



SC23
Denver, CO | i am hpc.

Node-Level Performance Engineering

<https://tiny.cc/NLPE-SC23>

Georg Hager, Thomas Gruber, Gerhard Wellein
Erlangen National High Performance Computing Center (NHR@FAU)

SC23 Full-Day Tutorial
Monday, November 13, 2023



Node-Level Performance Engineering

<https://tiny.cc/NLPE-SC23>

Georg Hager, Thomas Gruber, Gerhard Wellein
Erlangen National High Performance Computing Center (NHR@FAU)

SC23 Full-Day Tutorial
Monday, November 13, 2023



Agenda

- Part I

- Introduction to compute node architecture
- Performance tools 1: topology and affinity
- Microbenchmarking as a tool
- Demo
- Introduction to the Roofline model
- Performance tools 2: hardware performance counters
- Demo

- Part II

- Case study: tall & skinny matrix-matrix multiplication
- Case study: Stencil codes
- Demo
- Case study: sparse matrix-vector multiplication
- Programming for Single Instruction Multiple Data (SIMD) parallelism
- Programming for ccNUMA

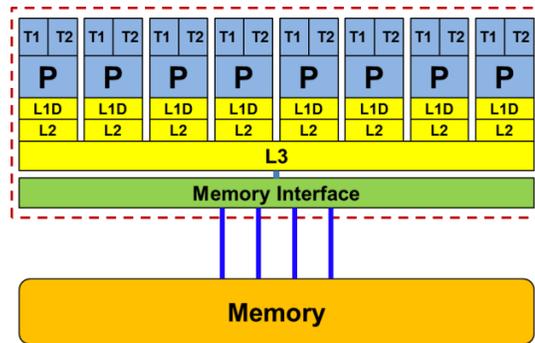
Prelude: Scalability 4 teh win!



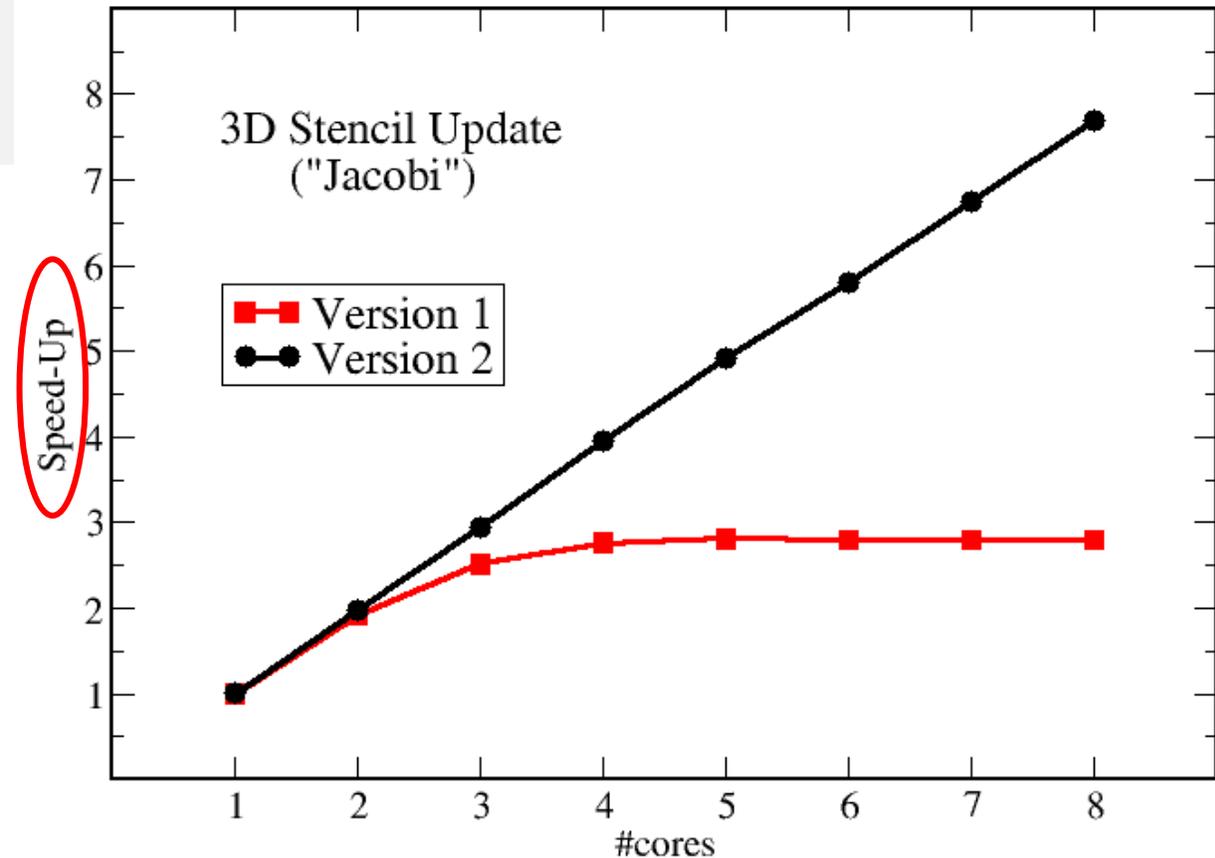
Scalability Myth: Code scalability is the key issue

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
    y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) +
                  x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Changing only the compile options makes this code scalable on an 8-core chip



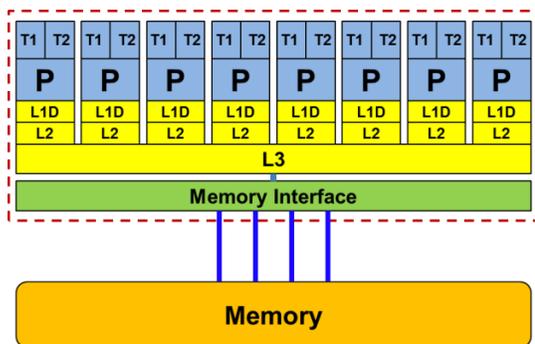
Prepared for the highly parallel era!



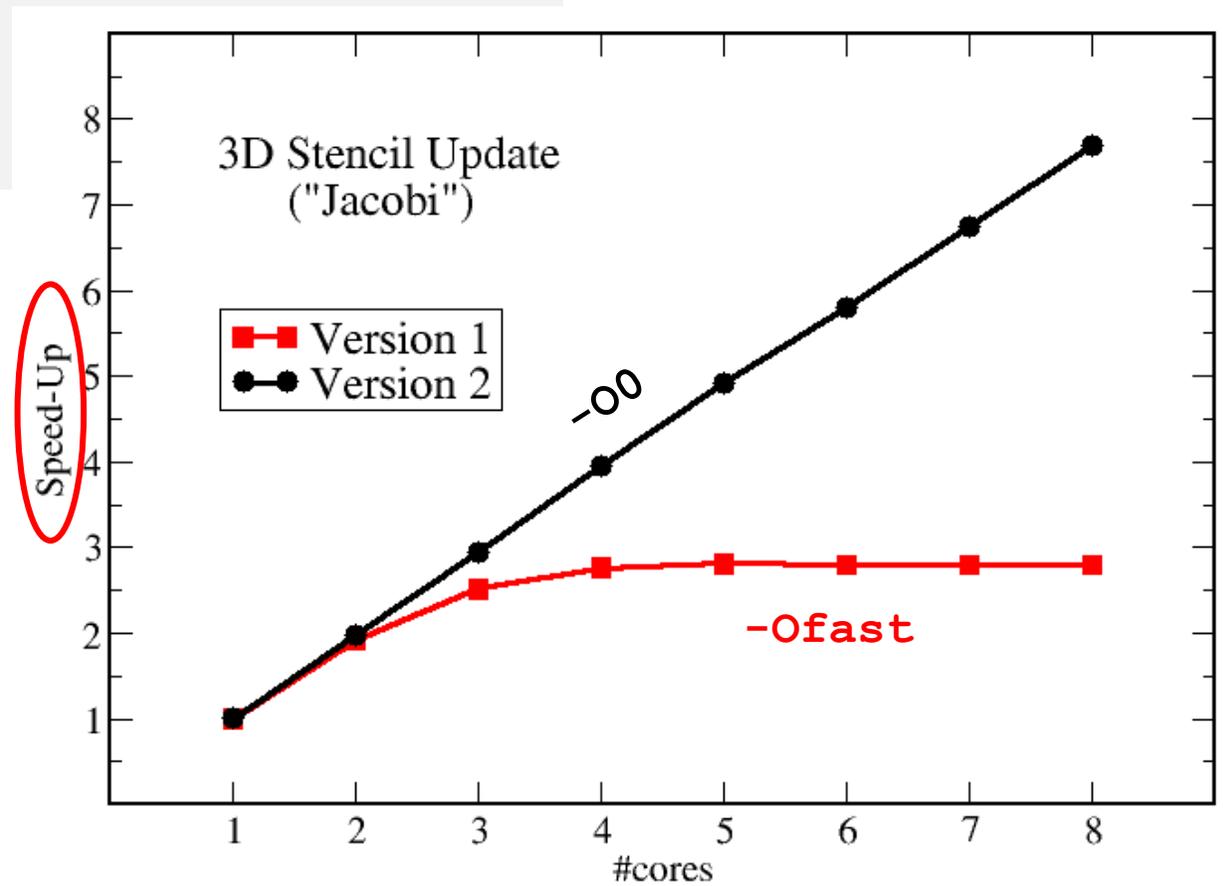
Scalability Myth: Code scalability is the key issue

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
    y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) +
                  x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Changing only the compile options makes this code scalable on an 8-core chip

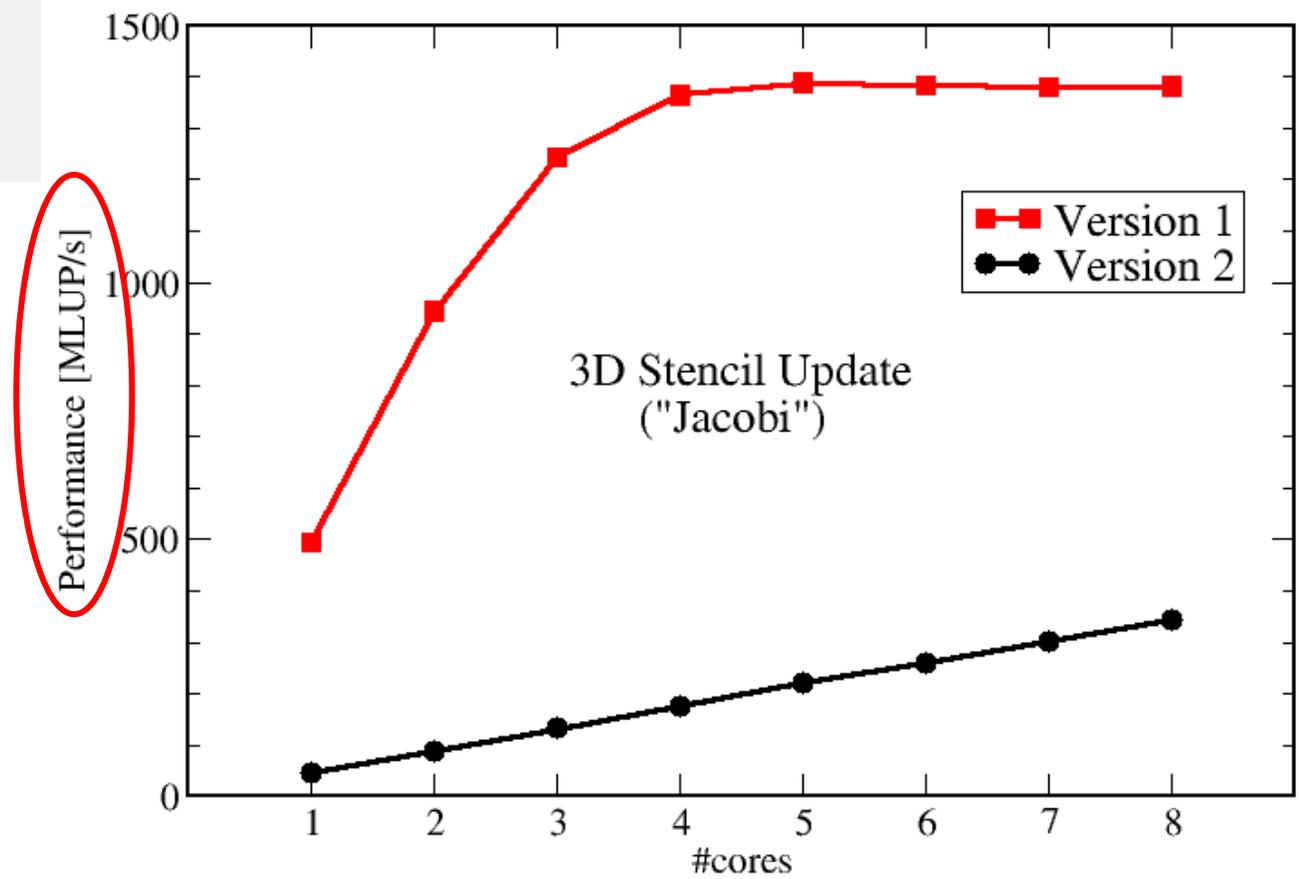
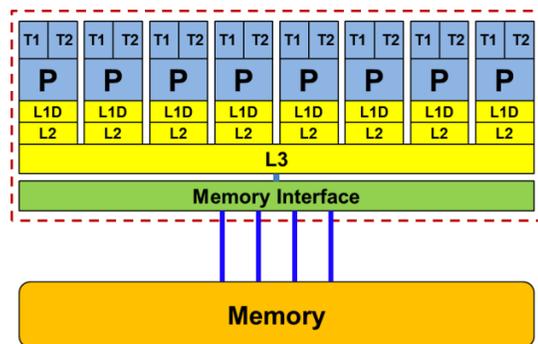


Prepared for the highly parallel era!



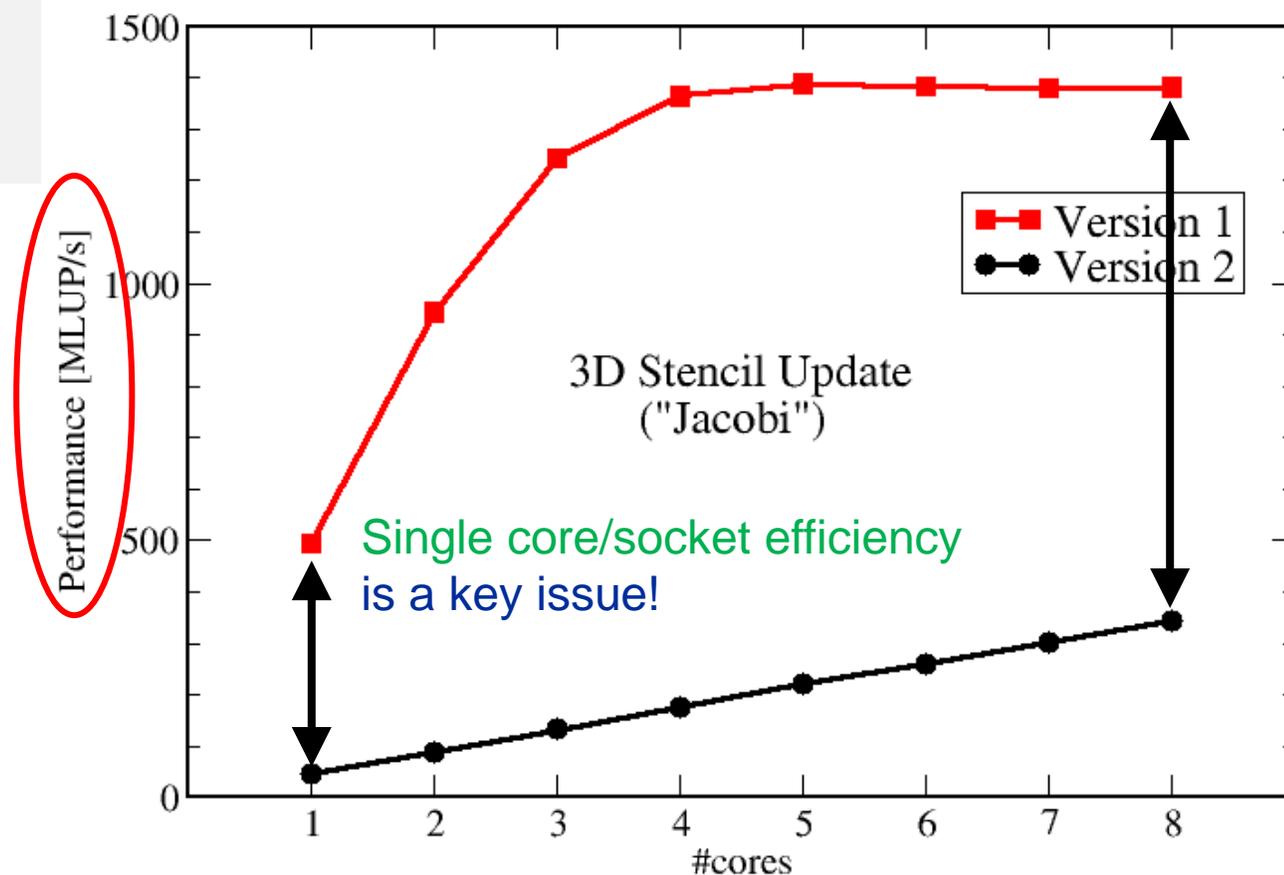
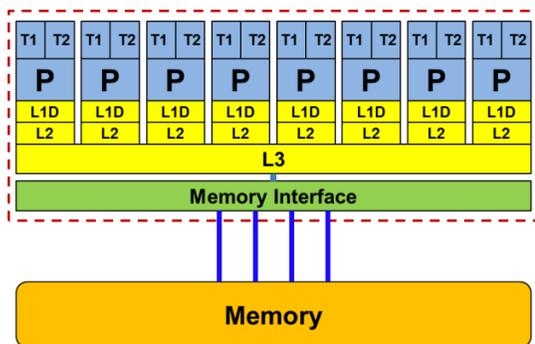
Scalability Myth: Code scalability is the key issue

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
    y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) +
                  x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
```



Scalability Myth: Code scalability is the key issue

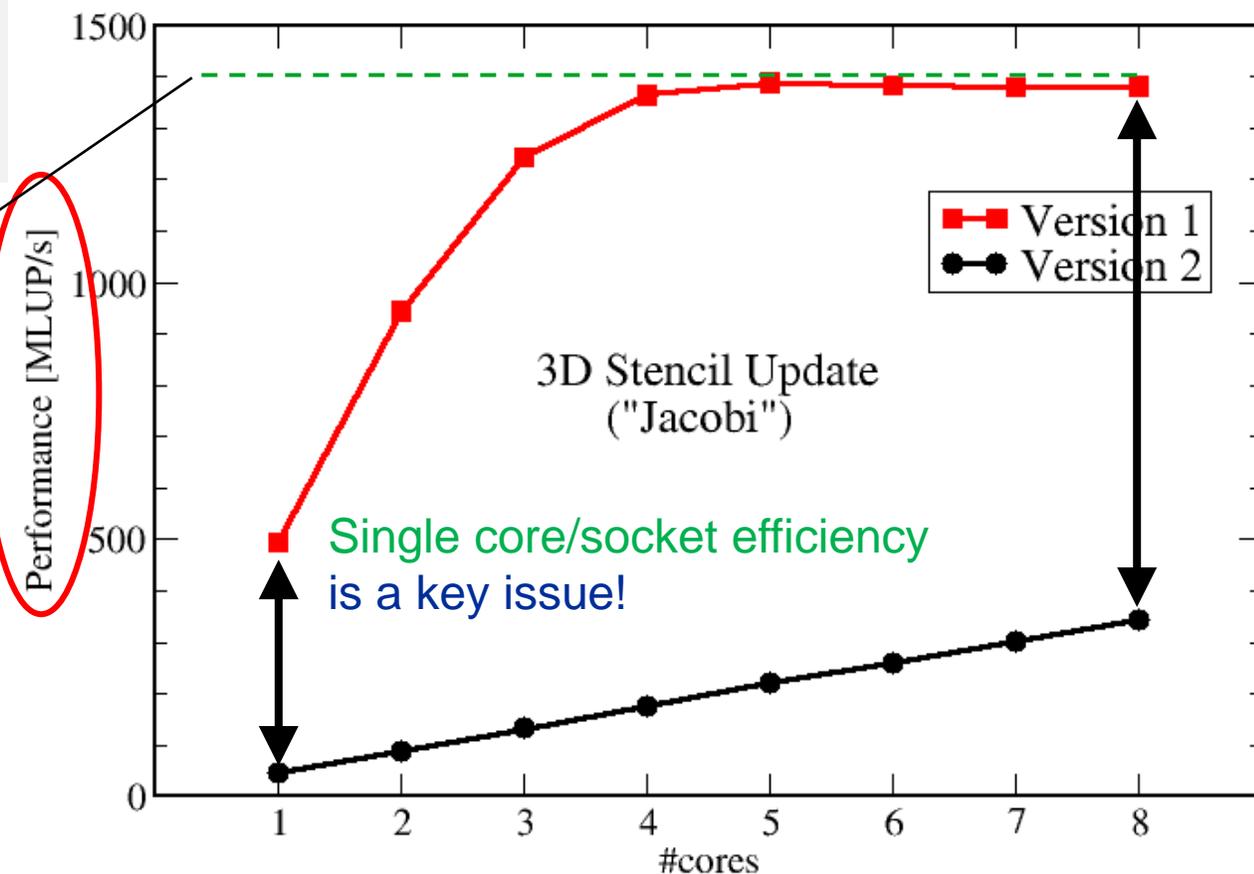
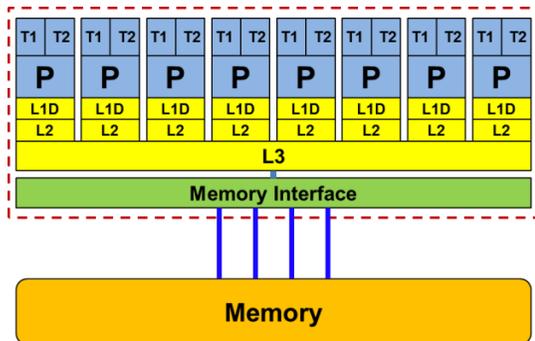
```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
    y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) +
                  x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
```



Scalability Myth: Code scalability is the key issue

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
    y(i,j,k) = b*( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) +
                  x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
  enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Upper limit from simple performance model:
35 GB/s & 24
Byte/update



Questions to ask in high performance computing

- Do I understand the performance behavior of my code?
 - Does the performance **behave in accordance with a model** I have made?
- What is the optimal performance for my code on a given machine?
 - **High Performance Computing == Computing at the bottleneck**
- Can I change my code so that the “optimal performance” gets higher?
 - **Circumventing/ameliorating** the impact of the **bottleneck**
- My model yields **wrong predictions** – what’s wrong?
 - This is the **good case**, because you **learn something**
 - Performance monitoring / microbenchmarking may help clear up the situation

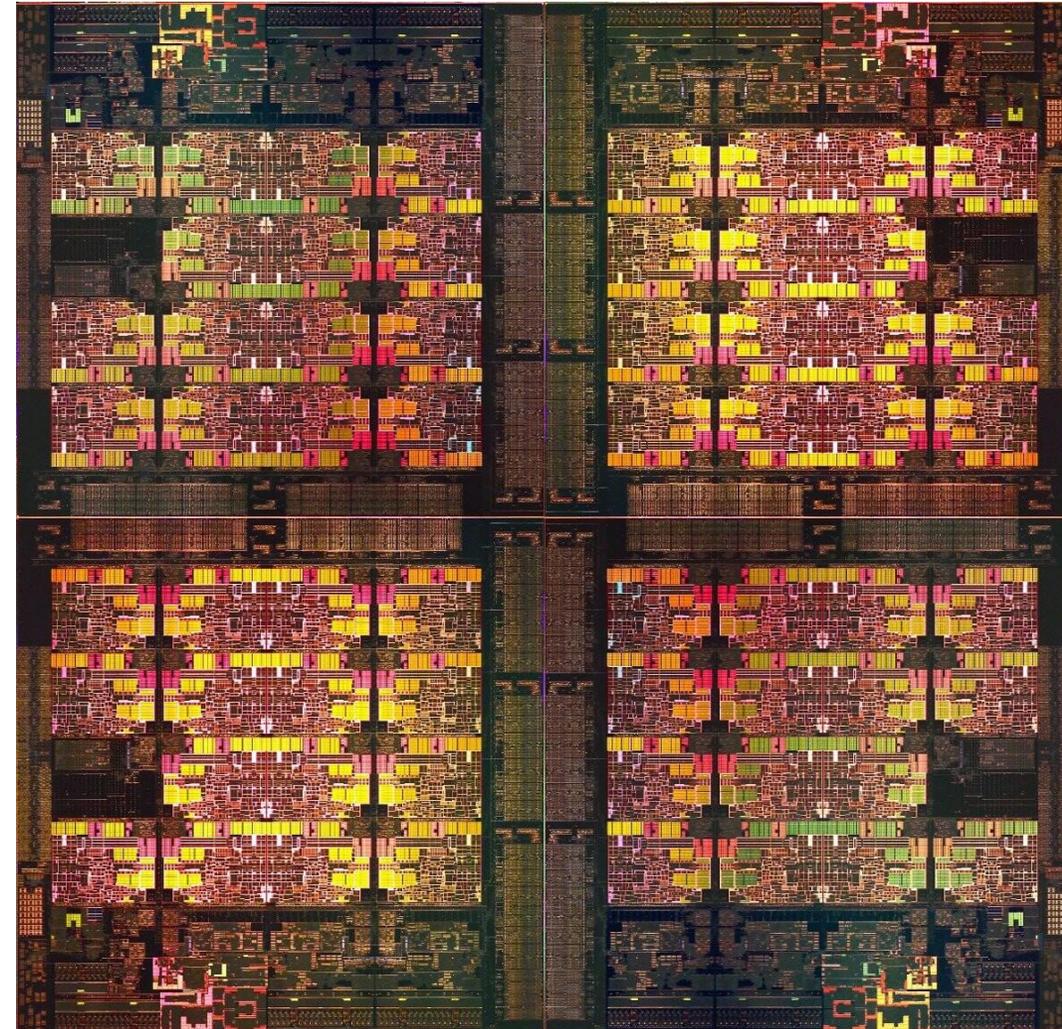
Modern computer architecture

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 😊



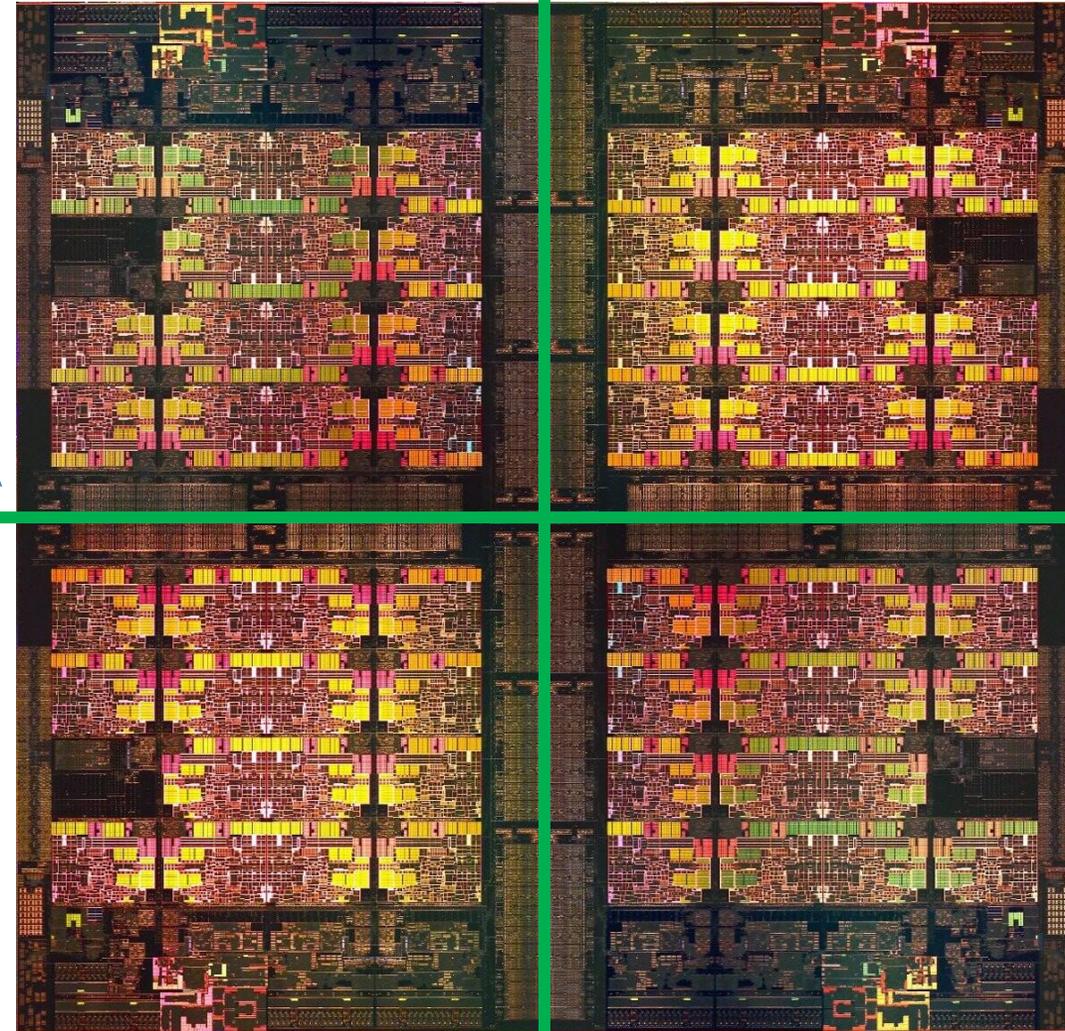
<https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated>

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



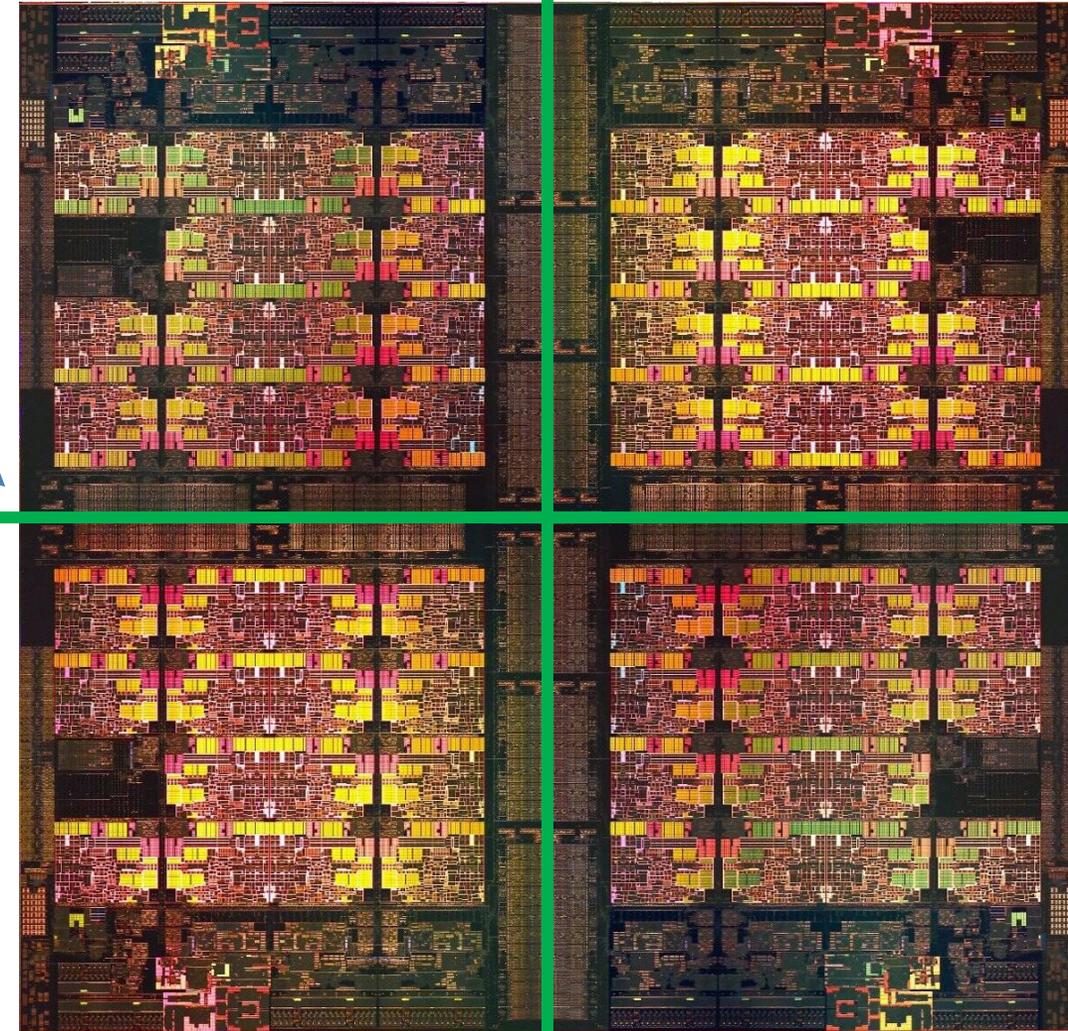
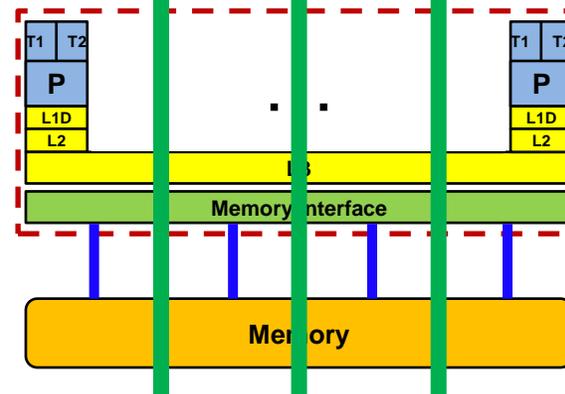
<https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated>

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

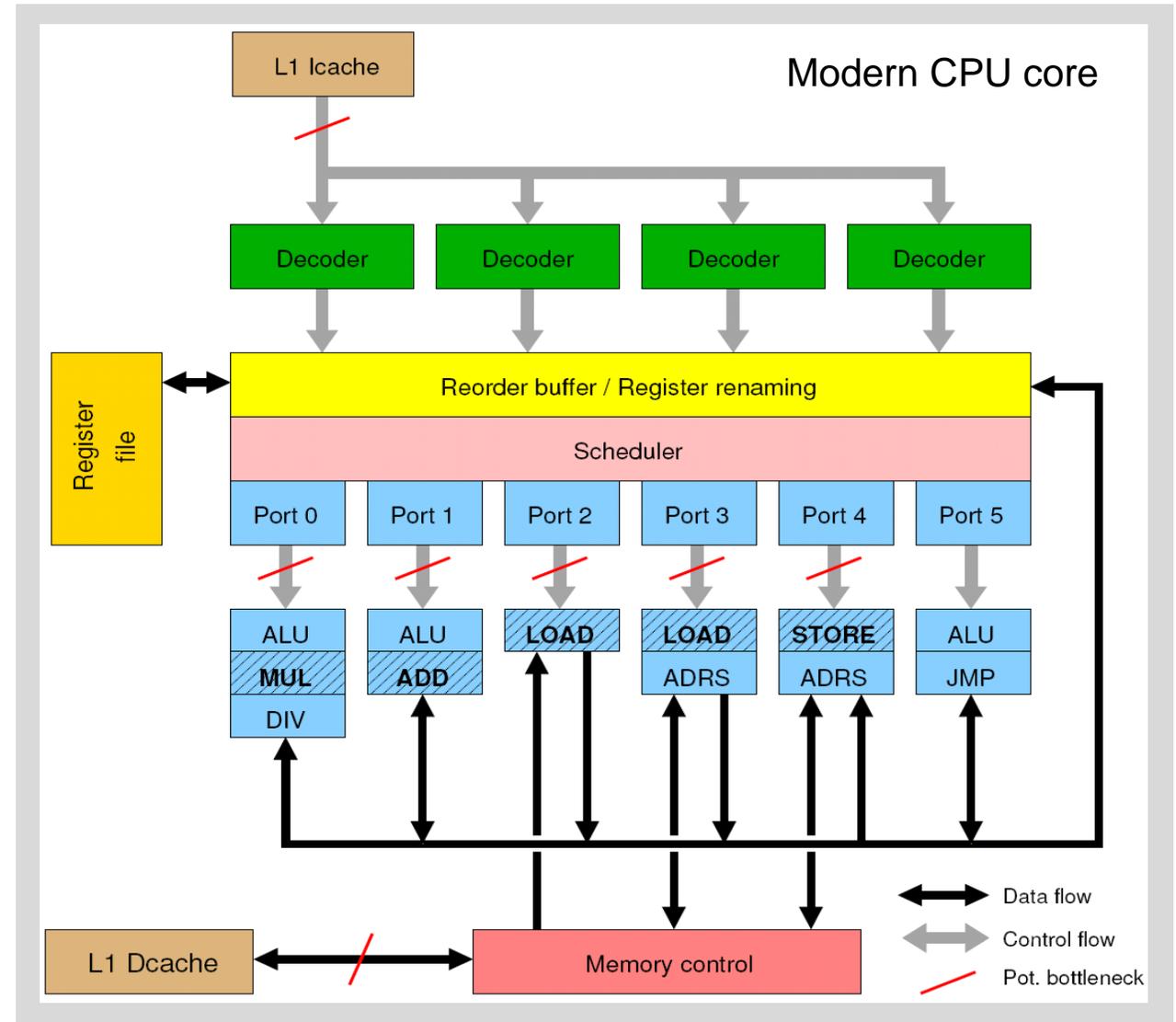


<https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated>

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

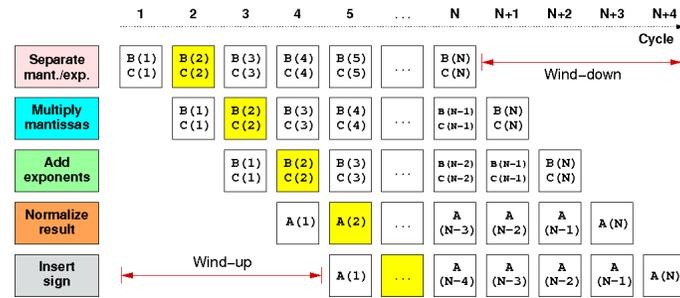
Pipelining, Superscalarity, SIMD, SMT



Important in-core features

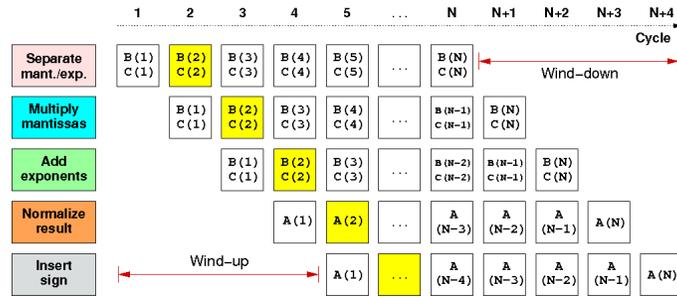
Important in-core features

Pipelining: Instruction execution in multiple steps

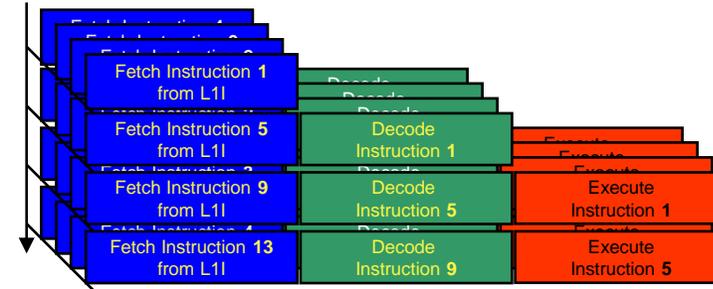


Important in-core features

Pipelining: Instruction execution in multiple steps

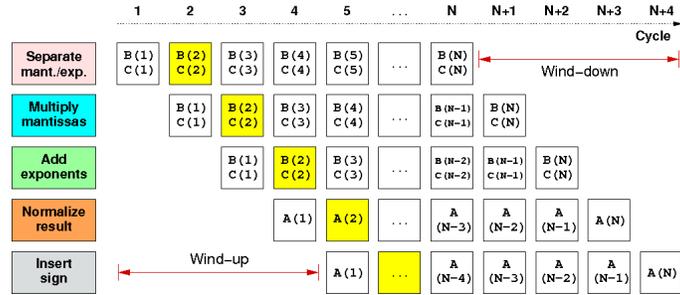


Superscalarity: Multiple instructions per cycle

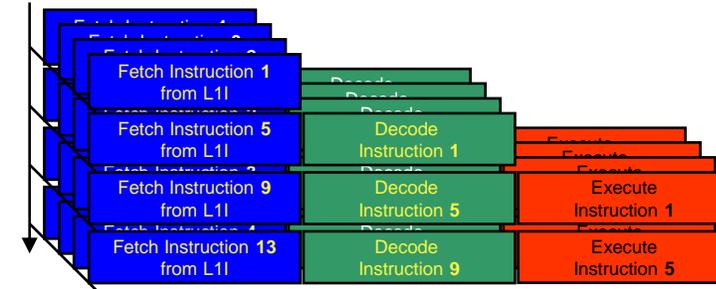


Important in-core features

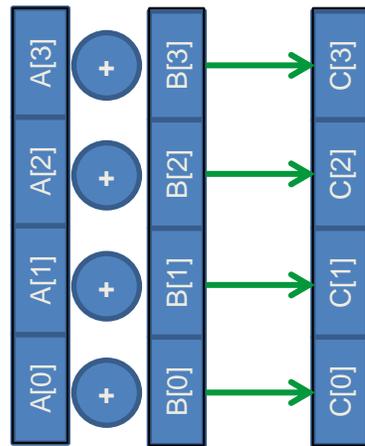
Pipelining: Instruction execution in multiple steps



Superscalarity: Multiple instructions per cycle

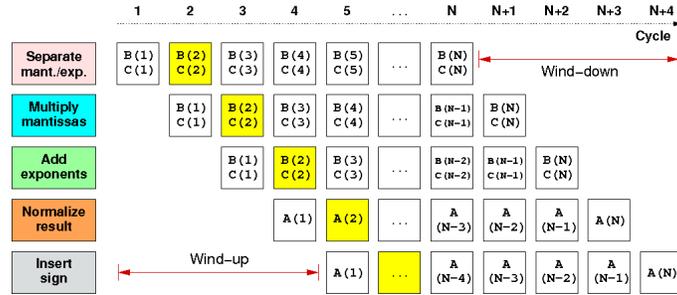


Single Instruction Multiple Data: Multiple operations per instruction

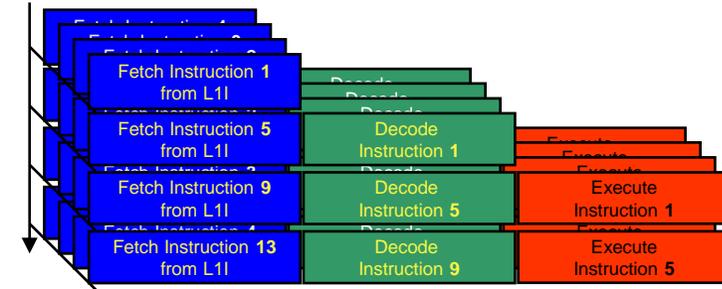


Important in-core features

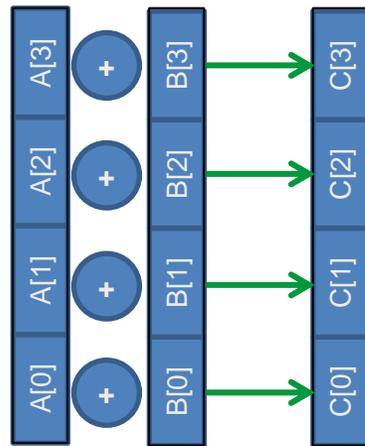
Pipelining:
Instruction execution in multiple steps



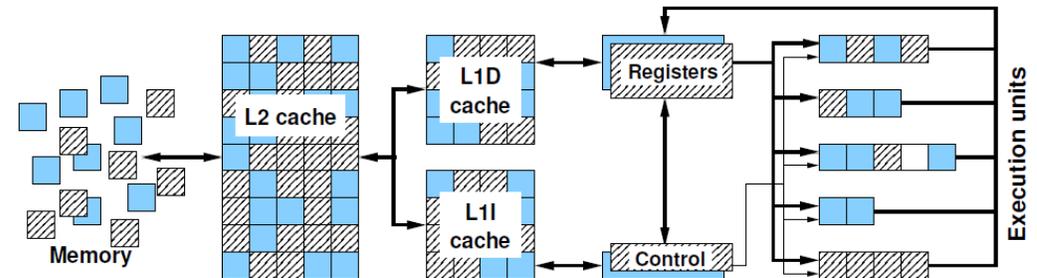
Superscalarity:
Multiple instructions per cycle



Single Instruction Multiple Data:
Multiple operations per instruction



Simultaneous Multi-Threading:
Multiple instruction sequences in parallel

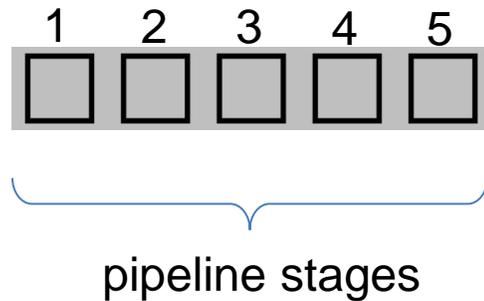


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

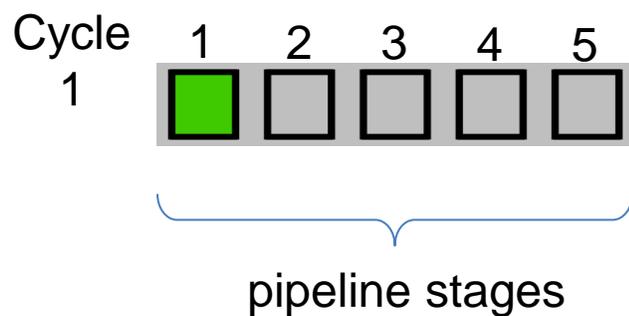


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

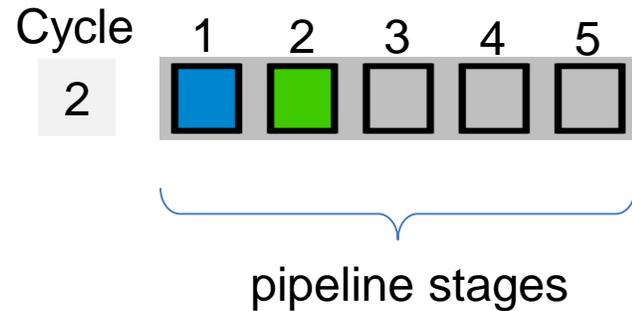


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

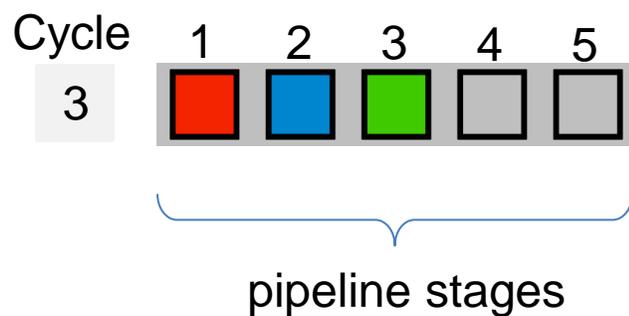


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

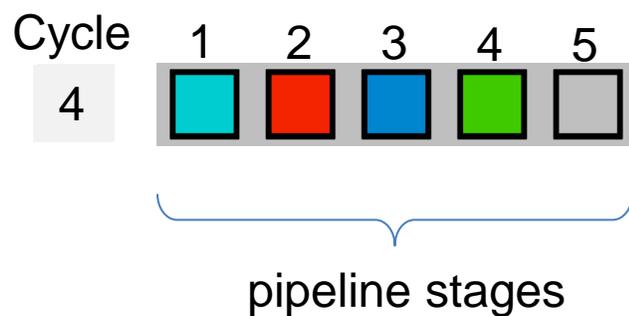


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

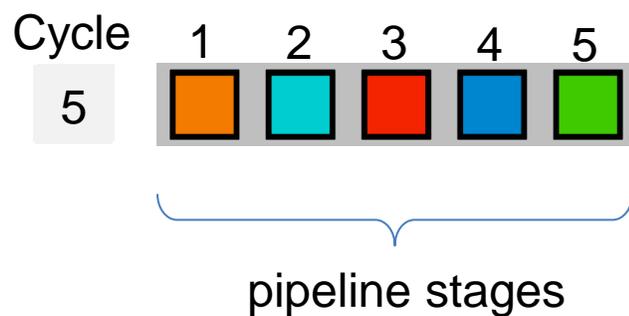


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)

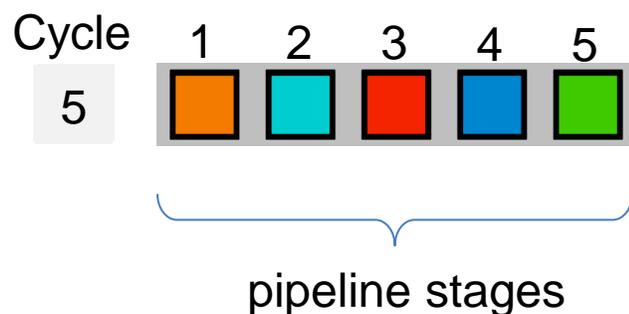


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (*latency*)



Throughput:

1 instruction per cycle after pipeline is full

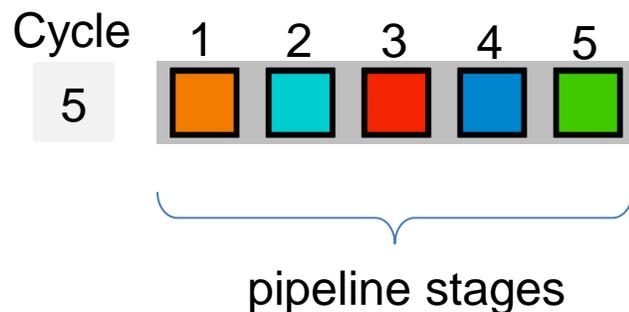
→ Speedup by factor 5

Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Single instruction takes 5 cycles (**latency**)



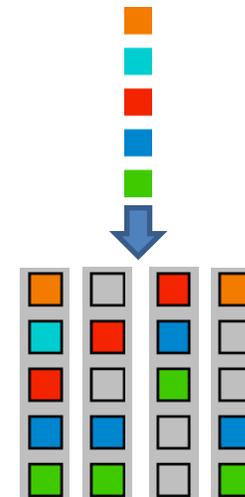
Throughput:

1 instruction per cycle after pipeline is full

→ Speedup by factor 5

Superscalar execution

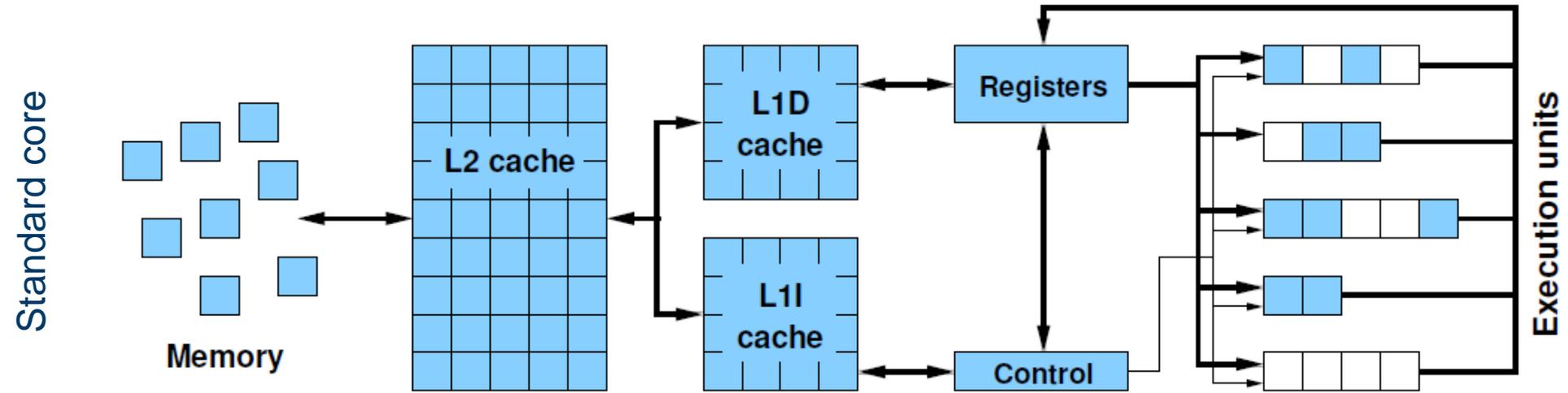
4-way **superscalar**:



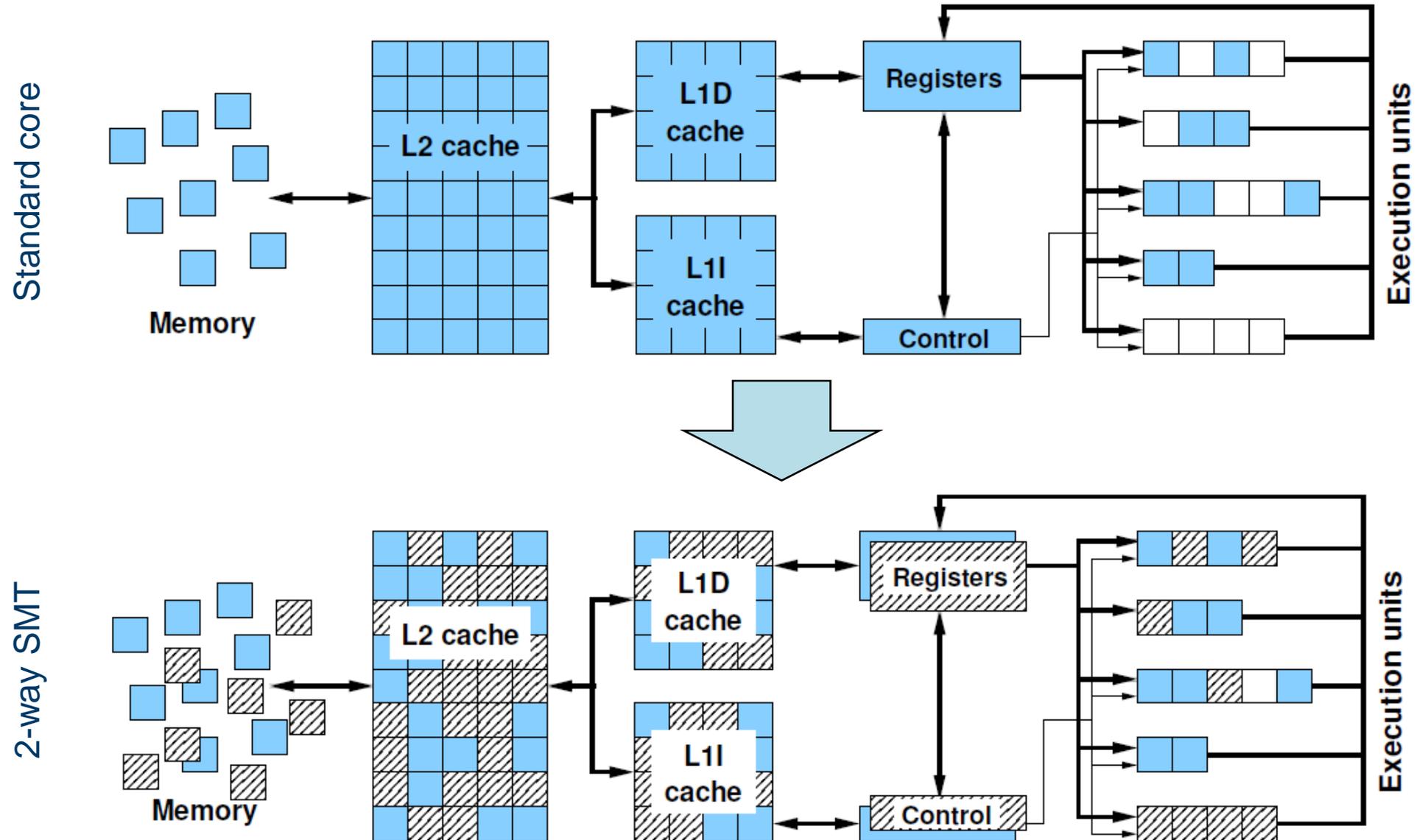
→ Massive boost in instruction throughput

→ **Instructions can be reordered on the fly**

Simultaneous multi-threading (SMT)

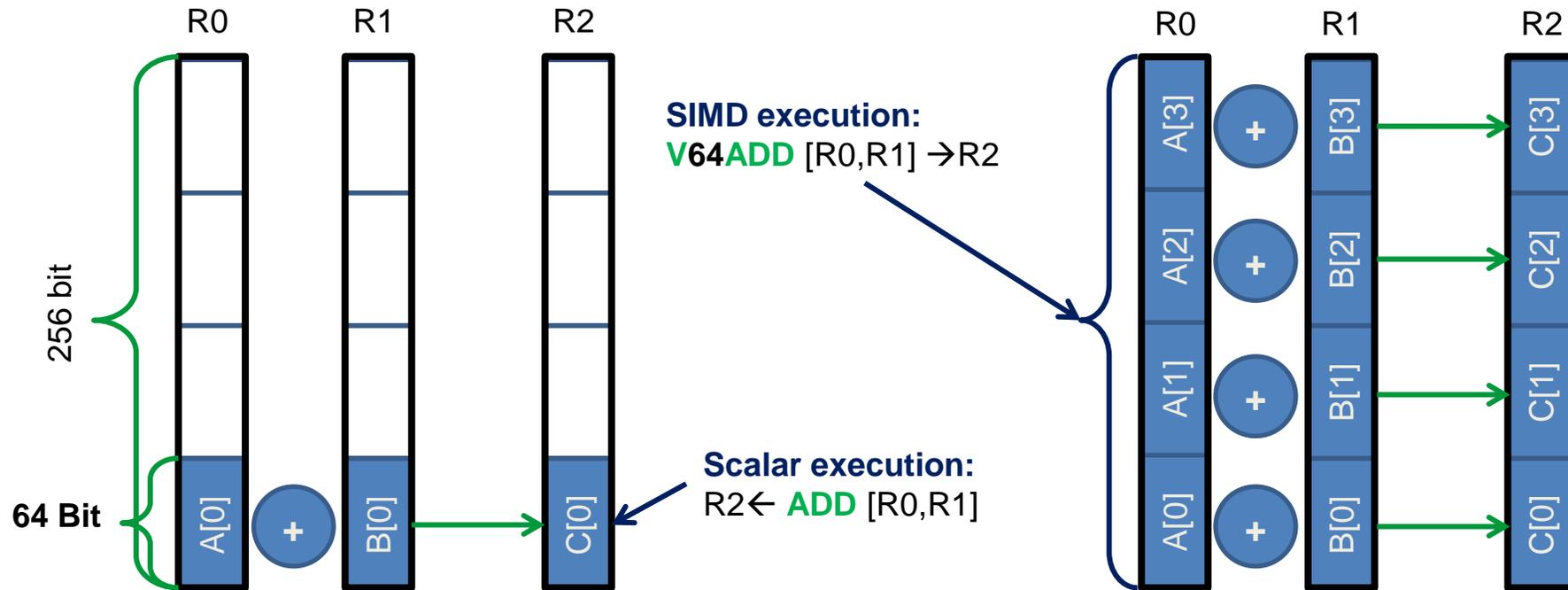


Simultaneous multi-threading (SMT)

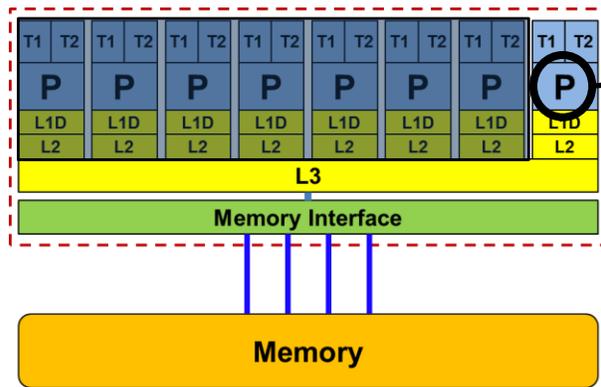


SIMD processing

- Single Instruction Multiple Data (SIMD) operations allow the execution of the same operation on “wide” registers from a single instruction
- 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]
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
Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
Sapphire Rapids	2	2	8	Q1/2023	Platinum 8480	2.0	64.0
AMD Zen (Naples)	2	2	2 (4)	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	Q4/2021	Epyc 7713	2.0	32.0
AMD Zen4 (Genoa)	2	2	4 (8)	Q4/2022	Epyc 9654	2.4	38.4
Fujitsu A64FX	2	2	8	Q2/2020	FX700	1.8	57.6

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?

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)

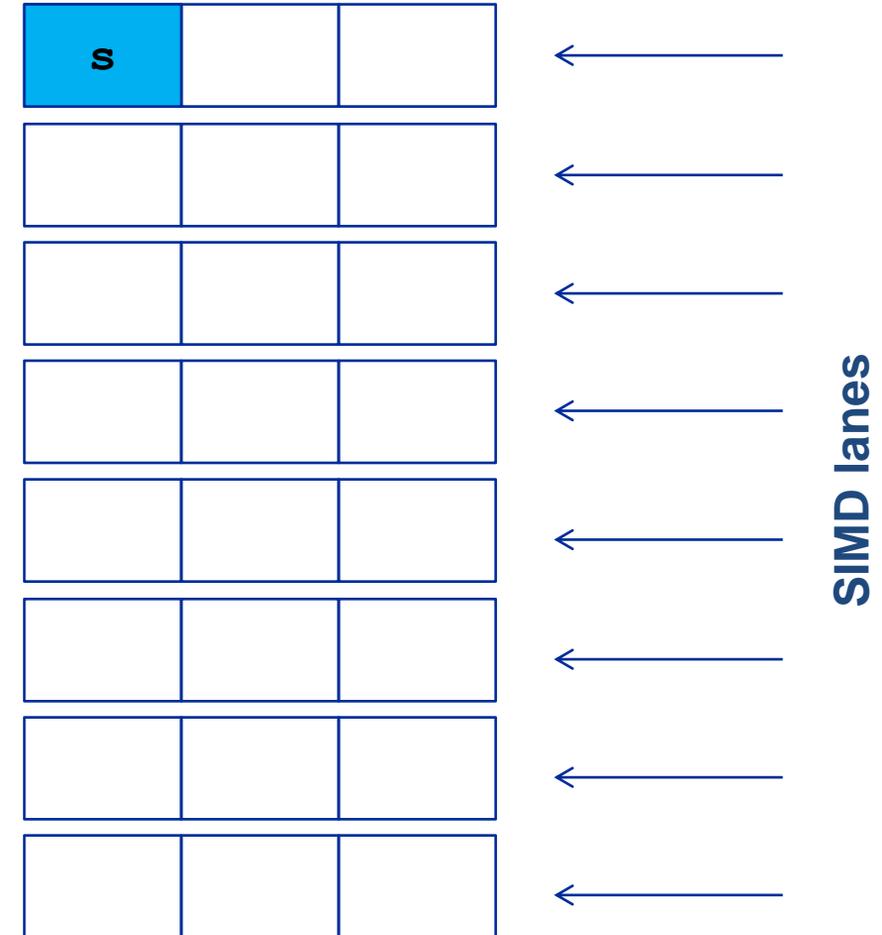
Plain scalar code, no SIMD

```
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

ADD pipes utilization:



Applicable peak for the sum reduction (I)

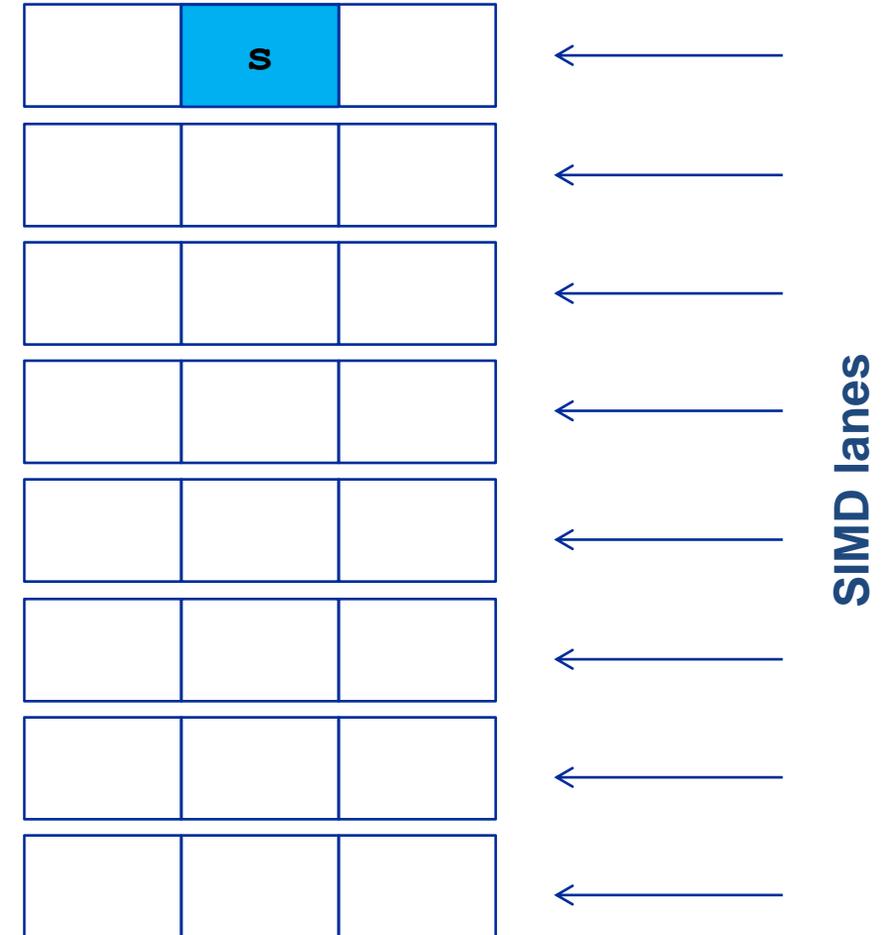
Plain scalar code, no SIMD

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

SIMD lane

```
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 (I)

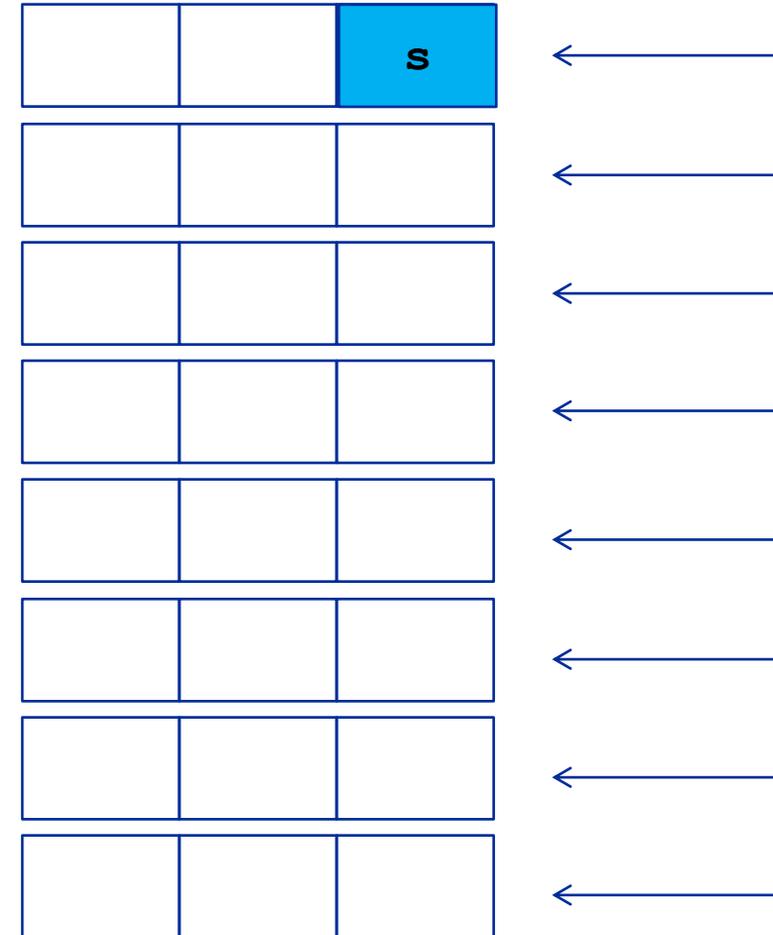
Plain scalar code, no SIMD

```
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

ADD pipes utilization:



SIMD lanes

→ 1/24 of ADD peak

Applicable peak for the sum reduction (II)

s1	s2	s3

→ 1/8 of ADD peak

Applicable peak for the sum reduction (II)

```
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 (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 (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;
```

s3	s1	s2

→ 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:

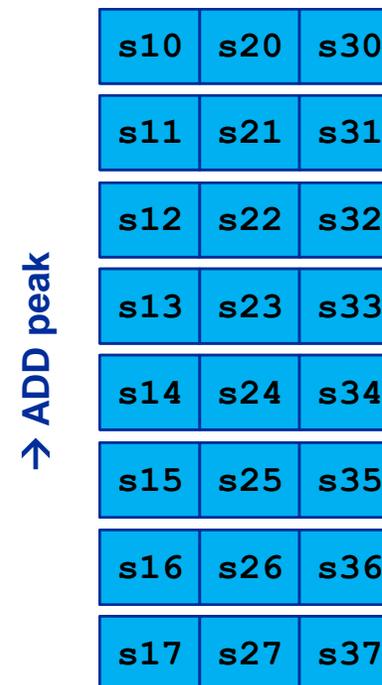
```
LOAD [r4.0,...,r4.7] ← [a(i),...,a(i+7)] # SIMD LOAD
LOAD [r5.0,...,r5.7] ← [a(i+8),...,a(i+15)] # SIMD
LOAD [r6.0,...,r6.7] ← [a(i+16),...,a(i+23)] # SIMD
```

```
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

Sum reduction

Questions

- When can this performance actually be achieved?

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

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?

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

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?

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

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?

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

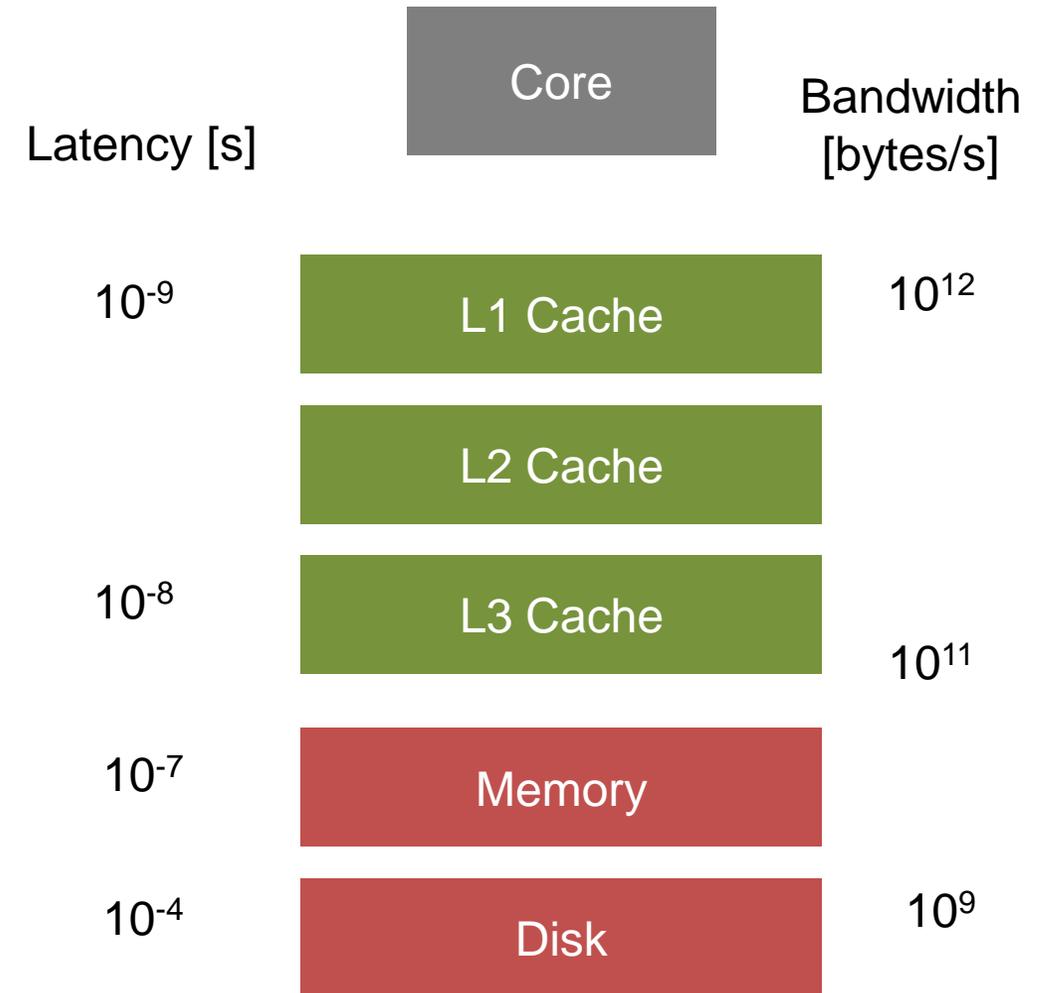
In-cache performance (L2, L3)

Main memory performance



Memory hierarchy

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

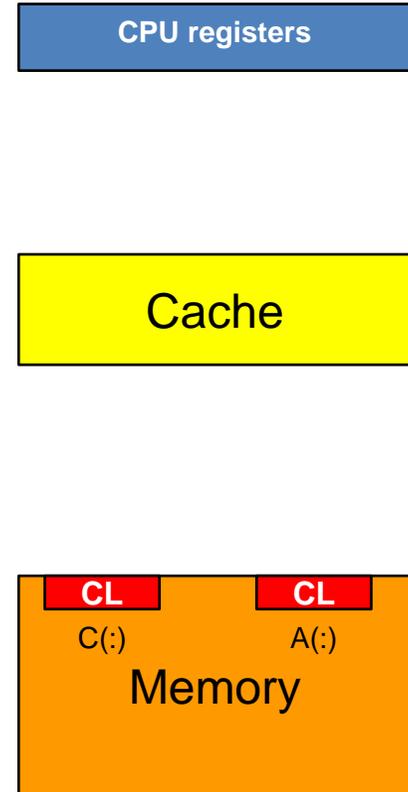


Purpose of many optimizations is to **load data from fast memory**

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?



- Example: Array copy $\mathbf{A}(:) = \mathbf{C}(:)$

Data transfers in a memory hierarchy

Caches help with getting instructions and data to the CPU “fast”

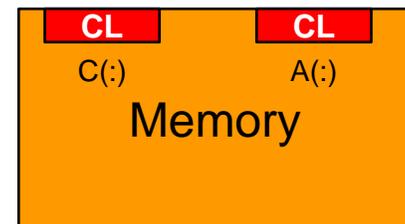
How does data travel from memory to the CPU and back?

- 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 $\mathbf{A}(:) = \mathbf{C}(:)$

CPU registers

Cache

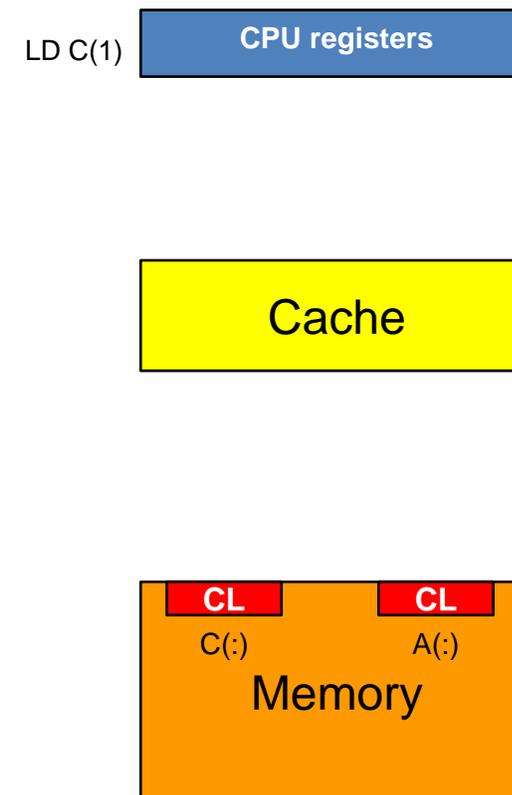


Data transfers in a memory hierarchy

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 $\mathbf{A}(:) = \mathbf{C}(:)$



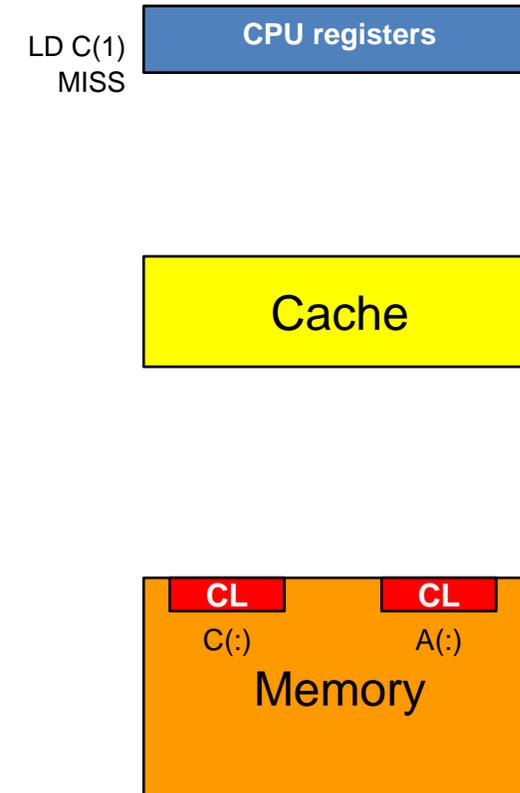
Data transfers in a memory hierarchy

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 $\mathbf{A}(:) = \mathbf{C}(:)$



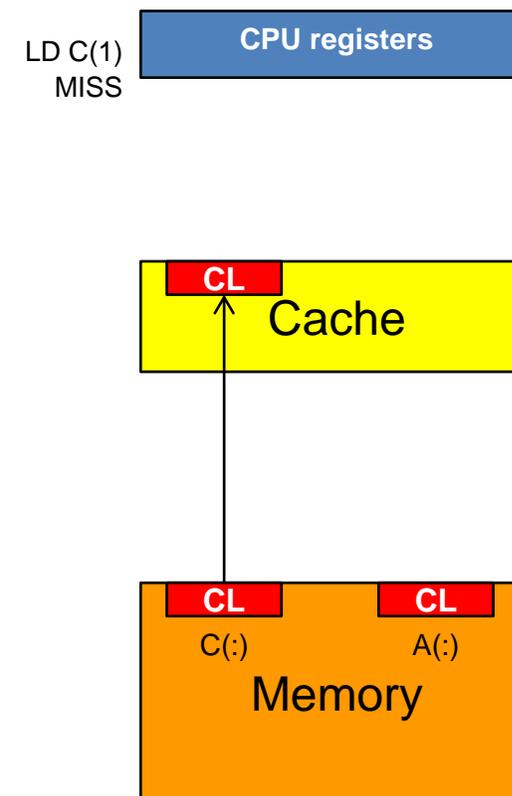
Data transfers in a memory hierarchy

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 $A(:) = C(:)$



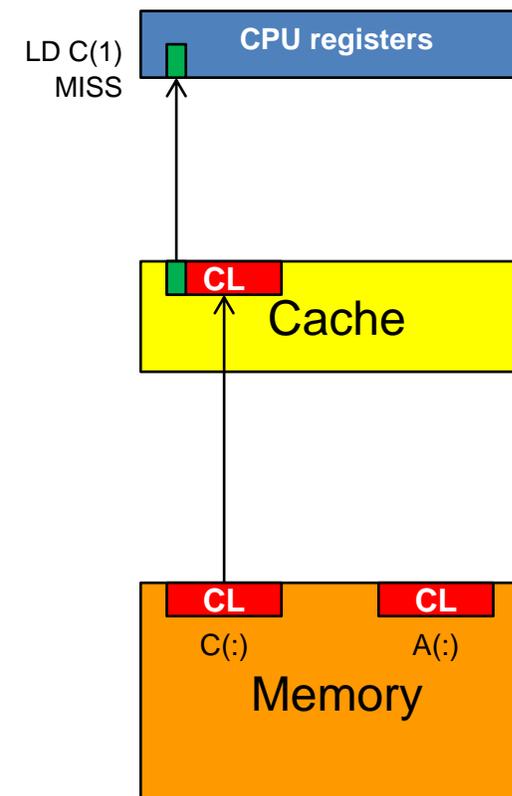
Data transfers in a memory hierarchy

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 $A(:) = C(:)$



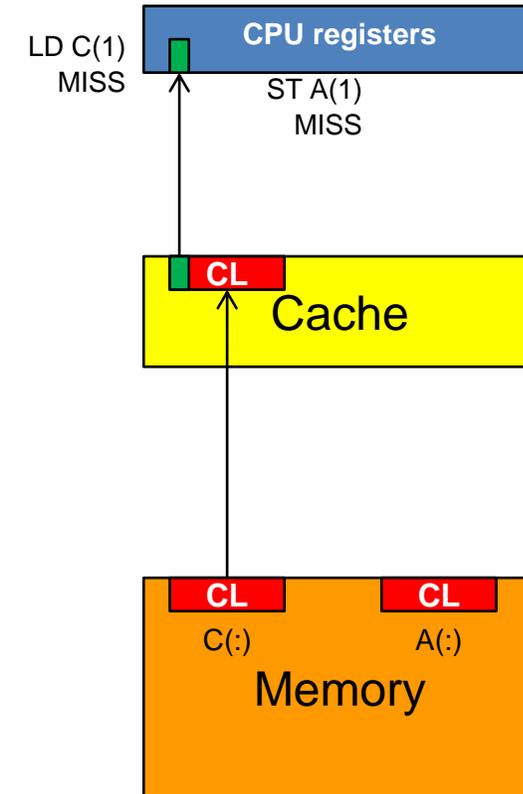
Data transfers in a memory hierarchy

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 $A(:) = C(:)$



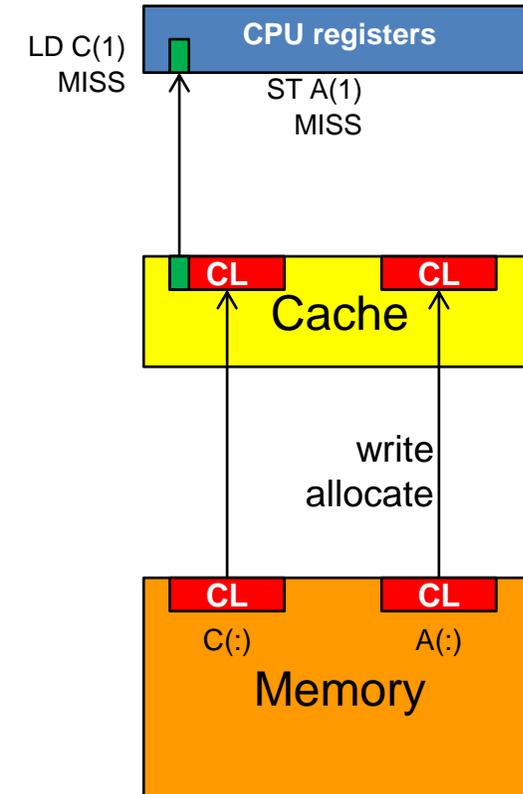
Data transfers in a memory hierarchy

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 $A(:) = C(:)$



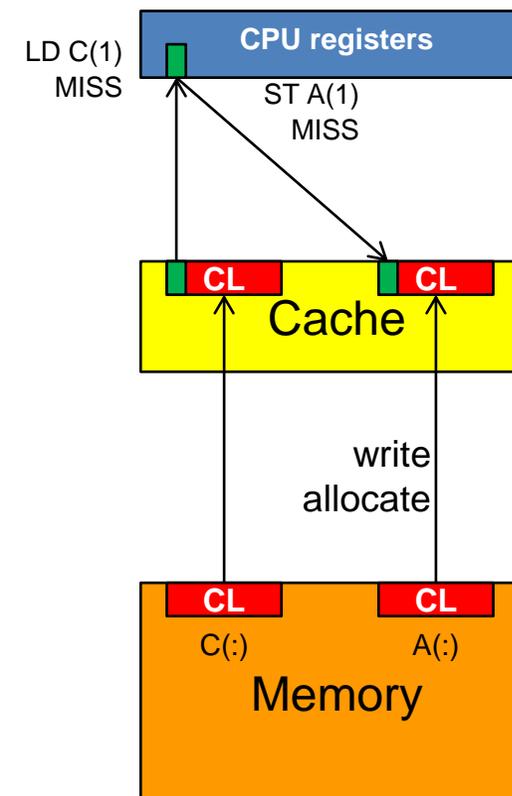
Data transfers in a memory hierarchy

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 $A(:) = C(:)$

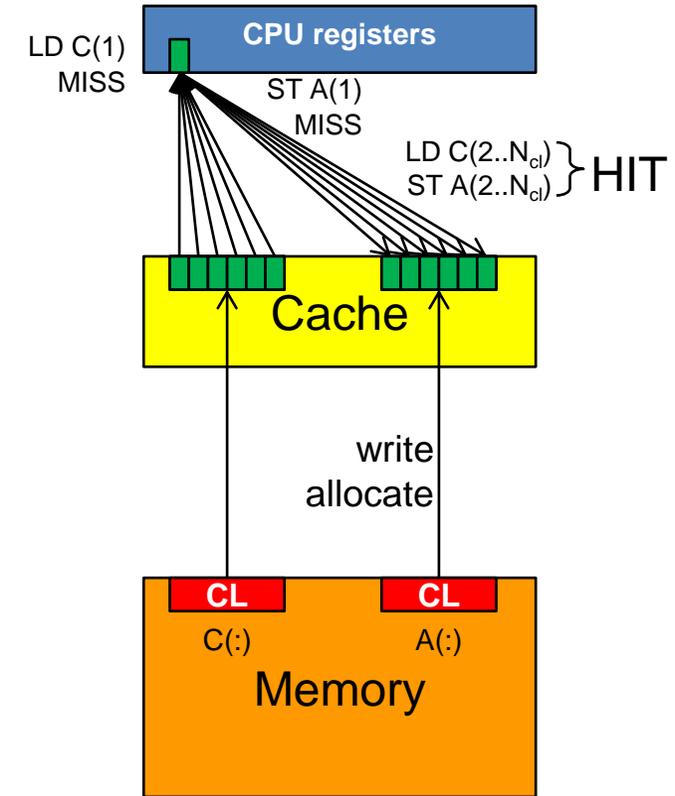


Data transfers in a memory hierarchy

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 $\mathbf{A}(:) = \mathbf{C}(:)$

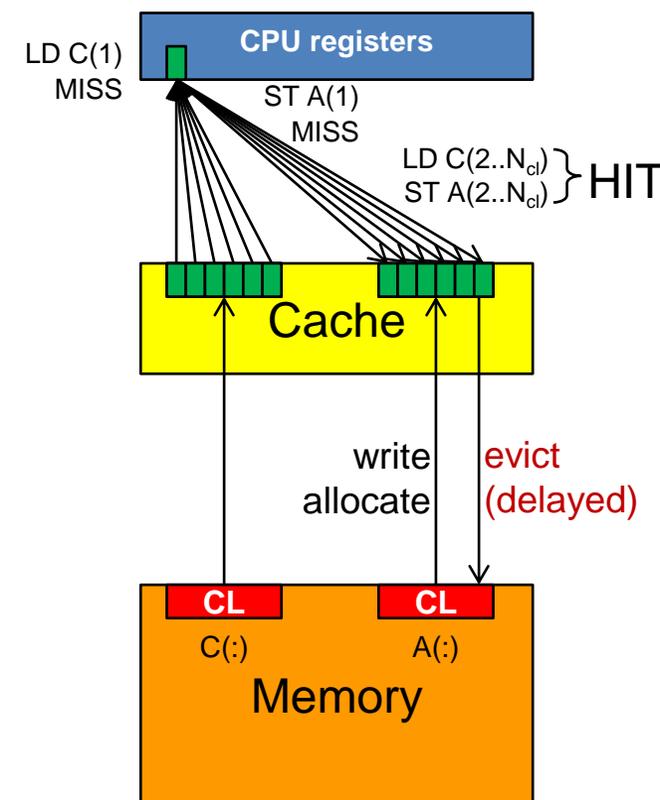


Data transfers in a memory hierarchy

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 $\mathbf{A}(:) = \mathbf{C}(:)$

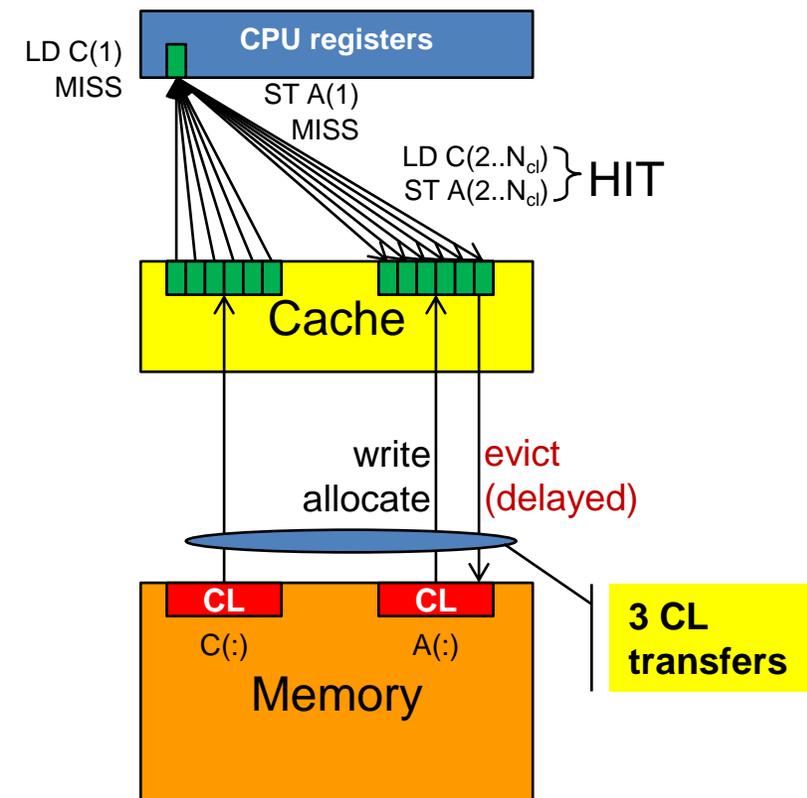


Data transfers in a memory hierarchy

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 $A(:) = C(:)$



Multicore

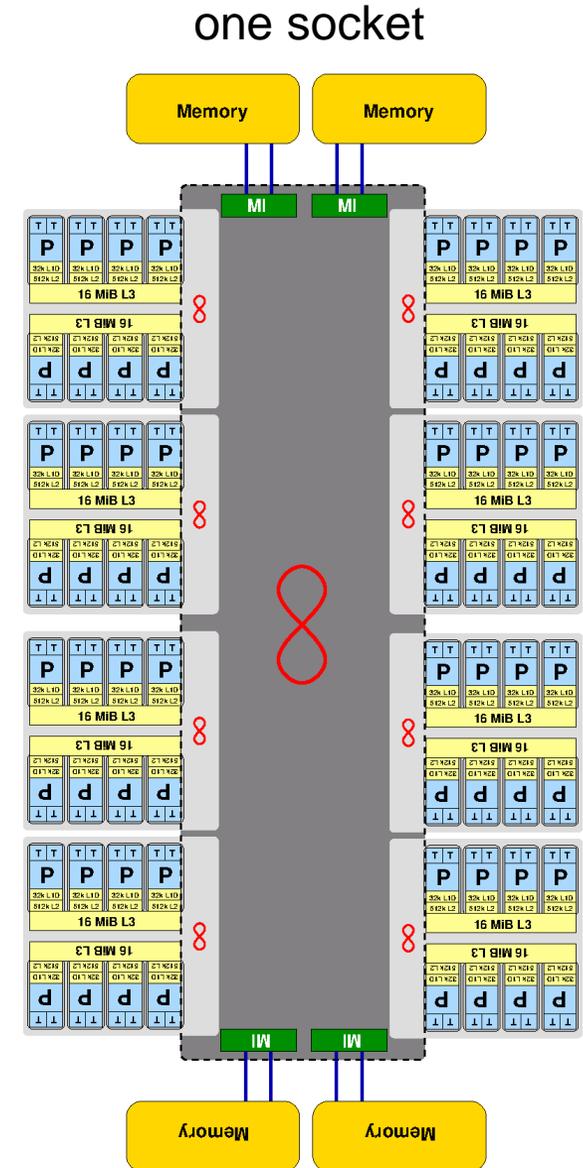
Node topology and performance



Putting the cores & caches together

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

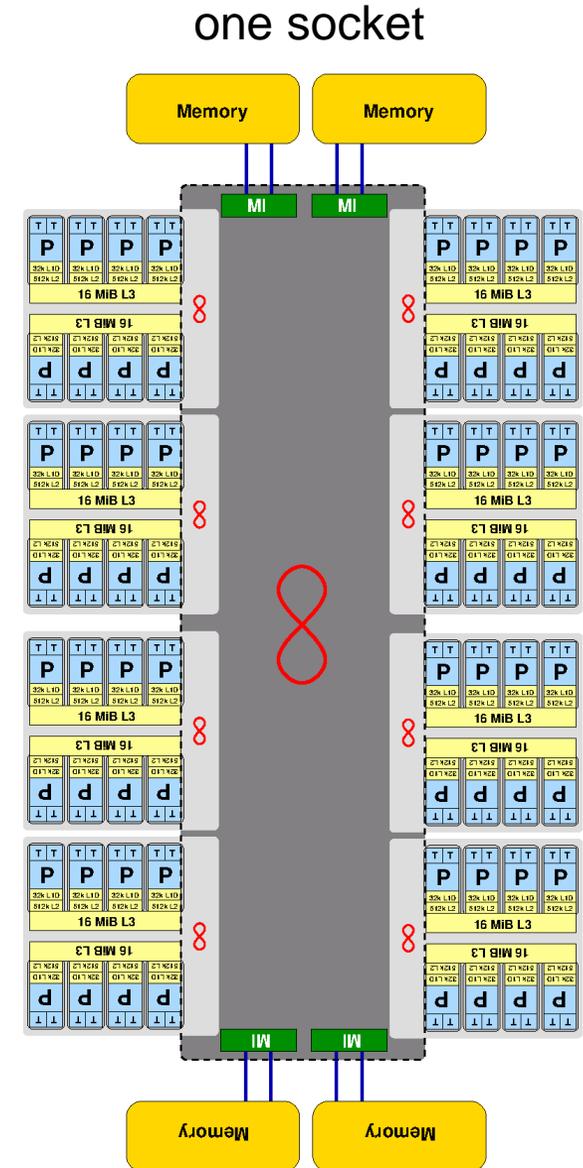
- Core features:
 - Two-way SMT
 - Two 256-bit SIMD FMA units (AVX2)
→ 16 flops/cycle (actually 24 because 2 ADDs can be done alongside)
 - 32 KiB L1 data cache per core
 - 512 KiB L2 cache per core



Putting the cores & caches together

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

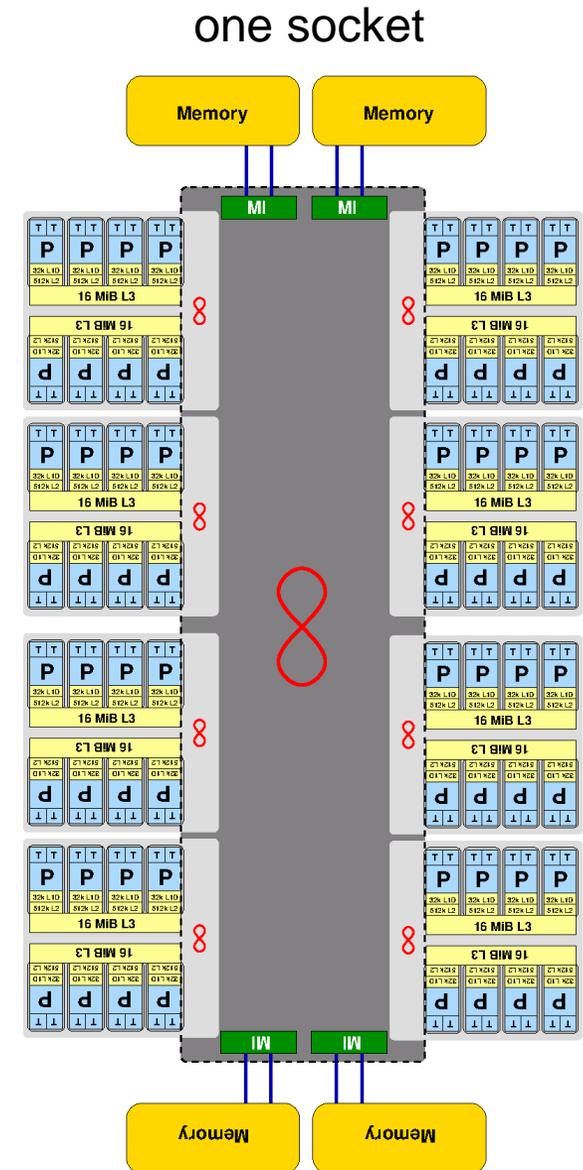
- Core features:
 - Two-way SMT
 - Two 256-bit SIMD FMA units (AVX2)
 - 16 flops/cycle (actually 24 because 2 ADDs can be done alongside)
 - 32 KiB L1 data cache per core
 - 512 KiB L2 cache per core
- 64 cores per socket hierarchically built up from
 - 16 CCX with 4 cores and 16 MiB of L3 cache
 - 2 CCX form 1 CCD (silicon die)
 - 8 CCDs connected to IO device “Infinity Fabric” (memory controller & PCIe)



Putting the cores & caches together

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

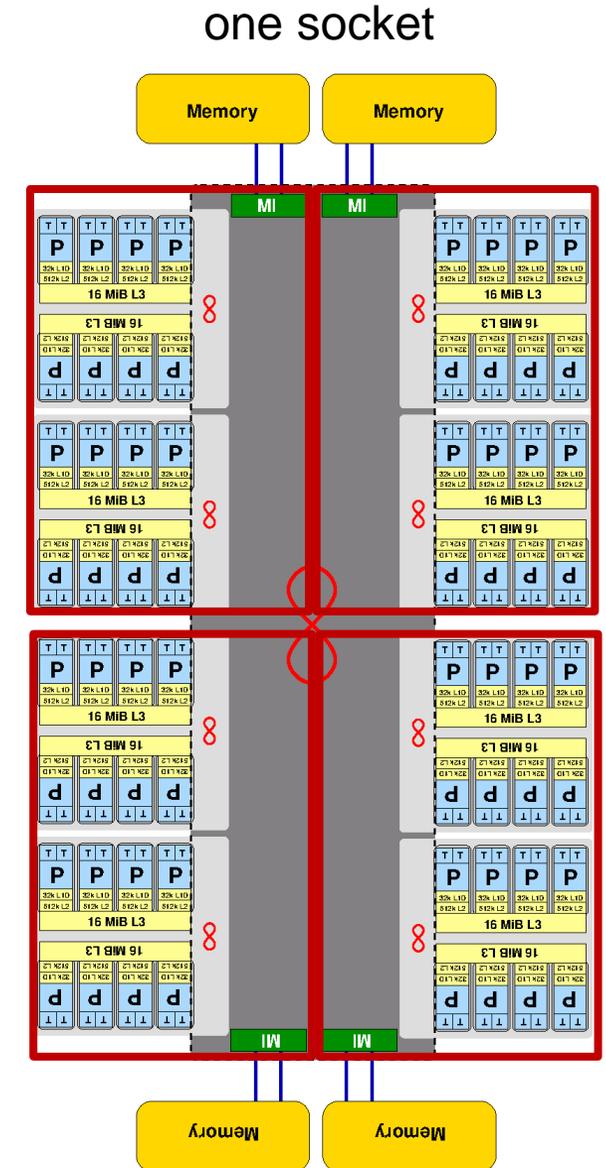
- Core features:
 - Two-way SMT
 - Two 256-bit SIMD FMA units (AVX2)
 - 16 flops/cycle (actually 24 because 2 ADDs can be done alongside)
 - 32 KiB L1 data cache per core
 - 512 KiB L2 cache per core
- 64 cores per socket hierarchically built up from
 - 16 CCX with 4 cores and 16 MiB of L3 cache
 - 2 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):
 - Node Per Socket (NPS)=1 , 2 or 4



Putting the cores & caches together

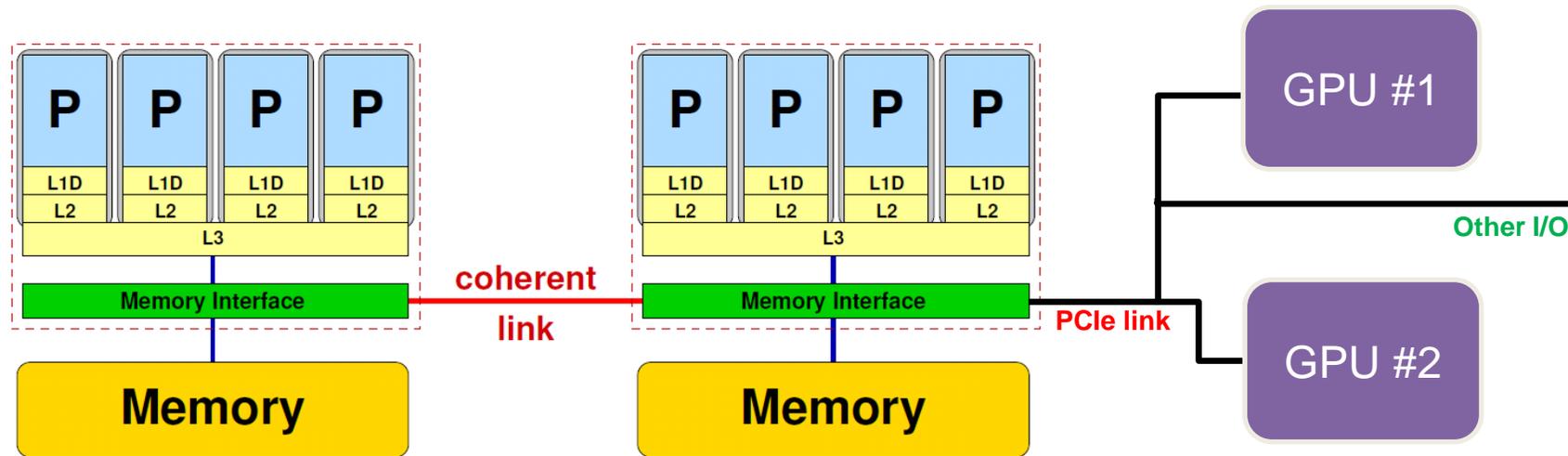
AMD Epyc 7742 64-Core Processor («Rome»)

- Core features:
 - Two-way SMT
 - Two 256-bit SIMD FMA units (AVX2)
 - 16 flops/cycle (actually 24 because 2 ADDs can be done alongside)
 - 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):
 - Node Per Socket (NPS)=1 , 2 or 4
 - NPS=4 → 4 ccNUMA domains



Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node

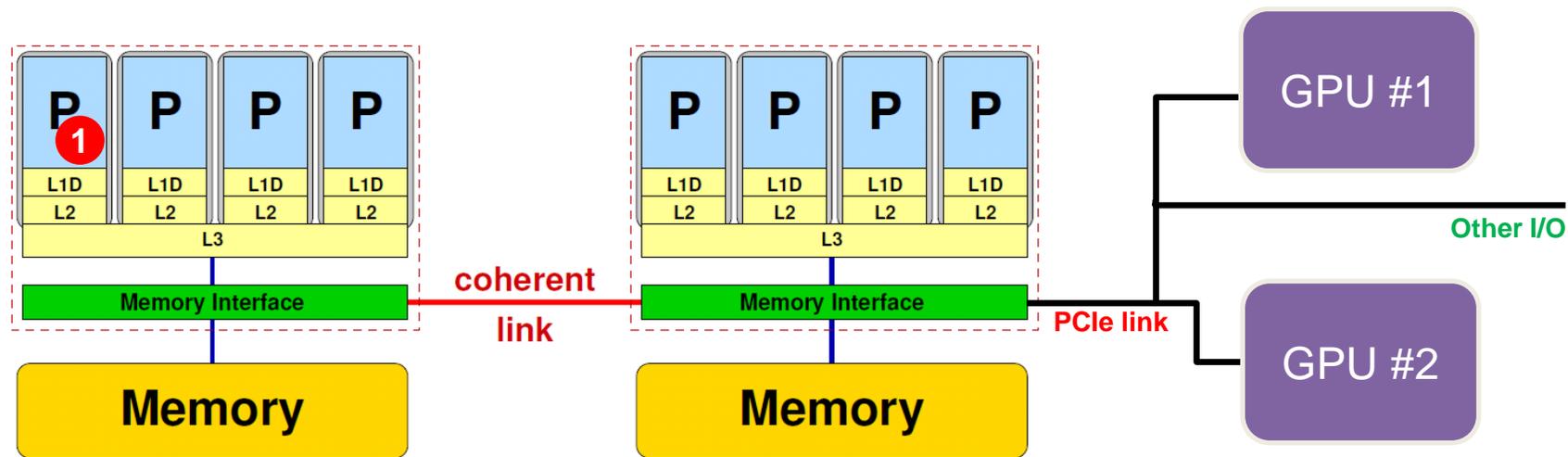


Parallel resources:

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

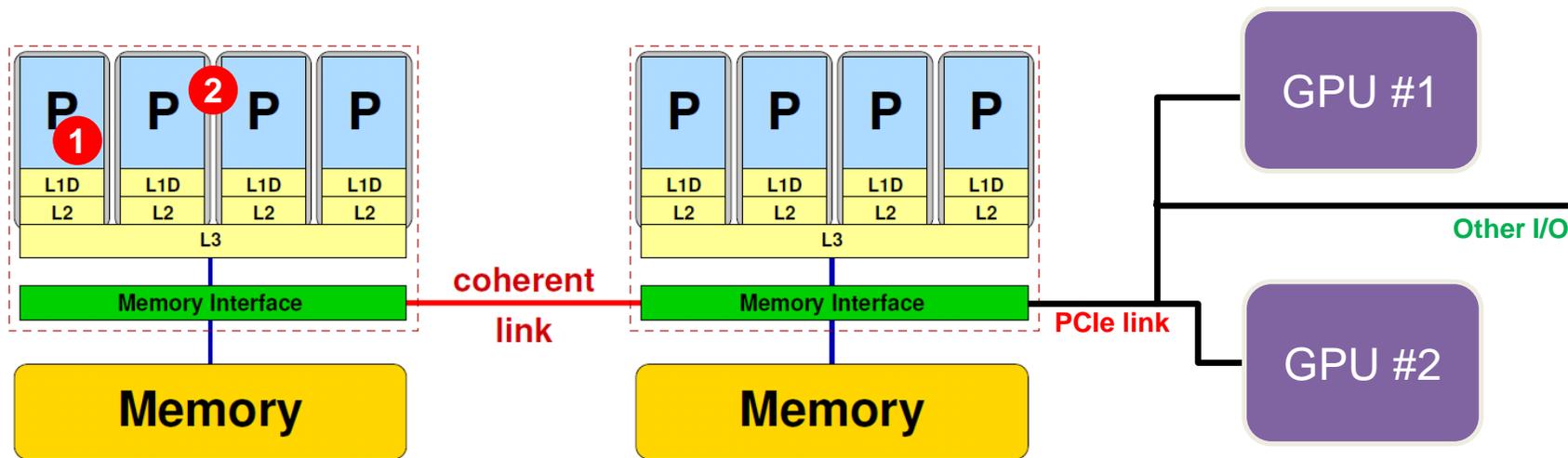
- Execution/SIMD units

1

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



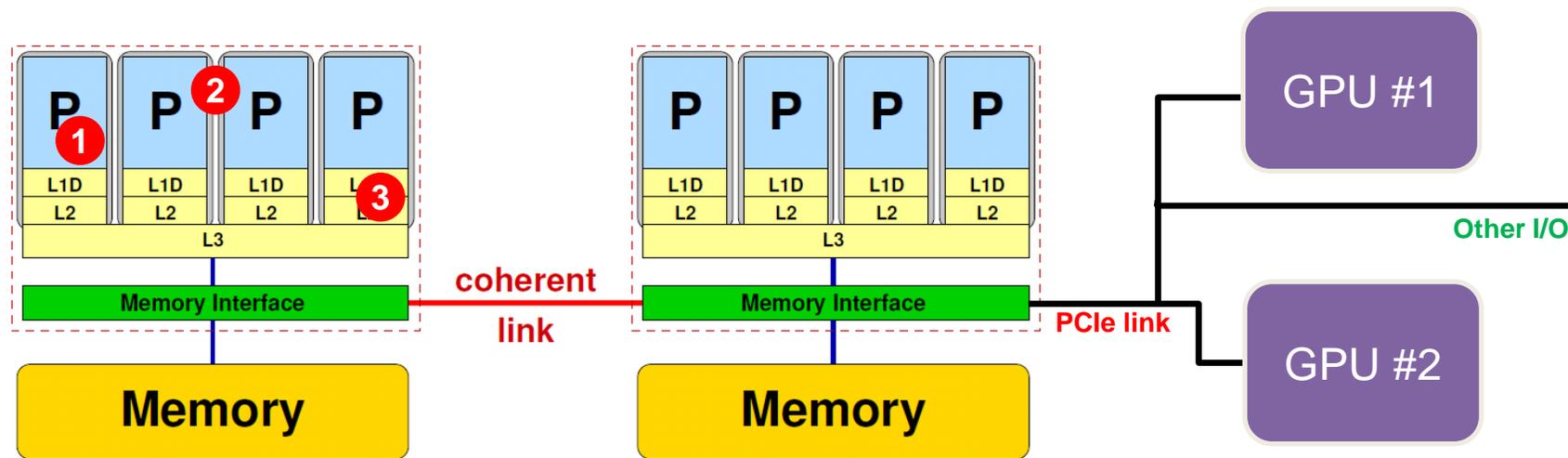
Parallel resources:

- Execution/SIMD units **1**
- Cores **2**

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



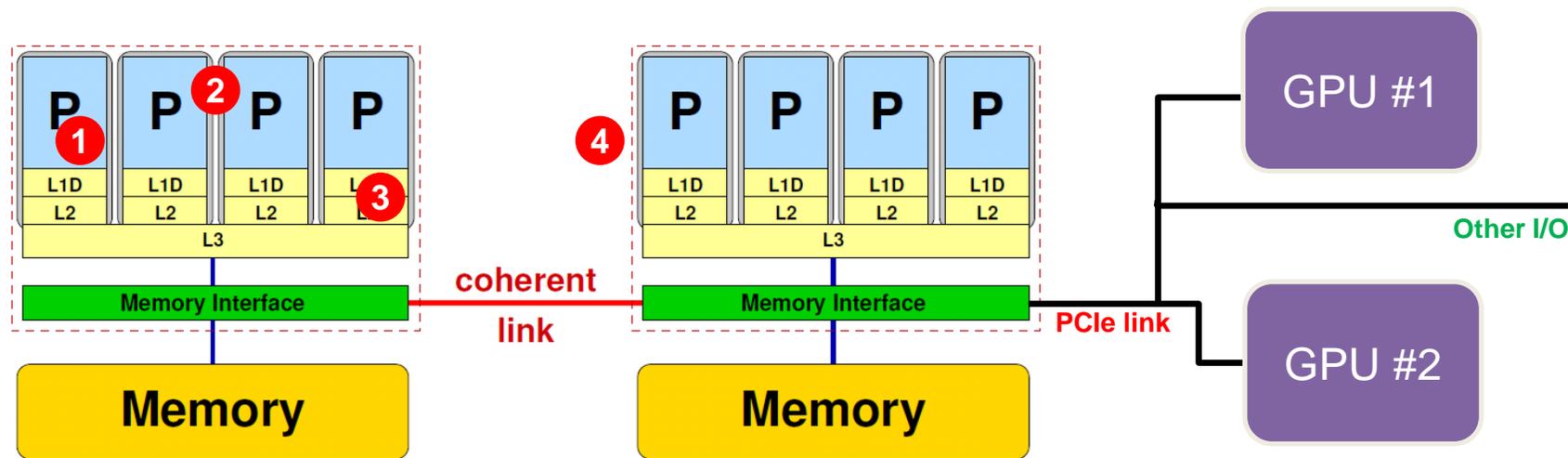
Parallel resources:

- Execution/SIMD units **1**
- Cores **2**
- Inner cache levels **3**

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



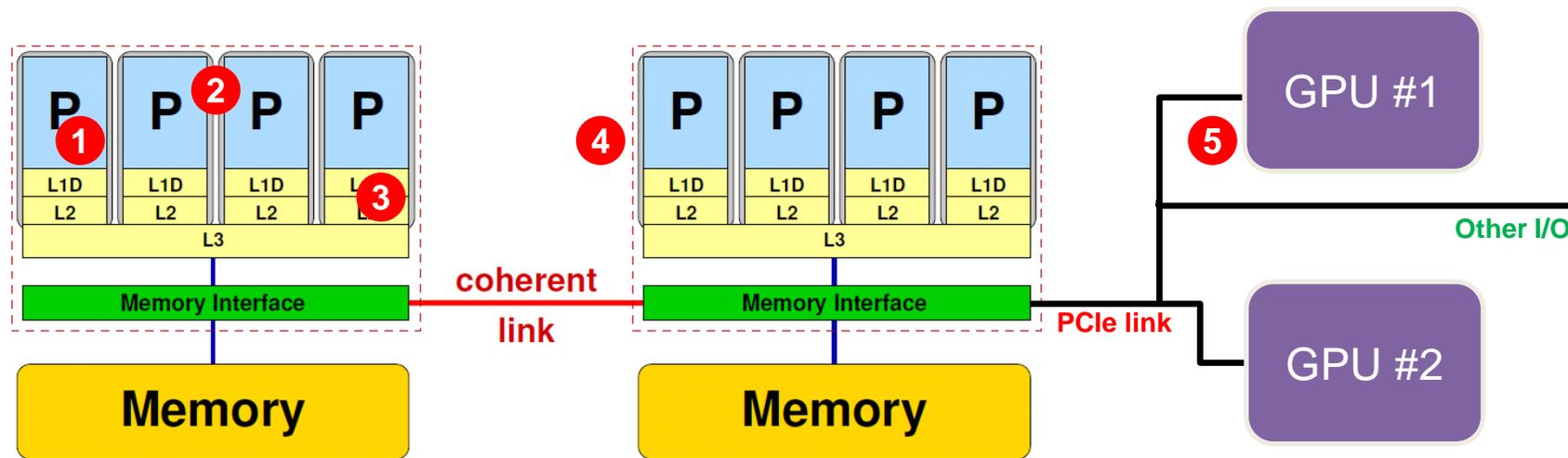
Parallel resources:

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

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



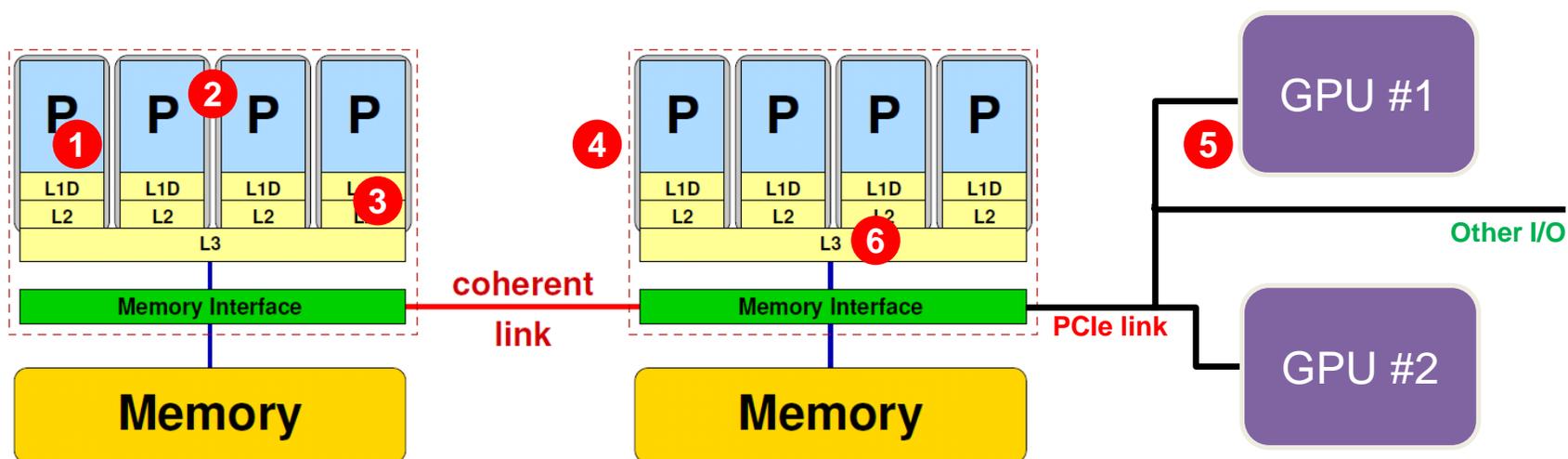
Parallel resources:

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

Shared resources:

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

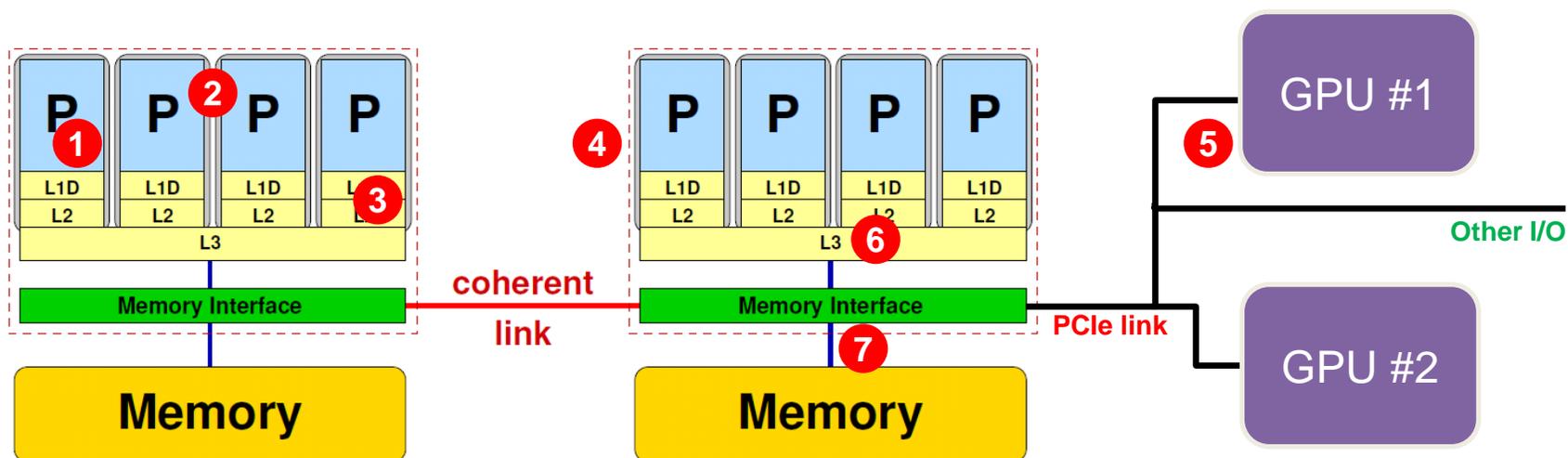
- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

Shared resources:

- Outer cache level per socket 6

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

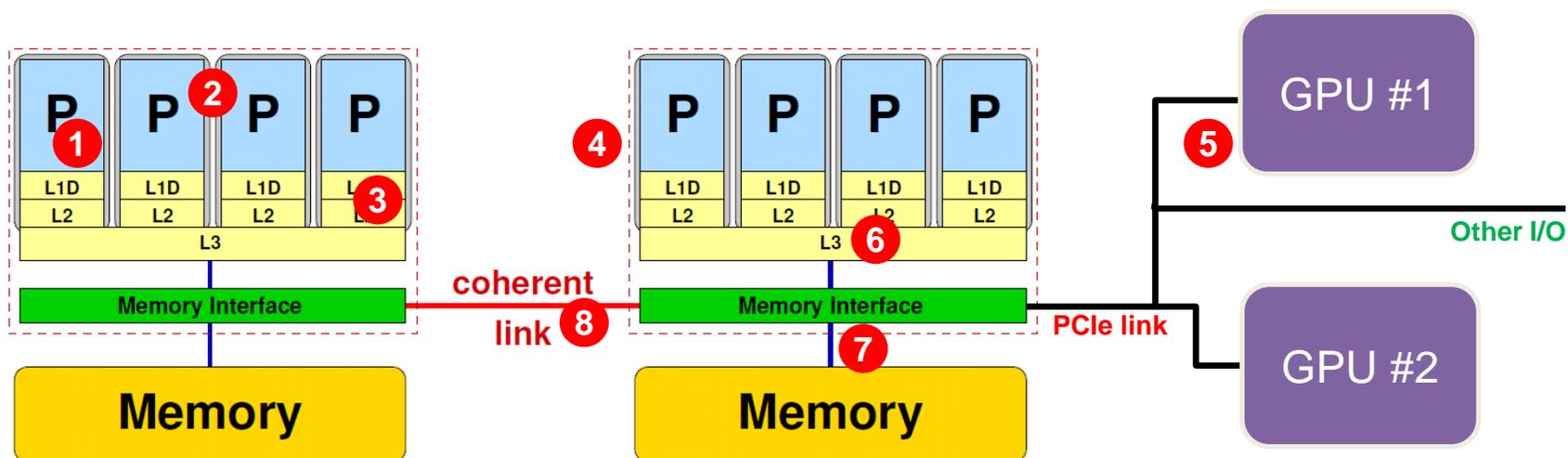
- Execution/SIMD units **1**
- Cores **2**
- Inner cache levels **3**
- Sockets / ccNUMA domains **4**
- Multiple accelerators **5**

Shared resources:

- Outer cache level per socket **6**
- Memory bus per socket **7**

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

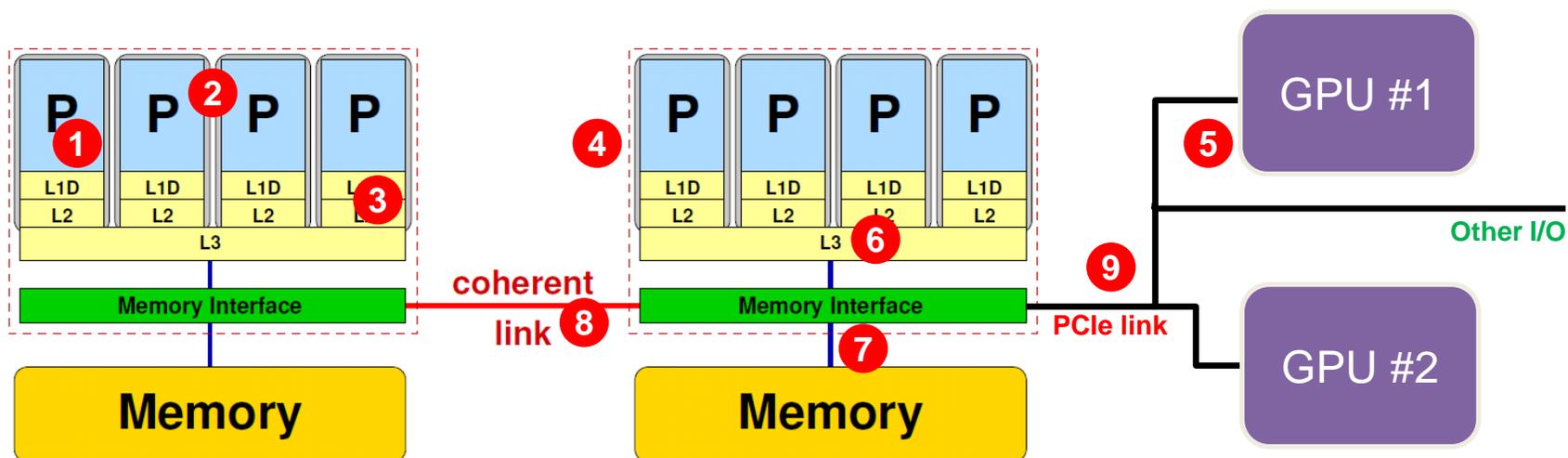
- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

Shared resources:

- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

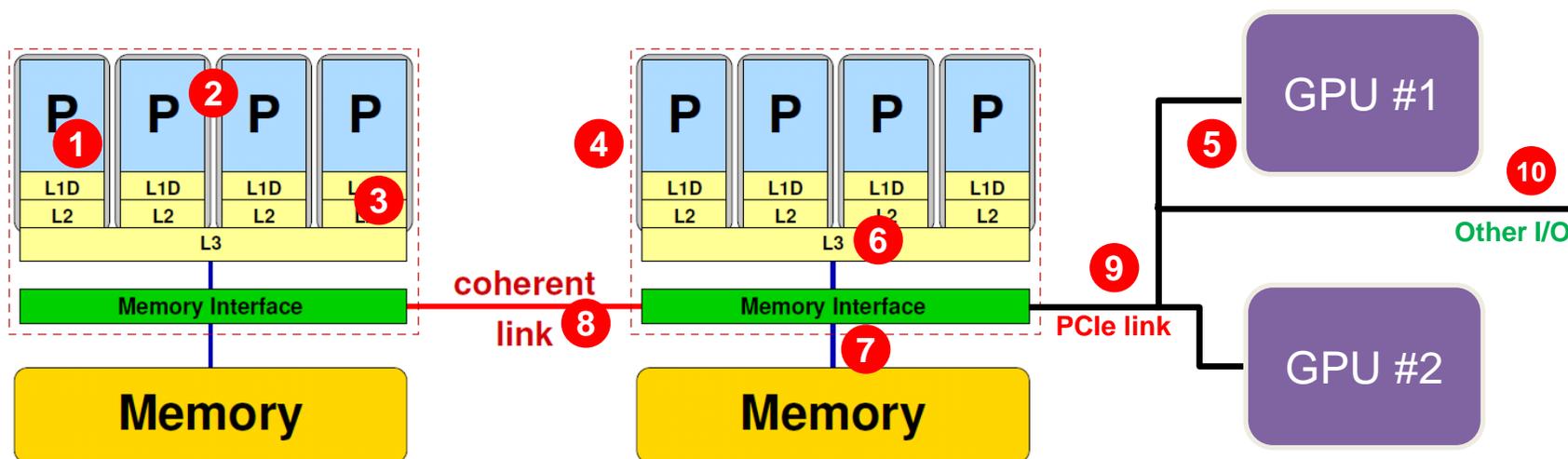
- Execution/SIMD units (1)
- Cores (2)
- Inner cache levels (3)
- Sockets / ccNUMA domains (4)
- Multiple accelerators (5)

Shared resources:

- Outer cache level per socket (6)
- Memory bus per socket (7)
- Intersocket link (8)
- PCIe bus(es) (9)

Parallelism in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

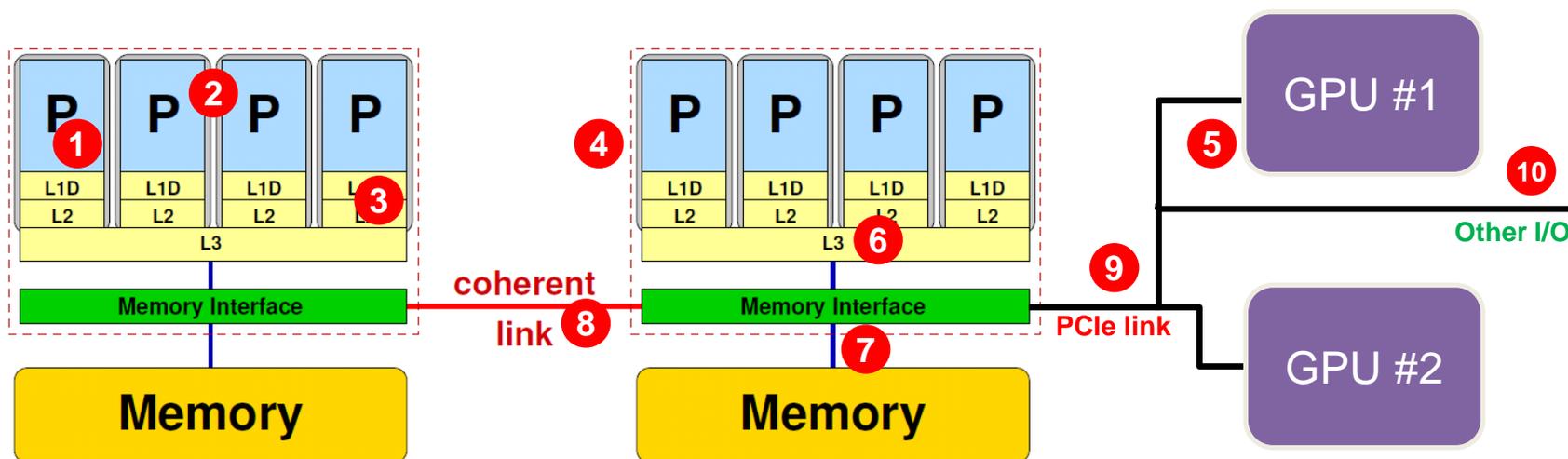
- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

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

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?

GPGPU accelerators

NVIDIA “Hopper” H100

vs.

AMD Zen4 “Genoa”



Nvidia H100 “Hopper” SXM5 specs

Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “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)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)



Nvidia H100 “Hopper” SXM5 specs

Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “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)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)



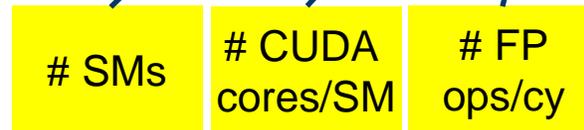
Nvidia H100 “Hopper” SXM5 specs

Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “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)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)



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



Nvidia H100 “Hopper” SXM5 specs

Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “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)
- 50 MiB L2 Cache
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)



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

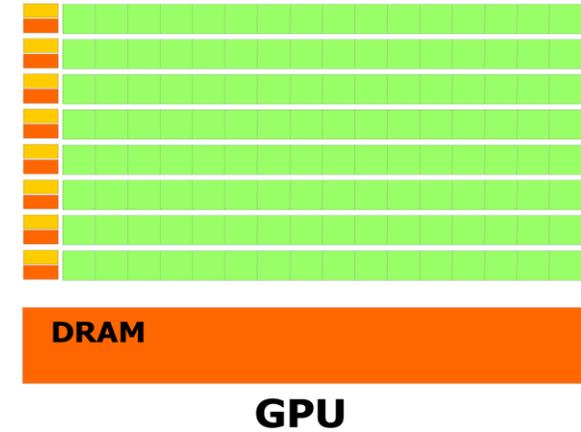
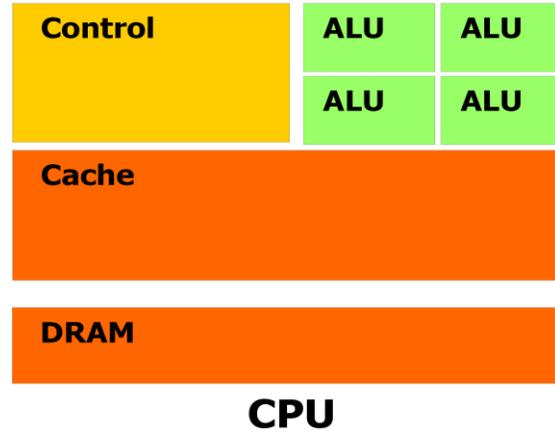
SMs # CUDA cores/SM # FP ops/cy

$n_{SM} = 144$
 $n_{core} = 64$
 $n_{FP} = 2 \frac{\text{flops}}{\text{cy}}$
 $f = 1.8 \frac{\text{Gcy}}{\text{s}}$

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	144 SMs @ ~1.8 GHz
FP32 Performance/core	76.8 GFlop/s	~ 230 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

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)
- **Exploiting performance: parallelism + bottleneck awareness**
 - “High Performance Computing” == computing at a bottleneck
- **Performance of programs** is sensitive to architecture

Multicore Performance and Tools

Part 1: Topology, affinity control, clock speed



Tools for Node-level Performance Engineering

- **Node Information**
`/proc/cpuinfo`, `numactl`, `hwloc`, `likwid-topology`, `likwid-powermeter`
- **Affinity control and data placement**
OpenMP and MPI runtime environments, `hwloc`, `numactl`, `likwid-pin`
- **Runtime Profiling**
Compilers, `gprof`, `perf`, `HPCToolkit`, Intel Amplifier, `gprof-ng`, ...
- **Performance Analysis**
Intel VTune, `likwid-perfctr`, PAPI-based tools, `HPCToolkit`, `perf`
- **Microbenchmarking**
`STREAM`, `likwid-bench`, `lmbench`, `uarch-bench`

Reporting topology

likwid-topology



<https://youtu.be/mxMWjNe73SI>

Output of `likwid-topology`

optional

