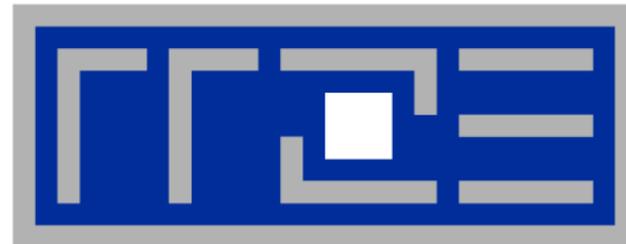


For final slides see:

<http://goo.gl/3pSrVL>



# Node-Level Performance Engineering

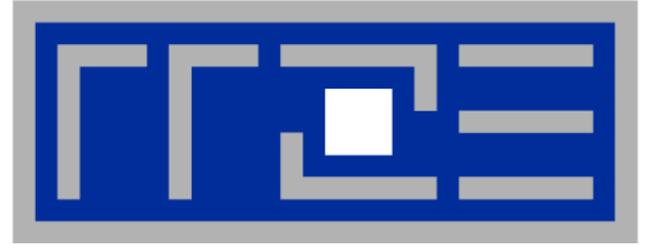
Georg Hager, Jan Treibig, Gerhard Wellein  
Erlangen Regional Computing Center (RRZE)  
and Department of Computer Science  
University of Erlangen-Nuremberg

ISC14 full-day tutorial  
June 22, 2014  
Leipzig, Germany





GW	▪ <b>Preliminaries</b>	09:00
	▪ <b>Introduction to multicore architecture</b> <ul style="list-style-type: none"><li>▪ Cores, caches, chips, sockets, ccNUMA, SIMD</li></ul>	
JT	▪ <b>LIKWID tools</b>	11:00
JT	▪ <b>Microbenchmarking for architectural exploration</b> <ul style="list-style-type: none"><li>▪ Streaming benchmarks: throughput mode</li><li>▪ Streaming benchmarks: work sharing</li><li>▪ Roadblocks for scalability: Saturation effects and OpenMP overhead</li></ul>	11:30
	▪ <b>Node-level performance modeling (part I)</b> <ul style="list-style-type: none"><li>▪ The Roofline Model</li></ul>	13:00
	▪ <b>Lunch break</b>	
GHa	▪ <b>Node-level performance modeling (part II)</b> <ul style="list-style-type: none"><li>▪ Case study: 3D Jacobi solver and model-guided optimization</li></ul>	14:00
JT	▪ <b>DEMO</b>	16:00
GHa	▪ <b>Optimal resource utilization</b> <ul style="list-style-type: none"><li>▪ SIMD parallelism</li><li>▪ ccNUMA</li><li>▪ Simultaneous multi-threading (SMT)</li></ul>	16:30
		18:00



**Prelude:**  
**Scalability 4 the win!**



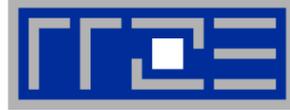
## **Lore 1**

**In a world of highly parallel computer architectures only highly scalable codes will survive**

## **Lore 2**

**Single core performance no longer matters since we have so many of them and use scalable codes**

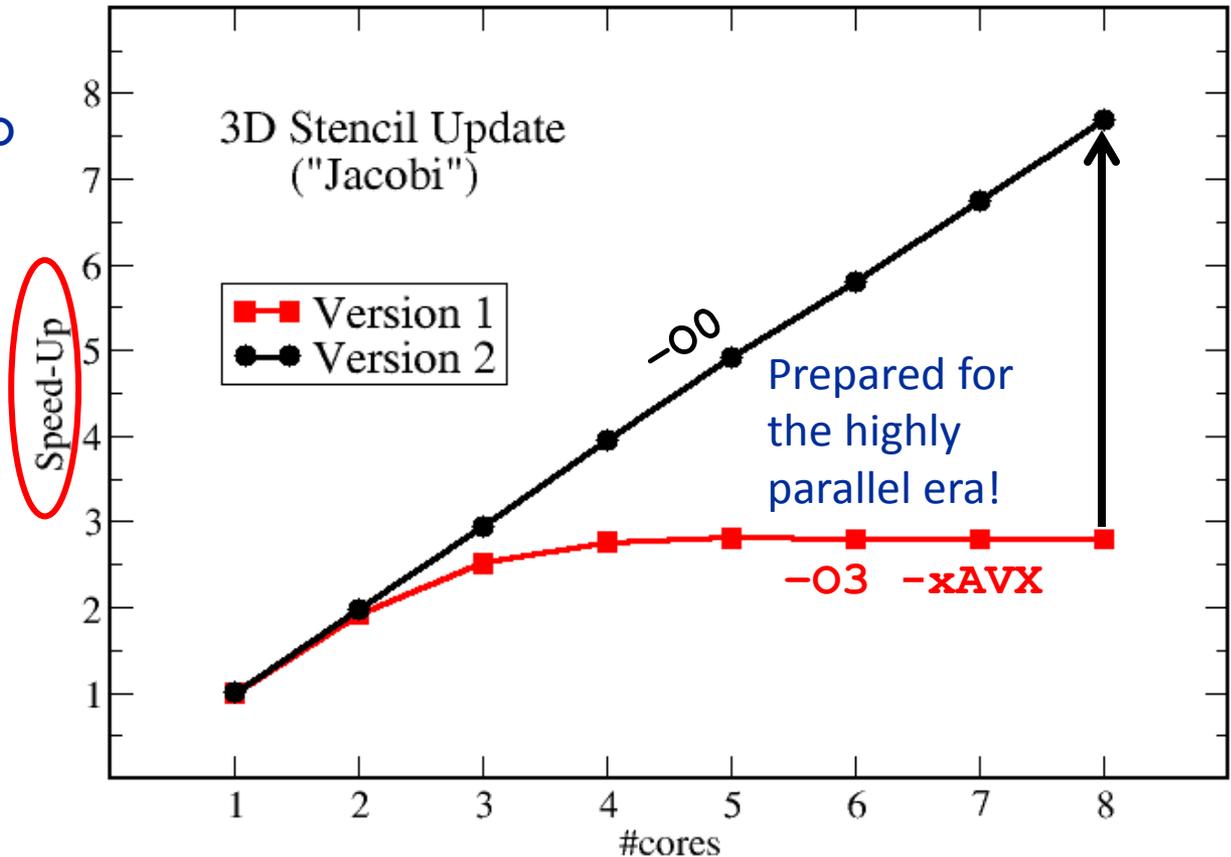
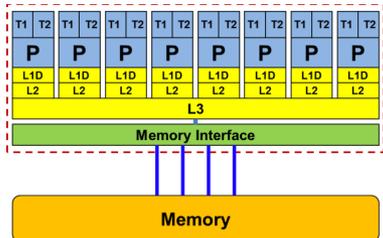
# Scalability Myth: Code scalability is the key issue



```

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

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



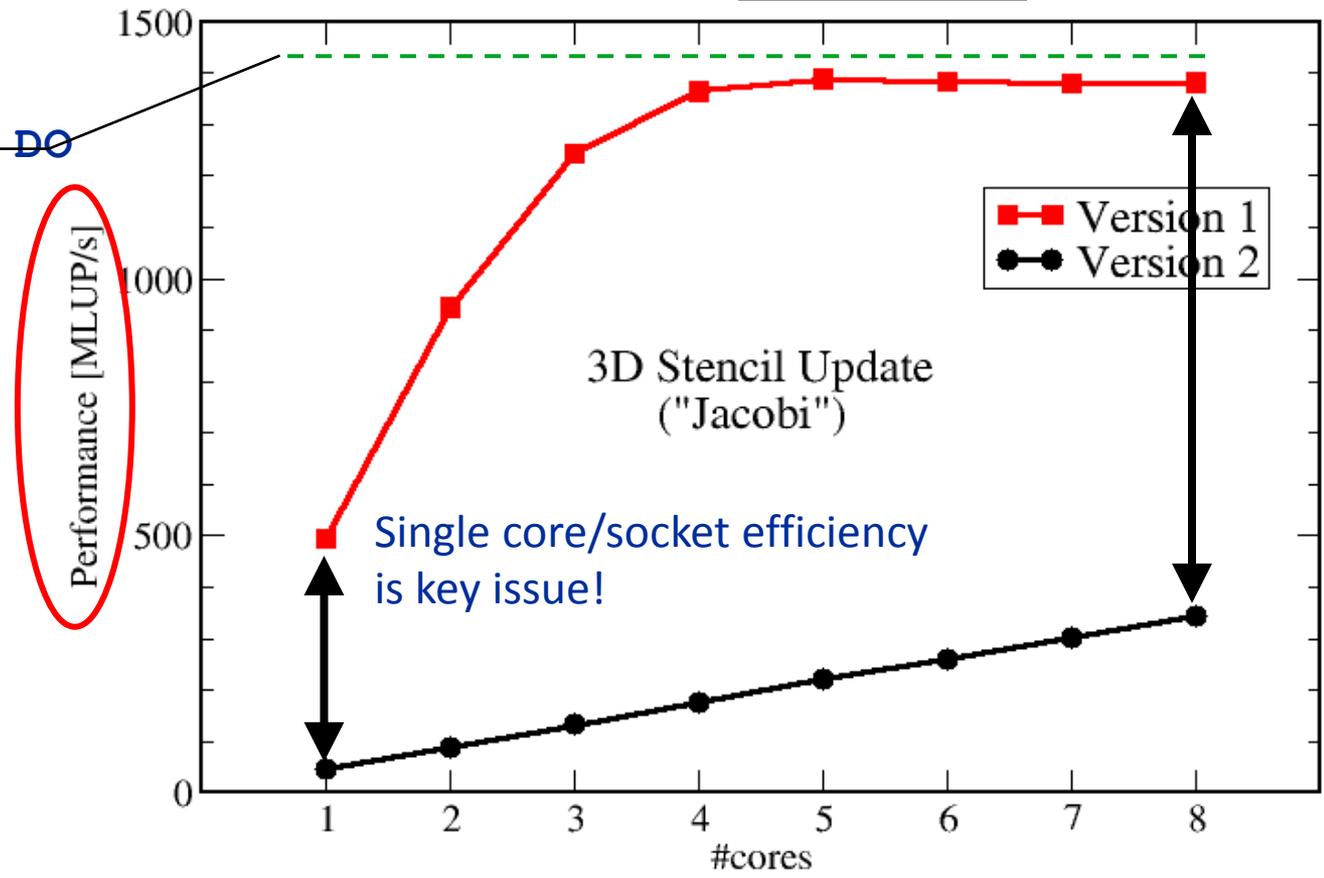
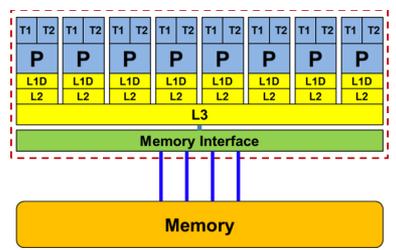
# Scalability Myth: Code scalability is the key issue



```

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

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





- **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



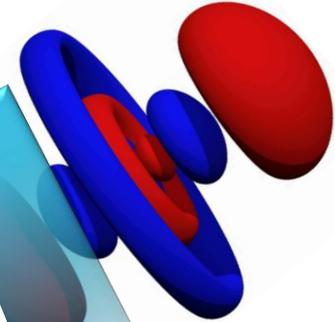
Newtonian mechanics



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

Fails @ small scales!

Nonrelativistic quantum mechanics

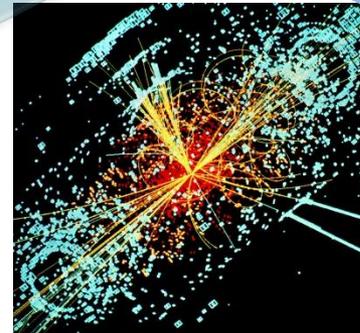


**If a model fails,  
we learn something!**

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

Fails @ even smaller scales!

Relativistic quantum field theory

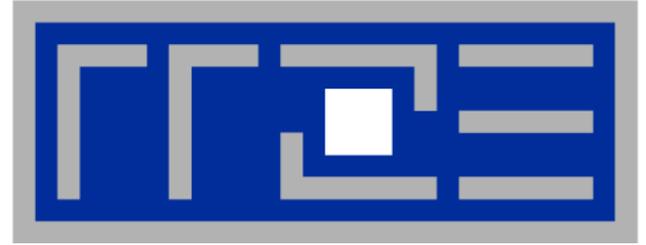


$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_c$$



**There is no alternative to knowing what is going on  
between your code and the hardware**

**Without performance modeling,  
optimizing code is like stumbling in the dark**



# **Introduction: Modern node architecture**

**Multi- and manycore chips and nodes**

**A glance at basic core features**

**Caches and data transfers through the memory hierarchy**

**Memory organization**

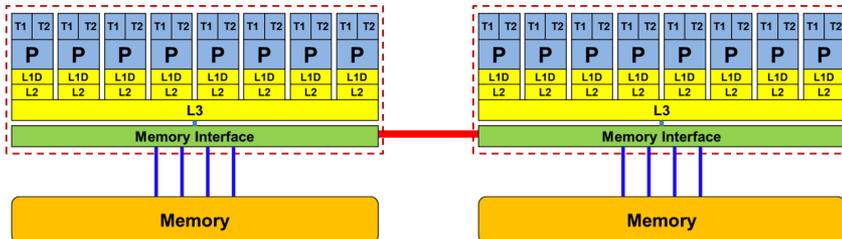
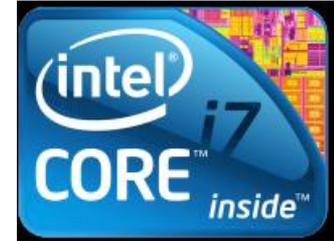
**Accelerators**

**Programming models**

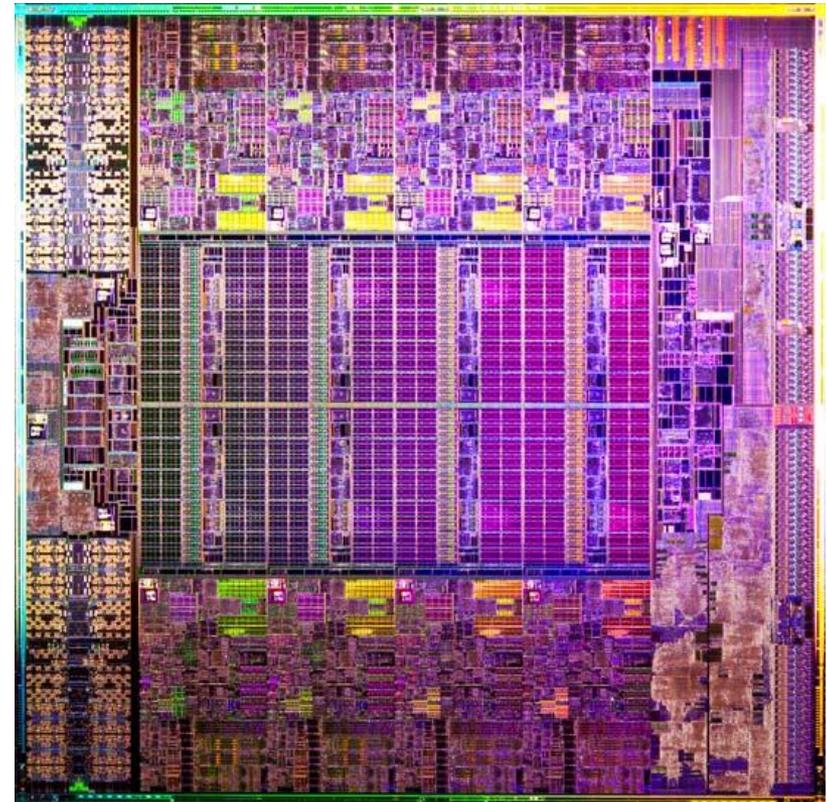
# Multi-Core: Intel Xeon 2600 (2012)



- Xeon 2600 “Sandy Bridge EP”:  
8 cores running at 2.7 GHz (max 3.2 GHz)
- Simultaneous Multithreading  
→ reports as 16-way chip
- **2.3 Billion** Transistors / 32 nm
- Die size: 435 mm<sup>2</sup>

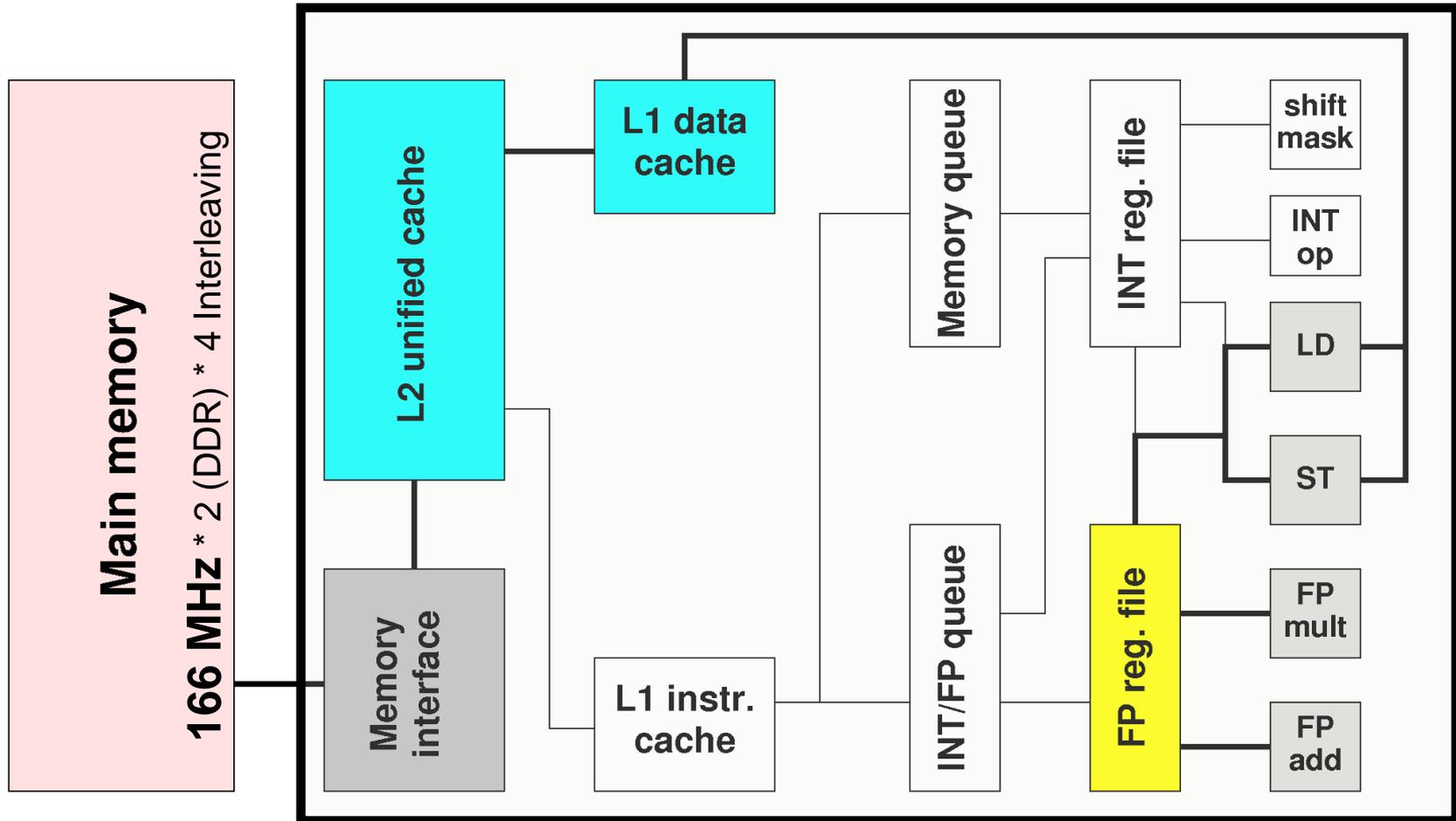


2-socket server





- (Almost) the same basic design in all modern systems

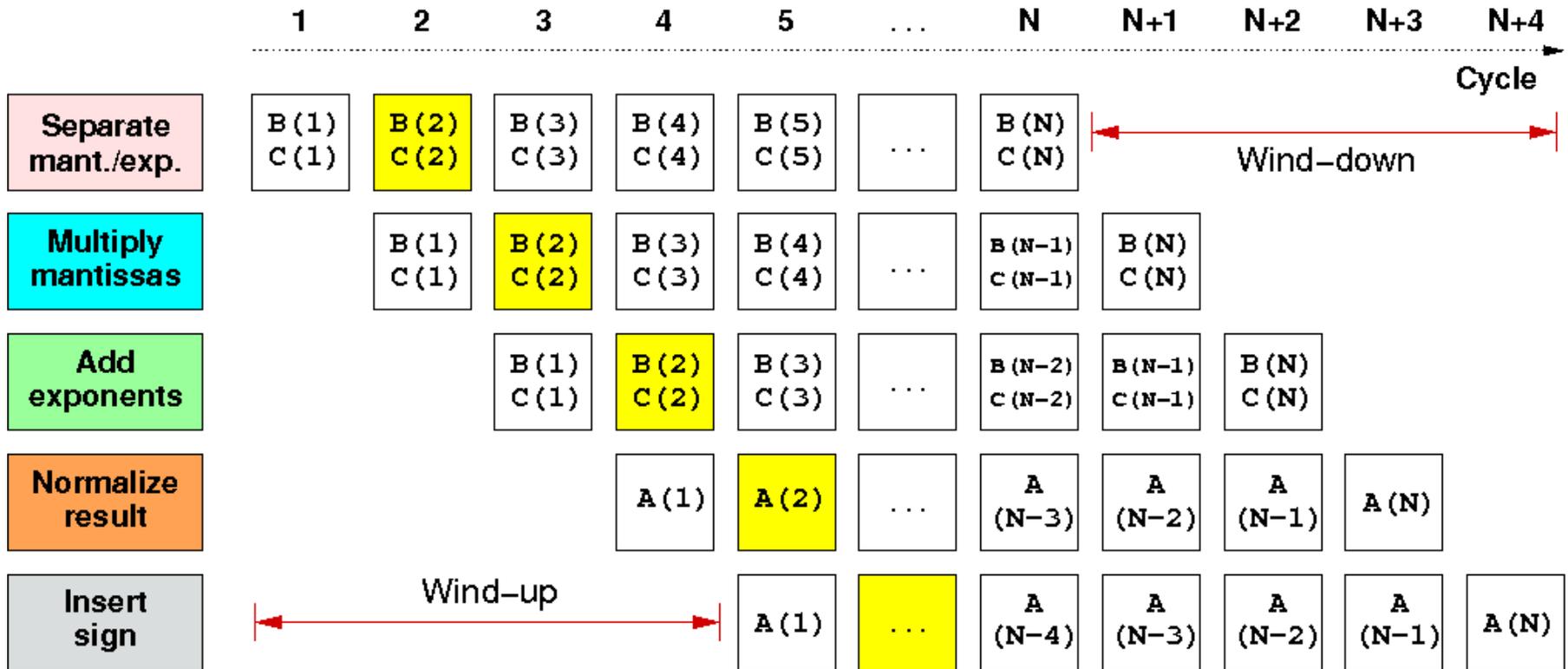


Not shown: most of the control unit, e.g. instruction fetch/decode, branch prediction,...



- **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 ( $\#Instructions \gg$  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: $A(i)=B(i)*C(i)$ ; $i=1,\dots,N$



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

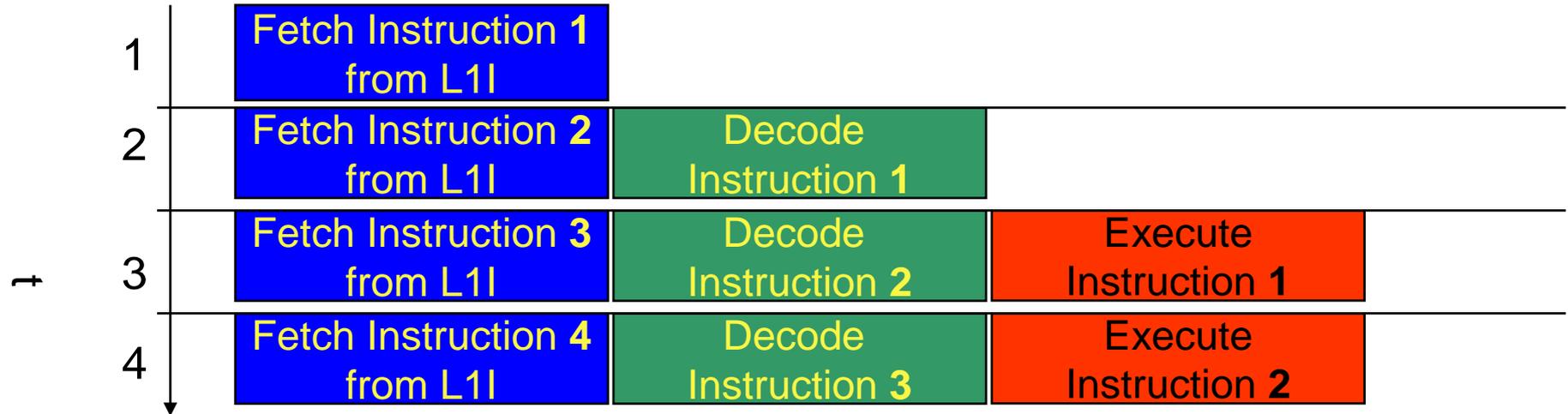
Wind-up/-down phases: Empty pipeline stages



- 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):

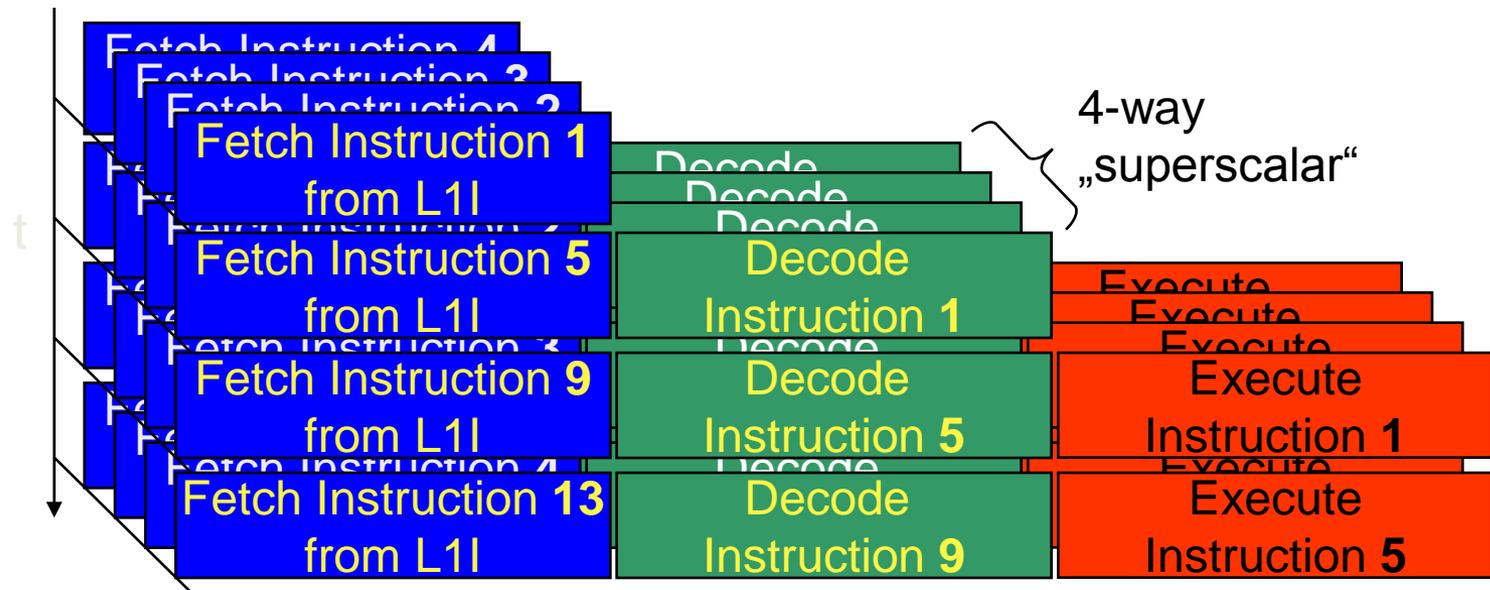


...

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



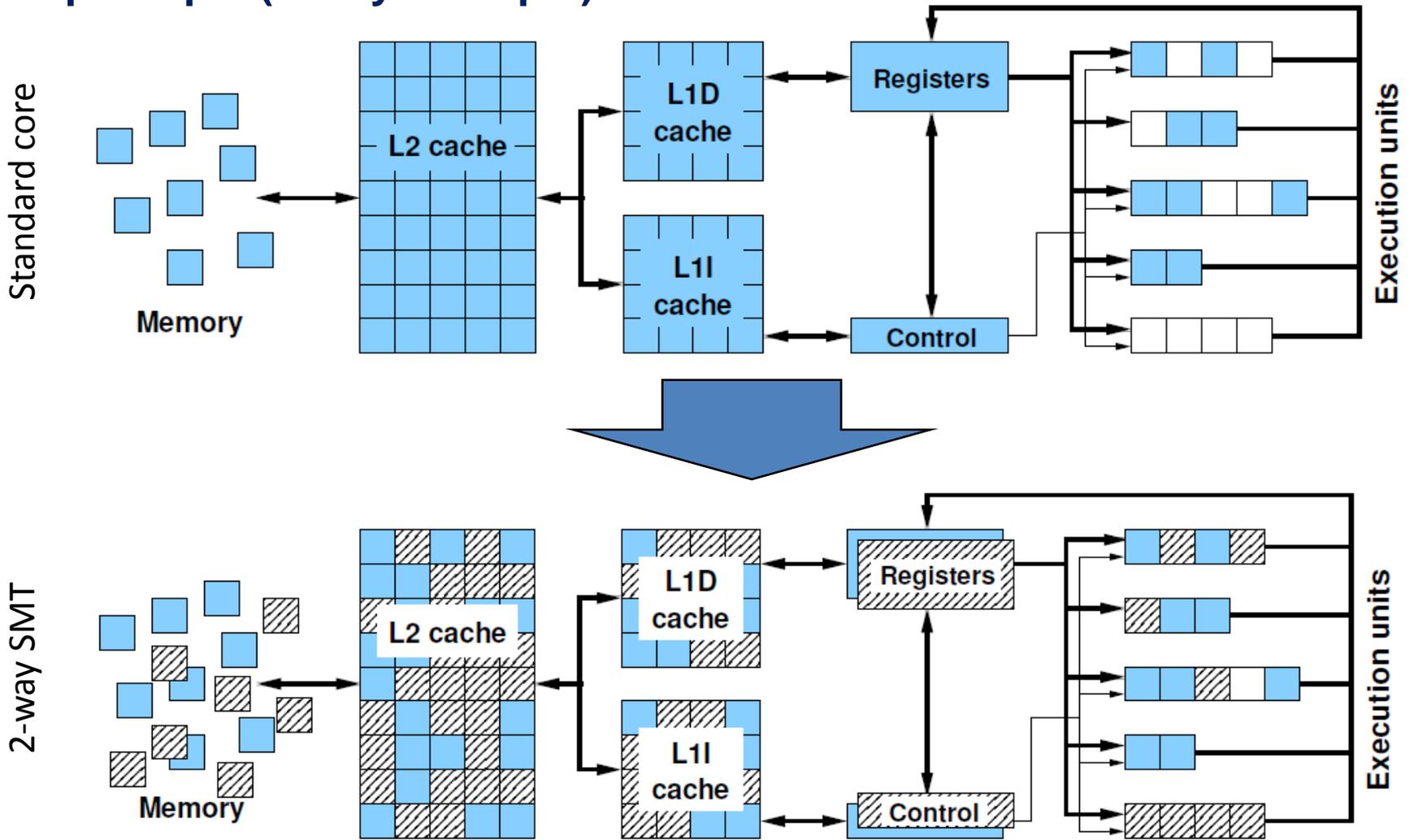
- Multiple units enable use of **I**nstruction **L**evel **P**arallelism (ILP):  
Instruction stream is “parallelized” on the fly



- 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

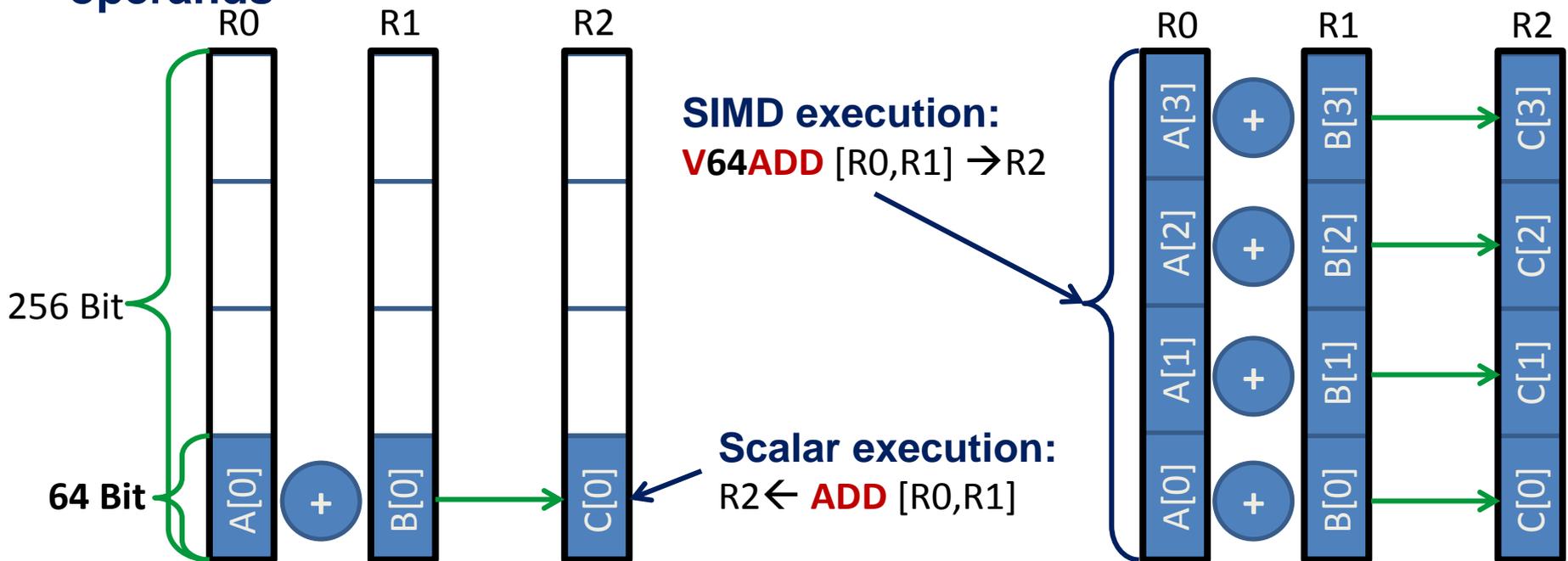


## SMT principle (2-way example):



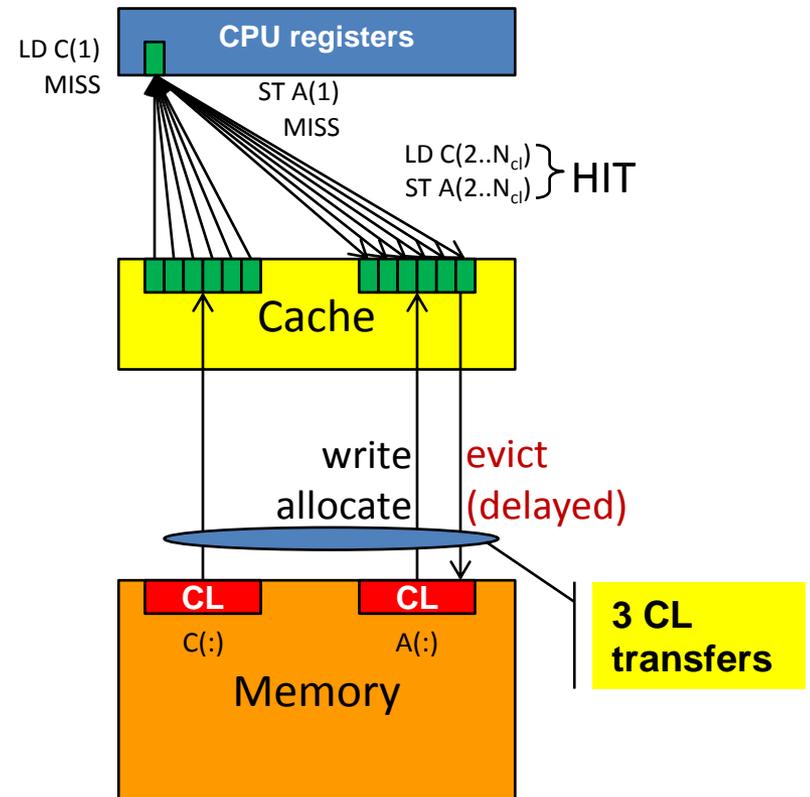


- **Single Instruction Multiple Data (SIMD)** operations allow the **concurrent execution of the same operation on “wide” registers**
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision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





- 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  $A(:) = C(:)$

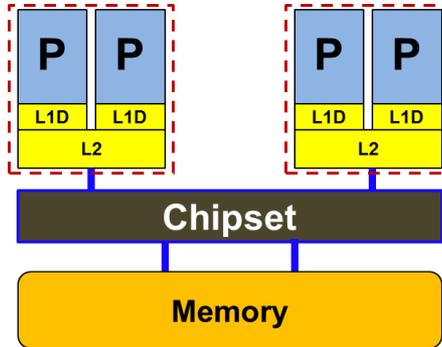


# Commodity cluster nodes: From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes



## Yesterday (2006): UMA

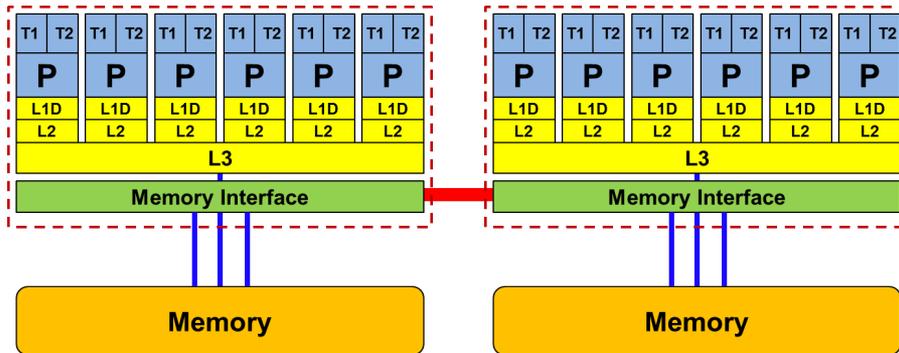


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system “anisotropy”

## Today: ccNUMA



Cache-coherent Non-Uniform Memory Architecture (**ccNUMA**)

**HT / QPI** provide scalable bandwidth at the price of ccNUMA architectures: *Where does my data finally end up?*

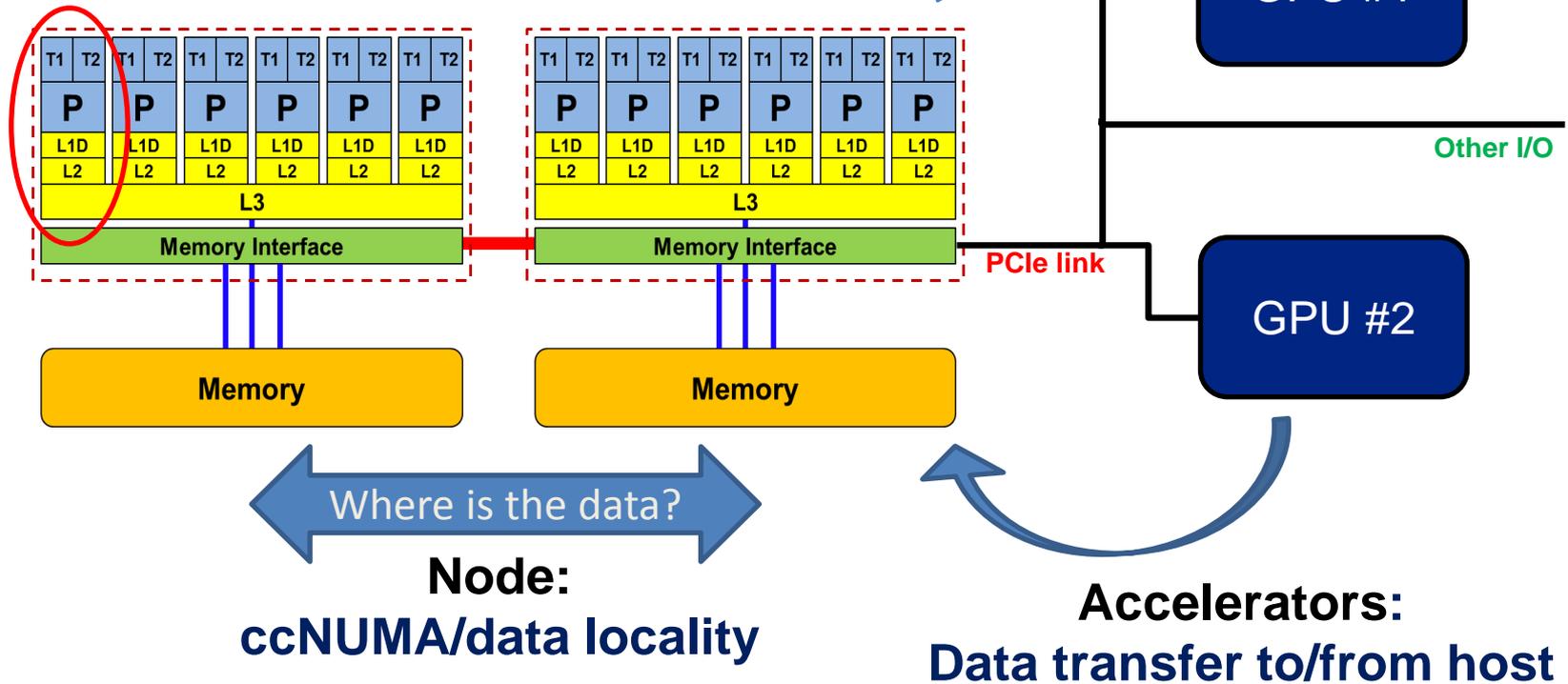


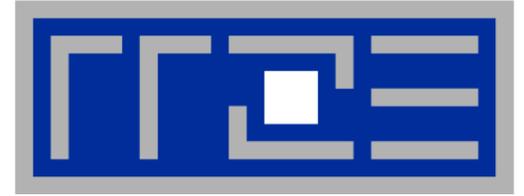


Heterogeneous programming is here to stay!  
SIMD + OpenMP + MPI + CUDA, OpenCL,...

