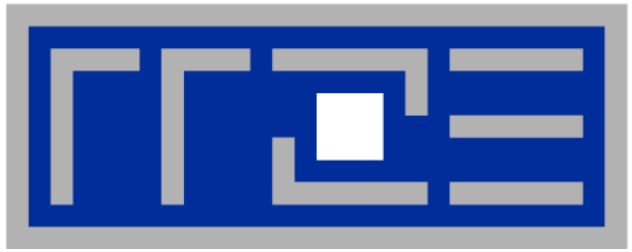
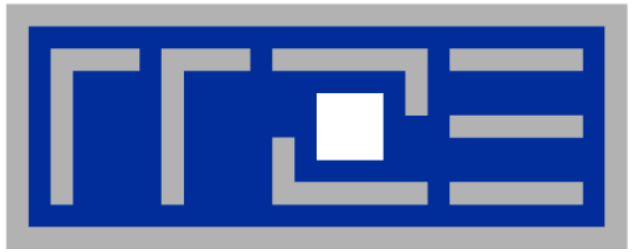


Introduction: Modern computer architecture

The stored program computer and its inherent bottlenecks

Multi- and manycore chips and nodes

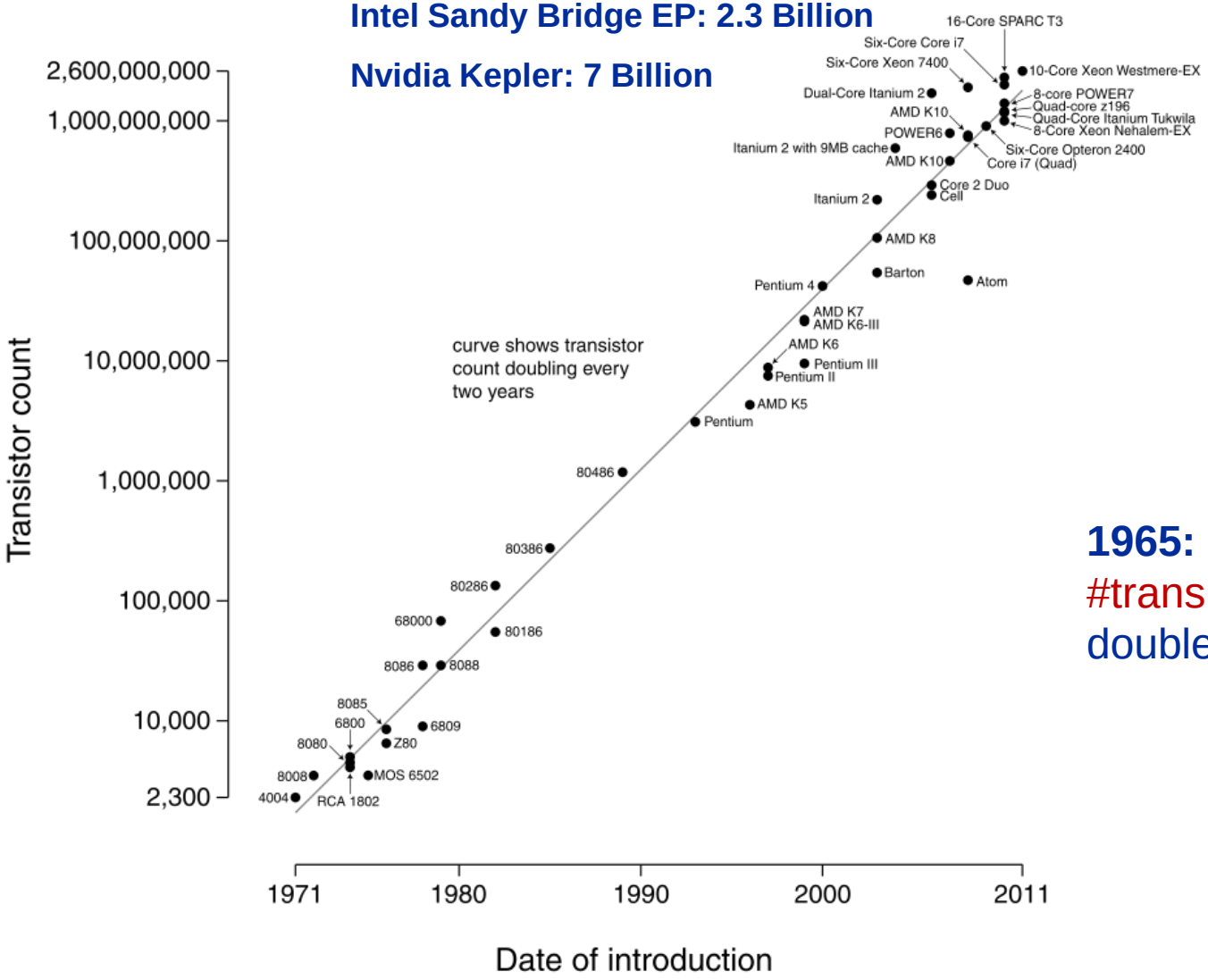


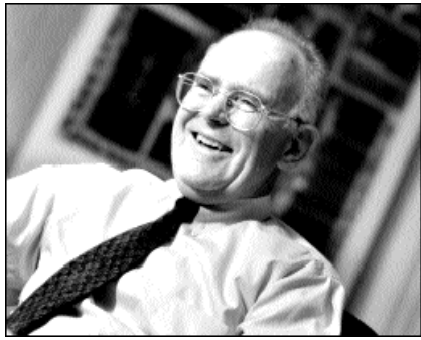
Motivation:
Multi-Cores – where and why

Introduction: Moore's law

Intel Sandy Bridge EP: 2.3 Billion

Nvidia Kepler: 7 Billion





1965: G. Moore claimed #transistors on "microchip" doubles every 12-24 months

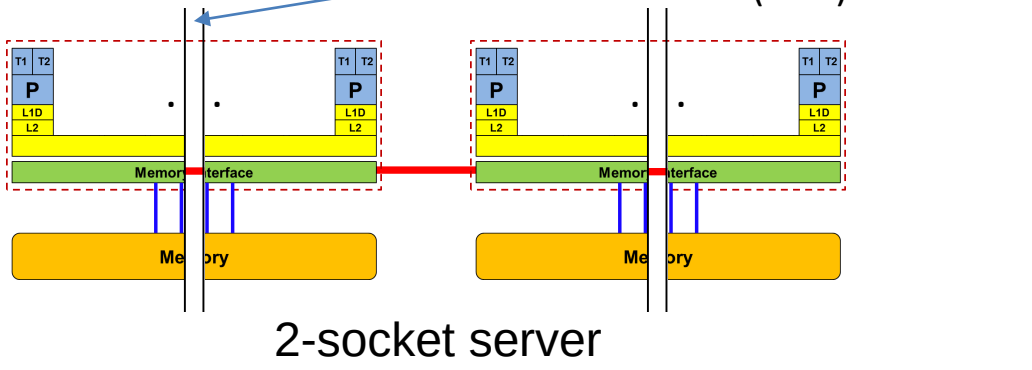
Multi-core today: Intel Xeon 2600v3 (2014)

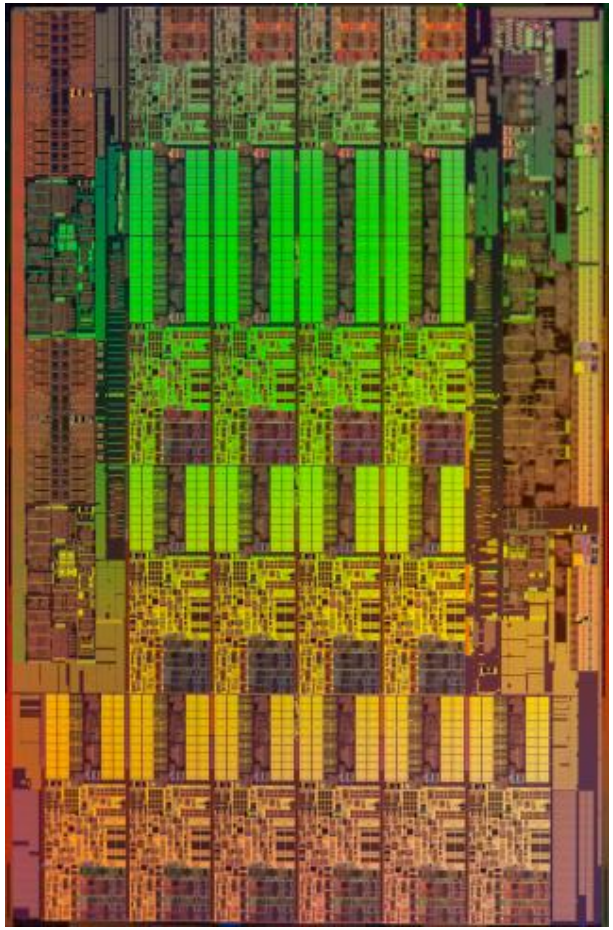
- **Xeon E5-2600v3 “Haswell EP”:**
  Up to 18 cores running at 2+ GHz (+ “Turbo Mode”: 3.5+ GHz)