```
$ likwid-topology
```

```
-----  
CPU name: Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz
```

```
CPU type: Intel Icelake SP processor
```

```
CPU stepping: 6
```

```
*****
```

Hardware Thread Topology

```
*****
```

```
Sockets:      2
```

```
Cores per socket: 36
```

```
Threads per core: 1
```

```
-----  
HWThread      Thread      Core      Die      Socket      Available
```

HWThread	Thread	Core	Die	Socket	Available
0	0	0	0	0	*
1	0	1	0	0	*
2	0	2	0	0	*
[...]					
69	0	69	0	1	*
70	0	70	0	1	*
71	0	71	0	1	*

All physical processor IDs

```
-----  
Socket 0:      ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 23 24 25 26 27 28 29 30 31 32 33 34 35 )
```

```
Socket 1:      ( 36 37 38 39 40 41 42 43 44 45 46 47 48 ... 59 60 61 62 63 64 65 66 67 68 69 70 71 )
```

Output of `likwid-topology`

optional

```
*****
Cache Topology
*****
Level:          1
Size:           48 kB
Cache groups:   ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ... ( 64 ) ( 65 ) ( 66 ) ( 67 ) ( 68 ) ( 69 ) ( 70 ) ( 71 )
-----
Level:          2
Size:           1.25 MB
Cache groups:   ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ... ( 64 ) ( 65 ) ( 66 ) ( 67 ) ( 68 ) ( 69 ) ( 70 ) ( 71 )
-----
Level:          3
Size:           54 MB
Type:           Unified cache
Associativity:  12
Number of sets: 73728
Cache line size: 64
Cache type:     Non Inclusive
Shared by threads: 36
Cache groups:   ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 23 24 25 26 27 28 29 30 31 32 33 34 35 )
                ( 36 37 38 39 40 41 42 43 44 45 46 47 48 ... 59 60 61 62 63 64 65 66 67 68 69 70 71 )
-----
```

Additional cache info
with `-c` option

Output of `likwid-topology`

optional

```
*****
NUMA Topology
*****
NUMA domains:          4 ←
-----
Domain:                0
Processors:            ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 )
Distances:             10 11 20 20
Free memory:           119059 MB
Total memory:          128553 MB
-----
Domain:                1
Processors:            ( 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 )
Distances:             11 10 20 20
Free memory:           128196 MB
Total memory:          129020 MB
-----
Domain:                2
Processors:            ( 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 )
Distances:             20 20 10 11
Free memory:           128033 MB
Total memory:          128978 MB
-----
Domain:                3
Processors:            ( 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 )
Distances:             20 20 11 10
Free memory:           128719 MB
Total memory:          129017 MB
-----
```

Output similar to
`numactl --hardware`

Sockets: 2
Threads per core: 1
Sub-NUMA clustering (SNC)
enabled, SMT disabled!

Enforcing thread/process affinity under Linux OS

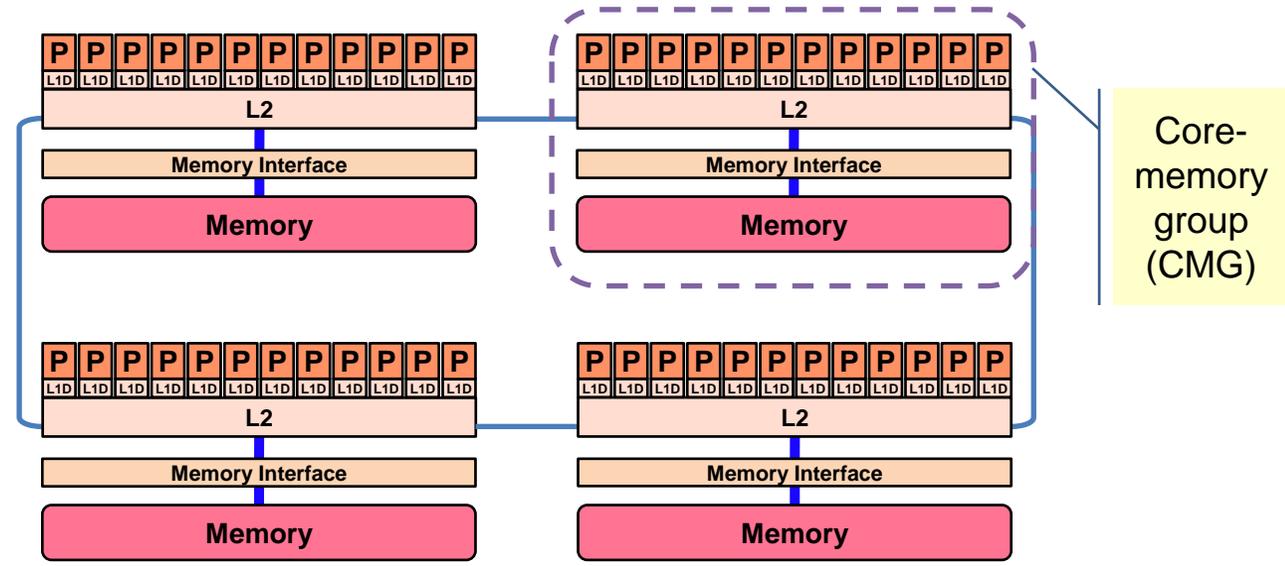
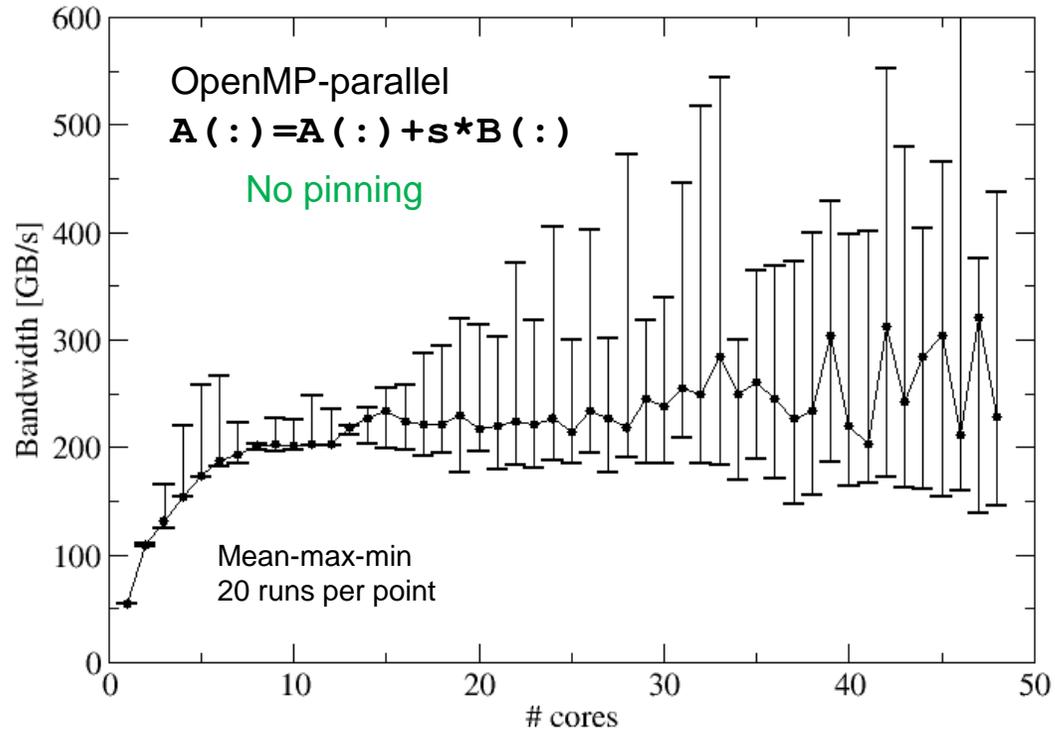
likwid-pin



<https://youtu.be/PSJKNQaqwB0>

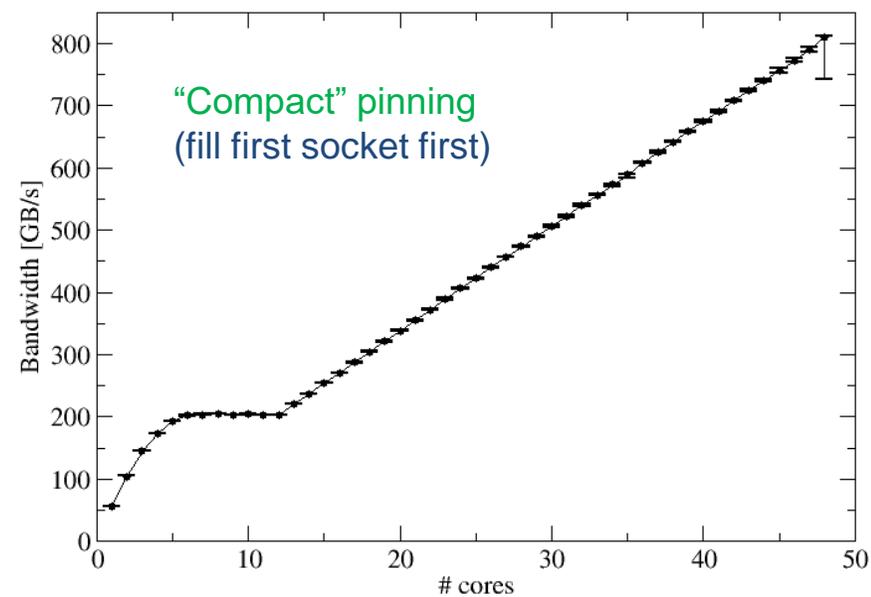
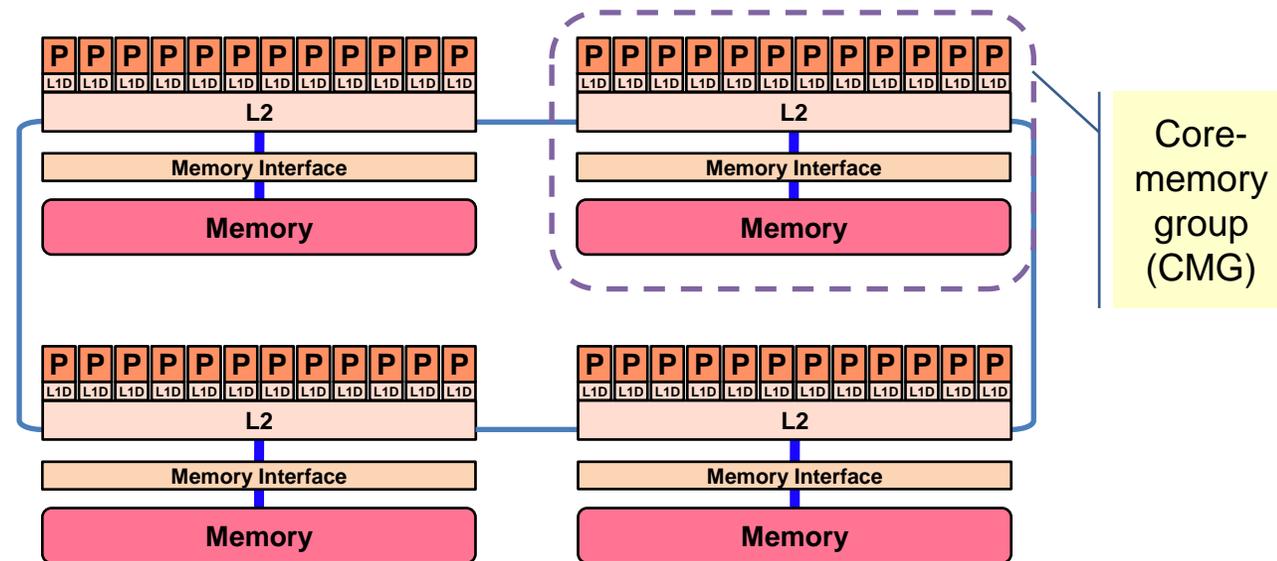
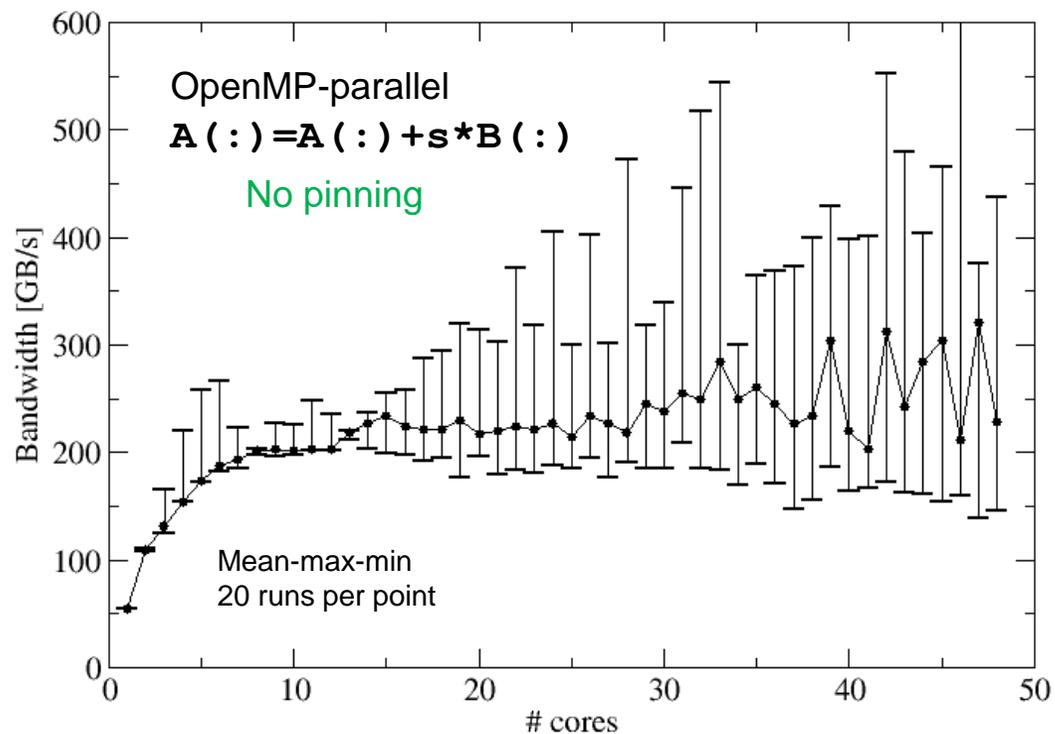
DAXPY test on A64FX

Anarchy vs. thread pinning



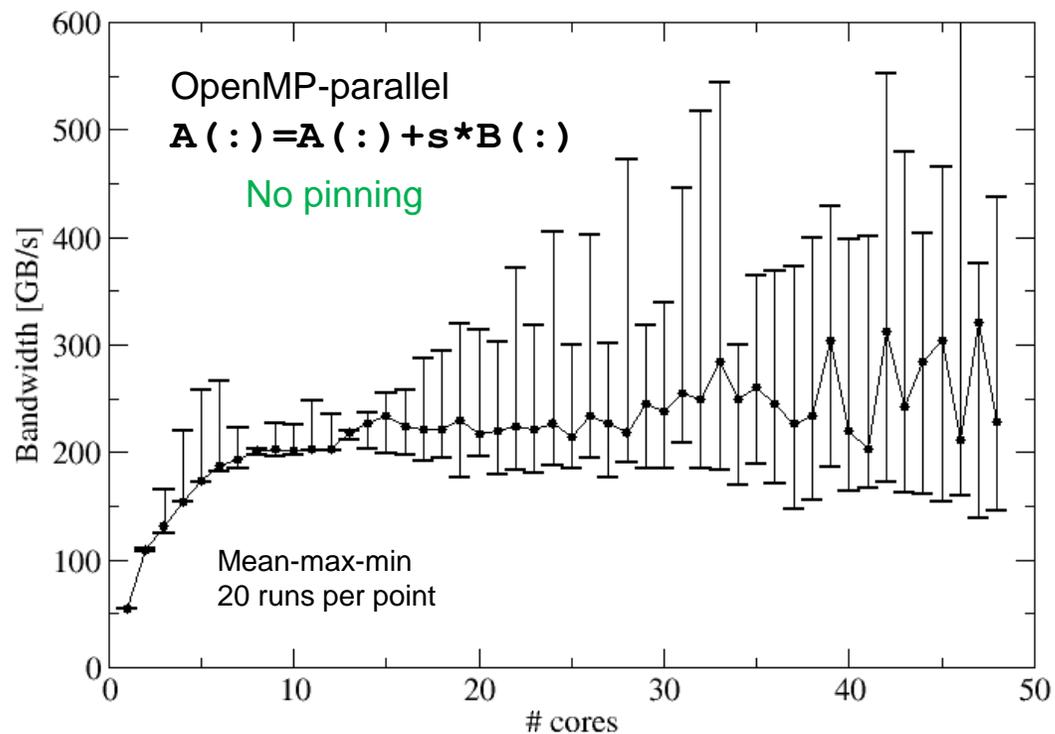
DAXPY test on A64FX

Anarchy vs. thread pinning



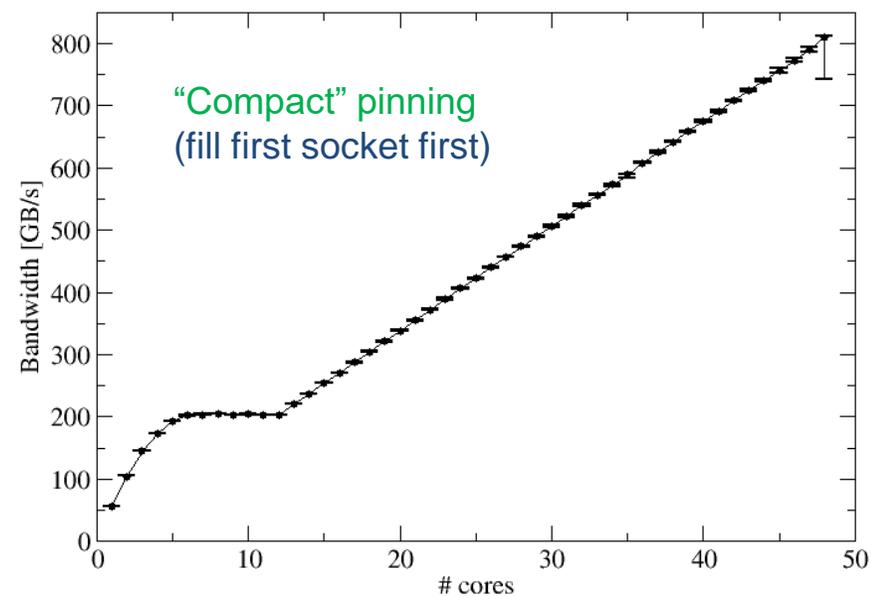
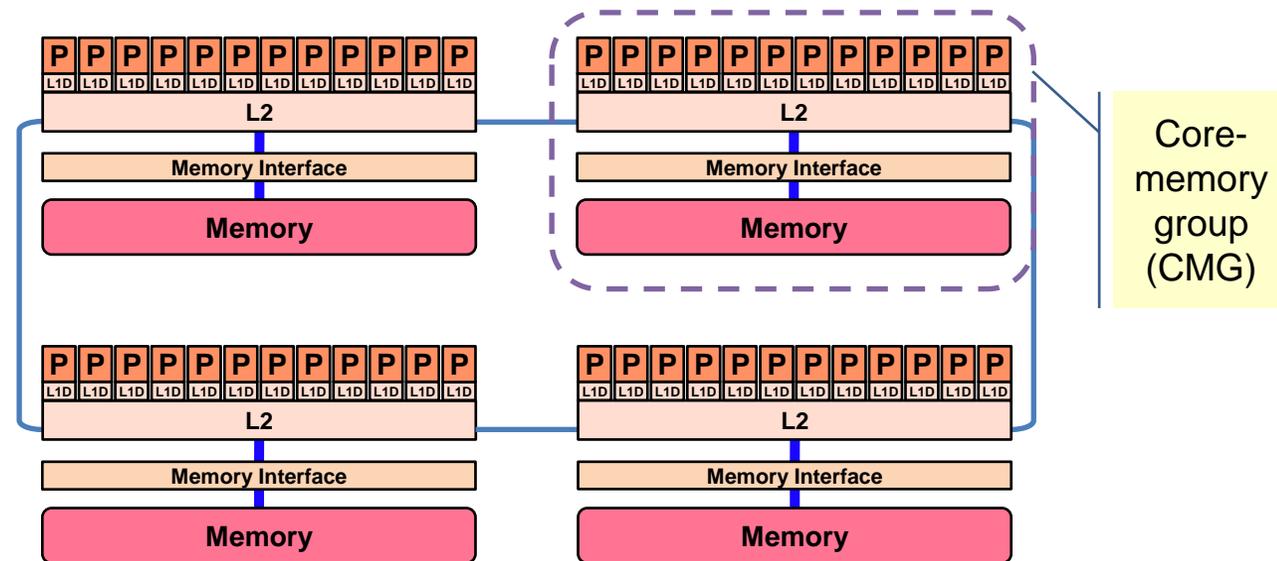
DAXPY test on A64FX

Anarchy vs. thread pinning



There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention



More thread/process affinity (“pinning”) options

- Highly OS-dependent system calls but available on all systems
 - Linux: `sched_setaffinity()`
 - Windows: `SetThreadAffinityMask()`
- Hwloc project (<http://www.open-mpi.de/projects/hwloc/>)
- Support for “semi-automatic” pinning
 - All modern compilers with OpenMP support
OpenMP 4.0 (`OMP_PLACES`, `OMP_PROC_BIND`)
 - CPUset reduction utils: `taskset` or `numactl`
 - Job scheduler like **SLURM**
- Affinity awareness in MPI libraries (OpenMPI, Intel MPI, ...)
- Or **likwid-pin** and **likwid-mpirun**

 <https://youtu.be/IKW0kRLnhyc>

Overview `likwid-pin`

- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library
→ **binary must be dynamically linked!**
- Supports **logical core numbering** within topological entities (thread domains)

- Simple usage with physical (kernel) core IDs:

```
$ likwid-pin -c 0-3,4,6 ./myApp parameters
```

```
$ OMP_NUM_THREADS=4 likwid-pin -c 0-9 ./myApp params
```

- Simple usage with logical IDs (“thread groups expressions”):

```
$ likwid-pin -c S0:0-7 ./myApp params
```

```
$ likwid-pin -c C1:0-2 ./myApp params
```

LIKWID terminology: Thread group syntax

- The OS numbers all processors (hardware threads) on a node
- The numbering is enforced at boot time by the BIOS
- LIKWID introduces **thread domains** consisting of hardware threads sharing a topological entity (e.g. socket or shared cache)
- A **thread domain** is defined by a **single character + index**

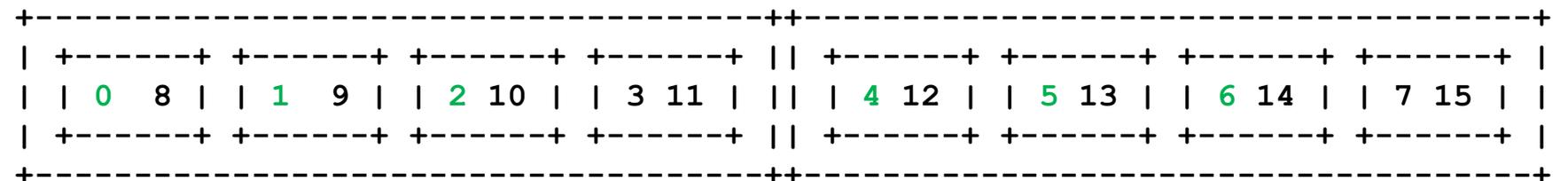
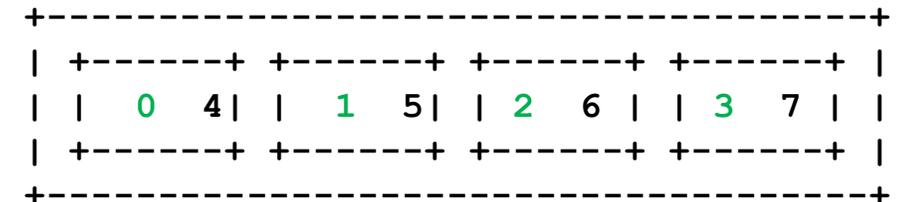
- Example for likwid-pin:

```
$ likwid-pin -c S0:0-3 ./a.out
```

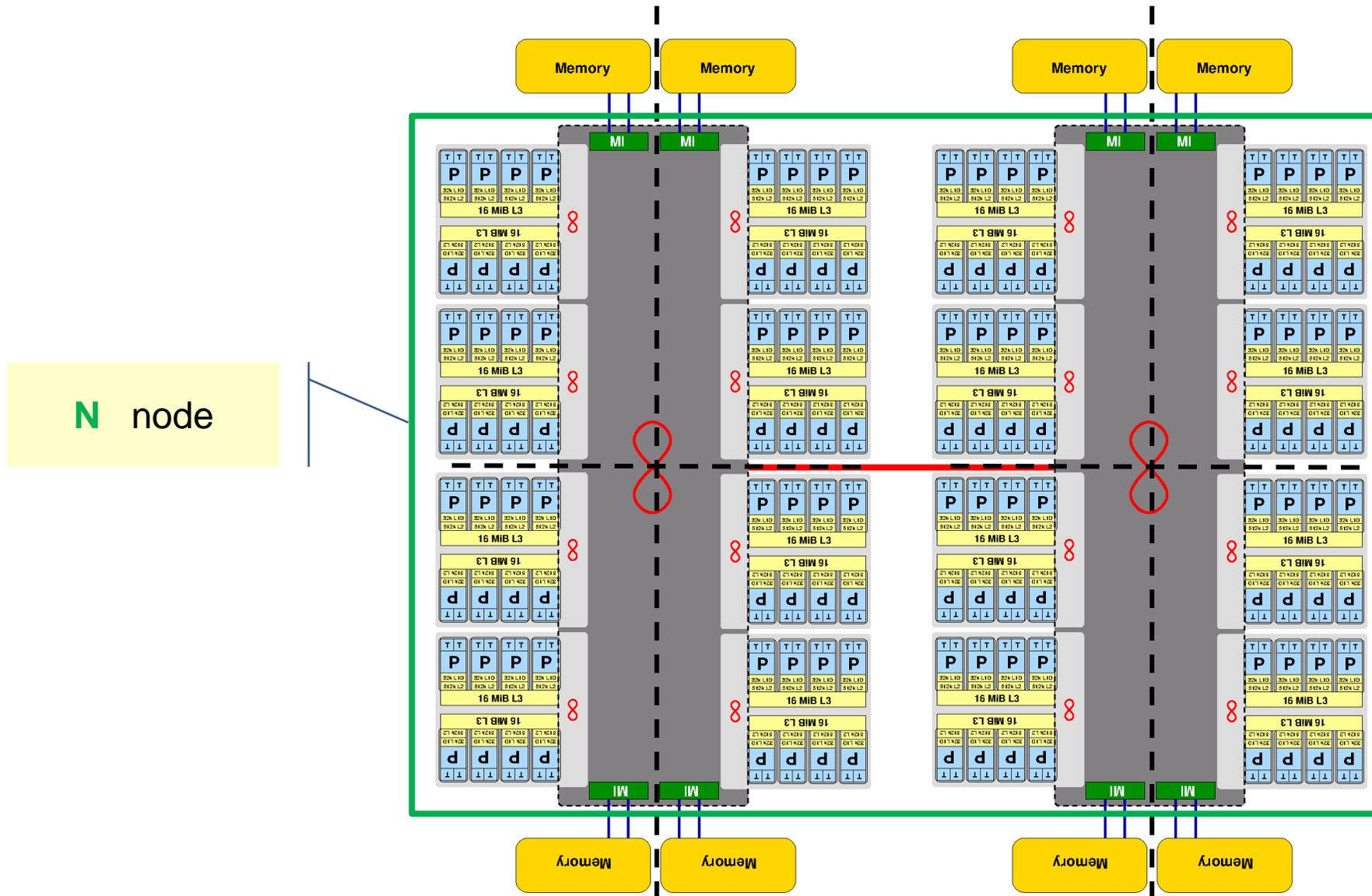
- Thread group expressions may be chained with @:

```
$ likwid-pin -c S0:0-2@S1:0-2 ./a.out
```

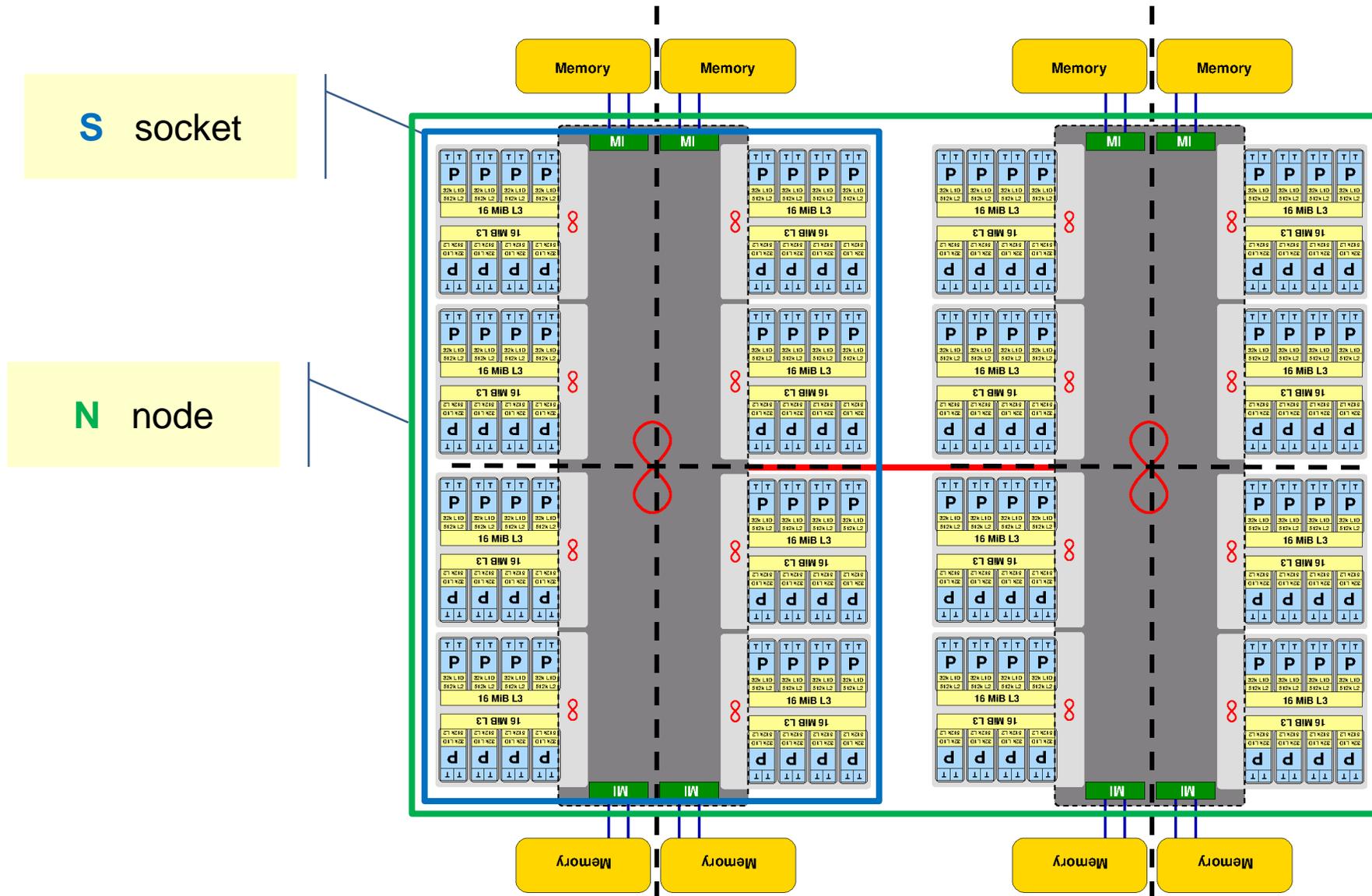
Physical cores first!



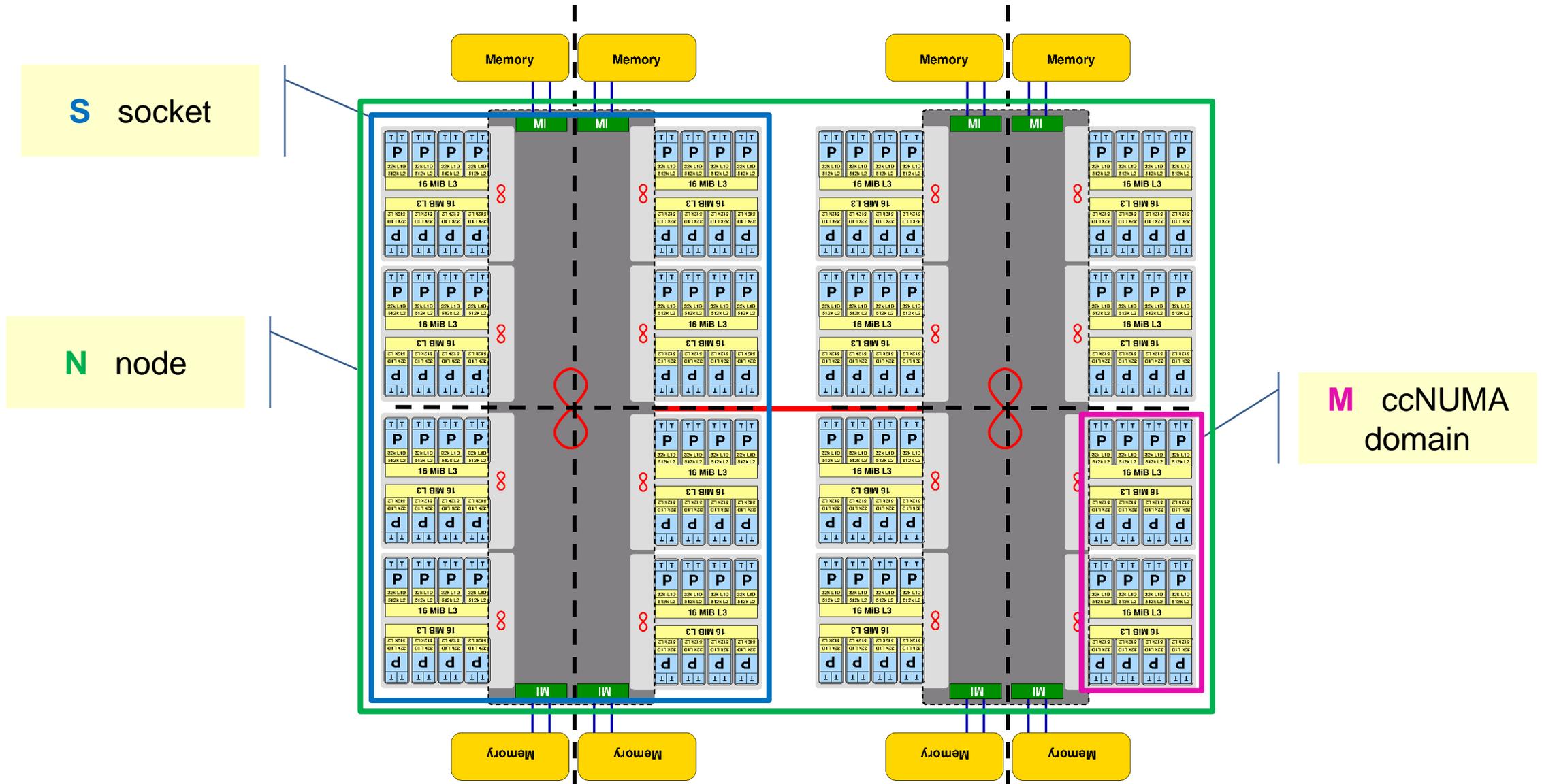
Available thread domains/unit prefixes (LIKWID 5.2)



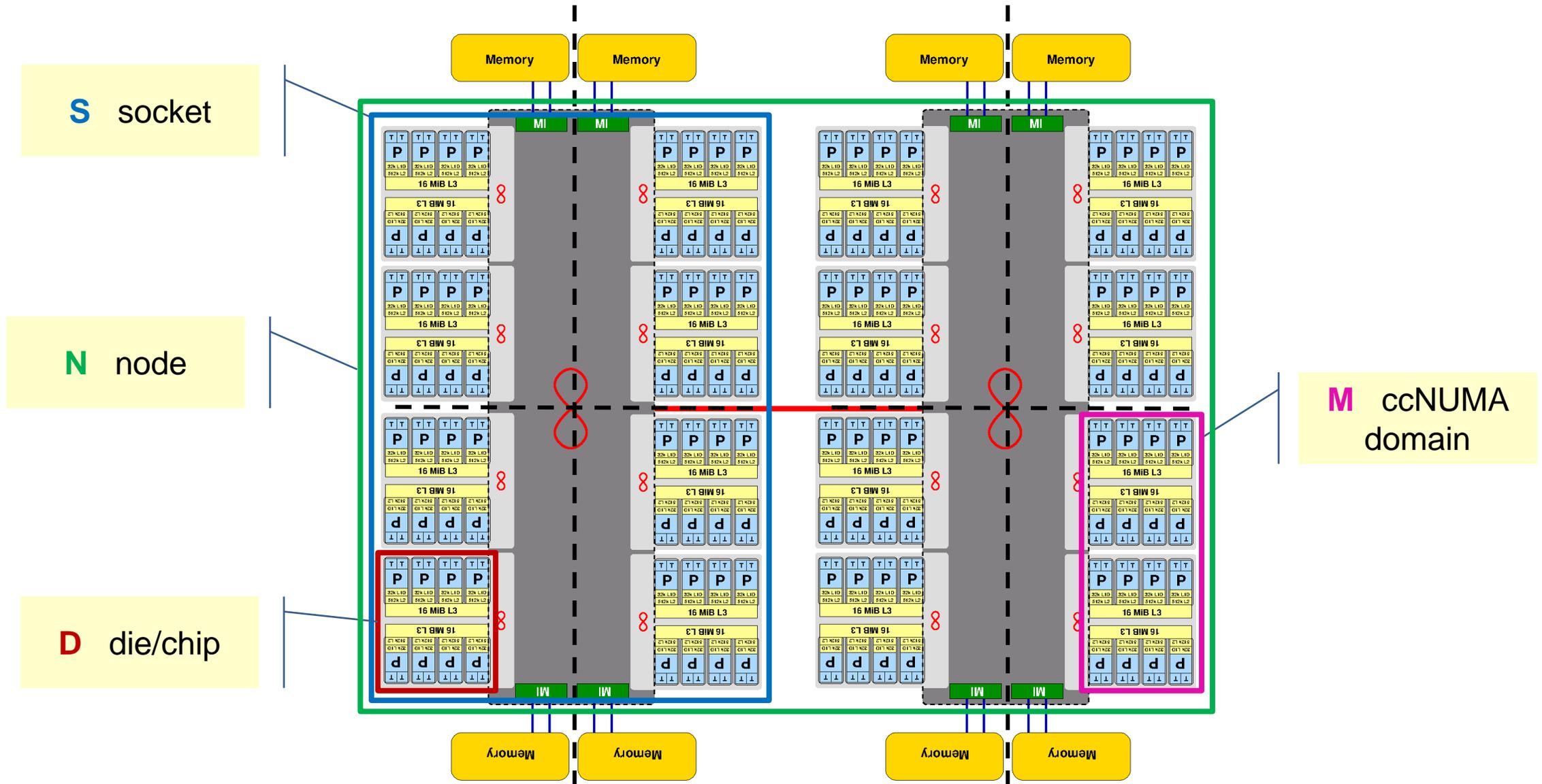
Available thread domains/unit prefixes (LIKWID 5.2)



Available thread domains/unit prefixes (LIKWID 5.2)



Available thread domains/unit prefixes (LIKWID 5.2)



Example: `likwid-pin` with Intel OpenMP

Running the STREAM benchmark with `likwid-pin`:

```
$ likwid-pin -c S0:0-3 ./stream
```

```
-----  
Double precision appears to have 16 digits of accuracy  
Assuming 8 bytes per DOUBLE PRECISION word  
-----
```

```
Array size = 20000000  
Offset = 32  
The total memory requirement is 457 MB  
You are running each test 10 times  
--
```

```
The *best* time for each test is used  
*EXCLUDING* the first and last iterations
```

```
[pthread wrapper]
```

```
[pthread wrapper] MAIN -> 0
```

```
[pthread wrapper] PIN_MASK: 0->1 1->2 2->3
```

```
[pthread wrapper] SKIP MASK: 0x0
```

```
threadid 47308666070912 -> core 1 - OK
```

```
threadid 47308670273536 -> core 2 - OK
```

```
threadid 47308674476160 -> core 3 - OK
```

```
[... rest of STREAM output omitted ...]
```

Main PID always pinned

Some threads might need to be skipped (e.g.runtime threads)

Pin all spawned threads in turn

OMP_PLACES and Thread Affinity

optional

Processor: smallest entity able to run a thread or task (hardware thread)

Place: one or more processors → thread pinning is done place by place

Free migration of the threads on a place between the processors of that place.

Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"`
- `OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"`
- `OMP_PLACES="{0:4}:8:4"`

OMP_PLACES and Thread Affinity

optional

Processor: smallest entity able to run a thread or task (hardware thread)

Place: one or more processors → thread pinning is done place by place

Free migration of the threads on a place between the processors of that place.

OMP_PLACES	Place ==
threads	Hardware thread (hyper-thread)
cores	All HW threads of a single core
sockets	All HW threads of a socket
abstract_name(num_places)	Restrict # of places available

Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"`
- `OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"`
- `OMP_PLACES="{0:4}:8:4"`

OMP_PLACES and Thread Affinity

optional

Processor: smallest entity able to run a thread or task (hardware thread)

Place: one or more processors → thread pinning is done place by place

Free migration of the threads on a place between the processors of that place.

OMP_PLACES	Place ==
threads	Hardware thread (hyper-thread)
cores	All HW threads of a single core
sockets	All HW threads of a socket
abstract_name (num_places)	Restrict # of places available

abstract name

Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"`
- `OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"`
- `OMP_PLACES="{0:4}:8:4"`

`<lower-bound>:<number of entries>[:<stride>]`

OMP_PLACES and Thread Affinity

optional

Processor: smallest entity able to run a thread or task (hardware thread)

Place: one or more processors → thread pinning is done place by place

Free migration of the threads on a place between the processors of that place.

OMP_PLACES	Place ==
threads	Hardware thread (hyper-thread)
cores	All HW threads of a single core
sockets	All HW threads of a socket
abstract_name(num_places)	Restrict # of places available

abstract name

Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"`
- `OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"`
- `OMP_PLACES="{0:4}:8:4"`

<lower-bound>:<number of entries>[:<stride>]

Caveat: Actual behavior is implementation defined!

OMP_PROC_BIND variable / proc_bind() clause

optional

Determines how places are used for pinning:

OMP_PROC_BIND	Meaning
FALSE	Affinity disabled
TRUE	Affinity enabled, implementation defined strategy
CLOSE	Threads bind to consecutive places
SPREAD	Threads are evenly scattered among places
MASTER	Threads bind to the same place as the master thread that was running before the parallel region was entered

If there are more threads than places, consecutive threads are put into individual places (“balanced”)

Some simple OMP_PLACES examples

optional

Intel Xeon w/ SMT, 2x10 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=10  
OMP_PLACES=cores  
OMP_PROC_BIND=close
```

Always prefer abstract places
instead of HW thread IDs!

Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)

```
OMP_NUM_THREADS=8  
OMP_PLACES=sockets  
OMP_PROC_BIND=close      # spread will also do
```

Intel Xeon, 2 sockets, 4 threads per socket, binding to cores

```
OMP_NUM_THREADS=8  
OMP_PLACES=cores  
OMP_PROC_BIND=spread
```

MPI startup and hybrid pinning: `likwid-mpirun`

- How do you manage **affinity with MPI or hybrid MPI/threading?**
- In the long run a unified standard is needed
- Till then, `likwid-mpirun` provides a portable/flexible solution
- The examples here are for Intel MPI/OpenMP programs, but are also applicable to other threading models

Pure MPI:

```
$ likwid-mpirun -np 16 -nperdomain S:2 ./a.out
```

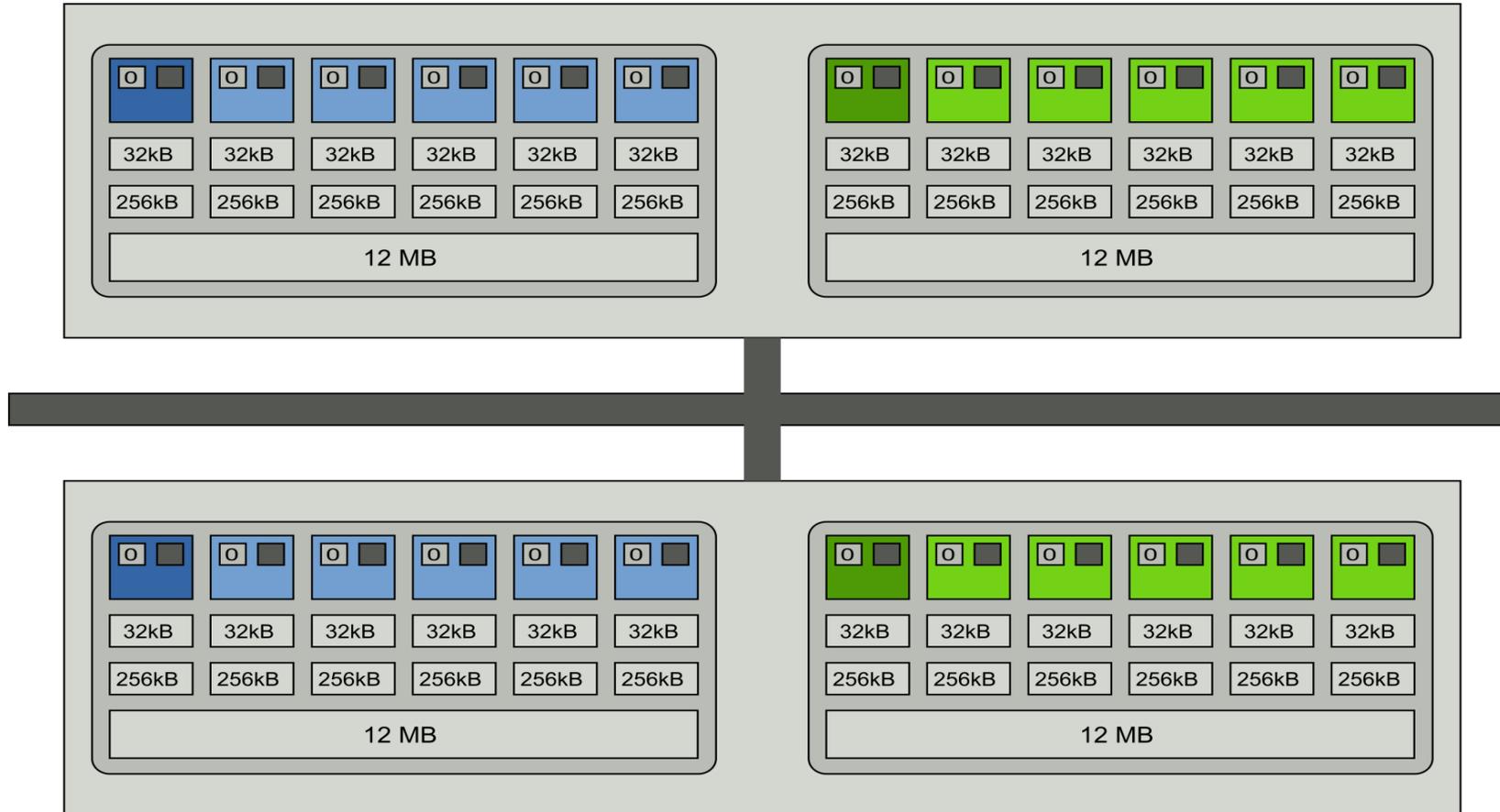
Hybrid:

```
$ likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
```

likwid-mpirun 1 MPI process per socket

```
$ likwid-mpirun -np 4 -pin S0:0-5_S1:0-5 ./a.out  
$ likwid-mpirun -np 4 -nperdomain S:1 6 ./a.out
```

optional



Intel MPI + compiler:

```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

Microbenchmarking for architectural exploration

Probing of the memory hierarchy

Saturation effects

OpenMP barrier overhead



Motivation for Microbenchmarking as a tool

- Isolate small kernels to:
 - Separate influences
 - Determine specific machine capabilities (light speed)
 - Gain experience about software/hardware interaction
 - Determine programming model overhead
 - ...
- Possibilities:
 - Readymade benchmark collections (epcc OpenMP, IMB)
 - STREAM benchmark for memory bandwidth
 - Implement own benchmarks (difficult and error prone)
 - **likwid-bench** tool: Offers collection of benchmarks and framework for rapid development of assembly code kernels

likwid-bench

- Microbenchmarking in high-level language is often difficult
- Solution: assembly-based **microbenchmarking framework**
 - e.g., **likwid-bench**

```
$ likwid-bench -t triad_avx512_fma -W S0:28kB:1
```

benchmark type

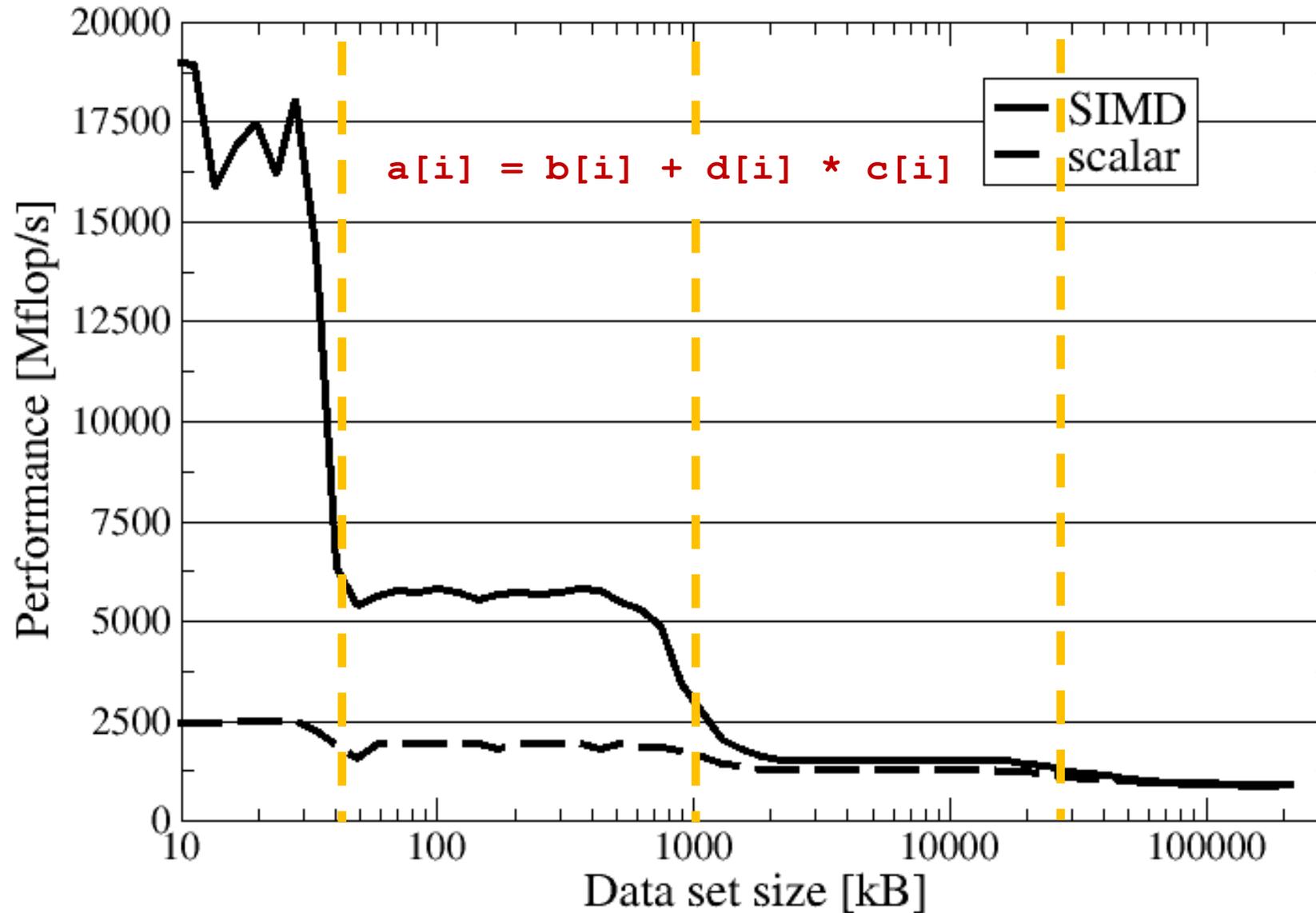
topological entity (see likwid-pin)

working set

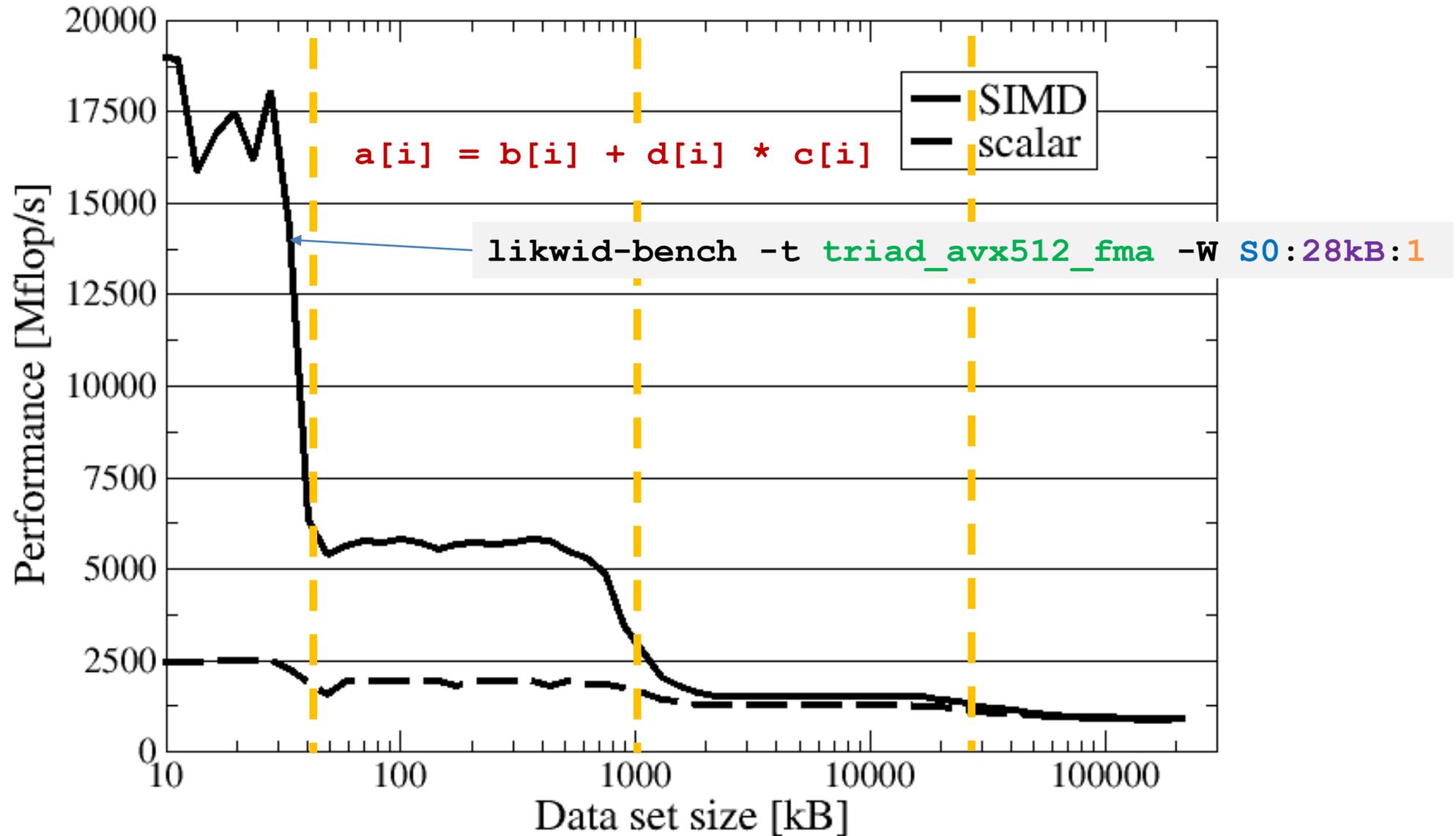
of threads

- Classic benchmark: Schönauer Triad $a[i] = b[i] + d[i] * c[i]$
This kernel is limited by data transfer performance for all memory levels on all architectures, ever!

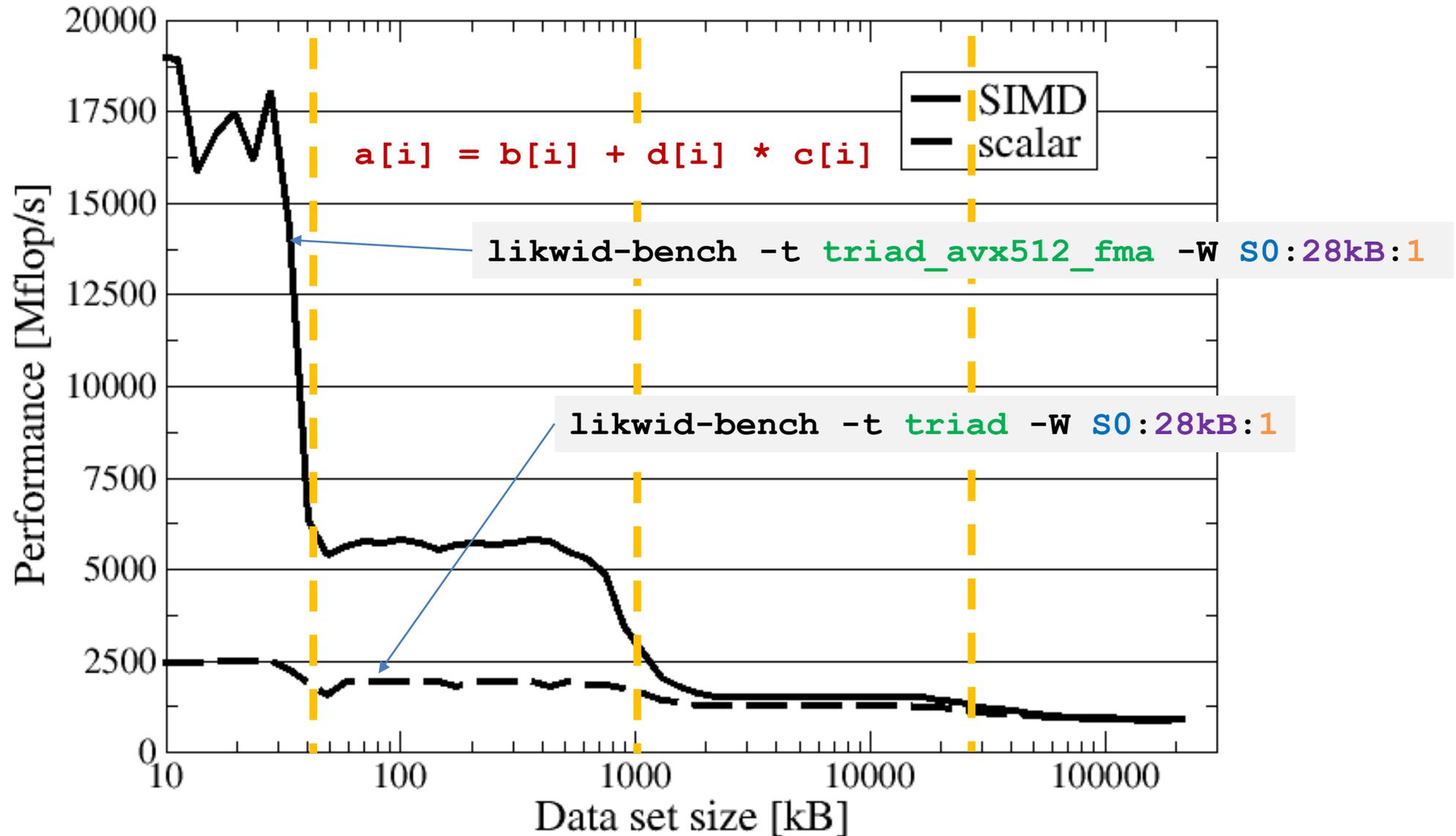
Schönauer triad on one CascadeLake core 2.5GHz



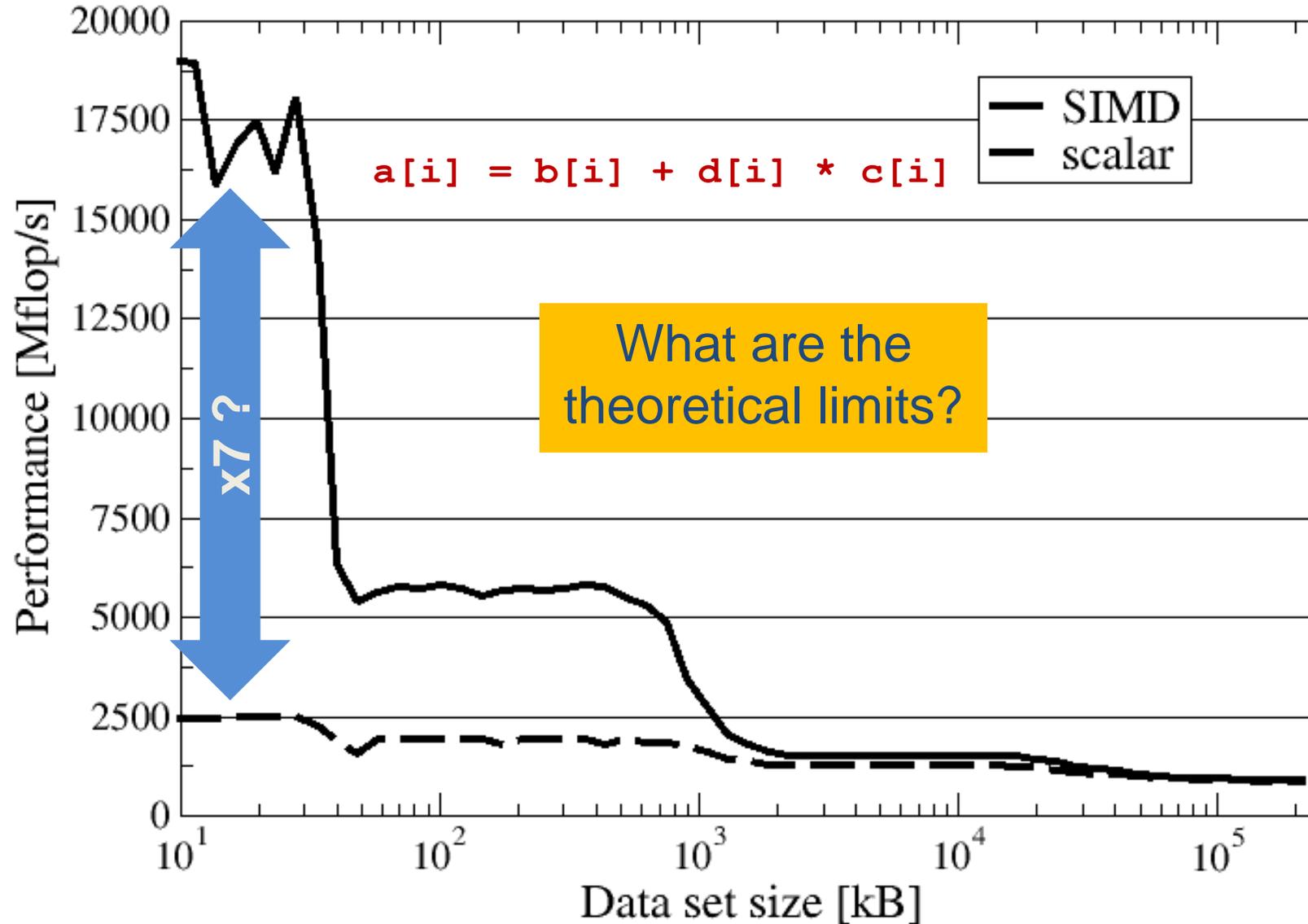
Schönauer triad on one CascadeLake core 2.5GHz



Schönauer triad on one CascadeLake core 2.5GHz



Schönauer triad on one CascadeLake core 2.5GHz



Throughput triad on one CascadeLake node (2.5 GHz)

- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma  
-W S0:$size:$threads:1:2
```

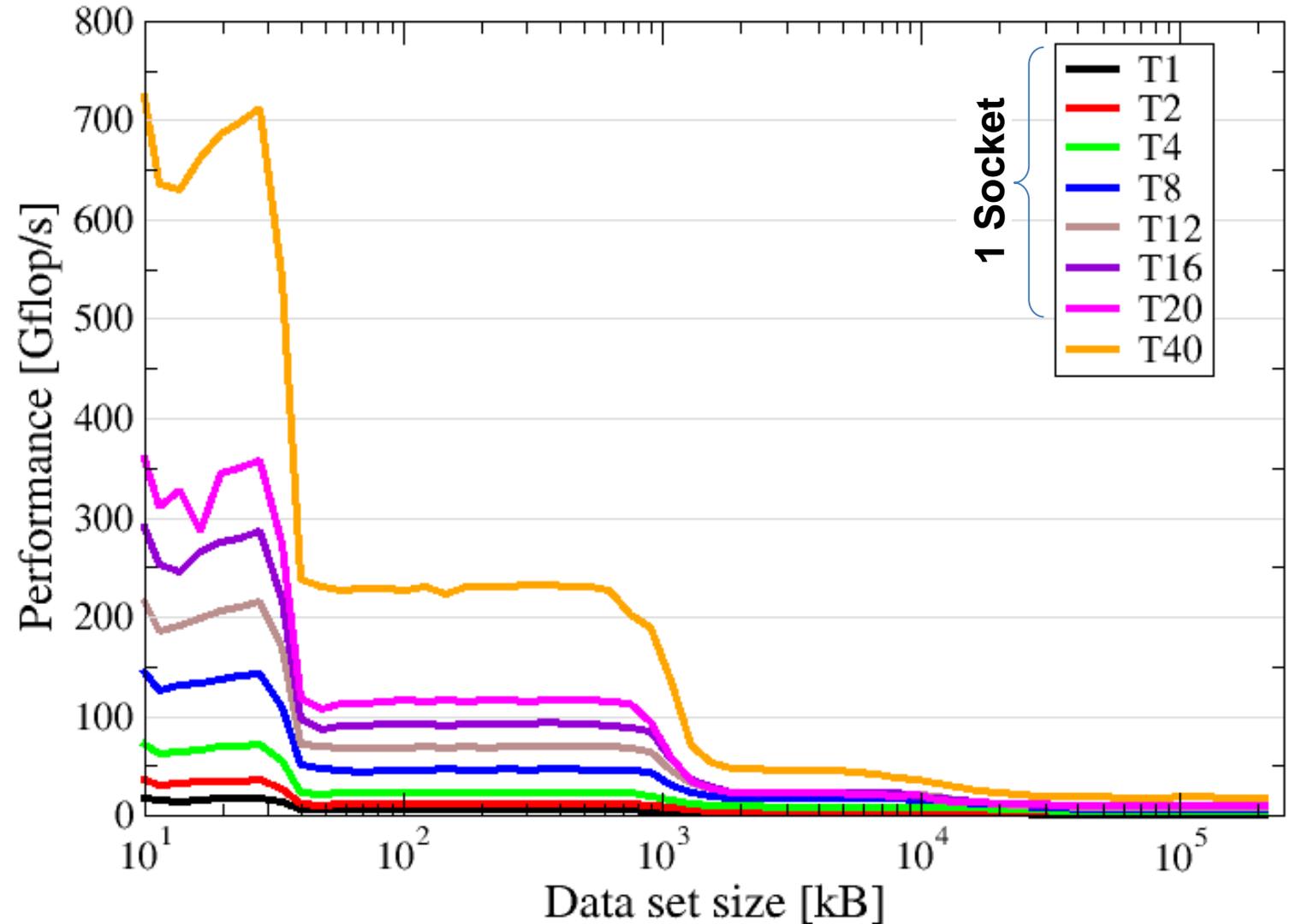
- Scan `$size` and `$threads`
- Pin threads in `chunks of 1` with `distance of 2` (skip SMT threads)

Throughput triad on one CascadeLake node (2.5 GHz)

- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma \  
-W S0:$size:$threads:1:2
```

- Scan `$size` and `$threads`
- Pin threads in `chunks of 1` with `distance of 2` (skip SMT threads)

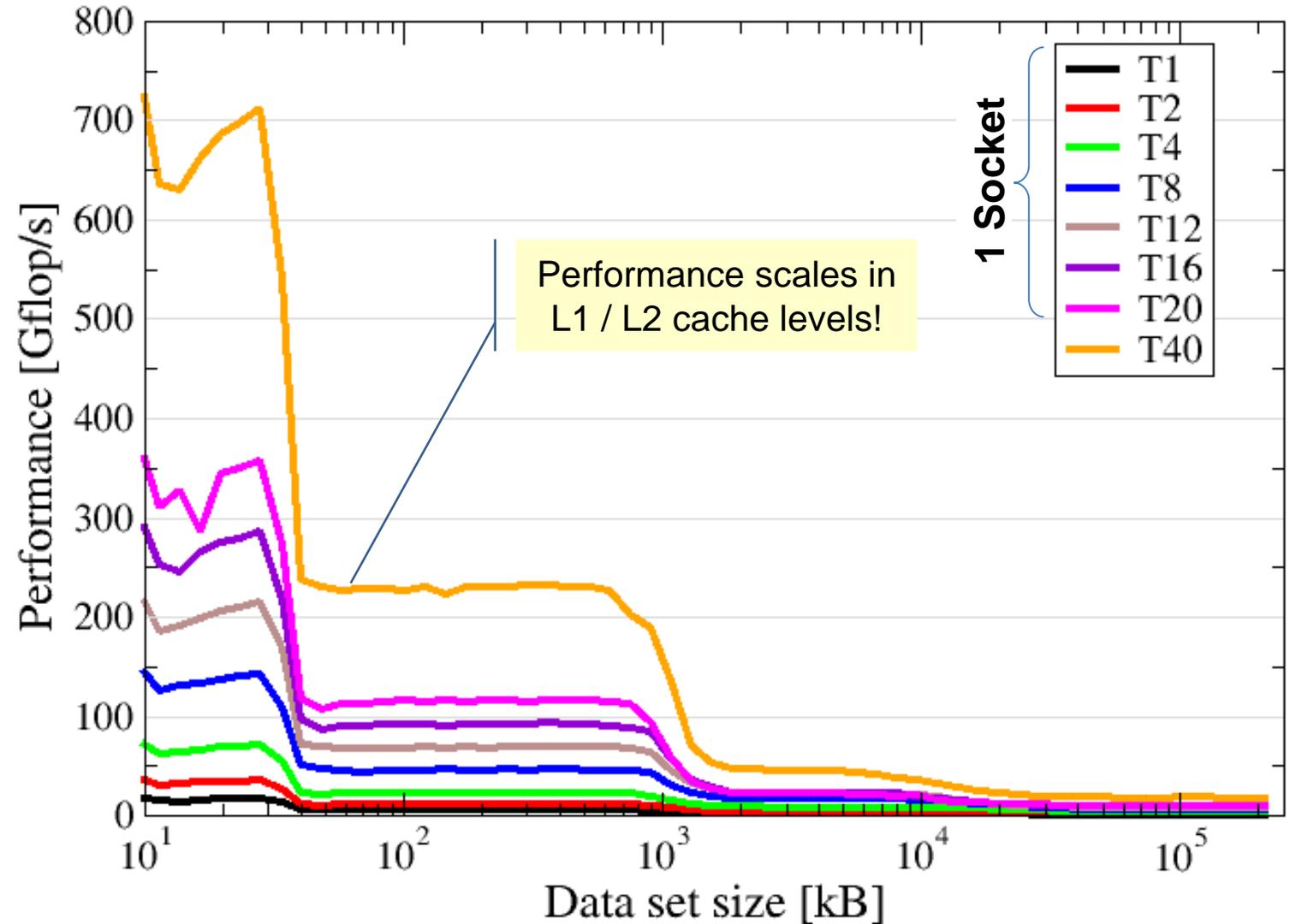


Throughput triad on one CascadeLake node (2.5 GHz)

- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma \  
-W S0:$size:$threads:1:2
```

- Scan **\$size** and **\$threads**
- Pin threads in **chunks of 1** with **distance of 2** (skip SMT threads)

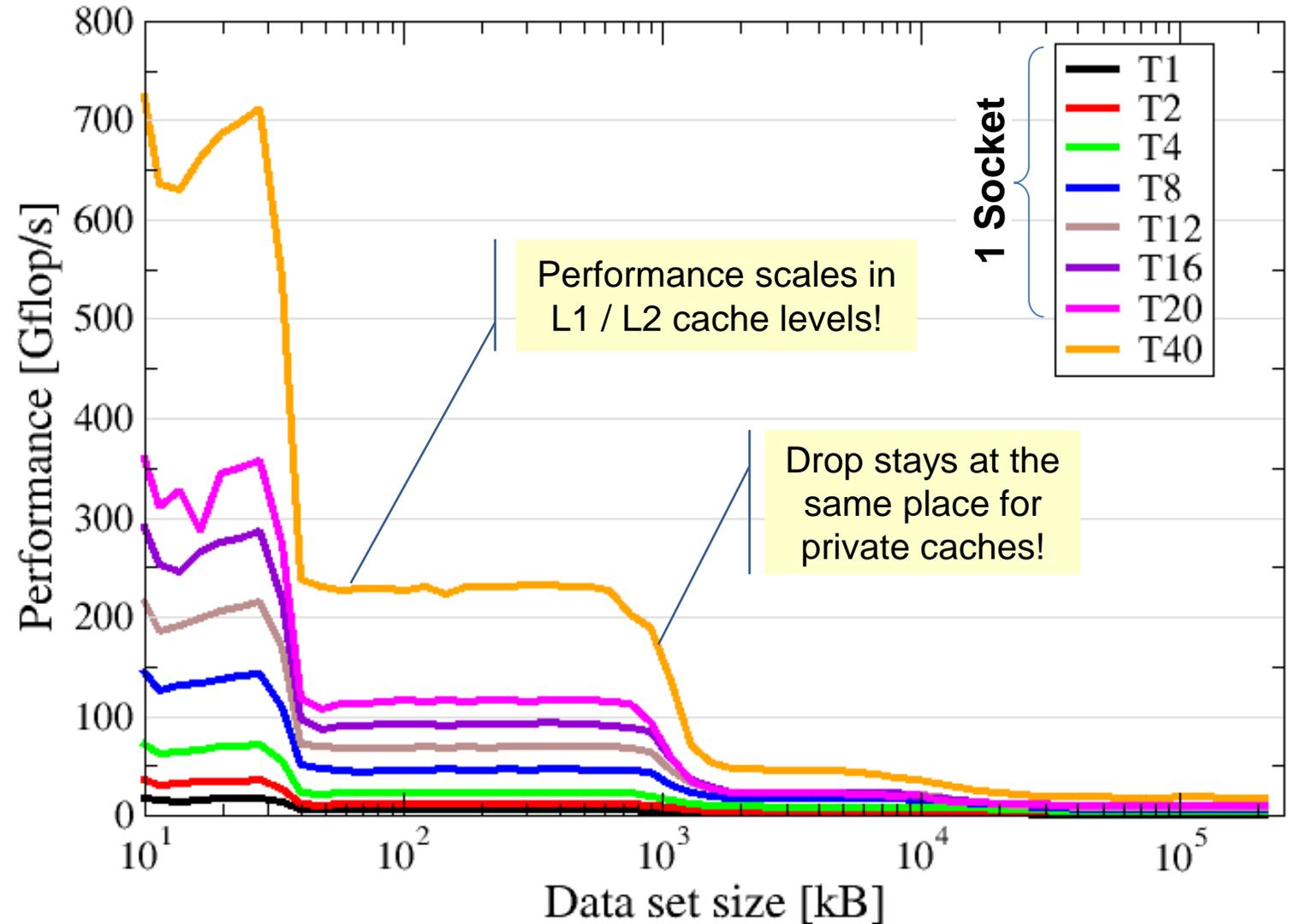


Throughput triad on one CascadeLake node (2.5 GHz)

- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma \  
-W S0:$size:$threads:1:2
```

- Scan **\$size** and **\$threads**
- Pin threads in **chunks of 1** with **distance of 2** (skip SMT threads)

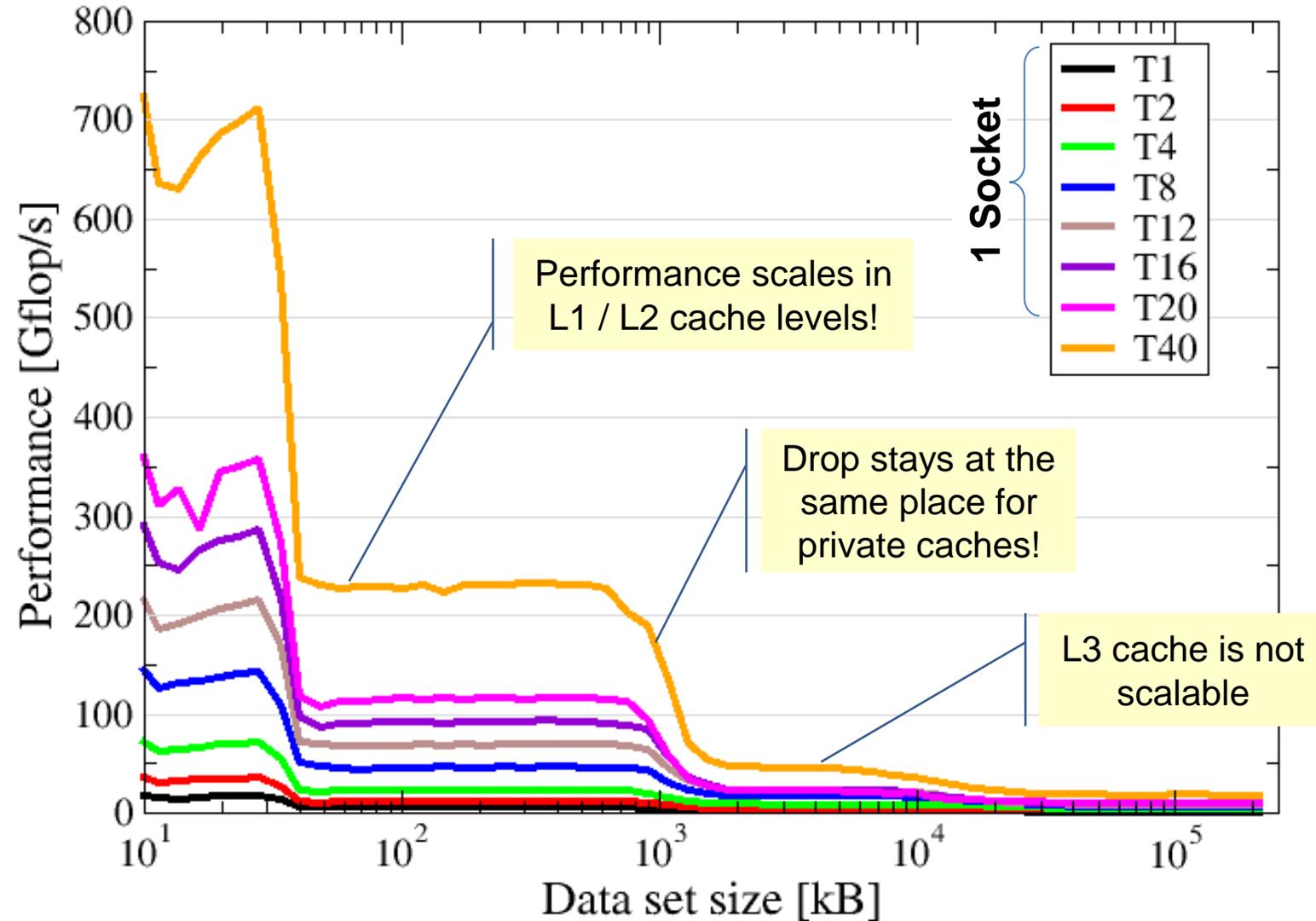


Throughput triad on one CascadeLake node (2.5 GHz)

- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma \  
-W S0:$size:$threads:1:2
```

- Scan **\$size** and **\$threads**
- Pin threads in **chunks of 1** with **distance of 2** (skip SMT threads)

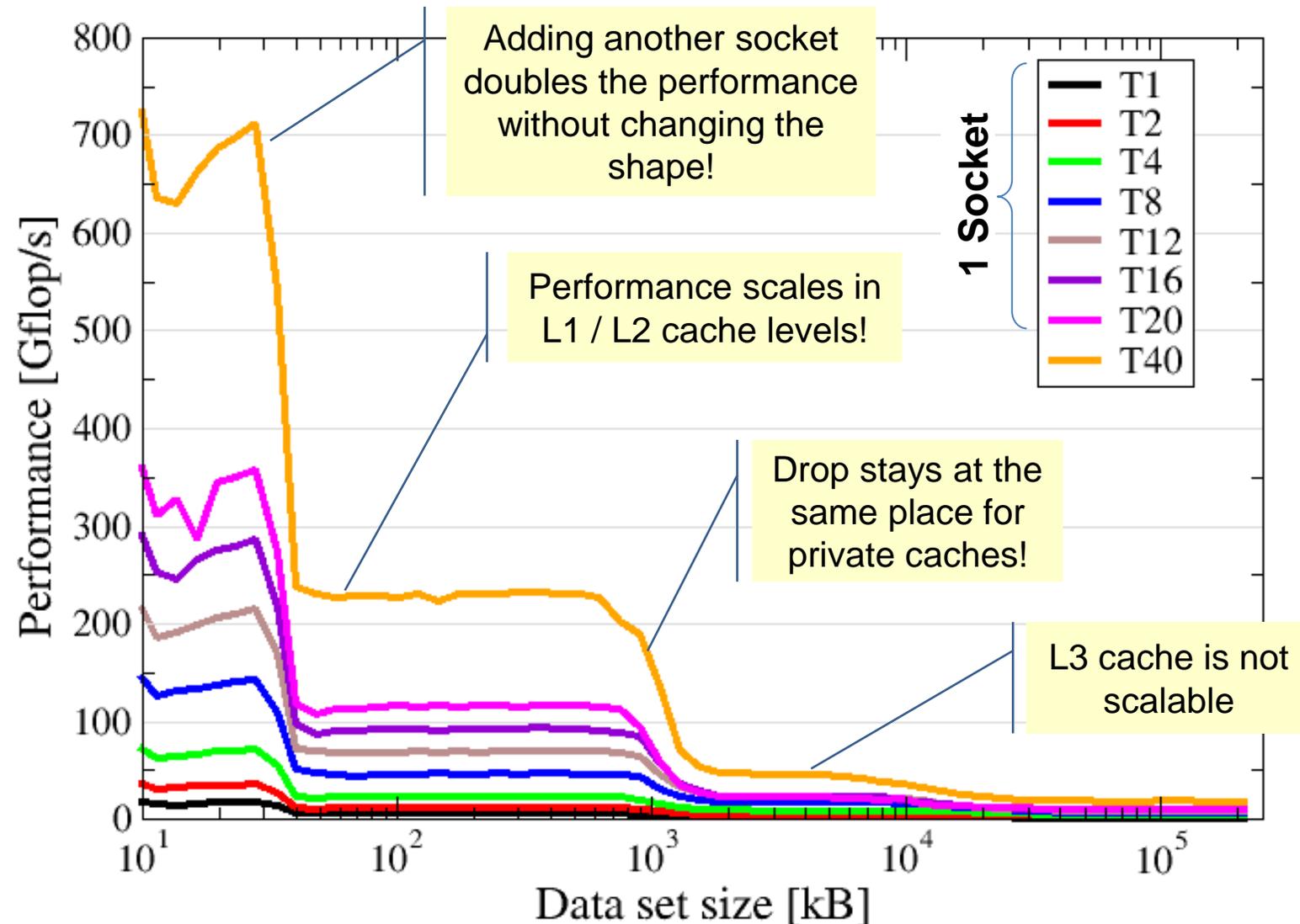


Throughput triad on one CascadeLake node (2.5 GHz)

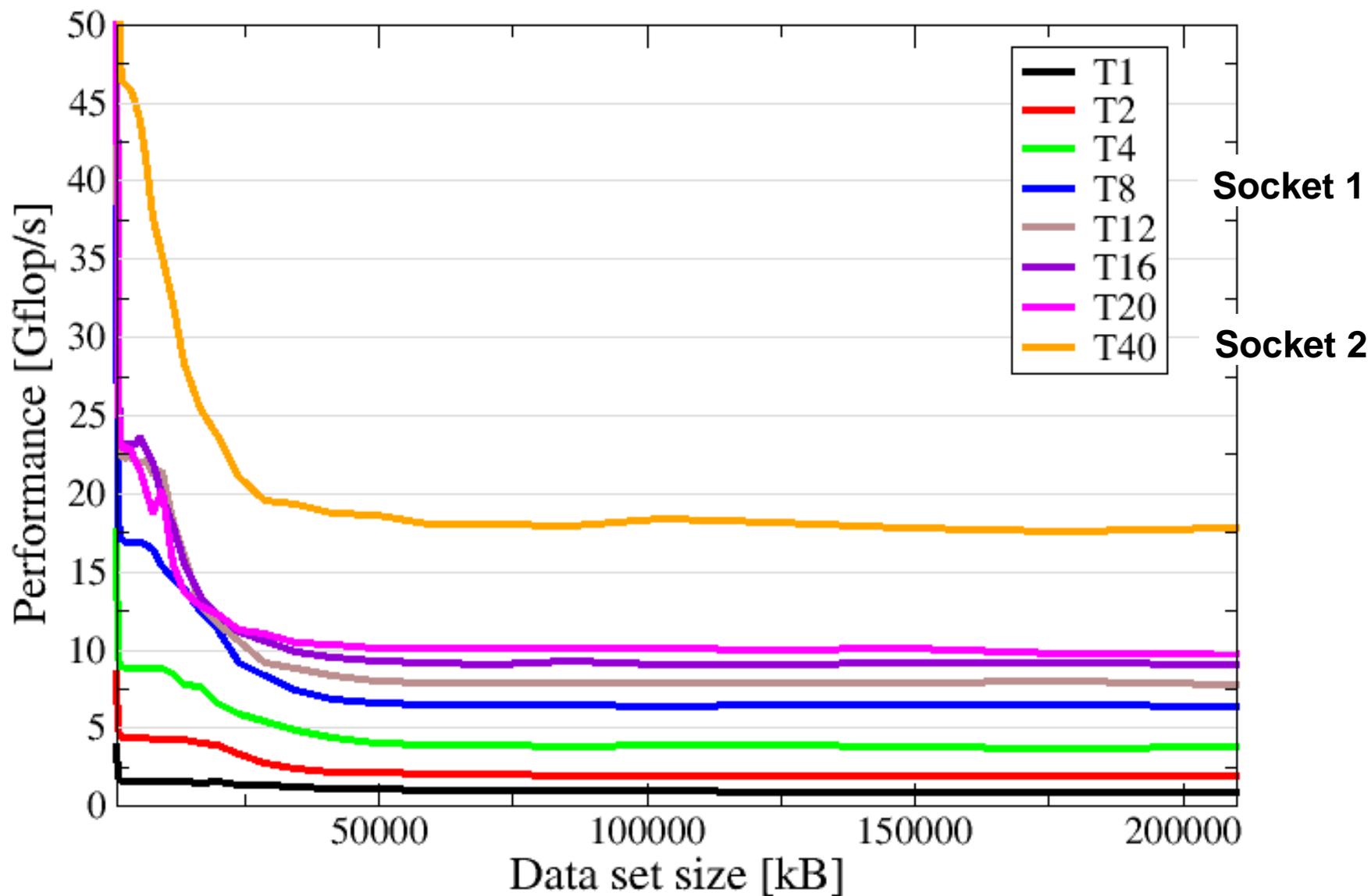
- How does the bandwidth scale across cores?
- Are there any bottlenecks?
- How large are the caches?

```
likwid-bench \  
-t triad_avx512_fma \  
-W S0:$size:$threads:1:2
```

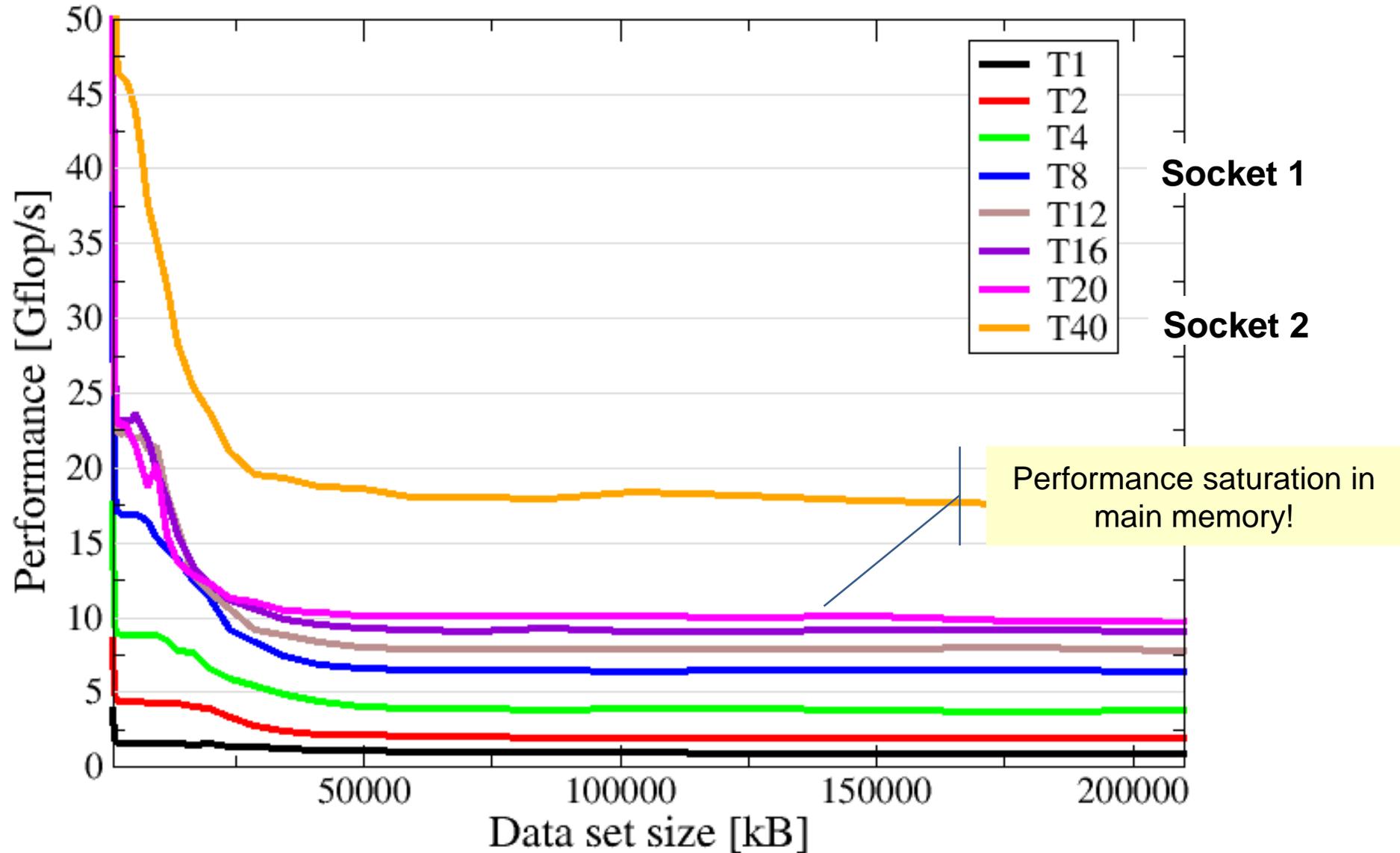
- Scan **\$size** and **\$threads**
- Pin threads in **chunks of 1** with **distance of 2** (skip SMT threads)



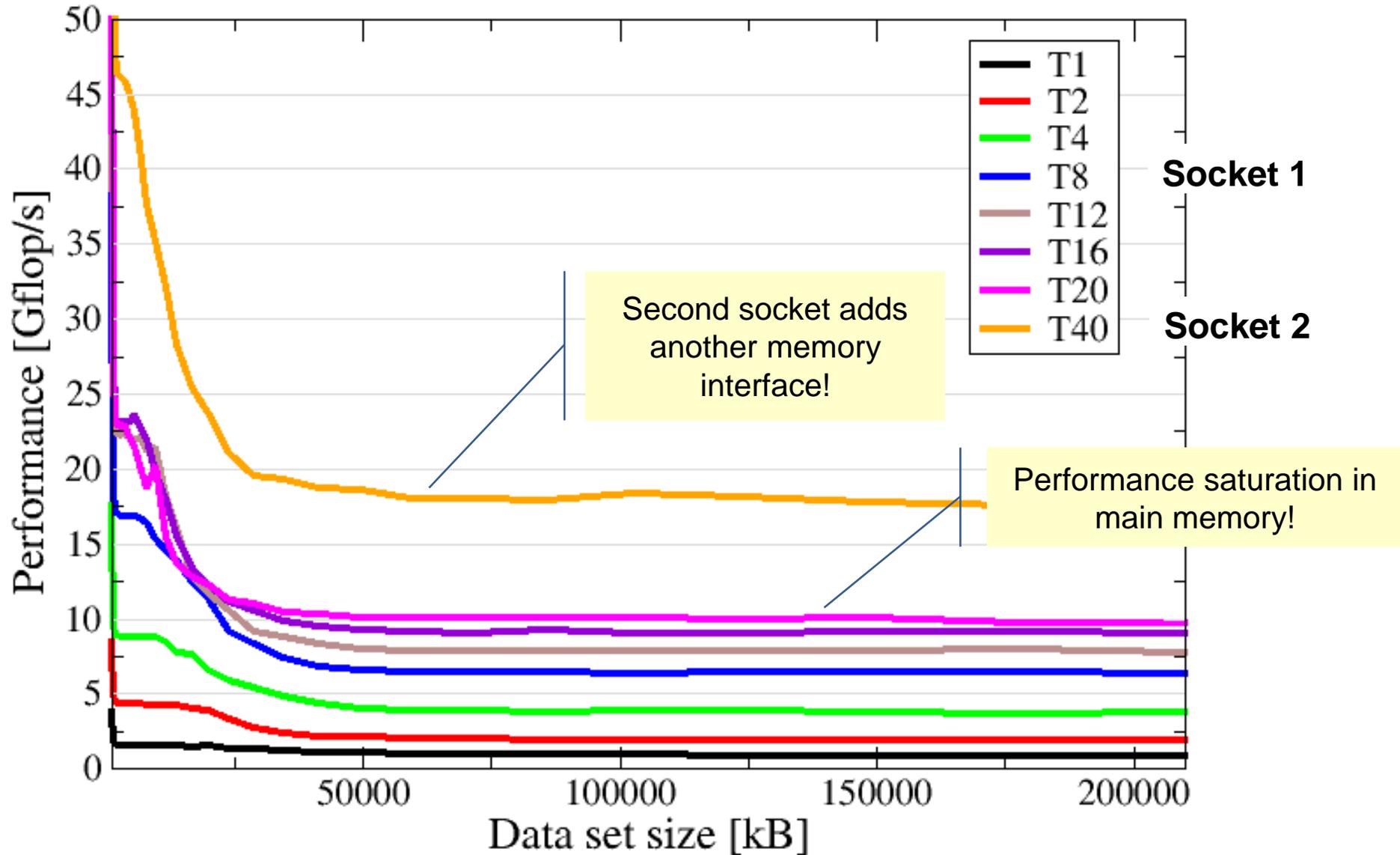
Throughput triad on CascadeLake (memory close-up)



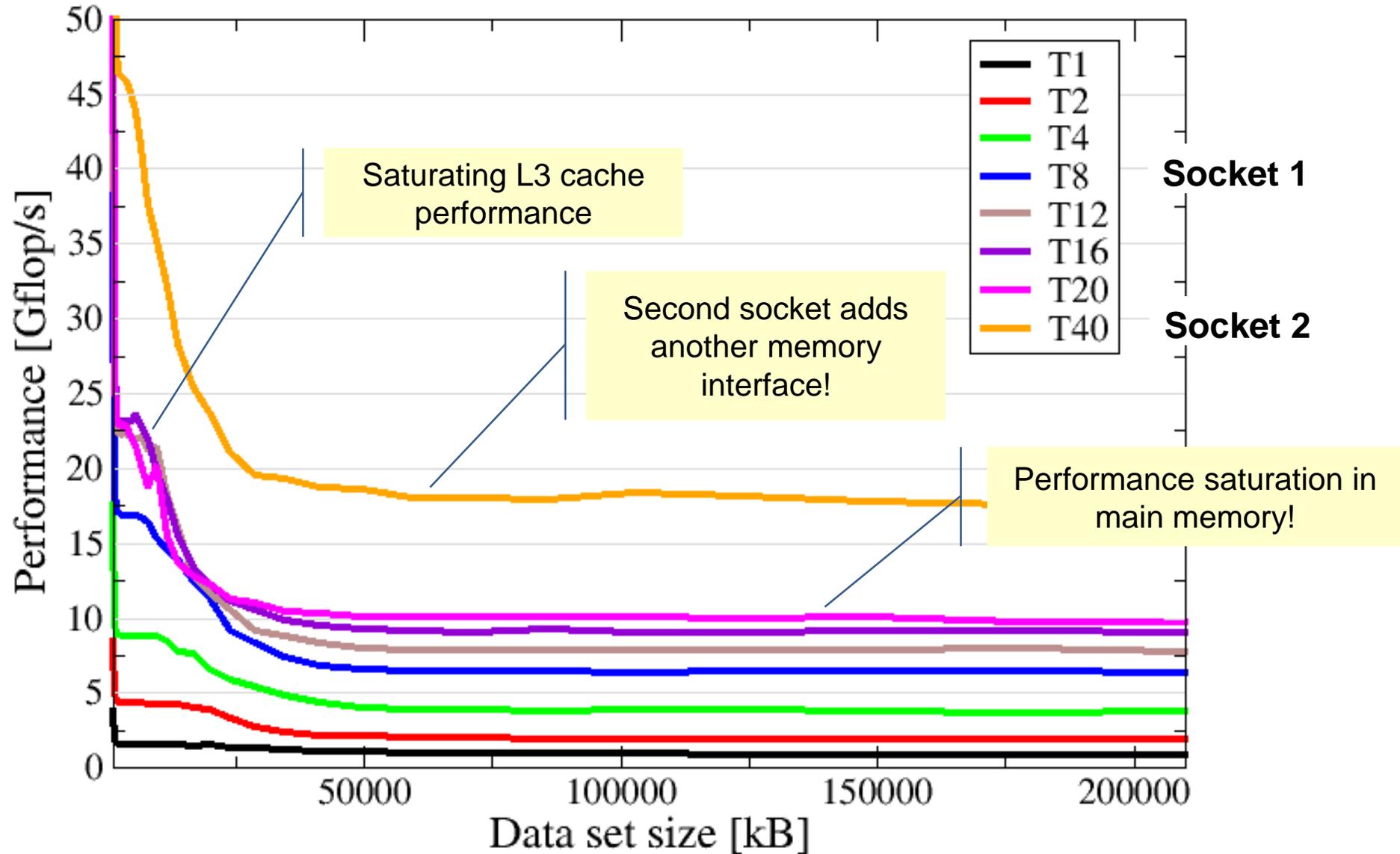
Throughput triad on CascadeLake (memory close-up)



Throughput triad on CascadeLake (memory close-up)

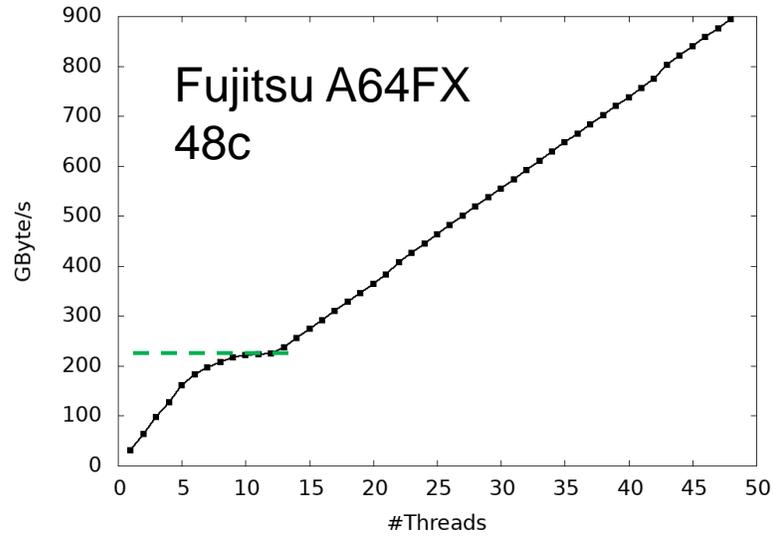


Throughput triad on CascadeLake (memory close-up)

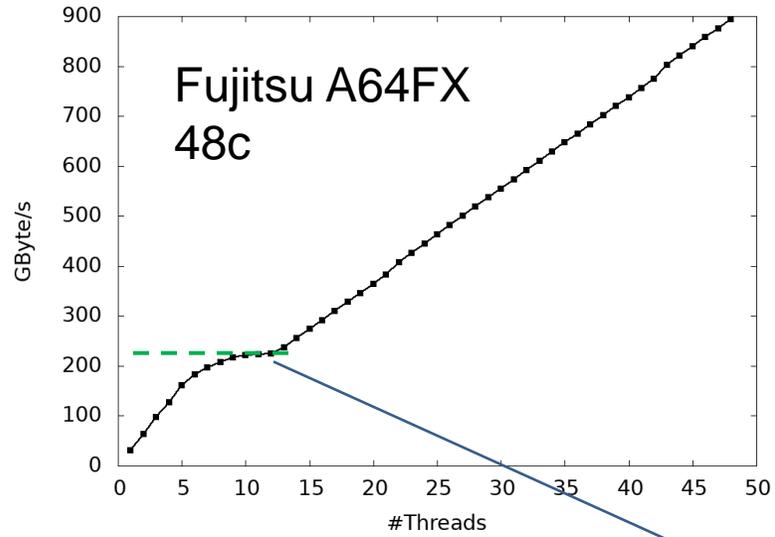


Memory bandwidth saturation (read-only)

Memory bandwidth saturation (read-only)

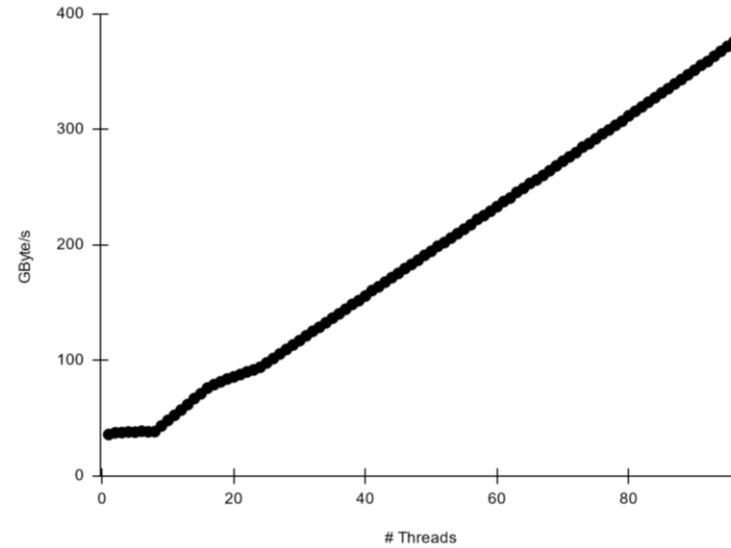
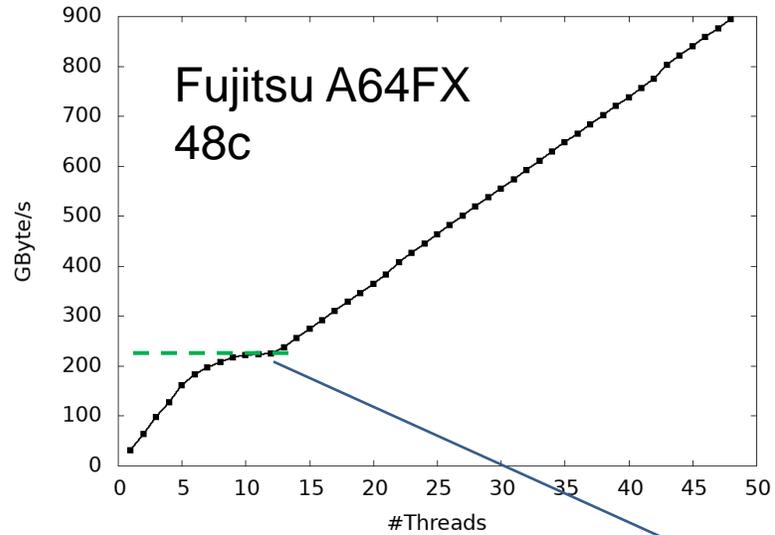


Memory bandwidth saturation (read-only)



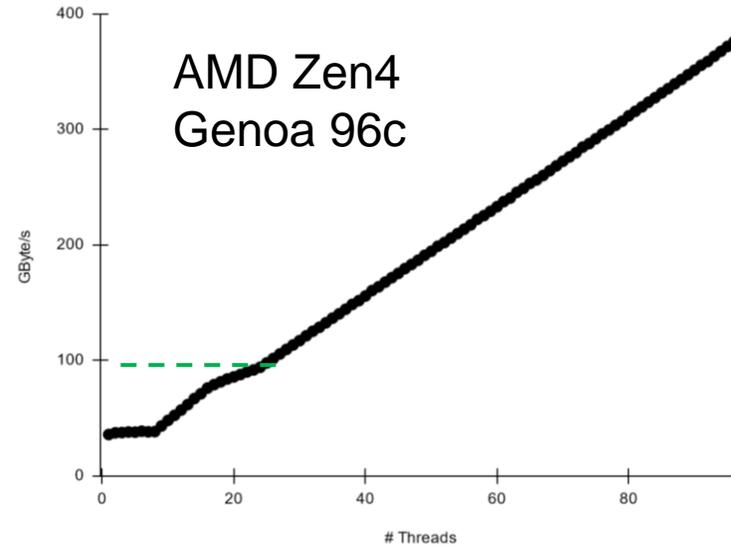
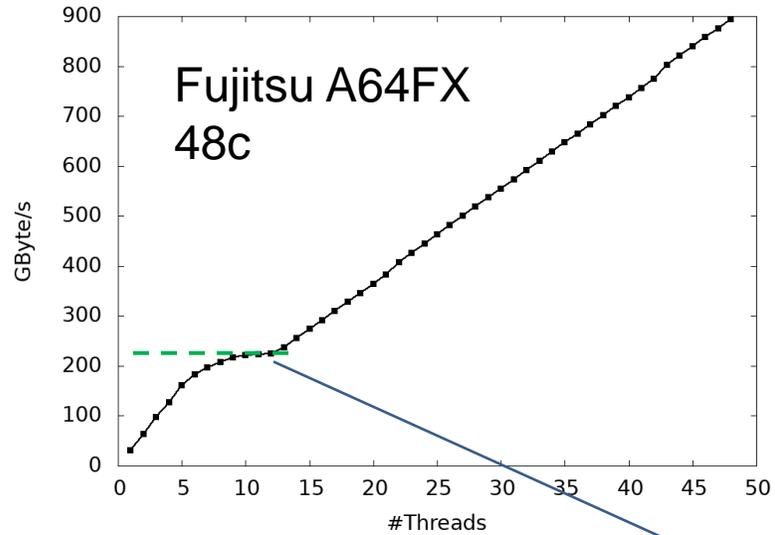
Bandwidth
saturation on 1st
ccNUMA domain

Memory bandwidth saturation (read-only)



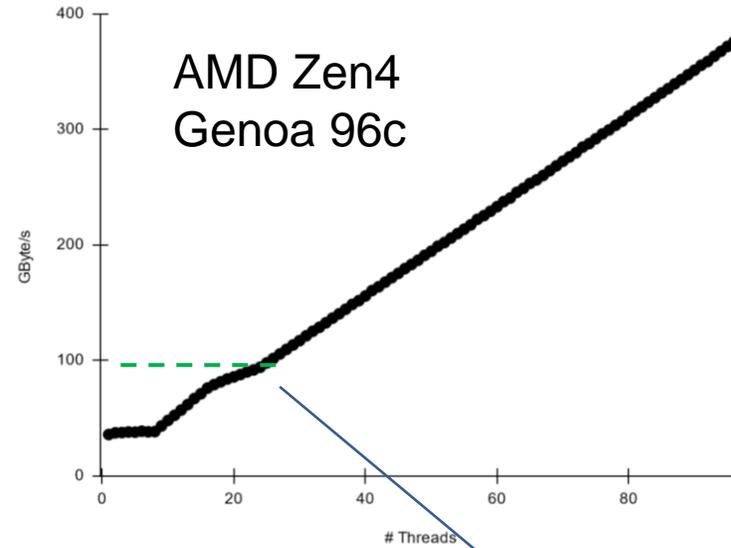
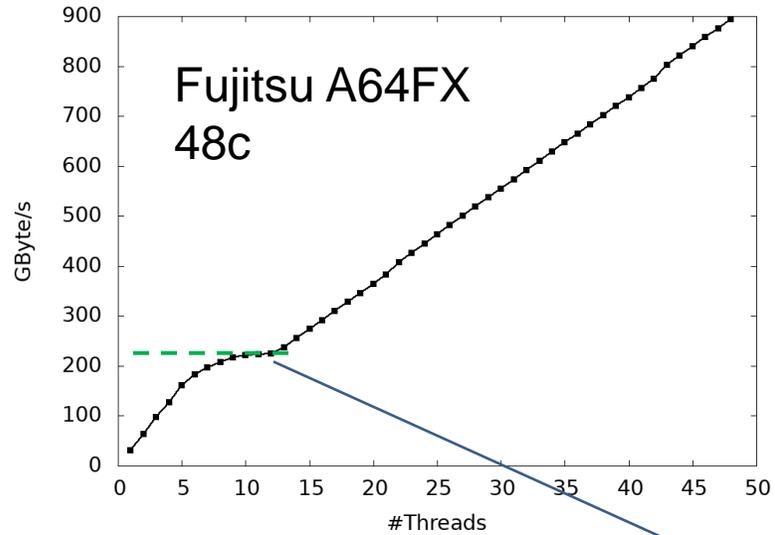
Bandwidth
saturation on 1st
ccNUMA domain

Memory bandwidth saturation (read-only)



Bandwidth
saturation on 1st
ccNUMA domain

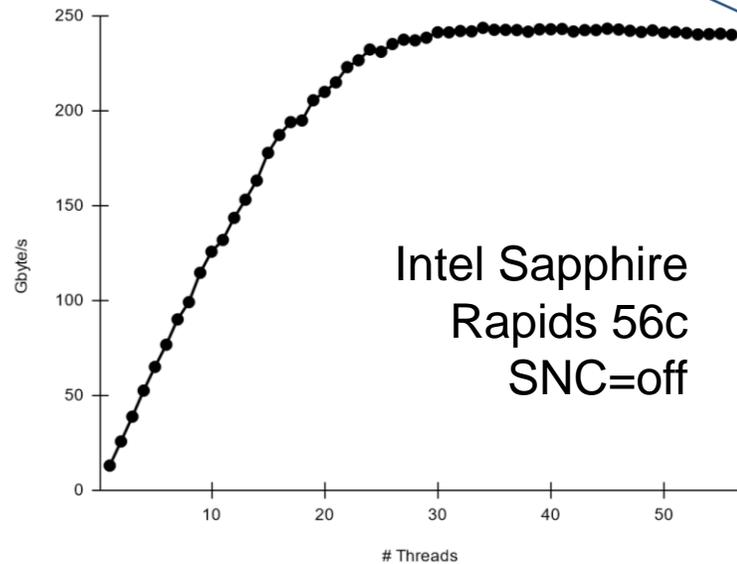
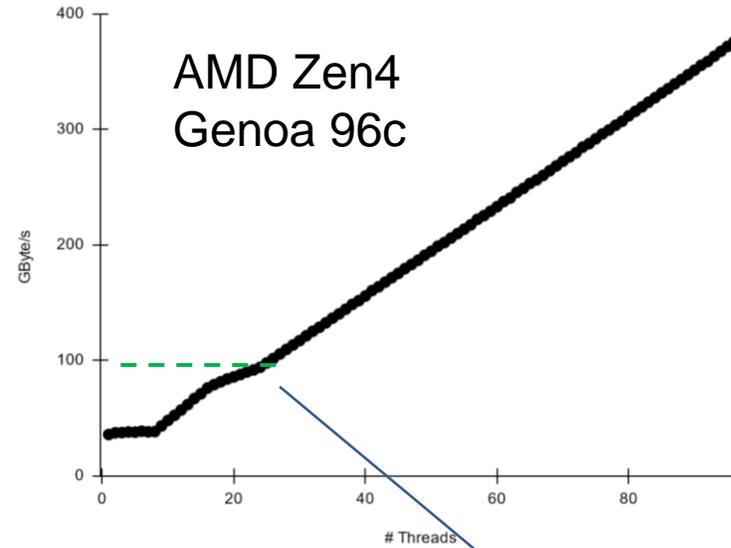
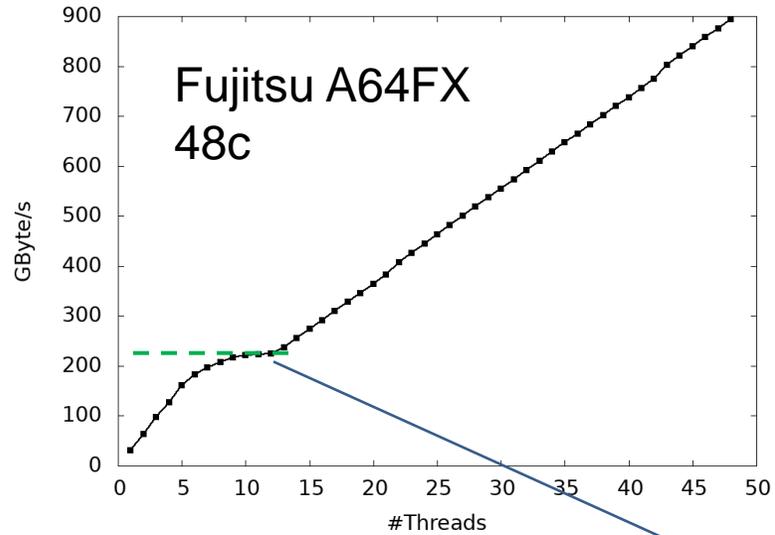
Memory bandwidth saturation (read-only)



Bandwidth
saturation on 1st
ccNUMA domain

Internal I/O
structure
visible

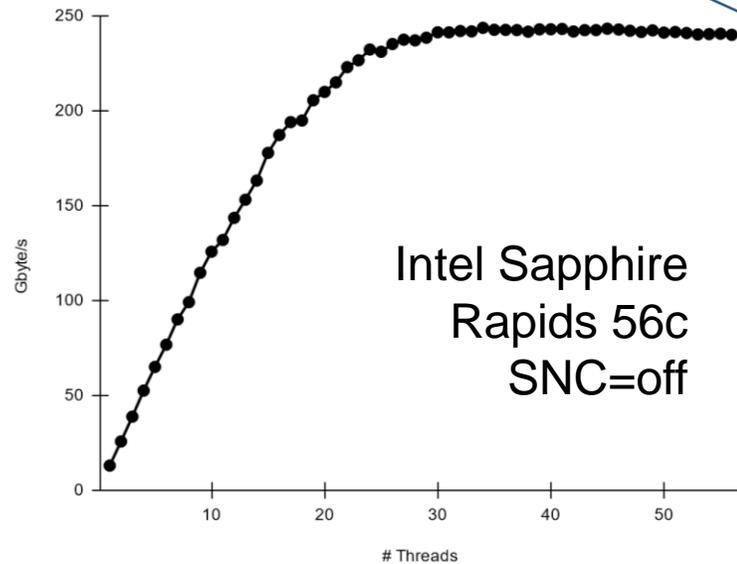
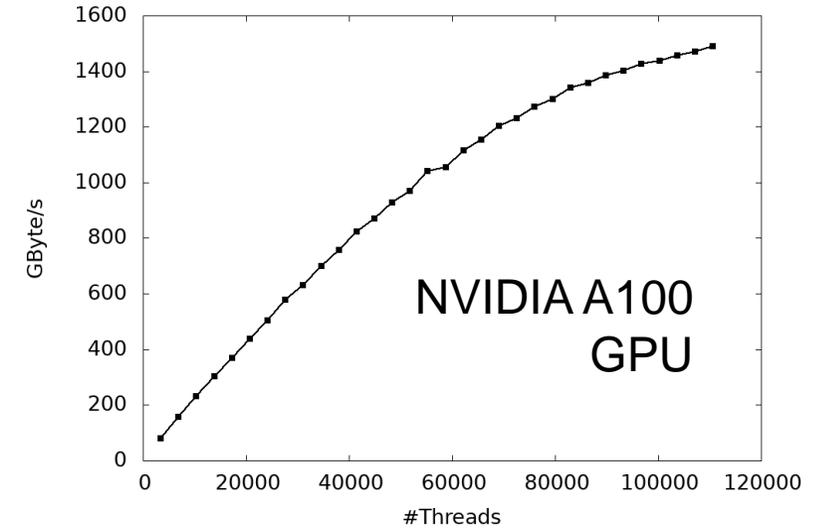
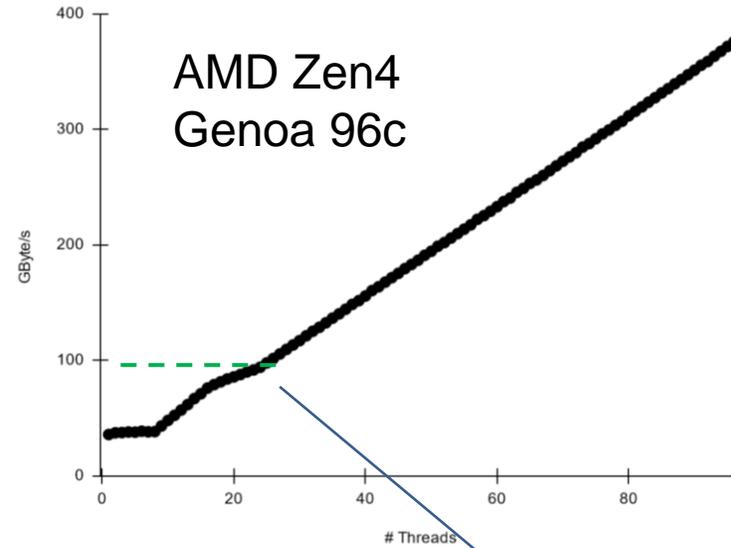
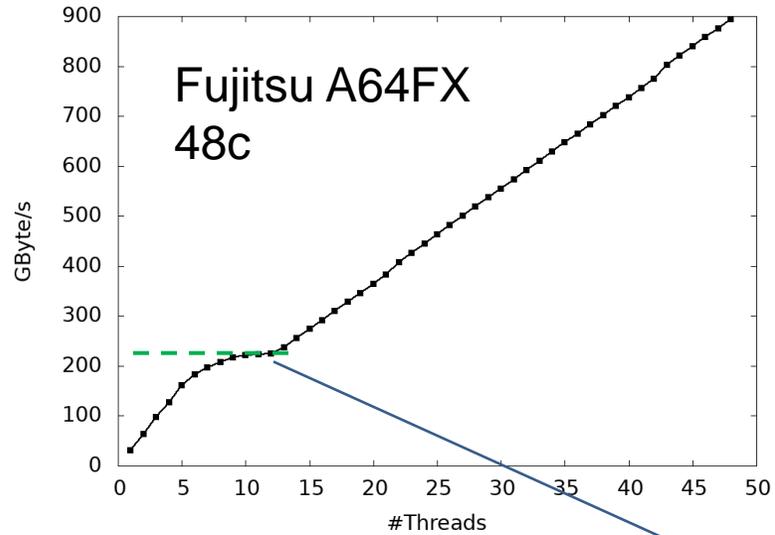
Memory bandwidth saturation (read-only)



Bandwidth saturation on 1st ccNUMA domain

Internal I/O structure visible

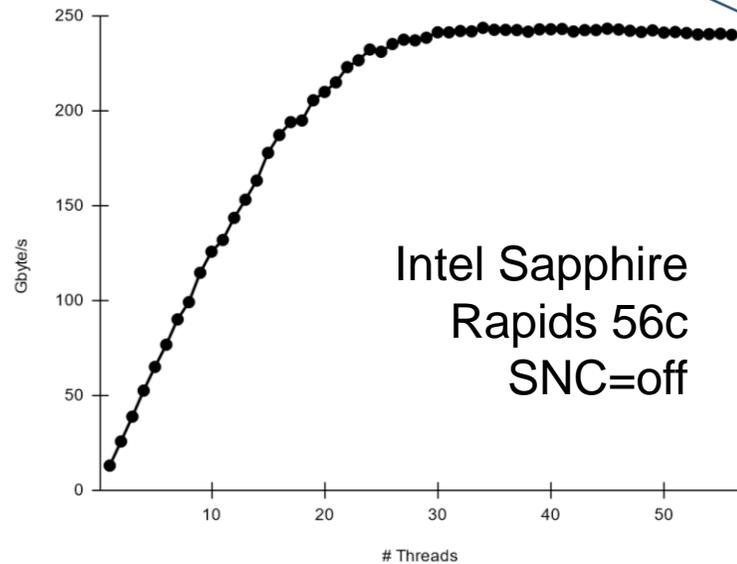
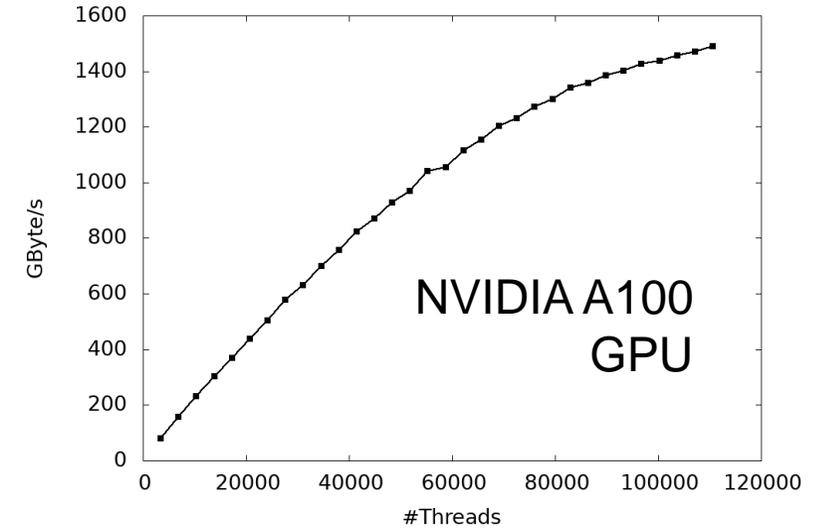
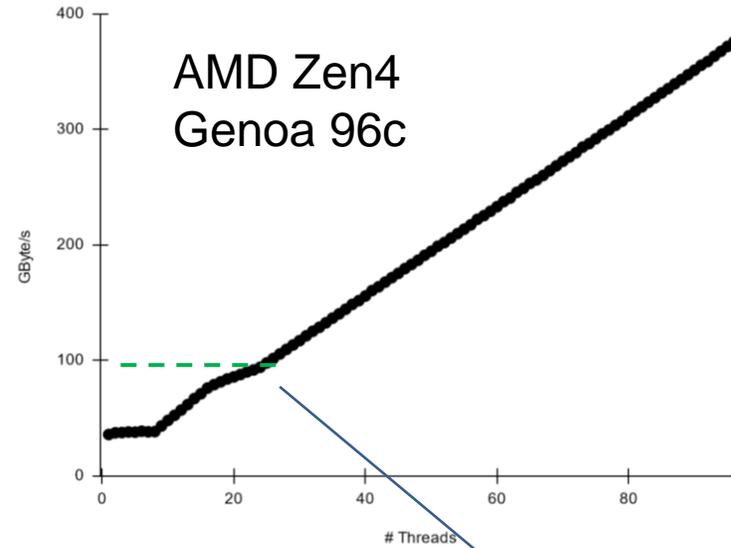
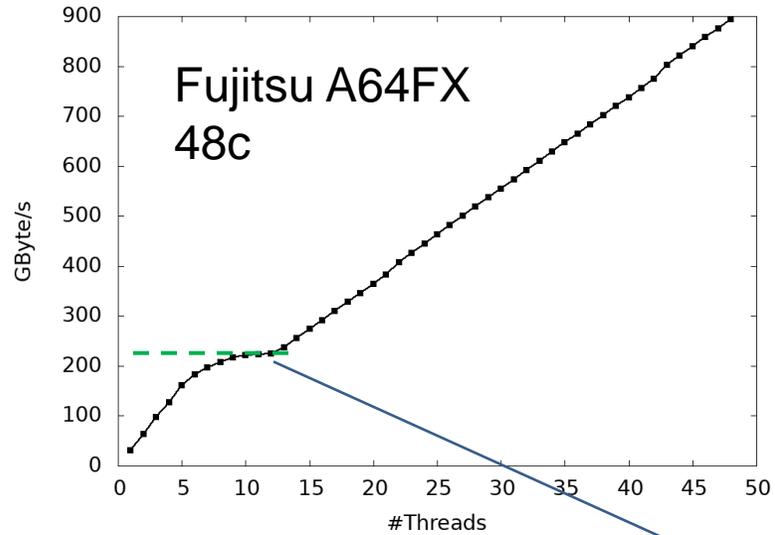
Memory bandwidth saturation (read-only)



Bandwidth saturation on 1st ccNUMA domain

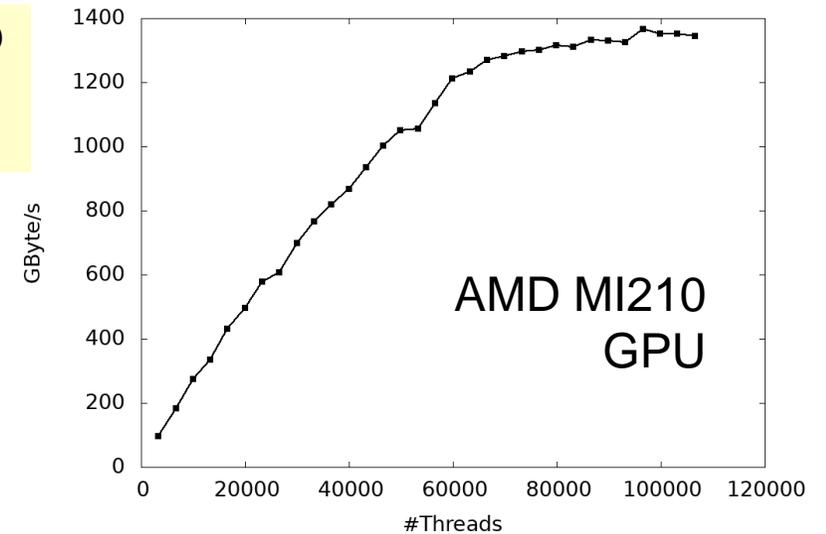
Internal I/O structure visible

Memory bandwidth saturation (read-only)

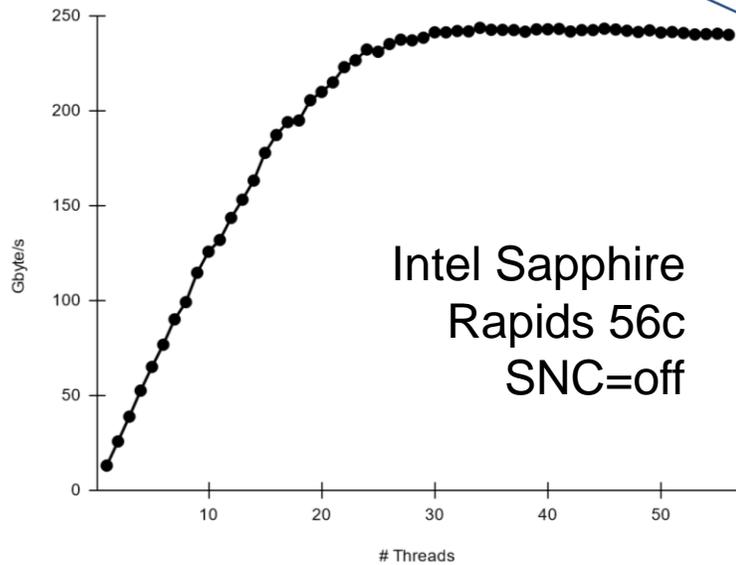
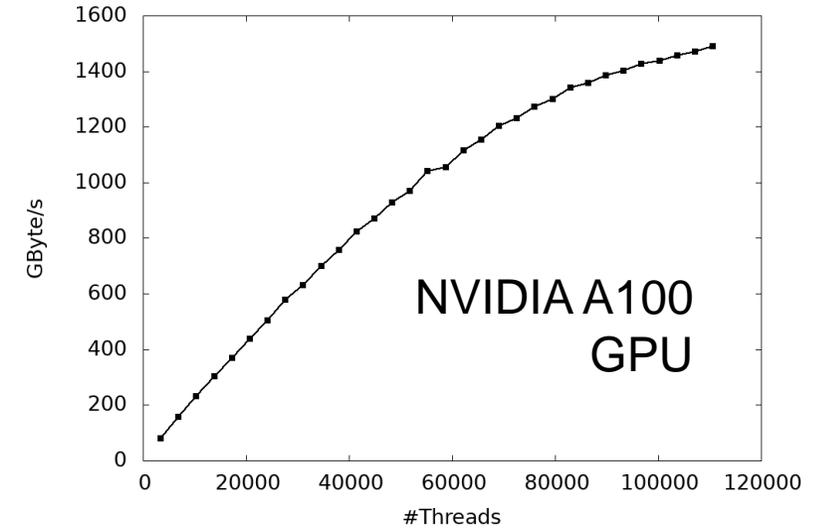
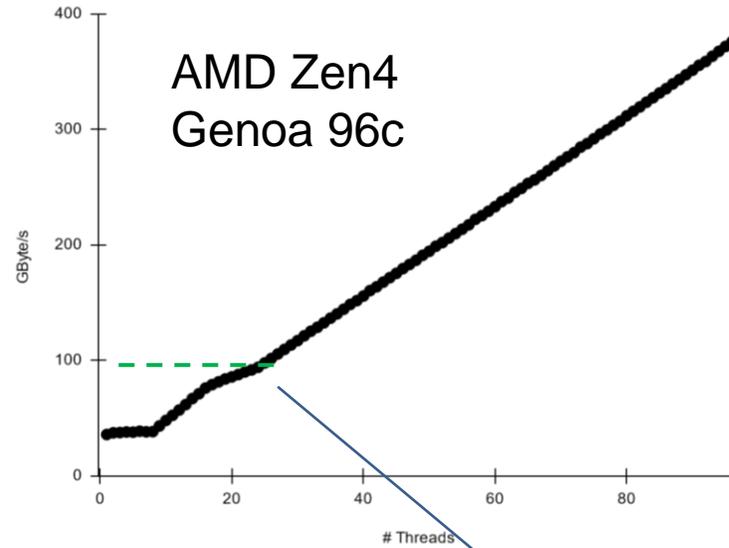
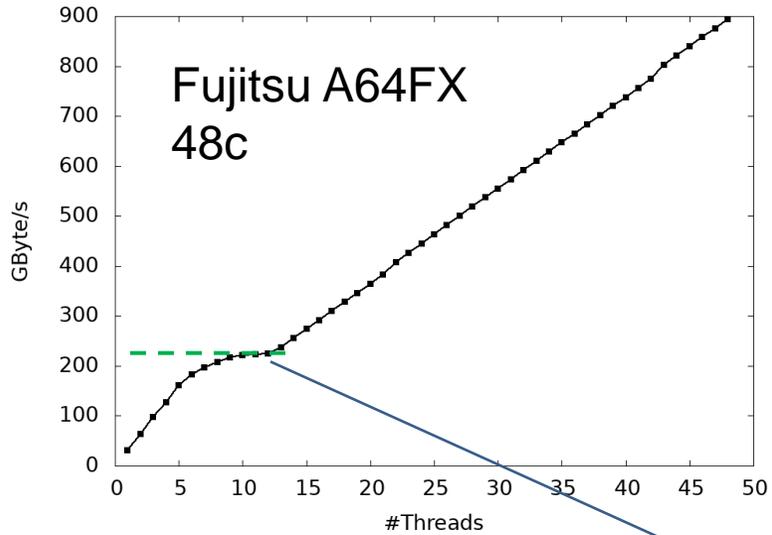


Bandwidth
saturation on 1st
ccNUMA domain

Internal I/O
structure
visible



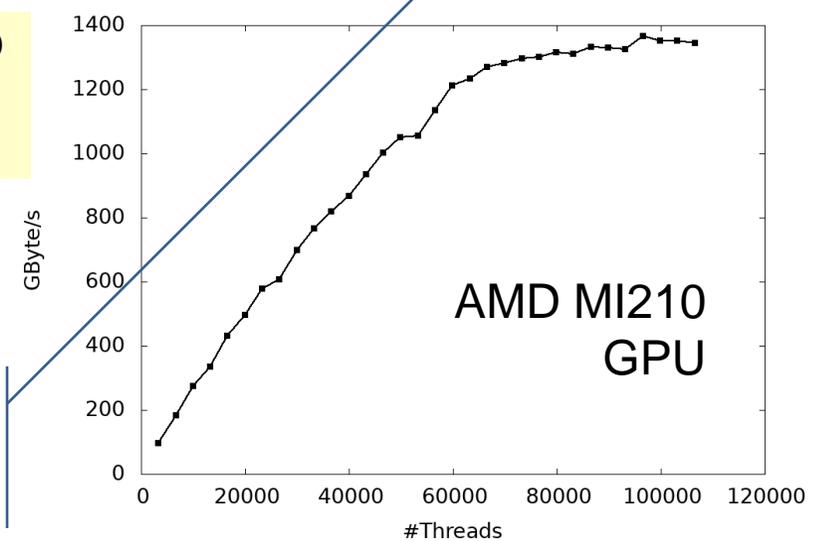
Memory bandwidth saturation (read-only)



Bandwidth
saturation on 1st
ccNUMA domain

Internal I/O
structure
visible

Massive thread
parallelism needed
on GPUs to saturate



The OpenMP-parallel vector triad benchmark

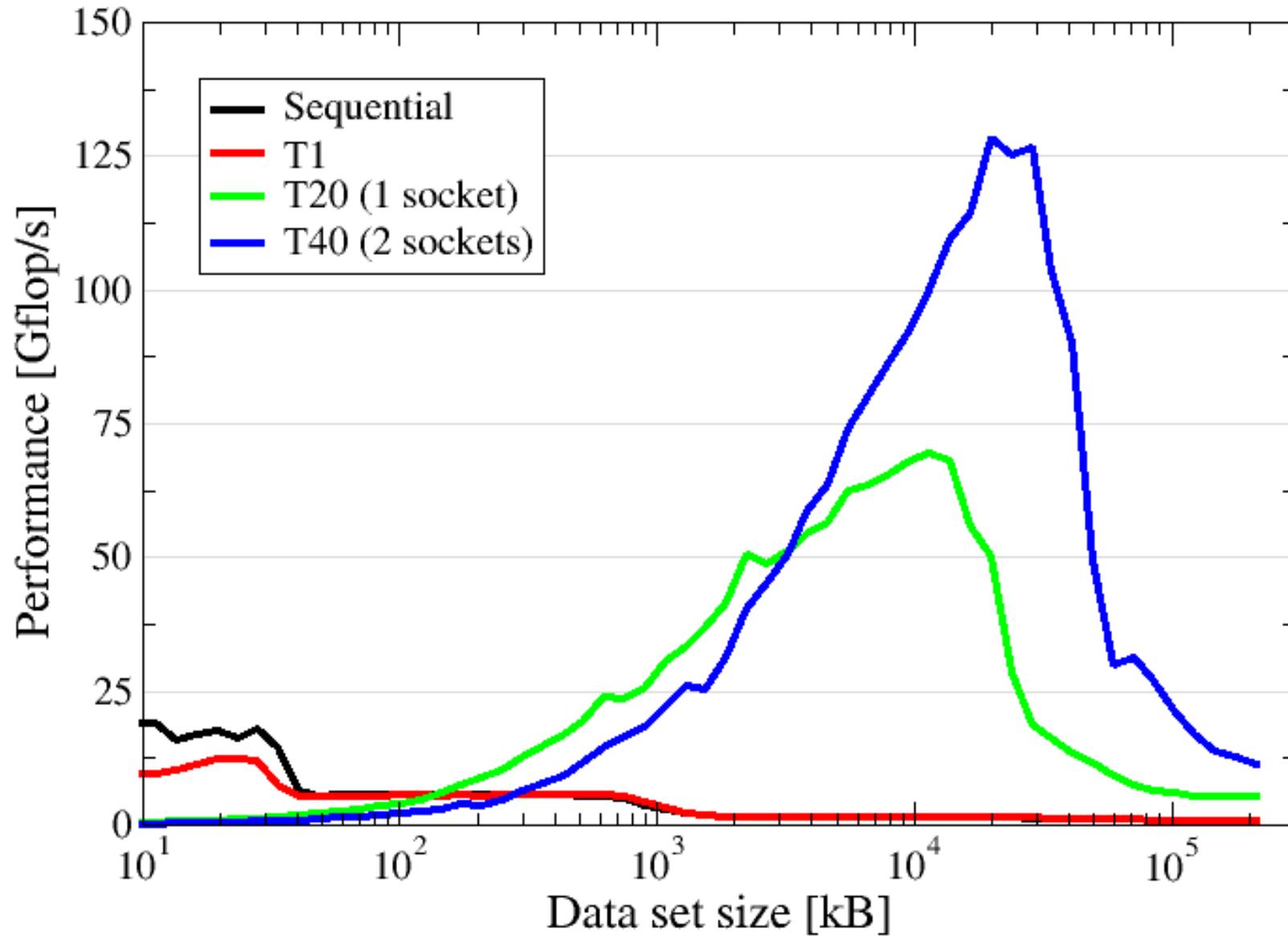
OpenMP worksharing in the benchmark loop

```
S = getTimeStamp();
#pragma omp parallel
{
    for(int j = 0; j < iter; j++) {
        #pragma omp for
        for (int i=0; i<N; i++) {
            a[i] = b[i] + d[i] * c[i];
        }
    }
}
E = getTimeStamp();
```

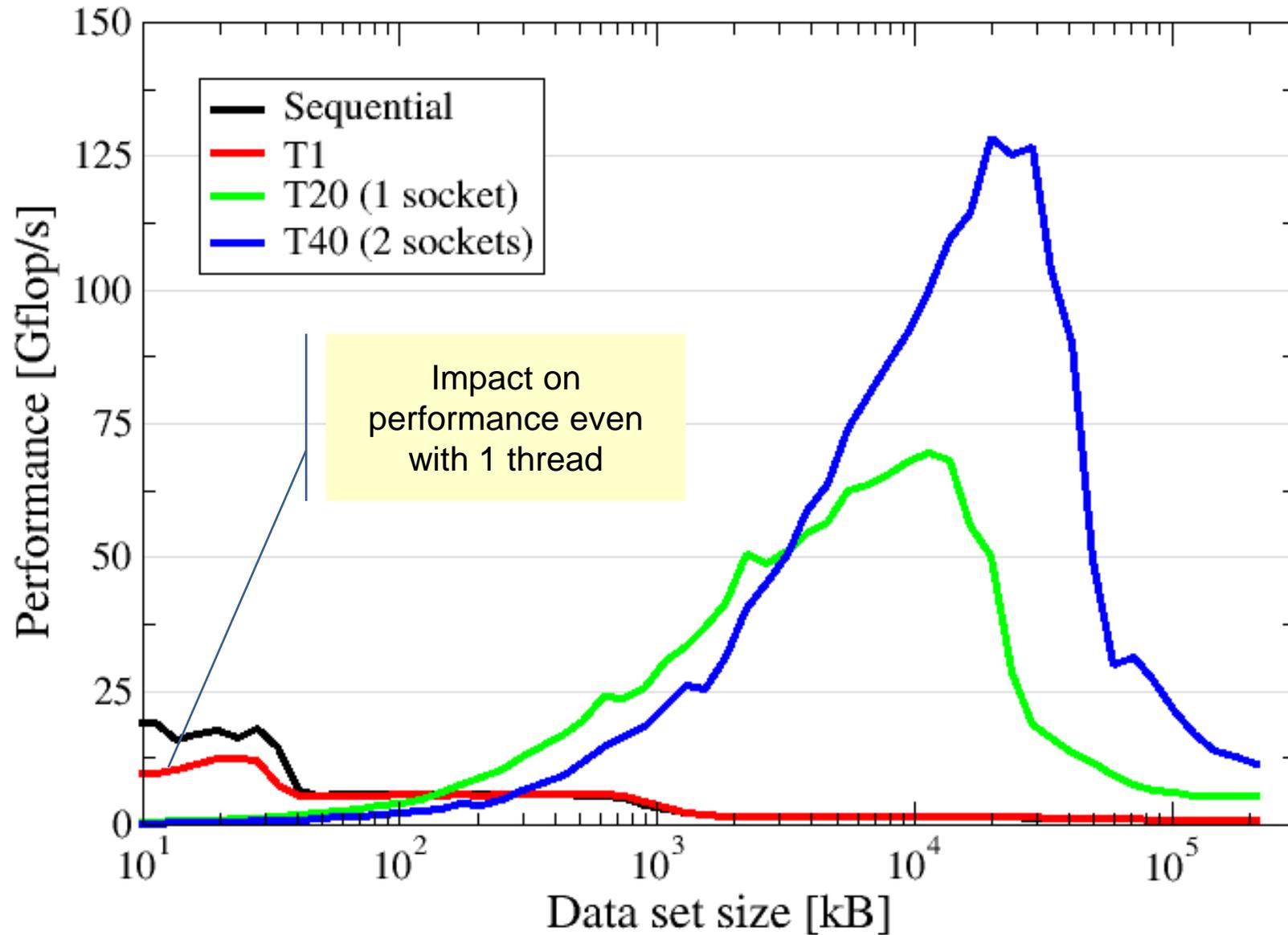


Implicit barrier

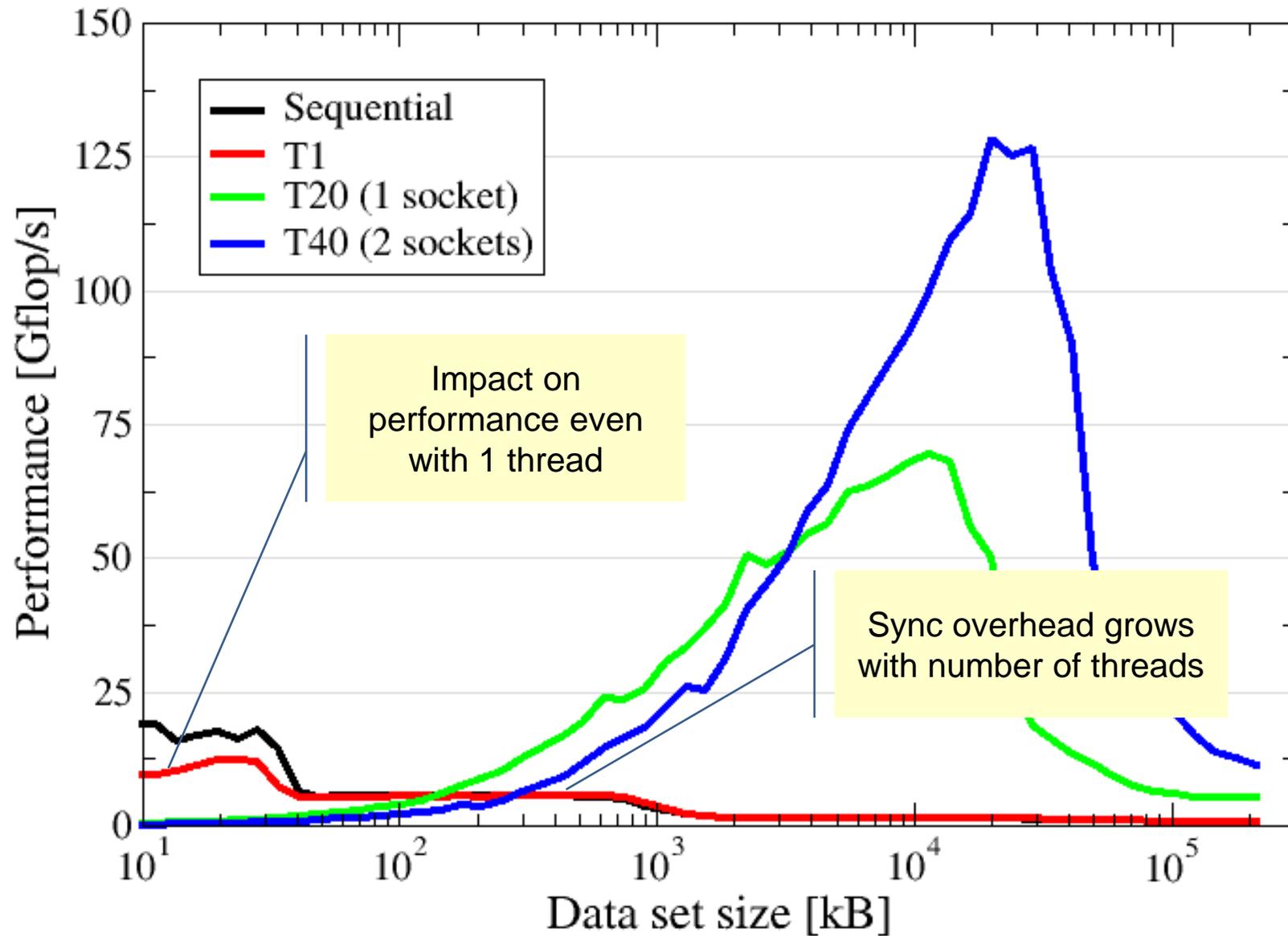
OpenMP vector triad on CascadeLake node (2.2 GHz)



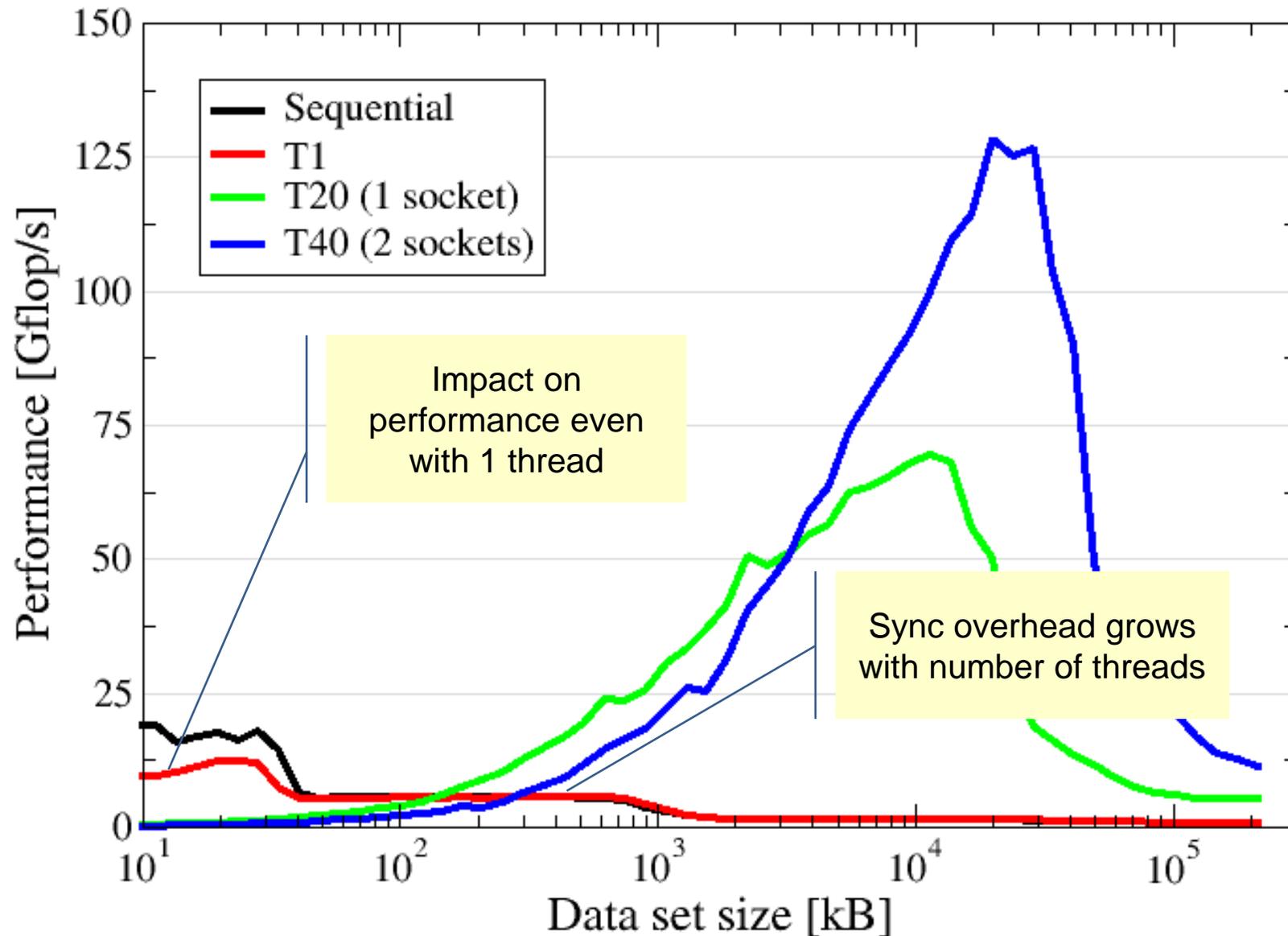
OpenMP vector triad on CascadeLake node (2.2 GHz)



OpenMP vector triad on CascadeLake node (2.2 GHz)



OpenMP vector triad on CascadeLake node (2.2 GHz)



- Typical barrier cost
- ~ 10000 cy full node
 - Scales with $\log(\#\text{cores})$
 - Depends on positions of threads (topology)

Conclusions from the microbenchmarks

- Microbenchmarks can yield surprisingly deep insights
- **Affinity matters!**
 - Almost all performance properties depend on the position of
 - Data
 - Threads/processes
 - Consequences
 - Know where your threads are running
 - Know where your data is (see later for that)
- **Bandwidth bottlenecks are ubiquitous**
- Synchronization overhead may be an issue
 - ... and depends on the system topology!
 - Many-core poses new challenges in terms of synchronization

“Simple” performance modeling: The Roofline Model

Loop-based performance modeling: Execution vs. data transfer

R.W. Hockney and I.J. Curington: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. *Parallel Computing* 10, 277-286 (1989). DOI: [10.1016/0167-8191\(89\)90100-2](https://doi.org/10.1016/0167-8191(89)90100-2)

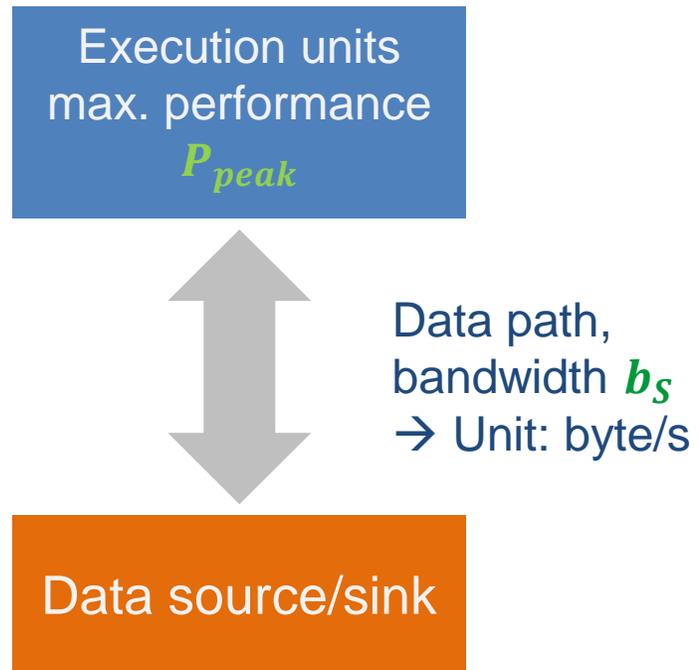
W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. Self-edition (2000)

S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)



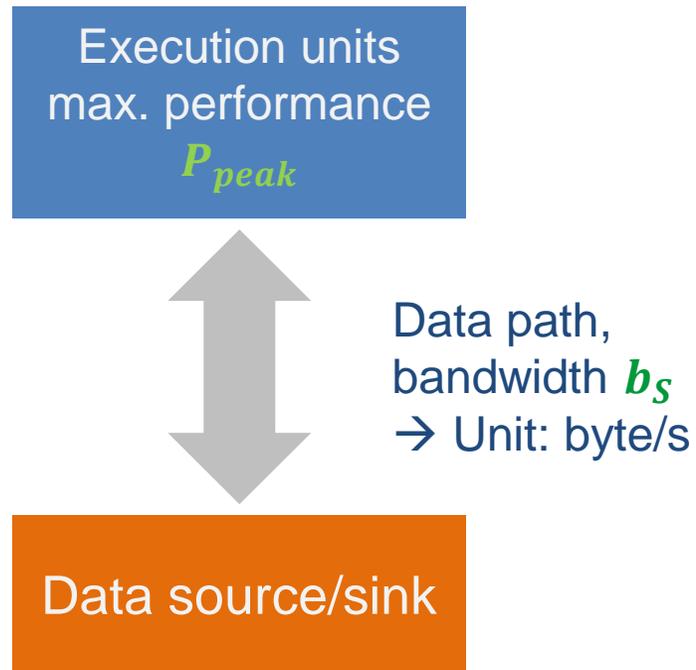
A simple performance model for loops

Simplistic view of the hardware:



A simple performance model for loops

Simplistic view of the hardware:



Simplistic view of the software:

```
do i = 1,<sufficient>  
  <complicated stuff doing  
    N flops causing  
    V bytes of data transfer>  
enddo
```

Computational intensity $I = \frac{N}{V}$
→ Unit: flop/byte

Naïve Roofline Model

How fast can tasks be processed at most? P [flop/s]

The bottleneck is either

- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

This is the “Naïve Roofline Model”

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- “Knee” at $P_{\text{peak}} = I \cdot b_S$:
Best use of resources
- Roofline is an “optimistic” model
(think “light speed”)

Naïve Roofline Model

How fast can tasks be processed at most? P [flop/s]

The bottleneck is either

- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{peak}, I \cdot b_S)$$

This is the “Naïve Roofline Model”

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- “Knee” at $P_{peak} = I \cdot b_S$:
Best use of resources
- Roofline is an “optimistic” model
(think “light speed”)

Naïve Roofline Model

How fast can tasks be processed at most? P [flop/s]

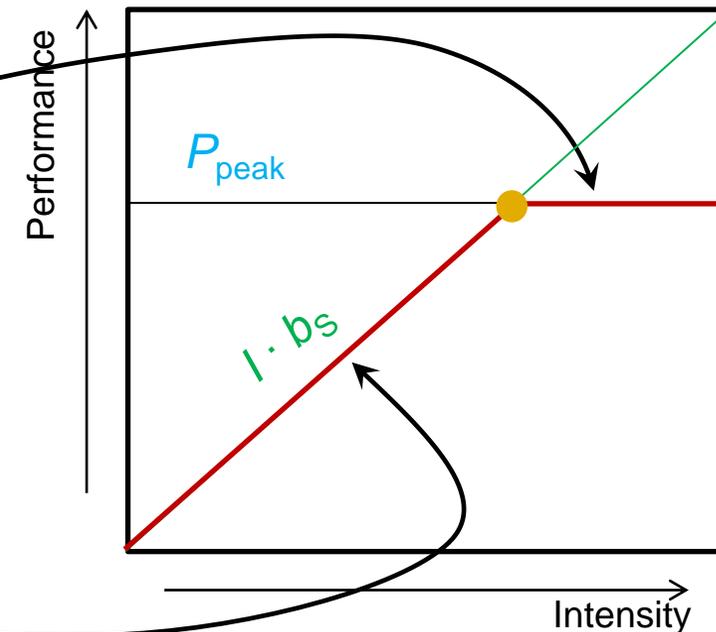
The bottleneck is either

- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{peak}, I \cdot b_S)$$

This is the “Naïve Roofline Model”

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- “Knee” at $P_{peak} = I \cdot b_S$:
Best use of resources
- Roofline is an “optimistic” model
(think “light speed”)



Roofline: application model and machine model

Roofline: application model and machine model

Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
- Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
- Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$

Roofline: application model and machine model

Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- } Machine model

Roofline: application model and machine model

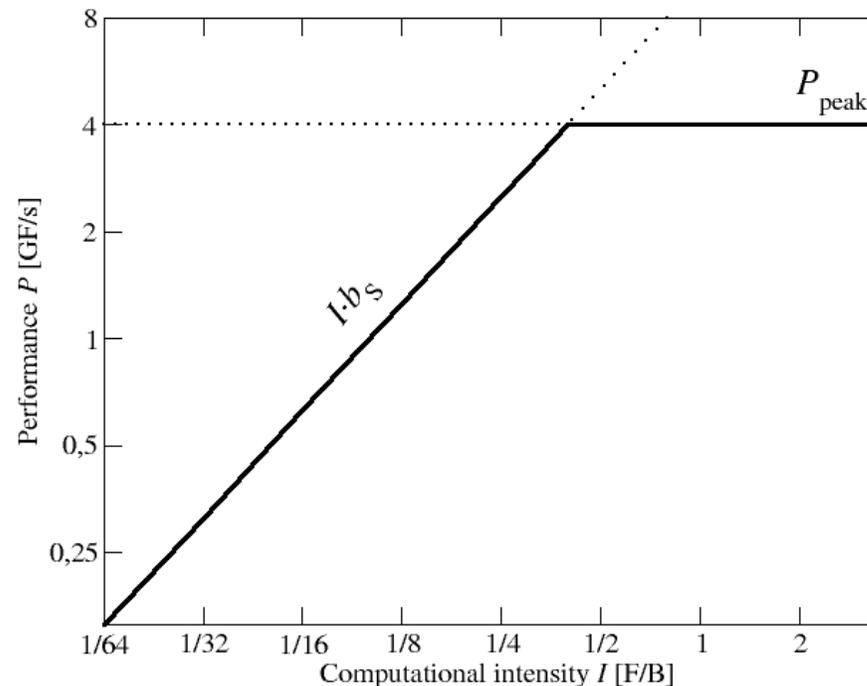
Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- Machine model
- Application model

Roofline: application model and machine model

Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- Machine model
- Application model



Roofline: application model and machine model

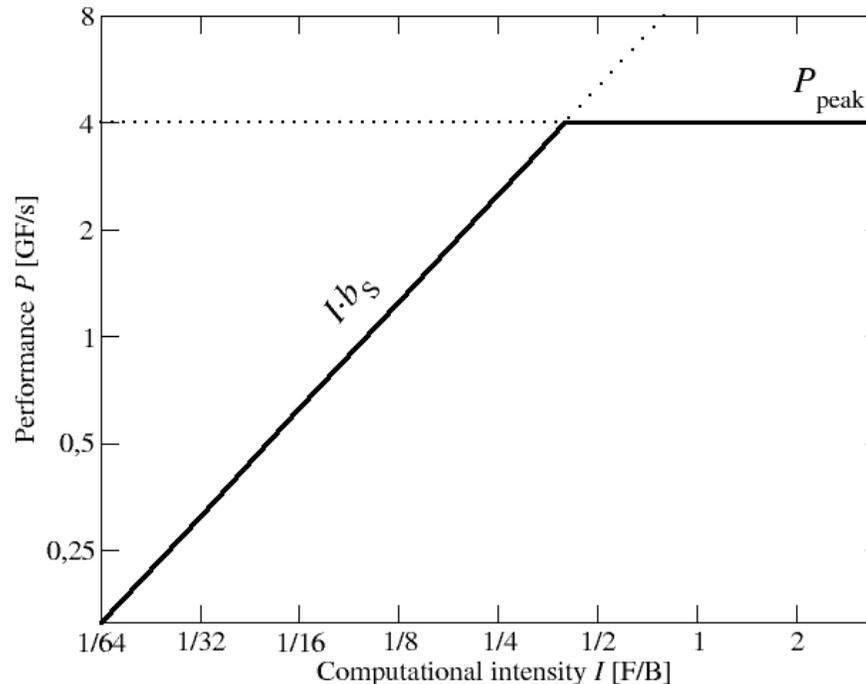
Apply the naive Roofline model in practice

- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- Machine model
- Application model

Machine properties:

$$P_{peak} = 4 \frac{GF}{s}$$

$$b_S = 10 \frac{GB}{s}$$



Roofline: application model and machine model

Apply the naive Roofline model in practice

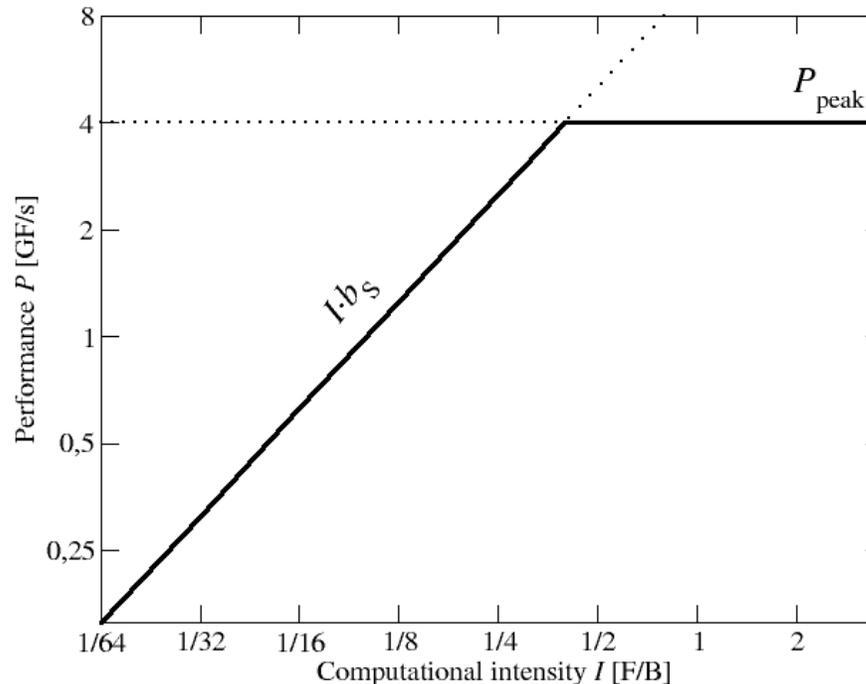
- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- Machine model
- Application model

Machine properties:

$$P_{peak} = 4 \frac{GF}{s}$$

$$b_S = 10 \frac{GB}{s}$$

Application property: I



Roofline: application model and machine model

Apply the naive Roofline model in practice

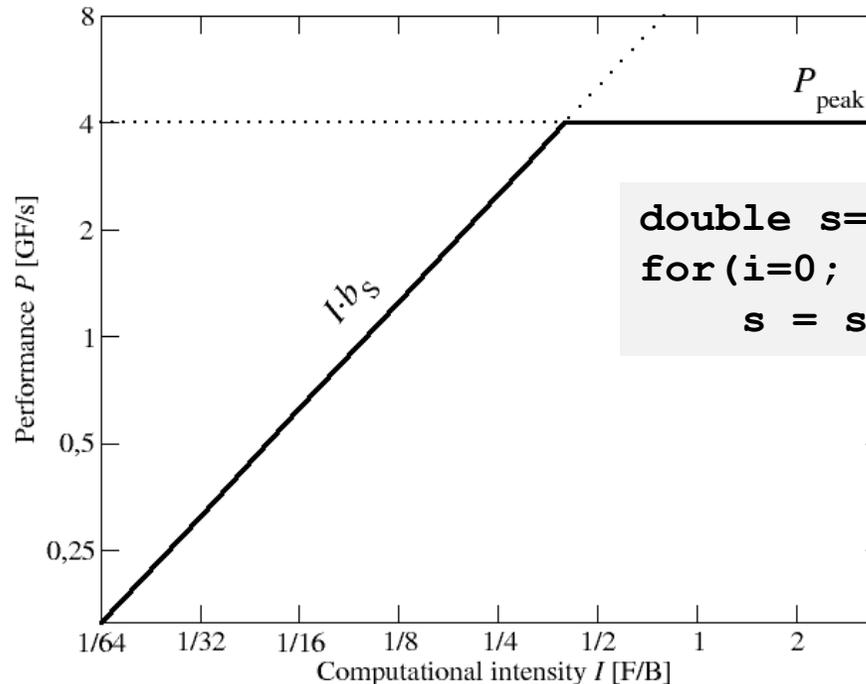
- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- Machine model
- Application model

Machine properties:

$$P_{peak} = 4 \frac{GF}{s}$$

$$b_S = 10 \frac{GB}{s}$$

Application property: I



```
double s=0, a[];  
for(i=0; i<N; ++i) {  
    s = s + a[i] * a[i];  
}
```

$$I = \frac{2F}{8B} = 0.25 \frac{F}{B}$$

Roofline: application model and machine model

Apply the naive Roofline model in practice

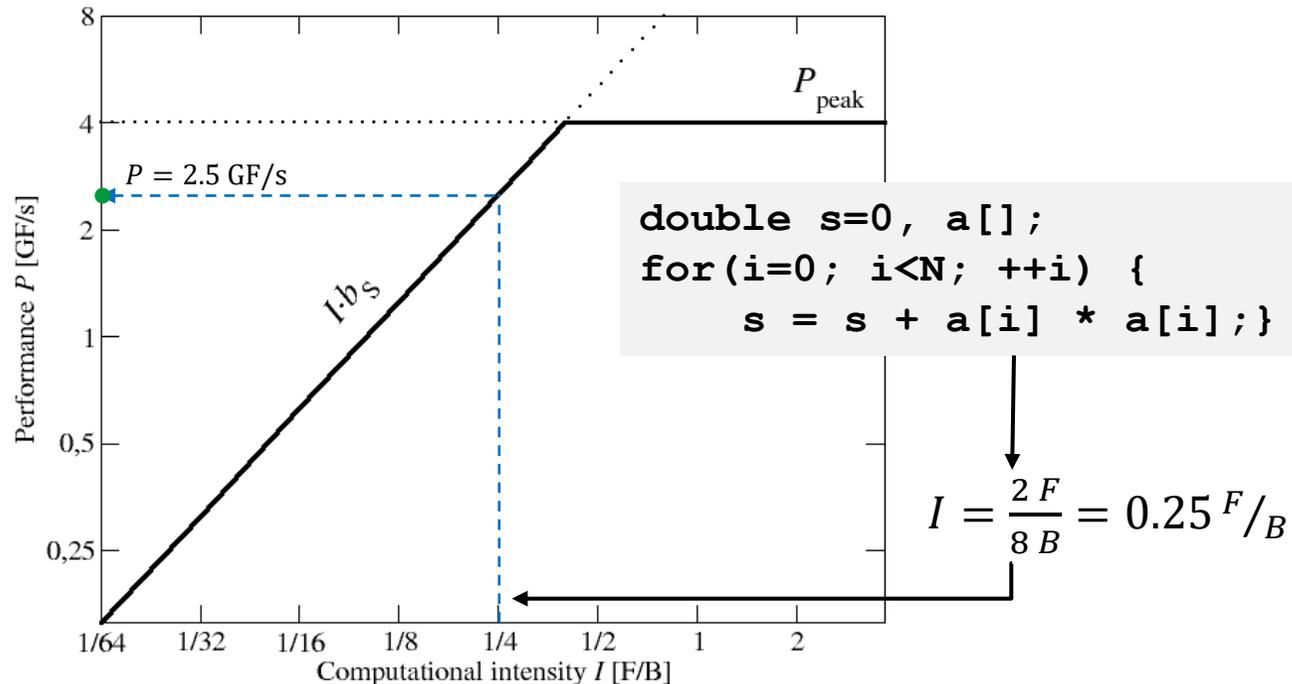
- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- } Machine model
} Application model

Machine properties:

$$P_{peak} = 4 \frac{GF}{s}$$

$$b_S = 10 \frac{GB}{s}$$

Application property: I



Prerequisites for the Roofline Model

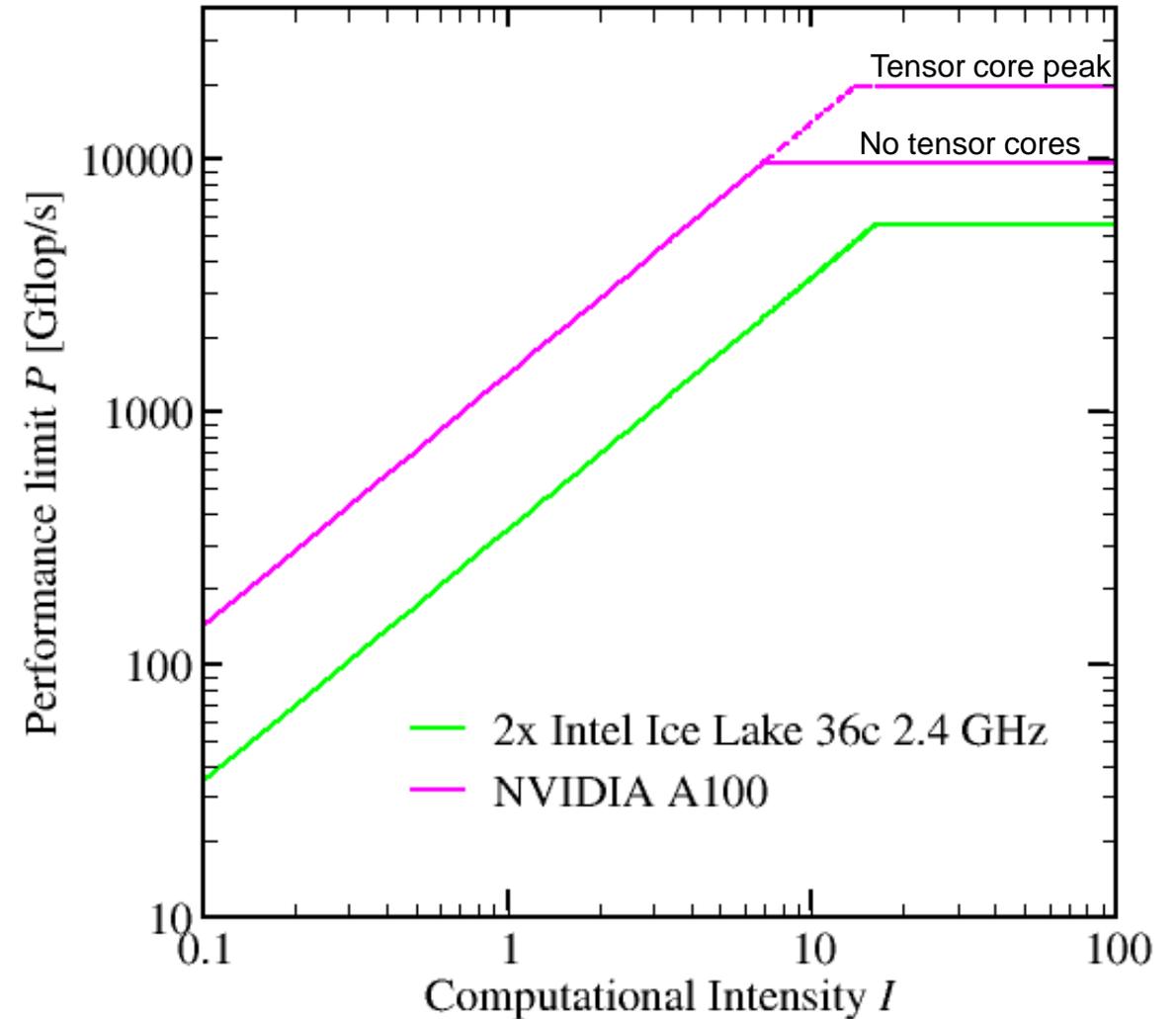
- Data transfer and core execution overlap perfectly!
 - Either the limit is core execution or it is data transfer
- Slowest limiting factor “wins”; all others are assumed to have no impact
 - If two bottlenecks are “close,” no interaction is assumed
- Data access latency is ignored, i.e. perfect streaming mode
 - Achievable bandwidth is the limit
- Chip must be able to saturate the bandwidth bottleneck(s)
 - Always model the full chip



Roofline for architecture and code comparison

With Roofline, we can

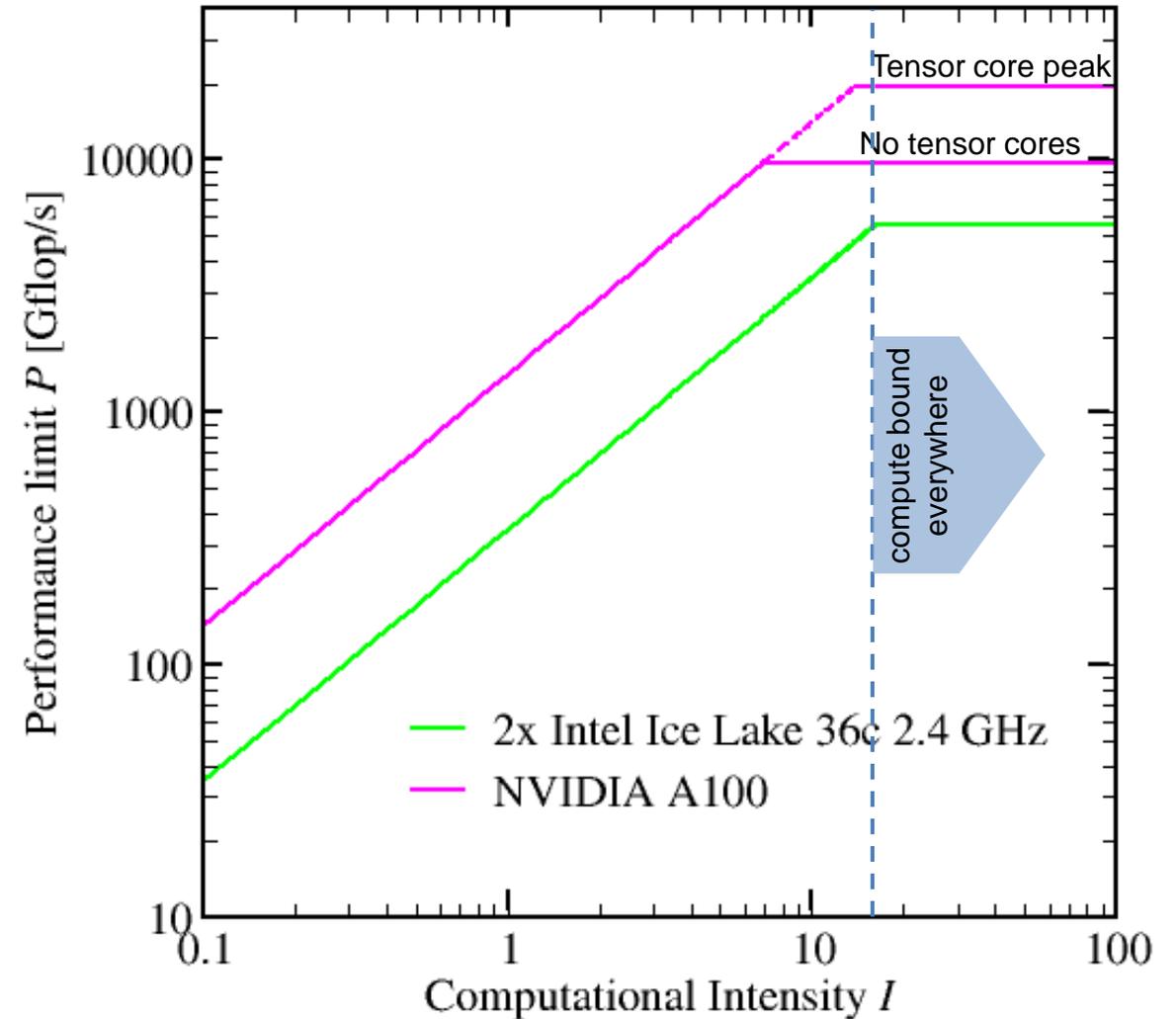
- Compare capabilities of different machines
- Compare performance expectations for different loops



Roofline for architecture and code comparison

With Roofline, we can

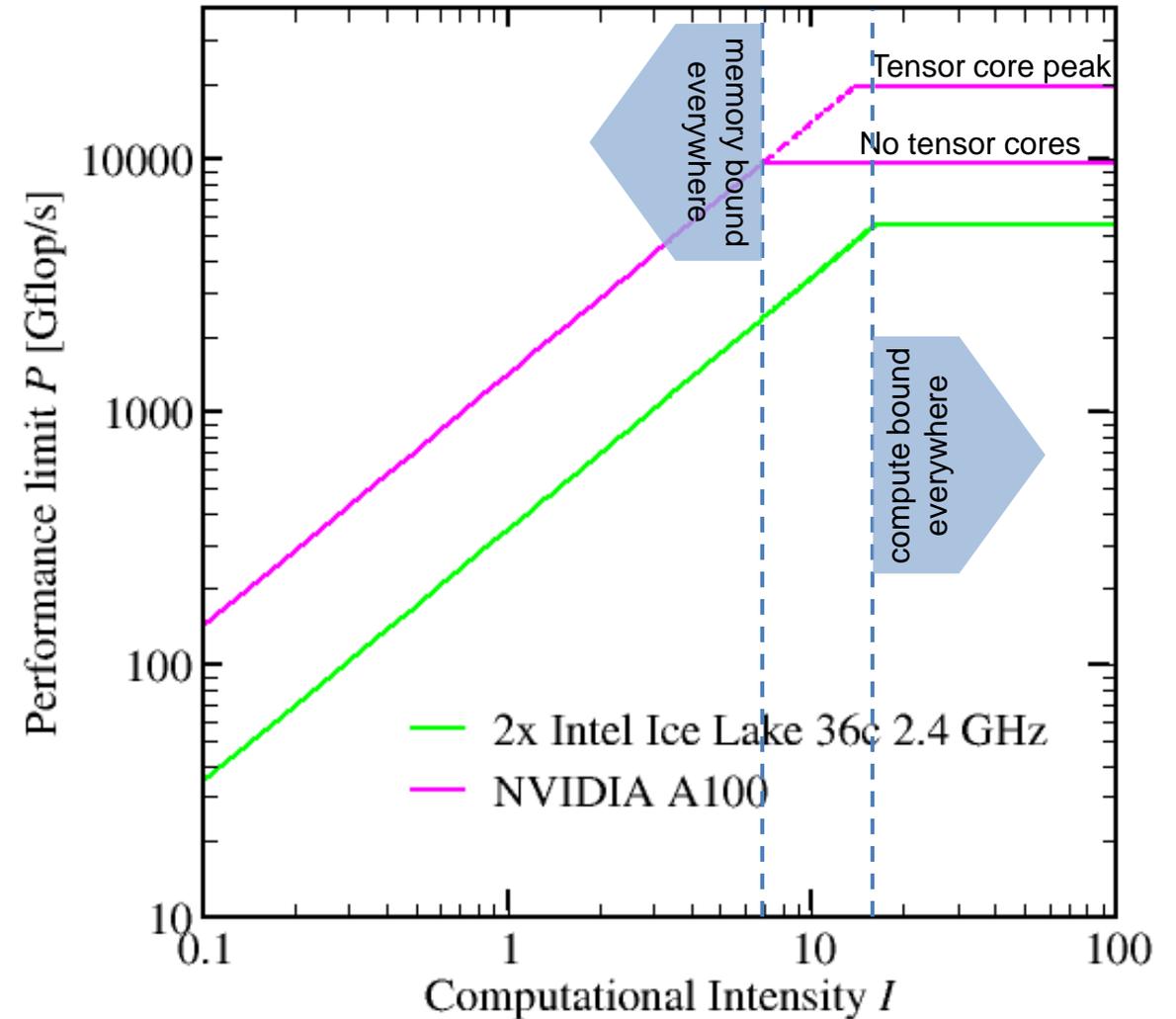
- Compare capabilities of different machines
- Compare performance expectations for different loops



