Chapter 1: modern processors (and their impact on software performance)

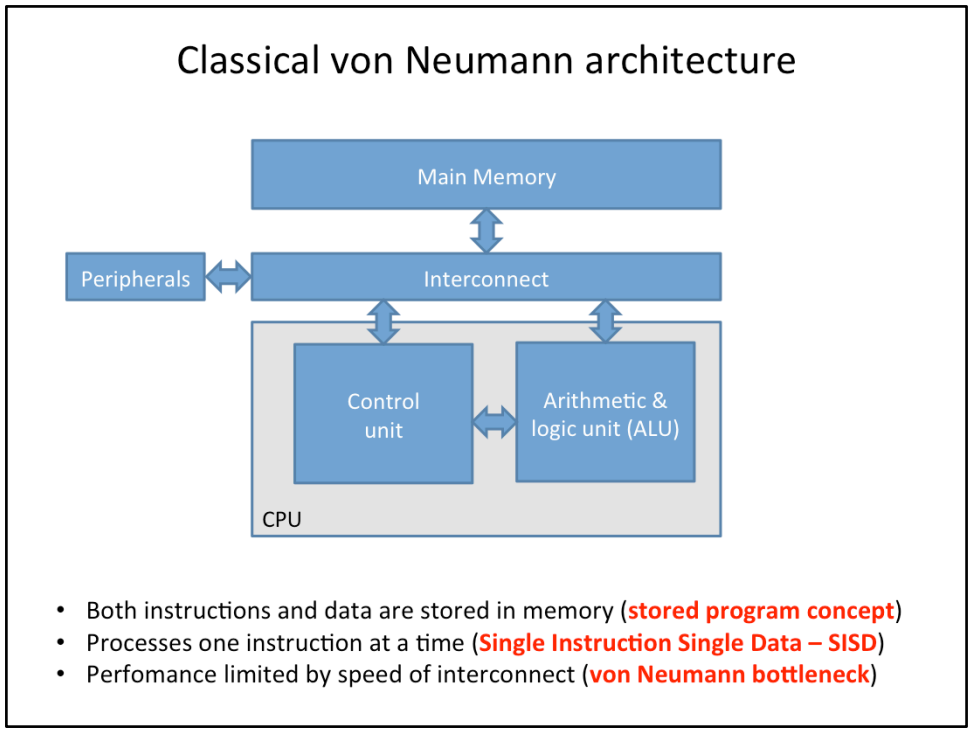
In this chapter, we will review certain architectural concepts (caching / pipelining / SIMD) that were introduced in earlier courses (like Computer Architecture) and discuss their impact on software performance, using a number of well-known examples.

Chapter 1: outline

- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK

Chapter 1: outline

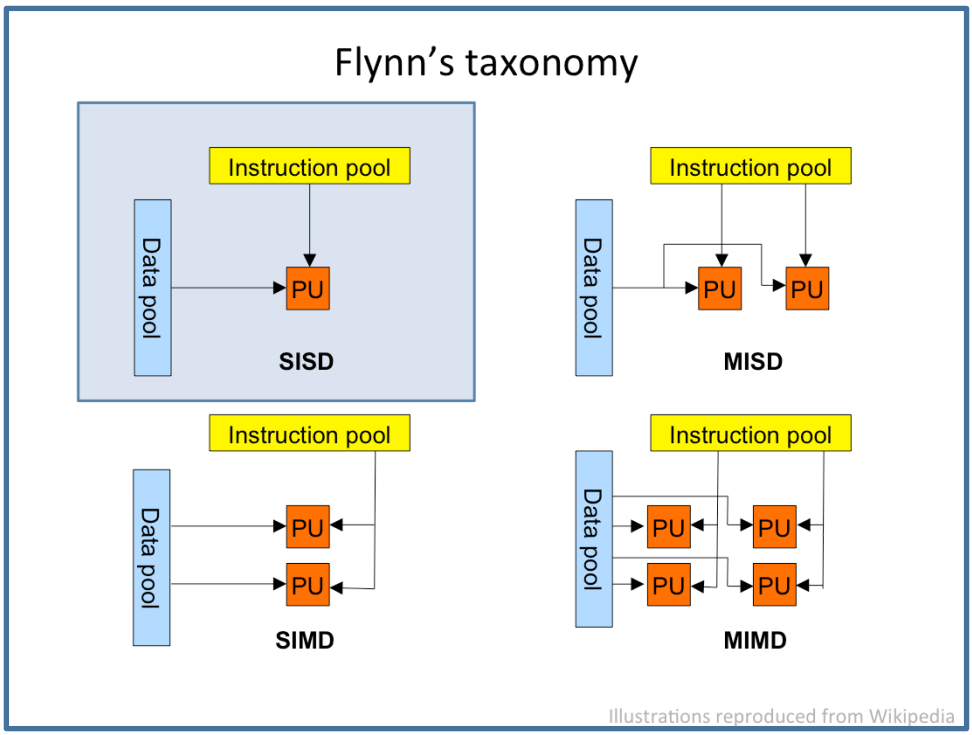
- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK



The classical von Neumann architecture consists of a **central processing unit** (CPU), **main memory** and an **interconnection** between CPU and memory.

- The CPU roughly consists of a **control unit**, which controls the program flow (order in which the instructions are executed) and an **arithmetic and logic unit** (ALU).
- The main memory contains both data and instructions ("**stored program concept**")
- The **interconnect** (typically used to be a bus concept, now largely replaced by point to point interconnects like Quickpath Interconnect (Intel) or Hypertransport (AMD)) controls the flow of data between memory, CPU and other **peripherals** (graphics card, network interface, etc.). The interconnect determines the rate at which instructions and data can be accessed by the CPU. As moderns CPUs can execute instructions more than 100 times faster than the rate at which data can be fetched from main memory, the interface is often called the **von Neumann bottleneck** or even the **DRAM gap**.

This architecture is **inherently sequential**, processing a single instruction at a time on a limited number of operands which are fetched from memory.

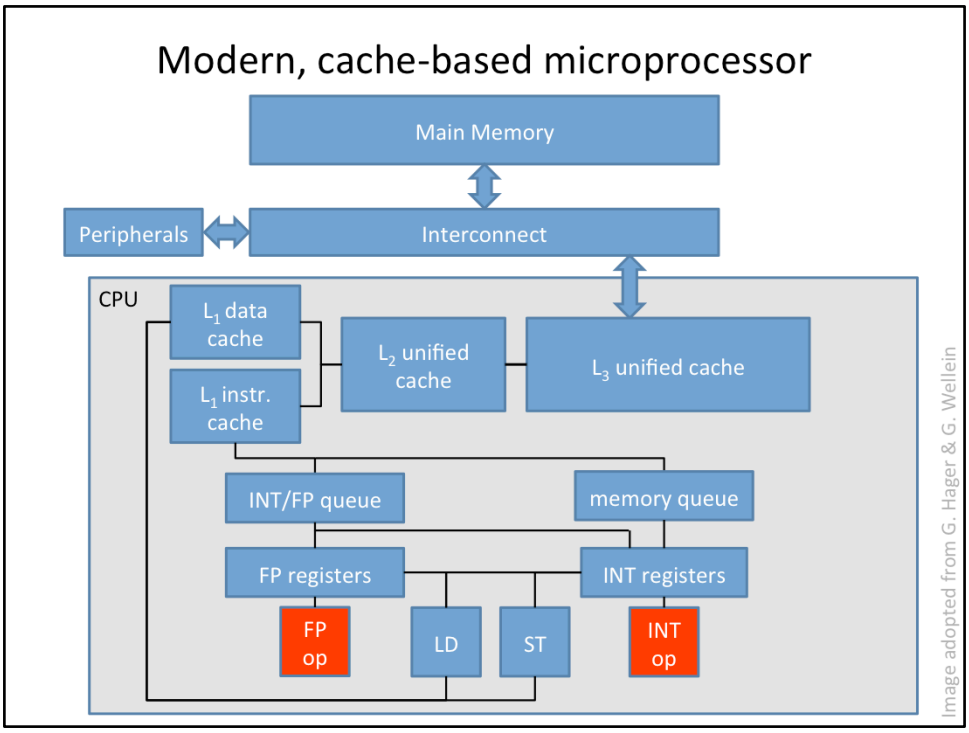


Taxonomy is the science of classification. Flynn created a classification for computer architectures based on

a) **The number of instruction streams** (single instruction stream versus multiple instruction streams)
b) **The number of data streams** (single data stream versus multiple data streams).

This leads to four classifications:

a) **SISD**: the classical von Neumann architecture.
b) **SIMD**: vector instructions, vector computers, or even GPUs can be regarded as such (see further).
c) **MIMD**: parallel architectures like symmetric multiprocessing, distributed-memory clusters.
d) **MISD**: often regarded upon as not very useful. One example could be the redundant execution of an instruction stream on the same data to check whether the results are consistent, to be able to recover from faults.



Many modifications to this design have been introduced to improve its performance. The slide shows a simplified block diagram of a modern, cache-based microprocessor.

- The **CPU registers**, typically divided in floating point and integer/general purpose registers, hold operands on which the instructions operate with almost no delay. Modern CPUs have tens of registers of different width (64 bit, 128 bit, 256 bit) for different purposes. The LD (load) and ST (store) units handle the data transfer from (cache) memory to the registers.
- The **arithmetic unit** is also typically divided in floating point (FP add, FP multiply) and integer operations.
- The **instruction queues** buffer and sort instructions. They can prefetch new instructions from memory and change the order in which the instructions are executed (**out-of-order execution**, see further).
- The **caches** hold data and/or instructions that are (expected) to be (re-)used soon.

Note that this scheme depicts a single-core CPU. Modern CPUs typically contain several cores, for which the cores typically share some of the components depicted here (e.g. the highest level cache).

Note that “unified” denotes that the cache is used to store both data and instructions. The L1 cache is usually separated in a data cache and instruction cache for improved performance.

Performance terminology

- **FLOPS or FLOPS/s**: FLoating-point OPerations per Second
  - Used to quantify peak floating-point performance for scientific codes
  - Sandy Bridge CPUs deliver up to 8 DP or 16 SP operations per cycle per core
  - At frequency of 3 GHz: peak performance of **24 (DP) or 48 (SP) GFLOPS/s** per core
  - Graphical Processing Unit (GPU): peak performance > **1 TFLOPS/s**
- **Bandwidth (BW)**: Rate of data transfer (bytes per second)
  - Modern DRAM: **10-25 GByte/s**
  - Modern L_1 cache: 32 bytes / cycle reading + 16 bytes / cycle writing = **96 GByte/s** reading + **48 GByte/s** writing (at 3GHz)
- **Latency**: Time to initiate a data transfer
  - Modern DRAM: **~100 clock cycles**
  - Modern L_1 cache: **~4-6 clock cycles**

⇨ RAM memory **has insufficient BW** to feed operands to a CPU operating at peak performance (cfr. von Neumann bottleneck). This explains the need for *cache memory*.

In communications, the latency is sometimes defined as “the time to send an zero-sized message”.

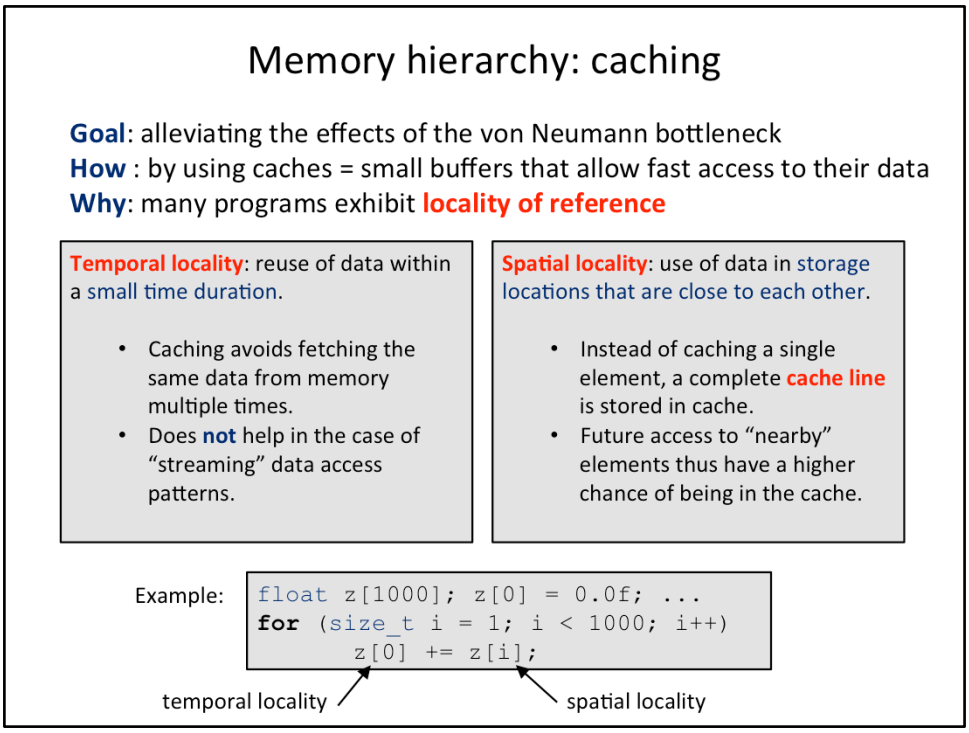
DP = double precision (8 bytes), SP = single precision (4 bytes)

Note that for computations that do not rely on floating-point arithmetic (e.g. a database), **MIPS (Millions of Instructions Per Second)** is a more appropriate way to measure CPU performance. The instructions per second is the product of the clock frequency (Hz) and the number of instructions per clock cycle (typically value ranges between 1 and 4).

The numbers provided in this slide are for the Intel Sandy Bridge architecture. The level 2 cache has a latency of 12 cycles and a bandwidth of 32 bytes / cycle. The level 3 cache has a latency of 26-31 cycles and a bandwidth of 32 bytes / cycle. The level 3 cache however, is shared between the different cores in a multi core CPU.

Chapter 1: outline

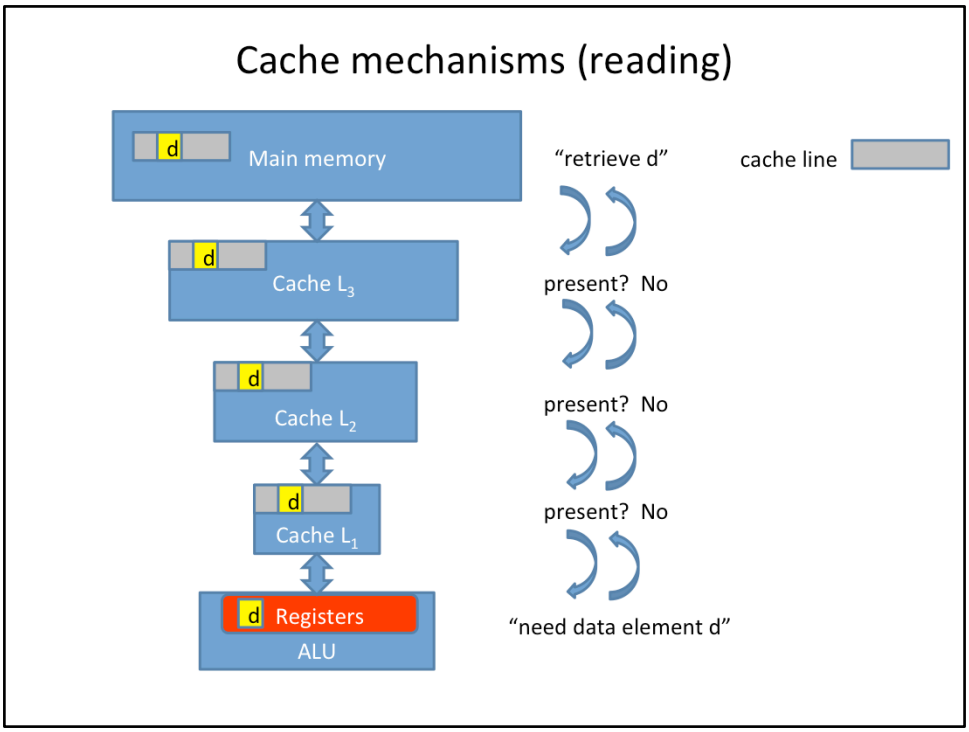
- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK



Caches are small, but very fast buffers which are typically integrated on the same chip as the CPU. Caches have a reduced latency and a higher bandwidth, compared to the main memory. Therefore, reading from (or writing to) cache is faster than reading data from (writing data to) the main memory. Caches are used to (temporarily) store either (a) **duplicated data from the main memory** or (b) **recently computed values that have not yet been written to the main memory**, in order to satisfy future accesses to those data in a fast manner. Typically, CPUs have two or three levels of cache memory (referred to by L1, L2 and L3), where the lower level caches are smaller, but also faster. Accessing a memory location that is stored inside the cache is called a **cache hit**. In the other case (**a cache miss**), the data needs to be retrieved from main memory (or higher level caches).

