
CERN

Behind the scenes perspective: Into the abyss of profiling for performance

Part I

Servesh Muralidharan & David Smith
IT-DI WLCG-UP
CERN
29 Aug'18

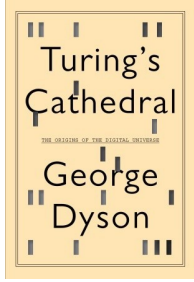
Format of Workshop

- Essentials (45 minutes)
  - Program and Computer Architecture
  - Parallelism
  - Compilers
  - Profiling and benchmarking
- Exercises: Matrix multiplication
- Hands on (90 minutes)
- Break (30 minutes)
- Exploiting parallelism (45 minutes)
  - SIMD
  - OpenMP
- Hands on (90 minutes)

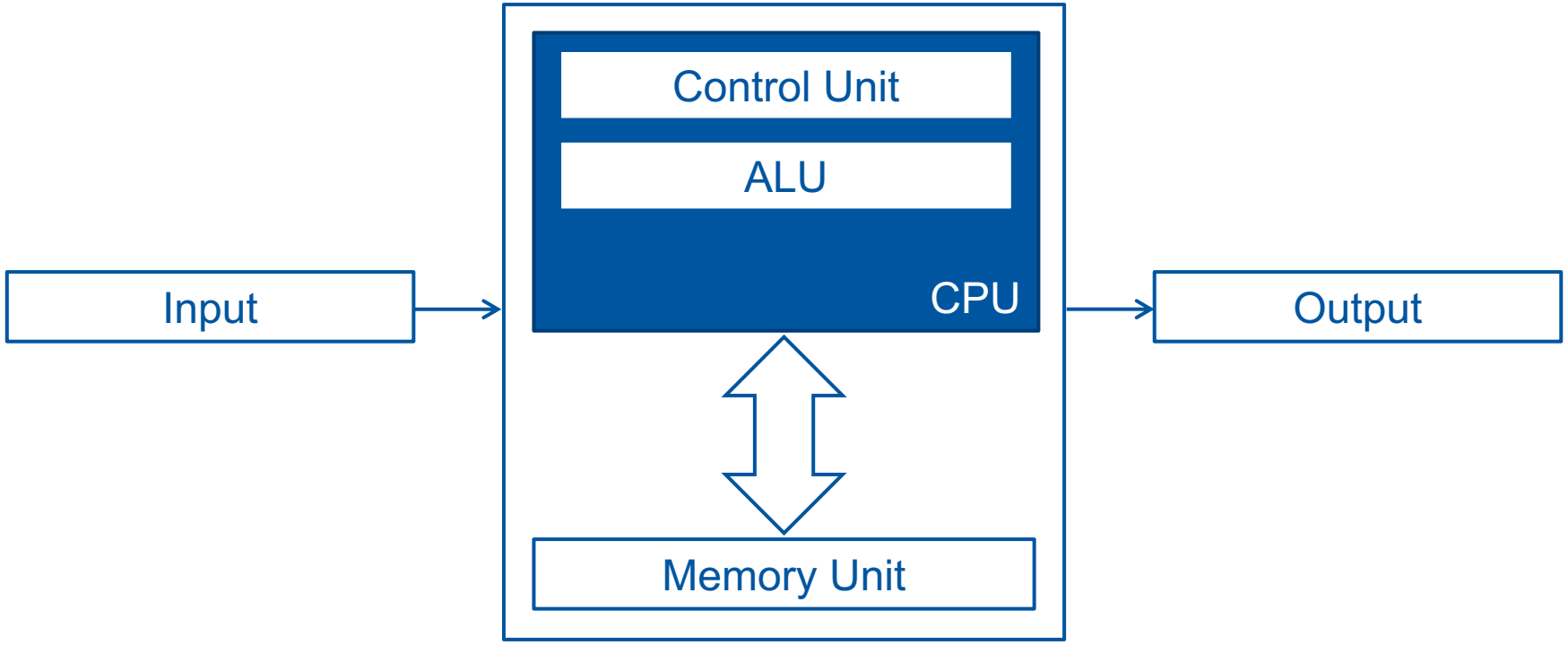
Concept of an Ideal Program

- 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

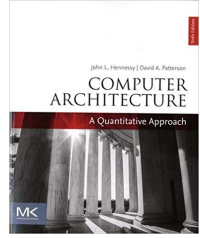
Essentials: Computer Architecture



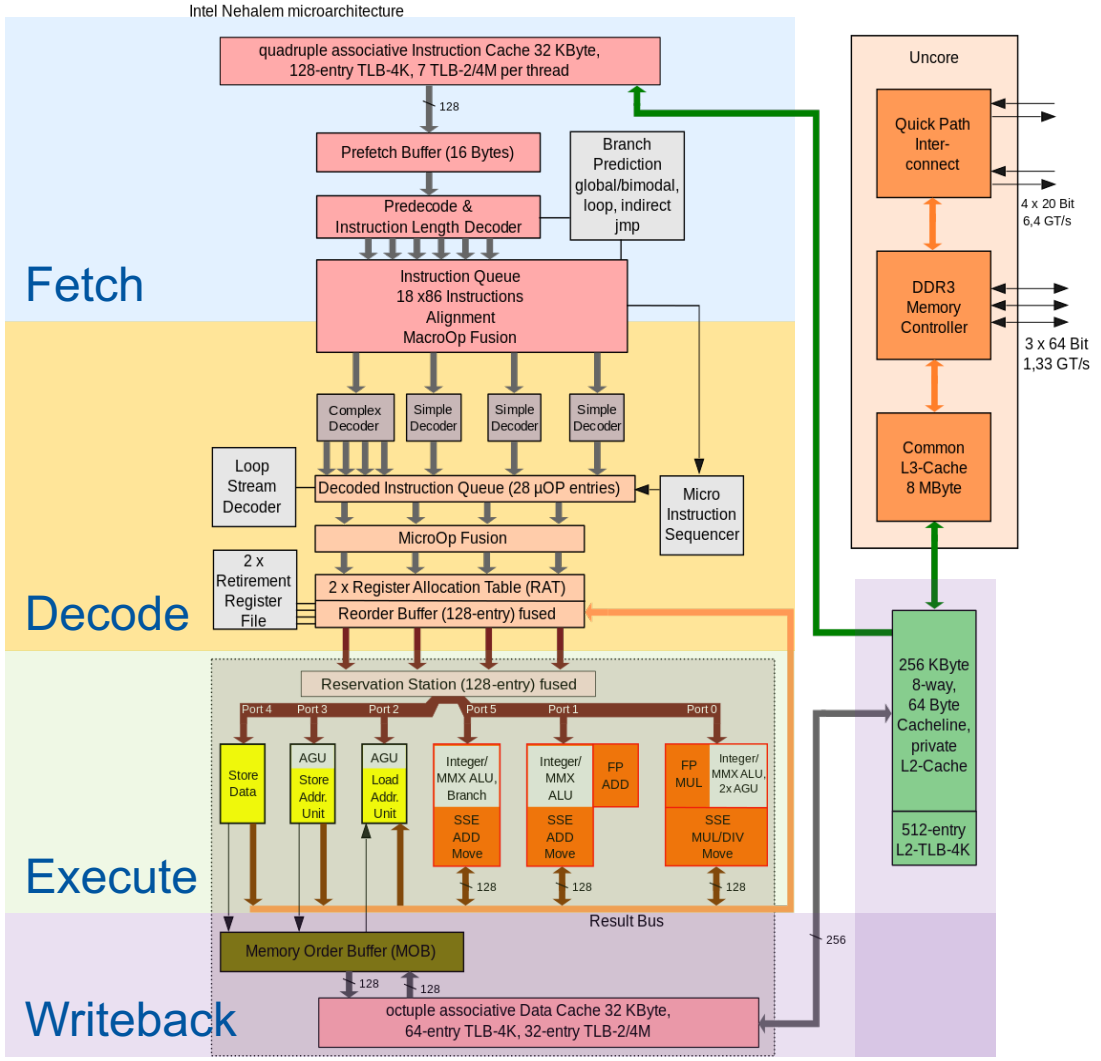
- Stored-Program computer
  - Von Neumann model
  - Program and data are stored in memory and then processed



