For slides and example code see:

http://goo.gl/wP2yYF

qrme.com

Node-Level Performance Engineering

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

Full-day tutorial
International Workshop “Quantum Dynamics: From Algorithms to Applications”
September 8, 2016
Institut für Physik, University of Greifswald, Germany

Agenda

GW

- **Preliminaries**
- **Introduction to multicore architecture**
  - Threads, cores, SIMD, caches, chips, sockets, ccNUMA

GH

- **Multicore tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks
  - Hardware bottlenecks
- **Node-level performance modeling (part I)**
  - The Roofline Model and dense MVM
- **Lunch break**

GW

- **Node-level performance modeling (part II)**
  - Case studies: Sparse MVM, Jacobi solver
- **Optimal resource utilization**
  - SIMD parallelism

GH

  - ccNUMA
  - OpenMP synchronization and multicores

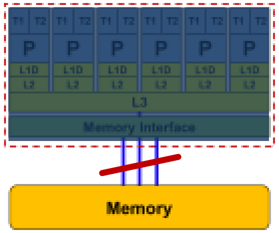
Quiz

- **What does "clock frequency" mean in computers?**

  The "heartbeat" of the CPU. A clock cycle is the smallest unit of time on a CPU chip. Typically < 1ns → $f \gtrsim 1\ \mathrm{GHz}$

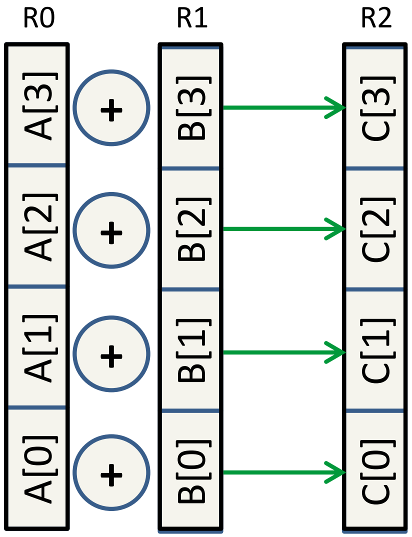
- **What is "memory bandwidth"?**

  Rate of data transfer between main memory (RAM) and CPU chip. Typical $b_S \approx 10 \ldots 100\ \mathrm{GB/s}$

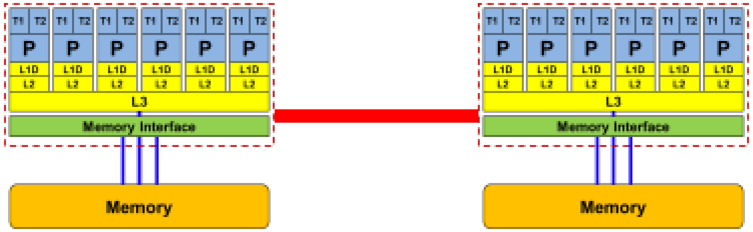


- **What is SIMD vectorization?**

  Single **I**nstruction **M**ultiple **D**ata. Data-parallel load/store and execution units.



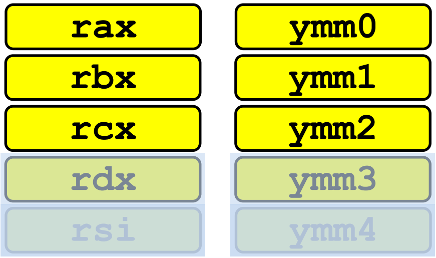
- **What is ccNUMA?**



Quiz

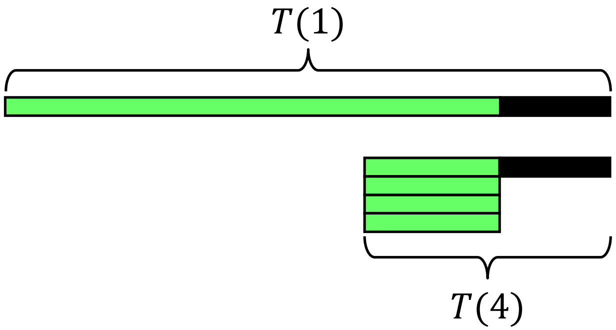
- **What is a register?**

A storage unit in the CPU core that can
take one single value (a few values in case of SIMD).
Operands for computations reside in registers.


rax ymm0
rbx ymm1
rcx ymm2
rdx ymm3
rsi ymm4

- **What is Amdahl’s Law?**

$$S_p = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N}}$$


$T(1)$

$T(4)$

- **What is a pipelined functional unit?**

An instruction execution unit on the core that executes
a certain task in several simple sub-steps.
The stages of the pipeline can act in parallel
on several instructions at once.

A conversation

From a student seminar on “Efficient programming of modern multi- and manycore processors”

Student: I have implemented this algorithm on the GPGPU, and it solves a system with 26546 unknowns in 0.12 seconds, so it is really fast.

Me: What makes you think that 0.12 seconds is fast?

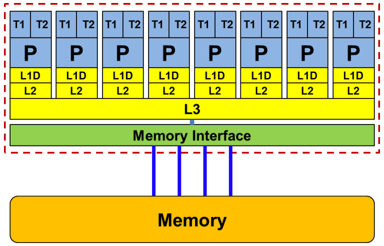
Student: It is fast because my baseline C++ code on the CPU is about 20 times slower.

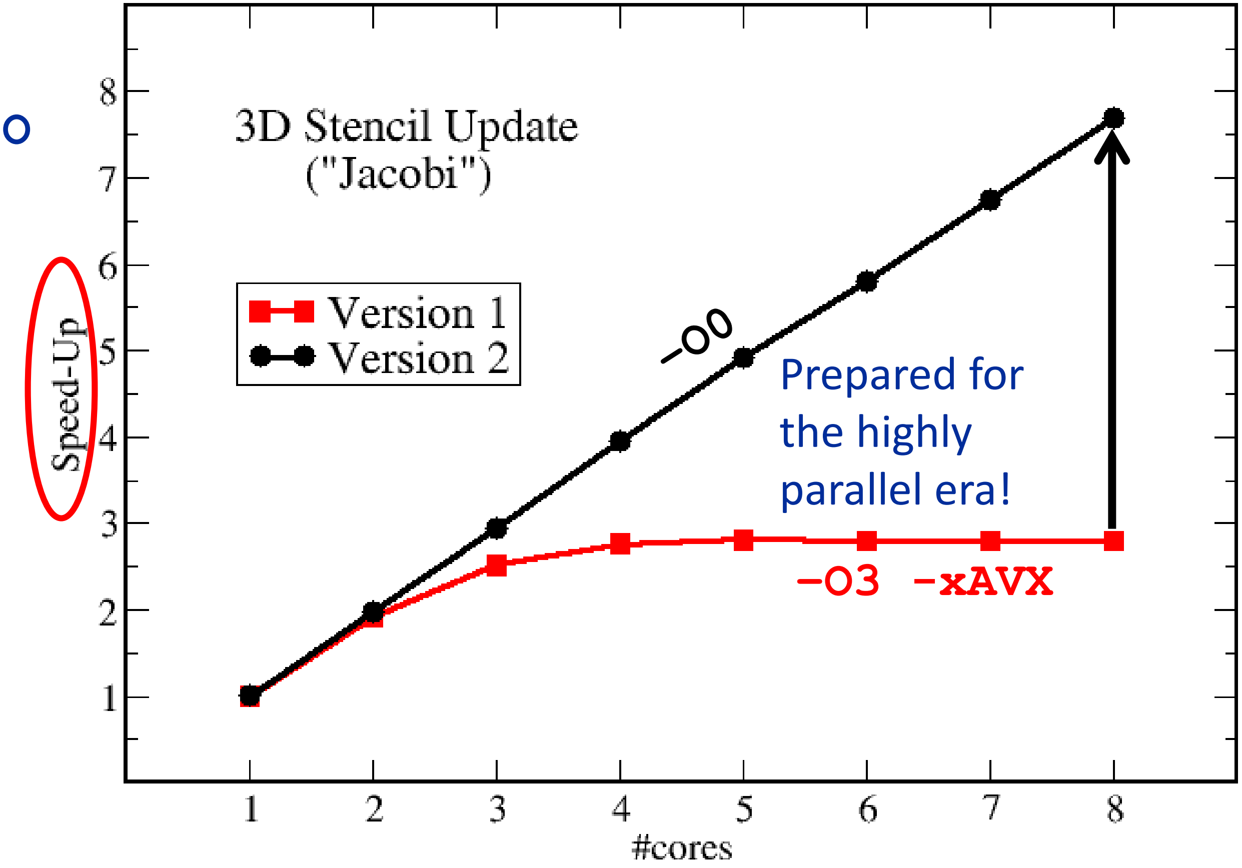
Prelude:
Scalability 4 the win!

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



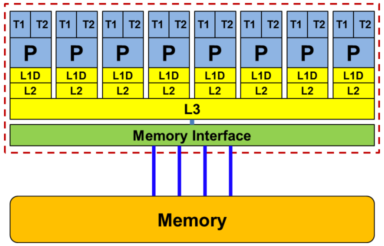


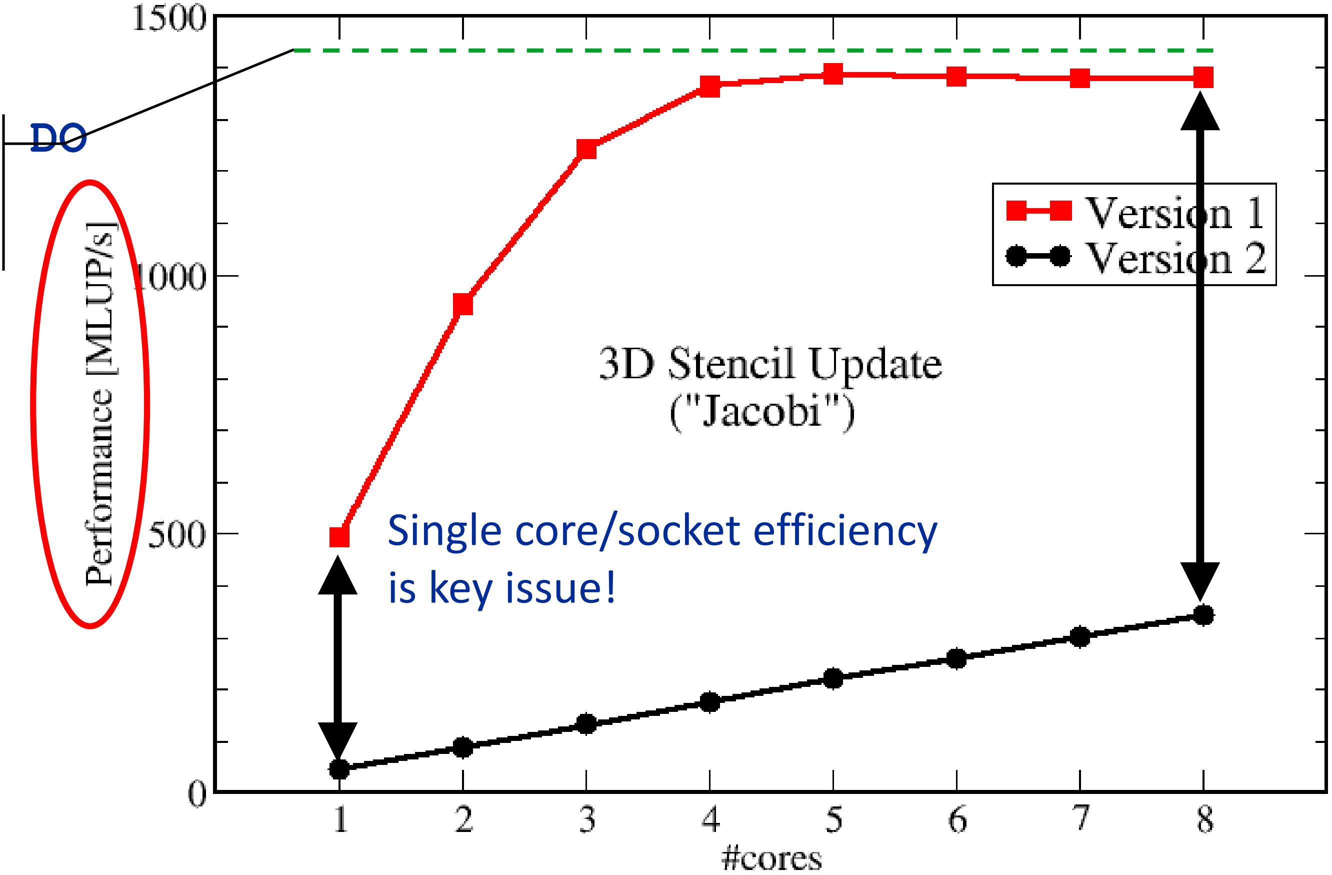
Prepared for the highly parallel era!

Scalability Myth: Code scalability is the key issue

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

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





Single core/socket efficiency is key issue!

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

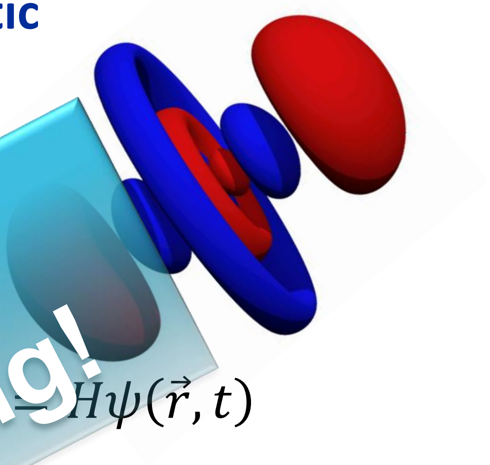
How model-building works: Physics

Newtonian mechanics

$$\vec{F} = m\vec{a}$$

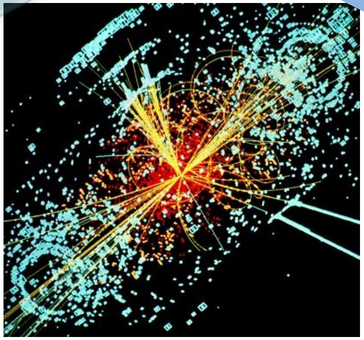
Fails @ small scales!

Nonrelativistic quantum mechanics



$$i\hbar\frac{\partial}{\partial t}\psi(\vec{r},t) = H\psi(\vec{r},t)$$

Fails @ even smaller scales!

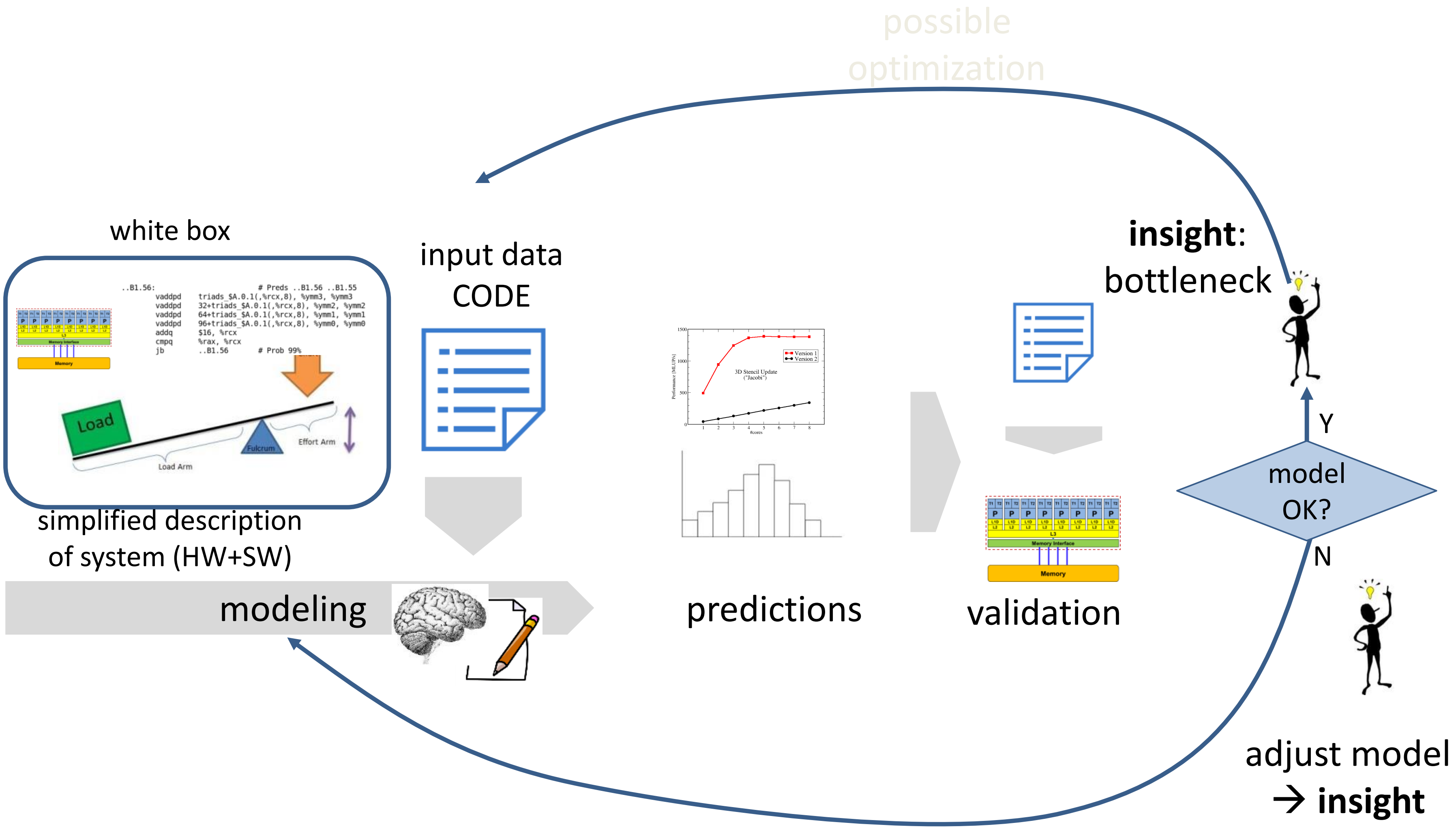


Relativistic quantum field theory

$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_c$$

If a model fails, we learn something!

Performance Engineering – no black boxes!



possible optimization

white box

```
..B1.56:                            # Preds ..B1.56 ..B1.55
        vaddpd    triads_$A.0.1(,%rcx,8), %ymm3, %ymm3
        vaddpd    32+triads_$A.0.1(,%rcx,8), %ymm2, %ymm2
        vaddpd    64+triads_$A.0.1(,%rcx,8), %ymm1, %ymm1
        vaddpd    96+triads_$A.0.1(,%rcx,8), %ymm0, %ymm0
        addq      $16, %rcx
        cmpq      %rax, %rcx
        jb        ..B1.56           # Prob 99%
```

simplified description of system (HW+SW)

input data CODE

modeling

predictions

validation

insight: bottleneck

model OK?

Y

N

adjust model → **insight**

Introduction:
Modern node architecture

A glance at basic core features:

pipelining, superscalarity, SMT

Caches and data transfers through the memory hierarchy

Accelerators

Bottlenecks & hardware-software interaction

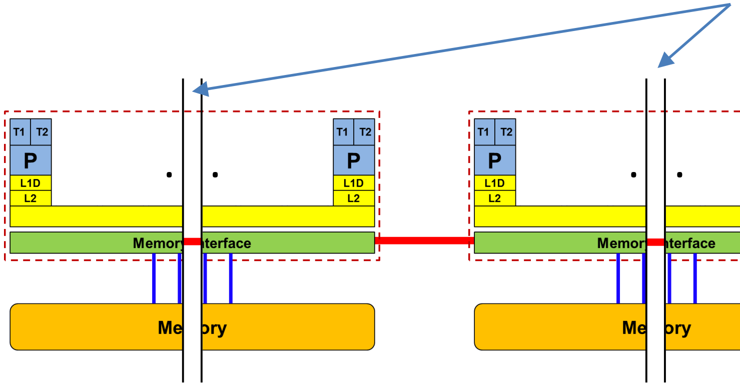
Multi-core today: Intel Xeon 2600v3 (2014)

- Xeon E5-2600v3 “Haswell EP”:
  Up to 18 cores running at 2+ GHz (+ “Turbo Mode”: 3.5+ GHz)

- Simultaneous Multithreading
  → reports as 36-way chip

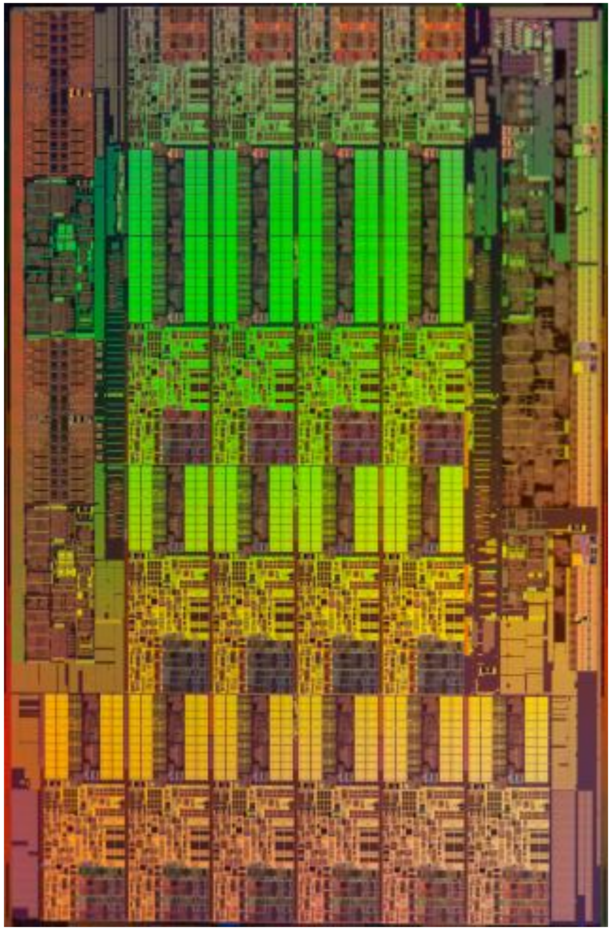
- 5.7 Billion Transistors / 22 nm

- Die size: 662 mm²

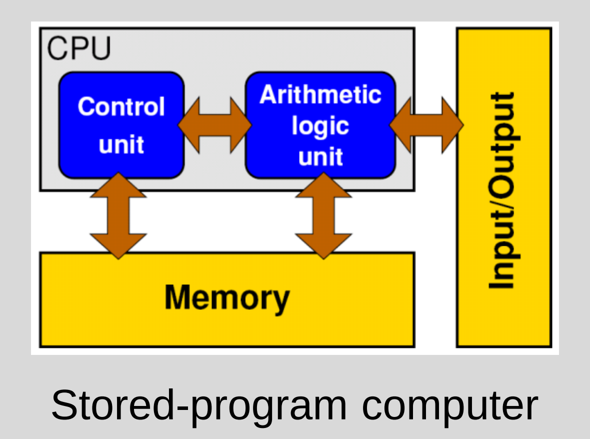


Optional:
“Cluster on Die”
(CoD) mode

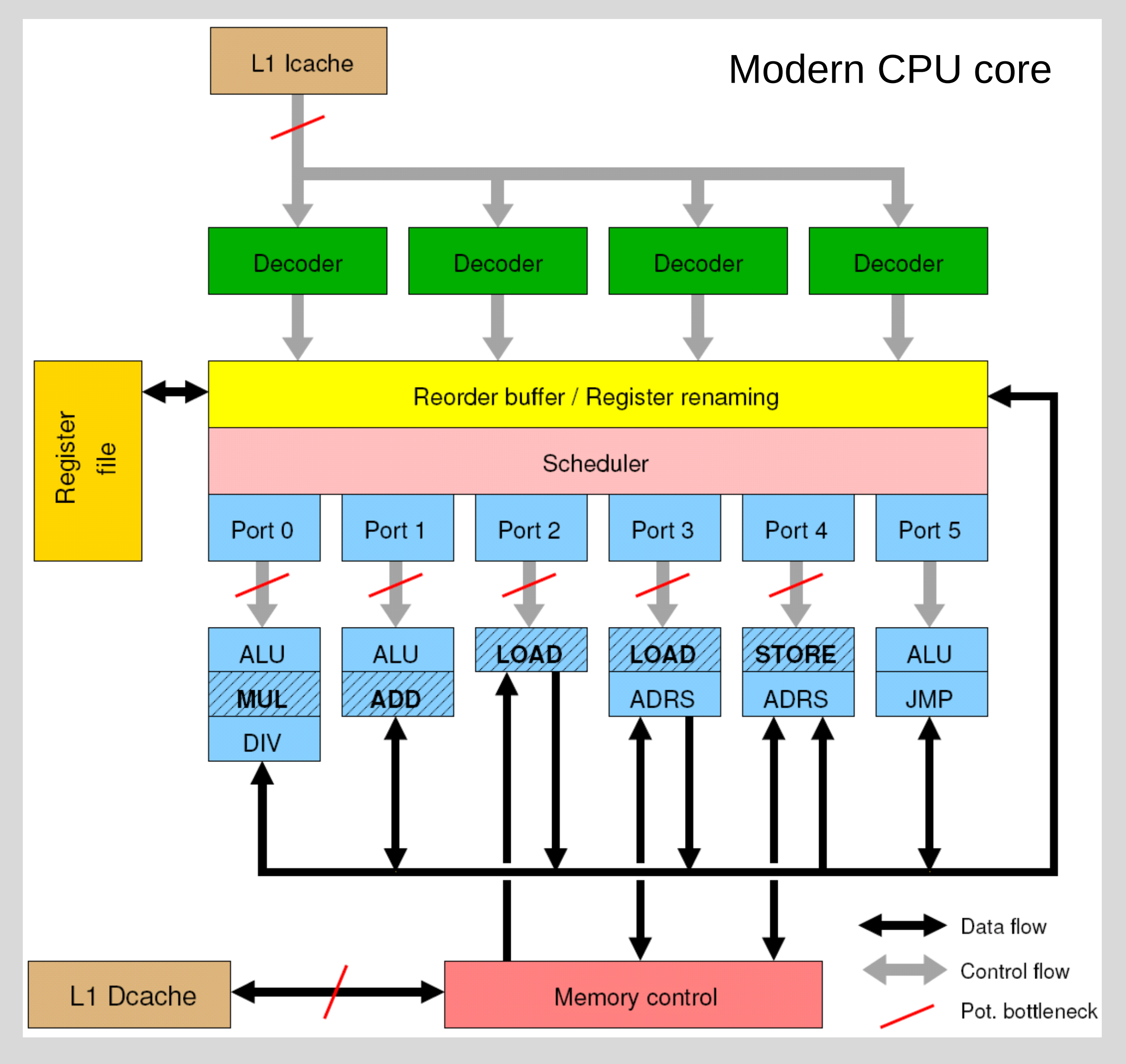
2-socket server



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

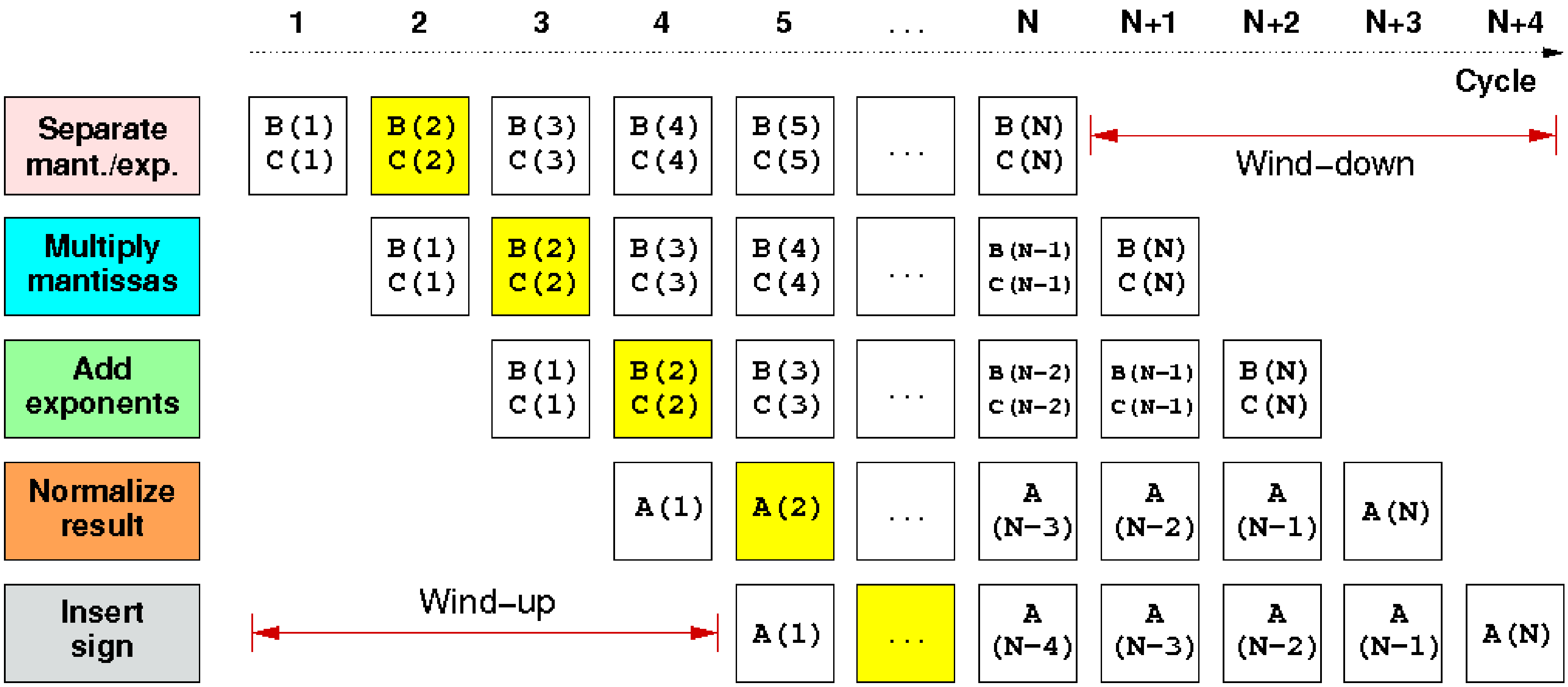
- **Idea:**
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

- **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 - startup times (#Instructions >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order

- **Pipelining is widely used in modern computer architectures**

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



	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4

Separate mant./exp.	B(1)
	C(1)
		B(2)
		C(2)
			B(3)
			C(3)
				B(4)
				C(4)
					B(5)
					C(5)
						...	B(N)
							C(N)				

Multiply mantissas		B(1)
		C(1)
			B(2)
			C(2)
				B(3)
				C(3)
					B(4)
					C(4)
						...	B(N-1)
							C(N-1)
								B(N)
								C(N)			

Add exponents			B(1)
			C(1)
				B(2)
				C(2)
					B(3)
					C(3)
						...	B(N-2)
							C(N-2)
								B(N-1)
								C(N-1)
									B(N)
									C(N)		

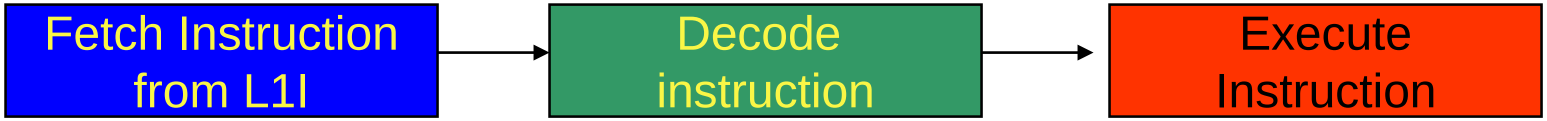
Normalize result				A(1)	A(2)	...	A(N-3)	A(N-2)	A(N-1)	A(N)	
Insert sign					A(1)	...	A(N-4)	A(N-3)	A(N-2)	A(N-1)	A(N)

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

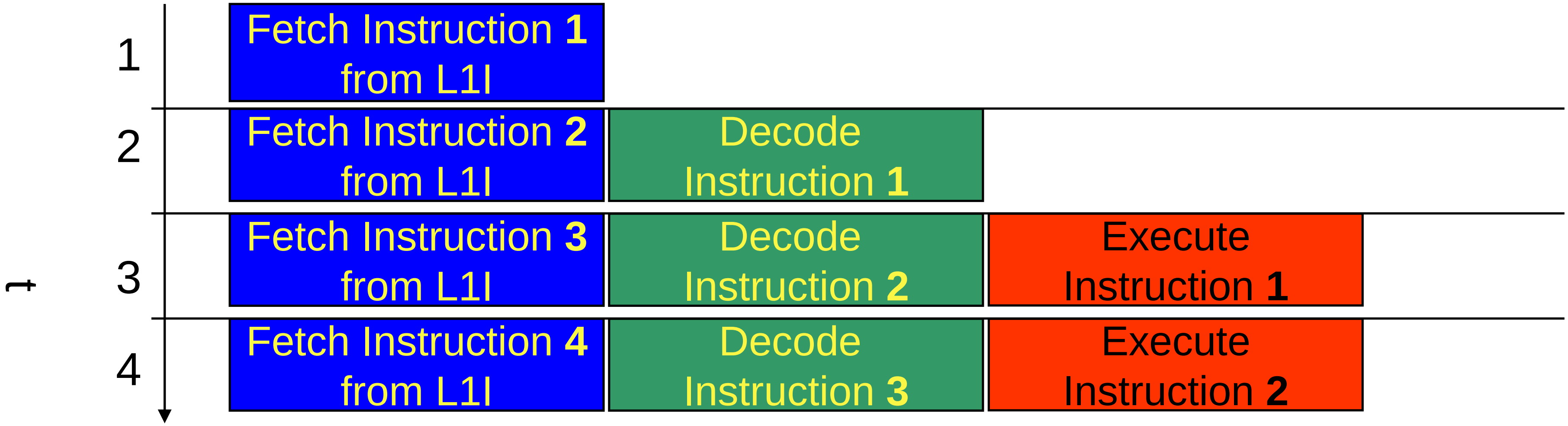
Wind-up/-down phases: Empty pipeline stages

Pipelining: The Instruction pipeline

- **Besides arithmetic & functional unit, instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:**



Hardware Pipelining on processor (all units can run concurrently):



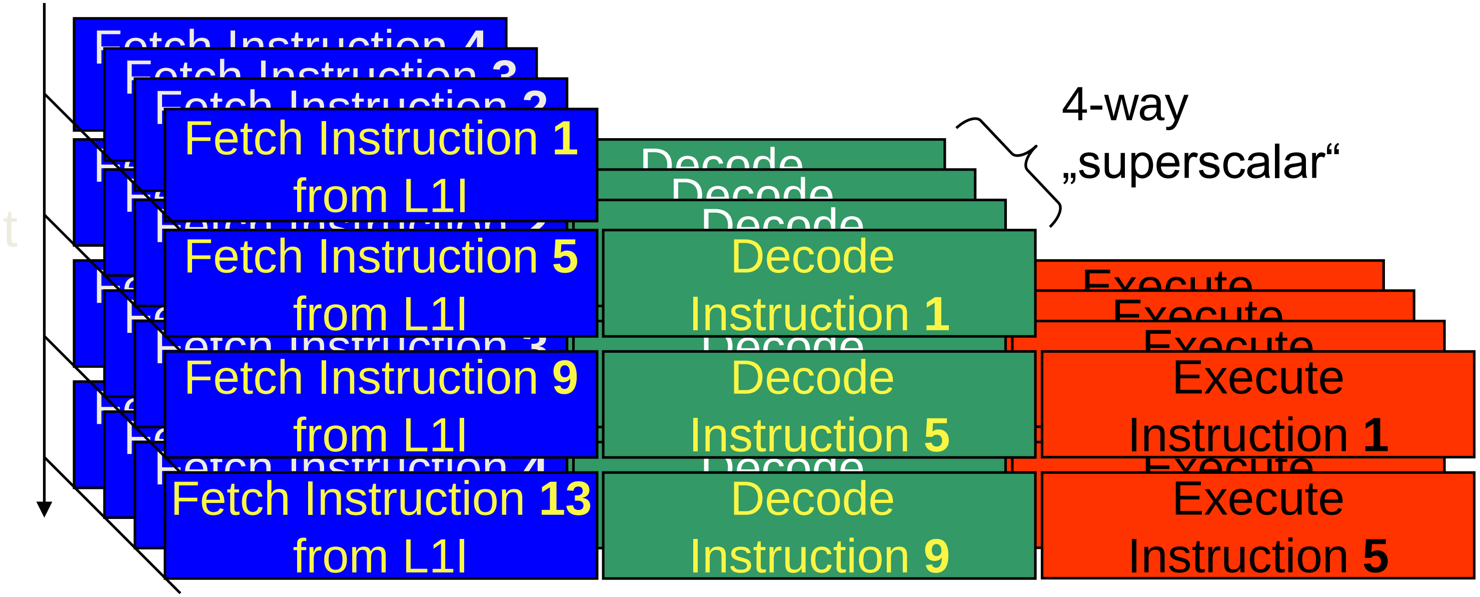
1	Fetch Instruction **1** from L1I		
2	Fetch Instruction **2** from L1I	Decode Instruction **1**	
3	Fetch Instruction **3** from L1I	Decode Instruction **2**	Execute Instruction **1**
4	Fetch Instruction **4** from L1I	Decode Instruction **3**	Execute Instruction **2**

…

- Branches can stall this pipeline! (Speculative Execution, Predication)
- Each unit is pipelined itself (e.g., Execute = Multiply Pipeline)

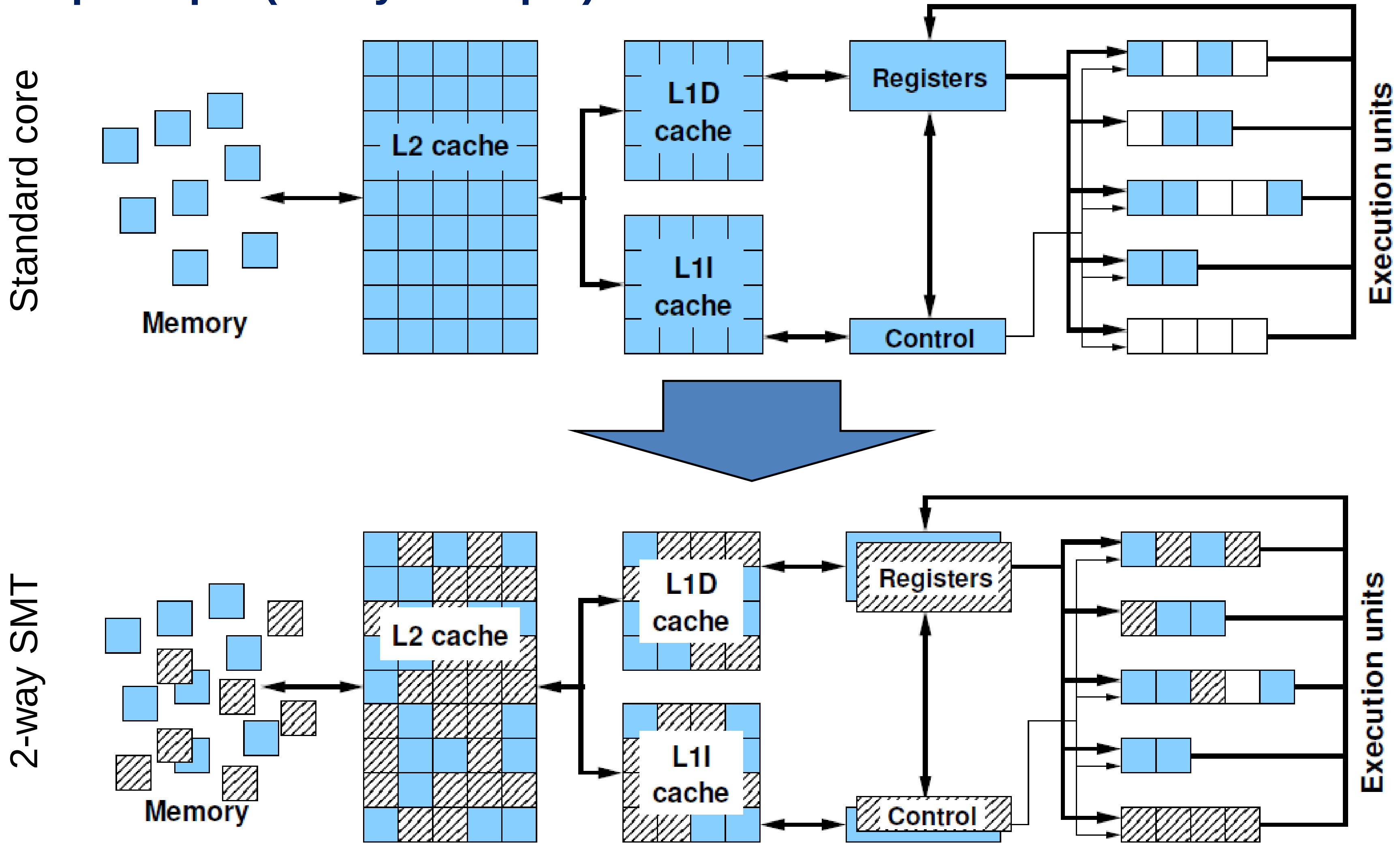
Superscalar Processors – Instruction Level Parallelism

- Multiple units enable use of Instrucion Level Parallelism (ILP):
  Instruction stream is “parallelized” on the fly



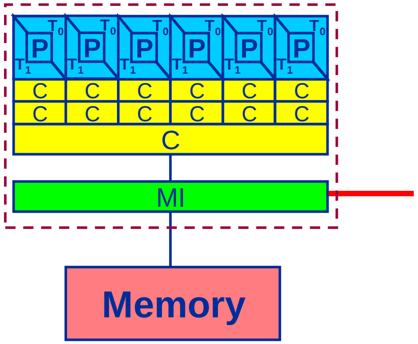
- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar &
  can perform 2 or 4 floating point operations per cycles

SMT principle (2-way example):



SMT impact

- SMT adds another layer of topology (inside the physical core)
- Caveat: SMT threads share all caches!
- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread’s executing instructions:



Thread 0:

```
do i=1,N
  a(i) = a(i-1)*c
enddo
```

Thread 1:

```
do i=1,N
  b(i) = s*b(i-2)+d
enddo
```

Dependency → pipeline stalls until previous MULT is over

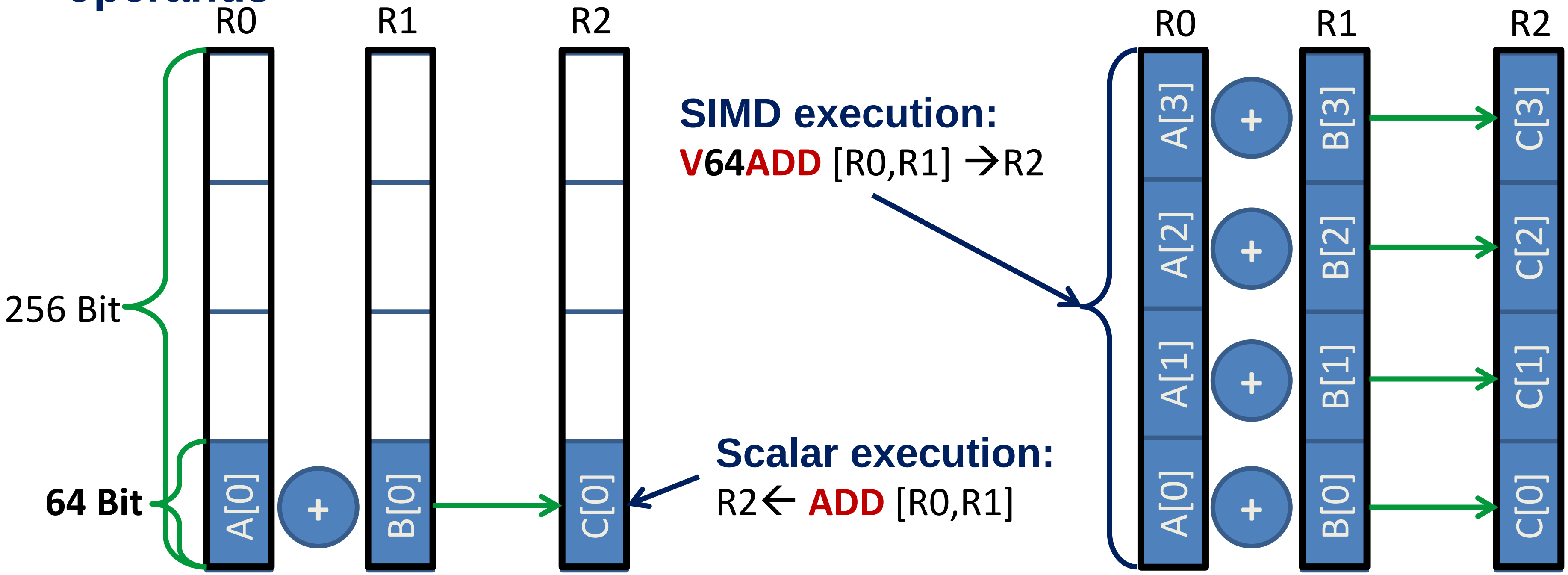
Unrelated work in other thread can fill the pipeline bubbles

- Beware: Executing it all in a single thread (if possible) may achieve the same goal without SMT:

```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = s*b(i-2)+d
enddo
```

Core details: SIMD processing

- **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
- **Adding two registers holding double precision floating point operands**

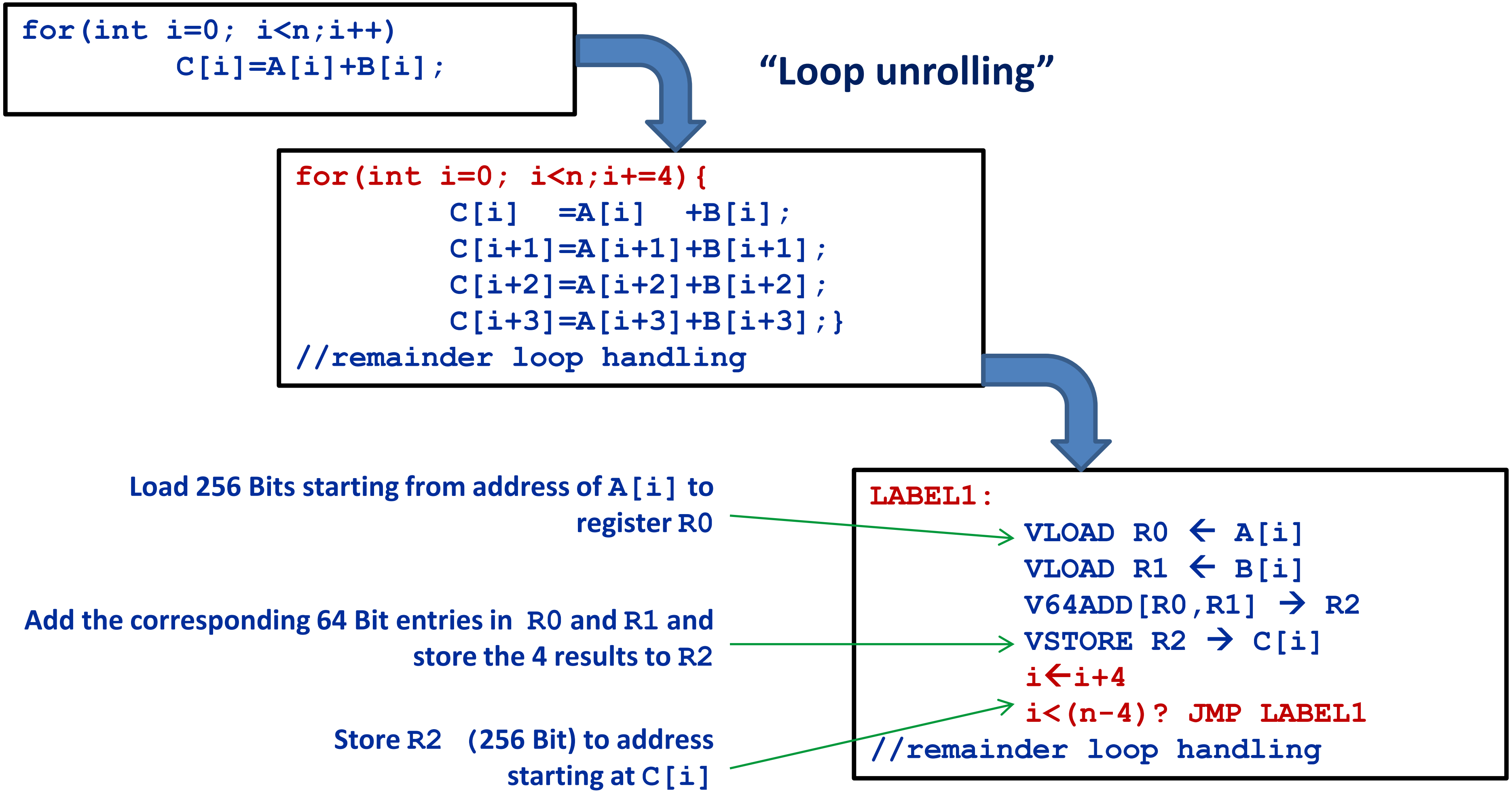


SIMD execution:
V64ADD [R0,R1] →R2

Scalar execution:
R2← ADD [R0,R1]

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
```

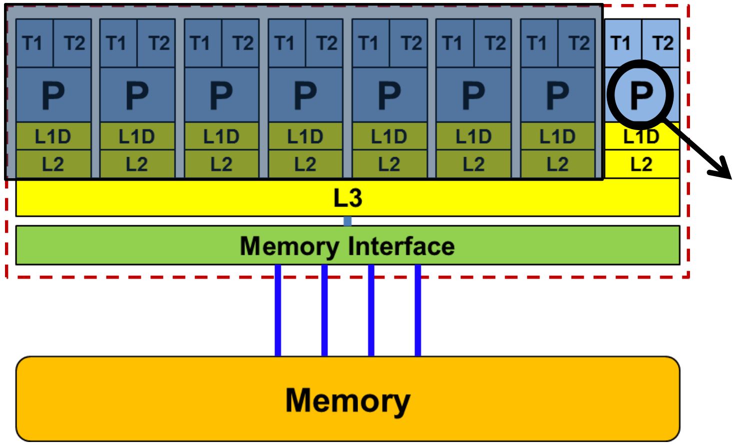
```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop handling
```

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]

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

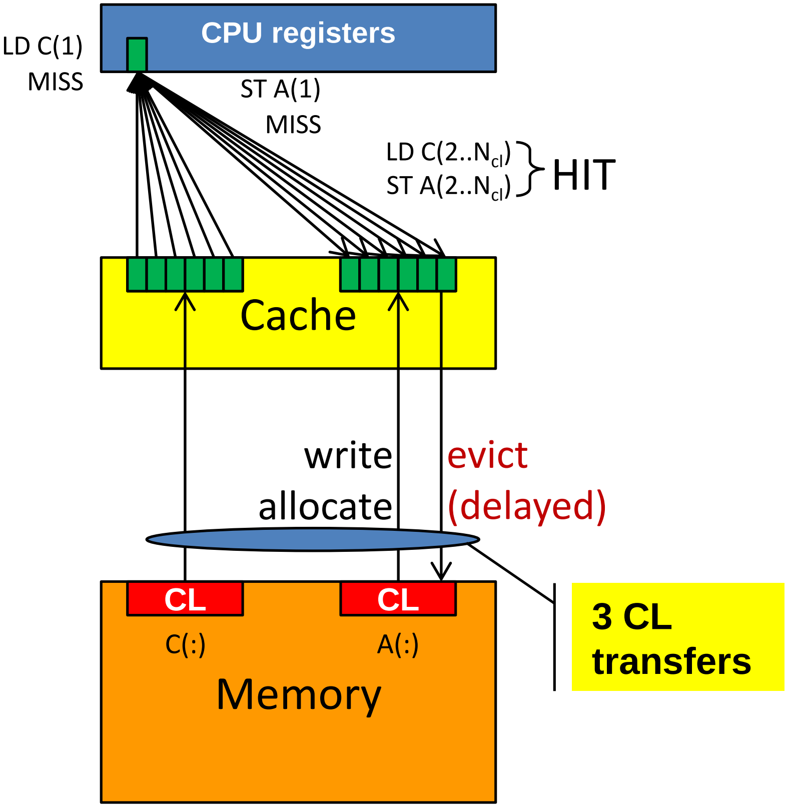
Clock Speed

Typical representatives	n_{super}^{FP} inst./cy	n_{FMA}	n_{SIMD} ops/inst.		Code	f [GHz]	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
IBM POWER8	2	2	2	Q2/2014	S822LC	2.93	23.4

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

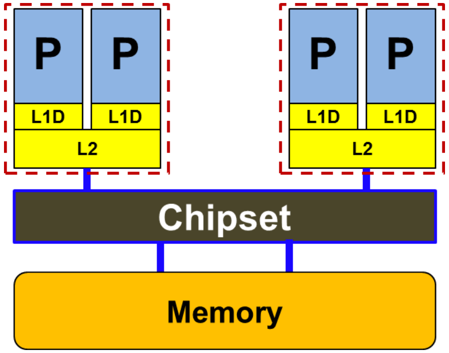
- Example: Array copy `A(:)=C(:)`



Commodity cluster nodes: From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes

Yesterday (2006): UMA



2-socket server

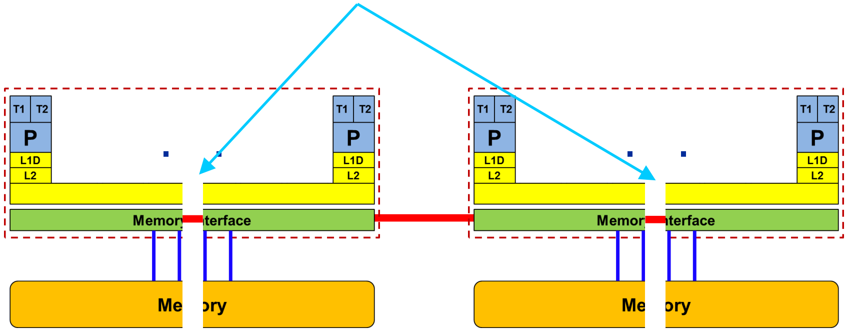
Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system “anisotropy”

Haswell(++):
“Cluster on Die”
(CoD) mode

Today: ccNUMA



2-socket server

Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

Two or more NUMA domains per node

ccNUMA provides scalable bandwidth but:
Where does my data finally end up?

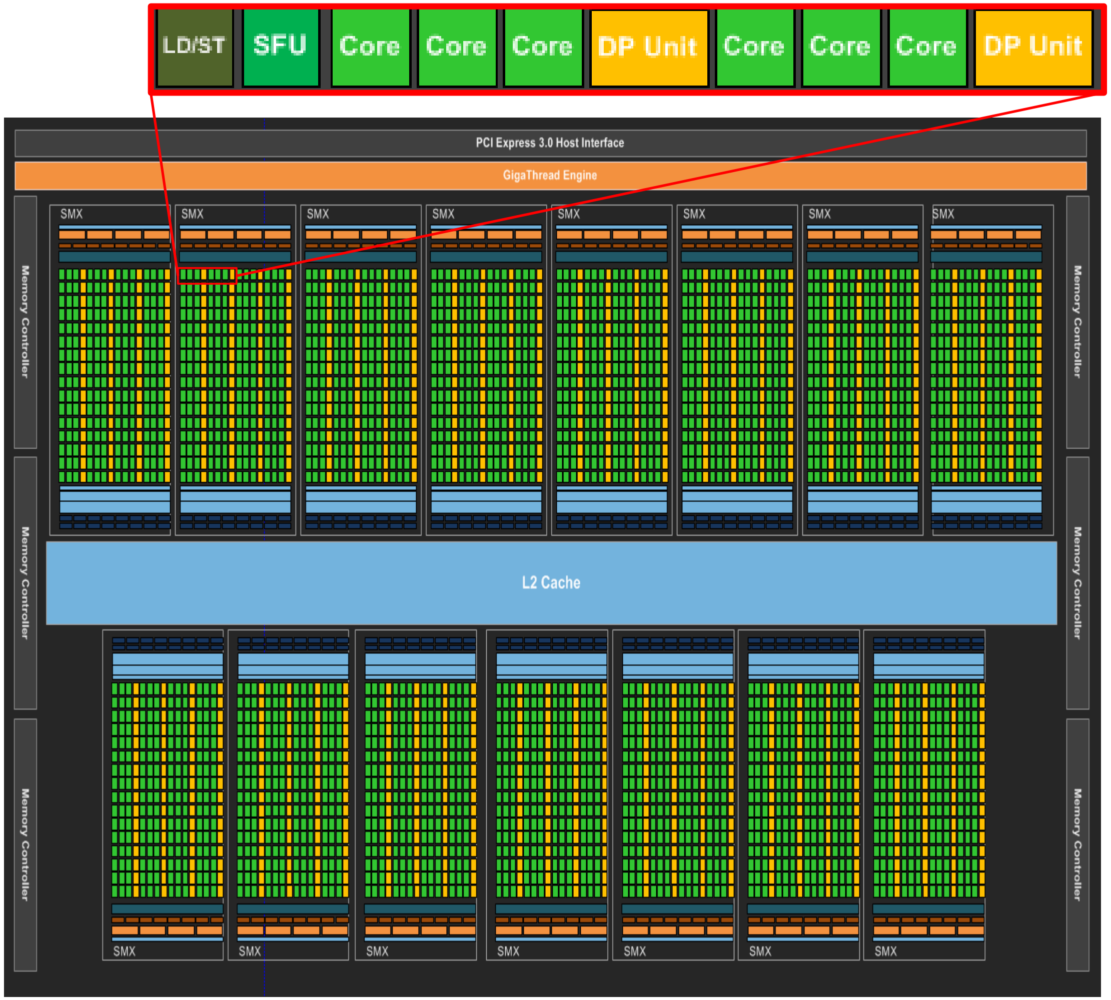
Interlude:
A glance at current accelerator technology

NVIDIA Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 “SMX” units
  - 192 (SP) “cores” each
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

- 3:1 SP:DP performance

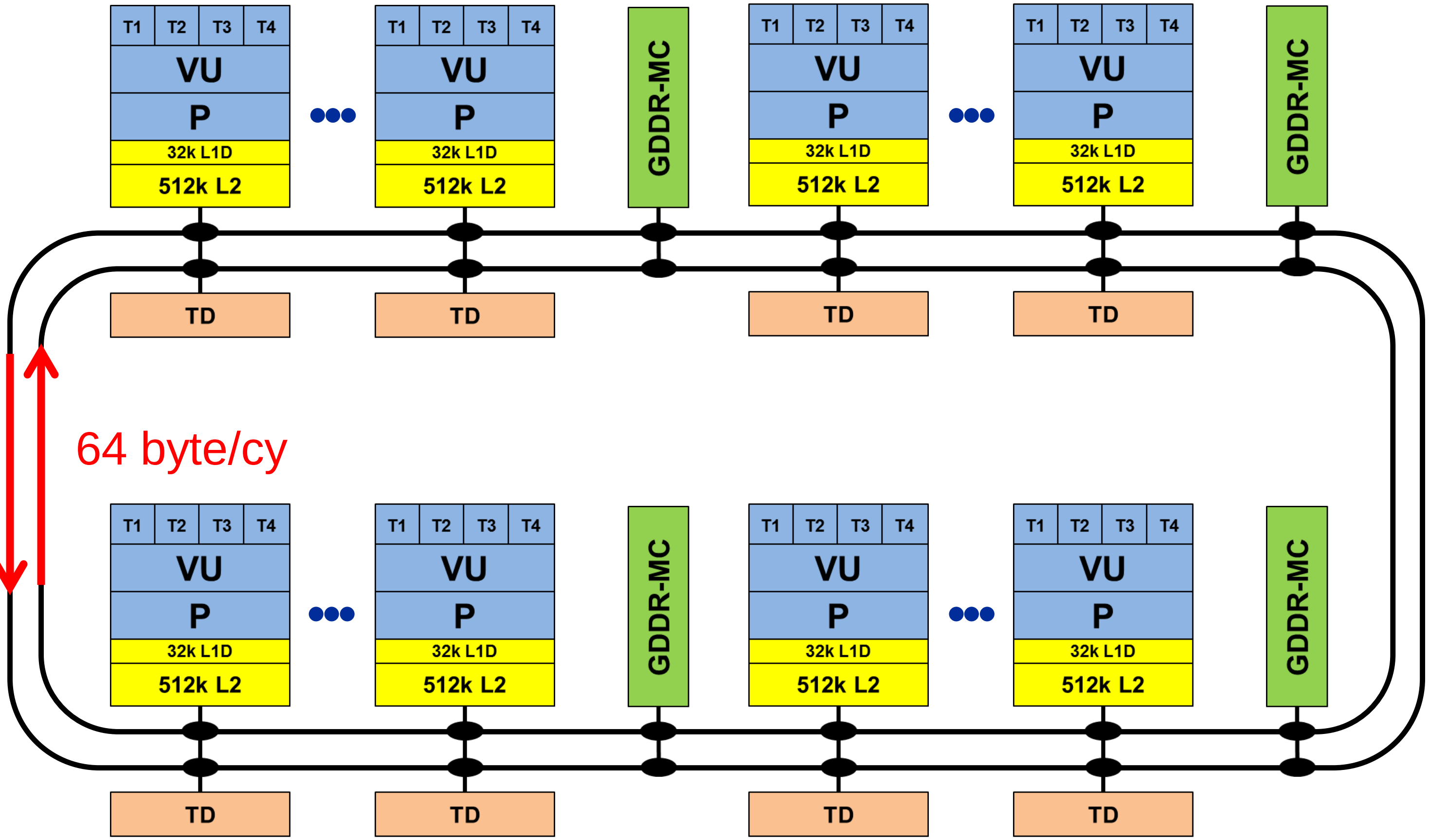


© NVIDIA Corp. Used with permission.

Intel Xeon Phi block diagram

Architecture

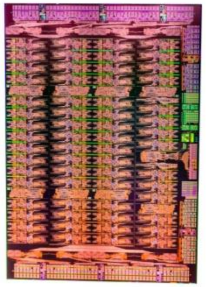
- 3B Transistors
- 60+ cores
- 512 bit SIMD
- ≈ 1 TFLOP DP peak
- 0.5 MB L2/core
- GDDR5
- 2:1 SP:DP performance



Comparing accelerators

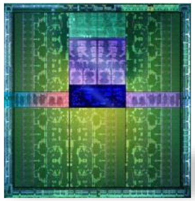
Intel Xeon Phi

- **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **480/960 SIMD DP/SP tracks**



- Clock Speed: ~1000 MHz
- Transistor count: ~3 B (22nm)
- Power consumption: ~250 W
- Peak Performance (DP): ~ 1 TF/s
- Memory BW: ~250 GB/s (GDDR5)
- Threads to execute: 60-240+
- Programming: Fortran/C/C++ +OpenMP + SIMD

NVIDIA Kepler K20



- 15 SMX units each with 192 "cores" → **960/2880 DP/SP "cores"**
- Clock Speed: ~700 MHz
- Transistor count: 7.1 B (28nm)
- Power consumption: ~250 W
- Peak Performance (DP): ~ 1.3 TF/s
- Memory BW: ~ 250 GB/s (GDDR5)
- Threads to execute: 10,000+
- Programming: CUDA, OpenCL, (OpenACC)

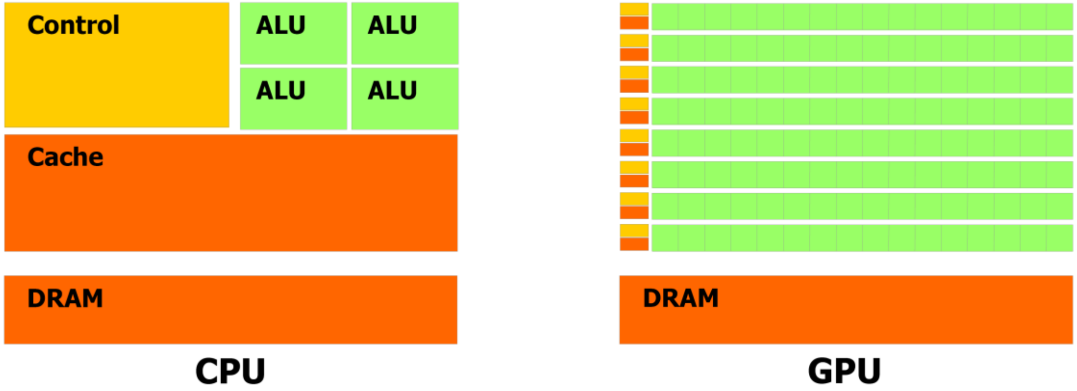
TOP500 rankings Nov 2012

- TOP7: "Stampede" at Texas Center for Advanced Computing
- TOP1: "Titan" at Oak Ridge National Laboratory

Trading single thread performance for parallelism:
GPGPUs vs. CPUs

GPU vs. CPU
light speed estimate:

1. **Compute bound: 2-10x**
2. **Memory Bandwidth: 1-5x**



	Intel Core i5 – 2500 (“Sandy Bridge”)	Intel Xeon E5-2680 DP node (“Sandy Bridge”)	NVIDIA K20x (“Kepler”)
Cores@Clock	4 @ 3.3 GHz	2 x 8 @ 2.7 GHz	2880 @ 0.7 GHz
Performance+/core	52.8 GFlop/s	43.2 GFlop/s	1.4 GFlop/s
Threads@STREAM	<4	<16	>8000?
Total performance+	210 GFlop/s	691 GFlop/s	4,000 GFlop/s
Stream BW	18 GB/s	2 x 40 GB/s	168 GB/s (ECC=1)
Transistors / TDP	1 Billion* / 95 W	2 x (2.27 Billion/130W)	**7.1 Billion/250W**

+ *Single Precision*

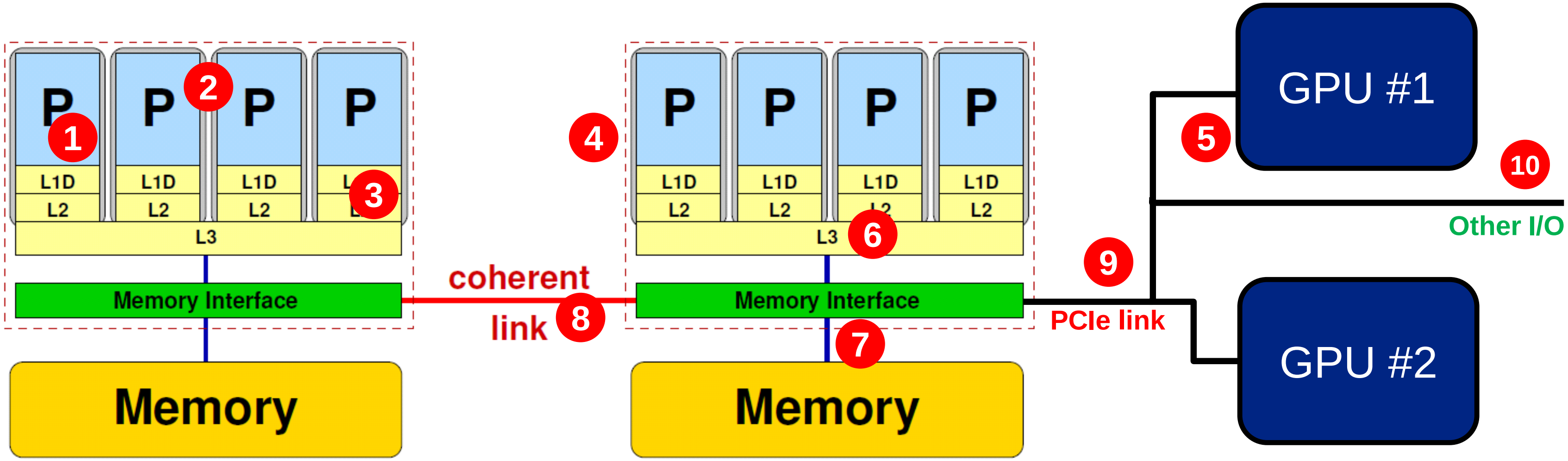
* *Includes on-chip GPU and PCI-Express*

Complete compute device

Node topology and programming models

Parallelism in a modern compute node

- **Parallel and shared resources within a shared-memory node**



Parallel resources:

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

Shared resources:

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

How does your application react to all of those details?

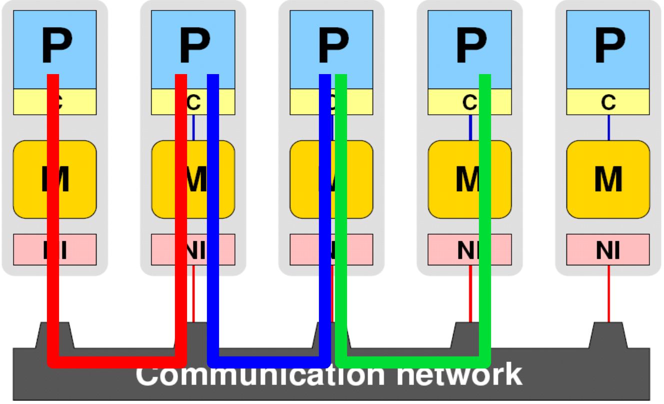
Parallel programming models
on modern compute nodes

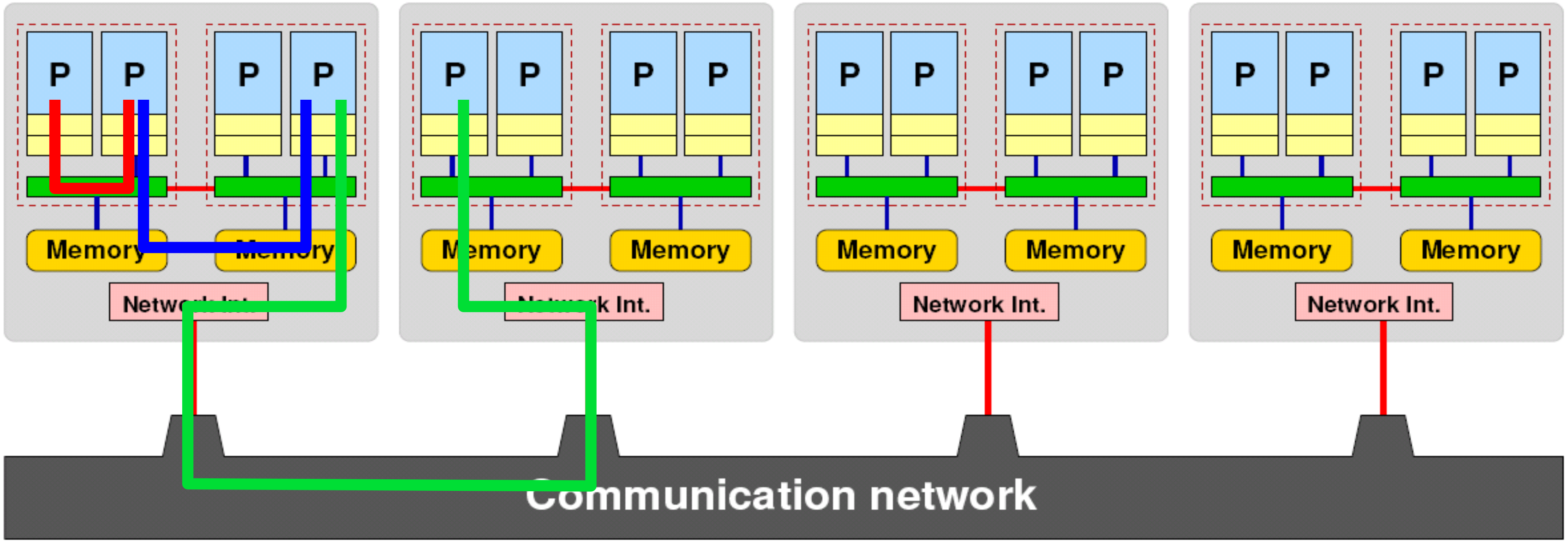
- **Shared-memory (intra-node)**
  - Good old MPI
  - OpenMP
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs,… you name it
- **“Accelerated”**
  - OpenMP 4.0+
  - CUDA
  - OpenCL
  - OpenACC
- **Distributed-memory (inter-node)**
  - MPI
  - PGAS (CAF, UPC, …)
- **Hybrid**
  - Pure MPI + X, X == <you name it>

All models require awareness of *topology* and *affinity* issues for getting best performance out of the machine!

Parallel programming models:
Pure MPI

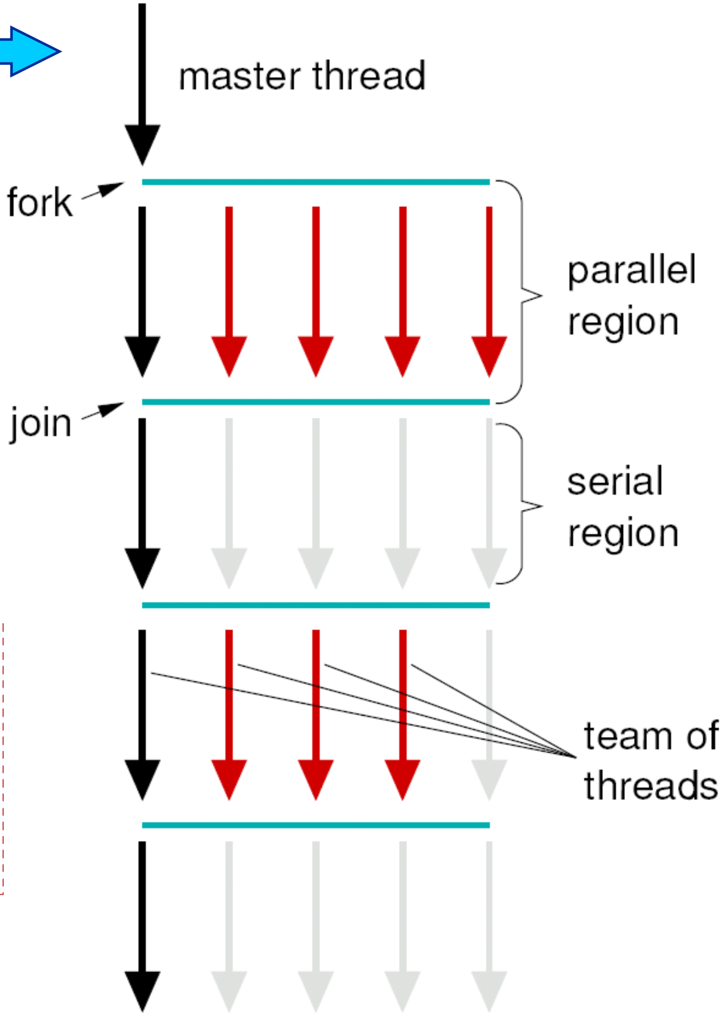
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

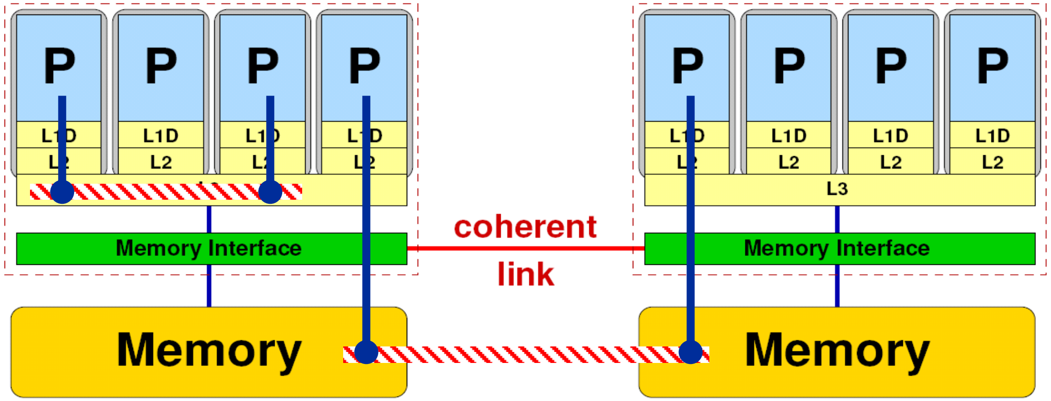




Parallel programming models:
Pure threading on the node

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

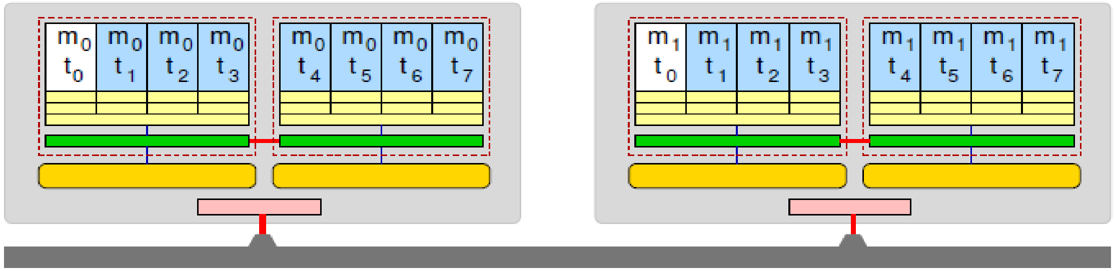




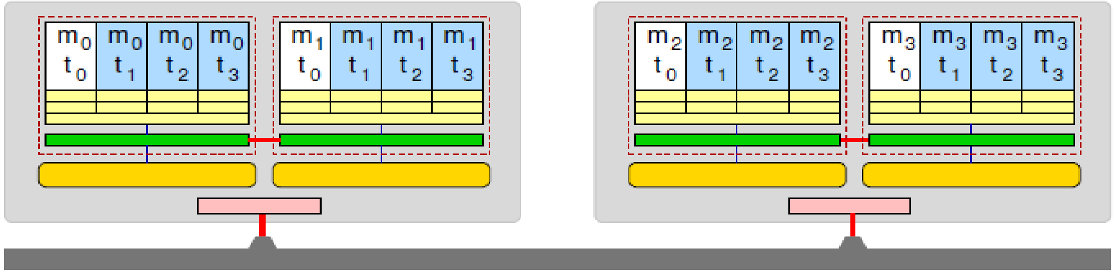
Parallel programming models: Lots of choices

Hybrid MPI+OpenMP on a multicore multisocket cluster

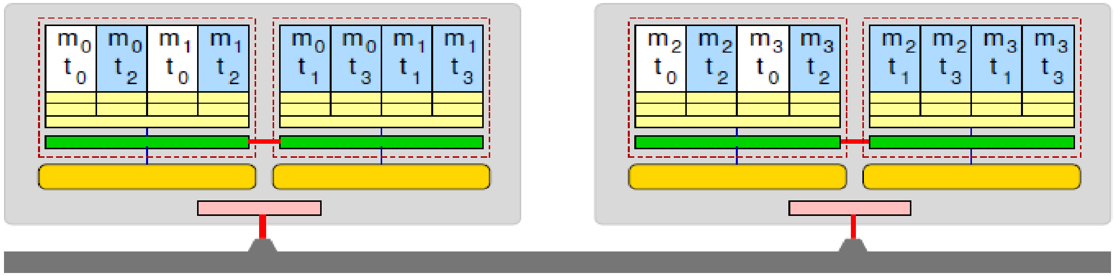
One MPI process / node


m_0 t_0 m_0 t_1 m_0 t_2 m_0 t_3
m_0 t_4 m_0 t_5 m_0 t_6 m_0 t_7
m_1 t_0 m_1 t_1 m_1 t_2 m_1 t_3
m_1 t_4 m_1 t_5 m_1 t_6 m_1 t_7

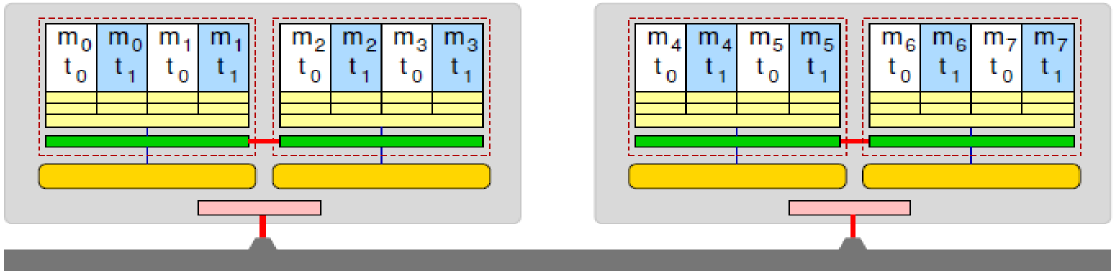
One MPI process / socket:
OpenMP threads on same
socket: **"blockwise"**


m_0 t_0 m_0 t_1 m_0 t_2 m_0 t_3
m_1 t_0 m_1 t_1 m_1 t_2 m_1 t_3
m_2 t_0 m_2 t_1 m_2 t_2 m_2 t_3
m_3 t_0 m_3 t_1 m_3 t_2 m_3 t_3

OpenMP threads pinned
"round robin" across
cores in node


m_0 t_0 m_0 t_2 m_1 t_0 m_1 t_2
m_0 t_1 m_0 t_3 m_1 t_1 m_1 t_3
m_2 t_0 m_2 t_2 m_3 t_0 m_3 t_2
m_2 t_1 m_2 t_3 m_3 t_1 m_3 t_3

Two MPI processes / socket
OpenMP threads
on same socket


m_0 t_0 m_0 t_1 m_1 t_0 m_1 t_1
m_2 t_0 m_2 t_1 m_3 t_0 m_3 t_1
m_4 t_0 m_4 t_1 m_5 t_0 m_5 t_1
m_6 t_0 m_6 t_1 m_7 t_0 m_7 t_1

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: 4-16) to massive (GPGPU: 1000'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
  - Apart from overheads, performance features are largely independent of the programming model

Multicore Performance and Tools

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, …

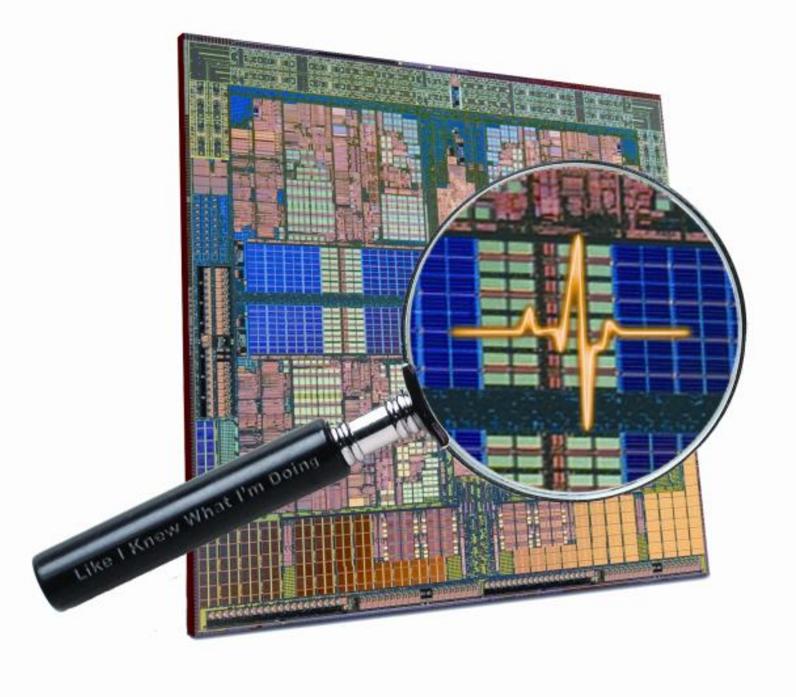
- **Performance Profilers**
  *Intel Vtune^{TM}, **likwid-perfctr**, PAPI based tools, Linux perf, …*

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

LIKWID performance tools

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

Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **Current tools:**
  - **likwid-topology**: Print thread and cache topology
  - **likwid-powermeter**: Measure energy consumption
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr:** Measure performance counters
  - **likwid-bench**: Microbenchmarking tool and environment
  - … some more

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!

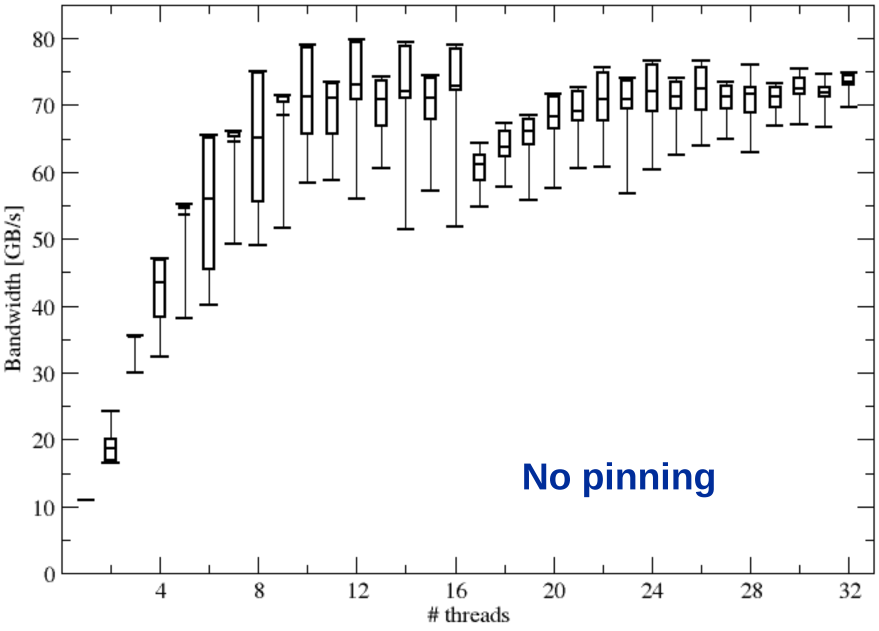
```
********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------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

