NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Elements of Performance Engineering

Georg Hager

Erlangen National High Performance Computing Center (NHR@FAU)

One-day workshop, University of Bonn, March 17, 2026

https://go-nhr.de/EPE26

Agenda

- Basics of node-level computer architecture
- Multicore performance and tools (topology, affinity)
- Basics of the Roofline model
- Case study: Stencils
- Case study: Sparse matrix-vector multiplication (SpMV)

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Modern computer architecture

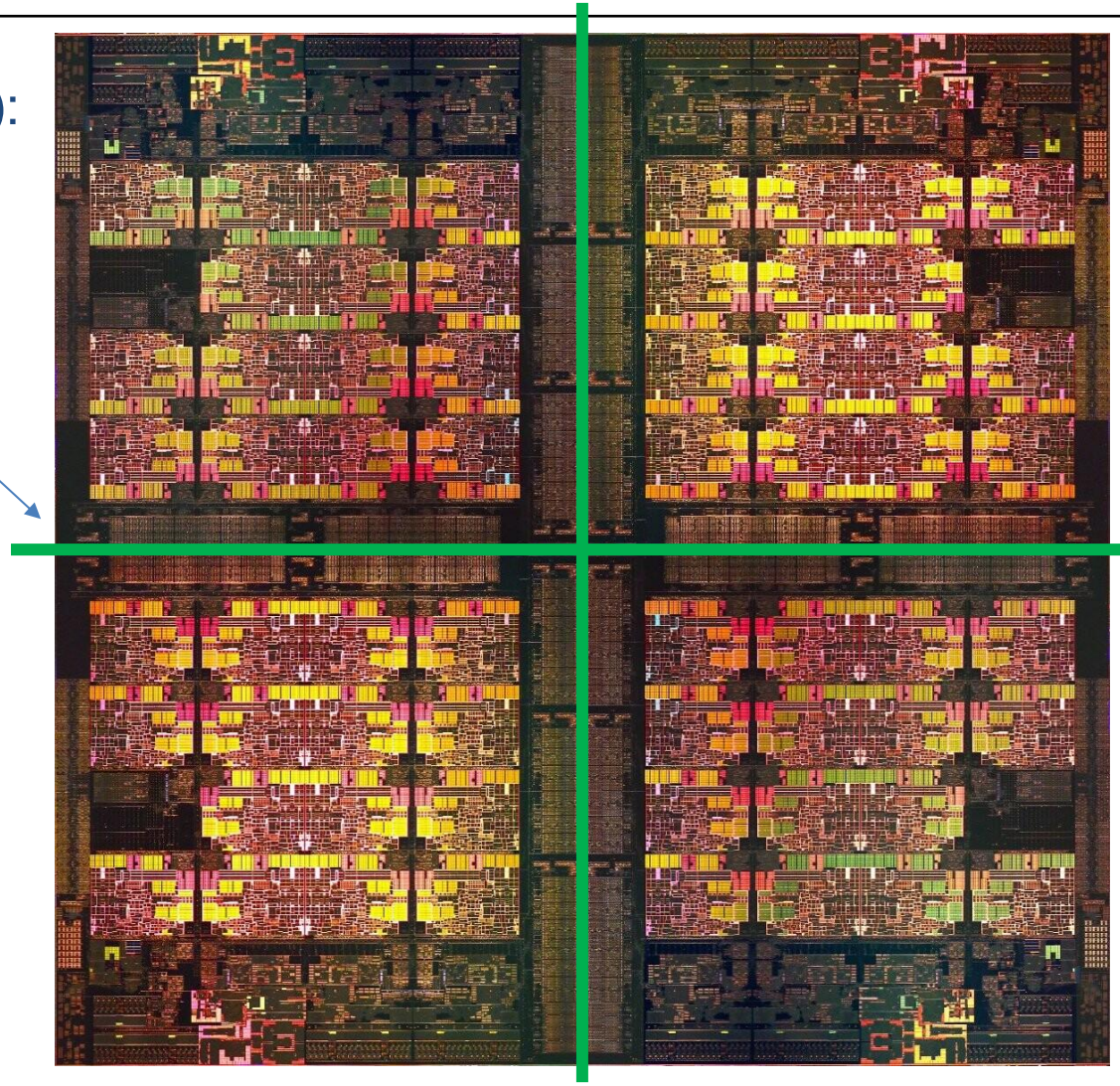
An introduction for software developers

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

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

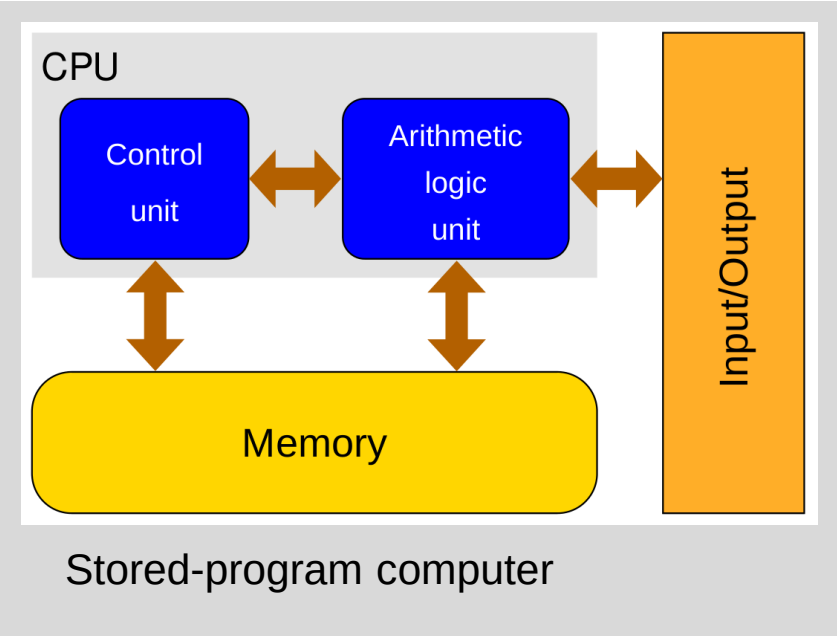
Optional: “Sub-NUMA Clustering” (SNC) mode boot option

→ One memory domain per die



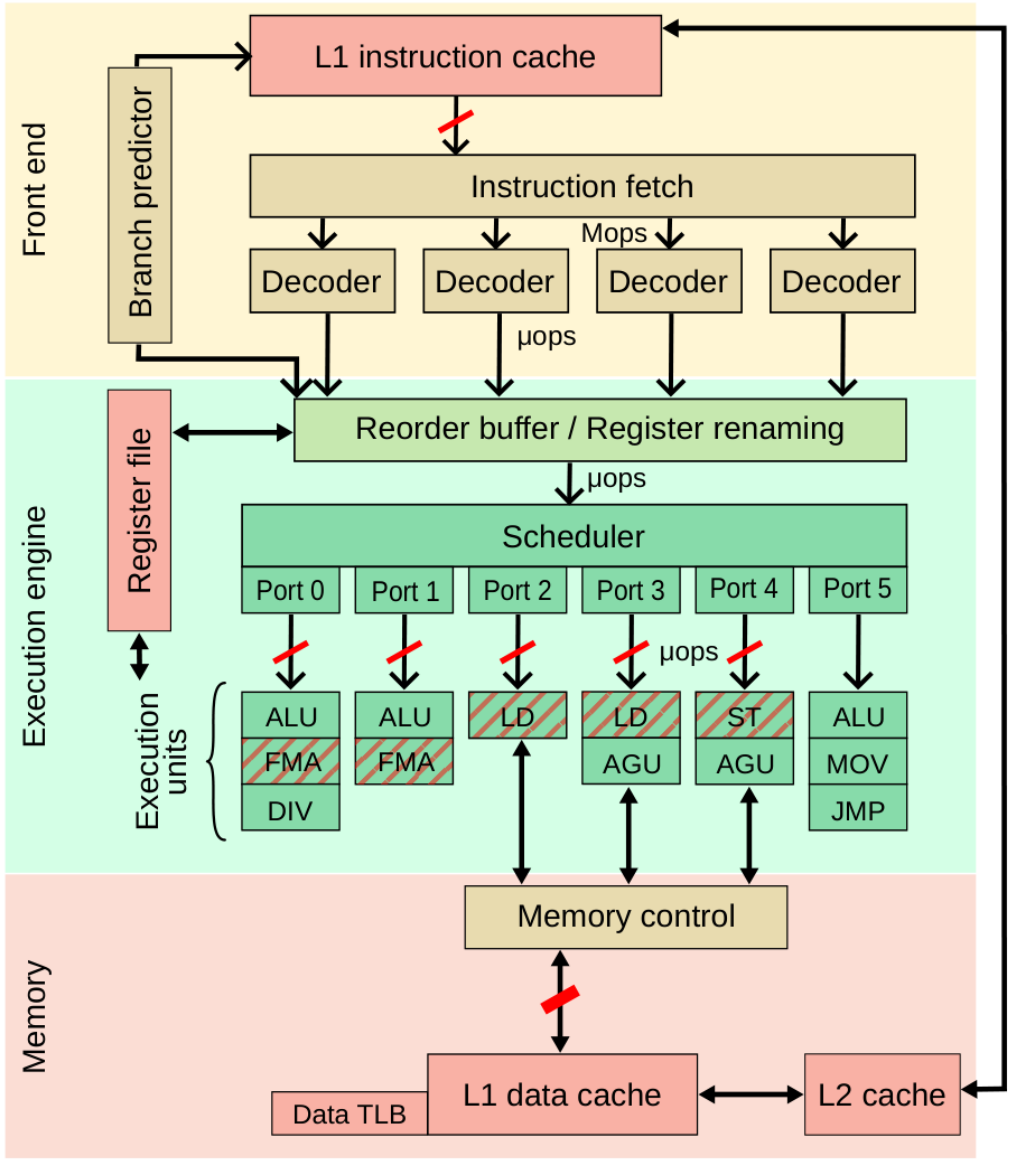
https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated

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



NHR FAU

Friedrich-Alexander-Universität
Erlangen-Nürnberg

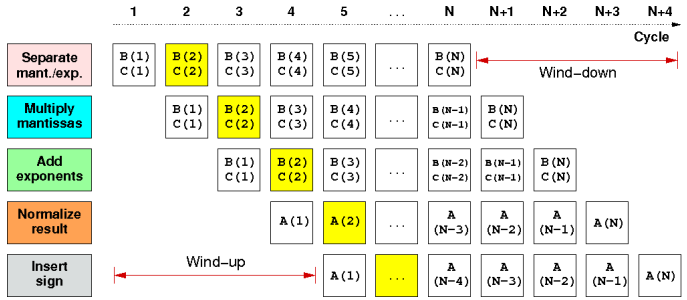
In-core features

Pipelining, Superscalarity, SIMD, SMT

Important in-core features

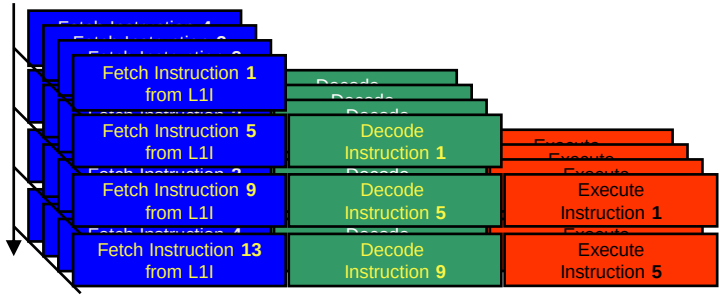
Pipelining:

Instruction execution in multiple steps



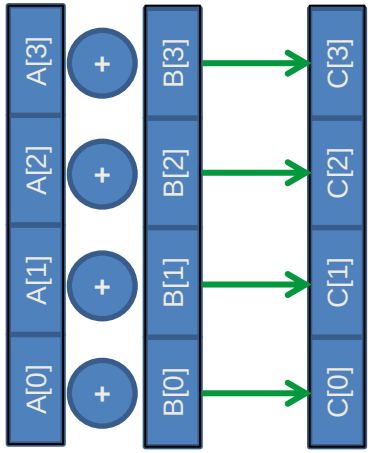
Superscalarity:

Multiple instructions per cycle



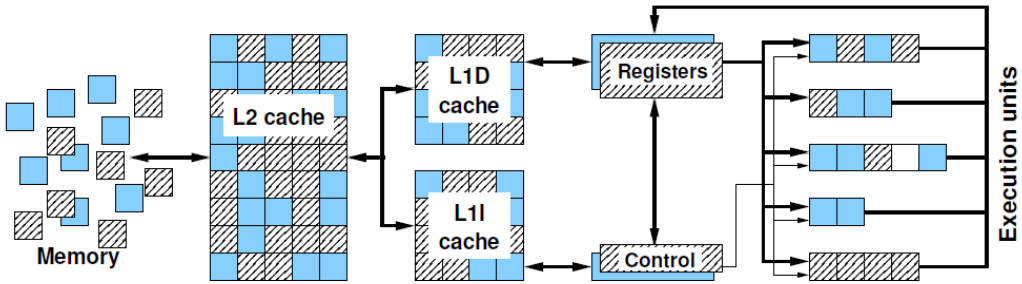
Single Instruction Multiple Data:

Multiple operations per instruction



Simultaneous Multi-Threading:

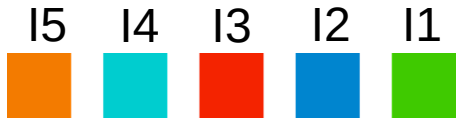
Multiple instruction sequences in parallel



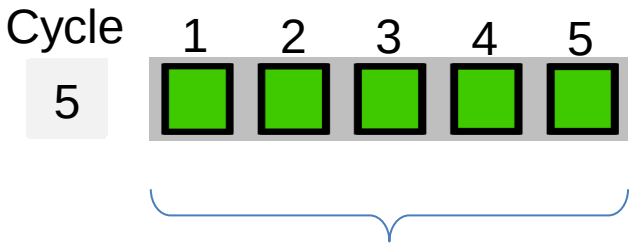
Instruction level parallelism (ILP): pipelining, superscalarity

Pipelining

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



Single instruction takes 5 cycles (latency)



Throughput:

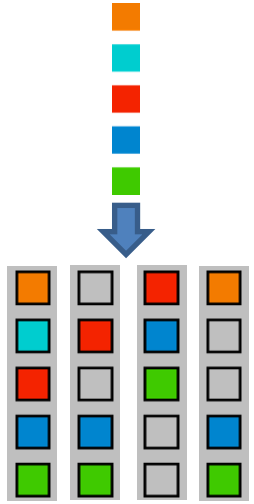
1 instruction per cycle after pipeline is full

→ 5x speedup

Superscalar execution

across multiple pipelines

4-way superscalar:



→ Massive boost in instruction throughput

→ Instructions can be reordered on the fly

Superscalar out-of-order execution and steady state

Instruction execution

STORE (Latency: 2 cy)

LOAD (Latency: 4 cy)

ADD (Latency: 3cy)

```
for(int i=1; i<n; ++i)
        a[i] = a[i] + c;
```

Cycle 1	load a[1]		
Cycle 2	load a[2]		
Cycle 3	load a[3]		
Cycle 4	load a[4]		
Cycle 5	load a[5]	add a[1]=c,a[1]	
Cycle 6	load a[6]	add a[2]=c,a[2]	
Cycle 7	load a[7]	add a[3]=c,a[3]	
Cycle 8	load a[8]	add a[4]=c,a[4]	store a[1]
Cycle 9	load a[9]	add a[5]=c,a[5]	store a[2]
Cycle 10	load a[10]	add a[6]=c,a[6]	store a[3]
Cycle 11	load a[11]	add a[7]=c,a[7]	store a[4]
Cycle 12	load a[12]	add a[8]=c,a[8]	store a[5]
Cycle 13	load a[13]	add a[9]=c,a[9]	store a[6]
Cycle 14	load a[14]	add a[10]=c,a[10]	store a[7]
Cycle 15	load a[15]	add a[11]=c,a[11]	store a[8]
Cycle 16	load a[16]	add a[12]=c,a[12]	store a[10]
…	…	…	…

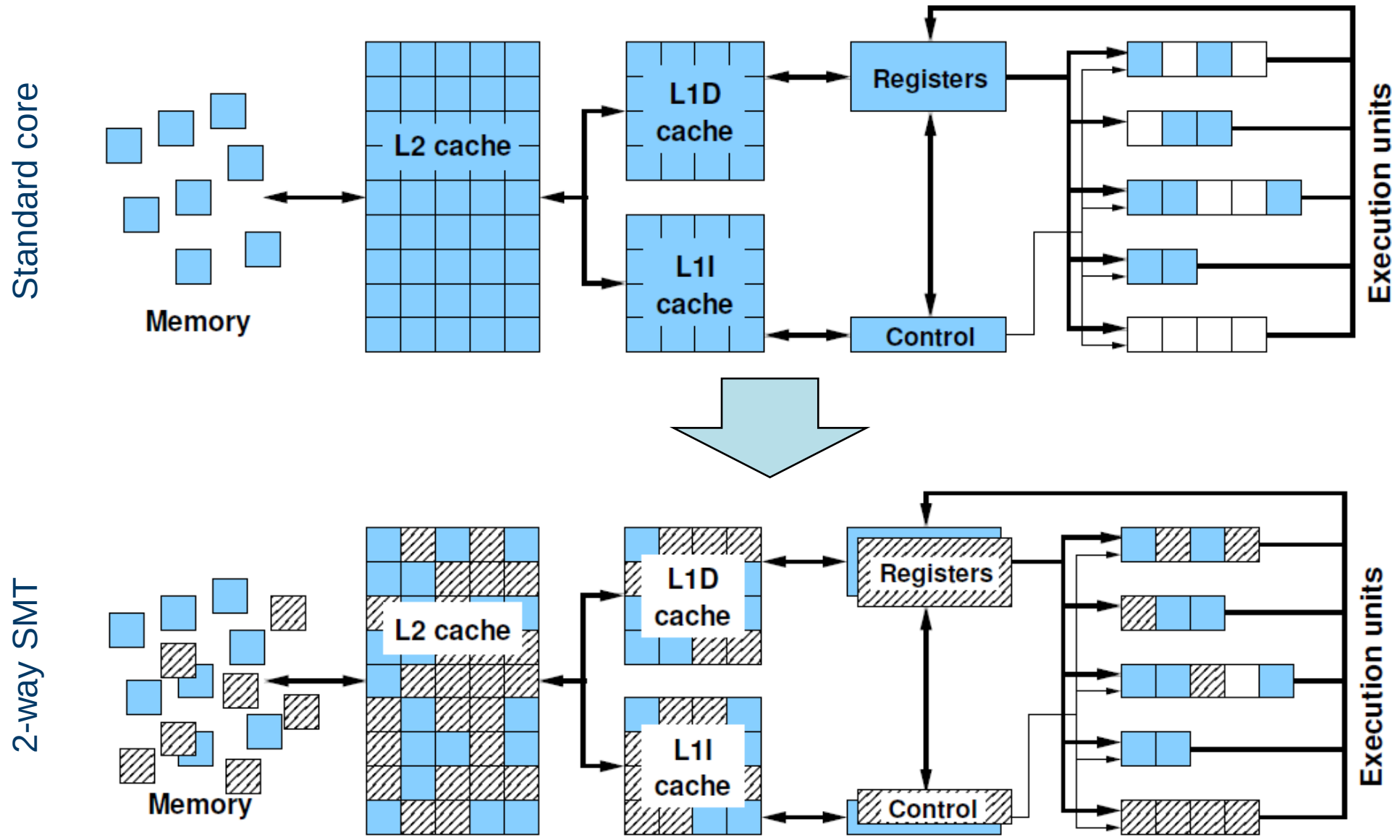
"Steady state:"
3 instructions/cy
("3-way superscalar execution")

Instructions Per Cycle: IPC=3

Cycles Per Instruction: CPI=0.33

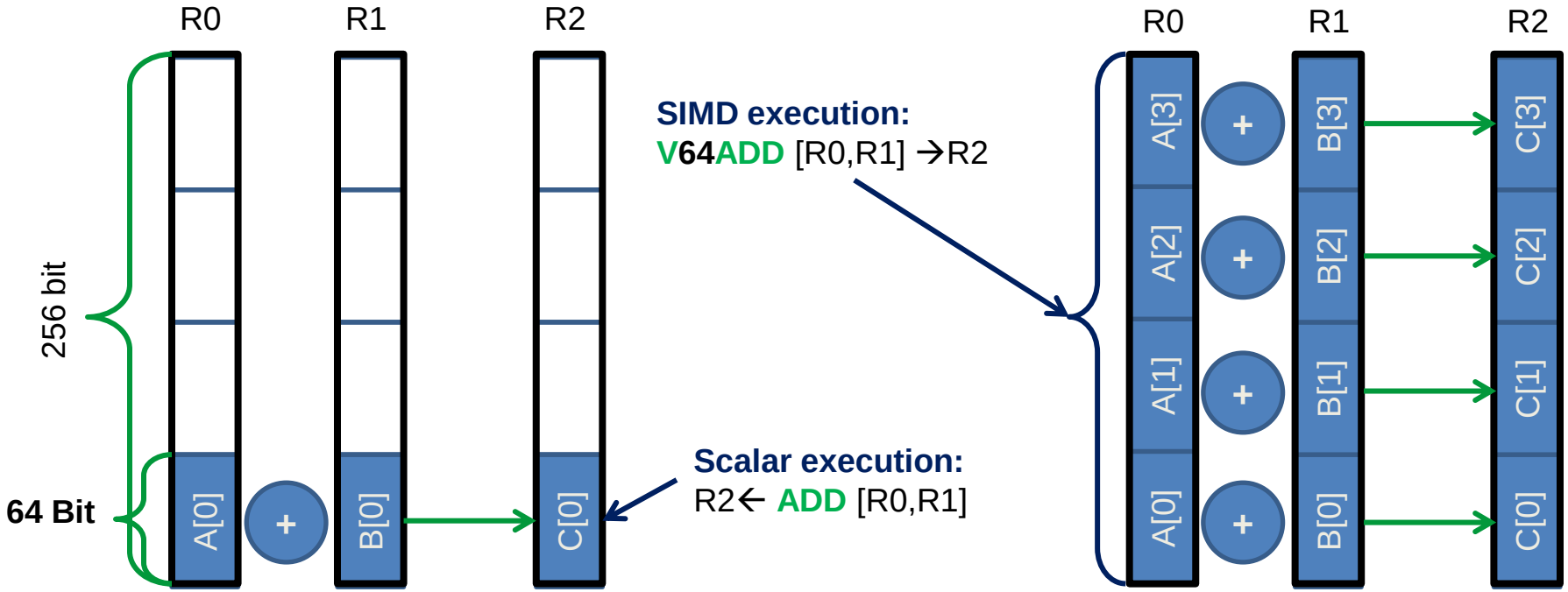
Hardware takes care of executing instructions as soon as their operands are available:
Out-Of-Order (OOO) execution

Simultaneous multi-threading (SMT)

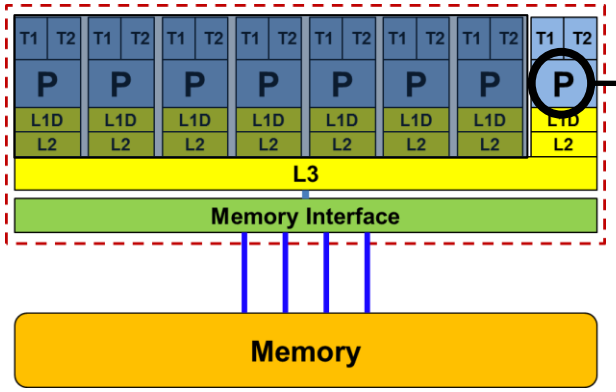


SIMD processing

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



Single-core DP floating-point performance



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

Super-scalarity

FMA factor

SIMD factor

Clock Speed

Name	$n_{\text{super}}^{\text{FP}}$ [instr/cy]	n_{FMA} [flops/lane]	n_{SIMD} [lanes/instr]	Introd.	f [Gcy/s]	$P_{\text{core}}^{\text{DP}}$ [Gflop/s]
Intel Westmere	2	1	2	Q1/10	2.66	10.6
Intel Ivy Bridge	2	1	4	Q3/13	2.2	17.6
IBM Power8	2	2	2	Q2/14	2.93	23.4
Intel Broadwell	2	2	4	Q1/16	2.3	36.8
Intel Knights Landing	2	2	8	Q2/16	1.3	41.6
Intel Skylake	2	2	8	Q3/17	2.4	76.8
AMD Zen 2 (Rome)	2	2	4	Q3/19	2.25	36.0
Fujitsu A64FX	2	2	8	Q2/20	2.2	70.4
AMD Zen 4 (Genoa)	2	2	4	Q3/22	2.4	38.4
Intel Sapphire Rapids	2	2	8	Q1/23	2.0	64.0
NVIDIA Grace	4	2	2	Q2/23	3.4	54.4

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Example: The sum reduction

A “simple” example: The sum reduction

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

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

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

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

→No pipelining?

→No SIMD?

Applicable peak for the sum reduction (I)

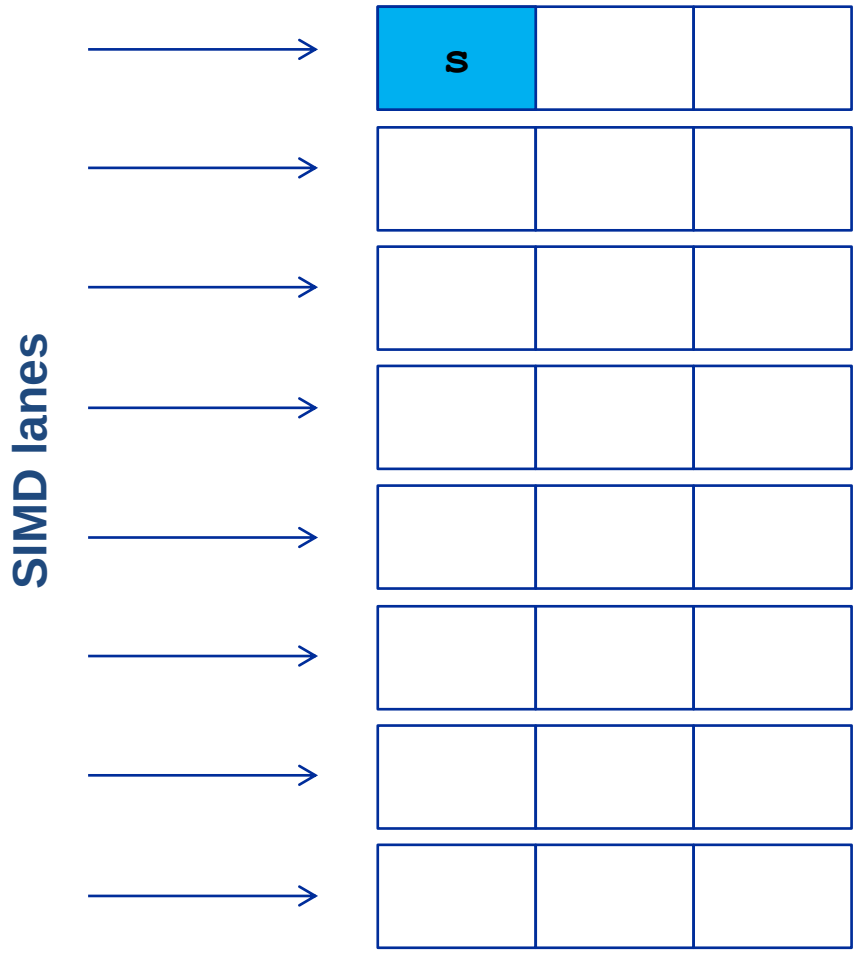
Plain scalar code, no SIMD

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

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:



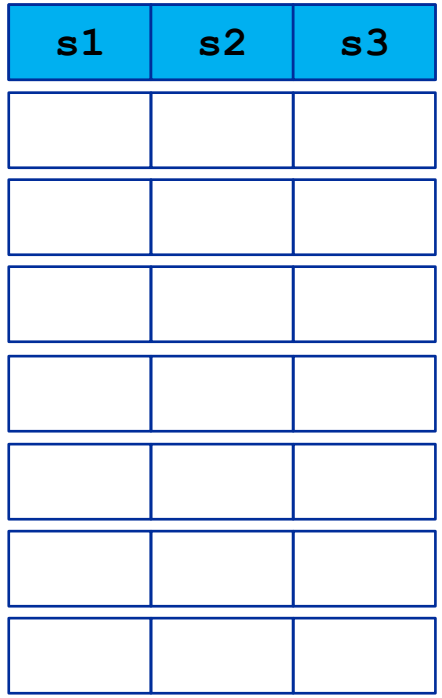
→ 1/24 of ADD peak

Applicable peak for the sum reduction (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;
```



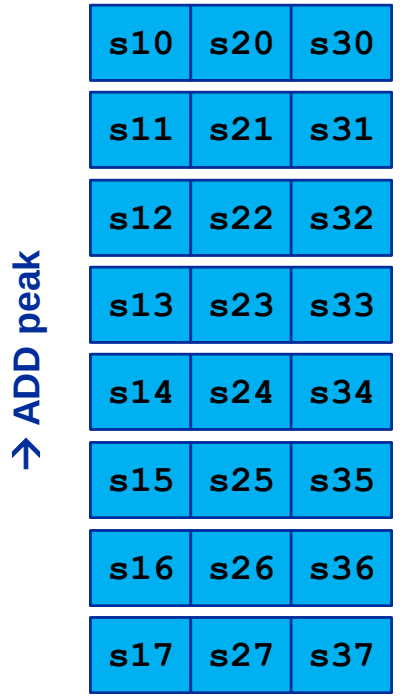
→ 1/8 of ADD peak

Applicable peak for the sum reduction (III)

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

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

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

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

  i+=24 →? loop
result ← r1.0+r1.1+...+r3.6+r3.7
```

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



Sum reduction

Questions

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

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

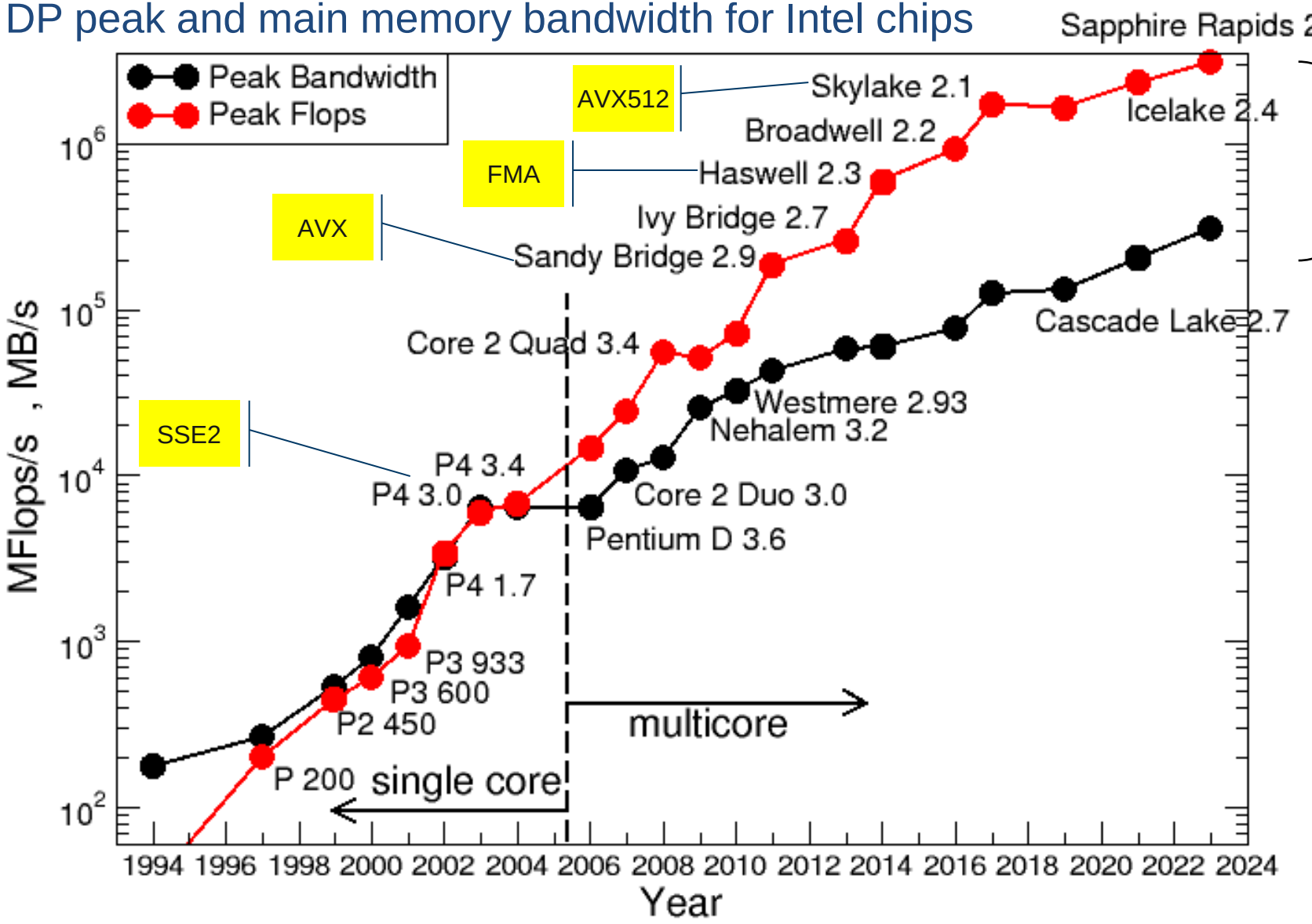
Memory Hierarchy

In-cache performance (L2, L3)

Main memory performance

Von Neumann bottleneck reloaded: “DRAM gap”