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)

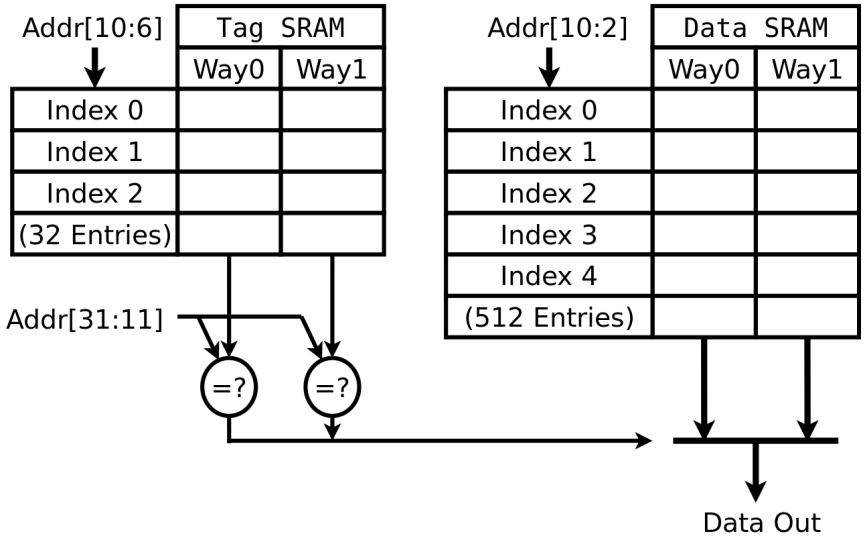


GT/s: gigatransfers per second

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

Cache lines

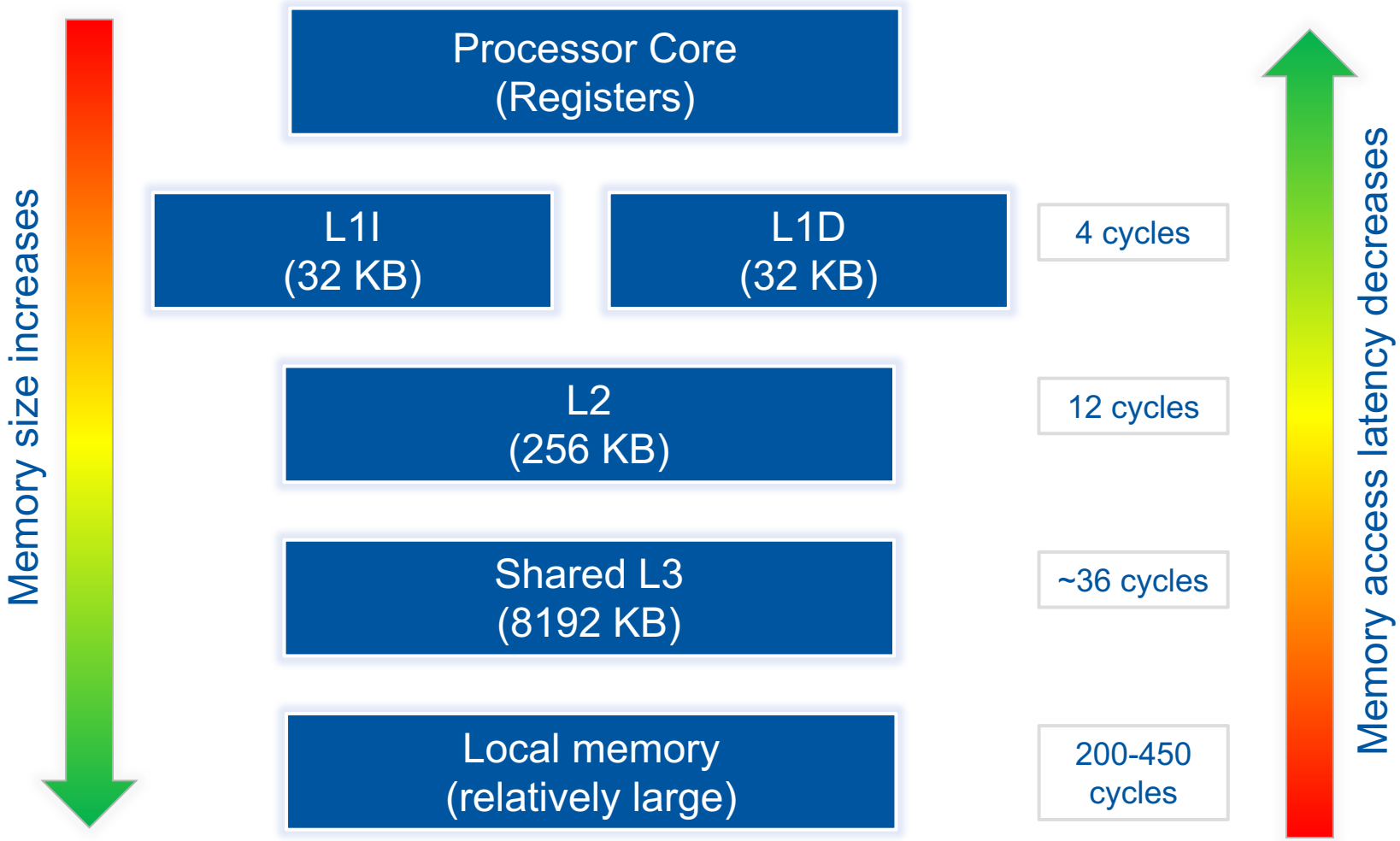
- Data is always fetched in units of a cache line
  - On x86 cache line size is 64 bytes
  - Usually data read from main memory still be stored in the cache, so even if you need less than the size of a cache line 64 bytes will be fetched
  - Caches have to be kept consistent if the same cache line is 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

