CERN

Physics Event Generator Computing Workshop

Hackathon

Servesh Muralidharan

IT-DI-WLCG, UP Team

CERN

28 Nov 2018

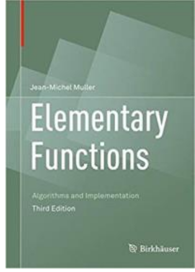
Logistics

- Connect to machines at CERN to run the exercises: user name your CERN sso

- Thanks to the teams at CERN for providing machines and access, including:

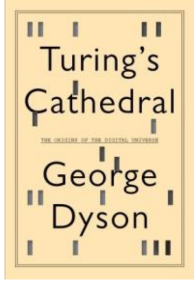
Luca Atzori	Guillermo Izquierdo
Maria Girone	Alberto Di Meglio
Fons Rademakers	Eric Bonfillou

Concept of an Ideal Program

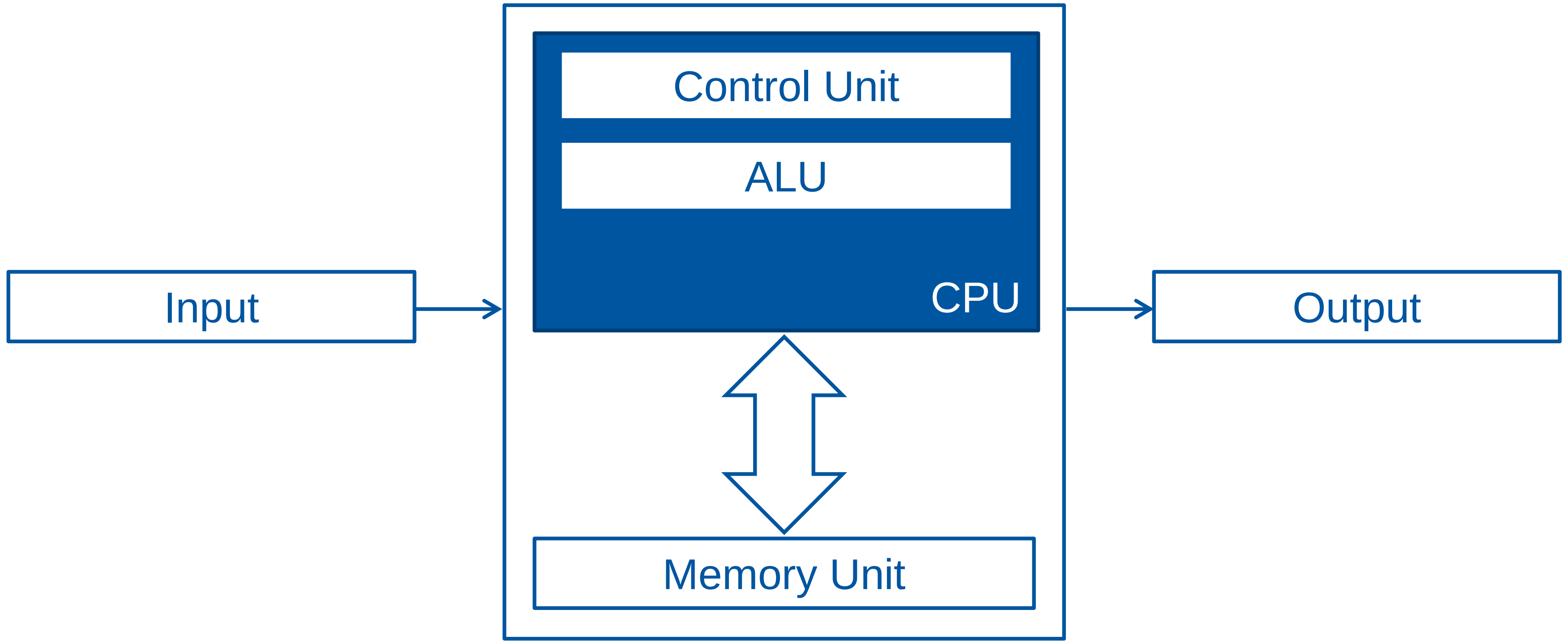

Jean-Michel Muller
Elementary Functions
Algorithms and Implementation
Third Edition
Birkhäuser

- Readability Vs Performance
- Compute Algorithm
  - **20% – 30% of code but has >90% of run time**
  - Most optimizations are applied here
  - *Big O notation*
    - Describes the worst case performance in terms of input size
    - Can represent time or space
    - Example: O(n) – Linear (Finding an item in an unsorted array)
  - Extremely readable code for **compilers**
    - Elegance and Obscurity
    - Compilers will love it and we only care about performance!!!
    - Be nice and use explicit comments for fellow human beings to understand
- Glue code
  - **70% – 80% of code but has <10% of run time**
  - Contains code used for structuring and connecting different blocks
  - Extremely readable code for **humans**
    - Compilers will hate you but that’s okay, we don’t care about performance here

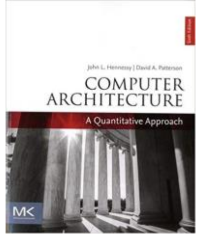
Essentials: Computer Architecture



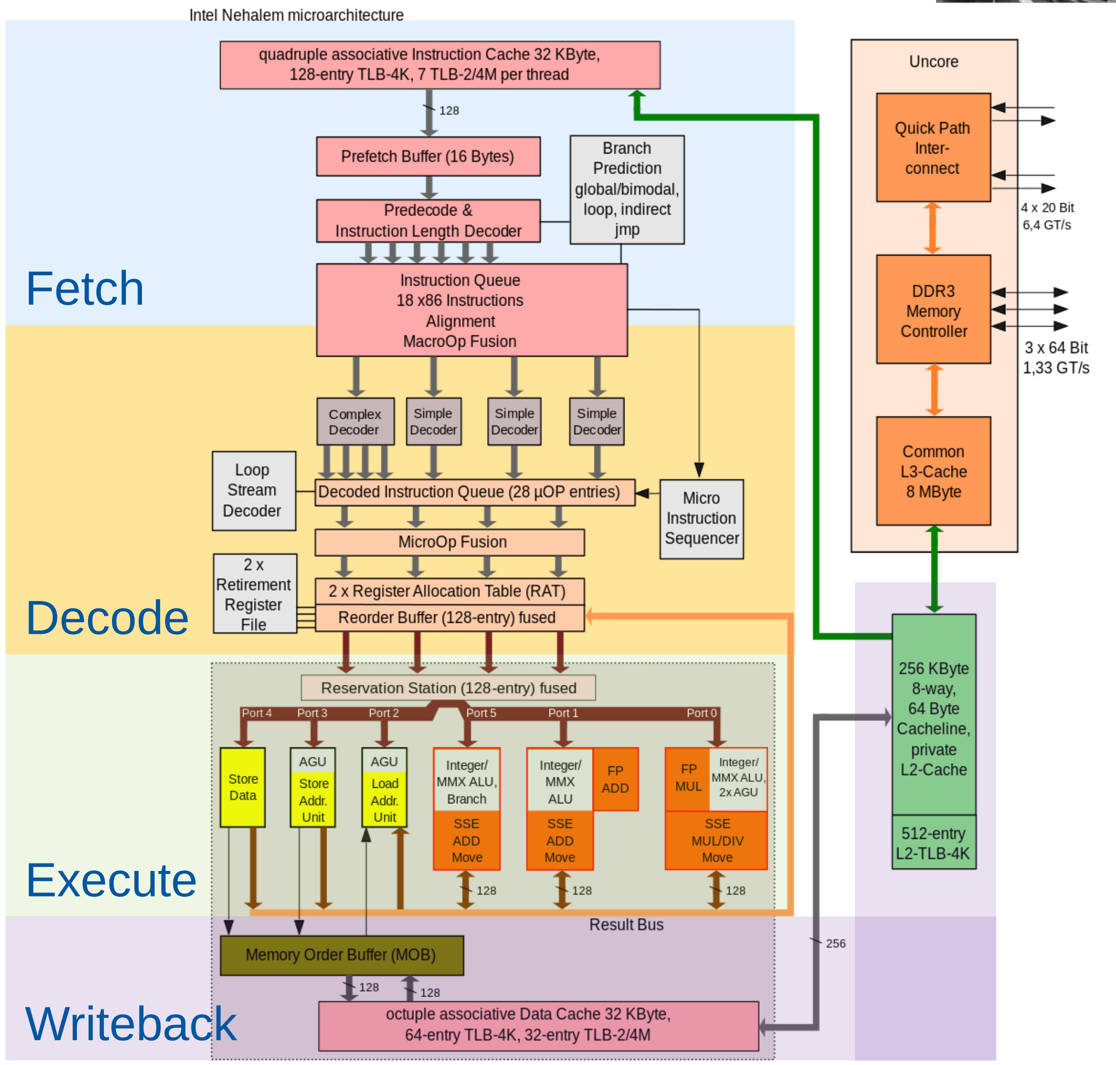
- Stored-Program computer
  - Von Neumann model
  - Program and data are stored in memory and then processed



Instruction processing



	Step	What happens
1	Fetch	Get the instruction (from memory or cache)
2	Decode	Translate the x86 machine codes op codes into (possibly multiple) internal operations
3	Execute	Do the instruction (may require memory access)
4	Writeback	Write the result to storage or make visible in the architectural registers (Retirement)



