KMA: A Dynamic Memory Manager for OpenCL

Roy Spliet
Delft University of Technology
The Netherlands

Lee Howes,
Benedict R. Gaster
AMD
USA

Ana Lucia Varbanescu
University of Amsterdam
The Netherlands

ABSTRACT

OpenCL is becoming a popular choice for the parallel programming of both multi-core CPUs and GPGPUs. One of the features missing in OpenCL, yet commonly required in irregular parallel applications, is dynamic memory allocation. In this paper, we propose KMA, a first dynamic memory allocator for OpenCL. KMA's design is based on a thorough analysis of a set of 11 algorithms, which shows that dynamic memory allocation is a necessary commodity, typically used for implementing complex data structures (arrays, lists, trees) that need constant restructuring at runtime. Taking into account both the survey findings and the status-quo of OpenCL, we design KMA as a two-layer memory manager that makes smart use of the patterns we identified in our application analysis: its basic functionality provides generic `malloc()` and `free()` APIs, while the higher layer provides support for building and efficiently managing dynamic data structures. Our experiments measure the performance and usability of KMA, using both microbenchmarks and a real-life case-study. Results show that when dynamic allocation is mandatory, KMA is a competitive allocator. We conclude that embedding dynamic memory allocation in OpenCL is feasible, but it is a complex, delicate task due to the massive parallelism of the platform and the portability issues between different OpenCL implementations.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management.*

Keywords

Dynamic memory allocation, Massive parallelism, Multi-/many-cores, OpenCL kernels.

General Terms

Algorithms,Performance,Languages

GPGPU-7 March 01 2014, Salt Lake City, UT, USA

1. INTRODUCTION

OpenCL [10] emerged in late 2008 (under the governance of the Khronos group) as a standard for programming heterogeneous, massively parallel architectures. Its popularity increases steadily in the high performance computing (HPC) community, and is being viewed as an interesting alternative to NVIDIA's CUDA for two major reasons: its vendor-independent specification and its promised portability. OpenCL is competitive against models like CUDA (for NVIDIA GPUs [8]) and OpenMP (for multi-core CPUs [17]), and it is currently supported by a large number of prominent vendors, including AMD, Apple, ARM, Intel, NVIDIA and Qualcomm. The model continues to evolve towards a user-friendly solution for implementing various types of applications.

The OpenCL programming model provides developers with an application model for a heterogeneous platform. This model includes several *devices* which run highly-parallel, compute intensive *kernels*, and a *host*, which typically manages the execution of the devices and kernels. A kernel is executed by multiple *work items*, organised in *work-groups*. All the work-items for each work-group are launched simultaneously by a program, and their execution is interleaved by a fine-grained scheduler on the device. Work-groups from a single kernel launch are unordered and are generally streamed through the device as capacity becomes available. Local, intra-group synchronisation is possible in OpenCL, while global, inter-group synchronisation is explicitly not a part of the model. Memory objects are managed by the host, and data must be copied to and from the device(s). This model applies particularly well for applications that require (large) regular data structures like 2D or 3D-grids, but could be insufficient to support irregular data structures like linked lists or trees.

Many traditional programming platforms offer dynamic memory allocation to support such irregular data structures. In contrast to static allocation, a dynamic allocator enables programs to determine and request memory for its data structures at run-time. Graph analysis and in-memory MapReduce problems are only two examples of large classes of problems that require a parallel application to work with such dynamic and irregular data structures. For performance reasons, the allocation subsystem must support concurrency and avoid memory wastage as much as possible.

The design of dynamic memory allocators has been extensively studied for both sequential- and parallel systems [3, 6, 11, 12]. CUDA also includes its own dynamic memory allocator [14]. Furthermore, there are already a couple of alternative CUDA memory allocator designs aimimg to improve the performance of the default one [19, 9]. Unfortu-

nately, none of these implementations can be ported directly to OpenCL, mainly due to portability challenges. Thus, despite memory allocators being well understood, OpenCL is still lacking one, a limitation that becomes obvious for developers who attempt to implement irregular data structures in their OpenCL kernels.

To overcome this limitation in OpenCL, we propose KMA, the first dynamic memory management system for *OpenCL kernels*. KMA is a generic solution that provides basic memory allocation/deallocation, but also allows for application-specific performance optimisations. Although we apply many of the lessons learned in prior work on low-level memory allocators and higher level optimisations, we encountered many new challenges related to the OpenCL platforms. Most noticeably, the different platforms implement different interpretations of OpenCL's relaxed memory ordering guarantees, limiting the portability of any proposed solution. In addition, thread safety cannot be enforced in OpenCL by the use of mutexes, devices in the OpenCL model do not manage their own memory, and OpenCL only allows for limited optimisation because synchronisation of work-items can only be guaranteed at the work-group boundaries.

For KMA, we present a design that tackles the problem at two levels: (1) a lower level memory allocator, providing the programmers with the full usability and flexibility of `malloc()` and `free()` at the price of a higher performance penalty, and (2) an optimised higher level data structure, providing more specialised memory management functions. The low level allocator is implemented using mostly lock-free algorithms to achieve a good scalability without requiring mutexes. Requests are served from pre-allocated memory. On the high-level, we implement the prefix-sum reduction approach proposed by Huang et. al [9] to combine memory allocations within work-groups and show how this increases an applications performance. We show performance figures on NVIDIA GPUs and x86 CPUs, explain why we had to sacrifice support for Intels' toolchain and AMD GPUs, and cover how the memory guarantees offered by OpenCL 2.0 could aid in improving portability.

The main contributions of our work are as follows:
(1) We propose KMA, the first dynamic memory manager for OpenCL.
(2) We empirically identify patterns for dynamic memory usage in parallel applications, and show how these patterns can be used to significantly improve the performance of KMA.
(3) We enumerate and discuss the requirements and challenges of building a system-wide memory allocator for OpenCL kernels, and discuss memory model changes that could improve KMA's portability.
(4) We discuss the advantages and limitations of KMA for real applications by means of a case-study, discussing both its usability and its performance.