- **Simultaneous Multithreading**
  → reports as 36-way chip

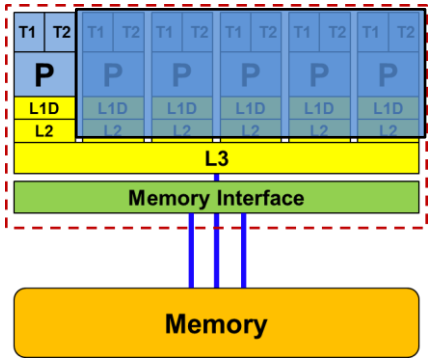
- **5.7 Billion Transistors / 22 nm**

- **Die size: 662 mm²**

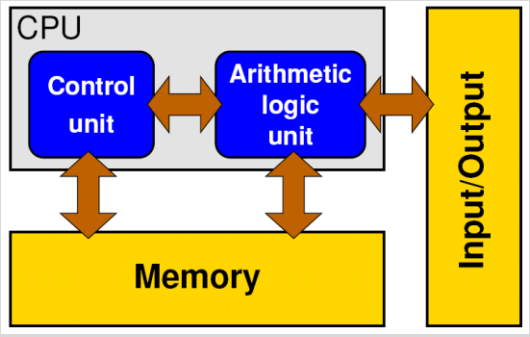




A deeper dive into core and chip architecture

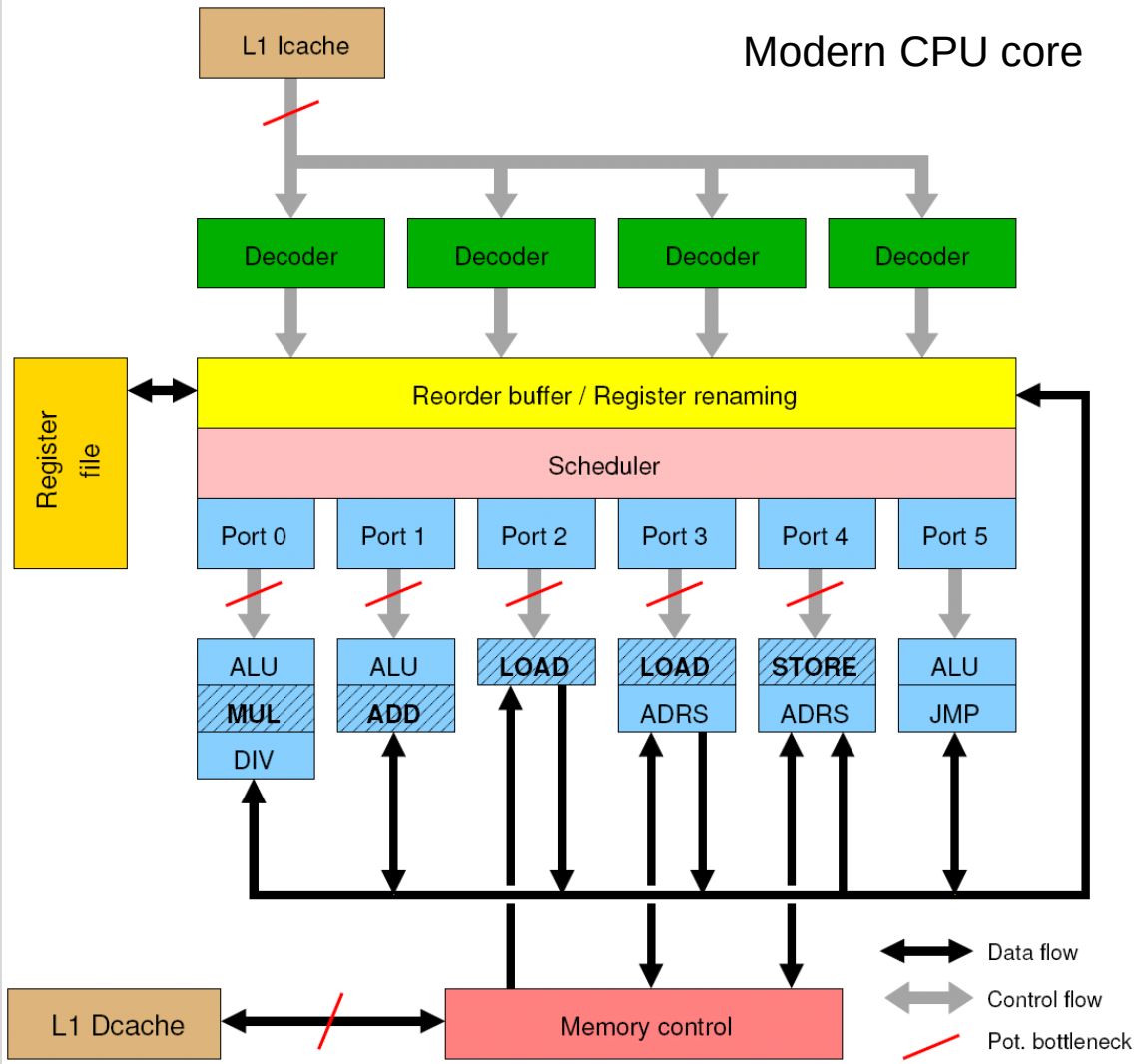


General-purpose cache based microprocessor core



Stored-program computer

- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks
- Flexible!



1. Instruction execution

This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

Instructions are the concept of "work" as seen by processor designers.

Not all instructions count as "work" as seen by application developers!

Example: Adding two arrays `A(:)` and `B(:)`

```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

Processor work:

```
LOAD r1 = A(i)
LOAD r2 = B(i)
ADD r1 = r1 + r2
STORE A(i) = r1
INCREMENT i
BRANCH → top if i<N
```

User work:
`N` Flops (ADDs)

2. Data transfer

Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

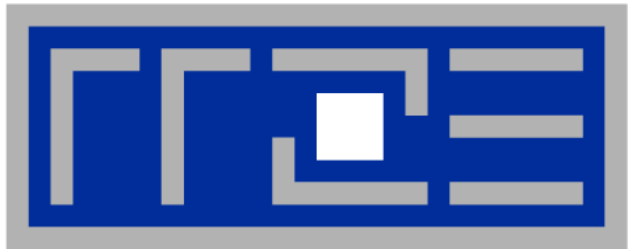
Example: Adding two arrays `A(:)` and `B(:)`