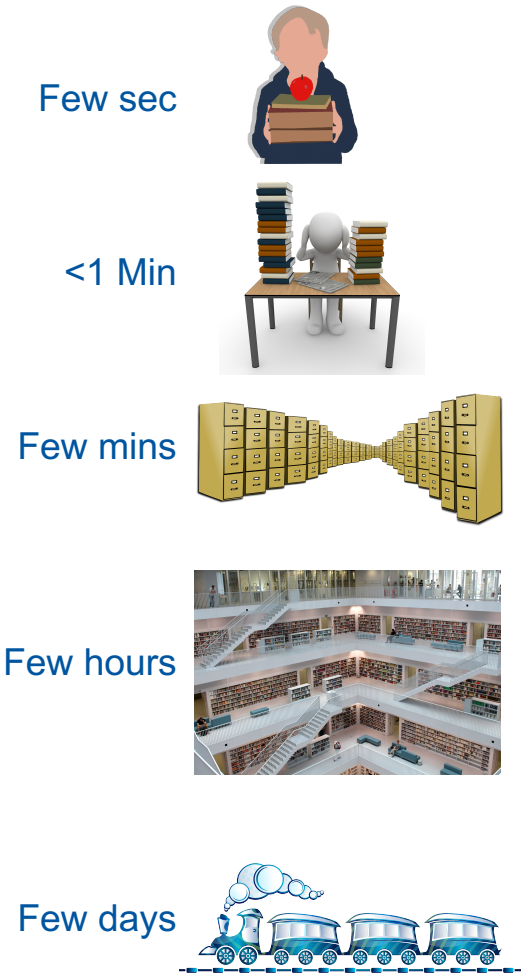
https://commons.wikimedia.org/wiki/File:Cache,associative-read.svg

Memory Hierarchy



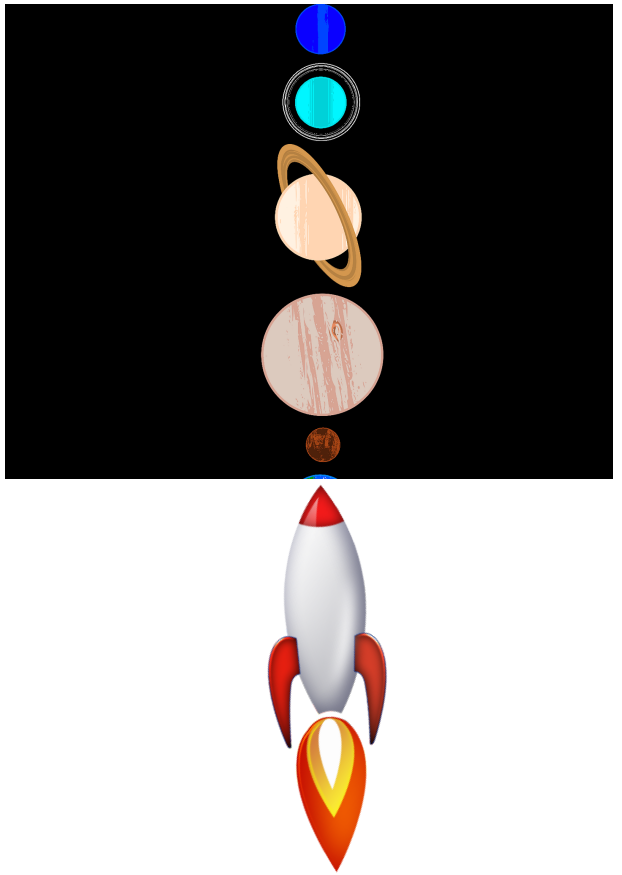
Approximate memory latencies on Intel Haswell CPUs

