	No	No	Recommended	Optional (up to 2 %)
	At the beginning of the run when the BBuffers are not already filled, the DivideBuffer routing can be the bottleneck. A second thread is useful if multiple GPUs are present and connected to different CPU dies. In that case, it is strongly recommended on Intel systems, where the obligatory repinning of the DMA thread occupies an additional CPU core anyway.			
Binary AMD Driver Patch (*11.2.5*)	Depends	Depends (up to 9 %)	No	Depends
	*Recommended on all AMD systems with the **Zero-Copy kernel output**.*			
GotoBLAS Patch (*11.2.3.1*)	*Mandatory if **GotoBLAS** is used as BLAS library. Newer systems like Interlagos and Sandy-Bridge should use MKL or ACML and do not need this.*			
MKL BLAS Library (*12.9*)	Depends	Depends	Mandatory	No
	*About as fast as GotoBLAS on older Intel CPUS, mandatory for **AVX** and for multiple GPU since it lowers the memory load.*			
ACML BLAS Library (*12.9*)	Depends	Depends	No	Depends
	*Only recommended for **AMD** CPUs with **AVX** support. On Magny-Cours CPUs GotoBLAS is significantly faster*			
GCC OpenMP Patch (*14.2.1*)	*Recommended for AMD **ACML** 5.2, since it adds support for ACML's variable thread count feature.*			
Transposed Matrices in Kernel (*11.2.5.1*)	*Usually, transposed B-matrix for **Cypress** and **Cayman** GPU, transposed A-matrix for **Tahiti** GPU. For very small matrices the kernel with the matching parameter is better since the pipeline cannot hide the latency for transposition.*			
Shift Even Matrix Rows (*14.1*)	*Enable on **Tahiti** GPU, disable on **Cypress** GPUs, on **Cayman** GPU, and for **power efficiency**.*			
InfiniBand RDMA (*11.3.3*)	*Improves performance by up to ≈ 2 % but leads to incompatibilities in combination with GPUs on certain platforms.*			
Include Padding during U-broadcast (*14.2.4.2*)	*This is incompatible with lookahead 2b but it is recommended otherwise.*			
MPI Data Types during U-broadcast (*14.2.4.2*)	*Should be used if the padding is not transferred during U-matrix broadcast. Currently slow with FDR InfiniBand.*			
64-bit DMA Transfers (*12.8.3.2*)	*Should be enabled on all **Cayman** GPUs to circumvent a problem with the DMA engine.*			
DMA Fallback without implicit synchronization (*12.5.2*)	*Fallback for AMD drivers where the implicit synchronization does not work. About unit[2]% slower.*			

Table G.1: CALDGEMM & HPL-GPU Settings

Appendix H

Test & Development Systems

The following systems were used for benchmarks:

	System	CPU	Clock	GPU	Clock	Memory	Clock	Operating System
[I]	**qon0** *(Development system of the author)*	Intel Nehalem Core i7-975 4C	3.78 GHz	NVIDIA GeForce GTX285	700 MHz	12 GB DDR3	1600 MHz	Windows Vista & Gentoo Linux 64-Bit
[II]	**qon1** *(Upgrade of qon0 with new CPU, GPU and Memory)*	Intel Nehalem Core i7-975 6C	4 GHz	NVIDIA GeForce GTX580 AMD Radeon 6970	772 MHz 880 MHz	24 GB DDR3	1600 MHz	Windows 7 & Gentoo Linux 64-Bit
[III]	**gpu-dev00** *(NVIDIA development system at FIAS)*	2 × Intel Nehalem Core i7-930 4C	2.8 GHz	NVIDIA GeForce GTX295 NVIDIA GeForce GTX480 NVIDIA GeForce GTX580	576 MHz 700 MHz 772 MHz	12 GB DDR3	1333 MHz	Ubuntu Linux 64-Bit
[IV]	**gpu-dev01** / **gpu-dev02** *(AMD dual-GPU development system at FIAS)*	2 × Intel Nehalem E5520 4C	2.27 GHz	AMD Radeon 5970	700 MHz	24 GB DDR3	1066 MHz	Ubuntu Linux 64-Bit / SUSE Enterprise Server 11.3 with Real-Time Extensions
[V]	**gpu-dev03 – gpu-dev08** *(Early development nodes with LOEWE-CSC configuration for HPL)*	2 × AMD Magny-Cours 6172 12C	2.1 GHz	AMD Radeon 5870	850 MHz	48 GB (02 / 03) / 64 GB (05-08) DDR3	1333 MHz	openSUSE Linux 11.2 64-Bit
[VI]	**gpu-dev09** *(Development system for multi-GPU DGEMM)*	2 × AMD Magny-Cours 6174 12C	2.2 GHz	3 × AMD Radeon 6970/ 5870/V7800/7970[2]	880 MHz	128 GB DDR3[1]	1333 MHz	openSUSE Linux 11.3 64-Bit
[VII]	**gpu-dev10** *(Development system for multi-GPU DGEMM on Interlagos)*	2 × AMD Interlagos 6278 16C	2.4 GHz	3 × AMD Radeon 7970	925 MHz	128 GB DDR3	1600 MHz	openSUSE Linux 11.3 64-Bit
[VIII]	**gpu-dev11** *(Development system for multi-GPU DGEMM on Sandy Bridge)*	2 × Intel Sandy Bridge E2650 8C	2.0 GHz	4 × AMD Radeon 7970	925 MHz	128 GB DDR3	1600 MHz	openSUSE Linux 12.1 64-Bit
[IX]	**LOEWE-CSC GPU Node** *(GPU-Nodes of the LOEWE-CSC)*	2 × AMD Magny-Cours 6172 12C	2.1 GHz	AMD Radeon 5870	700 - 850 MHz	64 GB DDR3	1333 MHz	Scientific Linux 64-Bit
[X]	**LOEWE-CSC CPU Quad Node** *(Quad-Nodes of the LOEWE-CSC)*	4 × AMD Magny-Cours 6172 12C	2.1 GHz	-		128 GB DDR3	1333 MHz	Scientific Linux 64-Bit
[XI]	**cngpu01 – cngpu34** *(GPU-Compute-Nodes at CERN for the ALICE GPU Tracker)*	2 × AMD Magny-Cours 6172 12C	2.1 GHz	1 × NVIDIA GTX480	700 MHz	64 GB DDR3	1333 MHz	Ubuntu Linux 64-Bit
[XII]	**avx-dev** *(Intel based AVX development system)*	1 × Intel Sandy Bridge i5-2500 4C	3.3 GHz[3]	-		8 GB DDR3	1600 MHz	Linux 64-Bit
[XIII]	**dual6990**	1 × AMD Magny-Cours 6176 12C	2.3 GHz	2 × AMD Radeon 6990	830 MHz	128 / 256 GB DDR3	1333 MHz	Linux 64-Bit
[XIV]	**kacst** *(Sanam Node)*	2 × Intel Sandy Bridge E2650 8C	2.0 GHz	2 × AMD S10000	950 MHz	128 GB DDR3	1600 MHz	SLES 11 SP2
[XV]	**tyan0**	2 × Intel Xeon X5680 6C	3.33 GHz	3 × AMD FireStream 9350	700 MHz	24 GB DDR3	1333 MHz	Linux 64-Bit
[XVI]	**sm0** *(SuperMicro AMD blade system)*	2 × AMD Magny-Cours 6174 12C	2.2 GHz	-		64 GB DDR3	1333 MHz	Chaos
[XVII]	**sm1** *(SuperMicro Intel blade system)*	2 × Intel Xeon X5650 6C	2.67 GHz	-		64 GB DDR3	1333 MHz	Chaos
[XVIII]	**tesla** *(NVIDIA Fermi Tesla test syste)*	2 × Intel Xeon X5650 6C	2.67 GHz	2 × NVIDIA M2070	700 MHz	24 GB DDR3	1333 MHz	Linux 64-Bit

Table H.1: List of Benchmark and Development Systems

[1] The node offers 128 GB memory for scalability tests. To keep the HPL and the DGEMM results comparable with other systems, all single-GPU runs were restricted to 64 GB matrices where not explicitly stated otherwise.

[2] For some DMA comparison tests, the 6970 were exchanged by 5870 GPUs. For power efficiency tests, also V7800 GPUs were employed. In 2012, the node was upgraded with 7970 GPUs.

[3] In turbo mode the CPU clocks up to 3.7 GHz.

Appendix I

Source Codes

This appendix lists references on where the source code of all programs developed during this thesis can be obtained:

ALICE HLT TPC GPU Tracker: The standalone version can be obtained via subversion from: http://qon.zapto.org/var/svn/catracker/standalone. For running the tracker in the ALICE Off-line Project (http://aliweb.cern.ch/Offline/), the cagpu library is needed, which can be obtained from http://qon.zapto.org/var/svn/catracker/standalone/cagpubuild.

CALDGEMM / AMD Binary Driver Patch / GotoBLAS Patch / GCC OpenMP Patch: The patches for GotoBLAS, for the GCC OpenMP library (libgomp), and the binary patch for the AMD Driver are bundled with CALDGEMM in a git repository: git://code.compeng.uni-frankfurt.de/caldgemm.git.

The project page is located at http://code.compeng.uni-frankfurt.de/projects/caldgemm.

HPL-GPU: HPL-GPU is hosted via git at: git://code.compeng.uni-frankfurt.de/hpl.git (See also http://code.compeng.uni-frankfurt.de/projects/caldgemm).

QEnc: The QEnc encoding library is available via subversion from: http://qon.zapto.org/var/svn/qenc.

DMA / MPI Benchmark Suite: The DMA benchmark suite for PCIe bandwidth measurements, verification of full duplex asynchronous DMA capabilities, and MPI benchmarks is available at http://qon.zapto.org/var/svn/benchsuite.

List of Figures

List of Tables

List of Listings

Index

Glossary

ACML	AMD Core Math Library
ADL	AMD Display Library
ALICE	A Large Ion Collider Experiment
ALU	Arithmetical Logical Unit
AMD	Advanced Micro Devices
AOp	Arithmetical Operation
API	Application Programming Interface
ATLAS	A Toroidal LHC Apparatus
AVX	Advanced Vector Instructions
BGEMM	Binary General Matrix Multiplication with Matrix
BLAS	Basic Linear Algebra Subprograms
CAL	Compute Abstraction Layer
CERN	Conseil Européenne pour la Recherche Nucléaire
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
CRS	Cauchy-Reed-Solomon
CSC	Center for Scientific Computing
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DAQ	Data Acquisition
DAXPY	Double Precision Alpha X Plus Y
DCOPY	Double Precision Copy Vector
DDR	Double Data Rate
DGEMM	Double Precision General Matrix Multiplication with Matrix
DGEMV	Double Precision General Matrix Multiplication with Vector
DIMM	Dual In-line Memory Module
DLACPY	Double Precision Linear Algebra Matrix Copy
DLATCPY	Double Precision Linear Algebra Matrix Transposed Copy
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSCAL	Double Precision Scale Vector
DSP	Digital Signal Processor
DTRSM	Double Precision Triangular System Solver versus Matrix
DTRSV	Double Precision Triangular System Solver versus Vector
DTRTRI	Double Precision Triangular System Inversion
ECC	Error Correction Code
FDR	Fourteen Data Rate
FIFO	First In First Out
Flop	Floating Point Operation
FMA	Fused Multiply Add
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GDDR	Graphics DDR Memory

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HCA Host Channel Adapter

HLT High Level Trigger

HPC High Performance Computing

HPL High Performance Linpack

ICC Intel C++ Compiler

IEEE Institute of Electrical and Electronics Engineers

IGEMM Integer General Matrix Multiplication with Matrix

IL Intermediate Assembler Language

ISA Instruction Set Architecture

ITS Inner Tracking System

LAN Local Area Network

LAPACK Linear Algebra PACKage

LASWP Linear Algebra Swap

LDS Local Data Storage

LHC Large Hadron Collider

LOEWE Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz

LRU Least Recently Used

LUT LookUp Table

MDS Maximum Distance Separable

MIC Many Integrated Cores

MIMD Multiple Instruction Multiple Data

MKL Math Kernel Library

MPI Message Passing Interface

NUMA Non Uniform Memory Architecture

Op Operation

OpenCL Open Compute Language

OpenGL Open Graphics Language

OpenMP Open MultiProcessing

PCI Peripheral Component Interconnect

PCIe PCI Express

PDTRSV Panel Double Precision Triangular System Solver versus Vector

PTX Parallel Thread Execution

PUE Power Usage Effectiveness

QA Quality Assurance

QDR Quad Data Rate

QGP Quark Gluon Plasma

RAID Redundant Array of Independent/Inexpensive Disks

RAM Random Access Memory

RDMA Remote Direct Memory Access

RHS Right Hand Side

RMS Root Mean Square

SDK Software Development Kit

SDS Splitted Desktop Systems

SGEMM Single Precision General Matrix Multiplication with Matrix

SIMD Single Instruction Multiple Data

SSE Streaming SIMD Extensions

TBB Threading Building Blocks

TCO Total Cost of Ownership

TDP Thermal Design Power

TLB Translation Lookaside Buffer

TPC Time Projection Chamber

USB Universal Serial Bus

VGA Visual Graphics Adapter

VHDL VHSIC Hardware Description Language

VHSIC Very-High-Speed Integrated Circuits

VLIW Very Long Instruction Word

XOR Exclusive Or

Acknowledgements

I would like to thank all who contributed directly or indirectly to this thesis.