**Core:**  
SIMD vectorization  
SMT

**Socket:**  
Parallelization  
Shared Resources





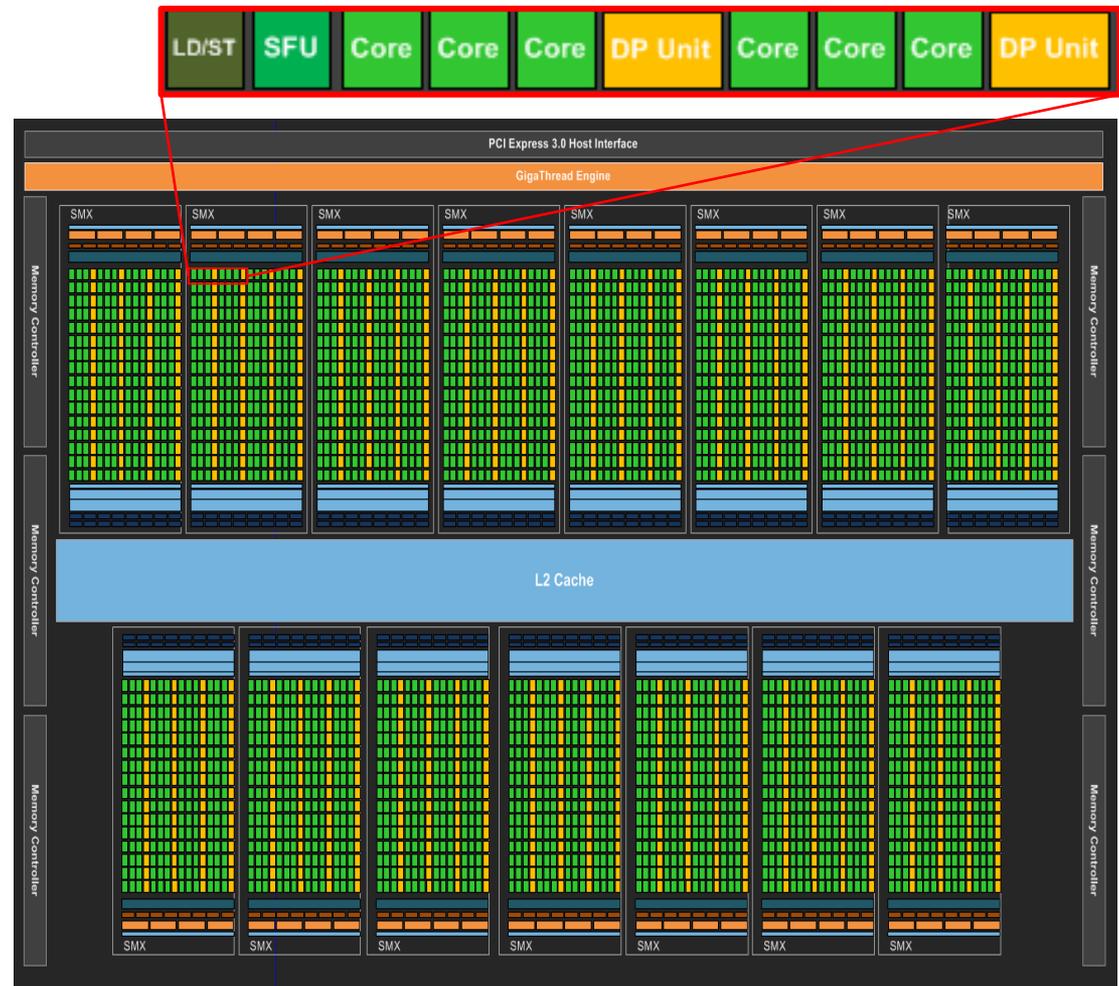
**Interlude:**  
**A glance at current accelerator technology**

# 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**



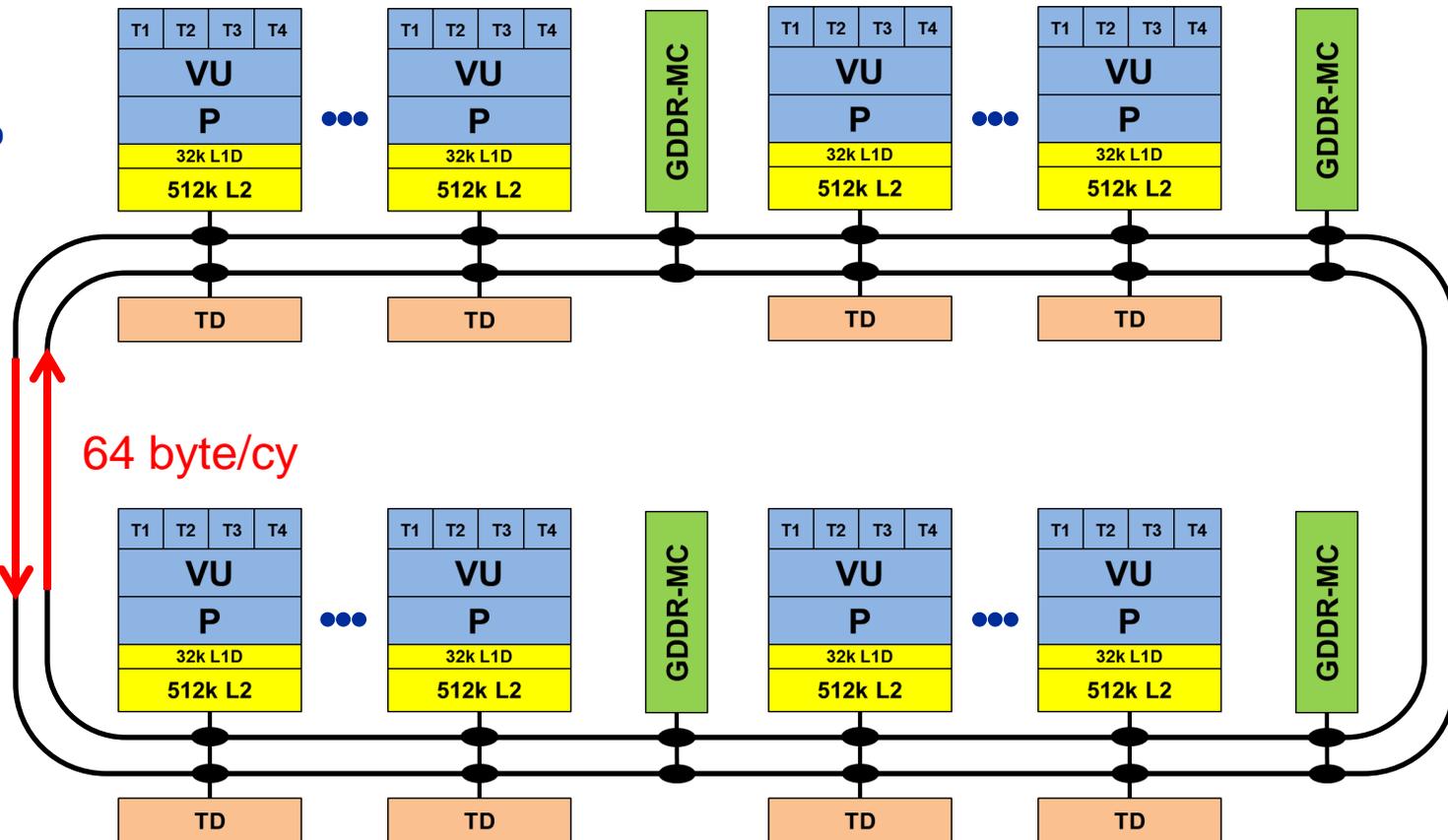
© NVIDIA Corp. Used with permission.

# Intel Xeon Phi block diagram



## Architecture

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



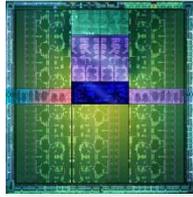
## Intel Xeon Phi

- **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)



- TOP7: “Stampede” at Texas Center for Advanced Computing

**TOP500  
rankings  
Nov 2012**

- TOP1: “Titan” at Oak Ridge National Laboratory

# Trading single thread performance for parallelism:

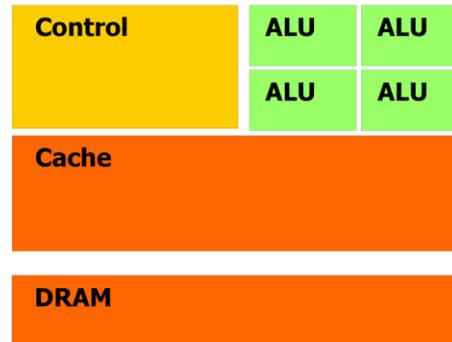
## GPGPUs vs. CPUs



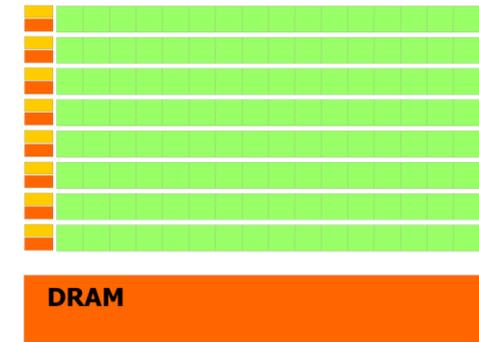
### GPU vs. CPU

light speed estimate:

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



CPU



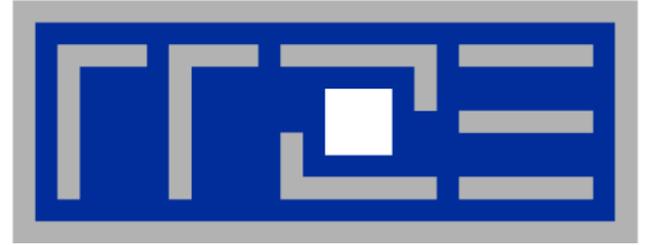
GPU

	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 <sup>+</sup> /core	52.8 GFlop/s	43.2 GFlop/s	1.4 GFlop/s
Threads@STREAM	<4	<16	>8000?
Total performance <sup>+</sup>	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)	<b>7.1 Billion/250W</b>

<sup>+</sup> Single Precision

\* Includes on-chip GPU and PCI-Express

Complete compute device

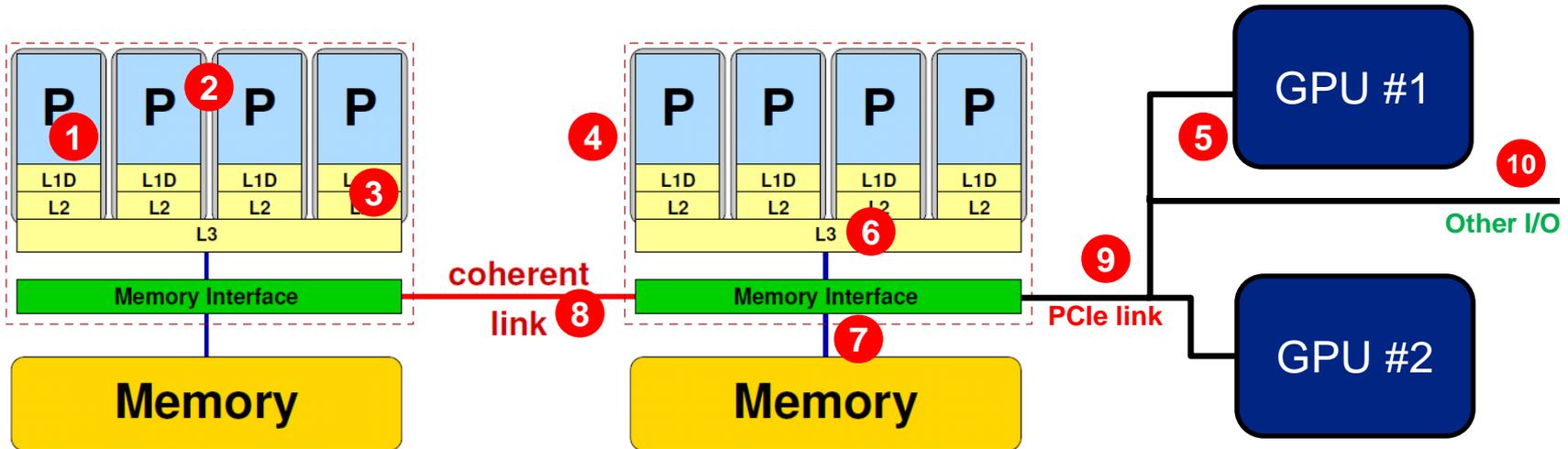


# **Node topology and programming models**

# 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?**



- **Shared-memory (intra-node)**
  - **Good old MPI** (current standard: 3.0)
  - **OpenMP** (current standard: 4.0)
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs,... you name it
- **Distributed-memory (inter-node)**
  - **MPI** (current standard: 3.0)
  - PVM (gone)
- **Hybrid**
  - **Pure MPI**
  - MPI+OpenMP
  - MPI + any shared-memory model
  - MPI (+OpenMP) + CUDA/OpenCL/...

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

# Parallel programming models:

## Pure MPI



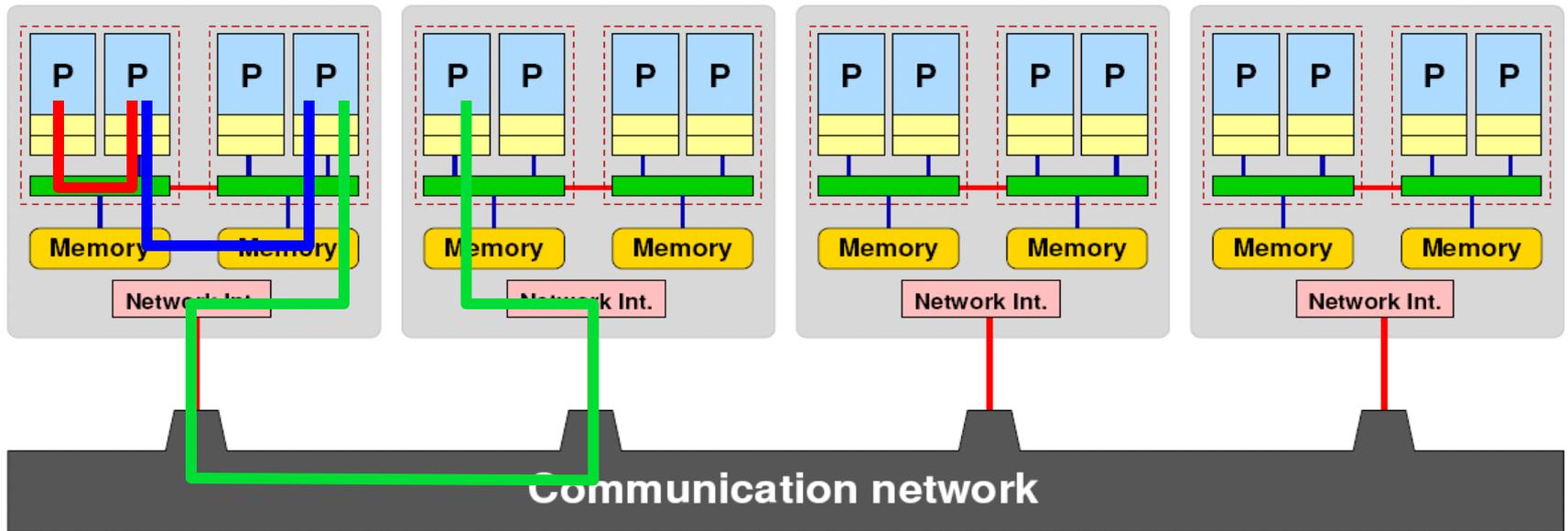
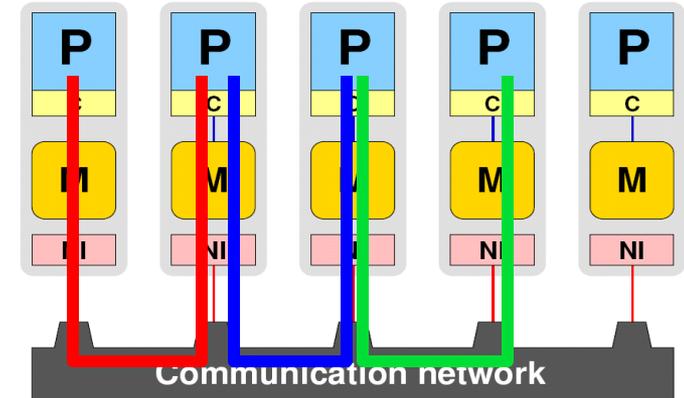
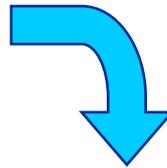
- Machine structure is invisible to user:

- 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

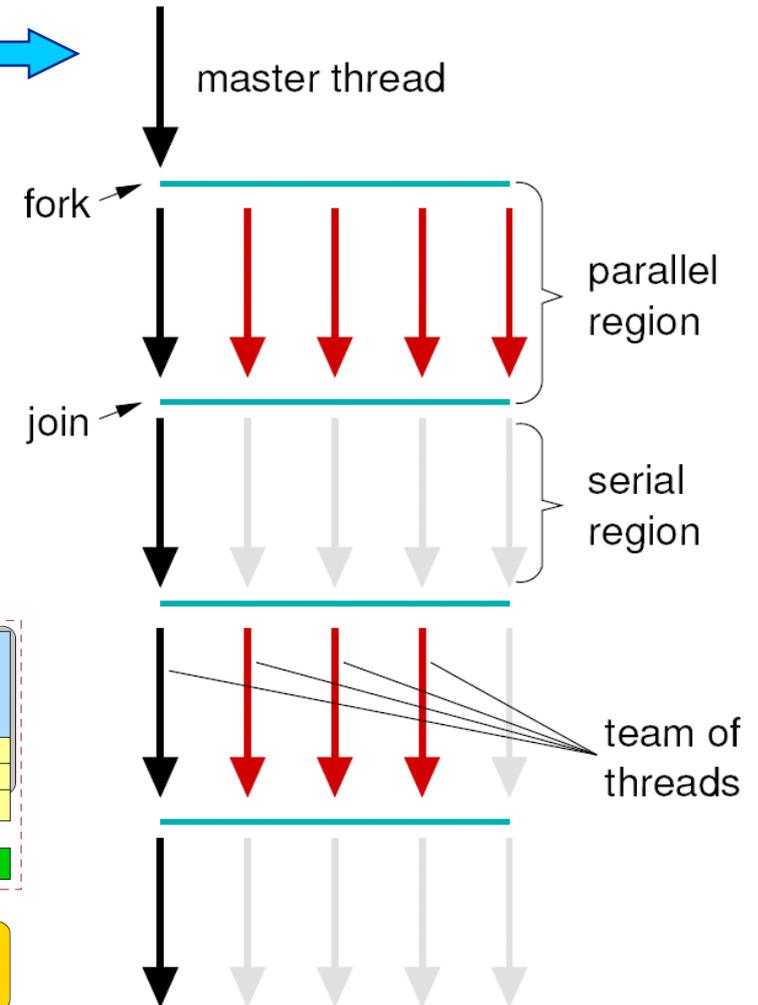
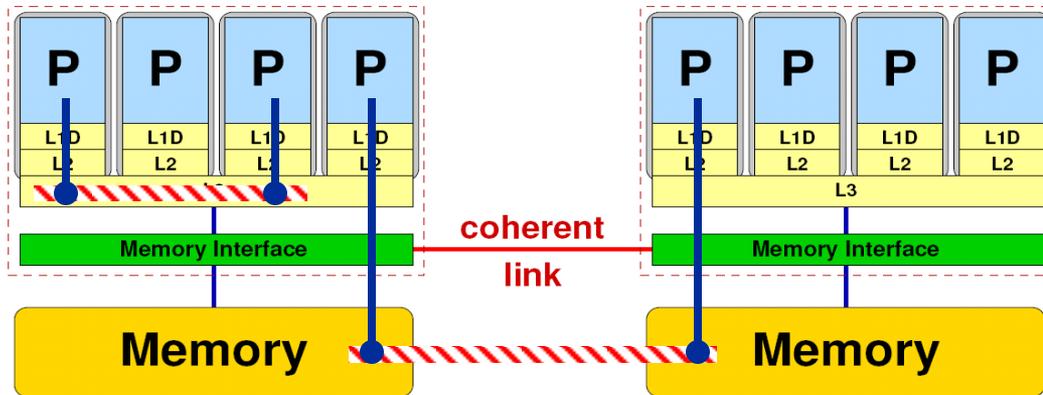


- Machine structure is invisible to user

- Very simple programming model
- Threading SW (OpenMP, pthreads, TBB,...) should know about the details

- Performance issues

- Synchronization overhead
- Memory access
- Node topology

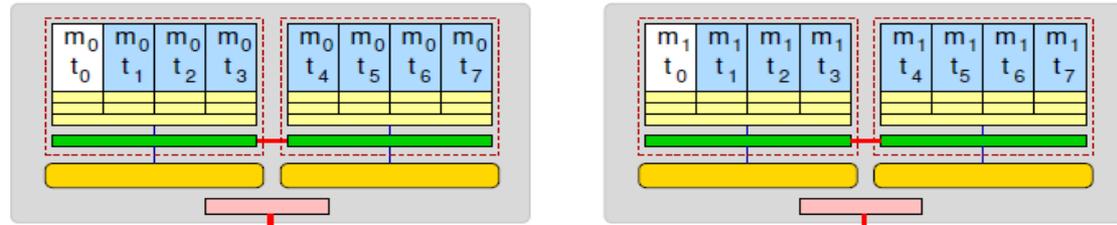


# Parallel programming models: Lots of choices

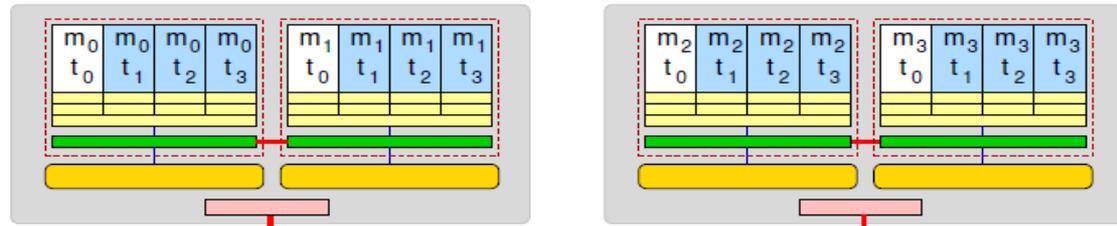
Hybrid MPI+OpenMP on a multicore multisocket cluster



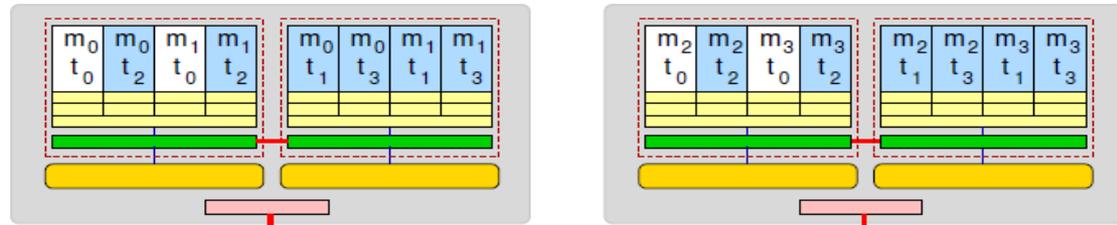
One MPI process / node



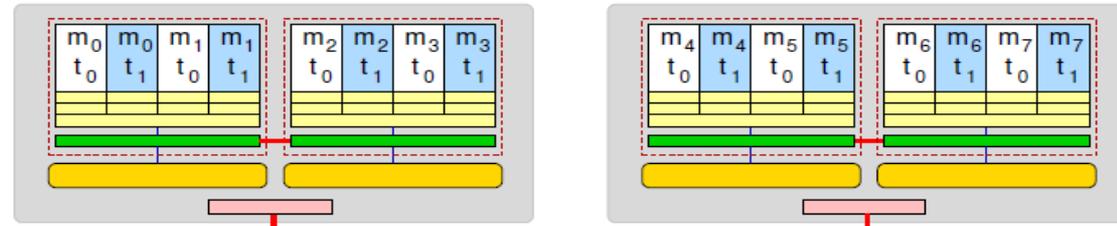
One MPI process / socket:  
OpenMP threads on same  
socket: “blockwise”



OpenMP threads pinned  
“round robin” across  
cores in node

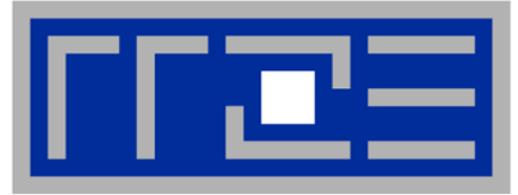


Two MPI processes / socket  
OpenMP threads  
on same socket





- **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**

### **Probing node topology**

- **Standard tools**
- **likwid-topology**

# How do we figure out the node topology?



- **Topology =**

- Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
- Which cores share which cache levels?
- Which hardware threads (“logical cores”) share a physical core?

- **Linux**

- `cat /proc/cpuinfo` is of limited use
- Core numbers may change across kernels and BIOSes even on identical hardware

- `numactl --hardware` prints ccNUMA node information



- Information on caches is harder to obtain

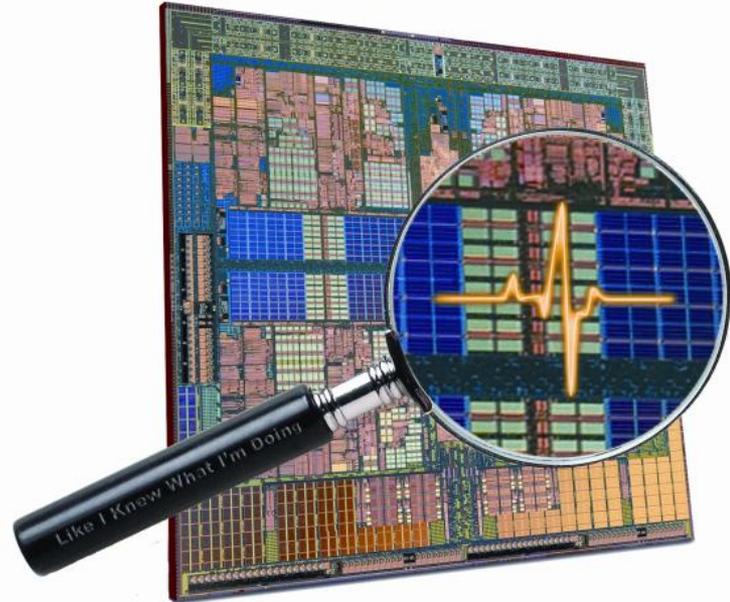
```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 8192 MB
node 3 free: 7840 MB
```

- **LIKWID** tool suite:

**L**ike  
**I**  
**K**new  
**W**hat  
**I**'m  
**D**oing

- Open source tool collection  
(developed at RRZE):

<http://code.google.com/p/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>

- **Command line tools for Linux:**

- easy to install
- works with standard linux 2.6 kernel
- simple and clear to use
- supports Intel and AMD CPU



- **Current tools:**

- **likwid-topology**: Print thread and cache topology
- **likwid-pin**: Pin threaded application without touching code
- **likwid-perfctr**: Measure performance counters
- **likwid-mpirun**: mpirun wrapper script for easy LIKWID integration
- **likwid-bench**: Low-level bandwidth benchmark generator tool
- ... some more

# Output of `likwid-topology -g`

on one node of Cray XE6 "Hermit"



```
-----  
CPU type:      AMD Interlagos processor
```

```
*****
```

## Hardware Thread Topology

```
*****
```

```
Sockets:      2  
Cores per socket: 16  
Threads per core: 1
```

```
-----  
HWThread      Thread      Core      Socket  
0              0           0         0  
1              0           1         0  
2              0           2         0  
3              0           3         0  
[...]  
16             0           0         1  
17             0           1         1  
18             0           2         1  
19             0           3         1  
[...]
```

```
-----  
Socket 0: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 )  
Socket 1: ( 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 )  
-----
```

```
*****  
Cache Topology
```

```
*****
```

```
Level:  1  
Size:   16 kB  
Cache groups: ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) ( 6 ) ( 7 ) ( 8 ) ( 9 ) ( 10 ) ( 11 ) ( 12 ) ( 13 )  
( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 ) ( 19 ) ( 20 ) ( 21 ) ( 22 ) ( 23 ) ( 24 ) ( 25 ) ( 26 ) ( 27 ) ( 28 )  
( 29 ) ( 30 ) ( 31 )
```

# Output of likwid-topology continued



```
-----  
Level: 2  
Size: 2 MB  
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 ) ( 16 17 ) ( 18  
19 ) ( 20 21 ) ( 22 23 ) ( 24 25 ) ( 26 27 ) ( 28 29 ) ( 30 31 )  
-----
```

```
Level: 3  
Size: 6 MB  
Cache groups: ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 ) ( 16 17 18 19 20 21 22 23 ) ( 24 25 26  
27 28 29 30 31 )  
-----
```

```
*****  
NUMA Topology  
*****  
NUMA domains: 4  
-----
```

```
Domain 0:  
Processors: 0 1 2 3 4 5 6 7  
Memory: 7837.25 MB free of total 8191.62 MB  
-----
```

```
Domain 1:  
Processors: 8 9 10 11 12 13 14 15  
Memory: 7860.02 MB free of total 8192 MB  
-----
```

```
Domain 2:  
Processors: 16 17 18 19 20 21 22 23  
Memory: 7847.39 MB free of total 8192 MB  
-----
```

```
Domain 3:  
Processors: 24 25 26 27 28 29 30 31  
Memory: 7785.02 MB free of total 8192 MB  
-----
```

# Output of likwid-topology continued



\*\*\*\*\*

Graphical:

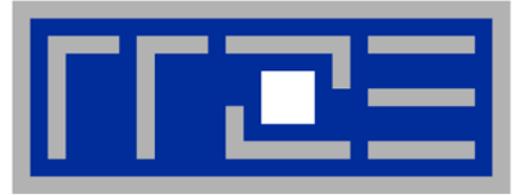
\*\*\*\*\*

Socket 0:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16kB															
2MB															
6MB								6MB							

Socket 1:

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
16kB															
2MB															
6MB								6MB							



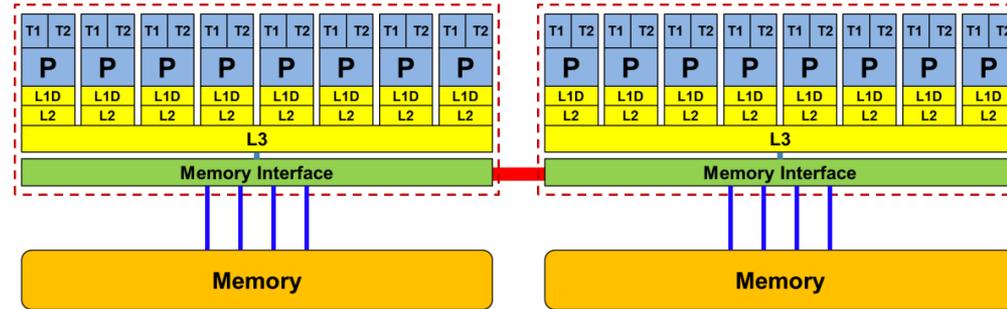
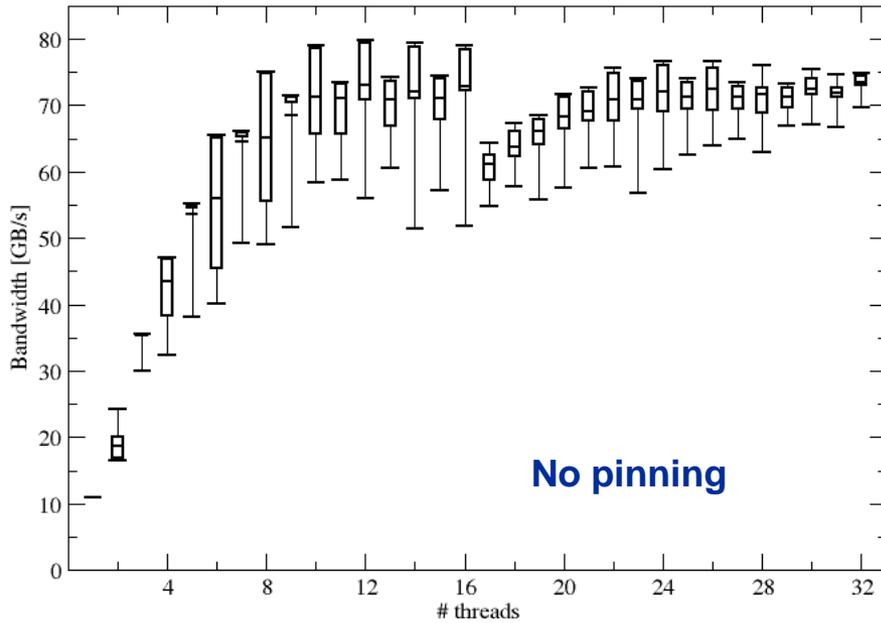
## **Enforcing thread/process-core affinity under the Linux OS**

- **Standard tools and OS affinity facilities  
under program control**
- **likwid-pin**

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

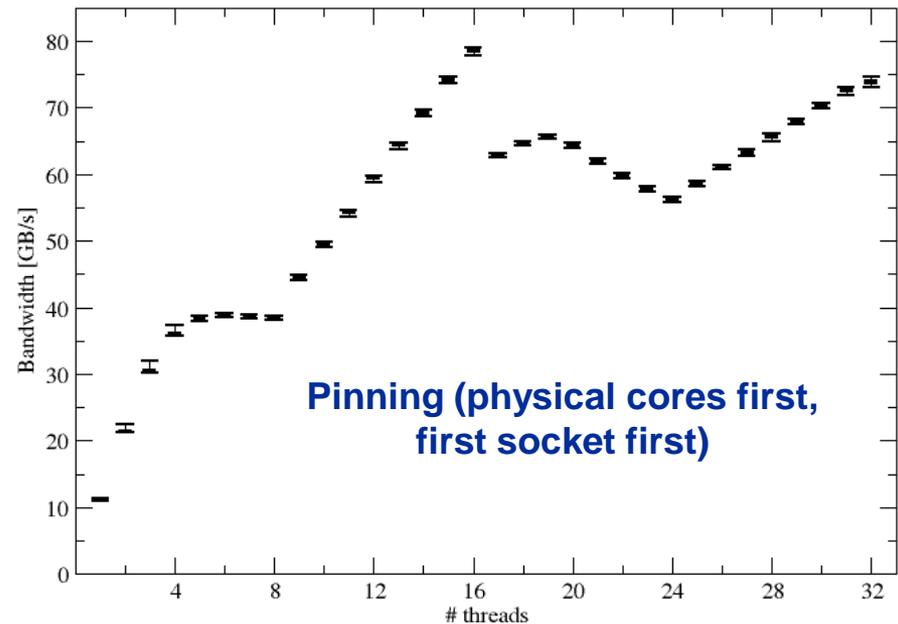


## *Anarchy vs. thread pinning*



There are several reasons for caring about affinity:

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





- **Highly OS-dependent system calls**
  - But available on all systems
  - Linux: `sched_setaffinity()`
  - Solaris: `processor_bind()`
  - Windows: `SetThreadAffinityMask()`
  - ...
- **Support for “semi-automatic” pinning in some compilers/environments**
  - All modern compilers with OpenMP support
  - PLPA → `hwloc`
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
  - OpenMP 4.0 (see OpenMP tutorial)
- **Affinity awareness in MPI libraries**
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - ...



- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → **binary must be dynamically linked!**
- Can also be used as a superior **replacement for taskset**
- Supports **logical core numbering** within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system
- **Usage examples:**
  - `likwid-pin -c 0,2,4-6 ./myApp parameters`
  - `likwid-pin -c S0:0-3 ./myApp parameters`



- Running the STREAM benchmark with likwid-pin:

```

$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK
-----
Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word
-----
[... some STREAM output omitted ...]
The *best* time for each test is used
*EXCLUDING* the first and last iterations
[pthread wrapper] PIN_MASK: 0->1 1->4 2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
    threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
    threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
    threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
    threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]

```

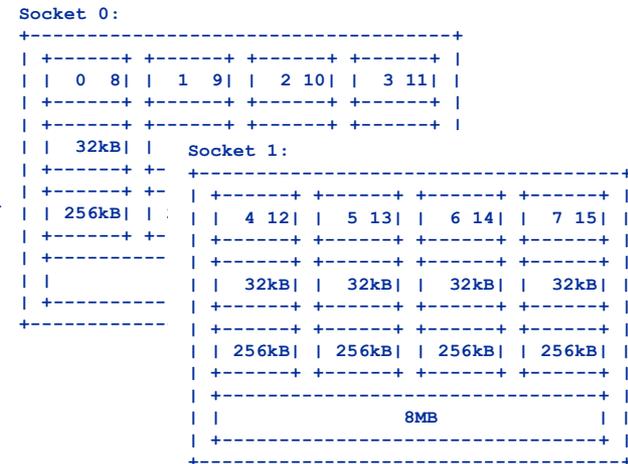
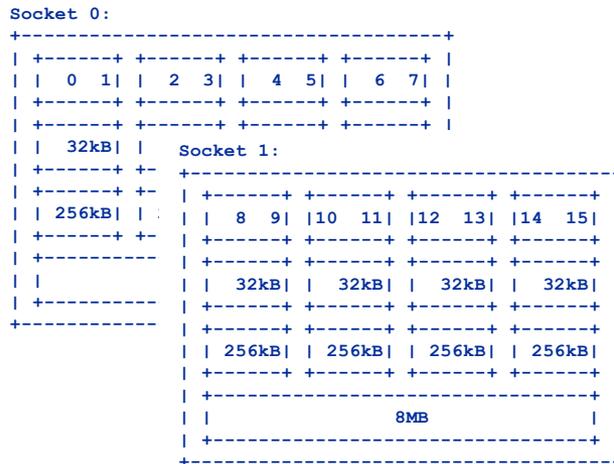
Main PID always  
pinned

Skip shepherd  
thread

Pin all spawned  
threads in turn



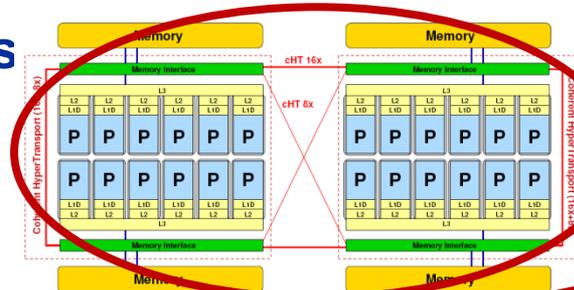
- Core numbering may vary from system to system even with identical hardware
  - Likwid-topology delivers this information, which can then be fed into likwid-pin
- Alternatively, likwid-pin can abstract this variation and provide a purely **logical** numbering (**physical cores first**)



- Across all cores in the node:  
`OMP_NUM_THREADS=8 likwid-pin -c N:0-7 ./a.out`
- Across the cores in each socket and across sockets in each node:  
`OMP_NUM_THREADS=8 likwid-pin -c S0:0-3@S1:0-3 ./a.out`

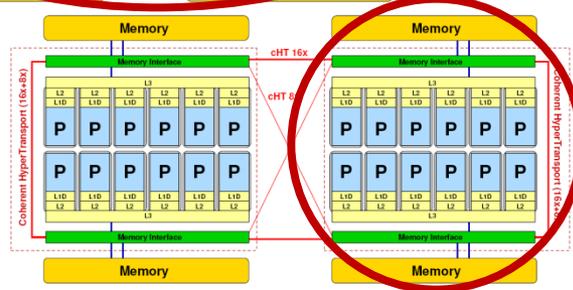
- Possible unit prefixes

**N** node

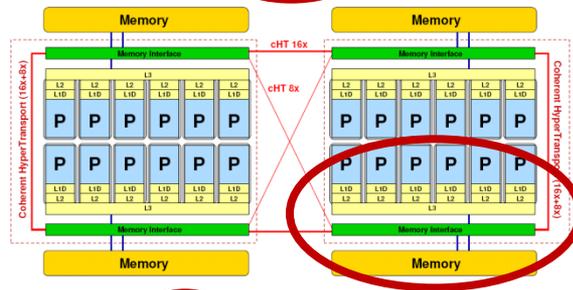


Default if -c is not specified!

