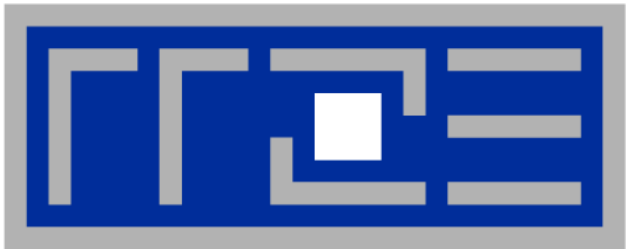
For final slides and example code see:

https://tiny.cc/NLPE-SC19



Node-Level Performance Engineering

Georg Hager, Jan Eitzinger, Gerhard Wellein
Erlangen Regional Computing Center (RRZE)
and Department of Computer Science
University of Erlangen-Nuremberg

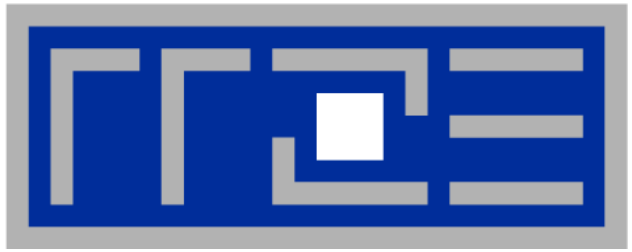
SC19 full-day tutorial
November 17, 2019
Denver, CO

SC19
Denver, CO | hpc is now.


: slide updated

Agenda

- **Preliminaries** — 08:30
- **Introduction to multicore architecture**
  - Threads, cores, SIMD, caches, chips, sockets, ccNUMA
- **Multicore tools (part I)** — 10:00
- **Microbenchmarking for architectural exploration** — 10:30
  - Streaming benchmarks
  - Hardware bottlenecks
- **Node-level performance modeling (part I)**
  - The Roofline Model — 12:00
- **Lunch break**
- **Multicore tools (part II)** — 13:30
- **Node-level performance modeling (part II)**
  - Case studies: Jacobi solver, sparse MVM, tall & skinny MM — 15:00
- **Optimal resource utilization** — 15:30
  - SIMD parallelism
  - ccNUMA
  - OpenMP synchronization and multicores — 17:00

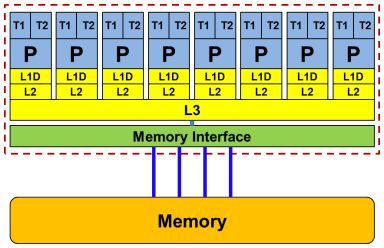


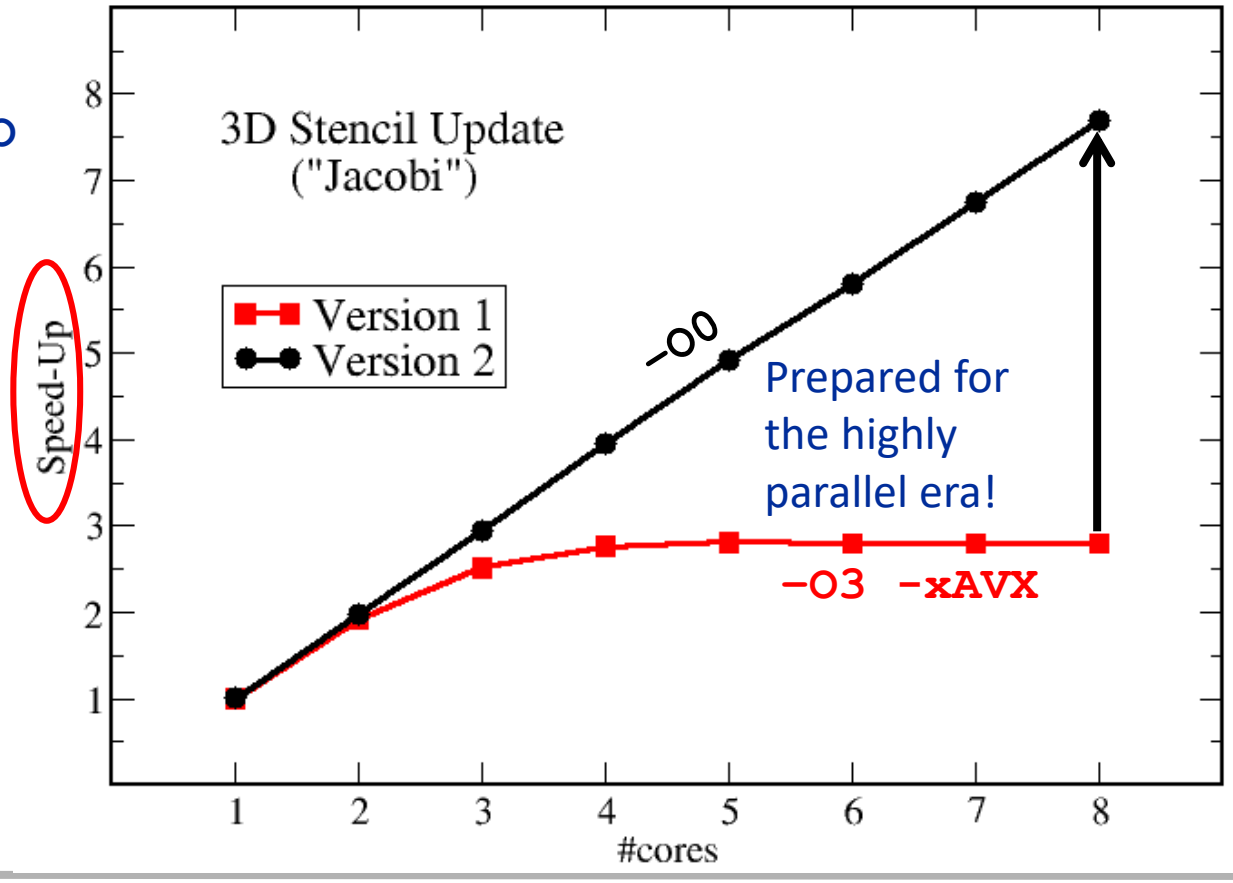
Prelude:
Scalability 4 the win!

Scalability Myth: Code scalability is the key issue

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

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

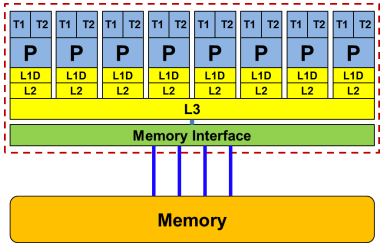


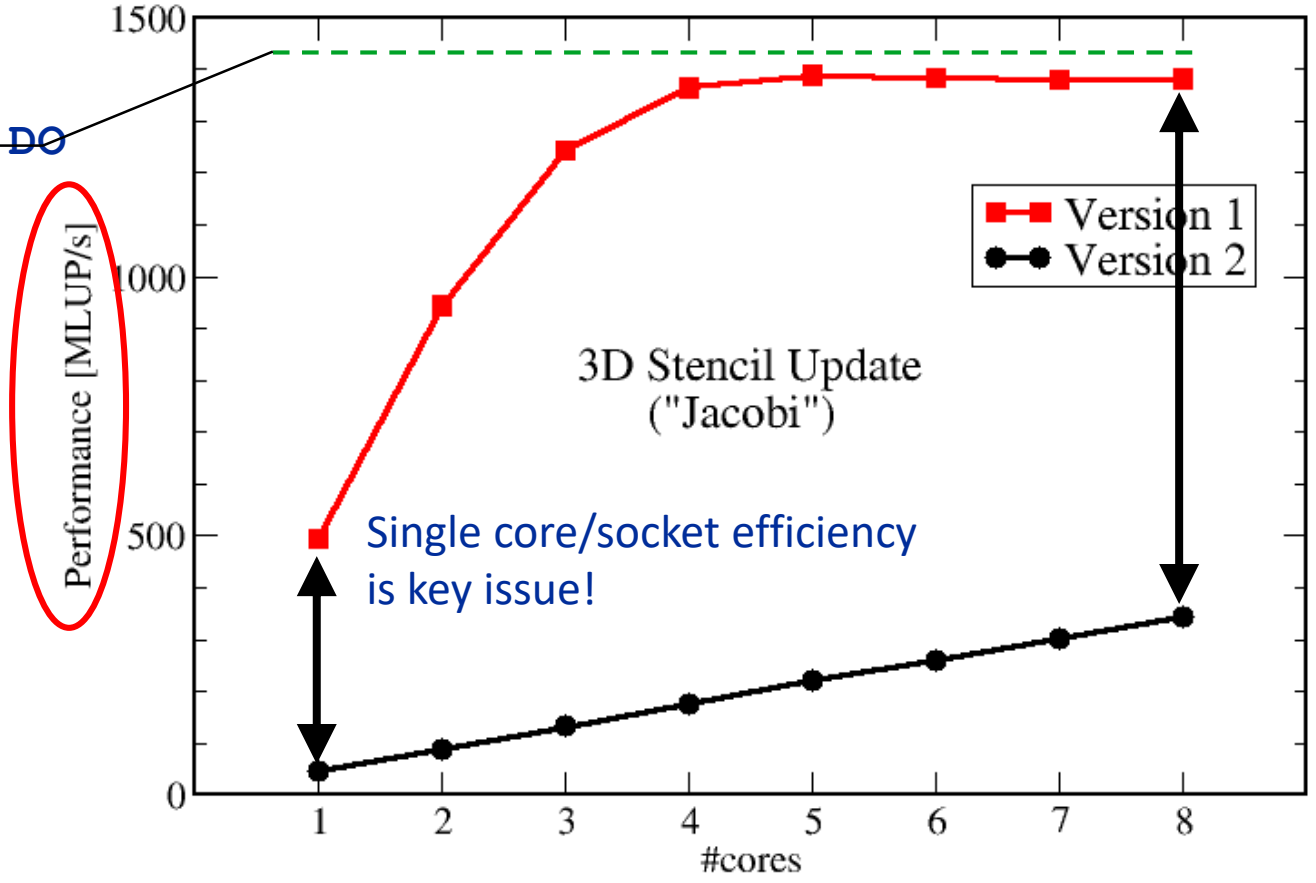


Scalability Myth: Code scalability is the key issue

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

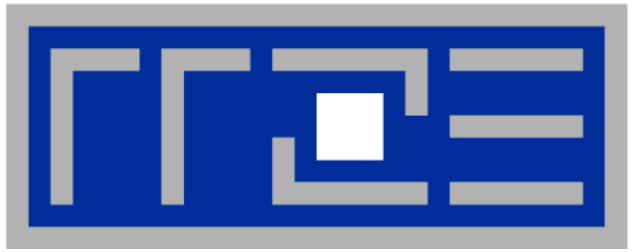
Upper limit from simple performance model:
35 GB/s & 24 Byte/update





Questions to ask in high performance computing

- **Do I understand the performance behavior of my code?**
  - Does the performance match a model I have made?
- **What is the optimal performance for my code on a given machine?**
  - High Performance Computing == Computing at the bottleneck
- **Can I change my code so that the "optimal performance" gets higher?**
  - Circumventing/ameliorating the impact of the bottleneck
- **My model does not work – what's wrong?**
  - This is the good case, because you learn something
  - Performance monitoring / microbenchmarking may help clear up the situation



Introduction:
Modern node architecture

A glance at basic core features:

pipelining, superscalarity, SMT, SIMD

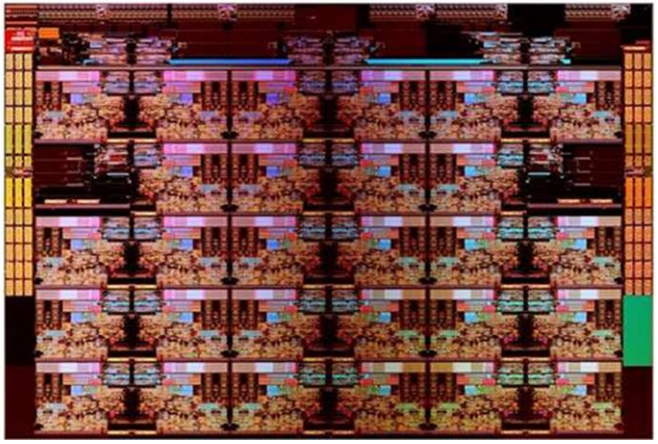
Caches and data transfers through the memory hierarchy

Accelerators

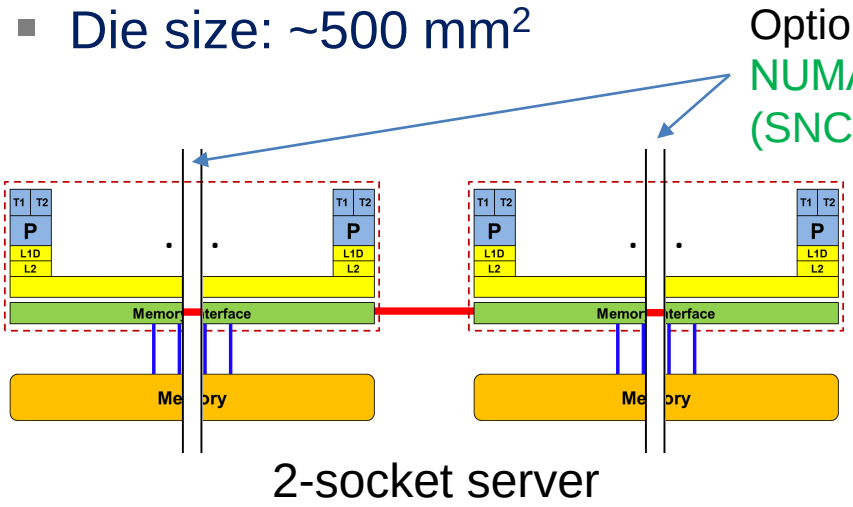
Bottlenecks & hardware-software interaction

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

- Xeon "Skylake SP" (Platinum/Gold/Silver/Bronze):
  Up to 28 cores running at 2+ GHz (+ "Turbo Mode": 3.8+ GHz)
  - Mesh interconnect
  - Reincarnated as "Cascade Lake" in 2018
- Simultaneous Multithreading
  → reports as 56-way chip

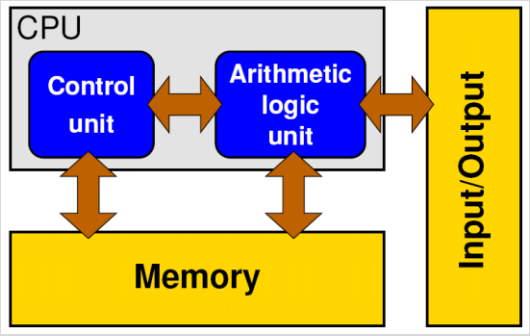


- 8 Billion Transistors / 14 nm
- Die size: ~500 mm²



2015: Broadwell architecture
- Ring instead of mesh interconnect
- Cluster on Die (analogous to SNC)
- Up to 24 cores

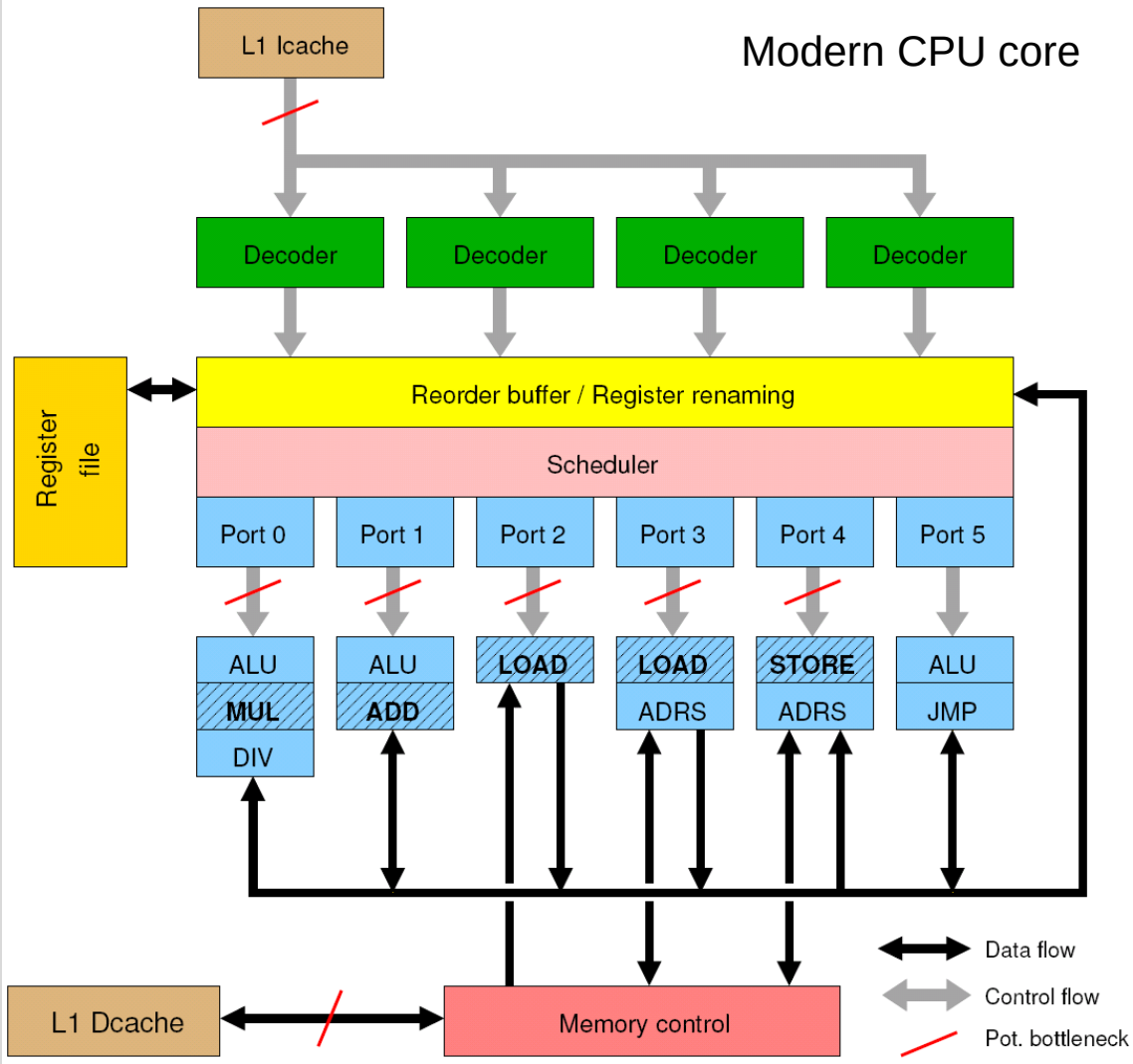
General-purpose cache based microprocessor core



Stored-program computer

- 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



Pipelining of arithmetic/functional units

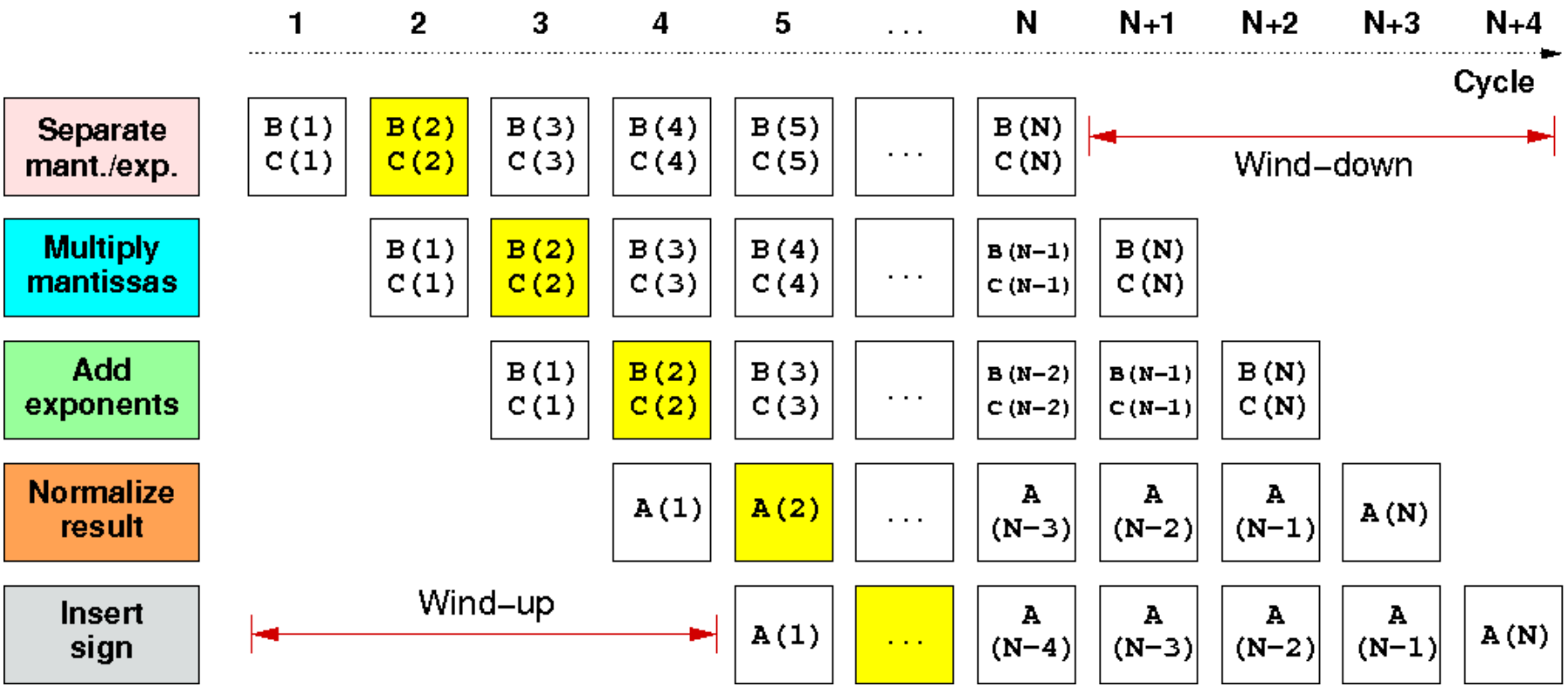
- Idea:
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same time, e.g., one cycle
  - Execute different steps on different instructions at the same time (in parallel)

- Allows for shorter cycle times (simpler logic circuits), e.g.:
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultaneously
  - one result at each cycle after the pipeline is full

- Drawback:
  - Pipeline must be filled – sufficient # of independent instructions required
  - Requires complex instruction scheduling by compiler/hardware
    - software-pipelining / out-of-order execution

- Pipelining is widely used in modern computer architectures

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

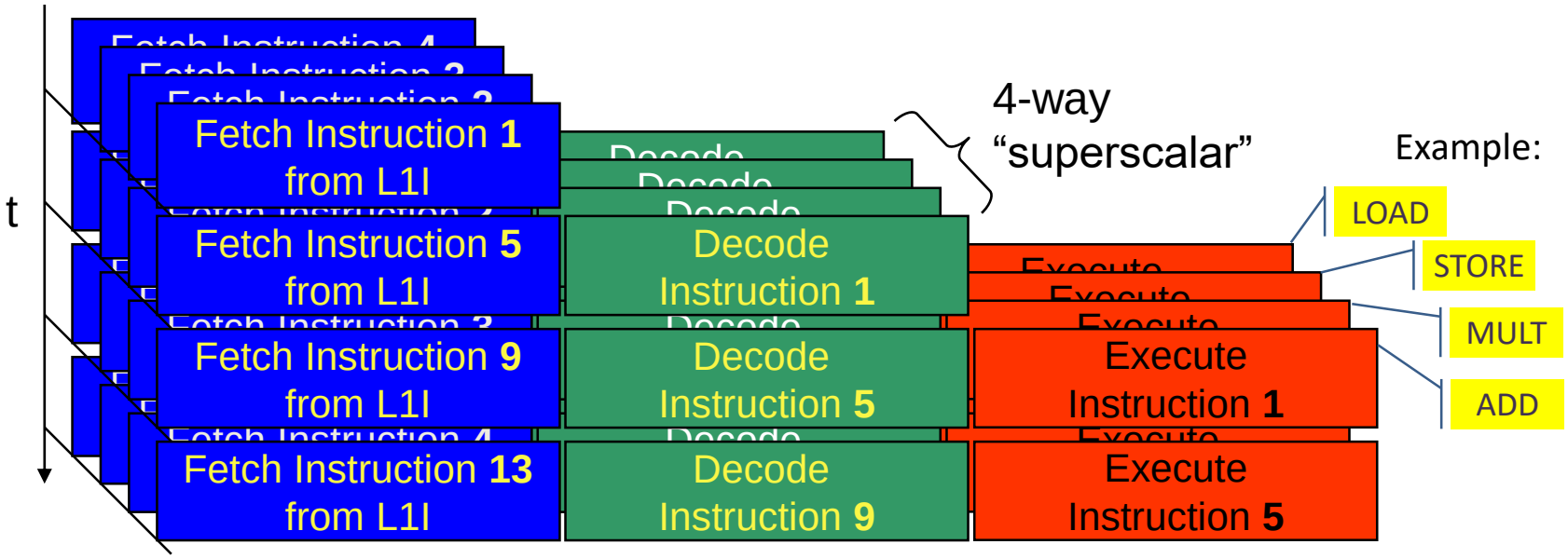


First result is available after 5 cycles (=latency of pipeline)!

Wind-up/-down phases: Empty pipeline stages

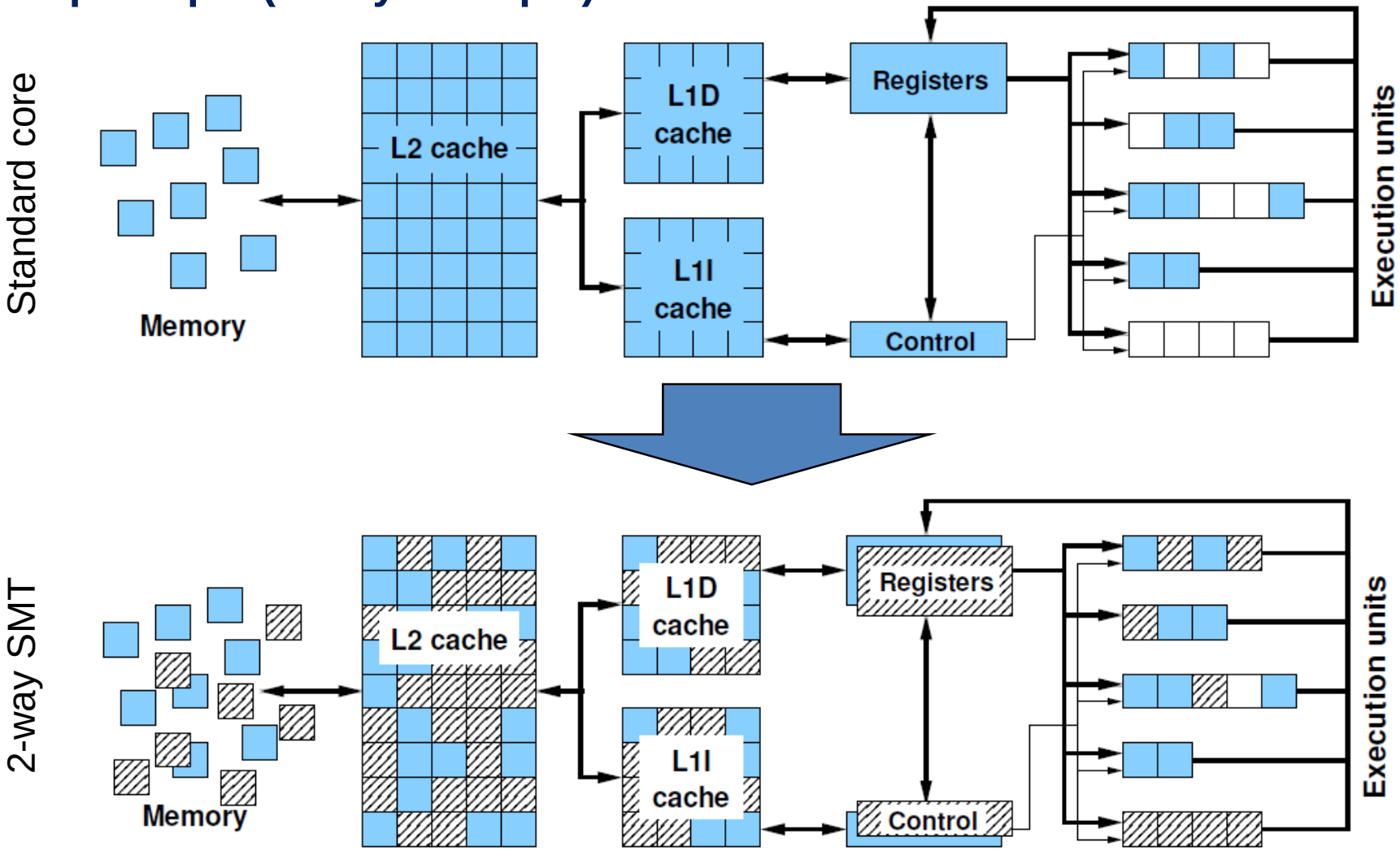
Superscalar Processors – Instruction Level Parallelism

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

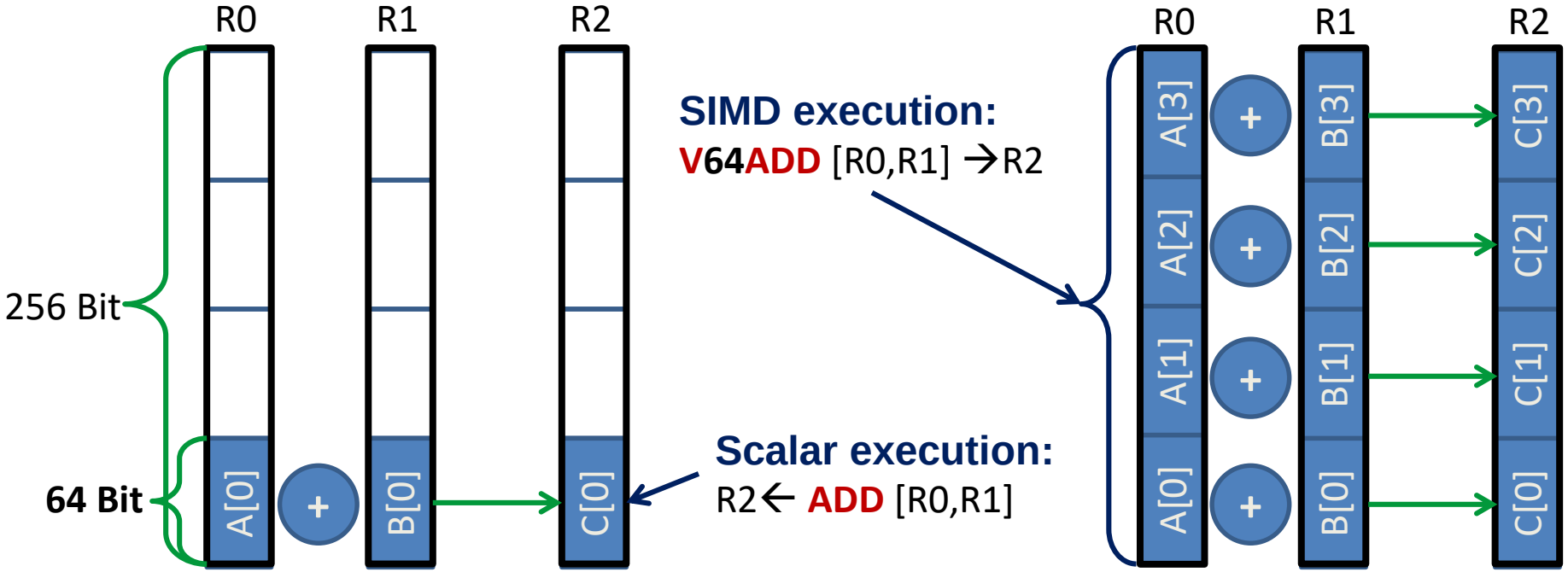


- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 4- to 6-way superscalar & can perform 2 floating-point instructions per cycle

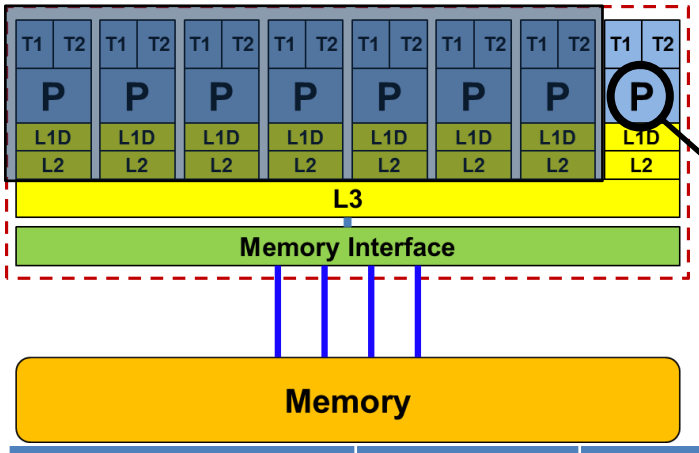
SMT principle (2-way example):



- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on "wide" registers
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
  - AXV512: you get it.
- Adding two registers holding double precision floating point operands



There is no single driving force for single core performance!



