Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Part I: Introduction to compute node architecture

General Concept

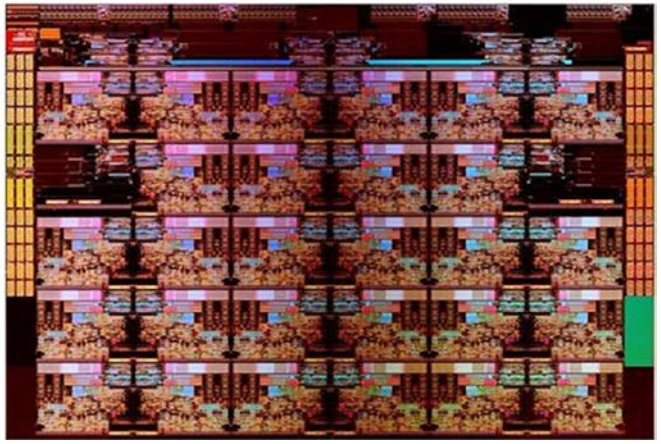
Single Core: Pipelining, Superscalarity, SMT, SIMD

Memory hierarchy and Data Transfers

GPU vs. CPU

Multi-core today: Intel Xeon Skylake SP (2017)

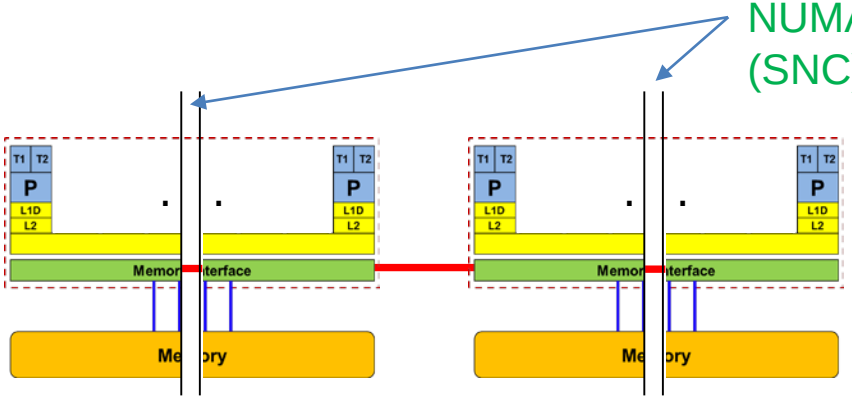
- Xeon "Skylake SP" (Platinum/Gold/Silver/Bronze):
  Up to 28 cores running at 2+ GHz (+ "Turbo Mode": 3.8+ GHz)
  - Reincarnated as "Cascade Lake" in 2018



WikiChip

- Simultaneous Multithreading (SMT)
  → reports as 56-way chip

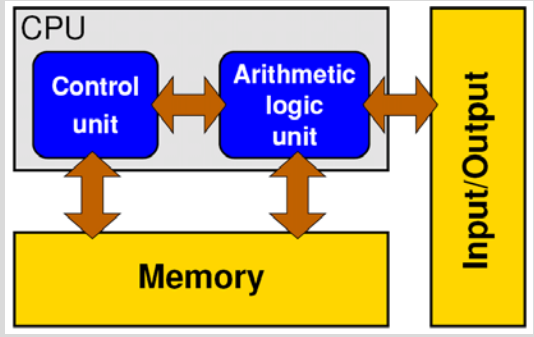
- 8 Billion Transistors / 14 nm
- Die size: ~500 mm^2



2-socket server

2015: Broadwell architecture
- Cluster on Die
  (analogous to SNC)
- Up to 24 cores