Adapted from S. Jarp, A. Nowak



Memory and Disk

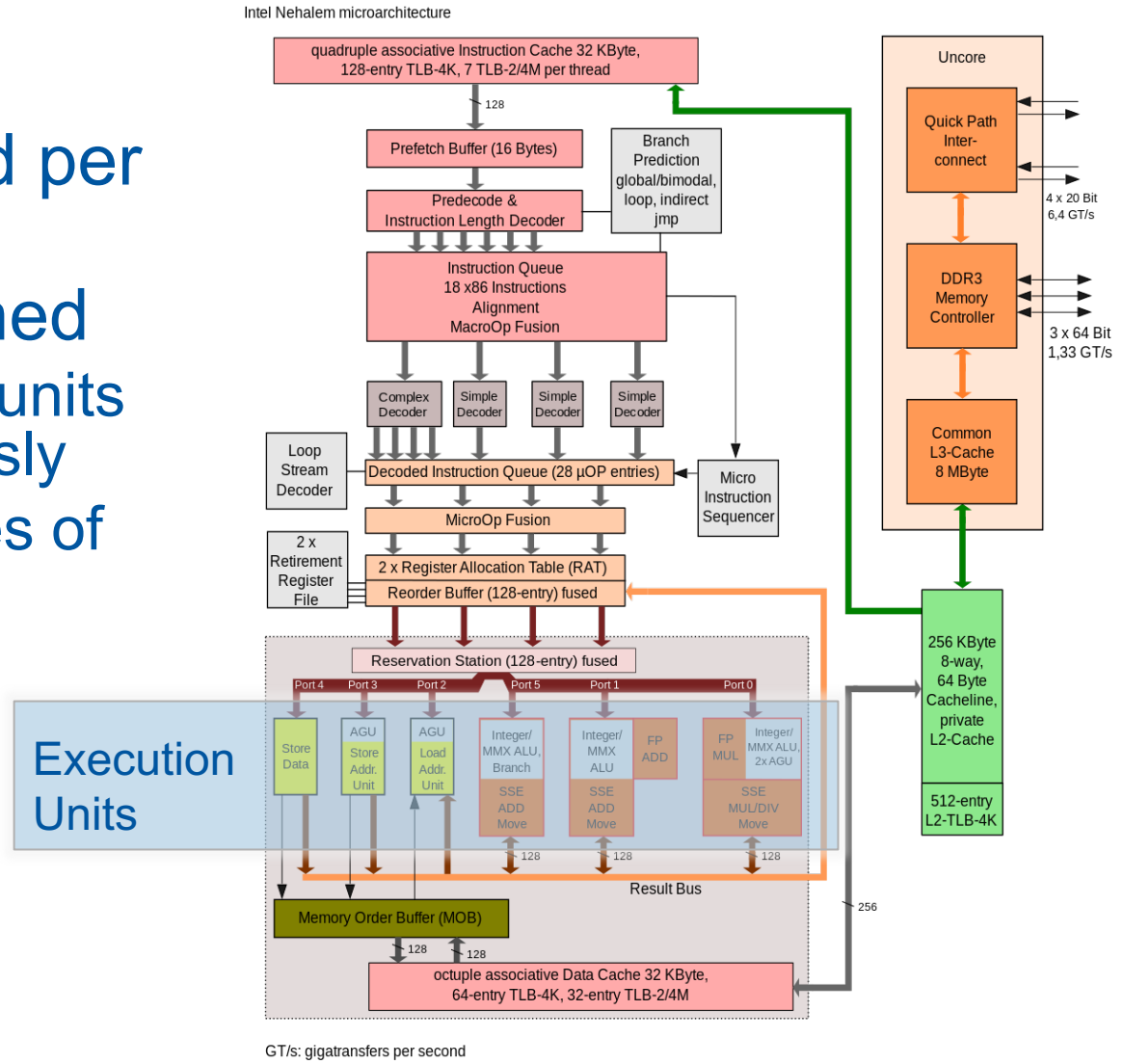
- Today we are concerned mostly with main memory (RAM) when talking storage outside the processor
  - Typically 1 to 100s of Gigabytes in size
- However often data will be on a storage device like:
  - Object storage, Disks, SSDs, NVMe devices
  - These will have latencies from 50 to +100,000 times longer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



- Essentials (45 minutes)
  - Program and Computer Architecture
  - **Parallelism**
  - Compilers
  - Profiling and benchmarking
- Exercises: Matrix multiplication
- Hands on (90 minutes)
- Break (30 minutes)
- Exploiting parallelism (45 minutes)
  - SIMD
  - OpenMP
- Hands on (90 minutes)

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*



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

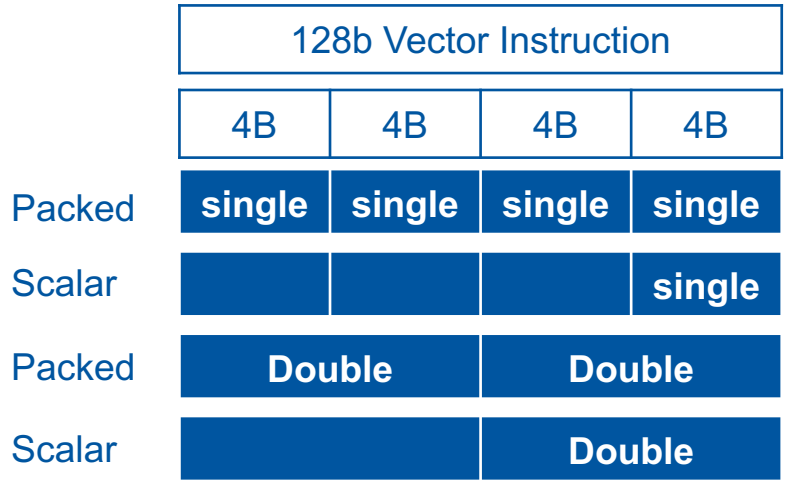
Instruction	1				2			
Fetch	X				X			
Decode		X				X		
Execute			X				X	
Write				X				X
Clock	1	2	3	4	5	6	7	8

