

Modern computer architecture

An introduction for software developers

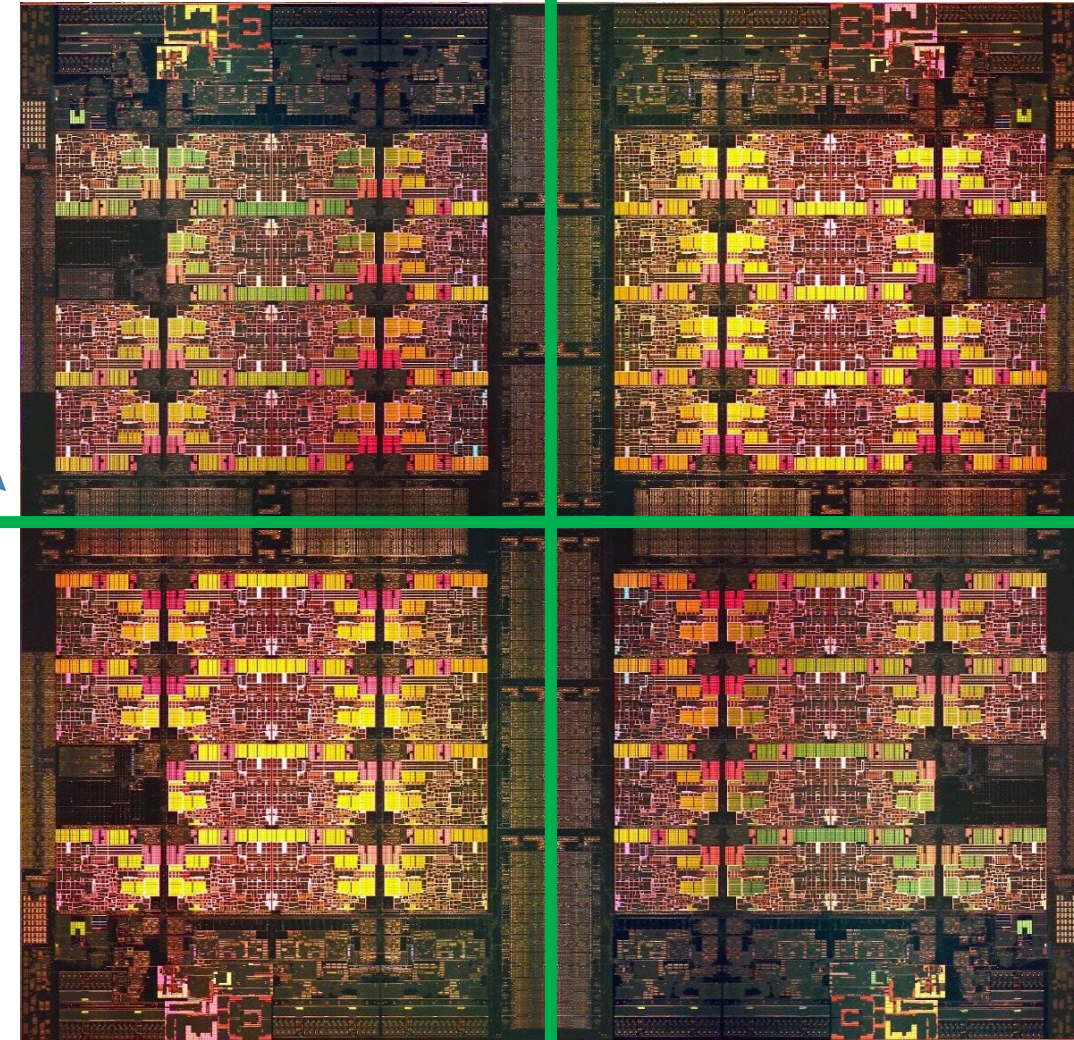
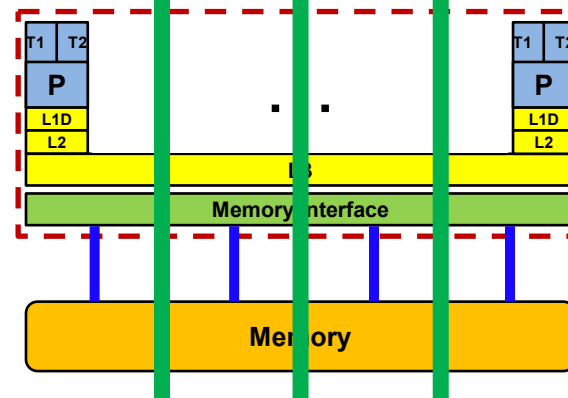


Multi-core today: Intel Xeon Sapphire Rapids (2023)

- Xeon “Sapphire Rapids” (Platinum/Gold/Silver/Bronze):
Up to 60 cores running at 1.7+ GHz
(+ “Turbo Mode” 4.8 GHz),
- Simultaneous Multithreading
→ reports as 120-way chip
- “Intel 7” process / up to 350 W
- Multi-die package (4 chips)
- Clock frequency:
flexible 😊

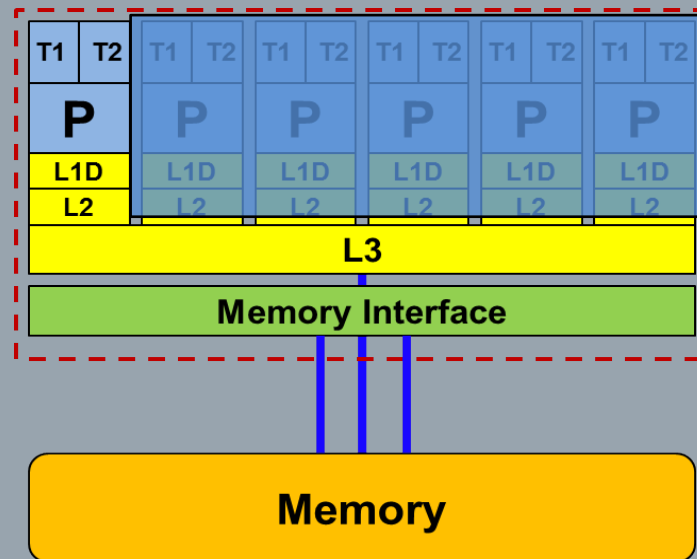
Optional: “Sub-NUMA Clustering” (SNC) mode
boot option

→ One memory domain
per die

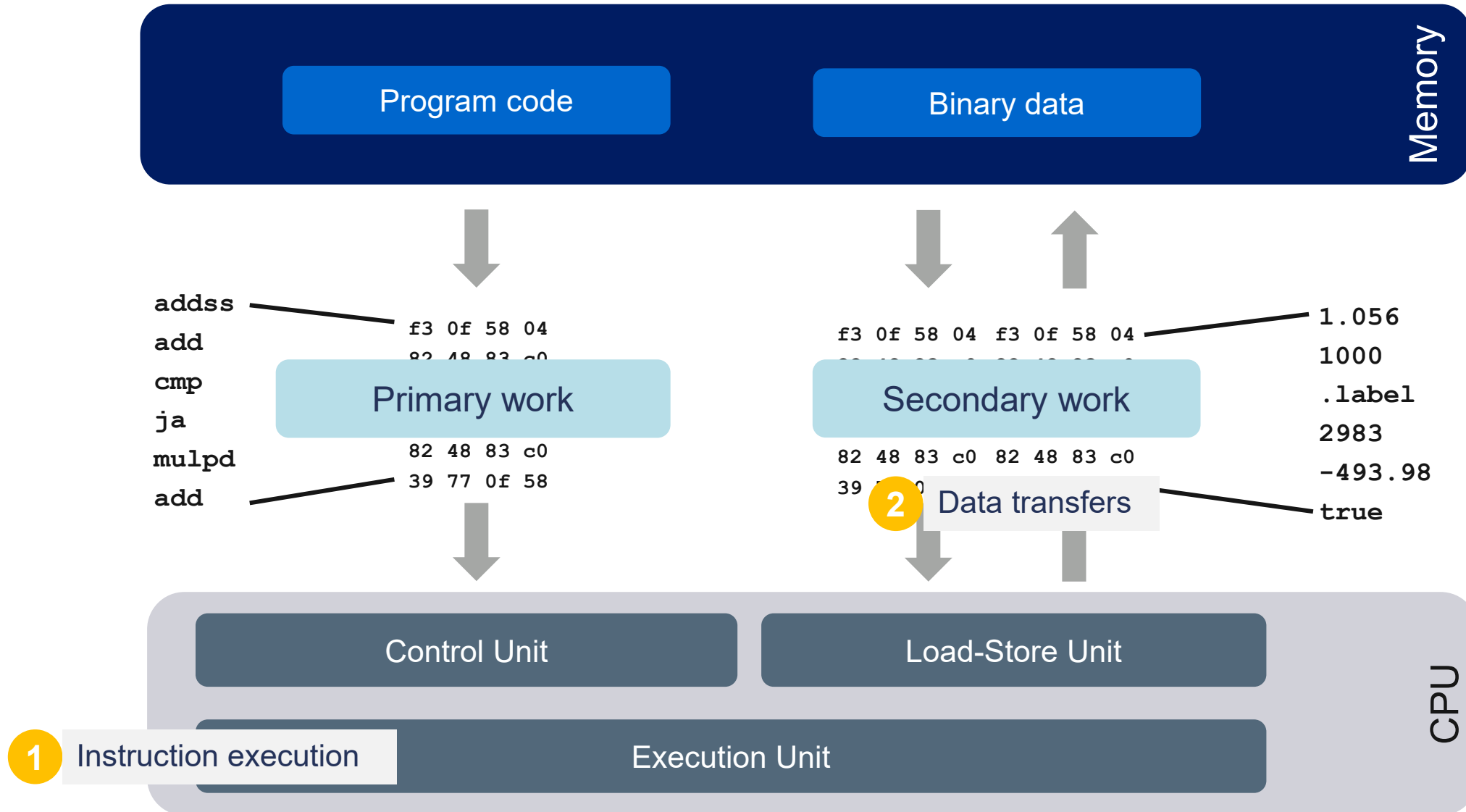


<https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated>

A deeper dive into core architecture



Stored Program Computer



From high level code to actual execution

```
for(int i=0; i<N; i++){  
    sum += a[i];  
}
```

addsd: Add 2nd argument to 1st argument and store result in 1st argument

Counter increment

Compare register content

Conditional jump to label if loop continues

Compiler

Load a[i] to register xmm2

&a[0]

```
..LABEL:  
    movsd  xmm2, [rdi+rdx*8]  
    addsd  xmm1, xmm2  
    inc    rdx  
    cmp    rax, rdx  
    jb     ..LABEL
```

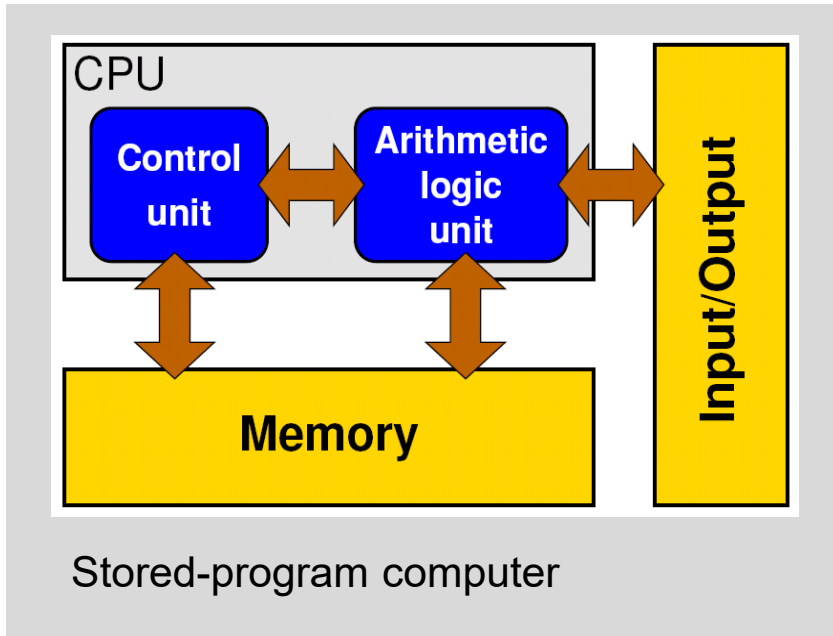
sizeof(double)