First of all my supervisor Prof. Volker Lindenstruth. I always felt that he put confidence in me and I had all the freedom I could have wanted to realize my ideas. When I was stuck in a point, a discussion usually showed up new ways to go.

I would like to thank Prof. Udo Kebschull for being the second assessor. Prof. Hans Jürgen Lüdde was my second supervisor in the HGS-Hire programm and, not being directly involved in the projects, was an excellent consultant if an external view was needed.

I own special thanks to Sergey Gorbunov, who invented the original tracking algorithm, helped a lot implementing the GPU tracker, had plenty of great ideas, and proofread parts of my thesis.

Matthias Bach helped a lot with server administration and with his profound knowledge about GPUs he was usually the first one to ask when my kernels did not behave as expected. I would also like to thank him for proofreading.

I want to thank Matthias Kretz, whose experience with Linux and C++ I could profit from to a huge extent. I would like to thank him for proofreading as well.

Sebastian Kalcher was seriously involved in deploying and installing the Sanam cluster at GSI. Without him the second rank in the Green500 had been impossible.

For installing the GPU tracker at CERN, I had great support from Timm Steinbeck, Torsten Alt, Thorsten Kollegger, Timo Breitner, Artur Szostak, and Olav Smorholm. I also thank Thorsten for Proofreading. Camilla Stokkevåg helped me out with the QA plots.

Thanks go to Andreas Ertelt, Jan de Cuveland, and Oliver Thomas for proofreading.

I would like to express my thanks to Udeepta Bordoloi who helped a lot with driver and hardware problems and also proofread parts of this thesis. Rod Macdonald provided much Linux support for AMD GPUs.

Timothy Lanfear put quite some effort in having the GPU tracker compiled with the Fermi CUDA Framework in the first place.

I would like to thank Petr Borodkin, Alex Tutubalin, Jean-Marie Verdun, Peyman Blumstengel, Zhongze Li, Hermann von Drateln, and Marc Romankewicz for their feedback to my work and for their support to the projects. Especially Petr spent quite some time finding good CPU mappings for CALDGEMM.

Jörg and Nadine Körner have been a reliable assistance for design and photography.

I would like to thank AMD, SuperMicro, Samsung, Splitted Desktop Systems, Adtech, ASUS and Rombo for their support to all projects.

Katharina Hübner provided many good ideas for the coding theory and I want to thank her for proofreading.

I would like to thank the HLT crew and all people who helped installing the LOEWE-CSC and the Sanam cluster. Without them many projects would not have been possible.

In the end, I would like to thank my family and all my friends who were always supporting me.

Bibliography

[Adv I] Advanced Micro Devices: "AMD Core Math Library".
URL: http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/

[Adv II] Advanced Micro Devices: "AMD Display Library (ADL) SDK".
URL: http://developer.amd.com/sdks/adlsdk/pages/default.aspx

[Adv 09] Advanced Micro Devices: "AMD Stream Computing Guide", [2009].

[Adv 10 I] Advanced Micro Devices: "3DNow!™ Instructions are Being Deprecated", [2010].
URL: http://blogs.amd.com/developer/2010/08/18/3dnow-deprecated/

[Adv 10 II] Advanced Micro Devices: "ATI Compute Abstraction Layer Intermediate Language Reference Manual", [2010].

[Adv 10 III] Advanced Micro Devices: "ATI Evergreen Family Instruction Set Architecture Reference Manual", [2010].

[Adv 12] Advanced Micro Devices: "AMD FirePro Server Graphics Redefining Supercomputing Performance through Power Efficient, Sustainable Technology", Press Release 14.11.2012 [2012].
URL: http://www.amd.com/us/press-releases/Pages/amd-firepro-server-2012nov14.aspx

[ALI I] ALICE Collaboration: "Alice Experiment Offline Project".
URL: http://aliceinfo.cern.ch/Offline/AliRoot/Manual.html

[ALI II] ALICE Collaboration: "ALICE Home Page".
URL: http://aliceinfo.cern.ch/

[ALI III] ALICE Collaboration: "ALICE Time Projection Chamber, the homepage of the ALICE TPC".
URL: http://aliceinfo.cern.ch/TPC

[ALI 04] ALICE Collaboration: "ALICE Technical Design Report of the Trigger, Data Acquisition, High-Level Trigger, and Control System", *Tech. rep.* [2004].
URL: https://edms.cern.ch/document/456354/2

[ALI 95] ALICE Collaboration: "Technical proposal for A Large Ion Collider Experiment at the CERN LHC", *Tech. rep.*, CERN [1995].
URL: http://cdsweb.cern.ch/record/293391/files/cer-000214817.pdf

[Amd 67] G. Amdahl: "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *in AFIPS Conference Proceedings, vol. 30, pp. 483–485* [1967].

[And+ 05] A. A. D. Andrade, R. Palazzo Jr.: "Linear Codes over Finite Rings", *TEMA Tend. Mat. Apl. Comput., vol. 2, no. 2: pp. 207–217* [2005].

[AT 67] E. Artin, J. Tate: "Class Field Theory" [W. A. Benjamin, Inc., 1967], ISBN 0-8053-0291-3.

[Bac 09] M. Bach: "Utilization of Graphics Processing Units in Applications for High Energy Physics", *Diploma thesis*, University of Heidelberg [2009].

[Bac+ 11 I] M. Bach, M. Kretz, V. Lindenstruth, D. Rohr: "Optimized HPL for AMD GPU and Multi-Core CPU Usage", *Computer Science - Research and Development, vol. 26, no. 3-4* [2011].
URL: http://www.springerlink.com/content/m232n83h73271228/

[Bac+ 11 II] M. Bach, O. Philipsen, C. Pinke, C. Schäfer, L. Zeidlewicz: "LatticeQCD using OpenCL", *in proceedings of the XXIX International Symposium on Lattice Field Theory - Lattice 2011, p. 7* [2011].
URL: http://arxiv.org/abs/1112.5280

[Bac+ 13] M. Bach, J. De Cuveland, H. Ebermann, D. Eschweiler, M. Kretz, M. Pollok, D. Rohr, H. J. Lüdde, V. Lindenstruth: "The LOEWE-CSC: A Comprehensive Approach for a Power Efficient General Purpose Supercomputer", *in 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 1–17* [2013].

[Bla 12] B. Bland: "Titan - Early Experience with the Titan System at Oak Ridge National Laboratory", *SC12 Keynote Slides* [2012].
URL: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/BuddyBland_Titan_SC12.pdf

[Blö+ 95] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, D. Zuckerman: "An XOR-Based Erasure-Resilient Coding Scheme", [1995].

[Bör 11] J. Börger: "Enabling the ALICE High-Level Trigger Event Reconstruction for OpenCL", *Master thesis*, Goethe University of Frankfurt [2011].

[Bra+ 12] A. Branover, D. Foley, M. Steinman: "AMD Fusion APU: Llano", *Ieee Micro, vol. PP, no. 2: p. 1* [2012], ISSN 02721732, doi:10.1109/MM.2012.2.
URL: http://www.computer.org/csdl/mags/mi/2012/02/mmi2012020028-abs.html

[Brü+ 04] O. S. Brüning, P. Collier, P. Lebrun, S. Myers, R. Ostojic, J. Poole, P. Proudlock: "LHC Design Report", *Tech. rep.*, CERN [2004].
URL: http://lhc.web.cern.ch/LHC/LHC-DesignReport.html

[CER] CERN: "ROOT - Architectural Overview".
URL: http://root.cern.ch/drupal/content/architectural-overview

[CER 06] CERN: "Overall view of the LHC - CERN Document Server", [2006].
URL: http://cds.cern.ch/record/987579?ln=en

[CER 11] CERN: "Events recorded by ALICE from the first lead ion collisions in 2011", [2011].
URL: https://cdsweb.cern.ch/record/1400435

[Cop 10] A. Copenbarger: "Illinois wins greenest self-built cluster at SC10", [2010].
URL: http://www.ncsa.illinois.edu/News/Stories/GreenGPU/

[Cop+ 87] D. Coppersmith, S. Winograd: "Matrix multiplication via arithmetic progressions", *Proceedings of the nineteenth annual ACM conference on Theory of computing, vol. 9, no. 3: pp. 1–6* [1987], doi:10.1145/28395.28396.
URL: http://dl.acm.org/citation.cfm?id=28396

[Dal 10] B. Dally: "GPU computing to exascale and beyond", *SC10 keynote slides* [2010].

[Don+ 03] J. J. Dongarra, P. Luszczek, A. Petitet: "The LINPACK Benchmark: past, present and future", *Concurrency and Computation: Practice and Experience, vol. 15, no. 9: pp. 803–820* [2003], ISSN 1532-0626, doi:10.1002/cpe.728.

[Don+ 90] J. Dongarra, J. D. Croz, S. Hammarling: "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software, vol. 16, no. 1: pp. 1–17* [1990], ISSN 00983500, doi:10.1145/77626.79170.
URL: http://portal.acm.org/citation.cfm?doid=77626.79170

[Dre 07] U. Drepper: "What Every Programmer Should Know About Memory", [2007].
URL: http://www.akkadia.org/drepper/cpumemory.pdf

[ECO] ECO Data Recovery: "Seagate Data Recovery".
URL: http://ecodatarecovery.wordpress.com/tag/seagate-data-recovery/

[Eme+ 12] D. Emeliyanov, J. Howard: "GPU-Based Tracking Algorithms for the ATLAS High-Level Trigger", *Journal of Physics: Conference Series, vol. 396, no. 1: p. 012018* [Dec. 2012], ISSN 1742-6588, doi:10.1088/1742-6596/396/1/012018.
URL: http://stacks.iop.org/1742-6596/396/i=1/a=012018?key=crossref.df9ce06ac107615d9c97381116e9f3ef

[End+ 10] T. Endo, S. Matsuoka, A. Nukada, N. Maruyama: "Linpack evaluation on a supercomputer with heterogeneous accelerators", *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS, pp. 1–8* [2010], ISSN 15302075, doi:10.1109/IPDPS.2010.5470353.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470353

[Fen 05] W.-c. Feng: "The Importance of Being Low Power in High Performance Computing", *CTWatch Quarterly, vol. 1, no. 3* [2005].
URL: http://www.ctwatch.org/quarterly/articles/2005/08/the-importance-of-being-low-power-in-high-performance-computing/

[Fen+ 10] W.-c. Feng, K. W. Cameron: "The Green500 List :: Environmentally Responsible Supercomputing :: The Green500", [2010].
URL: http://www.green500.org/greenlists

[Fra 11] Frankfurt Institute for Advanced Studies: "Frankfurter Superrechner-Technologie passt auf jeden Schreibtisch Weltrekord für französische Workstation durch LOEWE-CSC-Energiespartechnik", Press Release 22.6.2011 [2011].