2. BACKGROUND AND RELATED WORK

In this section we introduce the basic concepts needed to understand the context of this work, and survey the state-of-the-art in massively parallel memory allocators.

A dynamic memory allocator[1] is a generic name given to the system that manages a heap of memory and handles, in a centralised manner, the memory allocation and deallocation requests coming from applications. Generally, such a system (1) keeps track of the memory blocks on it's heap, including the allocation state of each block, (2) handles requests to allocate or free memory while constantly updating the state of the heap accordingly, and (3) communicates with the (main) operating system to alter the size of the heap when required. Typical measures of the performance of memory allocators are response time (the lower, the better) and memory utilisation (the higher, the better) [11]. The trade-off between these performance indicators is a metric we also consider relevant for our work.

[1]Along this paper, we often refer to the whole memory management system by the name of "memory allocator", a simplification also found in the literature.

2.1 Parallel memory allocators

There are two very popular heap management algorithms: Doug Lea's DLmalloc[11] as used in the GNU libc implementation, and the GPL-licensed Hoard algorithm[3]. Lea's allocator uses a single heap, and free blocks are placed into size-binned linked lists. Adjacent free blocks are coalesced forming larger blocks. Small and medium sized allocations use a best-fit algorithm, while large ones get served directly from the operating system. By using a large amount of size bins there is only a very limited amount of internal fragmentation. Hoard's allocator manages "superblocks" containing equally sized memory blocks. Allocating and grouping blocks of memory of the same size alleviates the external fragmentation (between any two allocated memory blocks), at the cost of more internal fragmentation, as the amount of required memory must always be rounded up to the size class border.

Such traditional memory allocators tend to scale poorly as cores are added to the system, due to the protective measures needed to keep the state of a centralised memory allocator valid, thus guaranteeing memory consistency. Locking the allocator when in use by a single thread would lead to severe penalties in highly-concurrent systems like GPUs. Hoard's allocator tackles this by using two types of heaps: a global heap and a per-process local heap. This approach limits the amount of conflicts that require locking, maximising the concurrency, but local heaps lead to severe overprovisioning and low overall memory utilisation. Alternatively, Dave Dice et al. proposed a mostly lock-free memory allocator for Solaris systems [6]. Michael Maged proposed LFMalloc [12], a portable lock-free memory allocator that makes use of the Compare-And-Store instruction of many modern processor architectures. LFMalloc scales nearly perfectly up to 16 processors, without any significant latency overhead.

None of these designs are immediately compatible with the OpenCL model. However, we have combined ideas found in Maged's and Lea's allocators towards a scalable design: we use a single, global heap, but use lock-free data structures to limit the impact of concurrency on performance.

2.2 Related Work

Our work builds upon ideas first implemented in "traditional" parallel memory allocators, as discussed in great detail in [18]. As the number of entities has a strong impact on the contention that the allocator will observe, this section focuses on related work in the sub-field of many-core memory allocators (i.e., allocators for platforms with hundreds to thousands entities).

Traditional parallel allocators [12, 6] fall short on SIMD or SIMT machines because concurrent allocations have to

be serialised Xiauhuang Huang et al. propose the xmalloc allocator [9] for CUDA GPUs to work around this issue: the memory requirements of all CUDA threads in a thread-group are gathered by means of a prefix-sum reduction[4] with a time complexity of $O(logn)$, and a single memory allocation is performed for all threads combined; the allocated memory is then distributed among the requesting threads. Their results show a great improvement in latency and excellent scaling with the number of cores in the GPU. We have generalised this solution as a family of optimisations for the high-level functions of KMA (see Section 4).

A different attempt to implement a kernel memory allocator for CUDA is presented by Steinberger et. al [19]. Based on Hoard's allocator, and using a local heap for each work-group, ScatterAlloc is shown to achieve good response time, scalability, and memory utilisation, outperforming both the built-in CUDA function *and* Xmalloc. ScatterAlloc uses a bit field to identify used chunks within a superblock, a technique that we adopt for our lock-free low-level allocator. As ScatterAlloc focuses only on NVIDIA's CUDA-compatible devices, it is unclear to us whether the proposed techniques can be preserved in a fully functional and portable implementation for OpenCL.

Another interesting attempt to address the performance of memory allocation in SIMD-like kernels is FDMalloc, presented by Widmer et al. [22]. The authors present a solution for reducing the number of calls actually made to the allocator through a voting mechanism. However, their solution requires a memory allocator to be available on the target platform (CUDA, in this case), bases its optimisations on existing patterns in the allocation requests, and does not allow threads to free their own allocations, providing a garbage collection instead. Compared to FDMalloc, KMA also provides a low level memory manager, and allows more flexibility in implementing and using the high level functions. Moreover, when OpenCL will support the required voting mechanisms [22], FDMalloc can be implemented as a high-level allocator in the KMA model.

3. REQUIREMENTS

To derive realistic requirements for dynamic memory allocation, as well as the patterns followed when using it in parallel applications, we evaluated a significant set of applications. This section outlines the findings of this survey and the derived requirements for KMA.

In our survey, we analysed representative applications from Asanovic's classification [2], looking for cases (if any!) when dynamic memory allocation is used to improve code [2]. Note that we did not consider C++ as a use-case in our study: while memory allocation is mandatory for any object oriented kernel language, the current proposal [16] does not include such features.

The list of analysed applications is prese nted in Table 1. Out of the twelve classes from [2], we found suitable parallel implementations for eight in the Rodinia [5] and Parboil [1] benchmarks; for dynamic programming, we chose a custom graph traversal [7] as a reasonable approximation.

Our analysis focused on (1) allocations and deallocations that occur more often than just at the beginning and end of a kernel execution, and (2) cases when pre-allocated buffers are insufficient due to some type of data-dependent behaviour. Thus, where source code was available in any parallel form, for CPUs or GPUs, we investigated the use of temporary data structures. When no source code was available, we focused on possible implementations of the algorithms. We even used the theoretical approach of Dynamic Programming to draw conclusions about the added value of having a heap allocator for such problems. For the classes of applications where no reasonably sized parallel sample application could be found (see 2, 10, 11 in Table 1), we leave a more detailed analysis for future work: existing applications already provide sufficient challenges for KMA has to address.