N in register **rax**

sum in register **xmm1**

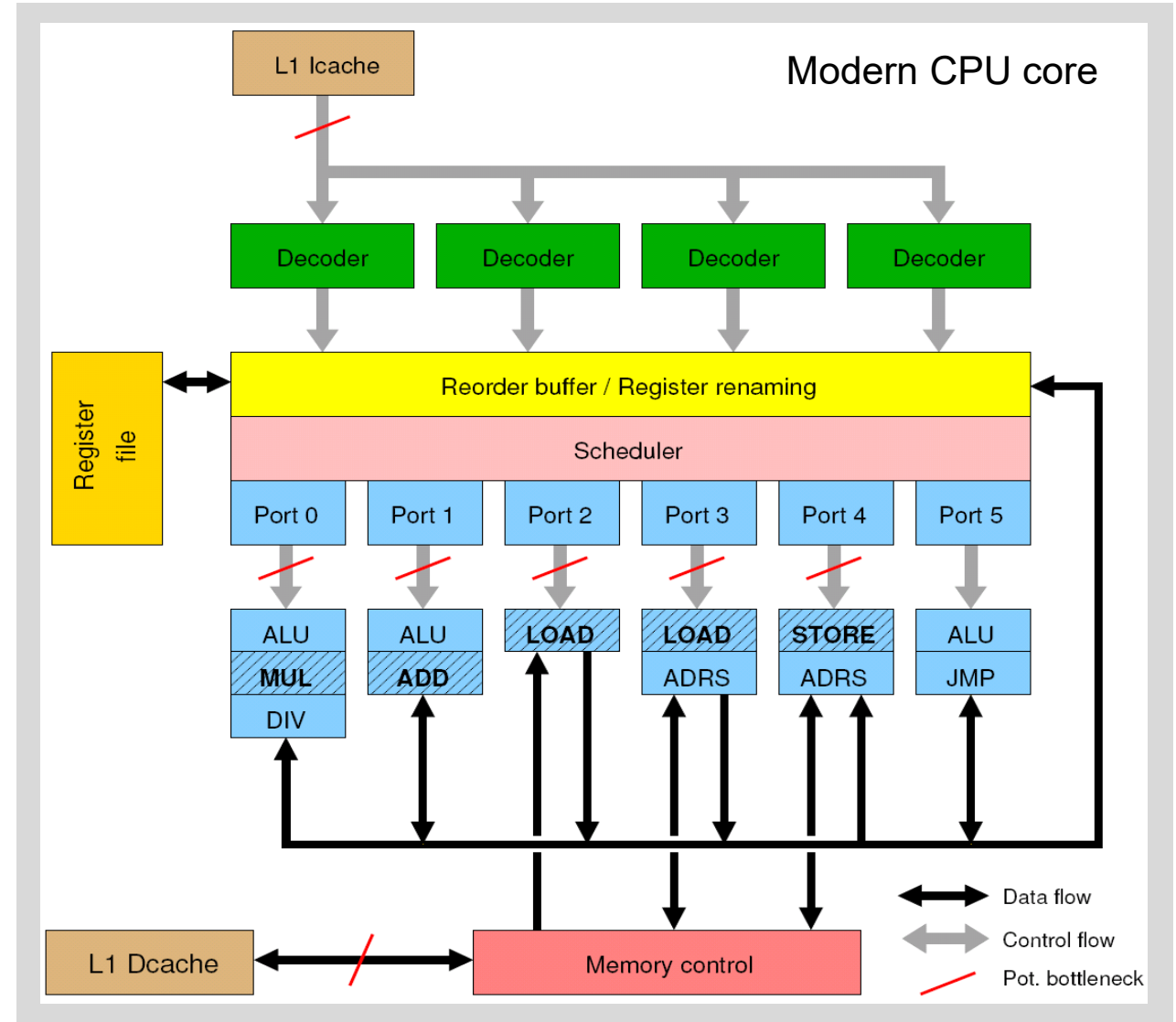
i in register **rdx**

General-purpose cache based microprocessor core



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

The **clock cycle** is the “**heartbeat**” of the core



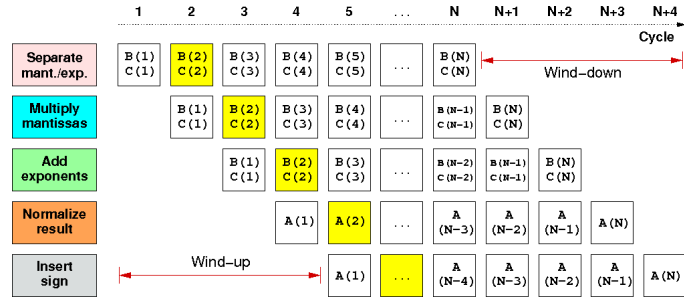
In-core features

Pipelining, Superscalarity, SIMD, SMT

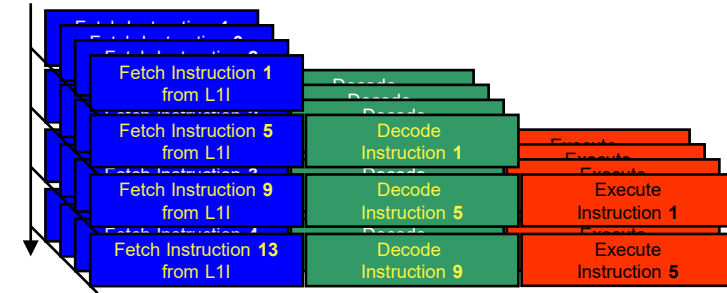


Important in-core features

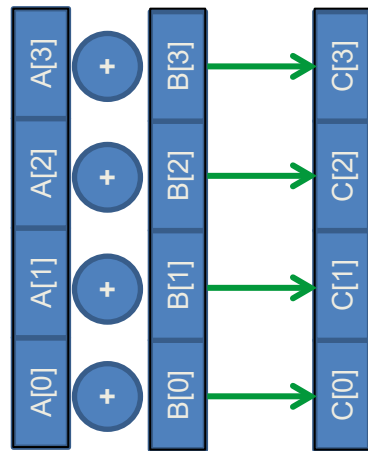
Pipelining:
Instruction execution in multiple steps



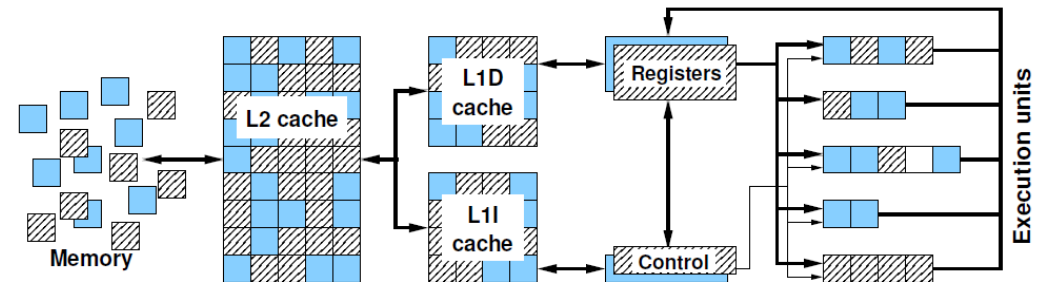
Superscalarity:
Multiple instructions per cycle



Single Instruction Multiple Data:
Multiple operations per instruction



Simultaneous Multi-Threading:
Multiple instruction sequences in parallel



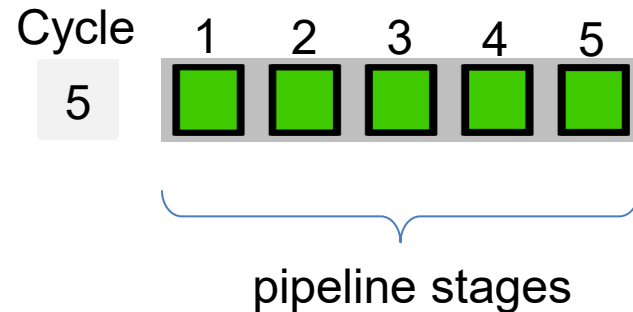
Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining

Independent instructions
(of one kind, e.g., ADD):



Single instruction takes 5 cycles (latency)



Throughput:

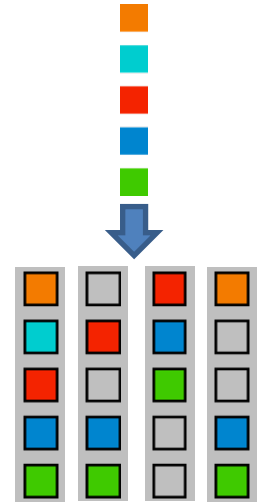
1 instruction per cycle after pipeline is full

→ 5x speedup

Superscalar execution

across multiple pipelines

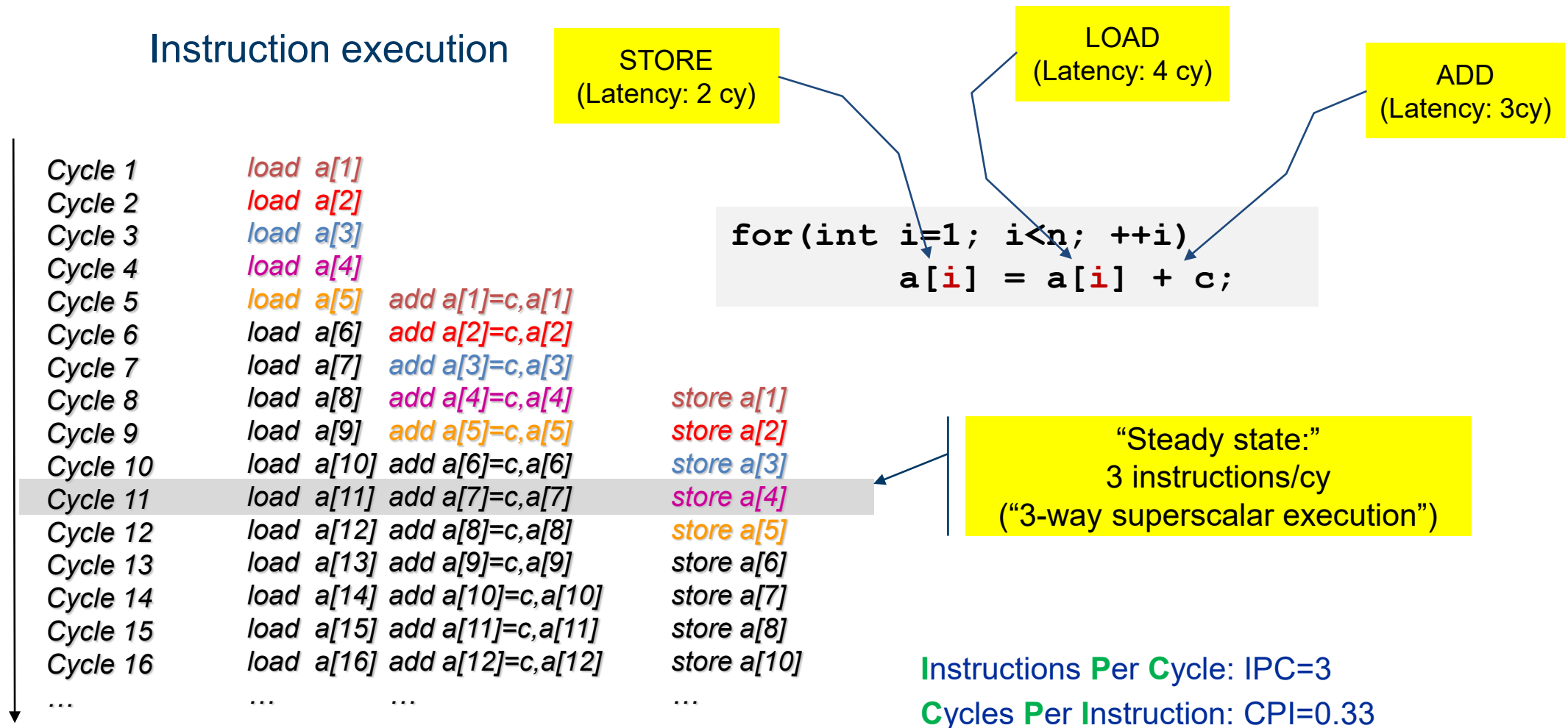
4-way superscalar:



