IN3200/IN4200: Chapter 1
Modern processors

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objectives of Chapter 1

- A *high-level overview* of the architecture of modern cache-based microprocessors
- Introduction of important concepts, which will be useful for writing efficient code later
- Discussion of inherent performance limitations

“Stored-program computer”

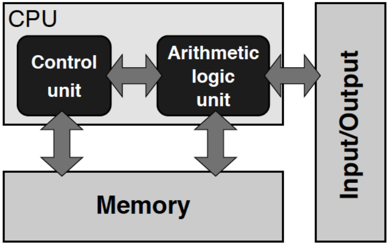


Figure 1.1: Stored-program computer architectural concept. The “program,” which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

The "stored-program computer" concept

- Instructions (produced by a *compiler*) and data are stored in memory
- Instructions are read and executed by a control unit
- An arithmetic/logic unit "does the work" which is coded in the instructions
- The speed of memory determines how fast instructions and data can be fed to the control and arithmetic units—limitation of performance
- I/O facilities enable interaction with users

CPU

CPU (central processing unit)—is the "brain" of a computer.

CPU incorporates control and arithmetic units (and many other components), together with appropriate interfaces to memory and I/O.

CPU has a "clock", which at each *clock cycle* synchronizes the logic units within the CPU to process instructions.

Cache-based microprocessor

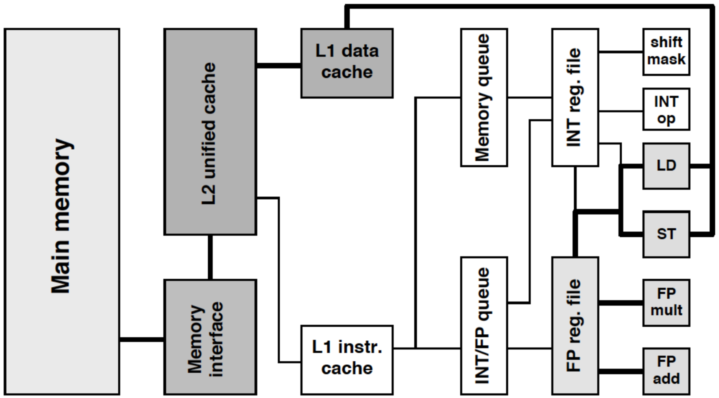


Figure 1.2: Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

Important hardware components

- Arithmetic units for floating-point (FP) and integer (INT) operations
- Registers hold operands to be accessed by instructions
- Load (LD) and store (ST) units handle instructions that transfer data to and from registers
- Instructions are sorted into several queues, waiting to be executed (probably not in the order they were issued)
- Caches hold data and instructions to be (re-)used

Pipelined functional units

Subdividing complex operations into simple components that can be executed using different functional units, it is possible to increase *instruction throughput*—the number of instructions executed per clock cycle.

This is the most elementary example of *instruction-level parallelism* (ILP).

Optimally pipelined execution leads to a throughput of one instruction per cycle per pipeline.

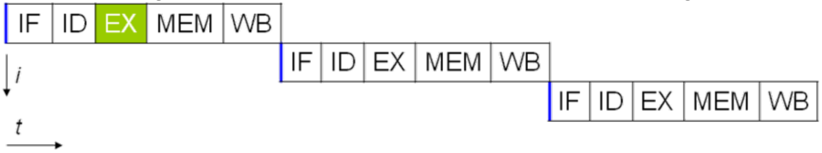
Pipelining

- Pipelining in microprocessors follows the same principle of assembly lines in manufacturing: Workers (functional units) are highly skilled and specialized for a single task.
- Each worker executes the same step, over and over again, on different objects.
- If it takes m different steps to finish the product, m products are continuously worked on, in different stages of completion.
- If all tasks take the same amount of time, and all workers are continuously busy, eventually (after the initial m steps) one product will be finished per time step.

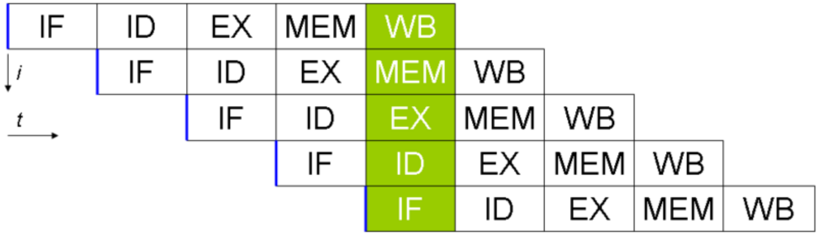
A simple example of no pipelining

For simplicity, let us suppose every instruction has five stages, each taking one cycle.