Maximum floating point (FP) performance:

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

Super-scalarity | FMA factor | SIMD factor | Clock Speed

Typical representatives	n_{super}^{FP} [inst./cy]	n_{FMA}	n_{SIMD} [ops/inst.]		Code	f [Gcy/s]	P_{core} [GF/s]
Nehalem	2	1	2	Q1/2009	X5570	2.93	11.7
Westmere	2	1	2	Q1/2010	X5650	2.66	10.6
Sandy Bridge	2	1	4	Q1/2012	E5-2680	2.7	21.6
Ivy Bridge	2	1	4	Q3/2013	E5-2660 v2	2.2	17.6
Haswell	2	2	4	Q3/2014	E5-2695 v3	2.3	36.8
Broadwell	2	2	4	Q1/2016	E5-2699 v4	2.2	35.2
Skylake	2	2	8	Q3/2017	Gold 6148	2.4	76.8
AMD Zen	2	2	2	Q1/2017	Epyc 7451	2.3	18.4
IBM POWER8	2	2	2	Q2/2014	S822LC	2.93	23.4

A “simple” example: The sum reduction

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

…In single precision 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

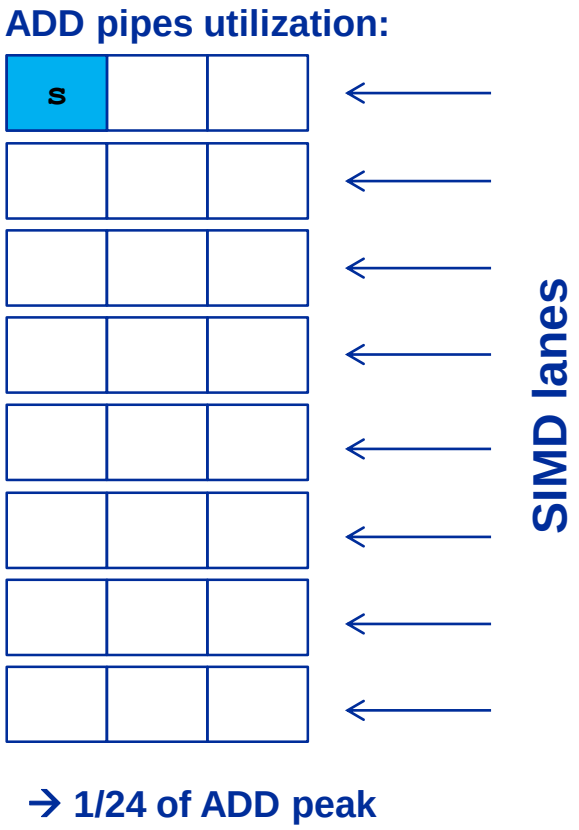
→ No pipelining?
→ No SIMD?

Applicable peak for the sum reduction (I)

Plain scalar code, no SIMD

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

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



Applicable peak for the sum reduction (II)

Scalar code, 3-way "modulo variable expansion"

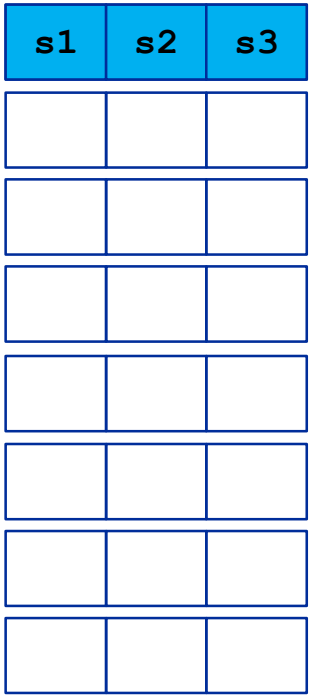
```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1

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

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

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

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



→ 1/8 of ADD peak

Applicable peak for the sum reduction (III)

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

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

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

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

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

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

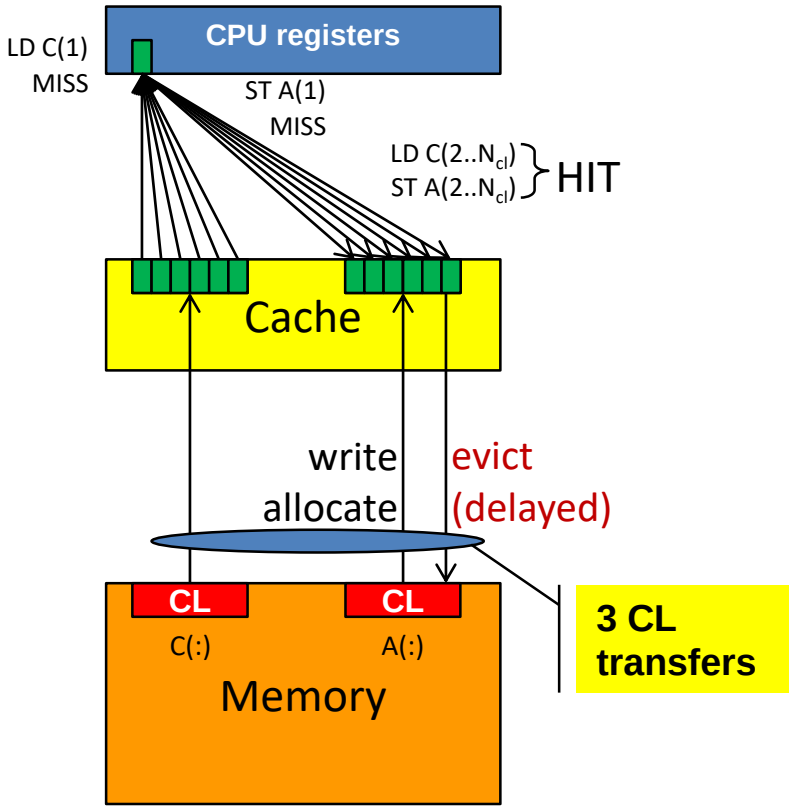
→ ADD peak

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

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

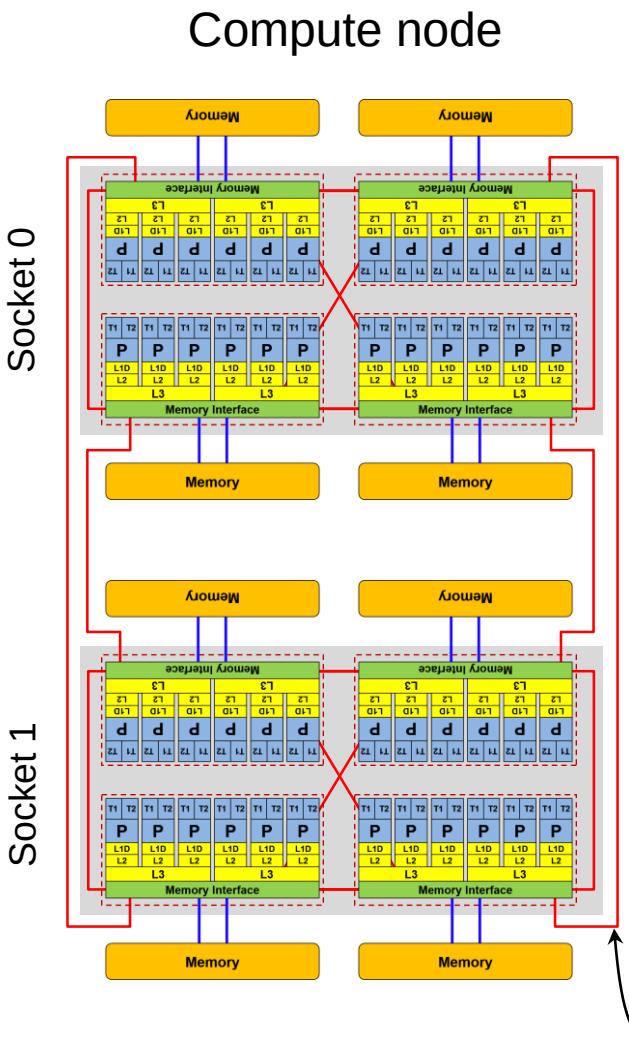
- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level
  → CL transfer required

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

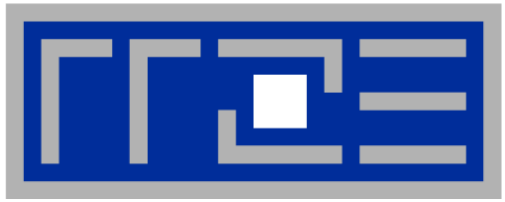


Putting the cores & caches together

AMD Epyc 7451 24-Core Processor («Naples»)



- 24 cores per socket
  - 4 chips w/ 6 cores each (“Zeppelin” die)
    - 3 cores share 8MB L3 (“Core Complex”, “CCX”)
- DDR4-2666 memory interface with 2 channels per chip
  - MemBW per node:
    16 ch x 8 byte x 2.666 GHz = 341 GB/s
- Two-way SMT
- Two 256-bit (actually 4 128-bit) SIMD FP units
  - AVX2, 8 flops/cycle
- 32 KiB L1 data cache per core
- 512 KiB L2 cache per core
- 2 x 8 MiB L3 cache per chip
  - 64 MiB L3 cache per socket
- ccNUMA memory architecture
- Infinity fabric between CCX’s and between chips



Interlude:
A glance at current accelerator technology

Nvidia “Volta” V100

vs.

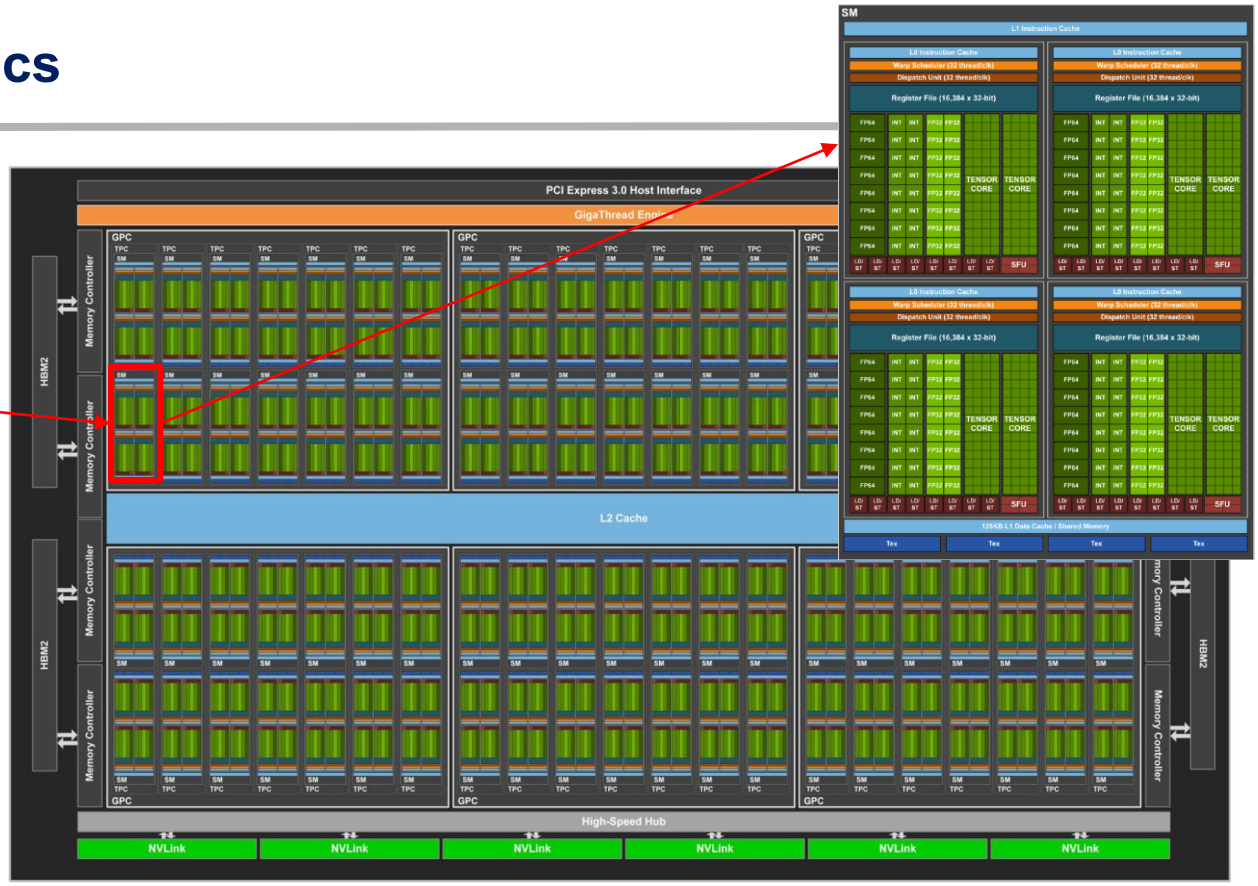
Intel Skylake “Platinum”

Nvidia V100 “Volta” specs

Architecture

- 21.1 B Transistors
- ~ 1.4 GHz clock speed
- ~ 80 “SM” units
  - 64 SP “cores” each (FMA)
  - 32 DP “cores” each (FMA)
  - 8 “Tensor Cores” each
  - 2:1 SP:DP performance

- ~7 TFlop/s DP peak
- 6 MiB L2 Cache
- 4096-bit HBM2
- MemBW ~ 900 GB/s (theoretical)
- MemBW ~ 830 GB/s (measured)



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

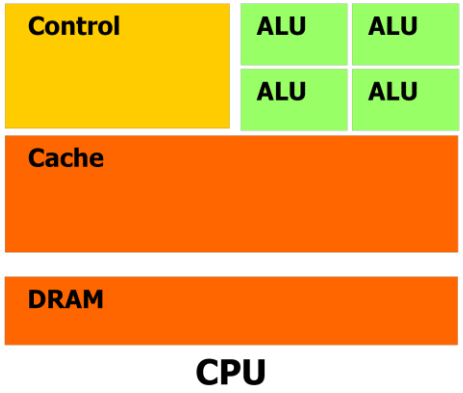
# SMs	# CUDA cores/SM	# FP ops/cy

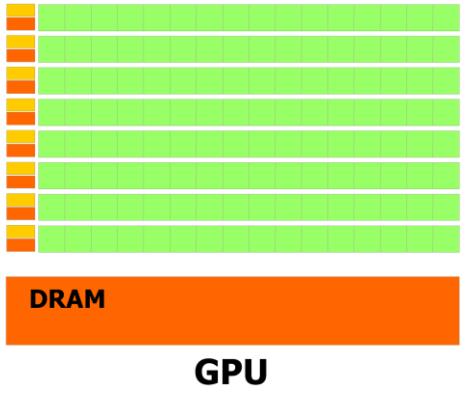
$$n_{SM} = 80$$
$$n_{core} = 32$$
$$n_{FP} = 2\frac{\text{flops}}{\text{cy}}$$
$$f = 1.4\frac{\text{Gcy}}{\text{s}}$$

Trading single thread performance for parallelism:
GPGPUs vs. CPUs

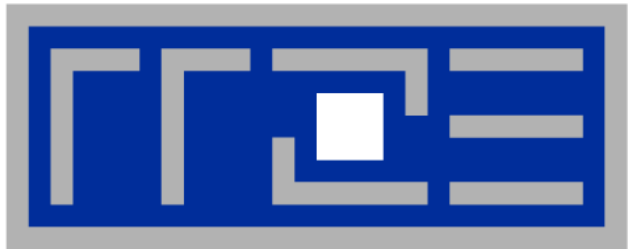
GPU vs. CPU
light speed estimate
(per device)

MemBW ~ 8-10x
Peak ~ 5-10x





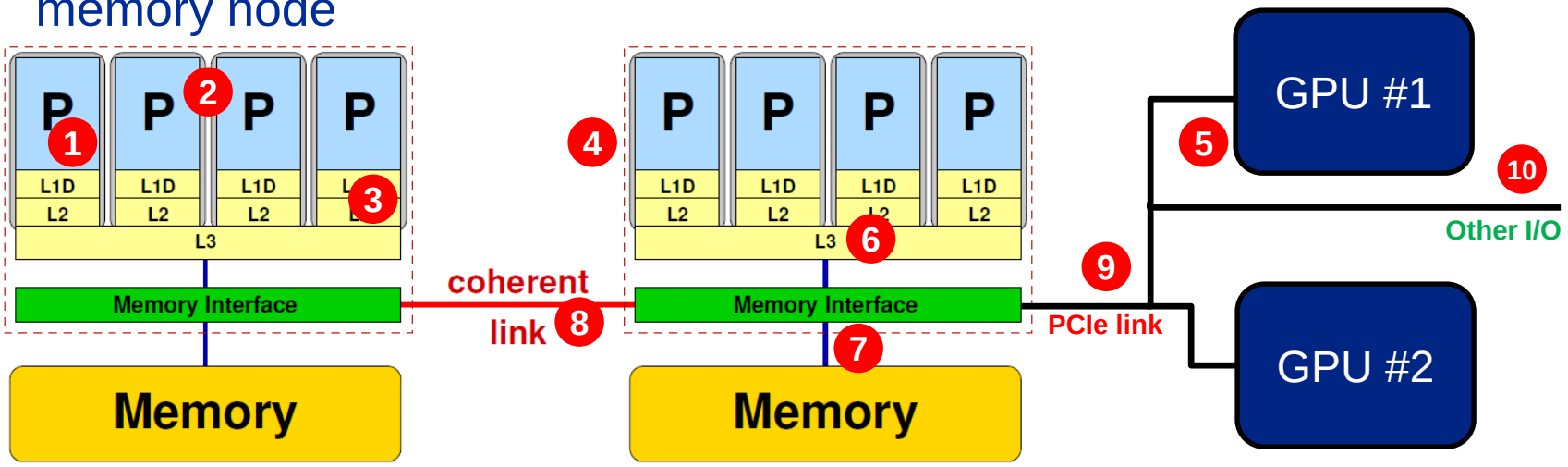
	2x Intel Xeon Platinum 8160	NVidia Tesla V100 “Volta”
Cores@Clock	2 x 24 @ ≥2.1 GHz	80 SMs @ ~1.4 GHz
SP Performance/core	≥ 134 GFlop/s	~179 GFlop/s
Threads@STREAM	~20	> 20000
SP peak	≥ 6 TFlop/s	~14 TFlop/s
Stream BW (meas.)	2 x 105 GB/s	830 GB/s
Transistors / TDP	~2x8 Billion / 2x150 W	21 Billion/250 W



Node topology and programming models

Parallelism in a modern compute node

- Parallel and shared resources (potential bottlenecks!) within a shared-memory node



Parallel resources:

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

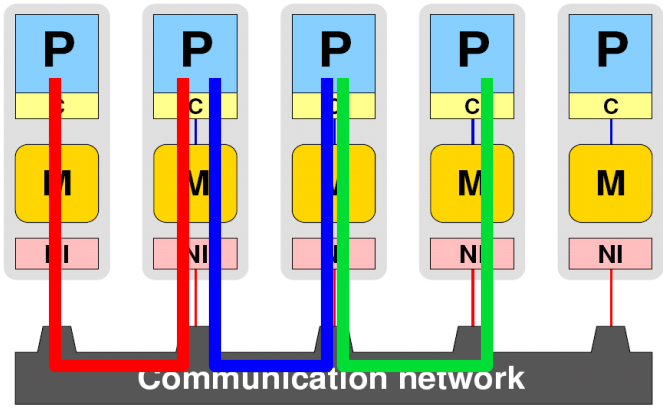
Shared resources:

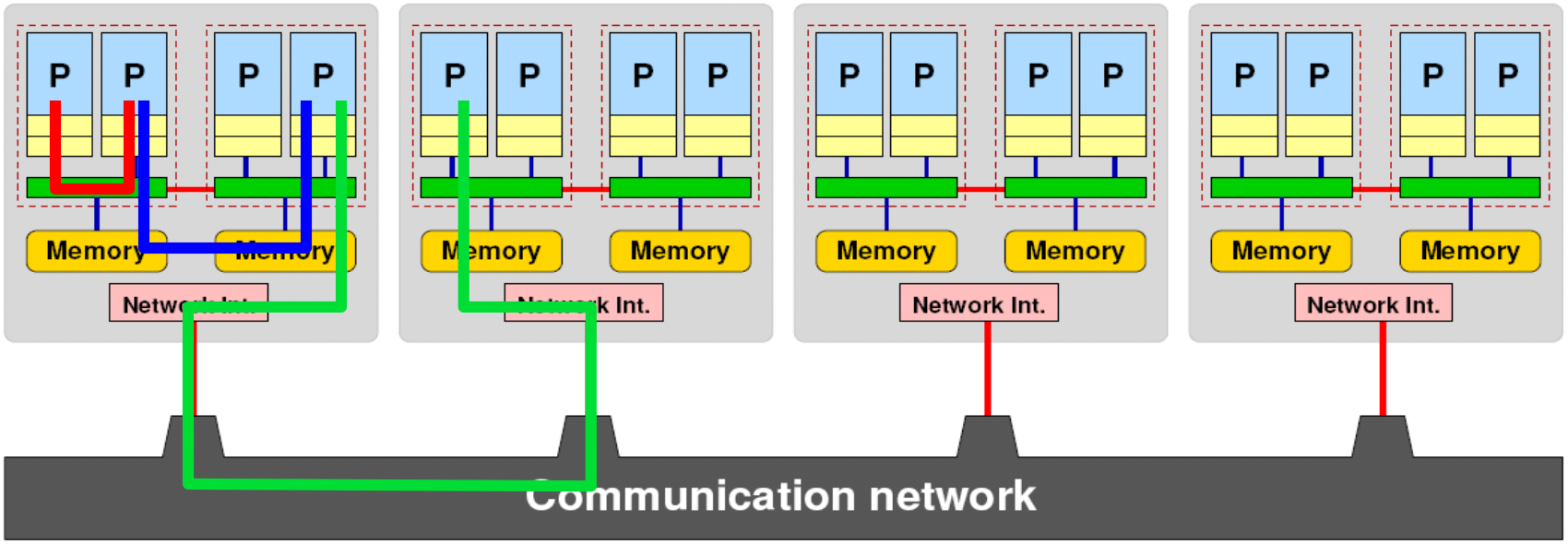
- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

How does your application react to all of those details?

Parallel programming models:
Pure MPI

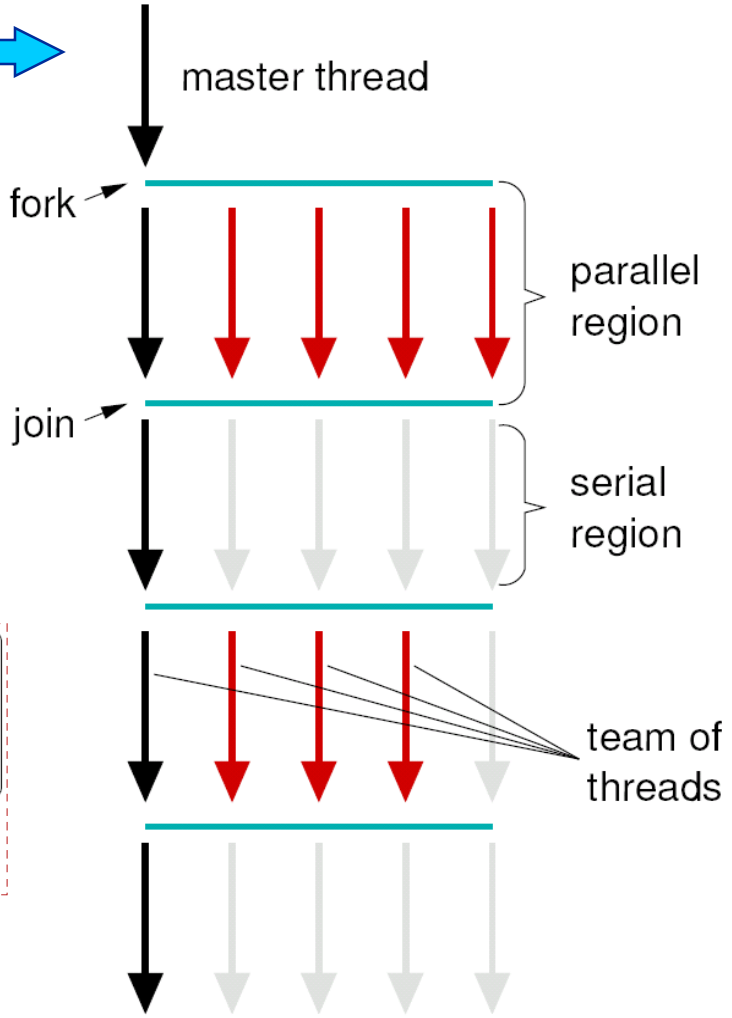
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

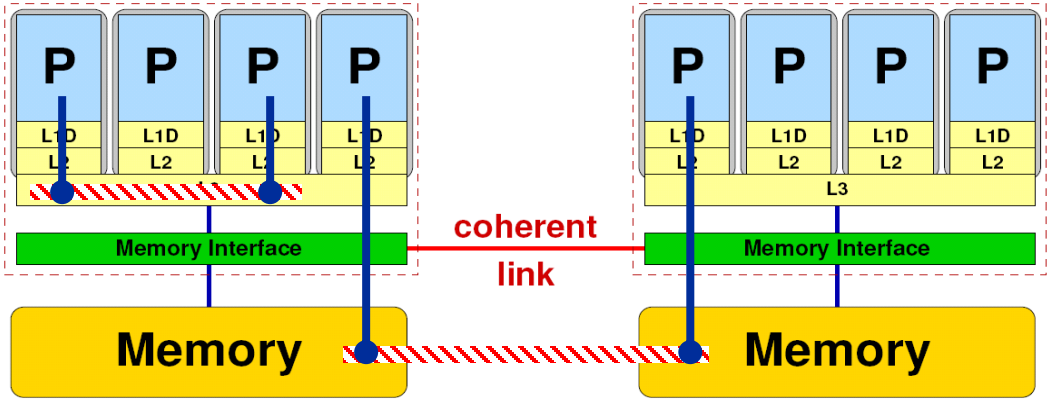




Parallel programming models:
Pure threading on the node

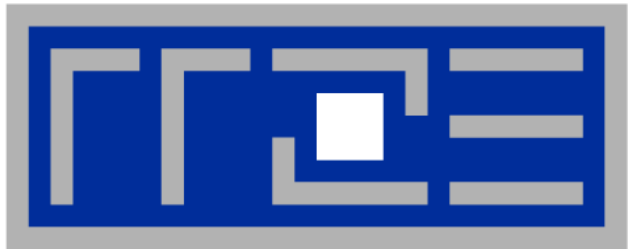
- **Machine structure is invisible to user**
  - → Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,…) should know about the details
  - Some support since OpenMP 4.0
- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology





Conclusions about architecture

- **Modern computer architecture has a rich "topology"**
- **Node-level hardware parallelism takes many forms**
  - Sockets/devices – CPU: 1-8, GPGPU: 1-6
  - Cores – moderate (CPU: 4-64) to massive (GPGPU: 10's-100's)
  - SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10's-100's)
  - Superscalarity (CPU: 2-6)
- **Exploiting performance: parallelism + bottleneck awareness**
  - **"High Performance Computing" == computing at a bottleneck**
- **Performance of programming models is sensitive to architecture**
  - Topology/affinity influences overheads
  - Standards do not contain (many) topology-aware features
    - Slowly improving, though (OpenMP 4.0, MPI 3.0)
  - Apart from overheads, performance features are largely independent of the programming model



Multicore Performance and Tools

Part 1: Topology and affinity

Tools for Node-level Performance Engineering

- Gather **Node Information**
  *hwloc, **likwid-topology**, likwid-powermeter*

- **Affinity control** and data placement
  *OpenMP and MPI runtime environments, hwloc, numactl, **likwid-pin***

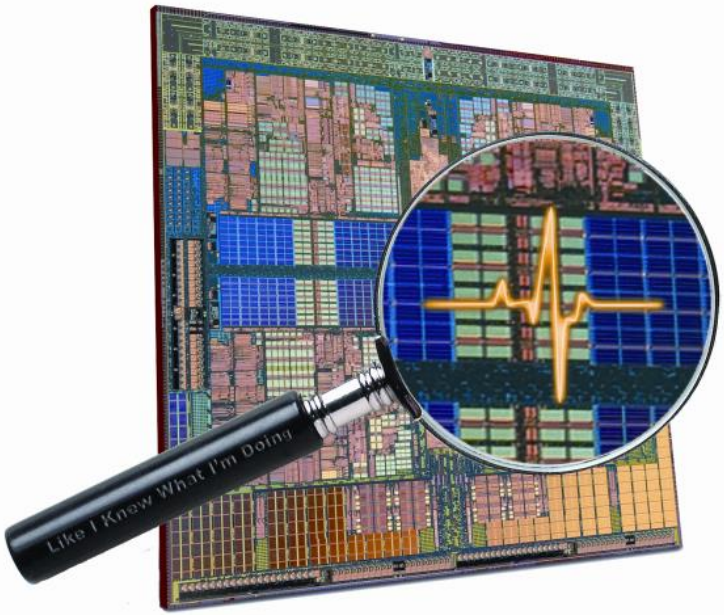
- **Runtime Profiling**
  Compilers, gprof, HPC Toolkit, Intel Amplifier,…

- **Performance Profilers**
  *Intel Vtune™, **likwid-perfctr**, PAPI based tools, HPC Toolkit, Linux perf, …*

- **Microbenchmarking**
  *STREAM, **likwid-bench**, lmbench*

LIKWID tool suite:

Like
I
Knew
What
I'm
Doing



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 http://arxiv.org/abs/1004.4431

Output of `likwid-topology -g`

on one node of Intel Haswell-EP

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU stepping:   2
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                        2
Cores per socket:               14
Threads per core:               2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1                 0             1               0               *

...
43               1              1               1               *
44               1              2               1               *
--------------------------------------------------------------------------------
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 )
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 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                          1
Size:                           32 kB
Cache groups:   ( 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
) ( 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 )
--------------------------------------------------------------------------------
Level:                          2
Size:                           256 kB
Cache groups:   ( 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
) ( 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 )
--------------------------------------------------------------------------------
Level:                          3
Size:                           17 MB
Cache groups:   ( 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 ) ( 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 )
--------------------------------------------------------------------------------
```

All physical processor IDs

Output of likwid-topology continued

```
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:                   4
--------------------------------------------------------------------------------
Domain:                         0
Processors:             ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 )
Distances:                      10 21 31 31
Free memory:                    13292.9 MB
Total memory:                   15941.7 MB
--------------------------------------------------------------------------------
Domain:                         1
Processors:             ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Distances:                      21 10 31 31
Free memory:                    13514 MB
Total memory:                   16126.4 MB
--------------------------------------------------------------------------------
Domain:                         2
Processors:             ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 )
Distances:                      31 31 10 21
Free memory:                    15025.6 MB
Total memory:                   16126.4 MB
--------------------------------------------------------------------------------
Domain:                         3
Processors:             ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
Distances:                      31 31 21 10
Free memory:                    15488.9 MB
Total memory:                   16126 MB
--------------------------------------------------------------------------------
```

Output of likwid-topology continued

Cluster on die mode and SMT enabled!