Caches are completely transparent to the programmer (i.e., the programmer does not need to know about its existence), however, many algorithms can be optimized for improved cache behavior. Caches are only useful in case the access pattern exhibits some **locality of reference**: temporal locality or spatial locality.

- **Temporal locality**: subsequent accesses that are close in time to the same data.
- **Spatial locality**: accesses to data that are stored physically close to each other in the main memory (that nearly have the same address).



When data is required (e.g. instructions or operands to perform an instruction) by the CPU, the L1 cache is checked for the presence of this data. If the data is present (a cache hit), this data is transferred to the CPU. In the other case (a cache miss), the L2 cache is checked and so on. In this example, the data is initially present in main memory only, and is subsequently stored in all cache levels for future use (**cfr. temporal locality**). Note that not only the data element "d" is retrieved from memory, but a complete **cache line** (typically 64 bytes) that contains d (**cfr. spatial locality**). This will strongly reduce the latency for future accesses to elements stored in the same cache line as "d". When storing the a cache line in memory, it has to replace an existing cache line. We say the old cache line is "**evicted**" from memory.

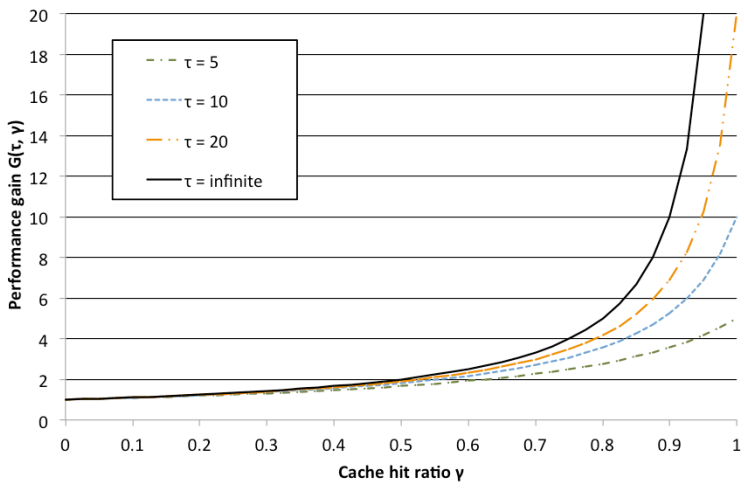
Cache performance model

- Access time to main memory is T_m *(both BW and latency)*
- Access time to cache is $T_c = T_m / \tau$
  - ⇨ Cache is τ times faster than main memory, $\tau \geq 1$
- Let γ be the **cache hit ratio,** $0 \leq \gamma \leq 1$
  - ⇨ Fraction of accesses that can be satisfied by the cache because of temporal and spatial locality.
- Average access $T_{av} = \gamma T_c + (1- \gamma)T_m$

Performance gain $G(\tau, \gamma) = Tm/Tav = \tau/\gamma + (1-\gamma)\tau$

In this simple performance model, we assume that the access time both refers to bandwidth and latency.

Cache performance model



- **G(τ, γ) = 1 for γ = 0** (all data from main memory)
- **G(τ, γ) = τ for γ = 1** (all data from cache)

Image reproduced from G. Hager & G. Wellein

Only for high cache hit ratios, there is a significant performance gain. For example, for a hit ratio of 50%, the maximum gain is only a factor of 2, even if the cache is access time is zero (tau = infinite).

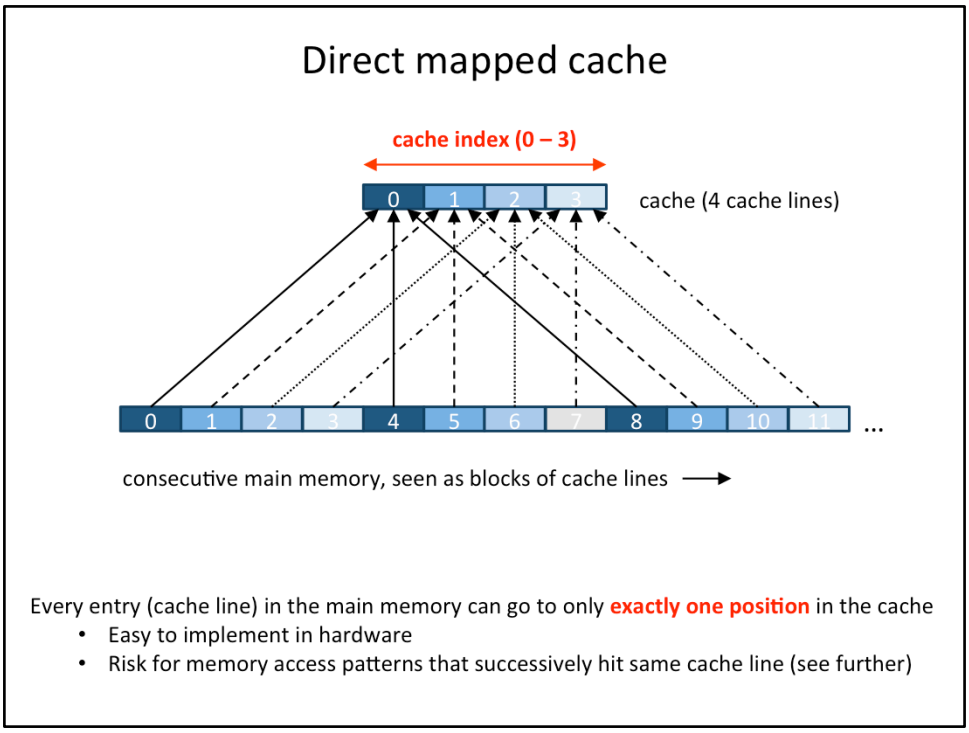
Cache writing

- On a **write hit**: two possibilities:
  - **Write-through**: update value in cache and backing store
    - Ensures consistency between cache and backing store
  - **Write-back**: write only to this cache level
    - Postpone writing to backing store until cache line is evicted
- On a **write miss**: two possibilities:
  - **Write allocate**: fetch cache line, followed by "write hit"
  - **No-write allocate**, a.k.a. nontemporal store: write directly to backing store
    - Write buffers to avoid excessive latencies
- Typical pairing:
  - Write-back in combination with write allocate
    - Benefits from subsequent writes to the same location
  - Write-through in combination with no-write allocate
    - No benefits from writing to same location
    - But also, no cache pollution from writing

Note: **backing store** = higher level cache or memory. For example: the backing store of the L1 cache is the L2 cache. The backing store of the L3 cache is the main memory (assuming there is no L4 cache).

Writing to the cache is slightly more involved than reading. In order to correctly understand and apply the definitions in this slide, the first question to ask is always "is the memory location to write to already present in the cache level that we are observing?"

- **Yes (write hit)**:
  - Write-through cache: the data in this cache level + backing store is updated to the new value. Note that if the backing store is again a cache of the write-through type, that its respective backing store will be updated as well, and so on. Note that the definition by Pacheco is therefore somewhat inaccurate: a L1 write-through cache does not necessarily mean that the data in main memory is immediately updated as well, because the L2 cache can be of the write-back type.
  - Write-back cache: the data is only written to this cache level, but not to the higher level cache levels. The data is therefore not consistent across the memory hierarchy. The cache line is marked as "**dirty**" (modified) and only written to the backing store when it is evicted (i.e., replaced by another

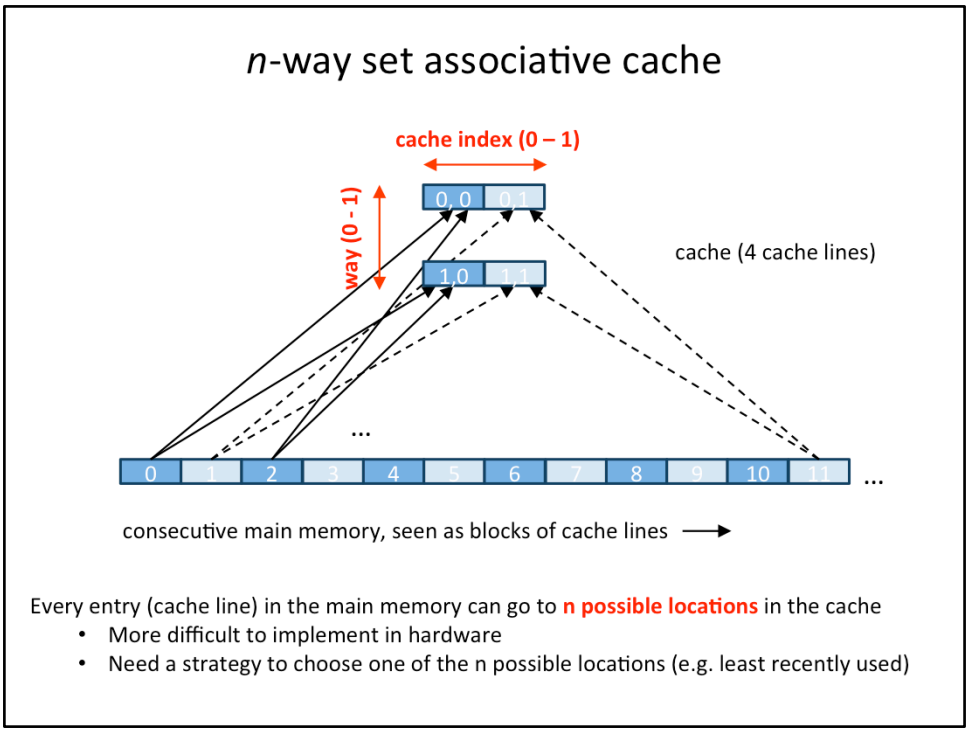


In a direct mapped cache, the cache index (= location in which a cache line can be stored) is simply

(memory address (expressed as bytes) / the number of bytes per cache line) % the number of cache lines in the cache

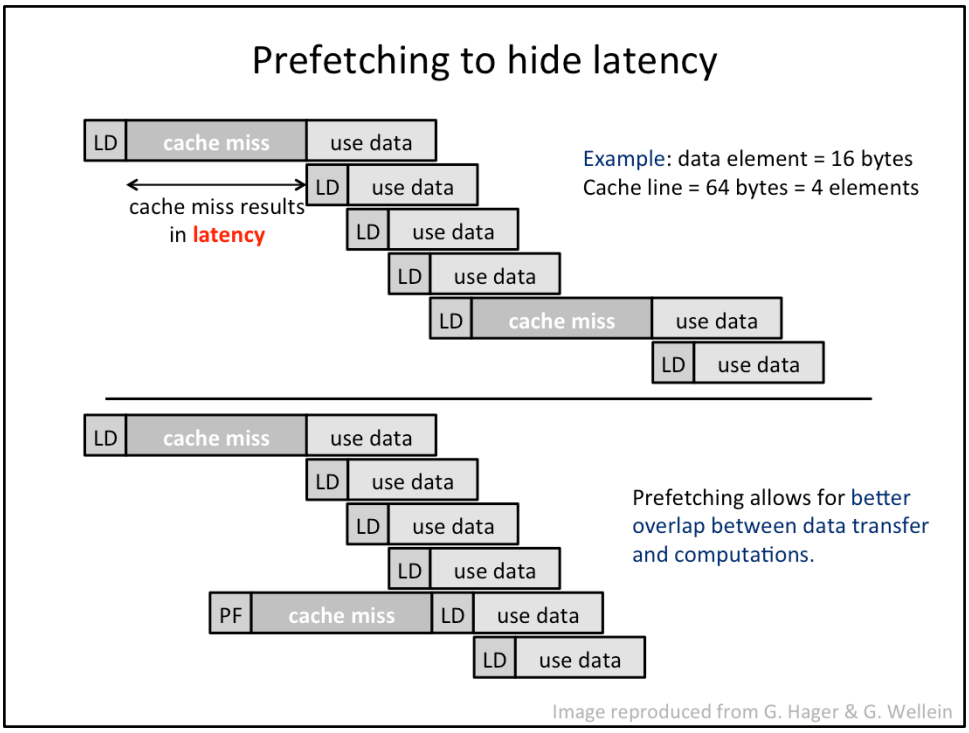
Here % denotes the modulo operation.

To see whether a memory location is present in the cache or not, only a single cache location needs to be checked. A direct mapped cache is vulnerable to so-called cache trashing: this is the phenomenon where successive memory accesses are mapped onto the same cache line, leading to a rapid succession of loading and evicting of data from and to that specific cache line.



In case a memory location can go to any cache line in the cache, the cache is said to be **fully associative**. This offers great flexibility, but is difficult to implement in hardware without excessive circuitry requirements. Indeed, in case a cache needs to be checked for the presence of a certain data element, the complete cache needs to be searched for that element.

Usually, a cache is somewhere "in between" a direct mapped cache and a fully associative cache: an **n-way set associative cache** is a cache where a certain data element can be stored at exactly n different places in the cache. Typical values for n are 4 to 16. E.g. for Sandy Bridge, the L1 and L2 caches are 8-way set associative, the L3 cache is 12-way set associative.



Because a cache miss results in rather long latencies, **prefetching** can be used to initiate the retrieval of data from main memory to the cache before the data is actually needed to perform the computations. A modern CPU contains hardware that monitors the data access patterns. For simple patterns (e.g. linear access), it can issue prefetching instructions in order to reduce the latency of the memory access. The compiler can also insert prefetching instructions if appropriate. Prefetching is only effective when memory data transfer and CPU instructions can overlap (asynchronous data transfer). This is the case for modern CPUs.

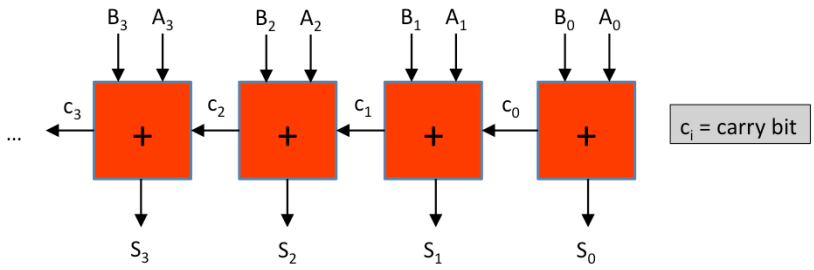
Important note: prefetching does not augment the bandwidth between CPU on the one hand and cache / main memory on the other hand. It can therefore not improve performance of code that is bandwidth limited.

Chapter 1: outline

- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK

Parallelism in a CPU core: bit-level parallelism

- E.g. adding two binary numbers: S = A + B

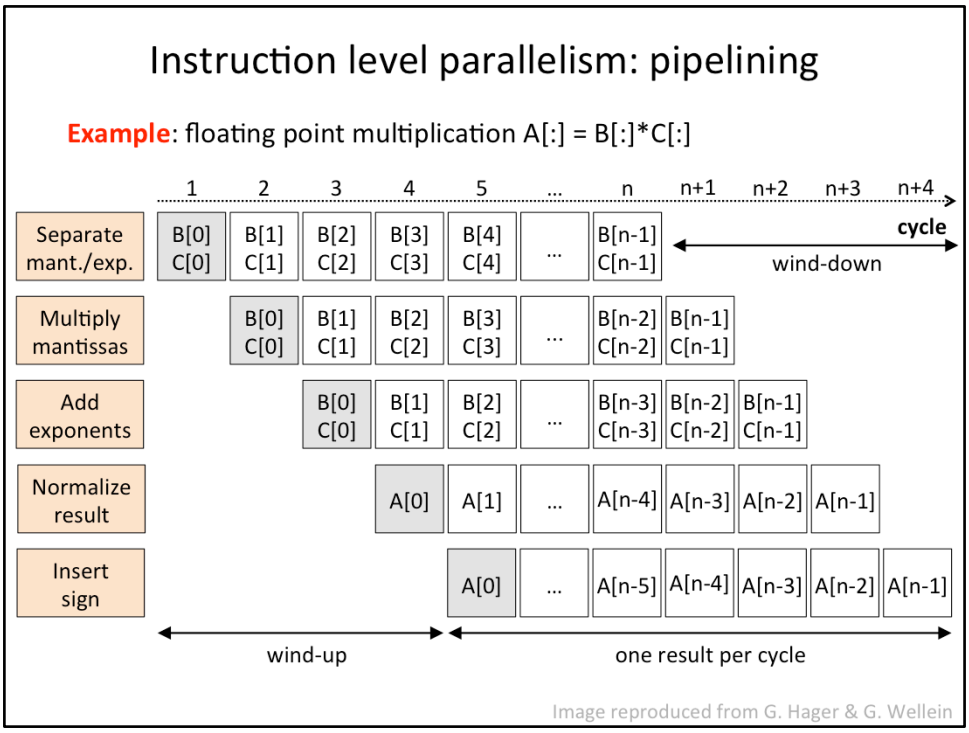


- Requires a fixed number of CPU cycles, regardless of A and B
- Determined by CPU word size
  - 4 bit e.g. Intel 4004 (1970)
  - 8 bit e.g. Intel 8008 (1971)
  - 16 bit e.g. IMP-16 (1973)
  - 32 bit e.g. AT&T Bellmac-32A (1981)
  - 64 bit e.g. AMD x86-64 (2000)
- Automatically done by CPU (with user & compiler awareness)

Bit level parallelism is a form of parallelism in which a CPU acts upon the different bits of a word in parallel. For example, a 32-bit CPU can add two binary 32-bit numbers A and B in a single clock cycle, regardless of the exact numbers that are encoded by A and B. If A and B would be 64-bit numbers, the same CPU would require two cycles: one for adding the least significant 32-bit portion of A and B, and a second cycle for adding the most significant portion of A and B. Therefore, increasing the word size of a CPU also increases the degree of bit-level parallelism.