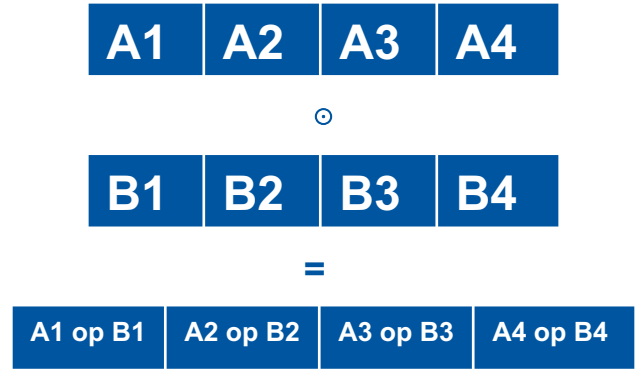
With pipelining approach

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

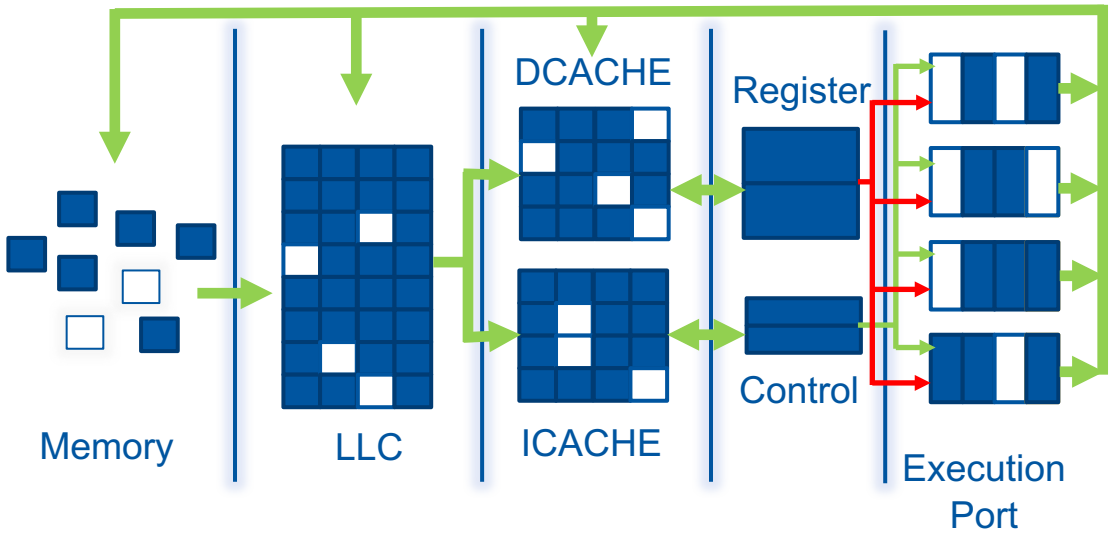
- There are vector registers
  - e.g. in processors with Intel AVX there are registers YMM0 – YMM15 which are 256 bits in length
- Example of one instruction that operates on multiple data
- Data may be placed into these in various ways
  - Scalar (just some of the width for one value)
  - Packed (a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

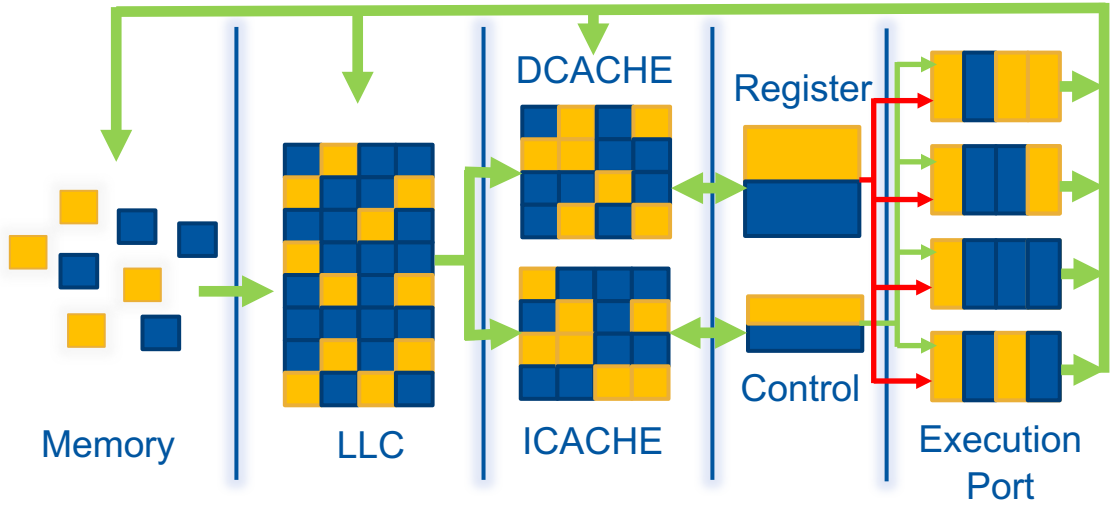




4th form of HW Parallelism: Hardware threads A.K.A Resource Multiplexing

- 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)



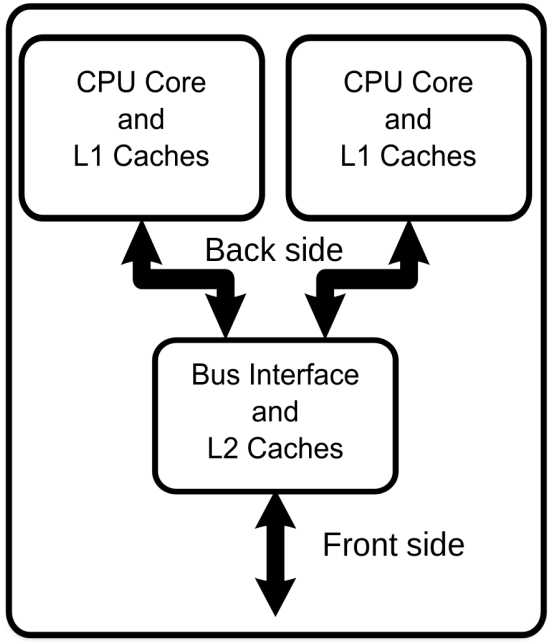


HW Thread 2

HW Thread 1

5th form of HW Parallelism: Multicore

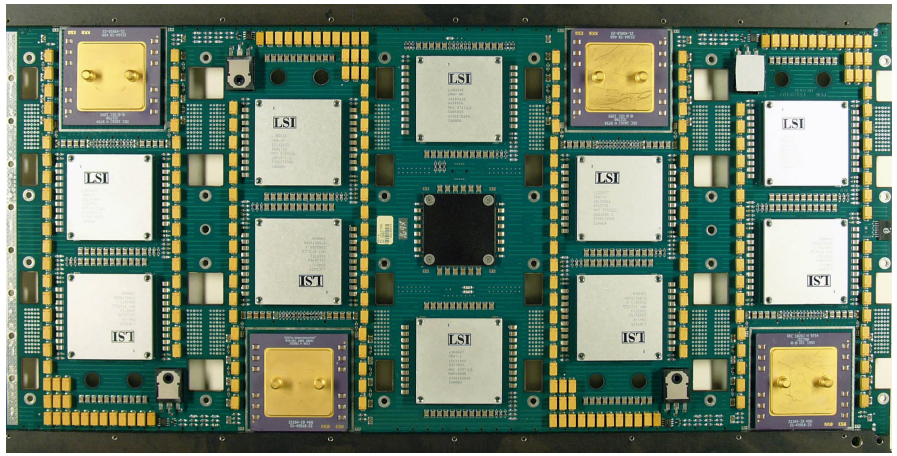
- Core
  - The execution logic, cache, and facilities for storing execution state. e.g. register files