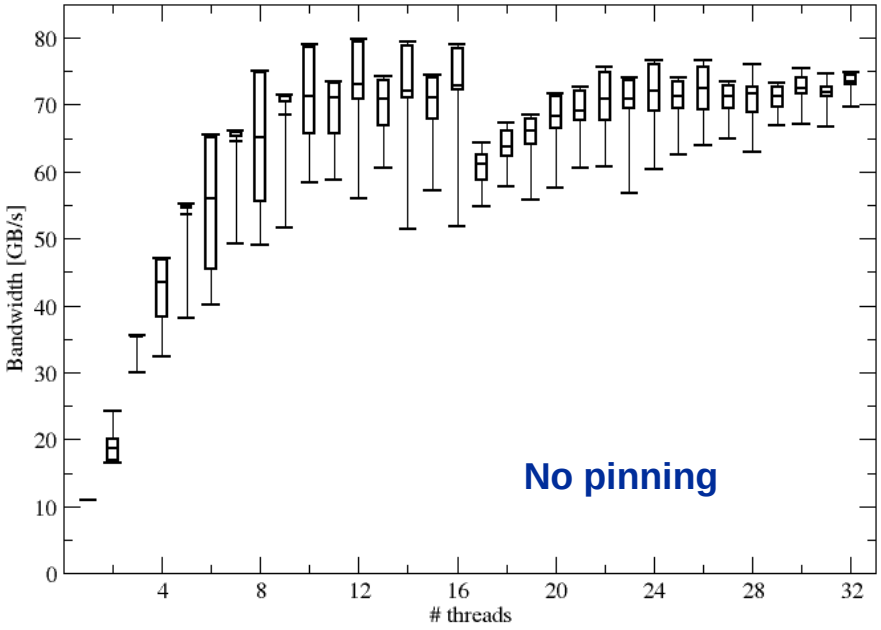
```
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  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  | | 14 42  | | 15 43  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  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 | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------------+ |
| |                                        17MB                                         | |                                        17MB                                         | |
| +-------------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
Socket 1:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| | 16 44  | | 17 45  | | 18 46  | | 19 47  | | 20 48  | | 21 49  | | 22 50  | | 23 51  | | 24 52  | | 25 53  | | 26 54  | | 27 55  | | 28 56  | | 29 57  | | 30 58  | | 31 59  | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| |  32kB  | |  32kB  | |  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 | |  256kB | |  256kB | |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ +--------+ |
| +-------------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------------+ |
| |                                        17MB                                         | |                                        17MB                                         | |
| +-------------------------------------------------------------------------------------+ +-------------------------------------------------------------------------------------+ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

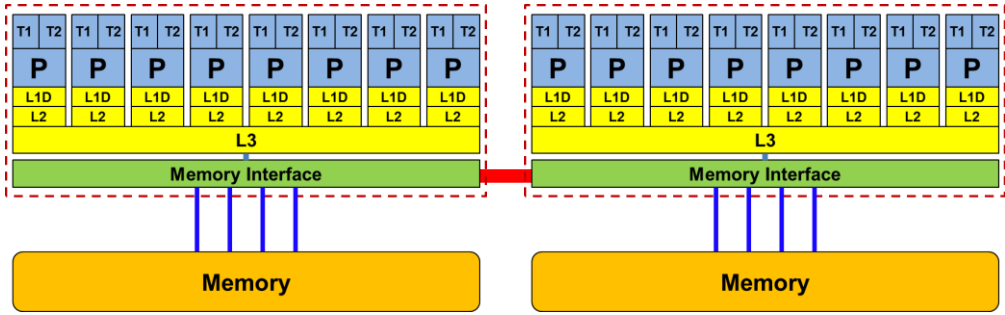
Enforcing thread/process-core affinity

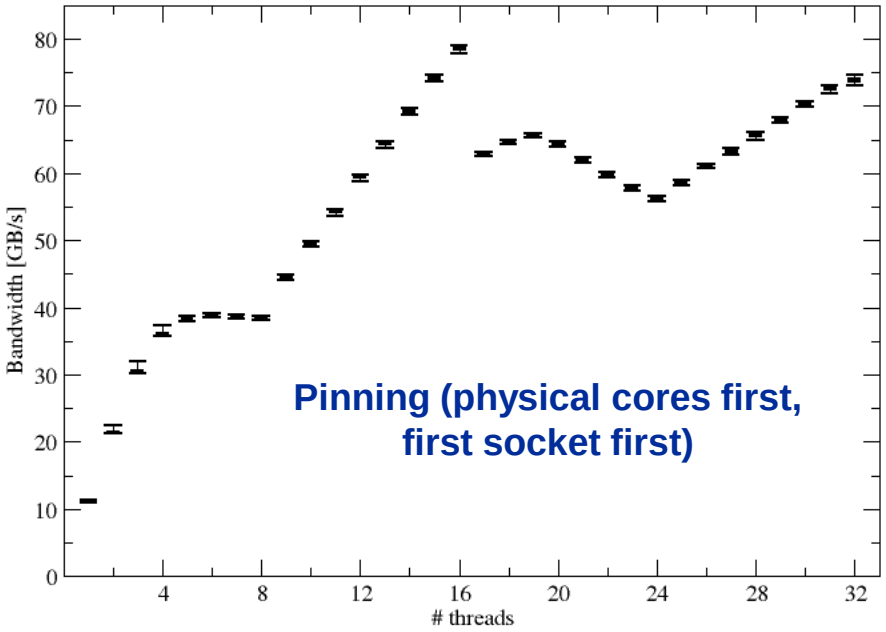
likwid-pin

OpenMP affinity mechanisms

Example: OpenMP STREAM benchmark on 16-core system:
Anarchy vs. thread pinning







There are several reasons for caring about affinity:

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

Likwid-pin

Overview

- 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!
- Can also be used as a superior replacement for taskset
- Supports logical core numbering within a node

- 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 parameters`

- Simple usage with logical core IDs (“thread groups”):
  - `likwid-pin -c S0:0-7  ./myApp parameters`
  - `likwid-pin -c C1:0-2 ./myApp parameters`

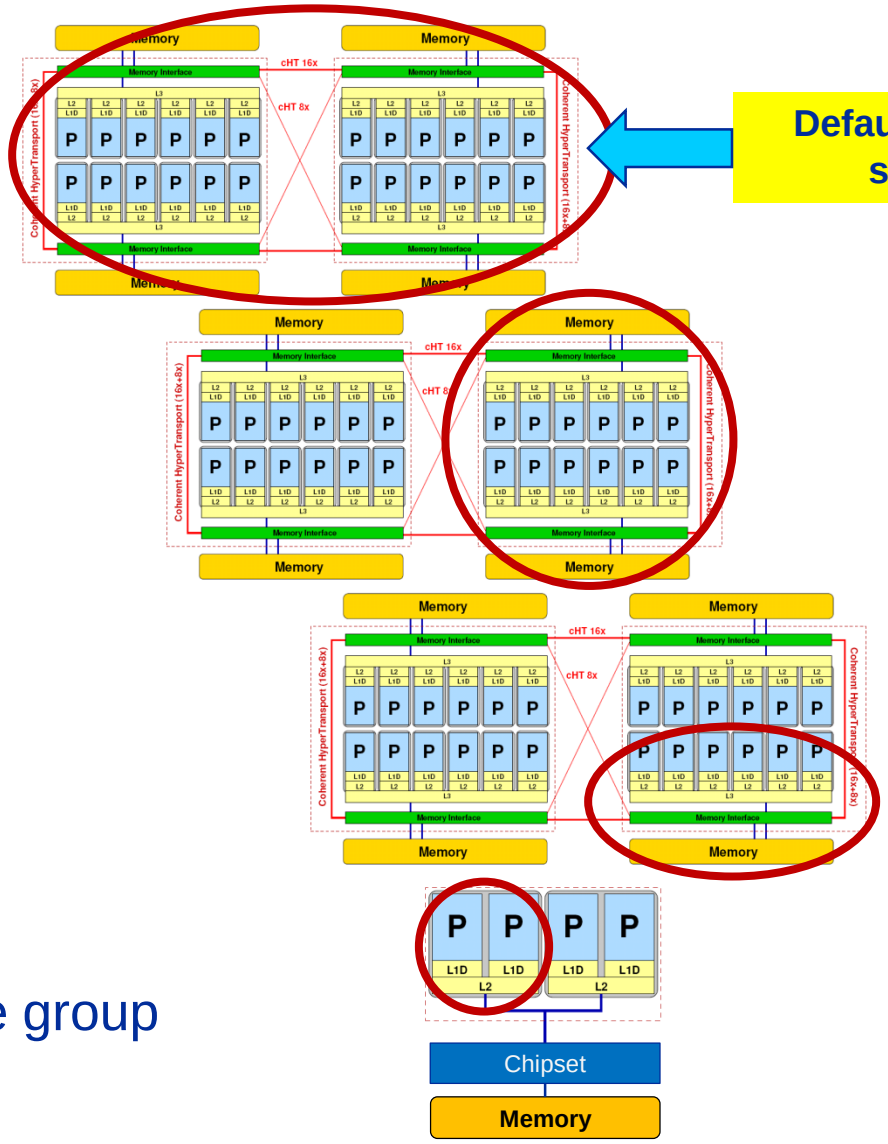
Possible unit prefixes

N node

S socket

M NUMA domain

C outer-level cache group



Default if `-c` is not specified!

- **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] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3
[pthread wrapper] SKIP MASK: 0x1
        threadid 140668624234240 -> SKIP
        threadid 140668598843264 -> core 1 - OK
        threadid 140668594644992 -> core 2 - OK
        threadid 140668590446720 -> core 3 - OK

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

Main PID always pinned

Skip shepherd thread if necessary

Pin all spawned threads in turn

OpenMP affinity: OMP_PLACES

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

- 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

- 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
```

likwid-mpirun

MPI startup and Hybrid pinning

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

Pure MPI:

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

Hybrid:

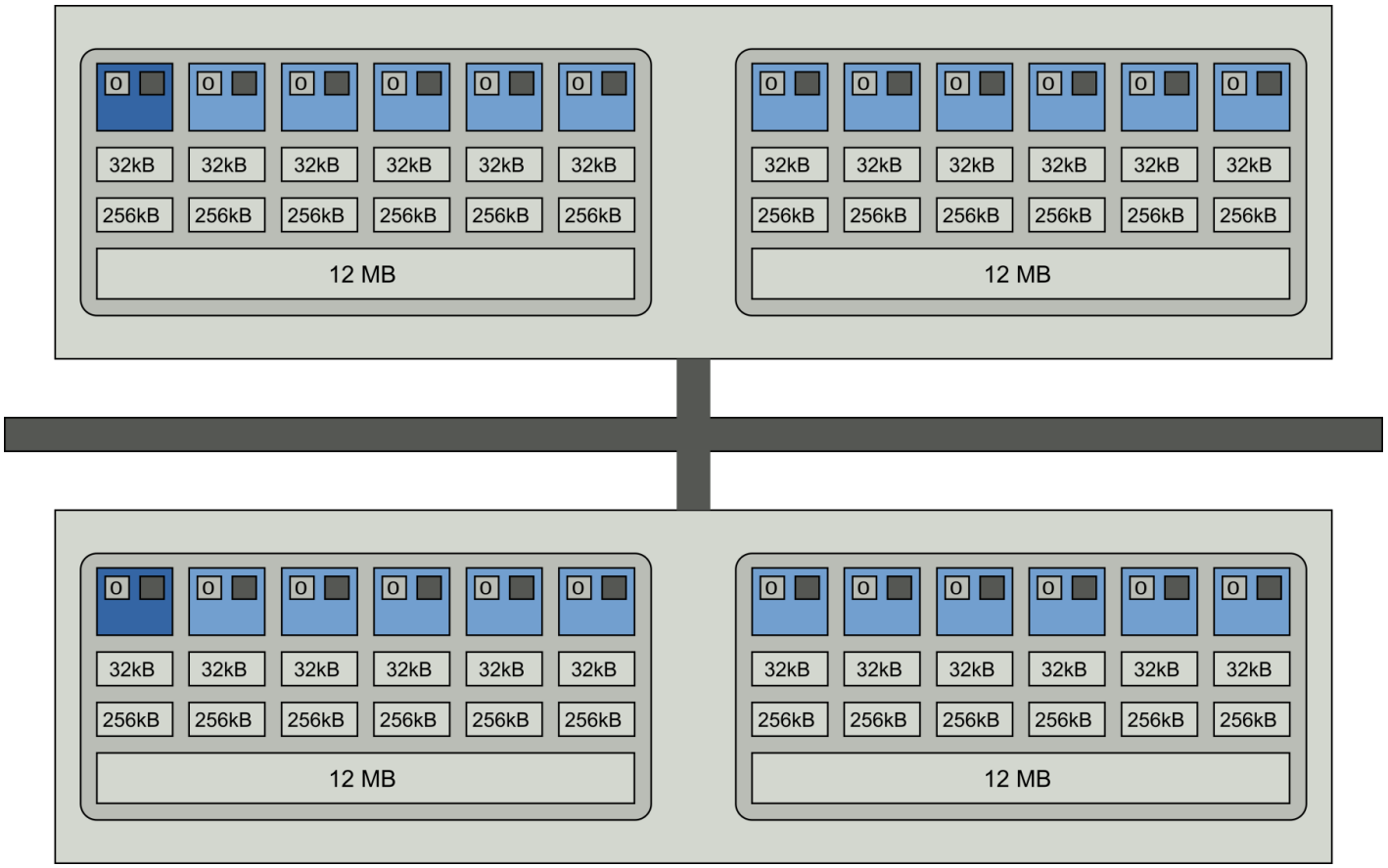
```
likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
```

likwid-mpirun

1 MPI process per node

optional

```
likwid-mpirun -np 2 -pin N:0-11   ./a.out
```



Intel MPI+compiler:

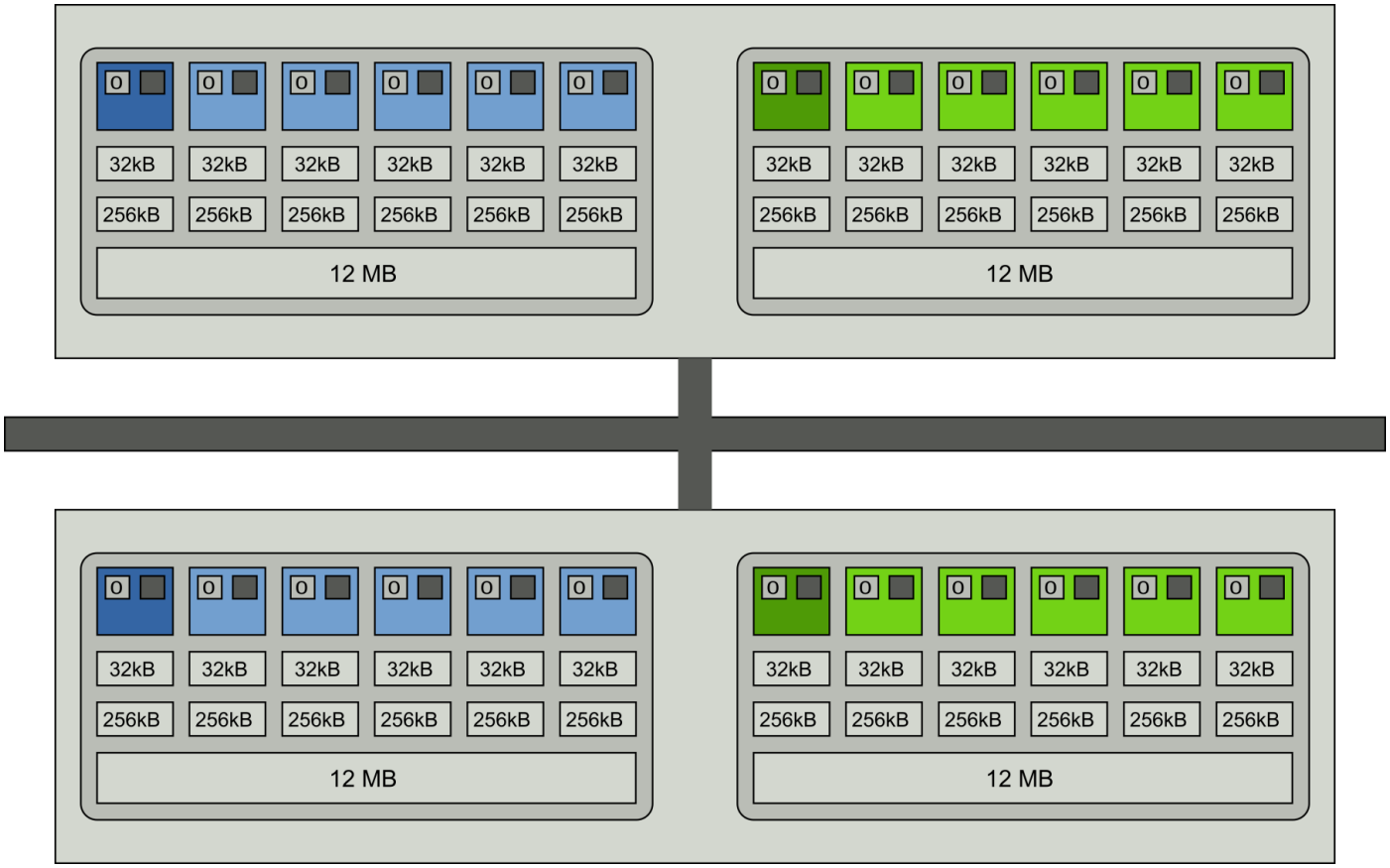
```
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out
```

likwid-mpirun

1 MPI process per socket

optional

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



Intel MPI+compiler:

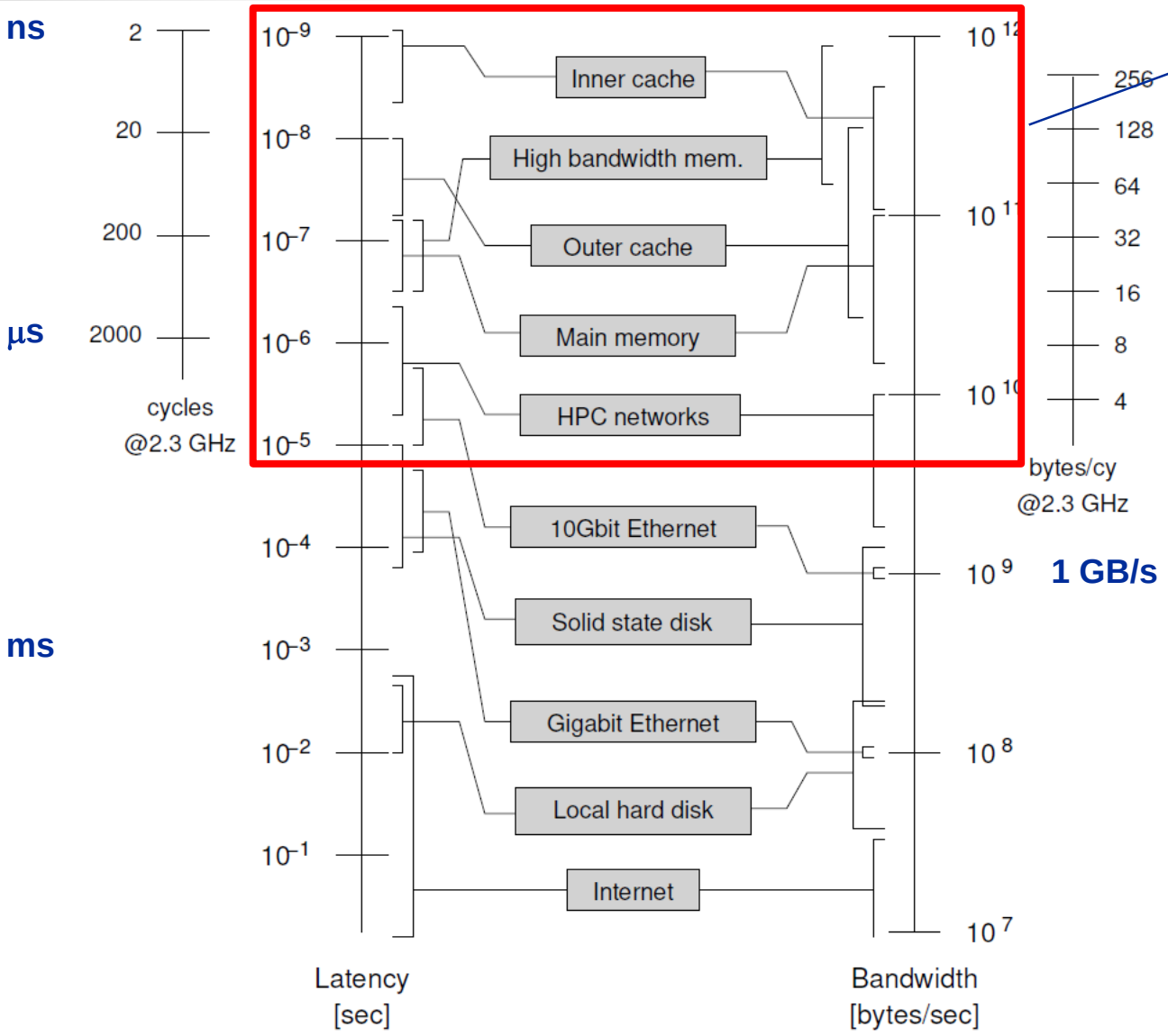
```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \
     -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

Microbenchmarking for architectural exploration (and more)

Probing of the memory hierarchy

Saturation effects in cache and memory

Latency and bandwidth in modern computer environments



HPC plays here

Avoiding slow data paths is the key to most performance optimizations!

But how "slow" are these data paths anyway?

The parallel vector triad benchmark

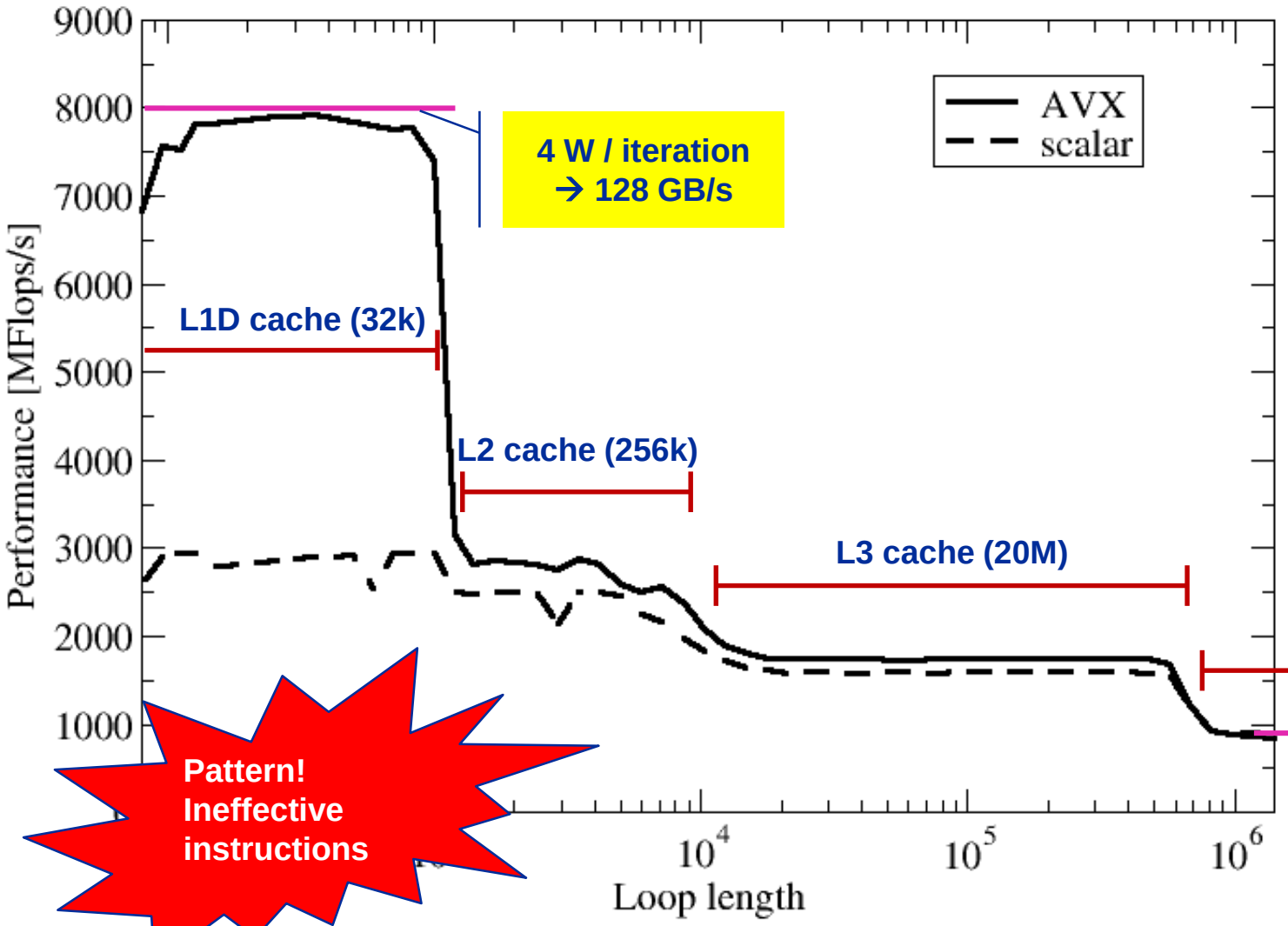
A "swiss army knife" for microbenchmarking

```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A
stime = timestamp()
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
etime = timestamp()
Mflops = (2.d0*NITER)*N / (etime-stime) / 1.0e6
```

Prevents smarty-pants compilers from doing "clever" stuff

- Report performance for different N, choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all architectures, ever!

A(:)=B(:)+C(:)*D(:) on one Sandy Bridge core (3 GHz)



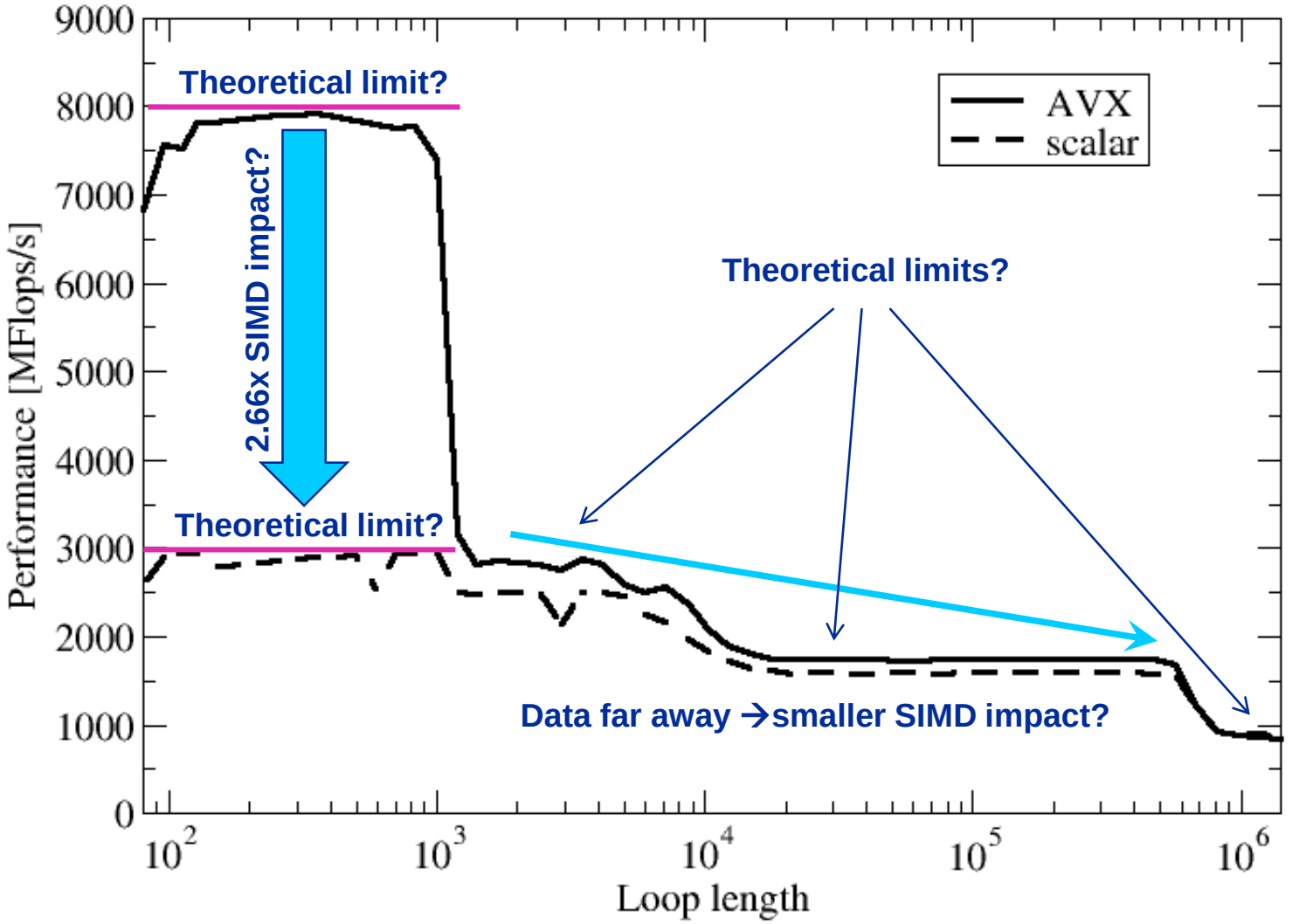
Are the performance levels plausible?

What about multiple cores?

Do the bandwidths scale?

`A(:)=B(:)+C(:)*D(:)` on one Sandy Bridge core (3 GHz):

Observations and further questions



See later for answers!

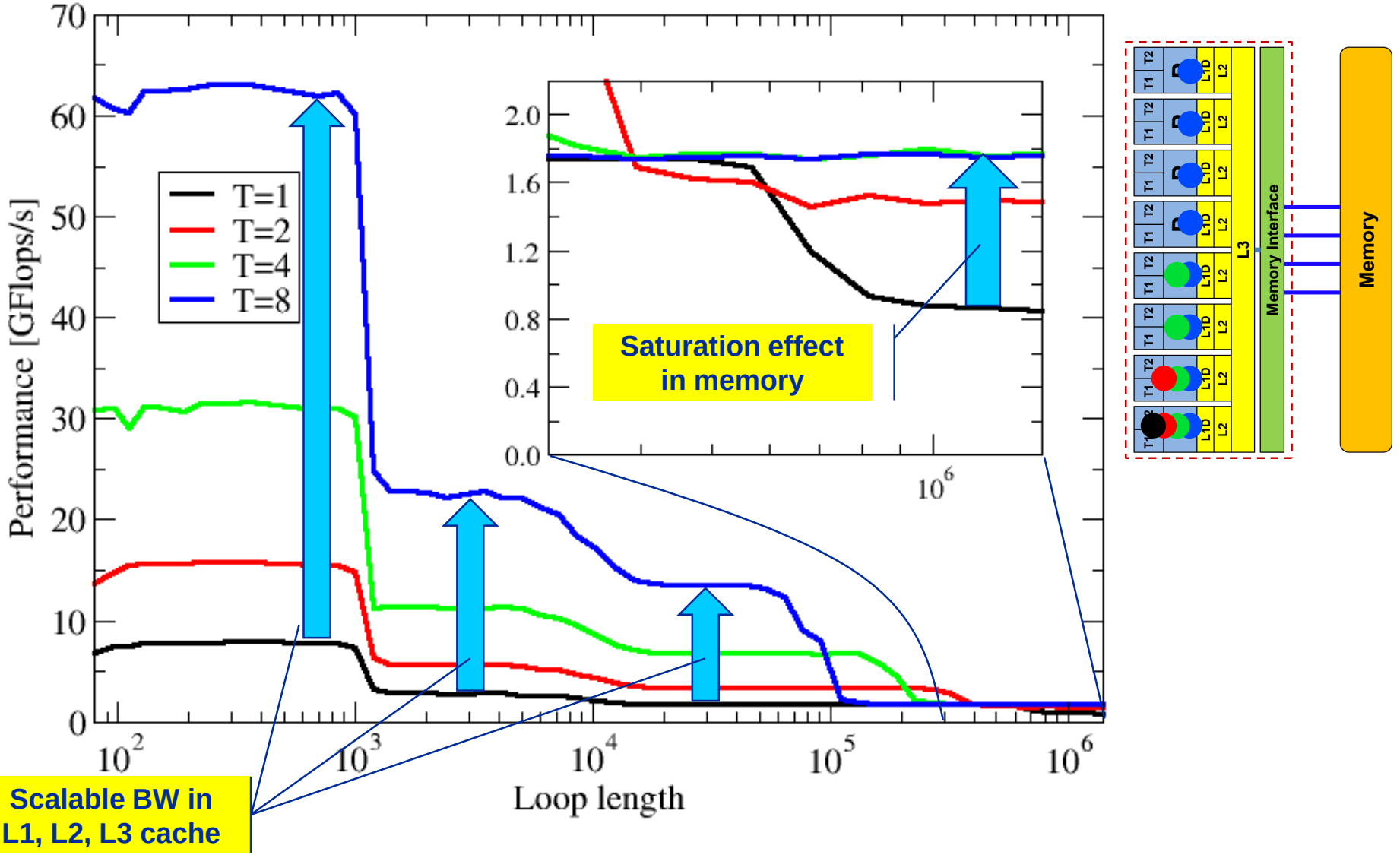
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

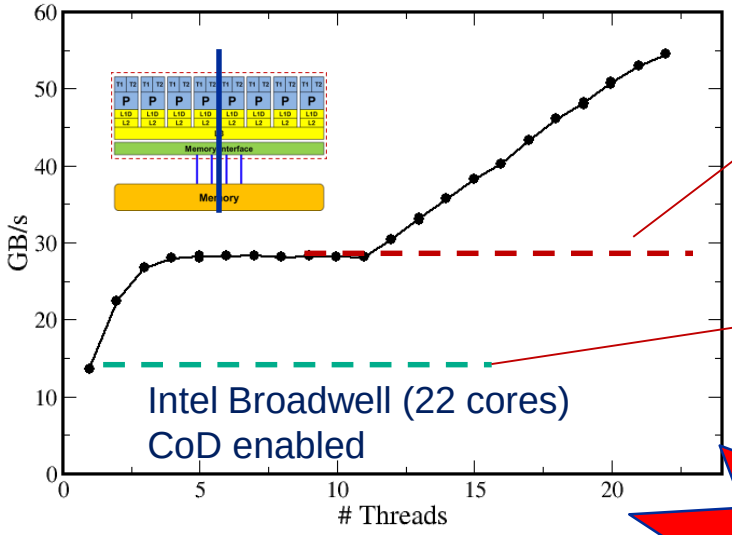
→ pure hardware probing, no impact from OpenMP overhead

