ERLANGEN REGIONAL
COMPUTING CENTER

The practitioner's cookbook for good parallel performance on multi- and many-core systems

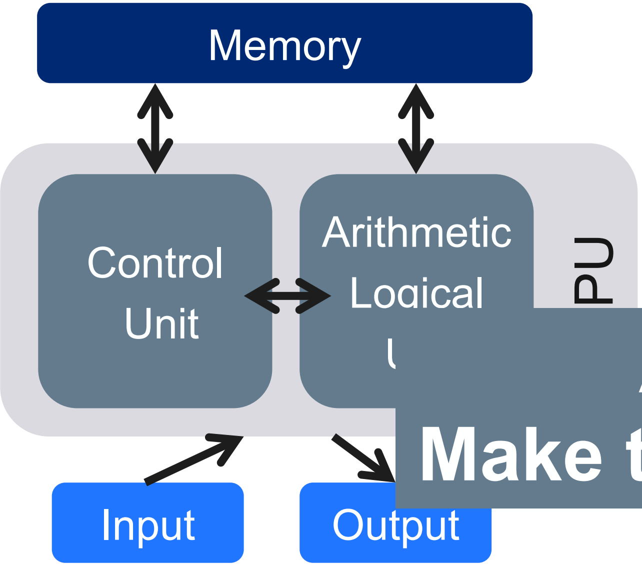
J. Treibig

PPOPP14, 16.2.2014

Schedule

Time	Topic
8:30am – 10:00am	Overview, Introduction to computer architecture
10:00am – 10:30am	Coffee break
10:30am – 12:00am	Performance Engineering, Micro-Benchmarking
12:00pm – 1:30pm	Lunch break
1:30pm – 3:00pm	Performance Modeling, SIMD, NUMA, SMT
3:00pm – 3:30pm	Coffee break
3:30 pm – 5:00pm	LIKWID tools, Accelerators, Case Studies

Where it all started: Stored Program Computer



Architect’s view:
Make the common case fast !

EDSAC 1949

Maurice Wilkes, Cambridge

- Provide improvements for **relevant** software
- What are the **technical** opportunities?
- **Economical** concerns
- Multi-way **special purpose**

Basic Resources: Instruction throughput and data movement

1. **Instruction execution**

   This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

2. **Data transfer bandwidth**

   Data transfers are a consequence of instruction execution and therefore a secondary resource.

Thinking in Bottlenecks

- A bottleneck is a performance limiting setting
- A microarchitecture exposes numerous bottlenecks

Observation 1:

Most applications face a single bottleneck at a time!

Observation 2:

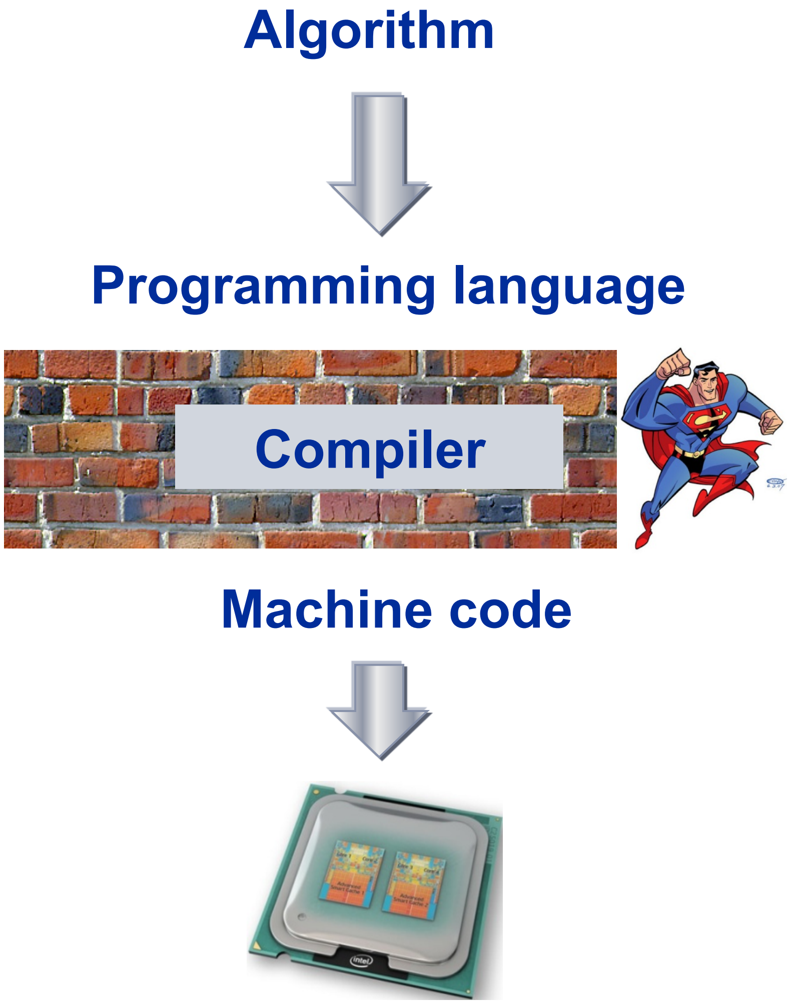
There is a limited number of relevant bottlenecks!

Hardware-Software Co-Design?
From algorithm to execution

Notions of work:

- Application Work
  - Flops
  - LUPS
  - VUPS

- Processor Work
  - Instructions
  - Data Volume



Algorithm

Programming language

Compiler

Machine code

Example: Threaded vector triad in C

Consider the following code:

```
#pragma omp parallel private(j)
{
for (int j=0; j<niter; j++) {
#pragma omp for
   for (int i=0; i<size; i++) {
      a[i] = b[i] + c[i] * d[i];
   } /* global synchronization */
}
}
```

Setup:

32 threads running on a dual socket 8-core SandyBridge-EP

gcc 4.7.0

Every single synchronization in this setup costs in the order of **60000 cycles** !

Why hardware should not be exposed

Such an approach is not portable …

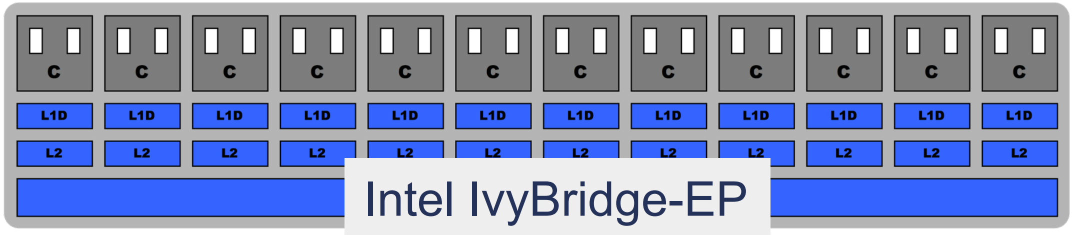
Hardware issues frequently change …

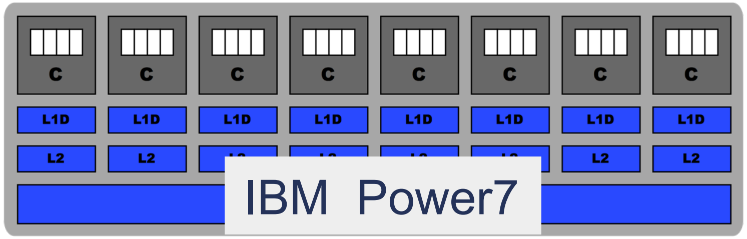
Those nasty hardware details are too difficult to learn for the average programmer …

Important fundamental concepts are stable and portable (ILP, SIMD, memory organization).

The basic principals are simple to understand and every programmer should know them.

The driving forces behind performance





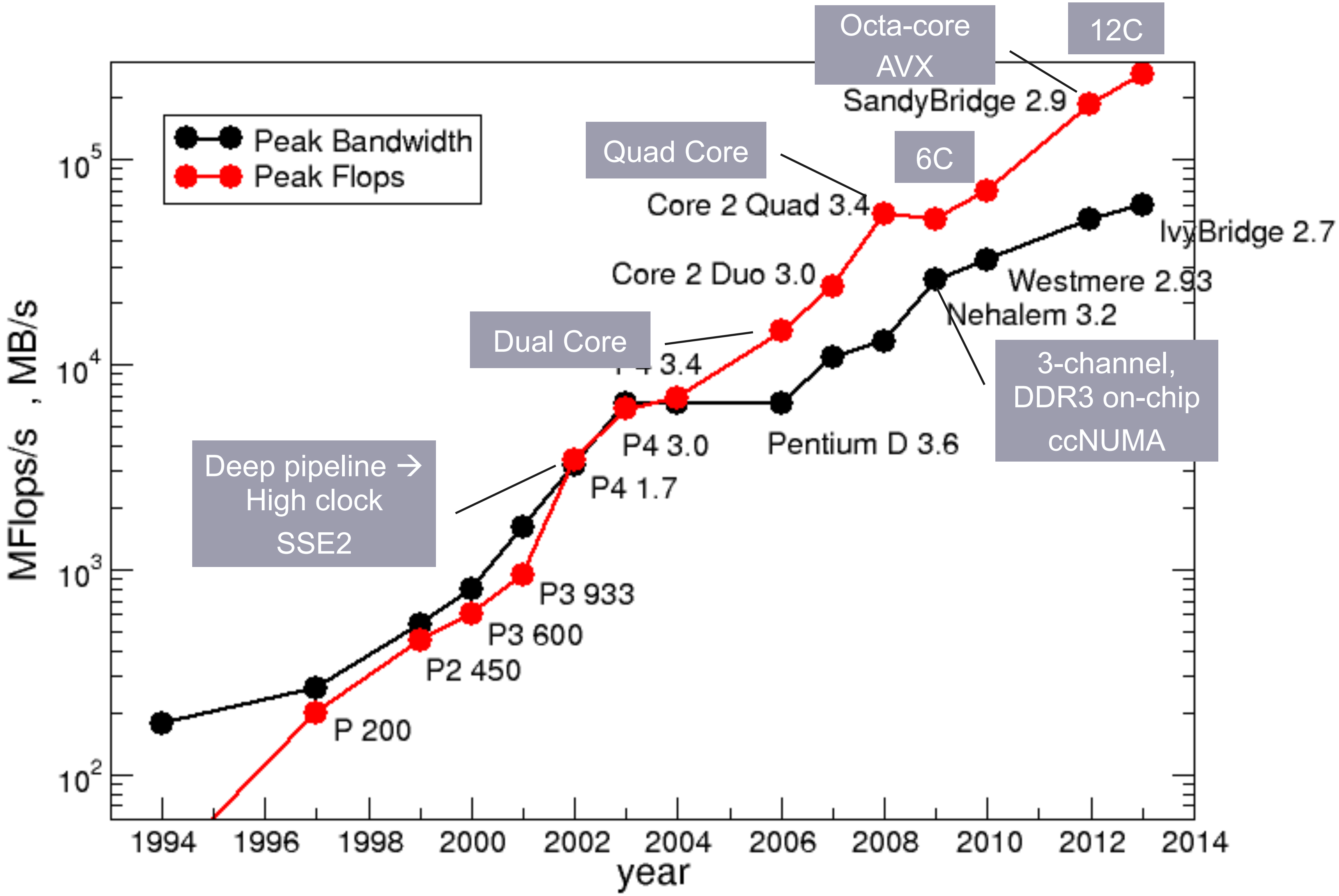
$$P = n_{core} * F * S * \nu$$

	Intel IvyBridge-EP	IBM Power7
Number of cores n_{core}	12	8
FP instructions per cycle F	2	2 (DP) / 1 (SP)
FP ops per instructions S	4 (DP) / 8 (SP)	2 (DP) / 4 (SP) - FMA
Clock speed [GHz] ν	2.7	3.7
Performance [GF/s] P	**259 (DP) / 518 (SP)**	**236 (DP/SP)**

TOP500 rank 1 (1996)

But: P=5.4 GF/s or 14.8 GF/s(dp) for serial, non-SIMD code

Timeline of technology developments



What needs to be done on one slide

- Reduce work
- Reduce data volume (over slow data paths)

- Make use of parallel resources
  - Load balancing
  - Serial fraction

- Identify relevant bottleneck(s)
  - Eliminate bottleneck
  - Increase resource utilization

Final Goal: Fully exploit offered resources for your specific code!

HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION

- ILP
- SIMD
- SMT
- Memory hierarchy

Common technologies

- Instruction Level Parallelism (**ILP**)
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution

Cycle Pipeline latency

Stages

Bubbles Wind-up

Wind-down

CPI

Scheduler

Hazard

- Memory Hierarchy

Caches Write allocate

Temporal locality Cache-line

- Branch Prediction Unit, Hardware Prefetching

Speculative execution

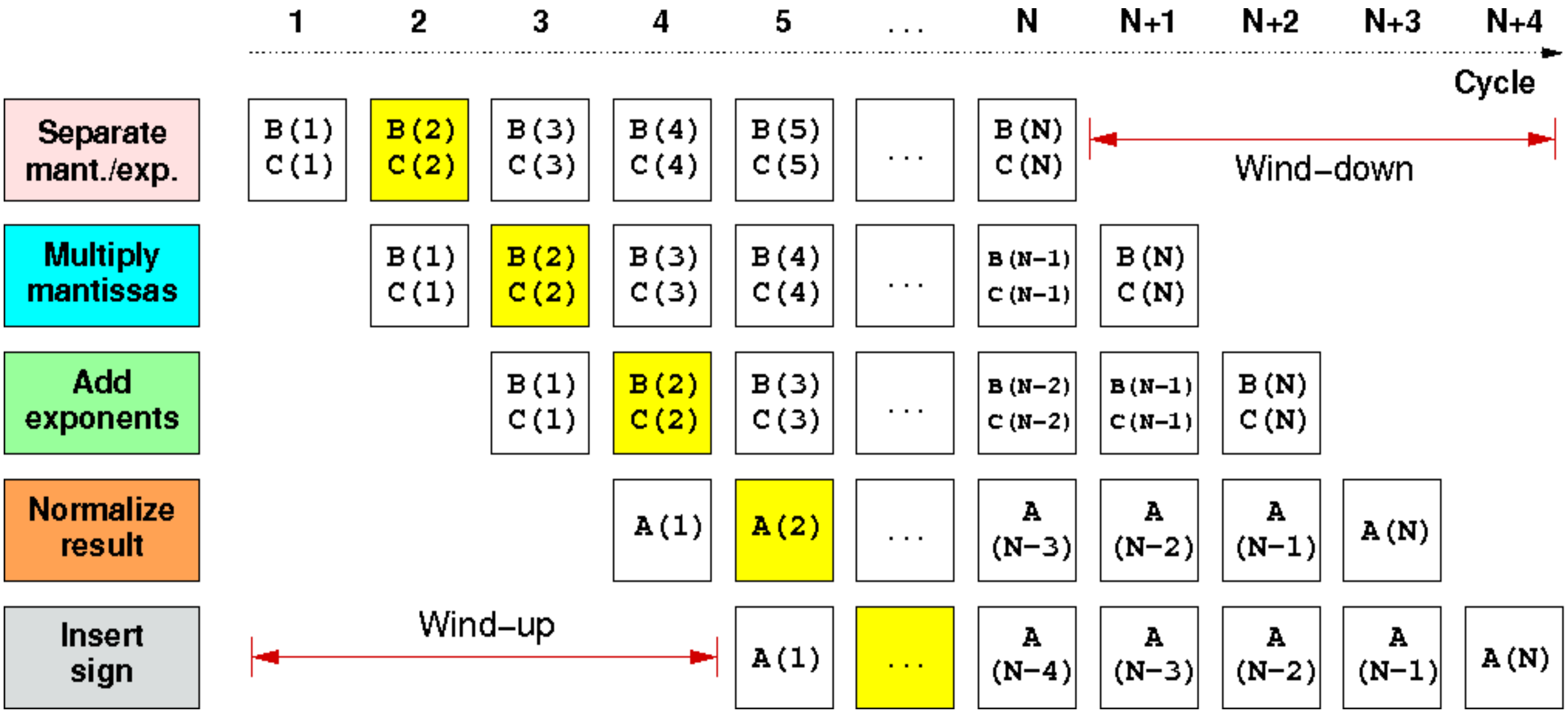
- Single Instruction Multiple Data (**SIMD**)

Lanes Register width

Packed Scalar

- Simultaneous Multithreading (**SMT**)

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



Cycle

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

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

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

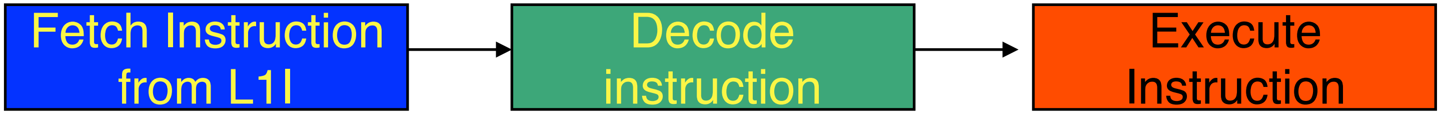
Add exponents			B(1)
			C(1)
				B(2)
				C(2)
					B(3)
					C(3)
						...	B(N-2)
							C(N-2)
								B(N-1)
								C(N-1)
									B(N)
									C(N)		

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

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

Pipelining: The Instruction pipeline

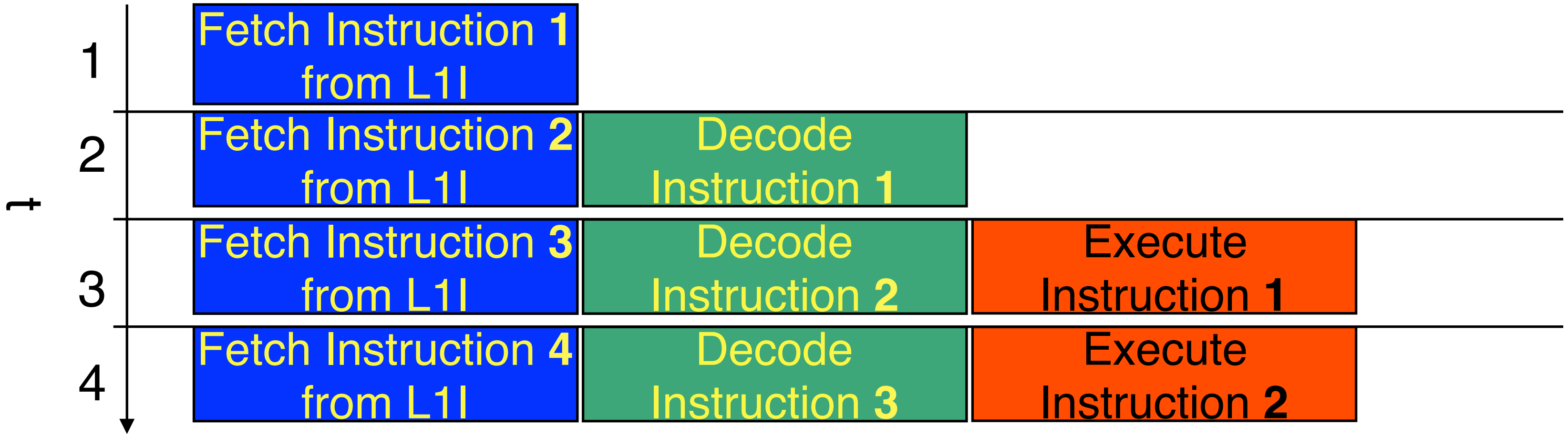
Besides ALUs, instruction execution itself is also pipelined:



Fetch Instruction from L1I

Decode instruction

Execute Instruction



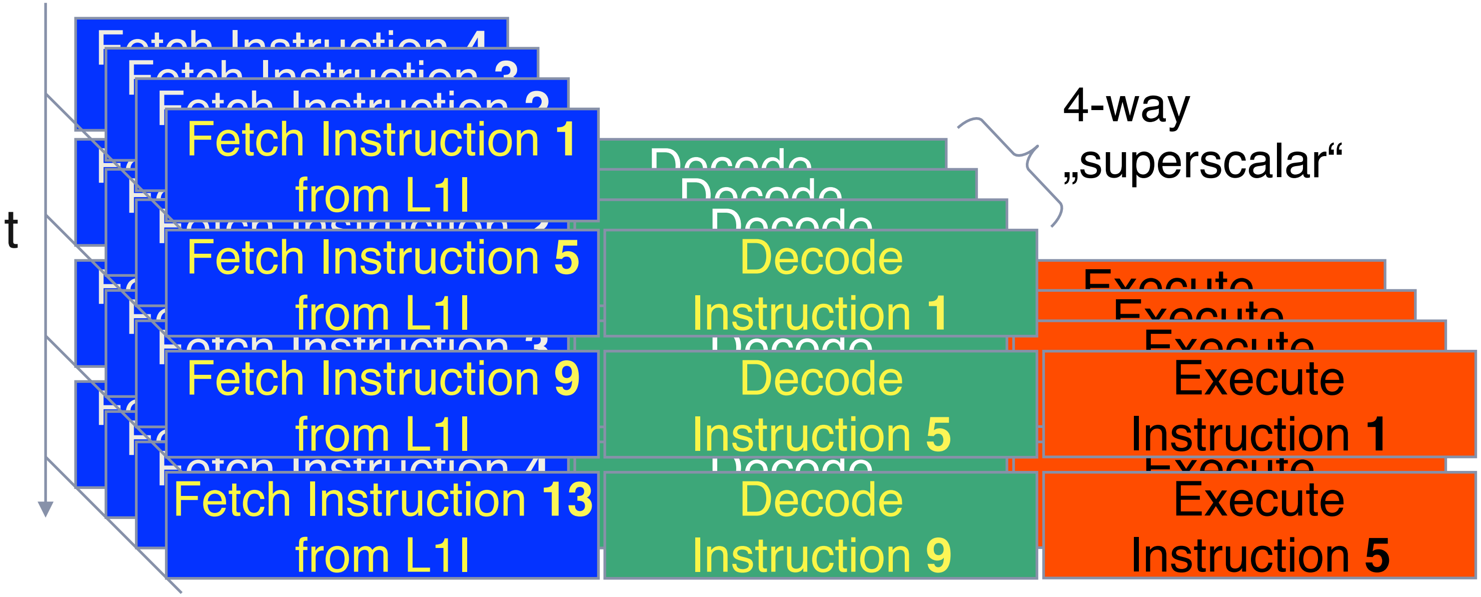
t			
1	Fetch Instruction **1** from L1I		
2	Fetch Instruction **2** from L1I	Decode Instruction **1**	
3	Fetch Instruction **3** from L1I	Decode Instruction **2**	Execute Instruction **1**
4	Fetch Instruction **4** from L1I	Decode Instruction **3**	Execute Instruction **2**

…

Each unit is pipelined itself (e.g., Execute = Multiply Pipeline).

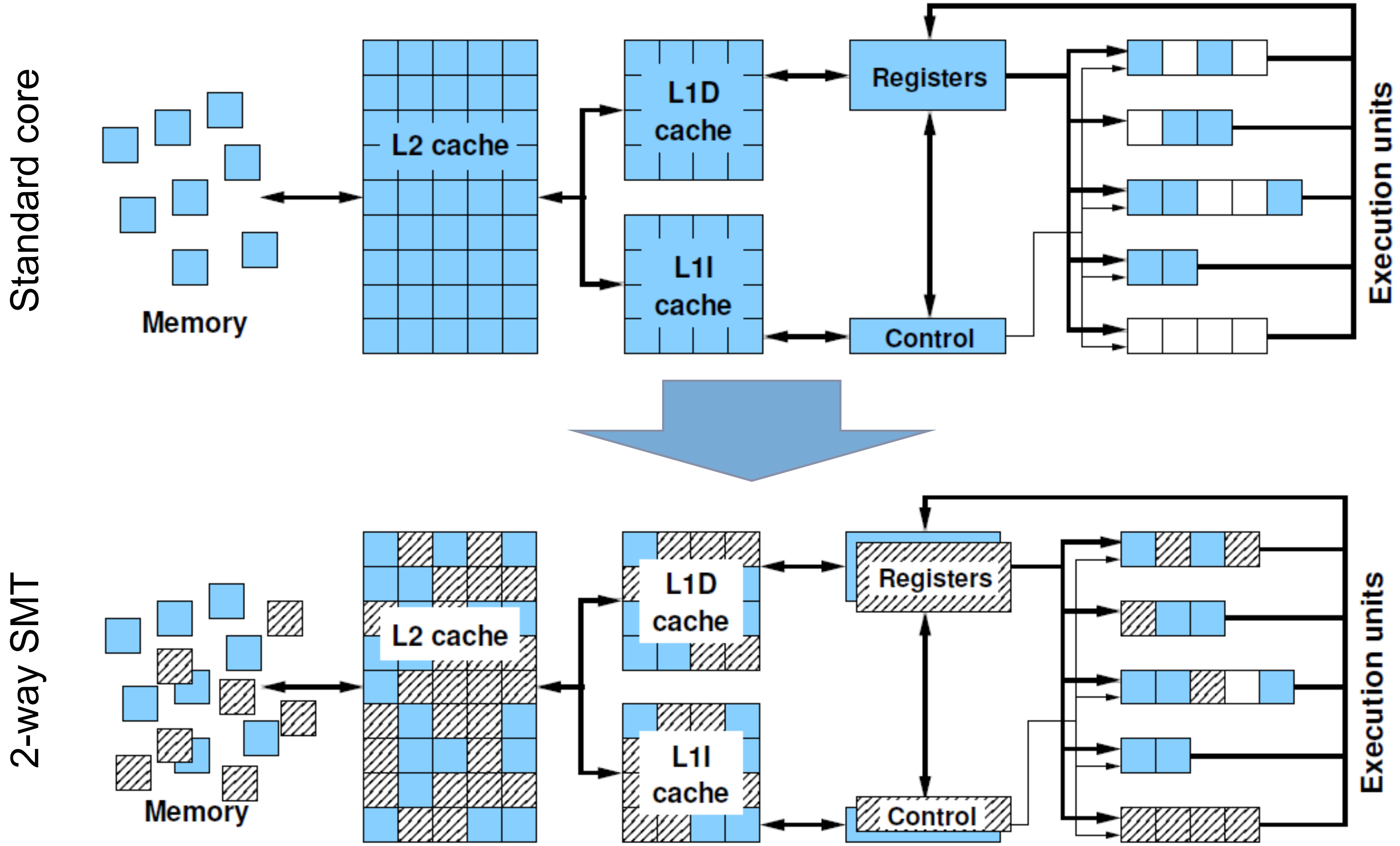
Superscalar Processors
Instruction Level Parallelism

Multiple units enable to "parallelize" the sequential instruction stream on the fly



Modern processors are 3- to 6-way superscalar

Core details: Simultaneous multi-threading (SMT)

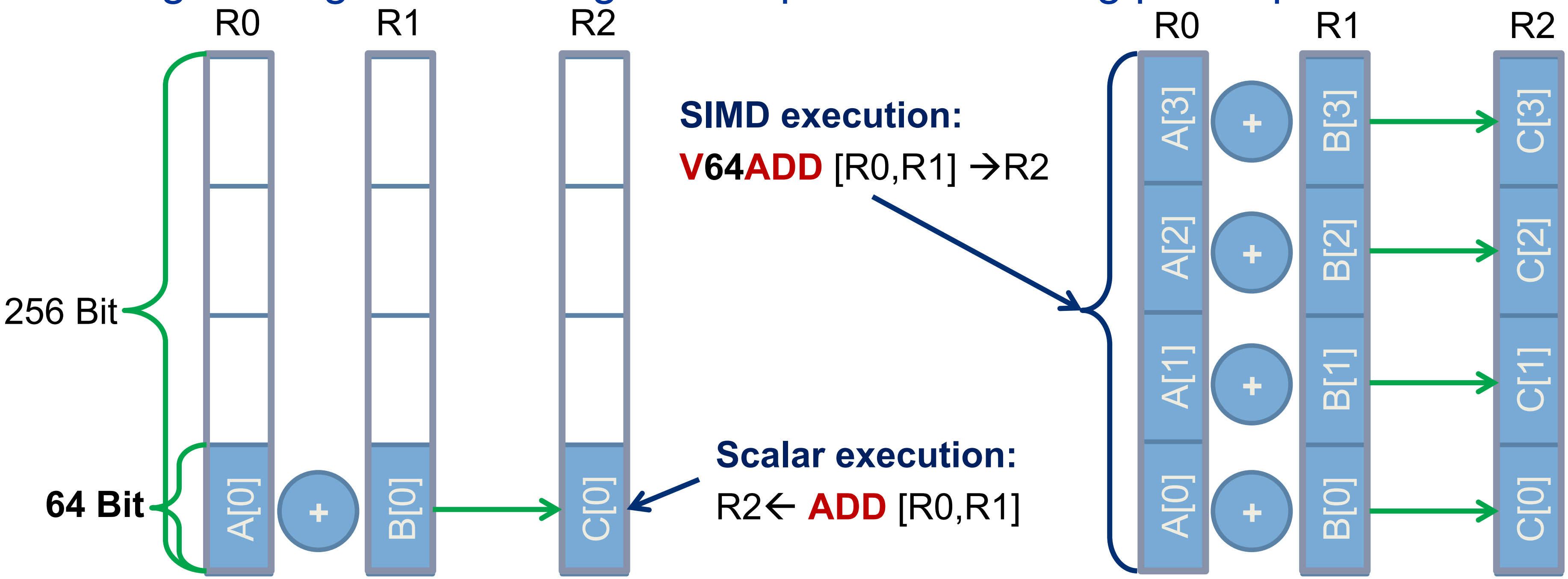


Core details: SIMD processing

Single Instruction Multiple Data (SIMD) allows the concurrent execution of the same operation on “wide” registers.

- SSE: register width = 128 Bit → 2 DP floating point operands
- AVX: register width = 256 Bit → 4 DP floating point operands

Adding two registers holding double precision floating point operands



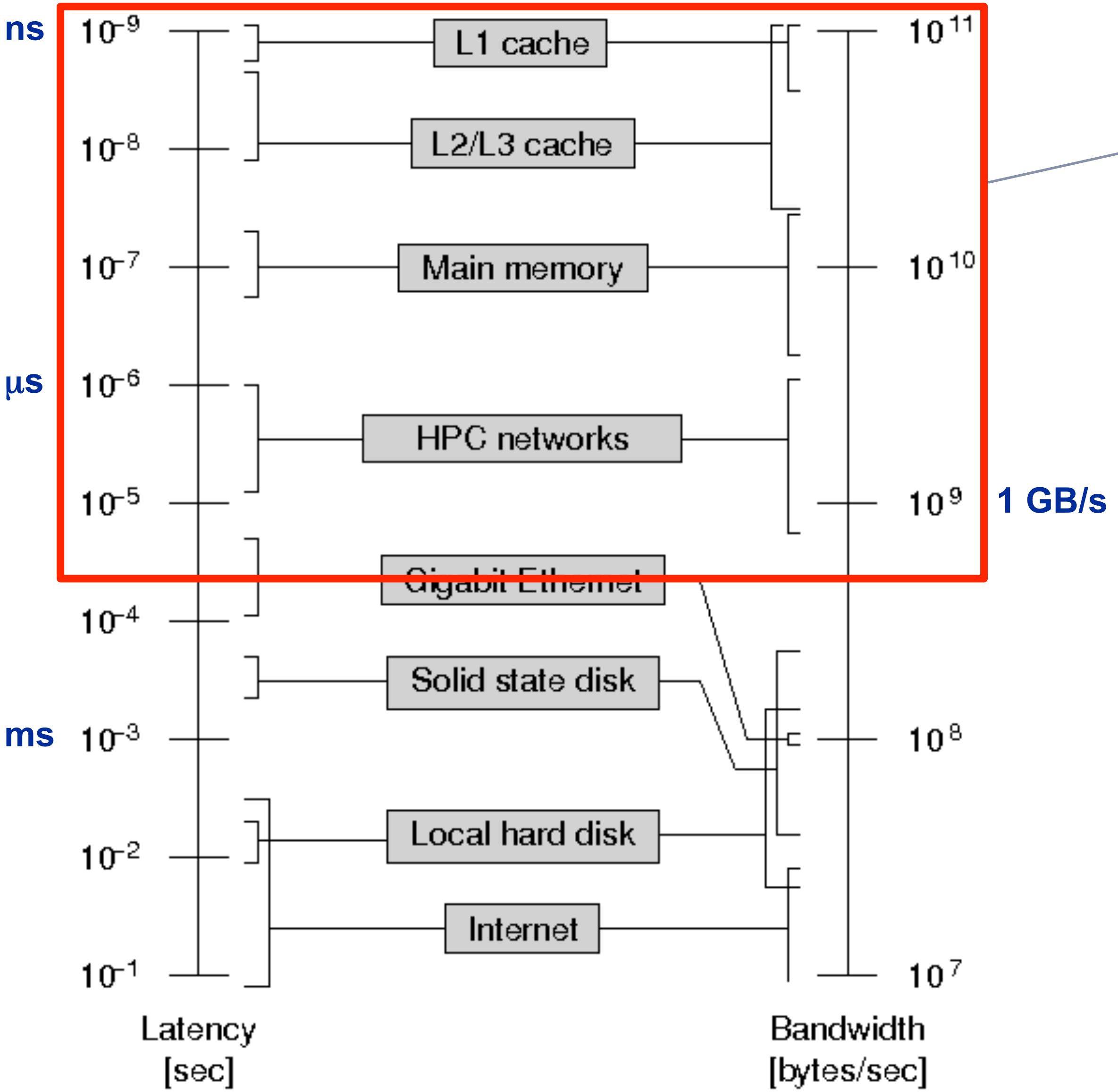
SIMD execution:

V64ADD [R0,R1] →R2

Scalar execution:

R2← **ADD** [R0,R1]

Latency and bandwidth in modern computer environments



HPC plays here

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

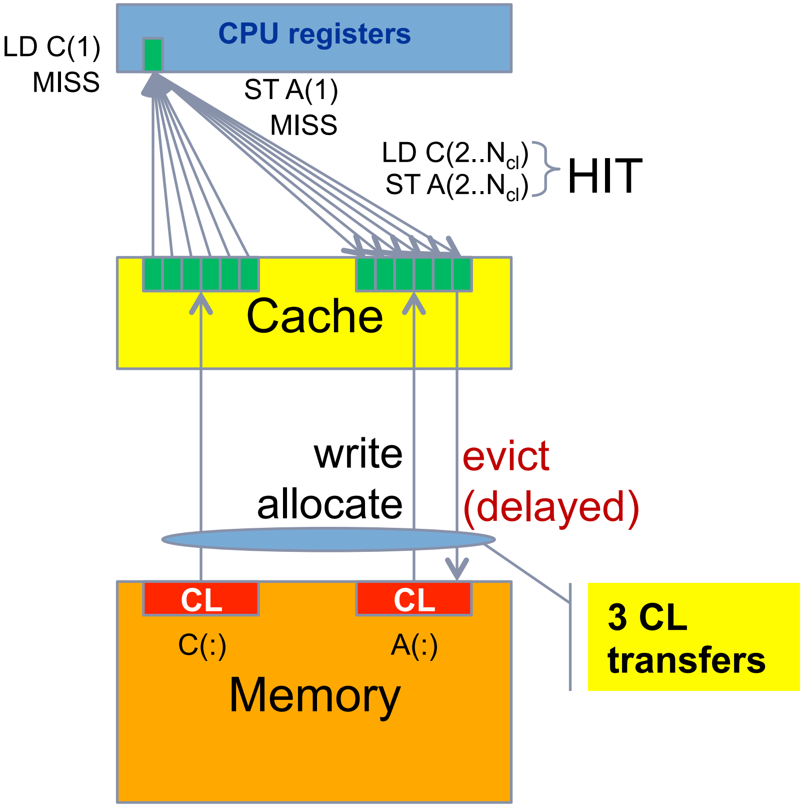
Registers and caches: Data transfers in a memory hierarchy

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 data in a cache level
→ CL transfer required

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



Consequences for data structure layout

- Promote temporal and spatial locality
- Enable packed (block wise) load/store of data
- Memory locality (placement)
- Avoid false cache line sharing
- Access data in long streams to enable efficient latency hiding

Above requirements may collide with object oriented programming paradigm: **array of structures** vs **structure of arrays**

Conclusions about core architectures

- All efforts are targeted on increasing **instruction throughput**
- Every hardware optimization puts an **assumption** against the executed software
- One can distinguish transparent and **explicit** solutions

- Common technologies:
  - Instruction level parallelism (**ILP**)
  - Data parallel execution (**SIMD**), does not affect instruction throughput
  - Exploit temporal data access locality (**Caches**)
  - Hide data access latencies (**Prefetching**)
  - Avoid hazards