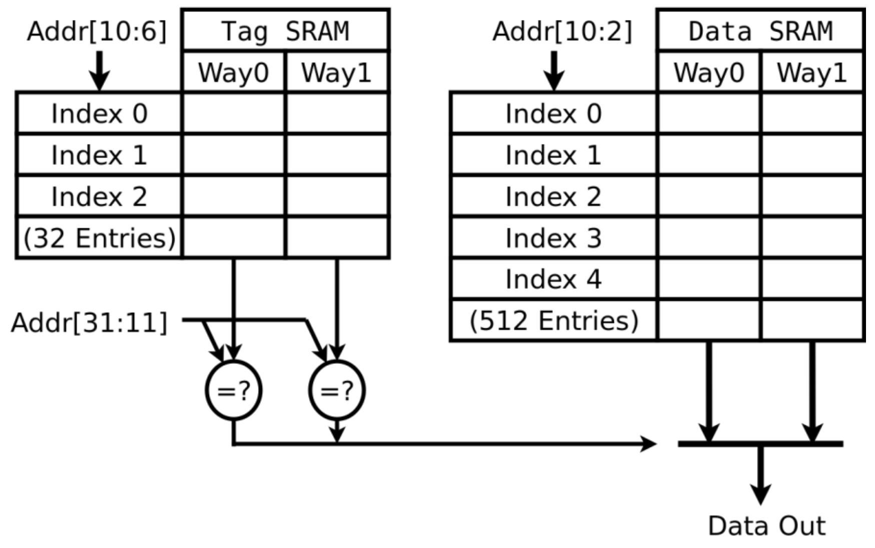
Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Fetch
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Decode
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
FP ADD
SSE ADD Move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Execute
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
Writeback
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

https://commons.wikimedia.org/wiki/File:Intel_Nehalem_arch.svg

Cache lines

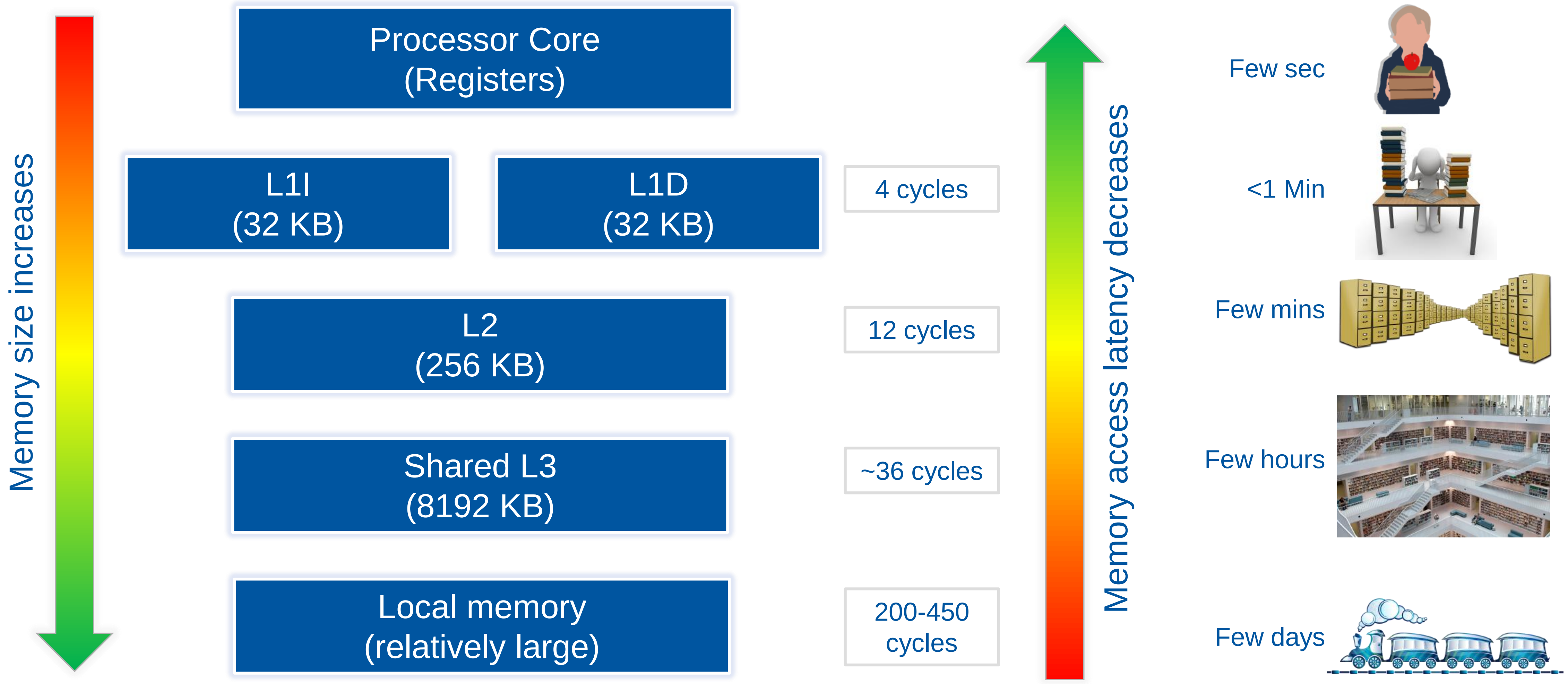
- Data is always fetched in units of a cache line
  - On x86 cache line size is 64 bytes
  - Usually data read from main memory still be stored in the cache, so even if you need less than the size of a cache line 64 bytes will be fetched
  - Caches have to be kept consistent if the same cache line is present in multiple caches: e.g. those associated to different cores or different processor sockets within a machine



4KB, 2-way set-associative 64B line cache read path

https://commons.wikimedia.org/wiki/File:Cache,associative-read.svg

Memory Hierarchy

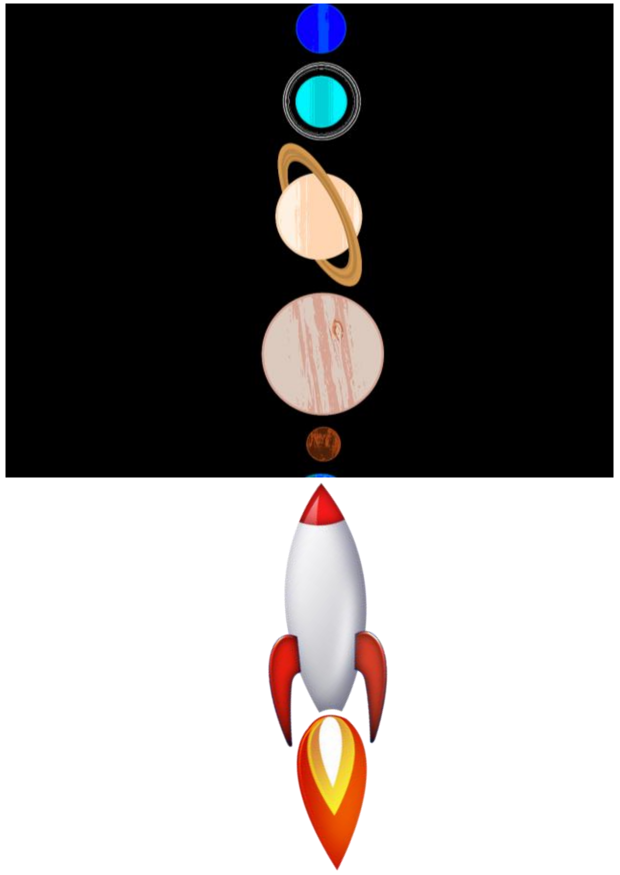


Approximate memory latencies on Intel Haswell CPUs

Adapted from S. Jarp, A. Nowak

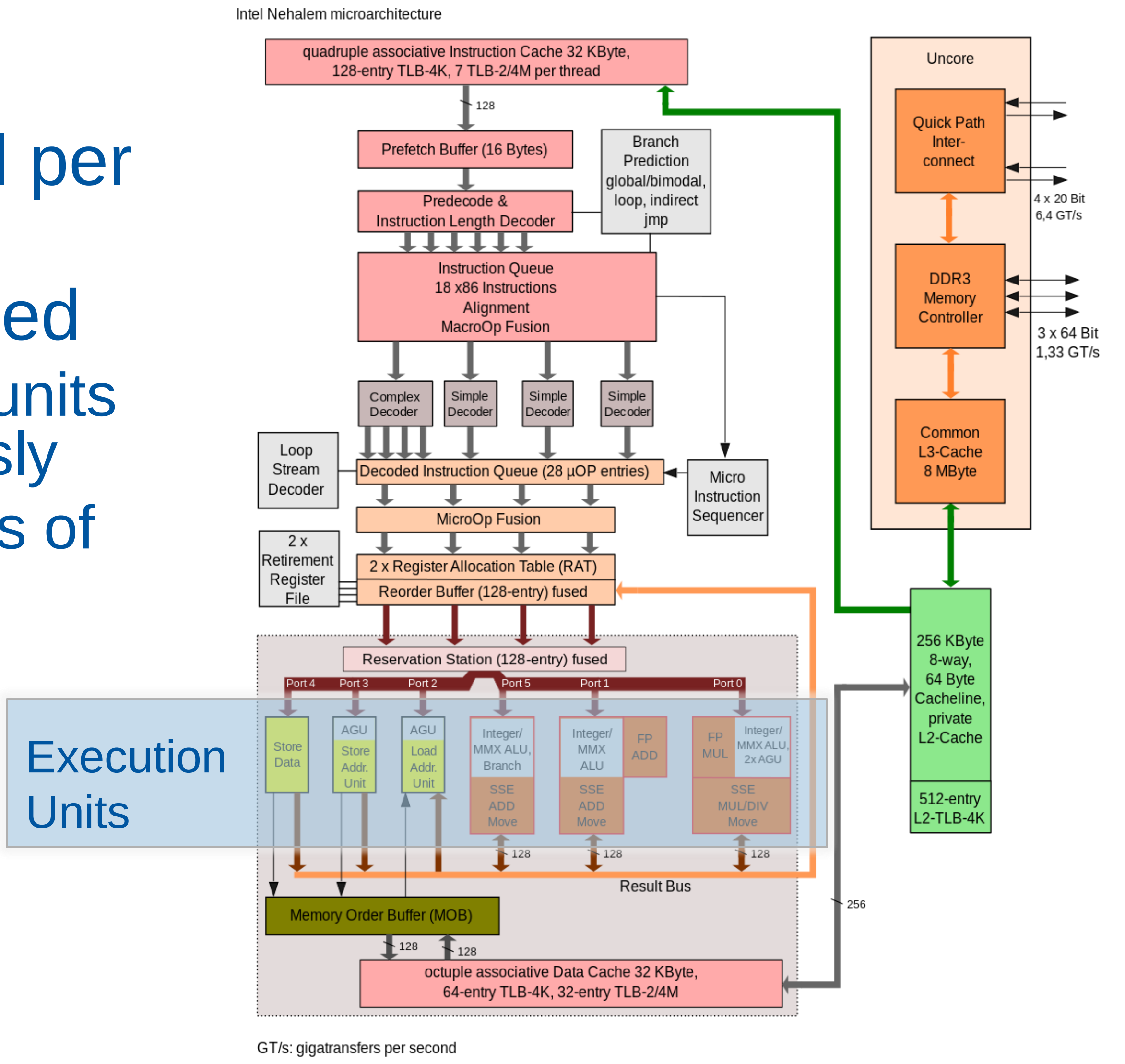
Memory and Disk

- Today we are concerned mostly with main memory (RAM) when talking storage outside the processor
  - Typically 1 to 100s of Gigabytes in size
- However often data will be on a storage device like:
  - Object storage, Disks, SSDs, NVMe devices
  - These will have latencies from 50 to +100,000 times longer than main memory
  - But often much larger capacity, multiple terabytes or petabytes



