



Vlaanderen  
is supercomputing

# Supercomputers for Starters

February 2024

Kurt Lust – CalcUA, VSC and LUMI User Support Team

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 1: Introduction

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)

# Goals

- Why would one consider using supercomputers?
- How does supercomputer hardware influence our choice of software and programming techniques? And does that also affect regular PCs or devices like tablets and smartphones?
- What can (should) we (not) expect from a supercomputer?

I'm not a programmer, do I need to know all this?

- Yes, because supercomputers are very expensive machines and thus must be used efficiently
  - and that depends on the problem you're trying to solve,
  - your choice of software,
  - and on the resources that you request when starting a program.
  - In fact, if your software cannot exploit the hardware sufficiently well or your problem is too small, you should use something else.

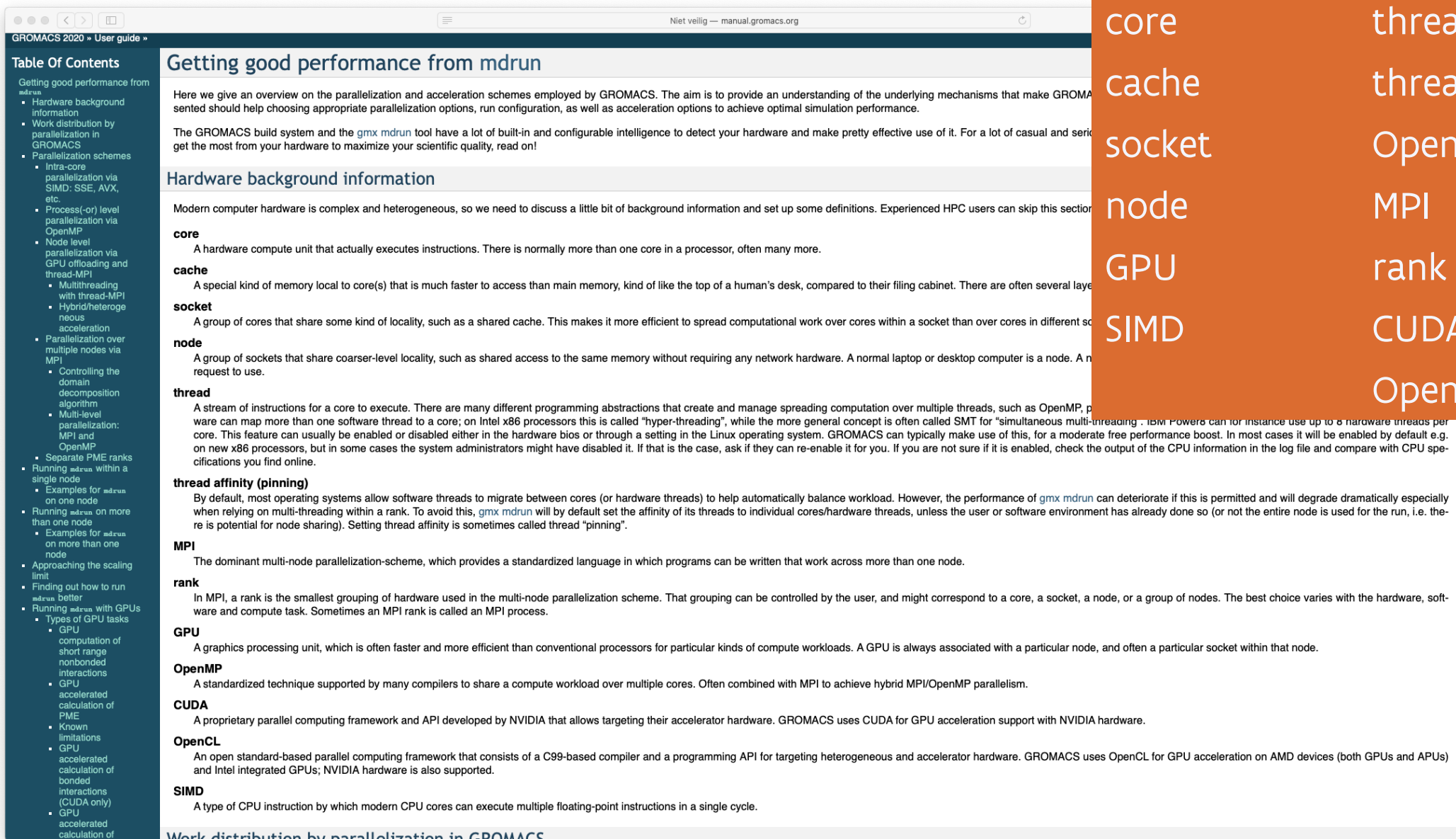
# Goals

- Why would one consider using supercomputers?
- How does supercomputer hardware influence our choice of software and programming techniques? And does that also affect regular PCs or devices like tablets and smartphones?
- What can (should) we (not) expect from a supercomputer?

I'm not a programmer, do I need to know all this?

- I prepared this lecture by looking at manuals of some software packages that our users use, including CP2K, OpenMX, QuantumESPRESSO, Gromacs and SAMtools and checked what those packages use.
  - The Gromacs manual even contains a section that tries to explain much of what we see here.

# Gromacs manual



The screenshot displays the GROMACS 2020 User guide manual page. The browser address bar shows 'Niet veilig — manual.gromacs.org'. The page title is 'GROMACS 2020 » User guide »'. The left sidebar contains a 'Table Of Contents' with a tree structure. The main content area is titled 'Getting good performance from mdrun' and includes an overview of parallelization and acceleration schemes, followed by a 'Hardware background information' section. The hardware section defines terms like core, cache, socket, node, thread, thread affinity, MPI, rank, GPU, SIMD, OpenMP, CUDA, and OpenCL. The right sidebar is an orange box listing these terms vertically.

**Table Of Contents**

- Getting good performance from `mdrun`
  - Hardware background information
  - Work distribution by parallelization in GROMACS
  - Parallelization schemes
    - Intra-core parallelization via SIMD: SSE, AVX, etc.
    - Process(-or) level parallelization via OpenMP
    - Node level parallelization via GPU offloading and thread-MPI
      - Multithreading with thread-MPI
      - Hybrid/heterogeneous acceleration
    - Parallelization over multiple nodes via MPI
      - Controlling the domain decomposition algorithm
      - Multi-level parallelization: MPI and OpenMP
    - Separate PME ranks
  - Running `mdrun` within a single node
    - Examples for `mdrun` on one node
  - Running `mdrun` on more than one node
    - Examples for `mdrun` on more than one node
  - Approaching the scaling limit
  - Finding out how to run `mdrun` better
  - Running `mdrun` with GPUs
    - Types of GPU tasks
      - GPU computation of short range nonbonded interactions
      - GPU accelerated calculation of PME
      - Known limitations
      - GPU accelerated calculation of bonded interactions (CUDA only)
      - GPU accelerated calculation of

## Getting good performance from mdrun

Here we give an overview on the parallelization and acceleration schemes employed by GROMACS. The aim is to provide an understanding of the underlying mechanisms that make GROMACS perform well. The presented should help choosing appropriate parallelization options, run configuration, as well as acceleration options to achieve optimal simulation performance.

The GROMACS build system and the `gmx mdrun` tool have a lot of built-in and configurable intelligence to detect your hardware and make pretty effective use of it. For a lot of casual and serious users, getting the most from your hardware to maximize your scientific quality, read on!

## Hardware background information

Modern computer hardware is complex and heterogeneous, so we need to discuss a little bit of background information and set up some definitions. Experienced HPC users can skip this section.

**core**

A hardware compute unit that actually executes instructions. There is normally more than one core in a processor, often many more.

**cache**

A special kind of memory local to core(s) that is much faster to access than main memory, kind of like the top of a human's desk, compared to their filing cabinet. There are often several layers of cache.

**socket**

A group of cores that share some kind of locality, such as a shared cache. This makes it more efficient to spread computational work over cores within a socket than over cores in different sockets.

**node**

A group of sockets that share coarser-level locality, such as shared access to the same memory without requiring any network hardware. A normal laptop or desktop computer is a node. A server rack is a node.

**thread**

A stream of instructions for a core to execute. There are many different programming abstractions that create and manage spreading computation over multiple threads, such as OpenMP, pthreads, etc. Hardware can map more than one software thread to a core; on Intel x86 processors this is called "hyper-threading", while the more general concept is often called SMT for "simultaneous multi-threading". IBM Power8 can for instance use up to 8 hardware threads per core. This feature can usually be enabled or disabled either in the hardware bios or through a setting in the Linux operating system. GROMACS can typically make use of this, for a moderate free performance boost. In most cases it will be enabled by default e.g. on new x86 processors, but in some cases the system administrators might have disabled it. If that is the case, ask if they can re-enable it for you. If you are not sure if it is enabled, check the output of the CPU information in the log file and compare with CPU specifications you find online.

**thread affinity (pinning)**

By default, most operating systems allow software threads to migrate between cores (or hardware threads) to help automatically balance workload. However, the performance of `gmx mdrun` can deteriorate if this is permitted and will degrade dramatically especially when relying on multi-threading within a rank. To avoid this, `gmx mdrun` will by default set the affinity of its threads to individual cores/hardware threads, unless the user or software environment has already done so (or not the entire node is used for the run, i.e. there is potential for node sharing). Setting thread affinity is sometimes called thread "pinning".

**MPI**

The dominant multi-node parallelization-scheme, which provides a standardized language in which programs can be written that work across more than one node.

**rank**

In MPI, a rank is the smallest grouping of hardware used in the multi-node parallelization scheme. That grouping can be controlled by the user, and might correspond to a core, a socket, a node, or a group of nodes. The best choice varies with the hardware, software and compute task. Sometimes an MPI rank is called an MPI process.

**GPU**

A graphics processing unit, which is often faster and more efficient than conventional processors for particular kinds of compute workloads. A GPU is always associated with a particular node, and often a particular socket within that node.

**OpenMP**

A standardized technique supported by many compilers to share a compute workload over multiple cores. Often combined with MPI to achieve hybrid MPI/OpenMP parallelism.

**CUDA**

A proprietary parallel computing framework and API developed by NVIDIA that allows targeting their accelerator hardware. GROMACS uses CUDA for GPU acceleration support with NVIDIA hardware.

**OpenCL**

An open standard-based parallel computing framework that consists of a C99-based compiler and a programming API for targeting heterogeneous and accelerator hardware. GROMACS uses OpenCL for GPU acceleration on AMD devices (both GPUs and APUs) and Intel integrated GPUs; NVIDIA hardware is also supported.

**SIMD**

A type of CPU instruction by which modern CPU cores can execute multiple floating-point instructions in a single cycle.

## Work distribution by parallelization in GROMACS

# SAMtools

**sort** `samtools sort [-l level] [-m maxMem] [-o out.bam] [-O format] [-n] [-T tmpprefix] [-@ threads] [in.sam|in.bam|in.cram]`

Sort alignments by leftmost coordinates, or by read name when **-n** is used. An appropriate **@HD-SO** sort order header tag will be added or an existing one updated if necessary.

The sorted output is written to standard output by default, or to the specified file (*out.bam*) when **-o** is used. This command will also create temporary files *tmpprefix.%d.bam* as needed when the entire alignment data cannot fit into memory (as controlled via the **-m** option).

## Options:

- l INT** Set the desired compression level for the final output file, ranging from 0 (uncompressed) or 1 (fastest but minimal compression) to 9 (best compression but slowest to write), similarly to **gzip**'s compression level setting.  
If **-l** is not used, the default compression level will apply.
- m INT** Approximately the maximum required memory per thread, specified either in bytes or with a **K**, **M**, or **G** suffix. [768 MiB]
- n** Sort by read names (i.e., the **QNAME** field) rather than by chromosomal coordinates.
- o FILE** Write the final sorted output to *FILE* rather than to standard output.
- O FORMAT** Write the final output as **sam**, **bam**, or **cram**.  
By default, samtools tries to select a format based on the **-o** filename extension; if output is to standard output or no format can be deduced, **bam** is selected.
- T PREFIX** Write temporary files to *PREFIX/nnnn.bam*, or if the specified *PREFIX* is an existing directory, to *PREFIX/samtools.mmm.mmm.tmp.nnnn.bam*, where *mmm* is unique to this invocation of the **sort** command.  
By default, any temporary files are written alongside the output file, as *out.bam.tmp.nnnn.bam*, or if output is to standard output, in the current directory as **samtools.mmm.mmm.tmp.nnnn.bam**.
- @ INT** Set number of sorting and compression threads. By default, operation is single-threaded.

Nice defaults for a '05 PC

What is a thread?  
How do I choose the number?

# VASP

From the VASP online manual and Wiki:

parallelisation (and data distribution)

LINUX cluster linked by Infiniband,

Message Parsing Interface (MPI)

OMP\_NUM\_THREADS

```
mpirun -bynode -np 8 -x OMP_NUM_THREADS=8 vasp
```

NCORE = total number cores / NPAR.

modern multi-core machines,

*massively parallel* systems

4 openMP threads.

**1 openMPI processes per socket (workaround)**

$$\text{NPAR} = \approx \sqrt{\text{number of cores}}$$

NPAR = number of cores per compute node

# Why supercomputing?

- Processing large datasets may require
  - more storage capacity than a workstation can deliver,
  - more bandwidth (memory or disk) than a regular server can deliver, and
  - more processing power than a workstation can deliver.
- Large simulations (e.g., partial differential equations)
  - may require (far) more processing power than a workstation can deliver
  - and often generate large data sets.
- Parameter analysis or Monte Carlo sampling:
  - A workstation may have enough processing power to process a single sample (or a few of them),
  - but what if we have 1000s of them?



# Supercomputing jobs

	Simulation	Data processing
Improve turnaround time Large memory capacity <i>Capability computing</i> <i>Hours per job</i>	Computational Fluid Dynamics, e.g. air flow around windmill Fluid-structure interactions Virtual crash test Simulation of complex molecules Climate modelling	Scientific visualisation of large data sets or simulation results Training a big AI model US postal: Electronic stamp
Improve throughput <i>Capacity computing</i> <i>Jobs per hour</i>	Parameter study: Simulate a system for multiple values of the parameters Test a range of molecules for some properties Risk analysis for banks	Gene sequencing  Data mining / search engines CERN LHC data processing Language research: Pre- processing of a text corpus

# What it does not

I have a Turbo Pascal program that runs too slow on my PC, so I want to run it on the supercomputer.

- Supercomputers only become supercomputers when running supercomputer software.
- All supercomputers nowadays are parallel computers combining sometimes thousands of regular CPUs to get the job done.
- The efforts to get your problem working on a supercomputer range from relatively minor to extensive. E.g.,
  - Parameter analysis or Monte Carlo sampling requires a relatively minor effort
  - Numerically solving a partial differential equation (e.g., fluid flow or mechanical stresses) requires a major effort but can deliver very good results
- But in many cases, someone else has done the work for you and software is already available.

# A parallel computer

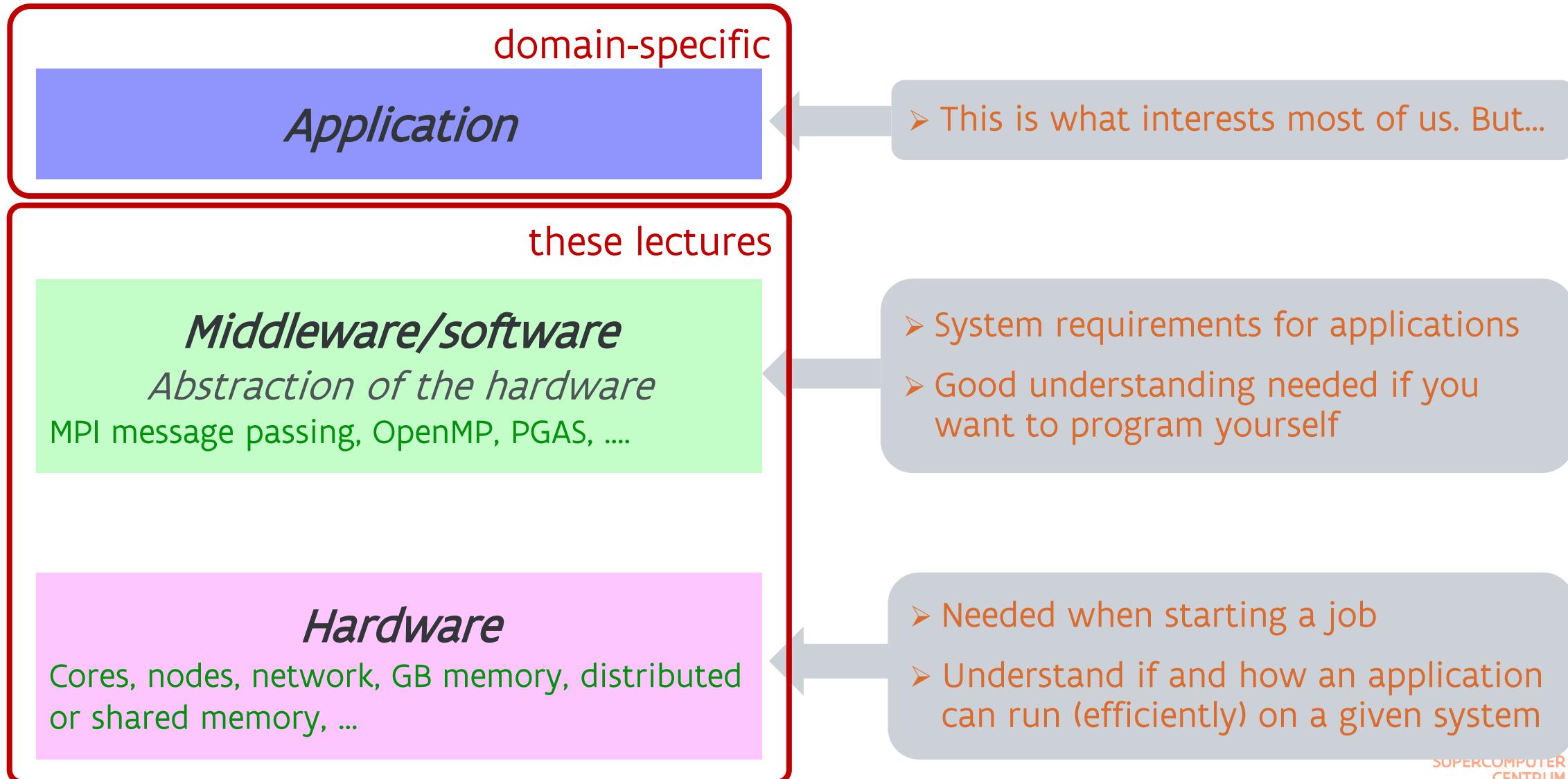
- A supercomputer is not a superscaled PC but a parallel computer in which
  - many processors work together to create a fast system
    - and this is multi-level parallelism
  - memory is organised in a hierarchy: from fast buffer memory close to the processor to slow disks
  - many hard disks and/or flash chips combine with the help of software into a powerful storage system
- And in the current state-of-technology this is far from transparent! (Well, it is transparent for correctness but not for performance)
- Hence the need for properly written software!
- PC's are just getting there (since roughly 2017)
- And tablets and smartphones are there also
  - For a while one could even argue that smartphones and tablets are better parallel computers than the average PC

Part II

Part III

Part IV

# A layered architecture



# A layered architecture

## *Application*

- Part VII discusses what we can expect from parallel computing

## *Middleware/software*

*Abstraction of the hardware*

MPI message passing, OpenMP, PGAS, ...

- Part VI discusses popular middleware

## *Hardware*

Cores, nodes, network, GB memory, distributed or shared memory, ...

- Part II-V discuss hardware aspects

# A compartmentalised supercomputer

## Login section

- Access point for users

## Storage section

- Run the file system

## Management section

- Controls the system

## Compute section(s)

- Where the actual computations are done



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 2: Processors in supercomputers

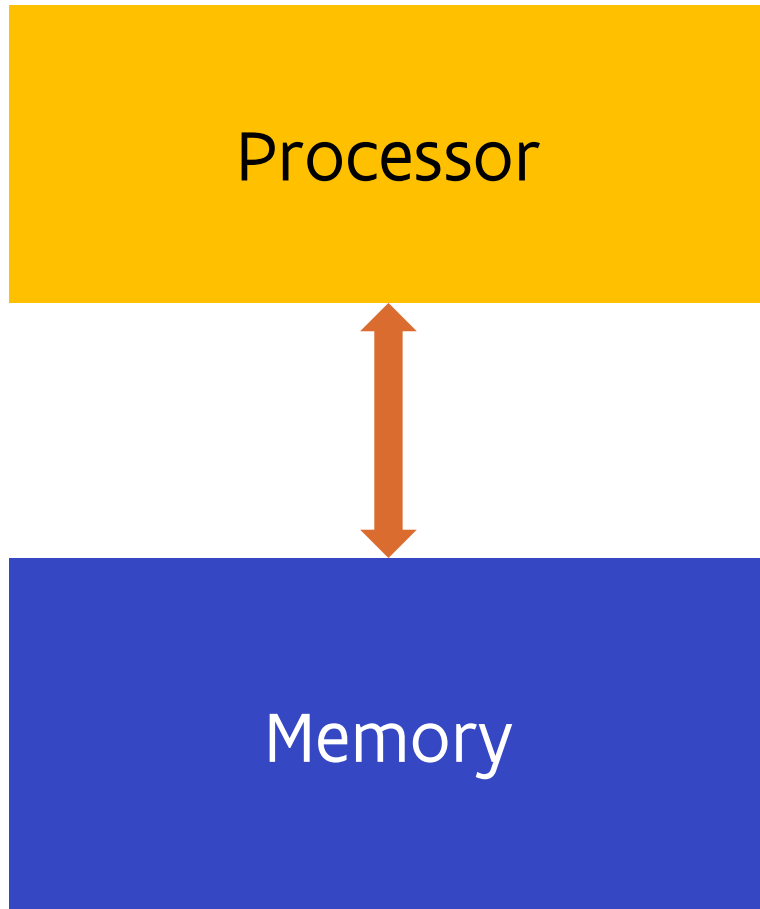
# The CPU: 1 GHz $\neq$ 1 GHz

GHz does not measure how much work a CPU can do.

- Supercomputers use CPUs derived from those in PCs or smartphones
  - But have enhancements for reliability
- These CPUs have gone through a long evolution to do more work per clock cycle:
  - More instructions per clock: **Instruction-Level Parallelism**
  - More work per instruction: **Vectorization (and matrix computing)**
- But this was not enough, so
  - More CPUs (“cores”) that share memory: **shared memory parallel computing**
  - Multiple “nodes” that collaborate by sending messages over a network: **distributed memory parallel computing**



# A simple computer

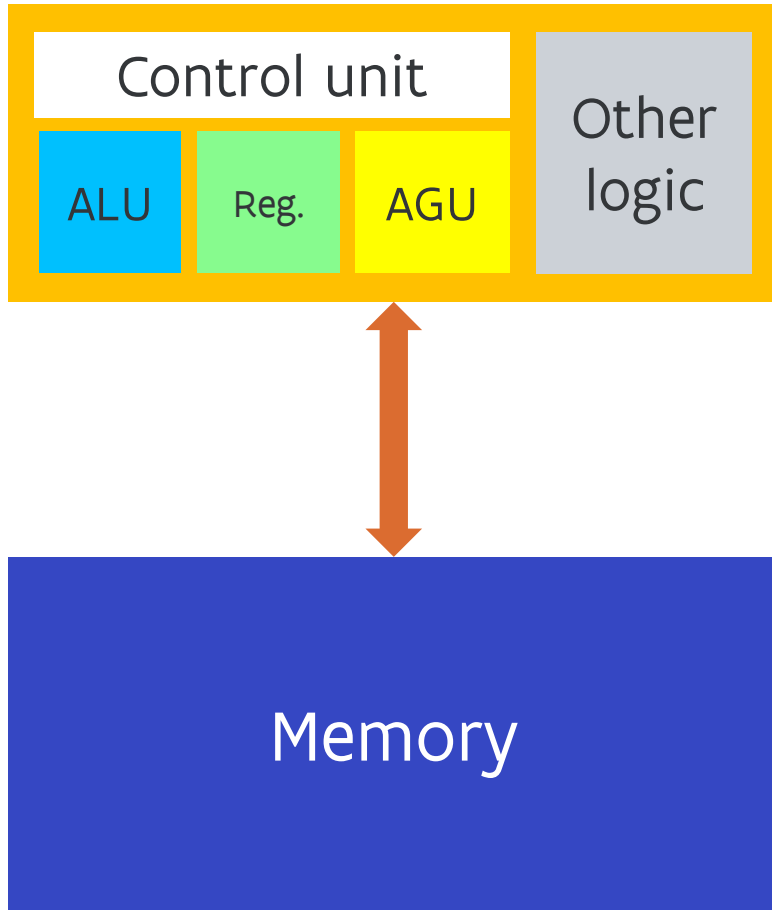


- Processor executes simple instructions (e.g., add two numbers)
- Memory stores the data in a linear structure
- A clock governs everything
- Processor is currently 1 chip (or a small part of a chip), but long ago this could consist of multiple chips (even 1000s of chips)

Cray 1 (1976):

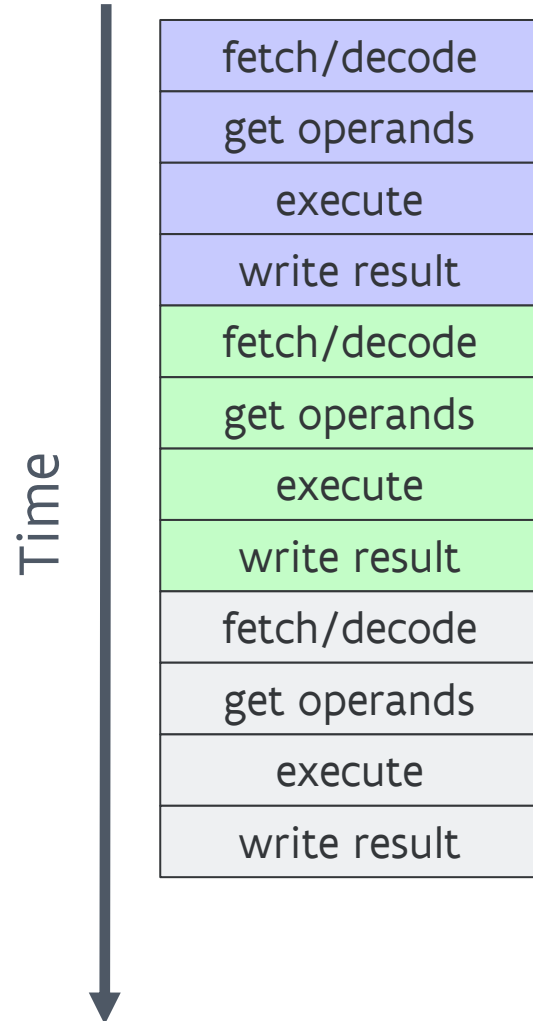
- Processor: 20,000 chips
- Memory: 73,728 chips,
- Mean time between failure = 50h

# A simple computer – A look inside



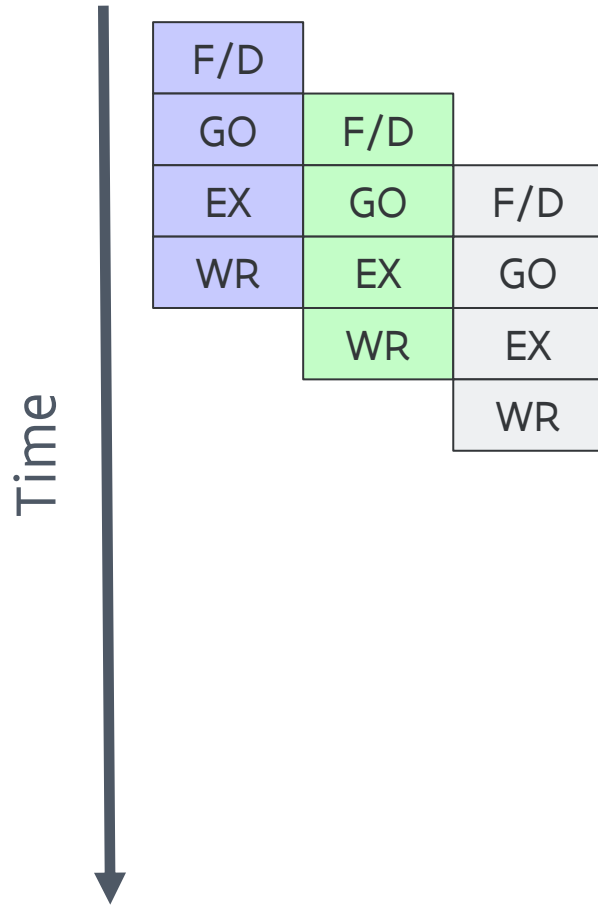
- **ALU:** Arithmetic and logical unit, does the actual computations
- **Registers:** Fast memory cells where ALU instructions fetch their data and write their results
- **AGU & memory controller:** To connect to the memory
- **Control unit:** Coordinates the work

