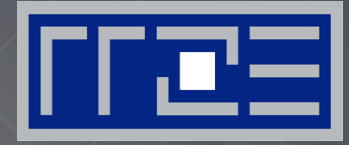


**ERLANGEN REGIONAL  
COMPUTING CENTER**



# **Node-Level Performance Engineering for Multicore Systems**

J. Treibig

PPoPP 2015, 6.2.2015

# The Rules™

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

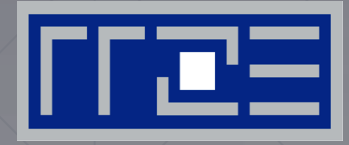
Without performance modeling,  
optimizing code is like stumbling in the dark

# Schedule

Time	Topic
8:30 – 10:00	Intro / Single-Core Performance
10:00 – 10:30	Coffee break
10:30 – 12:00	Node Performance / Performance Tools
12:00 – 14:00	Lunch
14:00 – 15:30	Performance Engineering Process
15:30 – 16:00	Coffee break
16:00 – 17:30	Performance Modeling / Case Studies



# WARMUP: PERFORMANCE QUIZ



# Quiz

- What is a “write-allocate” (a.k.a. read for ownership)?

A: Many cache architectures allocate a CL on a store miss.

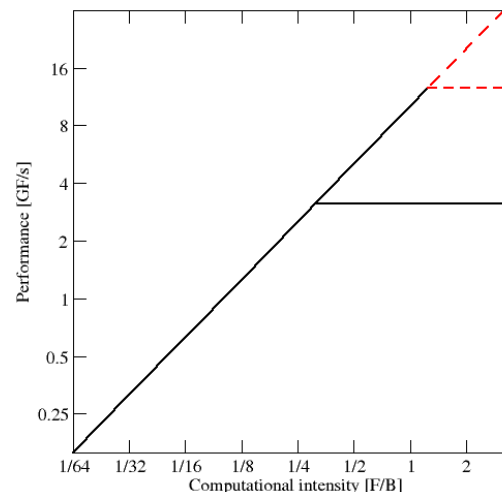
- What is Amdahl's Law?

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

- What is the Roofline Model?

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

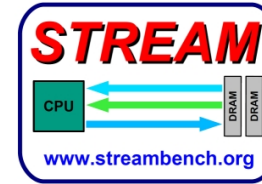
<sup>2</sup> S. Williams: [Auto-tuning Performance on Multicore Computers](#). UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)



# Quiz cont.

- How many cycles does a double-precision ADD/MULT/DIV take?

A: Intel IvyBridge, ADD 3 cycles, MULT 5 cycles, DIV 21 cycles



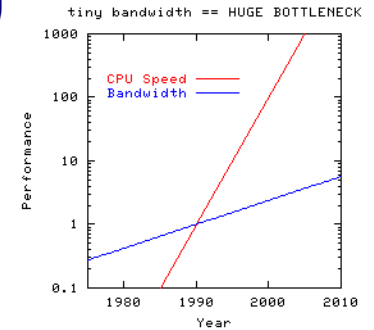
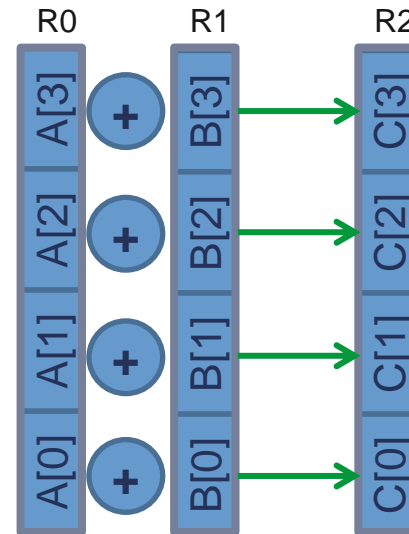
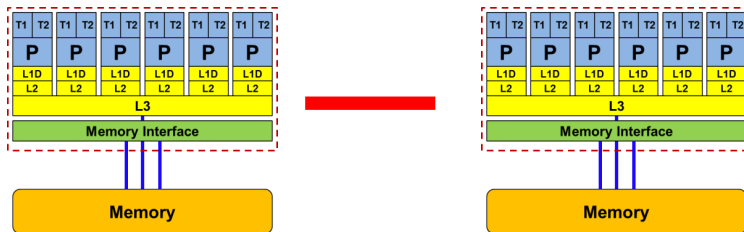
- Do you know the STREAM benchmarks?

A: Defacto standard HPC benchmark for (memory) bandwidth.

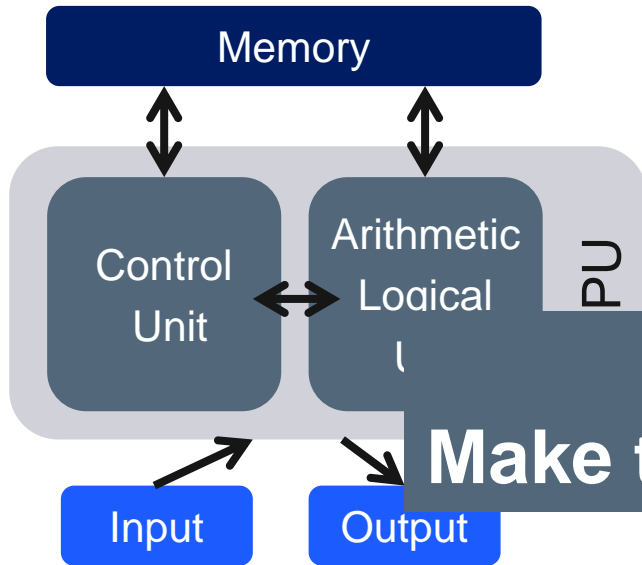
- What is SIMD vectorization?

A: Single instruction multiple data. Data parallel execution units.

- What is ccNUMA?



# 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**

# Excursion in memory bandwidth

*Some thoughts on efficiency ...*

Common lore: *Efficiency is the fraction of peak performance you reach!*

**Example: STREAM triad ( $A(:)=B(:)+C(:)*d$ ) with data not fitting into cache.**

**Intel Xeon X5482 (Harperstown 3.2 GHz): 553 Mflops/s (8 cores)**

**Efficiency 0.54% of peak**

**Intel Xeon E5-2680 (SandyBridge EP 2.7 GHz) 4357 Mflops/s (16 cores)**

**Efficiency 1.2% of peak**

**What can we do about it?**

**Nothing!**



# Excursion in memory bandwidth

*A better way to think about efficiency*

Reality: This code is bound by main memory bandwidth.

HPT 6.6 GB/s (8.8 GB/s with WA)



SNB 52.3 GB/s (69.6 GB/s with WA)

**Efficiency increase: None !**  
**Architecture improvement:**  
**8x**

In both cases this is near 100% of achievable memory bandwidth.

**To think about efficiency you should focus on the utilization of the relevant resource!**

# Hardware-Software Co-Design?

## From algorithm to execution

Notions of work:

- Application Work
  - Flops
  - LUPS
  - VUPS
- Processor Work
  - Instructions
  - Data Volume

Algorithm



Programming language



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.**

# Approaches to performance optimization

**Trial and error**

**Blind data driven**

**Highly skilled experts**

**Automated expert  
tools**

Highly complex  
Problem centric  
Tool centric

# Focus on resource utilization

## 1. Instruction execution

Primary resource of the processor.

## 2. Data transfer bandwidth

Data transfers as a consequence of instruction execution.

What is the **limiting resource**?

Do you fully **utilize** available **resources**?

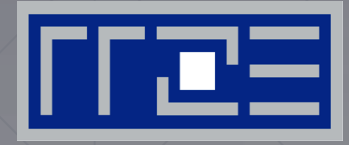
# What needs to be done on one slide

- Reduce computational 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

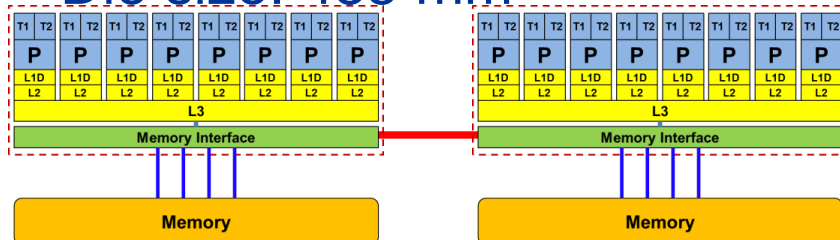
- Single Instruction Multiple Data (**SIMD**)

Lanes      Register width  
                 Packed      Scalar

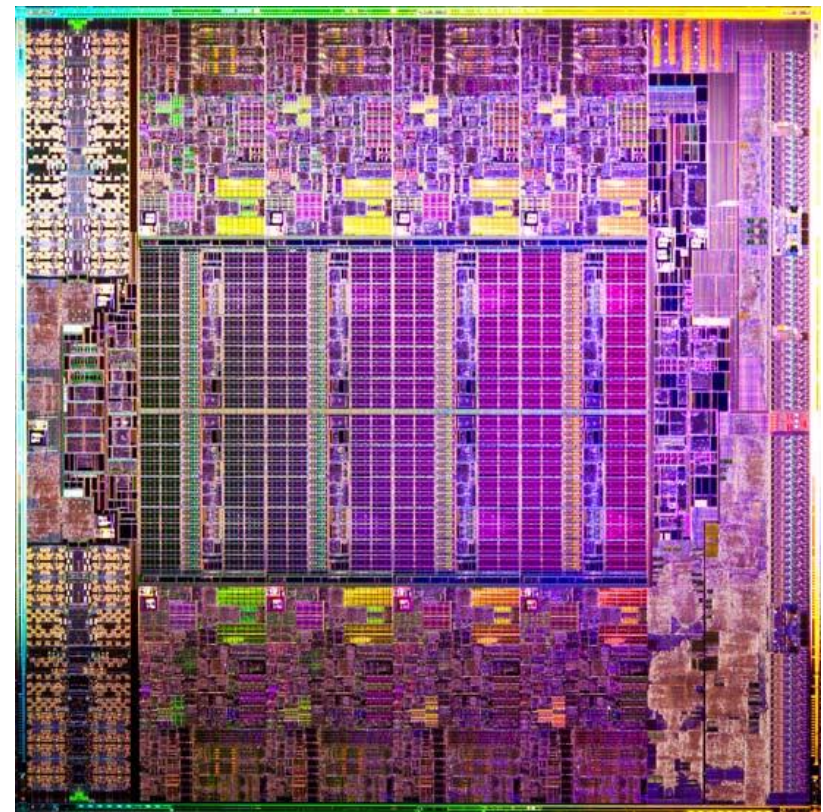
- Simultaneous Multithreading (**SMT**)

# Multi-Core: Intel Xeon 2600 (2012)

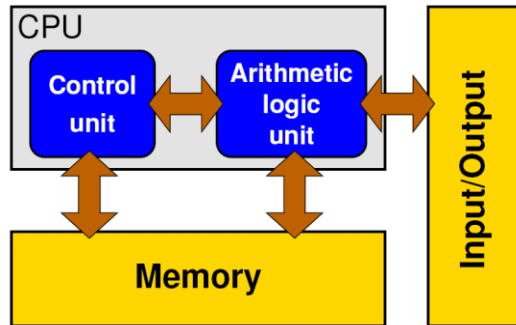
- Xeon 2600 “Sandy Bridge EP”:  
8 cores running at 2.7 GHz (max 3.2 GHz)
- Simultaneous Multithreading  
→ reports as 16-way chip
- **2.3 Billion** Transistors / 32 nm
- Die size: 435 mm<sup>2</sup>