The following picture shows the situation of no pipelining:



The situation with instruction pipelining



More about pipelining

Complex operations such as loading and storing data or performing floating-point arithmetic cannot be executed in a single cycle. The "fetch–decode–execute" pipeline is thus applicable, in which each stage can operate independently of the others.

These still complex tasks are usually broken down even further. The benefit of elementary subtasks is the potential for a higher clock rate as the functional units are kept simple.

Example of "vector product"

Arrays of floating-point values: `A B C`

```
for (i=0; i<N; i++)
  A[i] = B[i] * C[i];
```

Suppose a floating-point multiplication is decomposed into five subtasks. (A floating-point value is $(\mathrm{sign}) \times \mathrm{mantissa} \times 2^{\mathrm{exponent}}$.)

1. separation of mantissa and exponent on `B[i]` and `C[i]`
2. multiply mantissas of `B[i]` and `C[i]` (recall that a mantisa is a binary fraction with non-zero leading bit)
3. add exponents of `B[i]` and `C[i]`
4. normalize result
5. insert sign

Depiction of a pipeline

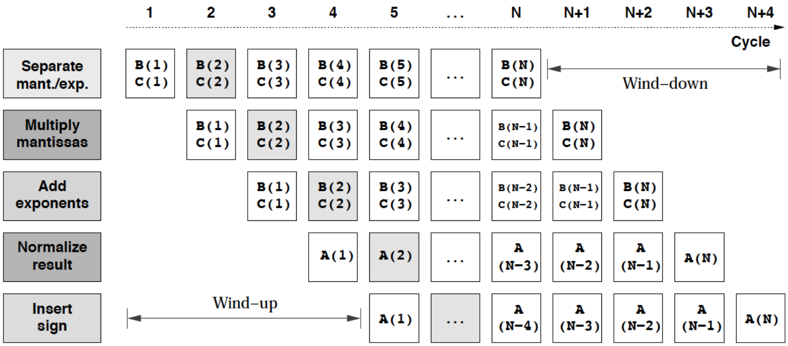


Figure 1.5: Timeline for a simplified floating-point multiplication pipeline that executes `A(:)=B(:)*C(:)`. One result is generated on each cycle after a four-cycle wind-up phase.

Simple mathematical model for pipelining

An m-stage pipeline has *latency* (or *depth*) of m cycles. The *wind-up* and *wind-down* periods are both $m-1$ cycles.

For a pipeline of depth m, executing N independent operations takes $N+m-1$ cycles. The speedup versus "no pipeling" is

$$\frac{T_{\mathrm{seq}}}{T_{\mathrm{pipe}}} = \frac{N \cdot m}{N+m-1}$$

The throughput, average number of operations finished per cycle, can be calculated as

$$\frac{N}{T_{\mathrm{pipe}}} = \frac{1}{1+\frac{m-1}{N}}$$

Example of pipeline throughput

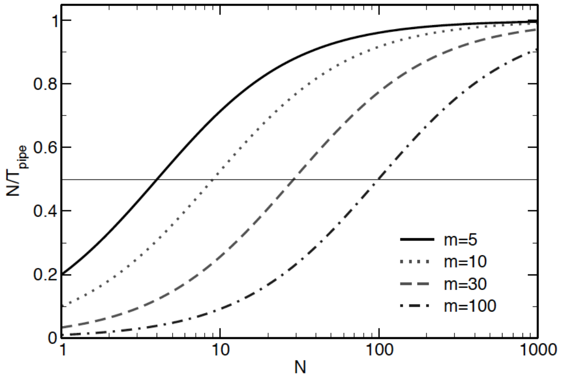


Figure 1.6: Pipeline throughput as a function of the number of independent operations. *m* is the pipeline depth.

Pipeline bubbles

Very complex calculations (like floating-point division or special math functions) tend to have very long latencies, and are only pipelined to a small level or not at all. In such cases, stalling the instruction stream becomes inevitable, leading to so-called "*pipeline bubbles*".

Avoiding such complex functions, if possible, is a useful technique for code optimization (to be discussed in Chapter 2).

Superscalarity

Goal: To produce more than one "result" per cycle.