→ Massive boost in
instruction throughput

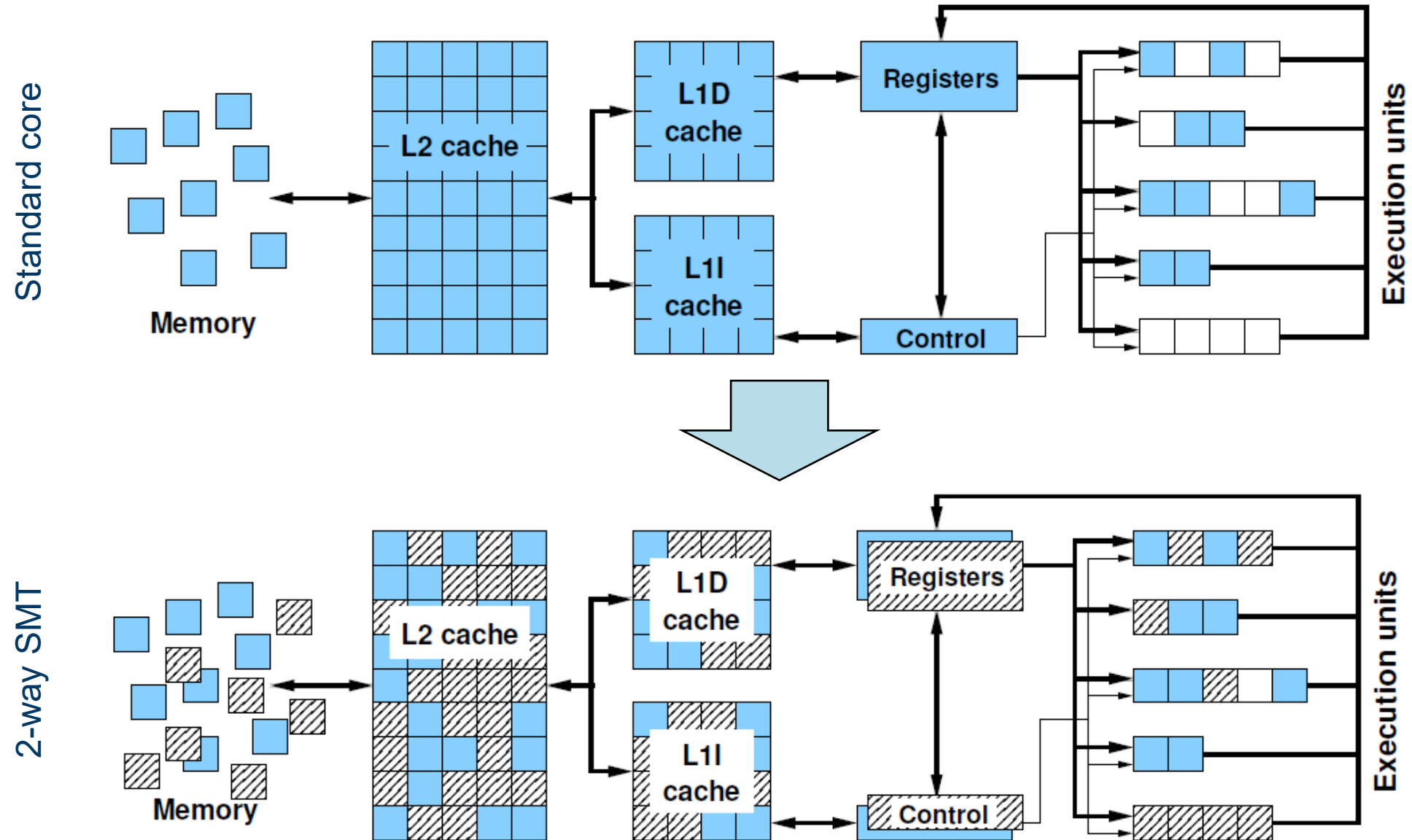
→ Instructions can be
reordered on the fly

Superscalar out-of-order execution and steady state



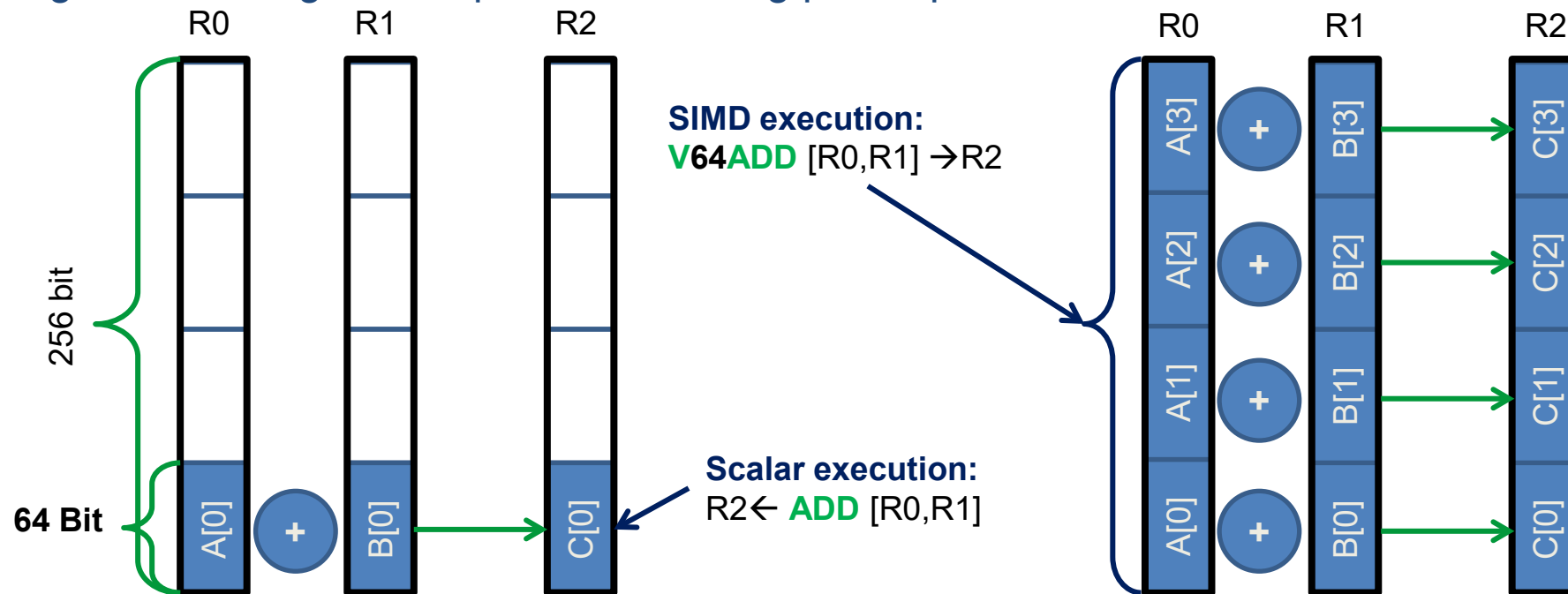
Hardware takes care of executing instructions as soon as their operands are available:
Out-Of-Order (OOO) execution

Simultaneous multi-threading (SMT)

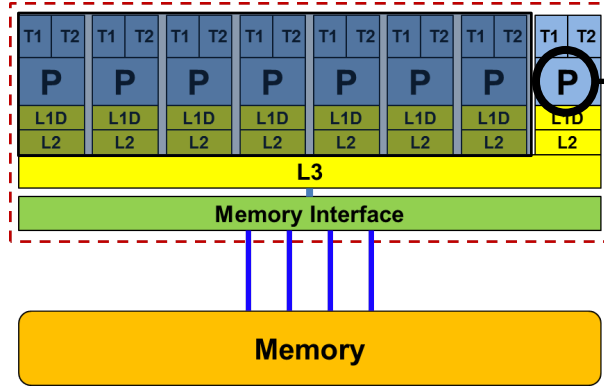


SIMD processing

- **Single Instruction Multiple Data (SIMD)** operations allow the execution of the **same operation** on “wide” registers from a **single instruction**
- 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
 - AVX-512: ... you guessed it!
- Adding two registers holding double precision floating point operands:



Single-core DP floating-point performance



$$P_{core} = n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$

Super-scalarity

FMA factor

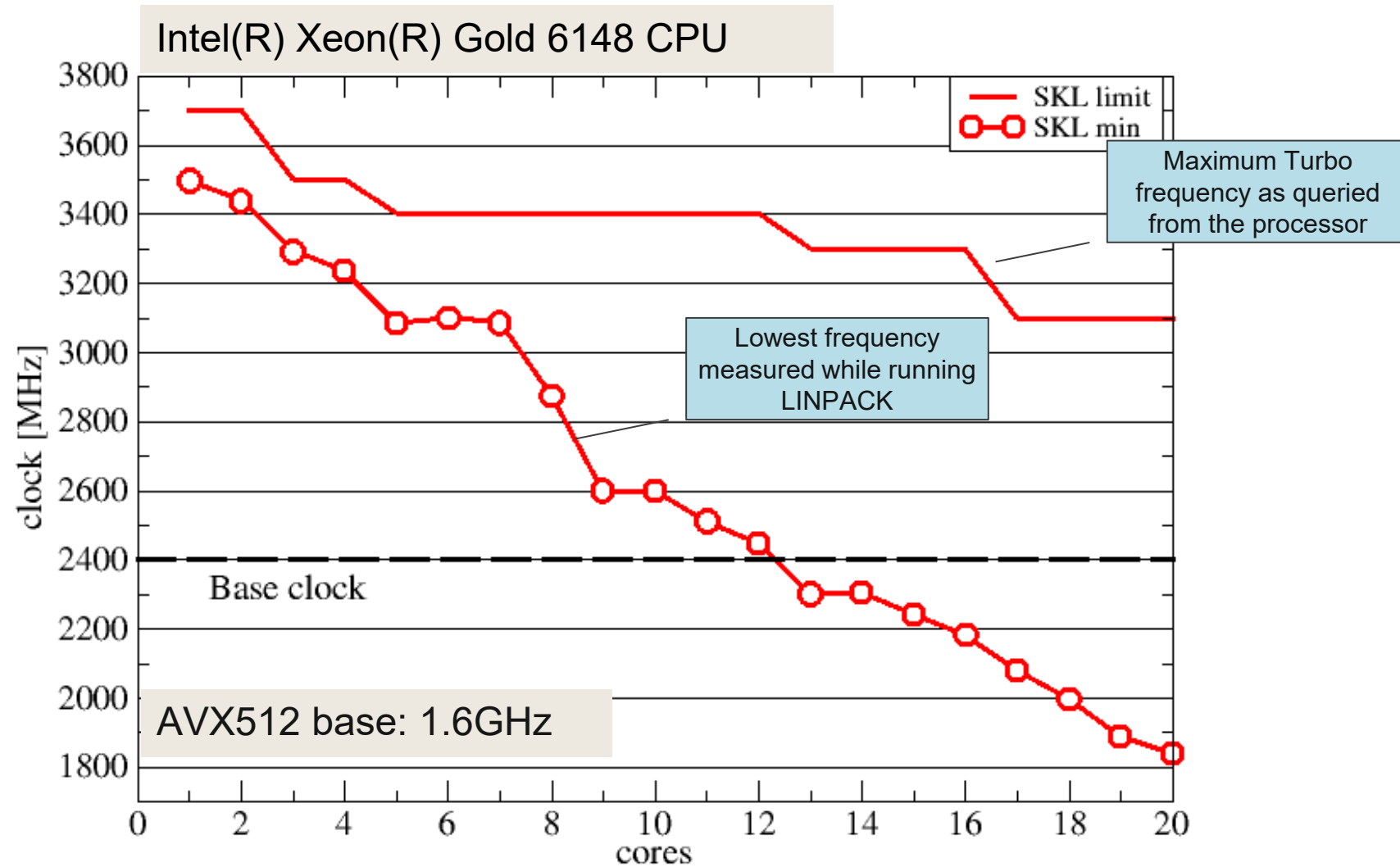
SIMD factor

Clock Speed

Typical representatives	n_{super}^{FP} [inst./cy]	n_{FMA}	n_{SIMD} [ops/inst.]	@market	Ex. model	f [Gcy/s]	P_{core} [GF/s]
Intel Nehalem	2	1	2	Q1/2009	X5570	2.93	11.7
Intel Sandy Bridge	2	1	4	Q1/2012	E5-2680	2.7	21.6
Intel Haswell	2	2	4	Q3/2014	E5-2695 v3	2.3	36.8
Intel Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
Intel Ice Lake	2	2	8	Q2/2021	Platinum 8360Y	2.4	76.8
AMD Zen (Naples)	2	2	2	Q1/2017	Epyc 7451	2.3	18.4
AMD Zen2 (Rome)	2	2	4	Q4/2019	Epyc 7642	2.3	36.8
AMD Zen3 (Milan)	2	2	4	Q4/2020	Epyc 7713	2.0	32.0
Fujitsu A64FX	2	2	8	Q2/2020	FX700	1.8	57.6

Multi-core today: Turbo mode

The processor **dynamically** overclocks to exploit more of the **TDP** envelope if fewer cores are active




Example: The sum reduction



A “simple” example: The sum reduction

```
for (int i=0; i<N; i++) {  
    sum += a[i];  
}
```



...In **single precision** on an **AVX**-capable core (ADD latency = 3 cy)

How fast can this loop possibly run with data in the L1 cache?

