

NHR  **FAU** at **ISC 2023**

MAY 21 – 25
#ISC23



A Beginner's Introduction to Node-Level Computer Architecture and Performance

Dr. Georg Hager

Erlangen National High Performance Computing Center
(NHR@FAU)

go-nhr.de/NLCA23

Agenda

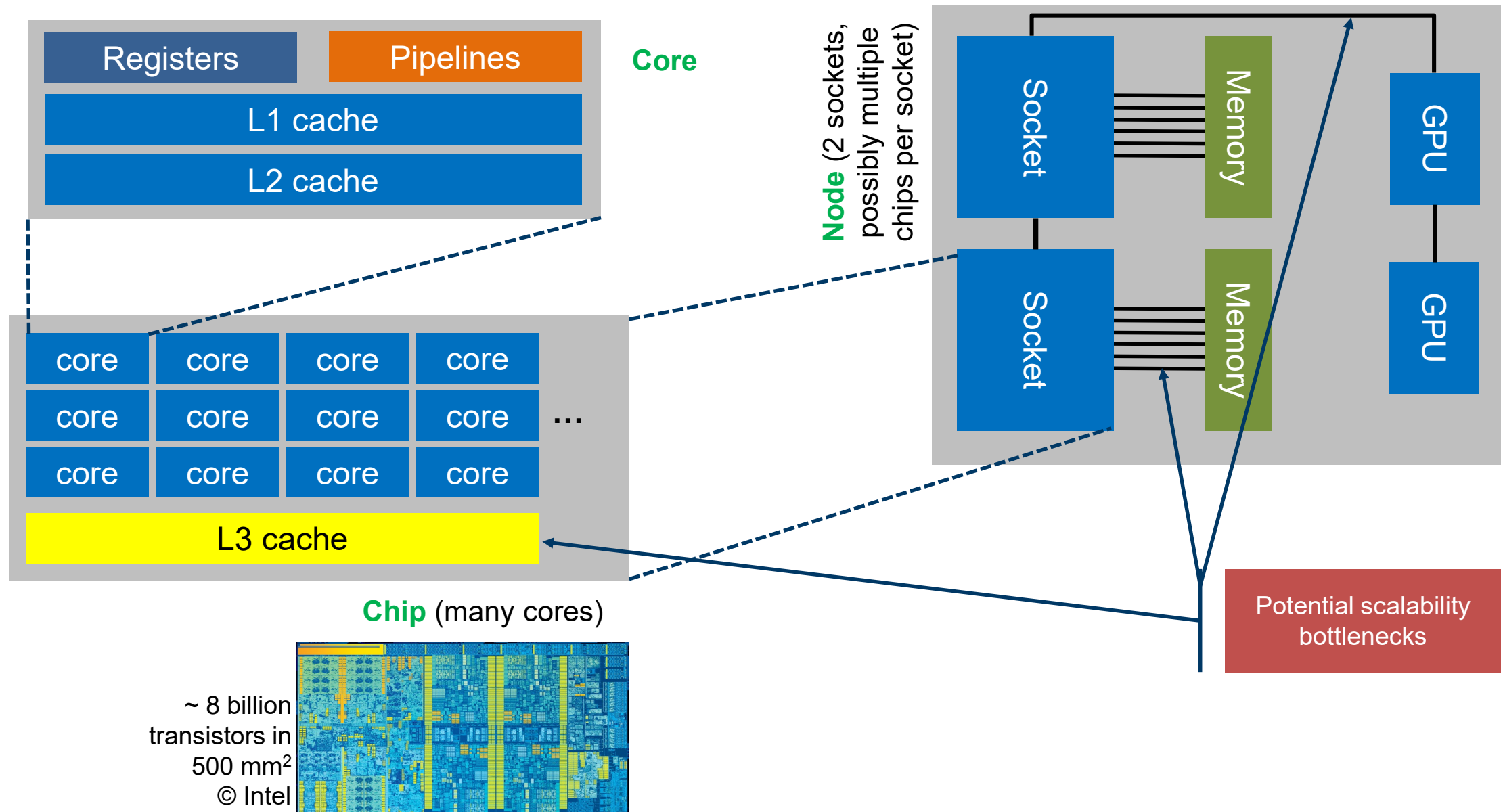
- **Basic CPU/GPU and node architecture**
 - Core: Pipelines, SIMD, out-of-order processing, SMT
 - Cache hierarchy
 - Memory interface
 - Basic performance phenomenology and bottlenecks
- **Hardware-software interaction**
 - The naive Roofline model
 - Examples: sum reduction, stencils
- **Common-sense code analysis**
 - Using hardware performance counters
 - Characterizing code with hardware counters

Modern computer architecture

An introduction for software developers

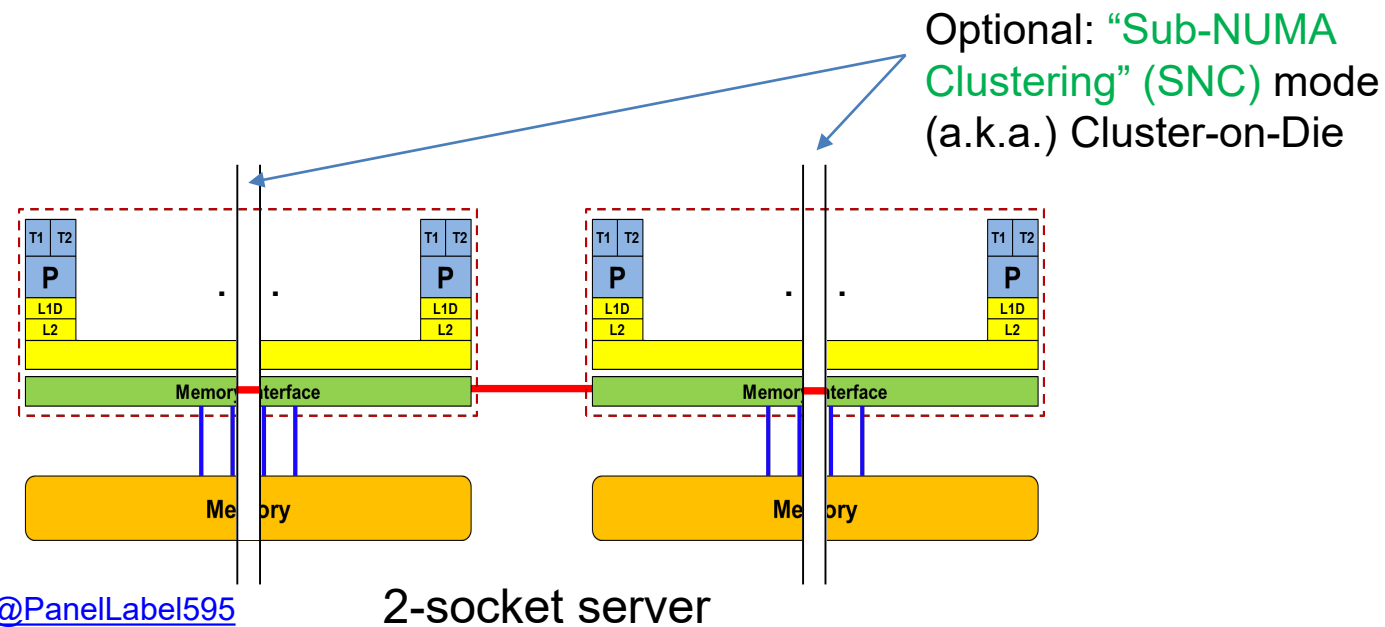


Node topology of HPC systems



Multi-core today: Intel Xeon Ice Lake (2021)

- Xeon “Ice Lake SP” (Platinum/Gold/Silver/Bronze):
Up to 40 cores running at 2+ GHz (+ “Turbo Mode” 3.7 GHz),
- Simultaneous Multithreading
→ reports as 80-way chip
- ~15 Billion Transistors / ~10 nm / up to 270 W
- Die size: up to ~600 mm²
- Clock frequency:
flexible 😊



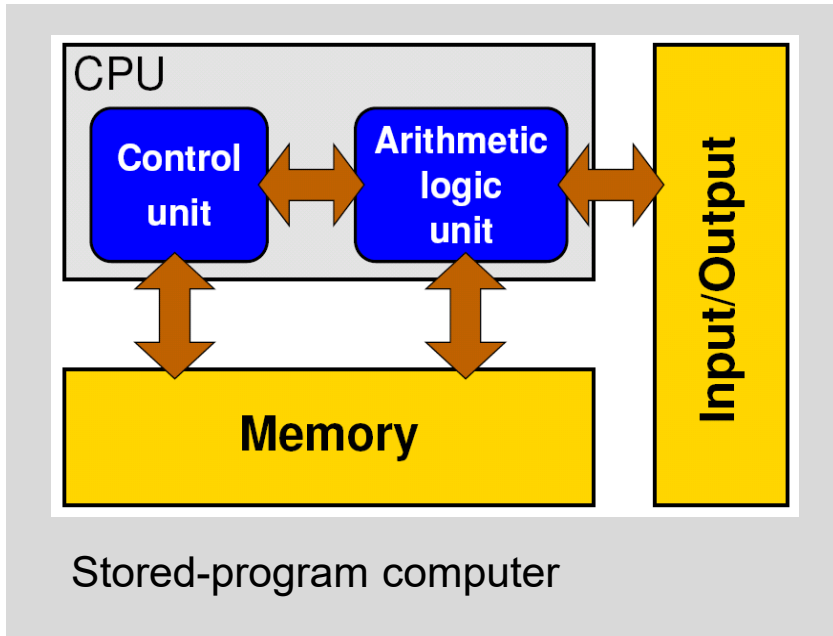
<https://ark.intel.com/content/www/us/en/ark.html#@PanelLabel595>

In-core features

Pipelining, Superscalarity, SIMD, SMT

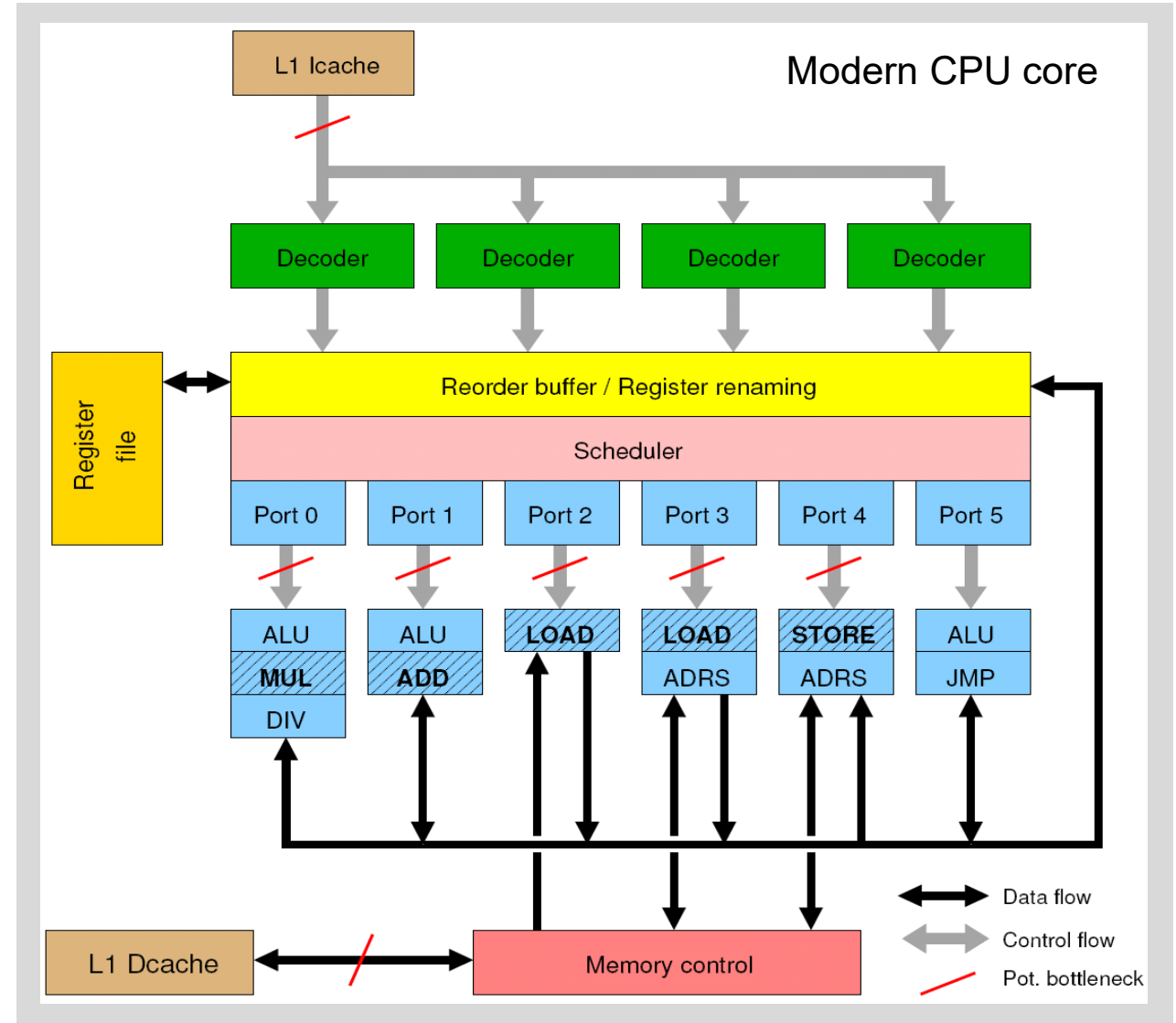


General-purpose CPU microprocessor core



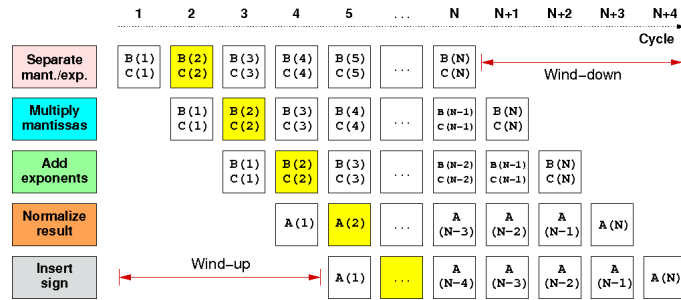
- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks

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

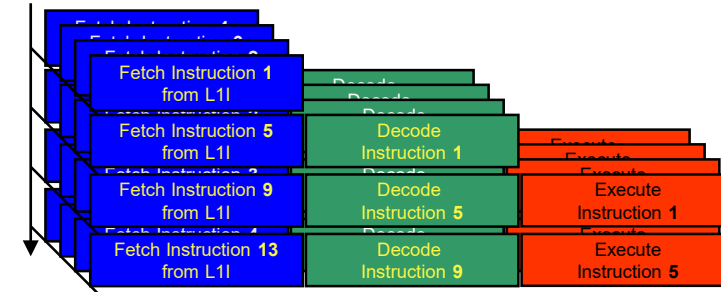


Important in-core features

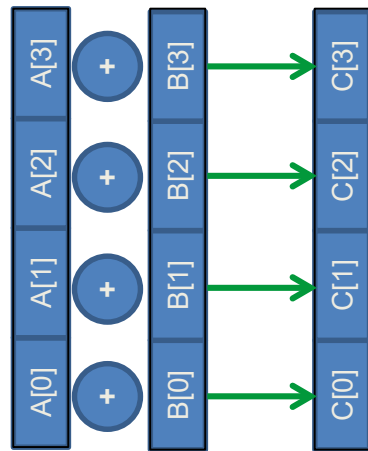
Pipelining:
Instruction execution in multiple steps



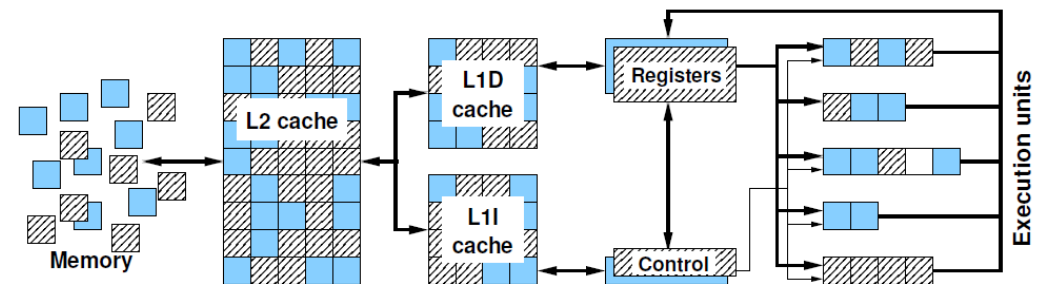
Superscalarity:
Multiple instructions per cycle



Single Instruction Multiple Data:
Multiple operations per instruction



Simultaneous Multi-Threading:
Multiple instruction sequences in parallel

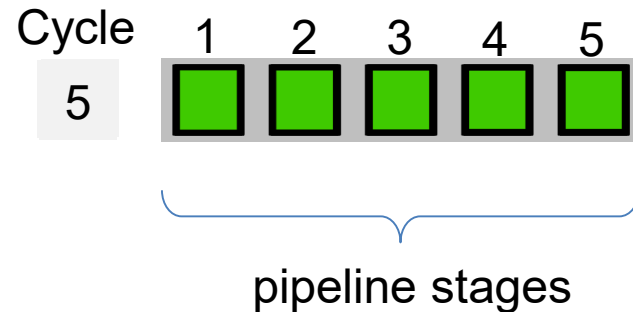


Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining



Example: A single instruction takes 5 cycles (latency)



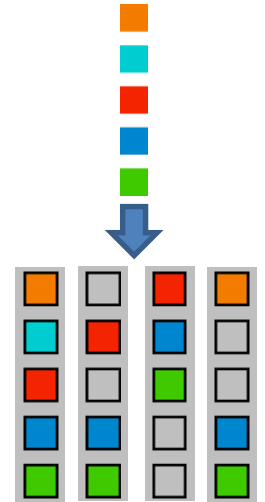
Throughput:

1 instruction per cycle after pipeline is full

→ Speedup by factor 5

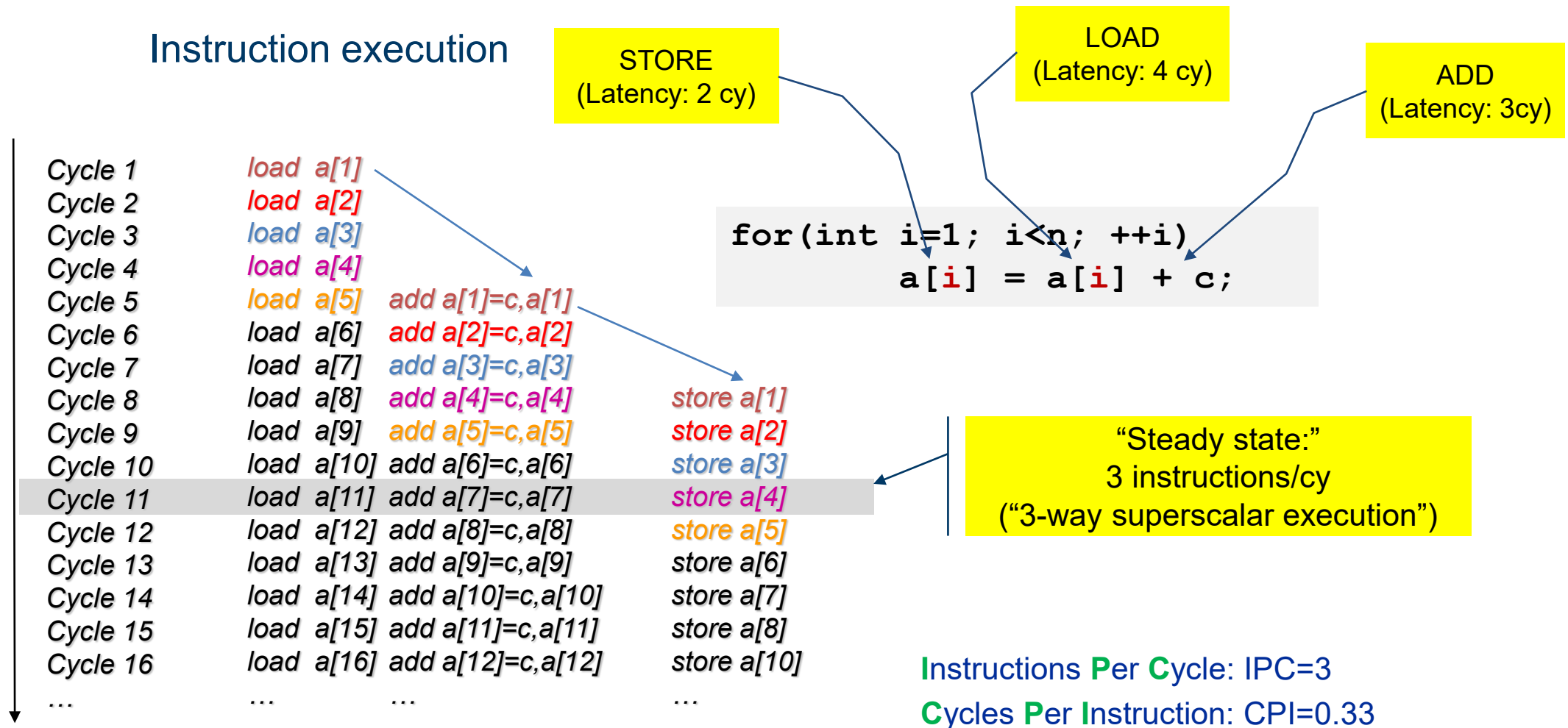
Superscalar execution

4-way superscalar:



- Massive boost in instruction throughput
- Instructions can be reordered on the fly
- Dependencies are obeyed

Superscalar out-of-order execution and steady state



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

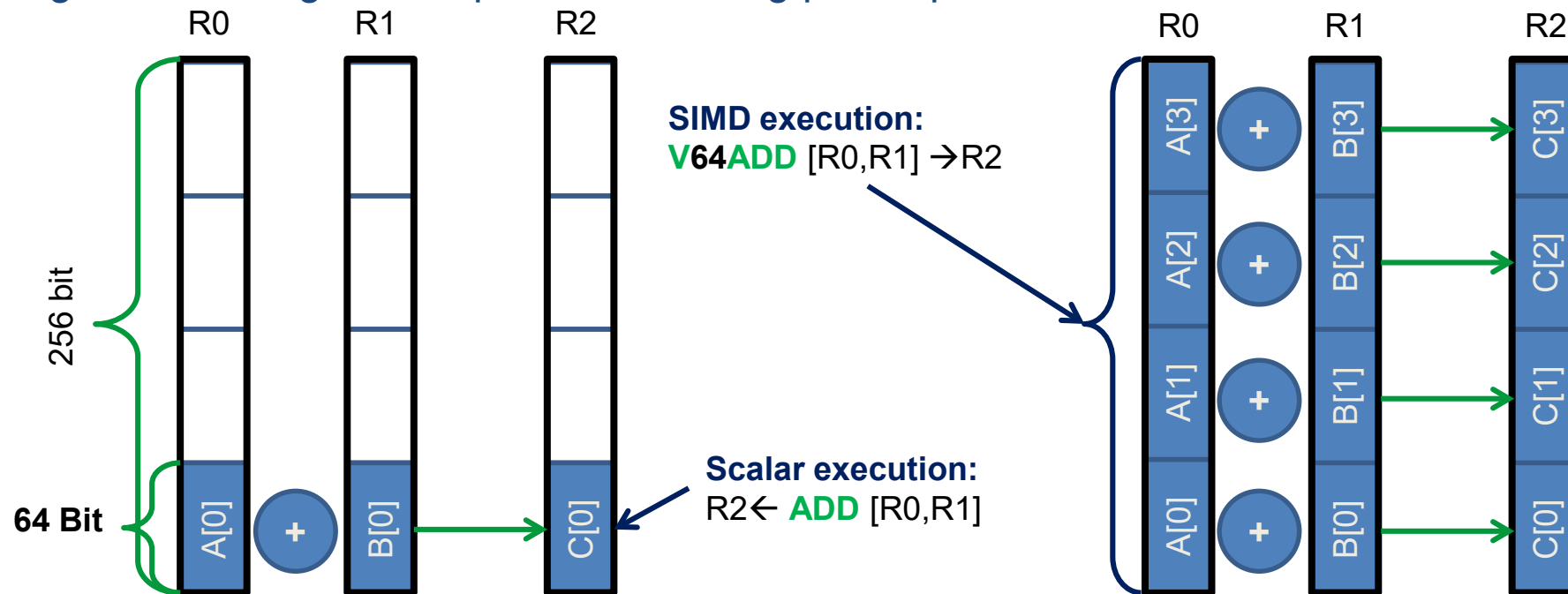
Quiz time: OoO Execution

<https://moodle.nhr.fau.de/mod/h5pactivity/view.php?id=2360>



SIMD processing

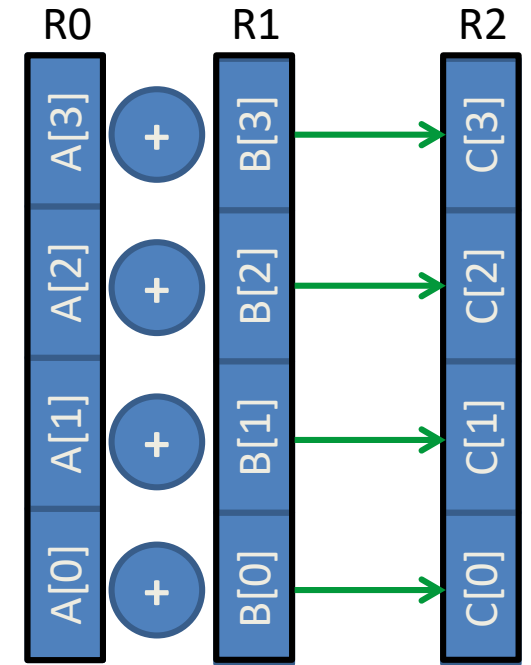
- **Single Instruction Multiple Data (SIMD)** operations allow the execution of the **same operation** on “wide” registers from a **single instruction**
- x86 SIMD instruction sets:
 - SSE: register width = 128 Bit → 2 double precision floating point operands
 - AVX: register width = 256 Bit → 4 double precision floating point operands
 - AVX-512: ... you guessed it!
- Adding two registers holding double precision floating point operands:



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

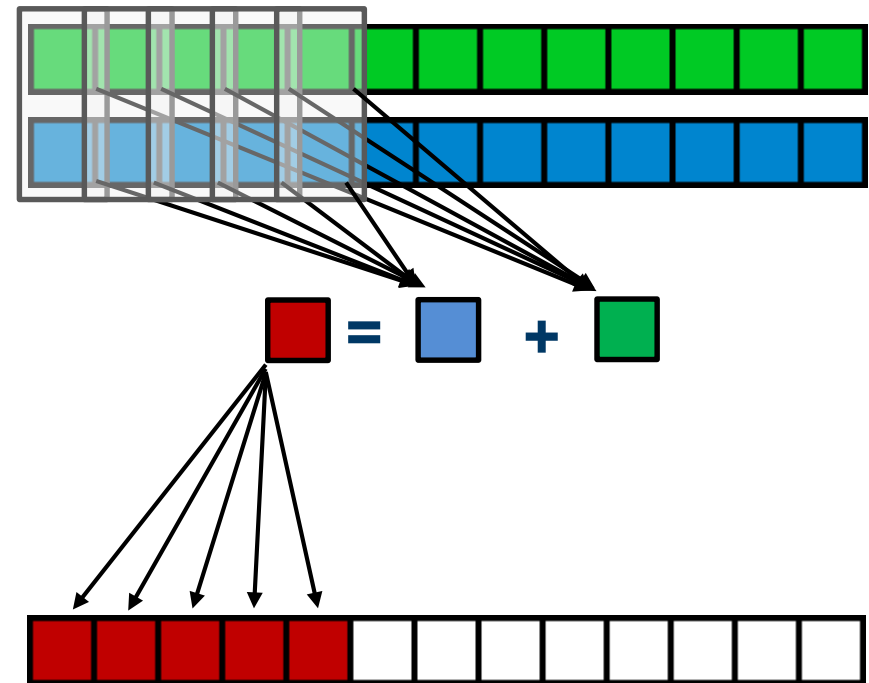
```
for (int j=0; j<size; j++) {  
    A[j] = B[j] + C[j];  
}
```

Register width

- 1 operand



Scalar execution



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)

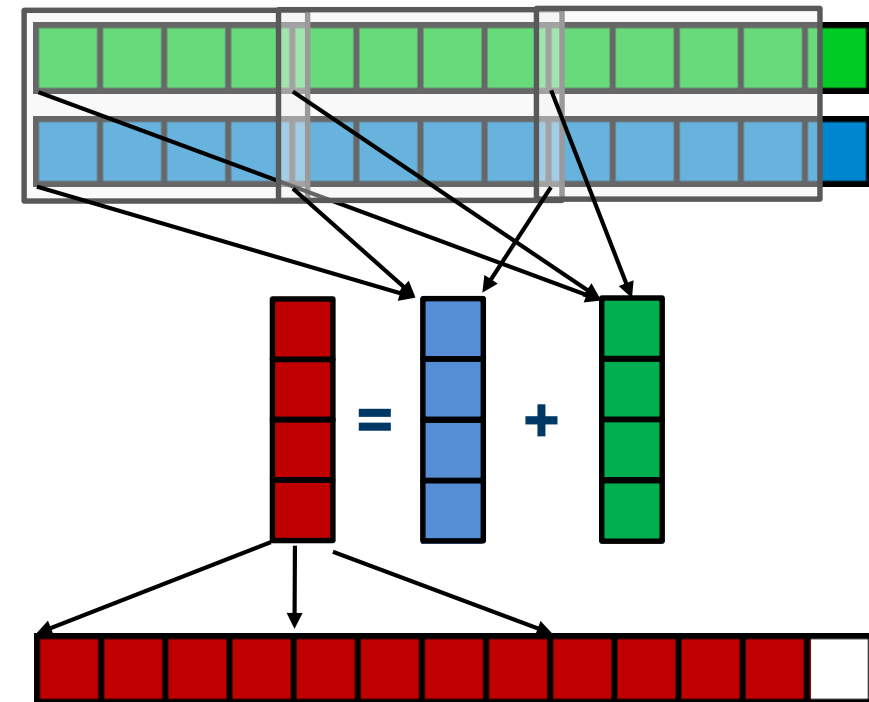


- 8 operands (AVX512)

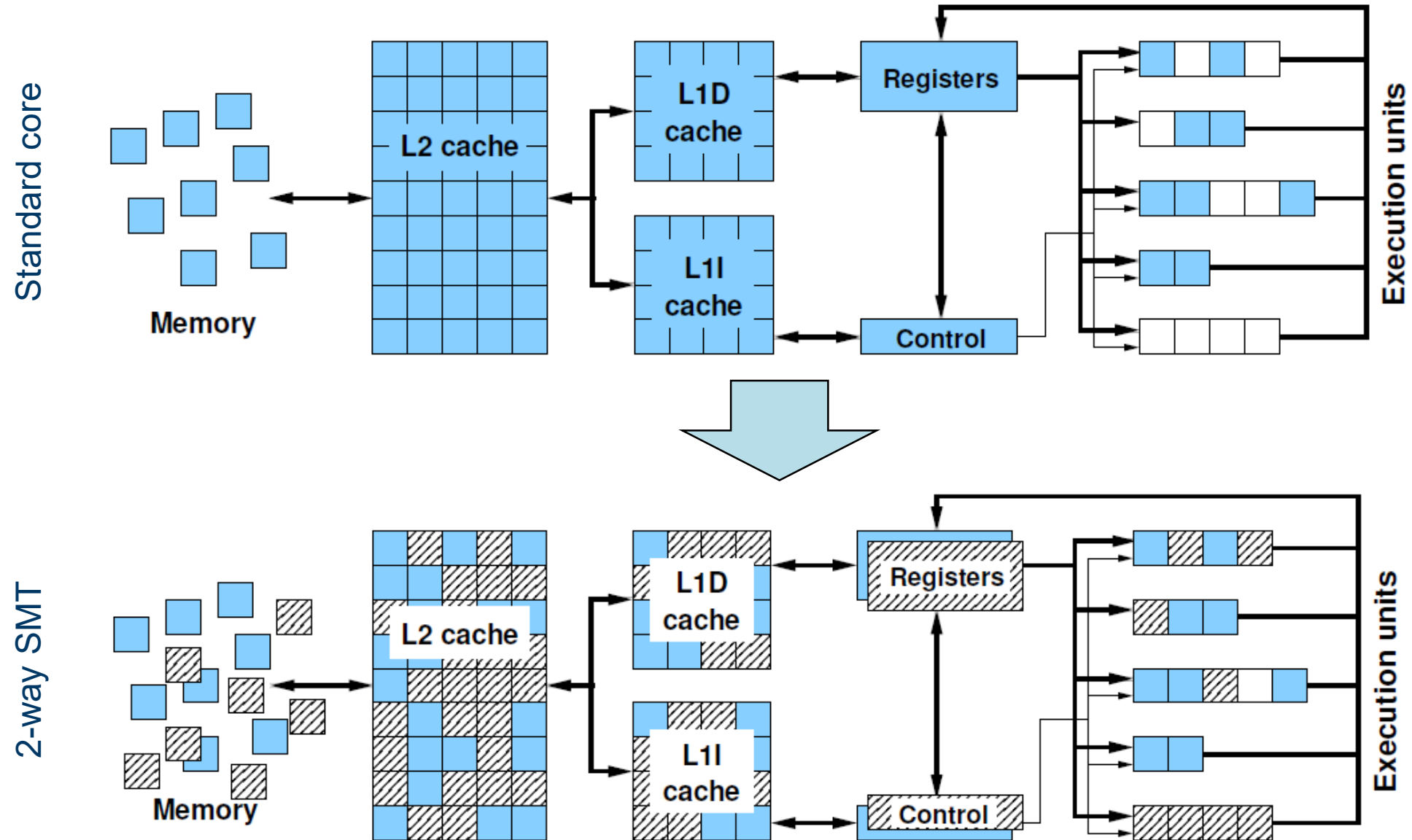


Best code requires vectorized LOADs, STOREs, and arithmetic!

SIMD execution



Simultaneous multi-threading (SMT)

