universität innsbruck

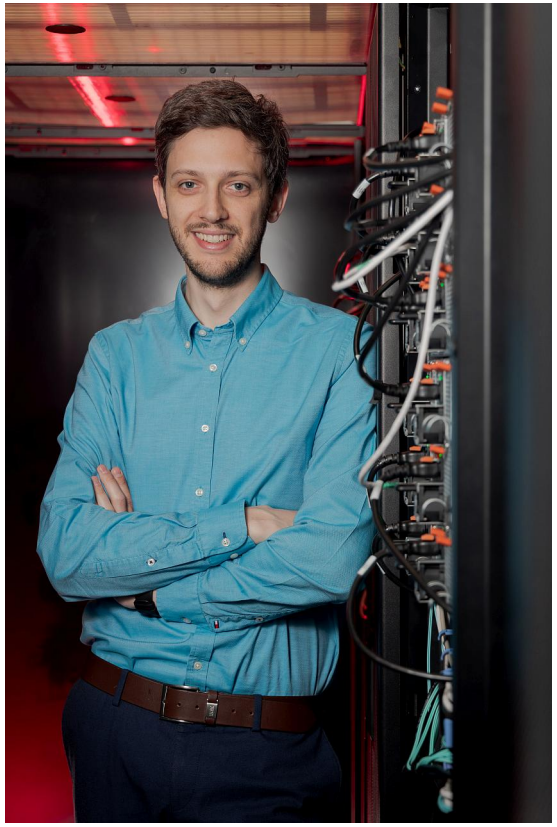
FZ HPC

1st Summer School in HPC and AI (UniBZ, June 2021)
Introduction & A Crash Course in Modern Hardware

Philipp Gschwandtner, Research Center HPC, University of Innsbruck
with special thanks to Lukas Einkemmer and Rolf Rabenseifner on whose original slide sets parts of this course are based

Who is This Instructor?

- Senior Scientist at Research Center HPC, University of Innsbruck (https://dps.uibk.ac.at/~philipp)
  - studied computer science, focus on parallel programming, benchmarking and tuning
  - aid researchers at UIBK in developing and optimizing parallel applications
- research interests in and around HPC
  - measurement/optimization/modeling of performance, energy, efficiency, …
  - APIs, programming models, runtime systems, compilers, …
- collaboration with other researchers and universities, such as UniBZ



Extremely professional-looking photo of Philipp Gschwandtner

What are we Going to Discuss Today?

- motivation
  - Why do we need to know any details about hardware?
  - Why use OpenMP at all?

- main characteristics
  - Hardware: CPUs, caches, bottlenecks
  - OpenMP: parallelism, programming model, worksharing, synchronization, environment, …

What This Course is (not) Intended for

- my intentions are NOT to
  - give long specifications
  - show full-blown applications
  - discuss a level of detail that cannot be processed in a single day

- my intentions are to
  - place OpenMP in the parallel programming and HPC landscape
  - outline its advantages and drawbacks
  - discuss its most basic and important building blocks for assembling parallel applications and **work on them in practical exercises**
  - give you enough input to be able to research additional information yourself

Approximate Schedule

- 08:45 Join online
- **09:00 Welcome**
- **09:10 Introduction to modern hardware (talk)**
- **10:10 An overview of OpenMP (talk)**
- 10:35 Coffee
- **10:50 OpenMP programming and execution model (talk+practical)**
- 12:30 Lunch
- **13:30 OpenMP worksharing directives (talk+practical)**
- 15:00 Coffee
- **15:15 More OpenMP (talk+practical)**
- **16:00 Summary + Q & A**