```
double precision, dimension(:), allocatable :: A,B,C,D
!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
!$OMP SINGLE
stime = timestamp()
!$OMP END SINGLE
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  <<obscure dummy call>>
enddo
!$OMP SINGLE
etime = timestamp()
!$OMP END SINGLE
!$OMP END PARALLEL
Mflops = (2.d0*NITER)*N*num_threads / (etime-stime) / 1.0e6
```

Throughput vector triad on Sandy Bridge socket (3 GHz)

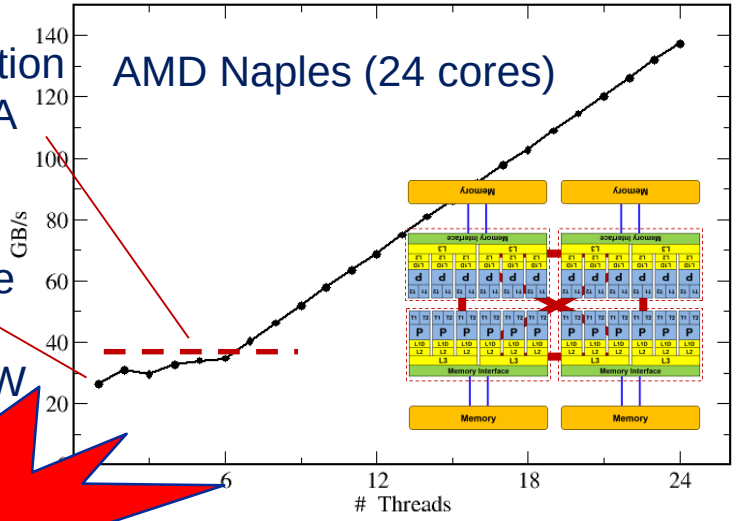


Attainable memory bandwidth: Comparing architectures

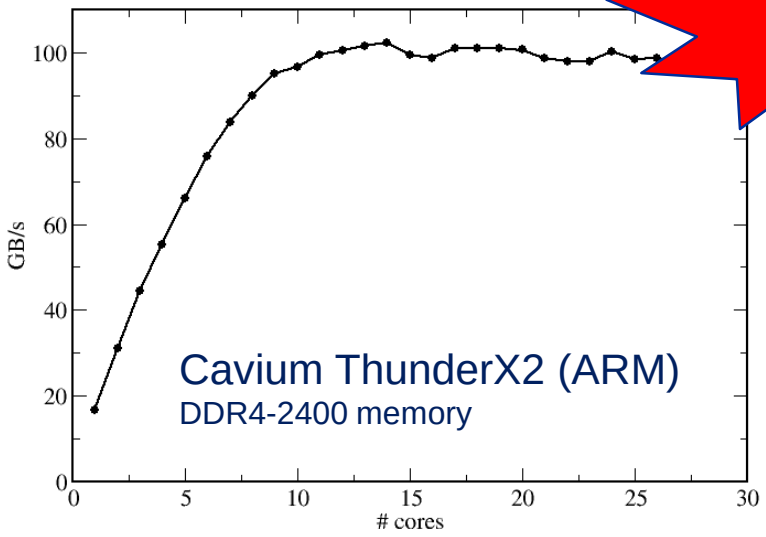


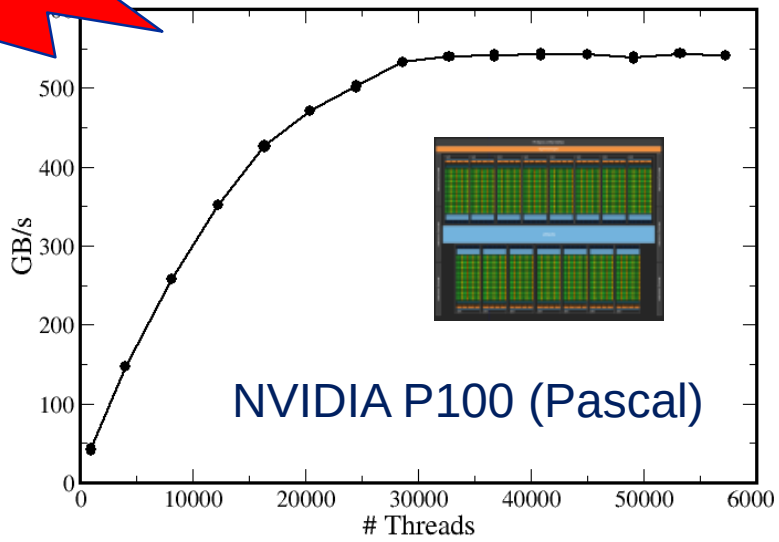
BW saturation in NUMA domain

Single core does not saturate BW



Pattern! Bandwidth saturation

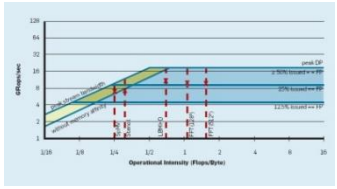




Conclusions from the microbenchmarks

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is

- **Bandwidth bottlenecks are ubiquitous**



“Simple” performance modeling: The Roofline Model

Loop-based performance modeling: Execution vs. data transfer
Example: array summation

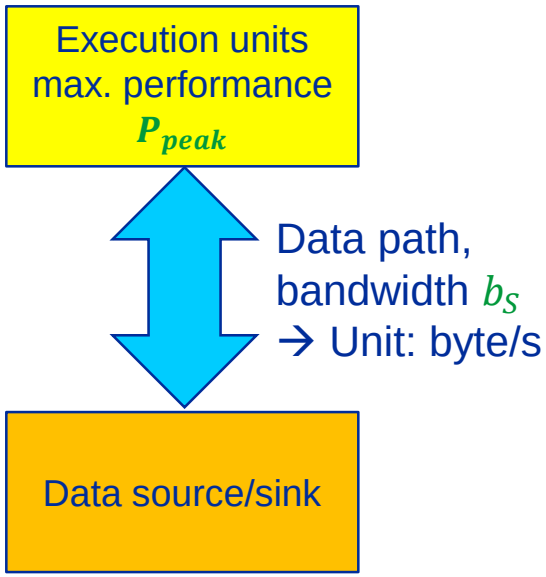
R.W. Hockney and I.J. Curington: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. Parallel Computing 10, 277-286 (1989). DOI: 10.1016/0167-8191(89)90100-2

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)

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

A simple performance model for loops

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

The bottleneck is either

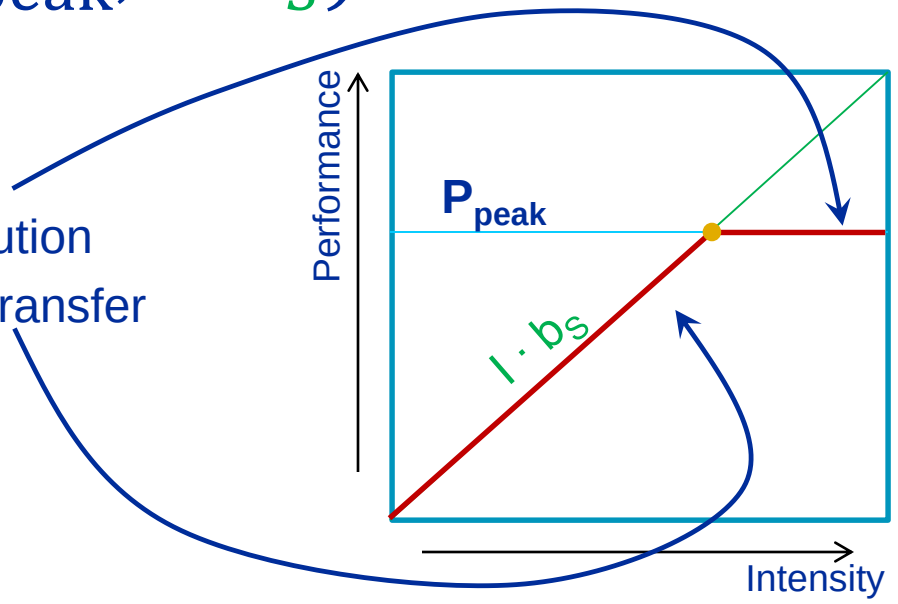
- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

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

This is the "Naïve Roofline Model"

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- "Knee" at $P_{max} = I \cdot b_S$:
  Best use of resources

- Roofline is an "optimistic" model ("light speed")



Apply the naive Roofline model in practice

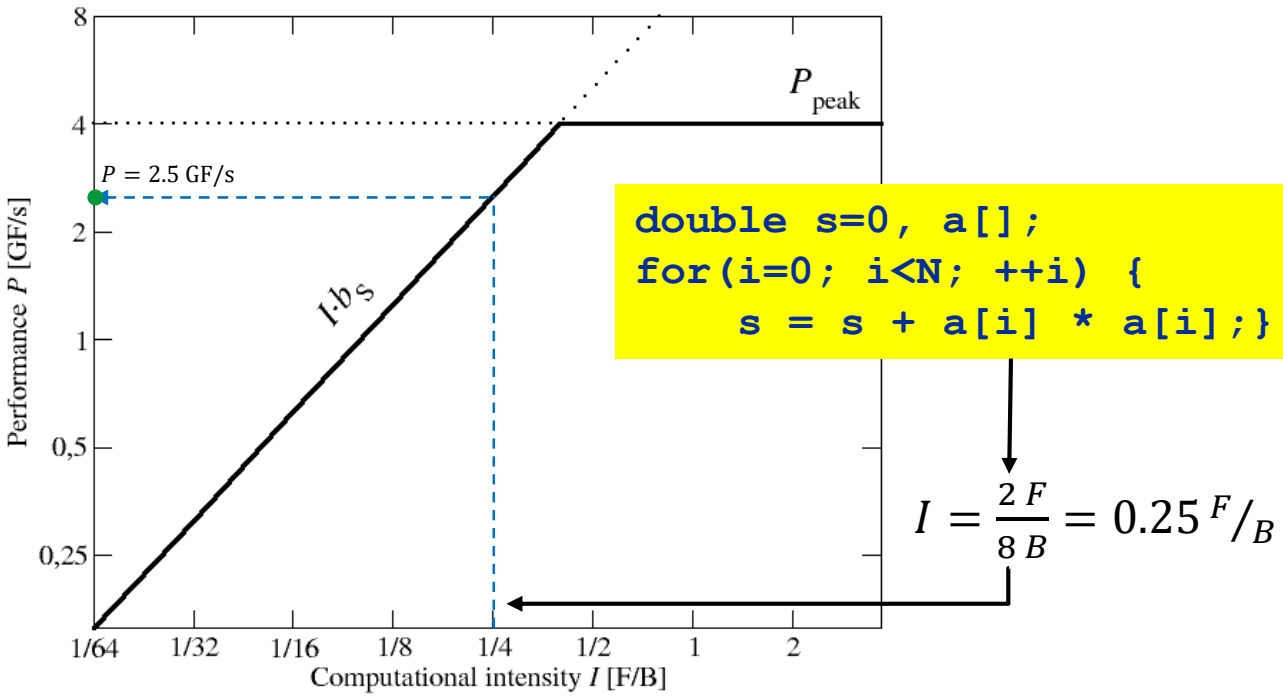
- Machine parameter #1: Peak performance: $P_{peak}\ \left[\frac{F}{s}\right]$
- Machine parameter #2: Memory bandwidth: $b_S\ \left[\frac{B}{s}\right]$
- Code characteristic: Computational Intensity: $I\ \left[\frac{F}{B}\right]$

Machine properties:

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

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

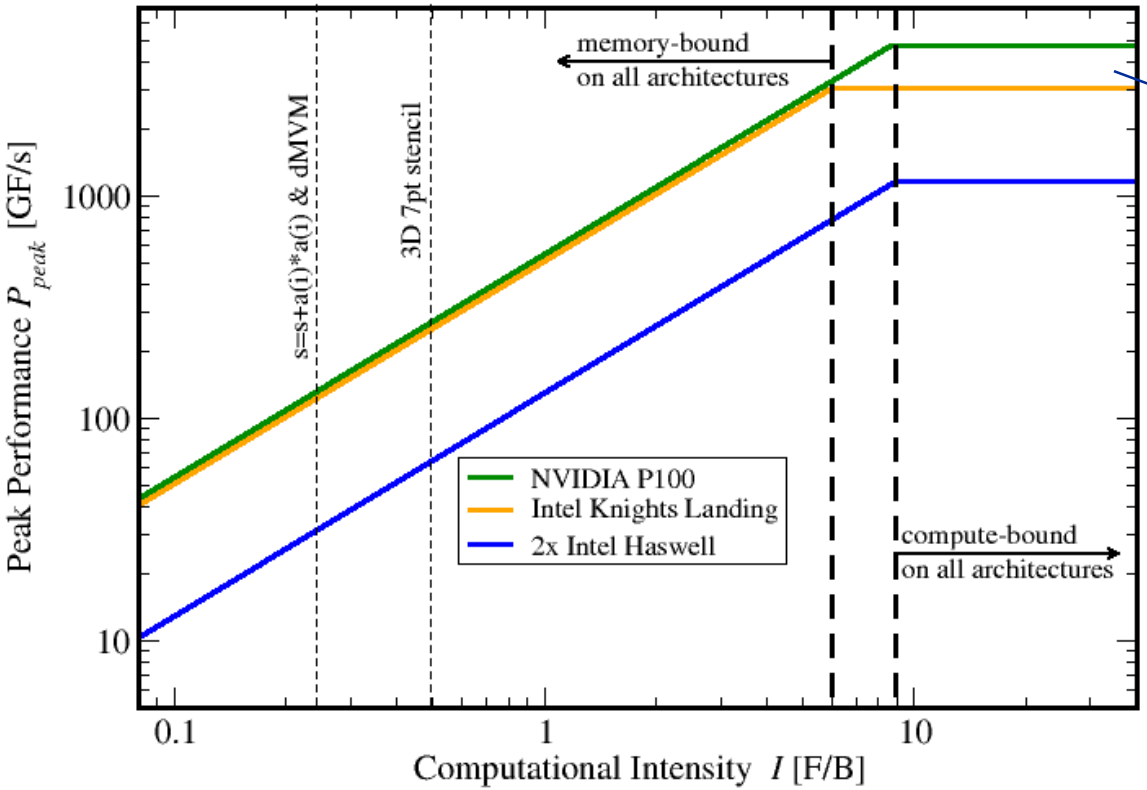
Application property: I



The Roofline Model – Basics

optional

Compare capabilities of different machines



Assuming full DP SIMD peak

- RLM always provides upper bound – but is it realistic?
- If code is not able to reach this limit (e.g. contains add operations only) machine parameter need to redefined (e.g., $P_{peak} \rightarrow P_{peak}/2$)

The Roofline Model
(a slightly refined version for better in-core prediction)

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

2. I = Computational intensity ("work" per byte transferred) over the slowest data path utilized (code balance $B_C = I^{-1}$)
   → e.g., I = 0.167 Flop/Byte → B_C = 6 Byte/Flop

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

Expected performance:

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

[Byte/s]

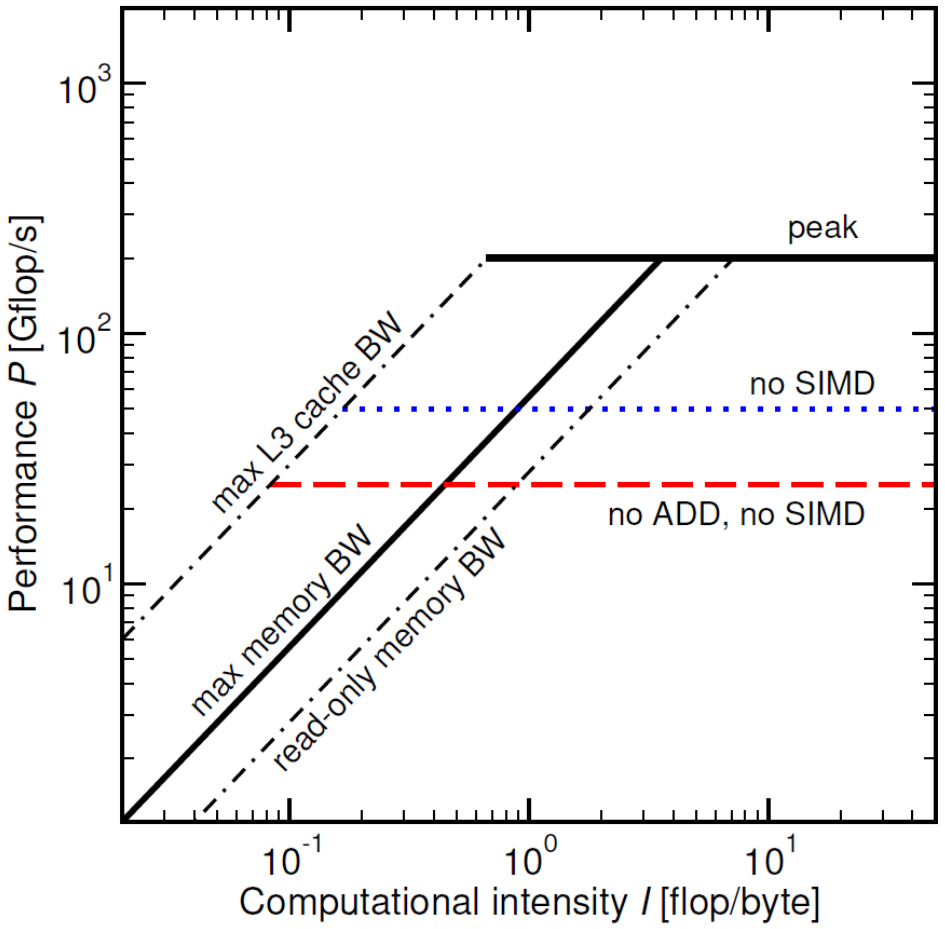
[Byte/Flop]

Refined Roofline model: graphical representation

optional

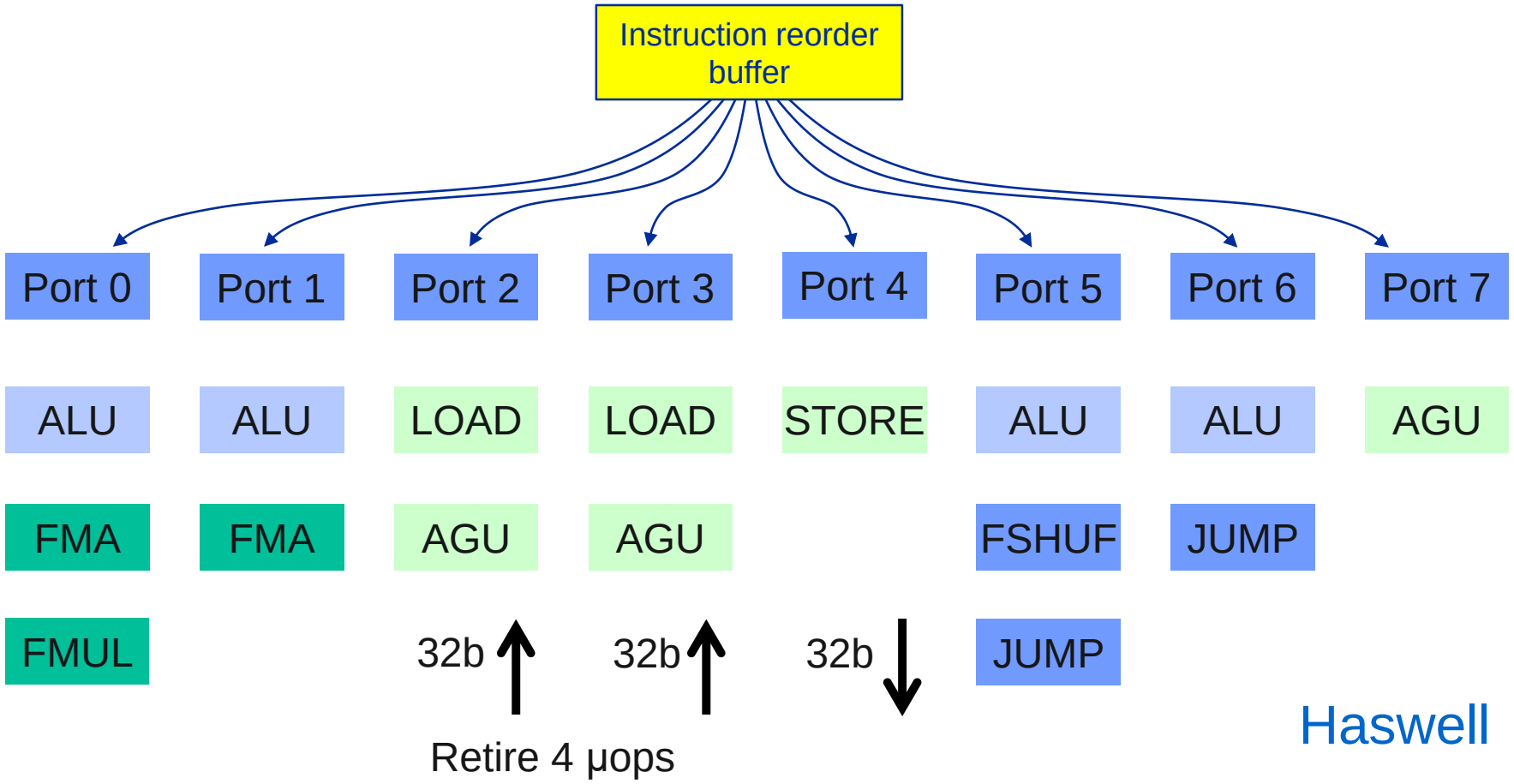
Multiple ceilings may apply

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



Estimating per-core P_{max} on a given architecture

Haswell port scheduler model:



Example: Estimate P_{max} of vector triad on Haswell

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

Minimum number of cycles to process one AVX-vectorized iteration (one core)?

→ Equivalent to 4 scalar iterations

Cycle 1: LOAD + LOAD + STORE
Cycle 2: LOAD + LOAD + FMA + FMA
Cycle 3: LOAD + LOAD + STORE

Answer: 1.5 cycles

Example: Estimate P_{max} of vector triad on Haswell (2.3 GHz)

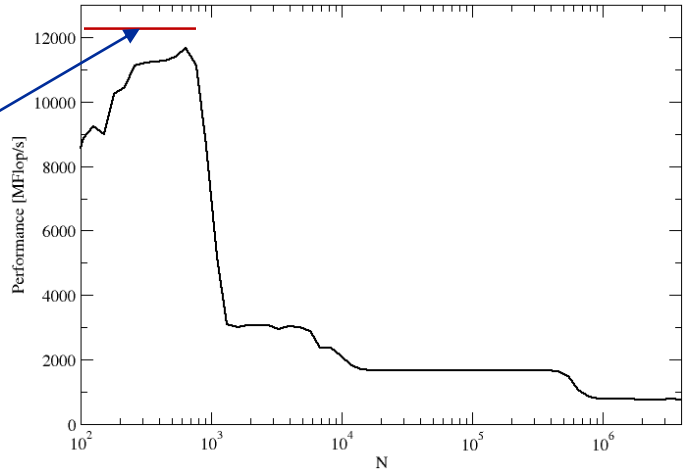
```
double  *A, *B, *C, *D;
for (int i=0; i<N; i++) {
   A[i] = B[i] + C[i] * D[i];
}
```

What is the performance in GFlops/s per core and the bandwidth in GBytes/s?

One AVX iteration (1.5 cycles) does 4 x 2 = 8 flops:

$$\frac{2.3 \cdot 10^9 \text{ cy/s}}{1.5 \text{ cy}} \cdot 4 \text{ updates} \cdot \frac{2 \text{ flops}}{\text{update}} = \mathbf{12.27} \frac{\mathbf{Gflops}}{\mathbf{s}}$$

$$6.13 \cdot 10^9 \frac{\text{updates}}{\text{s}} \cdot 32 \frac{\text{bytes}}{\text{update}} = 196 \frac{\text{Gbyte}}{\text{s}}$$



P_{max} + bandwidth limitations: The vector triad

Vector triad `A(:)=B(:)+C(:)*D(:)` on a 2.3 GHz 14-core Haswell chip

Consider full chip (14 cores):

Memory bandwidth: b_S = **50 GB/s**

Code balance (incl. write allocate):
B_c = (4+1) Words / 2 Flops = 20 B/F → ***I* = 0.05 F/B**

→ $I \cdot b_S$ **= 2.5 GF/s** (0.5% of peak performance)

P_{peak} / core = 36.8 Gflop/s ((8+8) Flops/cy x 2.3 GHz)
P_{max} / core = 12.27 Gflop/s (see prev. slide)

→ P_{max} **= 14 * 12.27 Gflop/s =172 Gflop/s** (33% peak)

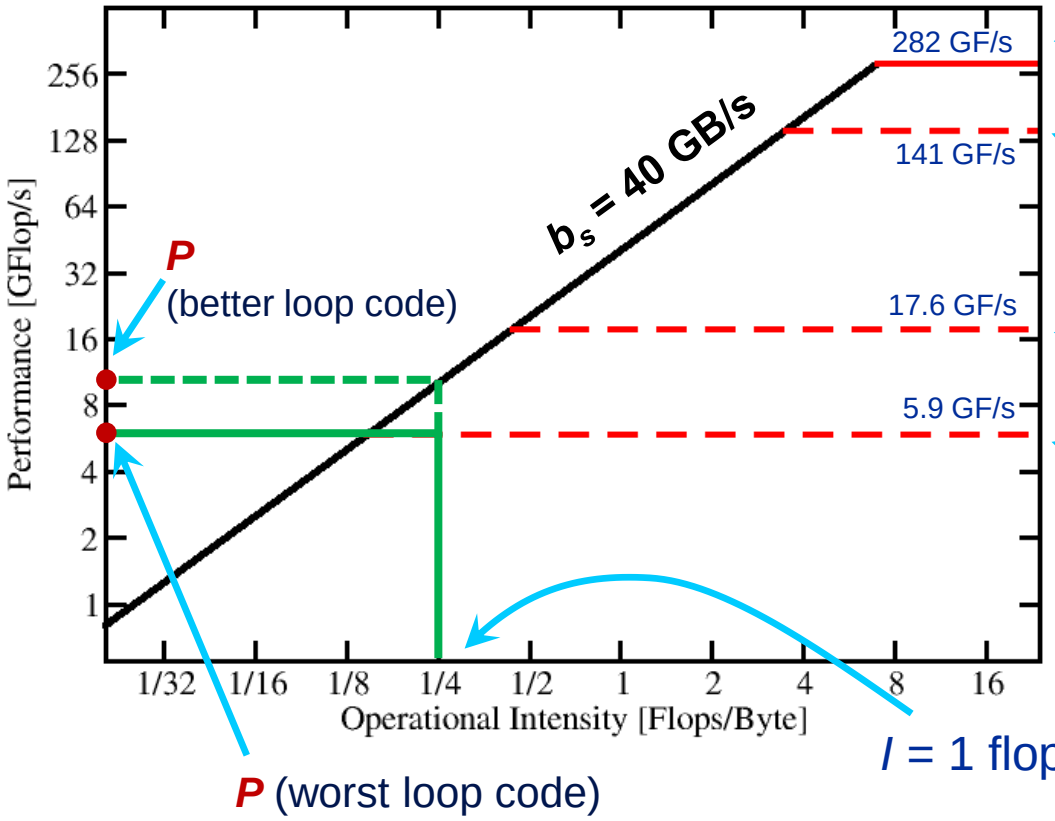
$$P = \min(P_{\max}, I \cdot b_S) = \min(172, 2.5)\ \mathrm{GFlop/s} = 2.5\ \mathrm{GFlop/s}$$

A not so simple Roofline example

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

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

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



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

ADD peak (best possible code)

no SIMD

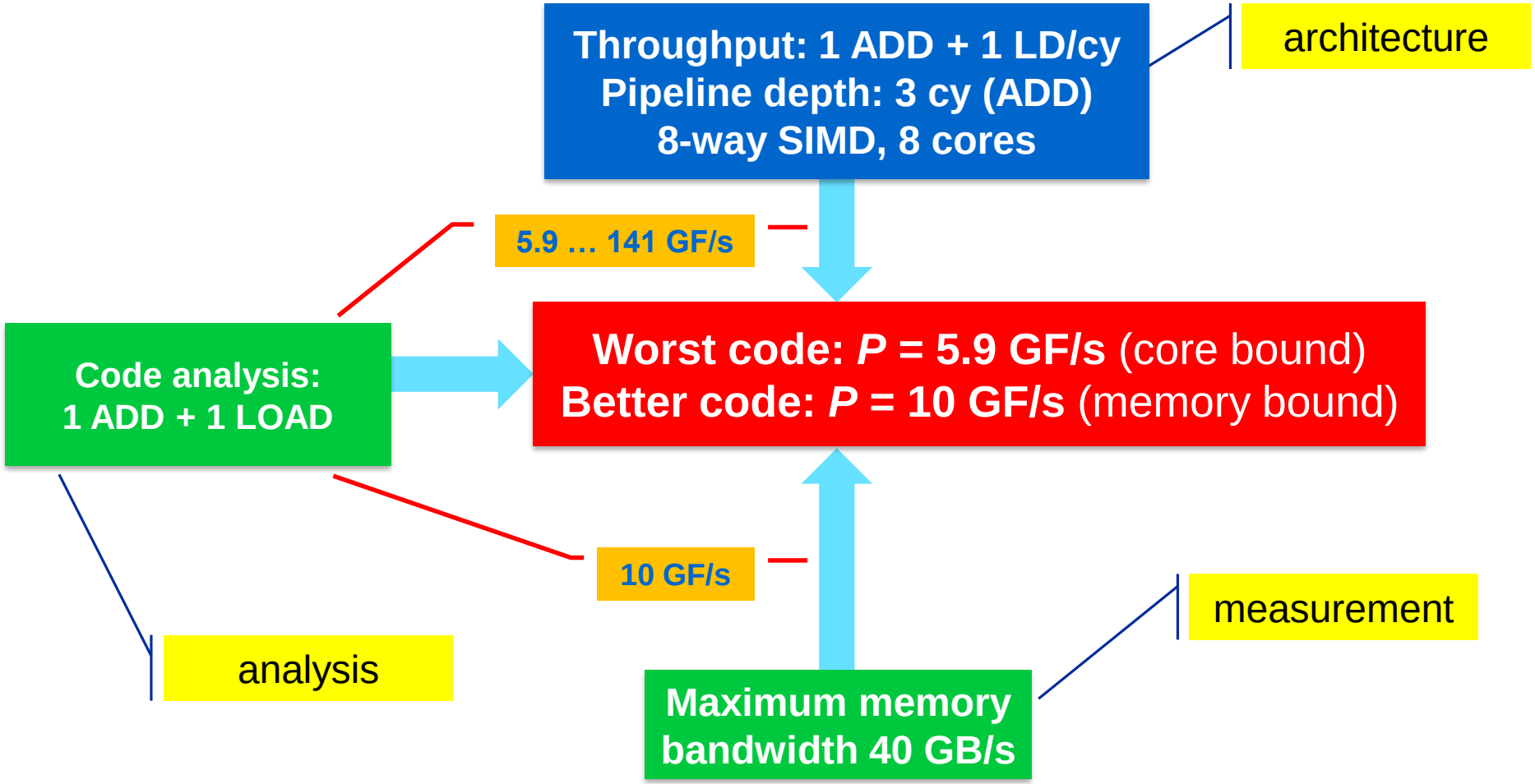
3-cycle latency per ADD if not unrolled

See architecture intro

I = 1 flop / 4 byte (SP!)

Input to the roofline model

… on the example of `do i=1,N; s=s+a(i); enddo`
in single precision



Prerequisites for the Roofline Model

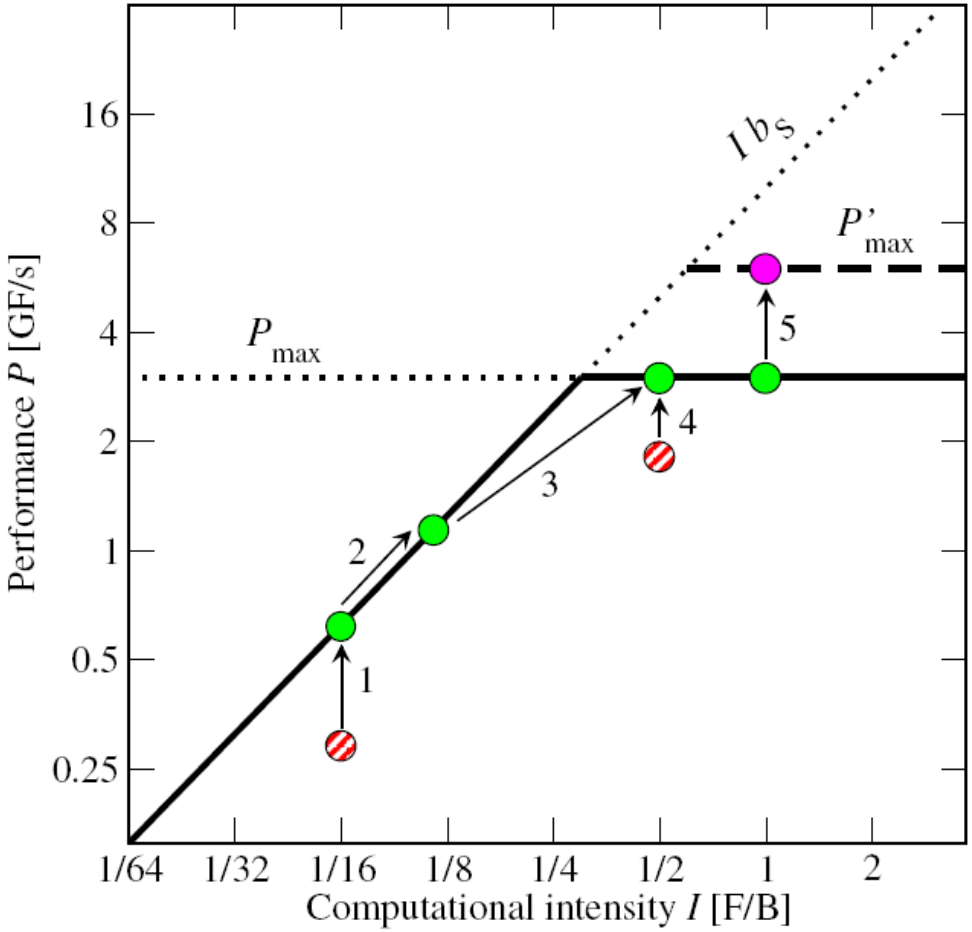
- **The roofline formalism is based on some (crucial) prerequisites:**
  - There is a clear concept of "work" vs. "traffic"
    - "work" = flops, updates, iterations…
    - "traffic" = required data to do "work"
  - Attainable bandwidth of code = input parameter! Determine effective saturated bandwidth of the chip via simple streaming benchmarks to model more complex kernels and applications

- **Assumptions behind the 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
  - Latency effects are ignored, i.e. perfect streaming mode
  - "Steady state" code execution (no wind-up/-down effects)

Typical code optimizations in the Roofline Model

optional

1. Hit the BW bottleneck by good serial code
   (e.g., Perl → Fortran)
2. Increase intensity to make better use of BW bottleneck
   (e.g., loop blocking → see later)
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` → see later)
5. Shift P_{max} by accessing additional hardware features or using a different algorithm/implementation
   (e.g., scalar → SIMD)



Multicore performance tools: Probing performance behavior

likwid-perfctr

Probing performance behavior

- **How do we find out about the performance properties and requirements of a parallel code?**
  - Profiling via advanced tools is often overkill
- **A coarse overview is often sufficient**
  - likwid-perfctr (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - Simple end-to-end measurement of hardware performance metrics
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with **`likwid-perfctr -a`**

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
FLOPS_X87: X87 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
```