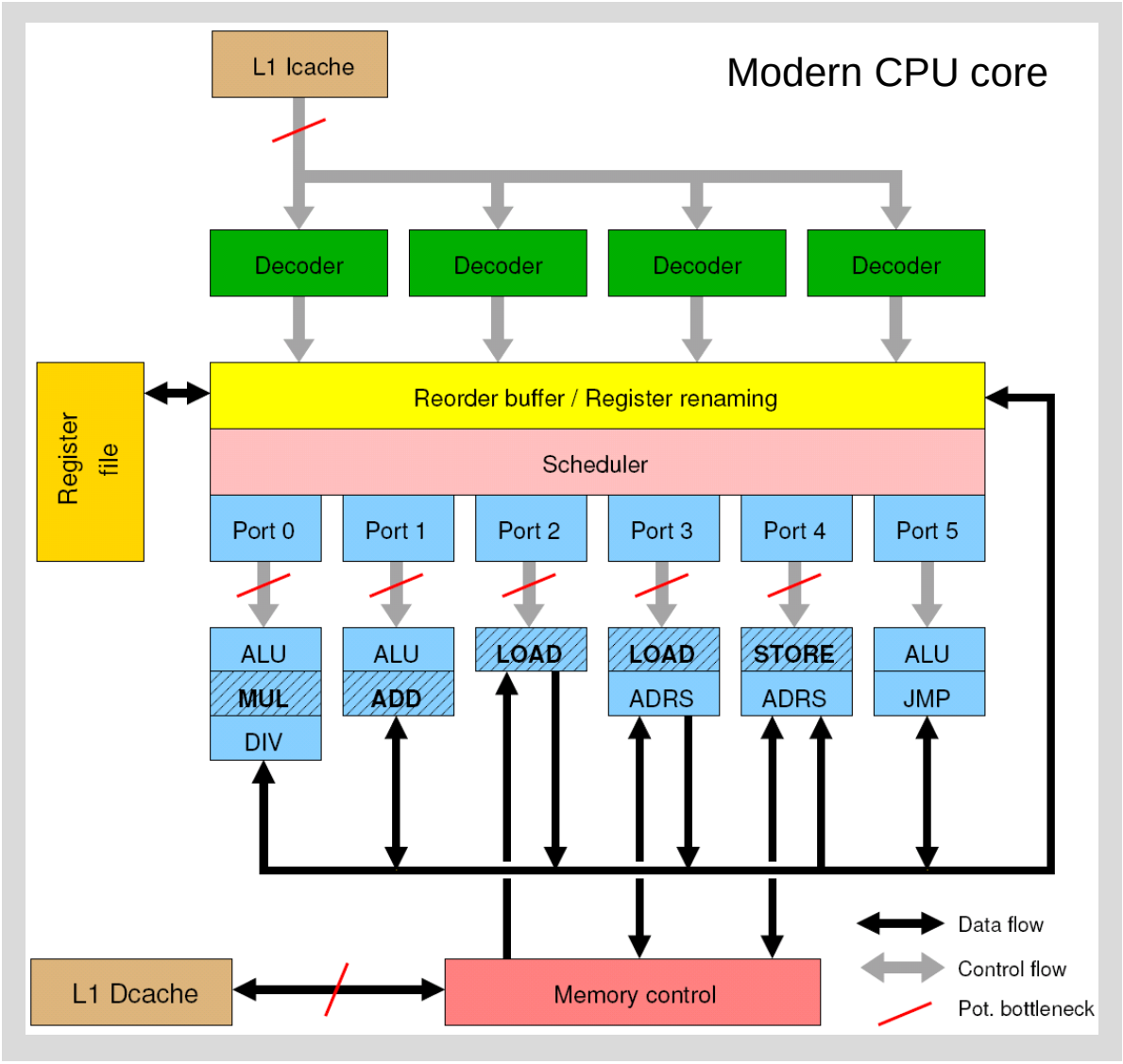
General-purpose cache based microprocessor core



Stored-program computer

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

The clock cycle is the “heartbeat” of the core



Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Part I: Introduction to compute node architecture

Single core: Pipelining, Superscalarity, SMT, SIMD

High Performance Computing

Pipelining of arithmetic/functional units

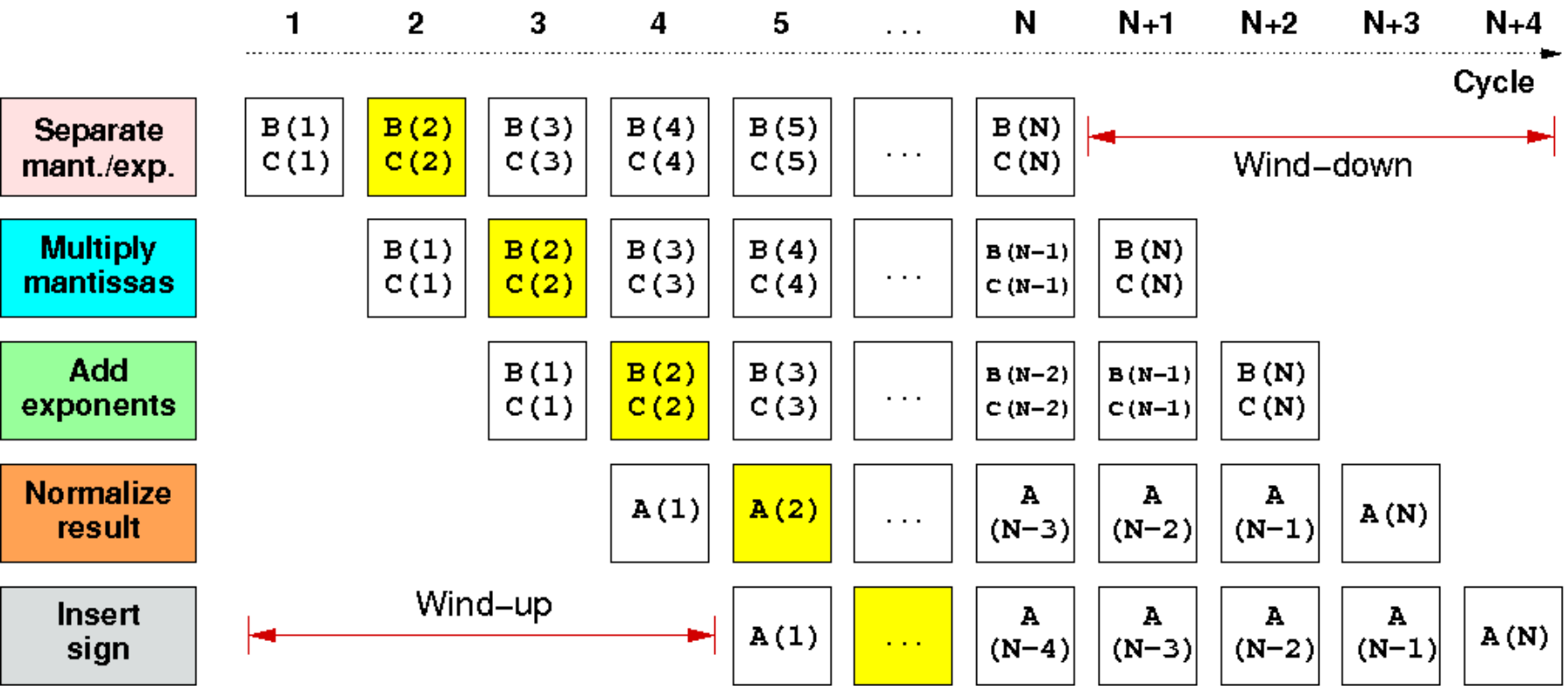
- Idea:
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same time, e.g., one cycle
  - Execute different steps on different instructions at the same time (in parallel)