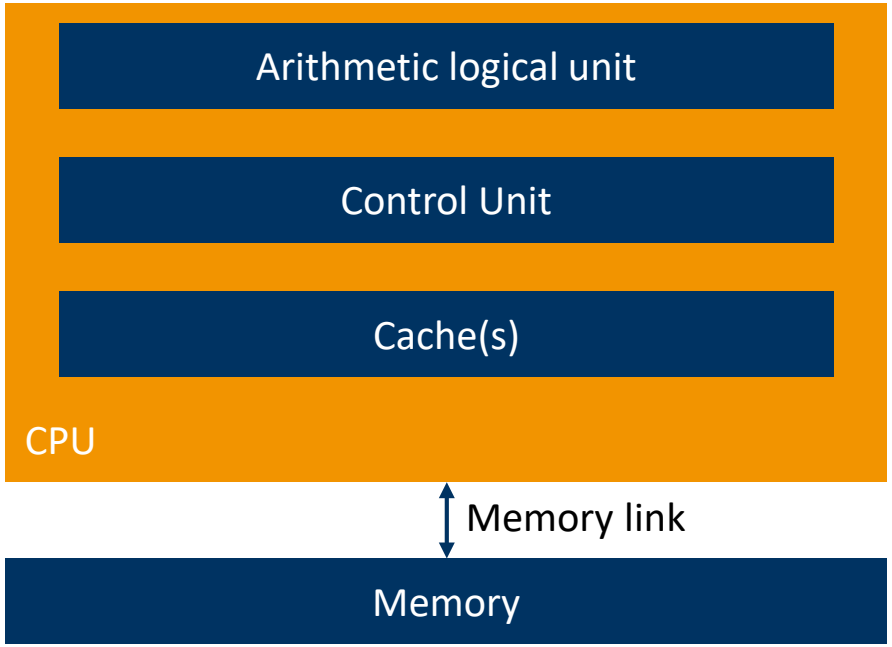
A Crash Course in Modern Hardware

Motivation

- Understanding **hardware** is important to understand **performance**.

Components of a Computer

- CPU (central processing unit)
  - performs arithmetic operations, conditionals, loops, etc.
- Memory
  - stores data used for processing (main memory, caches, permanent storage, etc.)



CPU

- The CPU executes a sequence of instructions (referred to as machine code)
  - Simplified example of assembly/machine code on the right

```
mov 0x38AF2 r1
mov 0xA03DD r2
add r1 r2
mov r2 0x38AF2
```

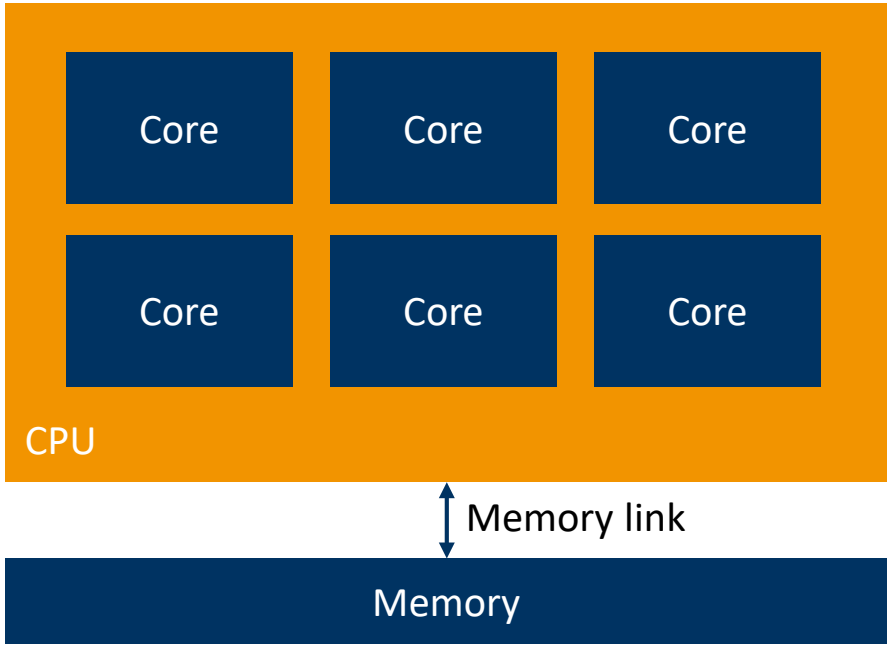
- Instructions can be grouped as follows
  - Memory instructions (write or read from main memory)
  - Arithmetic operations on registers
  - Control instructions (comparisons, jumps)