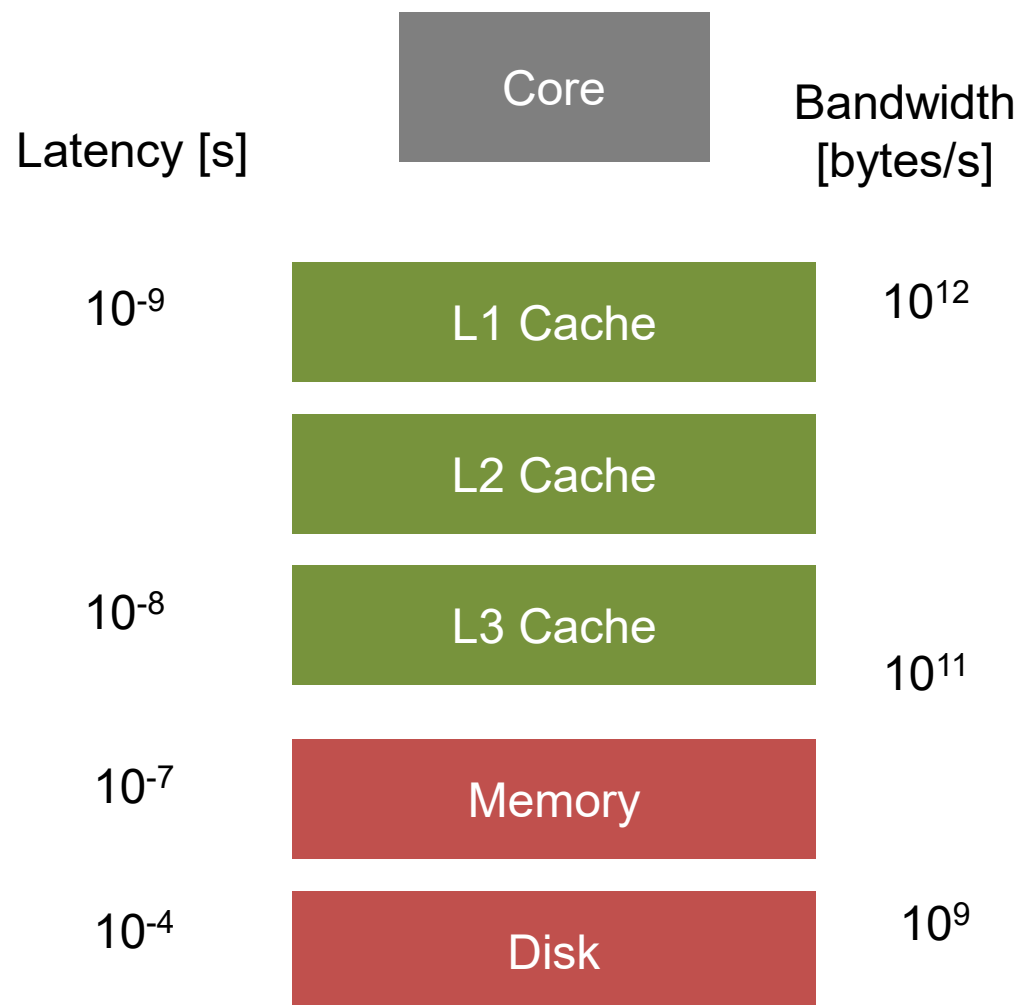


Memory Hierarchy and Affinity



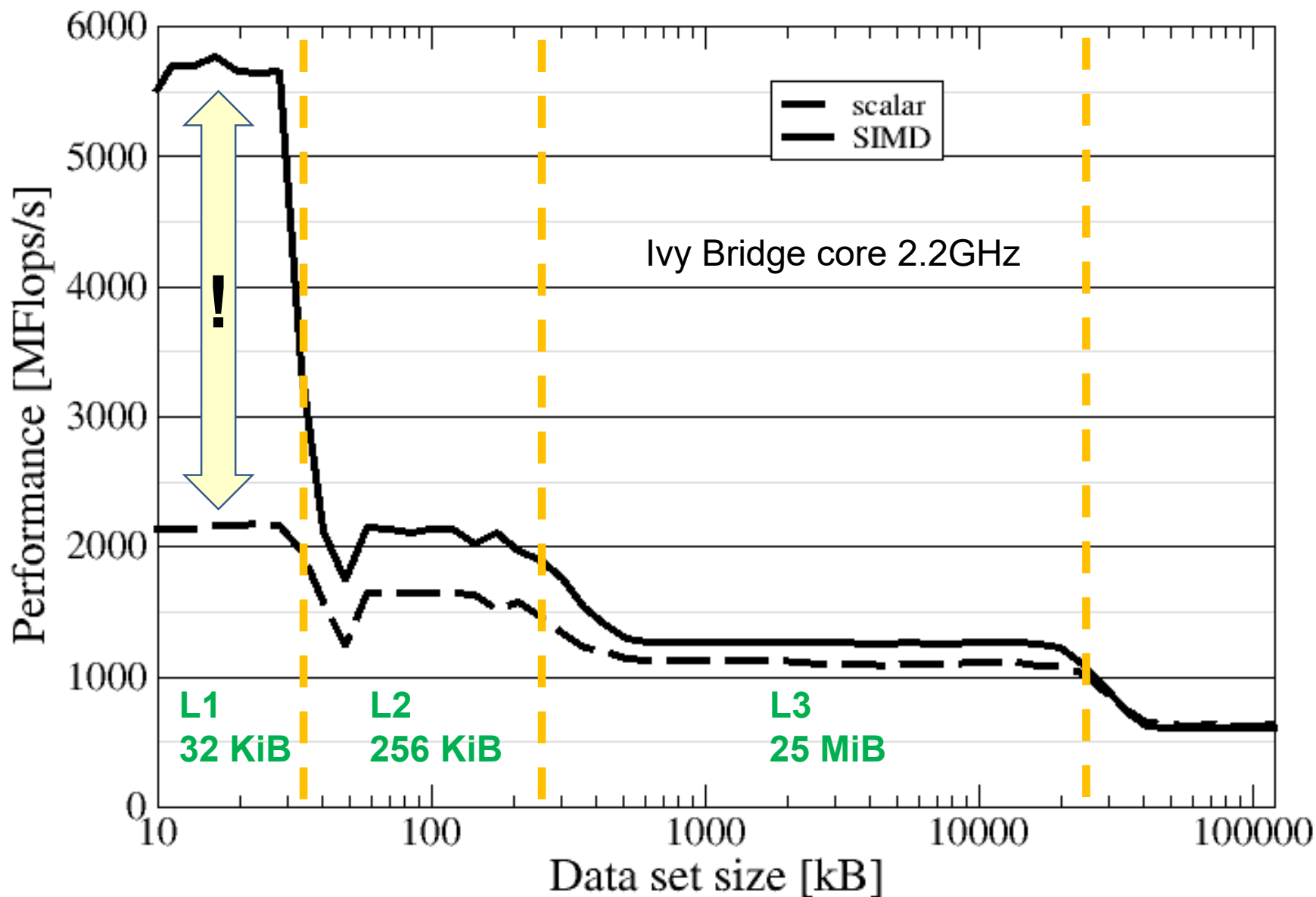
Memory hierarchy

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**

Getting data from far away

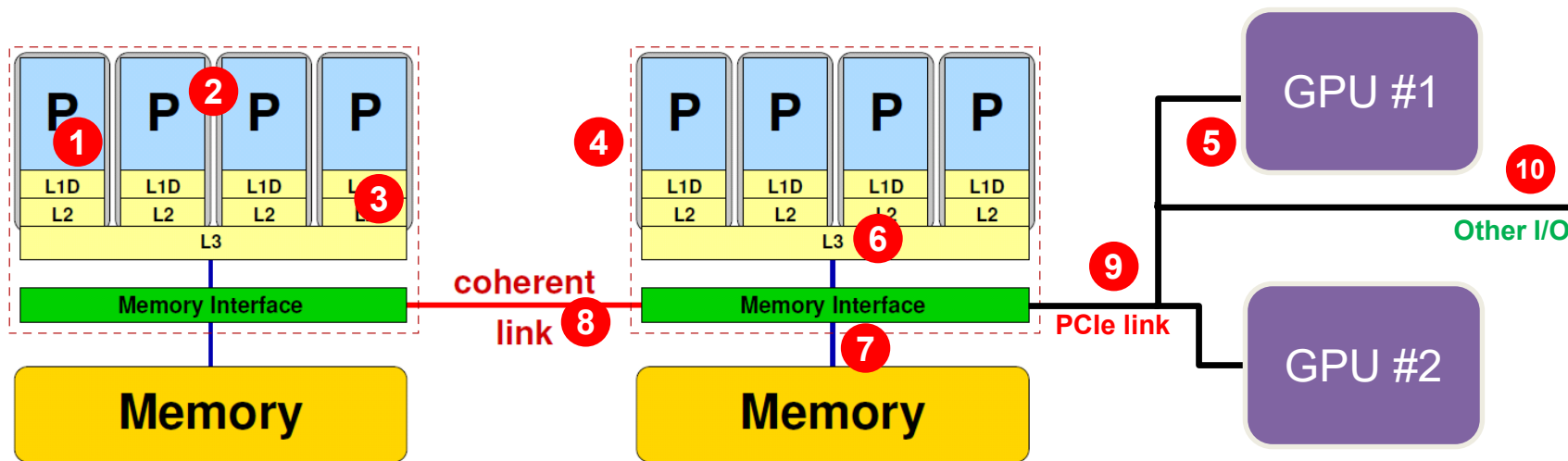


$$A(:) = B(:) + C(:) * D(:)$$

Varying loop length,
repeat many times

Parallelism and bottlenecks in a modern compute node

Parallel and shared resources within a shared-memory node



Parallel resources:

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

Shared resources:

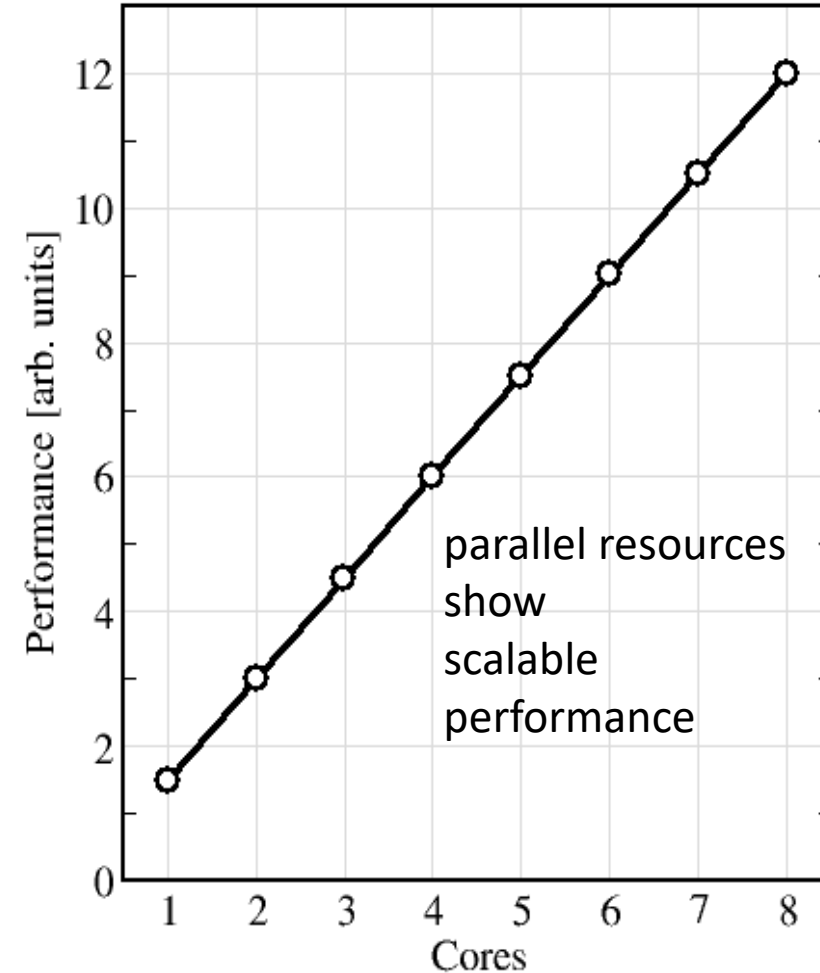
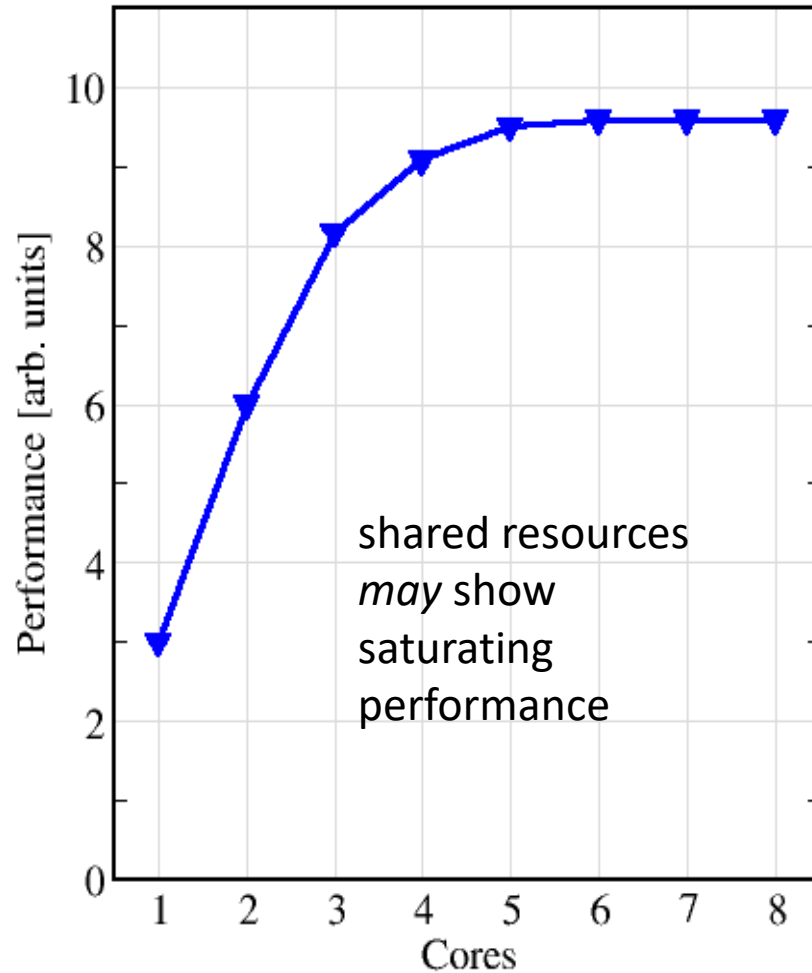
- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

How does your application react to all of those details?

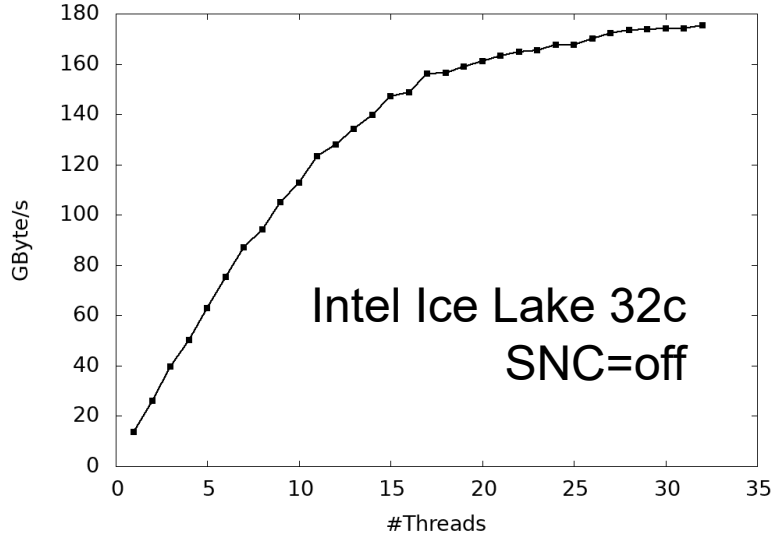
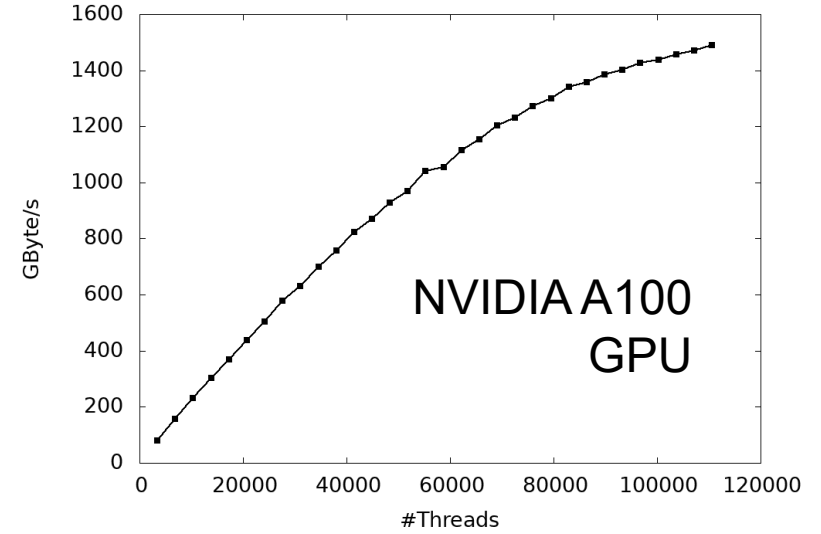
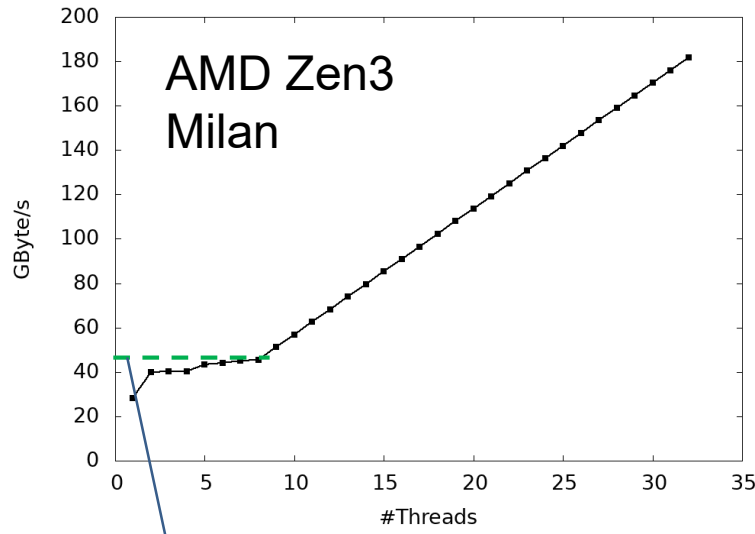
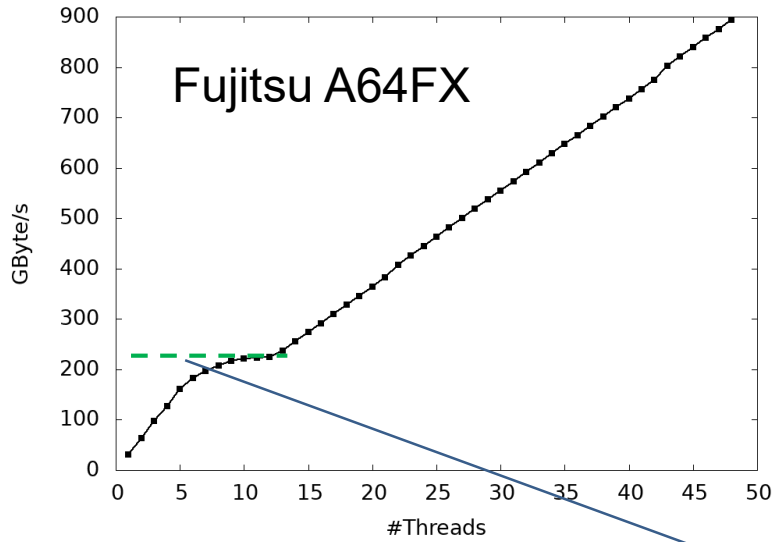
Scalable and saturating behavior

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

One of the most important performance signatures

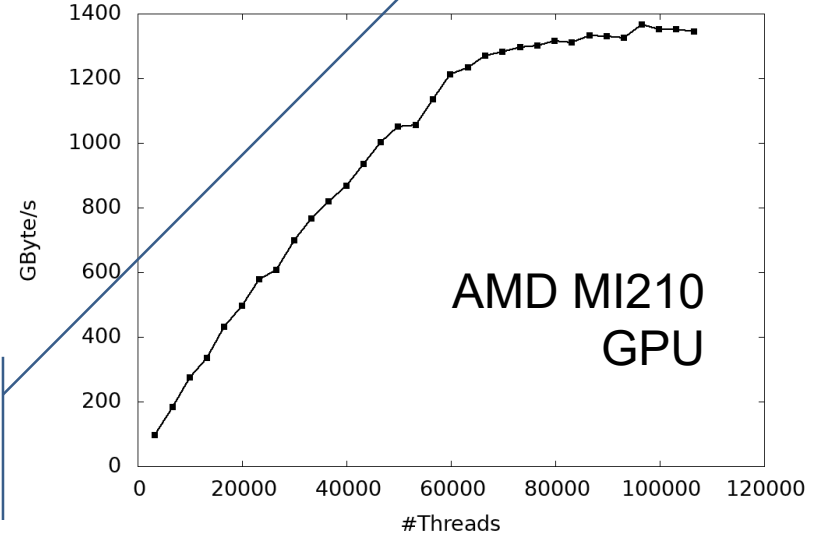


Memory bandwidth saturation (read-only)



Bandwidth saturation on 1st ccNUMA domain

Massive thread parallelism needed on GPUs to saturate



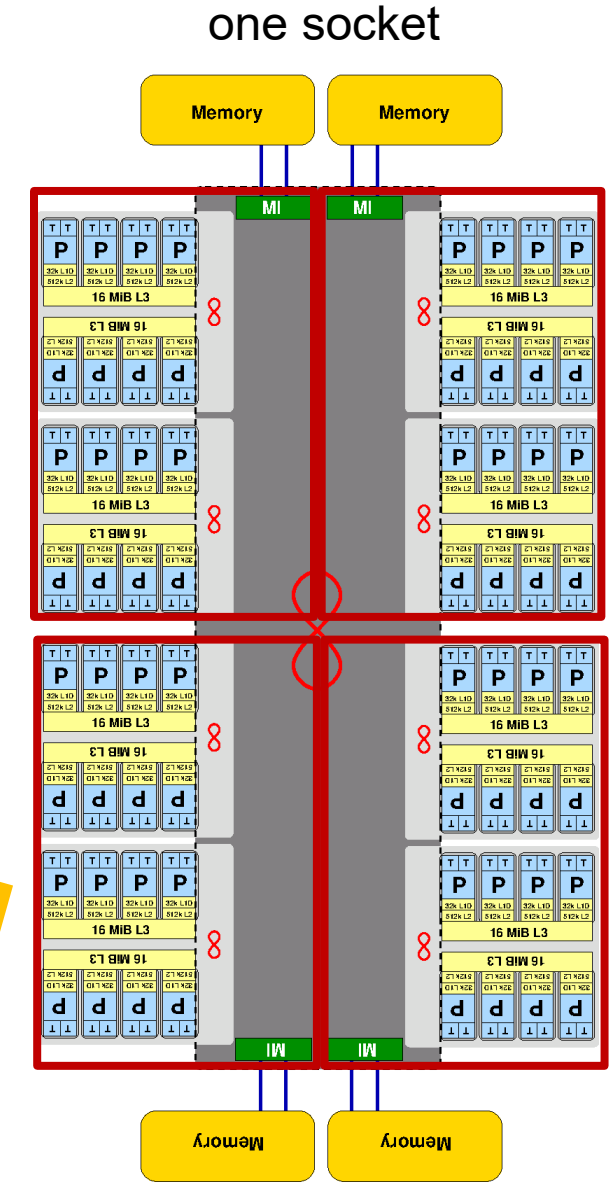
Putting the cores & caches together

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**

Tools for topology exploration:

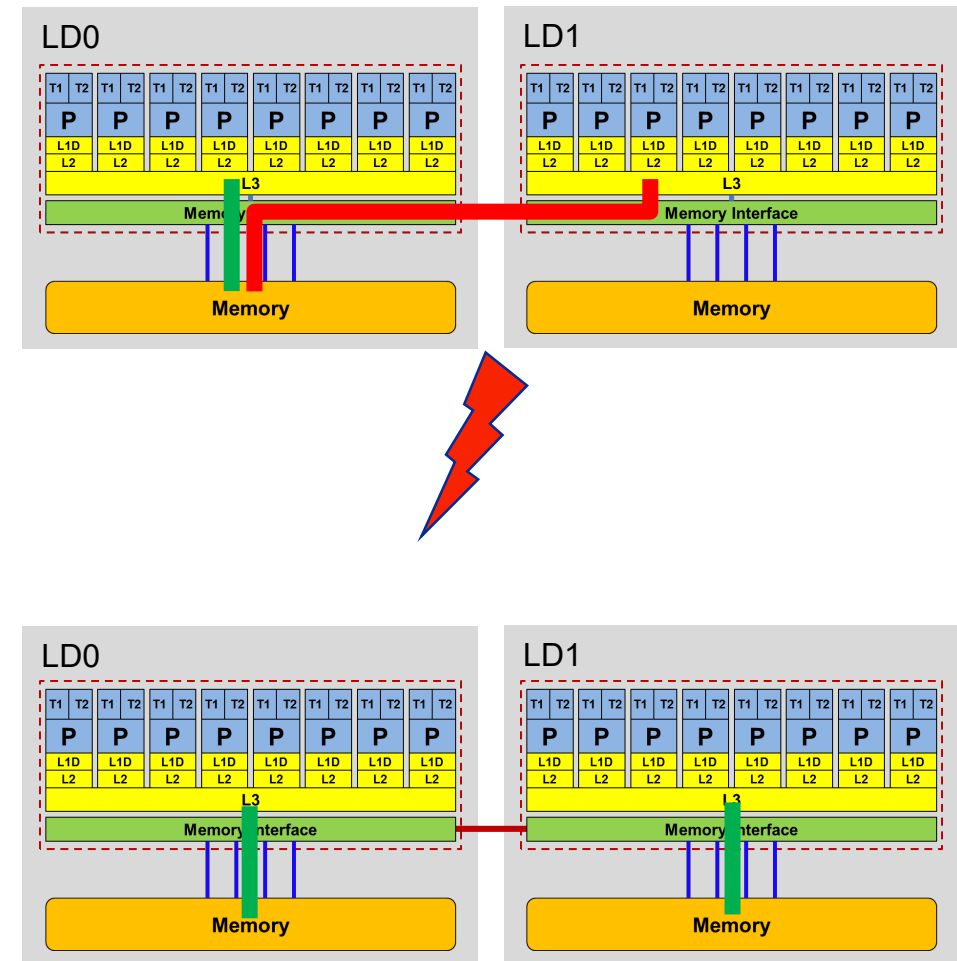
- likwid-topology
- lstopo (hwloc)
- cpuinfo (Intel)
- ...