Note that this form of parallelism does not mean that the values S_i are generated at the exact same moment. In this example, the sum is generated through a so-called ripple carry adder, in which the carry information in the figure propagates from right to left. There are hardware implementations that generate a sum faster than a ripple carry adder. However, from a user's perspective, the S_i bits are generated in a single instruction, and therefore "in parallel", even though this is strictly speaking not true from a physical point of view.

A second example of bit-level parallelism is e.g. a bus design, in which a number of parallel connections can effectively transfer e.g. 64 bits of data from/to DRAM in a single cycle.



Consider a car factory where the process of assembling a car can be split into a sequence of operations. At the beginning of the assembly line, the chassis is created. Next, the car moves to the next unit in the assembly line where e.g. the car's engine is put into the chassis after which the car again moves to next unit, etc. A purely sequential approach would be one where a single car is completely assembled before considering work on a second car. This way of working results in a very poor use of the different units in the assembly line. A **pipelining** approach on the other hand is one where all assembly units are operating simultaneously, on different cars. Every time unit, a single car leaves the assembly unit.

Instructions can also be split into different subtasks. For example, a multiplication can be split into five parts which are executed by **different pieces of hardware**, as indicated in the slide. Consider the example where two vectors B and C containing floating point numbers are multiplied element by element. During the first cycle, the mantissa and exponent of B[0] and C[0] are separated and handed to the second pipeline stage. During the second cycle, the mantissas for B[0] and C[0] are being multiplied by a different piece of hardware. At the same time (during the second cycle), the first pipeline stage is again available to separate the exponent and mantissa of B[1] and C[1]. At the end of cycle 5, the first result A[0] is ready. From this point on, a final result is produced every clock cycle and all five pipeline stages are operating at the same time (i.e., parallelism). The initial 4 cycles are called a

Instruction level parallelism: pipelining

- Let **N** be the number of successive instructions
- Let **m** be the number of pipeline stages (pipeline *depth*)
- Not using pipelining takes T_{seq} = mN clock cycles
- Pipelined execution takes T_{pipe} = N + m – 1 clock cycles
- Speedup S_{pipe}:

$$S_{pipe} = Tseq/Tpipe = mN/N+m-1$$

($S_{pipe} \rightarrow$ m for N $\rightarrow \infty$)

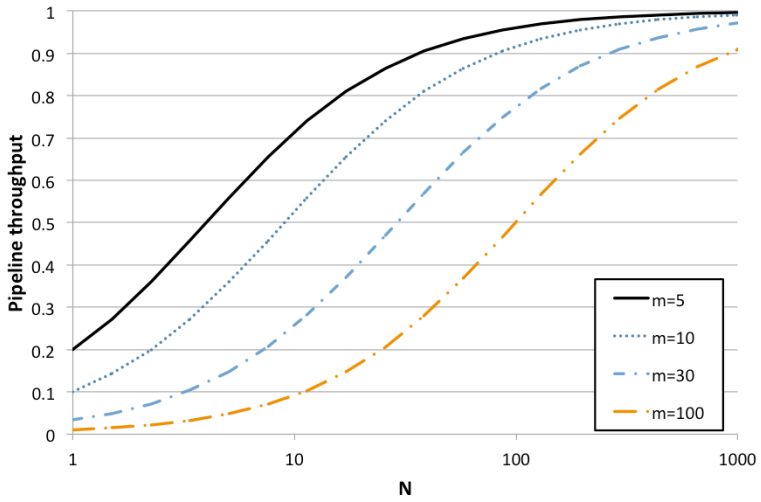
- Pipeline **throughput** = number of generated results per cycle

$$p = N/Tpipe = 1/1+m-1/N$$ (p → 1 for N $\rightarrow \infty$)

Note that **not** using a pipelining approach yields a throughput of 1/m (every m^{th} cycle, a single result is generated). Pipelining does not reduce the number of cycles to generate a single result, it merely increases the instruction throughput by keeping different parts of the CPU busy during every clock cycle.

From the speedup calculation, it might seem that the higher m, the better. This is however only true if a higher m goes hand in hand with a proportional increase in CPU clock frequency. To understand this, let us first assume that the clock frequency is kept constant. By looking at Tseq = mN, we can see that a higher m is merely slowing down the sequential model (no pipelining). Therefore, the fact that the speedup Spipe is proportional to m for large N is fully attributed to the fact that we are making the non-pipelined version slower.
In fact, the true benefit of pipelining lies in the fact that a higher m leads to simpler pipeline stages (less work to do at each stage) and that we can therefore increase the clock frequency proportionally. In that case, Tseq is constant (when expressed in wall clock time). The pipelined model is still producing a single result per cycle, but because the clock frequency is higher, more results are being produced per second. Note however that the clock cycle duration can not be shorter than the slowest component in the pipeline. Therefore, m cannot be taken excessively large. A typical value for a modern CPU is somewhere between 10 and 35 pipeline stages.

Instruction level parallelism: pipelining

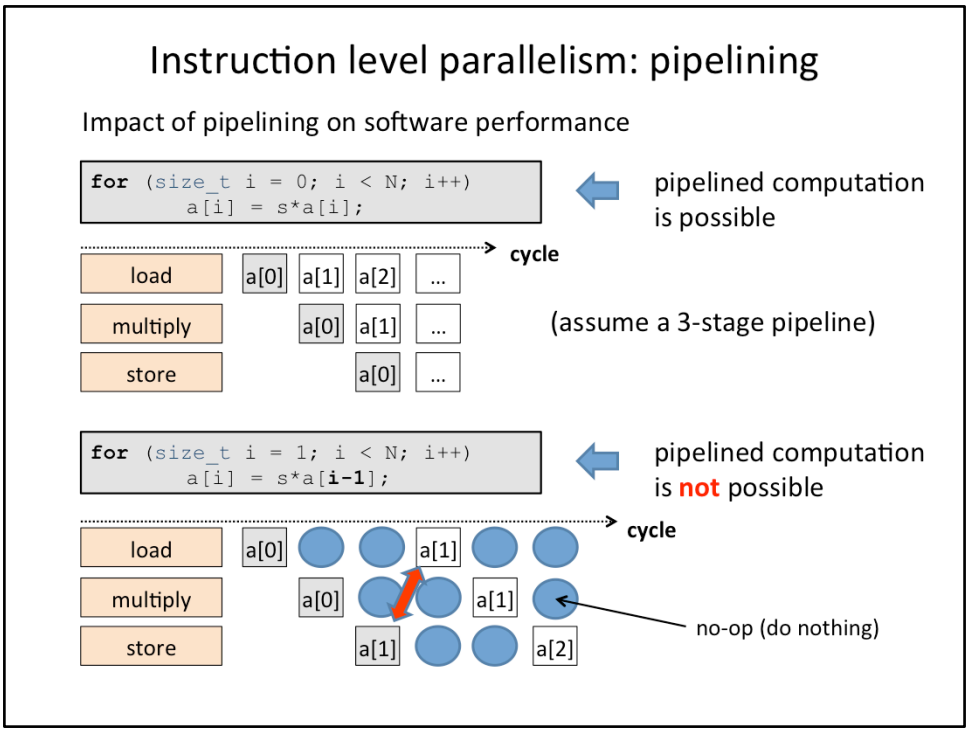


- **Throughput → 1 for N → infinite** (regardless of m)
- **Throughput = 0.5 for N = m-1**

Image reproduced from G. Hager & G. Wellein

Again, one should be careful when interpreting these results. Note that the throughput is defined as the number of results **per cycle**, and that the clock frequency is normally a function of the pipeline depth m (see notes on the previous slide).

This graph merely shows the effect of the wind-up phase on the throughput. The deeper the pipeline, the higher N should be to obtain a good throughput.



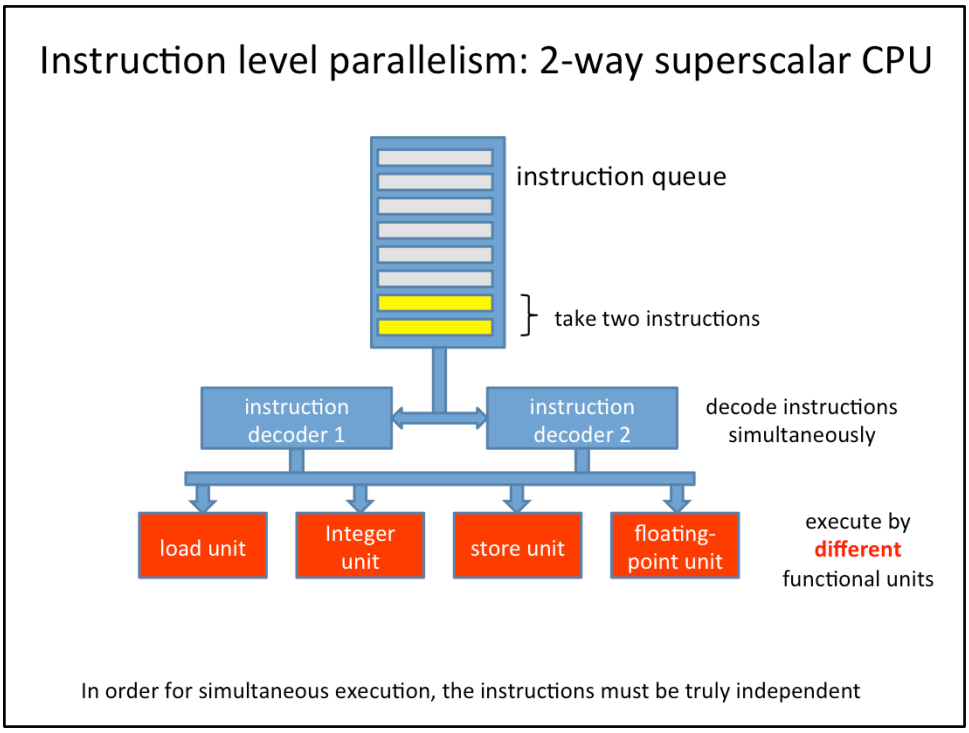
The existence of pipelining can have an enormous influence on software performance. The first example can be pipelined, yielding a throughput of one if we neglect the effect of the wind-up phase (large N).

In the second example, the computation of a[i] must be delayed until the computations of a[i-1] have fully completed. This is called a **pipeline stall or a pipeline bubble** and is caused by a so-called **loop-carried dependency**, i.e., the result of a specific loop iteration depends on the result of another iteration.

Instruction level parallelism: superscalar CPU

- At best, a pipeline produces a **single result per cycle**
- Superscalar CPUs try to process **several instructions at the same time** by duplicating certain functional execution units
  - Hardware can decode several instructions at the same time
  - Address calculations are performed by several integer units
  - Typically one or two multiply-add pipelines that perform a = b + c*d with a throughput of one each
- Cache must be fast enough to sustain more than a single load/store per cycle
- Instructions are issued from a **single instruction stream**
  - Hardware determines which instructions can be simultaneously issued at run-time
  - Need to keep track of instruction dependencies
  - Superscalarity therefore differs from multi-core technology

Superscalar processors are CPUs that can execute more **than one instruction per cycle from a single, sequential instruction stream** (i.e. a single threaded process – see further for these exact definitions). To achieve this, multiple instructions are decoded at the same time and dispatched to redundant execution units. Because there is only a single instruction stream, superscalarity should not be confused with multi-core technology. As a matter of fact, superscalar CPUs are classified as SISD by Flynn's taxonomy (see further), whereas a multicore CPU is considered to be an MIMD processor.



This slides depicts a **2-way superscalar CPU**. From the instruction queue, two instructions are fetched and decoded simultaneously. They are executed by **different and/or redundant functional units**. The instructions should be truly independent, i.e., one instruction should not rely on the outcome of the execution of the other. Furthermore, they should not access the same functional units. In the very best case, a 2-way superscalar CPU is twice as fast. However, in practice, it is very difficult to achieve a reasonable speedup from a superscalar design. This is because instructions are typically not independent, and because of the von Neumann bottleneck.

Usually, the instructions from the instruction queue are reordered, to be able to select instructions that can truly overlap (**out-of-order execution**). Of course, out-of-order execution should not alter the results of the program, and can therefore only be applied to a limited extent.

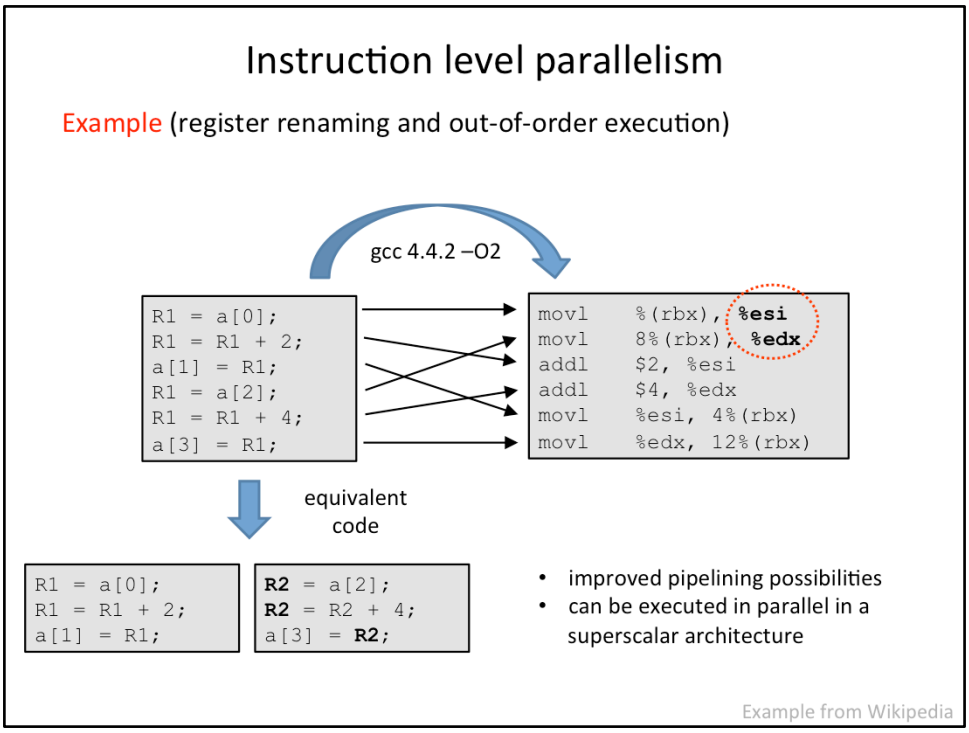
Instruction level parallelism

Other techniques used to improve instruction per cycle throughput

- **Out-of-order execution**: change the order in which instructions are executed, to improve pipelining behavior or superscalar execution.
- **Register renaming**: improve parallelism by issuing an instruction using different registers then specified in the code.
- **Branch prediction**: attempt to guess whether a specific branch will be taken in a conditional jump (if-then-else).
- **Speculative execution**: execution of a portion of the code for which it is yet uncertain that it needs to be executed (after branch prediction).