CPU Performance

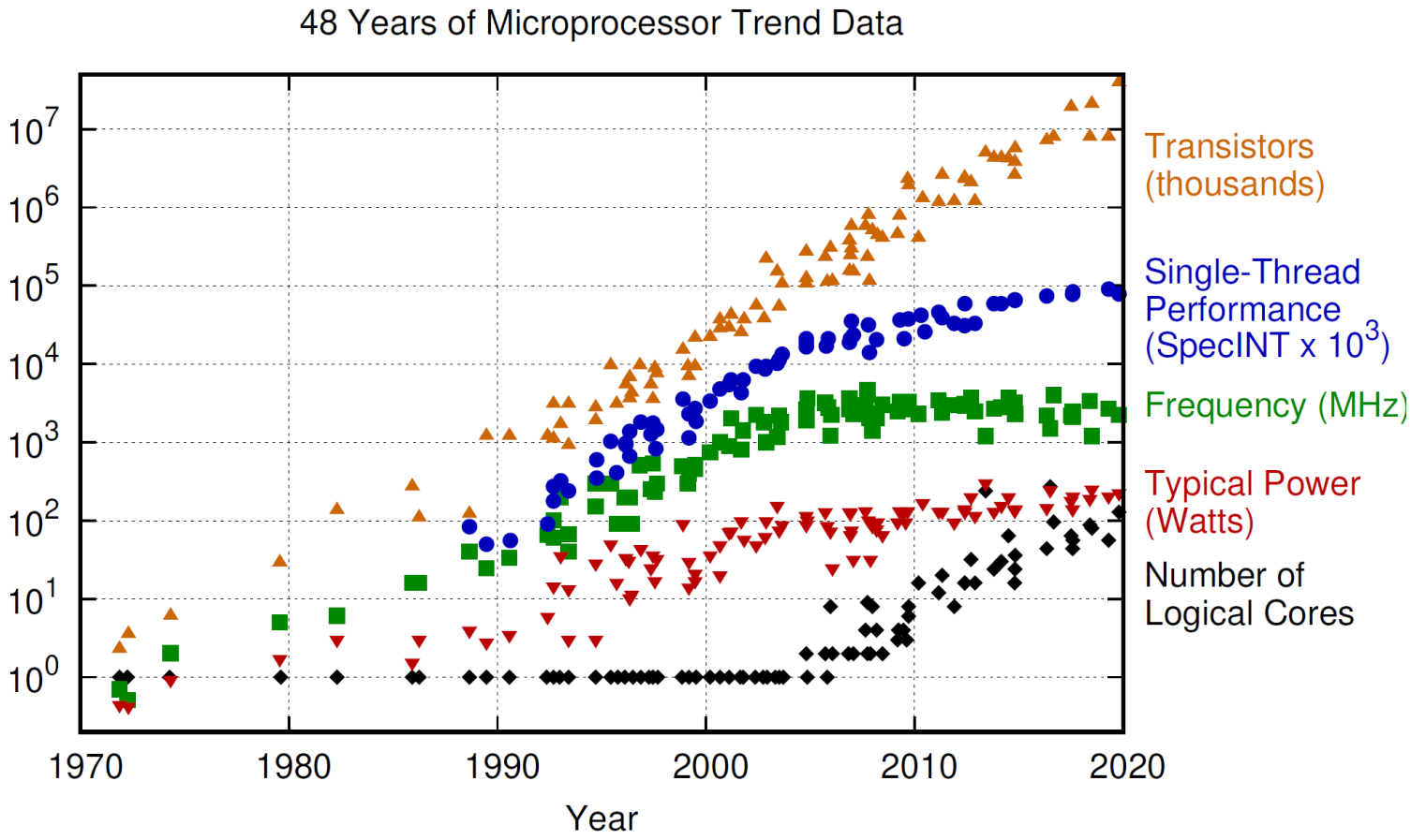
- Performance is usually measured in floating point operations per second (FLOPS)
  - Amount of arithmetic operations that can (theoretically) be performed per second
- A 3 GHz CPU that can perform one floating point operation per clock cycle equals 3 GFLOPS.
- To attain this level of performance might not be possible in practice
  - Algorithm-dependent
  - Implementation-dependent

Modern CPUs

- Increasing clock frequency has not been viable for a while
  - Power dissipation scales as a square/cubic function of the frequency
  - But transistors get smaller
- Modern CPUs look like the illustration on the right
- Multiple cores on the same chip
  - Usually the sequential execution units are referred to as cores
  - Modern CPUs easily hold 8-64 cores