- Multiple instructions are fetched and decoded concurrently
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units
- Multiple floating-point pipelines run in parallel
- Caches are fast enough to sustain more than one load or store operation per cycle

Superscalarity is a special form of parallel execution, and a variant of ILP.

Out-of-order execution and compiler optimization must work together to fully exploit superscalarity.

SIMD

The SIMD (single-instruction-multiple-data) concept became widely known with the first vector supercomputers in 1970s.

Modern cache-based processors have instruction set extensions for both integer and floating-point operations. They allow the concurrent execution of arithmetic operations on “wide” registers, each holding multiple numerical values.

Example of SIMD

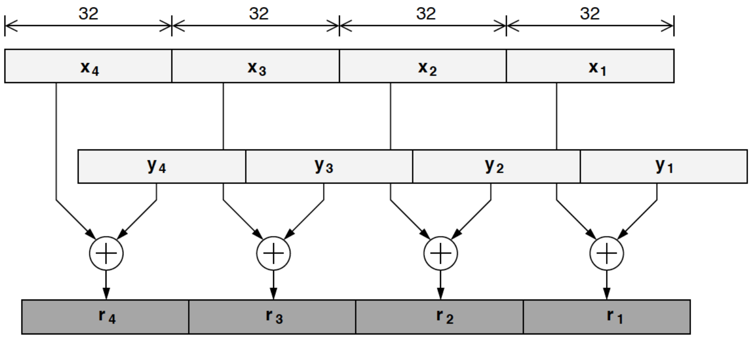


Figure 1.8: Example for SIMD: Single precision FP addition of two SIMD registers (x,y), each having a length of 128 bits. Four SP flops are executed in a single instruction.

Memory hierarchy

Data can be stored in a computer system in many different ways.

CPU has a set of registers, which can be accessed without delay.

In addition, there are several levels of *cache*, holding copies of recently used data items.

Main memory of a computer is much slower (than the caches).

Depiction of memory hierarchy

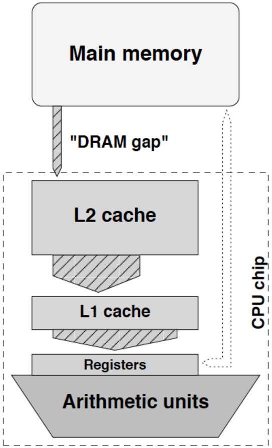
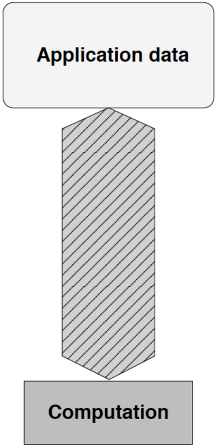


Figure 1.3: (Left) Simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. (Right) The “DRAM gap” denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.



Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die.

- L1 (level 1) data cache
- L1 instruction cache
- L2 and L3 unified caches

The purpose of cache—reducing the impact of main memory's small bandwidth and high latency.

Cache hit and miss

Whenever the CPU issues a read request (“load”) for transferring a data item to a register, the L1 data cache is checked. If the wanted data item is found in L1, this is called a *cache hit*, otherwise a *cache miss* occurs.

In case of a cache miss in L1, data must be fetched from upper cache levels or, in the worst case, from main memory.

Cache eviction

If a data item needs to be loaded into a cache where all cache entries are occupied. One of the occupant entries has to be *evicted* by a hardware-implemented algorithm (typically following the *least-recently used* strategy) to give space.

Temporal locality

If data items loaded into a cache are to be used again "soon enough", then this is called good *temporal locality*.

Suppose accessing a data item in cache is a factor of τ faster than accessing the main memory. Let β denote the cache reuse ratio. Suppose access time to main memory is denoted by T_{m}, access time to cache thus $T_{\mathrm{c}} = T_{\mathrm{m}}/\tau$.