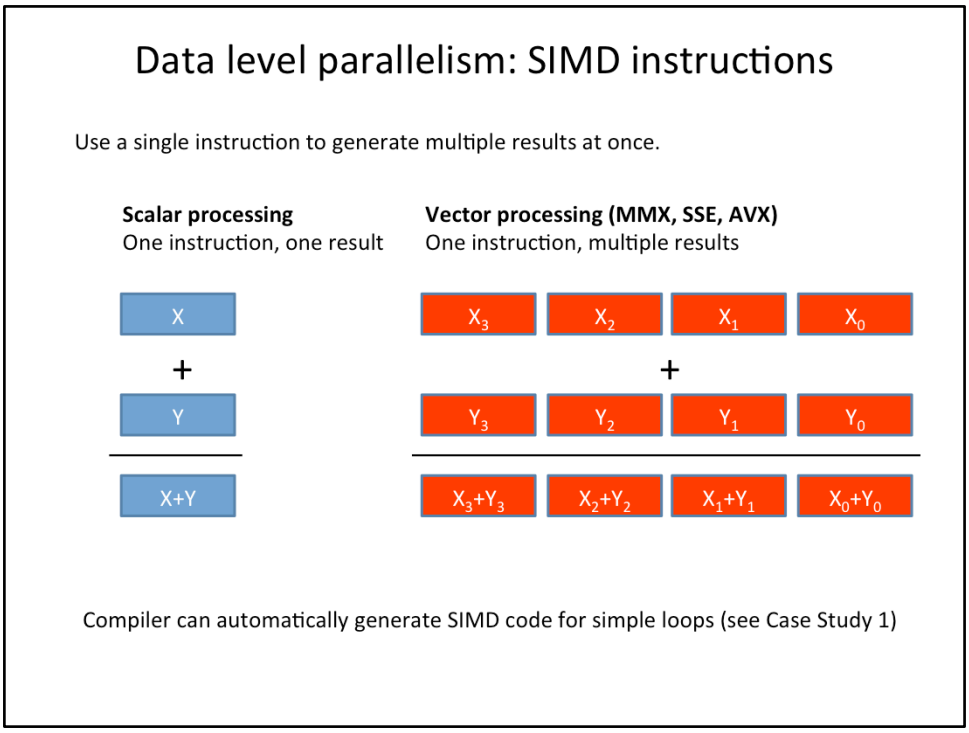
These are techniques that go hand in hand with pipelining and superscalar design, in order to improve the instruction throughput.

Out-of-order execution: see previous slide.
Register renaming: execute instructions using different registers than is specified by the assembly code. For example, two consecutive instructions using the same register(s) cannot be pipelined or executed simultaneously. Simply using another available register for one of the instructions will prevent the instructions from being serialized (see next slide for an example).
Branch prediction: a conditional branch (if – then – else construction) determines the order in which instructions are executed. Therefore, pipelined computations must be stalled until the condition is evaluated. In a tight for-loop, this might be detrimental for performance. Branch prediction is a hardware solution in which the hardware will try to guess the outcome of a specific condition, before it is fully evaluated.
Speculative execution: Speculative execution goes hand in hand with branch prediction. Speculative execution of the execution of a number of instructions for which it is not fully certain that the result will be needed and/or correct. Speculative execution and branch prediction go hand in hand. Also, hardware-issued prefetching can be seen as a form of speculative execution. The hardware simply ignores results that were incorrectly generated through speculative execution.



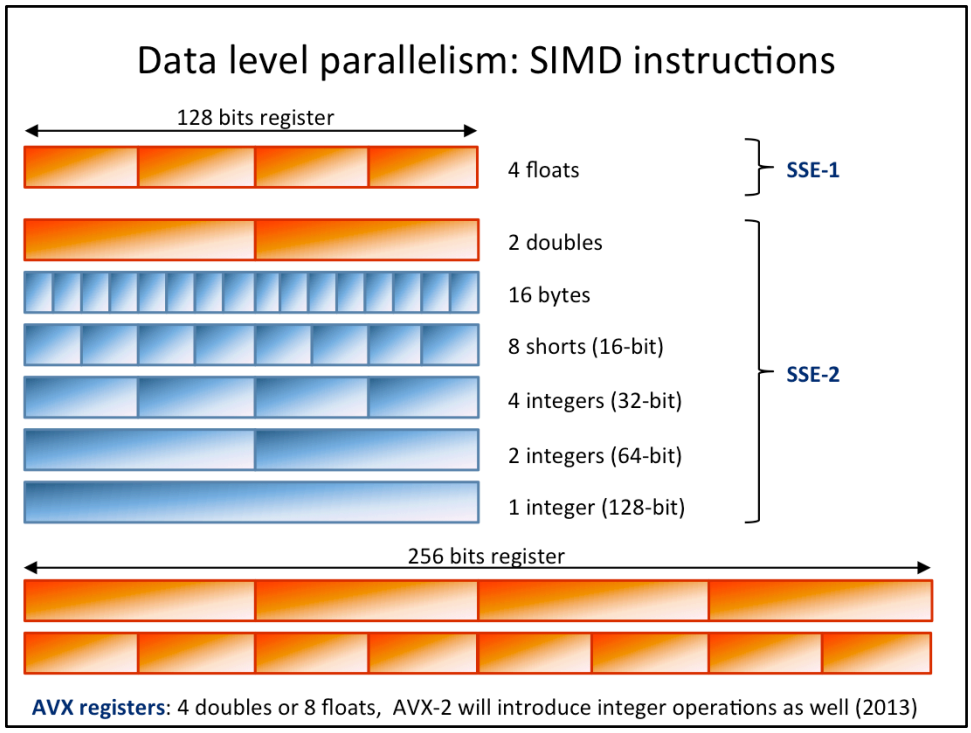
This simple example shows both the concept of register renaming and out-of-order execution. In this case, these optimizations were performed by the compiler itself.

In the source code, the temporary variable R1 is used twice to store intermediate results. In case two temporary variables would be used, the two blocks of code could be executed simultaneously. The compiler effectively generates assembly code that uses different registers (esi and edx register). Also, the order in which the operations are executed is changed. By putting the load instructions as early as possible, the latency from the second load can overlap with the computation of the first value.

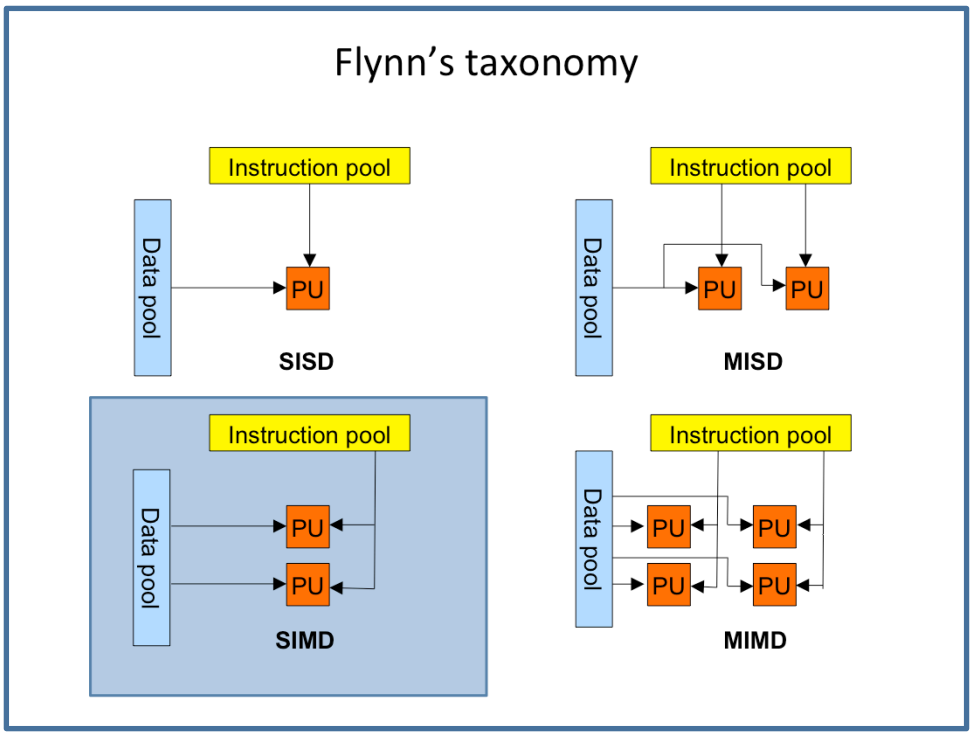


SIMD instructions (Single Instruction, Multiple Data) are currently supported by modern CPUs. SIMD instructions can perform operations like additions or multiplications on several data elements in a single instruction. Historically, the MMX or MultiMedia eXtension instruction set could operate on 64 bit registers, effectively capable of performing two operations on two 32-bit integers in a single instruction. This was later extended by the SSE (Streaming SIMD eXtensions) instructions, featuring 128-bit wide registers (called xmm) that could operate on four single-precision floating point elements. This was later expanded in SSE2 to also accommodate two double precision numbers (in the same xmm registers) and even integers (see also next slide). SSE3 and SSE4 add support for even more advanced instructions.

Very recent architectures like Intel's Sandy Bridge or AMD's Bulldozer architectures support the Advanced Vector Extensions (AVX). The existing xmm registers have been widened to 256 bit and are now called ymm registers, capable of accommodating four double-precision or eight single-precision floating point numbers. Note that the existing SSE instructions are still valid, operating on the lower 128 bit portion of the registers.



This slides depicts several combinations of how the SSE and AVX registers can be used.



In Flynn’s taxonomy, SSE/AVX instructions belong to the SIMD category.

Chapter 1: outline

- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK

Vector triad source code

```
void vectorTriad(double * __restrict a,
                 const double * __restrict b,
                 const double * __restrict c,
                 const double * __restrict d,
                 const size_t N)
{
        for (size_t i = 0; i < N; i++)
                a[i] = b[i] + c[i] * d[i];
}
```

```
...
for (size_t i = 0; i < nRepeats; i++)
        vectorTriad(a, b, c, d, N);
...
```

restrict keyword (C99 standard) denotes that memory referred to by a, b, c and d is disjoint (= non-overlapping)

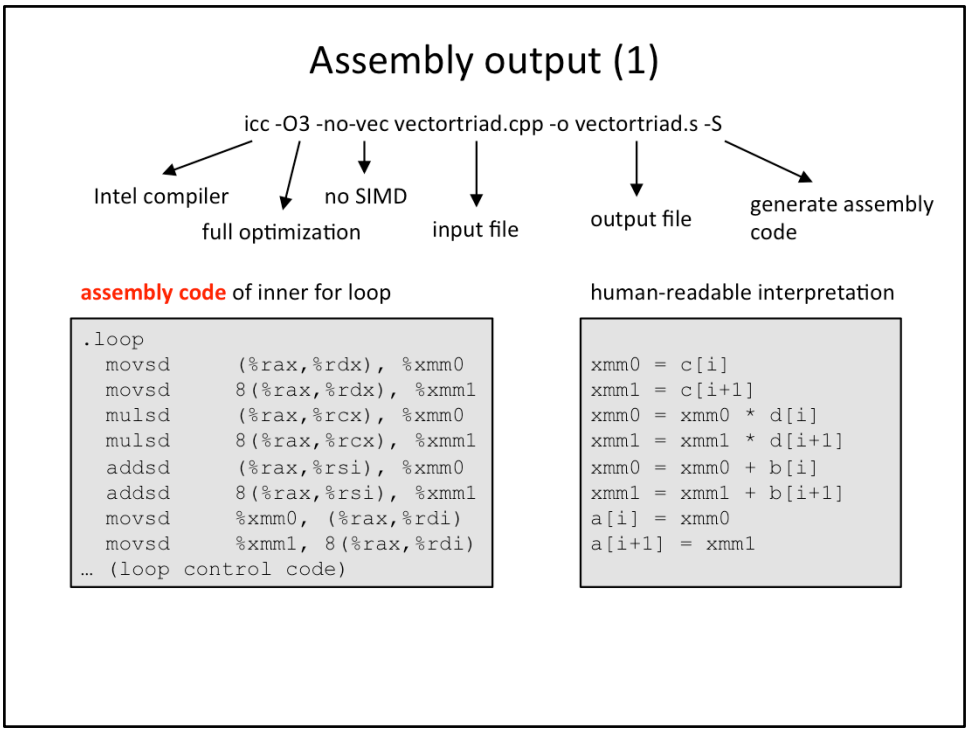
Per inner loop iteration:

- **2 flops** (1 addition, 1 multiplication)
- **3 load** operations (b, c, d) and **1 store** operation (a)

The so-called vector triad example contains an inner loop operating on four vectors a, b, c and d. Every iteration, three elements are loaded (b[i], c[i], d[i]), a single result is written (a[i]) and two FLOPS are performed (one multiplication, one addition).

The vectorTriad code is executed nRepeat times in the outer loop. This allows us to see the effects of caching. For very small N, the a, b, c and d vectors will be present in the cache after the first call to the vectorTriad routine (i.e., the first outer loop iteration). Every subsequent outer loop iteration, the vectorTriad code will be completely streamed from cache. Conversely, for very large N, this data is streamed from main memory.

The restrict keyword denotes that the memory blocks referred to by a, b, c and d are non-overlapping. Therefore, the compiler knows that writing to vector a does not change the data in vectors b, c or d. This allows the compiler to fully exploit pipelining possiblities and SIMD instructions.



For the vector triad example, we use the Intel compiler. The same can be achieved with the GCC compiler, but the assembly output of the Intel compiler is somewhat cleaner and therefore easier to understand.

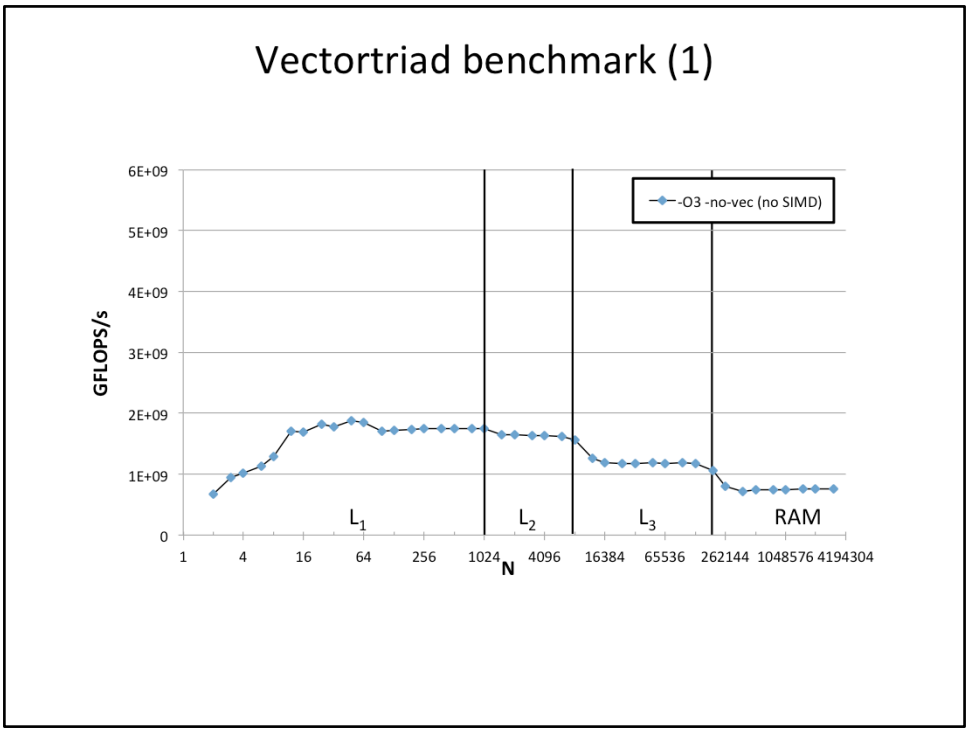
First, we do not allow the compiler to use vectorization (SIMD instructions). This is prohibited by the "-no-vec" flag.