Modern CPUs cont'd



Vectorization

- Each core can perform vector operations in a single clock cycle
  - e.g. 256 bit registers: 4 double, 8 floats
  - Fused multiply-add
- Vectorization is only possible if the CPU supports that specific instruction
  - Ideally handled by the compiler
  - OpenMP includes support for vectorization
- Types of parallelization
  - Vectorization is single instruction multiple data (SIMD)
  - Core-level parallelism is multiple instruction multiple data (MIMD)

Performance of a Modern CPU

- The theoretically achieved FLOPS can be calculated as:
  - (3 GHz) * (16 cores) * (4 SIMD) * (2 fused multiply-add) * (2 ALUs) = 768 GFLOPS

- Let us consider multiplying a vector by a scalar (example on the right)
  - Requires one memory read and one memory write per floating point operation
  - Achieving 768 GFLOPS would require a memory transfer rate (bandwidth) of approx. 1.5 TB/s
  - State of the art hardware achieves only 50-150 GB/s

```
for(int i = 0; i < n; i++) {
    y[i] = 3 * x[i];
}
```

Compute Bound vs. Memory Bound

- A problem is **compute bound** if the performance is dictated by how many arithmetic operations the CPU can perform
  - Numerical integration, solving dense linear systems (LU), Monte Carlo methods, etc.
- A problem is **memory bound** if the performance is dictated by the bandwidth of main memory
  - Structured and unstructured grid codes, solving sparse linear systems, FFT, etc.
  - Performance measured in achieved GB/s
- How many memory instructions on the right?

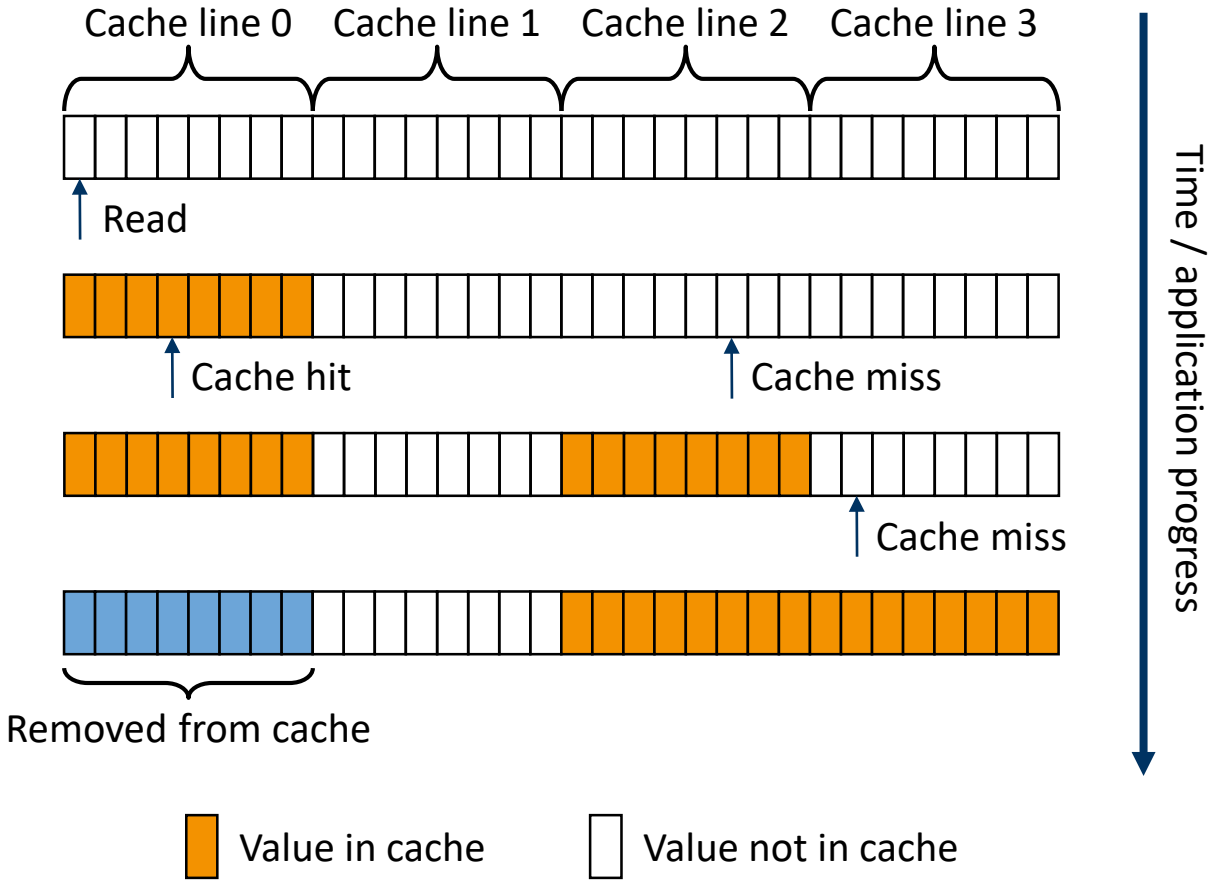
```
for(int i = 0; i < n-1; i++) {
    y[i] = x[i+1] - x[i];
}
```