Roofline for architecture and code comparison

With Roofline, we can

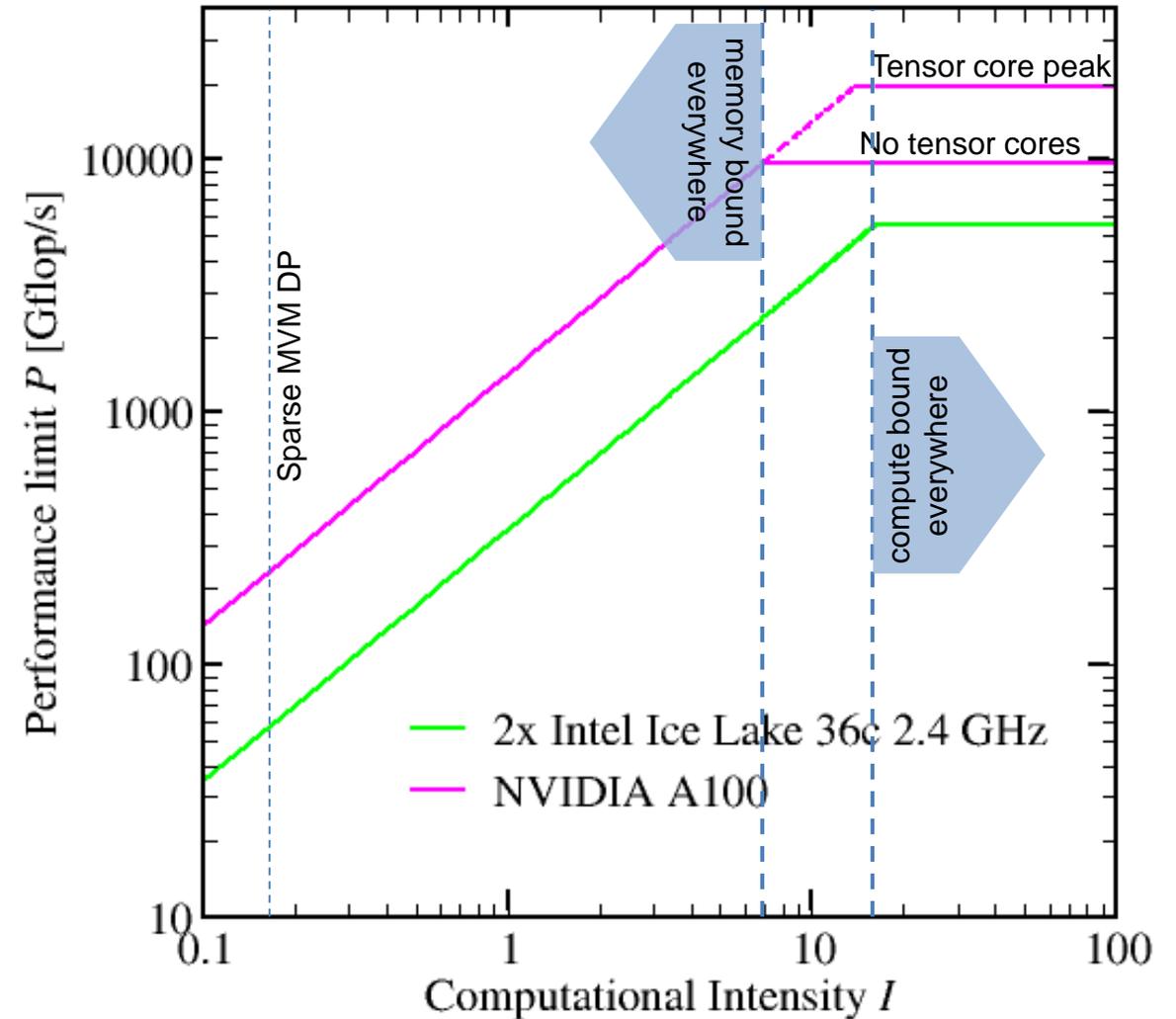
- Compare capabilities of different machines
- Compare performance expectations for different loops



Roofline for architecture and code comparison

With Roofline, we can

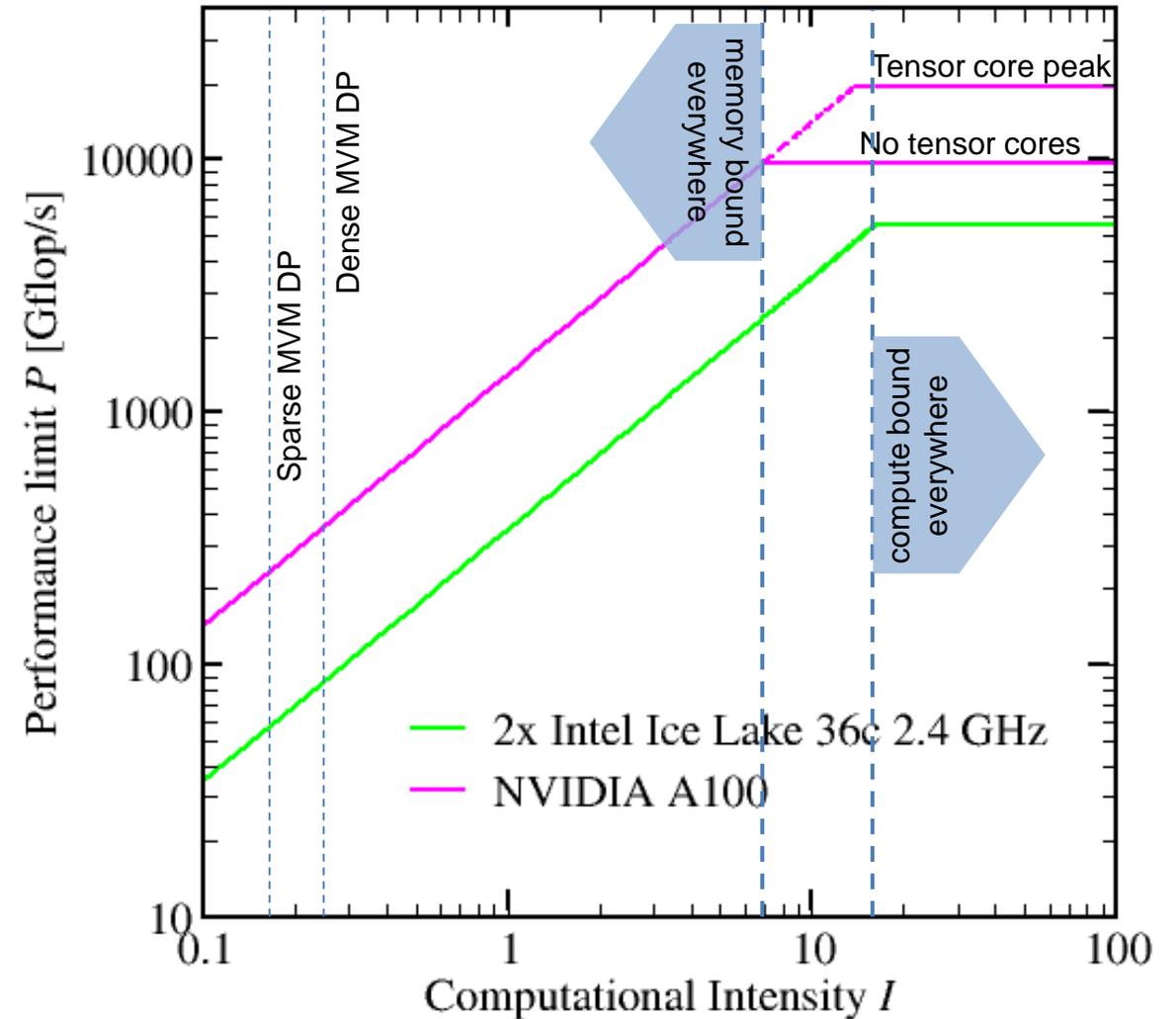
- Compare capabilities of different machines
- Compare performance expectations for different loops



Roofline for architecture and code comparison

With Roofline, we can

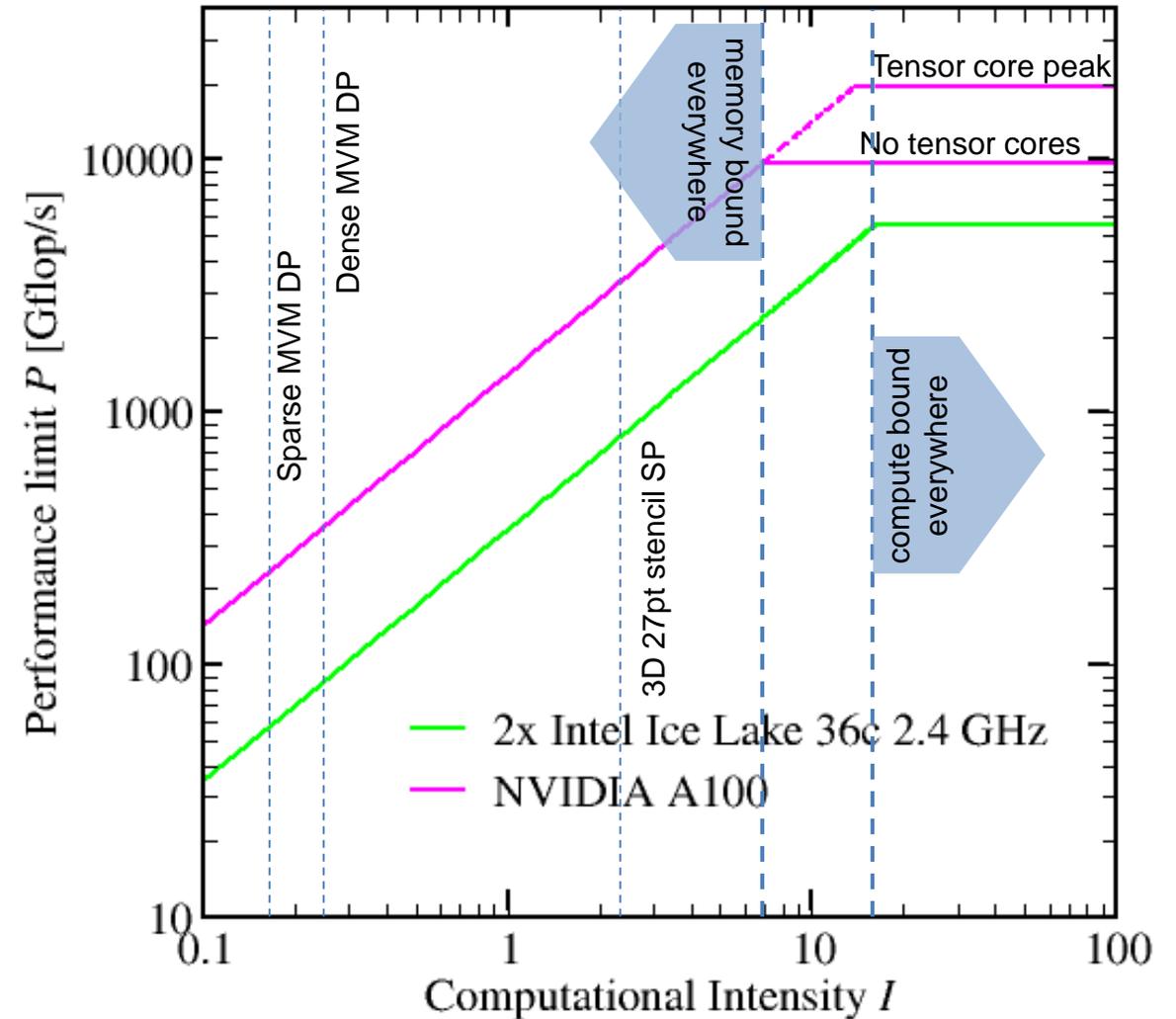
- Compare capabilities of different machines
- Compare performance expectations for different loops



Roofline for architecture and code comparison

With Roofline, we can

- Compare capabilities of different machines
- Compare performance expectations for different loops

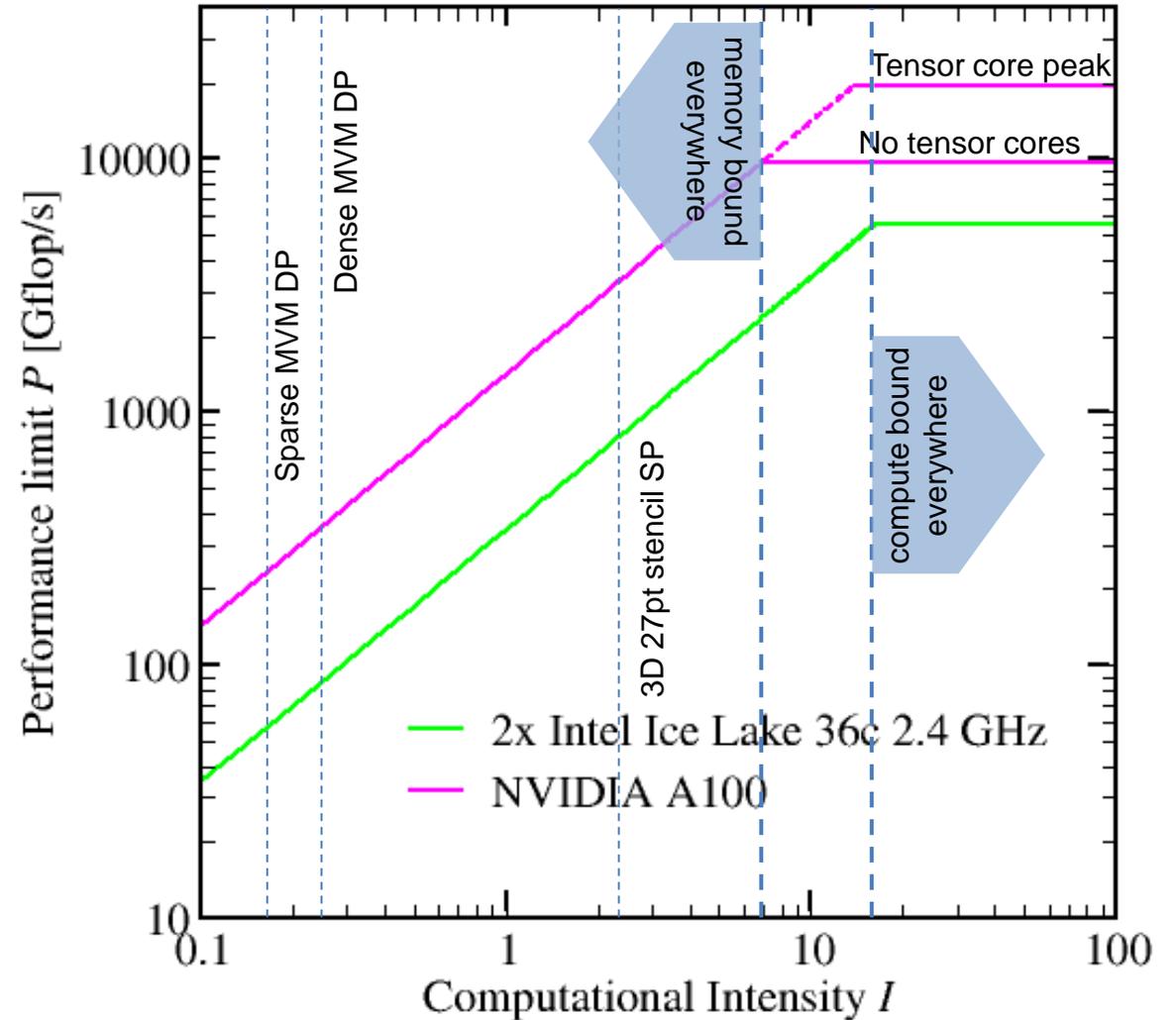


Roofline for architecture and code comparison

With Roofline, we can

- Compare capabilities of different machines
- Compare performance expectations for different loops

- Roofline always provides upper bound – but is it realistic?
 - Simple case: Loop kernel has loop-carried dependencies → cannot achieve peak
 - Other bandwidth bottlenecks may apply



A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s

A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s
2. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
→ e.g., $I = 0.167$ Flop/Byte → $B_C = 6$ Byte/Flop

A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s
2. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
→ e.g., $I = 0.167$ Flop/Byte → $B_C = 6$ Byte/Flop

“Flop” is not the only
useful unit of work!

A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s
2. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
→ e.g., $I = 0.167$ Flop/Byte → $B_C = 6$ Byte/Flop
3. b_S = Applicable (saturated) peak bandwidth of the slowest data path utilized
→ e.g., $b_S = 56$ GByte/s

“Flop” is not the only
useful unit of work!

A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s
2. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
→ e.g., $I = 0.167$ Flop/Byte → $B_C = 6$ Byte/Flop
3. b_S = Applicable (saturated) peak bandwidth of the slowest data path utilized
→ e.g., $b_S = 56$ GByte/s

“Flop” is not the only useful unit of work!

Performance limit:

$$P = \min(P_{\max}, I \cdot b_S) = \min\left(P_{\max}, \frac{b_S}{B_C}\right)$$

[Byte/s] (pointing to b_S)
[Byte/Flop] (pointing to B_C)

A refined Roofline Model

1. P_{\max} = Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily P_{peak})
→ e.g., $P_{\max} = 176$ GFlop/s
2. I = Computational intensity (“work” per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
→ e.g., $I = 0.167$ Flop/Byte → $B_C = 6$ Byte/Flop
3. b_S = Applicable (saturated) peak bandwidth of the slowest data path utilized
→ e.g., $b_S = 56$ GByte/s

“Flop” is not the only useful unit of work!

Performance limit:

$$P = \min(P_{\max}, I \cdot b_S) = \min\left(P_{\max}, \frac{b_S}{B_C}\right)$$

[Byte/s] (pointing to b_S)
[Byte/Flop] (pointing to B_C)

R.W. Hockney and I.J. Curington: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. Parallel Computing 10, 277-286 (1989). DOI: 10.1016/0167-8191(89)90100-2

W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. Self-edition (2000)

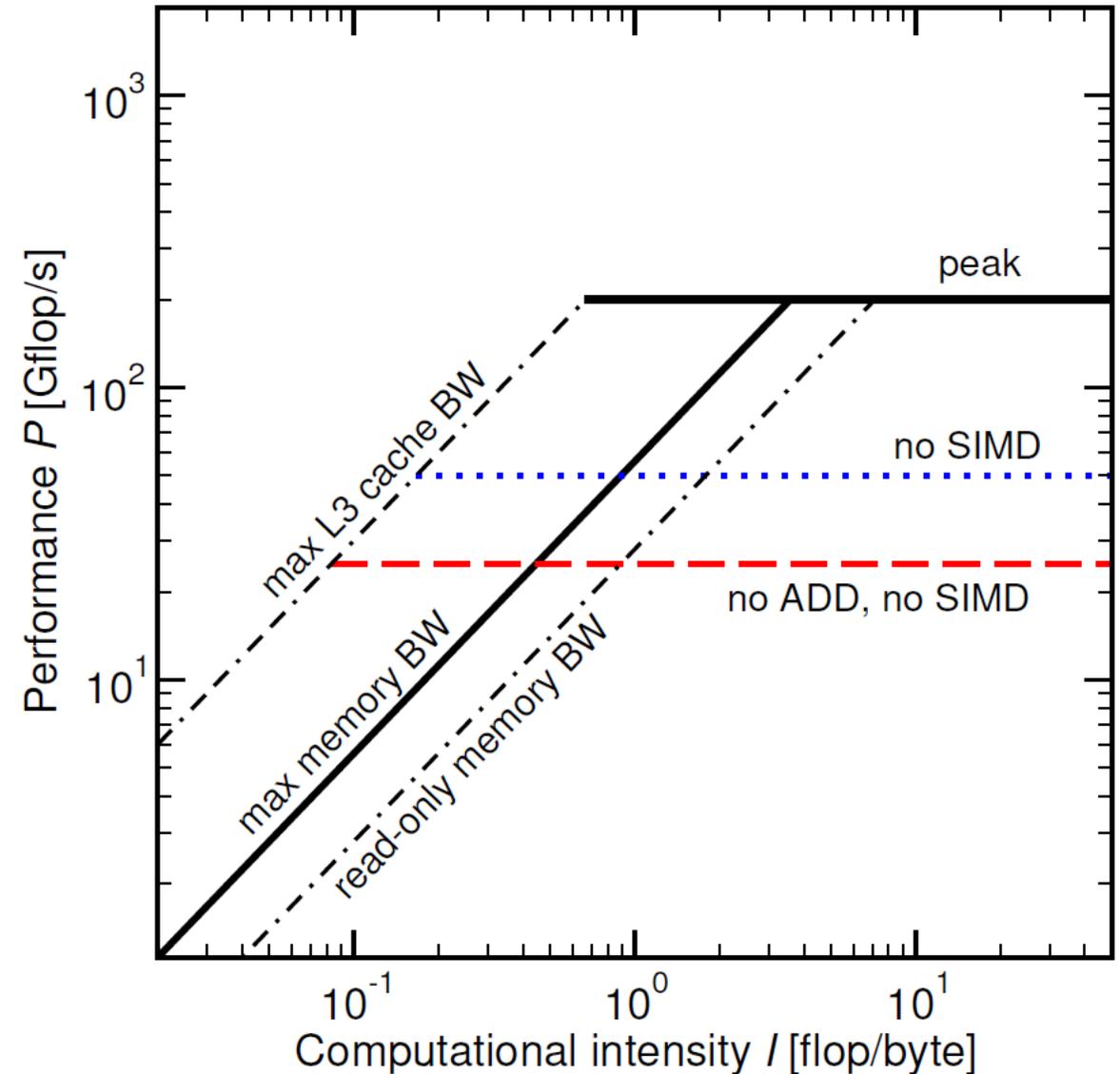
S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

Refined Roofline models: graphical representation

Multiple ceilings may apply

- Different **bandwidths / data paths**
→ different inclined ceilings
- Different P_{\max}
→ different flat ceilings

In fact, P_{\max} should always come from **code analysis**; generic ceilings are usually impossible to attain



Hardware features of (some) Intel Xeon processors

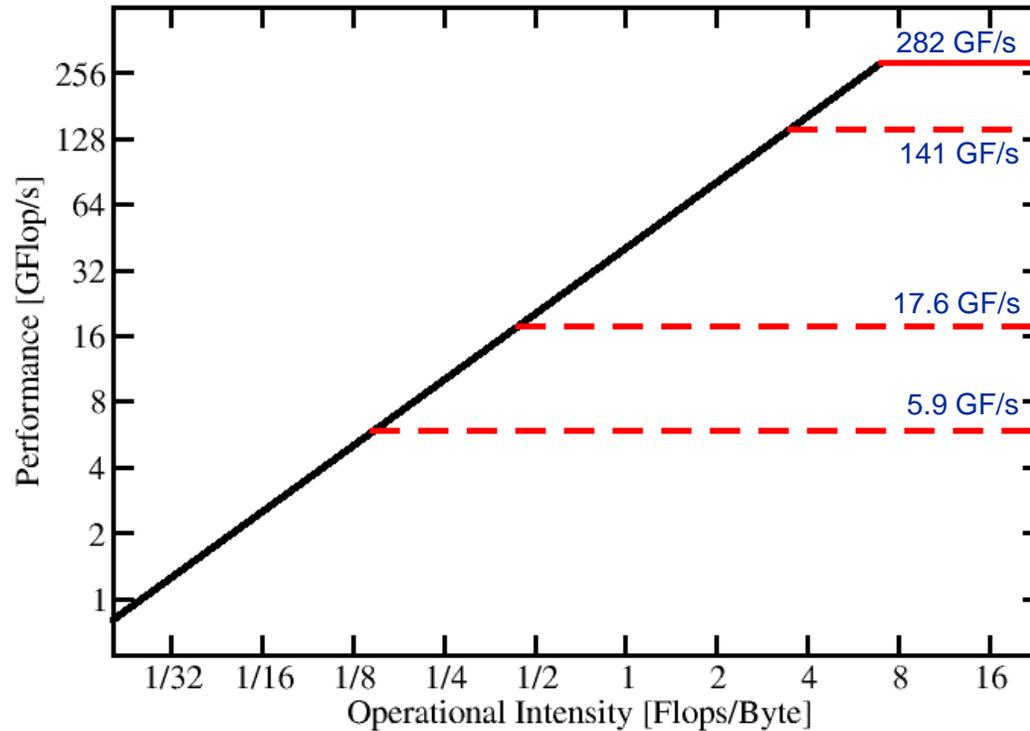
Microarchitecture	Ivy Bridge EP	Broadwell EP	Cascade Lake SP	Ice Lake SP
Introduced	09/2013	03/2016	04/2019	06/2021
Cores	≤ 12	≤ 22	≤ 28	≤ 40
LD/ST throughput per cy:				
AVX(2), AVX512	1 LD + ½ ST	2 LD + 1 ST	2 LD + 1 ST	2 LD + 1 ST
SSE/scalar	2 LD 1 LD & 1 ST			
ADD throughput	1 / cy	1 / cy	2 / cy	2 / cy
MUL throughput	1 / cy	2 / cy	2 / cy	2 / cy
FMA throughput	N/A	2 / cy	2 / cy	2 / cy
L1-L2 data bus	32 B/cy	64 B/cy	64 B/cy	64 B/cy
L2-L3 data bus	32 B/cy	32 B/cy	16+16 B/cy	16+16 B/cy
L1/L2 per core	32 KiB / 256 KiB	32 KiB / 256 KiB	32 KiB / 1 MiB	48 KiB / 1.25 MiB
LLC	2.5 MiB/core inclusive	2.5 MiB/core inclusive	1.375 MiB/core exclusive/victim	1.5 MiB/core exclusive/victim
Memory	4ch DDR3	4ch DDR3	6ch DDR4	8ch DDR4
Memory BW (meas.)	~ 48 GB/s	~ 62 GB/s	~ 115 GB/s	~ 160 GB/s

Source: <https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html>

A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

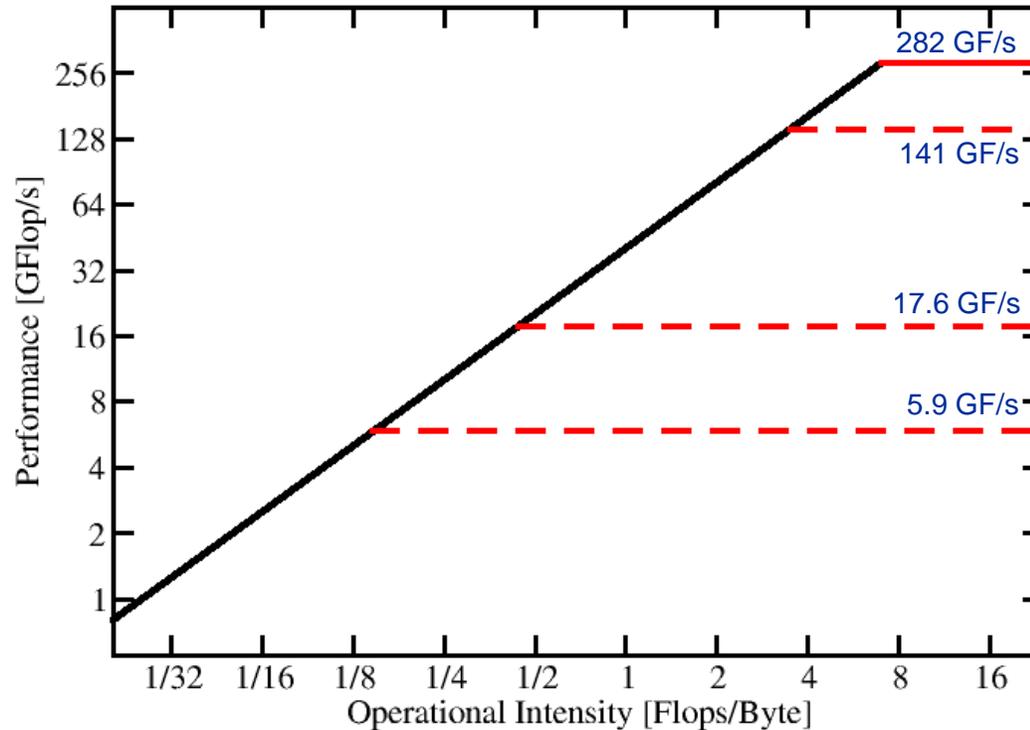


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_S)$$

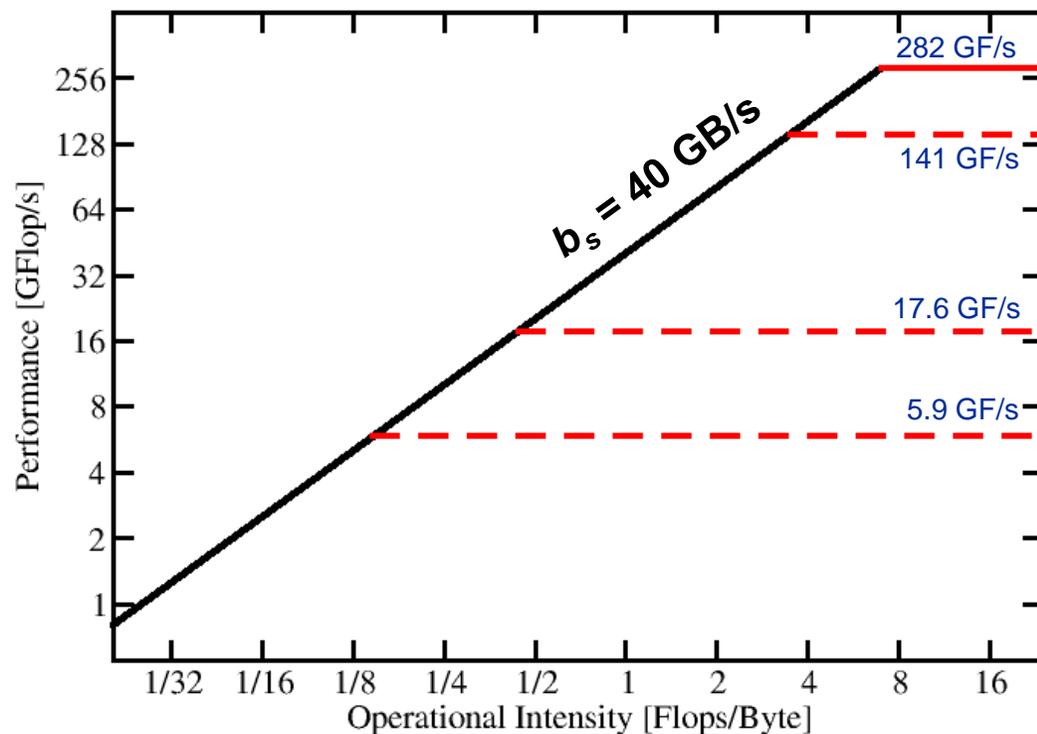


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_S)$$

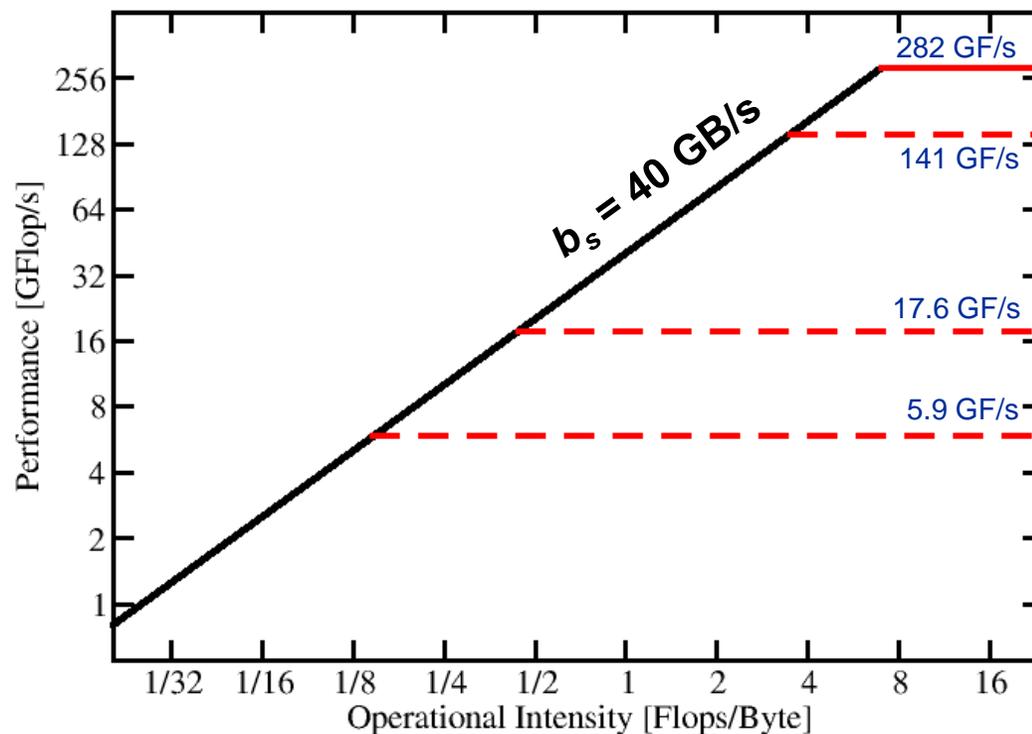


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$



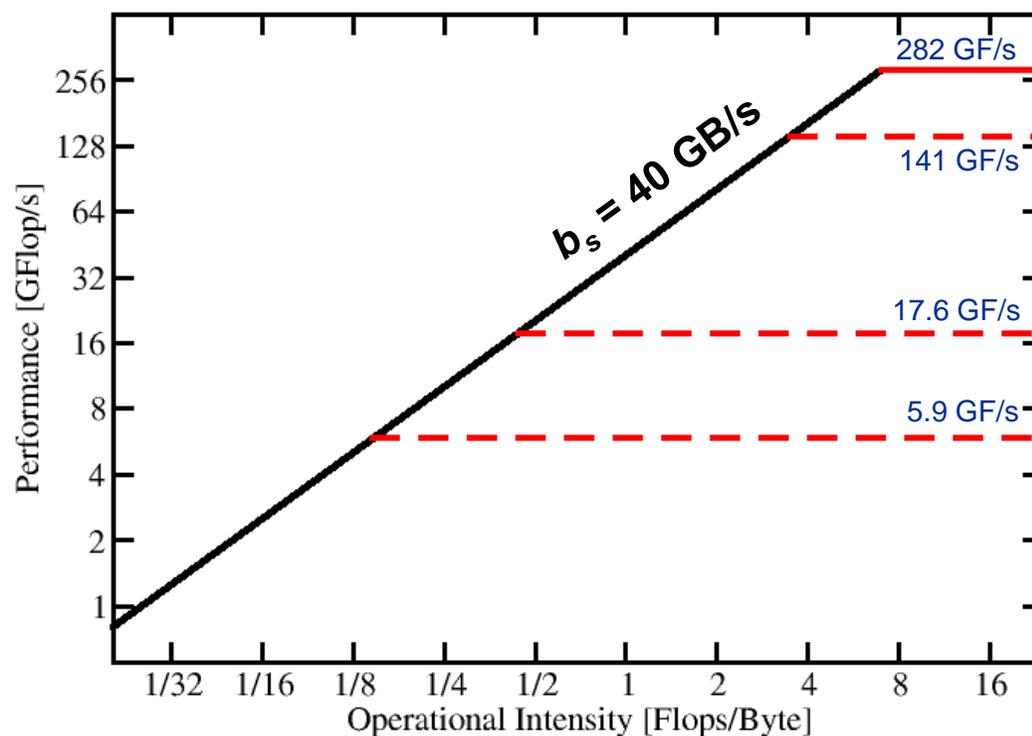
Machine peak
(ADD+MULT)
Out of reach for this
code

A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$



Machine peak
(ADD+MULT)
Out of reach for this
code

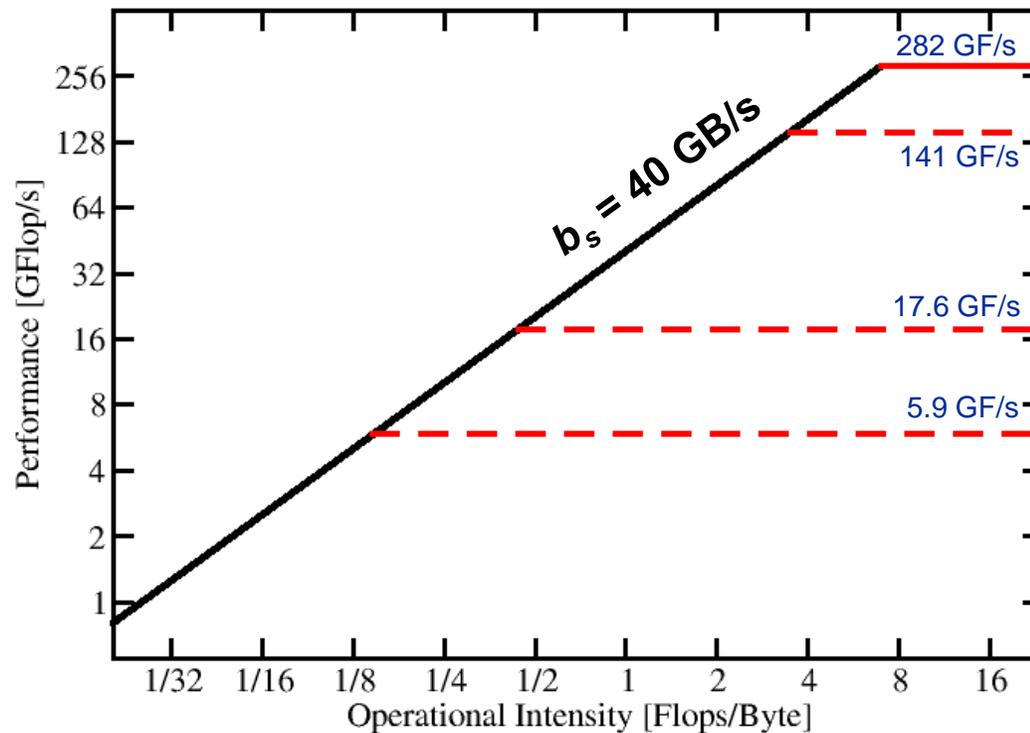
ADD peak
(best possible
code)

A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$



Machine peak
(ADD+MULT)
Out of reach for this
code

ADD peak
(best possible
code)

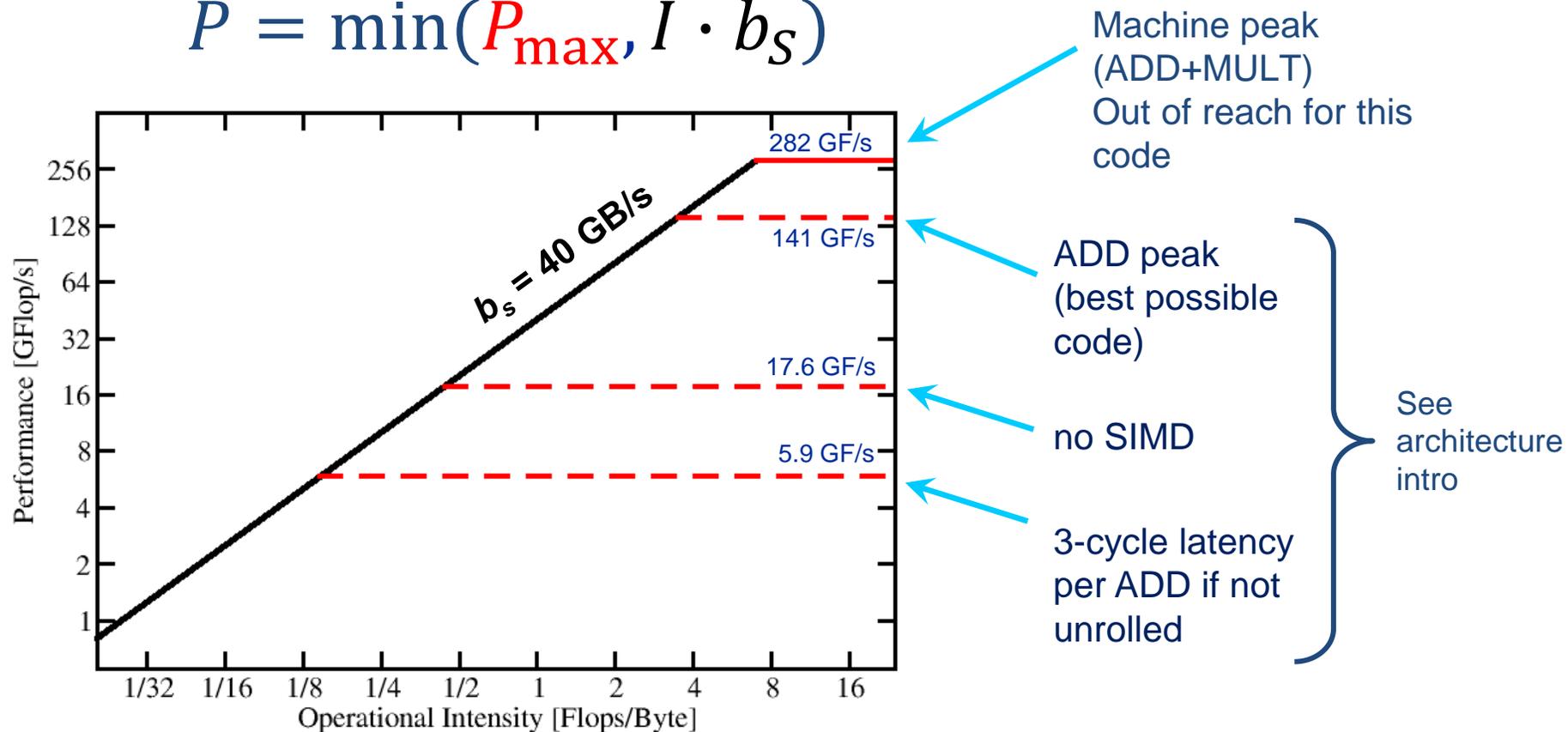
no SIMD

A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$

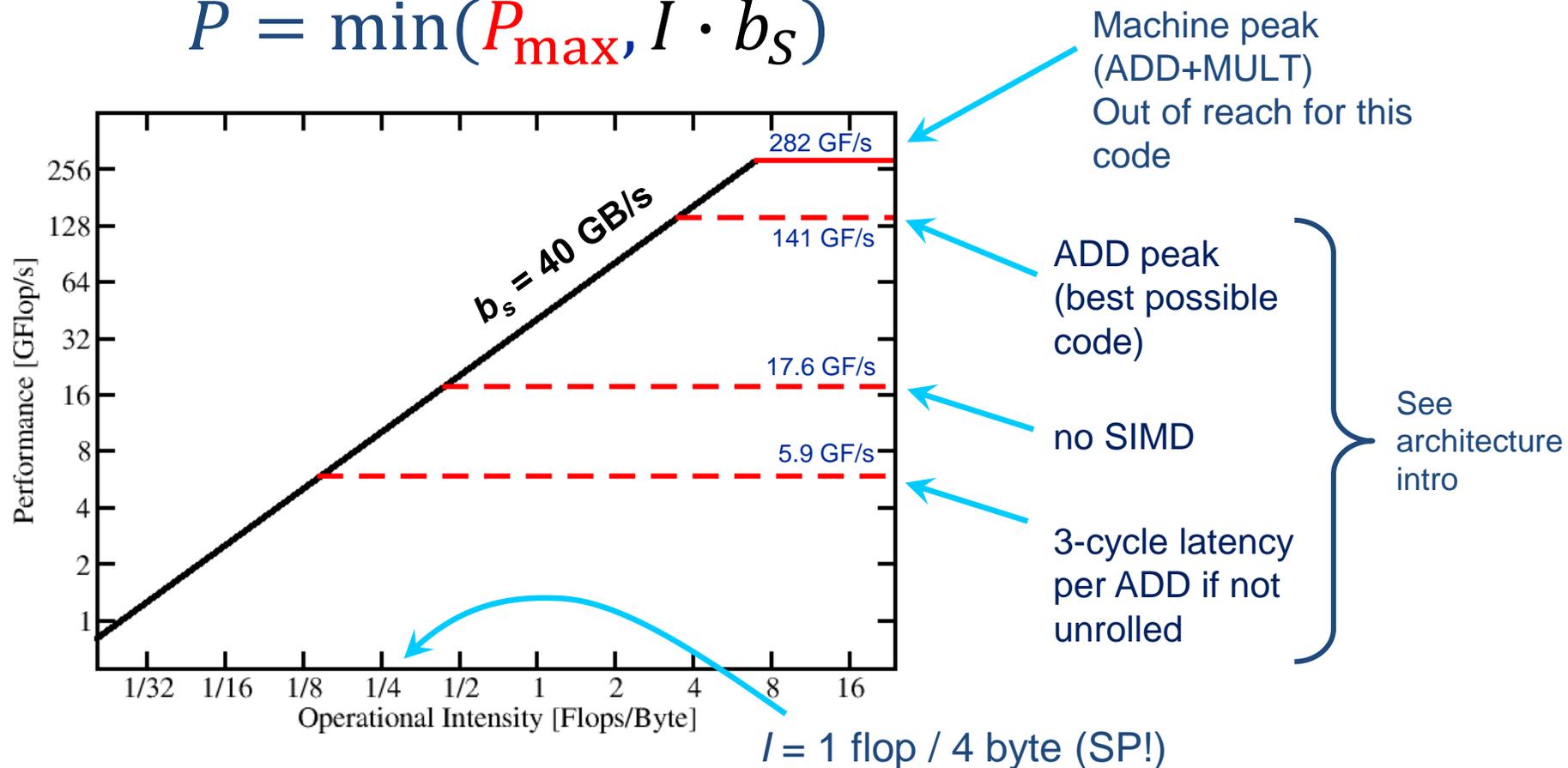


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$

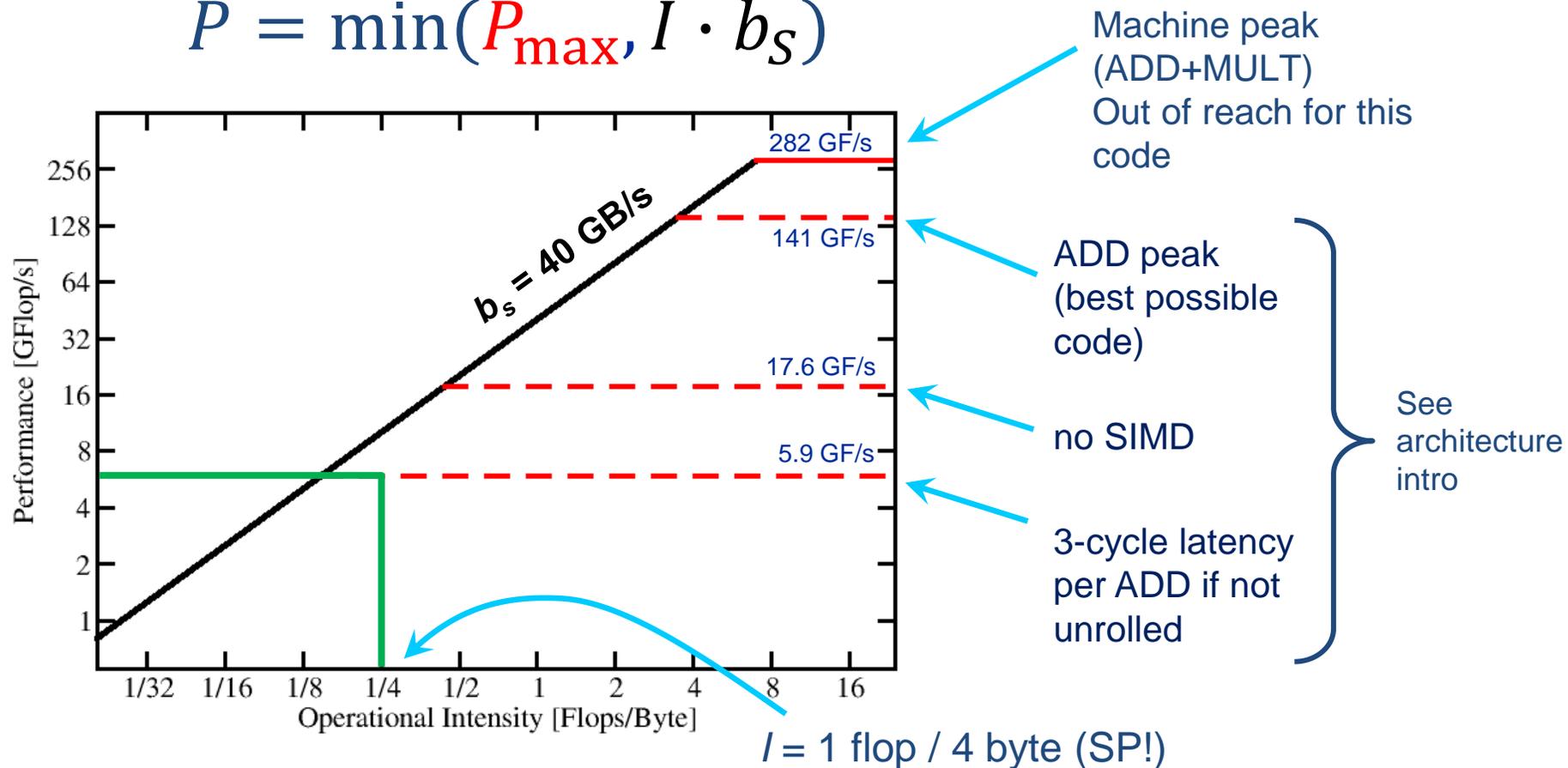


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$

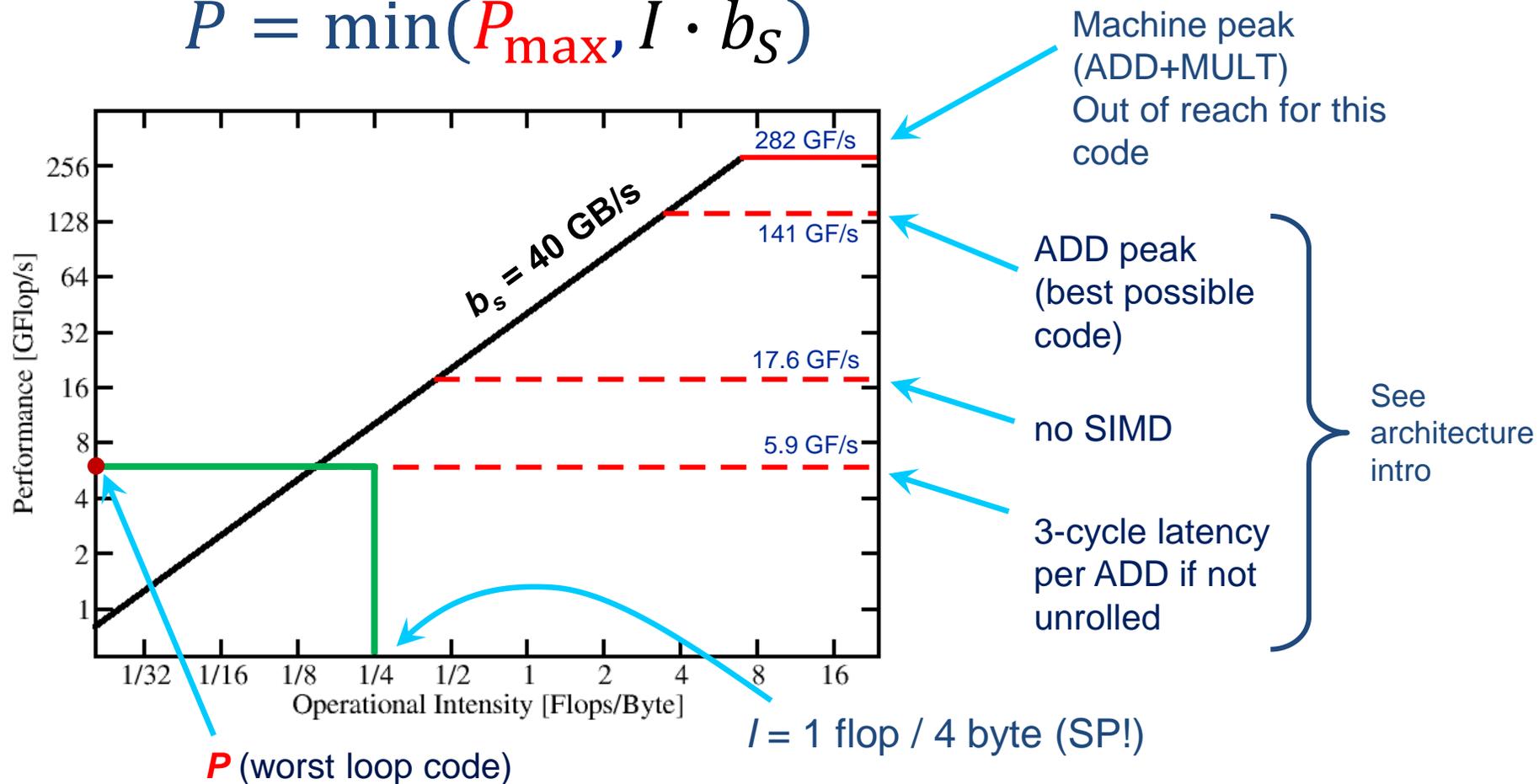


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ "large" N

$$P = \min(P_{\max}, I \cdot b_s)$$

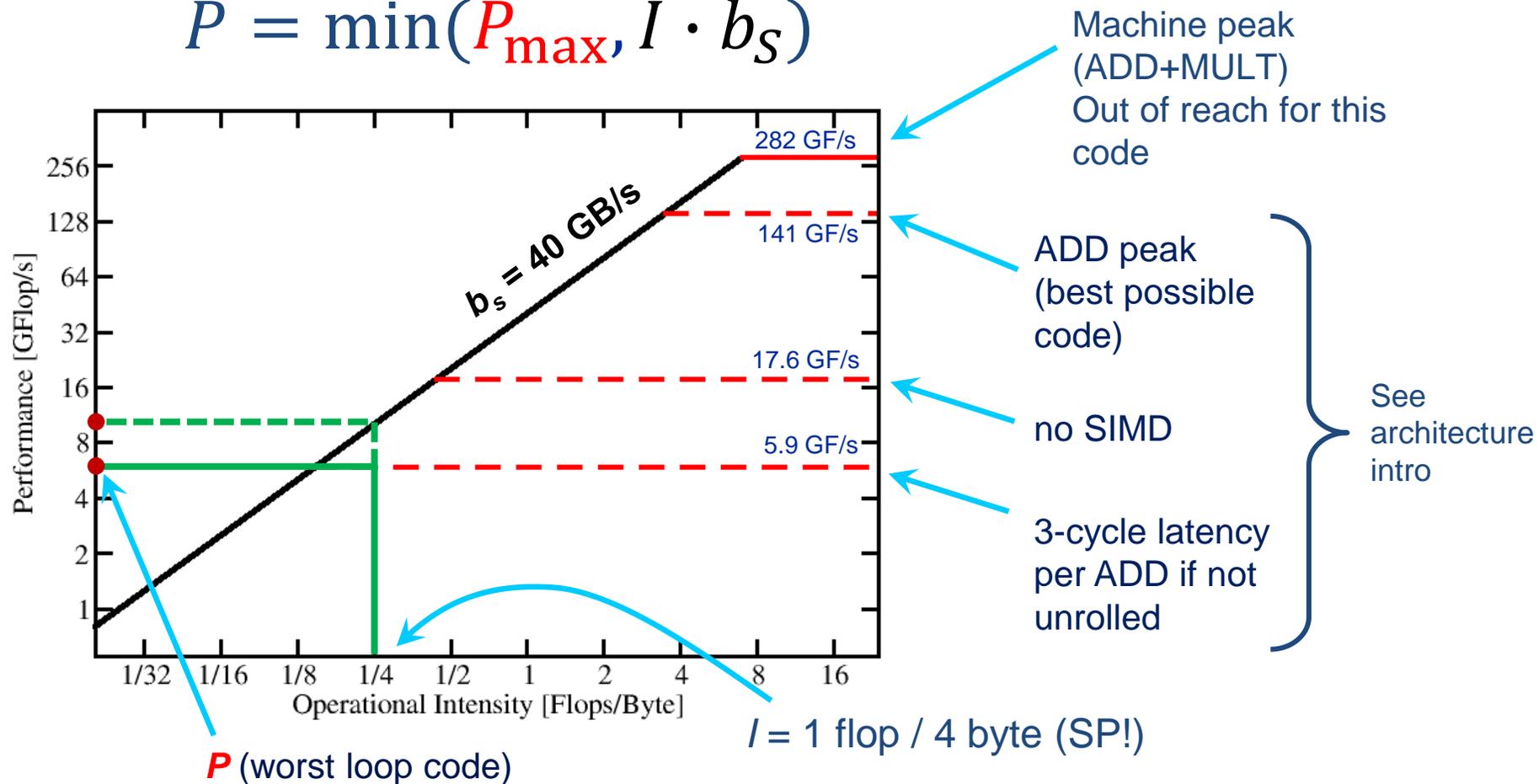


A not so simple Roofline example

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

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

$$P = \min(P_{\max}, I \cdot b_s)$$

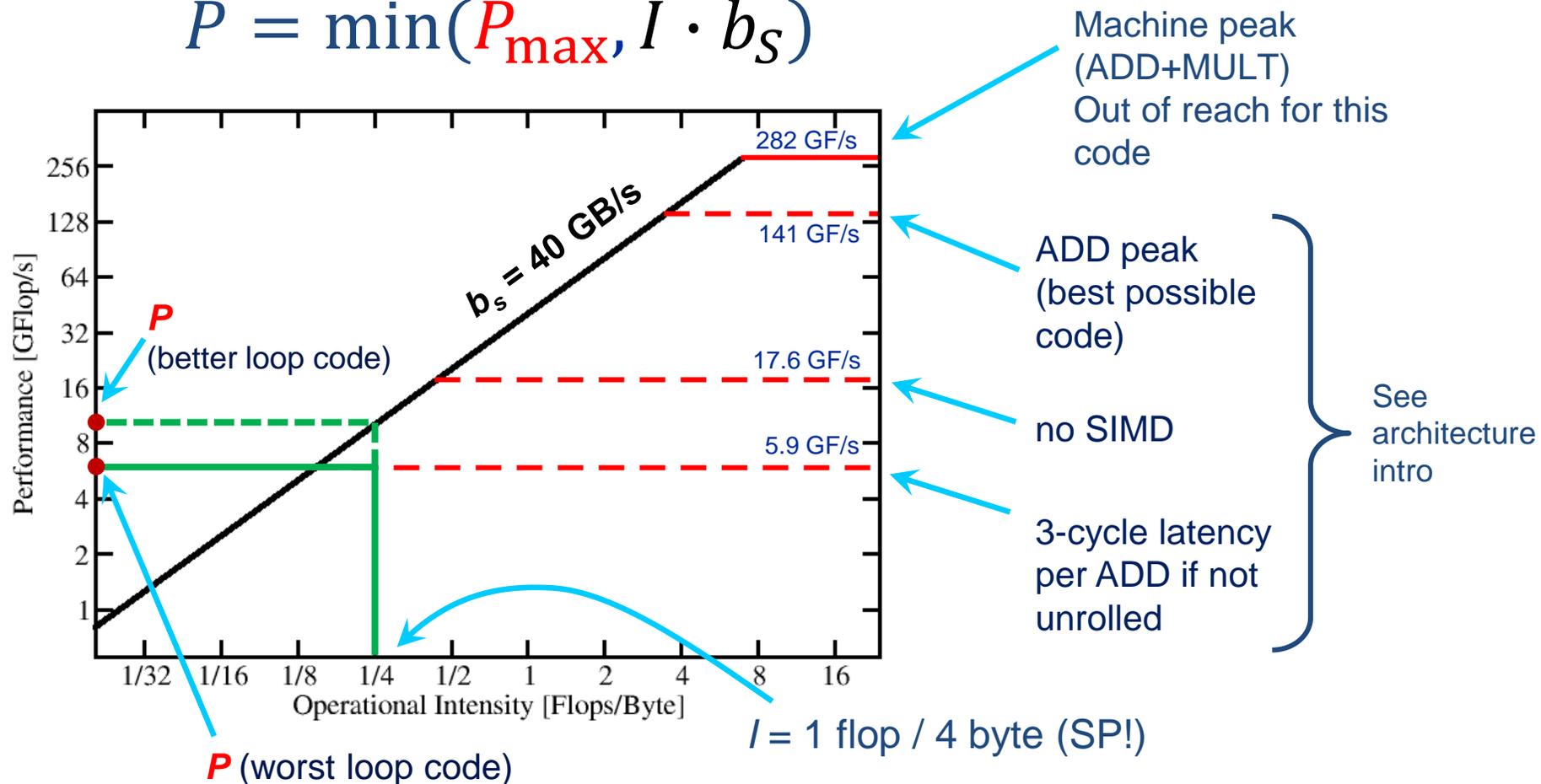


A not so simple Roofline example

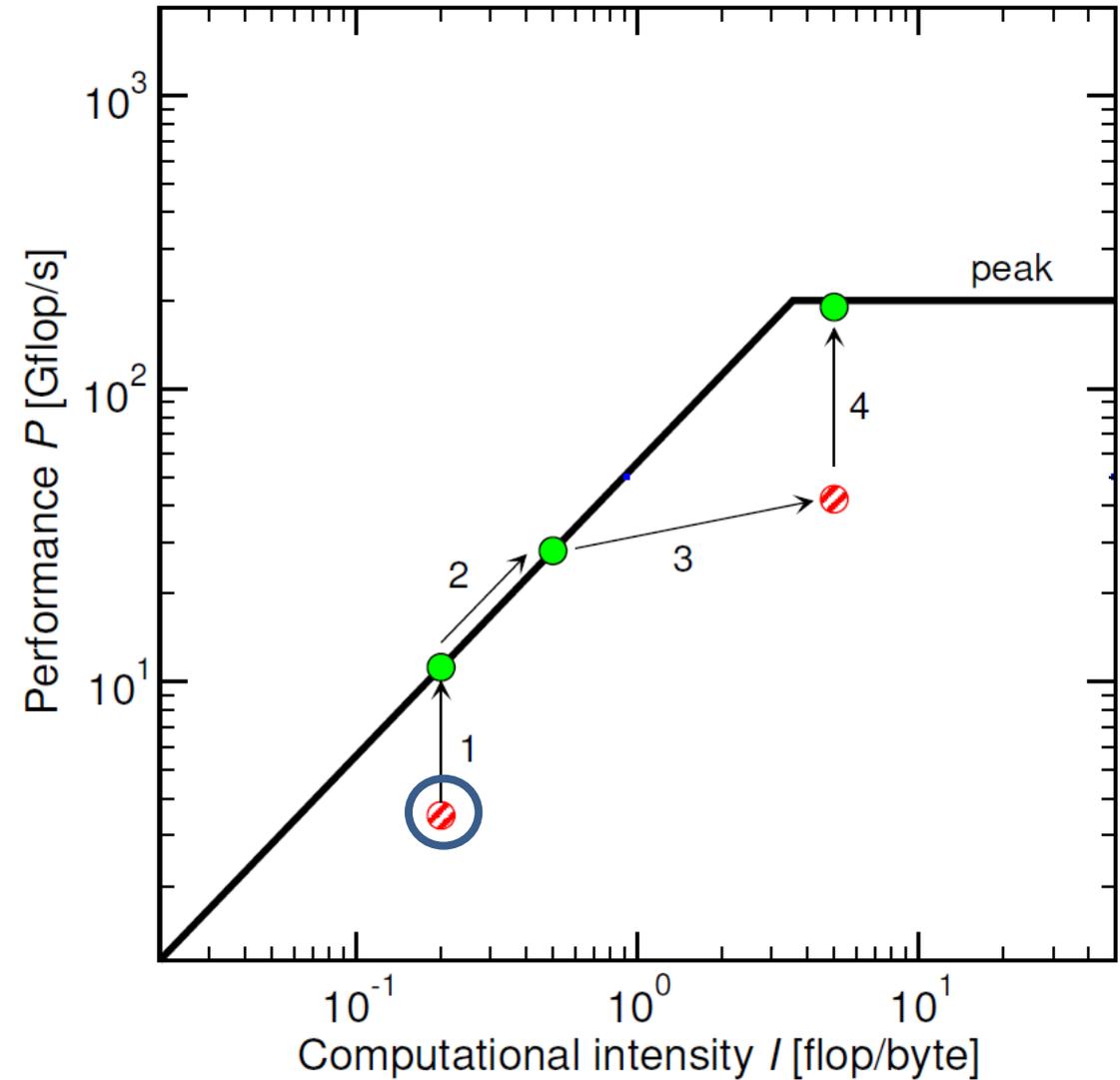
Example: `do i=1,N; s=s+a(i); enddo`

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

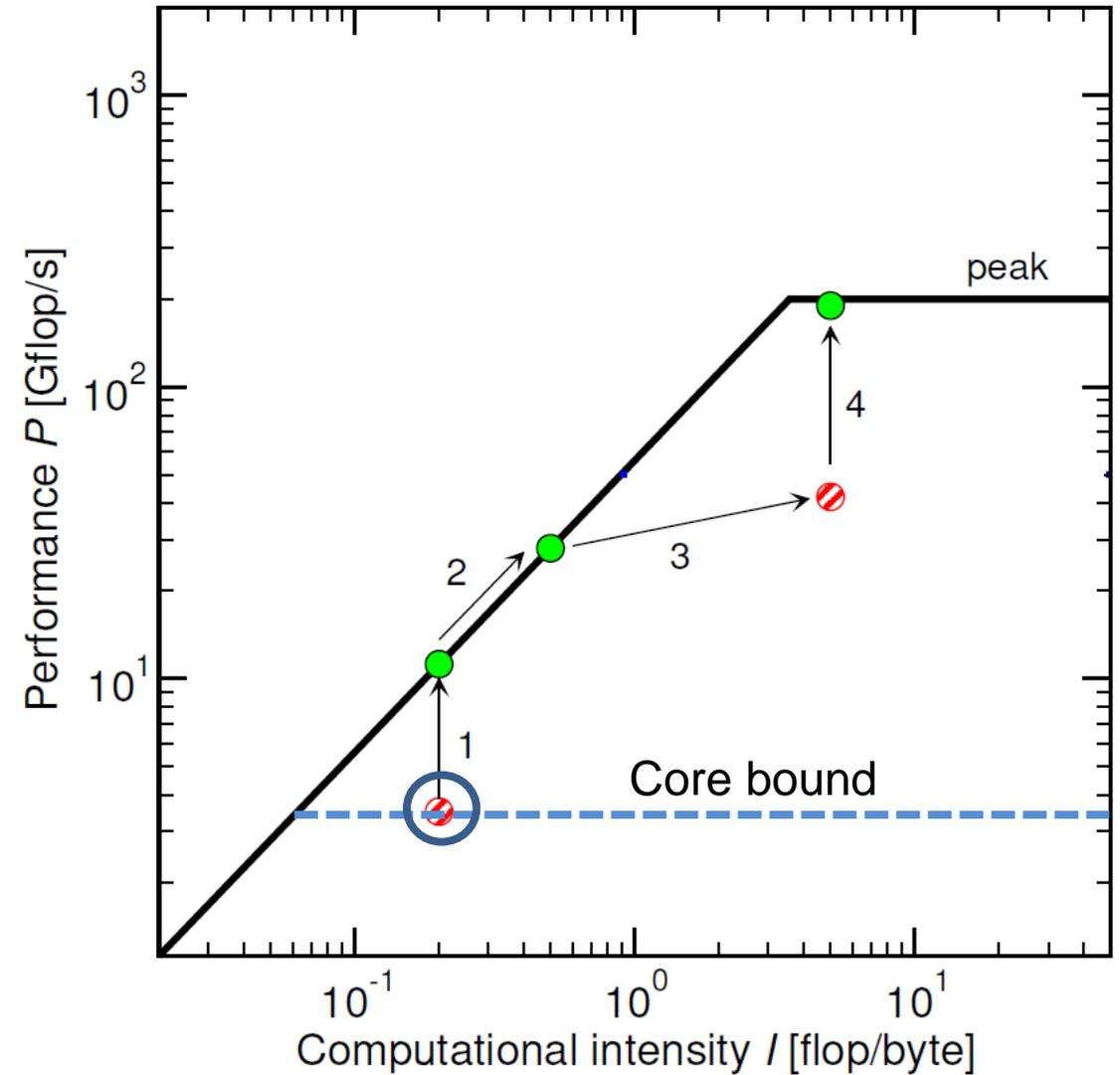
$$P = \min(P_{\max}, I \cdot b_s)$$



Tracking code optimizations in the Roofline Model

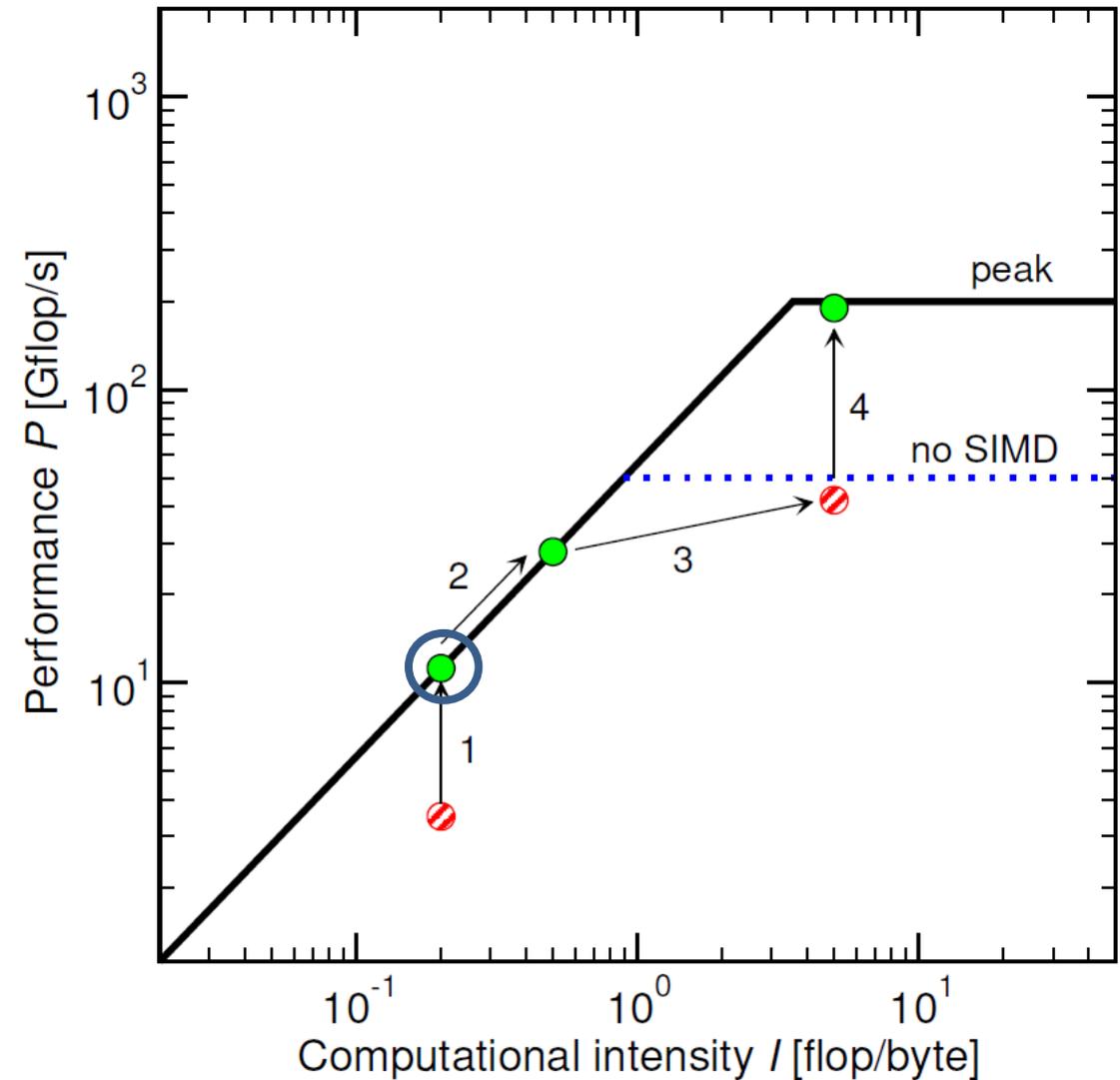


Tracking code optimizations in the Roofline Model



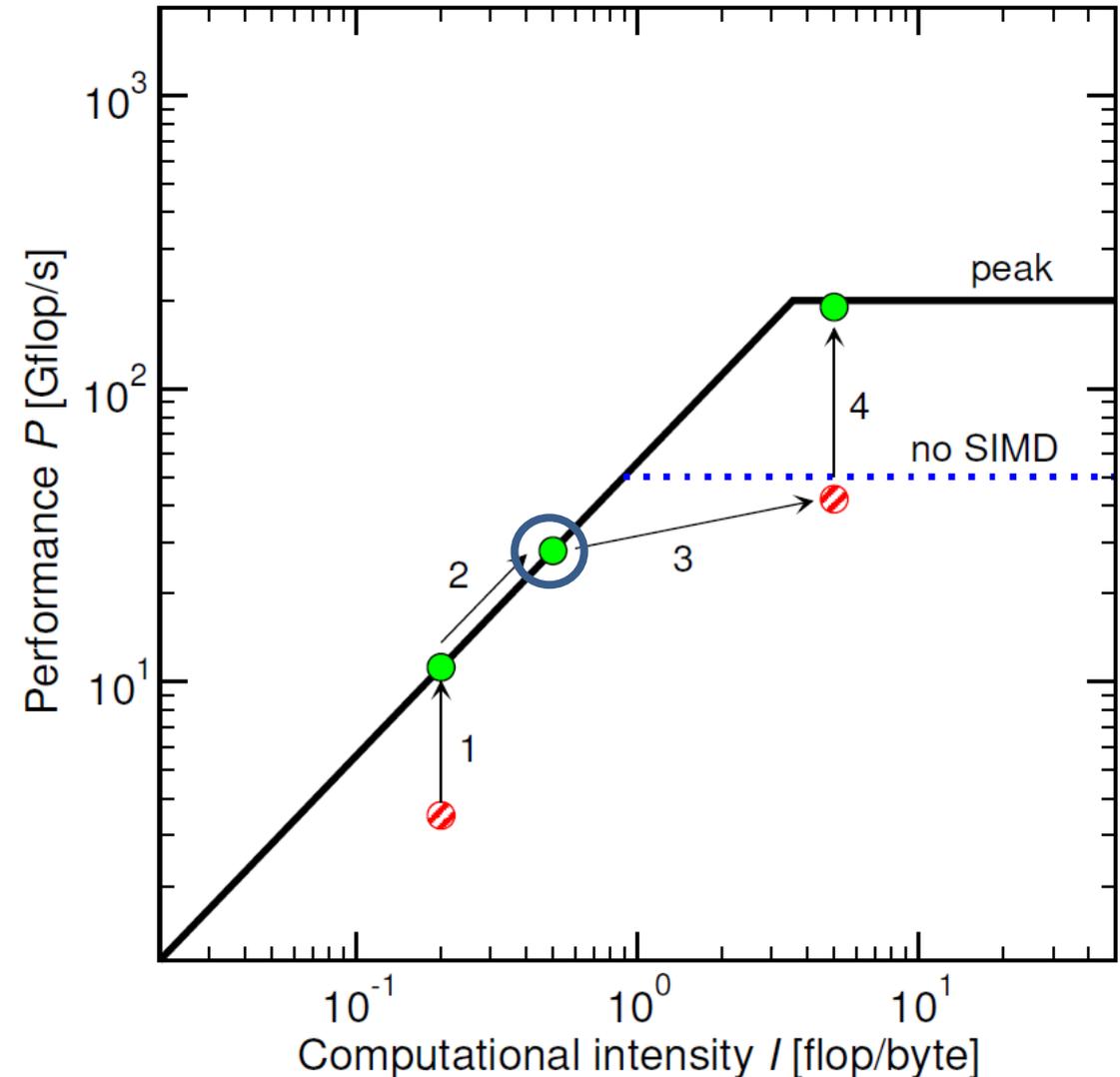
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
(e.g., plain Python → Fortran)



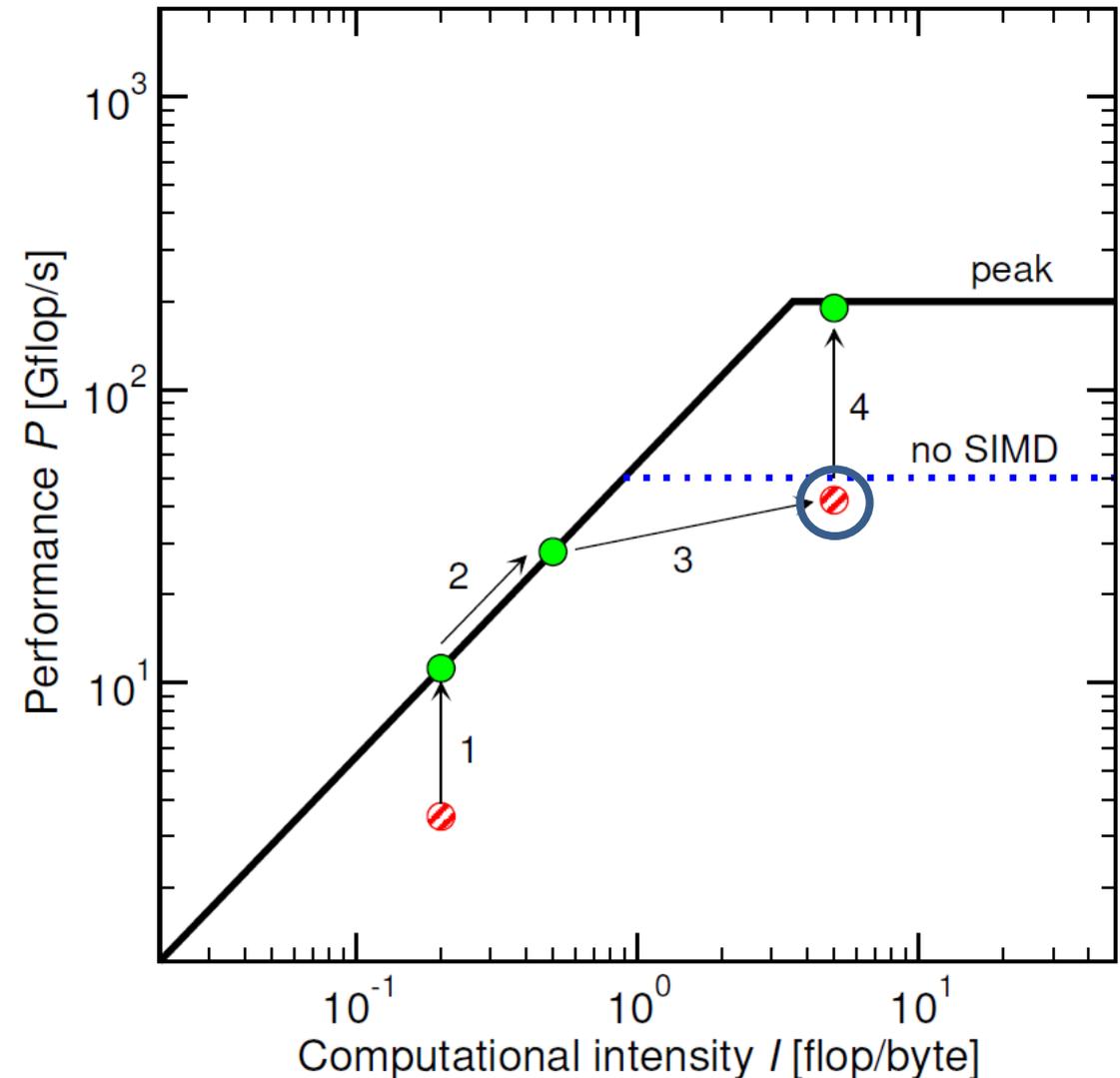
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
(e.g., plain Python → Fortran)
2. Increase intensity to make better use of BW bottleneck
(e.g., spatial loop blocking)



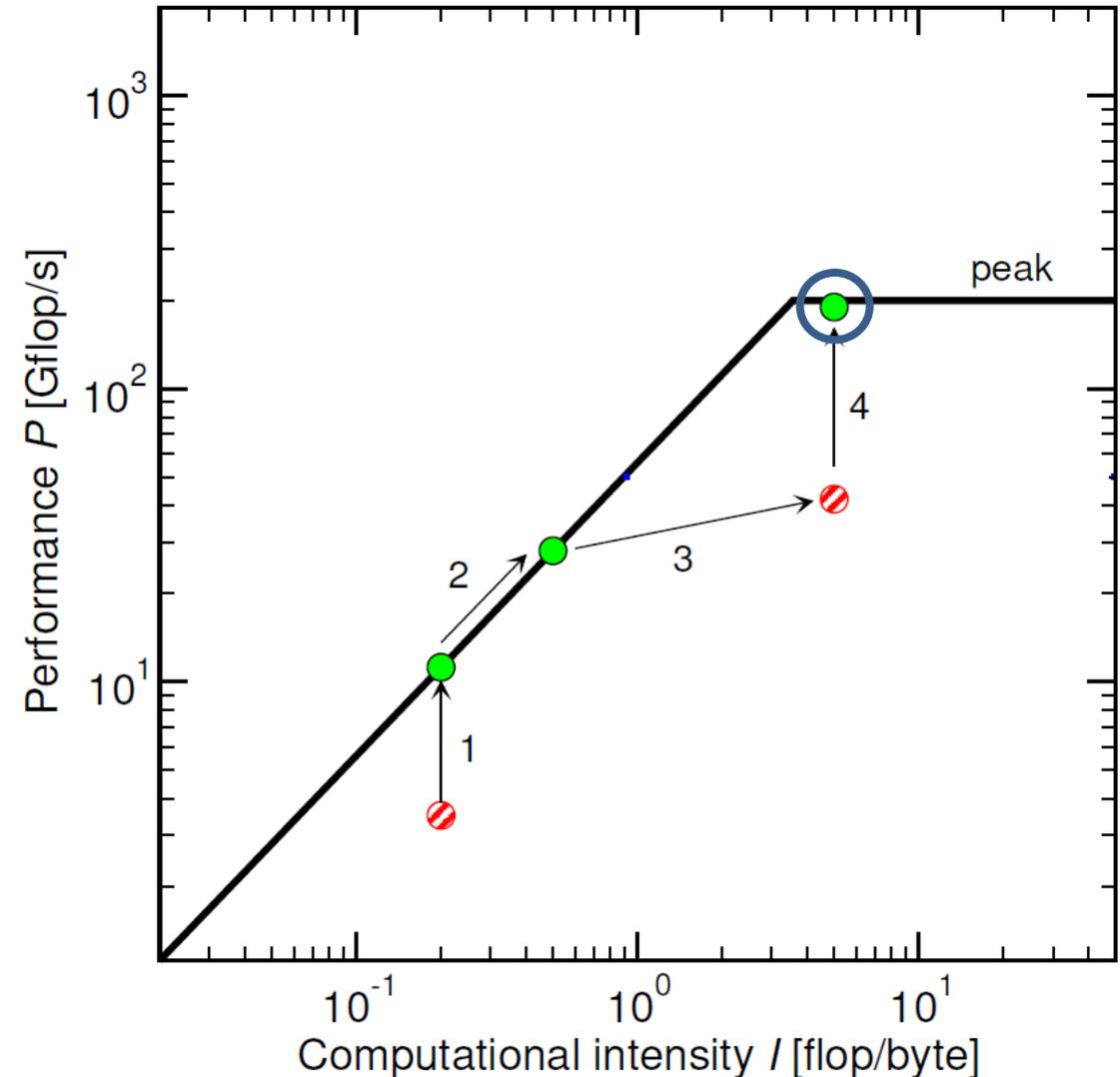
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
(e.g., plain Python → Fortran)
2. Increase intensity to make better use of BW bottleneck
(e.g., spatial loop blocking)
3. Increase intensity and go from memory bound to core bound
(e.g., temporal blocking)



Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
(e.g., plain Python → Fortran)
2. Increase intensity to make better use of BW bottleneck
(e.g., spatial loop blocking)
3. Increase intensity and go from memory bound to core bound
(e.g., temporal blocking)
4. Hit the core bottleneck by good serial code
(e.g., `-fno-alias`, SIMD intrinsics)



Diagnostic / phenomenological Roofline modeling

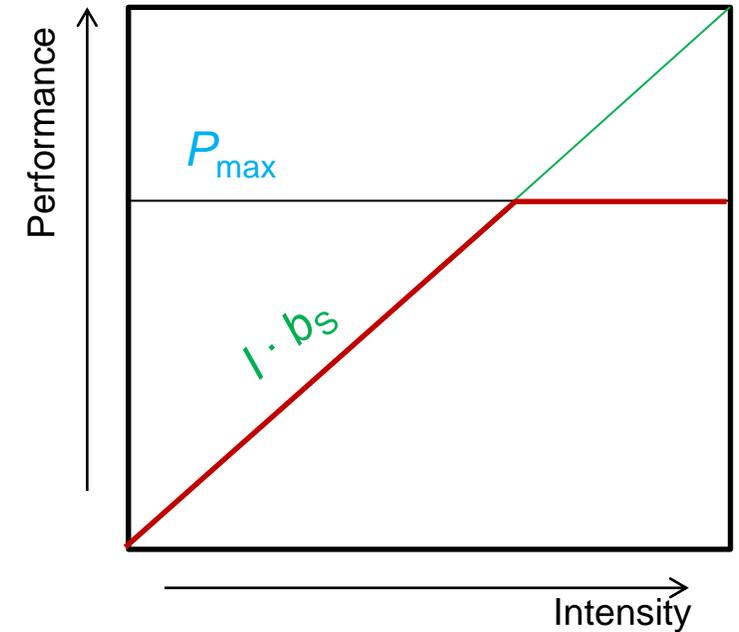


Diagnostic modeling

- What if we cannot predict the intensity/balance?
 - Code very complicated
 - Code not available
 - Parameters unknown
 - Doubts about correctness of analysis

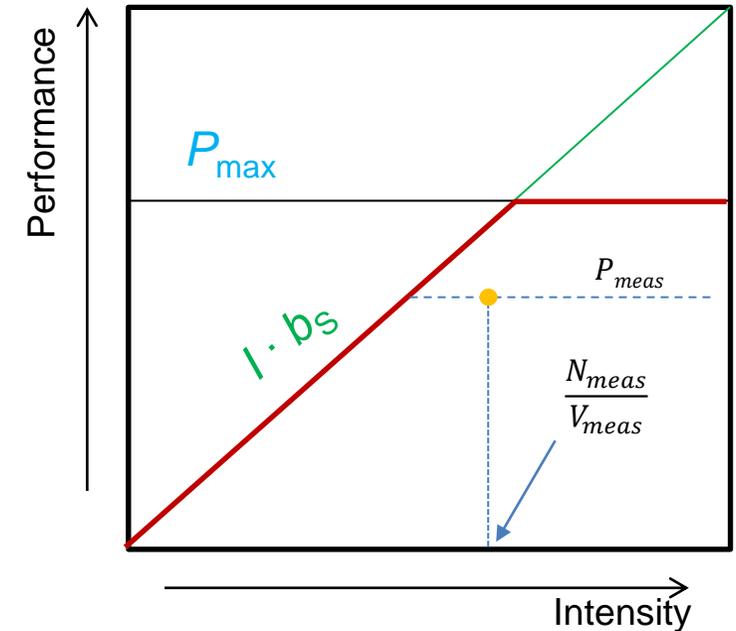
Diagnostic modeling

- What if we cannot predict the intensity/balance?
 - Code very complicated
 - Code not available
 - Parameters unknown
 - Doubts about correctness of analysis



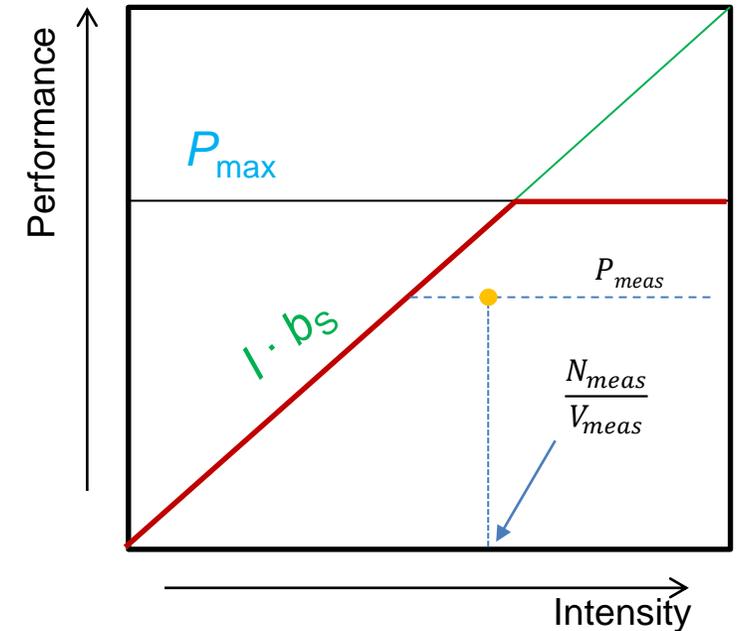
Diagnostic modeling

- What if we cannot predict the intensity/balance?
 - Code very complicated
 - Code not available
 - Parameters unknown
 - Doubts about correctness of analysis
- Measure data volume V_{meas} (and work N_{meas})
 - Hardware performance counters
 - Tools: likwid-perfctr, PAPI, Intel Vtune,...