1st form of HW Parallelism: Instruction level parallelism (Absolutely Free*)

- Multiple instructions can be decoded per cycle
- Multiple instructions can be dispatched
  - Because there are multiple execution units (ports) they may execute simultaneously
  - The execution units take different types of instructions when the port is available
- Called *superscalar architecture*


Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Execution Units
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
FP ADD
SSE ADD Move
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K
GT/s: gigatransfers per second

https://commons.wikimedia.org/wiki/File:Intel_Nehalem_arch.svg

2nd form of HW Parallelism: Pipelining (Absolutely Free*)

- Only possible if the execution flow of the code is understood by the processor
- Ex: Compiler hints, Strided memory access, Loop bounds

Without pipelining approach

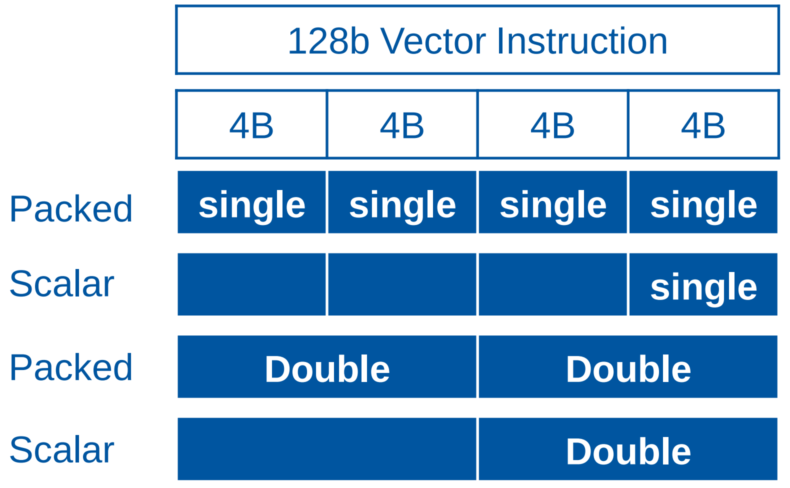
Instruction	1				2			
Fetch	X				X			
Decode		X				X		
Execute			X				X	
Write				X				X
Clock	1	2	3	4	5	6	7	8

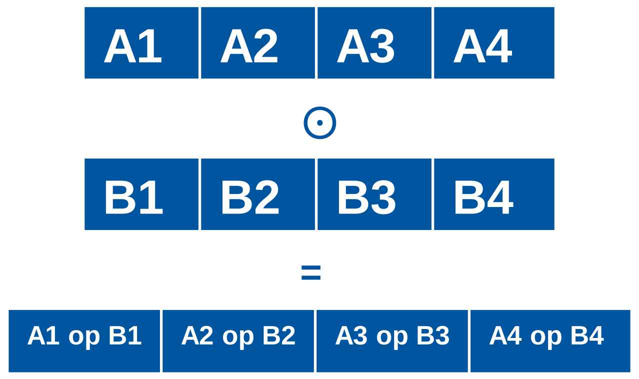
With pipelining approach

Instruction	1	2			
Fetch	X	X			
Decode		X	X		
Execute			X	X	
Write				X	X
Clock	1	2	3	4	5

3rd form of HW Parallelism: Vector Instructions

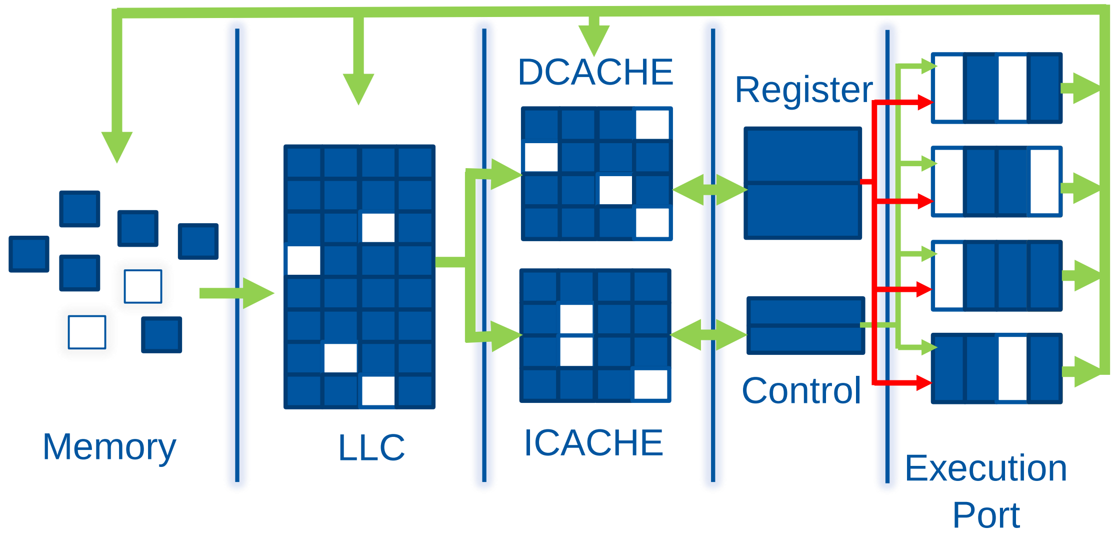
- There are vector registers
  - e.g. in processors with Intel AVX there are registers YMM0 – YMM15 which are 256 bits in length
- Example of one instruction that operates on multiple data
- Data may be placed into these in various ways
  - Scalar (just some of the width for one value)
  - Packed (a number of values one after the other)

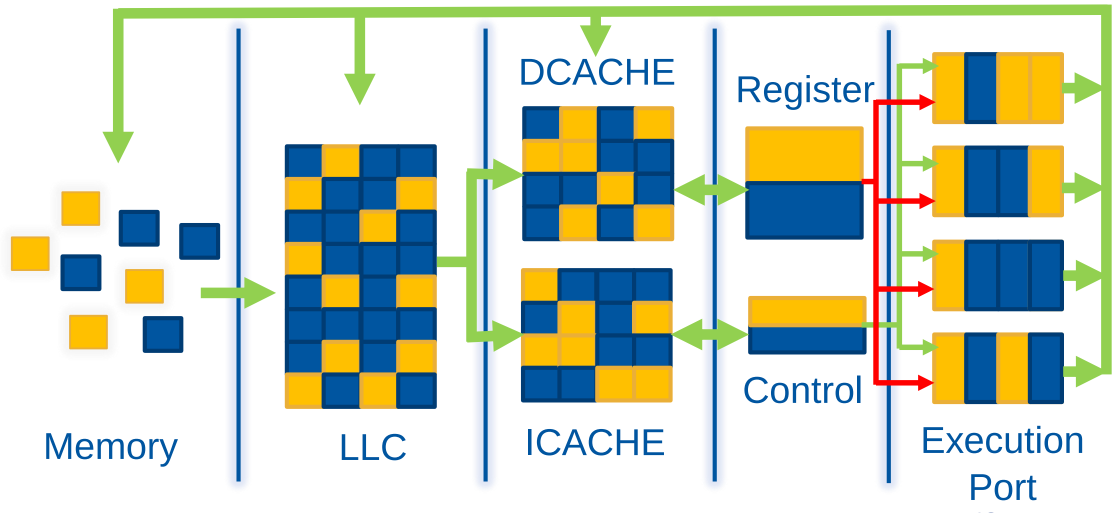




4th form of HW Parallelism: Hardware threads A.K.A Resource Multiplexing

- Hardware thread
  - The register files and instruction pointer to provide the architectural execution environment. e.g. rax, rbx, the instruction pointer and other registers
  - One ore more hardware threads share the resources of a core
  - Instruction executed by a core is tagged as belonging to the associated hardware thread
  - Intel called this hyperthreading (HT) or generally Simultaneous Multithreading (SMT)

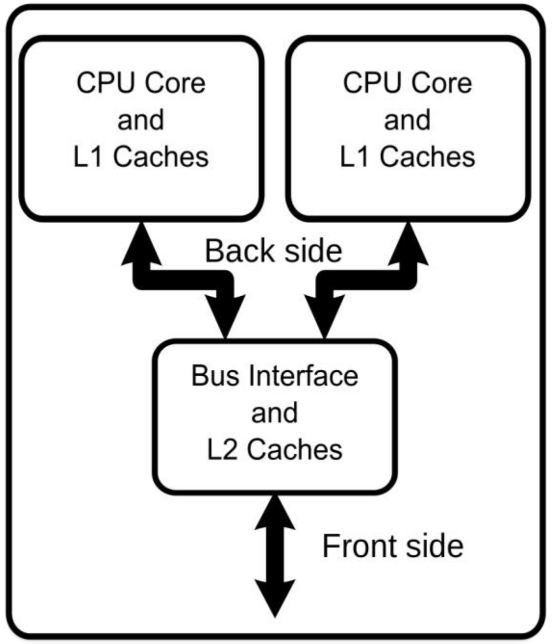




🟨 HW Thread 2
🟦 HW Thread 1

5th form of HW Parallelism: Multicore

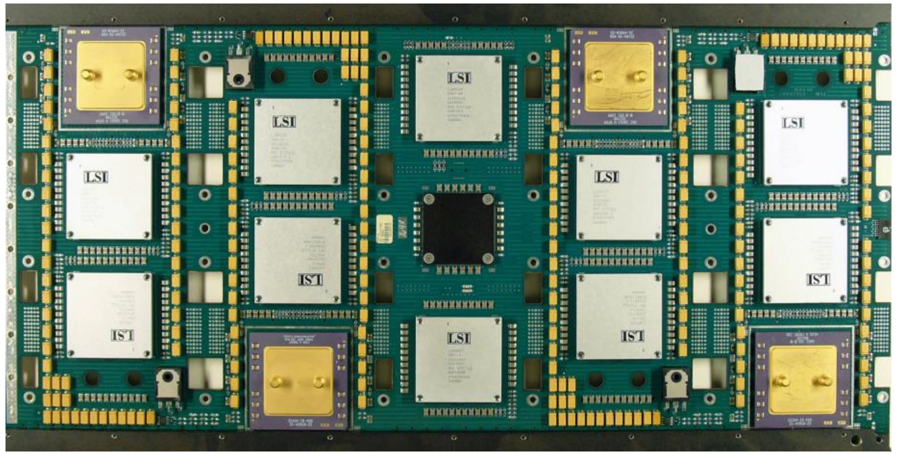
- Core
  - The execution logic, cache, and facilities for storing execution state. e.g. register files