2-socket server

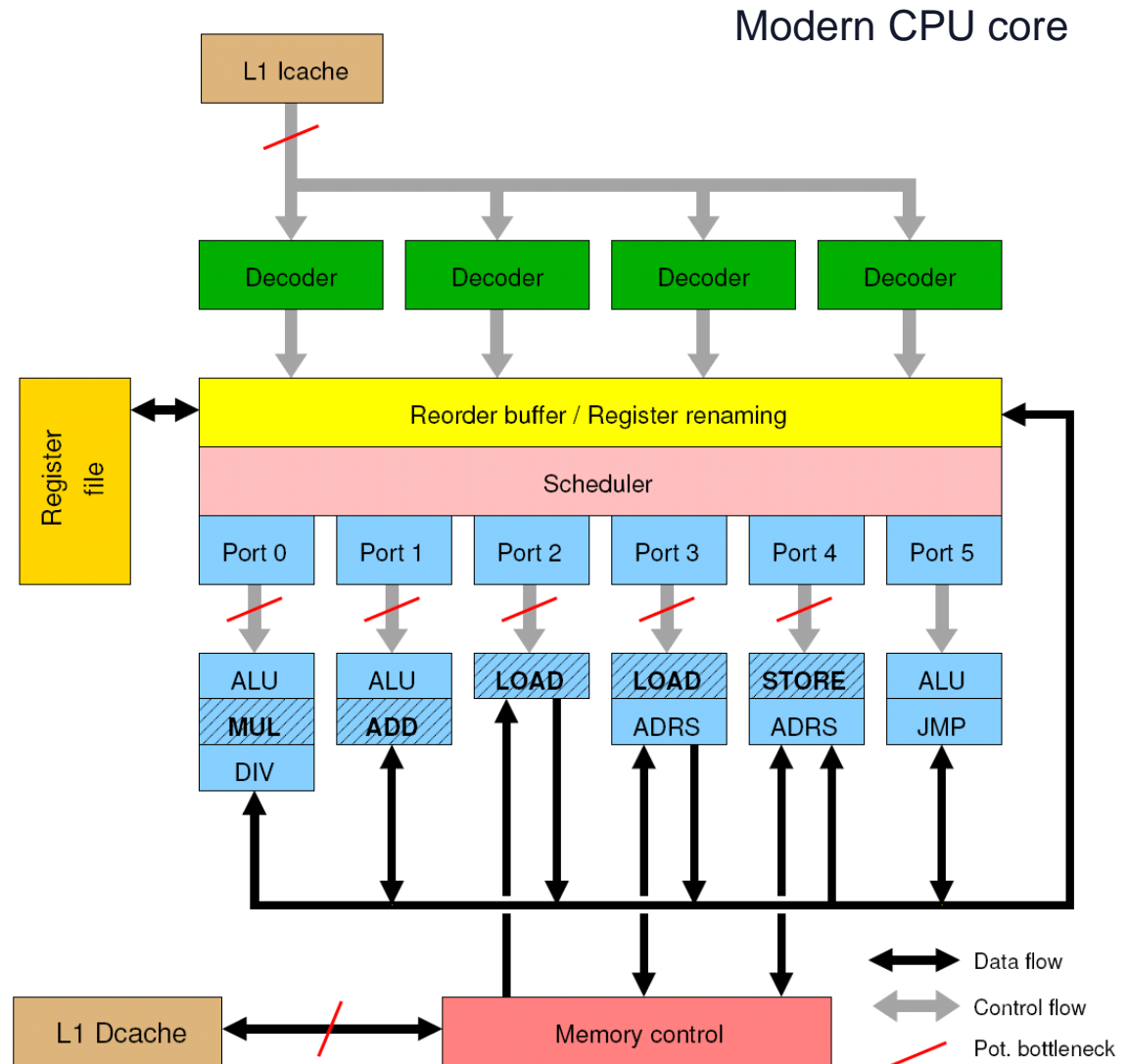


# General-purpose cache based microprocessor core



Stored-program computer

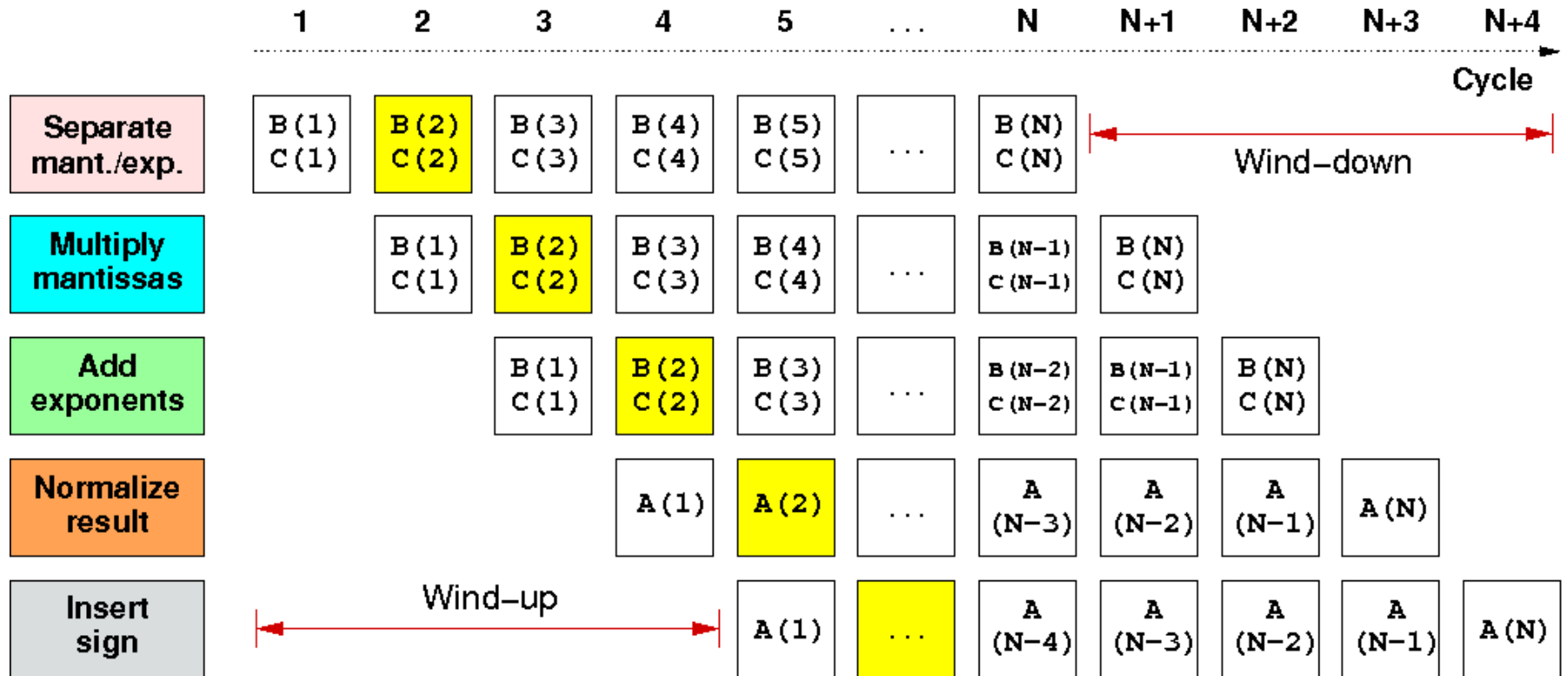
- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks



# Pipelining of arithmetic/functional units

- **Idea:**
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)
- **Allows for shorter cycle times** (simpler logic circuits), e.g.:
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultaneously
  - one result at each cycle after the pipeline is full
- **Drawback:**
  - Pipeline must be filled - startup times (#Instructions >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order
- Pipelining is **widely used** in modern computer architectures

# 5-stage Multiplication-Pipeline:

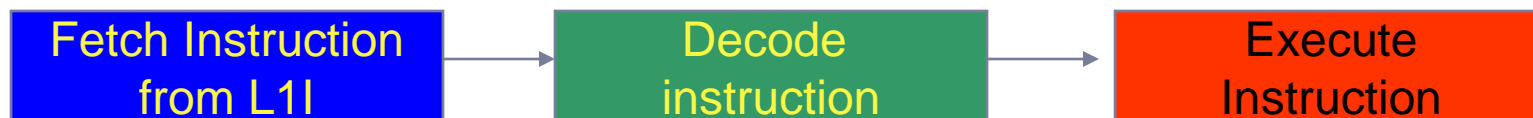
$$A(i) = B(i) * C(i) ; i = 1, \dots, N$$


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

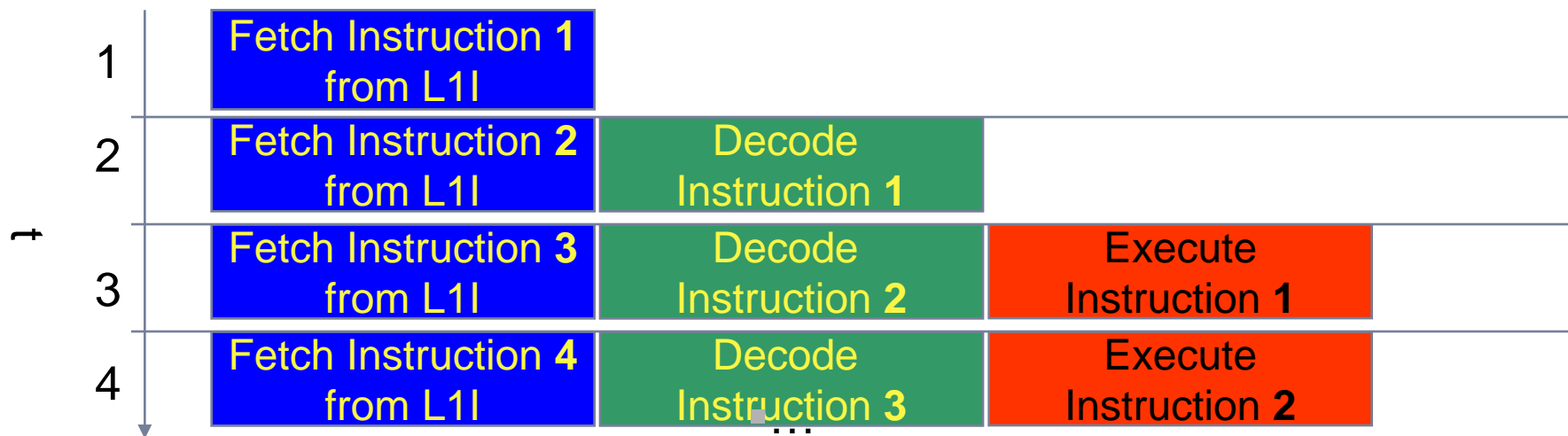
Wind-up/-down phases: Empty pipeline stages