[Frü+ 00] R. Frühwirth, M. Regler: "Data analysis techniques for high-energy physics" [Cambridge University Press, 2000], ISBN 0521635489.

[Fun+ 13] D. Funke, T. Hauth, V. Innocente: "CMS Track Reconstruction Investigation of OpenCL and Cellular-Automata", [2013].
URL: http://indico.cern.ch/getFile.py/access?contribId=3&resId=0&materialId=slides&confId=235844

[Goe] Goethe University of Frankfurt Center for Scientific Computing: "LOEWE-CSC Cluster".
URL: http://csc.uni-frankfurt.de/index.php?id=51
URL: http://compeng.uni-frankfurt.de/index.php?id=86&L=1

[Gor+ 08] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, W. F. J. Müller: "Fast SIMDized Kalman filter based track fit", *Computer Physics Communications, vol. 178: pp. 374–383* [2008].

[Gor+ 11] S. Gorbunov, D. Rohr, K. Aamodt, T. Alt, H. Appelsh, A. Arend, M. Bach, B. Becker, T. Breitner, et al.: "ALICE HLT High Speed Tracking on GPU", *IEEE Transactions on Nuclear Science, vol. 58, no. 4* [2011].

[Gor 12] S. Gorbunov: "On-line reconstruction algorithms for the CBM and ALICE experiments", *Dissertation thesis*, Goethe University of Frankfurt [2012].

[Hei+ 13] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, P. Dubey: "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel® Xeon Phi Coprocessor", *2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 126–137* [May 2013], doi:10.1109/IPDPS.2013.113.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6569806

[Hid+ 01] Y. Hida, X. S. Li, D. H. Barley: "Algorithms for Quad-Double Precision Floating Point Arithmetic", *15th IEEE Symposium on Computer Arithmetic, pp. 155–162* [2001].

[Int] Intel Corporation: "Intel Threading Building Blocks Reference Manual".
URL: http://www.threadingbuildingblocks.org/

[Int 08] Intel Corporation: "Larrabee: A Many-Core x86 Architecture for Visual Computing", *ACM Transactions on Graphics, vol. 27* [2008].
URL: http://software.intel.com/file/18198/

[Int 10] Intel Corporation: "Intel Unveils New Product Plans for High-Performance Computing", Press Release 31.5.2010 [2010].
URL: http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm

[Int 11] Intel Corporation: "Intel ® 64 and IA-32 Architectures Optimization Reference Manual", [2011].

[KAC] King Abdulaziz City for Science and Technology: "Official Website".
URL: http://www.kacst.edu.sa/

[Kal+ 11] S. Kalcher, V. Lindenstruth: "Accelerating Galois Field Arithmetic for Reed-Solomon Erasure Codes in Storage Applications", *in 2011 IEEE International Conference on Cluster Computing, pp. 290–298* [IEEE, 2011], ISBN 9781457713552, doi:10.1109/CLUSTER.2011.40.
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6061147

[Kal 13] S. Kalcher: "An Erasure-Resilient and Compute-Efficient Coding Scheme for Storage Applications", *Dissertation thesis*, Goethe University of Frankfurt [2013].

[Kal 60] R. E. Kalman: "A new approach to linear filtering and prediction problems", *Journal Of Basic Engineering, vol. 82 Series D: pp. 35–45* [1960], doi:10.1109/ICASSP.1982.1171734.
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.6247&rep=rep1&type=pdf

[Kam+ 08] S. Kamil, J. Shalf, E. Strohmaier: "Power Efficiency in High Performance Computing", *Proc of the 2008 IEEE Intl Symp on Parallel and Distributed Processing, pp. 1–8* [2008], ISSN 15302075, doi:10.1109/IPDPS.2008.4536223.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4536223

[Khr] Khronos Group: "OpenCL Specifications".
URL: http://www.khronos.org/opencl/

[Knu 97] D. E. Knuth: "Seminumerical algorithms", *vol. 2* of *The art of computer programming* [Addison-Wesley Longman, 1997], ISBN 978-0201896848.
URL: http://adsabs.harvard.edu/abs/1981acp..book.....K

[Kre 09] M. Kretz: "Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading", *Diploma thesis*, University of Heidelberg [2009].

[Kre+ 11] M. Kretz, V. Lindenstruth: "Vc: A C++ library for explicit vectorization", *Software Practice and Experience* [2011], ISSN 1097024X, doi:10.1002/spe.1149.
URL: http://dx.doi.org/10.1002/spe.1149

[Kur+ 12] J. Kurzak, P. Luszczek, M. Faverge, J. Dongarra: "LU Factorization with Partial Pivoting for a Multicore System with Accelerators", *IEEE Transactions on Parallel and Distributed Systems, vol. 6, no. 1: pp. 1–1* [2012], ISSN 1045-9219, doi:10.1109/TPDS.2012.242.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6280548

[Lan 05] S. Lang: "Algebra (Graduate Texts in Mathematics)" [Springer, 2005], ISBN 978-0387953854.

[Man 04] R. Mankel: "Pattern Recognition and Event Reconstruction in Particle Physics Experiments", *Rep. Prog. Phys. 67, pp. 553–622* [2004].
URL: http://arxiv.org/abs/physics/0402039

[Mat+ 10] S. Matsuoka, T. Endo, N. Maruyama, H. Sato, S. Takizawa: "The Total Picture of TSUBAME 2.0", *TSUBAME e-Science Journal, vol. 1: pp. 16–18* [2010].

[McC 95] J. D. McCalpin: "STREAM: Sustainable Memory Bandwidth in High Performance Computers", [1995].
URL: http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/

[Mel 10] Mellanox Technologies: "ConnectX® Single/Dual-Port Adapter Cards supporting up to 40Gb/s InfiniBand", [2010].
URL: http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX_VPI.pdf

[Meu+] H. Meurer, E. Strohmaier, J. Dongarra, H. Simon: "Top 500 Supercomputing Sites".
URL: http://top500.org

[MPI] MPI Forum: "Message Passing Interface Standard".
URL: http://www.mpi-forum.org/docs/docs.html

[MS] Microsoft: "C++ Accelerated Massive Parallelism (C++ AMP)".
URL: http://msdn.microsoft.com/en-us/library/hh265137(v=vs.110).aspx

[Nak 10] N. Nakasato: "A Fast GEMM Implementation On a Cypress GPU", *1st International Workshop on Performance Modeling Benchmarking and Simulation of High Performance Computing Systems PMBS 10* [2010].
URL: http://dl.acm.org/citation.cfm?id=1964227

[Neu 99] J. Neukirch: "Algebraic number theory" [Springer, 1999], ISBN 3-540-65399-6.

[NSW 08] J. Neukirch, A. Schmidt, K. Wingberg: "Cohomology of number fields", *vol. 323* of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]* [Springer, 2008], ISBN 9783540378884.

[NVI] NVIDIA Corporation: "CUBLAS library".
URL: http://docs.nvidia.com/cuda/cublas/index.html

[NVI 11] NVIDIA Corporation: "CUDA C Programming Guide 4.0 (June 2011)", [2011].
URL: http://docs.nvidia.com/cuda/index.html

[NVI 12] NVIDIA Corporation: "CUDA PTX: Parallel Thread Instruction ISA Version 3.1", [2012].
URL: http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf

[OMP] OpenMP Architecture Review Board: "The OpenMP® API specification for parallel programming".
URL: http://www.openmp.org

[Pla 05] J. S. Plank: "Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications", *Tech. rep.*, Citeseer [2005].
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.8887&rep=rep1&type=pdf

[Pla 97] J. S. Plank: "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems", *Software Practice and Experience, vol. 27, no. 9: pp. 995–1012* [1997].

[Pre+ 92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: "Numerical Recipes in C: The Art of Scientific Computing", *vol. 1* of *ISBN 0-521-43108-5* [Cambridge University Press, 1992], ISBN 0521431085, doi:10.2307/1269484.

[Ree+ 60] I. S. Reed, G. Solomon: "Polynomial codes over certain finite fields", *Journal of the Society for Industrial and Applied Mathematics, vol. 8, no. 2: pp. 300–304* [1960], ISSN 03684245, doi:10.1137/0108018.
URL: http://www.jstor.org/stable/2098968

[Roh 10 I] D. Rohr: "ALICE TPC Online Tracking on GPU based on Kalman Filter", *Diploma thesis*, University of Heidelberg [2010].

[Roh+ 10 II] D. Rohr, M. Kretz, M. Bach: "Technical Report, CALDGEMM and HPL", *Tech. rep.*, University of Frankfurt [2010].

[Roh+ 11] D. Rohr, M. Bach, M. Kretz, V. Lindenstruth: "Multi-GPU DGEMM and HPL on Highly Energy Efficient Clusters", *IEEE Micro, Special Issue, CPU, GPU, and Hybrid Computing* [2011], doi:10.1109/MM.2011.66.

[Roh 12 I] D. Rohr: "ALICE TPC Online Tracker on GPUs for Heavy-Ion Events", *in 13th International Workshop on Cellular Nanoscale Networks and their Applications, pp. 298–303* [2012].

[Roh+ 12 II] D. Rohr, S. Gorbunov, A. Szostak, M. Kretz, T. Kollegger, T. Breitner, T. Alt: "ALICE HLT TPC Tracking of Pb-Pb Events on GPUs", *Journal of Physics: Conference Series, vol. 396, no. 1: p. 12044* [2012].
URL: http://stacks.iop.org/1742-6596/396/i=1/a=012044

[Sch+ 11] C. Schmitt, The Atlas Collaboration: "Track finding using GPUs", *in 14th International Workshop On Advanced Computing And Analysis Techniques In Physics Research*, [2011].
URL: http://cdsweb.cern.ch/record/1379496

[Sch 59] S. Schechter: "On the inversion of certain matrices", *Mathematics of Computation, vol. 13, no. 66: pp. 73–73* [May 1959], ISSN 0025-5718, doi:10.1090/S0025-5718-1959-0105798-2.
URL: http://www.ams.org/jourcgi/jour-getitem?pii=S0025-5718-1959-0105798-2

[Sco+ 10] T. R. W. Scogland, H. Lin, W.-c. Feng: "A first look at integrated GPUs for green high-performance computing", *Computer Science Research and Development, vol. 25, no. 3: pp. 125–134* [2010], ISSN 18652034, doi:10.1007/s00450-010-0128-y.
URL: http://www.springerlink.com/index/10.1007/s00450-010-0128-y

[SDS+ 11] SDS, CEA, FIAS, AMD: "mBox and mBlade : GPGPU computing pushed to the low power world . 1GFlops / Watt barrier is now the past .", Press Release 22.6.2011 [2011].
URL: http://www.splitted-desktop.com/

[Ser 98] G. Seroussi: "Table of Low-Weight Binary Irreducible Polynomials", *Tech. rep.*, Hewlett Packard [1998].
URL: http://www.hpl.hp.com/techreports/98/

[Sha+ 06] S. Sharma, C.-H. Hsu, W.-C. Feng: "Making a case for a Green500 list", *in Proceedings 20th IEEE International Parallel Distributed Processing Symposium, p. 343* [IEEE, 2006].
URL: http://www.green500.org/resources/pubs/making-case-green500-list

[Sha 48] C. E. Shannon: "The mathematical theory of communication", *Bell System Technical Journal, vol. 27 Series The mathematical theory of communication: pp. 379–423* [1948], ISSN 00058580.

[Shi 13] G. Shipman: "The Titan Supercomputer", [2013].

[Ste 04] T. Steinbeck: "A Modular and Fault-Tolerant Data Transport Framework", *Dissertation thesis*, University of Heidelberg [2004].

