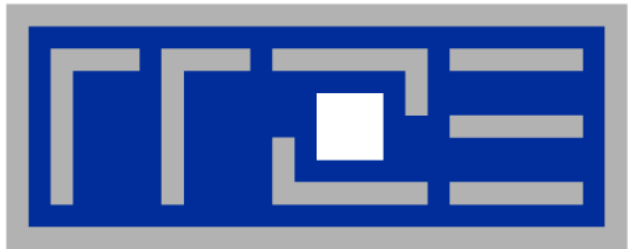


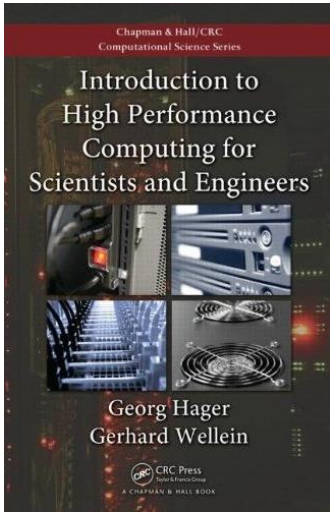
Node-Level Performance Engineering

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

PPAM 2013 tutorial
September 8, 2013
Warsaw, Poland

Slides

- **Where can I find these *gorgeous* slides?**
- **Other information?**
  - http://blogs.fau.de/hager
- **Anything else?**



ISBN 978-1439811924

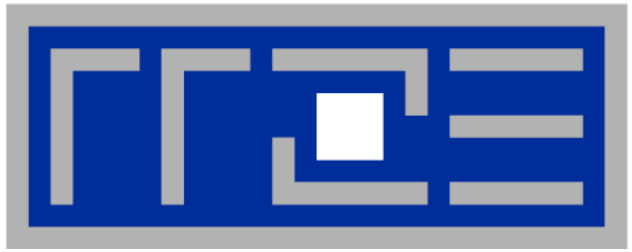
http://goo.gl/eocNpa

Agenda

- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA

Agenda

- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA



Prelude:
Scalability 4 teh win!

Scalability Myth: Code scalability is the key issue

Lore 1

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

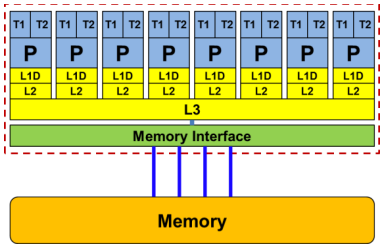
Lore 2

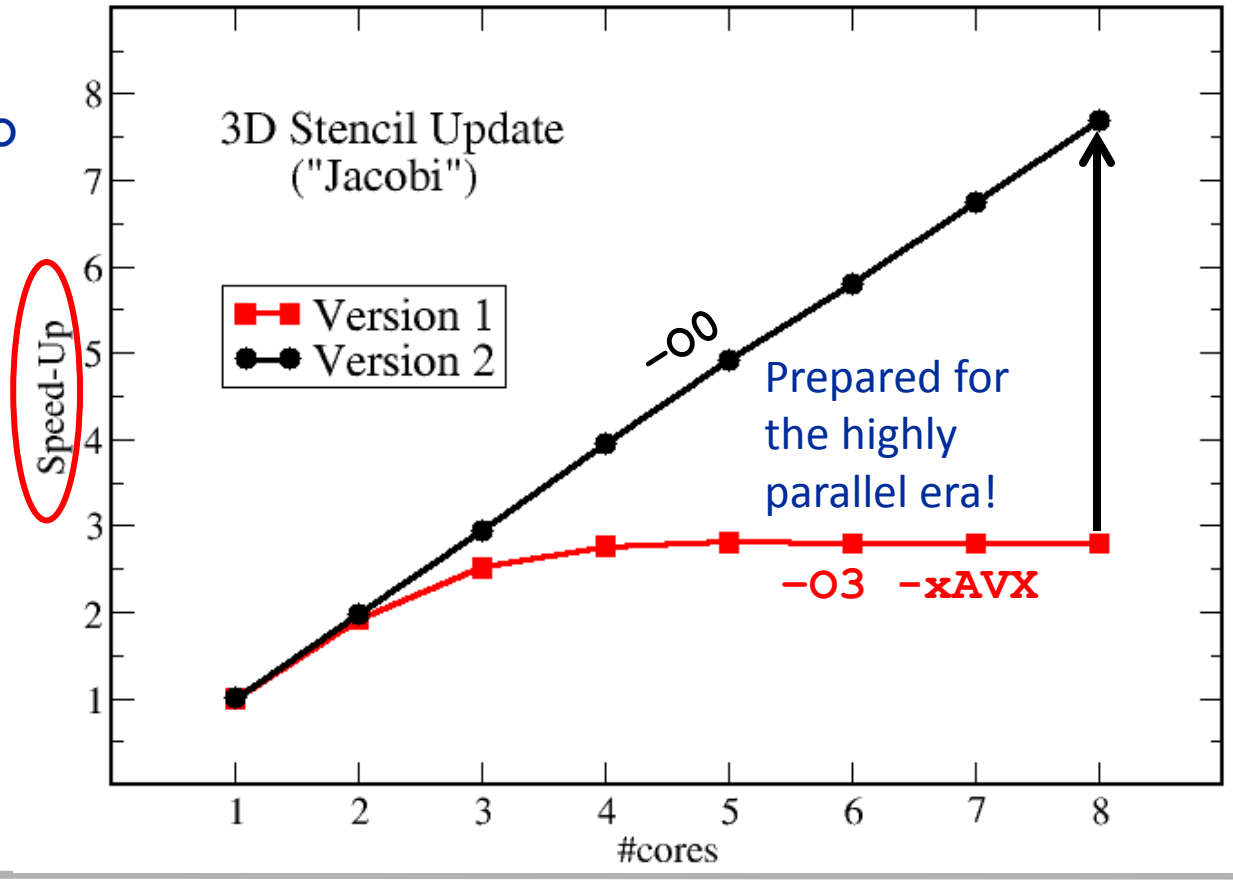
Single core performance no longer matters since we have so many of them and use scalable codes