	Class	Application	Source	Implement	Use malloc
1	Finite State Machine	Level-7 filtering	Paper Hellas University[21]		List of states
2	Combinatorial	-			
3	Graph Traversal	Graph analysis	TU Delft code [15]	OpenCL	List of nodes
4	Structured Grid	Heart Wall	Rodinia code	OpenMP	none
5	Dense Linear Algebra	K-Means	Rodinia code	OpenMP	none
6	Sparse Matrix	SPMV	Parboil code	Cuda	none
7	Spectral (FFT)	FFT	Parboil code	Cuda	none
8	Dynamic Programming	Dijkstra	Theory		none
9	N-Body and Particle Methods	Barnes-Hut	Texas State U.[3] Code	OpenCL	Octree
10	MapReduce	-			
11	Backtracking	-			
12	Unstructured Grid	Back Propagation	Rodinia code	OpenMP	none

Table 1: Selected programs for use-case study and a brief overview of our findings.

To filter out the cases when memory allocation is not beneficial, we make the following observations. First, we should focus on temporary data structures because they are equivalent with kernel local data, and thus potential targets for kernel memory allocation. Second, variables that are only locally accessed should use local memory instead of dynamically allocated global memory; the large data structures with shared access are the challenging cases. Finally, variables whose `malloc()` and `free()` object counts are constant or directly connected to the problem size have known memory requirements, making host-based static allocation more efficient. Due to these observations, classes 4, 5, 6, 7, 8, and 11 from Table 1 are uninteresting for our KMA design [18].

The remaining classes of applications - Finite State Machine, Barnes-Hut, and MapReduce - have similar patterns in using memory allocation. In the case of non-deterministic finite state machines, a global list of possible future states is needed, dimensioned at run-time and heavily dependent on the input data. Similarly, in a graph traversal such as BFS, the nodes that are just being visited - the so-called BFS frontier [15] - are stored in a list of unknown size. At run-time, and depending on the starting node and on the graph structure, this list needs to be dynamically adjusted. Using a dynamic list can save a lot of communication with the host. Finally, in the Barnes-Hut algorithm, a dynamically sized octree of nodes is built after each iteration. Writing a Barnes-Hut implementation without overprovisioning and complicated index computations can be achieved if new octree nodes can be allocated from a global kernel heap. This could effectively eliminate a round-trip to the host system required to build this octree.

Our survey showed cases when kernel memory allocation is necessary: either for ease of programming and fast prototyping, or to address data-dependent behaviour. Furthermore, in many cases, users tend to use the basic allocate/deal-

[2]Code improvement is traditionally subjective and difficult to define. In this context, we mean code easier to read-/write/maintain.

[3]http://www.gpucomputing.net/?q=node/1314

locate functionality to build customised high-level dynamic data structures that can be altered (in shape and/or size) at runtime. Combined, these observations lead to two important requirements for KMA: (1) for flexibility, it has to provide a generic memory manager, with *generic* `malloc()` and `free()` functions, and (2) for performance and usability, it should provide *support* and *optimisations* for *high-level dynamic data-structures* and functions for their management.

4. DESIGN AND IMPLEMENTATION

In this section we discuss OpenCL's challenges and our solutions for KMA's design.

4.1 OpenCL challenges for memory allocation

We identify four challenges: (1) contention for synchronisation primitives in a high throughput, heavily threaded platform model, (2) a weak memory model with few ordering guarantees, (3) the mapping of OpenCL C's SPMD code lane-wise onto a SIMD execution unit, and (4) the lack of global synchronisation.

Contention for locks causes serialisation of parallel threads. Lock-free data structures are traditional solutions to avoid such contention, but they are extremely challenging to implement. In OpenCL, implementing lock-free data structures and even locks themselves is further complicated by the presence of a very weak memory model. Strictly under the OpenCL 1.2 specification, memory visibility is only guaranteed by atomic operations (which are themselves weakly ordered), by kernel boundaries and (to other work-items within a work-group only) by barrier synchronisation primitives. The effect of this weakness is that is it challenging to guarantee that data structures beyond simple atomically changed values are correctly updated in the parallel setting. Specifically, some of the choices made for the OpenCL implementations become vendor-specific, effectively resulting in different memory consistency models implemented by NVIDIA, Intel, and AMD SDKs[4].

Next, the specific mapping of SPMD code to SIMD units is a choice for the vendor implementation. For example, it is correct behaviour for a developer to write code where some work-items branch around allocation calls, and assume some allocations are masked out in the overall execution. However, to support portable, per-work-item memory allocation with this functionality, different solutions - some that are not portable, others that are simply inefficient - are mandatory for different types of device - e.g., GPUs and CPUs. Finally, OpenCL only defines global synchronisation and memory visibility at the boundaries of parallel kernels (or kernel-like entities, like copy operations). Thus, "officially", global synchronisation is not possible inside kernels. Global barriers based on spinning until a value changes are feasible for GPUs [23], but rely on assumptions about the underlying memory model and hence are not portable.

4.2 KMA: A two-layer memory allocator

We designed our allocator to address two main objectives: support generic, fully-flexible memory allocation *and* provide ready-to-use management of dynamic data structures (see Section 3). Thus, KMA has two layers: a low-level traditional memory allocator and a high-level, customized layer that uses the generic allocator with increased programmability and performance optimisations.

The low-level memory allocator

The goal of the low level memory allocator is twofold: to provide a simple abstraction layer on top of the memory capabilities offered by the OpenCL platform and to provide the programmer with a familiar interface. Thus, our basic solution provides the kernels with a large heap and the traditional `malloc()` and `free()` POSIX C API calls. The heap will be hosted in global memory, will be accessible to all threads, and it is intended for large, device-based data structures [5].

KMA implements a heap using a regular OpenCL read-write buffer, associated with a given device and a given command queue. A heap is instantiated on the host side (using `clSBMalloc_create()`[6]); its size is set by the user specifying the number of desired superblocks and the superblock size. We note here a first important OpenCL restriction: this heap cannot be resized at the runtime request of a kernel, because the device cannot asynchronously communicate this request to the host. Therefore, the heap is statically sized, which typically requires some degree of memory overprovisioning. Limiting memory wastage in this scenario is entirely application-specific.