# Executing instructions



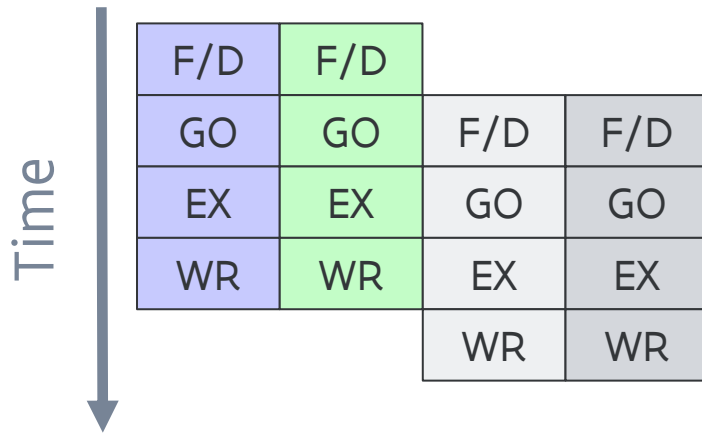
- Instructions execute one after another
- But instruction execution consists of multiple phases
- One step/phase per clock tick: 0.25 instructions per clock in this example
- But note: Different phases use different logic on the chip

# Instruction-level parallelism: Pipelining



- Different phases use different logic on the chip
  - So can we create overlap in the processing of instructions?
- Pipelining: Compare to a car assembly line
- In this simple model: Ideally 1 instruction per clock, 4 times faster
  - But this requires that the next instruction doesn't need the result of the previous one
- **Instruction-Level Parallelism:** The CPU is working on multiple instructions simultaneously
- Supercomputers: IBM System/360 Model 91 (1964), CDC7600 (1967)
- Used in PC CPUs since the mid '80s: i386 (1985) and i486 (1989)

# Instruction-level parallelism: Superscalar execution

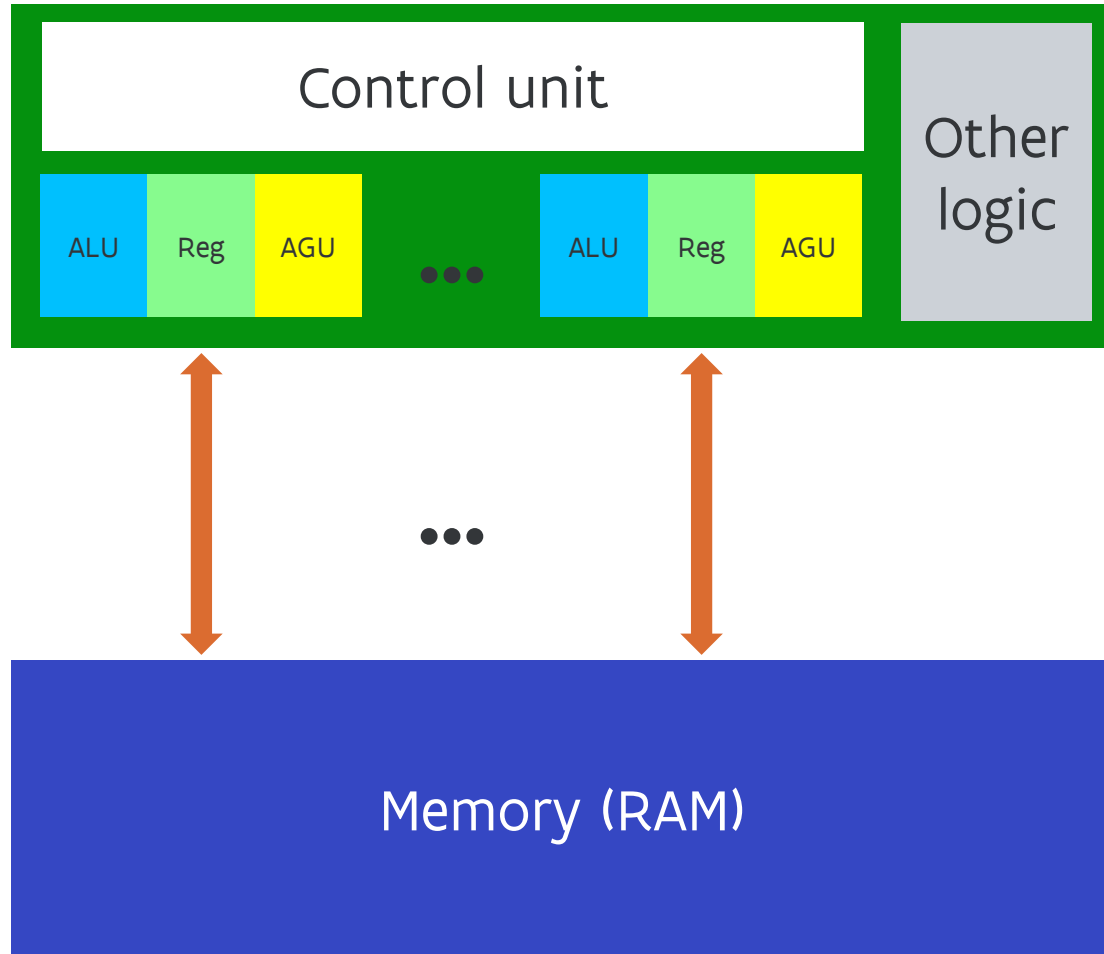


- ▶ Now we could increase the number of ALUs and AGUs on the processor
  - Start multiple instructions simultaneously
- ▶ **Superscalar execution**, also a form of ILP
- ▶ Potential for >1 instruction/clock
- ▶ Supercomputers: IBM System/360 Model 91 (1964), CDC6600 (1964 - before pipelining!)
- ▶ PC technology: 90's, Pentium (1993) and Pentium Pro (1995)
  - And has evolved with hardware reordering of instructions
- ▶ Exploitation of ILP is largely done by the compiler and CPU hardware
- ▶ But whether a compiler can exploit ILP also depends on the program itself.
  - Frequent testing tends to kill ILP

# Data parallelism through vector computing

- Pipelined and superscalar execution: complicated logic and hence a lot of power
  - But a common case that is well suited for superscalar execution is working with vectors
- So designing CPUs with vector instructions (“wider ALUs”) can boost speed without the full power requirements of a superscalar processor.
  - An example of **Single Instruction stream, Multiple Data stream (SIMD)** architecture
- Popular in supercomputers in the 70s-80s, but then almost disappeared.
  - CDC STAR-100 (1974), Cray-1 (1976, first with vector registers)
  - NEC still makes vector computers (architecture: 256-wide DP, 32-wide DP execution x 8 steps)
- However, now returning in general-purpose computers (but shorter vector length!)
  - MMX (1996) / SSE (1999) / AVX (2011) instructions in x86 processors (but only short vectors)
  - AVX-512 in (defunct) Xeon Phi for HPC, Skylake X and Ice Lake (16-wide SP, 8-wide DP)
  - NEON in ARM (4-wide SP, 2-wide DP)
  - SVE in ARM for Fujitsu supercomputer (architecture: 2048-bit, 32-wide DP, implementation: 512-bit, 8-wide DP) and in ARM v9 (Samsung S22 and later, NVIDIA Grace CPU)
  - AMD GCN and CDNA GPU’s (16-wide SP hardware, 64-wide SP instructions) and RDNA GPU’s

# Data-level parallelism: Other SIMD



- Many “processors”, but they share the control unit and must all execute the same instruction on different data.
- Historical example: Thinking Machines Connection Machine CM-1 (1983)
- Modern example: NVIDIA GPUs (10-100+ SIM(D)(T) processors on a chip)
- Difficult to program efficiently!

# Conclusion:

## 2 levels of parallelism in the CPU

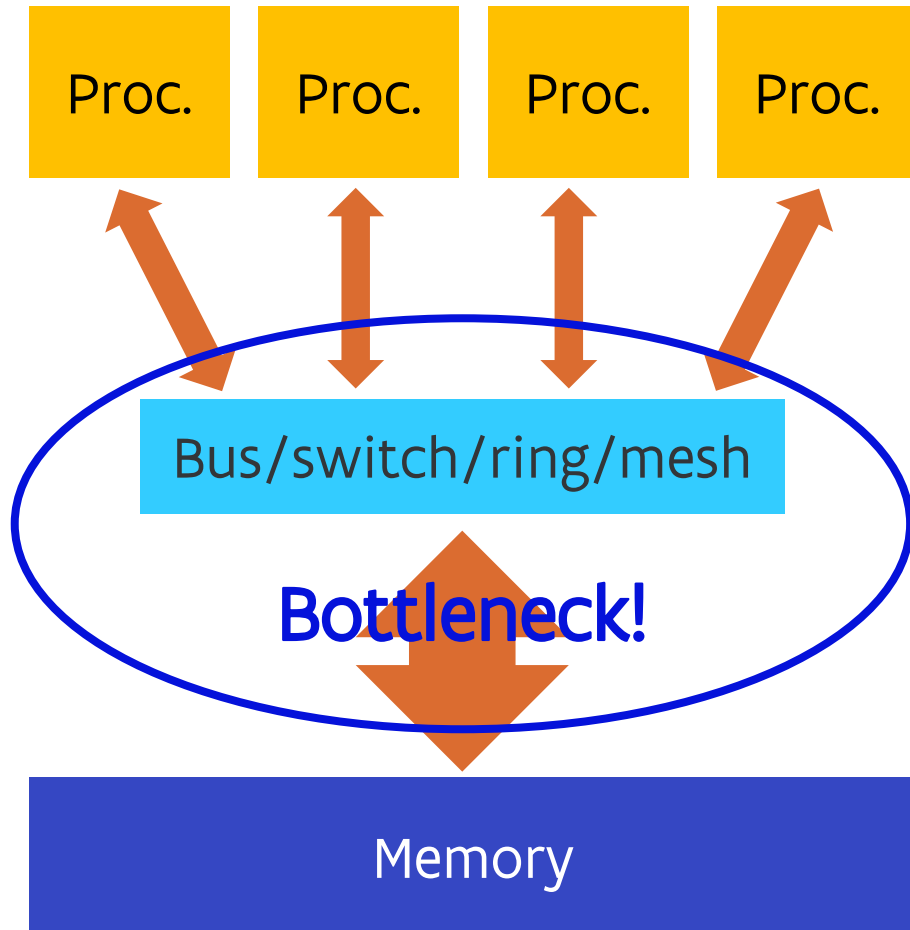
CPUs can do more work per clock by doing:

- More instructions per clock cycle: Instruction-level parallelism
  - Mostly hard work for the CPU control logic and the compiler
  - Some work for the application developer
  - You can expect a gain from this technology without even recompiling your application
- More work per instruction: Data-level parallelism through SIMD/vectorisation
  - Hard work for the compiler
    - Most programming languages not very helpful: They don't offer enough information to the compiler
  - Therefore the compiler is only moderately successful in vectorising the code, so work for the application developer.
  - No gain without recompiling for vector instructions



# Symmetric multiprocessing

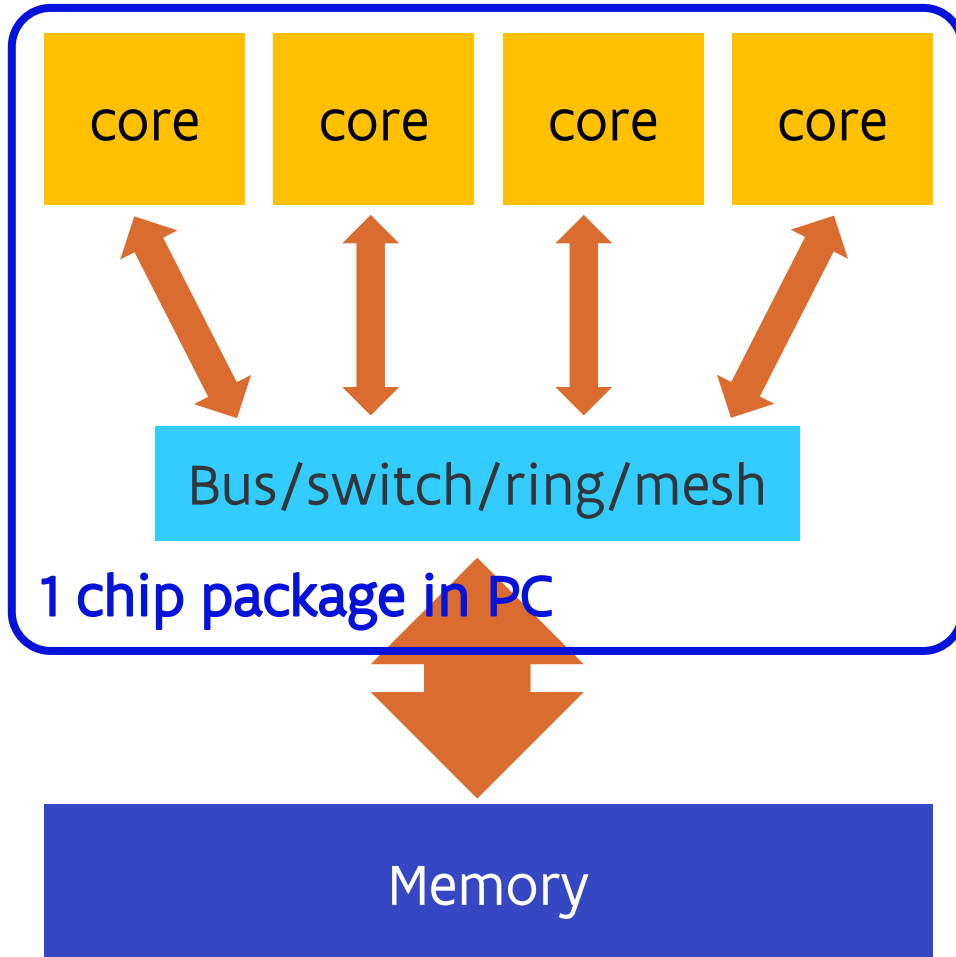
- Increase performance by using multiple processors



- Multiple independent processors, each working on their own data elements: MIMD (Multiple Instruction, Multiple Data)
- All processors equal: symmetric multiprocessing (SMP)
- Every processor equal access to all memory: Shared memory with Uniform Memory Access
- But there is a potential bottleneck: the bus to memory

# Symmetric multiprocessing

- Increase performance by using multiple processors

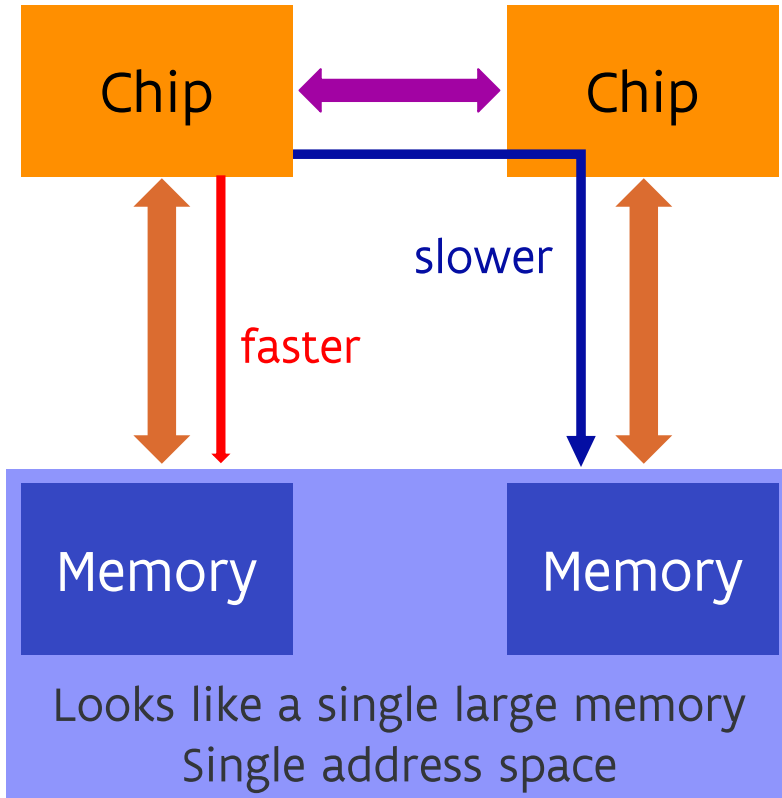


- Evolution of terminology: As multiple “processors” were integrated on a single die (“chip”), it became unclear what was the processor:
  - Core: The unit on the chip that by itself could execute a program
  - Package: Contains one or more dies with one or more cores each
  - Socket: Often the package is plugged into a socket, OS uses the term socket
  - Processor: Often used for the package
- Shared memory multiprocessing is everywhere:
  - PC and smartphone processors
  - GPU: Multiple SIMD procs!

# Shared-memory multiprocessing

## Non-Uniform Memory Access

- Split the memory, but maintain a global address space



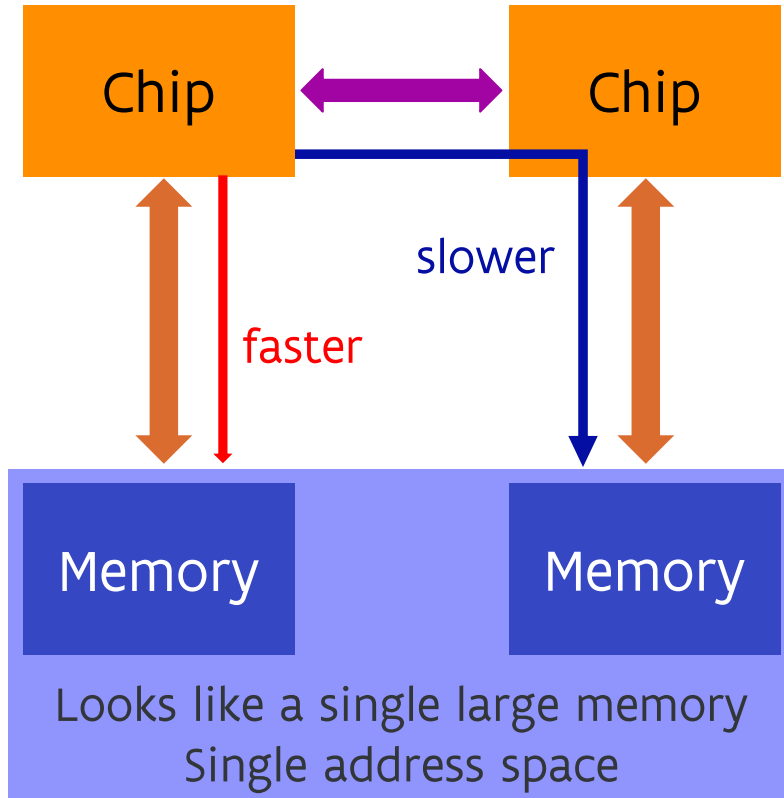
- Each (multi-core) chip has some memory attached to it.
- Chips are connected via a special-purpose network (UPI = UltraPath, Infinity Fabric, future: CXL)
- Each core on each chip can still directly reach all memory, but access to “local” memory faster than to “remote” memory: NUMA
- Transparent with respect to correctness of programs but not fully transparent when it comes to performance

# Shared-memory multiprocessing

## Non-Uniform Memory Access

➤ Split the memory, but maintain a global address space

➤ Examples:

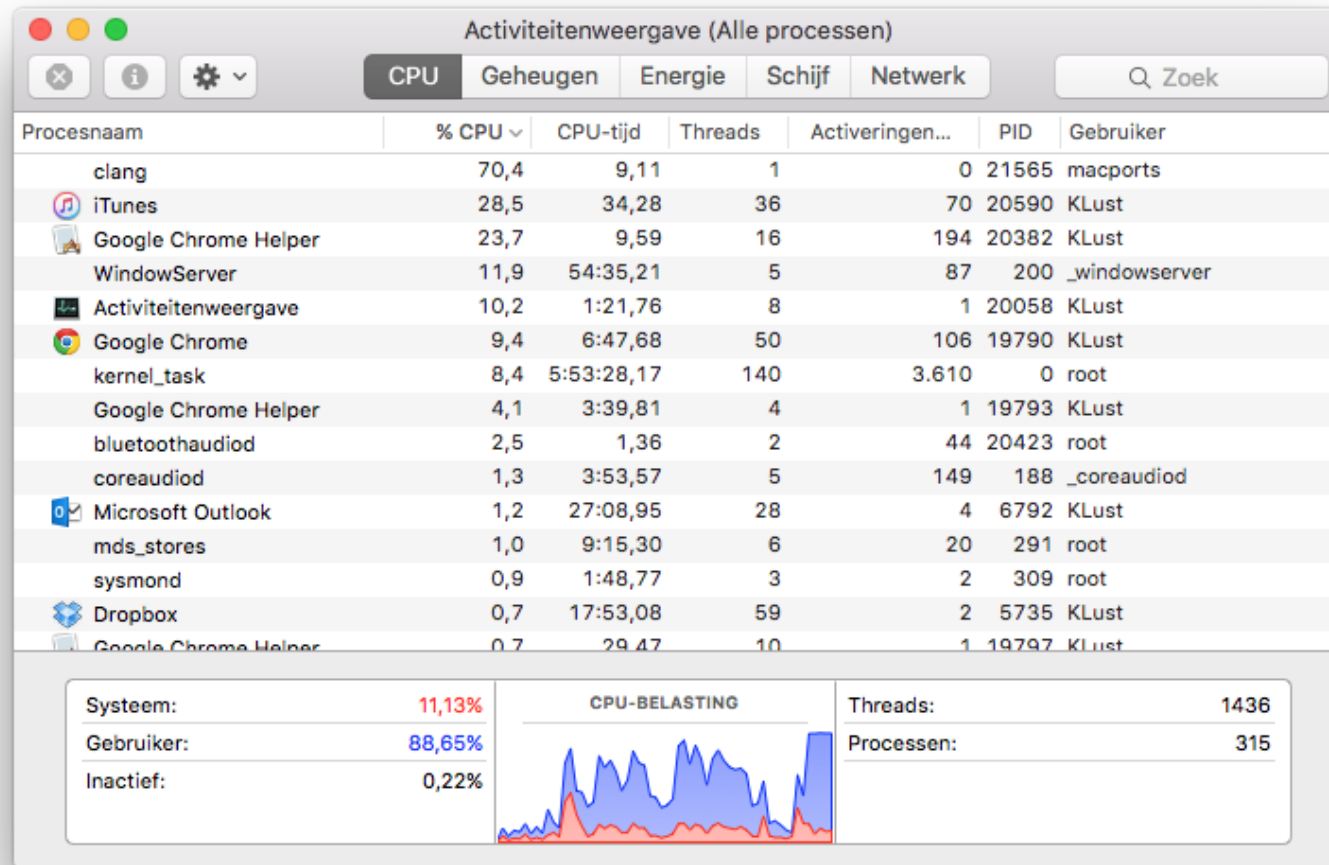


- All current multiple-socket server CPUs, e.g., Intel Xeon or AMD Epyc
- SGI Altix UV / HPE Superdome Flex: up to 64 chips
- AMD EPYC gen 1 (Naples): Up to 4 dies with each 2 memory controllers and 2 groups of 4 cores, with an interconnect between the dies.
- Some Intel Xeon CPUs and more recent AMD EPYC CPUs can also be split logically into two or more clusters on a single die

# What does this look like in software?

- **Process:** Operating system concept.
  - Created when you start an app on your phone, click on a .exe-file in Windows or type a command in Linux.
  - Has an amount of memory that other processes cannot access, can have exclusive access to files etc.
- In the old days (think MS DOS), there was one stream of instructions executed in the context of a process.
  - A single application would not be able to exploit multi-core processors!
- **(OS) thread:**
  - An instruction stream that is executed in a process
  - So every thread can see all memory of that process (though there is some thread-private memory)
  - Threads can run on different cores, though 1 core can also execute multiple threads by continuously switching between them.
  - But be careful with compute threads in scientific computing applications!

# What does this look like in software?



My Mac while compiling some code, playing a YouTube video in Chrome and some music in iTunes, all at the same time.

# Hardware threads

- Different names for similar technologies: Simultaneous Multi-Threading (generic term, IBM POWER, AMD Zen), Hyper-Threading (Intel), hardware threading (Oracle)
- Often not enough parallelism in a single instruction stream to exploit all resources on a core
  - So run multiple instruction streams on a single core
  - The core effectively behaves as multiple virtual cores, but those virtual cores compete for the same resources
- Gain depends on the architecture and the application.
  - Intel Xeon or Core i7: the gain of the second thread is often moderate
  - On Intel Xeon Phi KNC generation you need to use at least two hardware threads to get the maximal performance.
- Fairly transparent in software (though not 100% performance transparent)
- Hyperthreading enabled on some node types at the VSC.

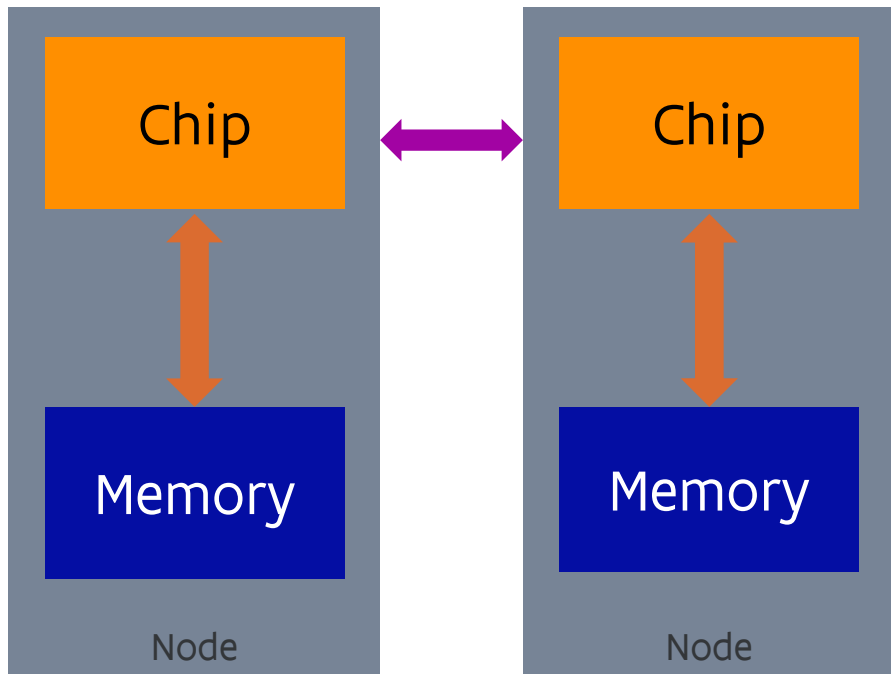
# Programming shared memory

- Automatic parallelisation is probably even less successful than automatic vectorisation
- So as a programmer, you'll need to work, and there are several approaches:
  - See section VI on middleware
- So this implies that your application has to be written to exploit this level of parallelism
  - Though it can be easily exploited by running multiple copies of your program with a different data set, each on a single core
  - Even as a user you have to be aware of this as you often need to tell your program how many threads it should use



# Distributed memory

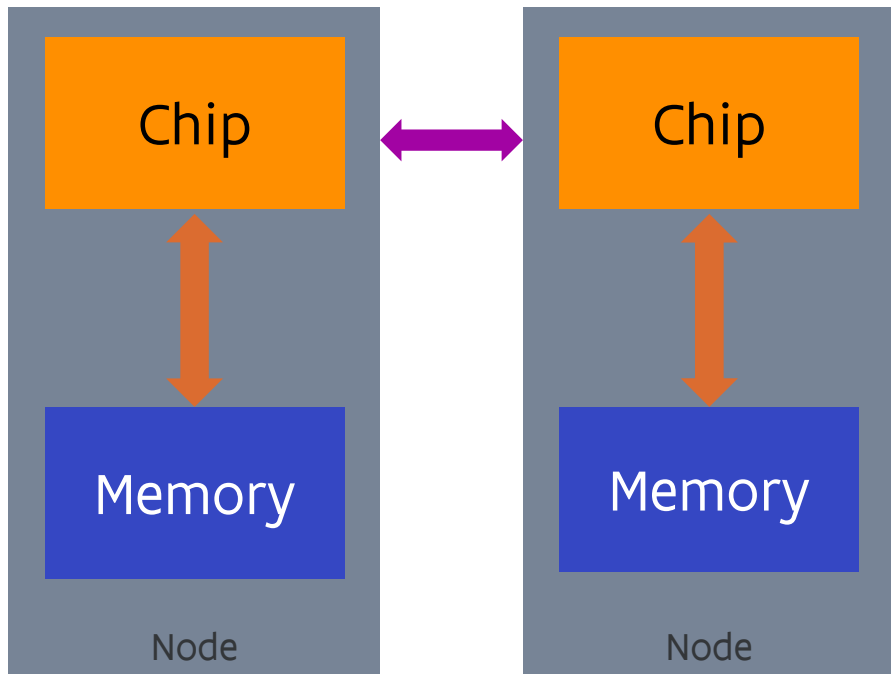
- NUMA also has its scaling limits:
  - Hardware: Technological challenges limit scaling
  - Software: Operating systems do not scale well enough
  - Economical: Large fast networks are expensive



- Solution:
  - Take a number of pretty standard computers (called the “nodes”)
  - Link together with a network (“interconnect”)
- Communication over a network by sending and receiving (application-initiated) messages
  - No joint global address space (at least not in hardware)!
  - Evolution to interconnects with limited memory semantics.