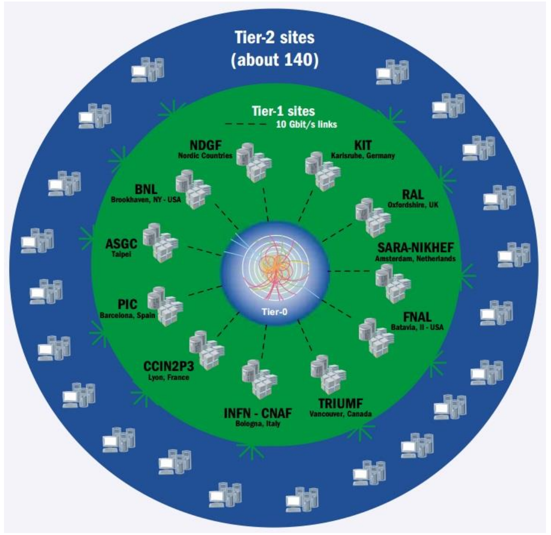
https://commons.wikimedia.org/wiki/File:Dual_Core_Generic.svg

6th form of HW Parallelism and on... Multisocket, Cluster, Grid…

https://commons.wikimedia.org/wiki/File:High_Performance_Computing_Center_Stuttgart_HLRS_2015_10_Cray_XC40_Hazel_Hen.jpg

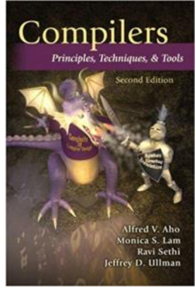


https://commons.wikimedia.org/wiki/File:Processor_board_cray-2_hg.jpg

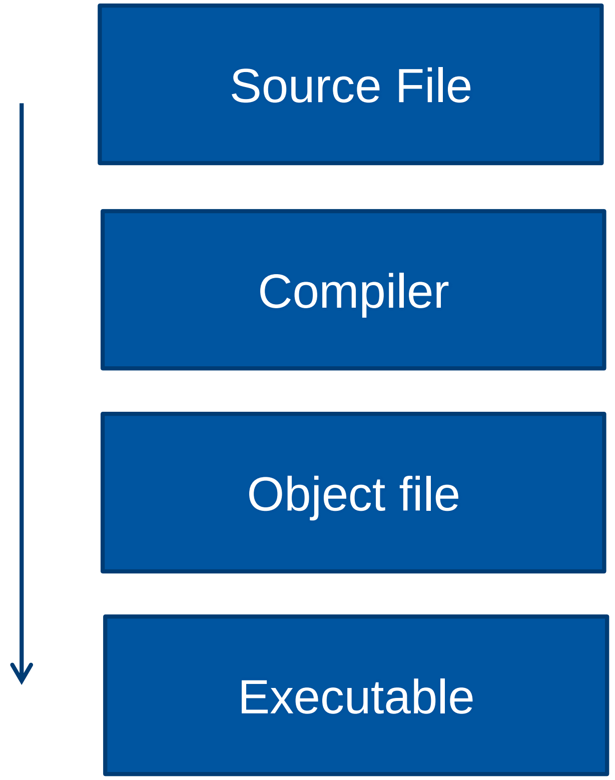


https://sciencenode.org/feature/large-hadron-colliders-worldwide-computer.php

The Compiler



- Compiler is the bridge between your code and the hardware
- Consist of a front-end and back-end
- Front-end:
  - Language focused
- Back-end:
  - Machine focus: architecture specific analysis
  - Optimization
  - Code generation in native machine code



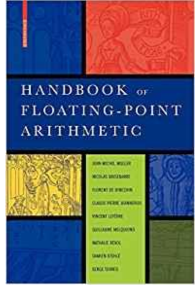
Compiler

- Compiler is one of the layers which can help in having well performing code
- Compiler features can give you performance with no change of your code (‘for free’)
  - Each compiler is different
  - Different releases of a compiler can behave differently, to give different performance or even different results

Compiler

- Provide your compiler as much information as possible
  - Typically make loops explicit
    - Let the compiler generate code which can reason on the number of iterations
      - Don’t break out of the loop early
      - Try to keep memory access contiguous
      - Keep arithmetic operations together
  - Some flags may change results
  - Tune for your target architecture if you can

Floating Point essentials



- Floating point numbers are a way to represent real numbers, stored as a significand (mantissa), exponent and sign
- $N = (-1)^s \times 1.ccc\ldots \times b^{qqq\ldots}$
  - Finite number of floating point numbers of a given width (e.g. doubles) whereas an uncountable number of real numbers
- Therefore while the FP operations are precisely defined by IEEE-754 the result will usually need to be rounded.
  - Usually b = 2 and ccc.. and qqq… are stored as binary representation. b=10 (decimal) is also specified by the standard
  - Some rational numbers which can be represented as terminating decimals are recurring when using base 2
    - numbers like 0.1 can only be represented as a truncated number (i.e. are rounded) in floating point

Floating Point

- Some basic properties of operations on real numbers do not hold on floating point numbers
  - E.g. associativity. Generally:

$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$$

(*a* Ä *b*) Ä *c* ¹ *a* Ä (*b* Ä *c*)

Where

$\oplus$ denotes floating point addition,

Ä denotes floating point multiplication

Matrix Multiplication

A=$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$ B=$\begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{bmatrix}$

C=AB=$\begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{bmatrix}$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

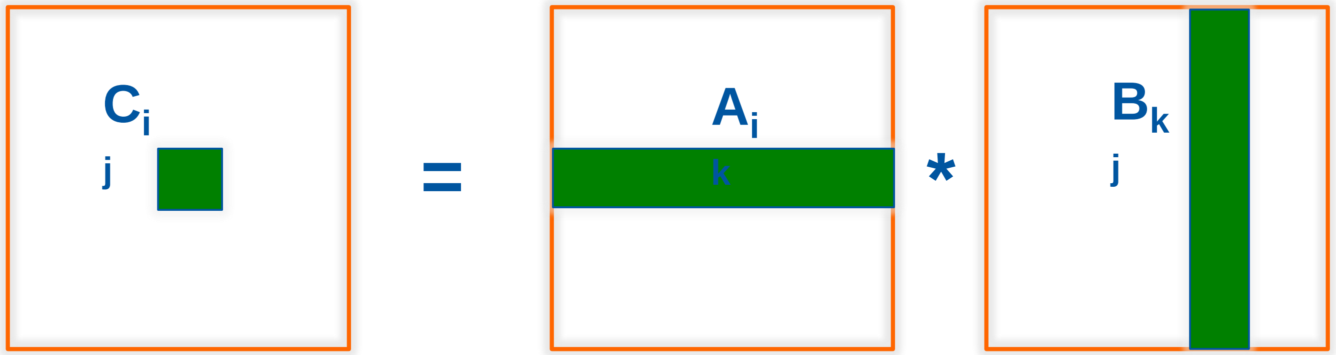
Why Matrix Multiplication???

- Matrix multiplication is fundamental to large numerical computations
  - Large math problems can be decomposed into linear equations and then solved using MM
  - Motivation behind the development of high performance math libraries such as BLAS, MKL, etc./
  - LINPACK benchmark used to rank Supercomputers is based on matrix operations
- Computer scientists are obsessed with matrix multiplication optimizations
  - For very good reasons!!!
  - The naive version has $O(n^3)$ complexity
    - However, several algorithms that do it faster already exists
  - Arithmetic Intensity
    - Naïve algorithm has a very low Floating point operations per byte
      - Fundamental computation involves 2 operations for every 3 numbers
  - Several hundred papers
- For the hands on we will use matrix multiplication as an illustration
  - However, if you want to do matrix multiplication in your application use a common library
    - If you want to know why we can give you more than hundred different reasons (with proof!!!)