ccNUMA – cache-coherent Non-Uniform Memory Architecture

- ccNUMA:
 - Whole memory is **transparently accessible** by all cores
 - 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)



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
```

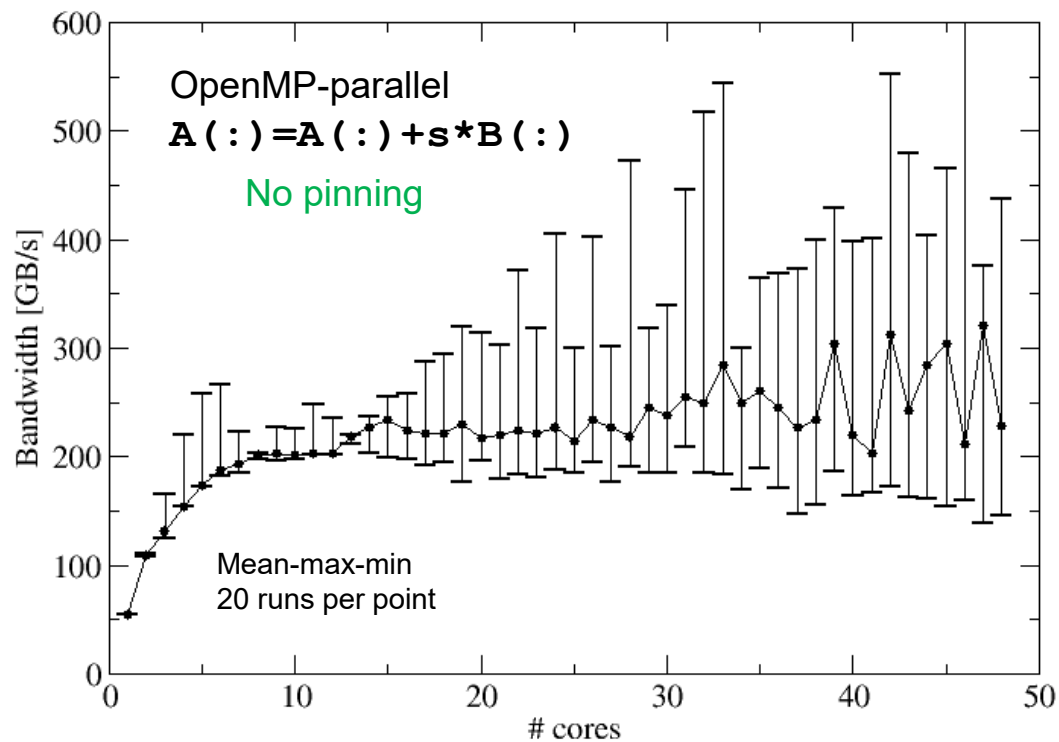
```
!$OMP end do
```

```
!$OMP end parallel
```



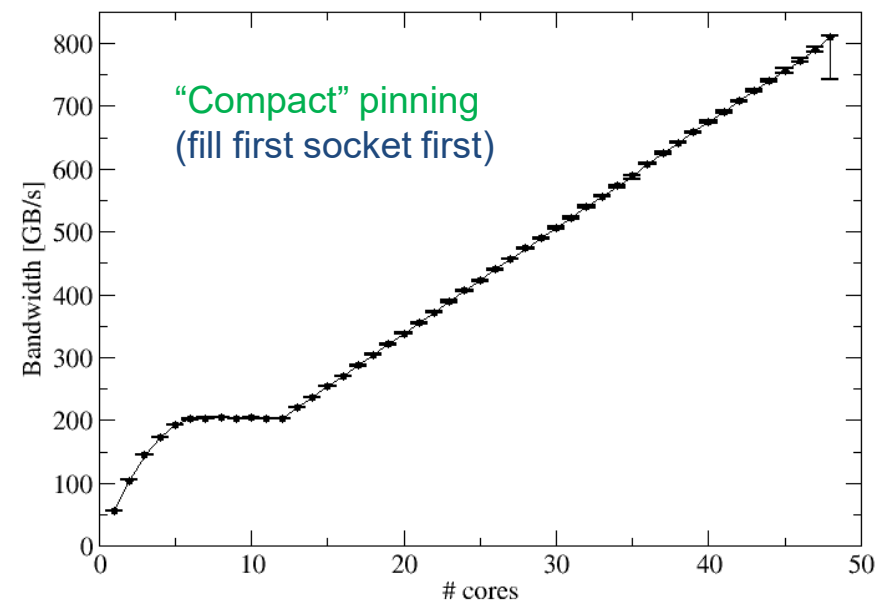
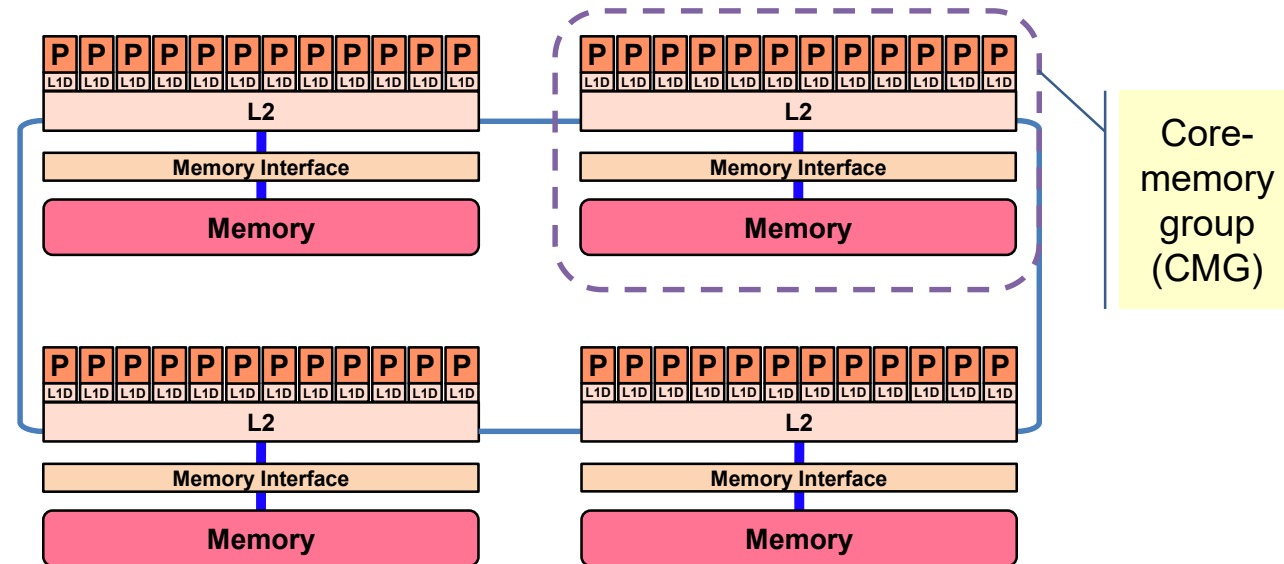
DAXPY test on A64FX

Anarchy vs. thread pinning



There are several reasons for caring about affinity:

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



Means of enforcing thread or process affinity

- **OpenMP**
 - Compiler-specific facilities
 - OpenMP standard (`OMP_PROC_BIND`, `OMP_PLACES`)
 - `likwid-pin` from the LIKWID tool suite (<https://github.com/RRZE-HPC/likwid>)
- **MPI**
 - Implementation-specific facilities
 - SLURM resource manager options to `srun`
 - `likwid-mpirun`
- **Hybrid MPI+OpenMP**
 - `likwid-mpirun`
 - SLURM (but inferior since individual threads are not pinned)
 - Hoping for the best and that MPI and OpenMP implementations work together

GPGPU accelerators

NVIDIA “Ampere” A100

vs.

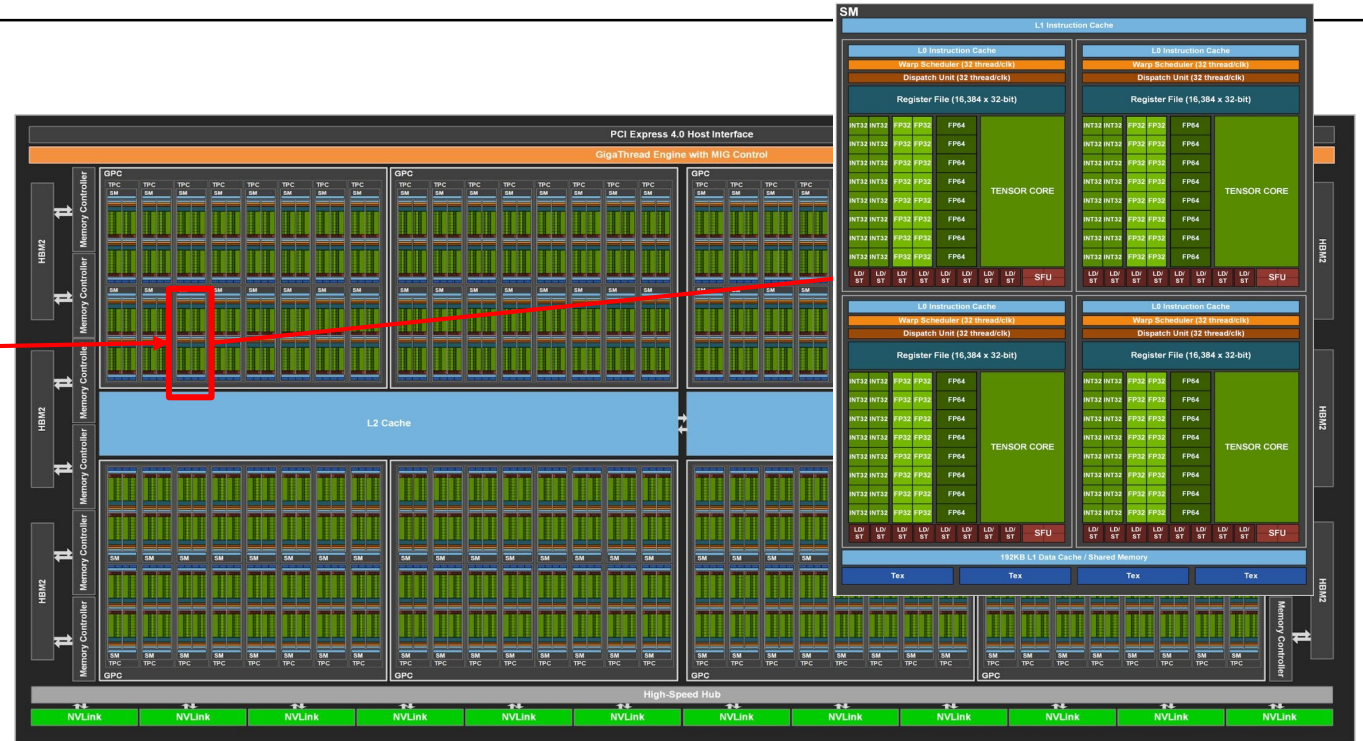
AMD Zen2 “Rome”



Nvidia A100 “Ampere” SXM4 specs

Architecture

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



© Nvidia

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

↙

SMs

↙

CUDA
cores/SM

↙

FP
ops/cy

$$n_{SM} = 108$$

$$n_{core} = 32$$

$$n_{FP} = 2 \frac{\text{flops}}{\text{cy}}$$

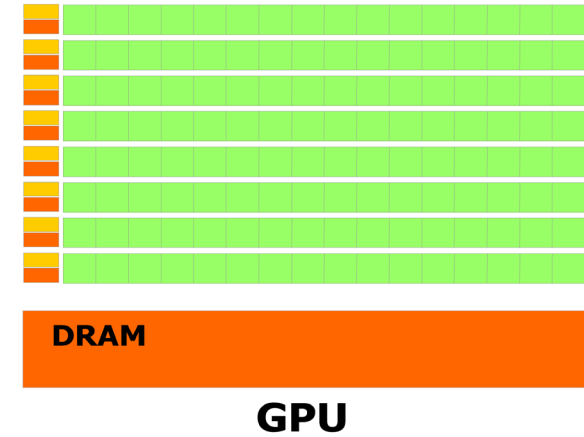
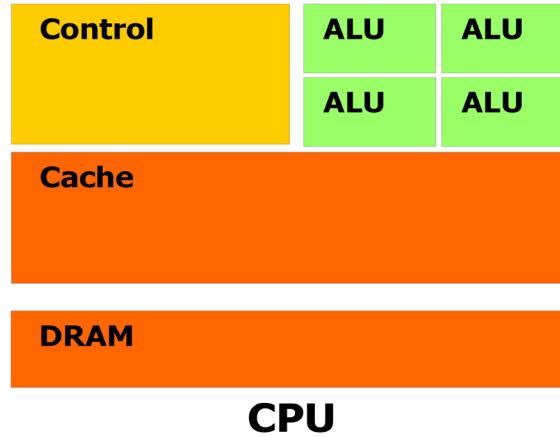
$$f = 1.4 \frac{\text{Gcy}}{\text{s}}$$

Trading single thread performance for parallelism: GPGPUs vs. CPUs

GPU vs. CPU

light speed estimate
(per processor chip)

MemBW ~ 7 – 10x
Peak ~ 4 – 8x



	CPU	GPU
	2 x AMD EPYC 7742 "Rome"	NVidia Tesla A100 "Ampere"
Cores@Clock	2 x 64 @ 2.25 GHz	108 SMs @ ~1.4 GHz
FP32 Performance/core	72 GFlop/s	~179 GFlop/s
Threads@STREAM	~16	~ 100000
FP32 peak	9.2 TFlop/s	~19.5 TFlop/s
Stream BW (meas.)	2 x 180 GB/s	1400 GB/s
Transistors / TDP	~2x40 Billion / 2x225 W	54 Billion/400 W

Quiz time



“Simple” predictive 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)

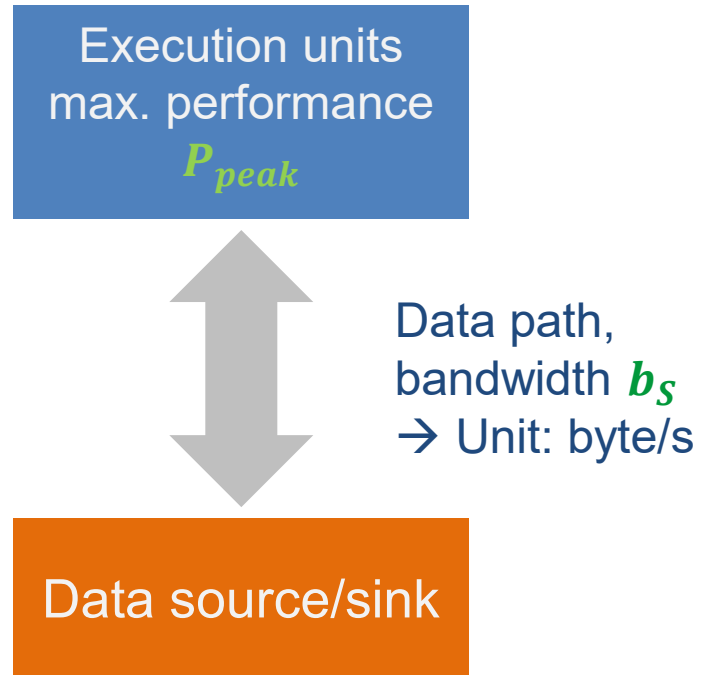
W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. Self-edition (2000)

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:



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

Also in use: Code balance $B_c = \frac{V}{N}$
→ Unit: byte/flop

Other metrics for work are possible

Naïve Roofline Model

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

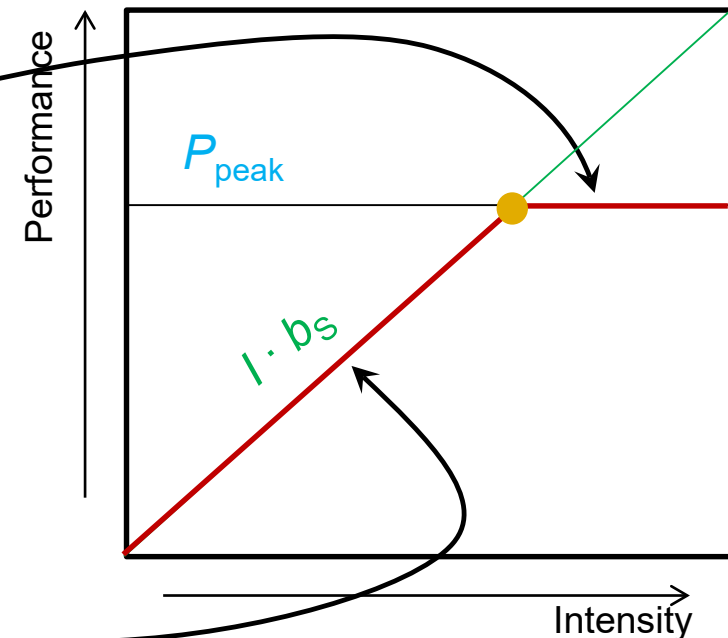
The bottleneck is either

- 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

Apply the naive Roofline model in practice

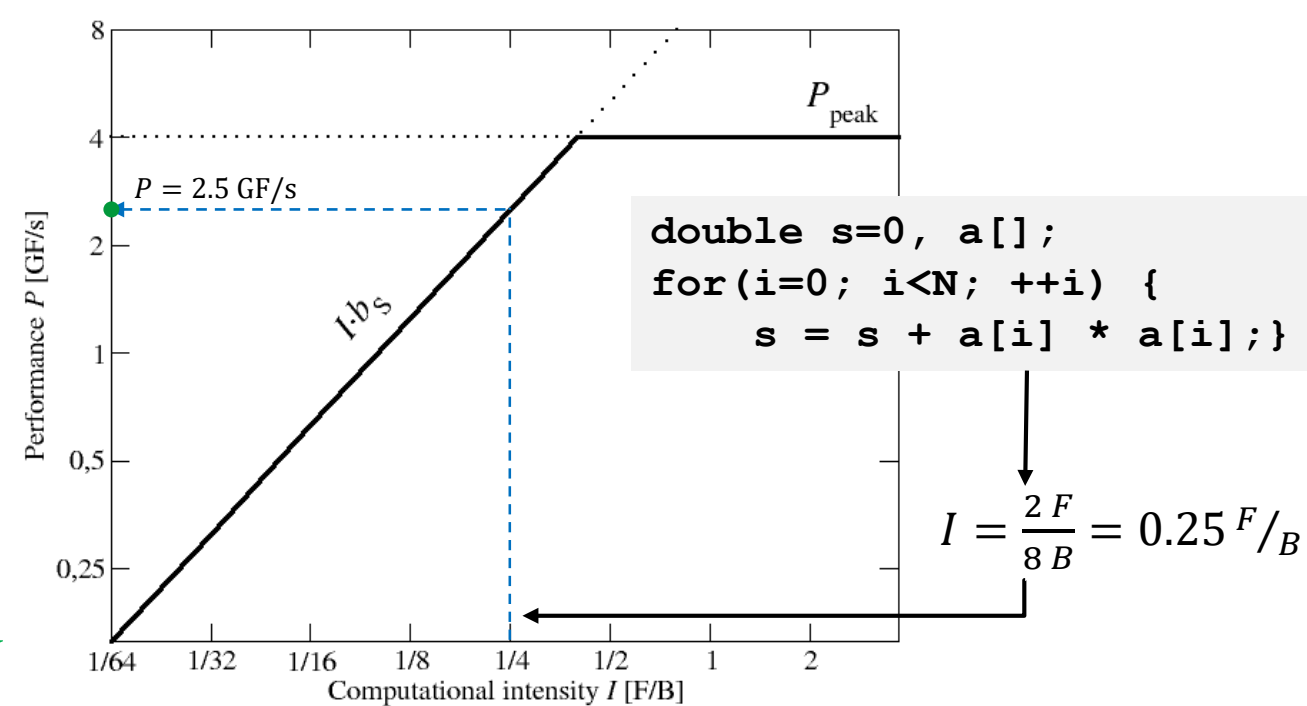
- Machine parameter #1: Peak performance: $P_{peak} \left[\frac{F}{s} \right]$
 - Machine parameter #2: Memory bandwidth: $b_S \left[\frac{B}{s} \right]$
 - Code characteristic: Computational intensity: $I \left[\frac{F}{B} \right]$
- } Machine model
} Application model

Machine properties:

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

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

Application property: I

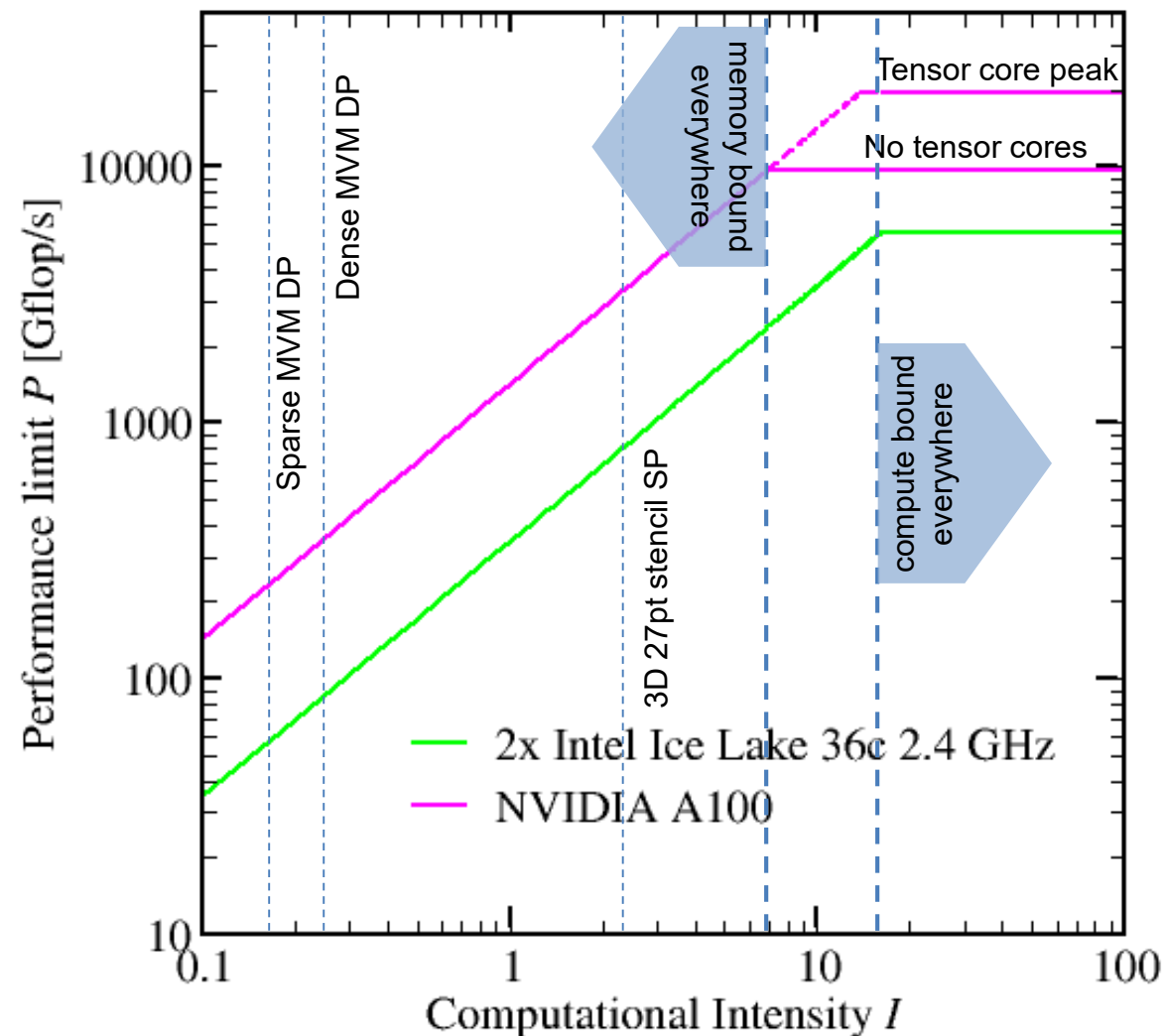


Roofline for architecture and code comparison

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 → flat ceiling may be code specific ($P_{peak} \rightarrow P_{max}$)
 - Other bandwidth bottlenecks may apply → there may be other sloped ceilings




Example: The sum reduction

A not-so-simple loop



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)

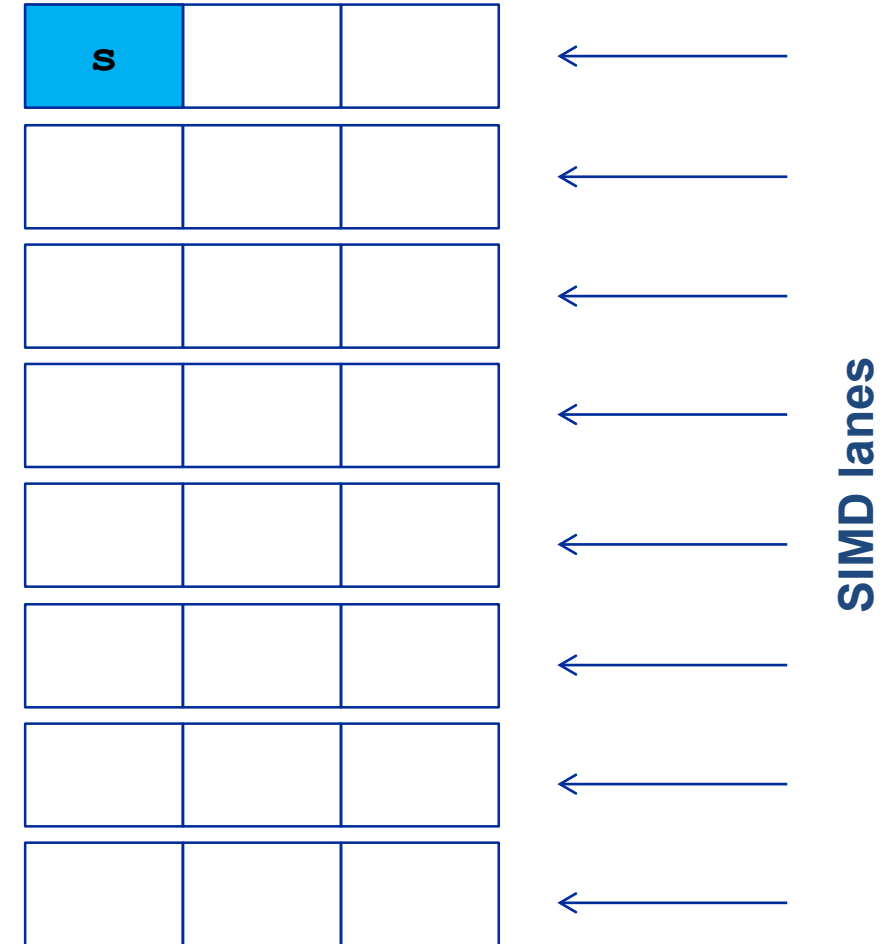
Plain scalar code, no SIMD