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

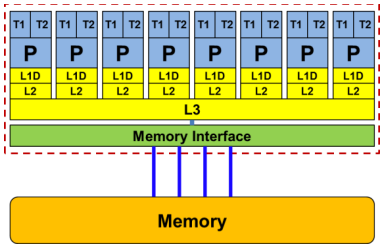


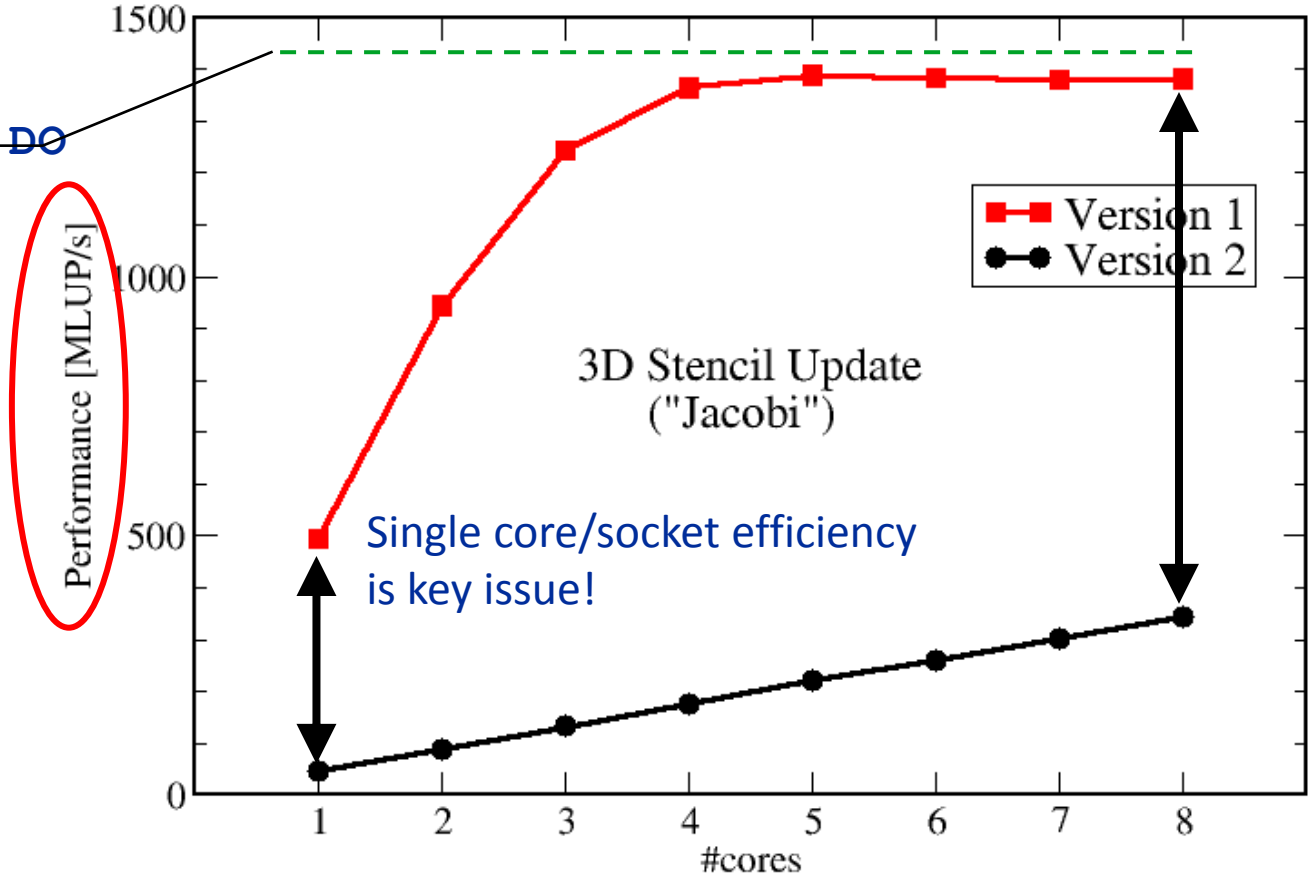


Scalability Myth: Code scalability is the key issue

```
!$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





Questions to ask in high performance computing

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

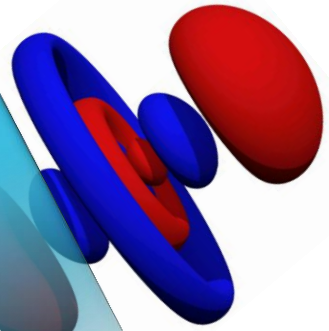
How model-building works: Physics

Newtonian mechanics

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

Fails @ small scales!

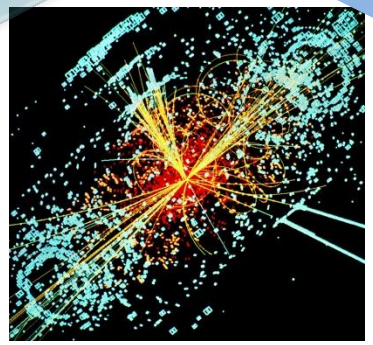
Nonrelativistic quantum mechanics



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

Fails @ even smaller scales!

If a model fails, we learn something!



Relativistic quantum field theory

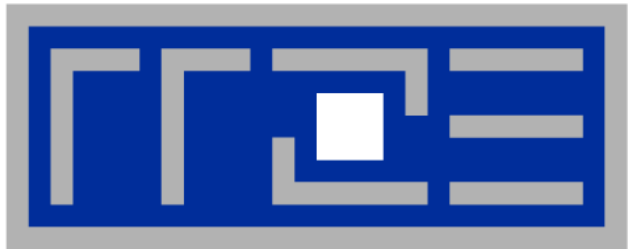
$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_c$$

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

Agenda

- Preliminaries
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- LIKWID tools
- Microbenchmarking for architectural exploration
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- Node-level performance modeling
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- Optimal resource utilization
  - SIMD parallelism
  - ccNUMA



Introduction:
Modern node architecture

Multi- and manycore chips and nodes

A glance at basic core fatures

Caches and data transfers through the memory hierarchy

Memory organization

Accelerators

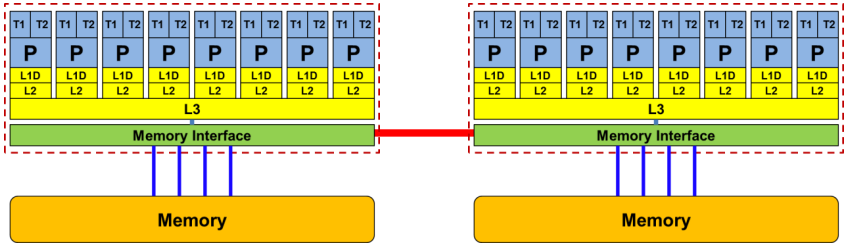
Multi-Core: Intel Xeon 2600 (2012)

- **Xeon 2600 “Sandy Bridge EP”:**
  8 cores running at 2.7 GHz (max 3.2 GHz)

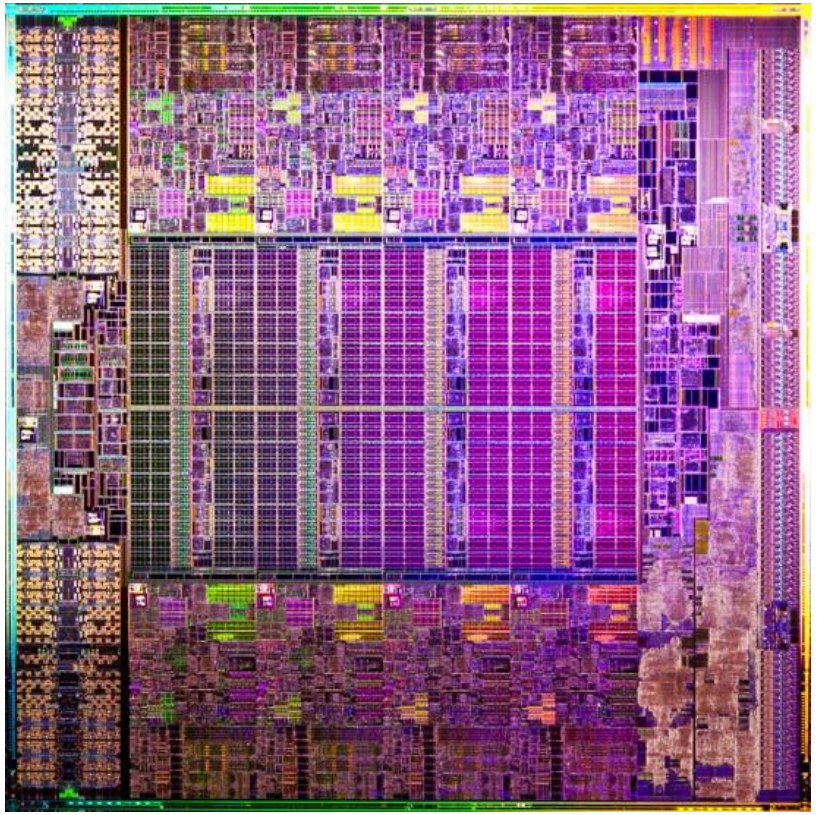
- **Simultaneous Multithreading**
  → reports as 16-way chip

- **2.3 Billion Transistors / 32 nm**

- **Die size: 435 mm²**

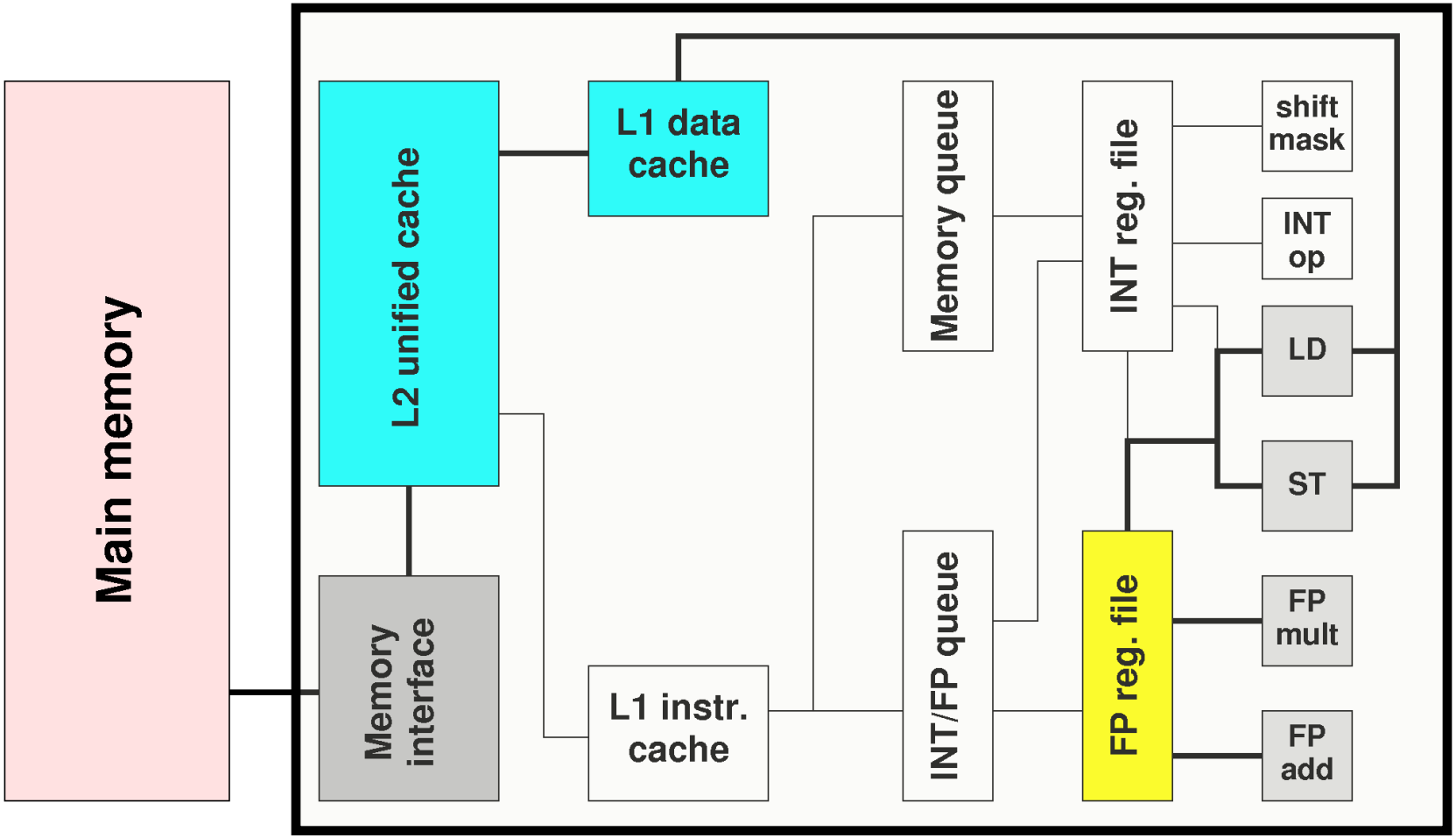


2-socket server



General-purpose cache based microprocessor core

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



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

Pipelining of arithmetic/functional units

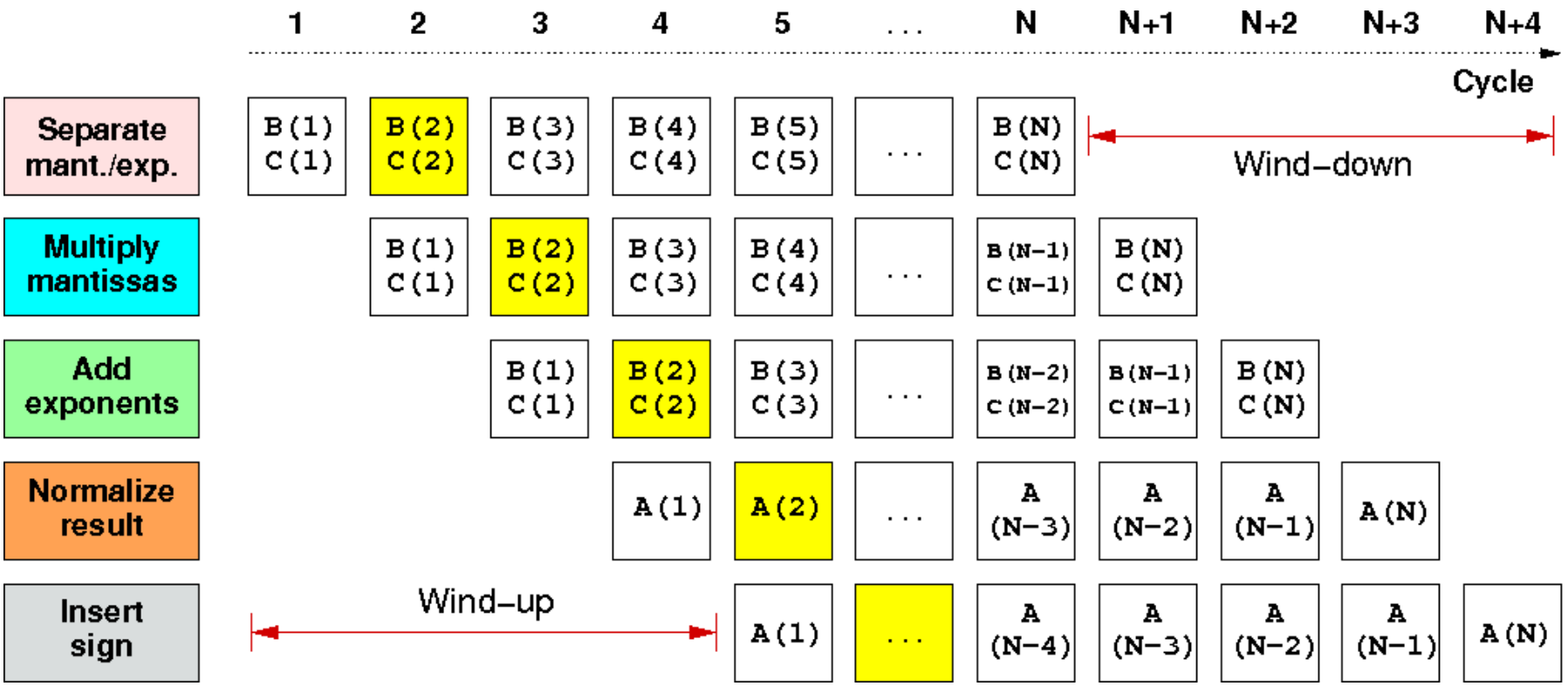
- **Idea:**
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

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

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

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

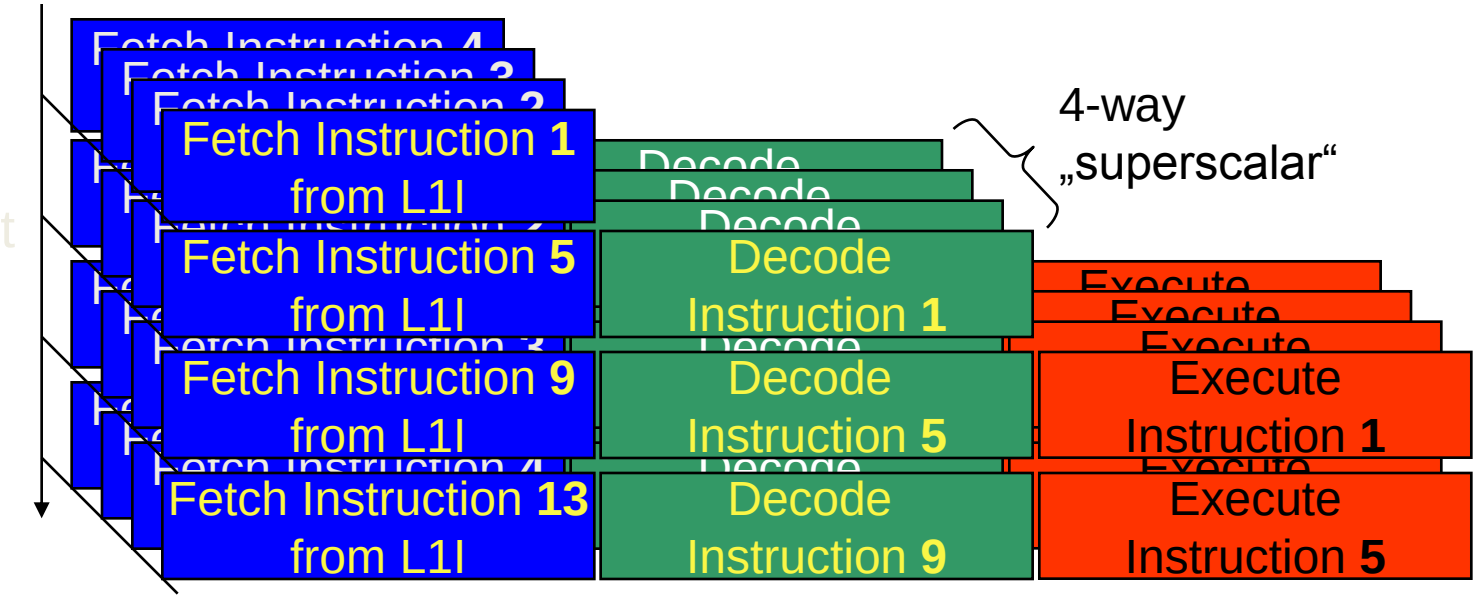
5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N



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

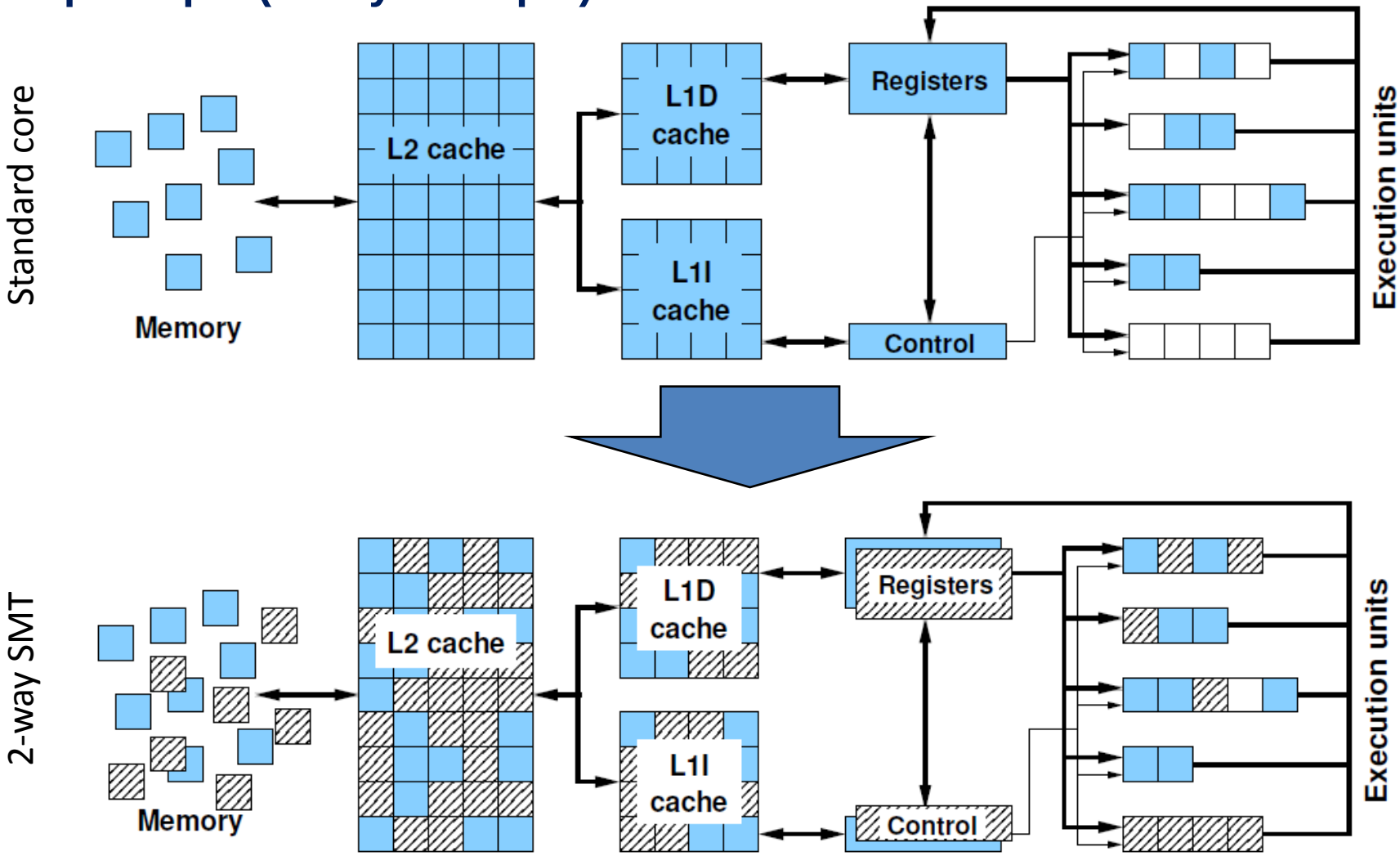
Wind-up/-down phases: Empty pipeline stages

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

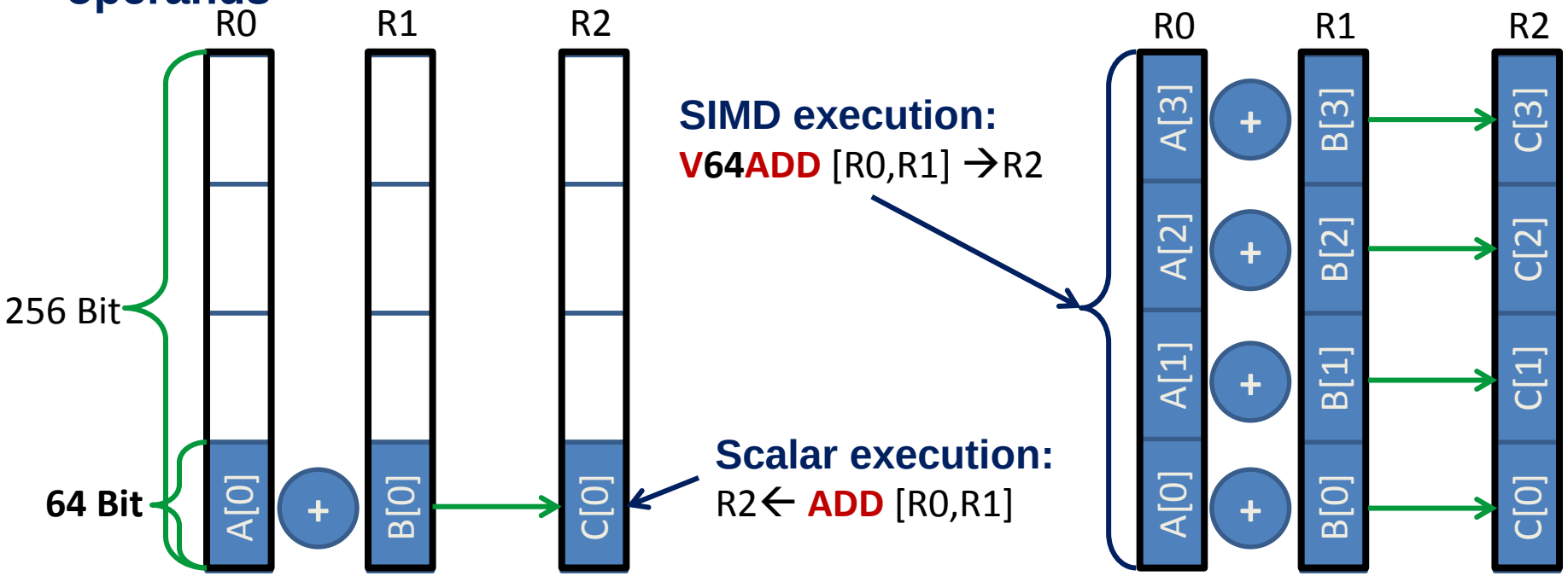


- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycles

SMT principle (2-way example):



- **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**

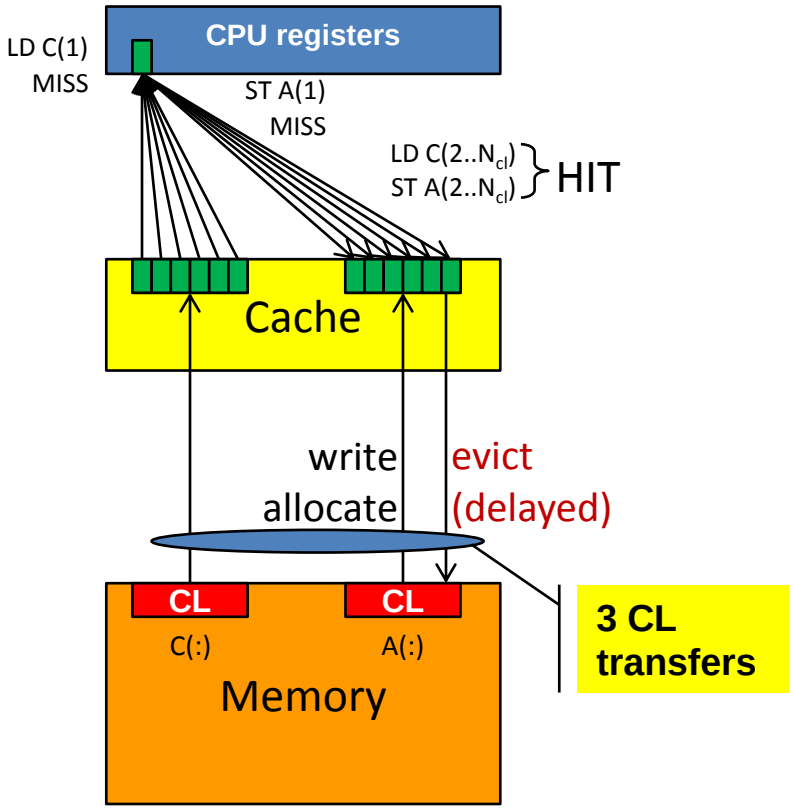


Registers and caches: Data transfers in a memory hierarchy

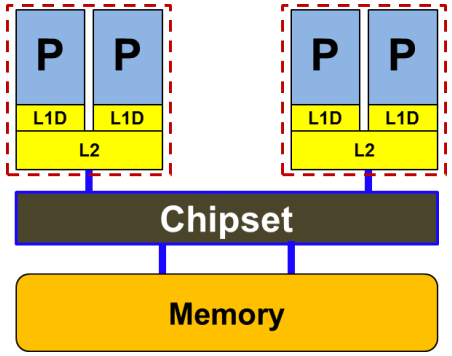
- How does data travel from memory to the CPU and back?

- 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(:)`



Yesterday (2006): Dual-socket Intel "Core2" node:

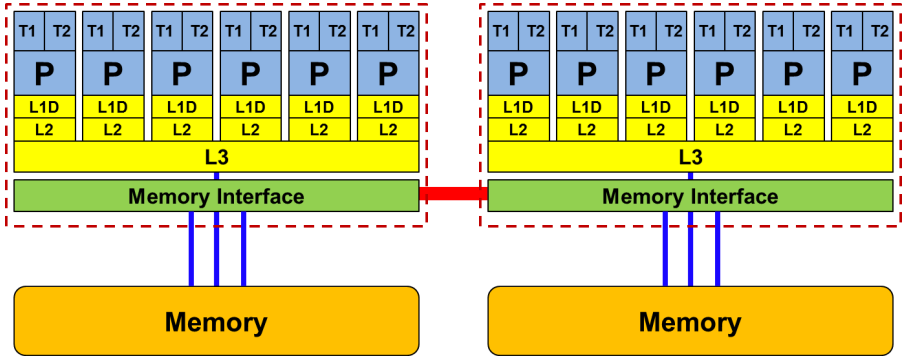


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system "anisotropy"

Today: Dual-socket Intel (Westmere,…) node:



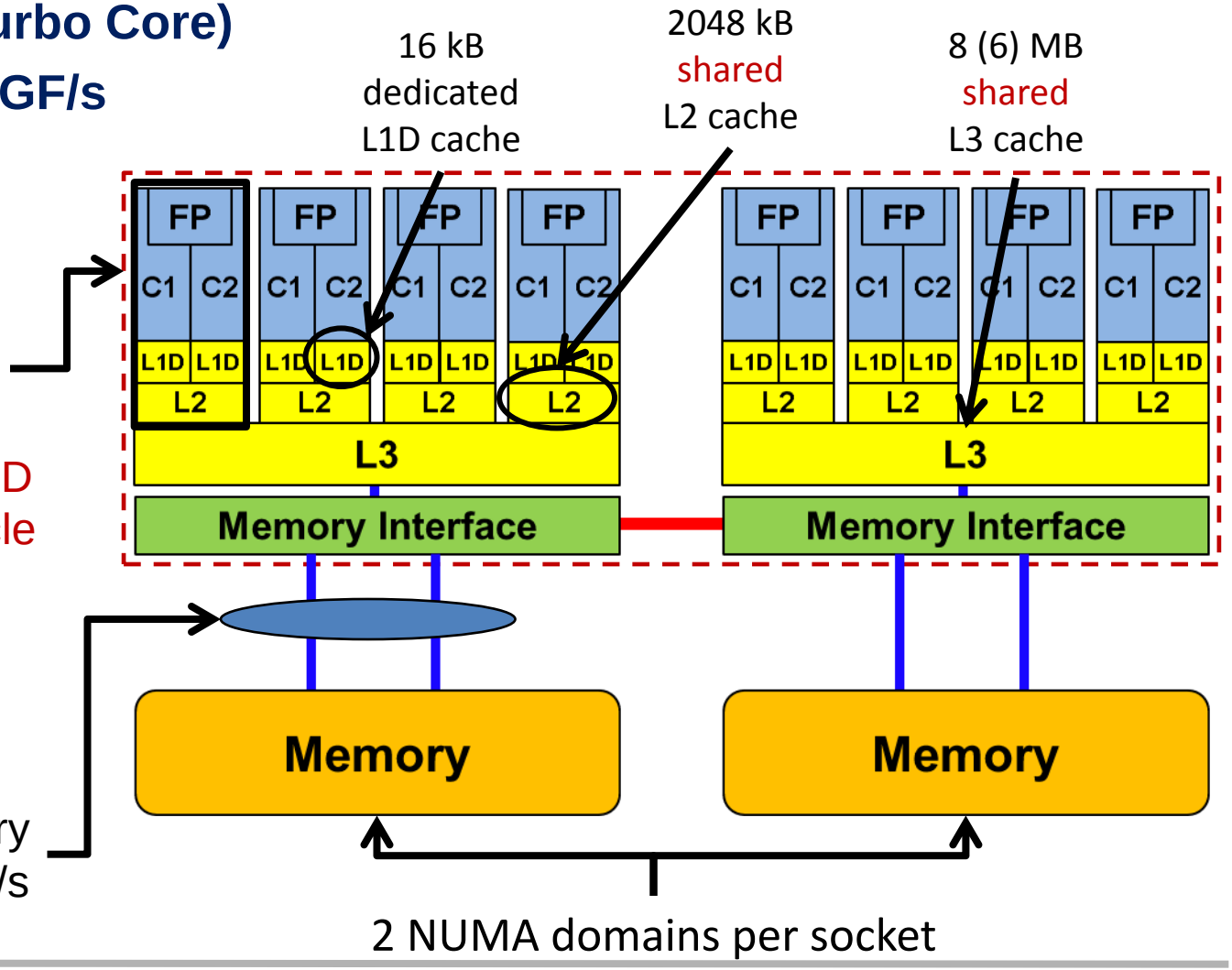
Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

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

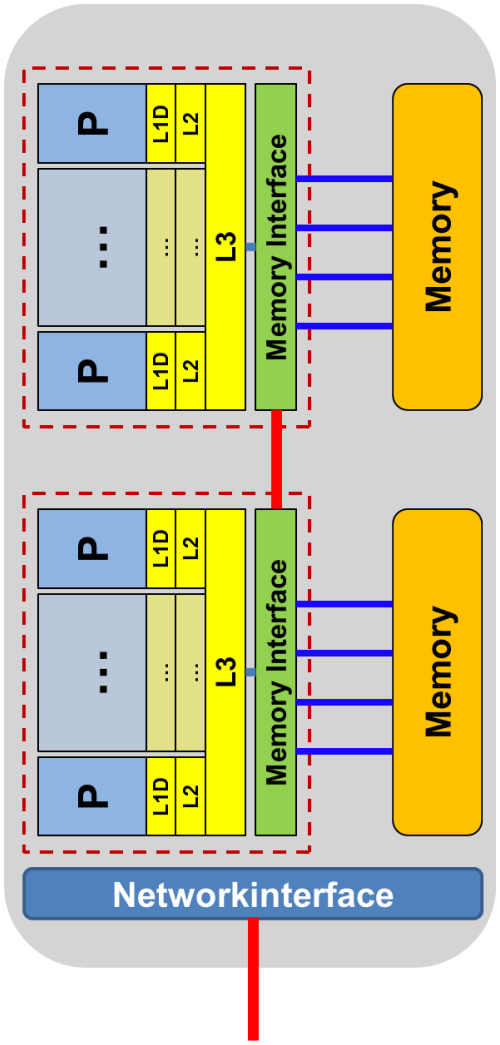
On AMD it is even more complicated → ccNUMA within a socket!

- **Up to 16 cores (8 Bulldozer modules) in a single socket**
- **Max. 2.6 GHz (+ Turbo Core)**
- **P_{max} = (2.6 x 8 x 8) GF/s = 166.4 GF/s**

Each Bulldozer module:
- 2 “lightweight” cores
- 1 FPU: 4 MULT & 4 ADD (double precision) / cycle
- Supports AVX
- Supports FMA4

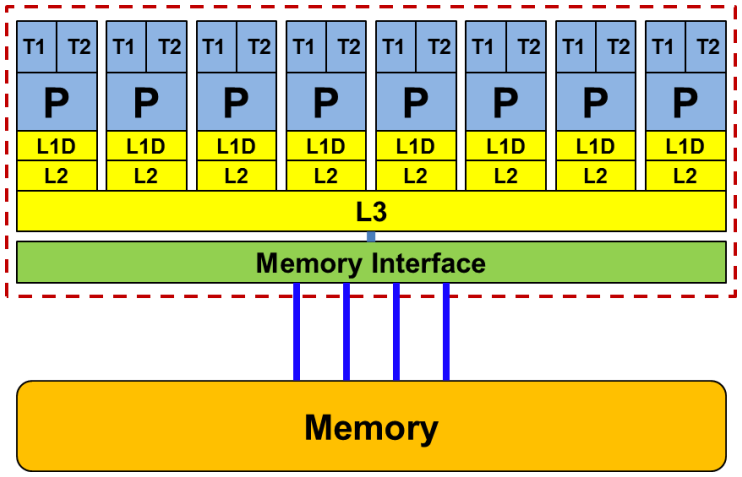


Cray XC30 “SandyBridge-EP” 8-core dual socket node



- 8 cores per socket 2.7 GHz (3.5 @ turbo)
- DDR3 memory interface with 4 channels per chip
- Two-way SMT
- Two 256-bit SIMD FP units
  - SSE4.2, AVX
- 32 kB L1 data cache per core
- 256 kB L2 cache per core
- 20 MB L3 cache per chip

There is no single driving force for chip performance!



Intel Xeon
"Sandy Bridge EP" socket
4,6,8 core variants available

Floating Point (FP) Performance:

$$P = n_{core} * F * S * \nu$$

n_{core}	number of cores:	8
F	FP instructions per cycle: (1 MULT and 1 ADD)	2
S	FP ops / instruction: (256 Bit SIMD registers – "AVX")	4 (dp) / 8 (sp)
ν	Clock speed :	~2.7 GHz

TOP500 rank 1 (mid-90s) → **P = 173 GF/s (dp) /** 346 GF/s (sp)

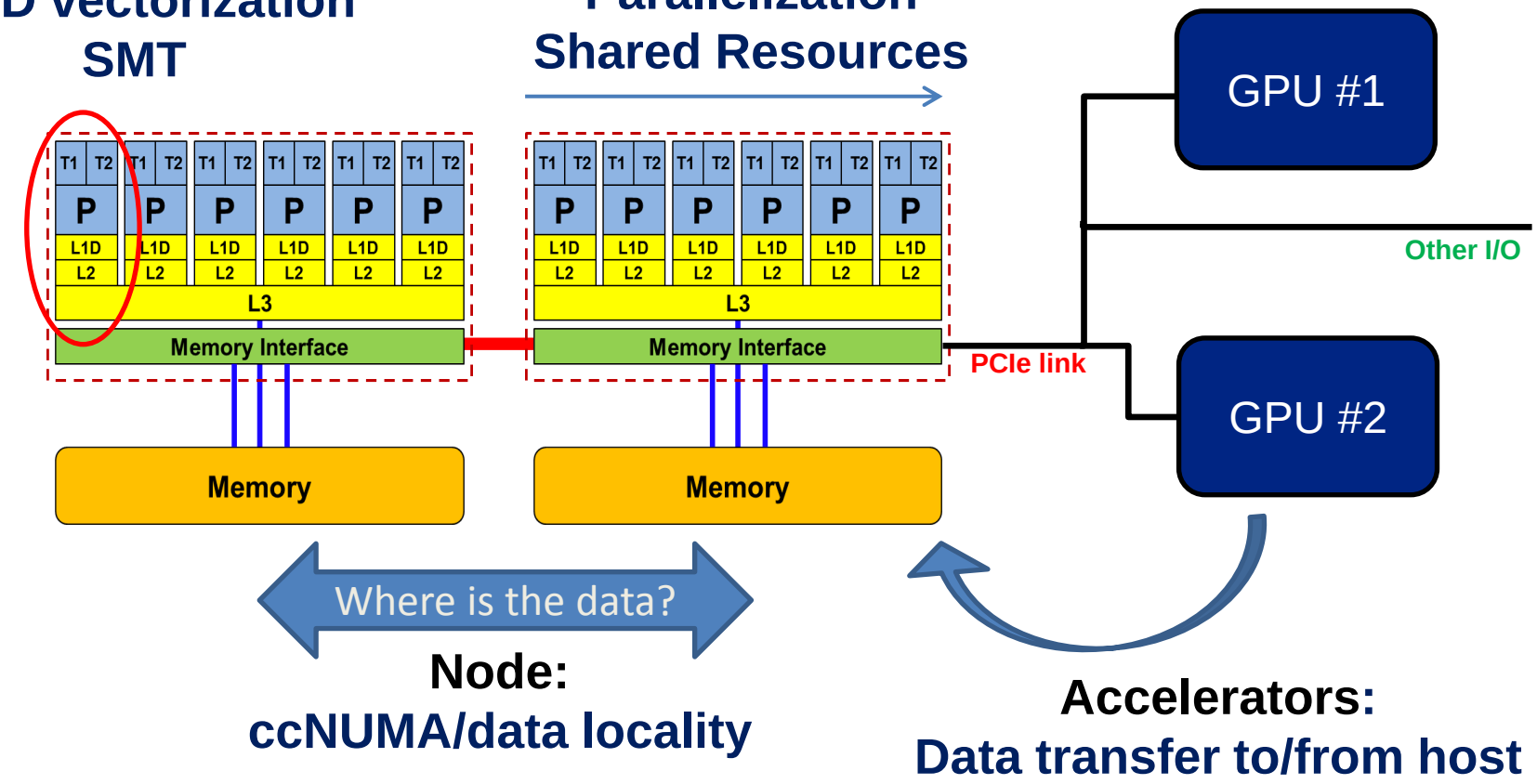
But: P=5.4 GF/s (dp) for serial, non-SIMD code

Challenges with modern compute nodes

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

Core:
SIMD vectorization
SMT

Socket:
Parallelization
Shared Resources

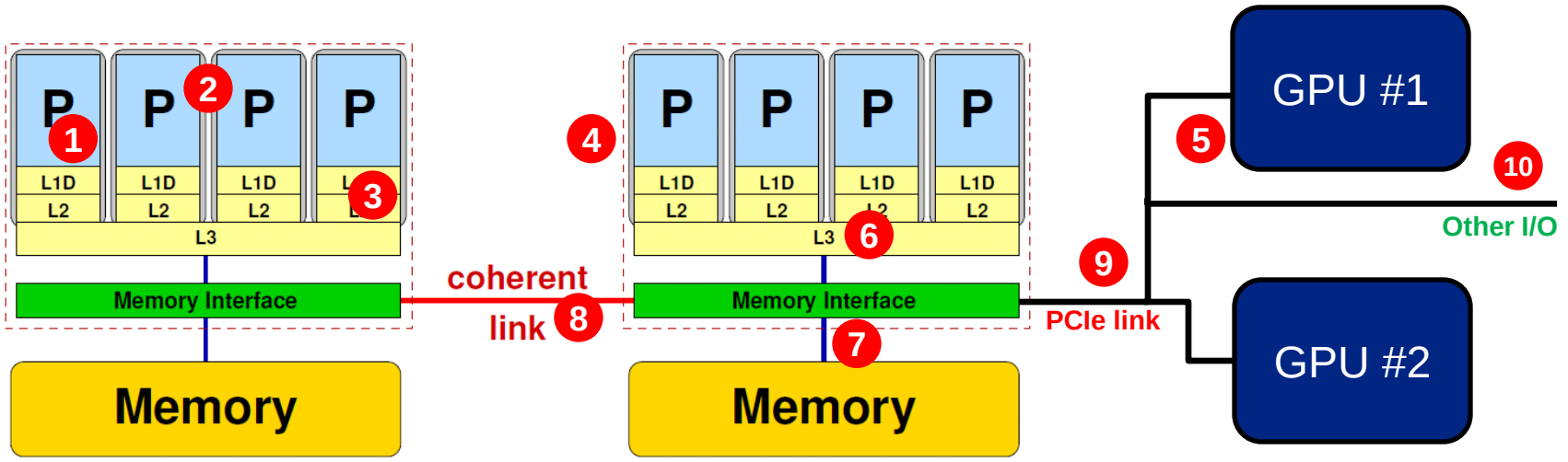


Node:
ccNUMA/data locality

Accelerators:
Data transfer to/from host

Parallelism in a modern compute node

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



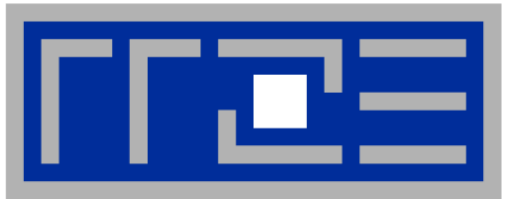
Parallel resources:

- 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?



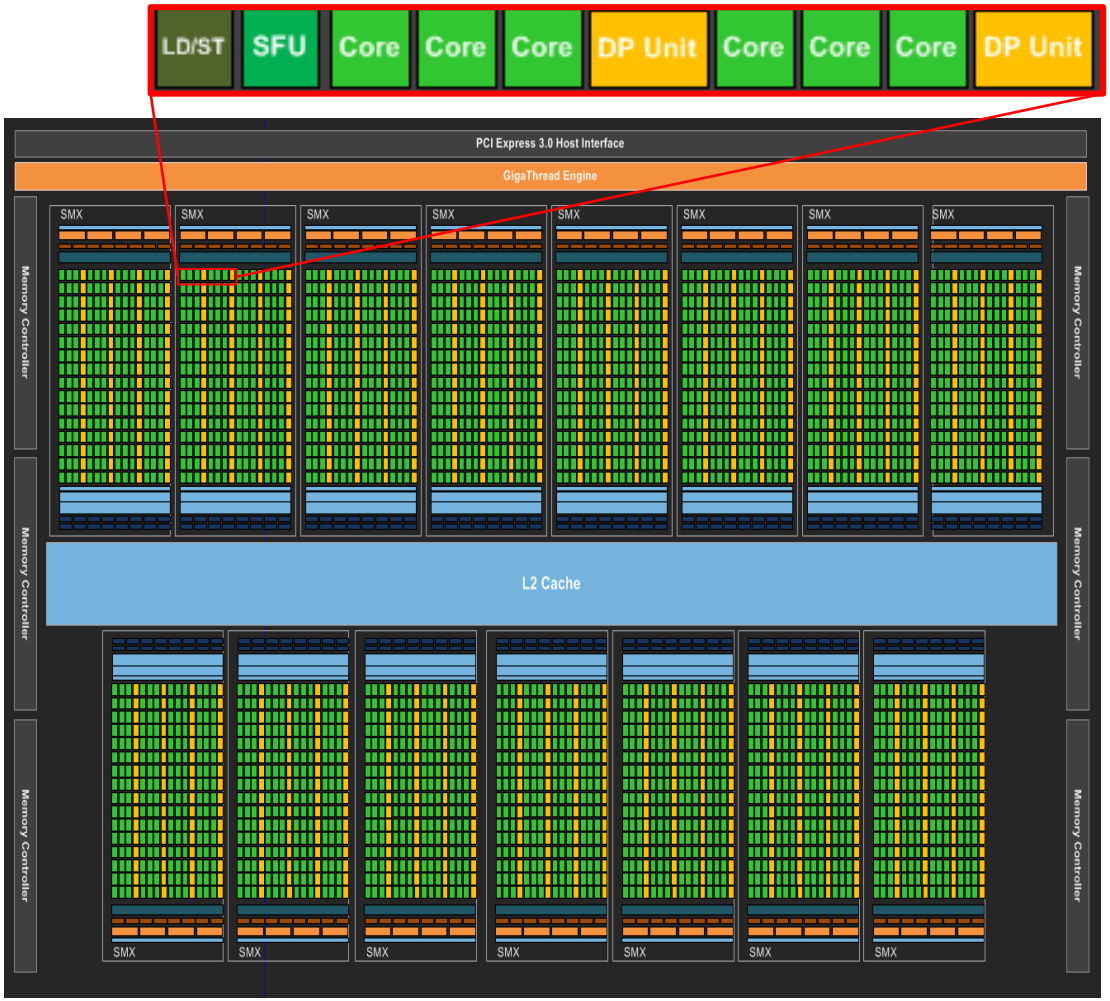
Interlude:
A glance at current accelerator technology

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

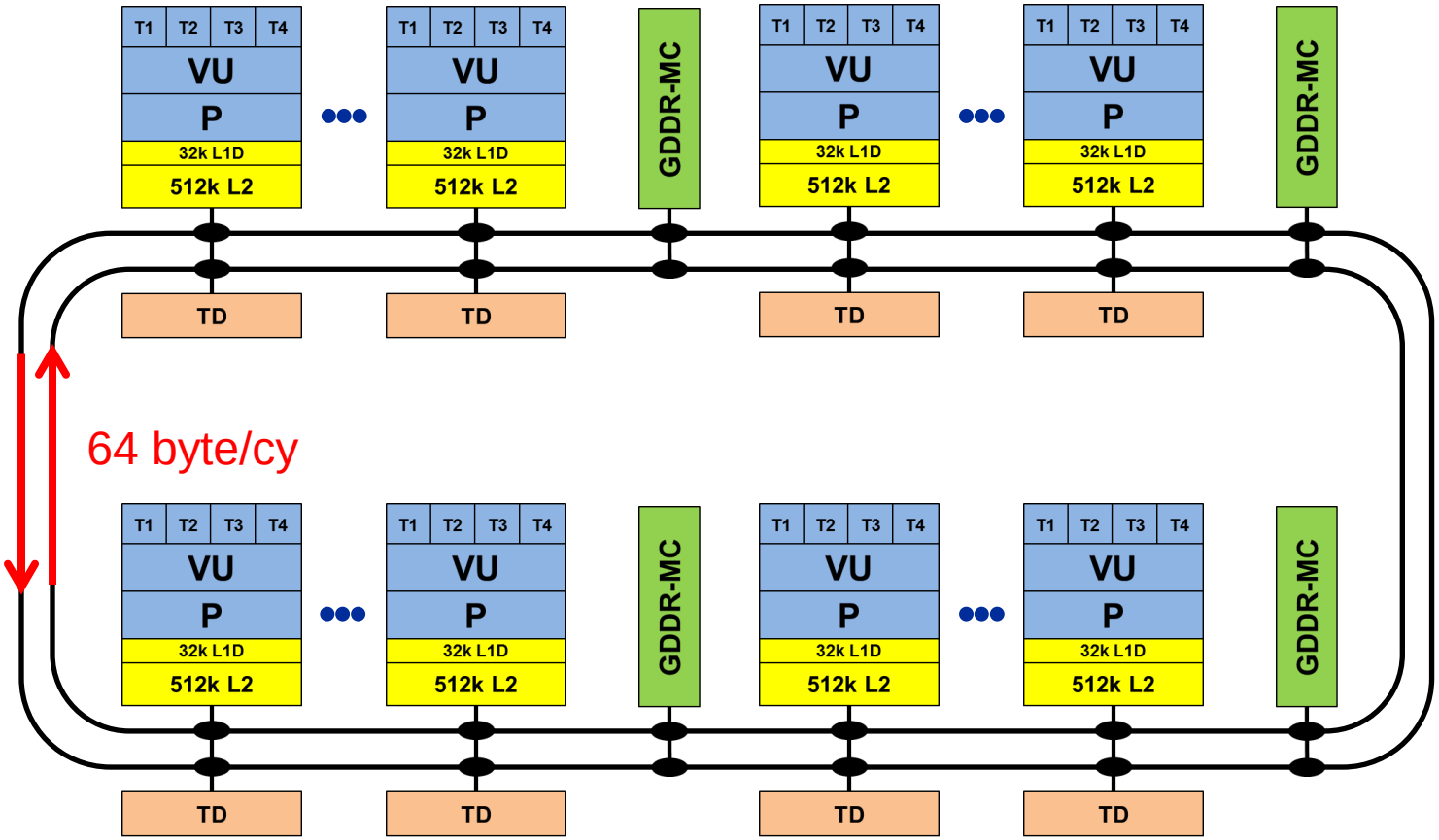
- 3:1 SP:DP performance



Intel Xeon Phi block diagram

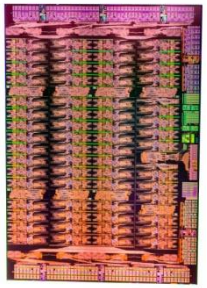
Architecture

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



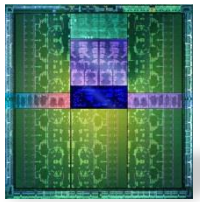
Comparing accelerators

- **Intel Xeon Phi**
  - **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **480/960 SIMD DP/SP tracks**
  - Clock Speed: ~1000 MHz
  - Transistor count: ~3 B (22nm)
  - Power consumption: ~250 W



  - Peak Performance (DP): ~ 1 TF/s
  - Memory BW: ~250 GB/s (GDDR5)
  - Threads to execute: 60-240+
  - Programming: Fortran/C/C++ +OpenMP + SIMD

- **NVIDIA Kepler K20**



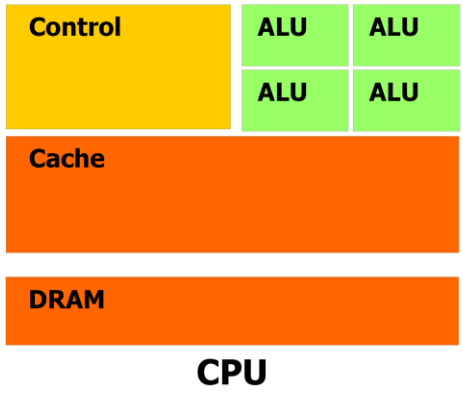
  - 15 SMX units each with 192 "cores" → **960/2880 DP/SP "cores"**
  - Clock Speed: ~700 MHz
  - Transistor count: 7.1 B (28nm)
  - Power consumption: ~250 W
  - Peak Performance (DP): ~ 1.3 TF/s
  - Memory BW: ~ 250 GB/s (GDDR5)
  - Threads to execute: 10,000+
  - Programming: CUDA, OpenCL, (OpenACC)

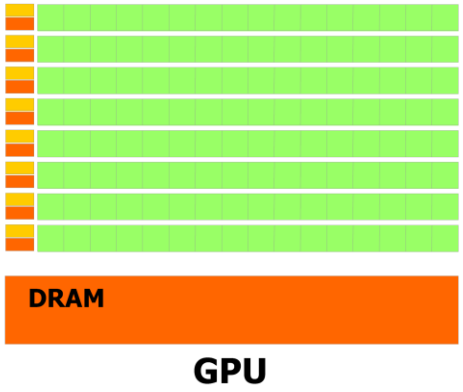
Top7: "Stampede" at Texas Center for Advanced Computing	**TOP500 rankings Nov 2012**	Top1: "Titan" at Oak Ridge National Laboratory

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**





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

⁺ *Single Precision*

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

Complete compute device

Conclusions about architecture

- **Modern computer architecture has a rich "topology"**
- **Node-level hardware parallelism takes many forms**
  - Sockets/devices – CPU: 1-8, GPGPU: 1-6
  - Cores – moderate (CPU: 4-16) to massive (GPGPU: 1000's)
  - SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10's-100's)
  - Superscalarity (CPU: 2-6)
- **Exploiting performance: parallelism + bottleneck awareness**
  - **"High Performance Computing" == computing at a bottleneck**
- **Performance of programs is sensitive to architecture**
  - Topology/affinity influences overheads of popular programming models
  - Standards do not contain (many) topology-aware features
    - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
  - Apart from overheads, performance features are largely independent of the programming model

Agenda

- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA

Multicore Performance and Tools

Probing node topology

- Standard tools
- likwid-topology

How do we figure out the node topology?

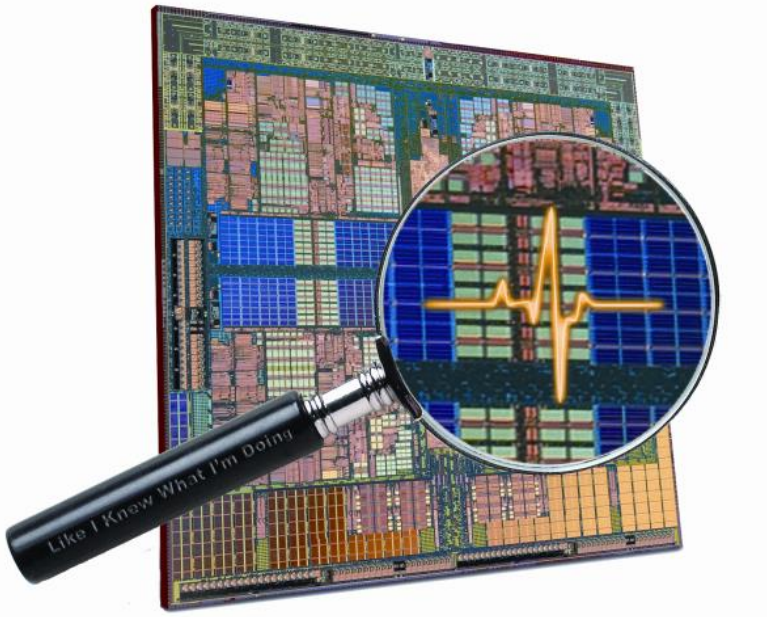
- **Topology =**
  - Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?
- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers 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
```