Once instantiated, the heap is initialised on the device itself, by means of a special kernel that builds the necessary data structures and initial state. Note that, due to this device-based procedure, KMA adds no data transfers between the host and the device. After initialisation, a heap (a simple `cl_mem` object) can be passed as an argument to any subsequent kernel call that wants to use it. The programmer is then free to call `malloc(heap, size)` and `free(heap)` inside kernels that use a heap [7].

High-level custom data structures

One of the common uses of `malloc()` and `free()` is implementing and managing custom dynamic data structures. Some of these structures have specific usage patterns, where arbitrary interleaving of `malloc()` and `free()` does not occur. For instance, a list that can grow arbitrarily, but individual objects are never removed (for example, in a graph traversal algorithm) is using multiple allocations, but a single deallocation. The knowledge of such usage patterns allows us to provide KMA with two important features: (1) a skeleton for designing and implementing custom data structures, which improves programmability, and (2) pattern-driven optimisations (e.g., such as that presented in [9]), thus improving the performance of the basic, low-level allocator.

Implementation: Memory allocator

Blocks and superblocks - The structure of a heap object (see Figure 1) consists of a a state header followed by a number of "superblocks" given by the user at initialisation. Superblocks are further split up in allocatable space - i.e.,

[4]These are the SDKs we have used in this work.

[5]Data structures allocated on the device cannot be directly transferred to the host because the pointers belong to different memory spaces

[6]The complete description of the API is available in [18].

[7]In principle, multiple heaps can be used inside the same kernel, although this clearly defeats the purpose of using a global heap to solve overprovisioning

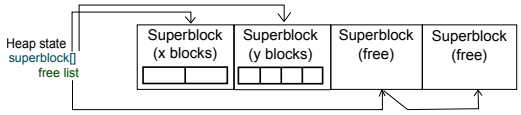

Figure 1: The structure of KMA.

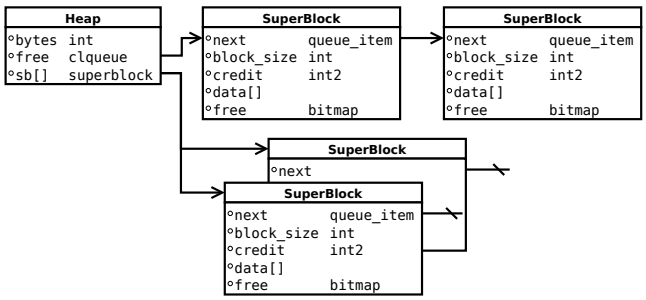

Figure 2: The design of KMA

blocks of different sizes, to be used by `malloc()`- and administration space, used to keep each block's state (e.g., if allocated or free) [6, 11, 9].

The administration of superblocks is described in Figure 2. Superblocks can be either active, inactive or free. Free superblocks are linked together in a "free"-list, but they cannot be coalesced. Thus, the size of an allocated block cannot be larger than the size of a superblock. Depending on the application, programmers are recommended to carefully tune this parameter, thus tuning memory utilisation (i.e., limiting memory wastage).

The possible sizes of blocks inside a superblock are approximated by $\{2^n \mid 2 \leq n < \log(\text{superblock\_size})\}$. Several heuristics are used to determine the actual block size [18], minimising the amount of space wasted on almost-fitting large blocks. A bitmap at the end of each superblock indicates the state (occupied or not) or each block. When a new superblock is required (e.g., because the current allocation request does not fit), one is taken from the free-list, prepared and connected to the superblock list in the heap state. Now the superblock is *active*. As soon as all its blocks are allocated, a superblock is disconnected from this list and becomes *inactive*. Freeing all the blocks inside a superblock causes it to be added to the free-list, to be reused later on.

Algorithms - Our goal is to use lock-free data structures and algorithms, to allow multiple memory operations in parallel (and avoiding the quirks of OpenCL synchronisation mechanisms). The algorithm used for allocating a block of memory has three steps:
(1) Find a superblock with the desired block-size, by checking whether there is an active superblock attached to the right entry in the superblock hashmap (`sb` in Figure 1). If found, no further action should be taken. Otherwise, the hashmap entry is marked to signal that one thread is preparing a superblock. A superblock is then taken from the free-list, its data structures are initialised, and the hashmap entry is updated with a pointer to this superblock.
(2) Reserve a slot inside this superblock, a simple atomic operation on the state of the superblock. This state contains the total number of blocks, and the number of free blocks inside the superblock. After decrementing the number of blocks and updating this value using Compare-and-Swap, a block is reserved.
(3) Find a free block in the superblock and mark it *taken*. This is accomplished by iterating over the blocks bitmap of the superblock. When the first free block is identified, it is updated using an atomic OR operation. If this operation succeeds, the block is allocated. If not, the thread continues its search until the allocation succeeds.

Freeing a block is done by an opposite process: first the state variable is incremented to indicate there is one free slot. The specific block is then freed by an atomic AND on the bitmap. If the superblock turns out to be completely free it is then returned to the free queue.

Free list - Generic linked lists are the most obvious choice to implement the *free*-list, but available generic algorithms, like the one proposed by H. Sundell et al.[20], do not meet the demands for our memory allocator. For example, lock-free linked lists can be corrupted when the cursor of an work-item iterating the list points to an element that is removed, and solutions proposed so far degrade performance by placing these cursors in global memory or do not suffice because they postpone deletion of nodes in the list.

By making superblocks all of equal size, there is no need for a data structure whose elements can be iterated over. Now the free-list can be a implemented as a queue or stack. The lock-free queue algorithm we use is described by M. Maged et al.[13]. Although this algorithm does not allow the queue to run empty, and therefore wastes one superblock of memory, it is one of the simplest and most practical lock-free queue algorithms found in literature.

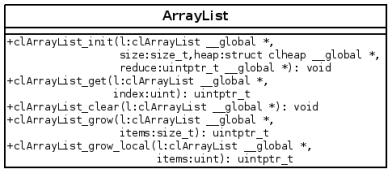

Figure 3: The design of ArrayList.