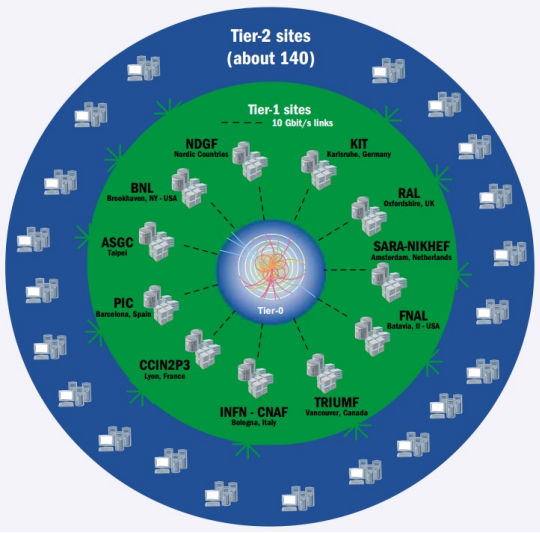
https://commons.wikimedia.org/wiki/File:Dual_Core_Generic.svg

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

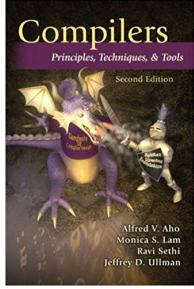


https://commons.wikimedia.org/wiki/File:Processor_board_cray-2_hg.jpg

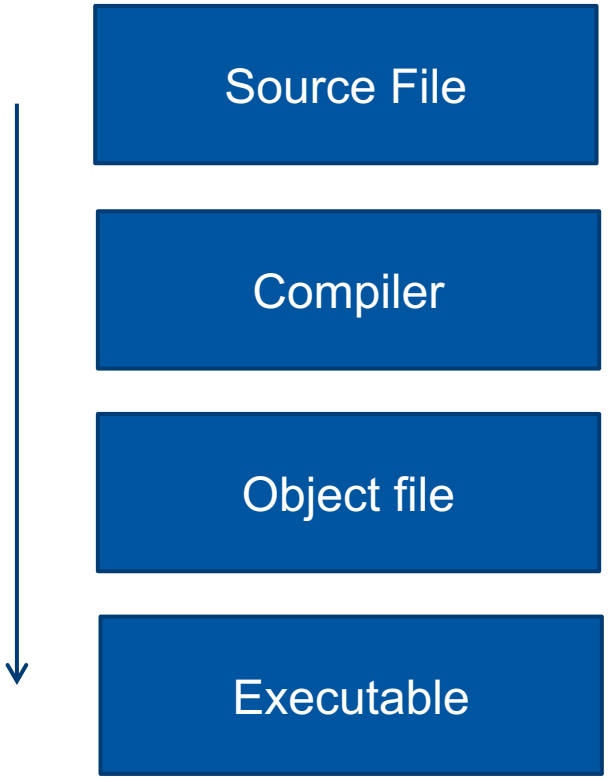


https://sciencenode.org/feature/large-hadron-colliders-worldwide-computer.php

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
- $N = (-1)^s \times 1.ccc\ldots \times b^{qqq\cdots}$
  - 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:

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

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

Where

$\oplus$ denotes floating point addition,

$\otimes$ denotes floating point multiplication

Matrix Multiplication

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

$$C=AB=\begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{np} \end{bmatrix} \qquad c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Why Matrix Multiplication???

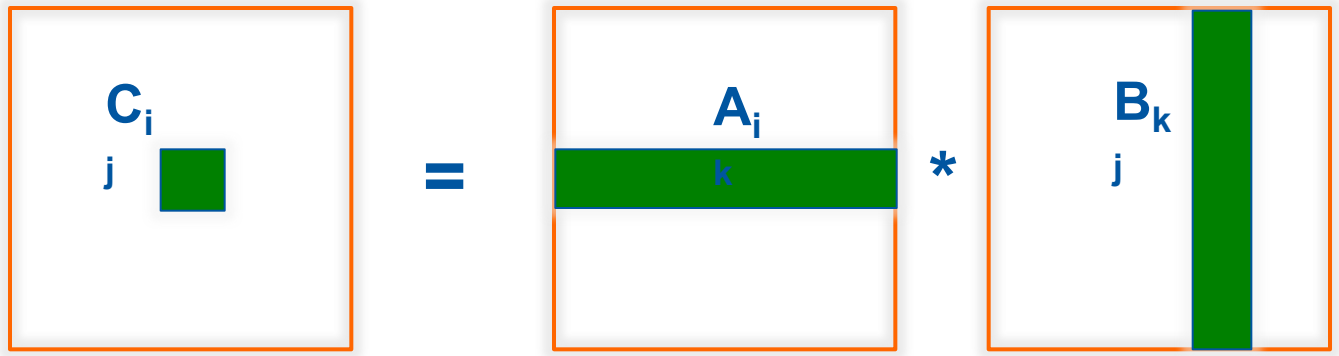
- Matrix multiplication is fundamental to large numerical computations
  - Large math problems can be decomposed into linear equations and then solved using MM
  - Motivation behind the development of high performance math libraries such as BLAS, MKL, etc./
  - LINPACK benchmark used to rank Supercomputers 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!!!)