How do we figure out the node topology?

- **LIKWID tool suite:**

  Like
  I
  Knew
  What
  I'm
  Doing



- **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
DOI: 10.1109/ICPPW.2010.38
Preprint: http://arxiv.org/abs/1004.4431

Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **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`

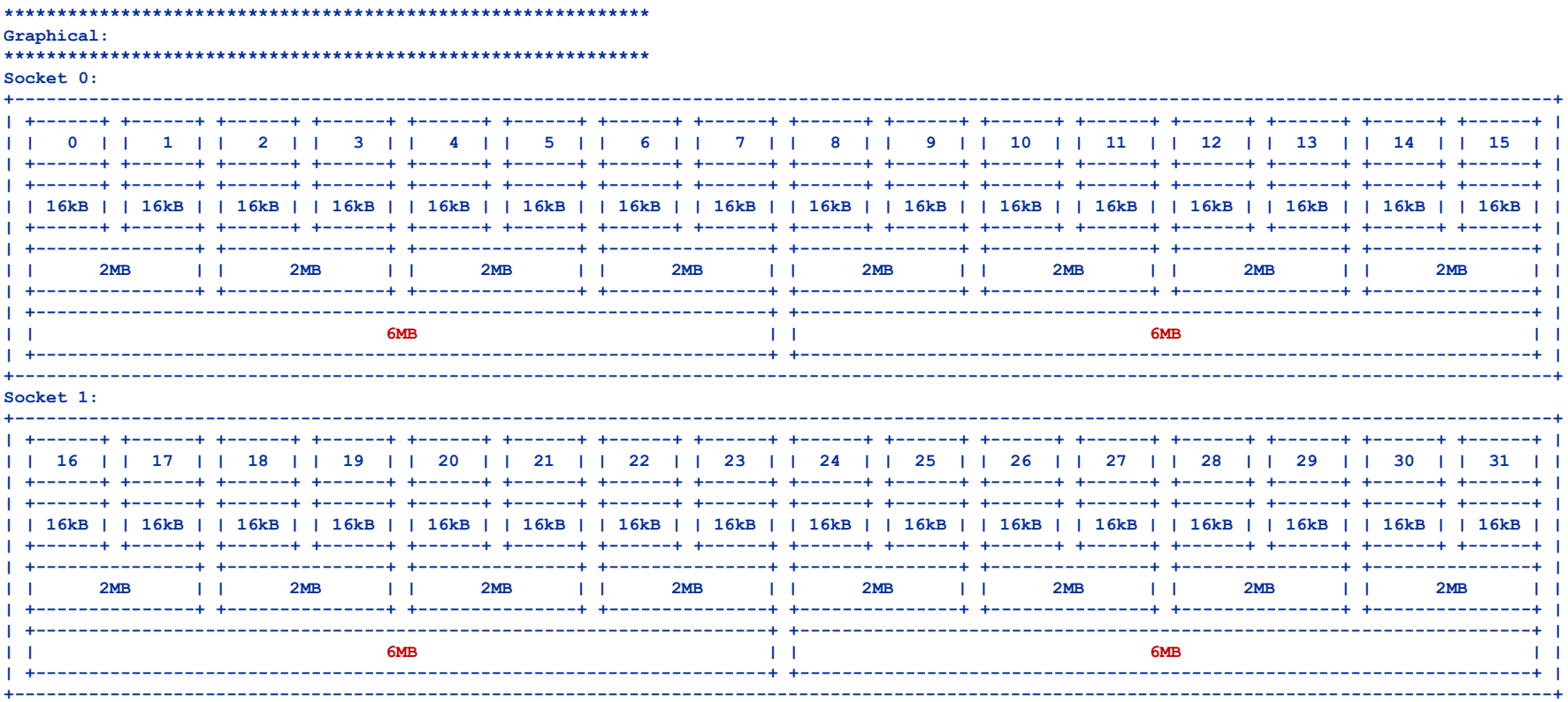
on one node of Cray XE6 "Hermit"

```
--------------------------------------------------------------
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
--------------------------------------------------------------
```

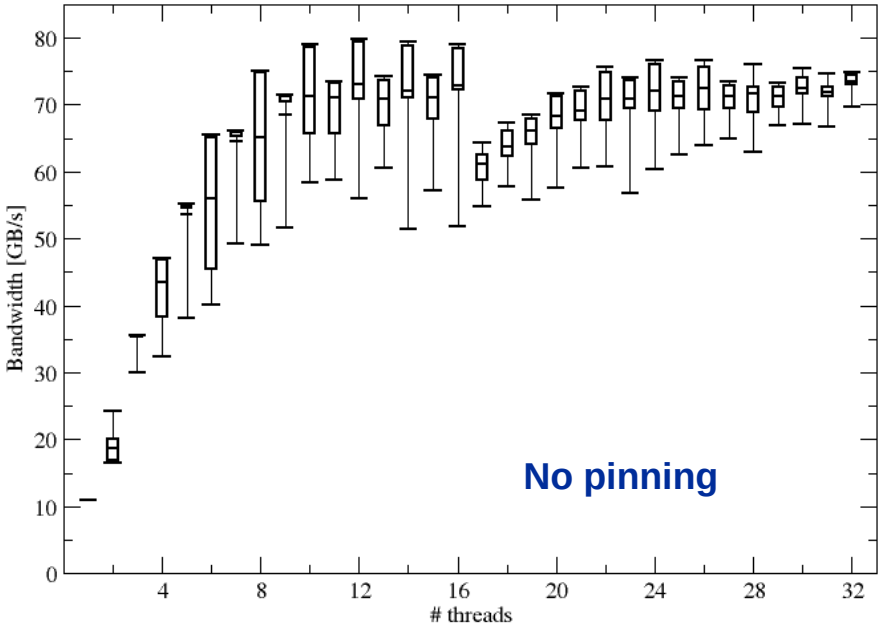
Output of likwid-topology continued

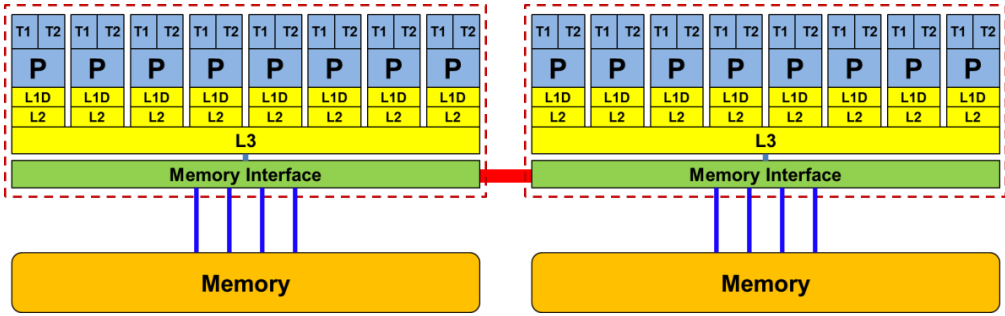


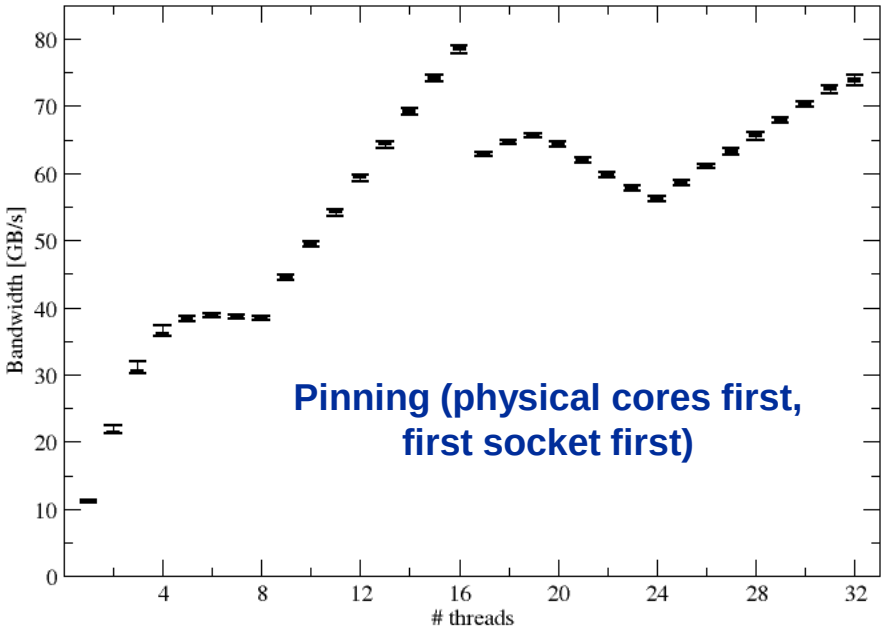
Enforcing thread/process-core affinity under the Linux OS

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

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







There are several reasons for caring about affinity:

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

Generic thread/process-core affinity under Linux

Overview

- `taskset [OPTIONS] [MASK | -c LIST ] \`
  `[PID | command [args]...]`

- `taskset` **restricts processes/threads to a *set of CPUs*. Examples:**

```
taskset 0x0006 ./a.out
taskset -c 4 33187
```

- **Processes/threads can still move within the set!**

- **Alternative: let process/thread bind itself by executing syscall**

```
#include <sched.h>
int sched_setaffinity(pid_t pid, unsigned int len,
                      unsigned long *mask);
```

- **Disadvantage: which CPUs should you bind to on a non-exclusive machine?**

- **Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA**

Generic thread/process-core affinity under Linux

- **Complementary tool:** `numactl`

  Example: `numactl --physcpubind=0,1,2,3 command [args]`
  Restricts process to specified physical core numbers

  Example: `numactl --cpunodebind=1 command [args]`
  Restricts process to specified ccNUMA node(s)

- **Many more options (e.g., interleave memory across nodes)**
  - → see section on ccNUMA optimization

- **Diagnostic command (see earlier):**
  `numactl --hardware`

- **Again, this is not suitable for a shared machine**

More thread/Process-core affinity (“pinning”) options

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

    Linux: **`sched_setaffinity()`**, PLPA (see below) → hwloc
    Windows: **`SetThreadAffinityMask()`**
    …
- **Support for “semi-automatic” pinning in some compilers/environments**
  - Intel compilers > V9.1 (**`KMP_AFFINITY`** environment variable)
  - PGI, Pathscale, GNU
  - SGI Altix **`dplace`** (works with logical CPU numbers!)
  - Generic Linux: **`taskset`**, **`numactl`**, **`likwid-pin`** (see below)
  - OpenMP 4.0
- **Affinity awareness in MPI libraries**
  - OpenMPI
  - Intel MPI
  - Cray MPI
  - …

Likwid-pin

Overview

- **Pins processes and threads to specific cores without touching code**
- **Directly supports pthreads, gcc OpenMP, Intel OpenMP**
- **Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!**
- **Can 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:**
  - Physical numbering (as given by likwid-topology):
    ```
    likwid-pin -c 0,2,4-6 ./myApp parameters
    ```
  - Logical numbering by topological entities:
    ```
    likwid-pin -c S0:0-3 ./myApp parameters
    ```