Enforcing thread/process-core affinity under the Linux OS

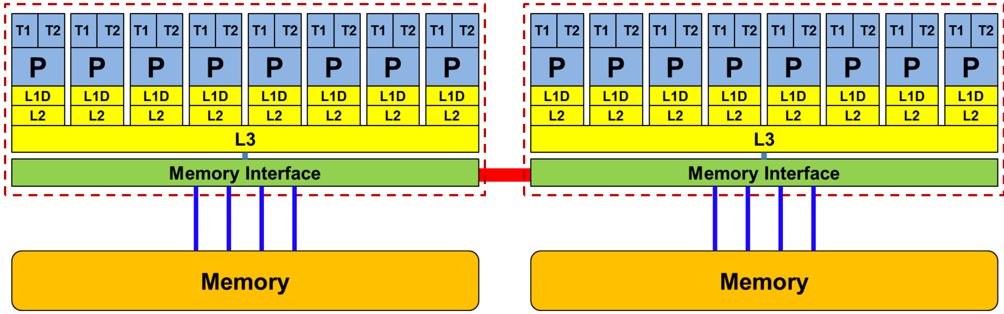
Standard tools and OS affinity facilities under program control

likwid-pin

Example: STREAM benchmark on 16-core Sandy Bridge:
Anarchy vs. thread pinning

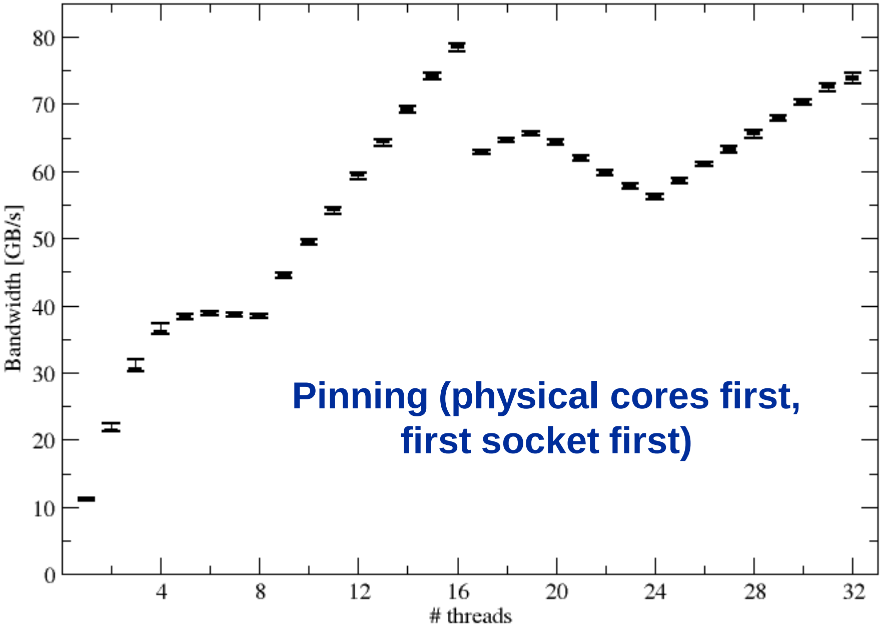


No pinning



There are several reasons for caring about affinity:

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



Pinning (physical cores first, first socket first)

More thread/Process-core affinity ("pinning") options

- **Highly OS-dependent system calls**
  - But available on all systems

    Linux: `sched_setaffinity()`
    Windows: `SetThreadAffinityMask()`

- **Hwloc project** (http://www.open-mpi.de/projects/hwloc/)
- **Support for "semi-automatic" pinning in some compilers/environments**
  - All modern compilers with OpenMP support
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`** (see below)
  - OpenMP 4.0 (see OpenMP tutorial)
- **Affinity awareness in MPI libraries**
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - …

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

- Usage examples:
  - `likwid-pin -c 0-3,4,6  ./myApp parameters`
  - `likwid-pin -c S0:0-7  ./myApp parameters`
  - `likwid-pin -c N:0-15 ./myApp parameters`

- **`OMP_NUM_THREADS`** is set by the tool if not set explicitly

LIKWID terminology

Thread group syntax

- The OS numbers all processors (hardware threads) on a node
- The numbering is enforced at boot time by the BIOS and may have nothing to do with topological entities
- LIKWID concept: **thread group** consisting of HW threads sharing a topological entity (e.g., socket, shared cache,…)
- A thread group is defined by a single **character + index**

- Example:
  ```
  likwid-pin -c S1:0-3,6,7 ./a.out
  ```
- Group expression chaining with `@`:
  ```
  likwid-pin -c S0:0-3@S1:0-3 ./a.out
  ```

numbering across physical cores first within the group

```
+-----------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| | 0   4 | | 1   5 | | 2   6 | | 3   7 | |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |   32kB| |   32kB| |   32kB| |   32kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  256kB| |  256kB| |  256kB| |  256kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------+ |
| |                 8MB                 | |
| +-------------------------------------+ |
+-----------------------------------------+
```

- Alternative expression based syntax:
  ```
  likwid-pin -c E:S0:4:2:4 ./a.out
  ```

```
E:<thread domain>:<num threads>:<chunk size>:<stride>
```

- Expression syntax is convenient for Xeon Phi:
  ```
  likwid-pin -c E:N:120:2:4 ./a.out
  ```

compact numbering within the group

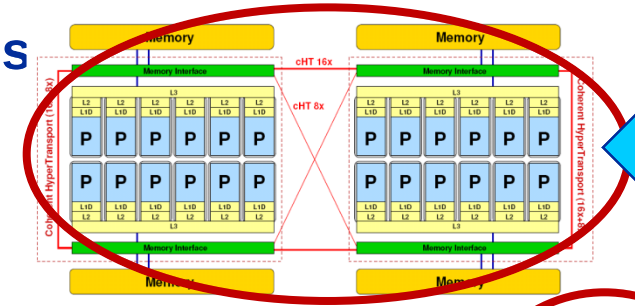
```
+-----------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| | 0   1 | | 2   3 | | 4   5 | | 6   7 | |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |   32kB| |   32kB| |   32kB| |   32kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  256kB| |  256kB| |  256kB| |  256kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------------------------------------+ |
| |                 8MB                 | |
| +-------------------------------------+ |
+-----------------------------------------+
```

Likwid

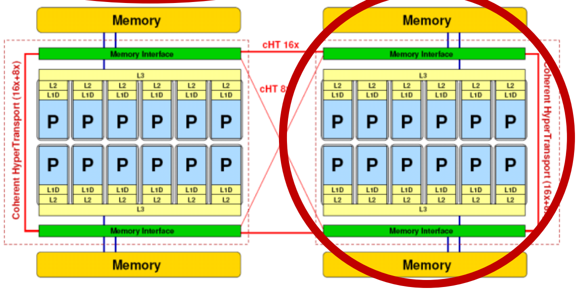
Currently available thread domains

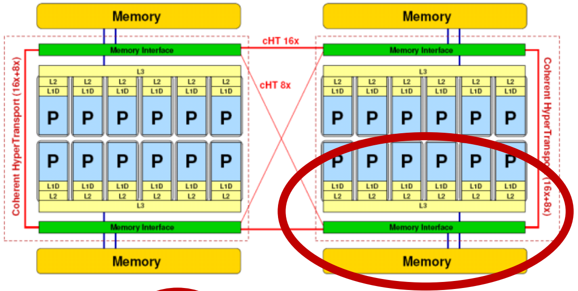
- **Possible unit prefixes**

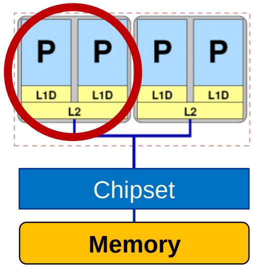
N	**node**
S	**socket**
M	**NUMA domain**
C	**outer level cache group**



Default if –c is not specified!







Likwid-pin

Example: Intel OpenMP

- **Running the STREAM benchmark with likwid-pin:**

```
$ likwid-pin -c S0:0-3 ./stream
----------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
----------------------------------------------
 Array size =   20000000
 Offset     =         32
 The total memory requirement is  457 MB
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
[pthread wrapper] 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

Setting the clock frequency

- **The “Turbo Mode” feature makes reliable benchmarking harder**
  - CPU can change clock speed at its own discretion
- **Clock speed reduction may save a lot of energy**

- **So how do we set the clock speed? → LIKWID to the rescue!**

```
$ likwid-setFrequencies -l
Available frequencies:
2.301, 2.3, 2.2, 2.1, 2.0, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, 1.2
$ likwid-setFrequencies -p
Current frequencies:
CPU 0: governor  performance frequency 2.301 GHz
CPU 1: governor  performance frequency 2.301 GHz
CPU 2: governor  performance frequency 2.301 GHz
CPU 3: governor  performance frequency 2.301 GHz
[...]
$ likwid-setFrequencies -f 2.3
$
```

Turbo mode

Multicore performance tools: Probing performance behavior

likwid-perfctr

likwid-perfctr

Basic approach to performance analysis

1. Runtime profile / Call graph (gprof): Where are the hot spots?
2. Instrument hot spots (prepare for detailed measurement)
3. Find performance signatures

Possible signatures:

- Bandwidth saturation
- Instruction throughput limitation (real or language-induced)
- Latency impact (irregular data access, high branch ratio)
- Load imbalance
- ccNUMA issues (data access across ccNUMA domains)
- Pathologic cases (false cacheline sharing, expensive operations)

likwid-perfctr can help here

Goal: Come up with educated guess about a performance-limiting motif (Performance Pattern)

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.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**

likwid-perfctr

Best practices for runtime counter analysis

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

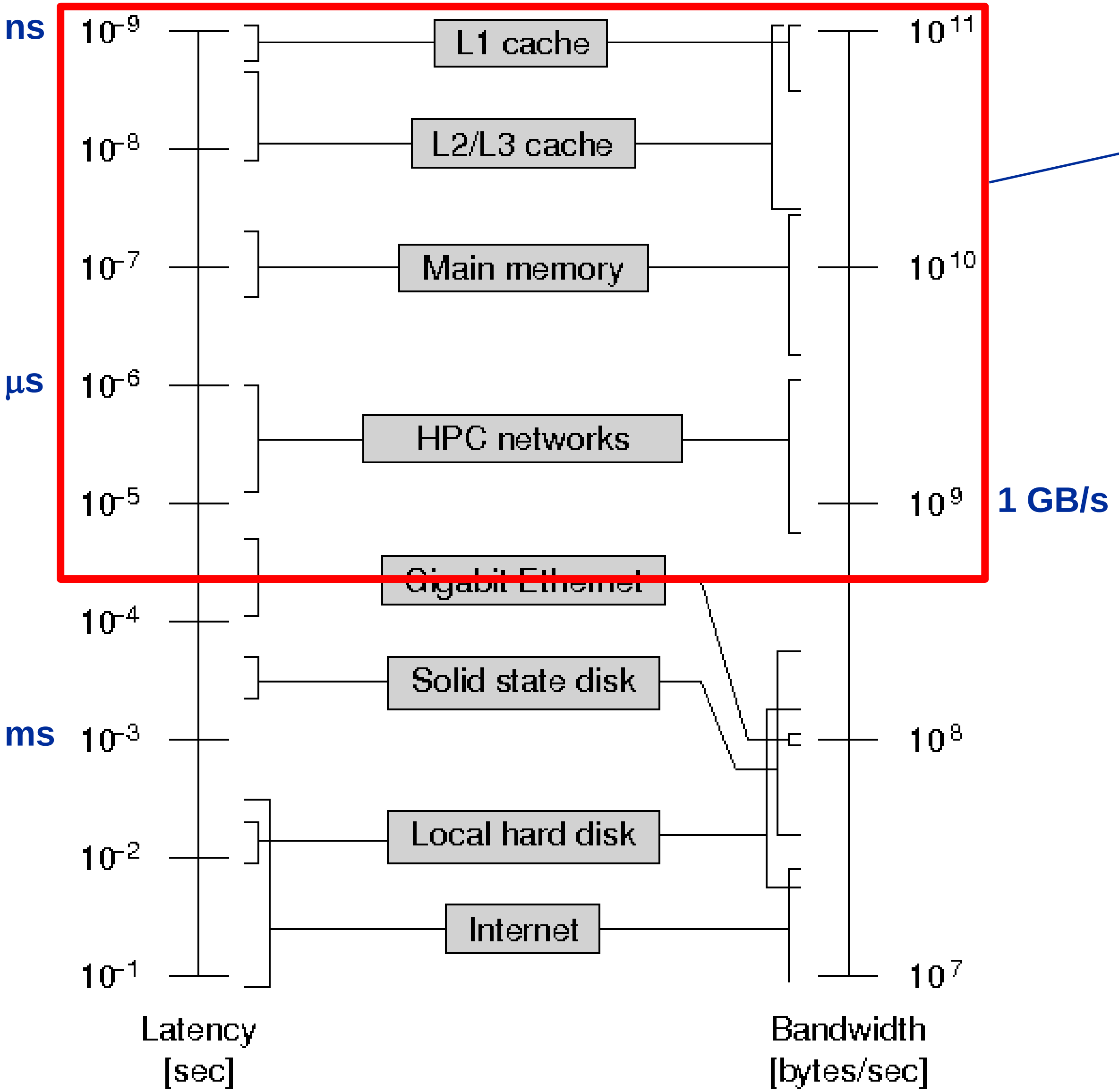
- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are sometimes misleading

Microbenchmarking for architectural exploration (and more)

Probing of the memory hierarchy

Saturation effects in cache and memory

Latency and bandwidth in modern computer environments

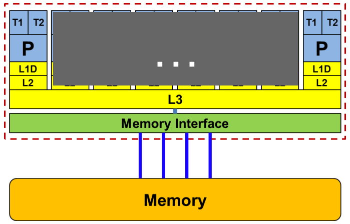


HPC plays here

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

Intel Xeon E5 multicore processors

Microarchitecture	SandyBridge-EP	IvyBridge-EP	Haswell-EP
Shorthand	SNB	IVB	HSW
Xeon Model	E5-2680	E5-2690 v2	E5-2695 v3
Year	03/2012	09/2013	09/2014
Clock speed (fixed)	2.7 GHz	2.2 GHz	2.3 GHz
Cores/Threads	8/16	10/20	14/28
Load/Store throughput per cycle			
AVX(2)	1 LD & 1/2 ST	1 LD & 1/2 ST	2 LD & 1 ST
SSE/scalar	2 LD ‖ 1 LD & 1 ST	2 LD ‖ 1 LD & 1 ST	2 LD & 1 ST
L1 port width	2×16+1×16 B	2×16+1×16 B	2×32+1×32 B
ADD throughput	1 / cy	1 / cy	1 / cy
MUL throughput	1 / cy	1 / cy	2 / cy
FMA throughput	n/a	n/a	2 / cy
L2-L1 data bus	32 B	32 B	64 B
L3-L2 data bus	32 B	32 B	32 B
LLC size	20 MiB	25 MiB	35 MiB
Main memory	4×DDR3-1600	4×DDR3-1866	4×DDR4-2133
Peak memory BW	51.2 GB/s	51.2 GB/s	68.3 GB/s
Load-only BW	43.6 GB/s (85%)	46.1 GB/s (90%)	60.6 GB/s (89%)
T_{L3Mem} per CL	3.96 cy	3.05 cy	2.43 cy



FP instructions throughput per core

Max. data transfer per cycle between caches

Peak main memory bandwidth

The parallel vector triad benchmark

A “swiss army knife” for microbenchmarking

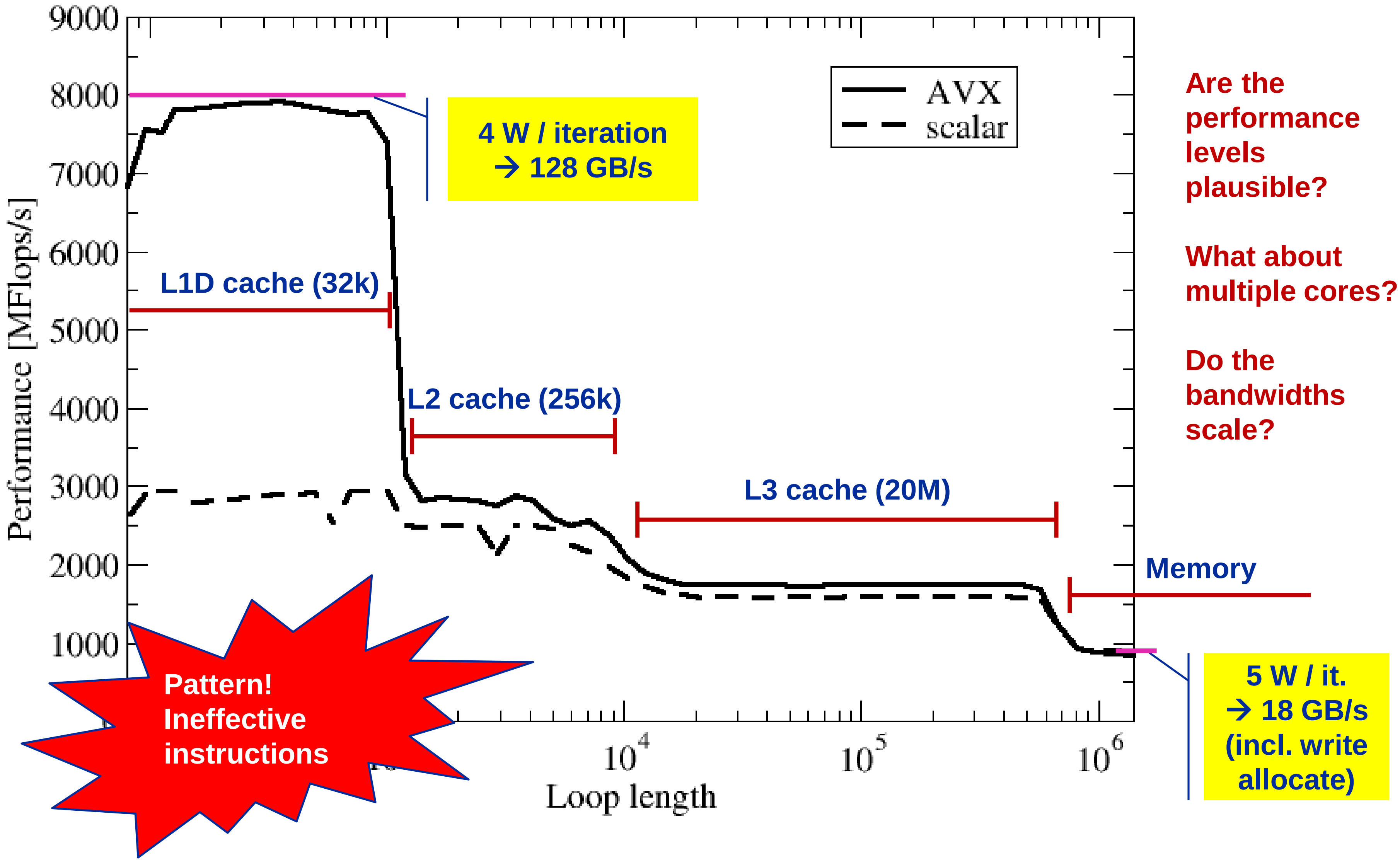
Simple streaming benchmark:

```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

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
```

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 current architectures!**

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



4 W / iteration
→ 128 GB/s

5 W / it.
→ 18 GB/s
(incl. write allocate)

Pattern!
Ineffective instructions

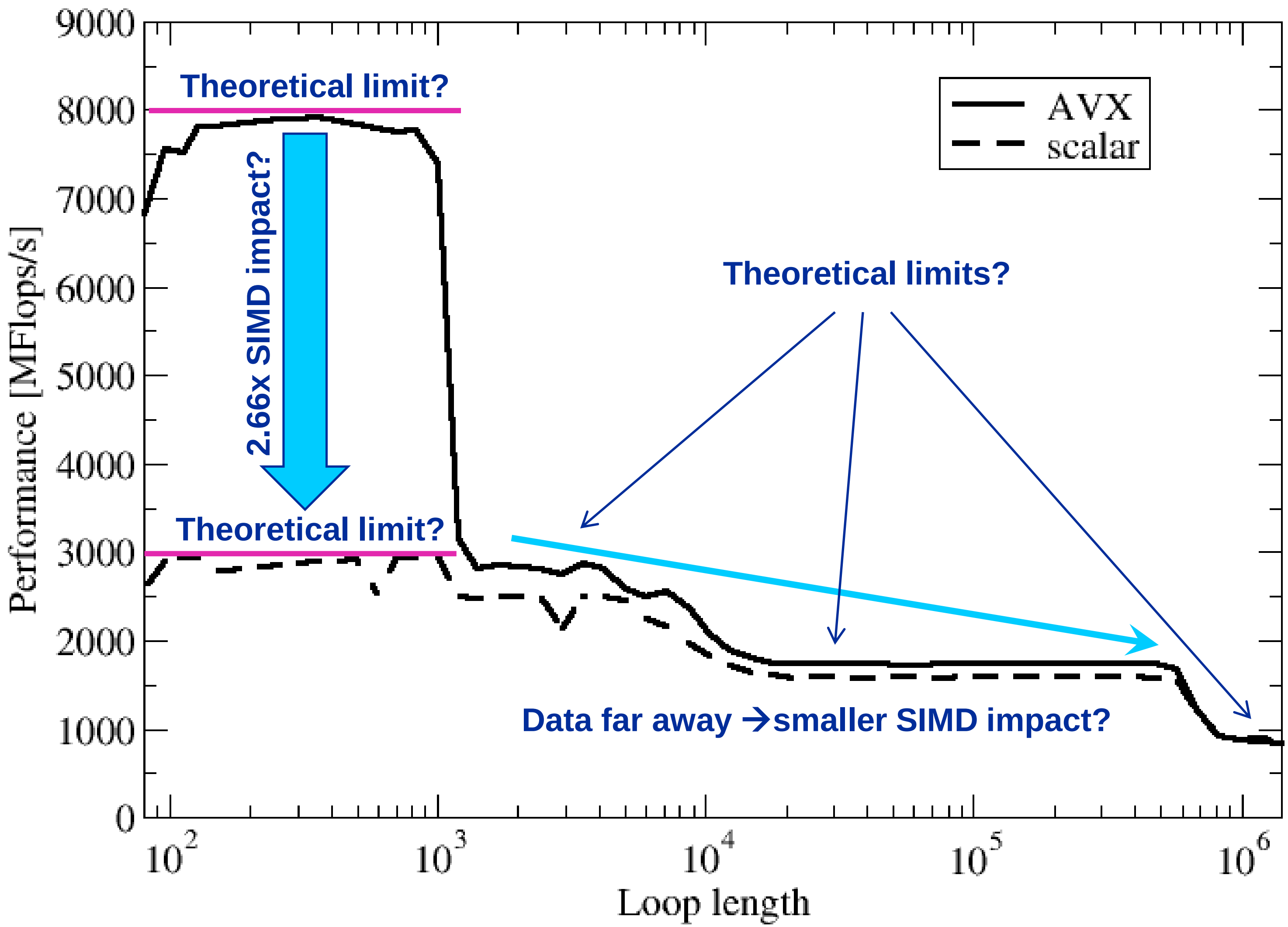
Are the performance levels plausible?

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!

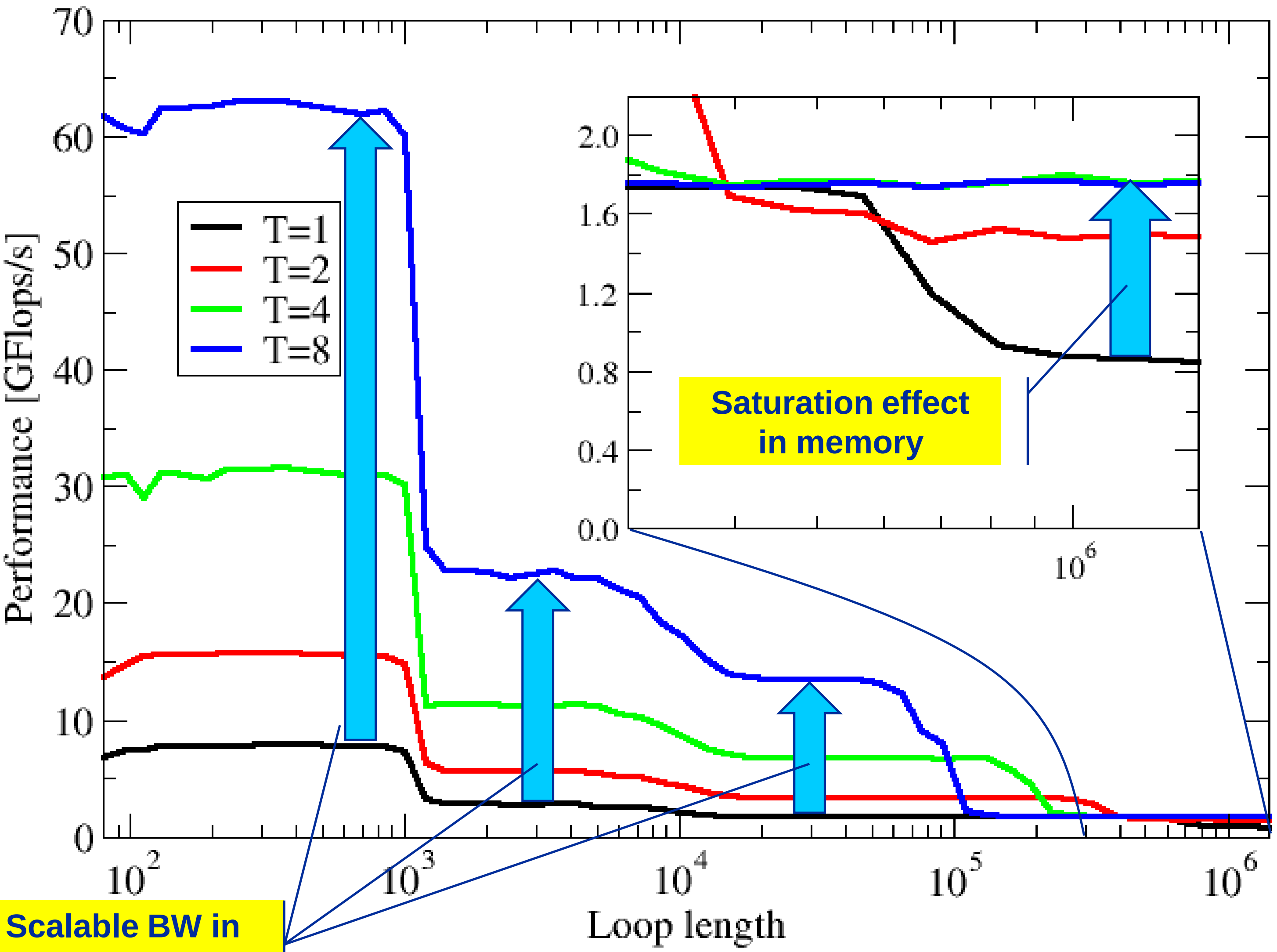
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

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

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

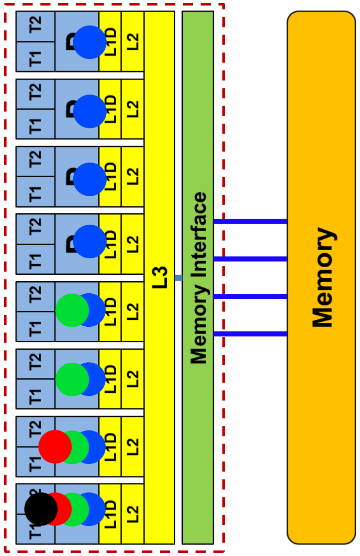
→ pure hardware probing, no impact from OpenMP overhead

Throughput vector triad on Sandy Bridge socket (3 GHz)

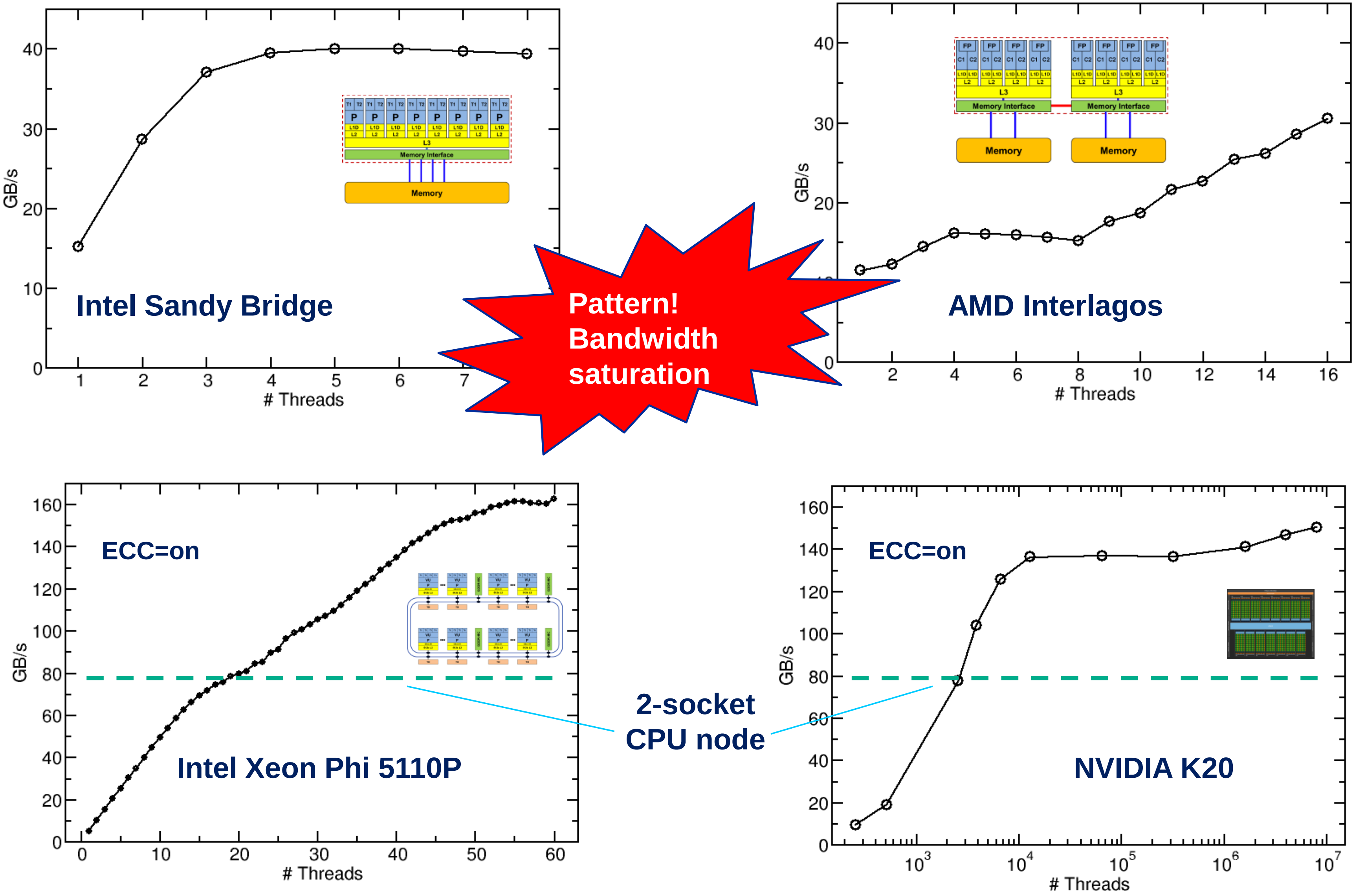


Saturation effect in memory

Scalable BW in L1, L2, L3 cache



Attainable memory bandwidth: Comparing architectures



Intel Sandy Bridge

AMD Interlagos

Pattern! Bandwidth saturation

ECC=on

Intel Xeon Phi 5110P

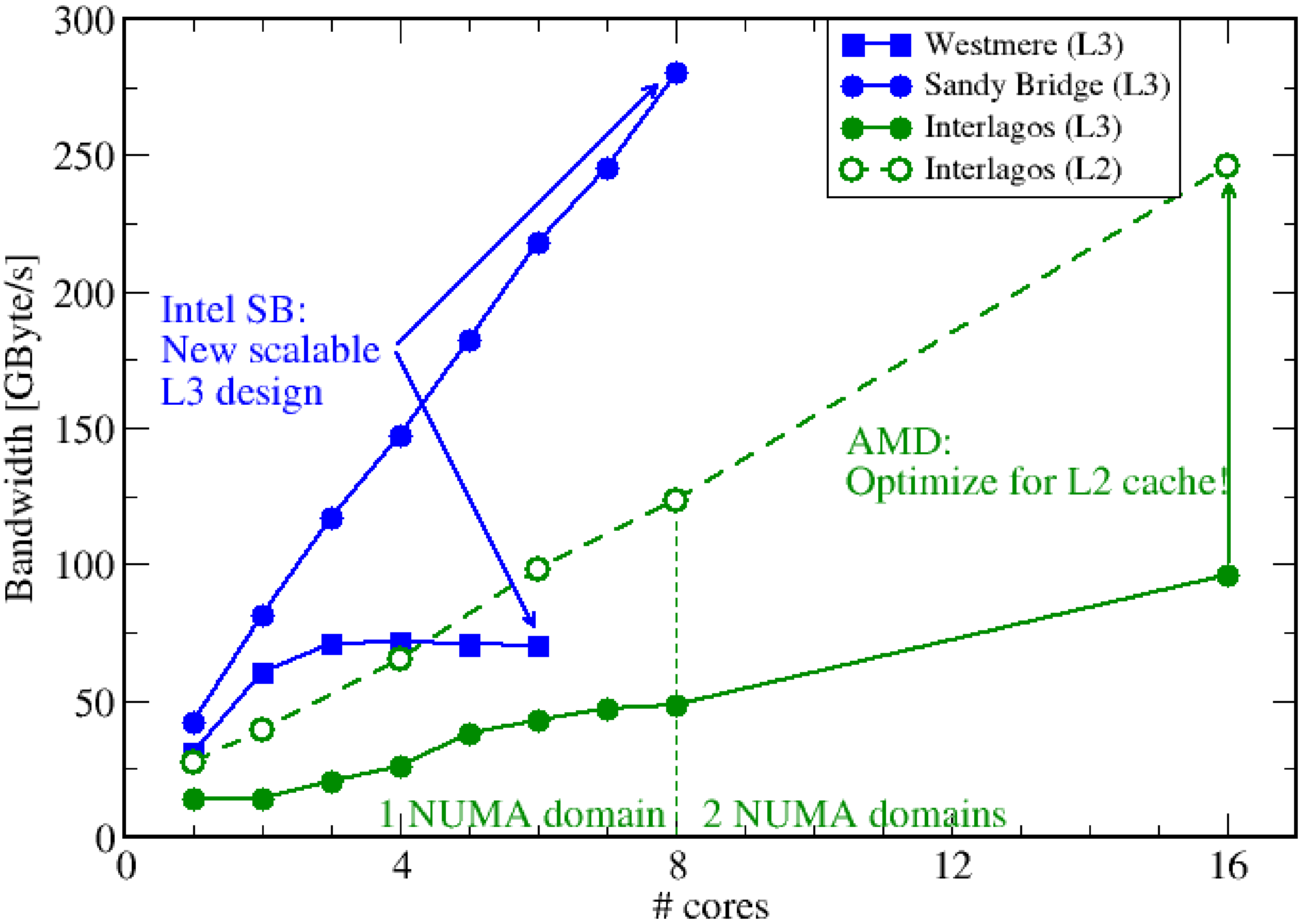
ECC=on

NVIDIA K20

2-socket CPU node

Bandwidth limitations: Outer-level cache

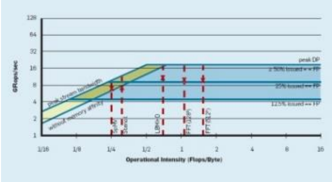
Scalability of shared data paths in L3 cache



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

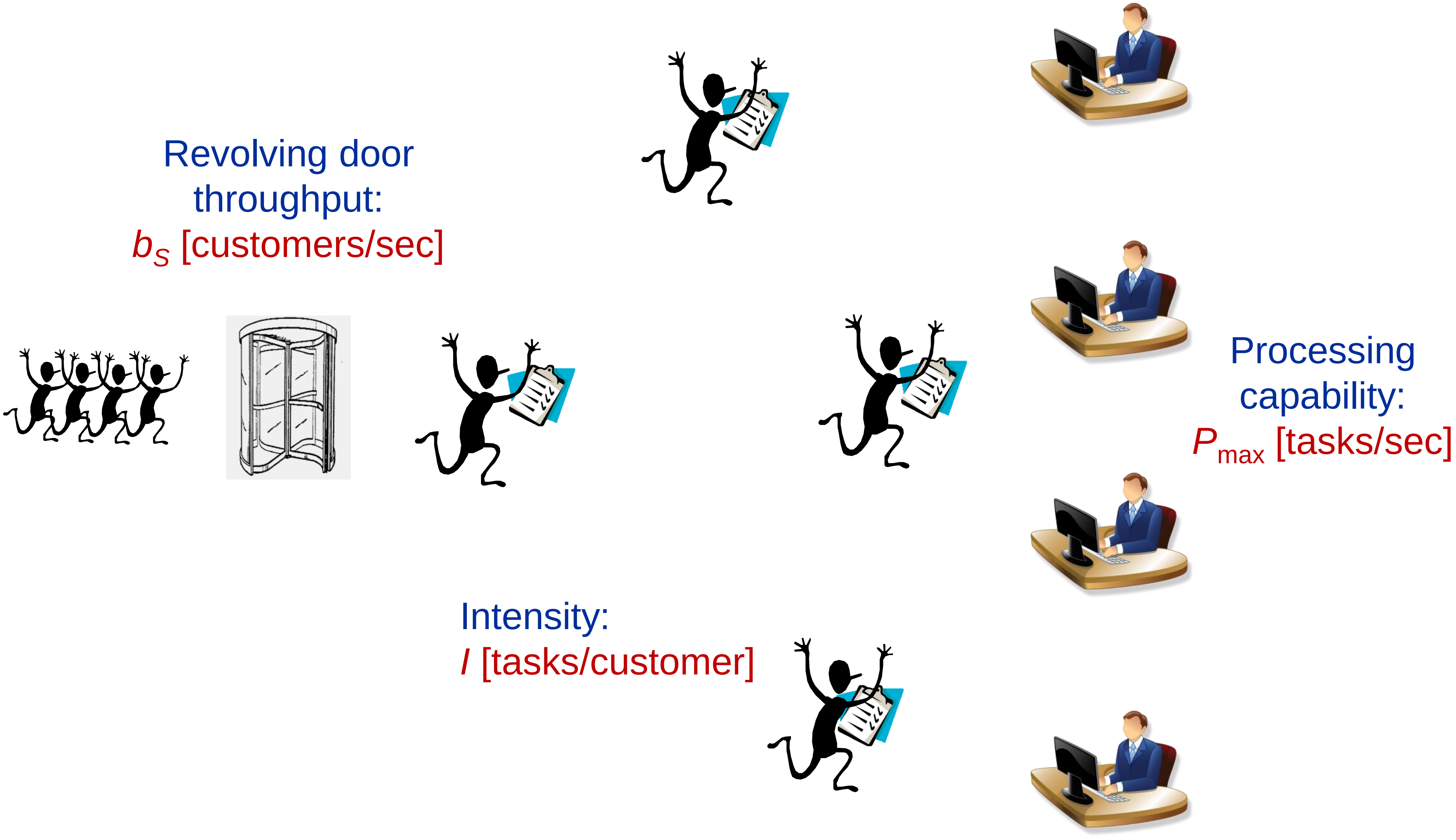
Loop-based performance modeling: Execution vs. data transfer
Example: array summation
Example: dense & sparse matrix-vector multiplication
Example: a 3D Jacobi solver
Model-guided optimization

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)

Prelude: Modeling customer dispatch in a bank



Revolving door throughput:
b_S [customers/sec]

Intensity:
I [tasks/customer]

Processing capability:
P_{max} [tasks/sec]

Prelude: Modeling customer dispatch in a bank

How fast can tasks be processed? P [tasks/sec]

The bottleneck is either

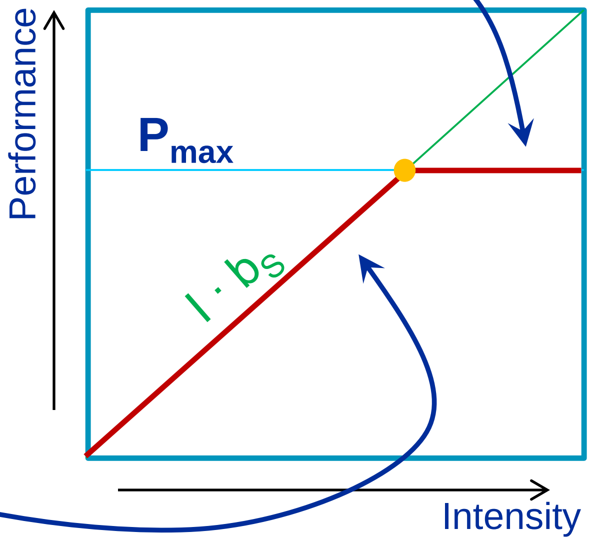
- The service desks (max. tasks/sec): $P_{\max}$
- The revolving door (max. customers/sec): $I \cdot b_S$

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

This is the “Roofline Model”

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

- **Roofline is an “optimistic” model: (“light speed”)**



The Roofline Model

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

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

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

Expected performance:

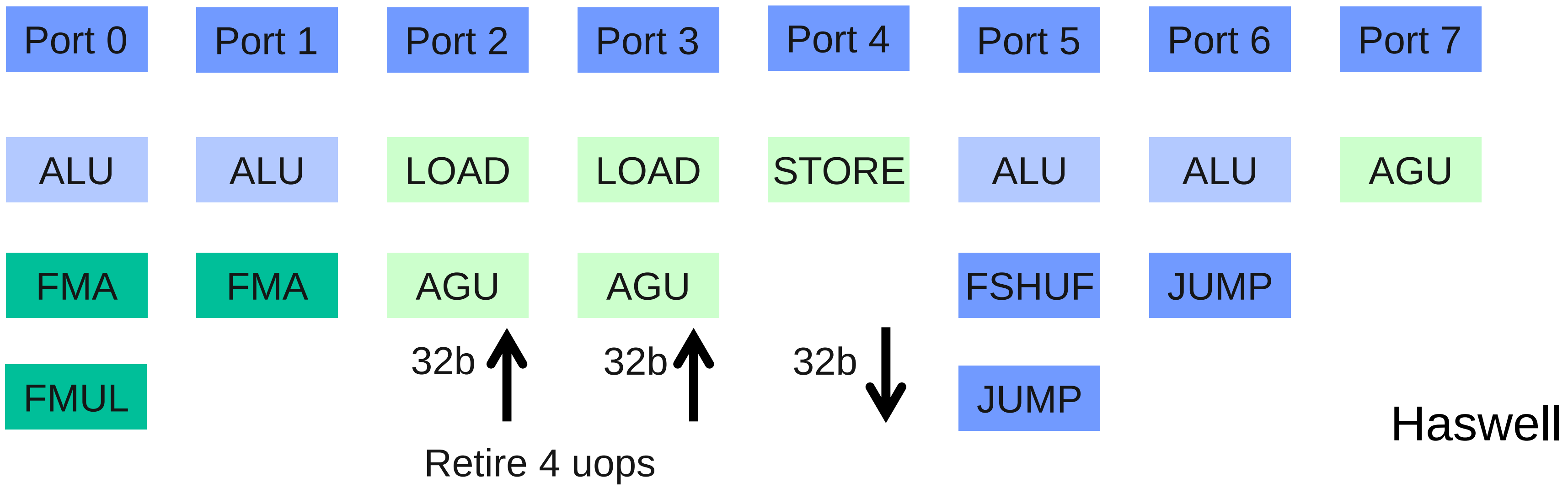
$$P = \min(P_{\max}, I \cdot b_S) = \min\left(P_{\max}, \frac{b_S}{B_C}\right)$$

[Byte/s]

[Byte/Flop]

Preliminary: Estimating P_{max}

Every new CPU generation provides incremental improvements.



Instruction mix	Execution time
1 ADD	1 cy
2 ADD	2 cy
1 MUL	1 cy
2 MUL	1 cy
1 ADD + 1 MUL	1 cy
2 FMA	1 cy

Instruction mix	Execution time
1 LOAD	1 cy
1 STORE	1 cy
1 LOAD and 1 STORE	1 cy
2 LOADs and 1 STORE	1 cy

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)

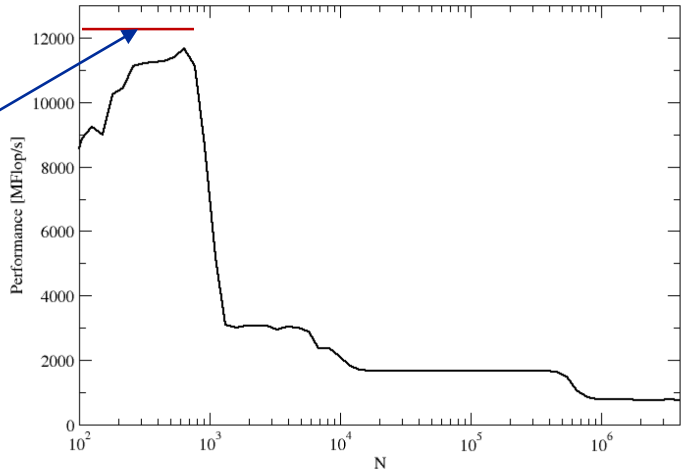
```
double  *A, *B, *C, *D;
for (int i=0; i<N; i++) {
   A[i] = B[i] + C[i] * D[i];
}
```

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:

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

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



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)

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

Code balance: more examples

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

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

I = 0.042 F/B

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

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

I = 0.083 F/B

Scalar – can be kept in register

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

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

I = 0.5 F/B

Scalar – can be kept in register

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

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

I = 0.25 F/B

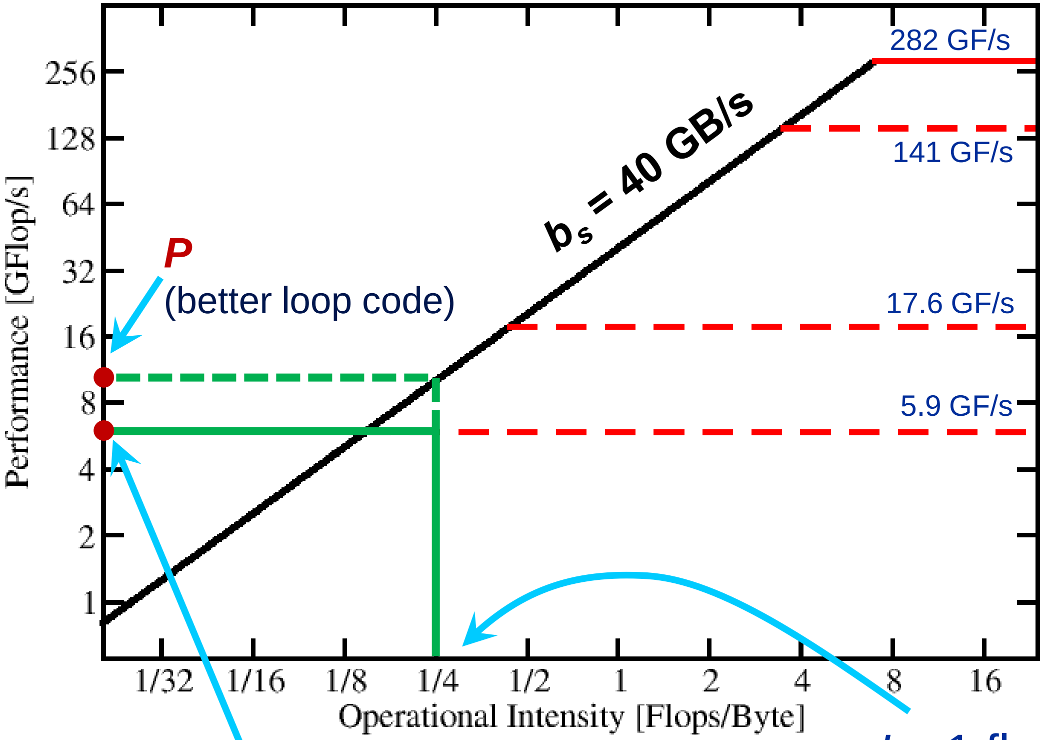
Scalar – can be kept in register

A not so simple Roofline example

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

in single precision on a 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 pipelined

How do we get these? → See next!

P (better loop code)

P (worst loop code)

I = 1 flop / 4 byte (SP!)

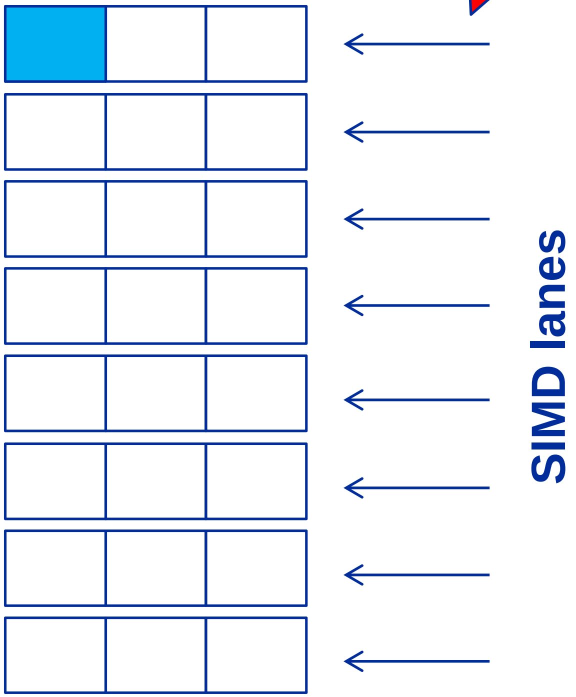
Applicable peak for the summation loop

Plain scalar code, no SIMD

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

Pattern! Pipelining issues

ADD pipes utilization:



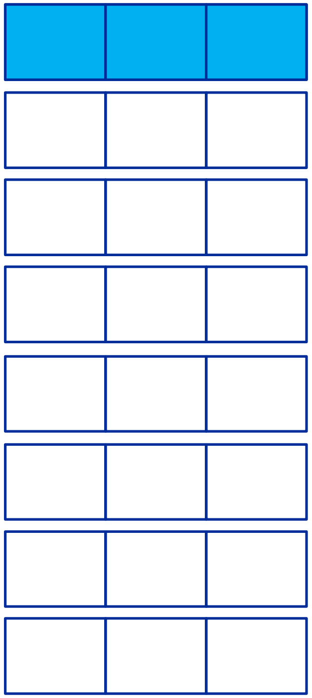
→ 1/24 of ADD peak

Applicable peak for the summation loop

Scalar code, 3-way unrolling