4.2.1 High-level data structure: ArrayList

ArrayList is a prototype of a top layer data structure and its management functions, inspired by Java's ArrayList.

ArrayList uses the low-level allocator and its APIs as a back-end, while the user interfaces exclusively with ArrayList functions (Figure 3). Once the ArrayList "object" is created, users can add elements to the ArrayList by growing it per work-item (`grow()`), per work-group (`grow_local()`), can get access to the nth element (`get()`), and can clear the list (`clear()`). The safety of the clearing operation has to be ensured by the way users programs the application: because we lack global synchronisation, there is no way we can guarantee that no OpenCL threads will be accessing the list, in which case list clearing results in memory corruption and/or null-pointer exceptions; the application might have this knowledge and use it to preserve thread-safety.

The Arraylist data is structured as a list of blocks, each annotated with the number of elements inside, and linked together by a queue. The `grow()` function is a simple allocation and insertion of a new element in the list. The `grow_local()` function provides optimised allocation: it is assumed this is called by the whole group of threads. These symmetrical calls are then replaced by a single call, made by a single thread knowing the total amount of required

memory.

In this prototype, we use XMallocs approach [9], but other schemes can be envisioned. These can be easily embedded in the provided skeleton of ArrayList. Moreover, additional optimisations can be performed on the gather/scatter of the requested memory allocations, to insure more favourable memory access patterns.

5. EXPERIMENTS AND RESULTS

In this section we present several performance figures for KMA. Our goal here is to quantify three aspects of the allocator: portability, performance impact, and scalability. To do so, we designed two types of experiments: microbenchmarking experiments, to measure the allocator in isolation, and a "live" experiment, designed to quantify the impact of KMA on a real application.

For the evaluation of KMA, a variety of OpenCL devices were used. The test systems and configurations that correctly executed all experiments are listed in Table 2.

5.1 Evaluating the low-level allocator

To understand the performance impact and scalability of the low-level memory allocator, we constructed two microbenchmarks: one that measures the latency of `malloc()` and `free()` for a single thread, and one that measures the scalability by reporting the latency of the routines when executed in a multi-threaded environment. We report kernel execution times, averaged over 10 runs.

For latency estimation, we measure the execution time of a microbenchmarking kernel (run by a single thread) that allocates and frees a 4-byte block of memory multiple times. The results show a linear correlation between the number of iterations and the total execution time, showing a constant latency of the operations. It is also visible that the results heavily depend on the platform family and generation.

For 10000 iterations on the NVIDIA GeForce GT640, the latency of a single `malloc()` and `free()` is approximately 9.8μs. Compared to a measured latency of 17.7μs when using CUDA's `malloc()` routine on the same GPU, KMA is almost twice as fast. In this single-threaded benchmark CPUs are much faster (than GPUs), averaging 0.6μs per allocation on the AMD FX-6300 and showing even lower latencies for the tested Intel CPUs.

To measure scalability, we use the same microbenchmark running on many threads. The results are shown in Figure 5 for the nVidia GT640 and the AMD FX-6300.In this case, each thread called `malloc()` and `free()` 100 times, with varying amounts of block sizes.

With 4608 threads and more than 45 times as many allocations as in the first experiment, the test program executes in 0.20 seconds on the NVIDIA GeForce GT640. This translates to serving 22.5x the number of requests per seconds.

Device	Type	Cores	Software
NVIDIA GeForce GT640	GPU	384	CUDA 5.0.35
NVIDIA GeForce GTX480	GPU	448	CUDA 5.0.35
NVIDIA GeForce GTX680	GPU	1536	CUDA 5.0.35
NVIDIA Tesla C2050	GPU	448	CUDA 5.0.35
NVIDIA Tesla K20m	GPU	2496	CUDA 5.0.35
AMD FX-6300	CPU	6	AMD APP 2.8
Intel Xeon E5-2620	CPU	6(12)	AMD APP 2.7
Intel Xeon E5620	CPU	8(16)	AMD APP 2.7
Intel Xeon X5650	CPU	6(12)	AMD APP 2.7

Table 2: Hardware platforms used for experiments.

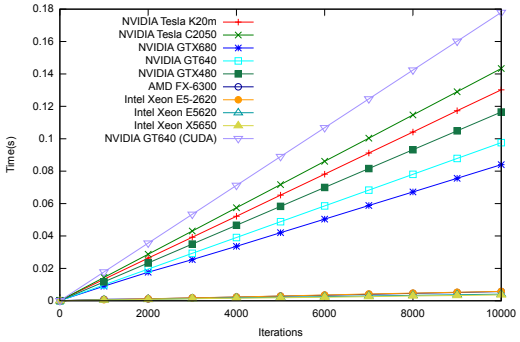

Figure 4: Execution time for one thread.

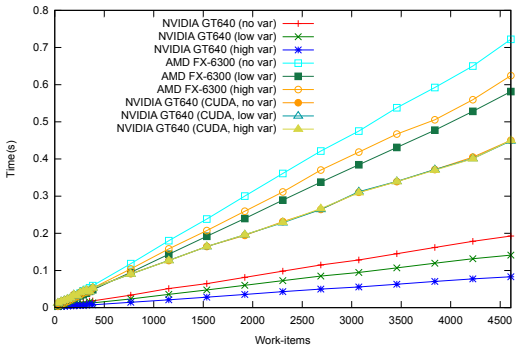

Figure 5: Execution time for multiple threads running 100 allocations per thread. Variance refers to allocation sizes: no var = no difference, low var = small differences, and high var = large differences.

The big difference in time per allocation between this experiment and the previous shows parallelism is exploited quite well with the lock-free algorithm. When there is more variance in the allocated block size, a greater level of parallelism is achieved, leading to even better performance. Again the trend is roughly linear, growing as the amount of work grows. Our implementation also shows a significant advantage over CUDA's heap allocator, being 2.4 to 5.4 faster.