We now look at the generated assembly instructions:

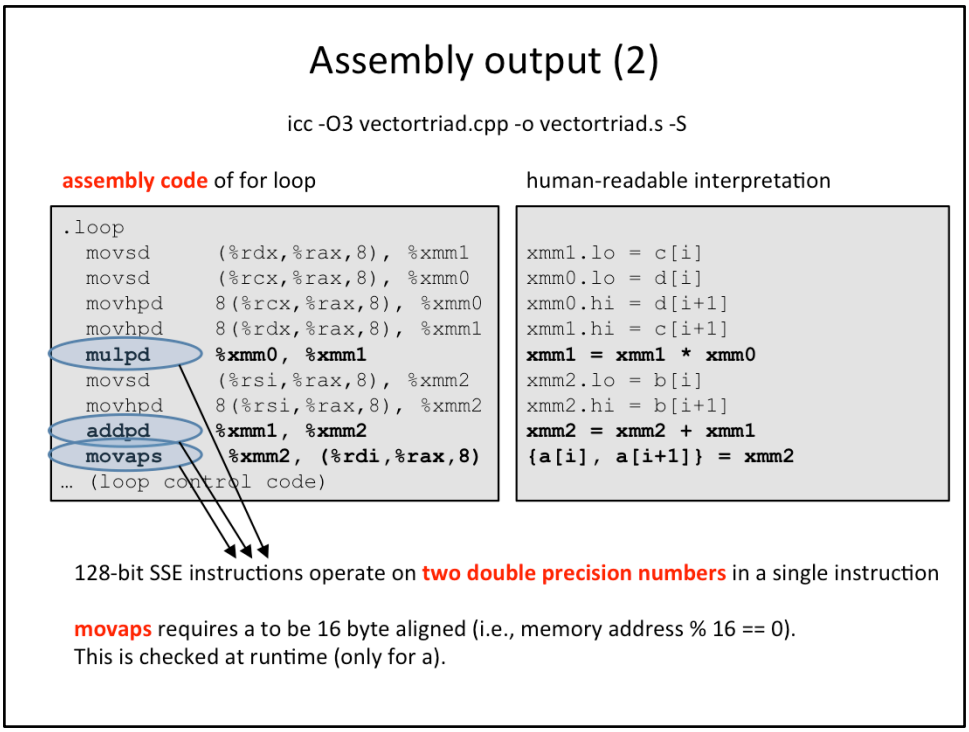
- movsd (move scalar double): move data from operand 1 to operand 2
- mulsd (multiply scalar double): multiply data in operand 1 with data from operand 2, store result in operand 2.
- addsd (add scalar double): add data in operand 1 to data from operand 2, store result in operand 2.

These instructions operate on a single double-precision number. The xmm registers are 128 bits wide, however, only the least significant 64 bits are used. Operands can refer to registers (e.g. xmm) or to a memory location. The memory address specification should be interpreted as follows:

```
displacement(base, index, scale)
```



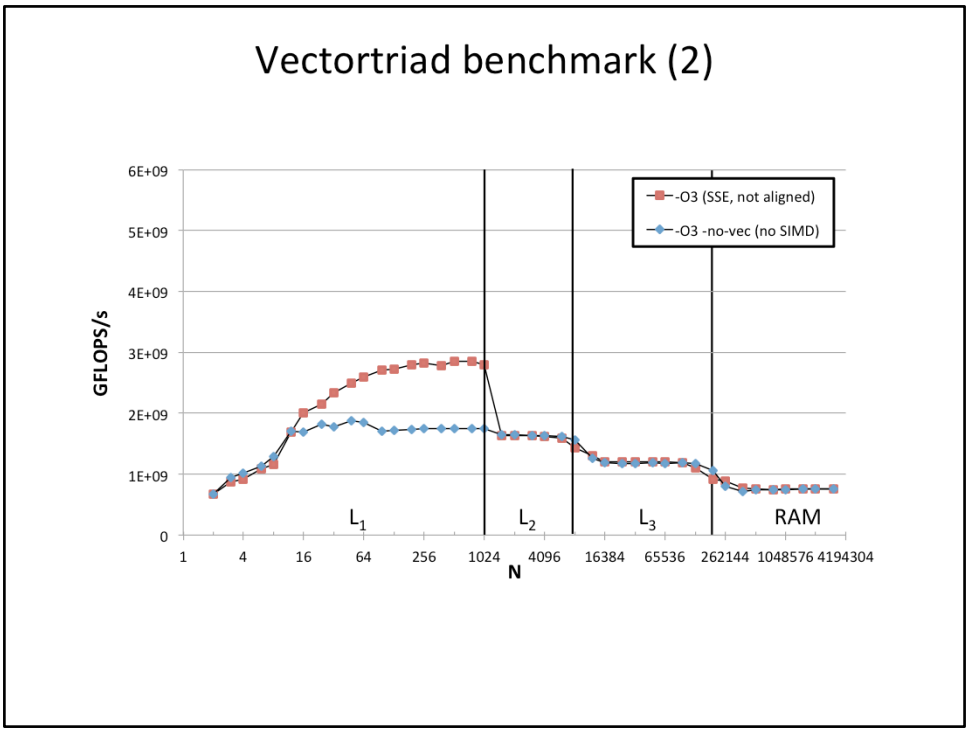
By benchmarking the FLOPS/s for varying N, we can observe the different cache levels. For very small N, there is significant loop overhead and hence poor performance. The vertical lines denote the value of N for which the four vectors can be kept in a specific cache level. Note the distinct performance characteristics for the different size of N. Streaming data from RAM is clearly slower than streaming data from cache. However, the difference is relatively modest. In fact, because we do not yet use SIMD instructions, the code is CPU bound rather when streaming from the L1 cache. This will be made clear in the next slides.



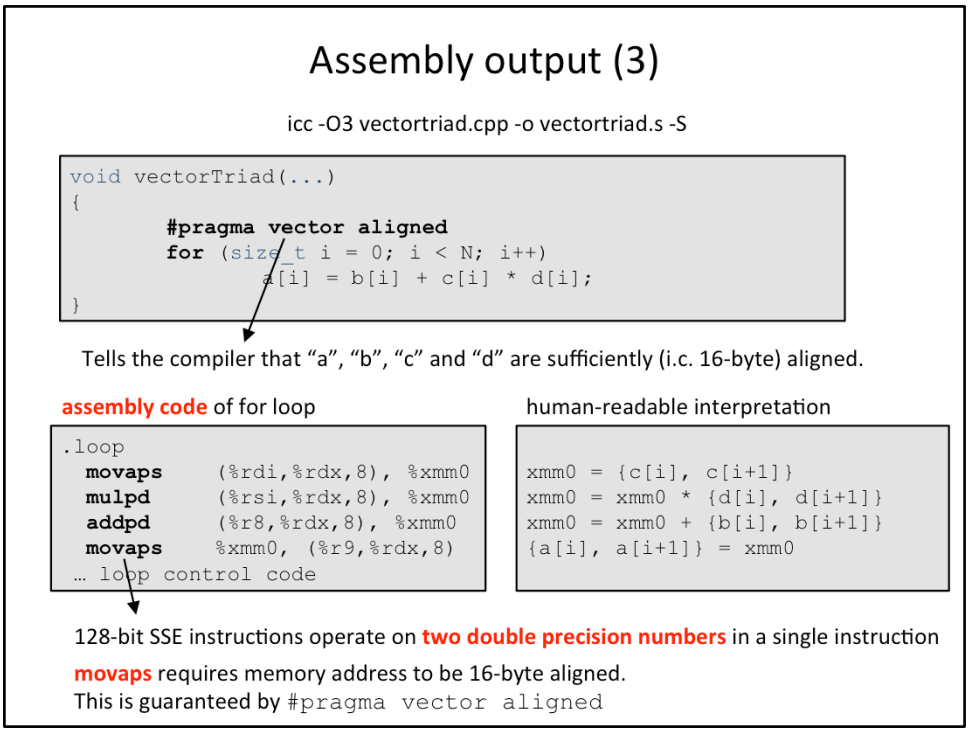
We now instruct the compiler to use SSE instructions (simply leave out the –no-vec directive and use –O3).

The compiler now loads two doubles in each of the xmm registers by using the **movsd** (move a double to the lower halve of the xmm register) and **movhpd** (move high packed double: move a double to the upper halve of the xmm register) instructions. The **mulpd** and **addpd** (multiply/add packed double) are SIMD instructions which effectively operate on the full 128-bit xmm registers in a single instruction, hence doubling the throughput. The **movapd** (move aligned packed double) moves two doubles in a single instruction. In case one of the operands is a memory address (as is the case here), the address specified needs to be 16-byte aligned. Curiously, the compiler opts to use the **movaps** (move aligned packed single-precision) instruction. This moves four floats (again 128-bit) but yields the same result.

We did not explicitly instruct the compiler to use aligned move operations. However, the compiler generated code to check the alignment of vector “a” at runtime. If “a” is aligned, then the code shown above is executed. If “a” is not aligned, then different code (not shown) is executed. The only difference is that movaps is then replaced by two instructions: movsd and movhpd. The compiler could also check the alignment of b, c and d. However, that would give rise to 2^4 = 16 different



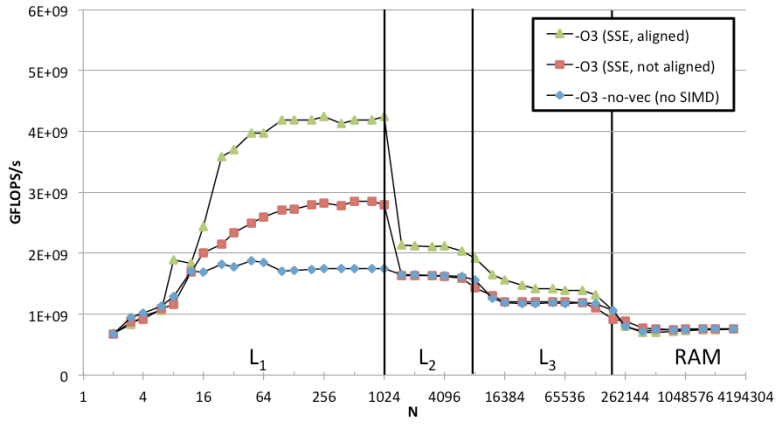
Note that using SIMD instructions almost doubles performance when streaming from the L1 cache. When streaming from the L2, L3 or RAM, the code remains memory bandwidth bound, and is therefore not accelerated by using SIMD instructions.



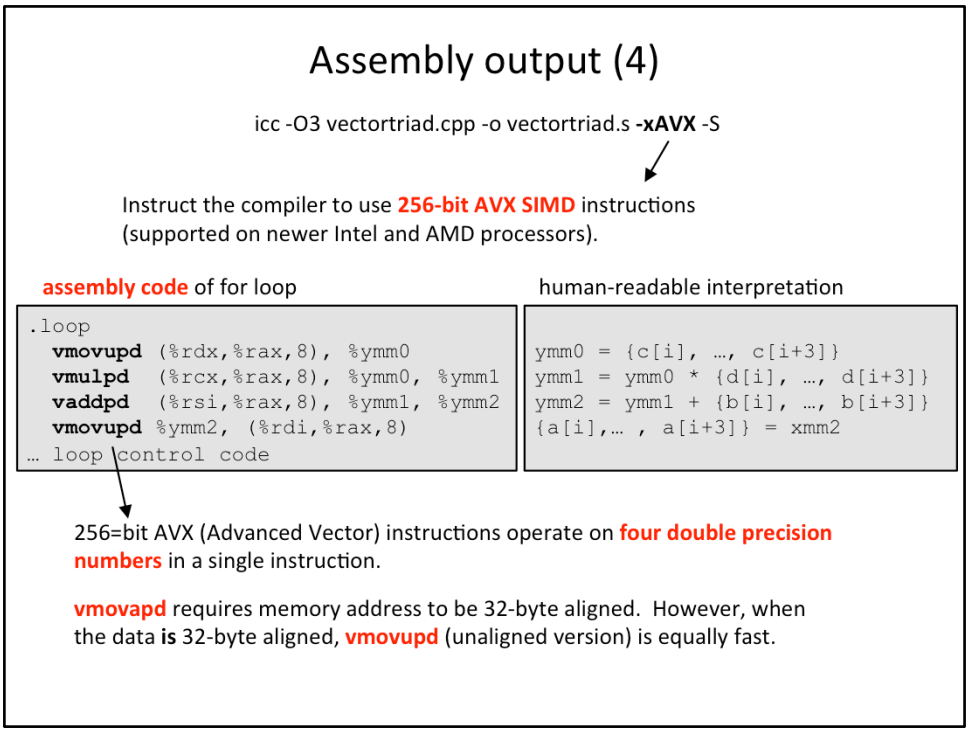
We now instruct the compiler to use aligned loads and stores. For the Intel compiler, this is achieved by putting a #pragma vector aligned compiler directive above the inner loop. The compiler is then informed that a, b, c and d are sufficiently aligned for use of the SIMD move instructions (16-byte alignment in this case). This results in the very simple assembly code…

Note that the compiler uses 4-way loop unrolling (not shown in this slide).

Vectortriad benchmark (3)

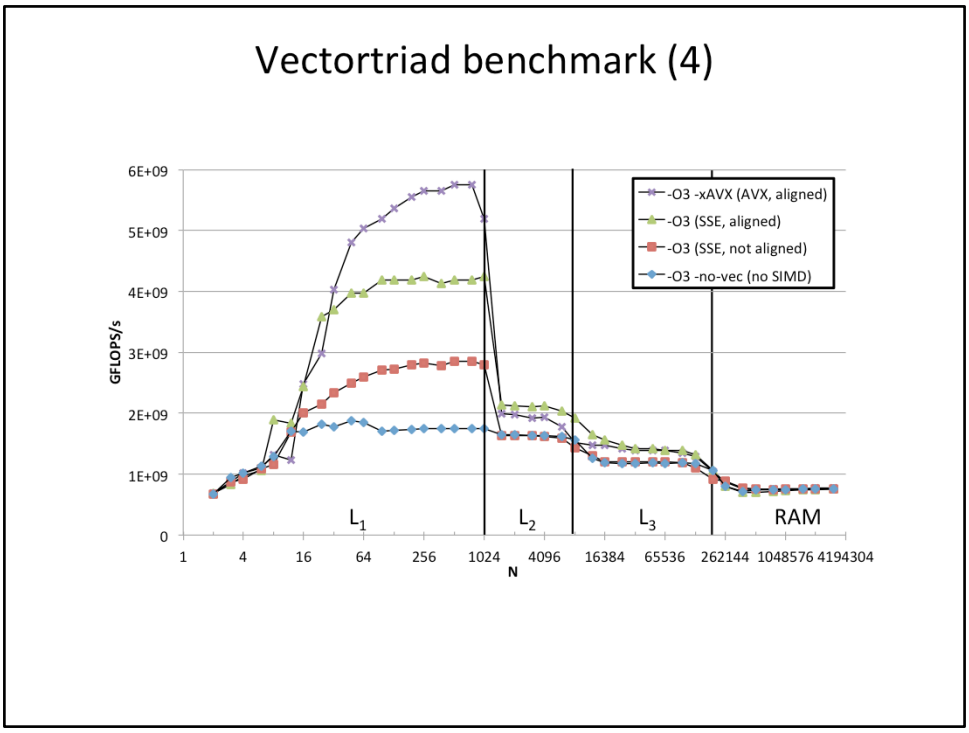


… again improving our vector triad performance.



Finally, we instruct our compiler to use AVX SIMD instructions. This uses the 256-bit ymm registers and requires the data to be 32-byte aligned. SIMD instructions operate on four double-precision numbers in a single instruction.

Note that the compiler uses 4-way loop unrolling (not shown in this slide). Hence, per loop iteration, 16 results are generated. A minimum of 12 cycles per loop iterations are required on a Sandy Bridge architecture. Therefore, the maximum FLOPS/s can be expressed as frequency (Hz) x 16 x 2 / 12. Using a 2.2 GHz CPU, this yields 5.86 GFLOPS/s.



Yielding again improved performance when streaming from the L1 cache.

Note that the measured maximum FLOPS/s for the vector triad is very close to the theoretical maximum of 5.86 GFLOPS/s for N ~ 512. However, this is still far from the CPU's peak FLOPS/s capabilities of 17.6 GFLOPS/s (per core !). This is because the ratio of load/stores to flops is simply too high for the vector triad example. You can find code examples on the internet, that will push the CPU to its limits, by operating on registers only. Most scientific codes operate at only a fraction of the CPU's peak performance, especially when using compiler-generated code.