DP peak and main memory bandwidth for Intel chips



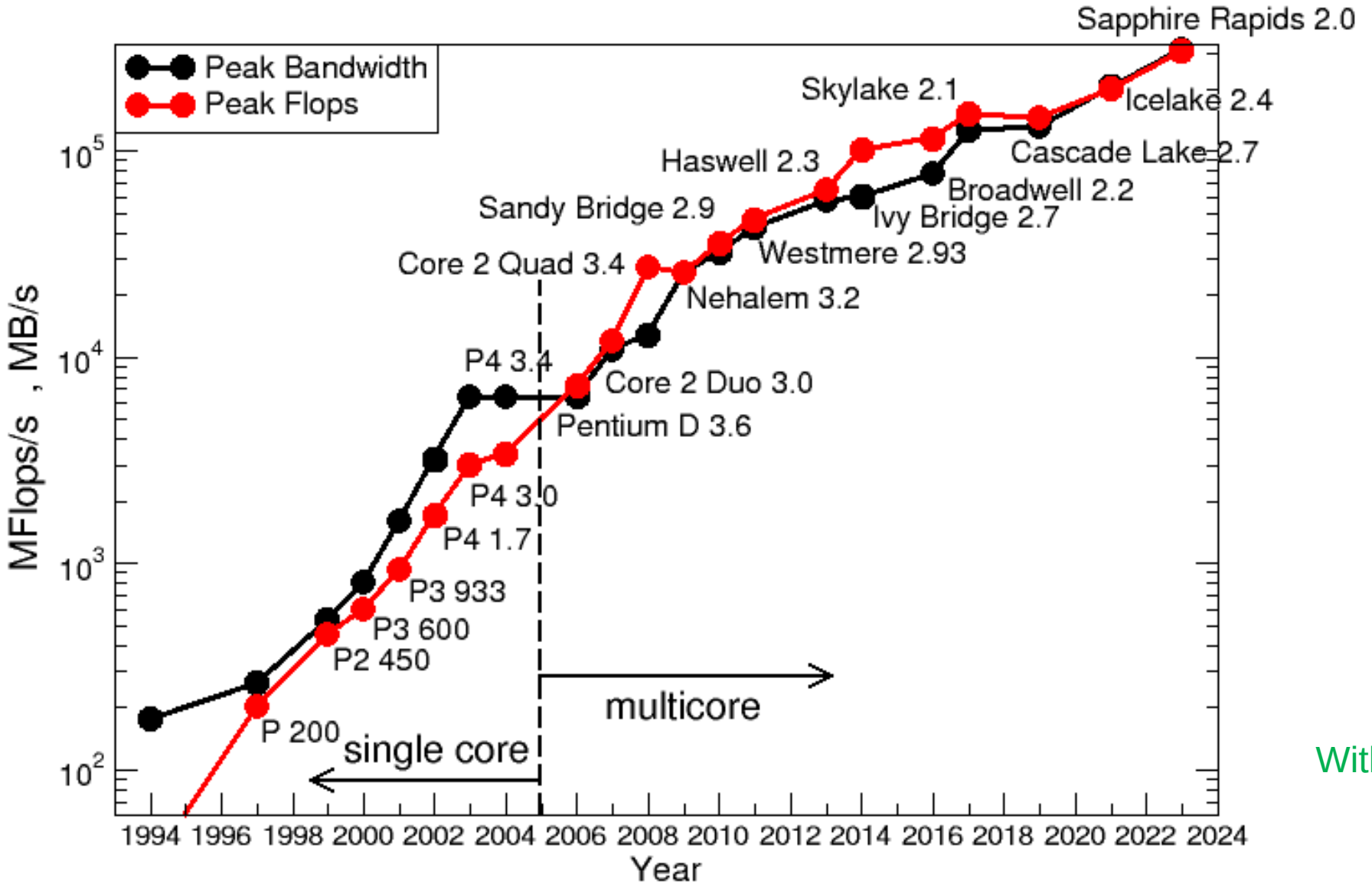
≥ 10 F/B

Main memory access speed not sufficient to keep CPU busy…

Main drivers of gap: SIMD, FMA

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

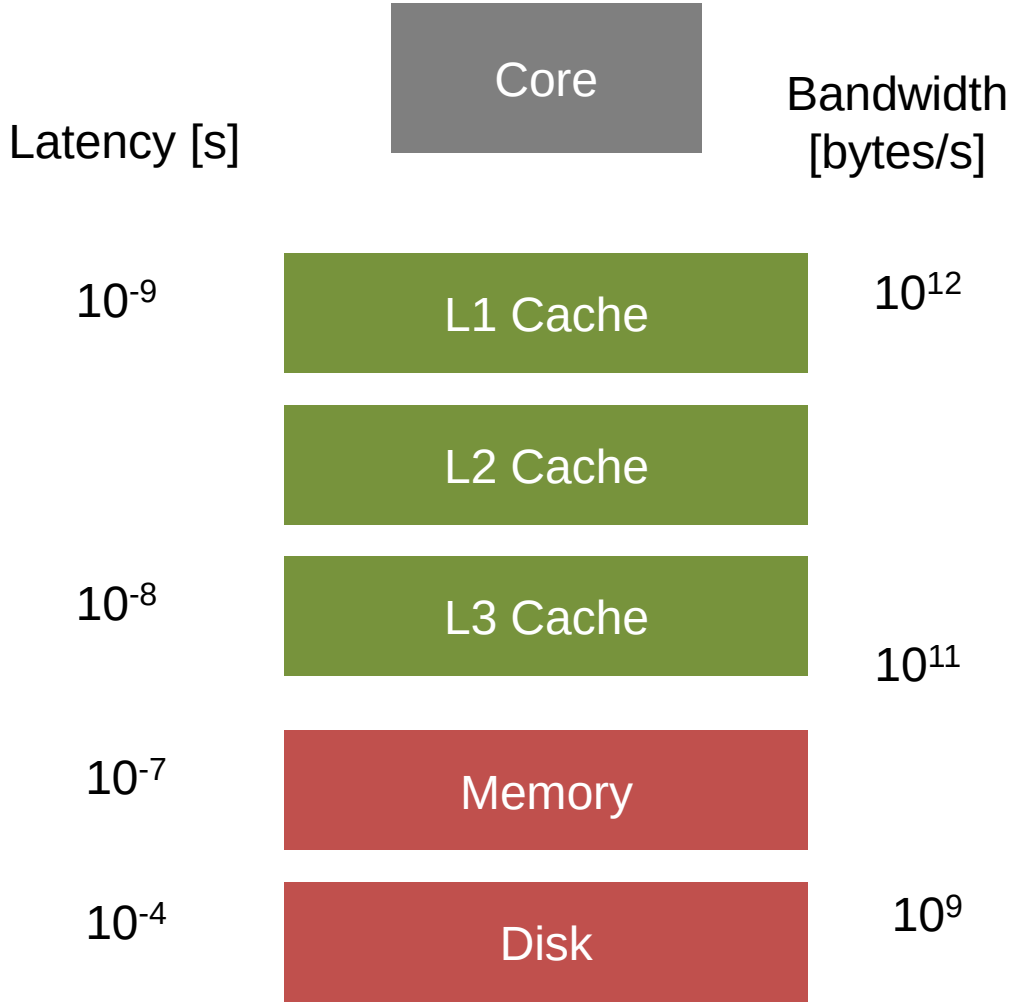
The “stripped” von Neumann bottleneck



Without SIMD, FMA: no gap!

Memory hierarchy

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

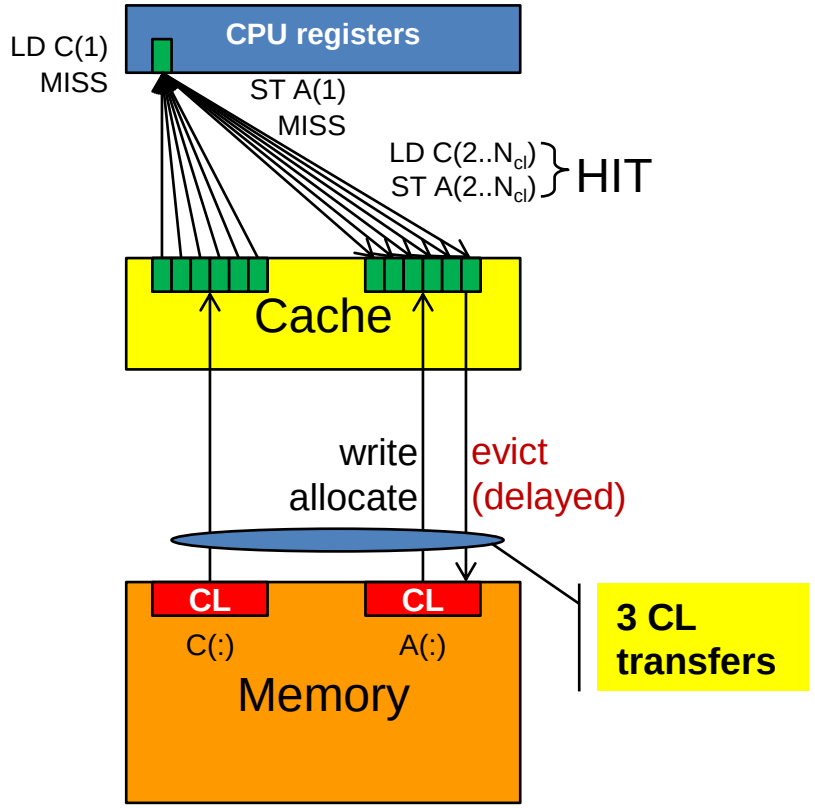


Purpose of many optimizations: use data in fast memory

Data transfers in a memory hierarchy

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

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



- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- Registers can only “talk” to the L1 cache
- MISS: Load or store instruction does not find the data in acache level
  → CL transfer required

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

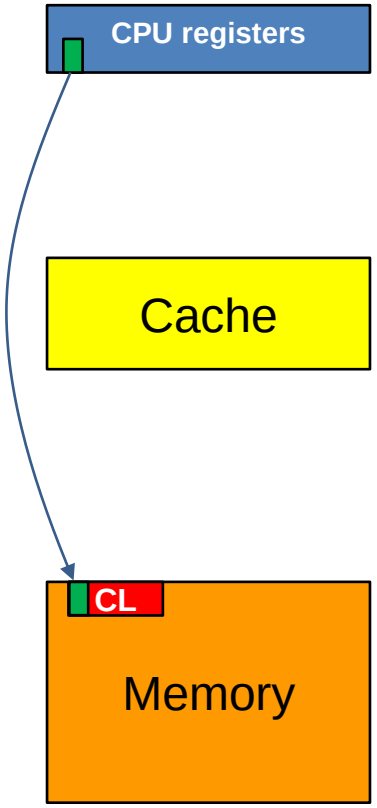
Avoiding the write-allocate transfer

Disadvantages of write-allocate:

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

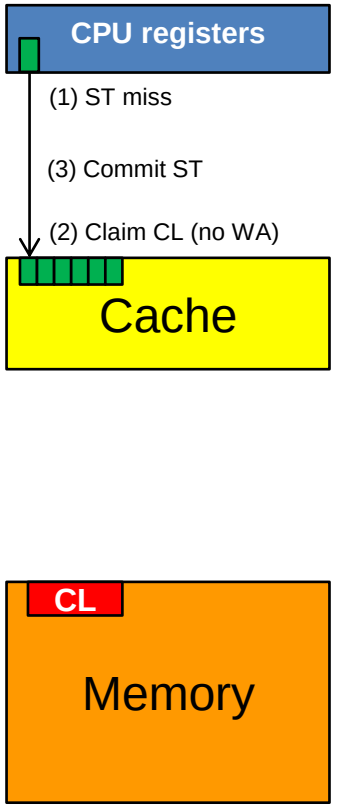
Solution 1:
Nontemporal stores

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

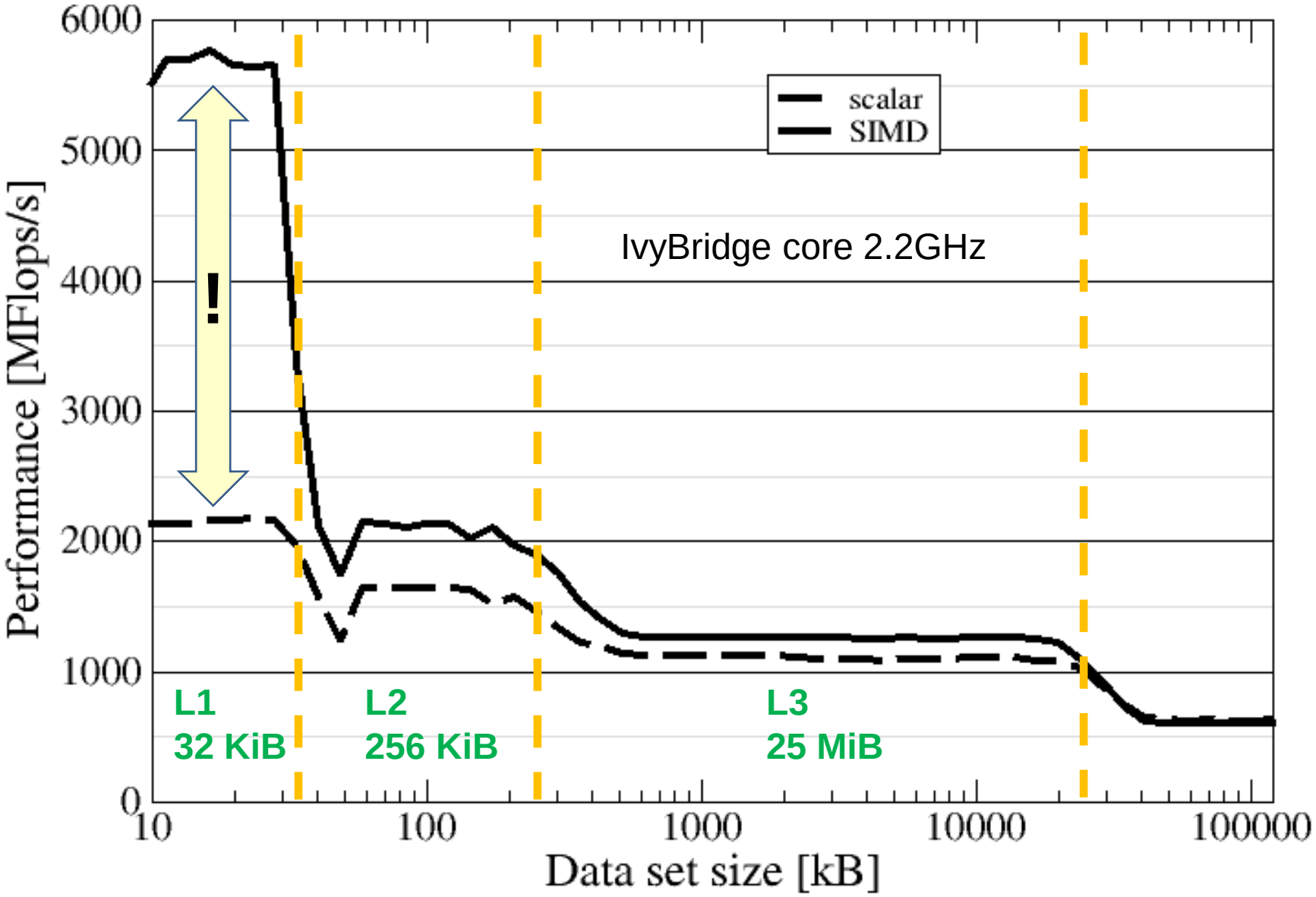


Solution 2:
Cache line claim

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



Getting the data from far away




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

Varying loop length,
repeat many times

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Multicore Chips

Memory bandwidth scaling

Node topology and performance

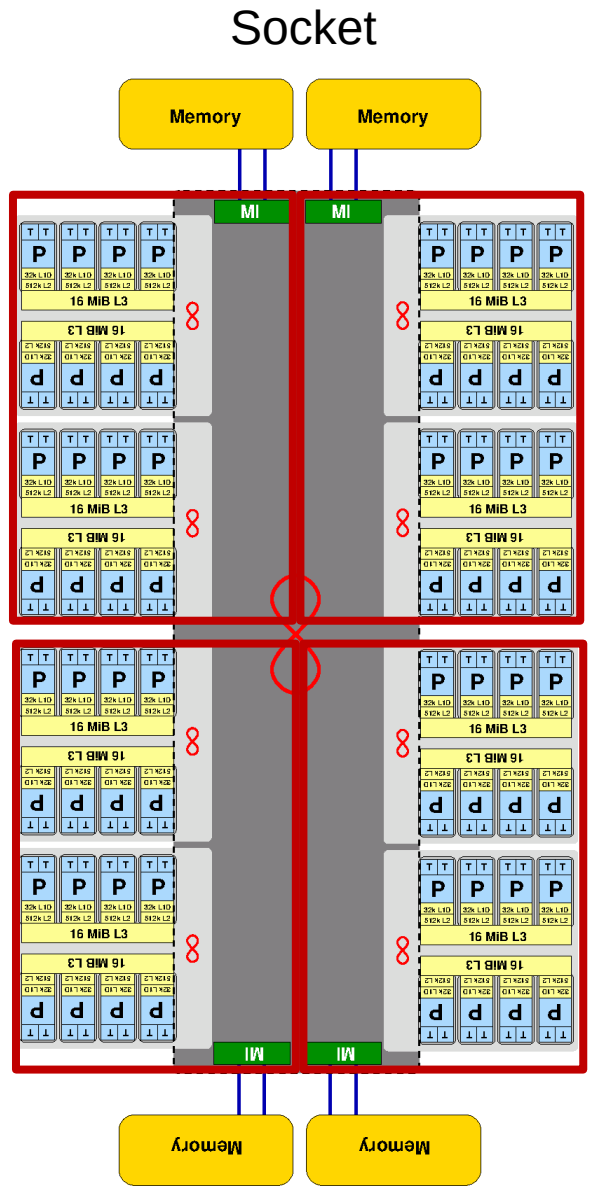
Putting the cores & caches together
AMD Epyc 7742 64-Core Processor («Rome»)

- Core features:
  - Two-way SMT
  - Two 256-bit SIMD FMA units (AVX2)
    →16 flops/cycle
  - 32 KiB L1 data cache per core
  - 512 KiB L2 cache per core

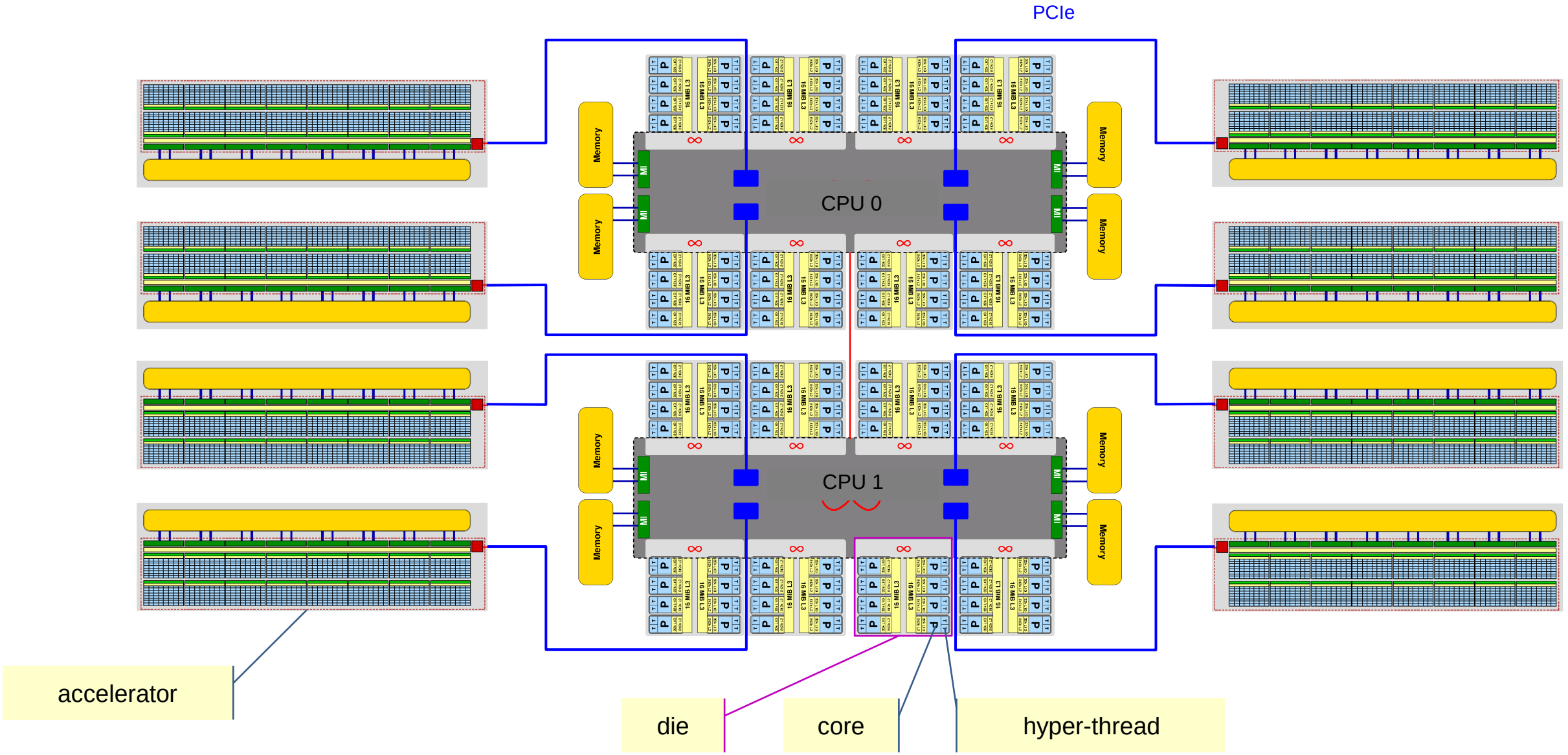
- 64 cores per socket hierarchically built up from
  - 16 CCX with 4 cores and 16 MiB of L3 cache
  - 2 CCX form 1 CCD (silicon die)
  - 8 CCDs connected to IO device “Infinity Fabric” (memory controller & PCIe)

- 8 channels of DDR4-3200 per IO device
  - MemBW: 8 ch x 8 byte x 3.2 GHz = 204.8 GB/s

- ccNUMA feature (boot time option):
  - Nodes Per Socket (NPS)=1 , 2 or 4
  - **NPS=4 → 4 ccNUMA domains**



Adding accelerators to the node



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Interlude: A glance at accelerator technology

NVIDIA “Hopper” H100

vs.

AMD Zen4 “Genoa”

Nvidia H100 “Hopper” SXM5 (700 W) specs

Architecture

- 80 B Transistors
- ~ 2.0 GHz clock speed (turbo)
- ~ 132 “SM” units
  - 128 SP “cores” each (FMA)
  - 64 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance

© Nvidia

- ~ 34 TFlop/s DP peak (FP64 - turbo)
- 50 MiB L2 Cache

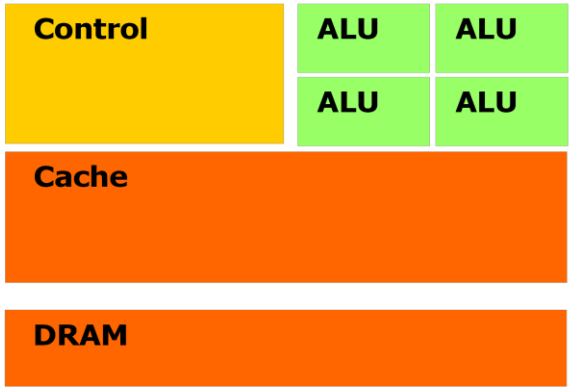
- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)

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

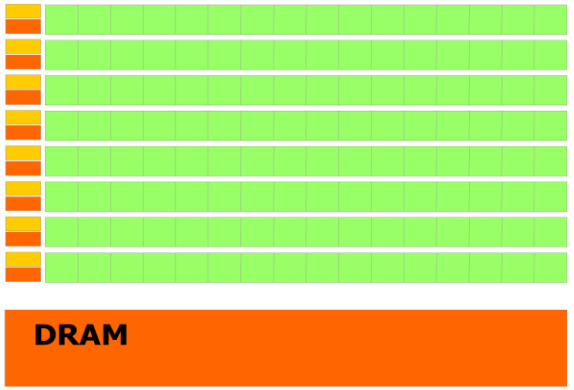
# SMs	# CUDA cores/SM	# FP ops/cy

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

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



CPU



GPU

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

Conclusions about architecture

- Performance is a result of
  - How **many instructions** you require to implement an algorithm
  - How **efficiently** those instructions are **executed** on a processor
  - Runtime contribution of the triggered **data transfers**
- Modern computer architecture has a rich “topology”
- Node-level hardware parallelism takes many forms
  - Sockets/devices – CPU: 1-4 or more, GPGPU: 1-8
  - Cores – moderate (CPU: 20-128, GPGPU: 10-100)
  - SIMD – moderate (CPU: 2-16) to massive (GPGPU: 10’s-100’s)
  - Superscalarity (CPU: 2-6)
- Performance of programs is sensitive to architecture
  - Topology/affinity influences overheads of popular programming models
  - Standards do not contain (many) topology-aware features
    - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
  - Apart from overheads, performance features are largely independent of the programming model

NHR FAU

Multicore Performance and Tools

Part 1: Topology, affinity, 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, Slurm, numactl, ***likwid-pin***
- **Runtime Profiling**
  Compilers, gprof, perf, HPC Toolkit, Intel Amplifier, …
- **Performance Analysis**
  Intel VTune, ***likwid-perfctr****, PAPI-based tools, HPC Toolkit, Score-P Tools, Linux perf*
- **Microbenchmarking**
  STREAM, ***likwid-bench****, lmbench, uarch-bench*

LIKWID performance tools

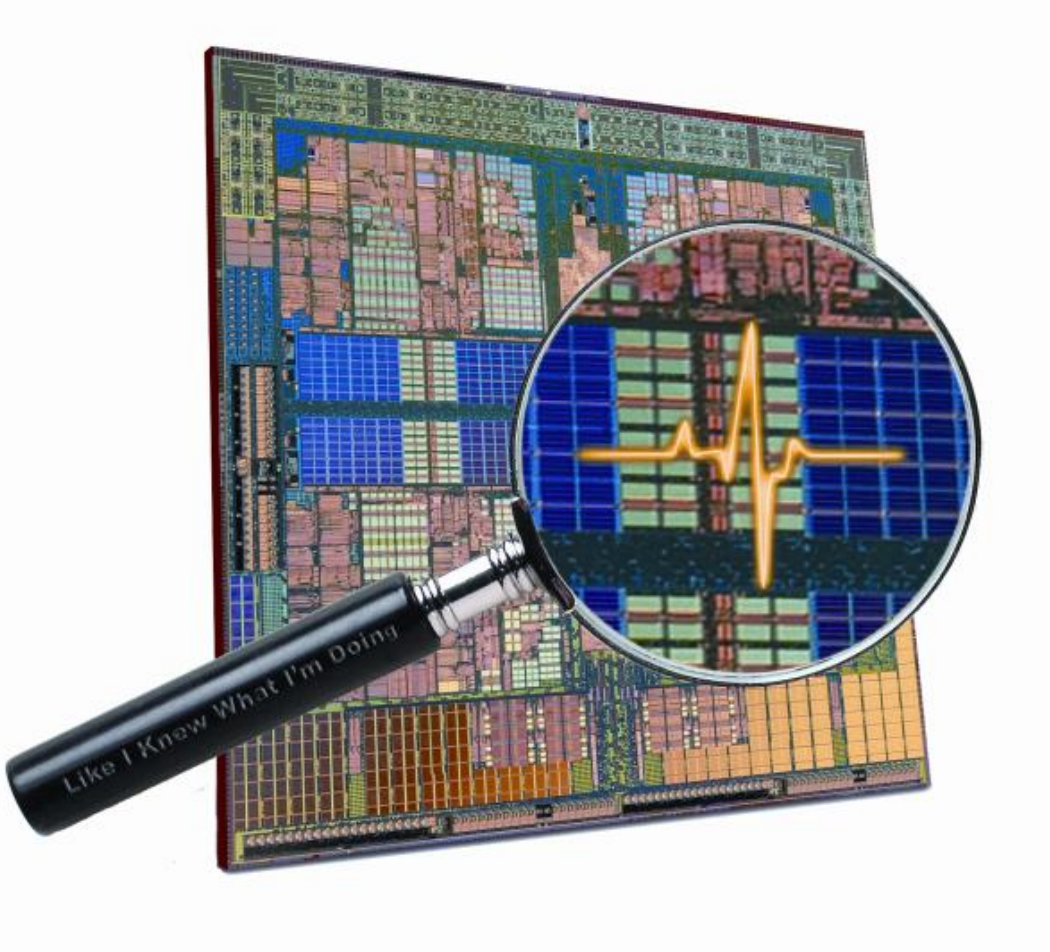
LIKWID tool suite:

Like
I
Knew
What
I'm
Doing

https://youtu.be/6uFl1HPq-88

Open source tool collection
(developed at RRZE):

https://github.com/RRZE-HPC/likwid



J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* PSTI2010, Sep 13-16, 2010, San Diego, CA. DOI: 10.1109/ICPPW.2010.38

LIKWID Tool Suite

- Command line tools for Linux:

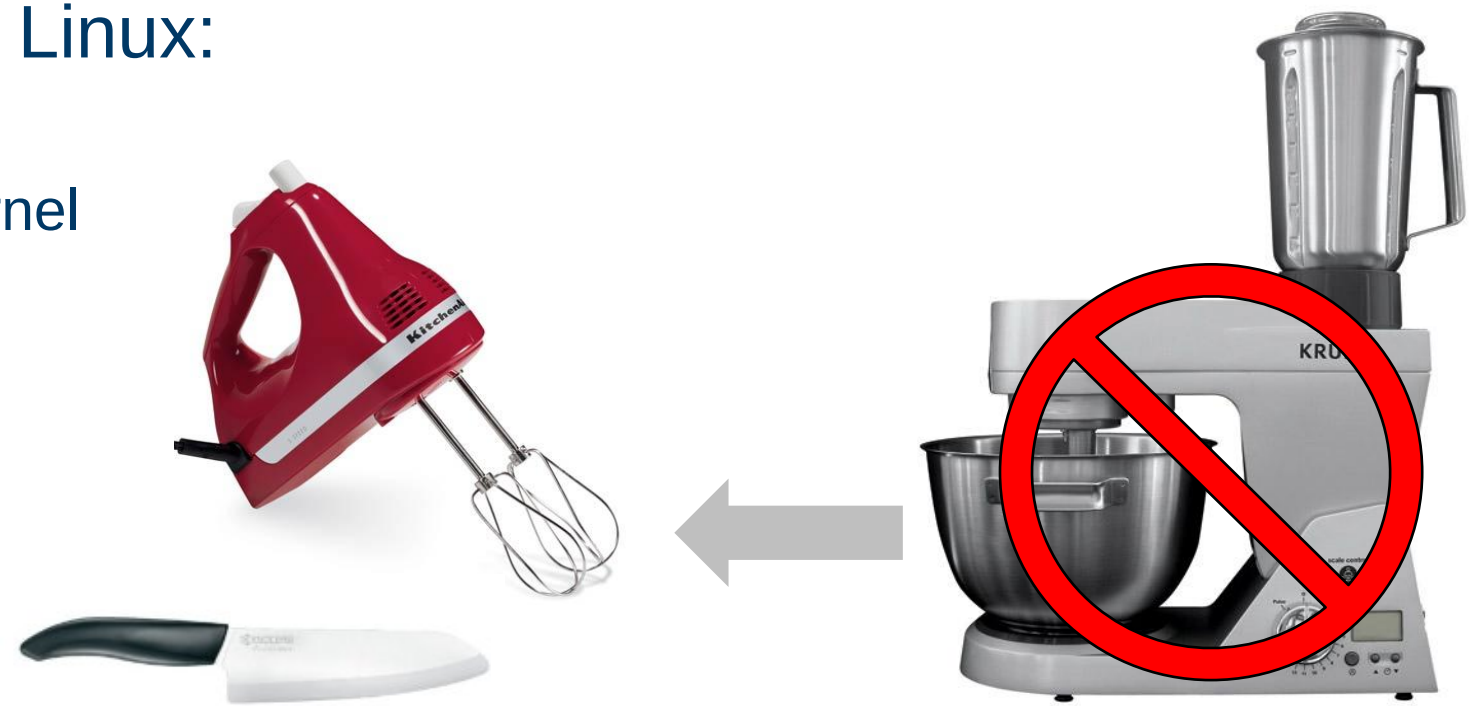
  easy to install

  works with standard Linux kernel

  simple and clear to use

  supports most X86 CPUs

  (also ARMv8, POWER9 and Nvidia GPUs)



- Current tools:

likwid-topology - Print thread and cache topology

likwid-pin - Pin threaded application without touching code

likwid-perfctr - Measure performance counters

likwid-powermeter - Measure energy consumption

likwid-bench - Microbenchmarking tool and environment

… some more

Reporting topology

likwid-topology

https://youtu.be/mxMWjNe73SI

Output of `likwid-topology`

on one node of Intel Icelake SP

```
$ 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
0               0               0           0           0             *
1               0               1           0           0             *
[…]
70              0               70          0           1             *
71              0               71          0           1             *
--------------------------------------------------------------------------------
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 )
--------------------------------------------------------------------------------
********************************************************************************
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
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 )
--------------------------------------------------------------------------------
```

All OS hardware thread IDs

Output of `likwid-topology` continued

```
********************************************************************************
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!

Output of `likwid-topology -g`

Intel Haswell EP, 14 cores per socket
Cluster on Die (CoD) mode and SMT enabled!