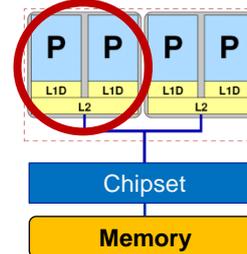
**S** socket



**M** NUMA domain



**C** outer level cache group





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

### Compact pinning:

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

Scattered pinning across all domains of the designated type :

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

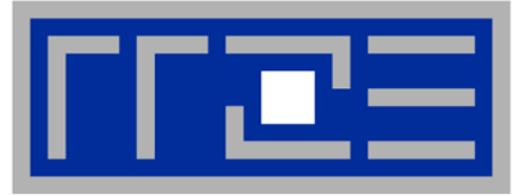
- Examples:

```
likwid-pin -c E:N:8 ... # equivalent to N:0-7
```

```
likwid-pin -c E:N:120:2:4 ... # Phi: 120 threads, 2 per core
```

- Scatter across all NUMA domains:

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



## **Multicore performance tools: Probing performance behavior**

**likwid-perfctr**



1. **Runtime profile / Call graph (gprof)**
2. **Instrument those parts which consume a significant part of runtime**
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)**



- 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



```
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe
```

```
-----
CPU type:      Intel Core Lynnfield processor
CPU clock:    2.93 GHz
-----
```

```
Measuring group FLOPS_DP
```

```
YOUR PROGRAM OUTPUT
```

Event	core 0	core 1	core 2	core 3
INSTR_RETIRED_ANY	1.97463e+08	2.31001e+08	2.30963e+08	2.31885e+08
CPU_CLK_UNHALTED_CORE	9.56999e+08	9.58401e+08	9.58637e+08	9.57338e+08
FP_COMP_OPS_EXE_SSE_FP_PACKED	4.00294e+07	3.08927e+07	3.08866e+07	3.08904e+07
FP_COMP_OPS_EXE_SSE_FP_SCALAR	882	0	0	0
FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION	0	0	0	0
FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION	4.00303e+07	3.08927e+07	3.08866e+07	3.08904e+07

Always  
measured

Configured metrics  
(this group)

Metric	core 0	core 1	core 2	core 3
Runtime [s]	0.326242	0.32672	0.326801	0.326358
CPI	4.84647	4.14891	4.15061	4.12849
DP MFlops/s (DP assumed)	245.399	189.108	189.024	189.304
Packed MUOPS/s	122.698	94.554	94.5121	94.6519
Scalar MUOPS/s	0.00270351	0	0	0
SP MUOPS/s	0	0	0	0
DP MUOPS/s	122.701	94.554	94.5121	94.6519

Derived  
metrics



### Things to look at (in roughly this order)

- **Load balance** (flops, instructions, BW)
- **In-socket memory BW saturation**
- **Shared cache 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 overrated**
  - If I really know my code, I can often *calculate* the misses
  - Runtime and resource utilization is much more important



- 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 regions 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



## Measuring energy consumption with LIKWID

# Measuring energy consumption

*likwid-powermeter and likwid-perfctr -g ENERGY*



- **Implements Intel RAPL interface (Sandy Bridge)**
- **RAPL = “Running average power limit”**

-----  
CPU name: Intel Core SandyBridge processor  
CPU clock: 3.49 GHz  
-----

Base clock: 3500.00 MHz  
Minimal clock: 1600.00 MHz

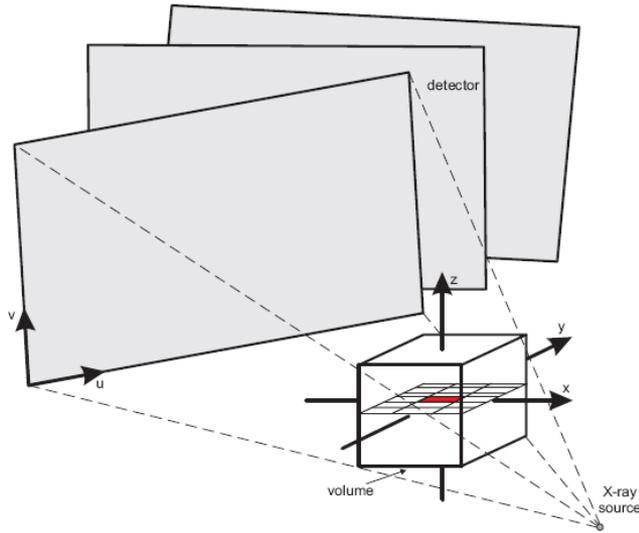
### Turbo Boost Steps:

C1 3900.00 MHz  
C2 3800.00 MHz  
C3 3700.00 MHz  
C4 3600.00 MHz  
-----

Thermal Spec Power: 95 Watts  
Minimum Power: 20 Watts  
Maximum Power: 95 Watts  
Maximum Time Window: 0.15625 micro sec  
-----

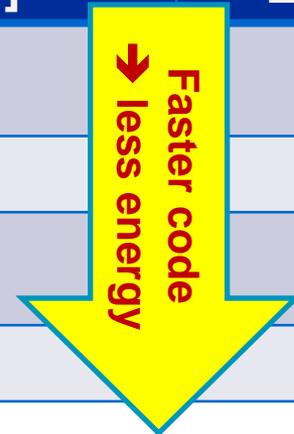
# Example:

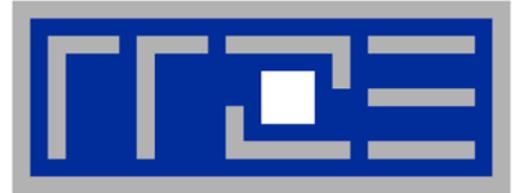
A medical image reconstruction code on Sandy Bridge



## Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

Test case	Runtime [s]	Power [W]	Energy [J]
8 cores, plain C	<b>90.43</b>	90	8110
8 cores, SSE	29.63	93	2750
8 cores (SMT), SSE	22.61	102	2300
8 cores (SMT), AVX	<b>18.42</b>	111	2040





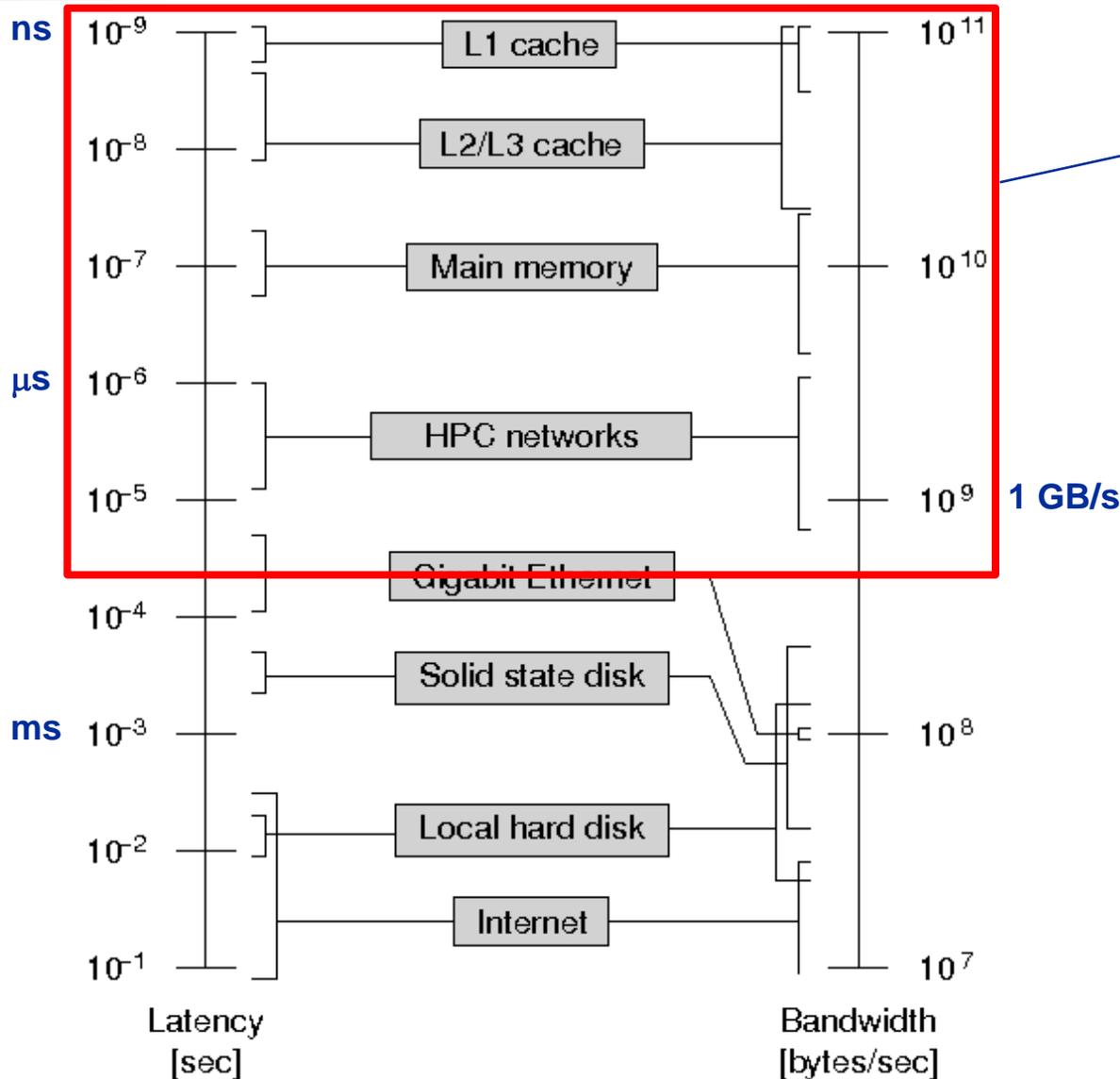
# **Microbenchmarking for architectural exploration**

**Probing of the memory hierarchy**

**Saturation effects in cache and memory**

**Typical OpenMP overheads**

# Latency and bandwidth in modern computer environments



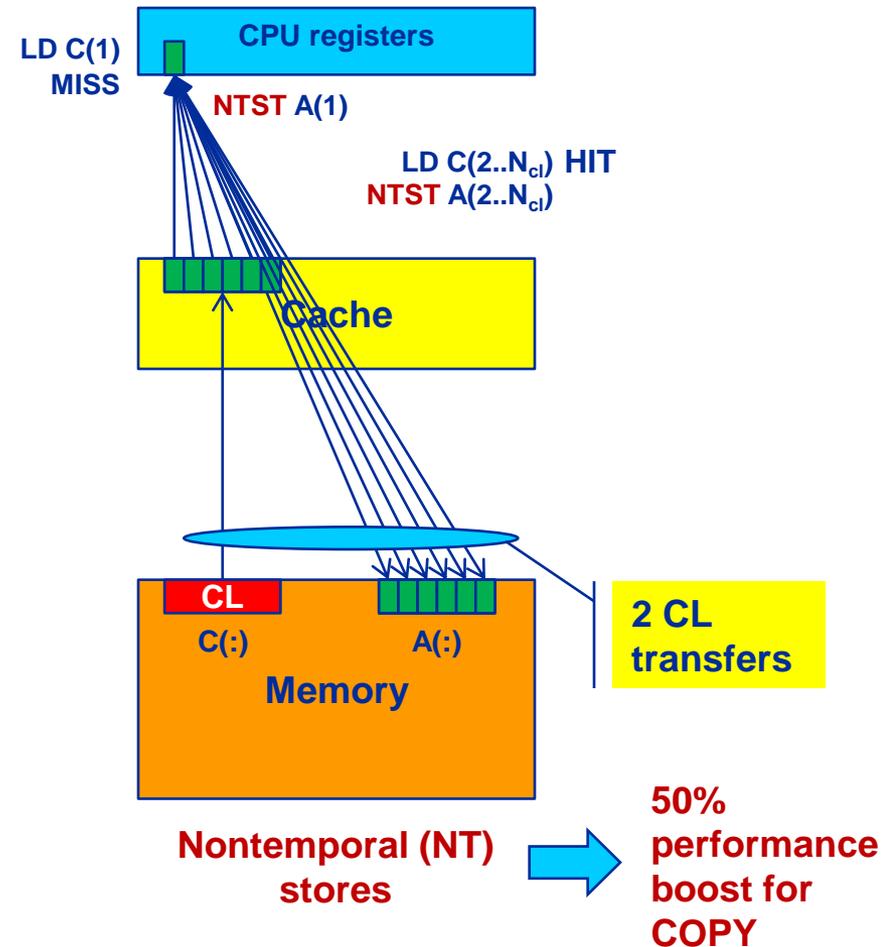
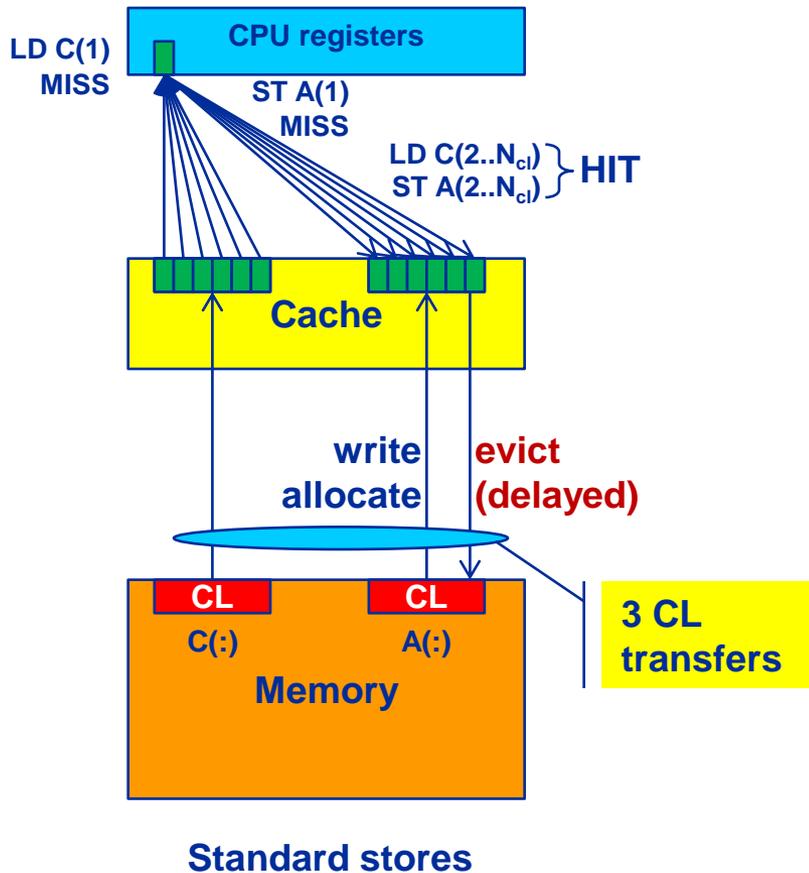
HPC plays here

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

# Recap: Data transfers in a memory hierarchy



- How does data travel from memory to the CPU and back?
- Example: Array copy  $A(:) = C(:)$





## Simple streaming benchmark:

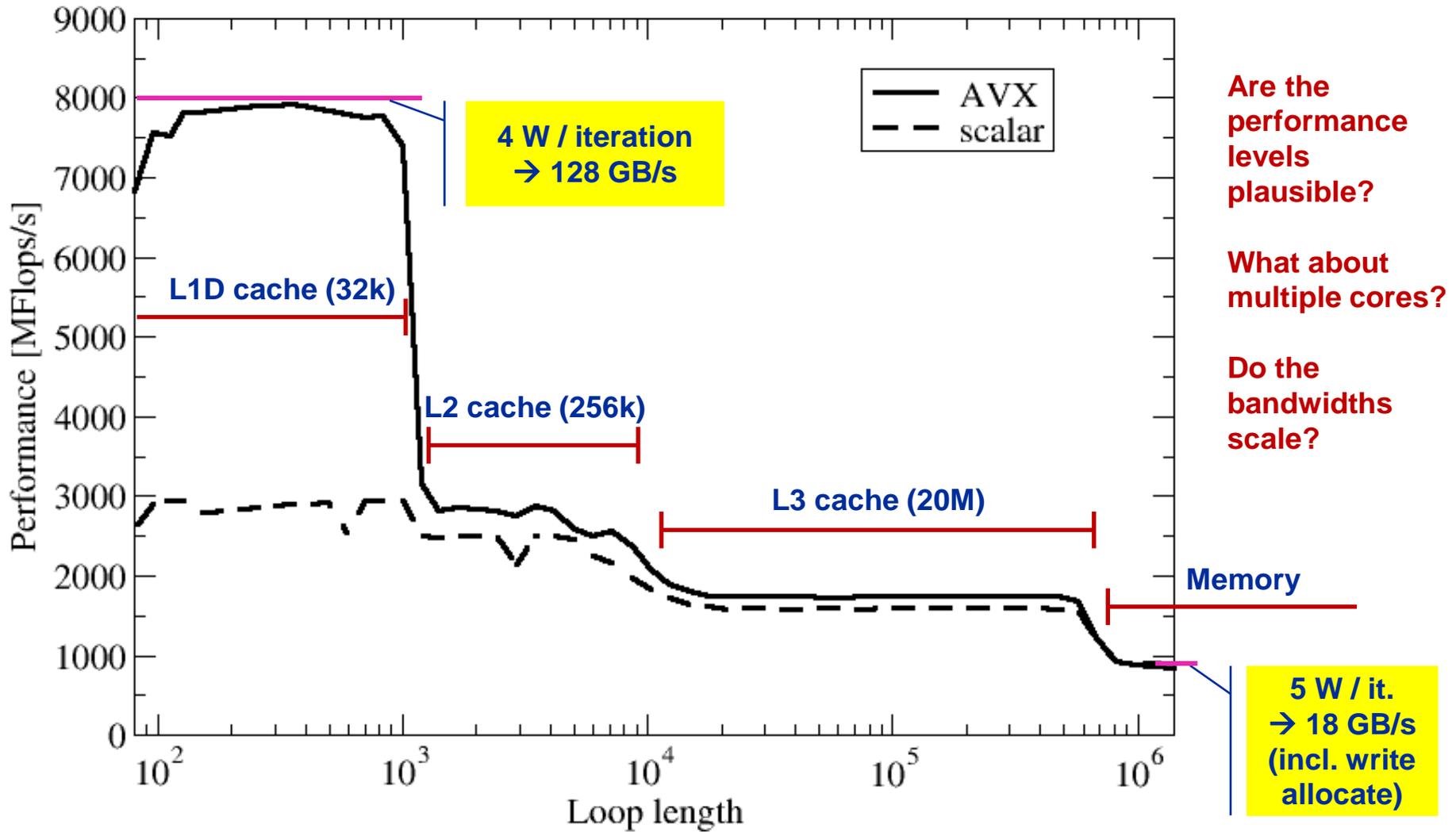
```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A
```

```
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)



Are the performance levels plausible?

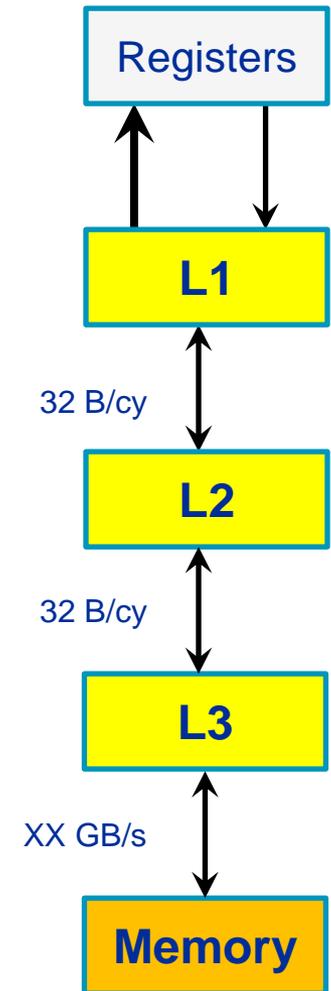
What about multiple cores?

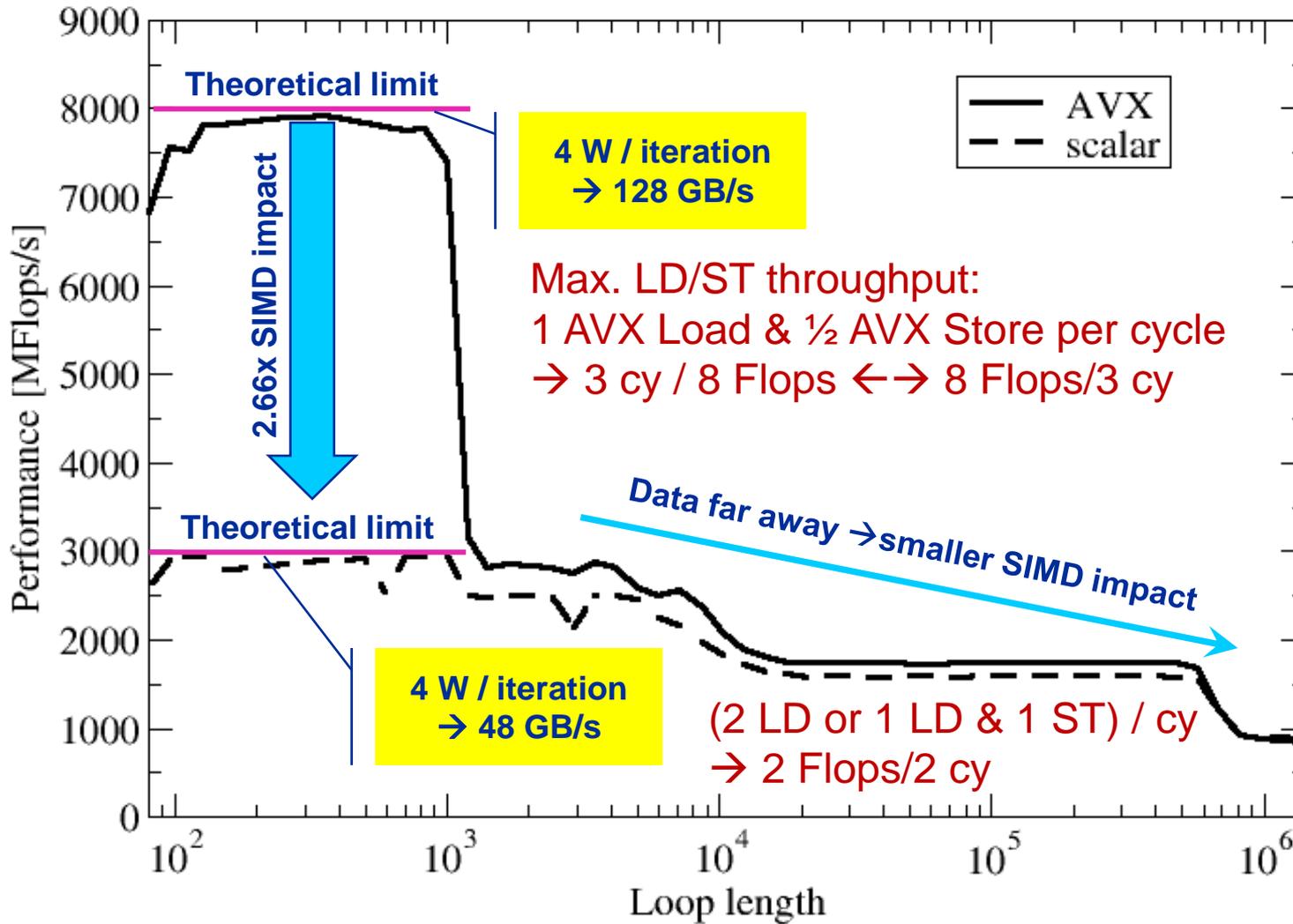
Do the bandwidths scale?

# Throughput capabilities of the Intel Sandy Bridge



- **Per cycle with AVX**
  - 1 load instruction (256 bits) **AND** ½ store instruction (128 bits)
  - 1 AVX MULT and 1 AVX ADD instruction (4 DP / 8 SP flops each)
  - Overall maximum of 4 micro-ops
- **Per cycle with SSE or scalar**
  - 2 load instruction **OR** 1 load and 1 store instruction
  - 1 MULT and 1 ADD instruction
  - Overall maximum of 4 micro-ops
- **Data transfer between cache levels (L3 ↔ L2, L2 ↔ L1)**
  - 256 bits per cycle, half-duplex (i.e., full CL transfer == 2 cy)





See later for more on SIMD benefits



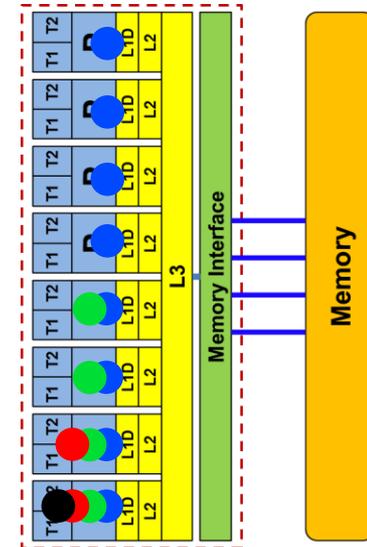
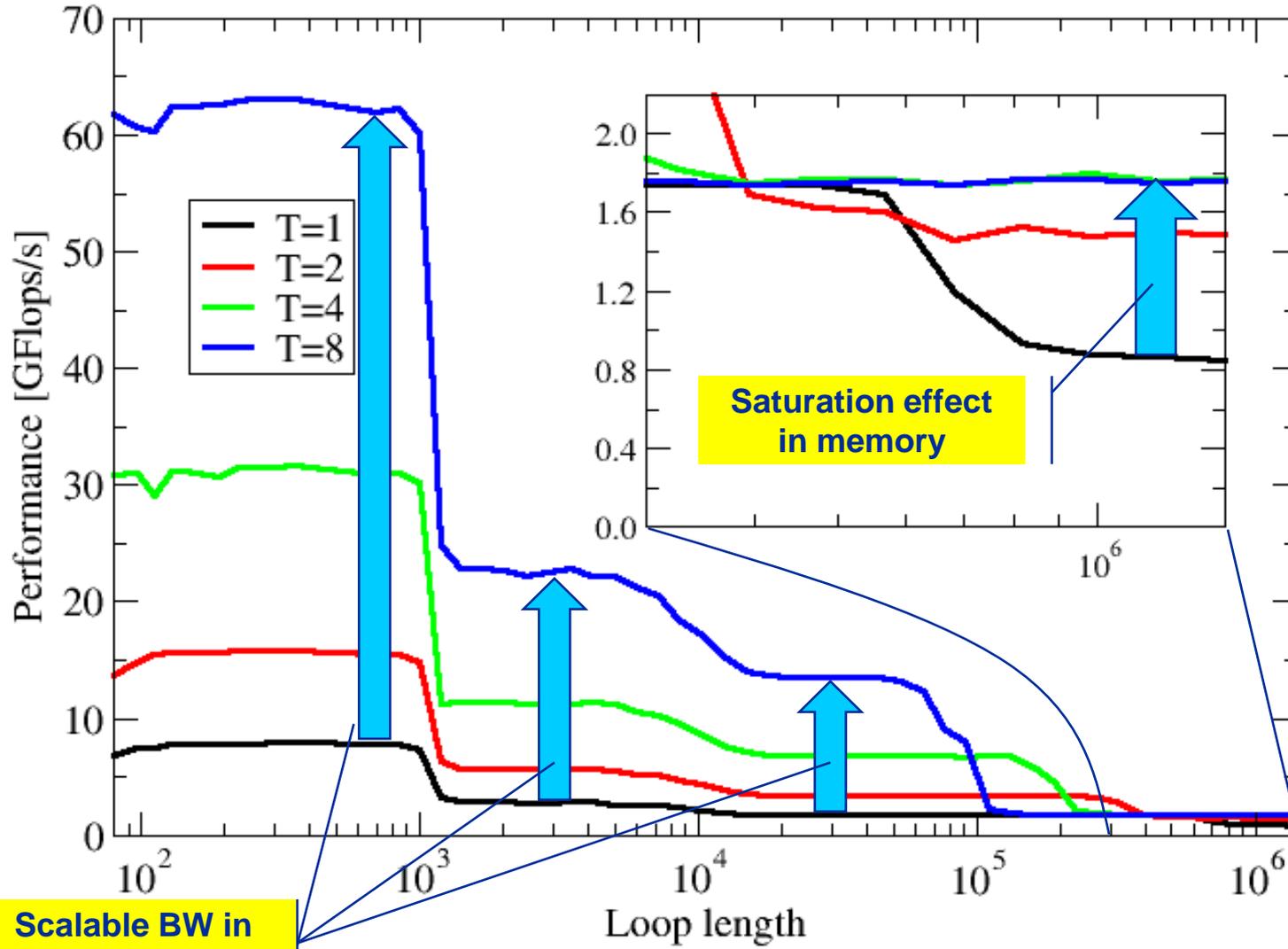
Every core runs its own, independent triad benchmark

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

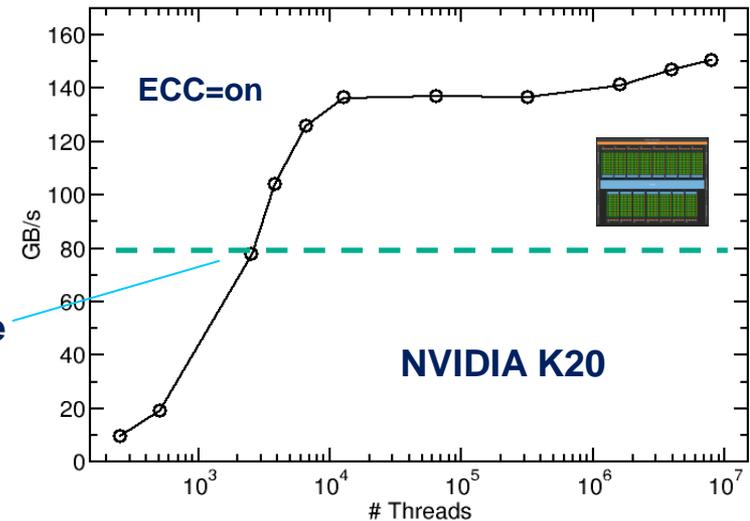
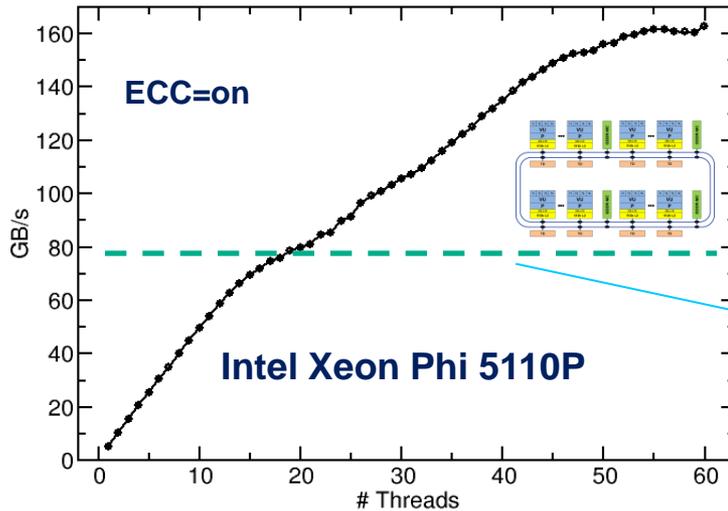
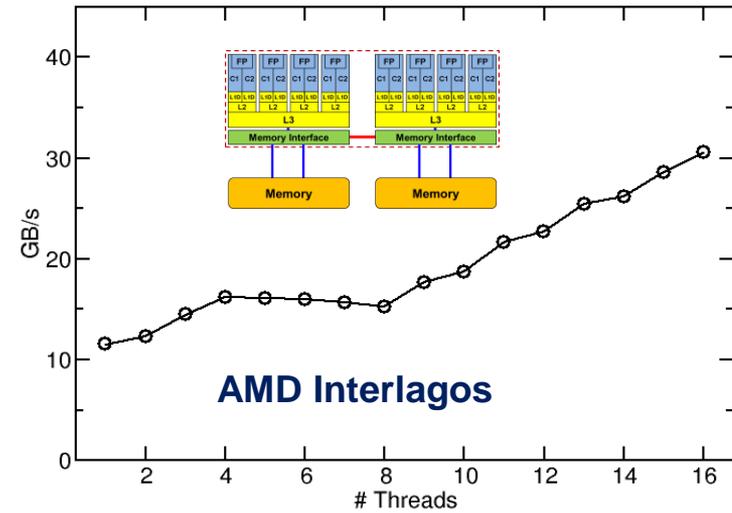
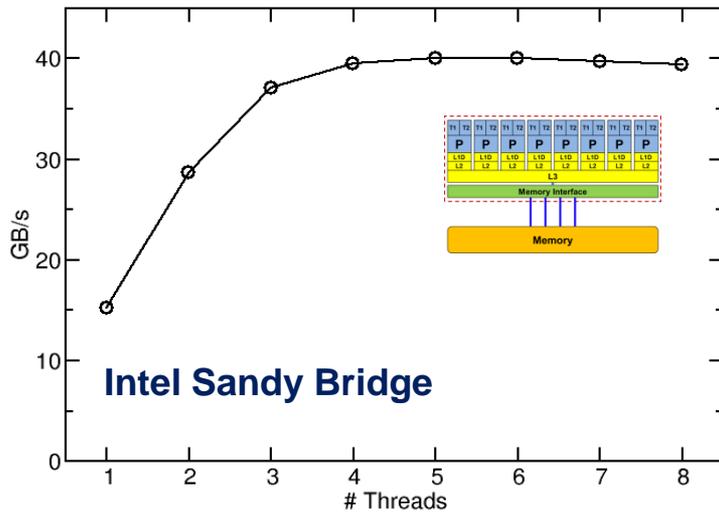
!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL
```

→ pure hardware probing, no impact from OpenMP overhead

# Throughput vector triad on Sandy Bridge socket (3 GHz)

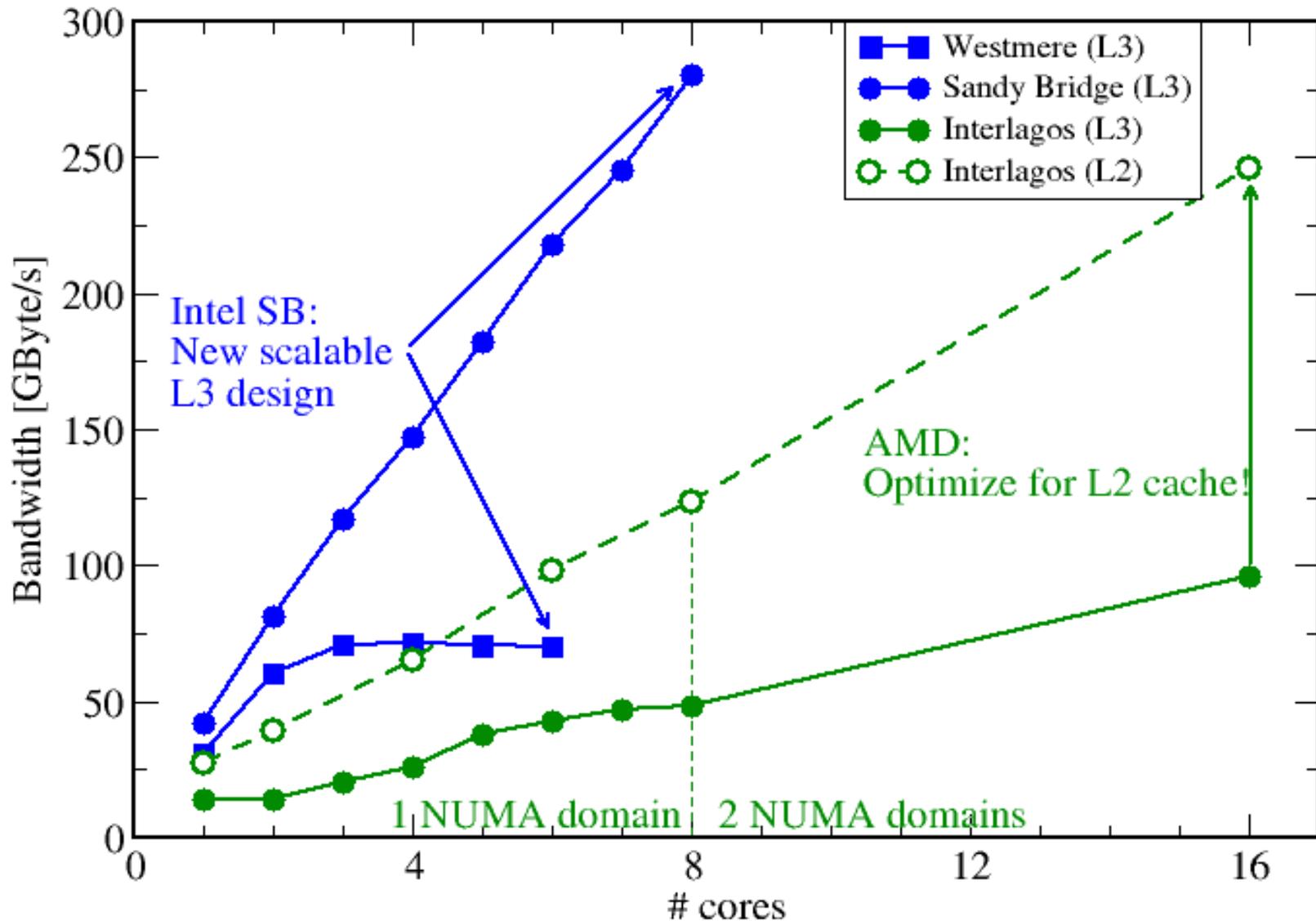


# Attainable memory bandwidth: Comparing architectures



# Bandwidth limitations: Outer-level cache

## Scalability of shared data paths in L3 cache





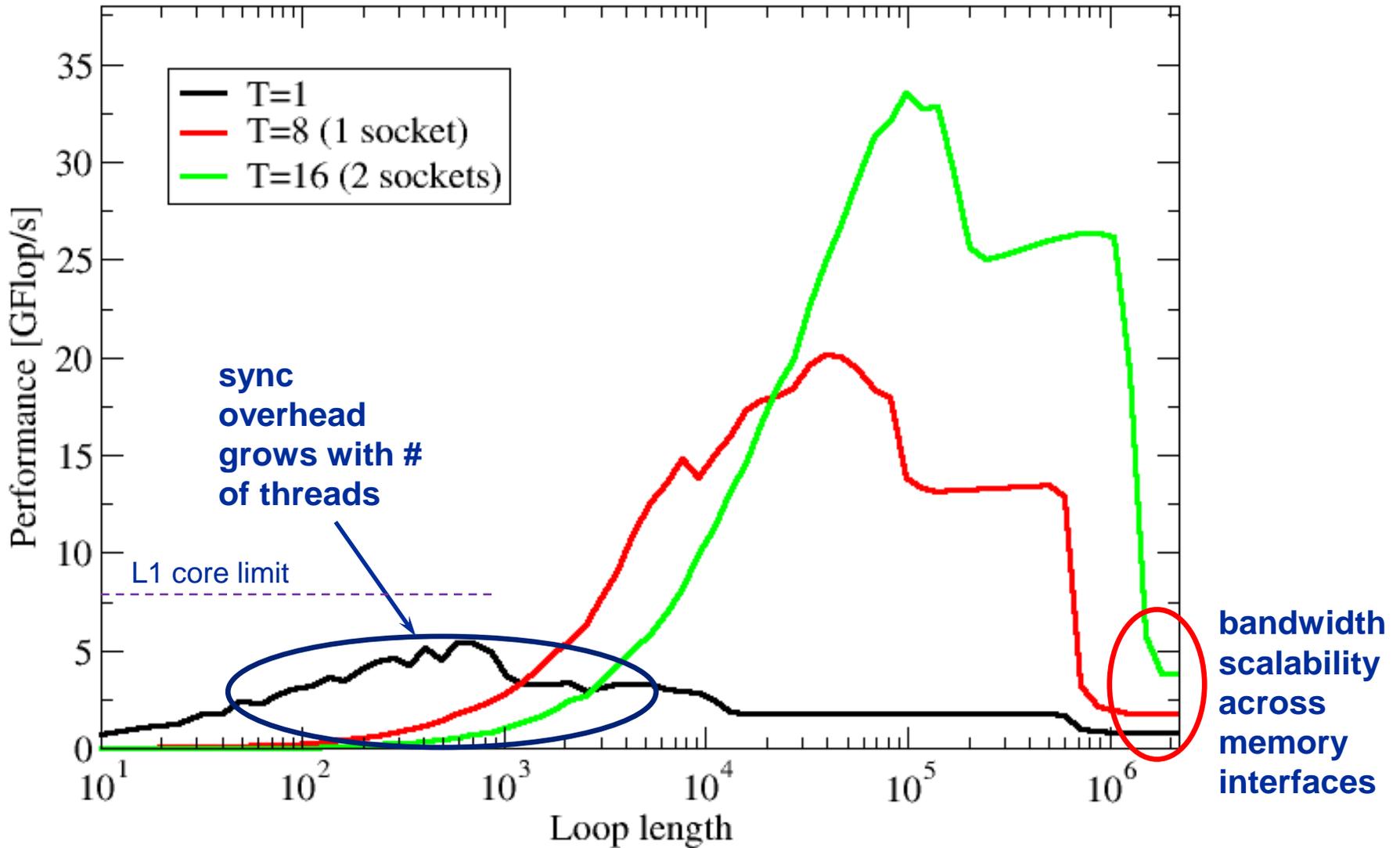
## OpenMP work sharing in the benchmark loop

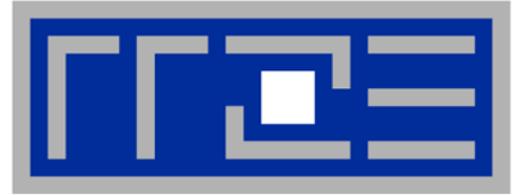
```
double precision, dimension(:), allocatable :: A,B,C,D

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 socket (3 GHz)





## **OpenMP performance issues on multicore**

**Synchronization (barrier) overhead**

# 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 programs.

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 SandyBridge-EP

Barrier overhead in CPU cycles



2 Threads	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Shared L3	384	5242	4616
SMT threads	2509	3726	3399
Other socket	1375	5959	4909

 Gcc still not very competitive

Intel compiler 

Full domain	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Socket	1497	14546	14418
Node	3401	34667	29788
Node +SMT	6881	59038	58898

# 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	<b>1597</b>	2825	3557
Full chip	10604	12800	15573	<b>18490</b>

That does not look bad for 240 threads!

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

**3.75 x cores** (16 vs 60) on Phi

**2 x more operations per cycle** on Phi

**2.7 x more barrier penalty (cycles)** on Phi

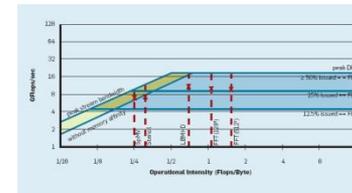
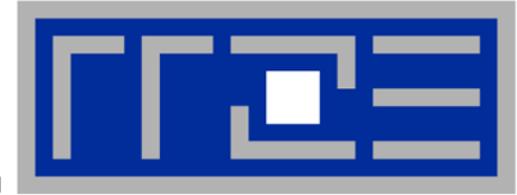


**7.5 x more work done** on Xeon Phi per cycle

One barrier causes  $2.7 \times 7.5 = 20x$  more pain 😊.



- **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**
  
- **Synchronization overhead may be an issue**
  - ... and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization



## “Simple” performance modeling: The Roofline Model<sup>(1)</sup>

Loop-based performance modeling: Execution vs. data transfer  
Example: array summation  
Example: A 3D Jacobi solver  
Model-guided optimization

<sup>(1)</sup> Samuel Williams, Andrew Waterman, David Patterson, Communications of the ACM, Vol. 52 No. 4, Pages 65-76 10.1145/1498765.1498785  
<http://cacm.acm.org/magazines/2009/4/22959-roofline-an-insightful-visual-performance-model-for-multicore-architectures/fulltext>



1.  $P_{\max}$  = **Applicable peak performance** of a loop, assuming that data comes from L1 cache (this is not necessarily  $P_{\text{peak}}$ )
2.  $I$  = **Computational intensity** (“work” per byte transferred) over the slowest data path utilized (“the bottleneck”)
  - Code balance  $B_C = I^{-1}$
3.  $b_S$  = **Applicable peak bandwidth** of the slowest data path utilized

Expected performance:

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

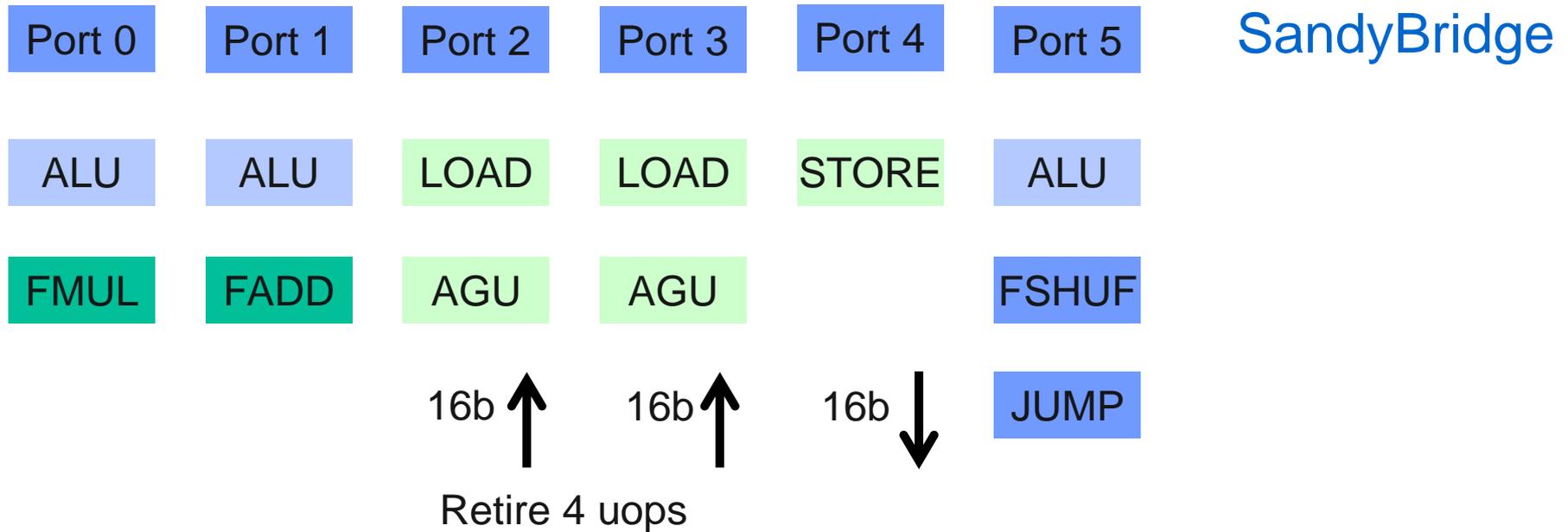
The diagram shows the equation  $P = \min(P_{\max}, I \cdot b_S)$ . Two yellow boxes with blue borders are positioned above the equation. The left box contains the text "[F/B]" and has a line pointing to the variable  $I$  in the equation. The right box contains the text "[B/s]" and has a line pointing to the variable  $b_S$  in the equation.

<sup>1</sup> W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. (2000)

<sup>2</sup> S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)



How to perform a instruction throughput analysis on the example of Intel's port based scheduler model

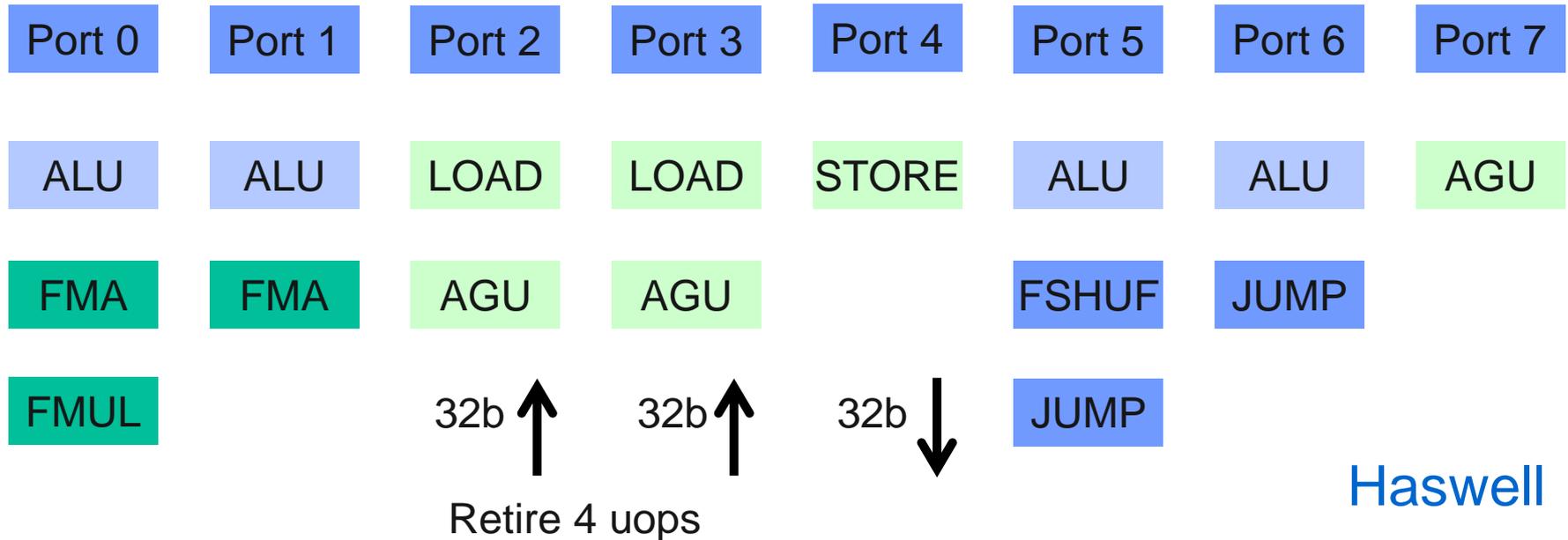


**First-order assumption:** All instructions in a loop are fed independently to the various ports/pipelines

**Complex cases** (dependencies, hazards): Add penalty cycles / use tools (Intel IACA, Intel Amplifier)



Every new CPU generation provides incremental improvements.





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

How many cycles to process **one AVX-vectorized iteration** (one core)?

→ Equivalent to 4 scalar iterations

Cycle 1: LOAD +  $\frac{1}{2}$  STORE + MULT + ADD

Cycle 2: LOAD +  $\frac{1}{2}$  STORE

Cycle 3: LOAD

**Answer: 3 cycles**



```
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** and the bandwidth in **MBytes/s**?