Aligning memory

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment,
                   size_t size);
```

On exit, `memptr` points to allocated buffer
- `alignment`: alignment boundary, e.g. 8, 16 or 32
- `size`: size of buffer (in bytes)

Especially useful for SIMD instructions
- SSE: requires **16-byte** aligned memory
- AVX: requires **32-byte** aligned memory

In principle, when allocating memory in C/C++ using malloc or new, the pointer to the first element of the allocated data can contain any memory offset (expressed in bytes). On modern CPUs, it is often beneficial for performance if that this offset is a multiple of 8, 16 or even 32 (see the vector triad example). This is called memory alignment. To create memory that is aligned on e.g. a 16-byte boundary, one can use the Posix function in the slide. It is however also simple to do this by hand. If a 16-byte aligned memory buffer is needed, allocate a buffer that is 15 bytes larger than desired, and create a new pointer to the first element for which the address is a multiple of 16.

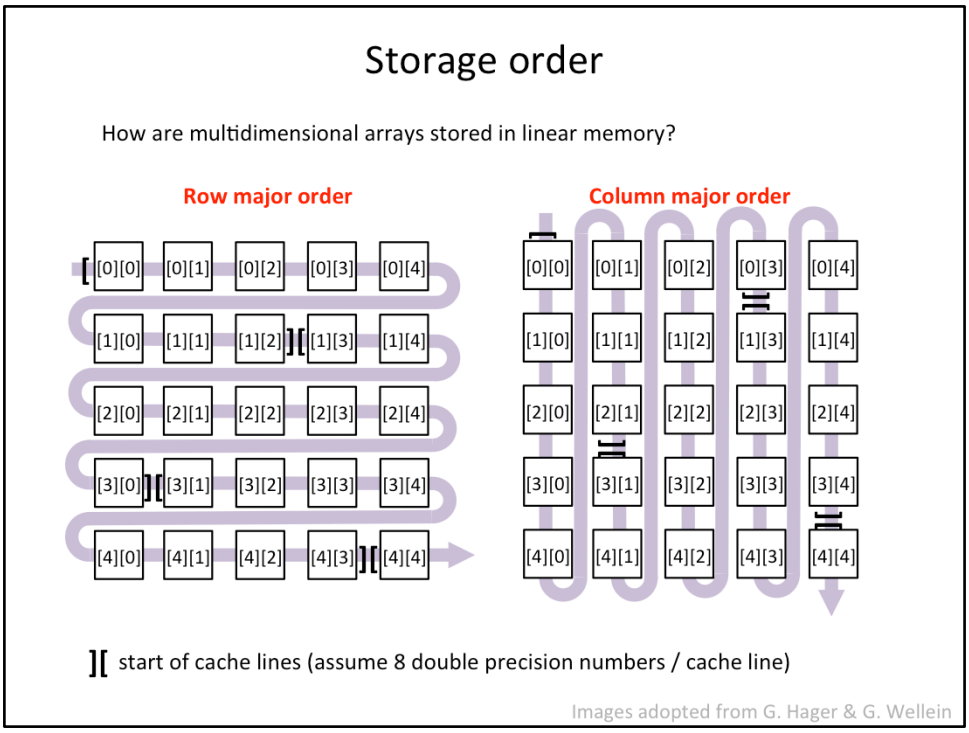
Compilers usually already align data on 4 or 8-byte boundaries for performance. One should especially pay attention to the alignment boundaries when SIMD instructions are needed.

Chapter 1: outline

- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK

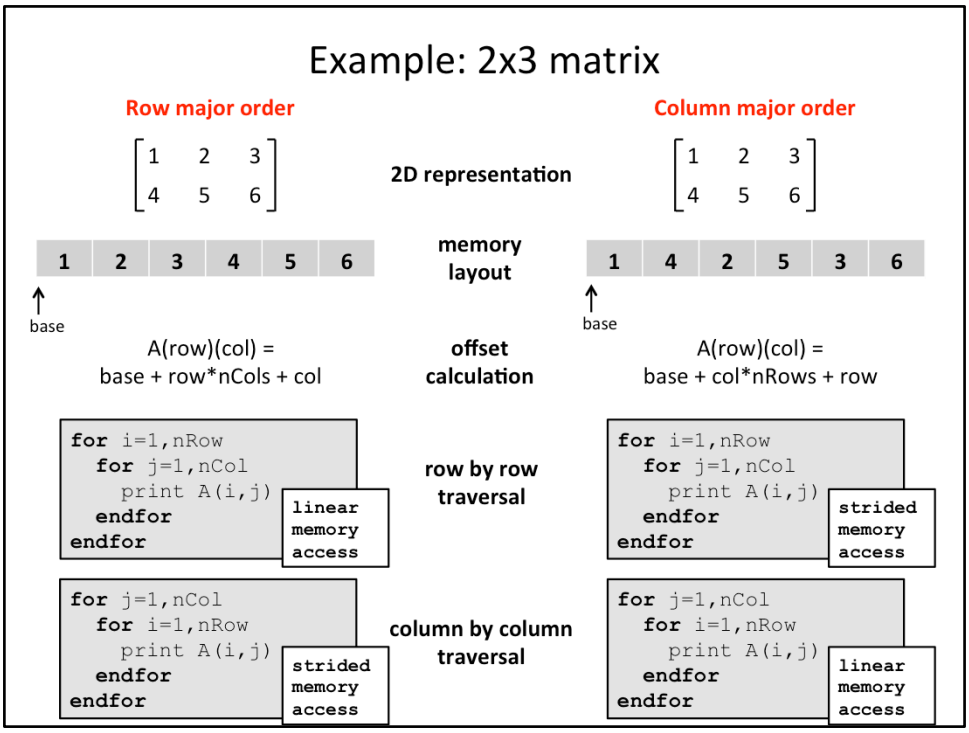
In case study one, we dealt with streaming data access and we investigated the influence of caching and SIMD instructions. All access was linear.

We will now consider a matrix-vector multiplication, where we focus on how we can store two dimensional data in memory, and how this choice affects performance.



Multidimensional arrays need to be stored onto memory that is addressed in a linear fashion (using a single address). Therefore, in the case of a matrix, we need to map the two-dimensional matrix onto the one-dimensional memory. There are two natural possibilities to achieve this: row major ordering, in which consecutive elements on the same row are stored consecutively in memory, or the column major format, in which consecutive elements on the same column are stored consecutively in memory.

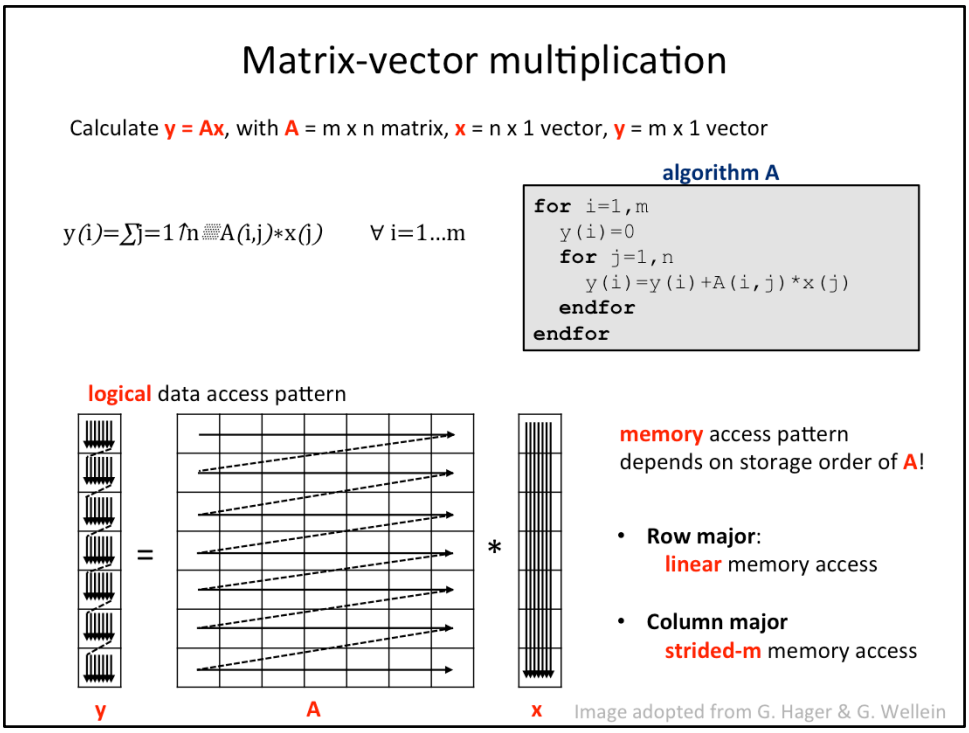
The brackets depict the cache lines. When using a row major storage order, accessing element [0][0] will load all elements [0][0], [0][1], … [1][2] in cache (we assume that the first element starts on a cache line, this is generally not true). Conversely, using a column major order, accessing the same element [0][0] will load elements [0][0], [1][0], …, [2][1] in cache. Therefore, we should carefully construct any algorithms operating on matrices, in order to take maximum advantage of spatial locality.



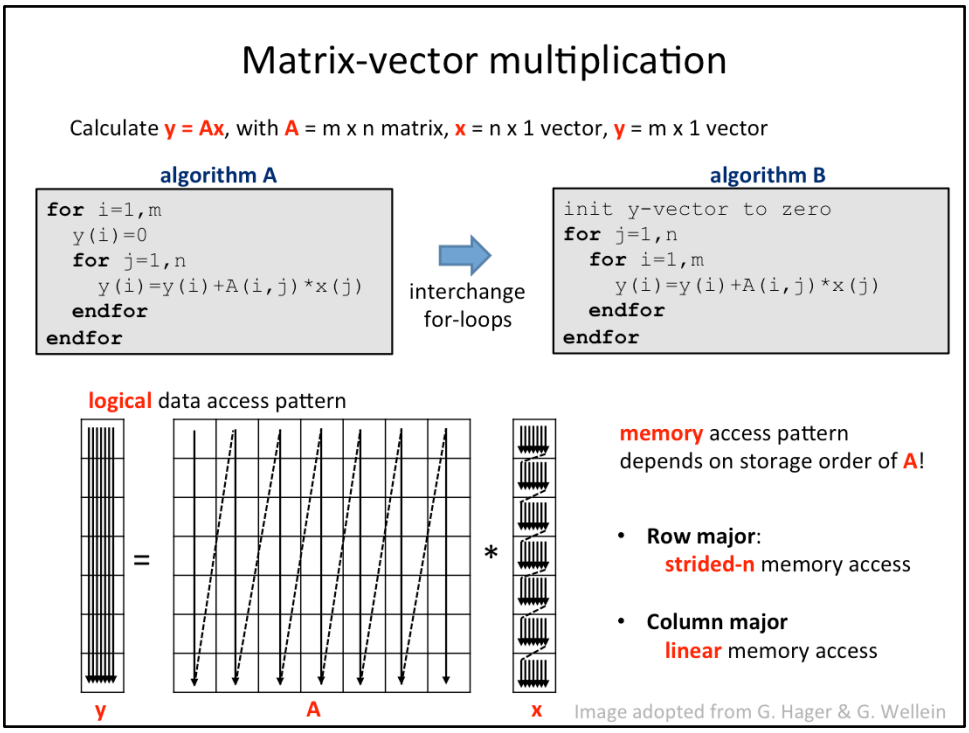
Left column : row major order
Right column: column major order

Note that we are still dealing with the same matrix, in a mathematical sense. The only thing that differs is the order in which the elements are stored in memory. This of course affects how we should calculate the offset to a certain element A(row)(col).

A row by row traversal of a matrix stored in column major format leads to **strided memory access,** the same is true for a column by column traversal when using the row major format.



We consider a matrix-vector multiplication. Algorithm A contains pseudocode for what is arguably the most natural way to implement this. It corresponds to a row by row traversal of A (every element of A is accessed exactly once during the course of the algorithm). Clearly, it is beneficial to use a row-major storage scheme for matrix A, when using algorithm A. Vector x is loaded m times in a streaming fashion. Each element of vector y is accessed to n times in the inner loop (in fact, the summation can be done in a register, and written to y[i]).

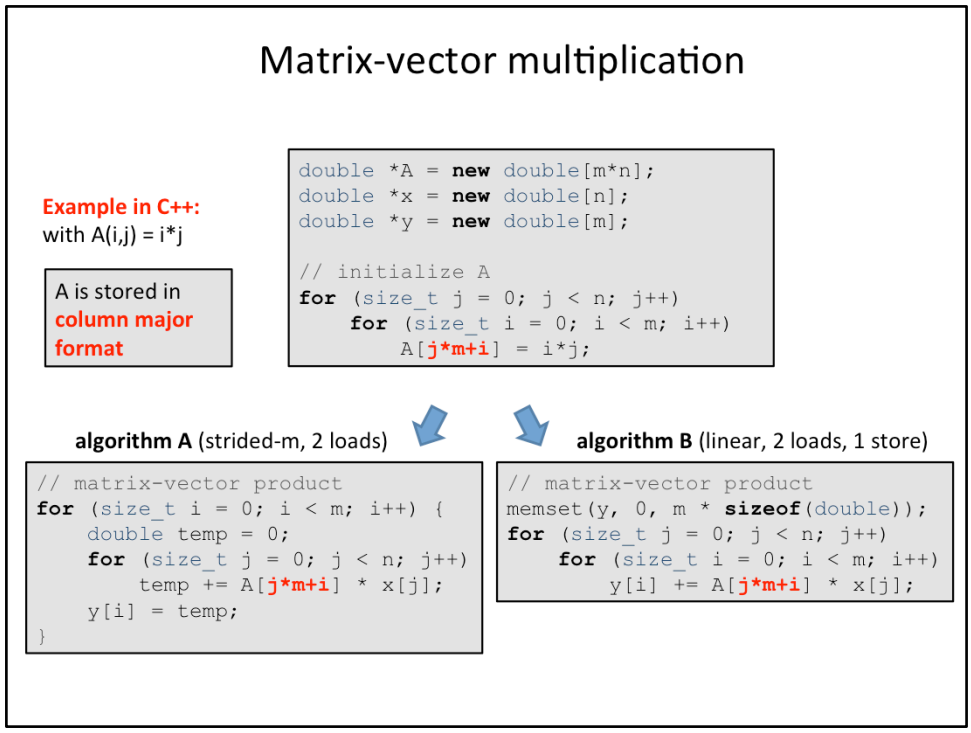


A second, equally valid algorithm for the matrix-vector product can be obtained by exchanging the two for-loops, giving rise to a column by column traversal of matrix A (algorithm B). When using algorithm B, it is beneficial for performance to use a column-major storage for matrix A, as this will lead to linear access. Also, the access patterns of y and x are interchanged. The attentive reader may notice that it is no longer possible to accumulate the intermediate values of y[i] in the inner loop in a register.

Matrix-vector multiplication

Summary

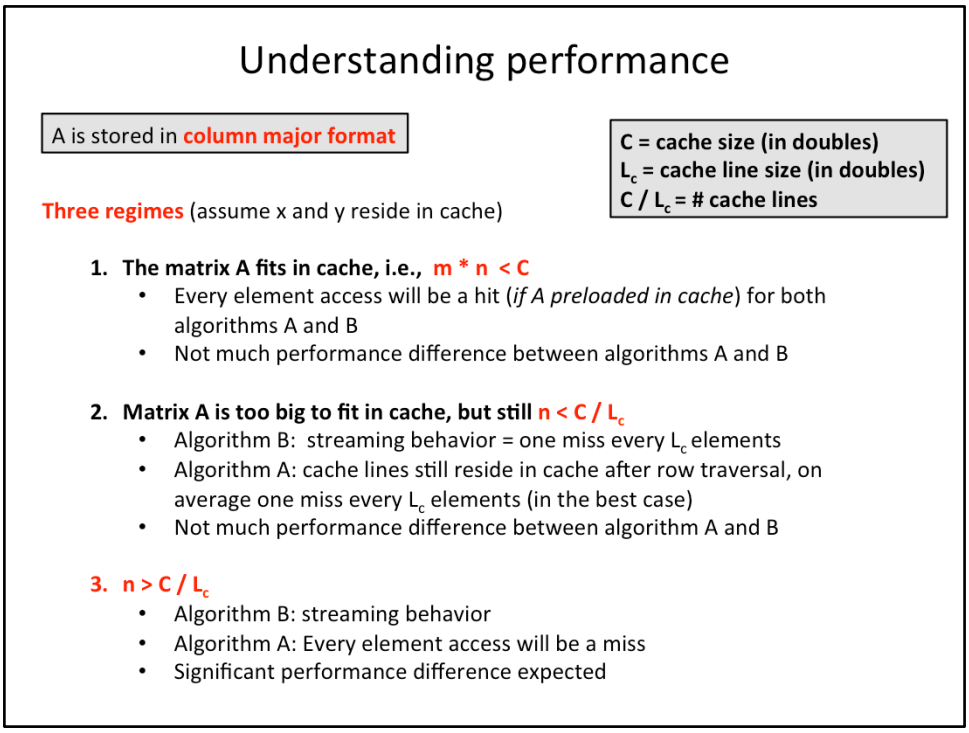
	matrix storage order	
	row-major	**column-major**
algorithm A (row by row)	**linear**	**strided-m**
algorithm B (column by column)	**strided-n**	**linear**



The C++ example shows the matrix initialization code, with A(i, j) = i*j where A is stored in column-major format.

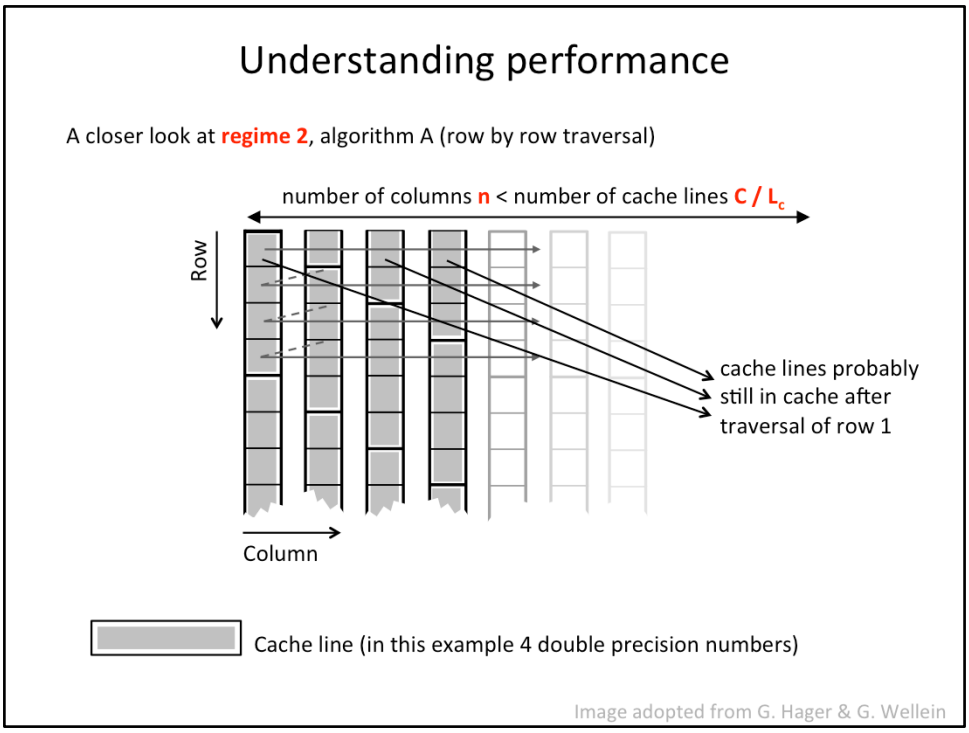
The lower-left box shows the implementation for algorithm A (row by row traversal). This gives rise to a strided-m memory access pattern for A in the inner loop. The inner loop consists of two loads and two FLOPS. Note that the "temp" variable can be kept in register during the execution of the inner loop and that only m writes (to y) are performed in the outer loop. This is an advantage of algorithm A.

The lower-right box shows the implementation for algorithm B (column by column traversal), giving rise to a linear memory access pattern for A. The inner loops consists of two loads, two FLOPS and one store.



We take a closer look at the expected performance for both algorithms, when the matrix A is stored in **column major format** (strided memory access).

- **Regime 1**: The matrix A is small enough to completely fit in cache. If matrix A is already in cache prior to the matrix-vector product execution (because of e.g. a previous operation on matrix A), every element access is a cache hit for both algorithm A and B. In case matrix A is yet not in cache, every cache miss will result in the retrieval of a complete cache line. This cache line contains other elements of matrix A as well, which will lead to future cache hits. In case of a typical cache size of 64 bytes (8 doubles), this means that a every cache miss is followed by future 7 cache hits on average. A cache miss will generally not lead to the eviction of a cache line containing elements of matrix A, in case the least recently used (LRU) eviction strategy is used. This discussion is true for both algorithms A and B, i.e. independent of how the elements are accessed.
- **Regime 2:** The matrix is too large to completely fit in cache, however, the number of columns is smaller than the number of cache lines.
    - Algorithm B results in streaming memory access for matrix A. In case of a cache miss, a complete cache line will be retrieved from memory. The next few memory accesses are located in the same cache line and will therefore a hit. In case of a typical cache size of 64 bytes (8 doubles), this means that a cache miss is followed by 7 cache hits.



A closer look at regime 2. During the first row traversal, the cache is sufficiently large to contain all cache lines that contain the elements on the first row of matrix A. These cache lines also contain a lot of elements on the subsequent rows, leading to a behavior that is comparable to linear memory access, and hence, a similar performance.

Understanding performance: benchmark

g++ -O2 matvec.cpp -o matvec

no SIMD, we want to focus on **memory access pattern**

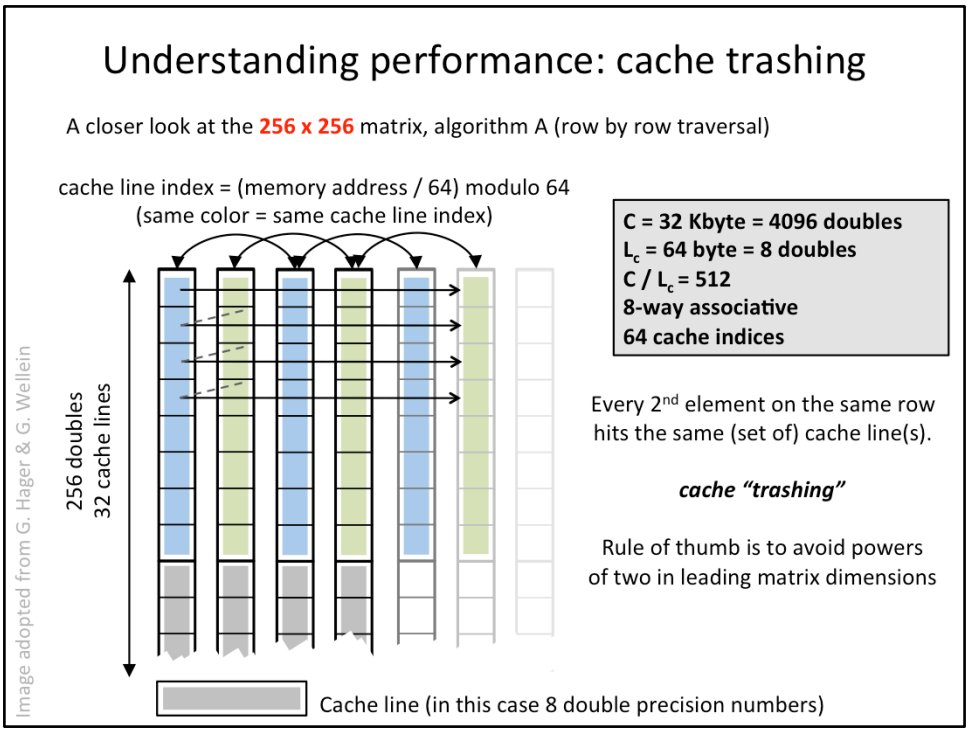
C = 32 Kbyte = 4096 doubles
L_c = 64 byte = 8 doubles
C / L_c = 512

		algorithm A (strided-m)		algorithm B (linear)	
matrix dimensions	**regime**	**FLOPS/s**	**L_1 miss rate**	**FLOPS/s**	**L_1 miss rate**
50x50	1	1.66E9	0.0%	1.51E9	0.0%
100x100	2	1.53E9	6.1%	1.42E9	4.1%
1000x1000	3	9.03E8	55.2%	1.30E9	4.1%
252x252	**2**	1.41E9	6.2%	1.41E9	4.1%
256x256	**2**	9.17E8	50%	1.45E9	4.1%

?