```
do i=1, N
  A(i) = A(i) + B(i)
enddo
```

Data transfers:
8 byte: `LOAD r1 = A(i)`
8 byte: `LOAD r2 = B(i)`
8 byte: `STORE A(i) = r2`
Sum: **24 byte**

Crucial question: What is the bottleneck?

- Data transfer?
- Code execution?



Microprocessors – Pipelining

Pipelining of arithmetic/functional units

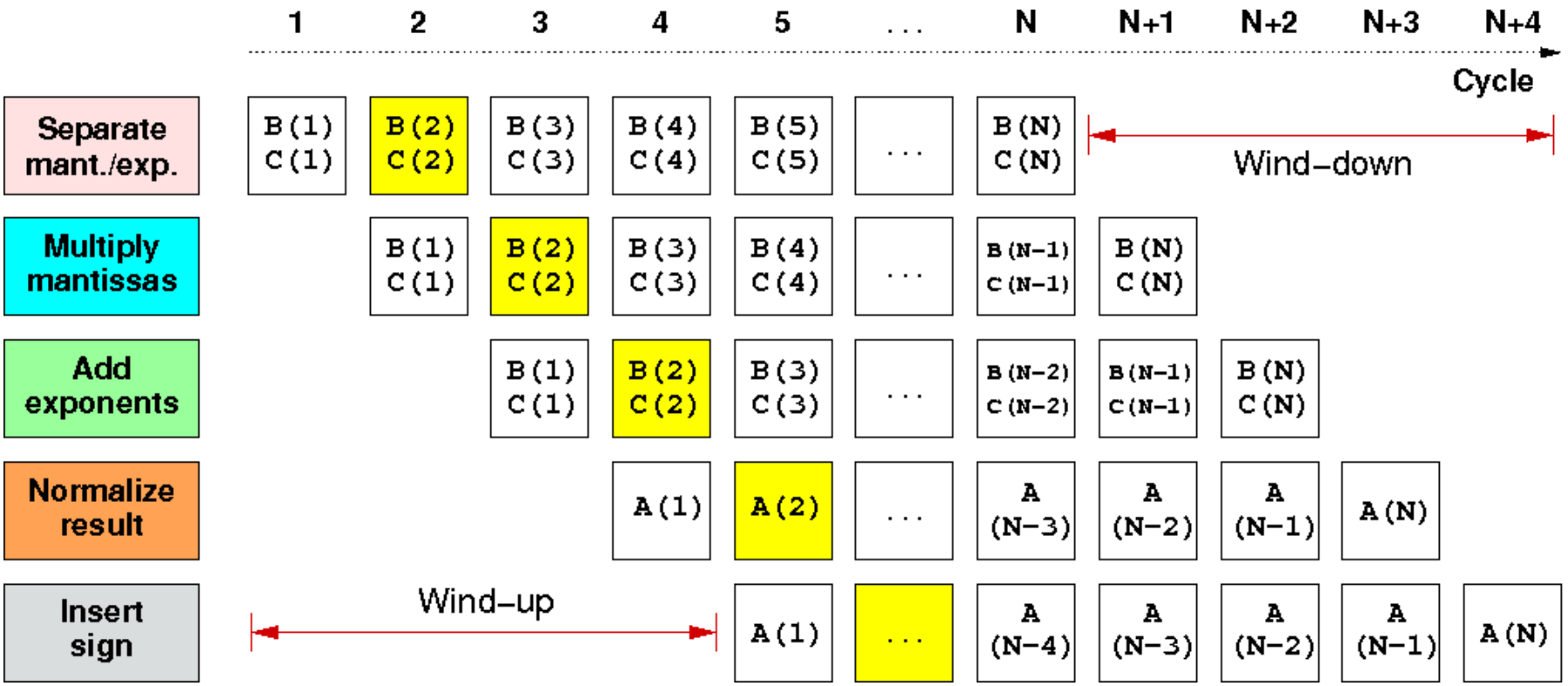
- **Idea:**
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g., a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

- **Allows for shorter cycle times (simpler logic circuits), e.g.:**
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultaneously
  - one result at each cycle after the pipeline is full

- **Drawback:**
  - Pipeline must be filled - startup times (#Instructions >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order

- **Pipelining is widely used in modern computer architectures**

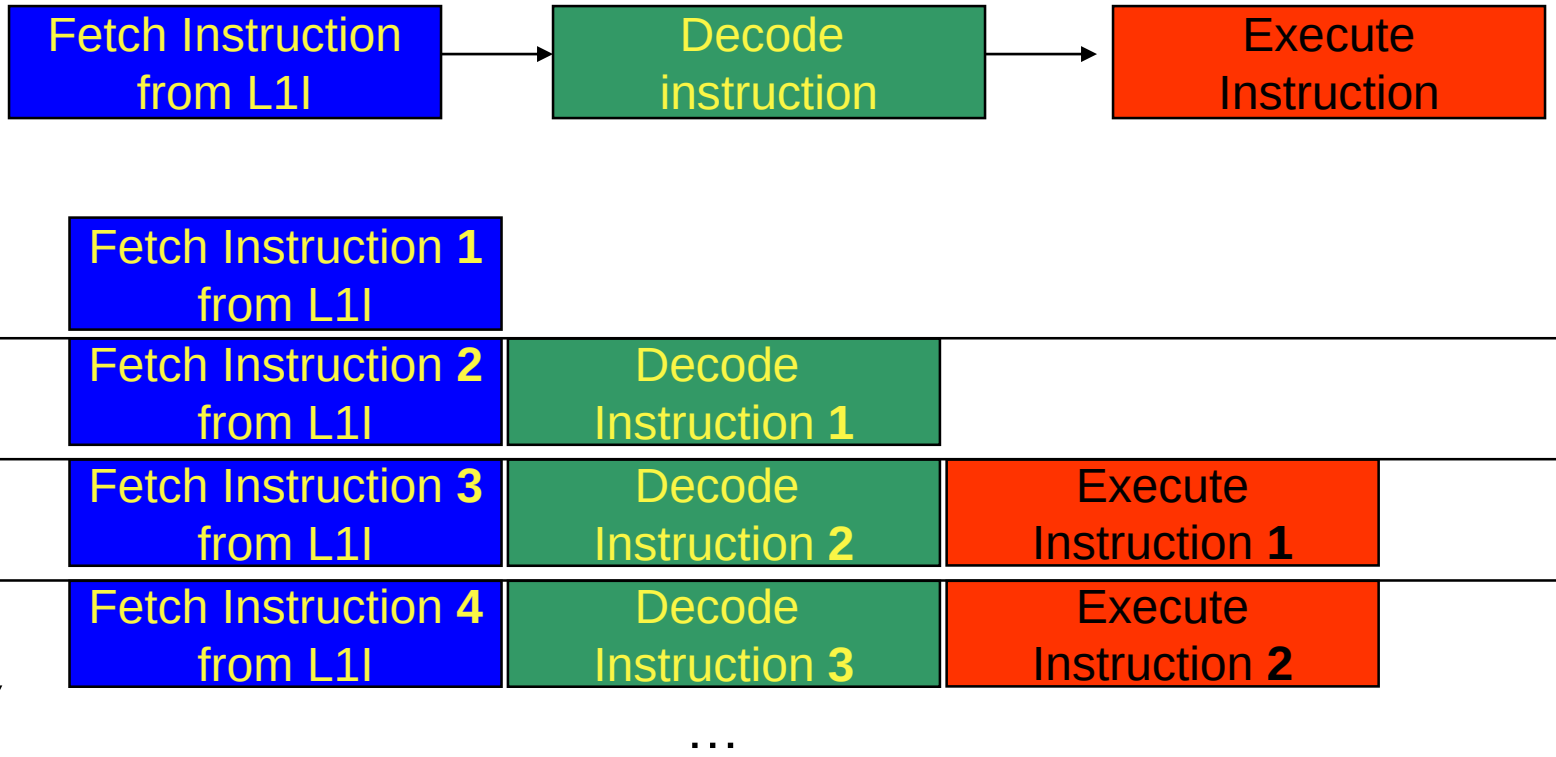
5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N



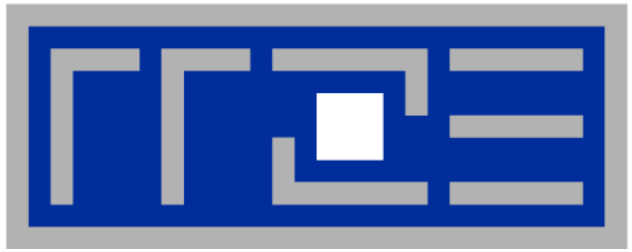
First result is available after 5 cycles (=latency of pipeline)!

Wind-up/-down phases: Empty pipeline stages

- **Besides arithmetic & functional unit, instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:**

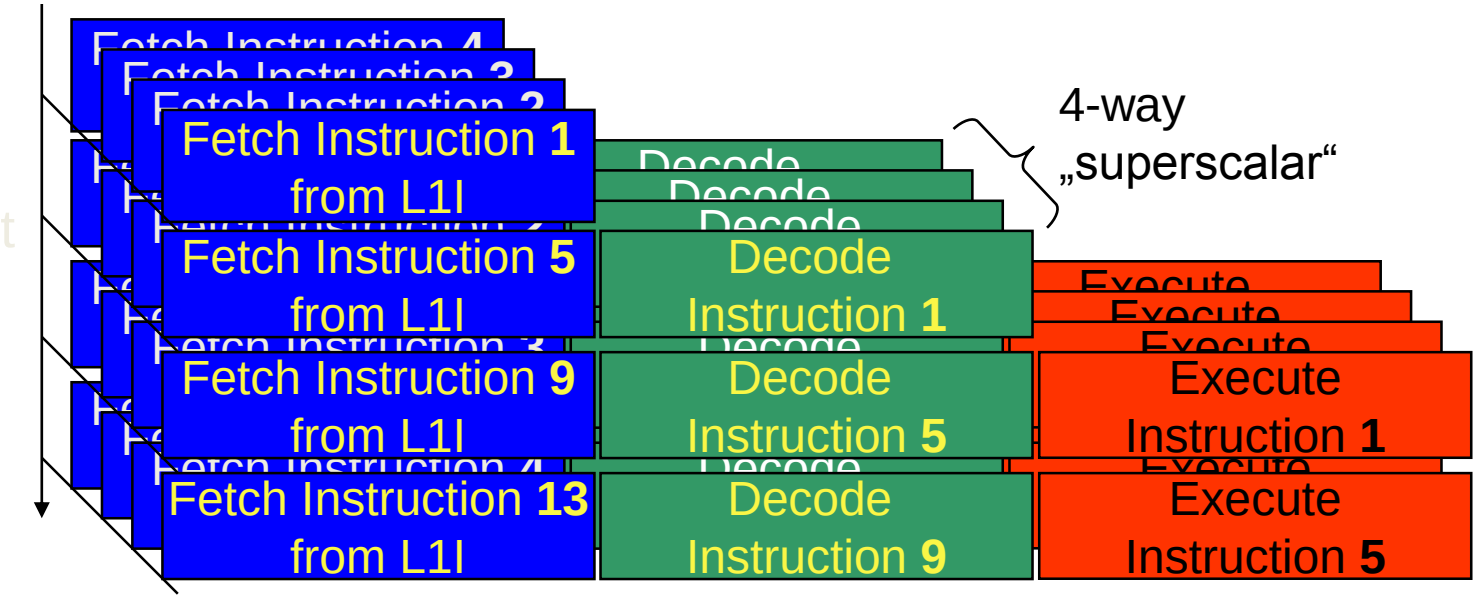


  - Branches can stall this pipeline! (Speculative Execution, Predication)