Diagnostic modeling

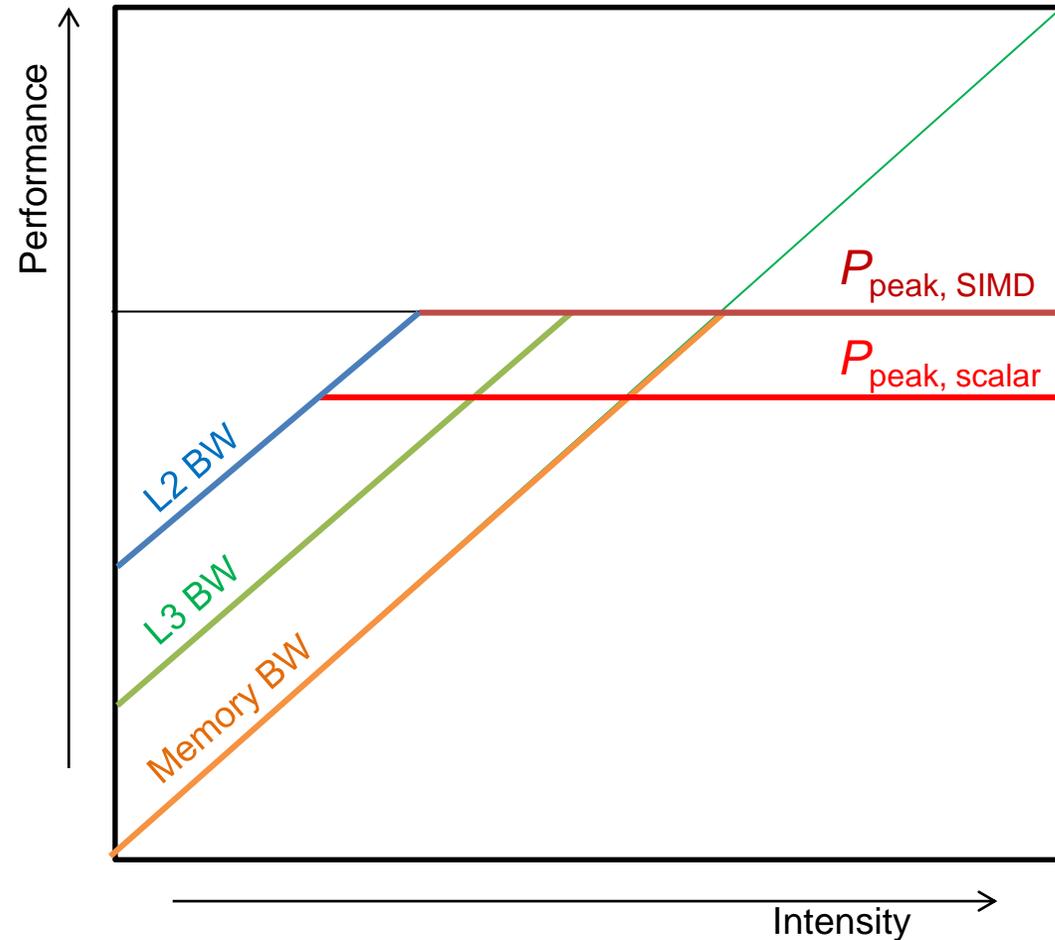
- What if we cannot predict the intensity/balance?
 - Code very complicated
 - Code not available
 - Parameters unknown
 - Doubts about correctness of analysis
- Measure data volume V_{meas} (and work N_{meas})
 - Hardware performance counters
 - Tools: likwid-perfctr, PAPI, Intel Vtune,...
- Insights + benefits
 - Compare analytic model and measurement → validate model
 - Can be applied (semi-)automatically
 - Useful in performance monitoring of user jobs on clusters



Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

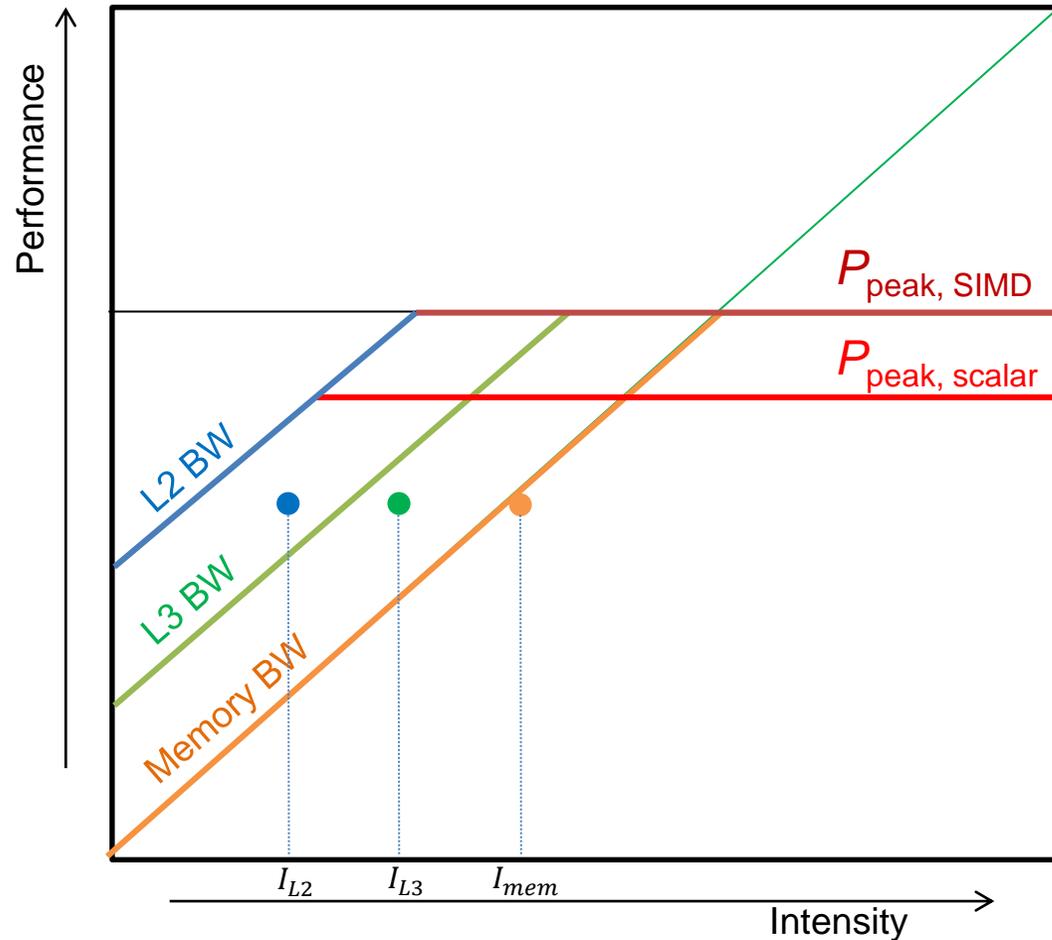
Kernel 2 ▲

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

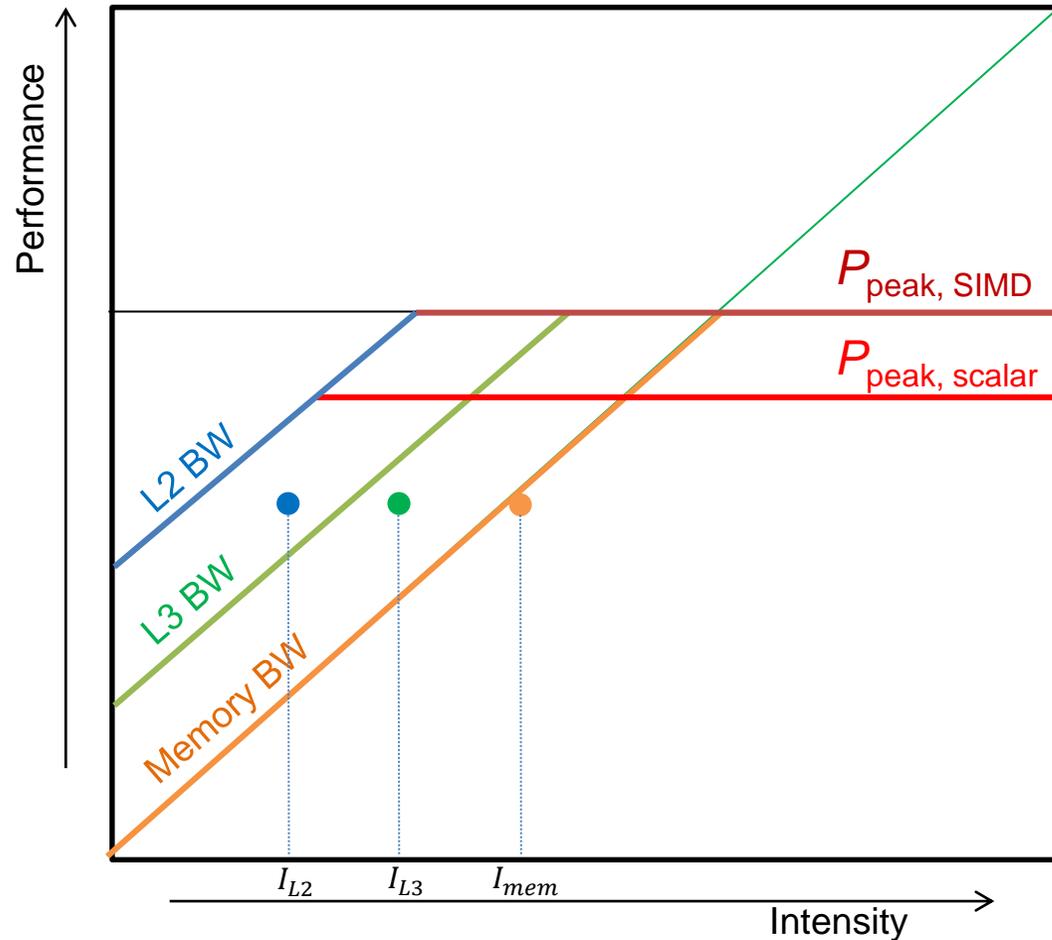
Kernel 2 ▲

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
→ indicates **memory bound**

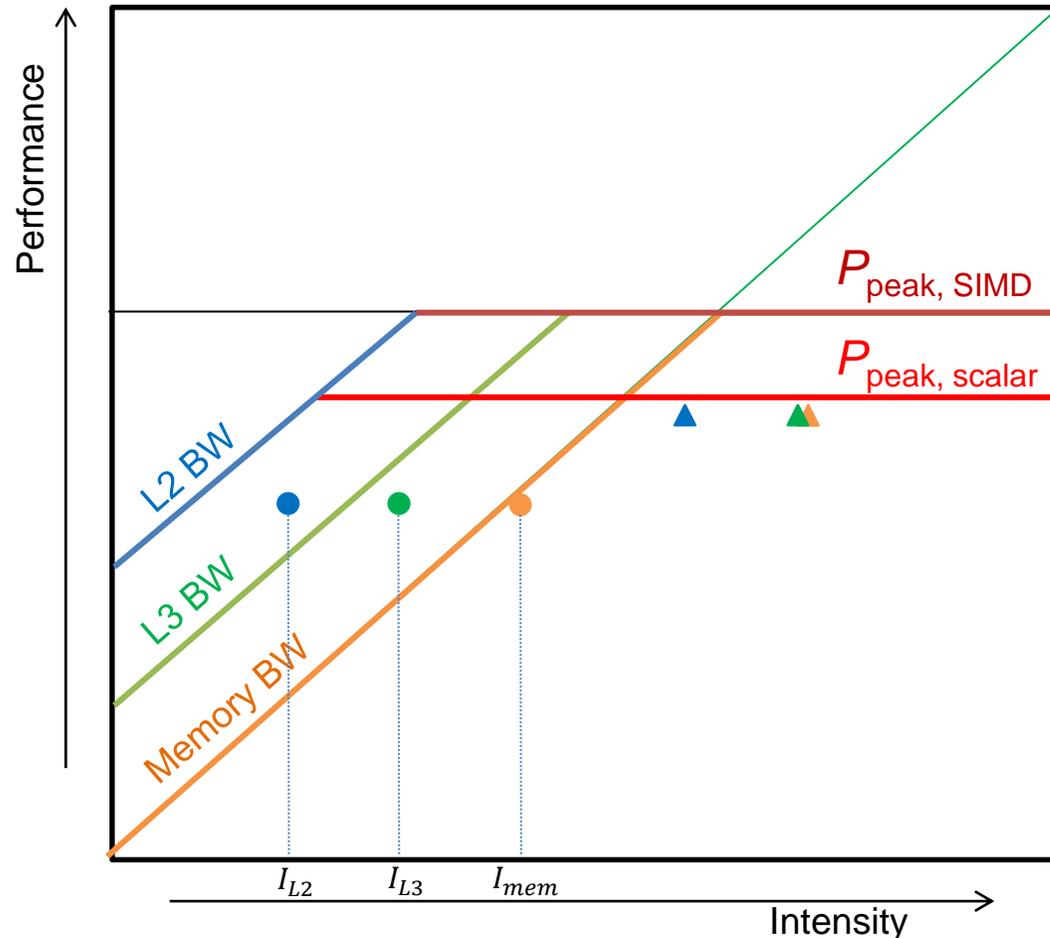
Kernel 2 ▲

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
→ indicates **memory bound**

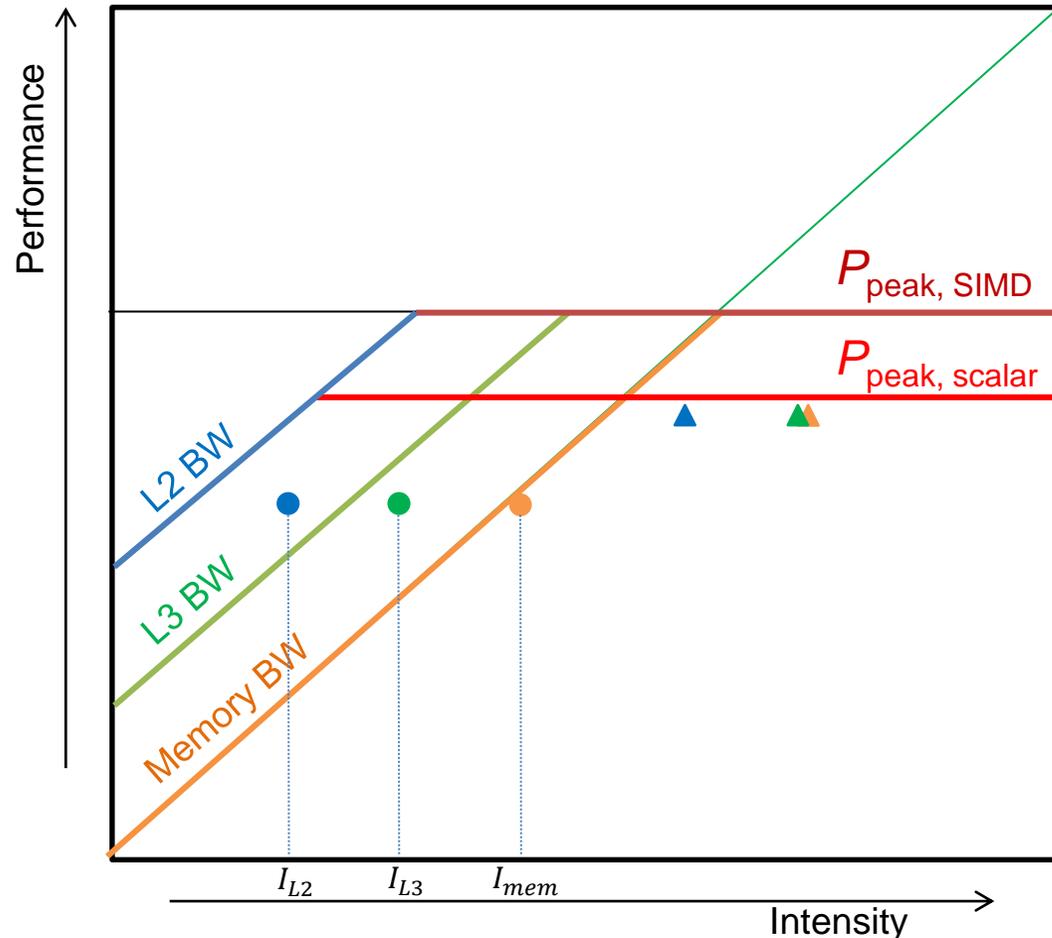
Kernel 2 ▲

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
→ indicates **memory bound**

Kernel 2 ▲

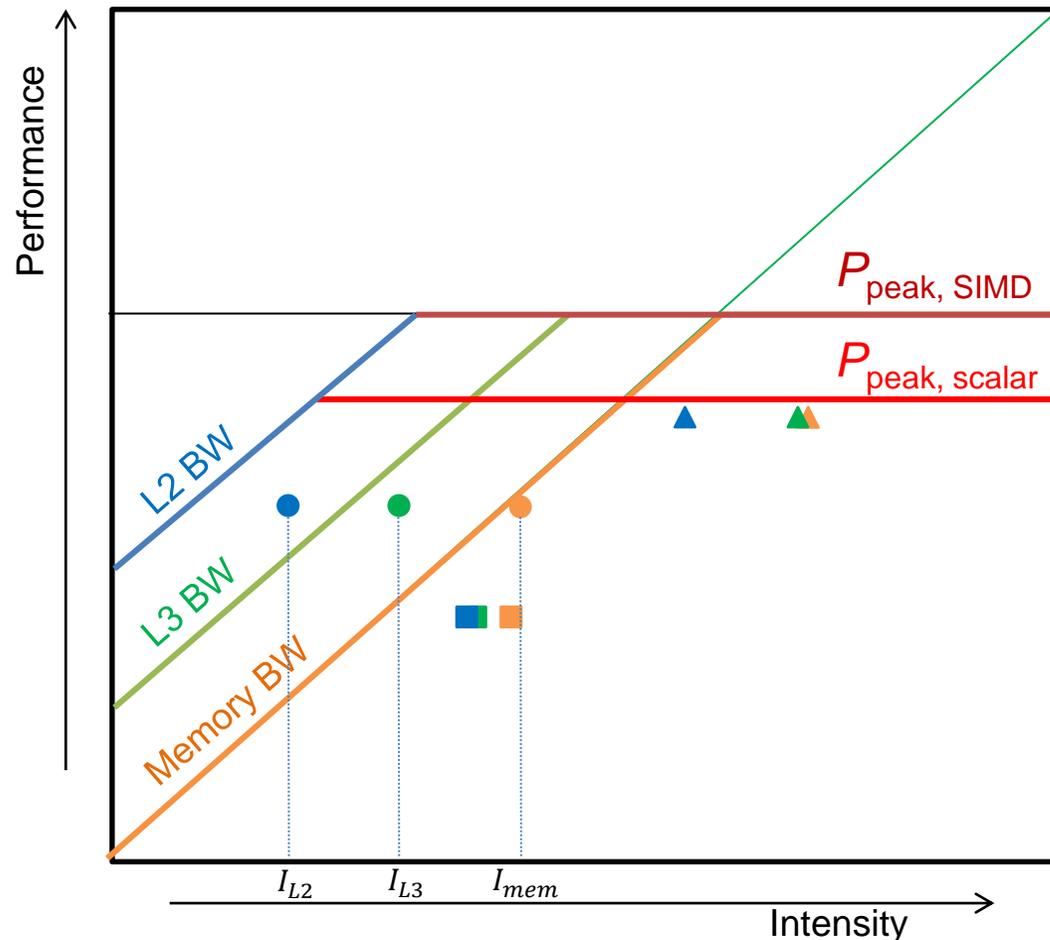
- Performance not near any BW ceiling
- Performance close to scalar peak ceiling
→ indicates **scalar core-bound peak code**

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
→ indicates **memory bound**

Kernel 2 ▲

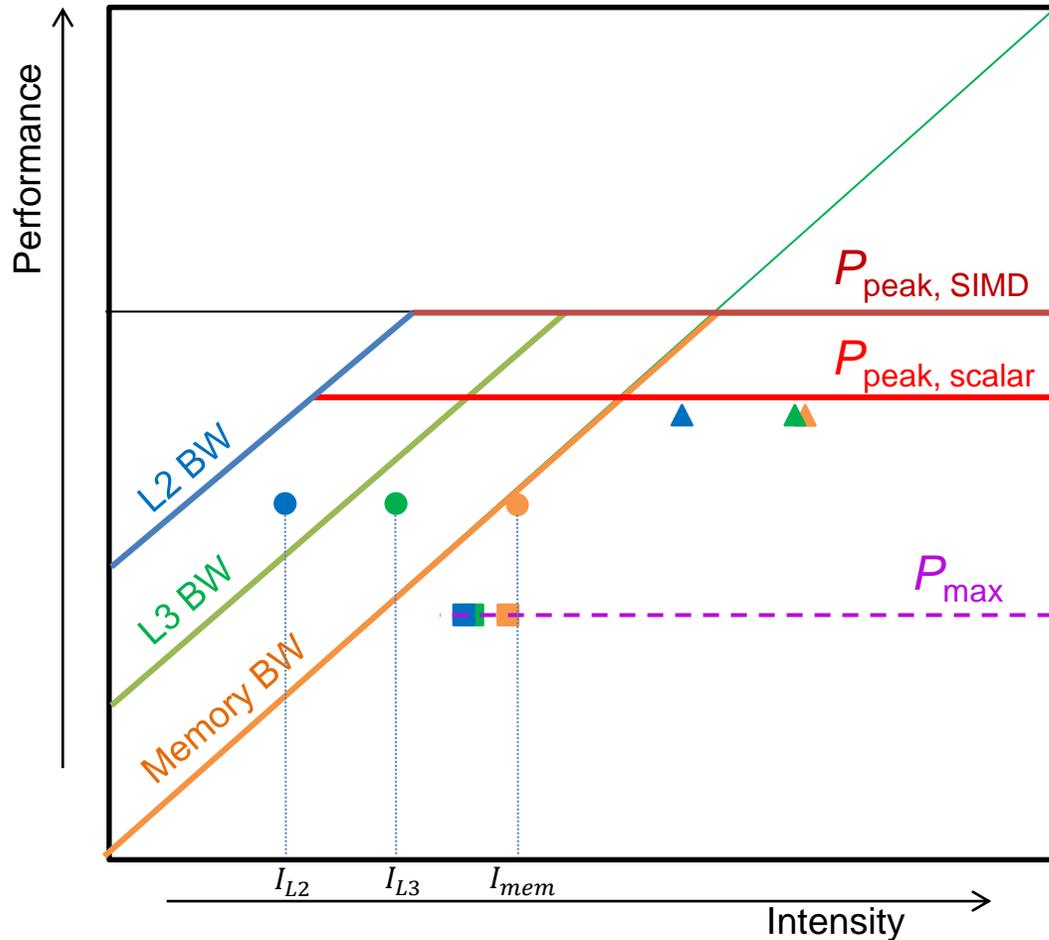
- Performance not near any BW ceiling
- Performance close to scalar peak ceiling
→ indicates **scalar core-bound peak code**

Kernel 3 ■

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks

→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \dots$)



Kernel 1 ●

- Performance close to memory BW ceiling but far away from others
→ indicates **memory bound**

Kernel 2 ▲

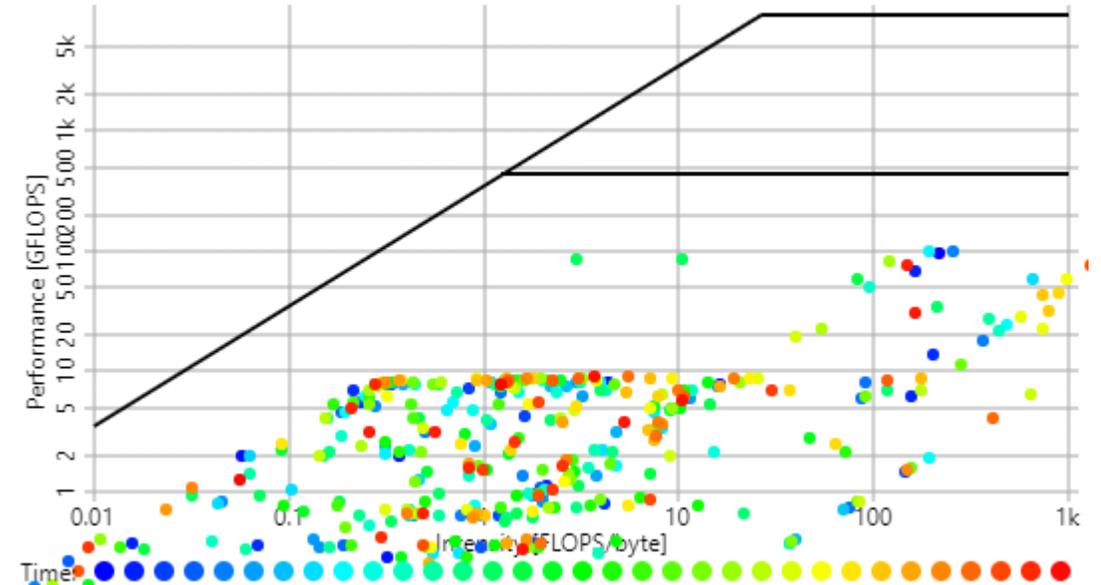
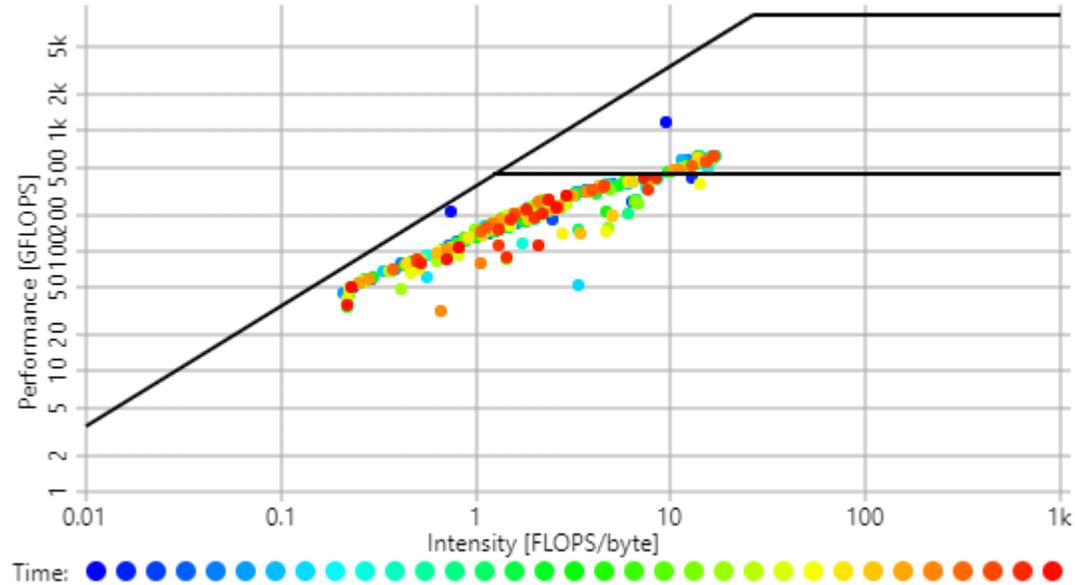
- Performance not near any BW ceiling
- Performance close to scalar peak ceiling
→ indicates **scalar core-bound peak code**

Kernel 3 ■

- Performance not anywhere near any ceiling
→ There must be an (as yet) **unknown in-core performance limit P_{max}**

Roofline and performance monitoring of clusters

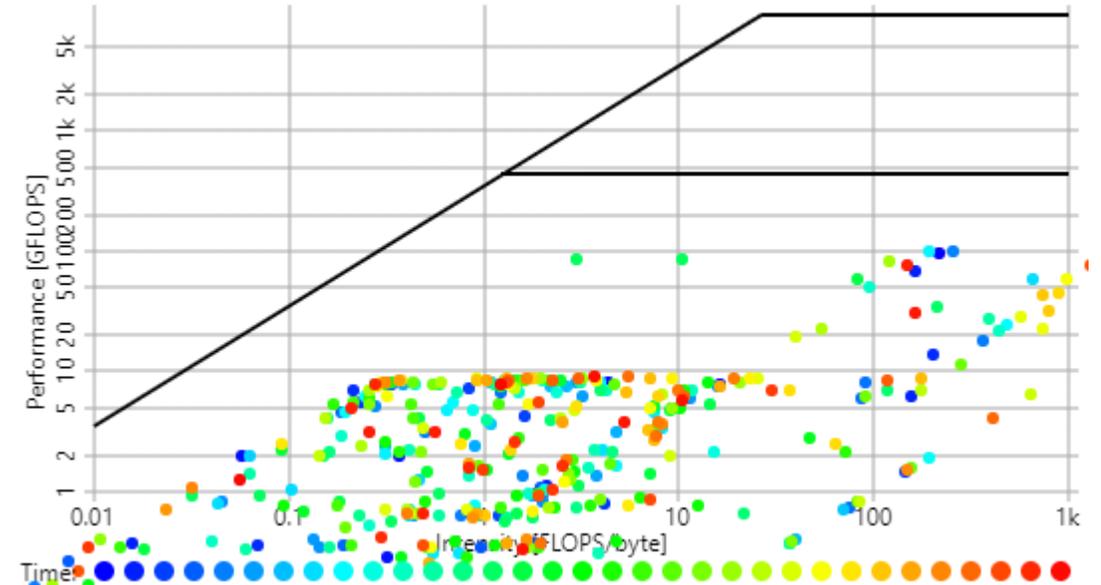
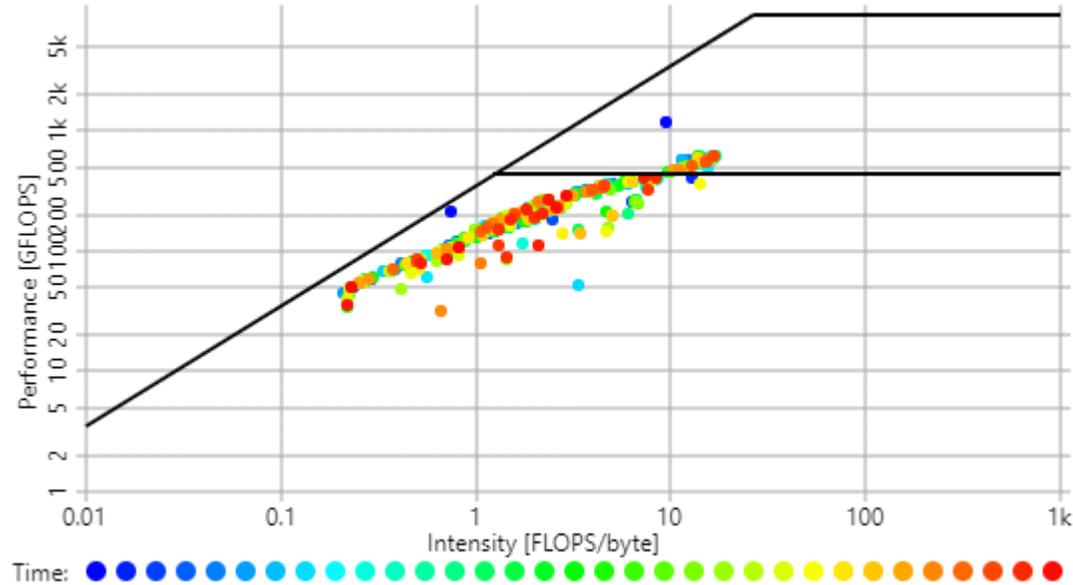
Two cluster jobs...



Cluster monitoring framework: [ClusterCockpit](https://clustercockpit.org) <https://clustercockpit.org>
Come visit Booth #1311 (LRZ)

Roofline and performance monitoring of clusters

Two cluster jobs...



Which of them is
“good” and which is
“bad”?



Cluster monitoring framework: [ClusterCockpit](https://clustercockpit.org) <https://clustercockpit.org>
Come visit Booth #1311 (LRZ)

Roofline conclusion

- Roofline = simple first-principle model for upper performance limit of data-streaming loops
 - Machine model (P_{max}, b_S) + application model (I)
 - Conditions apply, extensions exist
- Two modes of operation
 - Predictive: Calculate I , calculate upper limit, validate model, optimize, iterate
 - Diagnostic: Measure I and P , compare with roof
- Challenge of predictive modeling: Getting P_{max} and I right

Performance analysis with hardware metrics

likwid-perfctr



Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code?

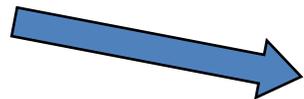
Profiling via advanced tools is often overkill

- A coarse overview is often sufficient: **likwid-perfctr**

- Simple measurement of hardware performance metrics

- Preconfigured and extensible metric groups, list with

likwid-perfctr -a:



- Operating modes:

- Wrapper
- Stethoscope
- Timeline
- Marker API

BRANCH: Branch prediction miss rate/ratio
CLOCK: Clock frequency of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
ENERGY: Power and energy consumption

Best practices for Performance profiling

Focus on **resource utilization** and **instruction mix!**

Metrics to measure:

- Operation throughput (Flops/s)
- Overall instruction throughput (CPI or IPC)
- **Instruction breakdown:**
 - FP instructions
 - loads and stores
 - branch instructions
 - other instructions
- Instruction breakdown to **SIMD width** (scalar, SSE, AVX, AVX512 for X86). (only arithmetic instructions on most architectures)
- **Data volumes** and **bandwidths** to
 - main memory (GB and GB/s)
 - cache levels (GB and GB/s)

Useful **diagnostic metrics** are:

- Clock frequency (GHz)
- Power (W)

All above metrics can be acquired using performance groups:

MEM_DP, MEM_SP, BRANCH, DATA, L2, L3

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]  
-----
```

```
<<<< PROGRAM OUTPUT >>>>
```

```
-----  
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhaltd [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]  
-----
```

```
<<<< PROGRAM OUTPUT >>>>
```

```
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

Always
measured for
Intel CPUs

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]  
-----
```

```
<<<< PROGRAM OUTPUT >>>>
```

```
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

Always measured for Intel CPUs

Configured metrics (this group)

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]  
-----
```

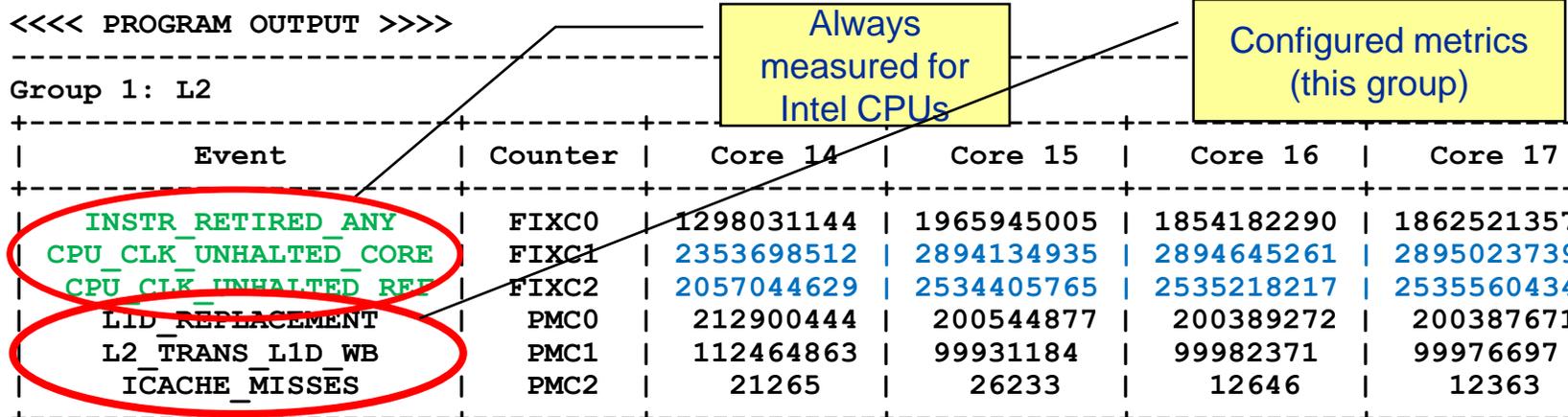
```
<<<< PROGRAM OUTPUT >>>>
```

```
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241



likwid-perfctr stethoscope mode

- **likwid-perfctr** counts events on hardware threads
it has no notion of what kind of code is running (if any)

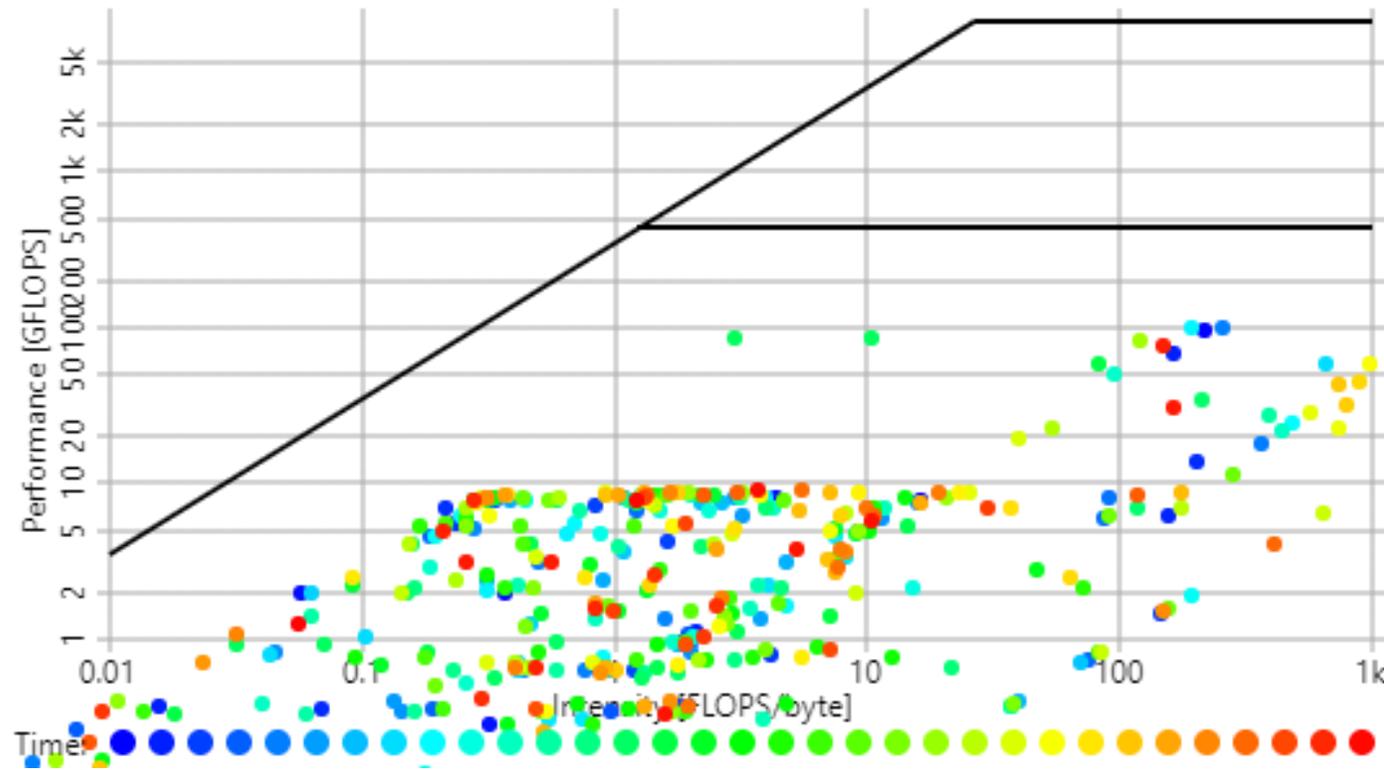
This allows you to “listen” to what is currently happening,
without any overhead:

```
$ likwid-perfctr -c N:0-11 -g FLOPS_DP -S 10s
```

- It can be used as cluster/server monitoring tool
- A frequent use is to measure a certain part of a long running parallel application from outside

likwid-perfctr stethoscope example

Using Roofline for monitoring “live” jobs on a cluster
Based on measured BW and Flop/s data via **likwid-perfctr**



Cluster monitoring framework: [ClusterCockpit](https://clustercockpit.org) <https://clustercockpit.org>
Come visit Booth #1311 (LRZ)

likwid-perfctr with MarkerAPI

- The MarkerAPI can restrict measurements to **code regions**
- The API only reads counters.
The configuration of the counters is still done by **likwid-perfctr**
- Multiple named regions allowed, accumulation over multiple calls
- Inclusive and overlapping regions allowed

- **Caveat:** Marker API can cause overhead; do not call too frequently!

- Fortran API exists

```
#include <likwid-marker.h>

LIKWID_MARKER_INIT; // must be called from serial region
. . .
LIKWID_MARKER_START("Compute");
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .
LIKWID_MARKER_CLOSE; // must be called from serial region
```

likwid-perfctr with MarkerAPI: source code transformations

```
#pragma omp parallel for  
  <loop>
```



```
#pragma omp parallel  
{  
  LIKWID_MARKER_START("Compute");  
  #pragma omp for  
  <loop>  
  LIKWID_MARKER_STOP("Compute");  
}
```

```
some_parallel_f()
```



```
#pragma omp parallel  
{  
  LIKWID_MARKER_START("foo");  
}  
some_parallel_f()  
#pragma omp parallel  
{  
  LIKWID_MARKER_STOP("foo");  
}
```

Compiling, linking, and running with marker API

Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

Link:

```
cc -L /path/to/liblikwid program.o -o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

MPI/hybrid:

```
likwid-mpirun -np 4 -pin <PINEXPR> -g <GROUP> -m ./program
```

→ One separate block of output for every marked region

Compiling, linking, and running with marker API

Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

Activate LIKWID
macros (C only)

Link:

```
cc -L /path/to/liblikwid program.o -o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

MPI/hybrid:

```
likwid-mpirun -np 4 -pin <PINEXPR> -g <GROUP> -m ./program
```

→ One separate block of output for every marked region

Compiling, linking, and running with marker API

Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

Activate LIKWID
macros (C only)

Link:

```
cc -L /path/to/liblikwid program.o -o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

Activate
markers

MPI/hybrid:

```
likwid-mpirun -np 4 -pin <PINEXPR> -g <GROUP> -m ./program
```

→ One separate block of output for every marked region

Summary of hardware performance monitoring

- Useful **only if you know what you are looking for**
 - PM bears potential of acquiring massive amounts of data for nothing!
- **Resource-based metrics** are most useful
 - Cache lines transferred, work executed, loads/stores, cycles
 - Instructions, CPI, cache misses may be misleading
- Caveat: **Processor work != user work**
 - Waiting time in libraries (OpenMP, MPI) may cause lots of instructions
 - → distorted application characteristic
- Another very useful application of PM: **validating performance models!**
 - Roofline is data centric → measure data volume through memory hierarchy

Case study:

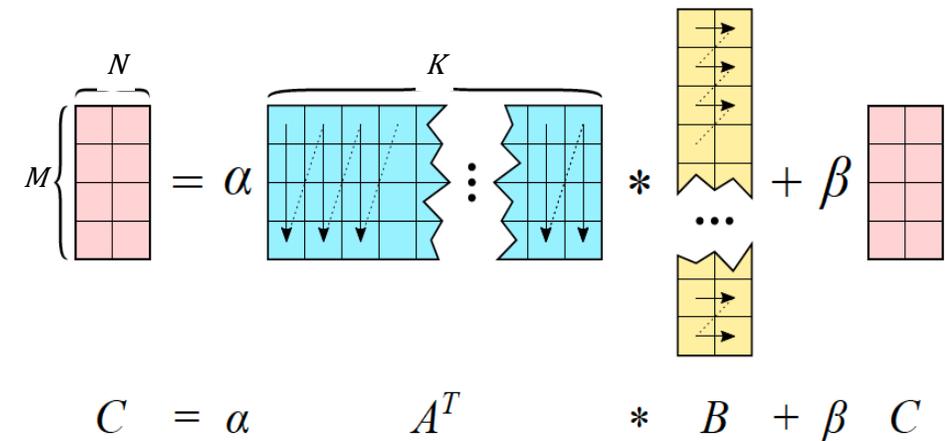
Tall & Skinny Matrix-Transpose Times
Tall & Skinny Matrix (TSM TTS M)
Multiplication



TSMTTSM Multiplication

- **Block of vectors** → Tall & Skinny Matrix (e.g. $10^7 \times 10^1$ dense matrix)
- Row-major storage format (see SpMVM)
- Block vector subspace orthogonalization procedure requires, e.g., computation of scalar product between vectors of two blocks

- → **TSMTTSM** Multiplication



The diagram illustrates the TSMTTSM multiplication equation. On the left, a pink matrix C of size $M \times N$ is shown. This is equal to a scalar α multiplied by a blue matrix A^T of size $M \times K$. The matrix A^T is composed of several blocks, each containing a vertical vector of arrows pointing downwards. To the right of A^T is a vertical ellipsis, followed by a yellow matrix B of size $K \times N$. The matrix B is composed of several blocks, each containing a horizontal vector of arrows pointing to the right. This is followed by a plus sign and a scalar β multiplied by the pink matrix C . Below the diagram, the equation is written as $C = \alpha A^T * B + \beta C$.

$$C = \alpha A^T * B + \beta C$$

Assume: $\alpha = 1$; $\beta = 0$

TSMTTSM Multiplication

General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, (e.g., Intel MKL¹):

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn} , \quad m = 1..M, n = 1..N$$

Matrix sizes:

Square (SQ): M=N=K=15,000

Tall&Skinny (TS): M=N=16 ; K=10,000,000

¹Intel Math Kernel Library (MKL) 11.3

TSMTTSM Multiplication

General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, (e.g., Intel MKL¹):

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn} , \quad m = 1..M, n = 1..N$$

System	P _{peak} [GF/s]	b _s [GB/s]	Size	Perf.	Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s	52 GB/s	SQ	160 GF/s	91%
			TS	16.6 GF/s	6%
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s	65 GB/s	SQ	550 GF/s	95%
			TS	22.8 GF/s	4%

Matrix sizes:

Square (SQ): M=N=K=15,000

Tall&Skinny (TS): M=N=16 ; K=10,000,000

¹Intel Math Kernel Library (MKL) 11.3

TSMTTSM Multiplication

General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, (e.g., Intel MKL¹):

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn}, \quad m = 1..M, n = 1..N$$

double

System	P _{peak} [GF/s]	b _s [GB/s]	Size	Perf.	Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s	52 GB/s	SQ	160 GF/s	91%
			TS	16.6 GF/s	6%
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s	65 GB/s	SQ	550 GF/s	95%
			TS	22.8 GF/s	4%

complex double

Matrix sizes:

Square (SQ): M=N=K=15,000

Tall&Skinny (TS): M=N=16 ; K=10,000,000

¹Intel Math Kernel Library (MKL) 11.3

TSMTTSM Multiplication

General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, (e.g., Intel MKL¹):

$$C_{mn} = \sum_{k=1}^K A_{mk} B_{kn}, \quad m = 1..M, n = 1..N$$

System	P _{peak} [GF/s]	b _s [GB/s]	Size	Perf.	Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s	52 GB/s	SQ	160 GF/s	91%
			TS	16.6 GF/s	6%
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s	65 GB/s	SQ	550 GF/s	95%
			TS	22.8 GF/s	4%

double

complex double

TS@MKL:
Good or bad?

Matrix sizes:

Square (SQ): M=N=K=15,000

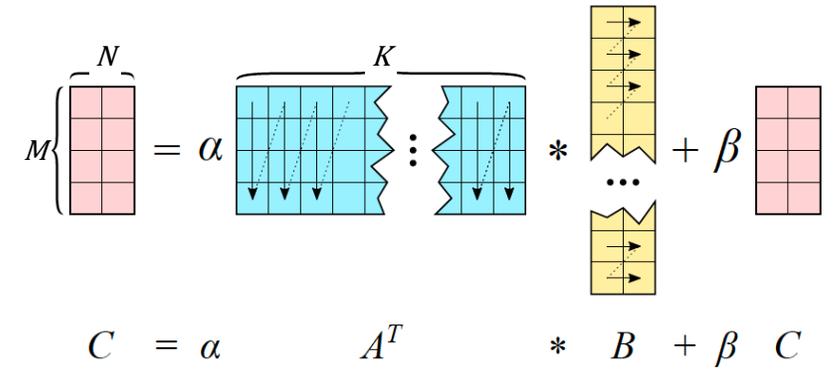
Tall&Skinny (TS): M=N=16 ; K=10,000,000

¹Intel Math Kernel Library (MKL) 11.3

TSMTTSM Roofline model

Computational intensity

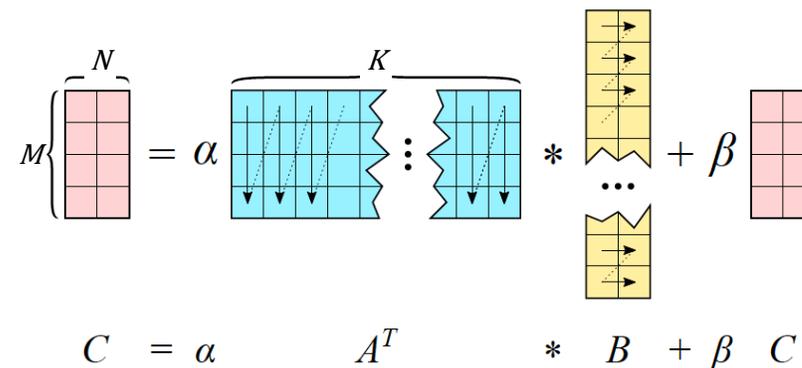
$$I = \frac{\text{\#flops}}{\text{\#bytes (slowest data path)}}$$



TSMTTSM Roofline model

Computational intensity

$$I = \frac{\text{\#flops}}{\text{\#bytes (slowest data path)}}$$



Optimistic model (minimum data transfer) assuming $M = N \ll K$ and double precision:

$$I_d \approx \frac{2KMN}{8(KM + KN)} \frac{F}{B} = \frac{MF}{8B}$$

complex double:

$$I_z \approx \frac{8KMN}{16(KM + KN)} \frac{F}{B} = \frac{MF}{4B}$$

TSMITTSM Roofline performance prediction

Now choose $M = N = 16 \rightarrow I_d \approx \frac{16 F}{8 B}$ and $I_z \approx \frac{16 F}{4 B}$, i.e. $B_d \approx 0.5 \frac{B}{F}$, $B_z \approx 0.25 \frac{B}{F}$

Intel Xeon E5 2660 v2 ($b_s = 52 \frac{GB}{s}$) $\rightarrow P = 104 \frac{GF}{s}$ (double)

Measured (MKL): $16.6 \frac{GF}{s}$

Intel Xeon E5 2697 v3 ($b_s = 65 \frac{GB}{s}$) $\rightarrow P = 240 \frac{GF}{s}$ (double complex)

Measured (MKL): $22.8 \frac{GF}{s}$

TSMTTSM Roofline performance prediction

Now choose $M = N = 16 \rightarrow I_d \approx \frac{16 F}{8 B}$ and $I_z \approx \frac{16 F}{4 B}$, i.e. $B_d \approx 0.5 \frac{B}{F}$, $B_z \approx 0.25 \frac{B}{F}$

Intel Xeon E5 2660 v2 ($b_s = 52 \frac{GB}{s}$) $\rightarrow P = 104 \frac{GF}{s}$ (double)

Measured (MKL): $16.6 \frac{GF}{s}$

Intel Xeon E5 2697 v3 ($b_s = 65 \frac{GB}{s}$) $\rightarrow P = 240 \frac{GF}{s}$ (double complex)

Measured (MKL): $22.8 \frac{GF}{s}$

\rightarrow Potential speedup: 6–10x vs. MKL

Can we implement a better TSMTTSM kernel than Intel?

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5   #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8       #pragma simd
9         for (int acol = 0; acol < m; acol++) {
10            c_tmp[bcol*m+acol] +=
11              a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12              a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13          }
14        }
15      }
16
17   #pragma omp critical
18   for (int bcol = 0; bcol < n; bcol++) {
19     #pragma simd
20     for (int acol = 0; acol < m; acol++) {
21       c[bcol*m+acol] += c_tmp[bcol*m+acol];
22     }
23   }
24 }
```

Can we implement a better TSMTTSM kernel than Intel?

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5   #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8       #pragma simd
9         for (int acol = 0; acol < m; acol++) {
10          c_tmp[bcol*m+acol] +=
11            a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12            a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13        }
14      }
15    }
16
17   #pragma omp critical
18   for (int bcol = 0; bcol < n; bcol++) {
19     #pragma simd
20     for (int acol = 0; acol < m; acol++) {
21       c[bcol*m+acol] += c_tmp[bcol*m+acol];
22     }
23   }
24 }
```

Long Loop (k): Parallel

Can we implement a better TSMTTSM kernel than Intel?

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5   #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8       #pragma simd
9         for (int acol = 0; acol < m; acol++) {
10          c_tmp[bcol*m+acol] +=
11            a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12            a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13        }
14      }
15    }
16
17   #pragma omp critical
18   for (int bcol = 0; bcol < n; bcol++) {
19     #pragma simd
20     for (int acol = 0; acol < m; acol++) {
21       c[bcol*m+acol] += c_tmp[bcol*m+acol];
22     }
23   }
24 }
```

Thread-local copy of small (results) matrix

Long Loop (k): Parallel

Can we implement a better TSMTTSM kernel than Intel?

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5   #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8 #pragma simd
9       for (int acol = 0; acol < m; acol++) {
10         c_tmp[bcol*m+acol] +=
11           a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12           a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13       }
14     }
15 }
16
17 #pragma omp critical
18 for (int bcol = 0; bcol < n; bcol++) {
19 #pragma simd
20   for (int acol = 0; acol < m; acol++) {
21     c[bcol*m+acol] += c_tmp[bcol*m+acol];
22   }
23 }
24 }
```

Thread-local copy of small (results) matrix

Long Loop (k): Parallel

Reduction on
small result matrix

Can we implement a better TSMTTSM kernel than Intel?

```
1 #pragma omp parallel
2 {
3   double c_tmp[n*m] = {0.};
4
5   #pragma omp for
6   for (int row = 0; row < k-1; row+=2) {
7     for (int bcol = 0; bcol < n; bcol++) {
8 #pragma simd
9       for (int acol = 0; acol < m; acol++) {
10        c_tmp[bcol*m+acol] +=
11          a[(row+0)*m + acol] * b[(row+0)*n + bcol] +
12          a[(row+1)*m + acol] * b[(row+1)*n + bcol];
13      }
14    }
15  }
16
17 #pragma omp critical
18 for (int bcol = 0; bcol < n; bcol++) {
19 #pragma simd
20   for (int acol = 0; acol < m; acol++) {
21     c[bcol*m+acol] += c_tmp[bcol*m+acol];
22   }
23 }
24 }
```

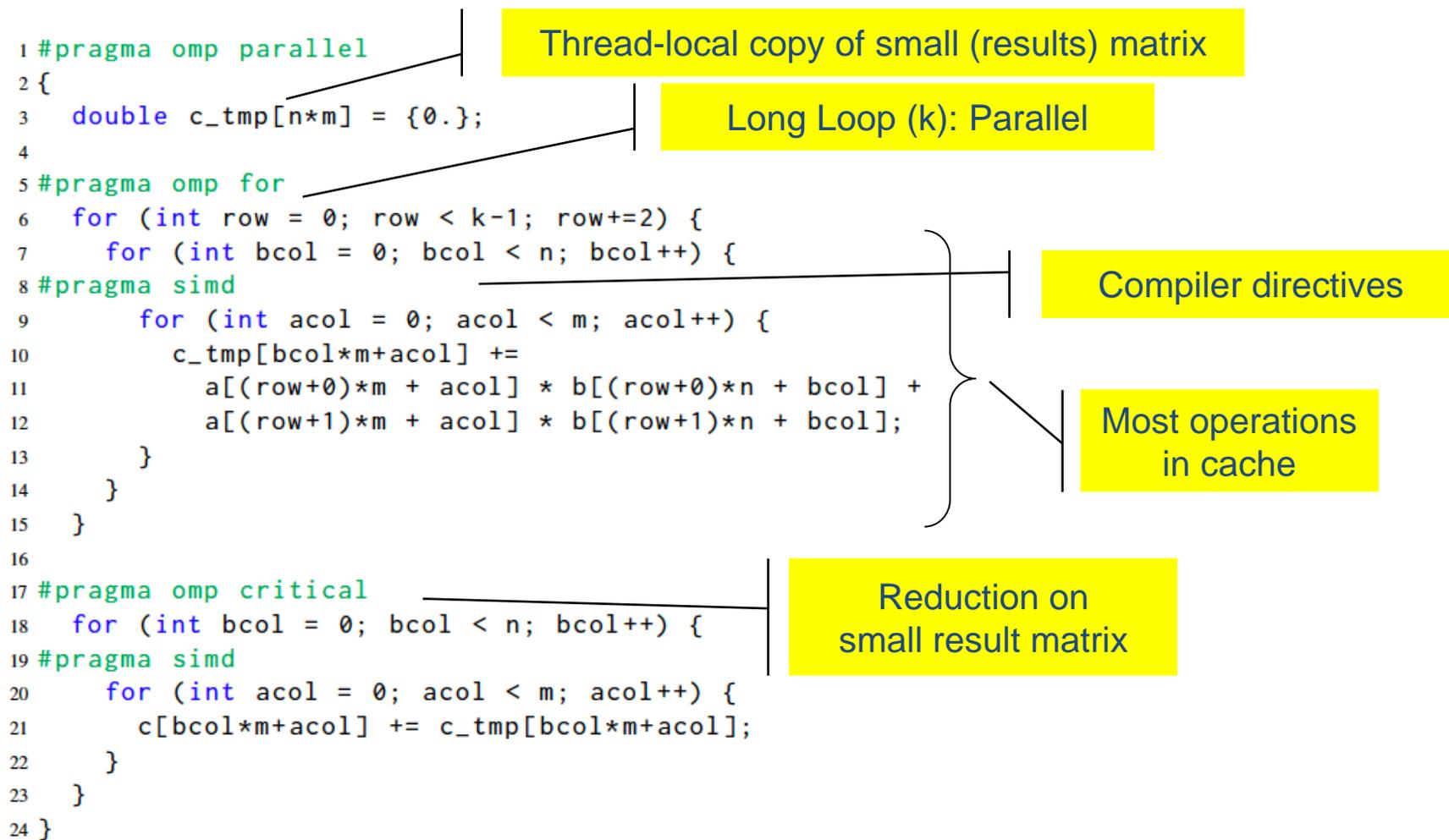
Thread-local copy of small (results) matrix

Long Loop (k): Parallel

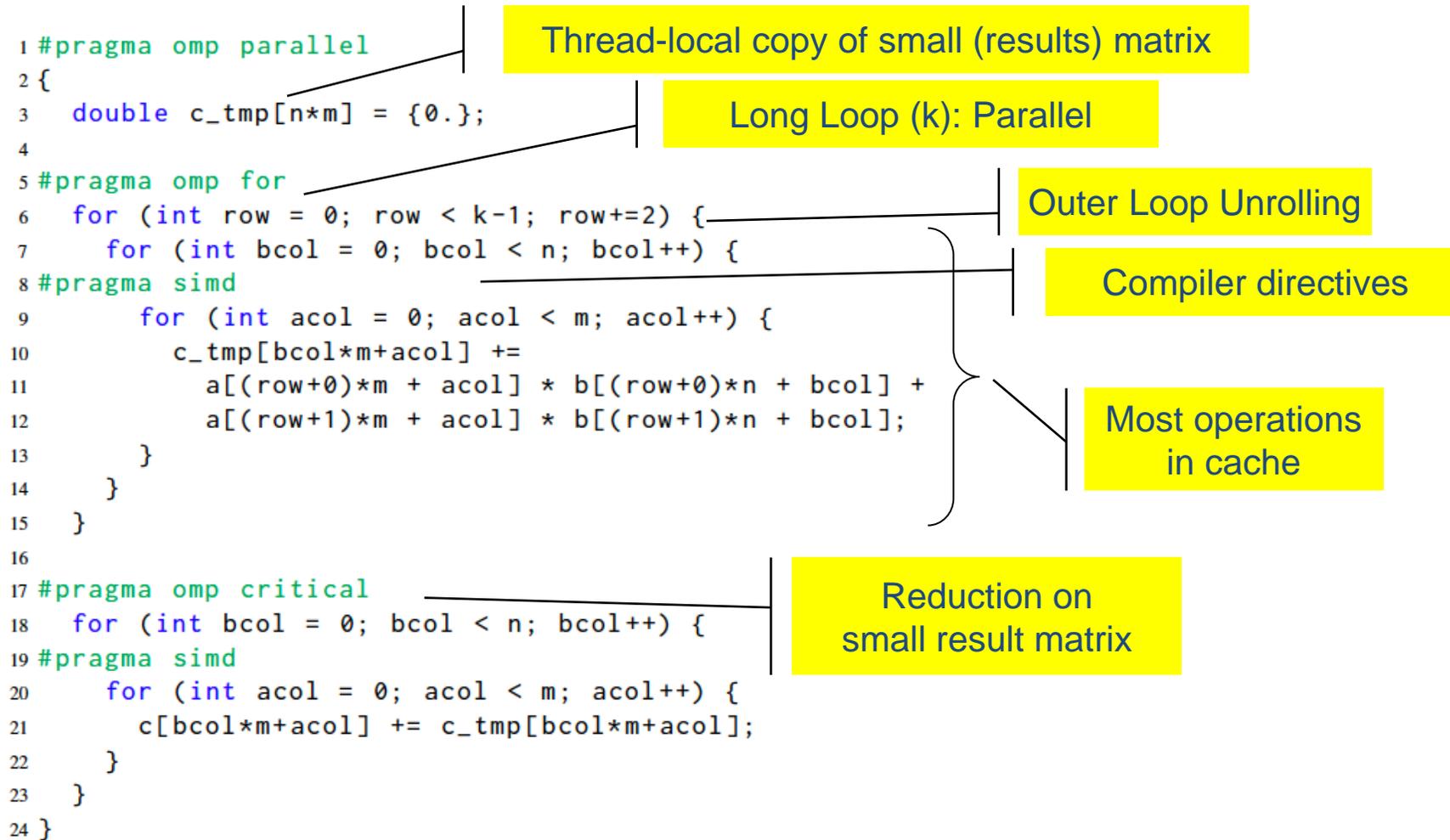
Most operations
in cache

Reduction on
small result matrix

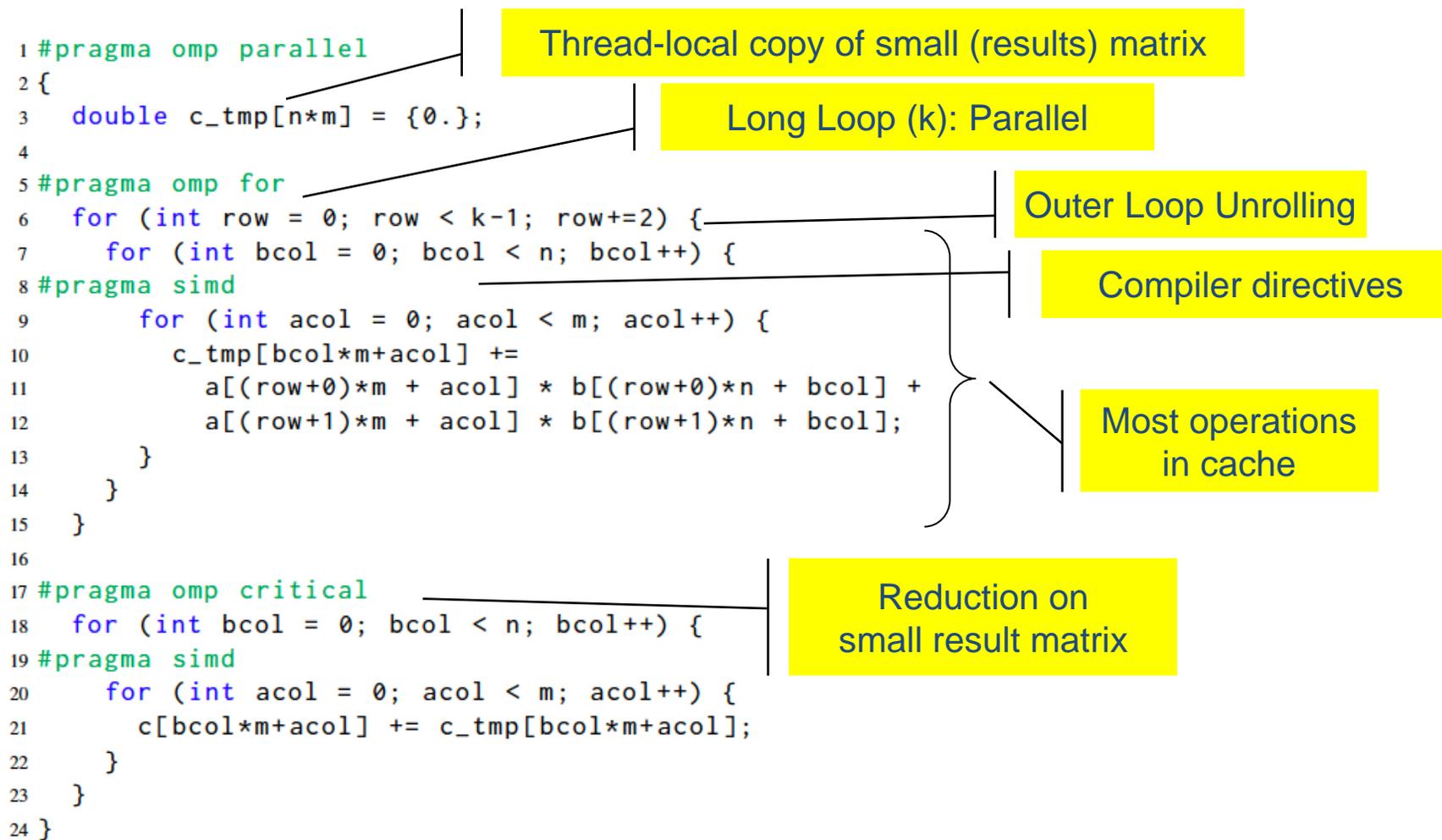
Can we implement a better TSMTTSM kernel than Intel?



Can we implement a better TSMTTSM kernel than Intel?



Can we implement a better TSMTTSM kernel than Intel?



Not shown: Inner Loop boundaries (n,m) known at compile time (kernel generation), k assumed to be even

TSMITTSM MKL vs. “hand crafted” (OPT)

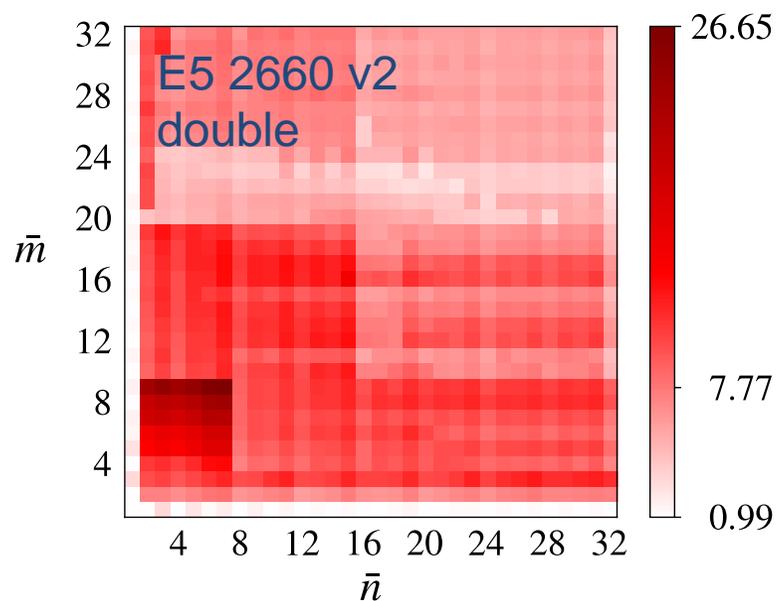
TS: M=N=16 ; K=10,000,000

System	P_{peak} / b_s	Version	Performance	RLM Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s 52 GB/s	TS OPT	98 GF/s	94 %
		TS MKL	16.6 GF/s	16 %
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s 65 GB/s	TS OPT	159 GF/s	66 %
		TS MKL	22.8 GF/s	9.5 %

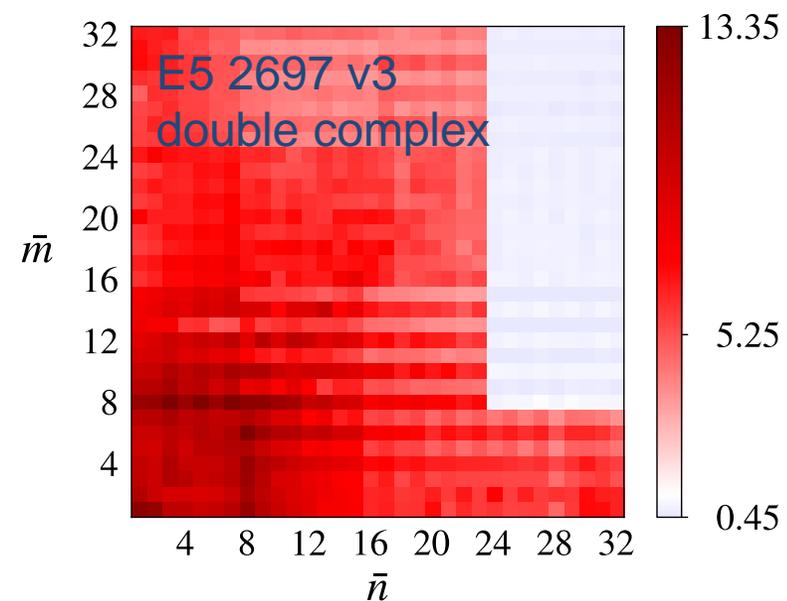
TSMITTSM MKL vs. “hand crafted” (OPT)

TS: M=N=16 ; K=10,000,000

System	P_{peak} / b_s	Version	Performance	RLM Efficiency
Intel Xeon E5 2660 v2 10c@2.2 GHz	176 GF/s 52 GB/s	TS OPT	98 GF/s	94 %
		TS MKL	16.6 GF/s	16 %
Intel Xeon E5 2697 v3 14c@2.6GHz	582 GF/s 65 GB/s	TS OPT	159 GF/s	66 %
		TS MKL	22.8 GF/s	9.5 %



Speedup
vs. MKL:
5x – 25x



TSMTTSM conclusion

- Typical example of **model-guided optimization**
- “Invisible” P_{\max} ceiling with Intel MKL
- Hand-coded implementation ran much closer to limit

- **Caveat:** this is to exemplify the method; current MKL versions might have improved!

Case study: A Jacobi smoother

The basics in two dimensions

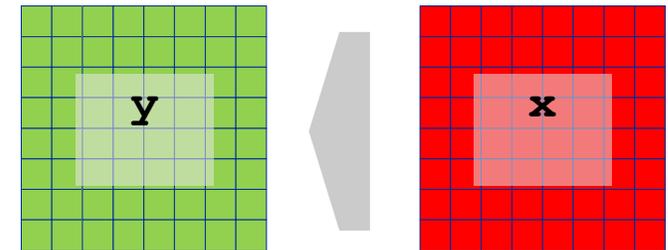


Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- ... but the **regular access structure** allows for **matrix-free coding**

```
do iter = 1, max_iterations  
    Perform sweep over regular grid:  $y(:) \leftarrow x(:)$   
    Swap  $y \leftrightarrow x$   
enddo
```

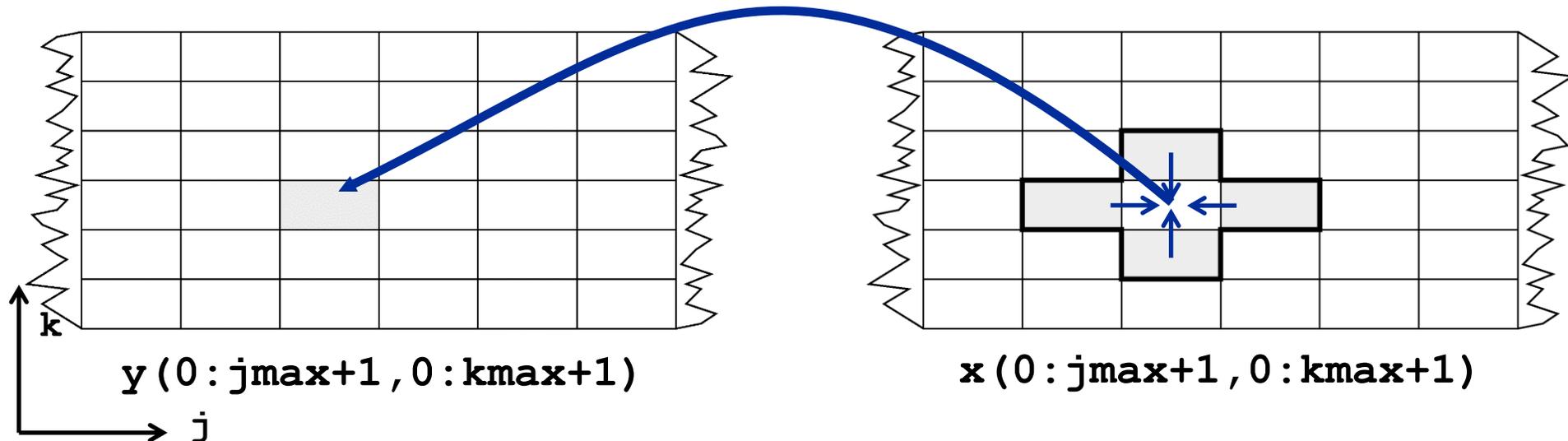
- Complexity of implementation and performance depends on
 - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type, ...
 - discretization, e.g. 7-pt or 27-pt in 3D,...



Jacobi-type 5-pt stencil sweep in 2D

sweep

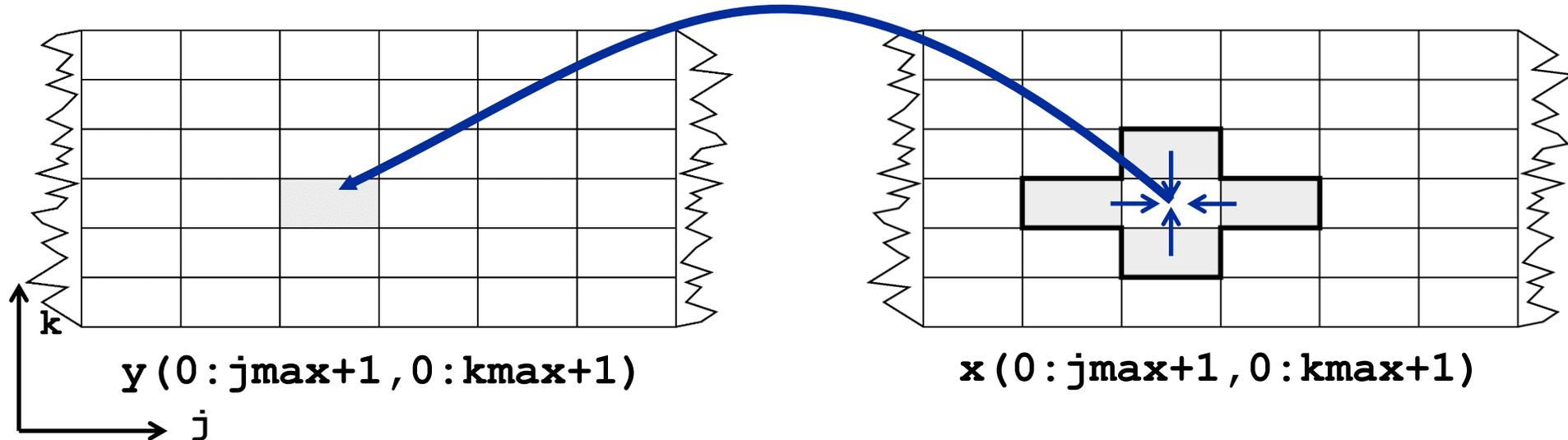
```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



Jacobi-type 5-pt stencil sweep in 2D

sweep

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

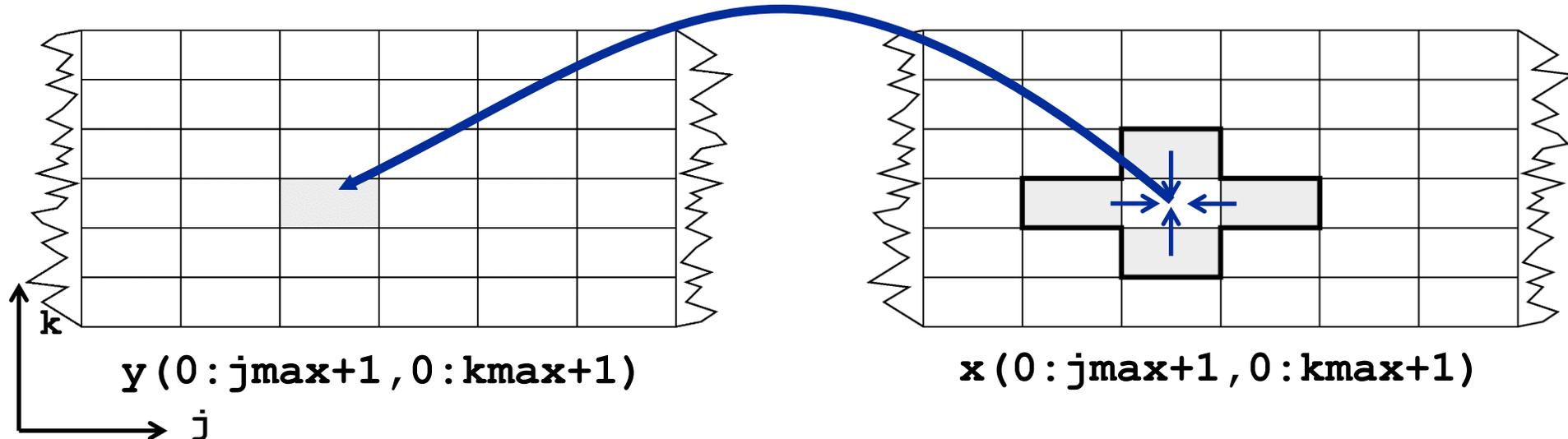


Jacobi-type 5-pt stencil sweep in 2D

sweep

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

Lattice site
update
(LUP)

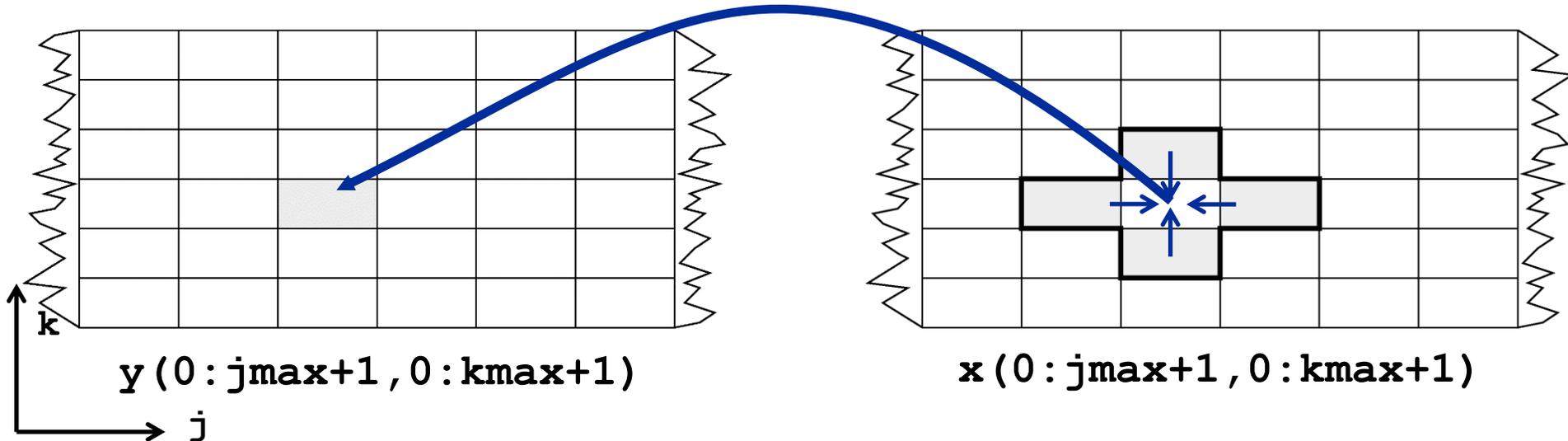


Jacobi-type 5-pt stencil sweep in 2D

```
do k=1, kmax
  do j=1, jmax
    y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

sweep

Lattice site update (LUP)



Appropriate performance metric: “Lattice site updates per second” [LUP/s]
(here: Multiply by 4 FLOP/LUP to get FLOP/s rate)

Jacobi 5-pt stencil 2D: data transfer analysis

sweep

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

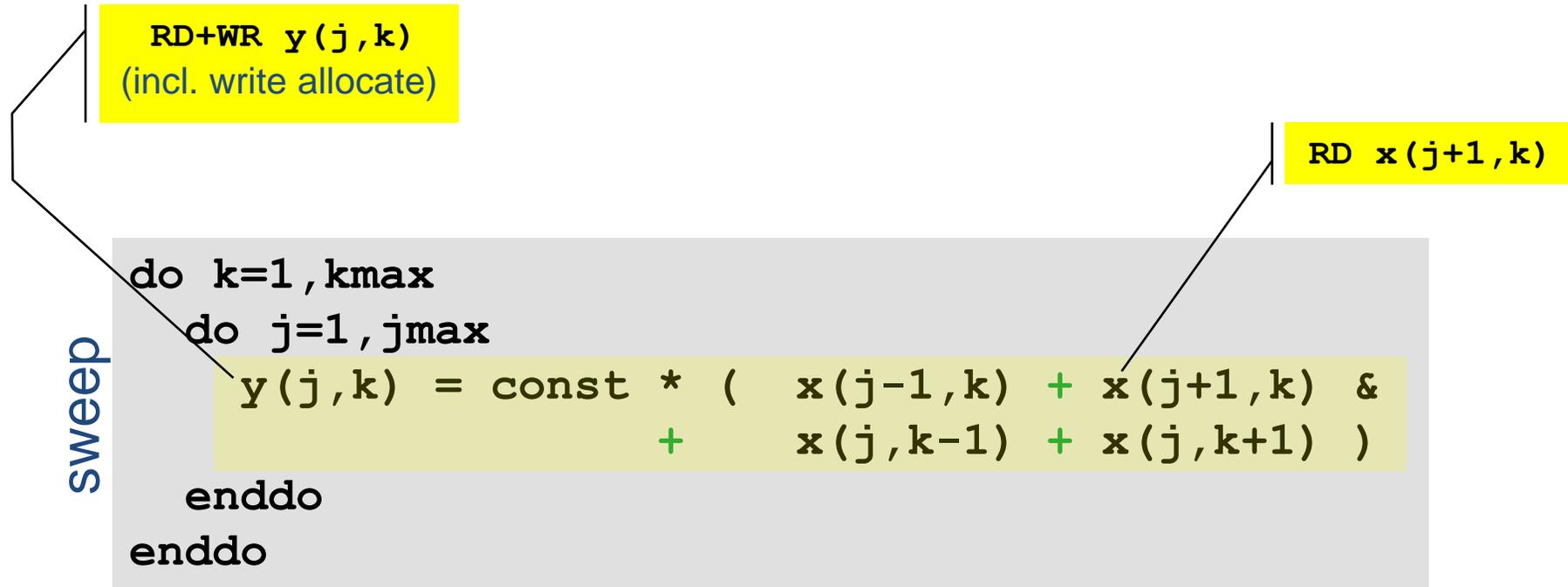
Jacobi 5-pt stencil 2D: data transfer analysis

RD+WR $y(j,k)$
(incl. write allocate)

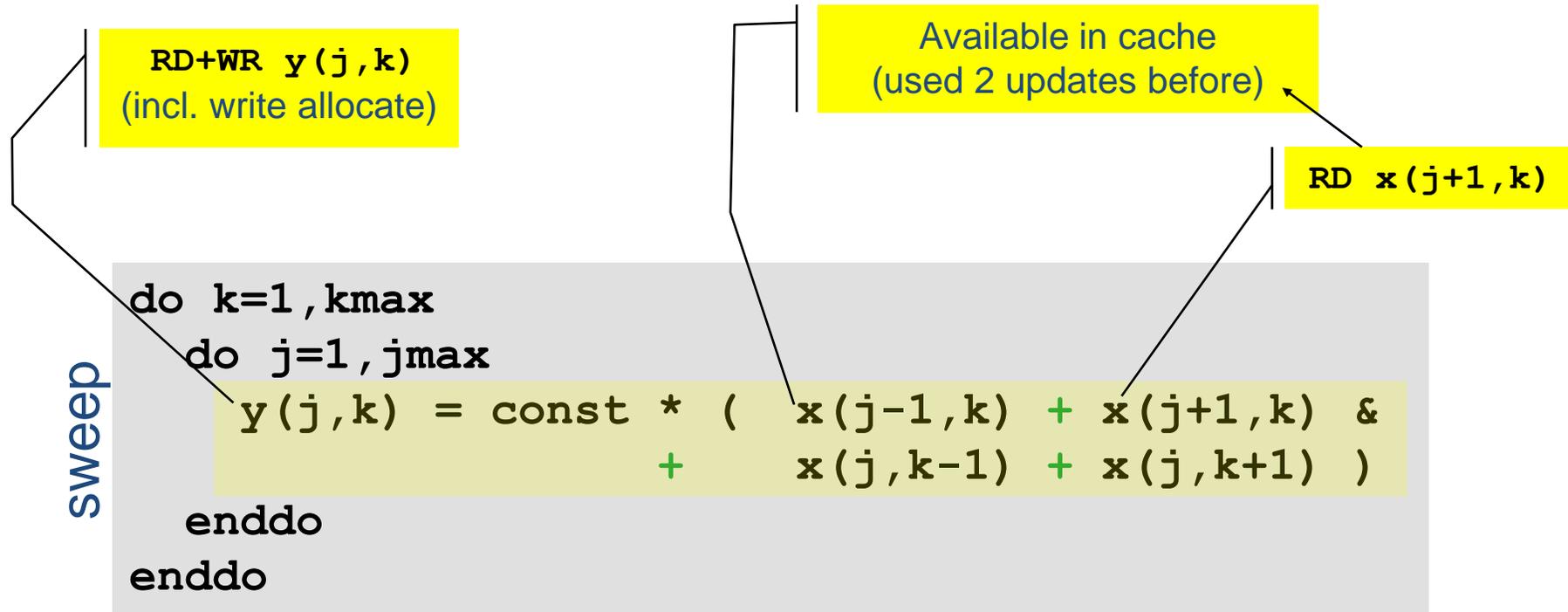
sweep

```
do k=1,kmax
do j=1,jmax
   $y(j,k) = \text{const} * ( x(j-1,k) + x(j+1,k) \&$ 
     $+ x(j,k-1) + x(j,k+1) )$ 
enddo
enddo
```

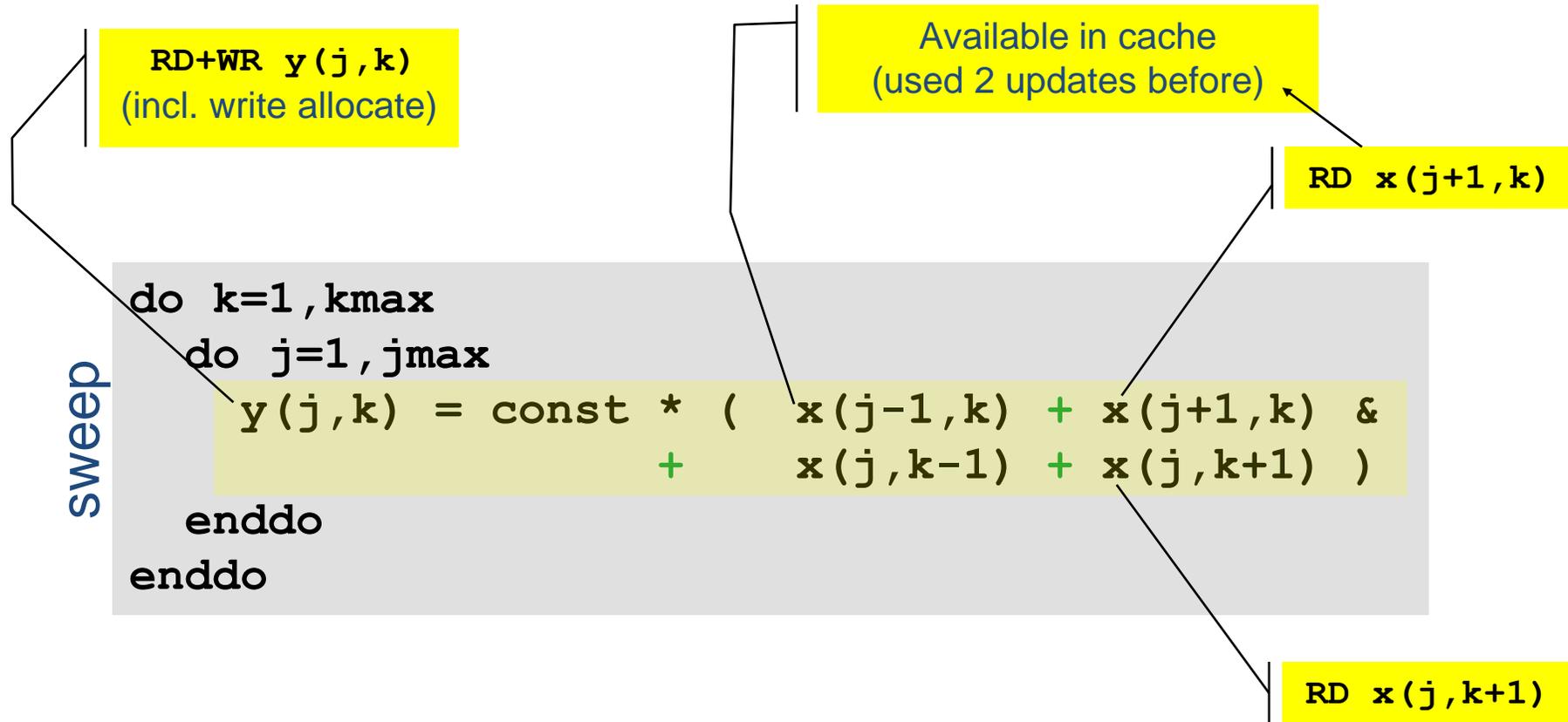
Jacobi 5-pt stencil 2D: data transfer analysis



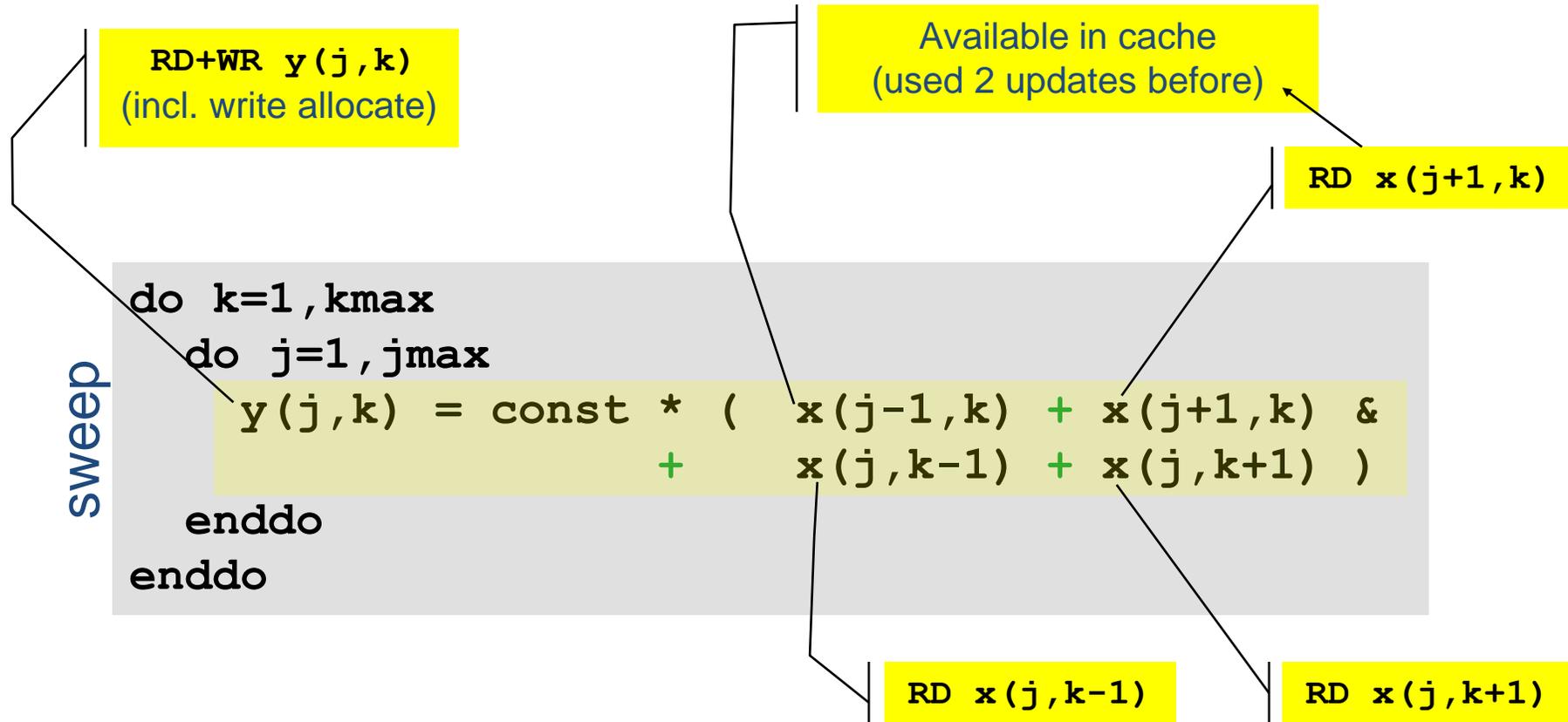
Jacobi 5-pt stencil 2D: data transfer analysis



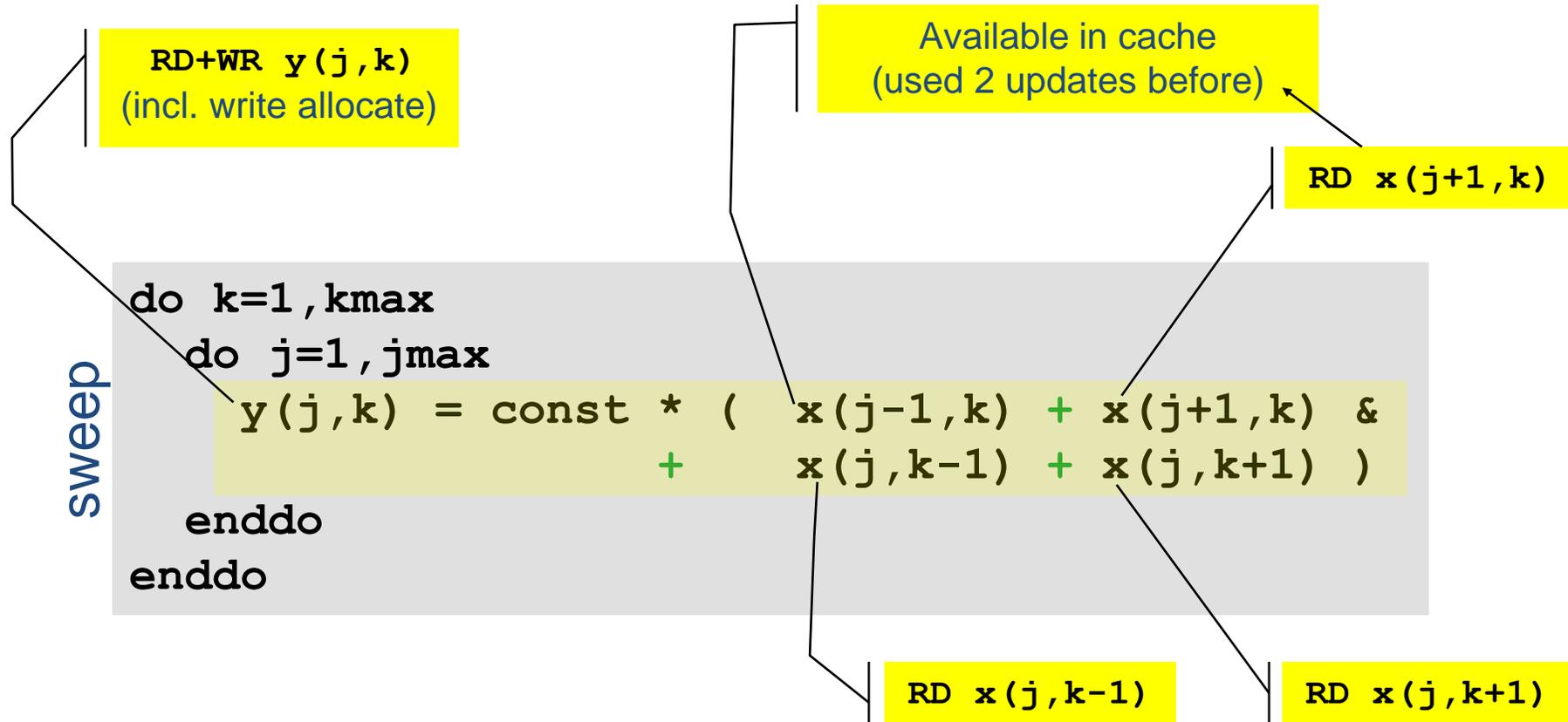
Jacobi 5-pt stencil 2D: data transfer analysis



Jacobi 5-pt stencil 2D: data transfer analysis



Jacobi 5-pt stencil 2D: data transfer analysis

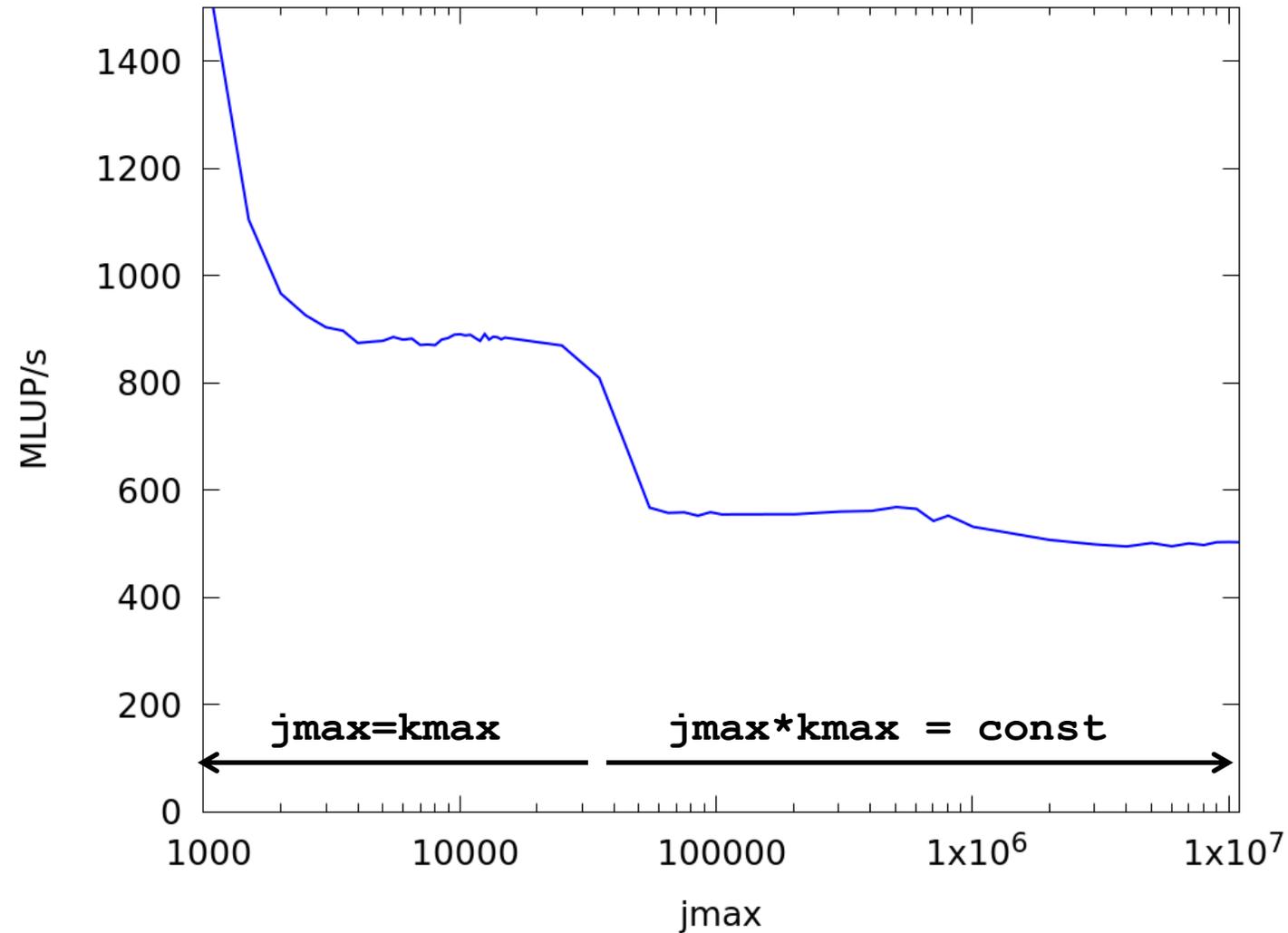


Naive balance (incl. write allocate):

$\mathbf{x}(\ :, \ :)$: 3 RD +
 $\mathbf{y}(\ :, \ :)$: 1 WR+ 1 RD

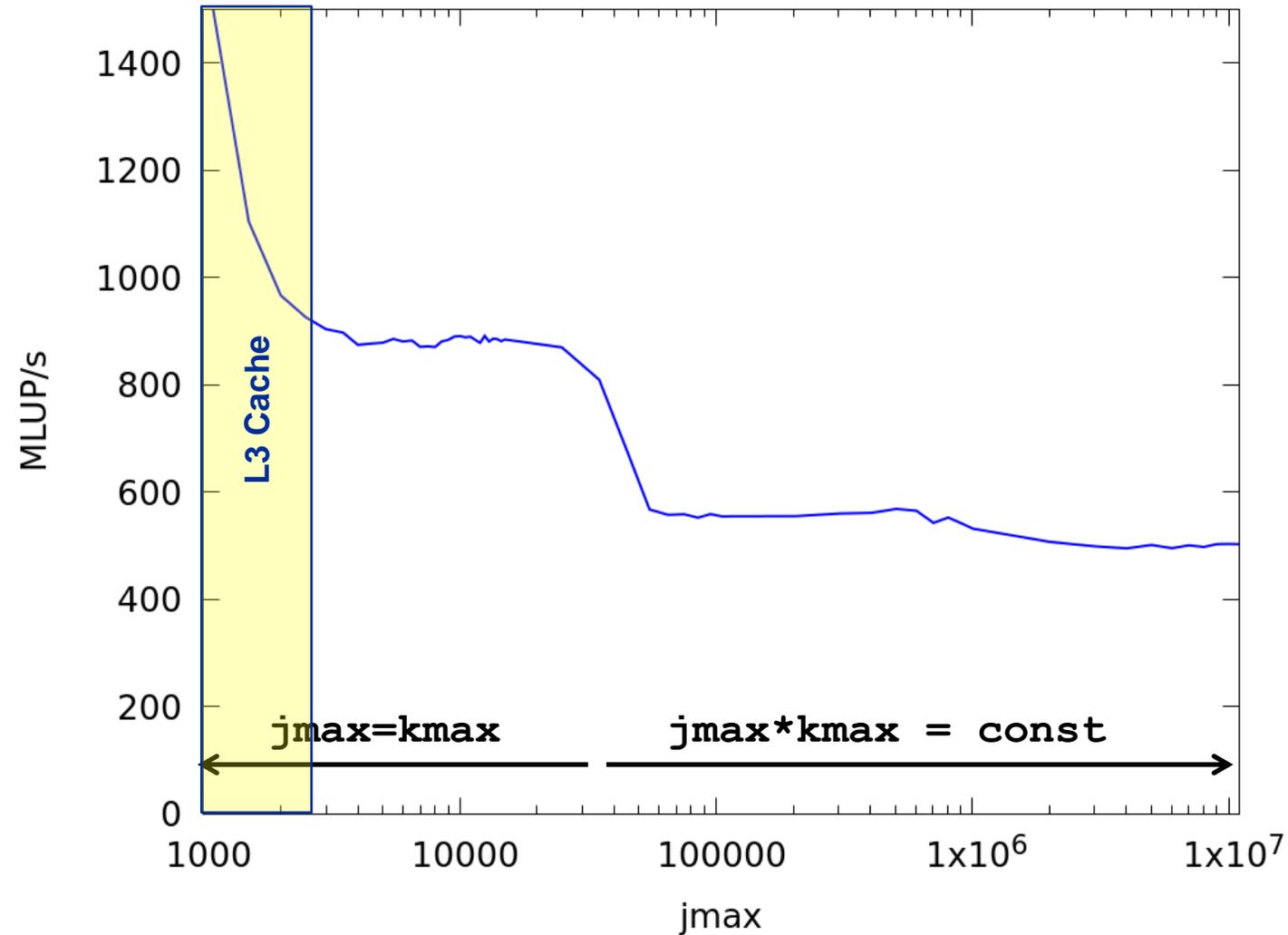
→ $B_C = 5 \text{ Words} / \text{LUP} = 40 \text{ B} / \text{LUP}$ (assuming double precision)

Jacobi 5-pt stencil 2D: Single-core performance



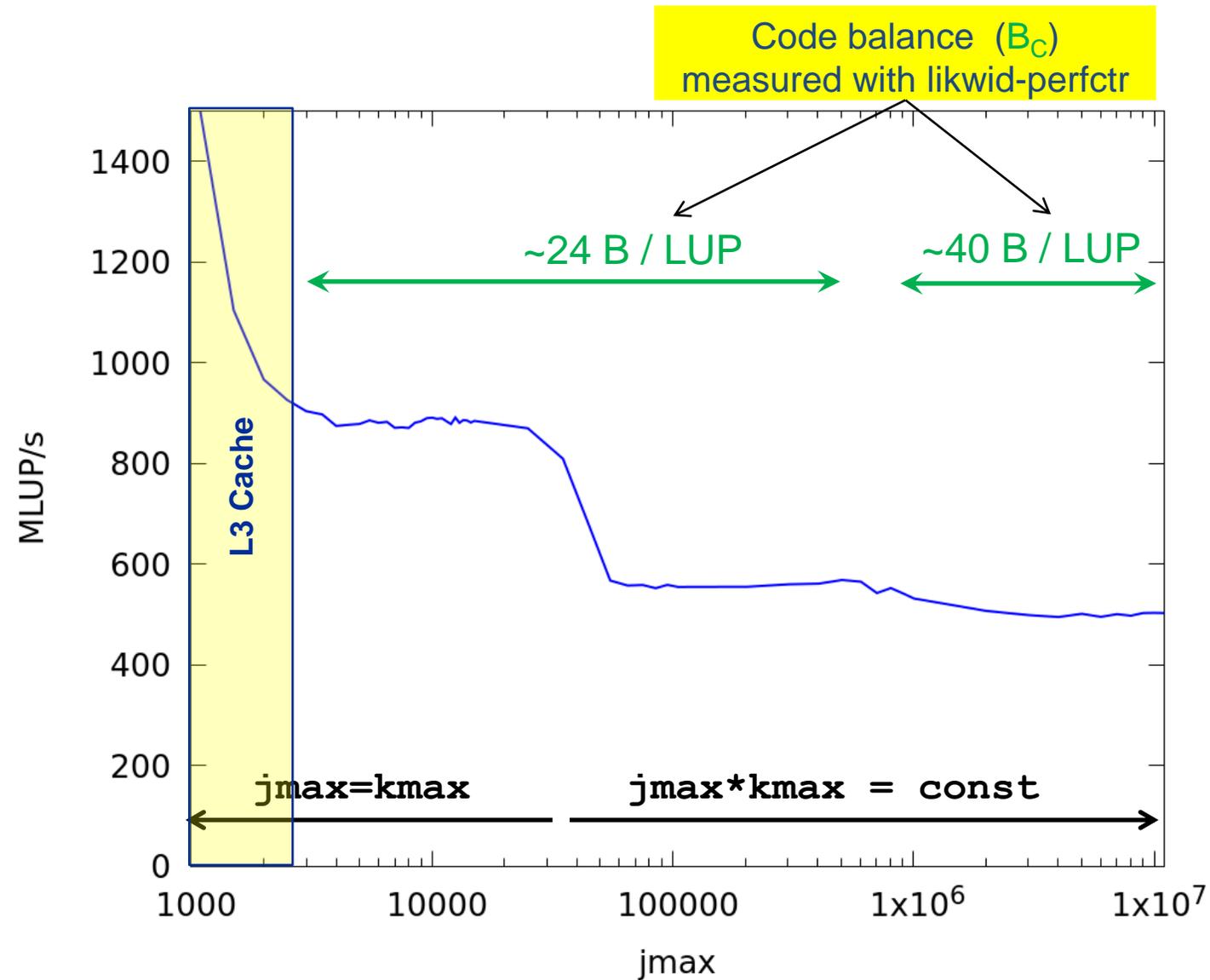
Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Jacobi 5-pt stencil 2D: Single-core performance



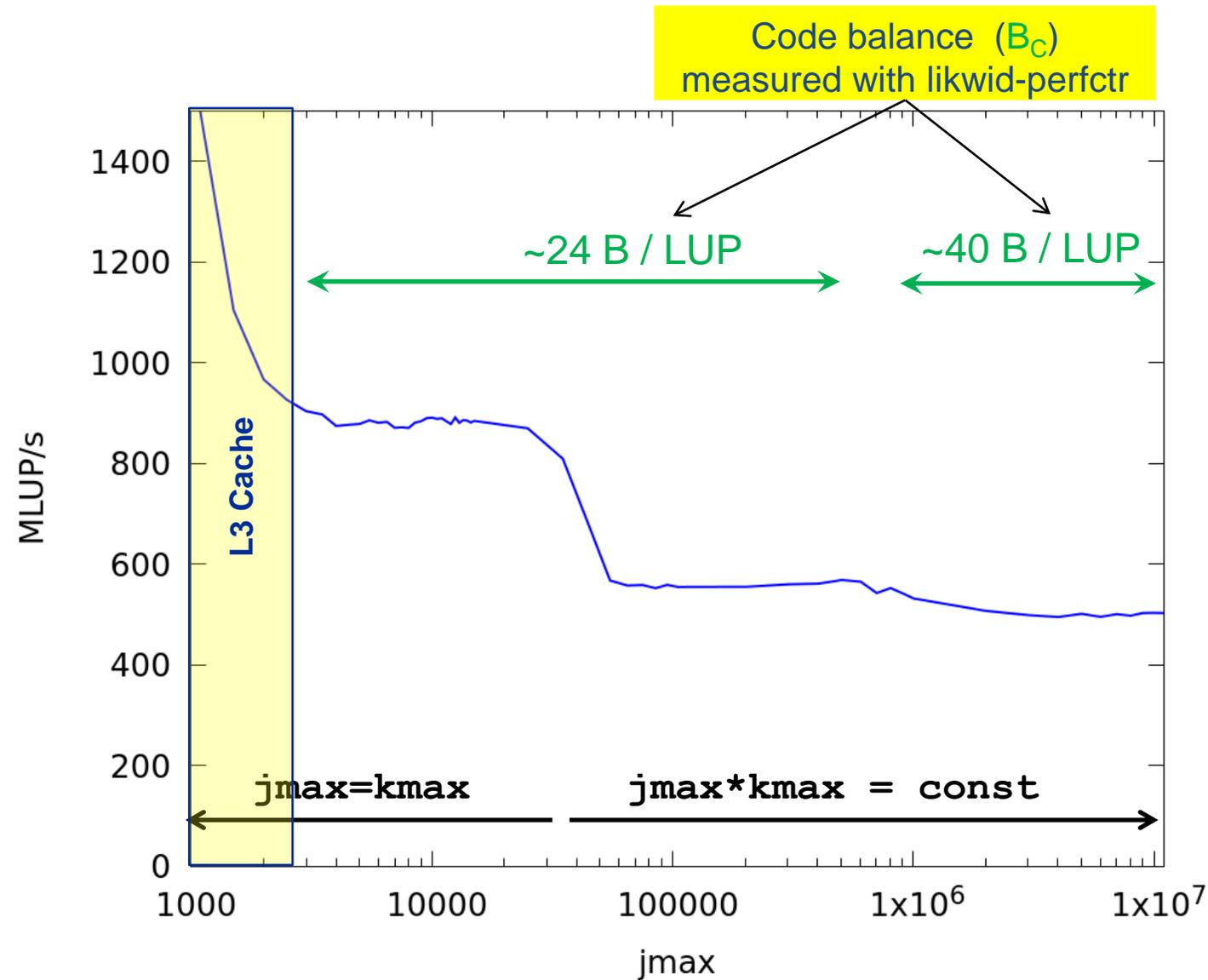
Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Jacobi 5-pt stencil 2D: Single-core performance



Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

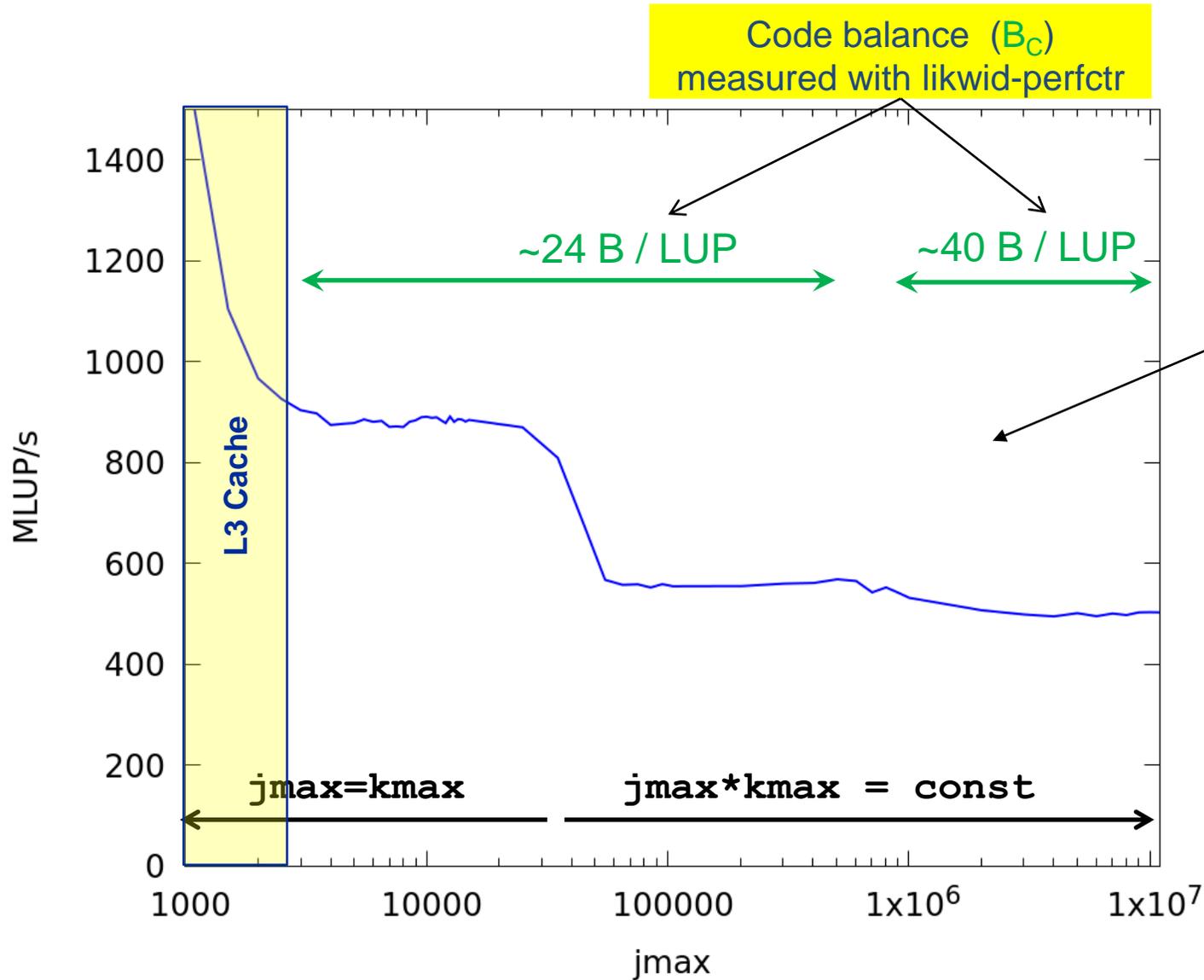
Jacobi 5-pt stencil 2D: Single-core performance



Questions:

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Jacobi 5-pt stencil 2D: Single-core performance



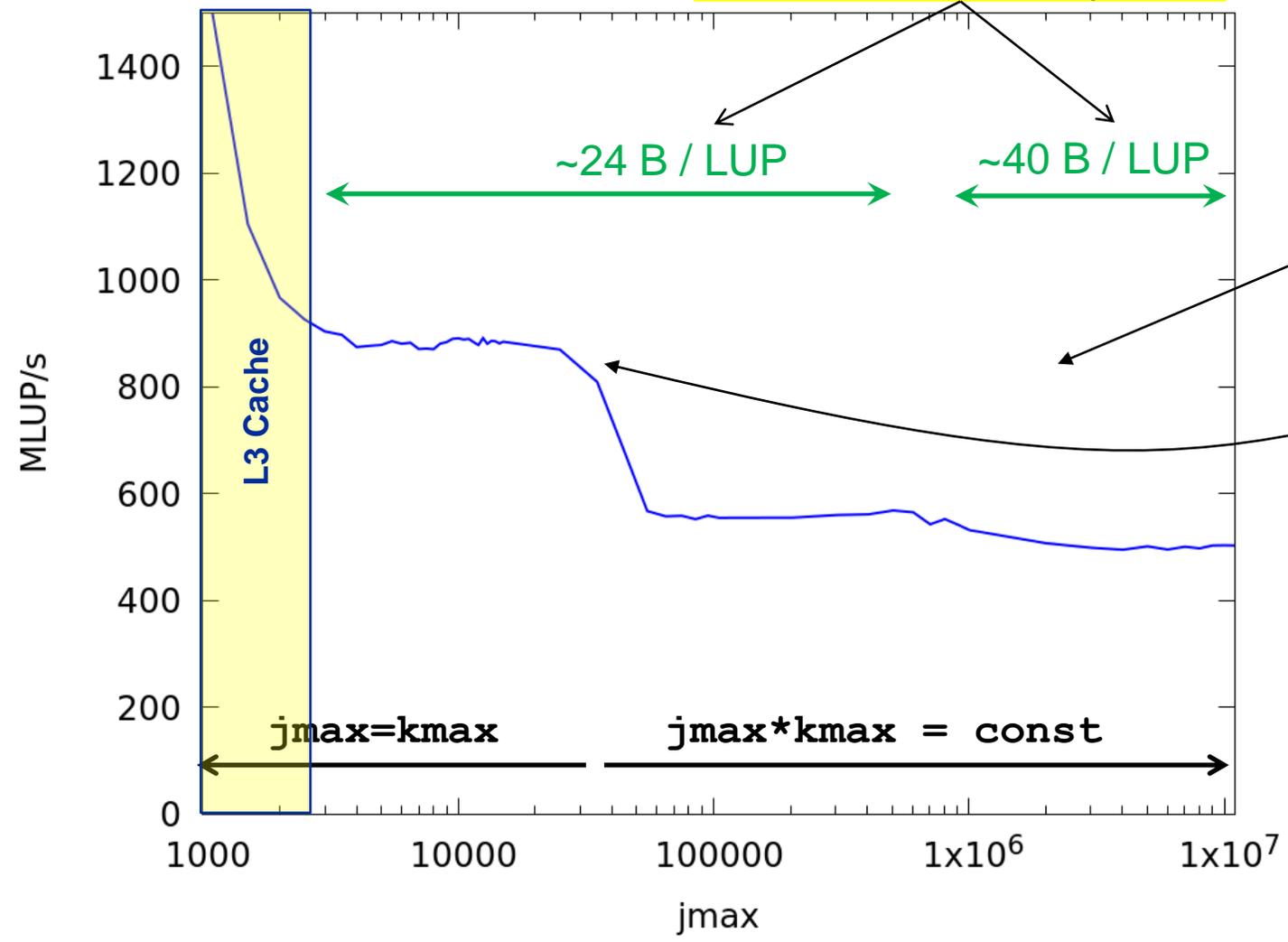
Questions:

1. How to achieve 24 B/LUP also for large j_{\max} ?

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Jacobi 5-pt stencil 2D: Single-core performance

Code balance (B_c)
measured with likwid-perfctr



Questions:

1. How to achieve 24 B/LUP also for large j_{\max} ?
2. How to sustain >800 MLUP/s for $j_{\max} > 10^4$?

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

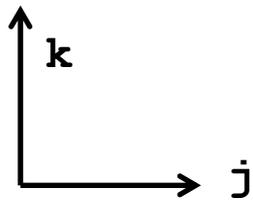
Case study: A Jacobi smoother

Layer conditions



Analyzing the data flow

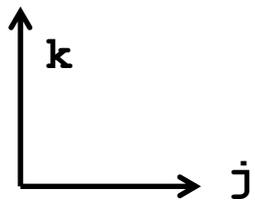
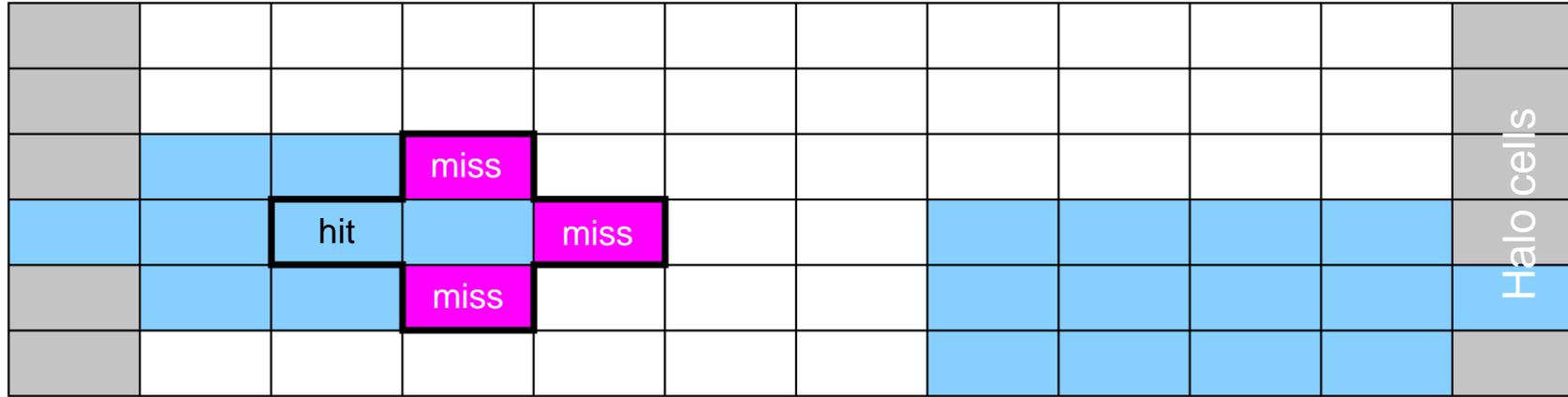
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume “Least Recently Used” replacement strategy)



$x(0:j_{\max}+1, 0:k_{\max}+1)$

Analyzing the data flow

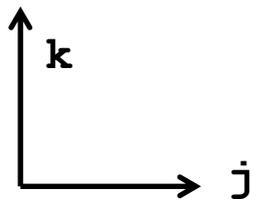
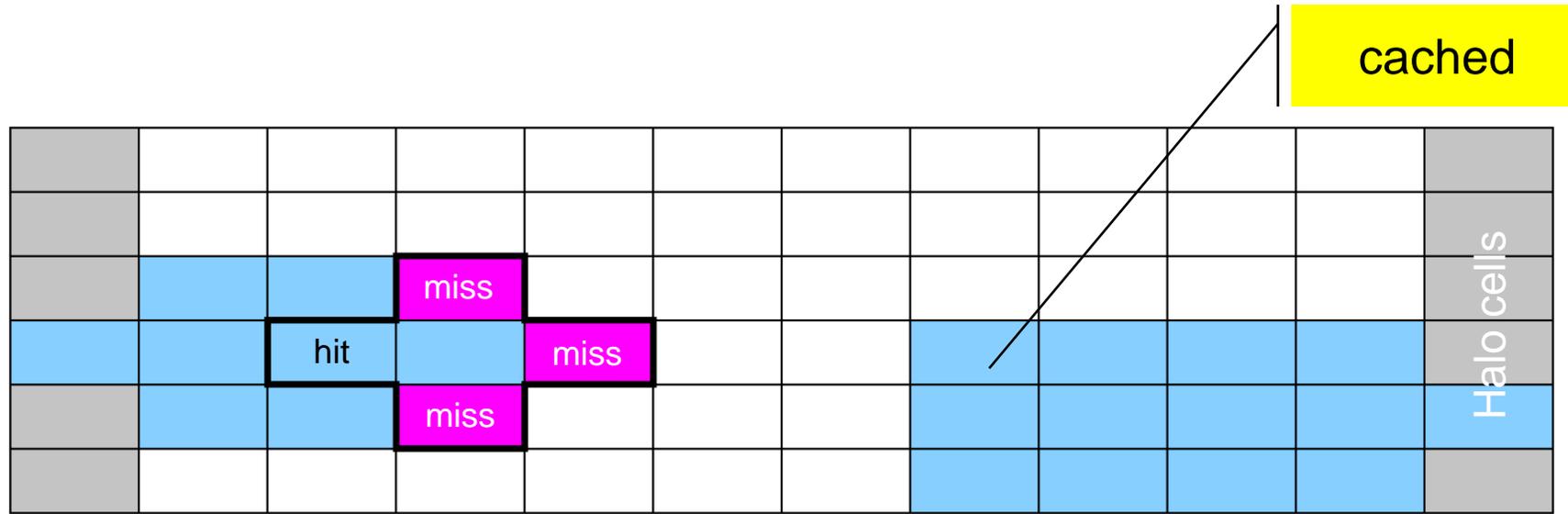
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume “Least Recently Used” replacement strategy)



$x(0:j_{\max}+1, 0:k_{\max}+1)$

Analyzing the data flow

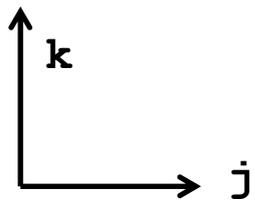
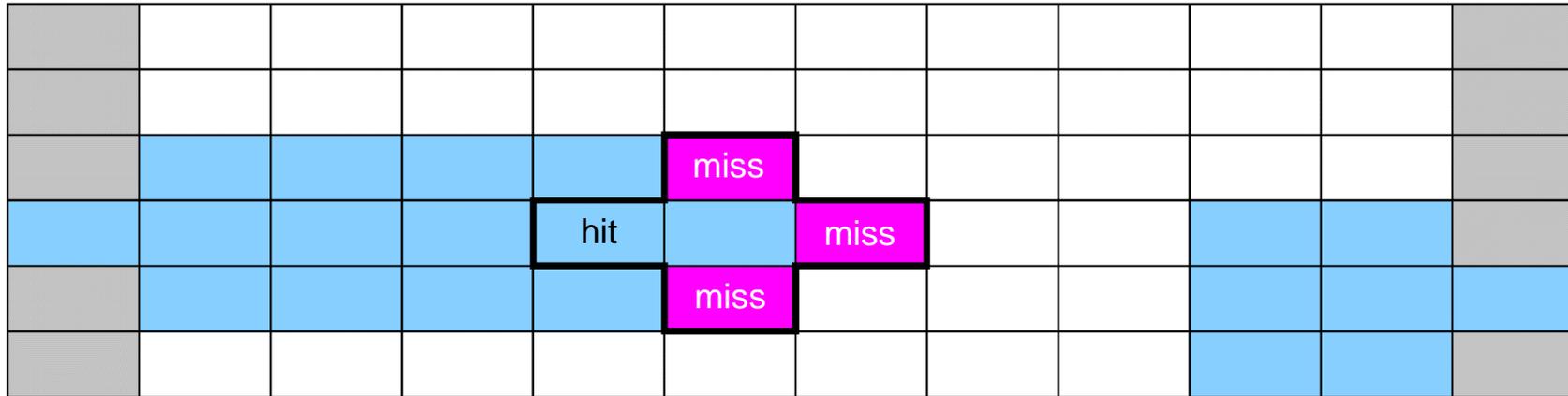
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume “Least Recently Used” replacement strategy)



$x(0:j_{\max}+1, 0:k_{\max}+1)$

Analyzing the data flow

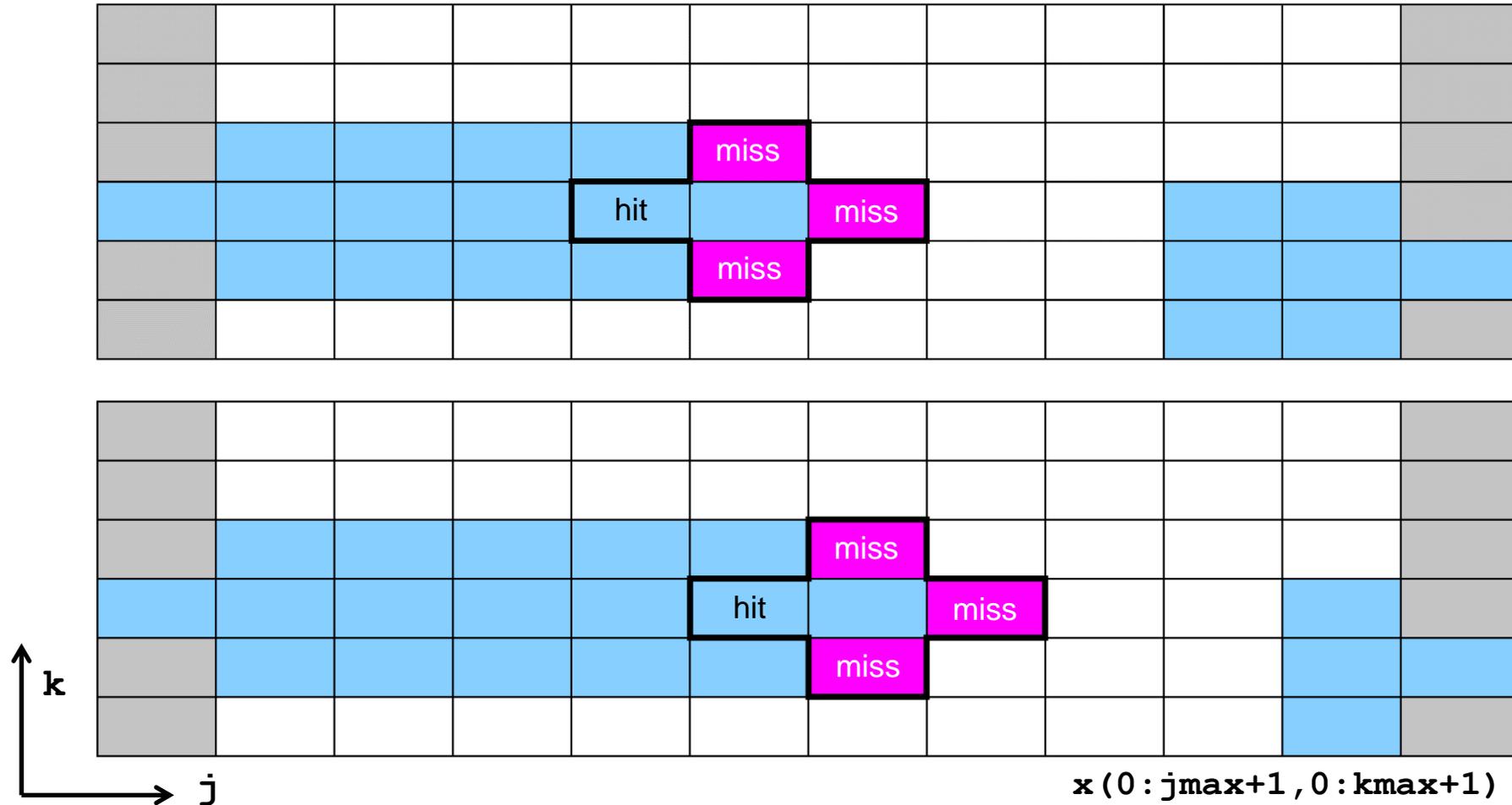
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume „Least Recently Used“ replacement strategy)



$x(0:jmax+1, 0:kmax+1)$

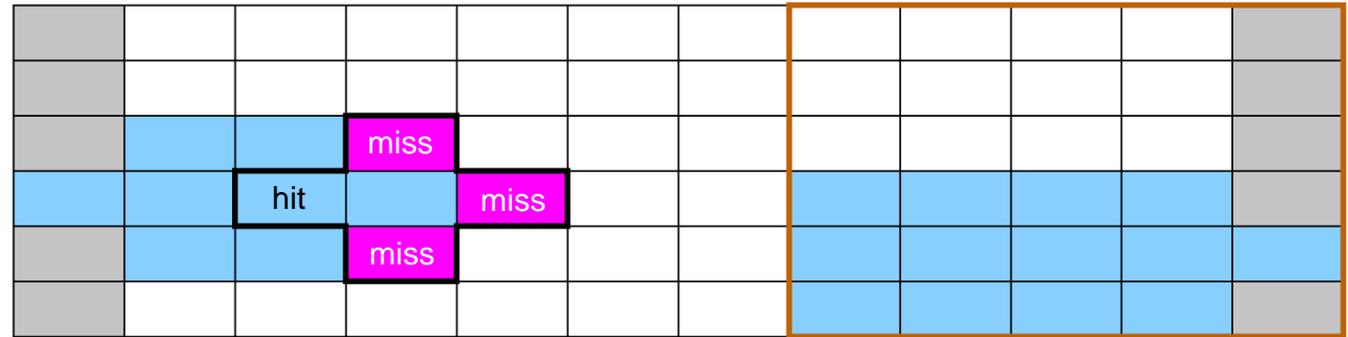
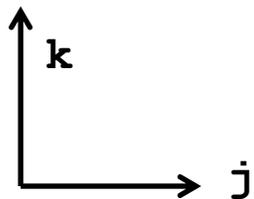
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume „Least Recently Used“ replacement strategy)

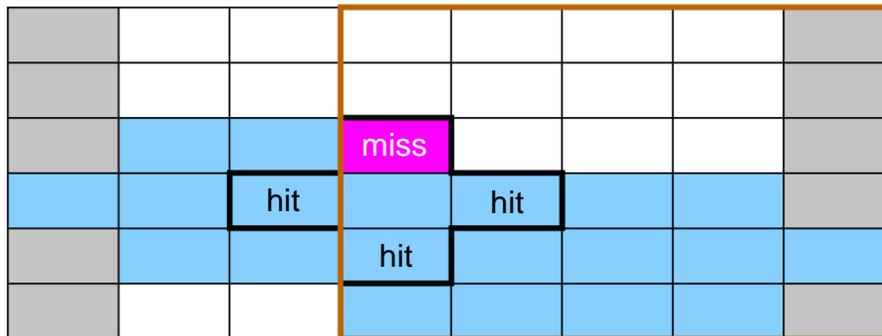


Analyzing the data flow

Reduce inner (j-) loop dimension successively



$x(0:j_{\max 1}+1, 0:k_{\max}+1)$



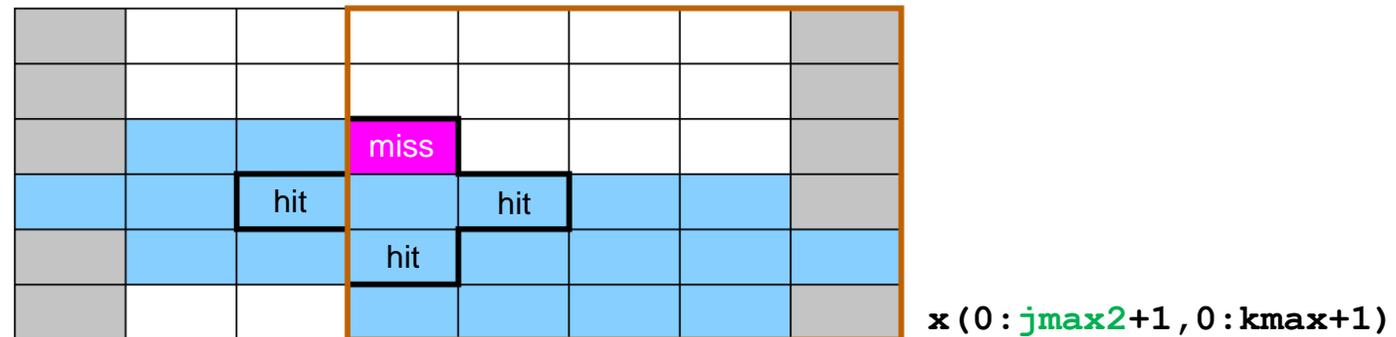
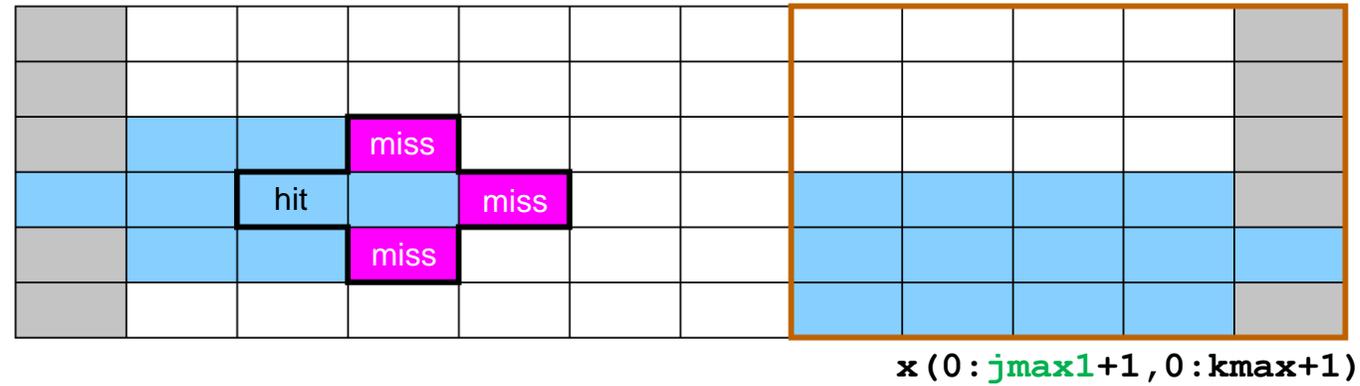
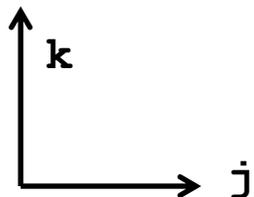
$x(0:j_{\max 2}+1, 0:k_{\max}+1)$

Analyzing the data flow

Reduce inner (j-) loop dimension successively



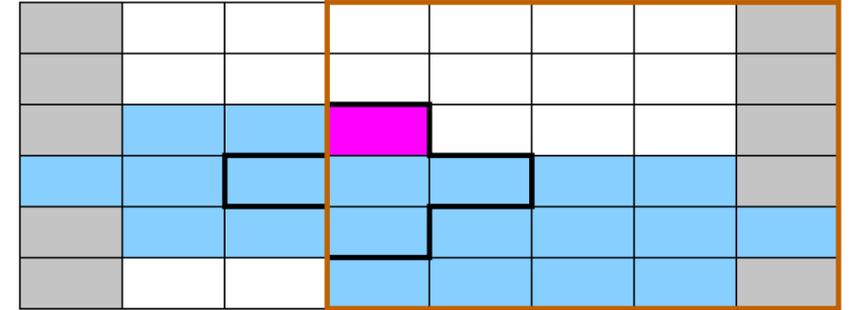
Best case: 3 "layers" of grid fit into the cache!



Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

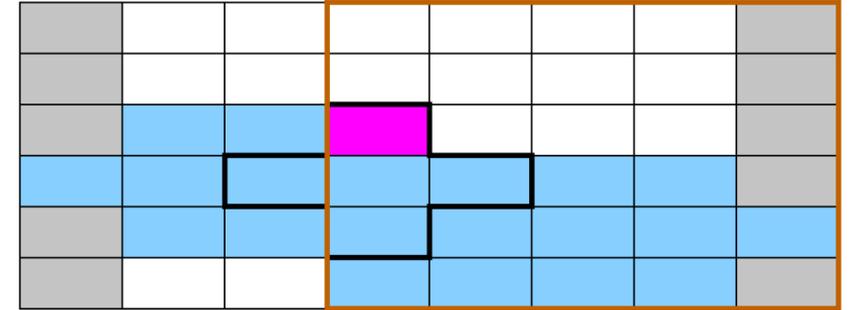
```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