```
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  0 28  | |  1 29  | |  2 30  | |  3 31  | |  4 32  | |  5 33  | |  6 34  | |  7 35  | |  8 36  | |  9 37  | | 10 38  | | 11 39  | | 12 40  | | 13 41  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------+ +-------------------------------------------------------------------------+ |
| |                                   17MB                                  | |                                   17MB                                  | |
| +-------------------------------------------------------------------------+ +-------------------------------------------------------------------------+ |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
Socket 1:
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| | 14 42  | | 15 43  | | 16 44  | | 17 45  | | 18 46  | | 19 47  | | 20 48  | | 21 49  | | 22 50  | | 23 51  | | 24 52  | | 25 53  | | 26 54  | | 27 55  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |  32kB  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------+ +-------------------------------------------------------------------------+ |
| |                                   17MB                                  | |                                   17MB                                  | |
| +-------------------------------------------------------------------------+ +-------------------------------------------------------------------------+ |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

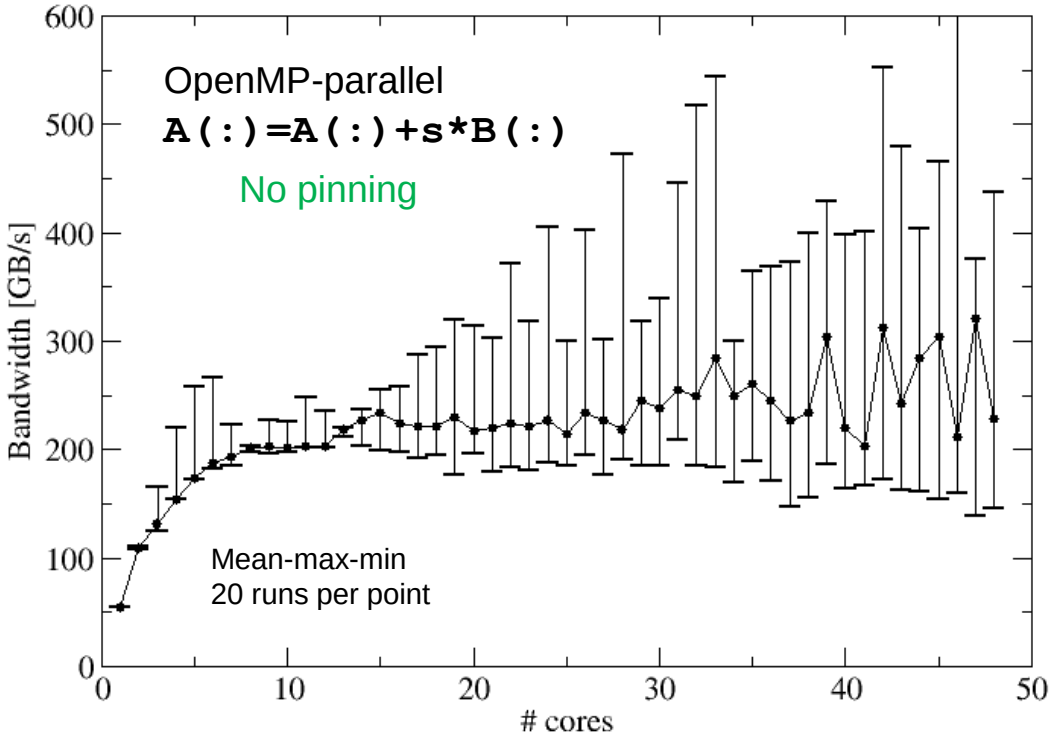
Enforcing thread/process affinity under the Linux OS

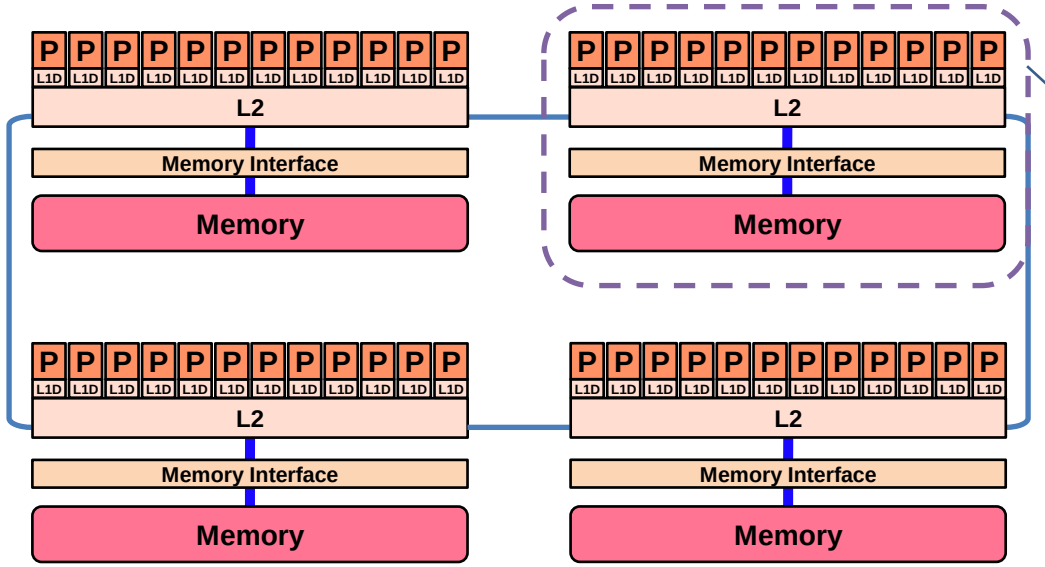
likwid-pin

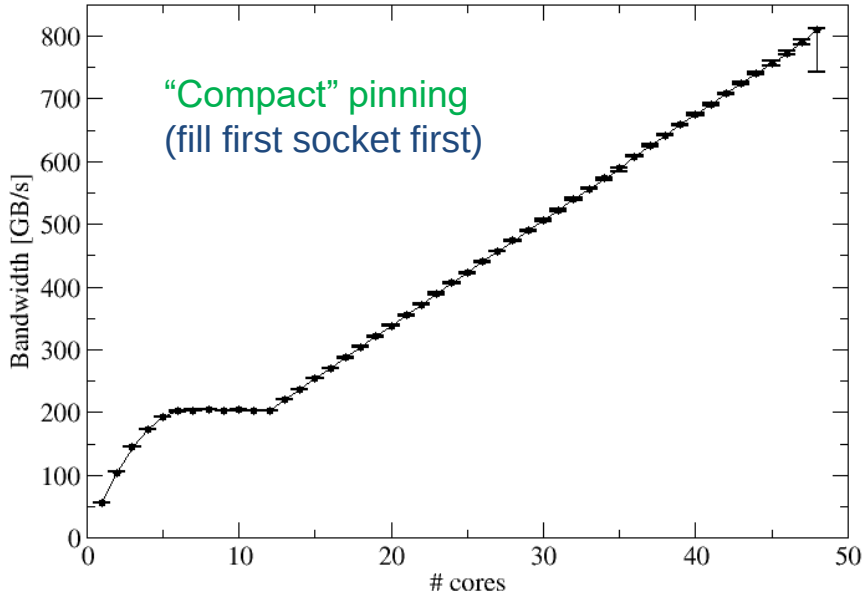
https://youtu.be/PSJKNQaqwB0

DAXPY test on A64FX

Anarchy vs. thread pinning



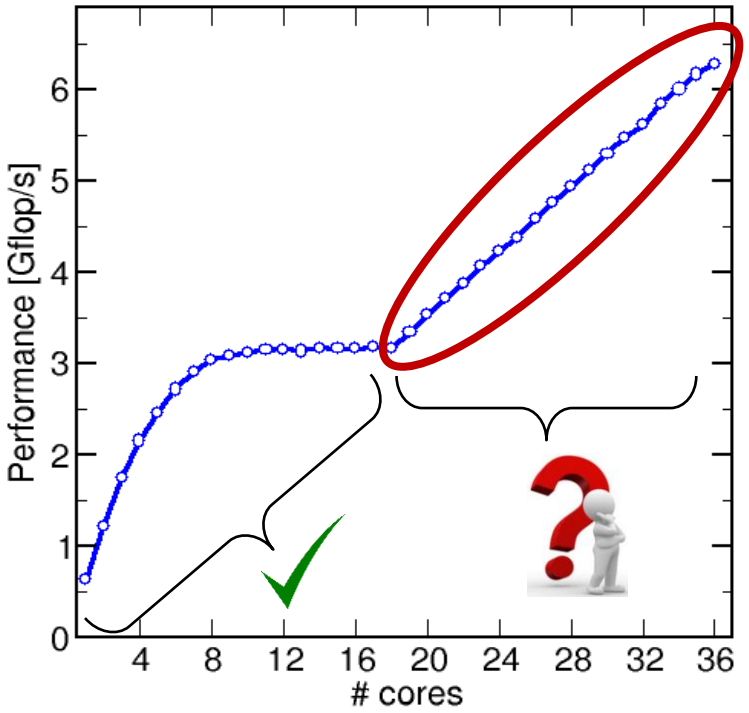




There are several reasons for caring about affinity:

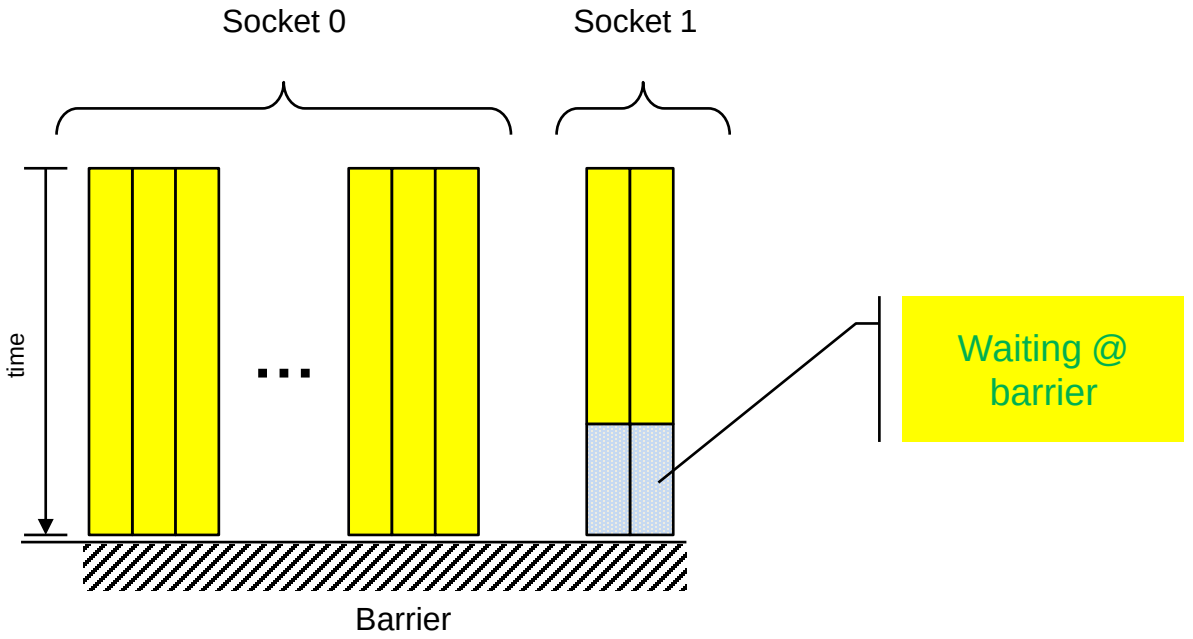
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

Interlude: Why the weird scaling behavior?




```
!$omp parallel do schedule(static)
 do i = 1,N
   a(i) = b(i) + s * c(i)
!$omp end parallel do
```

implicit barrier



- Every thread has the same workload
- Performance of left socket is saturated
- Barrier enforces waiting of “speeders” at sync point
- Average performance of each “right” core == average performance of each “left” core → linear scaling

More thread/process affinity (“pinning”) options

- Highly OS-dependent system calls but available on all systems
  - Linux: **`sched_setaffinity()`**
  - Windows: **`SetThreadAffinityMask()`**
  - OS X: **`thread_policy_set()`**
- 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, …)

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 core IDs (“thread groups”):

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

Physical HW threads first!

```
+-----------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  4| |  1  5| | 2  6 | | 3  7 | |
| +------+ +------+ +------+ +------+ |
+-----------------------------------+
```

- Example for likwid-pin:

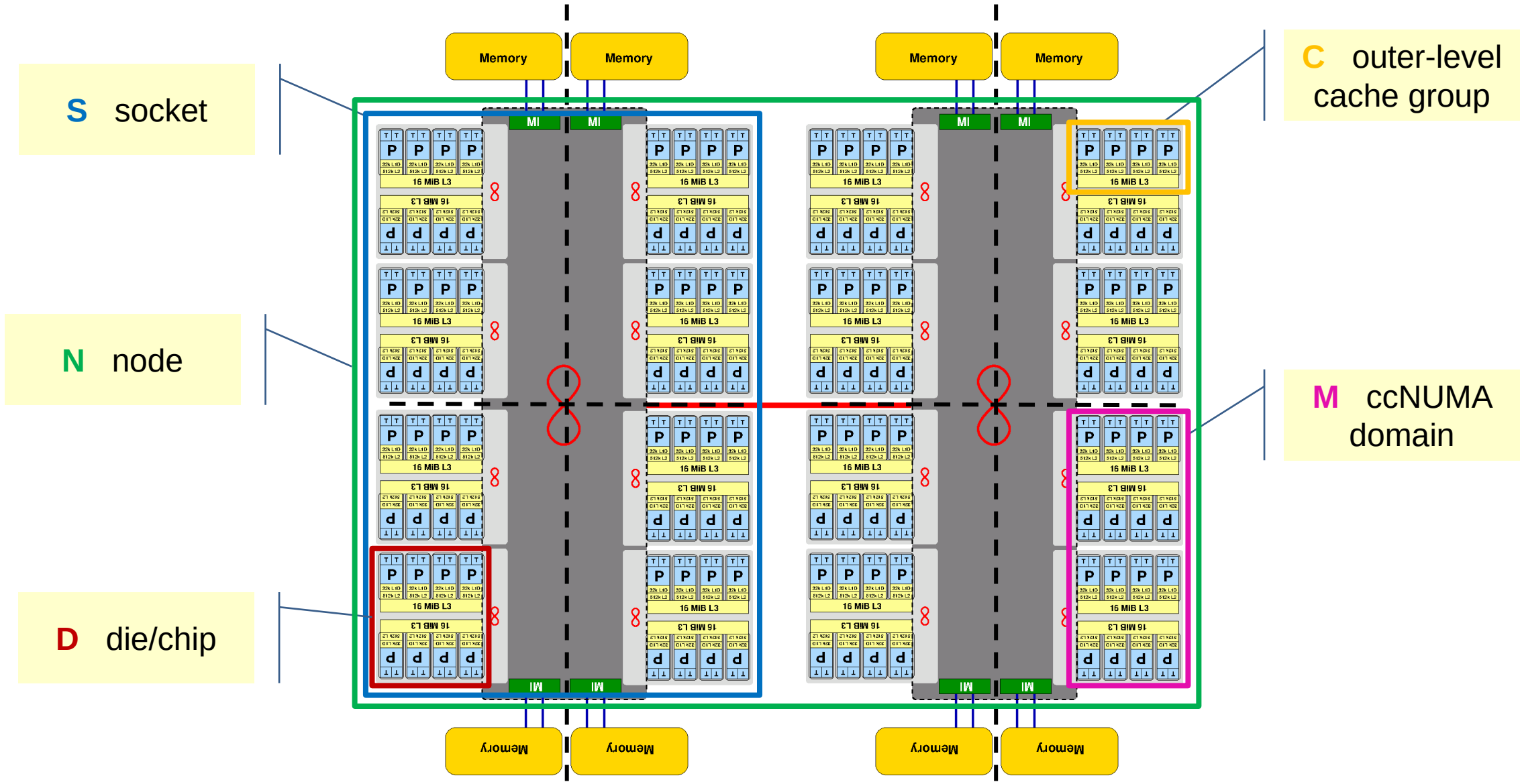
```
$ likwid-pin -c S0:0-3 ./a.out
```

- Thread group expressions may be chained with `@`:

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

```
+-----------------------------------++-----------------------------------+
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
| | 0  8 | | 1  9 | | 2 10 | | 3 11 | || | 4 12 | | 5 13 | | 6 14 | | 7 15 | |
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
+-----------------------------------++-----------------------------------+
```

Available thread domains/unit prefixes (as of LIKWID 5.3)



Advanced options for pinning: Expressions

- Expressions are more powerful in situations where the pin mask would be very long or clumsy

  Compact pinning (counting through hardware threads):

```
$ likwid-pin -c E:<thread domain>:\
    <number of threads>\
    [:<chunk size>:<stride>] ...
$ likwid-pin -c E:N:4
```

“Compact” placement!

```
+-----------------------------------+
| +------+ +------+ +------+ +------+ |
| |  0  4| |  1  5| | 2  6 | | 3  7 | |
| +------+ +------+ +------+ +------+ |
+-----------------------------------+
```

  Scattered pinning across all domains of the designated type:

```
$ likwid-pin -c <domaintype>:scatter
```

- Examples:

```
$ likwid-pin -c E:N:8:1:2 ...
```

```
+-----------------------------------++-----------------------------------+
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
| | 0  8 | | 1  9 | | 2 10 | | 3 11 | || | 4 12 | | 5 13 | | 6 14 | | 7 15 | |
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
+-----------------------------------++-----------------------------------+
```

- Scatter across all NUMA domains:

```
$ likwid-pin -c M:scatter
```

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

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.

abstract name

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"`

<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 Phi with 72 cores, 32 cores to be used, 2 threads per physical core

```
OMP_NUM_THREADS=64
OMP_PLACES=cores(32)
OMP_PROC_BIND=close    # spread will also do
```

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 (will probably not happen)
- 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:

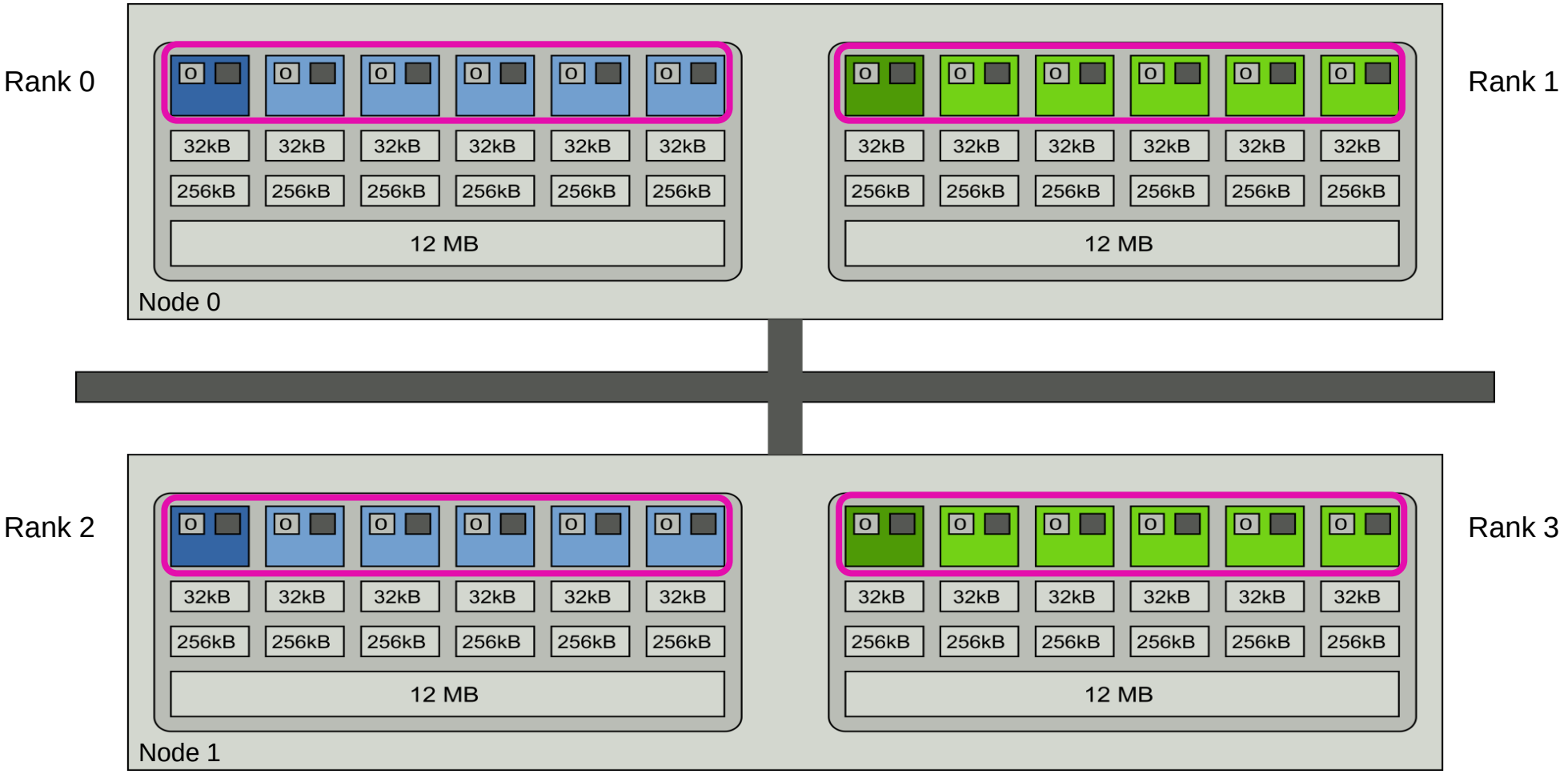
```
$ likwid-mpirun -np 16 -nperdomain S:2 ./a.out
```

Hybrid:

```
$ likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
$ likwid-mpirun -np 16 -nperdomain S:1 -t 2 ./a.out
```

likwid-mpirun 1 MPI process per socket

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



Intel MPI+compiler:

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

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

“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

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

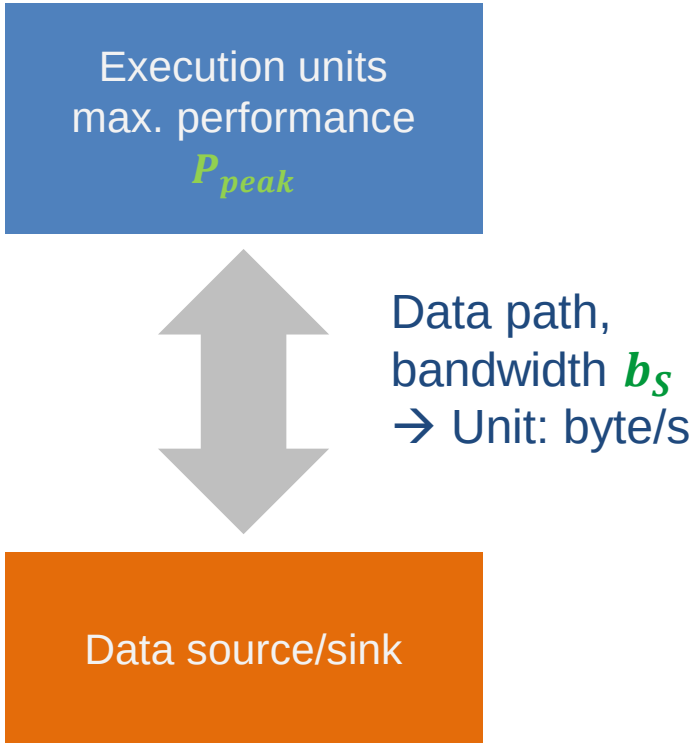
S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

Analytic white-box performance models

An analytic white-box performance model is a simplified mathematical description of the hardware and its interaction with software. It is able to predict the runtime/performance of code from “first principles.”

A simple performance model for loops

Simplistic view of the hardware:



Simplistic view of the software:

```
! may be multiple levels
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? P [flop/s]

The bottleneck is either

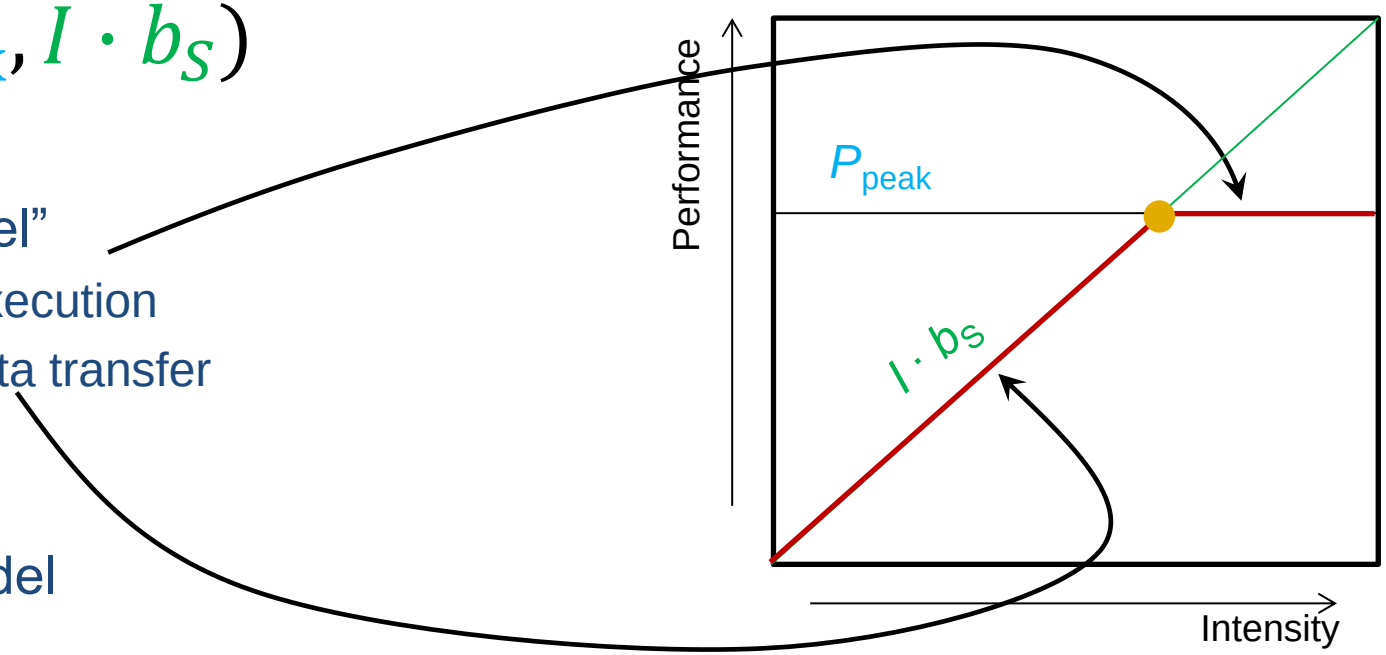
- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

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

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")



The Roofline Model in computing – Basics

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 $\left[\frac{B}{s}\right]$

Machine model

- Code characteristic: Computational intensity: I $\left[\frac{F}{B}\right]$

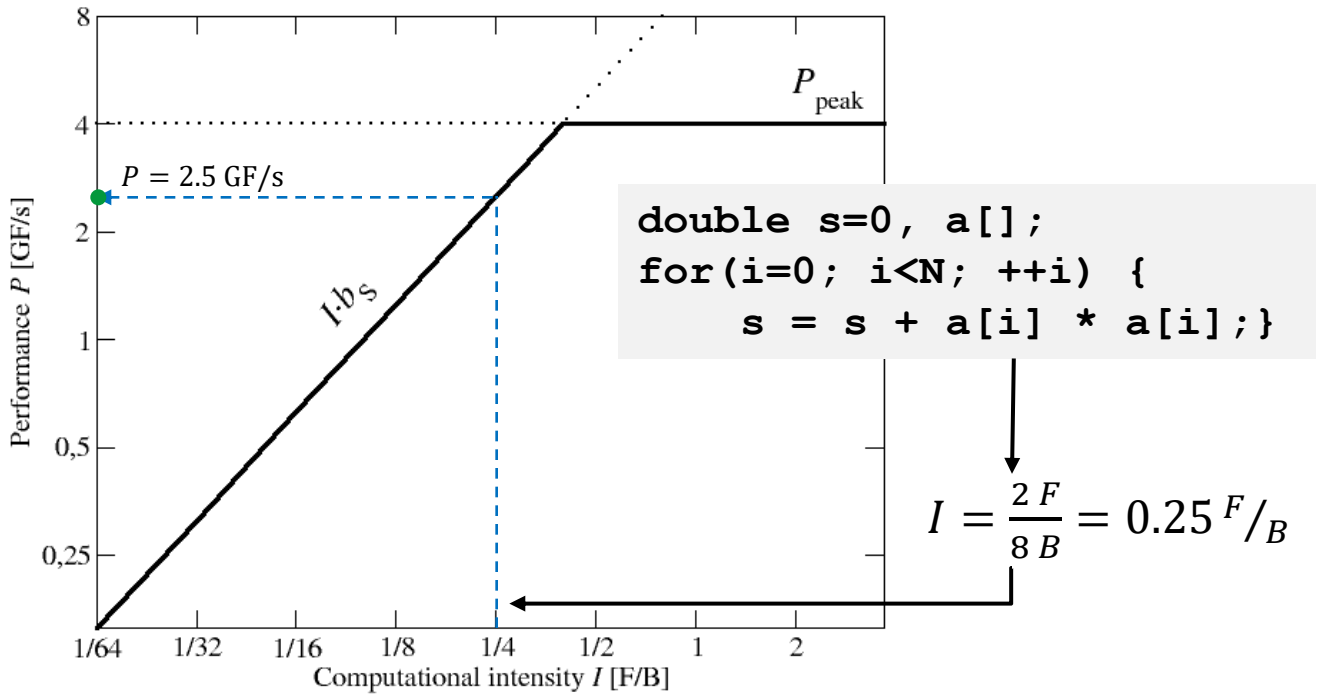
Application model

Machine properties:

$$\boldsymbol{P_{peak}} = 4\,\frac{\mathrm{GF}}{\mathrm{s}}$$

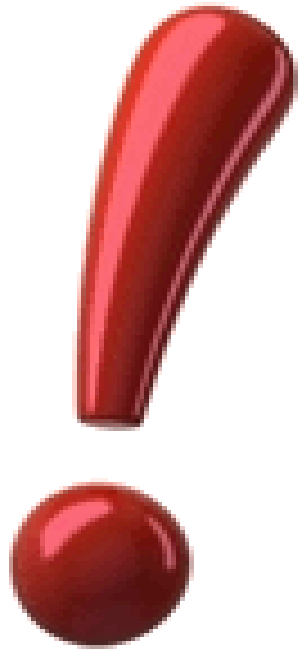
$$\boldsymbol{b_S} = 10\,\frac{\mathrm{GB}}{\mathrm{s}}$$

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

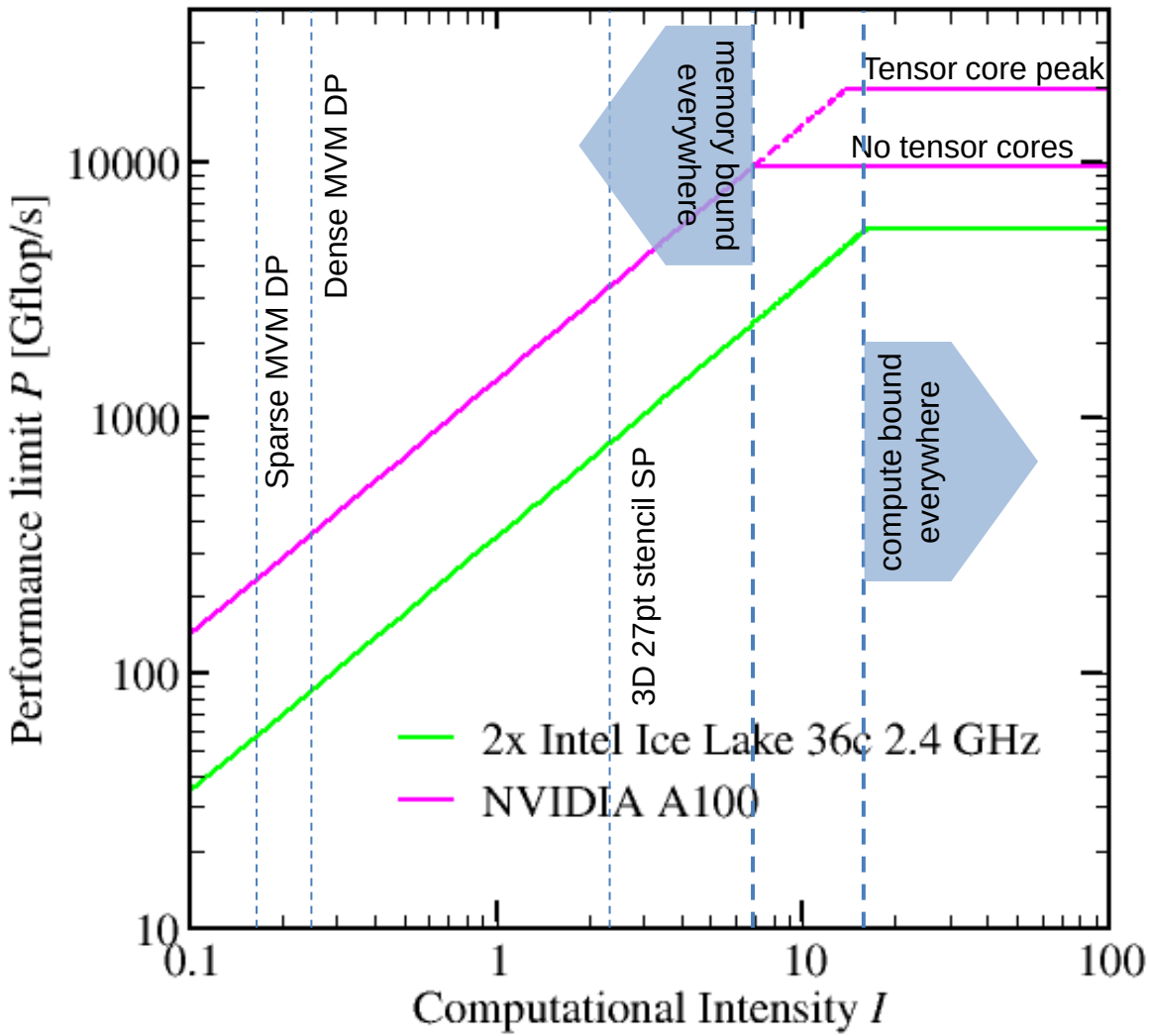


Roofline for architecture and code comparison

With Roofline, we can

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

- Roofline 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

2. b_S = Applicable (saturated) peak bandwidth of the slowest data path utilized
   → e.g., b_S = 56 GByte/s

3. 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!

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

[Byte/s]

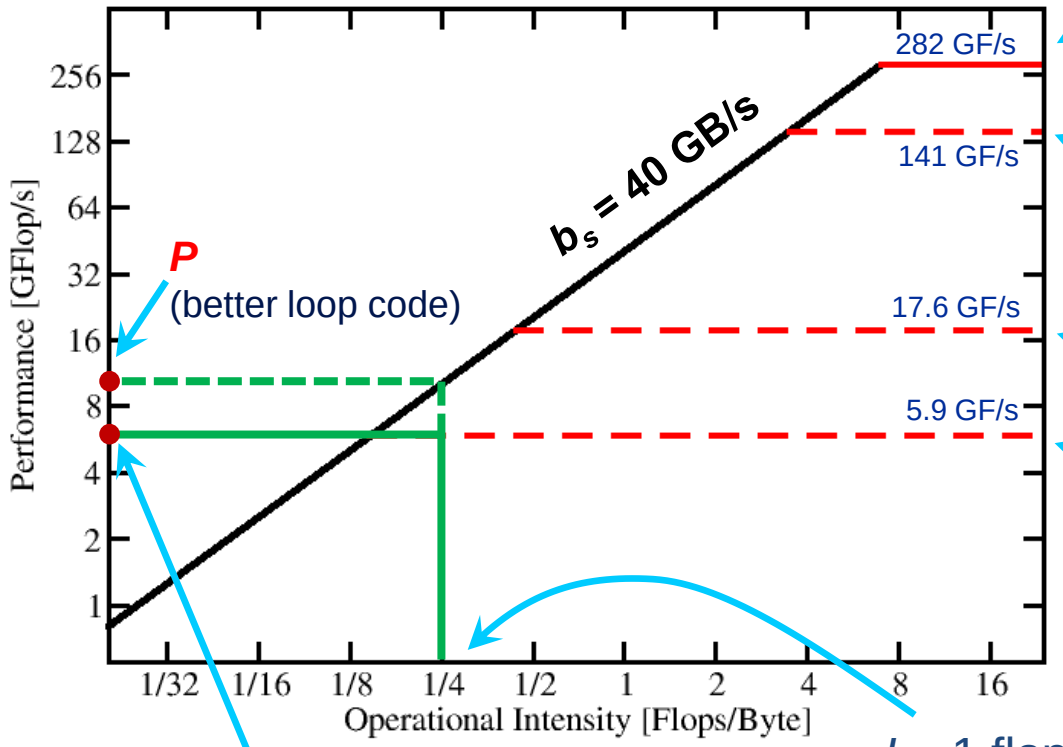
[Byte/Flop]

Full Roofline for the sum reduction from the intro

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

3-cycle latency per ADD if not unrolled

See architecture intro

I = 1 flop / 4 byte (SP!)

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

Code balance: more examples

```
double a[], b[];
for(i=0; i<N; ++i)
    a[i] = a[i] + b[i];
```

B_C = 24B / 1F = 24 B/F

I = 0.042 F/B

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

B_C = 24B / 2F = 12 B/F

I = 0.083 F/B

Scalar – can be kept in register

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

B_C = 4B / 2F = 2 B/F

I = 0.5 F/B

Scalar – can be kept in register

```
float s=0, a[], b[];
for(i=0; i<N; ++i)
    s = s + a[i] * b[i];
```

B_C = 8B / 2F = 4 B/F

I = 0.25 F/B

Scalar – can be kept in register

Streaming, perfect spatial locality, no temporal locality → simple

```
float s=0, a[], b[];
for(i=0; i<N; ++i)
  for(j=0; j<N; ++j)
    b[i][j] = a[i][j]
            + a[i-1][j]
            + a[i+1][j];
```

B_C = 16B / 2F or 8B / 2F or even 20 B / 2F ???

Possible cache reuse → tricky!

And what about this?