Finally, we note that note that CPUs show a similar behaviour: fairly linear scalability. However, the execution time is much larger, due to the lower utilzation of SIMD hardware in the underlying OpenCL CPU toolchain, and the different design goals for the devices (single threaded throughput versus high memory bandwidth utilisation). For the high variance experiments, CPUs perform worse simply because there is more work that needs to be done, while available parallelism has already been fully exploited.

5.2 Evaluating the ArrayList prototype

To ideally evaluate the performance of ArrayList, an experiment similar to that in Section 5.1 should be conducted. Unfortunately, there is no portable way of ahieving global synchronisation without exiting the kernel (see Section 4.1), and thus no way to decide when all elements should be freed while maintaining a predictable state of the global data structure. Using global synchronisation by cutting up the kernel, the run-time will be dominated by the cost of such synchronisation, and will not show the performance of the routine itself.

Thus, we focused on the performance of allocation, and we set up a simple test case where each work item allocates either 4 or 8 bytes of memory 10 times in a row. In the `malloc()` case, every thread allocates its own memory. In the ArrayList case, a local prefix-sum reduction is used to reduce the number of calls to `malloc()` to once per work-group per iteration. The memory is not freed, so the two programs have the same end result. The results of this experiment are shown in Figure 6.

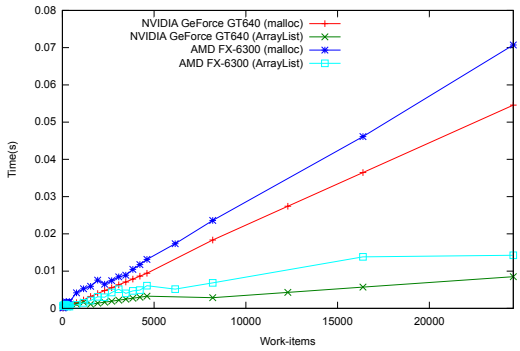

Figure 6: Comparing ArrayList to Malloc

The results show a clear benefit when using the ArrayList implementation. For small numbers of work-items, the execution time is negligible. As the number of work-items increases, the performance benefit is significant. For 8192 work-items, we observe an outlier, likely caused by a non-ideal work-group size for the previous test cases, chosen to be multiples of 384. For 16384 threads on a GPU, the ArrayList test case executes 7 times faster than the `malloc()` test case. Improved speed is also visible for the CPUs.

Although the usability of the ArrayList depends greatly on the use case, this experiment shows that the "two-layered" KMA is not only a useful to improve programmability, but it can also facilitate optimisations for accessing custom data structures, when the use case permits.

5.3 Use case: edge list to graph

To show the use for KMA we implemented a small use case: a program that converts an unsorted list of arbitrary edges into a directed graph (e.g., as part of a graph-colouring- or a shortest-path algorithm). Given the unsorted list of edges, for which node ids and number of links per node are not known in advance, using statically allocated data structures such as arrays or hashmaps is challenging: their size is unknown, making significant overprovisioning mandatory. By using a memory allocator instead, we can allocate and free nodes as we go, thus minimising the memory overhead and keeping the code easy to read.

Our binary-tree is built from *node* and *edge* objects. A *node* object has a *key*, *left* and *right* pointers, and a linked-list to store the edges of the graph. An *edge* is an object containing a pointer to the next object in the list, and a pointer to the destination node. The node and edge objects are created by using the low-level KMA `malloc()` routine, after which they are added to the tree. If one attempts to add a node that was previously added, the error is detected and the node is freed again. The code snippets for the implementation of these functions is presented in Listings 1, 2 and 3. Each one of them illustrates a way to use `malloc()` and/or `free()` in such a program.

```
1  bool clTree_add(*tree, *node) {
2      uintptr_t *cur = &(tree->root);
3      clTree_node *cn;
4      node->left = node->right = NULL;
5      while(1) {
6          cn = atom_cmpxchg(cur, NULL, node);
7          if(cn == NULL) return true;
8          if(cn->key == node->key) return false;
9
10         if(node->key < cn->key) cur = &cn->left;
11         else cur = &cn->right;
12     }
13 }
```

Listing 1: Adding a node to the tree.

Listing 1 shows the `clTree_add()` routine for adding an allocated node to the tree. It takes a pointer to the topmost element of the tree, the *root*, as a parameter and assigns this as its *current* position in line 2. The routine attempts to add the node at the current position by means of the compare and exchange operation on line 6, where it tries to replace an empty pointer with a pointer to the newly created node. Failure of this CAS-operation indicates the memory location pointed to by the current pointer was taken by another node. When this occurs the routine traverses one level deeper as shown on line 10 and 11, after which the CAS-operation is retried on this new location. This process repeats until either adding the node was successful or the routine encounters a node with the same key as the node it is trying to add.

```
1  graph_node *ensure(*heap, *tree, int key) {
2      graph_node *node = NULL;
3      while(node == NULL) {
4          node = clTree_get(tree, key);
5          if(!node) {
6              node = malloc(heap,sizeof(graph_node));
7              if(!node)
8                  continue; /* or fail */
9              node->tree.key = key;
10             sll_init(&node->links);
11             if(!clTree_add(tree, &node->tree)) {
12                 free(heap, node);
13                 node = NULL;
14             }
15         }
16     }
17     return node;
18 }
```

Listing 2: Searching a node in the tree: if not found, add. Note the use of `malloc()` and `free()` for node allocation and deallocation.

The routine in Listing 2 will ensure that a node with a given key exists. It first issues a search for a node with the requested key using the `clTree_get()` function on line 4. This function is a regular tree search for which we omit the details for brevity. When the search returns no match, the routine allocates a new node from the heap on line 6 using `malloc()` and initialises this node. On line 11 it then tries to attach this node to the tree by calling `clTree_add()`. If

this returns true the node was successfully attached and it is returned, otherwise the node is freed again after which the routine will retry. On the second try, `clTree_get()` shall return a valid match unless the node was removed in the meanwhile.

```
1 source = ensure(heap, tree, edge->source);
2 sink = ensure(heap, tree, edge->sink);
3 link = malloc(heap, sizeof(graph_link));
4 link->sink = sink;
5 sll_add(&source->links, &link->q);
```

Listing 3: Using malloc to allocate a link

Listing 3 shows the code used to add a single edge to the tree. In the first two lines it calls the `ensure()` routine to ensure the source and sink nodes attached to this edge exist. Then it allocates a link object using `malloc()`, initialises the pointer and uses a singly-linked-list operation to attach this link to the source node.