# Distributed memory

- No direct access to the memory of another node
  - Hence at least 1 process per node
  - And software to let those processes communicate by sending messages over the network.



- Harder to program, but far more scalable than shared memory
- Beowulf cluster hype (1994 on): Let's build this with cheap&lousy standard components
- In practice, you need high-quality components and sometimes dedicated technology for good performance and high enough reliability

# Cheap&lousy - Anecdotes

- 2002, ASCI Q @ Los Alamos National Laboratory: 2048 4-socket nodes, DEC Alpha EV-68
  - Initial mean-time-between-crashes of 1 hour due to cosmic ray sensitivity
  - After better shielding: MTBC of 6 hours
  - Cause: Data path without error correction in DEC Alpha CPU
- 2003, Big Mac @ Virginia Tech: 1,100 PowerMac G5 cluster
  - Started crashing before it was completely booted
  - Cause: No ECC memory and therefore too sensitive to cosmic rays
- 2009, Jaguar, Cray XT5 system at Oak Ridge National Laboratory
  - Then the largest system in the world. 360 TB of memory, 18774 nodes with 2 quad core AMD processors.
  - 350 correctable ECC errors per minute!
  - On average an uncorrectable 2-bit error every 24 hours
- See [IEEE spectrum article “How to kill a supercomputer: Dirty power, cosmic rays, and bad solder”](#)

# Programming distributed memory

- Automatic strategies through the compiler never made it past the research phase
- So very hard work for the programmer
  - Though there are computer languages that help
  - See section VI on middleware
- So this implies that your application must be written to exploit this level of parallelism
  - Though it can be easily exploited by running one or more copies of your program independently on each node

# A modern supercomputer...

A modern supercomputer often uses all these tricks!

- Most modern supercomputers are **distributed memory machines**, a network of compute nodes and some specialised nodes.
- Each node is a **shared memory machine**,
  - often of the NUMA type with **two or more CPU packages (sockets)**
  - where each socket contains a **multi-core CPU** very similar to the processor in your PC but with more cores
  - and each core often operates as two (or more) virtual cores through **hardware threads**.
- Each processor core on such a chip nowadays supports **vector instructions**
- and has **extensive instruction-level parallelism**.
- Add a GPU (or other) accelerator and things become even more complicated.

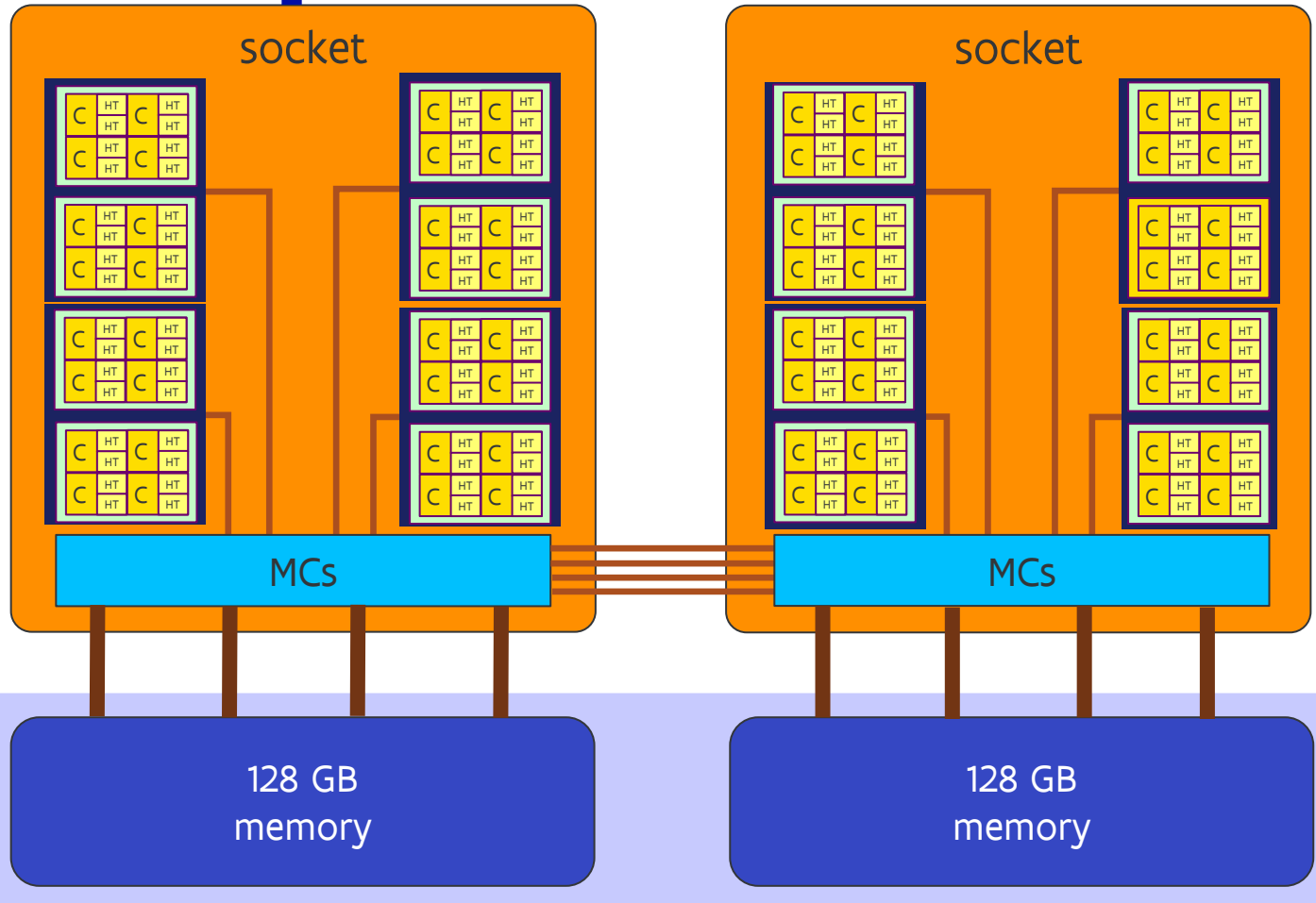
PC and smartphone

Most servers, some supercomputers

Almost all supercomputers

# Vaughan (AMD Rome) node

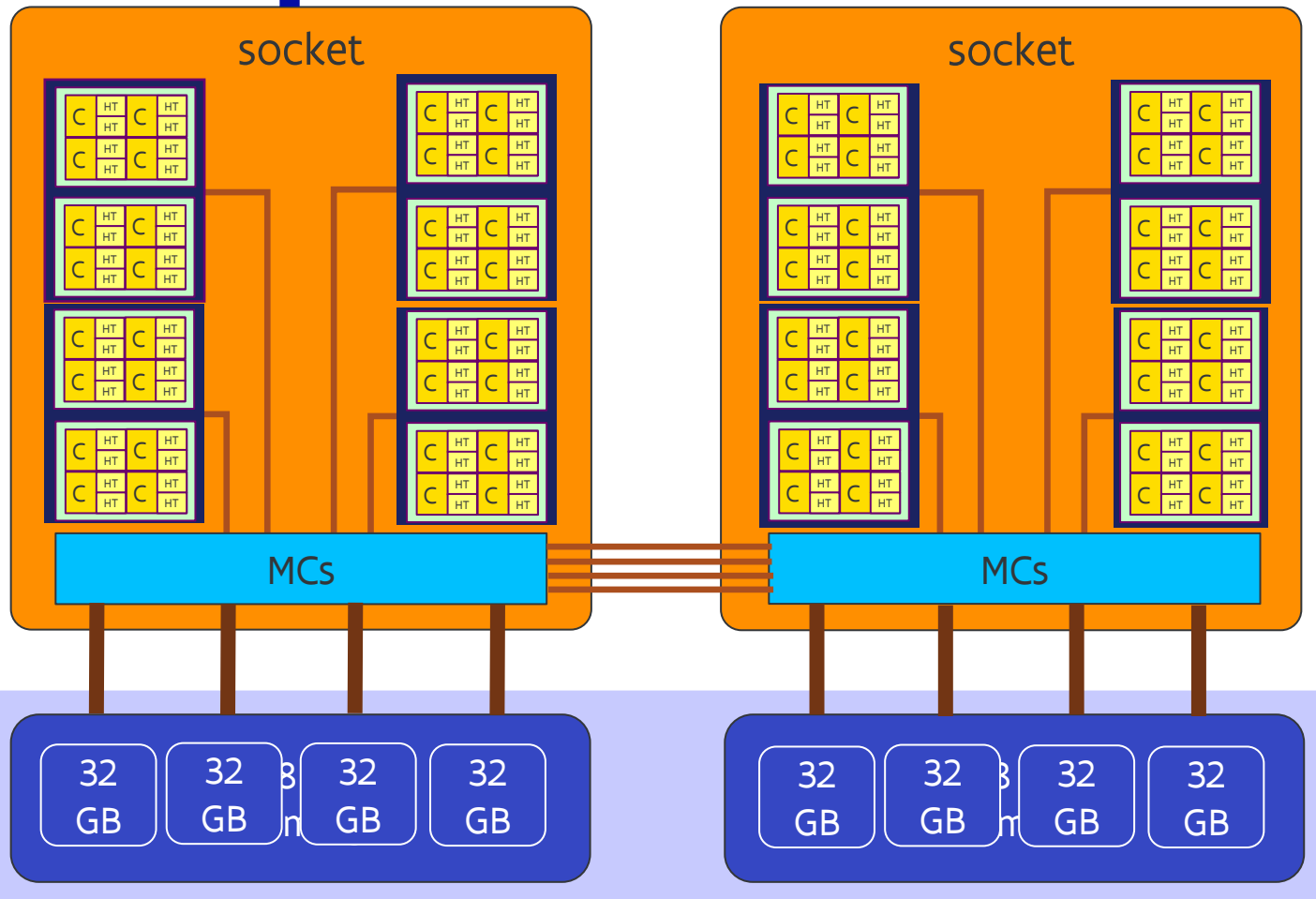
To interconnect



- Vaughan @ UAntwerpen
- Hortense 1 @ UGent (but 16 core complexes per socket)
- LUMI @ CSC but AMD Milan with 8 complexes of 8 cores per socket in the CPU section, and new nodes in Hortense

# Vaughan (AMD Rome) node

To interconnect



- Memory controller in 4 quarters, serves 1 (Vaughan) or 2 CCD (Hortense, LUMI), so the system even behaves as a NUMA system at the socket level.

# Fast evolution

1977: Cray 1A (ECMWF)	2007: Apple iPhone
80MHz CPU, 160-250 Mflops	412MHz CPU, 412 Mflops
8 MB RAM	128 MB RAM
2.4GB disk space	Up to 8GB permanent storage (flash)
5,500 kg	0.135 kg
115 kW	A few Watts
\$8.86M	\$500
MTBF: 50 hours	Weeks without reboot, years without repair
Needed a CDC as front-end	Comes with a tiny built-in monitor

- A typical PC in 1977 (TRS 80 model 1, Apple II) had a clock speed of 1-2MHz, no floating point hardware and because of that only reached on the order of 100-200 flops.



## Fast evolution (2)

1994: KU Leuven SP2	2014: My laptop (MacBook Air)
16 66MHz IBM Power2+ CPUs	1 dual-core Intel CPU@1.3 GHz
4.25 Gflops DP	41.6 Gflops DP (more with turbo boost)
11/1994: Position 198 in Top500 11/1995: Position 409 in Top500	11/2014: Position 198 is 364 Tflops 11/2015: Position 409 is 391 Tflops 06/2015: Position 500 is 402 Tflops
3 GB RAM	8GB RAM
80GB disk space	256 GB disk space (SSD)
1 ton?	Less than 2kg (with power supply)
>10 kW?	<45W
1M€?	€1300

# Can a PC be faster than a supercomputer today?

**YES!**

- Because they are optimised for different tasks
- Power consumption of electronics on a chip: Twice the clock speed for a given core architecture requires far more than twice the power.
- Assume a factor 4:
  - Suppose we could run 4 cores at 4 GHz for 100W of power,
  - Then we could run 16 cores at 2 GHz for the same 100W of power.
  - So if the performance of the cores would be the same per clock tick in both scenarios, the 16-core configuration is twice as fast for applications that can make effective use of all cores.
  - However, if an application cannot use multiple cores efficiently, the 4-core configuration will be faster.

# Can a PC be faster than a supercomputer today?

**YES!**

- Your desktop PC is optimized to run both moderately parallel applications and serial applications well. Hence the choice was made to use fewer cores at a higher clock speed.
  - For a long time, PC's had very few cores that ran at a very high clock speed.
  - This is changing, but very high “turbo boost” frequencies save us for single-threaded applications
- The processors of our clusters however are mostly optimized for applications with a high degree of parallelism. Hence they have a lot of cores but at a lower clock speed than most desktop PCs.
- So if your software/workflow is not parallel, running it on a supercomputer is a waste of money.
- Supercomputers need appropriate software to function as a supercomputer! In fact, the key point of supercomputing since the mid '80s has been adapting software to be able to use cheaper hardware.

# Don't believe me?

The table is for a pretty old generation of Intel CPUs but one we can still understand.

	HPC power budget		Maximum performance		
	E5-2623v3	E5-2660v3	E5-2637v4	E5-2698v4	E5-2699v4
Cores	4	10	4	20	22
Clock	3 GHz	2.6 GHz	3.5 GHz	2.3 GHz	2.3 GHz
Power	105 W	105 W	135 W	135 W	145 W
L3	10 MB	25 MB	15 MB	50 MB	55 MB
Memory bandwidth	59 GB/s	68 GB/s	76.8 GB/s	76.8 GB/s	76.8 GB/s
Gflops peak DP/core	48	41.6	56	36.8	36.8
Gflops peak DP	192	416	224	736	810



x2.16



x3.3

x3.6

## Part II: Lessons learnt

- 4 levels of parallelism in supercomputer, 3 of them also on PCs
- Hardware of a supercomputer is optimized for parallel performance and not for sequential performance.
  - No parallelism = no supercomputing
- Computers (and compilers) have evolved. Hence:
  - A 10-year old binary won't run efficiently and might not run at all.
  - Code that remained unmodified for 20 years will not exploit all the parallelism on modern computers either. In fact, it won't even be efficient on your PC.
- A supercomputer contains 1000's of times more parts than a regular PC. Hence it probably won't be as reliable, so it is not a good idea to try to run a multi-day job without making sure it stores enough data to restart an interrupted computation.

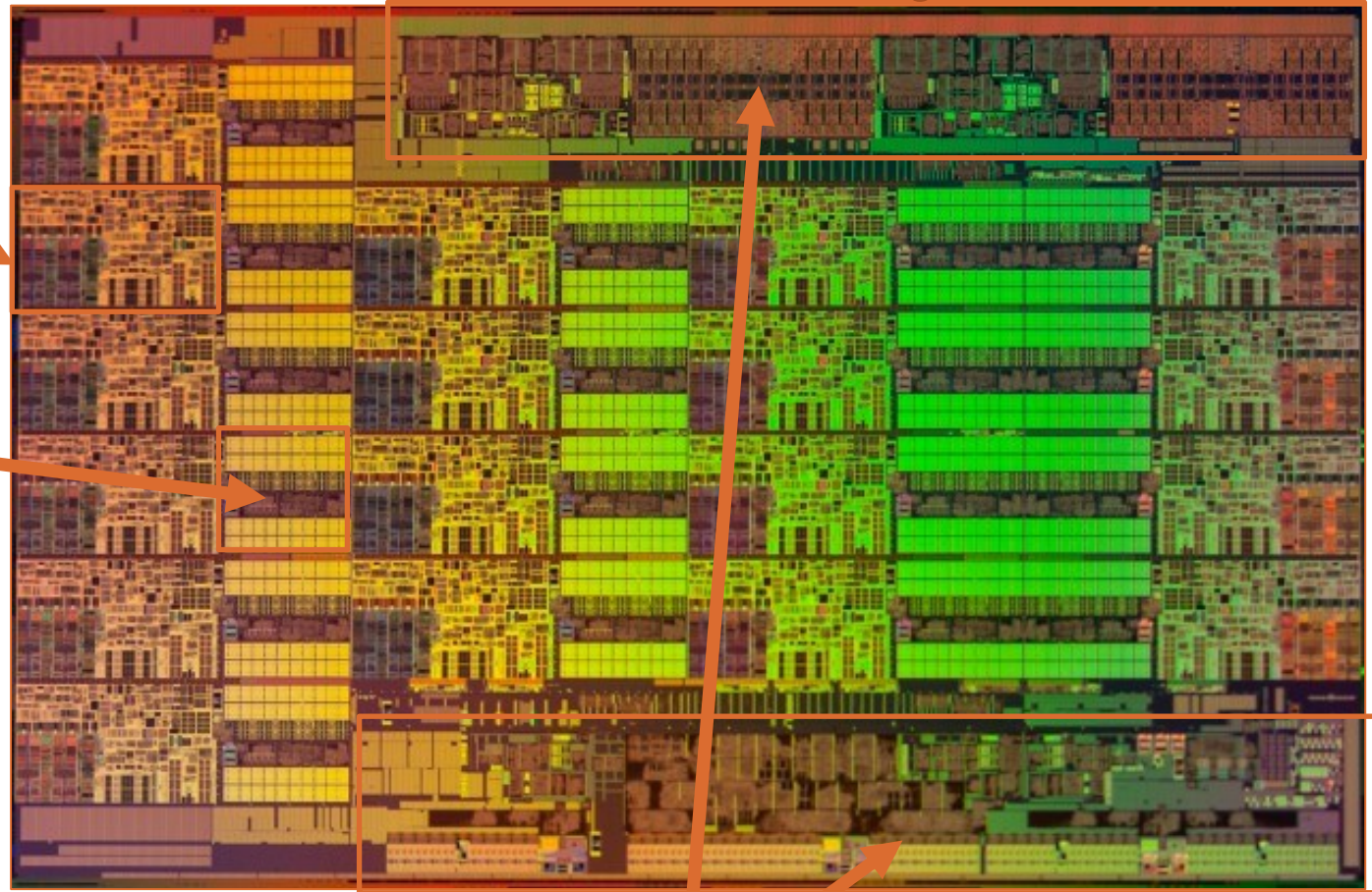
# **Intermezzo: A look at the processor**

# CPU die shot

Intel E5 v3 14-18C die (Haswell generation)

Core (18 in total)

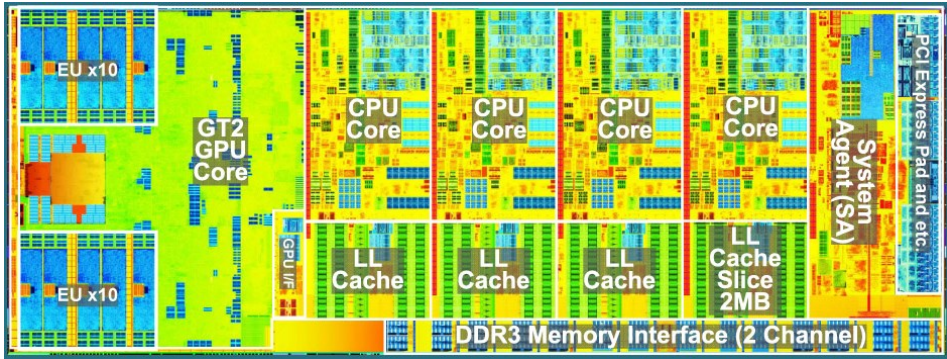
L3 cache segment  
(18 in total)



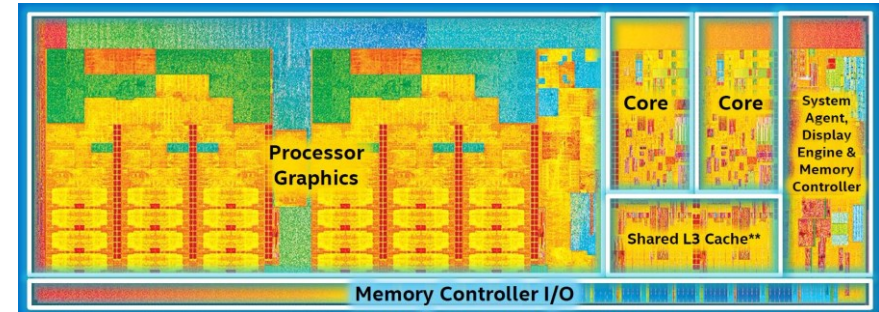
Connections to the outside world: Memory and PCIe slots



# Regular PC processor die



Haswell-generation chip for regular laptop/desktop, same scale as on previous slide.



Broadwell generation chip for low-power laptops, e.g., a 2015 MacBook Air (1 generation later), same scale.



# A CPU package

CPU die: invisible, sits between the heat spreader and the substrate

heat spreader



Substrate: routes electric contacts from the chip to the outside world, and can contain other components or even multiple chips (dies)

2011 pins to connect to the motherboard

capacitors for denoising power supply



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 3: The memory hierarchy

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)

# The memory performance gap

## ➤ Evolution of computers 1996-2023:

- latency (time to first bit arriving) of memory subsystem:
  - 1996: Pentium Pro with PC100 SDRAM: 300 ns or 60 clock cycles
  - 2023: AMD Genoa with DDR5 memory: 120 ns or 288 clock cycles
- processor chip memory bandwidth
  - 1996: Pentium Pro: 64-bit 66 MHz bus: 0.52 GB/s
  - 2023: AMD Genoa: 12 64-bit channels @ 4800 MHz DR: 461 GB/s
- floating point speed (double precision)
  - 1996: Pentium Pro, 200 MHz, 0.2 Gflops peak DP
    - 300 ns = 60 flops
    - 200 Mflops = 1.6 GB/s results generated
  - 2023: AMD Epyc 9654: 96C, 2.4GHz: 3686 Gflops peak DP
    - 120 ns = 442,320 flops
    - 3686 Gflops = 29,488 GB/s results generated

ns /2.5

cy x4.8

x890

X 18,430

X 3

X 64

# The memory performance gap:

## Speed-limiting factors

- What limits memory speed?
  - larger memory is slower due to physical constraints
    - larger size = longer travel distances in the memory device
    - and also further away from the CPU so longer distance to the CPU
  - and is slower due to economical constraints:
    - faster memory cells are more expensive (latency)
    - can't afford the same number of links to off-chip as to on-chip memory (bandwidth)
- On-die memory can be very fast, but is expensive and limited in capacity:
  - different type of memory cell that takes more space per bit
  - wide paths are possible
- On-package memory: DRAM chips on the substrate
  - very wide bus (Apple M1 Max: 512 bit, AMD MI100, NVIDIA V100 and Fujitsu A64fx: 4096 bit, NVIDIA A100: 5120 bits, NEC SX-Aurora TSUBASA 1<sup>st</sup>/2<sup>nd</sup> gen: 6144 bits, AMD MI250: 8192 bit)
  - but limited capacity

# The memory performance gap:

## A solution

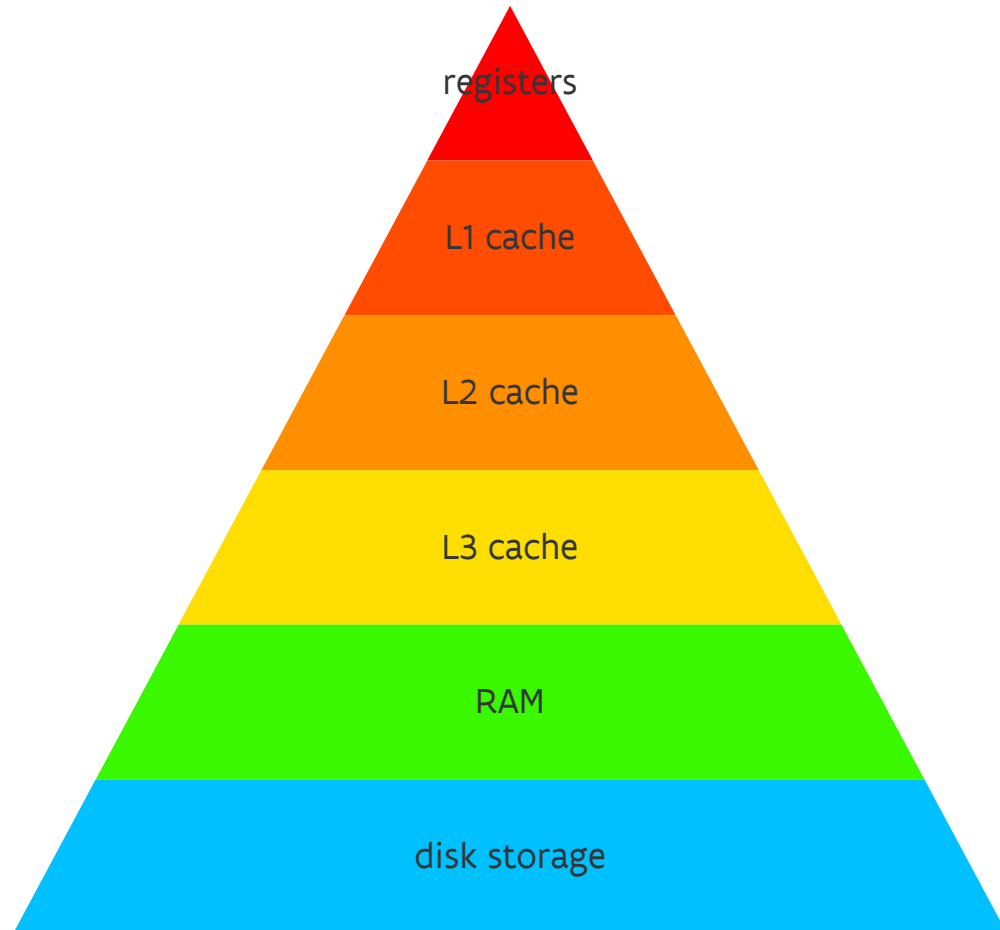
- To mitigate the memory performance gap problem, small memory “buffers” called cache memory have been added to the CPU chip:
  - most caches are transparent to the programmer with respect to correctness: the system manages the caches automatically
  - but not transparent with respect to performance
    - optimize memory access patterns to improve the cache hit rate
- In fact, modern systems have a cache hierarchy with three or four levels
  - a small level 1 (L1) cache, e.g., 32 kB, and often specialized for certain “data”: instructions, integer data, ...
  - a larger but slower L2 cache per core, e.g., 256 kB, 512 kB or even 1 MB or 2 MB
  - a L3 cache shared by all cores, 2-4 MB/core is not uncommon. E.g., the E5-2699v3: 45 MB cache (2.5 MB/core), AMD 7H12: 256 MB cache (4 MB/core), AMD Milan-X has 12 MB/core and more
  - Intel Sapphire Rapids MAX has 64 GB of HBM memory in the package