```
float s=0, a[], b[];
int idx[];
for(i=0; i<N; ++i)
    s = s + a[i]
          * b[idx[i]];
```

We'll get to it!

Is there anything to ease the construction of the model?

Code balance B_C

- Close inspection and hard thinking
- Simplifying assumptions
  - “What is the minimum possible amount of traffic?”
  - “What is the worst case?”

- Tools
  - Kerncraft
    https://github.com/RRZE-HPC/kerncraft

In-core $P_{\max}$

- Inspection of assembly code and manual modeling
- Simplifying assumptions
  - “What is the required minimum number of arithmetic/load/store instructions?”
  - $P_{\max} = P_{peak}$

- Tools
  - OSACA
    https://github.com/RRZE-HPC/OSACA
  - MAQAO/CQA
    https://maqao.org/

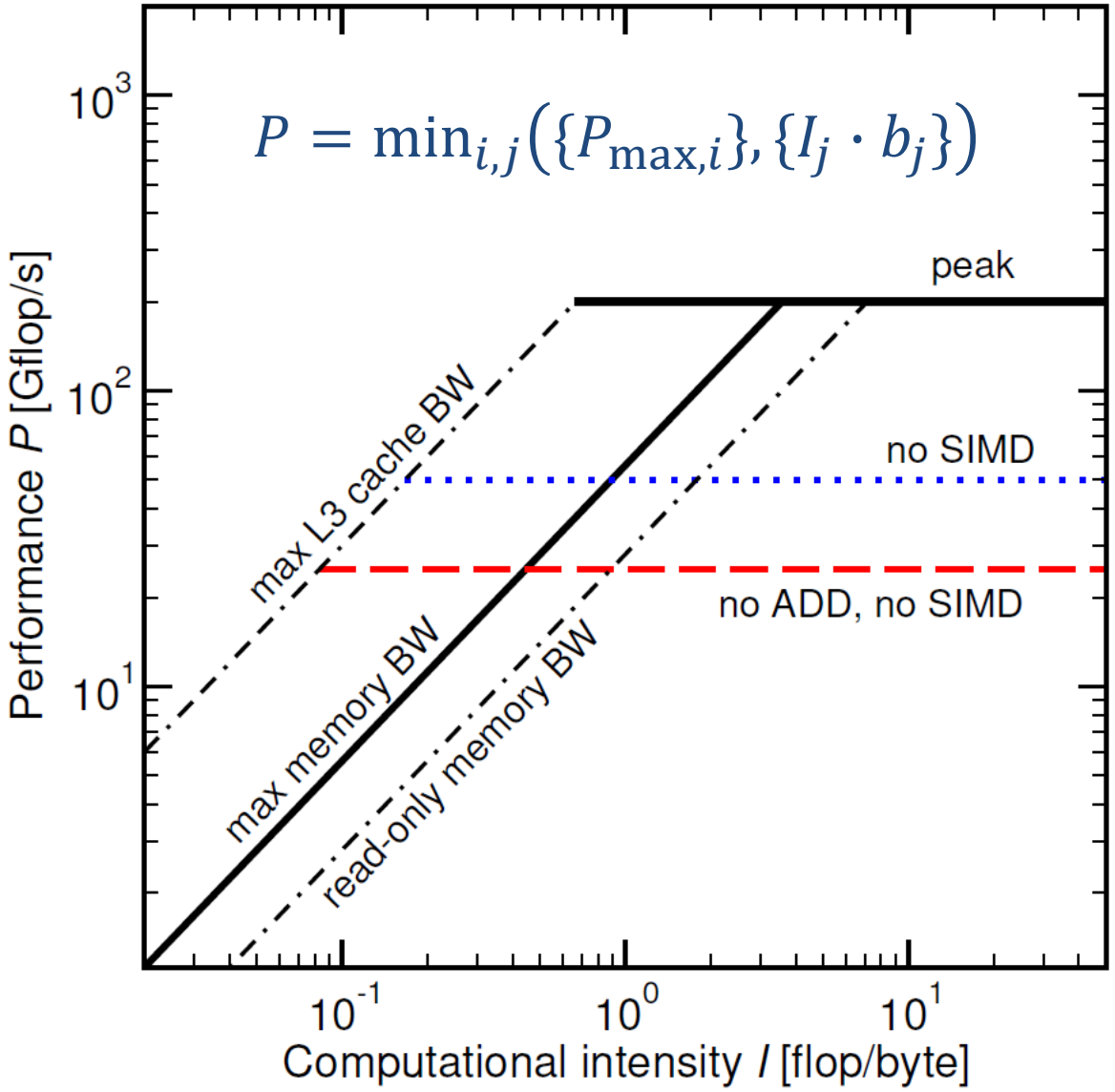
Refined Roofline model: graphical representation

Multiple ceilings may apply

- Different bandwidths / data paths
  → different inclined ceilings
  → possibly different I for one kernel

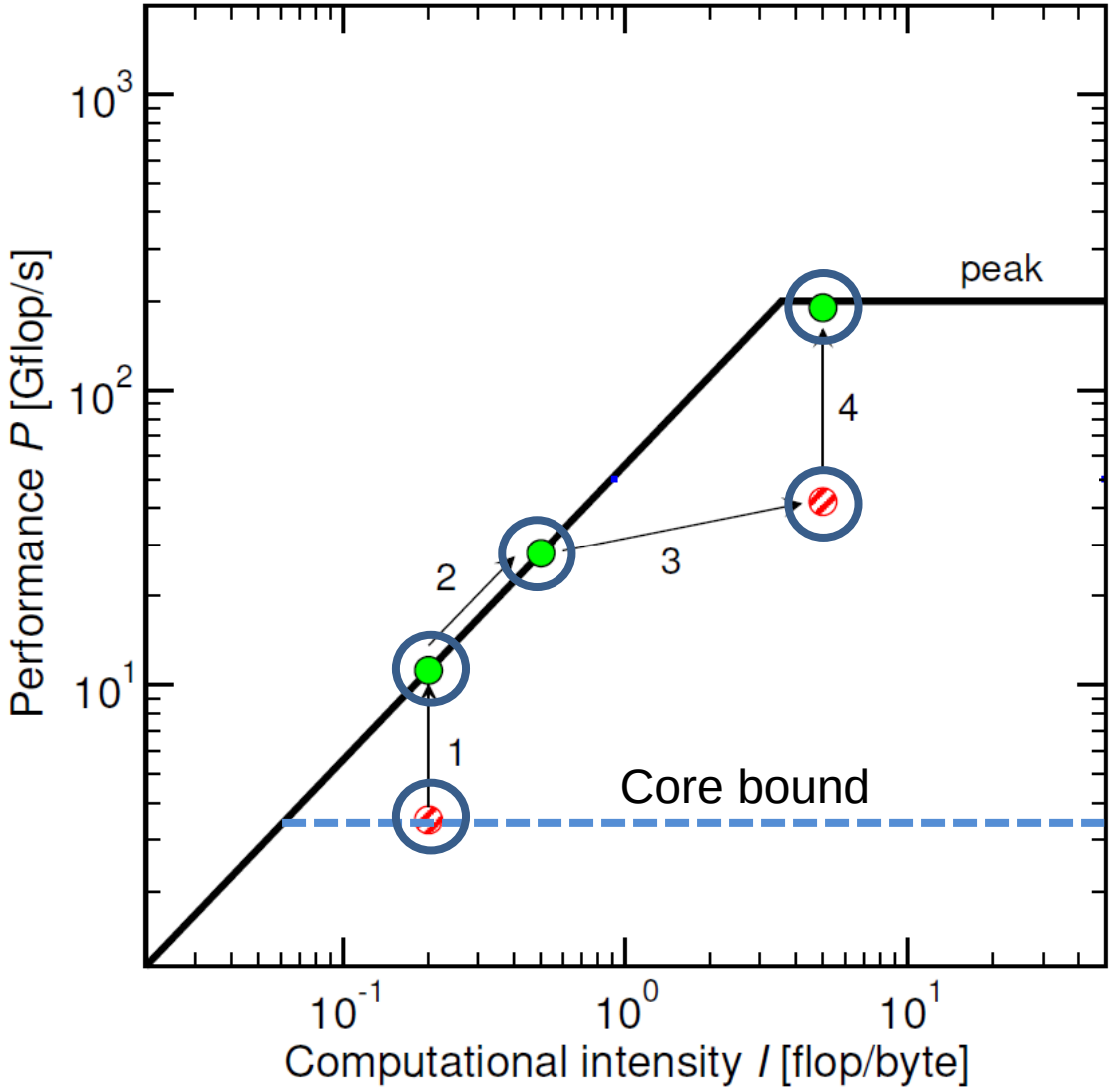
- Different P_{max}
  → different flat ceilings

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



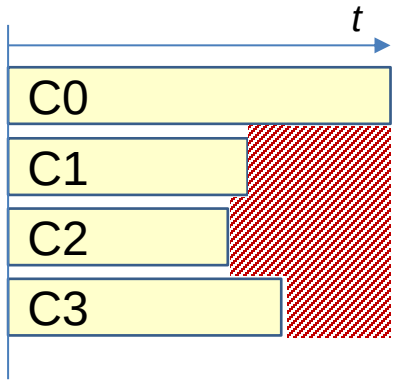
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
   (e.g., Ninja C++ → 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)

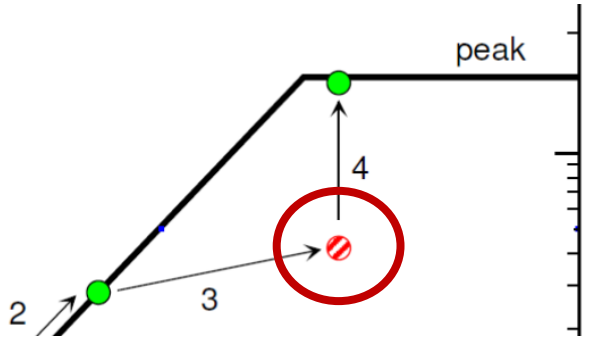


Roofline: How can it “fail”?

… assuming that you did the math right?



- Load imbalance
  - May be impossible to saturate memory bandwidth
  - This includes serial code



- “Slow code”
  - “Invisible” performance ceiling due to inefficient instructions or inefficient execution

- Erratic memory access patterns
  - Latency rains on your parade

```
for(int i=0; i<N; ++i)
  a[i] = s * b[index[i]];
```

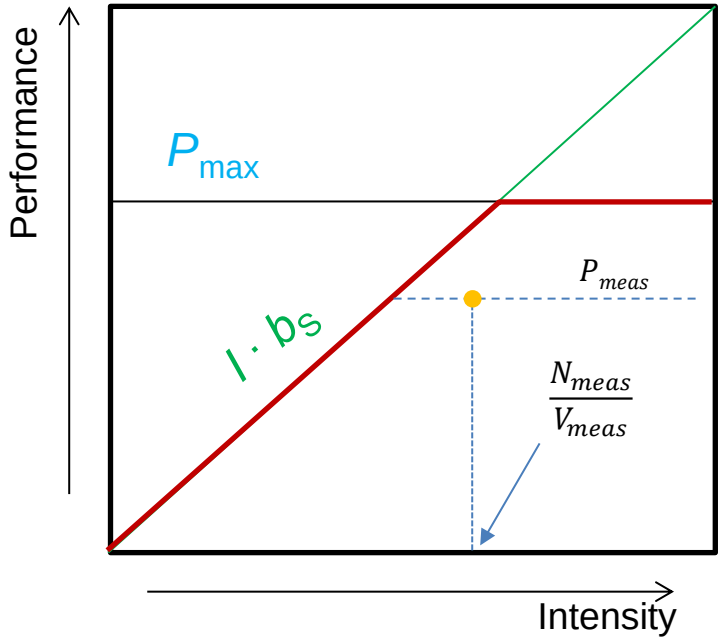
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Diagnostic / phenomenological Roofline modeling

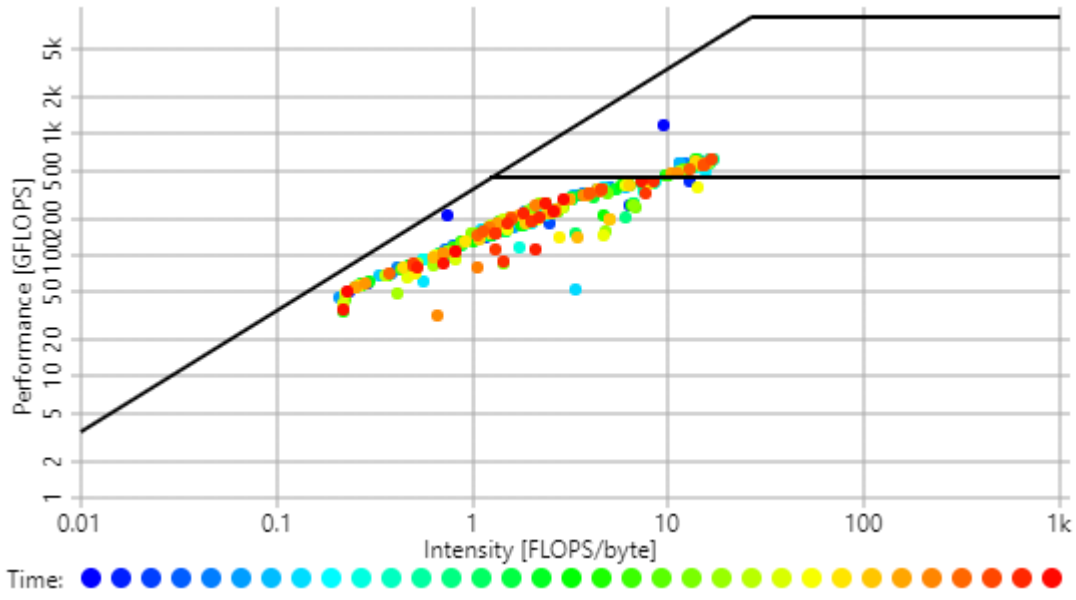
Diagnostic modeling

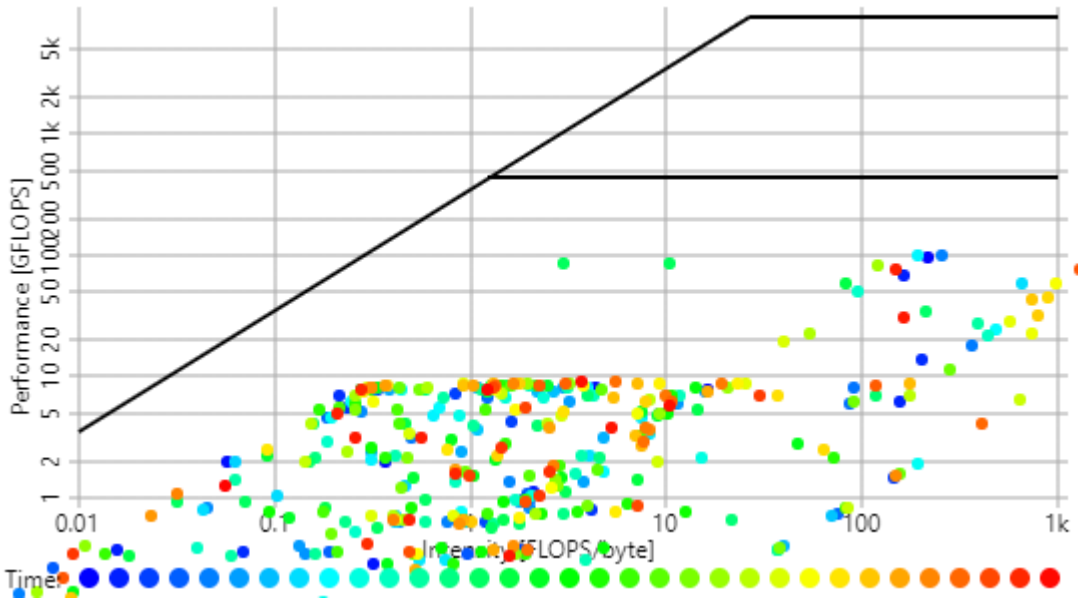
- What if we cannot predict the intensity/balance?
  - Code very complicated
  - Code not available
  - Parameters unknown
  - Doubts about correctness of analysis
- Measure data volume V_{meas} (and work N_{meas})
  - Hardware performance counters
  - Tools: likwid-perfctr, PAPI, Intel Vtune,…
- Insights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

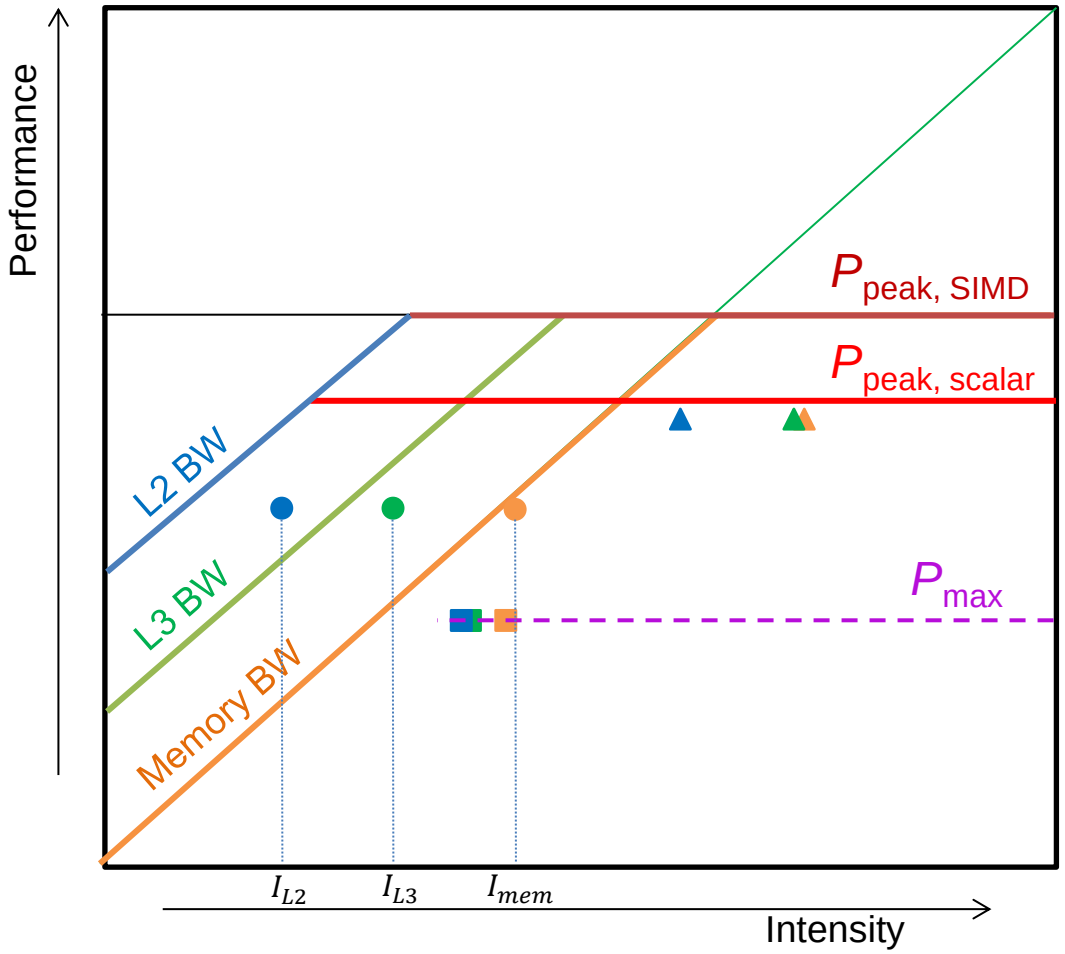




Which of them is “good” and which is “bad”?

Diagnostic modeling of a complex code (3 kernels)

Multiple bandwidth bottlenecks
→ need I for each one ($I_{mem}, I_{L3}, I_{L2}, \ldots$)



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 conclusion

- Roofline = simple first-principle model for upper performance limit of data-streaming loops
  - Machine model ($P_{max}, b_S, \ldots$) + application model ($I_{mem}, \ldots$)
  - Conditions apply, extensions exist

- Two modes of operation; per kernel:
  - Predictive: Calculate I_j, calculate upper limit, validate model, optimize, iterate
  - Diagnostic: Measure I_j and P, compare with ceilings

- Challenge of predictive modeling: Getting P_{max} and I right

NHR FAU

Multicore Performance and Tools

Part 2: Performance Analysis with hardware metrics

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 end-to-end measurement of hardware performance metrics

Operating modes:

- Wrapper
- Stethoscope
- Timeline
- Marker API

Preconfigured and extensible metric groups, list with `likwid-perfctr -a`

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache 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
TLB: TLB miss rate/ratio
ENERGY: Power and energy consumption
```

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:  Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz[…]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+-------------+-------------+-------------+-------------+
|         Event         | Counter | HWThread 36 | HWThread 37 | HWThread 38 | HWThread 39 |
+-----------------------+---------+-------------+-------------+-------------+-------------+
|   INSTR_RETIRED_ANY   |  FIXC0  |  1409713380 |  1393263859 |  1394342491 |  1388917034 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |  2095261718 |  2088036330 |  2075539220 |  2058287996 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |  2103679392 |  2121235200 |  2100479808 |  2075658144 |
|    TOPDOWN_SLOTS_     |  FIXC3  | 10476308590 | 10440181650 | 10377696100 | 10291439980 |
|    L1D_REPLACEMENT    |   PMC0  |   142720376 |   142481840 |   142482162 |   142434419 |
|    L2_TRANS_L1D_WB    |   PMC1  |    54986306 |    54864382 |    54868339 |    54815549 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |      381869 |        2094 |        7399 |        7718 |
+-----------------------+---------+-------------+-------------+-------------+-------------+
[… statistics output omitted …]
+--------------------------------+-------------+-------------+-------------+-------------+
|             Metric             | HWThread 36 | HWThread 37 | HWThread 38 | HWThread 39 |
+--------------------------------+-------------+-------------+-------------+-------------+
|       Runtime (RDTSC) [s]      |      1.0092 |      1.0092 |      1.0092 |      1.0092 |
|      Runtime unhalted [s]      |      0.8751 |      0.8721 |      0.8669 |      0.8597 |
|           Clock [MHz]          |   2384.7406 |   2356.8484 |   2365.8917 |   2374.2844 |
|              CPI               |      1.4863 |      1.4987 |      1.4885 |      1.4819 |
|  L2D load bandwidth [MBytes/s] |   9050.5857 |   9035.4589 |   9035.4794 |   9032.4518 |
|  L2D load data volume [GBytes] |      9.1341 |      9.1188 |      9.1189 |      9.1158 |
| L2D evict bandwidth [MBytes/s] |   3486.9462 |   3479.2144 |   3479.4653 |   3476.1177 |
| L2D evict data volume [GBytes] |      3.5191 |      3.5113 |      3.5116 |      3.5082 |
|    L2 bandwidth [MBytes/s]     |  12561.7480 |  12514.8061 |  12515.4139 |  12509.0589 |
|    L2 data volume [GBytes]     |     12.6777 |     12.6303 |     12.6309 |     12.6245 |
+--------------------------------+-------------+-------------+-------------+-------------+
```

Always measured for Intel CPUs

Configured metrics (this group)

Derived metrics

`likwid-perfctr` stethoscope mode

- likwid-perfctr counts events on cores; 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:

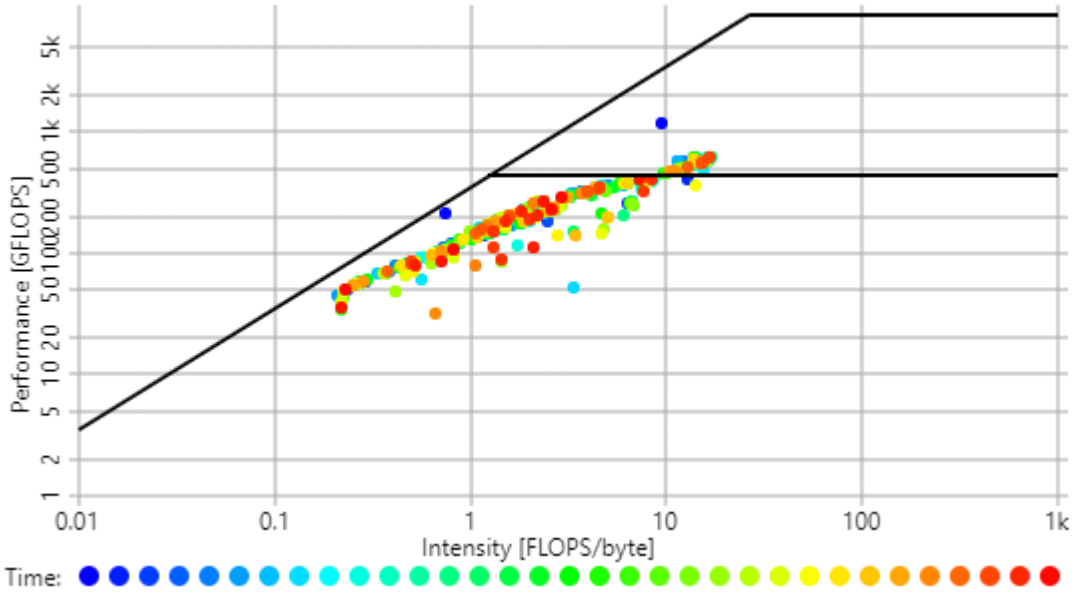
```
$ likwid-perfctr -c N:0-11 -g FLOPS_DP  -S 10s
```

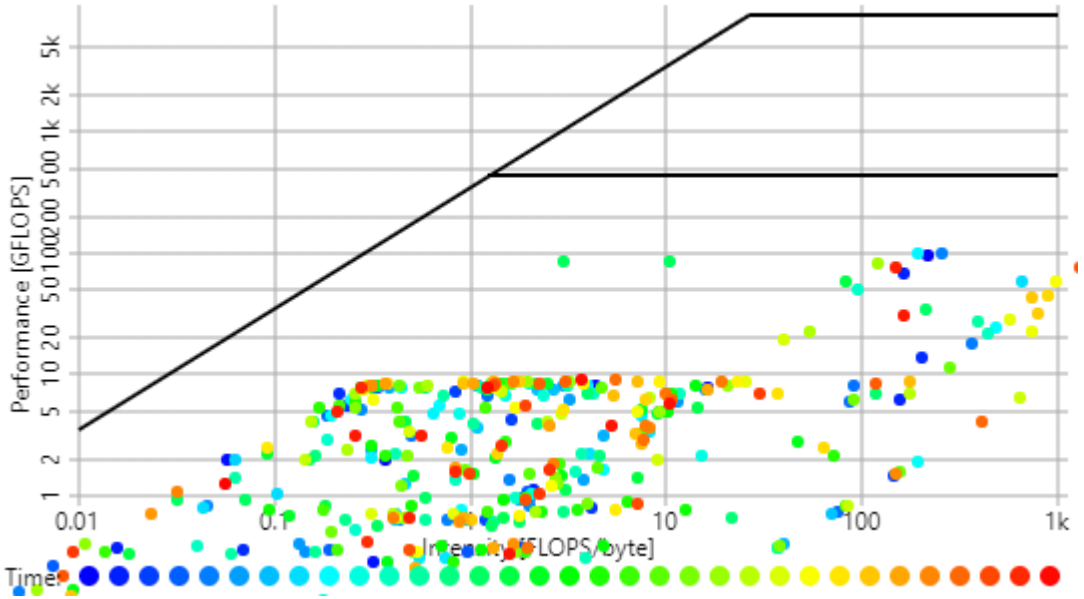
- Can be used as cluster/server monitoring tool

- Frequent use: monitor a long-running parallel application from outside

Cluster monitoring with likwid-perfctr

Two jobs on the NHR@FAU “Fritz” cluster





https://github.com/ClusterCockpit

`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 significant overhead; do not call too frequently!

```
#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: OpenMP code (C)

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

int main(...) {
  LIKWID_MARKER_INIT;
  #pragma omp parallel
  {
    LIKWID_MARKER_REGISTER("MatrixAssembly");
  }
  ...
  #pragma omp parallel
  {
    LIKWID_MARKER_START("MatrixAssembly");
    #pragma omp for
    for(int i=0; i<N; ++i) { /* Loop */ }
    LIKWID_MARKER_STOP("MatrixAssembly");
  }
  ...
  LIKWID_MARKER_CLOSE;
}
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerC

likwid-perfctr with MarkerAPI: OpenMP code (Fortran)

```
program p
  use likwid
  call likwid_markerInit
  !$omp parallel
    call likwid_markerRegisterRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  !$omp parallel
    call likwid_markerStartRegion(“MatrixAssembly”)
    !$omp do
    do i=1,N
     ! Loop
    enddo
    !$omp end do
    call likwid_markerStopRegion(“MatrixAssembly”)
  !$omp end parallel
  ...
  call likwid_markerClose
end program p
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90

`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
```

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:

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

→ One separate block of output for every marked region

So... what should I look at first?

Focus on resource utilization and instruction decomposition!