This example code gives a brief demonstration of how KMA's kernel interface differs little from the C counterpart. The overhead on the host-side application code is minimal: a new heap is created by calling `clSBMalloc_create()`. This routine will allocate the heap and launch a kernel to initialise it on the device, after which the heap ready for use by any kernel that receives the pointer to it as a parameter.

To test the performance of this tree algorithm implementation, we used a dataset containing 64K edges and about 10K nodes. Both on the CPU and GPU this test case was executed with a varying number of work-items. To show the performance impact of KMA, we compared its performance to a "poormans"-malloc: a ringbuffer-like structure with a head-pointer. It has no free routine, and the `malloc()` routine is a simple atomic addition on the head pointer. This implementation has very low complexity, at the cost of not being a generic solution: free memory cannot be re-used. The poormans-heap test code is also slightly less naive than the KMA code; to control the amount of wasted memory it holds on to an allocated object for a possible next iteration where in the KMA codepath this block would be freed and re-allocated on the next iteration.

On the NVIDIA GPU, the static set-up time for the data structures needed for KMA to use a heap with 512 superblocks of 4KB each is $1.5ms$. For the poormans-heap the total set-up time is $0.2ms$. The difference of $1.3ms$ is spent on the single threaded initialisation kernel that enqueues all the superblocks to the free-list. The CPU shows similar numbers: $1.15ms$ with KMA versus $0.1ms$ for the "poormans"-heap.

Figure 7 presents the execution time of our test case relative to the number of work-items. The first thing to observe is that for more than 400 work-items, the performance does not increase any further on GPUs. Overall, the full application executes in approximately $28.6ms$, where the implementation using the poormans-heap finishes in $12.6ms$. With the naive usage of KMA, the total overhead is 56% of the execution time. We did observe a performance-peak at 1536 threads, that can be explained by the more ideal mapping of work-groups on the physical GPU. On the AMD CPU performance does not change with the number of work-items, which is not surprising given its six cores. We obtain $37.9ms$ and $20.7ms$ for KMA vs. non-KMA, respectively, meaning

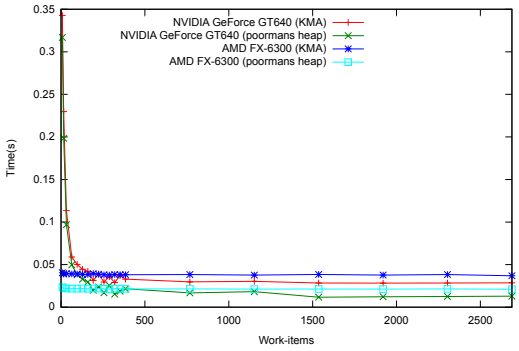

Figure 7: clTree performance: KMA vs. poormans heap.

that KMA adds a performance overhead of approximately 45%.

From these results, we conclude that there might be a big price to pay for the full functionality of a memory allocator. In some cases, this application-level penalty might become significant. However, our example has only measured the time spent on constructing the data structure, and not its use. For realistic cases, when complete applications combine data structure construction and usage, the relative overhead for using KMA will be proportionally smaller.

6. TO MALLOC OR NOT TO MALLOC?

This section discusses the strengths and weaknesses of KMA, and summarises the additional support needed to build a better KMA-2.

6.1 KMA: pros and cons

KMA is built to enhance OpenCL with a dynamic memory allocator for global memory by improving its programmability for irregular algorithms and data structures. Ideally, KMA should completely hide the complexity of explicit dynamic memory management behind the familiar `malloc()` and `free()` calls, or by using the customised higher-level APIs. However, in practice, a few issues need to be addressed before this statement becomes 100% true.

The most severe problems we encountered are related to portability on different OpenCL implementations:

- For Intel's SDK, the problem we encountered was related to the lack of flexibility in interpreting pointers as integers and vice-versa, which is required for arithmetic manipulation of the pointer.
- AMD's OpenCL implementation prevented us from running the memory allocator on their GPUs because the compiler offers too weak a memory ordering guarantee, though not an incorrect one. OpenCL 1.2 only requires that memory access reordering does not break the semantics of a single work-item, and that memory fences can be used to guarantee partial ordering between different work-items in a work-group. Global lock-free algorithms, like the ones used for our memory allocator, require stronger memory ordering guarantees to hold between different work-groups.
- NVIDIA's OpenCL implementation offers stronger memory ordering guarantees than required by the OpenCL 1.2 standard, translating a work-group wide memory

fence on global memory to a device-wide memory fence. This property allowed us to write a memory allocator that works correctly on NVIDIA GPUs. The same code works on AMD APP (AMD's OpenCL for CPUs) mainly because x86 CPUs utilise very little memory reordering techniques in the first place. On AMD GPUs there currently is no way to insert such device-wide memory fences.

On a different note, KMA focuses on global memory allocation only. We made this choice because we considered dynamic allocation is much more likely to be used on *large*, *shared* data objects rather than on cache-sized, work-group local objects. Furthermore, while local memory is typically improving the performance of GPGPU applications, other devices (especially CPUs, where local memory is not a dedicated memory space) suffer penalties when using it. Therefore, attempting to improve allocation by using local memory is a very complex endeavor, with unclear benefits, that we chose not to pursue.

In terms of performance, KMA does add a certain overhead by construction: dynamic allocation will always be slower than static allocation, as no memory manager has zero-latency. Thus, using KMA is a design decision any developer needs to take considering the trade off between the high-level KMA functionality, leading to higher productivity and maintainability, and its impact on performance.

6.2 Considerations for "KMA-2"

A lot of the design decisions taken for KMA are enforced by the platform limitations discussed in Section 4. Several directions can be be explored for a better KMA-2.