First Attempt

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

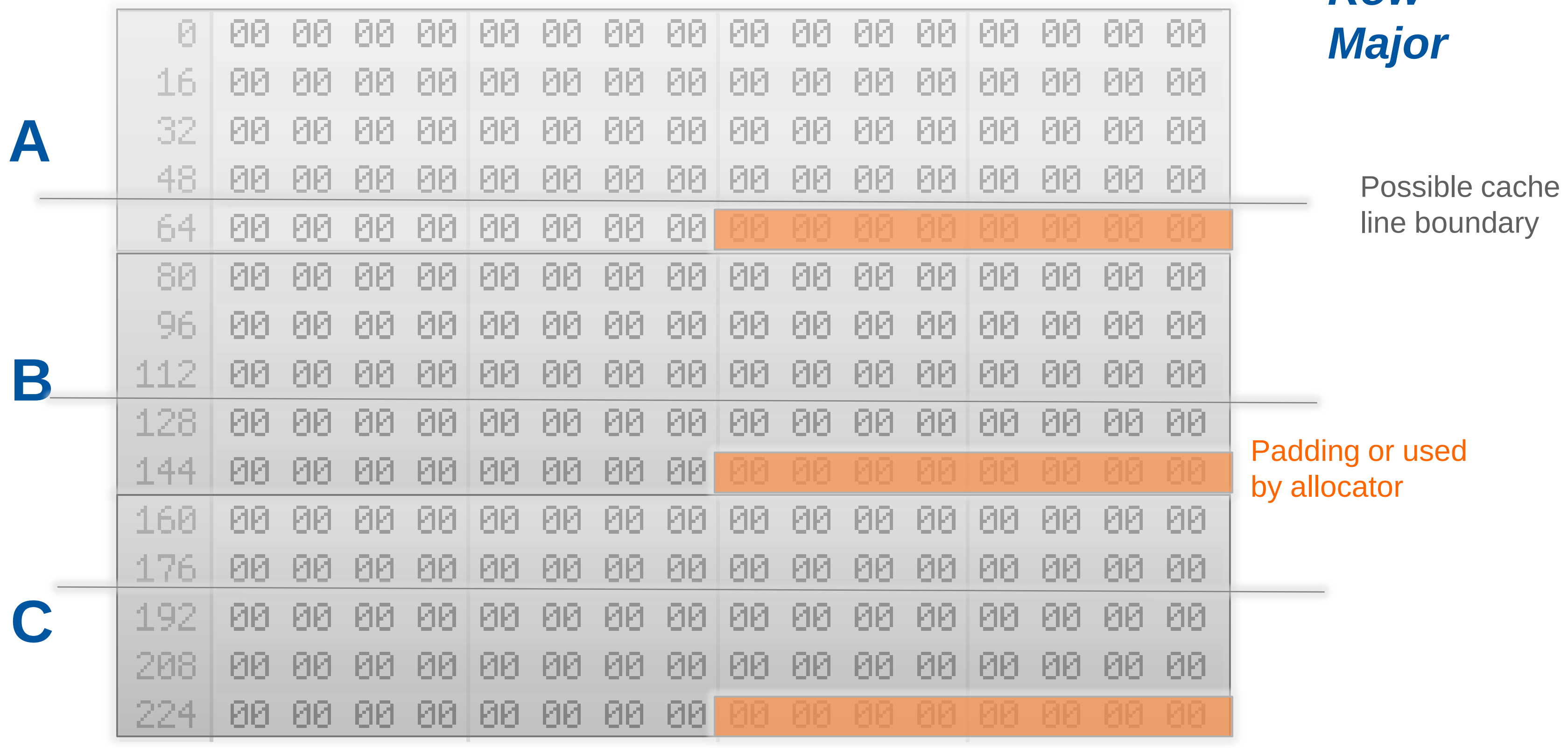
```
for(size_t i=0;i<n;++i) {
  for(size_t j=0;j<p;++j) {
    sum = 0;
    for(size_t k=0;k<m;++k) {
      sum += a[i*m+k] * b[k*p+j];
    }
    c[i*p+j] = sum;
  }
}
```



Possible memory layout

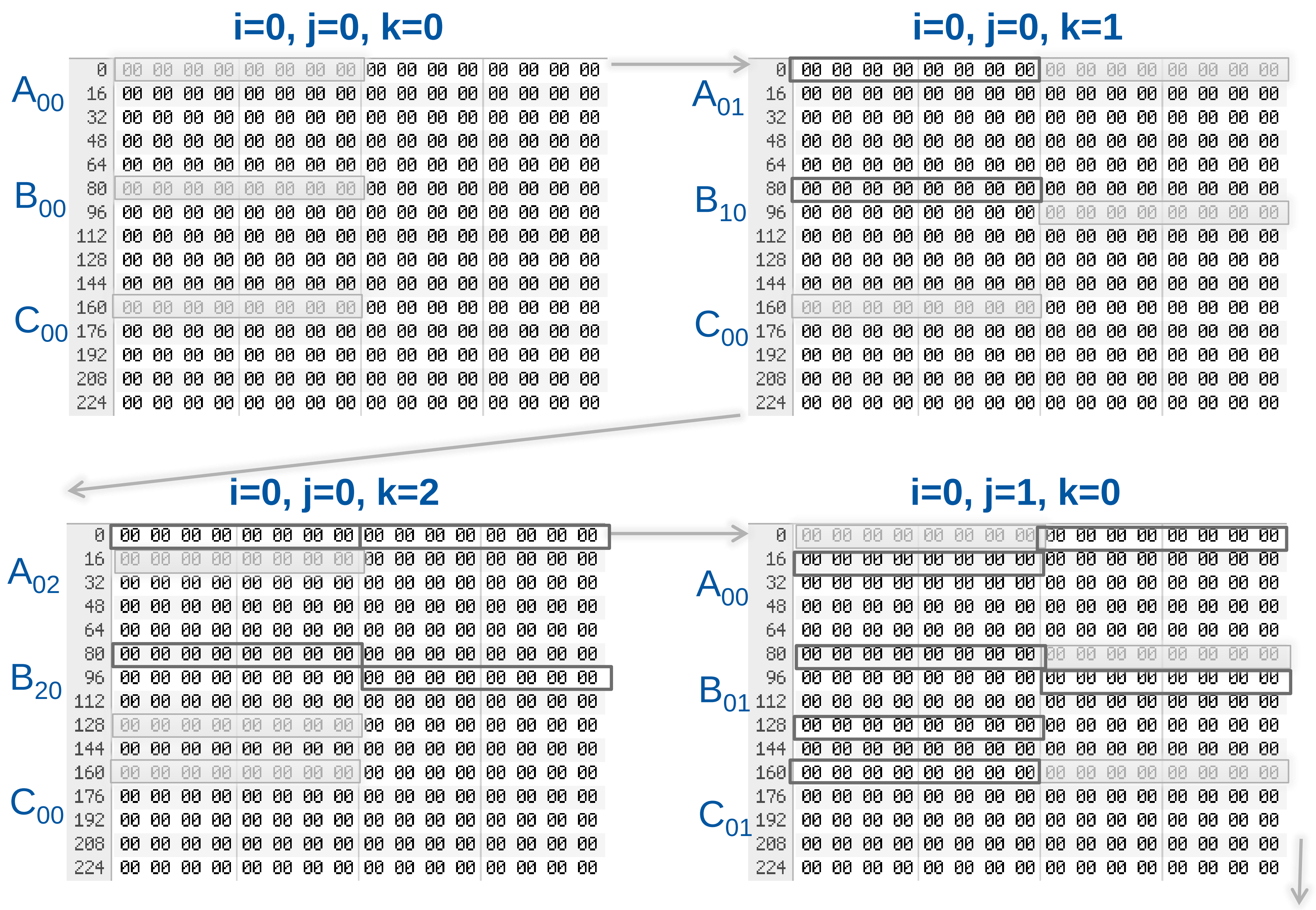
- e.g. A, B and C are 3 x 3 matrix of doubles

Row Major



```
  0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Memory access pattern



i=0, j=0, k=0

```
  0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

i=0, j=0, k=1

```
  0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

i=0, j=0, k=2

```
  0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

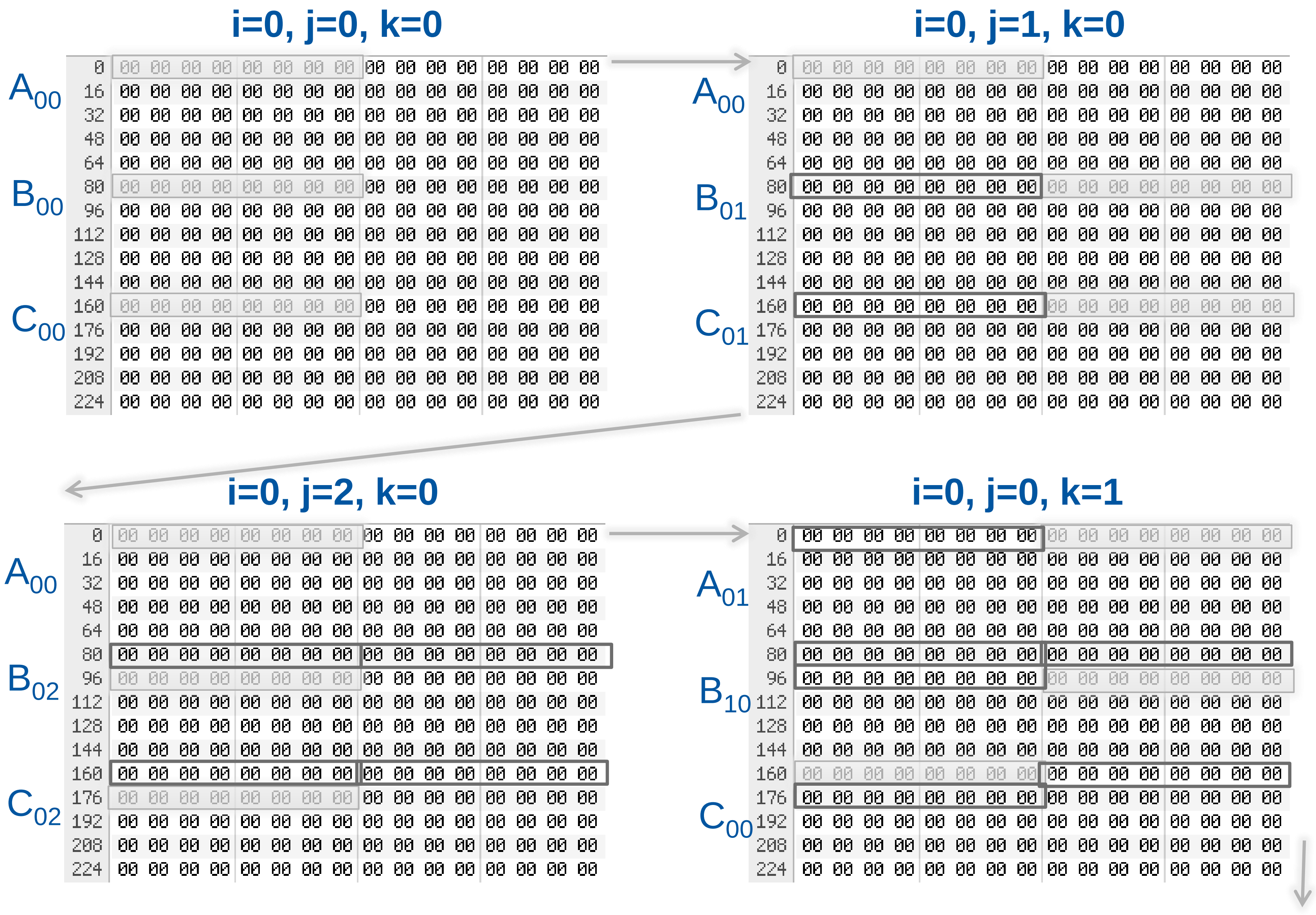
i=0, j=1, k=0

```
  0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Improvement: Order memory access

- By ordering the access differently can improve cache locality
- Try to access memory in a previously access cache line (e.g. improve spatial locality)

Change order