```
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
   ADD r2.0 ← r2.0 + r5.0
   ADD r3.0 ← r3.0 + r6.0

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

ADD pipes utilization:



→ 1/8 of ADD peak

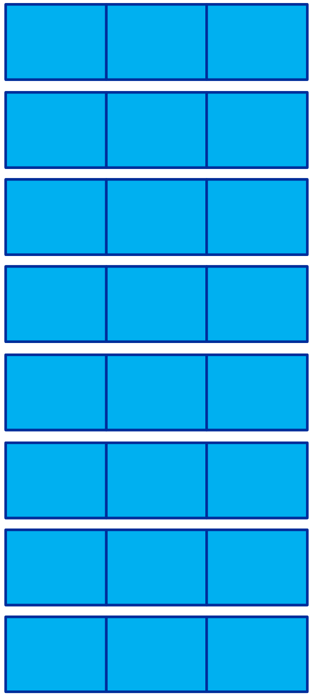
Applicable peak for the summation loop

SIMD-vectorized, 3-way unrolled

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

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

   VADD r1 ← r1 + r4
   VADD r2 ← r2 + r5
   VADD r3 ← r3 + r6

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

Pattern! ALU saturation

ADD pipes utilization:

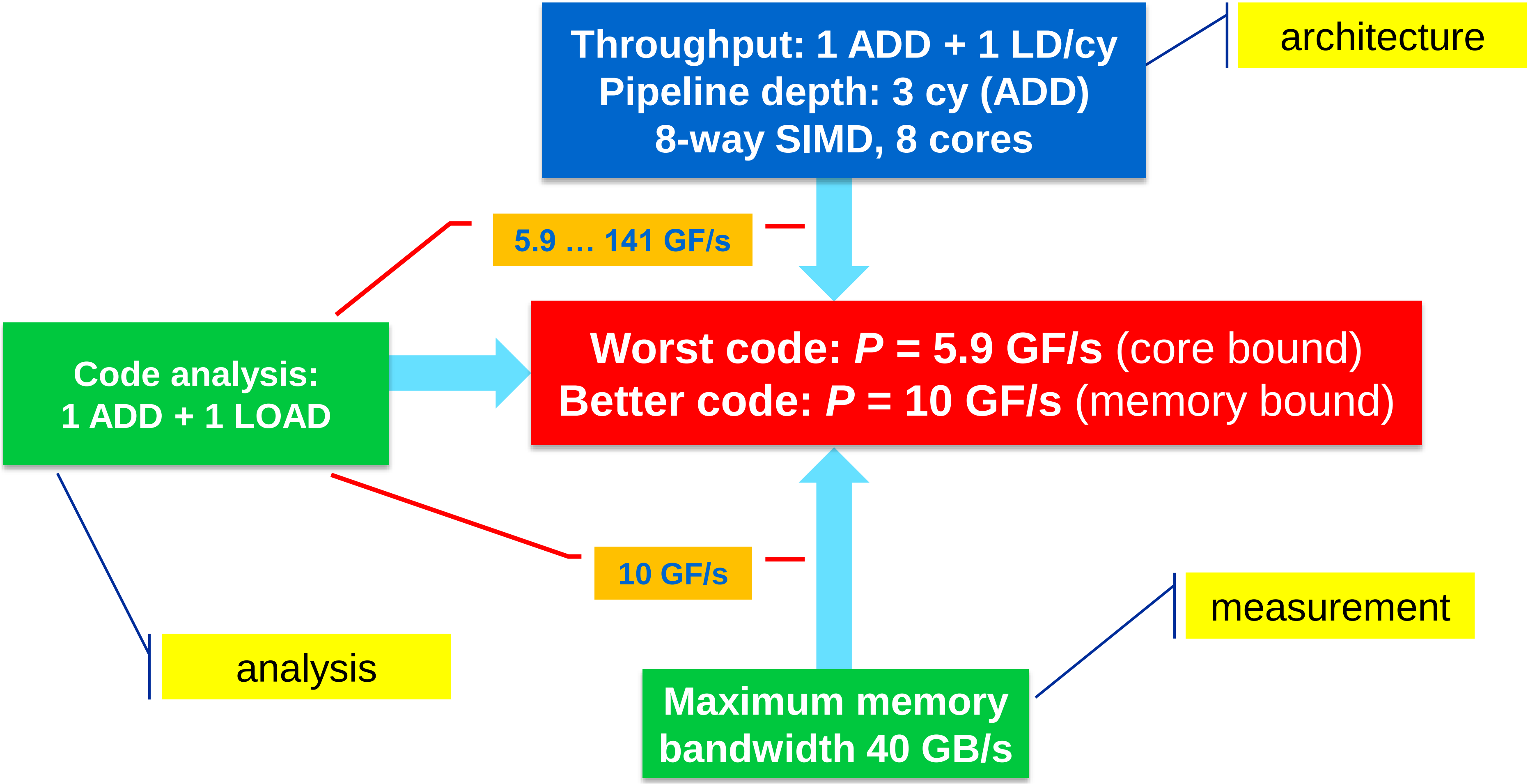


→ ADD peak

Input to the roofline model

… on the example of
in single precision

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



Throughput: 1 ADD + 1 LD/cy
Pipeline depth: 3 cy (ADD)
8-way SIMD, 8 cores

architecture

5.9 … 141 GF/s

Code analysis:
1 ADD + 1 LOAD

Worst code: *P* = 5.9 GF/s (core bound)
Better code: *P* = 10 GF/s (memory bound)

10 GF/s

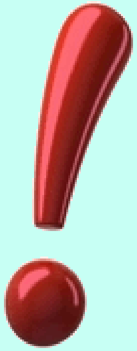
measurement

analysis

Maximum memory bandwidth 40 GB/s

Prerequisites for the Roofline Model

- **The roofline formalism is based on some (crucial) assumptions:**
  - 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 bandwidth via simple streaming benchmarks to model more complex kernels and applications
  - 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: perfect data streaming, “steady-state” execution, no start-up effects



Case study:
Dense Matrix Vector Multiplication

Example: Dense matrix-vector multiplication in DP (AVX)

```
do c = 1 , C
 do r = 1 , R
     y(r)=y(r) + A(r,c)* x(c)
 enddo
enddo
```

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

- **Assume C = R ≈ 10,000**
- **Applicable peak performance?**
- **Relevant data path?**
- **Computational Intensity?**

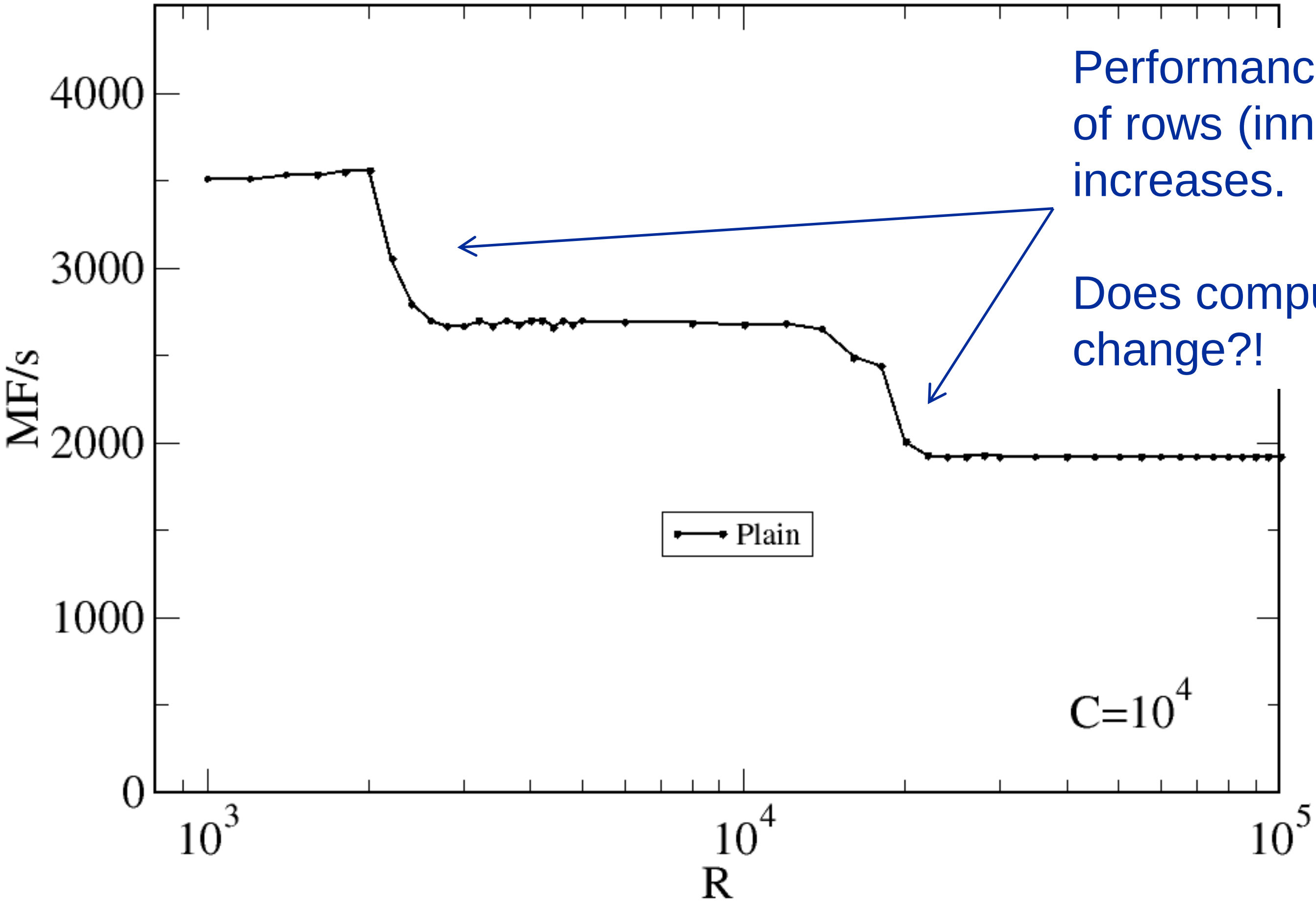
DMVM (DP) – AVX vectorization

- Vectorization strategy: 4-way inner loop unrolling
- One register holds `tmp` in each of its 4 entries (“broadcast”)

```
do c = 1,C
  tmp=x(c)
  do r = 1, R , 4   ! R is multiple of 4
     y(r)   = y(r)   + A(r,c)  * tmp
     y(r+1) = y(r+1) + A(r+1,c)* tmp
     y(r+2) = y(r+2) + A(r+2,c)* tmp
     y(r+3) = y(r+3) + A(r+3,c)* tmp
  enddo
enddo
```

- Loop kernel requires/consumes 3 AVX registers

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



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

Does computational intensity change?!

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

DMVM data traffic analysis

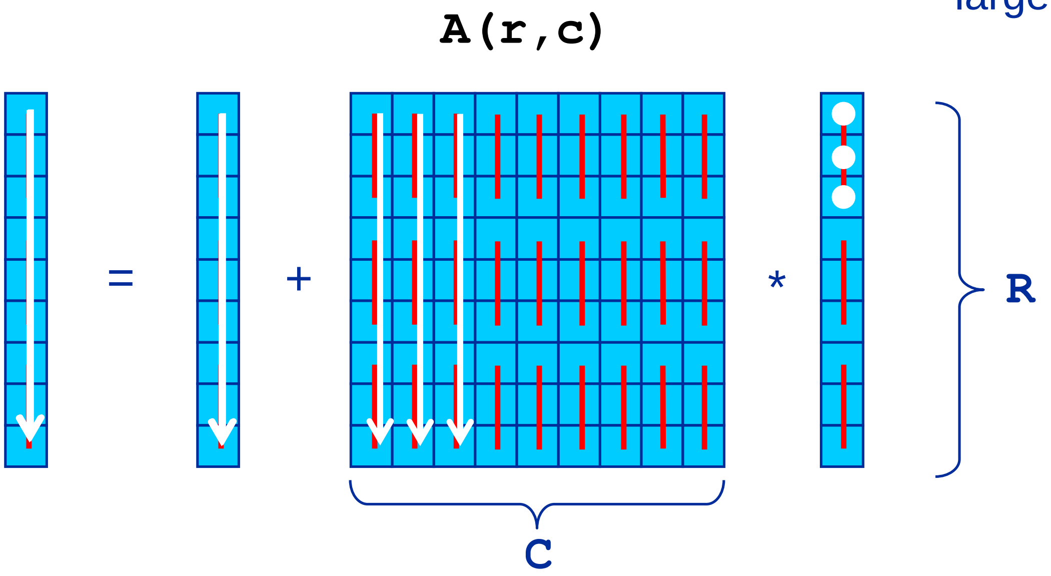
```
do c = 1 , C
  tmp=x(c)
  do r = 1 , R
     y(r)=y(r) + A(r,c)* tmp
  enddo
enddo
```

`tmp` stays in a register during inner loop

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

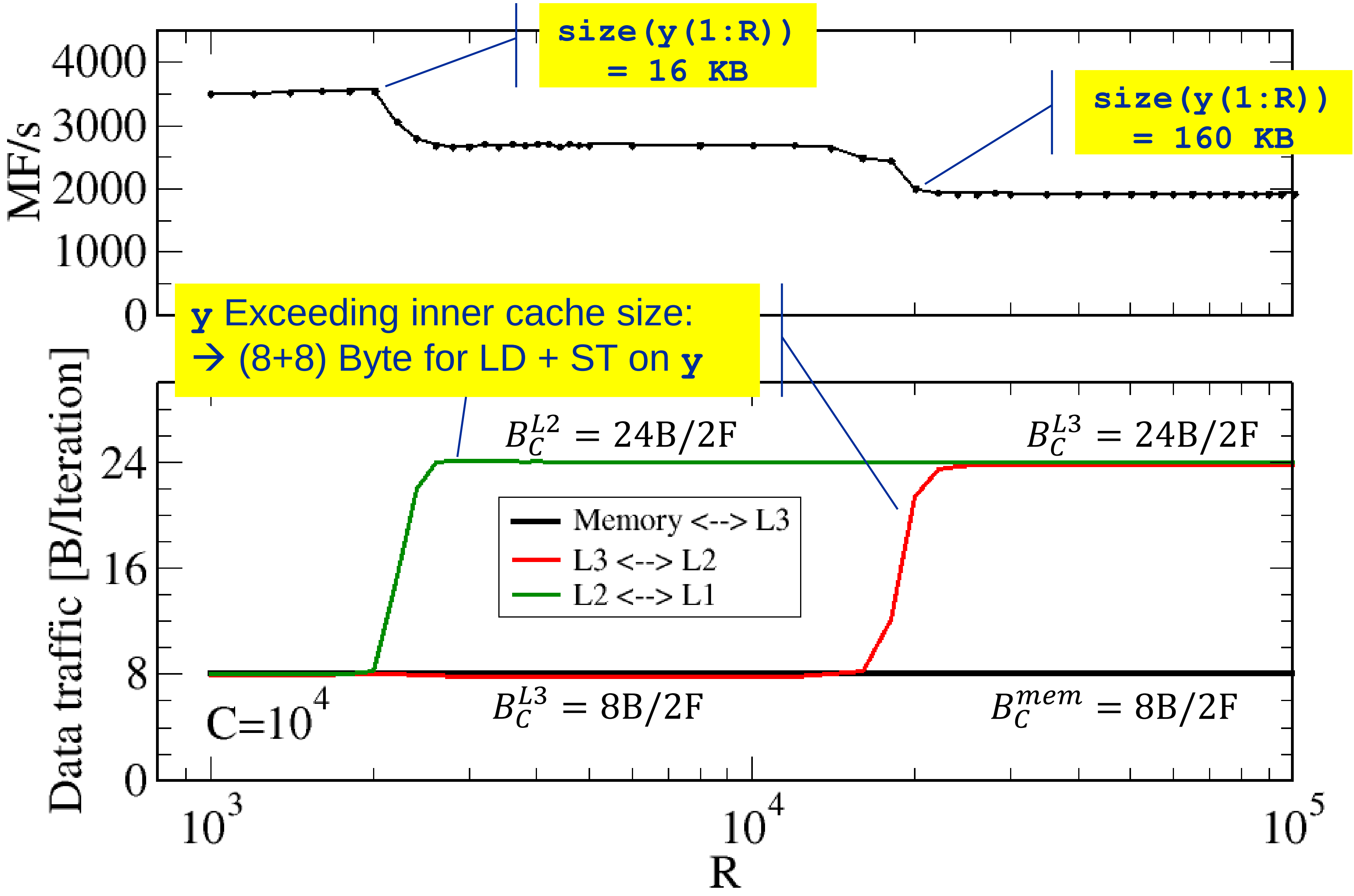
`y(:)` is loaded and stored in each outer iteration → for c>1 update `y(:)` in cache

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



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

DMVM (DP) – Single core data traffic analysis

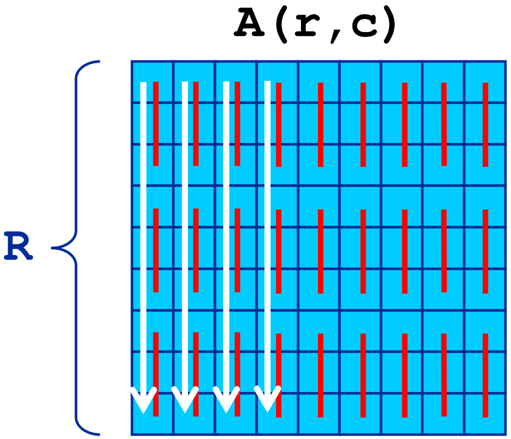


`size(y(1:R))`
`= 16 KB`

`size(y(1:R))`
`= 160 KB`

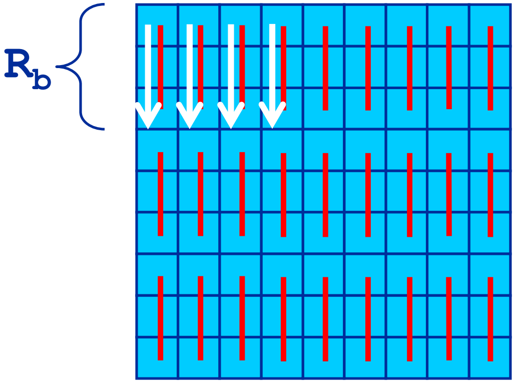
y Exceeding inner cache size:
→ (8+8) Byte for LD + ST on **y**

Reducing traffic by blocking



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

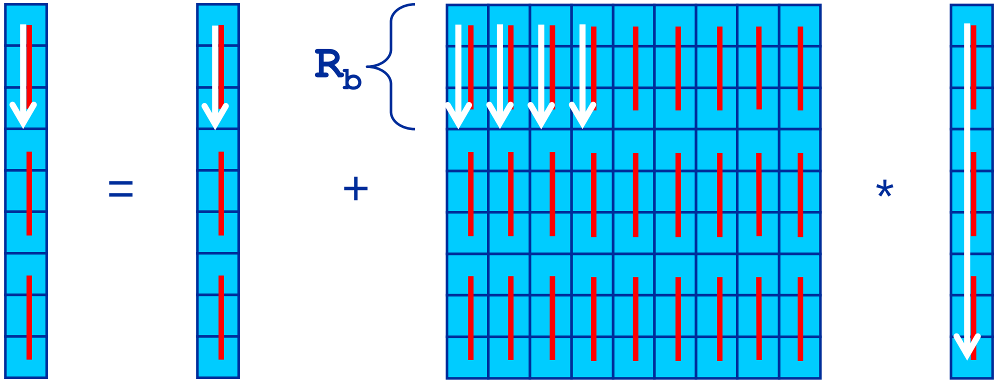
`y(:)` may not fit into some cache → more traffic for lower level



```
do rb = 1 , R , R_b
 rbS = rb
 rbE = min((rb+R_b-1), R)
 do c = 1 , C
   do r = rbS , rbE
      y(r)=y(r) + A(r,c)*x(c)
   enddo
 enddo
enddo
```

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

Reducing traffic by blocking

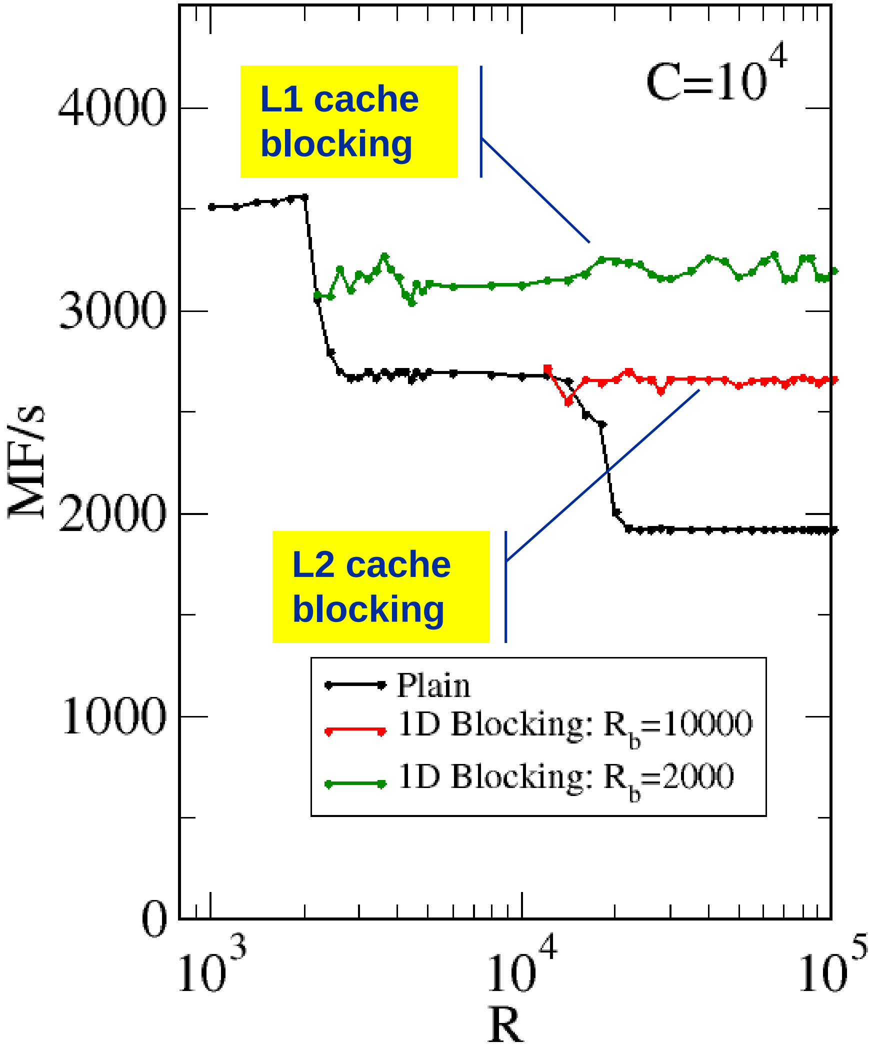


- **LHS only updated once in some cache level if blocking is applied**
- **Price: RHS is loaded multiple times instead of once!**
  - How often? → R / R_b times

- **Consequence: Traffic reduction of LHS & RHS by a factor of R / (R_b x 2C)**
  - Still a large reduction if block size can be made larger than about 10

DMVM (DP) – Reducing traffic by inner loop blocking

optional

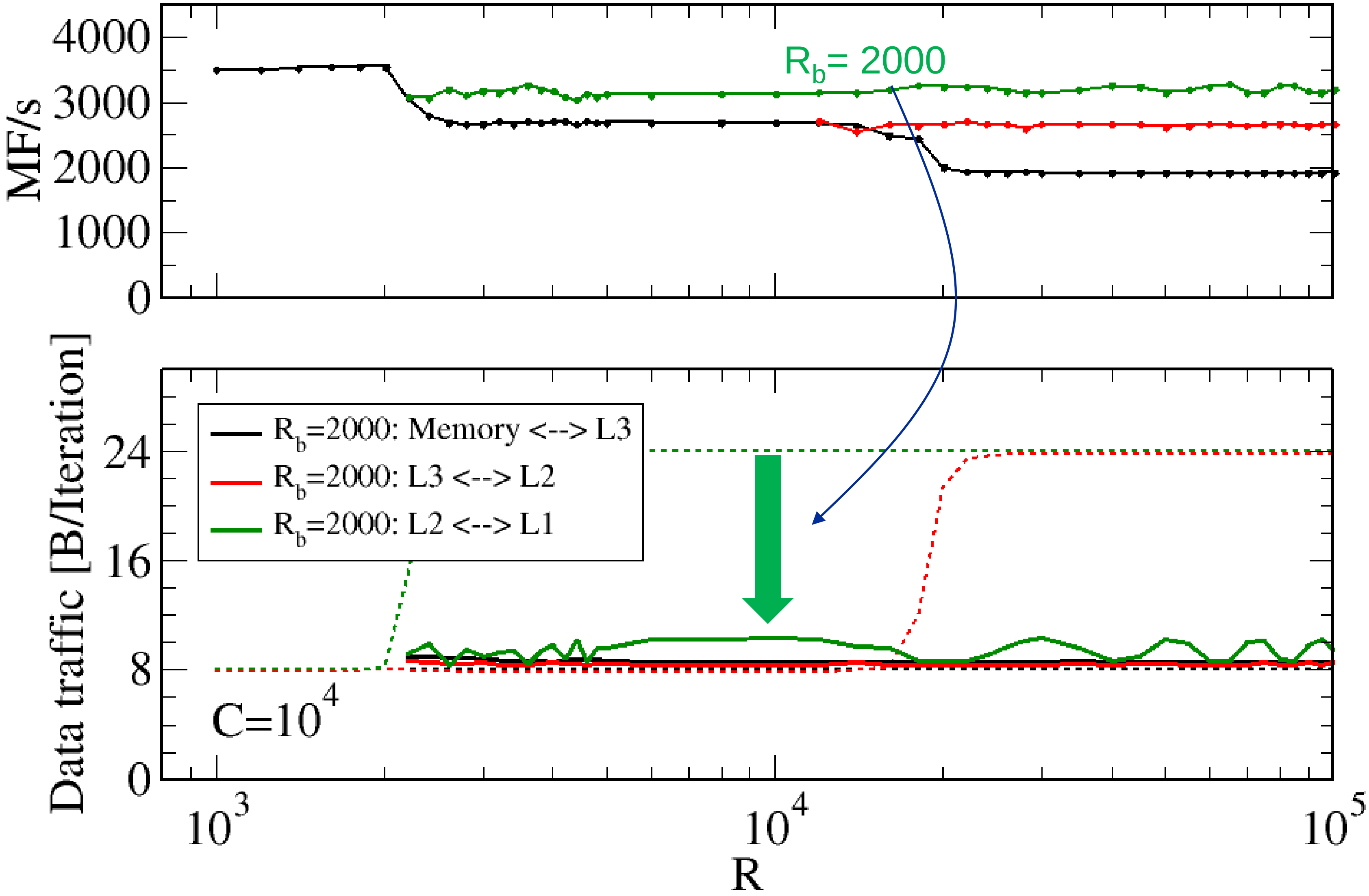


- “1D blocking” for inner loop
- Blocking factor R_b ←→ cache level

```
do rb = 1 , R , Rb

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

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

enddo
```

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

DMVM (DP) – Validation of blocking optimization



DMVM (DP) – OpenMP parallelization

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

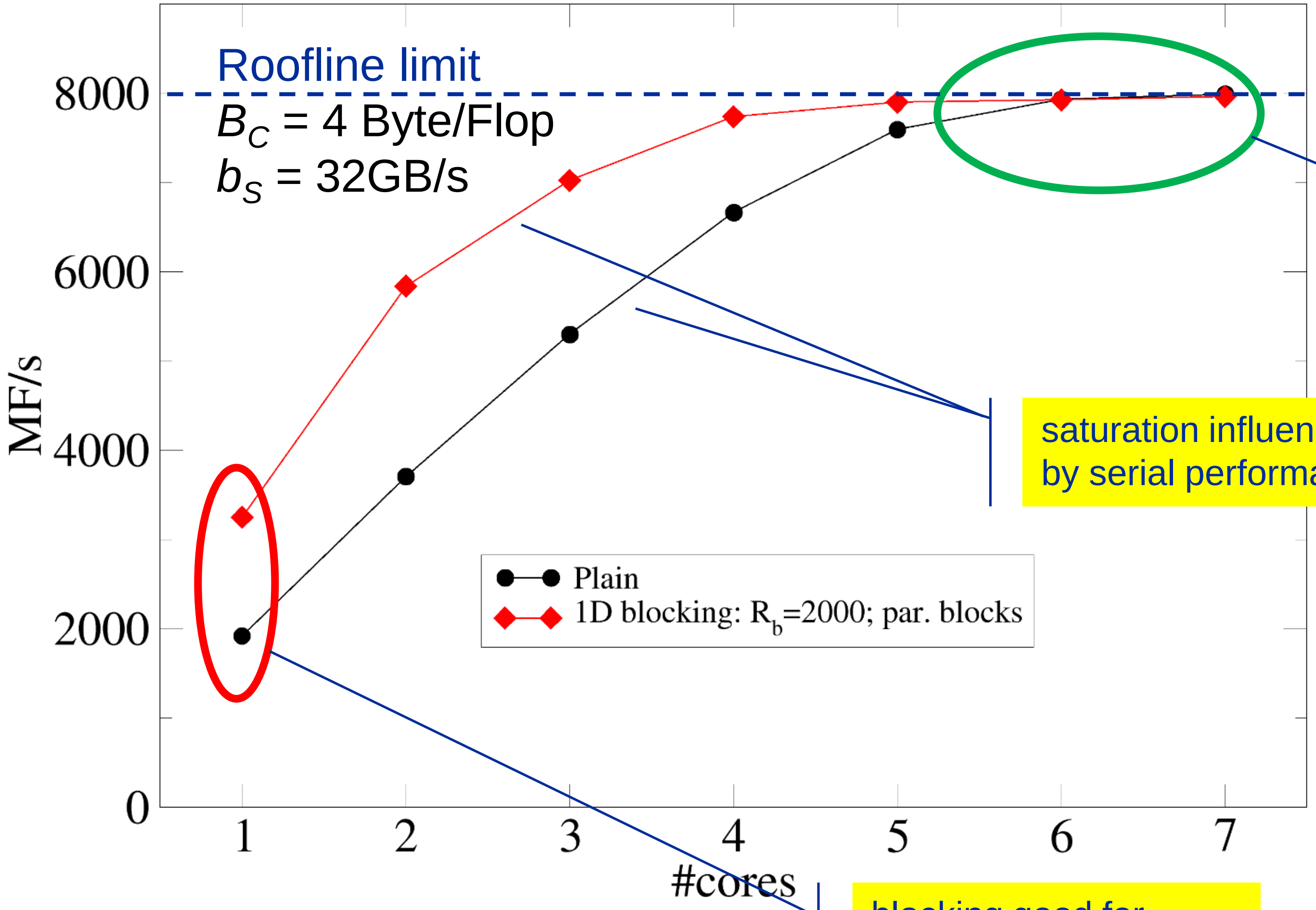
plain code

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

blocked code

DMVM (DP) – OpenMP parallelization & saturation

“Using more cores can heal bad single-core performance”



memory traffic unchanged → saturation unchanged!

saturation influenced by serial performance

blocking good for single thread performance (reduced in-cache traffic)

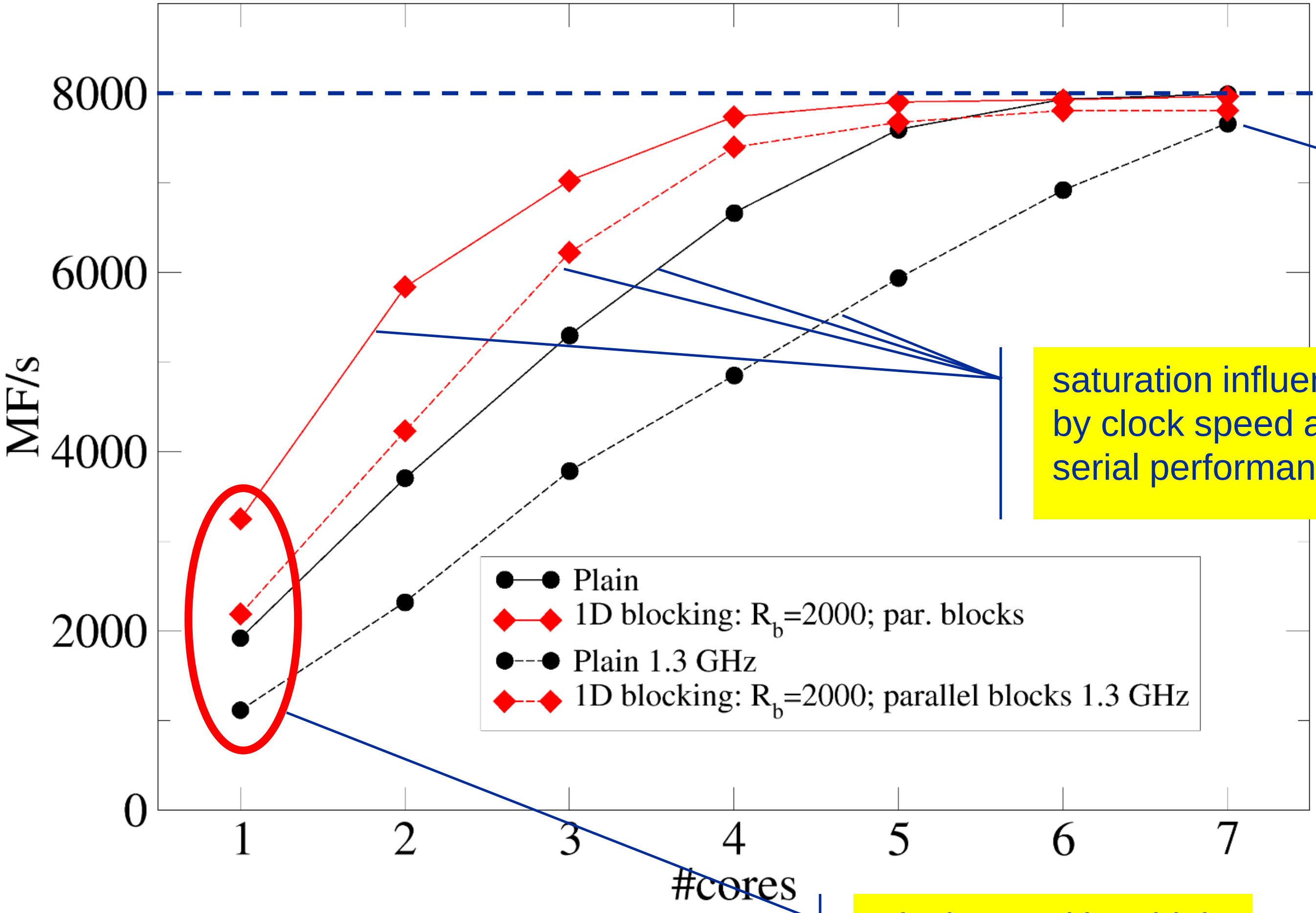
So, is blocking useless?
→ NO (see later)

Can we do anything to improve B_C^{mem}?
→ NO, not here

Intel Xeon E5 2695 v3 (Haswell-EP)
2.3 GHz base clock speed, $\boldsymbol{b_S}$ **= 32 GB/s**

DMVM (DP) – saturation & clock speed

optional



Even the worst code almost saturates the bandwidth @1.3 GHz

saturation influenced by clock speed and serial performance

Clock speed has high impact on single thread performance

Clock speed or parallelism can "heal" slow code!