# Pipelining: The Instruction pipeline

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



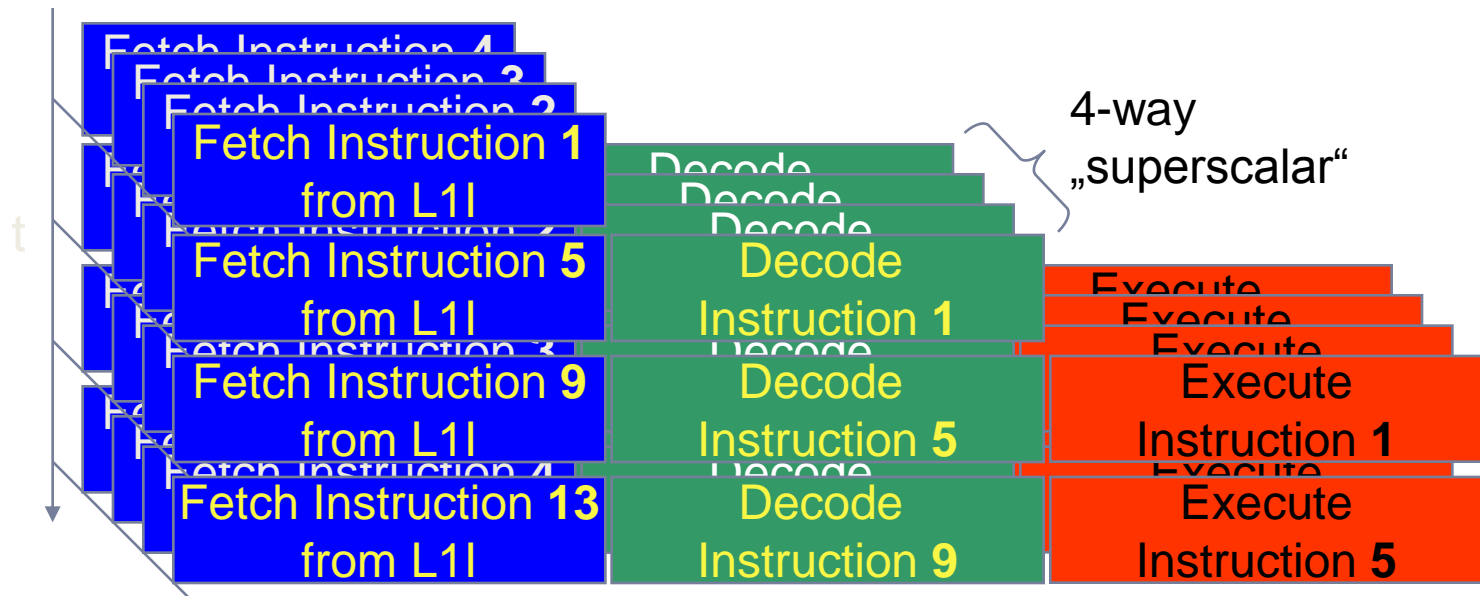
Hardware Pipelining on processor (all units can run concurrently):



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

# Superscalar Processors – Instruction Level Parallelism

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

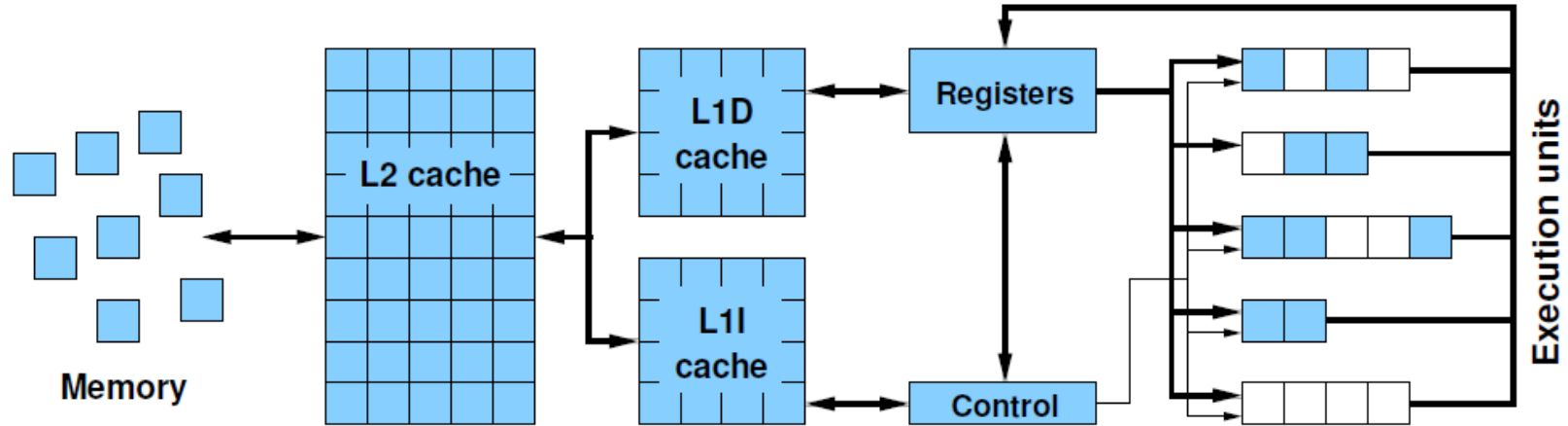


- Issuing  $m$  concurrent instructions per cycle:  $m$ -way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycles

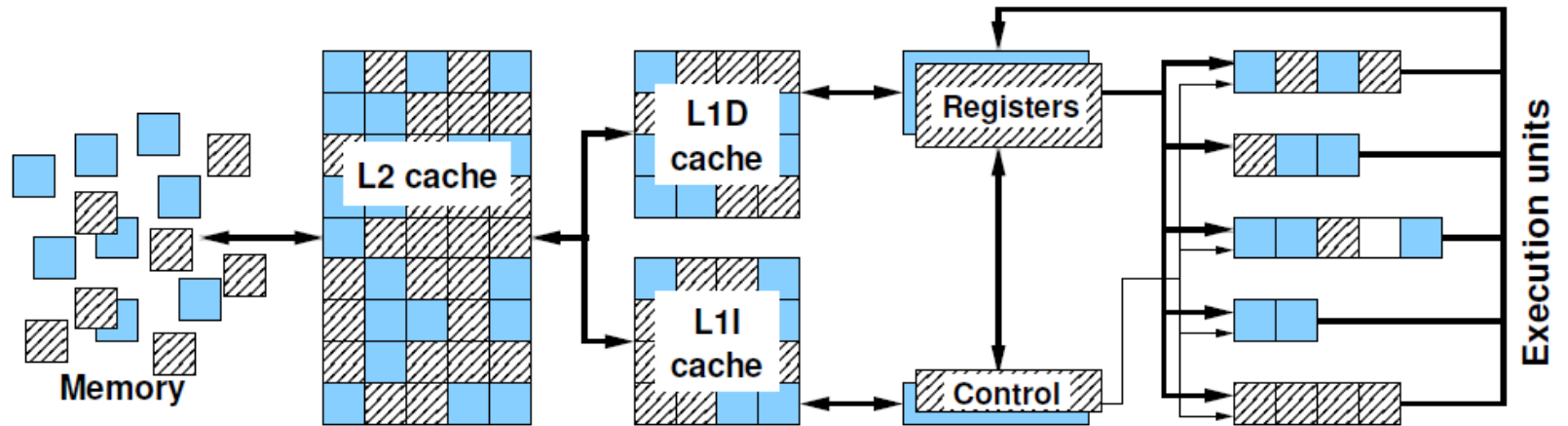


# Core details: Simultaneous multi-threading (SMT)

Standard core



2-way SMT



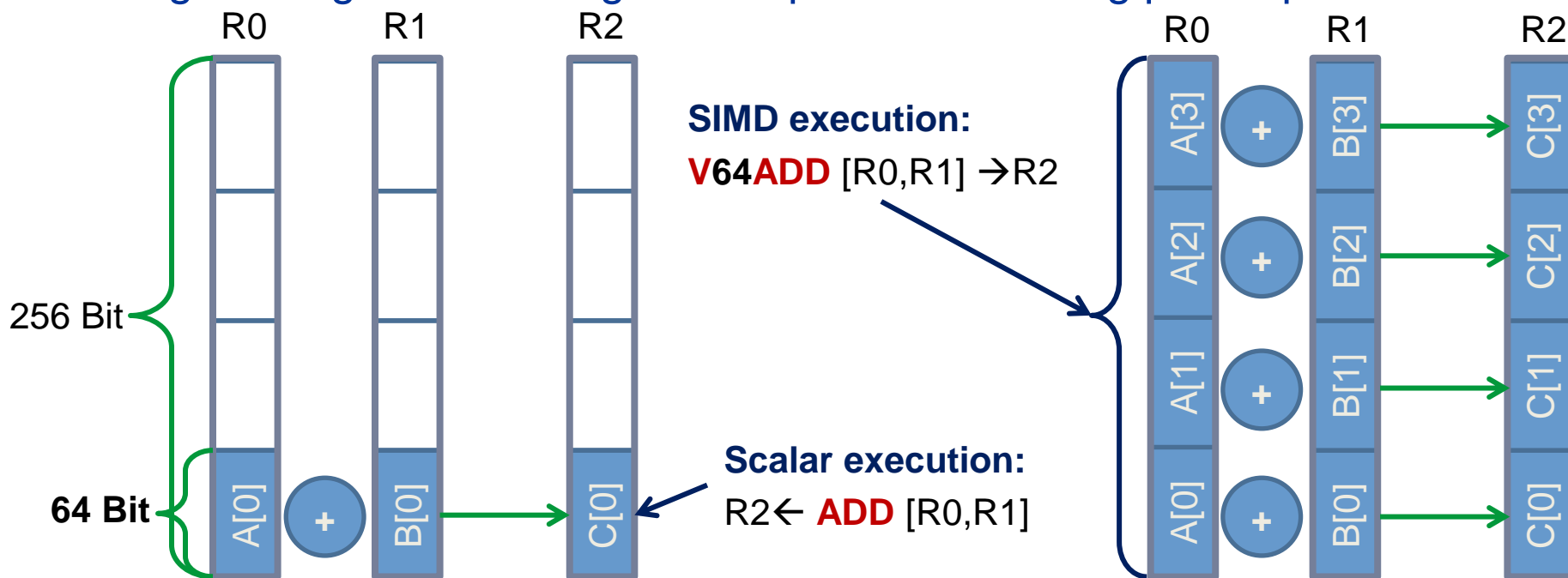


# Core details: SIMD processing

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 processing – Basics

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

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

“Loop unrolling”

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

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

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

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

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

# SIMD processing – Basics

No SIMD vectorization for loops with data dependencies:

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

“**Pointer aliasing**” may prevent SIMDfication

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

C/C++ allows that  $A \rightarrow \&C[-1]$  and  $B \rightarrow \&C[-2]$

$\rightarrow C[i] = C[i-1] + C[i-2]$ : **dependency**  $\rightarrow$  **No SIMD**

If “**pointer aliasing**” is not used, tell it to the compiler:

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

**restrict** keyword (C only!):

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

# Why and how?

## Why check the assembly code?

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

- Get the assembler code (Intel compiler):

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

- Disassemble Executable:

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

## The x86 ISA is documented in:

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

# Basics of the x86-64 ISA

- Instructions have 0 to 2 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode:  $\text{BASE} + \text{INDEX} * \text{SCALE} + \text{DISPLACEMENT}$
- C:  $\text{A}[\text{i}]$  equivalent to  $*(\text{A}+\text{i})$  (a pointer has a type:  $\text{A}+\text{i}*8$ )

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

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

```
401b9f: 0f 29 5c c7 30
401ba4: 48 83 c0 08
401ba8: 78 a6
```

```
movaps %xmm3,0x30(%rdi,%rax,8)
add    $0x8,%rax
js     401b50 <triad_asm+0x4b>
```

# Basics of the x86-64 ISA

16 general Purpose Registers (**64bit**):

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

alias with eight 32 bit register set:

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

Floating Point **SIMD** Registers:

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

`ymm0-ymm15` AVX (256bit)

SIMD instructions are distinguished by:

**AVX (VEX) prefix:**

`v`

**Operation:**

`mul, add, mov`

**Modifier:**

nontemporal (`nt`), unaligned (`u`), aligned (`a`), high (`h`)

**Width:**

scalar (`s`), packed (`p`)

**Data type:**

single (`s`), double (`d`)

# Case Study: Simplest code for the summation of the elements of a vector (single precision)

```
float sum = 0.0;
```

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

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

AT&T syntax:

```
addss 0(%rdx,%rax,4),%xmm0
```

Instruction code:

```
401d08:  f3 0f 58 04 82
```

```
401d0d:  48 83 c0 01
```

```
401d11:  39 c7
```

```
401d13:  77 f3
```

```
addss  xmm0, [rdx + rax * 4]
```

```
add    rax, 1
```

```
cmp    edi, eax
```

```
ja     401d08
```

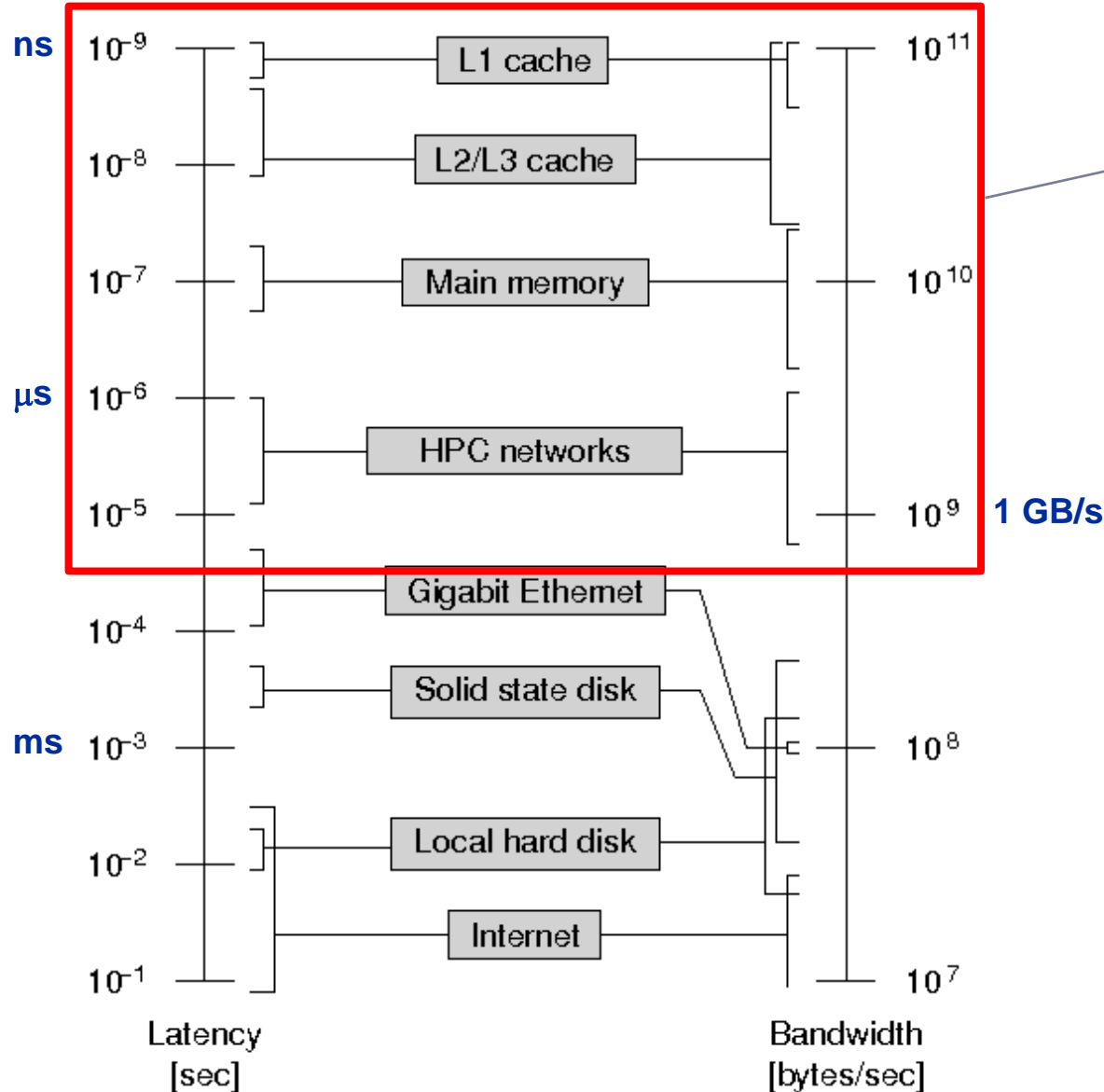
Instruction  
address

Opcodes

(final sum  
across xmm0  
omitted)

Assembly  
code

# Latency and bandwidth in modern computer environments



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



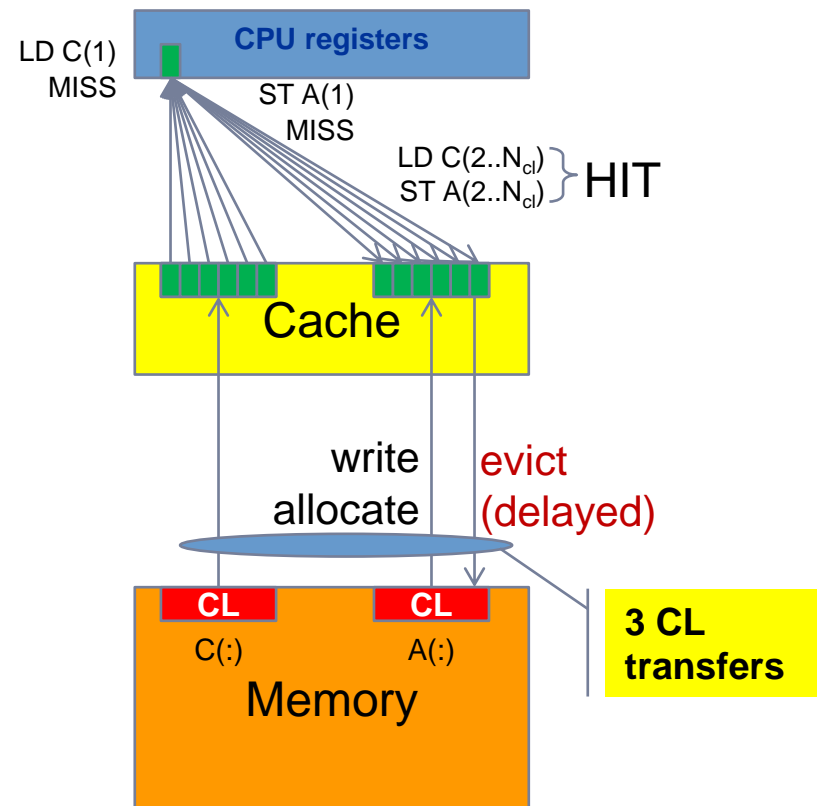
# Registers and caches: Data transfers in a memory hierarchy

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)

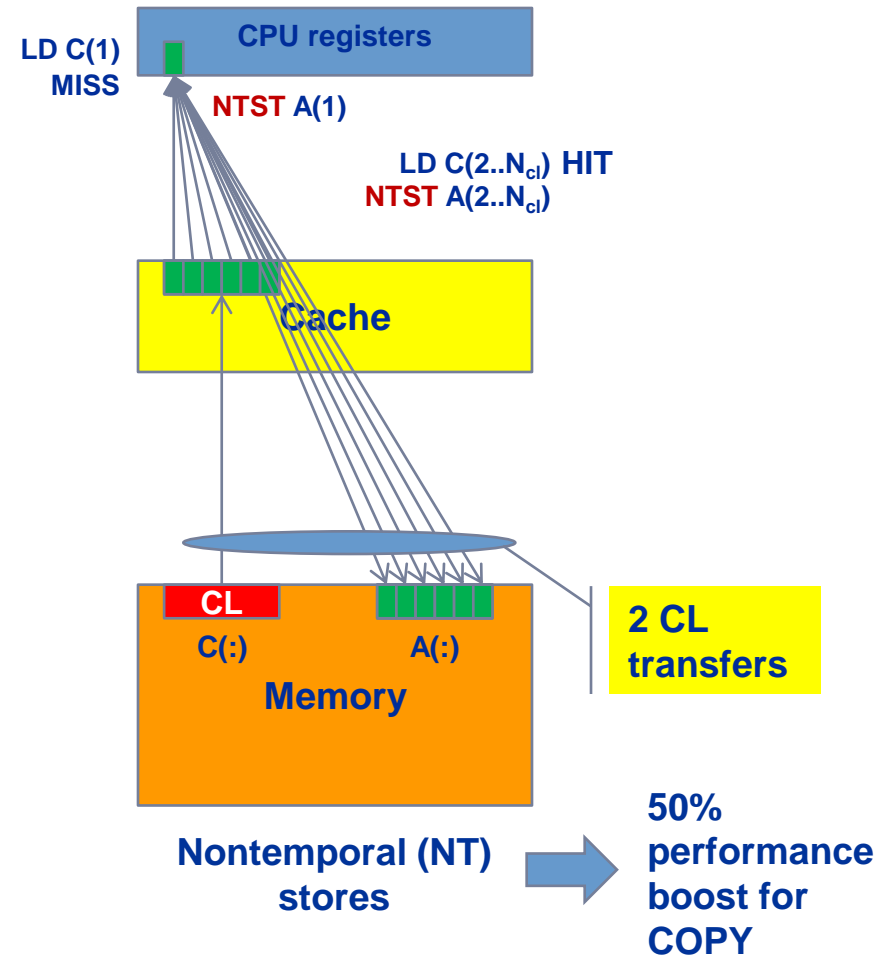
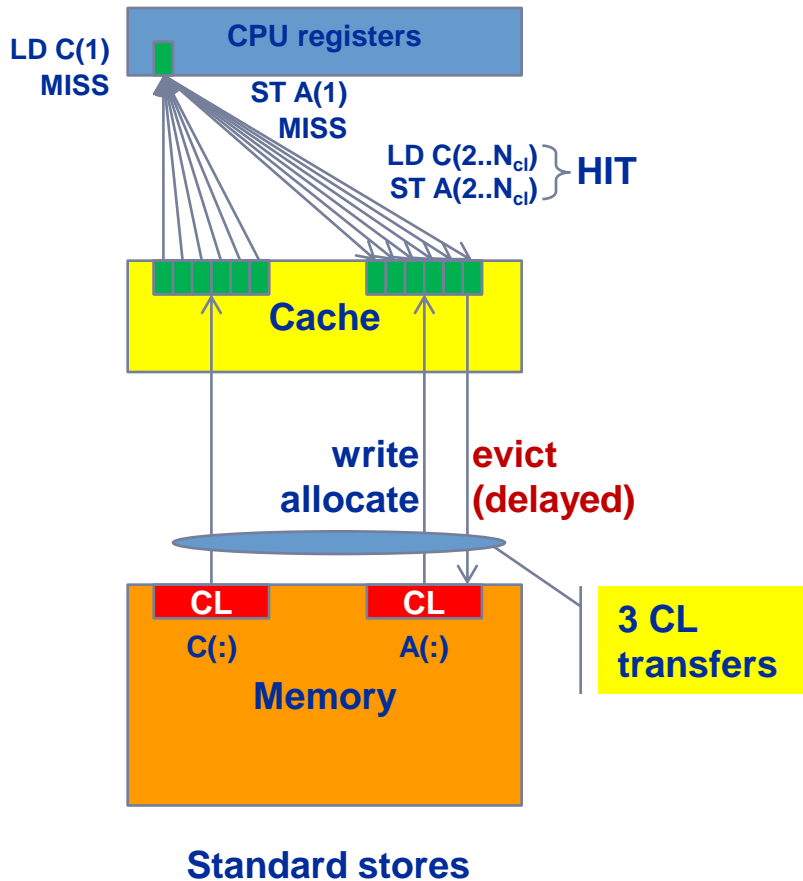
**MISS:** Load or store instruction does not find data in a cache level  
→ CL transfer required