```
  0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
  0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
  0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
  0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 96 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
128 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
144 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
176 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
192 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
208 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
224 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Cache Blocking (also called tiling)

- Divide a problem into pieces where work can be done on a subset of the data, and where the subset fits into the CPU caches, e.g. where the value of *block* is chosen so the three inner loops can run over data that fits inside the cache

```
/* This gives C = A*B + C */
for(size_t i=0;i<n;i+=block) {
  for(size_t j=0;j<p;j+=block) {
    for(size_t k=0;k<m;k+=block) {

      for(size_t ii=i; ii<min(i+block,n);++ii) {
        for(size_t jj=j; jj<min(j+block,p);++jj) {
          sum = c[ii*p+jj];
          for(size_t kk=k; kk<min(k+block,m);++kk) {
            sum += a[ii*m+kk] * b[kk*p+jj];
          }
          c[ii*p+jj] = sum;
        }
      }

    }
  }
}
```

Profiling

- Utilization of resources
  - What? Why? How?
  - Help understand performance i.e. Better bang for buck
- Profiling with a view to understand performance
  - Resource specific
    - Examine utilization, saturation or errors specific to a resource
  - Code specific
    - Investigate resources used by the program and its functions
  - Dependable results
    - Strict methodology that is free from bias
    - Reputability of the measurements

Things that impact profiling

- What affects the timings?
  - Other workloads
  - Concurrent access to disk, network, memory etc.
  - Frequency scaling (Intel Turbo Boost)
  - Non-uniform memory access (NUMA)
  - CPU scheduler
- There can still be large time variations
- Always do multiple runs, make sure each run has similar conditions, e.g. if you program accesses data from files
  - Remove staging in files
  - Empty filesystem caches
    - `echo 3 > /proc/sys/vm/drop_caches`

How to profile

- Profiling at the level of an application
  - Add timings into source code
  - Measure call counts etc
- Disadvantages
  - Creates overhead
  - It is not always possible to recompile the code
    - Production software is large and complex
    - Source code is not readily available

How to profile

- Kernel, OS and hardware offer a variety of tools and interfaces to measure high and low-level events with little performance impact. This can be done in an intrusive and non-intrusive way. Some examples for non-intrusive techniques:
  - Performance Monitoring Unit
  - /proc - is a virtual filesystem that represents the current state of the Linux kernel
  - Linux Control Groups (cgroups) limits and monitors resource usage of processes
- Some examples of intrusive techniques:
  - Dynamic linker allows one to replace symbols at runtime
    - Instrumented function, e.g. malloc

How to profile

- Performance Monitoring Unit
  - Number of registers, counters and features supported are CPU specific. The following can be measured:
    - CPU cycles
    - Branch predictions
    - Instructions
    - Cache accesses
    - Memory accesses
    - *Any many other things…*
  - Will use this today (via the tool *perf*)

How to profile – non-intrusive tools

- Performance Monitoring Unit – Problems:
  - Vast amount of counters – correlation between them can be difficult to understand
  - Encoding can change for different CPU models
- Tools to read counters from PMU:
  - perf https://perf.wiki.kernel.org/index.php/Tutorial
  - pmu-tools *is a wrapper around perf*
    - https://github.com/andikleen/pmu-tools

Exercise 1 – Naïve matrix multiplication

- Investigate the basic (naïve) matrix multiplication
  - The source and 'makefiles' are setup to build binaries with *gcc* and *icc*
  - The example is setup to multiply two square matrices of a size (order) given on the command line, using double precision floating point
- Time how long the multiplication takes, e.g. with order 2300
- Use perf to measure the number of cycles and number of retired instructions
  - Calculate the IPC count for the application

- Amount of work performed per clock tick (Instructions Per Cycle):

$$IPC = \frac{Retired\ Instructions}{CPU\ clock\ cycles}$$

```
perf stat –e cpu-cycles,instructions ./program
```

Exercise 2 – Change data access order

- Adapt the naïve multiplication and try every ordering of the 3 for loops
  - Measure the execution time of all the combinations
- Consider the order of data access
- For the i, k, j ordering
  - Measure IPC again
  - Measure the LLC cache misses
    - Remeasure the naïve case and compare

- Last level cache: relevant perf event names LLC-load-misses, LLC-prefetch-misses and for totals LLC-loads, LLC-prefetches

```
perf stat -e LLC-load-misses,LLC-prefetch-misses ./program
```

Exercise 3 – Rewrite with blocking

- Taking the i,k,j loop order for the previous example and adapt the multiplication routine to use cache blocking
  - Leave the block size as a parameter that you can adjust
  - Measure the runtime, IPC and LLC misses with varying blocksize:
    - try sizes 64, 128, 256, 512, 1024, 2048, 4096 matrix rows/columns (but you must use vectors of sufficiently large order)
  - Estimate the working size of the data being access for the optimum blocksize. How does it compare to the CPU cache size?
    - You can use the following to get information:

```
getconf -a | grep CACHE
```

Sources of Parallelism in Modern Architectures

1. Instruction level parallelism (ILP)
2. Pipelining
3. Vector Operations
4. Hardware Threads
5. Multicore
6. Multi Socket
7. Cluster
8. Grid

Architecture handles this with good coding practice

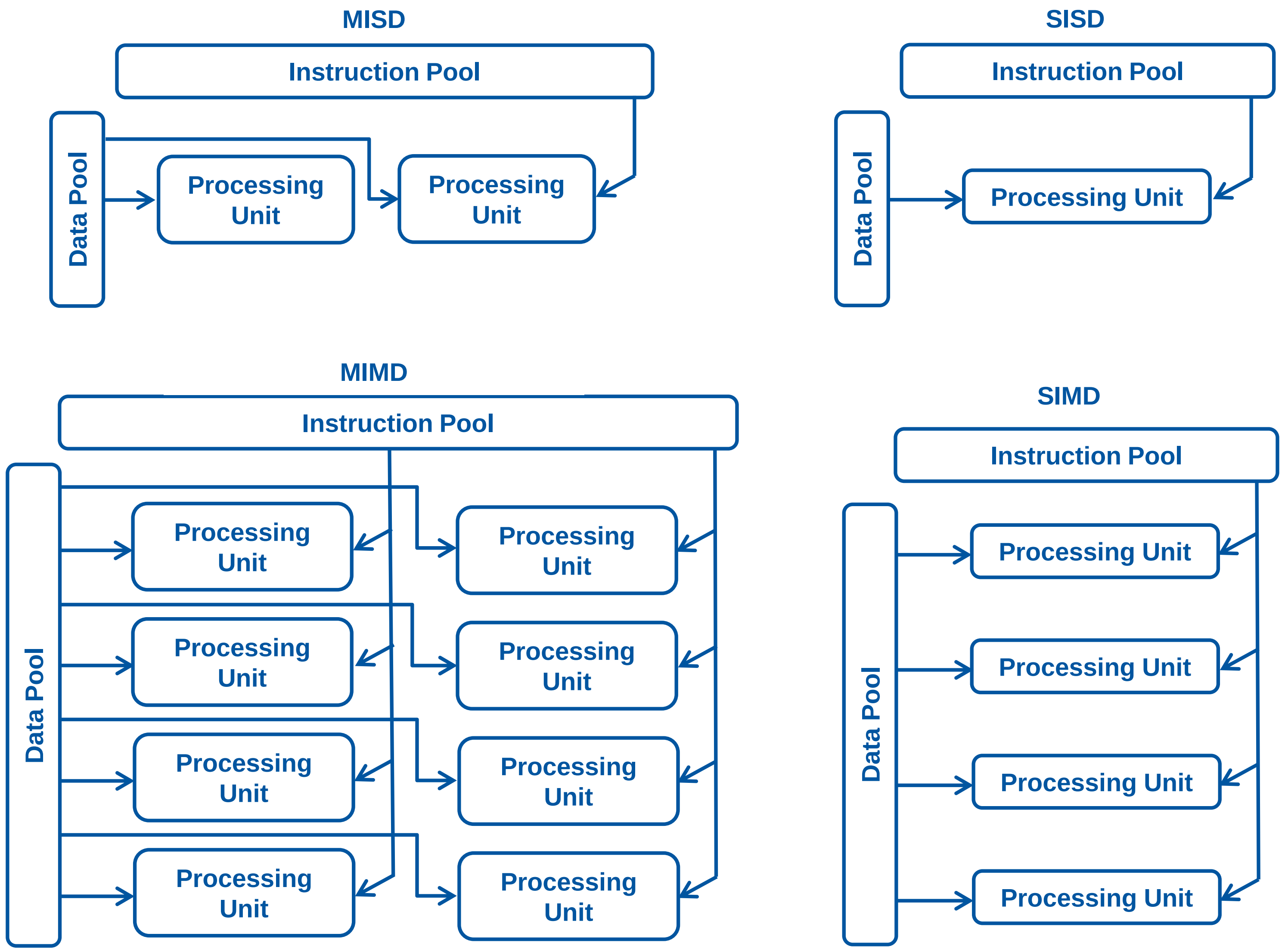
Can be implicitly generated by compiler understandable code or explicitly with intrinsic

Explicitly programmed using shared-memory parallel programming model

Explicitly programmed using distributed-memory parallel programming model

Flynn's taxonomy: <u>Can be</u> a programmer's guideline

- Proposed in 1966
- Instruction streams
  - single (SI) or multiple (MI)
- Data streams
  - single (SD) or multiple (MD)
- Types
  - SISD
  - SIMD
  - MISD
  - MIMD



Using SIMD

- Choose between code manageability and portability and speed:
- Use different levels of abstraction
  - Assembly
  - Intrinsic
  - Wrapper functions or classes in C or C++ (using intrinsics)
  - Custom languages like Cilk/Cilk++
  - Autovectorization

SIMD: SSE, AVX & FMA

- The use of SIMD instructions in vectorized code can give good performance gains

(S)SSE 1,2,3	SSE 4.1, 4.2	AVX	AVX2 + FMA	AVX-512
From 1999 Width 128b	From 2007 Width 128b	From 2011 Width 256b	From 2013 Width 256b	From 2016 Width 512b