likwid-perfctr

Example usage with preconfigured metric group (shortened)

```
$ likwid-perfctr -C N:0-3 -g FLOPS_DP  ./stream.exe
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
CPU type:       Intel Xeon IvyBridge EN/EP/EX processor
CPU clock:      2.20 GHz
--------------------------------------------------------------------------------
[... YOUR PROGRAM OUTPUT ...]
--------------------------------------------------------------------------------
Group 1: FLOPS_DP
+-------------------------------------+---------+------------+------------+------------+-------
|                Event                | Counter |   Core 0   |   Core 1   |   Core 2   |  Core
+-------------------------------------+---------+------------+------------+------------+-------
|          INSTR_RETIRED_ANY          |  FIXC0  |  521332883 |  523904122 |  519696583 |  519193
|        CPU_CLK_UNHALTED_CORE        |  FIXC1  | 1379625927 | 1381900036 | 1378355460 | 1376447
|        CPU_CLK_UNHALTED_REF         |  FIXC2  | 1389460886 | 1393031508 | 1387504228 | 1385276
| FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE|  PMC0   |  176216849 |  176176025 |  177432054 |  176367
| FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE|  PMC1   |    1554    |     599    |      72    |      27
|       SIMD_FP_256_PACKED_DOUBLE     |  PMC2   |       0    |       0    |       0    |       0
+-------------------------------------+---------+------------+------------+------------+-------

+----------------------+------------+------------+------------+--------------+
|        Metric        |   Core 0   |   Core 1   |   Core 2   |    Core 3    |
+----------------------+------------+------------+------------+--------------+
|  Runtime (RDTSC) [s] |   0.6856   |   0.6856   |   0.6856   |    0.6856    |
| Runtime unhalted [s] |   0.6270   |   0.6281   |   0.6265   |    0.6256    |
|      Clock [MHz]     | 2184.6742  | 2182.6664  | 2185.7404  |   2186.2243  |
|          CPI         |   2.6463   |   2.6377   |   2.6522   |    2.6511    |
|        MFLOP/s       |  514.0890  |  513.9685  |  517.6320  |   514.5273   |
|      AVX MFLOP/s     |      0     |      0     |      0     |       0      |
|    Packed MUOPS/s    |  257.0434  |  256.9838  |  258.8160  |   257.2636   |
|    Scalar MUOPS/s    |   0.0023   |   0.0009   |   0.0001   | 3.938426e-05 |
+----------------------+------------+------------+------------+--------------+
```

Always measured

Configured metrics (this group)

Derived metrics

likwid-perfctr

Marker API (C/C++ and Fortran)

- A **marker API** is available to restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by **`likwid-perfctr`**
- Multiple named region support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid-marker.h>
. . .
LIKWID_MARKER_INIT;                    // must be called from serial region
#pragma omp parallel
{
  LIKWID_MARKER_THREADINIT;            // only reqd. if measuring multiple threads
}
. . .
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
```

- **Activate macros with `-DLIKWID_PERFMON`**
- **Run `likwid-perfctr` with `-m` option to activate markers**

Things to look at (in roughly this order)

- Excess work
- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions, branches, mispredicted branches

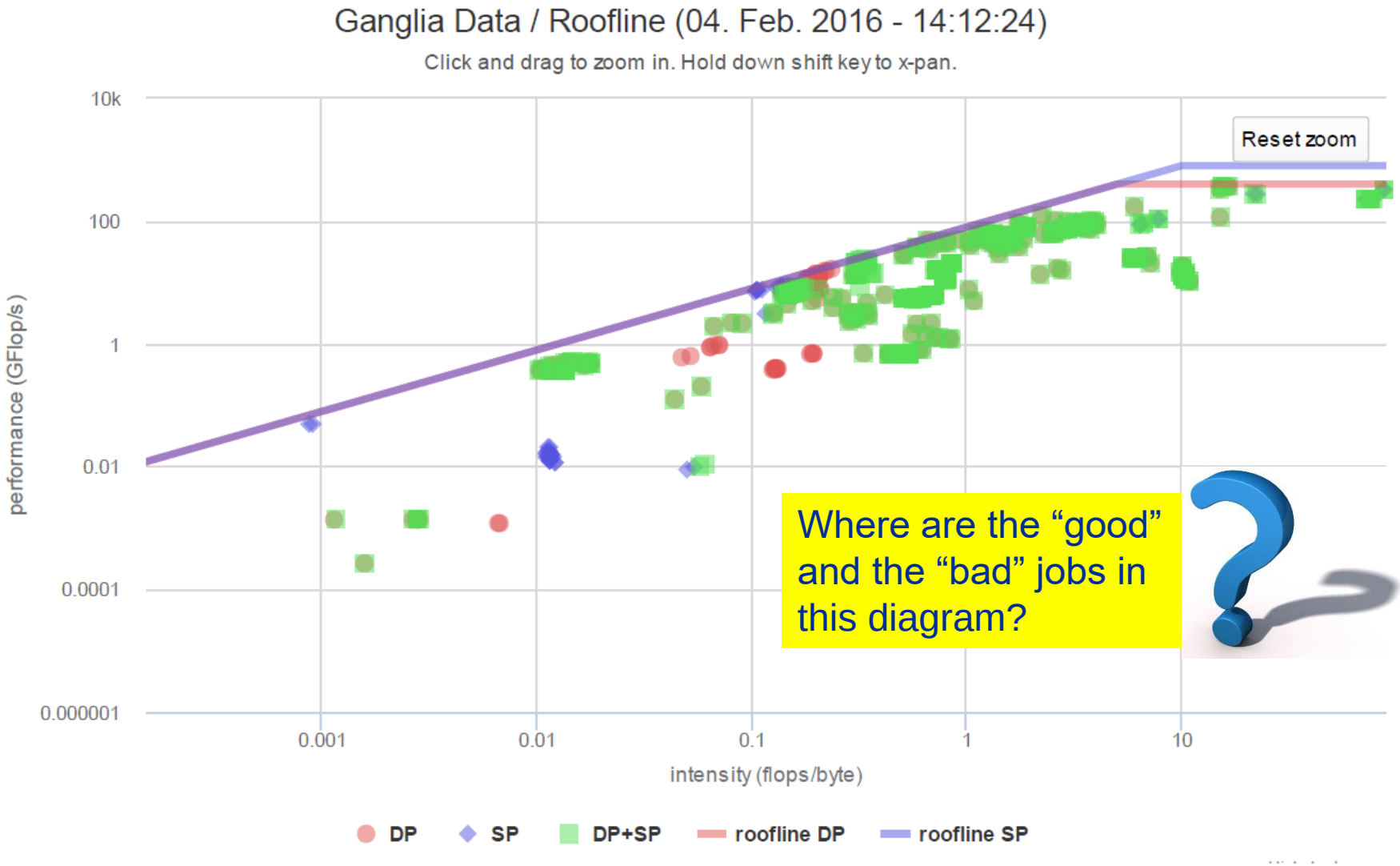
Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are sometimes misleading

Using Roofline for monitoring “live” jobs on a cluster

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

optional



Case study: A Jacobi smoother

The basic performance properties in 2D

Layer conditions

Optimization by spatial blocking

Stencil schemes

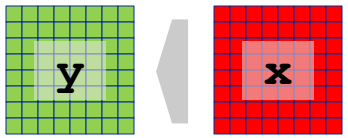
- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- but the regular access structure allows for matrix free coding

```
do iter = 1, max_iterations

  Perform sweep over regular grid: y(:) ← x(:)

  Swap y ←→ x

enddo
```



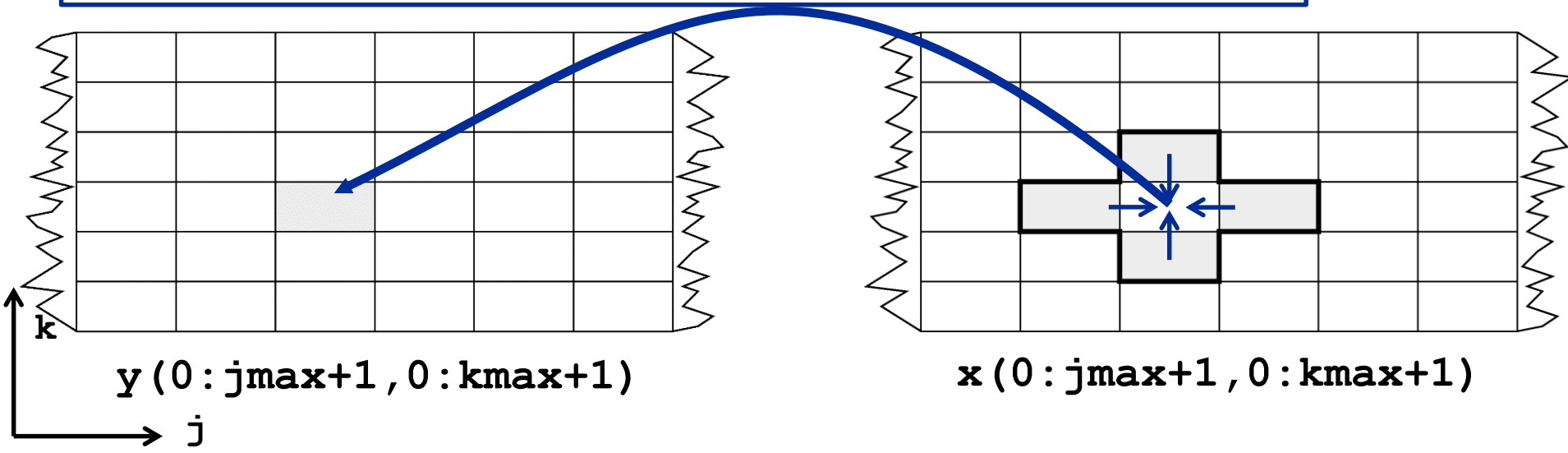
- Complexity of implementation and performance depends on
  - update scheme, e.g. Jacobi-type, Gauss-Seidel-type, …
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…

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
```

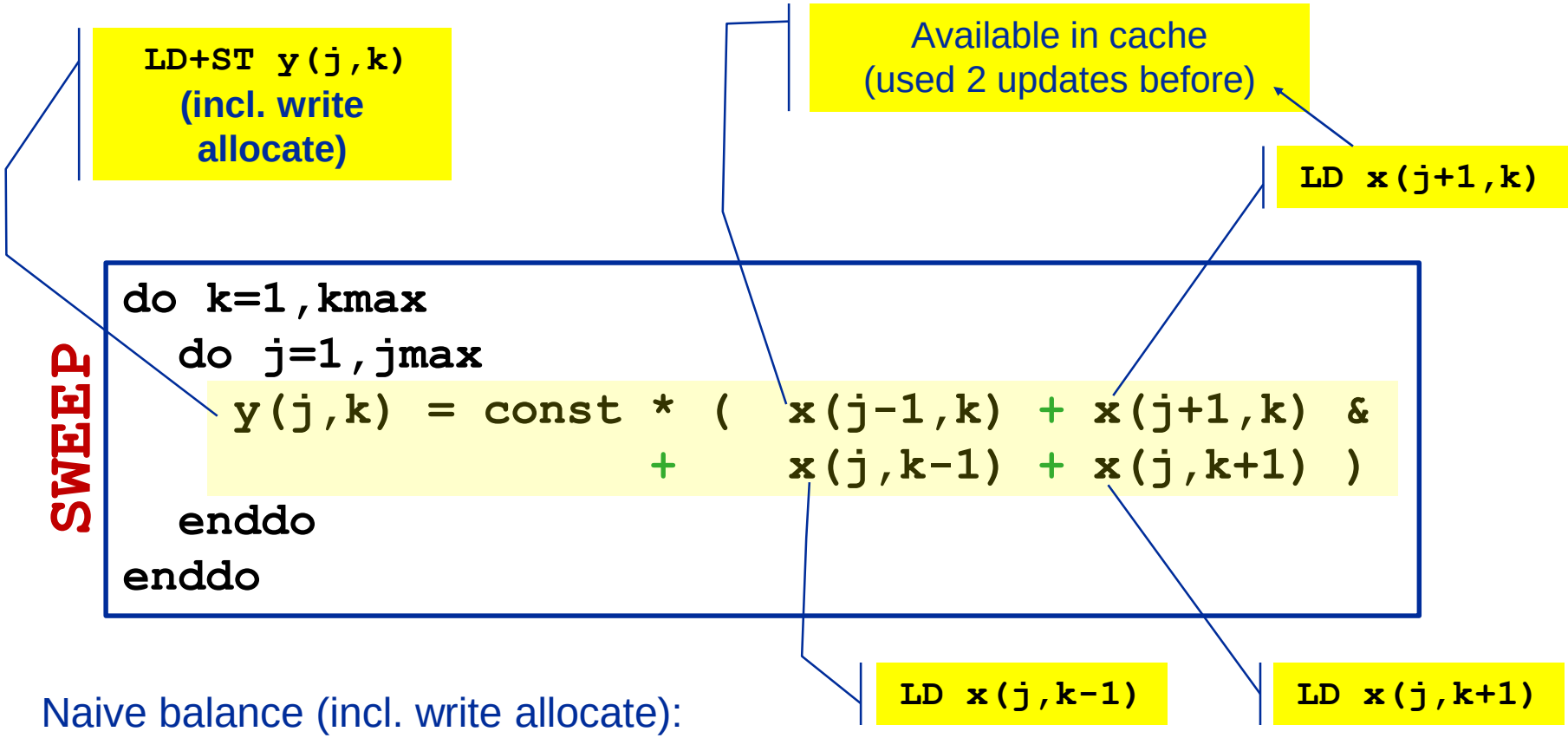
Lattice site Update (LUP)



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 in 2D: data transfer analysis

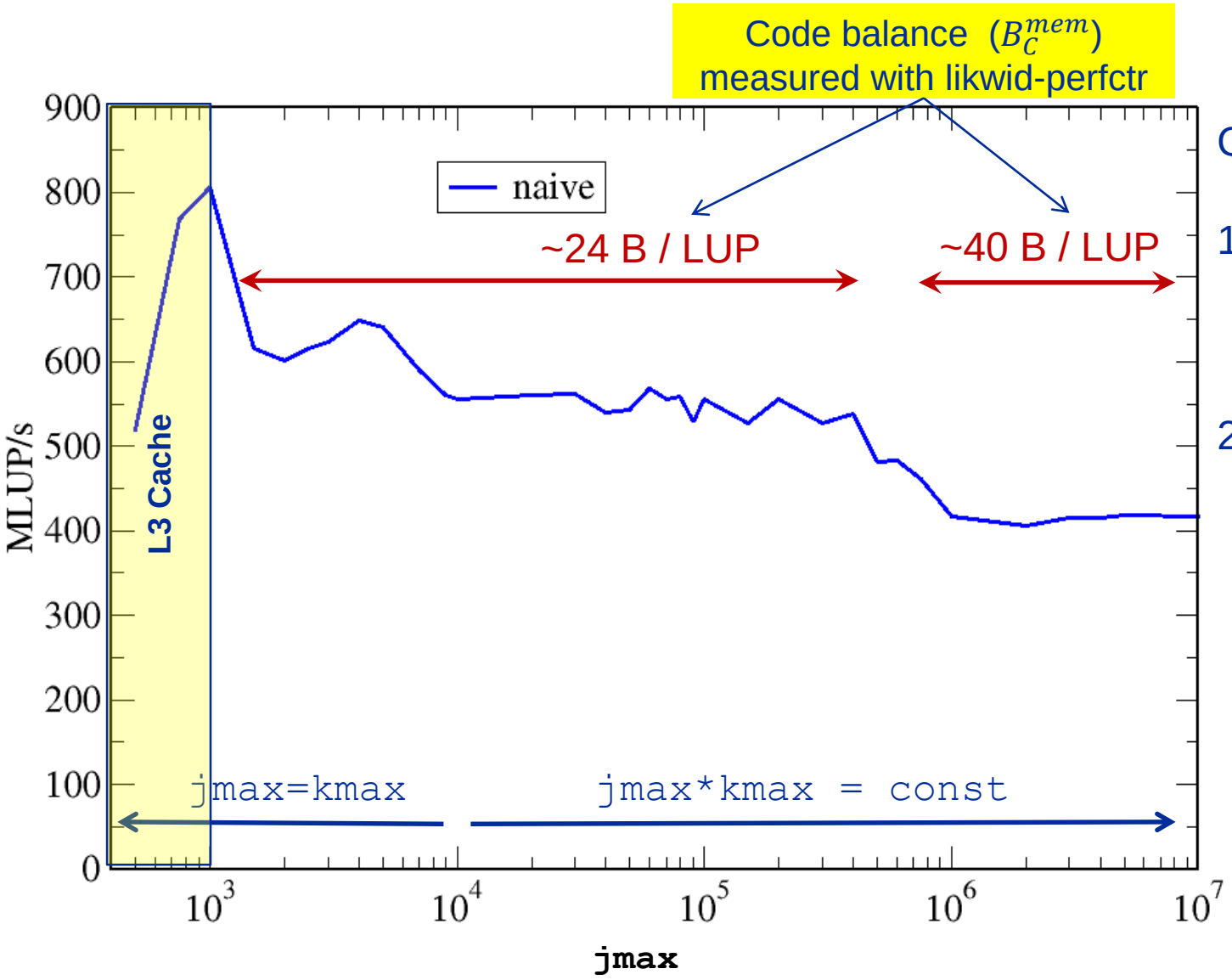


Naive balance (incl. write allocate):

x(:, :) : 3 LD +
y(:, :) : 1 ST+ 1LD

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

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



Questions:

1. How to achieve 24 B/LUP also for large **jmax**?
2. How to sustain >600 MLUP/s for **jmax > 10^4** ?

Intel Compiler
ifort V13.1
Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

Case study: A Jacobi smoother

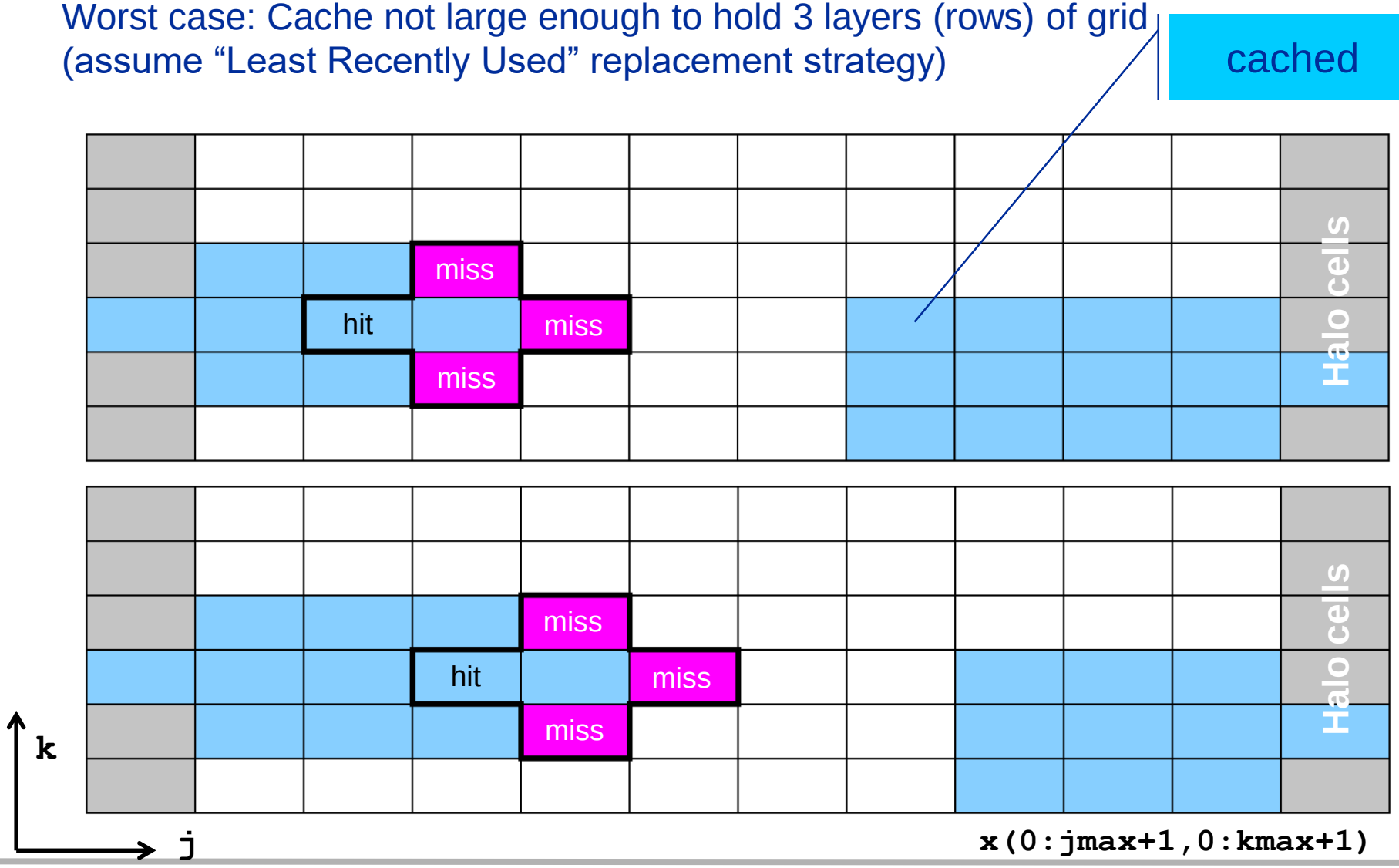
The basics in two dimensions

Layer conditions

Optimization by spatial blocking

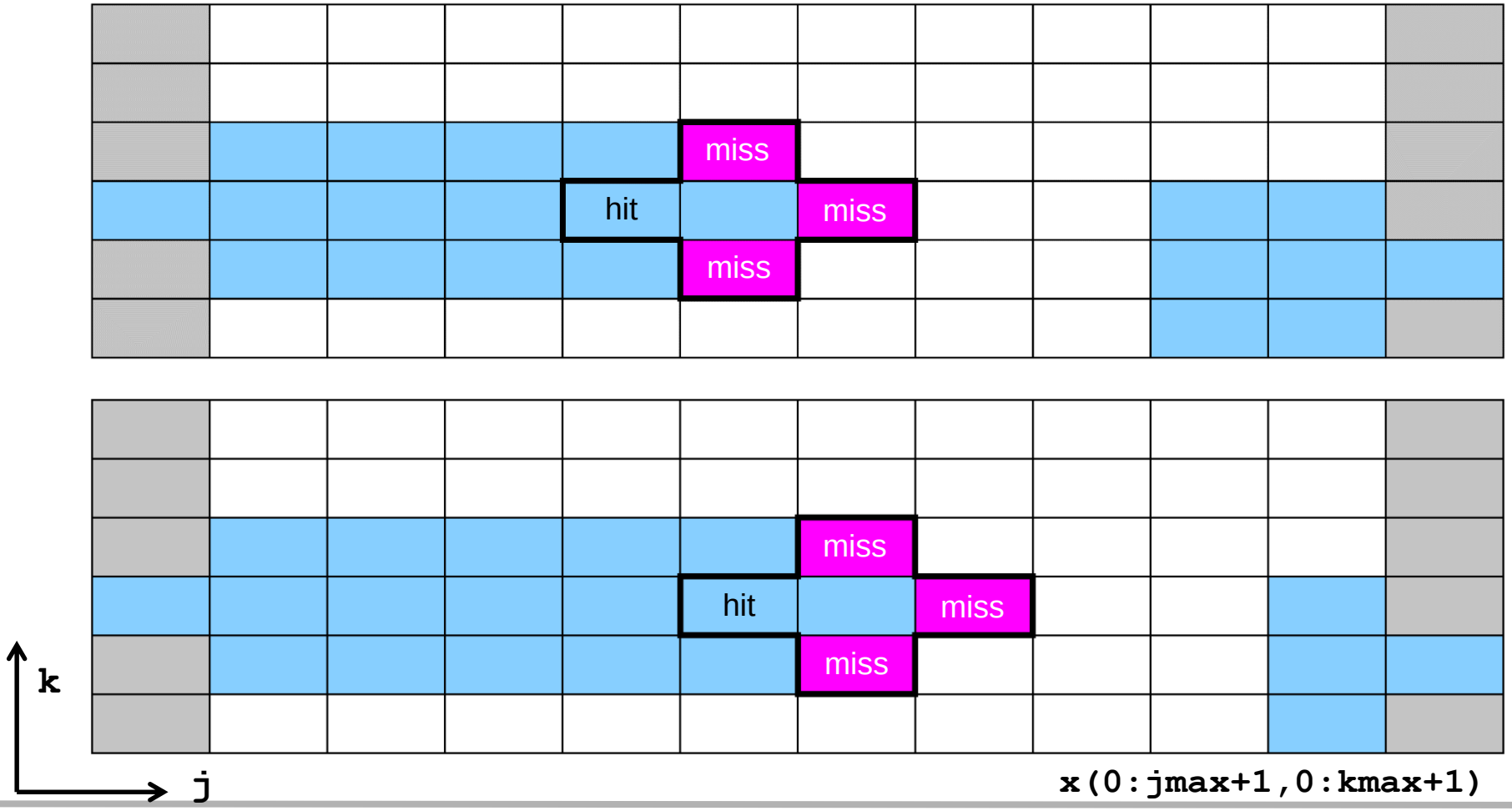
Analyzing the data flow

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



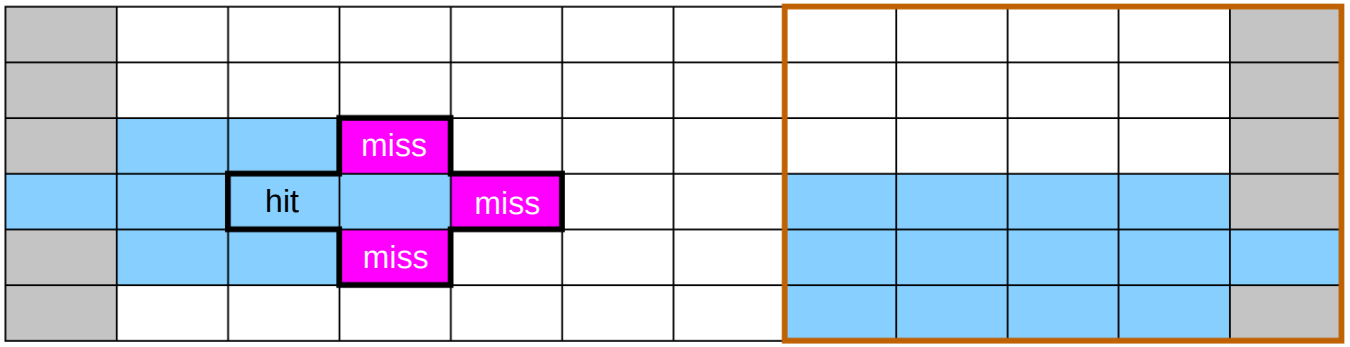
Analyzing the data flow

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

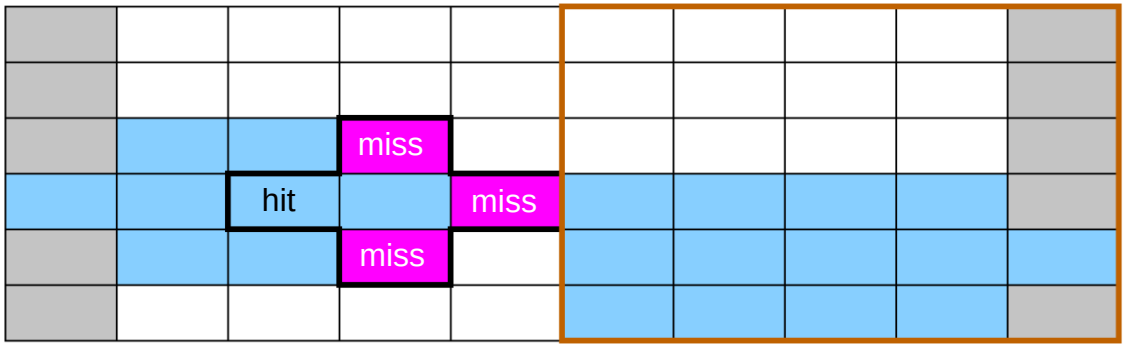


Analyzing the data flow

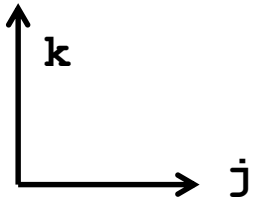
Reduce inner (j-) loop dimension successively

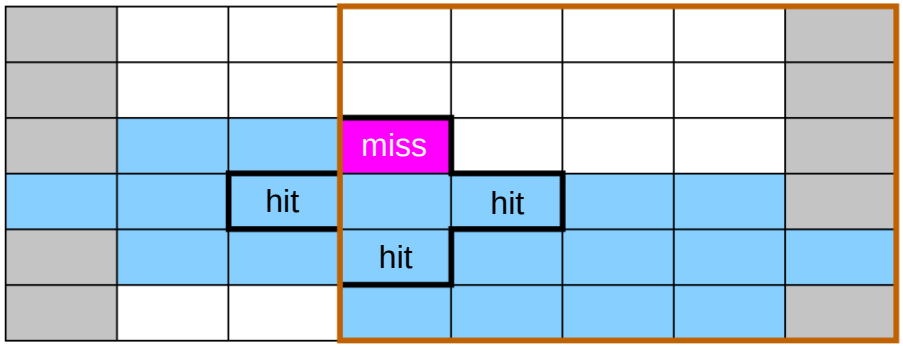


`x(0:jmax1+1,0:kmax+1)`



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

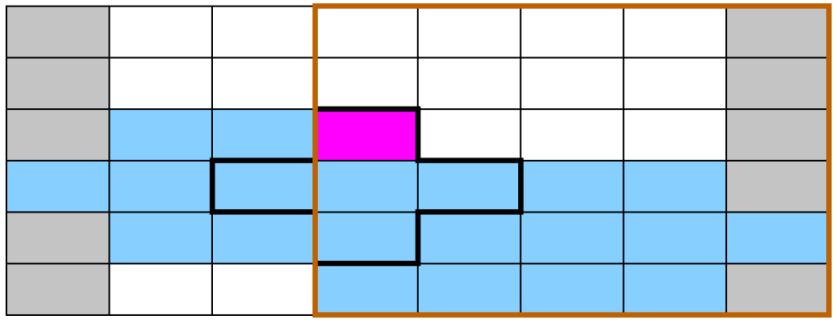




`x(0:jmax2+1,0:kmax+1)`

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil



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

3 * jmax * 8B < CacheSize/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., 3D 7-pt stencil)

Analyzing the data flow: Layer condition (2D 5-pt Jacobi)

y: (1 LD + 1 ST) / LUP

x: 1 LD / LUP

```
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
```

B_C = 24 B / LUP

YES

3 * jmax * 8B < CacheSize/2

Layer condition fulfilled?