# The memory performance gap:

## What can we do?

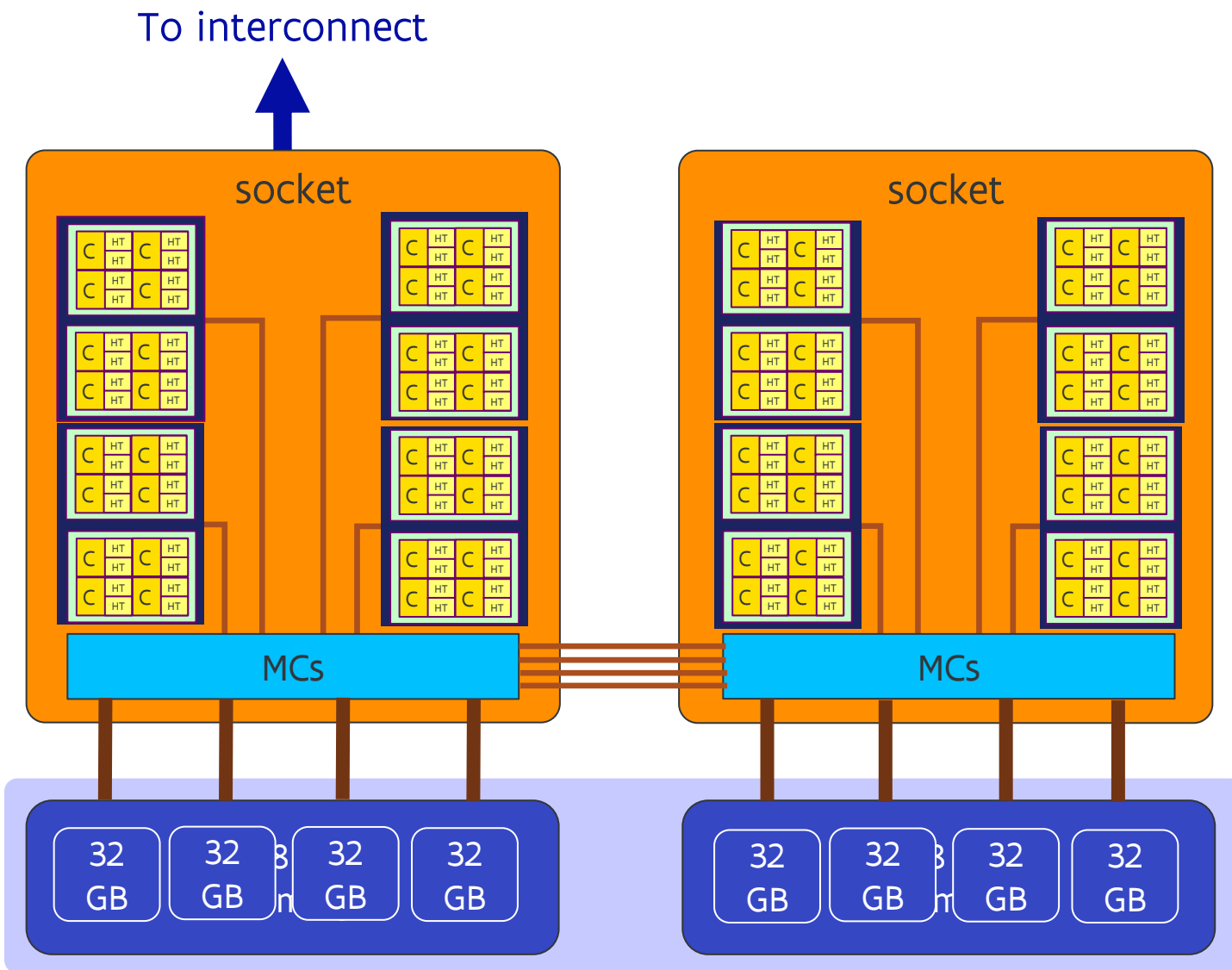
- It is important to organize data access in code in an appropriate way so that data access becomes local and predictable rather than random. Ideally we stream data through the processor.
  - We're not talking about a potential 2x performance increase, but rather 100x or 1000x compared to a really bad code!
  - For linear algebra, FFT, image processing, ..., very good libraries exist, use those and don't start your code from scratch!
    - Example: BLAS – basic vector and matrix operations and used in many other linear algebra libraries
      - Reference implementation in Fortran sucks on modern systems
      - But there are several excellent commercial and free implementations: Intel MKL, AMD Core Math Library, OpenBLAS, Bliss, Atlas, ...
  - Frameworks for other types of applications, e.g., solving PDEs.
- If you use code, it is important to realize that not all code is created equal and big performance differences do occur.

# The memory hierarchy



<b>Leibniz (UA) “Broadwell”</b>	<b>Vaughan (UA) AMD “Rome”</b>	<b>Hawk (HLRS) AMD “Rome”</b>
<1kB/hw thread ~7kB physical	<1kB/hw thread ~6kB physical	<1kB/hw thread ~6kB physical
2 x 32 kB/core	2 x 32 kB/core	2 x 32 kB/core
256 kB/core	512 kB/core	512 kB/core
35 MB/socket	128 MB/socket 16 MB / 4 cores	256 MB/socket 16 MB / 4 cores
128 GB/node (144) 256 GB/node (8)	256 GB/node	256 GB/node
no swap 600 TB 152 nodes	no swap 600 TB 152 nodes	no swap 26 PB 5632 nodes




# Vaughan (AMD Rome) node revisited




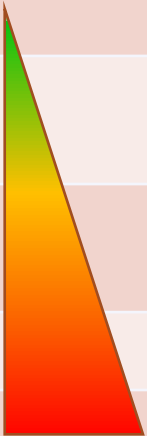

- L1 and L2 cache is private per core
- L3 cache shared per 4-core core complex (CCX) or per 8 cores on LUMI (AMD Milan)
  - Fast access to buffer memory of own CCX, but almost as slow as main memory access otherwise.
- AMD Rome: 2 CCX per die, share a link
- Memory controller in 4 quarters, each quarter serves 2 or 4 CCX (Rome) or 1 or 2 CCX (Milan)



# Combined hierarchy: AMD Rome (Vaughan and Hortense)

hierarchy layer		per	sharing	distance	data transfer delay	data transfer bandwidth
1	2 HW threads	core	L1I, L1D, L2			
2	4 cores	CCX	L3			
3	2 CCXs	CCD	link to I/O die			
4	1 (Va) / 2 (Ho) CCDs	NUMA node	DRAM channel (and PCIe lanes)			
5	4 NUMA nodes	socket	inter-socket link			
6	2 sockets	node	inter-node link			

# Combined hierarchy: AMD Milan (LUMI)

hierarchy layer		per	sharing	distance	data transfer delay	data transfer bandwidth
1	2 HW threads	core	L1I, L1D, L2			
2	8 cores	CCX	L3 Link to I/O die			
3	2 CCDs	NUMA node	DRAM channel (and PCIe lanes)			
4	4 NUMA nodes	socket	inter-socket link			
5	2 sockets	node	inter-node link			



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 4: Storing data on supercomputers

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)

# Files on a supercomputer

- Just as physics make it impossible to build a single processor core that is 1,000 times or 1,000,000 faster than a regular CPU core, it is not possible to make a hard disk that spins 1,000 times faster and has a capacity that is 1,000 times more than current hard disks.
- So how do we build a storage system for a supercomputer?
  - Same idea as for CPUs: build a large and fast system out of 100s or 1,000s of regular disks
  - And use clever software to make this work as if it is one large and very fast disk
  - Also used to make fast SSD drives for PCs and smartphones (which is why the 256 GB M2 13” MacBook Pro has slower storage than the 512 GB one).
- But just as not all programs can take benefit from multiple processors, not all programs can take benefit of such a disk setup.
  - A parallel setup only works when programs access large amounts of data in large files
  - And even though the parallelism can increase the bandwidth, the latency is still limited by that of each device

# Problem 1: Disks break

- Drives fail quite often (and flash-based SSDs aren't much better than hard disks)
  - If you have 1,000 drives, you can expect one to fail on average every 50-100 days.
  - Losing some data every 50 days is already quite bad...
  - But since all disks operate like one giant disk with files spread over multiple disks, you may actually lose a significant amount of data.
- Solution:
  - Use some disks to store enough information to recover lost data on another disk by using error correcting codes
  - Implies using larger “blocks”:
    - e.g., 8+2 configuration: “Block” size will be 8 times that of a regular disk, so typically 32 kB instead of 4 kB

# Problem 1: Disks break

- Performance consequences for writing data:
  - Whenever writing you need to update data on at least three disks (assuming 8+2 configuration),
  - But to be able to do that correctly, you'd have to read more information first to be able to update the parity/ECC information.
  - Unless of course you write a whole 32 kB-chunk at once.
- For reading data you can already benefit from the fact that data is on multiple disks that are read in parallel
- Similar techniques are also used, e.g., to protect RAM memory in servers and supercomputers or internally in flash memory drives

## Problem 2: File system block size

- A file system organizes files in one or more “blocks”. They are the smallest element a file system can allocate and manage.
- On a supercomputer that FS block size can be much larger than on a PC
  - PC: used to be 512 bytes, but on current file systems often 4 kB.
  - On a cluster, the block size is sometimes much larger (depending on the file system), e.g., 128 kB
    - Large disks, small block size = too many blocks to manage efficiently.
    - Larger “blocks” better fit with RAID (using multiple disks in a clever way to compensate for a failed disk).
- So a 100-byte file will really occupy 128 kB on file system with 128 kB “blocks” (and a little more since it also takes space in the file system table).
  - Extreme case: User who used 642.5 kB to store what was really 36 bytes of data in his program: 4 FP numbers and one integer and was not even storing the full precision of the FP number.
- Example: Previous UAntwerpen GPFS file system (now IBM Spectrum Scale)

## Problem 2: File system block size (2)

- Some systems (BeeGFS, Lustre) have a 2-level hierarchical architecture to deal with that.
  - Storage distributed across many object storage targets/servers
  - But each object storage target/server is more reasonably sized so can use smaller block sizes
  - Hierarchical structure:
    - File cut in chunks, Last chunk of a file may have a different size (but a multiple of the block size)
    - Each chunk is stored in a number of blocks of a single object storage target
      - Chunks of a file on the same object storage target are grouped (in a file)
      - Example: chunk size of 1 MB, stored across 2 object targets: First megabyte on first OST, second on second, third again in the same file on the first server, etc.
  - Users may need to help the system to choose the right chunk size and number of object storage targets used as the optimum depends on characteristics of both storage and file.
  - And parameters in file I/O libraries should match those used by the file system.
- Both types of file systems sometimes have provisions for dealing with very small files.



# Problem 3: Physics and the network

- Supercomputer storage sits much further from the CPU as the local drive on your laptop both physically and logically
  - Your program on your laptop can directly contact the file system of the OS and get the data from the drive
  - Shared networked storage adds layers and hence latency: network file system to network stack to network stack on the server to network file system server to regular file system and then back.
    - Parallel file systems may have a slightly optimized and shorter route but it remains much longer than to a local drive
- Consequences
  - Programs that open and close hundreds of small files sequentially in a short time may work slower than on a PC as your program will be waiting for data all the time
  - Unpredictable file access patterns are also to be avoided as any logic to prefetch data and hide the latency will fail

# Problem 4: Metadata

- The directory contains information about each file: name, access rights, some type info, data and time of creation or last access, ...
  - And a directory is also a special kind of file by itself.
  - So if you do many small disk accesses or store a lot of files in a directory and access them simultaneously, you'll create a bottleneck because a lot of information in the directory needs to be updated continuously.
- The typical problem scenario: A distributed memory application where each process creates a small file in the same shared directory to write its data, rather than use parallel file I/O to read/write a whole bunch of data in one multi-node file system operation.
  - Try to do this on a 200k core cluster and you'll be the system administrator's best friend for sure.
- And equally stupid: Open a file before every read or write and close it again immediately.

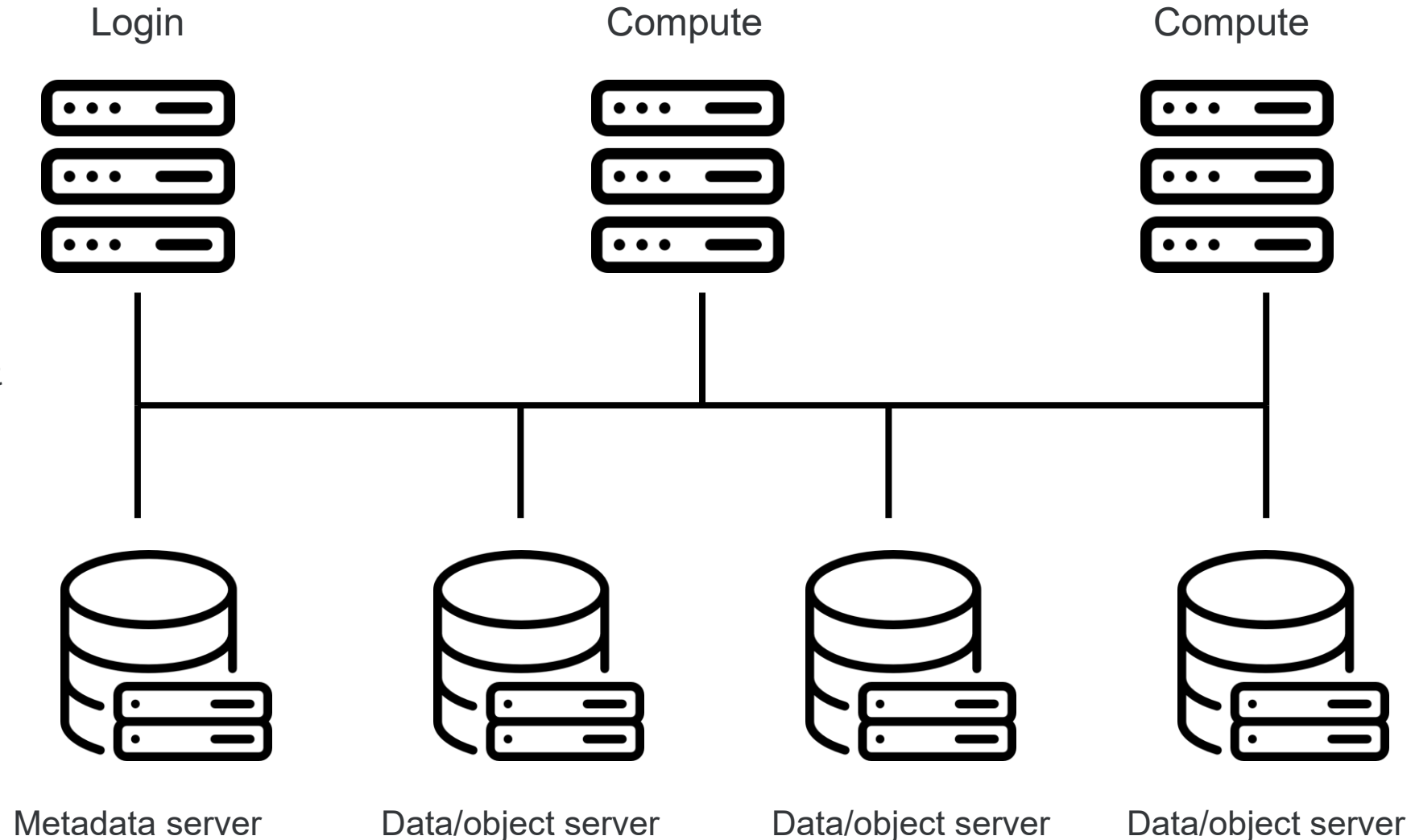
# Higher bandwidth through separation of data and metadata

- PC file system and many regular file server file systems:
  - Each block on disk can be used for either metadata or data
  - Flexible: small file sizes well supported
  - But difficult to get very high bandwidth unless very expensive storage technologies are used because of a single server per file bottleneck
- Supercomputers need a different kind of file system for high performance: a parallel file system
  - BeeGFS (new UAntwerp storage), Spectrum Scale/GPFS (e.g., VUB, UGent), Lustre (new KU Leuven storage; hortense; LUMI), Weka, VAST, ...
  - Separate metadata servers
    - Still a potential bottleneck in metadata access
    - Space determined at purchase, and often on fast and expensive storage
  - Data / object servers for the data
    - Once a file is opened, different processes of a parallel job can pump data in parallel to multiple data/object servers

# Higher bandwidth through separation of data and metadata

E.g., reading a file:

- Client queries MDS for file
- MDS returns location and layout
- Client uses that information to know which OSSes to talk to
- Client(s) requests file content from the OSSes
  - Multiple clients can talk to multiple OSS simultaneously



# Higher bandwidth through separation of data and metadata

## ➤ However:

- The bandwidth gain at the file level is only for large block read/writes in large files
- And the number of files is also limited by the space on the metadata drives
- Some metadata operations are more expensive
  - Certainly true for opening and closing files
  - On Lustre, `ls -l` is expensive too as it needs to talk to the object storage servers to compute the size of a file
- So this storage is not suitable for working with lots of small files or opening and closing files all the time
  - Supercomputer centres are starting to enforcing limits on what users can do on their “fastest” shared file systems!

# A storage revolution?

- Our storage on the cluster is relatively slow
  - Sustained joint bandwidth on /scratch on the order of 7-8 GB/s
  - Comparison: The fastest SSD in my 7 year old PC at home gets 1.5 GB/s write, 2.5 GB/s read bandwidth. Modern PC SSDs claim up to 10 GB/s read bandwidth when put in the right system (very few processors can handle this) and accessed in the right way.
- So why don't we use 120 big SSD drives rather than 120 hard disks for /scratch?
  - Don't expect 120 times 10 GB/s from such a setup unless you're prepared to buy more storage servers as the bandwidth a single server can deliver is limited
  - Don't expect 120 times 10 GB/s from such a setup unless you have proper file access patterns as many problems persist even with SSD storage
  - Moreover, there are two major problems with the SSD drives themselves...

# Flash drives (February 2022, price update September 2023)

	Seagate Exos X20	Seagate Nytro 3732	Seagate Nytro 3332	Samsung 980 Pro	Samsung 970 EVO Plus	Samsung 870 QVO
Technology	Spinning magnetic disks	3D eTLC NAND flash	3D eTLC NAND flash	TLC V-NAND flash	TLC V-NAND flash	QLC V-NAND flash
Market	datacenter (SAS)	datacenter (SAS)	datacenter (2x SAS)	prosumer (NVMe)	consumer (NVMe)	consumer (SATA)
Capacity	20 TB	3.2 TB	15.36 TB	2 TB	2 TB	8 TB
Read speed	0.28 GB/s	1.1 GB/s	1.05-2.1 GB/s	7 GB/s	3.5 GB/s	0.56 GB/s
Write speed	0.28 GB/s	1 GB/s	0.95-1 GB/s	5.1 GB/s	3.3 GB/s	0.53 GB/s
Latency	4,16 ms	50 µs ???	50 µs ???	50 µs ???	50 µs ???	100 µs ???
Endurance	?	58.4 PB	28 PB	1.2 PB	1.2 PB	2.88 PB
DWPD	?	10	1	0,33	0.33	0.2 (@5 year)
Data written/day	?	32 TB/day 8h50m	15.3 TB/day 4h15m	0.66 TB/day 2m9s	0.66 TB/day 3m20 s	1.5 TB/day 50 m
Price	0.025-0.05 €/GB	0,85 €/GB	0,31 €/GB	0.08 €/GB	0.06 €/GB	0.04 €/GB

# Is flash a storage revolution?

- Flash memory has
  - A durability issue
    - Current cheaper high-capacity flash chips can only handle 150-500 writes of every block of cells (and the better ones maybe around 1000 writes)
  - and a price issue: 10x regular hard disks in a cluster setup if you want good endurance
- Unpredictable slow-down under random small write load when the drive starts to fill up
  - Hard disks have a fragmentation problem that may reduce read and write speed, but contrary to popular belief, flash-based SSDs are not at all better when it comes to such write operations.



# Is flash a storage revolution? (2)

- At some point there was hope for better permanent memory technologies with much better write endurance and better access properties than flash (byte-addressable as regular memory instead of block-addressable) but development has stopped due to economic constraints
  - 3D XPoint (Intel, Micron)
  - memristor (HP, SanDisk)
- It looks like large database systems will be better served by RAM-on-CXL (big chunks of RAM but connected through a longer distance connection with higher latency and lower bandwidth), and battery backup.
  - But I have doubts despite the claims of some vendors who talk about composable hardware that this is a technology for HPC as our users have now already problems dealing with latency and NUMA characteristics of memory

# File system: To remember...

- Supercomputers like large files and large reads or writes. Just as with memory, streaming data to and from a file is much faster than random access.
- Avoid writing many small files
  - Running 1000s of small jobs, e.g., for a parameter study: Don't keep many small files per run until the post-processing phase, but accumulate the data right away in a large file
  - And there are technologies to help with that, e.g., databases (e.g., SQLite3) and HDF5 files
- Avoid opening and closing files all the time as this involves additional metadata operations
- Use MPI-2 parallel I/O or libraries such as HDF5, netCDF, ADIOS or SIONlib (or look for codes that use them) when working with large amounts of data
  - Think of it as creating a hierarchy: the file can act as a kind of file system for the data that belongs together.
- Avoid writing large text files. Binary files are as portable as text files nowadays, are more compact and if one knows the data structures written to it, one can easily compute where in the file what data should be, and reading and writing is a 100 times or more faster

# File system: To remember... (2)

- Scaling capacity is cheap
  - Often one only needs to add disk enclosures and disks, not so much servers, as there are drive interfaces that scale to a very high number of disks (like SAS).
- Scaling bandwidth is harder and more expensive
  - Adding disks is not enough, we need to add file servers too as each server has a finite bandwidth
  - But a single application can only benefit if it exploits parallel I/O.
- Scaling I/O Operations (IOPS) is the most expensive
  - Metadata access is much harder to parallelize, especially for access by a single user or to a single directory
  - Higher latency compared to an SSD in a PC because of additional layers of software and network access limit sequential IOPS
  - Higher total IOPS does not mean higher IOPS for a single threaded application with synchronous file access!

# 2020 UAntwerp storage system

- Linux home directory is on flash storage but have a very small capacity allocation to users which helps prevent abuse (roughly 3.5 TB)
- Disk volumes that are only written to by system managers are on flash storage: roughly 18 TB
  - Mainly for applications
- Regular file system on hard drives (roughly 50 TB)
  - Space for more permanent data and some special use cases
- A large parallel file system (BeeGFS) on hard drives (roughly 0.7 PB)
  - 120 hard drives for the object storage
  - SSD for the metadata
  - Optimized for larger files and bigger writes. We do note that some metadata operations are relatively expensive on this file system.



Vlaanderen  
is supercomputing

# Supercomputers for Starters

March 2023

Kurt Lust – CalcUA, VSC and LUMI User Support Team

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 5: Putting it all together (summary session 1)

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)

HPC

=

High-Performance Computing

≠

High-end Personal Computer

# Scaling

- Performance of hardware parts of computers is characterised by many parameters
  - Clock speed of a CPU
  - Latency of connections and various subsystems (e.g., memory and disks)
  - Bandwidth of various elements, compute capacity
  - Power consumption of parts
- Not all these parameters are as cheap to scale or improve over time at the same rate
  - Physical limitations have put a bound to improvements in CPU clock speed and latencies
  - Speed of light and speed of signals in copper wires is finite
  - Bandwidth growth of memory, disks and network connections tends to be slower than the growth of quoted peak performance of a computer system
- As a result it is not possible to build a computer where all those parameters are 100x better than in a PC or smartphone
  - For some work a **High-end PC** is unbeatable because of its compact size and thin software layers as it is a personal device



# Dennard scaling

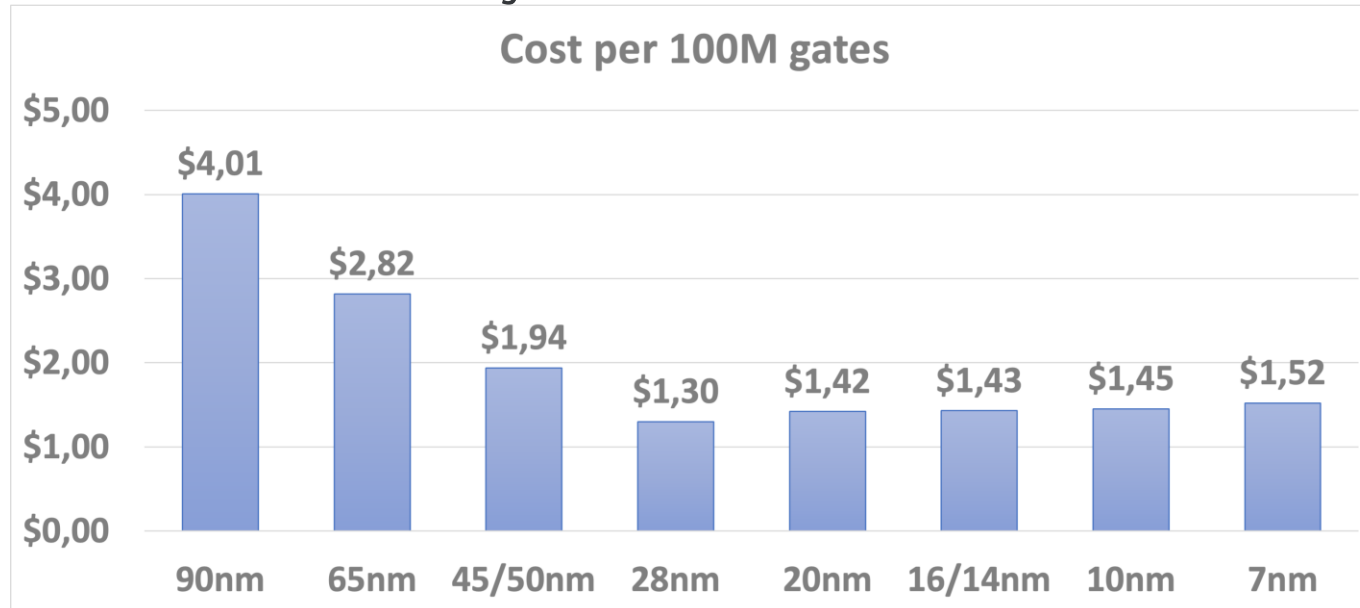
- For a long time, with every new generation of chip technology
  - Linear dimensions decreased by 30%
  - Surface dimensions decreased by 50%, i.e., transistor density doubled
  - Power density remained the same (as voltage and currents are proportional to linear dimension)
  - Circuit delays went down by 30%, so frequencies went up by 40%
- This broke down around 2006 though
  - Not all dimensions of all elements scale as well, so transistor density does not grow as fast anymore
  - Threshold voltage of semiconductors becomes relevant
  - Leakage power becomes dominant
  - Clock frequencies don't go up as fast anymore

## Dennard scaling (2)

- As a result of the breakdown of Dennard scaling
  - Chips have become very hot and power consumption of supercomputers a major concern
  - Hardly any further speed increases just from further reducing component size
    - So need to look harder for architectural improvements than before
  - Part of the reason why latencies of various components do not improve anymore
- Transferring data at high speeds also requires considerable power
  - Nowadays transferring two numbers from one end to the other end of a chip requires more power than a computation with those numbers
- PCs already operate in the domain of Dennard scaling breakdown, so no hope for a single processor that is much faster than that in a PC...