One AVX iteration (3 cycles) performs  $4 \times 2 = 8$  flops.

$(2.7 \text{ GHz} / 3 \text{ cycles}) * 4 \text{ updates} * 2 \text{ flops/update} = \mathbf{7.2 \text{ GFlops/s}}$

$4 \text{ GUPS/s} * 4 \text{ words/update} * 8 \text{ byte/word} = \mathbf{128 \text{ GBytes/s}}$



**Example: Vector triad  $A(:) = B(:) + C(:) * D(:)$   
on a 2.7 GHz 8-core Sandy Bridge chip (AVX vectorized)**

- $b_S = 40 \text{ GB/s}$
- $B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F}$  (including write allocate)  
→  $I = 0.4 \text{ F/W} = 0.05 \text{ F/B}$

→  $I \cdot b_S = 2.0 \text{ GF/s}$  (1.2 % of peak performance)

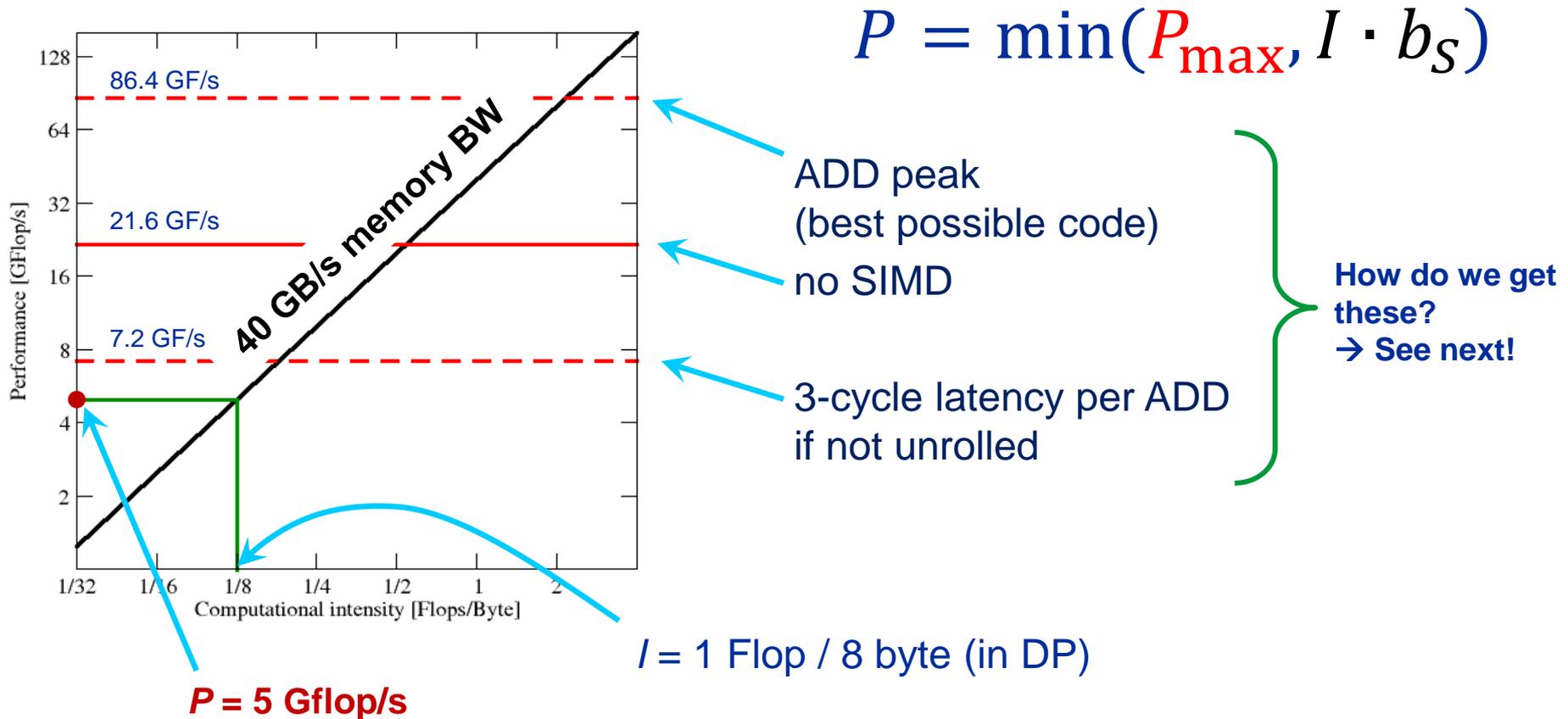
- $P_{\text{peak}} = 173 \text{ Gflop/s}$  (8 FP units x (4+4) Flops/cy x 2.7 GHz)
- $P_{\text{max}} = 8 \times 7.2 \text{ Gflop/s} = 57.6 \text{ Gflop/s}$  (33% peak)

$$P = \min(P_{\text{max}}, I \cdot b_S) = \min(57.6, 2.0) \text{ GFlop/s} \\ = 2.0 \text{ GFlop/s}$$

# A not so simple Roofline example

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

in double precision on a 2.7 GHz Sandy Bridge socket @ “large” N

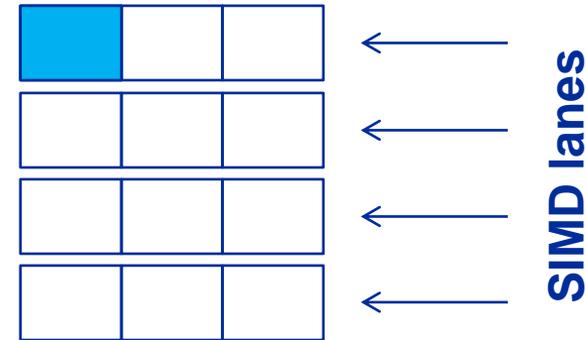




## Plain scalar code, no SIMD

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

ADD pipes utilization:



→ 1/12 of ADD peak



## Scalar code, 3-way unrolling

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

loop:

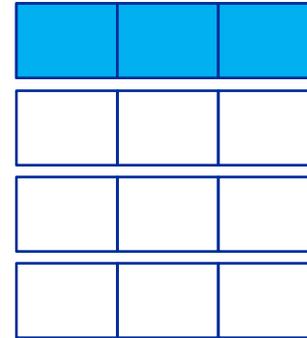
```
LOAD r4.0 ← a(i)  
LOAD r5.0 ← a(i+1)  
LOAD r6.0 ← a(i+2)
```

```
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:



→ 1/4 of ADD peak

# Applicable peak for the summation loop



## SIMD-vectorized, 3-way unrolled

```
LOAD [r1.0, ..., r1.3] ← [0, 0]
LOAD [r2.0, ..., r2.3] ← [0, 0]
LOAD [r3.0, ..., r3.3] ← [0, 0]
i ← 1
```

loop:

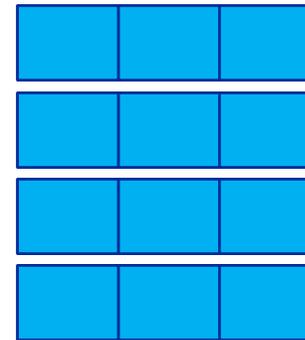
```
LOAD [r4.0, ..., r4.3] ← [a(i), ..., a(i+3)]
LOAD [r5.0, ..., r5.3] ← [a(i+4), ..., a(i+7)]
LOAD [r6.0, ..., r6.3] ← [a(i+8), ..., a(i+11)]
```

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

```
i+=12 →? loop
```

```
result ← r1.0+r1.1+...+r3.2+r3.3
```

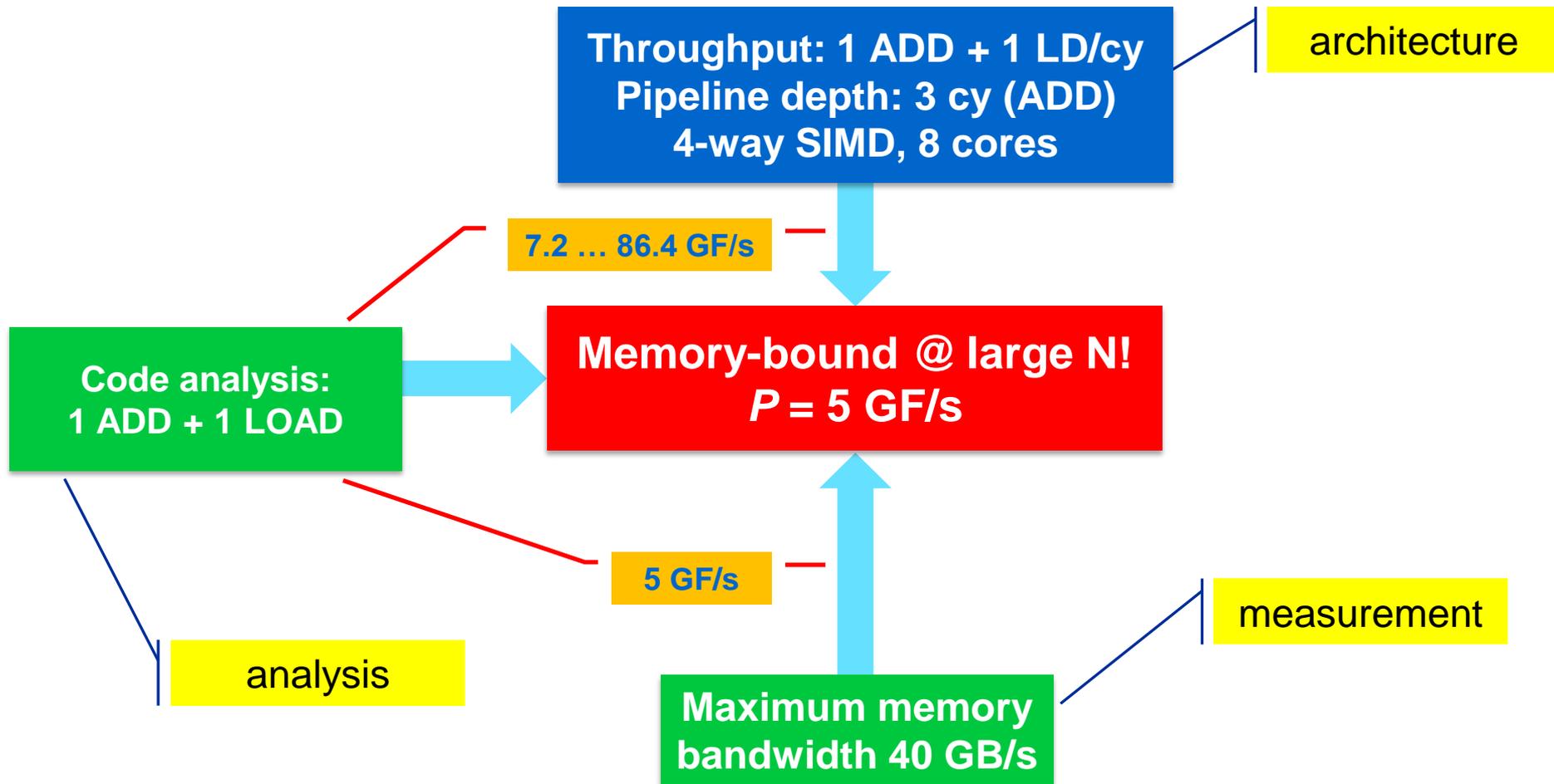
ADD pipes utilization:



→ ADD peak



... on the example of `do i=1,N; s=s+a(i); enddo`





- **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!**
  - **Slowest data path is modeled only;** all others are assumed to be infinitely fast
  - If data transfer is the limiting factor, the **bandwidth** of the slowest data path can be **utilized to 100%** (“saturation”)
  - Latency effects are ignored, i.e. **perfect streaming mode**



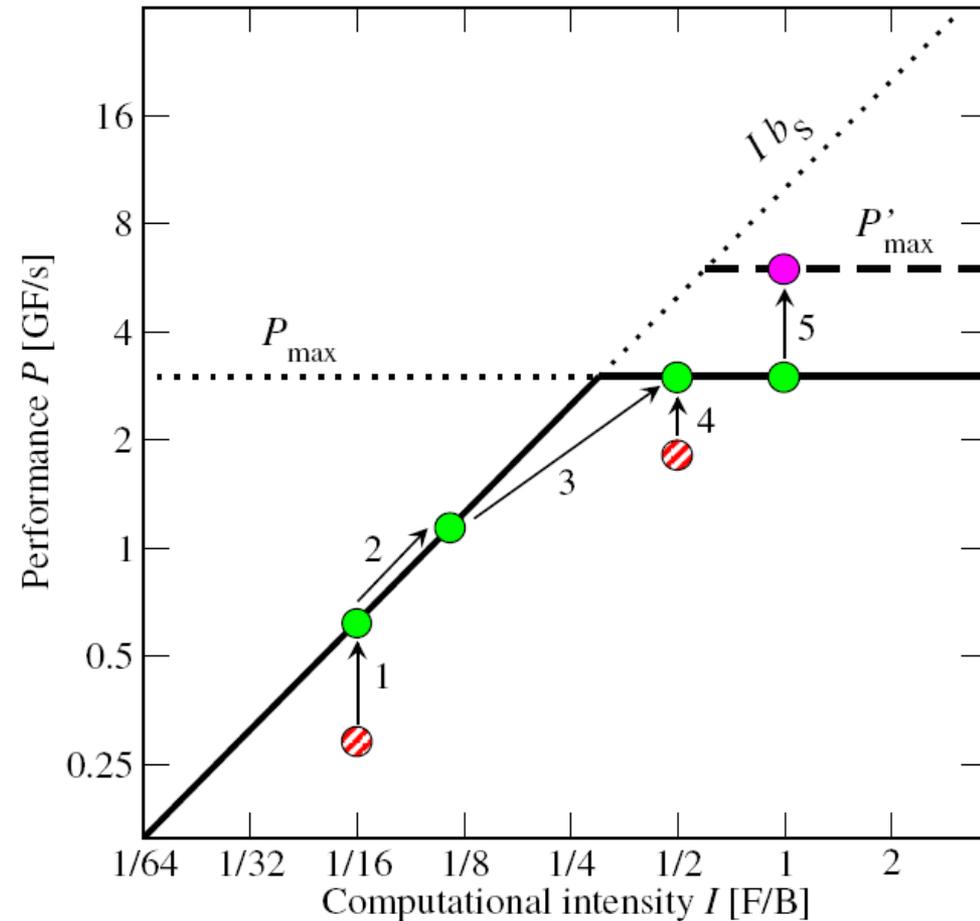
```
do i=1,N
  do j=1,N
    c(i)=c(i)+A(j,i)*b(j)
  enddo
enddo
```

➔

```
do i=1,N
  tmp = c(i)
  do j=1,N
    tmp = tmp + A(j,i)*b(j)
  enddo
  c(i) = tmp
enddo
```

- Assume  $N \approx 5000$
- Applicable peak performance?
- Relevant data path?
- Computational Intensity?

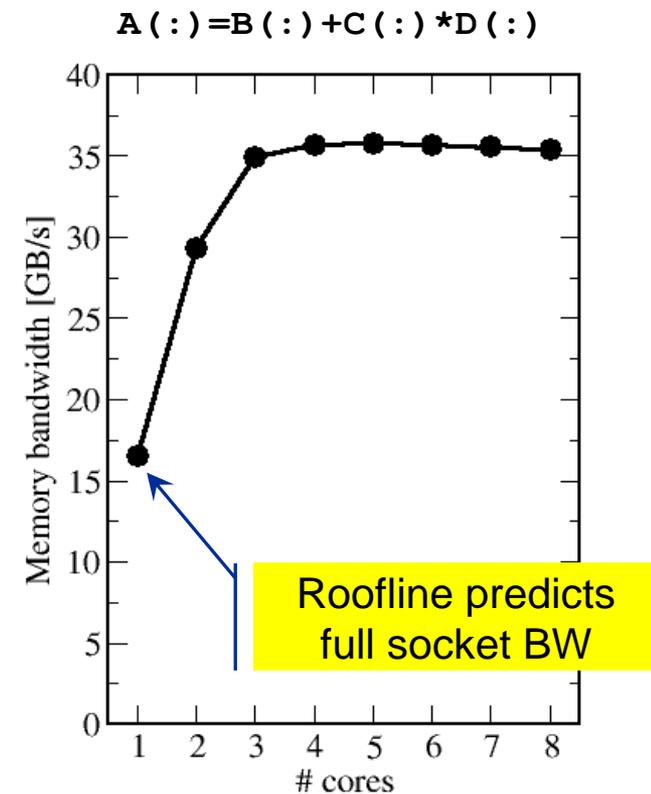
1. Hit the BW bottleneck by good serial code
2. Increase intensity to make better use of BW bottleneck
3. Increase intensity and go from memory-bound to core-bound
4. Hit the core bottleneck by good serial code
5. Shift  $P_{\max}$  by accessing additional hardware features or using a different algorithm/implementation

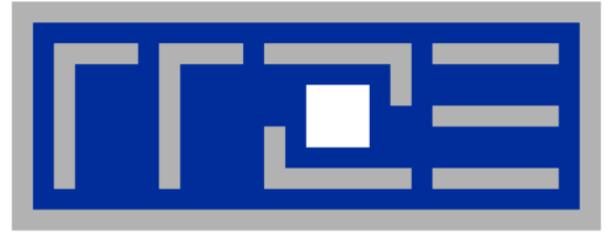


- **Saturation effects** in multicore chips are not explained
  - Reason: “saturation assumption”
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - Only increased “pressure” on the memory interface can saturate the bus  
→ need more cores!

- **ECM model gives more insight:**

G. Hager, J. Treibig, J. Habich, and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Concurrency and Computation: Practice and Experience (2013). DOI: [10.1002/cpe.3180](https://doi.org/10.1002/cpe.3180)  
Preprint: [arXiv:1208.2908](https://arxiv.org/abs/1208.2908)

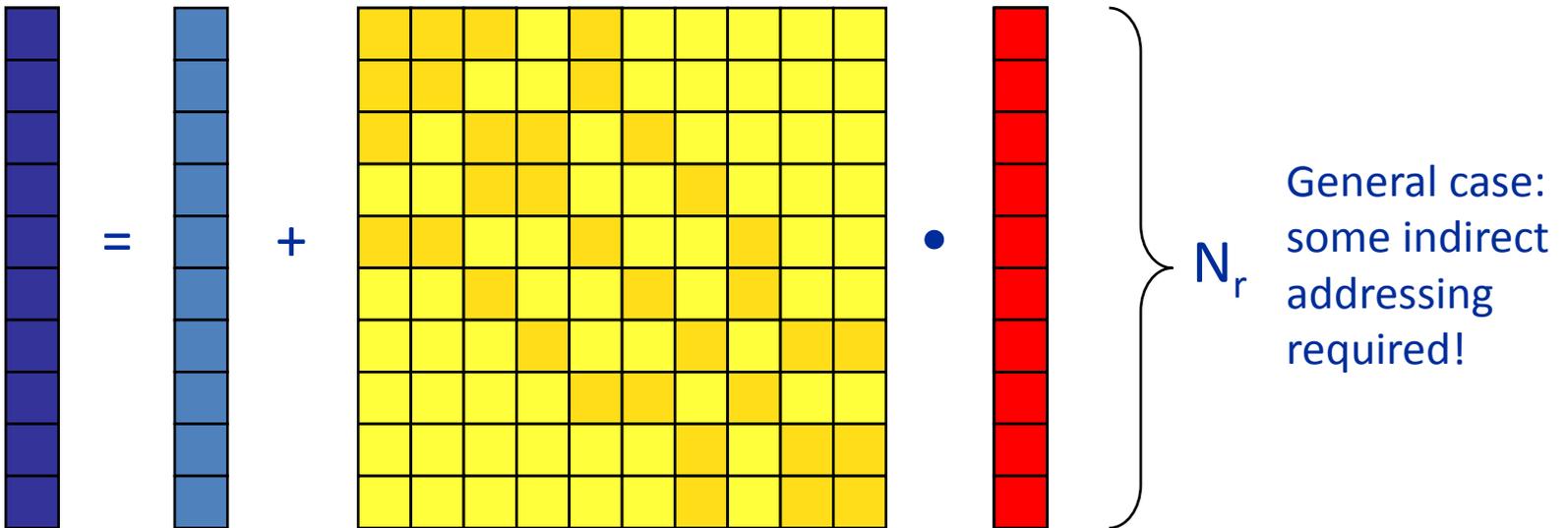




**Case study:**  
**Sparse Matrix Vector Multiplication**

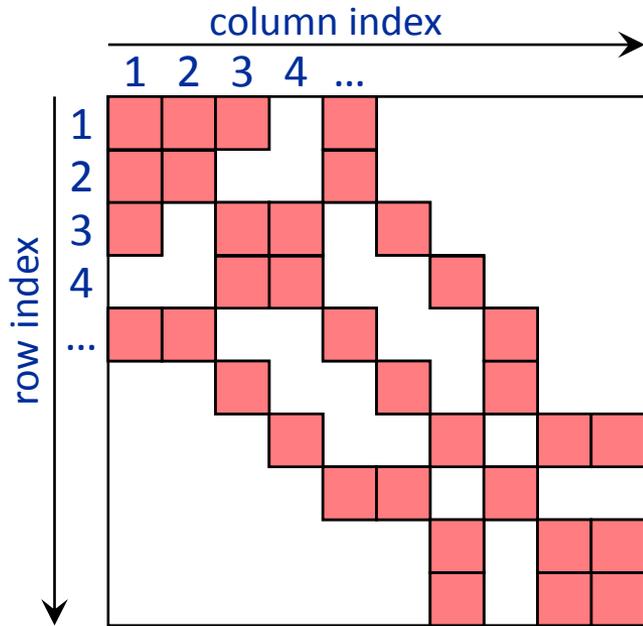


- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- Store only  $N_{nz}$  nonzero elements of matrix and RHS, LHS vectors with  $N_r$  (number of matrix rows) entries
- “Sparse”:  $N_{nz} \sim N_r$

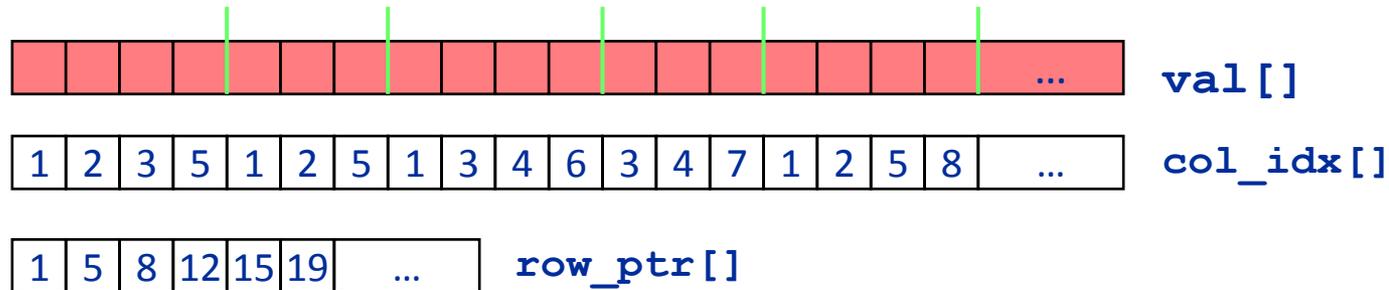




- For large problems, spMVM is inevitably **memory-bound**
  - **Intra-socket saturation effect** on modern multicores
- SpMVM is **easily parallelizable** in shared and distributed memory
- Data storage format is **crucial** for performance properties
  - Most useful general format on CPUs:  
Compressed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



- **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:  $N_r$ )





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

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

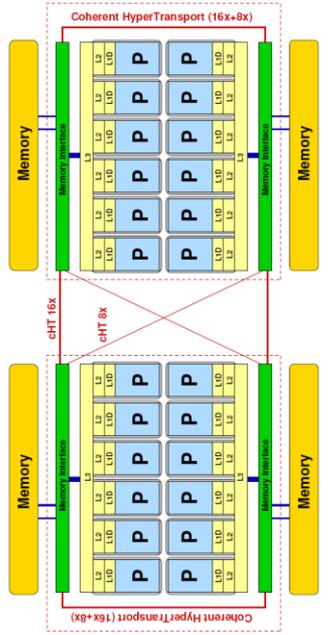
- Usually many spMVMs required to solve a problem
- **Following slides: Performance data on one 24-core AMD Magny Cours node**

# Application: Sparse matrix-vector multiply

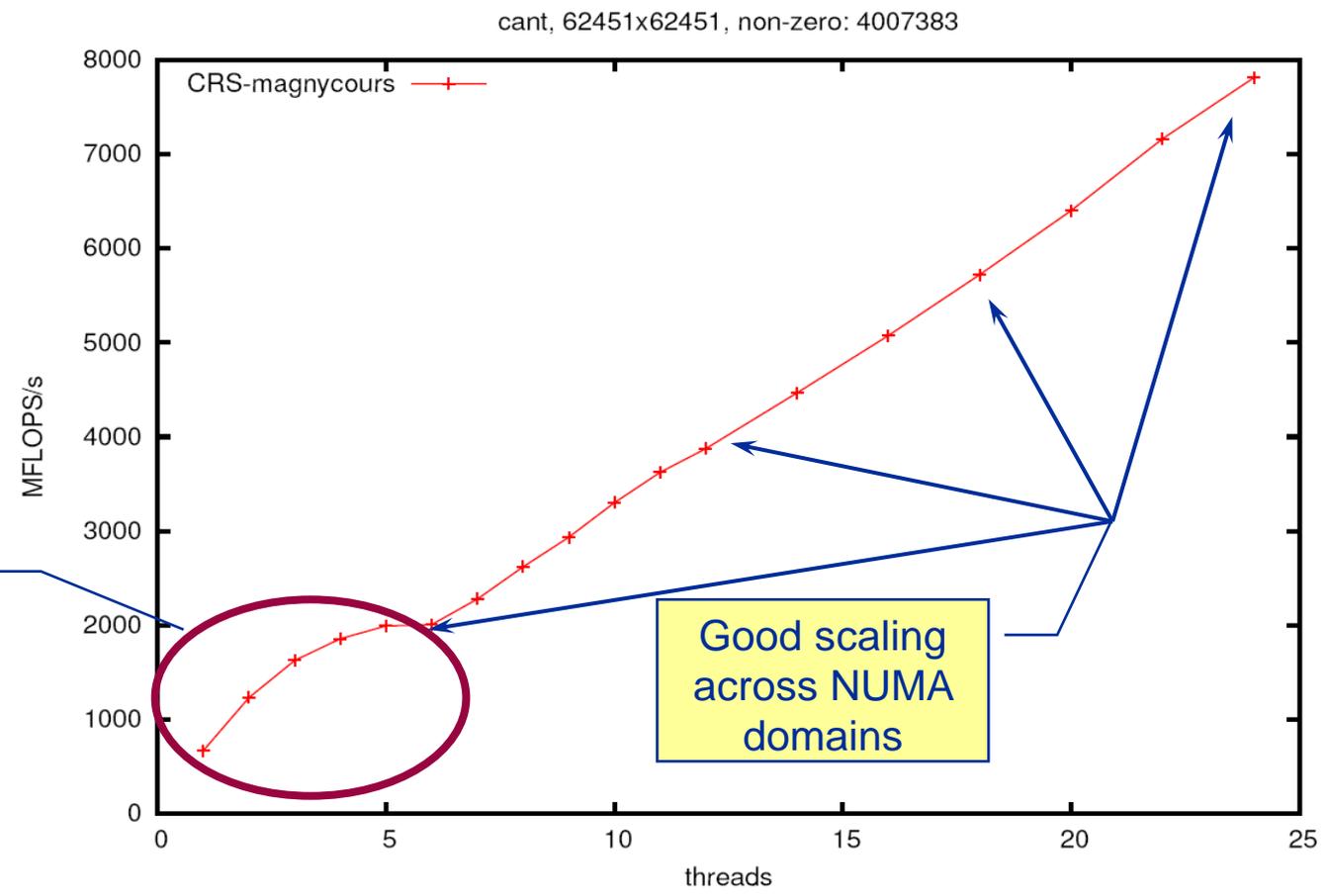
Strong scaling on one XE6 Magny-Cours node



## Case 1: Large matrix



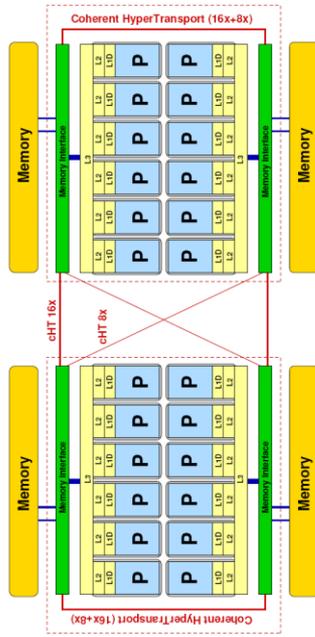
Intrasocket bandwidth bottleneck



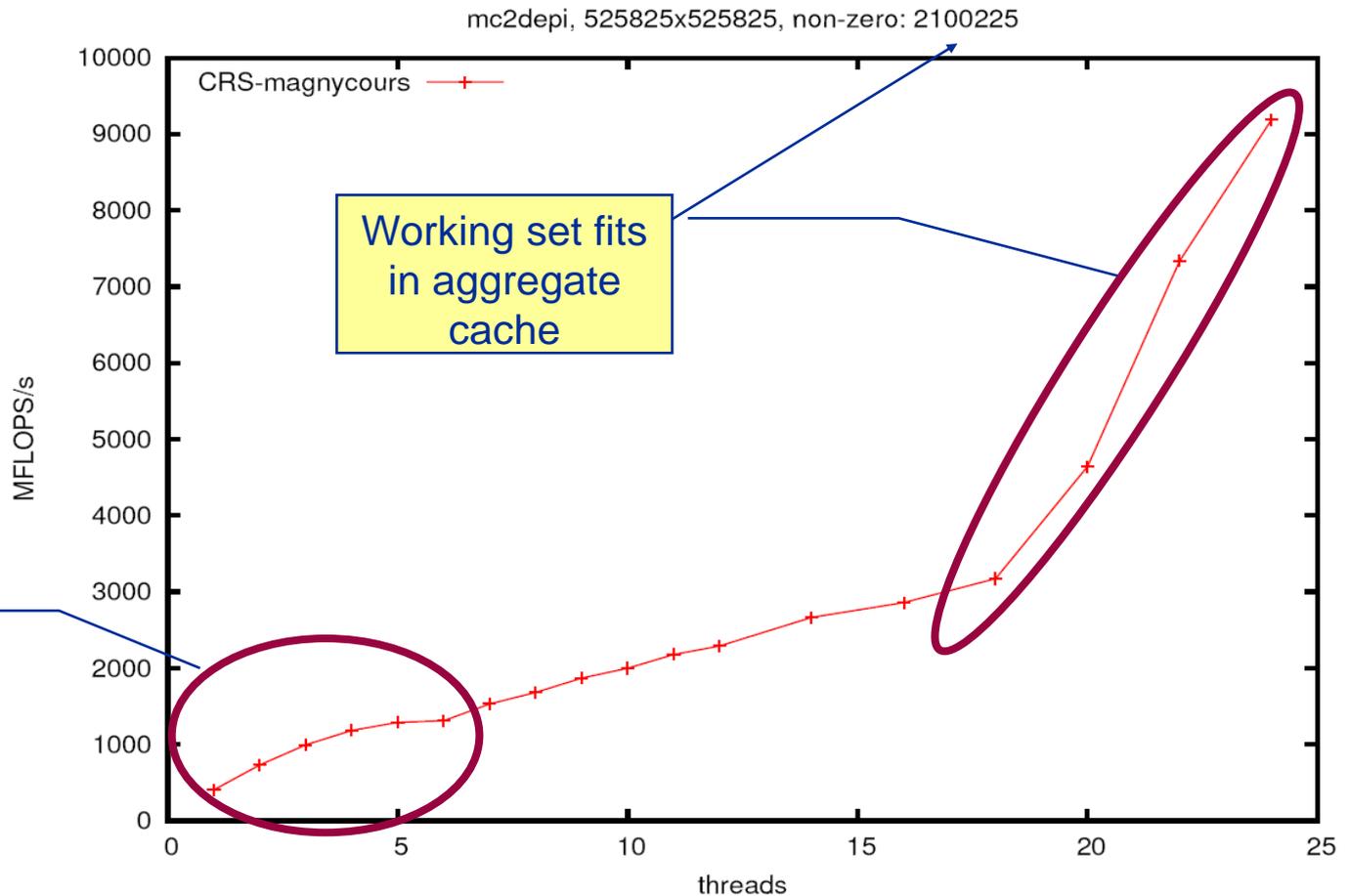
Good scaling across NUMA domains



### Case 2: Medium size



Intrsocket bandwidth bottleneck

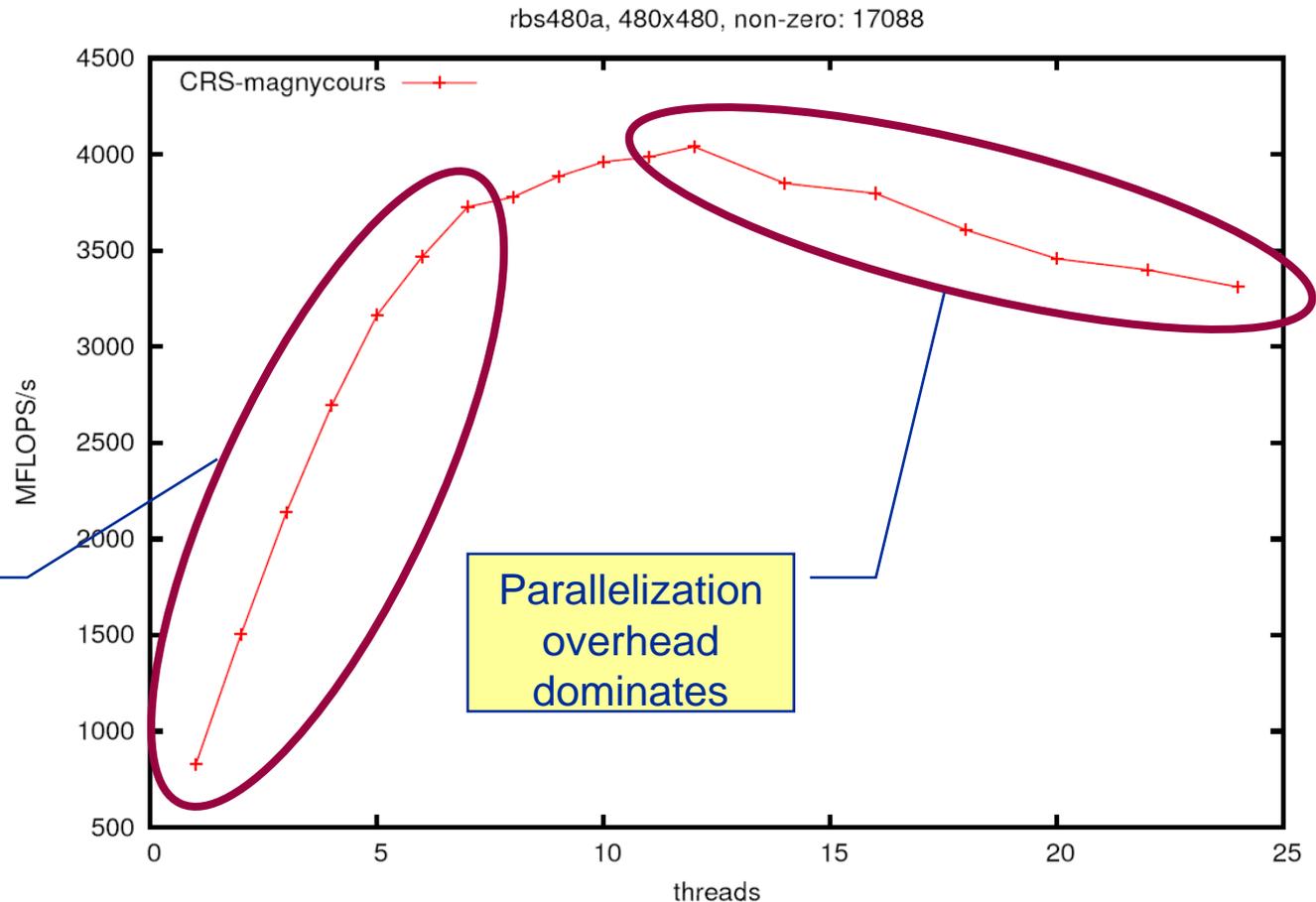




### Case 3: Small size



No bandwidth bottleneck





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