- **Loop-carried dependency** on summation variable
- Execution **stalls** at every ADD until previous ADD is complete

→ No pipelining?

→ No SIMD?

Applicable peak for the sum reduction (I)

Plain scalar code, no SIMD

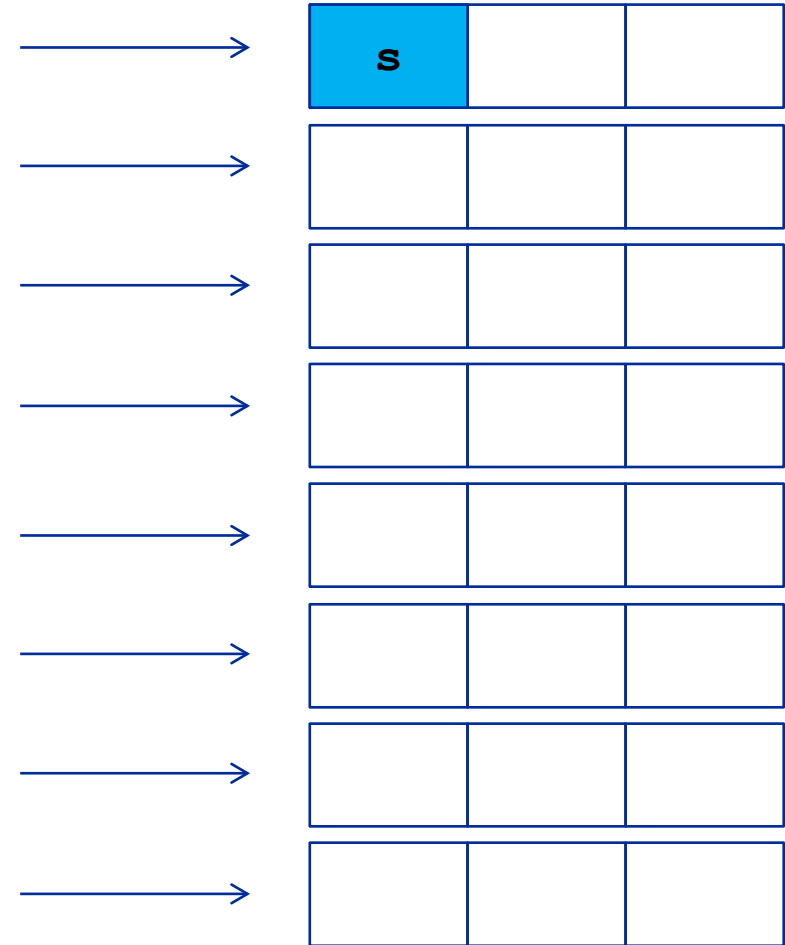
```
for (int i=0; i<N; i++){  
    sum += a[i];  
}
```

```
LOAD r1.0 ← 0  
i ← 1  
loop:  
    LOAD r2.0 ← a(i)  
    ADD r1.0 ← r1.0 + r2.0  
    ++i →? loop  
result ← r1.0
```

SIMD lane

SIMD lanes

ADD pipes utilization:



→ 1/24 of ADD peak

Applicable peak for the sum reduction (II)

Scalar code, 3-way “modulo variable expansion”

```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1
```

loop:

```
LOAD r4.0 ← a(i)
LOAD r5.0 ← a(i+1)
LOAD r6.0 ← a(i+2)
```

```
ADD r1.0 ← r1.0 + r4.0 # scalar ADD
ADD r2.0 ← r2.0 + r5.0 # scalar ADD
ADD r3.0 ← r3.0 + r6.0 # scalar ADD
```

```
i+=3 →? loop
result ← r1.0+r2.0+r3.0
```

```
for (int i=0; i<N; i+=3) {
    s1 += a[i+0];
    s2 += a[i+1];
    s3 += a[i+2];
}
sum = sum + s1+s2+s3;
```

s1	s2	s3

→ 1/8 of ADD peak

Applicable peak for the sum reduction (III)

SIMD vectorization (8-way MVE) x
pipelining (3-way MVE)

```
LOAD [r1.0,...,r1.7] ← [0,...,0]
LOAD [r2.0,...,r2.7] ← [0,...,0]
LOAD [r3.0,...,r3.7] ← [0,...,0]
i ← 1
```

loop:

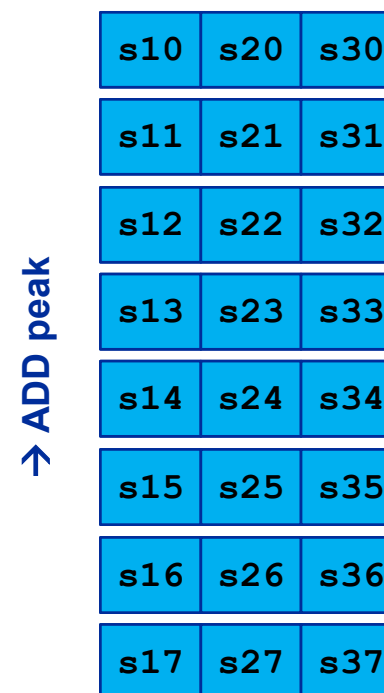
```
LOAD [r4.0,...,r4.7] ← [a(i),...,a(i+7)] # SIMD LOAD
LOAD [r5.0,...,r5.7] ← [a(i+8),...,a(i+15)] # SIMD
LOAD [r6.0,...,r6.7] ← [a(i+16),...,a(i+23)] # SIMD
```

```
ADD r1 ← r1 + r4 # SIMD ADD
ADD r2 ← r2 + r5 # SIMD ADD
ADD r3 ← r3 + r6 # SIMD ADD
```

i+=24 →? loop

result ← r1.0+r1.1+...+r3.6+r3.7

```
for (int i=0; i<N; i+=24){
  s10 += a[i+0]; s20 += a[i+8]; s30 += a[i+16];
  s11 += a[i+1]; s21 += a[i+9]; s31 += a[i+17];
  s12 += a[i+2]; s22 += a[i+10]; s32 += a[i+18];
  s13 += a[i+3]; s23 += a[i+11]; s33 += a[i+19];
  s14 += a[i+4]; s24 += a[i+12]; s34 += a[i+20];
  s15 += a[i+5]; s25 += a[i+13]; s35 += a[i+21];
  s16 += a[i+6]; s26 += a[i+14]; s36 += a[i+22];
  s17 += a[i+7]; s27 += a[i+15]; s37 += a[i+23];
}
sum = sum + s10+s11+...+s37;
```



Sum reduction

Questions

- When can this performance actually be achieved?
 - No **data transfer** bottlenecks
 - No other **in-core** bottlenecks
 - Need to execute (3 LOADs + 3 ADDs + 1 increment + 1 compare + 1 branch) in 3 cycles
- What does the **compiler** do?
 - If allowed and capable, the compiler will do this automatically
- Is the compiler **allowed** to do this at all?
 - Not according to language standards
 - High optimization levels can violate language standards
- What about the “accuracy” of the result?
 - Good question ;-)

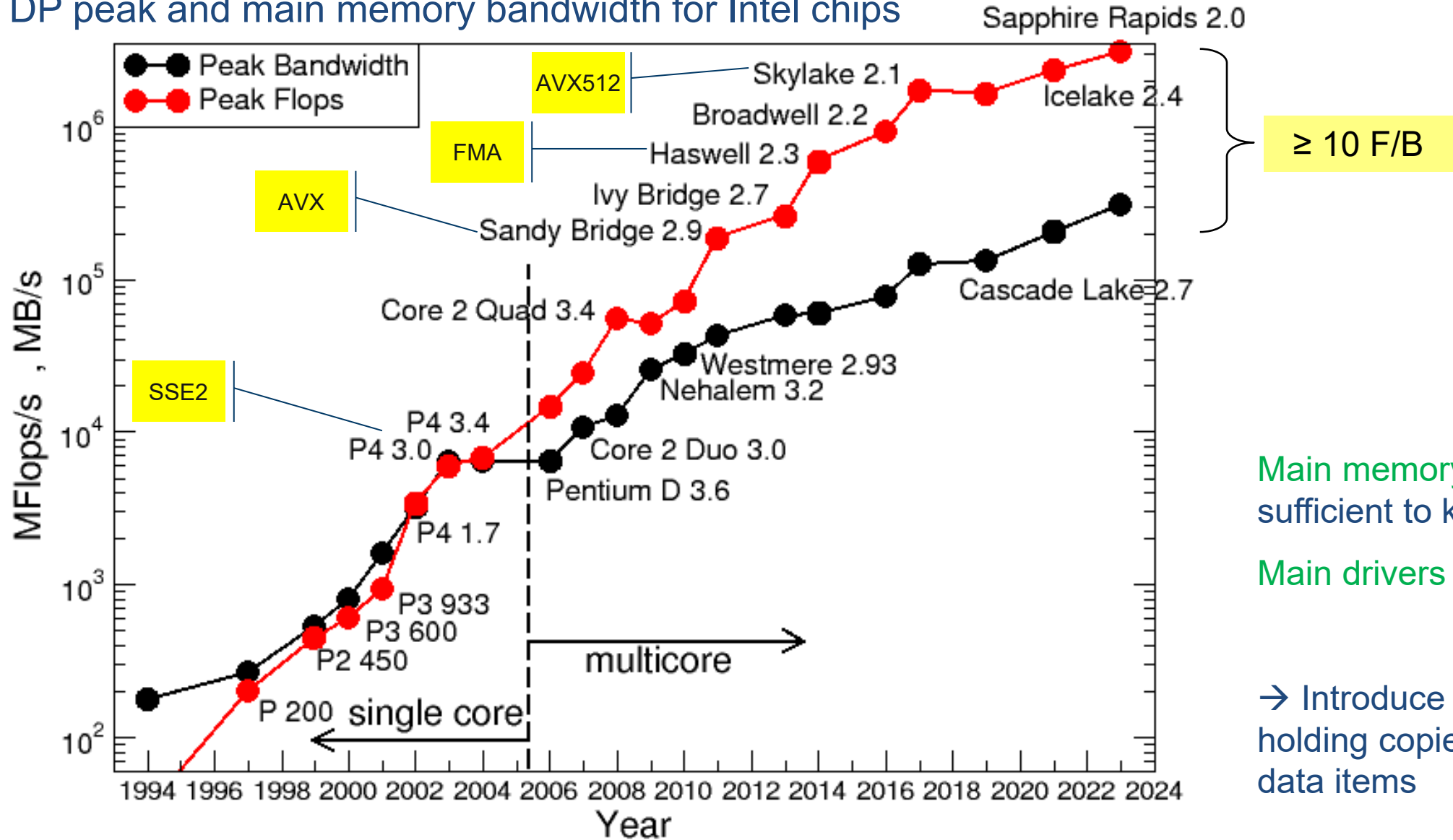
Memory Hierarchy

In-cache performance (L2, L3)
Main memory performance



Von Neumann bottleneck reloaded: “DRAM gap”