Metrics to measure:

- Operation throughput (Flops/s)
- Overall instruction throughput (IPC,CPI)
- Instruction breakdown:
  - FP instructions
  - loads and stores
  - branch instructions
  - other instructions
- Instruction breakdown to SIMD width (scalar, SSE, AVX, AVX512 for x86)
- Data volumes and bandwidths to main memory (GB and GB/s)
- Data volumes and bandwidth to different cache levels (GB and GB/s)

Useful diagnostic metrics are:

- Clock frequency (GHz)
- Power (W)

All the above metrics can be acquired using performance groups:

```
MEM_DP, MEM_SP, BRANCH, DATA, L2,  L3
```

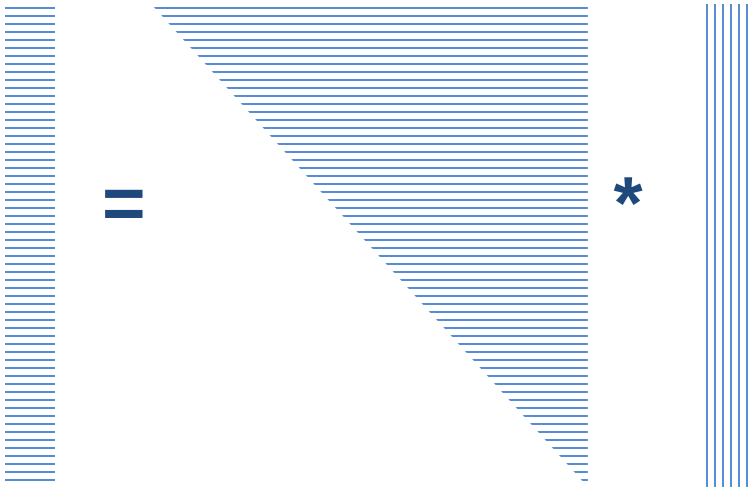
Example: triangular matrix-vector multiplication

```
#define N 10000 // matrix in memory
#define ROUNDS 10
// Initialization
fillMatrix(mat, N*N, M_PI);
fillMatrix(bvec, N, M_PI);

// Calculation loop
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
```

Prevent smart compilers from eliminating benchmark if **`cvec`** not used afterwards

Example: triangular matrix-vector multiplication

```
#include <likwid-marker.h>
[…] // defines, fillMatrix, init data
LIKWID_MARKER_INIT;
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        LIKWID_MARKER_START(“Compute”);
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        LIKWID_MARKER_STOP(“Compute”);
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
LIKWID_MARKER_CLOSE;
```



Example: triangular matrix-vector multiplication

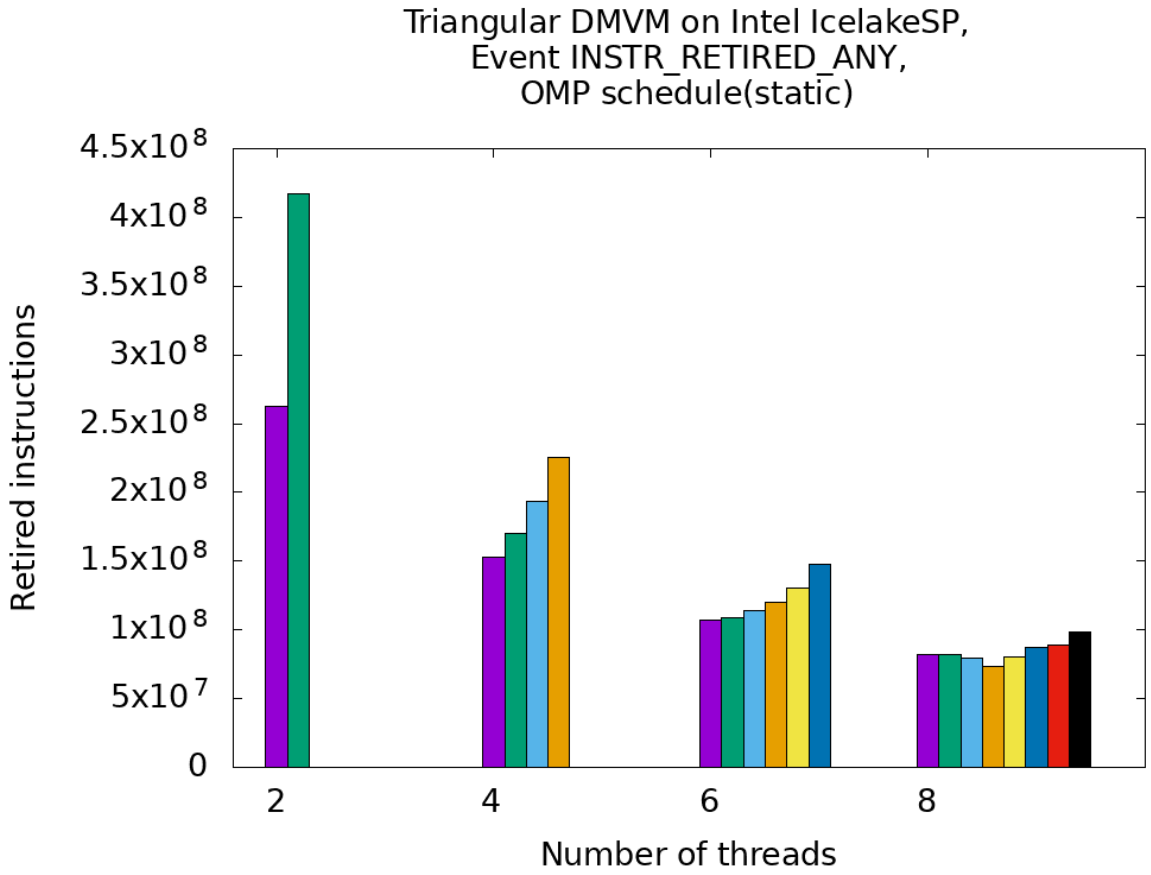
```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
--------------------------------------------------------------------------------
CPU type:  Intel Icelake SP processor
CPU clock: 2.39 GHz
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
Region Compute, Group 1: L2
+--------------------+------------+------------+------------+
|    Region Info     | HWThread 0 | HWThread 1 | HWThread 2 |
+--------------------+------------+------------+------------+
| RDTSC Runtime [s] |   0.198263 |   0.198364 |   0.198246 |
|     call count     |         10 |         10 |         10 |
+--------------------+------------+------------+------------+

+-----------------------+---------+------------+------------+------------+
|         Event         | Counter | HWThread 0 | HWThread 1 | HWThread 2 |
+-----------------------+---------+------------+------------+------------+
|   INSTR_RETIRED_ANY   |  FIXC0  |  194399400 |  269695800 |  341470000 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |  458193600 |  464605300 |  433236300 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |  473442400 |  469863600 |  465054300 |
|     TOPDOWN_SLOTS     |  FIXC3  | 2290968000 | 2323026000 | 2166181000 |
|    L1D_REPLACEMENT    |   PMC0  |   69660770 |   41754150 |    7610321 |
|    L2_TRANS_L1D_WB    |   PMC1  |      43768 |     263047 |     442018 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |       9698 |      11399 |      11571 |
+-----------------------+---------+------------+------------+------------+
```

???

Example: triangular matrix-vector multiplication

Retired instructions are misleading!

Waiting in implicit OpenMP barrier executes many instructions



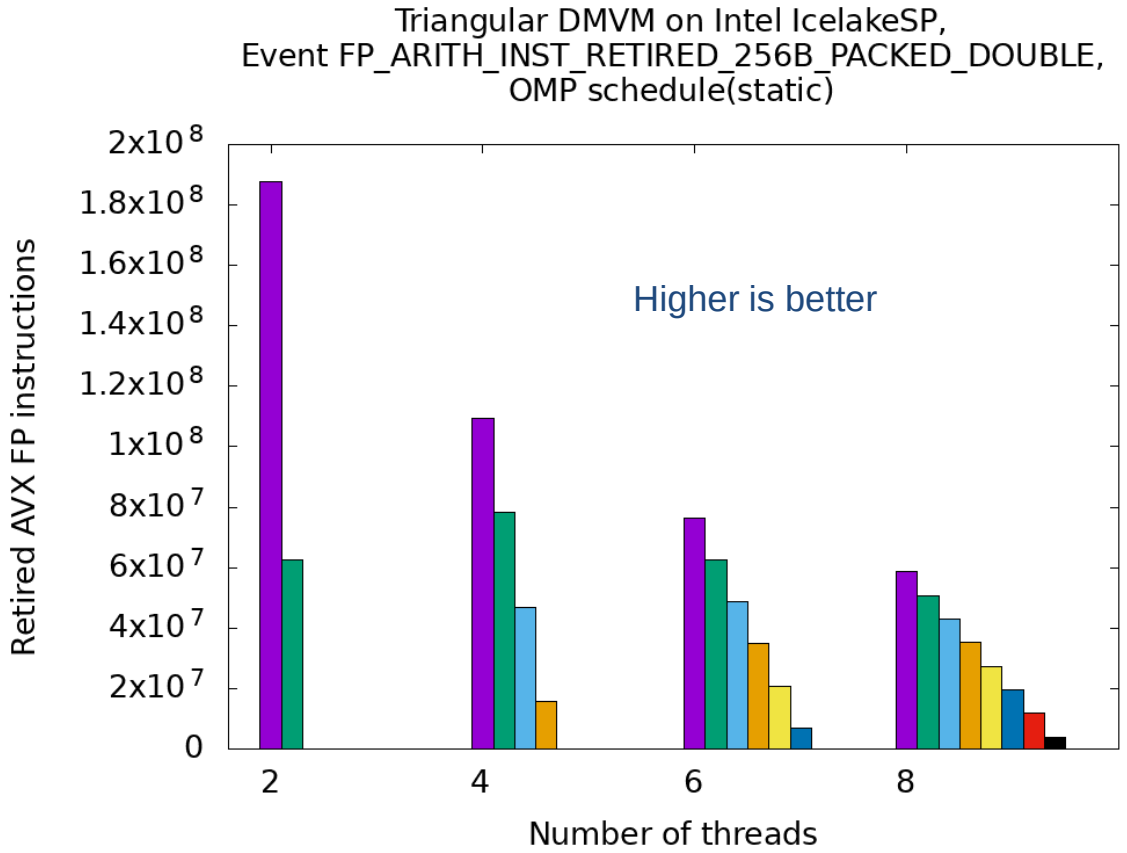
We need to measure actual work (or use a tool that can separate user from runtime lib instructions)

Example: triangular matrix-vector multiplication

Floating-point instructions reliable ↔ useful work metric

Caveats:

- FP instruction counters from SandyBridge to Haswell are only qualitatively correct
- Masked SIMD lanes cannot be counted directly on x86

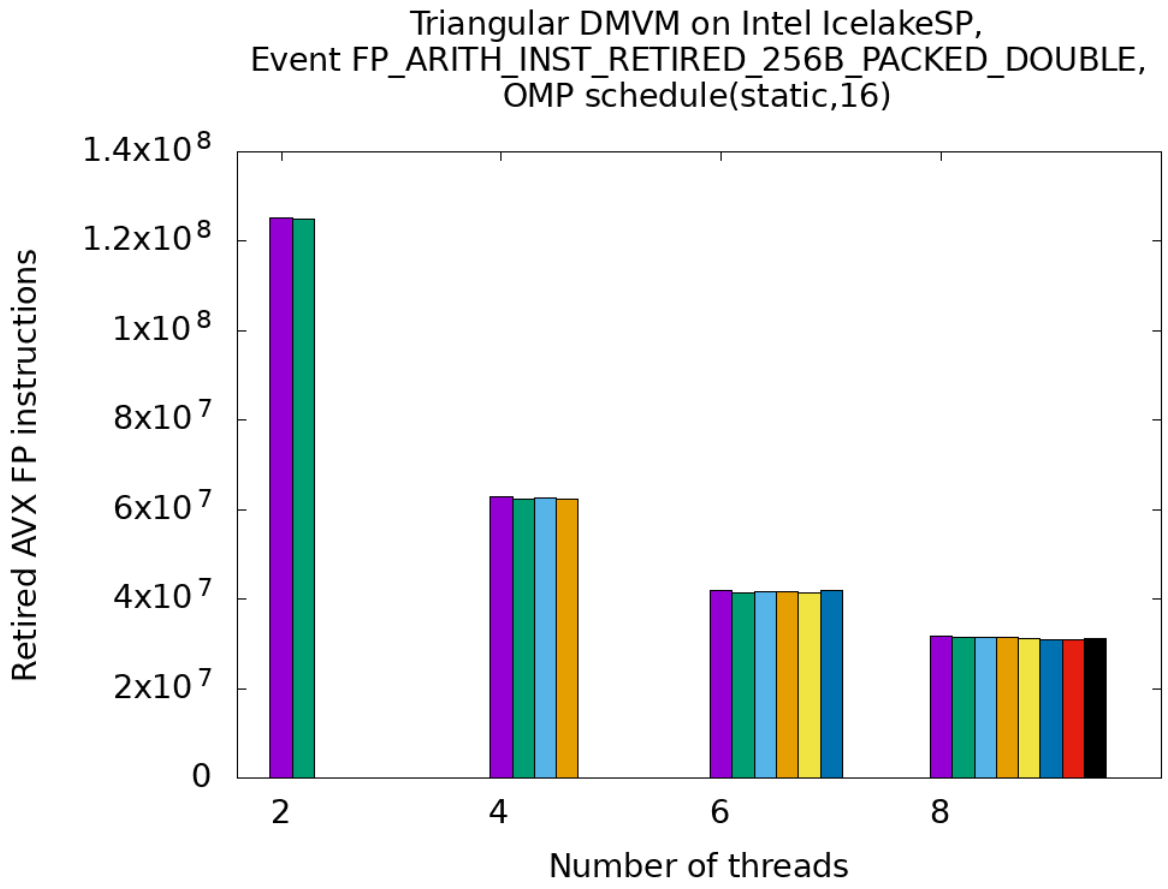


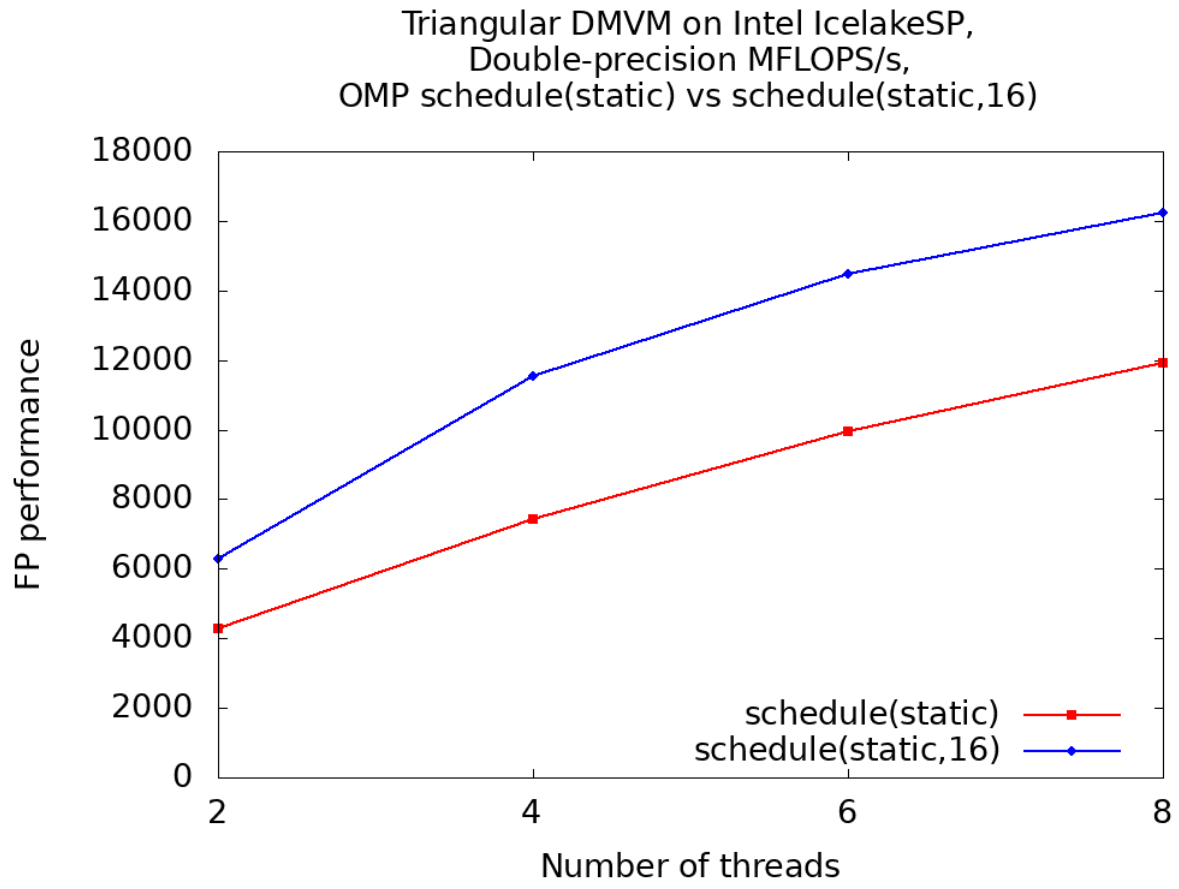
Example: triangular matrix-vector multiplication

Changing OMP schedule to static with chunk size 16 → smaller work packages per thread

No imbalance anymore!

Is it also faster?





Summary of hardware performance monitoring

- Useful only if you know what you are looking for
  - Hardware event counting bears the 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
  - Compilers may introduce instructions not visible in the source code
  - Some tools can distinguish runtime instructions from user code instructions
- Another very useful application of PM: validating performance models!
  - Roofline is data centric → measure data volume through memory hierarchy

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

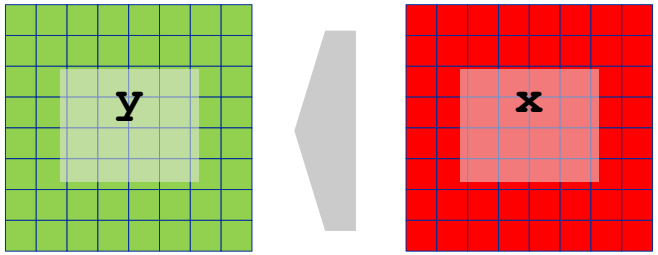
Case study: A Jacobi smoother

The basics in two dimensions

Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically 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(:) ← x(:)

  Swap y ←→ x

enddo
```



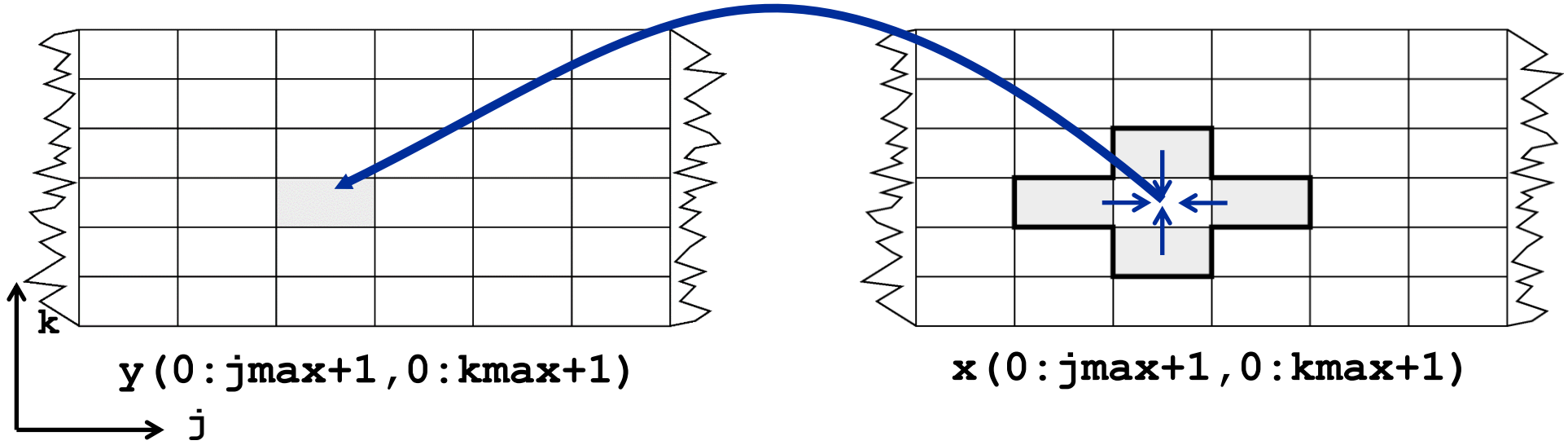
- Complexity of implementation and performance depends on
  - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type, …
  - discretization, e.g. 7-pt or 27-pt in 3D,…

Jacobi-type 5-pt stencil 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
```

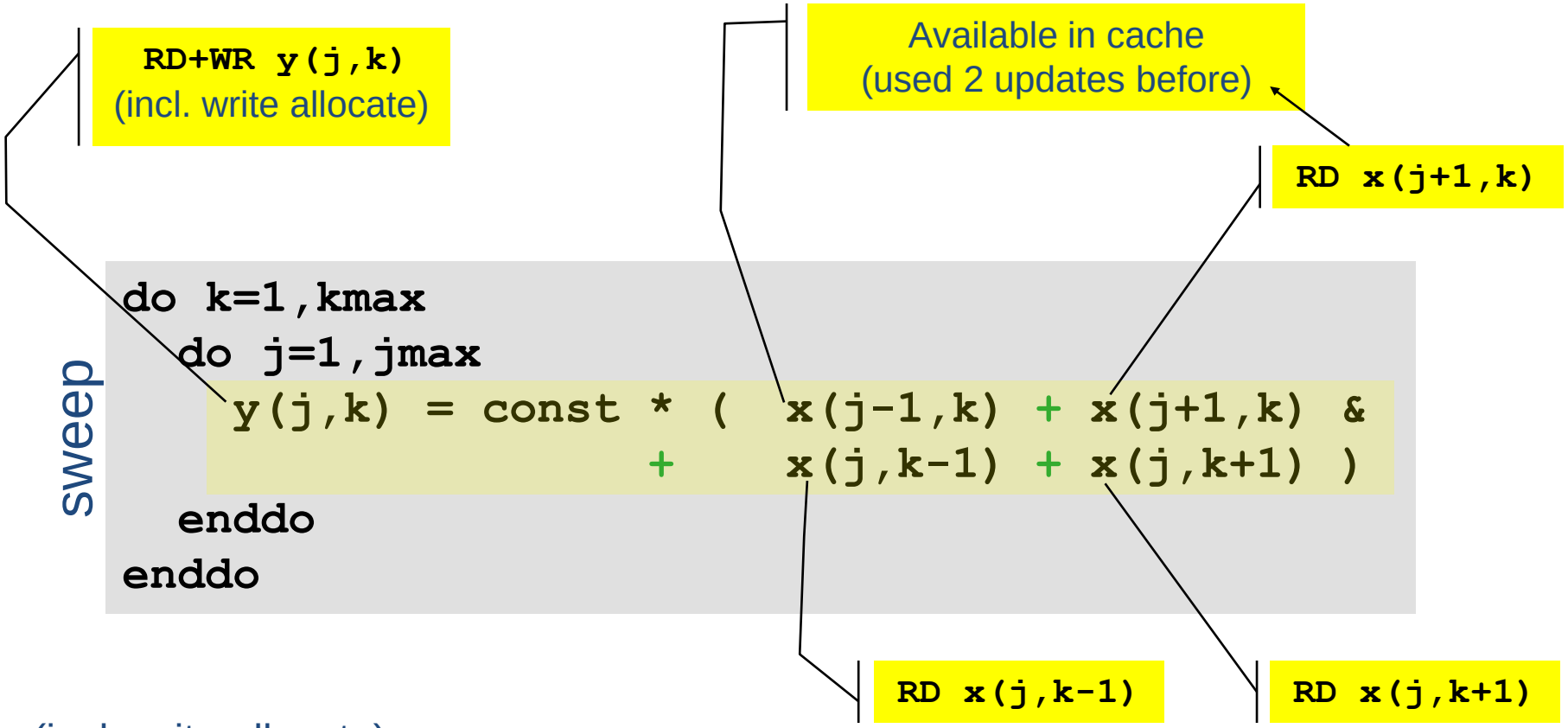
Lattice site update (LUP)



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

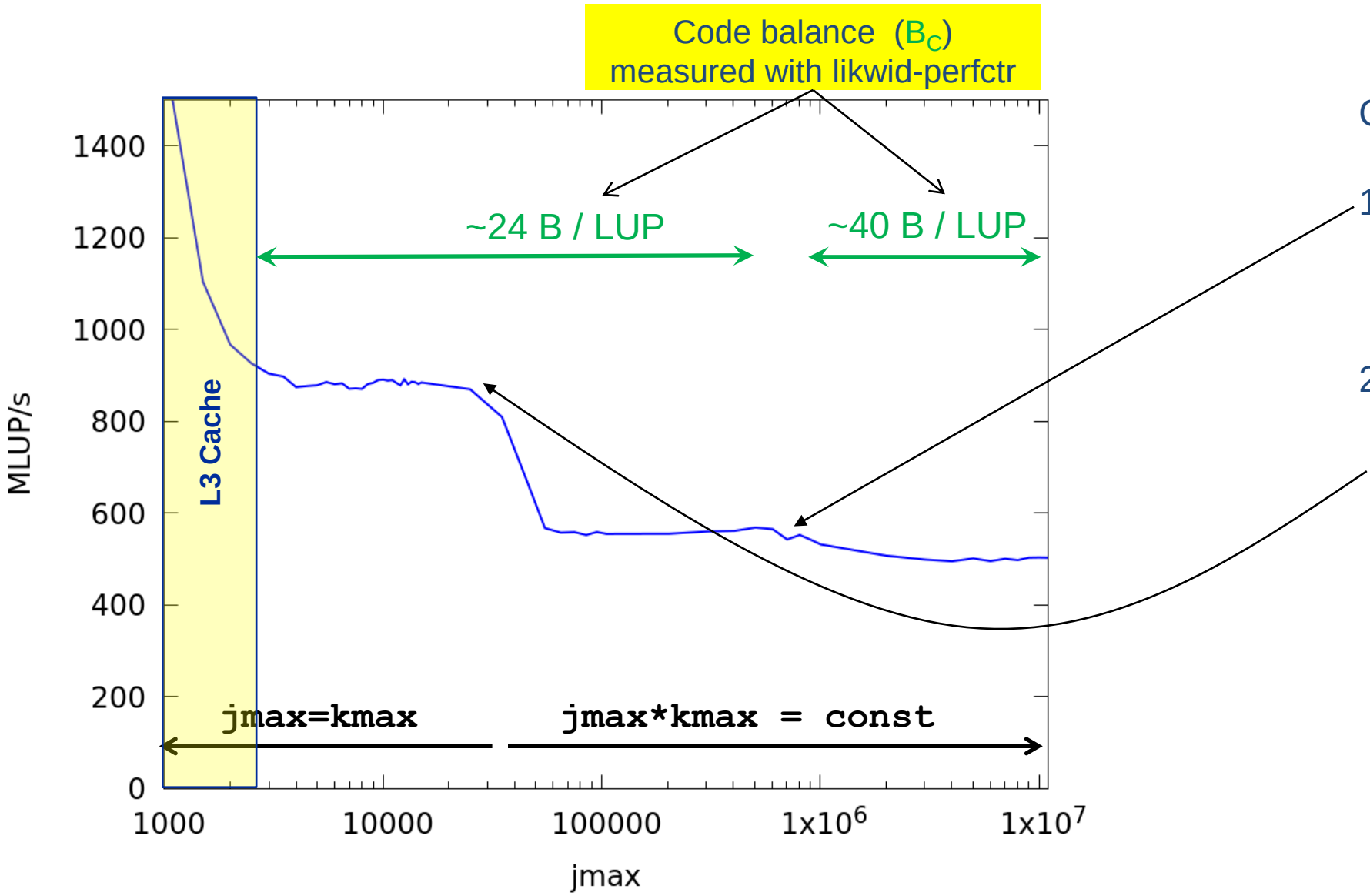


Naive balance (incl. write allocate):

`x( :, :)` : 3 RD +
`y( :, :)` : 1 WR+ 1 RD

→ B_C = 5 Words / LUP = 40 B / LUP (assuming double precision)

Jacobi 5-pt stencil 2D: Single core performance



Questions:

1. How to achieve 24 B/LUP also for large **`jmax`**?
2. How to sustain >800 MLUP/s for **`jmax > 10⁴`** ?

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

NHR FAU

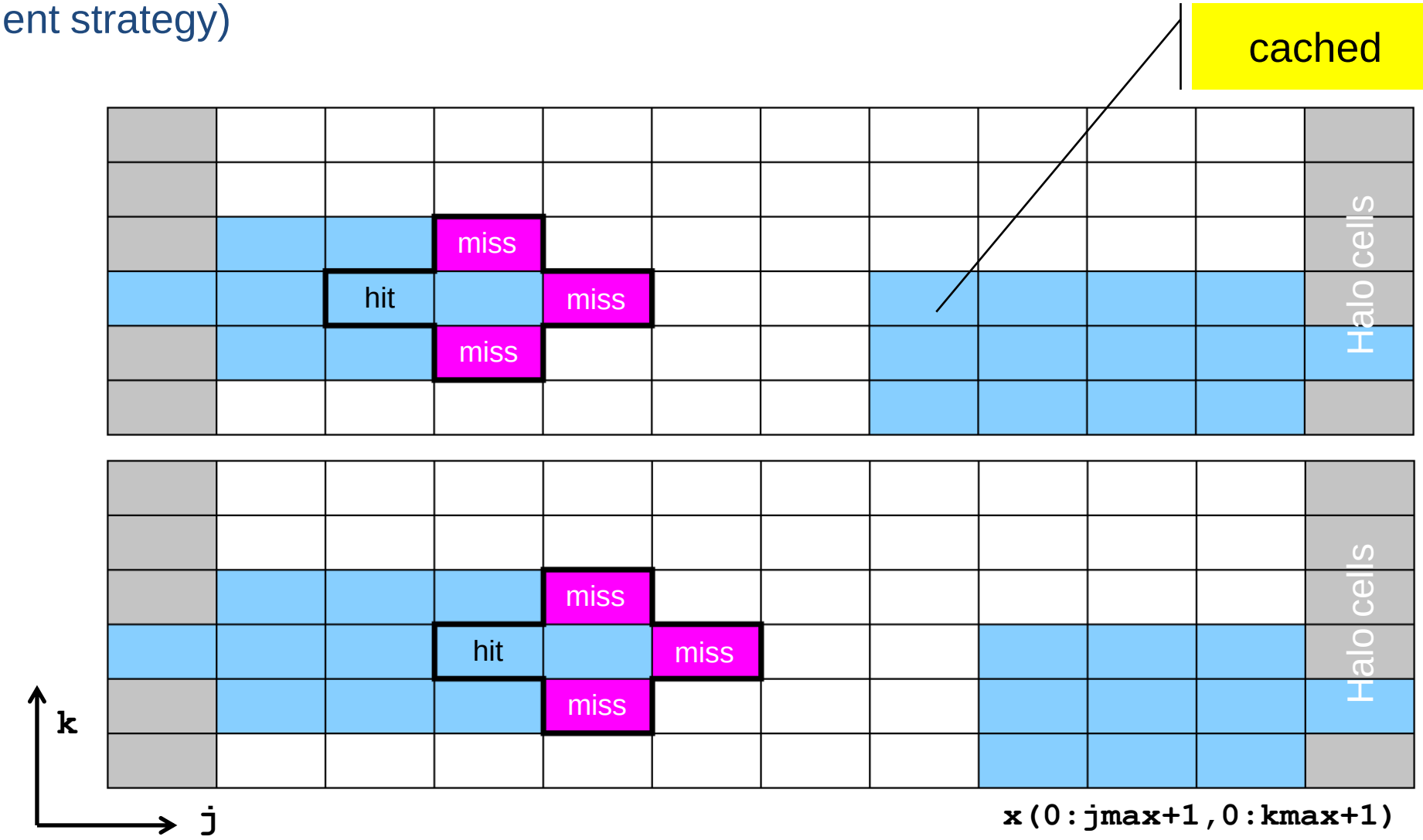
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: A Jacobi smoother

Layer conditions

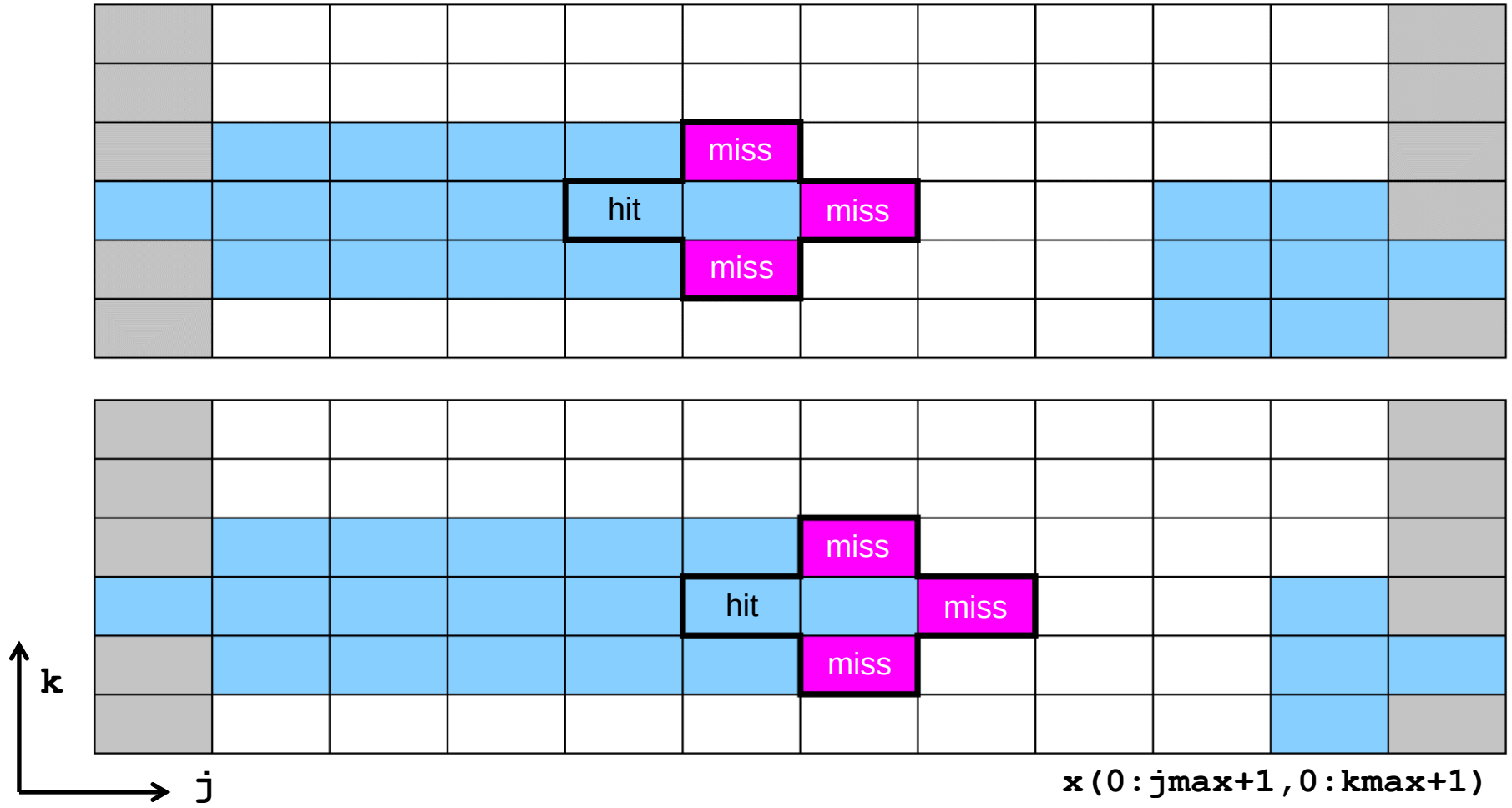
Analyzing the data flow

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



Analyzing the data flow

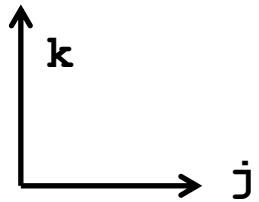
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume „Least Recently Used“ replacement strategy)