```

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

```

- **DP CRS comp. intensity**

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

- $\alpha$  quantifies traffic for loading RHS
  - $\alpha = 0 \rightarrow$  RHS is in cache
  - $\alpha = 1/N_{nzs} \rightarrow$  RHS loaded once
  - $\alpha = 1 \rightarrow$  no cache
  - $\alpha > 1 \rightarrow$  Houston, we have a problem!
- “Expected” performance =  $b_s \times I_{CRS}$
- Determine  $\alpha$  by measuring **performance and actual memory traffic**
  - Maximum memory BW may not be achieved with spMVM



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

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

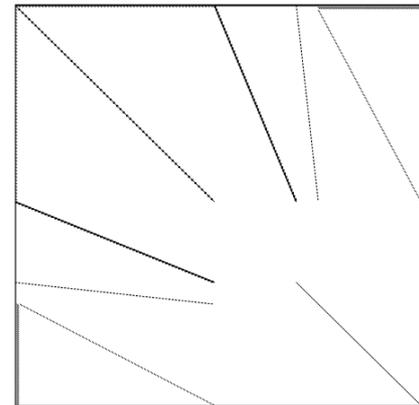
- Solve for  $\alpha$ : 
$$\alpha = \frac{1}{4} \left( \frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{8}{N_{nzs}} \right)$$

- Example: kkt\_power matrix from the UoF collection on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6, N_{nzs} = 7.1$
- $V_{meas} \approx 258 \text{ MB}$
- $\rightarrow \alpha = 0.43, \alpha N_{nzs} = 3.1$
- $\rightarrow$  RHS is loaded 3.1 times from memory
- and:

$$\frac{I_{CRS}^{DP}(1/N_{nzs})}{I_{CRS}^{DP}(\alpha)} = 1.15$$

15% extra traffic  $\rightarrow$  optimization potential!

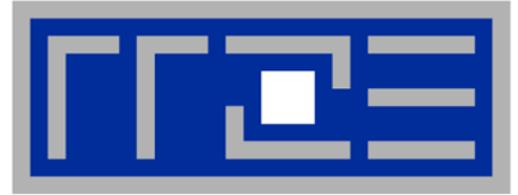




- **Conclusion from Roofline analysis**

- The roofline model does not work 100% 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!***



## **Case study: A Jacobi smoother**

**The basics in two dimensions**

Layer conditions

Validating the model in 3D

Optimization by spatial blocking in 3D



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

```
do iter = 1, max_iterations
```

```
    Perform sweep over regular grid:  $y(:) \leftarrow x(:)$ 
```

```
    Swap  $y \leftrightarrow x$ 
```

```
enddo
```

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

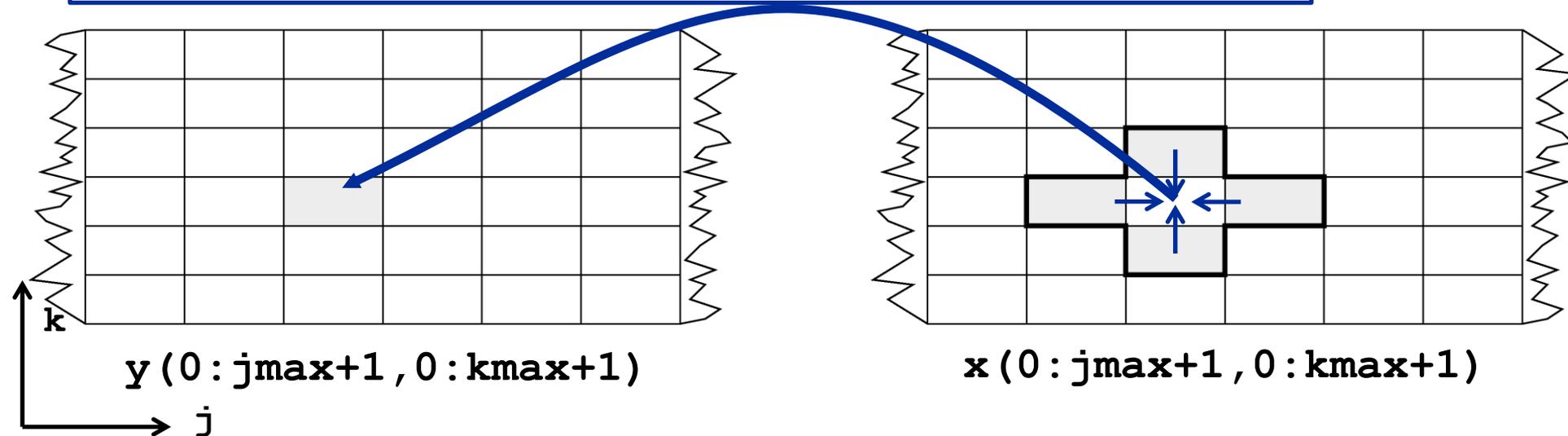
# Jacobi-type 5-pt stencil in 2D



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
```

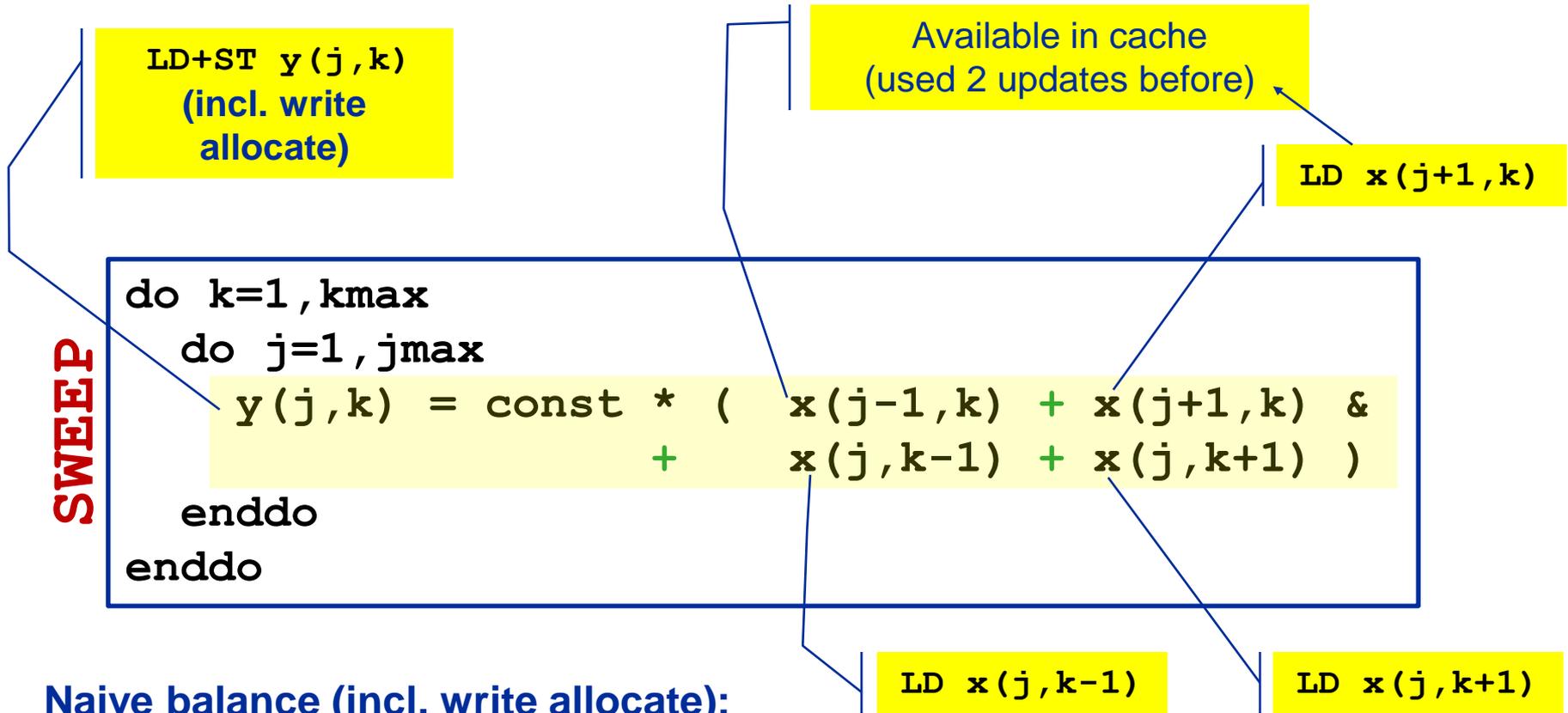
Lattice Update (LUP)



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



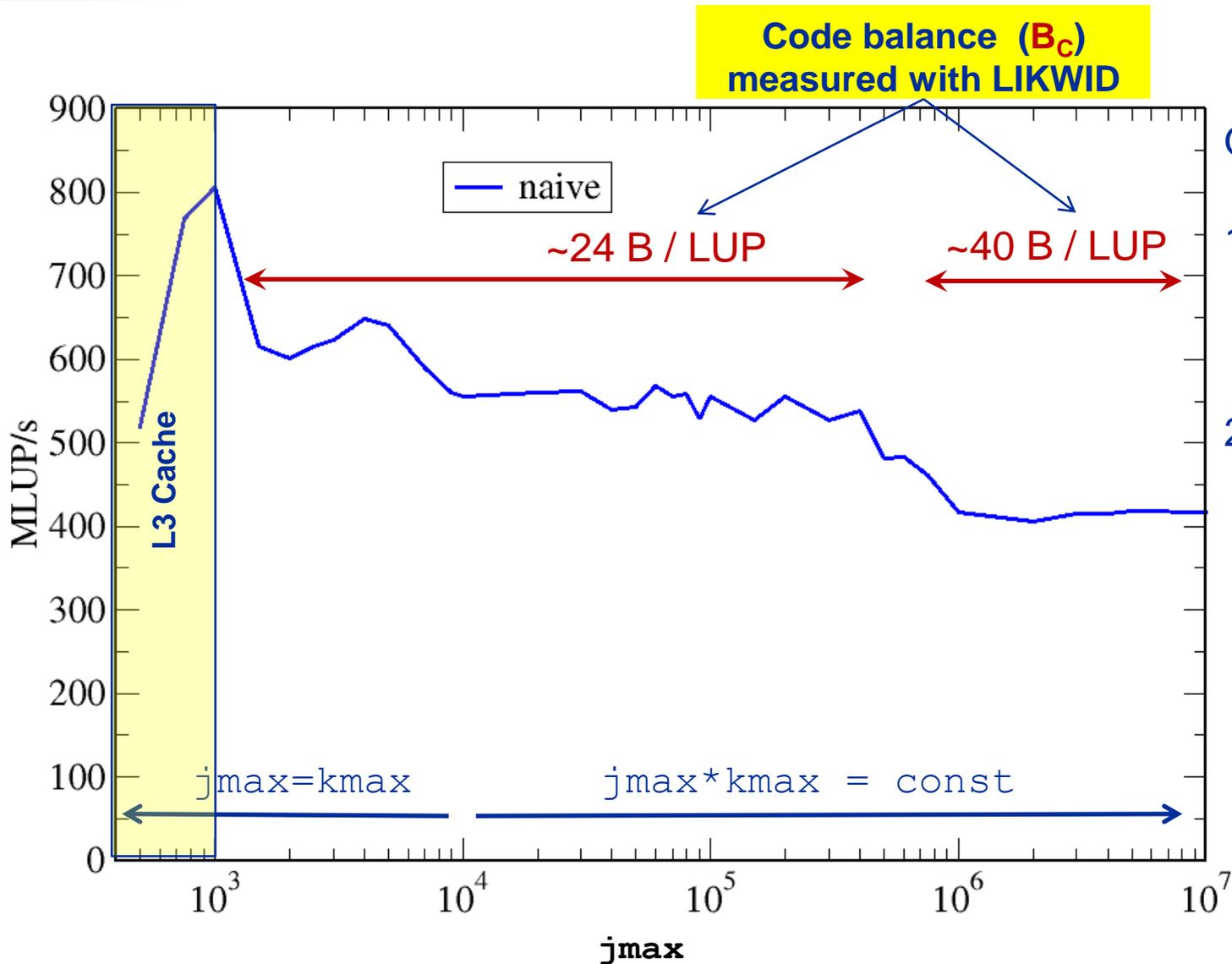
Naive balance (incl. write allocate):

$x( : , : ) : 3 \text{ LD} +$

$y( : , : ) : 1 \text{ ST} + 1 \text{ LD}$

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

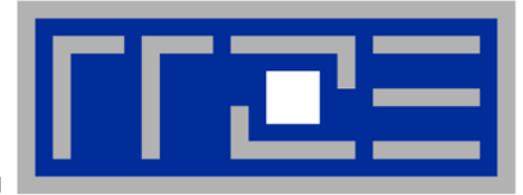
# Jacobi 5-pt stencil in 2D: Single core performance



Questions:

1. How to achieve 24 B/LUP also for large  $j_{\max}$ ?
2. How to sustain  $>600$  MLUP/s for  $j_{\max} > 10^4$  ?

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



## **Case study: A Jacobi smoother**

The basics in two dimensions

**Layer conditions**

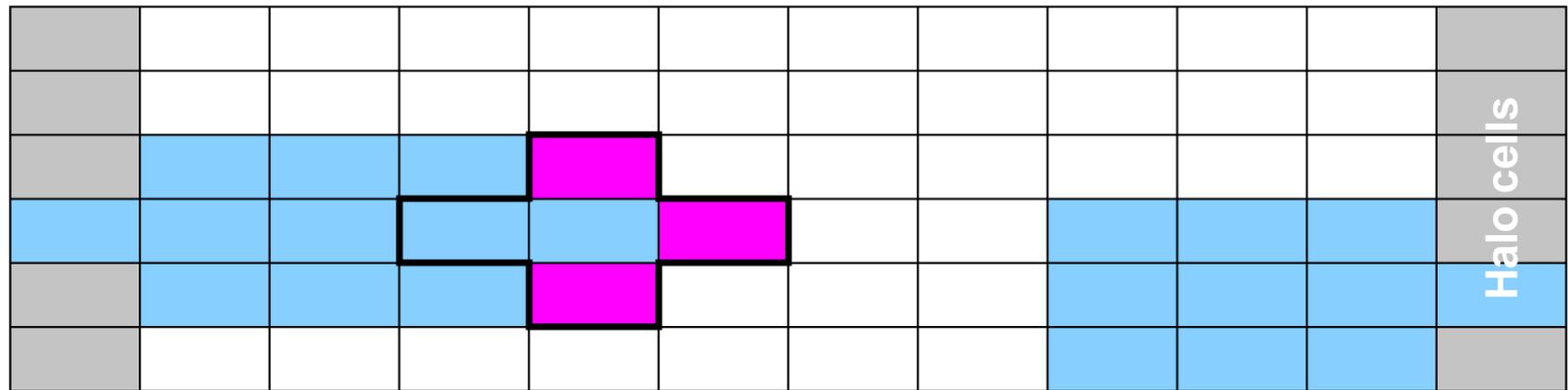
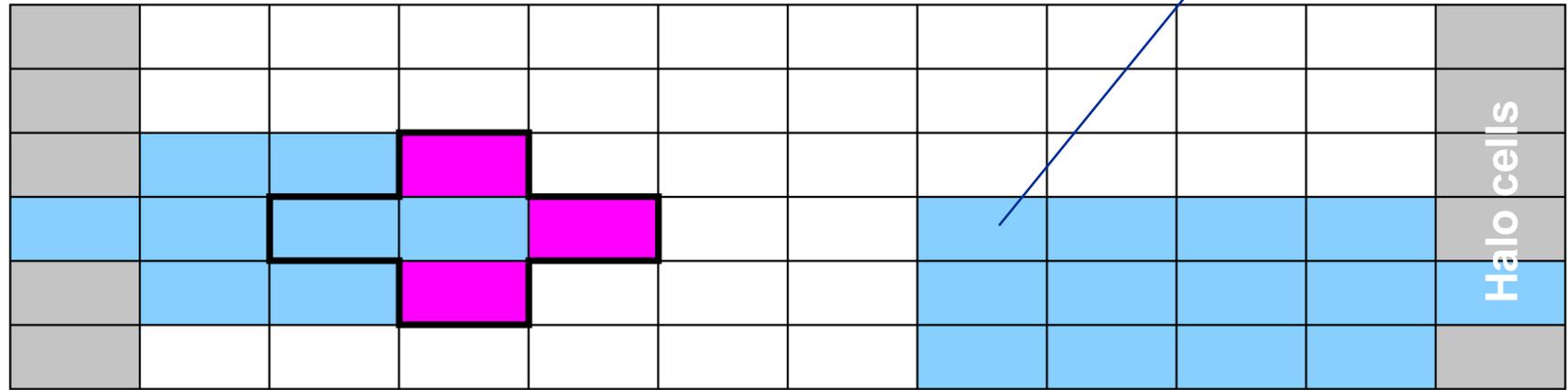
Validating the model in 3D

Optimization by spatial blocking in 3D



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

cached

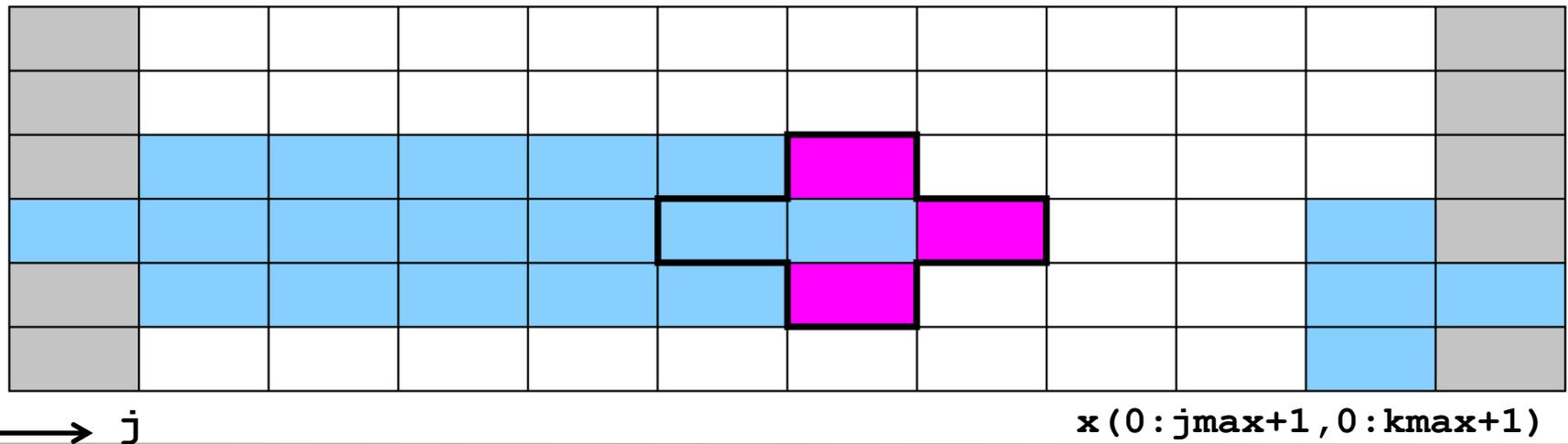
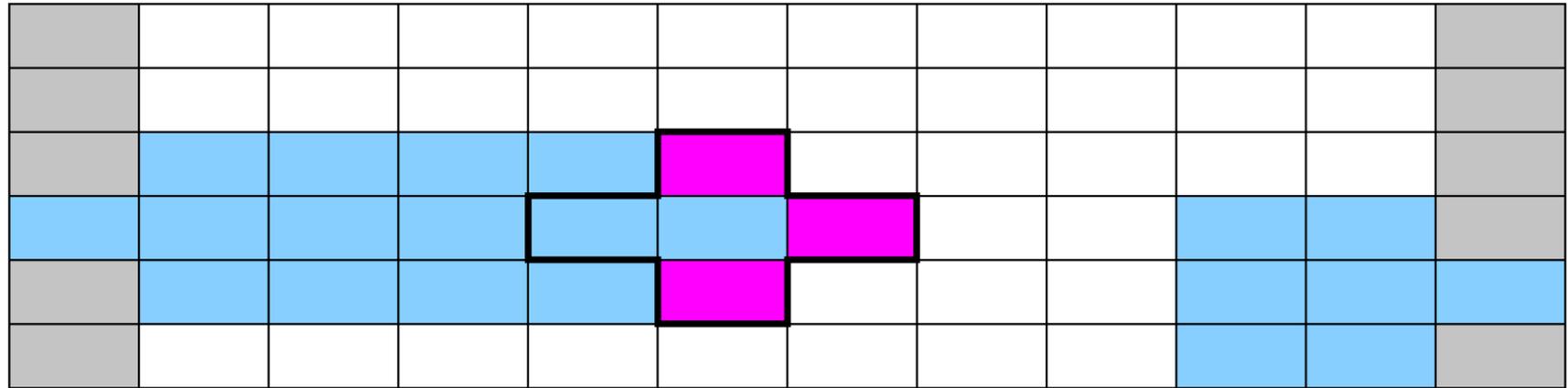


k  
↑

j  
→

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

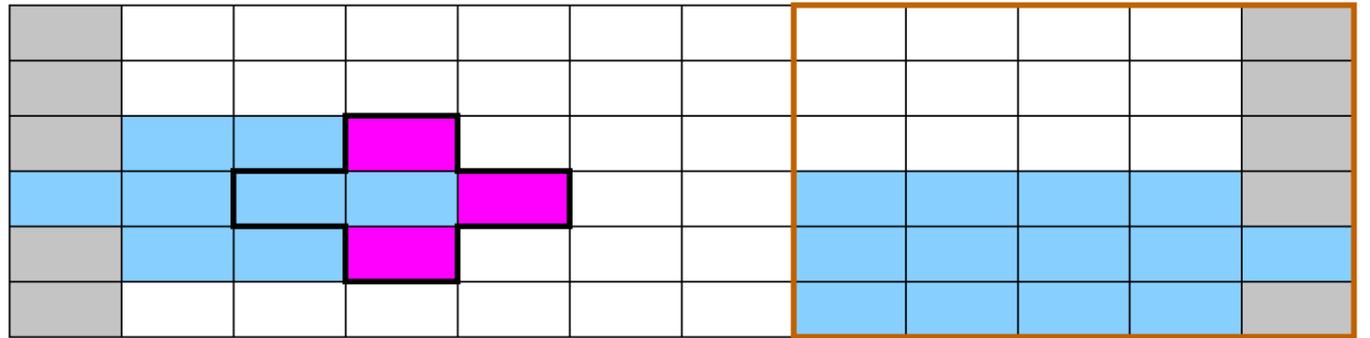
Worst case: Cache not large enough to hold 3 layers (rows) of grid  
(+assume „Least Recently Used“ replacement strategy)



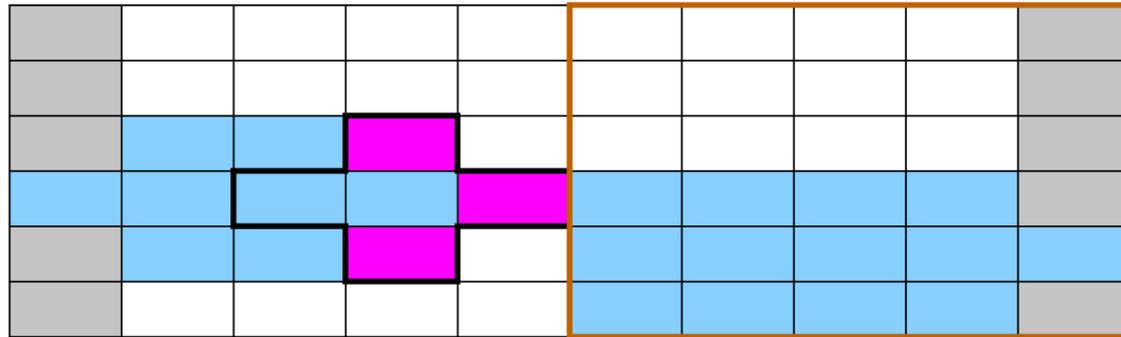
# Analyzing the data flow



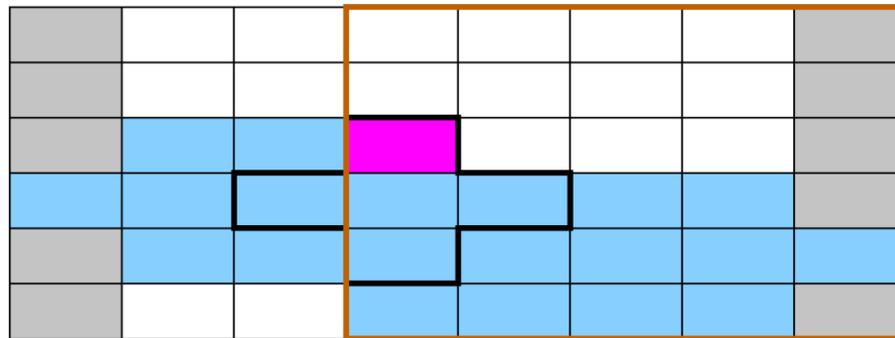
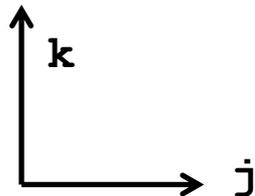
Reduce inner (j-) loop dimension successively



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



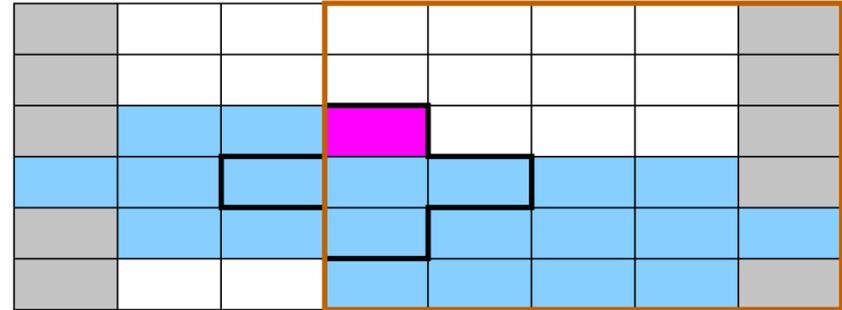
Best case: 3 „layers“ of grid fit into the cache!



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



## 2D 5-pt Jacobi-type stencil



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

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

“Layer condition”

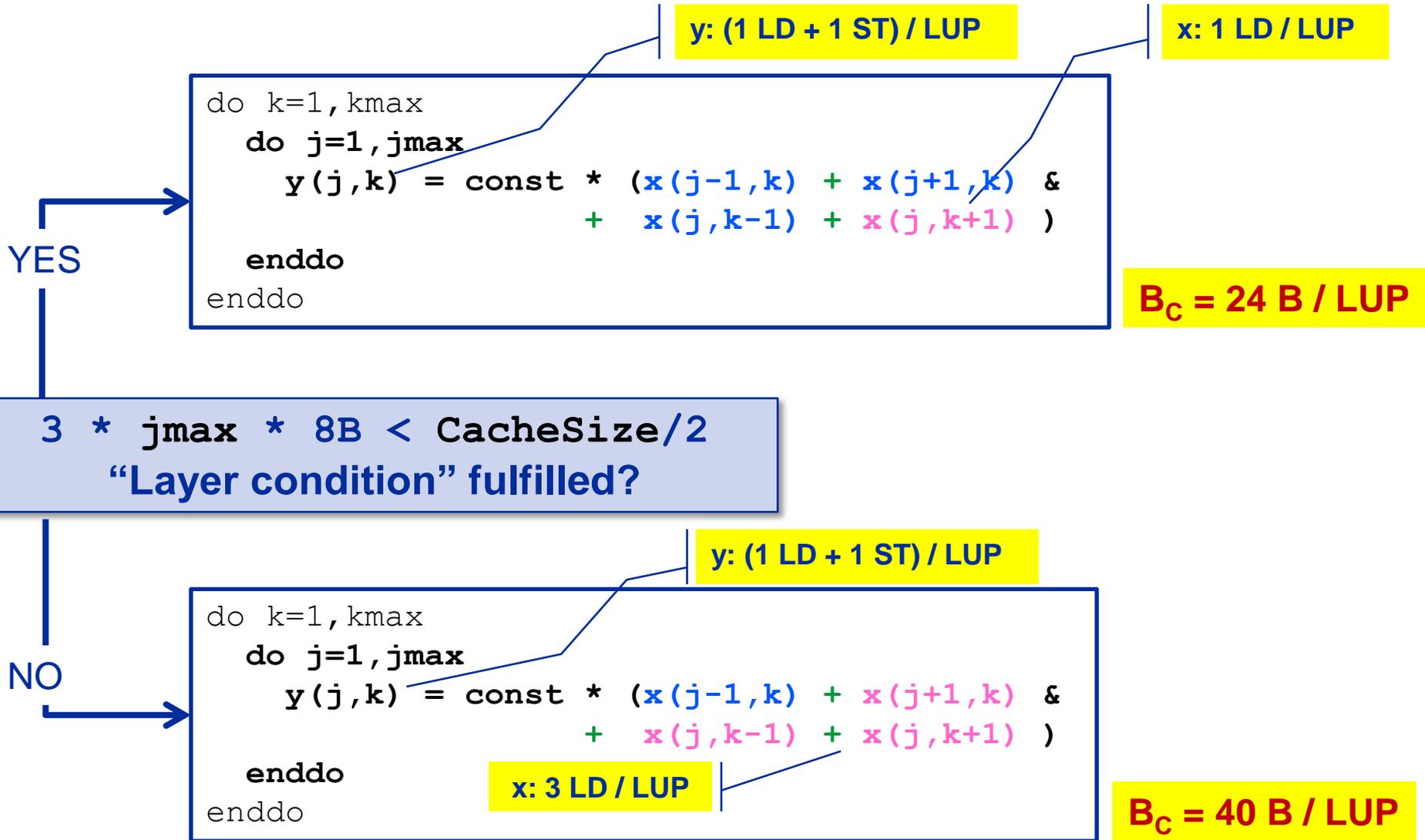
3 rows of  
j<sub>max</sub>

double  
precision

Safety margin  
(Rule of thumb)

Layer condition:

- Does not depend on outer loop length ( $k_{\max}$ )
- No strict guideline (cache associativity – data traffic for  $y$  not included)
- Needs to be adapted for other stencils (e.g., 3D 7-pt stencil)





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

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

## → Determine for given CacheSize an appropriate **jbblock** value:

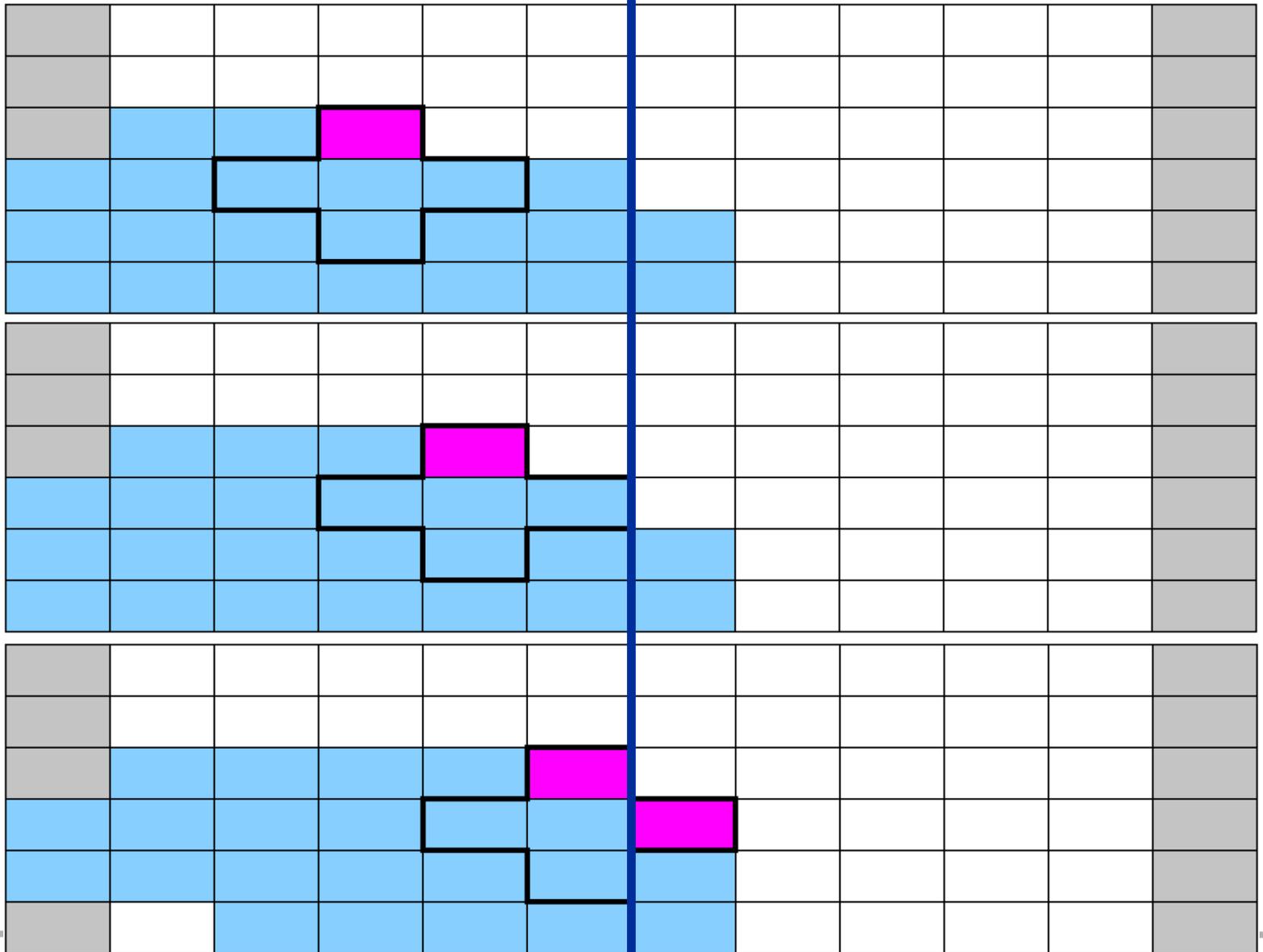
$$\mathbf{jbblock} < \mathbf{CacheSize} / 48 B$$

# Establish the layer condition by blocking

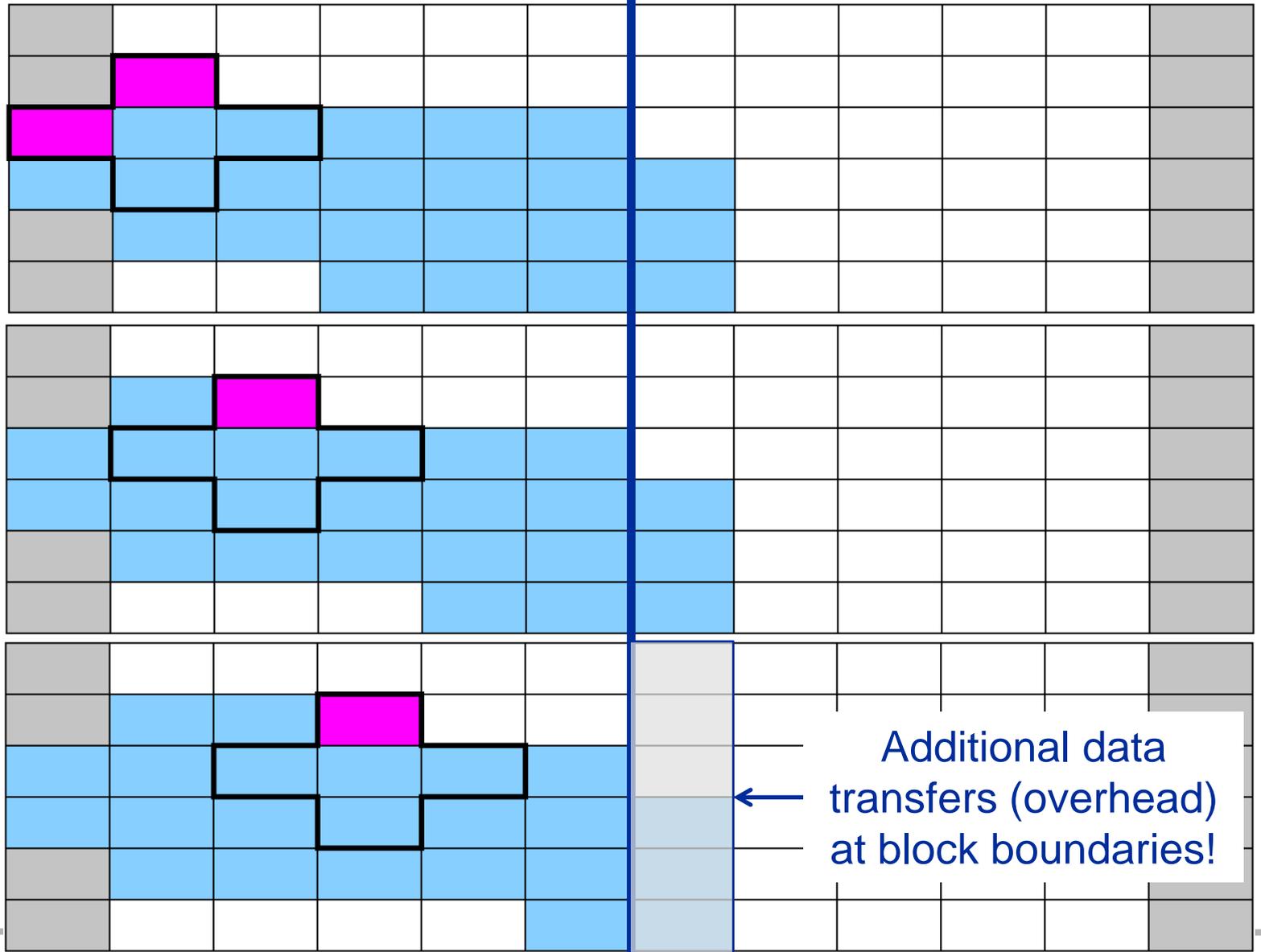


Split up domain into subblocks:

e.g. block size = 5



# Establish the layer condition by blocking

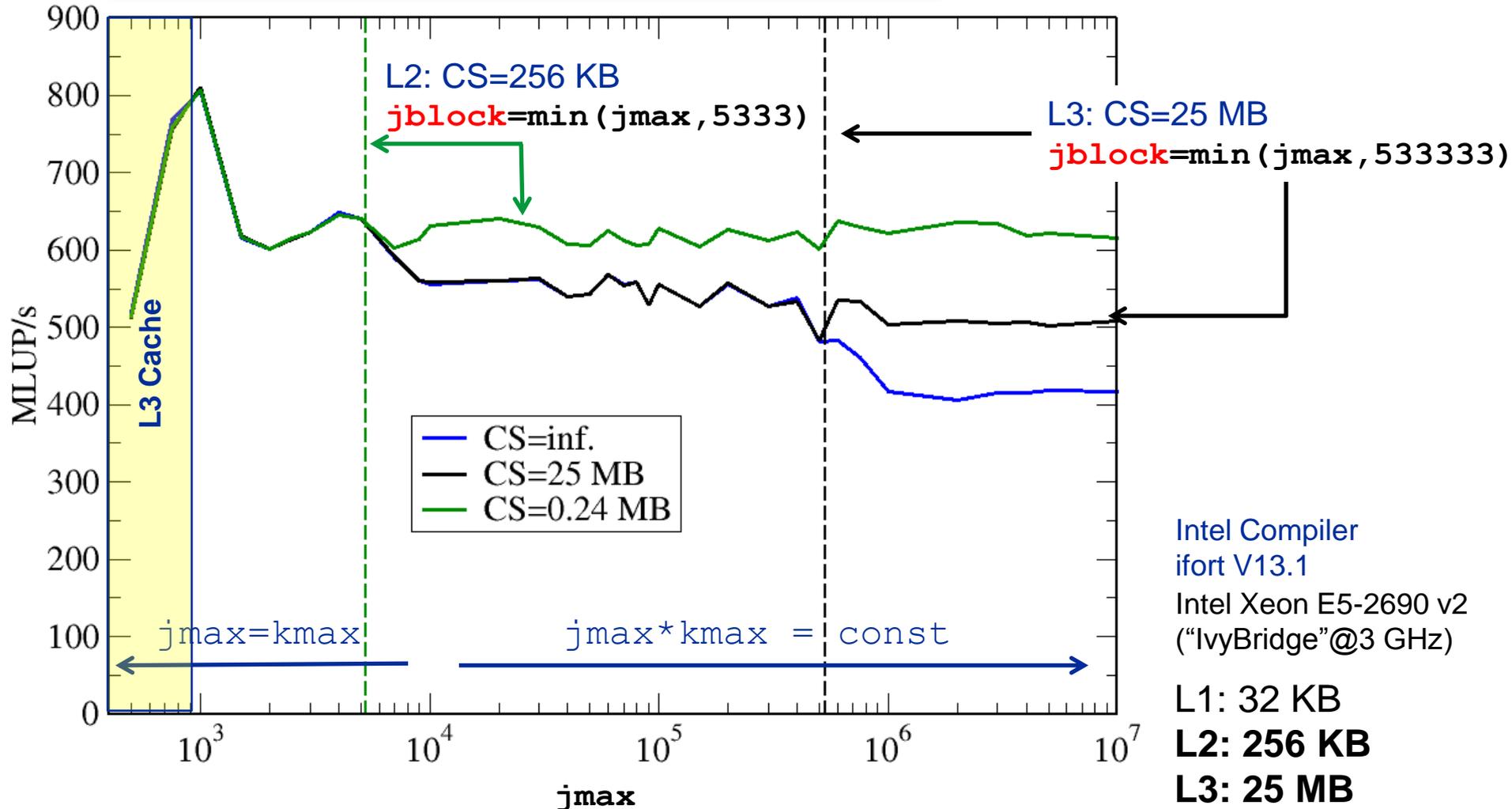


# Establish layer condition by spatial blocking

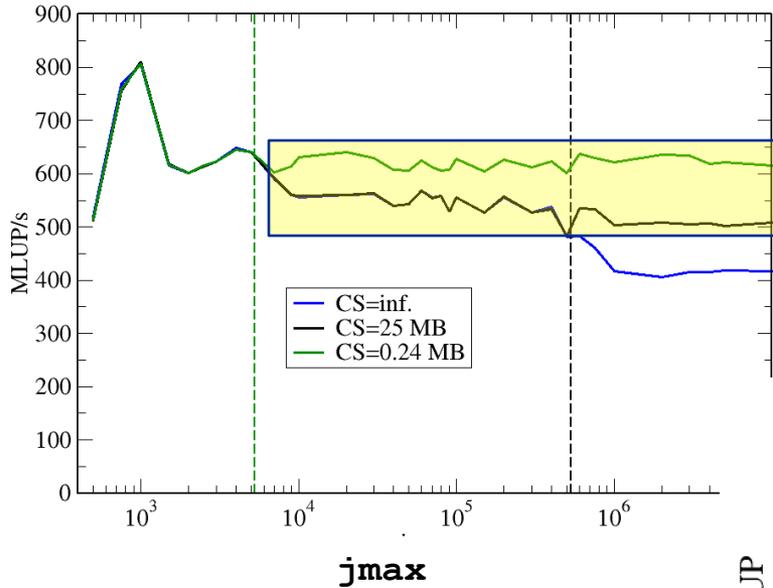


$$jblock < CacheSize / 48 B$$

Which cache to block for?