The average access time will be

$$T_{\mathrm{av}} = \beta T_{\mathrm{c}} + (1-\beta) T_{\mathrm{m}}$$

Performance gain due to cache can be calculated by

$$G(\tau,\beta) = \frac{T_{\mathrm{m}}}{T_{\mathrm{av}}} = \frac{\tau T_{\mathrm{c}}}{\beta T_{\mathrm{c}} + (1-\beta)\tau T_{\mathrm{c}}} = \frac{\tau}{\beta + (1-\beta)\tau}$$

Curves of performance gain

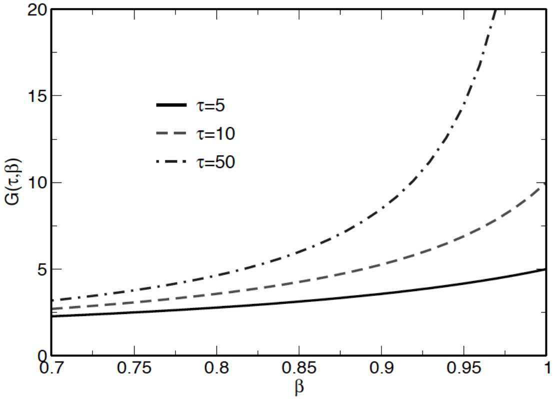


Figure 1.9: The performance gain from accessing data from cache versus the cache reuse ratio, with the speed advantage of cache versus main memory being parametrized by τ.

Cache lines

The content of a cache is organized as *cache lines*. (A cache line has space for multiple data items.) This is for reducing the latency penalty for *streaming*—large amounts of data are loaded into the CPU, modified, and written back without the potential of reuse "in time".

All data transfers between caches and main memory happen on the cache line level.

If a code has good *spatial locality*, that is, the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced.

Cache mapping

If a line of data items from main memory, to be loaded into cache, can be freely placed on any unoccupied cache line, it is called a *fully associative* mapping.

Unfortunately, it is hard to build large, fast and fully associative caches because of large bookkeeping overhead.

On the other end, a *directly-mapped* cache—a line of data items can be placed only on a prescribed cache line—runs the risk of low cache utilization.

Directly-mapped cache

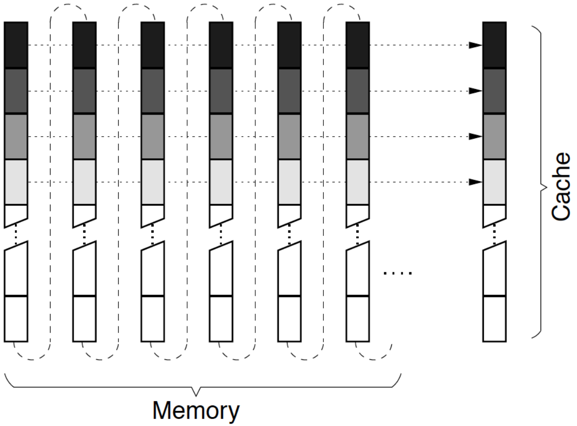


Figure 1.10: In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line (shaded boxes).

m-way associative

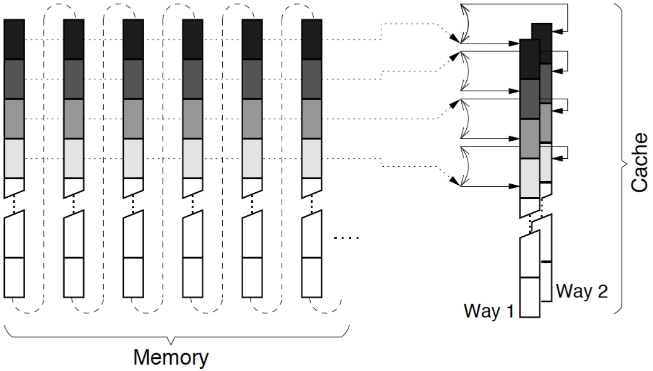


Figure 1.11: In an m-way set-associative cache, memory locations which are located a multiple of $\frac{1}{m}$th of the cache size apart can be mapped to either of m cache lines (here shown for $m = 2$).

The problem of "first cache miss"

Although exploiting spatial locality and cache lines can improve cache efficiency, there is still the problem of latency on the first miss.

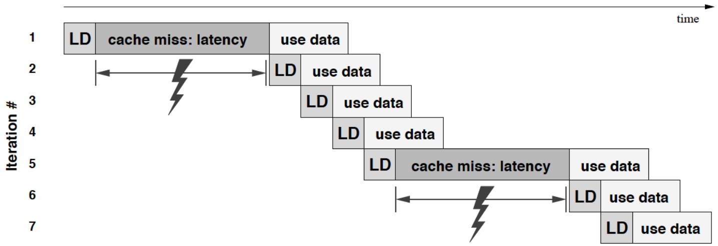


Figure 1.12: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.