[Ste+ 10] T. Steinke, K. Peter, S. Borchert: "Efficiency Considerations of Cauchy Reed-Solomon Implementations on Accelerator and Multi-Core Platforms", [2010].
URL: http://saahpc.ncsa.illinois.edu/10/papers/paper_12.pdf

[Str 69] V. Strassen: "Gaussian Elimination is not Optimal", *Numerische Mathematik, vol. 13: pp. 354–356* [1969].

[TAC] Texas Advanced Computing Center: "GotoBLAS Basic Linear Algebra Library".
URL: http://www.tacc.utexas.edu/tacc-projects/gotoblas2

[Tur 66] L. R. Turner: "Inverse of the Vandermonde matrix with applications", [1966].
URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.2019

[UoT] University of Tennesse: "High Performance Linpack Algorithm".
URL: http://www.netlib.org/benchmark/hpl/algorithm.html

[Vol+ 08] V. Volkov, J. Demmel: "Benchmarking GPUs to Tune Dense Linear Algebra", *in SC '08 ACM/IEEE conference on Supercomputing Proceedings, pp. 1–11* [2008].

[Wan+ 11] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, W.-X. Xu: "Optimizing Linpack Benchmark on GPU-Accelerated Petascale Supercomputer", *Journal of Computer Science and Technology, vol. 26, no. 5: pp. 854–865* [2011], ISSN 1000-9000.
URL: http://dx.doi.org/10.1007/s11390-011-0184-1

[Yas] Yasm Development Team: "The Yasm Modular Assembler Project".
URL: http://yasm.tortall.net/

Zusammenfassung

Motivation

Seit den siebziger Jahren folgte die Entwicklung integrierter Schaltkreise dem von Gordon Moore formulierten Gesetz, welches besagt, dass sich die Anzahl der Transistoren und damit grob gesagt auch die Leistungsfähigkeit von Computern alle achtzehn Monate verdoppelt. Zu Begin des einundzwanzigsten Jahrhunderts begann die Taktrate neuer Prozessoren zu stagnieren, da die Wärmeentwicklung das höchstmögliche Niveau erreicht hatte, das noch mit verhältnismäßigem Aufwand beherrschbar ist. Darüber hinaus ist der Stromverbrauch moderner Großrechner zu einem gravierenden Kostenfaktor geworden. Um das Verlangen der Anwender, die von der gewohnten Steigerung der Rechenleistung ausgehen, nach schnelleren Computern zu stillen, haben Hardwarehersteller begonnen, andere Wege zu gehen, ihre Chips zu verbessern. Dies umfasst die Steigerung der pro Takt verrichteten Arbeit, was aber nur in relativ geringem Umfang möglich ist, und insbesondere den Umstieg auf breit angelegte parallele Architekturen.

In den letzten Jahren haben parallel arbeitende Computer zu einer gewaltige Verbreitung gefunden. Jeder neuere Prozessor verfügt über mehrere Kerne und einfache Vektorinstruktionen. Leistungsfähiger und energieeffizienter sind Many-Core Prozessoren, wie z. B. Grafikkarten (**GPUs**), die mit sehr vielen parallel arbeitenden einfach gestrickten Kernen Hunderte von Aufgaben gleichzeitig bearbeiten, oder auch Prozessoren mit breiten Vektorregistern wie beispielsweise Intels Xeon Phi. Für parallel ausgelegte Programme bieten diese eine vormals nicht für möglich gehaltene Rechenleistung. Im Gegenzug sind sie jedoch konstruktionsbedingt ungeeignet für konventionelle seriell arbeitende Programme. Leider kann man deren Potenzial auch nicht durch einfaches Neukompilieren alter Software ausschöpfen.

Die Anpassung erfordert häufig ein Durchdenken des und Modifikationen am Algorithmus, um die mögliche Parallelität auszunutzen. Oftmals ist dies kompliziert, zeitaufwendig und die Rentabilität schwer abschätzbar. Viele Programmierer scheuen noch neuartige Prozessoren und viele Stimmen raten von deren Nutzung ab, weil sie schwer und nicht effizient programmierbar seien, und dies ihre Vorteile nivelliere.

Häufig lassen sich Anwendungen in serielle und parallele Unterroutinen aufteilen, wobei erstere schwer, gar nicht oder nur teilweise parallelisierbar sind. Daher sind viele moderne Cluster heterogen aufgebaut, mit breiten parallelen Prozessoren für parallele und schnellen für serielle Aufgaben. Geschickte Programmierung kann oft die langsamen seriellen Teile mit den parallelen überlappen und so deren Ausführungszeit verstecken. In jedem Fall sind jedoch hocheffiziente Programme obligatorisch, um nicht einen Großteil der Rechenleistung zu verschenken.

Diese Arbeit setzt sich mit drei Themen auseinander: Entwicklung schneller Algorithmen und deren Implementierung auf modernen Prozessoren und Grafikkarten, maximal erzielbare Effizienz in Bezug auf die spezifizierte Maximalrechenleistung und die Leistungsaufnahme sowie drittens Umsetzbarkeit und Grenzen von Programmen für Prozessoren, Grafikkarten und heterogene Systeme. Zu diesem Zweck werden drei völlig unterschiedliche Programme aus verschiedenen Gebieten, die im Umfang der Arbeit realisiert wurden, vorgestellt, analysiert und verglichen.

Ereignisrekonstruktion für ALICE

Der **L**arge **H**adron **C**ollider (LHC) der Europäischen Organisation für Kernforschung (CERN) in Genf ist momentan der weltweit leistungsstärkste Teilchenbeschleuniger. **A** **L**arge **I**on **C**ollider **E**xperiment (ALICE) ist eines der vier großen Experimente, die am LHC installiert sind, und dient hauptsächlich dem Studium von Schwerionenkollisionen. Hierbei werden die Kerne von Bleiatomen nahezu auf Lichtgeschwindigkeit beschleunigt und aufeinander geschossen. Die dabei erreichte Temperatur ist hoch genug, um ein so genanntes **Q**uark **G**luon **P**lasma zu erzeugen: eine Form der Materie, bei der Quarks und Gluonen nicht in Hadronen gebunden sind, sondern sich frei bewegen können, und die vermutlich wenige Sekundenbruchteile nach dem Urknall existierte. Eine der größten Herausforderungen bei der Analyse der Kollisionen besteht in der Rekonstruktion der Trajektorien (**Tracking**) abertausender Teilchen, die bei jedem Zusammenstoß entstehen. ALICE besitzt mehrere Detektoren zur Spurerkennung, der wichtigste ist die **T**ime **P**rojection **C**hamber (TPC): eine zylindrische mit Gas gefüllte Kammer. Die durchfliegenden Teilchen ionisieren Gasmoleküle, wobei man die Ionisierungspunkte (**Hits**) messen kann. Die Aufgabe der Spurrekonstruktion besteht nun darin, aus der gewaltigen Menge gemessener dreidimensionaler Raumpunkte (Hits) die Teilchenspuren zu rekonstruieren, die Messpunkte den einzelnen Teilchen zuzuordnen und die physikalischen Teilcheneigenschaften anhand der Flugbahn zu bestimmen.

Typische zentrale Blei-Ereignisse sind mehrere zehn Megabyte groß und werden mit einer Rate von einigen hundert Hz gemessen, was einer eingehenden Datenrate von bis zu 30 $^{\mathrm{GB}}/_{\mathrm{s}}$ entspricht, rechentechnisch eine enorme Herausforderung. Der ALICE **H**igh **L**evel **T**rigger (HLT) ist eine Rechenfarm aus circa 250 Computern, die in Echtzeit einen Großteil der Ereignisrekonstruktion durchführt. Aufgaben des HLT bestehen in einer schnellen Analyse der Daten, einer Entscheidung welche Daten gespeichert werden und einer Kompression ebendieser Daten. Die gespeicherten Daten werden später von der ALICE Offline Software ausgewertet, die genauere und mehr Analysen liefert, aber dafür wesentlich mehr Zeit in Anspruch nimmt. Um eine Spurrekonstruktion in Echtzeit zu ermöglichen, wurde der ALICE HLT TPC Tracking Algorithmus angepasst und auf NVIDIA Fermi Grafikkarten portiert. Nach einer Testphase wurden 64 HLT-Server mit GPUs ausgestattet und der Tracker dort installiert. Der GPU Tracker war während des gesamten Jahres 2012 ohne besonderen Vorkommnisse im Dauereinsatz und stellte ein vollständiges Tracking aller von ALICE aufgenommenen Events bereit. Abb. 1 zeigt einen Screenshot des Online Event Displays im ALICE Kontrollraum während des ersten Betriebs des GPU Trackers.

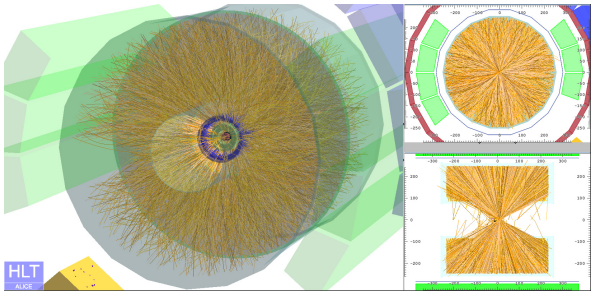


Abbildung 1: Screenshot des Online Event Display während des Ersten GPU Tracker Einsatzes

Zur besseren Ausnutzung der Datenlokalität teilt der HLT TPC Tracker das TPC Volumen in 36 gleichförmige **Sektoren** ein, die unabhängig voneinander bearbeitet werden. Die zueinander gehörenden, in den jeweiligen Sektoren gefundenen Tracksegmente werden danach durch den **Track Merger** zu vollständigen Tracks zusammengefasst. Der Algorithmus funktioniert im Grunde intern wie folgt: Zuerst werden im **Neighbors Finder** mit einer Heuristik auf lokaler Ebene kombinatorisch **Seeds** gesucht: sehr kurze Trackkandidaten bestehend aus wenigen Hits. Anhand des bekannten Verhaltens geladener Teilchen in dem den Detektor umgebenden Magnetfeld, werden vom **Tracklet Constructor** die wahrscheinlichsten Trackparameter bestimmt. Diese umfassen die Flugbahn des Teilchens sowie einige physikalische Eigenschaften wie die Ladung. Mittels der Trackparameter wird die Flugbahn des Teilchens durch die gesamte TPC extrapoliert. Darauf aufbauend werden weitere Hits gesucht, die nahe der Flugbahn liegen und vermutlich zu dieser Trajektorie gehören, dem Track zugeordnet und zur Verbesserung der Abschätzung der

Trackparameter genutzt. Am Ende filtert der **Tracklet Selector** Spuren mit Problemen bei der Parameterbestimmung heraus. Der Tracker benutzt den **Kalman Filter** zur Extrapolation der Spuren und zur Errechnung der Spurparameter.

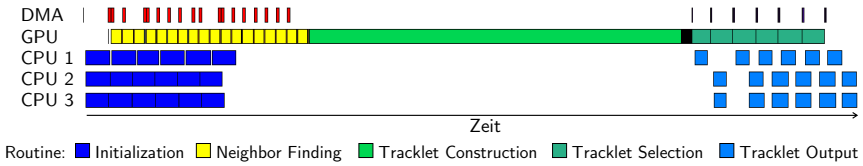


Abbildung 2: Pipeline des GPU Trackers mit mehrere CPU Threads

Einige am Tracking Algorithmus vorgenommene Optimierungen verkürzen die Bearbeitungszeit auf der GPU und verbessern die Resultate. Beispielsweise besitzt der GPU Tracker für die Datenein- und -ausgabe noch einige Routinen, die weiterhin auf dem Prozessor ablaufen, wobei eine Portierung auf die GPU wenig Sinn ergibt, da sie die Daten nur umformatieren und dabei lediglich ein- bis zweimal anfassen. Damit die GPU während diese Zeit und während der Zeit der Datenübertragung durch den PCI Express Bus nicht stillsteht, wird eine Pipeline eingesetzt, die das Tracking der einzelnen Sektoren nacheinander durchführt und dabei Vor- bzw. Nachbearbeitung auf der CPU, DMA Transfer und das eigentliche Tracking auf der GPU von aufeinanderfolgenden Sektoren zeitlich überlagert. Da ein Prozessorkern nicht schnell genug ist, um die GPU mit ausreichend Daten zu versorgen, arbeitet die Pipeline mit mehreren Kernen und nutzt diese reihum. Abb. 2 veranschaulicht den Vorgang.