Example: Array copy  $\mathbf{A}(:) = \mathbf{C}(:)$



# Recap: Data transfers in a memory hierarchy

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

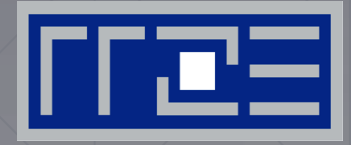
Above requirements may collide with object oriented programming paradigm: **array of structures** vs **structure of arrays**

# 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

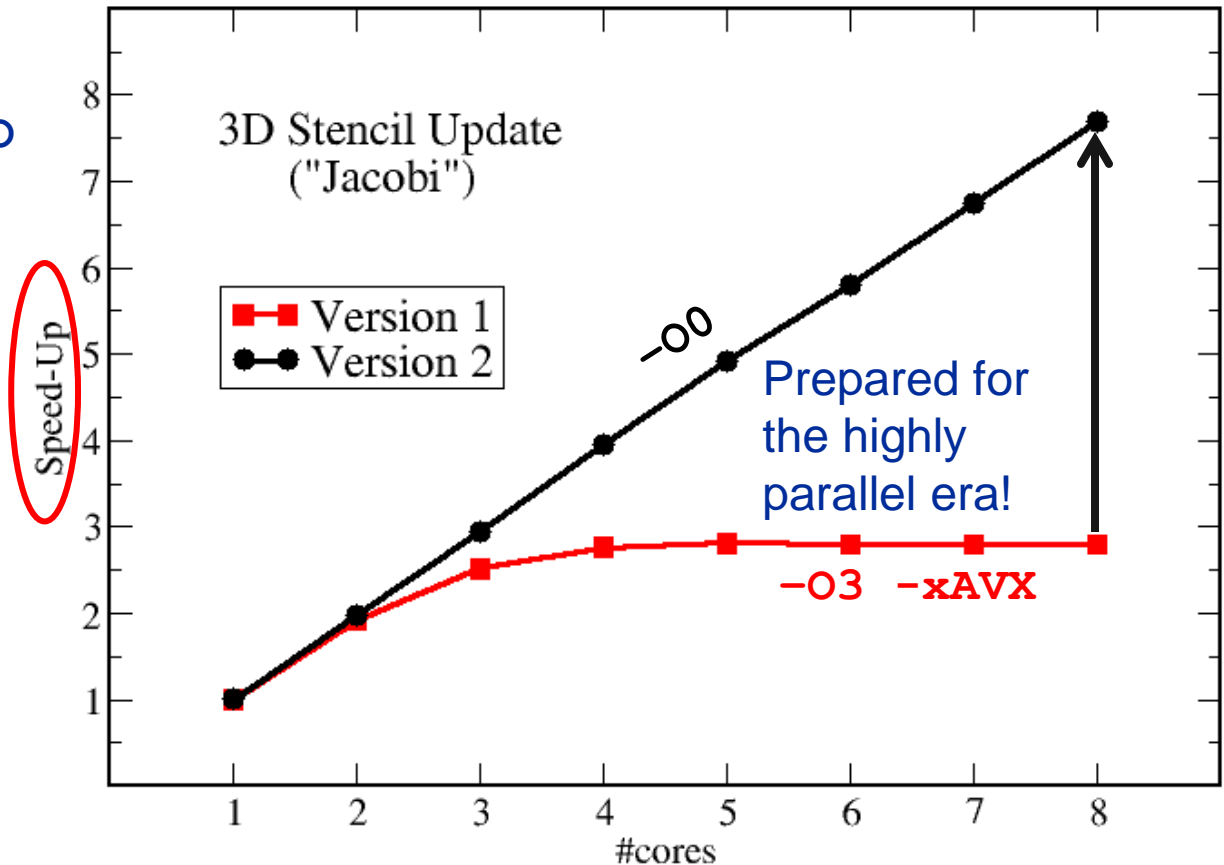
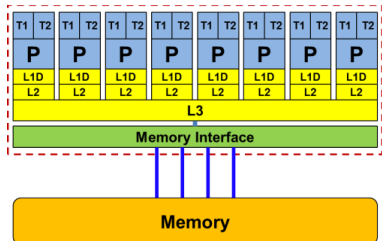
Single core performance no longer matters since we have so many of them and use scalable codes

# 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

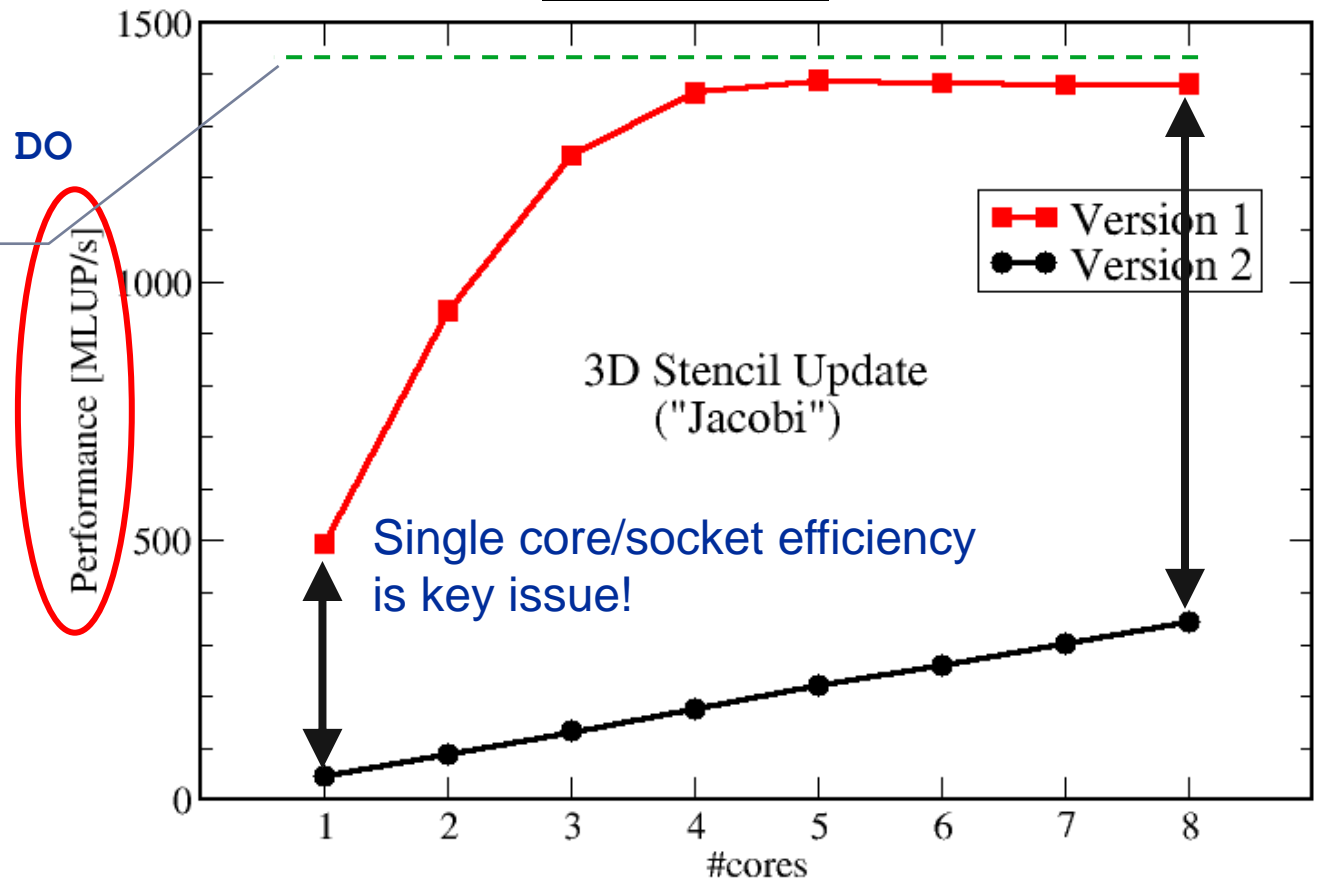
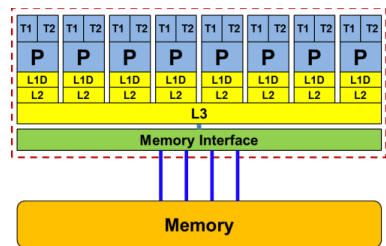