DP peak and main memory bandwidth for Intel chips

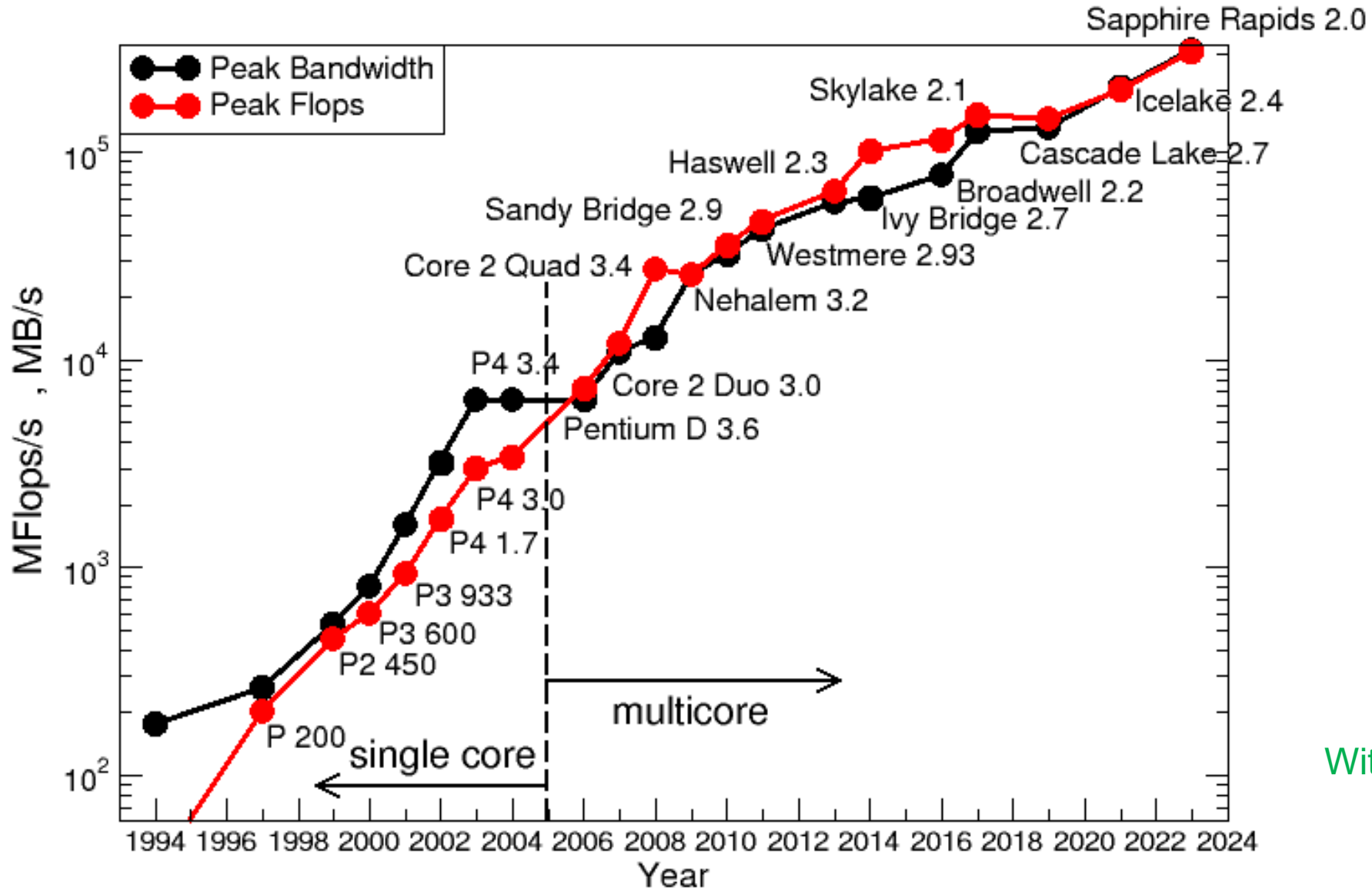


Main memory access speed not sufficient to keep CPU busy...

Main drivers of gap: SIMD, FMA

→ Introduce fast on-chip caches, holding copies of recently used data items

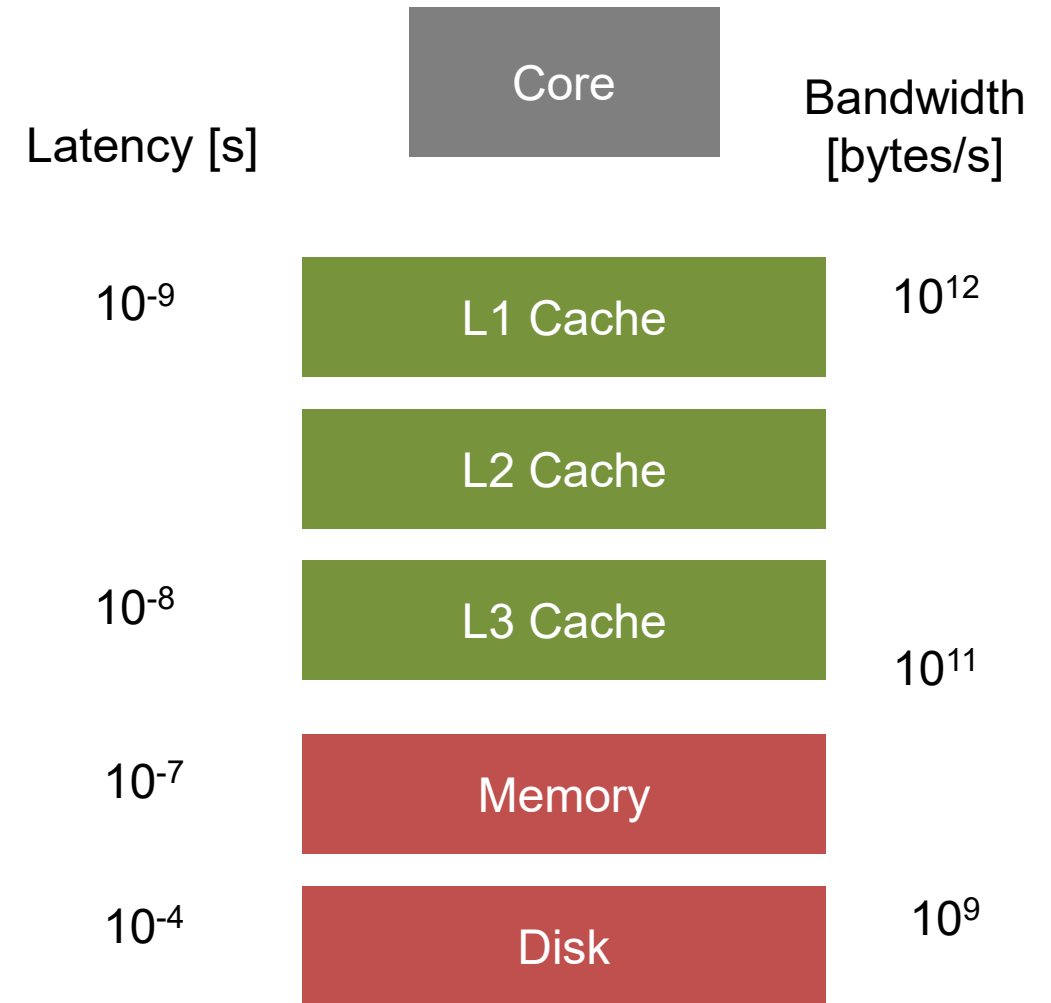
The “stripped” von Neumann bottleneck



Without SIMD, FMA: no gap!

Memory hierarchy

You can either build a **small** and **fast** memory or a **large** and **slow** memory



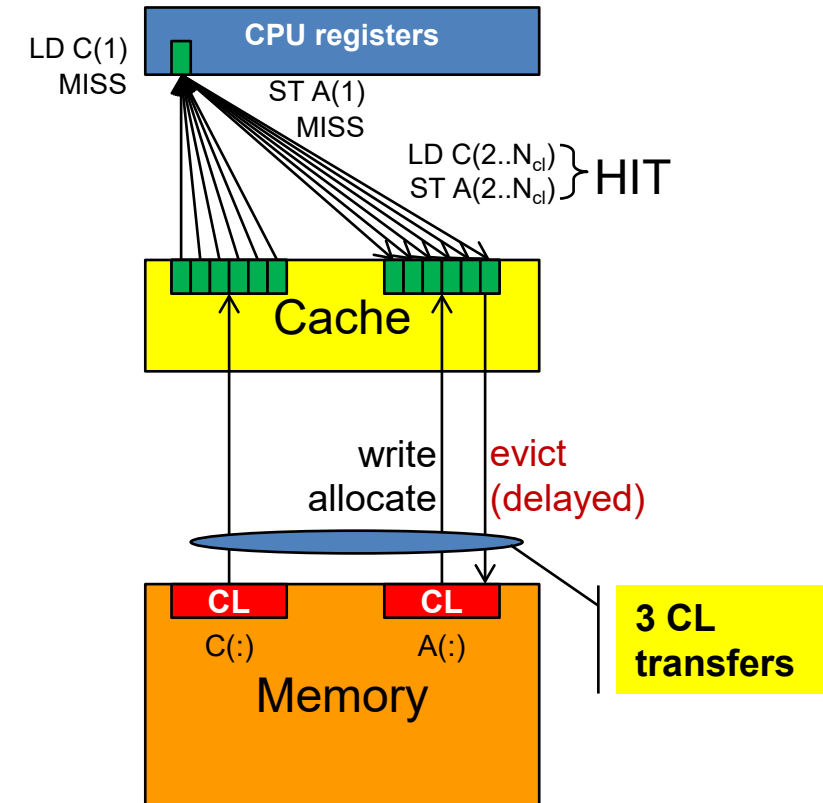
Purpose of many optimizations: use data in **fast memory**

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)
- Registers can only “talk” to the L1 cache
- MISS**: Load or store instruction does not find the data in a cache level
→ CL transfer required
- Example: Array copy $\mathbf{A}(:) = \mathbf{C}(:)$



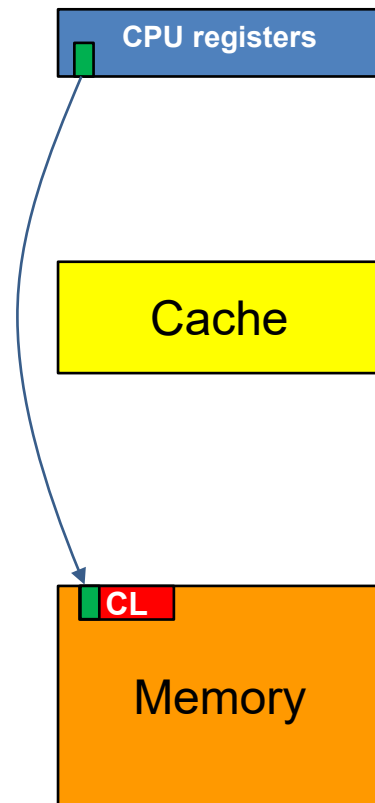
Avoiding the write-allocate transfer

Disadvantages of write-allocate:

- Cache pollution (if data not needed anytime soon)
- Additional data traffic

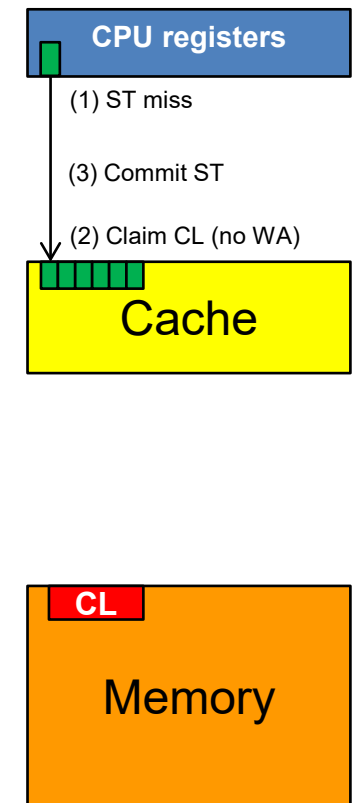
Solution 1: Nontemporal stores

- A.k.a. “streaming stores,” store instruction with a “nontemporal hint”
- Write “directly” to memory, ignoring the normal cache hierarchy
- Avoids cache pollution, but stored data ends up in memory