An mehreren Stellen ging der Algorithmus ursprünglich nicht vollständig deterministisch vor. Beispielsweise wird an gewissen Punkten der jeweils längere Track bevorzugt. Im seltenen Fall von zwei Tracks gleicher Länge ist dieses Kriterium nicht eindeutig und der erste Track wurde gewählt. Dies hat auf die Qualität der Ergebnisse überhaupt keinen Einfluss und funktionierte in der CPU Version über Jahre hinweg tadellos. Bei der parallel arbeitenden Grafikkarte jedoch, die alle Tracks nebenläufig erstellt, ist die Reihenfolge der Tracks nicht wohldefiniert. Daher können bei obigem nichtdeterministischen Kriterium durch unvermeidliche Schwankungen in der Bearbeitungsreihenfolge leicht verschiedene Resultate herauskommen. Diese unterschiedlichen Ergebnisse sind physikalisch alle gleichwertig, erschweren allerdings die Qualitätssicherung ungemein, da man die Ausgabe von zwei Läufen nicht auf unterster Ebene vergleichen kann. Alle solche Effekte, die von Nebenläufigkeit herrühren, wurden eliminiert; meist durch Wahl besserer Kriterien, wodurch nebenbei sogar Effizienz und Auflösung des Trackers verbessert wurden. An anderer Stelle tritt leider noch eine kleine Schwankung auf, die auf Computern unvermeidlich ist: Da computerbasierte Fließkommaberechnung nicht assoziativ ist (falls die "-ffast-math" Compilereinstellung aktiv ist), führen unterschiedliche Programmoptimierungen verschiedener Compiler zu leicht unterschiedlichen Ergebnissen; Dieses Phänomen tritt auch auf dem Prozessor bei Verwendung unterschiedlicher Compilereinstellungen auf. Vergleicht man die Ausgabe der CPU- und GPU-Tracker auf Basis von Tracks und zugeordneten Hits, ergibt sich im Mittel eine Abweichung bei $0,00024\,\%$ der Hits und $0,012\,\%$ der Tracks – eine gänzlich vernachlässigbare Größenordnung.

Als letztes Beispiel soll eine prinzipbedingte Schwäche der Aufteilung der TPC in Sektoren dienen. Falls eine Spur größtenteils in einem Sektor liegt, aber ein sehr kurzes Stück in einen benachbarten Sektor hineinreicht, ist es wahrscheinlich, dass dieses zweite kurze Segment nicht gefunden wird. Hierfür wurde ein Feature mit Namen **Global Tracking** implementiert, das nach solchen Kandidaten sucht, und dann die wohldefinierten Spurparameter innerhalb des Sektors mit dem langen gefunden Segment nutzt, um die fehlenden Hits im angrenzenden Sektor zu finden.

Abb. 3 analysiert die Qualität der Trackingergebnisse und stellt diese dem Offline Tracker gegenüber, welcher von der ALICE Offline Gruppe zur nachträglichen Datenanalyse eingesetzt wird. Hierbei sind **Findable Primary** Tracks Spuren, die dem Primären Vertex, also dem Kollisions-

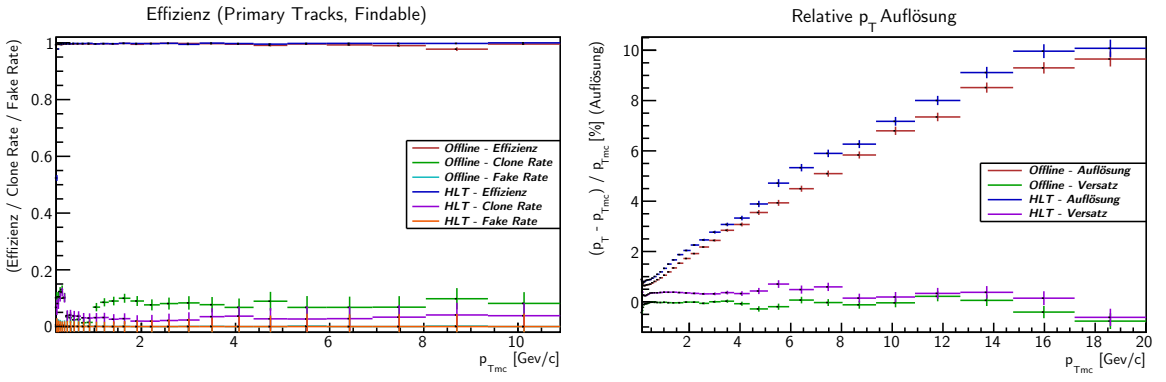


Abbildung 3: Qualität der Ergebnisse des GPU Trackers und des Offline Trackers

mittelpunkt, entstammen und gewissen von der ALICE Offline Gruppe definierten Kriterien für relevante Spuren genügen. Als Basis wird ein großer Satz simulierter Ereignisse genommen, deren echte Spuren bekannt sind, so dass diese mit den vom Tracker aus den simulierten Hits erzeugten Spuren verglichen werden können. Die Effizienz gibt den Anteil dieser Spuren wieder, die der Tracker gefunden hat. Die Clone Rate beschreibt den Anteil der Spuren, die doppelt gefunden aber nicht vom Merger zusammengefasst wurden. Die Fake Rate misst den Anteil der Spuren, die keiner realen Spur entsprechen und die Relative p_t Auflösung ist die relative Abweichung des vom Tracker berechneten Transversalimpulses vom simulierten Referenzwert.

Klar erkennbar erzielen beide Tracker quasi die höchstmögliche Effizienz (wobei der GPU Tracker sogar noch ein klein wenig besser ist) und die Fake Rate liegt bei null. Die Clone Raten sind sehr gering und beim GPU Tracker deutlich besser. Die Auflösung hingegen ist beim Offline Tracker genauer. Dies ist allerdings in keiner Hinsicht verwunderlich, da der HLT GPU Tracker eine schnelle Echtzeitrekonstruktion durchführt und der Offline Tracker zur späteren genauen Datenanalyse genutzt wird, wobei mehr Zeit und mehr Rechenleistung zur Verfügung stehen. Aus Performancegründen nimmt der HLT GPU Tracker deshalb ein paar Vereinfachungen vor, die die Echtzeitanalyse ermöglichen, sich aber hier in der Auflösung niederschlagen. In jedem Fall ist die erzielte Auflösung mehr als ausreichend für die Echtzeitanalyse. Zusätzlich wurden einige Korrekturen identifiziert, die der Offline Tracker anwendet, und die möglicherweise mit geringen Performanceeinbußen in Zukunft auch für den GPU Tracker implementiert werden können. Momentan wird von der HLT und der Offline Gruppe gemeinsam daran gearbeitet, den schnellen GPU Tracker mit der Offline Implementierung zur Bestimmung der Trackparameter zu verknüpfen, um so die Vorzüge beider Versionen zu kombinieren.

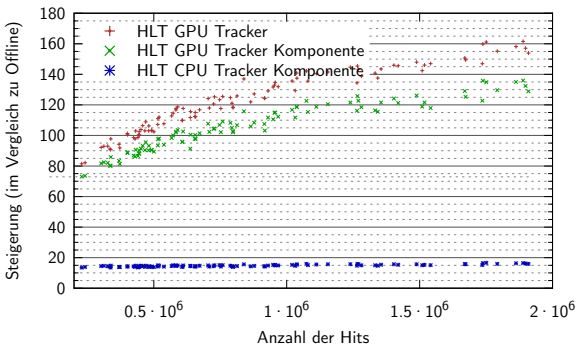


Abbildung 4: Vergleich der Performance von GPU, CPU und Offline Tracker

Vergleicht man die benötigte Rechenzeit der am HLT eingesetzten Grafikkarten vom Typ GTX480 und GTX580 mit den schnellsten momentan erhältlichen Prozessoren, so gewinnt die Grafikkarte, die obendrein günstiger ist, etwa um den Faktor drei. Für die Leistung im HLT ist eine andere Abschätzung sinnvoll. Da der HLT viele weitere prozessorintensive Arbeiten verrichtet (wie z. B. der Track Merger, Cluster Transformation und Suche nach sekundären Vertices), können ohnehin nicht alle CPU Ressourcen der Rechenknoten für die Spurrekonstruktion verwendet werden. Abb. 4 vergleicht daher die Rechenzeit des Offline Trackers und des HLT Trackers auf CPU und auf GPU, in Konfigurationen, die jeweils genau vier der CPU Kerne für die Spurrekonstruktion benutzen und alle verbleibenden Kerne für andere

Aufgaben freihalten. Die Messung für den HLT GPU Tracker berücksichtigt nur die eigentliche Rechenzeit des Trackers. Die HLT GPU Tracker Komponente beinhaltet den Overhead, der von der HLT Framework Software zum Übergeben der Datenpakete erzeugt wird. Letzteres ist für die CPU Implementierungen irrelevant, hat aber auf die Performance des GPU Trackers einen großen Einfluss und wurde deshalb im Umfang dieser Arbeit optimiert und benötigt nur noch ein Zehntel der früheren Zeit. Insgesamt gesehen, ist der CPU Tracker des HLT bereits etwa fünfzehn Mal so schnell wie die Offline Variante, der GPU Tracker übertrifft die CPU Version um einen weiteren Faktor zehn.

Abgesehen von der Rechenzeit bewirkt der GPU Tracker auch eine deutliche Senkung der Kosten des HLT. Um die Kostenersparnis abzuschätzen, muss man bedenken, dass Tracking auf der GPU etwa so schnell ist wie auf der CPU bei gemeinsamer Nutzung aller Rechenkerne eines HLT Servers. Die GPU nutzt dabei nur vier der 24 Kerne und lässt einen Großteil der Ressourcen für andere Aufgaben frei. So gesehen kann man durch Installieren einer GPU in einen ohnehin vorhandenen Server quasi einen vollständigen Server einsparen. Unter Berücksichtigung der Preise von 300 $ pro GPU und 5000 $ pro Server ergibt sich für den HLT durch den Einsatz der 64 GPUs eine Kostenersparnis von 300000 $ – zusätzlich benötigte Infrastruktur und Strom für mehr Server nicht mitgerechnet.

Abgesehen vom Tracker wurde auch der Track Merger für GPUs umgesetzt. Hier konnte jedoch gezeigt werden, dass sich aufgrund der limitierten PCI Express Bandbreite keine deutliche Verkürzung der Bearbeitungszeit erzielen lässt. Daher ist es sinnvoller, den Tracker, der davon deutlich mehr profitiert, auf der GPU auszuführen, und den Merger auf dem Prozessor zu belassen.

Heterogene HPC Benchmarks

Der Linpack Benchmark (oft auch **H**igh **P**erformance **L**inpack (HPL) genannt) ist das Standardwerkzeug zur Klassifikation der Rechenleistung von Hochleistungsrechnern. HPL löst ein dicht besetztes lineares Gleichungssystem iterativ mittels LU-Faktorisierung mit zeilenweiser Pivotisierung. Jede Iteration besteht aus der Panel-Faktorisierung, dem Panel-Broadcast, der Replikation der Pivotisierung (**LASWP** genannt), dem U-Matrix Update (**DTRSM**) und dem Trailing-Matrix Update. Der letzte Schritt ist eine Matrix-Matrix-Multiplikation, die von einer Routine mit Namen **DGEMM** durchgeführt wird und die das Gros der Rechenzeit beansprucht.