Intel Xeon E5 2695 v3 (Haswell-EP)
2.3 GHz base clock speed, $\boldsymbol{b_S}$ **= 32 GB/s**

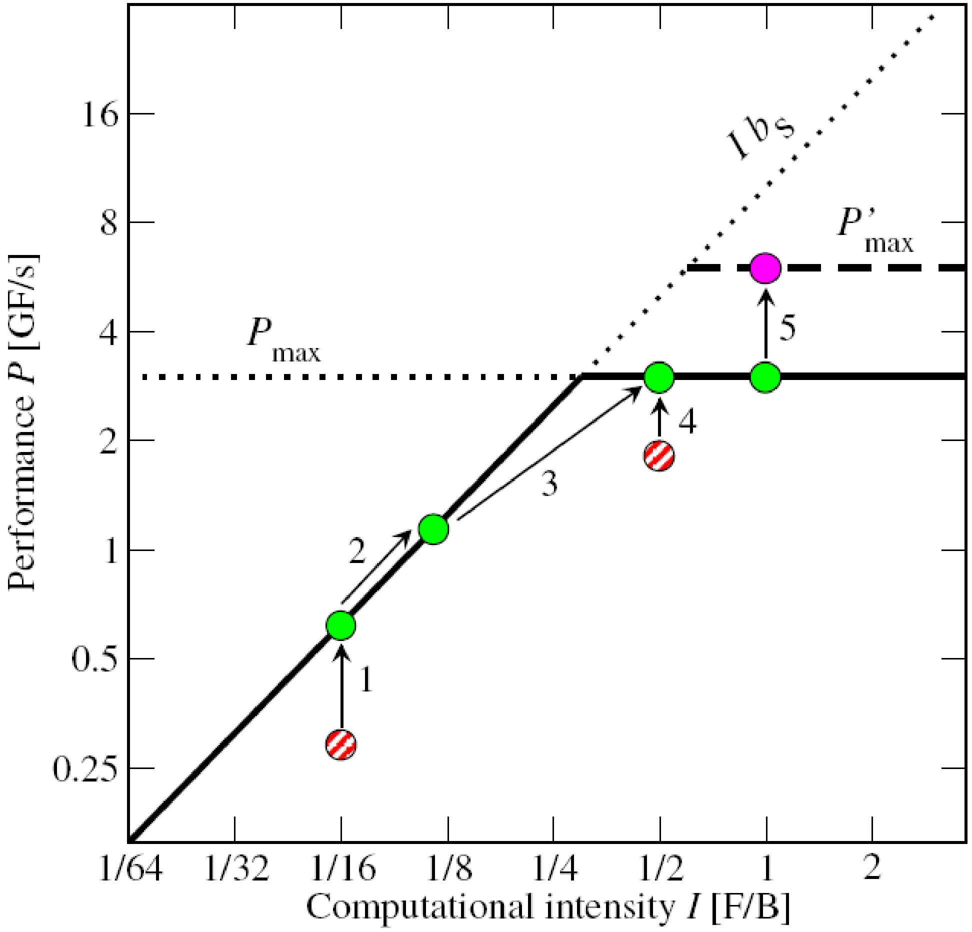
Conclusions from the dMVM example

optional

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

Typical code optimizations in the Roofline Model

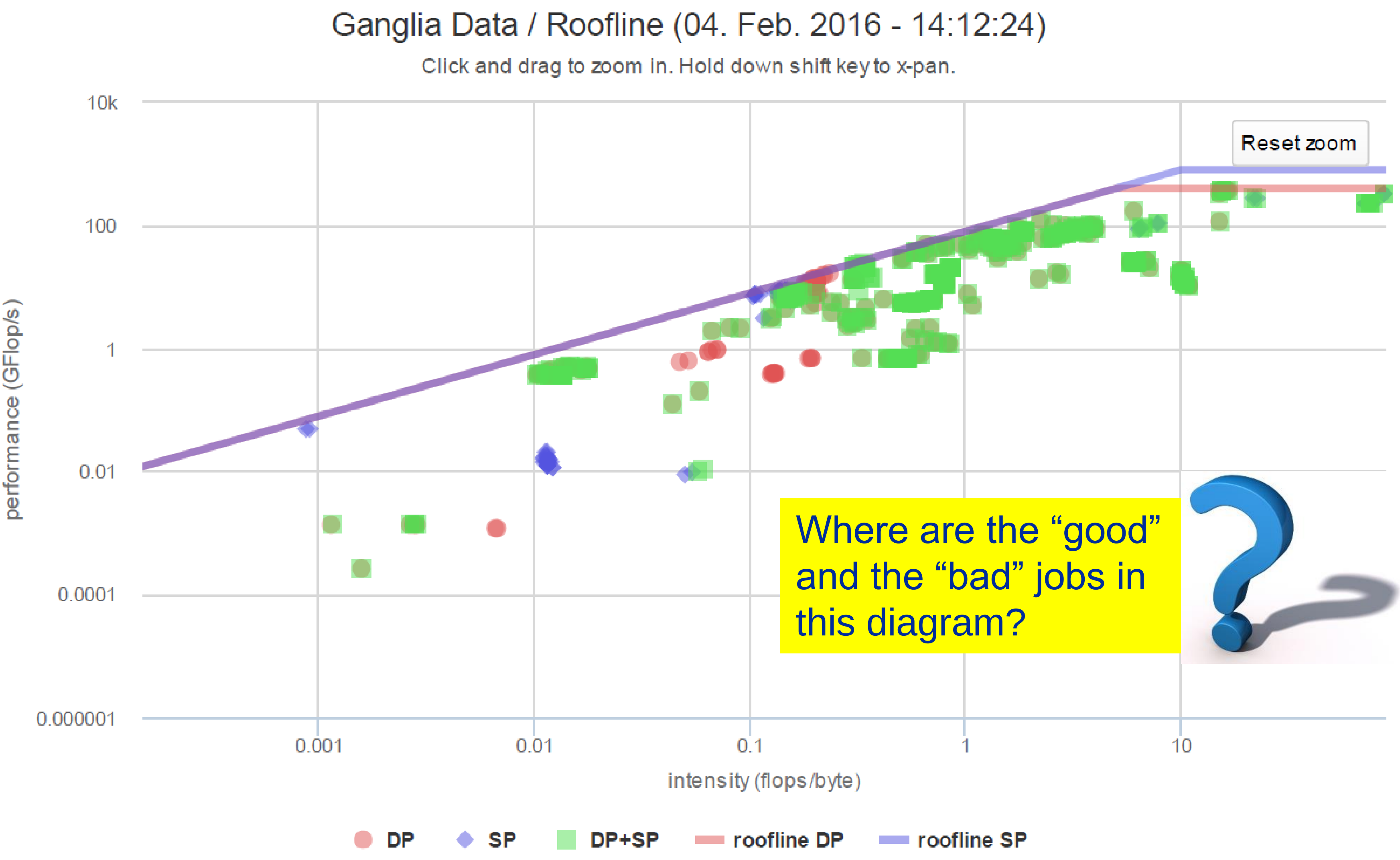
1. Hit the BW bottleneck by good serial code
   (e.g., Perl → Fortran)
2. Increase intensity to make better use of BW bottleneck
   (e.g., loop blocking → see later)
3. Increase intensity and go from memory-bound to core-bound
   (e.g., temporal blocking)
4. Hit the core bottleneck by good serial code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



Using Roofline for monitoring “live” jobs on a cluster

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

optional



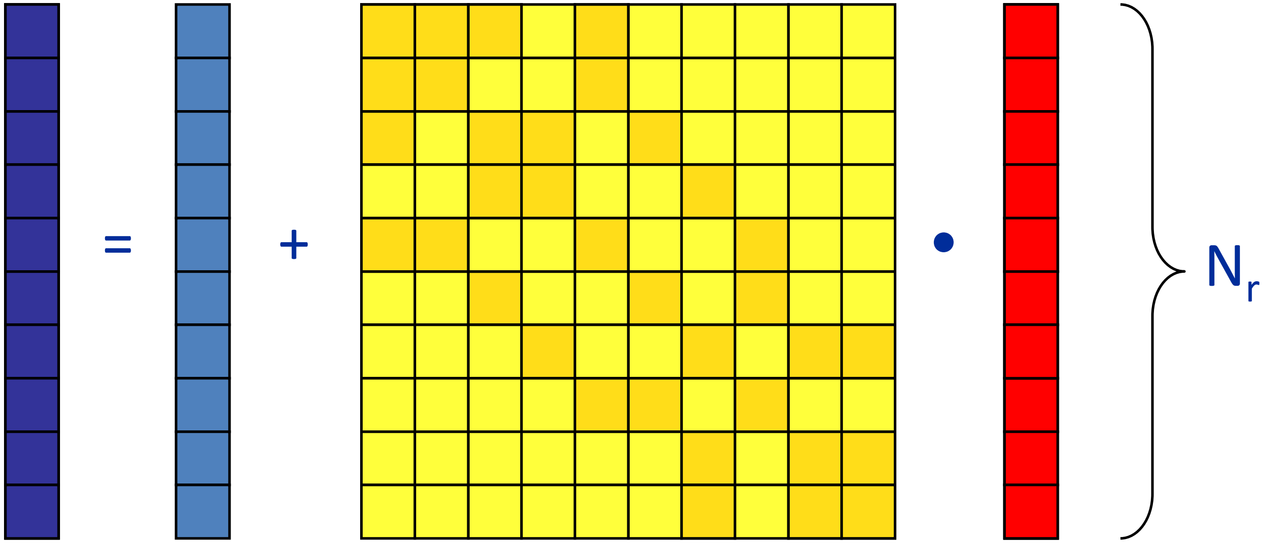
Where are the “good” and the “bad” jobs in this diagram?

Case study:
Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (spMVM)

- **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} ~ N_r**



General case: some indirect addressing required!

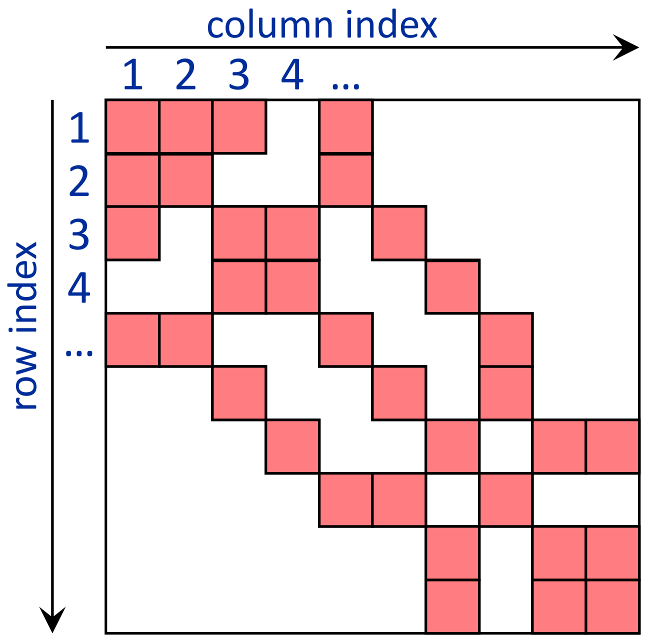
SpMVM characteristics

- **For large problems, spMVM is inevitably memory-bound**
  - Intra-socket saturation effect on modern multicores

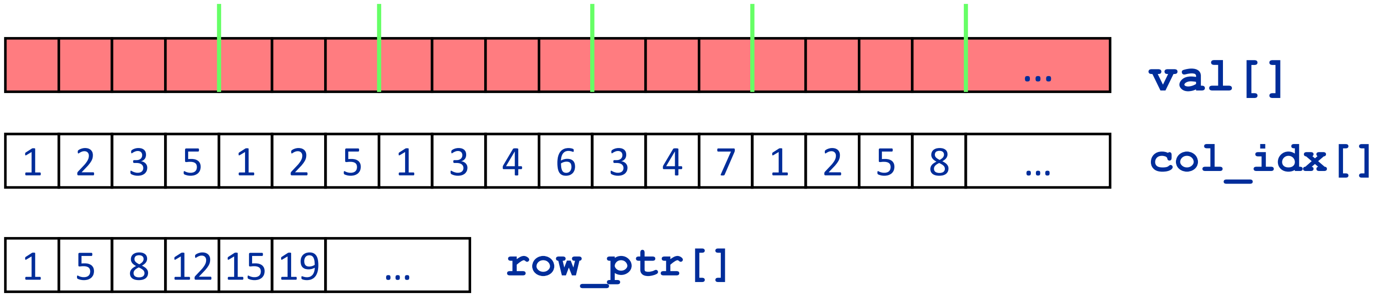
- **SpMVM is easily parallelizable in shared and distributed memory**

- **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)



																		…	**`val[]`**
1	2	3	5	1	2	5	1	3	4	6	3	4	7	1	2	5	8	…	**`col_idx[]`**

1	5	8	12	15	19	…	**`row_ptr[]`**

Case study: Sparse matrix-vector multiply

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

```
!$OMP parallel do
do i = 1,N_r
 do j = row_ptr(i), row_ptr(i+1) - 1
  c(i) = c(i) + val(j) * b(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

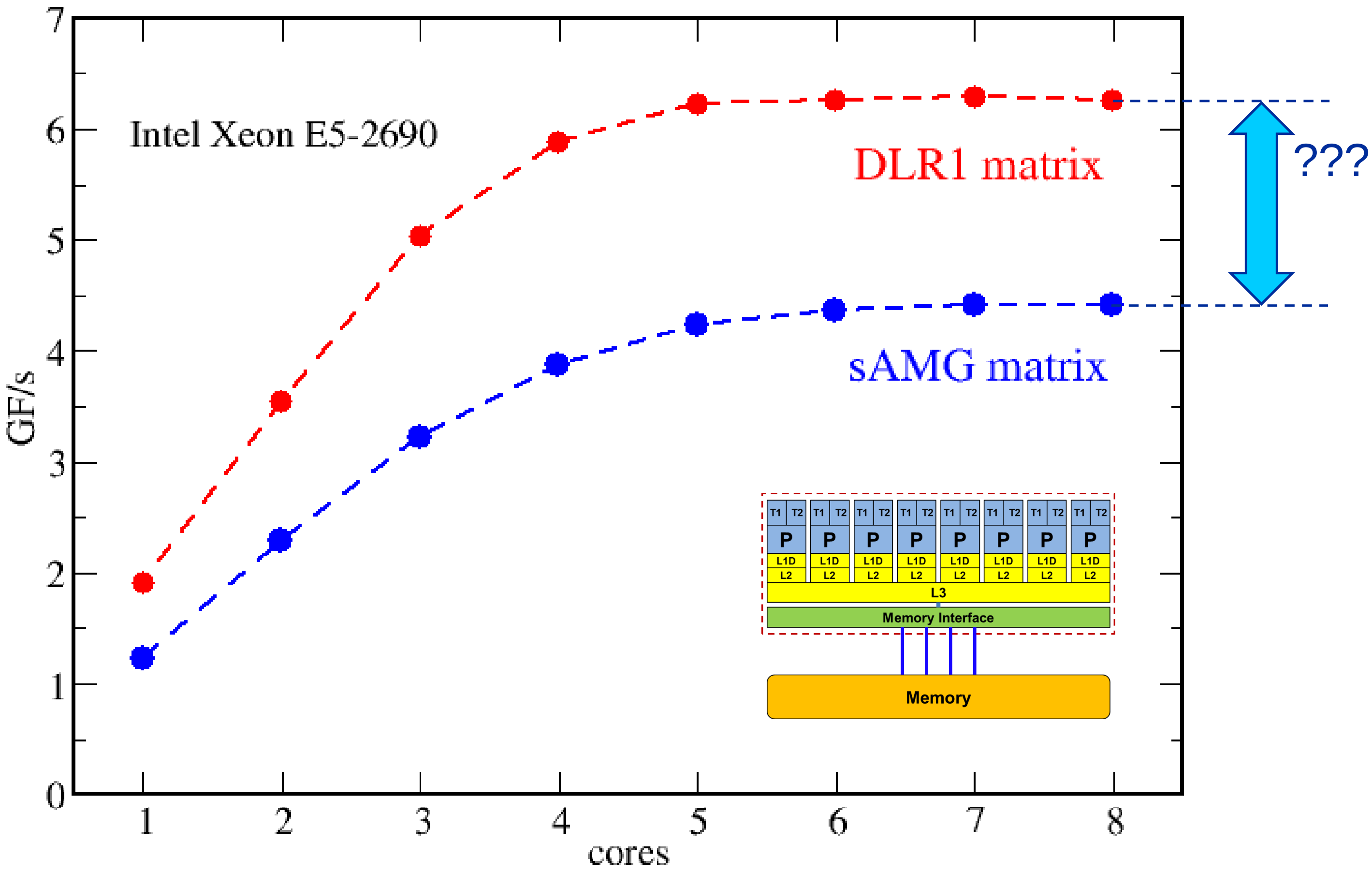
  - Usually many spMVMs required to solve a problem

- **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 spMVM?
- Optimization?



Example: SpMVM node performance model

- Sparse MVM in double precision w/ CRS data storage:

```
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
```

- DP CRS comp. intensity

$$I_{CRS}^{DP} = \frac{2}{\underbrace{\boxed{8} + \boxed{4} + \boxed{8\alpha} + \boxed{16/N_{nzr}}}} \frac{\text{flops}}{\text{byte}}$$

"best" in-memory intensity: $I = \frac{1}{6}\,\mathrm{F/B}$, or $B_C^{mem} = 6\ \mathrm{B/F}$

  - α quantifies traffic for loading RHS
    - $\alpha = 0$ → RHS is in cache
    - $\alpha = 1/N_{nzr}$ → RHS loaded once
    - $\alpha = 1$ → no cache
    - $\alpha > 1$ → Houston, we have a problem!
  - "Expected" performance = b_S x I_{CRS}
  - Determine α by measuring performance and actual memory traffic
    - Maximum memory BW may not be achieved with spMVM

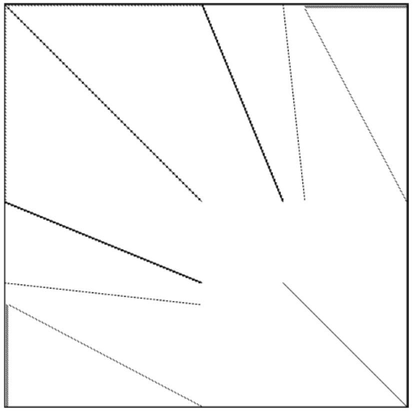
Determine RHS traffic

$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \frac{\text{flops}}{\text{byte}} = \frac{N_{nz} \cdot 2\ \text{flops}}{V_{meas}}$$

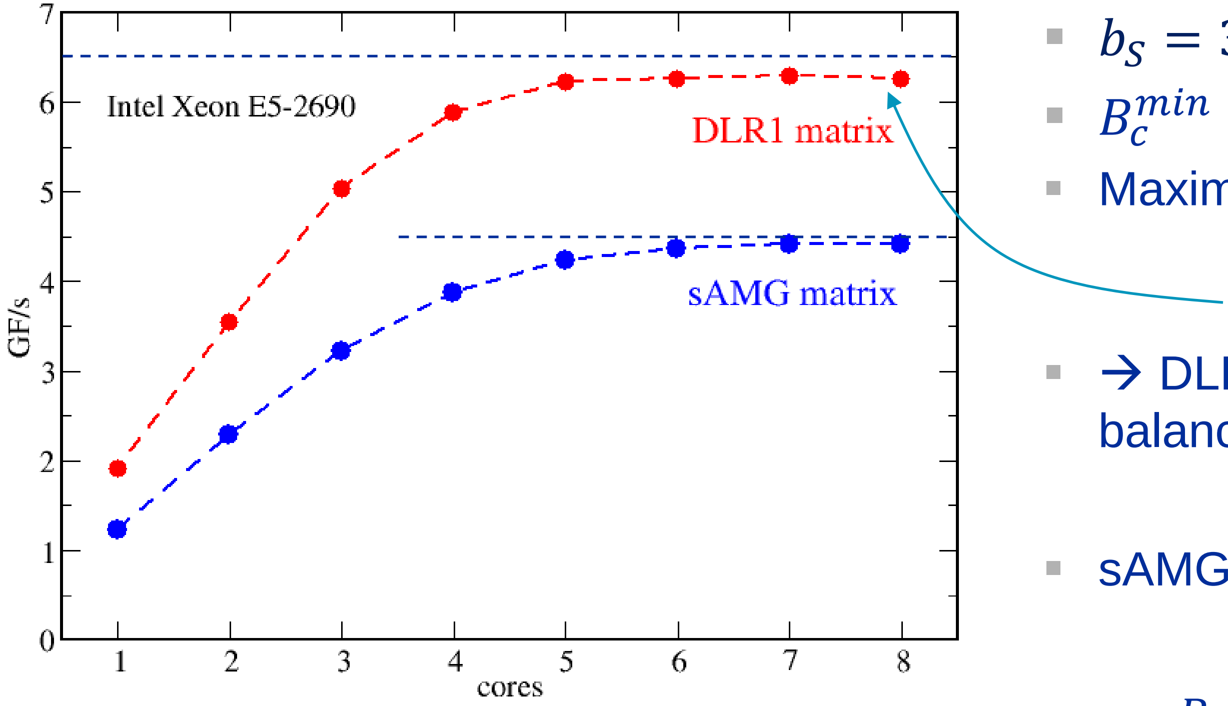
- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α: $$\alpha = \frac{1}{4}\left(\frac{V_{meas}}{N_{nz} \cdot 2\ \text{bytes}} - 6 - \frac{8}{N_{nzr}}\right)$$

- Example: kkt_power matrix from the UoF collection on one Intel SNB socket
  - $N_{nz} = 14.6 \cdot 10^6, N_{nzr} = 7.1$
  - $V_{meas} \approx 258\ \text{MB}$
  - → $\alpha = 0.43$, $\alpha N_{nzr} = 3.1$
  - → RHS is loaded 3.1 times from memory
  - and: $$\frac{I_{CRS}^{DP}(1/N_{nzr})}{I_{CRS}^{DP}(\alpha)} = 1.15$$

15% extra traffic → optimization potential!



Now back to the start…



- $b_S = 39\,\mathrm{GB/s}$
- $B_c^{min} = 6\,\mathrm{B/F}$
- Maximum spMVM performance:

$$P_{max} = 6.5\,\mathrm{GF/s}$$

- → DLR1 causes minimum code balance!
- sAMG matrix code balance:

$$B_c \leq \frac{b_S}{4.5\,\mathrm{GF/s}} = 8.7\,\mathrm{B/F}$$

- Why is this only an upper limit?
- What is the next step?
- Could we have predicted this qualitative difference?

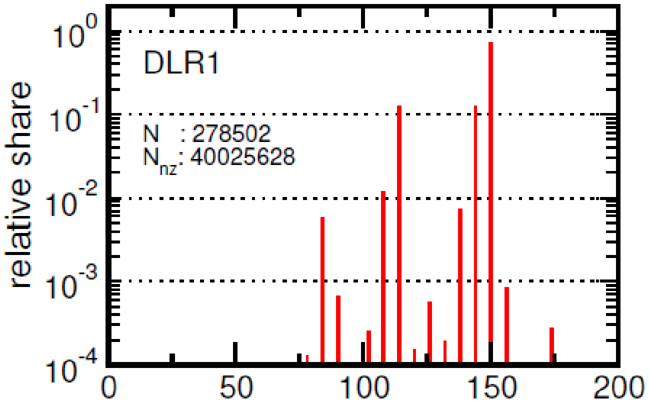
Sparse matrix testcases

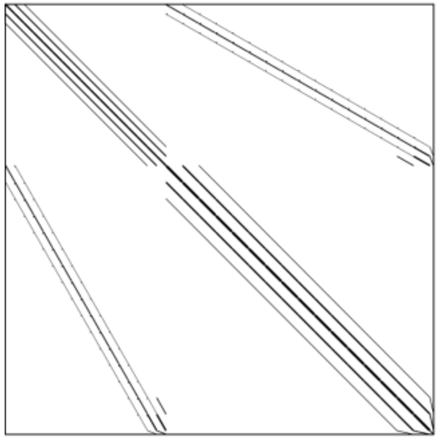
optional

“DLR1” (A. Basermann, DLR)

Adjoint problem computation (turbulent transonic flow over a wing) with the TAU CFD system of the German Aerospace Center (DLR)

Avg. non-zeros/row ~150

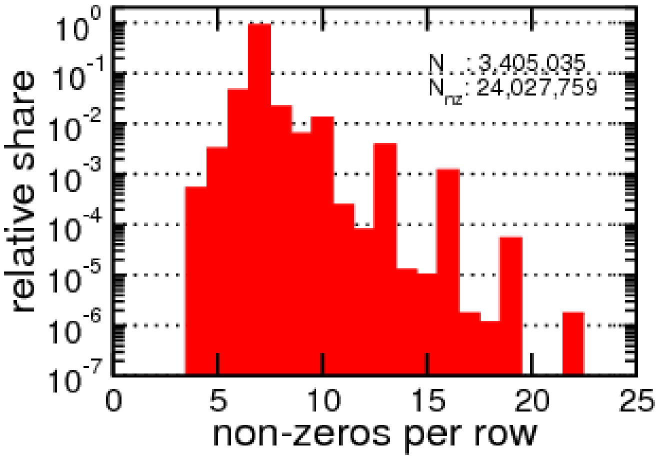


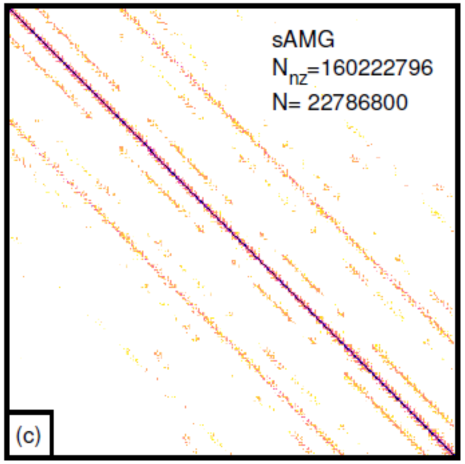


“sAMG” (K. Stüben, FhG-SCAI)

Matrix from FhG’s adaptive multigrid code sAMG for the irregular discretization of a Poisson problem on a car geometry.

Avg. non-zeros/row ~ 7

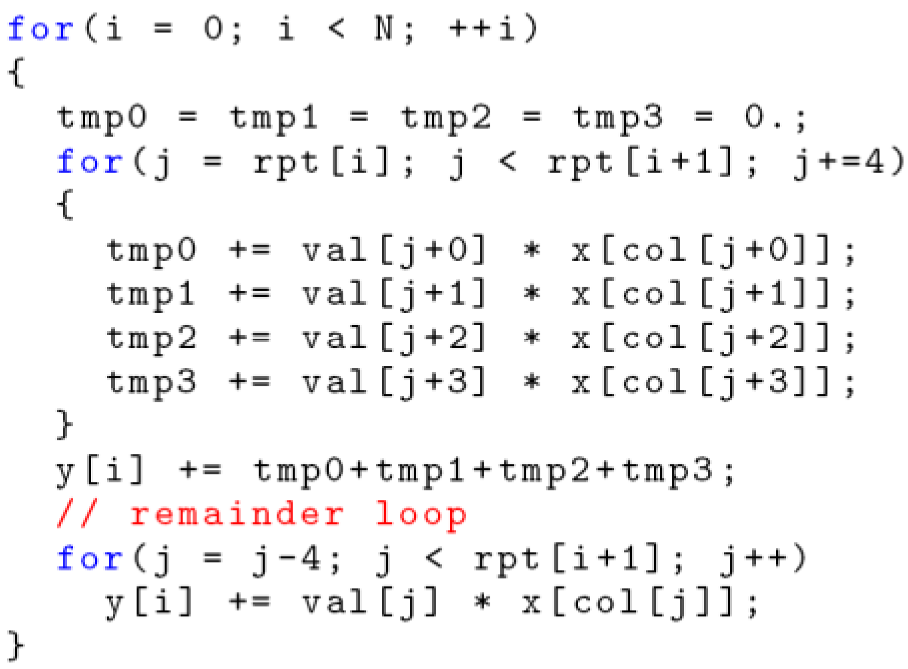




What about SIMD for spMVM?

CRS on wide SIMD units

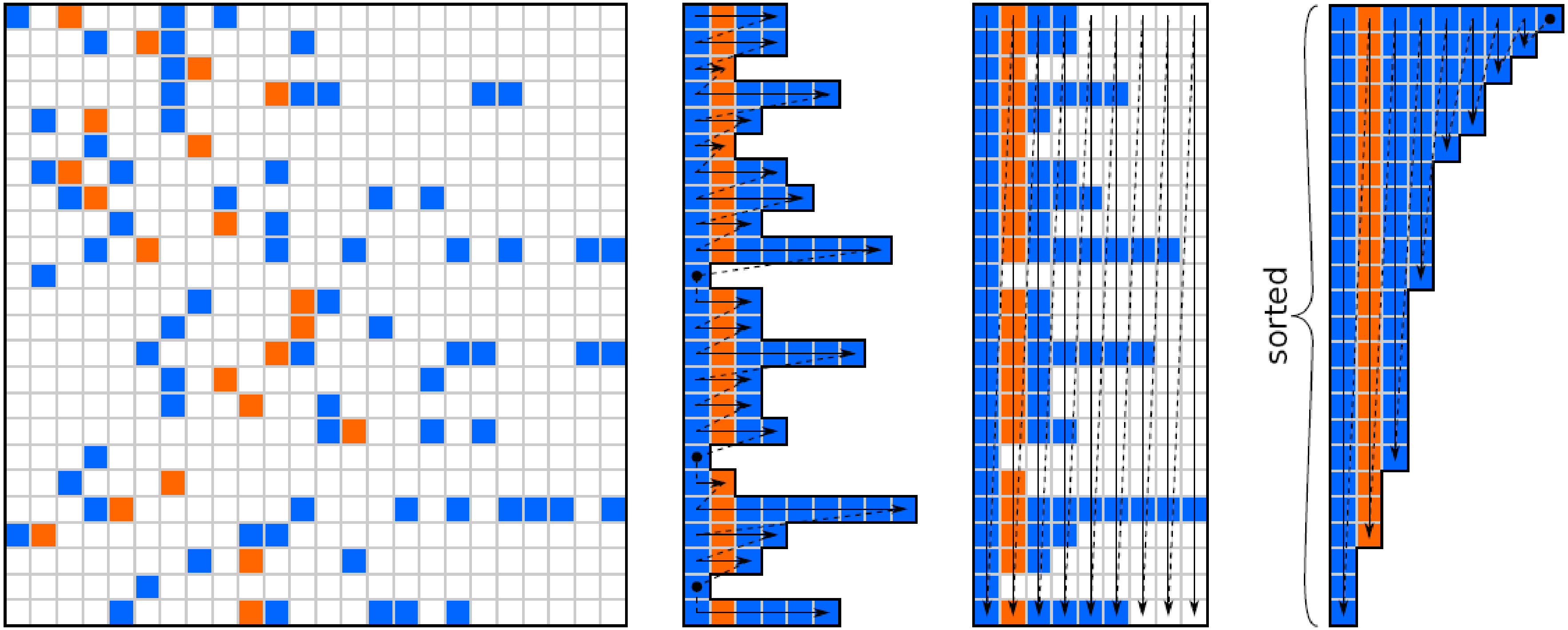
- **When number of nonzeros per row is small**
  - Significant remainder loop overhead (partial/no vectorization)
  - Large impact of row reduction
  - Alignment constraints require additional peeling
  - Small loop count
- **GPGPUs**
  - One warp per row: similar problems
  - Outer loop parallelization: Loss of coalescing



```
for(i = 0; i < N; ++i)
{
   tmp0 = tmp1 = tmp2 = tmp3 = 0.;
   for(j = rpt[i]; j < rpt[i+1]; j+=4)
   {
      tmp0 += val[j+0] * x[col[j+0]];
      tmp1 += val[j+1] * x[col[j+1]];
      tmp2 += val[j+2] * x[col[j+2]];
      tmp3 += val[j+3] * x[col[j+3]];
   }
   y[i] += tmp0+tmp1+tmp2+tmp3;
   // remainder loop
   for(j = j-4; j < rpt[i+1]; j++)
      y[i] += val[j] * x[col[j]];
}
```

- **Can we find a format that is better suited for SIMD?**
- **Is it relevant at all if the execution is memory bound?**

Standard sparse matrix storage formats



original matrix CRS ELLPACK JDS

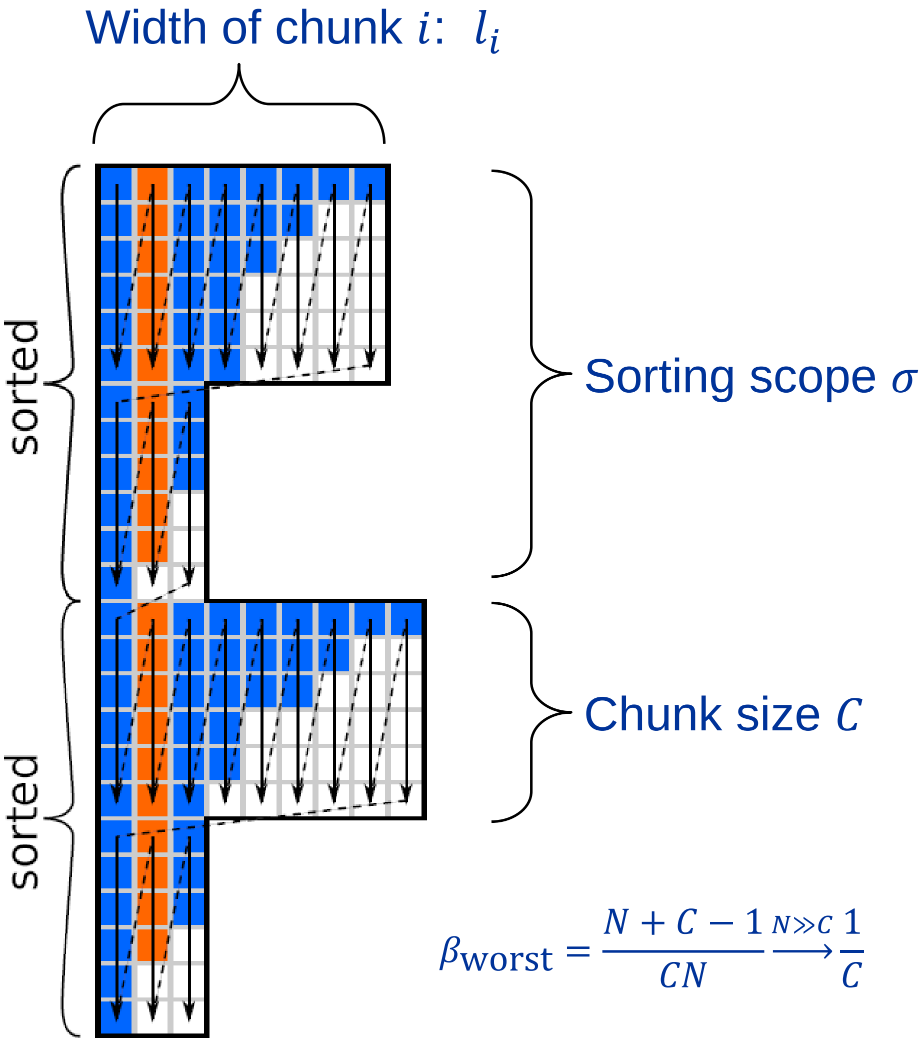
Constructing SELL-C-σ

Best of all worlds

1. Pick chunk size C (guided by SIMD/T widths)
2. Pick sorting scope σ
3. Sort rows by length within each sorting scope
4. Pad chunks with zeros to make them rectangular
5. Store matrix data in “chunk column major order”

“Chunk occupancy”: fraction of “useful” matrix entries

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



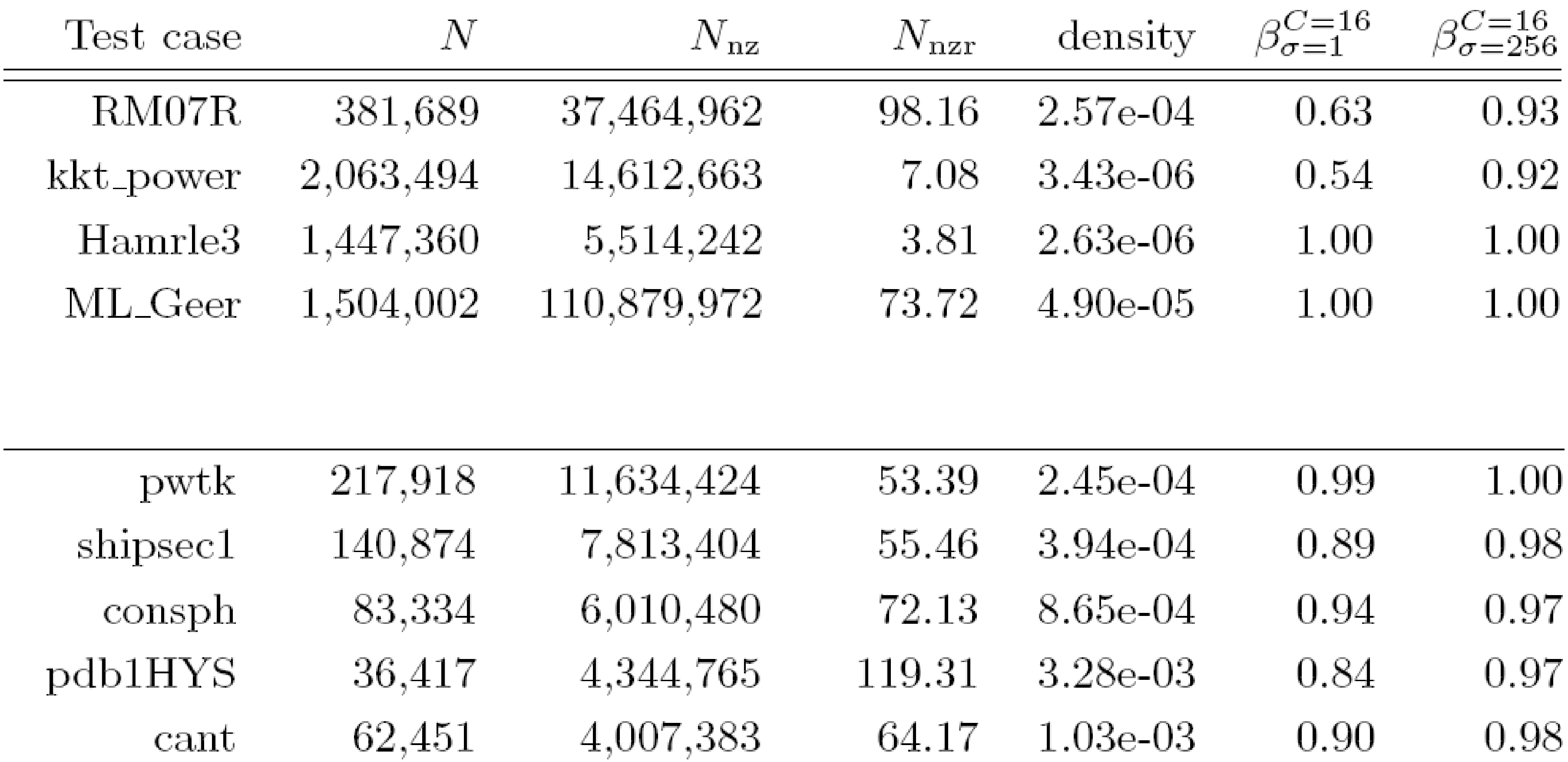
$$\beta_{\text{worst}} = \frac{N + C - 1}{CN} \xrightarrow{N \gg C} \frac{1}{C}$$

SELL-6-12
β=0.66

Matrix characterization

Improving β by row sorting

“Corner case” matrices from “Williams Group”:



Test case	N	N_{nz}	N_{nzr}	density	$\beta^{C=16}_{\sigma=1}$	$\beta^{C=16}_{\sigma=256}$
RM07R	381,689	37,464,962	98.16	2.57e-04	0.63	0.93
kkt_power	2,063,494	14,612,663	7.08	3.43e-06	0.54	0.92
Hamrle3	1,447,360	5,514,242	3.81	2.63e-06	1.00	1.00
ML_Geer	1,504,002	110,879,972	73.72	4.90e-05	1.00	1.00
pwtk	217,918	11,634,424	53.39	2.45e-04	0.99	1.00
shipsec1	140,874	7,813,404	55.46	3.94e-04	0.89	0.98
consph	83,334	6,010,480	72.13	8.65e-04	0.94	0.97
pdb1HYS	36,417	4,344,765	119.31	3.28e-03	0.84	0.97
cant	62,451	4,007,383	64.17	1.03e-03	0.90	0.98

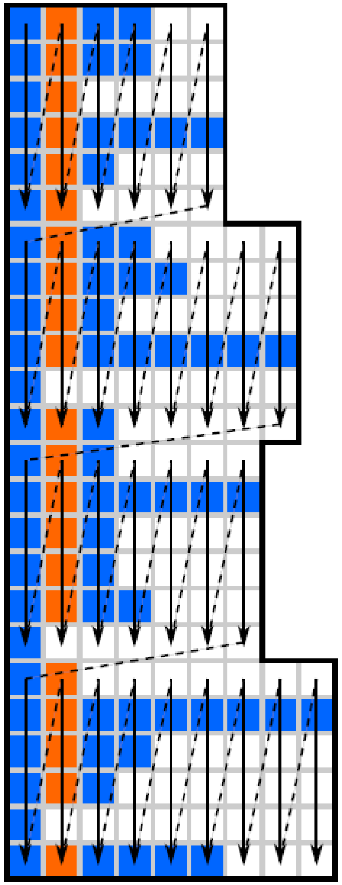
…

SELL-C-σ kernel