PRELUDE:
SCALABILITY 4 THE 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

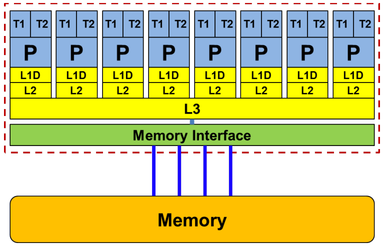
Lore 2

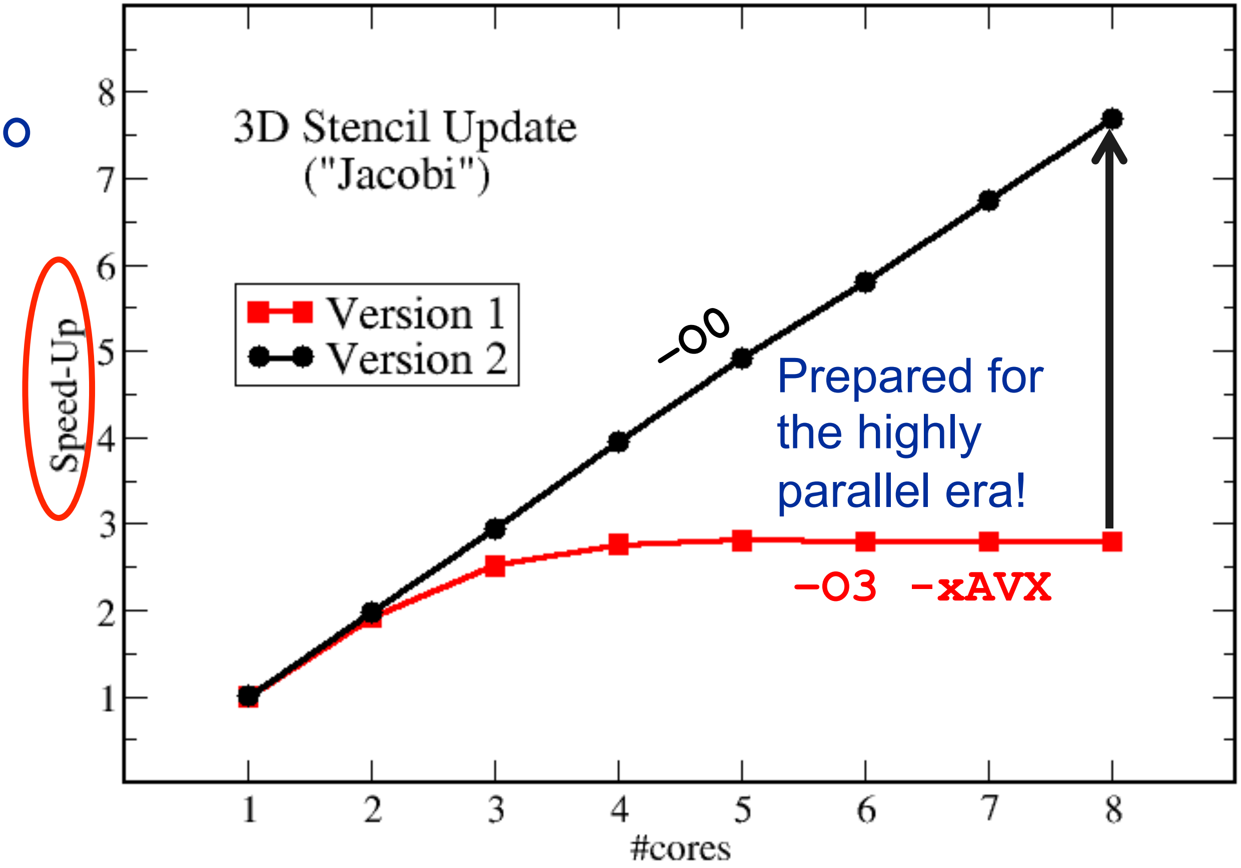
Single core performance no longer matters since we have so many of them and use scalable codes

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



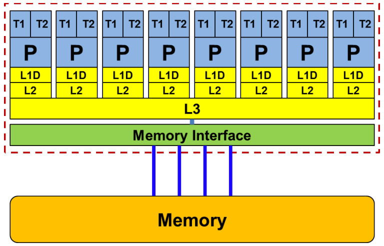


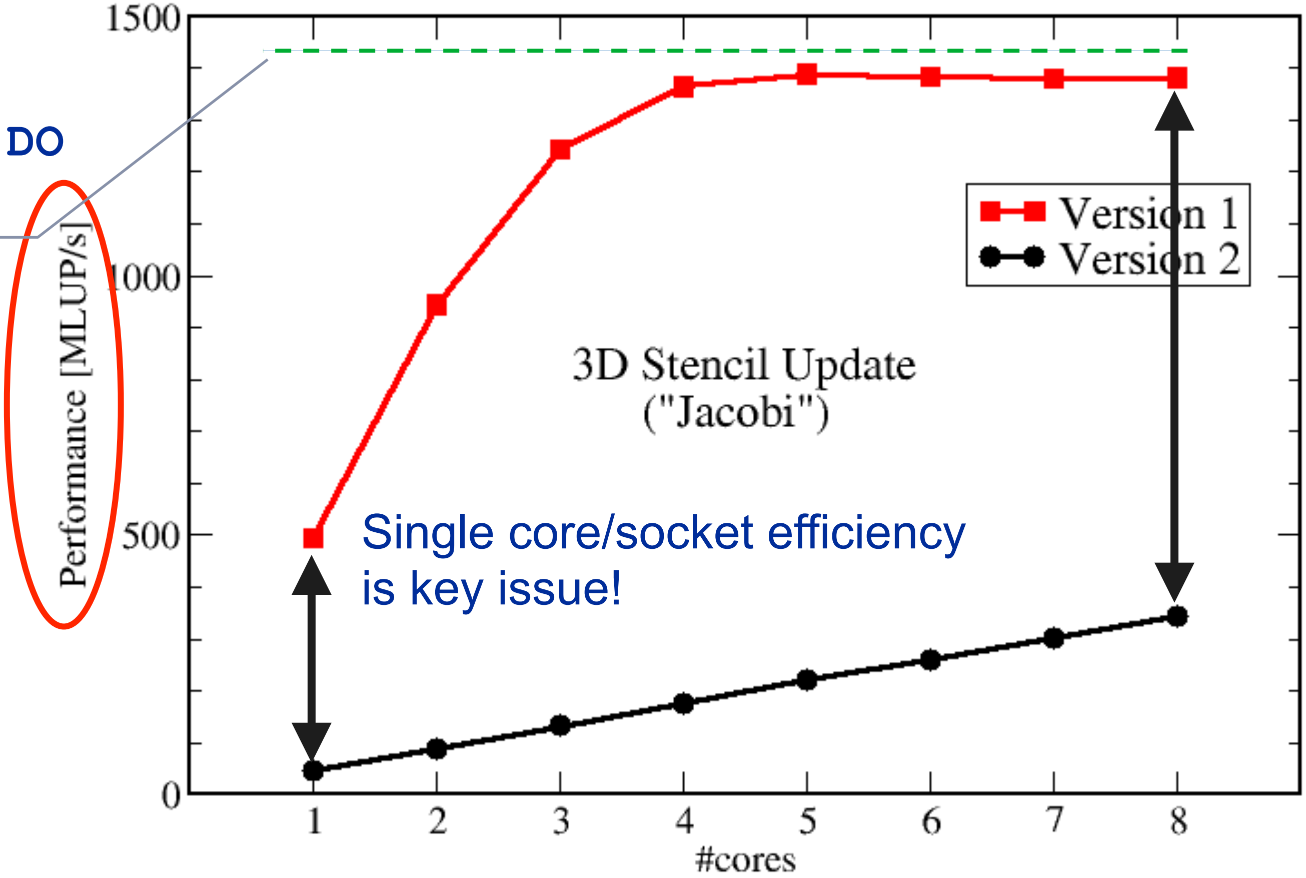
Prepared for the highly parallel era!

Scalability Myth: Code scalability is the key issue

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

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





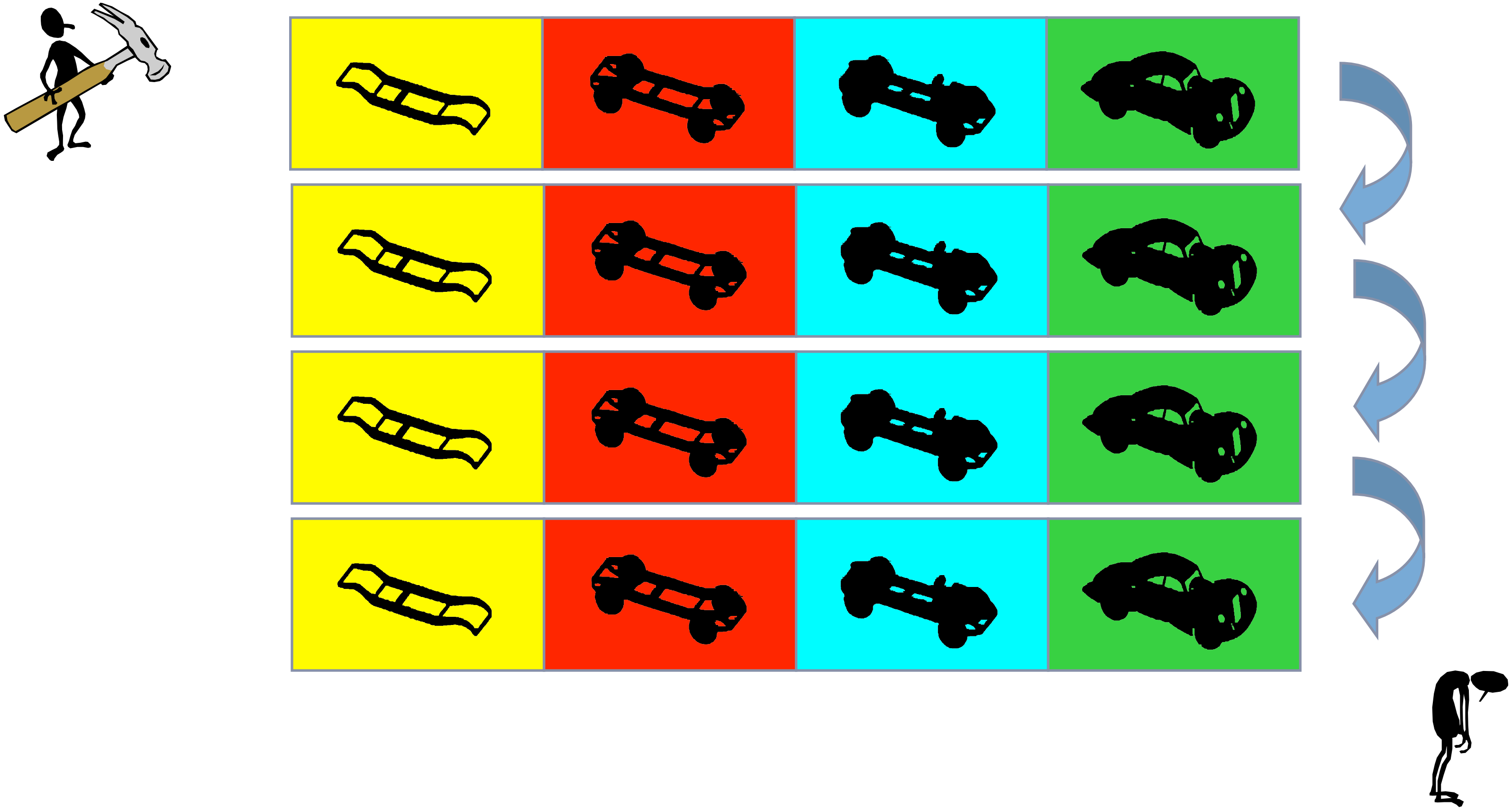
Single core/socket efficiency is key issue!

UNDERSTANDING PARALLELISM AND THE LIMITATIONS OF PARALLEL COMPUTING

Amdahls law

Understanding Parallelism:

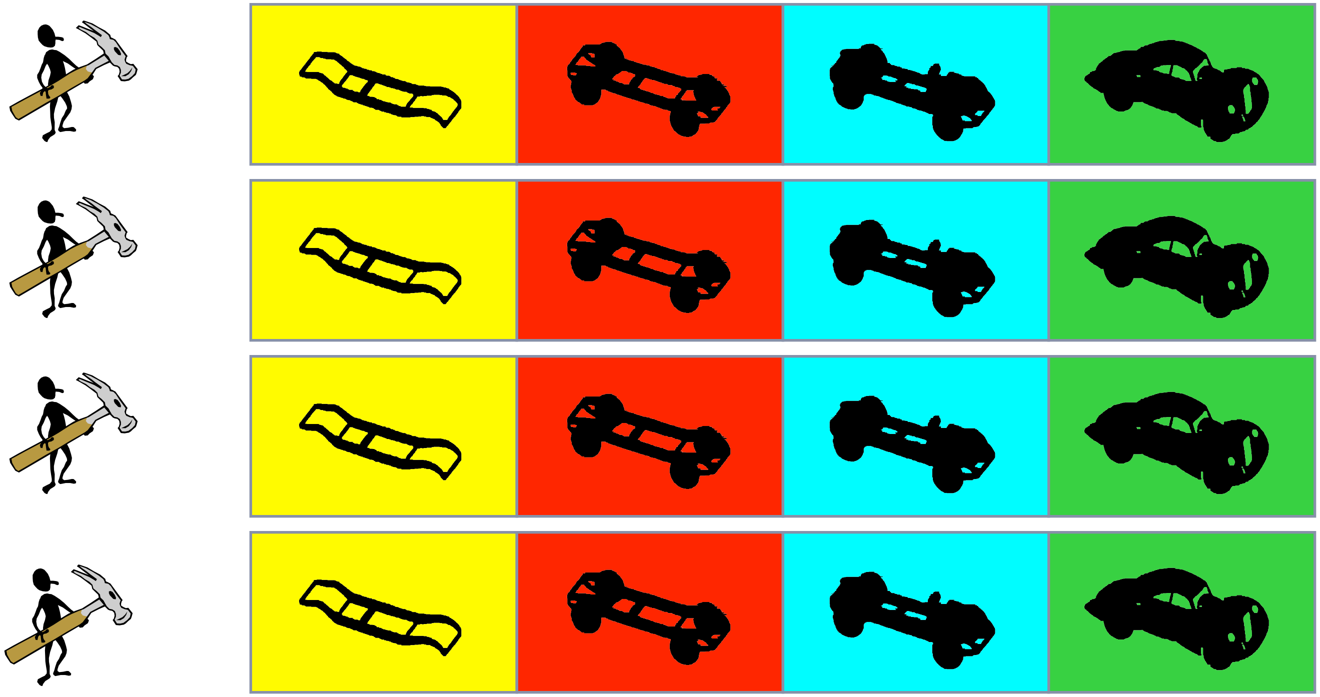
Sequential work



After 16 time steps: 4 cars

Understanding Parallelism:

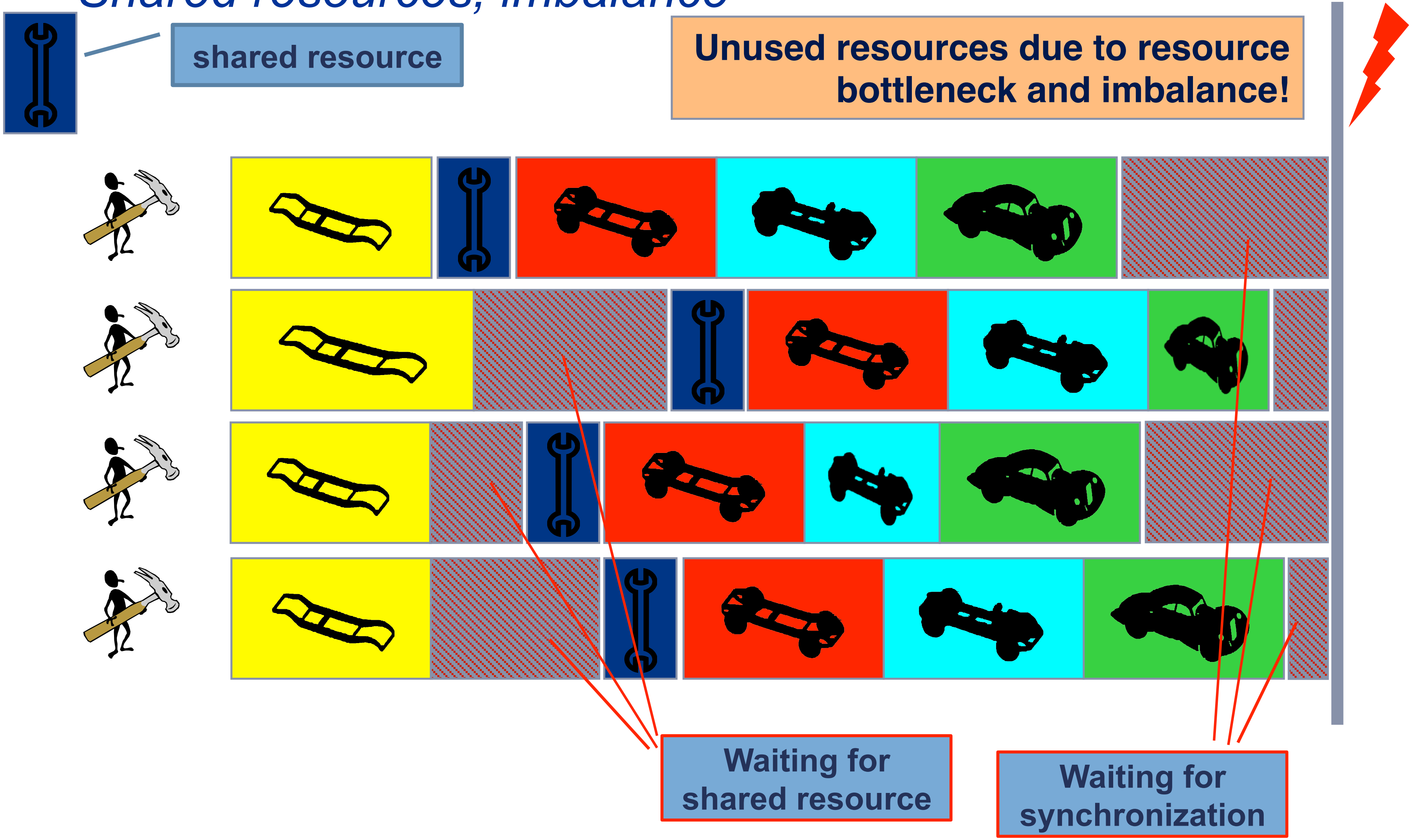
Parallel work



After 4 time steps: 4 cars

"*perfect speedup*"

Understanding parallelism:
Shared resources, imbalance



shared resource

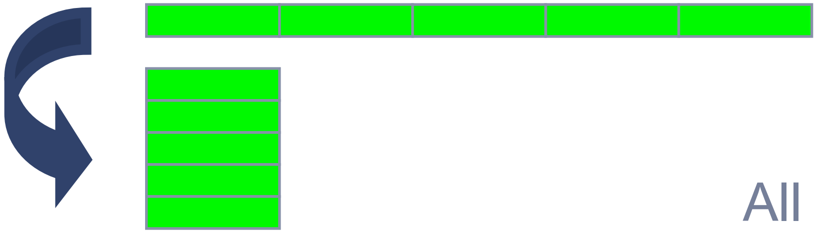
Unused resources due to resource bottleneck and imbalance!

Waiting for shared resource

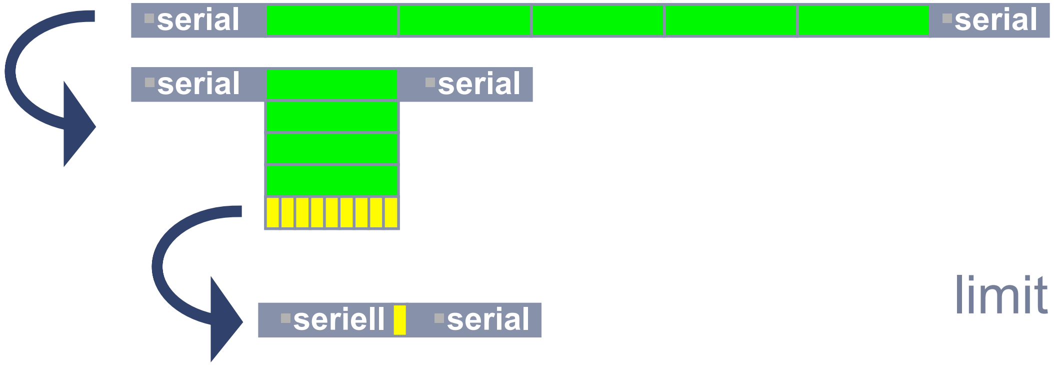
Waiting for synchronization

Limitations of Parallel Computing:

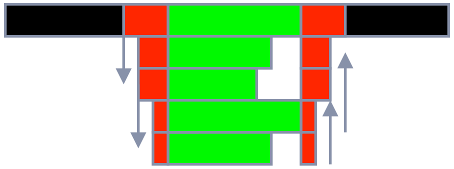
Amdahl's Law



Ideal world:
All work is perfectly parallelizable



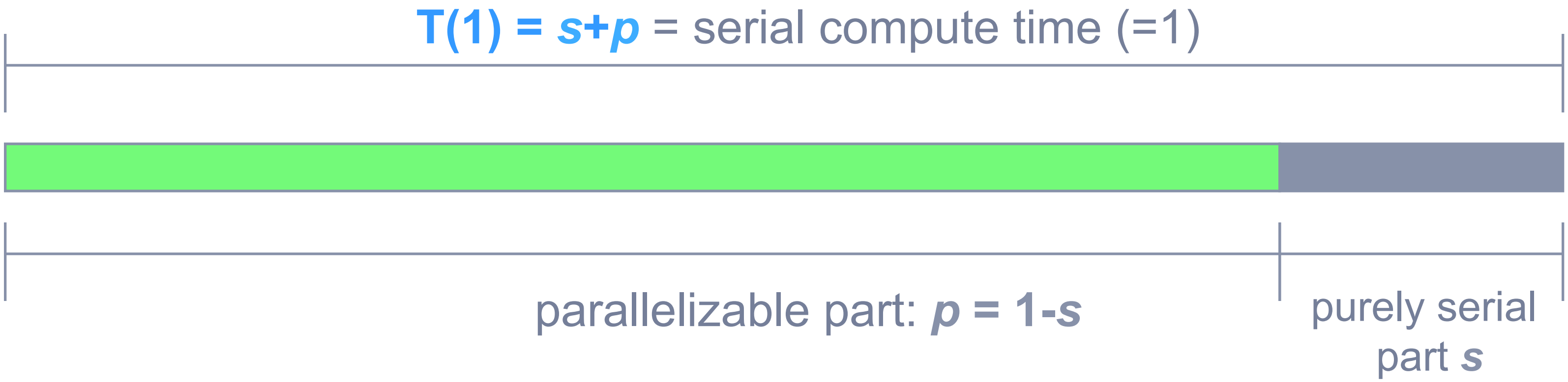
Closer to reality:
Purely serial parts
limit maximum speedup



Reality is even worse:
Communication and synchronization
impede scalability even further

Limitations of Parallel Computing:

Calculating Speedup in a Simple Model ("strong scaling")

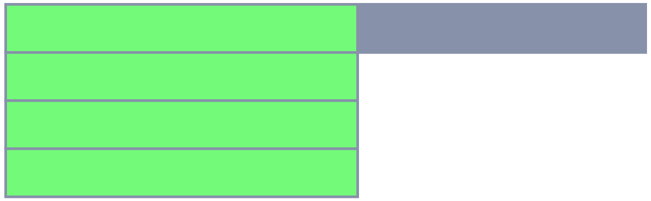


T(1) = *s*+*p* = serial compute time (=1)

parallelizable part: *p* = 1-*s*

purely serial part *s*

Parallel execution time: T(*N*) = *s*+*p*/*N*



General formula for speedup:
Amdahl's Law (1967)
"strong scaling"

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

Limitations of Parallel Computing:

Amdahl's Law ("strong scaling")

- Reality: No task is perfectly parallelizable
  - **Shared resources have to be used serially**
  - **Task interdependencies must be accounted for**
  - **Communication overhead (but that can be modeled separately)**

- Benefit of parallelization may be strongly limited
  - **"Side effect": limited scalability leads to inefficient use of resources**
  - **Metric: Parallel Efficiency**
    (*what percentage of the workers/processors is efficiently used*):

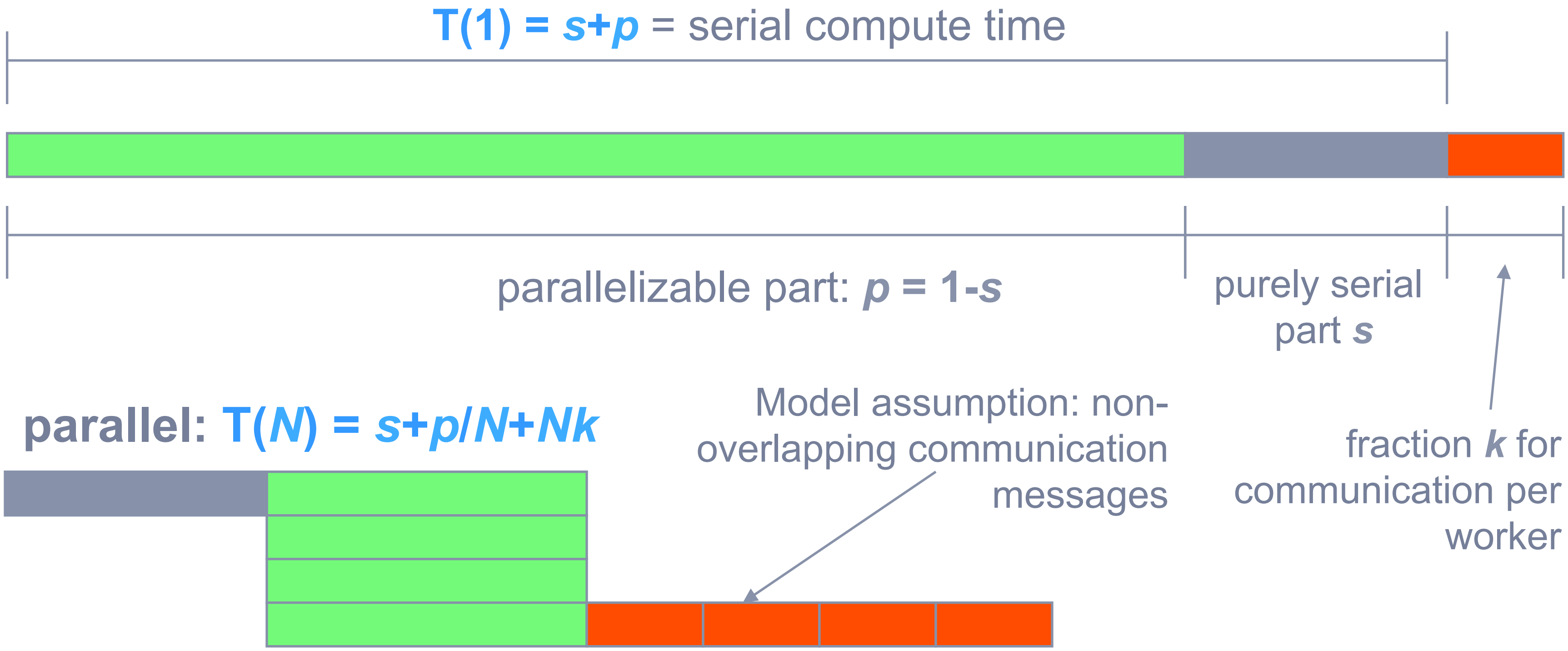
$$\varepsilon_p(N) = \frac{S_p(N)}{N}$$

- Amdahl case:

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

Limitations of Parallel Computing:

Adding a simple communication model for strong scaling



T(1) = *s*+*p* = serial compute time

parallelizable part: *p* = 1-*s*

purely serial part *s*

fraction *k* for communication per worker

parallel: T(*N*) = *s*+*p*/*N*+*Nk*

Model assumption: non-overlapping communication messages

General formula for speedup:

$$\boxed{S_p^k = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N} + Nk}}$$

Limitations of Parallel Computing:

Amdahl's Law ("strong scaling")

- Large N limits
  - **at k=0, Amdahl's Law predicts**

$$\boxed{\lim_{N\to\infty} S_p^0(N) = \frac{1}{s}}$$

- independent of *N* !

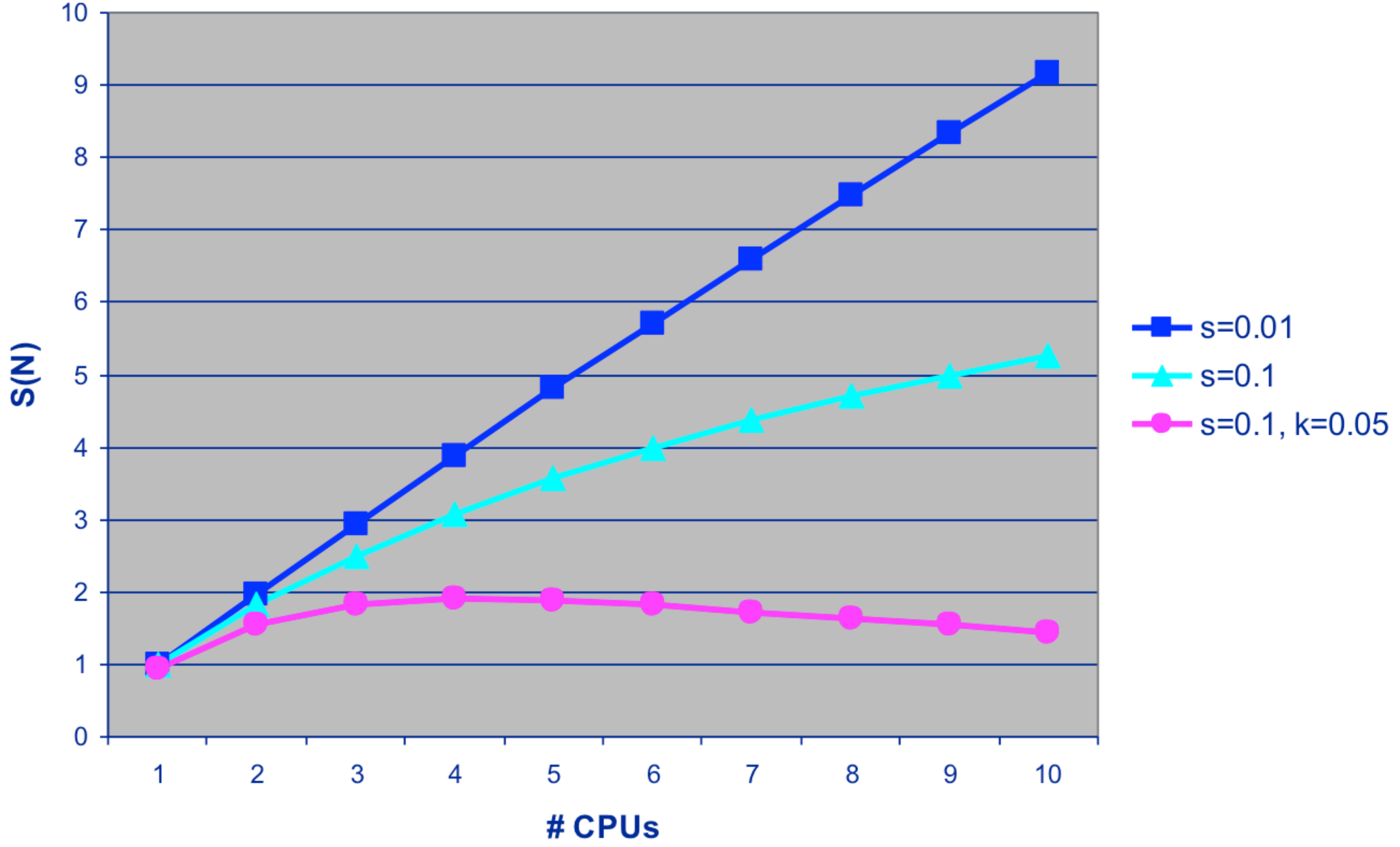
  - **at k≠0, our simple model of communication overhead yields a beaviour of**

$$\boxed{S_p^k(N) \xrightarrow{N>>1} \frac{1}{Nk}}$$

- Problems in real world programming
  - **Load imbalance**
  - Shared resources have to be used serially (e.g. IO)
  - Task interdependencies must be accounted for
  - Communication overhead

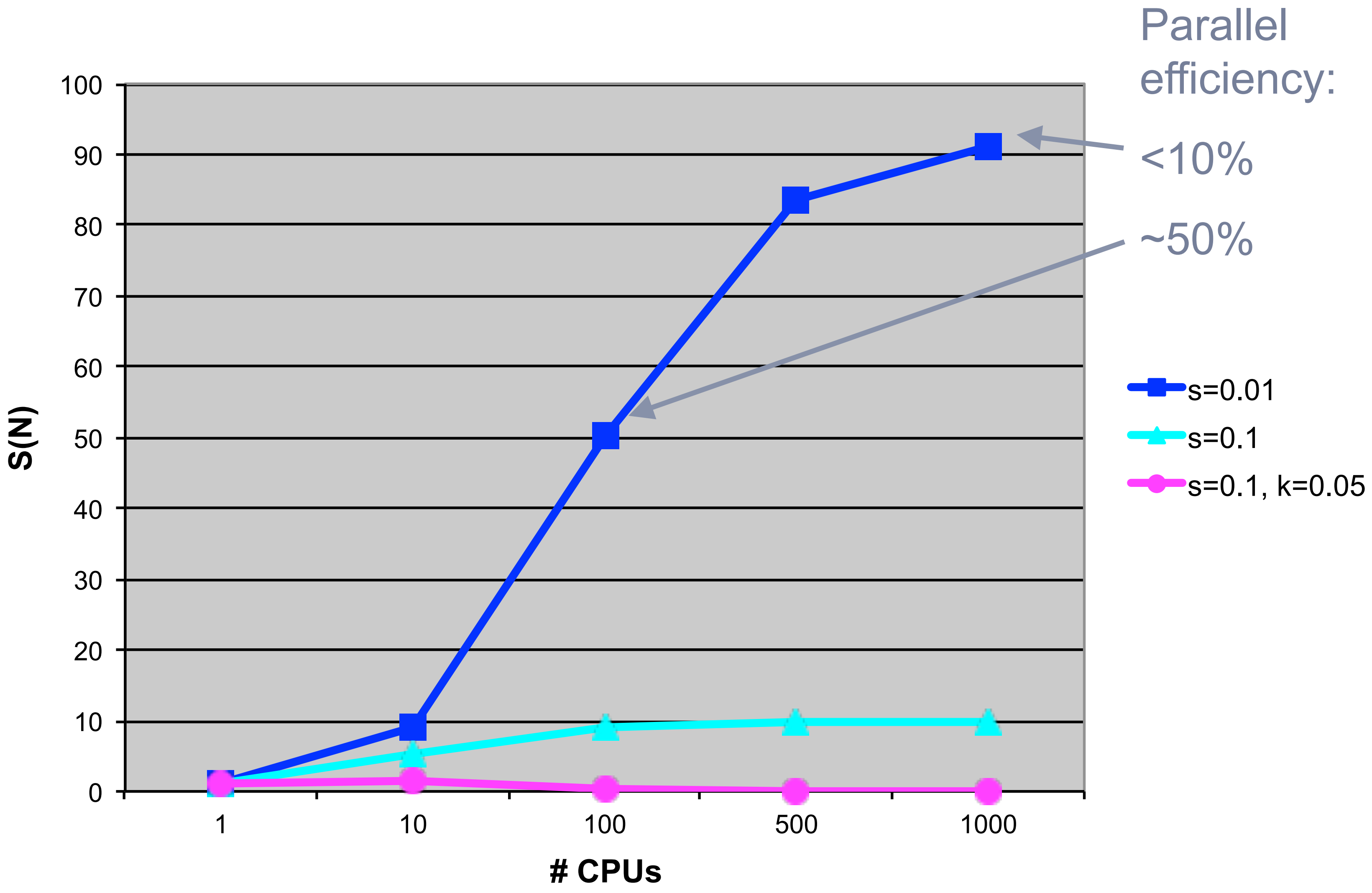
Limitations of Parallel Computing:

Amdahl´s Law ("strong scaling") + comm. model



Limitations of Parallel Computing:

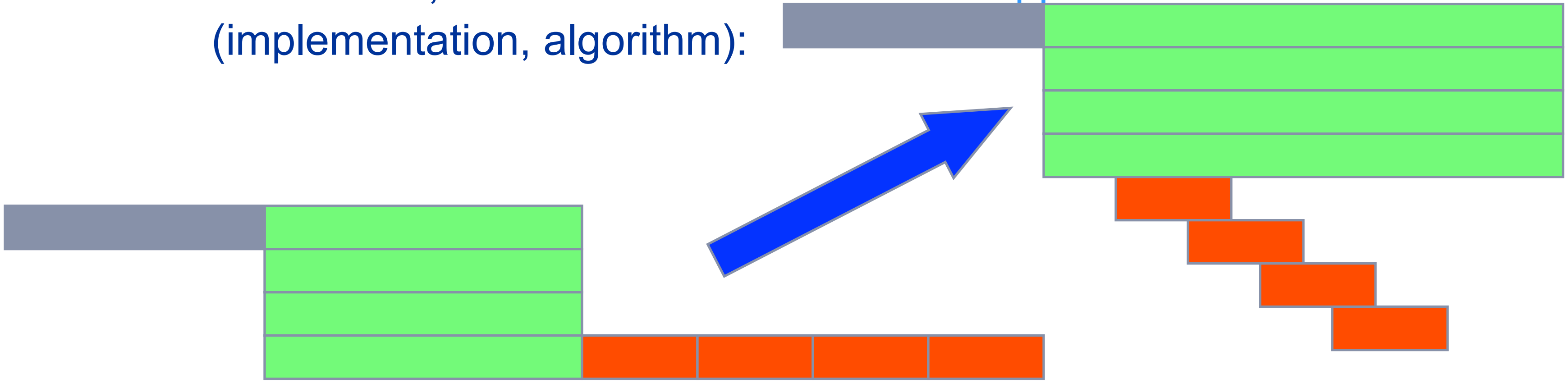
Amdahl´s Law (“strong scaling”)



Limitations of Parallel Computing:

How to mitigate overheads

- Communication is not necessarily purely serial
  - Non-blocking crossbar networks can transfer many messages concurrently – factor *Nk* in denominator becomes *k* (technical measure)
  - Sometimes, communication can be overlapped with useful work (implementation, algorithm):



  - Communication overhead may show a more fortunate behavior than *Nk*
  - "superlinear speedups“: data size per CPU decreases with increasing CPU count → may fit into cache at large CPU counts

Limits of Scalability: Serial & Parallel fraction

Serial fraction *s* may depend on

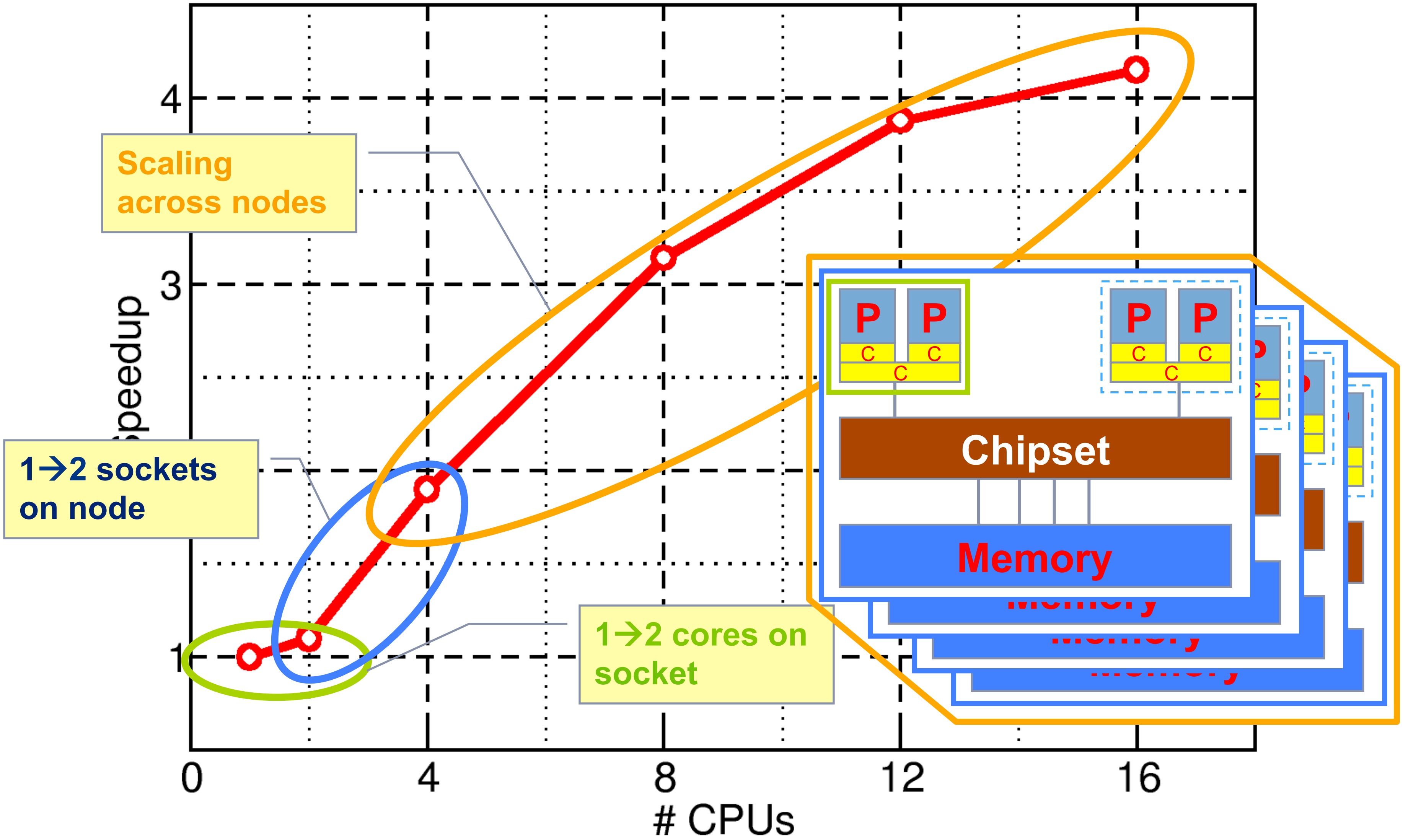
- Program / algorithm
  - Non-parallelizable part, e.g. recursive data setup
  - Non-parallelizable IO, e.g. reading input data
  - Communication structure
  - Load balancing (assumed so far: perfect balanced)
  - …
- Computer hardware
  - Processor: Cache effects & memory bandwidth effects
  - Parallel Library; Network capabilities; Parallel IO
  - …

Determine *s* "experimentally":

Measure speedup and fit data to Amdahl’s law – but that could be more complicated than it seems…

Scalability data on modern multi-core systems

An example



Scaling across nodes

1→2 sockets on node

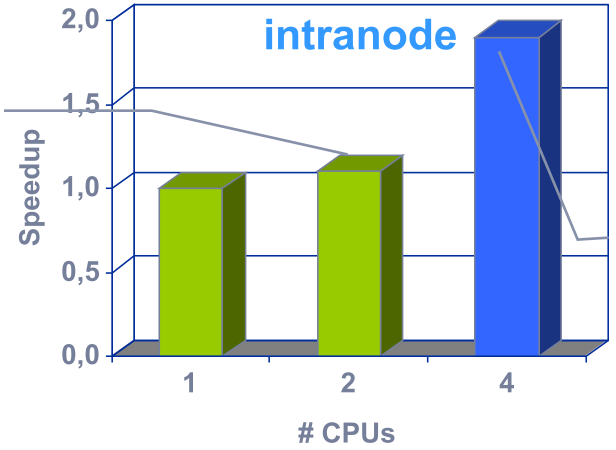
1→2 cores on socket

Scalability data on modern multi-core systems

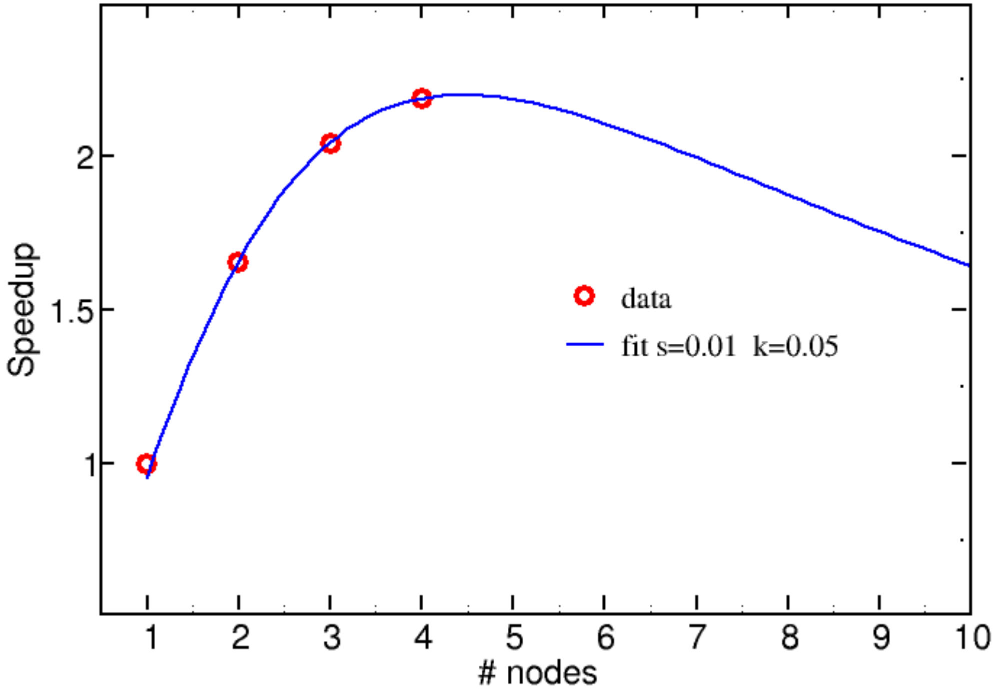
The scaling baseline

- Scalability presentations should be grouped according to the largest unit that the scaling is based on (the “scaling baseline”)

memory-bound code!



Good scaling across sockets



Amdahl model with communication: Fit

$$S(N) = \frac{1}{s + \frac{1-s}{N} + kN}$$

to inter-node scalability numbers (N = # nodes, >1)

Application to "accelerated computing"

- SIMD, GPUs, Cell SPEs, FPGAs, just *any optimization*…
- Assume overall (serial, un-accelerated) runtime to be $\boldsymbol{T_s = s + p = 1}$
- Assume ***p*** **can be accelerated** and run ***α*** **times faster**. We neglect any additional cost (communication…)
- To get a **speedup of** ***rα***, how small must *s* be? Solve for *s*:

$$r\alpha = \frac{1}{s + \frac{1-s}{\alpha}} \quad \Rightarrow \quad s = \frac{r^{-1} - 1}{\alpha - 1}$$

- At ***α*****=100** and ***r*** **=0.9** (for an overall speedup of 90), we get **s≈0.0011**, i.e. you must accelerate over 99.9% of serial runtime!
- Limited memory on accelerators may limit the achievable speedup

End part 1

TOPOLOGY OF MULTI-CORE / MULTI-SOCKET SYSTEMS

- Chip Topology
- Node Topology
- Memory Organisation

Building blocks for multi-core compute nodes

- **Core**: Unit reading and executing instruction stream
- **Chip**: One integrated circuit die
- **Socket**/Package: May consist of multiple chips

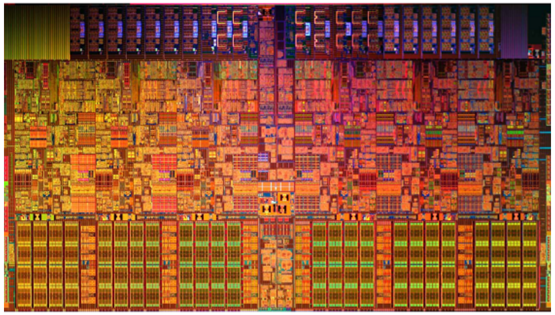
- Memory Hierarchy:
  - Private caches
  - Shared caches
  - **ccNUMA**: Replicated memory interfaces

Chip Topologies

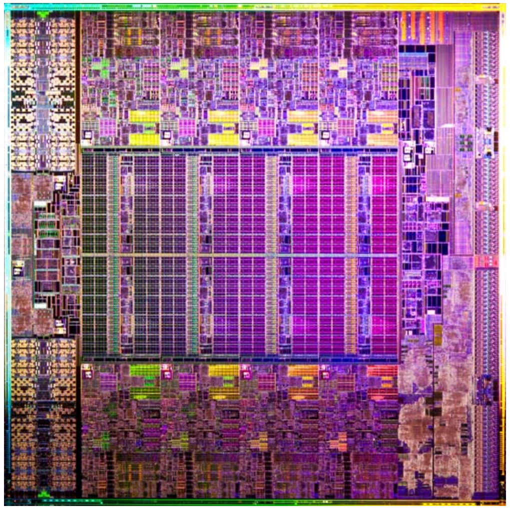
- Separation into core and uncore
- Memory hierarchy holding together the chip design
- L1 (L2) private caches
- L3 cache shared (LLC)

- Serialized LLC → not scalable

- Segmented ring bus, distributed LLC → scalable design

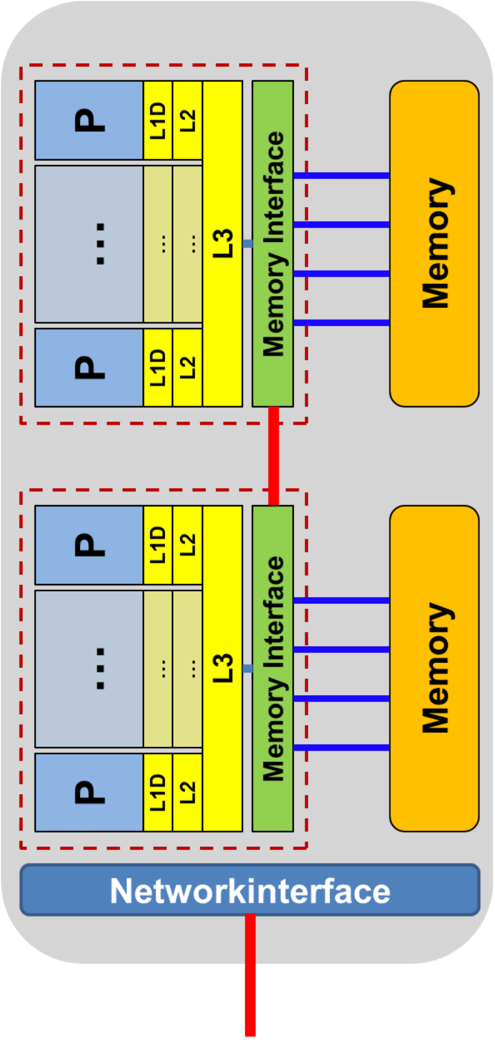


Westmere-EP, 6C, 32nm 248mm^2



SandyBridge-EP, 8C, 32nm 435mm^2

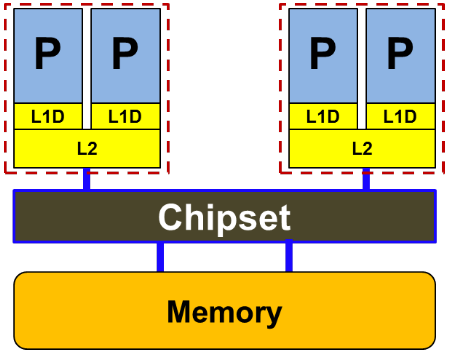
Cray XC30 “SandyBridge-EP” 8-core dual socket node



- **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

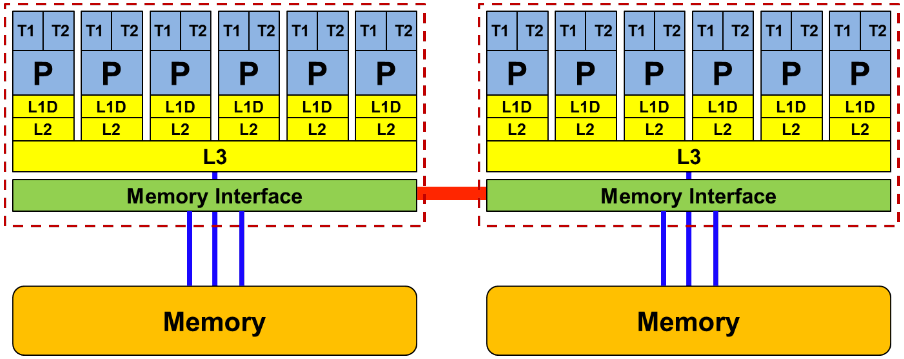
From UMA to ccNUMA
Memory architectures

Yesterday (2006): Dual-socket Intel “Core2” node:



- Uniform Memory Architecture (UMA)
- Flat memory ; symmetric MPs

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?*

Conclusions about Node Topologies

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)

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

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

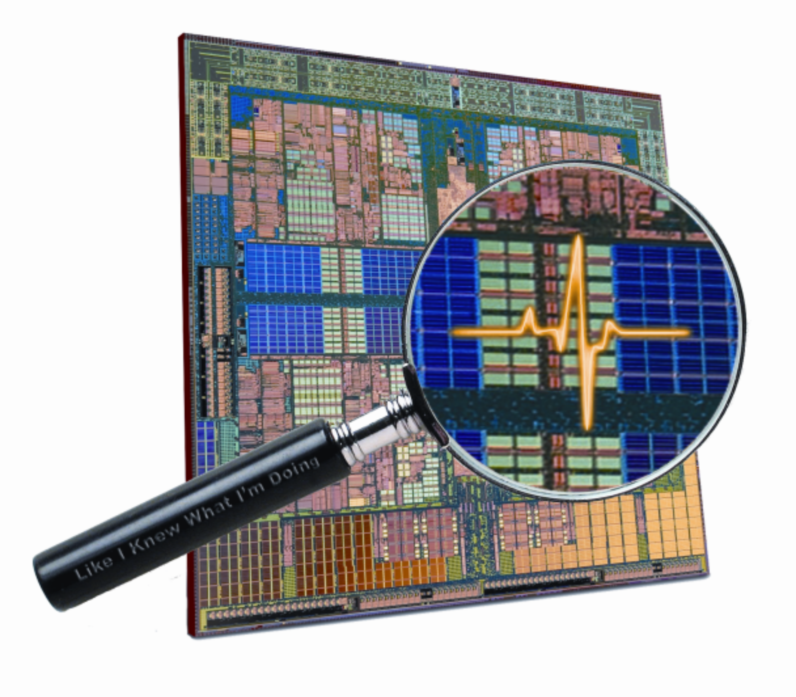
```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 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*. **Accepted for PSTI2010, Sep 13-16, 2010, San Diego, CA**

- **http://arxiv.org/abs/1004.4431**

Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux 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

Output of `likwid-topology -g`

on one node of Cray XE6

```
-------------------------------------------------------------
CPU type:       AMD Interlagos processor
*************************************************************
Hardware Thread Topology
*************************************************************
Sockets:                2
Cores per socket:       16
Threads per core:       1
-------------------------------------------------------------
HWThread        Thread          Core            Socket
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
[...]
16              0               0               1
17              0               1               1
18              0               2               1
19              0               3               1
[...]
-------------------------------------------------------------
Socket 0: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 )
Socket 1: ( 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 )
-------------------------------------------------------------

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

Output of likwid-topology continued

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

*************************************************************
NUMA Topology
*************************************************************
NUMA domains: 4
-------------------------------------------------------------
Domain 0:
Processors:  0 1 2 3 4 5 6 7
Memory: 7837.25 MB free of total 8191.62 MB
-------------------------------------------------------------
Domain 1:
Processors:  8 9 10 11 12 13 14 15
Memory: 7860.02 MB free of total 8192 MB
-------------------------------------------------------------
Domain 2:
Processors:  16 17 18 19 20 21 22 23
Memory: 7847.39 MB free of total 8192 MB
-------------------------------------------------------------
Domain 3:
Processors:  24 25 26 27 28 29 30 31
Memory: 7785.02 MB free of total 8192 MB
-------------------------------------------------------------
```

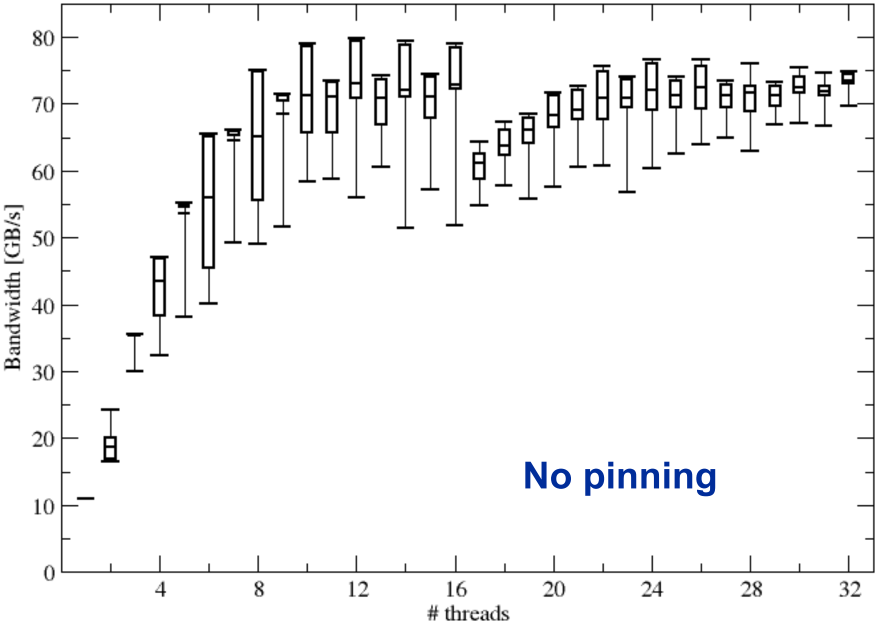
Output of likwid-topology continued

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

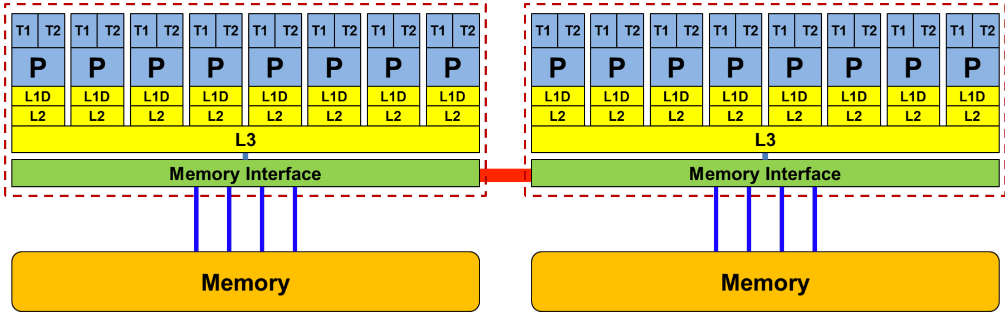
ENFORCING THREAD/PROCESS-CORE AFFINITY UNDER THE LINUX OS

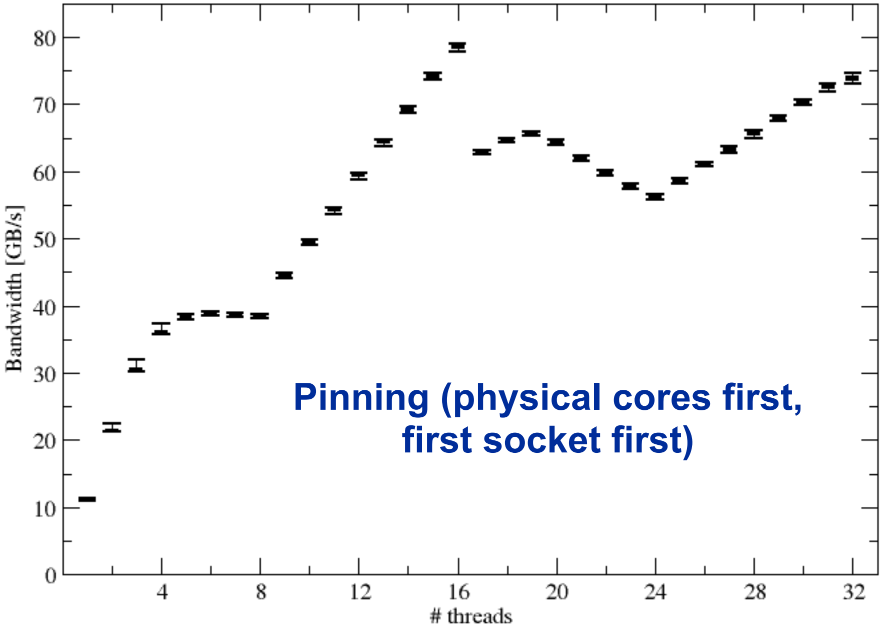
- Standard tools and OS affinity facilities under program control
- likwid-pin

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



No pinning





Pinning (physical cores first, first socket first)

There are several reasons for caring about affinity:

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

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

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

    Linux: `sched_setaffinity()`, PLPA → 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

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
    ```

Likwid-pin

Example: Intel OpenMP

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

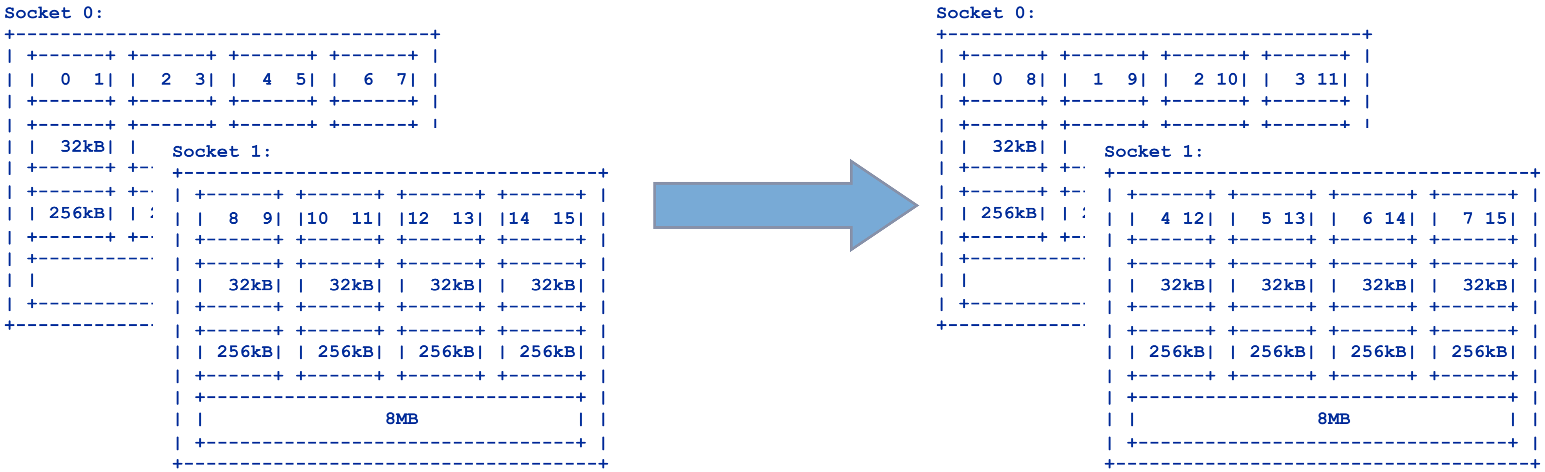
Likwid-pin

Using logical core numbering

Core numbering may vary from system to system

- 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**)



```
Socket 0:
+---------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| |  0  1| |  2  3| |  4  5| |  6  7| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  32kB| |
| +-------+ +--
| +-------+ +--
| | 256kB| | 2
| +-------+ +--
| +-----------
| |
| +-----------
+-------------

Socket 1:
+---------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| |  8  9| |10  11| |12  13| |14  15| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-----------------------------------+ |
| |                8MB                | |
| +-----------------------------------+ |
+---------------------------------------+
```

```
Socket 0:
+---------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| |  0  8| |  1  9| |  2 10| |  3 11| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  32kB| |
| +-------+ +--
| +-------+ +--
| | 256kB| | 2
| +-------+ +--
| +-----------
| |
| +-----------
+-------------