Prefetch

Prefetching supplies the cache with data ahead of the actual requirements from an application code.

Typically, a hardware pre-fetcher can detect regular access patterns and try to read ahead the needed data.

To completely hide the cache miss latency, the memory subsystem must be able to sustain a certain number of outstanding prefetch operations.

How many outstanding prefetch operations needed?

If T_ℓ is the cache miss latency and B is the bandwidth.

Suppose each cache line is of length L_{c} (in bytes), then loading a cache line due to cache miss takes a time of

$$T = T_\ell + \frac{L_{\mathrm{c}}}{B}$$

The number of cache lines that can be transferred (without paying the latency penalty) during time T is the number of outstanding prefetches, P, that the processor must be able to sustain. So we have

$$P = \frac{T_\ell + \frac{L_{\mathrm{c}}}{B}}{\frac{L_{\mathrm{c}}}{B}} = 1 + \frac{T_\ell}{\frac{L_{\mathrm{c}}}{B}}$$

Prefetch helps to overlap computation with data transfer

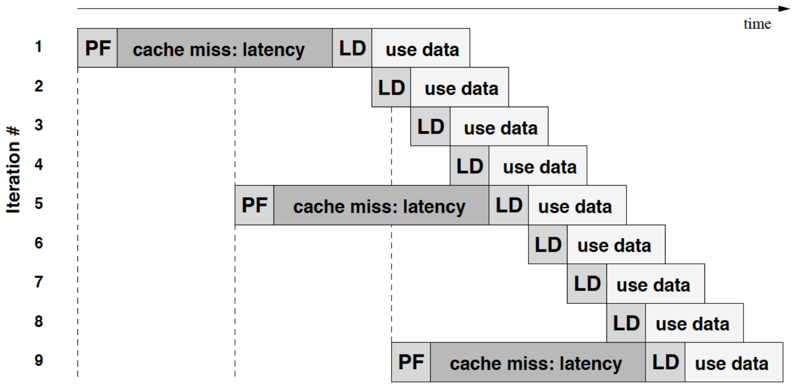


Figure 1.13: Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.

Multithreaded processors

All modern microprocessors are heavily pipelined. In case there are frequent “pipeline bubbles” caused by, for example,

- dependencies
- memory latencies
- insufficient loop length,
- branch mispredictions

The consequence is that a large part of the execution resources is idle (wasted resources).

Multithreading

Hyper threading or *simultaneous multithreading* (SMT) capabilities are thus built into modern processors.

- Multiple architectural states of a CPU core
- An architectural state comprises all data, stauts and control registers
- However, resources such as arithmetic units, caches, queues, memory interfaces are *not* duplicated

One CPU core “appears to be composed of several cores (also called *logical processors*). Multiple instruction streams, or threads, can be executed in parallel.

SMT

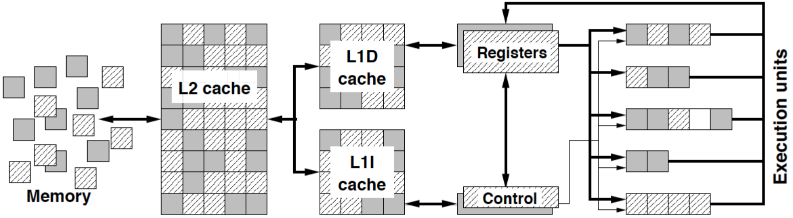


Figure 1.20: Simplified diagram of control/data flow in a (multi-)pipelined microprocessor with fine-grained two-way SMT. Two instruction streams (threads) share resources like caches and pipelines but retain their respective architectural state (registers, control units). Graphics by courtesy of Intel.

All threads share the same execution resources, so sometimes it is *possible* to fill pipeline bubbles that arise due to installs in one thread. SMT *may* enhance instruction throughput (instructions executed per cycle).

Whether the concept of SMT pays off is code-dependent and hardware-dependent!

Performance metrics

Theoretically, the components of a CPU core can operate at some maximum speed called *peak performance*.

Whether this limit can be reached for a specific application code depends on many factors (one of the key topics of Chapter 3).

Performance metrics:

- The performance at which the floating-point units generate results for multiply and add operations is measured as *floating-point operatins per second* (Flops/sec).
- The most important data paths are those to and from the caches and main memory. The performance, called *bandwidth*, of these paths is quantified in GBytes/sec.