We can use valgrind to simulate the cache behavior. We only focus on the L1 cache hit ratio. The L1 cache in this example is a 32 Kbyte cache (can hold 4096 doubles) and a cache line contains 8 doubles.

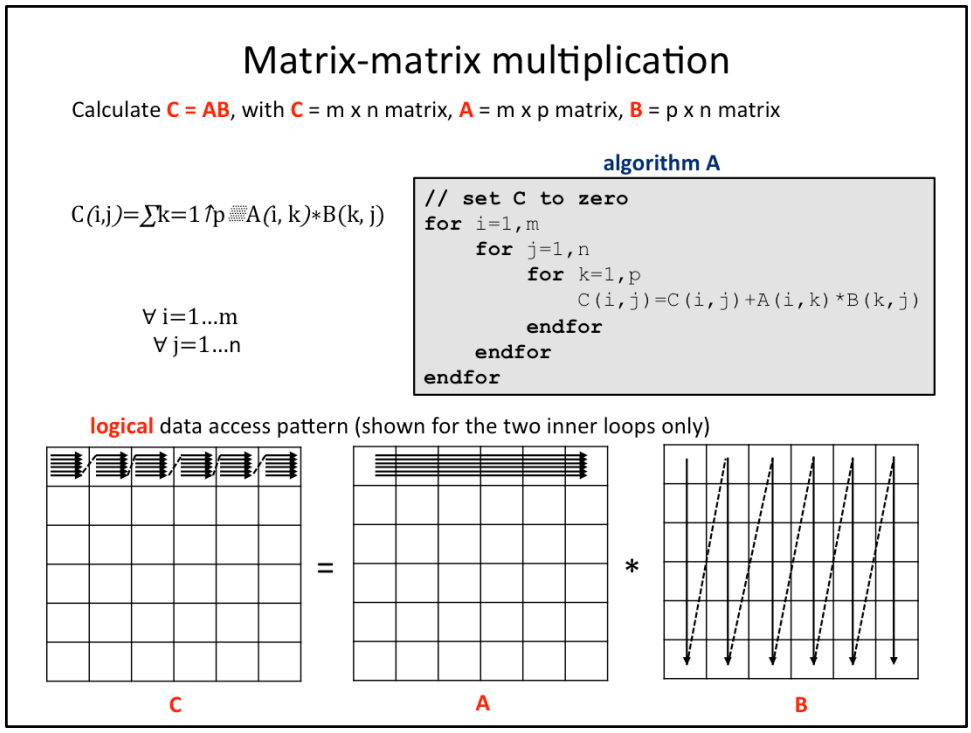
- **Regime 1**: Both algorithms A and B have zero cache misses (matrix A was preloaded in the cache).
- **Regime 2:** Both algorithms A and B have similar cache miss ratio. If we look at matrix A only, we would expect a miss ratio of 12.5% (for every miss, there are 7 cache hits). However, the percentages also contain the read accesses to vector x. Vector x is accessed many times and is therefore kept in cache at all times, resulting in a lower hit ratio. Explaining the precise percentages is a complicated matter, as this depends on how the cache lines occupied by x (and also vector y for writing !) conflict with the ones from matrix A. The fact that algorithm A leads to a higher FLOPS/s despite it higher L1 miss ratio is because the algorithm was implemented keeping writes to y[i] in register (using the "temp" variable) during the inner loop.
- **Regime 3:** For algorithm B, there is no difference in miss ratio, compared to regime 2. Algorithm A however, has a drastically higher L1 miss ratio and hence a reduced performance. Every access to element A is a miss, whereas every access to element x is a hit, yielding a percentage around 50%.



The problem is that every second element on the same row is mapped to the same **cache index**, because of the specific dimensions of the matrix. Because the cache is 8-way set associative, the cache is capable of storing the cache lines that contain the first 16 elements on the same row. However, accessing the 17th element will lead to the eviction of the cache line that contains the first element. Therefore, a large portion of the cache is effectively not used. This phenomenon is called **cache trashing**. Cache trashing can occur when strided memory access leads to the rapid loading and eviction of cache lines. It is a consequence of the limited associativity of a cache. As a general rule of thumb, one should avoid strided memory access with an offset that is a power of 2.

Chapter 1: outline

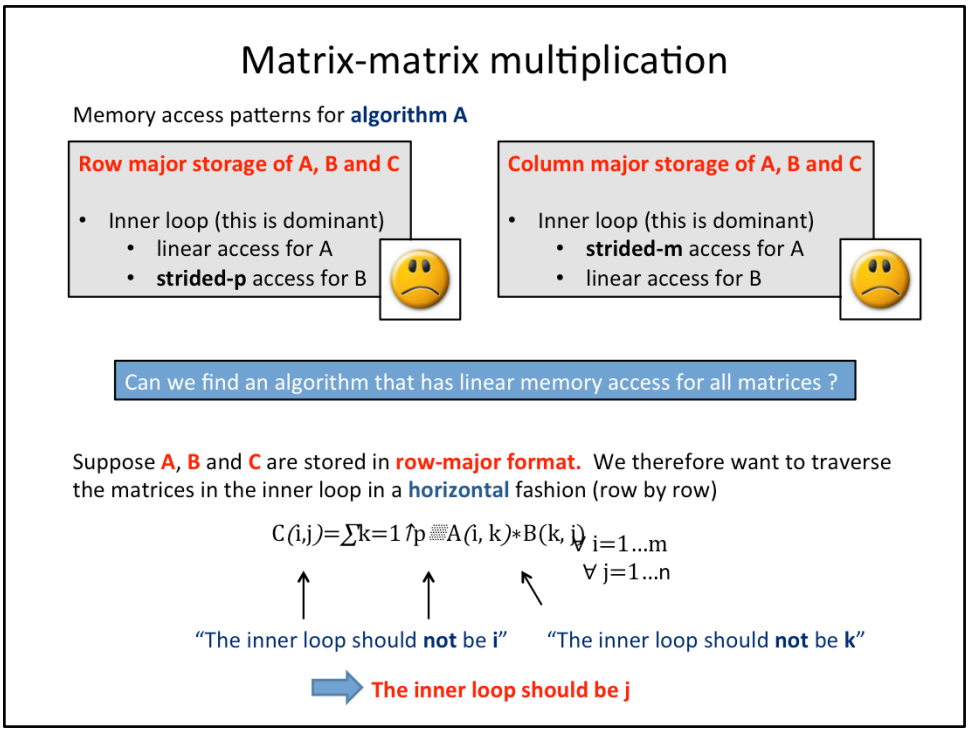
- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK



We now turn our attention to the case of a matrix-matrix multiplication. The biggest difference between the matrix-matrix multiplication and e.g. the matrix-vector multiplication, is that the computational complexity for a matrix-matrix multiplication is O(n*m*p) (hence $O(N^3)$ for square matrices), while the memory complexity is only O(n*m + m*p + p*n) (hence $O(N^2)$ for square matrices). This means that every element of each matrix is read from or written to many times during the execution of the algorithm. This in contrast to the vector triad (streaming memory access) and the matrix-vector product (also streaming access for the matrix) where the elements are used only once. (Note that in the case of the matrix-vector product, the elements of the vector were also accessed multiple times). Because in the matrix-matrix product, every element is accessed many times, it is of the utmost importance that consecutive accesses exhibit temporal locality in order to optimize the cache hit ratio and hence improve performance. In general, one should always consider cache behavior in the case the computational complexity is higher than the memory complexity (reuse of data).

In the next few slides, we first focus our attention on spatial locality (linear memory access). Once we have obtained an algorithm that exhibits good spatial locality, we will focus on the temporal locality.

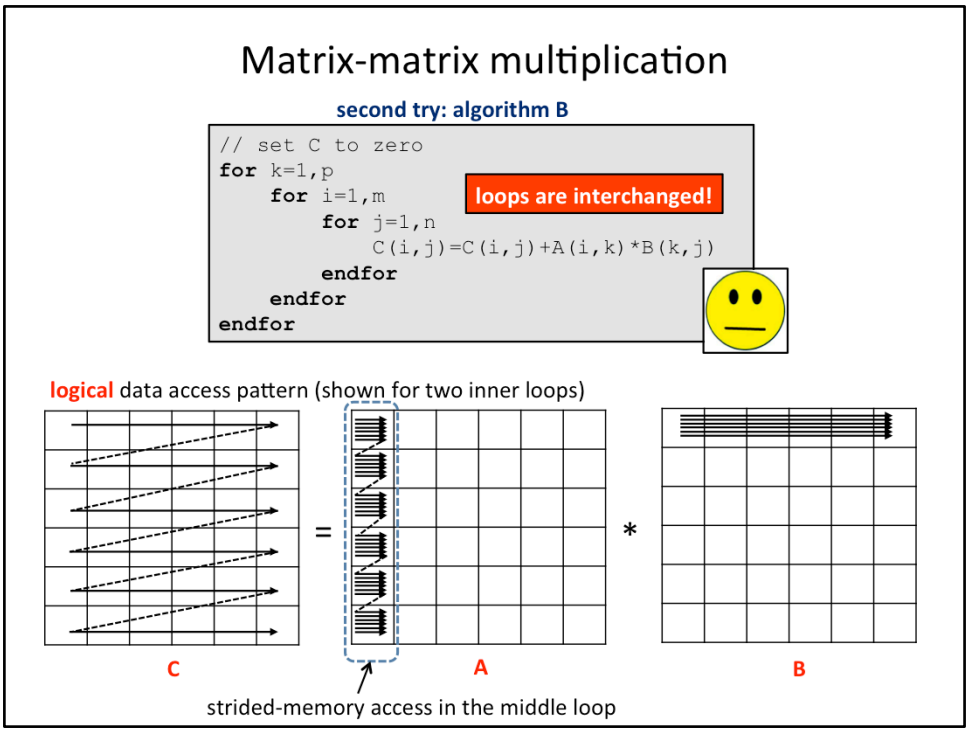
Algorithm A is essentially the most intuitive way of implementing the matrix-matrix



No matter whether we use the row major or column major storage format to store the matrices, we always have strided memory access in one of the matrices for algorithm A.

We assume from this point on that all matrices are stored in **a row-major format.** The question is now, can we find a matrix-matrix implementation that has linear memory access for all matrices?

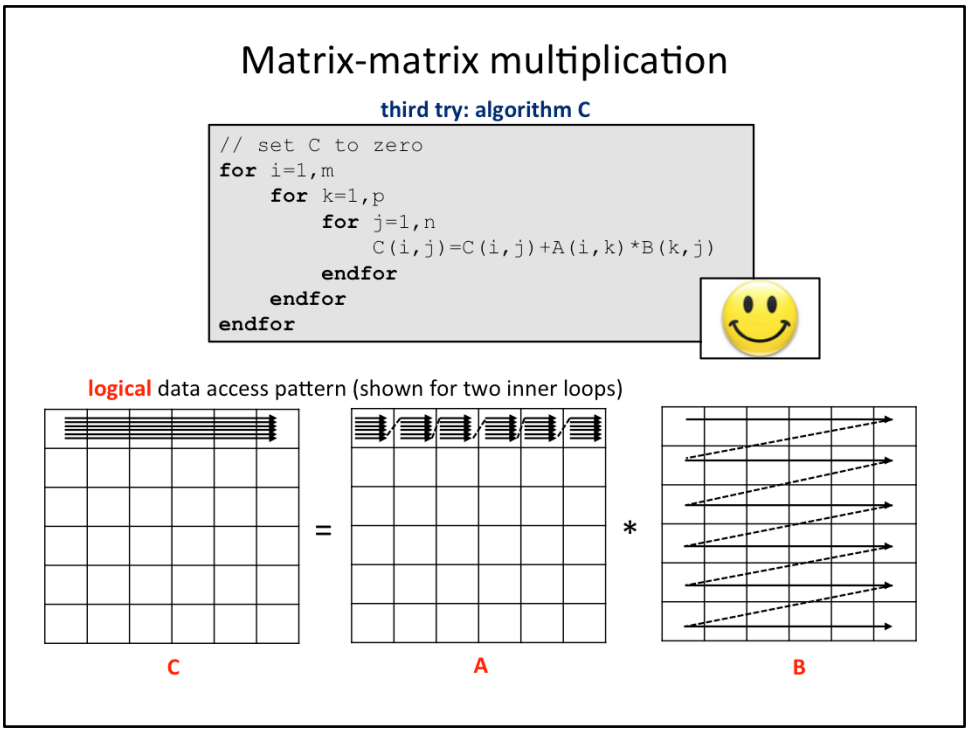
By looking at the definition, we can see that using index k in the inner loop (as in Algorithm A) will lead to strided memory access in matrix B. Similarly, using index i in the inner loop will lead to strided-memory access in both matrix C and matrix A. Therefore, the only possibility left is to choose index j in the inner loop.



(Note that we still assume that A, B and C are stored in **row-major format**).

In algorithm B, we have put index j in the inner loop and index k in the outer loop. Depicted is the logical data access pattern for this algorithm. Note that we have indeed linear memory access for all matrices in the inner loop. However, in the middle loop however, we still have strided-memory access for matrix A. This will lead to cache misses (assuming that the matrices are not too small). Note that algorithm B is therefore already a huge improvement compared to algorithm A.

Note that strictly speaking, the access pattern of matrix B is also strided. However, because of repeated accesses to the same row of B in the middle and inner loop, this will likely not result in cache misses, unless the matrix dimensions are so high that keeping a complete row of B in cache is not feasible. More importantly, the memory access pattern to matrix C is streaming in the middle and the inner loop. This will lead to a write-miss every time a cache line is crossed in matrix C. Because of the write-allocate cache strategy of most caches, a write-miss is usually more expensive than a read-miss. This is because the cache is used twice: once for retrieving a new cache line, and once for updating the corresponding value.



(Note that we still assume that A, B and C are stored in **row-major format**).

Algorithm C is our third try in which index j is still the inner loop, but where index i is taken as the outer loop. Similar to algorithm B, we have linear memory access in the inner loop for matrix A. Moreover, the access pattern of matrix C has improved. If we can assume that a complete row of C can be kept in cache, the number of write-misses is significantly reduced. Therefore, algorithm C has the best ordering of for loops.

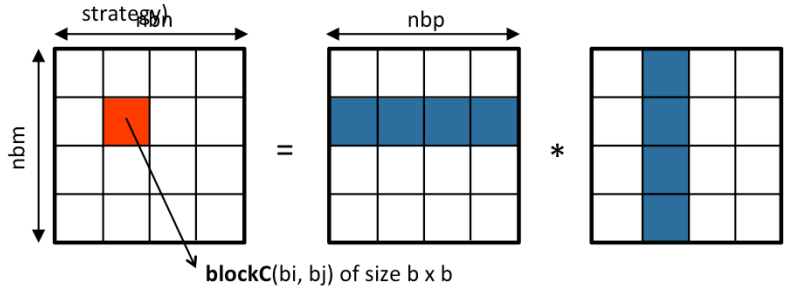
Blocked matrix-matrix product

NxN matrix-matrix product requires **$2N^3$ FLOPS** and **$3N^3$ read operations**

- However, there are only **$3N^2$ data elements** in total
- Each element is roughly **reused** N times
- Possibilities for additional ***temporal locality*** optimizations in cache

We would have perfect temporal locality if all three matrices would fit in cache

- Usually, matrices are too big
- However, we can break the complete matrix-matrix product in a sequence of smaller b x b matrix-matrix products that fit in the cache (blocking strategy)



$$\mathbf{blockC}(bi,bj) = \sum_{bk=1}^{nbp} \mathbf{blockA}(bi,bk) * \mathbf{blockB}(bk,bj) \quad \forall\, bi=1\ldots nbm,\ \forall\, bj=1\ldots nbn$$

We now turn our attention to the temporal locality issue. As mentioned before, every element is read from or written to several times during the execution of the matrix-matrix product. The goal is to keep subsequent accesses to the same element close in time, to that the algorithm can benefit from temporal locality. If the three matrices would fit in cache, we would have perfect temporal locality. Therefore, we reformulate the complete matrix-matrix product as a sequence of smaller matrix-matrix products of size bxb (where b is small enough such that $3b^2$ elements fit in cache – in fact a slightly larger b can be used as well while maintaining good performance). This leads to the blocked algorithm depicted in the slide.

Blocked matrix-matrix product

fourth try: algorithm D (outer loops only)