```
for(i = 0; i < N/4; ++i)
{

  for(j = 0; j < cl[i]; ++j)
  {
    y[i*4+0] += val[cs[i]+j*4+0] *
                 x[col[cs[i]+j*4+0]];
    y[i*4+1] += val[cs[i]+j*4+1] *
                 x[col[cs[i]+j*4+1]];
    y[i*4+2] += val[cs[i]+j*4+2] *
                 x[col[cs[i]+j*4+2]];
    y[i*4+3] += val[cs[i]+j*4+3] *
                 x[col[cs[i]+j*4+3]];
  }
}
```

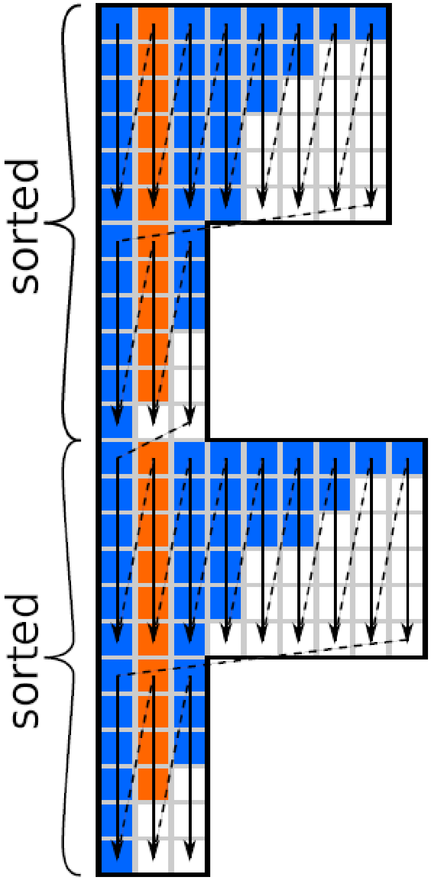
$C = 4$

Variants of SELL-C-σ



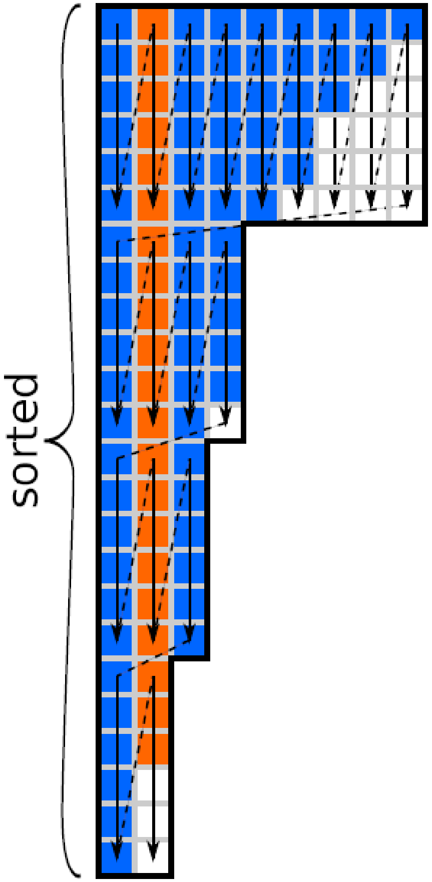
SELL-6-1

β=0.51



SELL-6-12

β=0.66



SELL-6-24

β=0.84

(SELL-1-1 == CRS / SELL-N-1 == ELLPACK)

Roofline performance model for SELL-C-σ

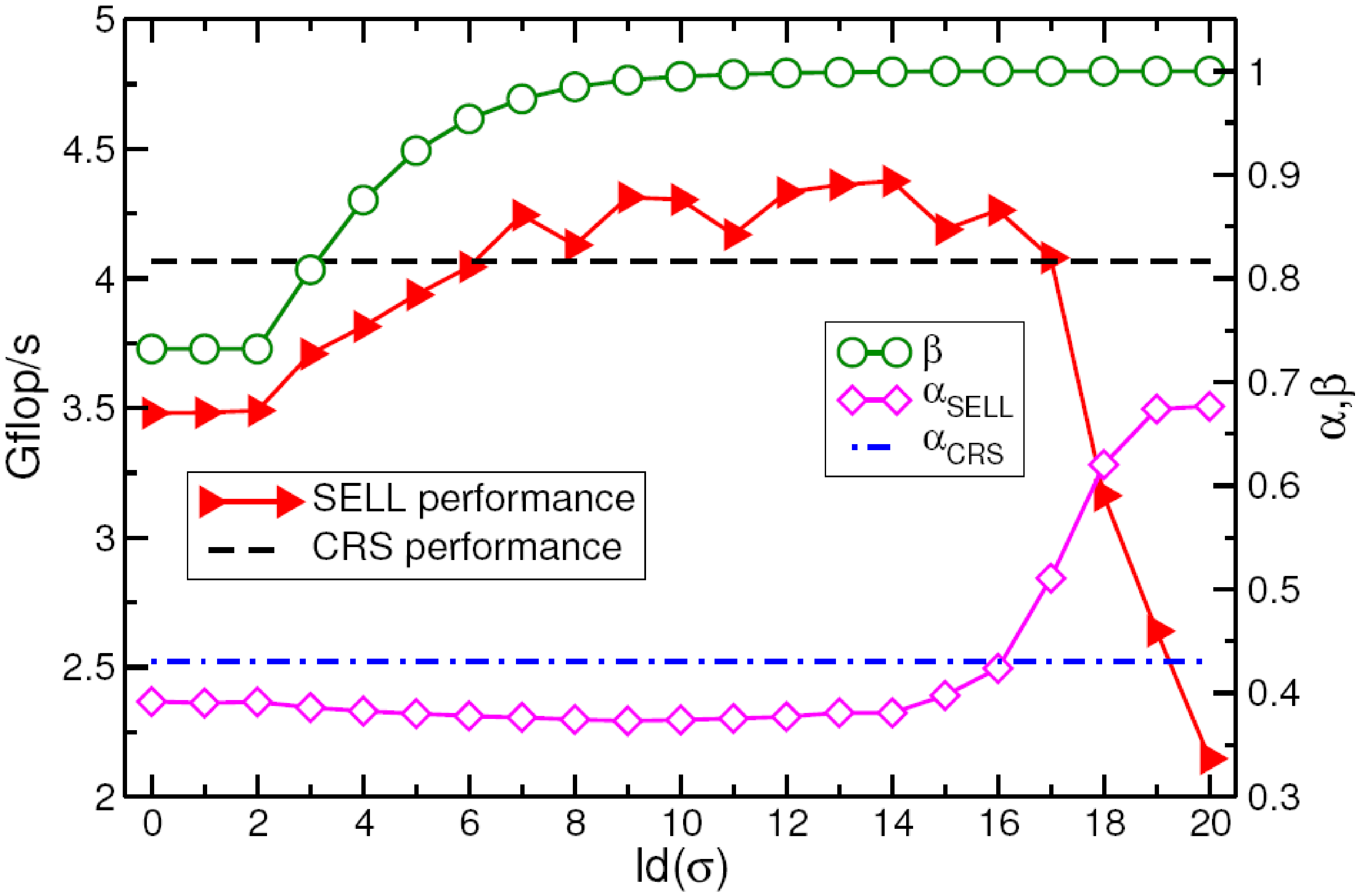
Code balance (double precision FP, 4-byte index):

$$B_{SELL}(\alpha,\beta,N_{nzr}) = \left(\overbrace{\boxed{\frac{1}{\beta}\left(\frac{8+4}{2}\right)}}^{\text{Matrix data \& column index}} + \frac{\overbrace{\boxed{8\alpha}}^{\text{RHS access}} + \overbrace{\boxed{16/N_{nzr}}}^{\text{LHS update}}}{2}\right)\frac{\text{bytes}}{\text{flop}}$$

$$= \left(\frac{6}{\beta} + 4\alpha + \frac{8}{N_{nzr}}\right)\frac{\text{bytes}}{\text{flop}}$$

$$P(\alpha,\beta,N_{nzr},b) = \frac{b}{B_{SELL}(\alpha,\beta,N_{nzr})}$$

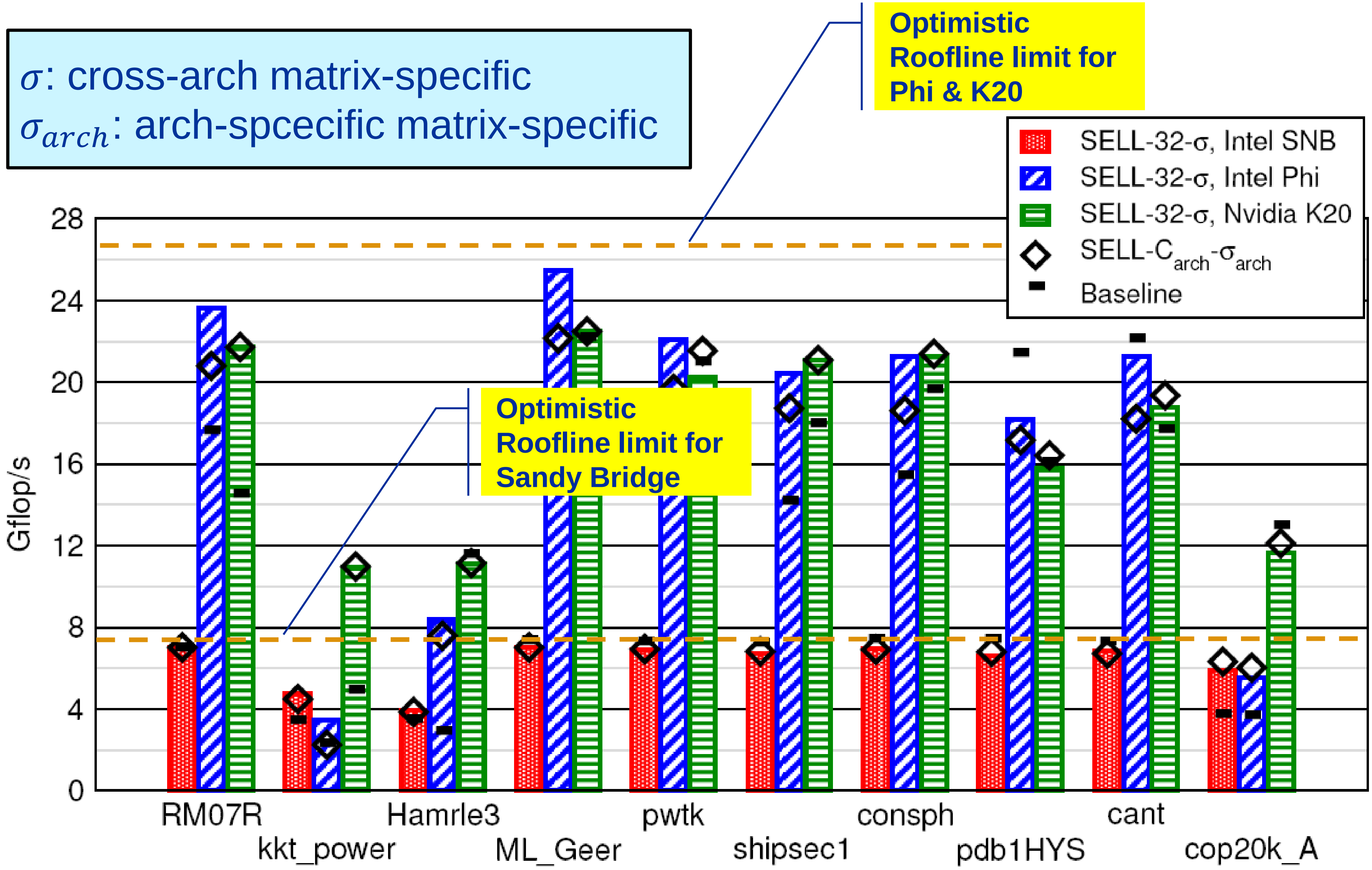
Impact of σ on α for kkt_power on SNB



Results for memory-bound matrices

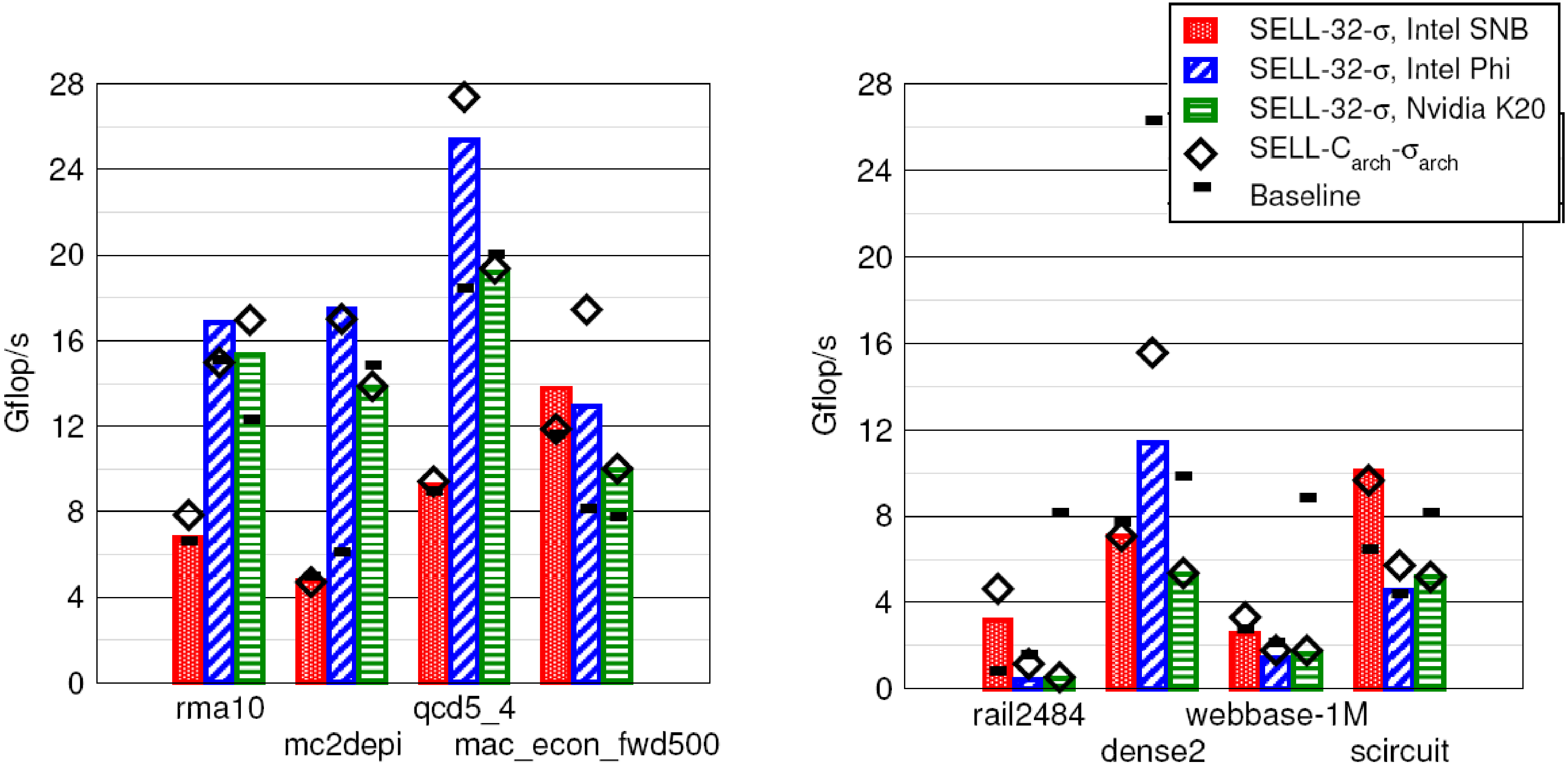
σ: cross-arch matrix-specific

σ_{arch}: arch-spcecific matrix-specific



Results for cache-bound and pathological cases

optional



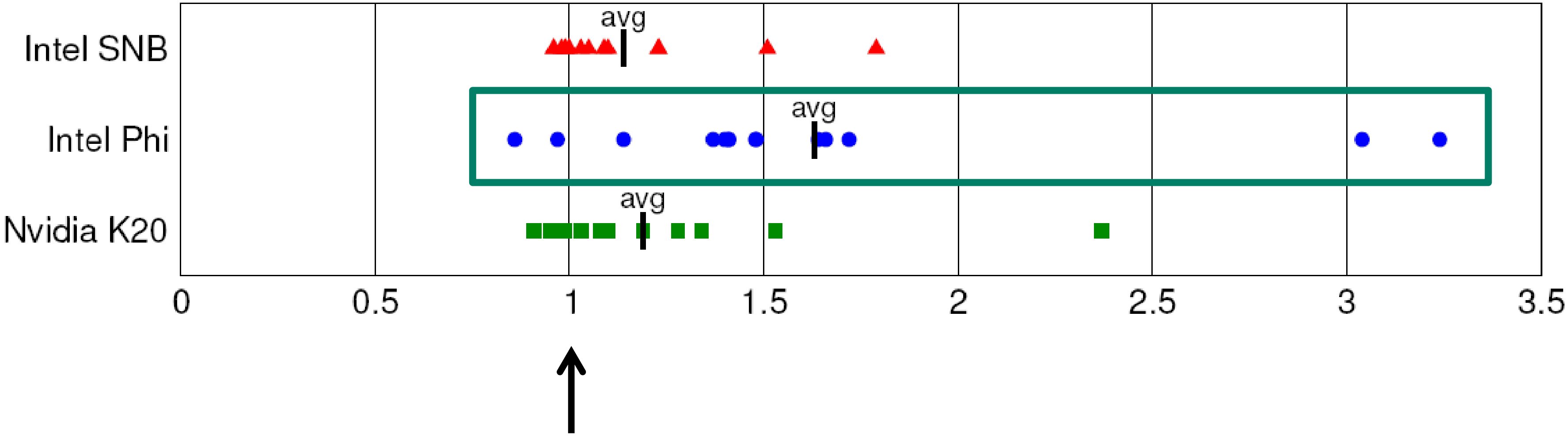
SELL-C-σ: Relative Performance Benefit

(16 square matrices)

optional

Generic SELL-32-σ format performance vs. baselines:

- CRS (from MKL) for SNB and Xeon Phi
- HYB (CUSPARSE) for K20



native format per architecture

Roofline analysis for spMVM

- **Conclusion from 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

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

- **SpMVM requires efficient data formats for best performance**
  - CRS OK for CPUs, SELL-C-σ & friends better for wide SIMD/manycore

Case study: A Jacobi smoother

The basic performance properties in 2D

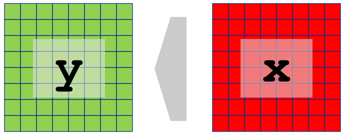
Layer conditions

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
```



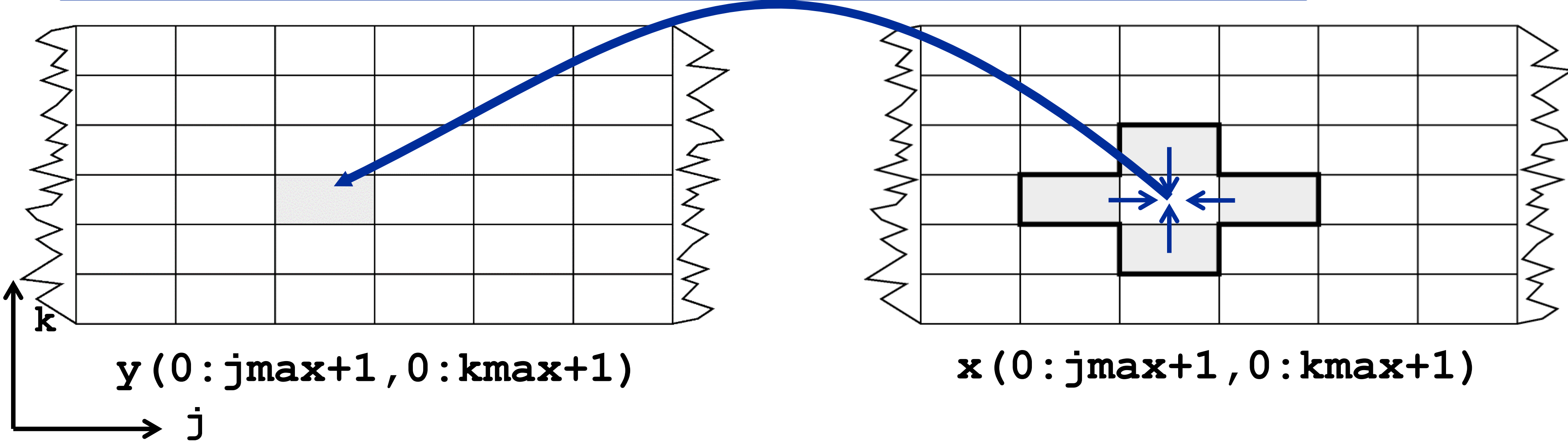
- Complexity of implementation and performance depends on
  - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type, …
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…

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
```

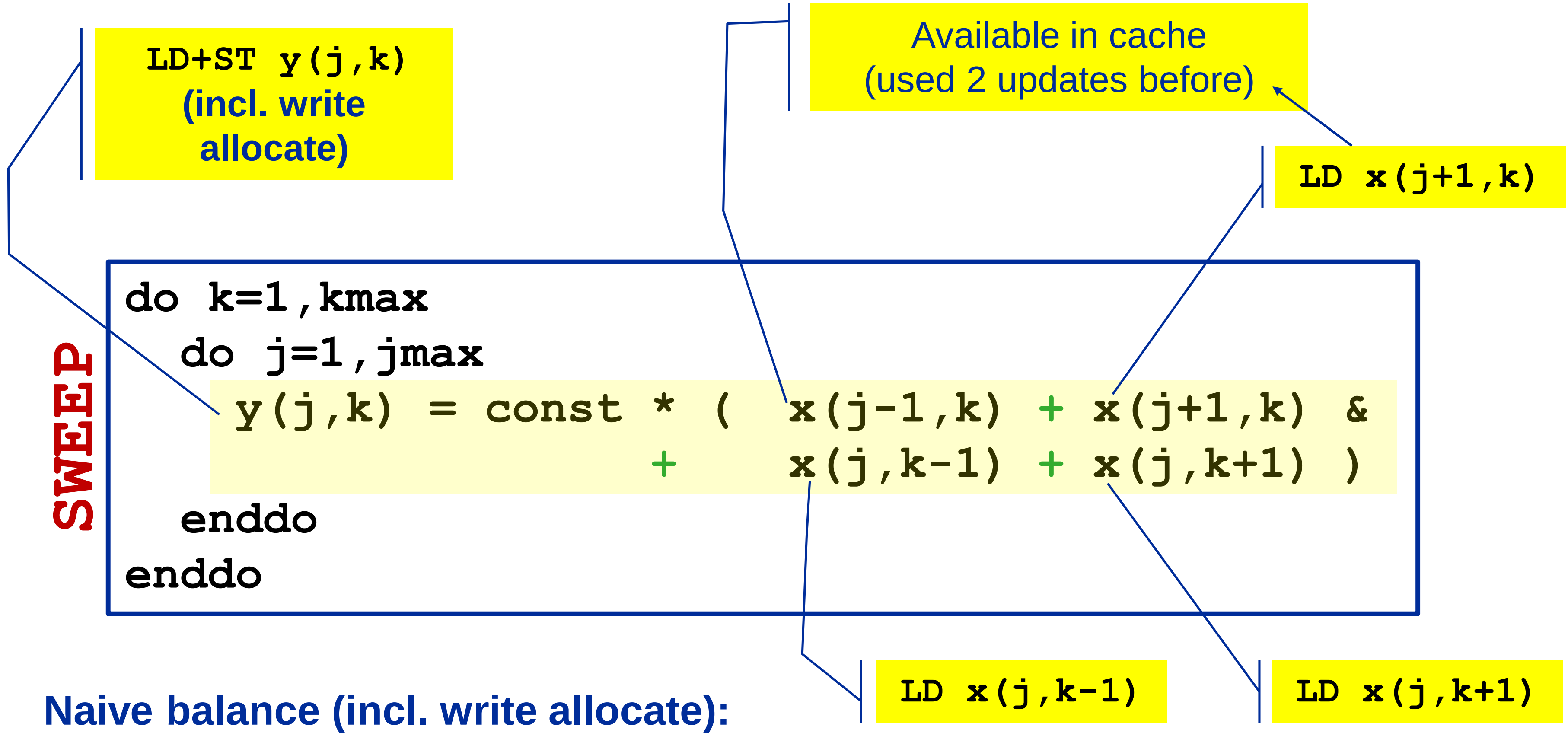
Lattice Update (LUP)



Appropriate performance metric: "**Lattice 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



LD+ST y(j,k)
(incl. write allocate)

Available in cache
(used 2 updates before)

LD x(j+1,k)

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
```

LD x(j,k-1)

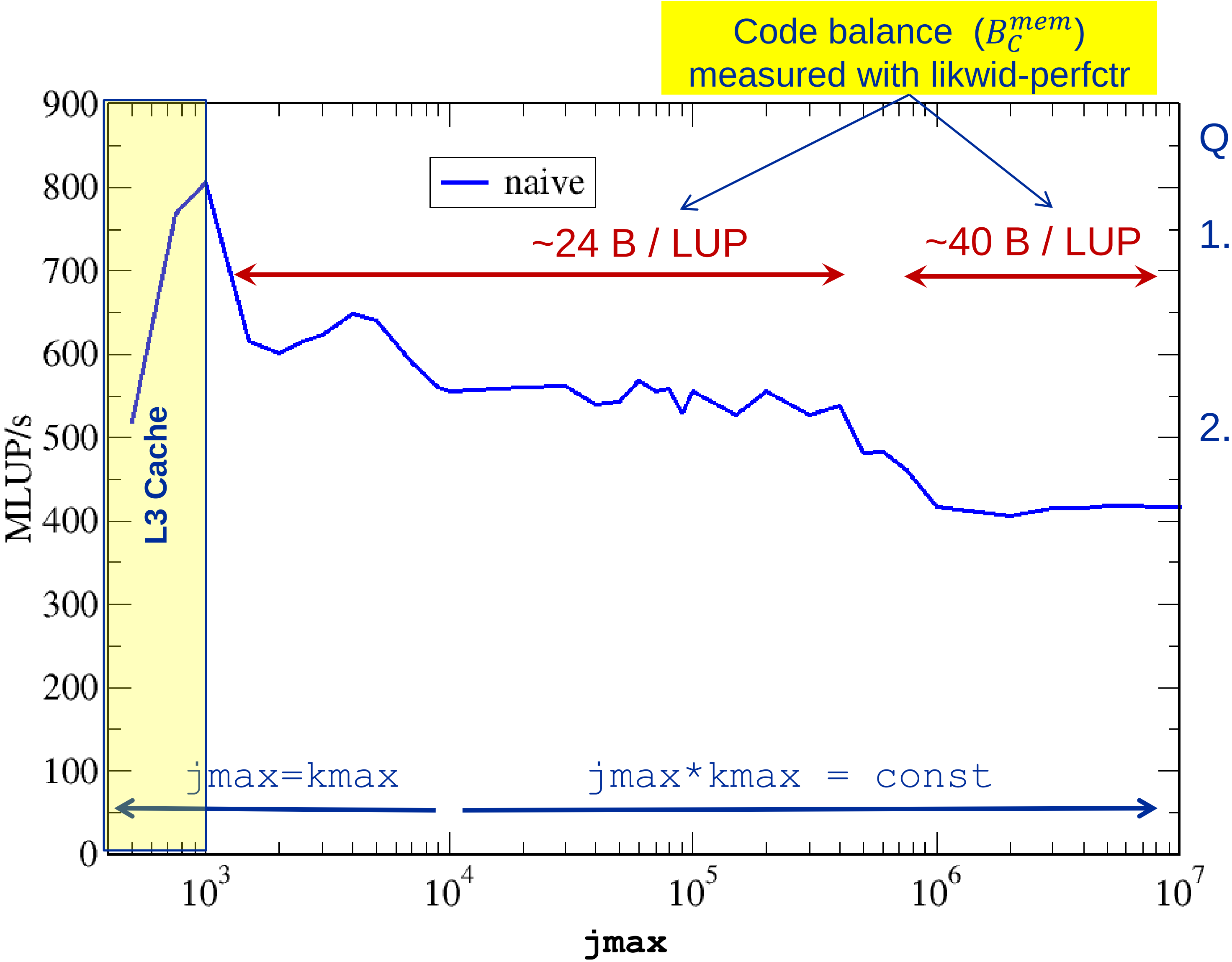
LD x(j,k+1)

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 > 10⁴`** ?

Intel Compiler
ifort V13.1

Intel Xeon E5-2690 v2
(“IvyBridge”@3 GHz)

Case study: A Jacobi smoother

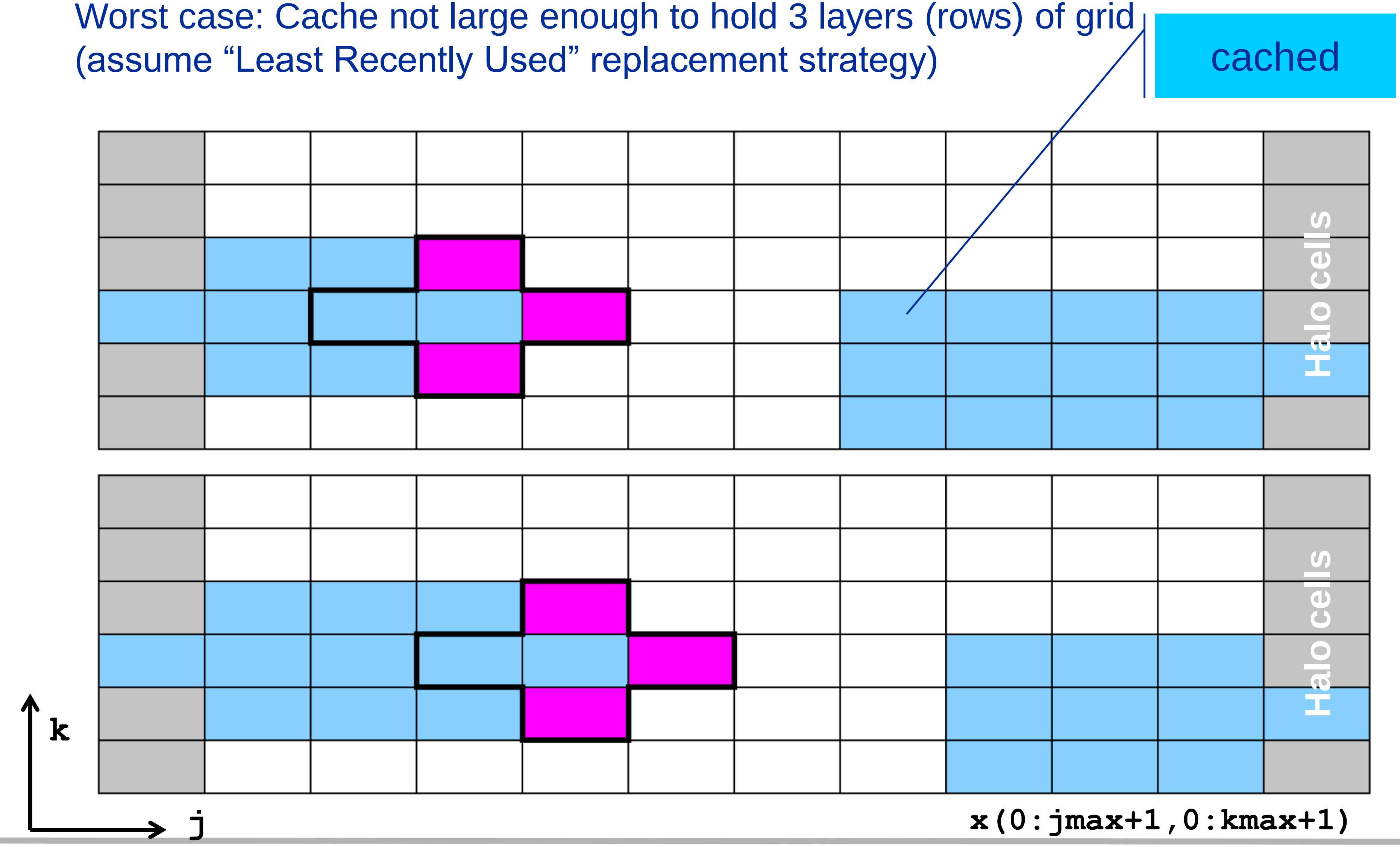
The basics in two dimensions

Layer conditions

Optimization by spatial blocking

Analyzing the data flow

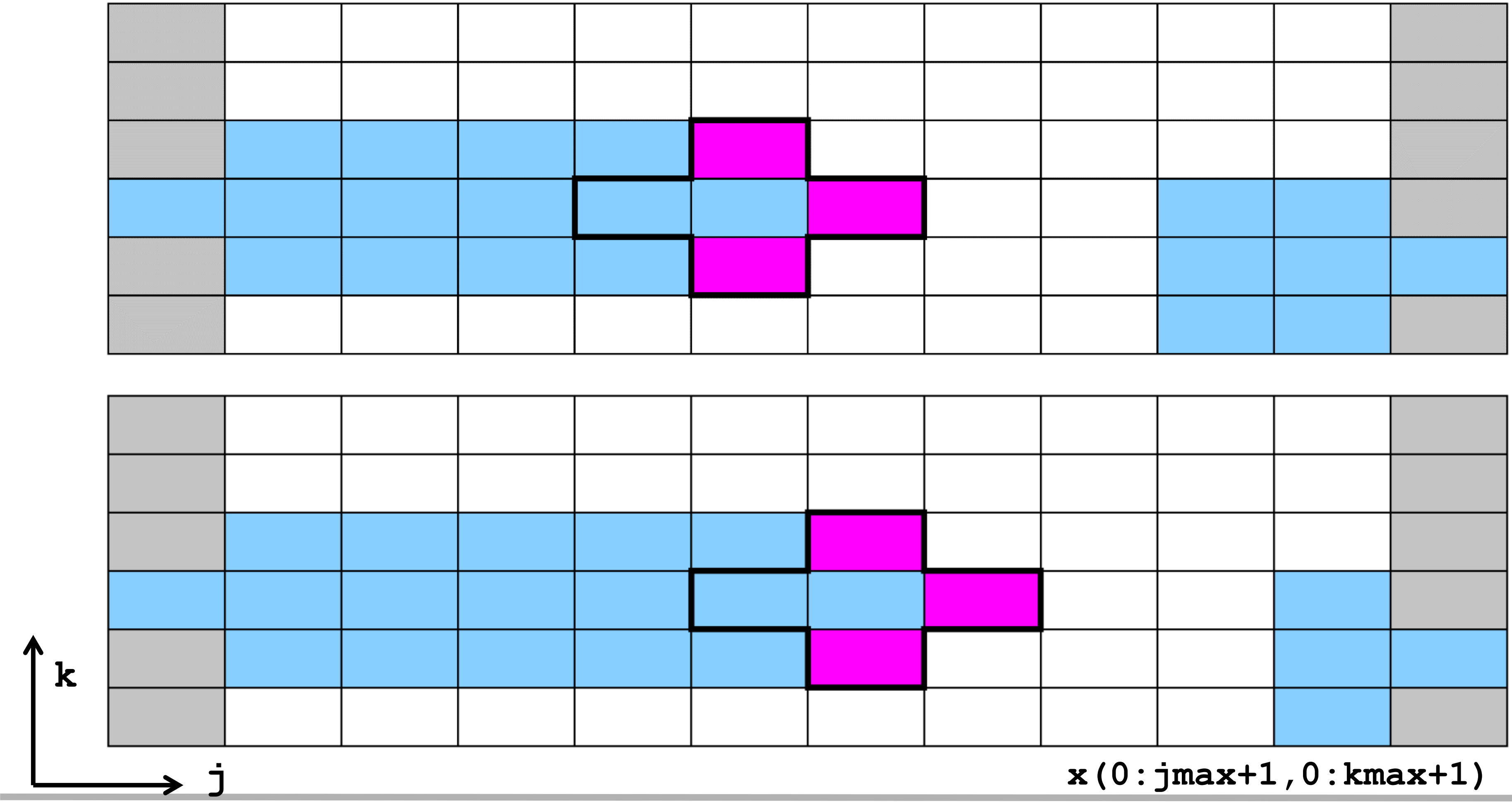
Worst case: Cache not large enough to hold 3 layers (rows) of grid
(assume “Least Recently Used” replacement strategy)



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

Analyzing the data flow

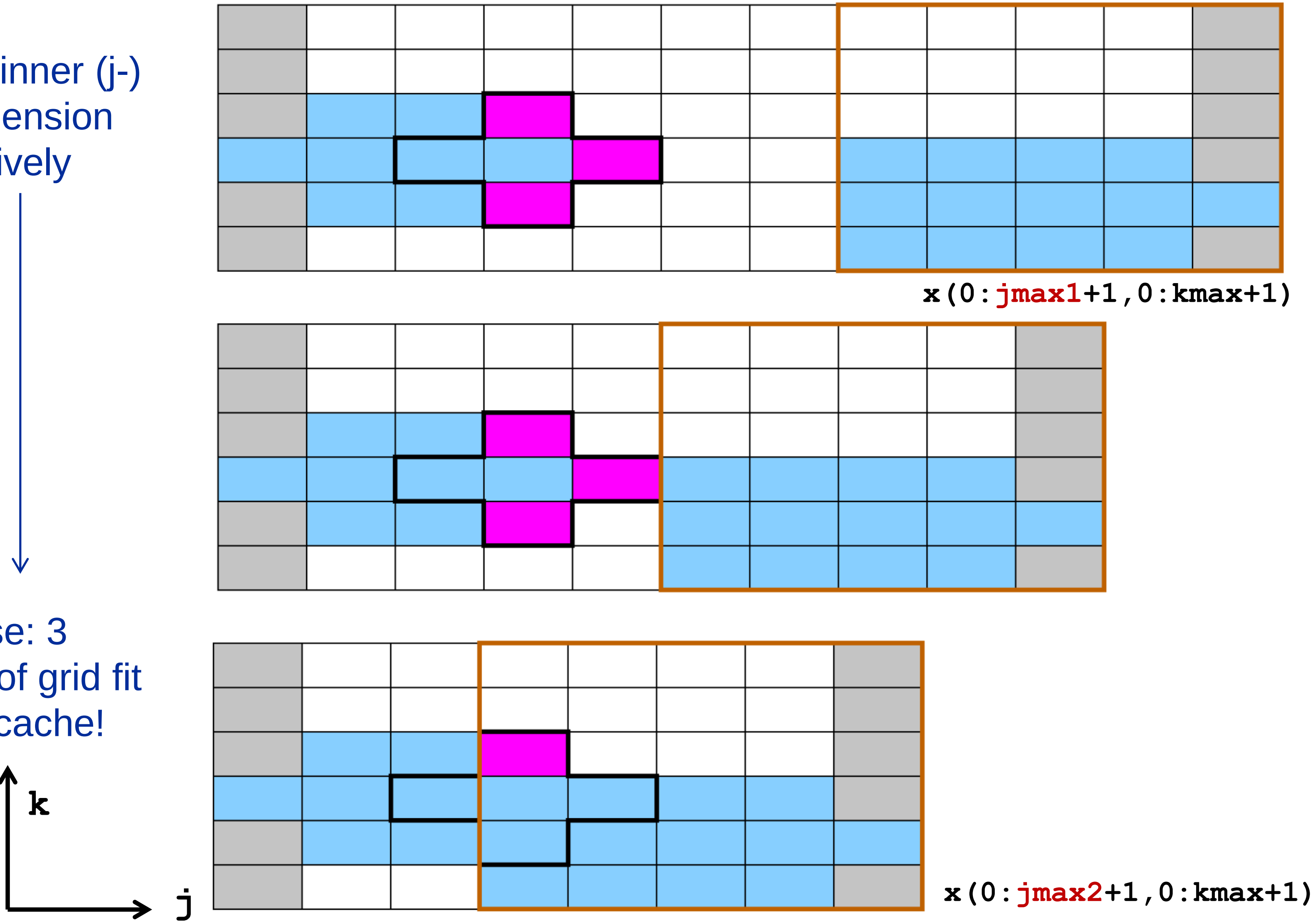
Worst case: Cache not large enough to hold 3 layers (rows) of grid (+assume „Least Recently Used“ replacement strategy)



Analyzing the data flow

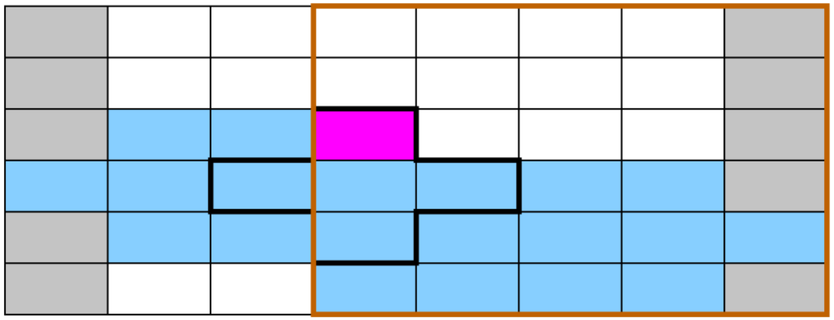
Reduce inner (j-) loop dimension successively

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



Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil



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

3 * jmax * 8B < CacheSize/2

"Layer condition"

3 rows of jmax

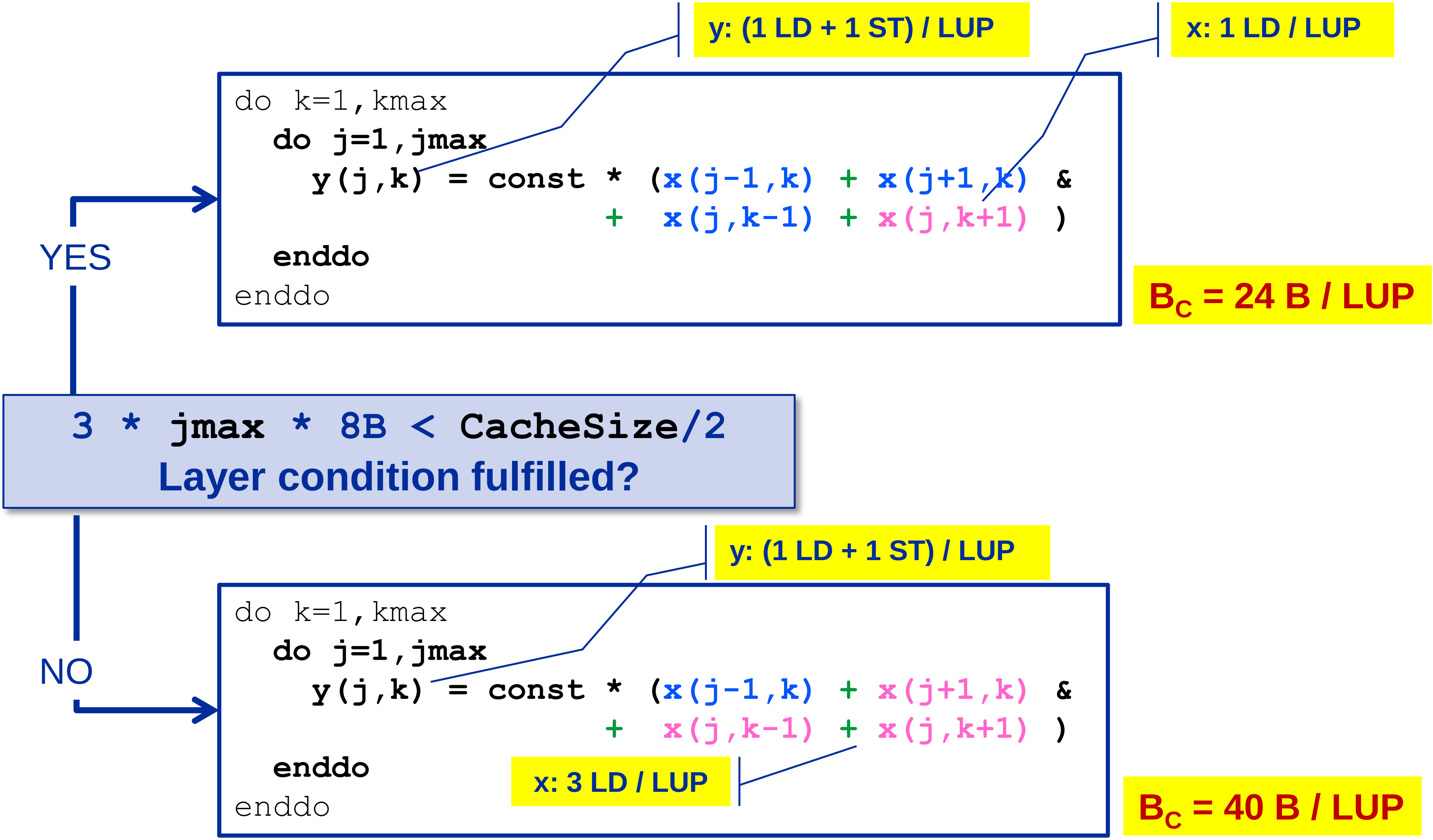
double precision

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

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

- **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`

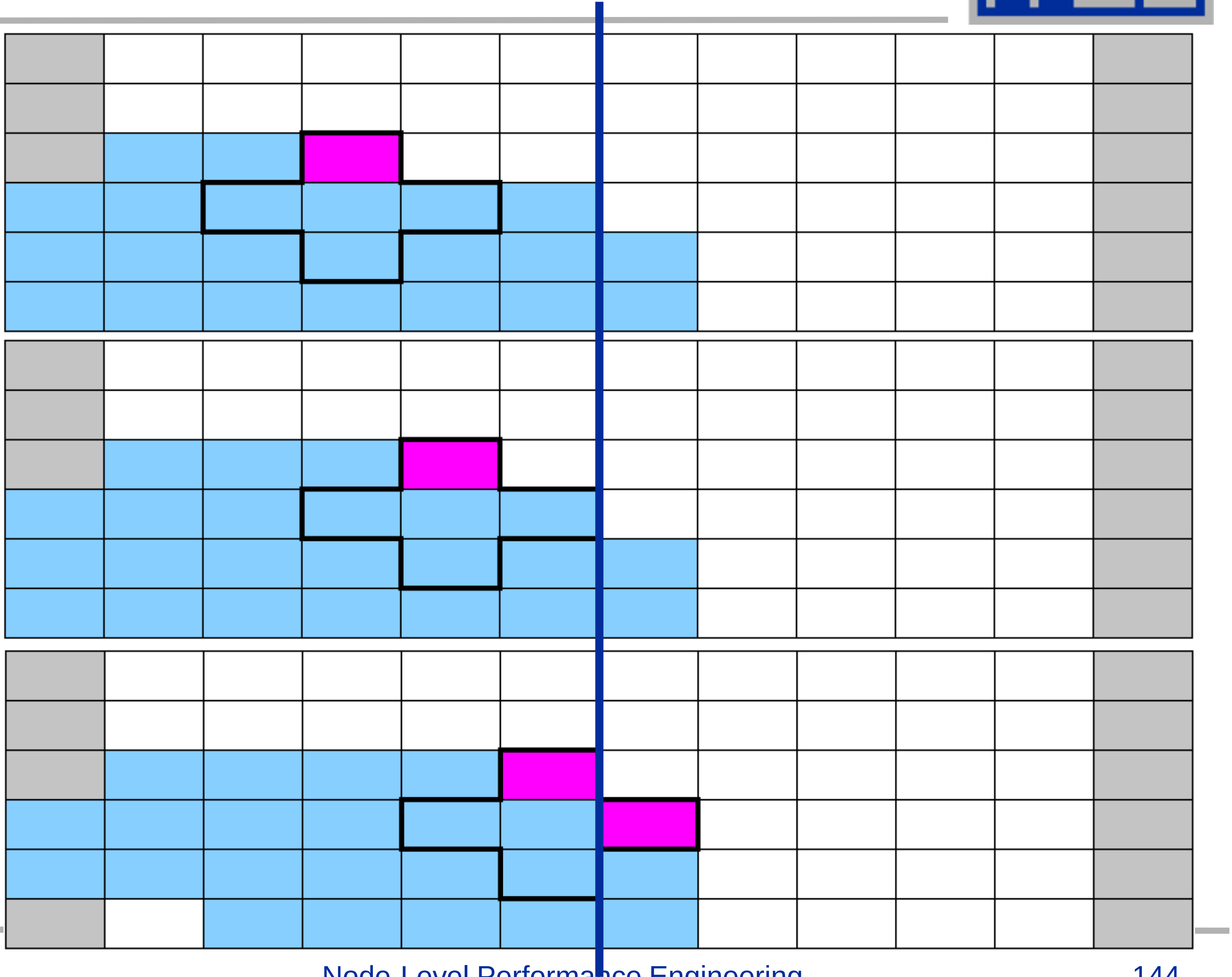
→ **Determine for given `CacheSize` an appropriate `jblock` value:**