Multicore processors

A higher clock frequency will allow a CPU to execute the instructions faster. However, Increasing the clock frequency can have a serious impact on the power dissipation.

On the other hand, reducing the clock frequency allows placing more than one CPU core on the same CPU die (or more generally, the same package), while keeping the same power envelope.

More about multicore CPU

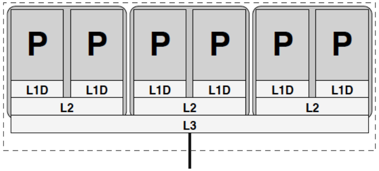


Figure 1.17: Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores (Intel "Dunnington"). L2 groups are dual-cores, and the L3 group is the whole chip.

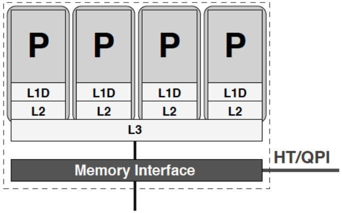
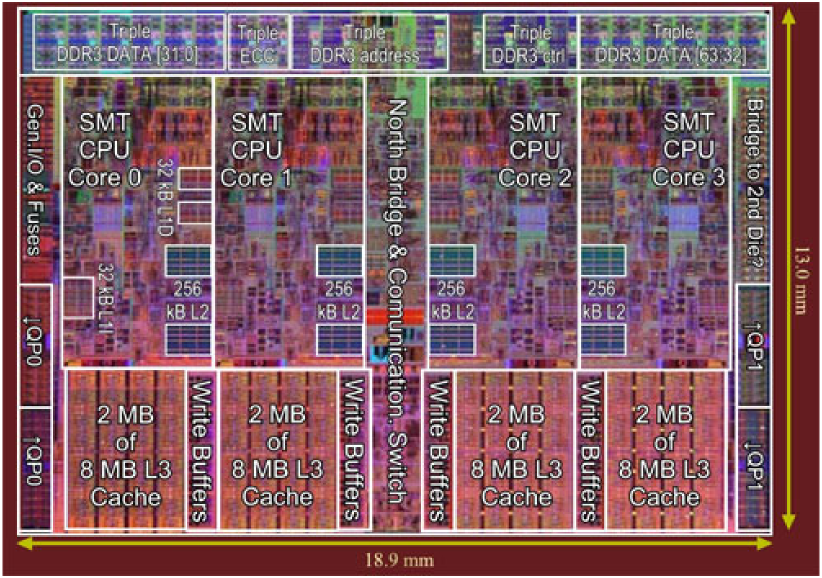


Figure 1.18: Quad-core processor chip with separate L1 and L2 and a shared L3 cache (AMD "Shanghai" and Intel "Nehalem"). There are four single-core L2 groups, and the L3 group is the whole chip. A built-in memory interface allows to attach memory and other sockets directly without a chipset.

The caches on the different levels can be private or shared. Sharing a cache enables superfast communication between the cores. An opposite effect of sharing can be cache bandwidth bottlenecks.

Example of a 4-core CPU



Intel Nehalem (actually a very old CPU)

Challenges with using multicore CPUs

- In order to use all the resources belonging to the multiple cores, parallel programming must be adopted. (This will be one of the topics for later lectures.)
- The memory bandwidth available per core can be a challenge. So programming techniques for memory traffic reduction will be even more important!

Vector processors

- An very important processor architecture for HPC in the past
- However, some of the concepts and techniques related to *vectorization* are still used today

Basic ideas

- Instructions operate on *vector registers* that can hold a large number of arguments
- The width of a vector register is called the *vector length* L_{v}
- MULT and ADD pipelines are *multitrack*
- One or several load, store or combined load/store pipes are connected *directly to main memory*