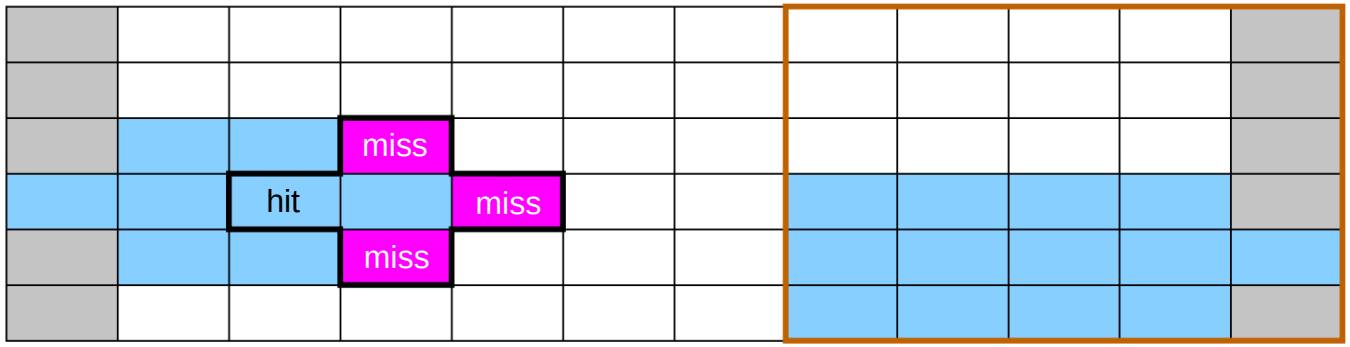


Analyzing the data flow

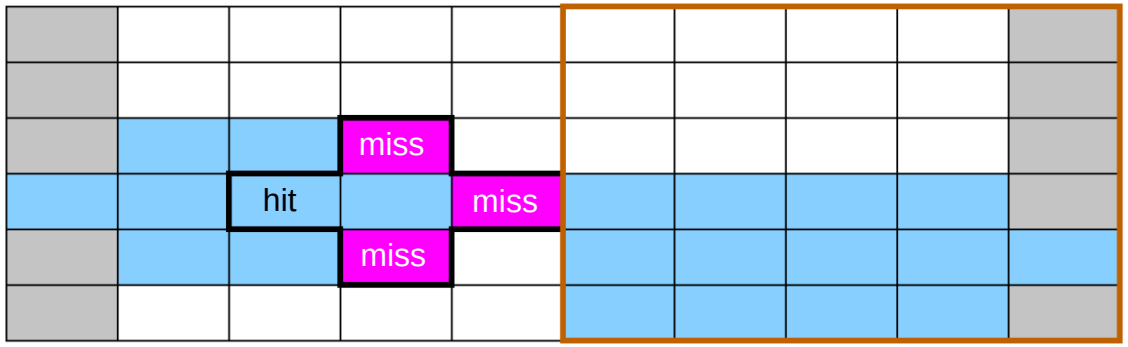
Reduce inner (j-) loop dimension successively

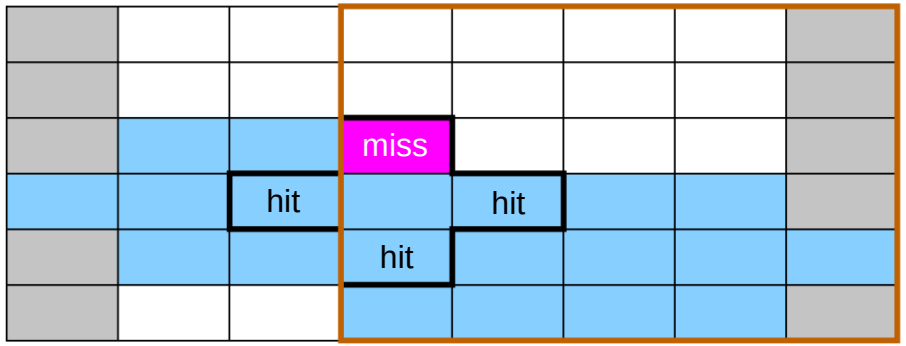
Best case: 3 “layers” of grid fit into the cache!





`x(0:jmax1+1,0:kmax+1)`

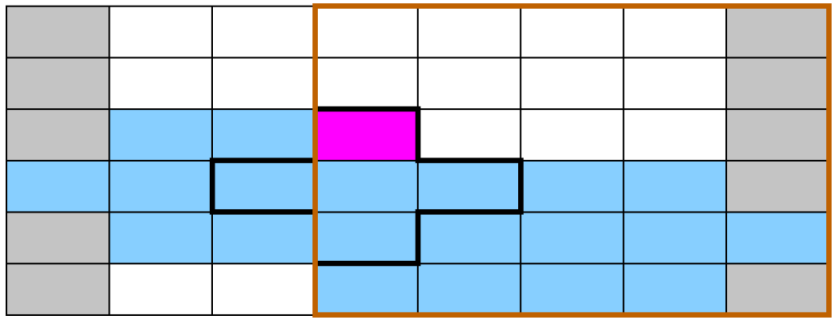




`x(0:jmax2+1,0:kmax+1)`

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil, cache size C



```
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
```

$$3 \times \mathbf{jmax} \times 8 \text{ byte} < \frac{C}{2}$$

"Layer condition"

3 rows of **jmax**

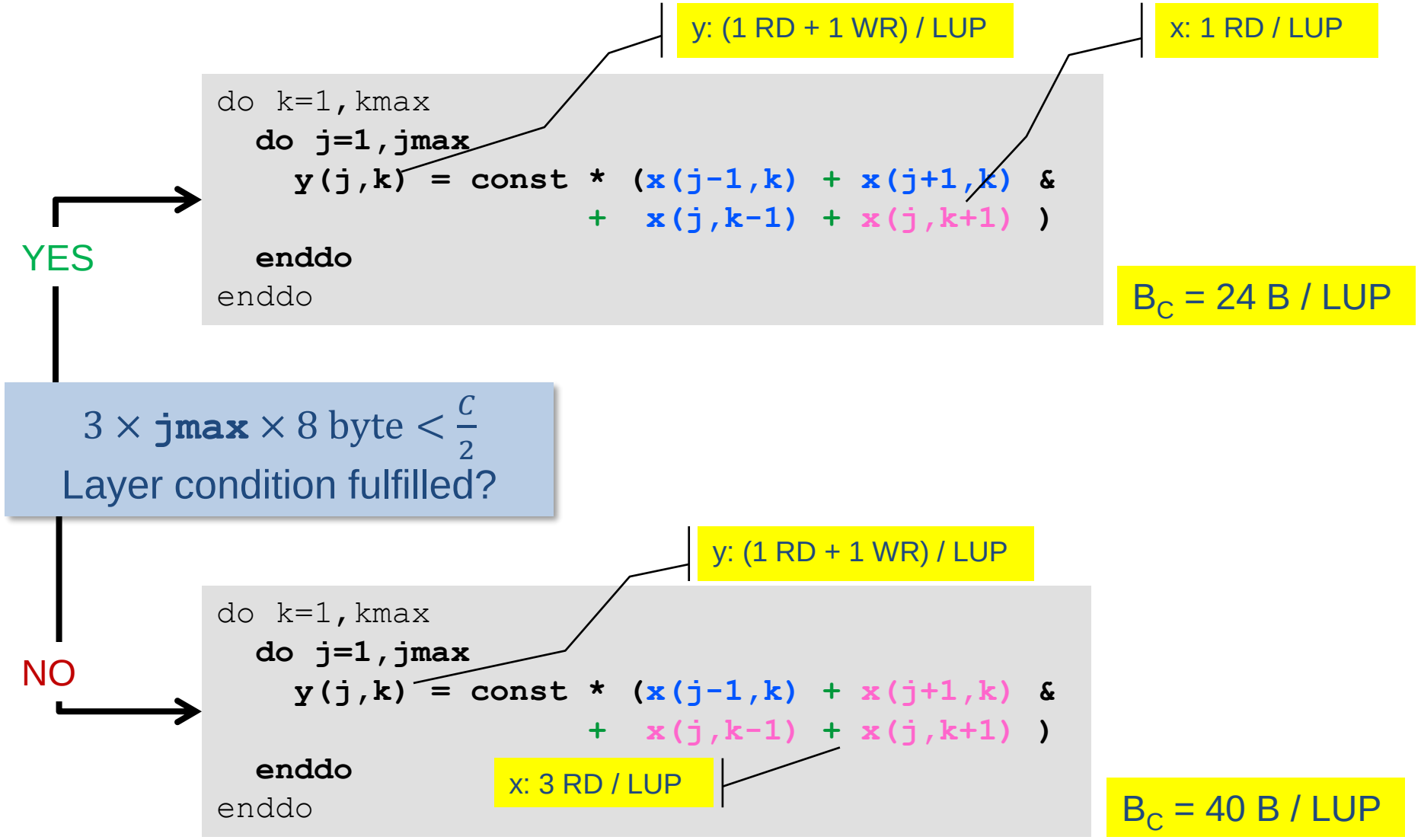
double precision

Safety margin (Rule of thumb)

Layer condition:

- Does not depend on outer loop length (**kmax**)
- 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 (2D 5-pt Jacobi)



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: A Jacobi smoother

Optimization by spatial blocking

Enforcing a layer condition (2D 5-pt Jacobi)

- How can we enforce a layer condition for all domain sizes ?
- Idea: Spatial blocking
  - Reuse elements of `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)

$$3 \times \mathbf{jblock} \times 8\ \text{byte} < \frac{C}{2}$$

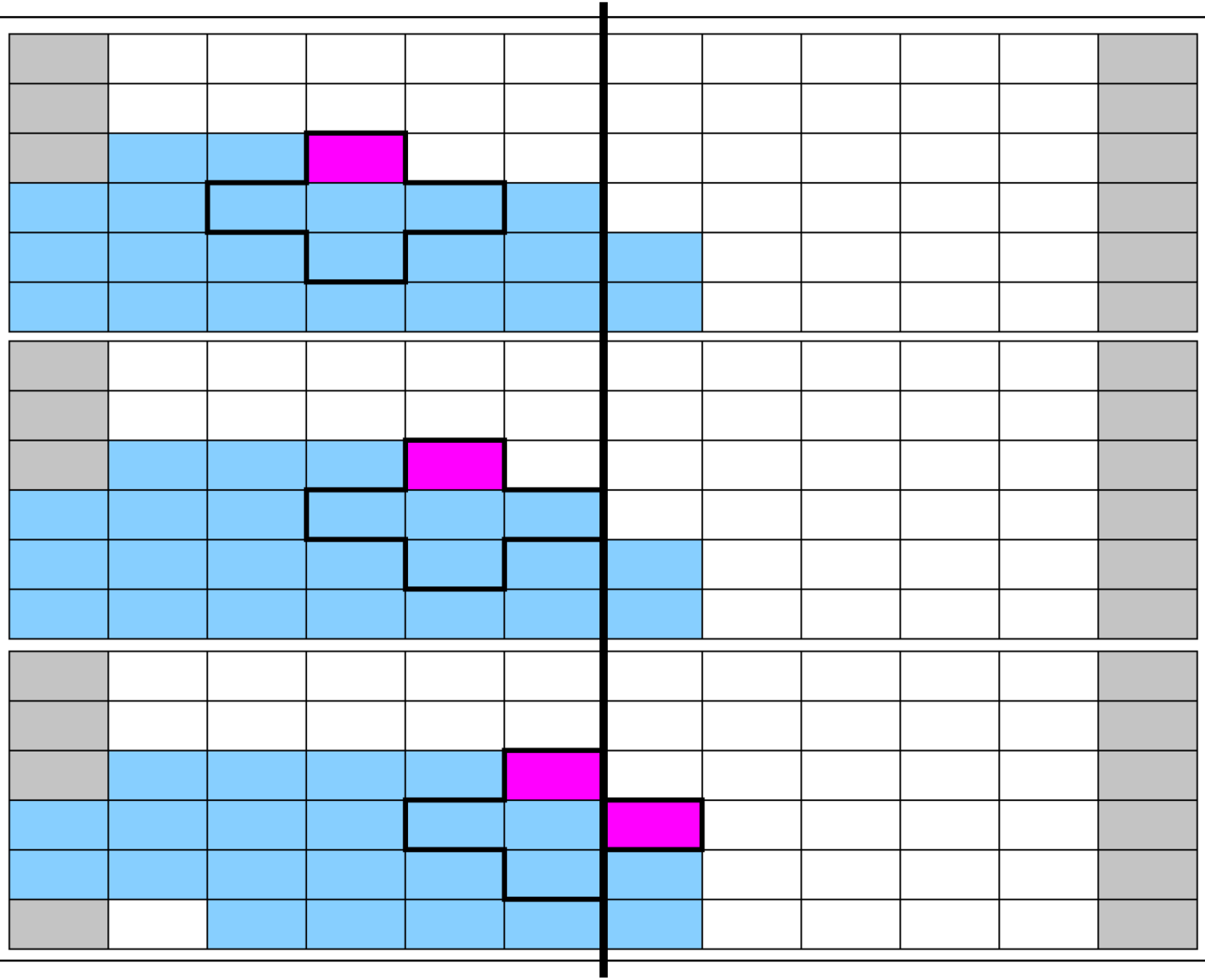
Determine for given C an appropriate `jblock` value:

$$\mathbf{jblock} < \frac{C}{48\ \text{byte}}$$

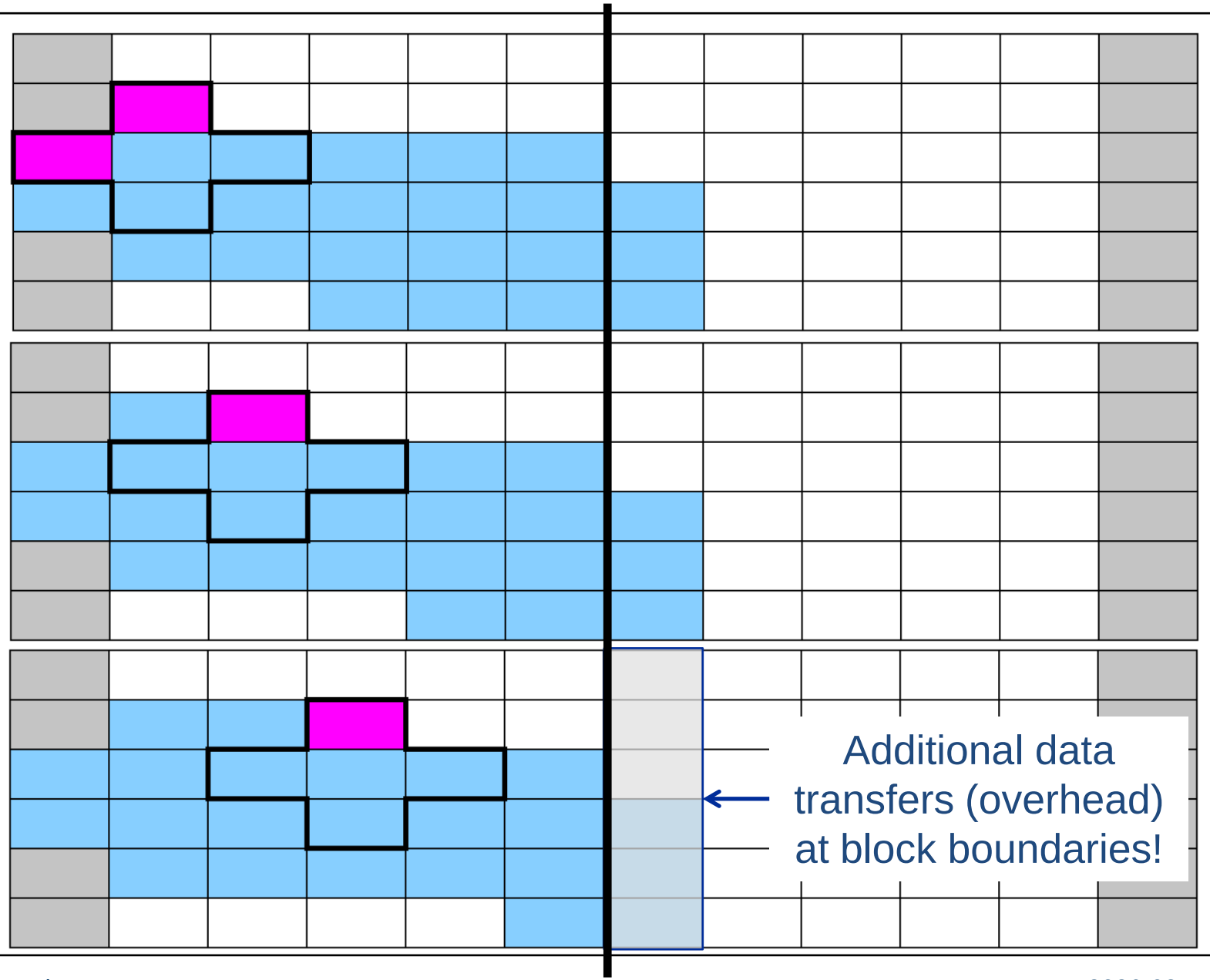
Establish the layer condition by blocking

Split domain into subblocks:

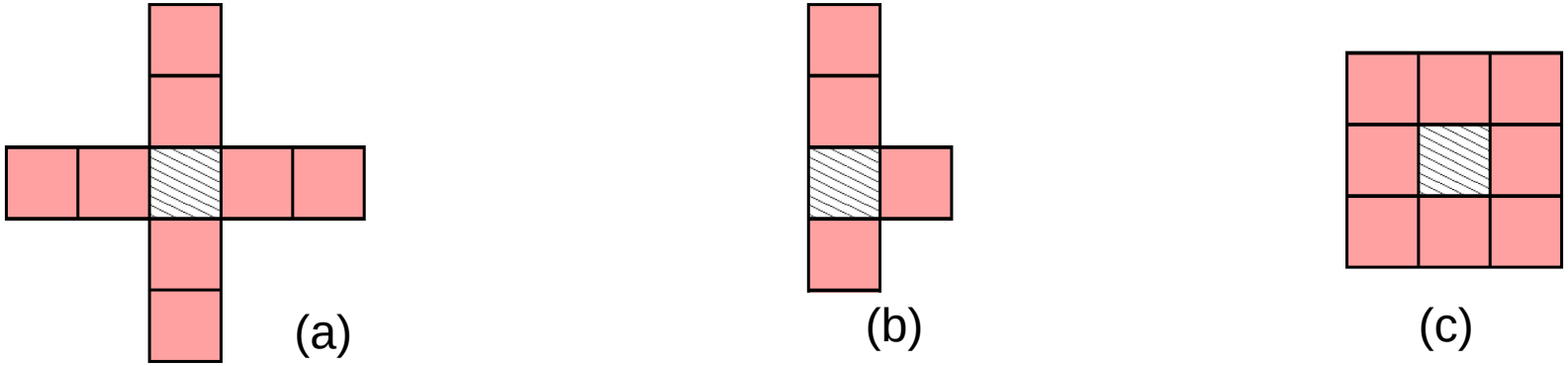
e.g. block size = 5



Establish the layer condition by blocking



Stencil shapes and layer conditions in 2D



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

b) Asymmetric: 4 layers

c) 2D box: 3 layers

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

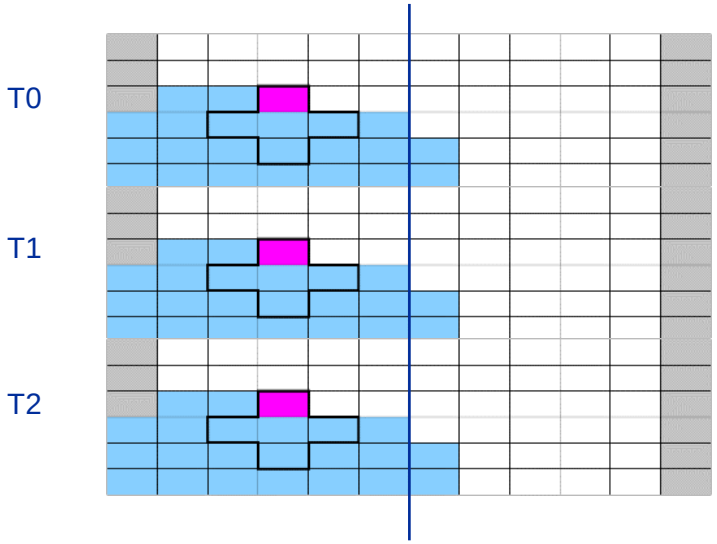
Case study: A Jacobi smoother

OpenMP parallelization

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!

Layer condition:

$$3 \times \texttt{jblock} \times 8 \text{ byte} < \frac{C_t}{2}$$

Cache size available per thread

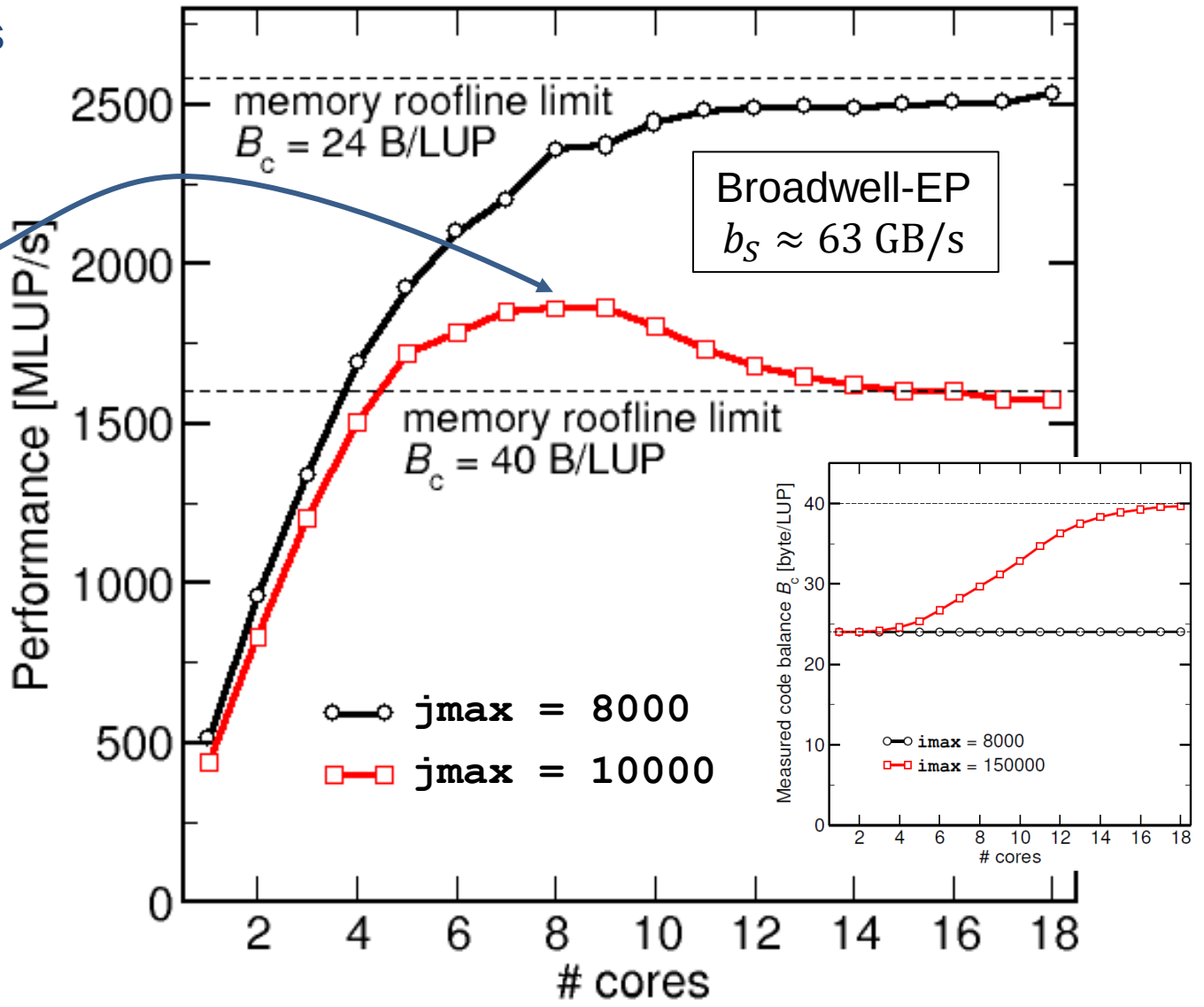
OpenMP parallelization and blocking for a shared cache

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
     - Layers either small enough to fit in core-private caches *or*
     - Shared cache big enough to hold all layers for all threads
  2. Adaptive blocking for shared cache:

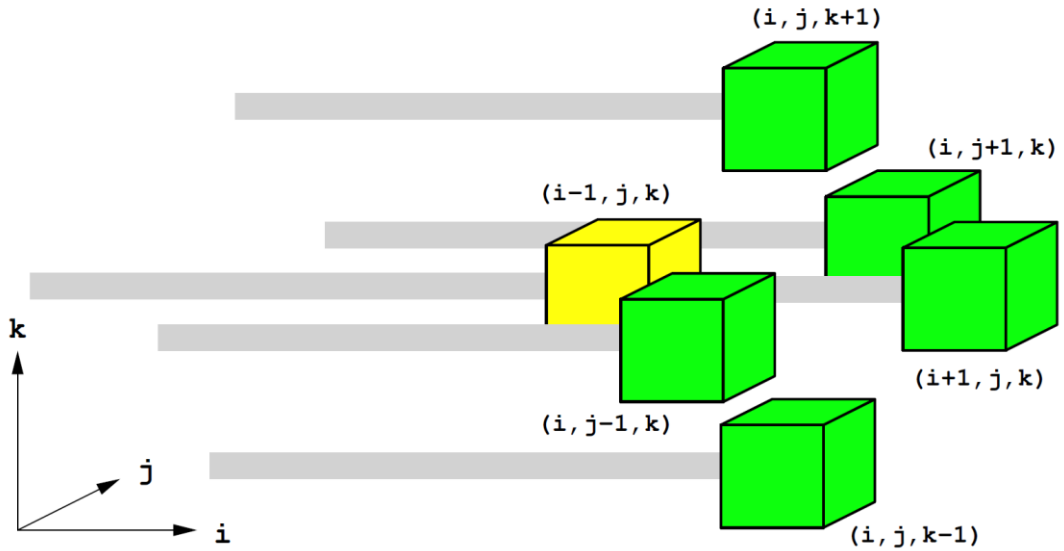
$$\texttt{jblock} = \frac{C}{\#threads \times 48\ \text{byte}}$$



What about a 3D 7-pt stencil?

There is actually more than one layer condition in 3D

- “Outer” LC:
  `3 * imax * jmax`
- “Inner” LC:
  `3 * imax` in the central layer



→ Code balance of

- 24 B/LUP if outer LC fulfilled
- 40 B/LUP if outer LC broken but inner LC fulfilled
- 56 B/LUP if inner & outer LC broken

Online Layer Condition calculator: **`http://tiny.cc/LayerConditions`**

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 speedup predicted by the model
- Optimal blocking factor can be estimated
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of “optimization space”
- Food for thought
  - 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

- J. Hornich, J. Hammer, G. Hager, T. Gruber, and G. Wellein: Collecting and Presenting Reproducible Intranode Stencil Performance: INSPECT. Supercomputing Frontiers and Innovations 6(3), 4-25 (2019). ISSN 2313-8734. Available with Open Access. DOI: 10.14529/jsfi190301

- T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes: Multi-dimensional intra-tile parallelization for memory-starved stencil computations. ACM Transactions on Parallel Computing 4(3), 12:1-12:32 (2017). DOI: 10.1145/3155290, Preprint: arXiv:1510.04995

- T. M. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. E. Keyes: Multicore-optimized wavefront diamond blocking for optimizing stencil updates. SIAM Journal on Scientific Computing 37(4), C439-C464 (2015). DOI: 10.1137/140991133, Preprint: arXiv:1410.3060

NHR FAU

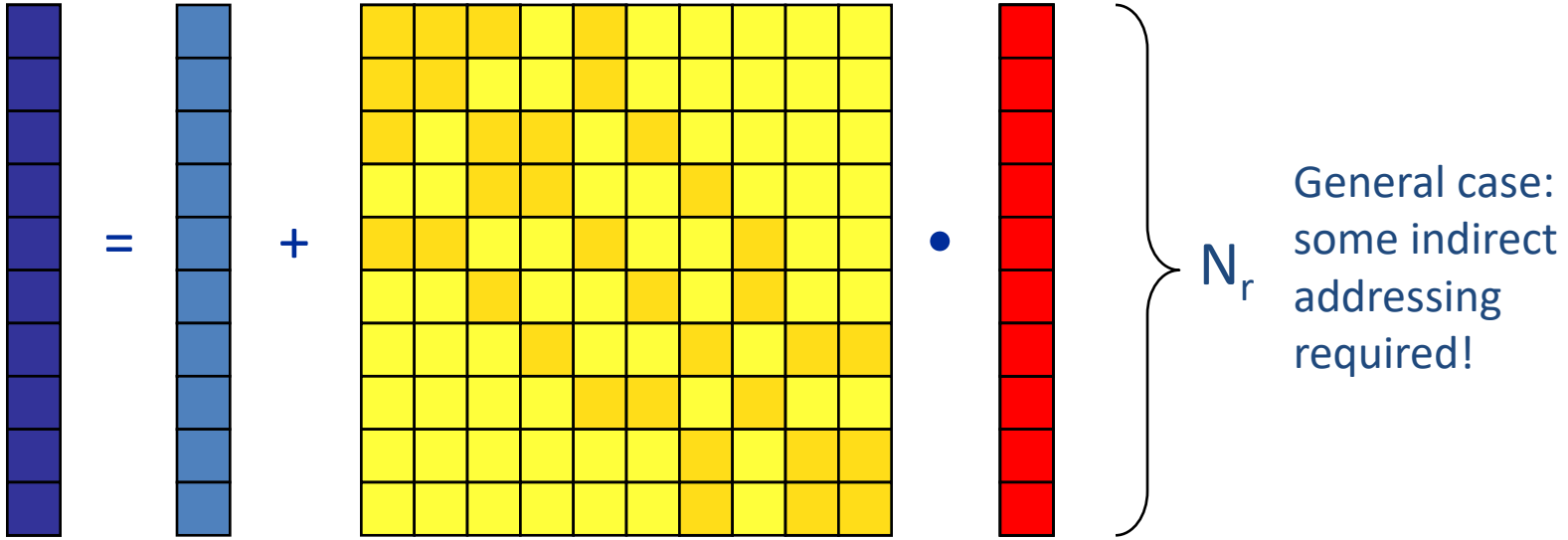
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: Sparse Matrix-Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in numerous sparse linear algebra solvers

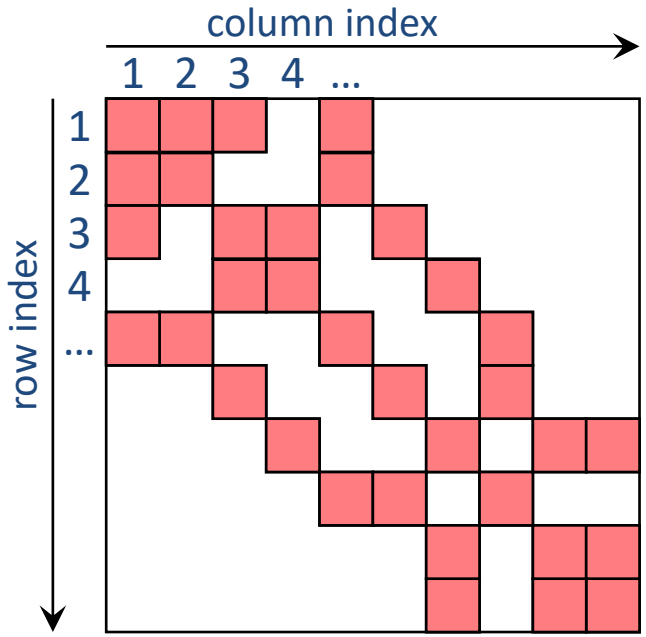
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- "Sparse": $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzr} = N_{nz}/N_r$