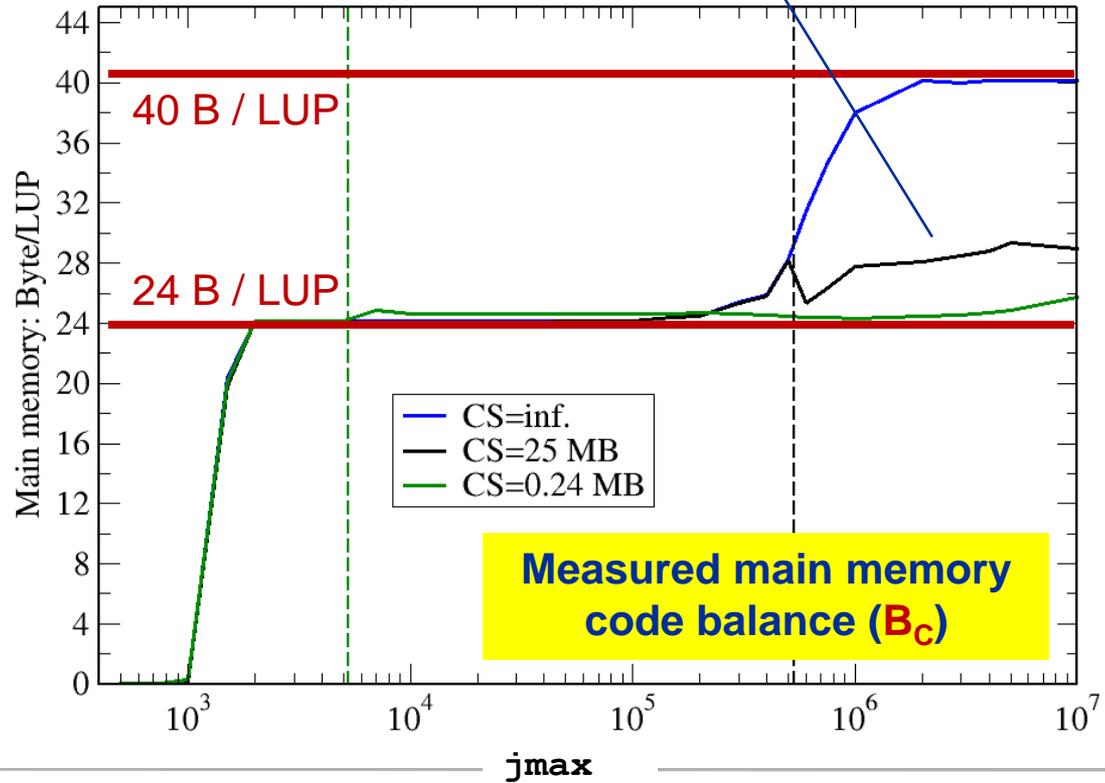


# Layer condition & spatial blocking: Memory code balance



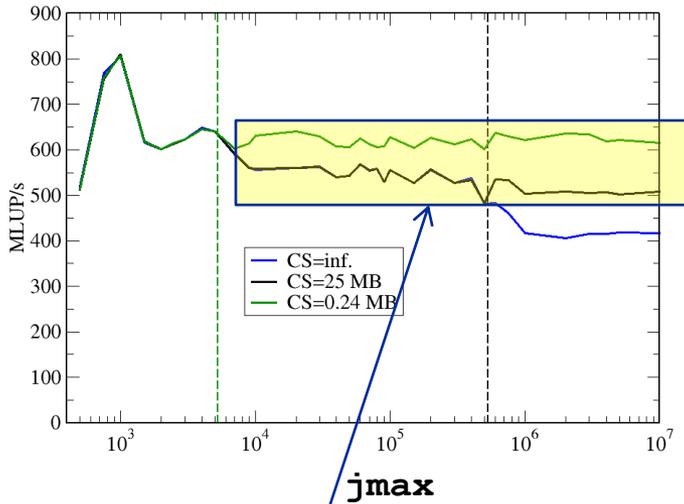
Main memory access is not reason for different performance

Blocking factor (CS=25 MB) too large

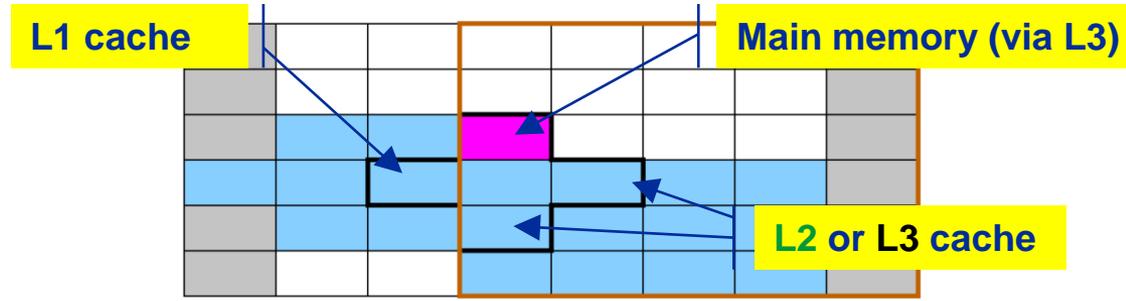


Intel Compiler  
ifort V13.1  
Intel Xeon E5-2690 v2  
("IvyBridge"@3 GHz)

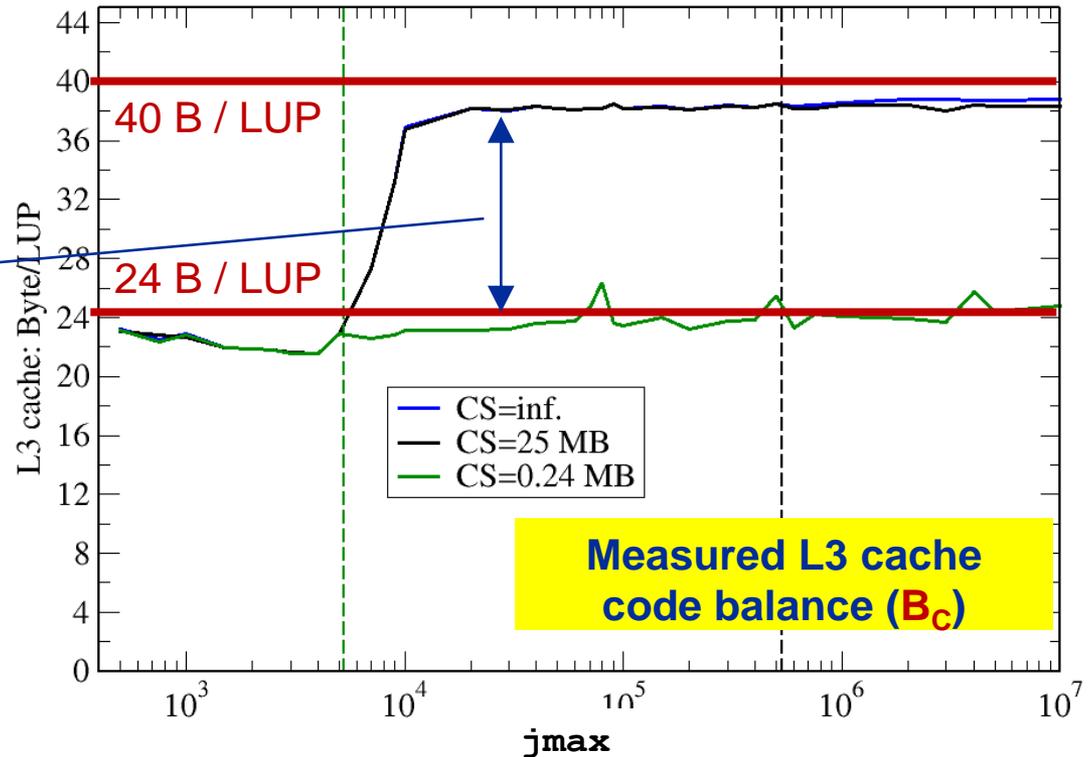
# Layer condition & spatial blocking: L3 cache balance



## Data accesses to L3 cache (blocking)



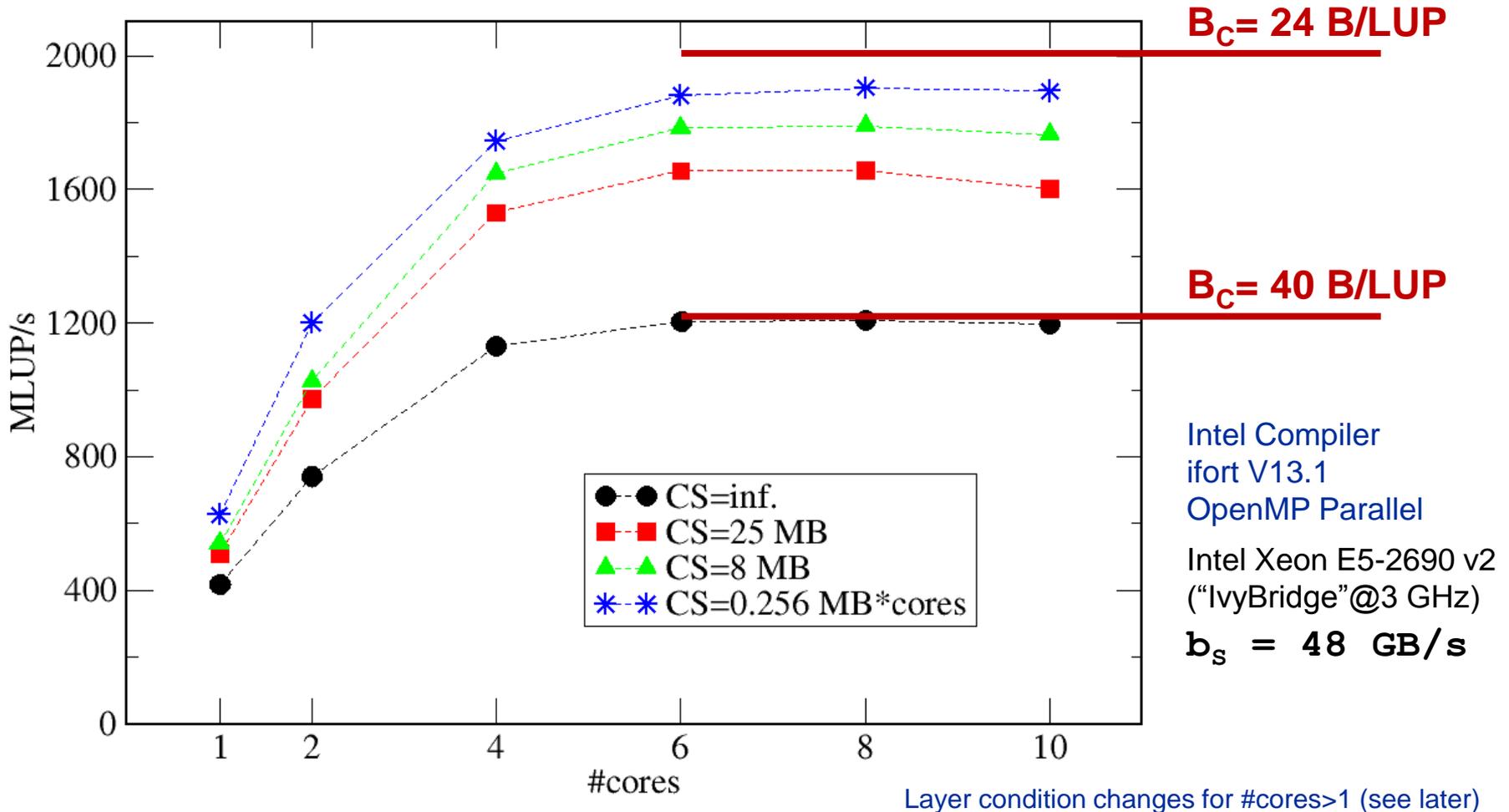
**Impact of total L3 traffic:  
24 B/LUP vs. 40 B/LUP**



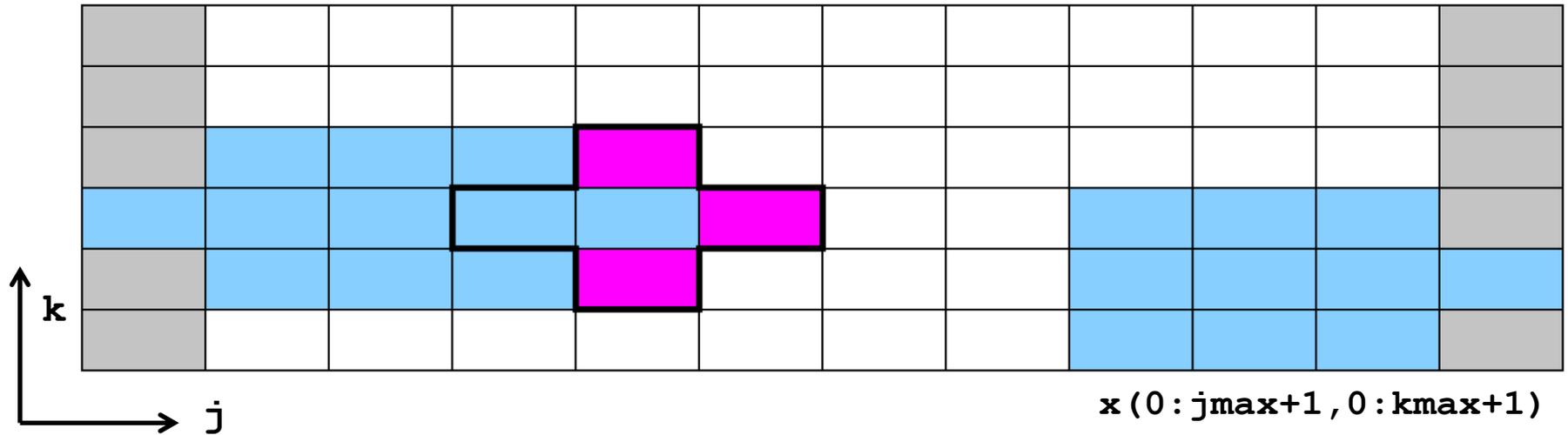
Intel Compiler  
ifort V13.1  
Intel Xeon E5-2690 v2  
("IvyBridge"@3 GHz)



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



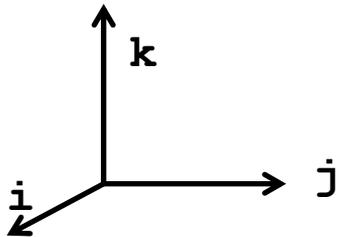
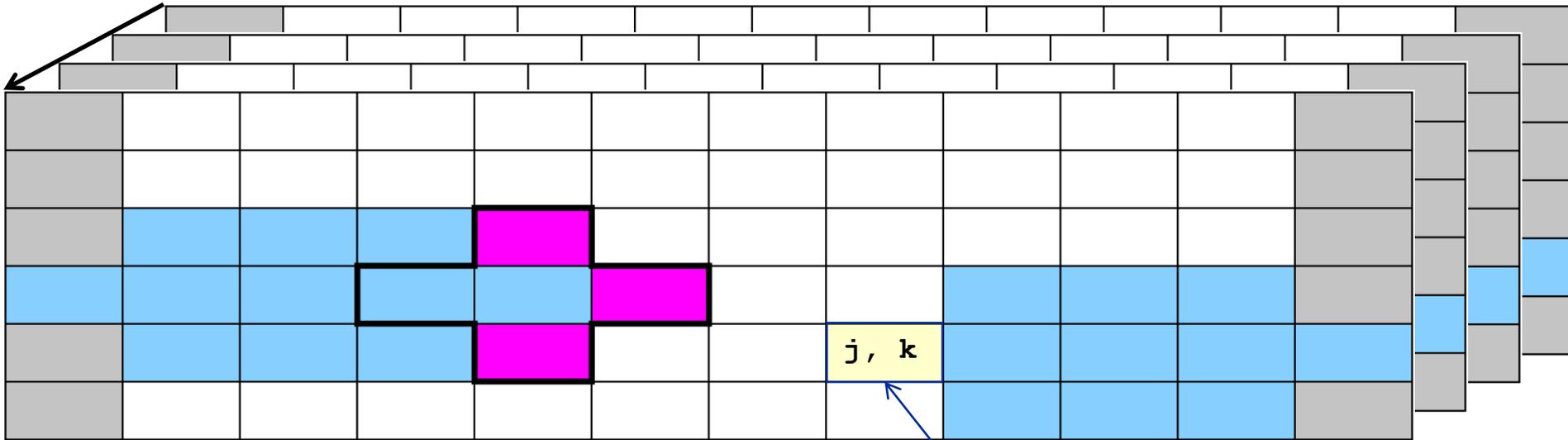
- 2D



## Towards 3D understanding

- Picture can be considered as 2D cut of 3D domain for (new) fixed **i**-coordinate:

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



- $\mathbf{x}(0:\mathbf{imax}+1, 0:\mathbf{jmax}+1, 0:\mathbf{kmax}+1)$  – Assume **i-direction** contiguous in main memory (Fortran notation)
- Stay at 2D picture and consider **one cell of j-k plane** as a contiguous slab of elements in **i-direction**:  $\mathbf{x}(0:\mathbf{imax}, \mathbf{j}, \mathbf{k})$

# Layer condition: From 2D 5-pt to 3D 7-pt Jacobi-type stencil



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

```
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
```

2D

$$B_C = 24 B / \text{LUP}$$

```
do k=1, kmax
  do j=1, jmax
    do i=1, imax
      y(i,j,k) = const * (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
```

3D

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

$$B_C = 24 B / \text{LUP}$$

# 3D 7-pt Jacobi-type Stencil (sequential)



“Layer condition”

$$3 * j_{\max} * i_{\max} * 8B < CS/2$$

“Layer condition” OK →  
5 accesses to  $x()$  served by cache

```
do k=1, kmax
```

```
  do j=1, jmax
    do i=1, imax
```

```
      y(i, j, k) = const. * ( 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
```

**Question:**

**Does parallelization/multi-threading change the layer condition?**



**Basic guideline:**  
Parallelize outermost loop

```
!$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
```

**Equally large chunks in k-direction**  
→ **“Layer condition” for each thread**

**“Layer condition”:**  $nthreads * 3 * jmax * imax * 8B < CS/2$

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



“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
```

$B_C = 24 B / LUP$

Intel® Xeon® Processor E5-2690 v2

10 cores@3 GHz

CacheSize = 25 MB (L3)

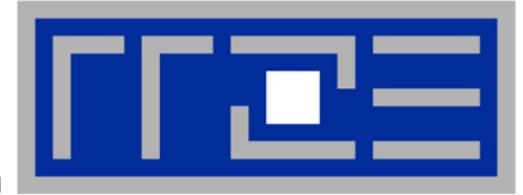
MemBW = 48 GB/s

**Roofline model:**

$\max MLUPs = MemBW / (24 B/LUP)$

**Best performance:**

$P = 2000 MLUPs$



## **Case study: A Jacobi smoother**

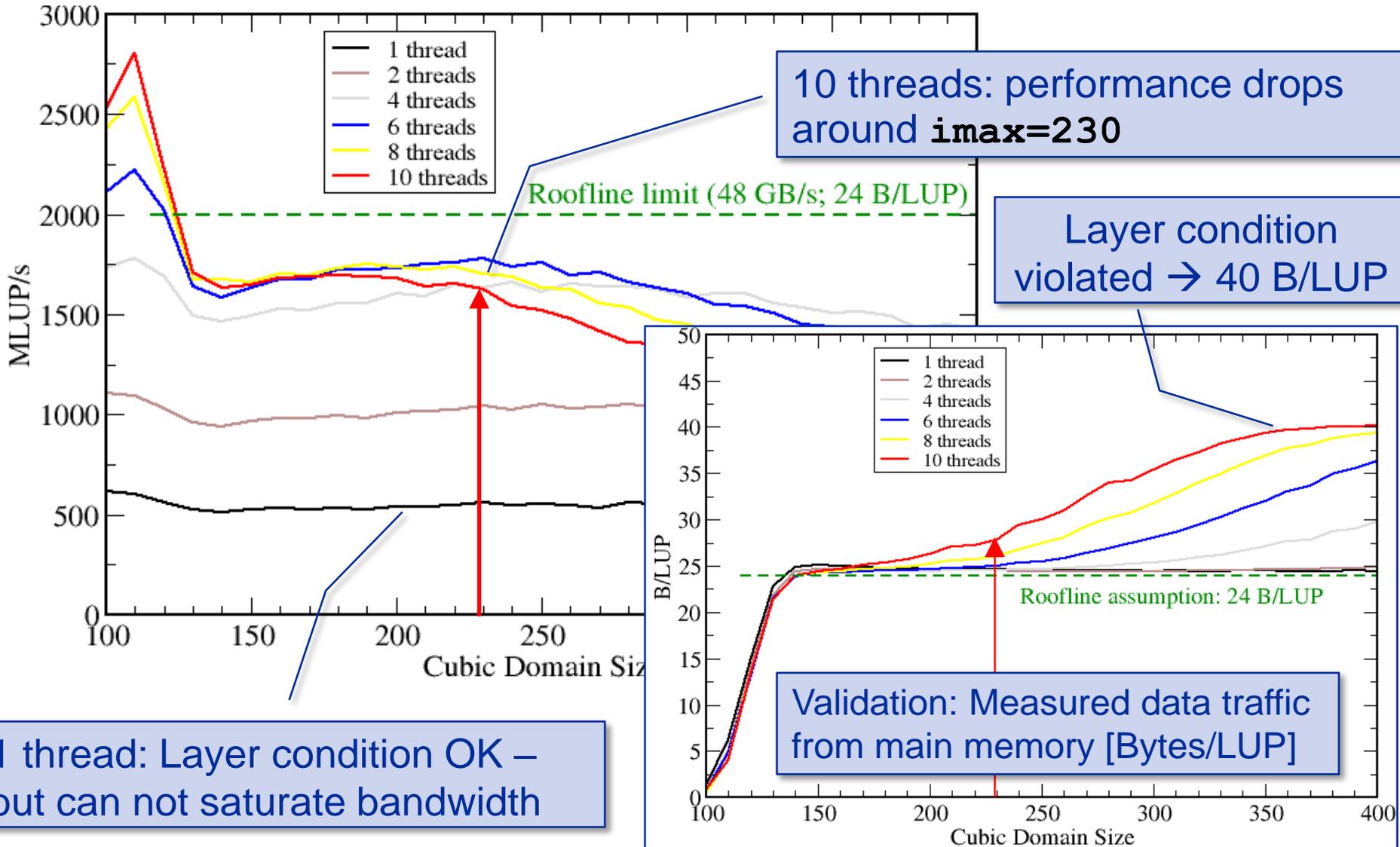
The basics in two dimensions

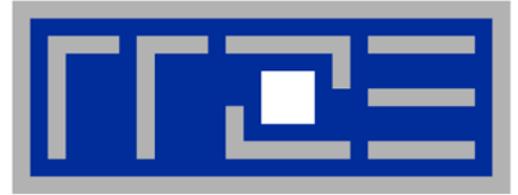
Layer conditions

**Validating the model in 3D**

Optimization by spatial blocking in 3D

# Jacobi Stencil – OpenMP parallelization (I)





## **Case study: A Jacobi smoother**

The basics in two dimensions

Layer conditions

Validating the model in 3D

**Spatial blocking in 3D**

# Jacobi Stencil – simple spatial blocking



```
do j=1,jmax,jblock ! Assume jmax is multiple of jblock
```

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
```

```
do k=1,kmax
```

```
do j=jb, (jb+jblock-1) ! Loop length jblock
```

```
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
```

```
enddo
```

“Layer condition” (j-Blocking)  
 $nthreads * 3 * jblock * imax * 8B < CS / 2$

Ensure layer condition by choosing **jblock** appropriately (Cubic Domains):

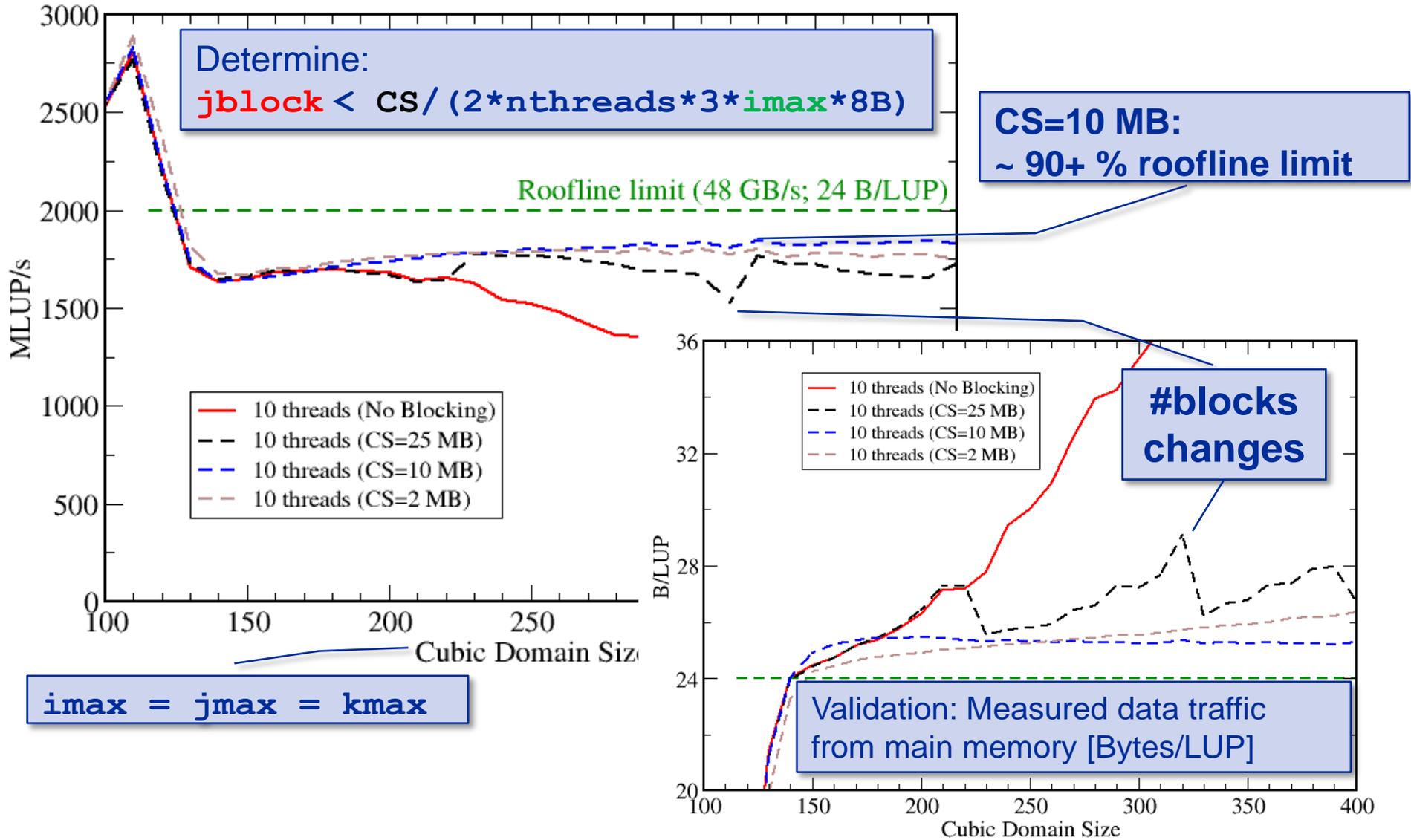
$$jblock < CS / (imax * nthreads * 48B)$$

Testsystem: Intel® Xeon® Processor E5-2690 v2 (10 cores / 3 GHz)

MemBW = 48 GB/s, CS = 25 MB (L3)

maxMLUPs = 2000 MLUPs

# Jacobi Stencil – simple spatial blocking



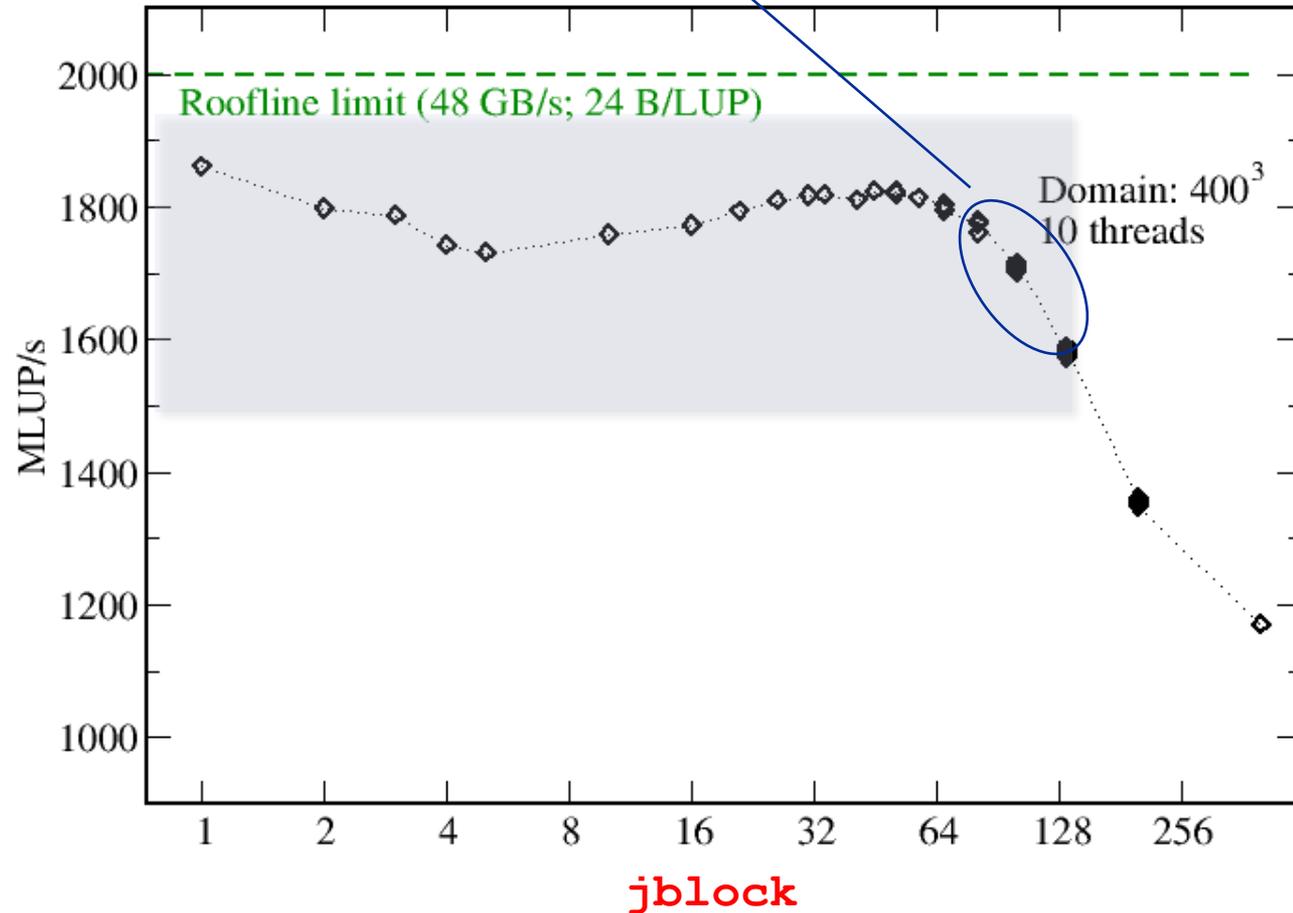
Layer condition estimates appropriate `jblock`:  
$$jblock < CS / (2 * nthreads * 3 * imax * 8B)$$

`CS=25 MB`  
`nthreads=10`  
`imax=400`

$jblock < 130$

`jblock=32, ..., 64`  
useful choices

Layer condition a bit too optimistic

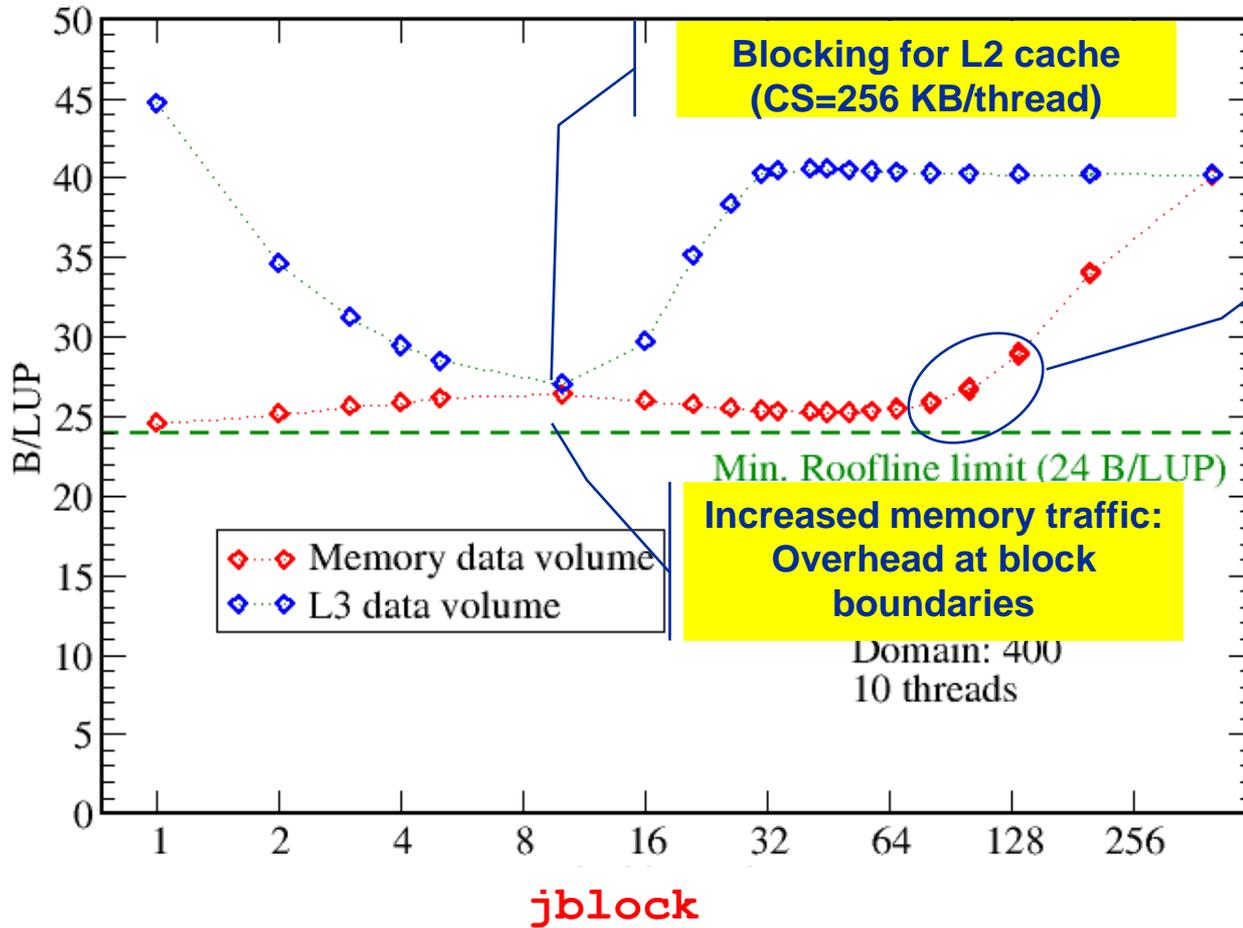




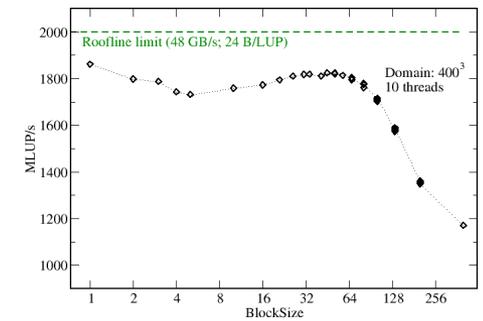
Layer condition estimates appropriate `jblock`:  

$$jblock < CS / (2 * nthreads * 3 * imax * 8B)$$

`jblock < 130`



**Layer condition a bit too optimistic**



# Jacobi Stencil – can we further improve?



```
do k=1, kmax
```

“Layer condition” OK →  
5 accesses to `x()` served by cache

```
do j=1, jmax
```

```
do i=1, imax
```

```

y(i,j,k) =const. * (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
```

Total data transfer / LUP:

```

(8+8) B/LUP for y() (ST+Write+Allocate)
+ 8 B/LUP for x(i,j,k+1)
→ 24 B/LUP

```

Total data transfer / LUP:

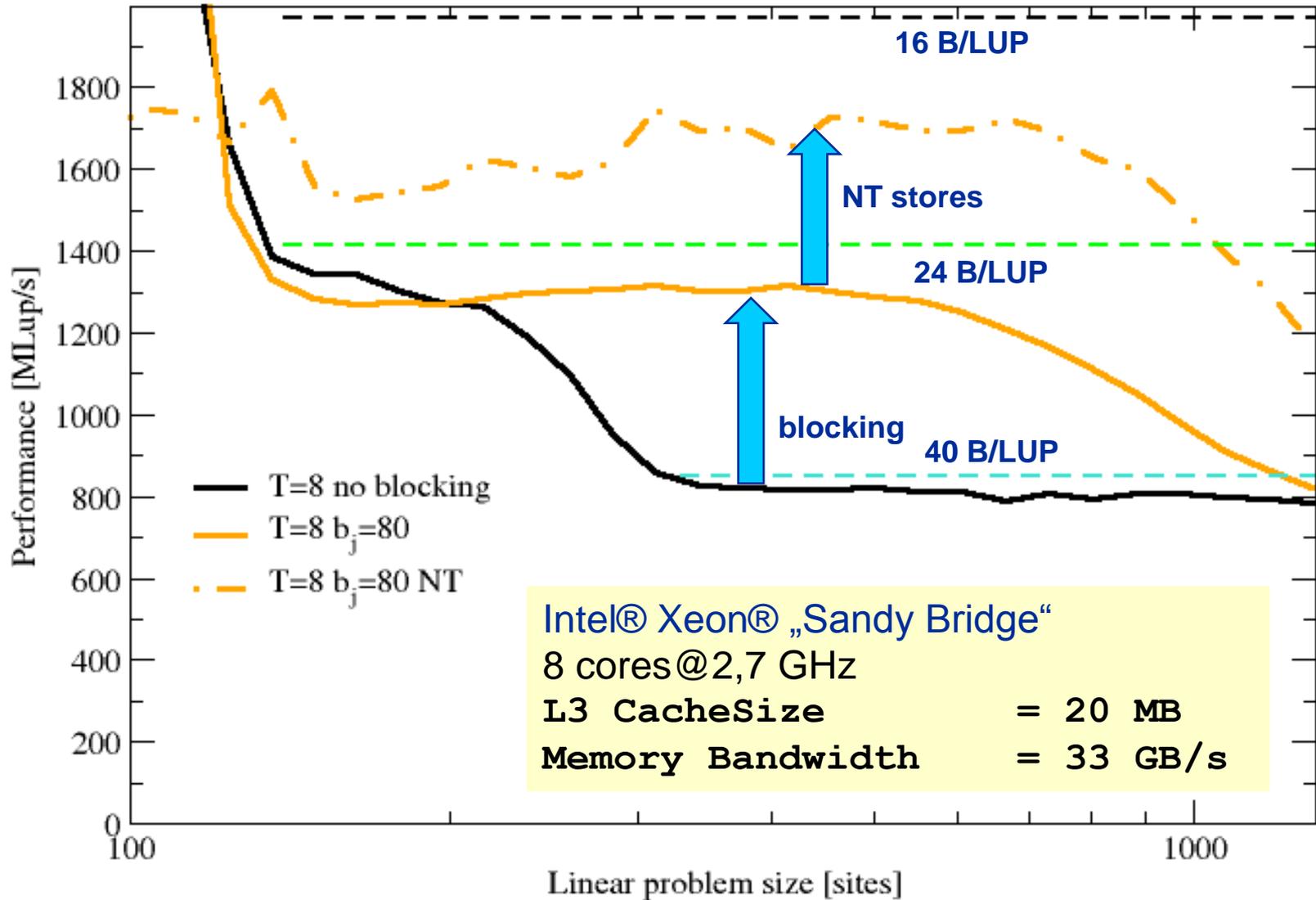
```

8 B/LUP for y() (NT-Store)
+ 8 B/LUP for x(i,j,k+1)
→ 16 B/LUP

```

Use NT-stores to  
avoid “Write Allocate”

# Jacobi Stencil – Blocking + NT-stores

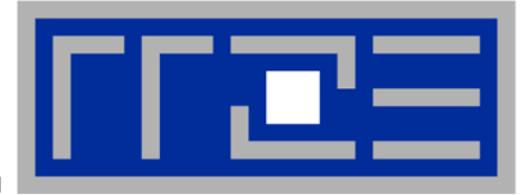




- We have **made sense** of the **memory-bound performance** vs. **problem size**
  - “**Layer conditions**” lead to **predictions of code balance**
  - Achievable memory bandwidth is input parameter
- “**What part of the data comes from where**” is a crucial question
- The model works only if the **bandwidth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

---

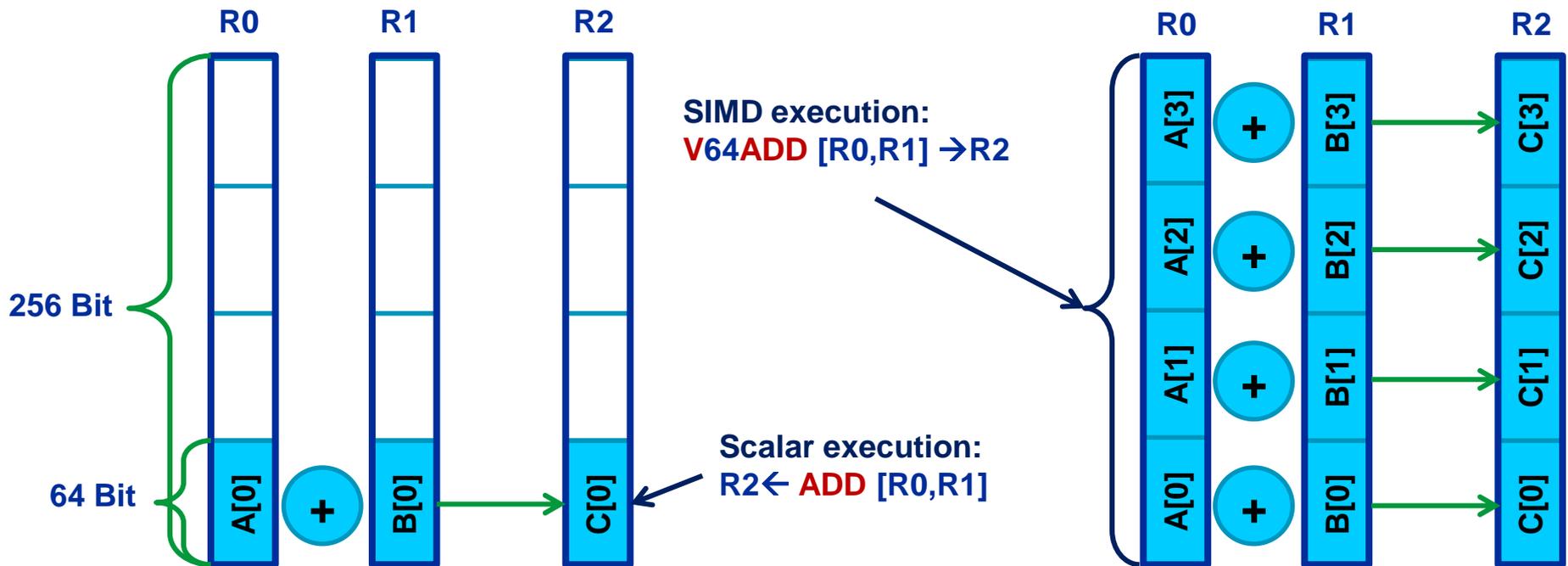
# DEMO



**Coding for**  
**Single Instruction Multiple Data 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**



- 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
```

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

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

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

**LABEL1:**

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



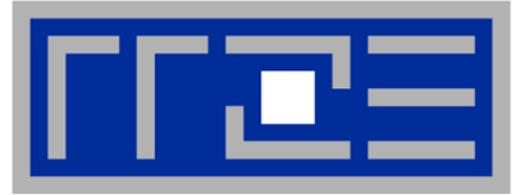
- **No SIMD vectorization for loops with data dependencies:**

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

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