Socket 1:
+---------------------------------------+
| +-------+ +-------+ +-------+ +-------+ |
| |  4 12| |  5 13| |  6 14| |  7 15| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| |  32kB| |  32kB| |  32kB| |  32kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-------+ +-------+ +-------+ +-------+ |
| | 256kB| | 256kB| | 256kB| | 256kB| |
| +-------+ +-------+ +-------+ +-------+ |
| +-----------------------------------+ |
| |                8MB                | |
| +-----------------------------------+ |
+---------------------------------------+
```

Across all cores in the node:

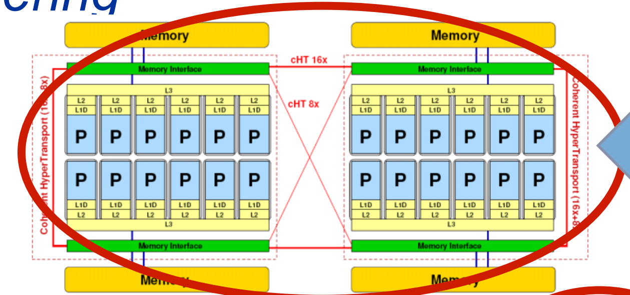
```
OMP_NUM_THREADS=8 likwid-pin -c N:0-7 ./a.out
```

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

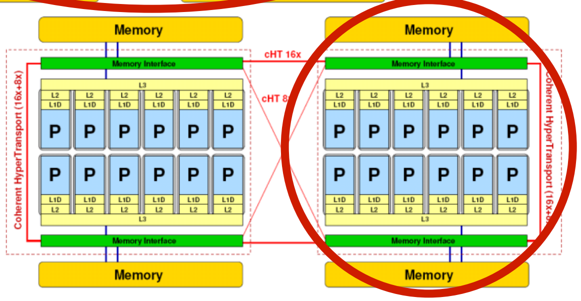
Using logical core numbering



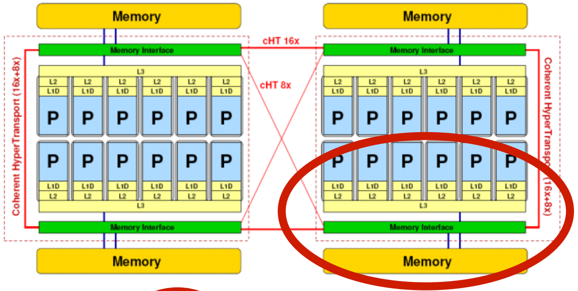
Default if -c is not specified!

Possible unit prefixes

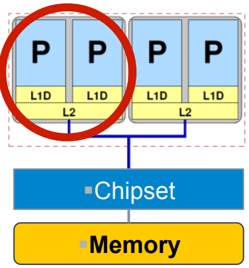
N node



S socket



M NUMA domain



C outer level cache group

DEMO

PATTERN-DRIVEN PERFORMANCE ENGINEERING PROCESS

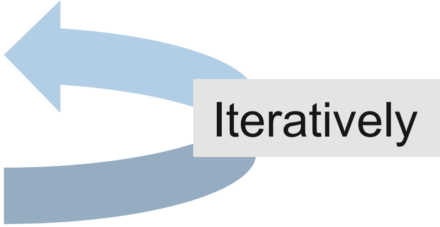
Basics of Benchmarking

Performance Patterns

Signatures

Basics of Optimization

1. Define relevant test cases
2. Establish a sensible performance metric
3. Acquire a runtime profile (sequential)
4. Identify hot kernels (Hopefully there are any!)
5. Carry out optimization process for each kernel


Iteratively

Motivation:

- Understand observed performance
- Learn about code characteristics and machine capabilities
- Deliberately decide on optimizations

Best Practices Benchmarking

Preparation

- Reliable timing (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Get exclusive System
- System state (Clock, Turbo mode, Memory, Caches)
- Consider to automate runs with a skript (Shell, python, perl)

Doing

- Affinity control
- Check: Is the result reasonable?
- Is result deterministic and reproducible.
- Statistics: Mean, Best ??
- Basic variations: Thread count, affinity, working set size (Baseline!)

Best Practices Benchmarking cont.

Postprocessing

- Documentation
- Try to understand and explain the result
- Plan variations to gain more information
- Many things can be better understood if you plot them (gnuplot, xmgrace)

Philosophy of pattern based approach

Motivated by a **resource utilization driven** view.

Provide a structured **iterative process** based on:

- Performance patterns
- A diagnostic performance model

Performance patterns are typical performance limiting bottlenecks

Patterns are indicated by **signatures** which can consist of:

- HPM data
- Scaling behavior
- Other data

Uses one of the most powerful tools available:

Your brain !

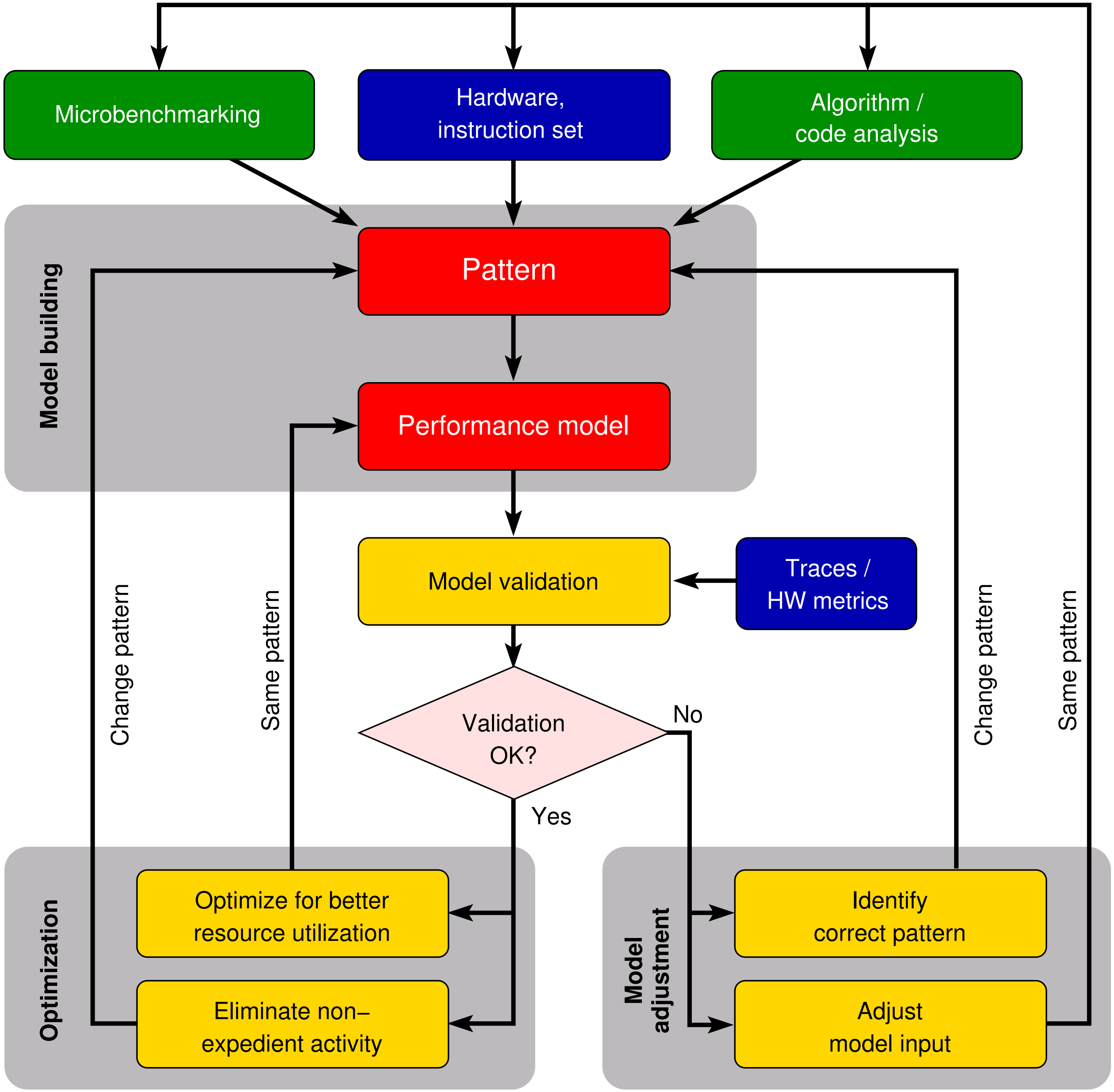
You are a investigator making sense of what’s going on.

Performance pattern classification

1. Maximum resource utilization
2. Hazards
3. Work related (Application or Processor)

The system offers two basic resources:

- **Execution of instructions** (primary)
- **Transferring data** (secondary)



Notions of work

- Application work
- Processor work

Pattern: qualitative

⇕

Model: quantitative

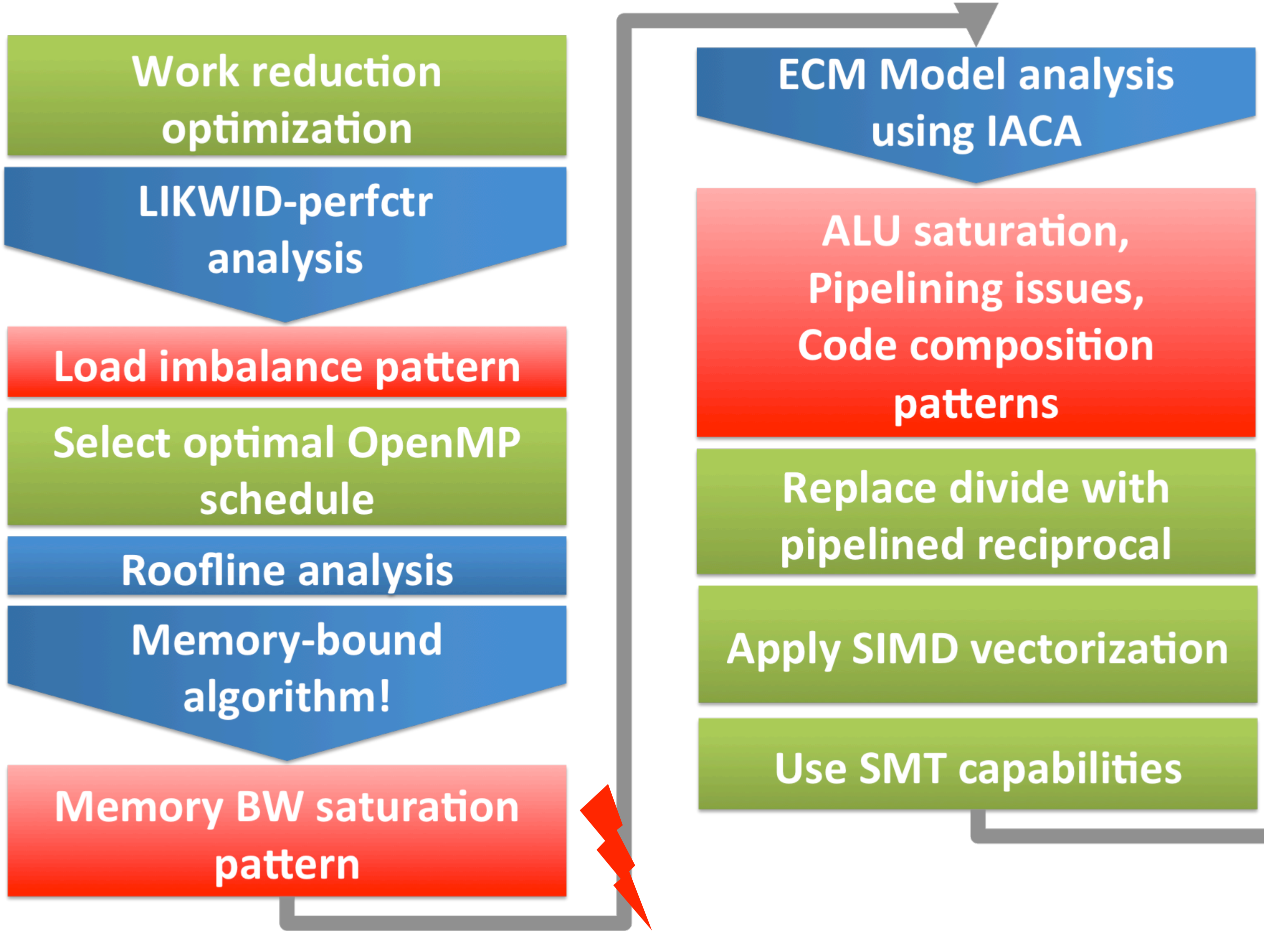
Find the relevant limiting bottleneck!

Pattern		Behavior
Bandwidth saturation		saturating speedup across cores sharing a data path
Limited Instruction throughput	Pipeline saturation	throughput at design limit
	Pipelining hazards	in-core throughput far from design limit, performance insensitive to data size
	Control flow issues	
Inefficient data access	Strided Access	simple BW models far too optimistic
	Erratic Access	
Microarchitectural anomalies		large discrepancy from simple performance models
False cacheline sharing		very low speedup, or slowdown / discrepancy from model only in parallel case
Bad ccNUMA page placement		bad/no scaling across locality domains, better performance w/ interleaved placement
Load imbalance		saturating/sub-linear speedup
Synchronization overhead		speedup going down as more cores are added / no speedup with small problem sizes
Code composition issues	Instruction overhead	low application performance, good scaling across cores, performance insensitive to problem size
	Expensive instructions	
	Ineffective instructions	

Pattern		Detection
Bandwidth saturation		Bandwidth meets BW of suitable streaming benchmark
Limited Instruction throughput	Pipeline saturation	Low CPI, 1:1 ratio of cy to specific instruction counts
	Pipelining hazards	Large integral ratio of cy to specific instruction counts, high CPI
	Control flow issues	High branch rate, high branch miss ratio
Inefficient data access	Strided Access	Low cache hit ratio, frequent line evics/replacements
	Erratic Access	See above, plus low BW utilization (latency)
Microarchitectural anomalies		Very hardware specific, memory aliasing, alignment …
False cacheline sharing		Frequent remote evicts
Bad ccNUMA page placement		Unbalanced bandwidth on memory interfaces/ high remote traffic
Load imbalance		Different amount of “work” across cores
Synchronization overhead		Large non-”work” instruction count / Low CPI
Code composition issues	Instruction overhead	Low CPI / large non-FP instruction count, low resource utilization
	Expensive instructions	Large CPI
	Ineffective instructions	Scalar instructions dominating in data-parallel loops

optional

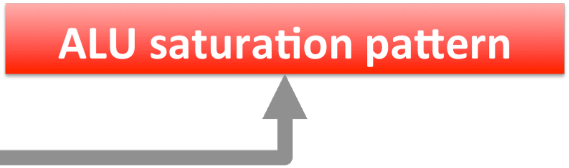
Example rabbitCT


Work reduction optimization
LIKWID-perfctr analysis
Load imbalance pattern
Select optimal OpenMP schedule
Roofline analysis
Memory-bound algorithm!
Memory BW saturation pattern
ECM Model analysis using IACA
ALU saturation, Pipelining issues, Code composition patterns
Replace divide with pipelined reciprocal
Apply SIMD vectorization
Use SMT capabilities

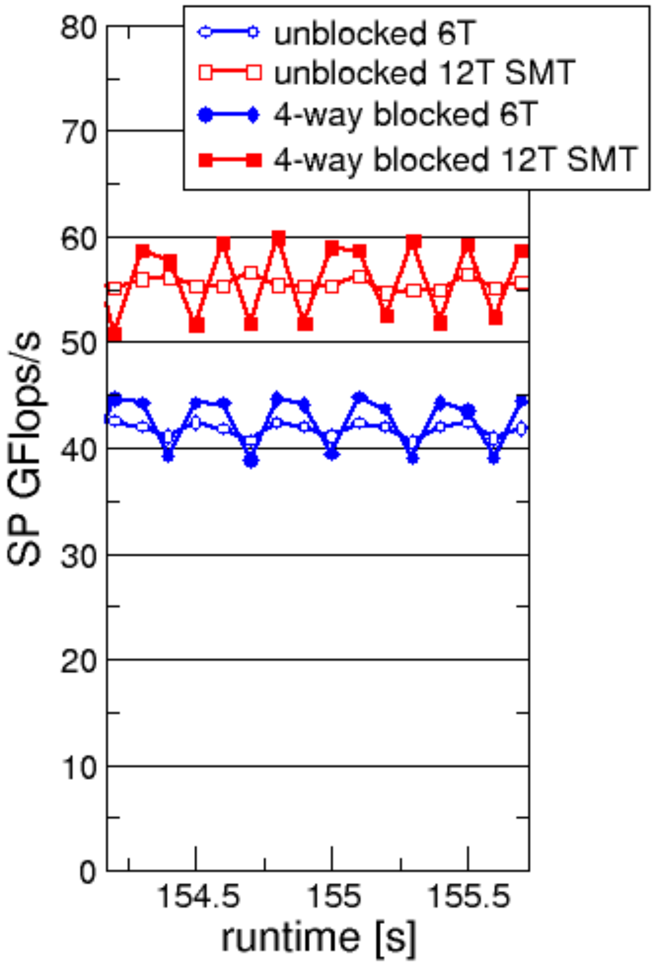
Result of effort:

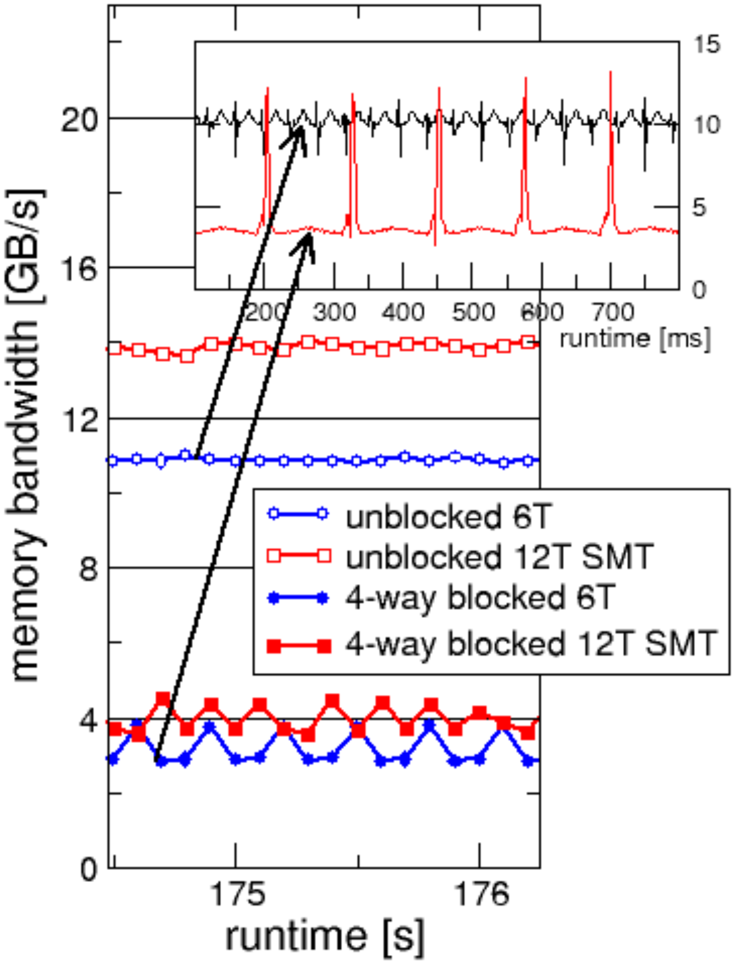
5-6 x improvement against initial parallel C code implementation

>50% of peak performance (SSE)



Ruling out memory bandwidth limitation





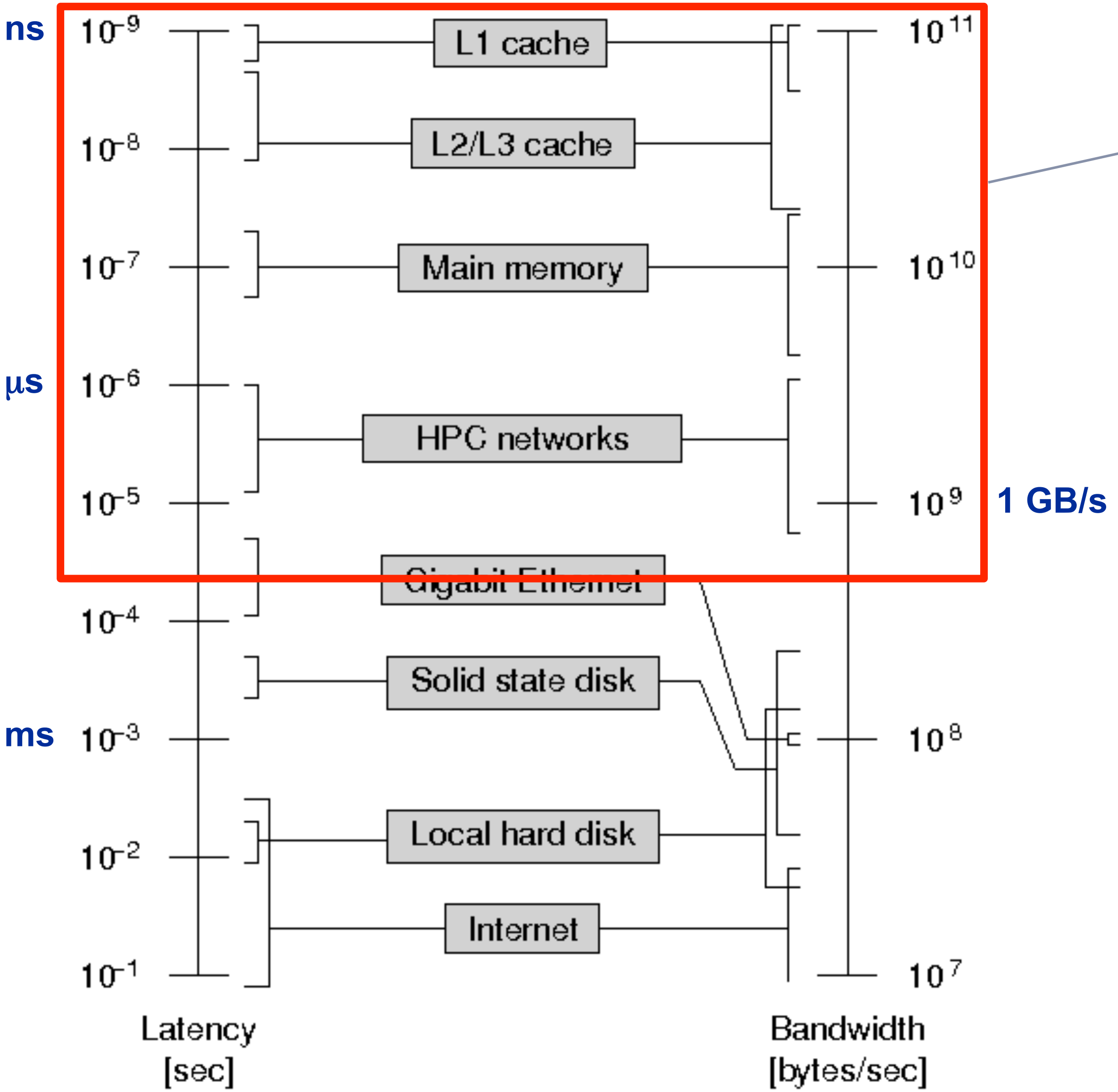
MICROBENCHMARKING FOR ARCHITECTURAL EXPLORATION

Probing of the memory hierarchy

Saturation effects in cache and memory

Typical OpenMP overheads

Latency and bandwidth in modern computer environments

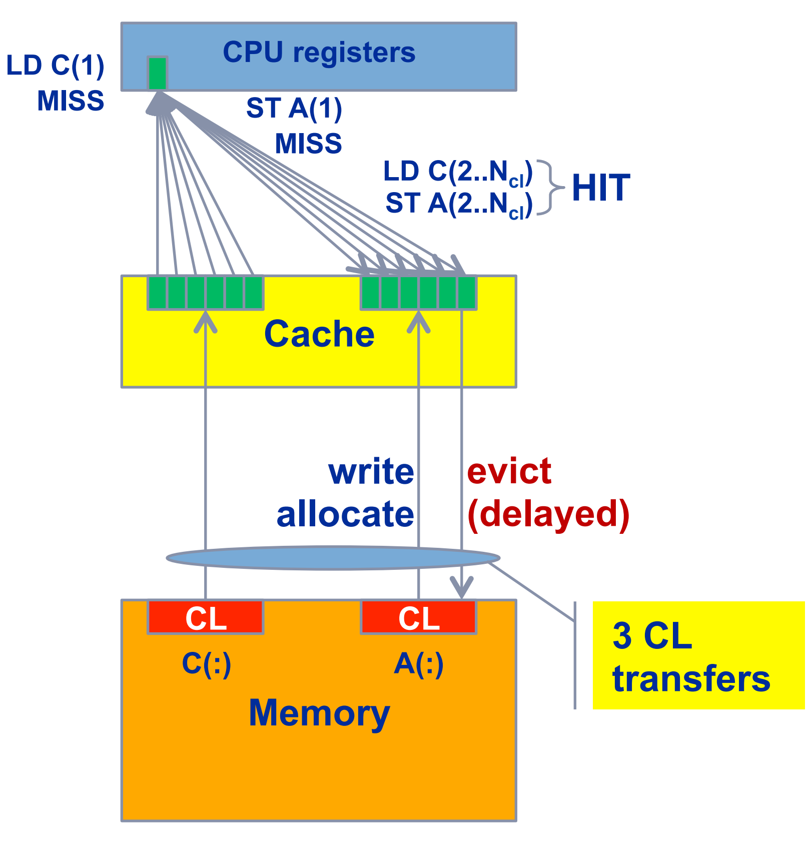


HPC plays here

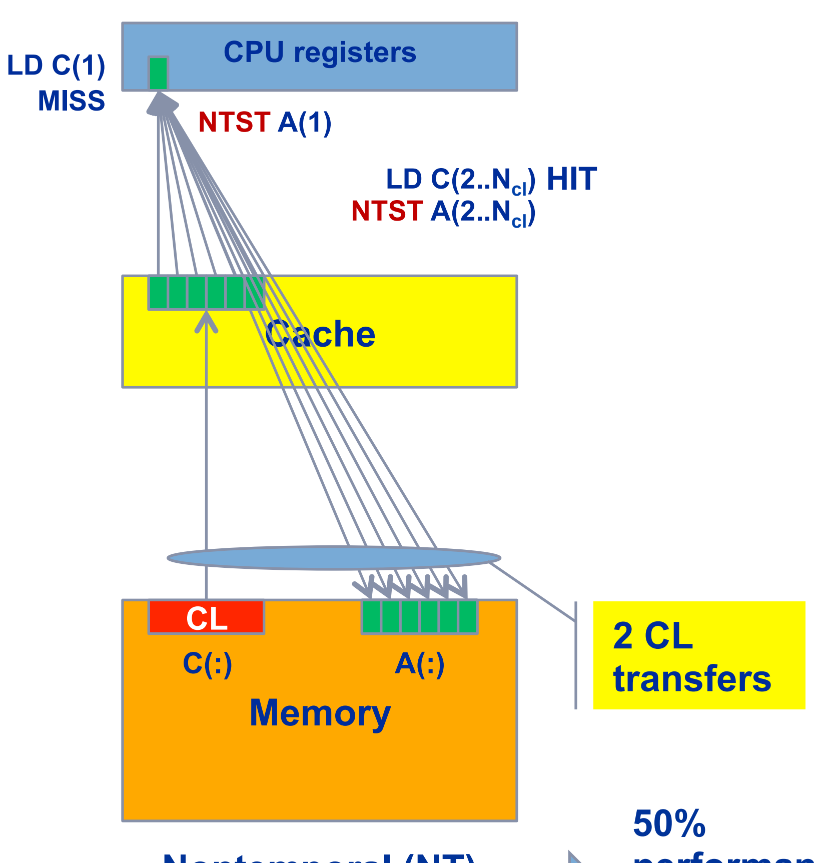
Avoiding slow data paths is the key to most performance optimizations!

Recap: Data transfers in a memory hierarchy

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



Standard stores



Nontemporal (NT) stores → 50% performance boost for COPY

The parallel vector triad benchmark
A “swiss army knife” for microbenchmarking

Simple streaming benchmark:

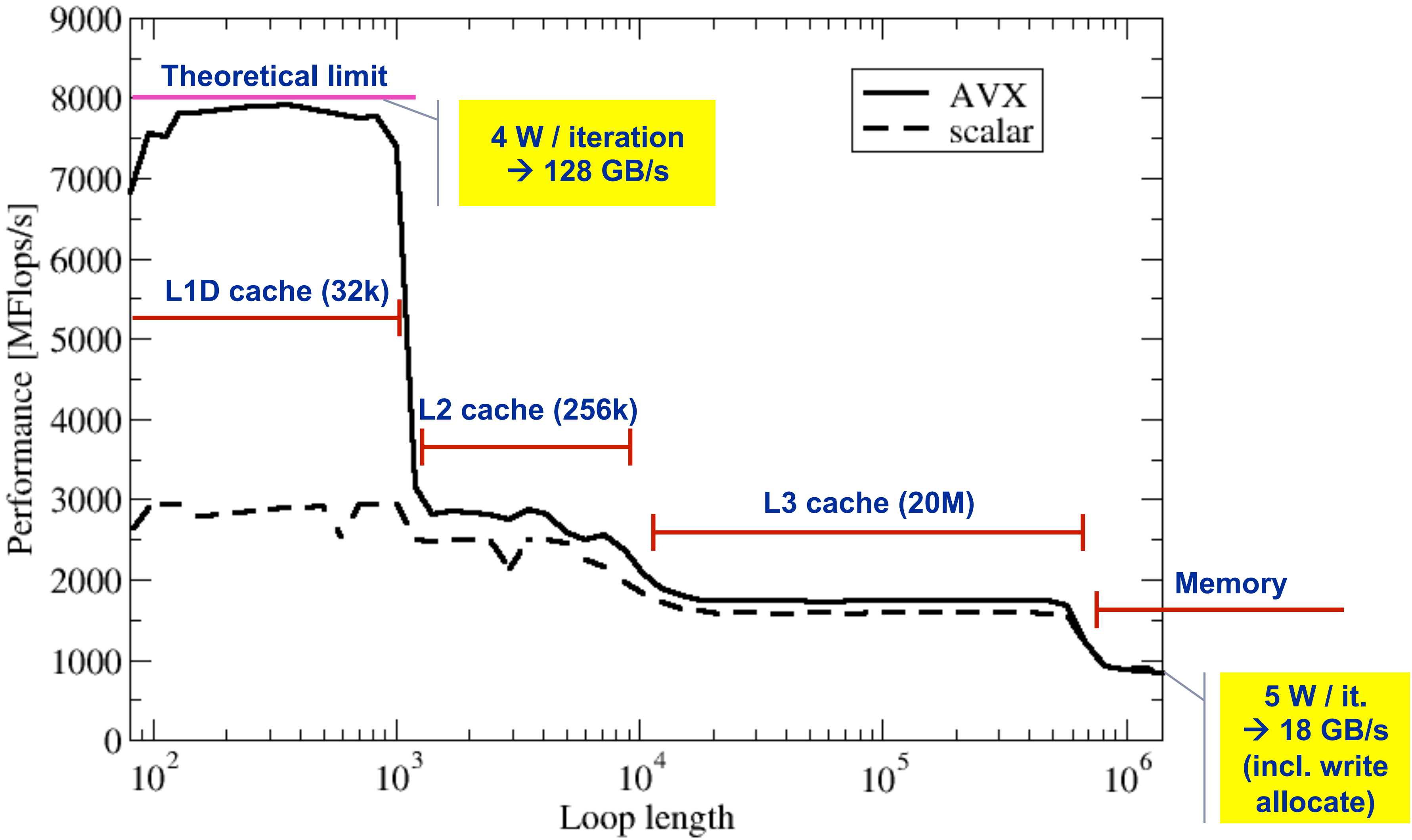
```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
```

Prevents smarty-pants compilers from doing “clever” stuff

Report performance for different N

This kernel is limited by data transfer performance for all memory levels on all current architectures!

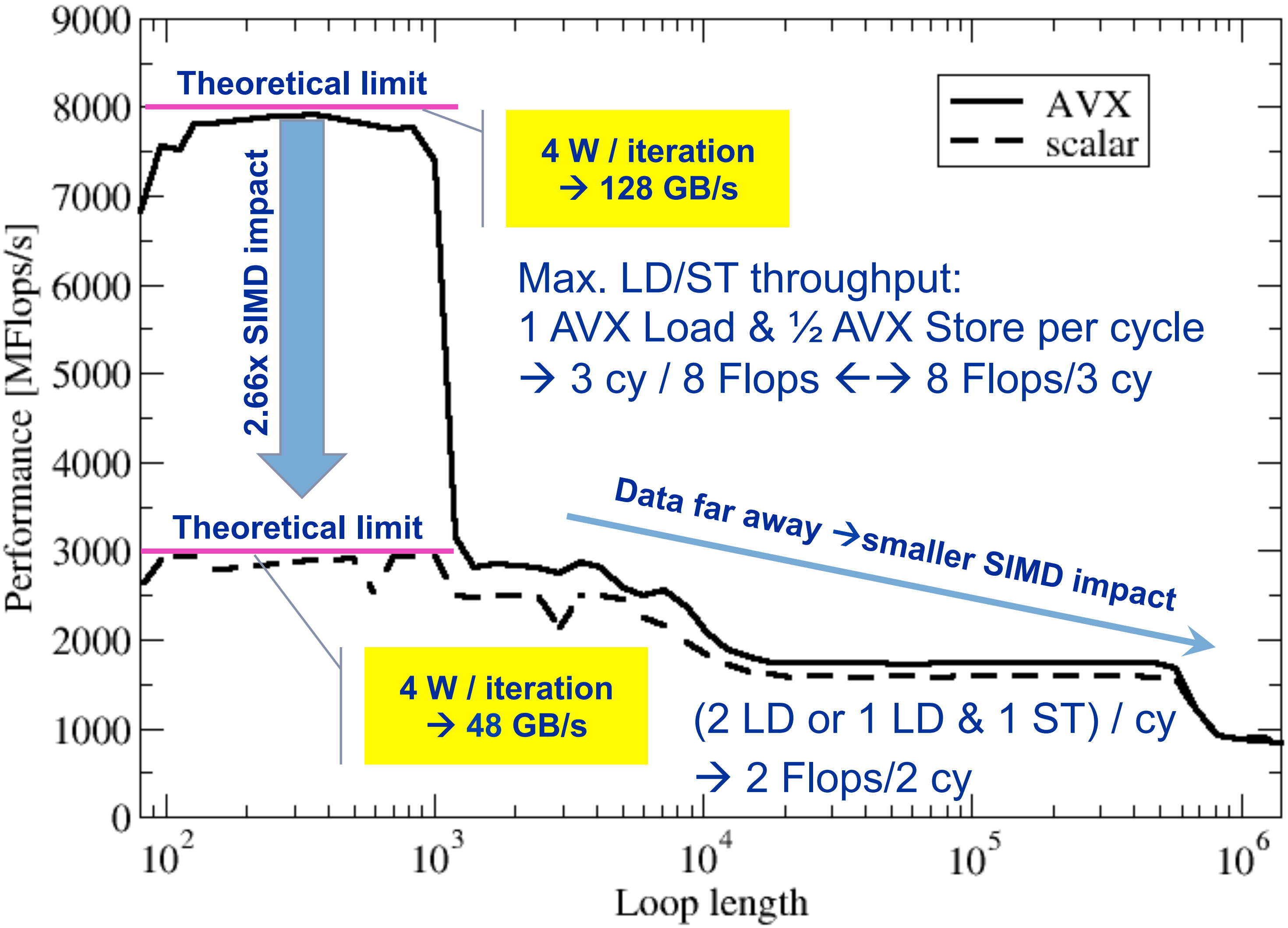
`A(:)=B(:)+C(:)*D(:)` on one Sandy Bridge core (3 GHz)



4 W / iteration
→ 128 GB/s

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

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



4 W / iteration
→ 128 GB/s

Max. LD/ST throughput:
1 AVX Load & ½ AVX Store per cycle
→ 3 cy / 8 Flops ←→ 8 Flops/3 cy

Data far away →smaller SIMD impact

4 W / iteration
→ 48 GB/s

(2 LD or 1 LD & 1 ST) / cy
→ 2 Flops/2 cy

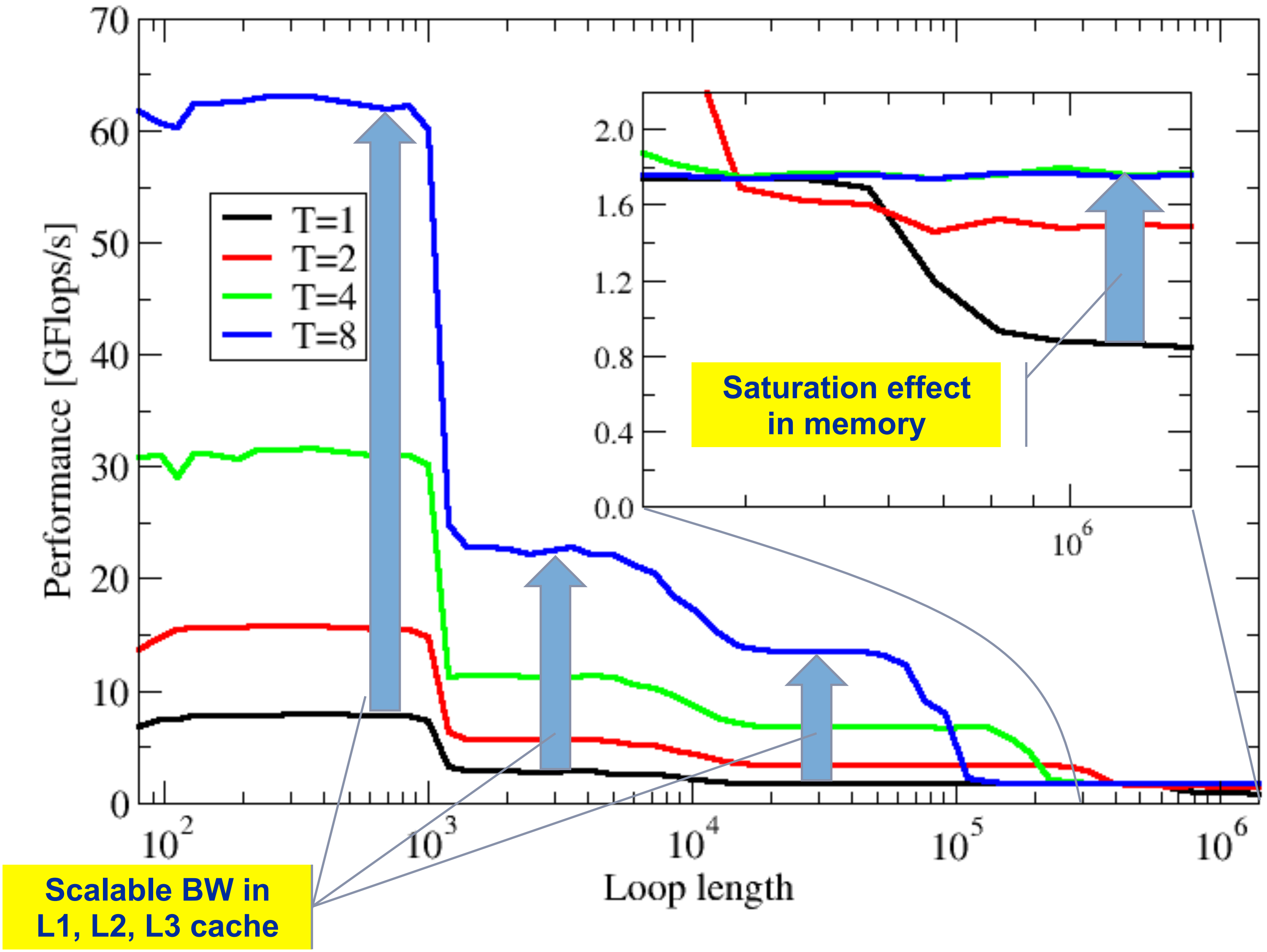
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

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

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

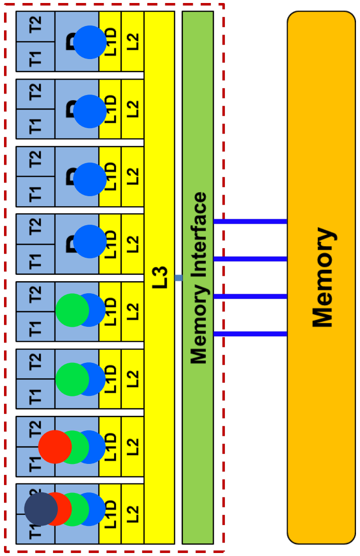
→ pure hardware probing, no impact from OpenMP overhead

Throughput vector triad on Sandy Bridge socket (3 GHz)

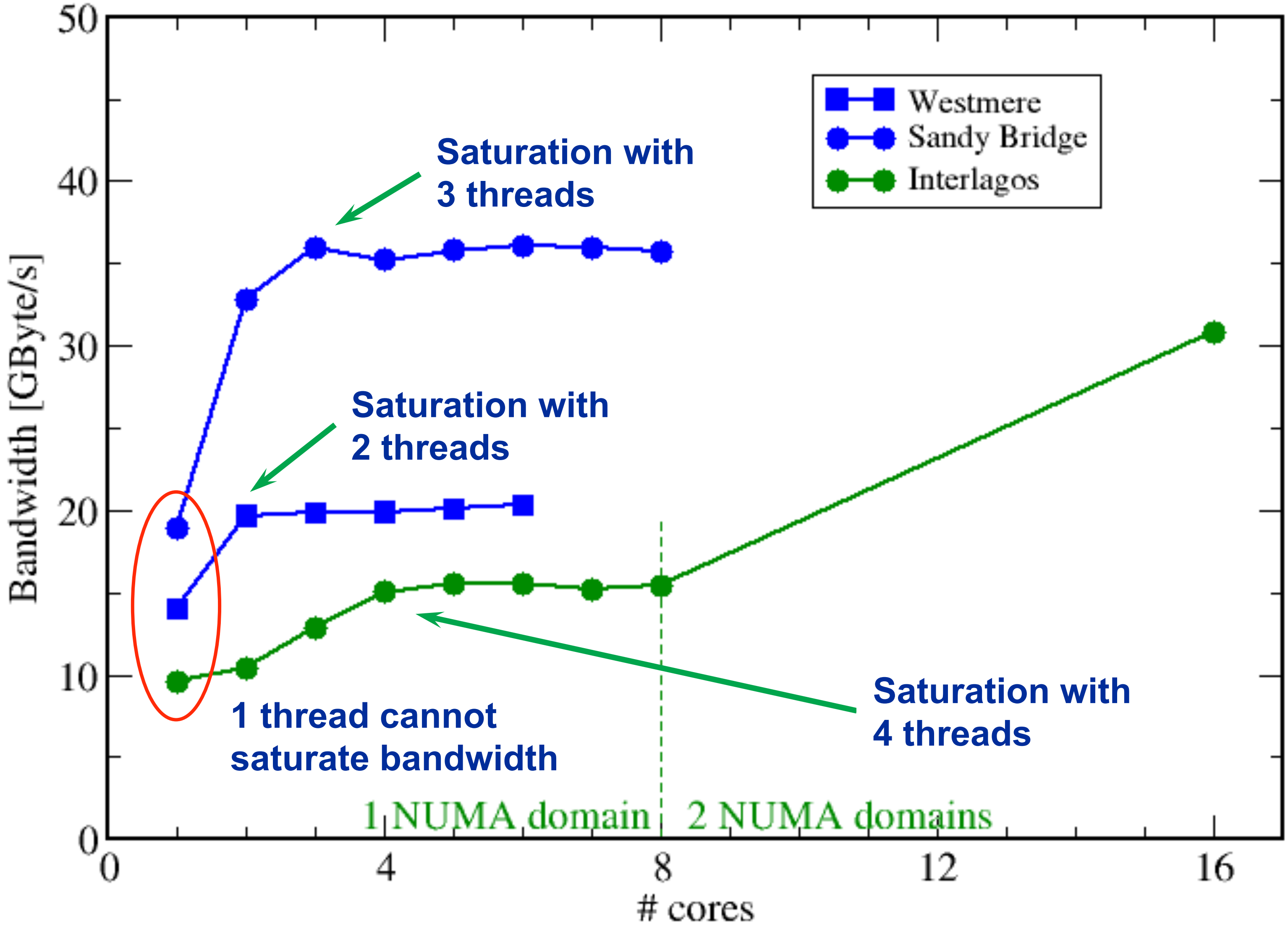


Saturation effect in memory

Scalable BW in L1, L2, L3 cache



Bandwidth limitations: Main Memory
Scalability inside a NUMA domain (V-Triad)



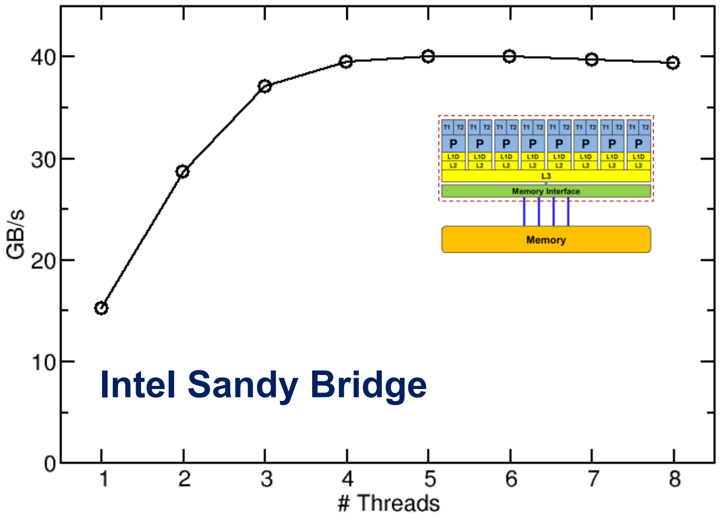
Saturation with 3 threads

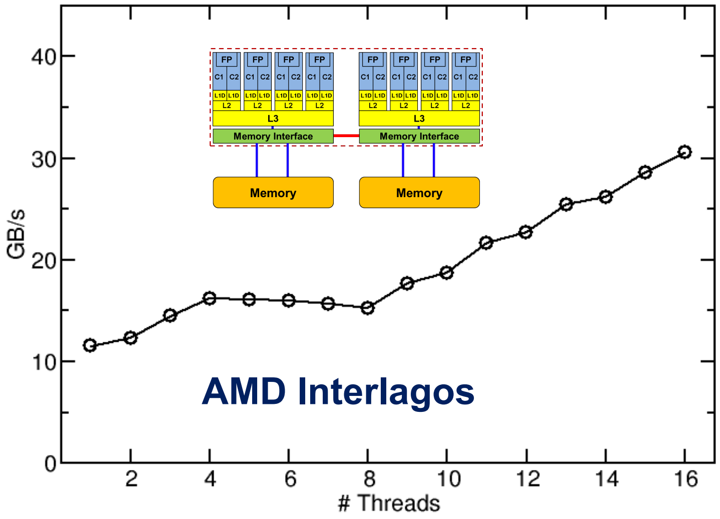
Saturation with 2 threads

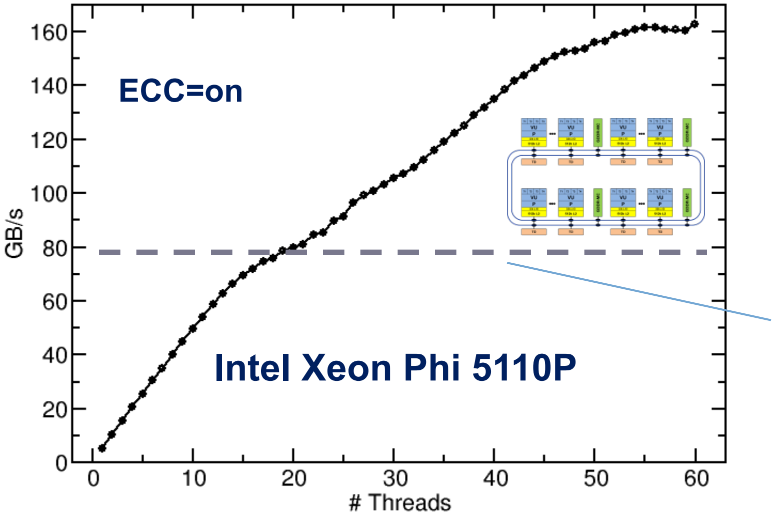
1 thread cannot saturate bandwidth

Saturation with 4 threads

Attainable memory bandwidth: Comparing architectures

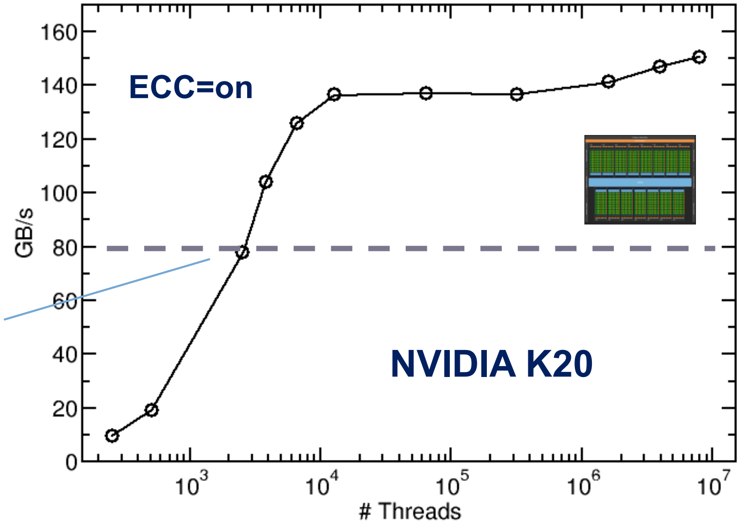

Intel Sandy Bridge


AMD Interlagos


ECC=on

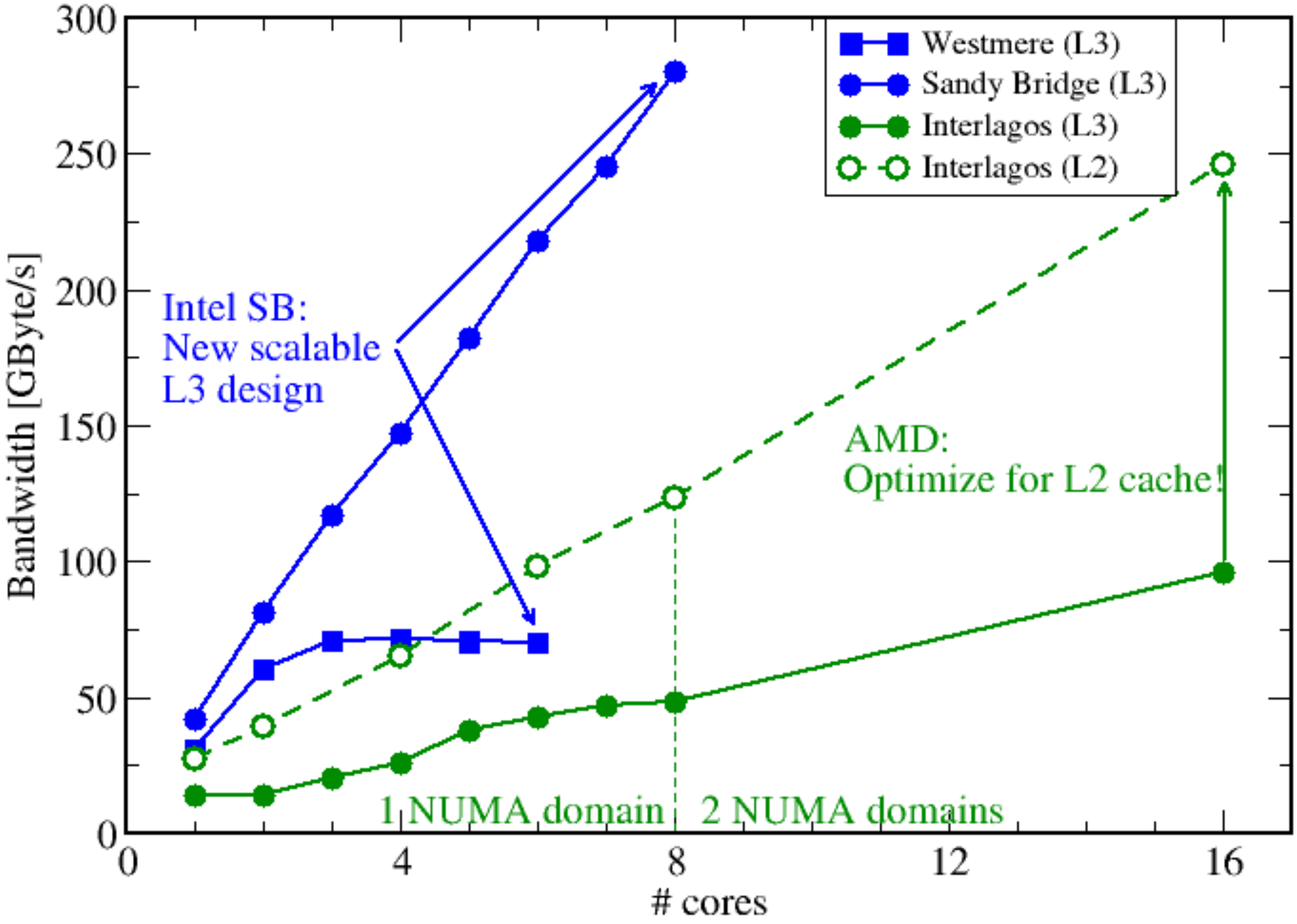
Intel Xeon Phi 5110P

2-socket CPU node


ECC=on

NVIDIA K20

Bandwidth limitations: Outer-level cache
Scalability of shared data paths in L3 cache

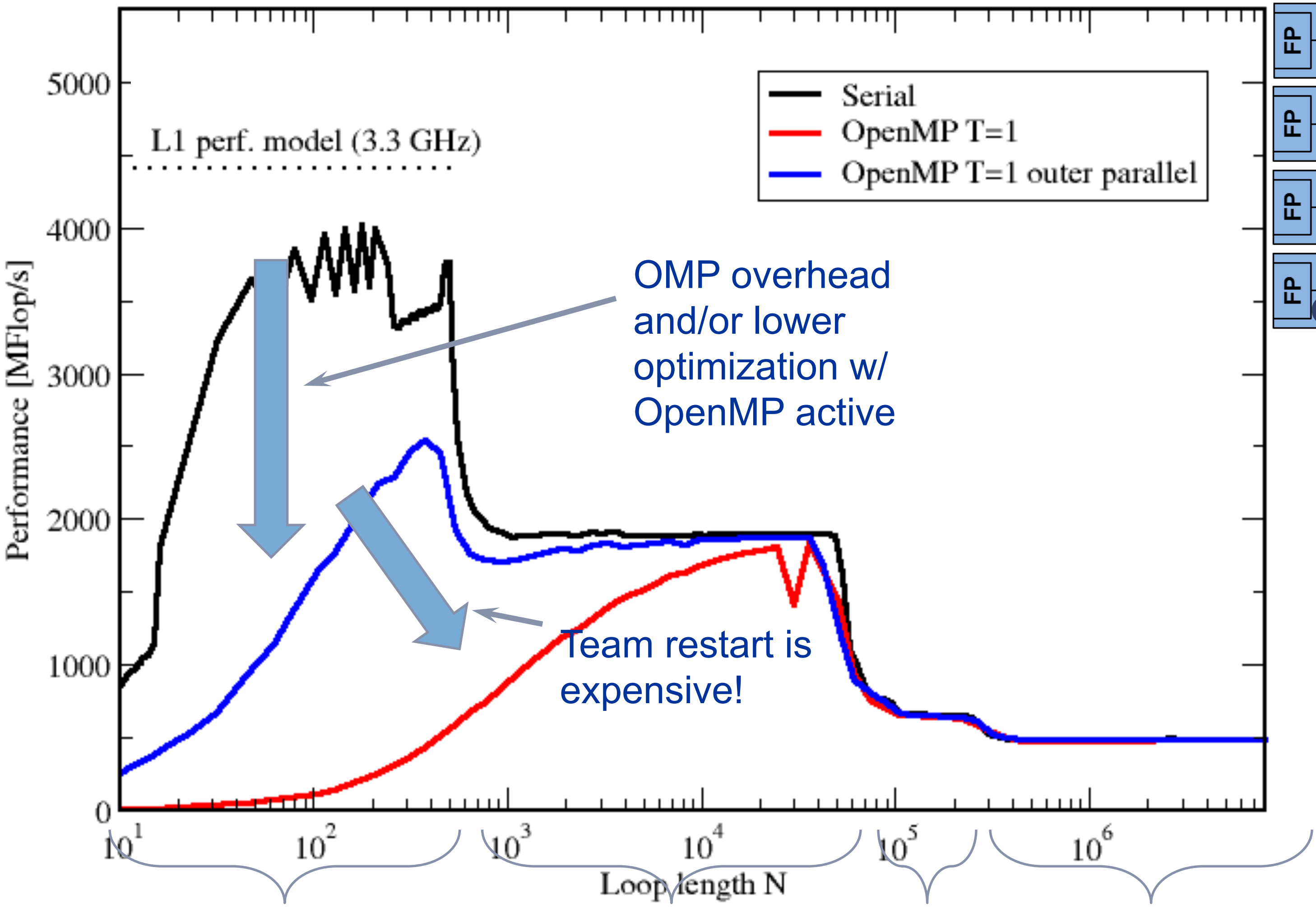


Parallel vector triad benchmark

We use the following code:

```
#pragma omp parallel private(j)
{
for (int j=0; j<niter; j++) {
#pragma omp for
   for (int i=0; i<size; i++) {
      a[i] = b[i] + c[i] * d[i];
   }
}}
```

The parallel vector triad benchmark
Single thread on Cray XE6 Interlagos node



OMP overhead and/or lower optimization w/ OpenMP active

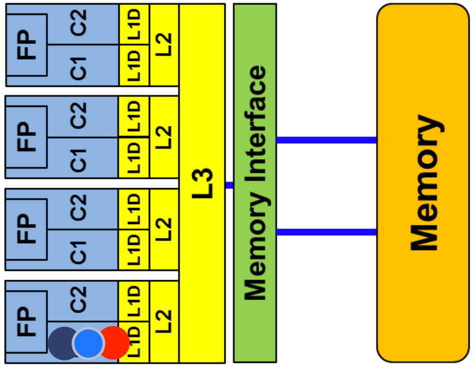
Team restart is expensive!

L1 cache

L2 cache

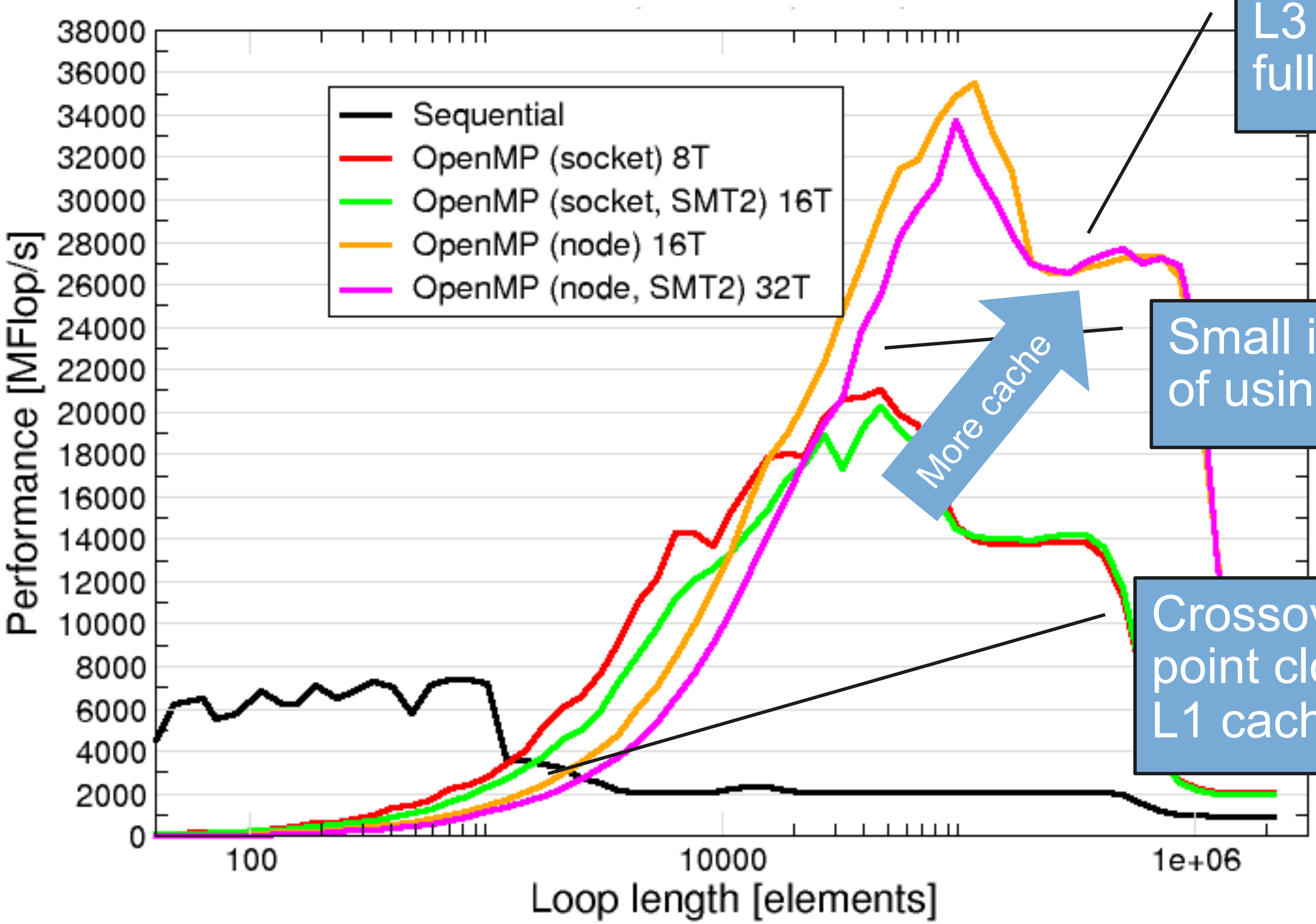
L3 cache

memory



→ use only outer parallel from now on!

Overhead OpenMP Synchronization SandyBridge-EP ICC 13.1

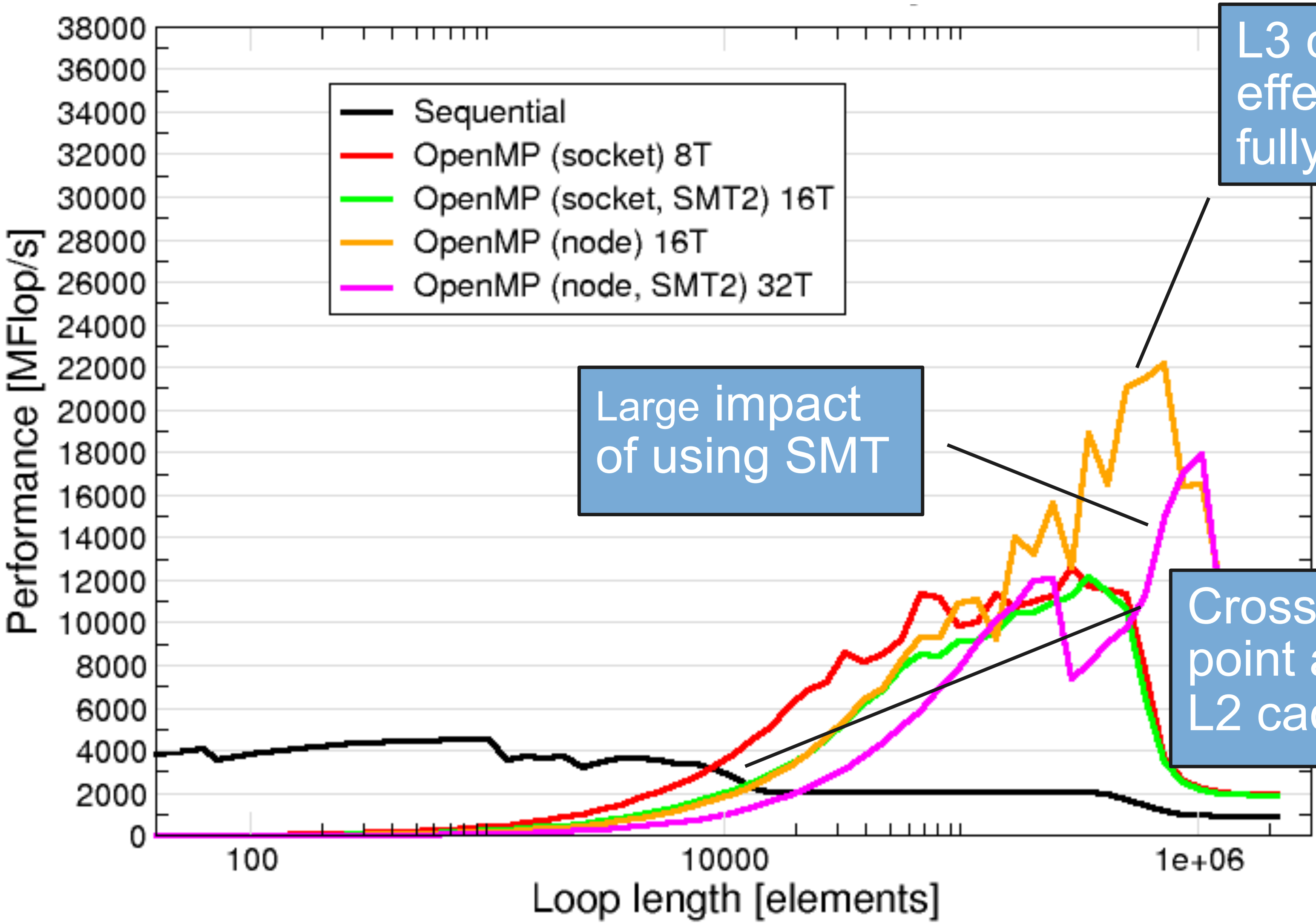


L3 cache fully usable

Small impact of using SMT

Crossover point close to L1 cachesize

Overhead Syncronization OpenMP SandyBridge-EP GCC 4.7.0



L3 cache effect not fully usable

Large impact of using SMT

Crossover point at end of L2 cachesize

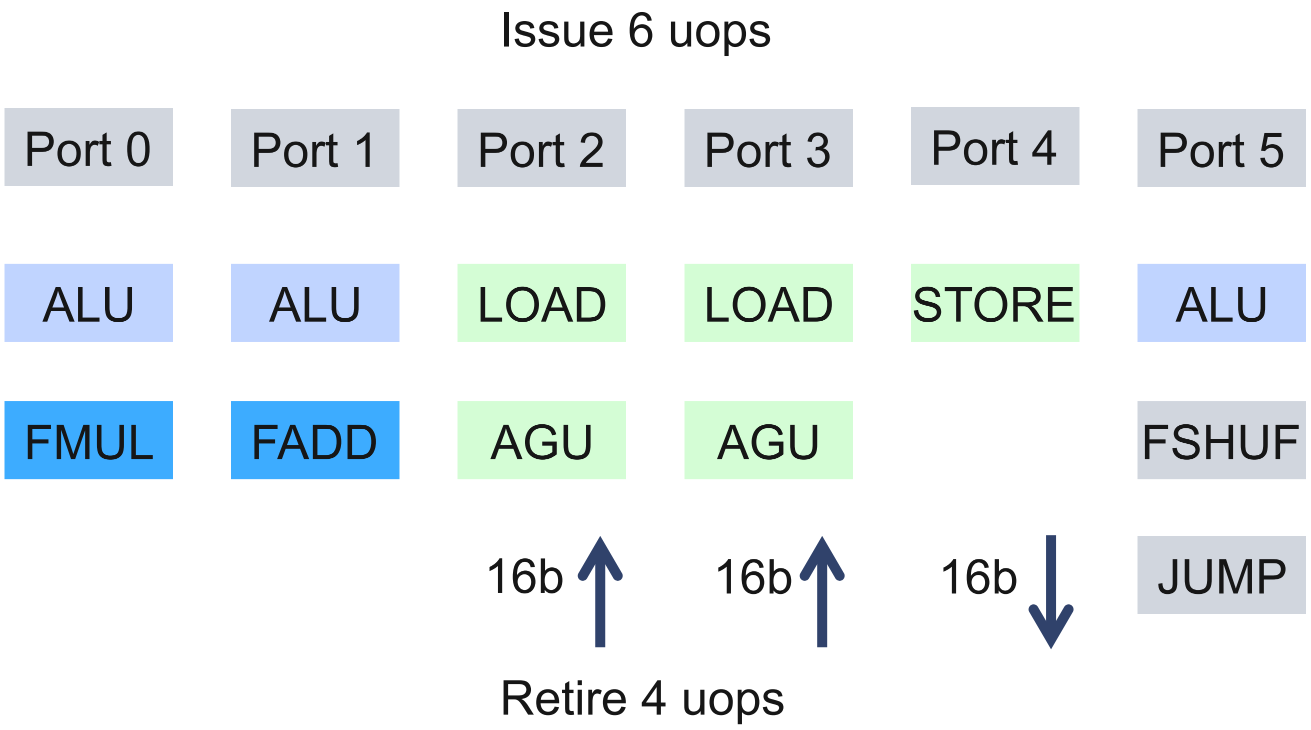
End part 2

“SIMPLE” PERFORMANCE MODELING: THE ROOFLINE MODEL

Loop-based performance modeling:
Execution vs. data transfer

Preliminary: Estimating Instruction throughput

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



Issue 6 uops

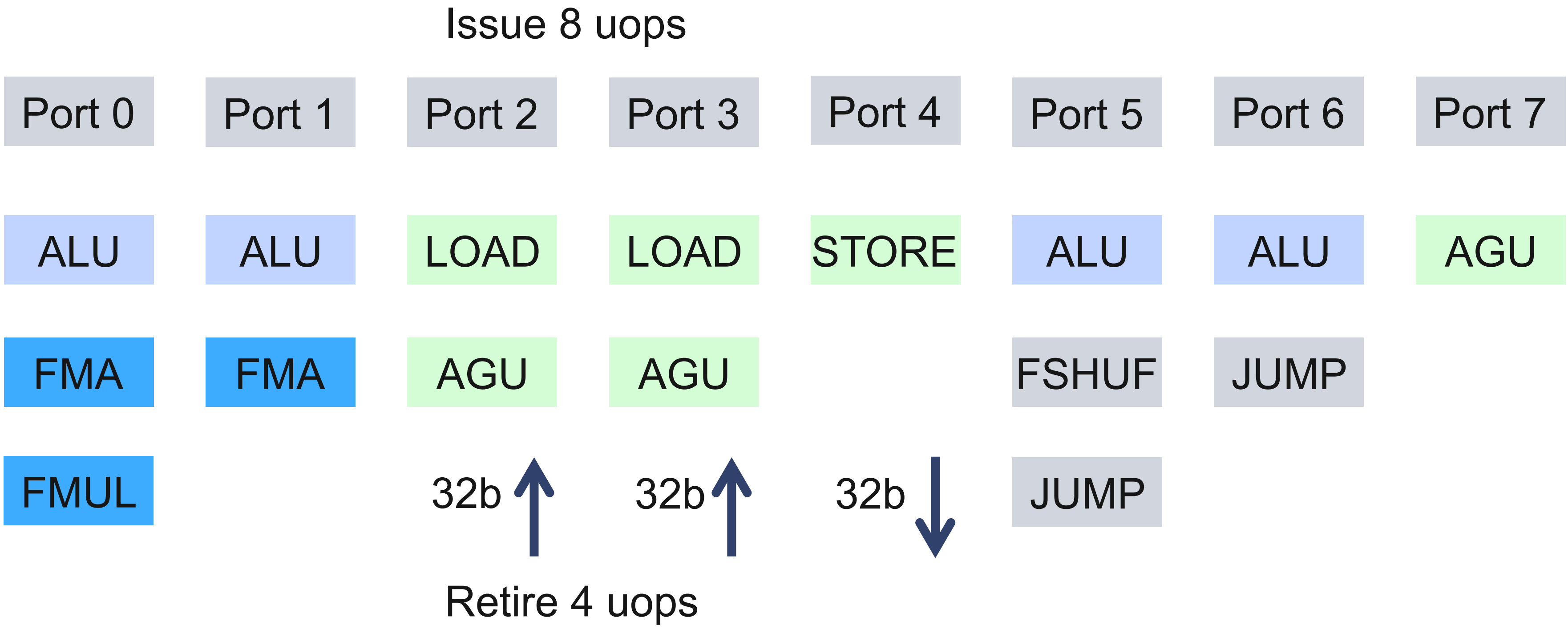
Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
ALU	ALU	LOAD	LOAD	STORE	ALU
FMUL	FADD	AGU	AGU		FSHUF
		16b ↑	16b ↑	16b ↓	JUMP

Retire 4 uops

SandyBridge

Preliminary: Estimating Instruction throughput

Every new generation provides incremental improvements.
The OOO microarchitecture is a blend between P6 (Pentium Pro) and P4 (Netburst) architectures.



Issue 8 uops

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
ALU	ALU	LOAD	LOAD	STORE	ALU	ALU	AGU
FMA	FMA	AGU	AGU		FSHUF	JUMP	
FMUL		32b ↑	32b ↑	32b ↓	JUMP		

Retire 4 uops

Haswell

Exercise: Estimate performance of triad on SandyBridge @3GHz

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

How many cycles to process one 64byte cacheline?

64byte equivalent to 8 scalar iterations or **2** AVX vector iterations.

Cycle 1: load and ½ store and mult and add

Cycle 2: load and ½ store

Cycle 3: load

Answer: 6 cycles

Exercise: Estimate performance of triad on SandyBridge @3GHz

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

Whats the performance in GFlops/s and bandwidth in MBytes/s ?

One AVX iteration (3 cycles) performs 4x2=8 flops.

(3 GHZ / 3 cycles) * 4 updates * 2 flops/update = **8 GFlops/s**

4 GUPS/s * 4 words/update * 8byte/word = **128 GBytes/s**

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:

$$P = \min(P_{max}, \overset{[F/B]}{I}\ \overset{[B/s]}{b_s})$$

[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)

“Simple” Roofline: The vector triad

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)

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

"Simple" Roofline: The vector triad

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)**

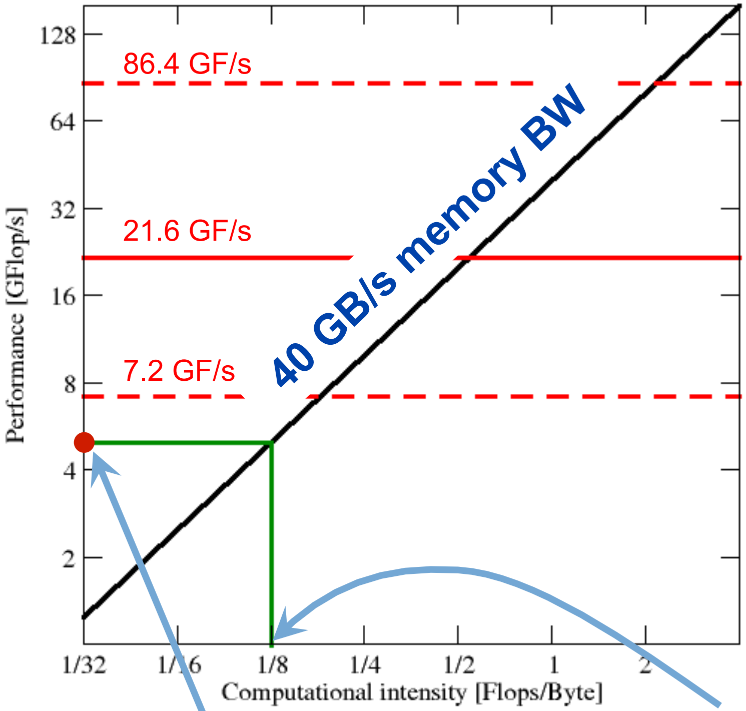
- P_{peak} = **1008 Gflop/s** (60 FP units x (8+8) Flops/cy x 1.05 GHz)
- P_{max}? → Observe LD/ST throughput maximum of 1 Load or 1 Store per cycle → 4 cy / 16 Flops → P_{max} = **252 Gflop/s** (25% of peak)

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

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



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)

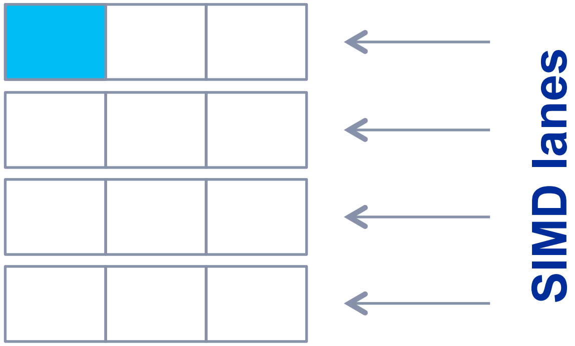
***P* = 5 Gflop/s**

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
```

ADD pipes utilization:



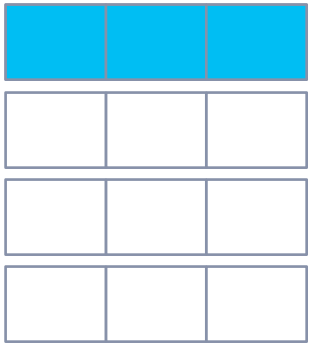
→ 1/12 of ADD peak

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:



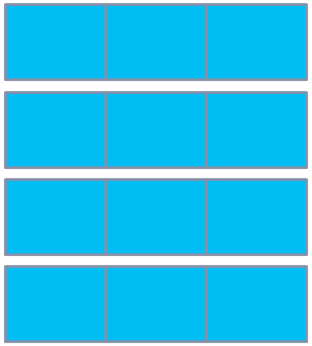
→ 1/4 of ADD peak

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
```

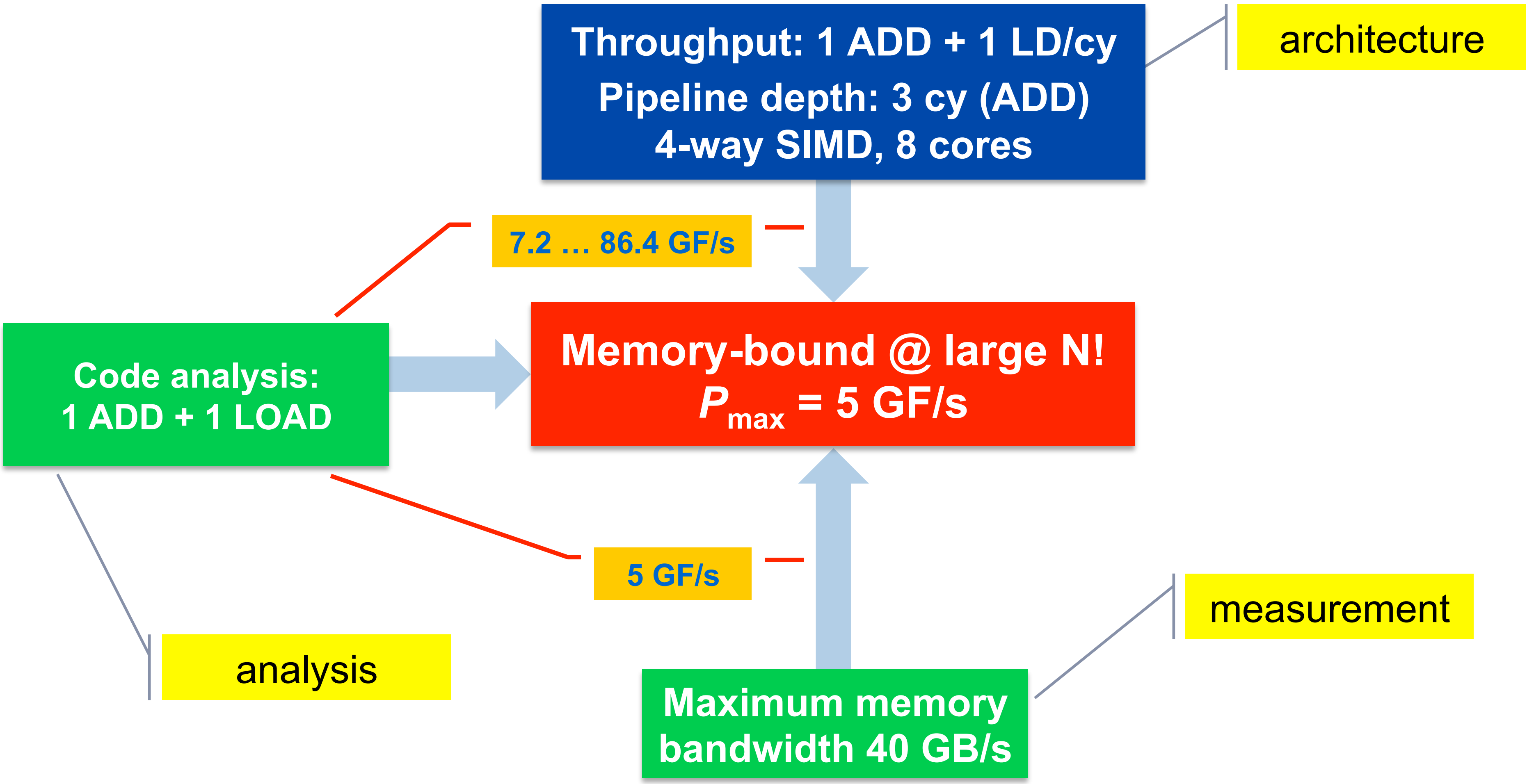
ADD pipes utilization:



→ ADD peak

Input to the roofline model

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



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

architecture

7.2 … 86.4 GF/s

Code analysis:
1 ADD + 1 LOAD

Memory-bound @ large N!
P_{max} = 5 GF/s

5 GF/s

analysis

measurement

Maximum memory bandwidth 40 GB/s

Assumptions for the Roofline Model

The roofline formalism is based on some (crucial) **assumptions**:

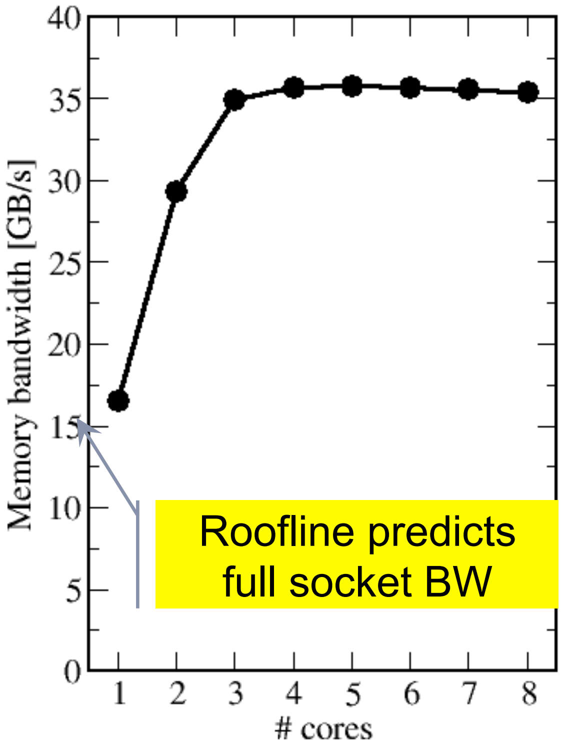
- There is a clear concept of **“work” vs. “traffic”**
  - “work” = flops, updates, iterations…
  - “traffic” = required data to do “work”
- **Attainable bandwidth of code = input parameter!** Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications
- **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**

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!

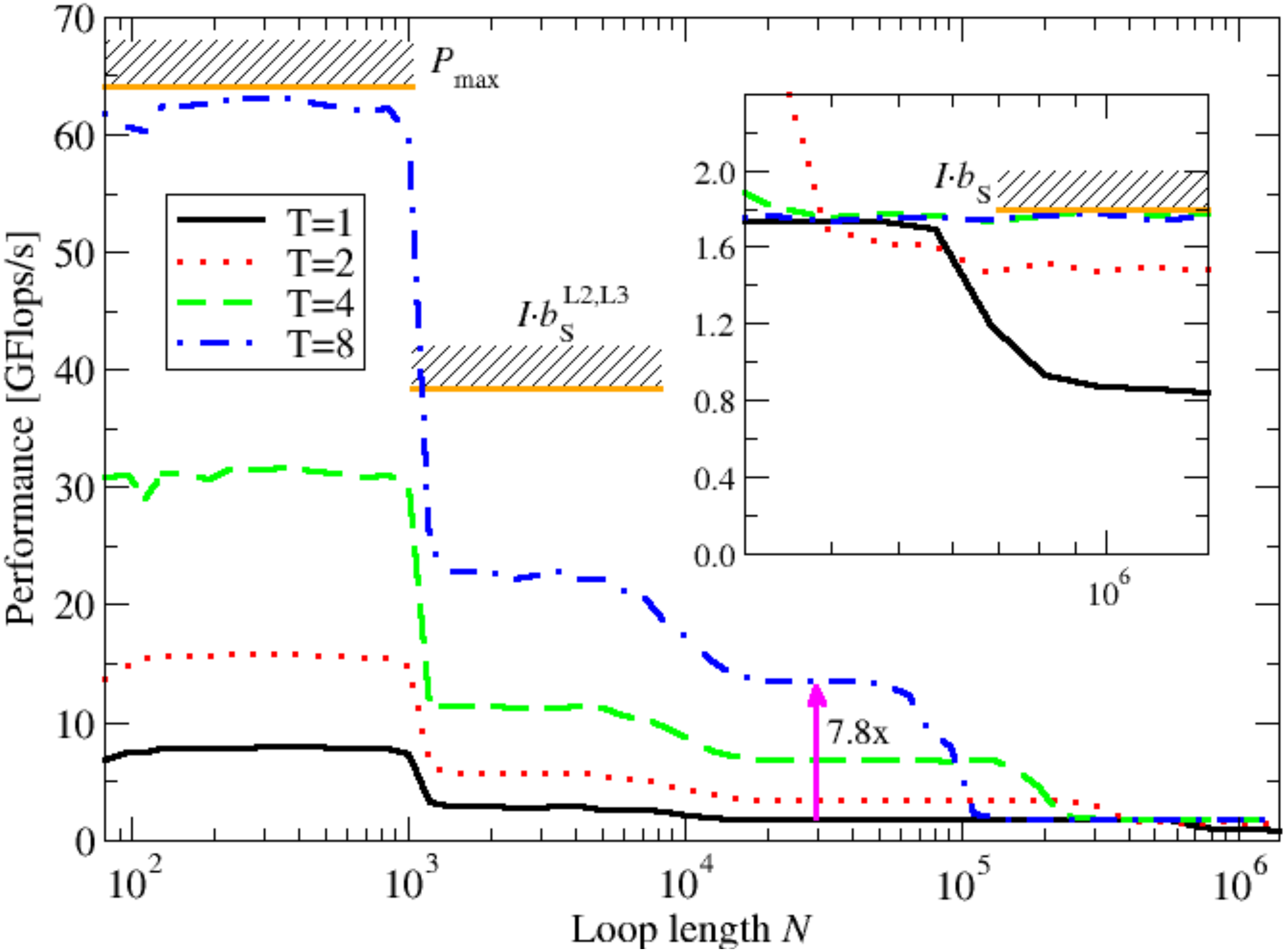
`A(:)=B(:)+C(:)*D(:)`



Roofline predicts full socket BW

ECM model gives more insight

Where the roofline model fails



ECM Model

ECM = “Execution-Cache-Memory”

Assumptions:

Single-core execution time is composed of

1. In-core execution
2. Data transfers in the memory hierarchy

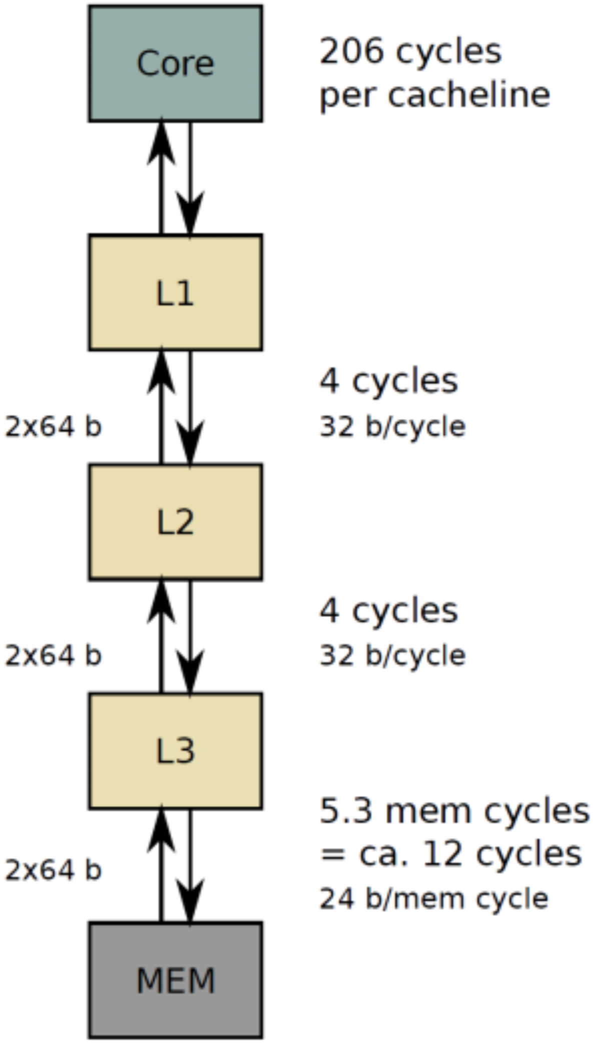
Data transfers may or may not overlap with each other or with in-core execution

Scaling is linear until the relevant bottleneck is reached

Input:

Same as for Roofline

+ **data transfer times** in hierarchy

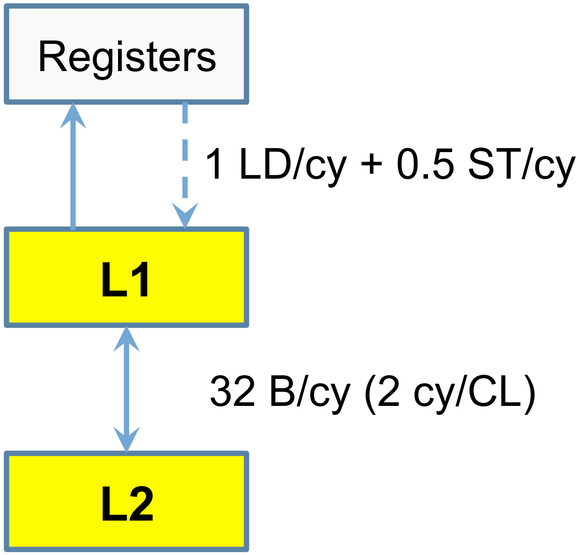


Example: Schönauer Vector Triad in L2 cache

REPEAT[`A(:) = B(:) + C(:) * D(:)`] @ double precision

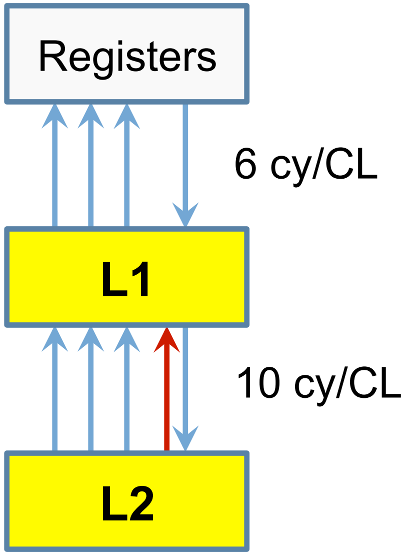
Analysis for Sandy Bridge core w/ AVX (unit of work: 1 cache line)

Machine characteristics:



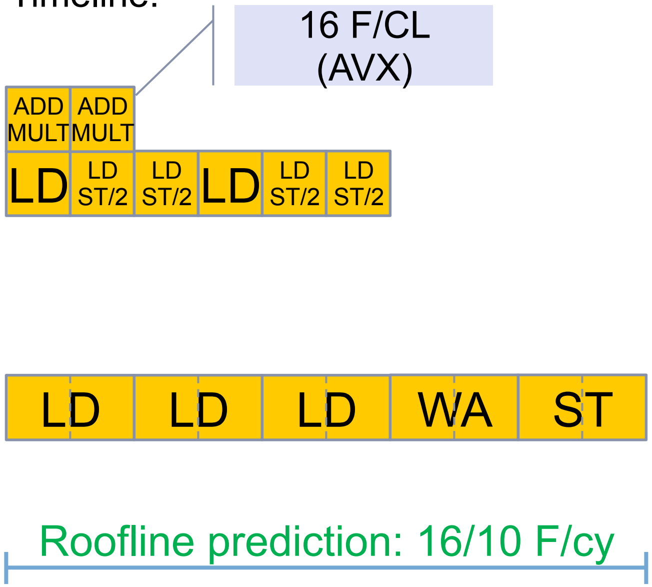
Arithmetic:
1 ADD/cy+ 1 MULT/cy

Triad analysis (per CL):



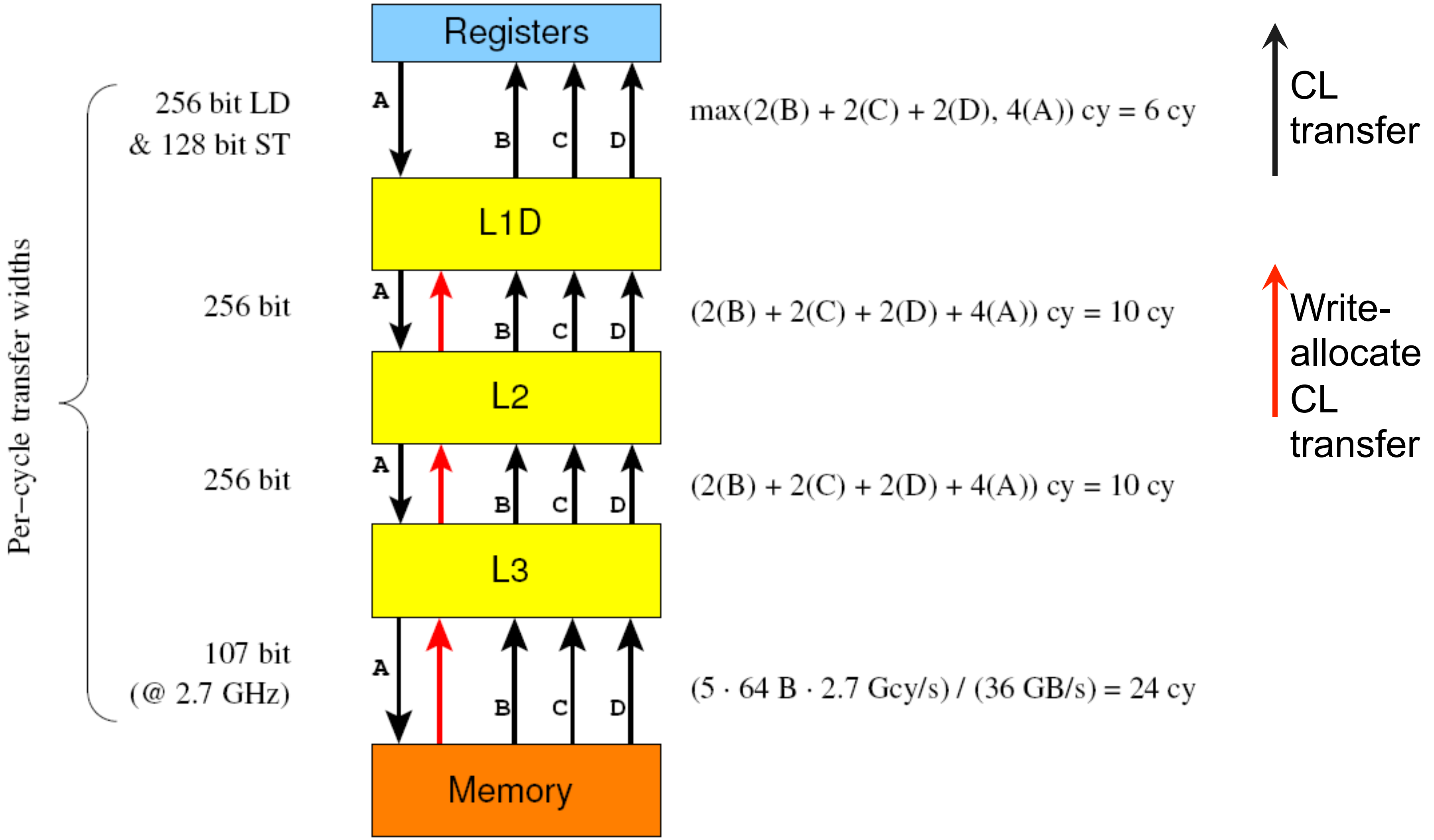
Arithmetic:
AVX: 2 cy/CL

Timeline:

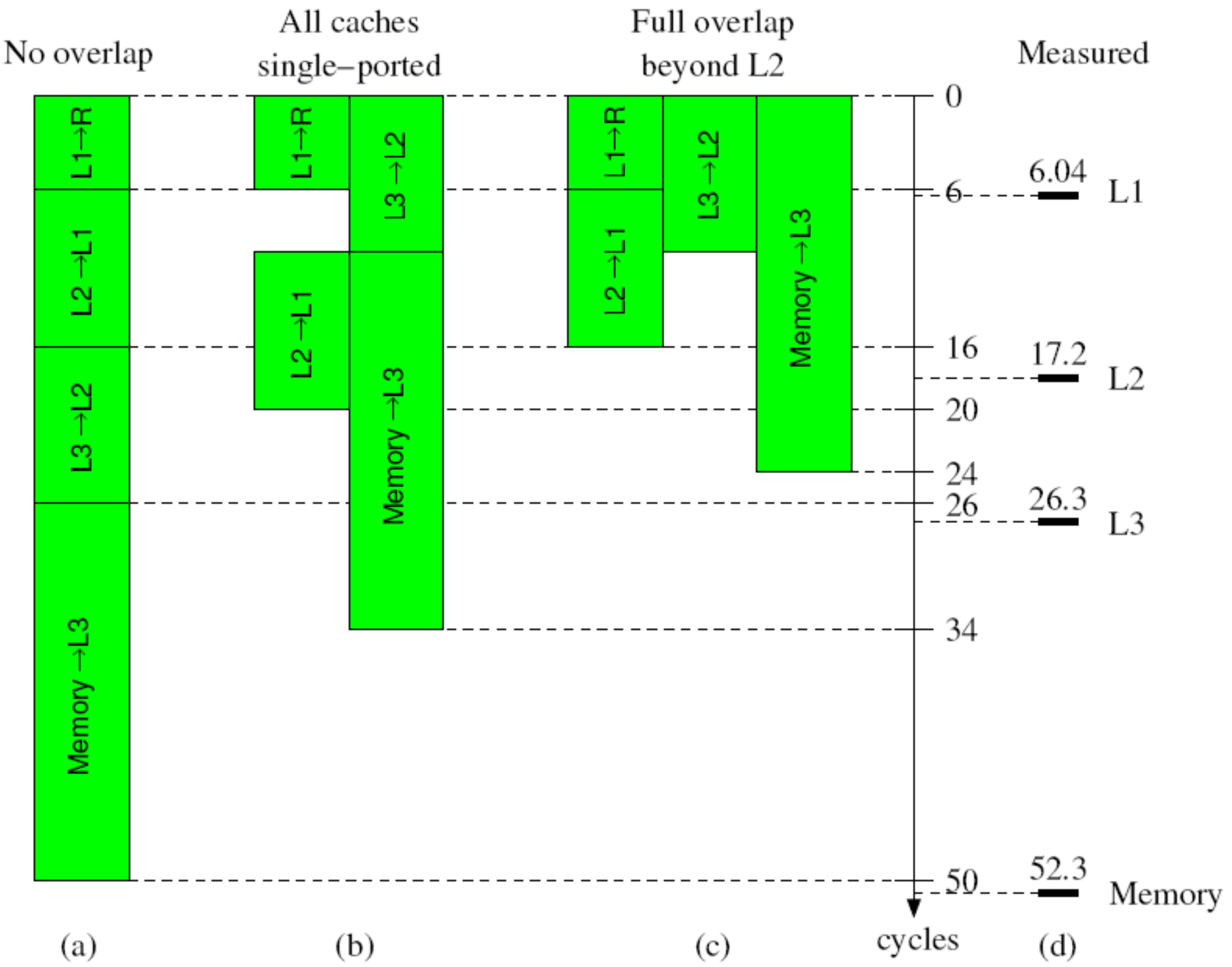


Measurement: 16F / ≈17cy

Example: ECM model for Schönauer Vector Triad `A(:)=B(:)+C(:)*D(:)` with AVX


Registers
L1D
L2
L3
Memory
A
B
C
D
Per–cycle transfer widths
256 bit LD & 128 bit ST
256 bit
256 bit
107 bit (@ 2.7 GHz)
max(2(B) + 2(C) + 2(D), 4(A)) cy = 6 cy
(2(B) + 2(C) + 2(D) + 4(A)) cy = 10 cy
(2(B) + 2(C) + 2(D) + 4(A)) cy = 10 cy
(5 · 64 B · 2.7 Gcy/s) / (36 GB/s) = 24 cy
CL transfer
Write-allocate CL transfer

Full vs. partial vs. no overlap



Results suggest **no overlap**!

Multicore scaling in the ECM model

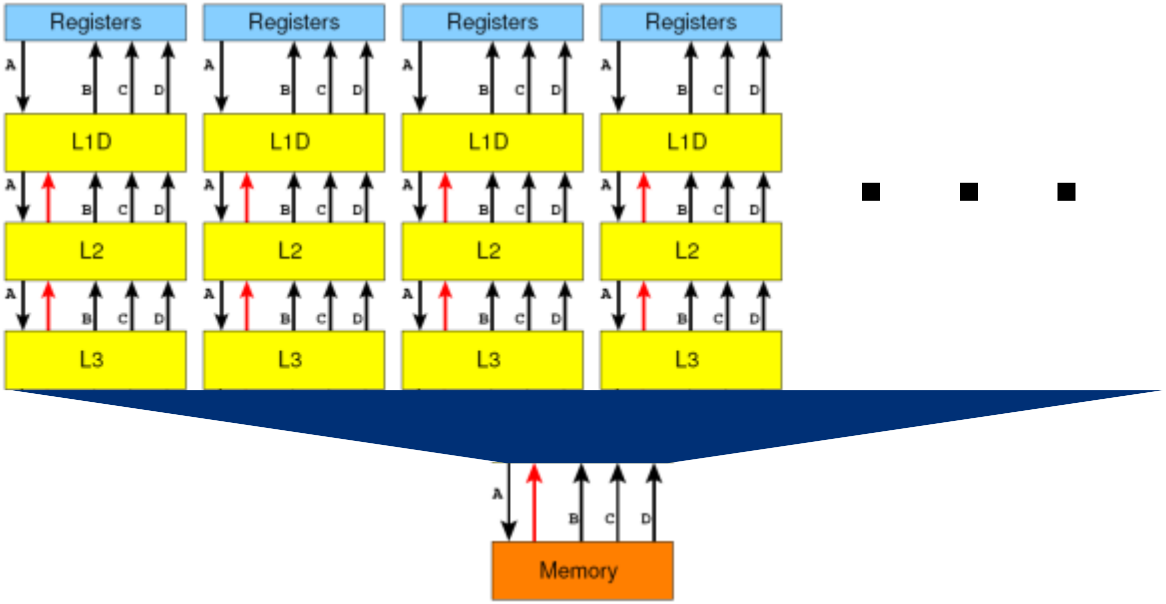
Identify relevant **bandwidth bottlenecks**

- L3 cache
- Memory interface

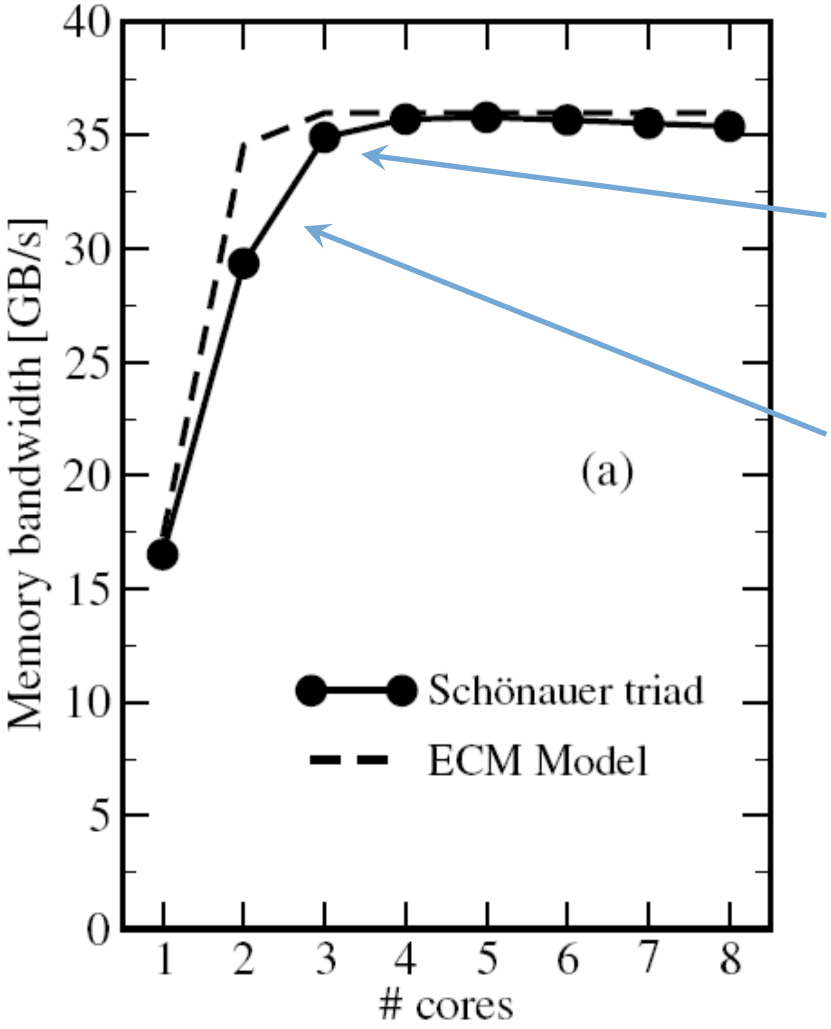
Scale single-thread performance until **first bottleneck** is hit:

$$\mathbf{P(t)=min(tP_0,P_{roof})}, \text{ with } \mathbf{P_{roof}=min(P_{max},I\ b_S)}$$

Example: Scalable L3 on Sandy Bridge



ECM prediction vs. measurements for `A(:)=B(:)+C(:)*D(:)`, no overlap



Model: Scales until saturation sets in

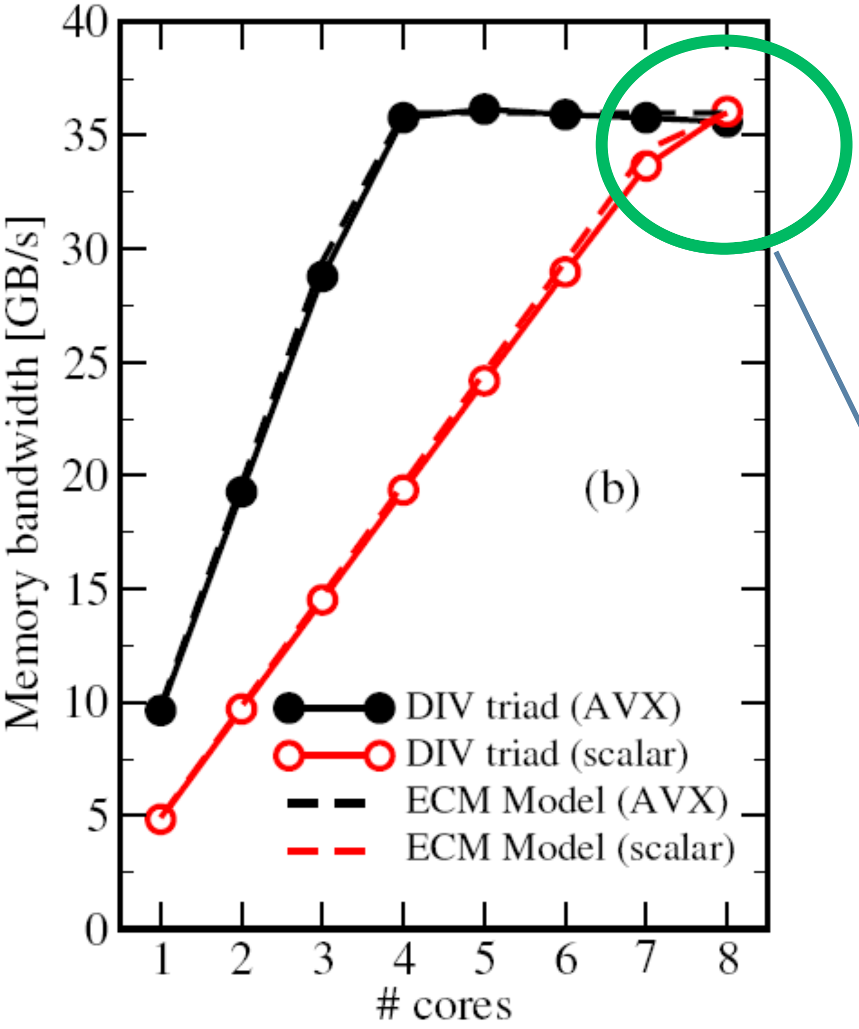
Saturation point (# cores) well predicted

Measurement: scaling not perfect

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

Check: Use “overlappable” kernel code

ECM prediction vs. measurements for `A(:)=B(:)+C(:)/D(:)` with full overlap



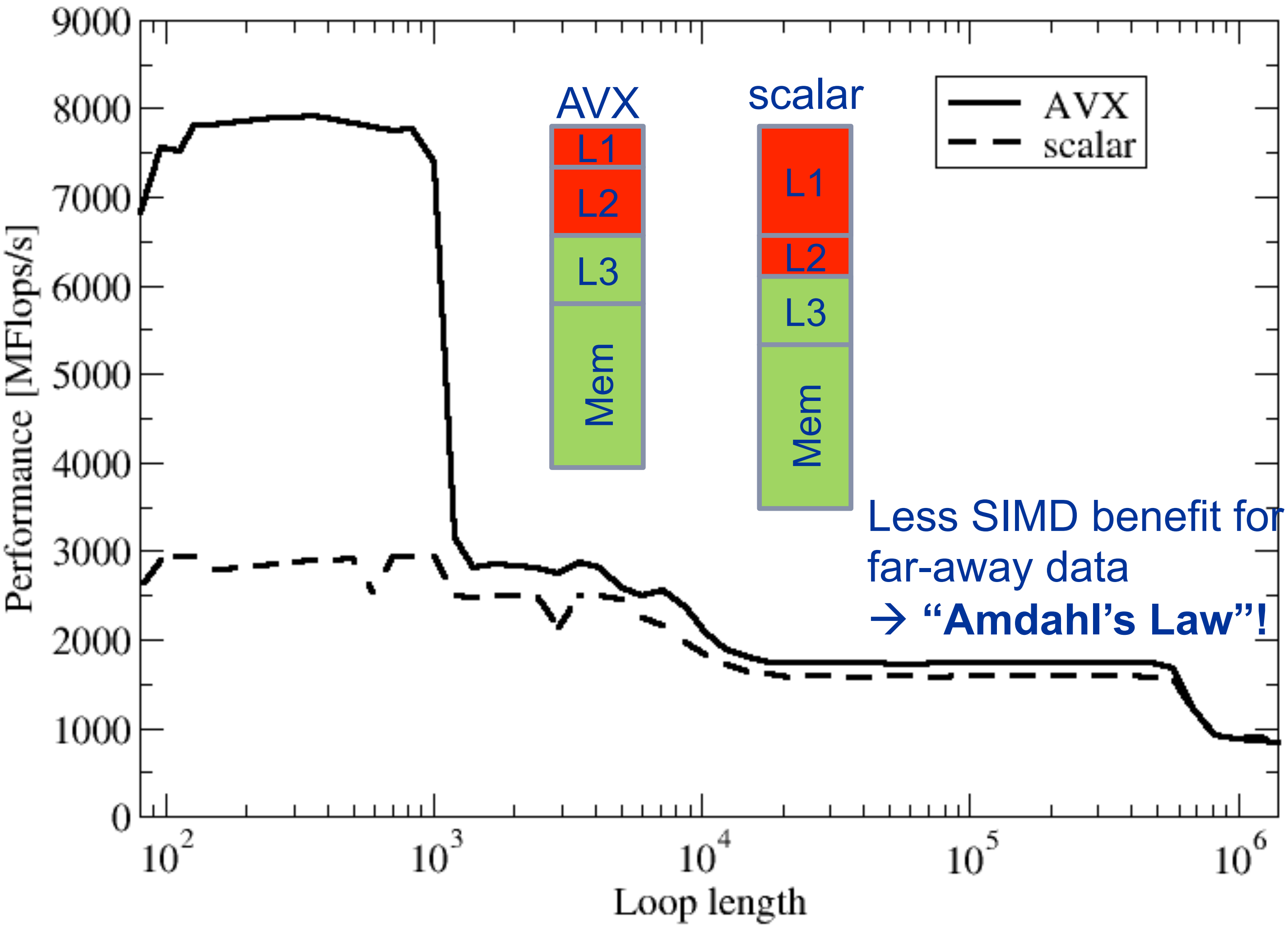
In-core execution is dominated by divide operation (44 cycles with AVX, 22 scalar)

→ **Almost perfect agreement** with ECM model

Parallelism "heals" bad single-core performance … just barely!

The impact of in-core optimizations

Reme



Less SIMD benefit for far-away data

→ **"Amdahl's Law"!**

Summary: The ECM Model

Saturation effects are ubiquitous; understanding them gives us opportunity to

- Find out about optimization opportunities
- Save energy by letting cores idle → see power model later on
- Putting idle cores to better use → asynchronous communication, functional parallelism

ECM correctly describes several effects

- Saturation for memory-bound loops
- Diminishing returns of in-core optimizations for far-away data
- Parallelism heals bad sequential code (sometimes…)
- Get clean picture of different runtime contributions

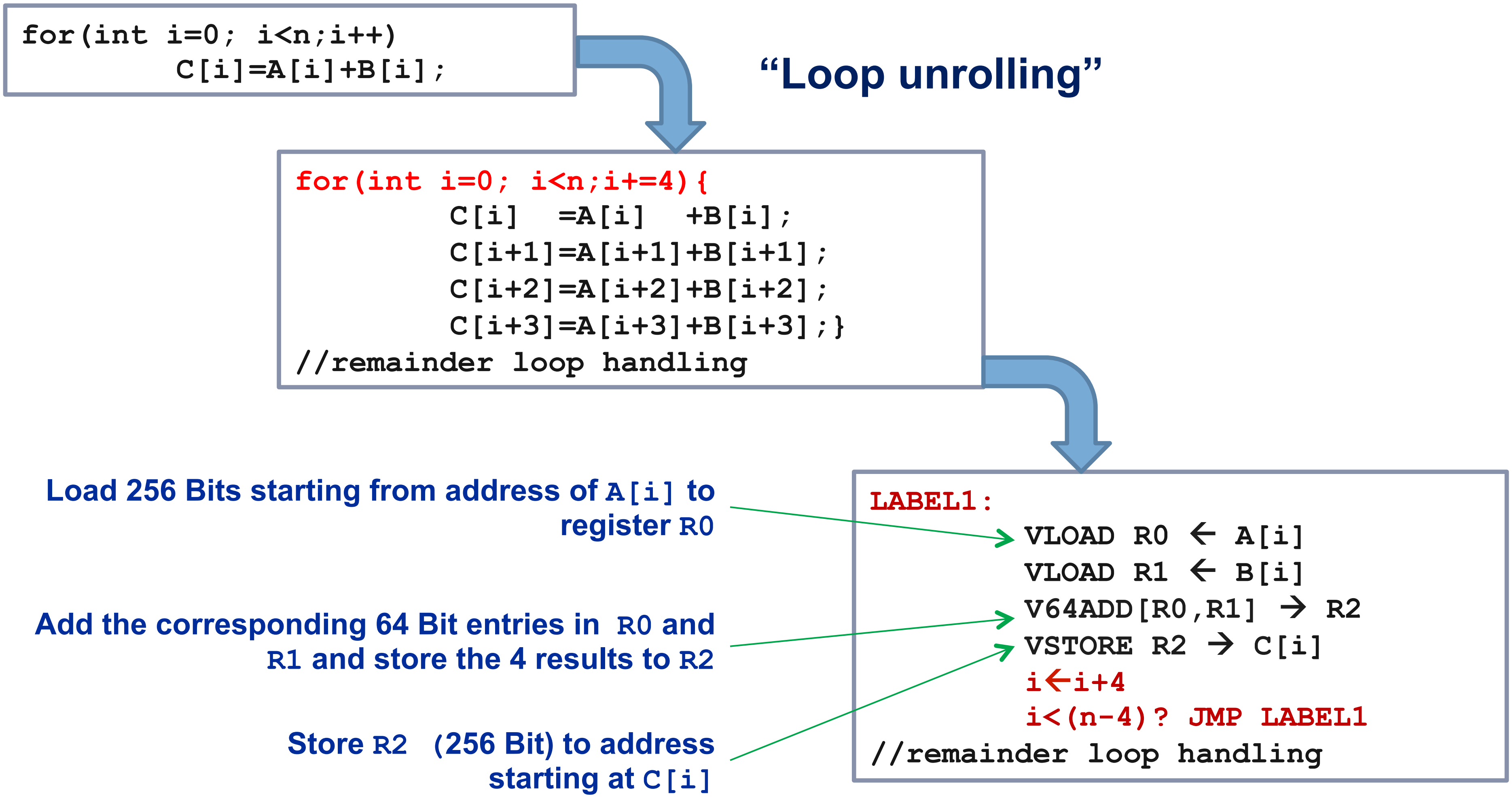
Simple models work best. Do not try to complicate things unless it is really necessary!

EXPLOITING PARALLEL RESOURCES ON MULTICORE NODES

- SIMD

SIMD processing – Basics

Steps (**done by the compiler**) for “SIMD processing”



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

“Loop unrolling”

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

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

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

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

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

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 or use **`restrict`**(C99)

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): Optimizations