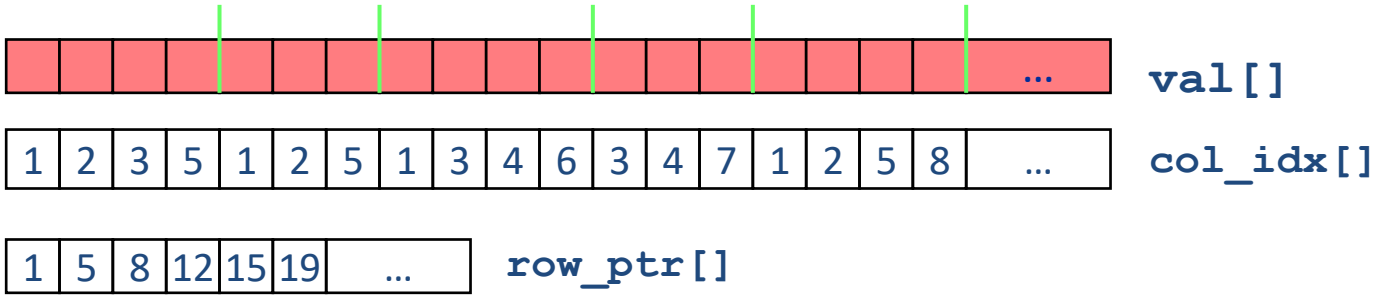
SpMVM characteristics

- 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



- **`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:

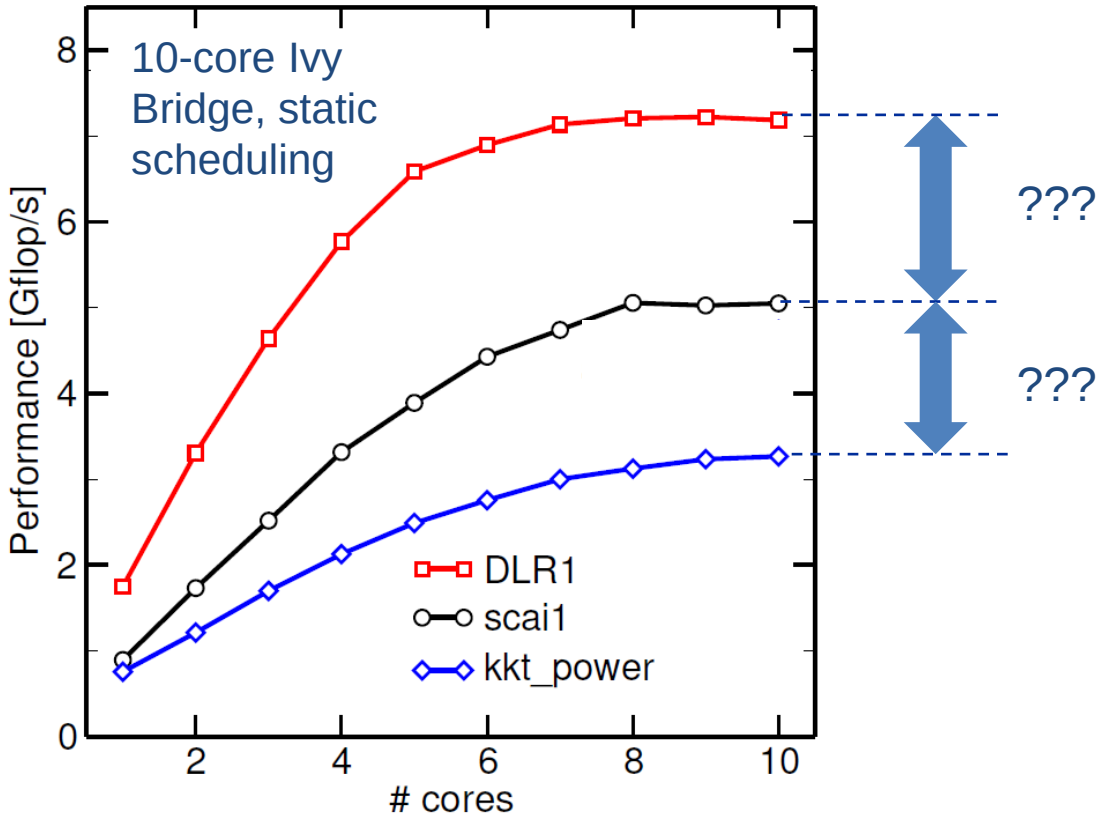
```
!$OMP parallel do schedule(???)
do i = 1,N_r
 do j = row_ptr(i), row_ptr(i+1) - 1
  C(i) = C(i) + val(j) * B(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

  - 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?



SpMV node performance model – CRS (1)

```
do i = 1, N_r
 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(N_nz)
integer*4  col_idx(N_nz)
integer*4  row_ptr(N_r)
real*8     C(N_r)
real*8     B(N_c)
```

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]: $2\,N_{nz}$

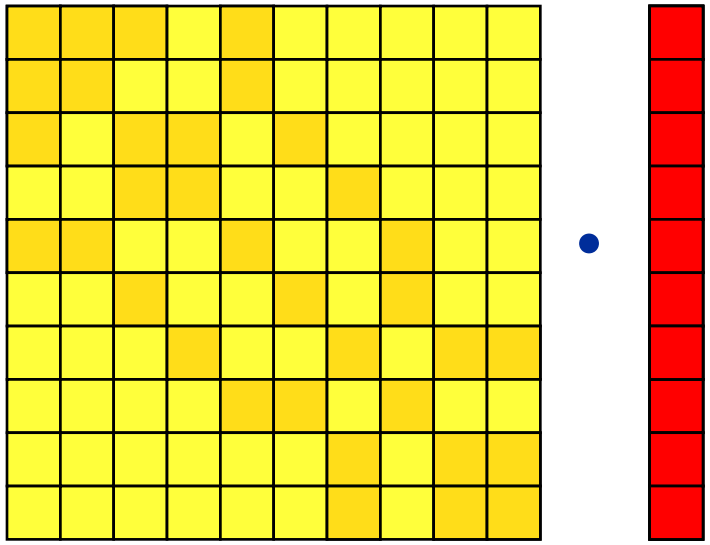
$$B_{C,min} = \frac{12\,N_{nz} + 20\,N_r + 8\,N_c}{2\,N_{nz}}\frac{B}{F} = \frac{12 + 20/N_{nzr} + 8/N_{nzc}}{2}\frac{B}{F}$$

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

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

SpMV node performance model – CRS (2)

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



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

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

Parameter (α) quantifies additional traffic for `B(:)` (irregular access):

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

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

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

Note: $B_C\left(1/N_{nzr}\right) = B_{C,min}$

The "α effect"

DP CRS code balance

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

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

$$= \left(6 + 4\,\alpha + \frac{10}{N_{nzr}}\right)\frac{B}{F}$$

Can we predict α?

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

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

Determine α (RHS traffic quantification)

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

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

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

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

- $N_{nz} = 14.6\cdot 10^6,\ N_{nzr} = 7.1$
- $V_{meas} \approx 258\ \mathrm{MB}$

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

→ RHS is loaded 2.5 times from memory

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

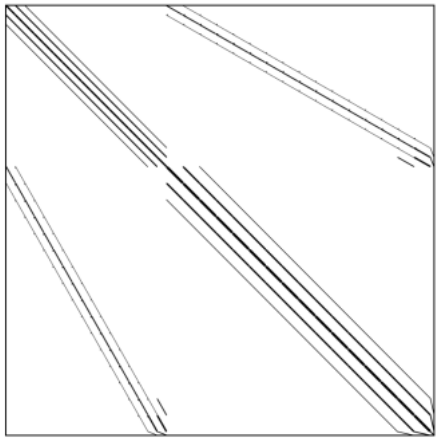
11% extra traffic → optimization potential!

Three different sparse matrices

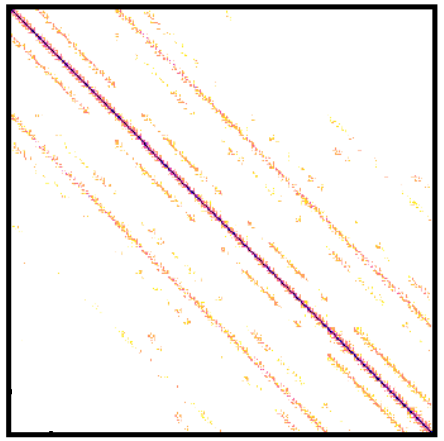
Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_S = 46.6\,\mathrm{GB/s}$

→ Roofline: $P_{opt} = {b_S}/{B_{C,min}}$

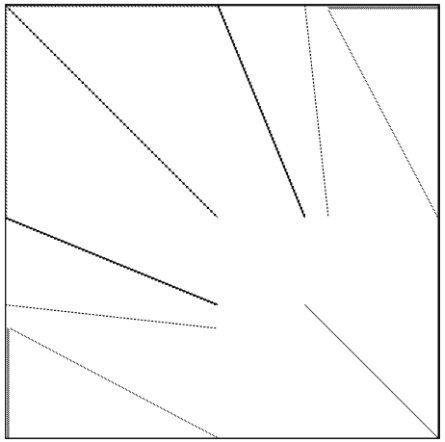
Matrix	N	N_{nzr}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

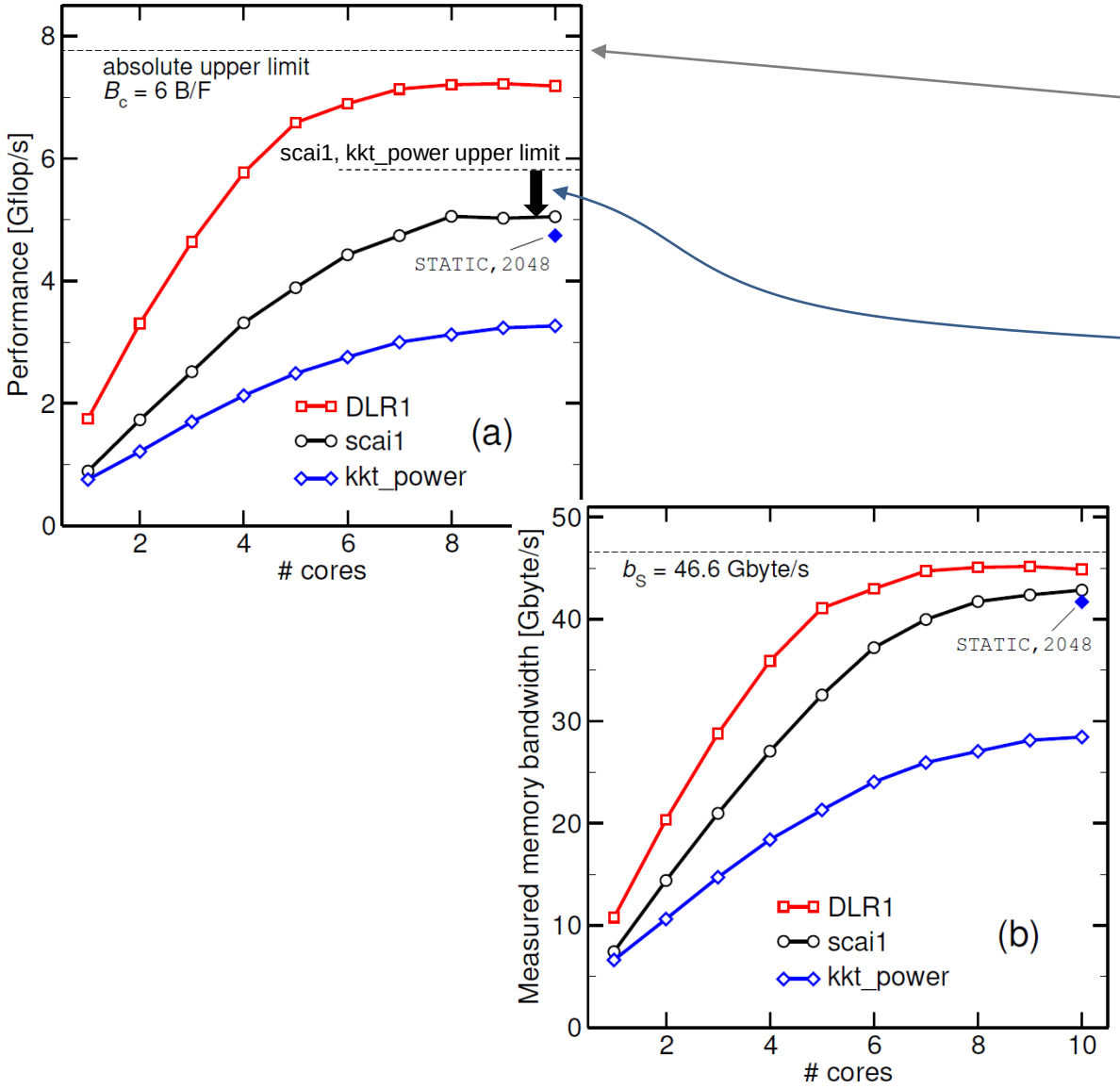


scai1



kkt_power

Now back to the start…



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

$$P_{max} = 7.8\,\mathrm{GF/s}$$

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

- **scai1** measured balance:

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

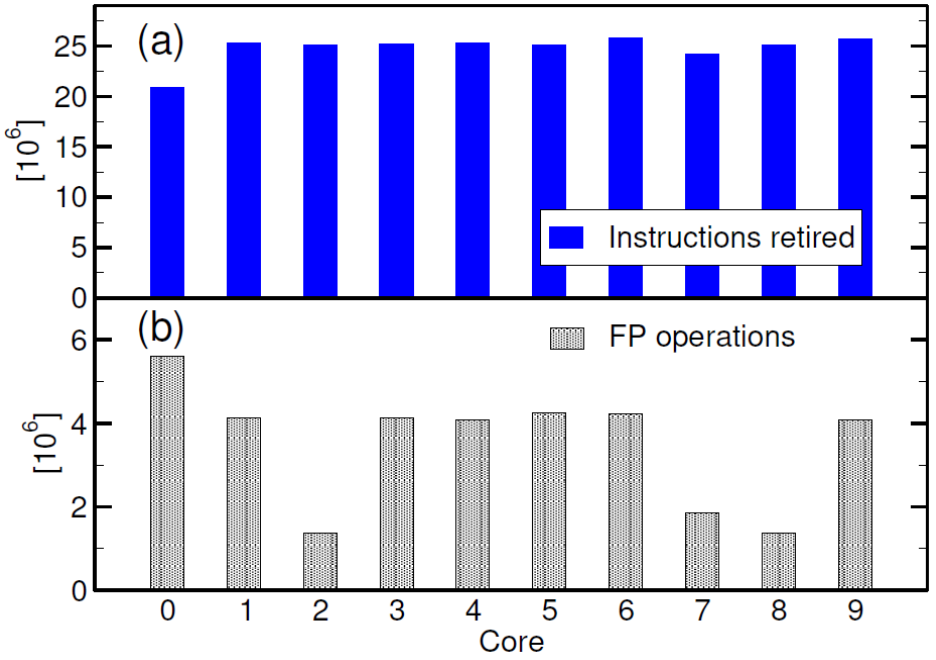
→ good BW utilization, slightly non-optimal α

- **kkt_power** measured balance:

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

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

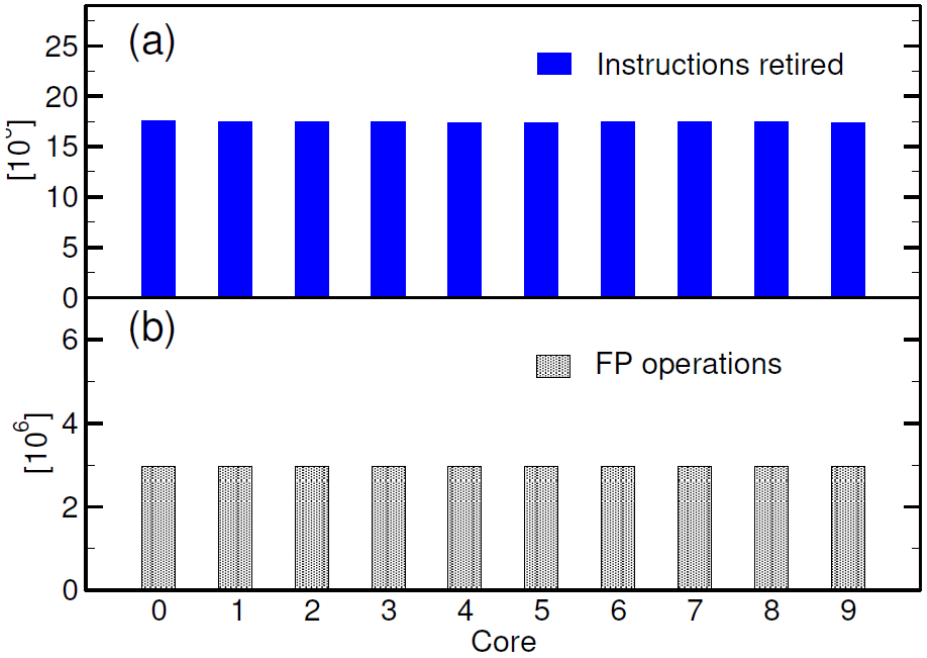
Investigating the load imbalance with kkt_power



Measurements with likwid-perfctr (MEM_DP group)

`static`

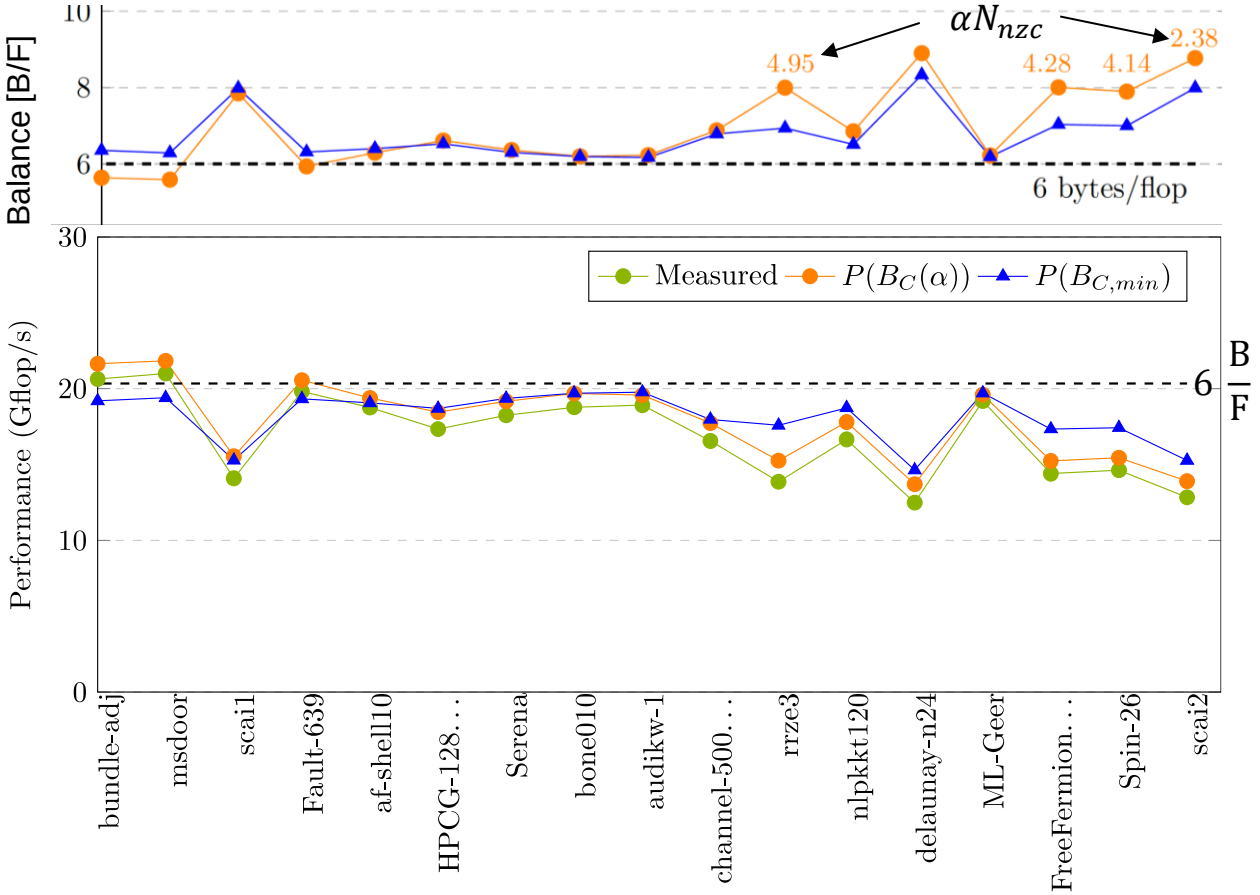
`static,2048`



→ 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)
$b_S = 122\ GB/s$

Matrices taken from: C. L. Alappat et al.: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.*
DOI: 10.1002/cpe.6512

When Roofline for SpMV may not work

Reasons for performance not attaining the limit

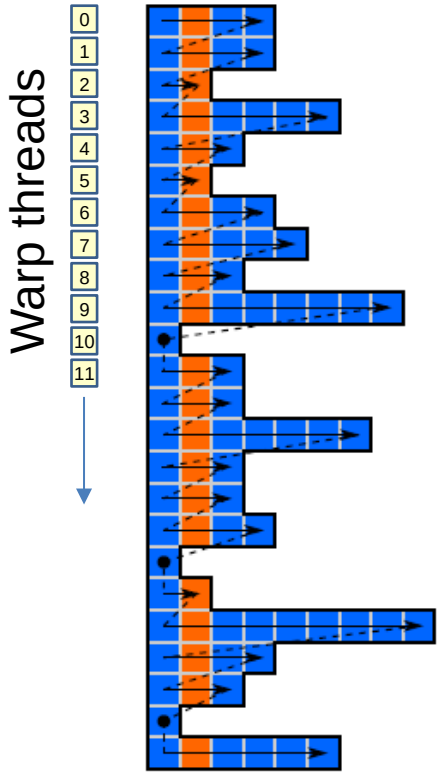
1. Intensity lower than the minimum
   - More RHS traffic than the optimistic limit ($\frac{4}{N_{nzr}}$ B/F)
2. "Slow code"
   - "invisible" performance ceiling due to inefficient instructions or inefficient execution
3. Load imbalance
   - A single process/thread cannot saturate the memory bandwidth
4. Erratic memory access patterns for RHS
   - Latency dominates

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,…



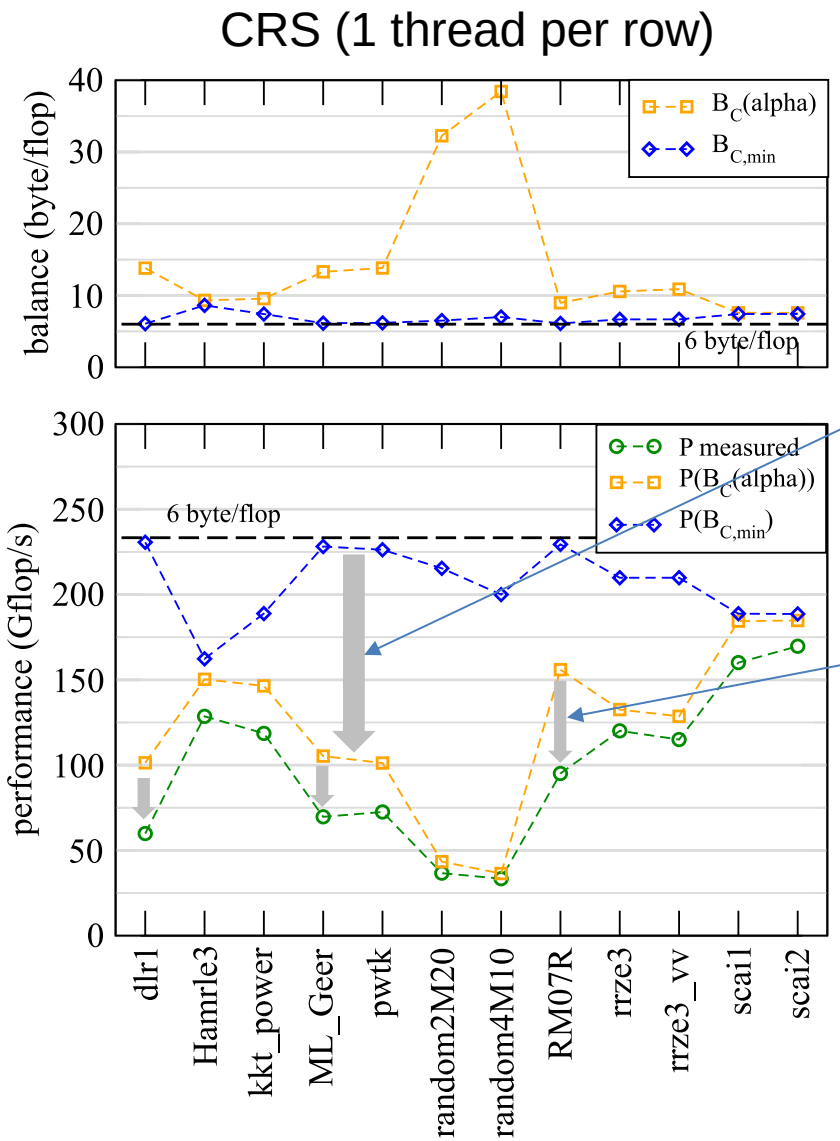
CRS SpMV in CUDA (y = Ax)