Memory hierarchy

- FLOPS have increased dramatically but memory speed has been lagging behind
  - There is a cost, capacity, and speed trade-off involved
- The result is a **memory hierarchy**
  - Caches on the CPU (fast, tens of megabytes)
  - Main memory (medium speed, tens of gigabytes)
  - Disk storage (slow, terabytes-petabytes)
- Caches are usually further divided
  - Modern CPUs usually have L1, L2, L3 caches
  - Caches are completely transparent to the programmer

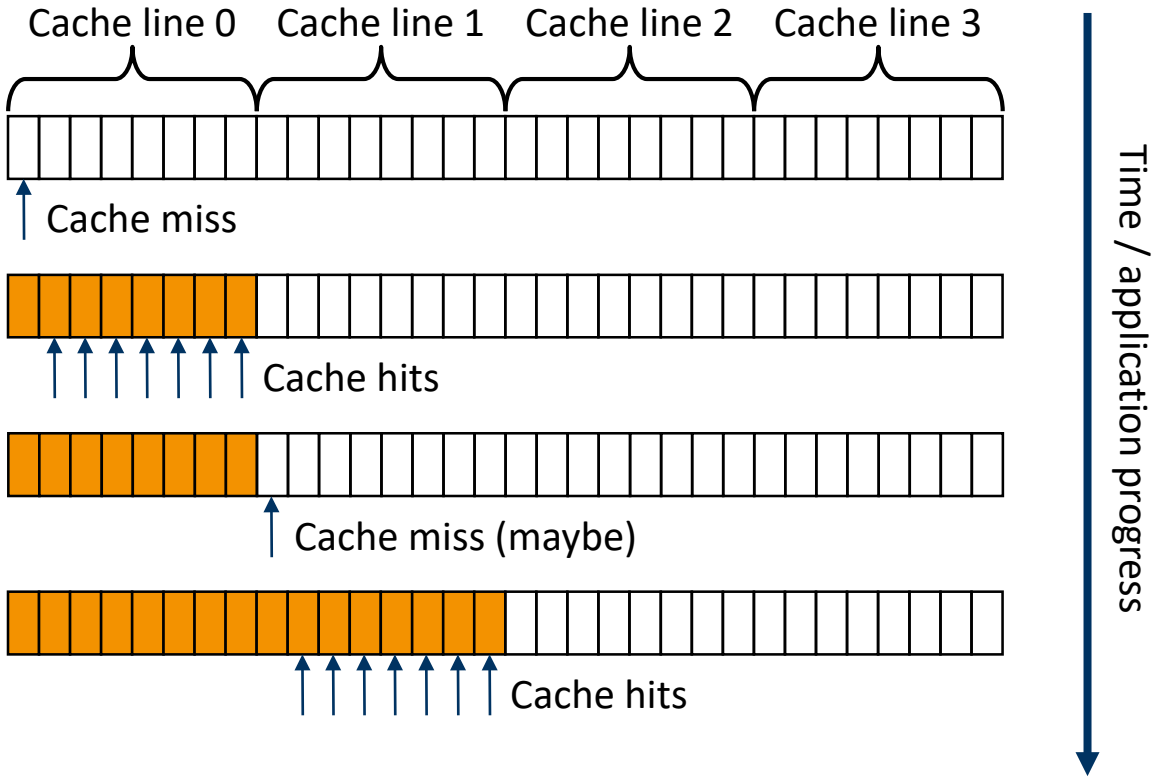
Caches

- Knowledge of how caches work is important for performance
  - Caches transfer data in chunks of fixed size (“cache lines”)
  - Usually 64-256 bytes in size (8-32 double-precision FP values)
- First read of any byte in a cache line transfers the entire cache line