```
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
```

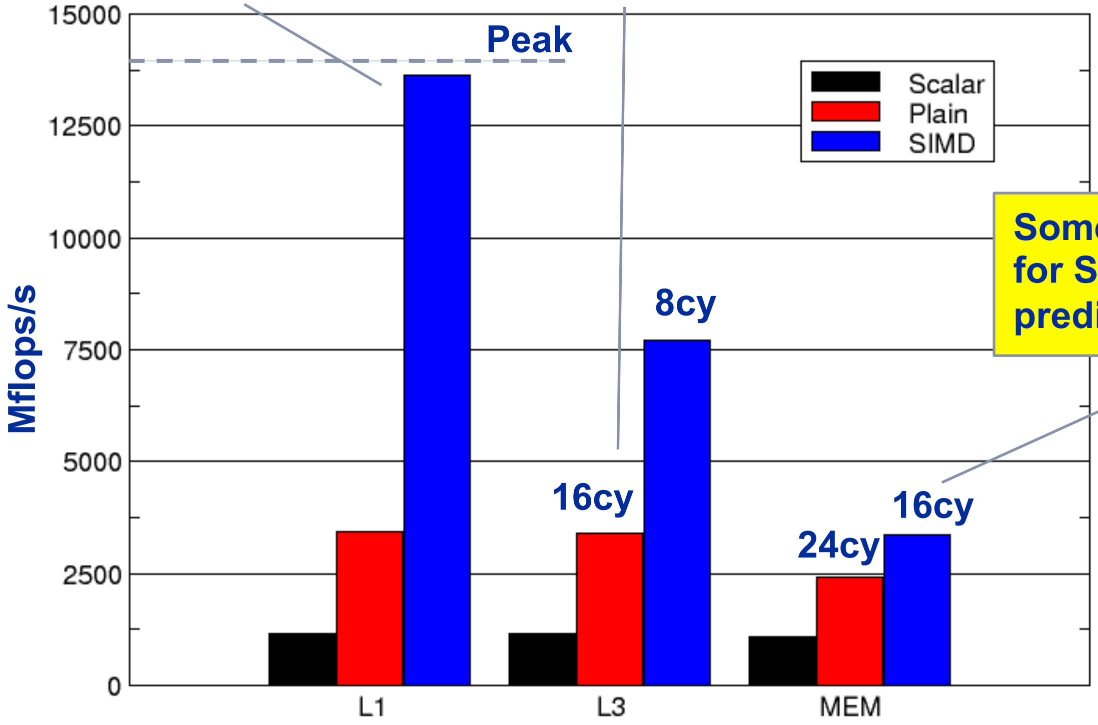
AVX SIMD vectorization

SIMD processing – single-threaded

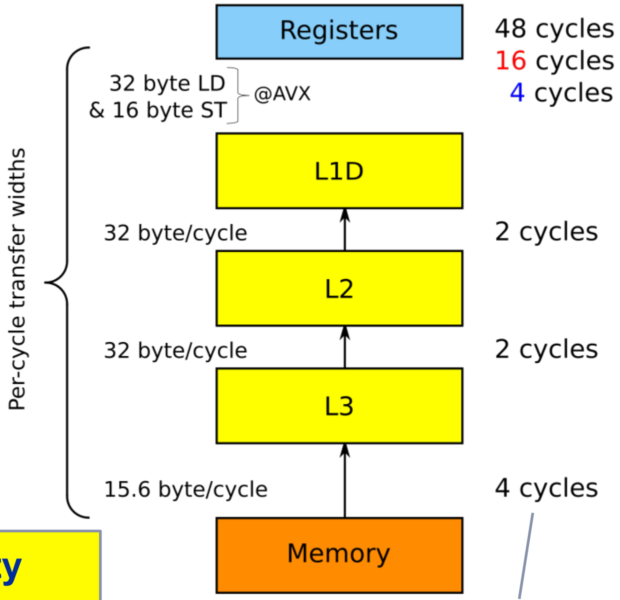
SIMD influences instruction execution in the core – other bottlenecks stay the same!

Full benefit in L1 cache

Data transfers are overlapped with execution



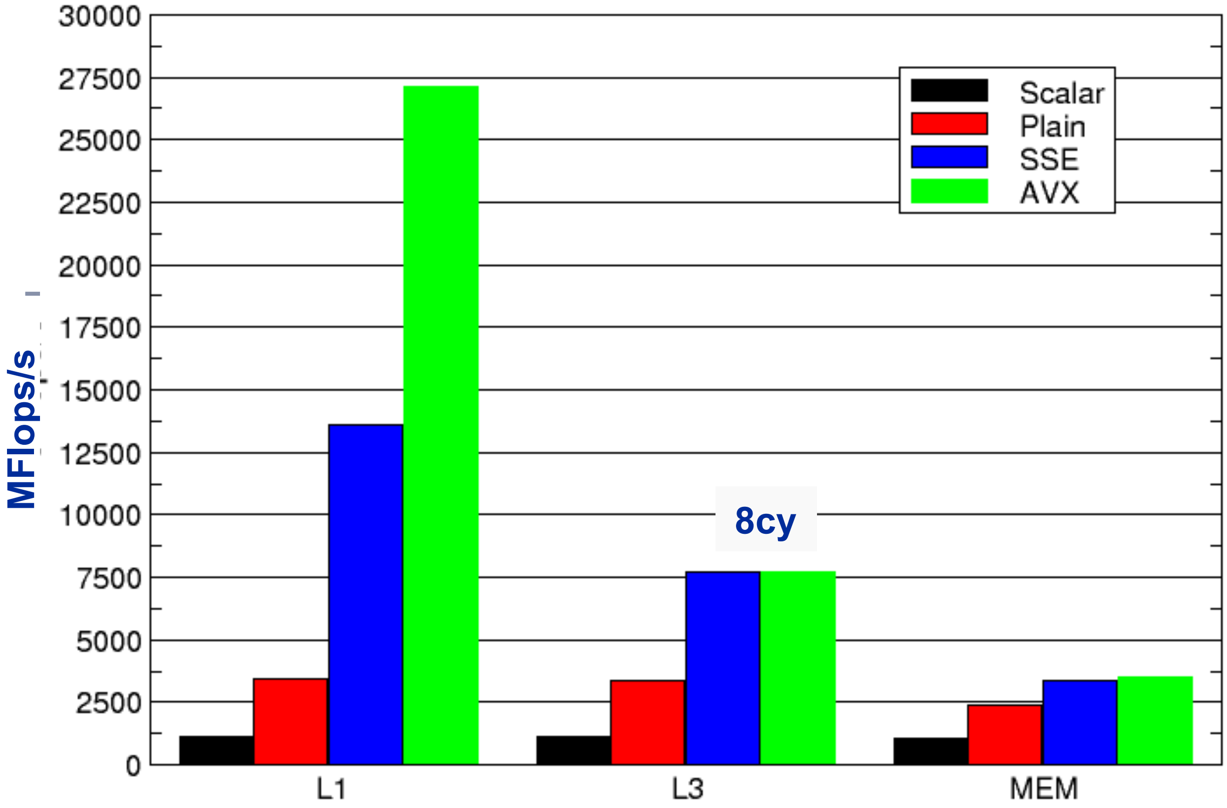
Some penalty for SIMD (12 cy predicted)



Per-cacheline cycle counts

Execution	Cache	Memory
48		
16	4	4
4		

And with AVX?



8cy

L3 Cache

SSE	**8 cycles**
AVX	**6 cycles**

	Cache	**Memory**
48		
16		
4	**4**	**4**
2		

diminishing returns (Amdahl)

With preloading:

AVX down to less than 7 cycles (8309 MFlops/s)

SIMD processing – Full chip (all cores)
Influence of SMT

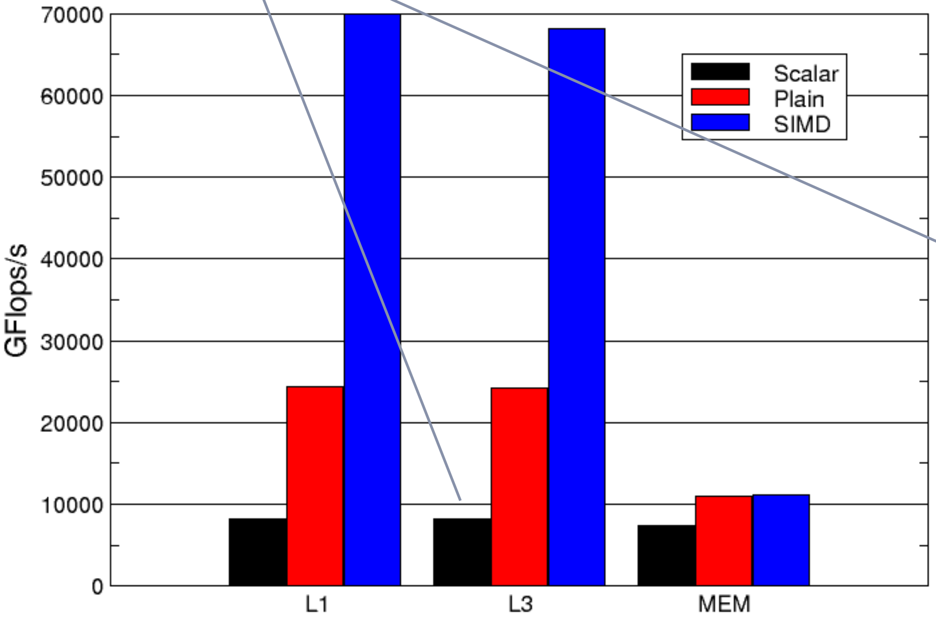
Bandwidth saturation is the primary performance limitation on the chip level!

Full scaling using SMT due to bubbles in pipeline

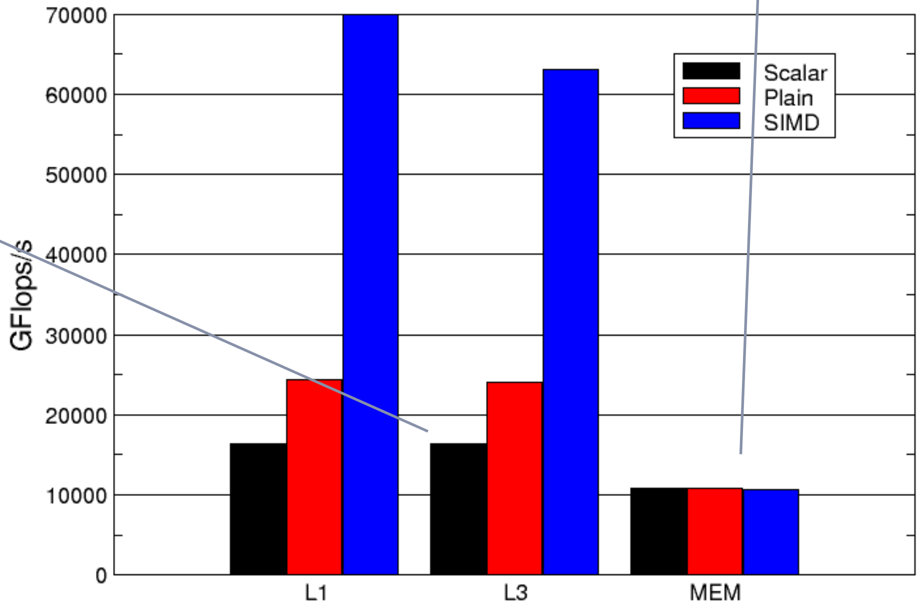
Conclusion: If the code saturates the bottleneck, all variants are acceptable!

All variants saturate the memory bandwidth

8 threads on physical cores



16 threads using SMT



How to leverage SIMD

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

Rules for vectorizable loops

optional

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

EXPLOITING PARALLEL RESOURCES ON MULTICORE NODES

- ccNUMA

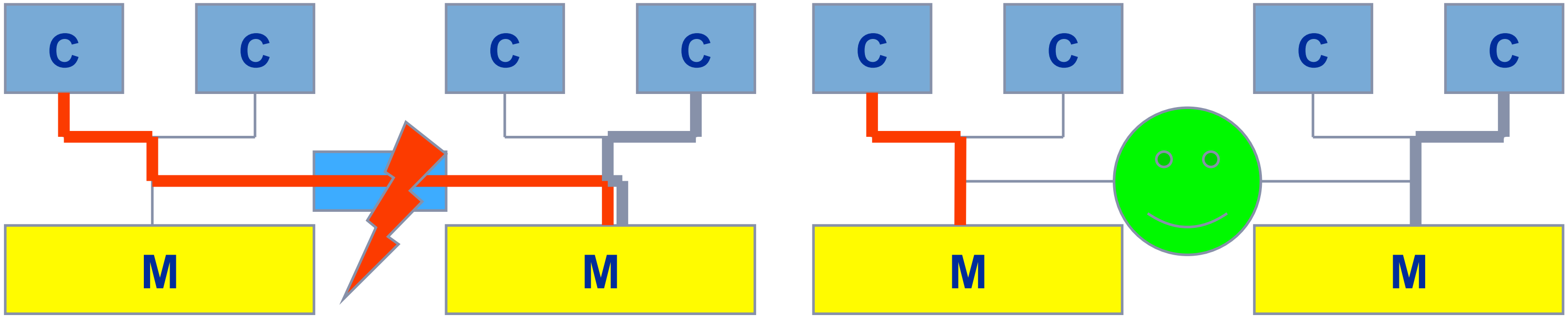
ccNUMA performance problems

"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?



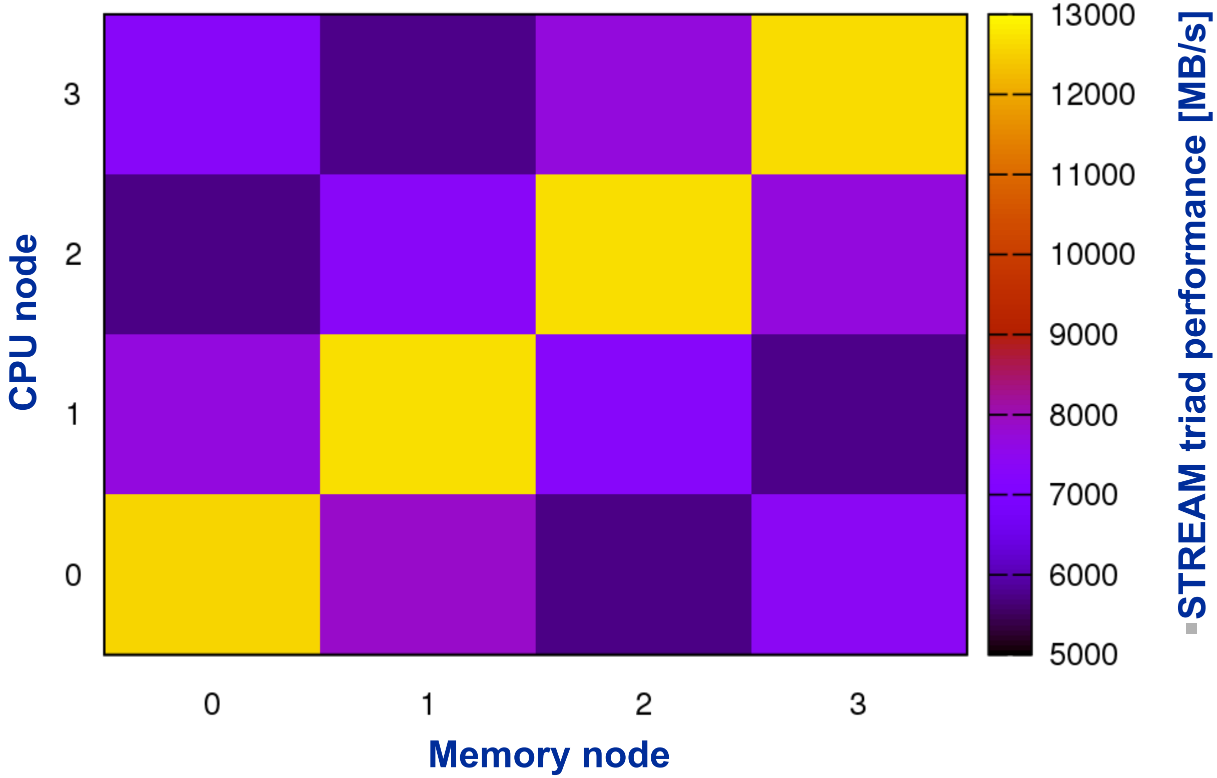
Page placement is implemented in units of OS pages (often 4kB)

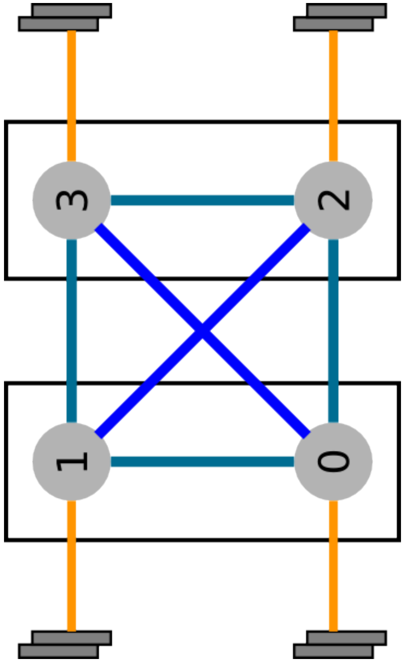
Cray XE6 Interlagos node

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





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:

ccNUMA map scan

```
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
```

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

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 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

How about **global objects**?

- Better not use them
- If communication vs. computation is favorable, might consider **properly placed copies** of global data

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

Consider using performance counters

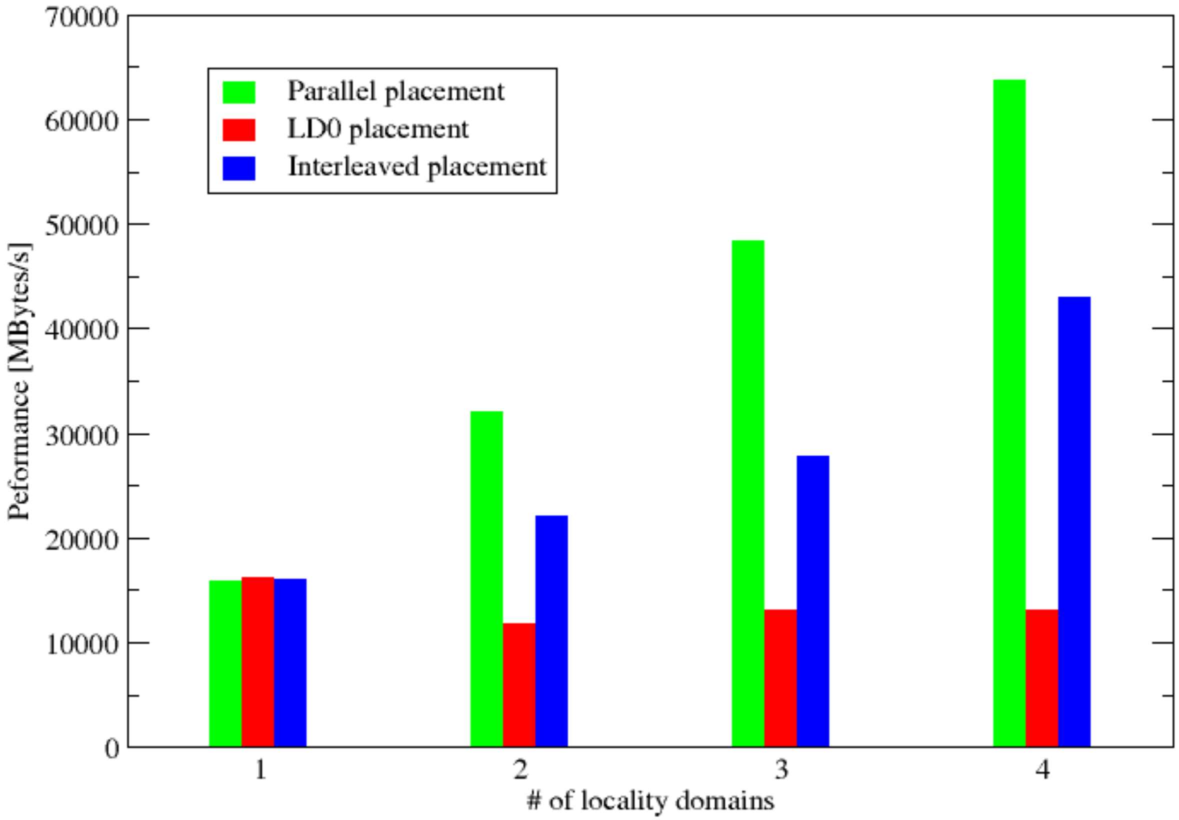
The curse and blessing of interleaved placement:

OpenMP STREAM on a Cray XE6 Interlagos node

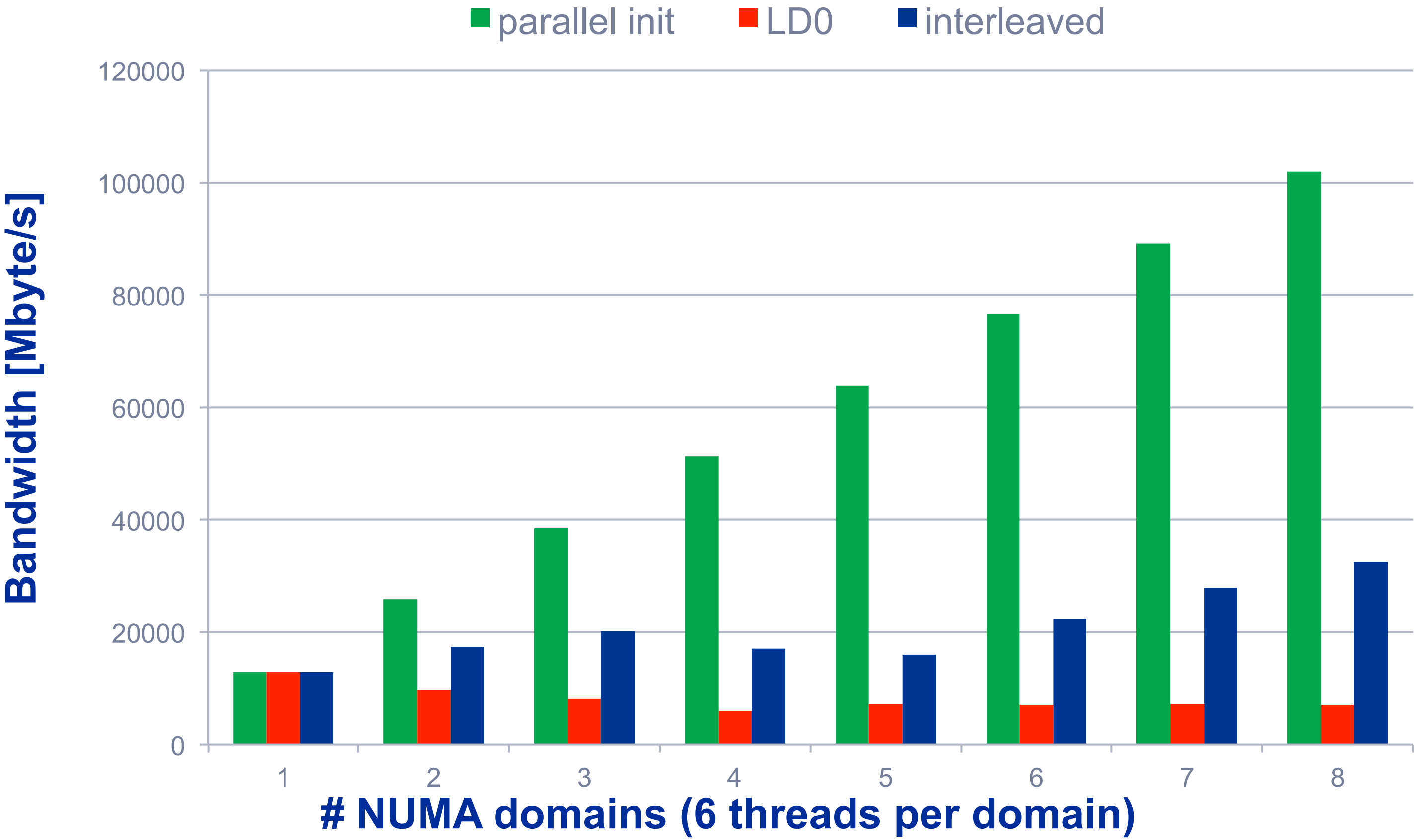
Parallel init: Correct parallel initialization

LD0: Force data into LD0 via `numactl –m 0`

Interleaved: `numactl --interleave <LD range>`



The curse and blessing of interleaved placement: *same on 4-socket (48 core) Magny Cours node*



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

MULTICORE PERFORMANCE TOOLS: PROBING PERFORMANCE BEHAVIOR

likwid-perfctr

likwid-perfctr

Basic approach to performance analysis

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

Possible signatures:

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

Probing performance behavior

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

```
BRANCH: Branch prediction miss rate/ratio
CACHE: Data cache miss rate/ratio
CLOCK: Clock of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
FLOPS_X87: X87 MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
TLB: TLB miss rate/ratio
```

likwid-perfctr

Example usage with preconfigured metric group

```
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP  ./stream.exe
-------------------------------------------------------------
CPU type:        Intel Core Lynnfield processor
CPU clock:       2.93 GHz
-------------------------------------------------------------
Measuring group FLOPS_DP
-------------------------------------------------------------
YOUR PROGRAM OUTPUT
+--------------------------------------+-------------+-------------+-------------+-------------+
|                Event                 |   core 0    |   core 1    |   core 2    |   core 3    |
+--------------------------------------+-------------+-------------+-------------+-------------+
|          INSTR_RETIRED_ANY           | 1.97463e+08 | 2.31001e+08 | 2.30963e+08 | 2.31885e+08 |
|        CPU_CLK_UNHALTED_CORE         | 9.56999e+08 | 9.58401e+08 | 9.58637e+08 | 9.57338e+08 |
|      FP_COMP_OPS_EXE_SSE_FP_PACKED   | 4.00294e+07 | 3.08927e+07 | 3.08866e+07 | 3.08904e+07 |
|      FP_COMP_OPS_EXE_SSE_FP_SCALAR   |     882     |      0      |      0      |      0      |
| FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION |      0      |      0      |      0      |      0      |
| FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION | 4.00303e+07 | 3.08927e+07 | 3.08866e+07 | 3.08904e+07 |
+--------------------------------------+-------------+-------------+-------------+-------------+
+---------------------------+------------+----------+----------+----------+
|          Metric           |   core 0   |  core 1  |  core 2  |  core 3  |
+---------------------------+------------+----------+----------+----------+
|        Runtime [s]        |  0.326242  | 0.32672  | 0.326801 | 0.326358 |
|            CPI            |  4.84647   | 4.14891  | 4.15061  | 4.12849  |
| DP MFlops/s (DP assumed)  |  245.399   | 189.108  | 189.024  | 189.304  |
|      Packed MUOPS/s       |  122.698   | 94.554   | 94.5121  | 94.6519  |
|      Scalar MUOPS/s       | 0.00270351 |    0     |    0     |    0     |
|        SP MUOPS/s         |     0      |    0     |    0     |    0     |
|        DP MUOPS/s         |  122.701   | 94.554   | 94.5121  | 94.6519  |
+---------------------------+------------+----------+----------+----------+
```

Always measured

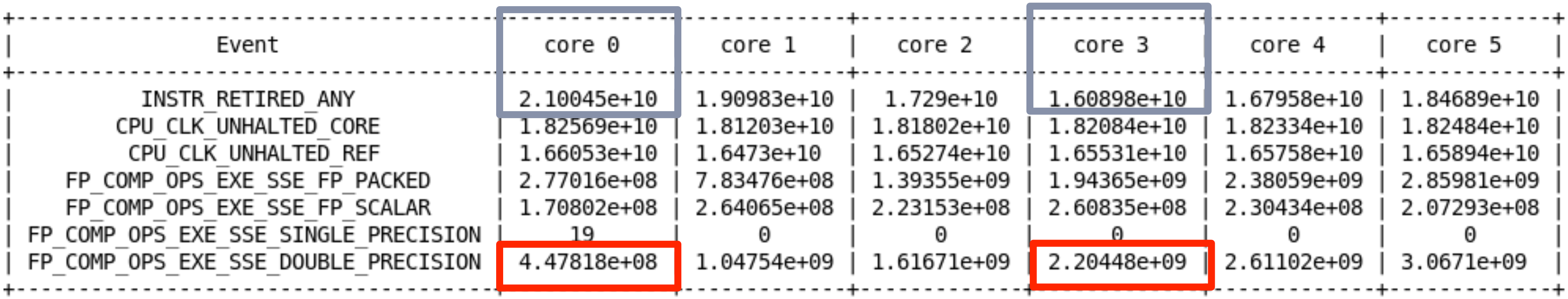
Configured metrics (this group)

Derived metrics

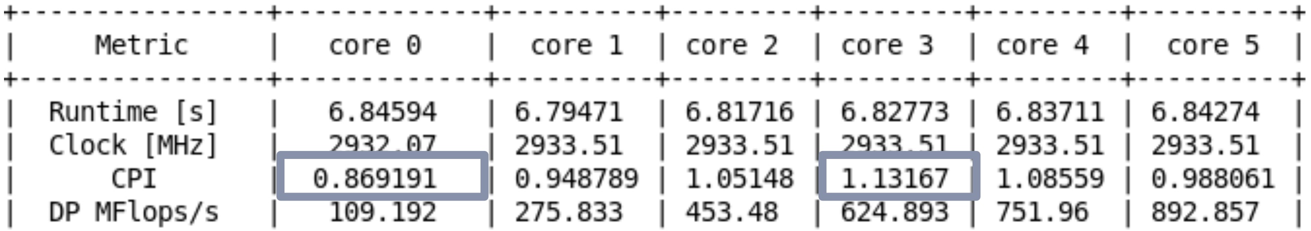
likwid-perfctr

Identify load imbalance…

- `Instructions retired / CPI` **may not be a good indication of useful workload – at least for numerical / FP intensive codes….**
- **Floating Point Operations Executed is often a better indicator**
- **Waiting / “Spinning” in barrier generates a high instruction count**



```
+-----------------------------------------------------------------------------------------------------------------------+
|                Event                  |   core 0    |   core 1    |   core 2    |   core 3    |   core 4    |   core 5    |
+-----------------------------------------------------------------------------------------------------------------------+
|          INSTR_RETIRED_ANY            | 2.10045e+10 | 1.90983e+10 |  1.729e+10  | 1.60898e+10 | 1.67958e+10 | 1.84689e+10 |
|         CPU_CLK_UNHALTED_CORE         | 1.82569e+10 | 1.81203e+10 | 1.81802e+10 | 1.82084e+10 | 1.82334e+10 | 1.82484e+10 |
|          CPU_CLK_UNHALTED_REF         | 1.66053e+10 | 1.6473e+10  | 1.65274e+10 | 1.65531e+10 | 1.65758e+10 | 1.65894e+10 |
|      FP_COMP_OPS_EXE_SSE_FP_PACKED    | 2.77016e+08 | 7.83476e+08 | 1.39355e+09 | 1.94365e+09 | 2.38059e+09 | 2.85981e+09 |
|      FP_COMP_OPS_EXE_SSE_FP_SCALAR    | 1.70802e+08 | 2.64065e+08 | 2.23153e+08 | 2.60835e+08 | 2.30434e+08 | 2.07293e+08 |
| FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION  |     19      |      0      |      0      |      0      |      0      |      0      |
| FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION  | 4.47818e+08 | 1.04754e+09 | 1.61671e+09 | 2.20448e+09 | 2.61102e+09 | 3.0671e+09  |
+-----------------------------------------------------------------------------------------------------------------------+
```



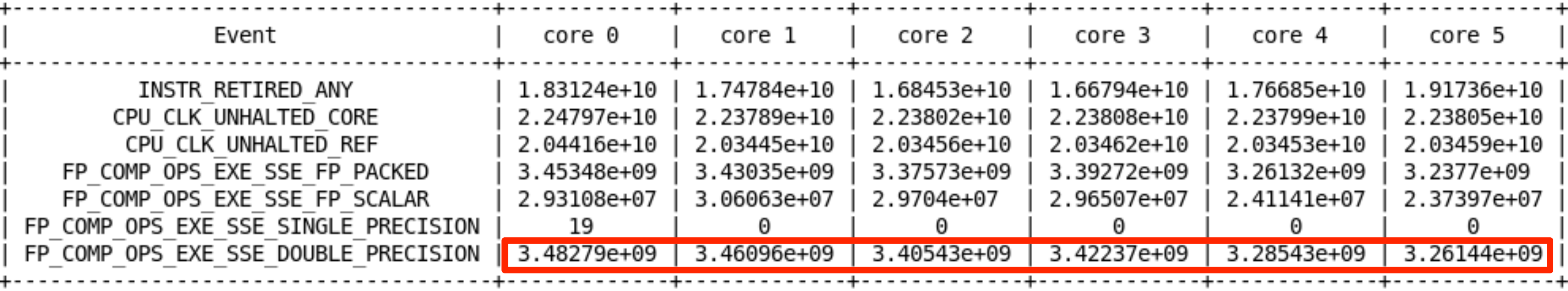
```
+-------------+-----------+----------+----------+----------+----------+----------+
|   Metric    |  core 0   |  core 1  |  core 2  |  core 3  |  core 4  |  core 5  |
+-------------+-----------+----------+----------+----------+----------+----------+
|  Runtime [s] |  6.84594  | 6.79471  | 6.81716  | 6.82773  | 6.83711  | 6.84274  |
|  Clock [MHz] |  2932.07  | 2933.51  | 2933.51  | 2933.51  | 2933.51  | 2933.51  |
|     CPI     | 0.869191  | 0.948789 | 1.05148  | 1.13167  | 1.08559  | 0.988061 |
| DP MFlops/s |  109.192  | 275.833  | 453.48   | 624.893  | 751.96   | 892.857  |
```

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

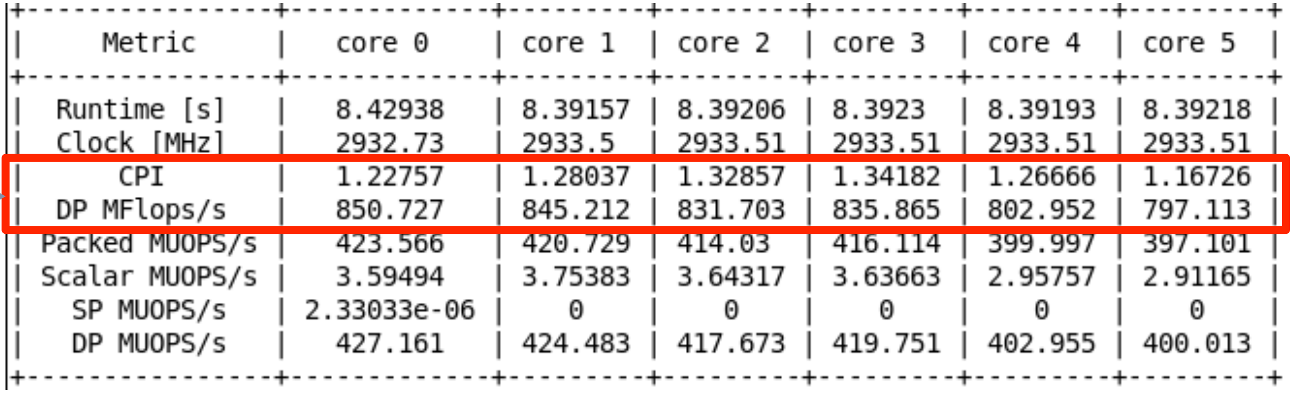
likwid-perfctr

… and load-balanced codes

- `env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out`