`jblock < CacheSize / 48 B`

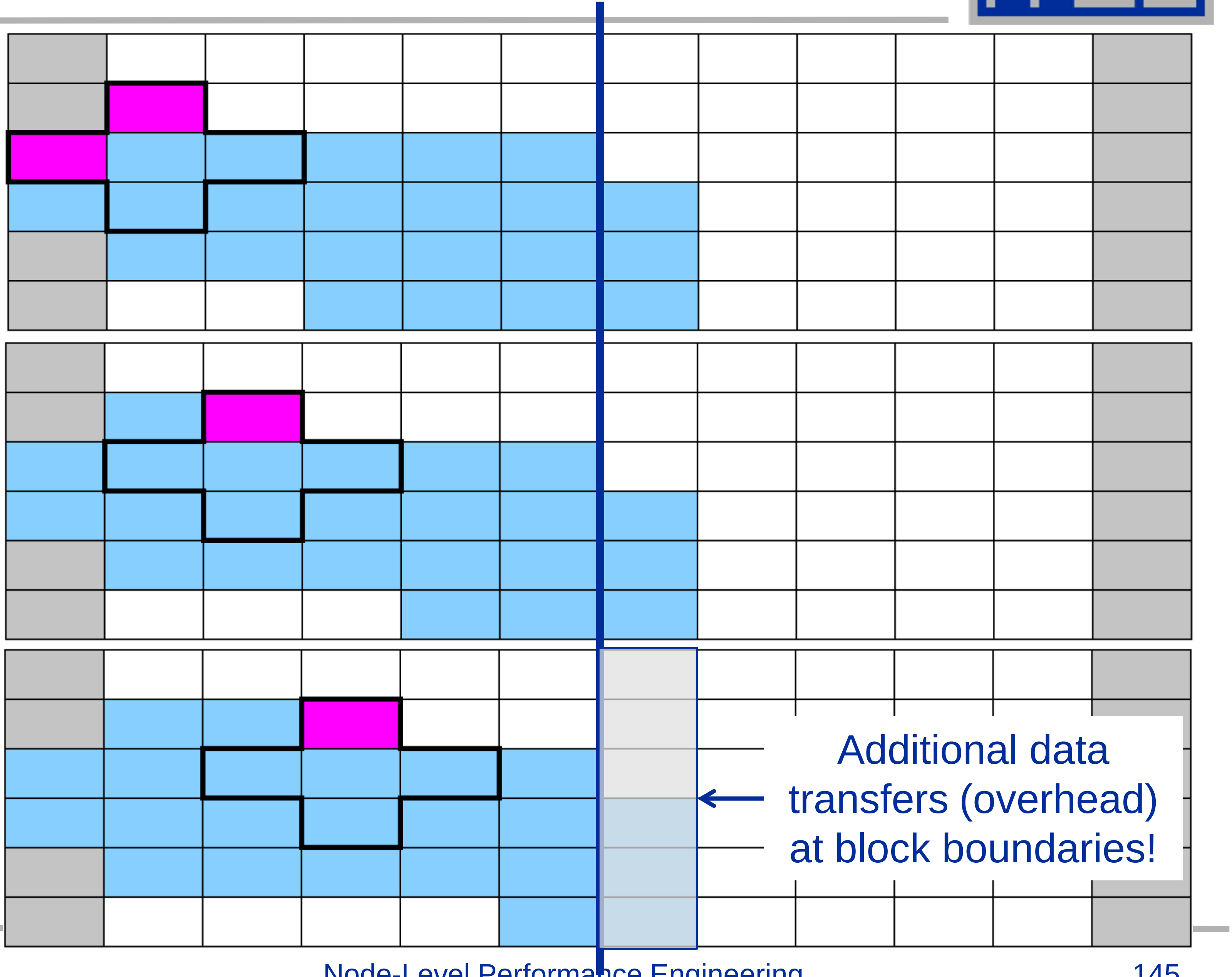
Establish the layer condition by blocking

Split up domain into subblocks:

e.g. block size = 5



Establish the layer condition by blocking

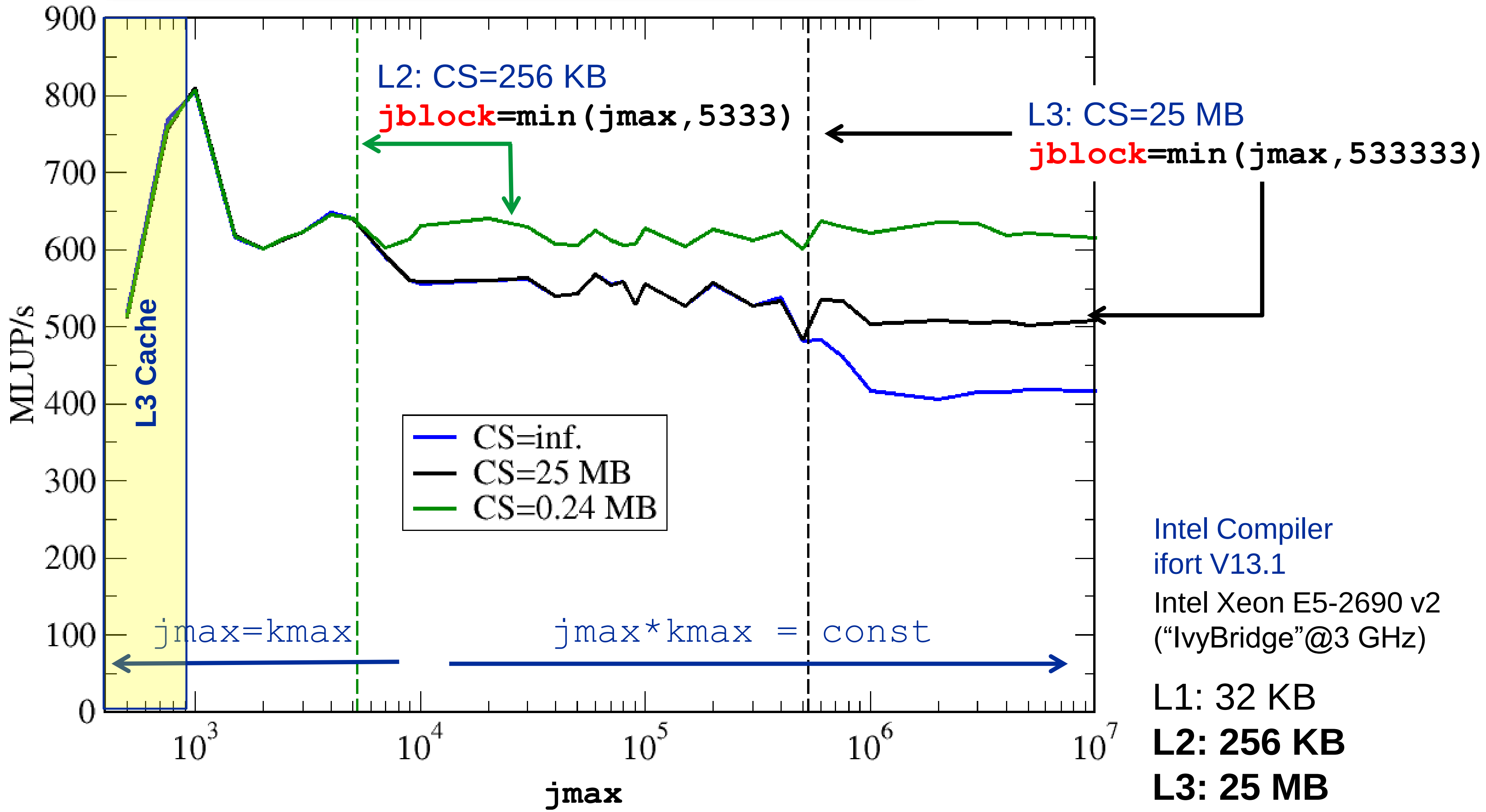


Additional data transfers (overhead) at block boundaries!

Establish layer condition by spatial blocking

`jblock < CacheSize / 48 B`

Which cache to block for?

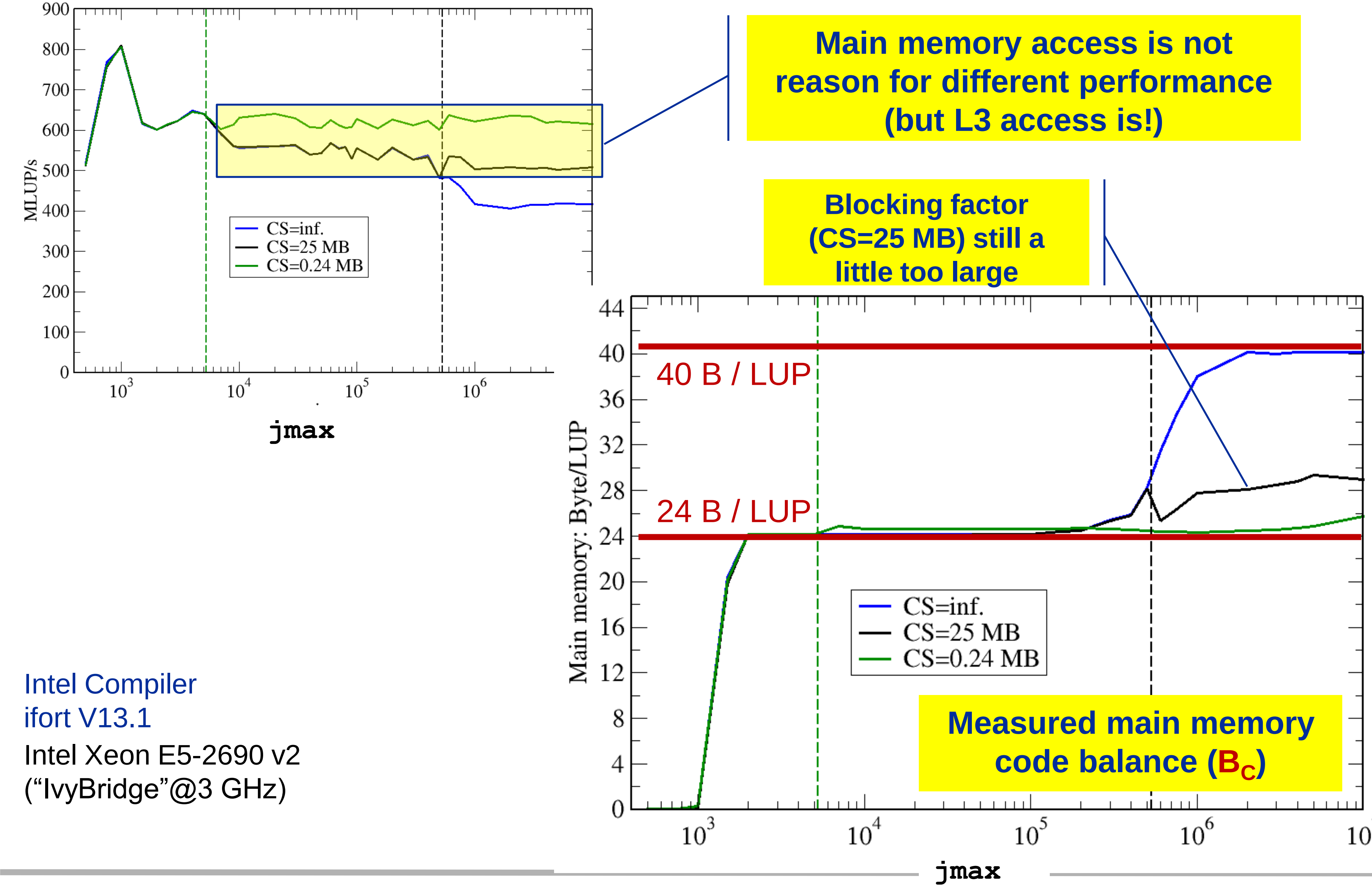


Intel Compiler
ifort V13.1

Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

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

Layer condition & spatial blocking: 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)

Jacobi Stencil – OpenMP parallelization

Basic guideline:
Parallelize outermost loop

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
   do j=1,jmax
      y(j,k) = 1/4.*(x(j-1,k)    +x(j+1,k) &
                   + x(j,k-1)    +x(j,k+1) )
   enddo
enddo
```

Equally large chunks in k-direction
→ "Layer condition" for each thread

"Layer condition" for shared cache:

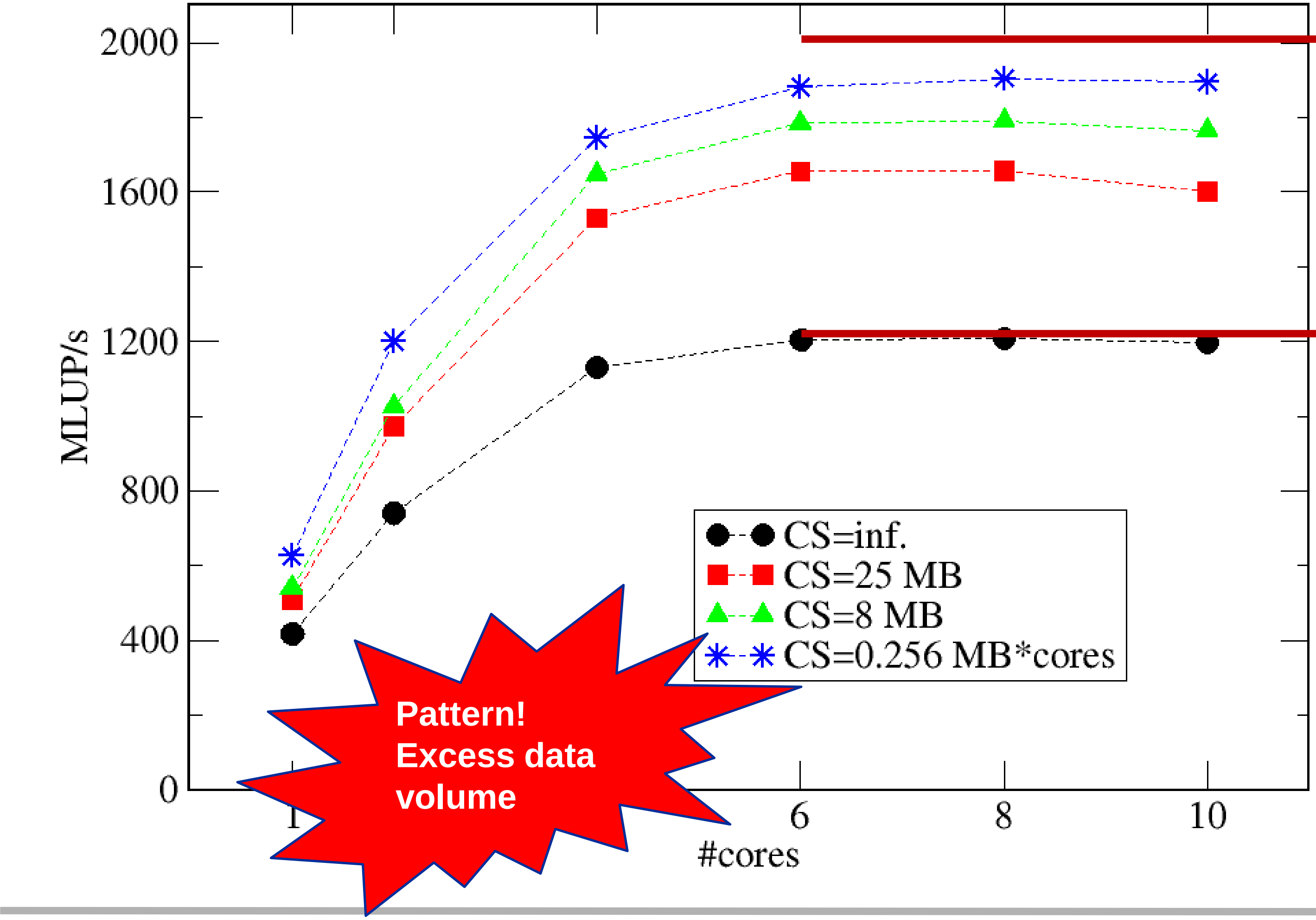
```
nthreads * 3 *imax * 8B < CS/2
```

Homework: what about if the cache to block for is not shared among the threads?

Socket scaling – validate Roofline model

$P = \min(P_{\max}, b_S/B_C)$

What is $P_{\max}$ here? → homework!



Pattern!
Excess data volume

B_C= 24 B/LUP

B_C= 40 B/LUP

Intel Compiler
ifort V13.1
OpenMP Parallel

Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

b_S = 48 GB/s

Jacobi Stencil in 3D

optional

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = 1/6.    *(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
```

“Layer condition” for shared cache:
`nthreads * 3 * jmax*imax * 8B < CS/2`

Layer condition (cubic domain; `CacheSize=25 MB`)
1 thread: `imax=jmax < 720` → 10 threads: `imax=jmax < 230`

Jacobi Stencil in 3D

optional

"Layer condition": `nthreads *3*jmax*imax*8B < CS/2`

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
   do j=1,jmax
      do i=1,imax
         y(i,j,k) = 1/6. *(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
```

B_C = 24 B / LUP

Roofline model:

$P = \min(P_{\max}, b_S / (24\ \text{B/LUP}))$

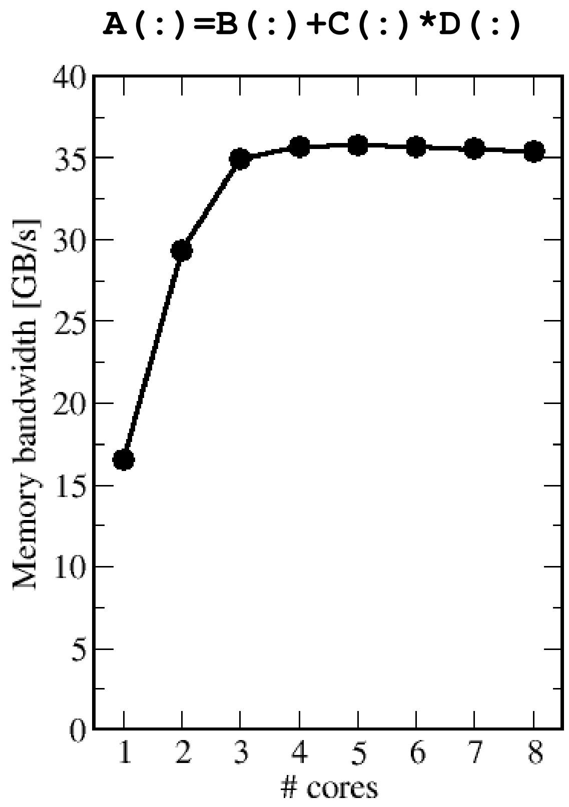
What is $P_{\max}$ here? → homework!

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**

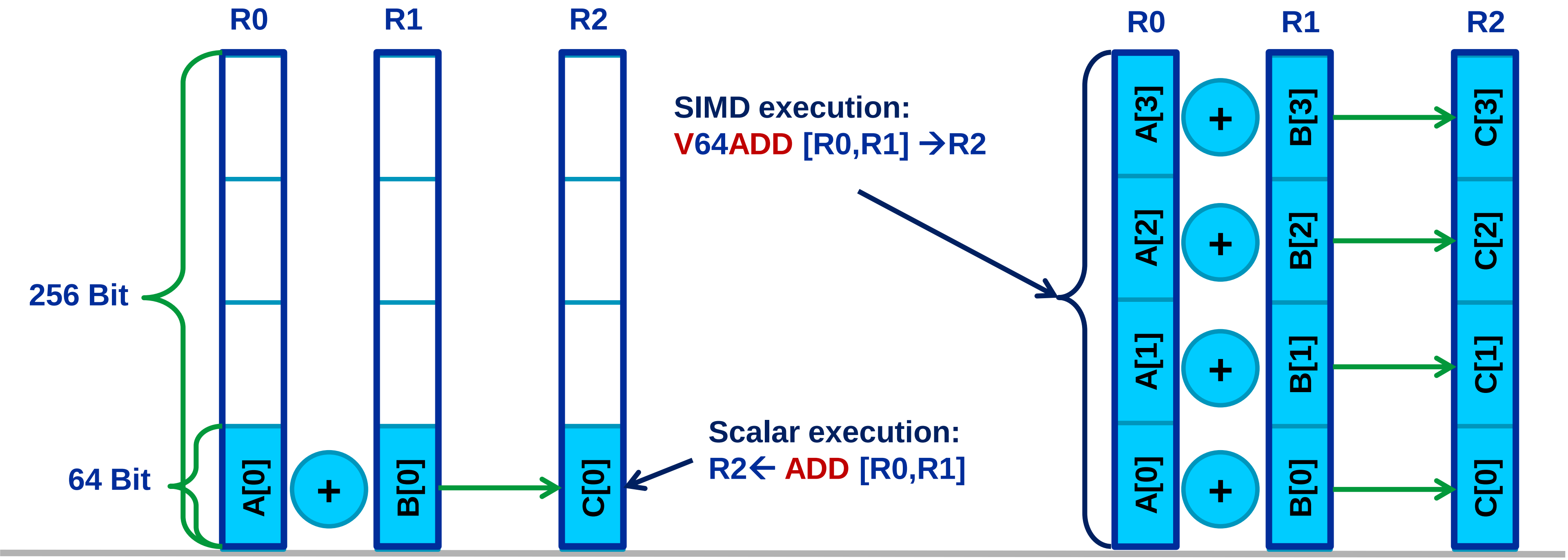
- **The ECM performance model gives more insight:**

  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

Coding for SingleInstructionMultipleData processing

SIMD processing – Basics

- **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
- **Adding two registers holding double precision floating point operands**

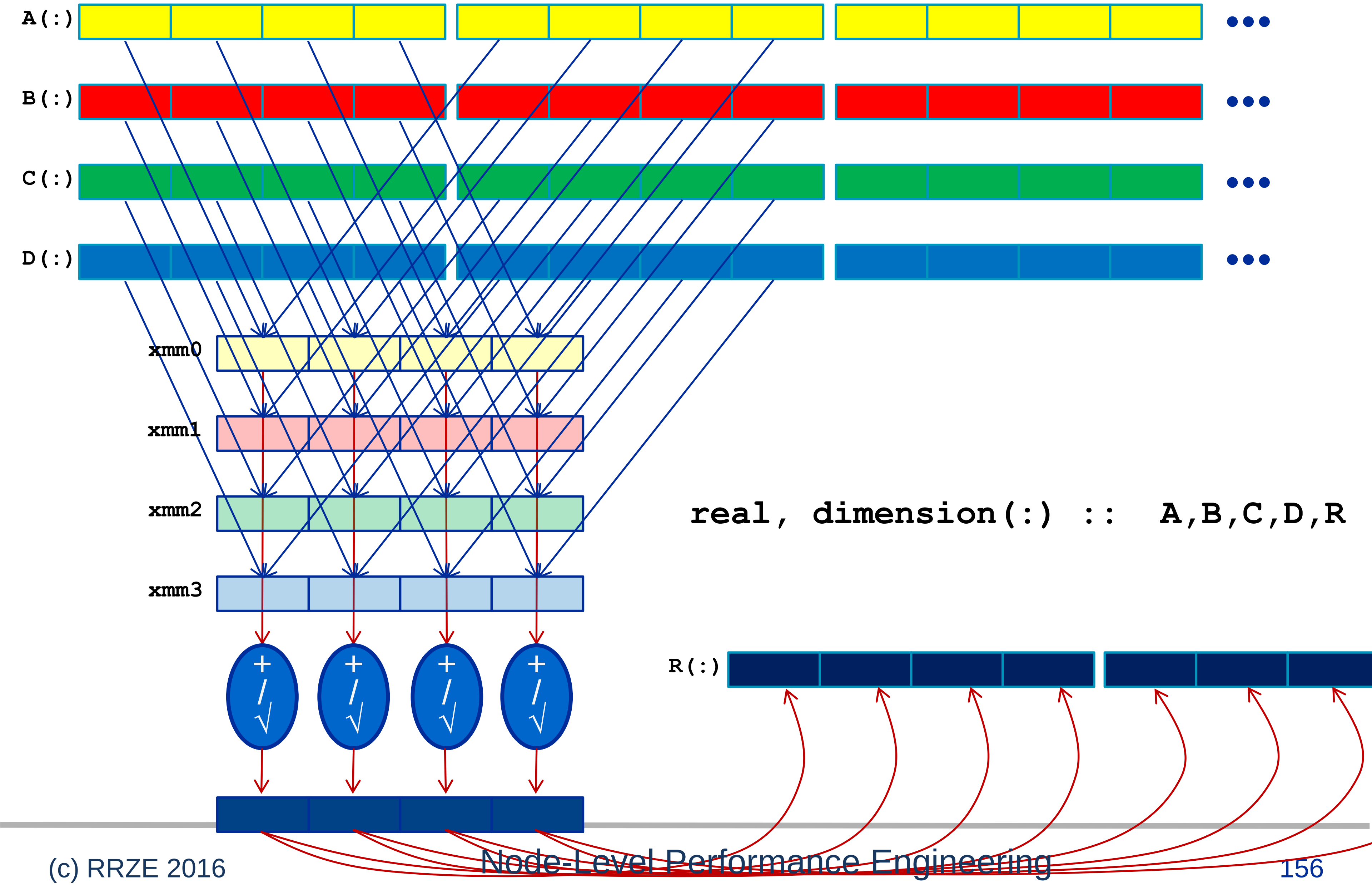


SIMD execution:
V64ADD [R0,R1] →R2

Scalar execution:
R2← ADD [R0,R1]

SIMD style structure of arrays processing

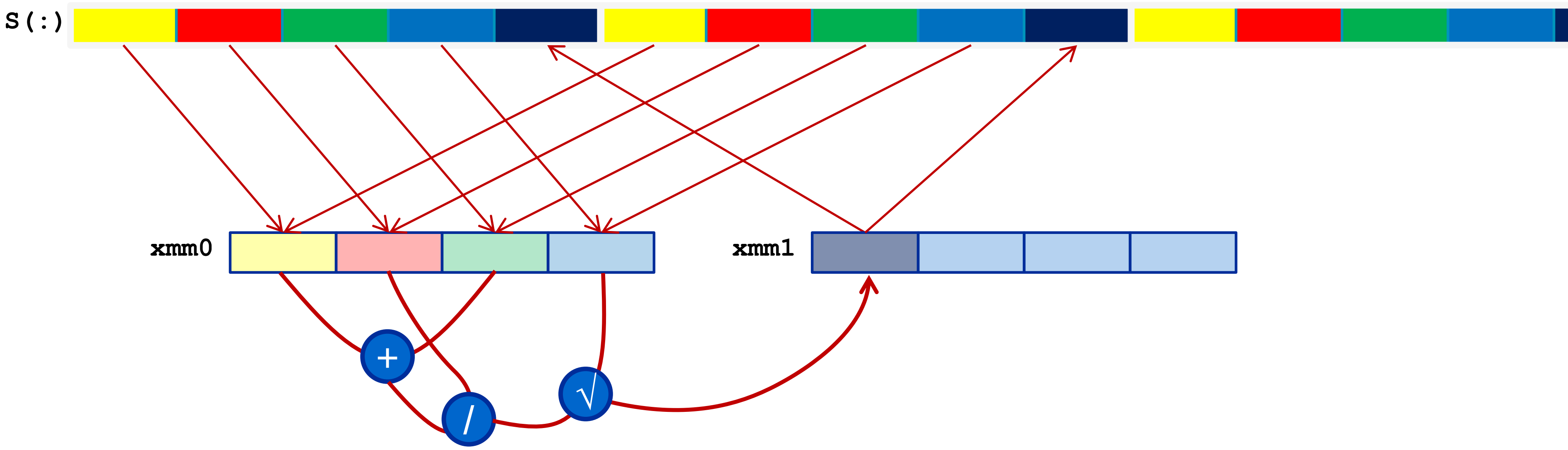
LOAD, STORE, arithmetic fully parallel



`real, dimension(:) :: A,B,C,D,R`

SIMD style of array of structures processing

Data access may be parallel, but arithmetic isn't



```
type struct
  real*4  A,B,C,D,R
end type struct

type(struct), dimension(:) :: S
```

→ Efficient SIMD requires appropriate data structures!

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” does not happen, tell it to 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) {…}
```

Vectorization compiler options (Intel)

optional

- **The compiler will vectorize starting with –`O2`.**
- **To enable specific SIMD extensions use the –x option:**
  - **`-xSSE2`** vectorize for SSE2 capable machines

  Available SIMD extensions:

  `SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX`

  - **`-xAVX(2)`** on Sandy Bridge (Haswell) processors

  Recommended option:

  - **`-xHost`** will optimize for the architecture you compile on

  On AMD Opteron: use plain **`–O3`** as the **`-x`** options may involve CPU type checks.

Vectorization compiler options

optional

- Controlling non-temporal stores (part of the SIMD extensions)

  - `-opt-streaming-stores always|auto|never`

`always` use NT stores, assume application is memory bound (use with caution!)

`auto` compiler decides when to use NT stores

`never` do not use NT stores unless activated by source code directive

Vectorization source code directives

optional

- Fine-grained control of loop vectorization
- Use `!DEC$` (Fortran) or `#pragma` (C/C++) sentinel to start a compiler directive

- `#pragma vector always`
  vectorize even if it seems inefficient (hint!)

- `#pragma novector`
  do not vectorize even if possible

- `#pragma vector nontemporal`
  use NT stores when allowed (i.e. alignment conditions are met)

- `#pragma vector aligned`
  specifies that all array accesses are aligned to 16-byte boundaries (DANGEROUS! You must not lie about this!)

User mandated vectorization

- **Since Intel Compiler 12.0 the `simd` pragma is available**
- **`#pragma simd` enforces vectorization where the other pragmas fail**
- **Prerequesites:**
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- **There are additional clauses: `reduction, vectorlength, private`**
- **Refer to the compiler manual for further details**

```
#pragma simd reduction(+:x)
  for (int i=0; i<n; i++) {
     x = x + A[i];
  }
```

- **NOTE: Using the #pragma simd the compiler may generate incorrect code if the loop violates the vectorization rules!**

x86 Architecture:
SIMD and Alignment

- **Alignment issues**
  - Alignment of arrays with SSE (AVX) should be on 16-byte (32-byte) boundaries to allow packed aligned loads and NT stores **(for Intel processors)**
    - **AMD has a scalar nontemporal store instruction**
  - Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say **`vector nontemporal`**
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but **Xeon Phi relies heavily on it!**
  - How is manual alignment accomplished?
- **Dynamic allocation of aligned memory (`align` = alignment boundary):**

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr,
                   size_t align,
                   size_t size);
```

optional

Reading x86 assembly code and exploiting SIMD parallelism

Understanding SIMD execution by inspecting assembly code

SIMD vectorization how-to

Intel compiler options and features for SIMD

Why and how?

optional

Why check the assembly code?

- **Sometimes the only way to make sure the compiler "did the right thing"**
  - Example: "LOOP WAS VECTORIZED" message is printed, but Loads & Stores may still be scalar!

- **Get the assembler code (Intel compiler):**

```
icc -S -masm=intel -O3  -xHost  triad.c  -o a.out
```

- **Disassemble Executable:**

```
objdump -d  ./a.out | less
```

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B
AMD64 Architecture Programmer's Manual Vol. 1-5

Basics of the x86-64 ISA

optional

- Instructions have 0 to 4 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: `A[i]` equivalent to `*(A+i)` (a pointer has a type: `A+i*8`)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps      %xmm4, 48(%rdi,%rax,8)
addq        $8, %rax
js          ..B1.4
```

```
401b9f: 0f 29 5c c7 30       movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08          add    $0x8,%rax
401ba8: 78 a6                js     401b50 <triad_asm+0x4b>
```

Basics of the x86-64 ISA

optional

16 general Purpose Registers (64bit):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32 bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`

Floating Point SIMD Registers:

`xmm0-xmm15` SSE (128bit) alias with 256-bit registers

`ymm0-ymm15` AVX (256bit)

SIMD instructions are distinguished by:

AVX (VEX) prefix:	`v`
Operation:	`mul, add, mov`
Modifier:	nontemporal (`nt`), unaligned (`u`), aligned (`a`), high (`h`)
Width:	scalar (`s`), packed (`p`)
Data type:	single (`s`), double (`d`)

Case Study: Vector Triad (DP) on IvyBridge-EP

optional

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

To get object code use `objdump -d` on object file or executable or compile with `-S`

Assembly code (-O1):

CLANG

```
LBB0_3
movsd   xmm0, [rdx]
mulsd   xmm0, [rcx]
addsd   xmm0, [rsi]
movsd   [rax], xmm0
add     rsi, 8
add     rdx, 8
add     rcx, 8
add     rax, 8
dec     edi
jne     LBB0_3
```

ICC

```
..B1.6:
movsd   xmm0, [r12+rax*8]
mulsd   xmm0, [r13+rax*8]
addsd   xmm0, [r14+rax*8]
movsd   [r15+rax*8], xmm0
inc     rax
cmp     rax, rbx
jl      ..B1.6
```

GCC

```
.L4:
movsd xmm0,[rbx+rax]
mulsd xmm0,[r12+rax]
addsd xmm0,[r13+0+rax]
movsd [rbp+0+rax],xmm0
add rax, 8
cmp rax, r14
jne .L4
```

7 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 (Intel compiler)

optional

SSE

```
..B1.19:
        movsd     xmm0,  [r15+rsi*8]
        movsd     xmm3,  [16+r15+rsi*8]
        movsd     xmm5,  [32+r15+rsi*8]
        movsd     xmm7,  [48+r15+rsi*8]
        movhpd    xmm0,  [8+r15+rsi*8]
        movhpd    xmm3,  [24+r15+rsi*8]
        movhpd    xmm5,  [40+r15+rsi*8]
        movhpd    xmm7,  [56+r15+rsi*8]
        mulpd     xmm0,  [r14+rsi*8]
        mulpd     xmm3,  [16+r14+rsi*8]
        mulpd     xmm5,  [32+r14+rsi*8]
        mulpd     xmm7,  [48+r14+rsi*8]
        movsd     xmm2,  [r13+rsi*8]
        movsd     xmm4,  [16+r13+rsi*8]
        movsd     xmm6,  [32+r13+rsi*8]
        movsd     xmm8,  [48+r13+rsi*8]
        movhpd    xmm2,  [8+r13+rsi*8]
        movhpd    xmm4,  [24+r13+rsi*8]
        movhpd    xmm6,  [40+r13+rsi*8]
        movhpd    xmm8,  [56+r13+rsi*8]
        addpd     xmm2, xmm0
        addpd     xmm4, xmm3
        addpd     xmm6, xmm5
        addpd     xmm8, xmm7
        movaps    [rdx+rsi*8], xmm2
        movaps    [16+rdx+rsi*8], xmm4
        movaps    [32+rdx+rsi*8], xmm6
        movaps    [48+rdx+rsi*8], xmm8
        add       rsi, 8
        cmp       rsi, r9
        jb        ..B1.19
```

3.86 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 –xHost

optional

AVX

```
..B1.15:
vmovupd    xmm2, [r15+rsi*8]
vmovupd    xmm10, [32+r15+rsi*8]
vmovupd    xmm3, [rdx+rsi*8]
vmovupd    xmm11, [32+rdx+rsi*8]
vmovupd    xmm0, [r14+rsi*8]
vmovupd    xmm9, [32+r14+rsi*8]
vinsertf128 ymm4, ymm2,[16+r15+rsi*8], 1
vinsertf128 ymm12,ymm10,[48+r15+rsi*8],1
vinsertf128 ymm5, ymm3,[16+rdx+rsi*8], 1
vinsertf128 ymm13,ymm11,[48+rdx+rsi*8],1
vmulpd     ymm7, ymm4, ymm5
vmulpd     ymm15, ymm12, ymm13
vmovupd    xmm4, [64+rdx+rsi*8]
vmovupd    xmm12, [96+rdx+rsi*8]
vmovupd    xmm3, [64+r15+rsi*8]
vmovupd    xmm11, [96+r15+rsi*8]
vmovupd    xmm2, [64+r14+rsi*8]
vmovupd    xmm10, [96+r14+rsi*8]
vinsertf128 ymm14,ymm9,[48+r14+rsi*8], 1
vinsertf128 ymm6,ymm0,[16+r14+rsi*8], 1
vaddpd     ymm8, ymm6, ymm7
vaddpd     ymm0, ymm14, ymm15
```

```
vmovupd    [r13+rsi*8], ymm8
vmovupd    [32+r13+rsi*8], ymm0
vinsertf128 ymm5, ymm3, [80+r15+rsi*8], 1
vinsertf128 ymm13,ymm11,[112+r15+rsi*8], 1
vinsertf128 ymm6, ymm4,  [80+rdx+rsi*8], 1
vinsertf128 ymm14,ymm12,[112+rdx+rsi*8], 1
vmulpd     ymm8, ymm5, ymm6
vmulpd     ymm0, ymm13, ymm14
vinsertf128 ymm7, ymm2, [80+r14+rsi*8], 1
vinsertf128 ymm15,ymm10,[112+r14+rsi*8], 1
vaddpd     ymm9, ymm7, ymm8
vaddpd     ymm2, ymm15, ymm0
vmovupd    [64+r13+rsi*8], ymm9
vmovupd    [96+r13+rsi*8], ymm2
add        rsi, 16
cmp        rsi, r9
jb         ..B1.15
```

2.44 instructions per loop iteration

Benefit of SIMD limited by *serial* fraction!

Case Study: Vector Triad (DP) –O3 –xHost
#pragma vector aligned

optional

SSE

```
..B1.7:
movaps     xmm0, [r13+rcx*8]
movaps     xmm2, [16+r13+rcx*8]
movaps     xmm3, [32+r13+rcx*8]
movaps     xmm4, [48+r13+rcx*8]
mulpd      xmm0, [rbp+rcx*8]
mulpd      xmm2, [16+rbp+rcx*8]
mulpd      xmm3, [32+rbp+rcx*8]
mulpd      xmm4, [48+rbp+rcx*8]
addpd      xmm0, [r12+rcx*8]
addpd      xmm2, [16+r12+rcx*8]
addpd      xmm3, [32+r12+rcx*8]
addpd      xmm4, [48+r12+rcx*8]
movaps     [r15+rcx*8], xmm0
movaps     [16+r15+rcx*8], xmm2
movaps     [32+r15+rcx*8], xmm3
movaps     [48+r15+rcx*8], xmm4
add        rcx, 8
cmp        rcx, rsi
jb         ..B1.7
```

2.38 instructions per loop iteration

AVX

```
..B1.7:
vmovupd    ymm0, [r15+rcx*8]
vmovupd    ymm4, [32+r15+rcx*8]
vmovupd    ymm7, [64+r15+rcx*8]
vmovupd    ymm10,[96+r15+rcx*8]
vmulpd     ymm2, ymm0, [rdx+rcx*8]
vmulpd     ymm5, ymm4, [32+rdx+rcx*8]
vmulpd     ymm8, ymm7, [64+rdx+rcx*8]
vmulpd     ymm11, ymm10, [96+rdx+rcx*8]
vaddpd     ymm3, ymm2, [r14+rcx*8]
vaddpd     ymm6, ymm5, [32+r14+rcx*8]
vaddpd     ymm9, ymm8, [64+r14+rcx*8]
vaddpd     ymm12, ymm11, [96+r14+rcx*8]
vmovupd    [r13+rcx*8], ymm3
vmovupd    [32+r13+rcx*8], ymm6
vmovupd    [64+r13+rcx*8], ymm9
vmovupd    [96+r13+rcx*8], ymm12
add        rcx, 16
cmp        rcx, rsi
jb         ..B1.7
```

1.19 instructions per loop iteration

Case Study: Vector Triad (DP) –O3 –xHost `#pragma vector aligned` on Haswell-EP

optional

```
..B1.7:
vmovupd     ymm2, [r15+rcx*8]
vmovupd     ymm4, [32+r15+rcx*8]
vmovupd     ymm6, [64+r15+rcx*8]
vmovupd     ymm8, [96+r15+rcx*8]
vmovupd     ymm0, [rdx+rcx*8]
vmovupd     ymm3, [32+rdx+rcx*8]
vmovupd     ymm5, [64+rdx+rcx*8]
vmovupd     ymm7, [96+rdx+rcx*8]
vfmadd213pd ymm2, ymm0, [r14+rcx*8]
vfmadd213pd ymm4, ymm3, [32+r14+rcx*8]
vfmadd213pd ymm6, ymm5, [64+r14+rcx*8]
vfmadd213pd ymm8, ymm7, [96+r14+rcx*8]
vmovupd     [r13+rcx*8], ymm2
vmovupd     [32+r13+rcx*8], ymm4
vmovupd     [64+r13+rcx*8], ymm6
vmovupd     [96+r13+rcx*8], ymm8
add         rcx, 16
cmp         rcx, rsi
jb          ..B1.7
```

AVX + FMA3

On X86 ISA instruction are converted to so-called µops (elementary ops like load, add, mult). For performance the number of µops is important.