  - Each unit is pipelined itself (e.g., Execute = Multiply Pipeline)



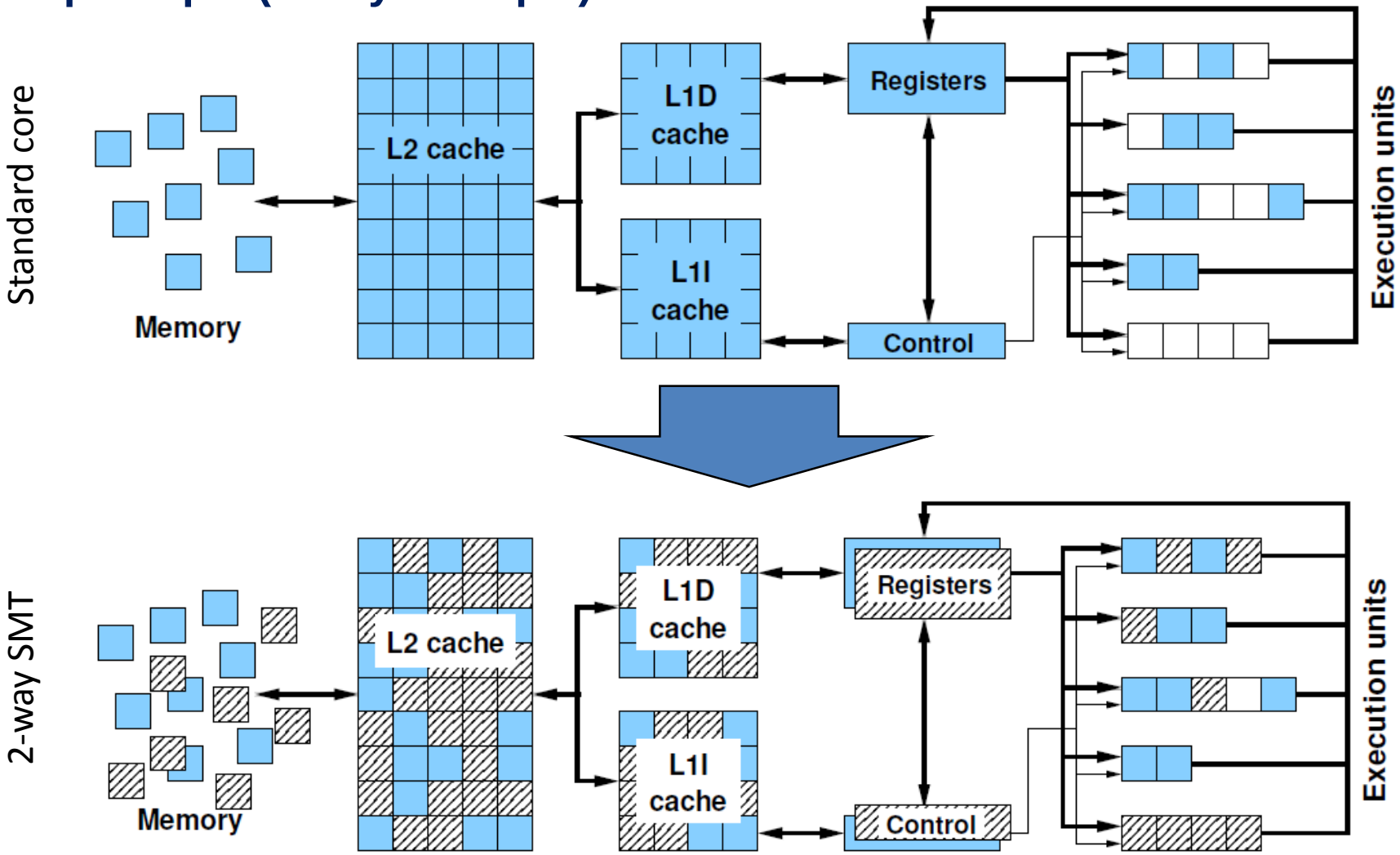
Microprocessors – Superscalarity and Simultaneous Multithreading

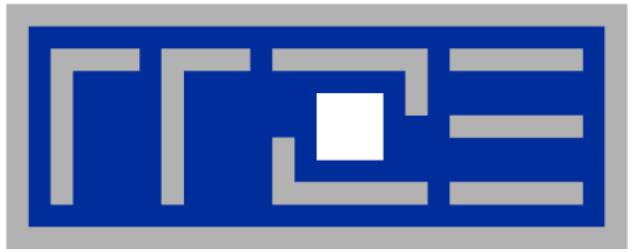
- Multiple units enable use of Instrucion Level Parallelism (ILP): Instruction stream is “parallelized” on the fly



- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 floating point instructions per cycles

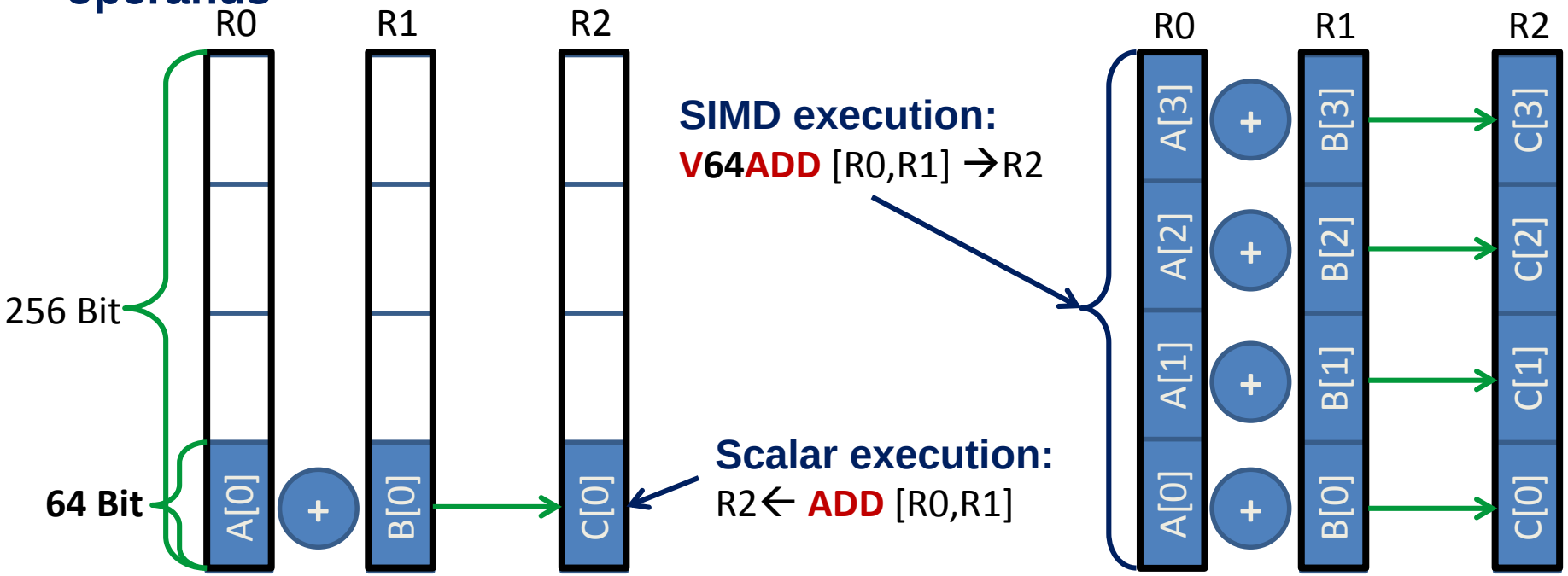
SMT principle (2-way example):

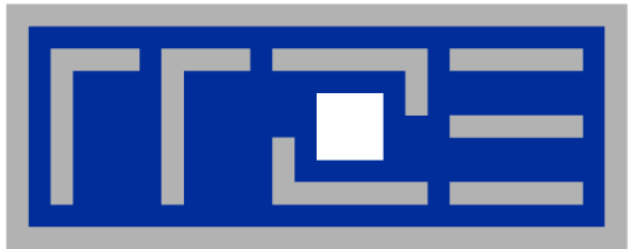




Microprocessors – Single Instruction Multiple Data (SIMD) a.k.a. vectorization

- **Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on "wide" registers**
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- **Adding two registers holding double precision floating point operands**





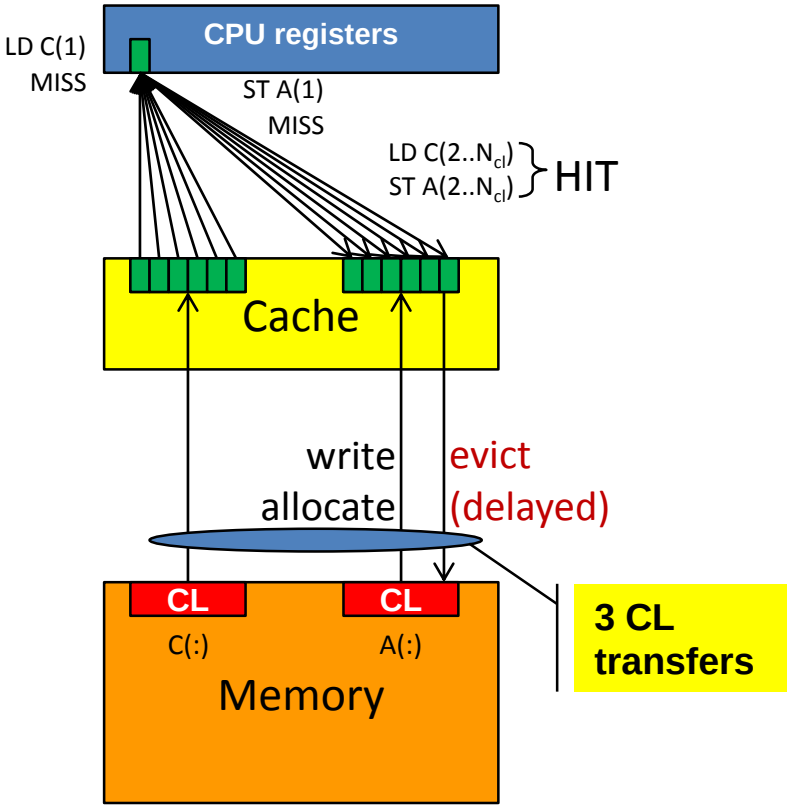
Microprocessors – Memory Hierarchy

Registers and caches: Data transfers in a memory hierarchy

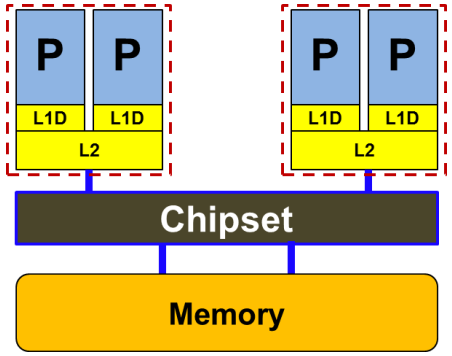
- **Caches help with getting instructions and data to the CPU "fast"**
- **How does data travel from memory to the CPU and back?**

- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level
  → CL transfer required

- Example: Array copy `A(:)=C(:)`