# Cost per transistor

- Data from the Marvell 2020 Investor Day:



- Processors still become faster, but price/performance of computers isn't improving as much anymore
- Need to find ways to do more work per transistor
  - So more architectural innovations needed...
- Or use better software to get more out of our hardware budget

# Keywords: Parallelism, hierarchy and streaming

- There are three keywords when developing software to obtain high performance
  - And today those three keywords are relevant even for PCs
- **Parallelism:**
  - Processor performance relies on parallelism through instruction level parallelism and vector/matrix computing
  - System level processing performance relies on parallelism through the use of multiple processors in shared memory and distributed memory setup
  - Memory performance in fact also relies on parallelism in memory accesses. A single core cannot saturate the memory bandwidth of a modern system (not even on a PC)
  - Storage performance relies on parallelism
    - Multiple devices accessed simultaneously to reach high bandwidth (RAID, parallel filesystem)
    - Processing multiple I/O requests simultaneously (and this is even more important for flash memory than for hard disks)
  - And this is not a new lesson
    - Supercomputers have employed parallelism that needs programmer help since the '70s.
    - PC's: Vector computing since the mid '90s, multicore since 2006

# Keywords: Parallelism, hierarchy and streaming (2)

## ➤ Hierarchy:

- Memory hierarchy: (typically) 3 levels of cache, then two or more levels of RAM memory
  - AMD CPUs have an even more pronounced hierarchical structure than Intel CPUs
- Hierarchy in parallelism in processing:
  - ILP and vectorisation at a very fine scale
  - Shared memory parallelism
  - Distributed memory parallelism
- Not yet discussed: GPUs also have a very hierarchical structure, both in hardware and low-level programming models
- Expect storage to also become more hierarchical than it is today
  - And storage formats such as netCDF, HDF5, ADIOS, ..., already create a hierarchy
- Exploiting the hierarchy is important, as is a proper mapping of the parallelism hierarchy onto the memory hierarchy
- And this is not a new lesson either. E.g., caches became an issue 40 years ago.

# Keywords: Parallelism, hierarchy and streaming (3)

## ➤ Streaming:

- Getting data flowing smoothly through the memory hierarchy, all the way from permanent storage to processing, is key to performance
- Data access should be in sufficiently large chunks
  - so that effective bandwidth is not reduced too much by latency and
  - so that no data in caches is wasted
- Data accesses should be predictable so that prefetching can work to further hide latency
- So random access to small blocks of data and to lots of small files is the worst thing you can do
- This is not a new message either, some level of streaming has been important since the '70s...

# Andy and Bill's law

What Andy giveth, Bill taketh away

## ➤ Context

- Andy Grove: CEO and later chairman of Intel, 1987-2004
- Bill Gates: CEO and chairman of Microsoft, 1975-2000
- In the '90s performance of microprocessors grew so fast that efficiency of software became an afterthought
  - The rise of ever more goodlooking GUIs started in that era
  - Software bloat with packages of which 90% of the people use only 10% of the features
  - Groves frustration was also that Bill Gates was so slow to exploit new features of his CPUs
  - But also in scientific computing...

# Andy and Bill's law (2)

What Andy giveth, Bill taketh away

➤ Variants for scientific computing:

- What Andy giveth, Cleve taketh away

The rise of Matlab

- What Andy giveth, James taketh away

Languages that hide how data is treated such as Java

- What Andy giveth, Guido taketh away

Python also saw the light of day in the '90s though it only became more popular for scientific computing around 2005



# Andy and Bill's law (3)

## What Andy giveth, Bill taketh away

- But we can no longer afford this attitude today:
  - We cannot rely on further improvements of sequential speed to be able to solve ever bigger problems.
  - In fact, we can not even any longer rely on a fast increase of parallel performance/dollar.
  - Languages that give us sufficient control over data storage and data flows, and where parallelism is not an add-on, are important for performance.
- Need to go back to the time when everybody paid attention to good algorithms and to a proper implementation
  - In some fields of scientific computing performance improvements have come as much from better numerical methods as from faster computers...
  - Also in the future much performance improvements will have to come from better software.
  - Quantum computers and optical computers will not save the world anytime soon

# Supercomputing is about software, not hardware

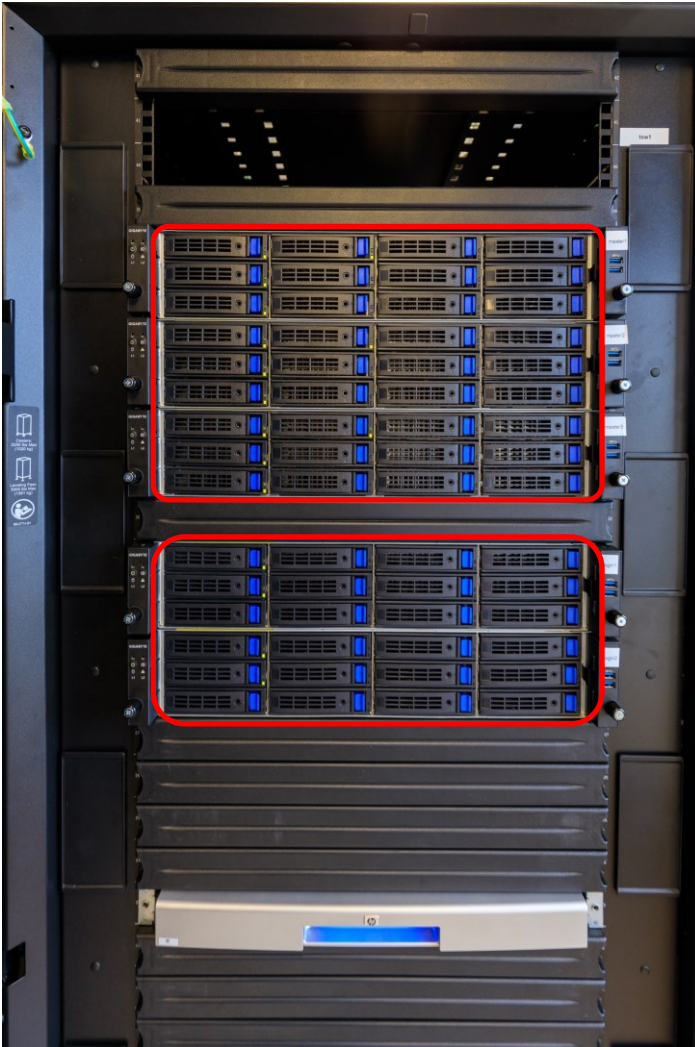
- '60s-'70s: smaller slower computers and bigger faster computers
- '70s-'80s: vector computing: Now programming techniques for supercomputers started to differ from those for regular (non-vector) computers, but supercomputer hardware was very specialised
- 80's-90's: Supercomputers build from variations of standard components to reduce costs, the software made the supercomputer
- This is more true than ever before
  - Supercomputers try to minimize the hardware costs more than ever
  - by using cleverly designed software to turn fairly standard hardware in a powerful computer
  - within limits of course as a certain level of reliability is needed.
  - This again stresses the importance of proper software at all levels (system software and application software) on supercomputers!

# Supercomputing is about software, not hardware (2)

- Supercomputers focus on different aspects than cloud infrastructures
  - Focus on latency and staying close to “bare metal” rather than isolation, security and personal environment
  - Focus on those aspects of scalability that enable capability computing rather than fine grained user control and management
  - Focus even more on hardware cost reduction than cloud infrastructures
  - Different exploitation model focussing on shared storage, on freeing resources for other users as soon as possible rather than long-term reservations for a particular user, and on starting the next task as fast as possible
    - Though this is also partly caused because supercomputer resources are often allocated “for free” after a competition so there is no incentive for a user to think economically

# Assembling the cluster: Access and admin part

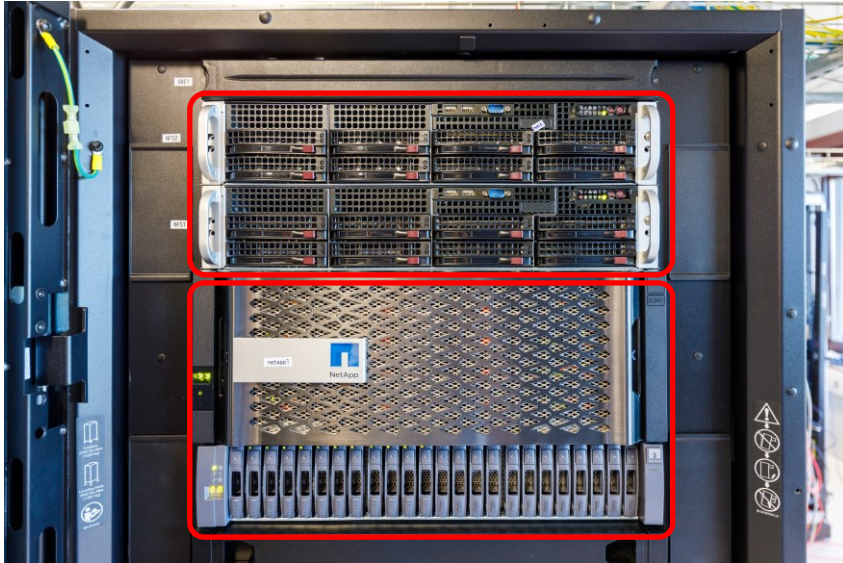
- UA cluster Vaughan
- Really nothing special, just regular servers



Admin nodes

Login nodes

# Assembling the cluster: Storage

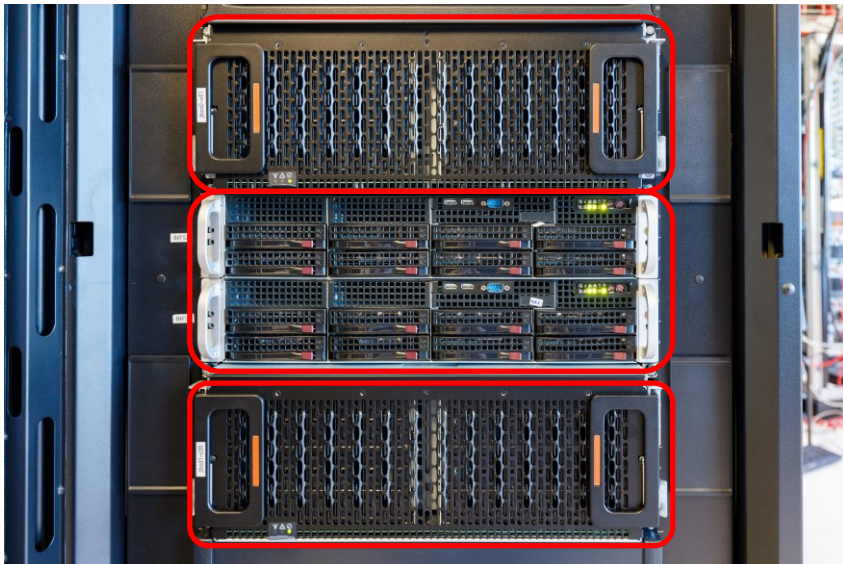


File servers for NFS

- Standard server

Various types of disks

- High-quality storage very popular in IT services
- /user, /apps, /data



Disks for scratch space

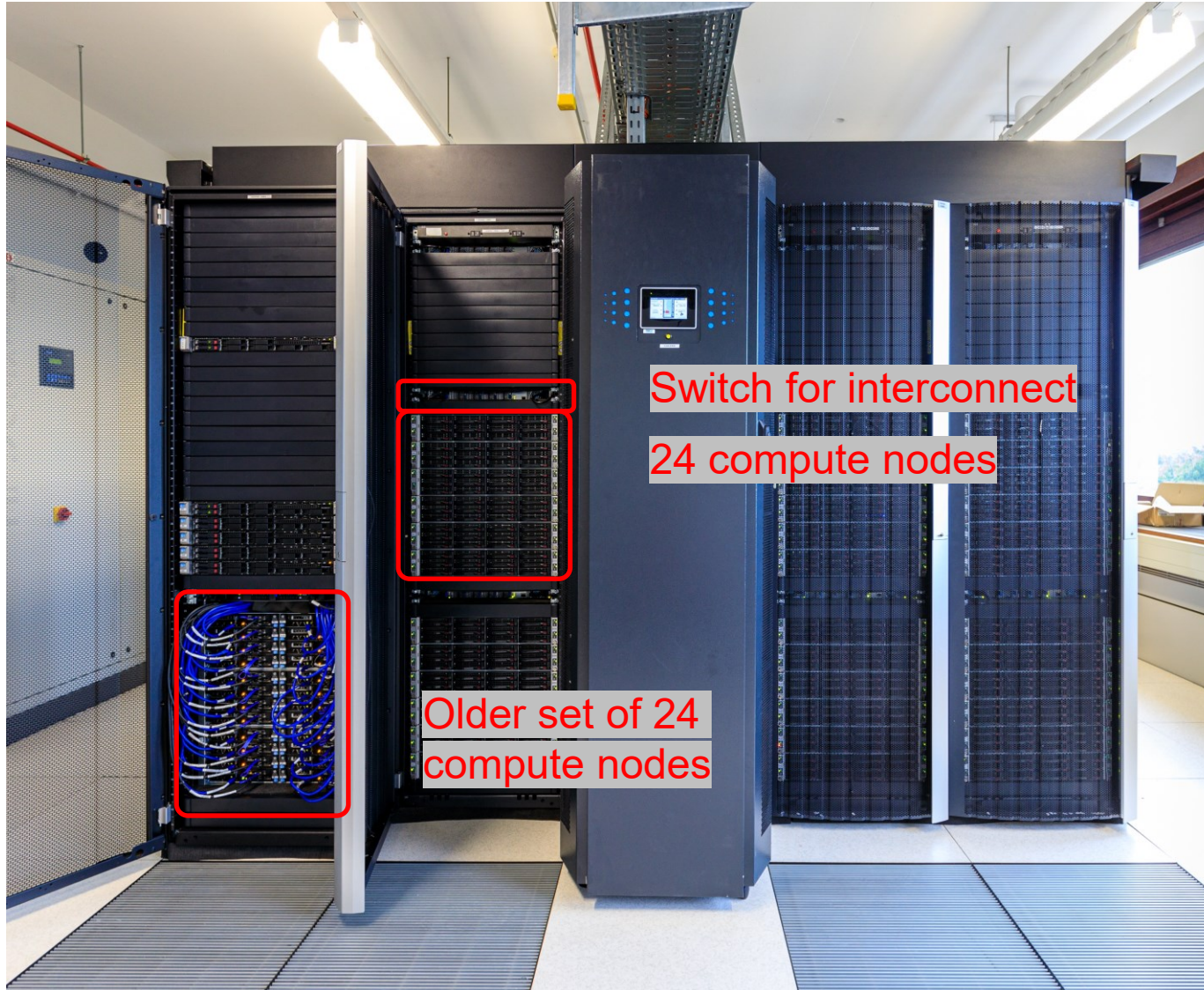
- Cheap array of 60 disks

File servers for scratch space ➤ Standard server

More disks for scratch space

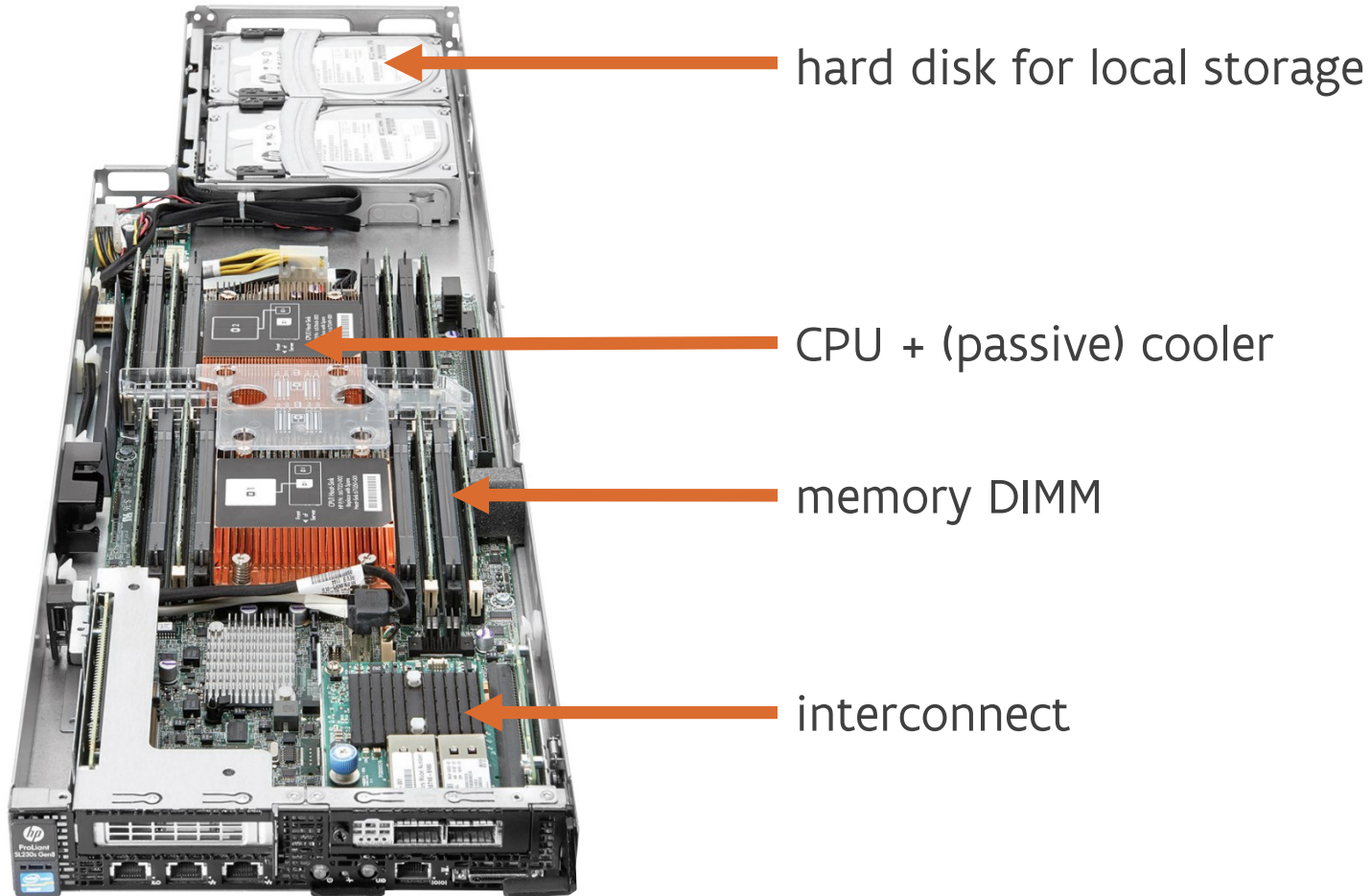


# Assembling the cluster: Compute nodes



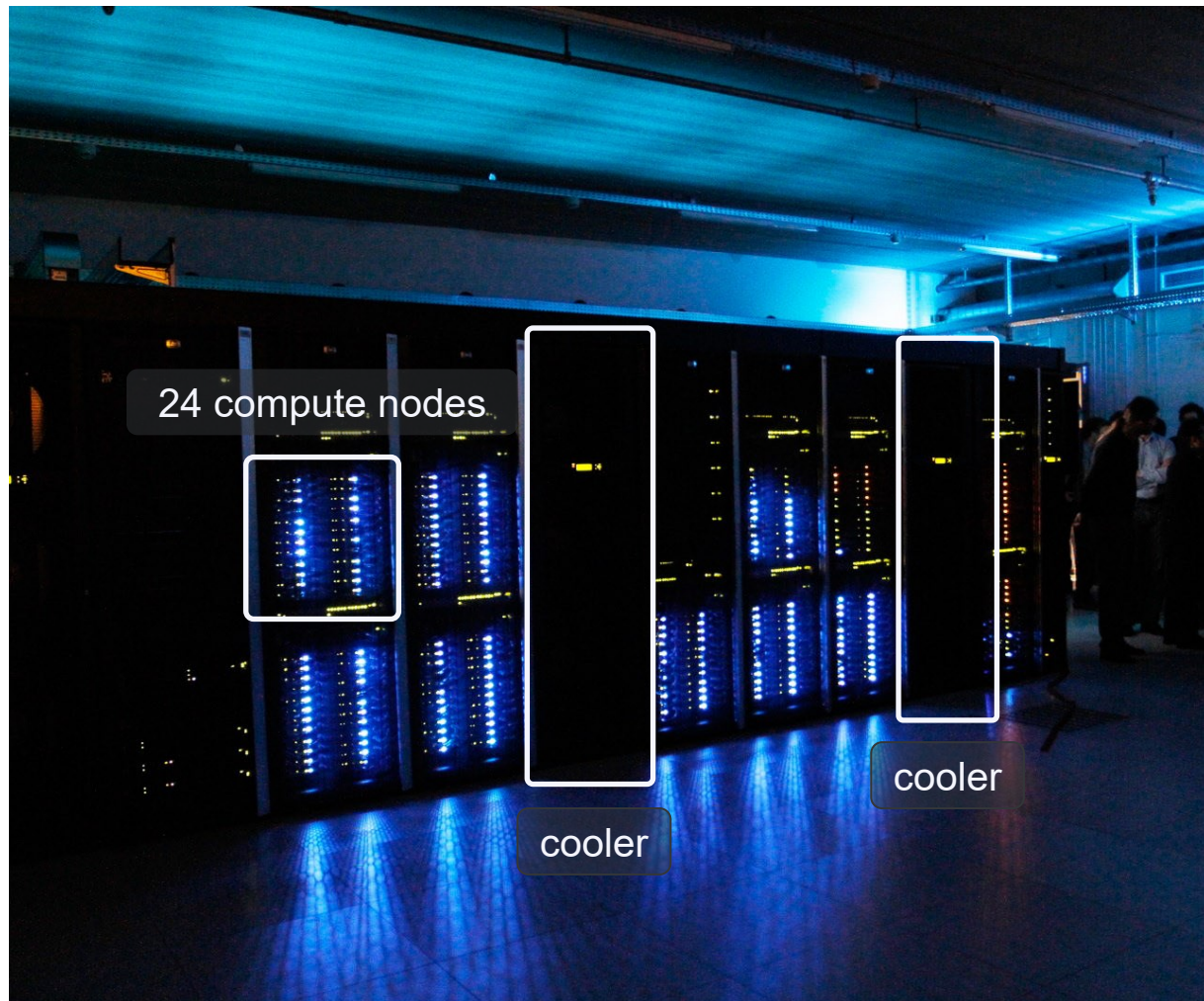
- Switch connects to 24 compute nodes and to “top-level” switches that in turn connect switches
- 4 compute nodes in the space of 1 server for storage or login nodes
- Groups of 24 compute nodes
- See next slide for the node

# Assembling the cluster: An (older) cluster node





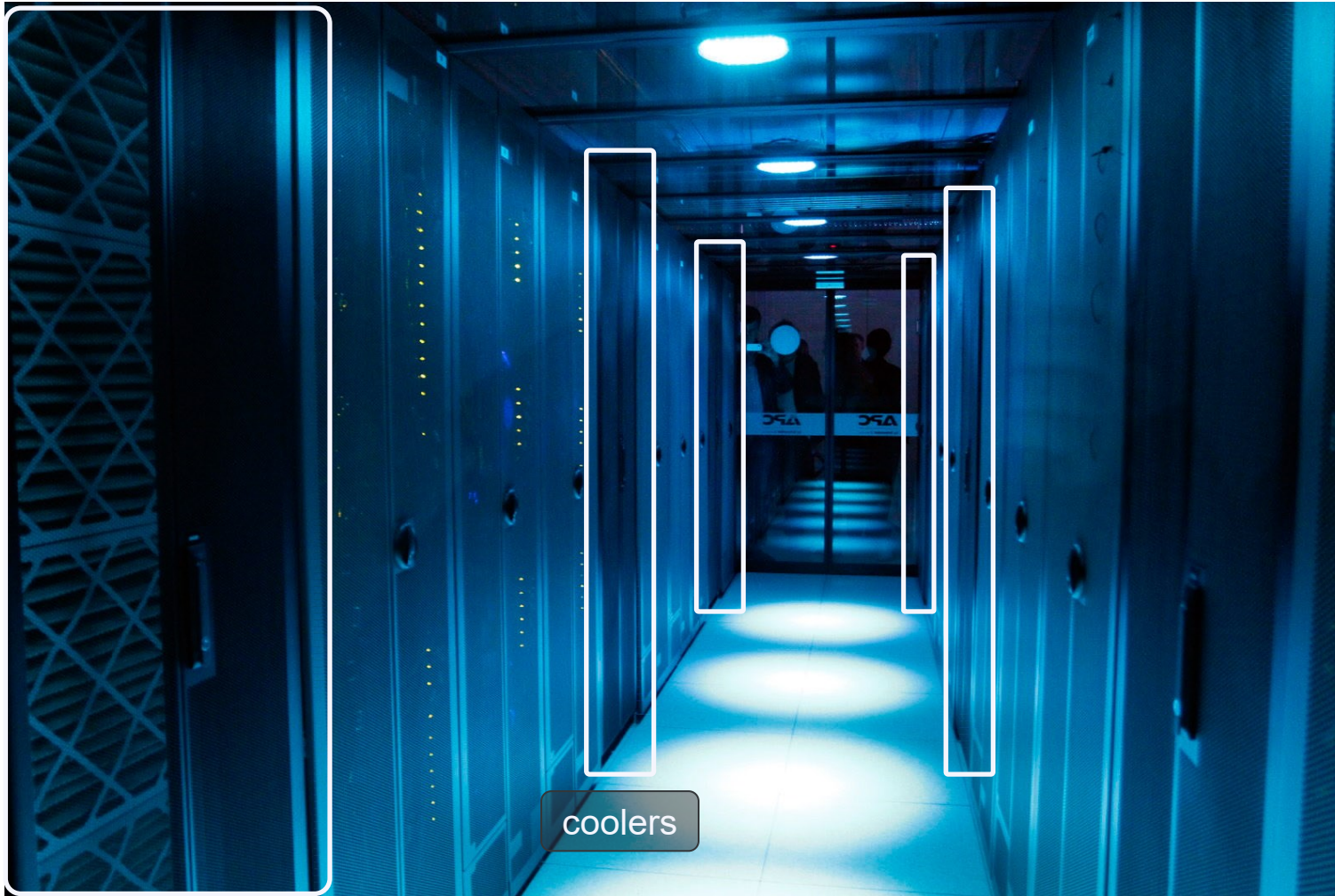
# The complete cluster



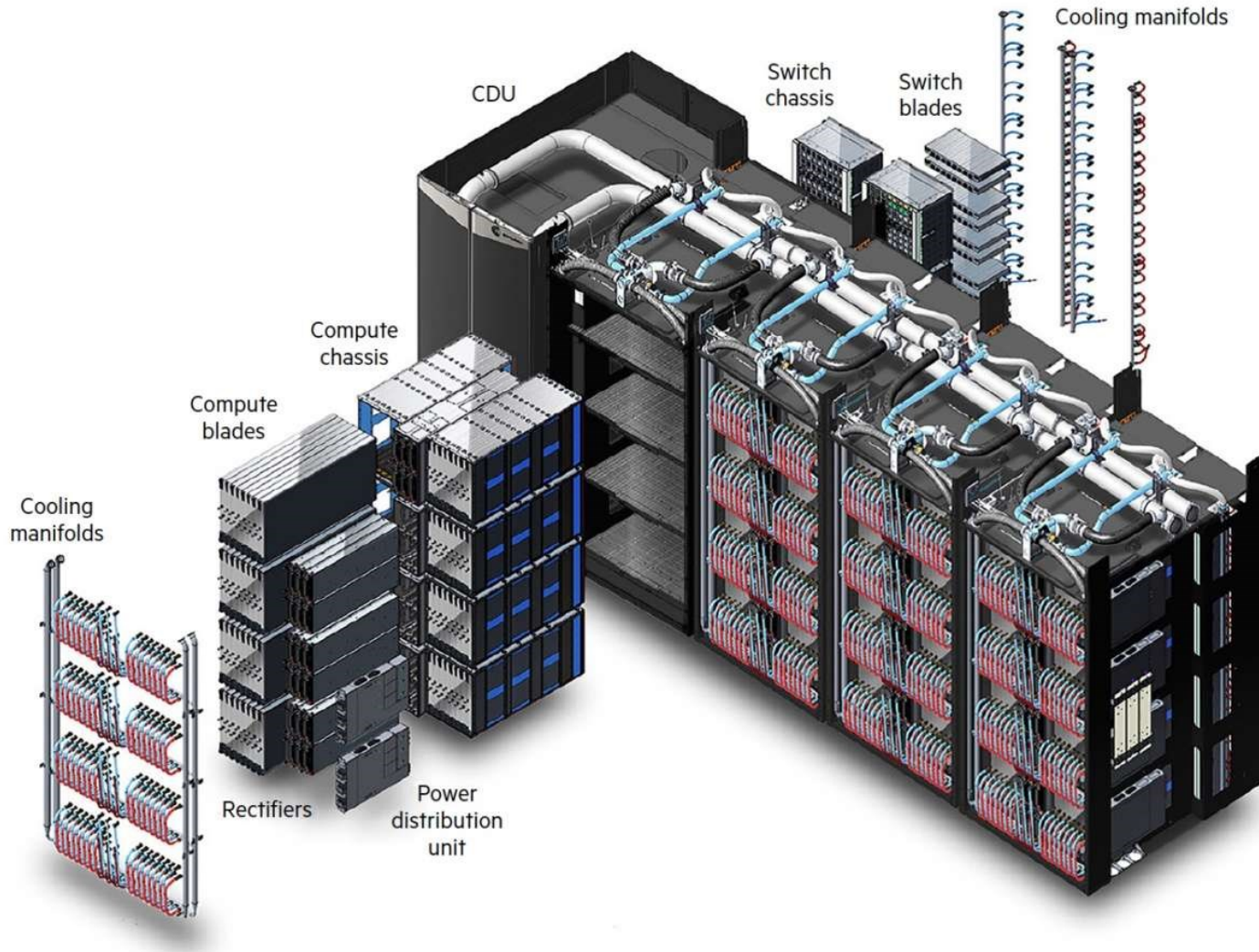
- Older picture of the first VSC Tier-1 cluster, but it better shows the structure