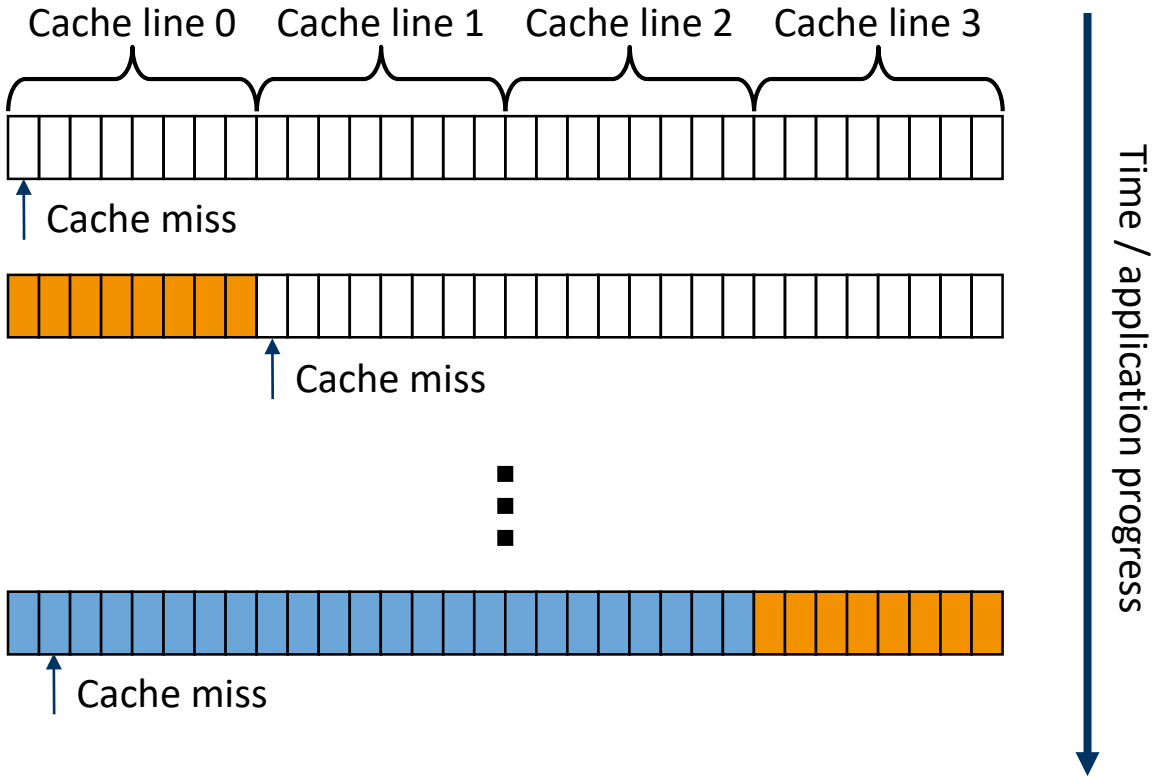
Memory Access Pattern

```
// Access with stride 1
for(int i = 0; i < n; i++) {
    out[i] = in[i];
}
```



Memory Access Pattern cont'd

```
// Access with stride 8
for(int j = 0; j < 8; j++) {
    for(int i = 0; i < n/8; i++) {
        out[j+8*i] = in[j+8*i];
    }
}
```