```
void scale_shift(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  $A \rightarrow \&C[-1]$  and  $B \rightarrow \&C[-2]$   
→  $C[i] = C[i-1] + C[i-2]$ : **dependency** → **No SIMD**
- **If “pointer aliasing” is not used, tell it to the compiler:**
  - **-fno-alias** (Intel), **-Msaferptr** (PGI), **-fargument-noalias** (gcc)
  - **restrict** keyword (C only!):  
`void f(double restrict *a, double restrict *b) {...}`



# **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 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 -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**



## 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

`yymm0-yymm15` 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: Simplest code for the summation of the elements of a vector (single precision)



```
float sum = 0.0;
```

```
for (int j=0; j<size; j++){  
    sum += data[j];  
}
```

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

AT&T syntax:  
`addss 0(%rdx,%rax,4),%xmm0`

## Instruction code:

```
401d08:  f3 0f 58 04 82  
401d0d:  48 83 c0 01  
401d11:  39 c7  
401d13:  77 f3
```

```
addss  xmm0, [rdx + rax * 4]  
add    rax, 1  
cmp    edi, eax  
ja     401d08
```

(final sum across xmm0 omitted)

Instruction address

Opcodes

Assembly code

# Summation code (single precision): Improvements



```
1:
addss xmm0, [rsi + rax * 4]
add    rax, 1
cmp    eax,edi
js 1b
```

3 cycles add pipeline latency

Unrolling with sub-sums to break up register dependency

```
1:
addss xmm0, [rsi + rax * 4]
addss xmm1, [rsi + rax * 4 + 4]
addss xmm2, [rsi + rax * 4 + 8]
addss xmm3, [rsi + rax * 4 + 12]
add    rax, 4
cmp    eax,edi
js 1b
```

```
1:
vaddps ymm0,...,[rsi + rax * 4]
vaddps ymm1,...,[rsi + rax * 4 + 32]
vaddps ymm2,...,[rsi + rax * 4 + 64]
vaddps ymm3,...,[rsi + rax * 4 + 96]
add    rax, 32
cmp    eax,edi
js 1b
```

AVX SIMD vectorization



## 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)
- `pmmmintrin.h` (SSE2)
- `immintrin.h` (AVX)
  
- `x86intrin.h` (all instruction set extensions)
- See next slide for an example

# Example: array summation using C intrinsics (SSE, single precision)

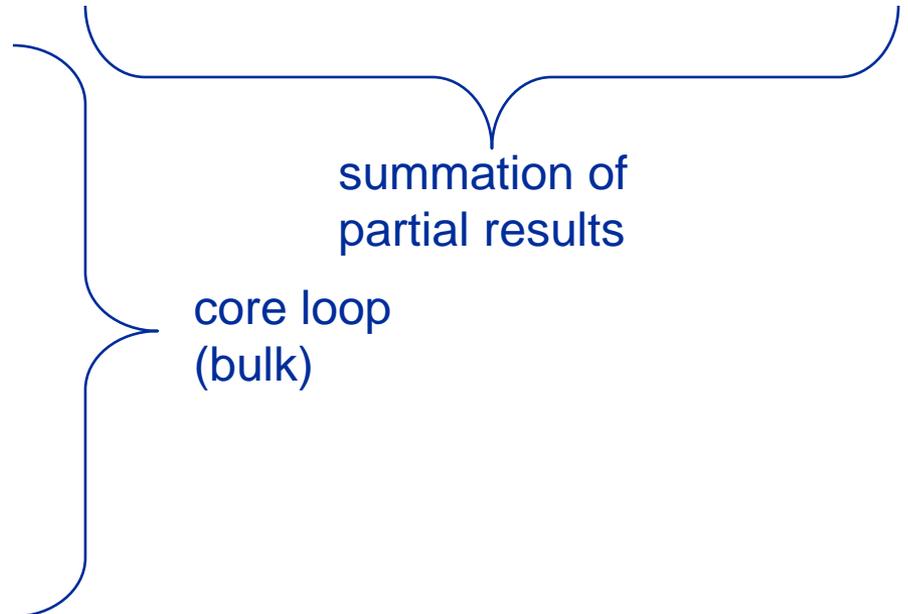


```
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = _mm_setzero_ps();
sum1 = _mm_setzero_ps();
sum2 = _mm_setzero_ps();
sum3 = _mm_setzero_ps();
```

```
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);
}
```

```
sum0 = _mm_add_ps(sum0, sum1);
sum0 = _mm_add_ps(sum0, sum2);
sum0 = _mm_add_ps(sum0, sum3);
sum0 = _mm_hadd_ps(sum0, sum0);
sum0 = _mm_hadd_ps(sum0, sum0);

_mm_store_ss(&scalar_sum, sum0);
```



# Example: array summation from intrinsics, instruction code



```
14: 0f 57 c9      xorps  %xmm1,%xmm1
17: 31 c0         xor    %eax,%eax
19: 0f 28 d1      movaps %xmm1,%xmm2
1c: 0f 28 c1      movaps %xmm1,%xmm0
1f: 0f 28 d9      movaps %xmm1,%xmm3
22: 66 0f 1f 44 00 00  nopw  0x0(%rax,%rax,1)
28: 0f 10 3e      movups (%rsi),%xmm7
2b: 0f 10 76 10   movups 0x10(%rsi),%xmm6
2f: 0f 10 6e 20   movups 0x20(%rsi),%xmm5
33: 0f 10 66 30   movups 0x30(%rsi),%xmm4
37: 83 c0 10      add    $0x10,%eax
3a: 48 83 c6 40   add    $0x40,%rsi
3e: 0f 58 df      addps  %xmm7,%xmm3
41: 0f 58 c6      addps  %xmm6,%xmm0
44: 0f 58 d5      addps  %xmm5,%xmm2
47: 0f 58 cc      addps  %xmm4,%xmm1
4a: 39 c7         cmp    %eax,%edi
4c: 77 da         ja     28 <compute_sum_SSE+0x18>
4e: 0f 58 c3      addps  %xmm3,%xmm0
51: 0f 58 c2      addps  %xmm2,%xmm0
54: 0f 58 c1      addps  %xmm1,%xmm0
57: f2 0f 7c c0   haddps %xmm0,%xmm0
5b: f2 0f 7c c0   haddps %xmm0,%xmm0
5f: c3           retq
```

Loop body



- **Intel compiler will try to use SIMD instructions when enabled to do so**

- “Poor man’s vector computing”
- Compiler can emit messages about vectorized loops (not by default):

```
plain.c(11): (col. 9) remark: LOOP WAS VECTORIZED.
```

- Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
  - Some obstructions will prevent the compiler from applying vectorization even if it is possible
- **You can use `source code directives` to provide more information to the compiler**



- **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`** on Sandy Bridge 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.



- **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



1. **Countable**
2. **Single entry and single exit**
3. **Straight line code**
4. **No function calls (exception intrinsic math functions)**

## **Better performance with:**

1. **Simple inner loops with unit stride**
2. **Minimize indirect addressing**
3. **Align data structures (SSE 16 bytes, AVX 32 bytes)**
4. **In C use the restrict keyword for pointers to rule out aliasing**

## **Obstacles for vectorization:**

- **Non-contiguous memory access**
- **Data dependencies**



- 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!)



- Since Intel Compiler 12.0 the **simd pragma** is available
- **#pragma simd** enforces vectorization where the other pragmas fail
- **Prerequisites:**
  - 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!



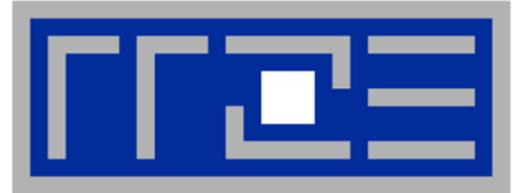
### ▪ **Alignment issues**

- 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);
```



# **Efficient parallel programming on ccNUMA nodes**

**Performance characteristics of ccNUMA nodes**

**First touch placement policy**

**C++ issues**

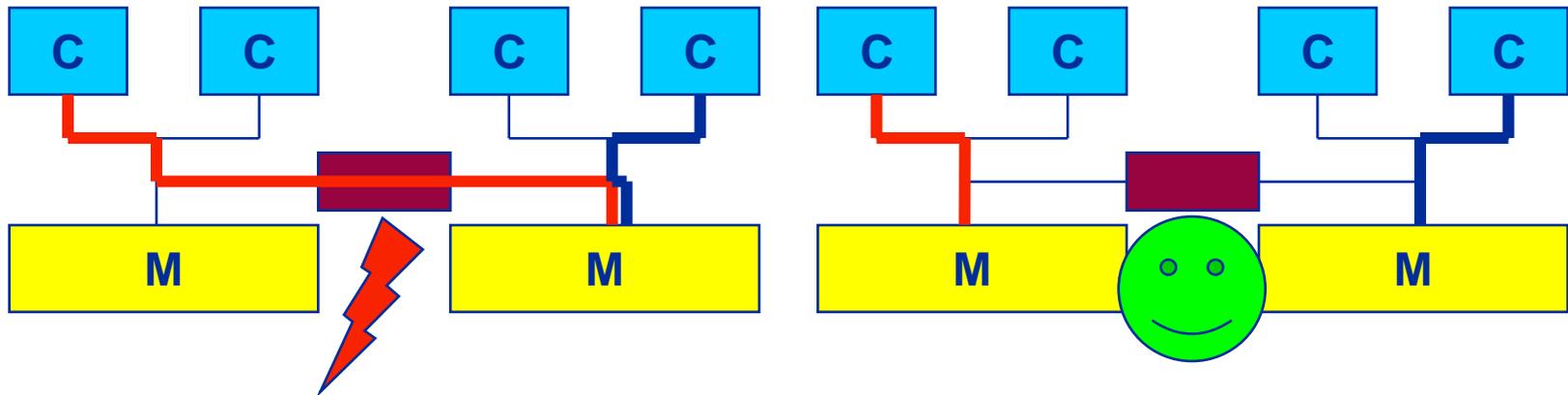
**ccNUMA locality and dynamic scheduling**

**ccNUMA locality beyond first touch**



## 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)

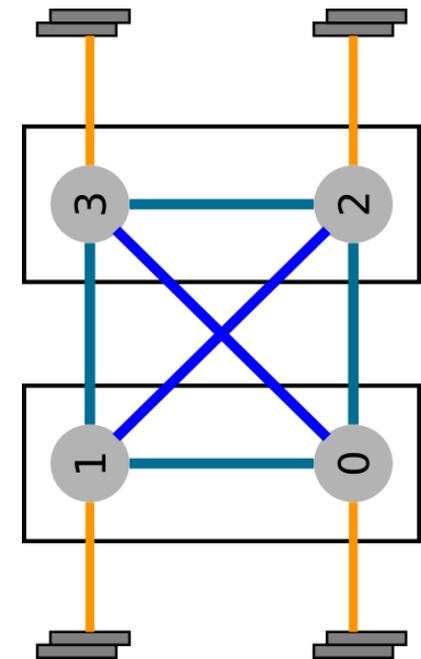
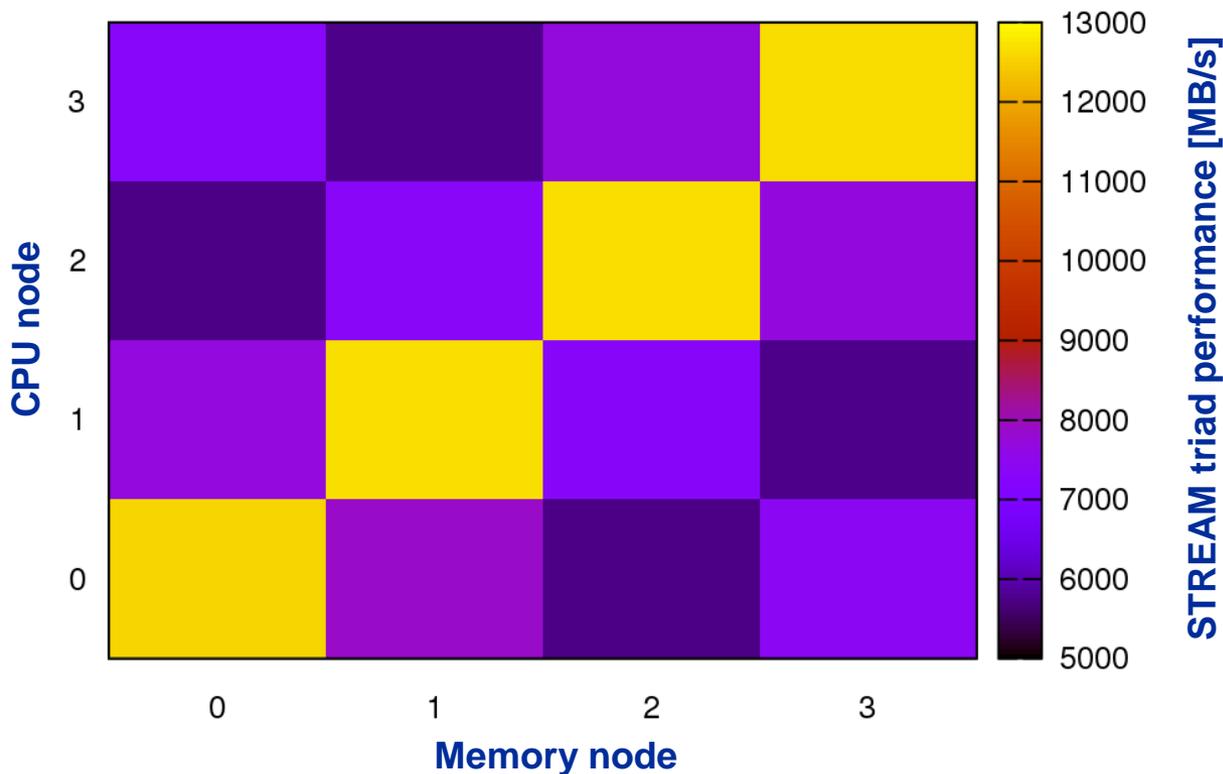
# Cray XE6 Interlagos node

4 chips, two sockets, 8 threads per ccNUMA domain



- ccNUMA map: **Bandwidth penalties for remote access**

- Run 8 threads per ccNUMA domain (1 chip)
- Place memory in different domain → 4x4 combinations
- STREAM triad benchmark using nontemporal stores





- **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?**



- "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

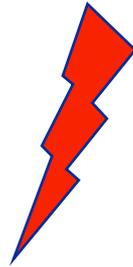
Mapping takes place here

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

- 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
```

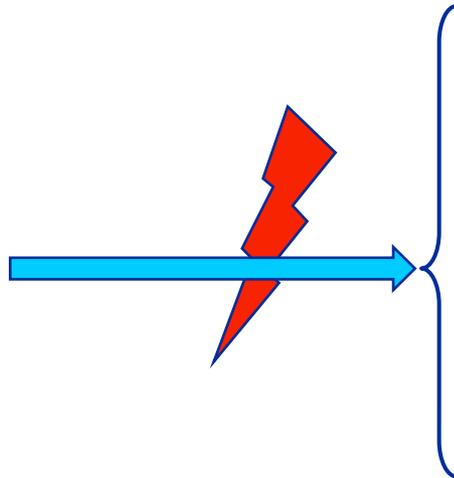


- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```
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
```





- **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



- **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



- **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<>`*



```
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)
```



- 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]	<b>4219</b>	0		<b>4163.45</b>	0	
Remote Write BW [MBytes/s]	<b>1706.19</b>	0		<b>1705.09</b>	0	
Remote BW [MBytes/s]	<b><u>5925.19</u></b>	0		<b><u>5868.54</u></b>	0	



Half of BW comes from other socket!

# If all fails...



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

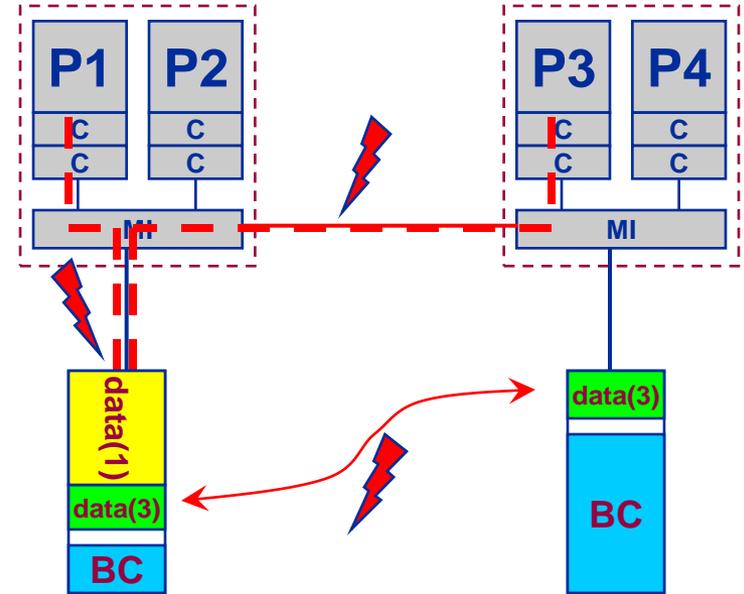
```
# numactl --hardware      # idle node!  
available: 2 nodes (0-1)  
node 0 size: 2047 MB  
node 0 free: 906 MB  
node 1 size: 1935 MB  
node 1 free: 1798 MB
```

```
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
```



- **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



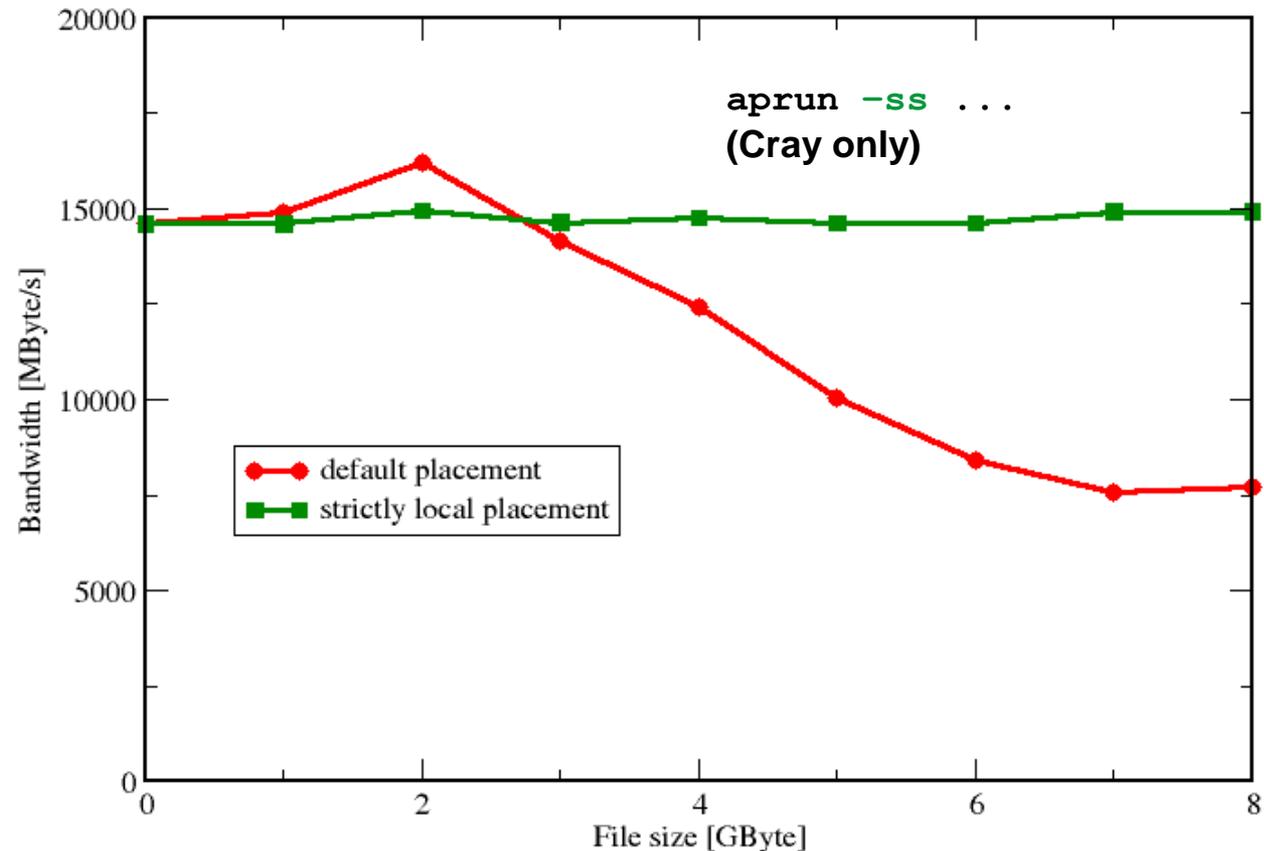
## Real-world example: ccNUMA and the Linux buffer cache

### 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!**





- Sometimes access patterns are just not nicely grouped into contiguous chunks:

```
double precision :: r, a(M)
!$OMP parallel do private(r)
do i=1,N
  call RANDOM_NUMBER(r)
  ind = int(r * M) + 1
  res(i) = res(i) + a(ind)
enddo
!OMP end parallel do
```

- Or you have to use tasking/dynamic scheduling:

```
!$OMP parallel
!$OMP single
do i=1,N
  call RANDOM_NUMBER(r)
  if(r.le.0.5d0) then
!$OMP task
    call do_work_with(p(i))
!$OMP end task
  endif
enddo
!$OMP end single
!$OMP end parallel
```

- In both cases page placement cannot easily be fixed for perfect parallel access



- **Worth a try: Interleave memory across ccNUMA domains to get at least some parallel access**

1. Explicit placement:

```
!$OMP parallel do schedule(static,512)  
do i=1,M  
  a(i) = ...  
enddo  
!$OMP end parallel do
```

Observe page alignment of array to get proper placement!

2. Using global control via `numactl`:

```
numactl --interleave=0-3 ./a.out
```

This is for **all** memory, not just the problematic arrays!

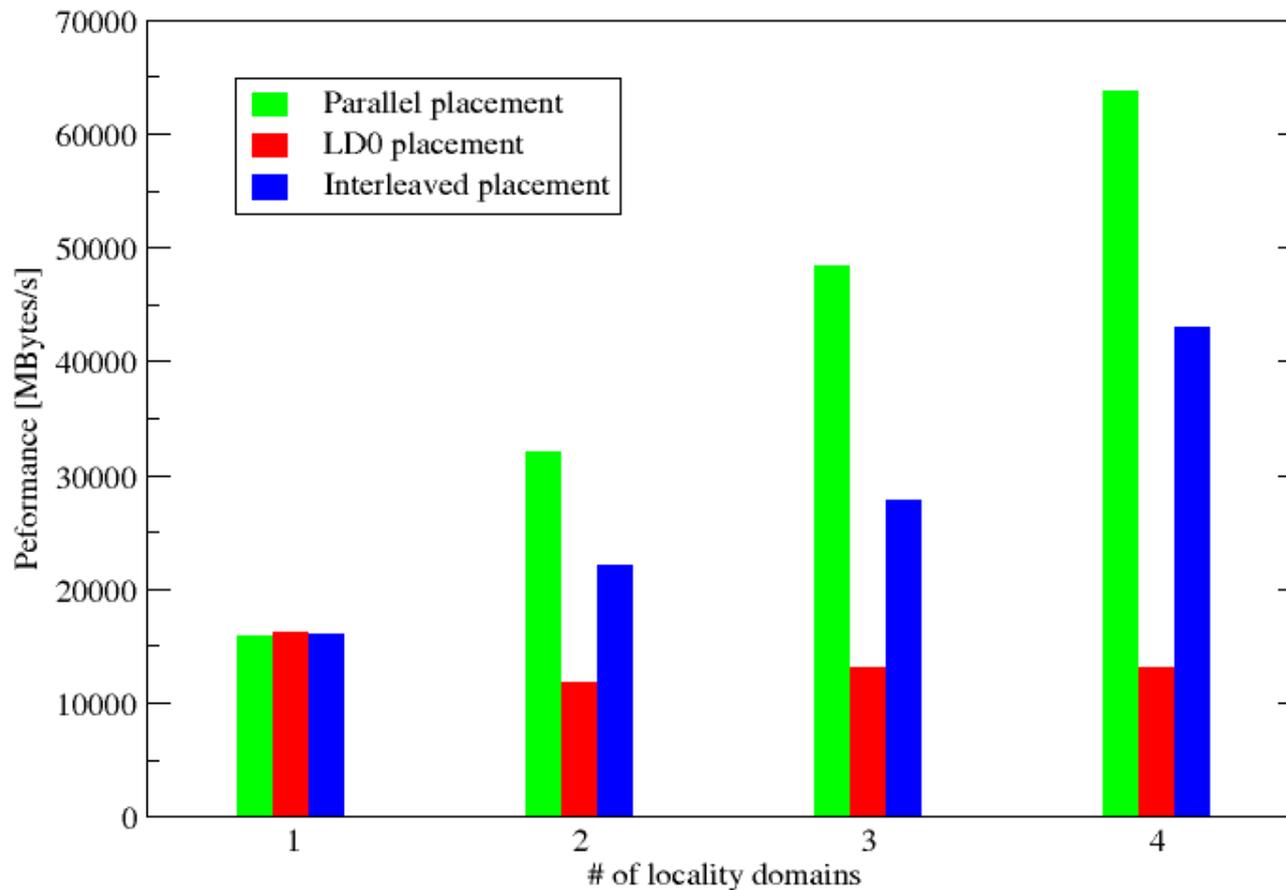
- **Fine-grained program-controlled placement via `libnuma` (Linux) using, e.g., `numa_alloc_interleaved_subset()`, `numa_alloc_interleaved()` and others**

# 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>`

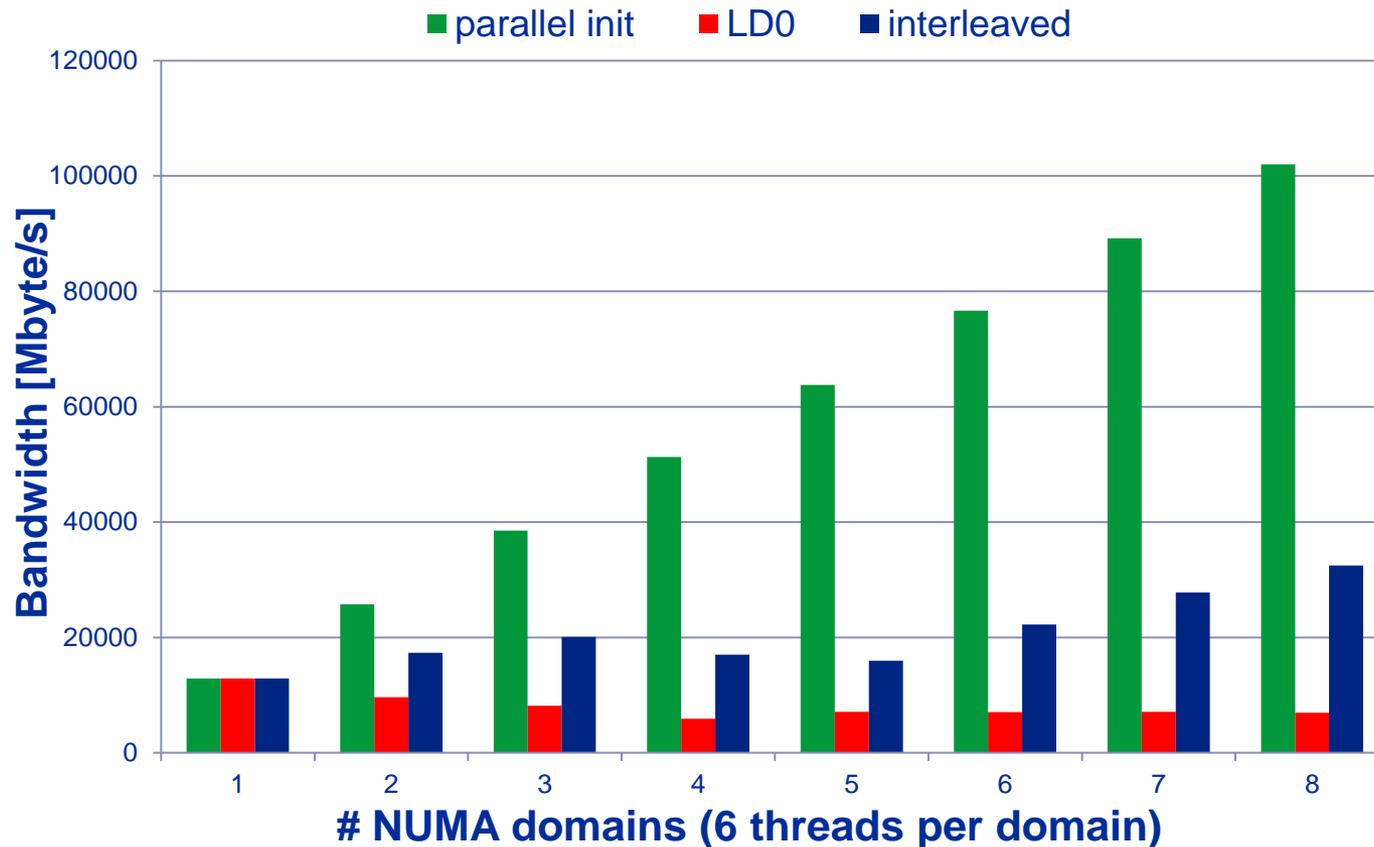


# The curse and blessing of interleaved placement:

*OpenMP STREAM triad on 4-socket (48 core) Magny Cours node*

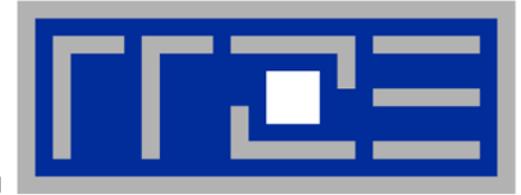


- **Parallel init:** Correct parallel initialization
- **LD0:** Force data into LD0 via `numactl -m 0`
- **Interleaved:** `numactl --interleave <LD range>`





- **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**
  - Consider round-robin placement
- **Buffer cache may impact proper placement**
  - Kick your admins
  - or apply sweeper code
  - If available, use runtime options to force local placement



# **Simultaneous multithreading (SMT)**

**Principles and performance impact**

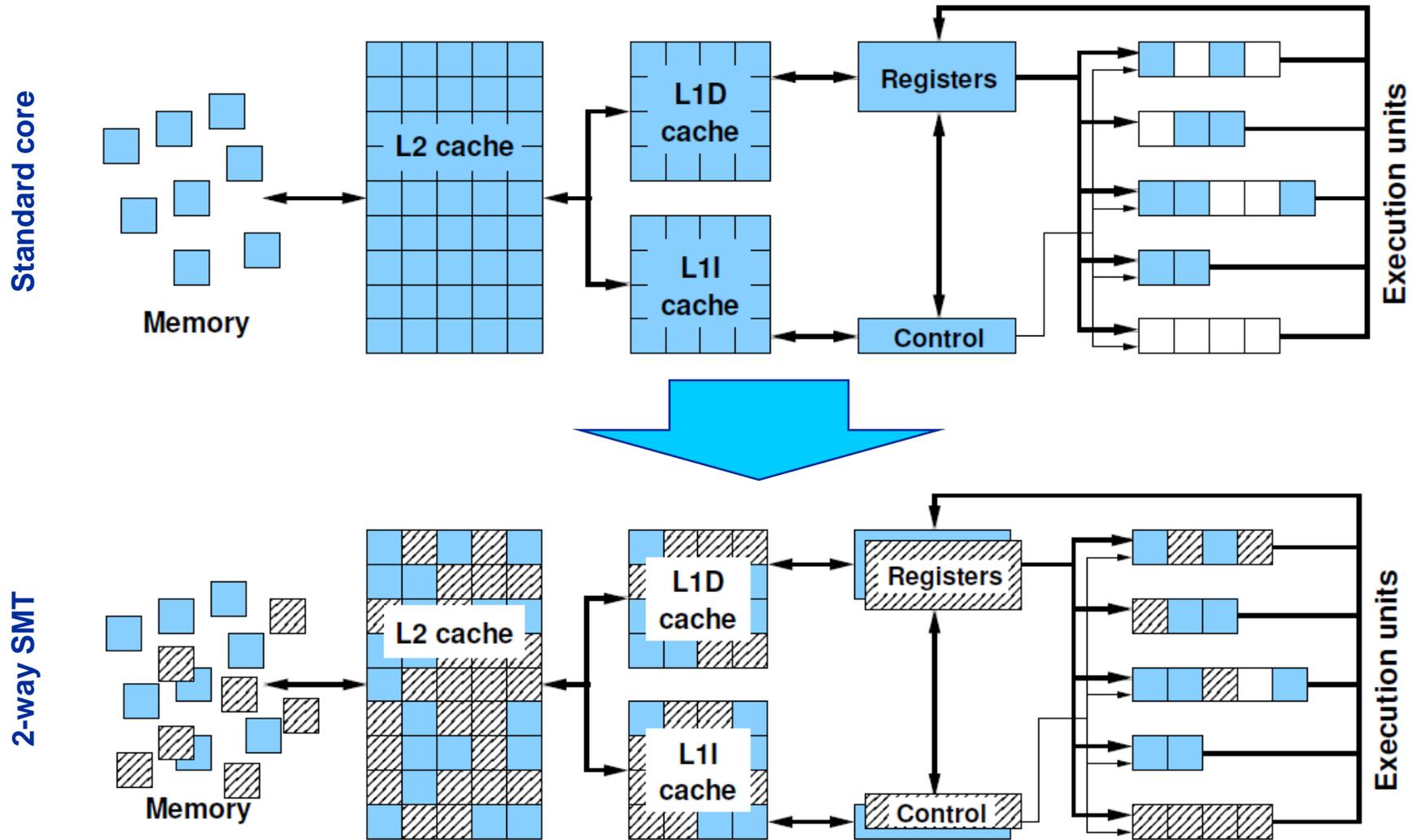
**SMT vs. independent instruction streams**

**Facts and fiction**

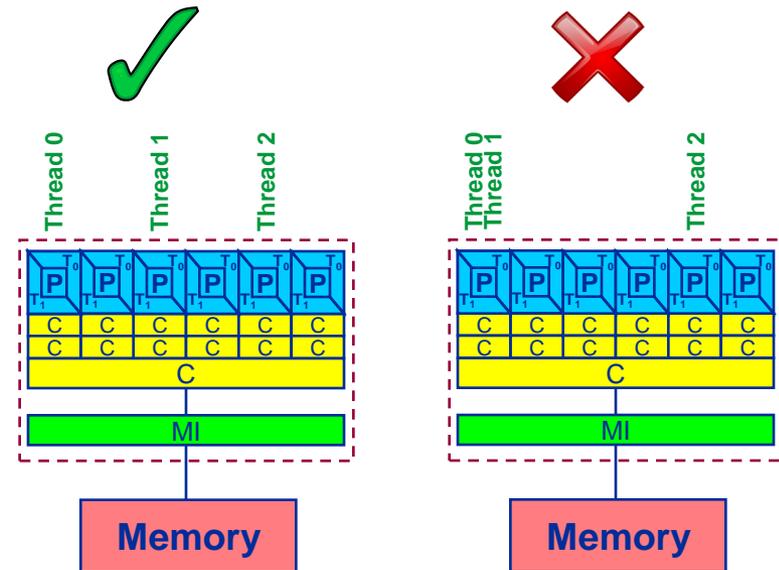
# SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently



## ■ SMT principle (2-way example):



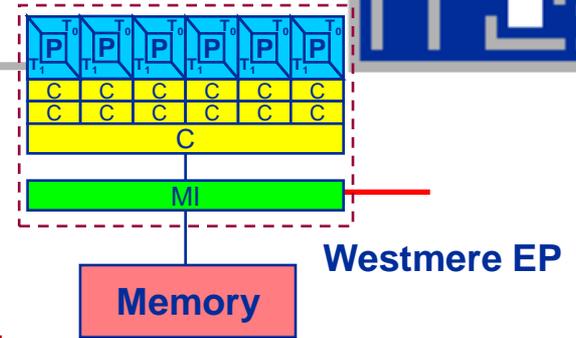
- **SMT is primarily suited for increasing processor throughput**
  - With multiple threads/processes running concurrently
- **Scientific codes tend to utilize chip resources quite well**
  - Standard optimizations (loop fusion, blocking, ...)
  - High data and instruction-level parallelism
  - Exceptions do exist
- **SMT is an important topology issue**
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - **Affinity matters!**
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.



# 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
```

Dependency → pipeline stalls until previous MULT is over

## Thread 1:

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

Unrelated work in other thread can fill the pipeline bubbles

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

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

# Simultaneous recursive updates with SMT



Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

**MULT Pipeline depth: 5 stages** → 1 F / 5 cycles for recursive update

Fill bubbles via:

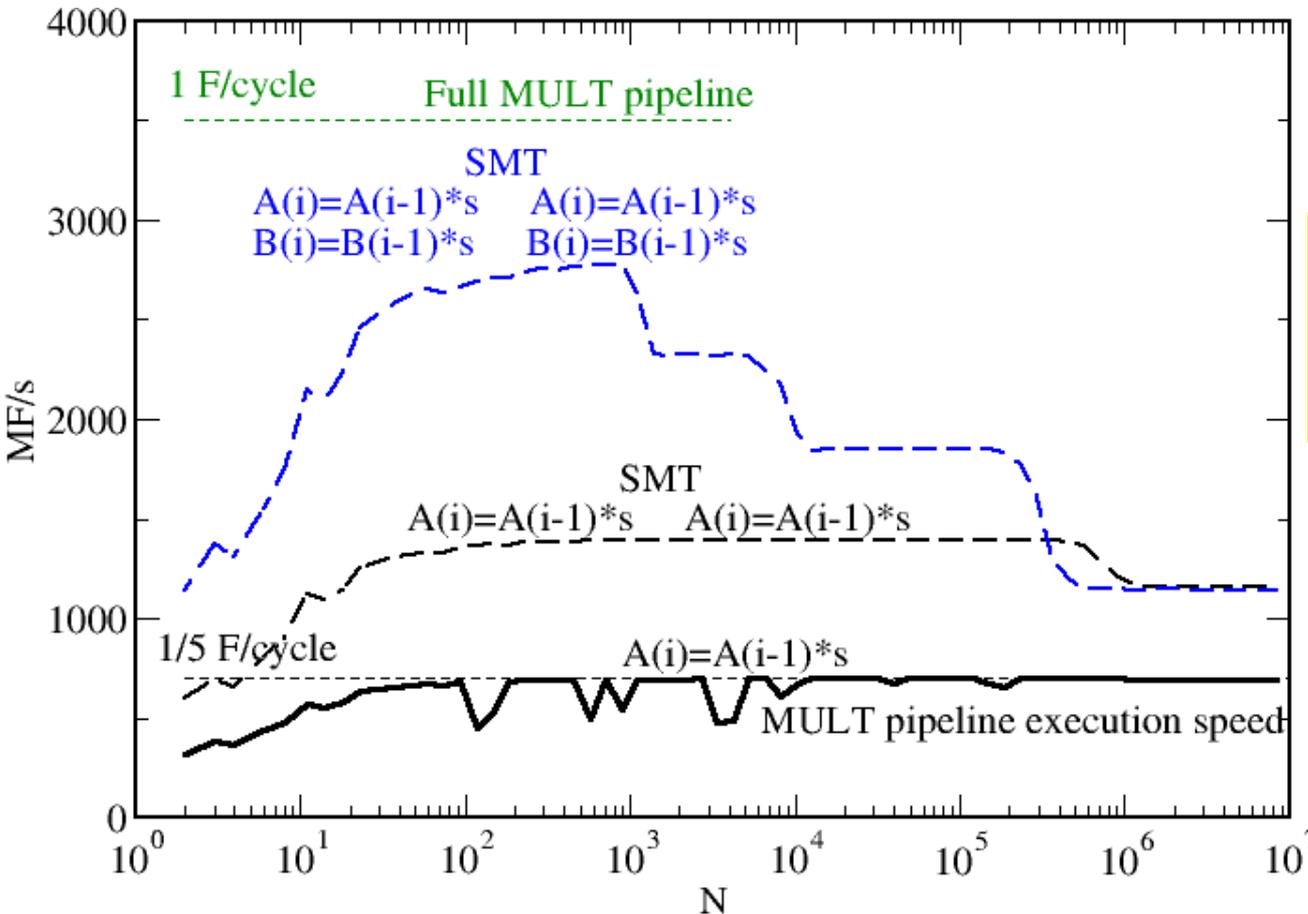
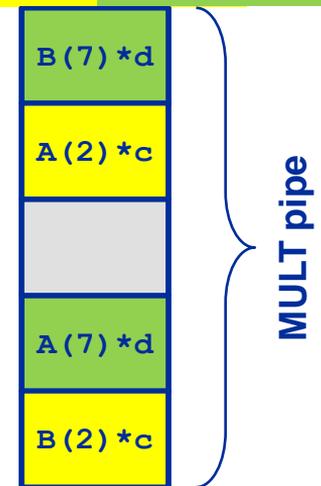
- SMT
- Multiple streams