# Scalability Myth: Code scalability is the key issue

```

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

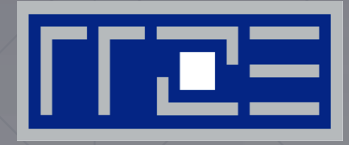
Upper limit from simple performance model:  
35 GB/s & 24 Byte/update





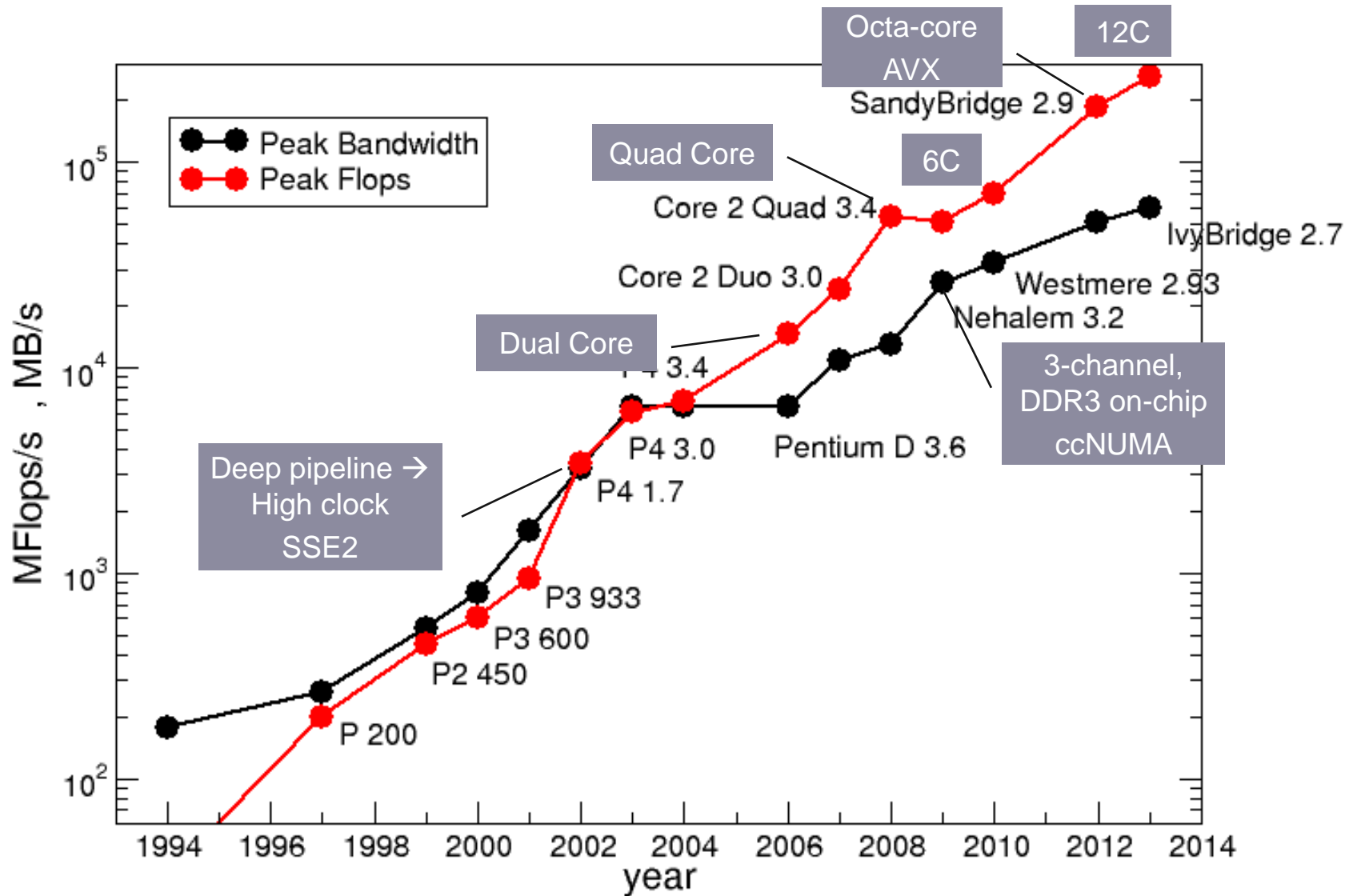


# TOPOLOGY OF MULTI-CORE / MULTI-SOCKET SYSTEMS



- Chip Topology
- Node Topology
- Memory Organisation

# Timeline of technology developments

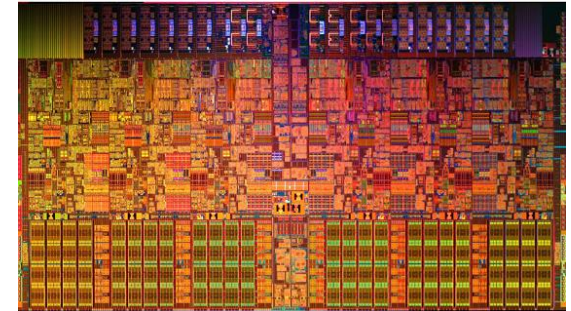


# Building blocks for multi-core compute nodes

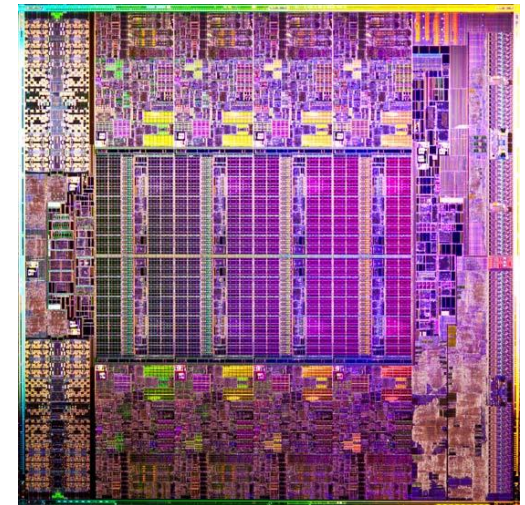
- **Core:** Unit reading and executing instruction stream
- **Chip:** One integrated circuit die
- **Socket/Package:** May consist of multiple chips
- **Memory Hierarchy:**
  - Private caches
  - Shared caches
  - **ccNUMA:** Replicated memory interfaces

# 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<sup>2</sup>

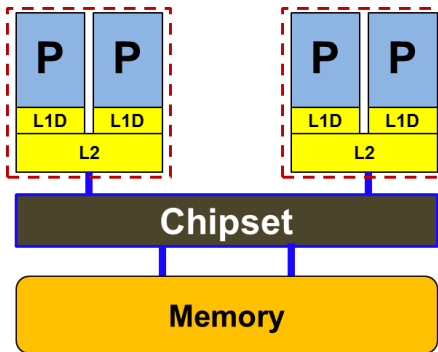


SandyBridge-EP, 8C, 32nm 435mm<sup>2</sup>

# From UMA to ccNUMA

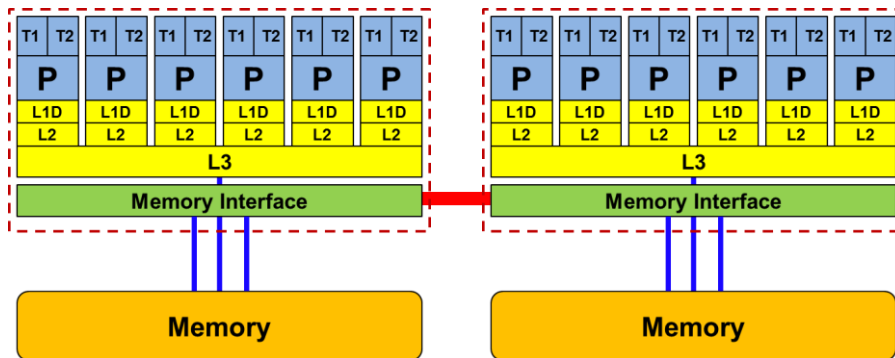
## Memory architectures

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



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

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



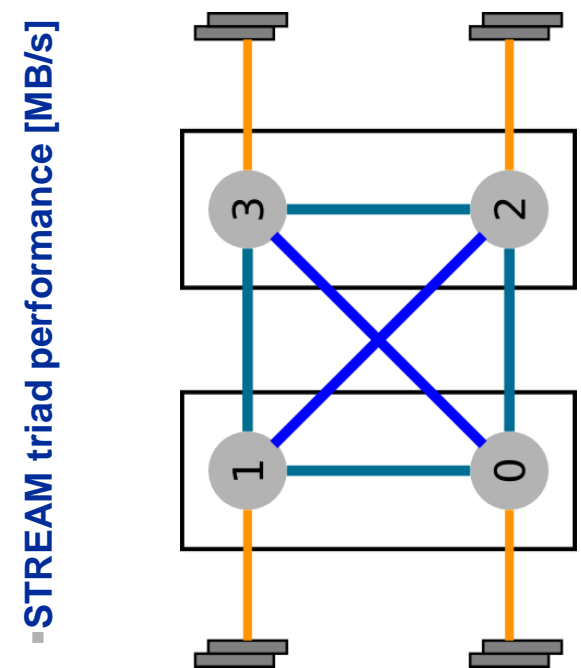
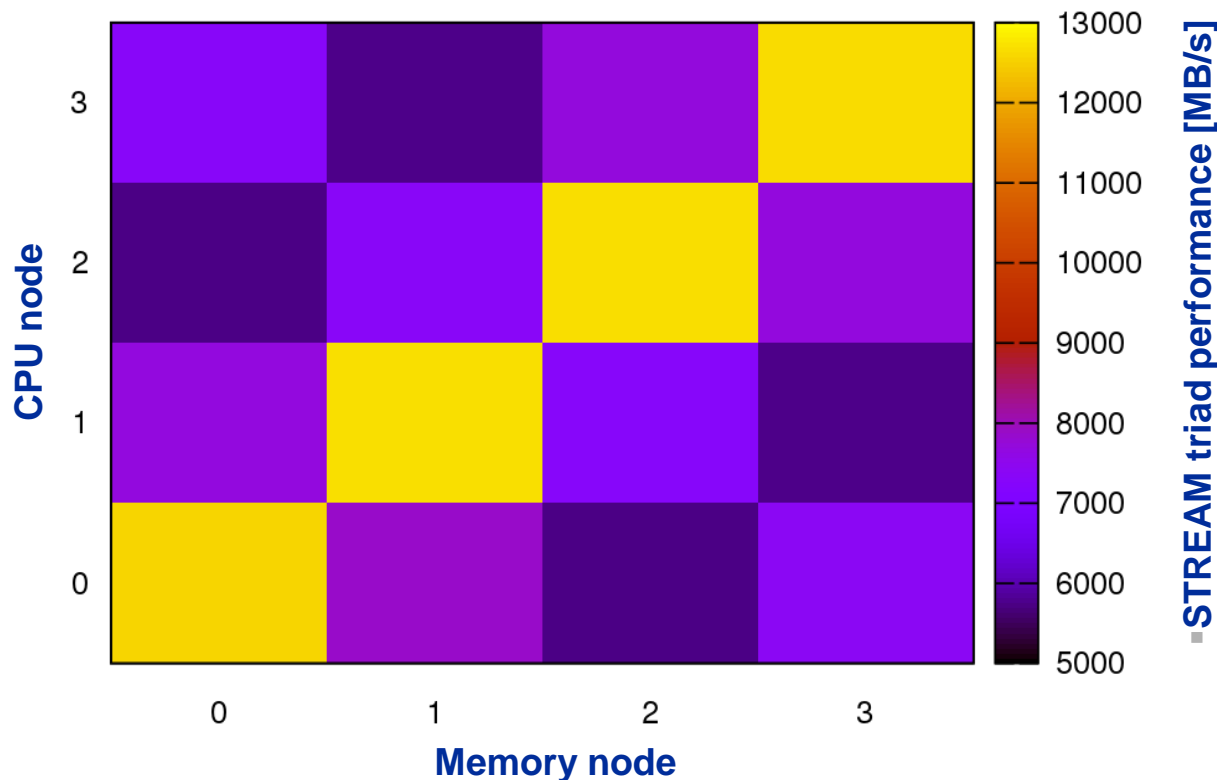
- Cache-coherent Non-Uniform Memory Architecture (**ccNUMA**)
- **HT / QPI** provide scalable bandwidth at the price of ccNUMA architectures: *Where does my data finally end up?*