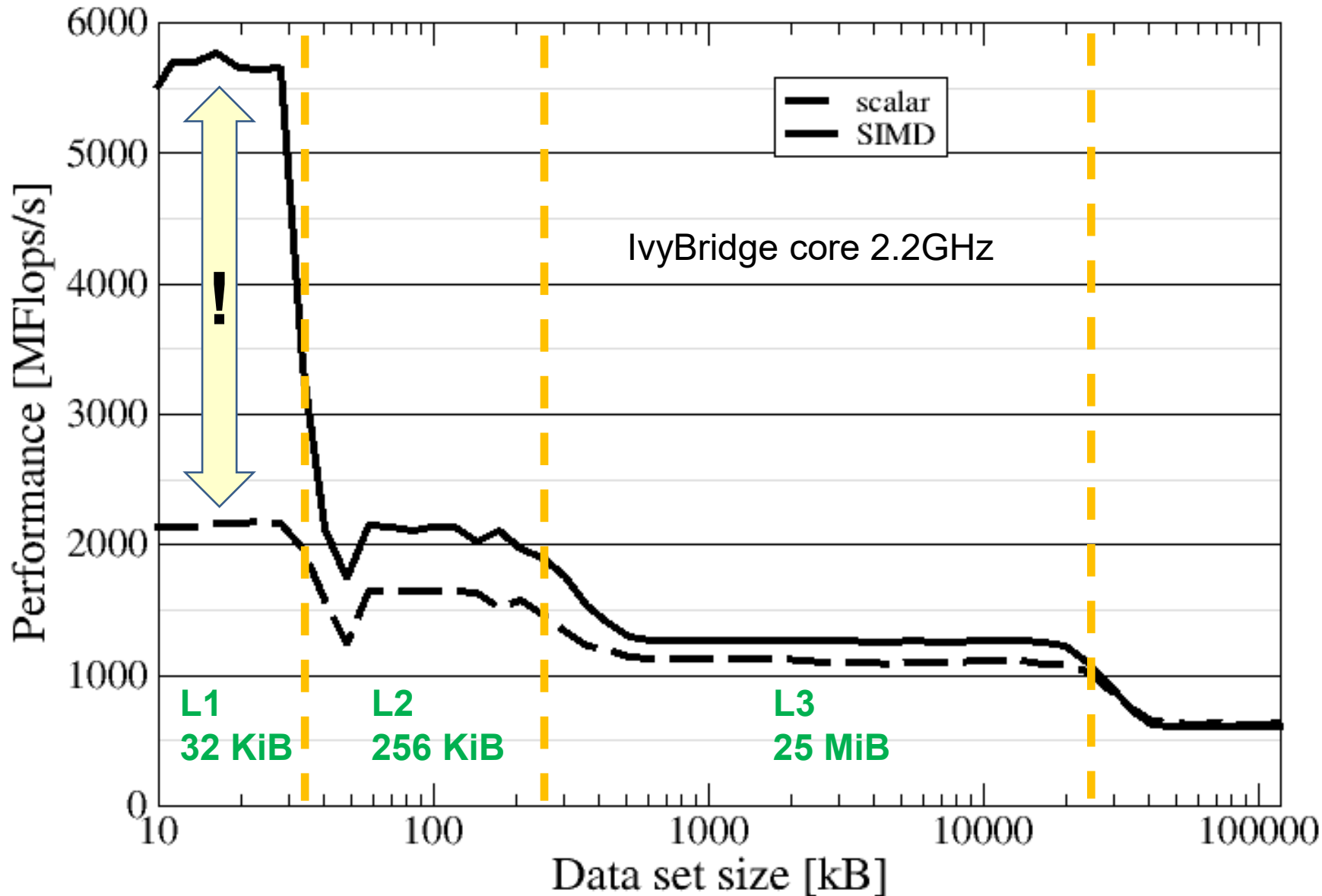


Solution 2: Cache line claim

- Special instructions (e.g., on POWER, A64FX) or automatic in hardware (Arm, Intel Ice Lake)
- Core claims CL in some level when guaranteed to be overwritten completely
- Allows stored data to remain in cache → does not reduce cache pollution



Getting the data from far away



$$A(:) = B(:) + C(:) * D(:)$$

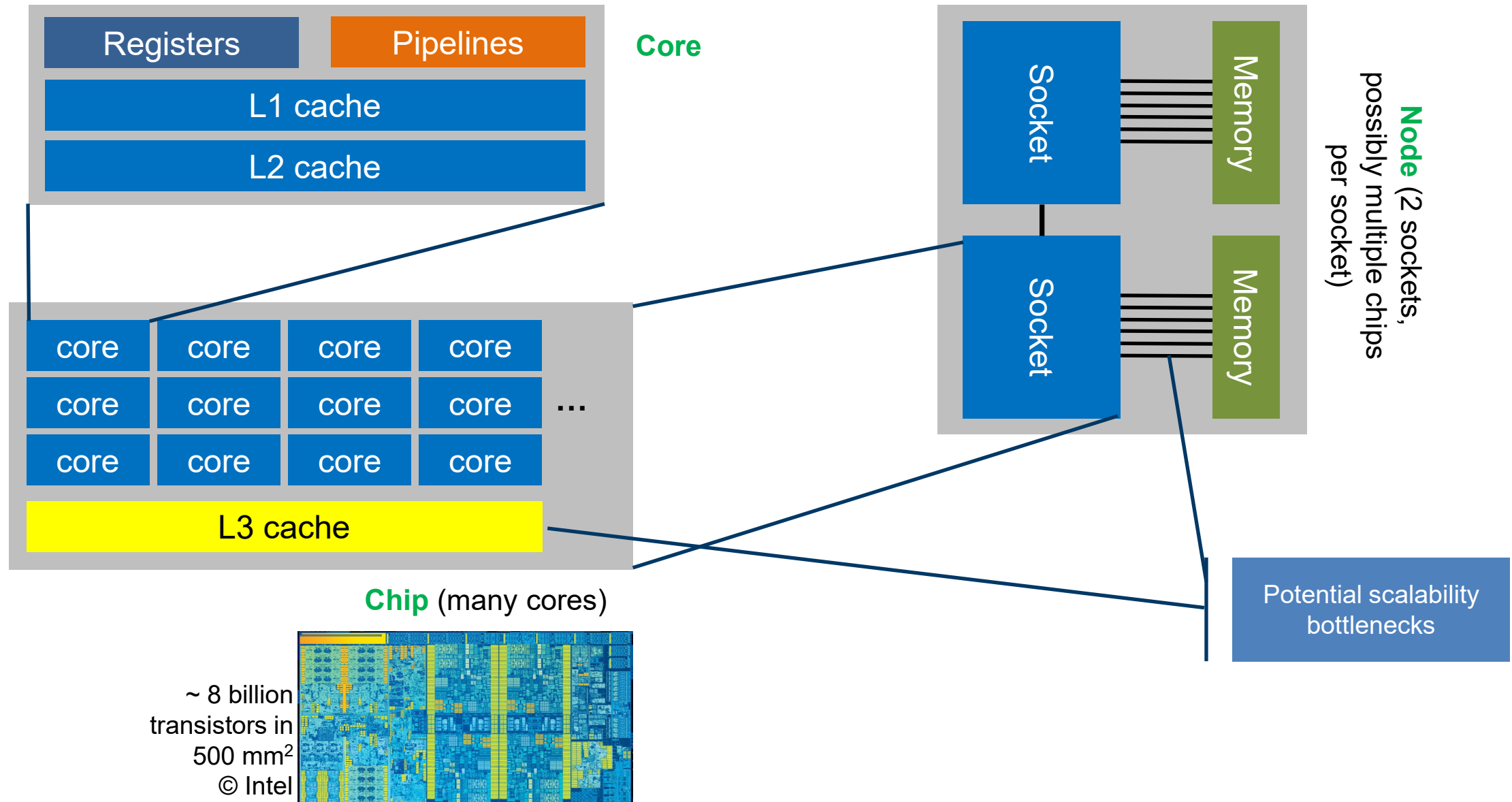
Varying loop length,
repeat many times

Multicore Chips

Memory bandwidth scaling
Node topology and performance



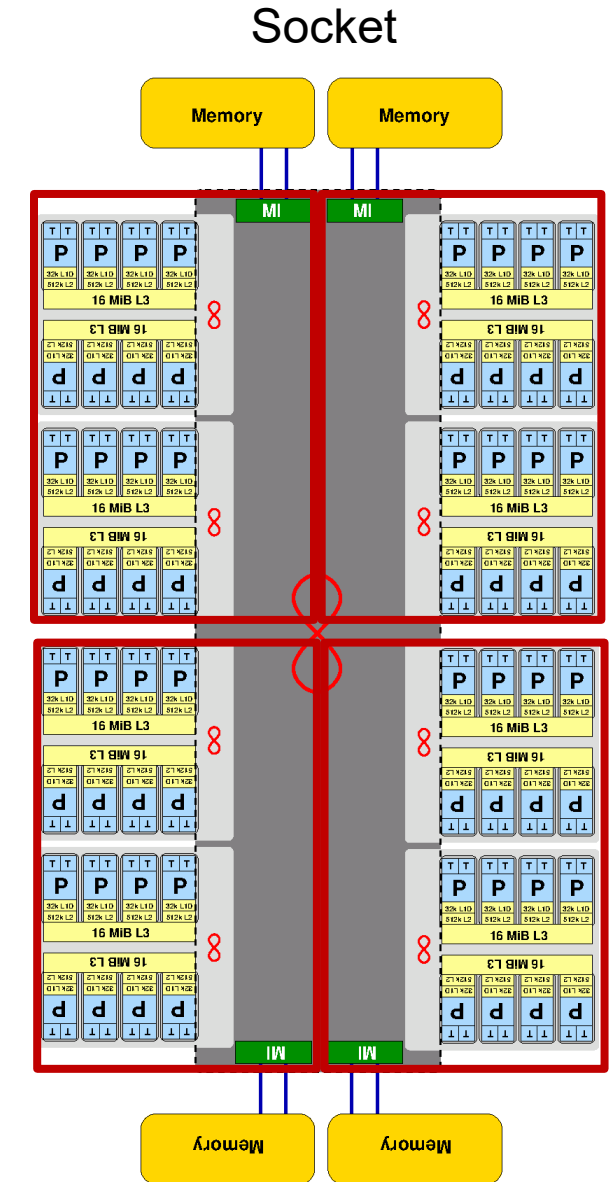
Node topology of HPC systems



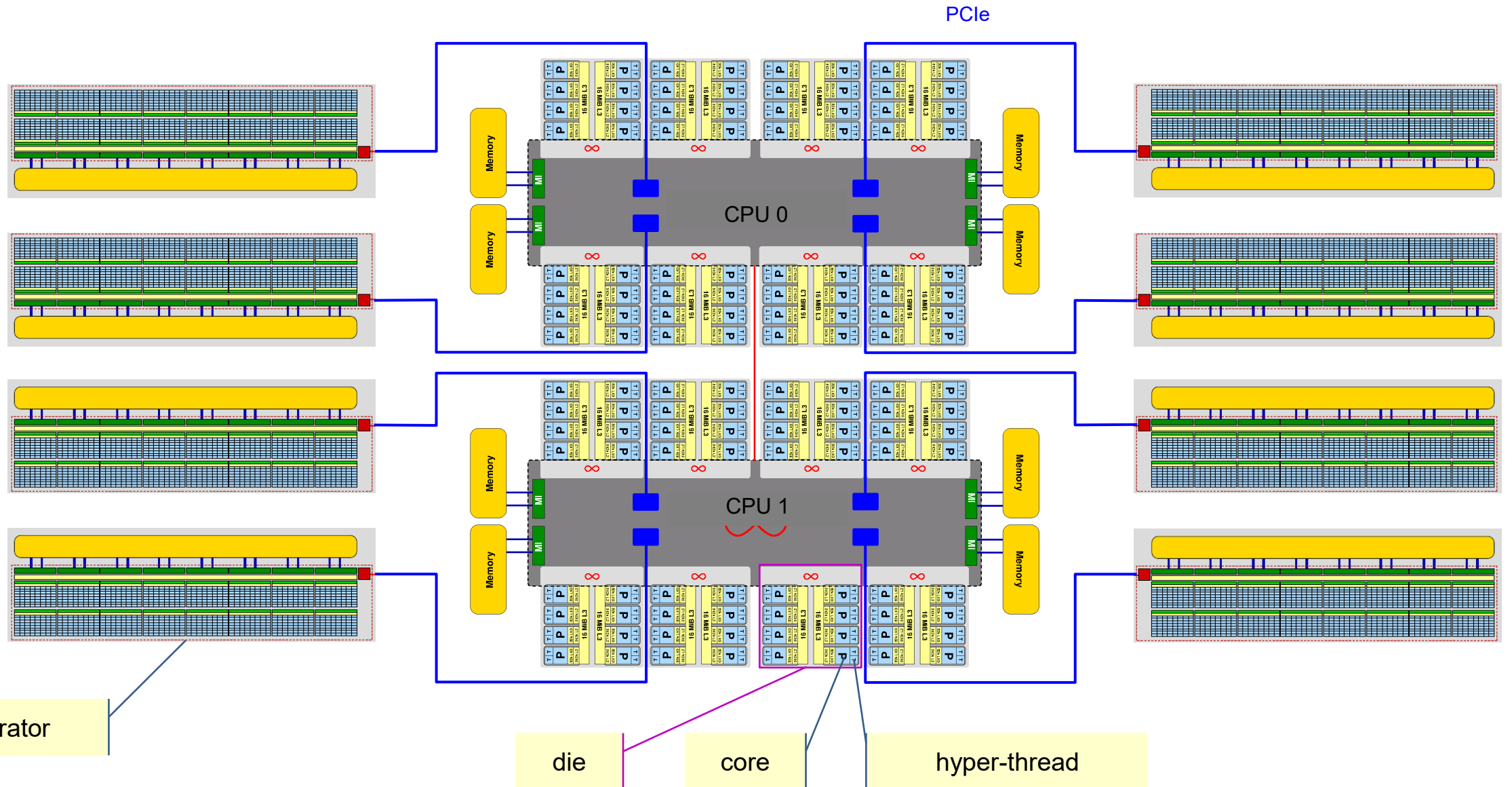
Putting the cores & caches together

AMD Epyc 7742 64-Core Processor («Rome»)