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

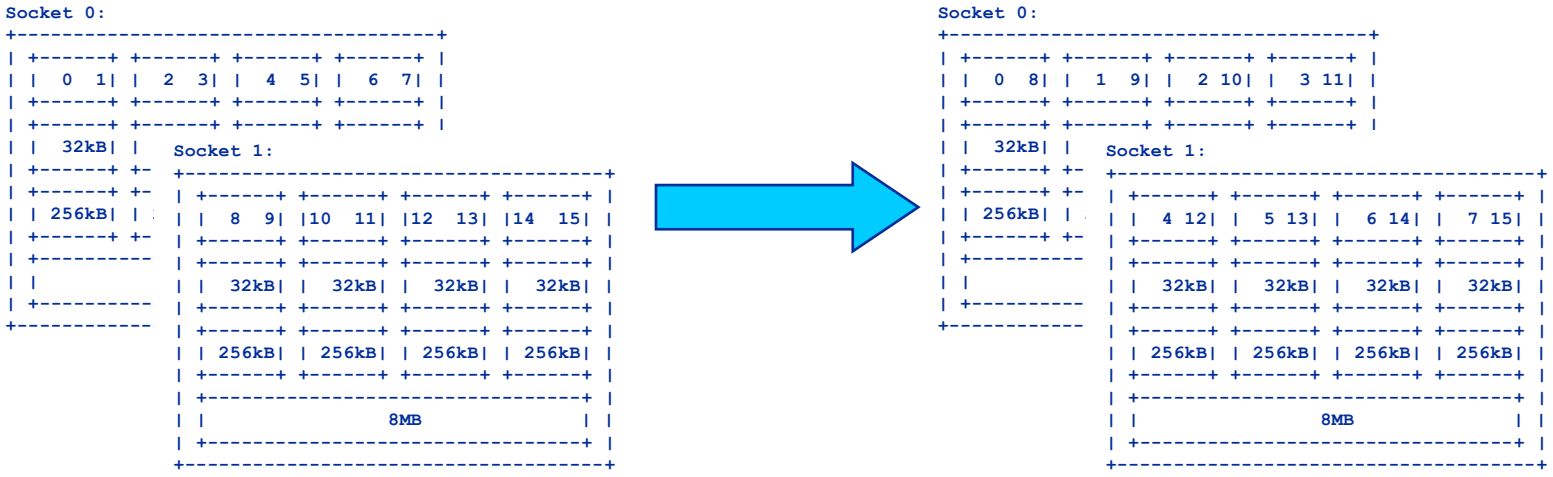
```
$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK
-----------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-----------------------------------------------
[... 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`

Likwid-pin

Using logical core numbering

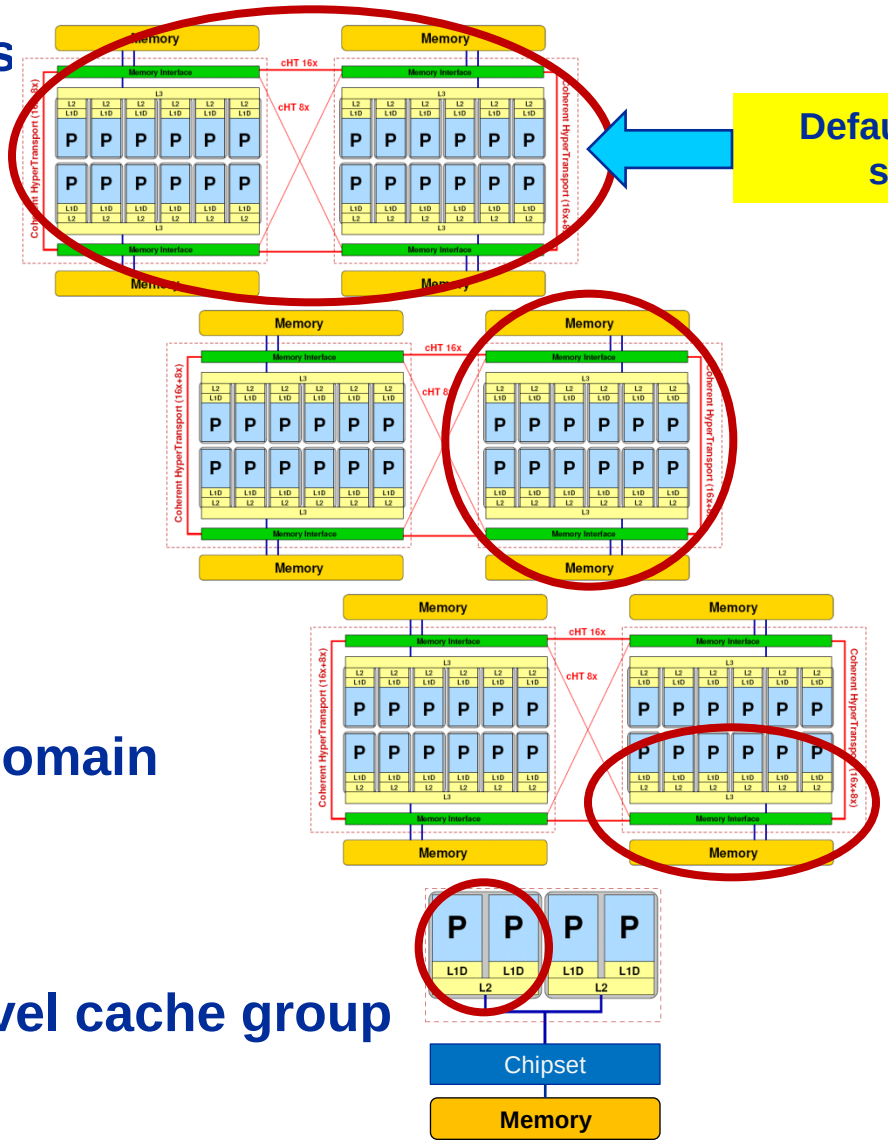
- **Possible unit prefixes**

N **node**

S **socket**

M **NUMA domain**

C **outer level cache group**



DEMO

Agenda

- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA

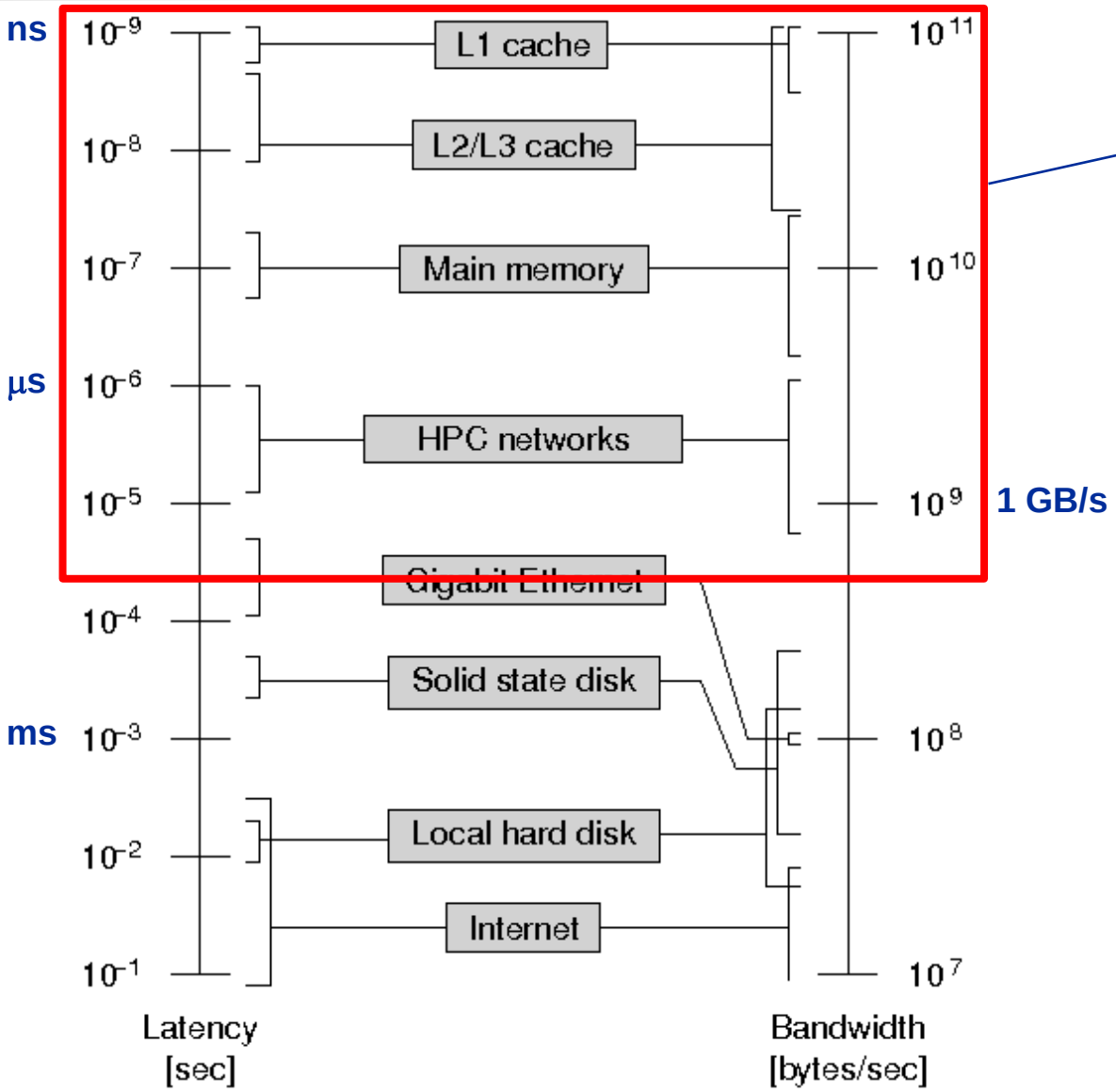
Microbenchmarking for architectural exploration

Probing of the memory hierarchy

Saturation effects in cache and memory

Typical OpenMP overheads

Latency and bandwidth in modern computer environments

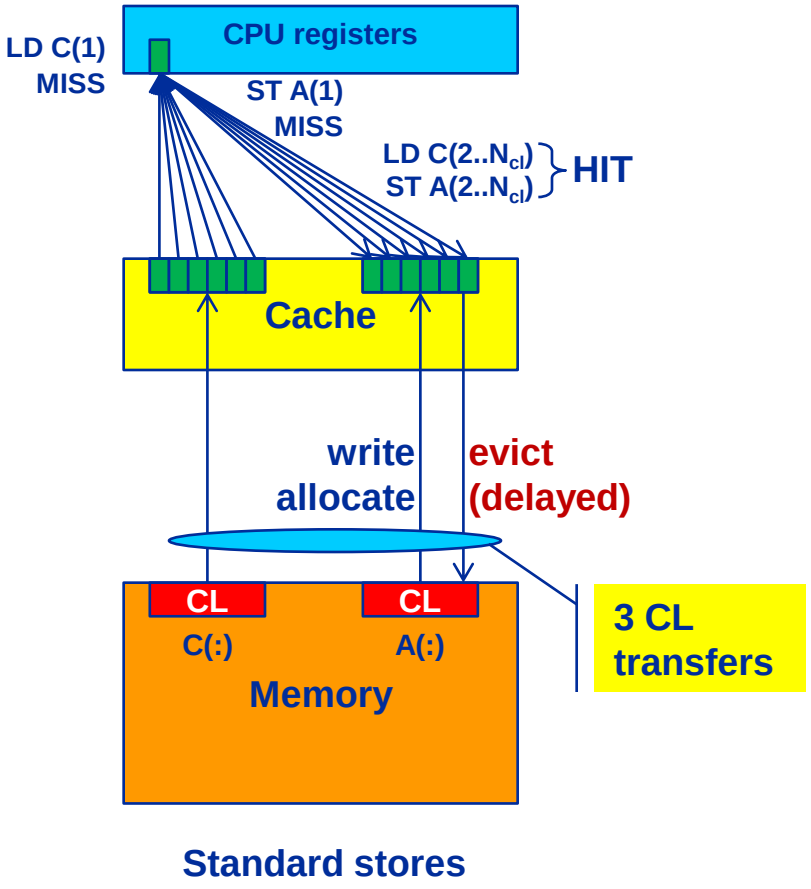


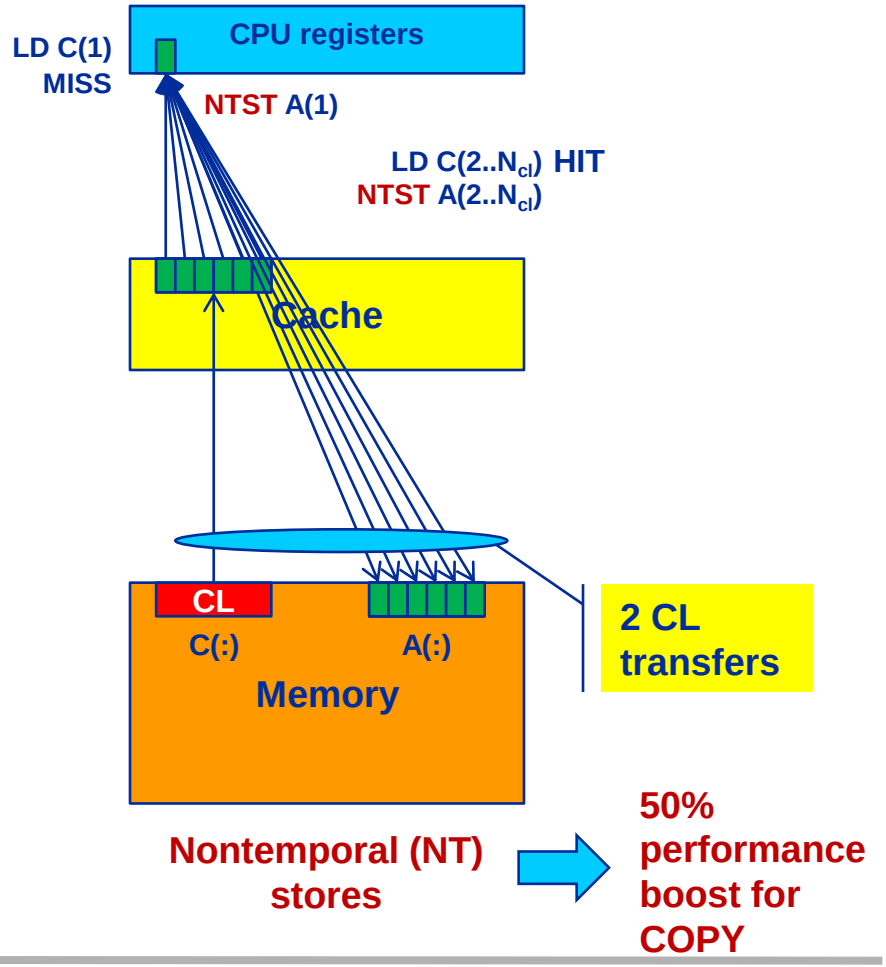
HPC plays here

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(:)`





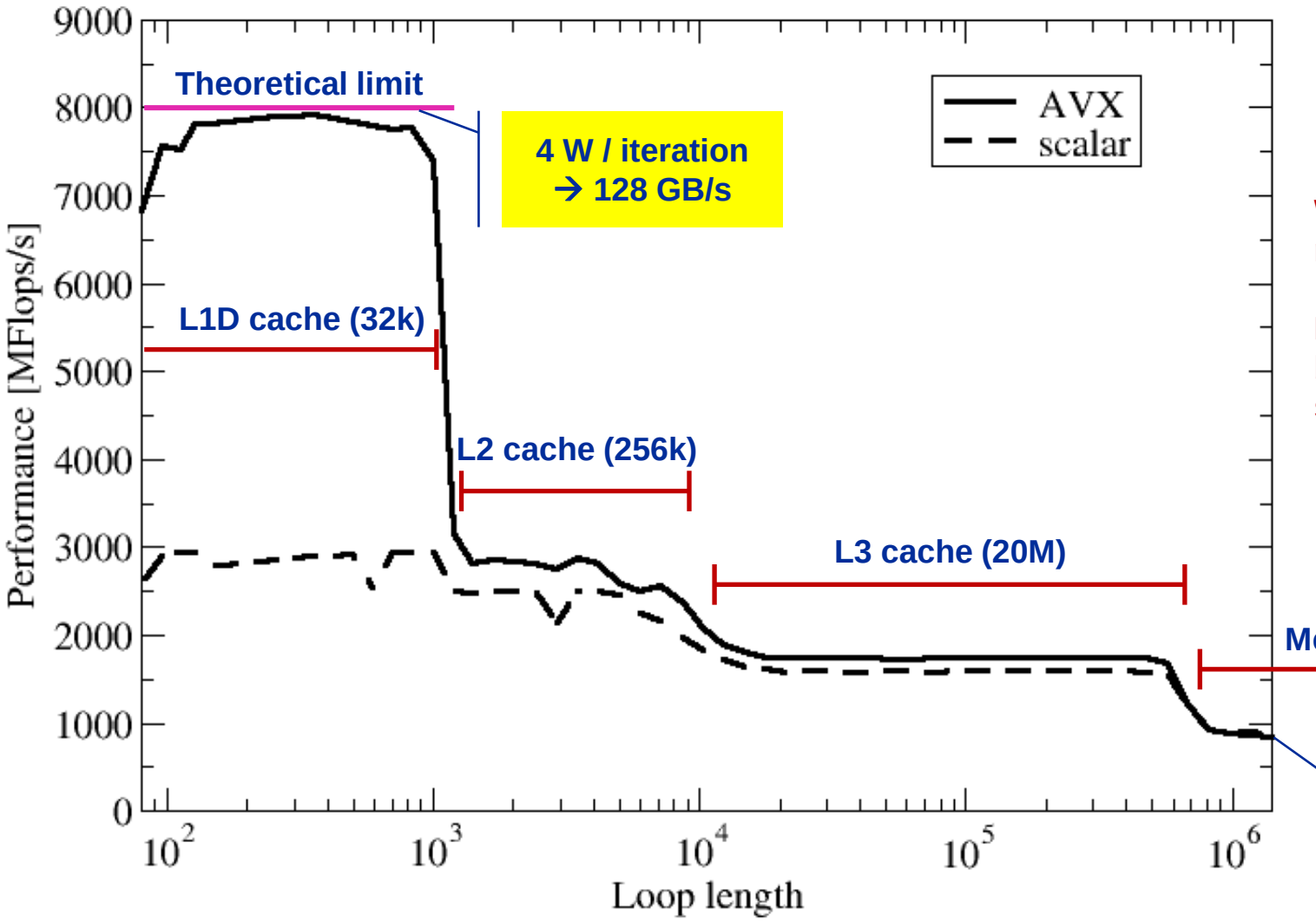
Simple streaming benchmark:

```
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!

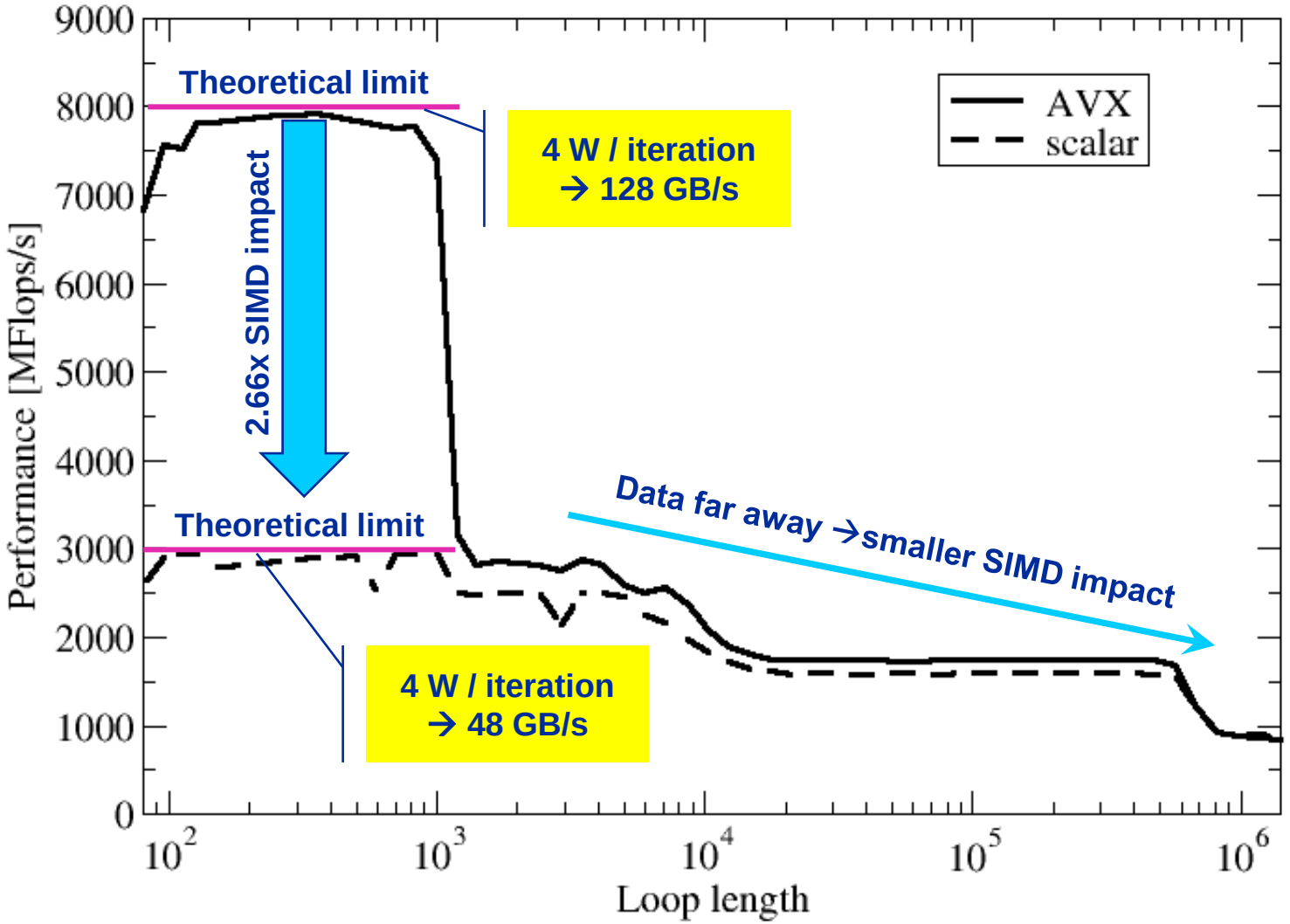
A(:)=B(:)+C(:)*D(:) on one Sandy Bridge core (3 GHz)



What about multiple cores?

Do the bandwidths scale?

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



See later for more on SIMD benefits

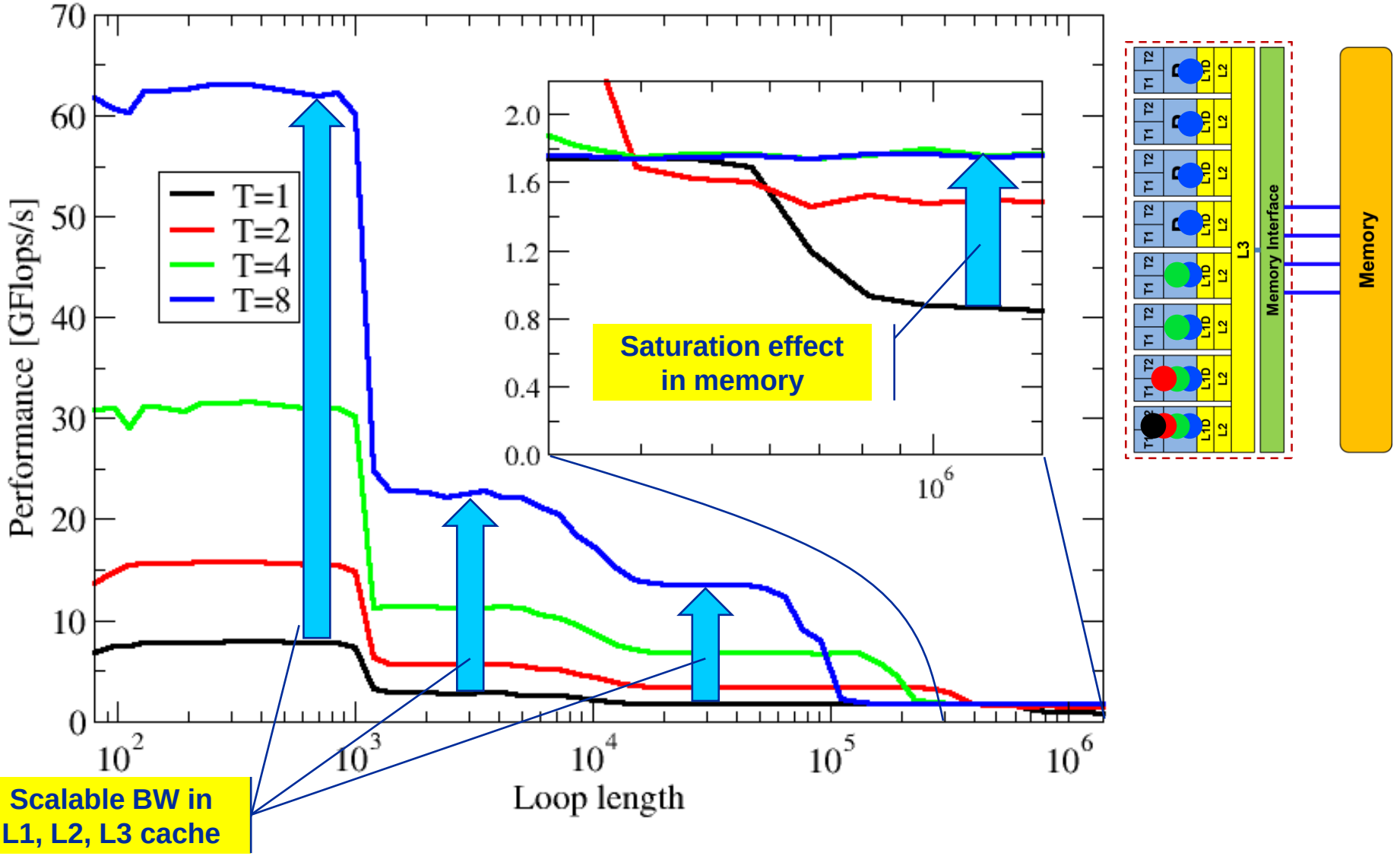
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

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

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

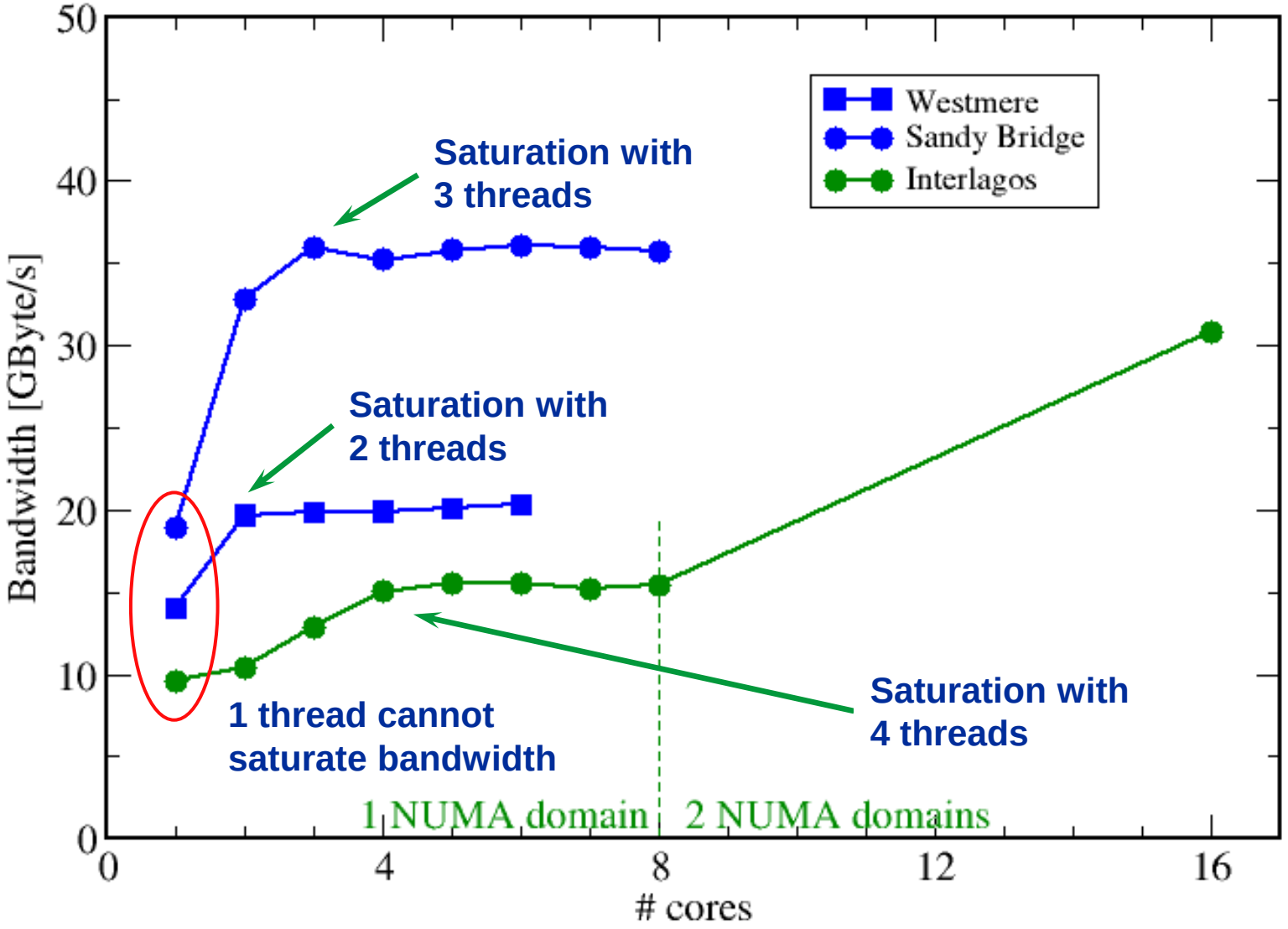
→ pure hardware probing, no impact from OpenMP overhead

Throughput vector triad on Sandy Bridge socket (3 GHz)