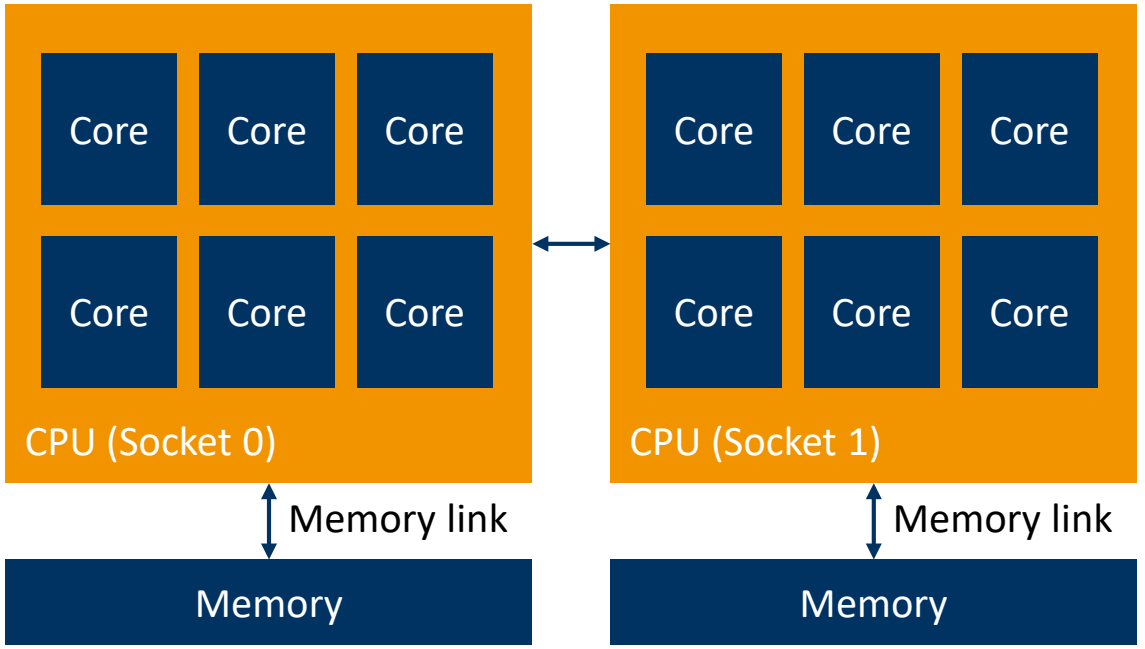
Latency

- Latency – how long the CPU has to wait between issuing a memory request and receiving the first byte – is an important performance consideration
  - There are physical limitations when reducing latency

- Modern CPUs employ a range of techniques to hide latency
  - Prefetching
    - Tries to load data likely used next into the cache
    - Before the actual memory instruction is issued
    - Particularly efficient for problems where memory locations close together are accessed in sequence
  - Instruction-Level Parallelism (ILP)
    - out-of-order execution: tries to delay instructions that still wait for memory
    - speculative execution: execute conditional code before checking the condition and verify afterwards

NUMA Domains

- Many systems are dual or quad socket now
- All cores can access the entire memory but speed might differ depending on which memory modules are accessed.
  - Non uniform memory accesses (NUMA)
  - Cores are grouped into NUMA domains
- Determine NUMA domains:
  `numactl --hardware`



First Touch

- For best performance, memory has to be placed “close” to where it is used
  - Neither C++ nor OpenMP provides a direct way to do that
  - This might not necessarily be desirable anyhow

- First touch principle
  - A memory location is mapped close to the core that first touches (reads or writes) it.
  - Parallelizing the initialization of data can make subsequent computation faster

- Different parts of an array can be placed on different NUMA domains

A Note on Optimization

- Modern CPUs are complicated
  - A basic understanding is vital but often measurement is necessary
- Rule of thumb penalties (in clock cycles) shown on the right
  - Exact numbers depend on specific hardware architecture

Operation	Cost in Cycles
arithmetics	1-5
L1 hit	1-10
function call	10-20
L3 hit	40
sin/cos	100
memory	200
disk	10^5

Further Reading

- Ulrich Drepper: What every programmer should know about memory
  - https://lwn.net/Articles/250967/?rss=1
- Colin Scott: Latency Numbers Every Programmer Should Know
  - https://colin-scott.github.io/personal_website/research/interactive_latency.html

Additional Topics not Covered Today

- Hyperthreading
  - It does NOT duplicate your cores
- Vectorization
  - SSE/AVX/SVE/VSX/…
- Distributed memory
  - Network technologies and topologies

Summary

- Understanding **hardware** is important to understand **performance**.
- Parallelism is required for high performance and must be explicitly used by the application developer.
- Know your bottlenecks!
  - compute bound, memory bound, cache optimality, etc.
  - Always employ data-driven optimization!

Image Sources

- Portrait photo: © Andreas Friedle
- 48 Years of Microprocessor Trend Data: https://zenodo.org/record/3947824