NO

y: (1 LD + 1 ST) / LUP

```
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
```

x: 3 LD / LUP

B_C = 40 B / LUP

Fulfilling the layer condition

Establish 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 !      Assume jmax is multiple of jblock
  do k=1,kmax
    do j= jb, (jb+jblock-1) ! Length of inner loop: jblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      +  x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

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

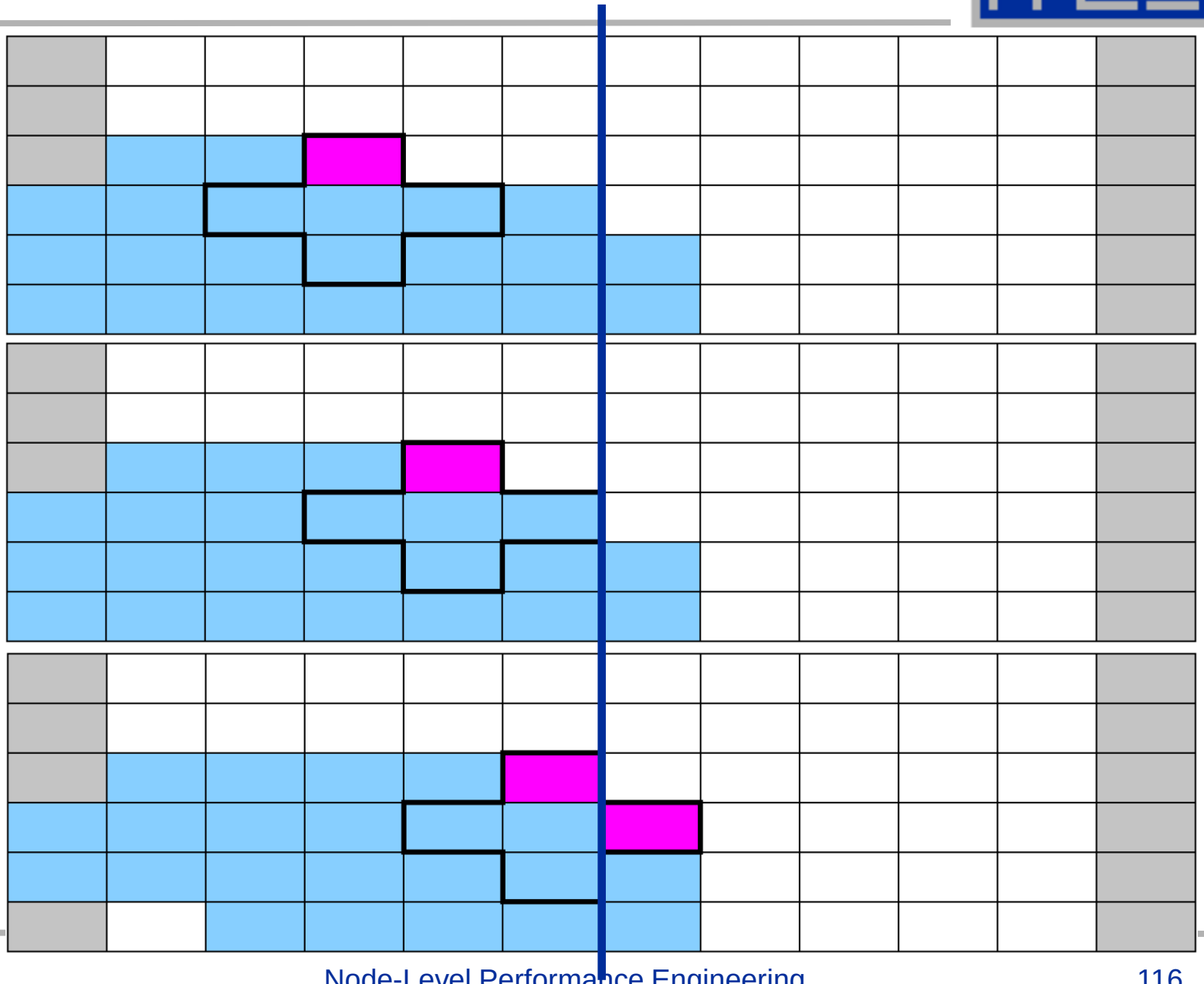
→ Determine for given **CacheSize** an appropriate **jblock** value:

jblock < CacheSize / 48 B

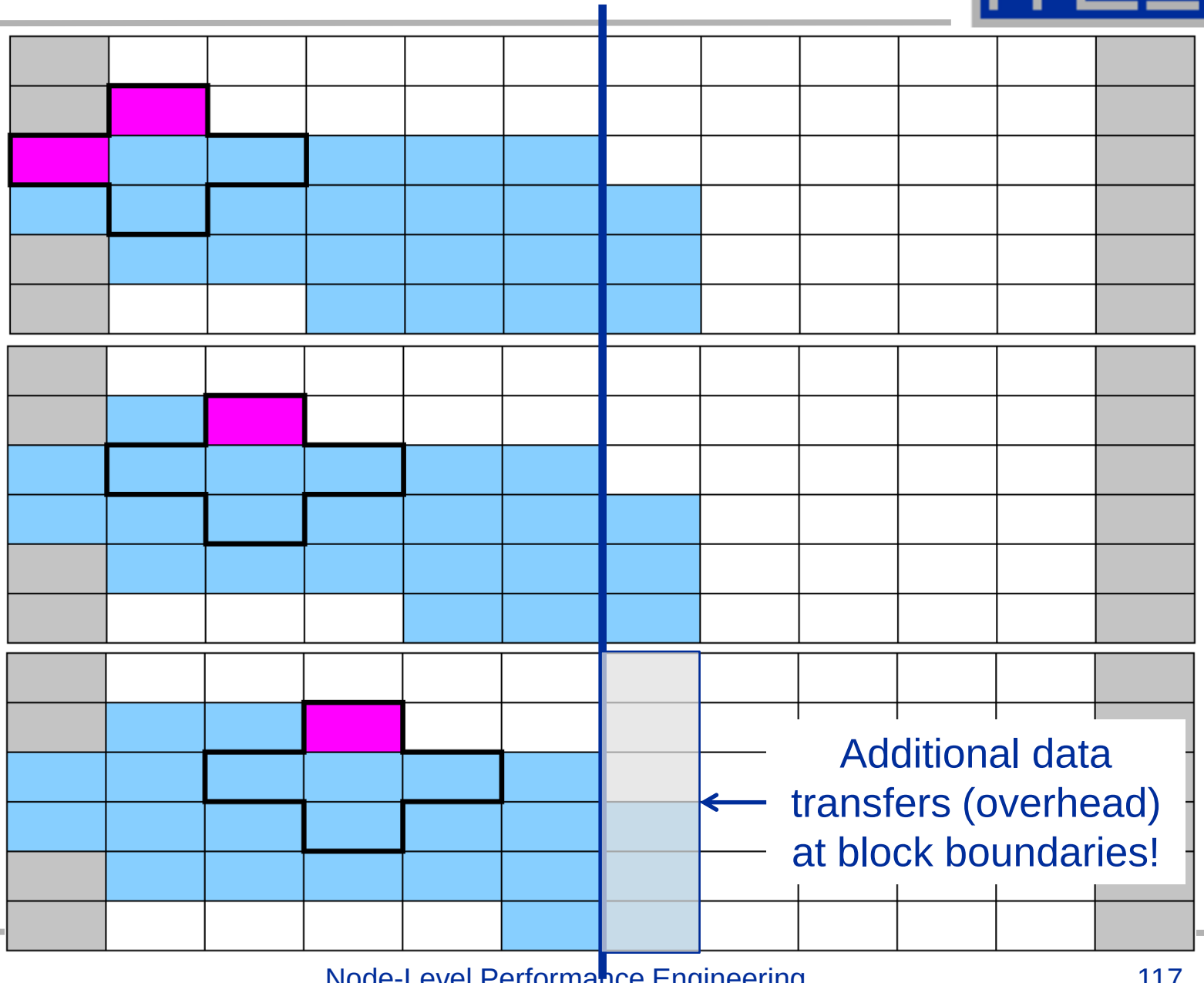
Establish the layer condition by blocking

Split up domain into subblocks:

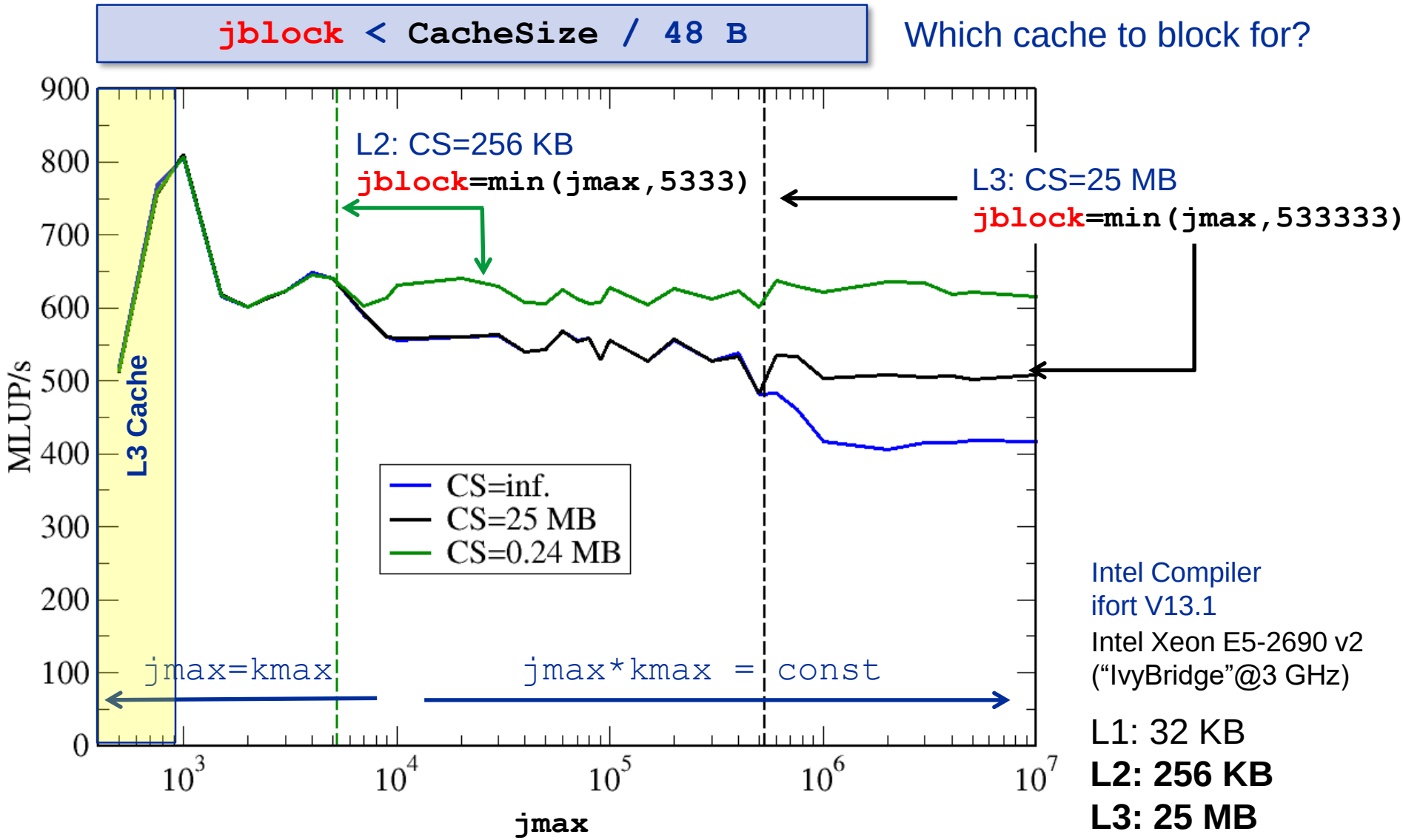
e.g. block size = 5



Establish the layer condition by blocking



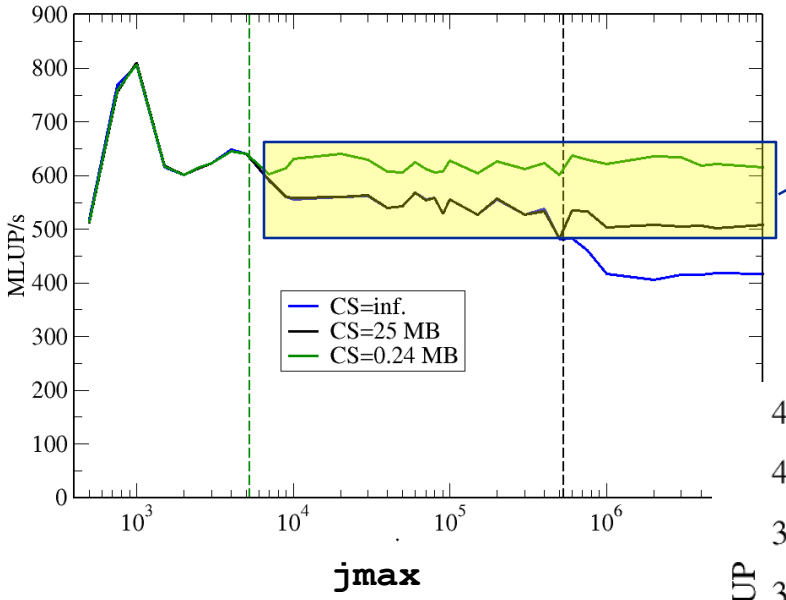
Establish layer condition by spatial blocking



Intel Compiler
ifort V13.1
Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

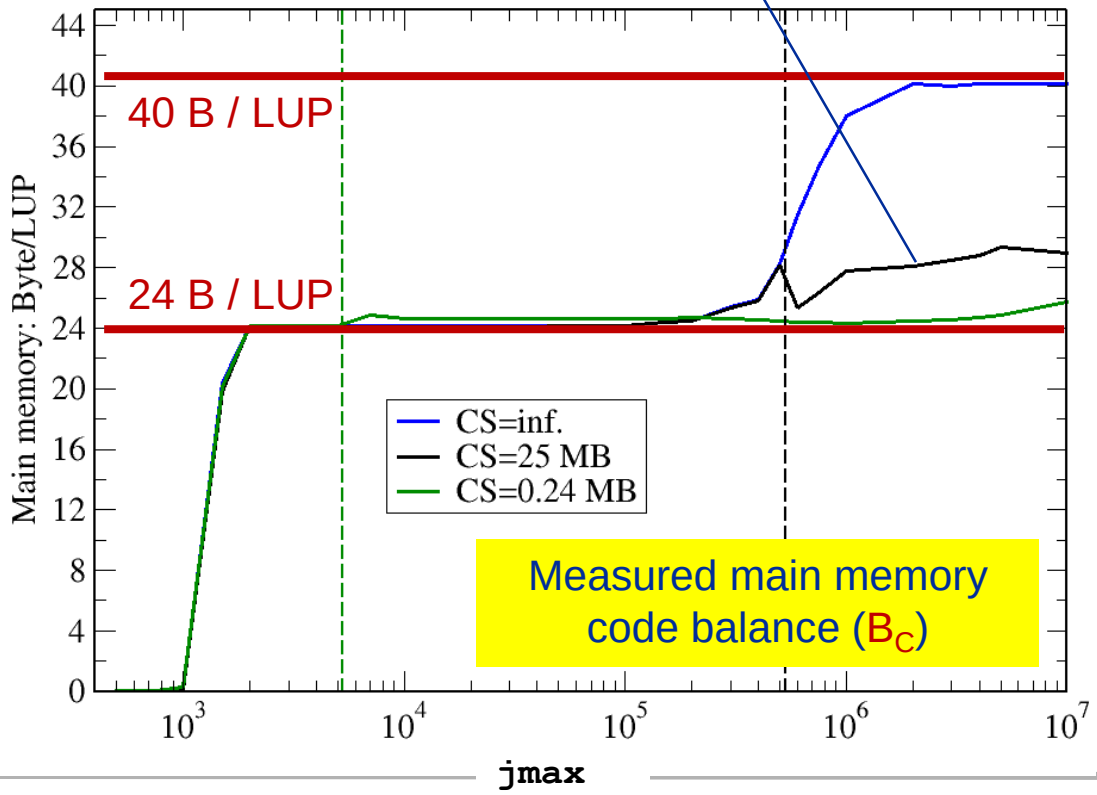
L1: 32 KB
L2: 256 KB
L3: 25 MB

Validating the hypotheis: Measure memory code balance



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

Blocking factor (CS=25 MB) still a little too large



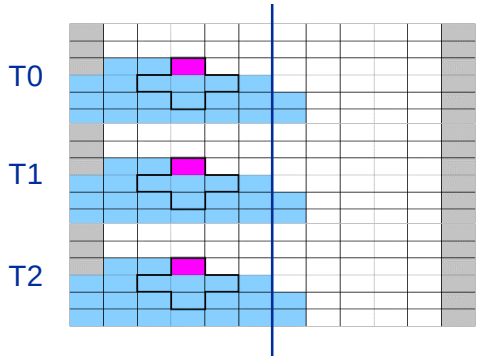
Measured main memory code balance (B_C)

Intel Compiler
ifort V13.1
Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

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!

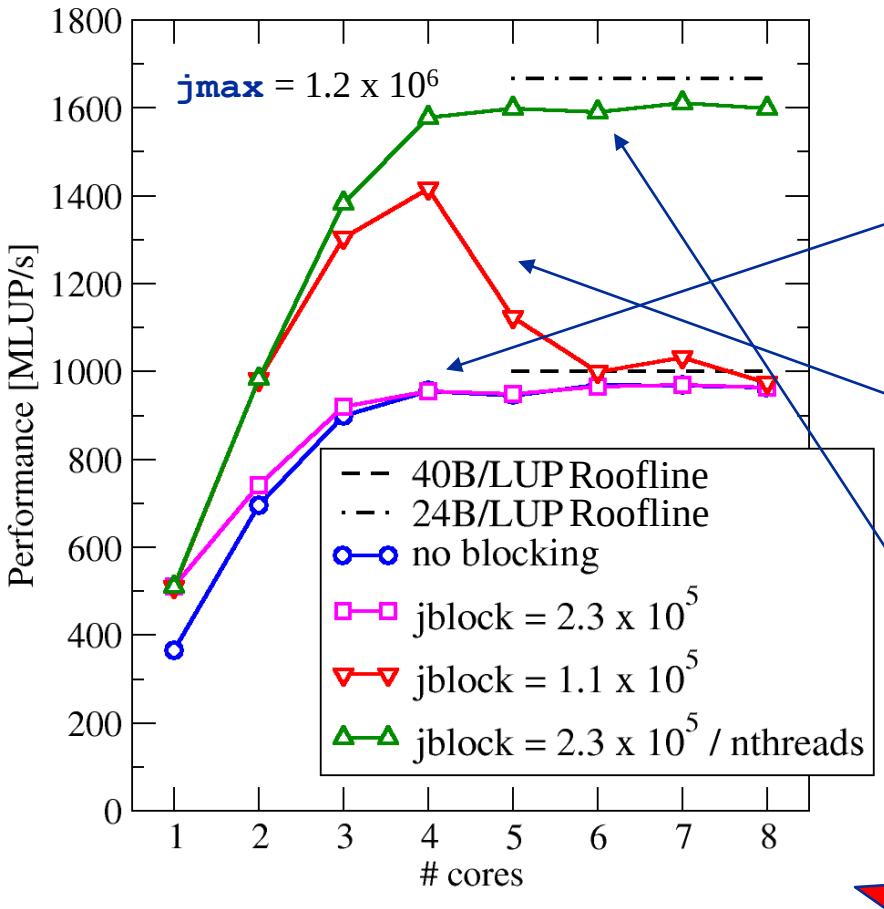
Layer condition:

$3 * jblock * 8B < CS_t/2$

Cache size available per thread

OpenMP parallelization and blocking for shared cache

optional



Example: 2D 5-point stencil on Sandy Bridge 8-core, 20 MB L3

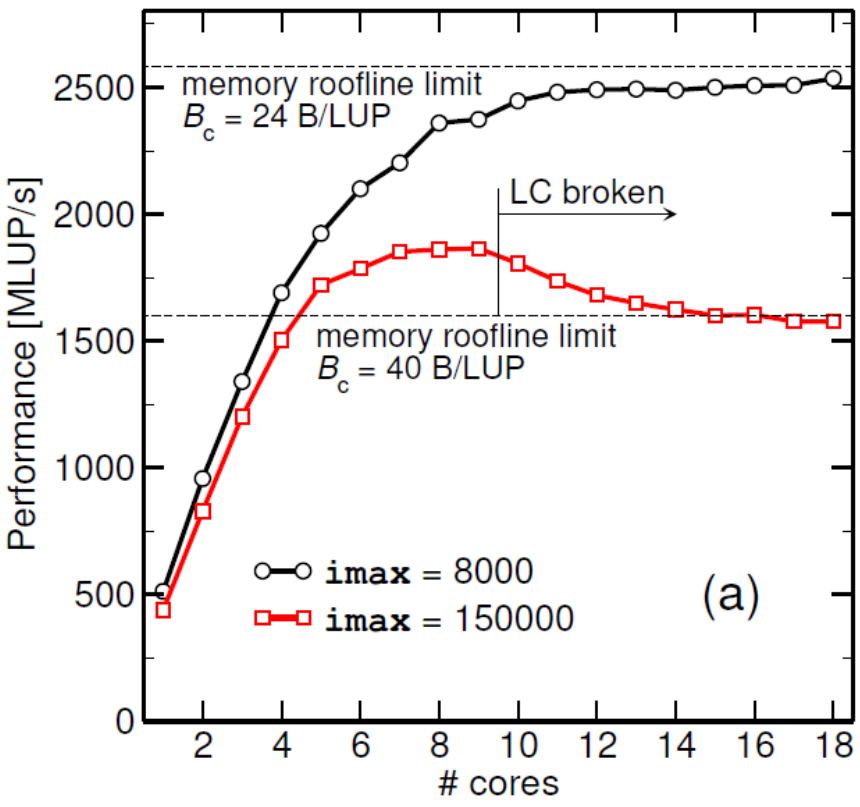
- Optimal **`jblock`** for 1 thread is too small for multiple threads
- Smaller but constant **`jblock`** works for few threads but not for all
- Optimal blocking for shared cache requires adaptive block size

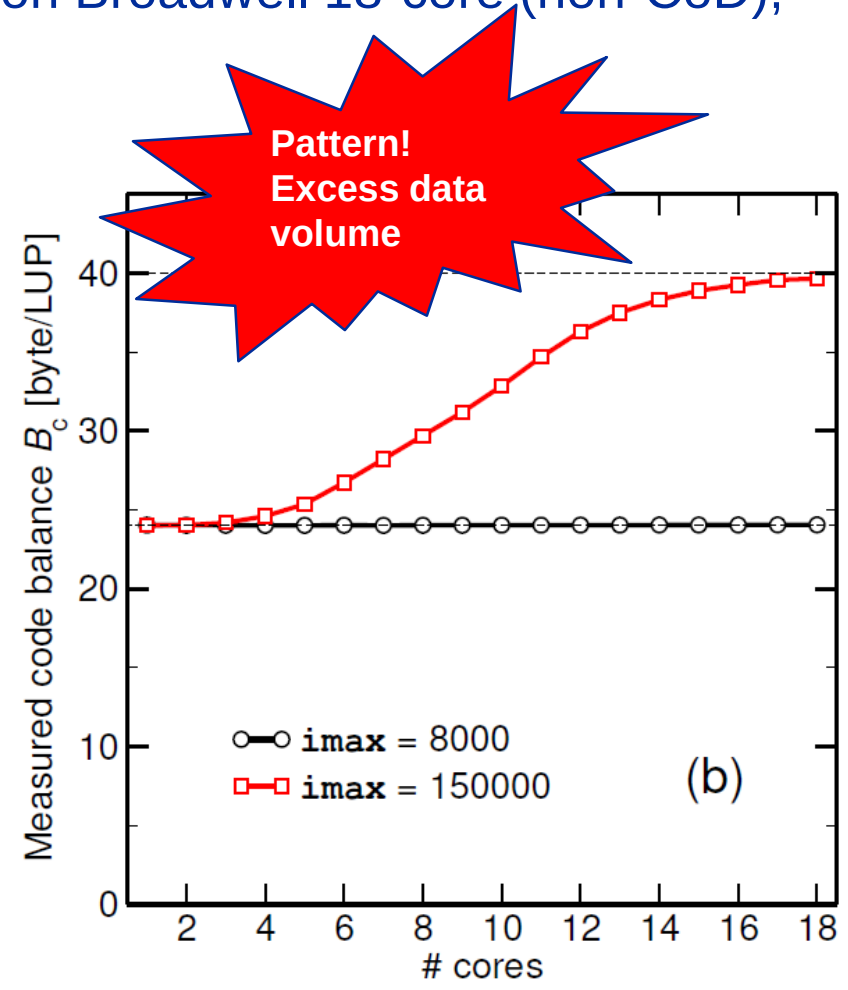
Pattern!
Excess data volume

OpenMP parallelization and blocking for shared cache

optional

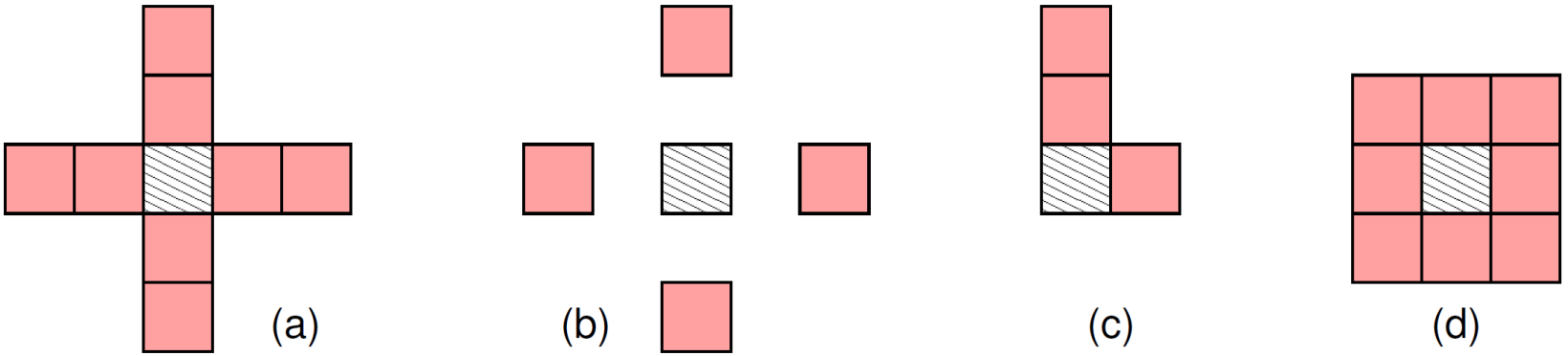
Example: 2D 5-point stencil on Intel Xeon Broadwell 18-core (non-CoD), 45 MiB of shared L3 cache





Stencil shapes and layer conditions

optional



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

b) Long-range $r = 2$ with gaps: 6 layers (2 per populated row)

c) Asymmetric: 3 layers

d) 2D box: 3 layers

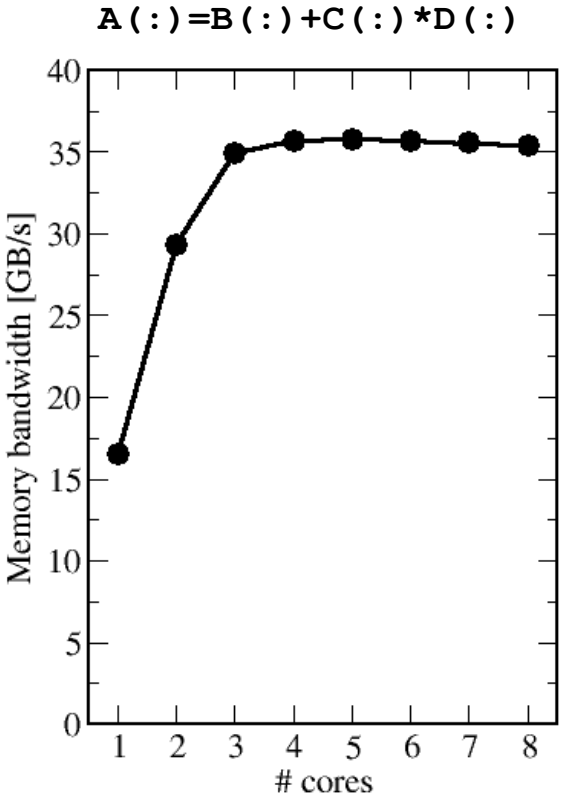
Conclusions from the Jacobi example

- 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 would change if we parallelized inner loops?