- 256b -> 8 floats or 4 doubles (possibility of 4 or 8 speedup)

Fused Multiply Add

- These are instructions which can do calculations of the form:

```
A <- A*C + B or
A <- B*C + A
```

- Can bring gain because one instructions replaces a multiply and add instruction (reduces throughput cycles and latency)
- Is also a SIMD instruction

FMA and floating point

- FMA is specified to round only once
  - Therefore FMA is a change which can affect the result of a calculation compared to using separate multiply and add operations
- Enable with:
- GCC:
  - Will use FMA if it is compiling for an architecture that has it
    - May be explicitly set with –mfma or –mno-fma

Autovectorization

- Depends on compiler and version
- Advantages
  - Can get great speedups (x2 or more) with little change of the source code
  - Compiler can generate a report to help
  - Source code remains architecture independent
- Disadvantages
  - Can be delicate. A small change of the source can stop the compiler autovectorizing, causing a large change in performance
  - Lots of compiler options that affect the autovectorization
  - Usually you could have bigger gains using intrinsics or assembly
  - Some compiler options which help autovectorization can change the way FP operations are done: you have to be aware when that may be important

Autovectorization

- Two main places where automatic vectorization can be done
  - In loops
    - compiler tries to do several iterations of the loop at once using SIMD
    - May unroll a number of the loops so to fill pipeline and improve ILP
    - May peel loops to allow aligned access to data
  - Combining similar independent instructions into vector instructions
    - Known as SLP vectorizer

Autovectorization difficulties

```
for(i=0;i<*p;++i) {
  A[i] = B[i] * C[i];
  sum += A[i];
}
```

(example from slide by Georg Zitzlsberger, Intel)

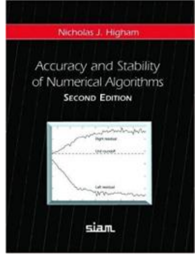
Possible to vectorize? Concerns may be:

- Is the loop range invariant during the loop
- Is A[] aliased with the other arrays or with sum, is sum aliased with B[] or C[]
- Is the + operator associative?
- Is the vectorized version expected to be faster?

Using autovectorization

- GCC:
  - Switch on using –ftree-vectorize
    - Off by default
  - Information on autovecotization analysis
    - using –ftree-vectorizer-verbose=X (X=0-7, 7 is most information)
    - Or examine generated code with gdb
  - May be necessary to use –ffast-math
    - Often with reductions (e.g. summations)
    - This will cause the compiler to relax certain some constrains, for example allow it to assume associative properties
- ICC:
  - On by default, modify with –x and -ax
    - Would switch off by using –no-vec
  - ICC defaults to ignoring parentheses to specify floating point associativity and generally can make more aggressive optimizations on floating point calculations
    - Controlled by -fprotect-parens and –fp-model
  - Information on autovectorization analysis
    - -qopt-report=2 –qopt-report-phase=vec
    - -opt-report-help and –opt-report-phase={hpo,ipo}
    - Or examine with gdb…

FP caution



- E.g. summing a number of values: One technique to reduce rounding error over long sums is Kahan summation

```
double sum = 0.0, C = 0.0, Y, T;
size_t i;
/* Kahan summation of values in A[i] */
for (i=0; i<length; i++) {
   Y = a[i] - C;
   T = sum + Y;
   C = (T-sum) - Y;
   sum = T;
}
```

- If the compiler reasons C=0 the correction is lost. Perhaps pairwise summation could be used instead.

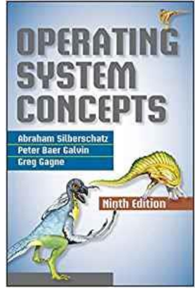
SoA vs AoS

- It is better to load the contents of a vector register from a contiguous piece of memory rather than gather the values
- Can arrange that the data layout fits this access pattern. e.g. use of Structure of Arrays instead of Array of Structures. e.g.

```
struct point {
      double x,y,z;
} location[1000];
```

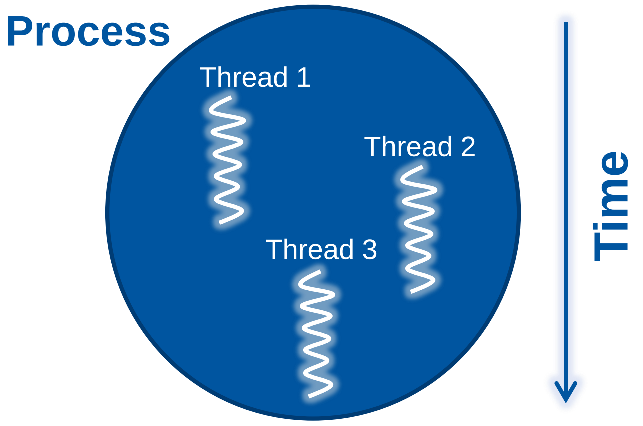
```
struct points {
      double x[1000];
      double y[1000];
      double z[1000];
} locations;
```

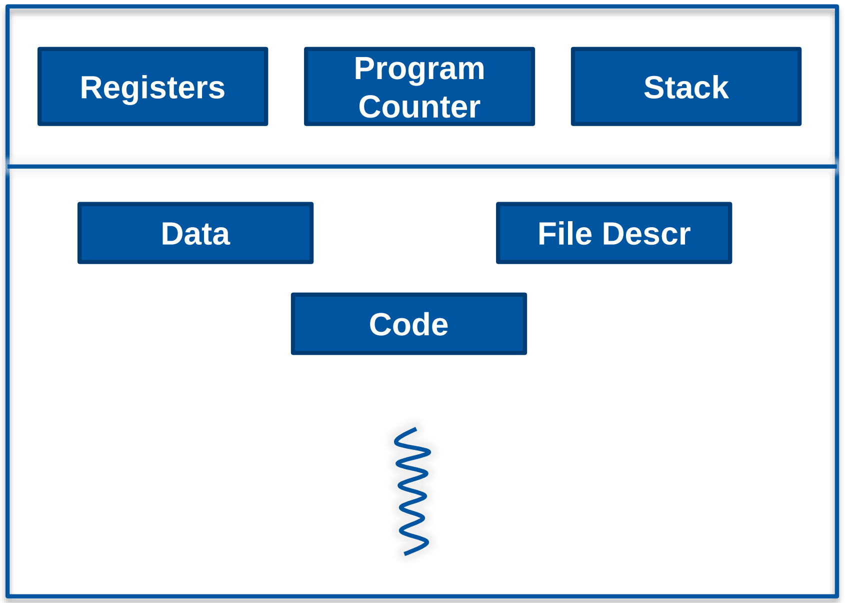
Software processes & threads

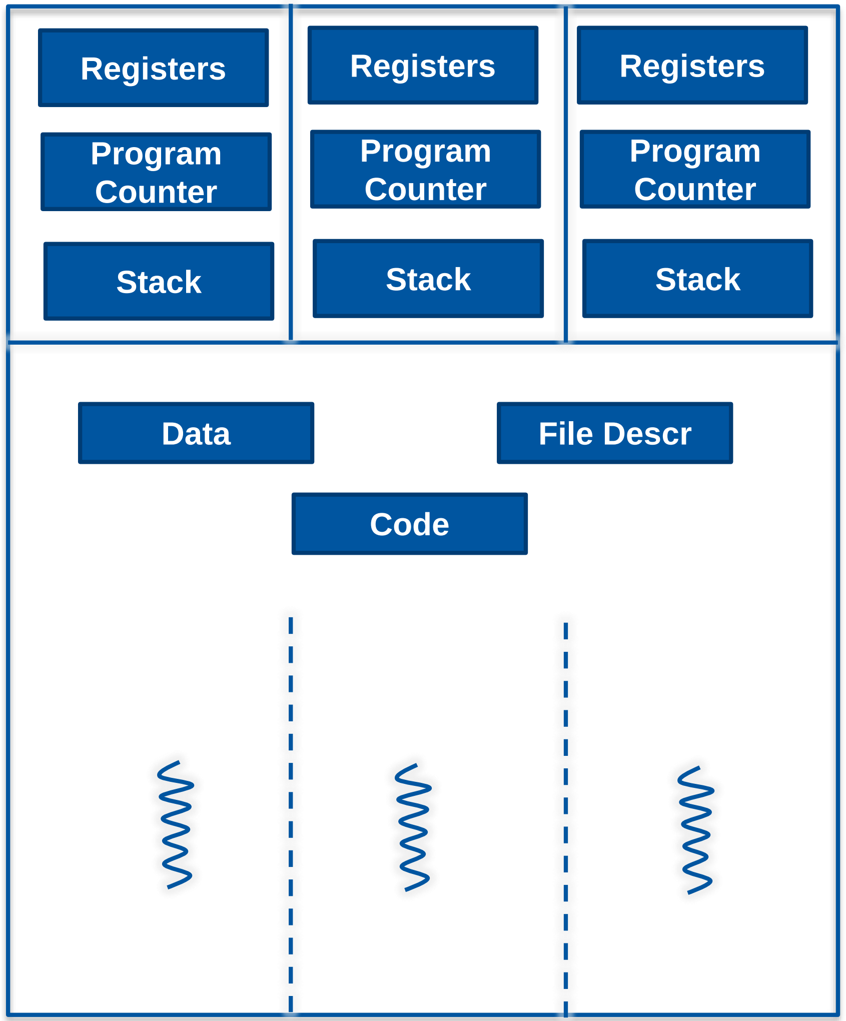


- Process (OS process):
  - Encapsulated entity of a program running in its own private address space
  - It consists of a private copy of program code and data along with file descriptors and permissions
  - It has a dedicated heap and stack space from which data is accessed and modified.
- Thread
  - Lightweight execution context that runs under a process
  - They share address space, program code and operating system resource of their parent process
  - Can be created and destroyed with low overhead in comparison to that of a process
  - Consists of a small amount of thread local storage space