```
// set C to zero
for bi=1,nbm
    for bk=1,nbp
        for bj=1,nbn
            blockC(bi,bj)=blockC(bi,bj)+blockA(bi,bk)*blockB(bk,bj)
        endfor
    endfor
endfor
```

b x b matrix-matrix product (three more loops)

In the b x b matrix-matrix product:

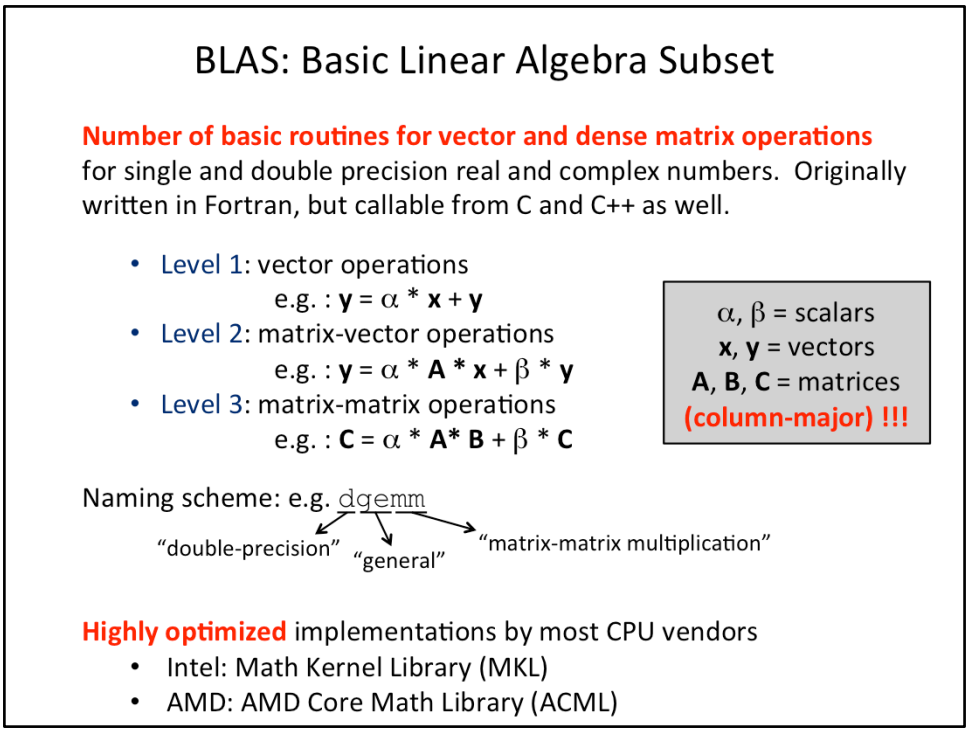
- $2b^3$ FLOPS and $3b^3$ element accesses
- Only $3b^2$ data elements in total
- **Reuse of memory of roughly a factor b !**

Algorithm D contains the pseudocode for the blocked matrix-matrix product. Note that e.g. blockC(bi, bj) refers to a bxb block of matrix C and that therefore, three additional for loops are needed (not shown in algorithm D) to multiply the bxb blocks (e.g. algorithm C).

In general, we get a reuse of roughly a factor of b ($O(b^3)$ memory accesses for only $O(b^2)$ data). Therefore, the bigger b, the bigger the reuse. However, b is limited by the cache size.

Chapter 1: outline

- Classical von Neumann architecture
- Modifications to von Neumann
  - Caching
  - Parallelism in a single CPU core
    - Bit level parallelism
    - Instruction level parallelism
      - pipelining
      - superscalar architecture
      - SIMD instructions
- Case study one: vector triad
- Case study two: matrix-vector multiplication
- Case study three: matrix-matrix multiplication
- High-performance libraries: BLAS and LAPACK

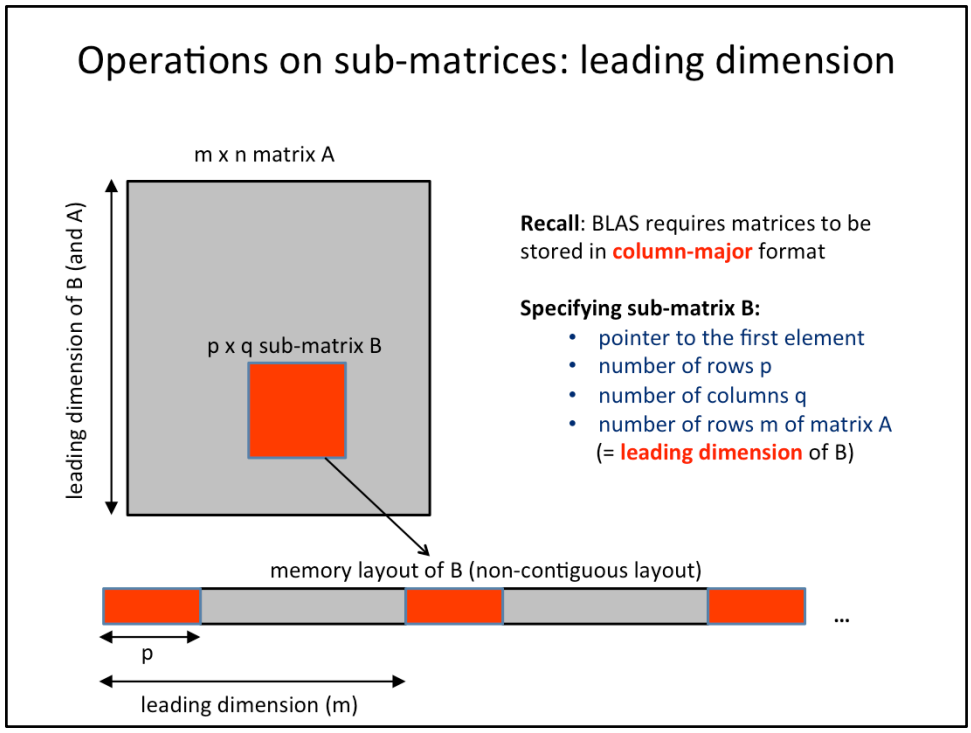


BLAS is a limited set of routines for very basic linear algebra operations. There are three categories of routines:

Level 1: routines for operations on vectors, e.g. scaling a vector, performing a dot product of a vector, etc.
Level 2: routines for operations on matrices and vectors: matrix-vector product and solving a system of equations y = A*x when A is a triangular matrix.
Level 3: routines for operations on matrices and matrices: matrix-matrix product.

BLAS supports general dense matrices, and a number of symmetric and banded matrices. Note however that BLAS does not support general sparse matrices. A quick reference guide to the BLAS routines is provided in Minerva.

Many CPU vendors provide highly optimized versions of the BLAS routines. For the x86 architecture, this is MKL for Intel and ACML for AMD. These libraries automatically detect the CPU they're running on, along with the size and type of the cache memory, in order to use the most optimized implementation for that BLAS routine. Most Linux distributions also provide BLAS libraries that are merely compiled version of the reference implementation. They are hence not especially optimized for a certain CPU.



Consider an m by n matrix A which is stored using the column major format. Now consider a p by q submatrix B of A. The problem is now that B is not stored contiguously in memory, because consecutive columns of B are interspaced with elements from A (see slide). In order for the BLAS routine to be able to calculate the memory address of each element of B, it is sufficient to additionally specify the number of rows (m) of the matrix A, in which matrix B is embedded. This is also called the leading dimension.

Obviously, in case BLAS operates on the complete matrix, the leading dimension is equal to the number of rows. In other words, the leading dimension of matrix A is also m.

LAPACK: Linear Algebra Package

Number of more advanced routines for linear algebra:

- Complementary to the BLAS
- **Used for:**
  - solving systems of equations
  - least squares problem
  - singular value decomposition
  - eigenvalue problems
- Written keeping **numerical stability** in mind.
- The implementations call the **BLAS routines** whenever possible
  - Most of the CPU cycles are spent inside BLAS routines
  - Highly optimized BLAS routines will therefore lead to highly optimized LAPACK routines.
  - Available in MKL (for Intel CPUs) and ACML (for AMD CPUs)

Again, use optimized LAPACK libraries whenever possible. Especially in the case of large matrices, try to use LAPACK as much as possible, in the interest of obtaining high CPU speed an numerical stability.

Calling Fortran routines from C or C++

- Function declaration in C++, e.g. for dgemv (BLAS routine)

```
extern "C" void dgemv_(const char *trans, const int *m,
       const int *n, const double *alpha, const double *A,
       const int *LDA, const double *x, const int *incx,
       const double *beta, double *y, const int *incy);
```

- Calling the function

```
int m = 100, int n = 100;
double *A, *x, *y;
// … allocate memory for A, x and Y; initialize A and x

const int inc = 1;
const double alpha = 1.0;
const double beta = 0.0;
char trans = 'N';

dgemv_(&trans, &m, &n, &alpha, A, &m, x, &inc, &beta, y,&inc);
```

BLAS and LAPACK are usually provided as a library with a Fortran interface. These can easily be linked against C/C++ programs. Note that BLAS and LAPACK do not include a C/C++ header file which contain the function declarations. Therefore, we need to create our own function declarations. In C++, such function declaration must start with extern “C” to notify the C++ compiler to use C-style linking conventions (which are largely compatible with Fortran). The function name itself is identical to the Fortran subroutine name, but then in lowercase and with a trailing “_”. This scheme is adopted by most compilers, but other naming schemes exist as well (e.g. two trailing underscore characters and/or all capitalized letters). All arguments are passed as pointers, even when they are scalar input parameters. Furthermore, the C/C++ types must correspond to the ones in Fortran, e.g., the Fortran REAL type corresponds to the float type, INTEGER to int, etc.

The end