```
+--------------------------------------+-----------+-----------+-----------+-----------+-----------+-----------+
|                 Event                |   core 0  |   core 1  |   core 2  |   core 3  |   core 4  |   core 5  |
+--------------------------------------+-----------+-----------+-----------+-----------+-----------+-----------+
|           INSTR_RETIRED_ANY          |1.83124e+10|1.74784e+10|1.68453e+10|1.66794e+10|1.76685e+10|1.91736e+10|
|         CPU_CLK_UNHALTED_CORE        |2.24797e+10|2.23789e+10|2.23802e+10|2.23808e+10|2.23799e+10|2.23805e+10|
|          CPU_CLK_UNHALTED_REF        |2.04416e+10|2.03445e+10|2.03456e+10|2.03462e+10|2.03453e+10|2.03459e+10|
|      FP_COMP_OPS_EXE_SSE_FP_PACKED   |3.45348e+09|3.43035e+09|3.37573e+09|3.39272e+09|3.26132e+09| 3.2377e+09|
|      FP_COMP_OPS_EXE_SSE_FP_SCALAR   |2.93108e+07|3.06063e+07| 2.9704e+07|2.96507e+07|2.41141e+07|2.37397e+07|
| FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION |     19    |     0     |     0     |     0     |     0     |     0     |
| FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION |3.48279e+09|3.46096e+09|3.40543e+09|3.42237e+09|3.28543e+09|3.26144e+09|
+--------------------------------------+-----------+-----------+-----------+-----------+-----------+-----------+
```



```
+-----------------+-------------+---------+---------+---------+---------+---------+
|     Metric      |    core 0   | core 1  | core 2  | core 3  | core 4  | core 5  |
+-----------------+-------------+---------+---------+---------+---------+---------+
|   Runtime [s]   |   8.42938   | 8.39157 | 8.39206 | 8.3923  | 8.39193 | 8.39218 |
|   Clock [MHz]   |   2932.73   | 2933.5  | 2933.51 | 2933.51 | 2933.51 | 2933.51 |
|       CPI       |   1.22757   | 1.28037 | 1.32857 | 1.34182 | 1.26666 | 1.16726 |
|   DP MFlops/s   |   850.727   | 845.212 | 831.703 | 835.865 | 802.952 | 797.113 |
|  Packed MUOPS/s |   423.566   | 420.729 | 414.03  | 416.114 | 399.997 | 397.101 |
|  Scalar MUOPS/s |   3.59494   | 3.75383 | 3.64317 | 3.63663 | 2.95757 | 2.91165 |
|    SP MUOPS/s   | 2.33033e-06 |    0    |    0    |    0    |    0    |    0    |
|    DP MUOPS/s   |   427.161   | 424.483 | 417.673 | 419.751 | 402.955 | 400.013 |
+-----------------+-------------+---------+---------+---------+---------+---------+
```

Higher CPI but better performance

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

Example 1:
Abstraction penalties in C++ code

- C++ codes which suffer from overhead (inlining problems, complex abstractions) need a lot more overall instructions related to the arithmetic instructions
  - Often (but not always) “good” (i.e., low) CPI → “Instruction overhead” pattern
  - Low-ish bandwidth
  - Low # of floating-point instructions vs. other instructions
  - High-level optimizations complex or impossible → “Excess data volume” pattern
- Example: Matrix-matrix multiply with expression template frameworks on a 2.93 GHz Westmere core

	Total retired instructions [10^{11}]	CPI	Memory Bandwidth [MB/s]	MFlops/s
Classic	**12.5**	**0.44**	**5300**	**1250**
Boost uBLAS	**10.1**	**4.6**	**630**	**156**
Eigen3	**2.1**	**0.41**	**371**	**8555**
Blaze/DGEMM	**2.0**	**0.32**	**531**	**11260**

likwid-perfctr

Stethoscope mode

- likwid-perfctr counts events on cores; it has no notion of what kind of code is running (if any)

  This enables to listen on what currently happens without any overhead:

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

- It can be used as cluster/server monitoring tool
- A frequent use is to measure a certain part of a long running parallel application from outside

likwid-perfctr

Marker API

- To measure only parts of an application a marker API is available.
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr application.
- Multiple named regions can be measured
- Results on multiple calls are accumulated
- Inclusive and overlapping Regions are allowed

```
#define LIKWID_PERFMON // comment to disable
#include <likwid.h>

LIKWID_MARKER_INIT;

LIKWID_MARKER_THREADINIT;
LIKWID_MARKER_START(“Compute”);
. . .
LIKWID_MARKER_STOP(“Compute”);

LIKWID_MARKER_START(“postprocess”);
. . .
LIKWID_MARKER_STOP(“postprocess”);

LIKWID_MARKER_CLOSE;
```

likwid-perfctr
Group files

- Groups are architecture-specific
- They are defined in simple text files
- Code is generated on recompile of likwid
- likwid-perfctr -a outputs list of groups
- For every group an extensive documentation is available

```
SHORT PSTI
EVENTSET
FIXC0 INSTR_RETIRED_ANY
FIXC1 CPU_CLK_UNHALTED_CORE
FIXC2 CPU_CLK_UNHALTED_REF
PMC0  FP_COMP_OPS_EXE_SSE_FP_PACKED
PMC1  FP_COMP_OPS_EXE_SSE_FP_SCALAR
PMC2  FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION
PMC3  FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION
UPMC0  UNC_QMC_NORMAL_READS_ANY
UPMC1  UNC_QMC_WRITES_FULL_ANY
UPMC2 UNC_QHL_REQUESTS_REMOTE_READS
UPMC3 UNC_QHL_REQUESTS_LOCAL_READS
METRICS
Runtime [s] FIXC1*inverseClock
CPI  FIXC1/FIXC0
Clock [MHz]  1.E-06*(FIXC1/FIXC2)/inverseClock
DP MFlops/s (DP assumed) 1.0E-06*(PMC0*2.0+PMC1)/time
Packed MUOPS/s   1.0E-06*PMC0/time
Scalar MUOPS/s 1.0E-06*PMC1/time
SP MUOPS/s 1.0E-06*PMC2/time
DP MUOPS/s 1.0E-06*PMC3/time
Memory bandwidth [MBytes/s] 1.0E-06*(UPMC0+UPMC1)*64/time;
Remote Read BW [MBytes/s] 1.0E-06*(UPMC2)*64/time;
LONG
Formula:
DP MFlops/s =  (FP_COMP_OPS_EXE_SSE_FP_PACKED*2 +  FP_COMP_OPS_EXE_SSE_FP_SCALAR)/ runtime.
```

Measuring energy consumption

likwid-powermeter and likwid-perfctr -g ENERGY

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

```
-------------------------------------------------------------
CPU name:       Intel Core SandyBridge processor
CPU clock:      3.49 GHz
-------------------------------------------------------------
Base clock:     3500.00 MHz
Minimal clock:  1600.00 MHz
Turbo Boost Steps:
C1 3900.00 MHz
C2 3800.00 MHz
C3 3700.00 MHz
C4 3600.00 MHz
-------------------------------------------------------------
Thermal Spec Power: 95 Watts
Minimum  Power: 20 Watts
Maximum  Power: 95 Watts
Maximum  Time Window: 0.15625 micro sec
-------------------------------------------------------------
```

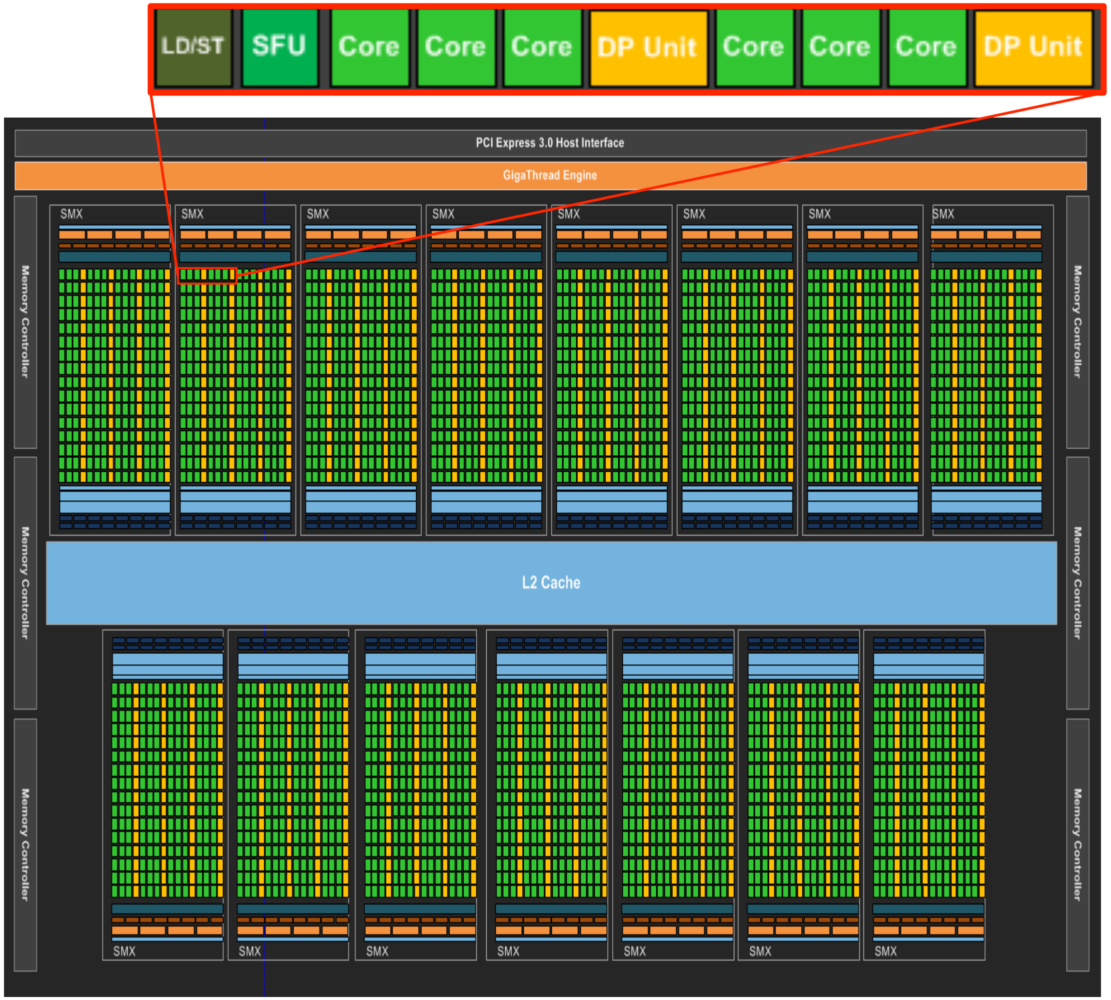
INTERLUDE:
A GLANCE AT CURRENT ACCELERATOR TECHNOLOGY

NVIDIA Kepler GK110 Block Diagram

Architecture

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

- 3:1 SP:DP performance



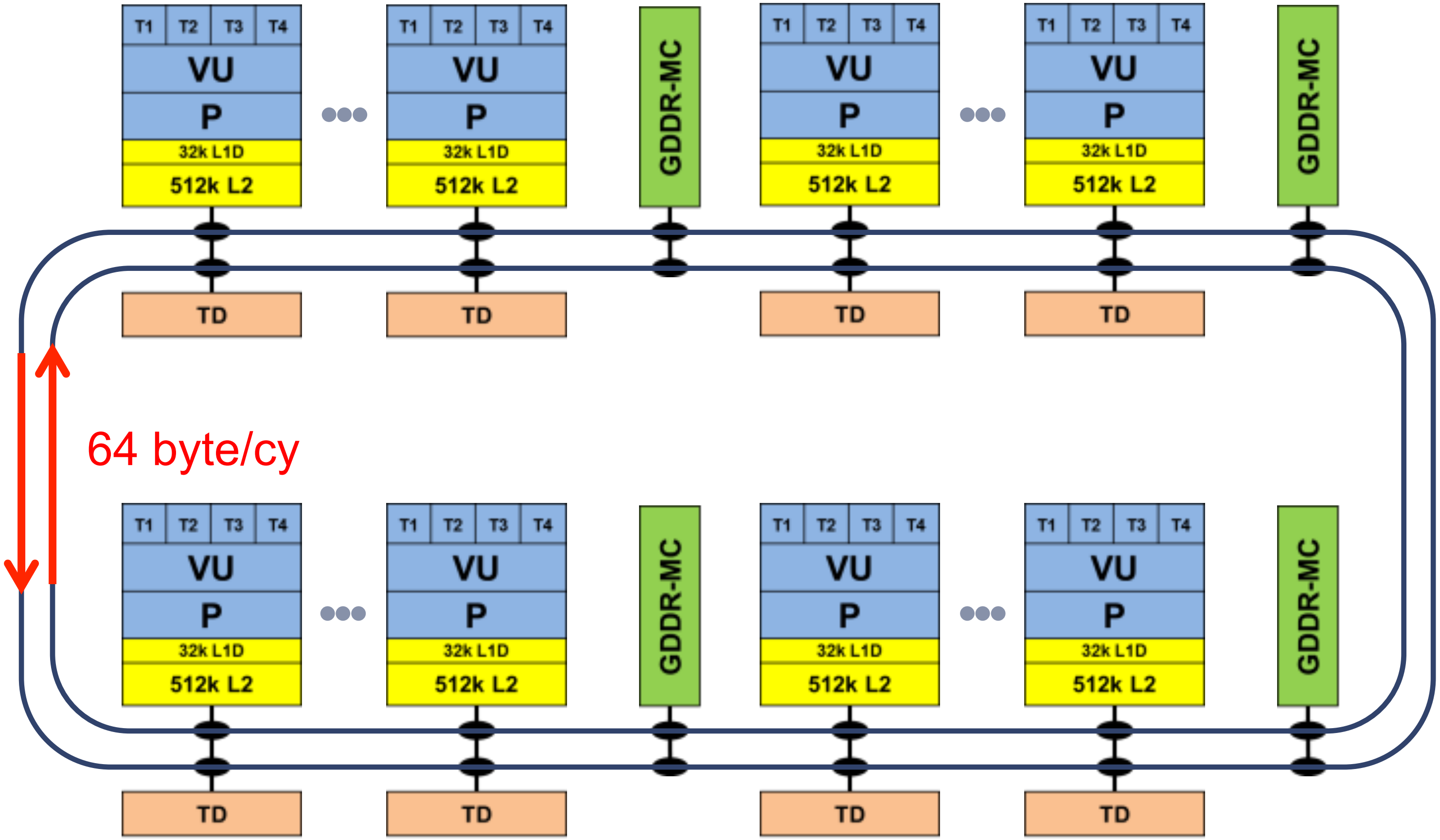
© NVIDIA Corp. Used with permission.

Intel Xeon Phi block diagram

Architecture

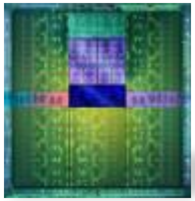
- 3B Transistors
- 60+ cores
- 512 bit SIMD
- ≈ 1 TFLOP DP peak
- 0.5 MB L2/core
- GDDR5

- 2:1 SP:DP performance



Comparing accelerators

- Intel Xeon Phi
  - **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **480/960 SIMD DP/SP tracks**
  - Clock Speed: ~1000 MHz
  - Transistor count: ~3 B (22nm)
  - Power consumption: ~250 W
  - Peak Performance (DP): ~ 1 TF/s
  - Memory BW: ~250 GB/s (GDDR5)
  - Threads to execute: 60-240+
  - Programming: Fortran/C/C++ +OpenMP + SIMD



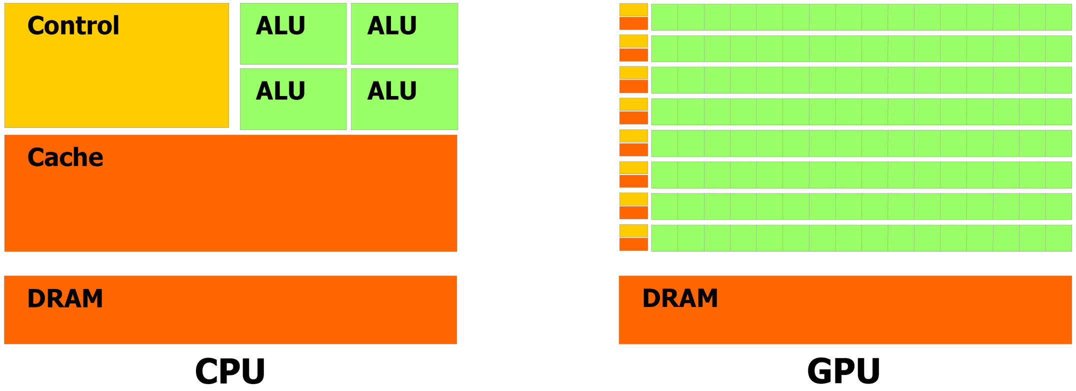
- NVIDIA Kepler K20
  - 15 SMX units each with 192 "cores" → **960/2880 DP/SP "cores"**
  - Clock Speed: ~700 MHz
  - Transistor count: 7.1 B (28nm)
  - Power consumption: ~250 W
  - Peak Performance (DP): ~ 1.3 TF/s
  - Memory BW: ~ 250 GB/s (GDDR5)
  - Threads to execute: 10,000+
  - Programming: CUDA, OpenCL, (OpenACC)

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 E5-2660v2 node (“Ivy Bridge”)	NVIDIA K20x (“Kepler”)
Cores@Clock	4 @ 3.3 GHz	2 x 10 @ 2.2 GHz	2880 @ 0.7 GHz
Performance⁺/core	52.8 GFlop/s	35.2 GFlop/s	1.4 GFlop/s
Threads@STREAM	<4	<20	>8000?
Total performance⁺	210 GFlop/s	704 GFlop/s	4,000 GFlop/s
Stream BW	18 GB/s	2 x 42 GB/s	168 GB/s (ECC=1)
Transistors / TDP	1 Billion* / 95 W	2 x (2.86 Billion/95 W)	**7.1 Billion/250W**

⁺ *Single Precision*

CASE STUDY: HPCCG

Performance analysis on:

- Intel IvyBridge-EP@2.2GHz
- Intel Xeon Phi@1.05GHz

Introduction to HPCCG (Mantevo suite)

```
for(int k=1; k<max_iter && normr > tolerance; k++ )
{
    oldrtrans = rtrans;
    ddot (nrow, r, r, &rtrans, t4);
    double beta = rtrans/oldrtrans;
    waxpby (nrow, 1.0, r, beta, p, p);
    normr = sqrt(rtrans);
    HPC_sparsemv(A, p, Ap);
    double alpha = 0.0;
    ddot(nrow, p, Ap, &alpha, t4);
    alpha = rtrans/alpha;
    waxpby(nrow, 1.0, r, -alpha, Ap, r);
    waxpby(nrow, 1.0, x, alpha, p, x);
    niters = k;
}
```

Components of HPCCG 1

ddot:

```
#pragma omp for reduction (+:result)
for (int i=0; i<n; i++) {
      result += x[i] * y[i];
}
```

2 Flops
2 * 8b L = 16b
2.2GHz/2c * 16 Flops =
17.6 GFlops/s or
140GB/s L1 or 46GB/s L2

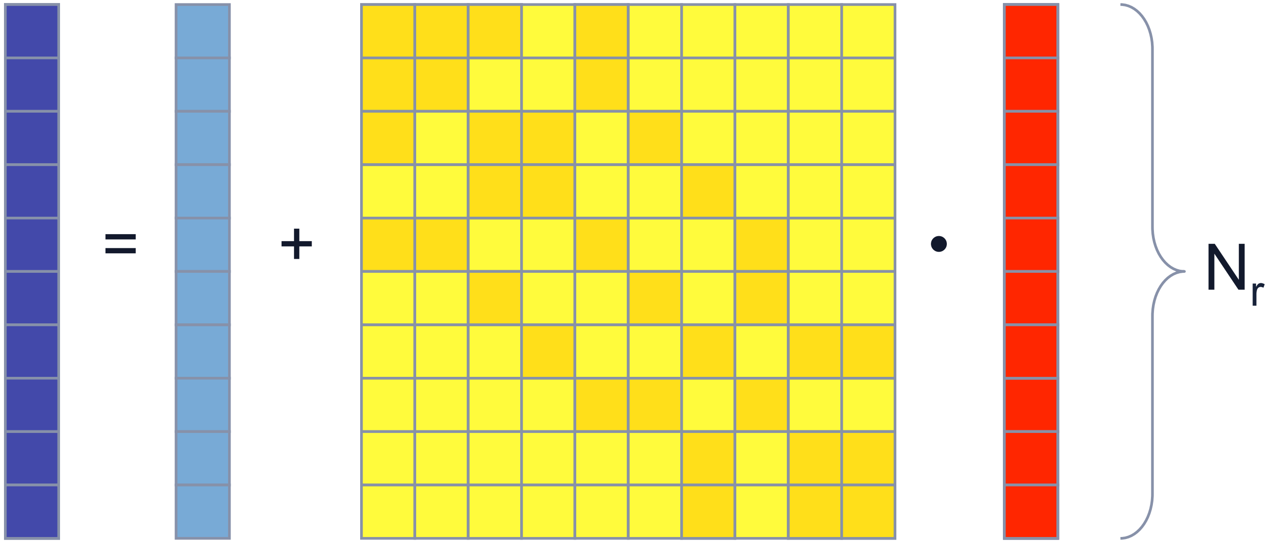
waxpby:

```
#pragma omp for
for (int i=0; i<n; i++) {
    w[i] = alpha * x[i] + beta * y[i];
}
```

3 Flops
2 * 8b L + 1 * 8b S = 24b
2.2GHz/4c * 24flops =
13.2 GFlops/s or
106GB/s L1 or 47GB/s L2

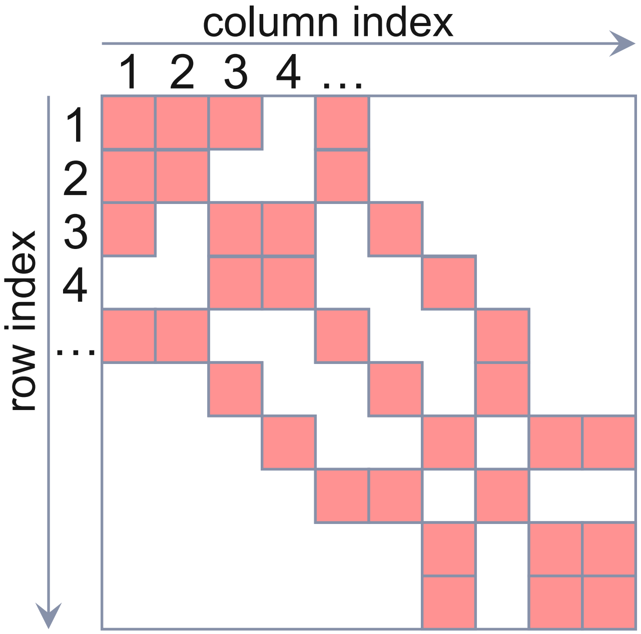
Sparse matrix-vector multiply (spMVM)

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

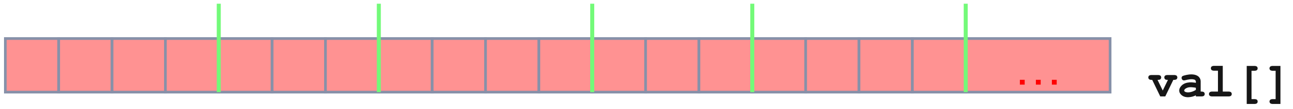


General case: some indirect addressing required!

CRS matrix storage scheme



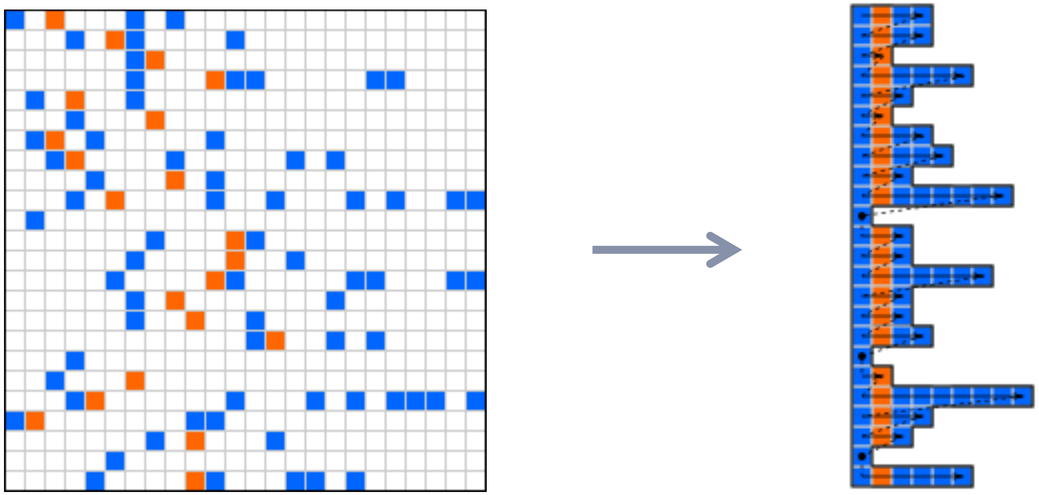
- `val[]` stores all the nonzeros (length N_{nz})
- `col_idx[]` stores the column index of each nonzero (length N_{nz})
- `row_ptr[]` stores the starting index of each new row in `val[]` (length: N_r)



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

1	5	8	12	15	19	…	`row_ptr[]`

CRS (Compressed Row Storage) – data format



- **Format creation**
  1. Store values and column indices of all non-zero elements **row-wise**
  2. Store starting indices of each column (**`rpt`**)

- **Data arrays**

```
double val[]
unsigned int col[]
unsigned int rpt[]
```

Components of HPCCG 2

```
#pragma omp for
for (int i=0; i< nrow; i++) {
   double sum = 0.0;
   double* cur_vals = vals_in_row[i];
   int*    cur_inds = inds_in_row[i];
   int     cur_nnz =  nnz_in_row[i];

   for (int j=0; j< cur_nnz; j++) {
      sum += cur_vals[j]*x[cur_inds[j]];
   }
   y[i] = sum;
}
```

2 Flops

1 * 4b L + 2 * 8b L = 20b

2.2GHz/2c * 16 Flops =

17.6 GFlops/s or

140GB/s L1 or 46GB/s L2

First Step: Runtime Profile (300^3)

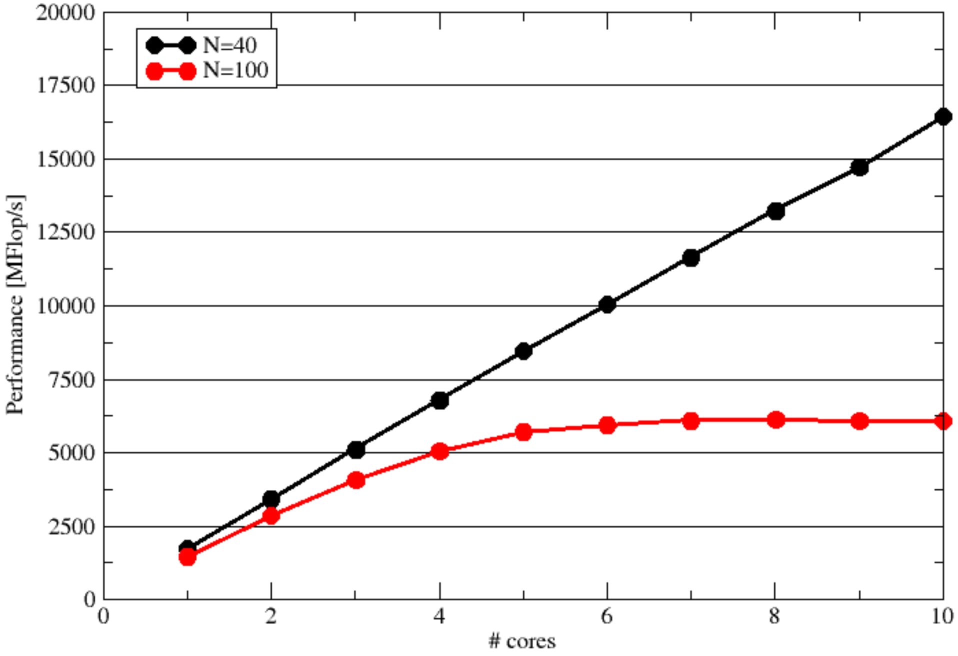
Intel IvyBridge-EP (2.2GHz, 10 cores/chip)

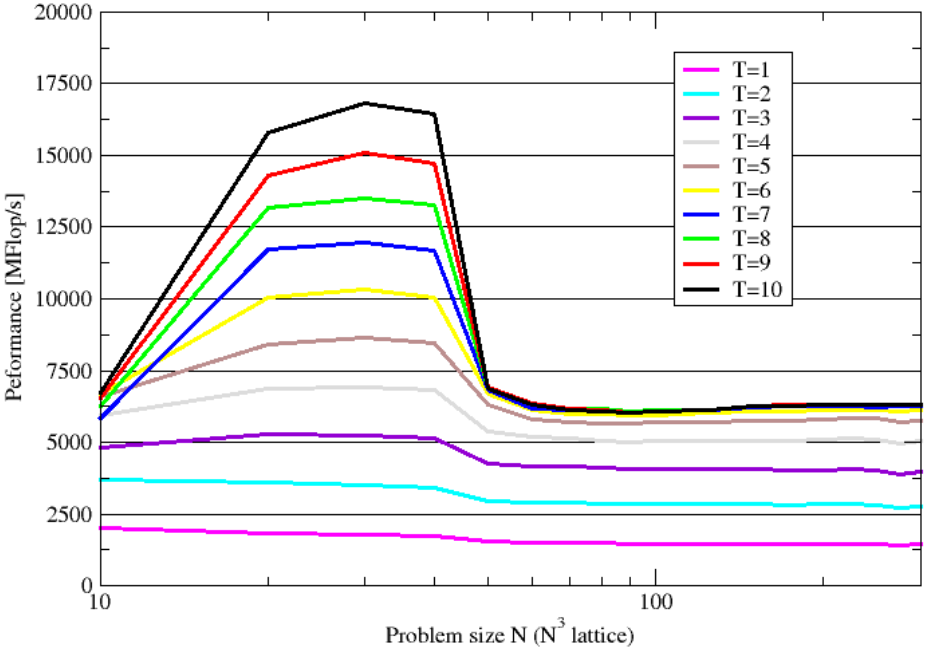
Routine	Serial	Socket
ddot	5%	5%
waxby	12%	16%
spmv	83%	79%

Intel Xeon Phi (1.05GHz, 60 cores/chip)

Routine	Chip
ddot	3%
waxby	8%
spmv	89%

Scaling behavior inside socket (IvyBridge-EP)





HPM measurement with LIKWID instrumentation on socket level

Routine	Time [s]	Memory Bandwidth [MB/s]	Data Volume [GB]
waxby 1	2,33	40464	93
waxby 2	2,37	39919	94
waxby 3	2,4	40545	96
ddot 1	0,72	46886	34
ddot 2	1,4	46444	64
spmv	33,84	45964	1555

Scaling to full node (180^3)

Performance [GFlops/s]

Routine	Socket	Node
ddot	6726	14547
waxby	3642	6123
spmv	6374	**6320**
Total	5973	6531

Memory Bandwidth measured [GB/s]

Routine	Socket 1	Socket 2	Total
ddot	44020	47342	91362
waxby	39795	28424	68219
spmv	43109	**2863**	45972

Optimization: Apply correct data placement

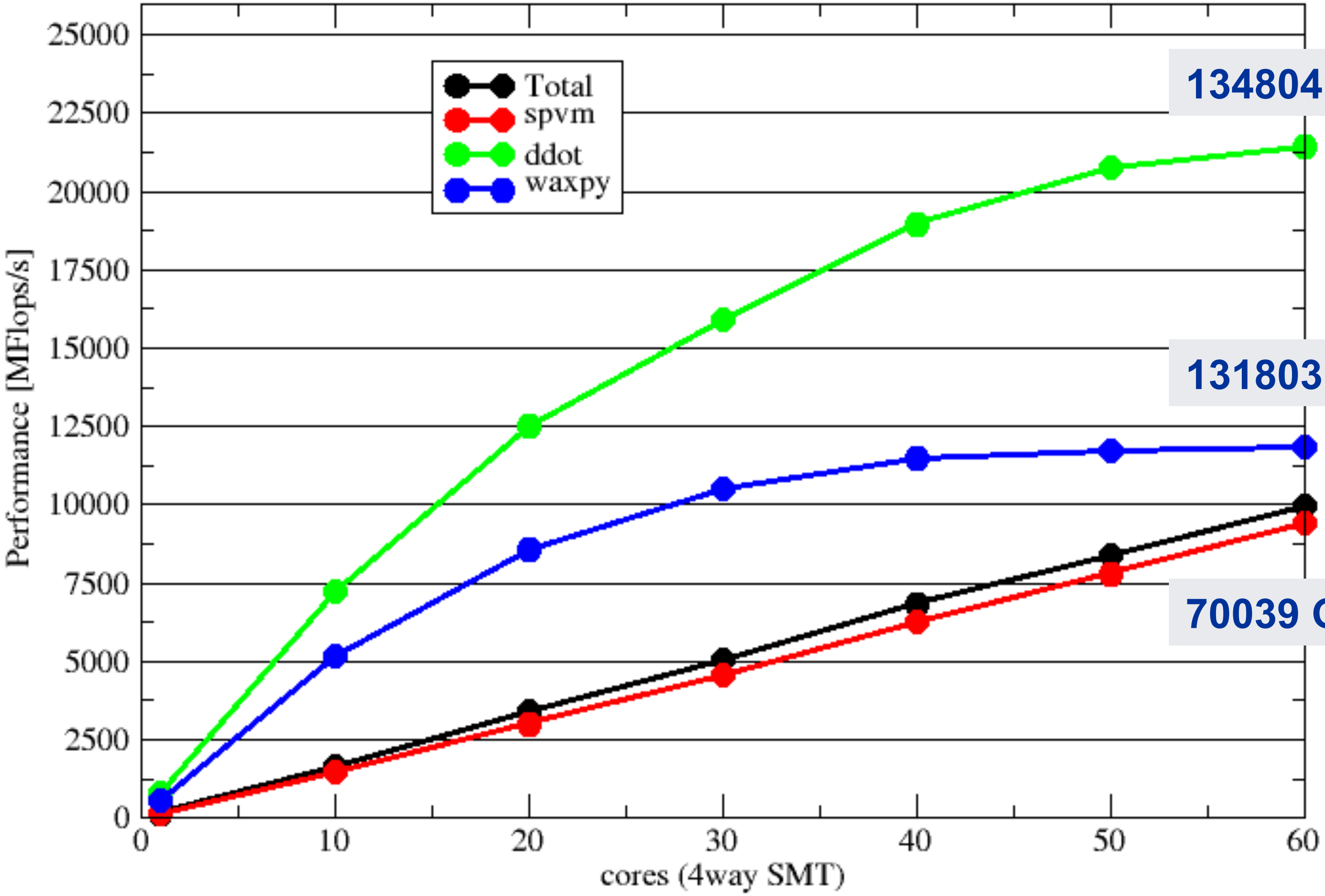
Matrix data was not placed. **Solution:** Add first touch initialization.

```
#pragma omp parallel for
  for (int i=0; i< local_nrow; i++){
      for (int j=0; j< 27; j++) {
          curvalptr[i*27 + j] = 0.0;
          curindptr[i*27 + j] = 0;
      }
}
```

Node performance: spmv 11692, total 10912

Routine	Socket 1	Socket 2	Total
ddot	46406	48193	94599
waxby	37113	24904	62017
spmv	45822	40935	86757

Scaling behavior Intel Xeon Phi



134804 GB/s

131803 GB/s

70039 GB/s

CASE STUDY: C++ SIMULATION CODE