Bandwidth limitations: Main Memory

Scalability of shared data paths inside a NUMA domain **(V-Triad)**

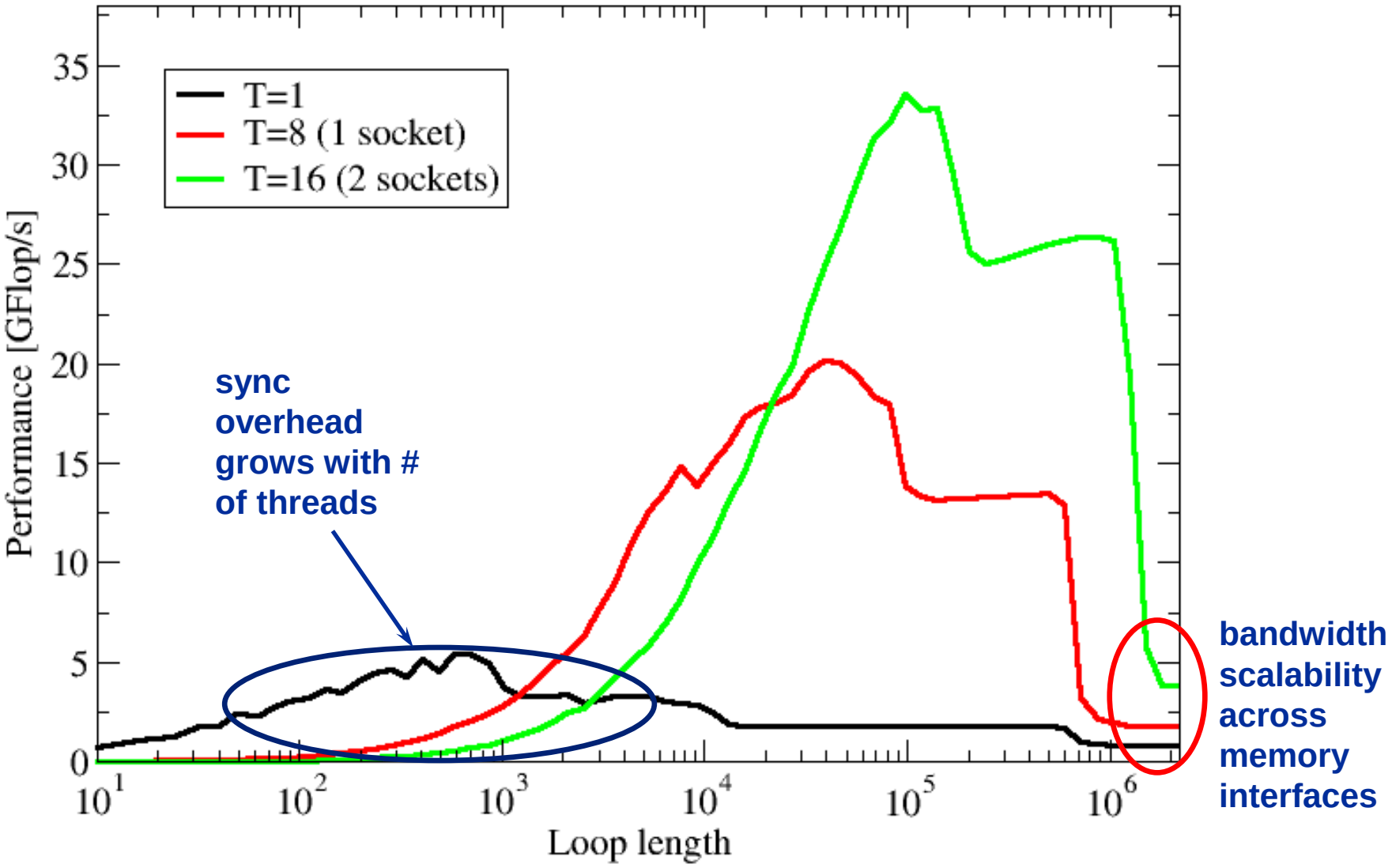


OpenMP work sharing in the benchmark loop

```
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 progams.

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization!

- **Next slide: 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

Conclusions from the microbenchmarks

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is
- **Bandwidth bottlenecks are ubiquitous**
- **Synchronization overhead may be an issue**
  - … and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization

Case study: OpenMP-parallel sparse matrix-vector multiplication (part 1)

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory

Case study: Sparse matrix-vector multiply

- **Important kernel in many applications (matrix diagonalization, solving linear systems)**
- **Strongly memory-bound for large data sets**
  - Streaming, with partially indirect access:

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

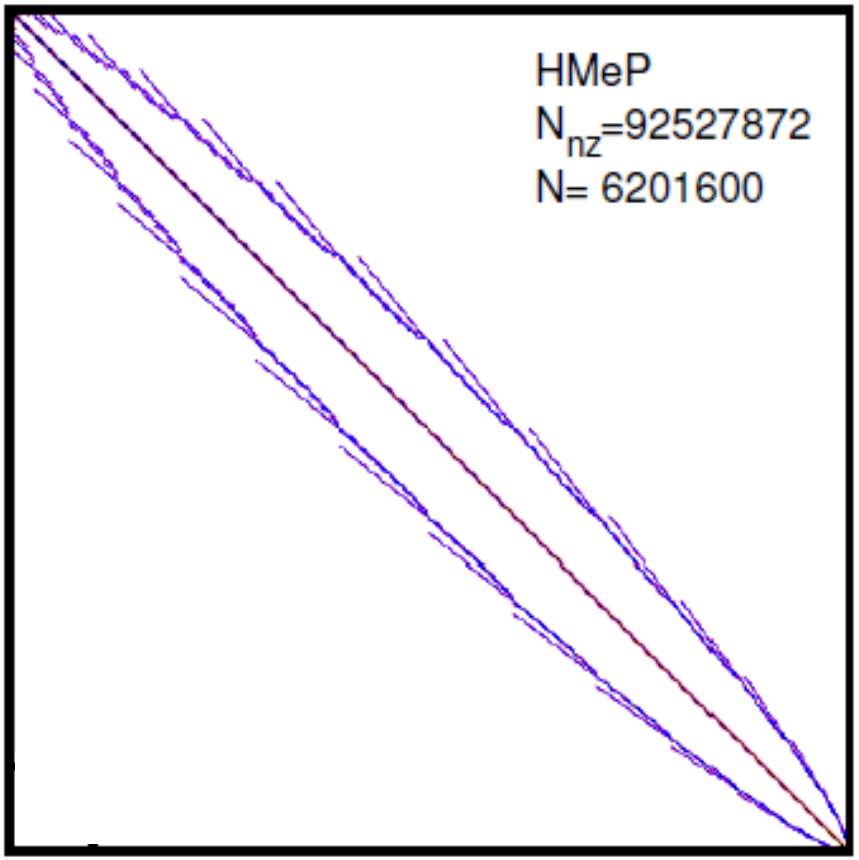
  - Usually many spMVMs required to solve a problem

- **Following slides: Performance data on one 24-core AMD Magny Cours node**

Bandwidth-bound parallel algorithms:
Sparse MVM

- **Data storage format is crucial for performance properties**
  - Most useful general format: Compressed Row Storage (CRS)
  - SpMVM is easily parallelizable in shared and distributed memory

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

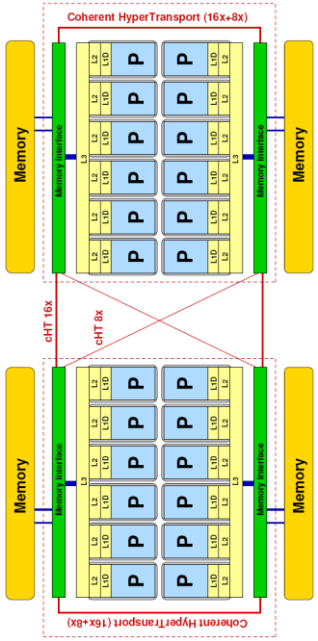


- **MPI-parallel spMVM is often communication-bound**
  - See later part for what we can do about this…

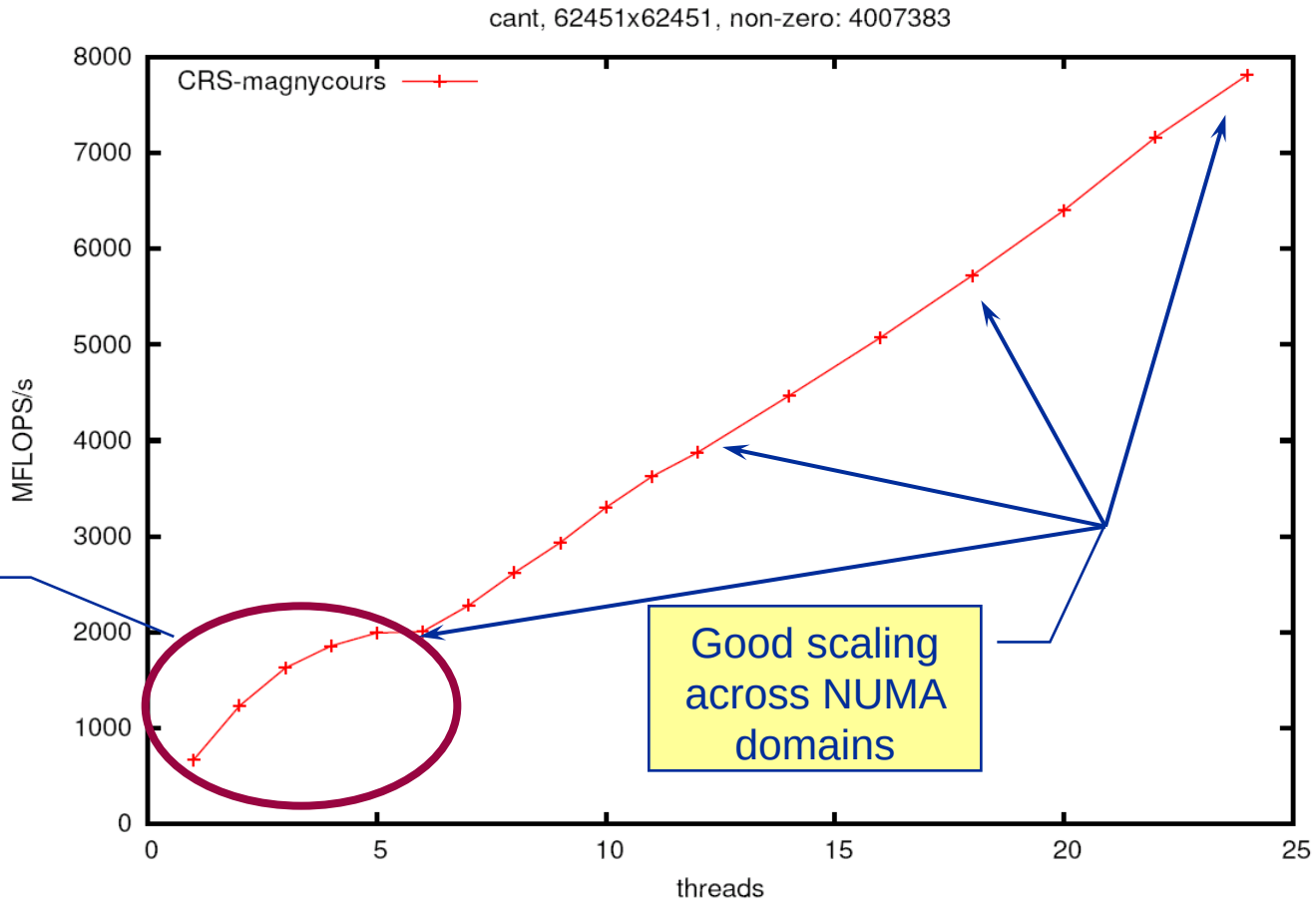
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 1: Large matrix**



Intrasocket bandwidth bottleneck

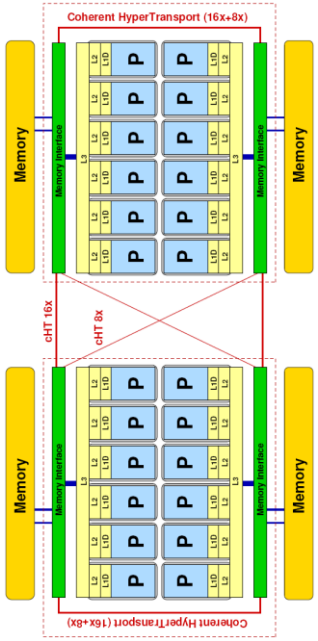


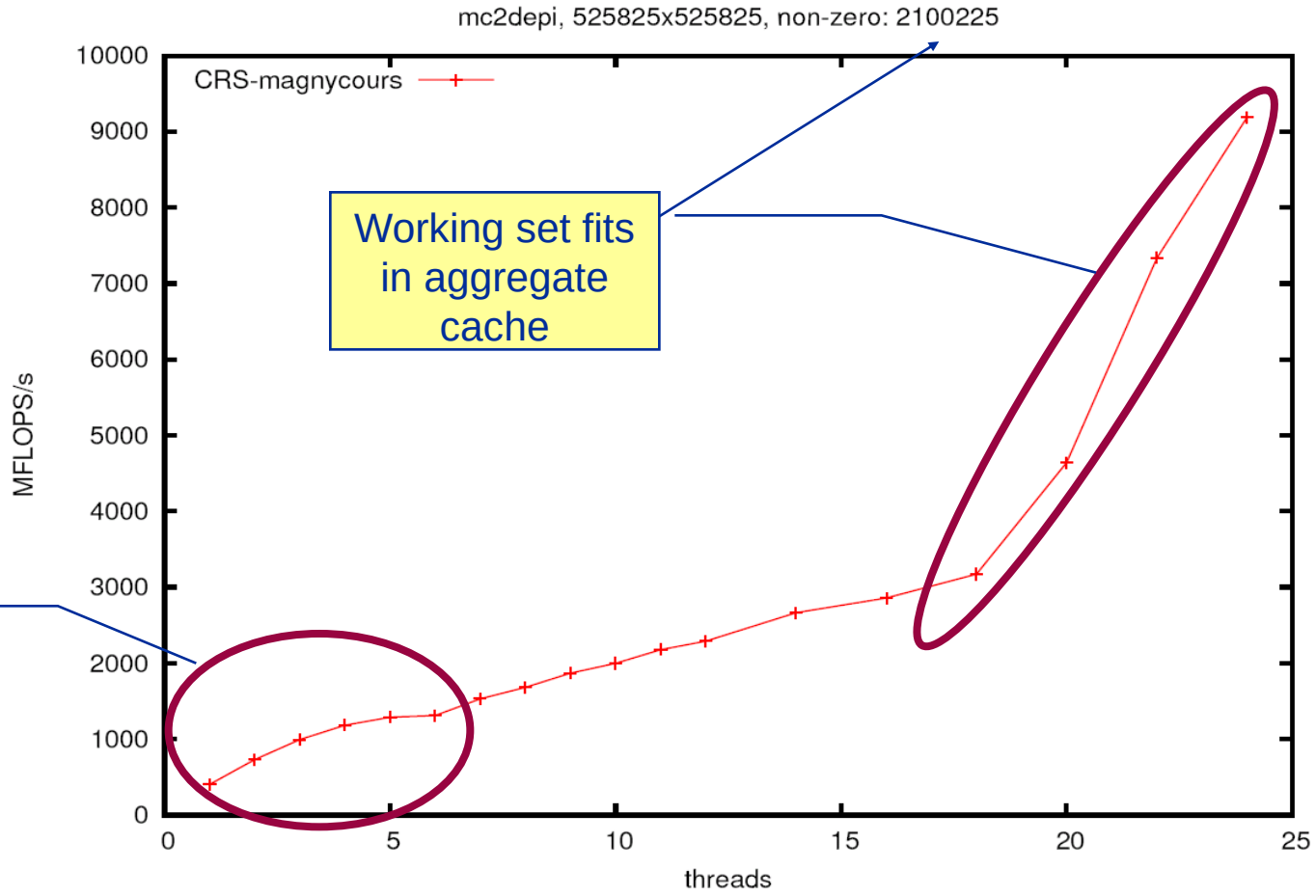
Good scaling across NUMA domains

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 2: Medium size**





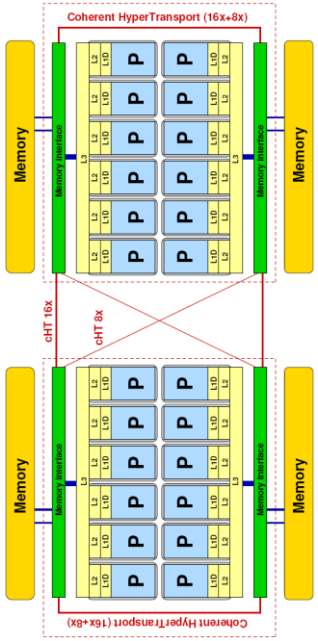
Working set fits in aggregate cache

Intrasocket bandwidth bottleneck

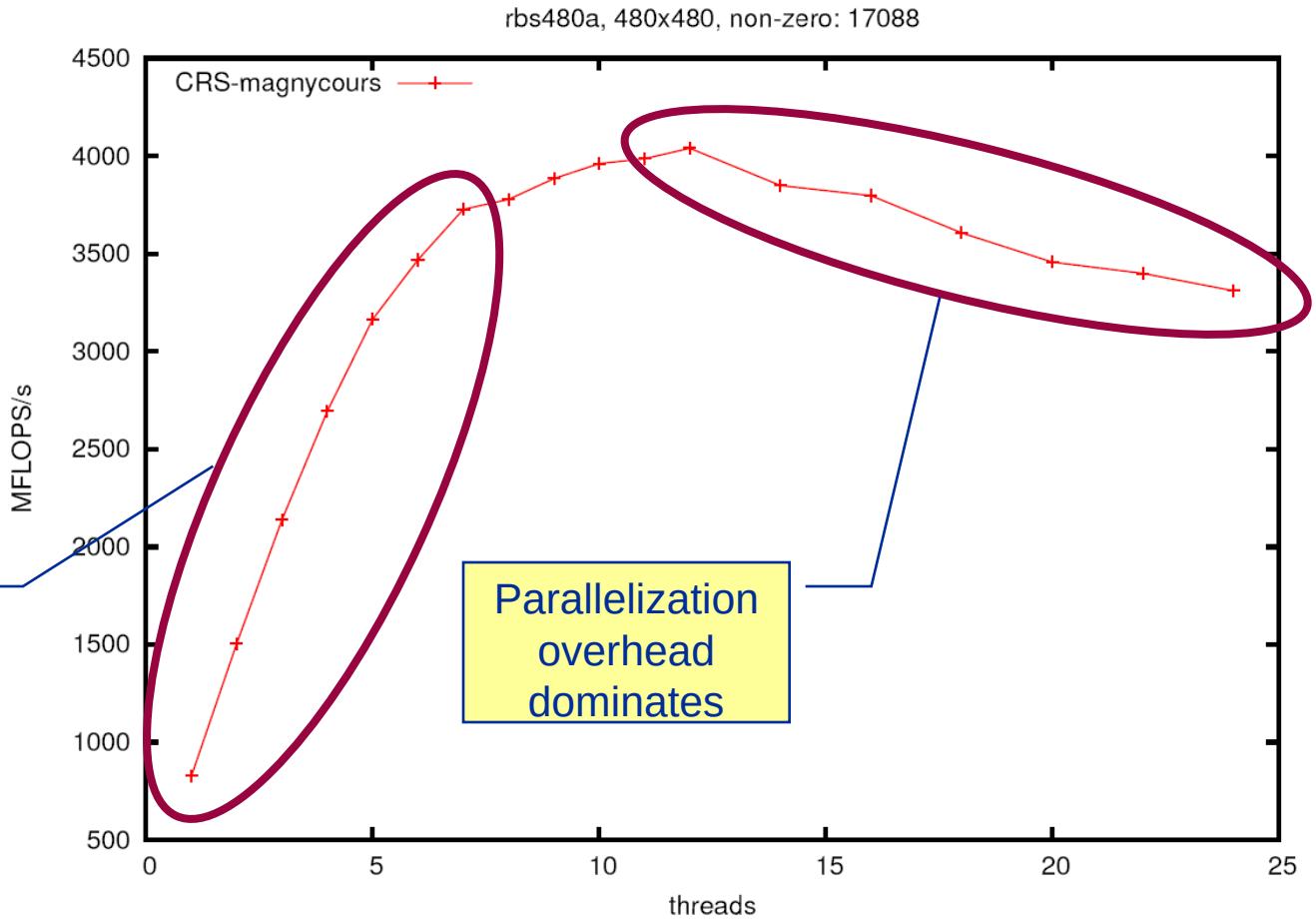
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 3: Small size**



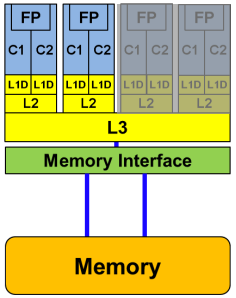
No bandwidth bottleneck



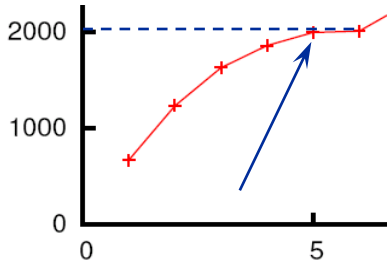
Parallelization overhead dominates

Conclusions from the spMVM benchmarks

- **If the problem is "large", bandwidth saturation on the socket is a reality**
  - → There are "spare cores"
  - Very common performance pattern
- **What to do with spare cores?**
  - Let them idle → saves energy with minor loss in time to solution
  - Use them for other tasks, such as MPI communication

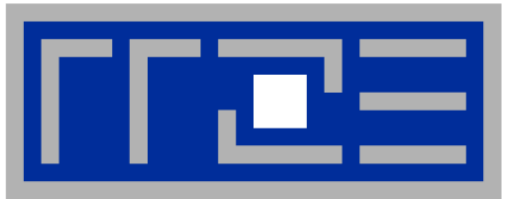


- **Can we predict the saturated performance?**
  - Bandwidth-based performance modeling!
  - What is the significance of the indirect access? Can it be modeled?
- **Can we predict the saturation point?**
  - … and why is this important?



Agenda

- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA



“Simple” performance modeling: The Roofline Model

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

The Roofline Model[1,2]

1. P_{max} = **Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessarily P_{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:

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

[1] W. Schönauer: Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers. (2000)
[2] S. Williams: Auto-tuning Performance on Multicore Computers. UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

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

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

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

- P_{peak} = **173 Gflop/s** (8 FP units x (4+4) Flops/cy x 2.7 GHz)
- P_{max}? → Observe LD/ST throughput maximum of 1 AVX Load and ½ AVX store per cycle → 3 cy / 8 Flops → P_{max} = **57.6 Gflop/s** (33% peak)

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

Example: Vector triad `A(:)=B(:)+C(:)*D(:)` on a 1.05 GHz 60-core Intel Xeon Phi chip (vectorized)

- b_S = **160 GB/s**
- B_c = (4+1) Words / 2 Flops = 2.5 W/F (including write allocate)
  → ***I* = 0.4 F/W = 0.05 F/B**

→ $I \cdot b_S$ = **8.0 GF/s (0.8 % of peak performance)**

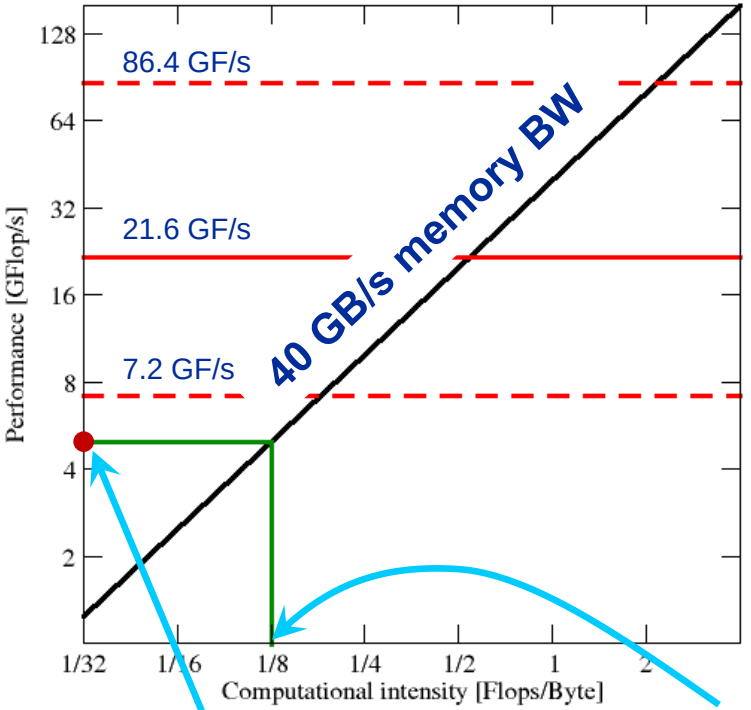
- P_{peak} = **1008 Gflop/s** (60 FP units x (8+8) Flops/cy x 1.05 GHz)
- P_{max}? → Observe LD/ST throughput maximum of 1 Load or 1 Store per cycle → 4 cy / 16 Flops → P_{max} = **252 Gflop/s** (25% of peak)

$$P = \min(P_{\max}, I \cdot b_S) = \min(252, 8.0)\ \mathrm{GFlop/s}$$
$$= 8.0\ \mathrm{GFlop/s}$$

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



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

ADD peak
(best possible code)

no SIMD

3-cycle latency per ADD
if not unrolled

How do we get these?
→ See next!

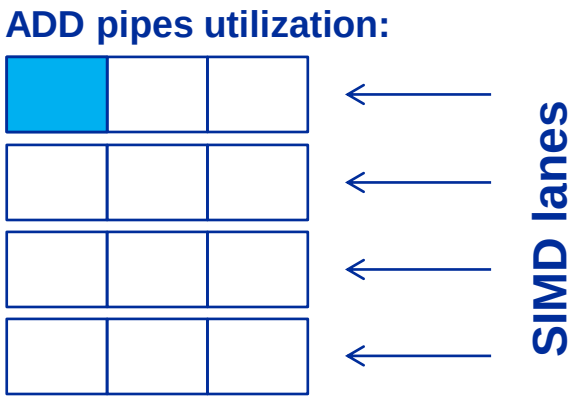
I = 1 Flop / 8 byte (in DP)

***P* = 5 Gflop/s**

Applicable peak for the summation loop

Plain scalar code, no SIMD

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



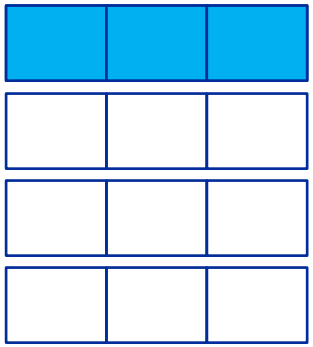
→ 1/12 of ADD peak

Applicable peak for the summation loop

Scalar code, 3-way unrolling

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

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

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

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

ADD pipes utilization:



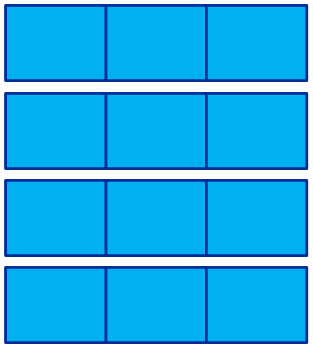
→ 1/4 of ADD peak

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
```

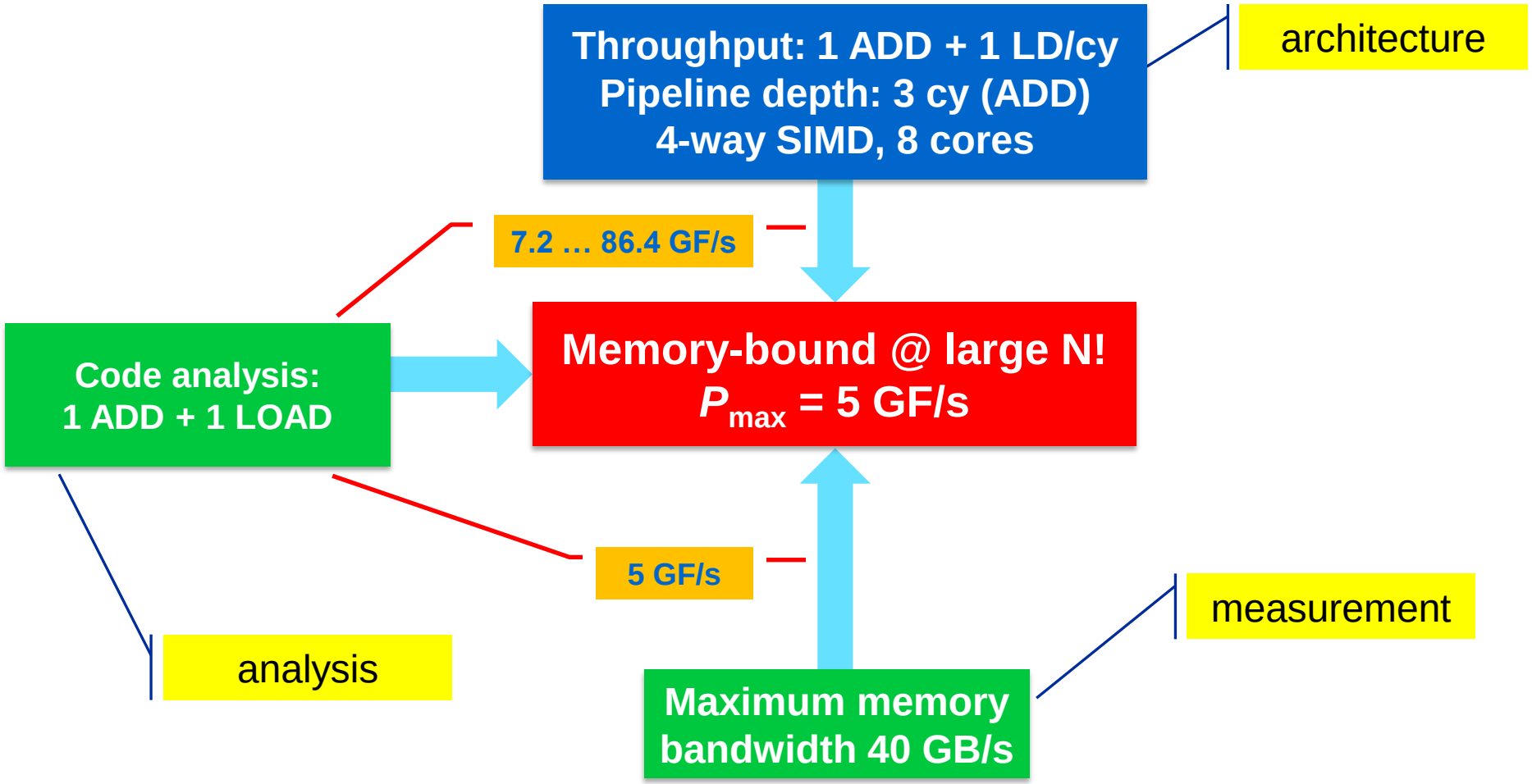
ADD pipes utilization:



→ ADD peak

Input to the roofline model

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



Assumptions for the Roofline Model

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

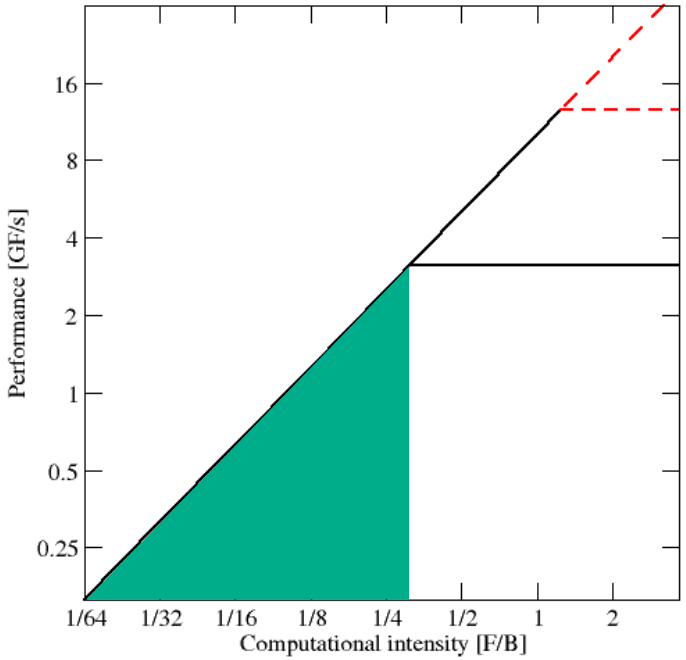
Factors to consider in the roofline model

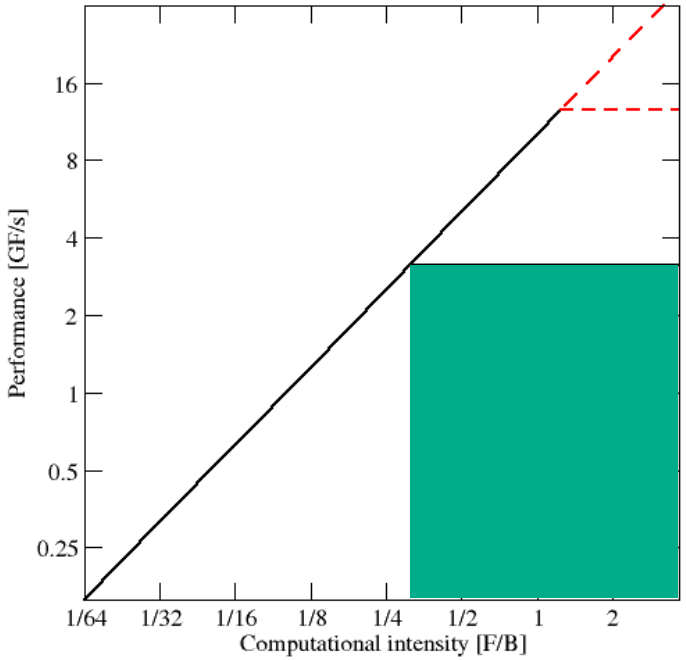
Bandwidth-bound (simple case)

- Accurate traffic calculation (write-allocate, strided access, …)
- Practical ≠ theoretical BW limits
- Erratic access patterns

Core-bound (may be complex)

- Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
- Limit is linear in # of cores



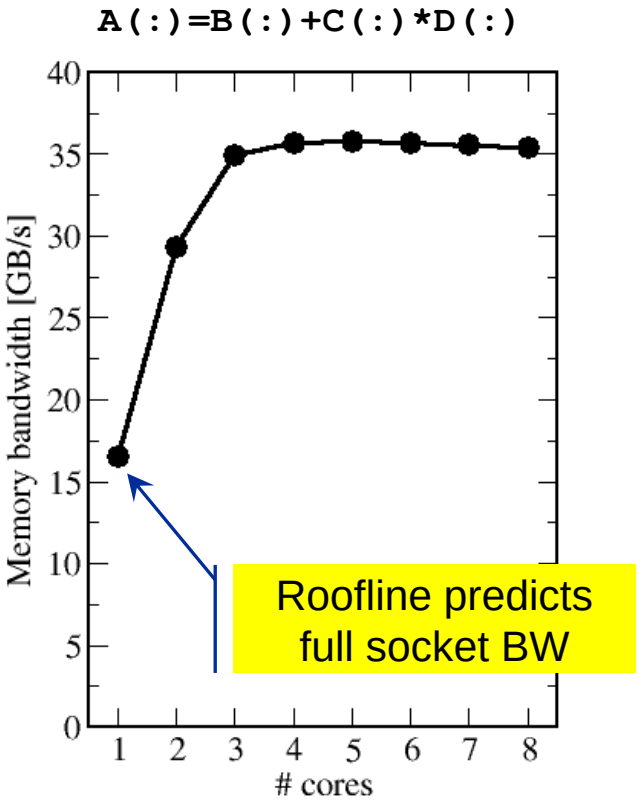


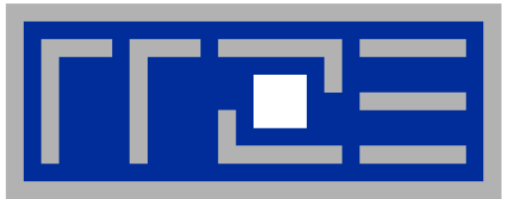
Shortcomings of the roofline model

- **Saturation effects in multicore chips are not explained**
  - Reason: "saturation assumption"
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - 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. Submitted. Preprint: arXiv:1208.2908





Case study: OpenMP-parallel sparse matrix-vector multiplication (part 2)

Putting Roofline to use where it should not work

Example: SpMVM node performance model

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

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

- **DP CRS comp. intensity**
  - κ quantifies extra traffic for loading RHS more than once

$$I_{\mathrm{CRS}} = \frac{2}{12 + 24/N_{\mathrm{nzr}} + \kappa}\ \frac{\mathrm{Flops}}{\mathrm{Byte}}$$

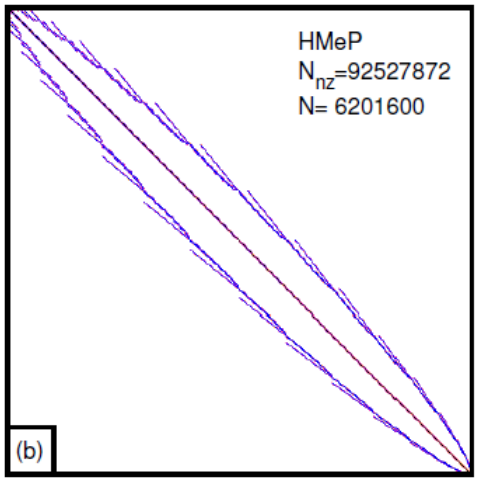
$$= \left(6 + \frac{12}{N_{\mathrm{nzr}}} + \frac{\kappa}{2}\right)^{-1} \frac{\mathrm{Flops}}{\mathrm{Byte}}$$

  - Expected performance = b_S x I_{CRS}
  - Determine κ by measuring performance and actual memory bandwidth
    - Maximum memory BW may not be achieved with spMVM

Roofline analysis for spMVM

- **Analysis for HMeP matrix on Nehalem EP socket**
  - BW used by spMVM kernel b = 18.1 GB/s → should get ≈ 2.66 Gflop/s spMVM performance if $\kappa = 0$
  - Measured spMVM performance = 2.25 Gflop/s
  - Solve 2.25 Gflop/s = b x I_{CRS} for $\kappa \approx 2.5$

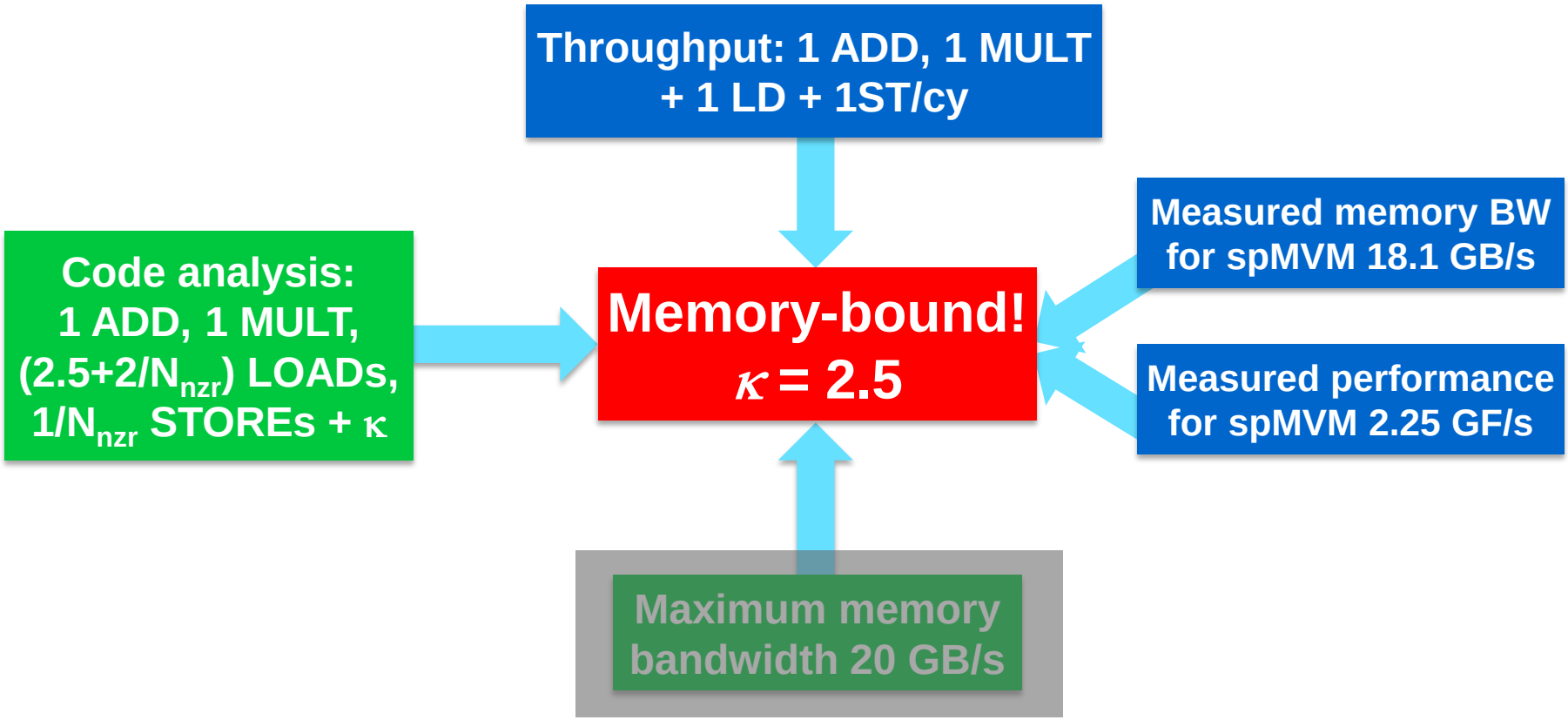
    → 37.5 extra bytes per row
    → **RHS is loaded 6 times** from memory
    → about **33% of BW** goes into **RHS**

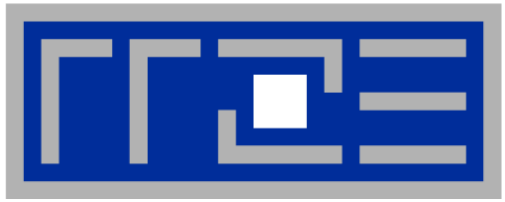


- **Conclusion: Even if the roofline model does not work 100%, we can still learn something from the deviations**

Input to the roofline model

… on the example of spMVM with HMeP matrix





Case study:
A 3D Jacobi smoother

The basics in two dimensions

Roofline performance analysis and modeling

A Jacobi smoother

- **Laplace equation in 2D:** $\Delta\Phi = 0$

- **Solve with Dirichlet boundary conditions using Jacobi iteration scheme:**

```
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1
t0 = 0 ; t1 = 1
do it = 1,itmax    ! choose suitable number of sweeps
  do k = 1,kmax
    do i = 1,imax
       ! four flops, one store, four loads
       phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
                      + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
    enddo
  enddo
  ! swap arrays
  i = t0 ; t0=t1 ; t1=i
enddo
```

Re-use when computing `phi(i+2,k,t1)`

WRITE ALLOCATE:
LD + ST `phi(i,k,t1)`

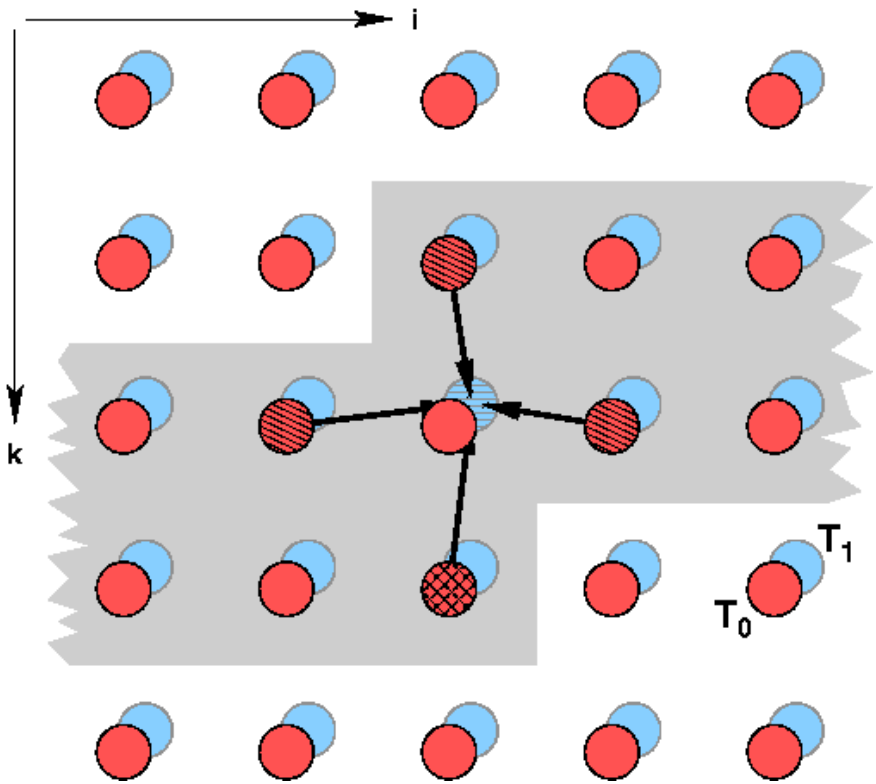
Naive balance (incl. write allocate):

`phi(:,:,t0)`: 3 LD +
`phi(:,:,t1)`: 1 ST+ 1LD

→ B_C = 5 W / 4 FLOPs = 1.25 W / F

Balance metric: 2 D Jacobi

- **Modern cache subsystems may further reduce memory traffic**
  - → **"layer conditions"**



If cache is large enough to hold at least 2 rows (shaded region): Each `phi(:,:,t0)` is loaded once from main memory and re-used 3 times from cache:

`phi(:,:,t0)`: 1 LD + `phi(:,:,t1)`: 1 ST+ 1LD
→B_C = 3 W / 4 F = 0.75 W / F

If cache is too small to hold one row:
`phi(:,:,t0)`: 2 LD + `phi(:,:,t1)`: 1 ST+ 1LD
→B_C = 5 W / 4 F = 1.25 W / F

Performance metrics: 2D Jacobi

- **Alternative implementation ("Macho FLOP version")**

```
do k = 1,kmax
  do i = 1,imax
     phi(i,k,t1) =  0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
                  + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
  enddo
enddo
```

- **MFlops/sec increases by 7/4 but time to solution remains the same**

- **Better metric (for many iterative stencil schemes):**
  Lattice Site Updates per Second (LUPs/sec)

  2D Jacobi example: Compute LUPs/sec metric via

$$P[LUPs/s] = \frac{it_{\max} \cdot i_{\max} \cdot k_{\max}}{T_{\text{wall}}}$$

2D → 3D

- **3D sweep:**

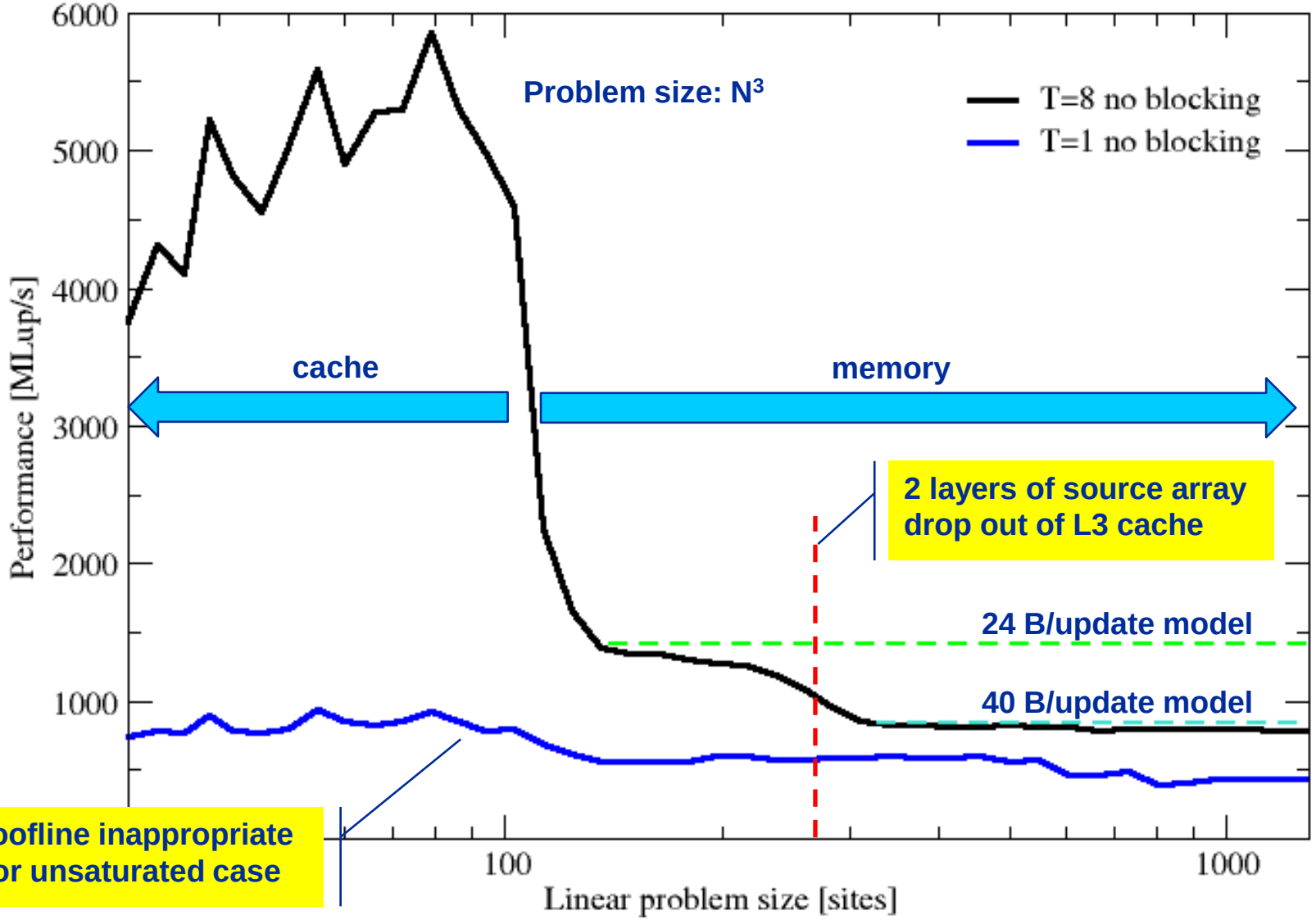
```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      phi(i,j,k,t1) = 1/6. *(phi(i-1,j,k,t0)+phi(i+1,j,k,t0) &
                           + phi(i,j-1,k,t0)+phi(i,j+1,k,t0) &
                           + phi(i,j,k-1,t0)+phi(i,j,k+1,t0))
    enddo
  enddo
enddo
```

- **Best case balance:** 1 LD (`phi(i,j,k+1,t0)`)
  1 ST + 1 write allocate (`phi(i,j,k,t1)`)
  6 flops
  → B_C = 0.5 W/F (24 bytes/LUP)

- **No 2-layer condition but 2 rows fit:** B_C = 5/6 W/F (40 bytes/LUP)
- **Worst case (2 rows do not fit):** B_C = 7/6 W/F (56 bytes/LUP)

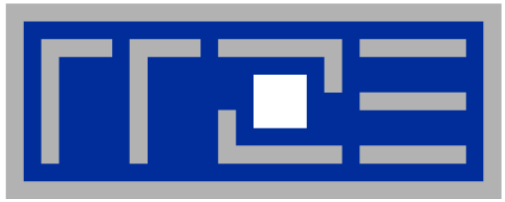
3D Jacobi solver

Performance of vanilla code on one Sandy Bridge chip (8 cores)



Conclusions from the Jacobi example

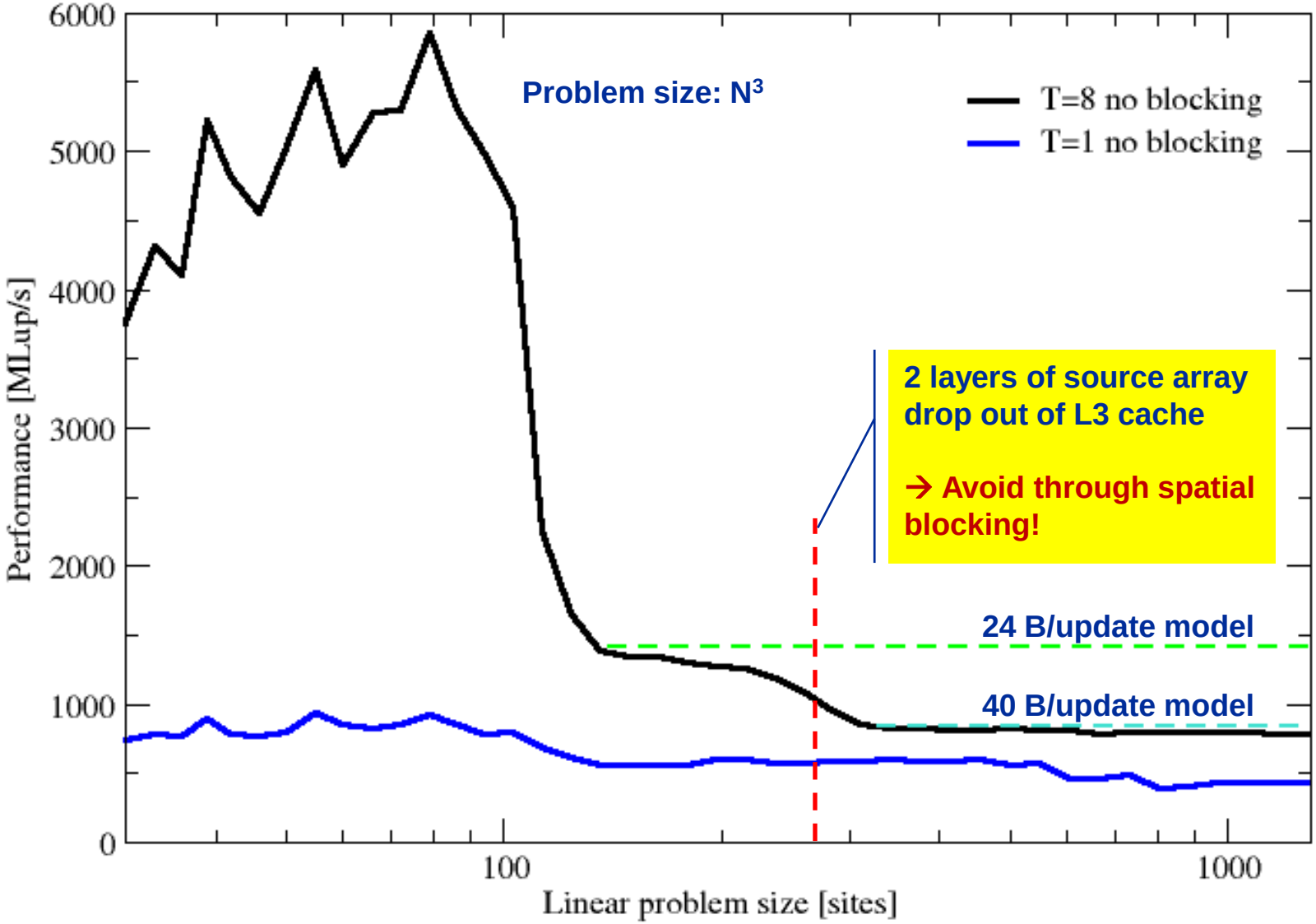
- **We have made sense of the memory-bound performance vs. problem size**
  - "Layer conditions" lead to predictions of code balance
  - Achievable memory bandwidth is input parameter
- **The model works only if the bandwidth is "saturated"**
  - In-cache modeling is more involved
- **Optimization == reducing the code balance by code transformations**
  - See below



Data access optimizations

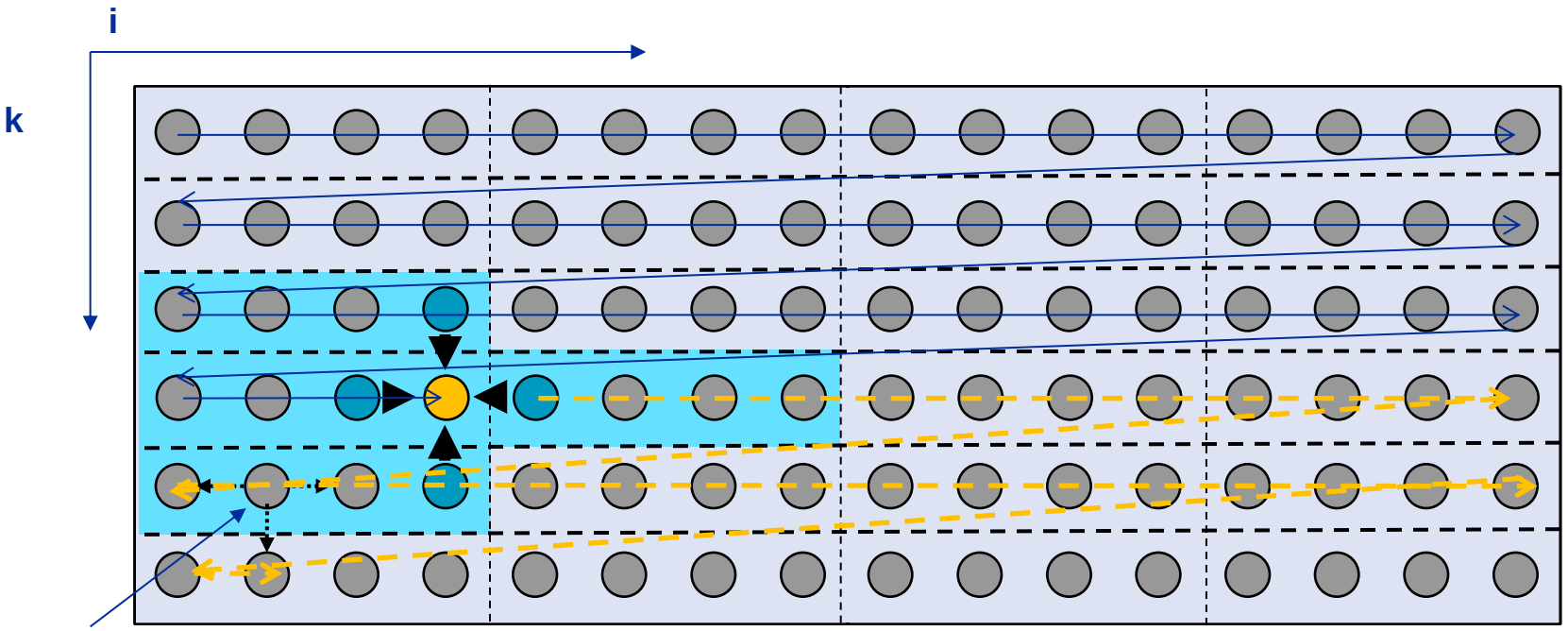
Case study: Optimizing the 3D Jacobi solver

Remember the 3D Jacobi solver on Sandy Bridge?