Shortcomings of the roofline model

- Saturation effects in multicore chips are not explained
  - Reason: "saturation assumption"
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - It is not sufficient to measure single-core STREAM to make it work
  - Only increased "pressure" on the memory interface can saturate the bus
    → need more cores!
- In-cache performance is not correctly predicted
- The ECM performance model gives more insight:

H. Stengel, J. Treibig, G. Hager, and G. Wellein: *Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model*. Proc. ICS15, the 29th International Conference on Supercomputing, June 8-11, 2015, Newport Beach, CA. DOI: 10.1145/2751205.2751240. Preprint: arXiv:1410.5010

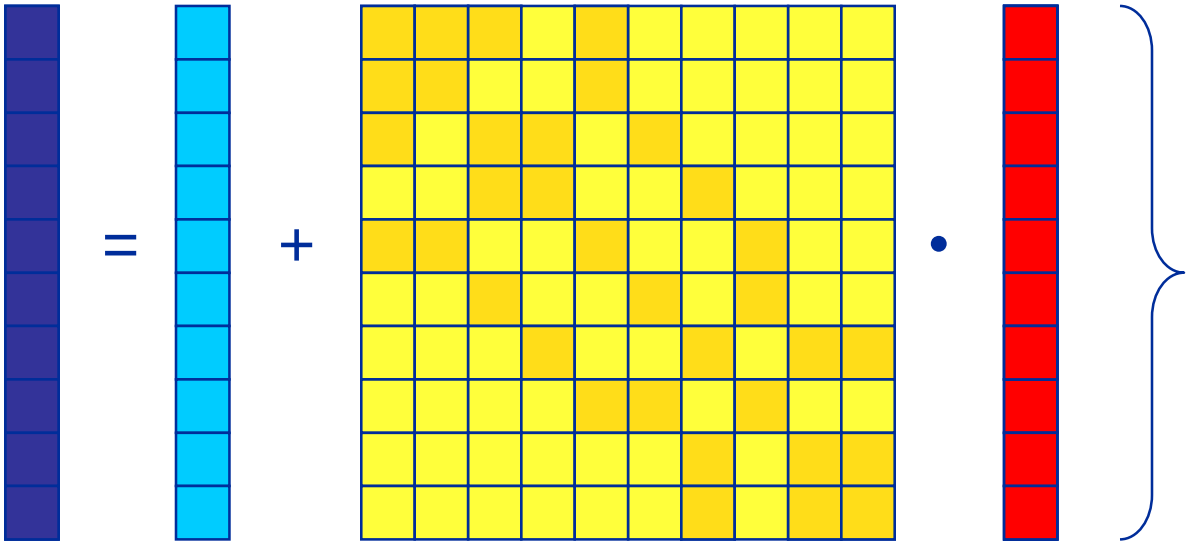


Case study:
Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson

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

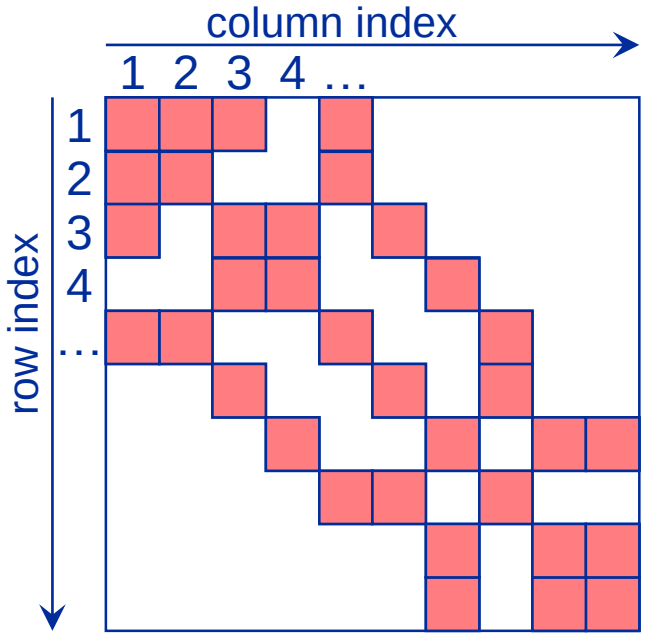


General case: some indirect addressing required!

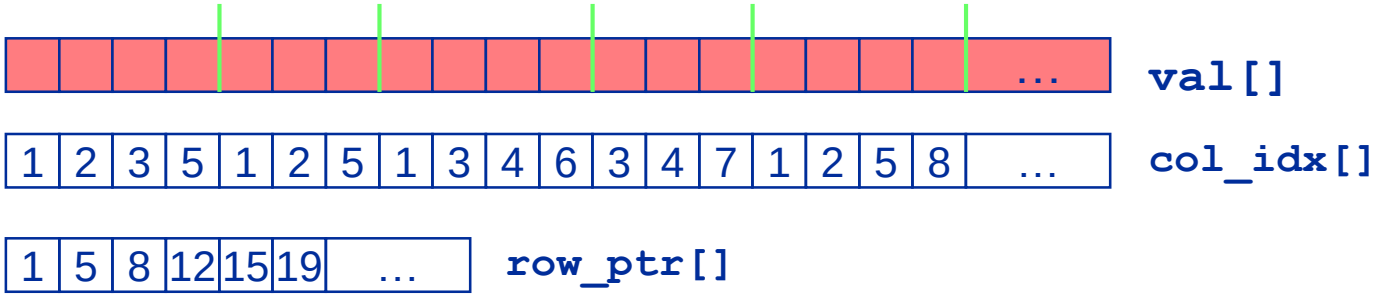
SpMVM characteristics

- 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…

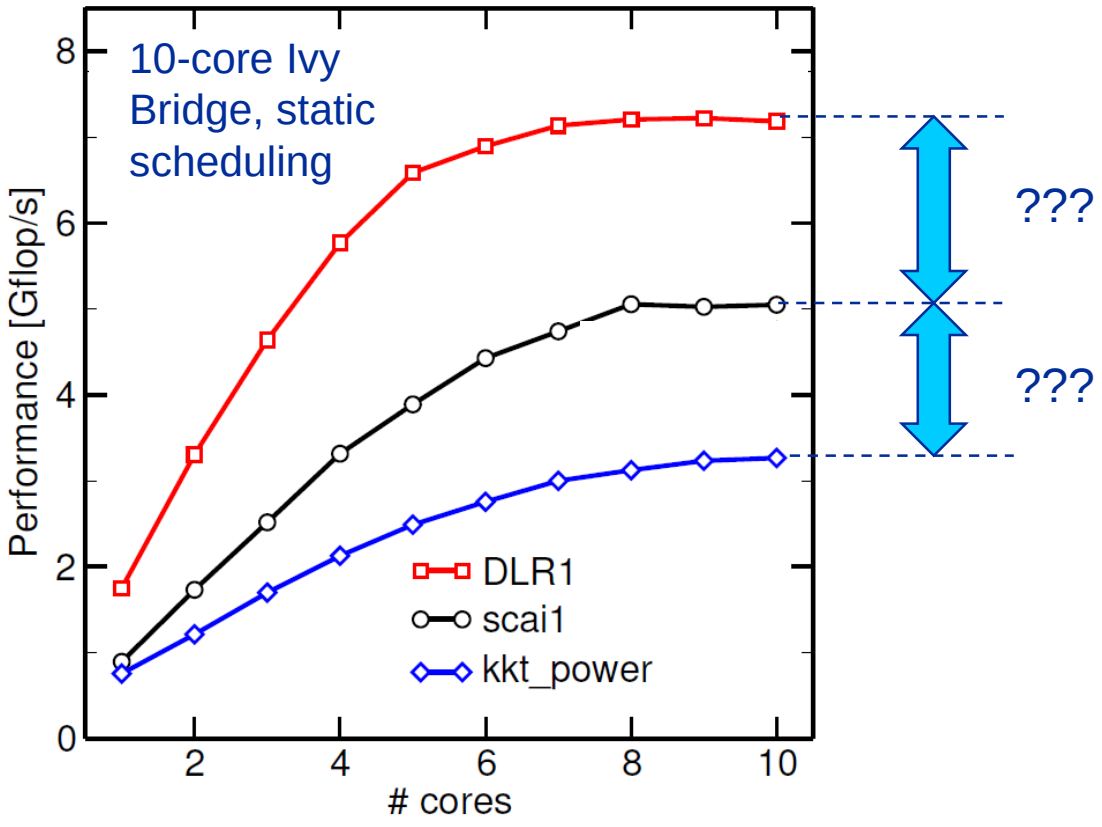
Performance characteristics

- 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?



Example: SpMV node performance model

Sparse MVM in double precision w/ CRS data storage:

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

$$B_c^{DP,CRS} = \frac{8+4+8\alpha+20/N_{nzr}}{2}\frac{\mathrm{B}}{\mathrm{F}} = \left(6+4\alpha+\frac{10}{N_{nzr}}\right)\frac{\mathrm{B}}{\mathrm{F}}$$

Absolute minimum code balance: $B_{\min} = 6\ \frac{\mathrm{B}}{\mathrm{F}}$

→ $I_{\max} = \frac{1}{6}\ \frac{\mathrm{F}}{\mathrm{B}}$

Hard upper limit for in-memory performance: $b_S/B_{\min}$

The "α effect"

DP CRS code balance

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

$$B_c^{DP,CRS}(\alpha) = \frac{8+4+8\alpha+20/N_{nzr}}{2}\frac{\mathrm{B}}{\mathrm{F}} = \left(6+4\alpha+\frac{10}{N_{nzr}}\right)\frac{\mathrm{B}}{\mathrm{F}}$$

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

Determine α (RHS traffic quantification)

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

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

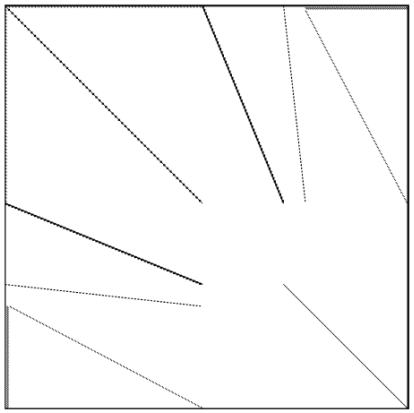
$$\alpha = \frac{1}{4}\left(\frac{V_{meas}}{N_{nz}\cdot 2\ \text{bytes}} - 6 - \frac{10}{N_{nzr}}\right)$$

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
- and:

$$\frac{B_c^{DP,CRS}(\alpha)}{B_c^{DP,CRS}(1/N_{nzr})} = 1.11$$

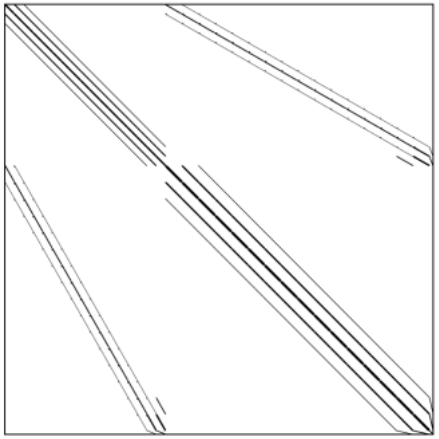
11% extra traffic → optimization potential!



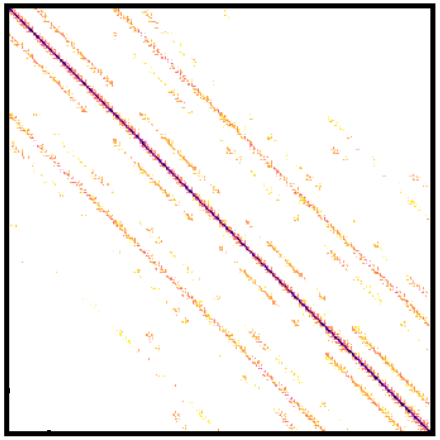
Three different sparse matrices

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

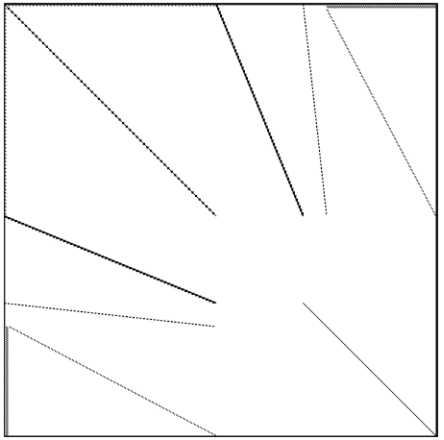
Matrix	N	N_{nzr}	B_c^{opt} [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

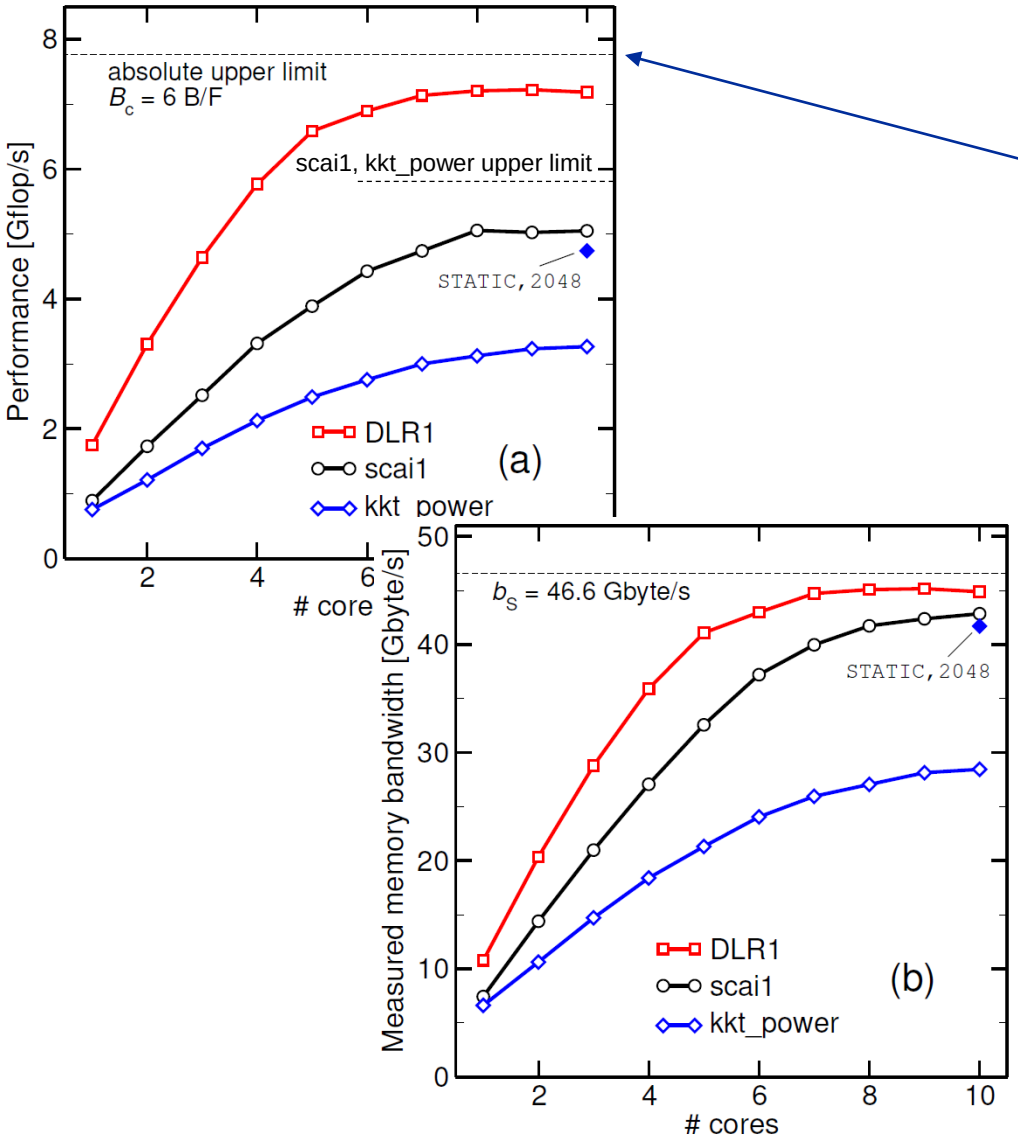


scai1



kkt_power

Now back to the start…



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

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

- DLR1 causes minimum CRS code balance (as expected)
- scai1 measured balance:

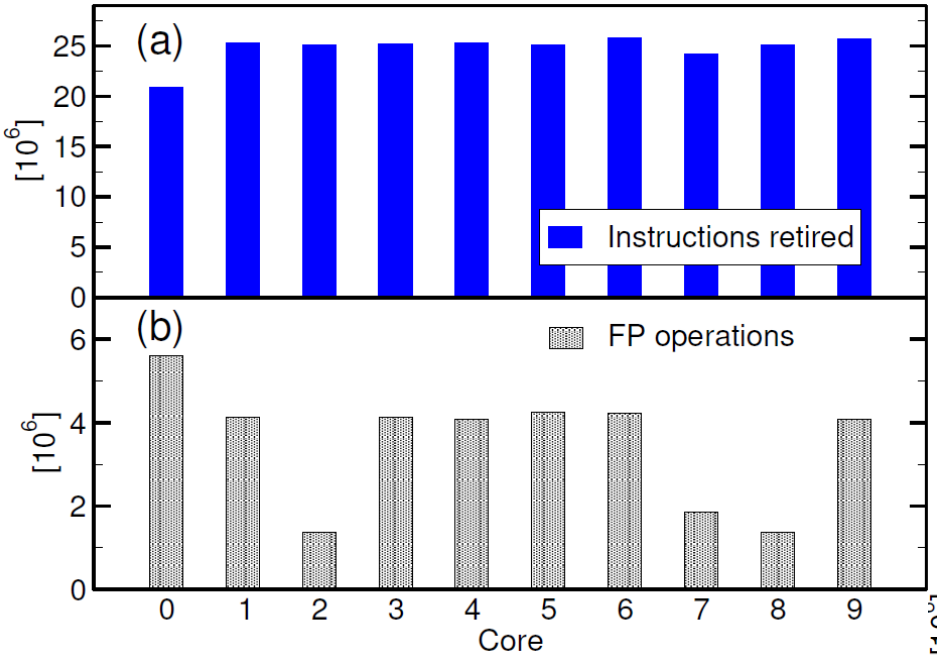
$$B_c^{meas} \approx 8.5\,\mathrm{B/F} > B_c^{opt}$$

- → good BW utilization, slightly non-optimal α
- kkt_power measured balance:

$$B_c^{meas} \approx 8.8\,\mathrm{B/F} > B_c^{opt}$$

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

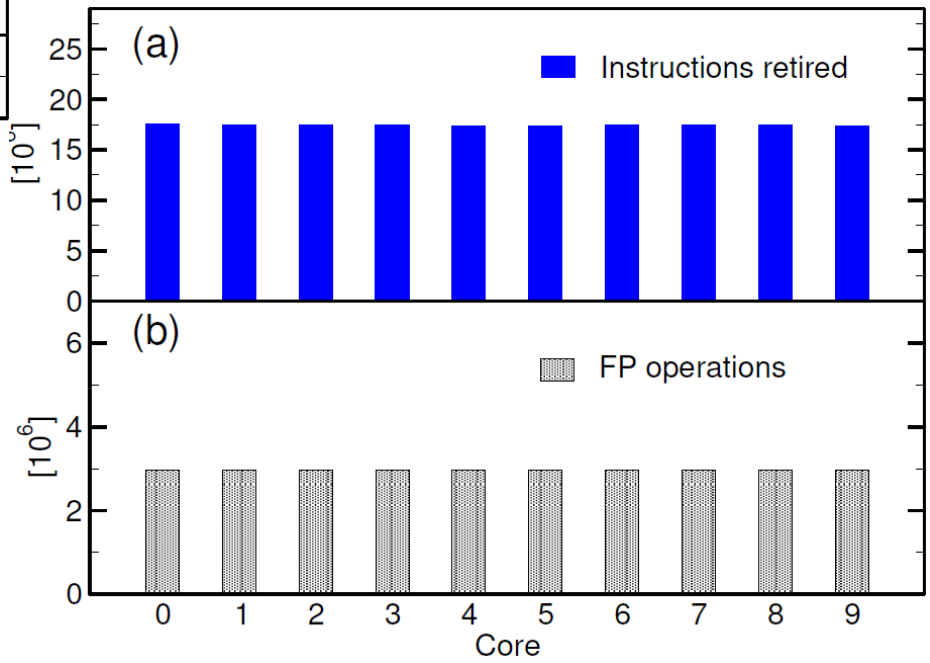
Investigating the load imbalance with kkt_power



Measurements with likwid-perfctr (MEM_DP group)

`static`

`static,2048`



→ Fewer overall instructions, (almost) BW saturation, 50% better performandce with load balancing

→ CPI value unchanged!

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!

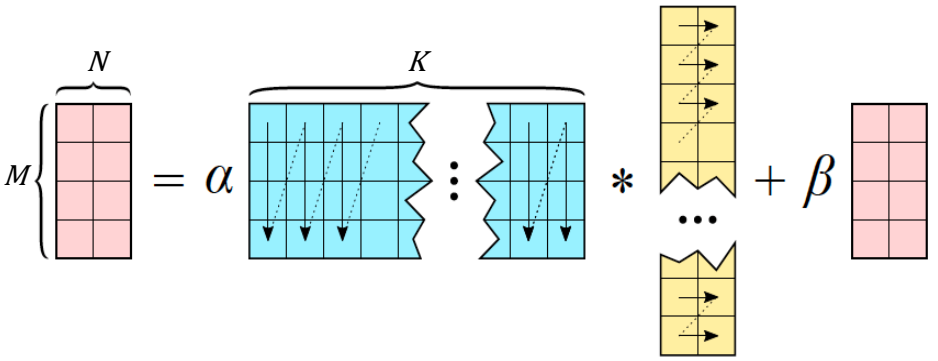
- Consequence: Modeling is not always 100% predictive. It‘s all about *learning more* about performance properties!

Case study: Tall & Skinny Matrix-Transpose Times Tall & Skinny Matrix (TSMTTSM) Multiplication

TSMTTSM Multiplication

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

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



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

TSMTTSM Multiplication

General rule for dense matrix-matrix multiply: Use vendor-optimized GEMM, e.g. from Intel MKL[1]:

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

double

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

complex double

TS@MKL: Good or bad?

Matrix sizes:

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

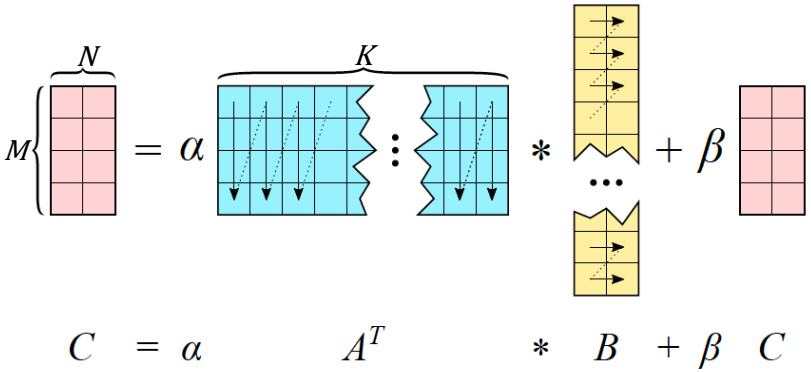
Tall&Skinny (TS): M=N=16 ; K=10,000,000

[1]Intel Math Kernel Library (MKL) 11.3

TSMTTSM Roofline model

Computational intensity

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



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

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

complex double:

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

TSMTTSM Roofline performance prediction

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

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

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

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

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

→ Potential speedup: 6–10x vs. MKL

Can we implement a better TSMTTSM kernel than Intel?

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

Thread local copy of small (results) matrix

Long Loop (k): Parallel

Outer Loop Unrolling

Compiler directives

Most operations in cache

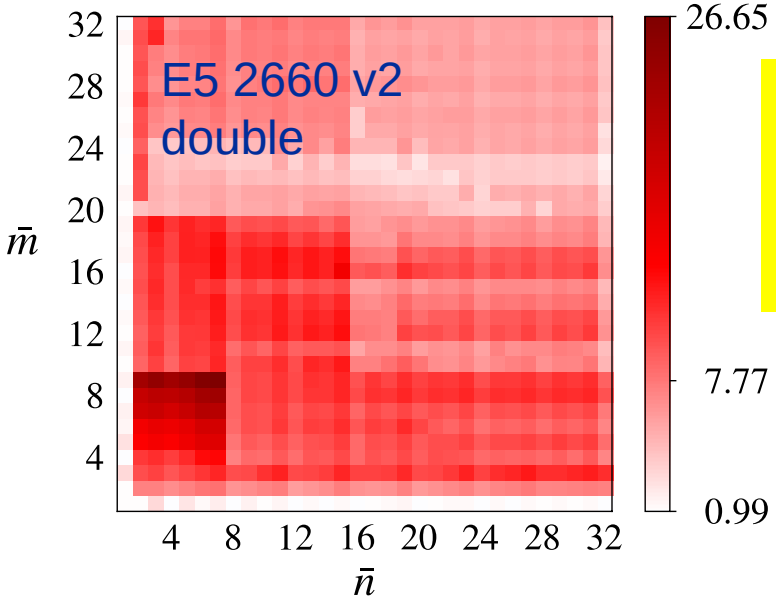
Reduction on small result matrix

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

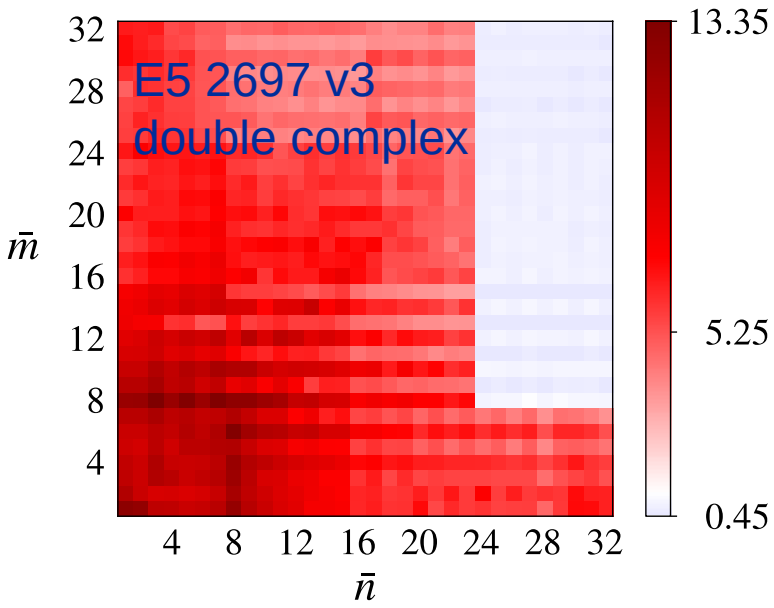
TSMTTSM MKL vs. “hand crafted” (OPT)

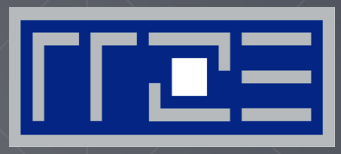
TS: M=N=16 ; K=10,000,000

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



Speedup vs. MKL: 5x – 25x





Single Instruction Multiple Data (SIMD)