```
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 (II)

Scalar code, 3-way “modulo variable expansion” (MVE)

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

loop:

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

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

```
i+=3 →? loop
```

```
result ← r1.0+r2.0+r3.0
```

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

s1	s2	s3

→ 1/8 of ADD peak

Applicable peak for the sum reduction (III)

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

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

loop:

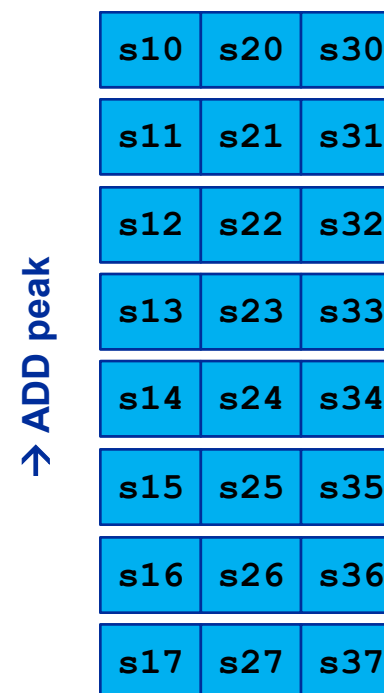
```
LOAD [r4.0,...,r4.7] ← [a(i),...,a(i+7)] # SIMD LOAD
LOAD [r5.0,...,r5.7] ← [a(i+8),...,a(i+15)] # SIMD
LOAD [r6.0,...,r6.7] ← [a(i+16),...,a(i+23)] # SIMD
```

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

i+=24 →? loop

result ← r1.0+r1.1+...+r3.6+r3.7

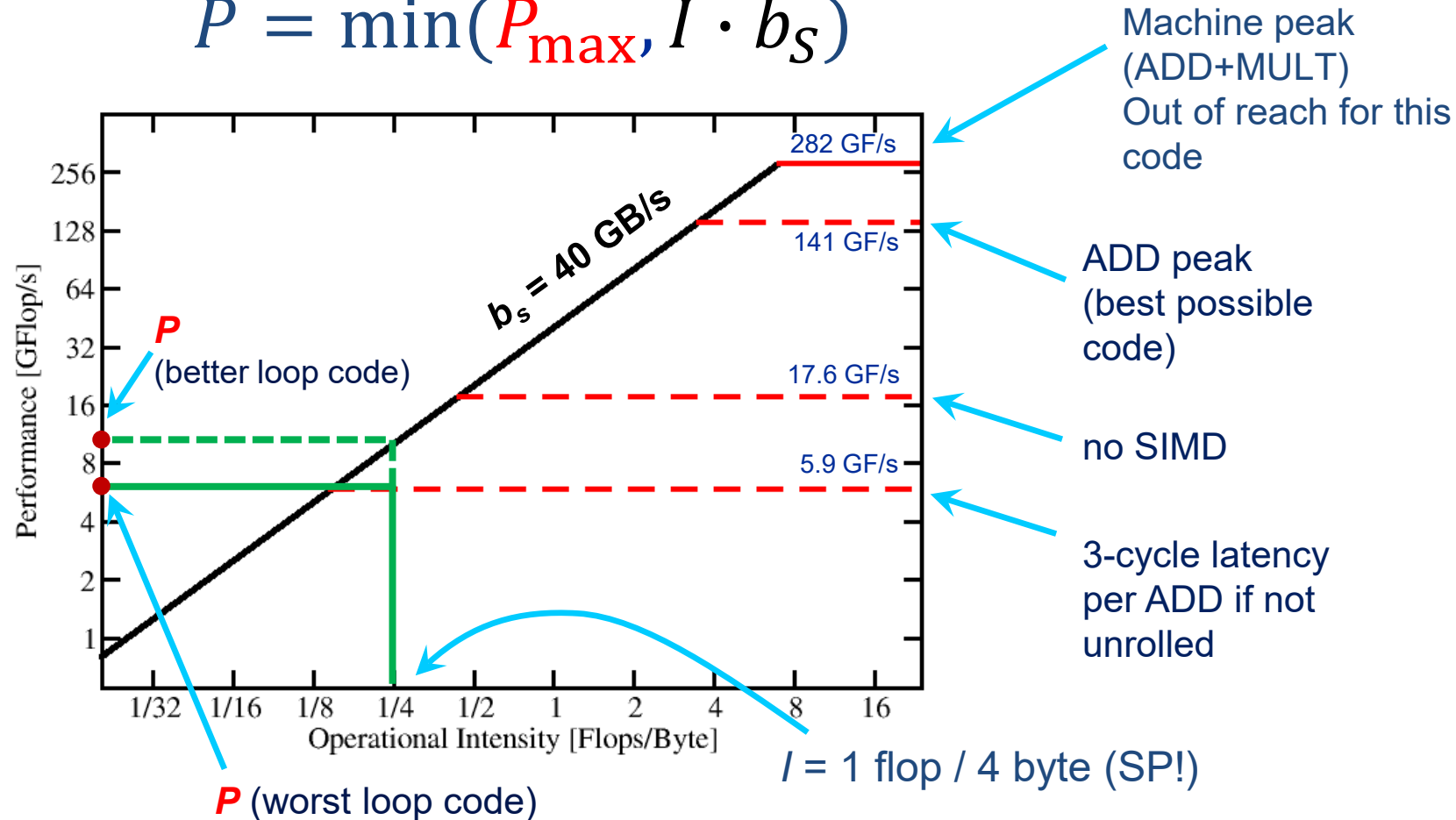
```
for (int i=0; i<N; i+=24){
  s10 += a[i+0]; s20 += a[i+8]; s30 += a[i+16];
  s11 += a[i+1]; s21 += a[i+9]; s31 += a[i+17];
  s12 += a[i+2]; s22 += a[i+10]; s32 += a[i+18];
  s13 += a[i+3]; s23 += a[i+11]; s33 += a[i+19];
  s14 += a[i+4]; s24 += a[i+12]; s34 += a[i+20];
  s15 += a[i+5]; s25 += a[i+13];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



Putting it together

Example: `for(int i=0; i<N; ++i) sum += a[i];`
in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ “large” N

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

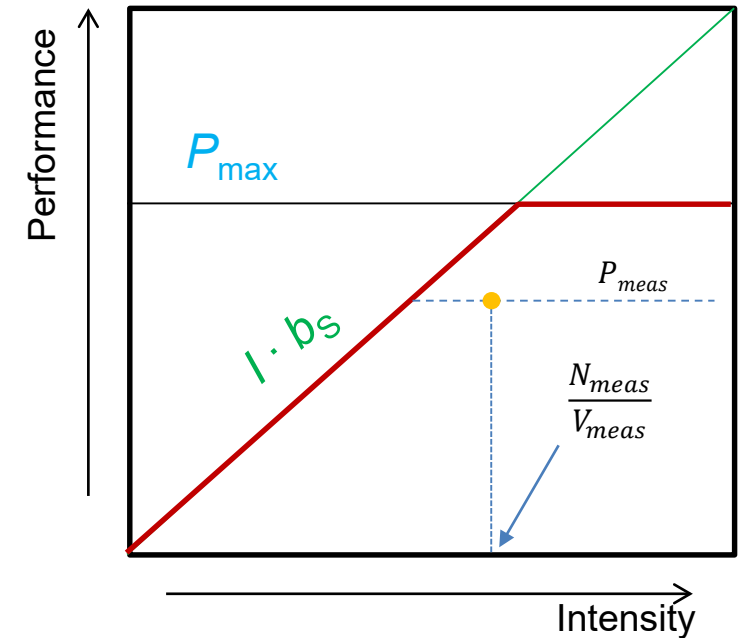


Diagnostic / phenomenological Roofline modeling



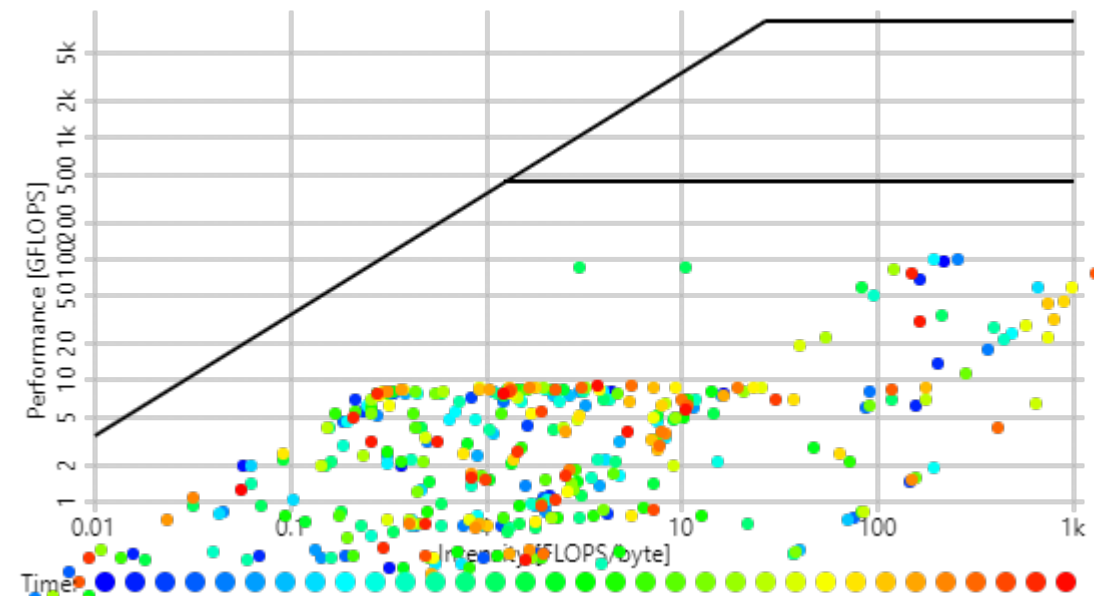
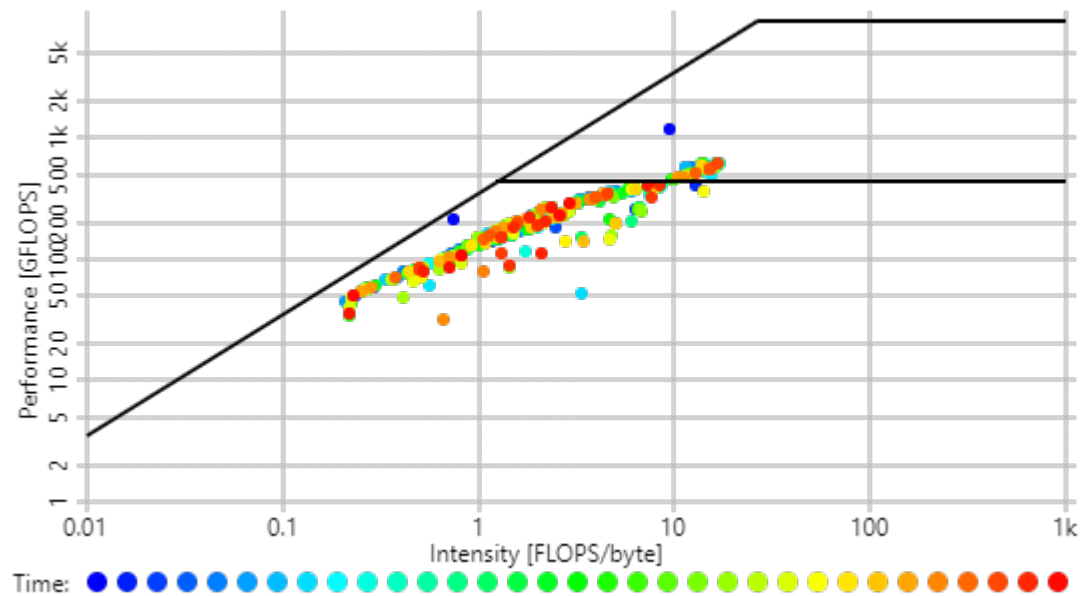
Diagnostic modeling

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



Roofline and performance monitoring of clusters

Two cluster jobs...



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



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?

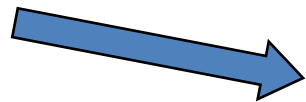
Profiling via advanced tools is often overkill

- 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 instruction 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 these 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 [...]  
-----
```