The paradigm of SIMD

Block diagram of vector processor

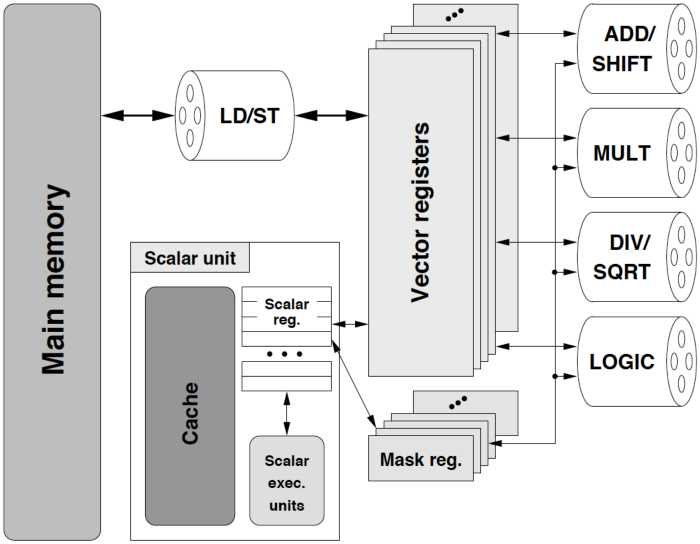


Figure 1.21: Block diagram of a prototypical vector processor with 4-track pipelines.

SIMD for $A = B + C$

Target calculation

```
for (s=0; s<N, s++)
  A[s] = B[s] + C[s];
```

A vectorization-capable compiler will automatically translate into the following pseudocode:

```
for (s=0; s<N, s+=L) {
  int E = min(N-1,s+L-1);
  vload V1(0:L-1) = B(s:E);
  vload V2(0:L-1) = C(s:E);
  vadd V3(0:L-1) = V1(0:L-1) + V2(0:L-1);
  vstore A(s:E) = V3(0:L-1);
}
```

`V1 V2 V3`: vector registers, `L`: vector length L_v

Vectorization

Writing a program so that the compiler can generate effective SIMD vector instruction is called *vectorization*.

Sometime this requires reformulation of code or inserting directives to help the compiler identify SIMD parallelism.

If a code cannot be vectorized, it makes no sense to use a vector computer!

A prerequisite for vectorization is true data independence across iterations of a loop. (Forward references are allowed, but not backward references.)

Branches in vectorized loops (example 1)

```
for (i=0; i<N; i++) {
  if (y[i] < 0.)
    x[i] = s*y[i];
  else
    x[i] = y[i]*y[i];
}
```

Mask registers

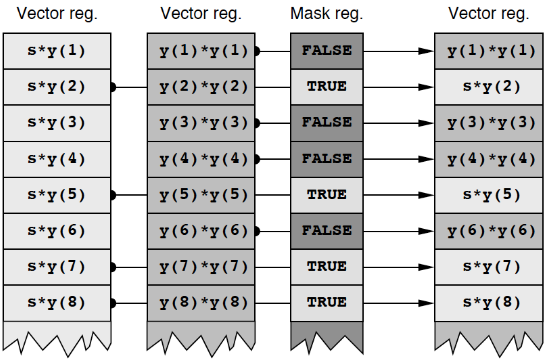


Figure 1.23: On a vector processor, a loop with an if/else branch can be vectorized using a mask register.

First, a vector of boolean values is generated by the logic pipeline. Then both branches are executed for all loop indices. Finally the boolean vector is used to choose the correct results.

Branches in vectorized loops (example 2)

```
for (i=0; i<N; i++) {
  if (y[i] > 0.)
    x[i] = sqrt(y[i]);
}
```

There is only the `if` branch (no `else` branch). Also, the `sqrt` calculation is expensive. Execution for all loop indices can be a huge waste of resource. What should be done in such a case?

The gather/scatter method

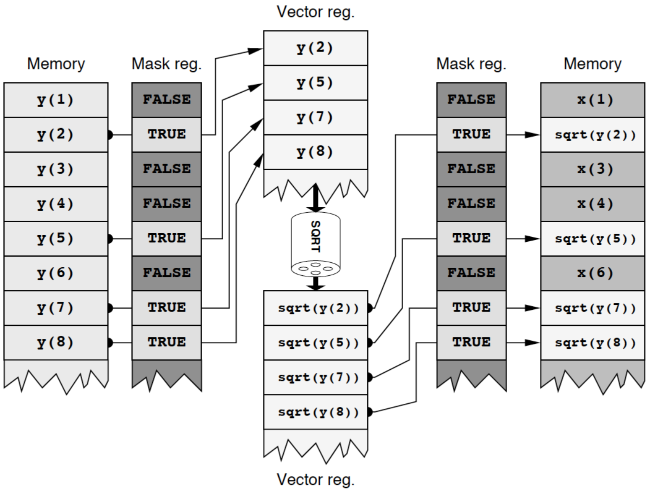


Figure 1.24: Vectorization by the gather/scatter method. Data transfer from/to main memory occurs only for those elements whose corresponding mask entry is true. The same mask is used for loading and storing data.