## The complete cluster (2)



# A real supercomputer: Cray EX (LUMI)



## ➤ Switches

- 2 network ports/blade
- 48 ports facing outside

## ➤ LUMI-C: CPU nodes

- 4 nodes/compute blade
- 2 CPUs/node
- 1 network port/node
- 2 switch blades/chassis

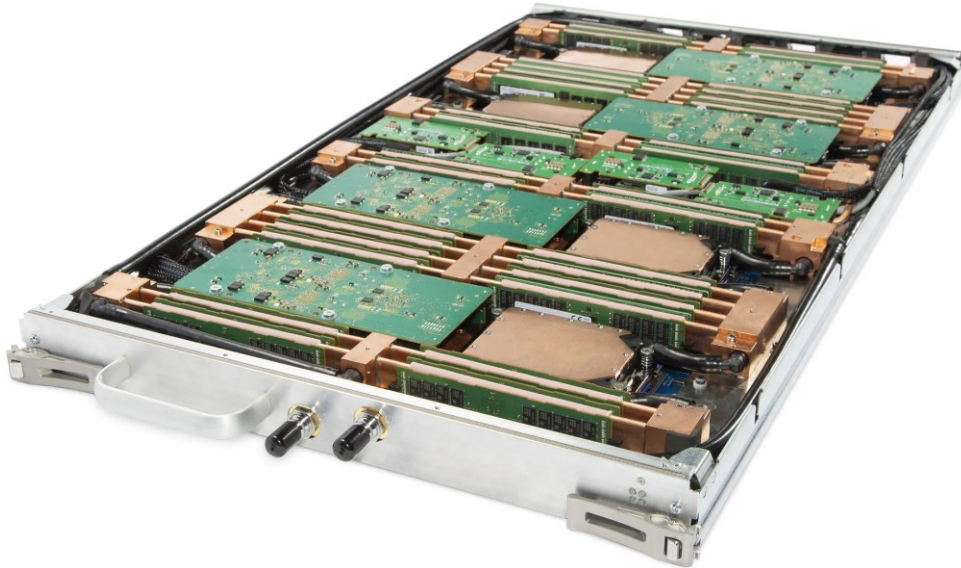
## ➤ LUMI-G: GPU nodes

- 2 nodes/compute blade
- 1 CPU & 4 GPUs/node
- 4 network ports/node
- 4 switch blades/chassis
- 5 kW per blade, >300 kW per cabinet

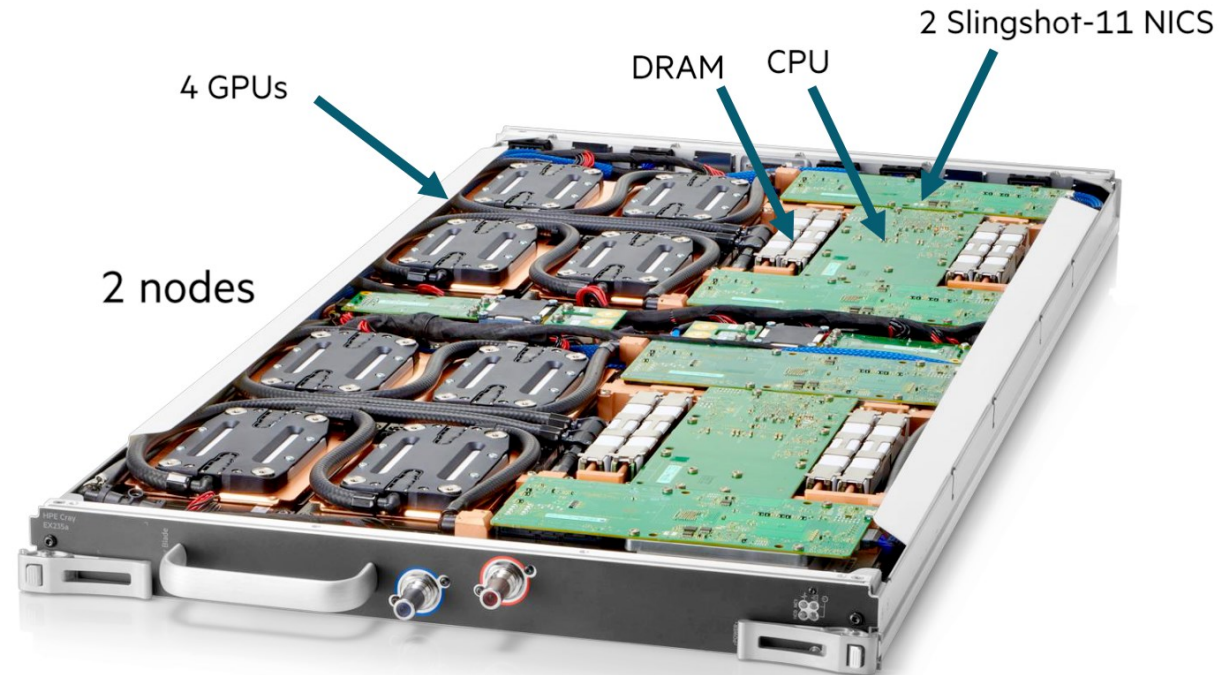


# A real supercomputer: Cray EX (LUMI)

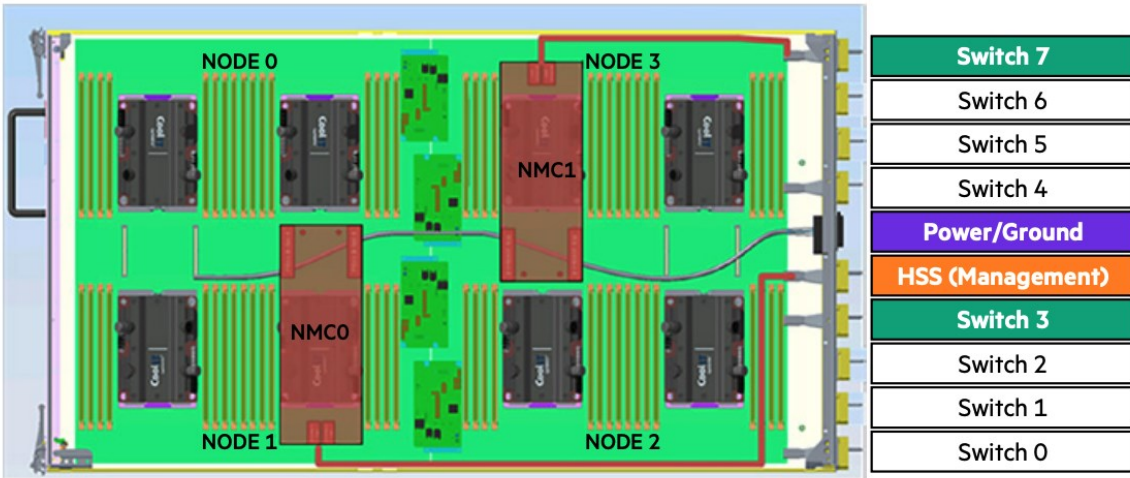
CPU node blade



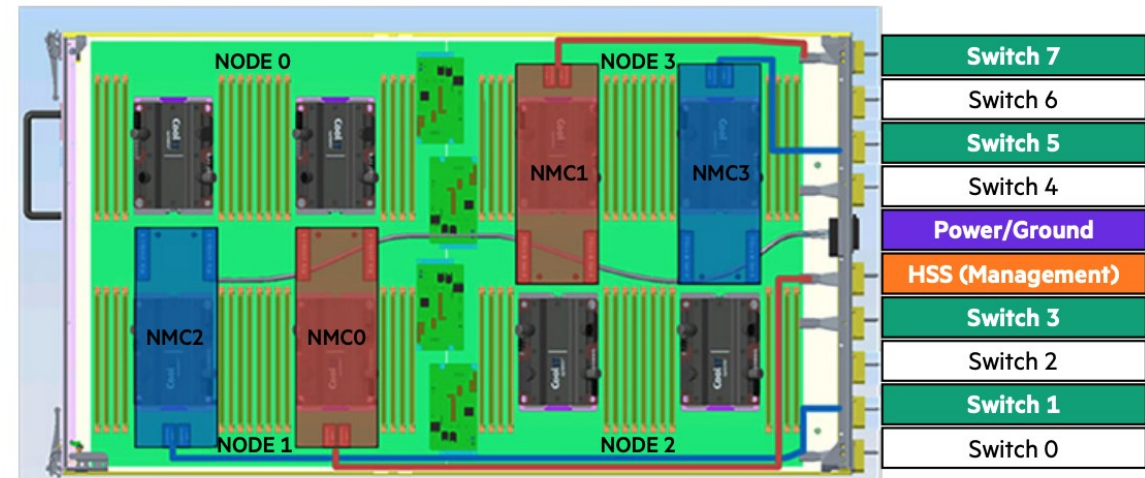
GPU node blade



# A real supercomputer: Cray EX (LUMI)



HPE Cray EX compute blade with single injection port per node



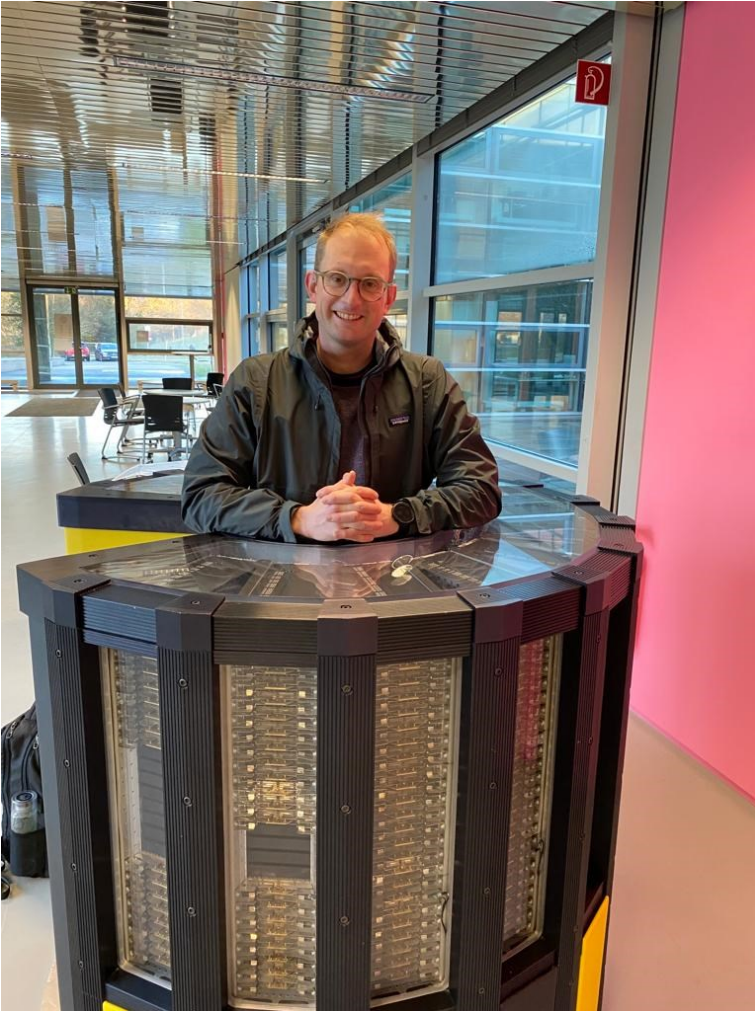
HPE Cray EX compute blade with dual injection port per node



# A real supercomputer: Cray EX (LUMI)



# Comparison: Cray-2



- Cray-2 bought by the University of Stuttgart in 1986
- One of the fastest machines in the world when installed, and still managed to appear at place 250 in the first Top-500 list (June 1993), though the fastest machine in that list was 40 times faster
- Roughly 240,000 chips of which 75,000 memory, spread over 750 packages
- Power consumption close to 200 kW = <2x CalcUA
  - Power consumption roughly 1% of the current fastest system
  - Special immersion cooling system
- Cost on the order of 40-50M\$ in 2023 dollars, just under 10% of today's exascale computers



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 6: Middleware: Turning the hardware into a usable supercomputer



# Why?

- In this section we focus on the software development paradigms

I'm not a programmer, should I know this?

- A supercomputer is more than some hardware + Linux
- In fact, there is a lot of additional software to turn the hardware + Linux into a supercomputer
  - And much of that is part of the programming environment.
- API (library functions, ...) often standardised, but the ABI (binary interface) is not
  - As a result, mixing compilers can be a problem,
  - and getting precompiled binaries to run is sometimes impossible if they were compiled for a different machine
- You have to realise that that software that sits between your application and the hardware
  - has to be ABI or API compatible with your application, and
  - has to be compatible with the hardware and OS kernel drivers/extensions,
  - and hence that software that comes as binaries can be problematic.



# Why?

- In this section we focus on the software development paradigms

I'm not a programmer, should I know this?

- Many scientific applications come as source code.
  - Helps to judge whether the code is ready for modern hardware
  - Helps to figure out which components you'll need on the cluster
- It also affects the way you start programs
  - Start through another program (e.g., almost all distributed memory programs)
  - May need some environment variables to tune the performance (e.g., shared & sometimes distributed memory programs)
- And we can no longer do all software installations for you, there is just too much code with low-quality installation scripts thrown at us.

# Shared memory

- Automatic shared memory parallelisation not very successful
- OpenMP compiler directives:
  - A compiler directive is a hint placed in the code in such a way that it should be neglected by compilers that don't know the pragma
    - C: #pragma
    - Fortran: Look like comments
  - Data-parallel (= each thread works on a part of the data) and task-parallel (= each thread works on a different task)
  - Standard, not vendor-specific, now at version 5.2 (since November 2021)
  - OpenMP 4 was a major revision introducing support for vectorisation and for offload to coprocessors (typically GPU)
  - Influence the runtime behaviour (number of threads, mapping on cores) through environment variables and/or a small set of library calls
  - Supported by the two main compilers in use at the VSC (Intel and GCC)  
Supported by all compilers on LUMI (GCC, AMD aocc/ROCm, Cray)

# Shared memory

- C++: Frameworks such as (Intel) Thread Building Blocks
  - Intel TBB is open-sourced and can be used with other compilers also
- Some languages have thread concepts or other concurrent processing concepts built into the language or its standard runtime library
  - Java: But forget about Java for distributed memory systems
  - C#: Microsoft environment, not really used on supercomputers
  - Go: Language from Google (but not suitable for supercomputers due to its poor memory management)
  - Julia (Matlab/Python alternative with better performance): Threading is a work-in-progress
- Use explicit OS threading, especially for task-based parallelism
  - Linux supports the POSIX standard in the Pthreads library
  - Low-level and cumbersome as you have to do all thread management by hand

# Vector computing

- Automatic vectorization in compilers is moderately successful.
- Vendor-specific compiler directives
  - Work only with that vendor's compiler.
- Standard for directives: OpenMP 4.0 (and later versions)
  - Many consider OpenMP 4.0 pragmas worse than the vendor-specific ones, but improvements are being developed.
  - Work with any compiler that supports the standard.
  - Major criticism: Too much prescriptive instead of descriptive, but 5.0 is a big improvement in this respect
- Use good libraries for your work (e.g., BLAS, FFTW, image processing, ...)
  - See the demo later in this course!
- Using compiler vendor & CPU-specific intrinsics and additional data types in C/C++/Fortran that translate into vector instructions
  - Use with care as you lose portability but could be an option for intensely used kernels.

# Distributed memory

- Automatic strategies through the compiler never made it past the research phase
- Explicit communication through messages most successful model
  - **MPI library** is the most successful one
  - MPI is standardized. This implies that software that compiles with one MPI library should also compile with any other MPI library adhering to the same version of the standard (unless it relies on a specific bug).
  - Many MPI programs skip the shared memory level, using 1 process per hardware thread, but on modern CPUs it may be more efficient to combine MPI with one of the shared memory programming techniques (most often OpenMP) – e.g., 1 process per node, socket or NUMA domain
    - ⇒ **Hybrid MPI/OpenMP applications**, e.g., QuantumESPRESSO, Gromacs, VASP
  - We support 2 MPI implementations on our clusters (and some vendor-specific ones on special machines)

# Distributed memory

- Some languages have distributed memory concurrent computing built into the language, e.g., Julia (Matlab/Python alternative), Charm++
- Partitioned Global Address Space (PGAS) programming languages
  - Distinguish between local and remote memory but allow to use the latter almost as if it is local memory
  - And it is up to the compiler to translate that in messages for the underlying hardware/OS/middleware combination
  - Fortran-derived language: Co-array Fortran, now part of Fortran 2008. Supported in the (classic) Intel compiler
  - C99-derived language: UPC (Unified Parallel C), not part of any standard
  - “New” language: Cray Chapel
  - But performance often sucks
  - Therefore sometimes used together with MPI in a hybrid code
  - SHMEM/OpenSHMEM, GASPI and single-sided communication in MPI-2 are library approaches based on the same idea
  - So far limited popularity (except probably one-sided MPI communications)

# Example applications

- QuantumESPRESSO:
  - Hybrid MPI/OpenMP program
  - Configuration determined through command line options and environment variables
  - Scales quite well when used in the right way
- GROMACS
  - Some modules are hybrid MPI/OpenMP
- SAMtools (bio-informatics)
  - Tend to run single-core by default, but if you check the manual you'd see it can actually exploit shared memory parallelism
  - And this is configured through command line options



Vlaanderen  
is supercomputing

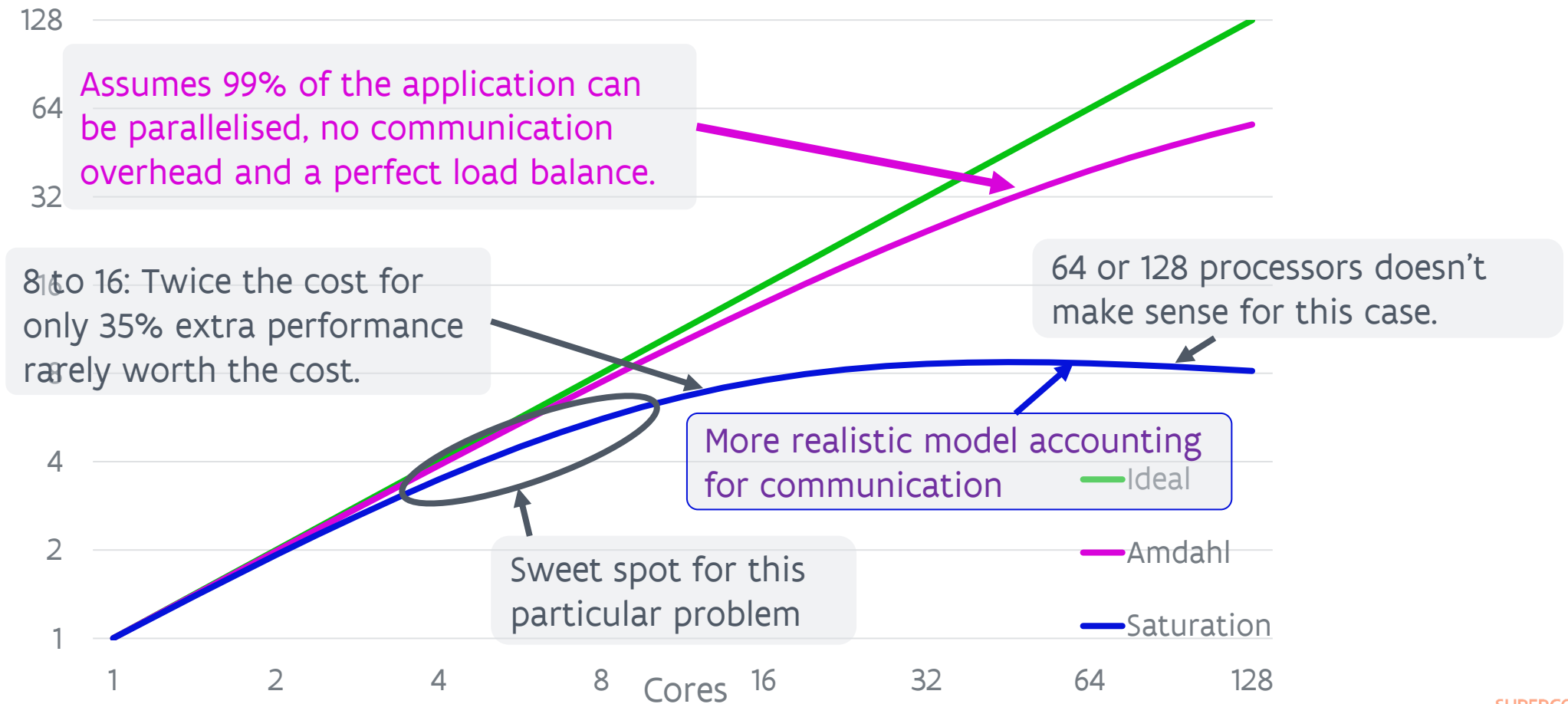
# Supercomputers for Starters

Part 7: What can we expect?



# Speed-up

- Or: how much can you gain by supercomputing?
- Using 100 processors should mean your job runs a 100 times faster, right?



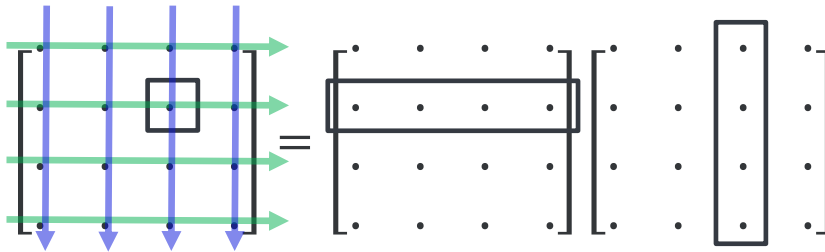
# Speed-up

- Using  $X$  processors will (almost) never speed up your application with a factor  $X$ .
  - There is always some overhead in using multiple cores.
  - There are rare cases though where you may see what is called a superlinear speedup due to cache effects (more cores and nodes = more cache for your program).
- There is no rule like program A runs best on  $X$  cores.
  - $X$  depends not only on the application.
  - But also on the problem being solved. The larger the problem, the larger the optimal number of cores.
  - And it also depends on the cluster: CPU characteristics, interconnect, ...
- Bigger problems = better speed-up

# **Illustration: Matrix multiplication Illustrating cache effect and speedup**

# Matrix multiplication

$$C = A * B$$



$$c_{i,j} = \sum_k a_{i,k} b_{k,j}$$

```
for i = 1 to N
  for j = 1 to N
    c(i,j) = 0
    for k = 1 to N
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

*ijk-variant*

```
for j = 1 to N
  for i = 1 to N
    c(i,j) = 0
    for k = 1 to N
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

*jik-variant*

# 6 variants of matrix multiplication

*ijk*

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
```

*ikj*

```
for i = 1 to N
  for k = 1 to N
    for j = 1 to N
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
```

*kij*

```
for k = 1 to N
  for i = 1 to N
    for j = 1 to N
      c(i,j) = c(i,j)+ a(i,k)*b(k,j)
```

*jik*

```
for j = 1 to N
  for i = 1 to N
    for k = 1 to N
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
```

*jki*

```
for j = 1 to N
  for k = 1 to N
    for i = 1 to N
      c(i,j) = c(i,j)+ a(i,k)*b(k,j)
```

*kji*