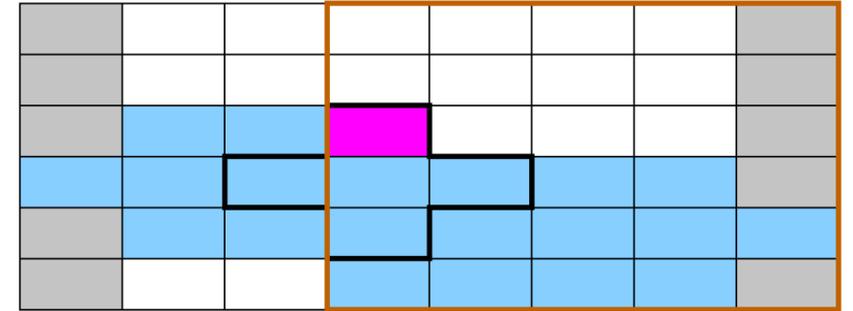
$$3 * j_{\max} * 8B < \text{CacheSize}/2$$

“Layer condition”

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



$$3 * j_{\max} * 8B < \text{CacheSize}/2$$

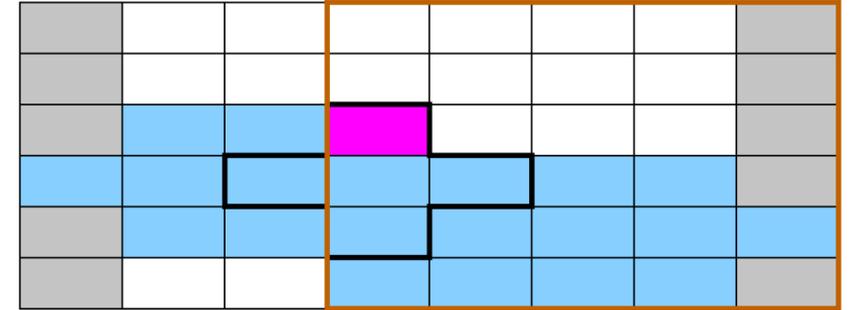
“Layer condition”

Safety margin
(Rule of thumb)

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



$$3 * j_{\max} * 8B < \text{CacheSize}/2$$

“Layer condition”

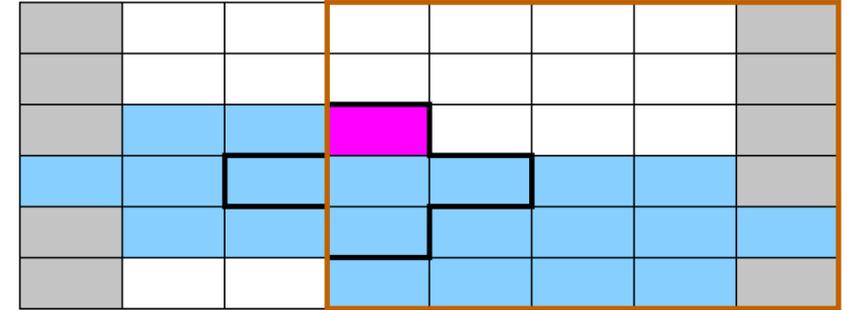
double
precision

Safety margin
(Rule of thumb)

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



$$3 * j_{\max} * 8B < \text{CacheSize}/2$$

“Layer condition”

3 rows of
j_{max}

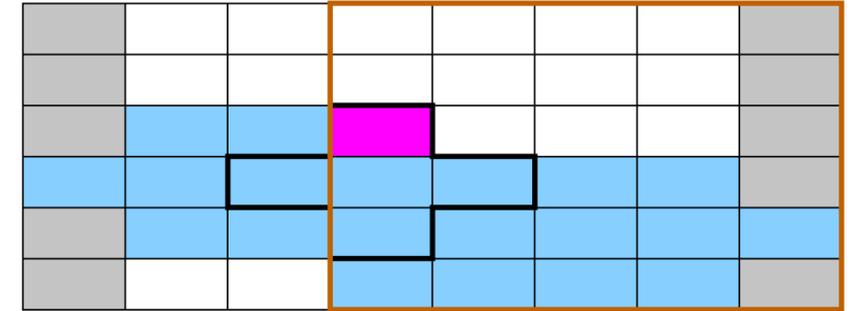
double
precision

Safety margin
(Rule of thumb)

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```



$$3 * j_{\max} * 8B < \text{CacheSize}/2$$

“Layer condition”

3 rows of
 j_{\max}

double
precision

Safety margin
(Rule of thumb)

Layer condition:

- Does not depend on outer loop length (k_{\max})
- No strict guideline (cache associativity, data traffic for y not included)
- Needs to be adapted for other stencils (e.g., long-range stencils)

Analyzing the data flow: Layer condition

$3 * j_{\max} * 8B < \text{CacheSize}/2$
Layer condition fulfilled?

Analyzing the data flow: Layer condition

$3 * j_{\max} * 8B < \text{CacheSize}/2$
Layer condition fulfilled?

NO

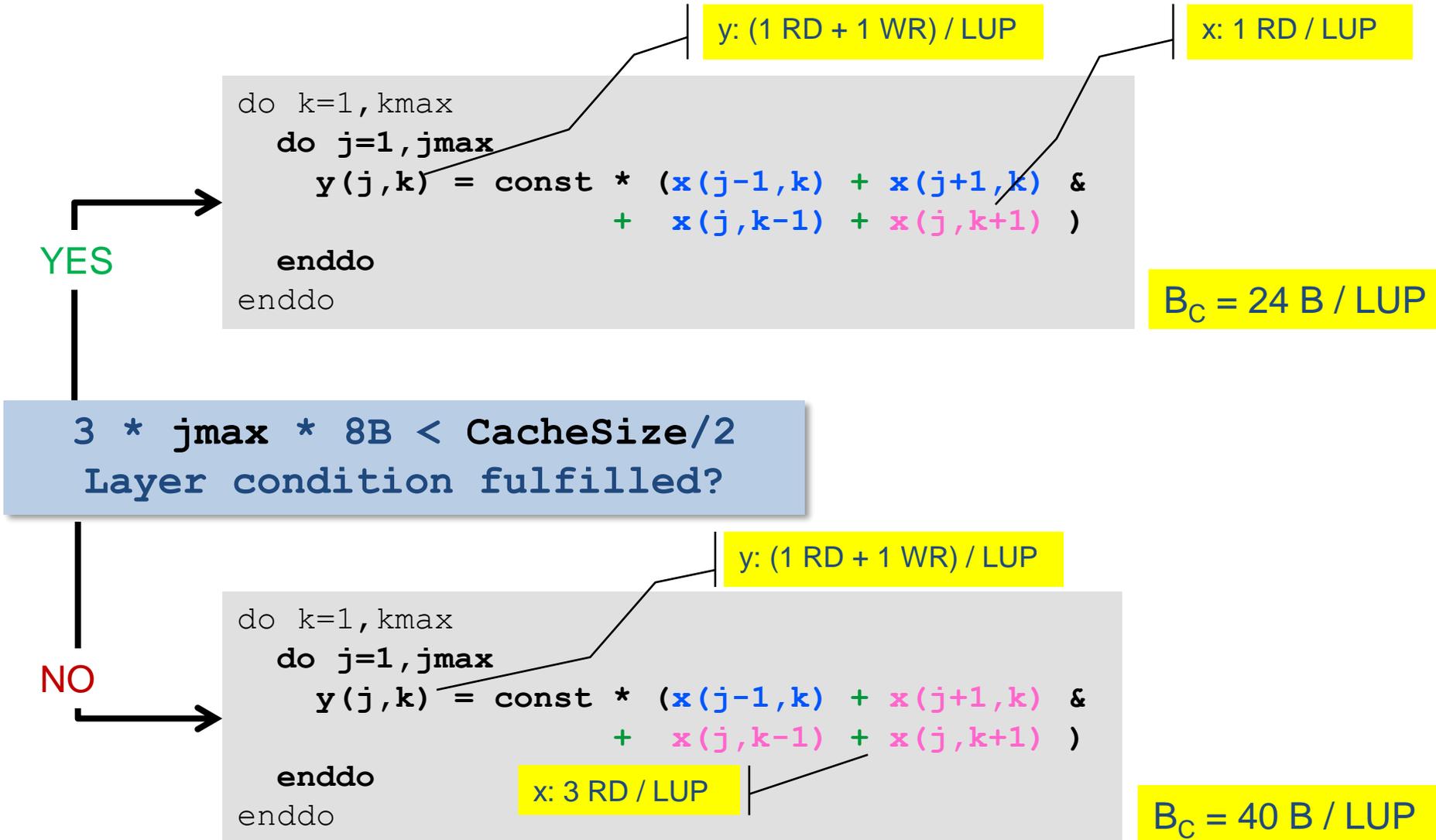
```
do k=1, kmax
  do j=1, jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      + x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

y: (1 RD + 1 WR) / LUP

x: 3 RD / LUP

$B_C = 40 B / LUP$

Analyzing the data flow: Layer condition



Case study: A Jacobi smoother

Optimization by spatial blocking



Enforcing a layer condition (2D 5-pt)

- How can we enforce a layer condition for all domain sizes ?
- Idea: **Spatial blocking**
 - Reuse elements of $\mathbf{x}()$ as long as they stay in cache
 - Sweep can be executed in any order, e.g. compute blocks in j-direction

Enforcing a layer condition (2D 5-pt)

- How can we enforce a layer condition for all domain sizes ?
- Idea: **Spatial blocking**
 - Reuse elements of $\mathbf{x}()$ as long as they stay in cache
 - Sweep can be executed in any order, e.g. compute blocks in j-direction

“Spatial Blocking” of j-loop:

```
do jb=1, jmax, jbblock !
  do k=1, kmax
    do j= jb, min(jb+jbblock-1, jmax) !inner loop length jbblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                       + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

Enforcing a layer condition (2D 5-pt)

- How can we enforce a layer condition for all domain sizes ?
- Idea: **Spatial blocking**
 - Reuse elements of $\mathbf{x}()$ as long as they stay in cache
 - Sweep can be executed in any order, e.g. compute blocks in j-direction

“Spatial Blocking” of j-loop:

```
do jb=1, jmax, jblock !
  do k=1, kmax
    do j= jb, min(jb+jblock-1, jmax) !inner loop length jblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
        + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

New layer condition (blocking)
 $3 * jblock * 8B < CacheSize/2$

Enforcing a layer condition (2D 5-pt)

- How can we enforce a layer condition for all domain sizes ?
- Idea: **Spatial blocking**
 - Reuse elements of $\mathbf{x}()$ as long as they stay in cache
 - Sweep can be executed in any order, e.g. compute blocks in j-direction

“Spatial Blocking” of j-loop:

```
do jb=1, jmax, jbblock !
  do k=1, kmax
    do j= jb, min(jb+jbblock-1, jmax) !inner loop length jbblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
        + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

New layer condition (blocking)
 $3 * \mathbf{jb}block * 8B < \mathbf{CacheSize}/2$

Determine for given **CacheSize** an appropriate **jb**block value:

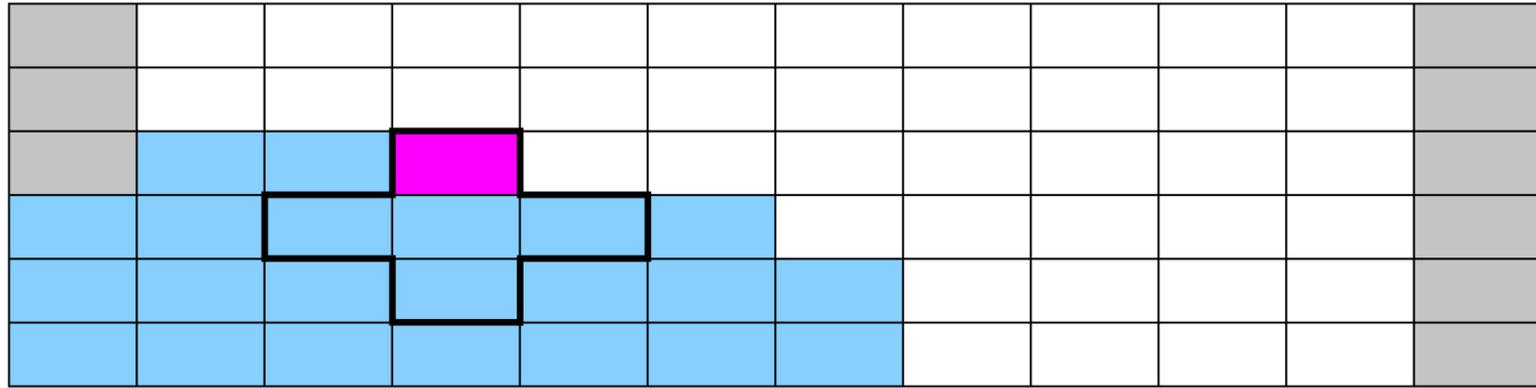
$\mathbf{jb}block < \mathbf{CacheSize} / 48B$

Establish the layer condition by blocking

Split
domain into
subblocks:

Establish the layer condition by blocking

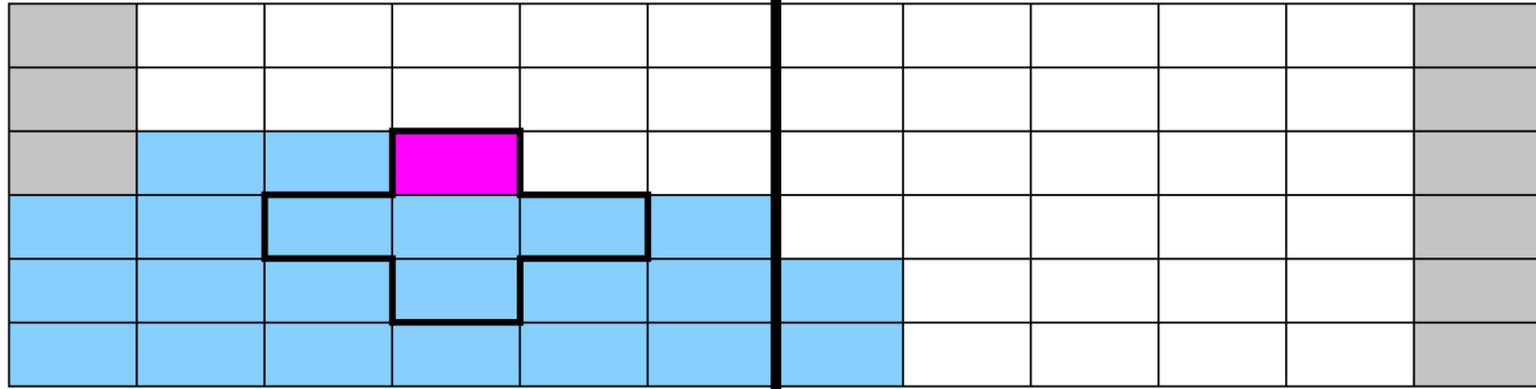
Split
domain into
subblocks:



Establish the layer condition by blocking

Split
domain into
subblocks:

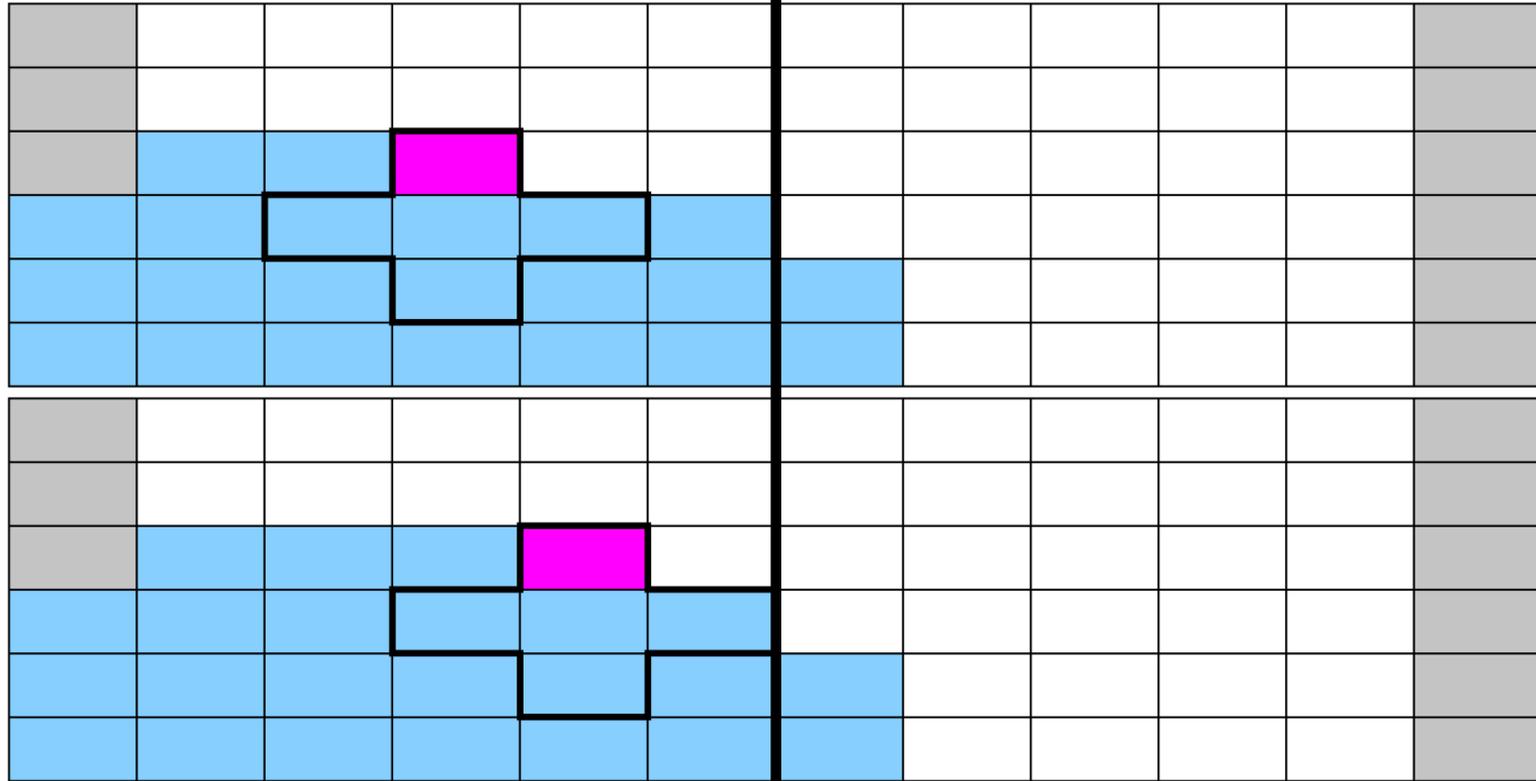
e.g. block
size = 5



Establish the layer condition by blocking

Split domain into subblocks:

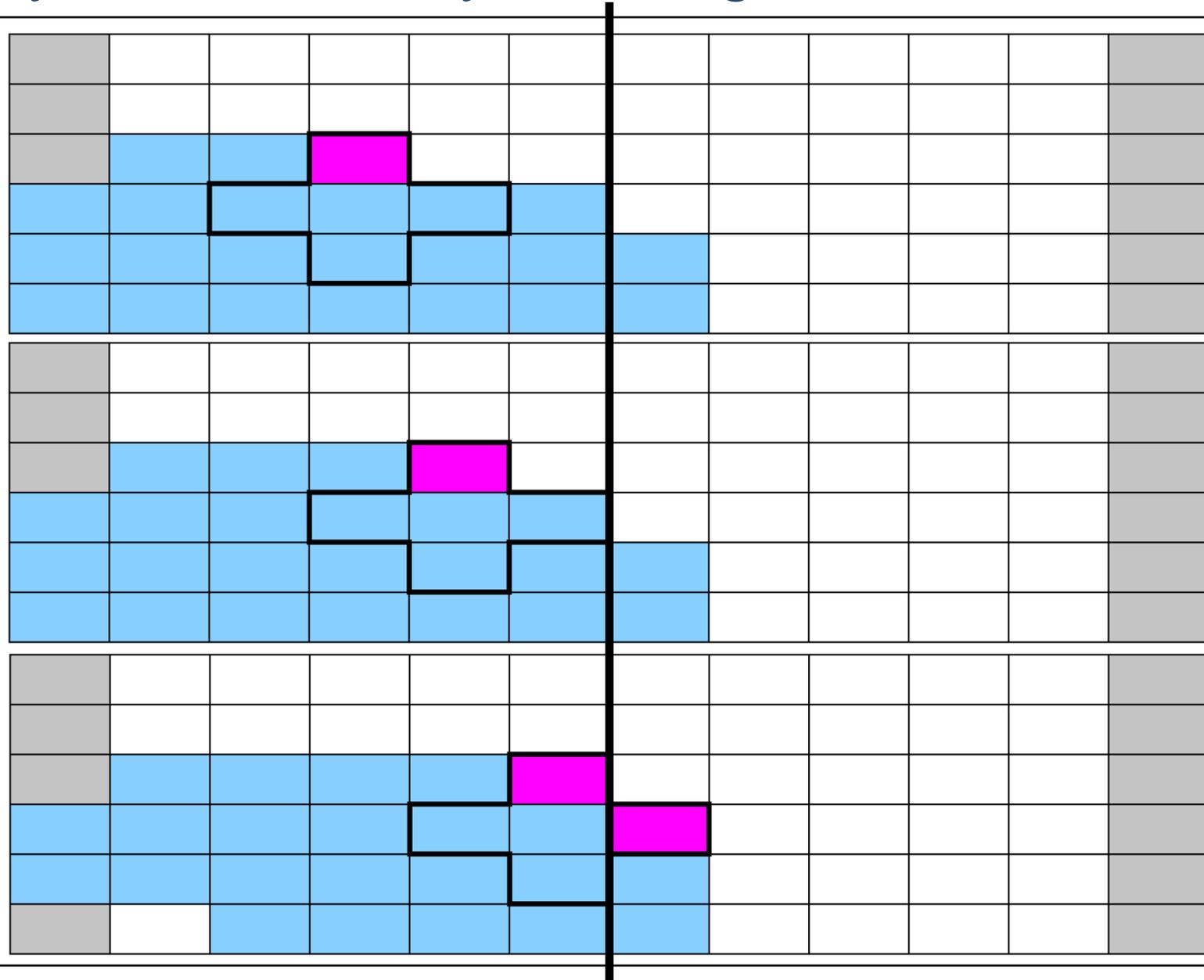
e.g. block size = 5



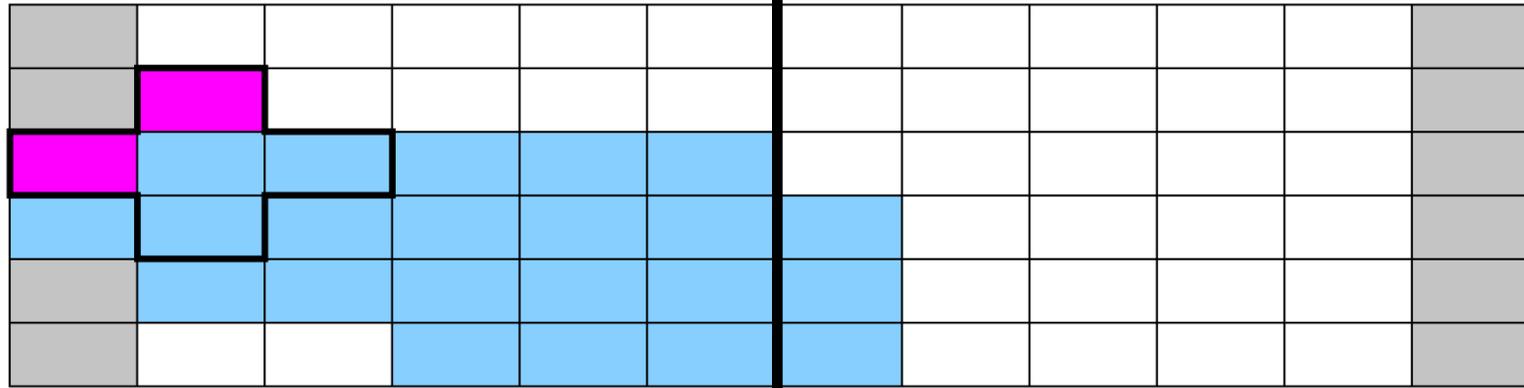
Establish the layer condition by blocking

Split domain into subblocks:

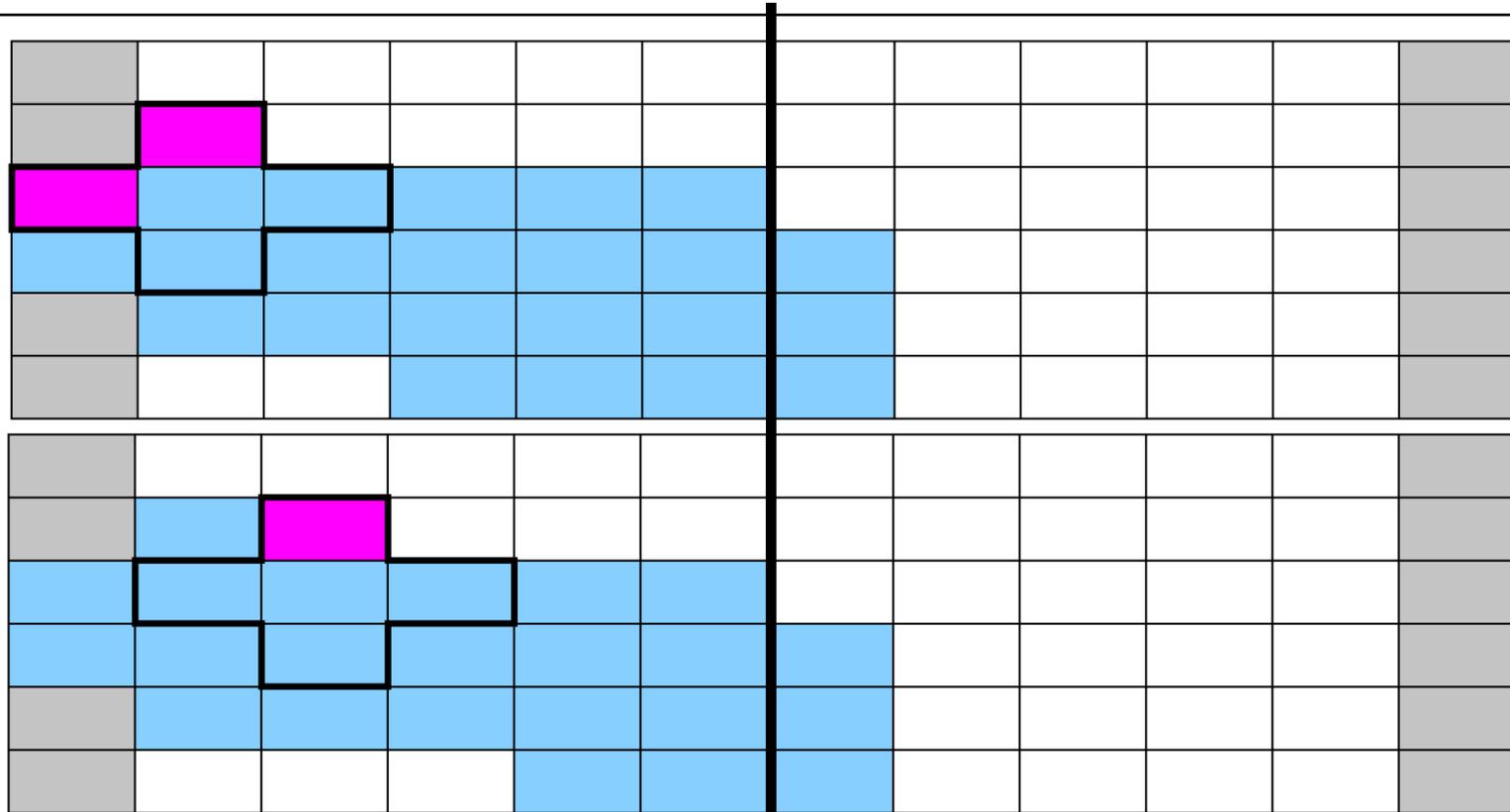
e.g. block size = 5



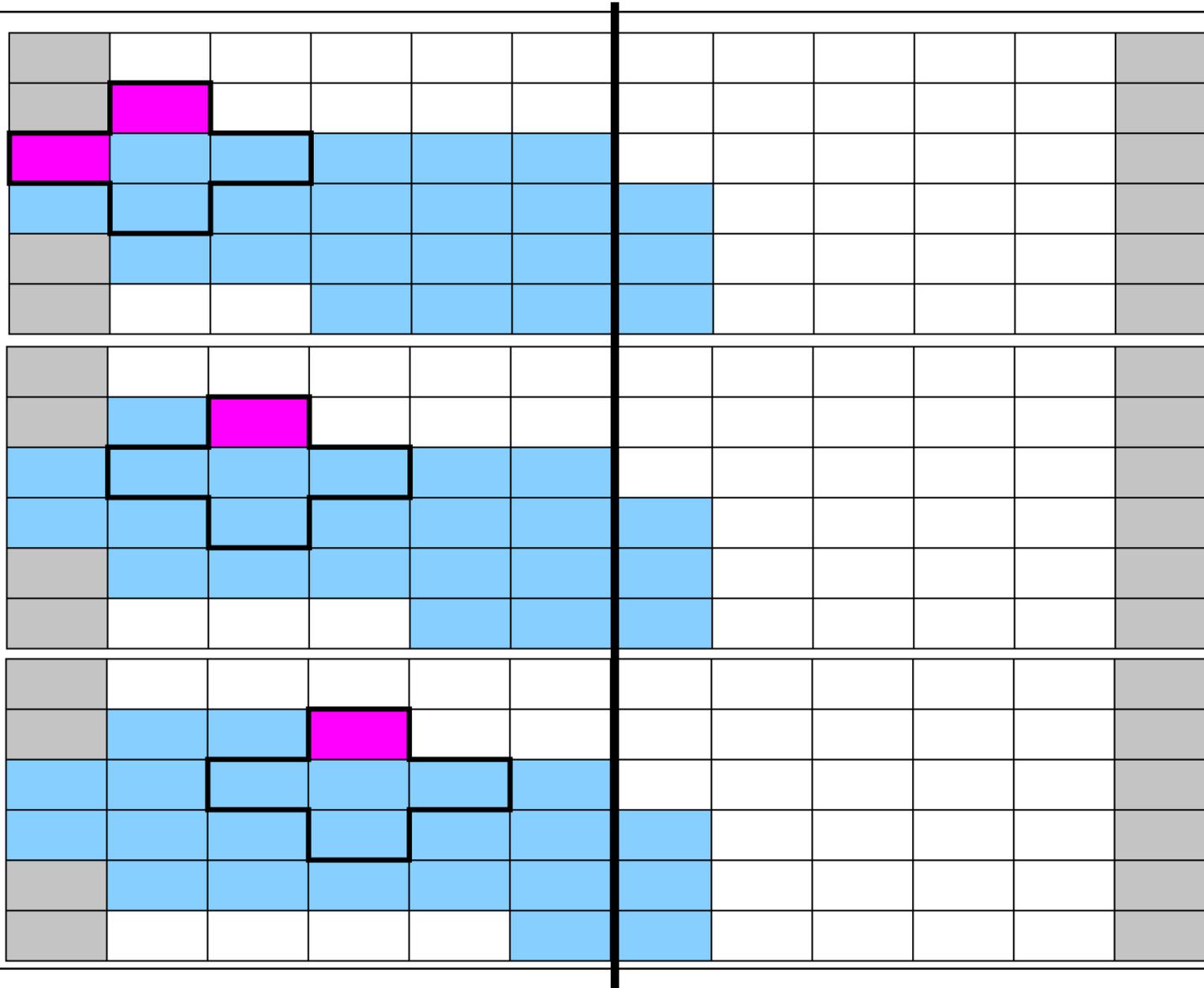
Establish the layer condition by blocking



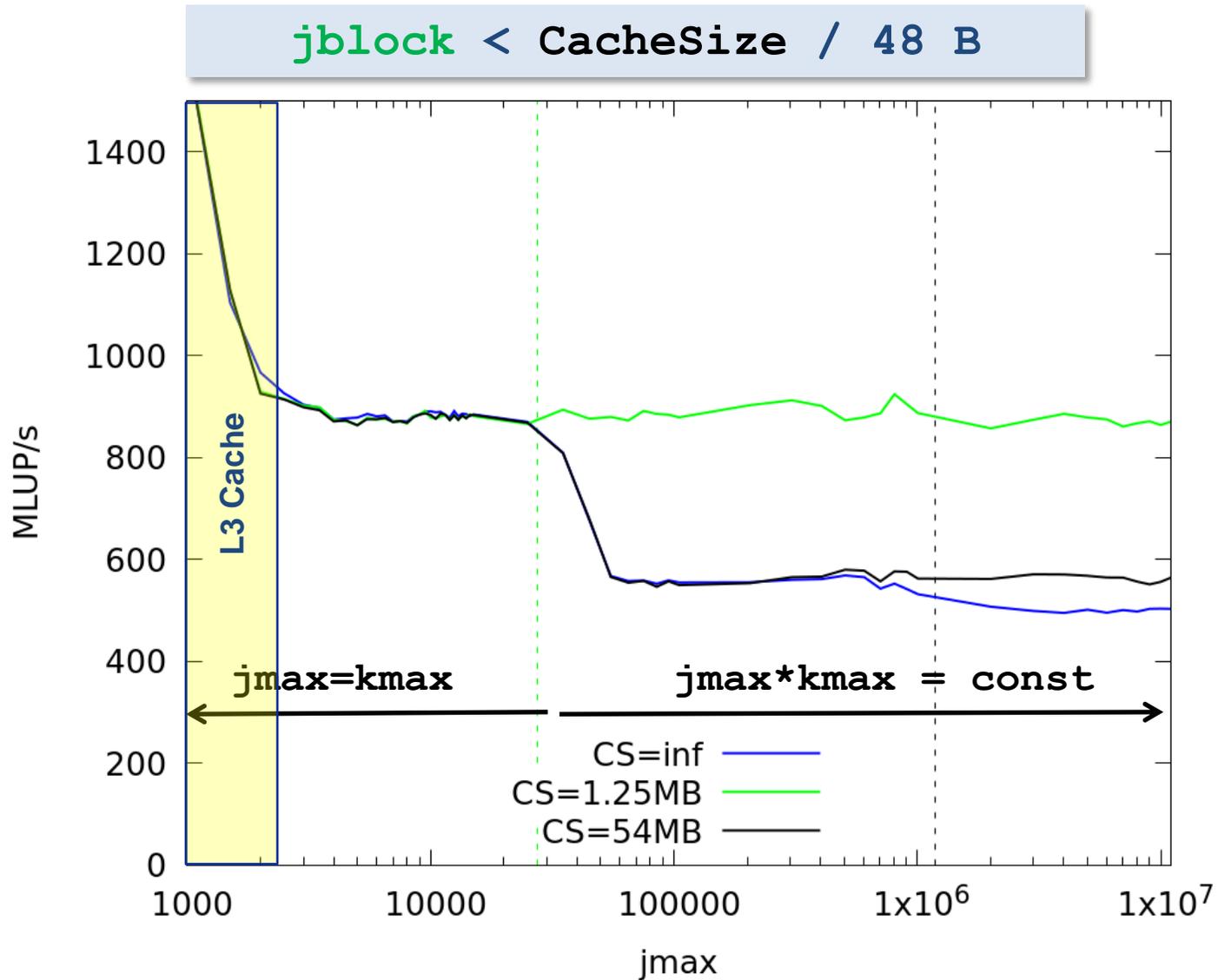
Establish the layer condition by blocking



Establish the layer condition by blocking



Establish layer condition by spatial blocking

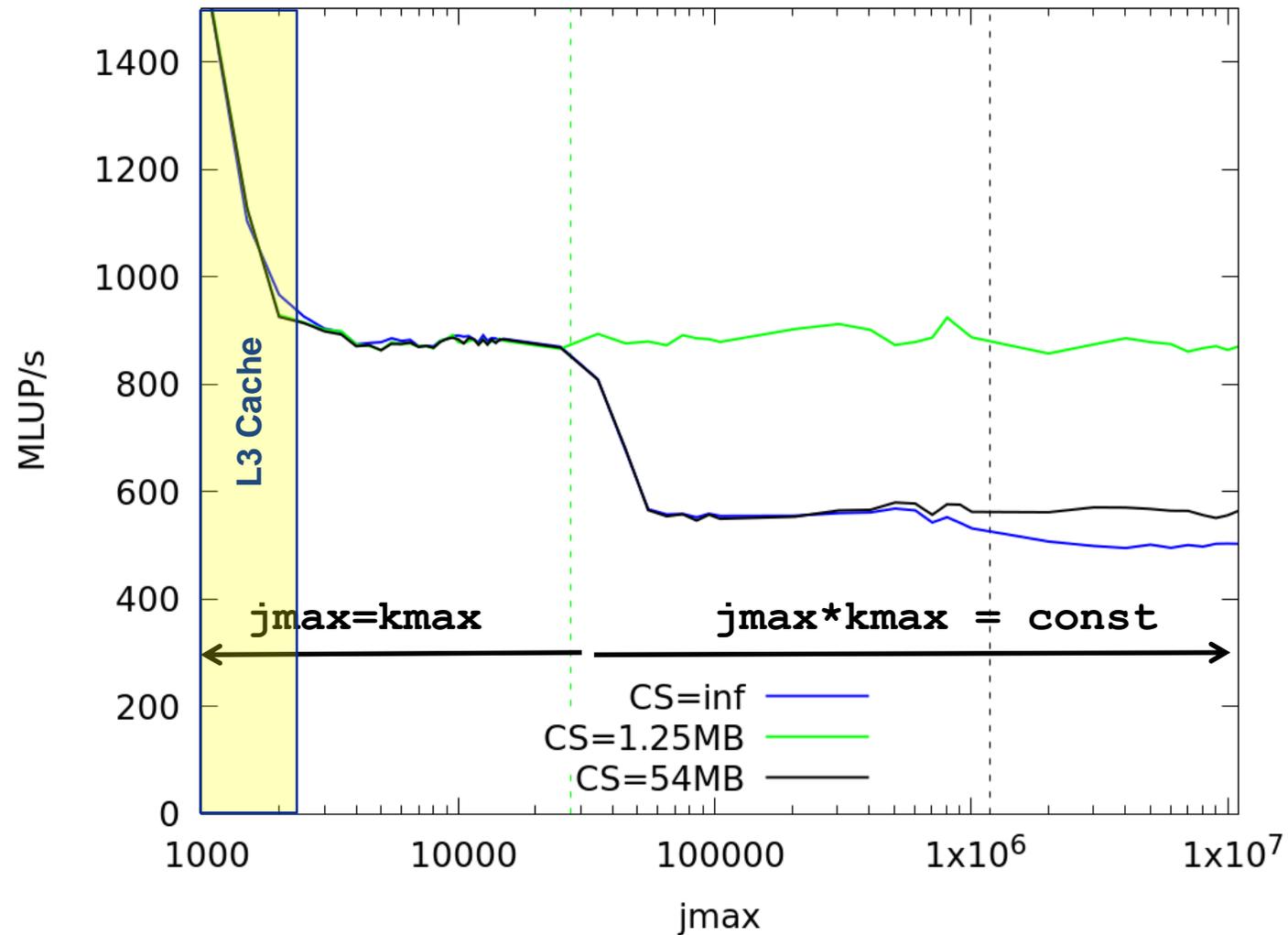


Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Establish layer condition by spatial blocking

$jblock < CacheSize / 48 B$

Which cache to block for?

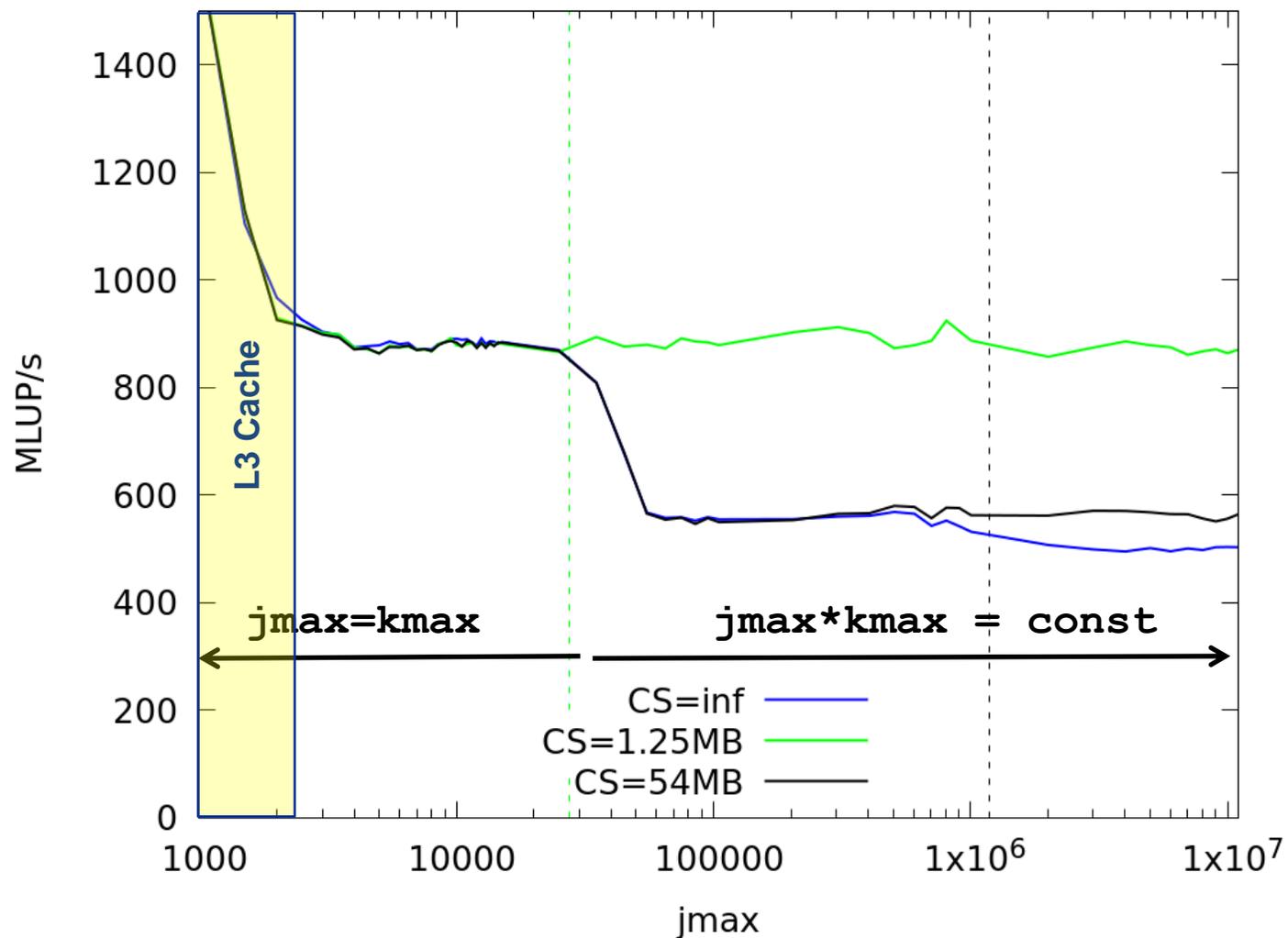


Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Establish layer condition by spatial blocking

$jblock < CacheSize / 48 B$

Which cache to block for?



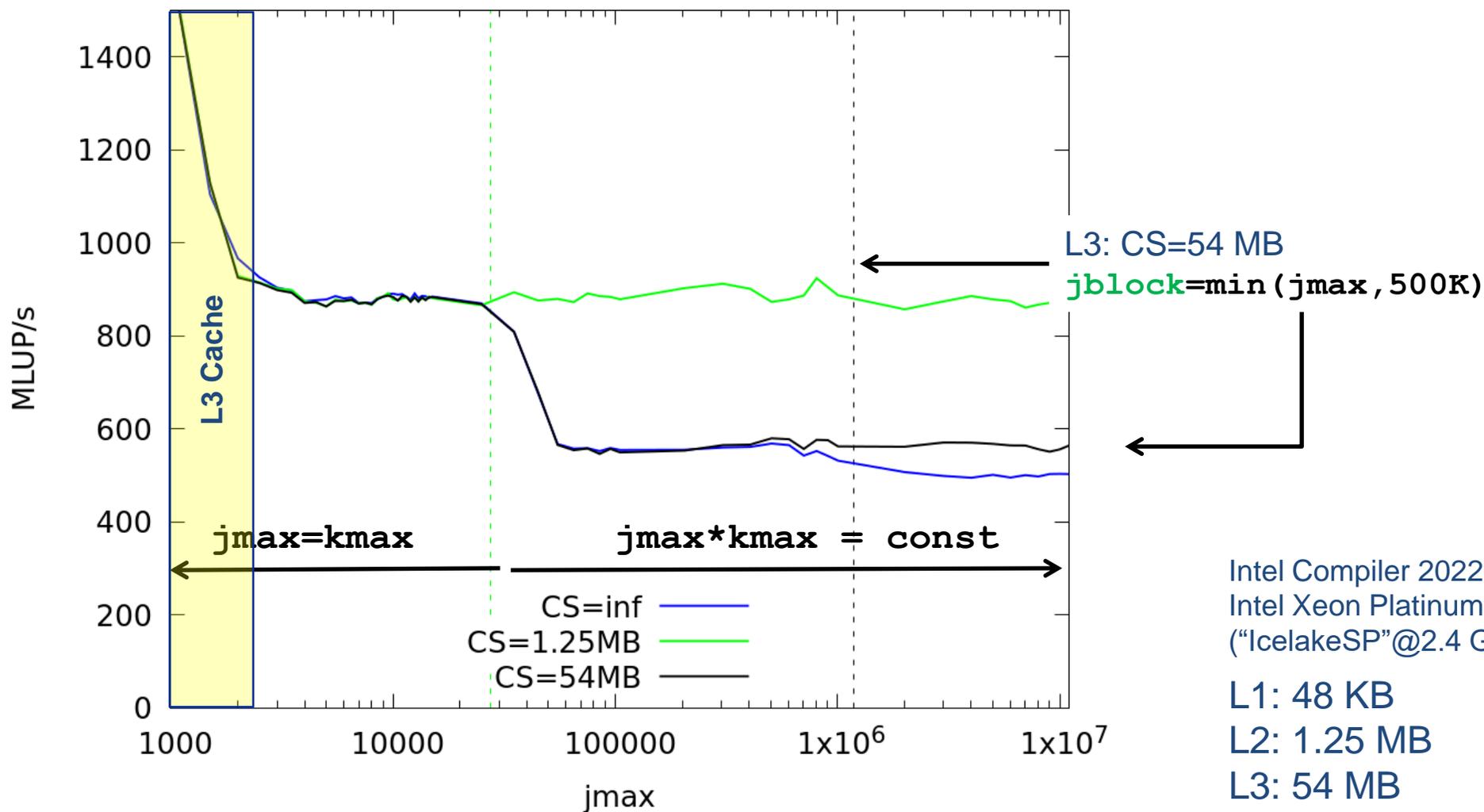
Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

L1: 48 KB
L2: 1.25 MB
L3: 54 MB

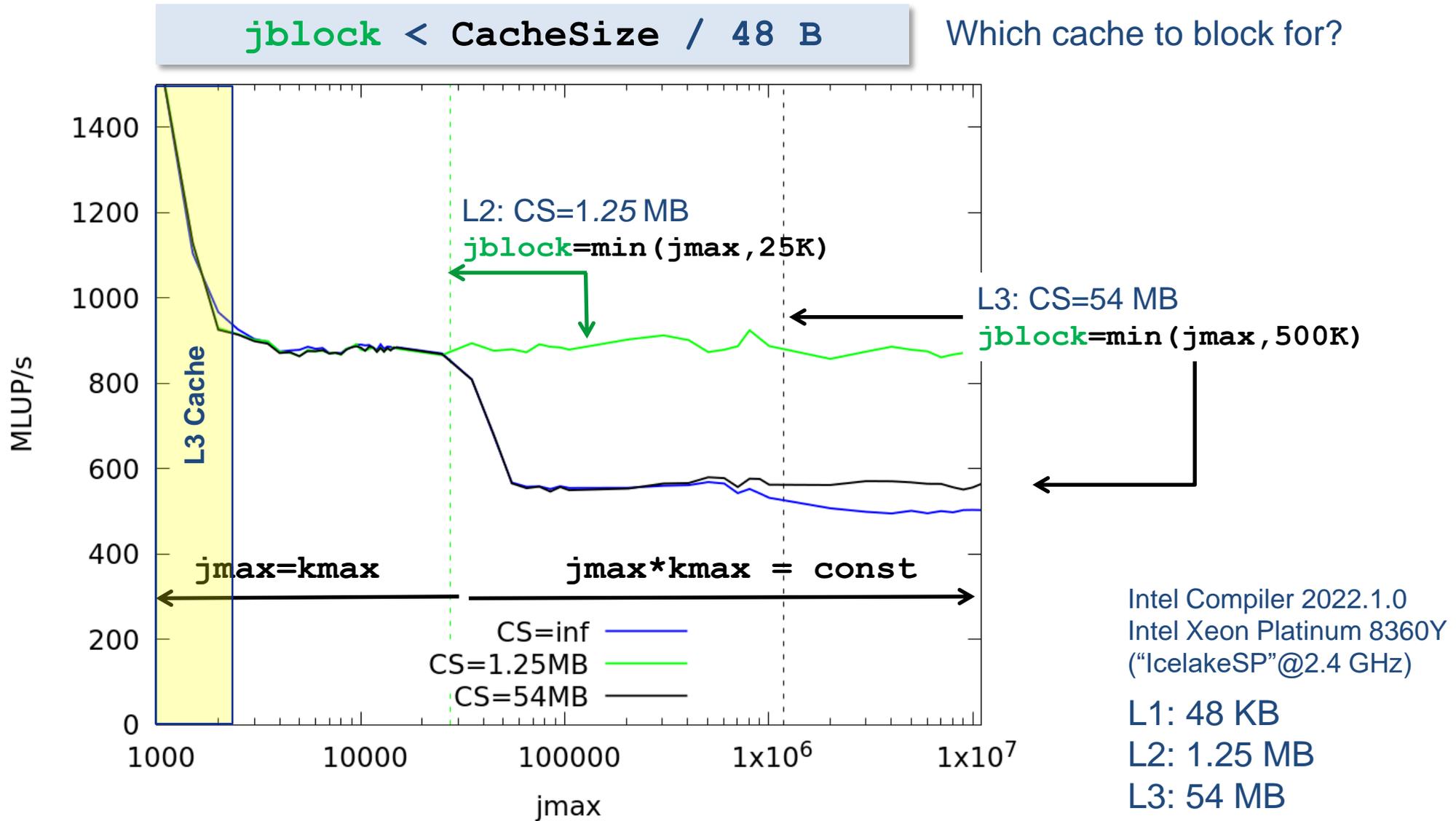
Establish layer condition by spatial blocking

$$jblock < CacheSize / 48 B$$

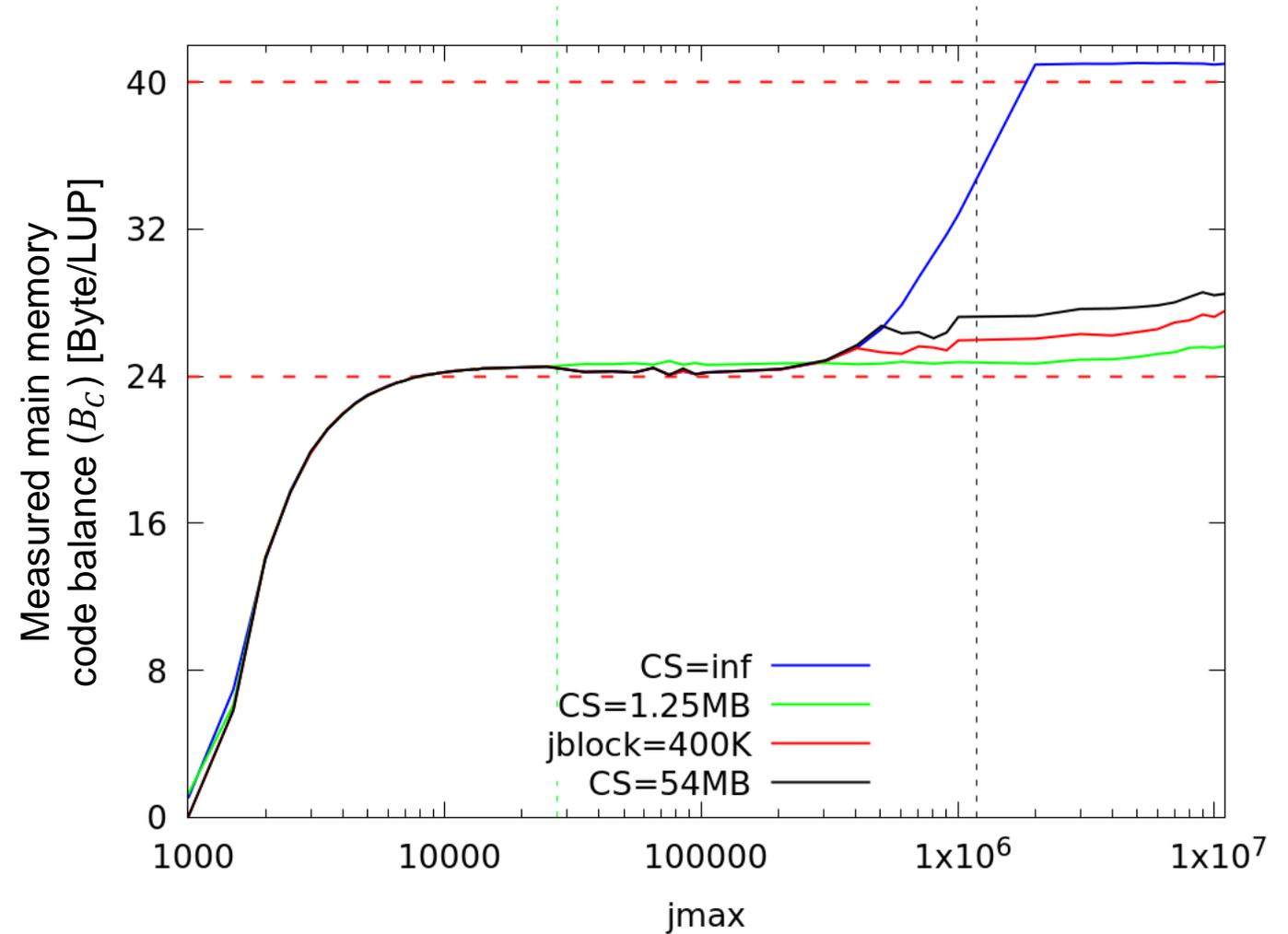
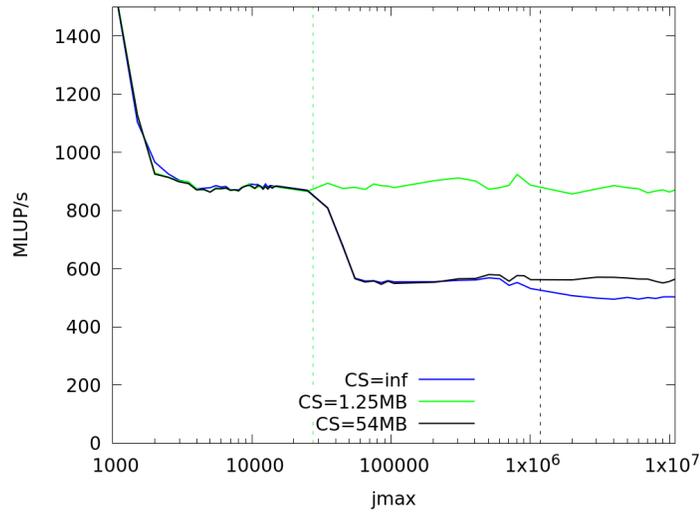
Which cache to block for?



Establish layer condition by spatial blocking

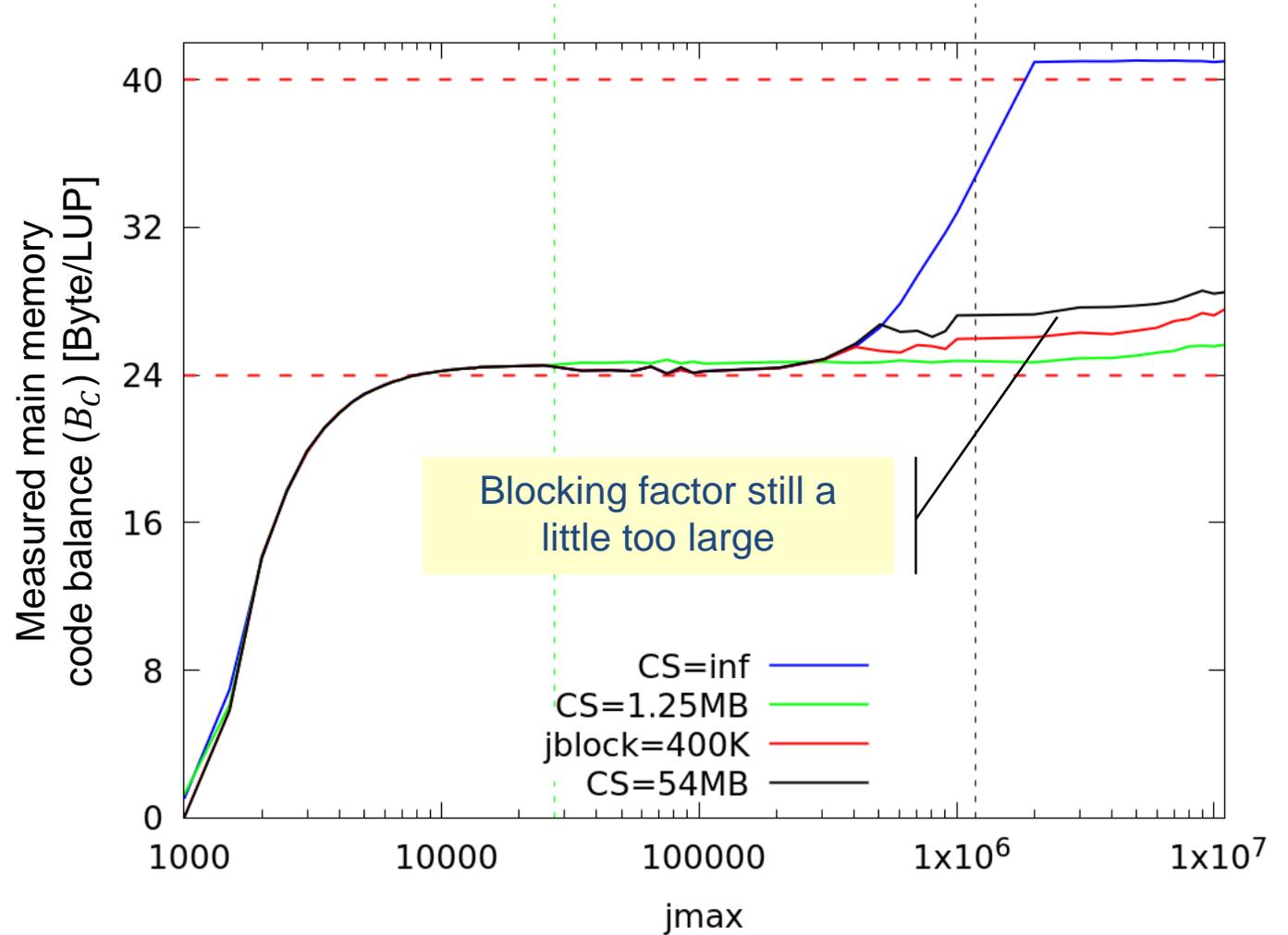
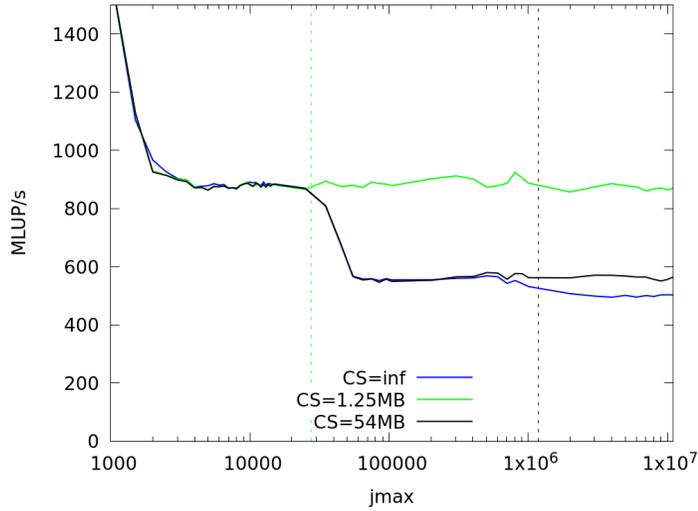


Validating the model: Memory code balance



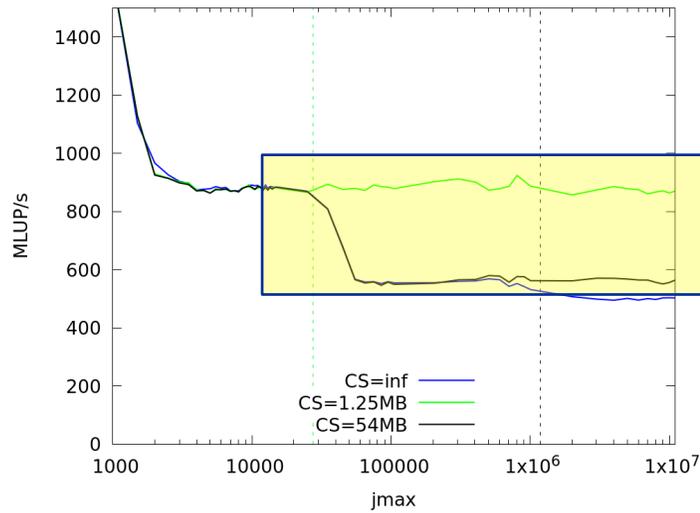
Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Validating the model: Memory code balance

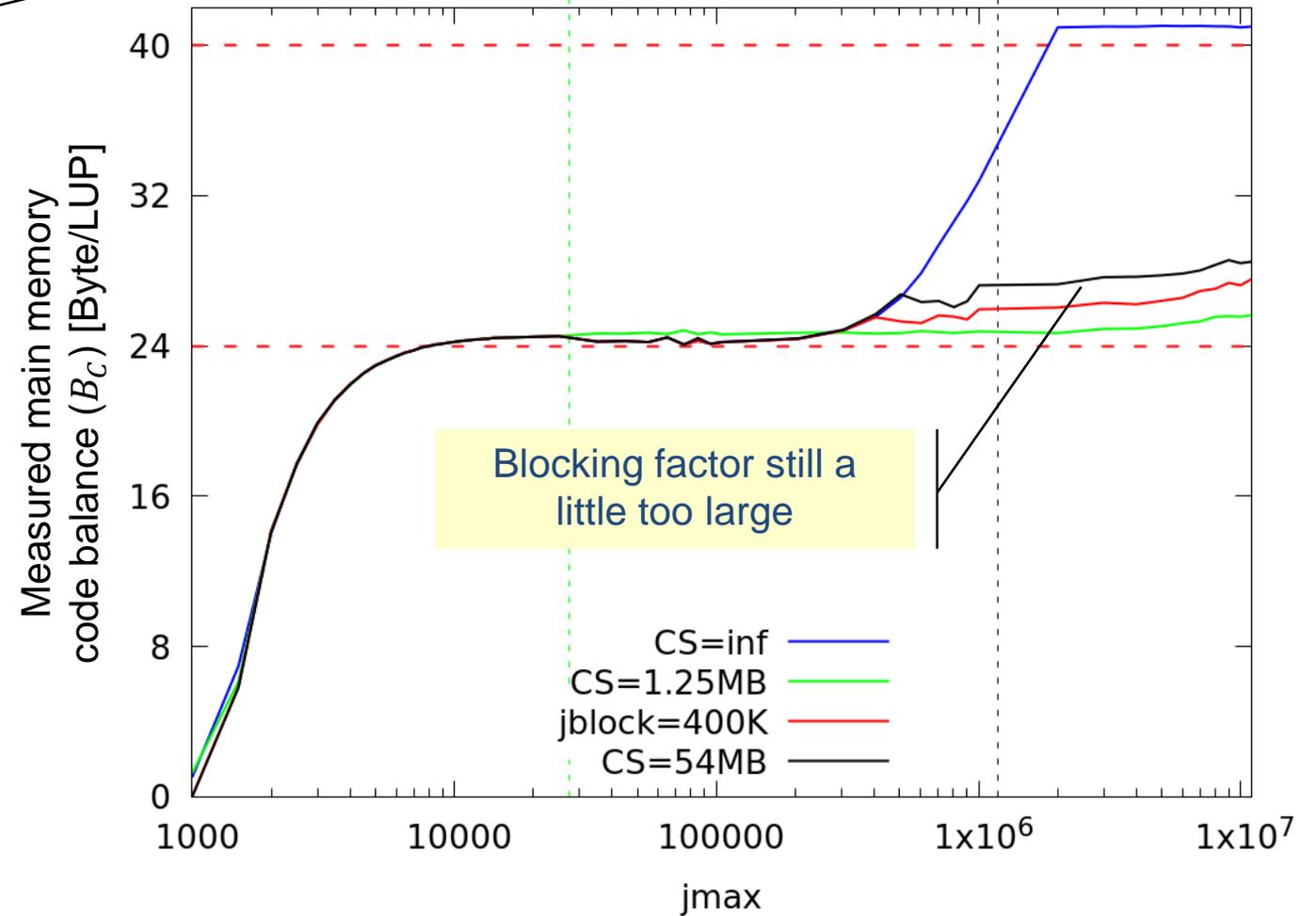


Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Validating the model: Memory code balance



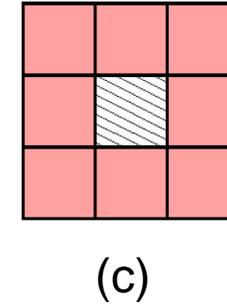
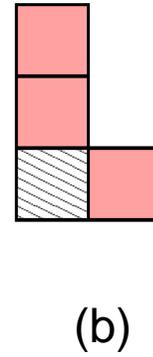
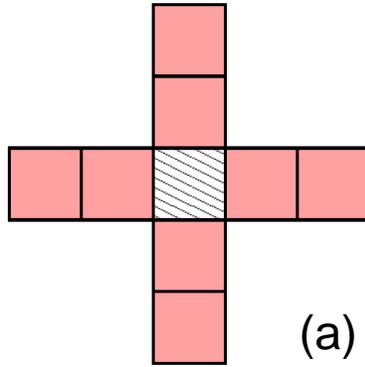
Main memory access is not reason for different performance (but L3 access is!)



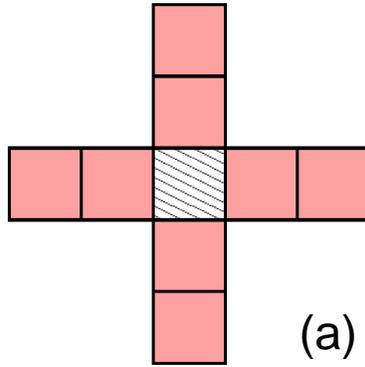
Blocking factor still a little too large

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

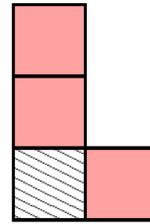
Stencil shapes and layer conditions in 2D



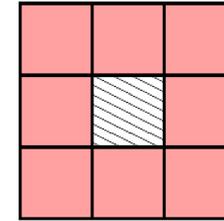
Stencil shapes and layer conditions in 2D



(a)



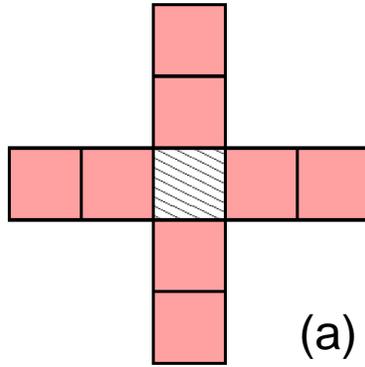
(b)



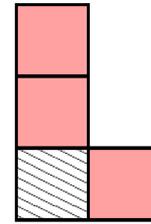
(c)

a) Long-range $r = 2$: 5 layers ($2r + 1$)

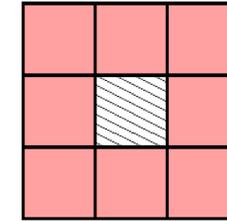
Stencil shapes and layer conditions in 2D



(a)



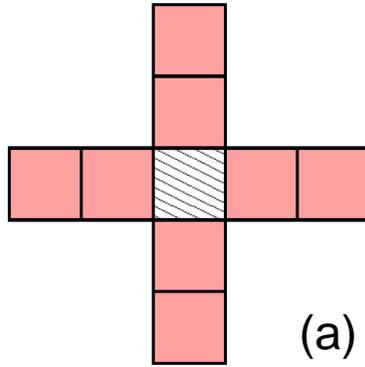
(b)



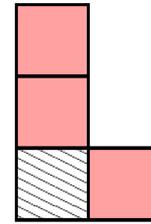
(c)

- a) Long-range $r = 2$: 5 layers ($2r + 1$)
- b) Asymmetric: 3 layers

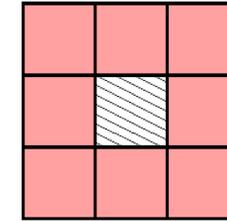
Stencil shapes and layer conditions in 2D



(a)



(b)



(c)

- a) Long-range $r = 2$: 5 layers ($2r + 1$)
- b) Asymmetric: 3 layers
- c) 2D box: 3 layers

Case study: A Jacobi smoother

OpenMP parallelization



OpenMP parallelization of the blocked 2D stencil

Straightforward OpenMP work sharing:

OpenMP parallelization of the blocked 2D stencil

Straightforward OpenMP work sharing:

```
do jb=1, jmax, jblock
!$OMP PARALLEL DO SCHEDULE(static)
  do k=1, kmax
    do j= jb, min(jb+jblock-1, jmax)
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
        + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
!$OMP END PARALLEL DO
enddo
```

OpenMP parallelization of the blocked 2D stencil

Straightforward OpenMP work sharing:

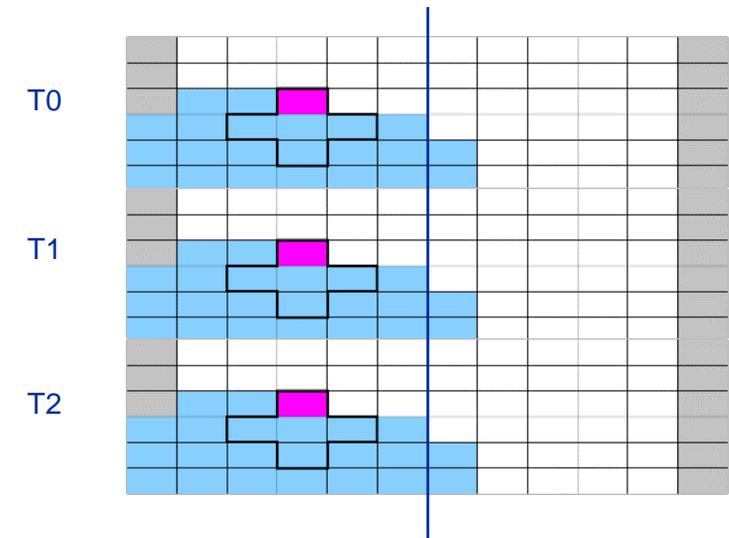
```
do jb=1, jmax, jblock
!$OMP PARALLEL DO SCHEDULE(static)
  do k=1, kmax
    do j= jb, min(jb+jblock-1, jmax)
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                       + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
!$OMP END PARALLEL DO
enddo
```

- Caveat: LC must be fulfilled per thread → shared cache causes smaller blocks!

OpenMP parallelization of the blocked 2D stencil

Straightforward OpenMP work sharing:

```
do jb=1, jmax, jbblock
!$OMP PARALLEL DO SCHEDULE(static)
  do k=1, kmax
    do j= jb, min(jb+jbblock-1, jmax)
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
        + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
!$OMP END PARALLEL DO
enddo
```



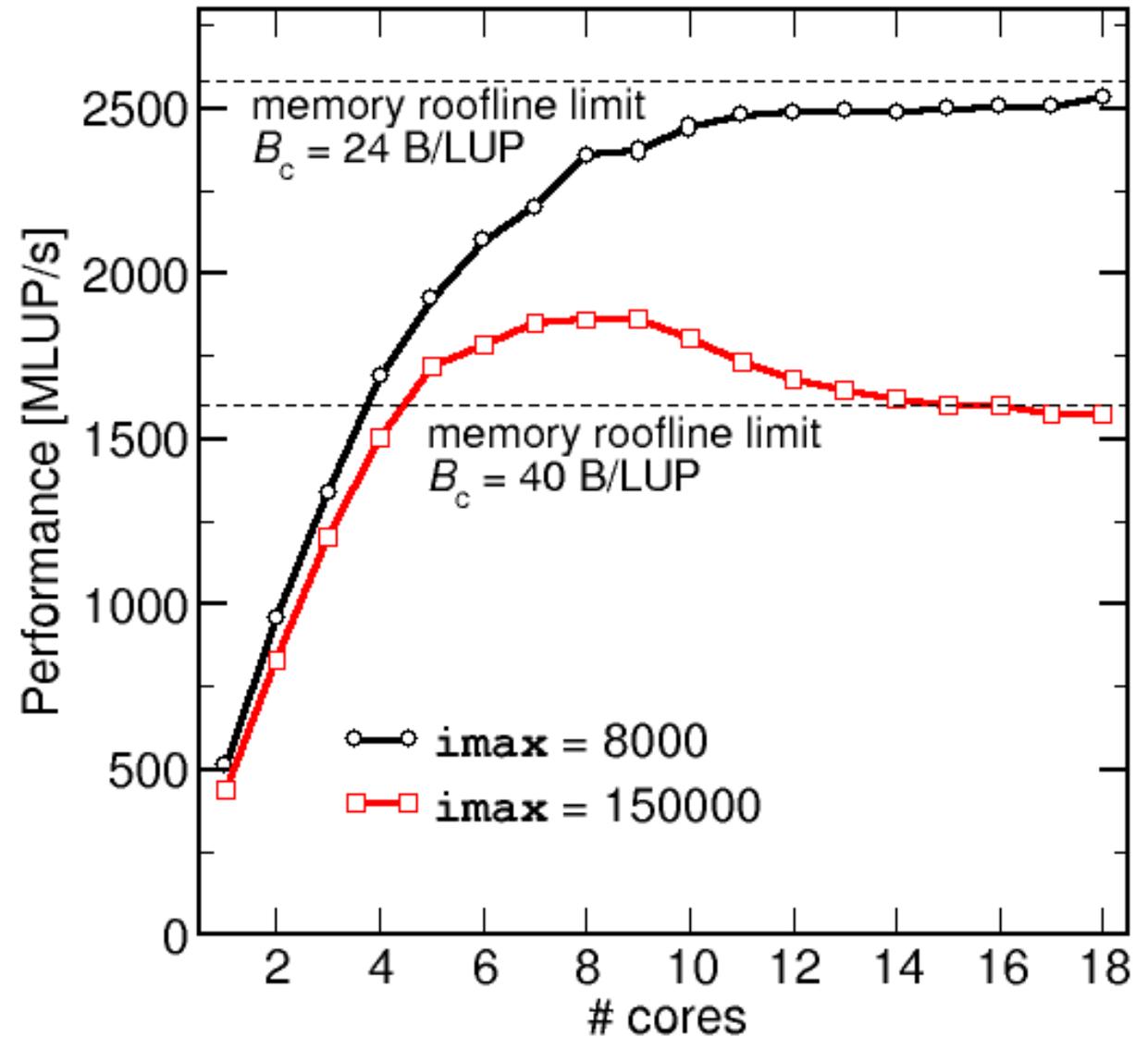
- Caveat: LC must be fulfilled per thread → shared cache causes smaller blocks!

Layer condition:
 $3 * \text{jbblock} * 8B < CSt/2$

Cache size available
per thread

OpenMP parallelization and blocking for a shared cache

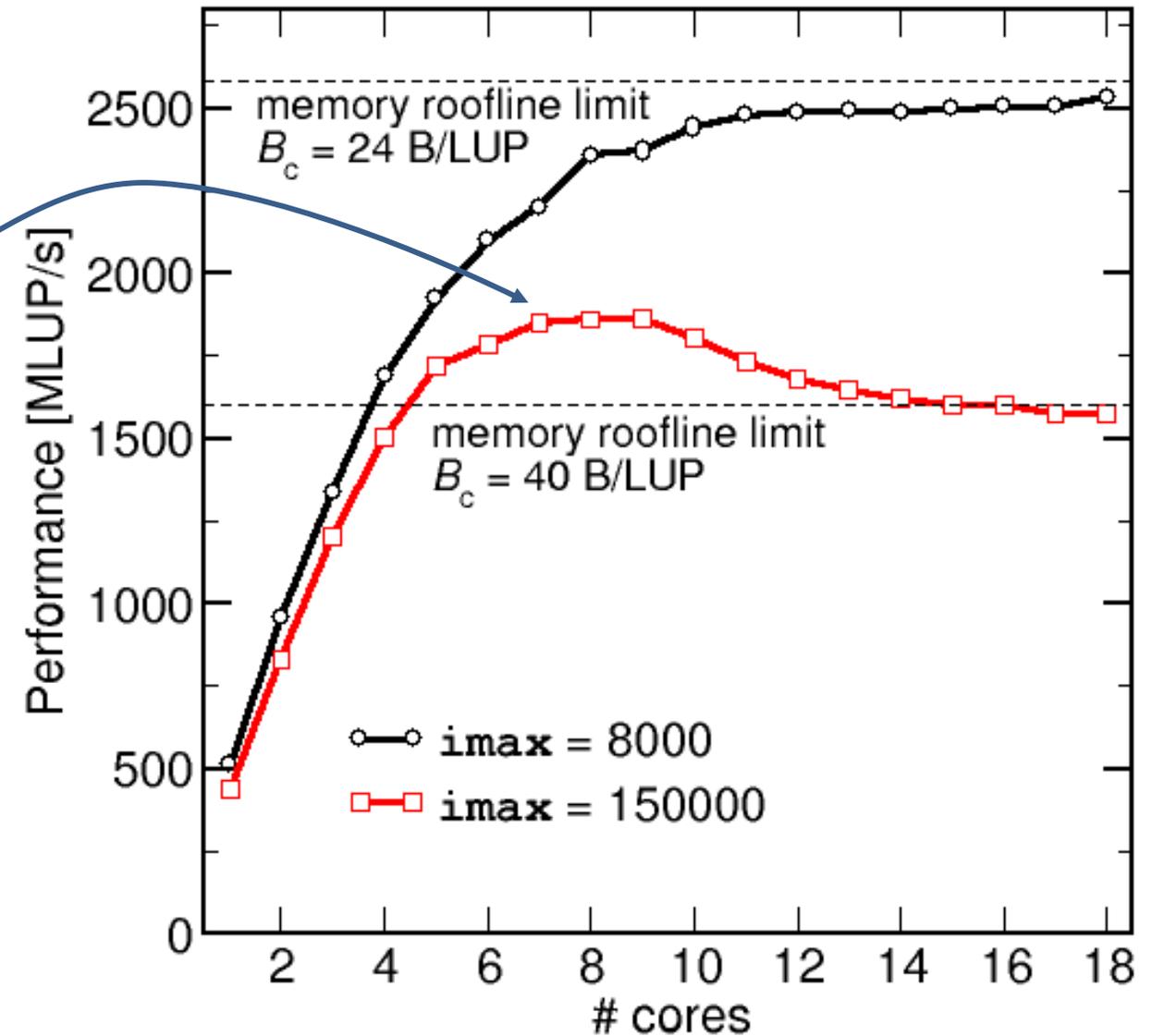
Layer conditions make for interesting effects



OpenMP parallelization and blocking for a shared cache

Layer conditions make for interesting effects

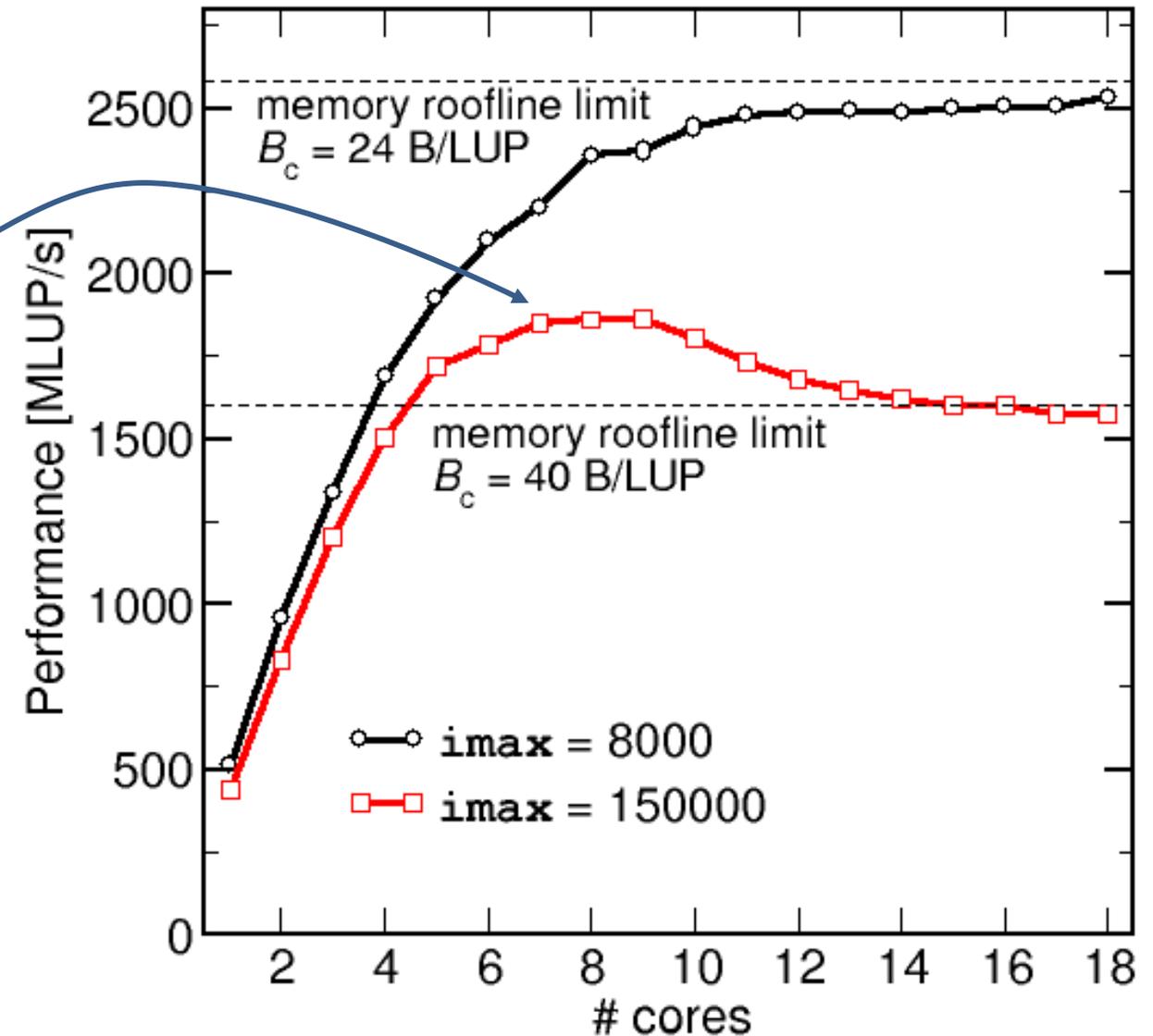
- Less and less shared cache available per thread as #threads goes up
- LC may break “along the way”



OpenMP parallelization and blocking for a shared cache

Layer conditions make for interesting effects

- Less and less shared cache available per thread as #threads goes up
- LC may break “along the way”
- Solutions
 1. Choose small enough block or domain size
 2. Adaptive blocking
 $\text{jblock} = \text{CS} / (\text{\#threads} * 48\text{B})$



Conclusions from the stencil example

- We have **made sense** of the memory-bound **performance** vs. problem size
 - “**Layer conditions**” lead to **predictions of code balance**
 - “**What part of the data comes from where**” is a crucial question
 - The model works only if the **bandwidth is “saturated”**
 - In-cache modeling is more involved

Conclusions from the stencil example

- We have **made sense** of the memory-bound **performance** vs. problem size
 - “**Layer conditions**” lead to **predictions of code balance**
 - “**What part of the data comes from where**” is a crucial question
 - The model works only if the **bandwidth is “saturated”**
 - In-cache modeling is more involved
- **Avoiding slow data paths** == re-establishing the most favorable layer condition
 - Improved code showed the **predicted speedup**
 - Optimal **blocking factor** can be estimated

Conclusions from the stencil example

- We have **made sense** of the memory-bound **performance** vs. problem size
 - “**Layer conditions**” lead to **predictions of code balance**
 - “**What part of the data comes from where**” is a crucial question
 - The model works only if the **bandwidth is “saturated”**
 - In-cache modeling is more involved
- **Avoiding slow data paths** == re-establishing the most favorable layer condition
 - Improved code showed the **predicted speedup**
 - Optimal **blocking factor can be estimated**
- **Food for thought**
 - Higher dimensions (beyond 2D)?
 - Multi-dimensional loop blocking – would it make sense?
 - Can we choose a “better” OpenMP loop schedule?
 - What about temporal blocking?