For *performance*, the main priority is to limit the memory wastage of KMA. For example, bigger blocks could be allocated when free blocks can be coalesced, which in turn requires an algorithm based on a lock-free ordered linked-list instead of one using a queue for the superblocks; however, it will also require a more complex KMA-2 backend. Alternatively, one can replace any lock-free algorithms with an efficient variant using mutex-style locks in OpenCL provided a feasible and well-performing implementation of these locks becomes possible.

For *portability*, we found that the memory model of OpenCL 1.2 is too relaxed to implement a portable lock-free algorithm. The newest OpenCL 2.0 specification offers a stronger, more controllable, memory model based on the C11 model and should support an implemention of our portable memory allocator without relying on undocumented behaviour of vendor implementations. When vendors implement OpenCL 2.0, we believe KMA-2 can be made fully portable across different families of platforms.

Of course, many new *features*, like supporting bigger block sizes for allocation, adding more high-level data structures, or using local memory to further improve performance can always be assessed, evaluated, and potentially implemented. However, we believe our KMA prototype provides the user with a sufficiently good base for prototyping and testing; new features will then naturally follow from users feedback.

7. CONCLUSIONS AND FUTURE WORK

In the context of massively parallel architectures - accelerators or not - dynamic memory allocation remains a difficult problem. Nevertheless, in search of better performance, many algorithms that involve irregular data structures are ported to OpenCL, and demand significant efforts to overcome OpenCL's restrictive, static-only memory allocation.

In this work, we provide an alternative to this situation: KMA, a first dynamic memory manager for OpenCL. KMA is based on a two-layer design: the lower-layer provides the familiar and flexible `malloc()` and `free()` routines, while the higher-level provides support for custom dynamic data structures and their management. An example of such a data structure (the ArrayList) is presented as a skeleton for future developments.

Our experiments show that KMA is fully functional and portable across several OpenCL platforms. The performance penalty for KMA is lower than that of CUDA's `malloc()` due to its simple and lock-free design, and it eventually scales linearly with the amount of work when parallelism is fully utilised. As the results reported by related studies [9, 19, 22] are comparable to those KMA achieved, we consider the performance of this first OpenCL dynamic memory manager very promising. Moreover, the ease-of use and demonstrated performance of the ArrayList prototype further proves our two-layer design is feasible. Overall, KMA is a useful tool for any developer that wishes to experiment with irregular data structures in OpenCL kernels.

Areas where KMA will improve are (1) reducing the memory fragmentation and overhead, (2) improving portability and (3) extending functionality. The first requires research towards more advanced algorithms for lock-free linked-list structures, or possibly the addition of mutex-style locks in the OpenCL standard. Portability problems are mainly caused by the very relaxed memory model prescribed by the OpenCL standard; more restrictive memory guarantees as described in the OpenCL 2.0 specification would solve this second issue. Research towards functionality should follow from user demands, but might include the allocation of bigger blocks than our current limit.

8. REFERENCES

[1] Parboil benchmark suite, 2010. http://impact.crhc.illinois.edu/parboil.php.

[2] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A view of the parallel computing landscape. *CACM 52* (Oct. 2009), 56–67.

[3] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not. 35* (November 2000), 117–128.

[4] Blelloch, G. E. Prefix sums and their applications. 35–60.

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE IISWC'09* (oct. 2009), pp. 44 –54.

[6] Dice, D., and Garthwaite, A. Mostly lock-free malloc. In *ISMM'02* (New York, NY, USA, 2002), ISMM'02, ACM, pp. 163–174.

[7] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271. 10.1007/BF01386390.

[8] Fang, J., Varbanescu, A. L., and Sips, H. A comprehensive performance comparison of cuda and opencl. In *ICPP'11* (September 2011).

[9] Huang, X., Rodrigues, C., Jones, S., Buck, I., and mei Hwu, W. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *IEEE CIT'10* (July 2010), pp. 1134 –1139.

[10] Khronos. *The OpenCL specification*, nov. 2012.

[11] Lea, D. A memory allocator, October 2000. http://g.oswego.edu/dl/html/malloc.html.

[12] Michael, M. M. Scalable lock-free dynamic memory allocation. In *PLDI'04* (New York, NY, USA, 2004), PLDI '04, ACM, pp. 35–46.

[13] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Tech. rep., Rochester, NY, USA, 1995.

[14] nVidia corp. *nVidia Cuda C Programming guide*, apr. 2012.

[15] Penders, A. Accelerating graph analysis with heterogeneous systems. Master's thesis, Delft University of Technology, December 2012. http://repository.tudelft.nl/view/ir/uuid:7f3eeb52-77bd-4fdb-84a9-ea9ca0a35b94.

[16] Rosenburg, O., Gaster, B. R., Zheng, B., and Lipov, I. *OpenCL Static C++ kernel language extension*, dec. 2011.

[17] Shen, J., Fang, J., Sips, H., and Varbanescu, A. L. Performance gaps between openmp and opencl for multi-core cpus. In *ICPPW'12* (September 2012).

[18] Spliet, R. A comprehensive study of dynamic memory management in opencl kernels. Master's thesis, Delft University of Technology, June 2013. http://repository.tudelft.nl/view/ir/uuid:b26fb6be-a3e1-4968-9044-01fc6a029926.

[19] Steinberger, M., Kenzel, M., Kainz, B., and Schmalstieg, D. Scatteralloc: Massively parallel dynamic memory allocation for the gpu. In *Innovative Parallel Computing (InPar), 2012* (2012), pp. 1–10.

[20] Sundell, H., and Tsigas, P. Lock-free deques and doubly linked lists. *JPDC 68*, 7 (July 2008), 1008–1020.

[21] Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E. P., and Ioannidis, S. Regular expression matching on graphics hardware for intrusion detection. In *RAID '09* (Berlin, Heidelberg, 2009), RAID '09, Springer-Verlag, pp. 265–283.

[22] Widmer, S., Wodniok, D., Weber, N., and Goesele, M. Fast dynamic memory allocator for massively parallel architectures. In *GPGPU-6* (New York, NY, USA, 2013), GPGPU-6, ACM, pp. 120–126.

[23] Xiao, S., and chun Feng, W. Inter-block GPU communication via fast barrier synchronization. In *IPDPS '10* (2010), IPDPS.