# ccNUMA

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



# 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

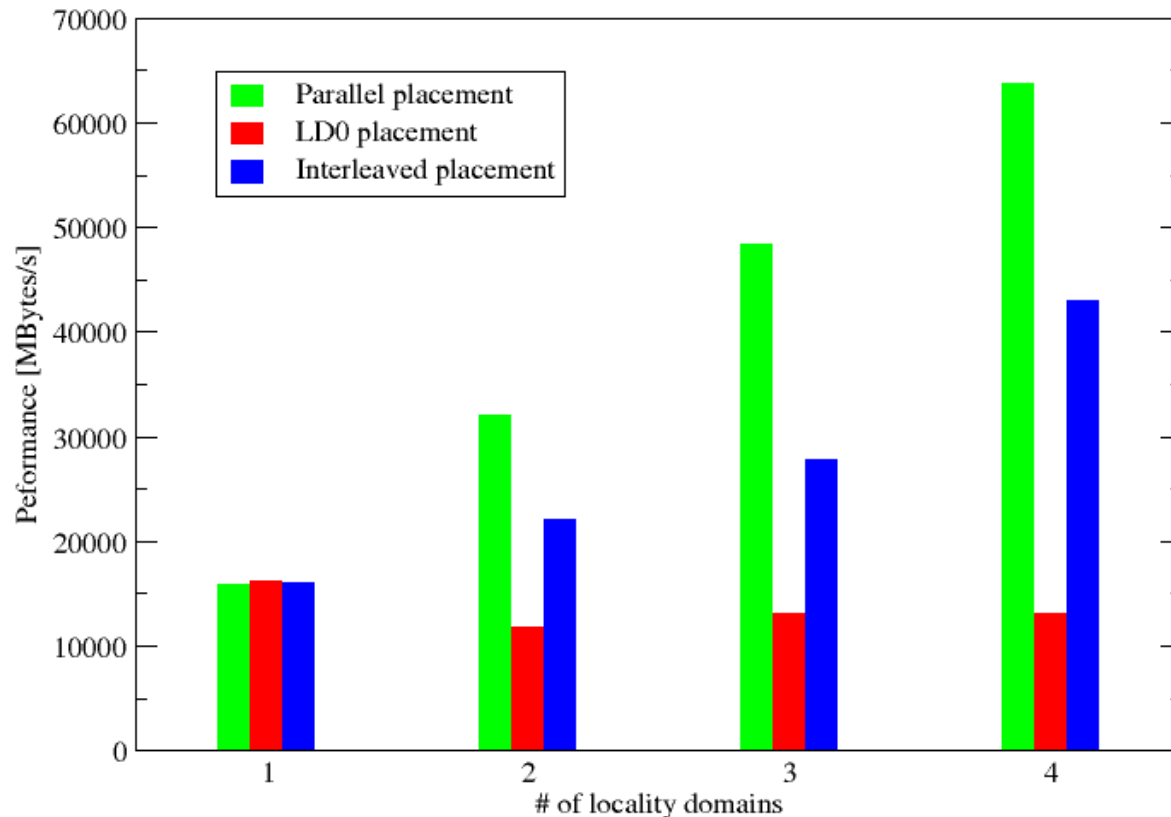
It is sufficient to touch a single item to map the entire page

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

Parallel init: Correct parallel initialization

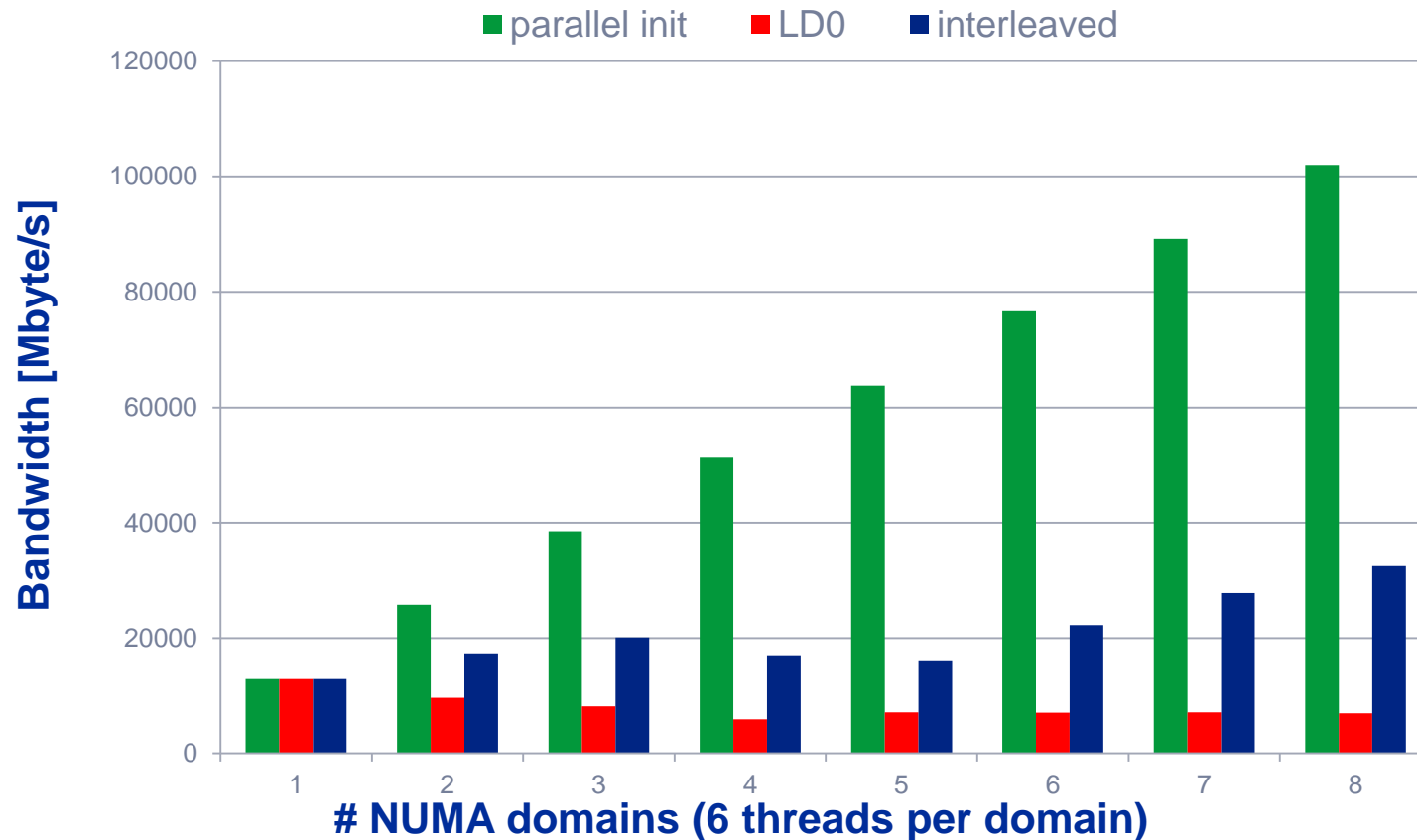
LD0: Force data into LD0 via `numactl -m 0`

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

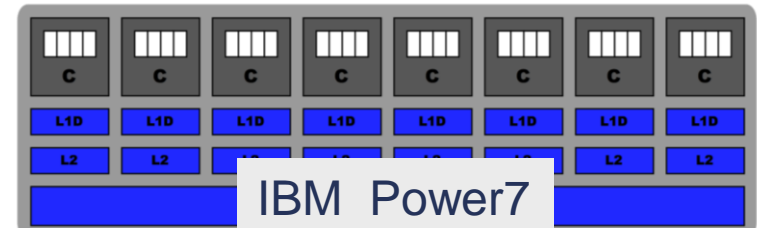
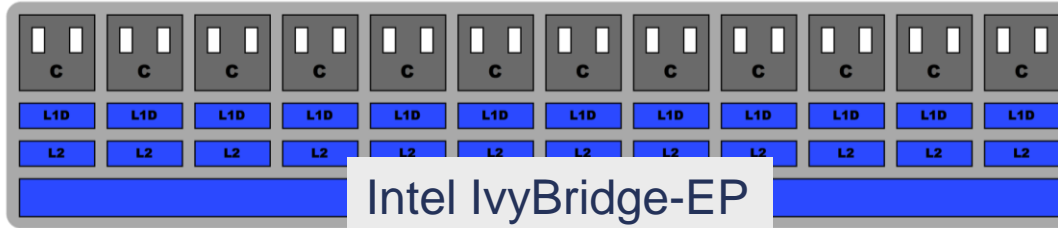




# The curse and blessing of interleaved placement: *same on 4-socket (48 core) Magny Cours node*



# The driving forces behind performance



$$P = n_{\text{core}} * F * S * v$$

	Intel IvyBridge-EP	IBM Power7
Number of cores $n_{\text{core}}$	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] $v$	2.7	3.7
<b>Performance [GF/s] <math>P</math></b>	<b>259 (DP) / 518 (SP)</b>	<b>236 (DP/SP)</b>

TOP500 rank 1 (1996)

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

# Parallel programming models *on modern compute nodes*

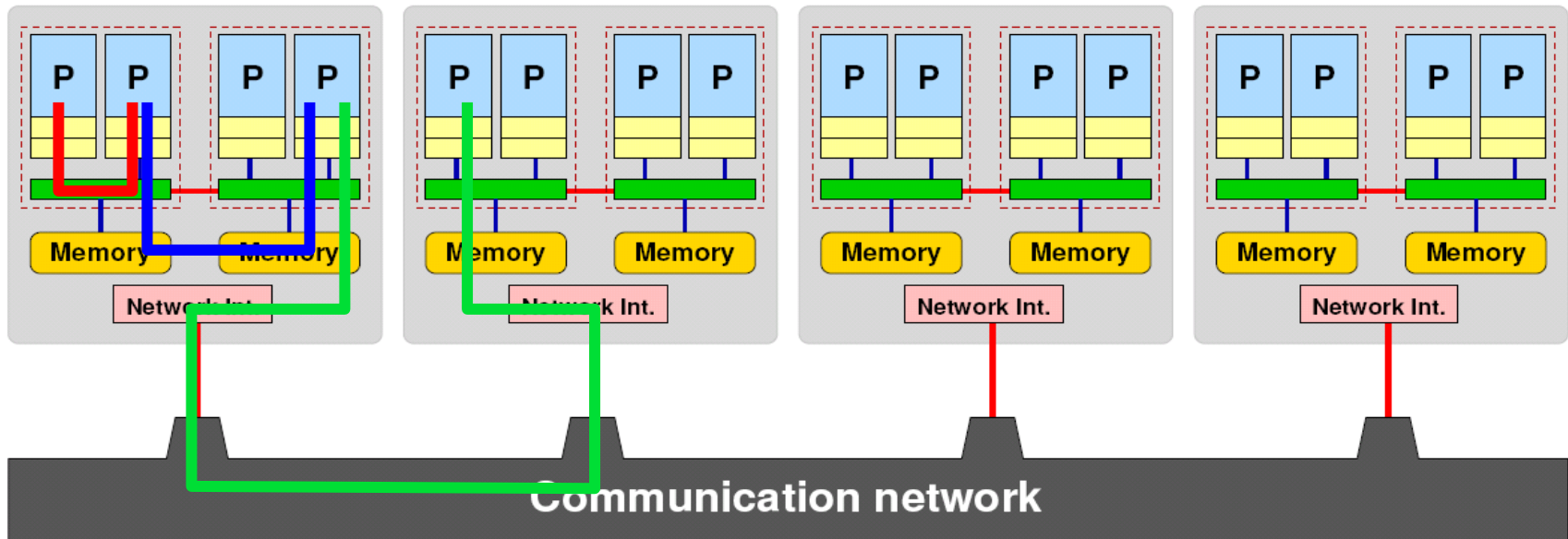
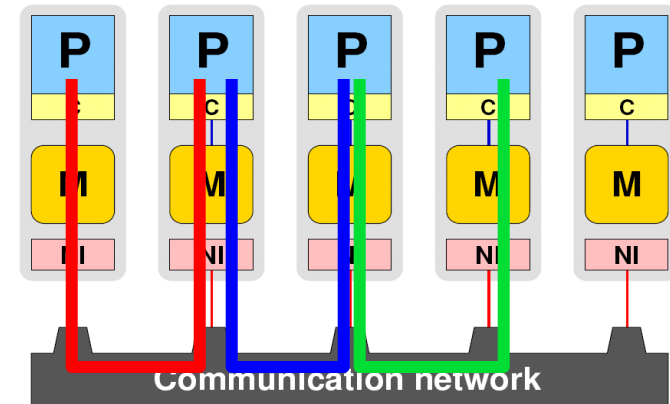
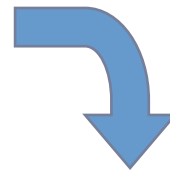
- Shared-memory (intra-node)
  - **Good old MPI** (current standard: 3.0)
  - **OpenMP** (current standard: 4.0)
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs,... you name it
- “Accelerated”
  - OpenMP 4.0
  - CUDA
  - OpenCL
  - OpenACC
- Distributed-memory (inter-node)
  - **MPI** (current standard: 3.0)