- 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 – sufficient # of independent instructions required
  - Requires complex instruction scheduling by compiler/hardware
    - software-pipelining / out-of-order execution

- Pipelining is widely used in modern computer architectures

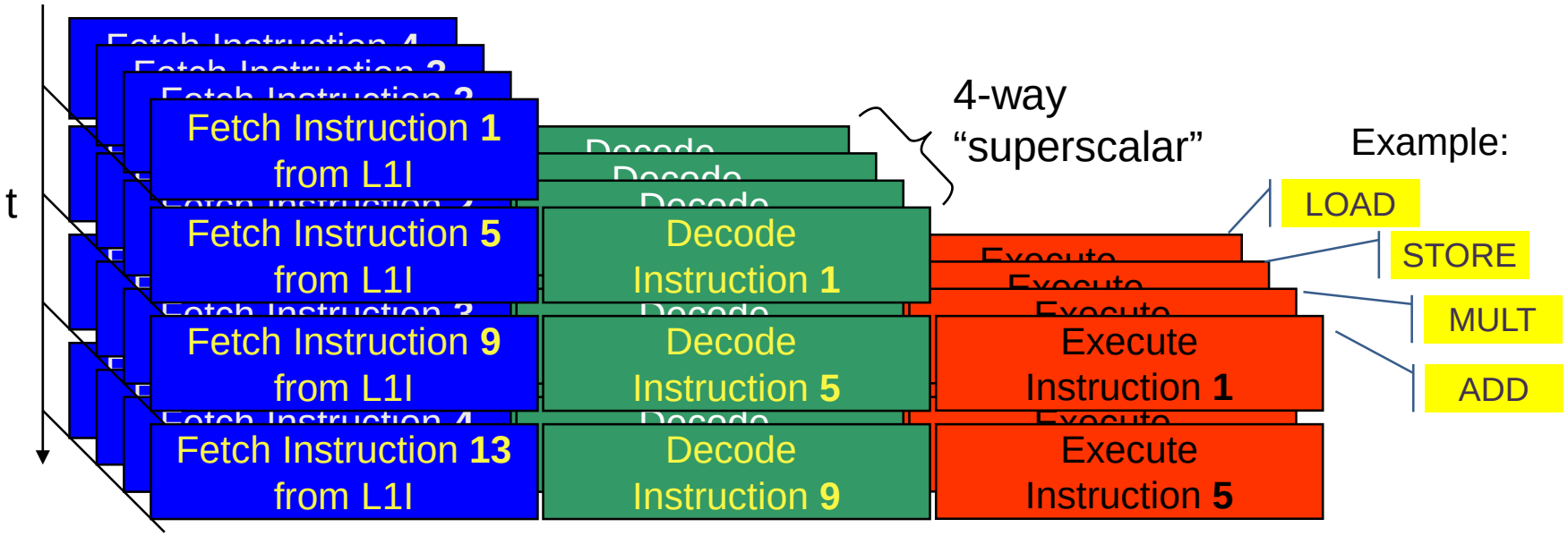
5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N