Stencil references

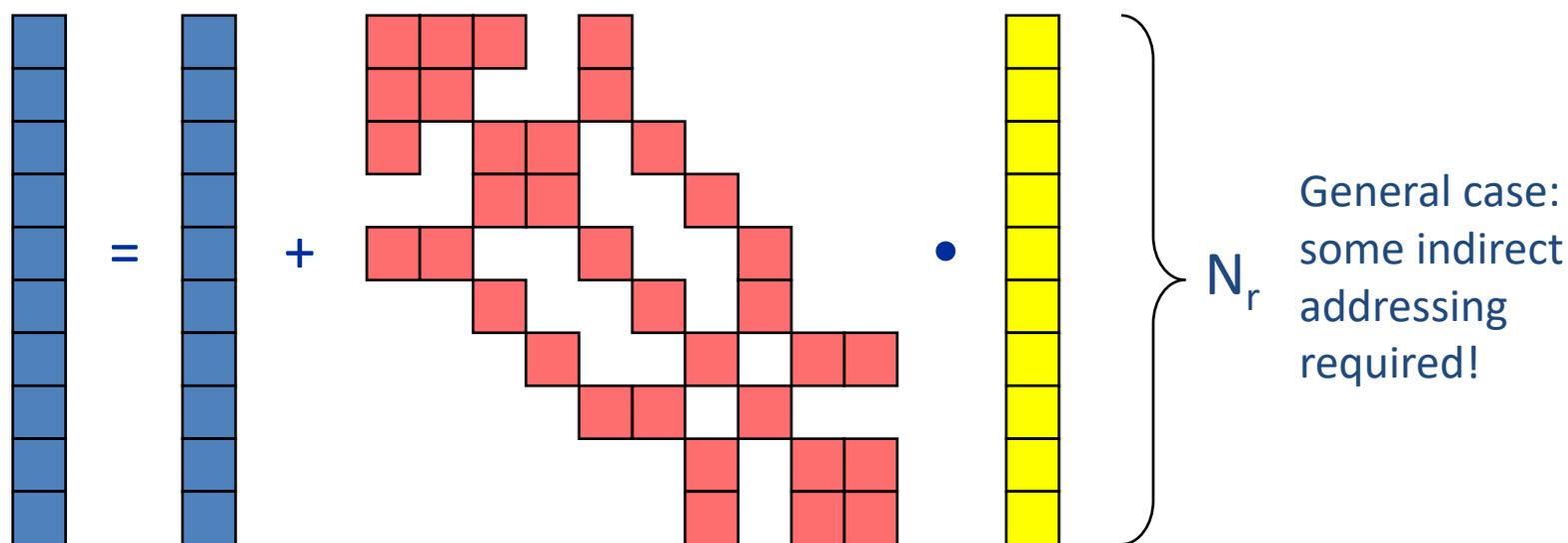
- J. Hammer, G. Hager, J. Eitzinger, and G. Wellein: [Automatic Loop Kernel Analysis and Performance Modeling With Kerncraft](#). Proc. [PMBS15](#), the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, in conjunction with ACM/IEEE Supercomputing 2015 ([SC15](#)), November 16, 2015, Austin, TX. [DOI: 10.1145/2832087.2832092](#), Preprint: [arXiv:1509.03778](#)
- H. Stengel, J. Treibig, G. Hager, and G. Wellein: [Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model](#). Proc. [ICS15](#), [DOI: 10.1145/2751205.2751240](#), Preprint: [arXiv:1410.5010](#)
- M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein: [Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations](#). Concurrency and Computation: Practice and Experience (2015). [DOI:10.1002/cpe.3489](#)
Preprint: [arXiv:1304.7664](#)
- J. Treibig, G. Wellein and G. Hager: [Efficient multicore-aware parallelization strategies for iterative stencil computations](#). Journal of Computational Science 2 (2), 130-137 (2011). [DOI 10.1016/j.jocs.2011.01.010](#)
- M. Wittmann, G. Hager, J. Treibig and G. Wellein: [Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters](#). Parallel Processing Letters 20 (4), 359-376 (2010).
- G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: [Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization](#). Proc. COMPSAC 2009. [DOI: 10.1109/COMPSAC.2009.82](#)

Case study: Sparse Matrix-Vector Multiplication



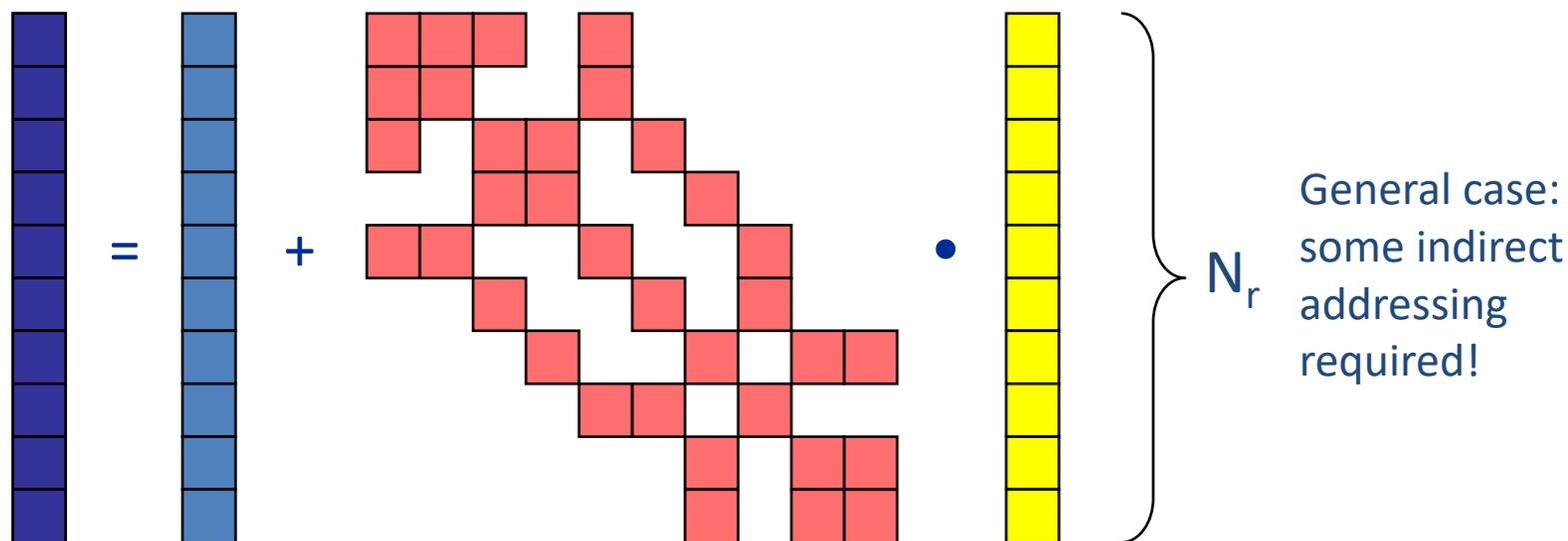
Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in some matrix diagonalization algorithms
 - Lanczos, Davidson, Jacobi-Davidson
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzs} = N_{nz}/N_r$



Sparse Matrix Vector Multiplication (SpMV)

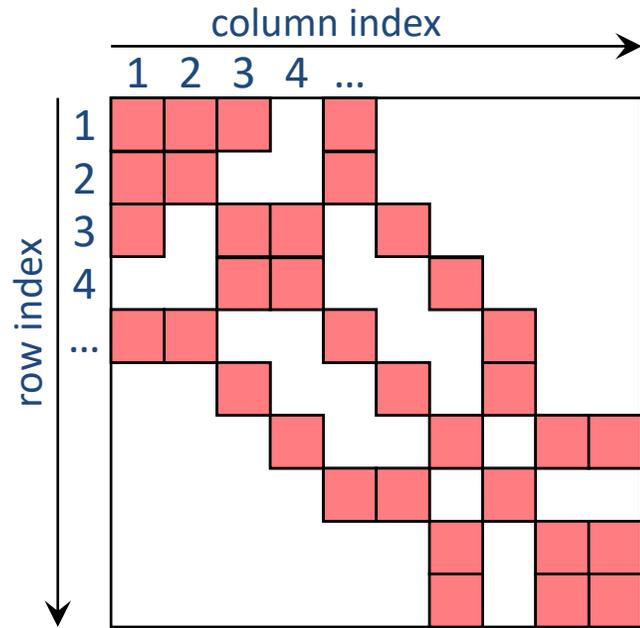
- Key ingredient in some matrix diagonalization algorithms
 - Lanczos, Davidson, Jacobi-Davidson
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzs} = N_{nz}/N_r$



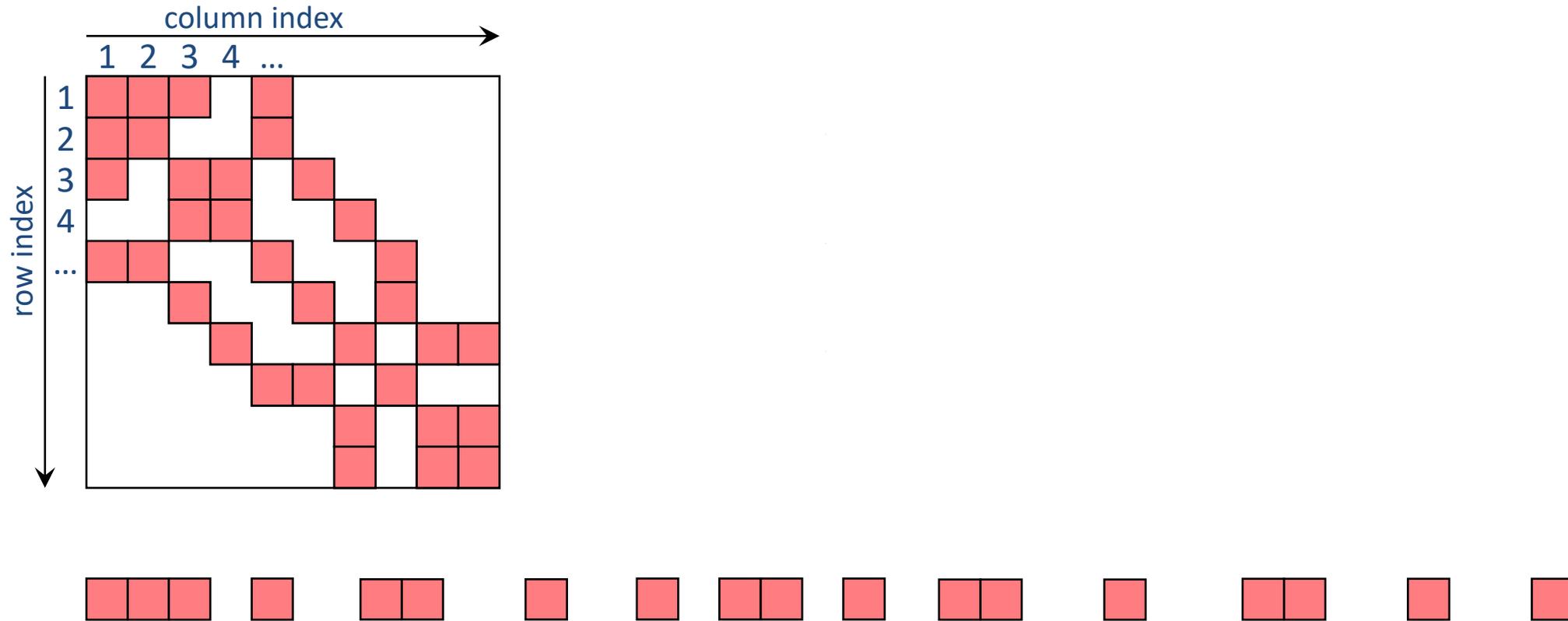
SpMVM characteristics

- For large problems, SpMV is inevitably **memory-bound**
 - **Intra-socket saturation effect** on modern multicores
- SpMV is **easily parallelizable** in shared and distributed memory
 - Load balancing
 - Communication overhead
- Data storage format is **crucial** for performance properties
 - Most useful general format on CPUs:
Compressed Row Storage (**CRS**)
 - Depending on compute architecture

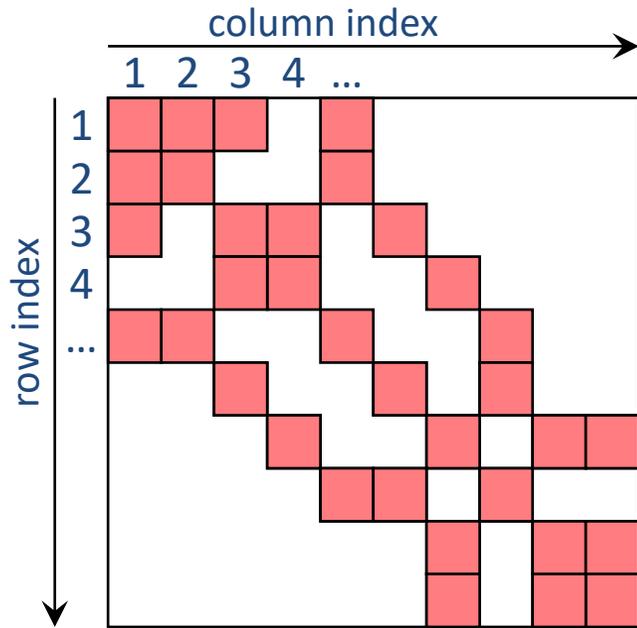
CRS matrix storage scheme



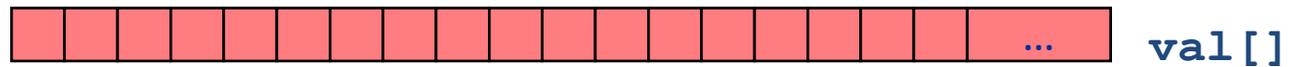
CRS matrix storage scheme



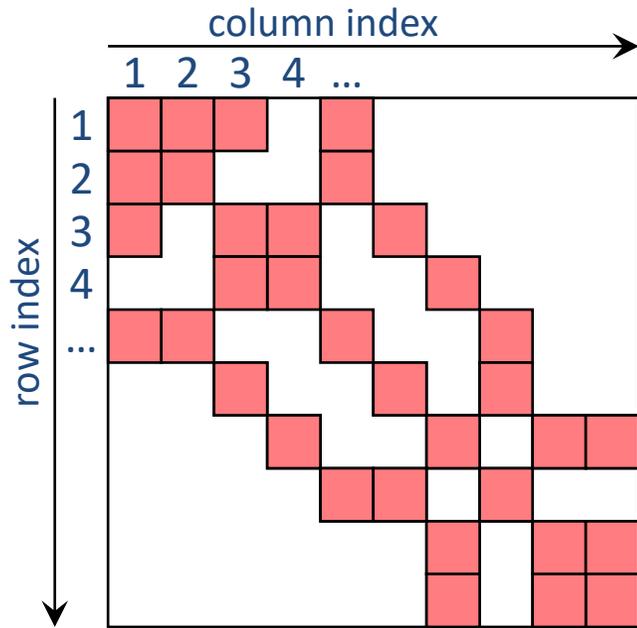
CRS matrix storage scheme



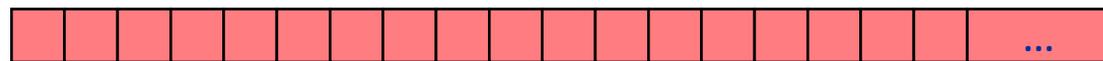
- `val []` stores all the nonzeros (length N_{nz})



CRS matrix storage scheme



- **val []** stores all the nonzeros (length N_{nz})
- **col_idx []** stores the column index of each nonzero (length N_{nz})

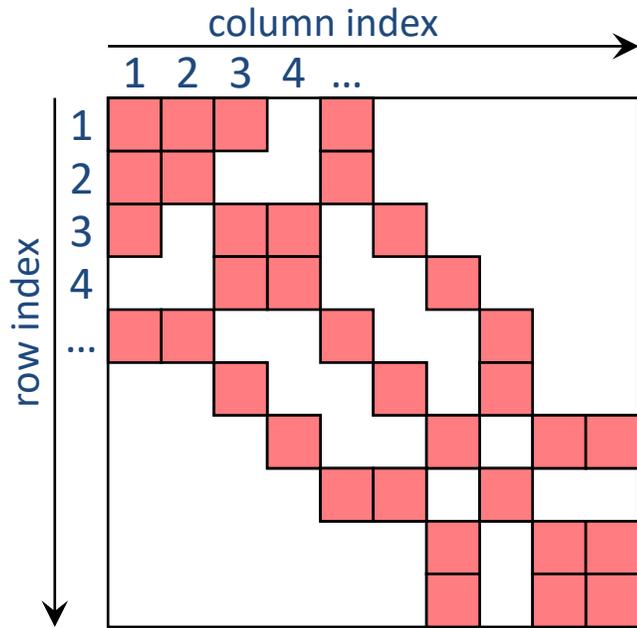


val []

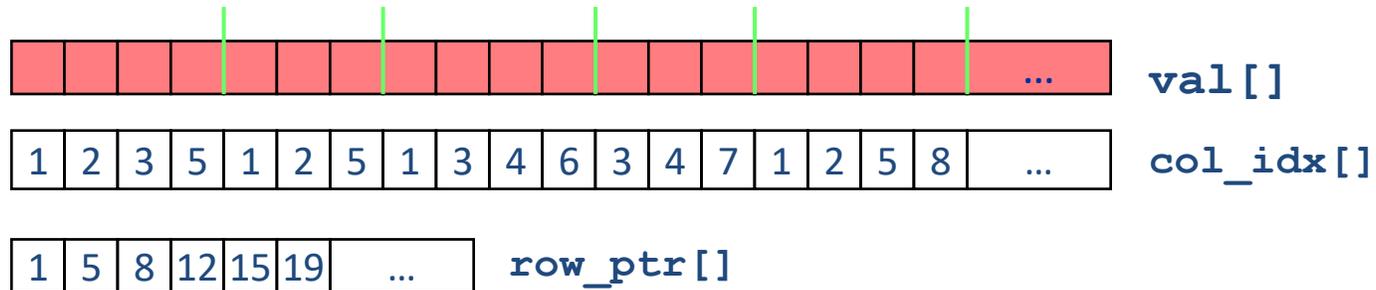


col_idx []

CRS matrix storage scheme



- **val []** stores all the nonzeros (length N_{nz})
- **col_idx []** stores the column index of each nonzero (length N_{nz})
- **row_ptr []** stores the starting index of each new row in **val []** (length: N_r)



Case study: Sparse matrix-vector multiply

- Strongly memory-bound for large data sets
 - Streaming, with partially indirect access:

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

- Usually many spMVMs required to solve a problem
- Now let's look at some performance measurements...

Case study: Sparse matrix-vector multiply

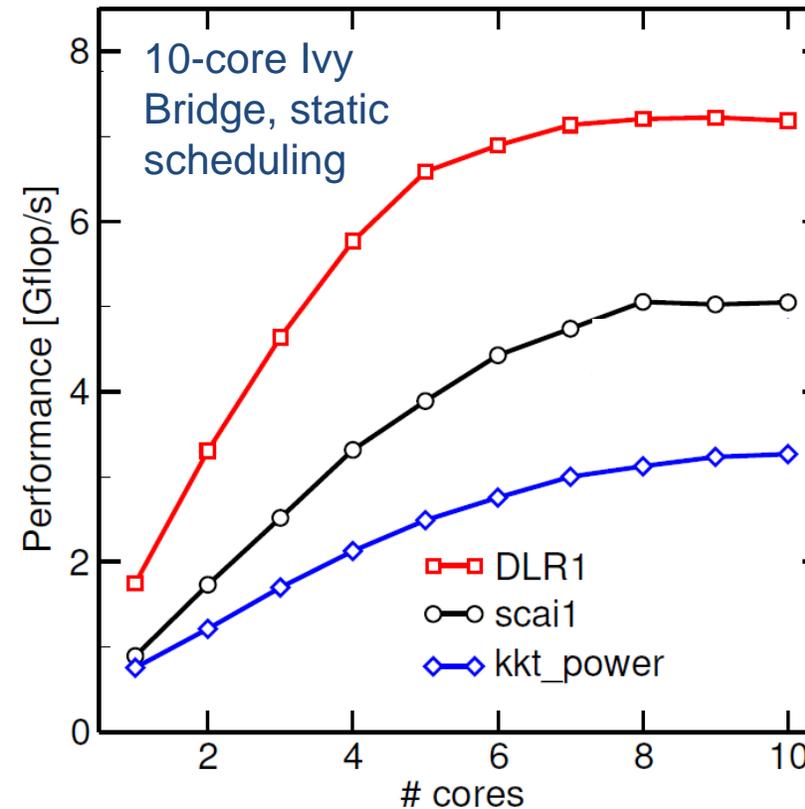
- Strongly memory-bound for large data sets
 - Streaming, with partially indirect access:

```
!$OMP parallel do schedule(???)
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
!$OMP end parallel do
```

- Usually many spMVMs required to solve a problem
- Now let's look at some performance measurements...

Performance characteristics

- Strongly memory-bound for large data sets → saturating performance across cores on the chip
- Performance seems to depend on the matrix
- Can we explain this?
- Is there a “light speed” for SpMV?
- Optimization?



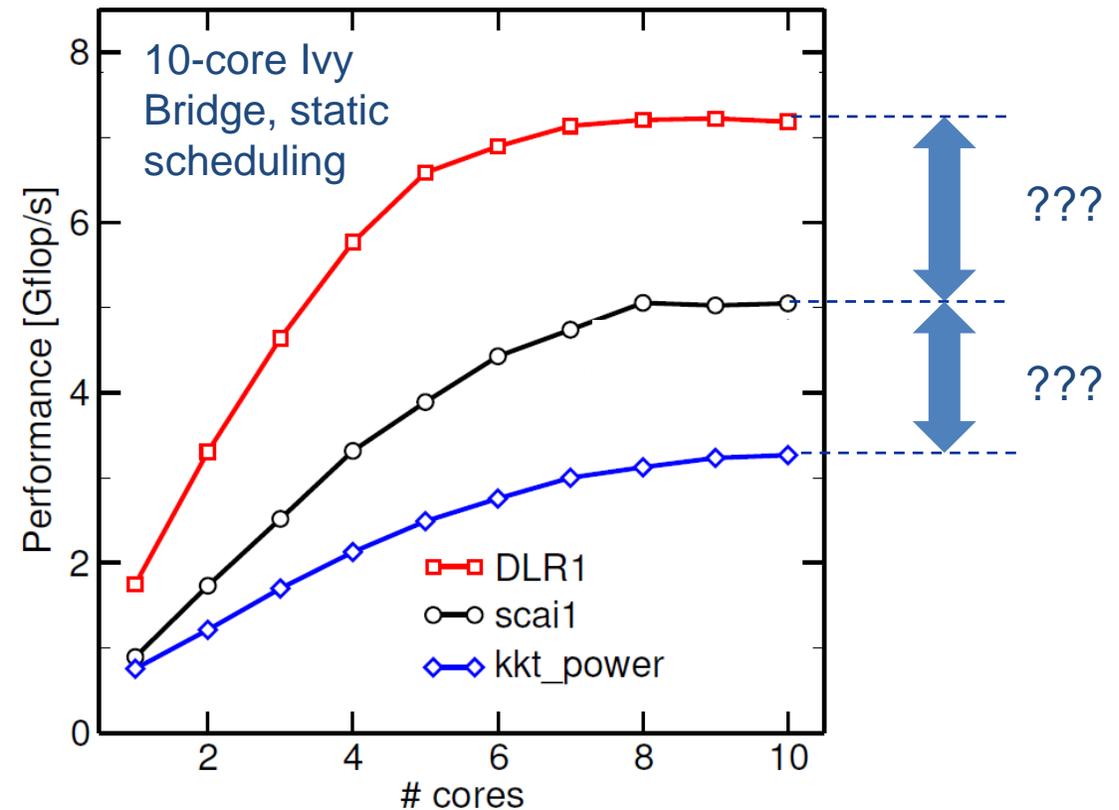
Performance characteristics

- Strongly memory-bound for large data sets → saturating performance across cores on the chip
- Performance seems to depend on the matrix

- Can we explain this?

- Is there a “light speed” for SpMV?

- Optimization?



Minimum amount of data traffic?

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

```
real*8    val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8    C(Nr)
real*8    B(Nc)
```

Min. load traffic [B]: $(8 + 4) N_{nz} + (4 + 8) N_r + 8 N_c$

Min. store traffic [B]: $8 N_r$

Total FLOP count [F]: $2 N_{nz}$

Minimum amount of data traffic?

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

```
real*8    val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8    C(Nr)
real*8    B(Nc)
```

Min. load traffic [B]: $(8 + 4) N_{nz} + (4 + 8) N_r + 8 N_c$

Min. store traffic [B]: $8 N_r$

Total FLOP count [F]: $2 N_{nz}$

$$B_{C,min} = \frac{12 N_{nz} + 20 N_r + 8 N_c}{2 N_{nz}} \frac{B}{F} =$$

Minimum amount of data traffic?

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

```
real*8    val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8    C(Nr)
real*8    B(Nc)
```

Min. load traffic [B]: $(8 + 4) N_{nz} + (4 + 8) N_r + 8 N_c$

Min. store traffic [B]: $8 N_r$

Total FLOP count [F]: $2 N_{nz}$

$$B_{C,min} = \frac{12 N_{nz} + 20 N_r + 8 N_c}{2 N_{nz}} \frac{B}{F} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

Nonzeros per row ($N_{nzc} = N_{nz}/N_r$) or column ($N_{nzc} = N_{nz}/N_c$)

Minimum amount of data traffic?

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

```
real*8    val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8    C(Nr)
real*8    B(Nc)
```

Min. load traffic [B]: $(8 + 4) N_{nz} + (4 + 8) N_r + 8 N_c$

Min. store traffic [B]: $8 N_r$

Total FLOP count [F]: $2 N_{nz}$

$$B_{C,min} = \frac{12 N_{nz} + 20 N_r + 8 N_c}{2 N_{nz}} \frac{B}{F} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

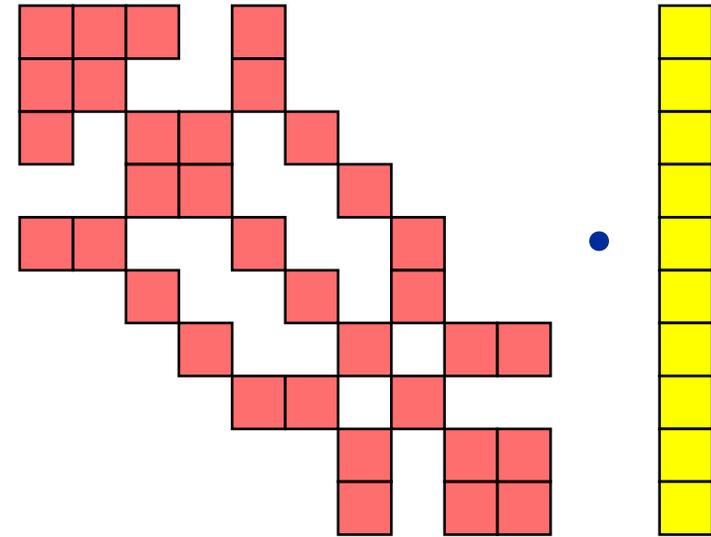
Nonzeros per row ($N_{nzc} = N_{nz}/N_r$) or column ($N_{nzc} = N_{nz}/N_c$)

$$\text{Lower bound for code balance: } B_{C,min} \geq 6 \frac{B}{F} \quad \rightarrow \quad I_{\max} \leq \frac{1}{6} \frac{F}{B}$$

SpMV node performance model – CRS (2)

```
do i = 1, Nr  
  do j = row_ptr(i), row_ptr(i+1) - 1  
    C(i) = C(i) + val(j) * B(col_idx(j))  
  enddo  
enddo
```

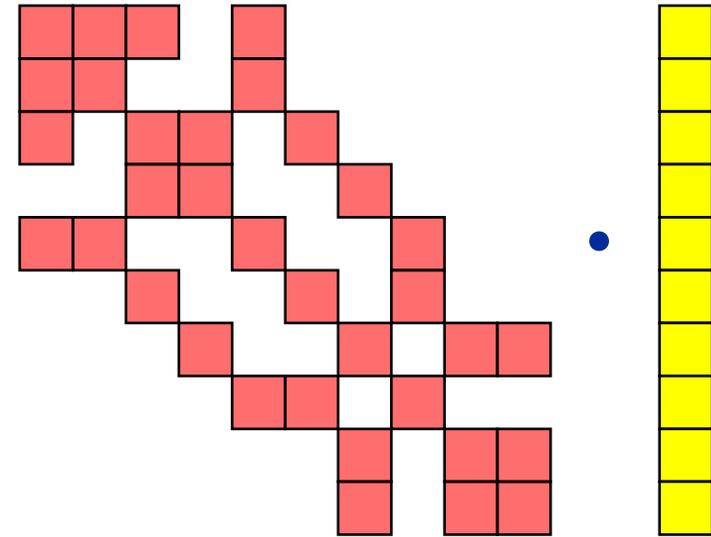
$$B_{C,min} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$



SpMV node performance model – CRS (2)

```
do i = 1, Nr  
  do j = row_ptr(i), row_ptr(i+1) - 1  
    C(i) = C(i) + val(j) * B(col_idx(j))  
  enddo  
enddo
```

$$B_{C,min} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

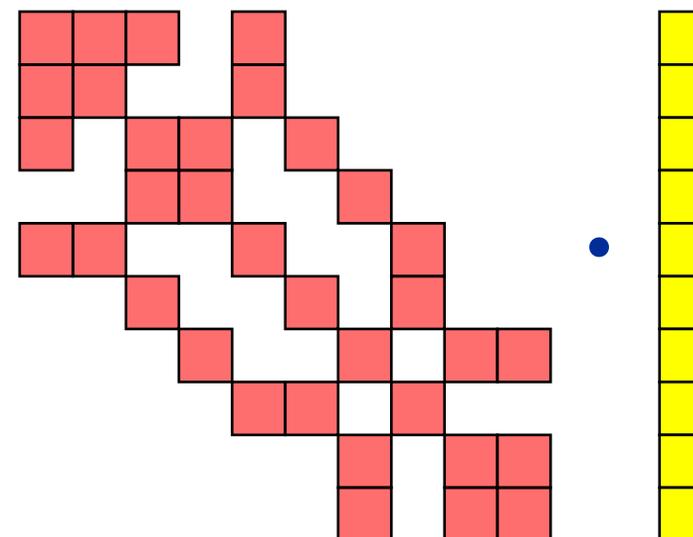


SpMV node performance model – CRS (2)

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

$$B_{C,min} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

$$B_C(\alpha) = \frac{12 + 20/N_{nzc} + 8\alpha}{2} \frac{B}{F}$$



Parameter (α) quantifies additional traffic for $\mathbf{B}(\cdot)$ (irregular access):

$$\alpha \geq 1/N_{nzc}$$

$$\alpha N_{nzc} \geq 1$$

SpMV node performance model – CRS (2)

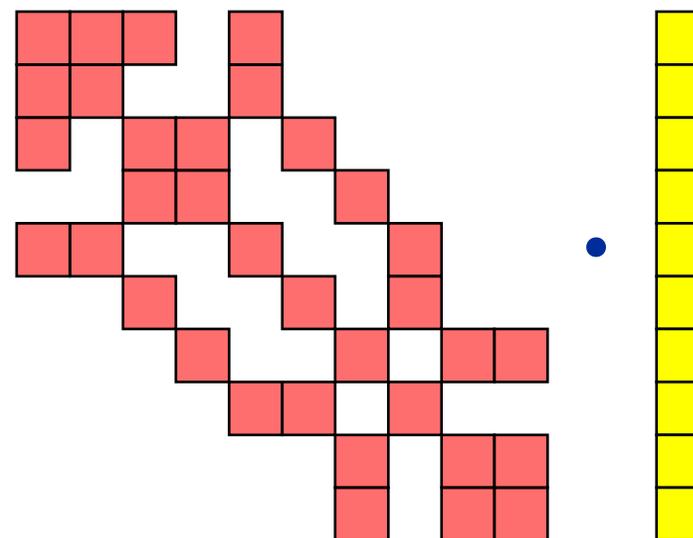
```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

$$B_{C,min} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

$$B_C(\alpha) = \frac{12 + 20/N_{nzc} + 8\alpha}{2} \frac{B}{F}$$

Consider square matrices: $N_{nzc} = N_{nzc}$ and $N_c = N_r$

Note: $B_C(1/N_{nzc}) = B_{C,min}$



Parameter (α) quantifies additional traffic for $B(\cdot)$ (irregular access):

$$\alpha \geq 1/N_{nzc}$$

$$\alpha N_{nzc} \geq 1$$

The “ α effect”

DP CRS code balance

- α quantifies the traffic for loading the RHS
 - $\alpha = 0 \rightarrow$ RHS is in cache
 - $\alpha = 1/N_{nzs} \rightarrow$ RHS loaded once
 - $\alpha = 1 \rightarrow$ no cache
 - $\alpha > 1 \rightarrow$ Houston, we have a problem!
- “Target” performance = b_S/B_C
- **Caveat:** Maximum memory BW may not be achieved with spMVM (see later)

$$B_C(\alpha) = \frac{12 + 20/N_{nzs} + 8\alpha}{2} \frac{B}{F}$$
$$= \left(6 + 4\alpha + \frac{10}{N_{nzs}} \right) \frac{B}{F}$$

The “ α effect”

DP CRS code balance

- α quantifies the traffic for loading the RHS
 - $\alpha = 0 \rightarrow$ RHS is in cache
 - $\alpha = 1/N_{nzs} \rightarrow$ RHS loaded once
 - $\alpha = 1 \rightarrow$ no cache
 - $\alpha > 1 \rightarrow$ Houston, we have a problem!
- “Target” performance = b_S/B_C
- **Caveat:** Maximum memory BW may not be achieved with spMVM (see later)

$$B_C(\alpha) = \frac{12 + 20/N_{nzs} + 8\alpha}{2} \frac{B}{F}$$
$$= \left(6 + 4\alpha + \frac{10}{N_{nzs}} \right) \frac{B}{F}$$

Can we predict α ?

- Not in general
- Simple cases (banded, block-structured): Similar to layer condition analysis

\rightarrow Determine α by measuring the actual memory traffic (\rightarrow measured code balance B_C^{meas})

Determine α (RHS traffic quantification)

$$B_C(\alpha) = \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F} = \frac{V_{meas}}{N_{nz} \cdot 2 F} (= B_C^{meas})$$

- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α :

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{10}{N_{nzs}} \right)$$

Determine α (RHS traffic quantification)

$$B_C(\alpha) = \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F} = \frac{V_{meas}}{N_{nz} \cdot 2 F} (= B_C^{meas})$$

- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α :

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{10}{N_{nzs}} \right)$$

Example: kkt_power matrix from the UoF collection on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6, N_{nzs} = 7.1$

- $V_{meas} \approx 258 \text{ MB}$

→ $\alpha = 0.36, \alpha N_{nzs} = 2.5$

→ RHS is loaded 2.5 times from memory

Determine α (RHS traffic quantification)

$$B_C(\alpha) = \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F} = \frac{V_{meas}}{N_{nz} \cdot 2 F} (= B_C^{meas})$$

- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α :

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{10}{N_{nzs}} \right)$$

Example: kkt_power matrix from the UoF collection on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6, N_{nzs} = 7.1$

- $V_{meas} \approx 258 \text{ MB}$

→ $\alpha = 0.36, \alpha N_{nzs} = 2.5$

→ RHS is loaded 2.5 times from memory

$$\frac{B_C(\alpha)}{B_{C,min}} = 1.11$$

Determine α (RHS traffic quantification)

$$B_C(\alpha) = \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F} = \frac{V_{meas}}{N_{nz} \cdot 2 F} (= B_C^{meas})$$

- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α :

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{10}{N_{nzs}} \right)$$

Example: kkt_power matrix from the UoF collection on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6, N_{nzs} = 7.1$

- $V_{meas} \approx 258 \text{ MB}$

→ $\alpha = 0.36, \alpha N_{nzs} = 2.5$

→ RHS is loaded 2.5 times from memory

$$\frac{B_C(\alpha)}{B_{C,min}} = 1.11$$

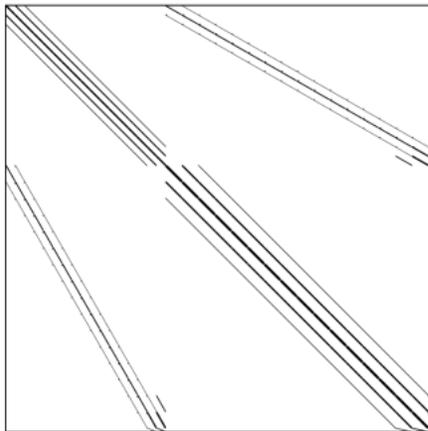
11% extra traffic → optimization potential!

Three different sparse matrices

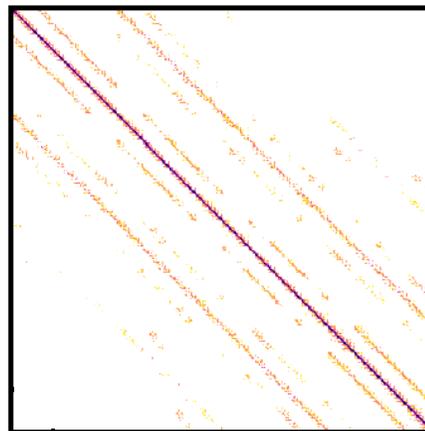
Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_s = 46.6$ GB/s

$$\rightarrow \text{Roofline: } P_{opt} = b_s / B_{C,min}$$

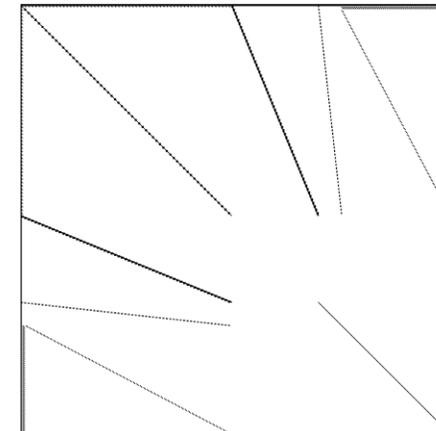
Matrix	N	N_{nzs}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

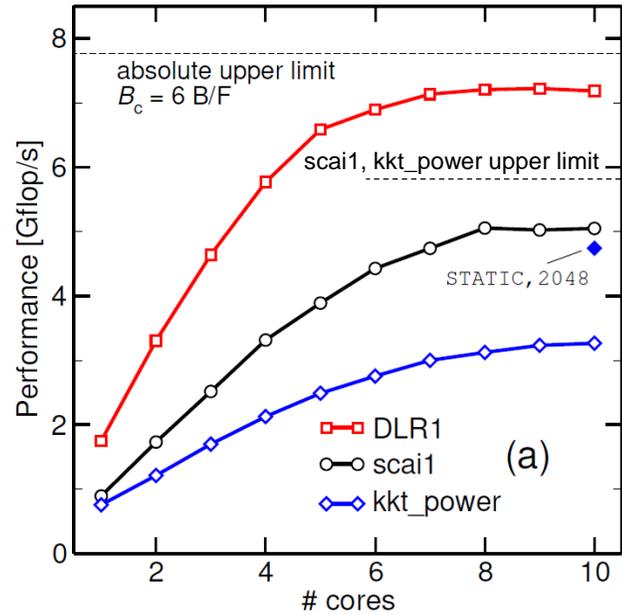


scai1



kkt_power

Now back to the start...

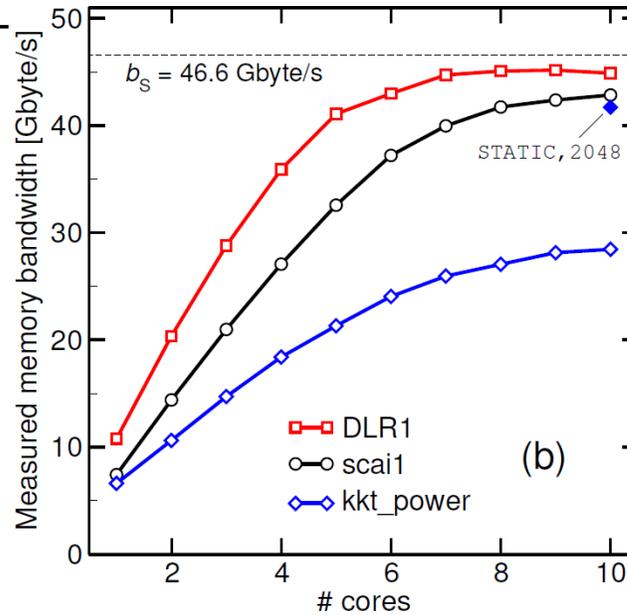
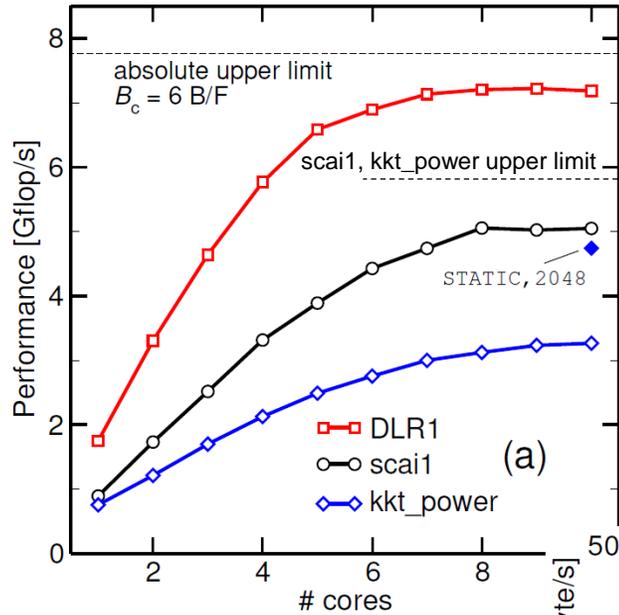


- $b_S = 46.6 \text{ GB/s}$, $B_c = 6 \text{ B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)

Now back to the start...

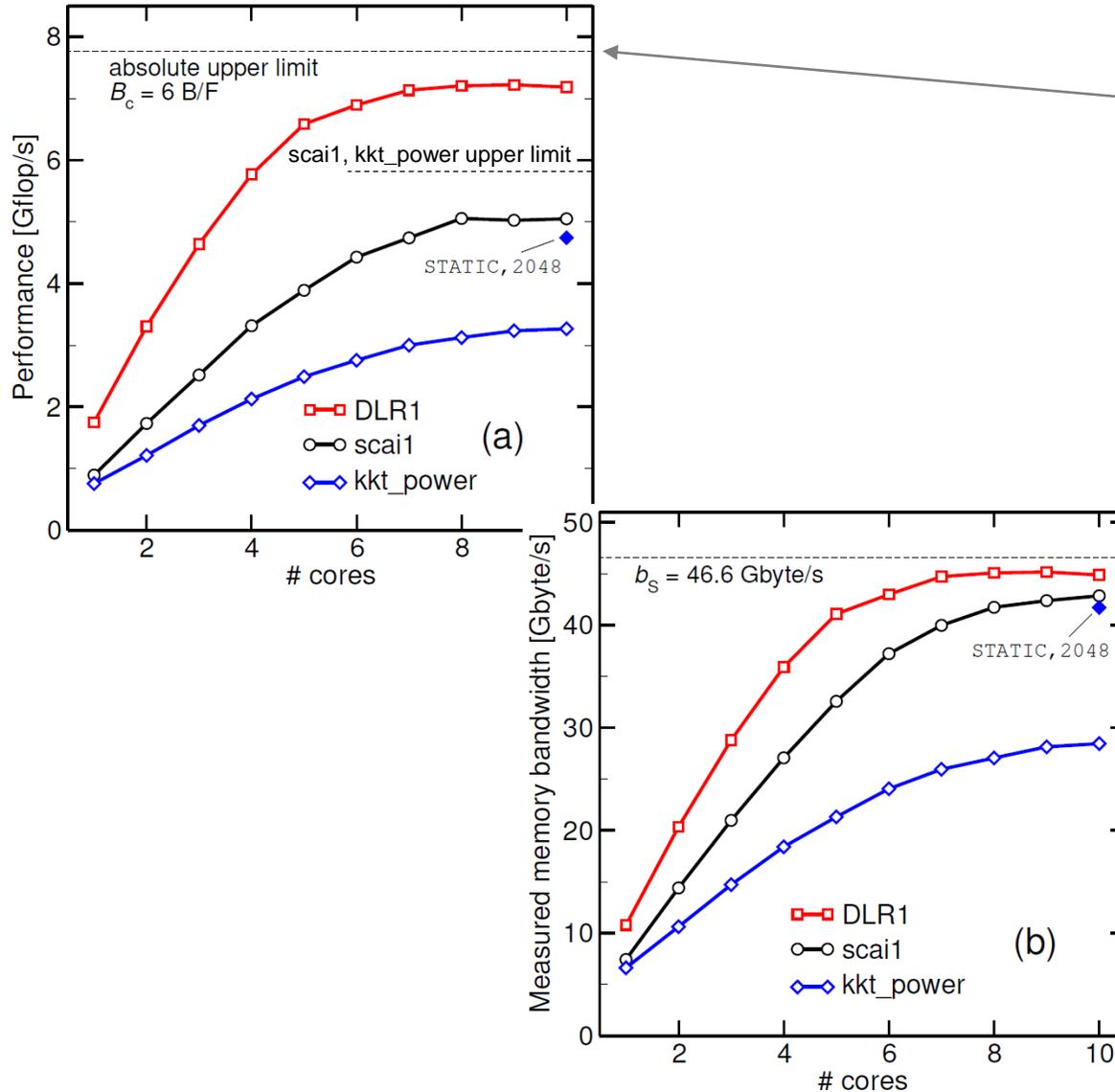


- $b_s = 46.6 \text{ GB/s}$, $B_c = 6 \text{ B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)

Now back to the start...

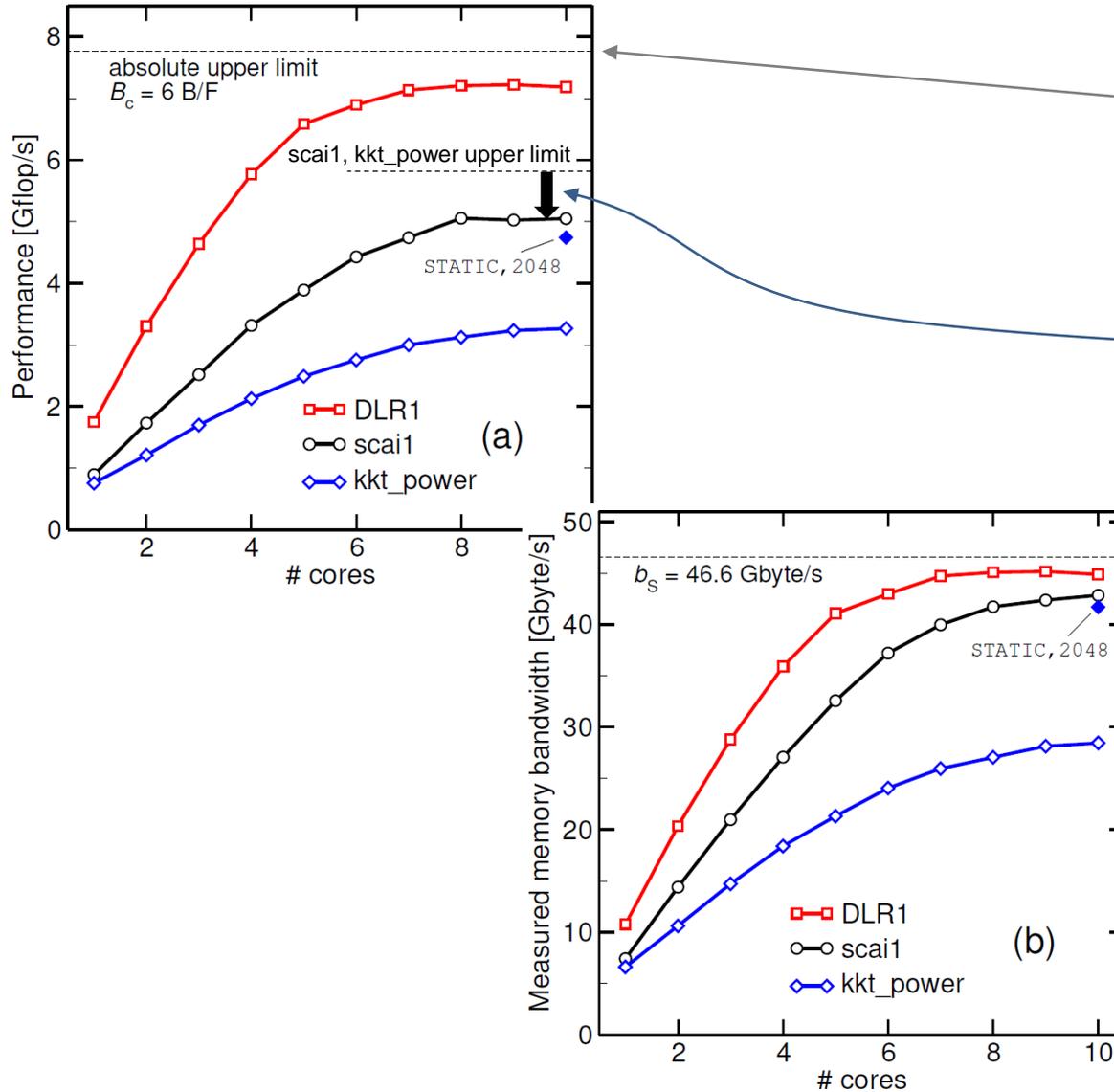


- $b_S = 46.6 \text{ GB/s}$, $B_c = 6 \text{ B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)
- **scai1** measured balance:
 $B_c^{meas} \approx 8.5 \text{ B/F} > B_{C,min}$ (6% higher than min)
 → good BW utilization, slightly non-optimal α

Now back to the start...

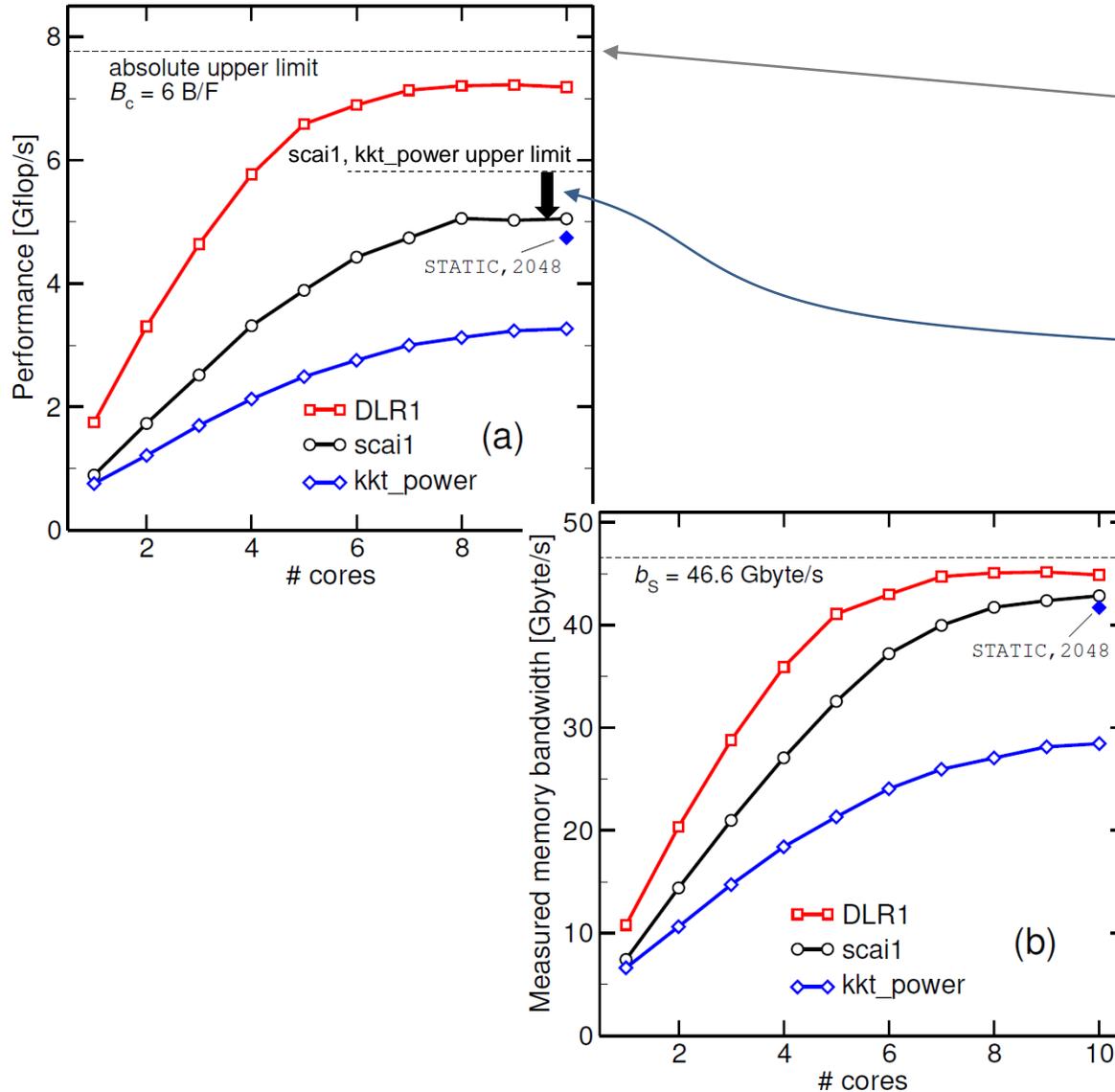


- $b_S = 46.6 \text{ GB/s}$, $B_c = 6 \text{ B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)
- **scai1** measured balance:
 $B_c^{meas} \approx 8.5 \text{ B/F} > B_{C,min}$ (6% higher than min)
 \rightarrow good BW utilization, slightly non-optimal α

Now back to the start...



- $b_S = 46.6 \text{ GB/s}$, $B_c = 6 \text{ B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)

- **scai1** measured balance:

$$B_c^{meas} \approx 8.5 \text{ B/F} > B_{C,min} \text{ (6\% higher than min)}$$

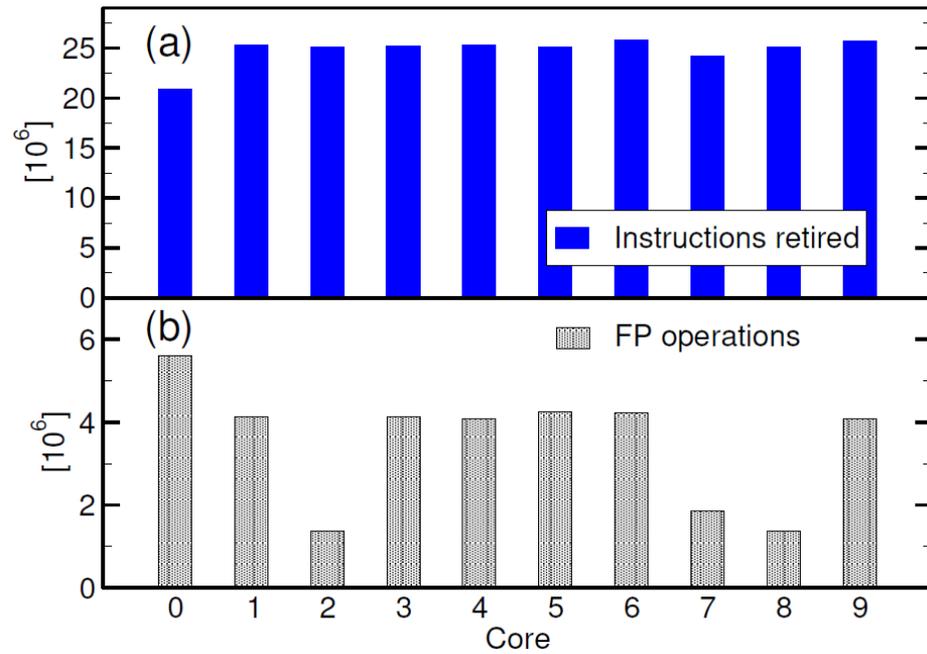
→ good BW utilization, slightly non-optimal α

- **kkt_power** measured balance:

$$B_c^{meas} \approx 8.8 \text{ B/F} > B_{C,min} \text{ (10\% higher than min)}$$

→ performance degraded by load imbalance, fix by block-cyclic schedule

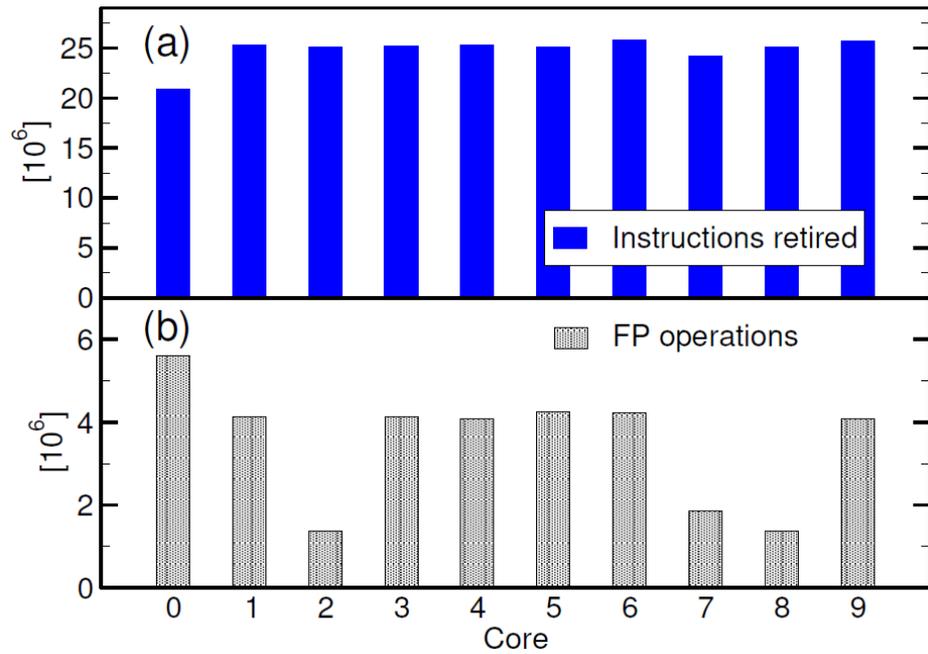
Investigating the load imbalance with kkt_power



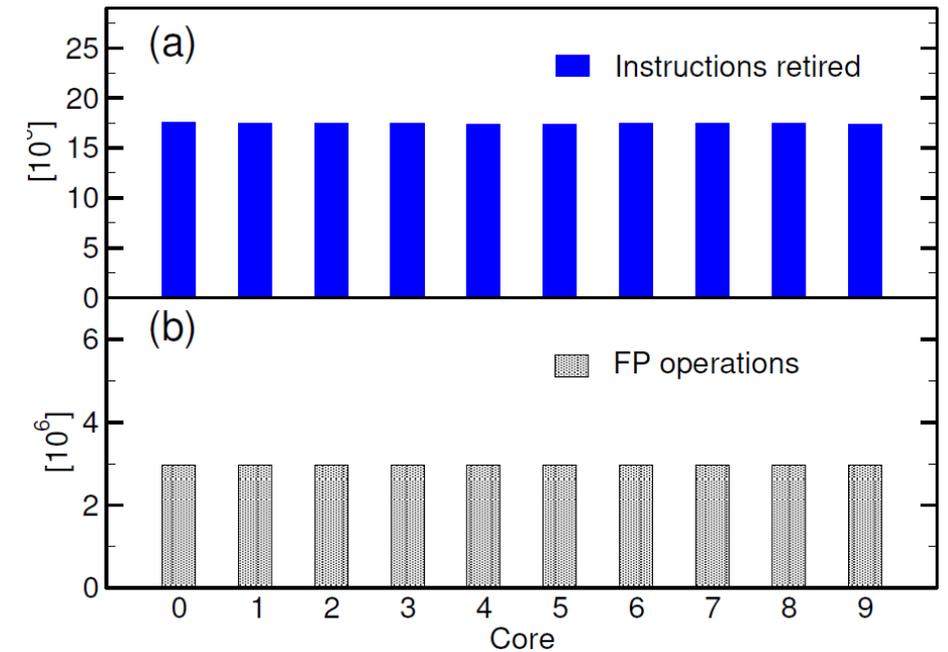
Measurements with likwid-perfctr
(MEM_DP group)



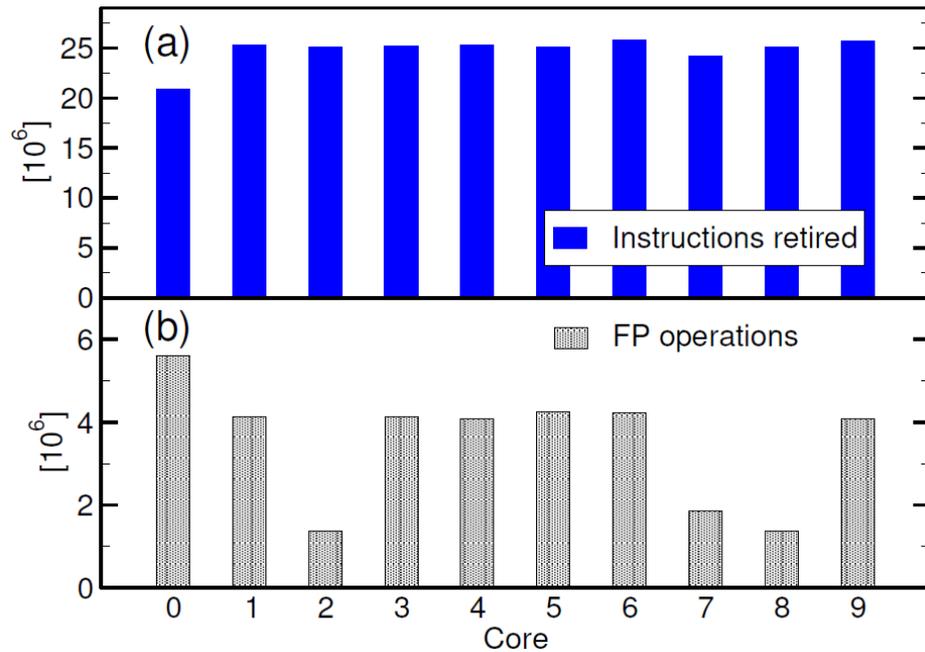
Investigating the load imbalance with kkt_power



Measurements with likwid-perfctr
(MEM_DP group)



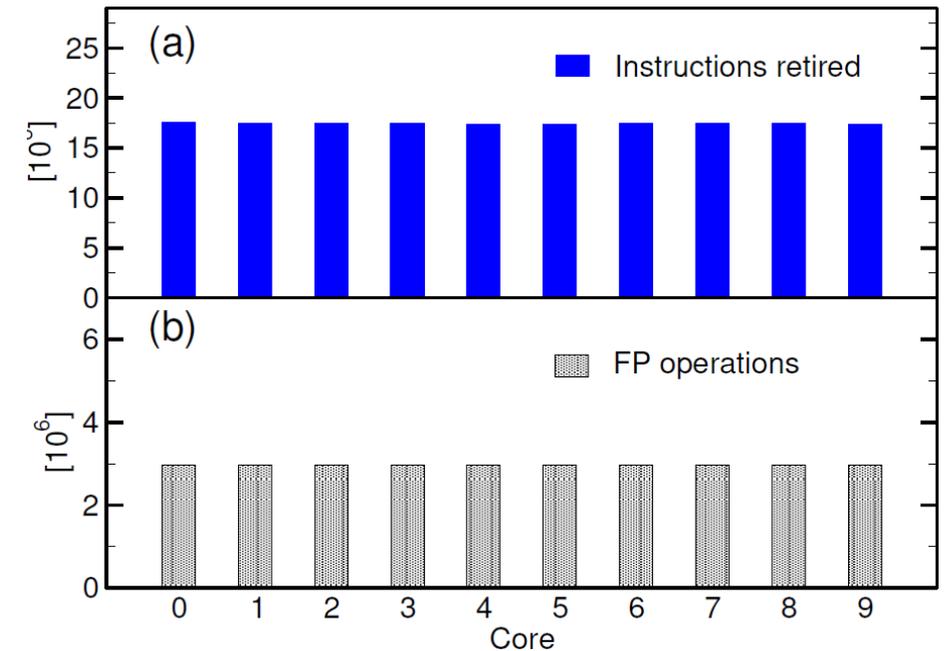
Investigating the load imbalance with kkt_power



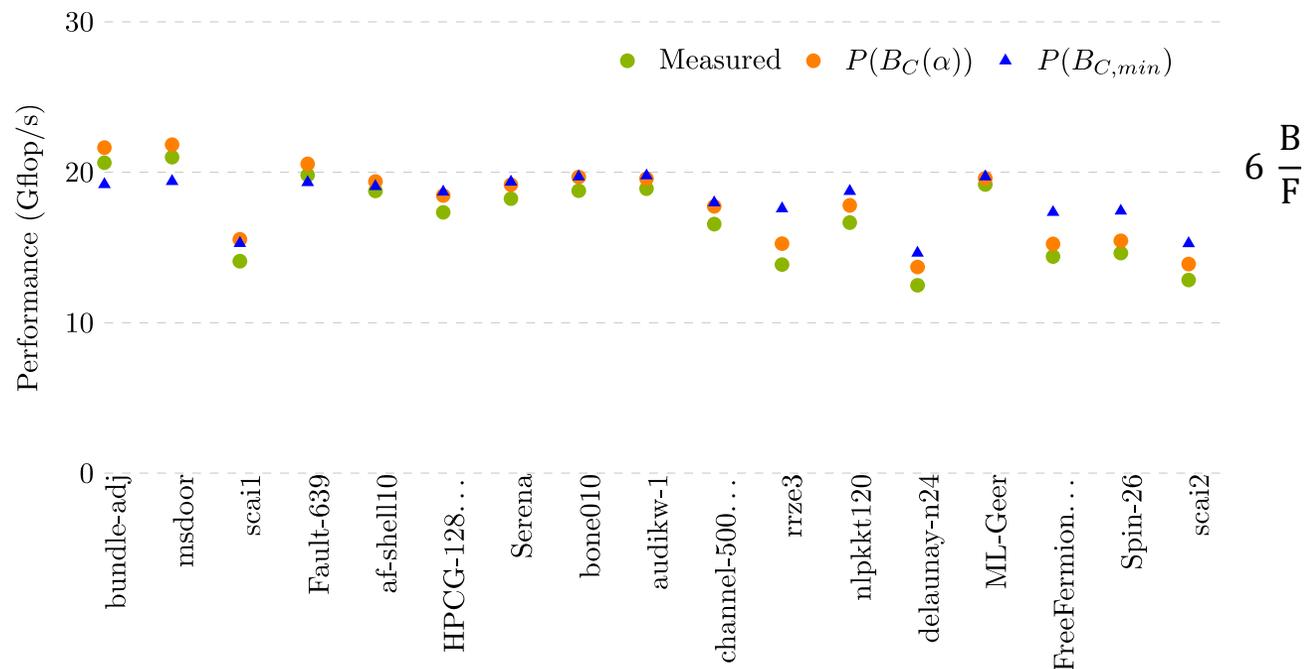
Measurements with likwid-perfctr (MEM_DP group)



- Fewer overall instructions, (almost) BW saturation, 50% better performance with load balancing
- CPI value unchanged!



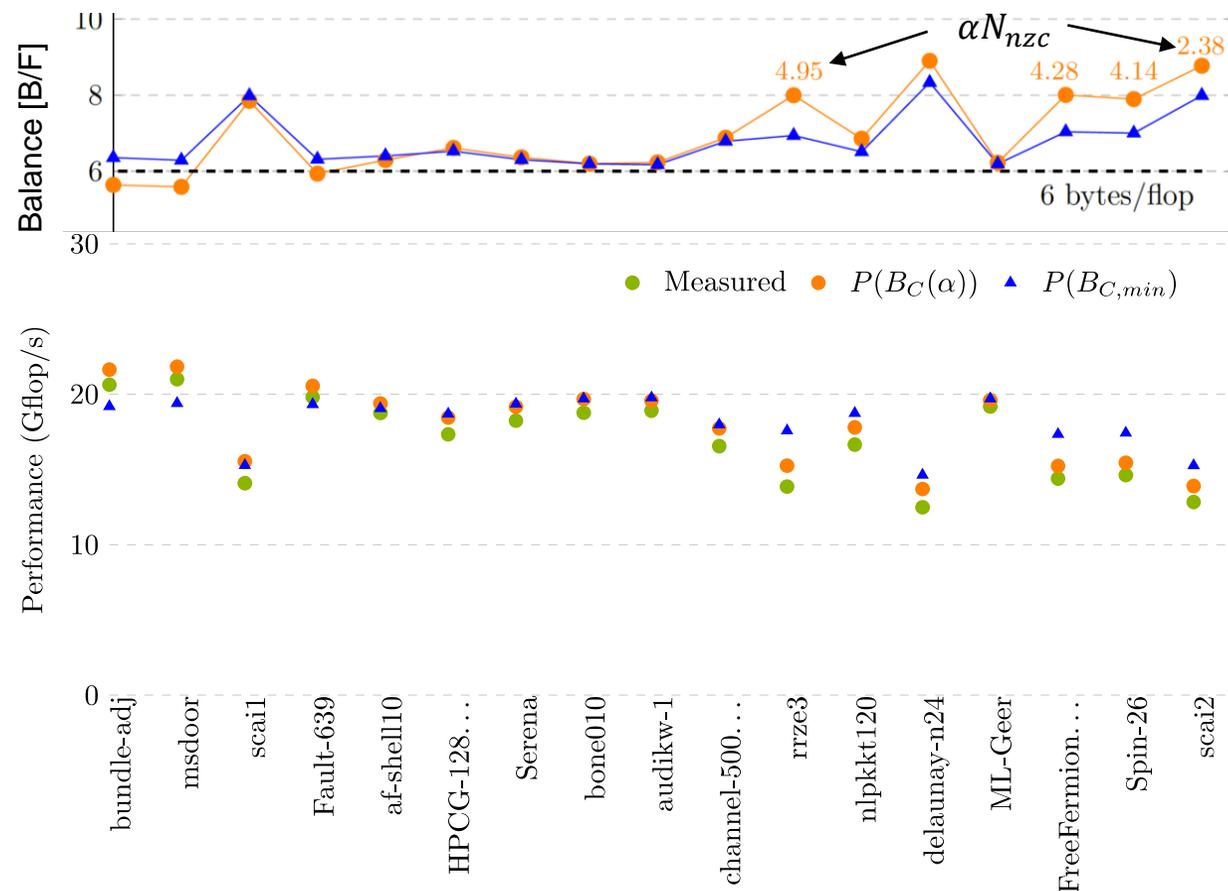
SpMV node performance model – CPU



Intel Xeon Platinum 9242
24c@2.8GHz (turbo)
 $b_S = 122 \text{ GB/s}$

Matrices taken from: C. L. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX*. Concurrency and Computation: Practice and Experience, e6512 (2021). DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

SpMV node performance model – CPU



6 $\frac{B}{F}$

Intel Xeon Platinum 9242
24c@2.8GHz (turbo)
 $b_s = 122 \text{ GB/s}$

Matrices taken from: C. L. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX*. Concurrency and Computation: Practice and Experience, e6512 (2021). DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

Roofline analysis for spMVM

- Conclusion from the Roofline analysis
 - The roofline model does not “work” for spMVM due to the RHS traffic uncertainties
 - We have “turned the model around” and measured the actual memory traffic to determine the RHS overhead
 - Result indicates:
 1. how much actual traffic the RHS generates
 2. how efficient the RHS access is (compare BW with max. BW)
 3. how much optimization potential we have with matrix reordering
- Do not forget about load balancing!
- SpMV is not the end of the story: $A \times \{x^1, x^2, \dots\}, A^p x, \dots$
- Consequence: Modeling is not always 100% predictive. It's all about *learning more about performance properties!*

Some publications

- C. Alappat, J. Thies, G. Hager, H. Fehske, and G. Wellein: *Algebraic Temporal Blocking for Sparse Iterative Solvers on Multi-Core CPUs*. Submitted. Preprint: [arXiv:2309.02228](https://arxiv.org/abs/2309.02228)
- C. L. Alappat, G. Hager, O. Schenk, and G. Wellein: *Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication*. IEEE Transactions on Parallel and Distributed Systems **34**(2), 581-597 (2023), DOI: [10.1109/TPDS.2022.3223512](https://doi.org/10.1109/TPDS.2022.3223512)
- C. L. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX*. Concurrency and Computation: Practice and Experience, e6512 (2021). **Available with Open Access**. DOI: [10.1002/cpe.65](https://doi.org/10.1002/cpe.65)
- C. L. Alappat, G. Hager, O. Schenk, J. Thies, A. Basermann, A. R. Bishop, H. Fehske, and G. Wellein: *A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication*. ACM Trans. Parallel Comput. **7**(3), Article 19 (June 2020), 37 pages. **Available with Open Access**. DOI: [10.1145/3399732](https://doi.org/10.1145/3399732).
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: [10.1137/130930352](https://doi.org/10.1137/130930352)

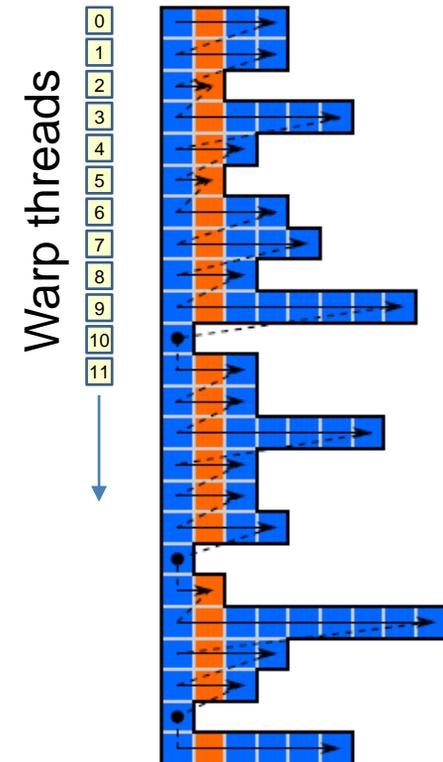
Sparse Matrix-Vector Multiplication on GPGPUs

optional



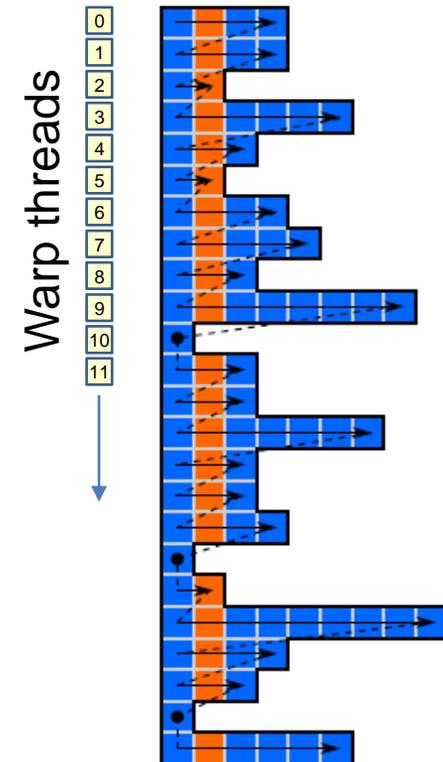
What about GPUs?

- GPUs need
 - Enough work per kernel launch in order to leverage their parallelism
 - Coalesced access to memory (consecutive threads in a warp should access consecutive memory addresses)



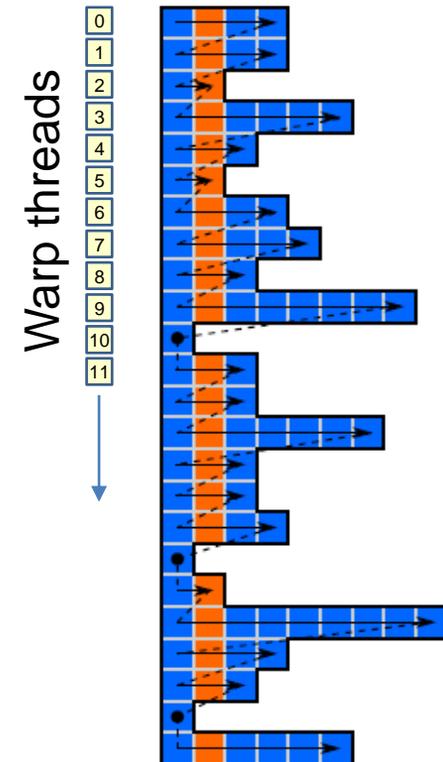
What about GPUs?

- GPUs need
 - **Enough work per kernel** launch in order to leverage their parallelism
 - **Coalesced access to memory** (consecutive threads in a warp should access consecutive memory addresses)
- Plain CRS for SpMV on GPUs is not a good idea
 1. **Short inner loop**
 2. **Different** amount of **work per thread**
 3. **Non-coalesced** memory access



What about GPUs?

- GPUs need
 - Enough work per kernel launch in order to leverage their parallelism
 - Coalesced access to memory (consecutive threads in a warp should access consecutive memory addresses)
- Plain CRS for SpMV on GPUs is not a good idea
 1. Short inner loop
 2. Different amount of work per thread
 3. Non-coalesced memory access
- Remedy: Use SIMD/SIMT-friendly storage format
 - ELLPACK, SELL-C- σ , DIA, ESB,...



CRS SpMV in CUDA ($y = Ax$)

```
template <typename VT, typename IT>
__global__ static void
spmvr_csr(const ST num_rows,
          const IT * RESTRICT row_ptrs, const IT * RESTRICT col_idxs,
          const VT * RESTRICT values,  const VT * RESTRICT x,
                                               VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x; // 1 thread per row

    if (row < num_rows) {
        VT sum{};
        for (IT j = row_ptrs[row]; j < row_ptrs[row + 1]; ++j) {
            sum += values[j] * x[col_idxs[j]];
        }
        y[row] = sum;
    }
}
```

$$B_c(\alpha) = \left(6 + 4\alpha + \frac{6}{N_{nzs}} \right) \frac{B}{F}$$

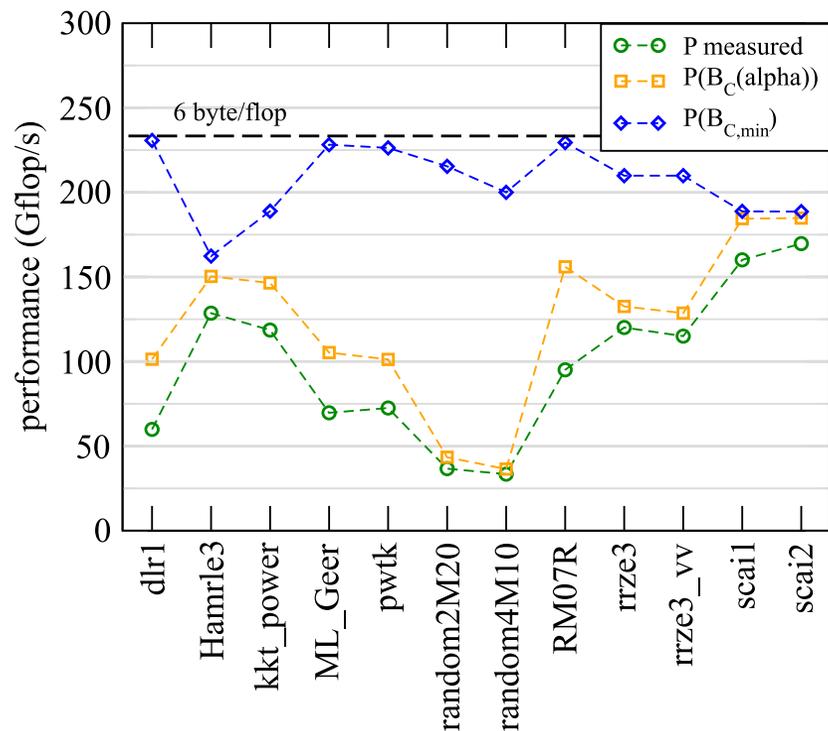
No write-allocate on GPUs for consecutive stores

SpMV CRS performance on a GPU

CRS (1 thread per row)

NVIDIA Ampere A100

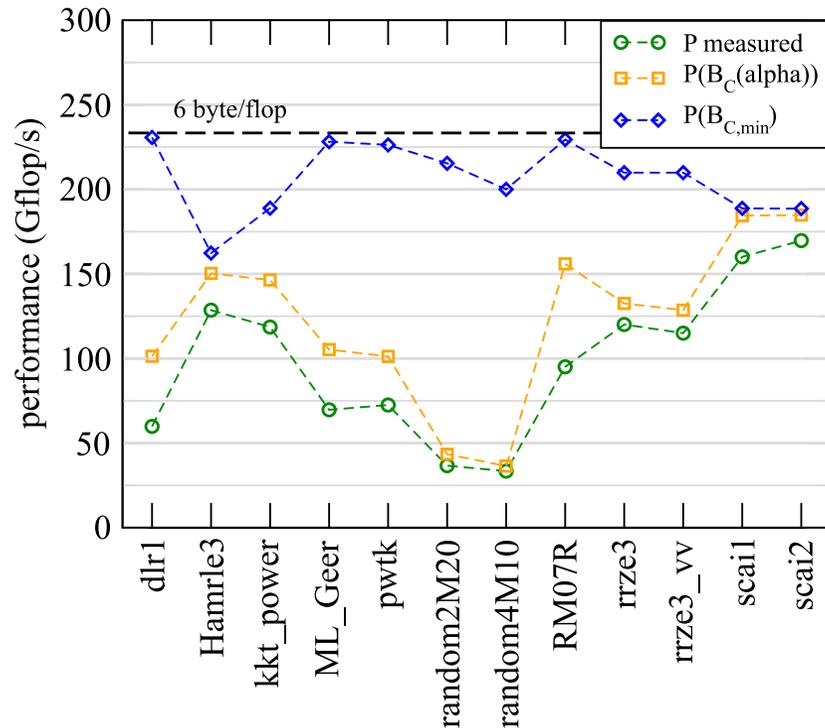
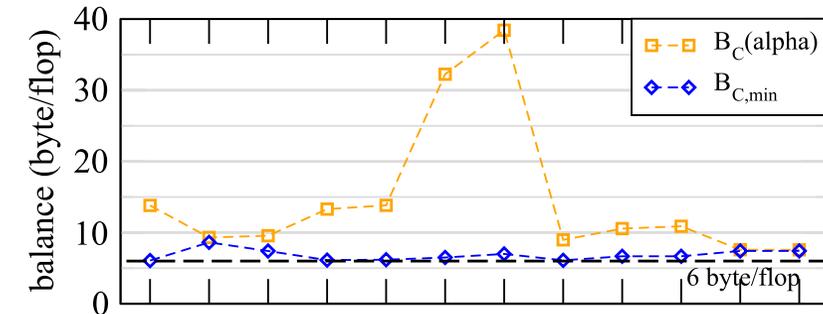
Memory bandwidth $b_S = 1400$ GB/s



SpMV CRS performance on a GPU

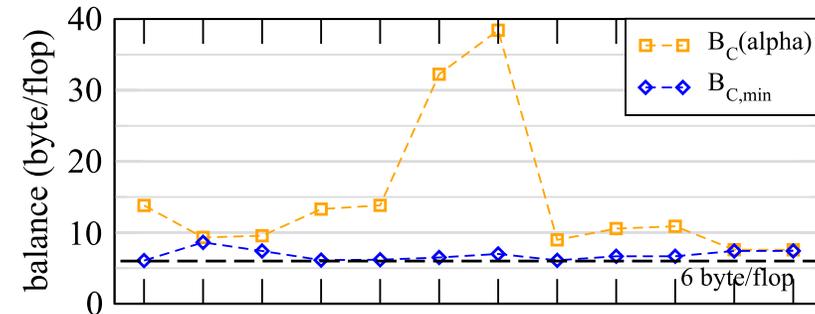
CRS (1 thread per row)

NVIDIA Ampere A100
Memory bandwidth $b_S = 1400$ GB/s



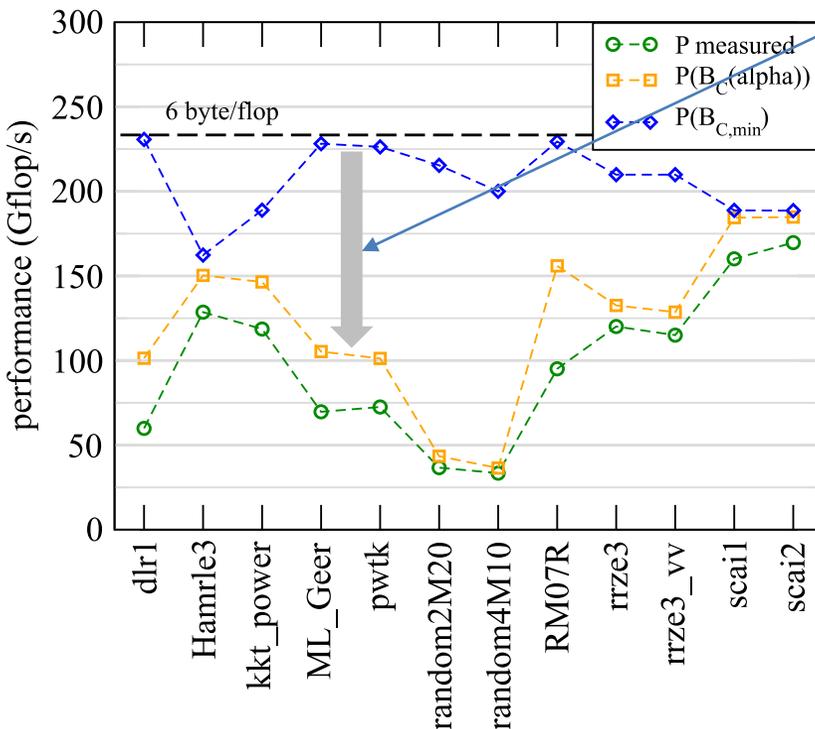
SpMV CRS performance on a GPU

CRS (1 thread per row)



NVIDIA Ampere A100

Memory bandwidth $b_S = 1400$ GB/s

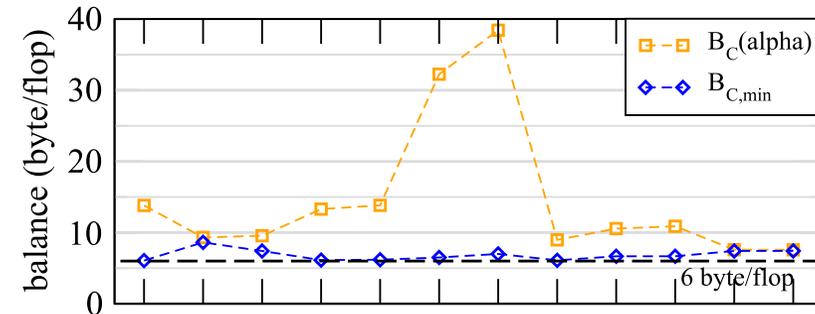


Strong “ α effect” – large deviation from optimal α for many matrices

- Many cache lines touched b/c every thread handles one row \rightarrow bad cache usage

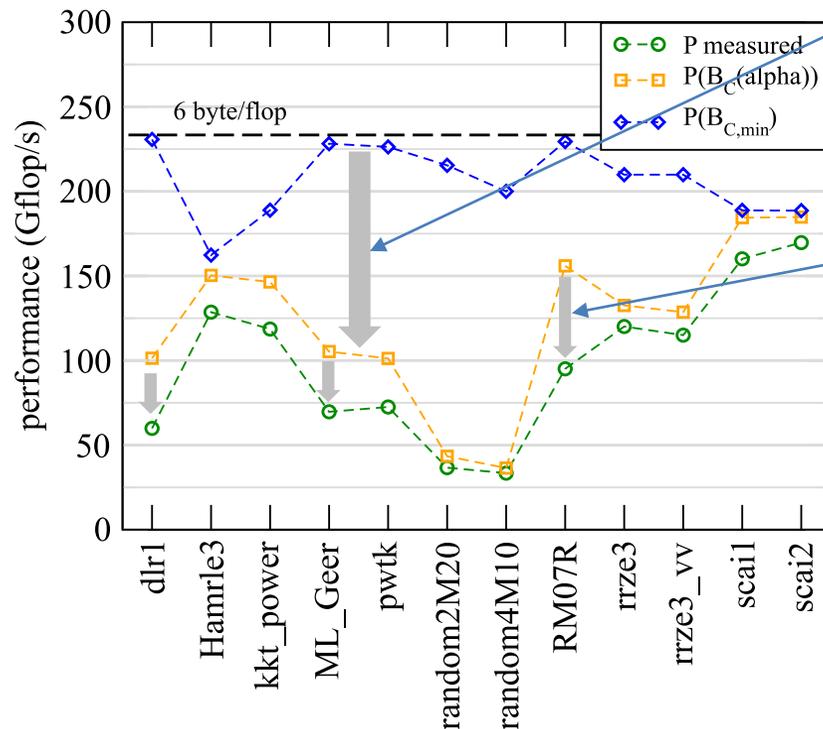
SpMV CRS performance on a GPU

CRS (1 thread per row)



NVIDIA Ampere A100

Memory bandwidth $b_S = 1400$ GB/s



■ Strong “ α effect” – large deviation from optimal α for many matrices

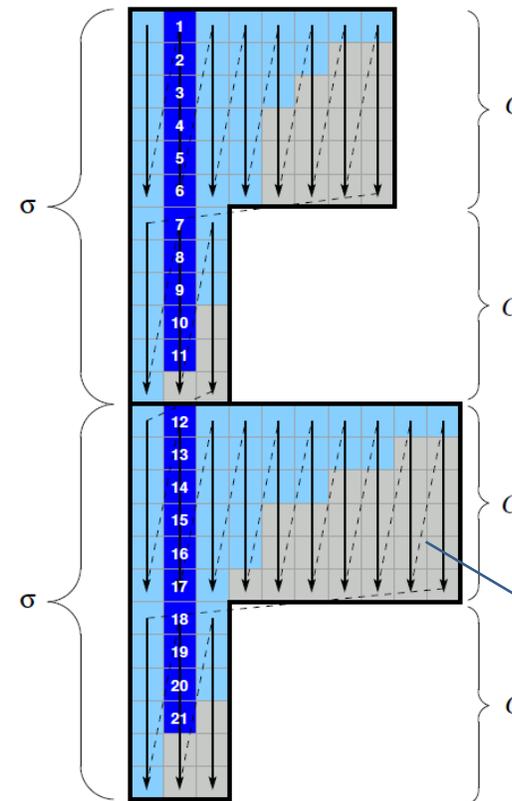
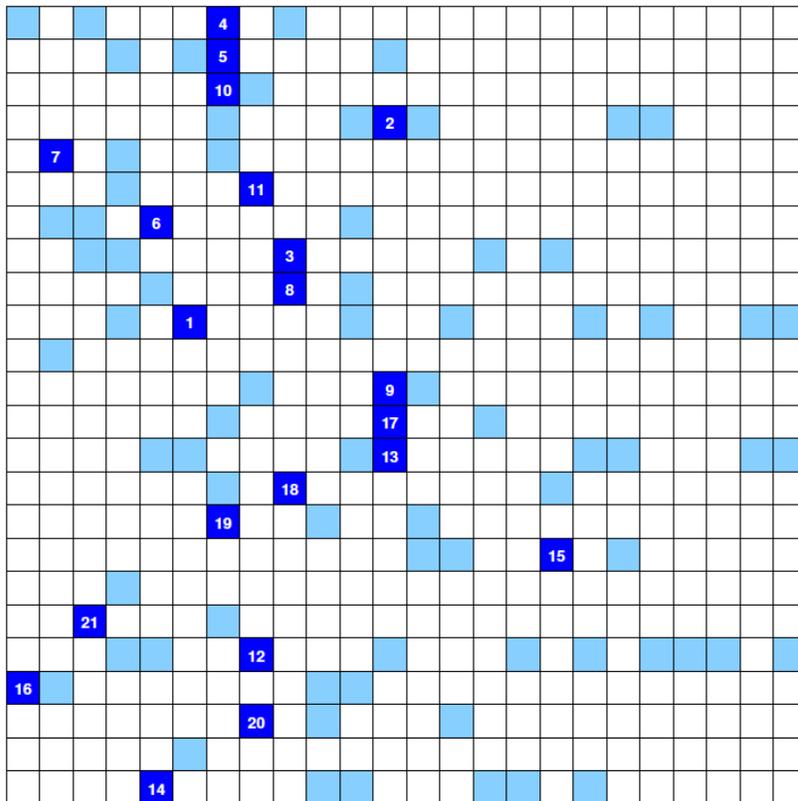
- Many cache lines touched b/c every thread handles one row \rightarrow bad cache usage

■ Mediocre memory bandwidth usage ($\ll 1400$ GB/s) in many cases

- Non-coalesced memory access
- Imbalance across rows/threads of warps

Idea

- Sort rows according to length within **sorting scope σ**
- Store nonzeros column-major in zero-padded **chunks of height C**



“Chunk occupancy”:

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$

l_i : width of chunk i

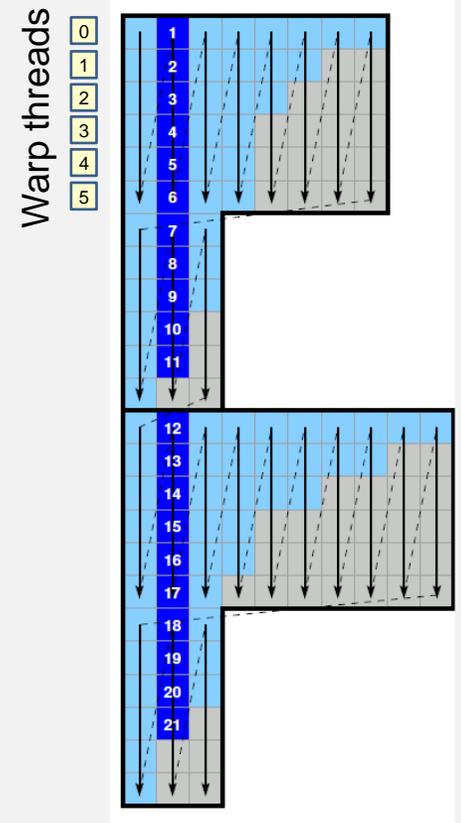
zero padding

SELL-C- σ SpMV in CUDA ($y=Ax$)