```
for k = 1 to N
  for j = 1 to N
    for i = 1 to N
      c(i,j) = c(i,j)+ a(i,k)*b(k,j)
```

# Fortran timings

- GNU Fortran, matrix size 2500x2500 (47.7MB/matrix),  
1 core of Xeon E5-2680v2 = 22.4 Gflops @ 2.8 GHz

Variant	Time (s)	Gflops
ijk	17.16	1.821
jik	24.35	1.283
ikj	63.68	0.491
jki	9.87	3.165
kij	40.77	0.766
kji	13.29	2.352
F95 MATMULT	9.51	3.285
OpenBLAS dgemm (1 thread)	1.27	24.60
OpenBLAS dgemm (20 threads)	0.08	396.42

× 6.5

× 7.7

- To explain these results: Look at memory accesses
  - Fortran stores arrays column by column

# jki-variant

for j = 1 to N

  for i = 1 to N

    c(i,j) = 0

  for k = 1 to N

    for i = 1 to N

      c(i,j) = c(i,j) + a(i,k)\*b(k,j)

i=1:  $c(1,2) = c(1,2) + a(1,3)*b(3,2)$

i=2:  $c(2,2) = c(2,2) + a(2,3)*b(3,2)$

i=3:  $c(3,2) = c(3,2) + a(3,3)*b(3,2)$

i=4:  $c(4,2) = c(4,2) + a(4,3)*b(3,2)$

$c(:,j) = c(:,j) + a(:,k)*b(k,j)$

Works with column vectors

Inner loop for j = 2, k = 3:

$$\begin{bmatrix} \cdot & \boxed{\cdot} & \cdot & \cdot \\ \cdot & \boxed{\cdot} & \cdot & \cdot \\ \cdot & \boxed{\cdot} & \cdot & \cdot \\ \cdot & \boxed{\cdot} & \cdot & \cdot \end{bmatrix} + = \begin{bmatrix} \cdot & \cdot & \boxed{\cdot} & \cdot \\ \cdot & \cdot & \boxed{\cdot} & \cdot \\ \cdot & \cdot & \boxed{\cdot} & \cdot \\ \cdot & \cdot & \boxed{\cdot} & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \boxed{\cdot} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

# jki-variant

for j = 1 to N

for i = 1 to N

c(i,j) = 0

for k = 1 to N

for i = 1 to N

c(i,j) = c(i,j) + a(i,k)\*b(k,j)

k=1: c(:,2) = c(:,2) + a(:,1)\*b(1,2)

k=2: c(:,2) = c(:,2) + a(:,2)\*b(2,2)

k=3: c(:,2) = c(:,2) + a(:,3)\*b(3,2)

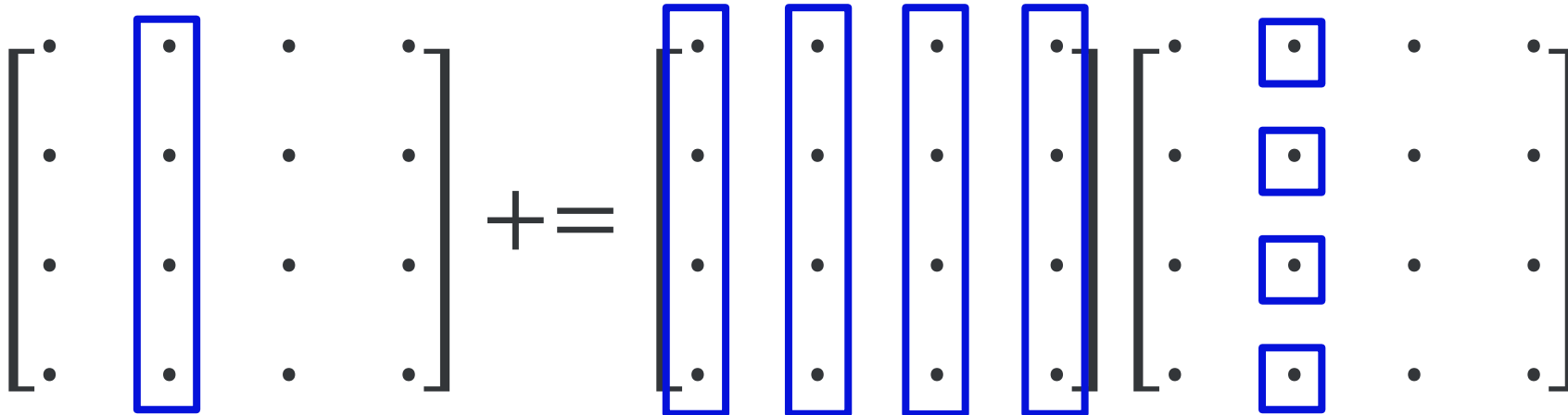
k=4: c(:,2) = c(:,2) + a(:,4)\*b(4,2)

A column by column

Single column of B

Single column of C, accessed N times

Inner two loops for j = 2:





# How can BLAS be even faster?

- What is BLAS?
  - Basic Linear Algebra Subprograms
  - Basic building block for several other libraries, including Lapack (solving linear systems and computing eigenvalues)
  - BLAS 1 (1979) defined vector operations to exploit vector computers
  - BLAS 2 (1986) added matrix-vector operations
  - BLAS 3 (1988) added matrix-matrix operations to better exploit a memory hierarchy
- The DGEMM code in BLAS is significantly more complex than ours: Matrix is split in small blocks that fit in cache and the matrix-matrix product is computed out of the matrix-matrix products of the smaller blocks
  - And some libraries may even involve assembler programming
- Blocking for cache reuse is a strategy used by many libraries

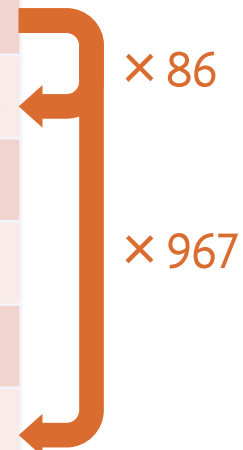
# And a little surprise...

Matrix 2500x2500 Method	GNU Fortran Gflops	Intel Fortran Gflops
ijk	1.82	1.60
jik	1.28	3.40
ikj	0.49	1.60
jki	3.17	3.40
kij	0.77	10.74
kji	2.35	10.68
F95 MATMULT	3.29	10.97
BLAS dgemm (single-threaded)	24.60	24.75
BLAS dgemm (20 threads)	396.42	417.64

- The Intel compiler is clever enough to reorder the inner two loops for optimal memory access in many cases.

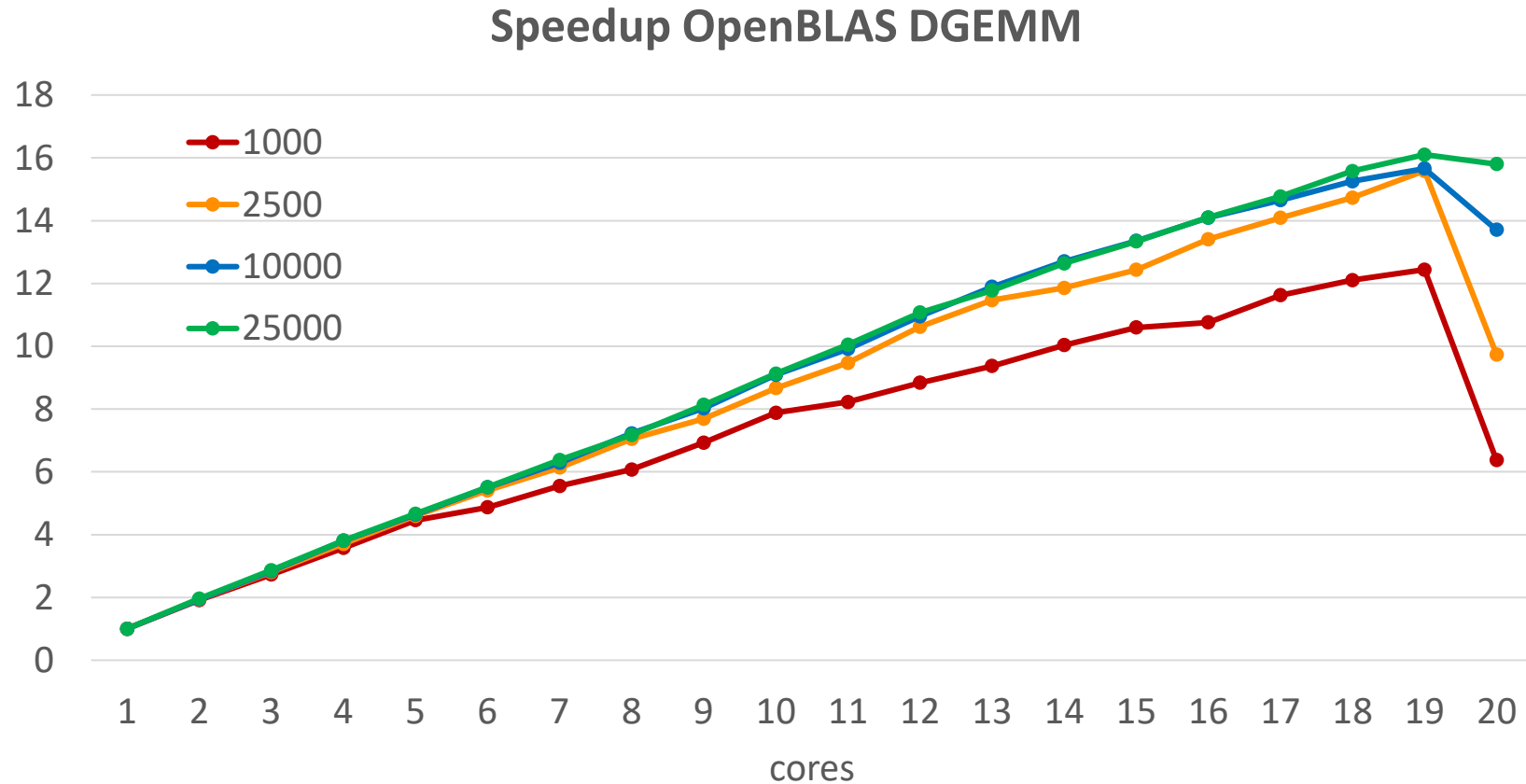
# On a laptop

Matrix 2500x2500, variant	Time (s)	Gflops
ijk	297.40	0.105
jik	295.53	0.106
ikj	1002.28	0.031
jki	11.67	2.678
kij	1002.48	0.031
kji	15.85	1.971
F95 MATMULT	17.06	1.832
OpenBLAS dgemm (1 thread)	3.42	29.979
OpenBLAS dgemm (2 threads)	1.80	57.349



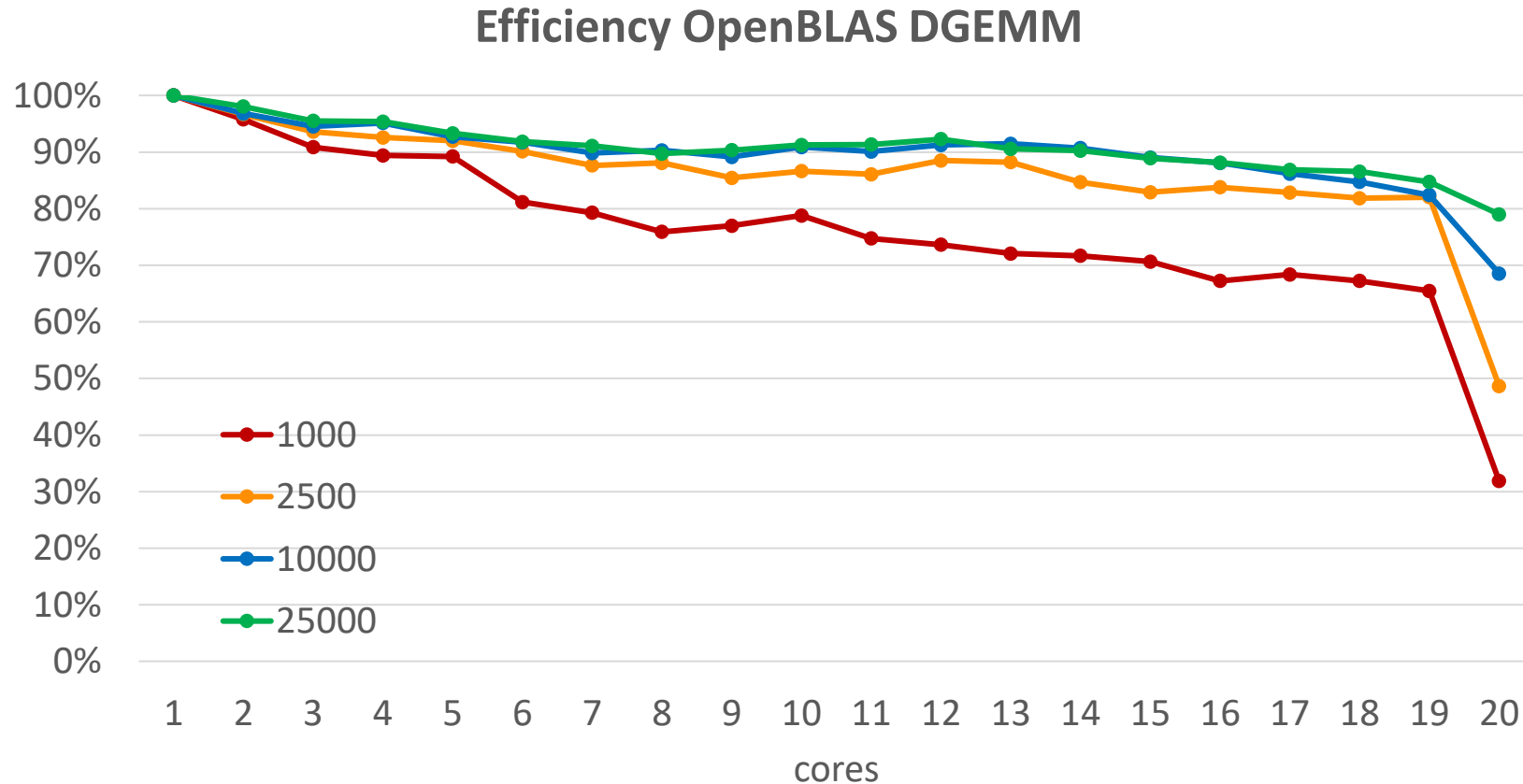
- This laptop (an older Intel Broadwell one) has less cache than a cluster CPU (even when counted per core) and fewer channels to memory. This explains the greater sensitivity to the correct memory access order.

# Scaling: Speedup



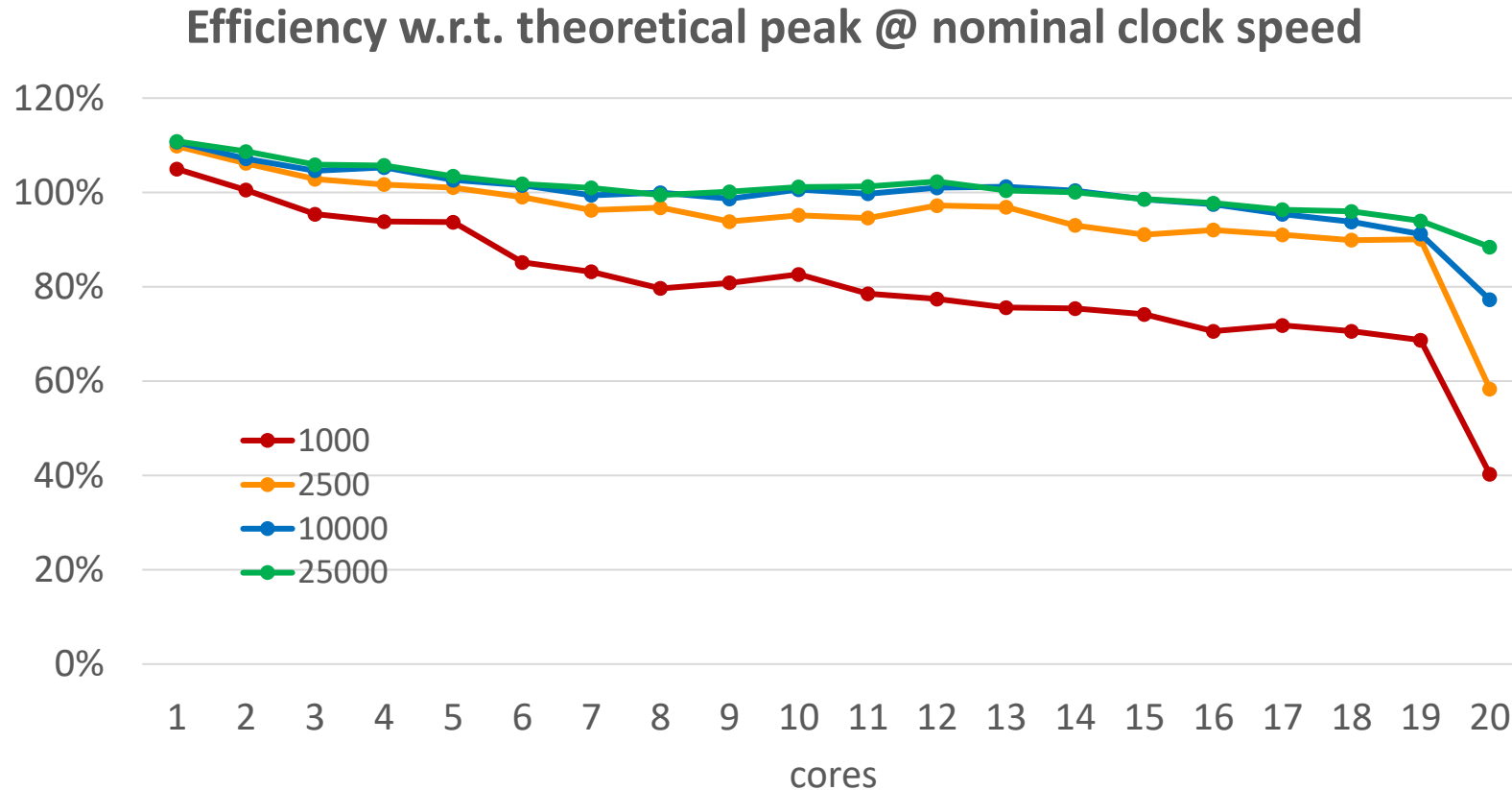
- Anomaly for 20 cores, probably due to interference with background work (operating system)
- Given number of cores: Speedup increases as problem size increases.

# Scaling: Efficiency



- Efficiency = speedup / # of cores
- Given number of cores: larger problem = higher efficiency
- Given problem size: more cores = lower efficiency

# And another surprise...



- The job runs faster than we would expect based on the nominal clock speed of the node.
- This is due to something called “turbo boost” by Intel: If the CPU runs cool enough, it is clocked higher (was enabled on that node, but disabled on leibniz and vaughan)

# Demo conclusions

- Not all codes are created equal, even if they implement the same operation/algorithm.
- Don't try to reinvent the wheel. It is very likely that there are already good libraries that you can use, or even a complete code for your problem, that is much better written than anything you can come up with in a short time.
- Pay attention to the order of memory accesses and exploit the memory hierarchy (even though some compilers can sometimes correct your mistakes)
- Scaling of code:
  - Fixed problem size: More cores = lower efficiency in most cases
  - Fixed number of cores: Bigger problem = higher efficiency in most cases
- On a modern computer system, benchmarking has become very difficult as there are so many unexpected elements that can influence performance in sometimes unpredictable or unexpected ways.



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 8: Accelerators

VLAAMS  
SUPERCOMPUTER  
CENTRUM

*Innovative Computing  
for A Smarter Flanders*

[vscentrum.be](https://vscentrum.be)



# Accelerators

- Accelerator is (usually) a coprocessor
  - Accelerates selected computations that could be done on a CPU, but cannot replace the CPU
  - Used to free the CPU from other work that could be done better in a specialised processor and popular in very large computer systems such as mainframes
- Accelerators appeared in the PC world also in the '90s
  - High-end sound cards had a processor specialised for signal processing algorithms (DSP)
  - Graphics card were originally very specialised and not really programmable hardware but in the early 00's became programmable also (NVIDIA GeForce 3, ATI Radeon 9300).
  - It didn't take long before scientists started experimenting with using that programmability to accelerate certain scientific computations. Manufacturers started adding features specifically for broader use and CUDA 1.0 was born in 2007.
  - As supercomputer developers are always looking for more speed at lower power, GPU computing, renamed GPGPU computing, became popular.
- As a GPU (and other accelerators) are coprocessors, it requires complicated programming with a **host program** and routines that are **offloaded** to the accelerator

# Types of accelerators

- Vector computing: Basically all modern “GPU”s, and then some others
  - NVIDIA Data Center series
    - Initially a more reliable version of the NVIDIA GeForce/Quadro GPU,
    - but evolving into a different product line, especially clear with the GPUs launched in 2022: Hopper for compute and Ada Lovelace for graphics
  - AMD (Radeon) Instinct GPUs: Separate architecture: CDNA, not RDNA
  - Intel is moving into that market also with the Intel Data Center GPU MAX (Xe<sup>HPC</sup> architecture)
  - NEC SX Aurora TSUBASA more and more used as an accelerator also

# Types of accelerators (2)

- Matrix operations: rank-k update or matrix-matrix multiplication
  - Motivated by operations in neural networks, but when done well useful for linear algebra also
  - NVIDIA: Tensor cores in the V100 and later chips
  - AMD: Matrix cores in the MI100 and later CDNA chips
  - Intel: Matrix engines in the Data Center GPU MAX (Ponte Vecchio)
  - Google TPU (Tensor Processing Unit)
- FPGA: Programmable logic, “build your own custom processor”
  - E.g., could build a specialised 2-bit processor to work with genetic data

# Types of accelerators (3)

- The name GPU has become misleading as the latest generation compute “GPUs” don’t support full hardware graphics acceleration anymore
  - E.g., no ray tracing units in either NVIDIA Hopper or AMD MI100/MI200 (Intel still has them in the GPU code named Ponte Vecchio)
  - Other graphics-only units more and more removed also
    - raster engine: Seems gone in NVIDIA Hopper H100 and AMD MI100/200
    - texture units
      - Present on NVIDIA Hopper but seem missing on AMD MI200
    - Video decode preserved for AI.
      - Encoding seems missing on H100
- So there is a growing distinction between compute GPUs for traditional HPC and AI, and rendering GPUs for visualisation and image generation

# Offloading to accelerators

- Most common model:
  - As CPU and accelerator don't have access to each others work memory you need to copy data between the memory spaces (and multiple times if both CPU and accelerator need the data interchangeably)
  - Multiple accelerators in a system may or may not share a memory space (NVIDIA NVLINK is a technology to share memory in a NVIDIA GPU system)
  - Pass control from the main program to the accelerator to execute code that can be accelerated
- Remember Amdahl and how communication overhead further restricts speed-up? For accelerators one has:
  - Not all code can be accelerated, so this will limit the speedup that can be obtained from an accelerator
  - Transfers between memories (even if they can to some extent overlap with computations) further limits the gains that can be obtained

# Offloading to accelerators

- If the amount of data that has to be copied is too large compared to the amount of computations that will be done with that data, we can even have a slowdown...
- Funny detail: Integrated GPUs in Intel and AMD PCs
  - CPU and GPU actually don't share memory spaces even if both use the same system RAM memory but have their reserved memory spaces (one exception: see next slide)
  - So data still needs to be copied between both memory spaces

# Offloading to accelerators (2)

➤ But this is about to end:

- Hardware extensions for a software-managed unified address space since NVIDIA P100
- USA pre-exascale systems based on IBM POWER CPU and NVIDIA GPU have NVLINK also between the CPU and GPU
  - So CPU and GPU share a memory space
  - Making it effectively more of a NUMA machine, one still needs to be careful where to store which data
- Some recent Apple A1x processors and the M-series are said to have a fully unified memory space between the CPU and the various accelerators
  - It's not only the very superscalar core that makes the M-series so fast. Its astonishing performance in photo- and videoprocessing apps comes from the accelerator architecture
- AMD: Special version of the Epyc CPU with the MI250X GPUs have unified and partially cache coherent memory (Frontier USA exascale system, EuroHPC LUMI system)
- Intel Xe “Ponte Vecchio” for 2022 Aurora supercomputer has unified and coherent memory
- NVIDIA is also working on its own CPU (Grace) to work with its GPUs (Hopper) (EuroHPC JUPITER exascale system)

# CPUs with accelerator features

- Successful accelerators were often integrated in CPUs, and this process is still going on
  - Vector accelerators:
    - Early vector instructions in Intel CPUs designed to compete with DSP
    - AVX-512 has its roots in a failed GPU
    - Fujitsu A64fx (used in one of the fastest computers in the world) uses vector instructions and a GPU-like memory architecture to reach near-GPU speeds in many applications
  - Matrix accelerators
    - IBM POWER10 has matrix instructions for deep learning in AI and for linear algebra
    - Intel has some too in some CPUs but only for low-precision inference and added a new instruction set called AMX in Sapphire Rapids (but still only for AI, int4/int8/bfloat16)
    - ARM V9-A adds Scalable Matrix Extensions for AI (int8/bfloat16)
- Interesting if even passing control to a coprocessor causes too much overhead
  - And can be programmed using the regular programming models



# Accelerators

## Where can you find them?

- UAntwerp: Small system (two nodes) for the first tests with NVIDIA P100, one node with 4 A100 GPUs and NVLINK and 2 servers with two AMD MI100 boards
- Production system for the VSC in Leuven (integrated in Genius) with NVIDIA GPU, mostly P100 with NVLINK, and 4 quad A100 nodes in the wICE cluster
- UGent has 10 nodes with quad NVIDIA V100 and 9 nodes with quad A100
- VUB has 4 nodes with dual NVIDIA P100 GPUs and 10 with dual A100 GPUs
- VSC Tier-1 system with 40 quad A100 nodes
- EuroHPC pre-exascale system LUMI meant for very large tasks
  - 2978 nodes with 4 AMD MI250X GPUs and 1 AMD Trento CPU
  - Almost 50 Tflops vector or 100 Tflops matrix performance in FP64 per GPU
  - Access via EuroHPC proposals and a Belgian programme as Belgium also invested in the machine
- More large GPU systems available available via EuroHPC (also NVIDIA)

# Accelerator programming

## ➤ It's a mess at the moment

### ➤ 3 main competing ecosystems

- NVIDIA ecosystem:
  - Largely proprietary to protect their market
  - Some support for open standards, but limited
- AMD: Latecomer so open sourced their ROCm software stack
- Intel: oneAPI initiative
  - Partly based on open standards
  - Partly open-sourced (in some cases only the API)

# Accelerator programming:

## Low-level

- Separate code for host and accelerator device
- NVIDIA CUDA probably the best-known environment in the HPC world
  - But proprietary NVIDIA so code is not very portable to other environments
  - CUDA C/C++/Fortran are subsets of these languages to develop offloaded code
  - And a large library of routines for popular tasks in scientific computing
- HIP – **H**eterogeneous-Computing **I**nterface for **P**ortability
  - CUDA-clone from AMD
  - Roughly at the level of CUDA 7 and 8
  - One-to-one match between HIP functions and CUDA functions
    - So works without loss of performance on NVIDIA
  - Allows to port CUDA code to AMD GPUs

# Accelerator programming:

## Low-level

### ➤ OpenCL

- Vendor-neutral standard controlled by the Khronos Group
- Works on GPUs from most vendors, some CPUs, some DSPs and even some FPGA systems, so your code is portable (though may require some tuning for other hardware)
  - This makes it very attractive for commercial software development
- Not as advanced as CUDA though
- Kernels in subset of C or C++
- Also in AMD ROCm stack and Intel oneAPI stack, and supported by NVIDIA
- Khronos Group itself is pushing users to higher-level models; newer parts of the standard not often implemented

# Accelerator programming:

## Compiler directives

### ➤ OpenACC

- First successful compiler directive model for GPU computing
- Single code for host and accelerator, the compiler does the hard work of organising the offloading
- Open standard, though it looks that in practice it is very much controlled by NVIDIA
- Compiler support
  - Commercial: Currently only NVIDIA. HPE Cray only in their Fortran compiler
  - Growing support in GCC, but only for some GPUs
  - Will likely also come in the Clang/Flang/LLVM ecosystem with the help of the USA exascale projects, but currently (LLVM 17) this support is very unfinished
- Currently at version 3.3 (released November 2022)

# Accelerator programming:

## Compiler directives

- OpenMP 4.0/4.5/5.0/5.1/5.2