Der LOEWE-CSC ist ein Supercomputer der Goethe Universität bestehend aus circa 800 Servern mit Magny-Cours Prozessoren und AMD Radeon 5870 GPUs, der im Herbst 2010 in Frankfurt installiert wurde. Sanam ist ein Großrechner der King Abdulaziz City for Science and Technology, der in Kooperation mit dem Frankfurt Institute for Advanced Studies geplant und installiert wurde. Die Evaluation der Hardware für Sanam wurde vollständig im Umfang dieser Arbeit vorgenommen. Da keine für AMD GPUs optimierten Versionen von DGEMM und HPL existierten, wurde eine DGEMM Bibliothek für GPUs mit Namen **CALDGEMM** und darauf aufbauend eine Linpack Implementierung (**HPL-GPU**) geschrieben, die beide als Open Source Software verfügbar sind.

CALDGEMM nutzt einen in AMD GPU Assembler (CAL) geschriebenen DGEMM Kernel, der auf den getesteten AMD GPUs circa 90 % der theoretischen Peakperformance erreicht. Um parallel zur GPU den Prozessor zu nutzen, setzt CALDGEMM schnelle DGEMM Routinen hochoptimierter BLAS Bibliotheken wie GotoBLAS, MKL oder ACML ein. Ein elaboriertes Framework bietet viele Features, welche die hohe Kernel Performance auch real im System zur Verfügung zu stellen. Dazu gehören vektorisierte Vor- und Nachbearbeitungsschritte auf dem Prozessor mit manuellem Prefetching, welche die Daten geeignet umformatieren und gewisse einfache Rechnungen übernehmen, um PCI Express Bandbreite zu sparen; eine Pipeline, die die Matrix aufteilt und dann die CPU Berechnungen, Datentransfers, und DGEMM Kernels auf der GPU zeitlich überlagert; ein dynamisches Caching von Daten auf der GPU, welches sicherstellt, dass alle Da-

ten nur einmal übertragen werden müssen; und ein dynamischer mehrphasiger Scheduler, der die Rechnungen auf mehrere CPUs und GPUs verteilt. Patches für GotoBLAS und für die OpenMP Bibliothek von GCC verhindern, dass mehrere Threads gleichzeitig denselben Kern belegen, ein binärer Patch des AMD Grafikkartentreibers ändert das Verhalten der Pufferverwaltung und verringert signifikant die Häufigkeit von Page Faults. Die LOEWE-Knoten erreichen damit beispielsweise 98,5 % der Kernelperformance in echten Anwendungen. Weiter erzielen sie im kombinierten CPU / GPU DGEMM 623,5 von 745,6 $^{\text{GFlop}}/_{\text{s}}$ theoretischer Peakperformance. Der zwei Jahre neuere mit vier GPU Chips ausgestattete Sanam erreicht 2923 von 3661 $^{\text{GFlop}}/_{\text{s}}$ auf einem Server.

Für HPL-GPU wurde ein neuartiger, auf mit GPU-Beschleunigern ausgestattete Computer zugeschnittener **Lookahead** entwickelt, der die seriellen Teile des HPL hinter der GPU Berechnung versteckt. Hierzu beginnt er bereits während des DGEMM der aktuellen Iteration mit der Faktorisierung des Panels und dem Broadcast der nächsten Iteration. Zusätzlich sind DGEMM, LASWP und DTRSM in einer Pipeline angeordnet. Der Scheduler sorgt dafür, dass die jeweils zuerst benötigten Ergebnisse der Matrix Multiplikation auch zuerst bereitstehen. Mittlerweile hat Intel die in CALDGEMM eingeführten Algorithmen in deren HPL Implementierung für Beschleuniger weitestgehend übernommen.

Moderne Cluster bestehen oft nicht mehr nur aus gleichartigen Knoten. Der LOEWE-CSC beispielsweise hat einige Knoten mit mehr CPU Kernen und mehr Speicher aber ohne GPU für spezielle CPU- und speicherintensive Anwendungen. Herkömmliche Linpackimplementierungen verteilen die Rechenlast völlig gleichmäßig auf alle vorhandenen Knoten und können mit solch einer heterogenen Situation nicht umgehen, denn sie bremsen damit alle Knoten auf das Niveau des langsamsten teilnehmenden Knotens herunter. HPL-GPU hat einen neuen Algorithmus, um die Matrix der vorhandenen Rechenleistung entsprechend zu verteilen. Hierfür musste insbesondere der Algorithmus zum Lösen von Dreieckssystemen modifiziert werden. Ein exemplarischer Test mit sechs Knoten drei verschiedener Leistungsklassen erreichte 96.9 % der akkumulierten Einzelleistungen aller Knoten und damit lediglich 3.1 % Granularitätsverlust.

Für multi-GPU Systeme ändern sich einige Paradigmen. Während es bei einer GPU möglich ist, den Kernel bis ins letzte Detail zu optimieren und diese Kernelperformance durch ein geschicktes Framework im System verfügbar zu machen, stellen Beschränkungen in der PCI Express und der Speicherbandbreite bei mehreren GPUs das vorherrschende Problem dar. Beispielsweise nutzt CALDGEMM bei nur einer GPU in Mehrsockelsystemen für GPU bezogene Aufgaben nur Prozessorkerne auf dem CPU-Sockel, der direkt die GPU anbindet. Bei mehreren GPUs ist es viel wichtiger, möglichst viel Aggregatspeicherbandbreite zu akkumulieren, und hierzu müssen Kerne auf allen Sockeln genutzt werden. Einige interne Parameter können anders gewählt werden, so dass die DGEMM Rechenleistung des Kernels zwar minimal abnimmt, dafür aber die benötigte Speicherbandbreite so stark reduziert wird, dass sich netto eine Beschleunigung ergibt. Während des Linpack Benchmarklaufes ändern sich mit jeder Iteration die Anteile der verschiedenen Schritte an der Gesamtrechenlast. Dem kann Rechnung getragen werden, indem gewisse Parameter laufend angepasst werden, so dass zu jedem Zeitpunkt die maximale Leistung ausgeschöpft wird.

Da GPUs nur sehr einfache, aber dafür viele Kerne einsetzen, sind sie für optimierte Anwendungen konstruktionsbedingt energieeffizienter als Prozessoren. Beim HPL ist es möglich, die Energieeffizienz weiter zu steigern, indem man weitere Teile des Algorithmus vom CPU auf die GPU auslagert und die Prozessoren damit soweit als möglich absichtlich brach liegen lässt. HPL-GPU bietet einen effizienzoptimierten Modus, der auf dem Sanam zwar eine um 11,1 % reduzierte Leistung, dafür aber eine um 23,2 % reduzierte Stromaufnahme ermöglicht. Auch für spezielle Systeme im Niedrigenergiebereich gibt es besondere Anpassungen, um deren verhältnismäßig langsame Prozessoren zu kompensieren. Mit optimierten Versionen von HPL-GPU konnte bereits Anfang 2011 sowohl auf einem solchen Niedrigenergiesystem von SDS als auch auf einem multi-GPU System jeweils eine Energieeffizienz von mehr als 1 $^{\text{GFlop}}/_{\text{J}}$ demonstriert werden.

Schließlich erreicht der Linpack 563 von 745 $^{\text{GFlop}}/_{\text{s}}$ Peakperformance (75,5 %) auf einem LOEWE-CSC Knoten und 2679 von 3661 $^{\text{GFlop}}/_{\text{s}}$ (73.2 %) auf einem Sanam Knoten. Läufe mit mehreren

Knoten liegen pro Knoten bei 90 % oder mehr der Einzelknotenperformance. Der LOEWE-CSC platzierte sich unter Verwendung von HPL-GPU im November 2010 mit 285 $^{\text{TFlop}}/_{\text{s}}$ auf Platz 22 in der Top500 Liste der schnellsten Supercomputer, Sanam erzielte im November 2013 mit 2,35 $^{\text{GFlop}}/_{\text{J}}$ den zweiten Platz in der Green500 Liste der energieeffizientesten Supercomputer. (Die Top500 Leistung beläuft sich auf 532 $^{\text{GFlop}}/_{\text{s}}$.) HPL-GPU bot damit auf dem LOEWE-CSC eine bis dato auf GPU-Clustern unerreichte Effizienz im Vergleich zur Peakperformance und erst kürzlich konnten andere Implementierungen von NVIDIA und Intel aufschließen, wobei NVIDIA z. B. auf dem Titan Supercomputer nur auf den Grafikkarten rechnet und die Prozessoren außen vorlässt, und Intel den Lookahead von HPL-GPU übernommen hat.

Um mit weiteren Architekturen kompatibel zu sein, kann CALDGEMM alternativ CUDA und OpenCL als Backend nutzen. Ein alternatives DMA Schema halbiert die erforderte Bandbreite zum Hauptspeicher, stellt dafür aber höhere Ansprüche an die DMA Engines. Ein Test mit NVIDIA Grafikkarten demonstrierte eine Skalierung dieses Schemas bis zu etwa 8 $^{\text{TFlop}}/_{\text{s}}$ DGEMM Performance auf einem Server.

Fehlertolerante Kodierung

Die Menge der von der Menschheit produzierten digitalen Daten steigt rasant an und hat inzwischen viele Exabyte pro Jahr erreicht. Dies stellt eine enorme Herausforderung bei der Datenspeicherung dar. Unzählige Speichermedien müssen parallel eingesetzt werden, wodurch das regelmäßige Versagen einzelner Medien vorprogrammiert ist. Die Kodierungstheorie ermöglicht die fehlertolerante Datenspeicherung und ist eine absolute Notwendigkeit für jedwede heutige Computerinfrastruktur. Darüber hinaus gibt es Verbindungen zur fehlertoleranten Datenübertragung, zur Fehlererkennung und zu ausfallsicheren redundanten Computersystemen, bei denen sichergestellt ist, dass der Ausfall einzelner Komponenten das Gesamtsystem nicht beeinträchtigt. Diese Arbeit handelt nur von fehlertoleranter Kodierung, die Prinzipien lassen sich aber leicht auf andere Gebiete übertragen.

Die Mathematik hinter vielen fehlertoleranten Codes findet in endlichen Körpern $\mathbb{F}_q$ mit einer Primzahlpotenz $q = p^l$ statt, wobei die Rechenoperationen nicht nativ von Computern unterstützt werden, sondern emuliert werden müssen, was aufwändig ist und viel Rechenzeit verschlingt. Im Folgenden bezeichnet ein (n, k)-Code einen Code, der n Datenwörter in $n + k$ Codewörter kodiert, so dass beim Verlust von bis zu k Codewörtern alle Datenwörter wiederhergestellt werden können. Prominente Beispiele solcher Codes sind **Reed-Solomon Codes**, deren Funktionsweise sich auf Eigenschaften der Vandermonde-Matrix stützen, oder darauf aufbauende **Cauchy-Reed-Solomon Codes**. Die eigentliche Kodierung bzw. die Wiederherstellung von Daten aus den Codewörtern wird über eine Matrix-Vektor-Multiplikation realisiert. Bei den obigen zwei Beispielen liegen die Werte in $\mathbb{F}_{2^l}$ bzw. in $\mathbb{F}_2$, meistens mit $l = 8$ oder $l = 16$, wobei immer gelten muss $2^l \geqslant n + k$.

Diese Arbeit behandelt die Kodierung in zwei Schritten. In einem ersten theoretischen Teil werden Codes diskutiert, deren Operationen sich besser auf die vorhandenen Rechenoperationen von Computern abbilden. Es wird eine Methode vorgestellt, mit Hilfe der Ganzheitsringe algebraischer Zahlkörper Codes mit Integer-Rechenoperationen in $\mathbb{Z}/2^b\mathbb{Z}$ zu erzeugen, die jenen aus [And$^+$ 05] entsprechen, die aber mit der vorgestellten Methode um den Faktor $\frac{3b}{2} \ln_2 p$ schneller erzeugt werden können. Cauchy-Reed-Solomon Codes können durch geeignete Vektorisierung so formuliert werden, dass die Rechenoperationen in $\mathbb{F}_2^b$ durchgeführt werden können. Beides können Computer effizient bewerkstelligen. Schließlich kann man die wiederholte Matrix-Vektor-Multiplikation, die per Konstruktion speicherbandbreitenlimitiert ist, in eine Matrix-Matrix-Multiplikation umschreiben und damit die volle Rechenleistung von Prozessoren ausschöpfen.