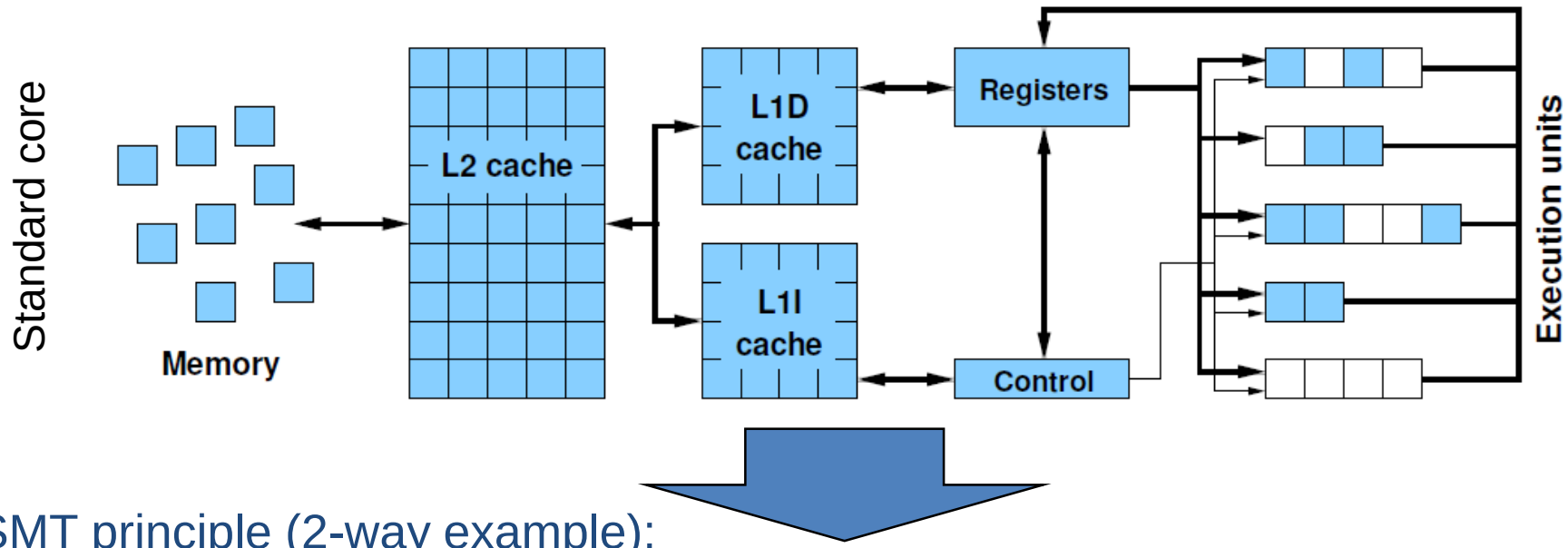
First result is available after 5 cycles (=latency of pipeline)!

Wind-up/-down phases: Empty pipeline stages

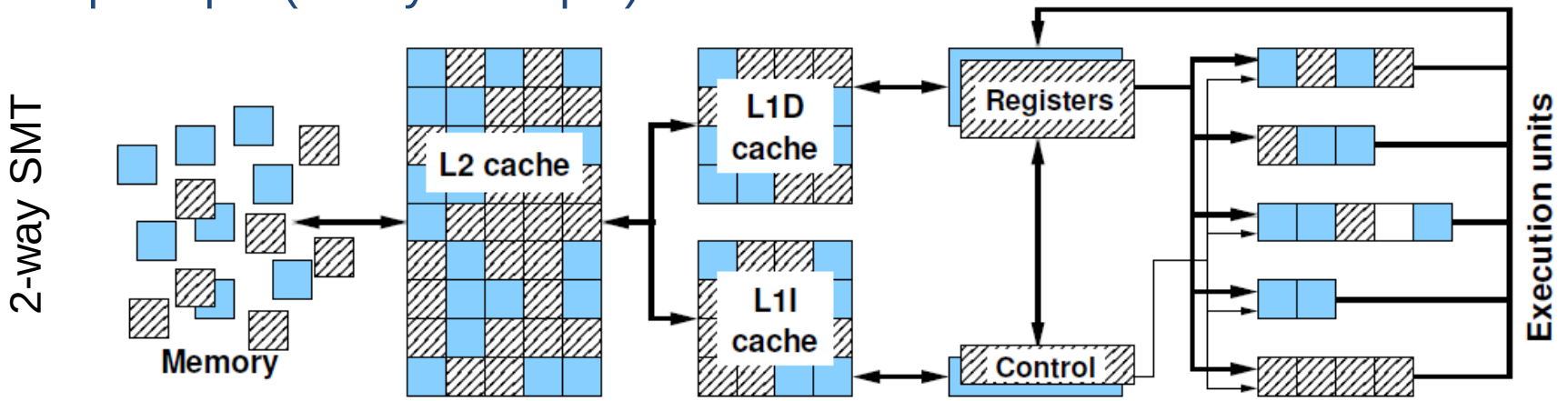
- Multiple units enable use of Instruction Level Parallelism (ILP): Instruction stream is “parallelized” on the fly
- Instructions from different loop iterations retired at the same time