Processes & threads







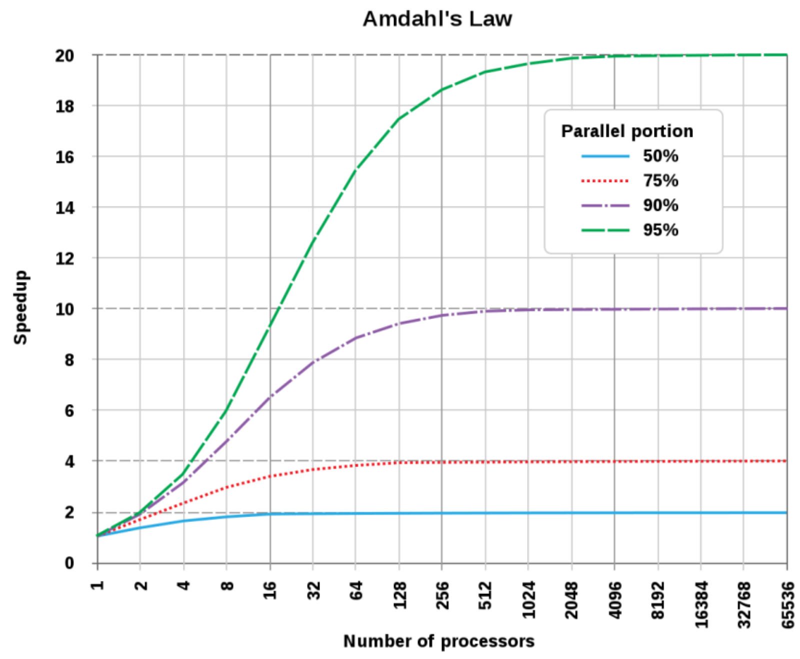
Parallel computing

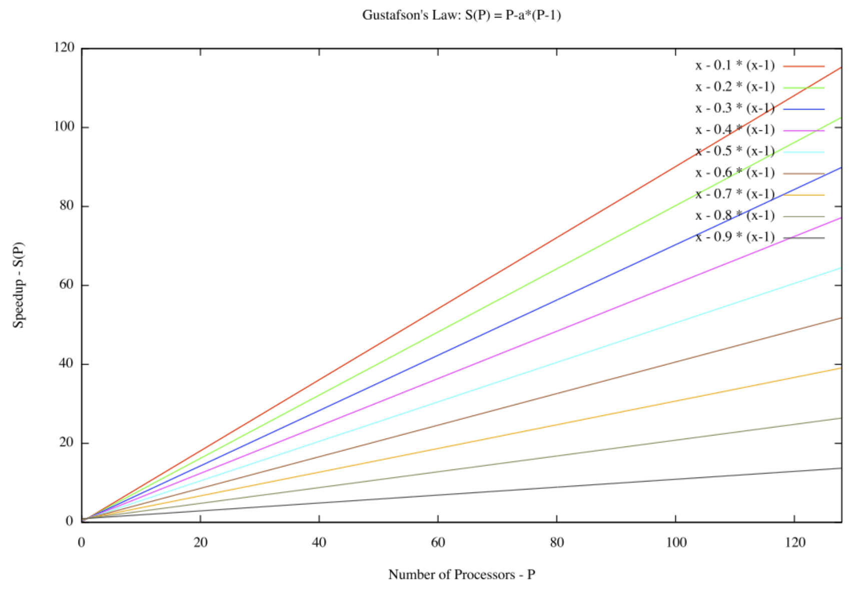
- Performing certain computations simultaneously using multiple resources
- **Amdahl's law** & **Gustafson's law**
  - Speedup only comes from the parallelizable part of the code
  - i.e. Serial part of the code will impact or limit the achievable performance

Amdahl's law
$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

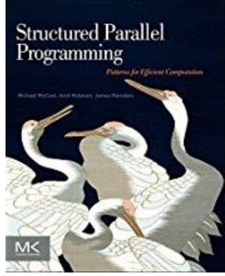
Gustafson's law
$$S_{\text{latency}}(s) = 1 - p + sp,$$

Where,
S_{latency} is the theoretical overall speedup
s is the speedup in the parallel part
p is the percentage of the execution time of serial part





Parallel Programming



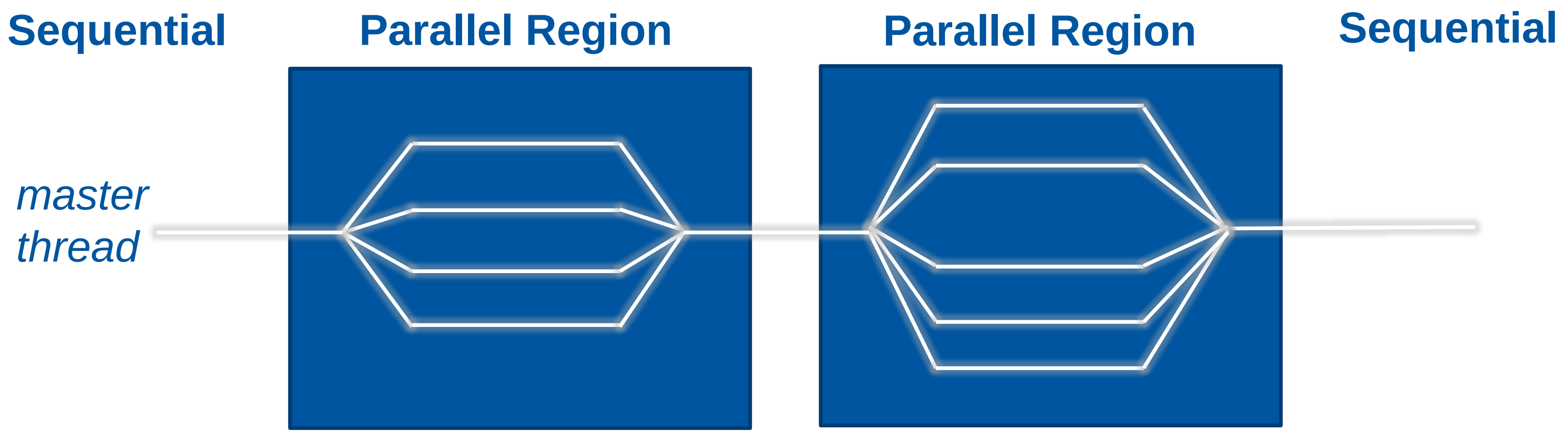
- Is a large topic. Many tools and techniques, a few:
  - pthreads is a standard API for managing threads
    - Fundamental API for threading in Linux
  - Cilk Plus
    - Language support by compiler extensions: appears as C/C++ with extensions
  - TBB (threading building blocks from Intel)
    - C++ large use of templates
    - commercial binary distribution with support or open source
  - C++11 threads
  - CUDA and OpenCL
    - GPU or CPU/GPU unified programming models
  - OpenMP
  - http://concurrency.web.cern.ch/GaudiHive
    - A framework from the HEP community

OpenMP

- A specification:
  - See https://www.openmp.org
  - Are compiler directives, routines and variables that can be used to specify high-level **parallelism** in C, C++ and Fortran
- GCC
  - 4.4 – OpenMP 3.0
  - 4.9 – OpenMP 4.0
  - 6.1 – OpenMP 4.5
- Clang
  - 3.7 – OpenMP 3.1
- Intel
  - 12 – OpenMP 3.1
  - 16 – OpenMP 4.0
  - 17 – OpenMP 4.5

OpenMP

- Code looks similar to a serial version
  - #pragma are used to indicate handling of parallel parts
  - Usually uses a fork-join model



OpenMP

- May need to compile with –fopenmp (check your compiler)

- Most OpenMP features are used through pragmas

```
#pragma omp construct [clause [clause] … ]
```

- You can change the number of threads via environment or an API or specify it in the pragma

```
export OMP_NUM_THREADS=16
```

Parallel regions

- Threads (up to the number configured) are created, if needed, when the pragma is crossed
- Threads execute the parallel region, the sequential part continues once all the threads have come to the end of the region
- Data is shared, but stack variables declared in the parallel region are private

```
#pragma omp parallel
{
     function_called_in_parallel();
}
function_sequential();
```

Parallel for-loops

- Loop iterations become threads
- Data is shared between threads (i.e. iterations), except loop index
- Threads wait at the end of the for loop
- The pragma is specified directly before the loop

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i < N; i++) {
    function(i);
  }
}
```

- The two pragmas above are equivalent to

```
#pragma omp parallel for
```

Sharing control

- Consider

```
double x, y;
#pragma omp parallel for
for(i=0;i<N;i++) {
  x = a[i]*4;
  y = b[i] * b[i];
  b[i] = x/y;
}
```

- This will probably not give the intended result: x and y are shared between the threads of the parallel for loop

The private clause

- Used to give each thread a private copy of a variable which was already declared outside
- The variable is uninitialized

```
double x, y;
#pragma omp parallel for private(x,y)
for(i=0;i<N;i++) {
  x = a[i]*4;
  y = b[i] * b[i];
  b[i] = x/y;
}
```

Variations on sharing control

- As well as *private*:
  - *firstprivate*: initializes each private copy to the value from the master thread
  - *lastprivate*: copies the value from the thread, which executed the last iteration of the loop, to the master thread
  - *shared*: is the default, but for documentation or if the default is changed you can uses this clause
  - Plus others, e.g. those which concern threadprivate variables

Exercise 4 – Use SIMD in the matrix multiplication

- Starting with the blocked version of the matrix multiplication see if autovectorization has an effect
  - The Makefile already has the matrixmul-simd target.
  - Autovectorization may work or might need a small change
    - Compare the execution time to a similar multiplication not using autovectorization
    - See if you have SIMD instructions (you may use Intel SDE, see next slide)
    - Check the compiler report if it didn't work

Intel SDE

- Is the software development emulator
  - In this case we can use it to count and classify different types of instructions

```
export PATH=/home/gss2018/exercises/sde-external-8.16.0-2018-01-30-lin:$PATH
sde -iform 1 -omix test.out -top_blocks 5000 -- ./my_executable
```

- Look in test.out for lines ending in _1, _2 or _4 representing scalar or packed operations, e.g.

```
 cat test.out | egrep '^\*.*_[124]'
```

Exercise 5 - OpenMP

- Use OpenMP to make the matrix multiplication from the previous exercise use multiple threads
  - Set OMP_NUM_THREADS=6, 12, 24, 36 and then 48
  - Run a multiplication and use top to look at the running process. Note that %CPU should be >100%.
  - Compare the runtime each time and
  - Use perf to measure the instructions and cycles; vary the number of threads and note how each changes

CERN