Im zweiten Teil werden neue schnelle Implementationen der besprochenen Codes vorgestellt, die in Form der Open Source Bibliothek QEnc frei zur Verfügung gestellt werden. Hierfür werden

zuerst vorhandene leistungsfähige DGEMM Bibliotheken zur Flieskomma-Matrixmultiplikation abgewandelt, um eine Implementierung für den Integerfall (IGEMM) in $\mathbb{Z}/2^b\mathbb{Z}$ sowie den Binärfall (BGEMM) in $\mathbb{F}_2^b$ zu erhalten. Für relativ große Matrizen ($n, k \geqslant 48$) erreichen diese Bibliotheken die spezifizierte Spitzenleistung der Prozessoren. Die Cauchy-Reed-Solomon Codes erlauben noch eine weitere Optimierung, die jedoch für Computer gewöhnlich nicht im Allgemeinen realisiert werden kann, sondern nur für fest vorgegebene Werte von n und k. Hierbei wird ausgenutzt, dass man Multiplikationen in $\mathbb{F}_2$ gar nicht durchführen muss (man kann ja nur mit 0 oder mit 1 multiplizieren, wobei man jeweils das Ergebnis schon kennt), und dass man darüber hinaus die Hälfte der Additionen auslassen kann, da der konstante Wert 0 addiert wird. Darüber hinaus wurde eine analoge Optimierung für die auf Integeroperationen in $\mathbb{Z}/2^b\mathbb{Z}$ basierende Kodierung entwickelt. Beide Versionen setzen aber eine a priori Kenntnis der jeweiligen Matrix voraus.

QEnc löst dieses Problem, indem zur Laufzeit binärer Assemblercode generiert und ausgeführt wird, der auf die spezifische Matrix hin optimiert ist (**Automorphe Kodierung**). Hierbei wird eine ganze Reihe an Optimierungen vorgenommen. Einige ähneln Compileroptimierungen für C Code – Jedoch ist es nicht zielführend, C Code zu generieren und zu kompilieren, da die Codes sehr groß werden und die Auswertungsbäume so komplex werden, dass die Compiler ihre Optimierungen nur schwer anwenden können. QEnc generiert SSE (XOR128) oder AVX (XOR256) Vektorinstruktionen mit folgenden Optimierungen:

- Einsatz von Prefetching Instruktionen und Streaming Stores.
- Techniken, die auch von Compilern eingesetzt werden, wie unter anderem Register Renaming zum Einsparen von Registern, Load / Store Interleaving zur besseren Auslastung der CPU Ausführungseinheiten, Einsatz von Ternary Instruktionen, Padding mit *NOP* Instruktionen für besseres Alignment, et cetera.
- Mathematische Optimierungen, wobei Teilergebnisse an anderer Stelle wiederbenutzt werden, um Instruktionen zu sparen.
- Einsatz der Blocking-Technik zur Reduktion der nötigen Speicherbandbreite.
- Nutzung des Strassen-Algorithmus zur schnellen Matrixmultiplikation.

Die Blocking-Technik iteriert nicht alle Schleifen nacheinander vollkommen, sondern wechselt hin und her. Dies ist möglich, so lange die Ergebnisse unabhängig sind, und hilft die aktuellen Daten immer in den CPU Caches zu halten. QEnc nutzt eine Vielzahl an Blocking-Stufen: ein Register-Blocking, ein zweistufiges Daten-Blocking sowie ein zweistufiges Blocking auf Instruktionsebene. Instruktionsblocking ist eine neue Technik die von QEnc eingeführt wurde. Da die Matrixeinträge in den Instruktionen enthalten sind, läuft ab einer bestimmten Matrixgröße der Instruktionscache voll und die Leistung leidet beträchtlich, wie Abb. 5 illustriert.

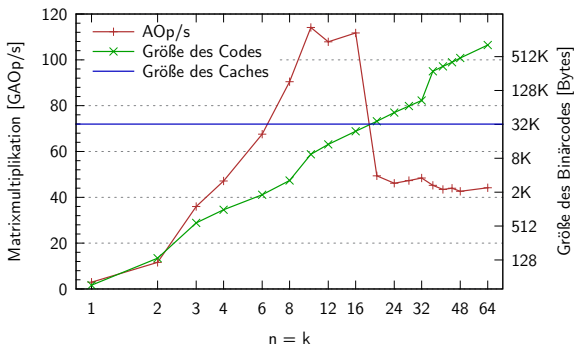


Abbildung 5: Leistungseinbruch von QEnc ohne Instruktionsblocking

Abb. 6 zeigt die beim Kodieren auf einem Sandy Bridge System erreichte Bandbreite sowie den Instruktionsdurchsatz an 32-bit Operationen. Die Speicherbandbreite entspricht genau dem doppelten der angegebenen Bandbreite, da die Eingangsdaten gelesen und gleich große Ausgangsdaten geschrieben werden. Die eingezeichnete Maximalbandbreite wurde durch synthetische low-level Assemblerbenchmarks gemessen und entspricht der höchsten erzielbaren Bandbreite auf dem System. Der maximale Instruktionsdurchsatz wurde aufgrund der Taktfrequenz berechnet unter

der Annahme, dass der Prozessor pro Takt drei Operationen durchführen kann. Dies ist allerdings nicht ganz korrekt und wird auch teilweise überschritten. Unter gewissen Umständen kann parallel zu den drei SSE Instruktionen noch eine vierte skalare Instruktion ausgeführt werden, und durch μ-op Fusion und Macro-op Fusion kann der Prozessor mehrere Instruktionen in einer Ausführungseinheit ausführen. Die Abbildung zeigt ganz klar, dass QEnc bei kleinen Matrizen bis $n, k \leqslant 8$ speicherbandbreitenlimitiert ist, darüber liegt eine Limitierung durch den maximalen Instruktionsdurchsatz vor. Bei sehr großen Matrizen verringert sich der Instruktionsdurchsatz wieder geringfügig sobald die Optimierungen des Strassen-Algorithmus greifen. Zwar können Teile des Strassen-Algorithmus prinzipbedingt keinen Maximaldurchsatz erreichen können, trotzdem ist er aber insgesamt schneller. Damit ist sehr deutlich, dass QEnc über alle Matrixgrößen hinweg an harte Grenzen der eingesetzten Hardware stößt. Messungen mit $k < n$ zeigen das gleiche Verhalten. Darüber hinaus kann QEnc auch differentielle Codes generieren. Hierbei wird in jedem Fall die maximale Speicherbandbreite erzielt.

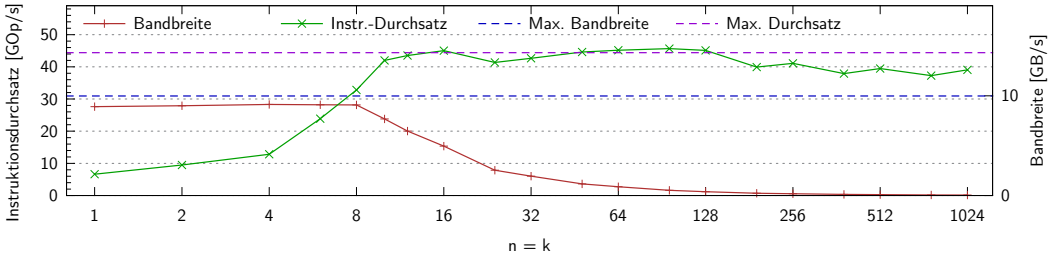


Abbildung 6: Erreichte Speicherbandbreite und Instruktionsdurchsatz der Automorphen Codes

Die bisherige Diskussion beschränkte sich auf einen einzelnen CPU Kern. QEnc kann über OpenMP mehrere Kerne nutzen und kann sowohl die auf einfacher Matrixmultiplikation beruhenden Codes als auch automorphe Codes auf Grafikkarten ausführen. Zu letzterem Zweck generiert QEnc OpenCL Code. Als dritte Möglichkeit kann QEnc VHDL Code generieren, um auf FPGAs zum Einsatz zu kommen. Abb. 7 gibt einen Überblick über die erreichte Rechenleistung, die, um eine gewisse Vergleichbarkeit zu bieten, als die Leistung angegeben ist, die eine naive Matrixmultiplikation benötigt, um die gleiche Gesamtleistung zu erzielen.

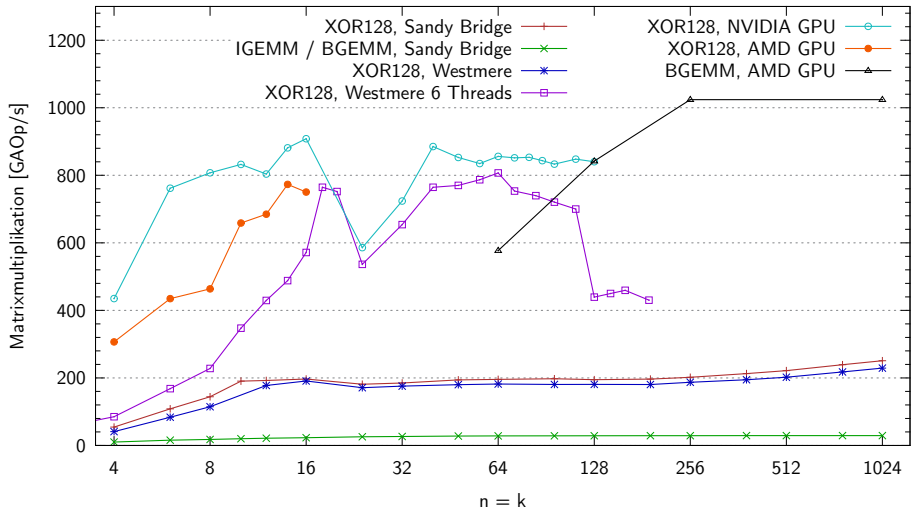


Abbildung 7: Performance der Matrixmultiplikation in QEnc

Man sieht, dass auf Prozessoren die einfache Matrixmultiplikation weit abgeschlagen ist. Der automorphe Code leistet beeindruckende 251 $^{\mathrm{GAOp}}/_{\mathrm{s}}$ mit nur einem Prozessorkern, eine Steigerung um das 9,1-Fache im Vergleich zur naiven Matrixmultiplikation mit SGEMM, IGEMM oder

BGEMM, bei denen selbst die schnellsten Bibliotheken jeweils nur etwa $27,5\,\mathrm{GAOp/s}$ bieten. Der Westmere Prozessor zeigt eine Beschleunigung von etwas mehr als dem Faktor vier unter Einsatz von sechs Threads. Die hohe Last auf den gemeinsamen Caches fordert hier einen gewissen Tribut. Auf Grafikkarten funktioniert die naive Matrixmultiplikation bei großen Matrizen besser, da der OpenCL Code nicht so optimal übersetzt wird wie der Assemblercode auf dem Prozessor. Abb. 8 gibt einen Überblick über die erreichte Bandbreite. Die GPUs demonstrieren immense Bandbreiten und der FPGA spielt in einer völlig anderen Liga. Allerdings sind aufgrund des beschränkten PCI Express Durchsatzes diese Bandbreiten in der Realität nicht verfügbar.