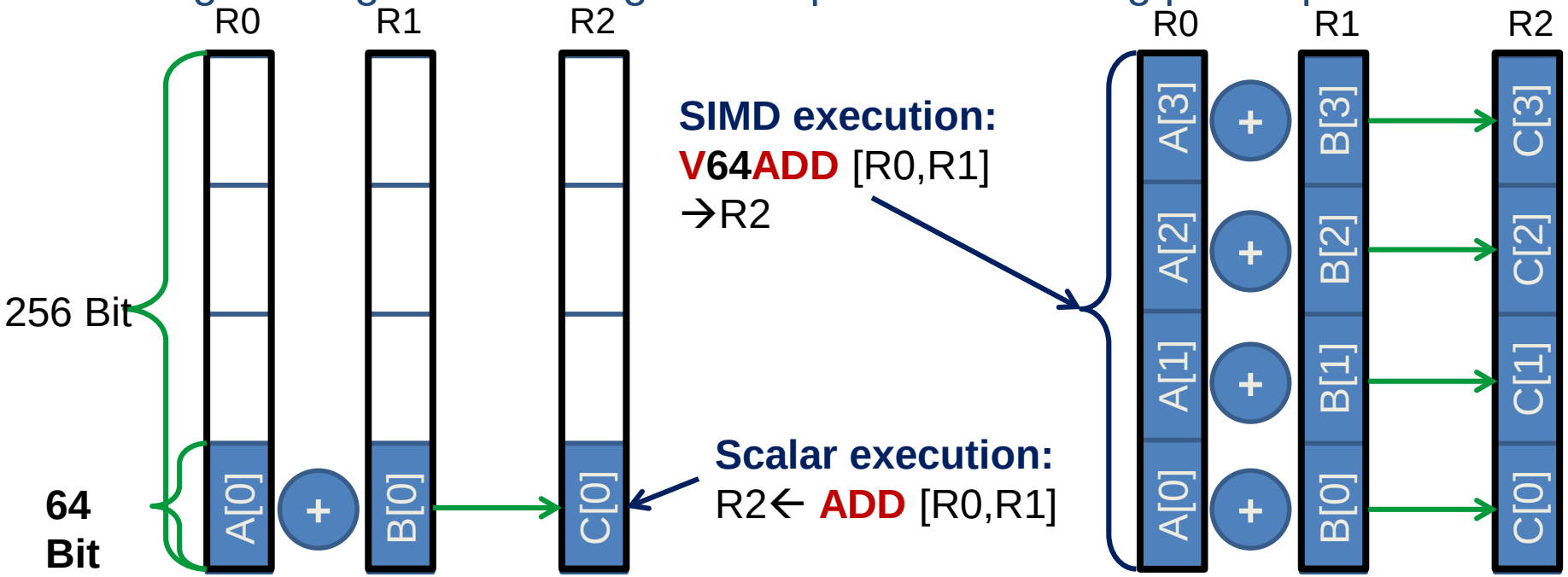
- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 4- to 6-way superscalar & can perform 2 floating-point instructions per cycle



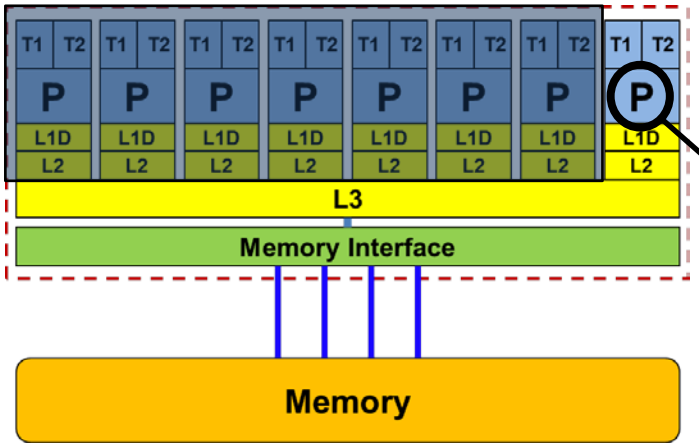
SMT principle (2-way example):



- 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(/2): register width = 256 Bit → 4 double precision floating point operands
  - AXV512: you get it.
- Adding two registers holding double precision floating point operands



There is no single driving force for single core performance!