Yesterday (2006): Dual-socket Intel "Core2" node

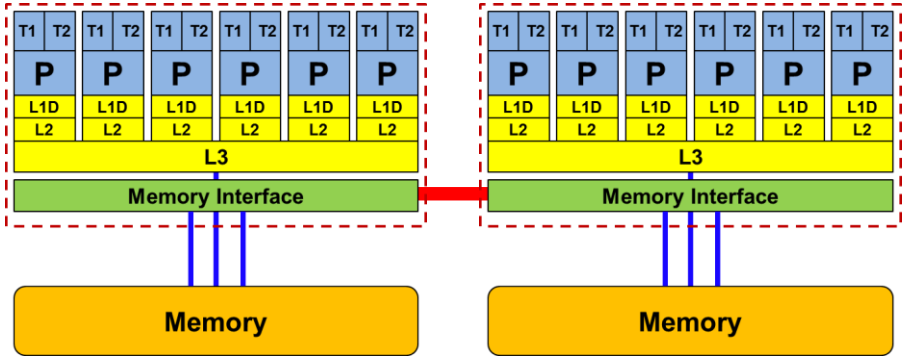


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system "anisotropy"

Today: Dual-socket node

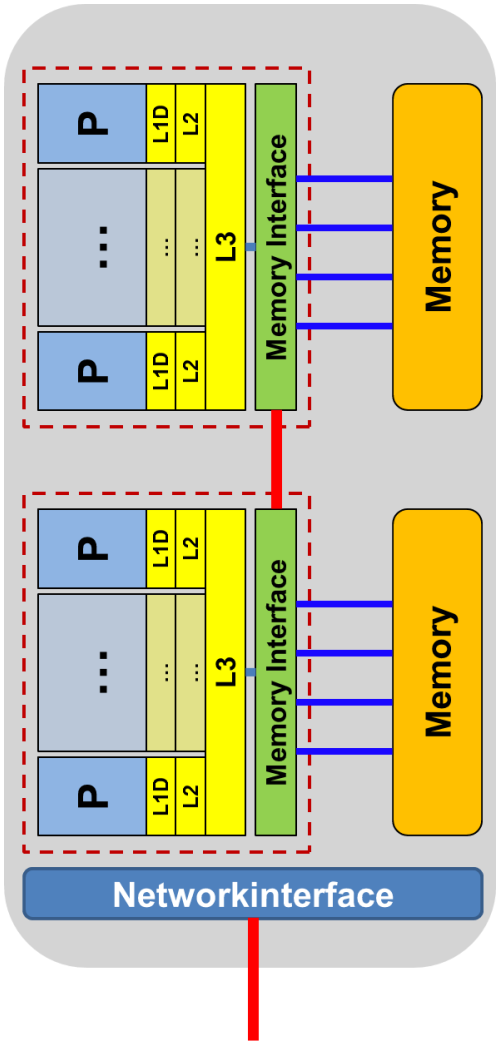


Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

HT / QPI provide scalable bandwidth at the price of ccNUMA architectures: *Where does my data finally end up?*

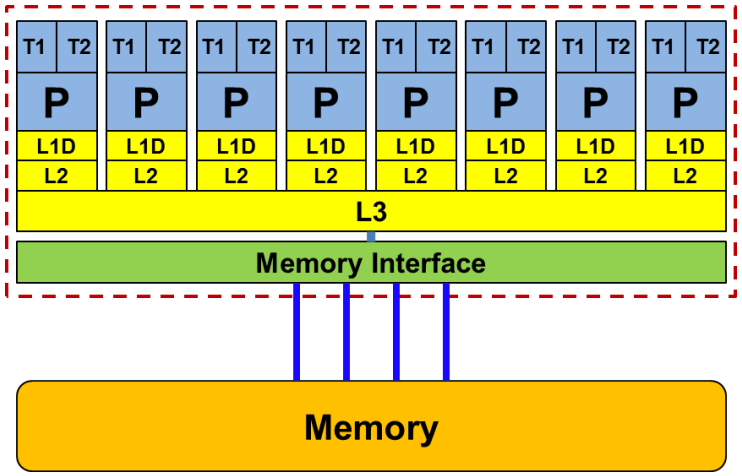
On AMD it is even more complicated → ccNUMA within a socket!

Cray XC30 “SandyBridge-EP” 8-core dual socket node



- 8 cores per socket 2.7 GHz (3.5 @ turbo)
- DDR3 memory interface with 4 channels per chip
- Two-way SMT
- Two 256-bit SIMD FP units
  - SSE4.2, AVX
- 32 kB L1 data cache per core
- 256 kB L2 cache per core
- 20 MB L3 cache per chip

There is no single driving force for chip performance!



Intel Xeon
"Sandy Bridge EP" socket
4,6,8 core variants available

Floating Point (FP) Performance:

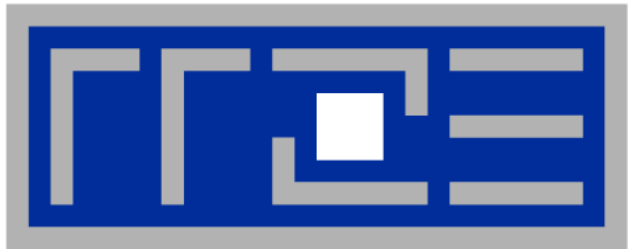
$$P = n_{core} * F * S * \nu$$

n_{core}	number of cores:	8
F	FP instructions per cycle: (1 MULT and 1 ADD)	2
S	FP ops / instruction: (256 Bit SIMD registers – "AVX")	4 (dp) / 8 (sp)
ν	Clock speed :	~2.7 GHz

TOP500 rank 1 (mid-90s)

P = 173 GF/s (dp) / 346 GF/s (sp)

But: P=5.4 GF/s for serial, non-SIMD code