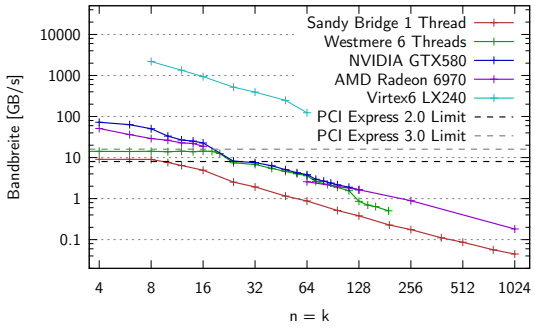


Abbildung 8: Übersicht der Bandbreite beim Kodieren auf CPU, GPU und FPGA

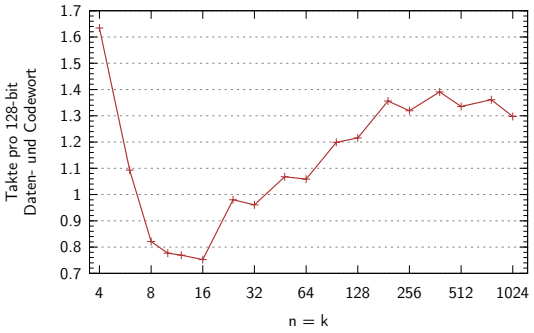


Abbildung 9: Anzahl der benötigten Taktzyklen pro 128-bit Daten- und Codewort

Die QEnc Implementierung benötigt weniger als drei arithmetische Operationen pro 128-bit Datenwort, um ein 128-bit Codewort zu erzeugen. Dies sind weit weniger als andere Implementierungen benötigen, und es ist anschaulich plausibel, dass man diese Zahl nicht viel weiter reduzieren kann. Pro durchgeführter arithmetischer Operation führt QEnc $1,23$ CPU Instruktionen aus. Berücksichtigt man die Operationen zum Laden und Speichern von Daten, für Prefetches und zum Kontrollfluss, so ist dies ein sehr geringer Overhead. Mit jedem Takt führt QEnc $2,61$ 128-bit SSE XOR-Operationen aus, wenn man nur die arithmetischen Operationen der Matrixmultiplikation zählt. Abb. 9 visualisiert wie viele Takte pro Code- und Datenwort nötig sind, und unterstreicht damit, wie wenig Spielraum es noch für Optimierungen gibt. Auf Prozessoren kodiert QEnc mit über $10\,\mathrm{GB/s}$ bei Matrixgrößen bis etwa $n = k = 20$, was mehrfach schneller ist als andere Implementierungen selbst bei kleinen Matrizen schaffen.

Vergleich der Ergebnisse

Um die Ergebnisse zu vergleichen wird die Metrik

$$\gamma = \frac{a_g/p_g}{a_c/p_c} = \frac{a_g/a_c}{p_g/p_c}$$

definiert, wobei a_g und a_c die auf GPU und CPU erreichte Leistung sind, p_g und p_c die theoretischen Spitzenwerte. Die Metrik γ vergleicht nicht den Leistungsunterschied zwischen GPU und CPU – dies ist nicht immer zweckdienlich, da die GPU konstruktionsbedingt der schnellere Prozessor ist – sondern setzt den Leistungszuwachs in Bezug zur Spitzenperformance. Die Anwendung dieser Metrik auf die hier vorgestellten Programme ergibt in fast allen Fällen einen Wert zwischen $0,7$ und $1,0$, d. h. dass die GPU Implementierungen ähnlich effizient arbeiten wie ihre CPU Analoga. Bei den Beispielen mit $\gamma < 0.7$ konnte immer ein eindeutiger Flaschenhals identifiziert werden, wie z. B. die PCI Express Bandbreite beim Track Merger. Es lässt sich folgern, dass bei guter Programmierung eine effiziente Implementierung der meisten Algorithmen auf GPUs möglich ist. Dies erfordert in der Regel mindestens eine Pipeline und asynchronen Datentransfer, entlohnt dafür aber mit Steigerungen der Leistung und der Energieeffizienz sowie mit einer günstigeren Total Cost of Ownership.

David Rohr

Neckargrün 1 – 68259 Mannheim
✉ *drohr@jwdt.org*

Curriculum Vitae

Personal Information

Date of Birth	September 18, 1983
Place of Birth	Mannheim
Citizenship	German

Employment

2010 – Today **Scientific Assistant**, *Johann Wolfgang Goethe University*, Frankfurt, Germany.

- 11/2012 – Commissioning & Installation of the Sanam Compute Cluster – Second Rank in November 2012 Green500 List.
- 12/2010 – Deployment of the ALICE HLT TPC GPU Tracker at the ALICE HLT.
- 11/2010 – Commissioning of the LOEWE-CSC Cluster – Rank 22 in the November 2010 Top500 List.
- Supervision of Master Theses.
- Teaching Assistant for:
  - SS2010 – Lecture in High Performance Computing.
  - WS2010/2011 – Practical Course in High Performance Computing.
- Member of ALICE Computing Workgroups CWG5, CWG7, and CWG12.

2007 – 2010 **Tutor in Mathematics**, *Ruprecht Karls University*, Heidelberg, Germany.
Linear Algebra 1/2, Algebra 1/2, Complex Analysis, Höhere Mathematik für Physiker.

Topic of Dissertation Thesis

Title **On Development, Feasibility, and Limits of Highly Efficient CPU and GPU Programs in Several Fields**.
Fast Parallel SIMDized GPU-accelerated Reed-Solomon Encoding, Heterogeneous Linpack Benchmark, and Event Reconstruction for the ALICE Experiment.

Supervisors Prof. Dr. V. Lindenstruth & Prof. Dr. U. Kebschull.

University Studies

09/2010 – Today **PhD Student**, *Johann Wolfgang Goethe University*, Frankfurt.

09/2010 – Today **Graduate School**, *Helmholtz Graduate School for Hadron and Ion Research*.

03/2010 **Physics Diploma**, *Ruprecht Karls University*, Heidelberg, Grade: Very Good (1.1).
Minor Subject: Computer Science.
Thesis Title: ALICE TPC Online Tracking on GPU based on Kalman Filter.
Supervisor: Prof. Dr. V. Lindenstruth.

08/2008 – 03/2010 **Scholarship by the Studienstiftung des Deutschen Volkes**.

02/2008 **Mathematics Vordiplom**, *Ruprecht Karls University*, Grade: Very Good (1.0).

10/2006 **Physics Vordiplom**, *Ruprecht Karls Universität*, Grade: Very Good (1.5).

10/2004 – 03/2010 **Studies in Physics and Mathematics**, *Ruprecht Karls University*, Heidelberg.

Scientific and Other Work Experience

07/2008 – 09/2008 **Participation at the IBM Extreme Blue Project**.

- Topic: Hardware Accelerators for Data Management Applications.
- Inventions / Patents:
  - US 8,380,737 B2: Computing Intersections of Sets of Numbers.
  - US 8,495,286 B2: Write Buffer for Improved DRAM Write Access Patterns.

09/2003 **Internship at the Max Planck Institute for Astrophysics**, *Garching, Munich*.
Topic: Age Determinations of Metal-Poor Field Stars.

10/2000 **Work Experience at Loci Computer Systems**, *Swansee*, Wales.

School and Civilian Service

10/2003 – 08/2004 **Civilian Service**, *Institute for Radiology*, Ludwig Maximilian University, Munich.
Development of Web-based Intranet System for Radiological Diagnostic Findings.

1994 – 2003 **School**, *Integrierte Gesamtschule Mannheim Herzogenried*.
Abitur with Overall Grade: Very Good (1.2).

1990 – 1994 **Elementary School**, *Gebrüder Grimm Elementary School*, Mannheim.

Languages

German	Native
English	Very Good
Italian, French, Latin	Basic

Computer Skills

Development	C, C++, PHP, HTML, JavaScript, SQL, Assembler, CUDA, OpenCL, CAL: *Very Good Skills*
	Visual Basic, Java, Bash, VHDL, Verilog, DirectX, OpenGL: *Moderate Skills*
	Fortran, Pascal, Perl, Python: *Basic Skills*
Applications	MS Office, OpenOffice, Adobe Creative Suite, Corel Draw Suite, Visual Studio: *Good Skills*
	ROOT: *Basic Skills*
Operating Systems	Windows, Linux: *Very Good Skills*

Publications

[1] M. Bach, J. De Cuveland, H. Ebermann, D. Eschweiler, M. Kretz, M. Pollok, D. Rohr, H. J. Lüdde, V. Lindenstruth: "The LOEWE-CSC: A Comprehensive Approach for a Power Efficient General Purpose Supercomputer", *in 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 1–17* [2013].

[2] D. Rohr: "ALICE TPC Online Tracker on GPUs for Heavy-Ion Events", *in 13th International Workshop on Cellular Nanoscale Networks and their Applications, pp. 298–303* [2012].

[3] D. Rohr, S. Gorbunov, A. Szostak, M. Kretz, T. Kollegger, T. Breitner, T. Alt: "ALICE HLT TPC Tracking of Pb-Pb Events on GPUs", *Journal of Physics: Conference Series*, *vol. 396, no. 1*: *p. 12044* [2012].

[4] D. Rohr, M. Bach, M. Kretz, V. Lindenstruth: "Multi-GPU DGEMM and HPL on Highly Energy Efficient Clusters", *IEEE Micro, Special Issue, CPU, GPU, and Hybrid Computing* [2011].

[5] M. Bach, M. Kretz, V. Lindenstruth, D. Rohr: "Optimized HPL for AMD GPU and Multi-Core CPU Usage", *Computer Science - Research and Development*, *vol. 26, no. 3-4* [2011].

[6] S. Gorbunov, D. Rohr, K. Aamodt, T. Alt, H. Appelsh, A. Arend, M. Bach, B. Becker, T. Breitner, et al.: "ALICE HLT High Speed Tracking on GPU", *IEEE Transactions on Nuclear Science*, *vol. 58, no. 4* [2011].

[7] D. Rohr: "ALICE TPC Online Tracking on GPU based on Kalman Filter", *Diploma thesis*, University of Heidelberg [2010].

[8] A. Arend, B. Becker, T. Breitner, S. Chattopadhyay, J. Cleymans, I. Das, O. Djuvsland, H. Erdal, R. Fearick, et al.: "ALICE HLT High Speed Tracking and Vertexing", *in 2010 17th IEEE-NPSS Real Time Conference*, *pp. 10–13* [2010].

[9] A. Weiss, M. Salaris, D. Rohr: "Age determinations of metal-poor field stars", *in Proceedings of the International Astronomical Union*, *vol. 1*, *p. 279* [2005].

Publications as Collaborator of ALICE

[1] ALICE Collaboration: "Measurement of inelastic, single- and double-diffraction cross sections in proton-proton collisions at the LHC with ALICE", *The European Physical Journal C*, *vol. 73, no. 6*: *pp. 1–20* [2013].

[2] ALICE Collaboration: "Charge correlations using the balance function in Pb-Pb collisions at $\sqrt{s_{NN}} = 2.76$ TeV", *Physics Letters B*, *vol. 723, no. 4-5*: *pp. 267 – 279* [2013].

[3] ALICE Collaboration: "Measurement of the inclusive differential jet cross section in pp collisions at $\sqrt{s} = 2.76$ TeV", *Physics Letters B*, *vol. 722, no. 4-5*: *pp. 262 – 272* [2013].

[4] ALICE Collaboration: "Net-charge fluctuations in Pb-Pb collisions at $\sqrt{s_{\rm NN}} = 2.76$ TeV", *Phys. Rev. Lett.*, *vol. 110*: *p. 152301* [Apr 2013].

....

[44] ALICE Collaboration: "Suppression of charged particle production at large transverse momentum in central Pb-Pb collisions at $\sqrt{s} = 2.76$ TeV", *Physics Letters B*, *vol. 696*, *no. 1-2*: *pp. 30 – 39* [2011].

[45] ALICE Collaboration: "Centrality dependence of the charged-particle multiplicity density at midrapidity in Pb-Pb collisions at $\sqrt{s_{\rm NN}} = 2.76$ TeV", *Phys. Rev. Lett.*, *vol. 106*: *p. 032301* [Jan 2011].

[46] ALICE Collaboration: "Charged-particle multiplicity density at midrapidity in central Pb-Pb collisions at $\sqrt{s_{NN}} = 2.76$ TeV", *Phys. Rev. Lett.*, *vol. 105*: *p. 252301* [Dec 2010].

[47] ALICE Collaboration: "Elliptic flow of charged particles in Pb-Pb collisions at $\sqrt{s_{NN}} = 2.76$ TeV", *Phys. Rev. Lett.*, *vol. 105*: *p. 252302* [Dec 2010].