- Core features:
 - Two-way SMT
 - Two 256-bit SIMD FMA units (AVX2)
→ 16 flops/cycle
 - 32 KiB L1 data cache per core
 - 512 KiB L2 cache per core
- 64 cores per socket hierarchically built up from
 - 16 CCX with 4 cores and 16 MiB of L3 cache
 - 2 CCX form 1 CCD (silicon die)
 - 8 CCDs connected to IO device “Infinity Fabric” (memory controller & PCIe)
- 8 channels of DDR4-3200 per IO device
 - MemBW: 8 ch x 8 byte x 3.2 GHz = 204.8 GB/s
- ccNUMA feature (boot time option):
 - Nodes Per Socket (NPS)=1 , 2 or 4
 - NPS=4 → 4 ccNUMA domains**



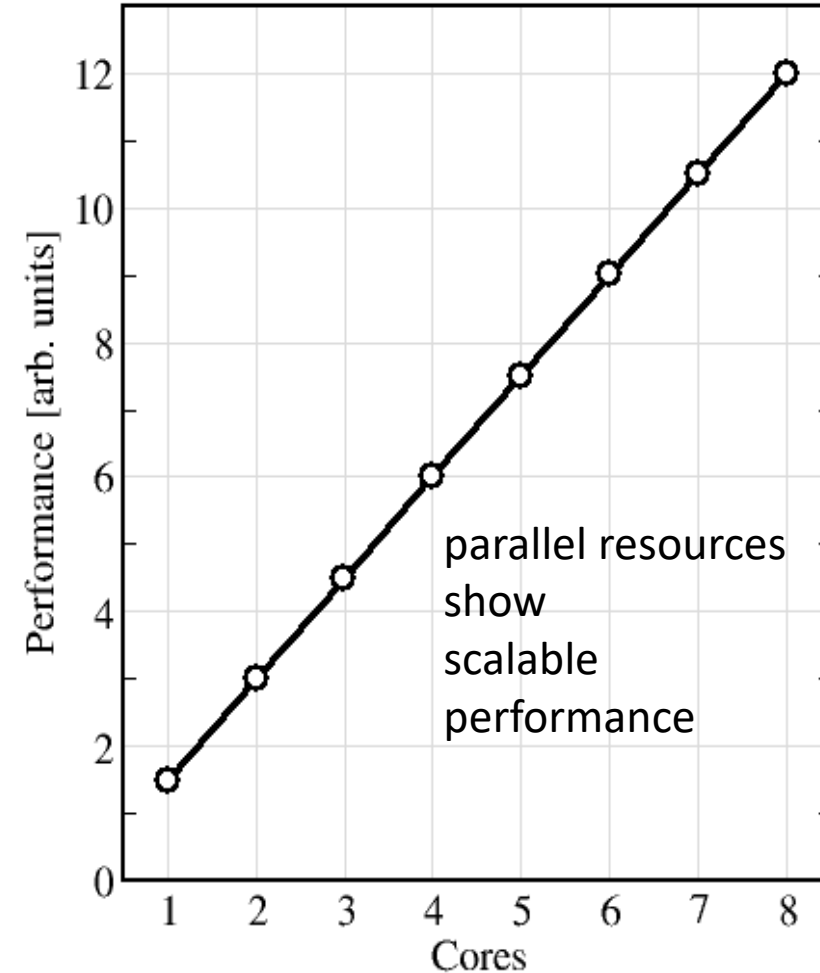
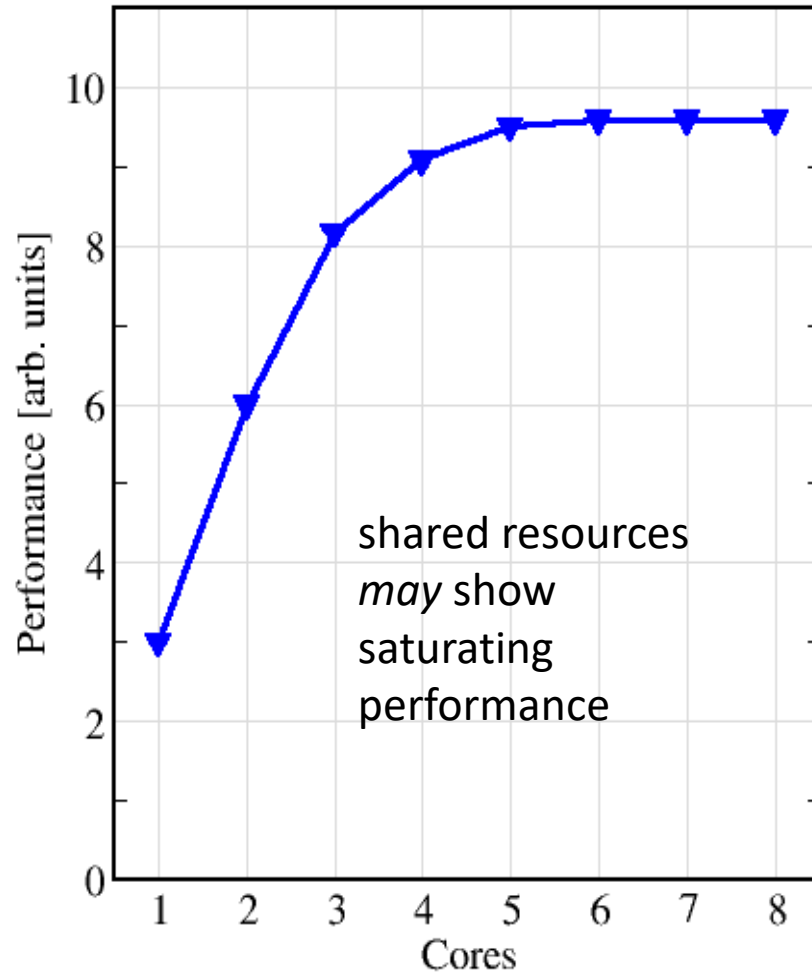
Adding accelerators to the node



Scalable and saturating behavior

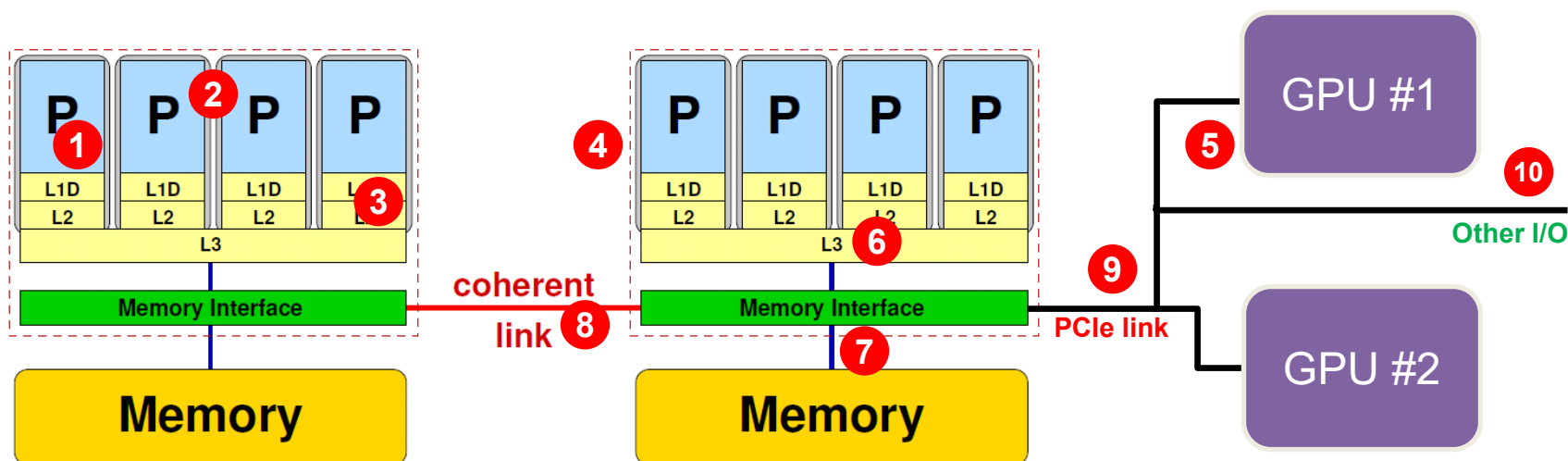
Clearly distinguish between “**saturating**” and “**scalable**” performance on the chip level

One of the most important performance signatures



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?

Interlude: A glance at accelerator technology

NVIDIA “Hopper” H100

vs.

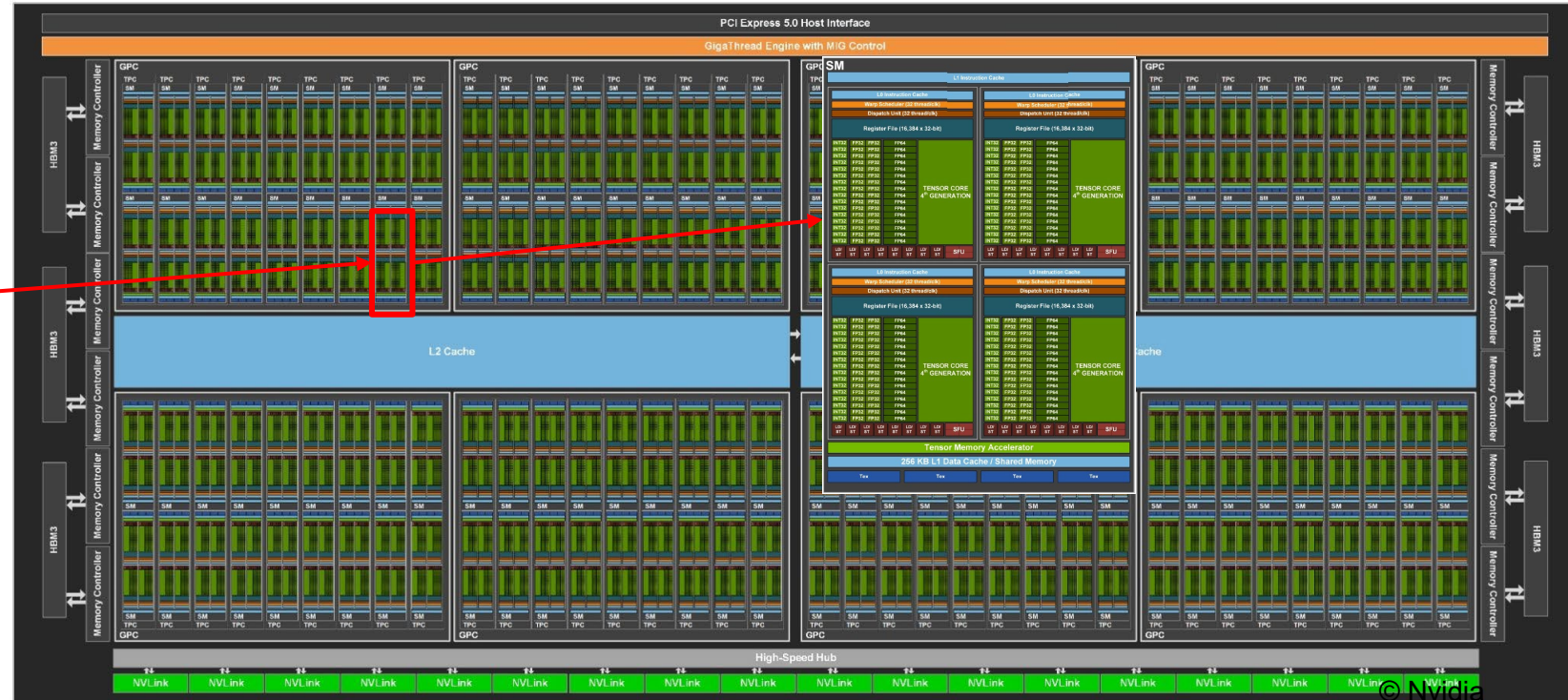
AMD Zen4 “Genoa”