Maximum floating point (FP) 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.]		Code	f [Gcy/s]	P_{core} [GF/s]
Nehalem	2	1	2	Q1/2009	X5570	2.93	11.7
Sandy Bridge	2	1	4	Q1/2012	E5-2680	2.7	21.6
Haswell	2	2	4	Q3/2014	E5-2695 v3	2.3	36.8
Broadwell	2	2	4	Q1/2016	E5-2699 v4	2.2	35.2
Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
AMD Zen	2	2	2	Q1/2017	Epyc 7451	2.3	18.4
AMD Zen2	2	2	4	Q3/2019	Epyc 7742	2.25	36.0
Fujitsu A64FX	2	2	8	Q2/2020	FX700	1.8	57.6
IBM POWER8	2	2	2	Q2/2014	S822LC	2.93	23.4

A “simple” example: The sum reduction

```
s = 0.0
do i = 1,N
  s = s + a(i)
enddo
```

…In single precision (32 Bit) 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

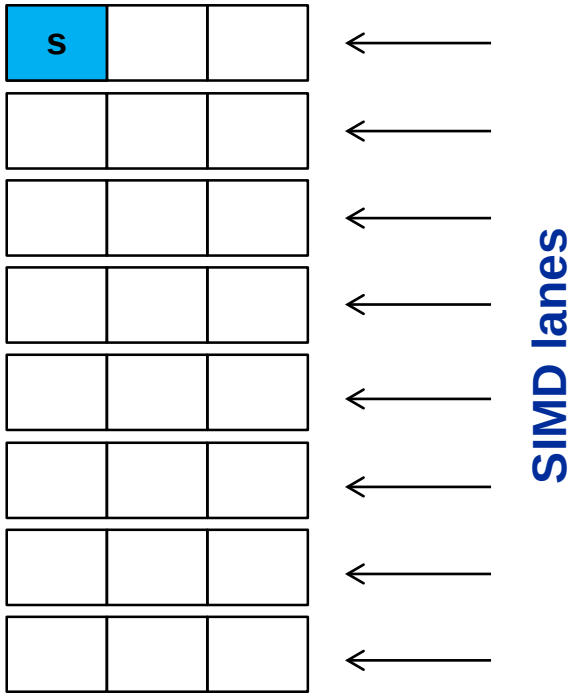
→ No pipelining?
→ No SIMD?

Plain scalar code, no SIMD

```
do i = 1,N
  s = s + a(i)
enddo
```

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

ADD pipes utilization:



→ 1/24 of ADD peak

Applicable peak for the sum reduction (II)

Scalar code, 3-way "modulo variable expansion"

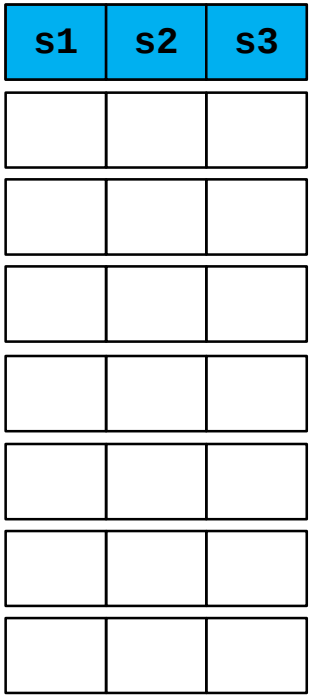
```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1

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
```

```
do i = 1,N,3
  s1 = s1 + a(i+0)
  s2 = s2 + a(i+1)
  s3 = s3 + a(i+2)
enddo
s = s + 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)

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