**Thread 0:**

```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```

**Thread 1:**

```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```

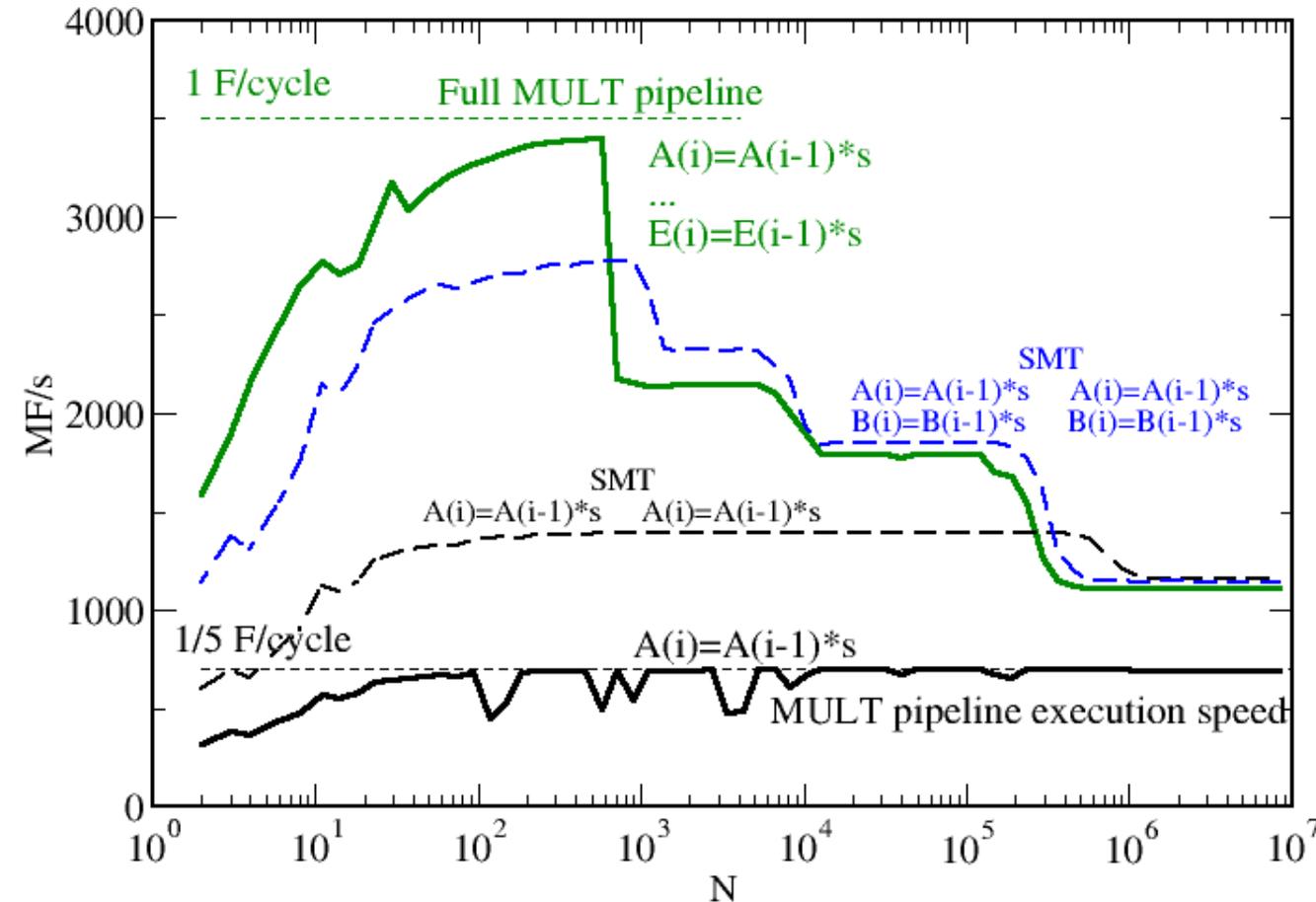


# Simultaneous recursive updates with SMT



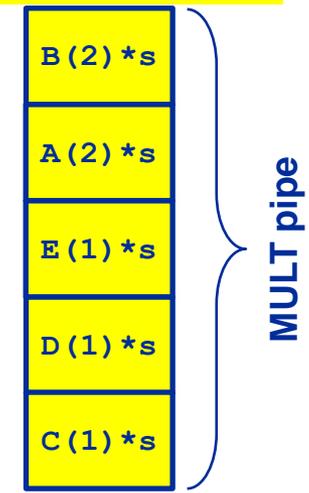
Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

**MULT Pipeline depth: 5 stages** → 1 F / 5 cycles for recursive update



```

Thread 0:
do i=1,N
  A(i)=A(i-1)*s
  B(i)=B(i-1)*s
  C(i)=C(i-1)*s
  D(i)=D(i-1)*s
  E(i)=E(i-1)*s
enddo
    
```



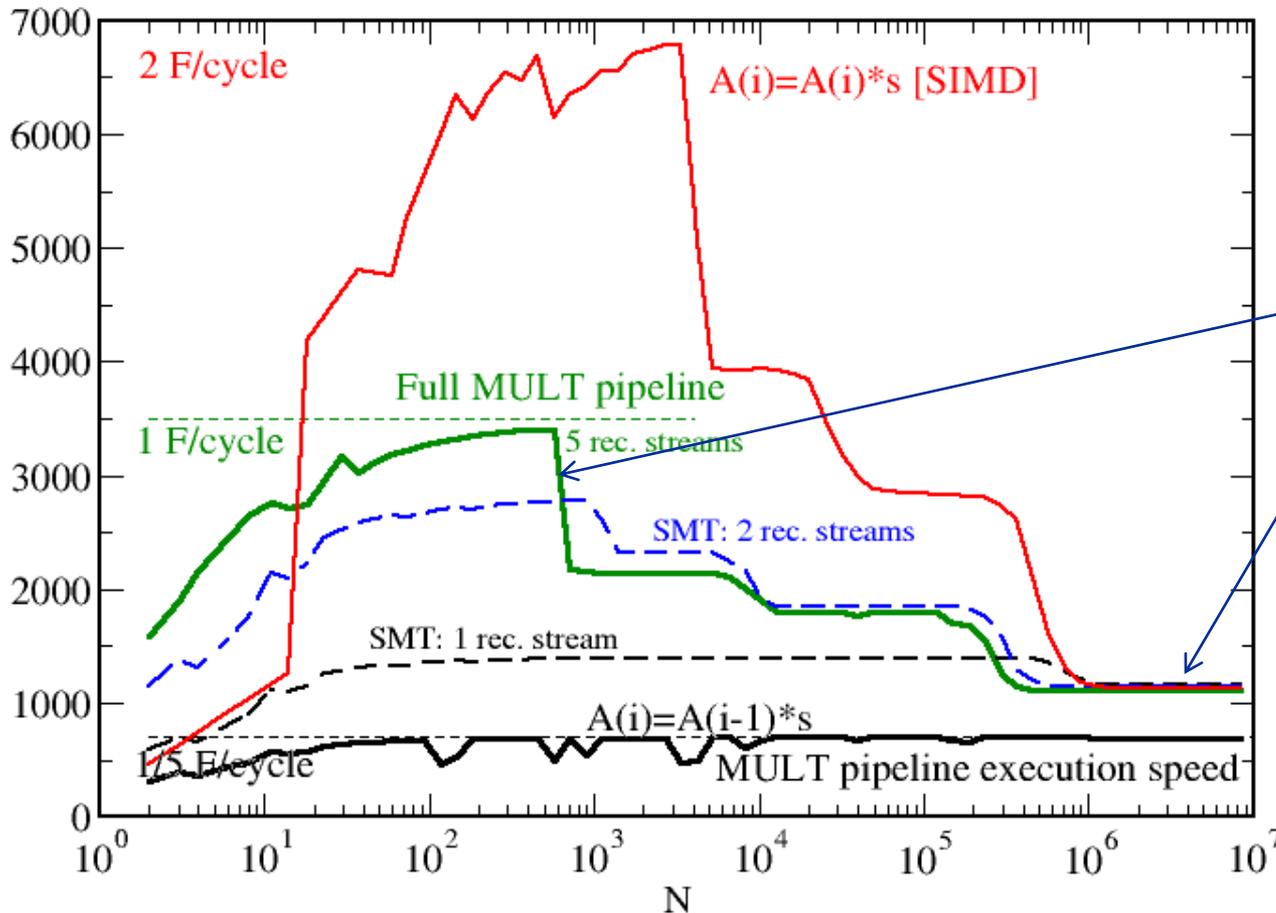
**5 independent updates on a single thread do the same job!**

# Simultaneous recursive updates with SMT



Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

Pure update benchmark can be **vectorized** → **2 F / cycle (store limited)**



## Recursive update:

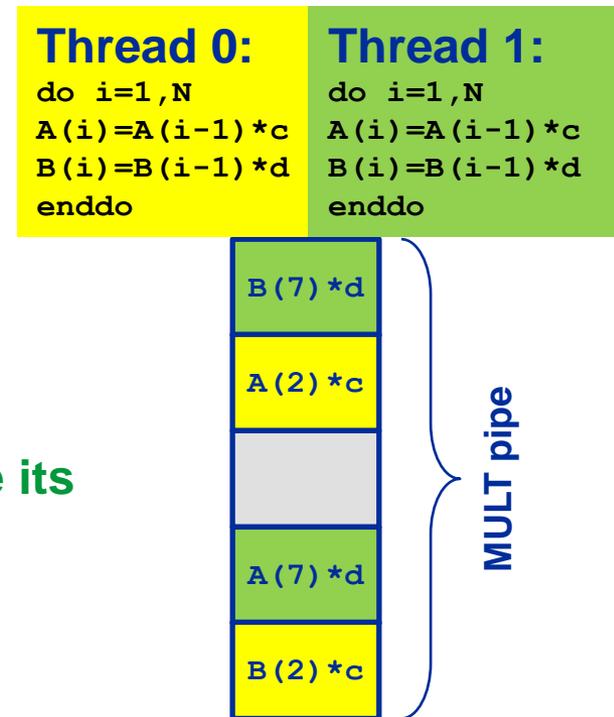
- SMT can fill pipeline bubbles
- A single thread can do so as well
- Bandwidth does not increase through SMT
- **SMT can not replace SIMD!**



- Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”

- Truth

- A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.
- If a pipeline is already full, SMT will not improve its utilization



# SMT myths: Facts and fiction (2)



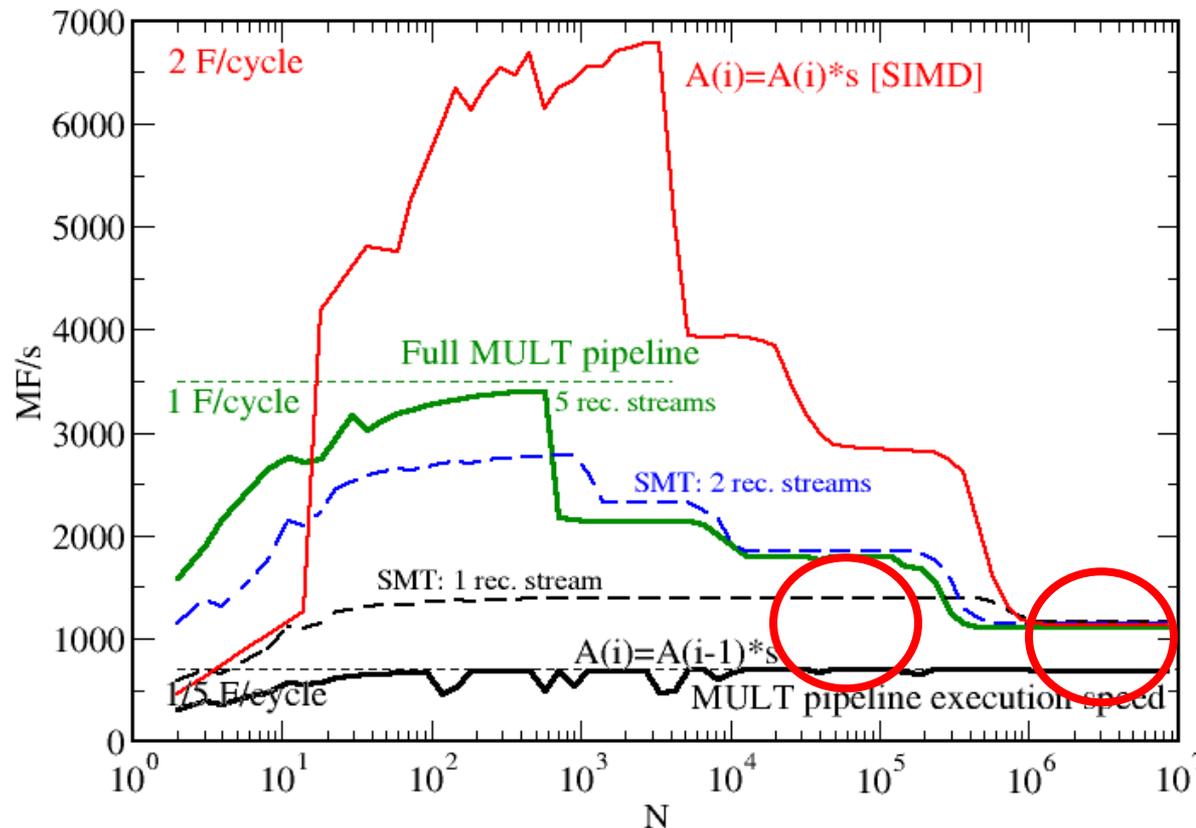
- Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

- Truth:

- If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.

- If the relevant bottleneck is not exhausted, SMT may help since it can fill bubbles in the LOAD pipeline.

This applies also to other “relevant bottlenecks!”

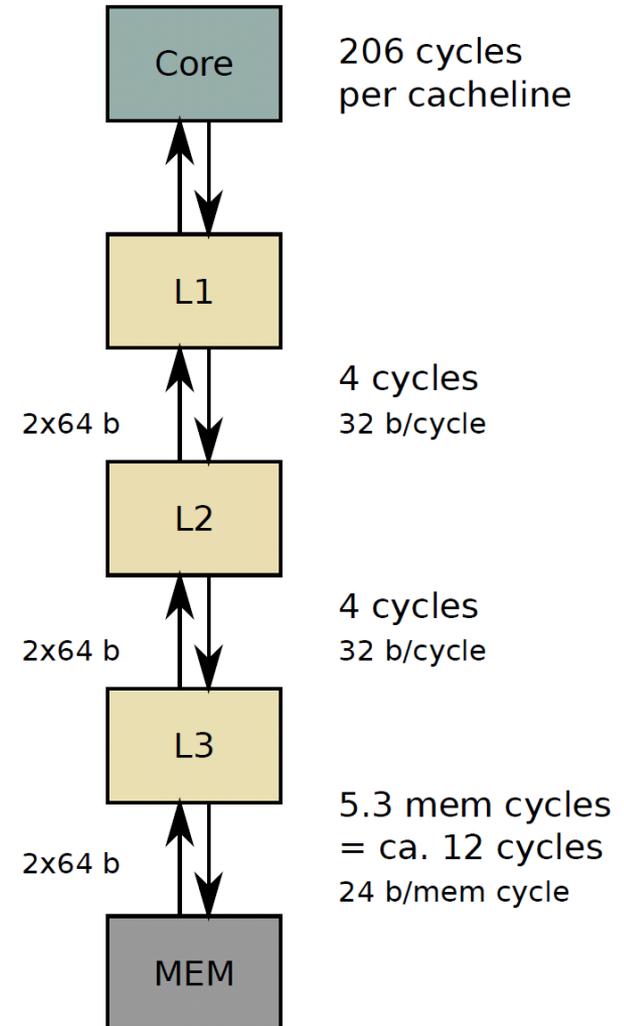


# SMT myths: Facts and fiction (3)



- **Myth:** “SMT can help bridge the latency to memory (more outstanding references).”
- **Truth:** Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets “wasted” in the cache hierarchy.

See also the “**ECM Performance Model**” later on.





### Goals for optimization:

1. **Map your work to an instruction mix with highest throughput using the most effective instructions.**
2. **Reduce data volume over slow data paths fully utilizing available bandwidth.**
3. **Avoid possible hazards/overhead which prevent reaching goals one and two.**

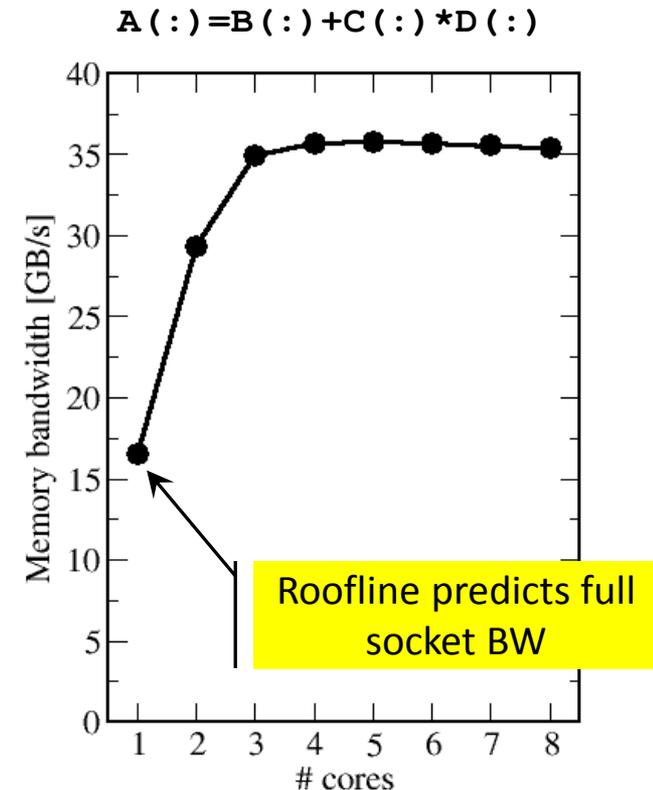


# Multicore Scaling: The ECM Model

Improving the Roofline Model



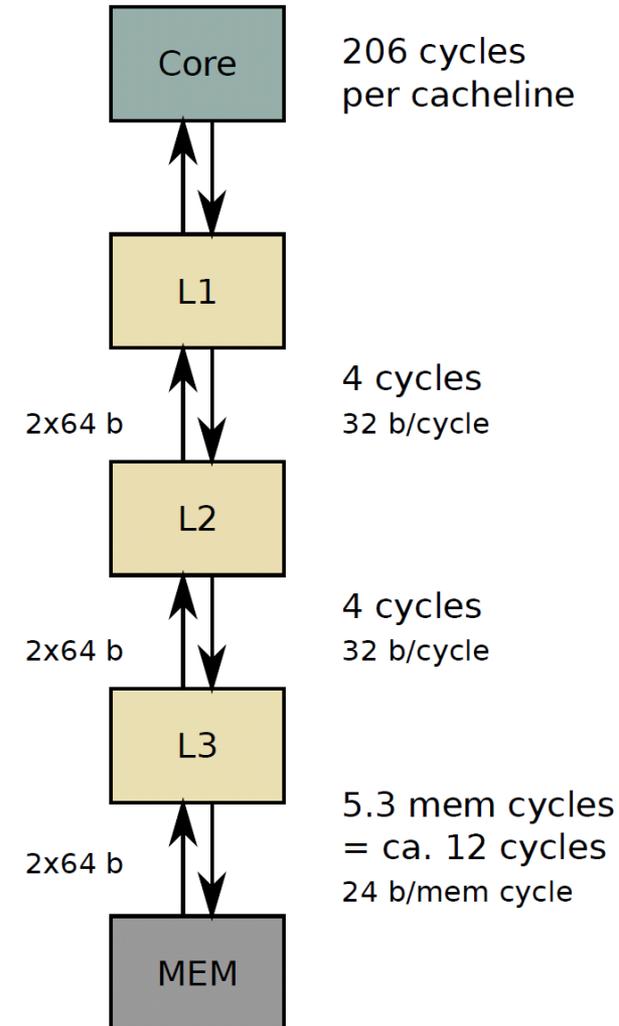
- **Assumes one of two bottlenecks**
  1. **In-core execution**
  2. **Bandwidth of a single hierarchy level**
- **Latency effects are not modeled → pure data streaming assumed**
- **In-core execution is sometimes hard to model**
- **Saturation effects in multicore chips are not explained**
  - **ECM model** gives more insight



G. Hager, J. Treibig, J. Habich, and G. Wellein: **Exploring performance and power properties of modern multicore chips via simple machine models**. *Concurrency and Computation: Practice and Experience* (2013). DOI: [10.1002/cpe.3180](https://doi.org/10.1002/cpe.3180) Preprint: [arXiv:1208.2908](https://arxiv.org/abs/1208.2908)



- **ECM = “Execution-Cache-Memory”**
- **Assumptions:**
- **Single-core execution time is composed of**
  1. In-core execution
  2. Data transfers in the memory hierarchy
- **Data transfers may or may not overlap with each other or with in-core execution**
- **Scaling is linear until the relevant bottleneck is reached**
- **Input:**
- **Same as for Roofline**
- **+ data transfer times in hierarchy**

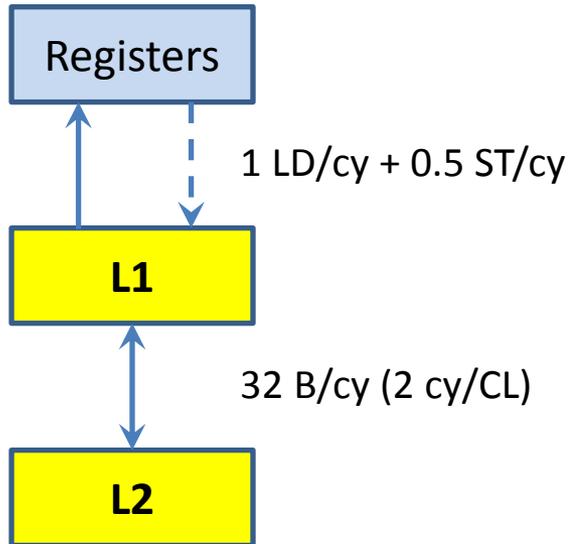


# Example: Schönauer Vector Triad in L2 cache



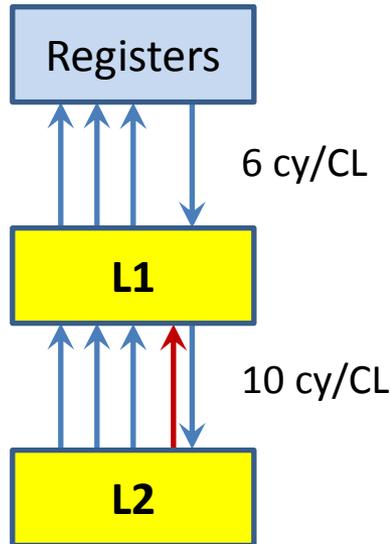
- REPEAT[  $A(:,) = B(:,) + C(:,) * D(:,)$  ] @ double precision
- Analysis for Sandy Bridge core w/ AVX (unit of work: 1 cache line)

Machine characteristics:



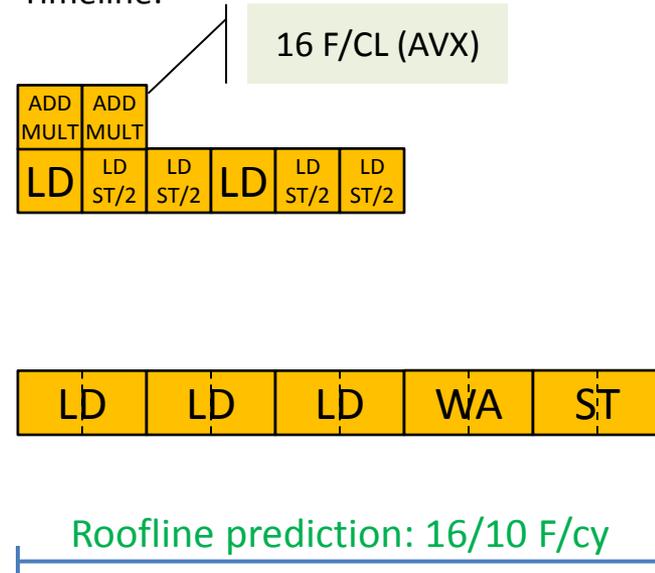
Arithmetic:  
1 ADD/cy+ 1 MULT/cy

Triad analysis (per CL):



Arithmetic:  
AVX: 2 cy/CL

Timeline:

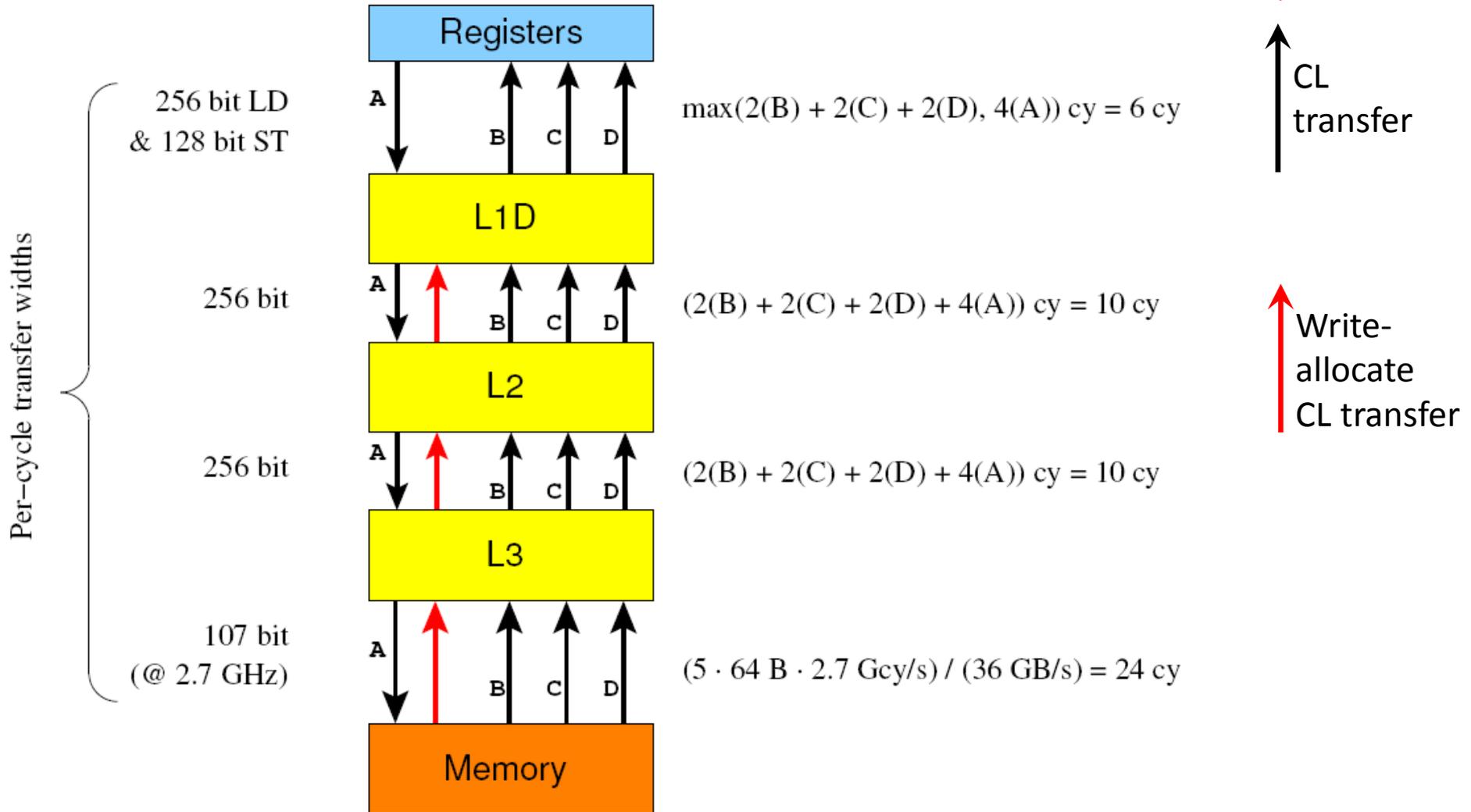


Measurement: 16F / ≈17cy

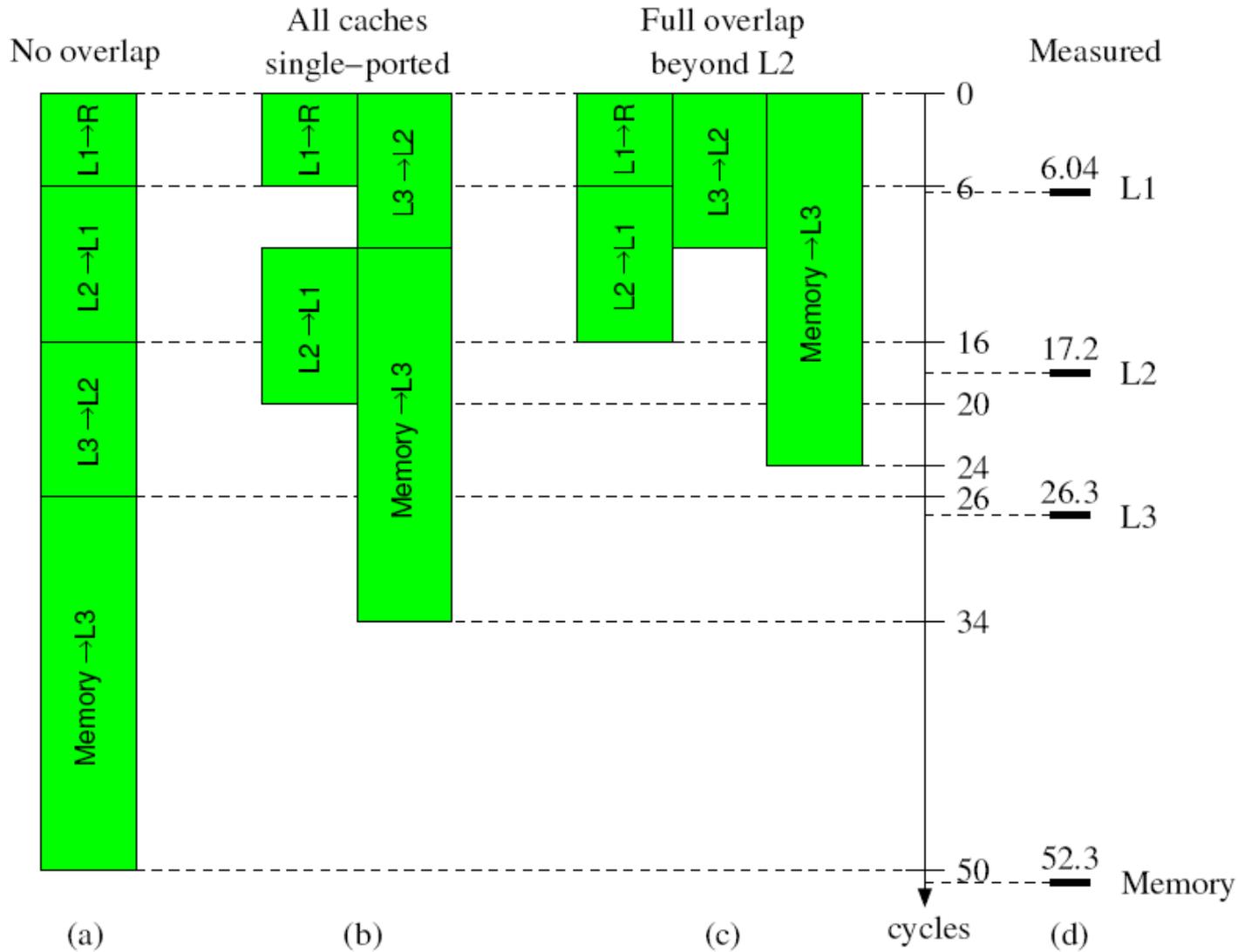


# Example: ECM model for Schönauer Vector Triad

$A(:) = B(:) + C(:) * D(:)$  on a Sandy Bridge Core with AVX



# Full vs. partial vs. no overlap



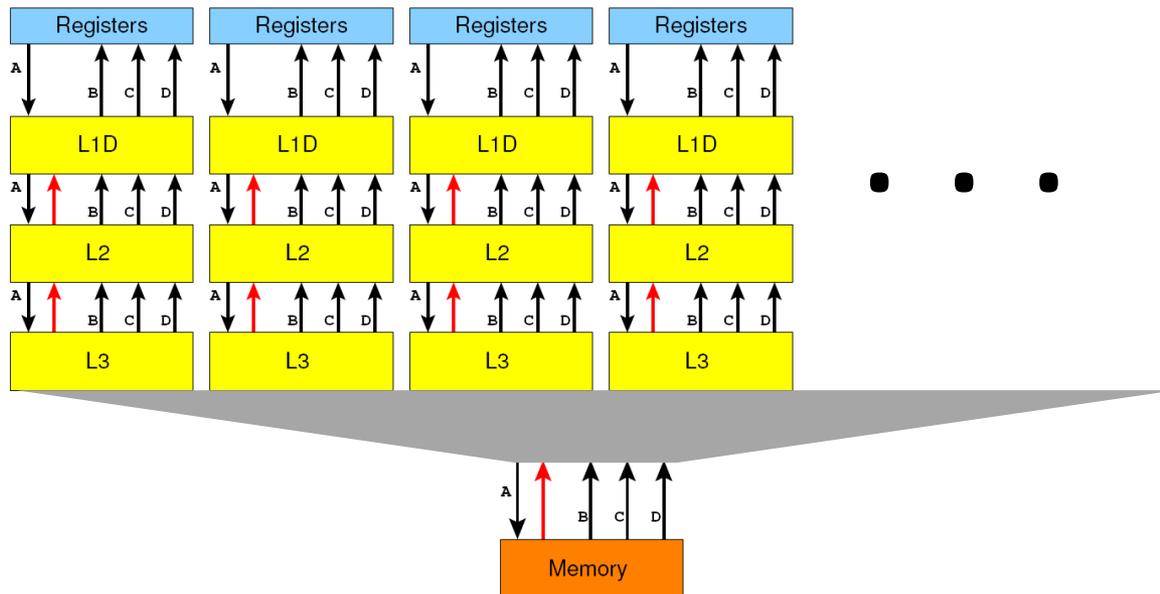
Results suggest **no overlap!**



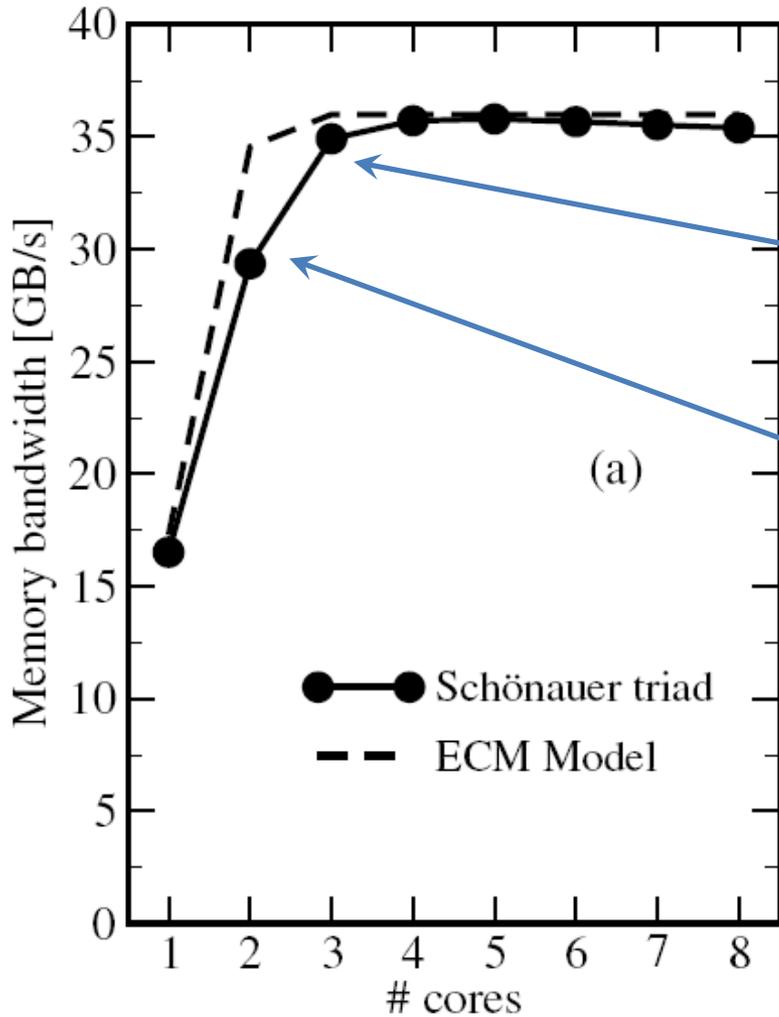
- Identify relevant **bandwidth bottlenecks**
  - L3 cache
  - Memory interface
- **Scale** single-thread performance until **first bottleneck** is hit:

$$P(n) = \min(nP_0, I \cdot b_S)$$

Example:  
Scalable L3  
on Sandy  
Bridge



# ECM prediction vs. measurements for $A(:,) = B(:,) + C(:,) * D(:,)$ on a Sandy Bridge socket (no-overlap assumption)



**Model: Scales until saturation sets in**

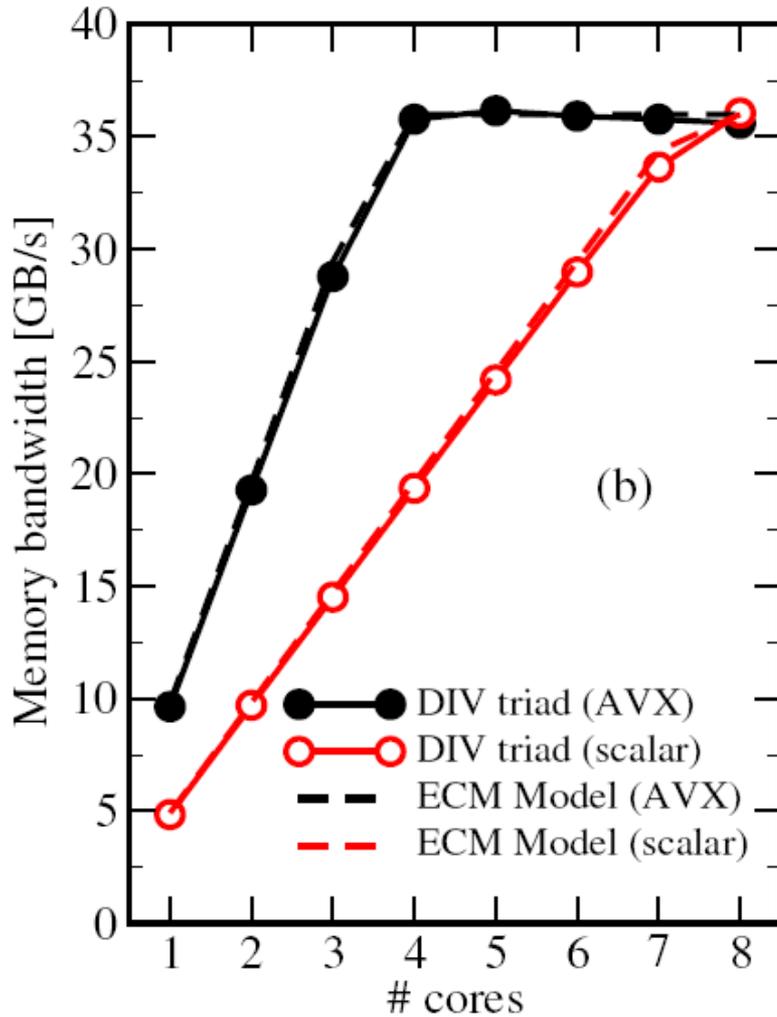
**Saturation point (# cores) well predicted**

**Measurement: scaling not perfect**

**Caveat: This is specific for this architecture and this benchmark!**

**Check: Use “overlappable” kernel code**

# ECM prediction vs. measurements for $A(:) = B(:) + C(:) / D(:)$ on a Sandy Bridge socket (full overlap assumption)



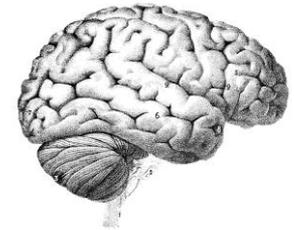
In-core execution is dominated by divide operation  
(44 cycles with AVX, 22 scalar)

→ **Almost perfect agreement with ECM model**



- **Saturation effects** are ubiquitous; understanding them gives us opportunity to
  - Find out about optimization opportunities
  - Save energy by letting cores idle → see power model later on
  - Putting idle cores to better use → asynchronous communication, functional parallelism
- **Simple models work best. Do not try to complicate things unless it is really necessary!**
- **Possible extensions to the ECM model**
  - Accommodate latency effects
  - Model simple “architectural hazards”

- **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 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



**THANK YOU.**



Bundesministerium  
für Bildung  
und Forschung

# Presenter Biographies



- **Georg Hager** holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now 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. See his blog at <http://blogs.fau.de/hager> for current activities, publications, and talks.
- **Jan 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.





- **ISC14 tutorial: Node-Level Performance Engineering**
- **Presenter(s): Georg Hager, Jan Treibig, Gerhard Wellein**

- **ABSTRACT:**

This tutorial covers performance engineering approaches on the compute node level. “Performance engineering” is more than employing tools to identify hotspots and blindly applying textbook optimizations. It is about developing a thorough understanding of the interactions between software and hardware. This process starts at the core, socket, and node level, where the code gets executed that does the actual “work.” Once the architectural requirements of a code are understood and correlated with performance measurements, the potential benefit of optimizations can often be predicted. We start by giving an overview of modern processor and node architectures, including accelerators such as GPGPUs and Xeon Phi. Typical bottlenecks such as instruction throughput and data transfers are identified using kernel benchmarks and put into the architectural context. The impact of optimizations like SIMD vectorization, ccNUMA placement, and cache blocking is shown, and different aspects of a “holistic” node-level performance engineering strategy are demonstrated. Using the LIKWID multicore tools we show the importance of topology awareness, affinity enforcement, and hardware metrics. The latter are used to support the performance engineering process by supplying information that can validate or falsify performance models.



## Books:

- G. Hager and G. Wellein: Introduction to High Performance Computing for Scientists and Engineers. CRC Computational Science Series, 2010. ISBN 978-1439811924

## Papers:

- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Concurrency and Computation: Practice and Experience (2013).  
DOI: [10.1002/cpe.3180](https://doi.org/10.1002/cpe.3180) Preprint: [arXiv:1208.2908](https://arxiv.org/abs/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. Preprint: [arXiv:1206.3738](https://arxiv.org/abs/1206.3738)
- 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](https://doi.org/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](https://doi.org/10.1177/1094342012442424)



Papers continued:

- 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](https://doi.org/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](https://doi.org/10.1142/S0129626410000296). Preprint: [arXiv:1006.3148](https://arxiv.org/abs/1006.3148)
- J. Treibig, G. Hager and G. Wellein: **LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments**. Proc. [PSTI2010](https://www.psti2010.org/), the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010.  
[DOI: 10.1109/ICPPW.2010.38](https://doi.org/10.1109/ICPPW.2010.38). Preprint: [arXiv:1004.4431](https://arxiv.org/abs/1004.4431)
- 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](https://doi.org/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](https://doi.org/10.1016/j.jocs.2011.01.010)



Papers continued:

- 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](https://doi.org/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](https://doi.org/10.1007/978-3-642-13872-0_1), Preprint: [arXiv:0910.4865](https://arxiv.org/abs/0910.4865).
- 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](https://doi.org/10.1177/1094342003017001005)