Jacobi iteration (2D): No spatial blocking

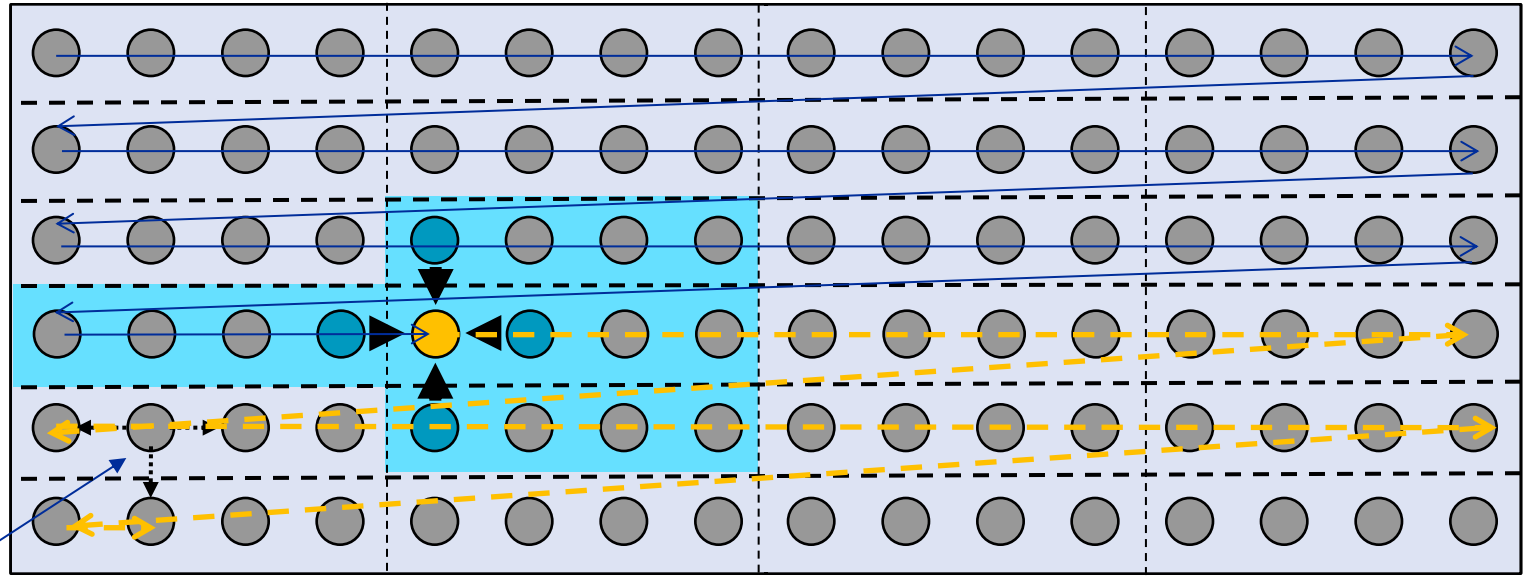
- **Assumptions:**
  - cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array



This element is needed for three more updates; but 29 updates happen before this element is used for the last time

Jacobi iteration (2D): No spatial blocking

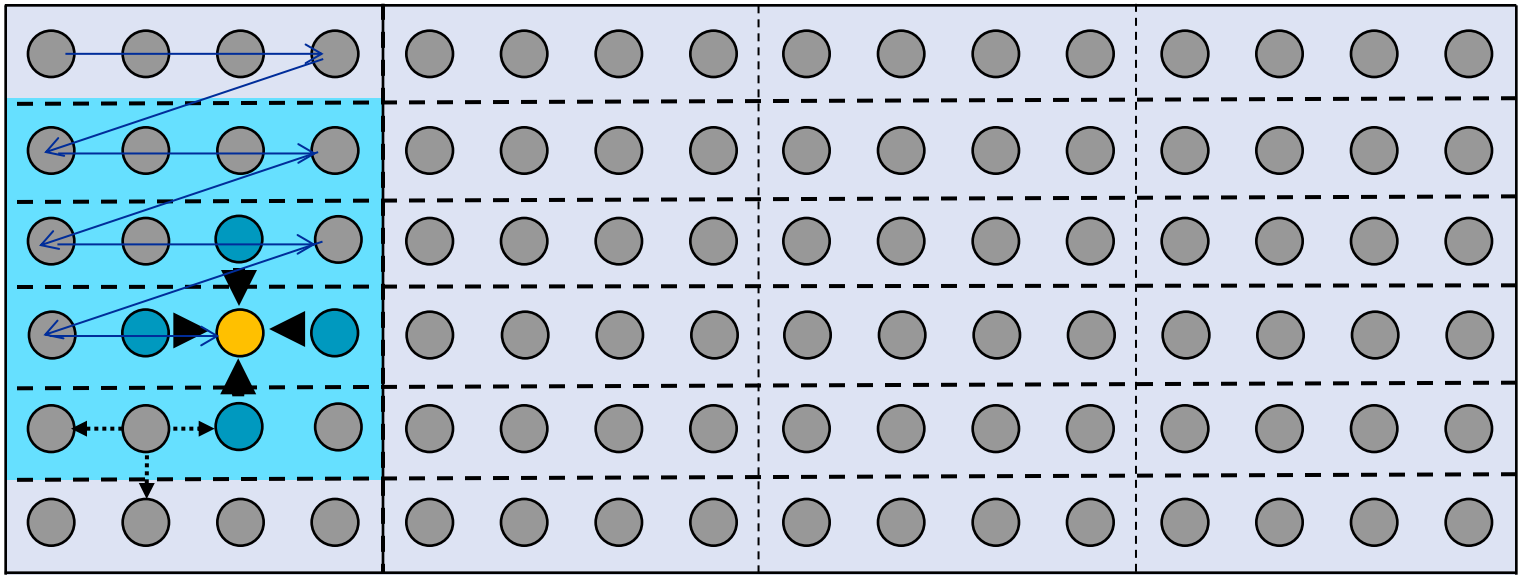
- **Assumptions:**
  - cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array



This element is needed for three more updates but has been evicted

Jacobi iteration (2D): Spatial blocking

- Divide system into blocks
- Update block after block
- Same performance as if three complete rows of the systems fit into cache



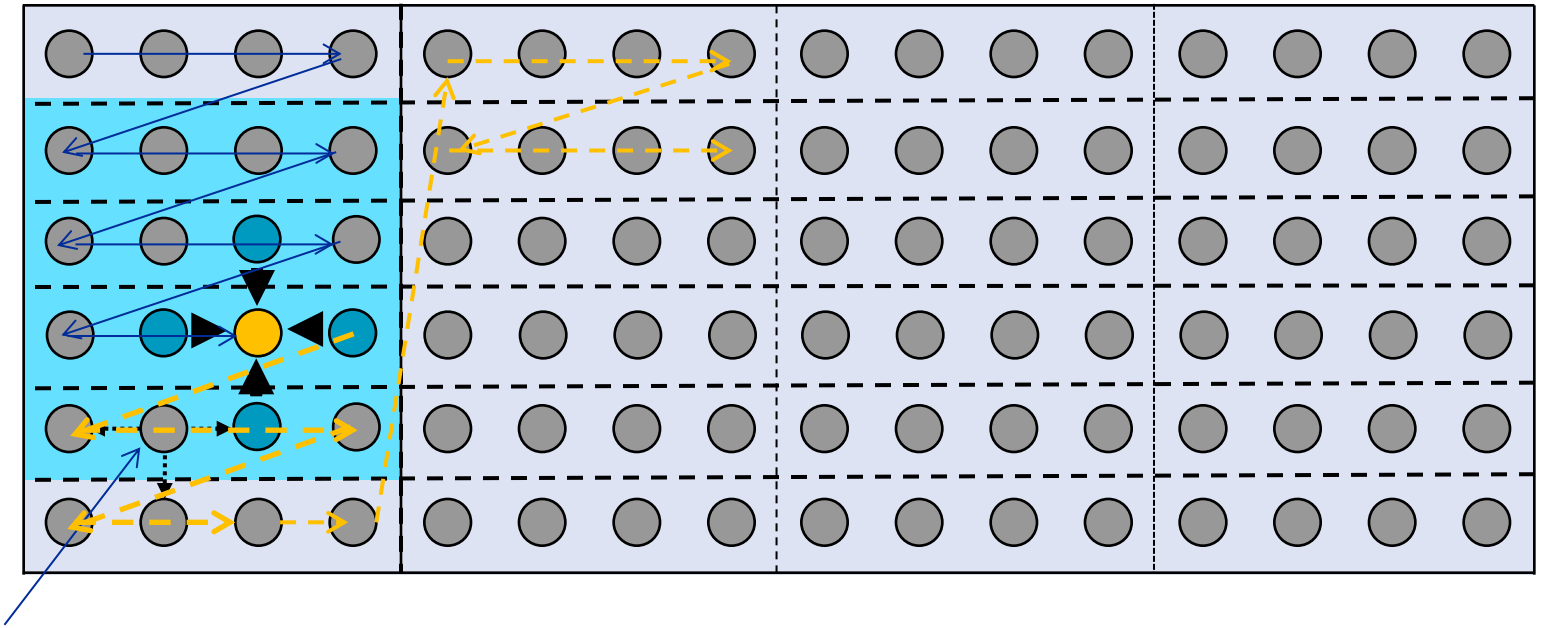
- Some excess traffic at boundaries may be unavoidable

Jacobi iteration (2D): Spatial blocking

- **Spatial blocking reorders traversal of data to account for the data update rule of the code**

→Elements stay sufficiently long in cache to be fully reused

→Spatial blocking improves temporal locality!
(Continuous access in inner loop ensures spatial locality)



This element remains in cache until it is fully used (only 6 updates happen before last use of this element)

Jacobi iteration (3D): Spatial blocking

- **Implementation:**

loop over i-blocks

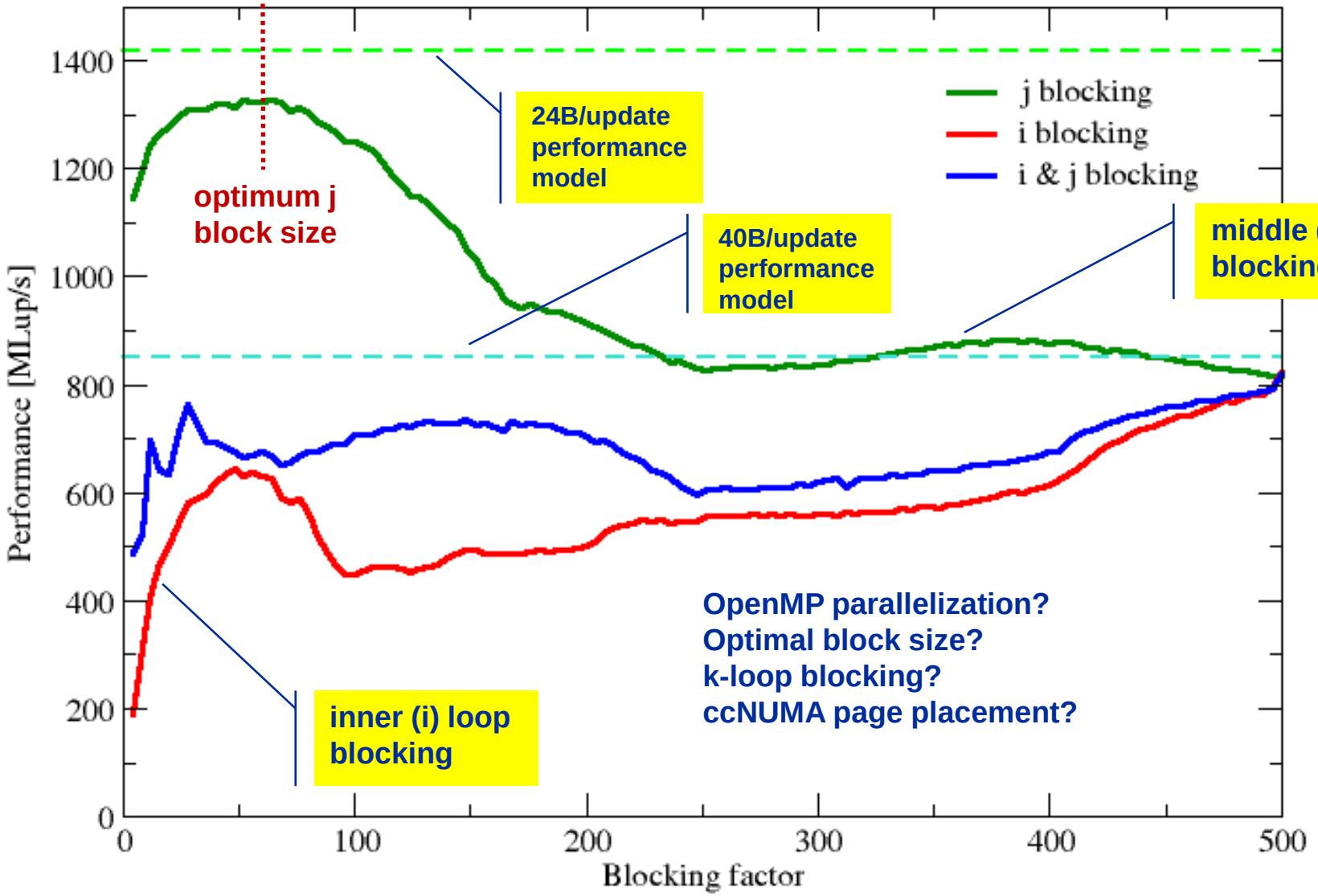
loop over j-blocks

```
do ioffset=1,imax,iblock
  do joffset=1,jmax,jblock
    do k=1,kmax
      do j=joffset, min(jmax,joffset+jblock-1)
        do i=ioffset, min(imax,ioffset+iblock-1)
          phi(i,j,k,t1) = ( phi(i-1,j,k,t0)+phi(i+1,j,k,t0)
                          + ... + phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )/6.d0
        enddo
      enddo
    enddo
  enddo
enddo
```

- **Guidelines:**
  - Blocking of inner loop levels (traversing continuously through main memory)
  - Blocking sizes large enough to fulfill “layer condition”
  - Cache size is a hard limit!
  - Blocking loops may have some impact on ccNUMA page placement

3D Jacobi solver (problem size 500^3)

Blocking different loop levels (8 cores Sandy Bridge)



Jacobi iteration (3D): Nontemporal stores

- **Intel x86: NT stores are packed SIMD stores with 16-byte aligned address**
  - Sometimes hard to apply
- **AMD x86: Scalar NT stores without alignment restrictions available**

- **Options for using NT stores**
  - Let the compiler decide → unreliable
  - Use compiler options
    - Intel: `-opt-streaming-stores never|always|auto`
  - Use compiler directives
    - Intel: `!DIR$ vector [non]temporal`
    - Cray: `!DIR$ LOOP_INFO cache[_nt](...)`
- **Compiler must be able to "prove" that the use of SIMD and NT stores is "safe"!**
  - "line update kernel" concept: Make critical loop its own subroutine

Jacobi iteration (3D): Nontemporal stores for Intel

Line update kernel (separate compilation unit or `-fno-inline`):

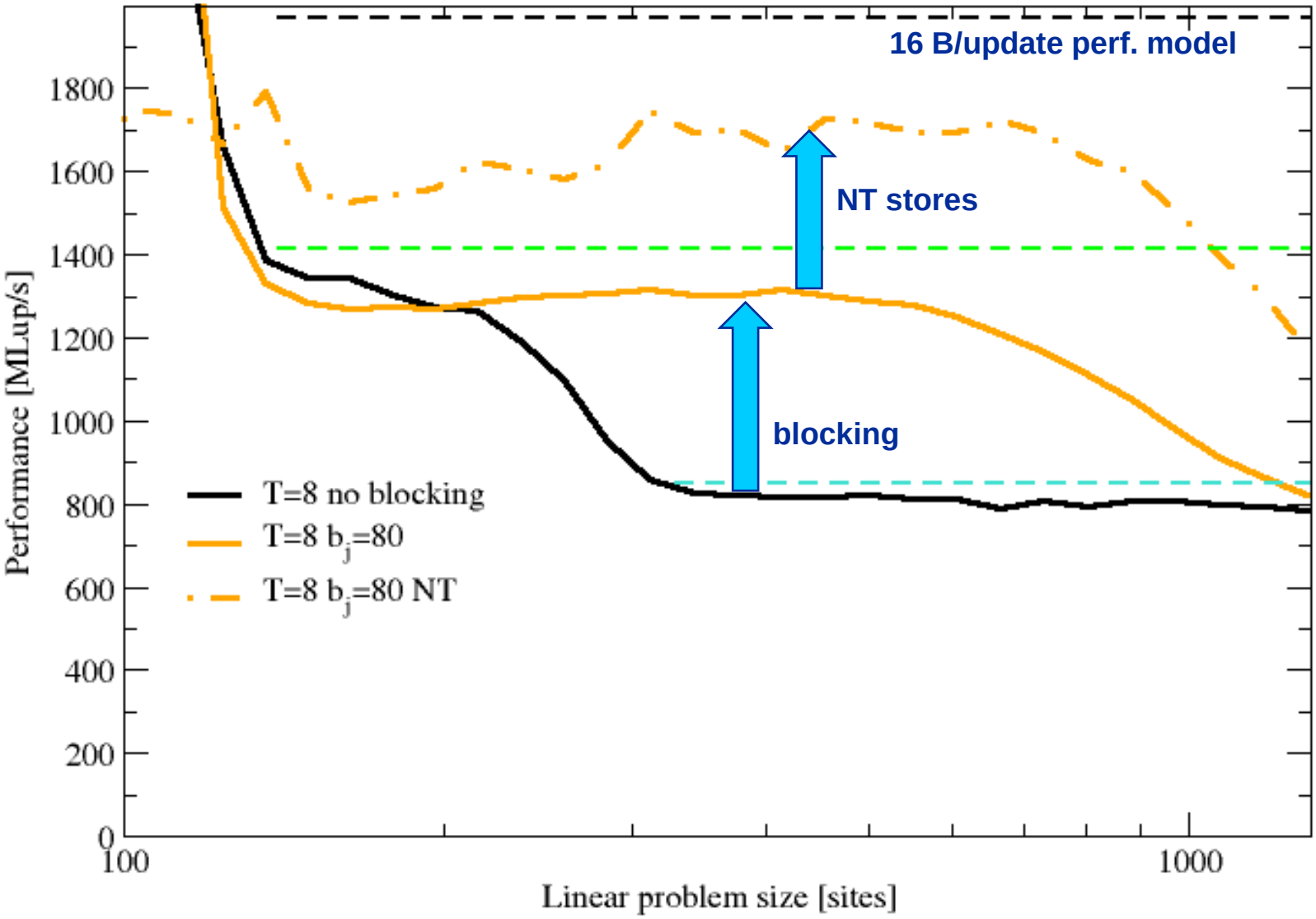
```
subroutine jacobi_line(d,s,top,bottom,front,back,n)
  integer :: n,i,start
  double precision, dimension(*) :: d,s,top,bottom,front,back
  double precision, parameter :: oos=1.d0/6.d0
!DEC$ VECTOR NONTEMPORAL
    do i=2,n-1
       d(i) = oos*(s(i-1)+s(i+1)+top(i)+bottom(i)+front(i)+back(i))
    enddo
end subroutine
```

Main loop:

```
do joffset=1,jmax,jblock
  do k=1,kmax
    do j=joffset, min(jmax,joffset+jblock-1)
      call jacobi_line(phi(1,j,k,t1),phi(1,j,k,t0),phi(1,j,k-1,t0), &
                       phi(1,j,k+1,t0),phi(1,j-1,k,t0),phi(1,j+1,k,t0)
                      ,size)

    enddo
  enddo
enddo
```

3D Jacobi solver

Spatial blocking + nontemporal stores



Conclusions from the Jacobi optimization example

- **“What part of the data comes from where” is a crucial question**
- **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”

Agenda

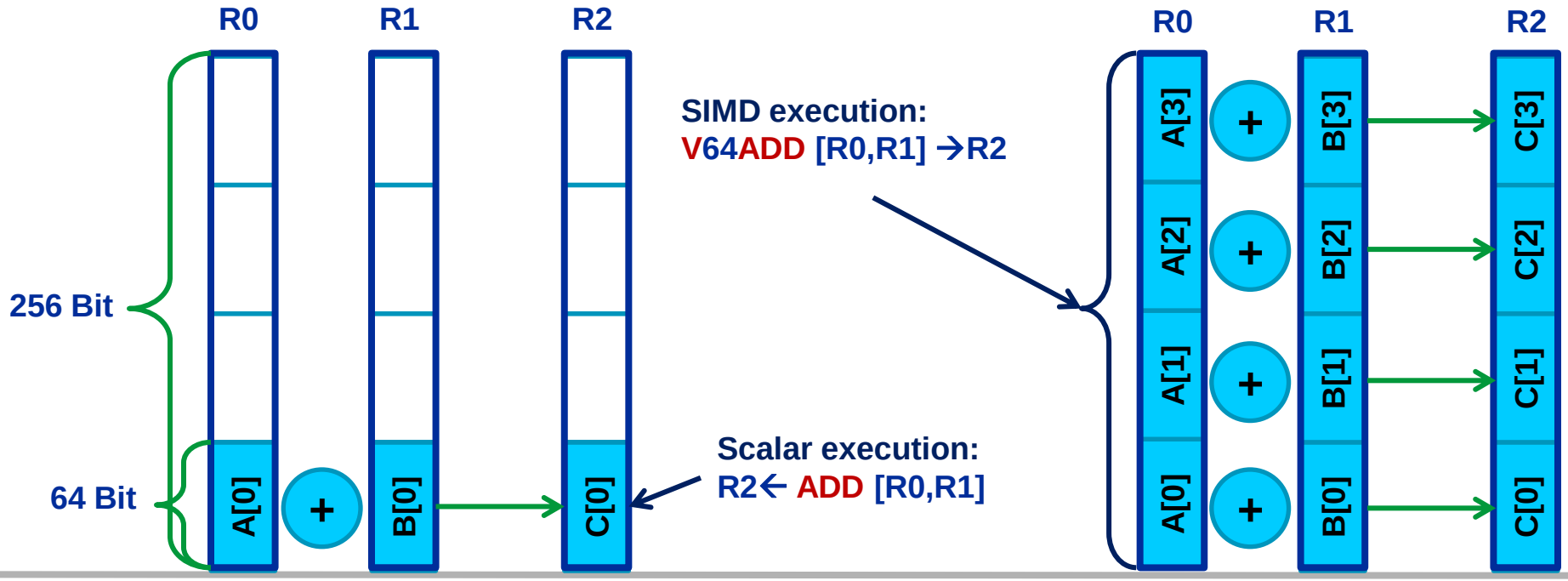
- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead
- **Node-level performance modeling**
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization
- **Optimal resource utilization**
  - SIMD parallelism
  - ccNUMA

Optimal utilization of parallel resources

Exploiting SIMD parallelism and reading assembly code
Programming for ccNUMA memory architecture

SIMD processing – Basics

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



SIMD processing – Basics

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

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

"Loop unrolling"

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

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

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

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

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

SIMD processing – Basics

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

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

- **"Pointer aliasing" may prevent SIMDfication**

```
void 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` → `&C[-1]` and `B` → `&C[-2]`
    → `C[i] = C[i-1] + C[i-2]`: **dependency → No SIMD**

  - **If "pointer aliasing" is not used, tell it to the compiler, e.g. use `-fno-alias` switch for Intel compiler → SIMD**

Reading x86 assembly code and exploting 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

`ymm0-ymm15` AVX (256bit)

SIMD instructions are distinguished by:

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

Case Study: 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`

Instruction code:

```
401d08:   f3 0f 58 04 82      addss  xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01         add    rax,1
401d11:   39 c7               cmp    edi,eax
401d13:   77 f3               ja     401d08
```

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)
- `pmmintrin.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**

Vectorization compiler options

- **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

Rules for vectorizable loops

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

Vectorization source code directives

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

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

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

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

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

User mandated vectorization

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

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

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

- **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

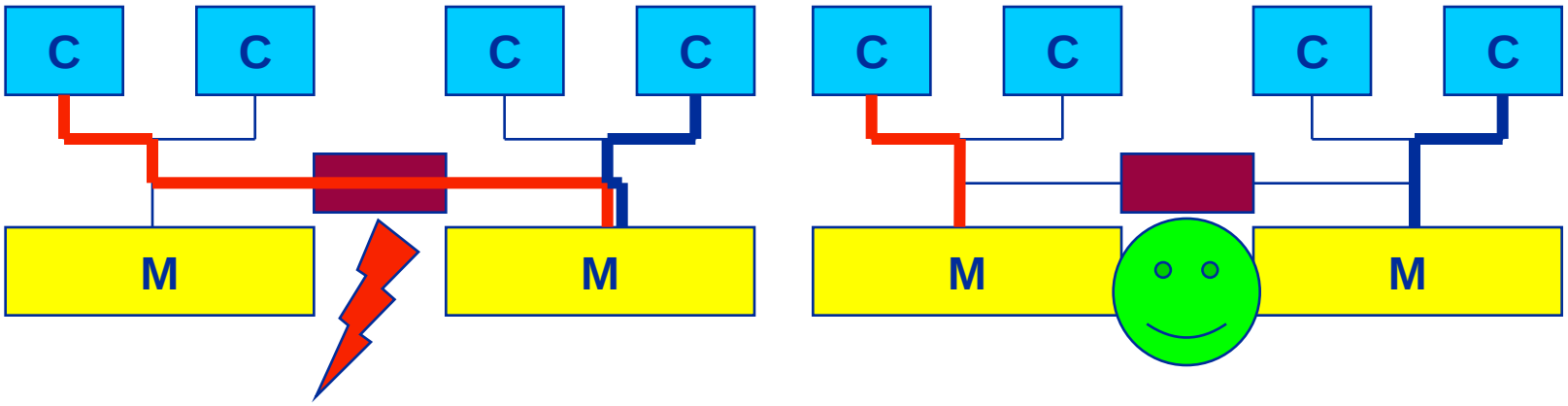
ccNUMA locality and dynamic scheduling

ccNUMA locality beyond first touch

ccNUMA performance problems

"The other affinity" to care about

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

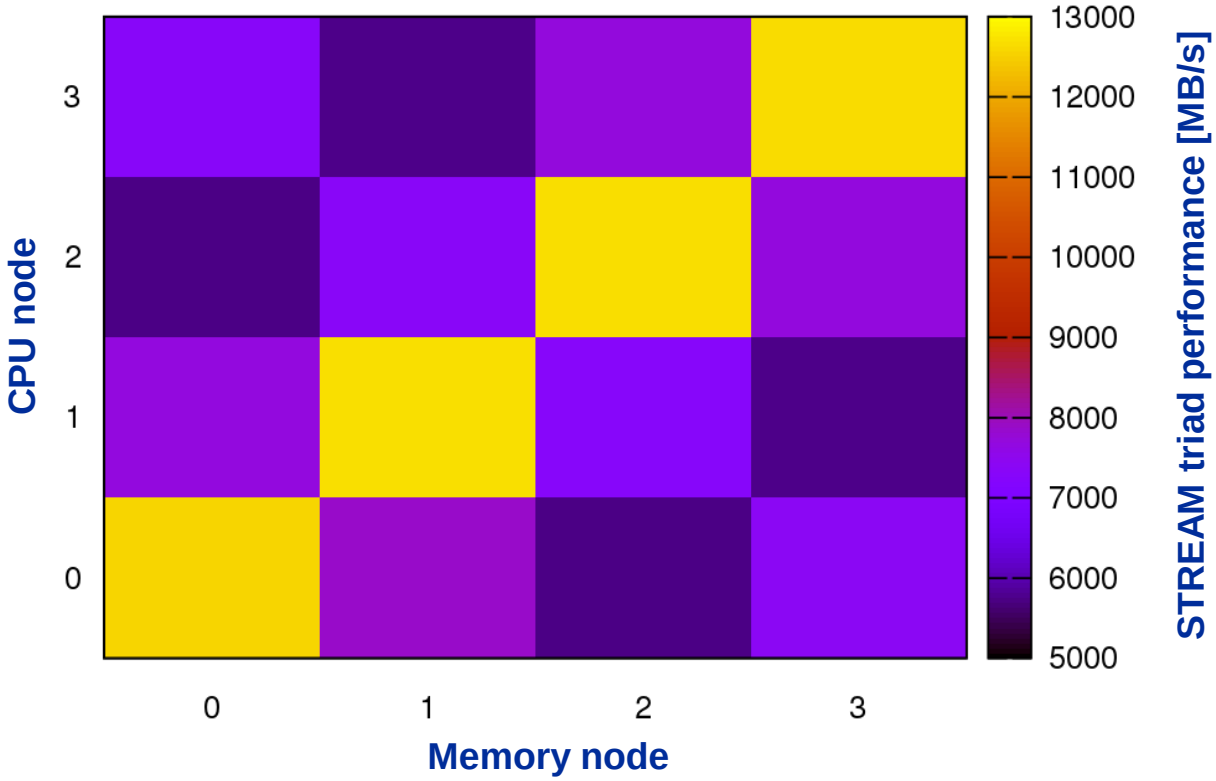


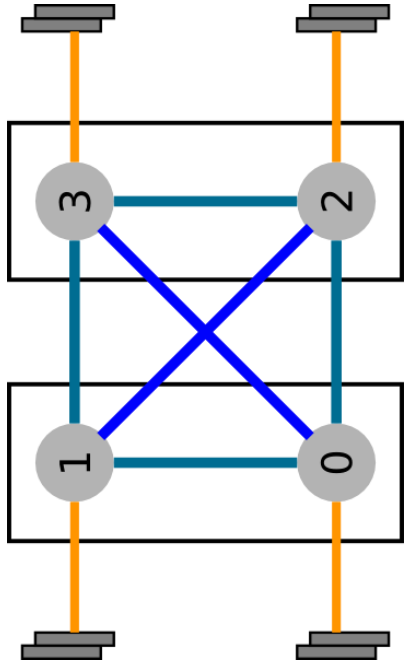
  - Page placement is implemented in units of OS pages (often 4kB, possibly more)

Cray XE6 Interlagos node

4 chips, two sockets, 8 threads per ccNUMA domain

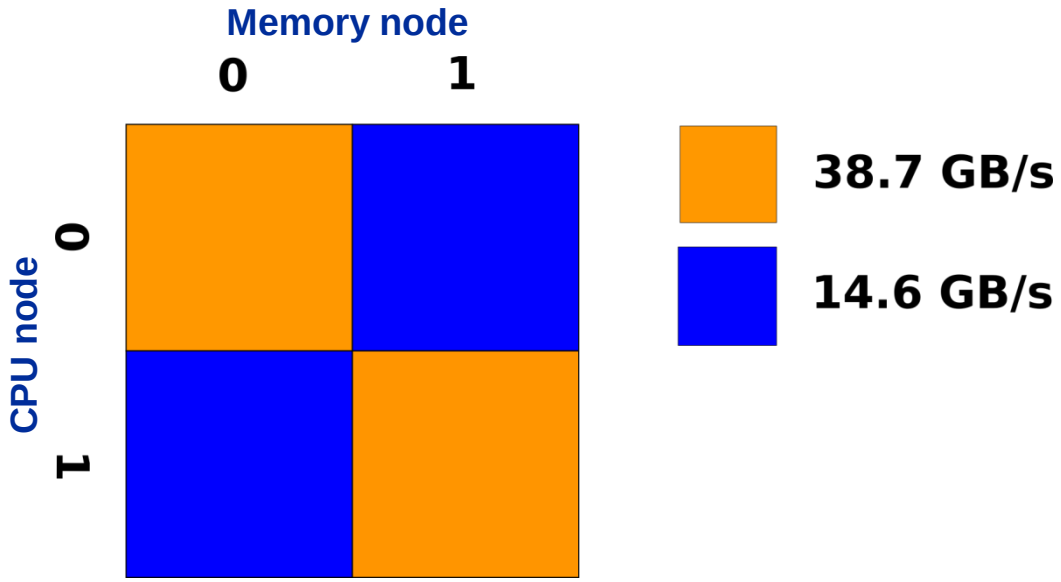
- **ccNUMA map: Bandwidth penalties for remote access**
  - Run 8 threads per ccNUMA domain (1 chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM triad benchmark using nontemporal stores





- **General rule:**

The more ccNUMA domains, the larger the non-local access penalty



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

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

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

- **Examples:**

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

ccNUMA map scan

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

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

ccNUMA default memory locality

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

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

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

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

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

Memory not mapped here yet

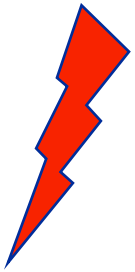
Mapping takes place here

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

Coding for ccNUMA data locality

- **Most simple case: explicit initialization**

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



A=0.d0



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

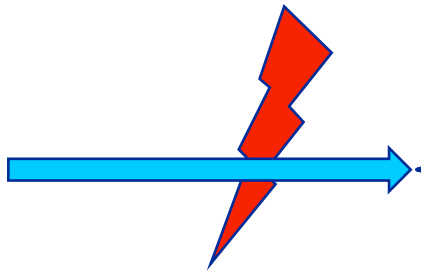


```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
...
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for ccNUMA data locality

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

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

READ(1000) A

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



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

Coding for Data Locality

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

Coding for Data Locality:
Placement of static arrays or arrays of objects

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

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

→ **placement problem with**

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

Coding for Data Locality:
Parallel first touch for arrays of objects

optional

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

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

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

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

parallel first touch

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

Coding for Data Locality:
NUMA allocator for parallel first touch in **`std::vector<>`**


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

Application:

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

Diagnosing Bad Locality

- **If your code is cache-bound, you might not notice any locality problems**
- **Otherwise, bad locality limits scalability at very low CPU numbers (whenever a node boundary is crossed)**
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…
- **Running with `numactl --interleave` might give you a hint**
  - See later
- **Consider using performance counters**
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel Westmere dual-socket system (Core i7, hex-core):

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

Using performance counters for diagnosing bad ccNUMA access locality

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

Only one memory BW per socket ("Uncore")

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

Half of BW comes from other socket!

If all fails…

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

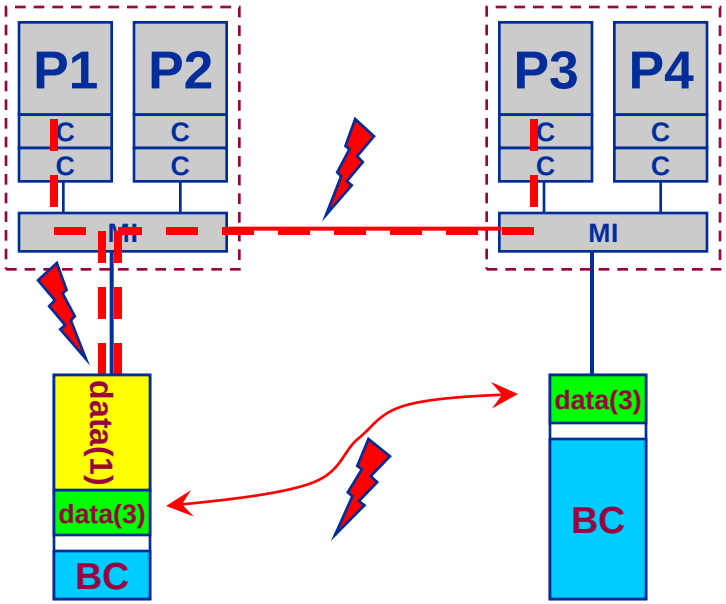
```
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

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

ccNUMA problems beyond first touch:
Buffer cache

optional

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



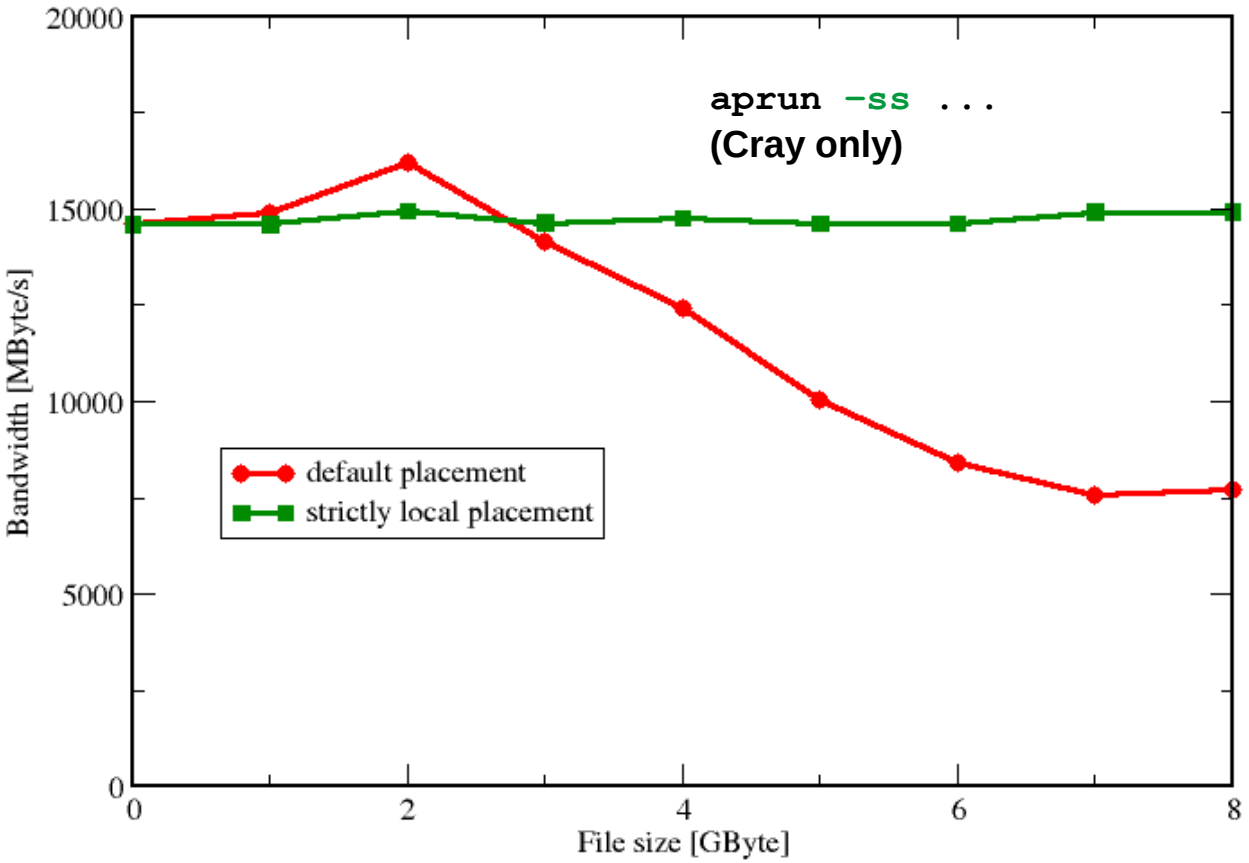
- **Remedies**
  - Drop FS cache pages after user job has run (admin's job)
    - seems to be automatic after aprun has finished on Crays
  - User can run "sweeper" code that allocates and touches all physical memory before starting the real application
  - **`numactl`** tool or **`aprun`** can force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

optional

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

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



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

→ restrict to local domain if possible!

ccNUMA placement and erratic access patterns

- **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**

ccNUMA placement and erratic access patterns

- **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!

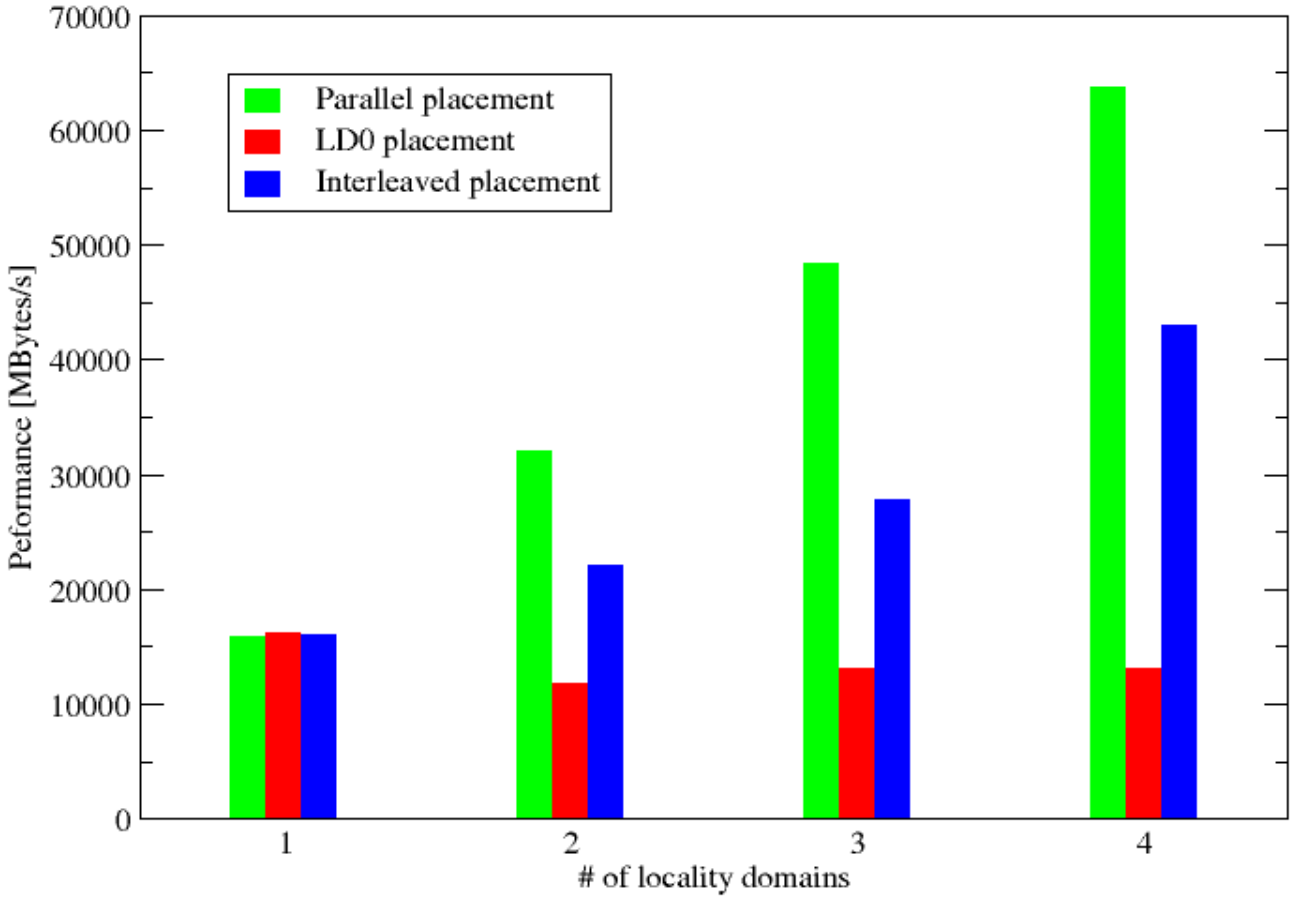
2. Using global control via `numactl`:

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

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

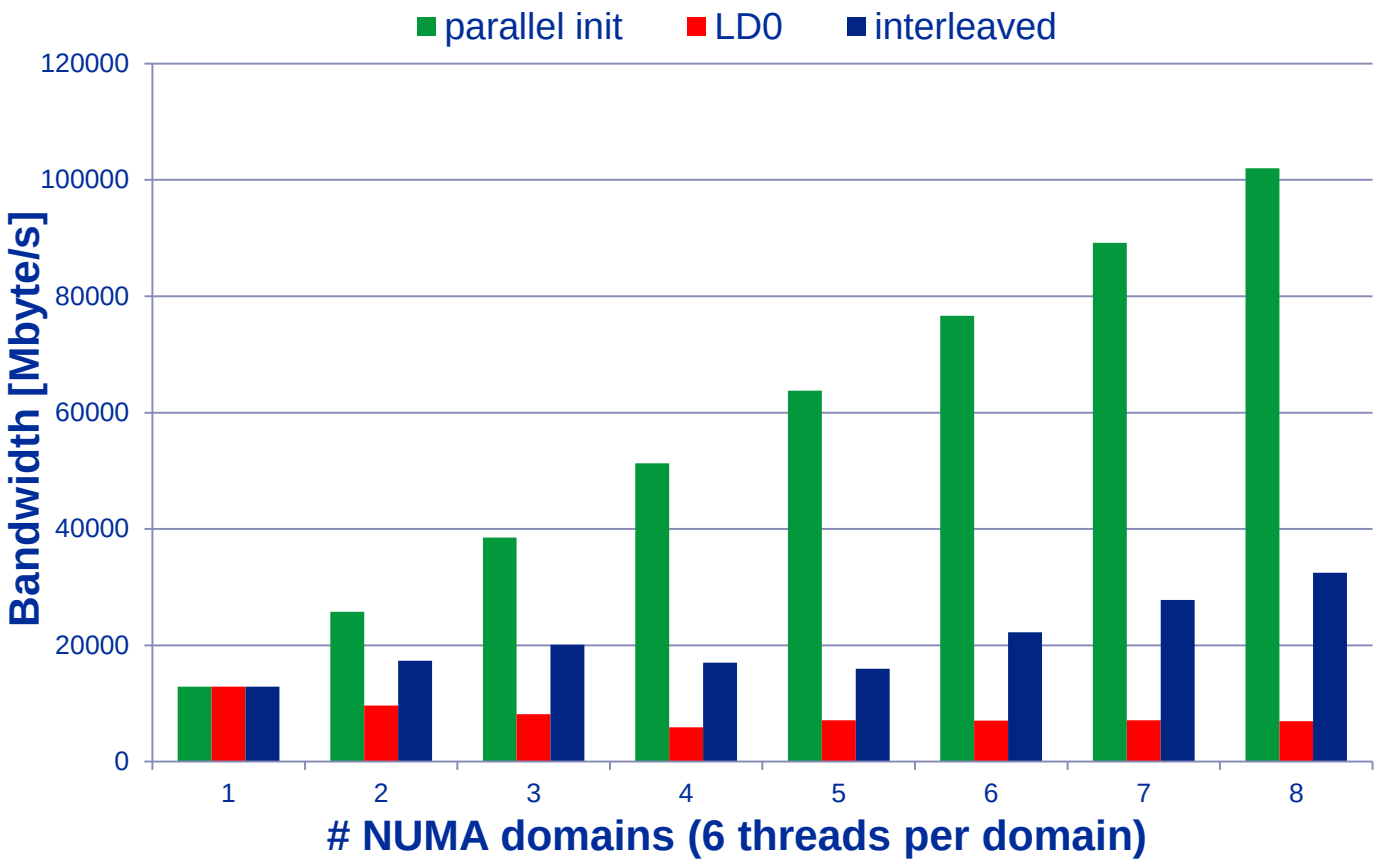
- **Fine-grained program-controlled placement via `libnuma` (Linux) using, e.g., `numa_alloc_interleaved_subset()`, `numa_alloc_interleaved()` and others**

- **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>`**



Summary on ccNUMA issues

- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with **`numactl --interleave`**
- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- **If dynamic scheduling cannot be avoided**
  - 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

DEMO

Tutorial conclusion

- **Multicore architecture == multiple complexities**
  - Affinity matters → pinning/binding is essential
  - Bandwidth bottlenecks → inefficiency is often made on the chip level
  - Topology dependence of performance features → know your hardware!
- **Put cores to good use**
  - Bandwidth bottlenecks → surplus cores → functional parallelism!?
  - Shared caches → fast communication/synchronization → better implementations/algorithms?
- **Simple modeling techniques 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., LIKWID tool suite

Quiz

Code:

```
double precision, dimension(100000000) :: a,b

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

GPGPU: 2880 cores, P_{peak}= 1.3 Tflop/s, b_S=160 Gbyte/s

Optimal performance?

Moritz Kreutzer
Markus Wittmann
Thomas Zeiser
Michael Meier

THANK YOU.

KONWIHR III
OMI4papps
HSMB

Bundesministerium für Bildung und Forschung
hpcADD
FEPA
SKALB

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.

References

Books:

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

Papers:

- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: A unified sparse matrix data format for modern processors with wide SIMD units. Submitted. Preprint: arXiv:1307.6209
- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Submitted. Preprint: arXiv:1208.2908
- J. Treibig, G. Hager and G. Wellein: Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. Preprint: arXiv: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
- 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

References

Papers continued:

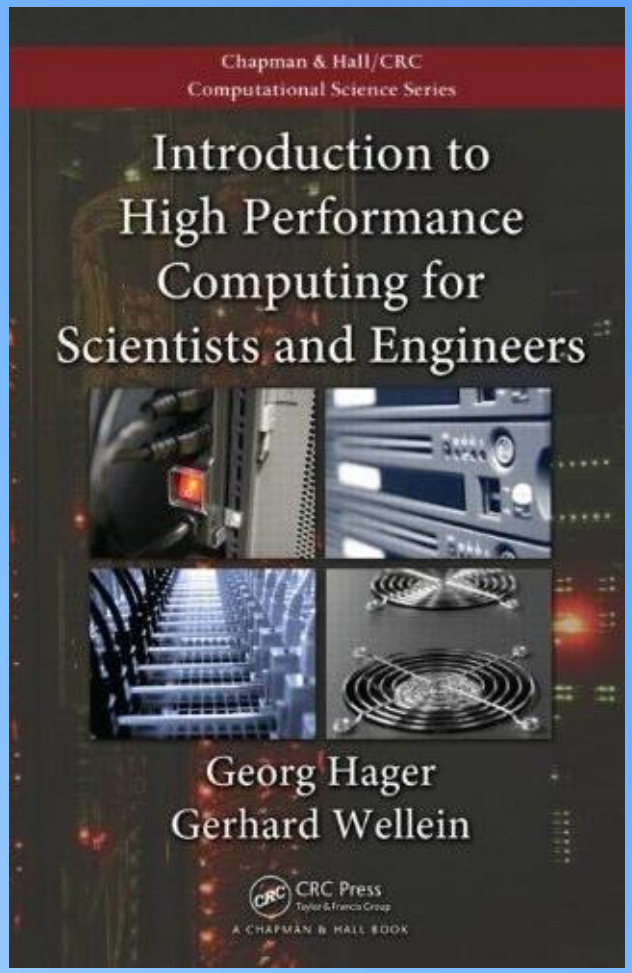
- G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. Proc. COMPSAC 2009.
  DOI: 10.1109/COMPSAC.2009.82
- M. Wittmann, G. Hager, J. Treibig and G. Wellein: Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. Parallel Processing Letters **20** (4), 359-376 (2010).
  DOI: 10.1142/S0129626410000296. Preprint: arXiv:1006.3148
- J. Treibig, G. Hager and G. Wellein: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010.
  DOI: 10.1109/ICPPW.2010.38. Preprint: arXiv:1004.4431
- G. Schubert, H. Fehske, G. Hager, and G. Wellein: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters 21(3), 339-358 (2011).
  DOI: 10.1142/S0129626411000254
- J. Treibig, G. Wellein and G. Hager: Efficient multicore-aware parallelization strategies for iterative stencil computations. Journal of Computational Science 2 (2), 130-137 (2011).
  DOI 10.1016/j.jocs.2011.01.010

References

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
- J. Treibig, G. Hager and G. Wellein: Multicore architectures: Complexities of performance prediction for Bandwidth-Limited Loop Kernels on Multi-Core Architectures. DOI: 10.1007/978-3-642-13872-0_1, Preprint: arXiv:0910.4865.
- 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:10.1177/10943420003017001005

HPC textbook



Georg Hager and Gerhard Wellein:

Introduction to High Performance Computing for Scientists and Engineers

CRC Press, ISBN 978-1439811924
356 pages
July 2010

"Georg Hager and Gerhard Wellein have developed a very approachable introduction to high performance computing for scientists and engineers. Their style and descriptions are easy to read and follow. … This book presents a balanced treatment of the theory, technology, architecture, and software for modern high performance computers and the use of high performance computing systems. The focus on scientific and engineering problems makes it both educational and unique. I highly recommend this timely book for scientists and engineers. I believe it will benefit many readers and provide a fine reference."

— From the Foreword by Jack Dongarra, University of Tennessee, Knoxville, USA

Georg Hager & Gerhard Wellein:

Introduction to High Performance Computing for Scientists and Engineers

- Covers basic sequential optimization strategies and the dominating parallelization paradigms, including shared-memory parallelization with OpenMP and distributed-memory parallel programming with MPI
- Highlights the importance of performance modeling of applications on all levels of a system's architecture
- Contains numerous case studies drawn from the authors' invaluable experiences in HPC user support, performance optimization, and benchmarking
- Explores important contemporary concepts, such as multicore architecture and affinity issues
- Includes code examples in Fortran and, if relevant, C and C++
- Provides end-of-chapter exercises with solutions in an appendix
- http://www.hpc.rrze.uni-erlangen.de/HPC4SE/

Introduction to High Performance Computing for Scientists and Engineers

Contents

Introduction to High Performance Computing for Scientists and Engineers

Contents continued