```
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
```

→ ADD peak

s10	s20	s30
s11	s21	s31
s12	s22	s32
s13	s23	s33
s14	s24	s34
s15	s25	s35
s16	s26	s36
s17	s27	s37

Sum reduction

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

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 ;-)

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Part I: Introduction to compute node architecture

Memory Hierarchy and Data Transfers

High Performance Computing

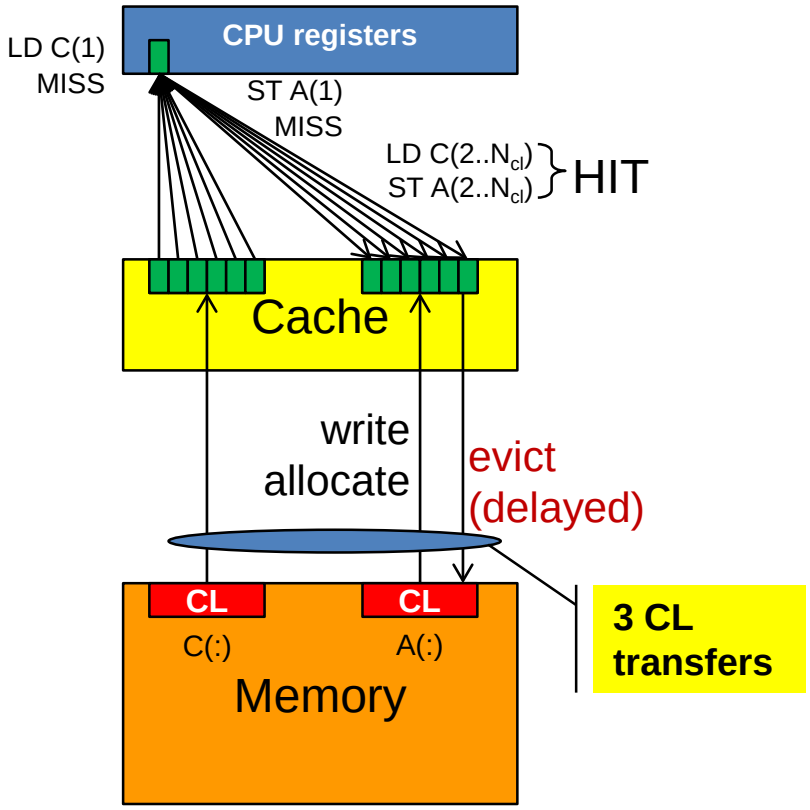
How does data travel from memory to the CPU and back?

- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except L1 ←→registers)

- Cache MISS: Load or store instruction does not find the data in a cache level
  → CL transfer required

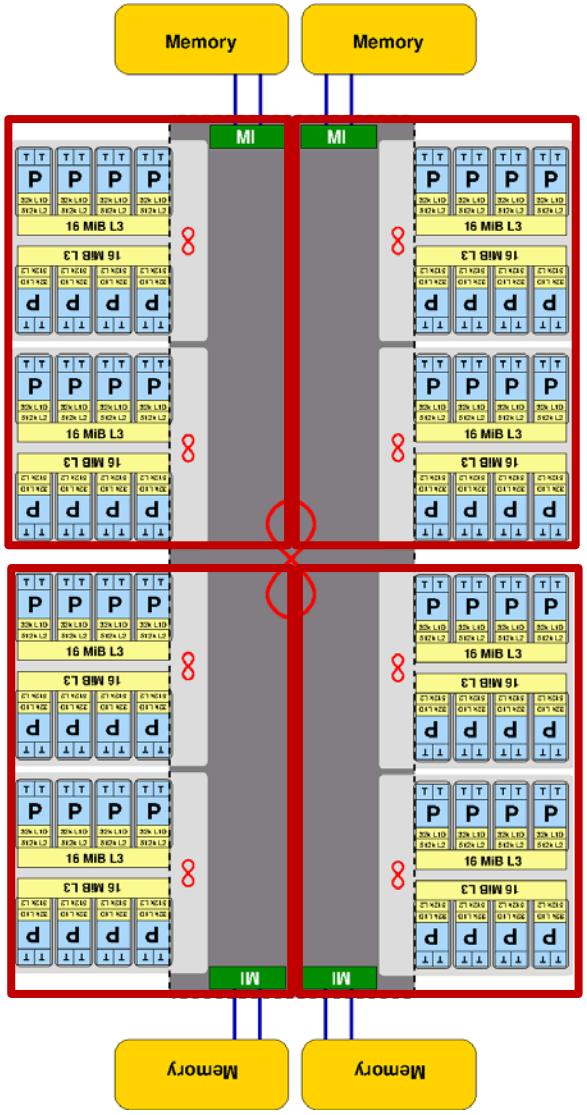
- Example: Array copy `A(:)=C(:)`

- "Store miss" → write allocate (WA)
- Techniques exists to avoid WA



Putting the cores & caches together
AMD Epyc 7742 64-Core Processor («Rome»)

Processor „Chip“ (Socket)



- 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 MB 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
- NUMA-feature (Boot time option):
  - Node Per Socket (NPS)=1 , 2 or 4
  - **NPS=4 → 4 UMA domains**

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Part I: Introduction to compute node architecture

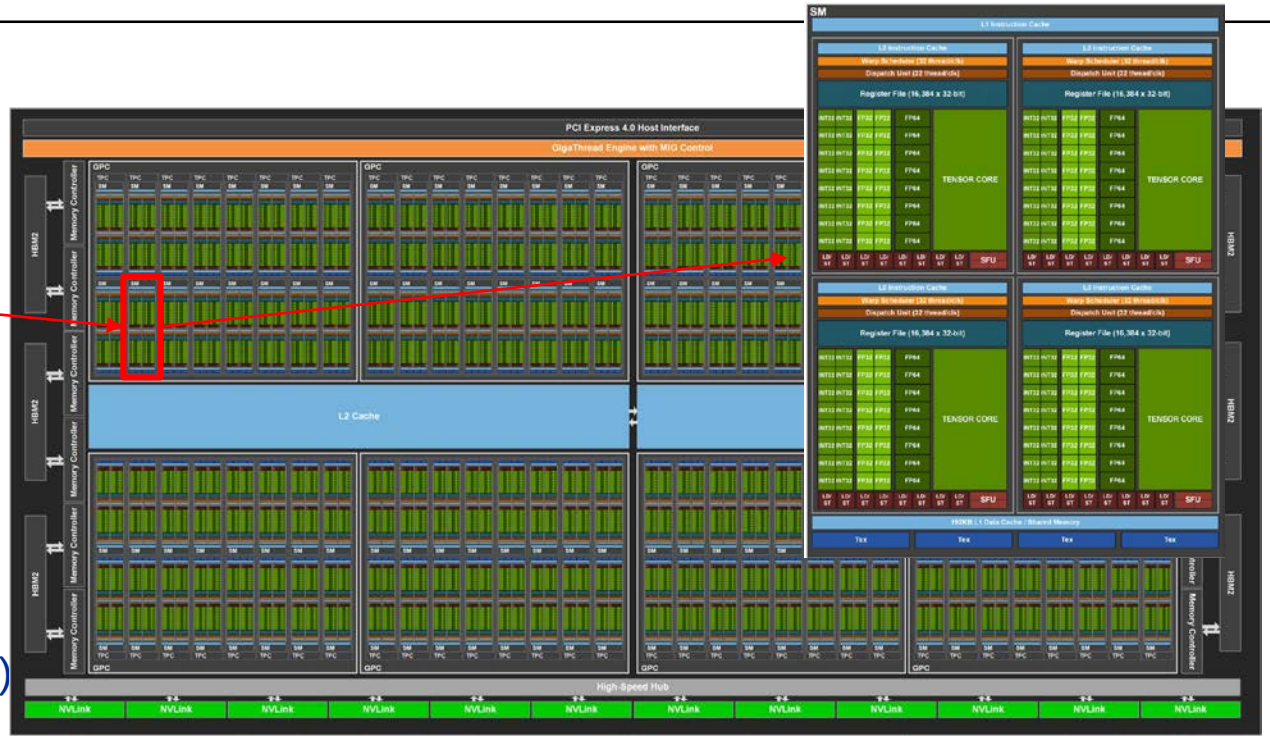
GPU vs CPU

High Performance Computing

Nvidia A100 “Ampere” SXM4 specs