```
template <typename VT, typename IT>
__global__ static void
spmv_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;
    }
}
```

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

No write-allocate on GPUs for consecutive stores

SpMV CRS performance on a GPU



NVIDIA Ampere A100
Memory bandwidth $b_S = 1400\ \mathrm{GB/s}$

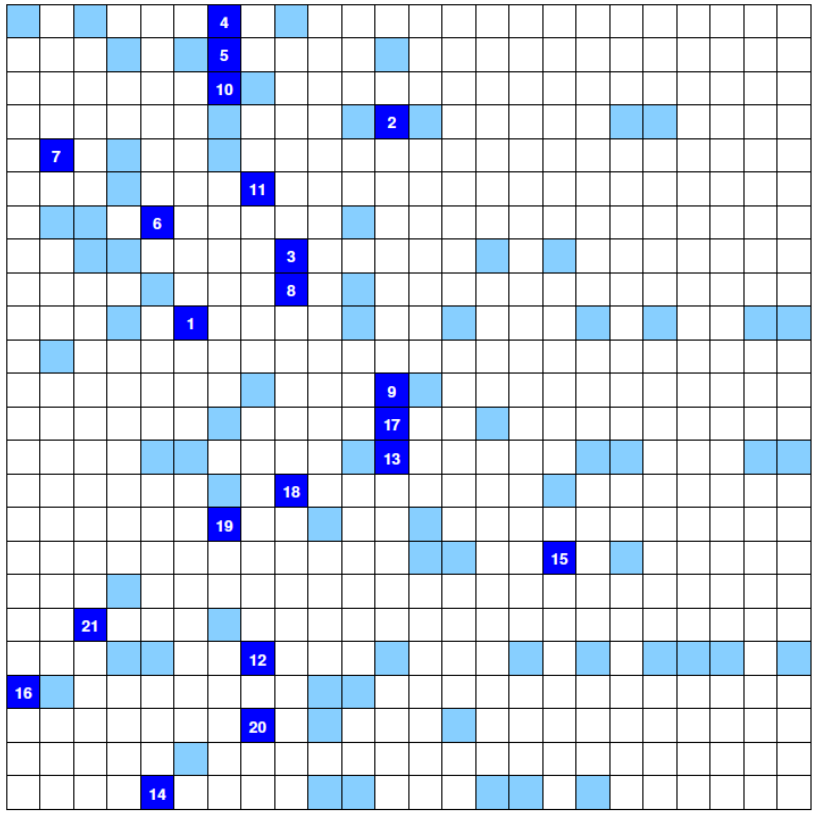
- Strong "α effect" – large deviation from optimal α for many matrices
  - Many cache lines touched b/c every thread handles one row → bad cache usage
- Mediocre memory bandwidth usage ($\ll 1400\ \mathrm{GB/s}$) in many cases
  - Non-coalesced memory access
  - Imbalance across rows/threads of warps

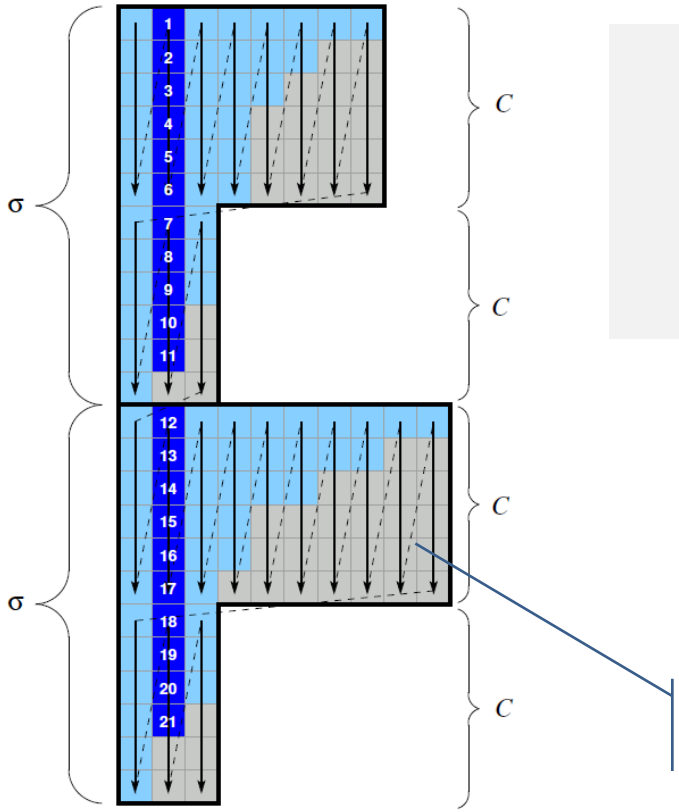
SELL-C-σ

M. Kreutzer et al.: *A Unified Sparse Matrix Data Format For Efficient General Sparse Matrix-vector Multiplication On Modern Processors With Wide SIMD Units*, SIAM SISC 2014, DOI: 10.1137/130930352

Idea

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





"Chunk occupancy":

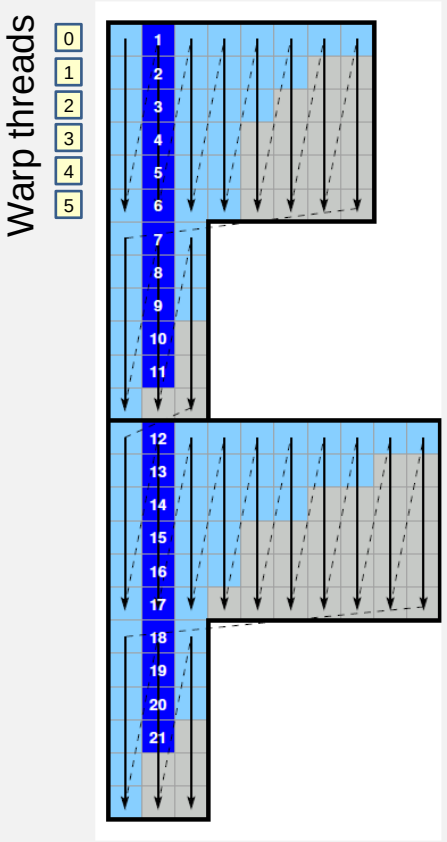
$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$

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)

Matrix data & column index

LHS update (write only)

chunk index

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

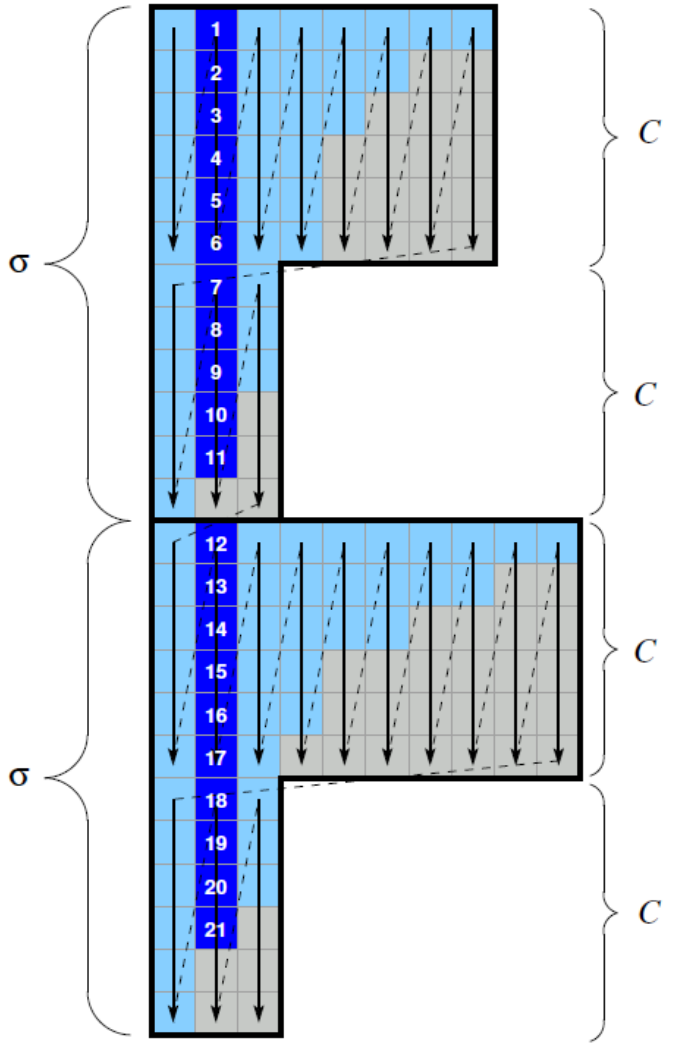
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzr}}\right)\frac{\text{bytes}}{\text{flop}}$$

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

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

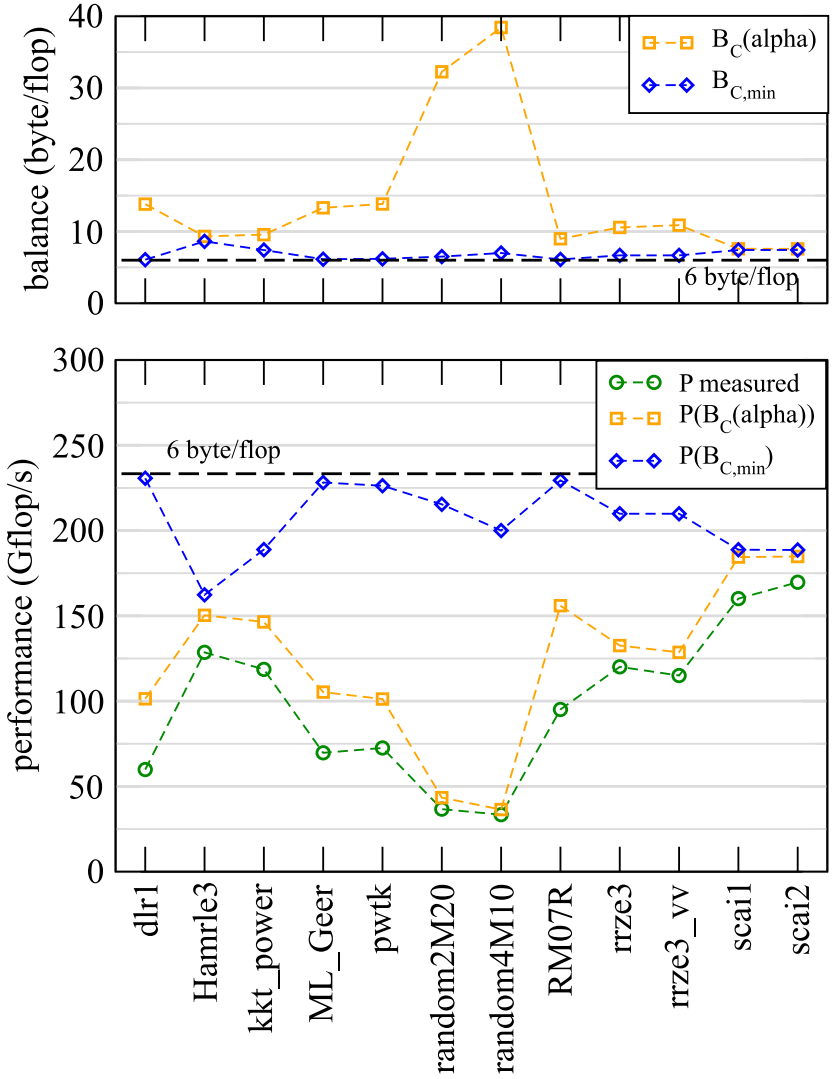
How to choose the parameters C and σ on GPUs?

- 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 → α depends on σ



SpMV node performance model – GPU

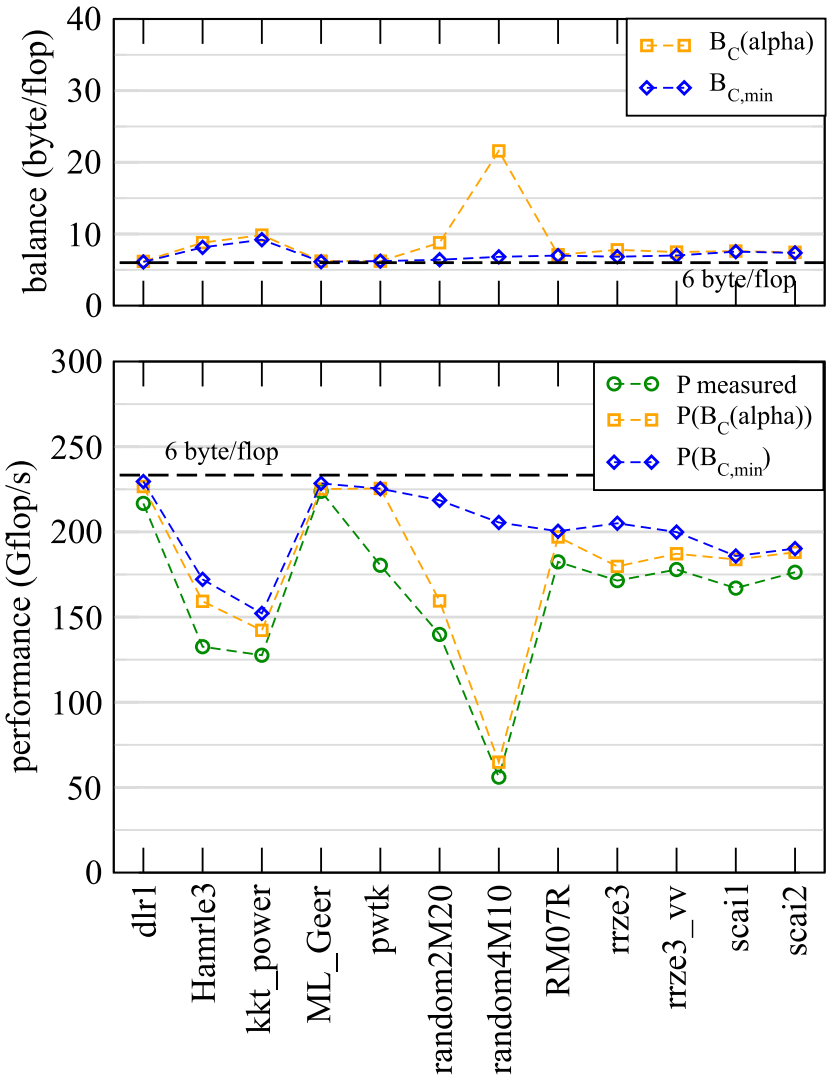
CRS (1 thread per row)



NVIDIA Ampere A100

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

SELL-32-128



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!
- Sparse matrix times multiple vectors bears the potential of huge savings in data volume
- Consequence: Modeling is not always 100% predictive. It‘s all about *learning more* about performance properties!

Tutorial conclusion

- Think simple!
  - cogwheels → data pumps
  - machine parts → bottlenecks
  - shot-in-the-dark optimizations → scientific thinking
- Back-of-the-envelope performance engineering is often sufficient
- Tools can help, but resist the temptation of data
- Your brain is the most powerful performance tool

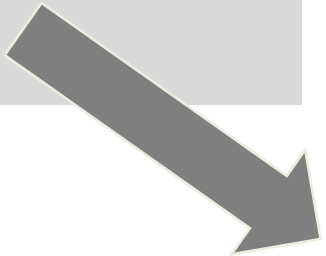
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case Study: Dense Matrix-Vector Multiplication

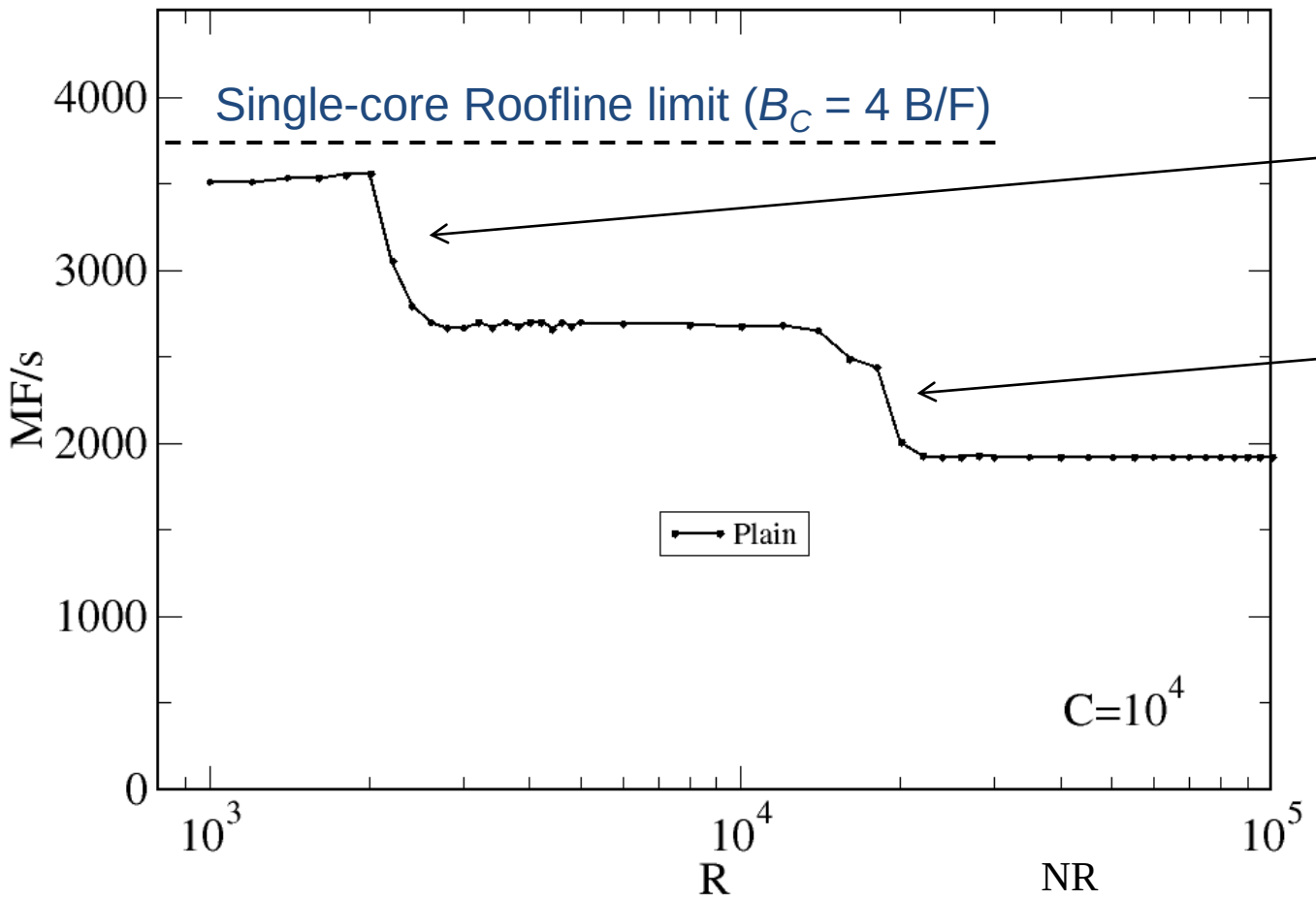
Dense matrix-vector multiplication in DP

```
do c = 1 , NC
 do r = 1 , NR
     y(r)=y(r) + A(r,c)* x(c)
 enddo
enddo
```



```
do c = 1 , NC
   tmp=x(c)
   do r = 1 , NR
      y(r)=y(r) + A(r,c)* tmp
   enddo
enddo
```

DMVM (DP) – Single core performance vs. column height



NC=10^4

Performance drops as number of rows (inner loop length) increases.

Does computational intensity change?

Intel Xeon E5 2695 v3 (Haswell-EP), 2.3 GHz, CoD mode, Core P_{max}=18.4 GF/s,
Caches: 32 KB / 256 KB / 35 MB, PageSize: 2 MB; ifort V15.0.1.133; b_S = 32 Gbyte/s

DMVM data traffic analysis

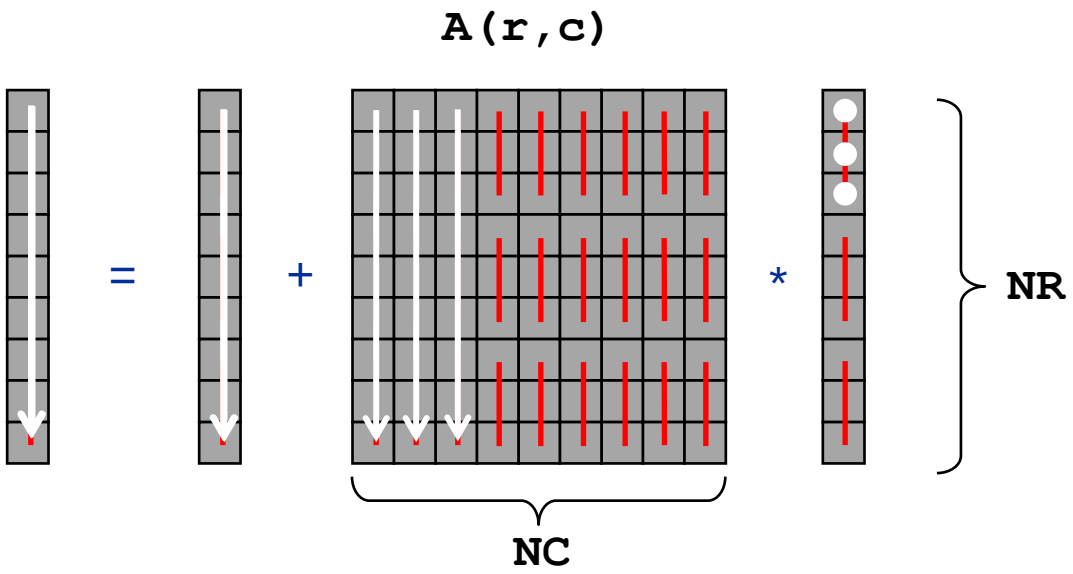
```
do c = 1 , NC
  tmp=x(c)
  do r = 1 , NR
     y(r)=y(r) + A(r,c)* tmp
  enddo
enddo
```

`tmp` stays in a register during inner loop

`A(:,:)` is loaded from memory – no data reuse

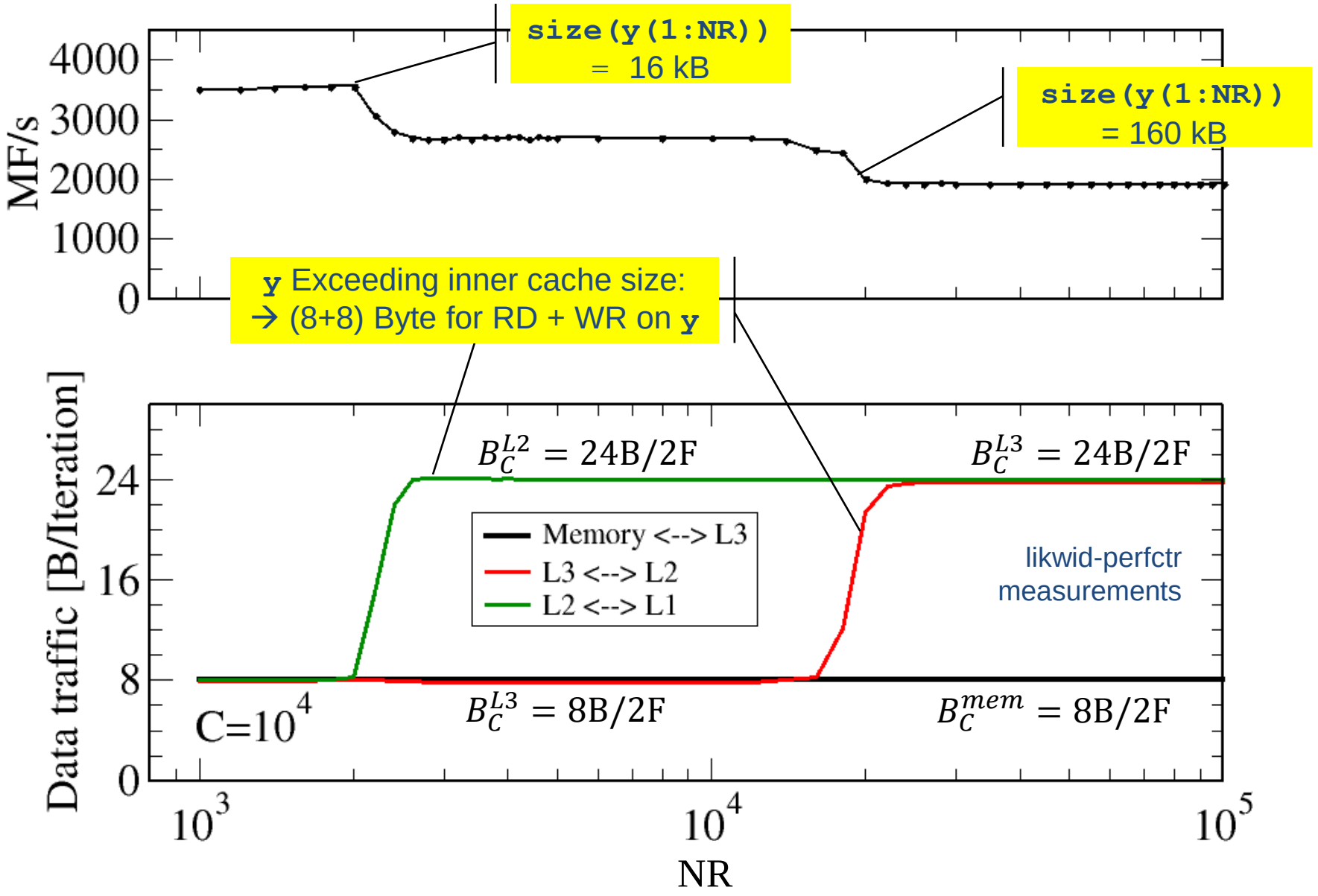
`y(:)` is loaded and stored in each outer iteration
→ for c>1 update `y(:)` in cache

`y(:)` may not fit in innermost cache → more traffic from lower level caches for larger `NR`

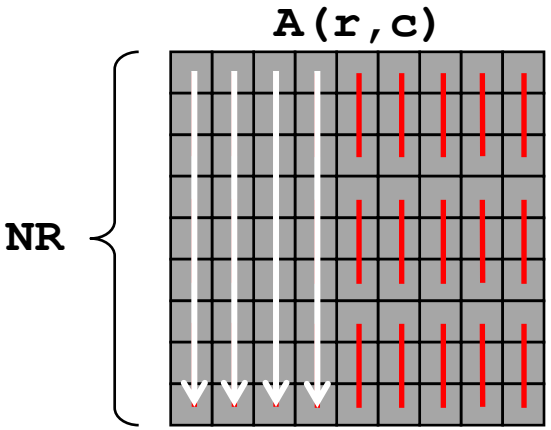


Analysis: Distinguish code balance in memory (B_C^{mem}) from code balance in relevant cache level(s) (B_C^{L3}, B_C^{L2},…)!

DMVM (DP) – Single core data traffic analysis

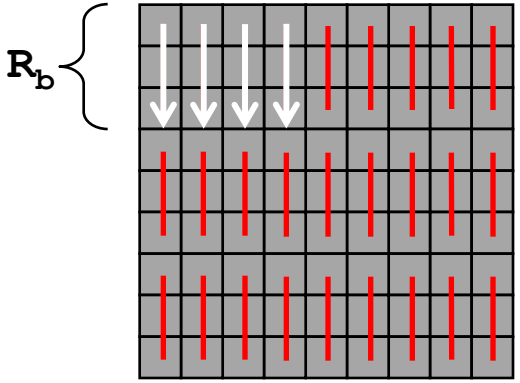


Reducing traffic by blocking




```
do c = 1 , NC
  tmp=x(c)
  do r = 1 , NR
     y(r)=y(r) + A(r,c)* tmp
  enddo
enddo
```

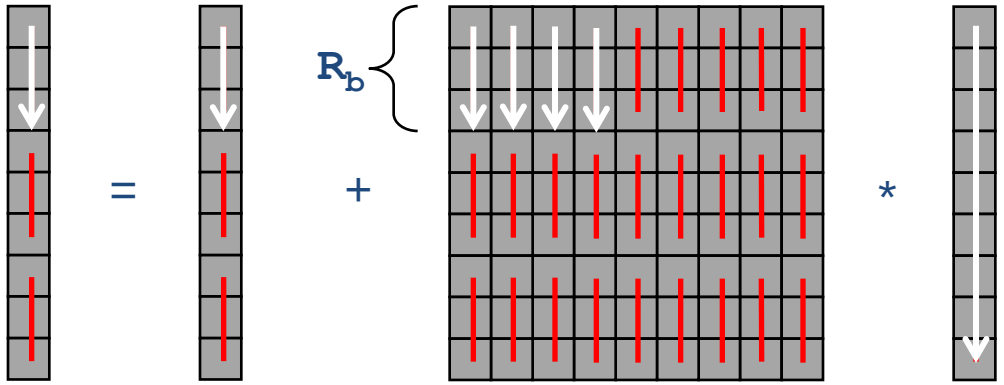
`y(:)` may not fit into some cache → more traffic for lower level




```
do rb = 1 , NR , Rb
 rbS = rb
 rbE = min((rb+Rb-1), NR)
 do c = 1 , NC
   do r = rbS , rbE
      y(r)=y(r) + A(r,c)*x(c)
   enddo
 enddo
enddo
```

`y(rbS:rbE)` may fit into some cache if R_b is small enough
→ traffic reduction

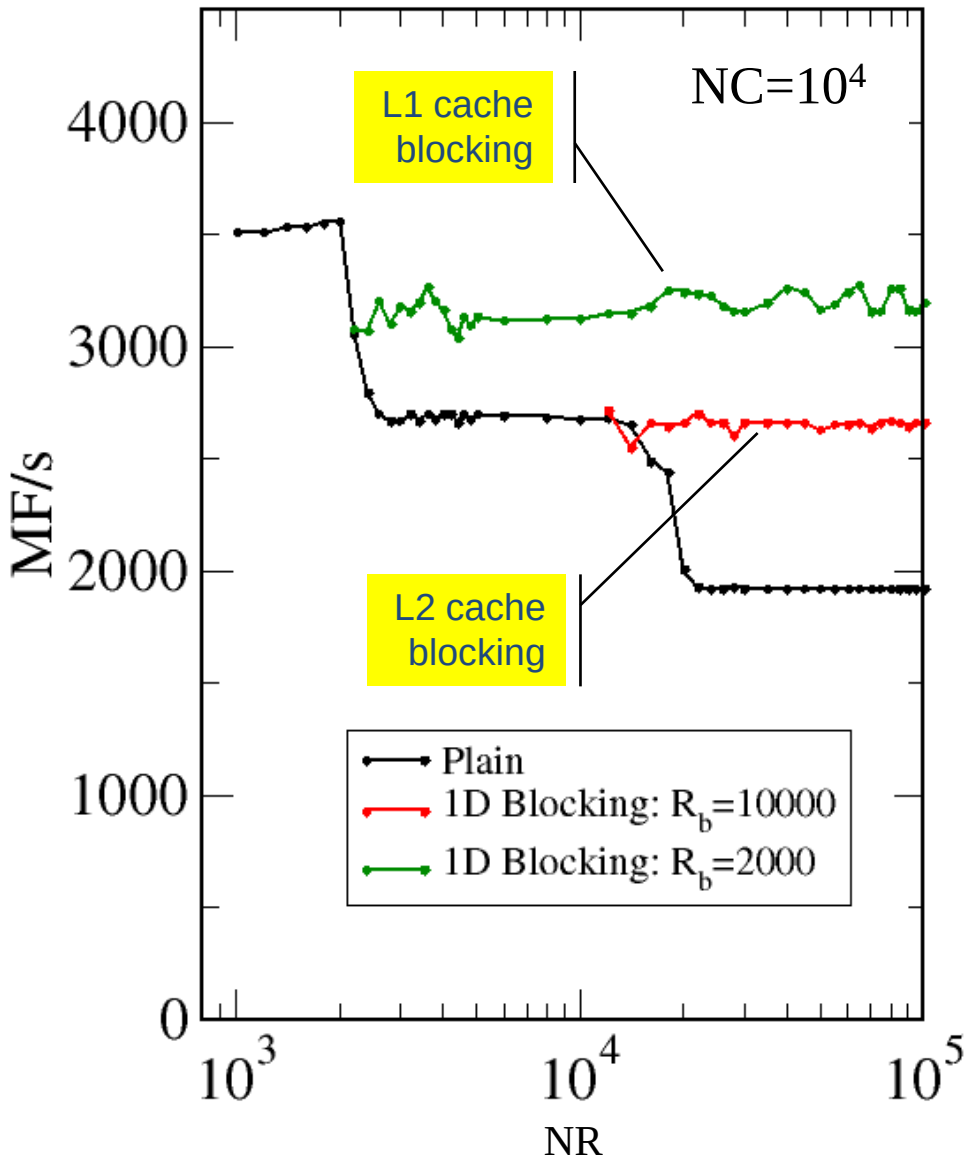
Reducing traffic by blocking



- LHS only updated once in some cache level if blocking is applied
- Price: RHS is loaded multiple times instead of once!
  - How often? → N_R / R_b times
  - RHS traffic: N_C x N_R / R_b
  - LHS traffic: 2 x N_R
  - Matrix: N_R x N_C

  Overall: $N_R \times \left(\frac{C}{R_b} + 2 + N_R\right) \approx N_R^2$ if $N_R, R_b \gg 1$ and $N_C = N_R$
- Without blocking: $N_R \times \left(\frac{N_C}{N_R} + 2N_C + N_R\right) \approx 3N_R^2$ if $N_R, R_b \gg 1$ and $N_C = N_R$

DMVM (DP) – Reducing traffic by inner loop blocking

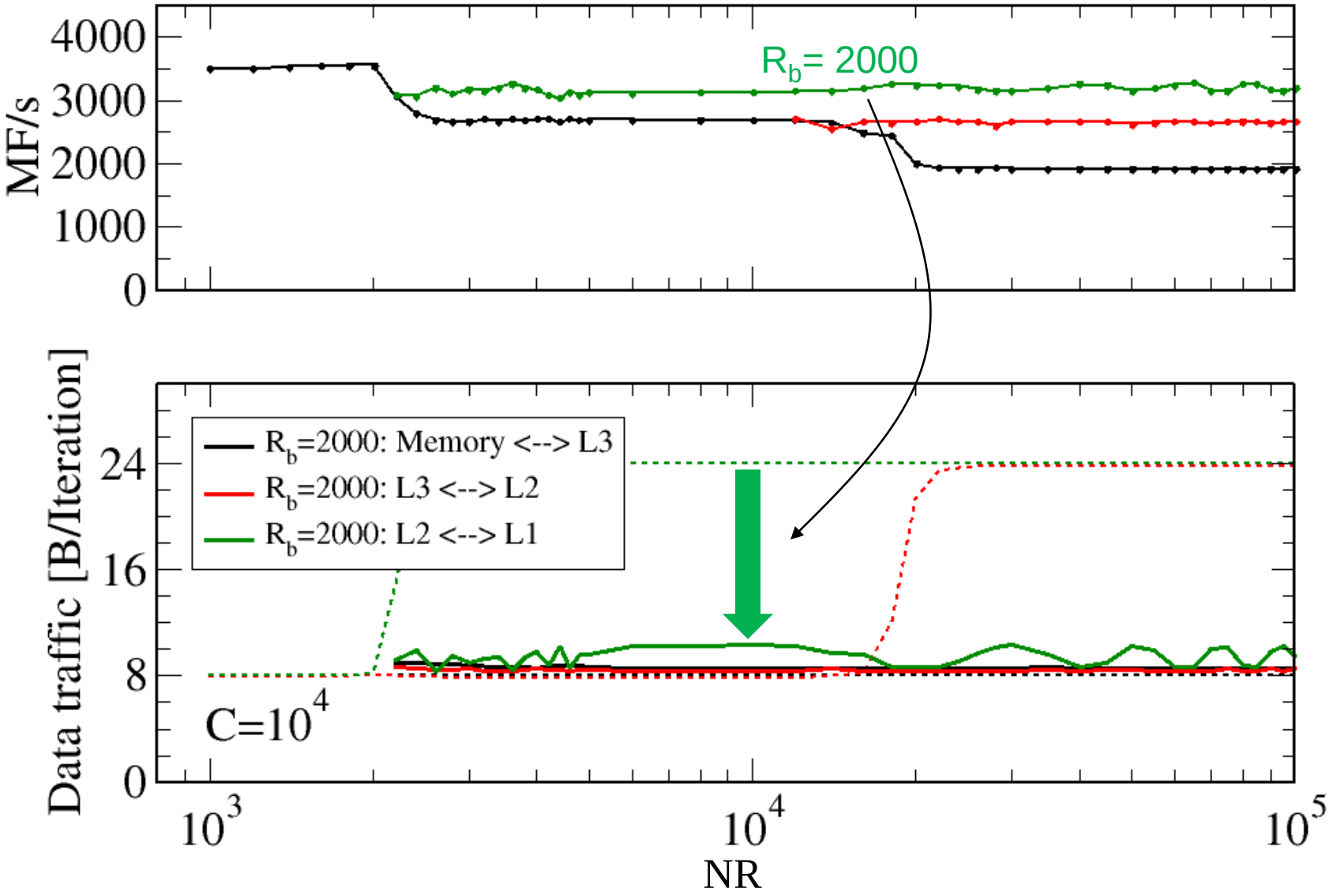


- "1D blocking" for inner loop
- Blocking factor $\mathbf{R_b}$ ←→ cache level

```
do rb = 1 , NR , Rb

 rbS = rb
 rbE = min((rb+Rb-1), NR)

 do c = 1 , NC
   do r = rbS , rbE
      y(r)=y(r) + A(r,c)*x(c)
   enddo
 enddo

enddo
```

→ Fully reuse subset of `y(rbS:rbE)` from L1/L2 cache

DMVM (DP) – Validation of blocking optimization



DMVM (DP) – OpenMP parallelization

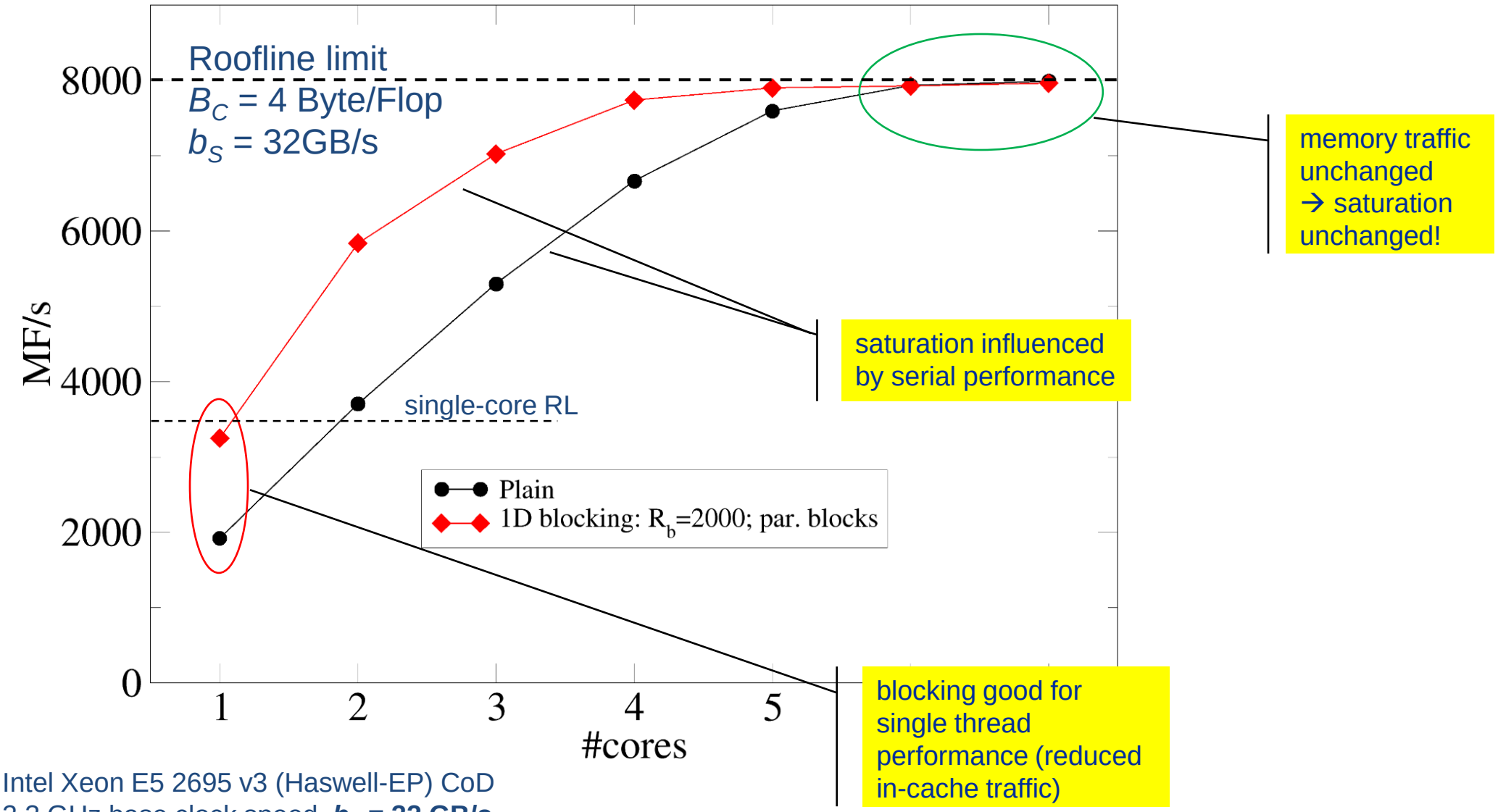
```
!$omp parallel do reduction(+:y)
do c = 1 , NC
  do r = 1 , NR
     y(r) = y(r) + A(r,c) * x(c)
enddo ; enddo
!$omp end parallel do
```

plain code

```
!$omp parallel do private(rbS,rbE)
do rb = 1 , NR , Rb
 rbS = rb
 rbE = min((rb+Rb-1), NR)
 do c = 1 , NC
   do r = rbS , rbE
      y(r) = y(r) + A(r,c) * x(c)
enddo ; enddo ; enddo
!$omp end parallel do
```

blocked code

DMVM (DP) – OpenMP parallelization & saturation



Intel Xeon E5 2695 v3 (Haswell-EP) CoD
2.3 GHz base clock speed, ***b*$_S$ = 32 GB/s**

Conclusions from the dMVM example

- We have found the reasons for the breakdown of single-core performance with growing number of matrix rows
  - LHS vector fitting in different levels of the cache hierarchy
  - Validated theory by performance counter measurements
- Inner loop blocking was employed to improve code balance in L3 and/or L2
  - Validated by performance counter measurements
- Blocking led to better single-threaded performance
- Saturated performance unchanged (as predicted by Roofline)
  - Because the problem is still small enough to fit the LHS at least into the L3 cache