Nvidia H100 “Hopper” SXM5 (700 W) specs

Architecture

- 80 B Transistors
- ~ 2.0 GHz clock speed (turbo)
- ~ 132 “SM” units
 - 128 SP “cores” each (FMA)
 - 64 DP “cores” each (FMA)
 - 4 “Tensor Cores” each
 - 2:1 SP:DP performance
- ~ 34 TFlop/s DP peak (FP64 - turbo)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)



$$P_{peak}^{DP} = n_{SM} \cdot n_{core} \cdot n_{FP} \cdot f$$

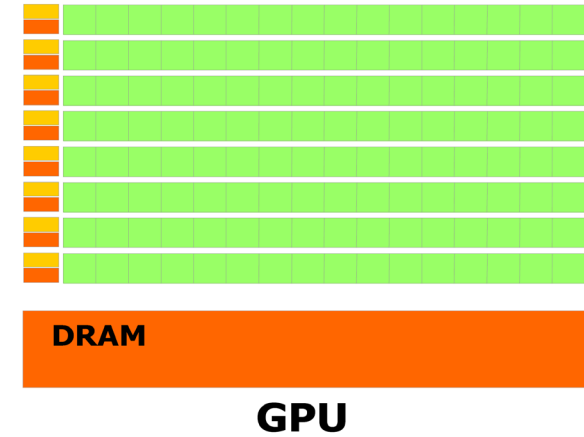
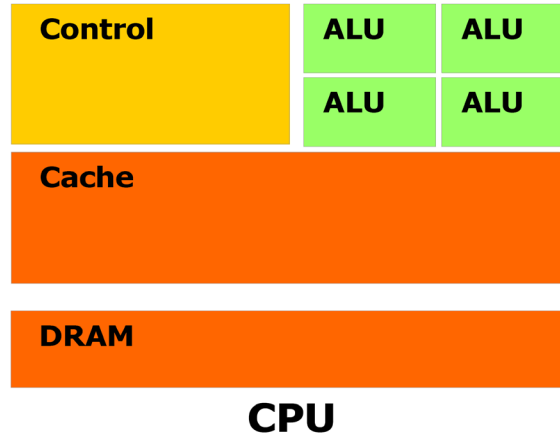
SMs # CUDA cores/SM # FP ops/cy

$$\begin{aligned} n_{SM} &= 132 \\ n_{core} &= 64 \\ n_{FP} &= 2 \frac{\text{flops}}{\text{cy}} \\ f &= 2.0 \frac{\text{Gcy}}{\text{s}} \end{aligned}$$

Trading single thread performance for parallelism: GPGPUs vs. CPUs

GPU vs. CPU

light speed estimate
(per processor chip)



	CPU	GPU
	2 x AMD EPYC 9654 "Genoa"	NVidia Tesla H100 SXM "Hopper"
Cores@Clock	2 x 96 @ 2.4 GHz	132 SMs @ ~2.0 GHz
FP32 Performance/core	76.8 GFlop/s	~ 256 GFlop/s
Threads@STREAM	~ 24	~ 100000
FP32 peak	14.7 TFlop/s	~ 67 TFlop/s
Stream BW (meas.)	2 x 360 GB/s	~ 3000 GB/s
Transistors / TDP	~ 2x 80 (?) Billion / 2x 360 W	80 Billion/700 W

Node topology and programming models



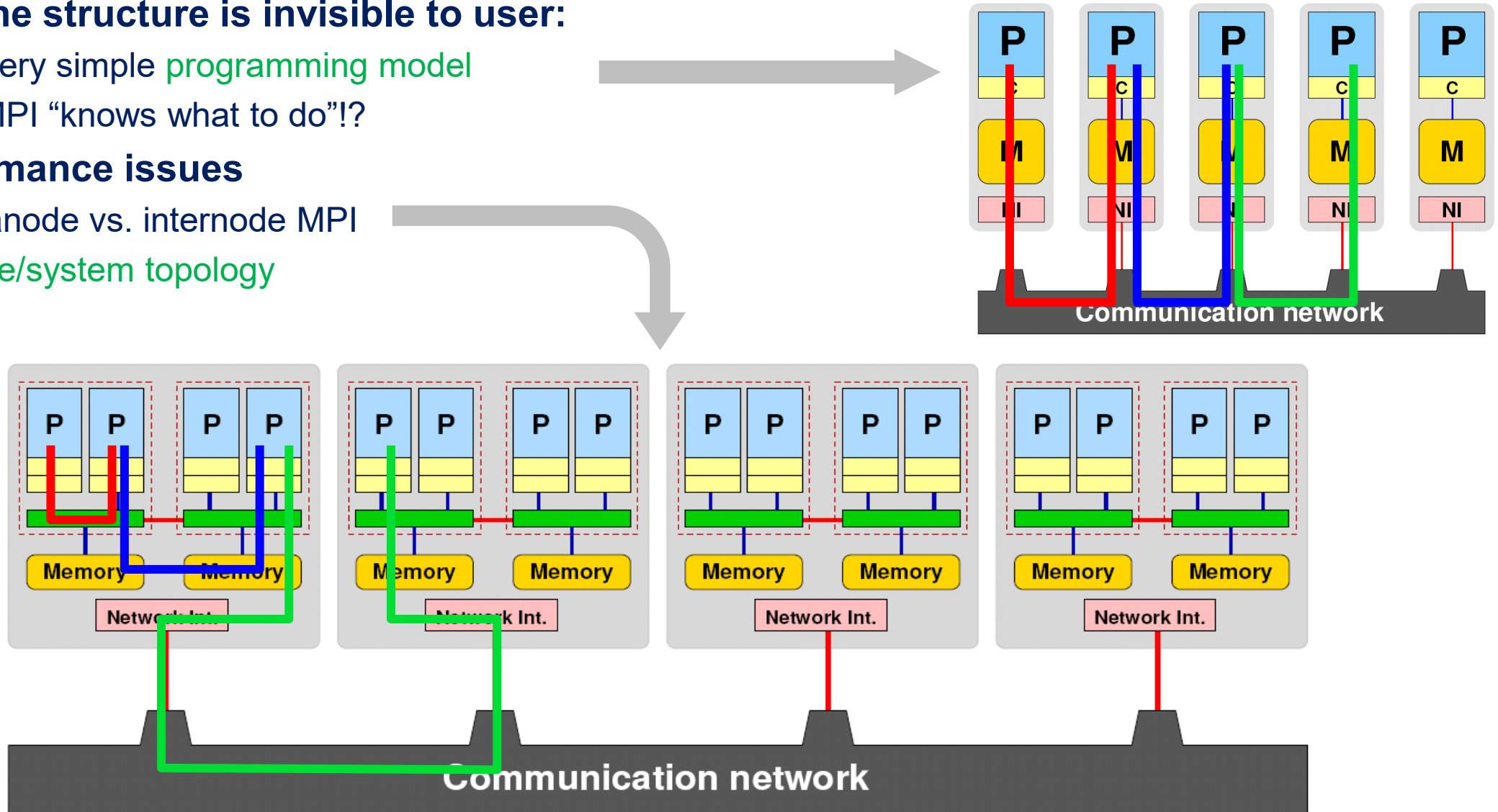
Parallel programming models: Pure MPI

- **Machine structure is invisible to user:**

- → Very simple programming model
- → MPI “knows what to do”!?

- **Performance issues**

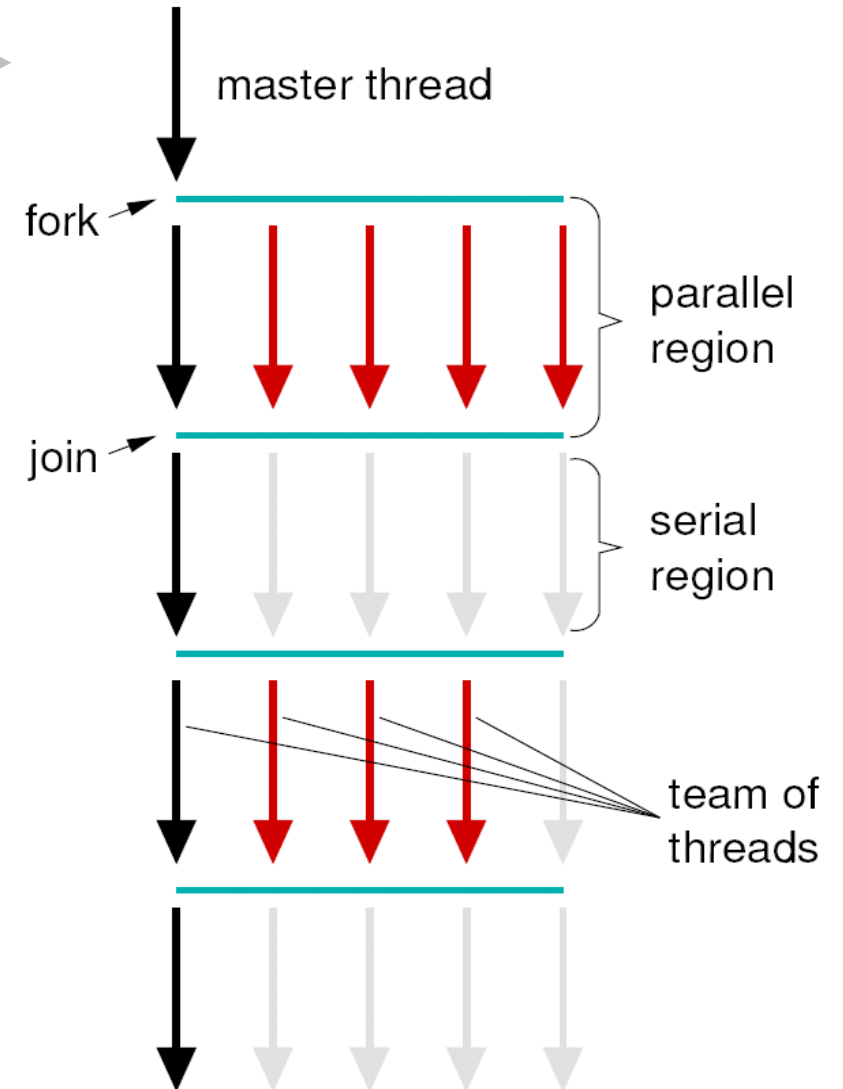
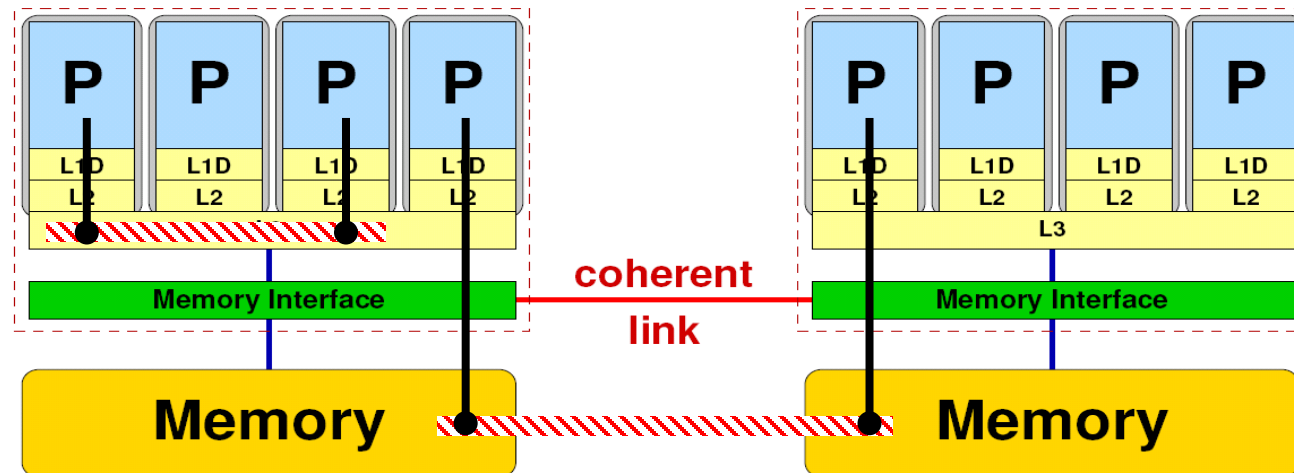
- Intranode vs. internode MPI
- Node/system topology



Parallel programming models: Pure threading

- Machine structure is invisible to user

- Very simple programming model
- Threading SW (OpenMP, pthreads, TBB,...) “should” know about the details
- OpenMP 4++: some support
- Performance issues
- Synchronization overhead
- Memory access
- Node topology



Conclusions about architecture

- Performance is a result of
 - How **many instructions** you require to implement an algorithm
 - How **efficiently** those instructions are **executed** on a processor
 - Runtime contribution of the triggered **data transfers**
- Modern computer architecture has a **rich “topology”**
- Node-level **hardware parallelism** takes many forms
 - Sockets/devices – CPU: 1-4 or more, GPGPU: 1-8
 - Cores – moderate (CPU: 20-128, GPGPU: 10-100)
 - SIMD – moderate (CPU: 2-16) to massive (GPGPU: 10’s-100’s)
 - Superscalarity (CPU: 2-6)
- **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