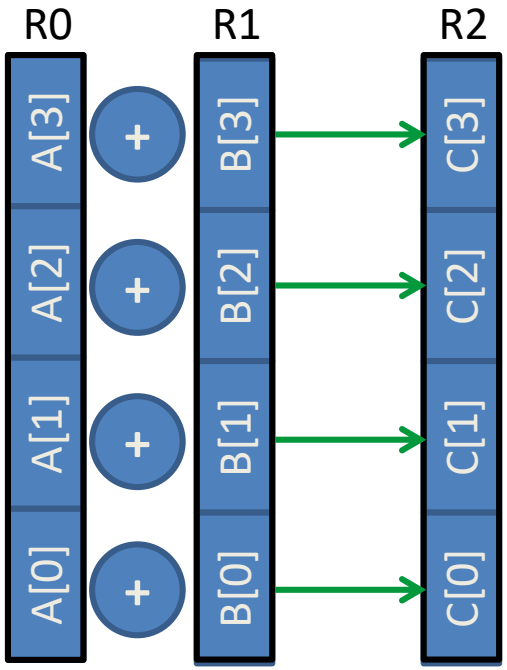
SIMD terminology

A word on terminology

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

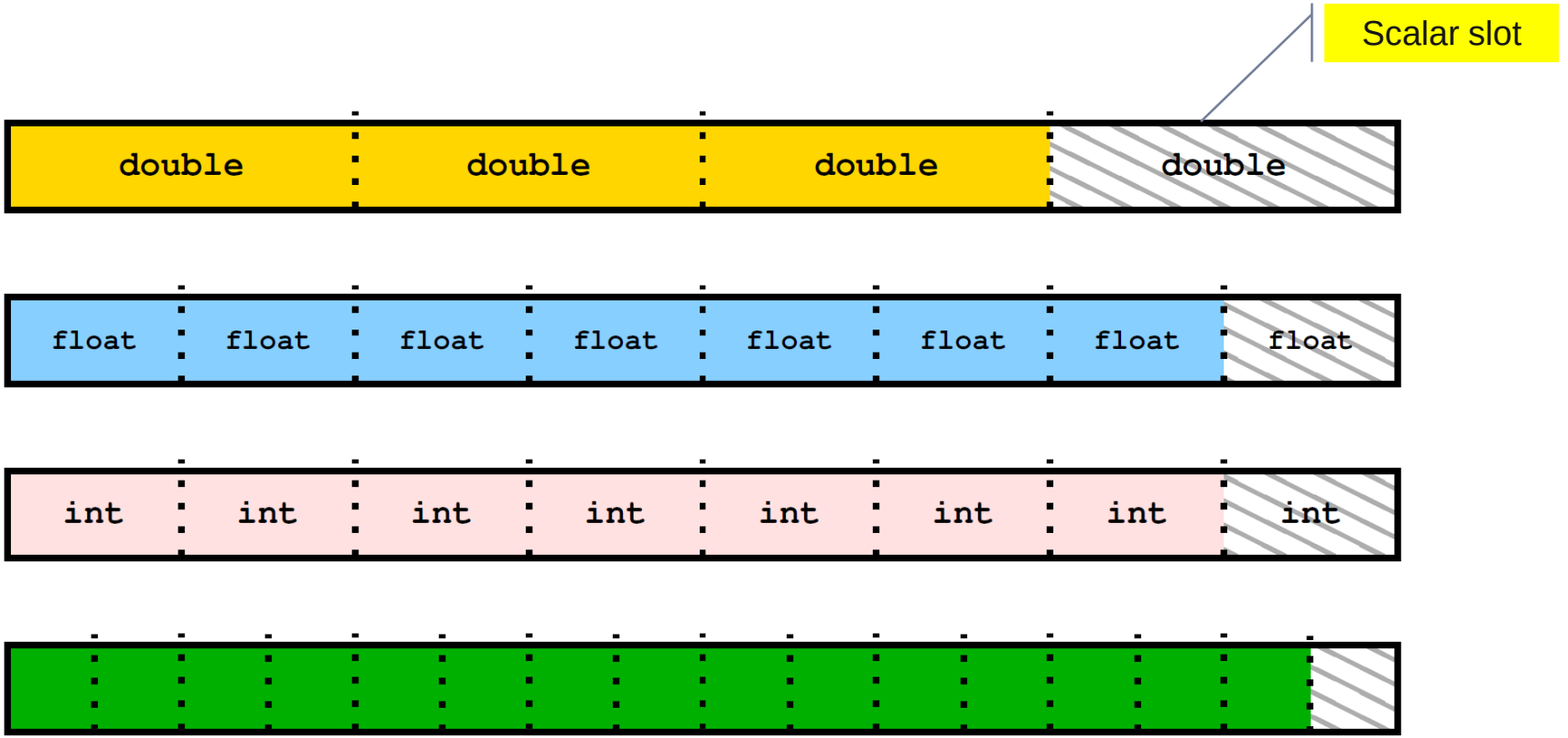
Today

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

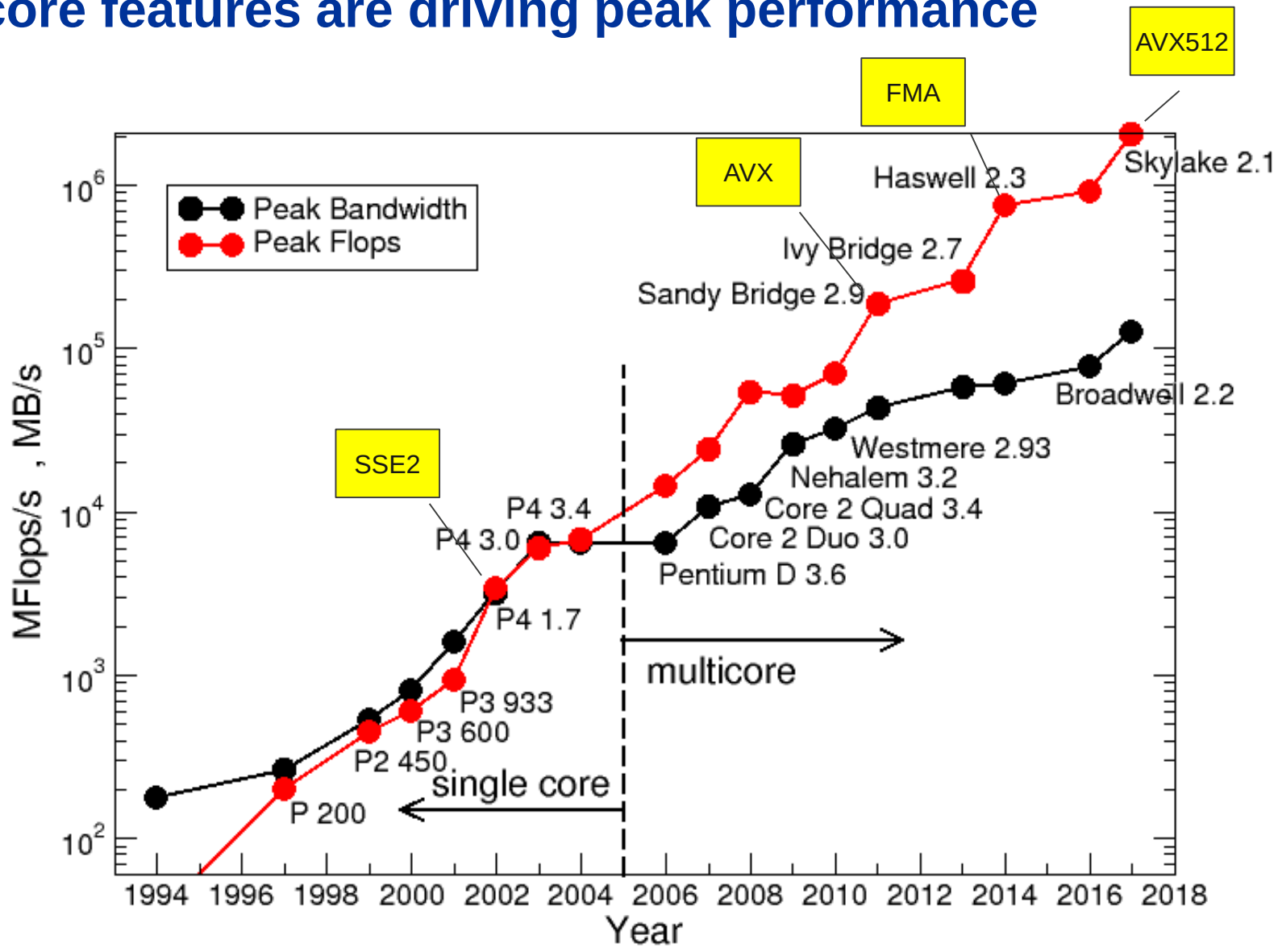


Example: Data types in 32-byte SIMD registers (AVX[2])

- Supported data types depend on actual SIMD instruction set



In-core features are driving peak performance



SIMD processing – Basics

Steps (done by the compiler) for "SIMD processing"

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

"Loop unrolling"

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

This should not be done by hand!

Load 256 Bits starting from address of `A[i]` to register `R0`

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

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

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

SIMD processing – Basics

No SIMD vectorization for loops with data dependencies:

```
for(int i=0; i<n;i++)
        A[i]=A[i-1]*s;
```

"Pointer aliasing" may prevent SIMDfication

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

C/C++ allows that **A → &C[-1]** and **B → &C[-2]**

→ **C[i] = C[i-1] + C[i-2]:** dependency → No SIMD

If "pointer aliasing" is not used, tell the compiler:

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

restrict keyword (C only!):

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

How to leverage SIMD: your options

Options:

- The compiler does it for you (but: aliasing, alignment, language, abstractions)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:

- **xmmintrin.h (SSE)**
- **pmmintrin.h (SSE2)**
- **immintrin.h (AVX)**
- **x86intrin.h (all extensions)**

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

User-mandated vectorization (OpenMP 4)

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

- Prerequesites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses:

`reduction, vectorlength, private, collapse, ...`

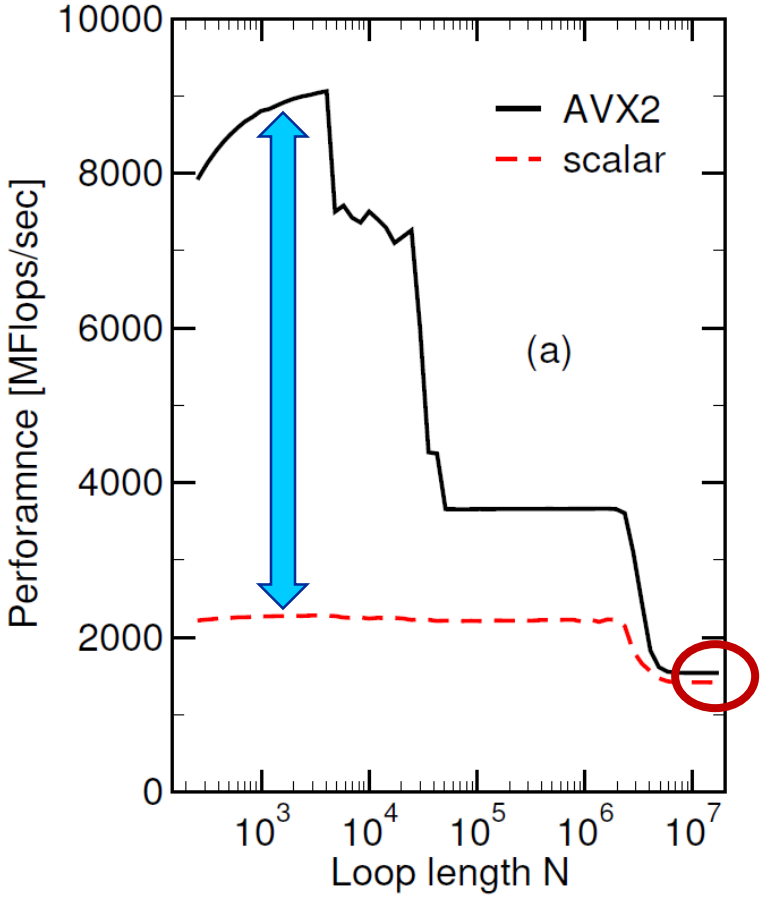
```
for (int j=0; j<n; j++) {
  #pragma omp simd reduction(+:b[j:1])
  for (int i=0; i<n; i++) {
     b[j] += a[j][i];
  }
}
```

SIMD is an in-core feature!

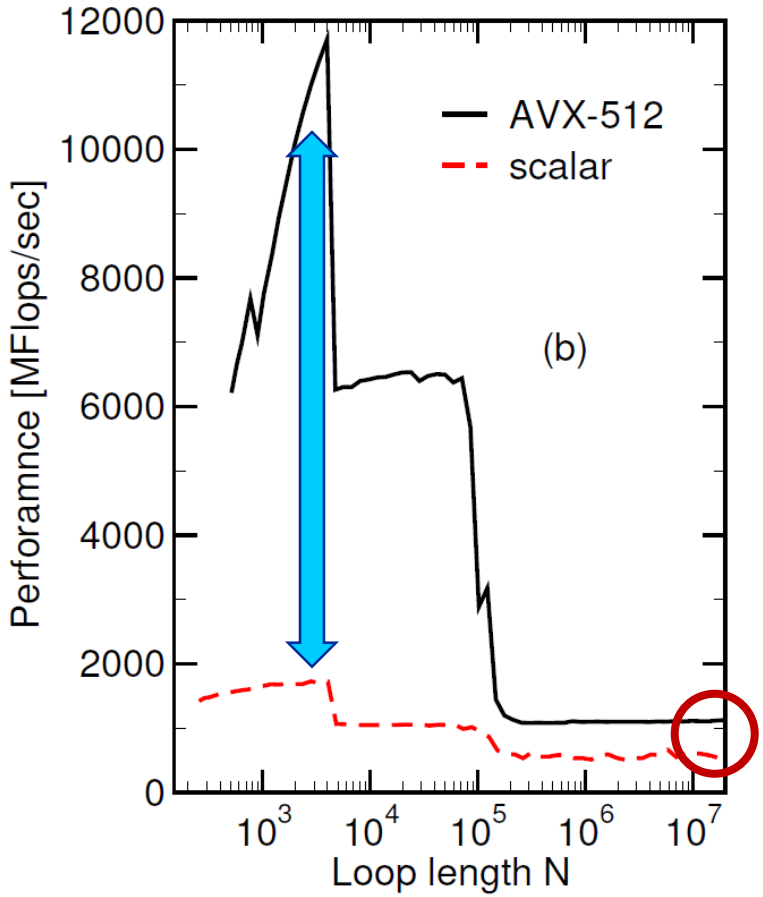
DP sum reduction (single core)

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

Intel Broadwell EP 2.3 GHz



Intel KNL 1.3 GHz



Rules for vectorizable loops

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:

1. Simple inner loops with unit stride (contiguous data access)
2. Minimize indirect addressing
3. Align data structures to SIMD width boundary
4. In C use the **`restrict`** keyword and/or **`const`** qualifiers and/or compiler options to rule out array/pointer aliasing

Efficient parallel programming on ccNUMA nodes

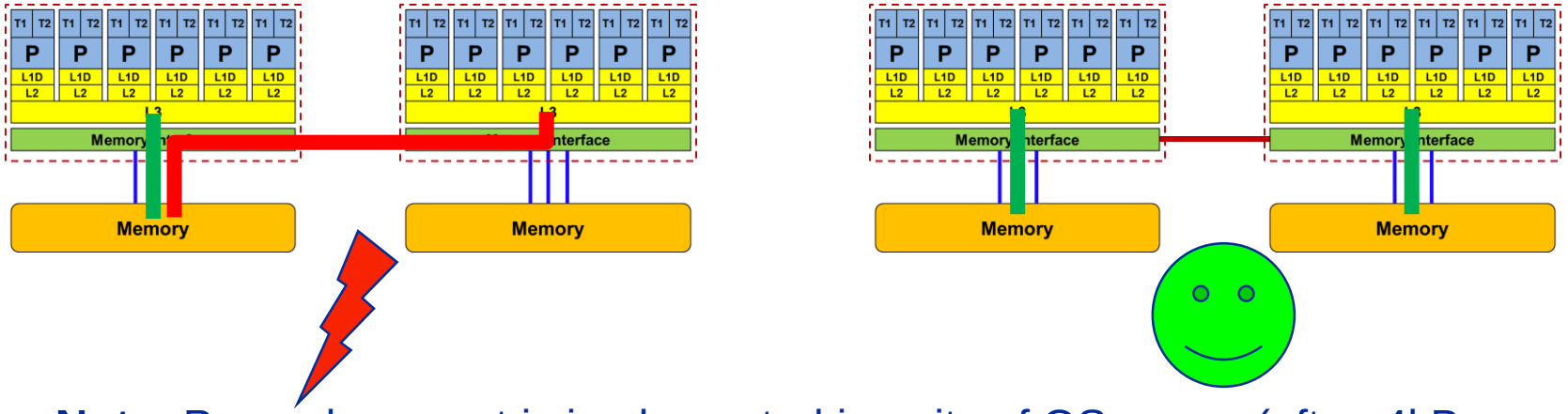
Performance characteristics of ccNUMA nodes

First touch placement policy

ccNUMA performance problems

"The other affinity" to care about

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

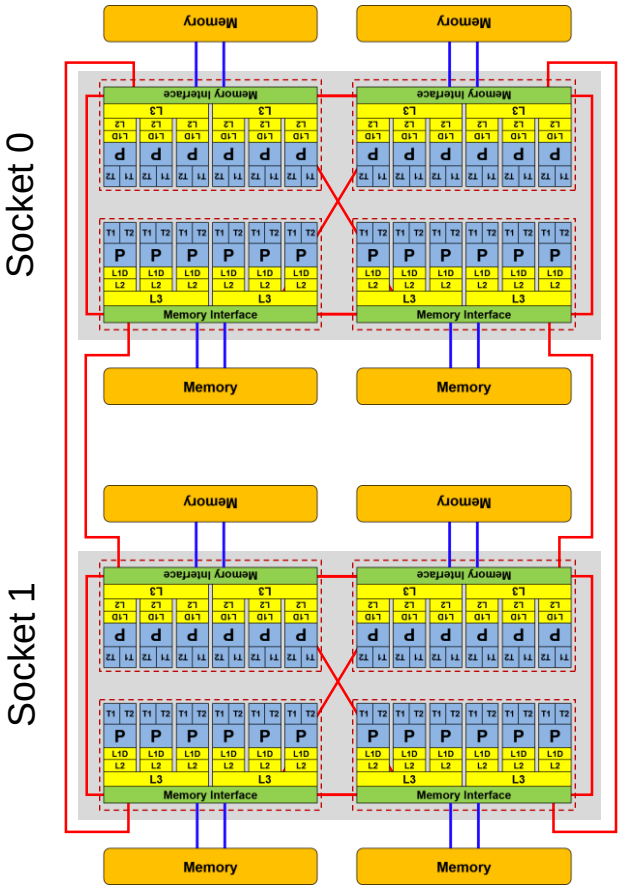


Note: Page placement is implemented in units of OS pages (often 4kB, possibly more)

How much bandwidth does nonlocal access cost?

Example: AMD “Epyc” 2-socket system (8 chips, 2 sockets, 48 cores): *STREAM Triad bandwidth measurements* [Gbyte/s]

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



numactl as a simple ccNUMA locality tool :

How do we enforce some locality of access?

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

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

- Examples:

```
for m in `seq 0 3`; do
  for c in `seq 0 3`; do
    env OMP_NUM_THREADS=8 \
        numactl --membind=$m --cpunodebind=$c ./stream
  done
done
```

ccNUMA map scan

```
env OMP_NUM_THREADS=4 numactl --interleave=0-3 \
                              likwid-pin -c N:0,4,8,12 ./stream
```

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

ccNUMA default memory locality

- **"Golden Rule" of ccNUMA:**

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

  - Except if there is not enough local memory available
  - This might be a problem, see later
- Caveat: “to touch" means “to write", not “to allocate"
- Example:

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

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

Memory not mapped here yet

Mapping takes place here

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

Coding for ccNUMA data locality

Most simple case: explicit initialization

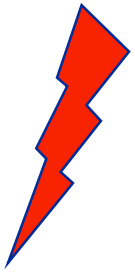
```
integer,parameter :: N=10000000
double precision A(N), B(N)



A=0.d0



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



```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

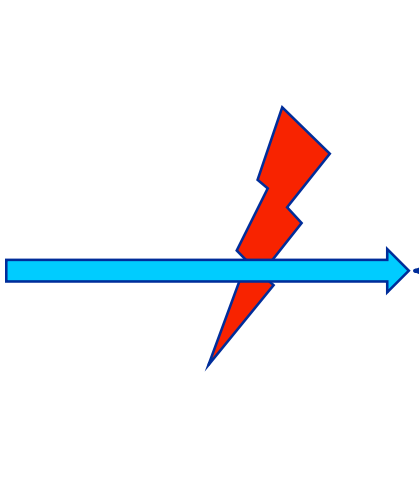
Coding for ccNUMA data locality

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

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

READ(1000) A

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



```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for Data Locality

- Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: `static`! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order
    - OpenMP 5.0 will have rudimentary memory affinity functionality
- How about global objects?
  - Better not use them
  - If communication vs. computation is favorable, might consider properly placed copies of global data
- C++: Arrays of objects and `std::vector<>` are by default initialized sequentially
  - STL allocators provide an elegant solution

Diagnosing bad locality

- If your code is cache bound, you might not notice any locality problems
- Otherwise, bad locality limits scalability
  (whenever a ccNUMA node boundary is crossed)
  - Just an indication, not a proof yet
- Running with **`numactl --interleave`** might give you a hint
  - See later
- Consider using performance counters
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel dual-socket system (IvyBridge, 2x10-core):

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

Using performance counters for diagnosing bad ccNUMA access locality

- Intel Ivy Bridge EP node (running 2x5 threads): measure NUMA traffic per core

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

- Summary output:

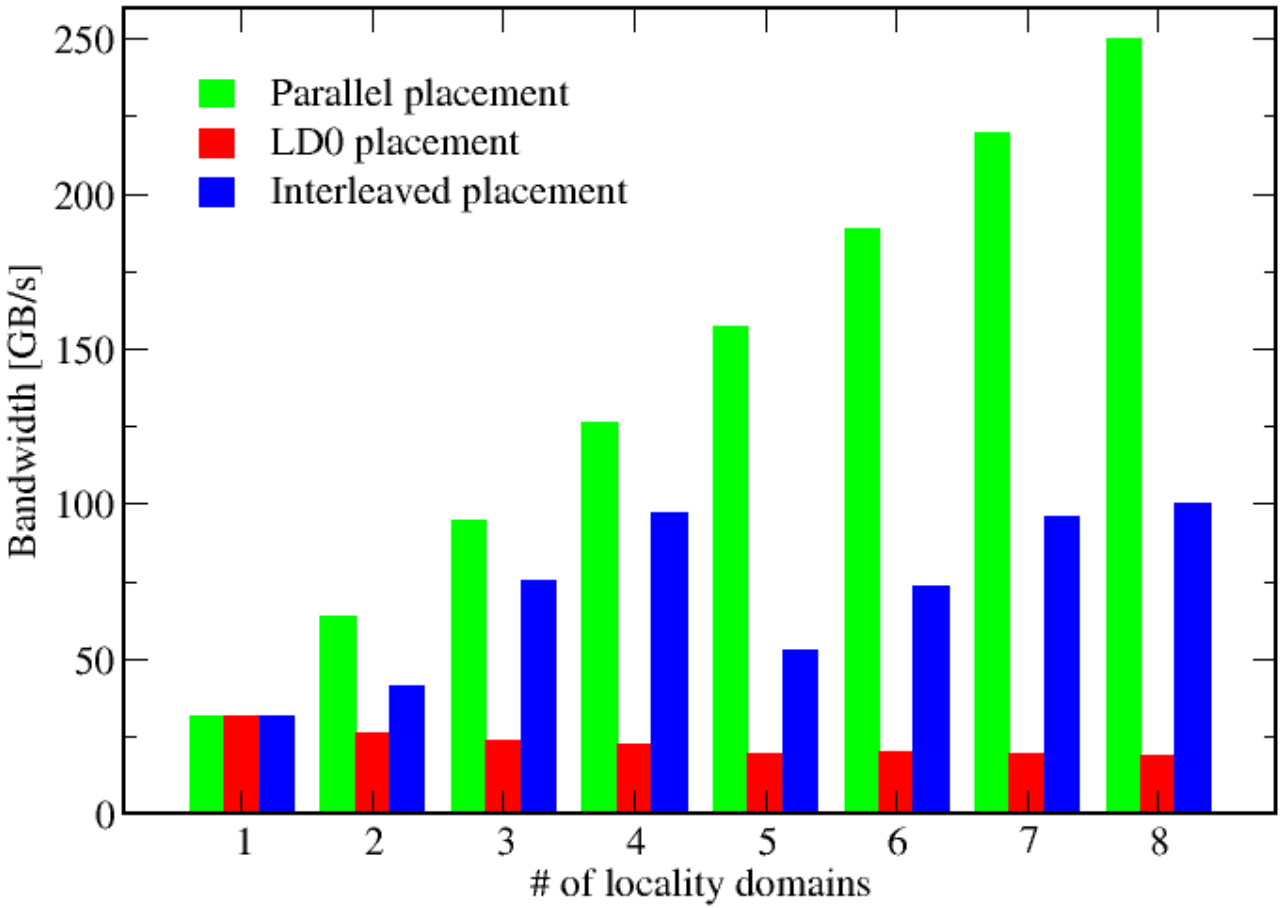
Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	4.050483	0.4050483	0.4050483	0.4050483
Runtime unhalted [s] STAT	3.03537	0.3026072	0.3043367	0.303537
Clock [MHz] STAT	32996.94	3299.692	3299.696	3299.694
CPI STAT	40.3212	3.702072	4.244213	4.03212
Local DRAM data volume [GByte] STAT	7.752933632	0.735579264	0.823551488	0.7752933632
Local DRAM bandwidth [MByte/s] STAT	19140.761	1816.028	2033.218	1914.0761
Remote DRAM data volume [GByte] STAT	9.16628352	0.86682464	0.957811776	0.916628352
Remote DRAM bandwidth [MByte/s] STAT	22630.098	2140.052	2364.685	2263.0098
Memory data volume [GByte] STAT	16.919217152	1.690376128	1.69339104	1.6919217152
Memory bandwidth [MByte/s] STAT	41770.861	4173.27	4180.714	4177.0861

- Caveat: NUMA metrics vary strongly between CPU models

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

The curse and blessing of interleaved placement:
OpenMP STREAM triad on a dual AMD Epyc 7451 (6 cores per LD)

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



Summary on ccNUMA issues

- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with **`numactl --interleave`**
- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- **NUMA balancing is active on many Linux systems today**
  - Automatic page migration
  - Slow process, may take many seconds (configurable)
  - Still a good idea to to parallel first touch
- **If dynamic scheduling cannot be avoided**
  - Consider round-robin placement as a quick (but non-ideal) fix
  - OpenMP 5.0 has some data affinity support

OpenMP performance issues on multicore

Barrier synchronization overhead

Topology dependence

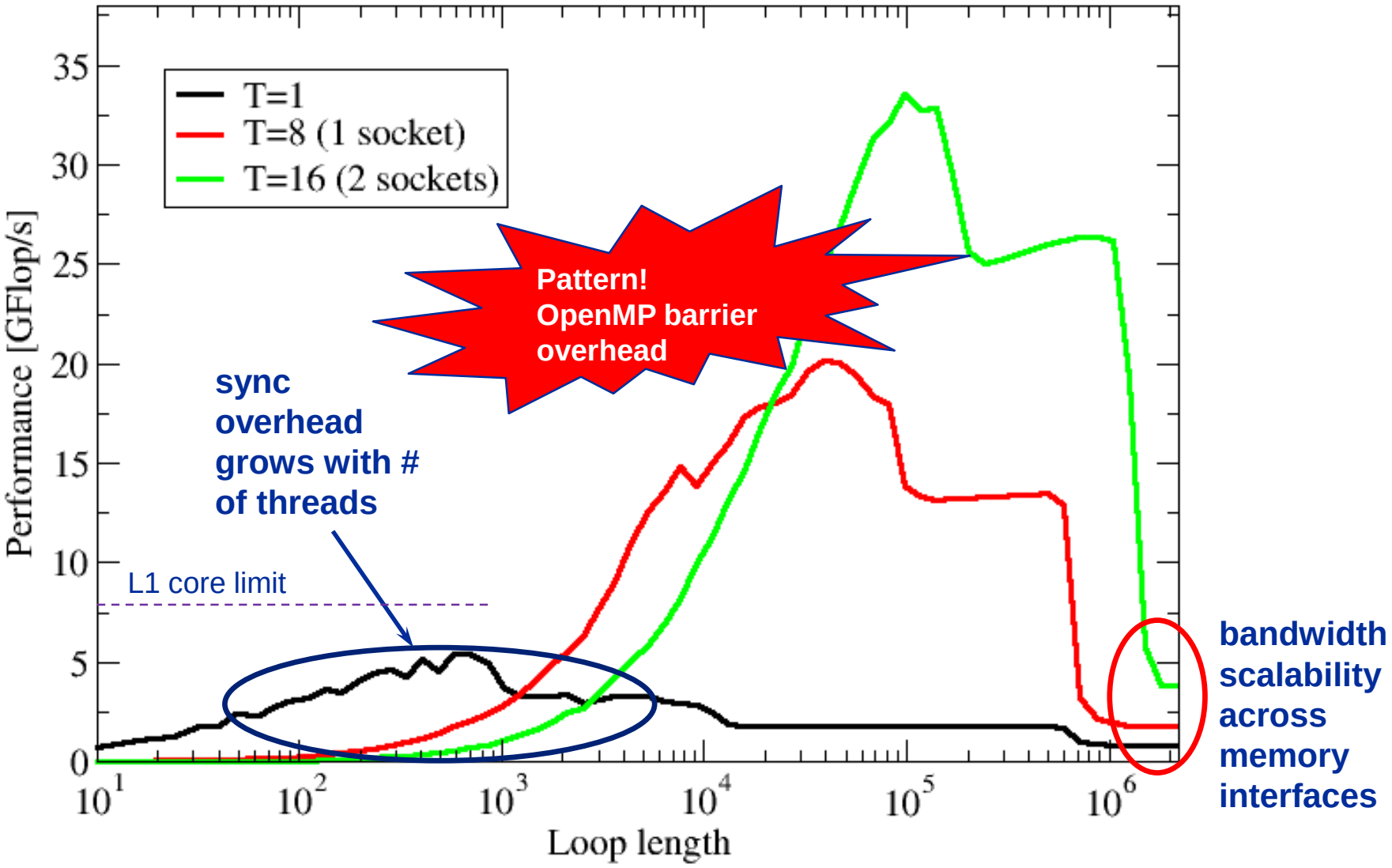
OpenMP work sharing in the benchmark loop

```
double precision, dimension(:), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
!$OMP PARALLEL private(i,j)
do j=1,NITER
!$OMP DO
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP END DO
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL
```

Implicit barrier

OpenMP vector triad on Sandy Bridge sockets (3 GHz)



```
!$OMP PARALLEL …
…
!$OMP BARRIER
!$OMP DO
…
!$OMP ENDDO
!$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP progams.

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization!

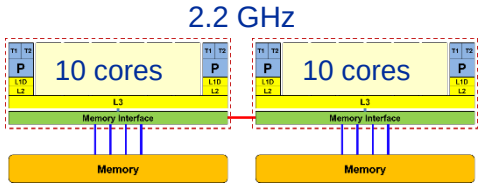
- **Next slides: Test OpenMP Barrier performance…**
- **for different compilers**
- **and different topologies:**
  - **shared cache**
  - **shared socket**
  - **between sockets**
- **and different thread counts**
  - **2 threads**
  - **full domain (chip, socket, node)**

Thread synchronization overhead on IvyBridge-EP

Barrier overhead in CPU cycles

2 Threads	Intel 16.0	GCC 5.3.0
Shared L3	**599**	**425**
SMT threads	**612**	**423**
Other socket	**1486**	**1067**

Strong topology dependence!



Full domain	Intel 16.0	GCC 5.3.0
Socket (10 cores)	1934	1301
Node (20 cores)	**4999**	**7783**
Node +SMT	5981	9897

Overhead grows with thread count

- **Strong dependence on compiler, CPU and system environment!**
- `OMP_WAIT_POLICY=ACTIVE` **can make a big difference**

Thread synchronization overhead on Xeon Phi 7210 (64-core)

Barrier overhead in CPU cycles (Intel C compiler 16.03)

2 threads on distinct cores: 730

	SMT1	SMT2	SMT3	SMT4
One core	n/a	**963**	1580	2240
Full chip	5720	8100	9900	**11400**

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Ivy Bridge node

3.2x cores (20 vs 64) on Phi

4x more operations per cycle per core on Phi

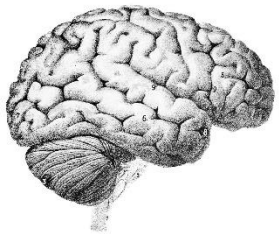
→ 4 · 3.2 = 12.8x more work done on Xeon Phi per cycle

1.9x more barrier penalty (cycles) on Phi (11400 vs. 6000)

→ One barrier causes 1.9 · 12.8 ≈ 24x more pain ☺.

Tutorial conclusion

- **Multicore architecture == multiple complexities**
  - Affinity matters → pinning/binding is essential
  - Bandwidth bottlenecks → inefficiency is often made on the chip level
  - Topology dependence of performance features → know your hardware!
- **Put cores to good use**
  - Bandwidth bottlenecks → surplus cores → functional parallelism!?
  - Shared caches → fast communication/synchronization → better implementations/algorithms?
- **Simple modeling techniques and patterns help us**
  - … understand the limits of our code on the given hardware
  - … identify optimization opportunities
  - … learn more, especially when they do not work!
- **Simple tools get you 95% of the way**
  - e.g., with the LIKWID tool suite



Moritz Kreutzer
Markus Wittmann
Thomas Zeiser
Michael Meier
Holger Stengel
Thomas Gruber
Faisal Shahzad
Christie L. Alappat
Ayesha Afzal
Dominik Ernst
Julian Hammer
Jan Laukemann

THANK YOU.

Presenter Biographies

Georg Hager holds a PhD in computational physics from the University of Greifswald. He is a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is required or recommended reading in many HPC-related courses around the world. See his blog at https://blogs.fau.de/hager for current activities, publications, and talks.

Jan Eitzinger (formerly Treibig) holds a PhD in Computer Science from the University of Erlangen. He is now a postdoctoral researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE). His current research revolves around architecture-specific and low-level optimization for current processor architectures, performance modeling on processor and system levels, and programming tools. He is the developer of LIKWID, a collection of lightweight performance tools. In his daily work he is involved in all aspects of user support in High Performance Computing: training, code parallelization, profiling and optimization, and the evaluation of novel computer architectures.

Gerhard Wellein holds a PhD in solid state physics from the University of Bayreuth and is a professor at the Department for Computer Science at the University of Erlangen. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.

Abstract

- **SC19 full-day tutorial: Node-Level Performance Engineering**
- **Presenter(s): Georg Hager, Gerhard Wellein**

- **ABSTRACT:**

The advent of multi- and manycore chips has led to a further opening of the gap between peak and application performance for many scientific codes. This trend is accelerating as we move from petascale to exascale. Paradoxically, bad node-level performance helps to "efficiently" scale to massive parallelism, but at the price of increased overall time to solution. If the user cares about time to solution on any scale, optimal performance on the node level is often the key factor. We convey the architectural features of current processor chips, multiprocessor nodes, and accelerators, as far as they are relevant for the practitioner. Peculiarities like SIMD vectorization, shared vs. separate caches, bandwidth bottlenecks, and ccNUMA characteristics are introduced, and the influence of system topology and affinity on the performance of typical parallel programming constructs is demonstrated. *Performance engineering* and *performance patterns* are suggested as powerful tools that help the user understand the bottlenecks at hand and to assess the impact of possible code optimizations. A cornerstone of these concepts is the roofline model, which is described in detail, including useful case studies, limits of its applicability, and possible refinements.