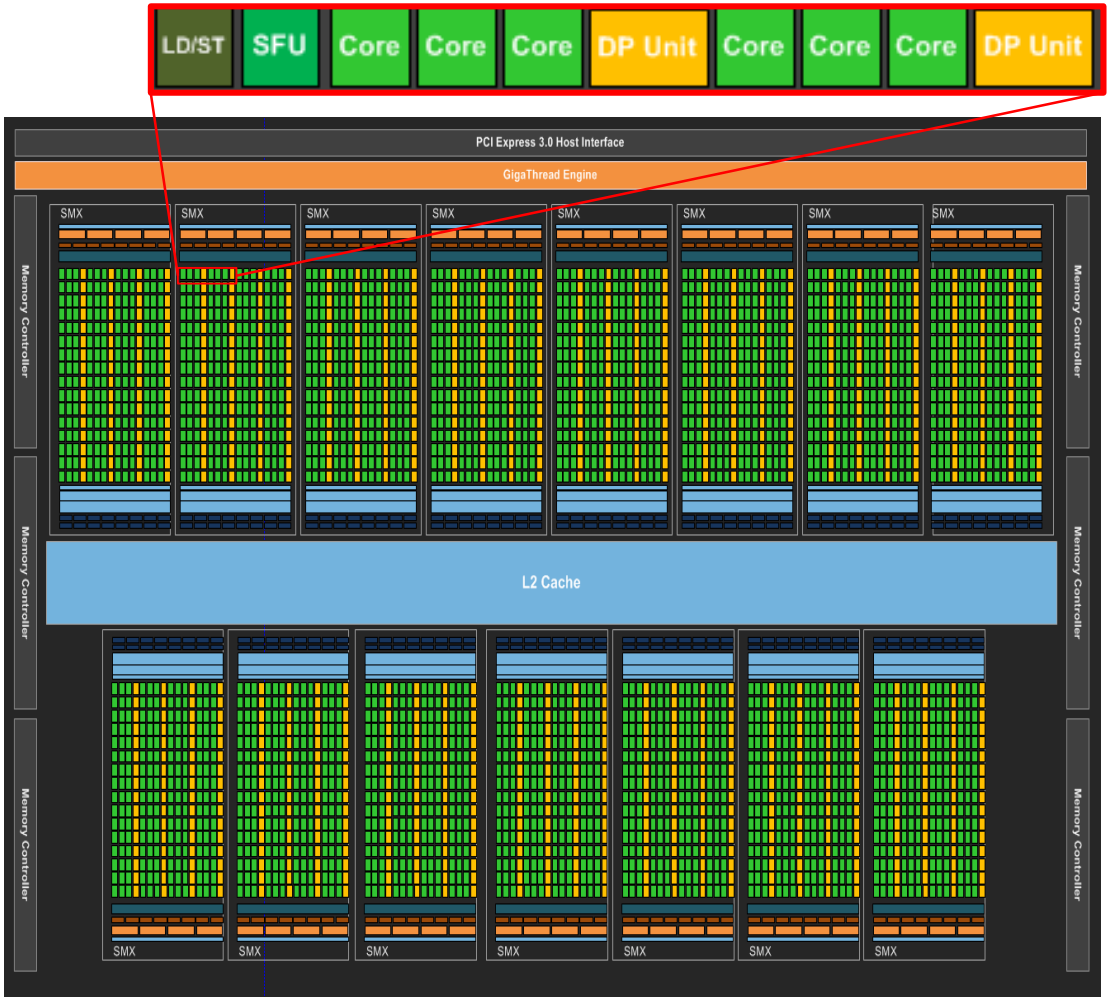
Interlude:
A glance at current accelerator technology

NVIDIA Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 “SMX” units
  - 192 (SP) “cores” each
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

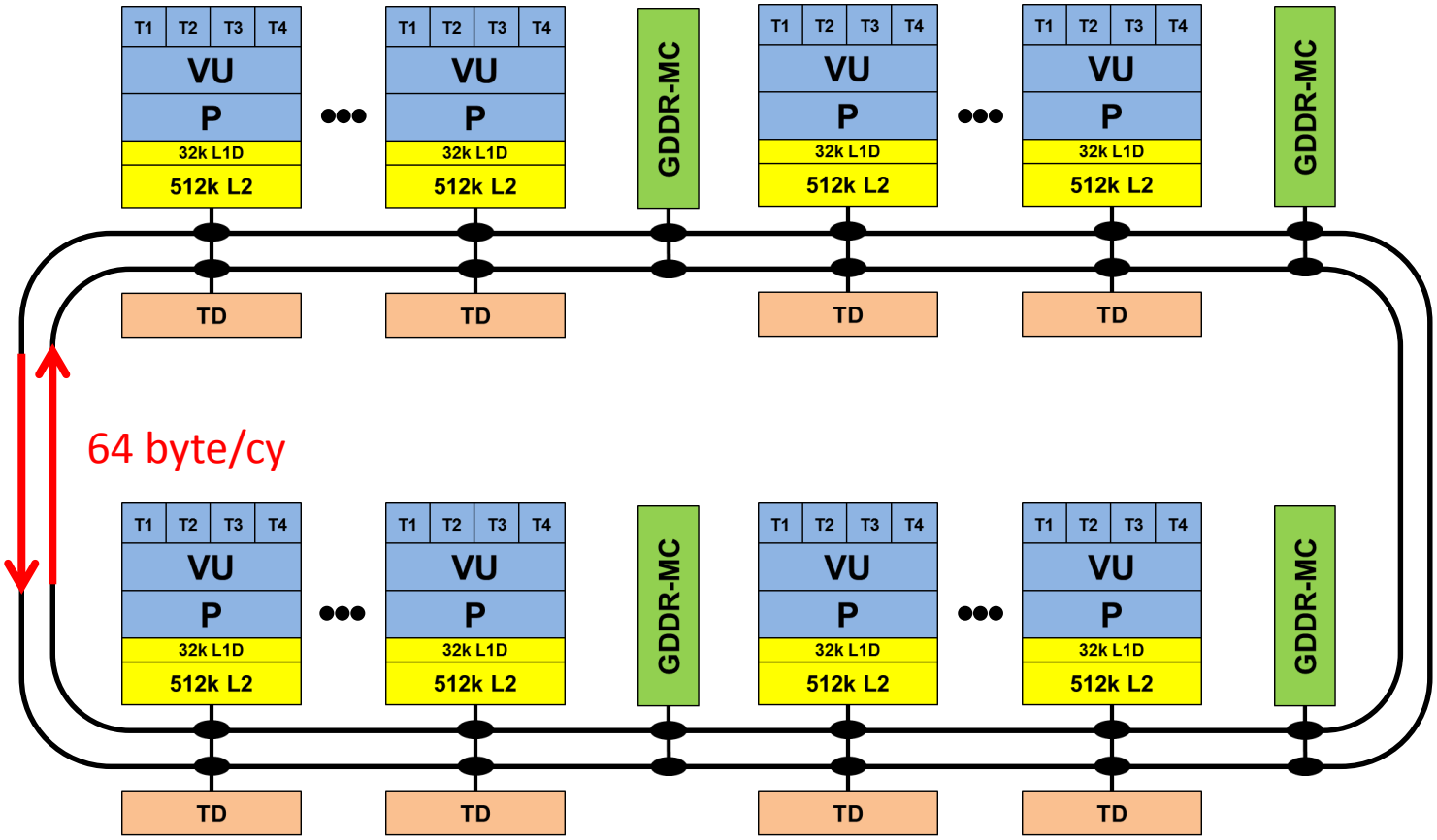
- 3:1 SP:DP performance



Intel Xeon Phi block diagram

Architecture

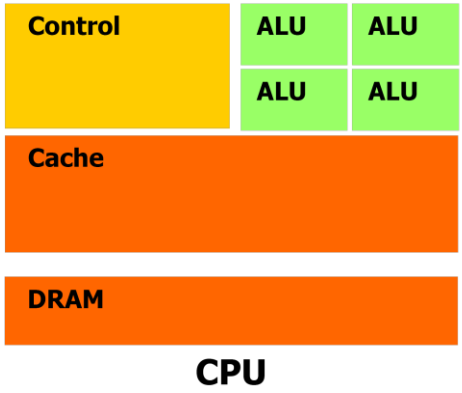
- 3B Transistors
- 60+ cores
- 512 bit SIMD
- ≈ 1 TFLOP DP peak
- 0.5 MB L2/core
- GDDR5
- 2:1 SP:DP performance

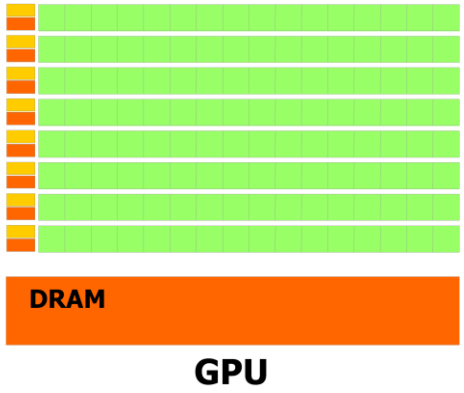


Trading single thread performance for parallelism:
GPGPUs vs. CPUs

GPU vs. CPU light speed estimate:

1. **Compute bound:** **2-10x**
2. **Memory Bandwidth:** **1-5x**



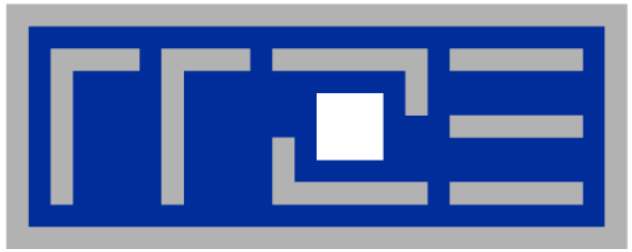


	Intel Core i5 – 2500 (“Sandy Bridge”)	Intel Xeon E5-2680 DP node (“Sandy Bridge”)	NVIDIA K20x (“Kepler”)
Cores@Clock	4 @ 3.3 GHz	2 x 8 @ 2.7 GHz	2880 @ 0.7 GHz
Performance⁺/core	52.8 GFlop/s	43.2 GFlop/s	1.4 GFlop/s
Threads@STREAM	<4	<16	>8000
Total performance⁺	210 GFlop/s	691 GFlop/s	4,000 GFlop/s
Stream BW	18 GB/s	2 x 40 GB/s	168 GB/s (ECC=1)
Transistors / TDP	1 Billion* / 95 W	2 x (2.27 Billion/130W)	**7.1 Billion/250W**

⁺ *Single Precision*

* *Includes on-chip GPU and PCI-Express*

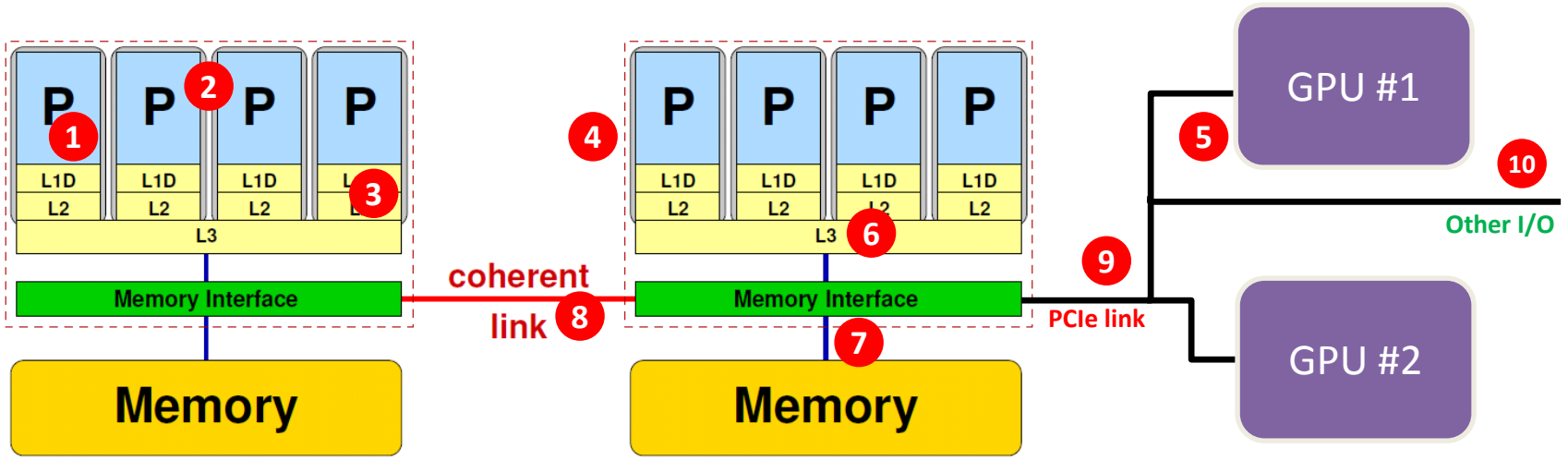