Architecture

- 54.2 B Transistors
- ~ 1.4 GHz clock speed
- ~ 108 “SM” units
  - 64 SP “cores” each (FMA)
  - 32 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance
- 9.7 TFlop/s DP peak (FP64)
- 40 MiB L2 Cache
- 40 GB (5120-bit) HBM2
- MemBW ~ 1555 GB/s (theoretical)
- MemBW ~ 1400 GB/s (measured)



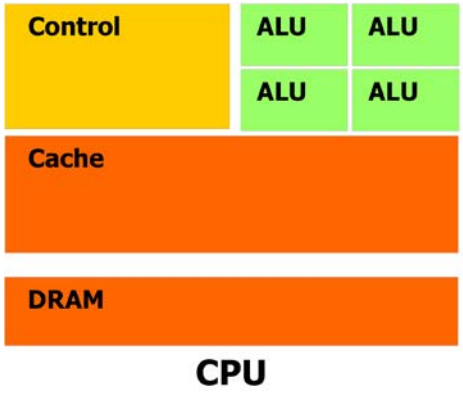
© Nvidia

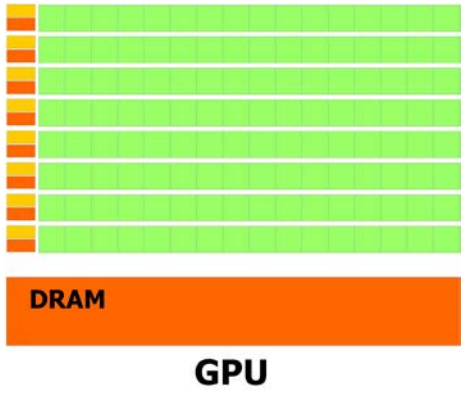
$$P^{DP}_{peak} = n_{SM} \cdot n_{core} \cdot n_{FP} \cdot f$$

# SMs	# CUDA cores/SM	# FP ops/cy

$$n_{SM} = 108$$
$$n_{core} = 32$$
$$n_{FP} = 2\frac{\text{flops}}{\text{cy}}$$
$$f = 1.4\frac{\text{Gcy}}{\text{s}}$$

GPU vs. CPU light speed estimate (per processor chip)





MemBW ~ 7 – 10x

Peak ~ 4 – 8x

	2 x AMD EPYC 7742 ”Rome”	NVidia Tesla A100 “Ampere”
Cores@Clock	2 x 64 @ 2.25 GHz	108 SMs @ ~1.4 GHz
FP32 Performance/core	72 GFlop/s	~179 GFlop/s
Threads@STREAM	~16	> 100000
FP32 peak	9.2 TFlop/s	~19.5 TFlop/s
Stream BW (meas.)	2 x 190 GB/s	1400 GB/s
Transistors / TDP	~2x40 Billion / 2x225 W	54 Billion/400 W

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: 4-64) to massive (GPGPU: 10’s-100’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**

Note:

- Performance features are largely independent of the programming model
- But programming models are sensitive to architecture
  - Topology/affinity influences overheads
  - Standards do not contain (many) topology-aware features