CERN

First Attempt

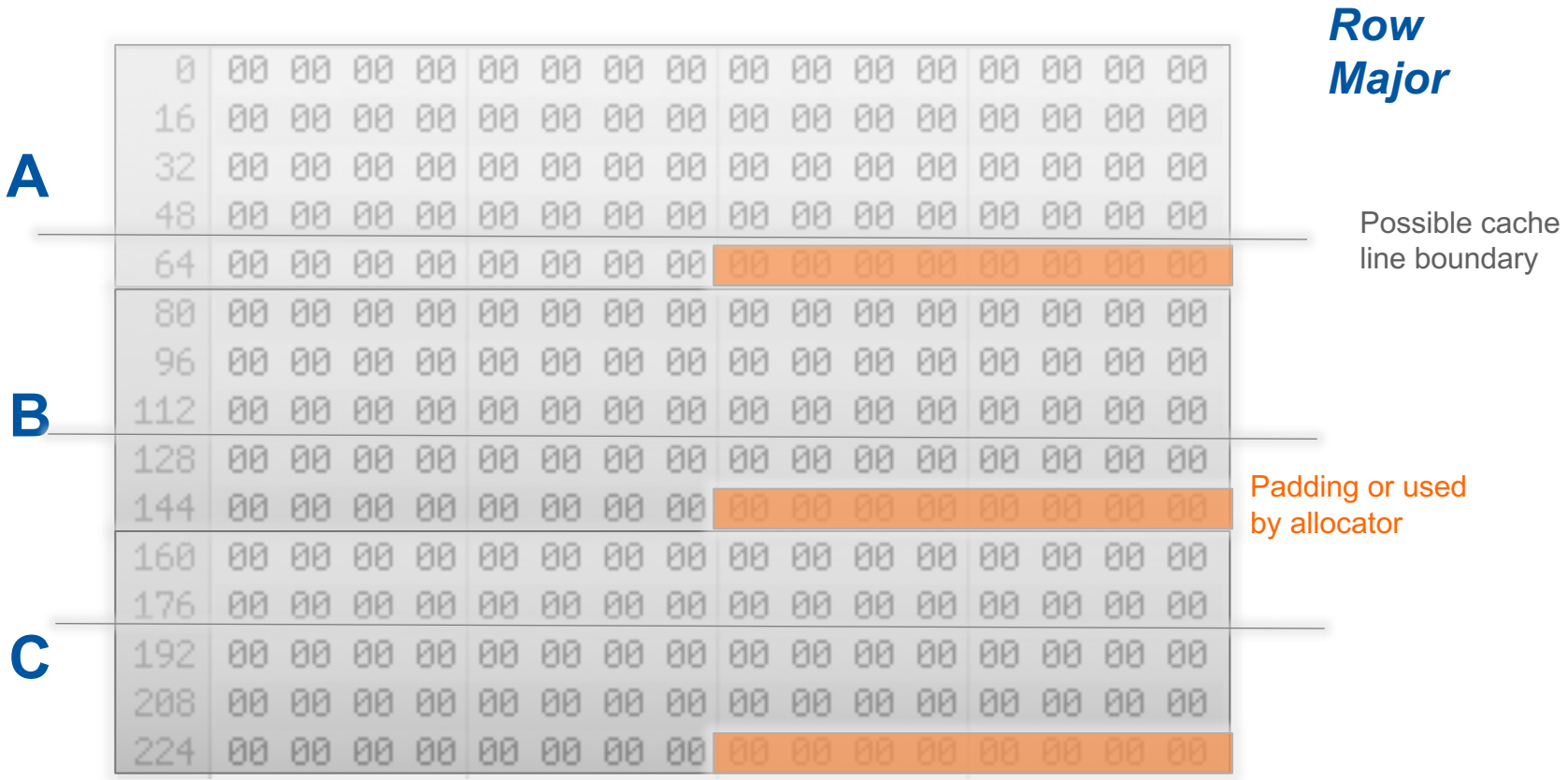
$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

```
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;
  }
}
```