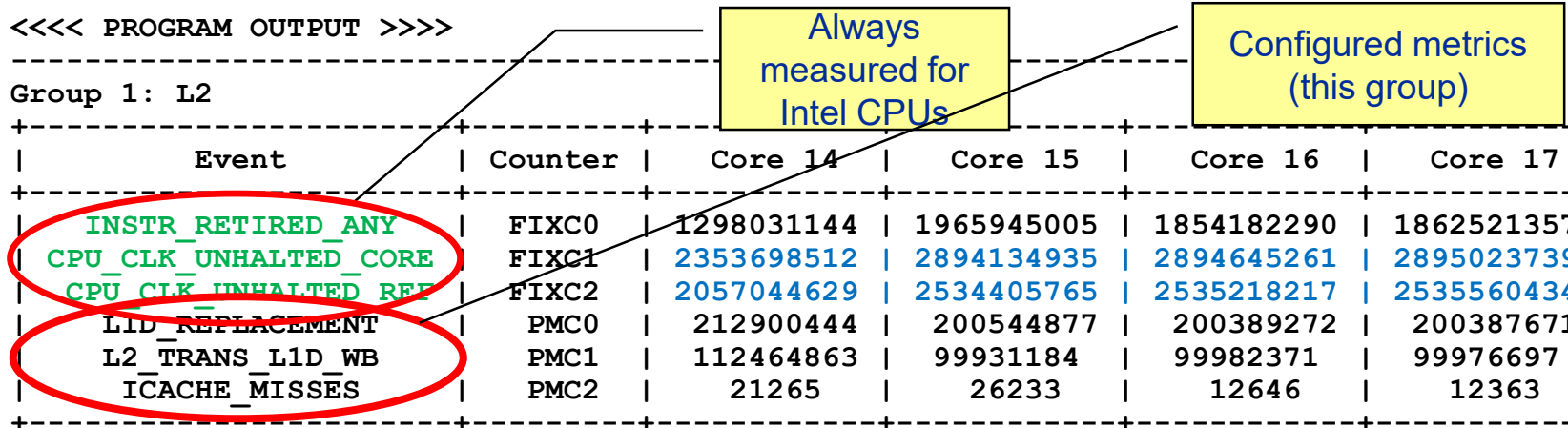
```
<<<< PROGRAM OUTPUT >>>>
```

```
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 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:

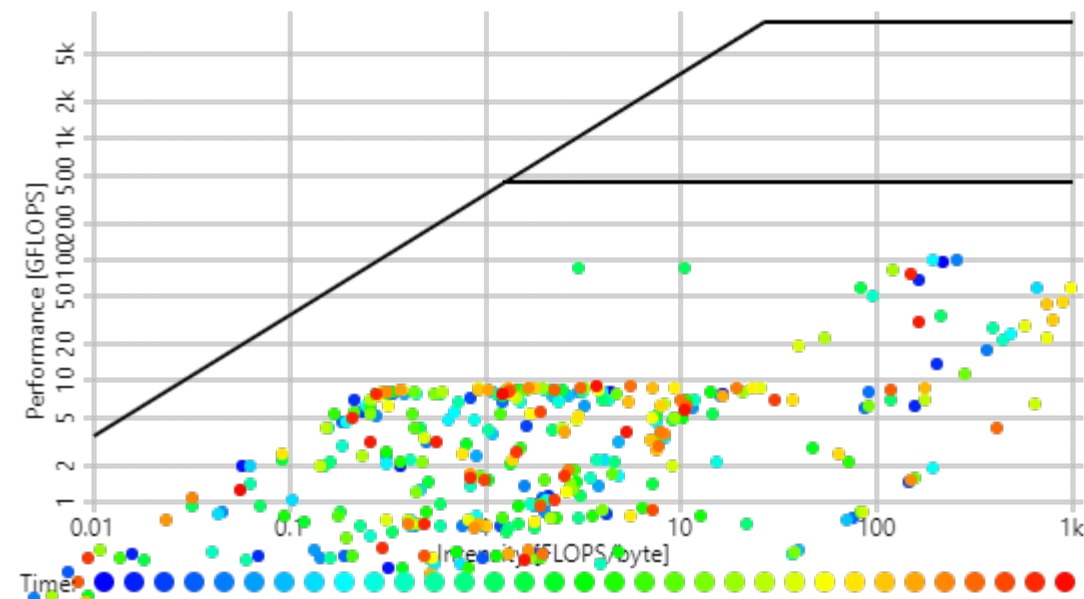
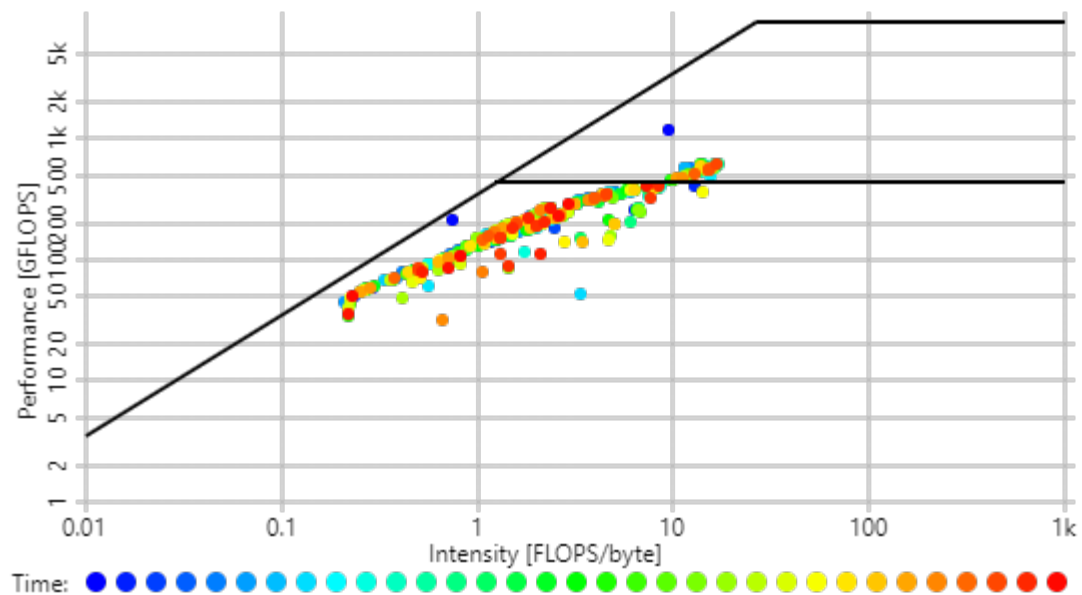
```
$ likwid-perfctr -c N:0-11 -g FLOPS_DP -S 10s
```

- 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

Roofline and performance monitoring of clusters

Using Roofline for monitoring “live” jobs on a cluster

Based on measured BW and Flop/s data via `likwid-perfctr`



likwid-perfctr with Marker API

- The MarkerAPI can restrict measurements to **code regions**
- The API only reads counters.
The configuration of the counters is still done by **likwid-perfctr**

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

LIKWID_MARKER_INIT;           // must be called from serial region
. . .
LIKWID_MARKER_START("Compute"); // call markers for each thread
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .
LIKWID_MARKER_CLOSE;         // must be called from serial region
```

- Available for C and Fortran (contributed für Julia and Python)

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

Example: Simple stencil algorithms

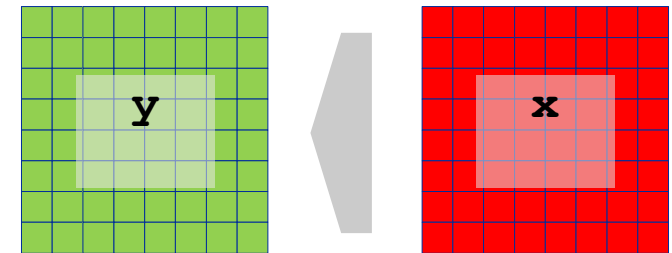


Stencil schemes

- 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,...

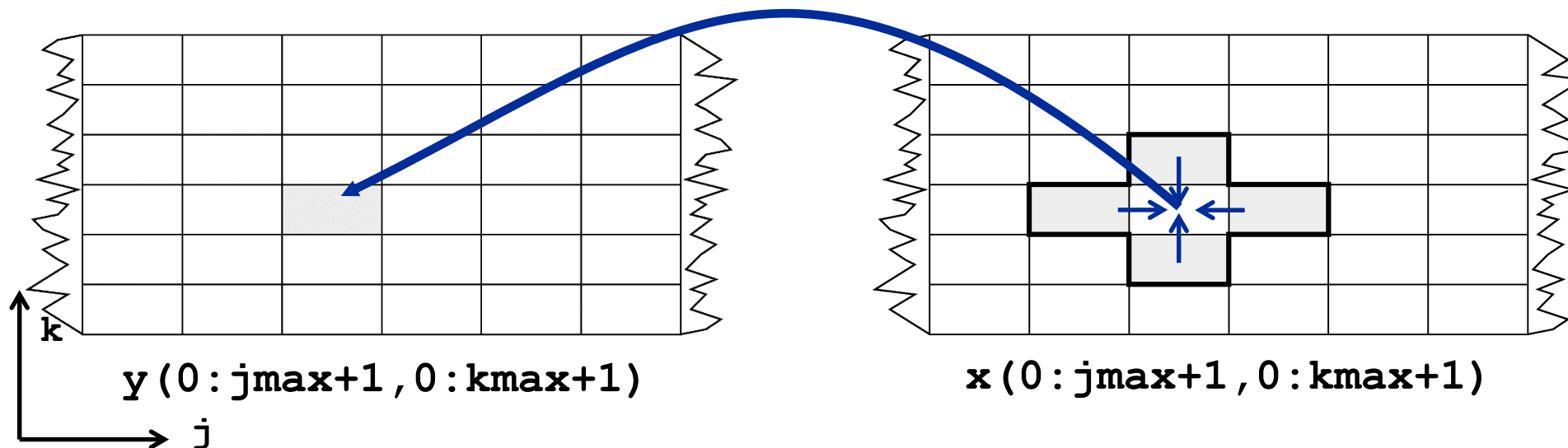


Jacobi-type 5-pt stencil sweep in 2D

```
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
```

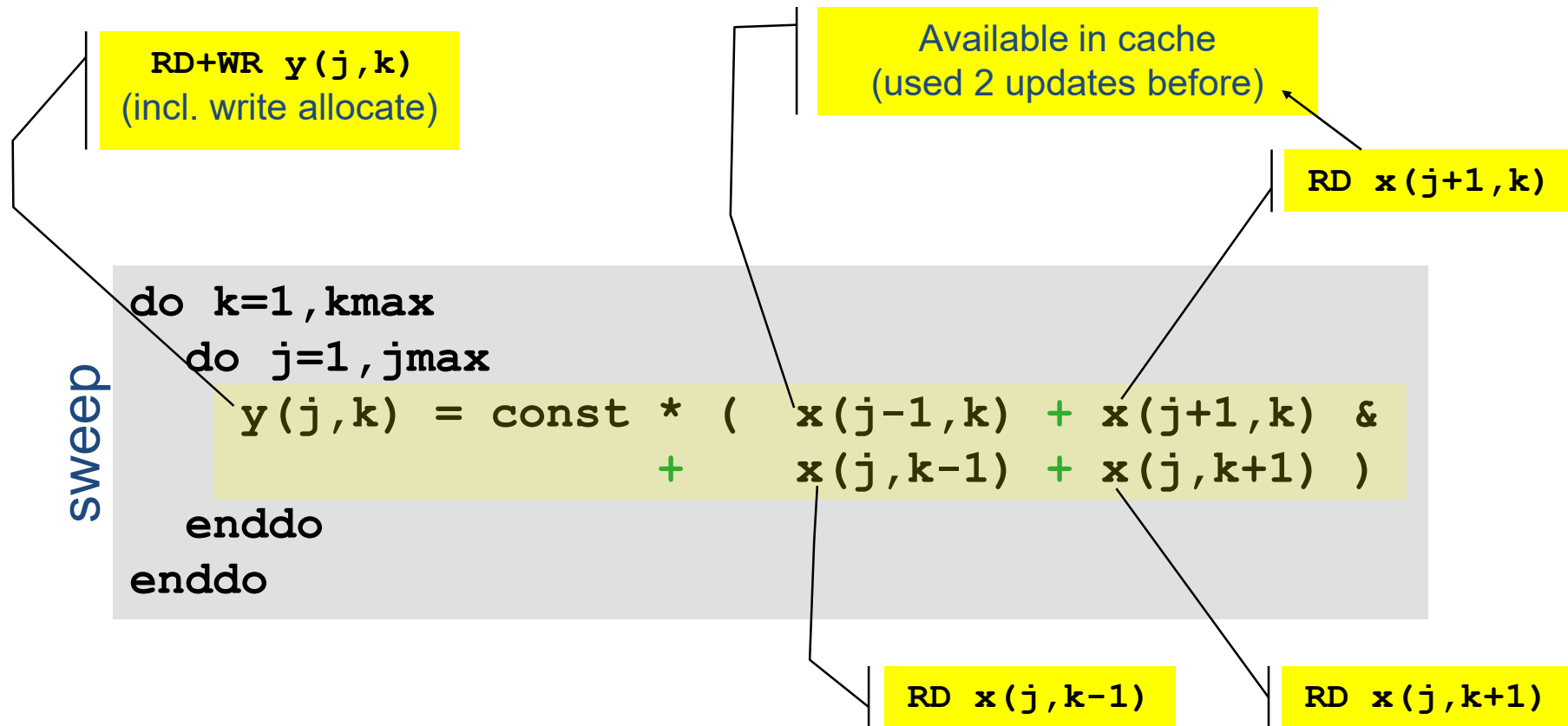
sweep

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

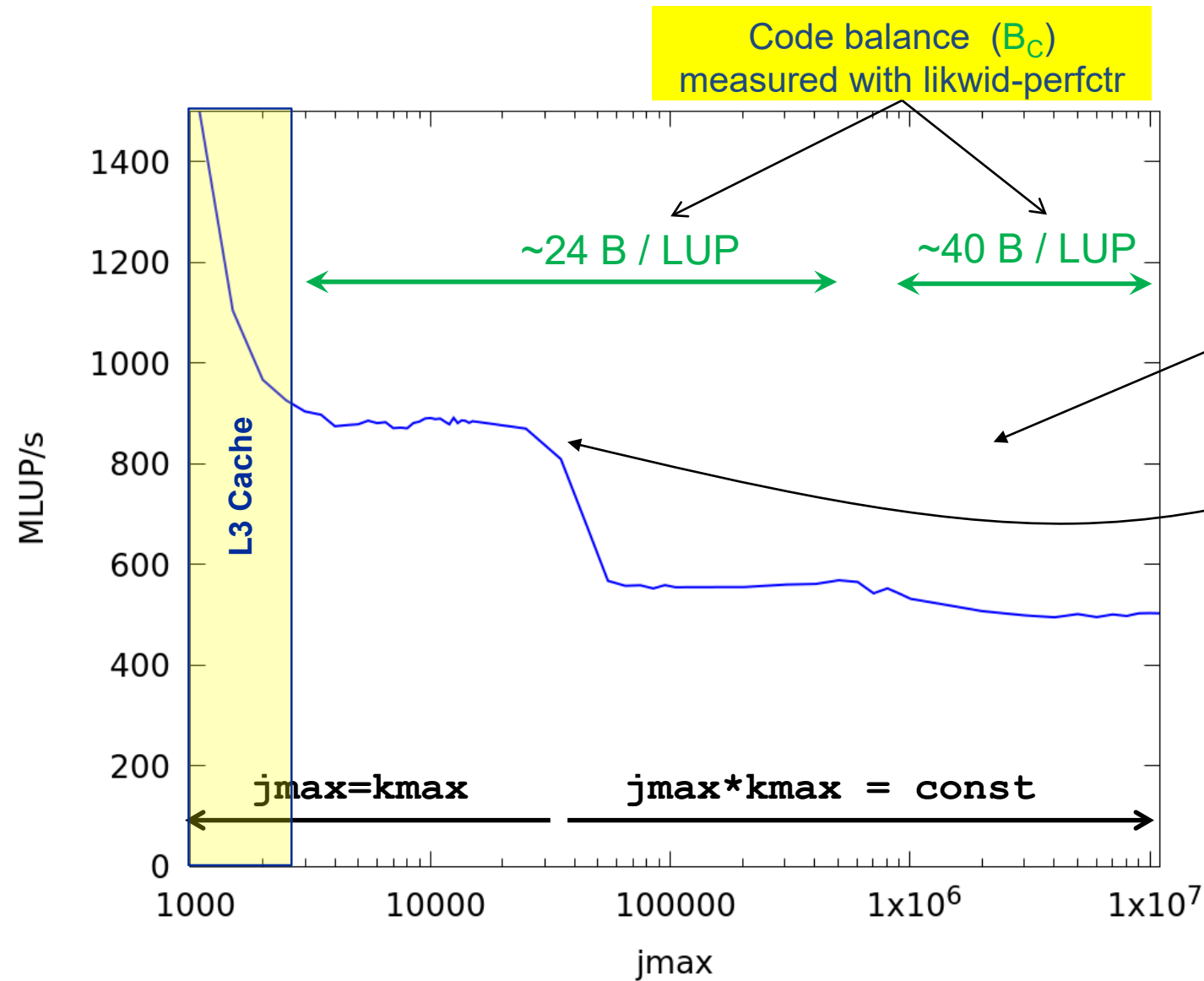


Naive balance (incl. write allocate):

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



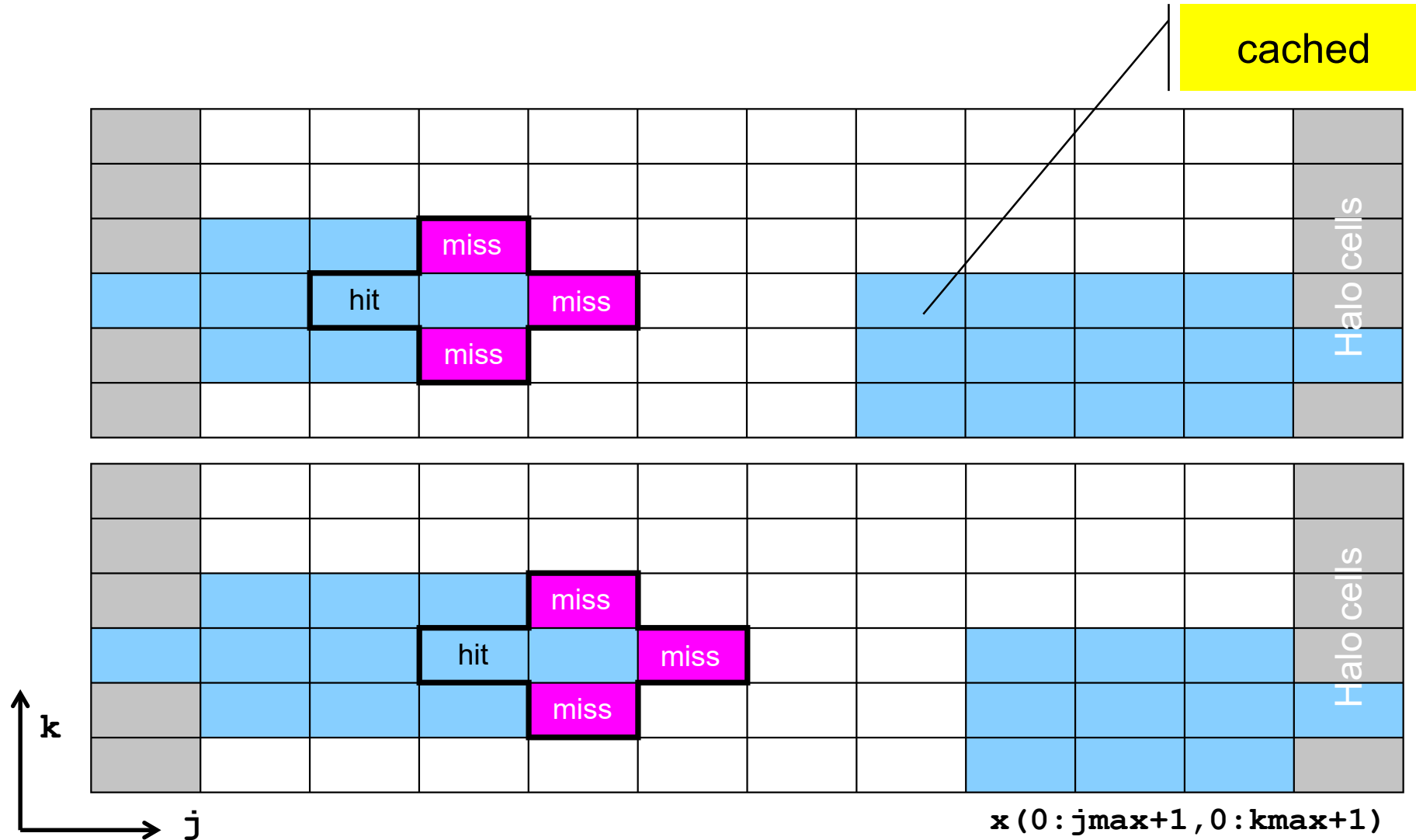
Questions:

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)

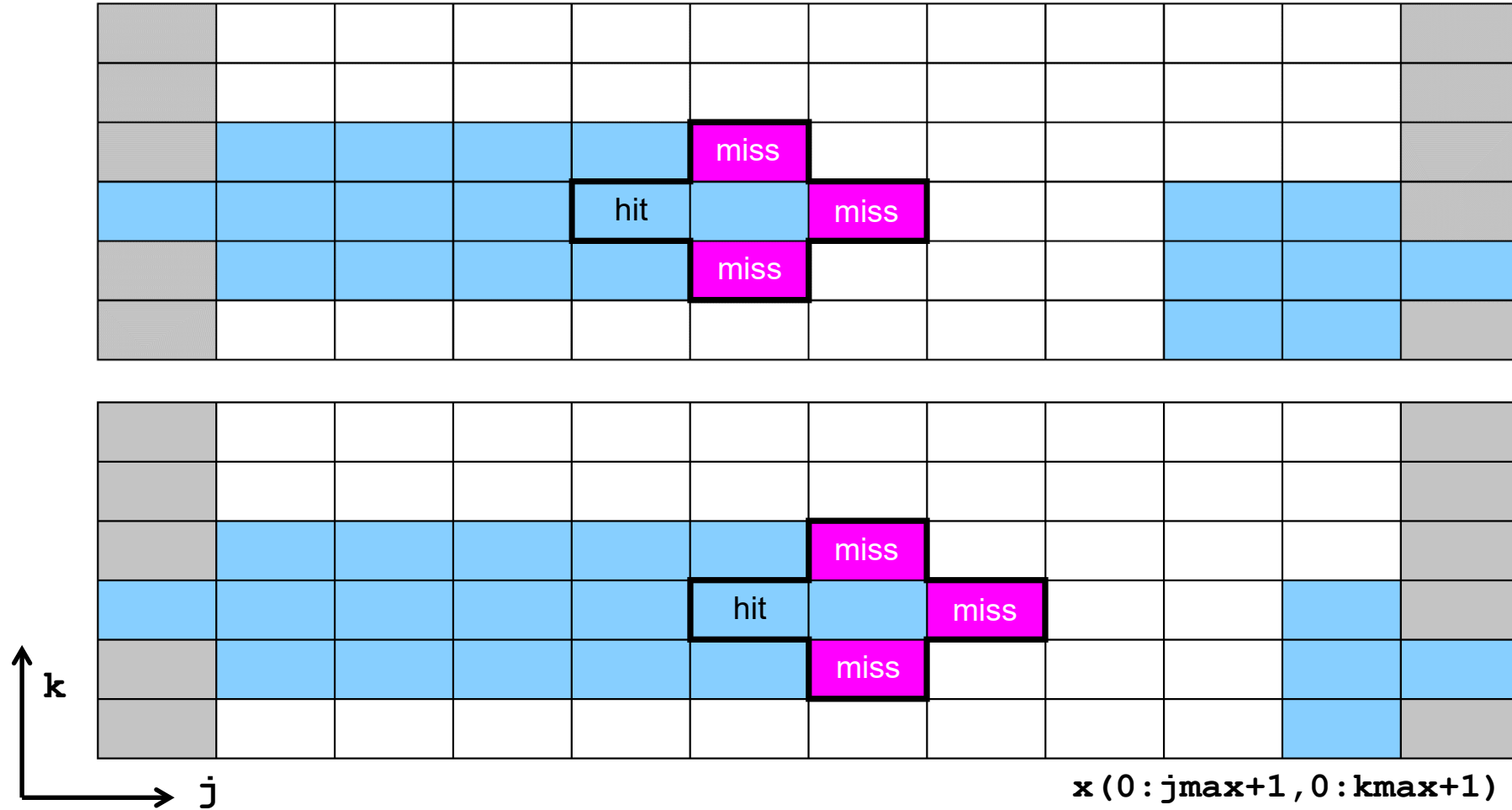
Analyzing the data flow

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



Analyzing the data flow

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

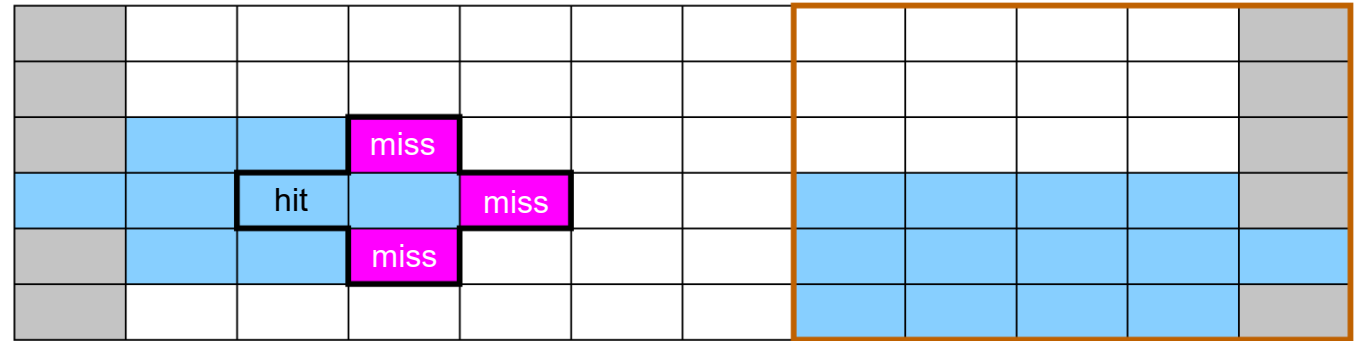
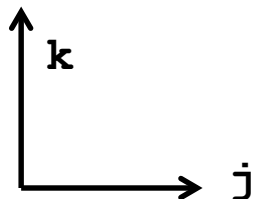


Analyzing the data flow

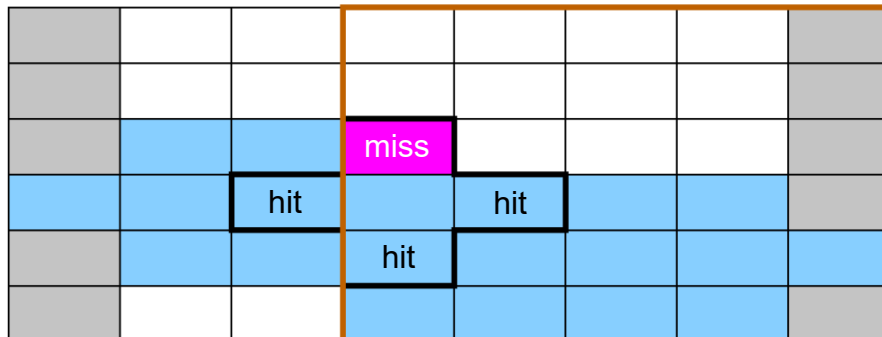
Reduce inner (j-) loop dimension successively



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



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

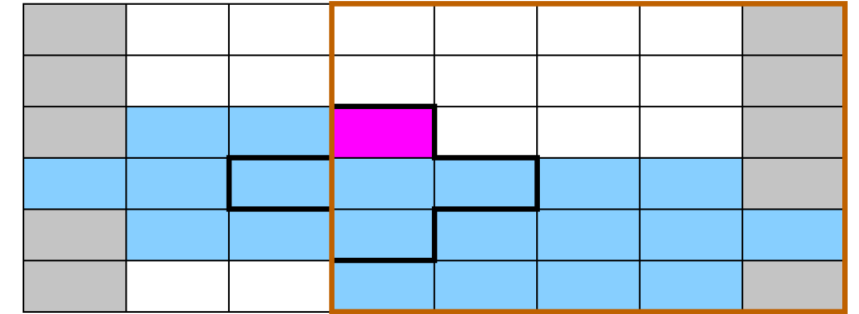


$x(0:j_{\max 2}+1, 0:k_{\max}+1)$

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}

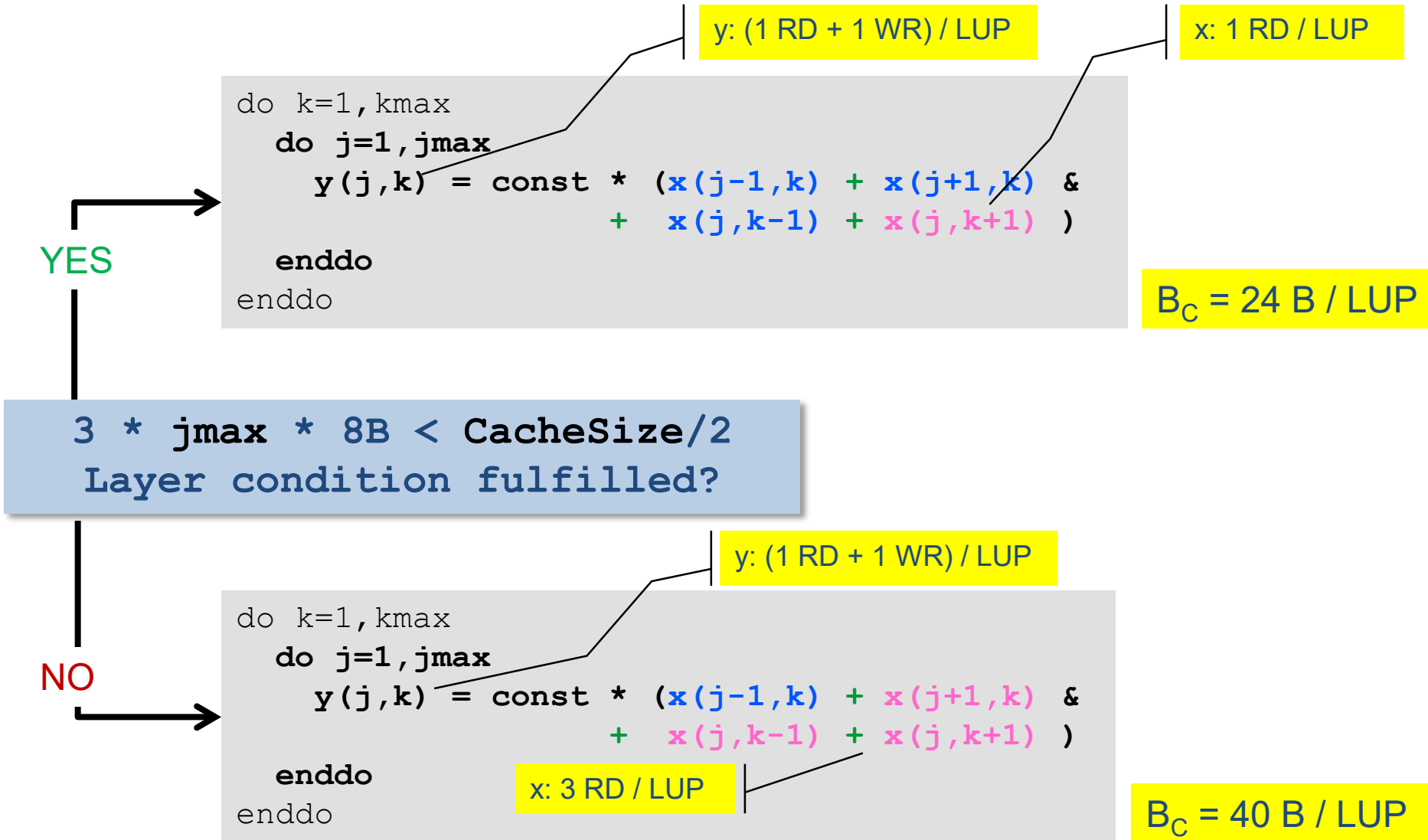
double
precision

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



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)
 $3 * \mathbf{jb}block * 8B < \mathbf{CacheSize}/2$

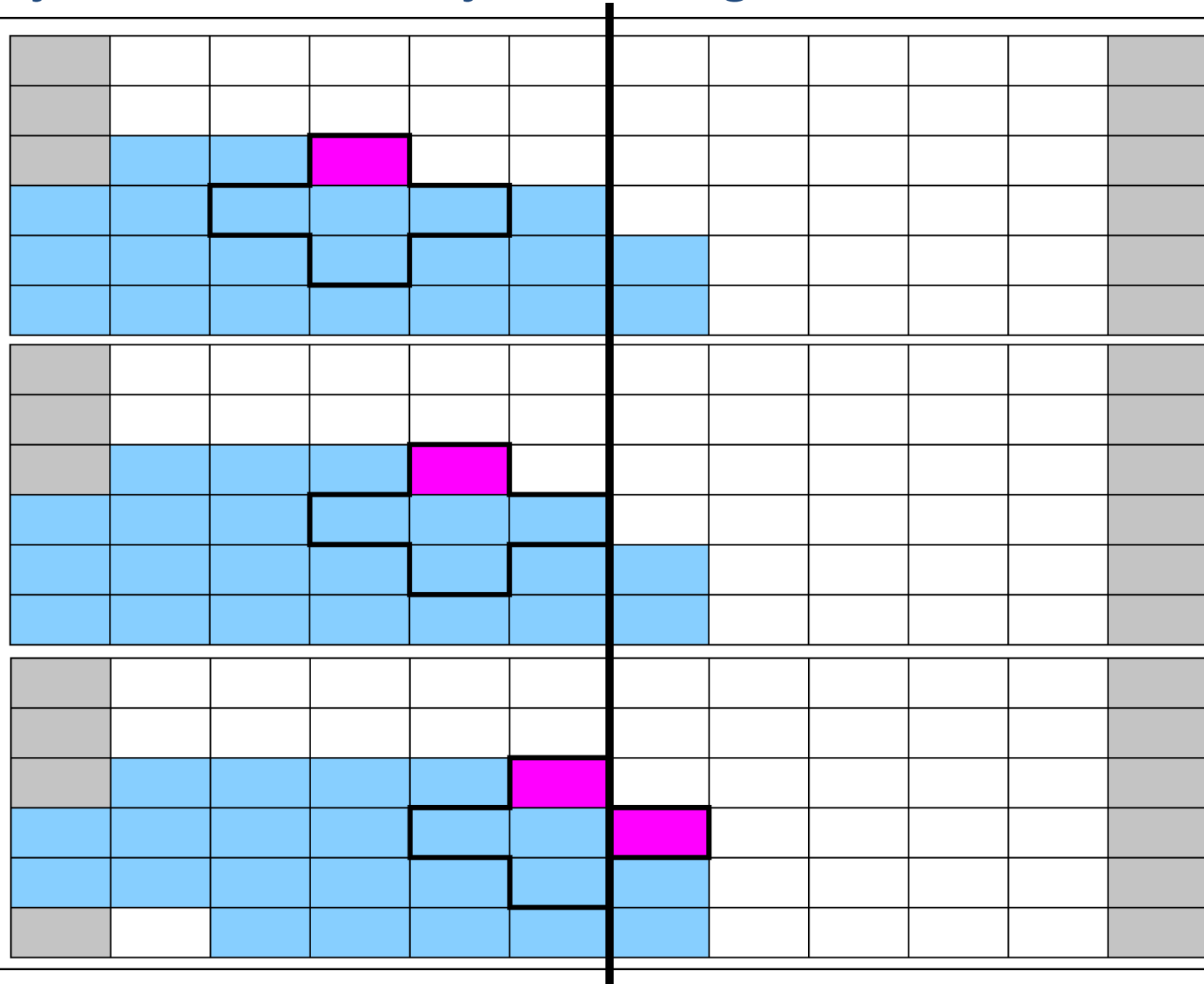
Determine for given **CacheSize** an appropriate **jb**block value:

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

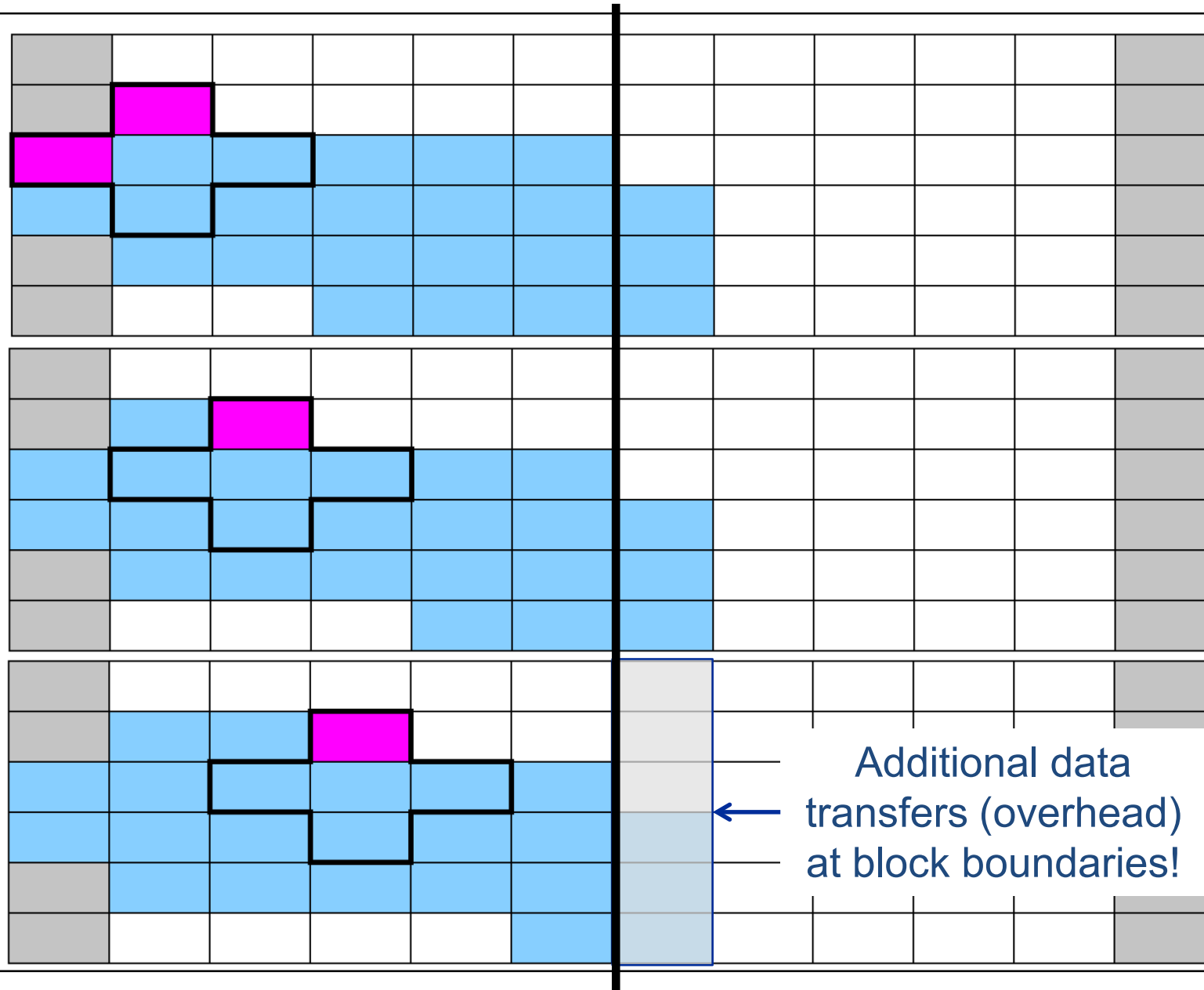
Establish the layer condition by blocking

Split domain into subblocks:

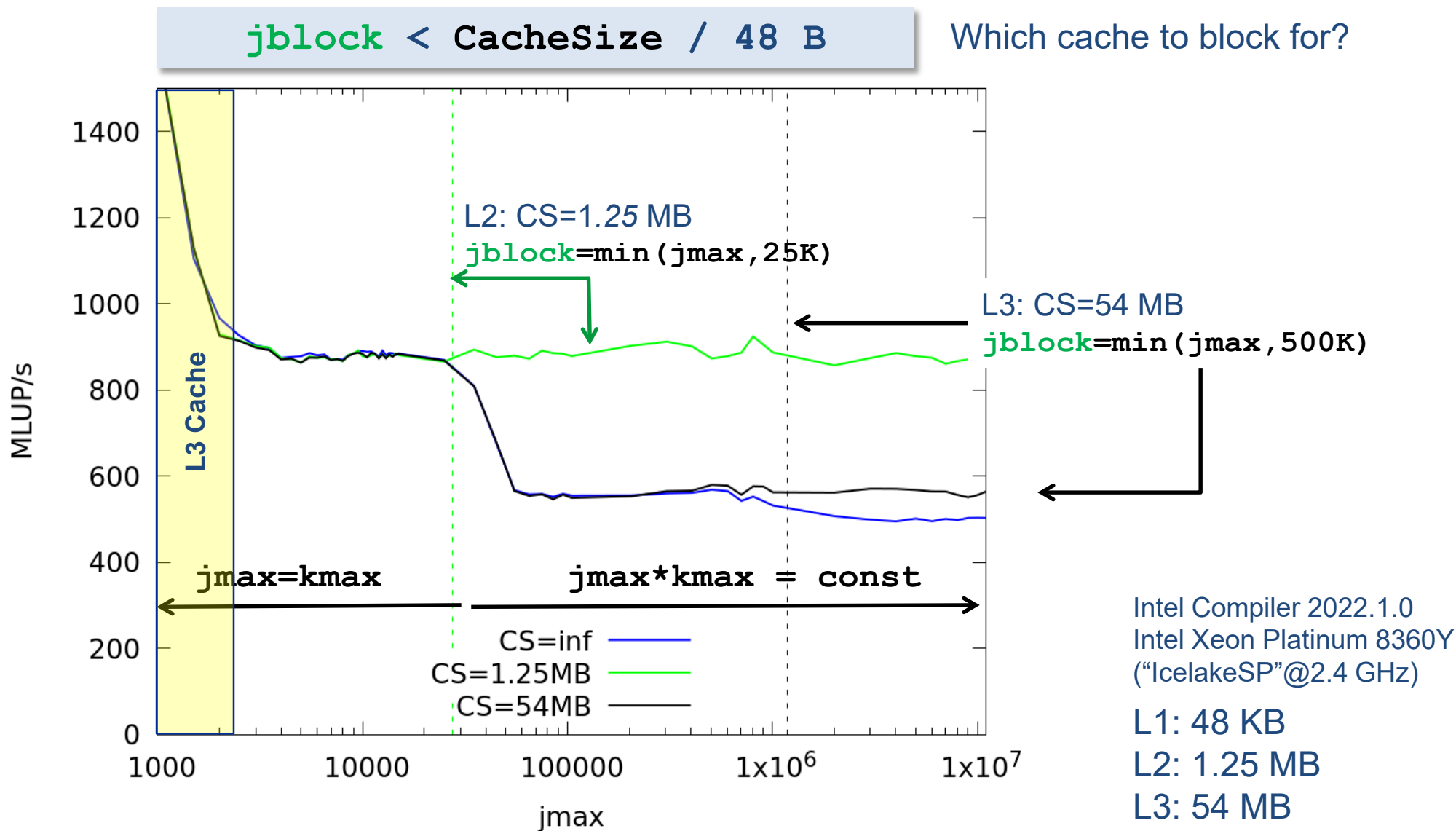
e.g. block size = 5



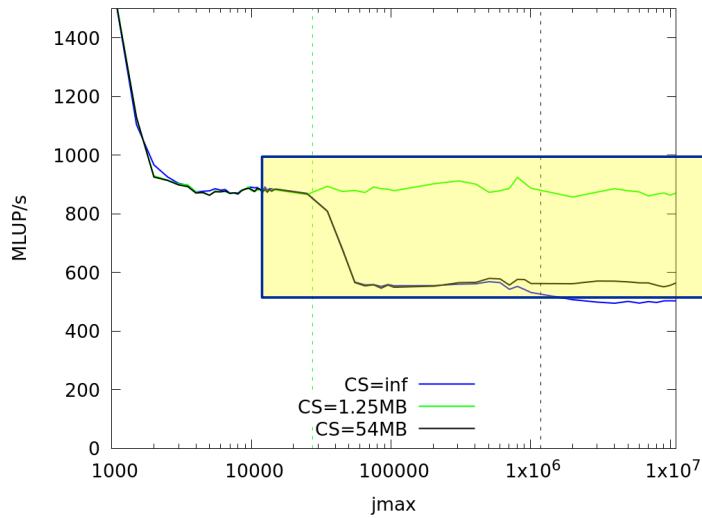
Establish the layer condition by blocking



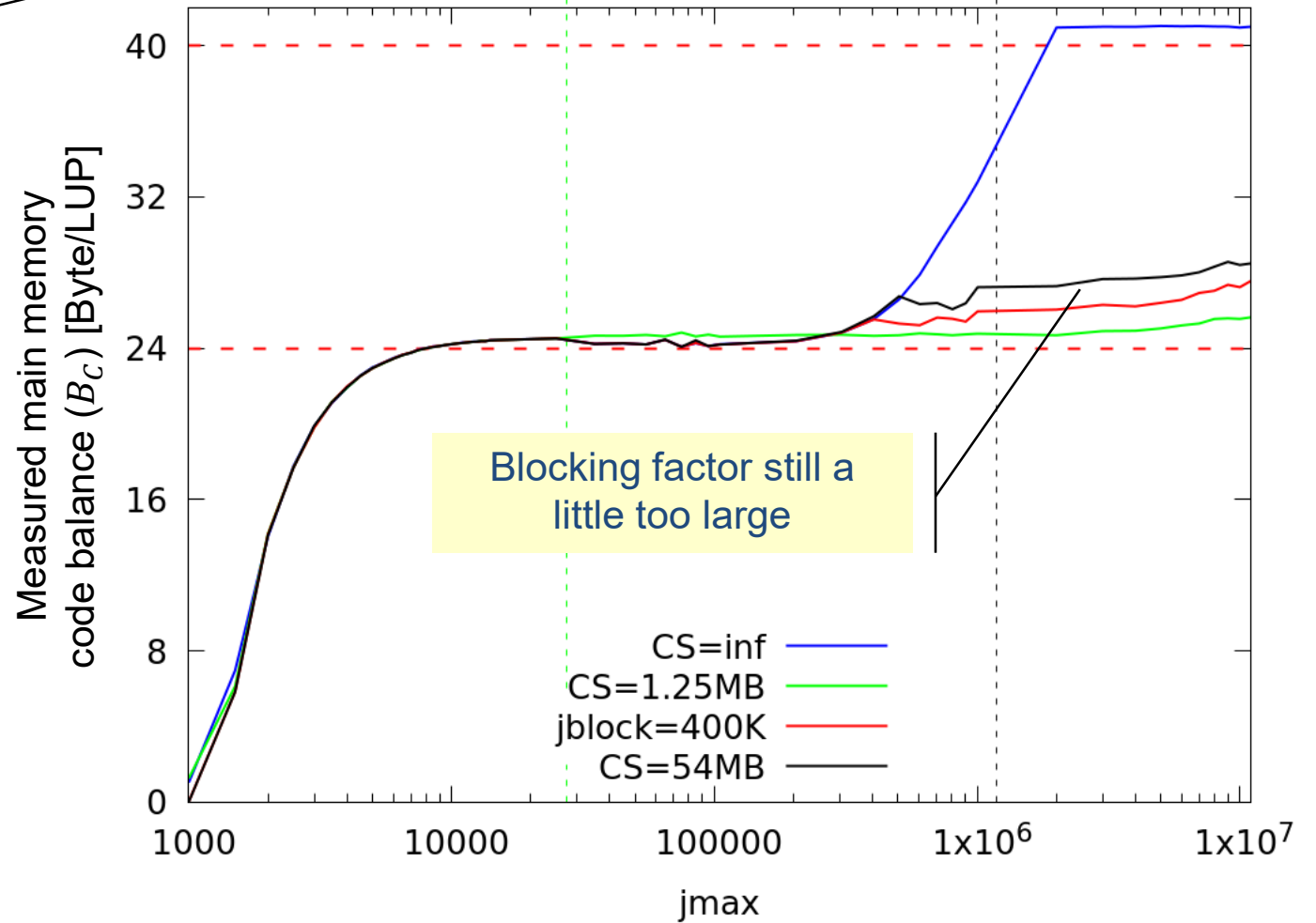
Establish layer condition by spatial blocking



Validating the model: Memory code balance



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



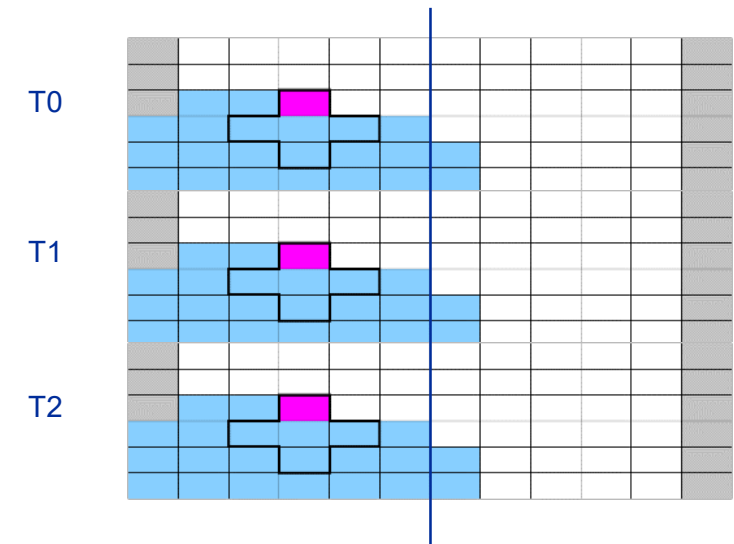
Blocking factor still a little too large

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

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 \rightarrow shared cache causes smaller blocks!

Layer condition:

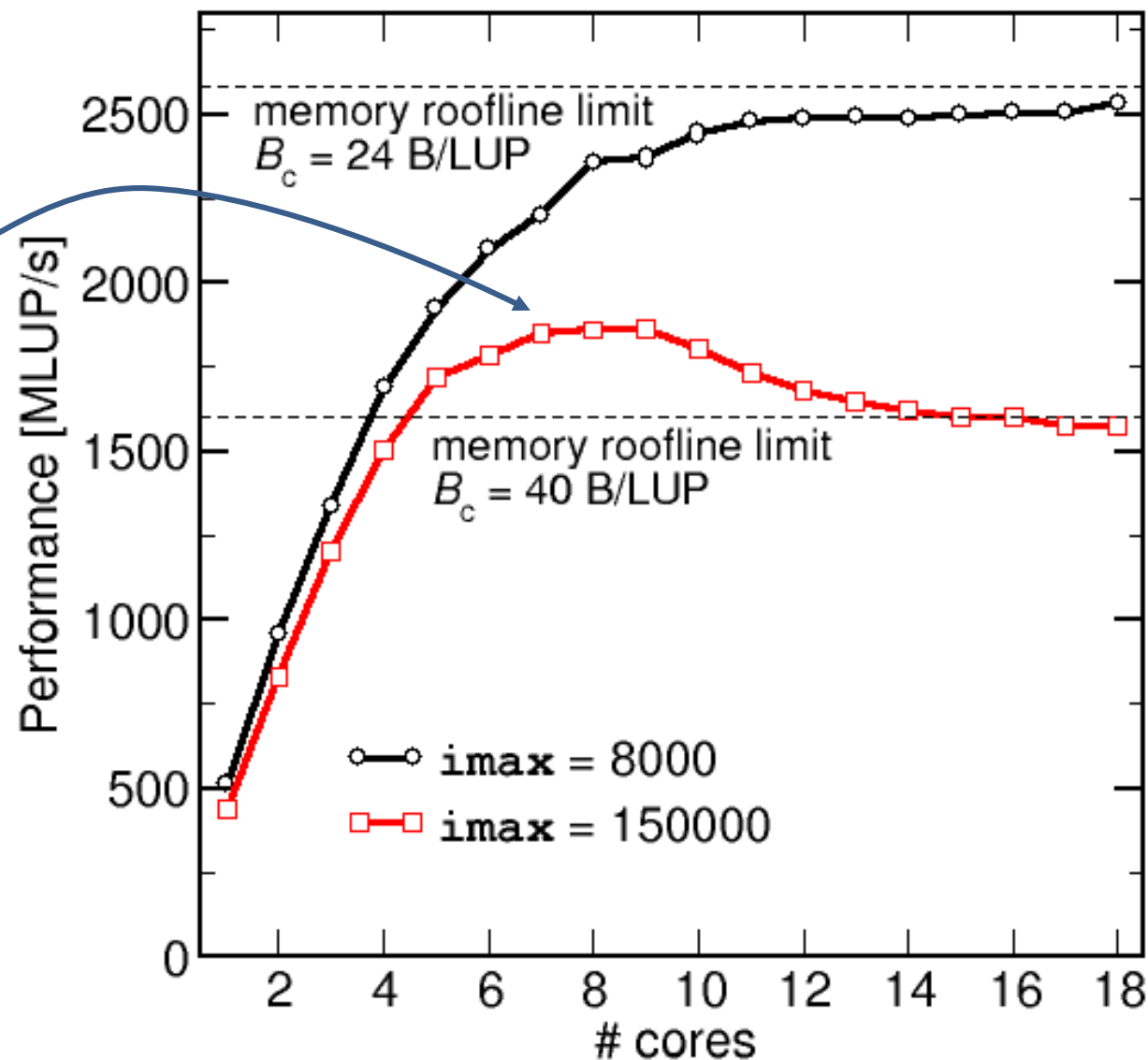
$$3 * \mathbf{jbblock} * 8B < CSt/2$$

Cache size available per thread

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
- **Avoiding slow data paths** == re-establishing the most favorable layer condition
 - Improved code showed the **predicted speedup**
 - Optimal **blocking factor** can be estimated
- **Manual analysis of stencil codes can be tedious**
 - Online Layer Condition Calculator:
<http://tiny.cc/LayerConditions>

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](#)

Quiz time



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!**

BACKUP



SIMD

The Basics



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;
```

- “**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**

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)

- 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 Icelake

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** (not available for new **icx**)

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

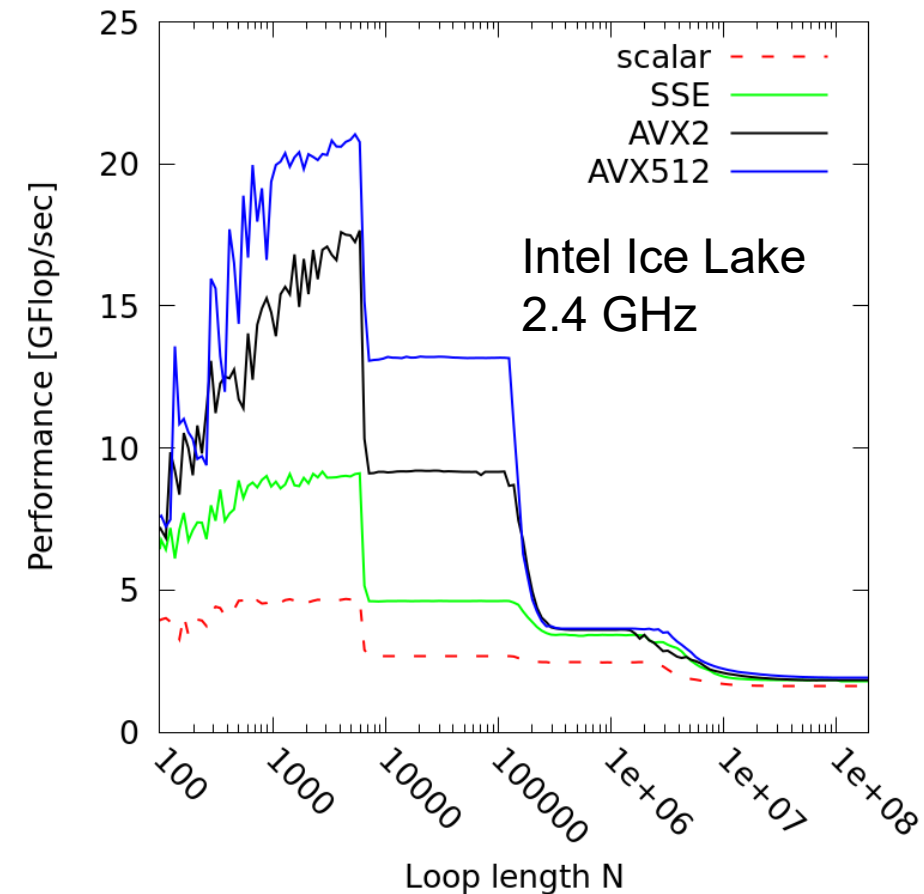
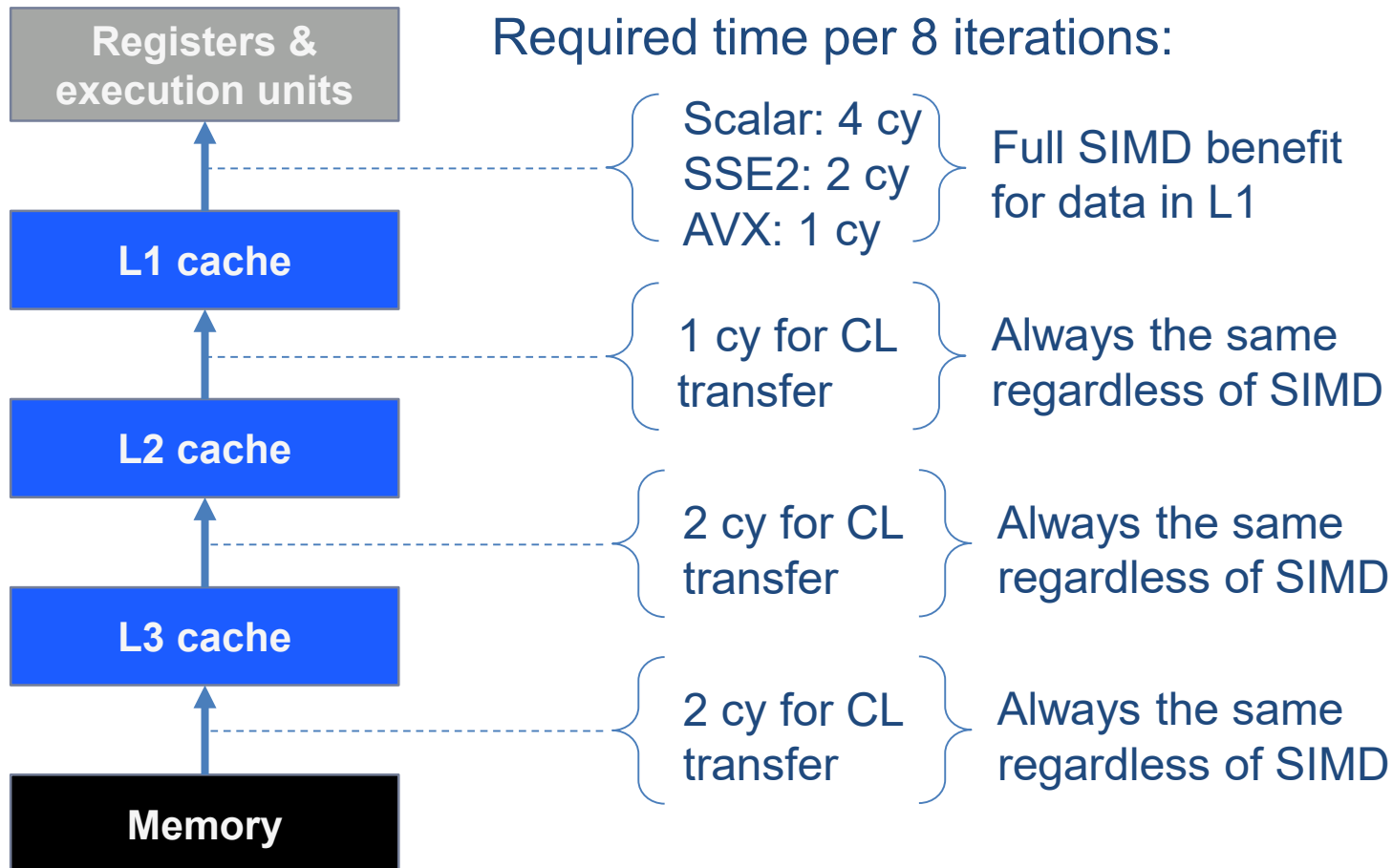
- Countable loop
- Innermost loop
- Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses:
reduction, vectorlength, private, collapse, ...

```
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];
```



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