  - Similar way of working as OpenACC, but based on the typical OpenMP constructs
  - Younger and thus far less mature than OpenACC
  - More vendors actively involved in the standardisation process
  - OpenMP 5.0 released at SC'18 in November, 5.1 at SC'20, 5.2 at SC'21
    - Improved support for debuggers, performance monitoring tools, etc.
    - More descriptive, less focus on prescriptive
    - Much improved support for accelerators, partly triggered by pre-exascale systems
    - OpenMP offload support in recent GNU and Intel compilers (Intel for its own GPUs only) and some recent Clang/LLVM builds
  - AMD ROCm and Intel oneAPI stack, also support in the NVIDIA compilers

# GPU programming:

## C++ extensions

- SYCL
  - Standard controlled by the Khronos Group
  - Really more a C++ 17 header library, but requires a SYCL-aware compiler for good code quality
  - Clear influences from OpenCL
  - Multiple compilers (often based on Clang/LLVM)
- Intel DPC++ (Data-Parallel C++) in oneAPI
  - Intel implementation of SYCL, though originally more an extension of SYCL
  - Intel implementation on top of LLVM
  - Plan to push upstream to Clang/LLVM repository
  - Implementation for NVIDIA and AMD by CodePlay, recently acquired by Intel
- C++AMP (Accelerated Massive Parallelism)
  - Microsoft-developed but open specification
  - C++ library + one small extension to the language
  - Deprecated

# GPU programming:

## Frameworks

### ➤ Frameworks/abstraction libraries

- Create code that is portable to regular CPUs and GPUs of various vendors (and sometimes other types of accelerators)
- C++-based
- Develop code that exploits all levels of parallelism except distributed memory.
- Examples:
  - Kokkos by Sandia National Labs – NVIDIA and AMD, but Intel GPUs still work-in-progress. It is the most extensive of the three mentioned here.
  - Raja by Lawrence Livermore National Laboratory – CUDA and HIP ready, but SYCL and OpenMP offload backends (for Intel) work-in-progress
  - Alpaka by CASUS Center for Advanced Systems Understanding, Helmholtz Zentrum Dresden Rossendorf – CUDA and HIP, SYCL and OpenMP backend for Intel GPU?



# GPU programming:

## Libraries

- You may be able to find a library for your needs that does the computations on a GPU...
  - NVIDIA CUDA ecosystem has a lot of libraries for various application domains
  - Intel has adapted all its libraries for its (and others) upcoming GPUs and other accelerators
    - Subset of the API for both CPU and GPU, part of the oneAPI spec
  - HPE Cray have accelerated versions of some of their LibSci routines for NVIDIA and AMD GPUs, selecting automatically between CPU and GPU versions (LUMI supercomputer)
  - AMD open-sources many of its libraries in the ROCm stack
  - Vendor-neutral libraries
    - MAGMA (Matrix Algebra on GPU and Multicore Architectures) is an early example
    - heFFTe is another example
- The ideal library supports both accelerated and CPU computing

# Status of GPU computing

- Subject to benchmark<sup>et</sup>ing: Overhyped with incomplete benchmarks (only benchmark the part that you can accelerate), marketing by numbers (redefine common terms to get bigger numbers), ...
  - Accelerator performance per dollar isn't improving much anymore for a given type of computations
  - The NVIDIA vendor lock-in and its success in the market made accelerators even more expensive
  - Only make sense if the speed-up at application level is a factor of 3 per accelerator card compared to a standard medium-sized dual socket node
- Yet by many seen as the future of supercomputing
- Accelerator features carry over to traditional CPUs and may be a better choice in some cases as CPUs have more memory available and programming is easier
  - There are problems where such a CPU may be the most economical solution!
- Remember the death of traditional vector computing...

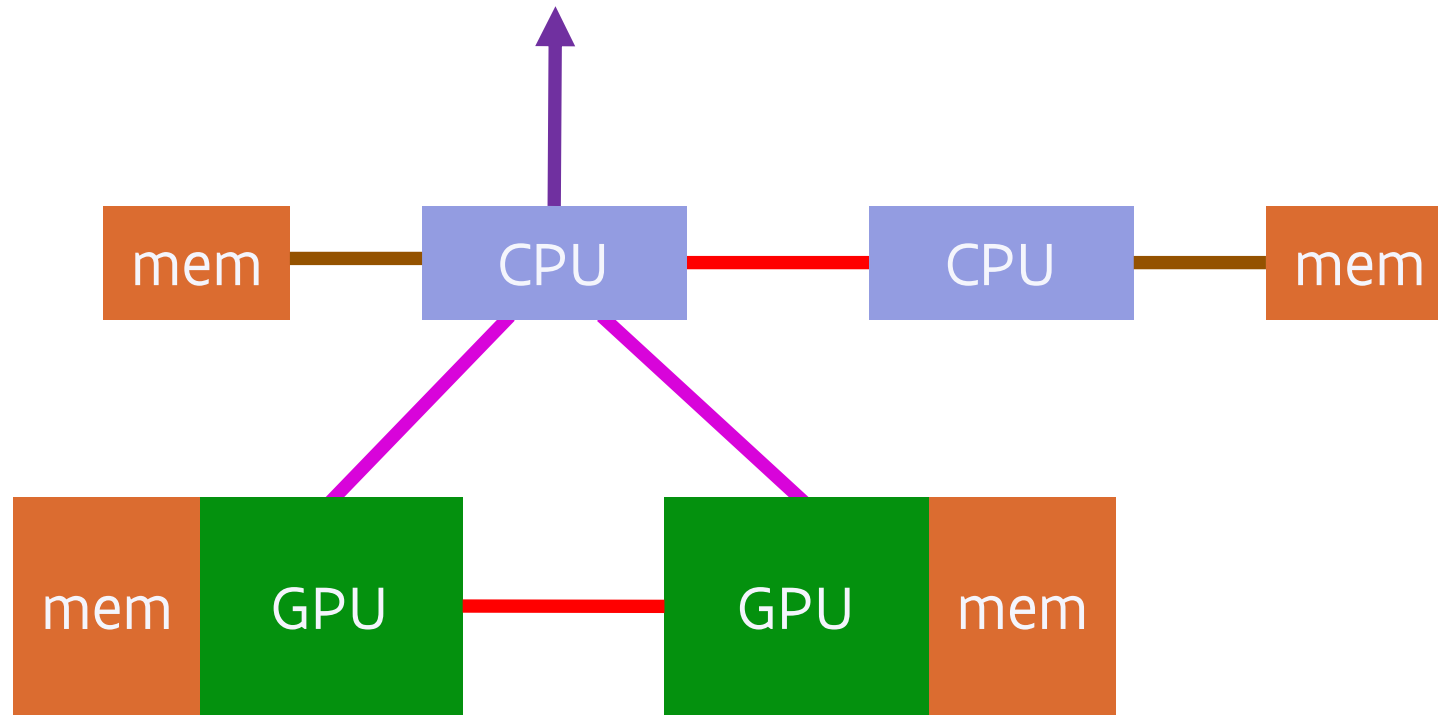
# Status of GPU computing: Problems and solutions

- **Limited memory:** Fast memory needed, so small amounts of memory (48GB in 2020, 80 GB in early 2021, 128 GB in 2022, 192 GB in early 2024)
  - **Solution:** Packaging improvements: More and larger memory stacks. 384 GB by 2025 or 2026?
- **Programming difficulty:** Separate memory spaces so lots of organising of copying back and forth between CPU and GPU memory
  - **Solution:** GPU sharing a memory space with CPU (cache-coherent and virtual memory space)
    - Hardware accelerated software managed solution as a stopgap in most modern GPUs
    - Already on some older NVIDIA GPUs with IBM POWER CPU
    - Appearing on other supercomputers
      - AMD MI250X in Frontier (USA exascale) and LUMI (EuroHPC pre-exascale)
      - Intel in Aurora (USA exascale system) with Sapphire Rapids CPU and Ponte Vecchio GPU
      - NVIDIA in 2024: Grace CPU and Hopper GPU
    - May make a NUMA-like memory model a reality and make it cheaper to transfer control between CPU and GPU as data copying may not always be needed
    - (Rumoured) extreme form: Apple M-series system-on-chip

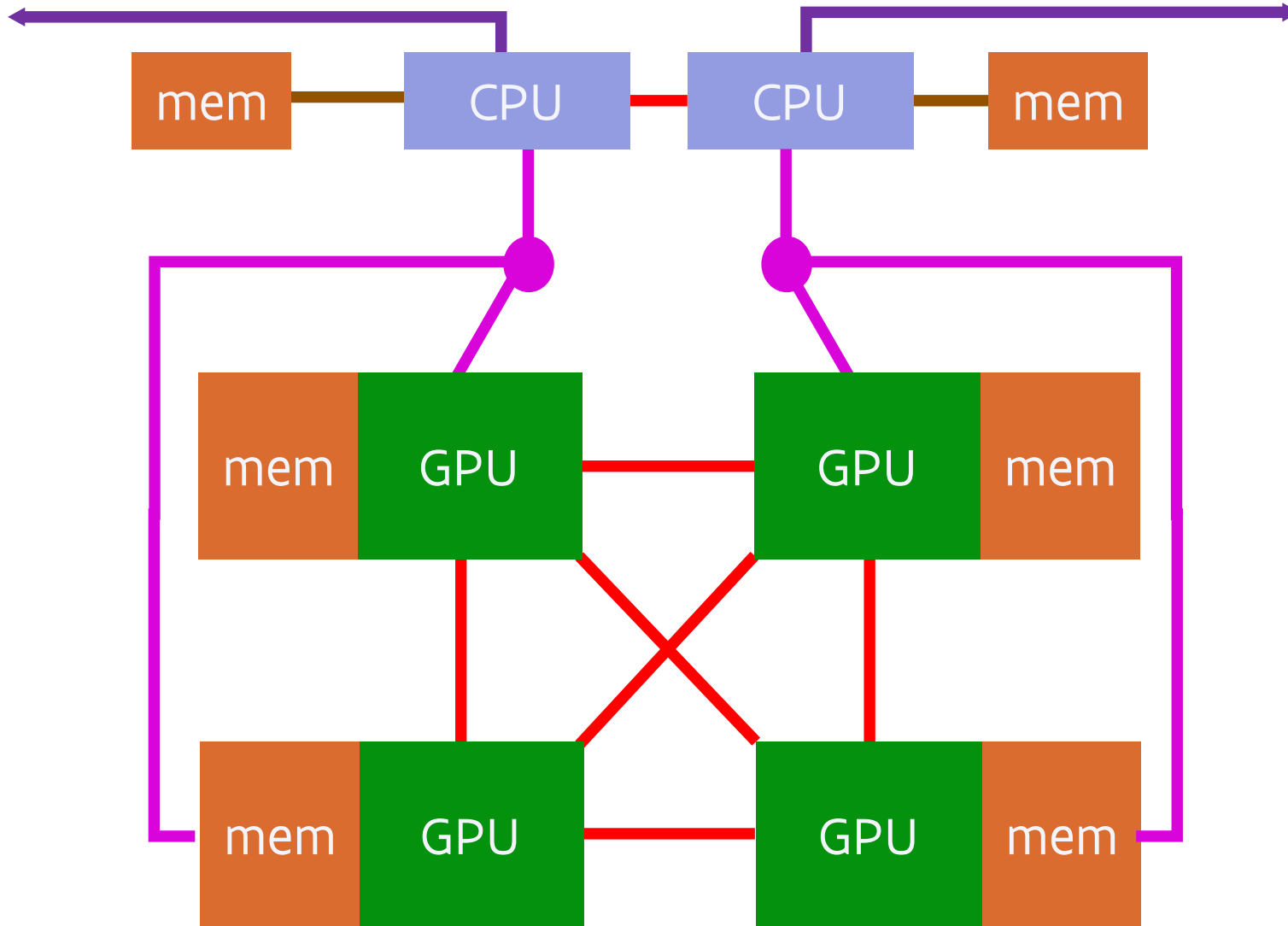
# Status of GPU computing

- **Rather slow for serial code**, so you often must use the host
  - This is mostly an issue because transferring between the fast scalar core in the CPU and the fast vector and matrix cores in the GPU is too expensive
  - **Solution:** Closer integration of CPU and GPU physically (distance) and logically (shared unified memory space) to make passing control between CPU and GPU as cheap as possible. See, e.g., the AMD MI300A.
- **Link to the CPU is the bottleneck**
  - **Solution:** Very close integration of CPU and GPU in a single package or on a single die so that CPU and GPU can share a *single physical memory space*. See, e.g., the NVIDIA Grace Hopper “superchip” or even more advanced, the AMD MI300A.

# 2016 GPU node

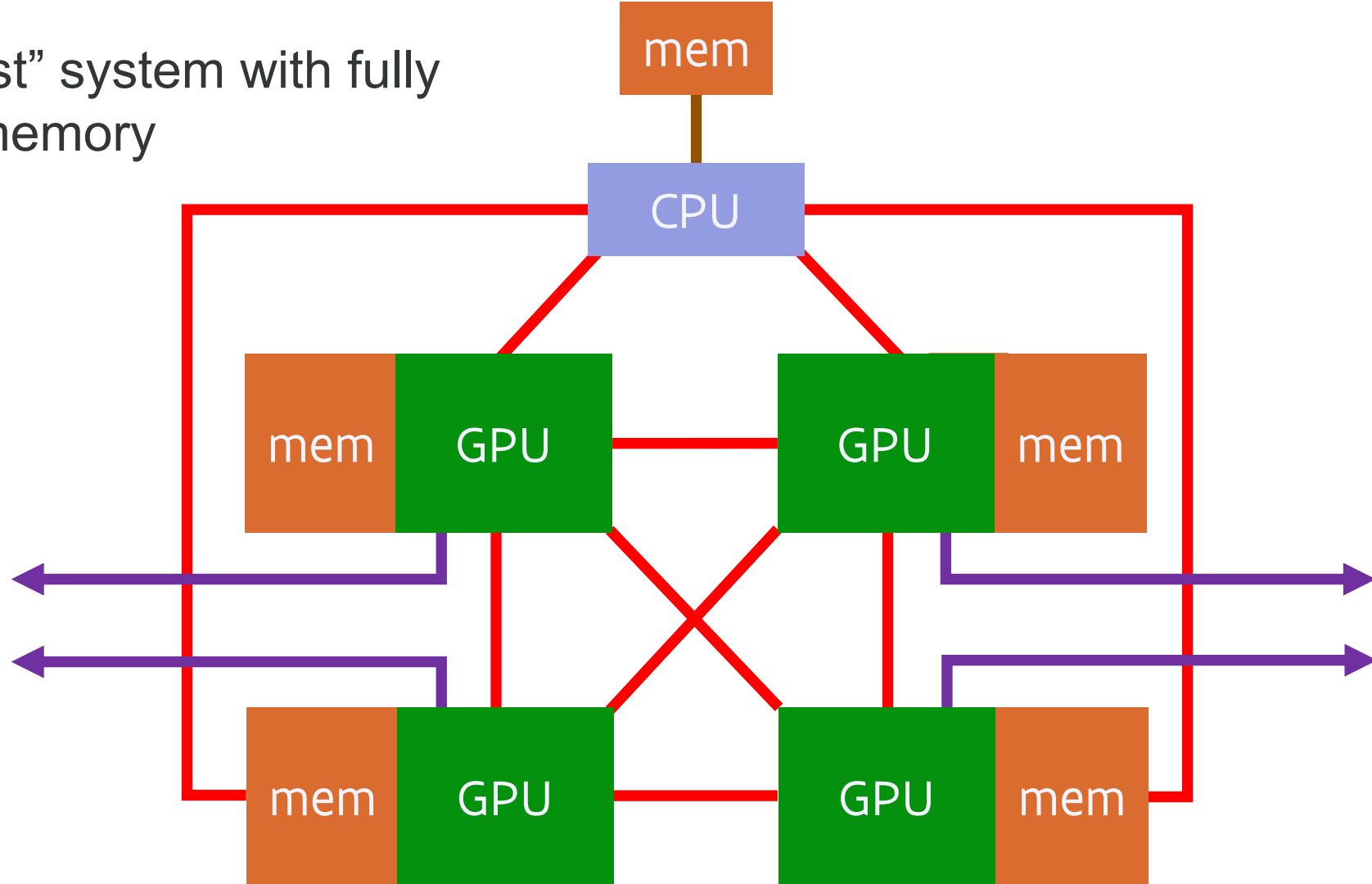


# A quad-GPU A100 supercomputer node

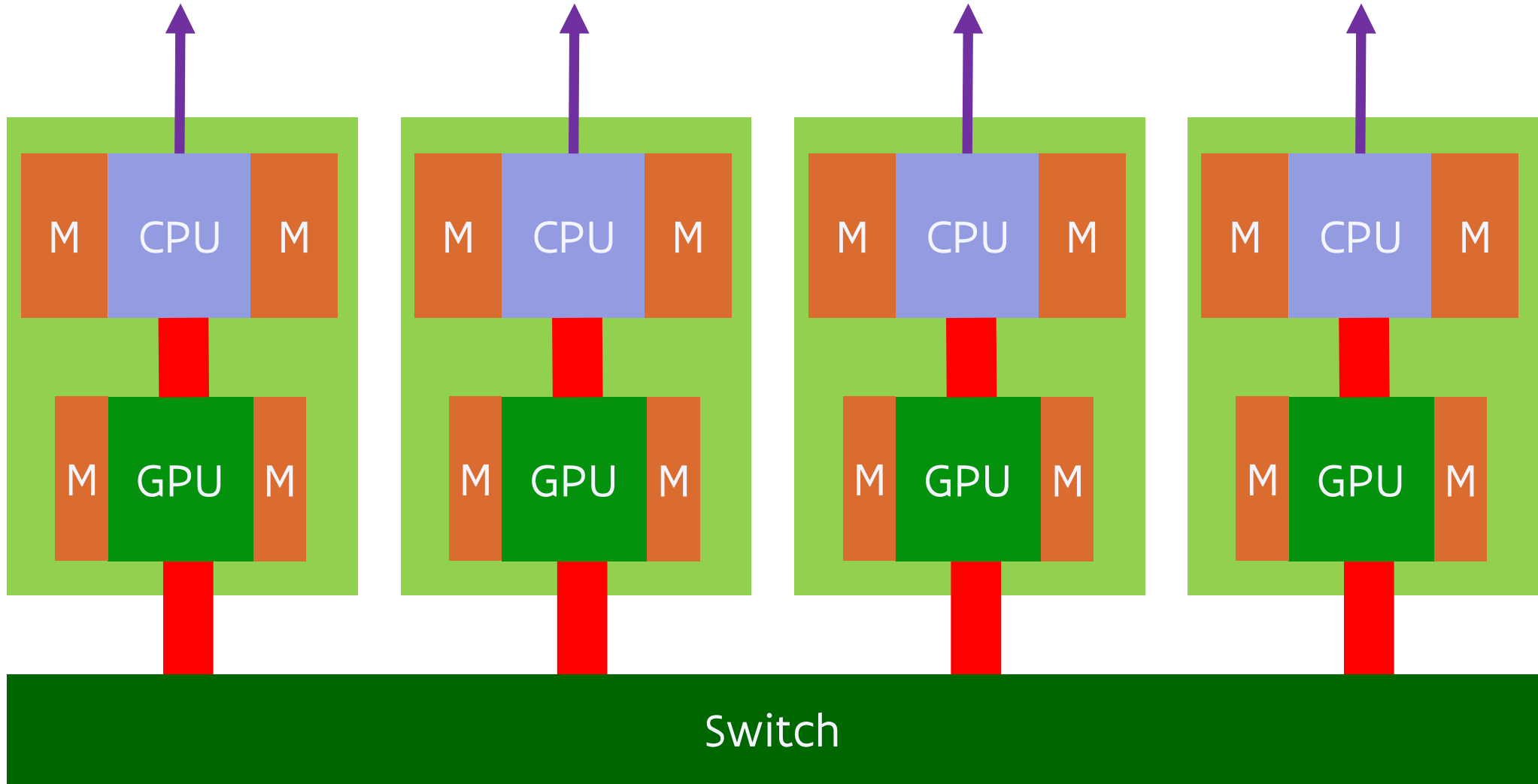


## Towards exascale: AMD MI250X in LUMI and Frontier

## “GPU first” system with fully unified memory

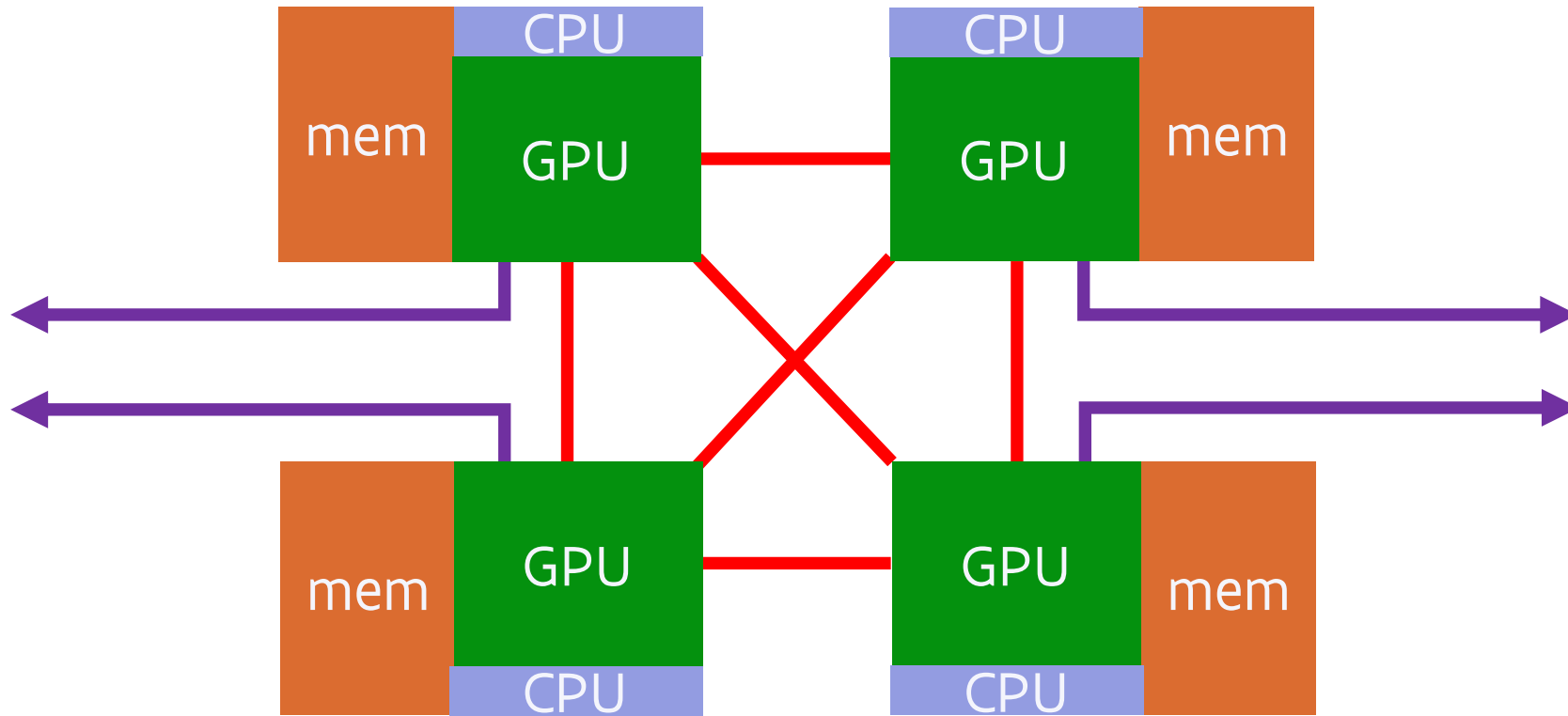


# NVIDIA Grace Hopper GH200





# AMD MI300A – El Capitan and Hunter



- USA: El Capitan  
Upcoming European system with a similar architecture: Hunter (HLRS)
- Intel was working on something similar in Falcon Shores, but it looks like in the end that one might still be a regular GPU (and has been postponed till 2025)



Vlaanderen  
is supercomputing

# Supercomputers for Starters

Part 9: Some conclusions

# Cost-conscious computing

- Supercomputing is very expensive. Some really big runs cost on the order of 10k EURO.
  - No academic user sees the full bill,
  - but someone is paying that bill.
  - So a supercomputer should be used efficiently.
- And this is not trivial
  - Though your non-supercomputer work will also benefit.
- And remember:
  - It is not a machine that will automatically magically run every program faster than a PC, though there is often a high degree of compatibility with Linux PC's
  - It is not an excuse for lousy programming. First optimize and then move to the supercomputer and not the other way around.

# Cost-conscious computing:

## Software users

- Select your packages with care and follow the evolutions in your field
  - Hardware evolves, some software packages evolve while other packages stay behind
  - There is a lot of work going on on software optimisation of many popular open source simulation packages
  - The most hyped package or technology is not always the best for your needs
  - But the package that your advisor used for their Ph.D. isn't always the best either.
- Learn how to use the packages efficiently, with a balance between execution efficiency and time-to-answer.
  - Most packages, especially the free ones, have no auto-tune facility; you need to do that work!
  - And for numerical simulation: Understand the limits of the models and numerics.

# Cost-conscious computing:

## Software developer

- Prototype languages are just for building prototypes.  
Not suitable for production runs (e.g., Matlab), though the situation is improving.
  - Julia is a nice alternative for those applications that even supports GPU computing
- Scripting languages are for gluing components together.  
Not suitable for whole applications that you'll run for months (e.g., Python, Perl).
  - Julia is a nice alternative for those applications also
- (Pre-)compiled applications (instead of running in a JIT virtual machine) are fashionable again.  
Look at what's happening in the mobile computing world!
- Automatic garbage collection is not a good friend with distributed memory applications.
- The wheel has been invented already. Use well-written frameworks and libraries to develop your application rather than “numerical recipes”.

# Cost-conscious computing:

## Why you should care

“Why should I care? I need to write my Ph.D. as fast as possible, produce papers and B.T.W., it’s (almost) free, isn’t it?”

- Resources are not infinite and most clusters run at a fairly high load
  - You’ll get more work done if your code is more efficient, thanks to a fair share policy, or limited allocations on larger clusters
  - If we want more money for more compute capacity, we’d better show our funding agencies we use the available capacity well
  - And our power budgets are limited also
- Don’t participate in climate marches if you don’t want to pay attention to efficiency
- An important task for the VSC is knowledge transfer to industry so that the industry can use HPC to become more competitive.
  - Compute time does come at a cost for industry...
  - Time-to-solution matters

# Cost-conscious computing:

## Why you should care (2)

“But by the time I’ve reworked my code or rolled out a better application in my research, faster computers will be around.”

- Those days when Intel engineers were optimising your program while you were having lunch or going to a party are long gone.  
Since about 2005 to be precise, when Dennard scaling started to break down.
- That’s when further increasing clock speeds became physically difficult and when further gains from improving instruction level parallelism also became small.
- Since then, further speed increases almost exclusively came from increased parallelism at the higher levels: wider vector instructions, more cores and more nodes.
- Even that is slowing down as we’re bumping into the physical limits of semiconductor technology and as the cost per transistor stopped decreasing.
- The evolution in computing is towards increasing performance per Watt as energy bills become prohibitively high (server farms, supercomputers) or portability matters (lightweight, so small battery). This implies more but slower cores so lower single-thread performance.

# Cost-conscious computing:

## Software quality matters

- As a consequence of the evolution of semiconductor technology:
  - Expect only moderate performance increases (if any) for a constant budget, unless switching to radically new architectures
  - Expect performance/Watt to increase less than before, unless radically new architectures bring a solution
- Hence further speed increases will have to come more and more from better software
  - Better algorithms and a better implementation
  - Exploit new architectures, which will require code changes
- The “computational” in computational science more than ever means having attention to the computations and willingness to understand what you’re doing at the computational level also.
  - You cannot drive a big truck or an F1 car with a regular driver’s license,
  - And you cannot use a supercomputer in the same way as a regular PC or a smartphone.



## Lesson 0

HPC

=

High-Performance Computing

≠

High-end Personal Computer

## Lesson 1

It is more and more the software  
that makes the supercomputer!

## Lesson 2

Streaming, parallelism and hierarchy

## Lesson 3

Efficient setup of a run depends on 3 elements:

1. The hardware of the supercomputer
2. The application
3. The (size of the) problem you're solving

## Lesson 4

It's crisis!

Transistors don't become cheaper anymore

## Lesson 1

It is more and more the software  
that makes the supercomputer!

# Questions?

# Further information

- [Course notes](#)
- [YouTube play list](#)
  - And a [shorter but similar introduction given at EPCC](#), with focus on computational chemistry
- Some tutorials on the web
  - [A Beginner's Guide to High-Performance Computing \(Oregon State University\)](#)
  - [Lawrence Livermore National Lab \(LLNL\) introduction, to parallel computing](#) takes a different path but is an interesting document (as are [their other courses](#))
- An [article in HPC Wire on the re-emergence of vector instructions](#)



# Further information

- Vastly different SSD speeds depending on capacity, showing that parallelism is important there too:
  - 2022 M2 MacBook Pro 13" (launched in June 2022)
    - [Review on ArsTechnica](#)
    - [Review on ExtremeTech](#)
  - 2016 iPhone 7 32 GB compared to the 128GB and 256GB SKUs
    - [ExtremeTech article](#), with a more technical explanation of what's going on.
    - [Article on CNET](#)

# Further information

- OpenMP:
  - Links on [the VSC documentation web site OpenMP page](#)
  - [Website of the OpenMP Architecture Review Board](#)
- [Intel Threading Building Blocks web site](#)
- C#
  - Microsoft Visual Studio manuals contain [a lot of information](#)
  - The [Mono framework](#) enables running C# programs on Linux and macOS
- Go: [The Go home page](#)
- [A tutorial about POSIX Threads Programming](#) by LLNL

# Further information

- MPI:
  - Links on [the VSC documentation web site MPI page](#)
  - [Open MPI implementation](#)
  - [MPICH implementation](#), the basis for Intel MPI and Cray MPI
- [The Julia programming language](#) and its [YouTube channel](#)
  - [Article about Julia adoption in HPC Wire](#)
- [Charm++ web site](#) (University of Illinois)

# Further information

- Coarray Fortran
  - [Web page at Rice](#) (outdated, development is very slow)
  - Elements incorporated in the [Fortran 2008 standard](#)
  - Support in recent versions of the Intel Fortran compiler
  - [Some support in recent GCC gfortran versions](#) (5.1 or newer best)
- Unified Parallel C:
  - [Berkeley UPC](#)
- [Chapel web site](#) and [Facebook page](#)
- [OpenSHMEM web site](#)
- GASPI
  - [Web page of the consortium working on GASPI in Germany](#)
  - [GPI-2 implementation by the Fraunhofer institute](#)

# Further information

- OpenCL and SYCL
  - The Khronos group that develops the standard, also maintains [OpenCL documentation](#) and a page with [SYCL resources](#)
- OpenACC
  - [OpenACC web page](#)
  - [PGI has a page with very useful links on its site](#)
- [Intel oneAPI](#) (includes DPC++) – USA Aurora exascale system
- [AMD ROCm](#) (includes HIP) – LUMI supercomputer and USA Frontier exascale system
- Frameworks:
  - [Kokkos](#) (Sandia)
  - [Raja](#) (LLNL) - [GitHub](#)
  - [Alpaka](#) (CASUS, Helmholtz)

# Interesting articles

- [NextPlatform: Compiling History to Understand the Future](#)