Complete compute device



Node topology and programming models

Parallelism in a modern compute node

- **Parallel and shared resources within a shared-memory node**



Parallel resources:

- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

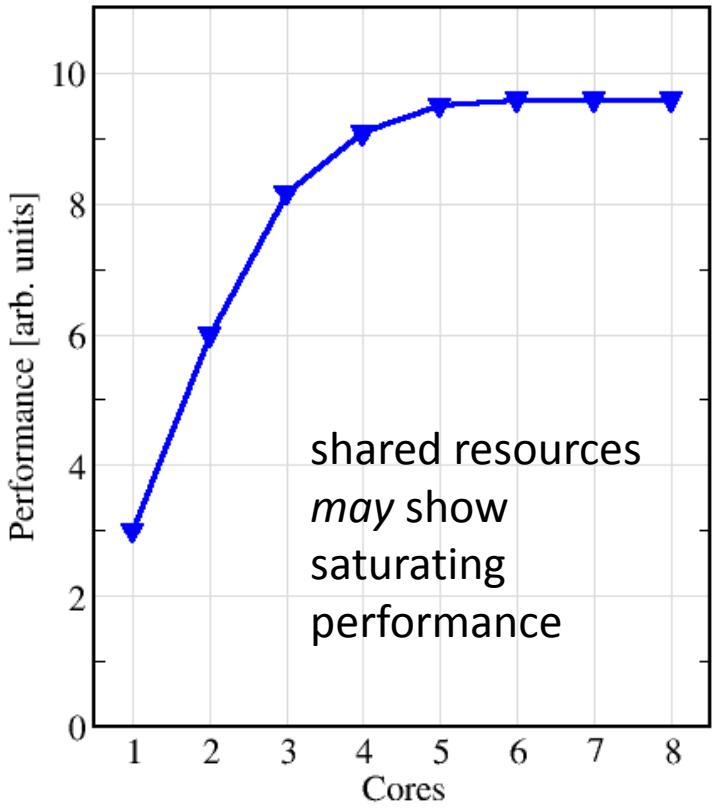
Shared resources:

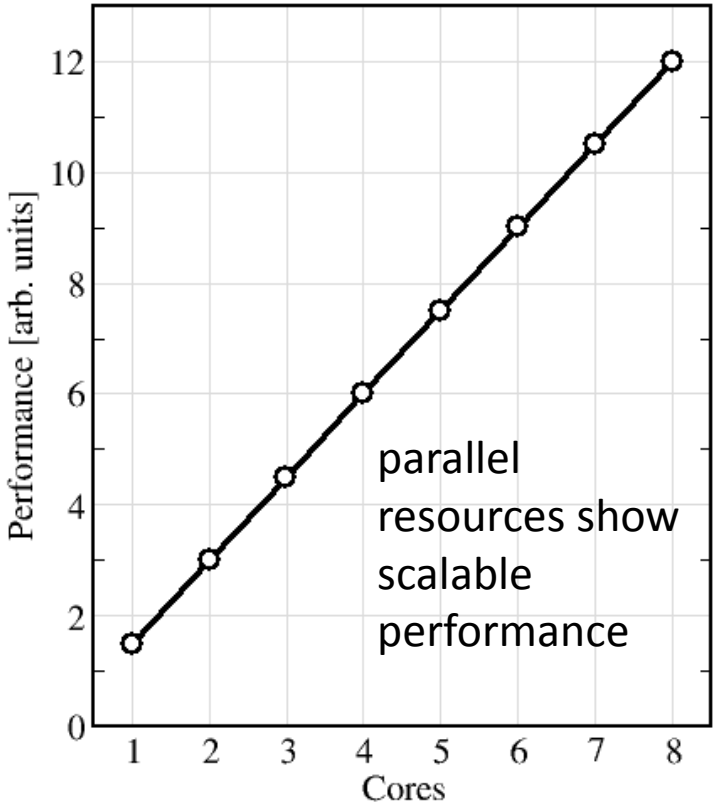
- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

How does your application react to all of those details?

Scalable and saturating behavior

- Clearly distinguish between "**saturating**" and "**scalable**" performance on the chip level

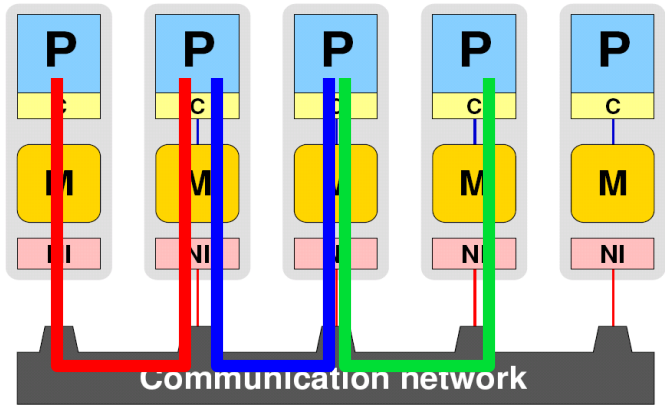


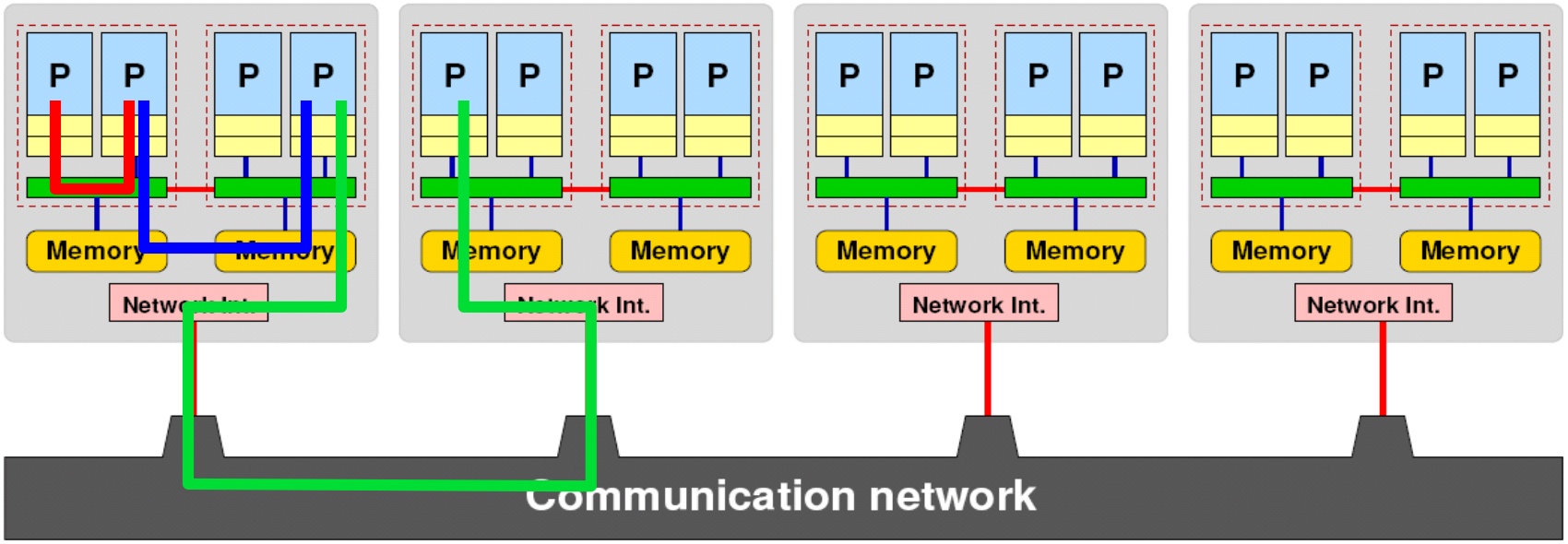


- **Shared-memory (intra-node)**
  - Good old MPI
  - OpenMP
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs,… you name it
- **Distributed-memory (inter-node)**
  - MPI
  - PVM (gone)
- **Hybrid**
  - Pure MPI
  - MPI+OpenMP
  - MPI + any shared-memory model
  - MPI (+OpenMP) + CUDA/OpenCL/…

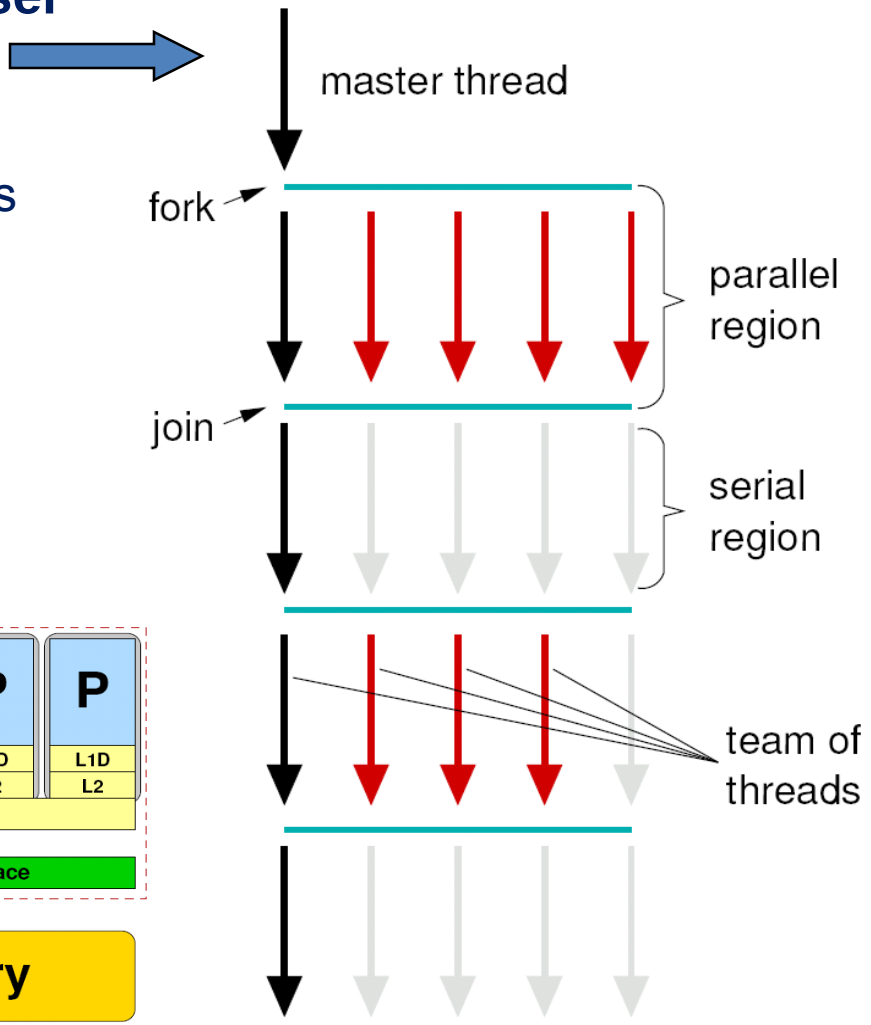
All models require awareness of *topology* and *affinity* issues for getting best performance out of the machine!

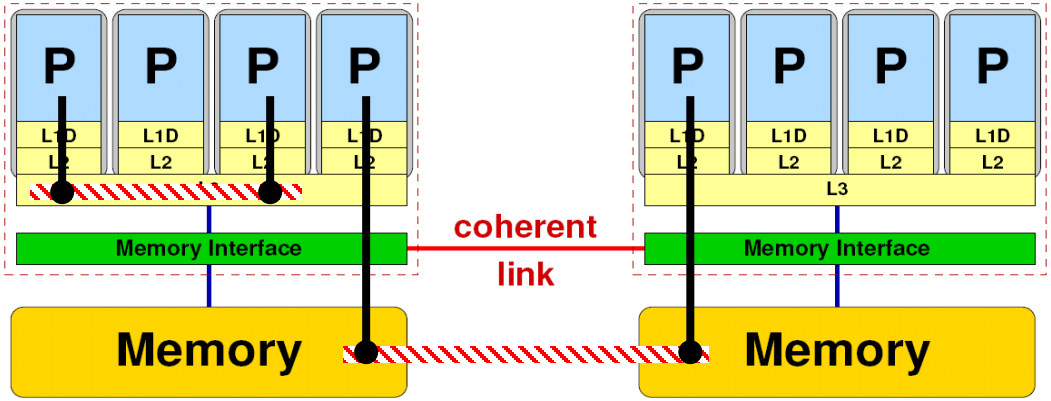
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology





- **Machine structure is invisible to user**
  - → Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,…) should know about the details
- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology





Parallel programming models: Lots of choices

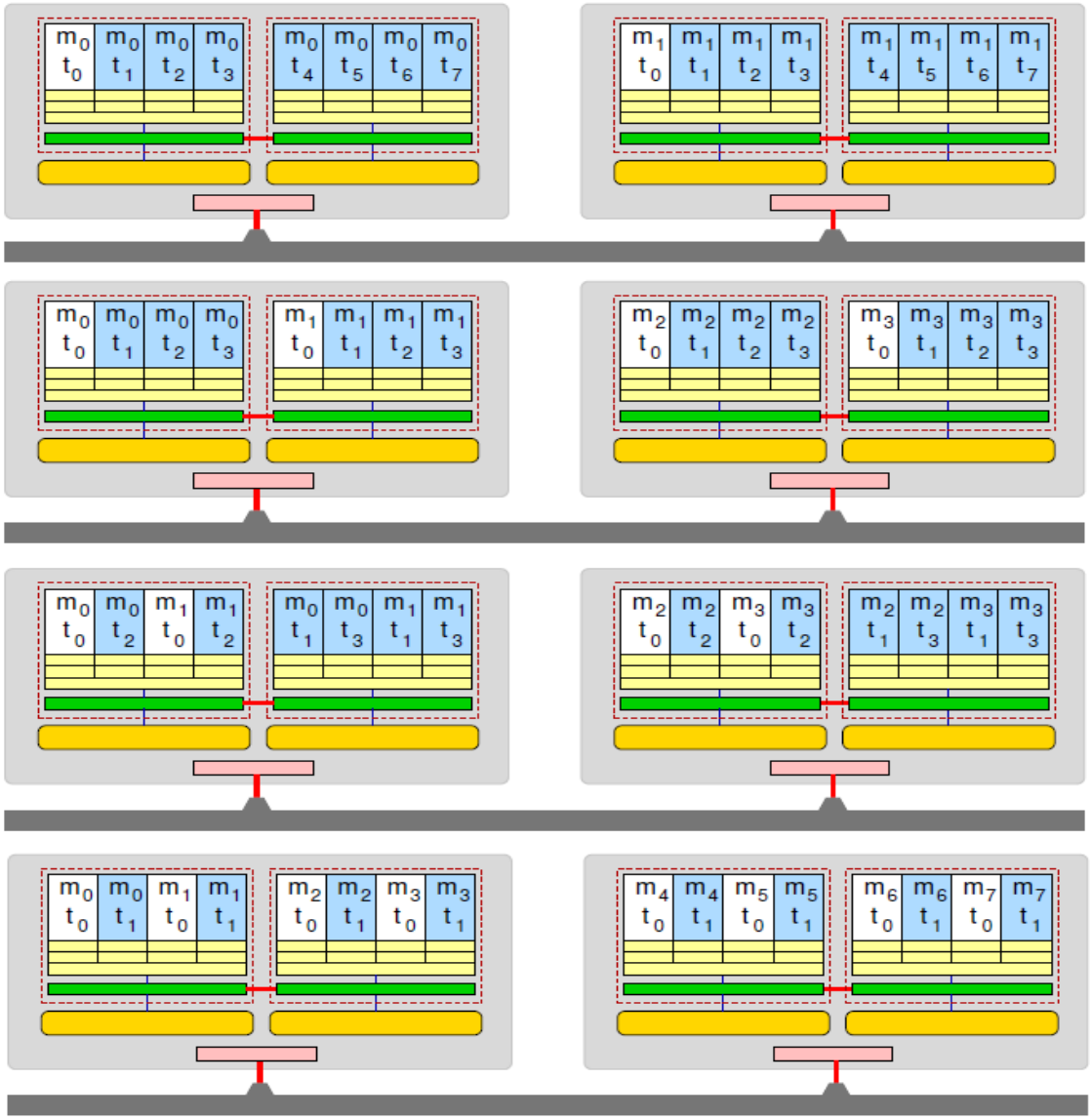
Hybrid MPI+OpenMP on a multicore multisocket cluster

One MPI process / node

One MPI process / socket:
OpenMP threads on same socket: **“blockwise”**

OpenMP threads pinned **“round robin”** across cores in node

Two MPI processes / socket
OpenMP threads on same socket



Conclusions about architecture

- **Modern computer architecture has a rich "topology"**
- **Node-level hardware parallelism takes many forms**
  - Sockets/devices – CPU: 1-8, GPGPU: 1-6
  - Cores – moderate (CPU: 4-16) to massive (GPGPU: 1000's)
  - SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10's-100's)
  - Superscalarity (CPU: 2-6)
- **Exploiting performance: parallelism + bottleneck awareness**
  - **"High Performance Computing" == computing at a bottleneck**
- **Performance of programs is sensitive to architecture**
  - Topology/affinity influences overheads of popular programming models
  - Standards do not contain (many) topology-aware features
    - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
  - Apart from overheads, performance features are largely independent of the programming model