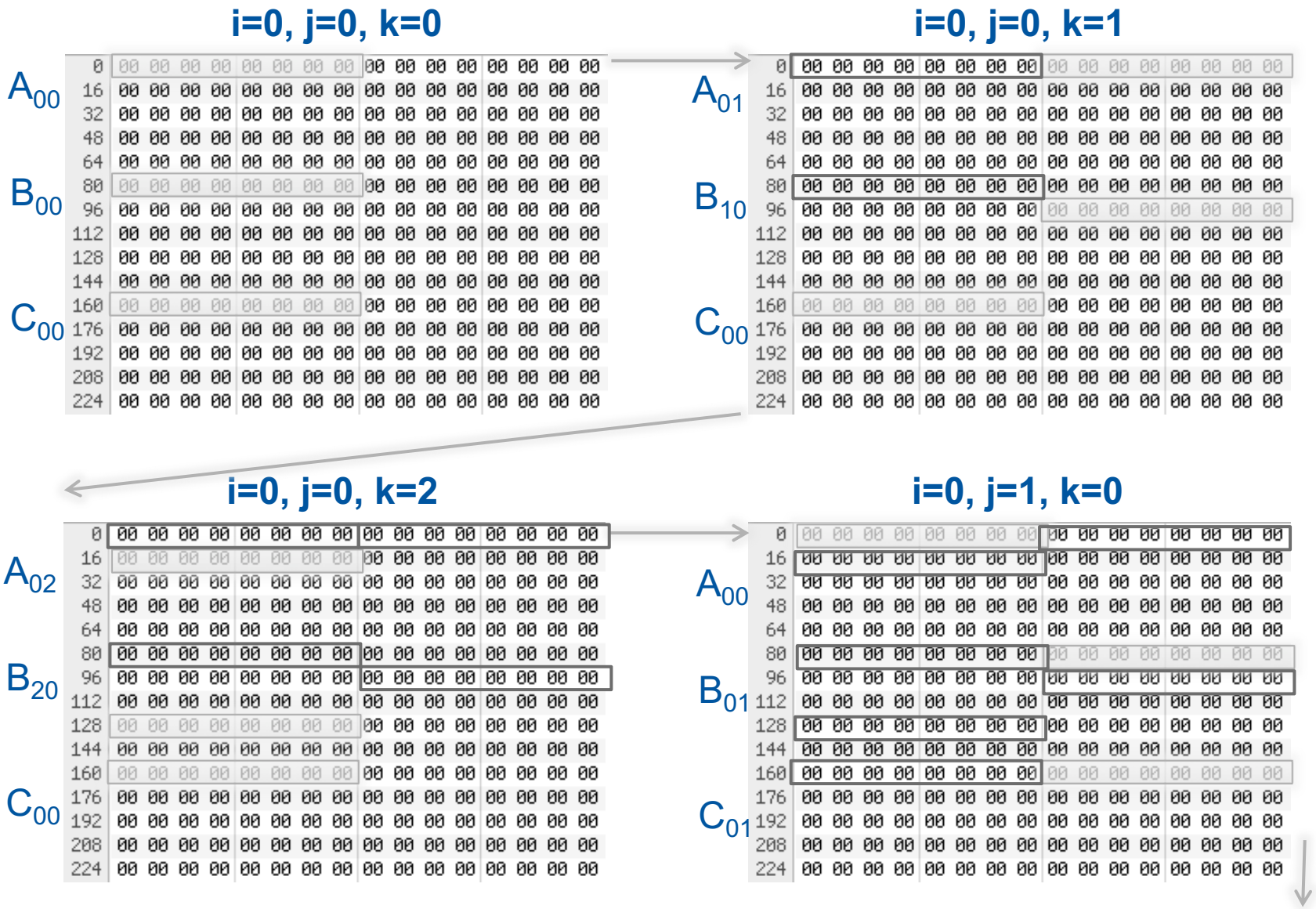


Possible memory layout

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



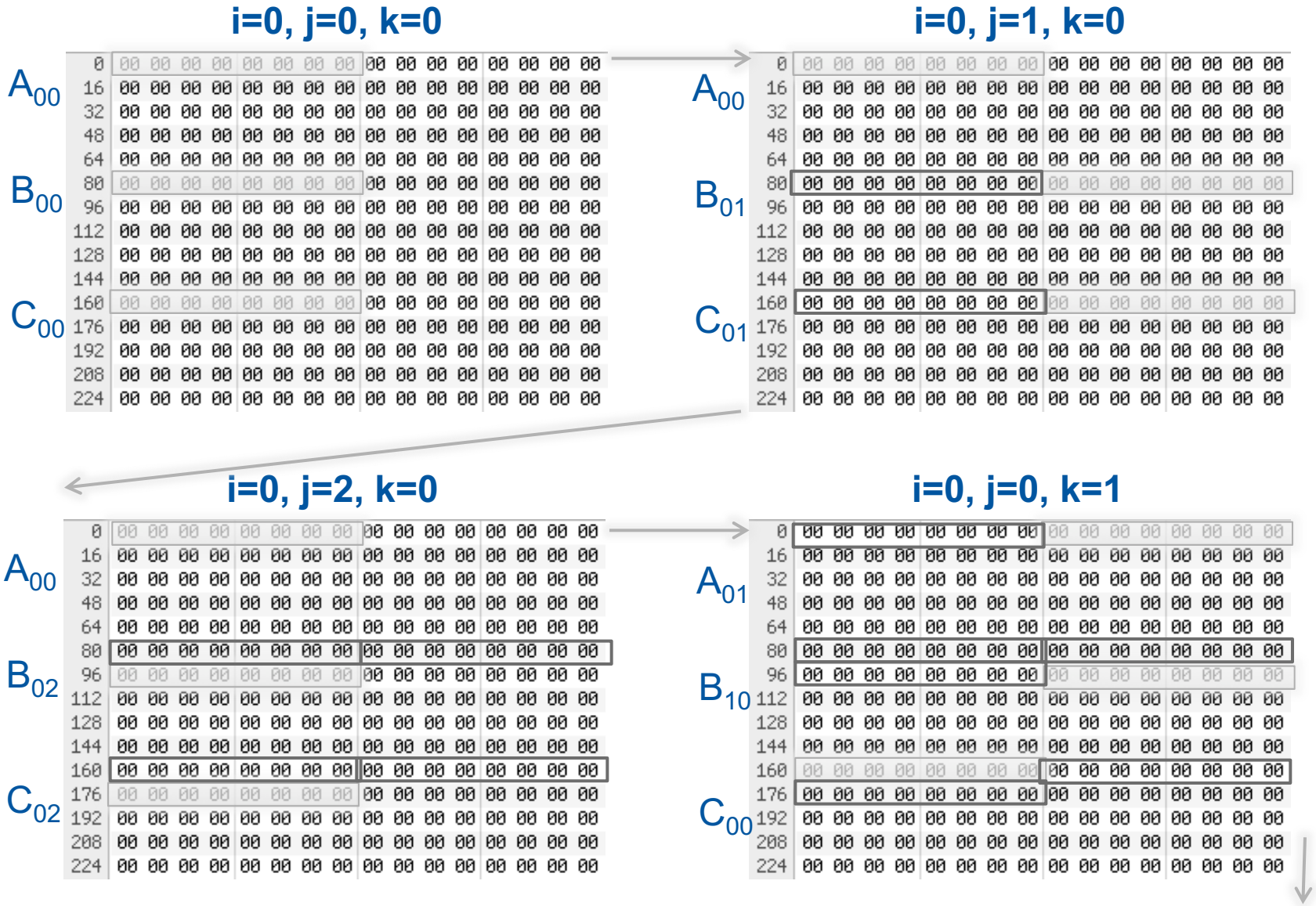
Memory access pattern



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



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

Exercises

- Connect to machines at CERN to run the exercises: user name is *gss2018*

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

Luca Atzori	Olof Barring	Ian Bird
Maria Girone	Stefan Lueders	Alberto Di Meglio
Guillermo Izquierdo Moreno		Wayne Salter
Markus Schulz	Hannah Short	Bernd Panzer-Steindel

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

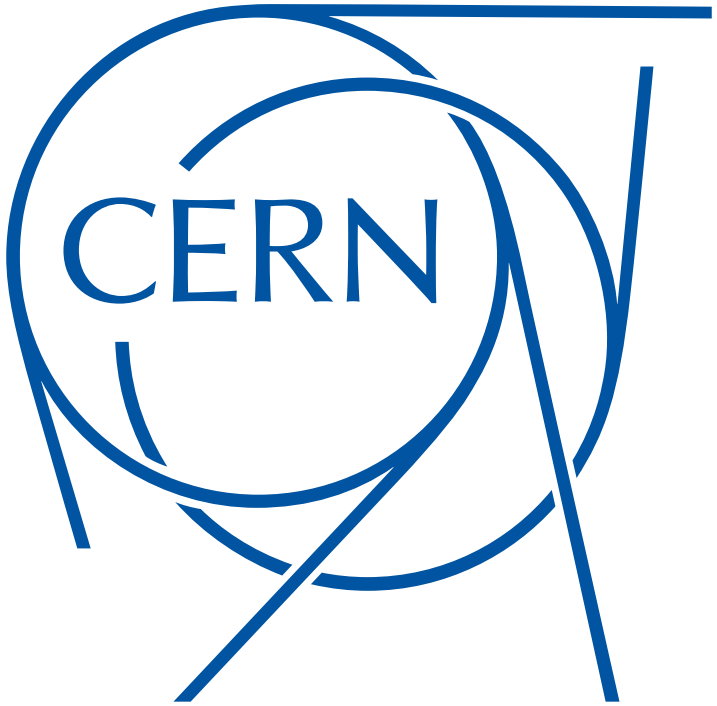
- Last level cache: relevant perf event names LLC-load-misses, LLC-prefetch-misses and for totals LLC-loads, LLC-prefetches

```
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
```


CERN

- olsnbb03
- olsnbb05
- olsnbb08
- olsnbb09
- olsnbb10
- olsnbb11
- olsnbb16
- olsnbb17
- olsnbc03
- olsnbc04
- olsnbc06