23 uops vs. 27 µops (AVX)

1.19 instructions per loop iteration

Mapping the ISA on a Microarchitecture

optional

Analysis performed for Haswell-EP

Throughput for arithmetic instructions:

Instruction mix	Execution time
1 ADD	1 cy
2 ADD	2 cy
1 MUL	1 cy
2 MUL	1 cy
1 ADD + 1 MUL	1 cy
2 FMA	1 cy

Throughput for loads and stores:

Instruction mix	Execution time
1 LOAD	1 cy
1 STORE	2 cy
1 LOAD and 1 STORE	1 cy
2 LOADs and 1 STORE	1 cy

- Throughput performance for steady state optimal execution
- Instruction throughput for scalar *or* SIMD instructions
- Load/Store units on Haswell are 32 byte wide. Was 16 bytes on previous Intel architectures.

SIMD processing – The whole picture

optional

SIMD influences instruction execution in the core – other runtime contributions stay the same!

AVX example (Haswell-EP):

Execution Units	3 cy
Caches	15 cy
Memory	21 cy

Per-cacheline (8 iterations) cycle counts

Comparing total execution time (cycles):

	Execution	Cache	Memory
Scalar	12	15	21
SSE	6	15	21
AVX	3	15	21

Total runtime with data loaded from memory:

Scalar	48 cy
SSE	42 cy
AVX	39 cy

SIMD is only effective if runtime is dominated by instruction execution!

How to leverage SIMD: your options

optional

Alternatives:

- The **compiler** does it for you (but: aliasing, alignment, language)
- 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);
}
```

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 for pointers to rule out aliasing

Efficient parallel programming on ccNUMA nodes

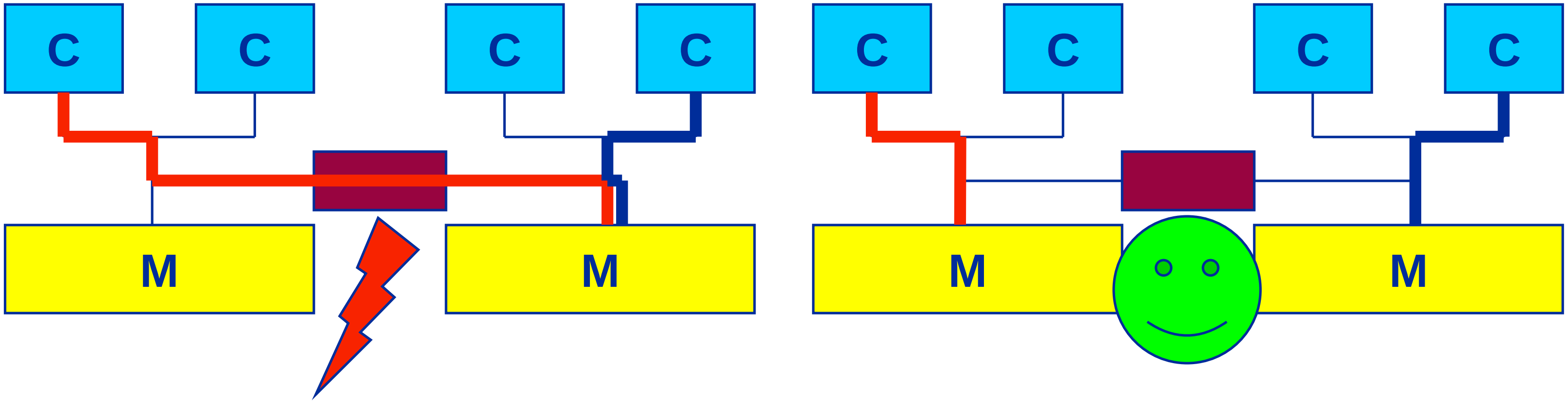
Performance characteristics of ccNUMA nodes

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?**

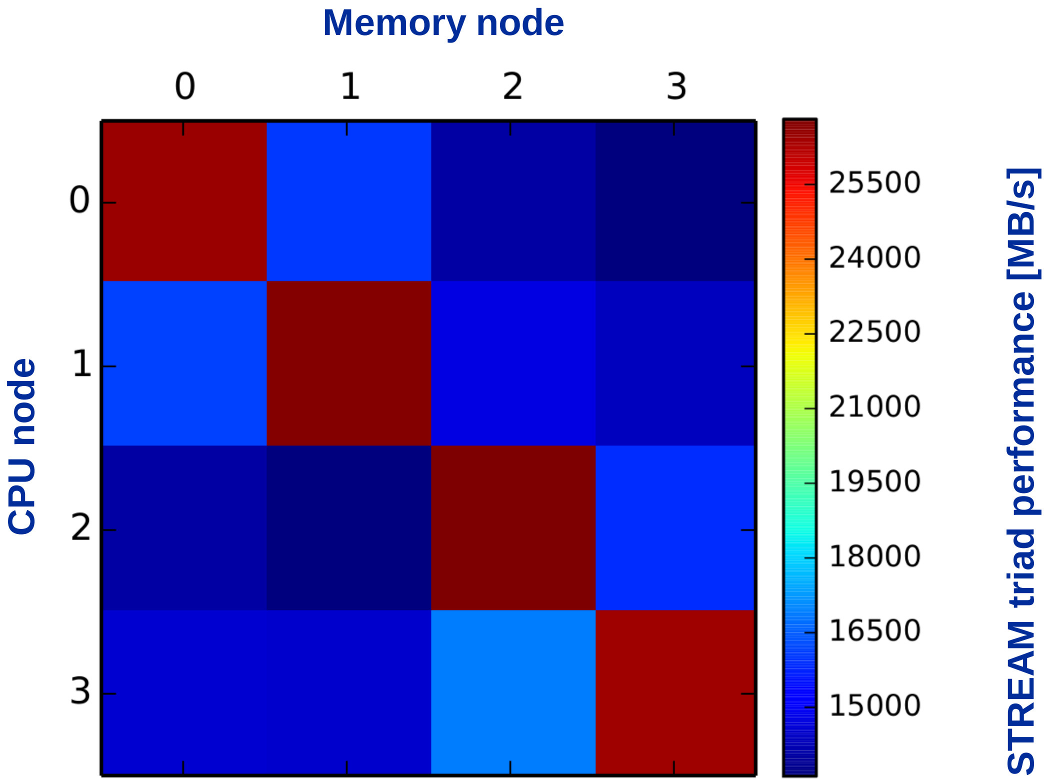


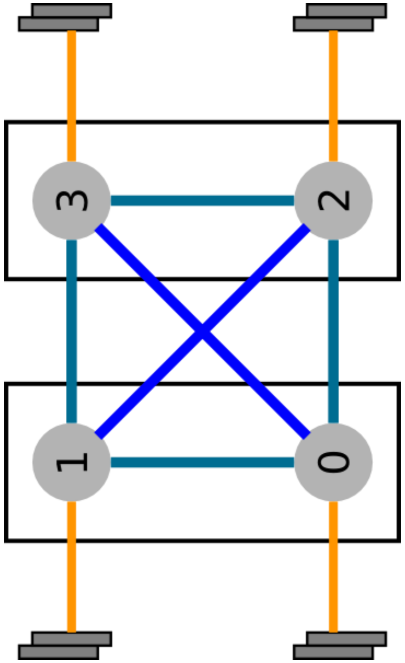
- Page placement is implemented in units of OS pages (often 4kB, possibly more)

Intel Broadwell EP node

*2 chips, 2 sockets, 11 cores per ccNUMA domain (**CoD** mode)*

- **ccNUMA map: Bandwidth penalties for remote access**
  - Run 11 threads per ccNUMA domain (half chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM copy benchmark using standard stores





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

optional

- `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
  enddo
enddo
```

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: "touch" means "write", not "allocate"**
- **Example:**

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

for(i=0; i<N; i++) // or i+=PAGE_SIZE
   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
```

⚡

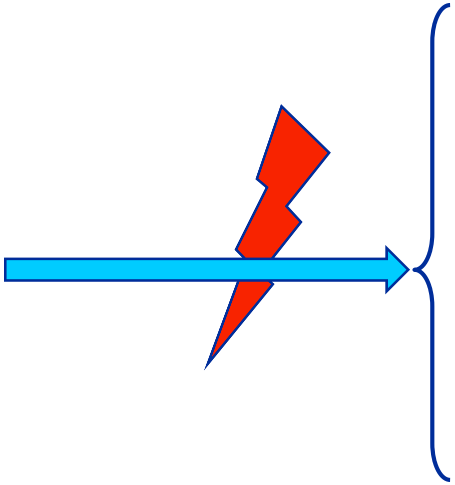
```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

😊

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
    - See below
- **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

Coding for Data Locality:

Placement of static arrays or arrays of objects

optional

- **Don't forget that constructors tend to touch the data members of an object. Example:**

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  inline D operator*(const D& o) throw() {
    return D(d*o.d);
  }
...
};
```

→ **placement problem with**

```
D* array = new D[1000000];
```

Coding for Data Locality:

Parallel first touch for arrays of objects

optional

- **Solution: Provide overloaded `D::operator new[]`**

```
void* D::operator new[](size_t n) {
  char *p = new char[n];    // allocate

  size_t i,j;
#pragma omp parallel for private(j) schedule(...)
  for(i=0; i<n; i += sizeof(D))
    for(j=0; j<sizeof(D); ++j)
      p[i+j] = 0;
  return p;
}

void D::operator delete[](void* p) throw() {
  delete [] static_cast<char*>p;
}
```

parallel first touch

- **Placement of objects is then done automatically by the C++ runtime via "placement new"**

Coding for Data Locality:

NUMA allocator for parallel first touch in `std::vector<>`

optional

```
template <class T> class NUMA_Allocator {
public:
  T* allocate(size_type numObjects, const void
              *localityHint=0) {
    size_type ofs,len = numObjects * sizeof(T);
    void *m = malloc(len);
    char *p = static_cast<char*>(m);
    int i,pages = len >> PAGE_BITS;
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) << PAGE_BITS;
      p[ofs]=0;
    }
    return static_cast<pointer>(m);
  }
...
};
```

Application:

```
vector<double,NUMA_Allocator<double> > x(10000000)
```

Diagnosing Bad Locality

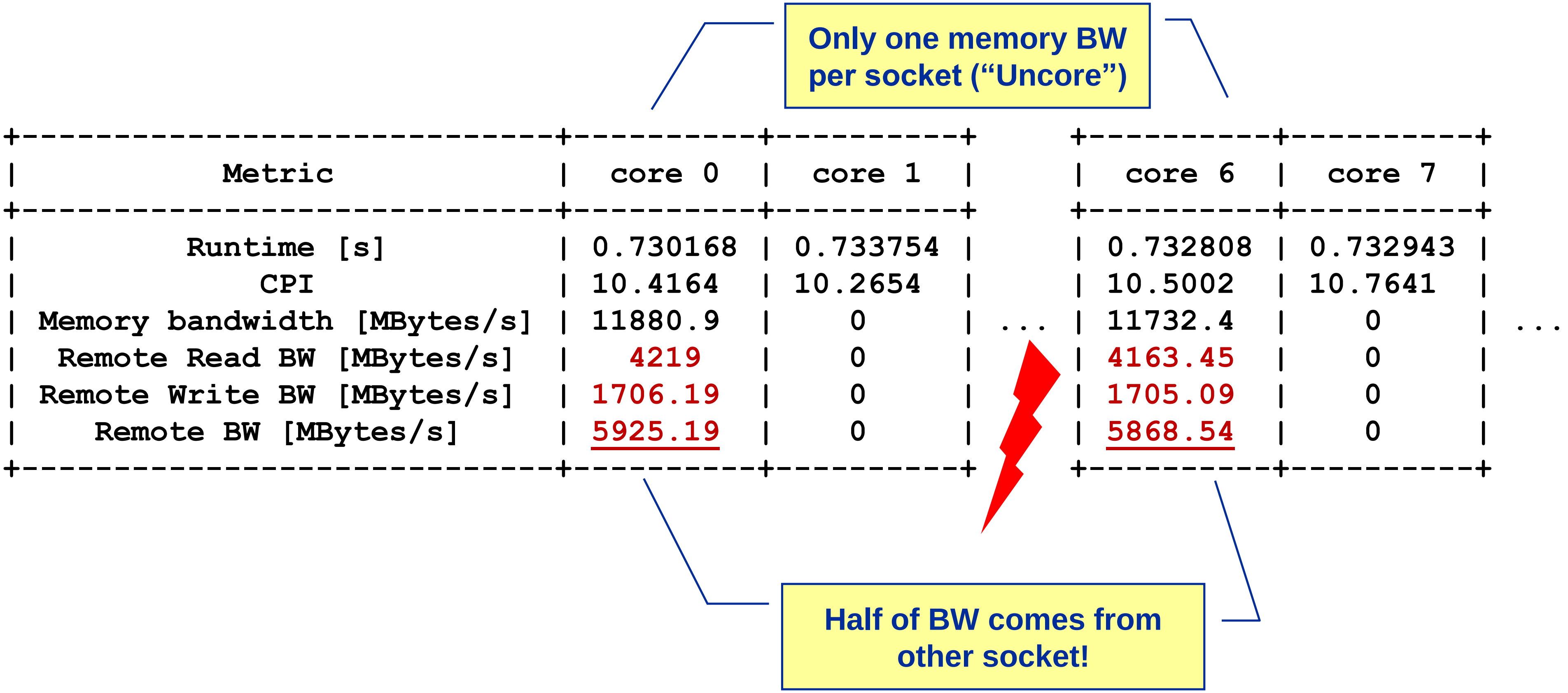
optional

- **If your code is cache-bound, you might not notice any locality problems**
- **Otherwise, bad locality limits scalability at very low CPU numbers (whenever a node boundary is crossed)**
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…
- **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 Westmere dual-socket system (Core i7, hex-core):

```
env OMP_NUM_THREADS=12 likwid-perfctr -g MEM –C N:0-11 ./a.out
```

Using performance counters for diagnosing bad ccNUMA access locality

- **Intel Westmere EP node (2x6 cores):**



Only one memory BW per socket (“Uncore”)

```
+-------------------------------+----------+----------+     +----------+----------+
|            Metric             |  core 0  |  core 1  |     |  core 6  |  core 7  |
+-------------------------------+----------+----------+     +----------+----------+
|         Runtime [s]           | 0.730168 | 0.733754 |     | 0.732808 | 0.732943 |
|             CPI               | 10.4164  | 10.2654  |     | 10.5002  | 10.7641  |
| Memory bandwidth [MBytes/s]   | 11880.9  |    0     | ... | 11732.4  |    0     | ...
|  Remote Read BW [MBytes/s]    |   4219   |    0     |     | 4163.45  |    0     |
| Remote Write BW [MBytes/s]    | 1706.19  |    0     |     | 1705.09  |    0     |
|     Remote BW [MBytes/s]      | 5925.19  |    0     |     | 5868.54  |    0     |
+-------------------------------+----------+----------+     +----------+----------+
```

Half of BW comes from other socket!

If all fails…

optional

- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

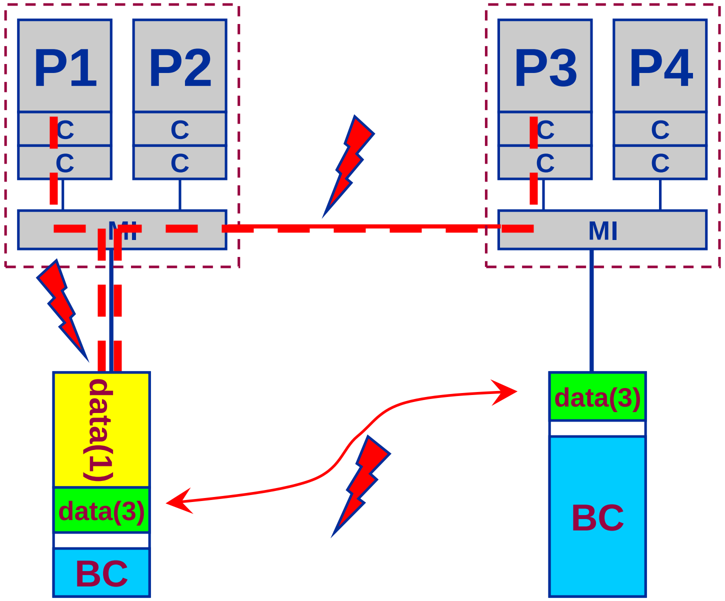
```
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
top - 14:18:25 up 92 days,  6:07,  2 users,  load average: 0.00, 0.02, 0.00
Mem:    4065564k total,  1149400k used,  2716164k free,    43388k buffers
Swap:   2104504k total,     2656k used,  2101848k free,  1038412k cached
```

ccNUMA problems beyond first touch:
Buffer cache

optional

- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - "sync" is not sufficient to drop buffer cache blocks



- **Remedies**
  - Drop FS cache pages after user job has run (admin's job)
    - seems to be automatic after aprun has finished on Crays
  - User can run "sweeper" code that allocates and touches all physical memory before starting the real application
  - `numactl` tool or `aprun` can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels

ccNUMA problems beyond first touch:
Buffer cache

optional

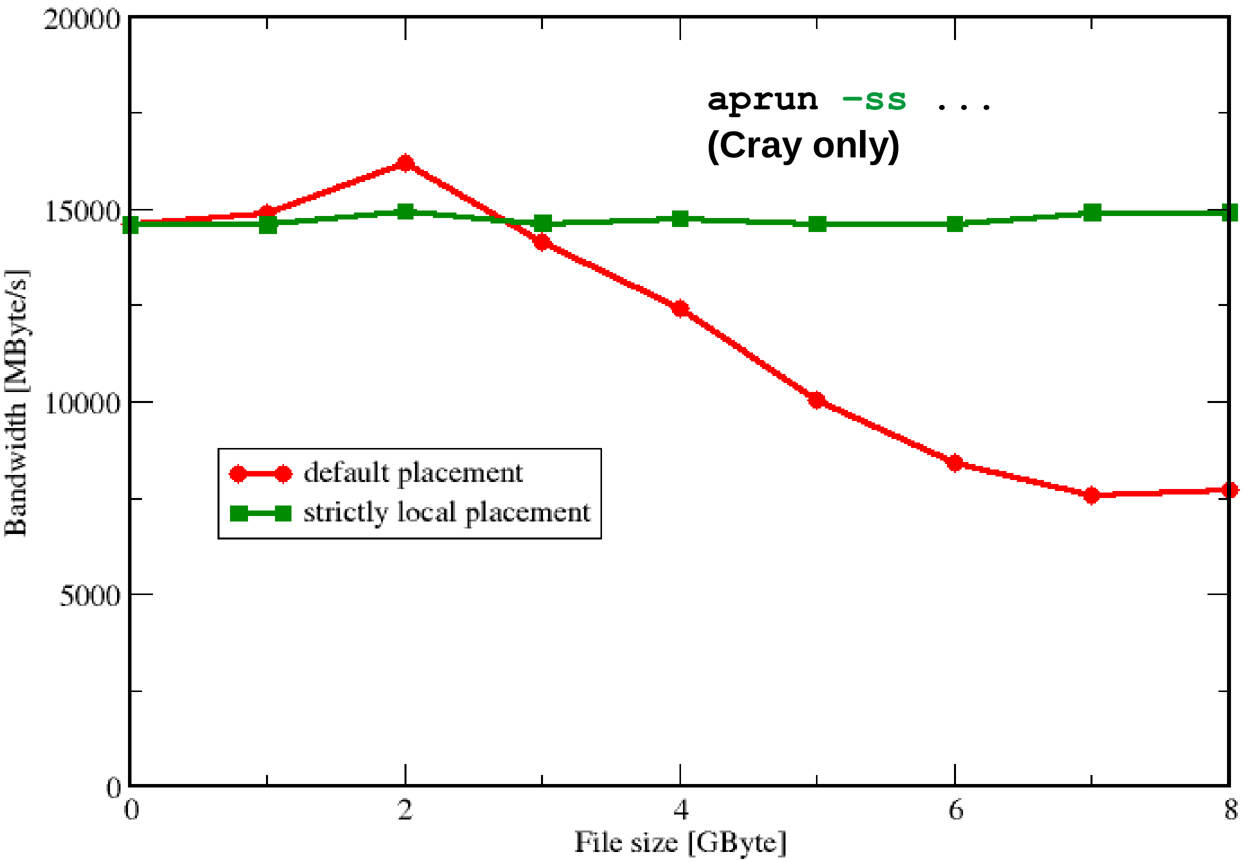
Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement

→ restrict to local domain if possible!



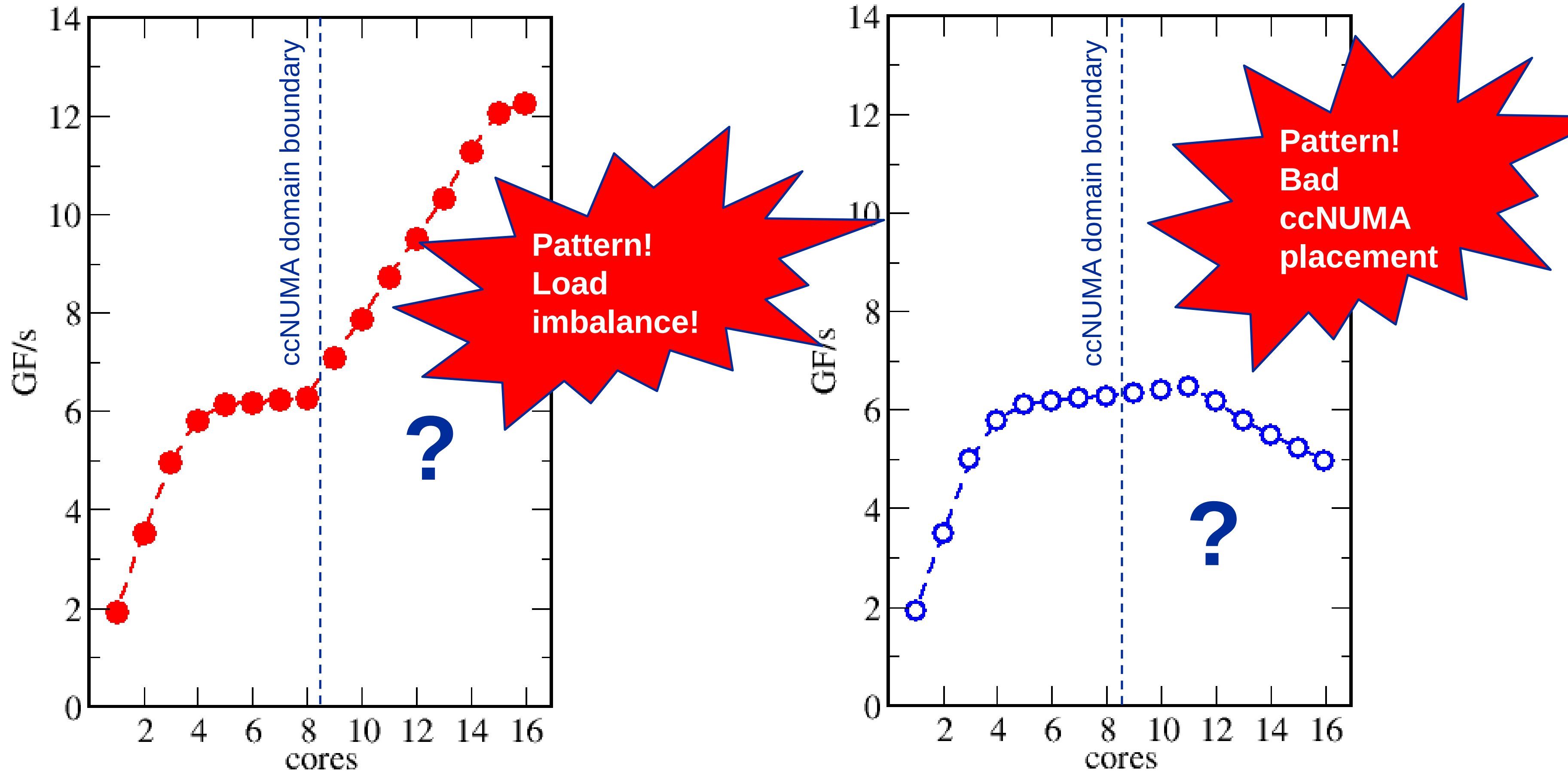
```
aprun -ss ...
```
(Cray only)

Back to the spMVM code: a little riddle

DLR1 matrix on 2x 8-core Sandy Bridge node

parallel first touch init.

serial init.

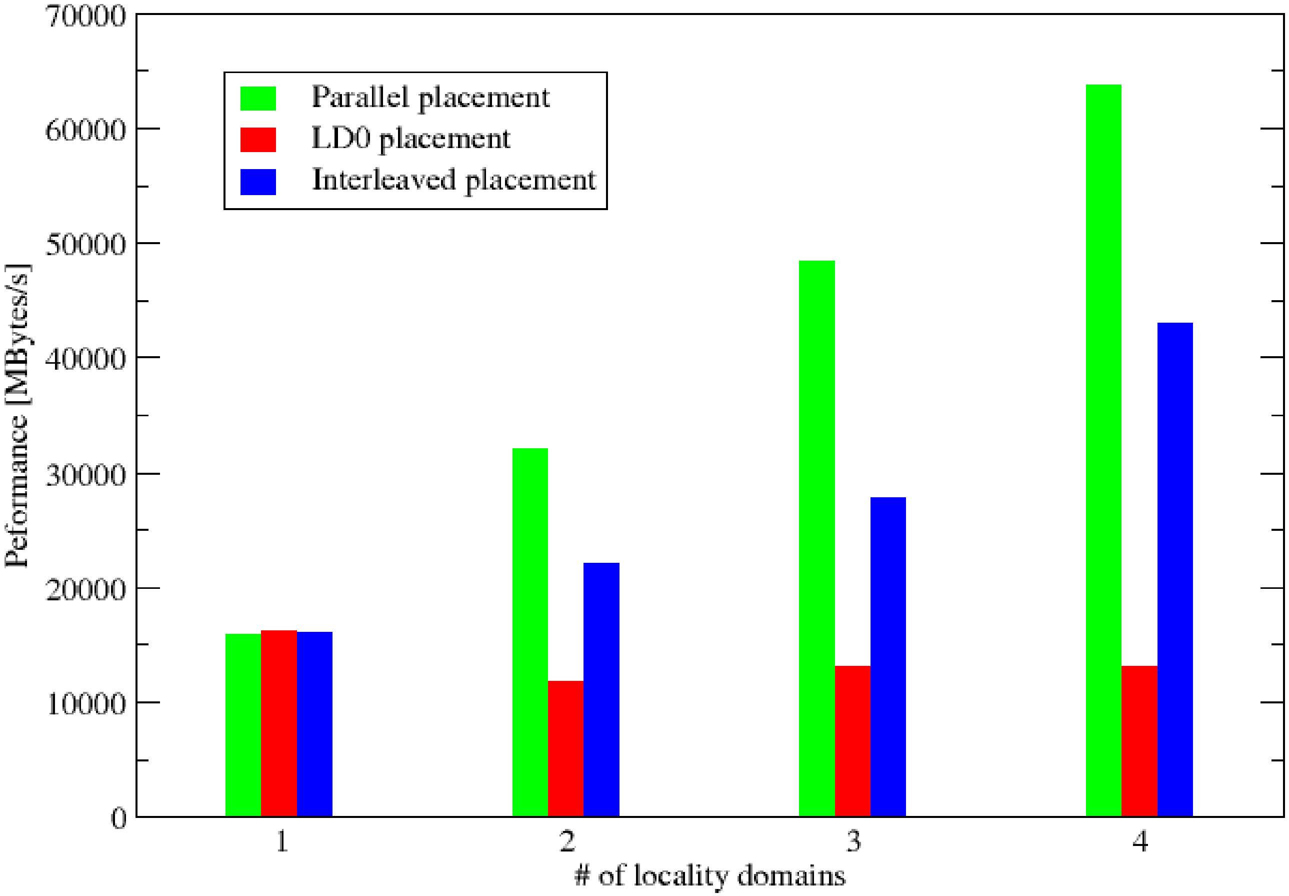


Pattern!
Load
imbalance!

Pattern!
Bad
ccNUMA
placement

The curse and blessing of interleaved placement:
OpenMP STREAM on a Cray XE6 Interlagos node

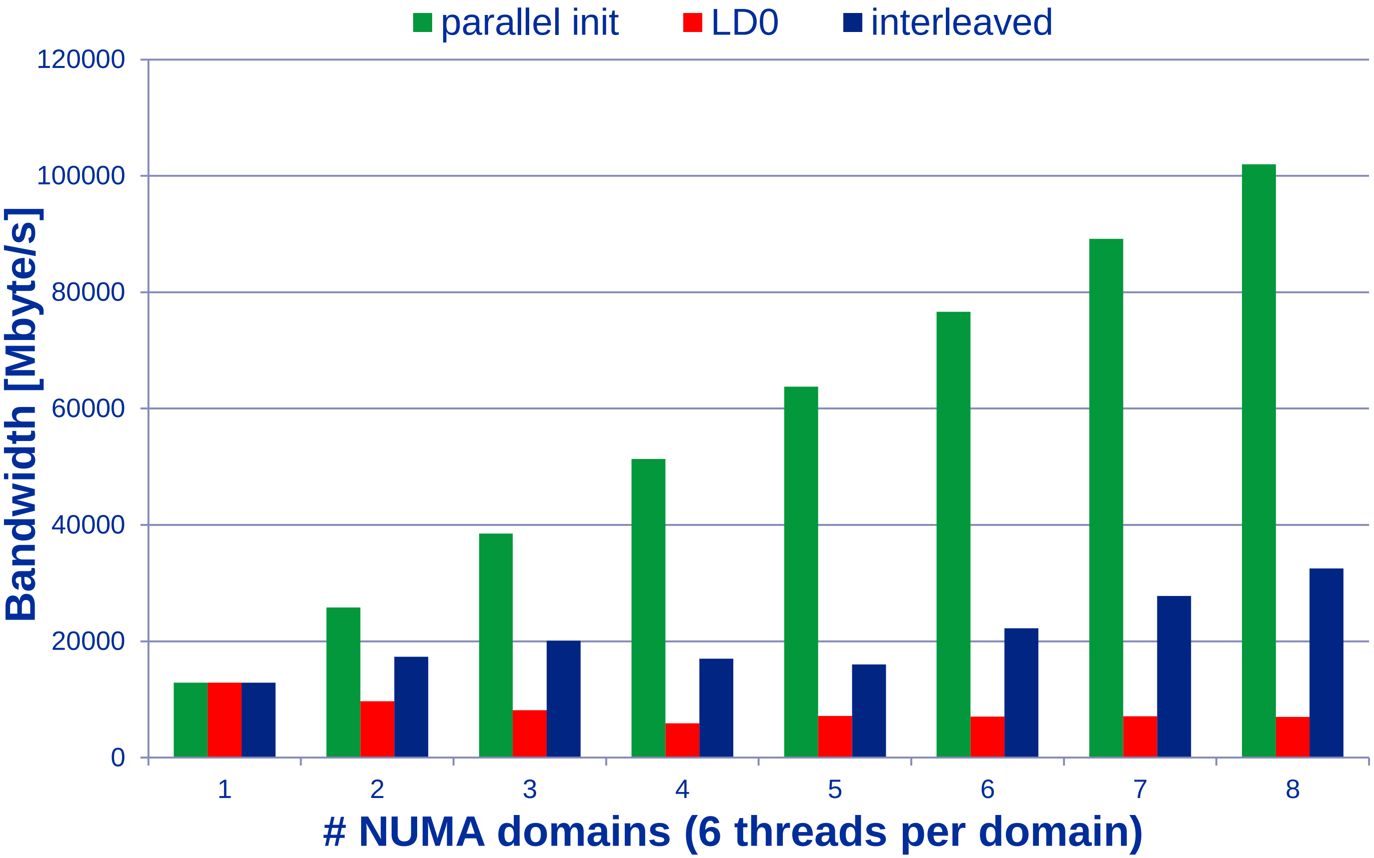
- **Parallel init: Correct parallel initialization**
- **LD0: Force data into LD0 via `numactl –m 0`**
- **Interleaved: `numactl --interleave <LD range>`**



The curse and blessing of interleaved placement:
OpenMP STREAM triad on 4-socket (48 core) Magny Cours node

optional

- **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

- **If dynamic scheduling cannot be avoided**
  - Distribute the data anyway, just do not use sequential placement!

- **OS file buffer cache may impact proper placement**

OpenMP performance issues on multicore

Barrier synchronization overhead

Topology dependence

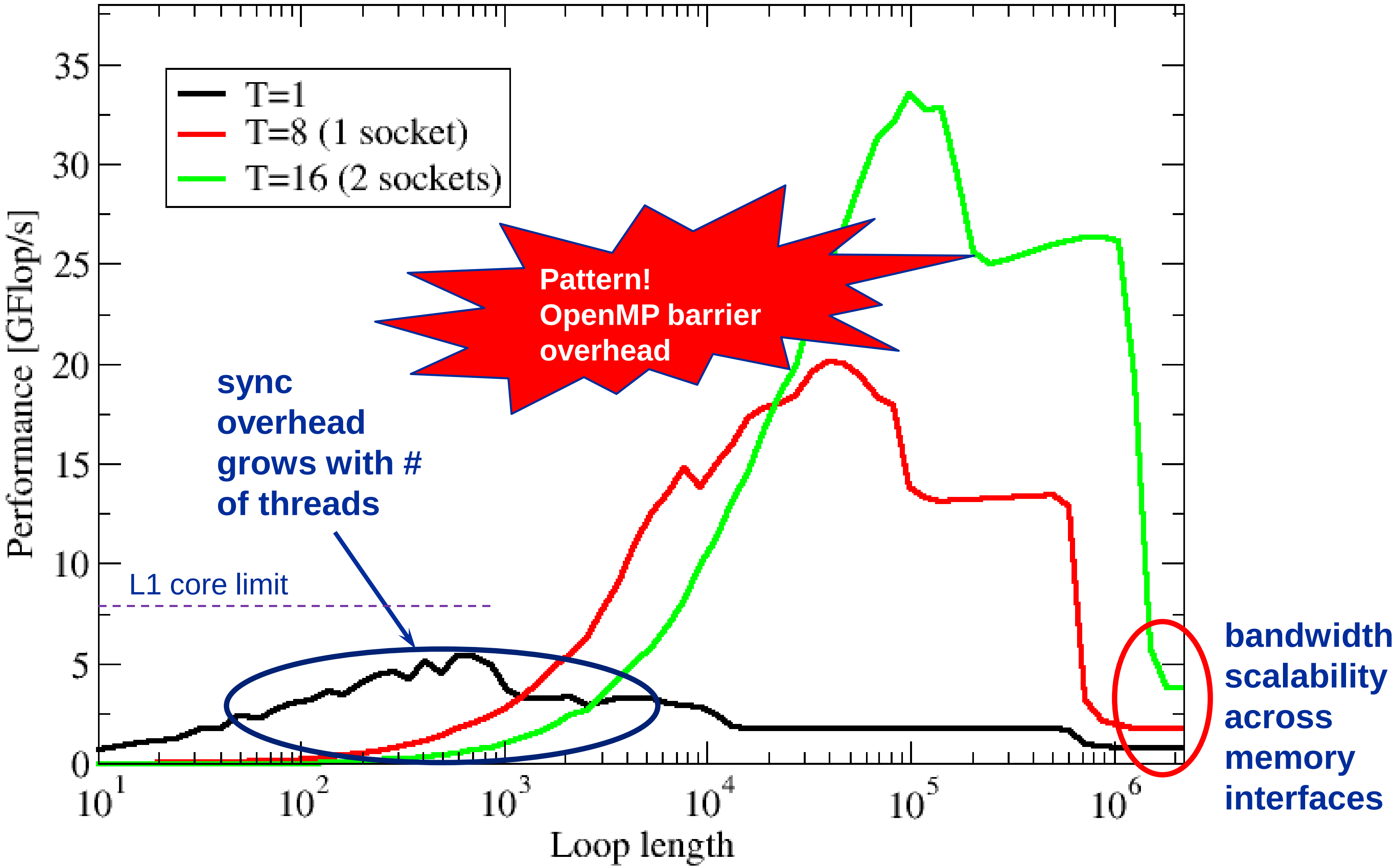
The OpenMP-parallel vector triad benchmark

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)



Pattern! OpenMP barrier overhead

sync overhead grows with # of threads

bandwidth scalability across memory interfaces

Welcome to the multi-/many-core era

Synchronization of threads may be expensive!

```
!$OMP PARALLEL …
…
!$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)

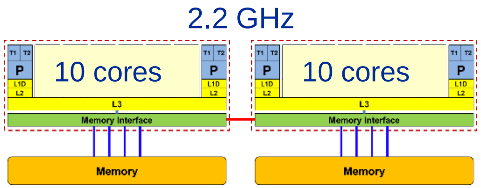
On x86 systems there is no hardware support for synchronization!

- **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 Intel Xeon Phi

Barrier overhead in CPU cycles

2 threads on distinct cores: 1936

	SMT1	SMT2	SMT3	SMT4
One core	n/a	**1597**	2825	3557
Full chip	10604	12800	15573	**18490**

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

3.75x cores (16 vs 60) on Phi

2x more operations per cycle on Phi

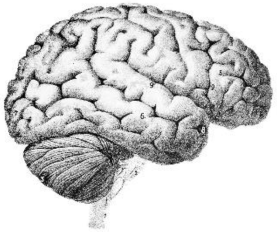
→ 2 · 3.75 = 7.5x more work done on Xeon Phi per cycle

2.7x more barrier penalty (cycles) on Phi

→ One barrier causes 2.7 · 7.5 ≈ 20x 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 Röhl
Faisal Shahzad
Julian Hammer

SPPEXA

KONWIHR III

THANK YOU.

Bundesministerium
für Bildung
und Forschung

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 http://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.

References

Book:

- G. Hager and G. Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Computational Science Series, 2010. ISBN 978-1439811924 http://www.hpc.rrze.uni-erlangen.de/HPC4SE/

Papers:

- J. Hofmann, D. Fey, M. Riedmann, J. Eitzinger, G. Hager, and G. Wellein: Performance analysis of the Kahan-enhanced scalar product on current multi- and manycore processors. Accepted for publication in Concurrency & Computation: Practice & Experience. Preprint: arXiv:1604.01890
- M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske: Increasing the performance of the Jacobi-Davidson method by blocking. SIAM Journal on Scientific Computing, **37**(6), C697–C722 (2015). DOI: 10.1137/140976017, Preprint:http://elib.dlr.de/89980/
- T. M. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. E. Keyes: Multicore-optimized wavefront diamond blocking for optimizing stencil updates. SIAM Journal on Scientific Computing **37**(4), C439-C464 (2015). DOI: 10.1137/140991133, Preprint: arXiv:1410.3060
- J. Hammer, G. Hager, J. Eitzinger, and G. Wellein: Automatic Loop Kernel Analysis and Performance Modeling With Kerncraft. Proc. PMBS15, the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, in conjunction with ACM/IEEE Supercomputing 2015 (SC15), November 16, 2015, Austin, TX. DOI: 10.1145/2832087.2832092, Preprint: arXiv:1509.03778

References

Papers continued:

- M. Kreutzer, G. Hager, G. Wellein, A. Pieper, A. Alvermann, and H. Fehske: Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems. Proc. IPDPS15. DOI: 10.1109/IPDPS.2015.76, Preprint: arXiv:1410.5242
- M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein: Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations. Concurrency and Computation: Practice and Experience (2015). DOI: 10.1002/cpe.3489 Preprint: arXiv:1304.7664
- H. Stengel, J. Treibig, G. Hager, and G. Wellein: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. Proc. ICS15, DOI: 10.1145/2751205.2751240, Preprint: arXiv:1410.5010
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: A unified sparse matrix data format for modern processors with wide SIMD units. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: 10.1137/130930352, Preprint: arXiv:1307.6209
- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Computation and Concurrency: Practice and Experience (2013). DOI: 10.1002/cpe.3180, Preprint: arXiv:1208.2908
- J. Treibig, G. Hager and G. Wellein: Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. DOI: 10.1007/978-3-642-36949-0_50. Preprint: arXiv:1206.3738

References

Papers continued:

- M. Wittmann, T. Zeiser, G. Hager, and G. Wellein: Comparison of Different Propagation Steps for Lattice Boltzmann Methods. Computers & Mathematics with Applications (Proc. ICMMES 2011). Available online, DOI: 10.1016/j.camwa.2012.05.002. Preprint:arXiv:1111.0922
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann and A. R. Bishop: Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. Workshop on Large-Scale Parallel Processing 2012 (LSPP12), DOI: 10.1109/IPDPSW.2012.211
- J. Treibig, G. Hager, H. Hofmann, J. Hornegger and G. Wellein: Pushing the limits for medical image reconstruction on recent standard multicore processors. International Journal of High Performance Computing Applications, (published online before print). DOI: 10.1177/1094342012442424
- G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. Proc. COMPSAC 2009. DOI: 10.1109/COMPSAC.2009.82
- M. Wittmann, G. Hager, J. Treibig and G. Wellein: Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. Parallel Processing Letters **20** (4), 359-376 (2010). DOI: 10.1142/S0129626410000296. Preprint: arXiv:1006.3148
- J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. DOI: 10.1109/ICPPW.2010.38. Preprint: arXiv:1004.4431

References

Papers continued:

- G. Schubert, H. Fehske, G. Hager, and G. Wellein: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters 21(3), 339-358 (2011).
DOI: 10.1142/S0129626411000254
- J. Treibig, G. Wellein and G. Hager: Efficient multicore-aware parallelization strategies for iterative stencil computations. Journal of Computational Science 2 (2), 130-137 (2011). DOI 10.1016/j.jocs.2011.01.010
- K. Iglberger, G. Hager, J. Treibig, and U. Rüde: Expression Templates Revisited: A Performance Analysis of Current ET Methodologies. SIAM Journal on Scientific Computing **34**(2), C42-C69 (2012). DOI: 10.1137/110830125, Preprint: arXiv:1104.1729
- K. Iglberger, G. Hager, J. Treibig, and U. Rüde: High Performance Smart Expression Template Math Libraries. 2nd International Workshop on New Algorithms and Programming Models for the Manycore Era (APMM 2012) at HPCS 2012, July 2-6, 2012, Madrid, Spain. DOI: 10.1109/HPCSim.2012.6266939
- J. Habich, T. Zeiser, G. Hager and G. Wellein: Performance analysis and optimization strategies for a D3Q19 Lattice Boltzmann Kernel on nVIDIA GPUs using CUDA. Advances in Engineering Software and Computers & Structures 42 (5), 266–272 (2011). DOI: 10.1016/j.advengsoft.2010.10.007
- J. Treibig, G. Hager and G. Wellein: Multicore architectures: Complexities of performance prediction for Bandwidth-Limited Loop Kernels on Multi-Core Architectures.
DOI: 10.1007/978-3-642-13872-0_1, Preprint: arXiv:0910.4865.

References

Papers continued:

- G. Hager, G. Jost, and R. Rabenseifner: Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In: Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 4-7, 2009. PDF
- R. Rabenseifner and G. Wellein: Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. International Journal of High Performance Computing Applications **17**, 49-62, February 2003.
  DOI:10.1177/1094342003017001005