```
template <typename VT, typename IT> __global__ static void
spmv_scs(const ST C, const ST n_chunks, const IT * RESTRICT chunk_ptrs,
         const IT * RESTRICT chunk_lengths, const IT * RESTRICT col_idxs,
         const VT * RESTRICT values, const VT * RESTRICT x, VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x;
    ST c    = row / C; // the no. of the chunk
    ST idx  = row % C; // index inside the chunk

    if (row < n_chunks * C) {
        VT tmp{};
        IT cs = chunk_ptrs[c]; // points to start indices of chunks

        for (ST j = 0; j < chunk_lengths[c]; ++j) {
            tmp += values[cs + idx] * x[col_idxs[cs + idx]];
            cs += C;
        }
        y[row] = tmp;
    }
}
```



Code balance of SELL-C- σ ($y=Ax$)

$$\begin{aligned} B_{SELL}(\alpha, \beta, N_{nzs}) &= \left(\frac{1}{\beta} \left(\frac{8 + 4}{2} \right) + \frac{8\alpha + \beta(8 + 4/C)/N_{nzs}}{2} \right) \frac{\text{bytes}}{\text{flop}} \\ &= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4 + 2/C)}{N_{nzs}} \right) \frac{\text{bytes}}{\text{flop}} \end{aligned}$$

$$\text{Optimal } \alpha = \frac{\beta}{N_{nzs}}$$

Code balance of SELL-C- σ ($y=Ax$)

Matrix data &
column index

$$B_{SELL}(\alpha, \beta, N_{nzs}) = \left(\frac{1}{\beta} \left(\frac{8+4}{2} \right) + \frac{8\alpha + \beta(8+4/C)/N_{nzs}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzs}} \right) \frac{\text{bytes}}{\text{flop}}$$

$$\text{Optimal } \alpha = \frac{\beta}{N_{nzs}}$$

Code balance of SELL-C- σ ($y=Ax$)

Matrix data &
column index

LHS update (write only)

$$B_{SELL}(\alpha, \beta, N_{nzs}) = \left(\frac{1}{\beta} \left(\frac{8+4}{2} \right) + \frac{8\alpha + \beta(8+4/C)/N_{nzs}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzs}} \right) \frac{\text{bytes}}{\text{flop}}$$

$$\text{Optimal } \alpha = \frac{\beta}{N_{nzs}}$$

Code balance of SELL-C- σ ($y=Ax$)

Matrix data & column index

LHS update (write only)

chunk index

$$B_{SELL}(\alpha, \beta, N_{nzr}) = \left(\frac{1}{\beta} \left(\frac{8+4}{2} \right) + \frac{8\alpha + \beta(8+4/C)/N_{nzr}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzr}} \right) \frac{\text{bytes}}{\text{flop}}$$

Optimal $\alpha = \frac{\beta}{N_{nzr}}$

Code balance of SELL-C- σ ($y=Ax$)

Matrix data & column index

LHS update (write only)

chunk index

$$B_{SELL}(\alpha, \beta, N_{nzs}) = \left(\frac{1}{\beta} \left(\frac{8+4}{2} \right) + \frac{8\alpha + \beta(8 + 4/C)/N_{nzs}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4 + 2/C)}{N_{nzs}} \right) \frac{\text{bytes}}{\text{flop}}$$

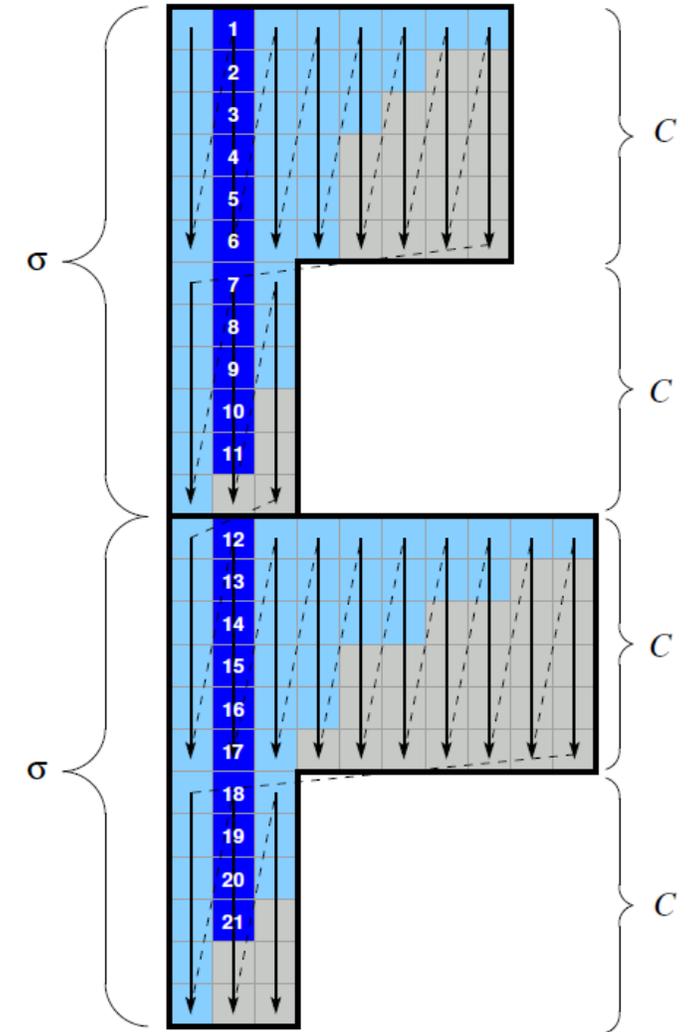
Optimal $\alpha = \frac{\beta}{N_{nzs}}$

When measuring B_C^{meas} , take care to use the “useful” number of flops (excluding zero padding) for work



How to choose the parameters C and σ on GPUs?

- C
 - $n \times$ warp size to allow good utilization of GPU threads and cache lines
- σ
 - As **small as possible**, as large as necessary
 - Large σ **reduces zero padding** (brings β closer to 1)
 - Sorting alters RHS access pattern \rightarrow **α depends on σ**

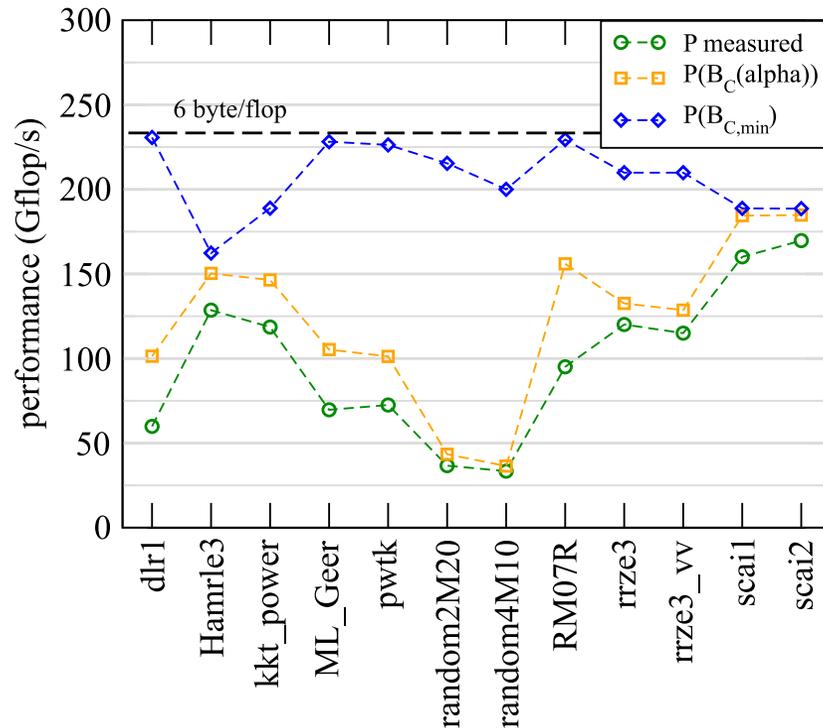


SpMV node performance model – GPU

CRS (1 thread per row)

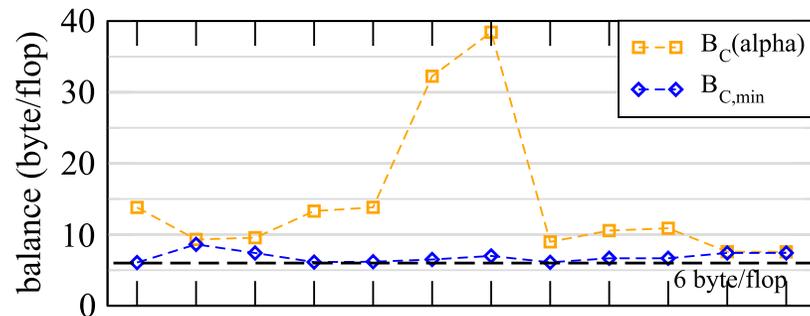
NVIDIA Ampere A100

$$b_S = 1400 \text{ GB/s}$$



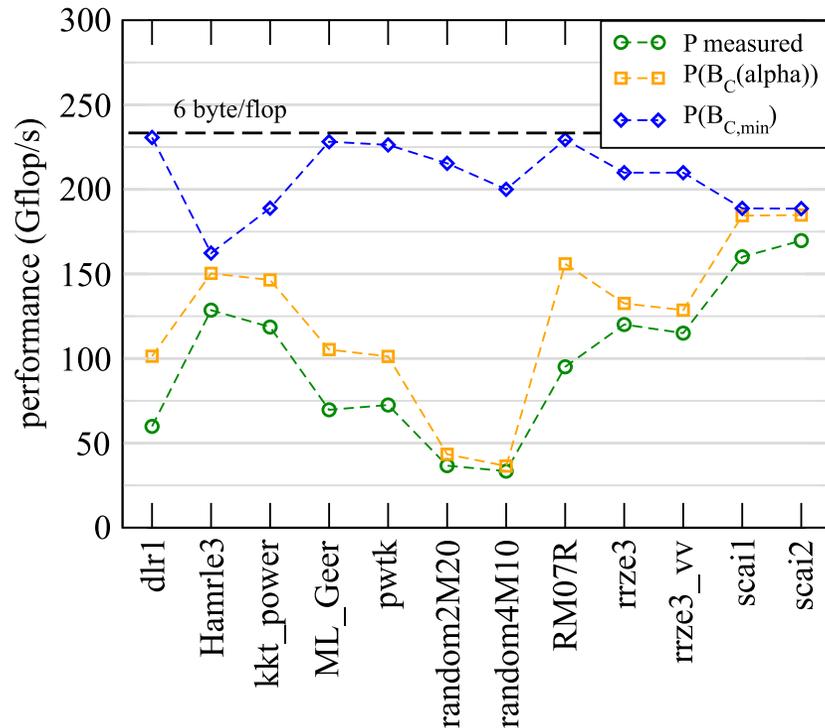
SpMV node performance model – GPU

CRS (1 thread per row)



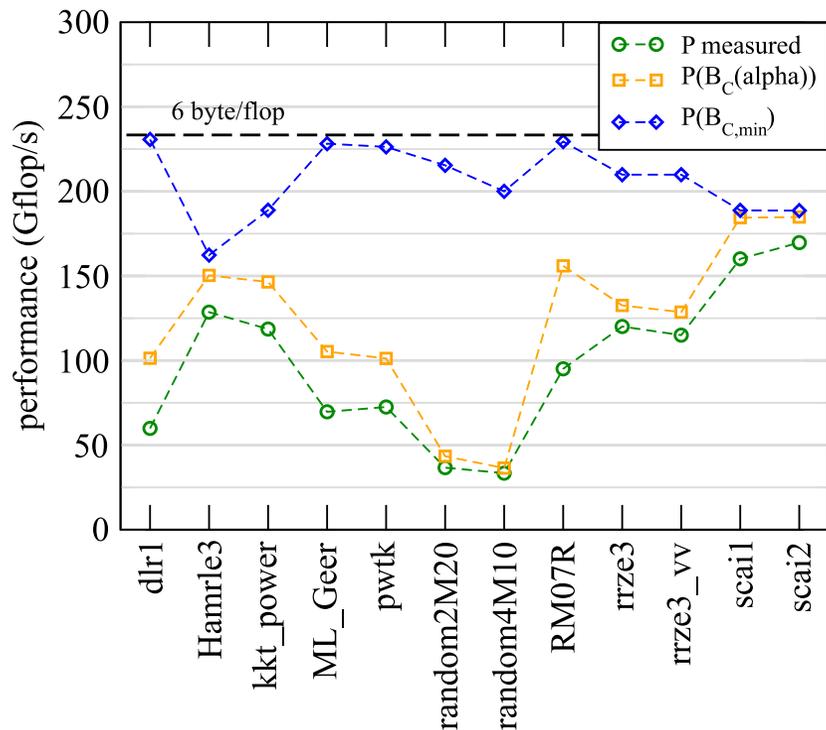
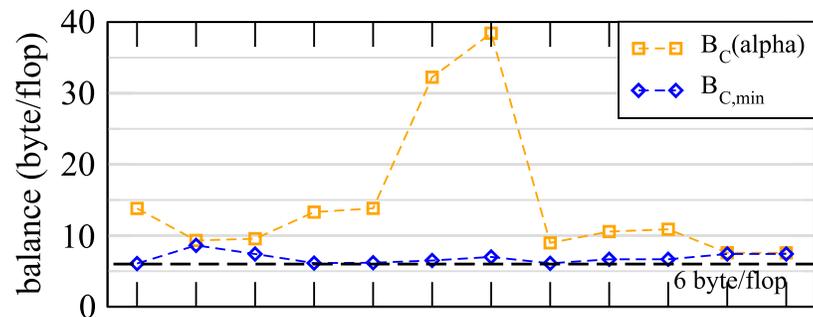
NVIDIA Ampere A100

$$b_S = 1400 \text{ GB/s}$$



SpMV node performance model – GPU

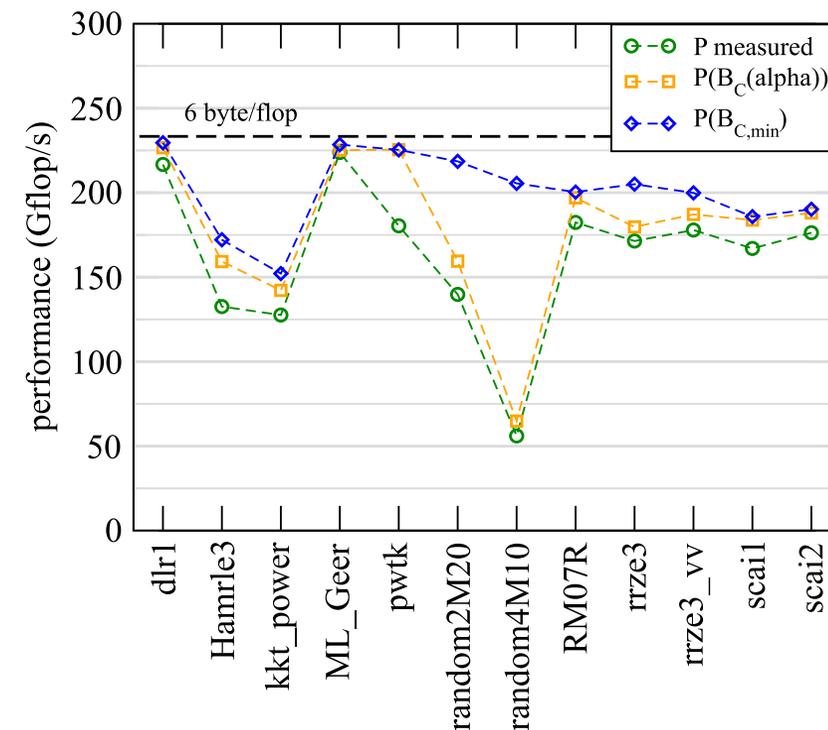
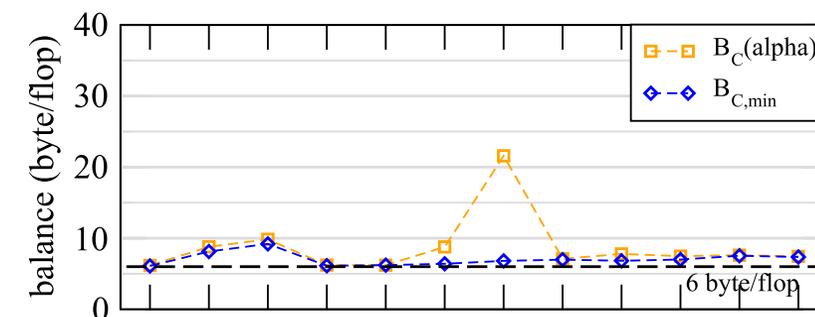
CRS (1 thread per row)



NVIDIA Ampere A100

$$b_S = 1400 \text{ GB/s}$$

SELL-32-128



Single Instruction Multiple Data (SIMD) processing



SIMD terminology

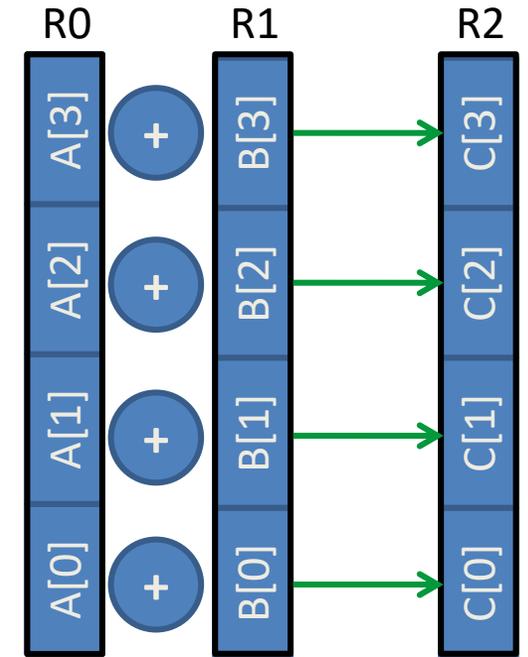
A word on terminology

- SIMD == “one instruction → several operations”
- “SIMD width” == number of operands that fit into a register
- No statement about parallelism among those operations
- Original vector computers: long registers, pipelined execution, but no parallelism (within the instruction)

SIMD terminology

A word on terminology

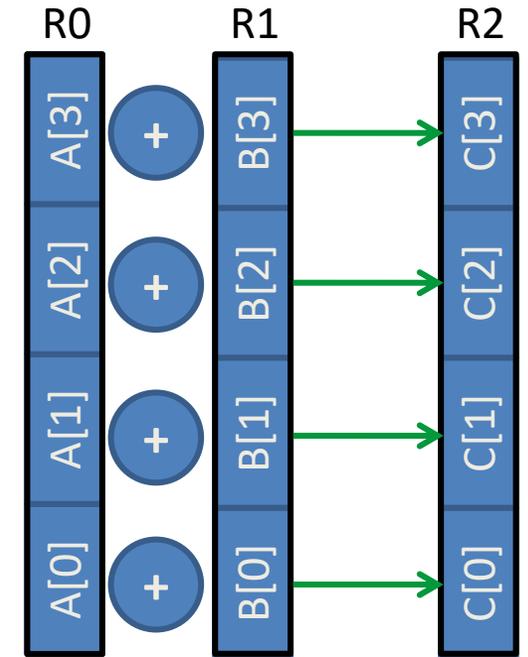
- SIMD == “one instruction → several operations”
- “SIMD width” == number of operands that fit into a register
- No statement about parallelism among those operations
- Original vector computers: long registers, pipelined execution, but no parallelism (within the instruction)



SIMD terminology

A word on terminology

- SIMD == “one instruction → several operations”
- “SIMD width” == number of operands that fit into a register
- No statement about parallelism among those operations
- Original vector computers: long registers, pipelined execution, but no parallelism (within the instruction)



Today

- x86: most SIMD instructions fully parallel
 - “Short Vector SIMD”
 - Some exceptions on some architectures (e.g., vdivpd)
- NEC Tsubasa: 32-way parallelism but SIMD width = 256 (DP)

Scalar execution units

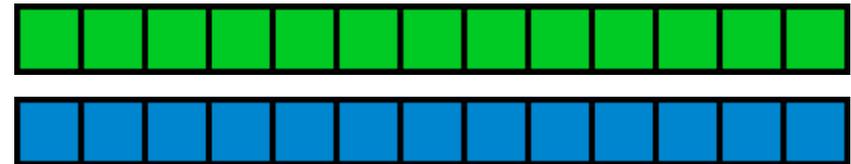
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



$$\square = \square + \square$$



Scalar execution units

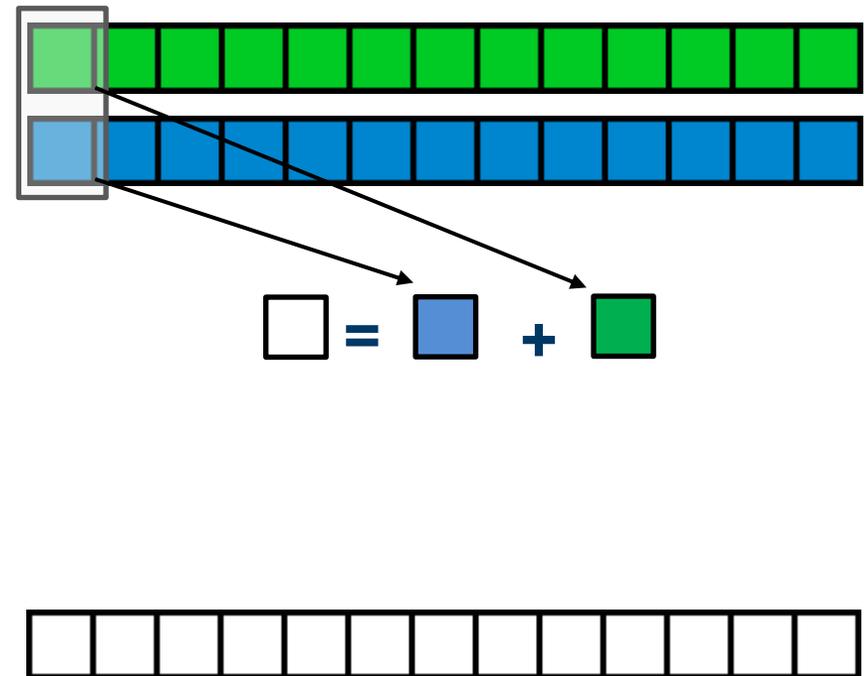
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Scalar execution units

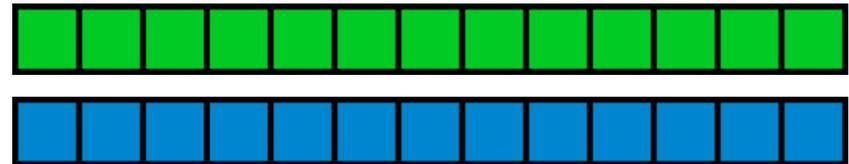
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



$$\text{Red square} = \text{Blue square} + \text{Green square}$$



Scalar execution units

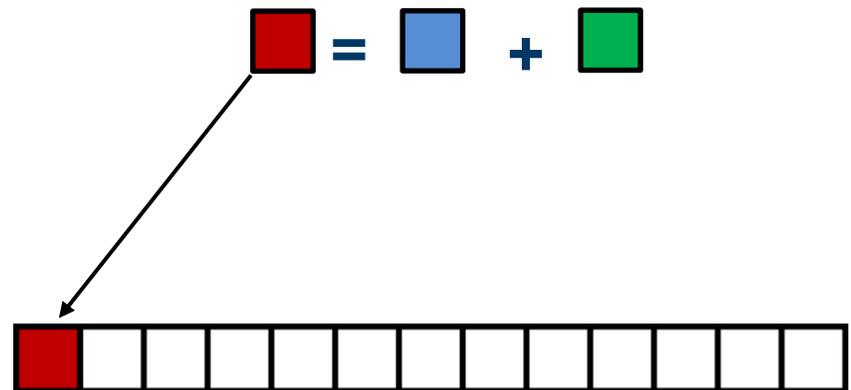
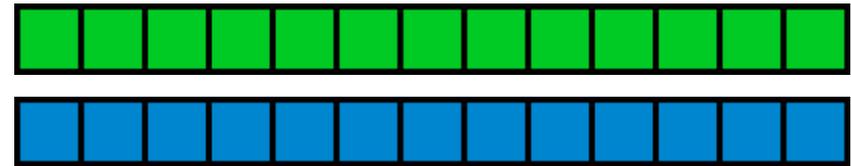
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Scalar execution units

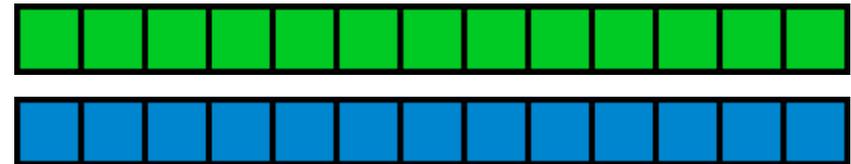
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



$$\text{red square} = \text{blue square} + \text{green square}$$



Scalar execution units

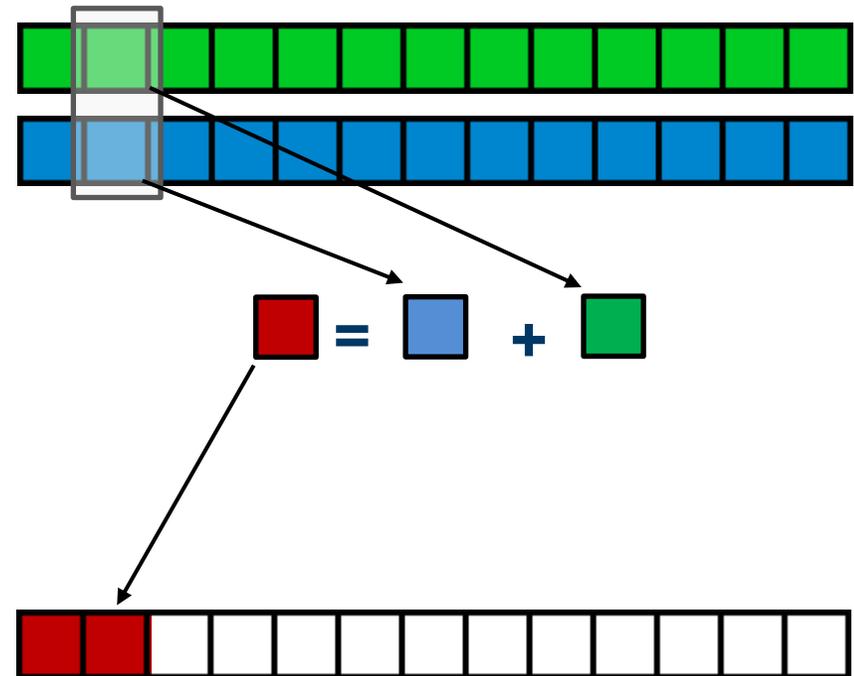
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Scalar execution units

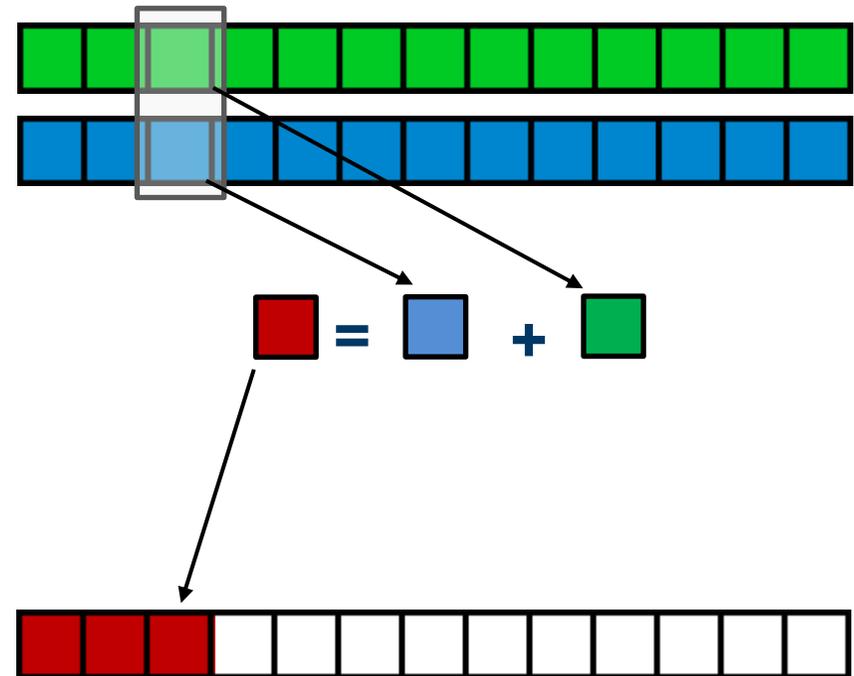
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Scalar execution units

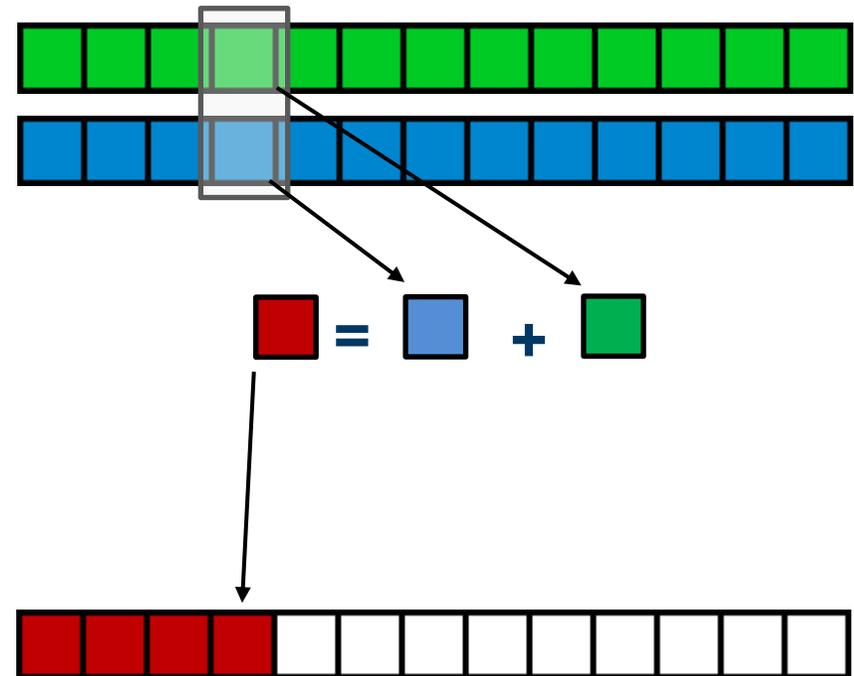
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Scalar execution units

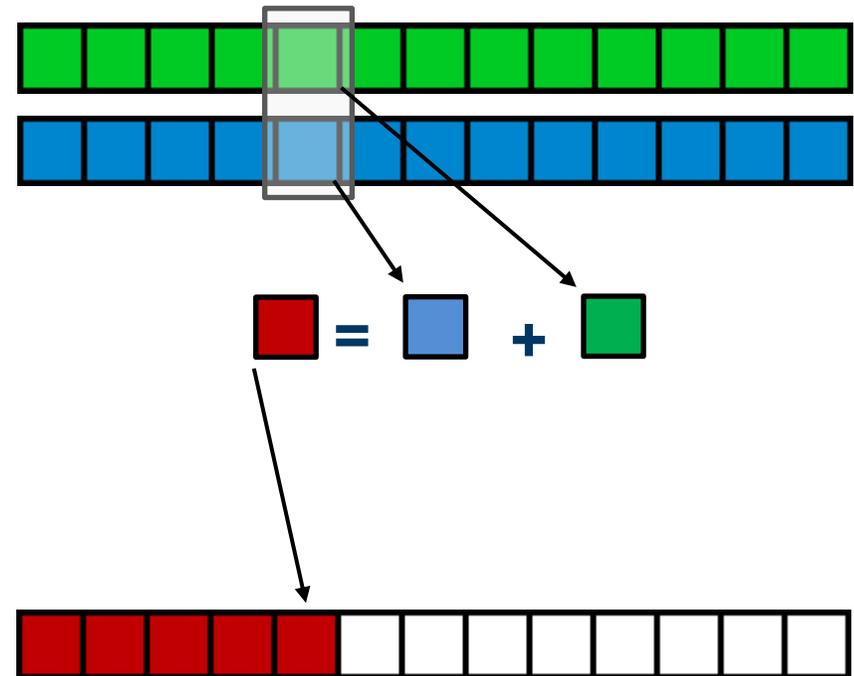
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



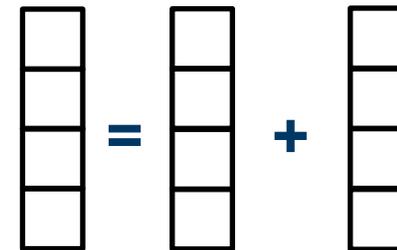
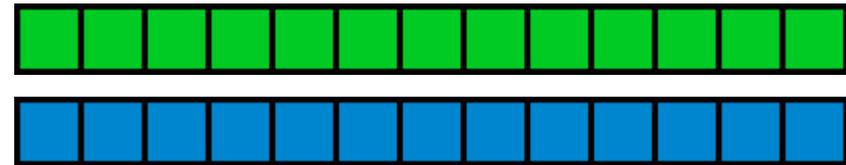
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



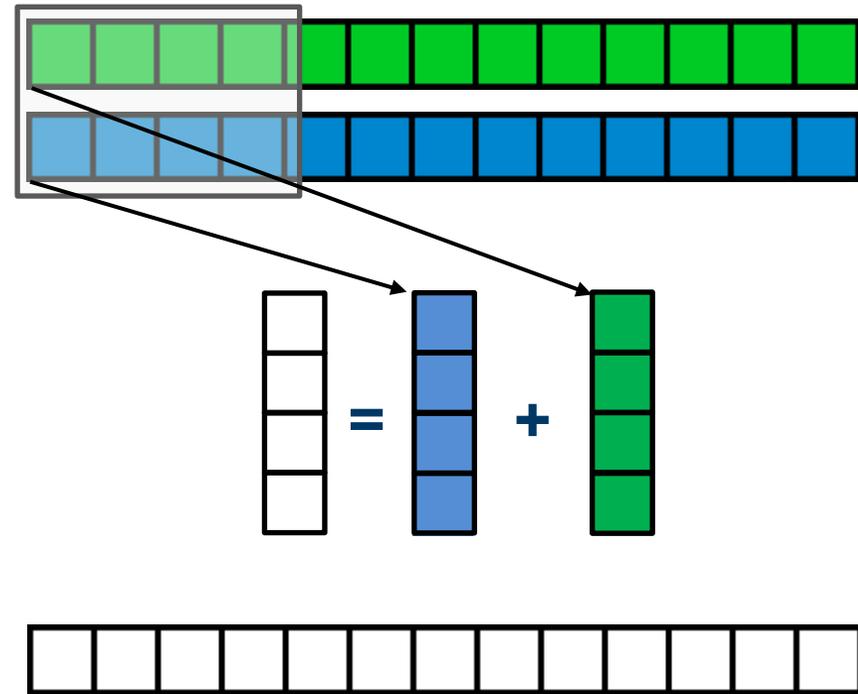
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



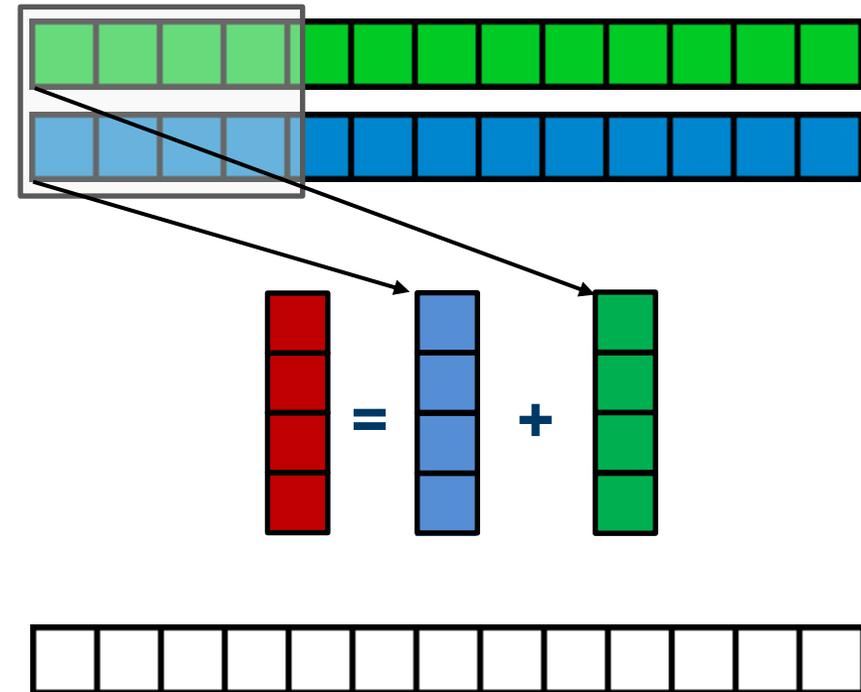
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



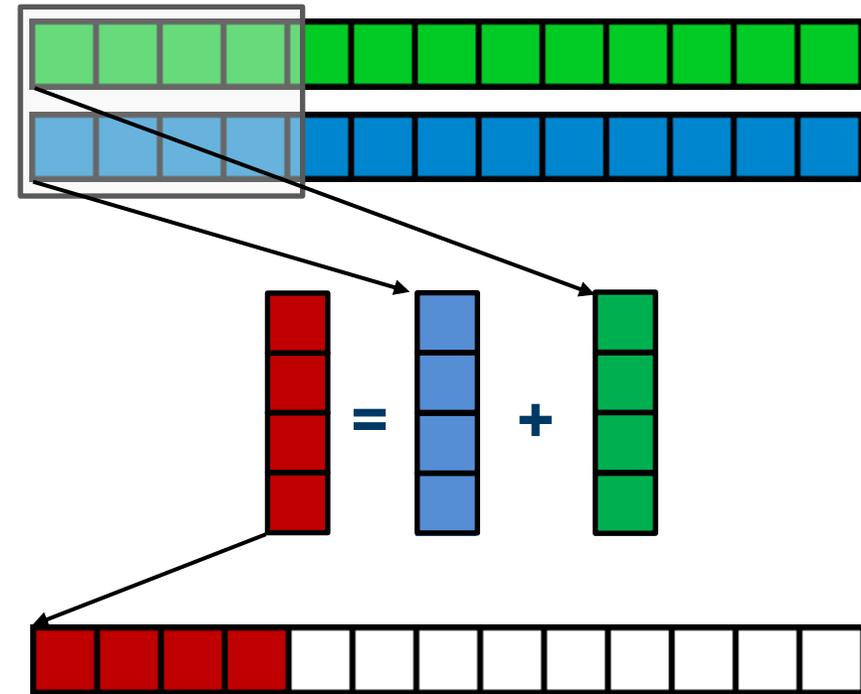
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



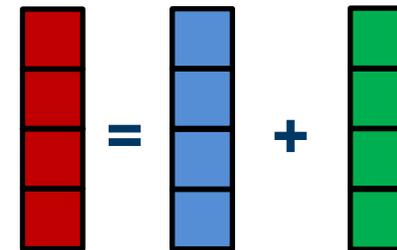
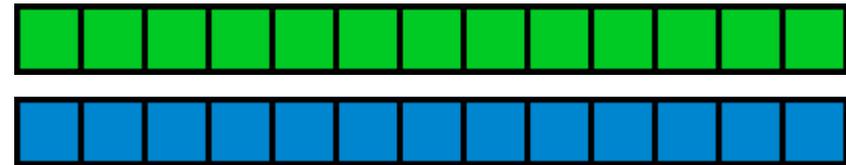
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



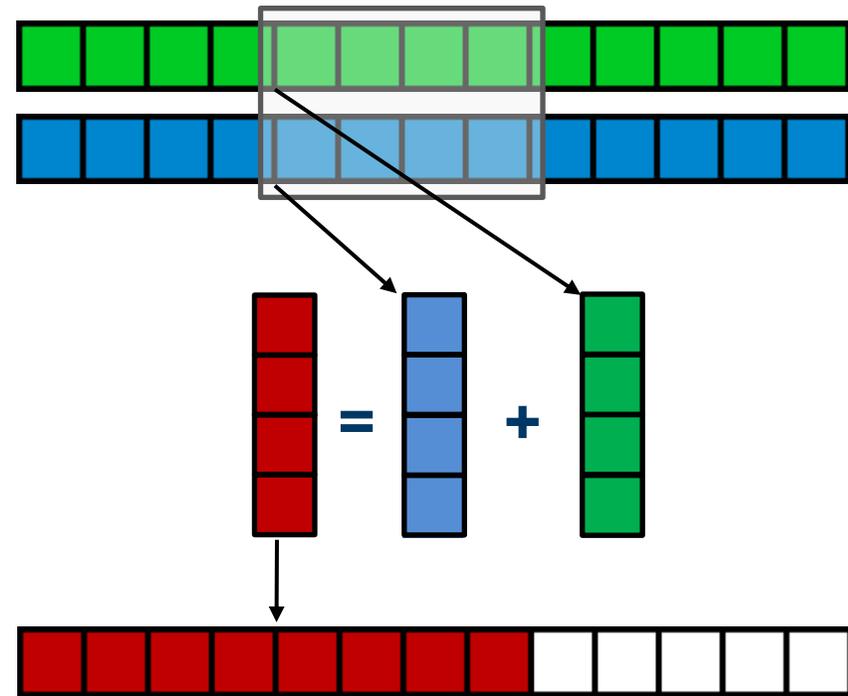
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



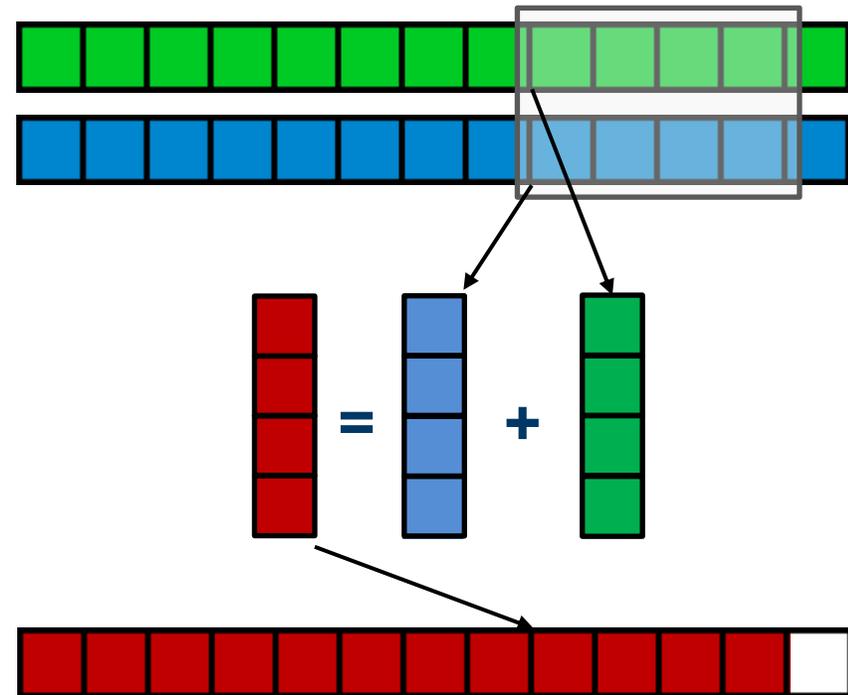
- 4 operands (AVX)



- 8 operands (AVX512)



SIMD execution



Data-parallel execution units (short vector SIMD)

```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



- 2 operands (SSE)



- 4 operands (AVX)

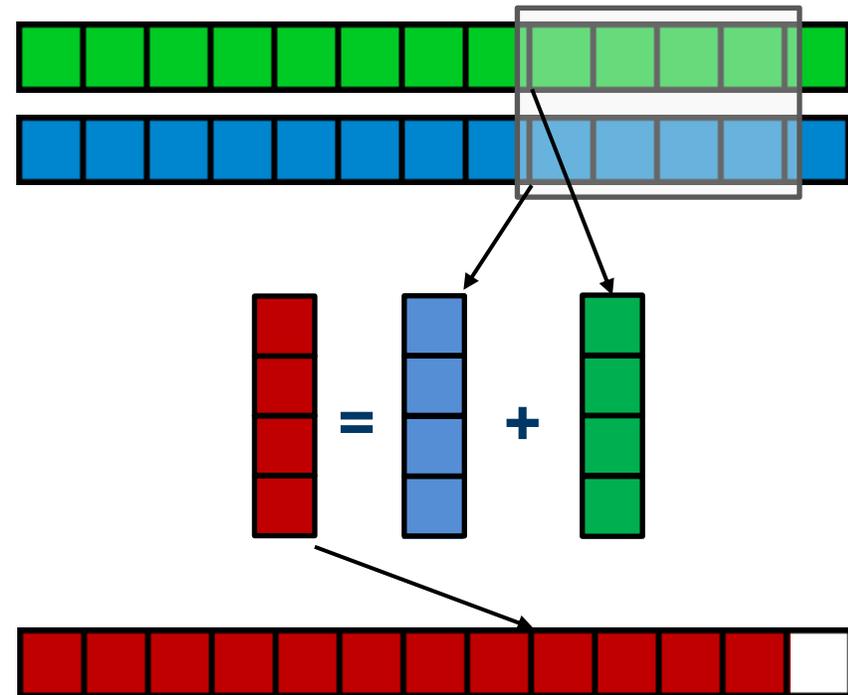


- 8 operands (AVX512)



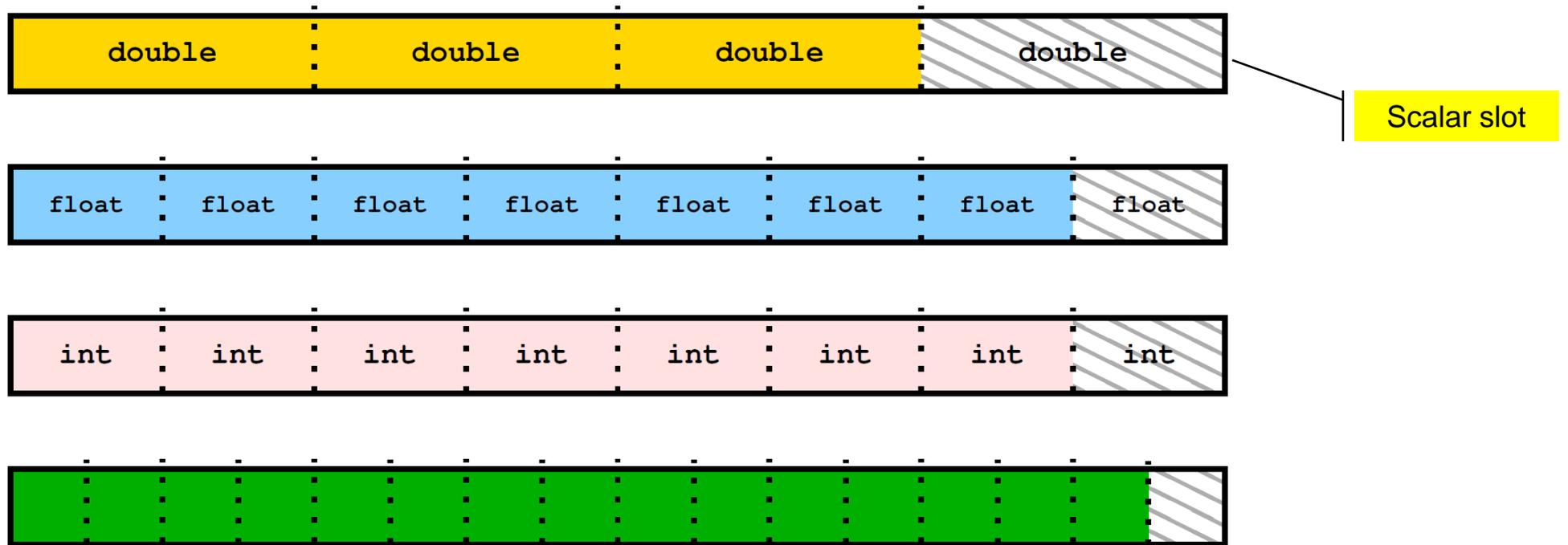
Best code requires vectorized LOADs, STOREs, and arithmetic!

SIMD execution



Data types in 32-byte SIMD registers

Supported data types depend on actual SIMD instruction set



SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

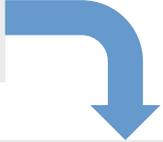
```
for(int i=0; i<n; i++)  
    C[i]= A[i] + B[i];
```

SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n; i++)  
    C[i]= A[i] + B[i];
```

“Loop unrolling”



```
for(int i=0; i<n; i+=4){  
    C[i]  = A[i]  + B[i];  
    C[i+1]= A[i+1]+ B[i+1];  
    C[i+2]= A[i+2]+ B[i+2];  
    C[i+3]= A[i+3]+ B[i+3];}  
//remainder loop handling
```

SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n; i++)  
    C[i]= A[i] + B[i];
```

“Loop unrolling”

```
for(int i=0; i<n; i+=4){  
    C[i]  = A[i]  + B[i];  
    C[i+1]= A[i+1]+ B[i+1];  
    C[i+2]= A[i+2]+ B[i+2];  
    C[i+3]= A[i+3]+ B[i+3];}  
//remainder loop handling
```

This should
not be done
by hand!



SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n; i++)  
    C[i]= A[i] + B[i];
```

“Loop unrolling”

```
for(int i=0; i<n; i+=4){  
    C[i]  = A[i]  + B[i];  
    C[i+1]= A[i+1]+ B[i+1];  
    C[i+2]= A[i+2]+ B[i+2];  
    C[i+3]= A[i+3]+ B[i+3];  
    //remainder loop handling
```

This should
not be done
by hand!



```
LABEL1:  
    VLOAD R0 ← A[i]  
    VLOAD R1 ← B[i]  
    V64ADD[R0,R1] → R2  
    VSTORE R2 → C[i]  
    i←i+4  
    i<(n-4)? JMP LABEL1  
    //remainder loop handling
```

SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n; i++)  
    C[i]= A[i] + B[i];
```

“Loop unrolling”

```
for(int i=0; i<n; i+=4){  
    C[i]  = A[i]  + B[i];  
    C[i+1]= A[i+1]+ B[i+1];  
    C[i+2]= A[i+2]+ B[i+2];  
    C[i+3]= A[i+3]+ B[i+3];  
    //remainder loop handling
```

This should
not be done
by hand!



Load 256 Bits starting from address of
A[i] to register R0, B[i] in R1

Add the corresponding 64 Bit entries in
R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

```
LABEL1:  
    VLOAD R0 ← A[i]  
    VLOAD R1 ← B[i]  
    V64ADD[R0,R1] → R2  
    VSTORE R2 → C[i]  
    i ← i+4  
    i < (n-4)? JMP LABEL1  
    //remainder loop handling
```

SIMD processing: Roadblocks

- No SIMD vectorization for loops with data dependencies:

```
for(int i=1; i<n; i++)  
    A[i] = A[i-1] * s;
```

SIMD processing: Roadblocks

- No SIMD vectorization for loops with data dependencies:

```
for(int i=1; i<n; i++)  
    A[i] = A[i-1] * s;
```

- “**Pointer aliasing**” may prevent vectorization

```
void f(double *A, double *B, double *C, int n) {  
    for(int i=0; i<n; ++i)  
        C[i] = A[i] + B[i];  
}
```

C/C++ allows: $A = \&C[-1]$ and $B = \&C[-2]$ $\rightarrow C[i] = C[i-1] + C[i-2]$

\rightarrow data dependency \rightarrow **no SIMD**

SIMD processing: Roadblocks

- No SIMD vectorization for loops with data dependencies:

```
for(int i=1; i<n; i++)  
    A[i] = A[i-1] * s;
```

- “**Pointer aliasing**” may prevent vectorization

```
void f(double *A, double *B, double *C, int n) {  
    for(int i=0; i<n; ++i)  
        C[i] = A[i] + B[i];  
}
```

C/C++ allows: $A = \&C[-1]$ and $B = \&C[-2]$ $\rightarrow C[i] = C[i-1] + C[i-2]$

\rightarrow data dependency \rightarrow **no SIMD**

- If **pointer aliasing does not occur** in code, tell the compiler:

-fno-alias (Intel), **-Msafeptr** (PGI), **-fargument-noalias** (gcc)

restrict keyword (C only!):

```
void f(double *restrict A, double *restrict B, double *restrict C, int n) {...}
```

How to leverage SIMD: your options

Options:

- The **compiler** does it for you
(but: aliasing, alignment, language, abstractions)
- Compiler directives (**pragmas**) – **OpenMP 4.0++** has ample support
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembly**

How to leverage SIMD: your options

Options:

- The **compiler** does it for you
(but: aliasing, alignment, language, abstractions)
- Compiler directives (**pragmas**) – **OpenMP 4.0++** has ample support
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembly**

Example: x86 SIMD (SSE) intrinsics

```
#include <x86intrin.h>
...
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

Vectorization compiler options (Intel)

optional

- The compiler will vectorize starting with **-O2**
- To enable specific SIMD extensions use the **-x** option:
-xSSE2, **-xSSE3**, **-xSSSE3**, **-xSSE4.1**, **-xSSE4.2**, **-xAVX**, ...
- **-xAVX** on Sandy/Ivy Bridge processors
- **-xCORE-AVX2** on Haswell/Broadwell
- **-xCORE-AVX512** on Skylake (certain models) and later

Recommended option:

- **-xHost** will optimize for the architecture you compile on
- To really enable 512-bit SIMD with current Intel compilers you need to set **-qopt-zmm-usage=high**

User-mandated vectorization (OpenMP 4)

- Since OpenMP 4.0 SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
Do not lie to the compiler!

User-mandated vectorization (OpenMP 4)

- Since OpenMP 4.0 SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
Do not lie to the compiler!

```
for (int j=0; j<n; j++) {  
    #pragma omp simd reduction(+:b[j:1])  
    for (int i=0; i<n; i++) {  
        b[j] += a[j][i];  
    }  
}
```

User-mandated vectorization (OpenMP 4)

- Since OpenMP 4.0 SIMD features are a part of the OpenMP standard
- `#pragma omp simd` enforces vectorization
- Essentially a standardized “go ahead, no dependencies here!”
Do not lie to the compiler!

- Prerequisites

- Countable loop
- Innermost loop

- Must conform to for-loop style of OpenMP worksharing constructs

- There are additional clauses:

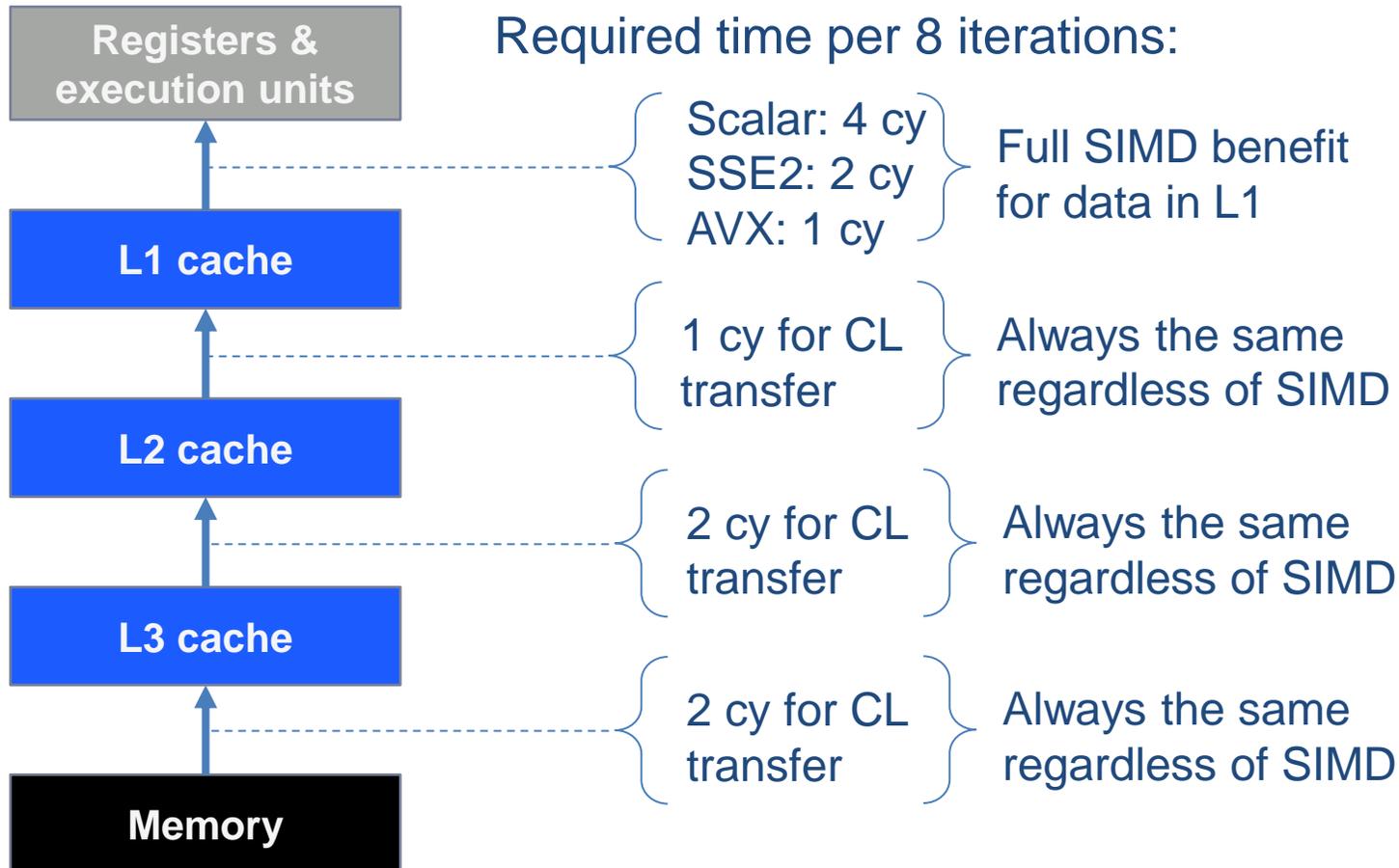
reduction, simdlen, private, collapse, ...

```
for (int j=0; j<n; j++) {  
    #pragma omp simd reduction(+:b[j:1])  
    for (int i=0; i<n; i++) {  
        b[j] += a[j][i];  
    }  
}
```

Limits of the SIMD benefit

Why does SIMD usually not give the expected speedup?
→ Analyze time contributions for data and execution

```
for(int i=0; i<size; i++)  
    sum += data[i];
```

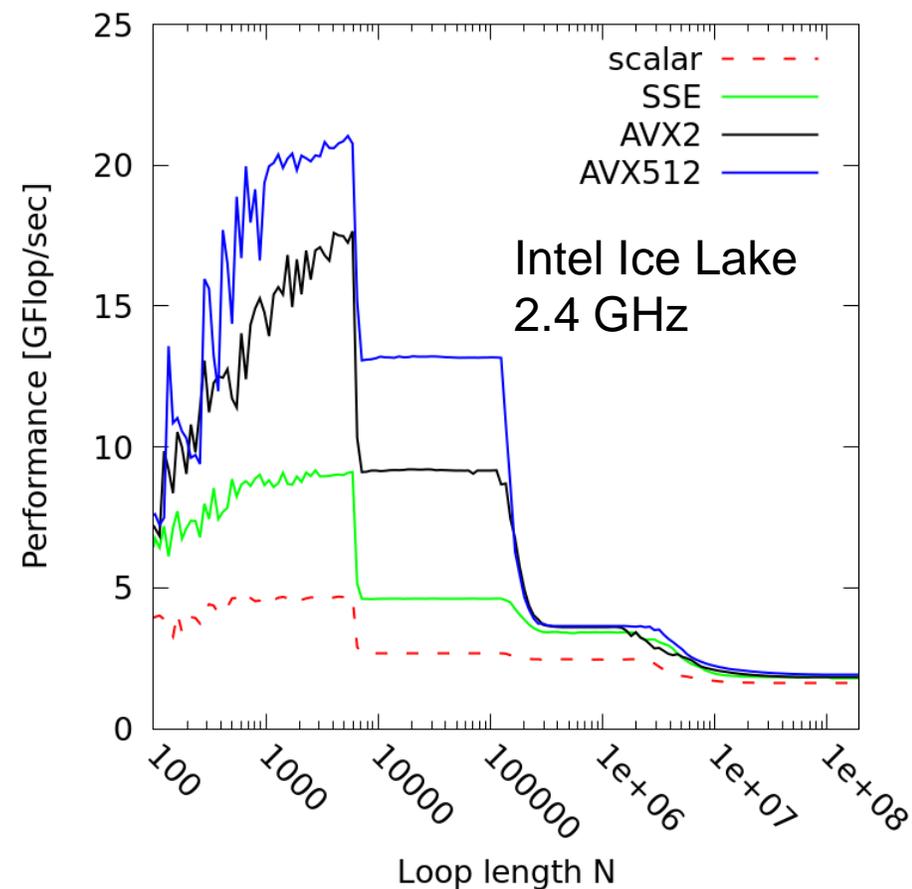
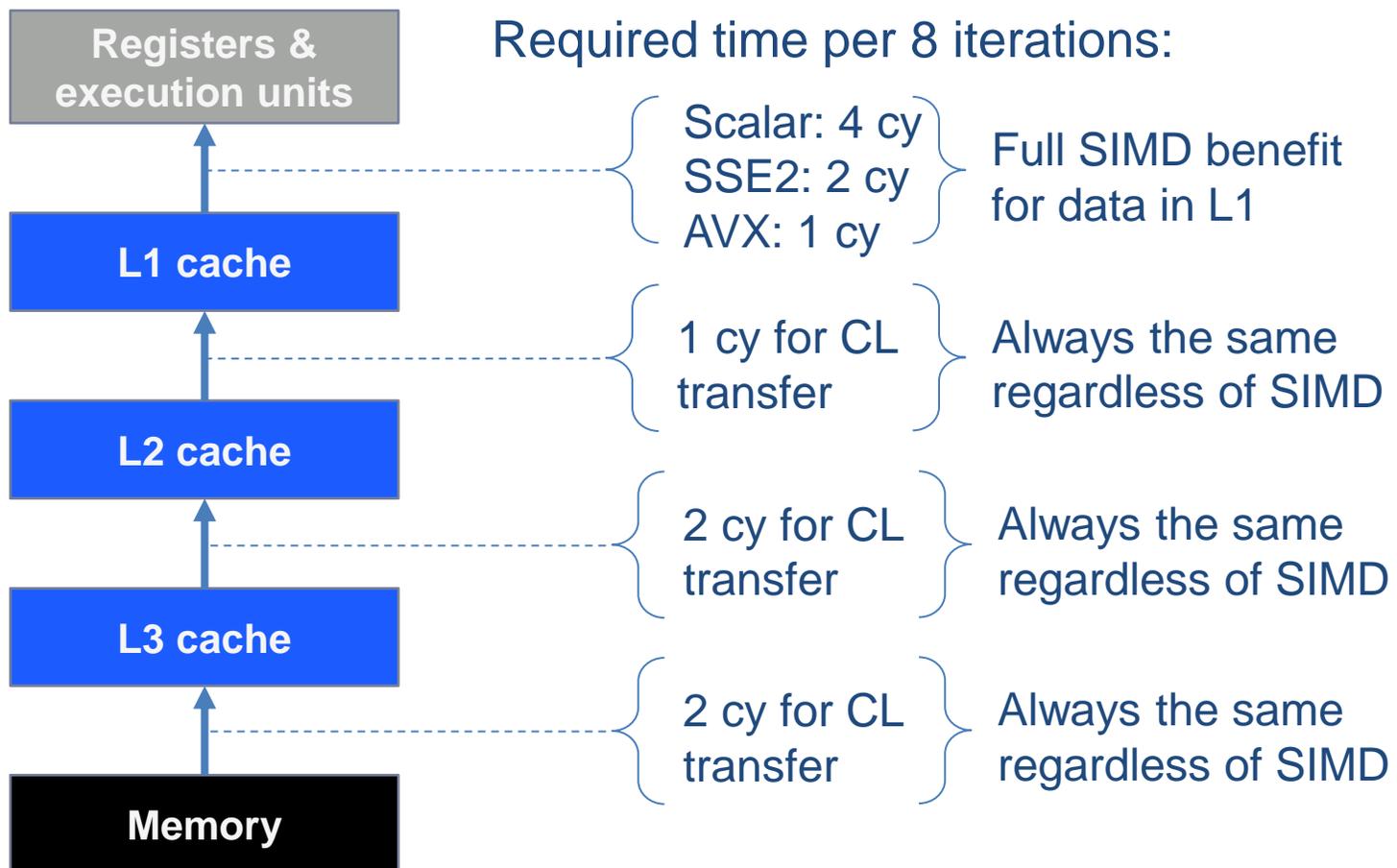


Limits of the SIMD benefit

Why does SIMD usually not give the expected speedup?

→ Analyze time contributions for data and execution

```
for(int i=0; i<size; i++)  
    sum += data[i];
```



Rules and guidelines for vectorizable loops

1. **Inner** loop
2. **Countable** (loop length can be determined at loop entry)
3. **Single entry** and single exit
4. **Straight line code** (no conditionals) – unless masks can be used
5. No function calls (exceptions: SIMD declared functions, intrinsic math)

Better performance with:

1. Simple inner loops with **unit stride** (contiguous data access)
2. **Minimize indirect** addressing
3. **Align** data structures to SIMD width boundary (minor impact)

In C use the **restrict** keyword and/or **const** qualifiers and/or compiler options to rule out array/pointer aliasing

**Keep it
simple,
stupid!**

SIMD conclusions

- Short-vector SIMD = data-parallel execution on the instruction level
- Best option: make the compiler employ SIMD instructions

- SIMD is an in-core feature
 - Boosts work per cycle in core (peak performance)
 - The further away the data, the less benefit
 - If the code is memory bound, you may not even care

Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

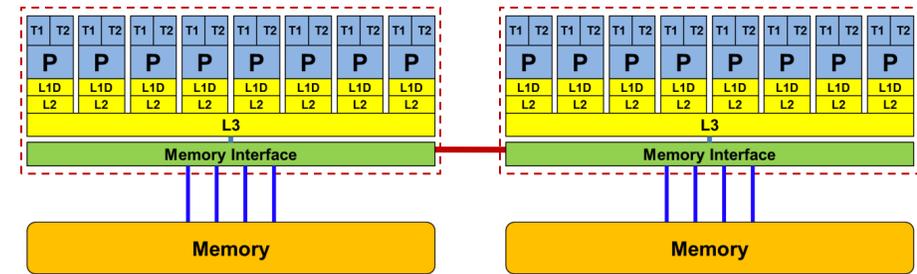
First touch placement policy



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

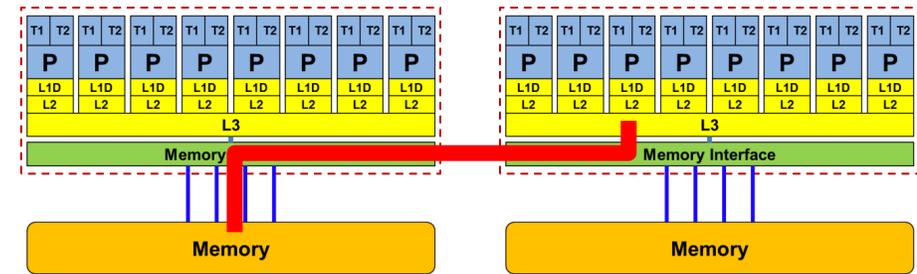
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

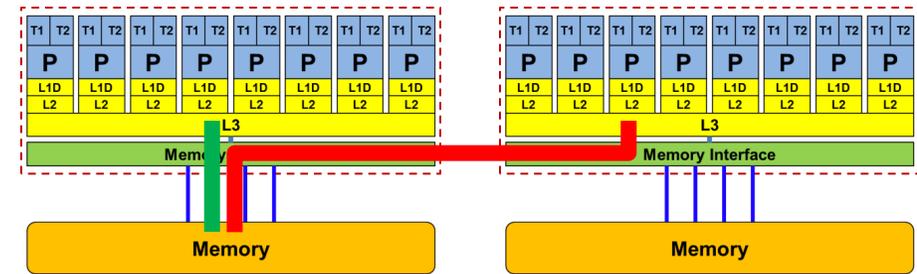
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

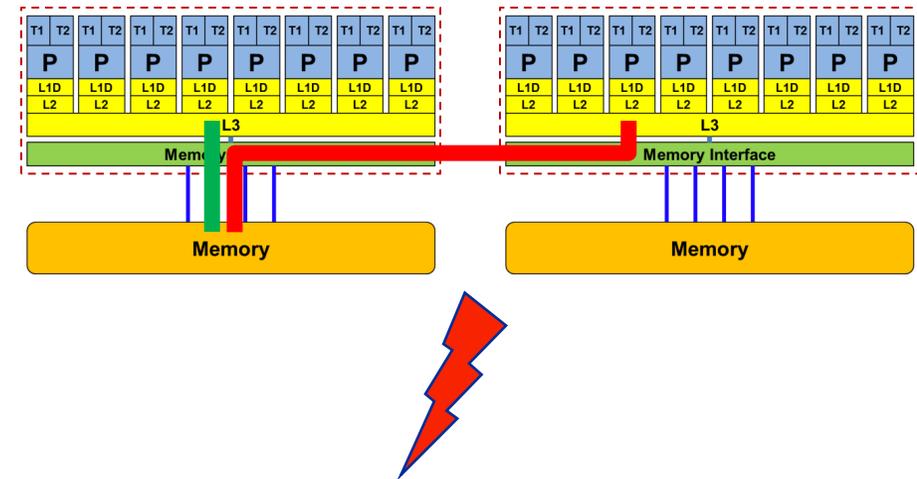
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

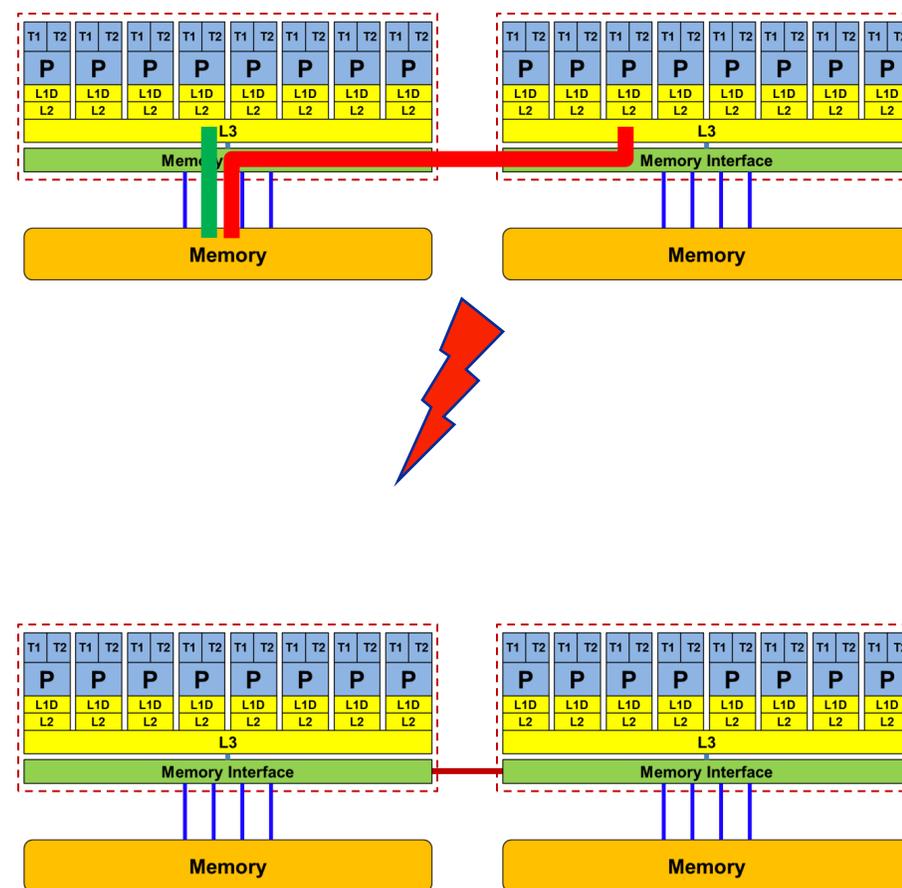
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

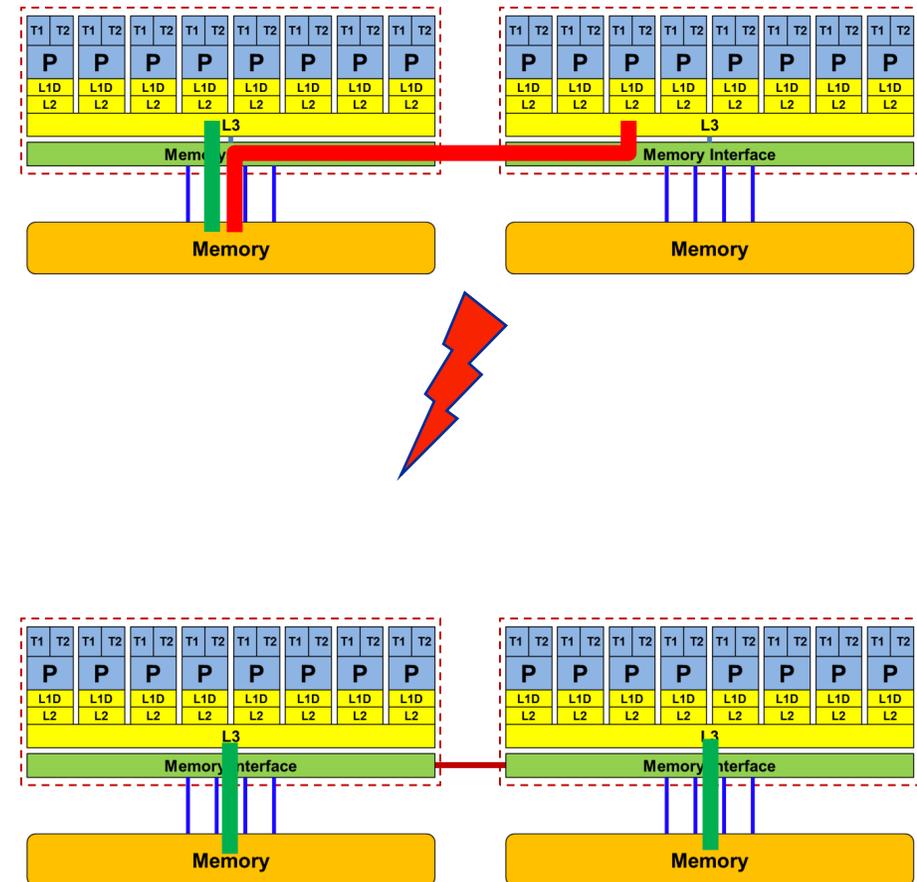
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

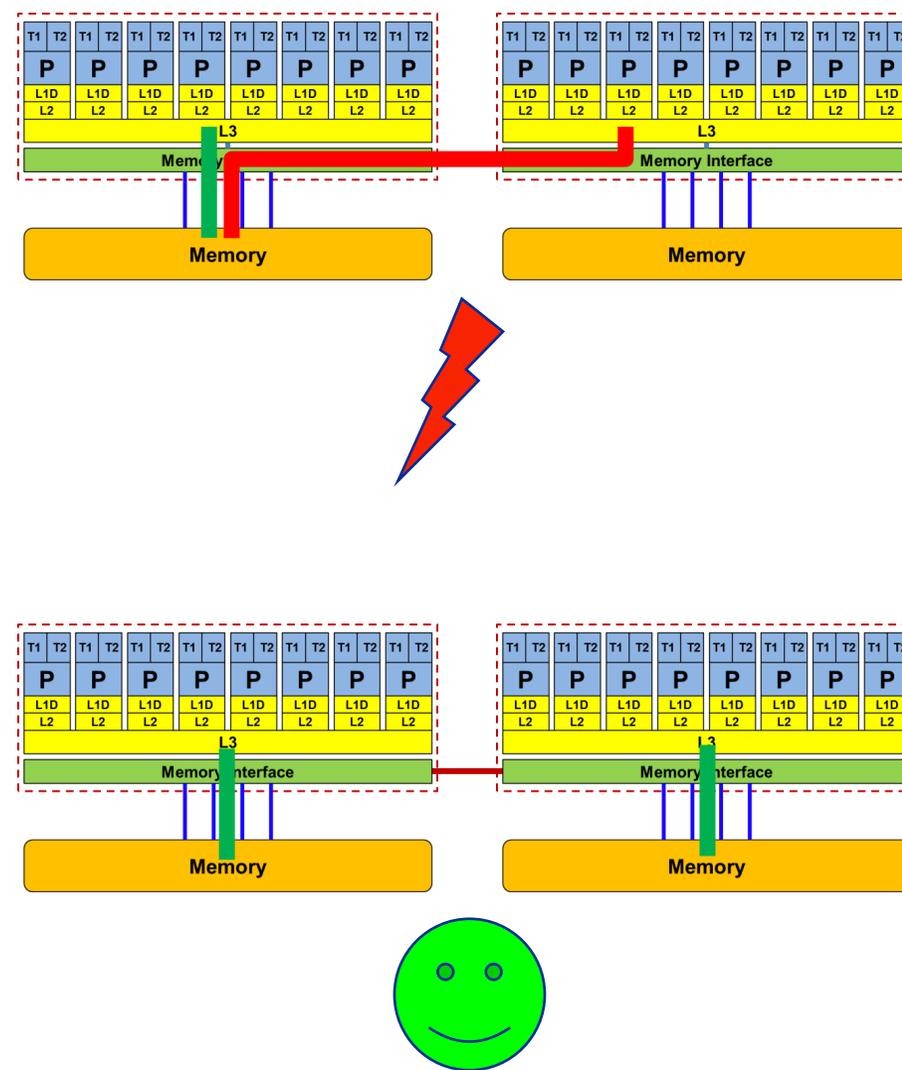
Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)



ccNUMA – The “other affinity”

- ccNUMA:
 - Whole memory is **transparently accessible** by all processors
 - but **physically distributed** across multiple locality domains (LDs)
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?

Note: Page placement is implemented in units of OS pages (often 4 KiB, possibly more)

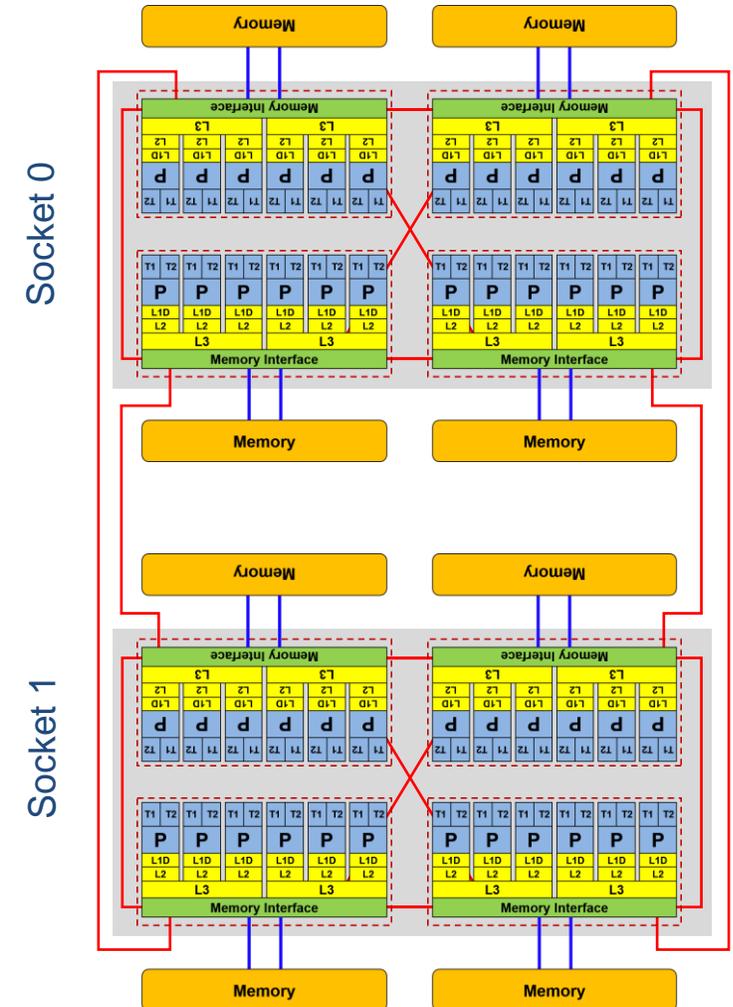


How much does nonlocal access cost?

Example: AMD “Naples” dual-socket system
(8 chips, 2 sockets, 48 cores):

STREAM Triad bandwidth measurements [Gbyte/s]

CPU node	0	1	2	3	4	5	6	7
MEM node								
0	32.4	21.4	21.8	21.9	10.6	10.6	10.7	10.8
1	21.5	32.4	21.9	21.9	10.6	10.5	10.7	10.6
2	21.8	21.9	32.4	21.5	10.6	10.6	10.8	10.7
3	21.9	21.9	21.5	32.4	10.6	10.6	10.6	10.7
4	10.6	10.7	10.6	10.6	32.4	21.4	21.9	21.9
5	10.6	10.6	10.6	10.6	21.4	32.4	21.9	21.9
6	10.6	10.7	10.6	10.6	21.9	21.9	32.3	21.4
7	10.7	10.6	10.6	10.6	21.9	21.9	21.4	32.5



numactl as a simple ccNUMA locality tool : *How do we enforce some locality of access?*

- **numactl** can influence the way a binary maps its memory pages:

```
numactl --membind=<nodes> a.out      # map pages only on <nodes>
      --preferred=<node> a.out      # map pages on <node>
                                      # and others if <node> is full
      --interleave=<nodes> a.out    # map pages round robin across
                                      # all <nodes>
```

- Examples:

```
for m in `seq 0 7`; do
  for c in `seq 0 7`; do
    env OMP_NUM_THREADS=6 \
      numactl --membind=$m likwid-pin -c M${c}:0-5 ./stream
  done
done
```

ccNUMA map scan
for Naples system

```
numactl --interleave=0-7 likwid-pin -c E:N:8:1:12 ./stream
```

numactl as a simple ccNUMA locality tool : *How do we enforce some locality of access?*

- **numactl** can influence the way a binary maps its memory pages:

```
numactl --membind=<nodes> a.out      # map pages only on <nodes>
      --preferred=<node> a.out      # map pages on <node>
                                      # and others if <node> is full
      --interleave=<nodes> a.out     # map pages round robin across
                                      # all <nodes>
```

- Examples:

```
for m in `seq 0 7`; do
  for c in `seq 0 7`; do
    env OMP_NUM_THREADS=6 \
      numactl --membind=$m likwid-pin -c M${c}:0-5 ./stream
  done
done
```

ccNUMA map scan
for Naples system

```
numactl --interleave=0-7 likwid-pin -c E:N:8:1:12 ./stream
```

- But what is the default without **numactl**?

ccNUMA default memory locality

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

(Except if there is not enough local memory available)

- **Caveat:** “to touch” means “to write,” not “to allocate”

ccNUMA default memory locality

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

(Except if there is not enough local memory available)

- **Caveat:** “to touch” means “to write,” not “to allocate”
- Example:

```
double *huge = (double*)malloc(N*sizeof(double));
```

```
for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)  
    huge[i] = 0.0;
```

- It is sufficient to touch a single item to map the entire page

ccNUMA default memory locality

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

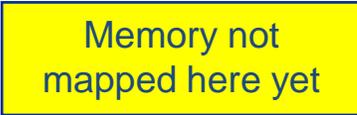
(Except if there is not enough local memory available)

- **Caveat:** “to touch” means “to write,” not “to allocate”

- Example:

```
double *huge = (double*)malloc(N*sizeof(double));
```

```
for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)  
    huge[i] = 0.0;
```



Memory not mapped here yet

- It is sufficient to touch a single item to map the entire page

ccNUMA default memory locality

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

(Except if there is not enough local memory available)

- **Caveat:** "to touch" means "to write," not "to allocate"

- Example:

```
double *huge = (double*)malloc(N*sizeof(double));
```

```
for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)  
    huge[i] = 0.0;
```

Memory not
mapped here yet

Mapping takes
place here

- It is sufficient to touch a single item to map the entire page

Coding for ccNUMA data locality

Simplest case: explicit initialization

```
integer,parameter :: N=10000000  
double precision A(N), B(N)
```

```
A=0.d0
```



```
!$OMP parallel do  
do i = 1, N  
    B(i) = function ( A(i) )  
end do  
!$OMP end parallel do
```

Coding for ccNUMA data locality

Simplest case: explicit initialization

```
integer,parameter :: N=10000000  
double precision A(N), B(N)
```

```
A=0.d0
```



```
!$OMP parallel do  
do i = 1, N  
    B(i) = function ( A(i) )  
end do  
!$OMP end parallel do
```

```
integer,parameter :: N=10000000  
double precision A(N),B(N)
```

```
!$OMP parallel  
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
    A(i)=0.d0
```

```
end do
```

```
!$OMP end do
```

```
...
```

```
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
    B(i) = function ( A(i) )
```

```
end do
```

```
!$OMP end do
```

```
!$OMP end parallel
```



Coding for ccNUMA data locality

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```
integer,parameter :: N=10000000  
allocate(A(N), B(N))
```

```
READ(1000) A
```



```
!$OMP parallel do  
do i = 1, N  
  B(i) = function ( A(i) )  
end do  
!$OMP end parallel do
```

Coding for ccNUMA data locality

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```
integer,parameter :: N=10000000
allocate(A(N), B(N))
```

READ(1000) A



```
!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```
integer,parameter :: N=10000000
allocate(A(N), B(N))
```

```
!$OMP parallel
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
  A(i)=0.d0
```

```
end do
```

```
!$OMP end do
```

```
!$OMP single
```

```
READ(1000) A
```

```
!$OMP end single
```

```
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
  B(i) = function ( A(i) )
```

```
end do
```

```
!$OMP end do
```

```
!$OMP end parallel
```



Coding for Data Locality

- Required condition: OpenMP **loop schedule** of initialization must be the same as in all computational loops
 - Only choice: **static!** Specify **explicitly** on all NUMA-sensitive loops, just to be sure...
 - Imposes some constraints on possible optimizations (e.g. load balancing)
 - Presupposes that all **worksharing loops** with the **same loop length** have the **same thread-chunk mapping**
 - If **dynamic scheduling/tasking** is unavoidable, the problem cannot be solved completely if a team of threads spans more than one LD
 - Static parallel first touch is still a good idea
- How about **global objects**?
 - Initialized before `main()` is called
 - If communication vs. computation is favorable, might consider **properly placed copies** of global data
- C++: Arrays of objects and `std::vector<>` are by default initialized sequentially
 - **STL allocators** provide an elegant solution

Diagnosing bad locality

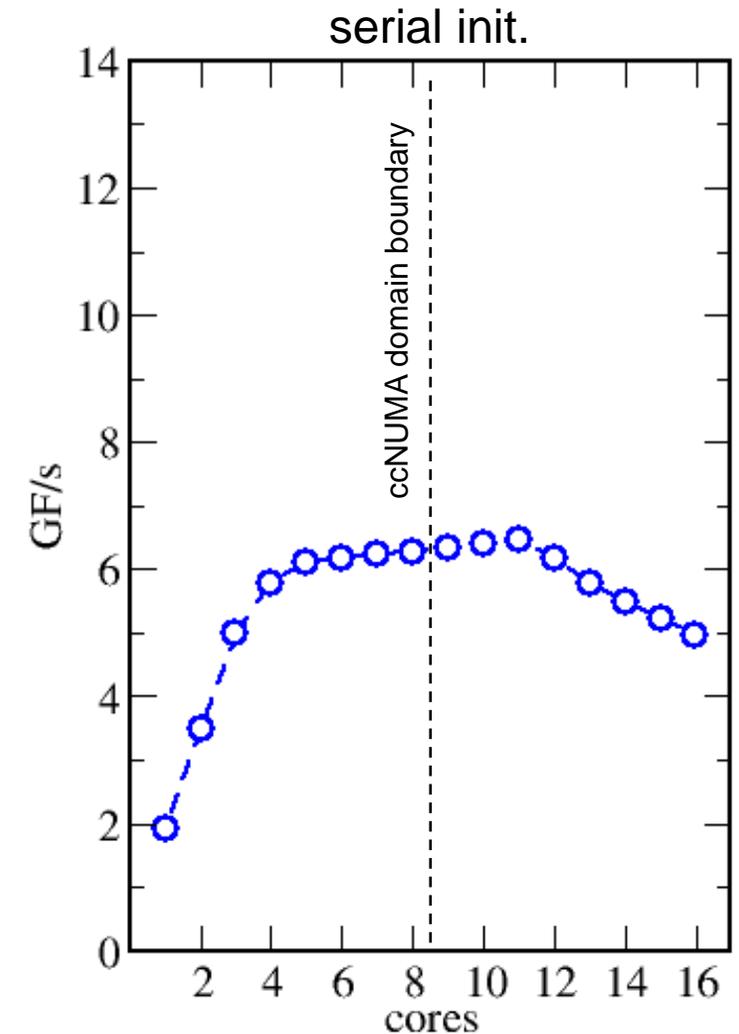
- If your code is cache bound, you might not notice any locality problems
- Otherwise, bad locality **limits scalability** (whenever a ccNUMA node boundary is crossed)
 - **Just an indication, not a proof yet**
- Running with **numactl --interleave** might give you a hint
 - See later
- Consider using **performance counters**
 - **likwid-perfctr** can be used to measure non-local memory accesses
 - Example:

```
$ likwid-perfctr -g NUMA -C M0:0-4@M1:0-4 ./a.out
```

Diagnosing bad locality

- If your code is cache bound, you might not notice any locality problems
- Otherwise, bad locality **limits scalability** (whenever a ccNUMA node boundary is crossed)
 - **Just an indication, not a proof yet**
- Running with **numactl --interleave** might give you a hint
 - See later
- Consider using **performance counters**
 - **likwid-perfctr** can be used to measure non-local memory accesses
 - Example:

```
$ likwid-perfctr -g NUMA -C M0:0-4@M1:0-4 ./a.out
```



Using performance counters for diagnosis

- Intel Ice Lake SP node (running 2x32 threads):
measure inter-socket traffic

```
$ likwid-perfctr -g UPI -C S0:0@S1:0 ./a.out
```

- Output:

Metric	HWThread 0	HWThread 32
Runtime (RDTSC) [s]	12.3681	12.3681
Runtime unhalting [s]	12.1108	8.2227
Clock [MHz]	3281.3537	3103.6518
CPI	5.4670	35.5873
Received data bandwidth [MByte/s]	22127.2106	21358.7412
Received data volume [GByte]	273.6708	264.1663
Sent data bandwidth [MByte/s]	21358.7391	22127.2191
Sent data volume [GByte]	264.1663	273.6709
Total data bandwidth [MByte/s]	43485.9496	43485.9603
Total data volume [GByte]	537.8370	537.8372

Caveat: NUMA metrics vary strongly between CPU models

Using performance counters for diagnosis

- Intel Ice Lake SP node (running 2x32 threads):
measure inter-socket traffic

```
$ likwid-perfctr -g UPI -C S0:0@S1:0 ./a.out
```

- Output:

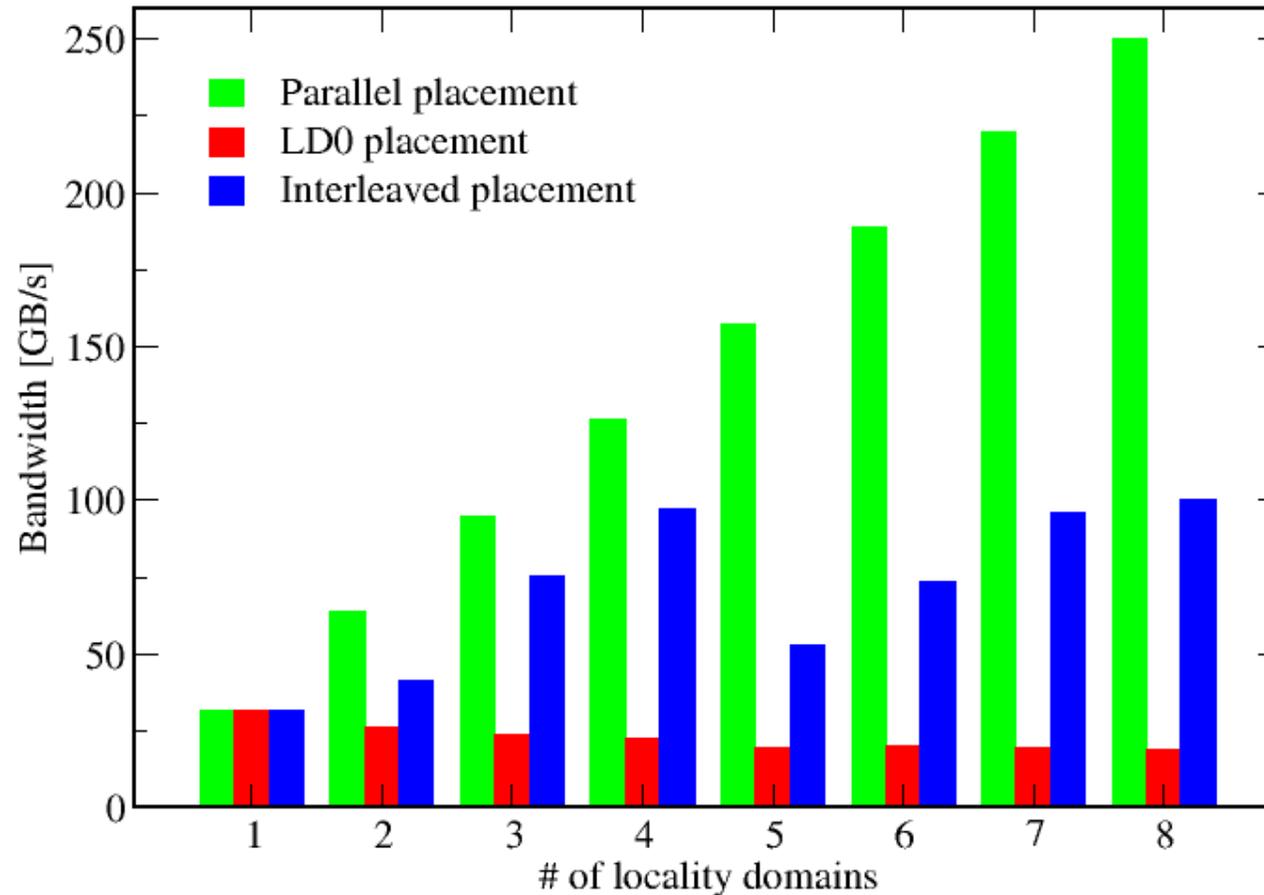
Metric	HWThread 0	HWThread 32
Runtime (RDTSC) [s]	12.3681	12.3681
Runtime unhalting [s]	12.1108	8.2227
Clock [MHz]	3281.3537	3103.6518
CPI	5.4670	35.5873
Received data bandwidth [MByte/s]	22127.2106	21358.7412
Received data volume [GByte]	273.6708	264.1663
Sent data bandwidth [MByte/s]	21358.7391	22127.2191
Sent data volume [GByte]	264.1663	273.6709
Total data bandwidth [MByte/s]	43485.9496	43485.9603
Total data volume [GByte]	537.8370	537.8372

Caveat: NUMA metrics vary strongly between CPU models

About half of the overall memory traffic is caused by the remote domain!

OpenMP STREAM triad on a dual AMD Epyc 7451 (“Naples”) (6 cores per LD)

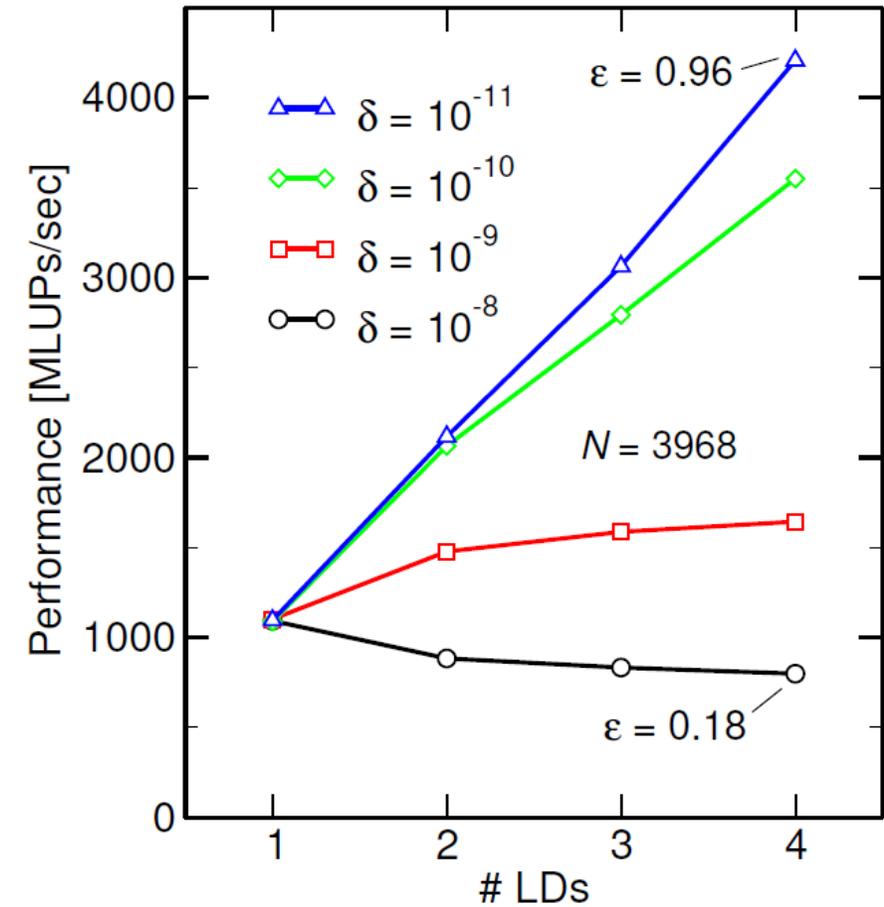
1. **Parallel init:** Correct parallel initialization
2. **LD0:** Force data into LD0 via `numactl -m 0`
3. **Interleaved:** `numactl --interleave <LD range>`



A weird observation

optional

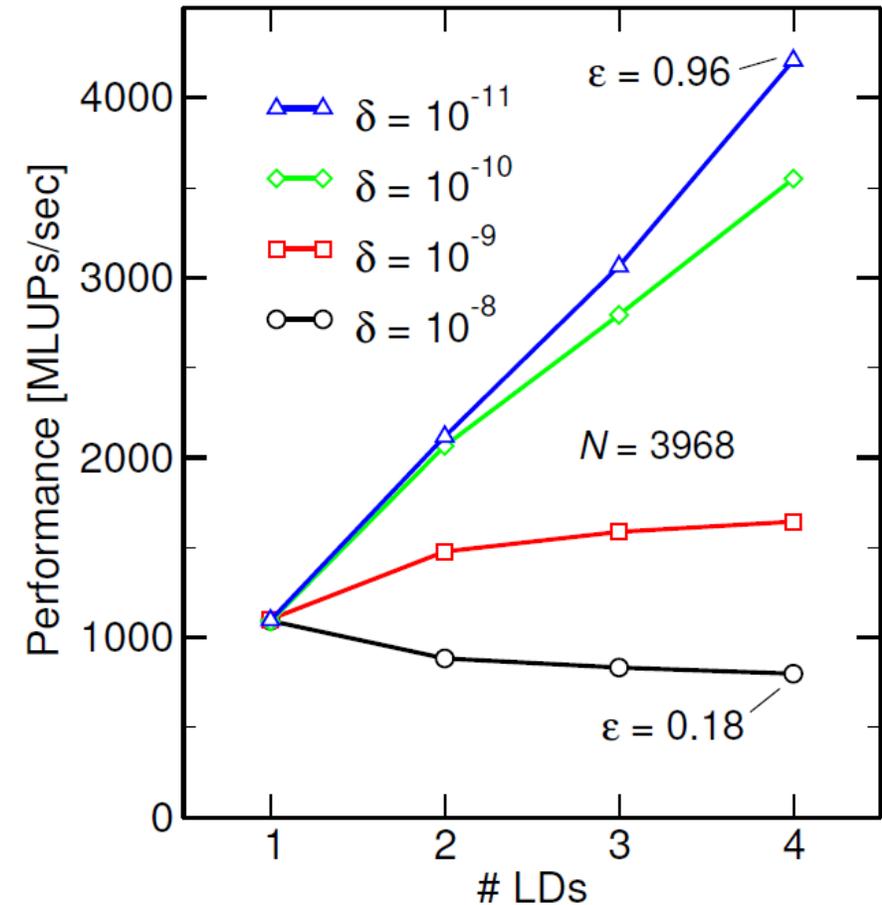
- Experiment: **memory-bound** Jacobi solver with sequential data initialization
 - **No parallel data placement** at all!
 - Expect no scaling across LDs
- Convergence threshold δ determines the runtime
 - The smaller δ , the longer the run



A weird observation

optional

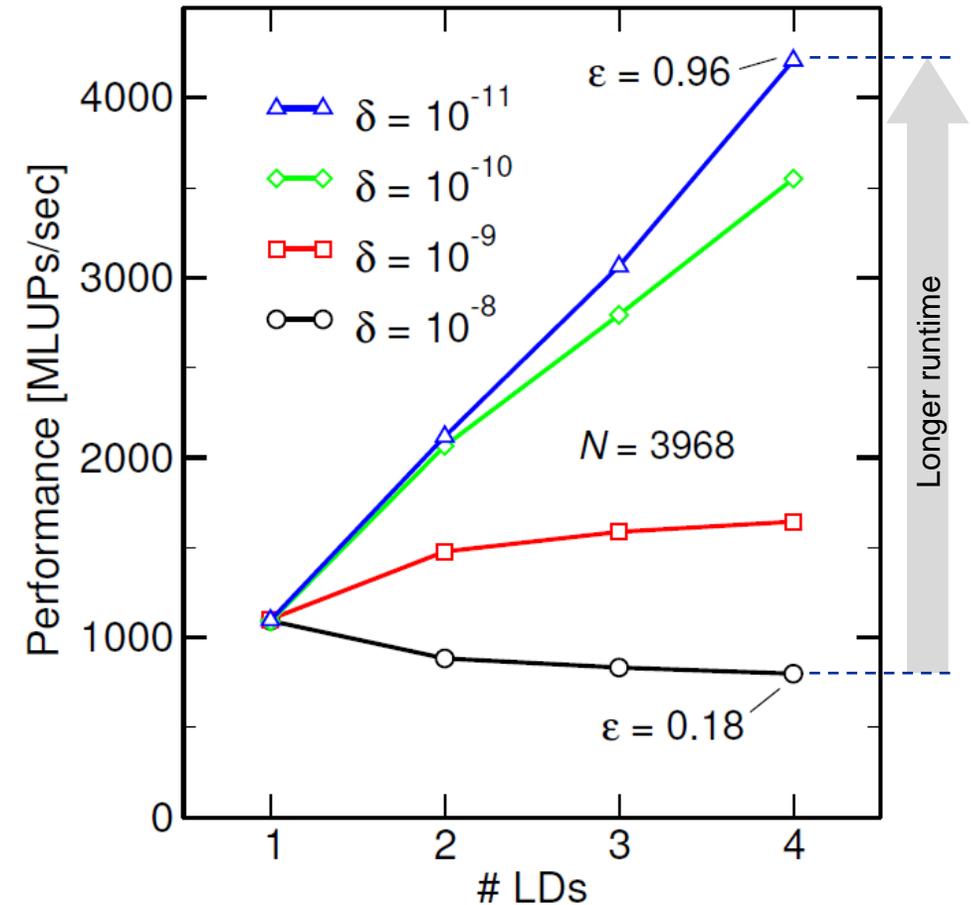
- Experiment: **memory-bound** Jacobi solver with sequential data initialization
 - **No parallel data placement** at all!
 - Expect no scaling across LDs
- Convergence threshold δ determines the runtime
 - The smaller δ , the longer the run
- Observation
 - **No scaling** across LDs for **large δ** (runtime 0.5 s)
 - **Scaling** gets **better** with **smaller δ** up to almost perfect efficiency ε (runtime 91 s)



A weird observation

optional

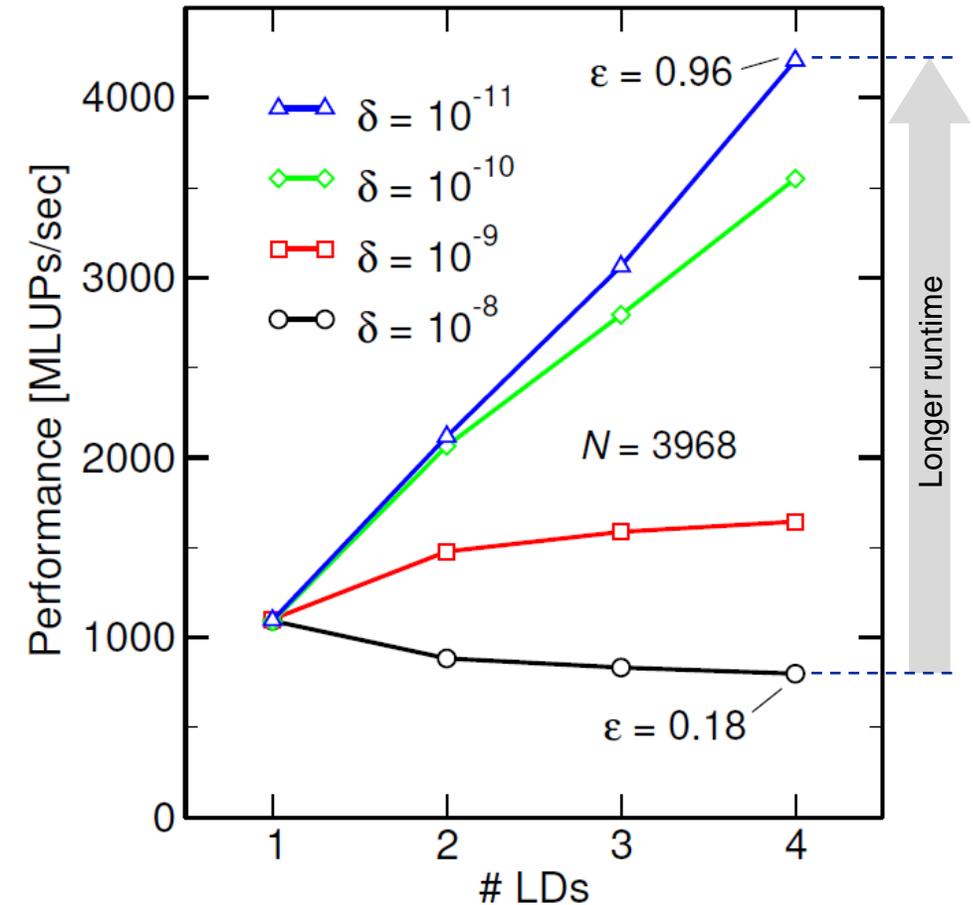
- Experiment: **memory-bound** Jacobi solver with sequential data initialization
 - **No parallel data placement** at all!
 - Expect no scaling across LDs
- Convergence threshold δ determines the runtime
 - The smaller δ , the longer the run
- Observation
 - **No scaling** across LDs for large δ (runtime 0.5 s)
 - **Scaling** gets better with smaller δ up to almost perfect efficiency ε (runtime 91 s)



A weird observation

optional

- Experiment: **memory-bound** Jacobi solver with sequential data initialization
 - **No parallel data placement** at all!
 - Expect no scaling across LDs
- Convergence threshold δ determines the runtime
 - The smaller δ , the longer the run
- Observation
 - **No scaling** across LDs for large δ (runtime 0.5 s)
 - **Scaling** gets better with smaller δ up to almost perfect efficiency ε (runtime 91 s)
- **Conclusion**
 - Something seems to “**heal**” the bad **access locality** on a time scale of tens of seconds



Riddle solved: NUMA balancing

optional

- Linux kernel supports automatic page migration

```
$ cat /proc/sys/kernel/numa_balancing
0
$ echo 1 > /proc/sys/kernel/numa_balancing # activate
```

- Active on all current Linux distributions, some performance impact for single core execution
- Parameters control aggressiveness

```
$ ll /proc/sys/kernel/numa*
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_delay_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_max_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_min_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_size_mb
```

- Default behavior is “take it slow”
- Do not rely on it! Parallel first touch is still a good idea!

Summary on ccNUMA issues

- Identify the problem
 - Is ccNUMA an issue in your code?
 - Simple test: run with `numactl --interleave`
 - Consider performance counters if available

Summary on ccNUMA issues

- Identify the problem
 - Is ccNUMA an issue in your code?
 - Simple test: run with `numactl --interleave`
 - Consider performance counters if available
- Apply first-touch placement in initialization loops
 - Consider loop lengths and static scheduling
 - C++ and global/static objects may require special care

Summary on ccNUMA issues

- Identify the problem
 - Is ccNUMA an issue in your code?
 - Simple test: run with `numactl --interleave`
 - Consider performance counters if available
- Apply first-touch placement in initialization loops
 - Consider loop lengths and static scheduling
 - C++ and global/static objects may require special care
- NUMA balancing is active on many Linux systems today
 - Automatic page migration
 - Slow process, may take many seconds (configurable)
 - Not a silver bullet
 - Still a good idea to do parallel first touch

Summary on ccNUMA issues

- Identify the problem
 - Is ccNUMA an issue in your code?
 - Simple test: run with `numactl --interleave`
 - Consider performance counters if available
- Apply first-touch placement in initialization loops
 - Consider loop lengths and static scheduling
 - C++ and global/static objects may require special care
- NUMA balancing is active on many Linux systems today
 - Automatic page migration
 - Slow process, may take many seconds (configurable)
 - Not a silver bullet
 - Still a good idea to do parallel first touch
- If dynamic scheduling cannot be avoided
 - Consider round-robin placement as a quick (but non-ideal) fix
 - OpenMP 5.0 has some data affinity support

Tutorial conclusion

- Know your system (node) **architecture**
- Enforce **affinity**
- **Back-of-the-envelope models** are extremely useful
- **Modeling is not always predictive**
- **Bottleneck awareness rules**
- Performance is not about tools. **Use your brain!**

Visit us at Booth
#1311
(LRZ)