  - PVM (gone)
- Hybrid
  - **Pure MPI + X, X == <you name it>**

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

# Parallel programming models:

## Pure MPI

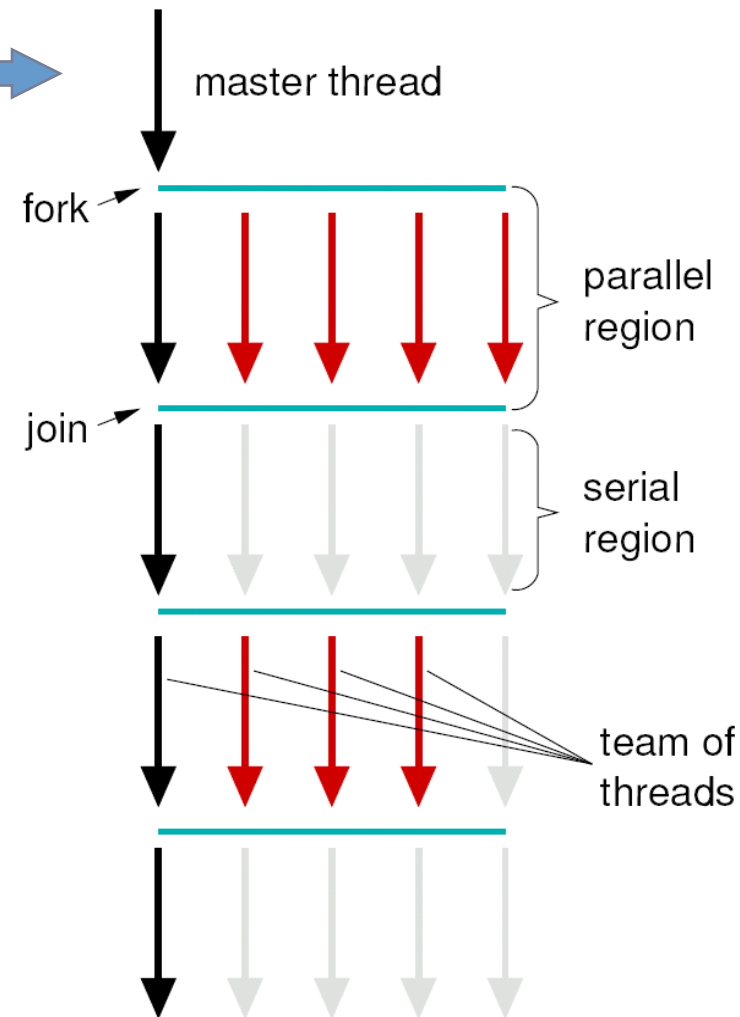
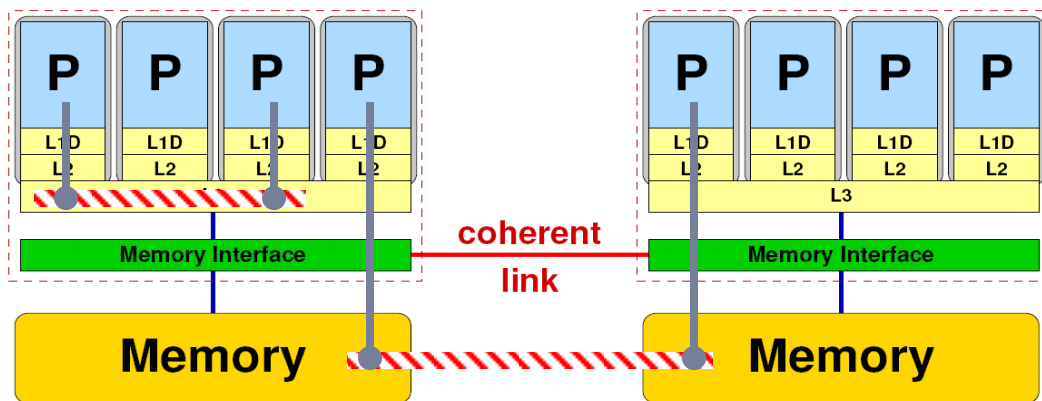
- Machine structure is invisible to user:
  - Very simple programming model
  - MPI “knows what to do”!?
- Performance issues
  - Intranode vs. internode MPI
  - Node/system topology



# Parallel programming models:

## *Pure threading on the node*

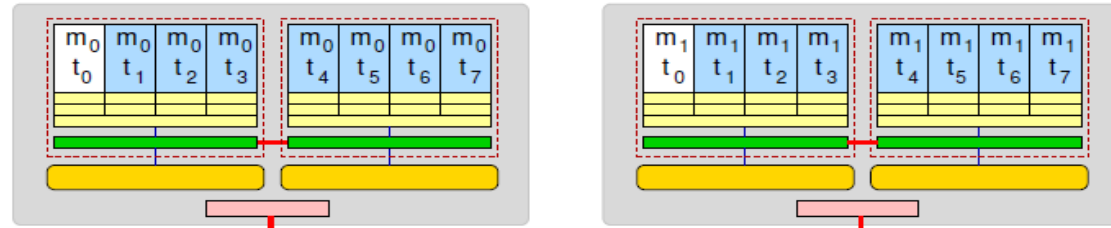
- Machine structure is invisible to user
  - Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,...) should know about the details
- Performance issues
  - Synchronization overhead
  - Memory access
  - Node topology



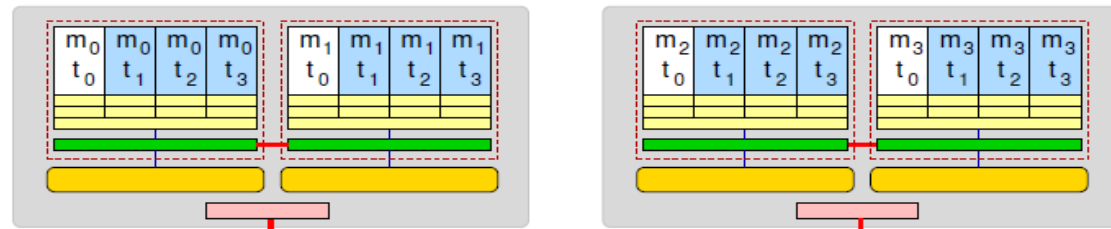
# Parallel programming models: Lots of choices

*Hybrid MPI+OpenMP on a multicore multisocket cluster*

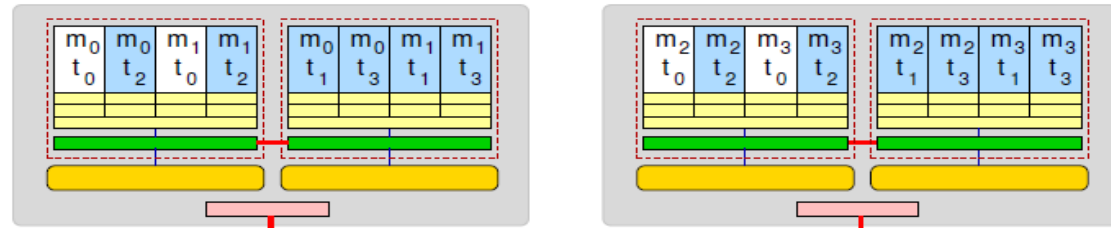
One MPI process / node



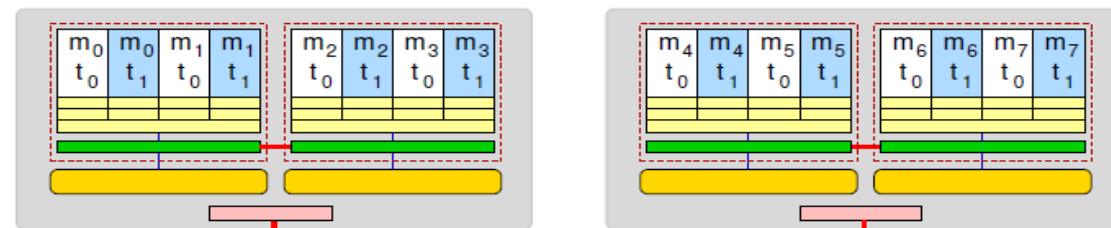
One MPI process / socket:  
OpenMP threads on same  
socket: “blockwise”



OpenMP threads pinned “round  
robin” across cores in node



Two MPI processes / socket  
OpenMP threads  
on same socket



# 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**

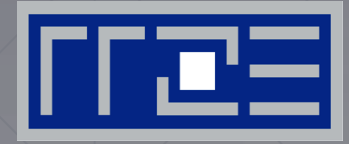
- **“High Performance Computing” == computing at a bottleneck**

**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



# INTERLUDE: A GLANCE AT CURRENT ACCELERATOR TECHNOLOGY

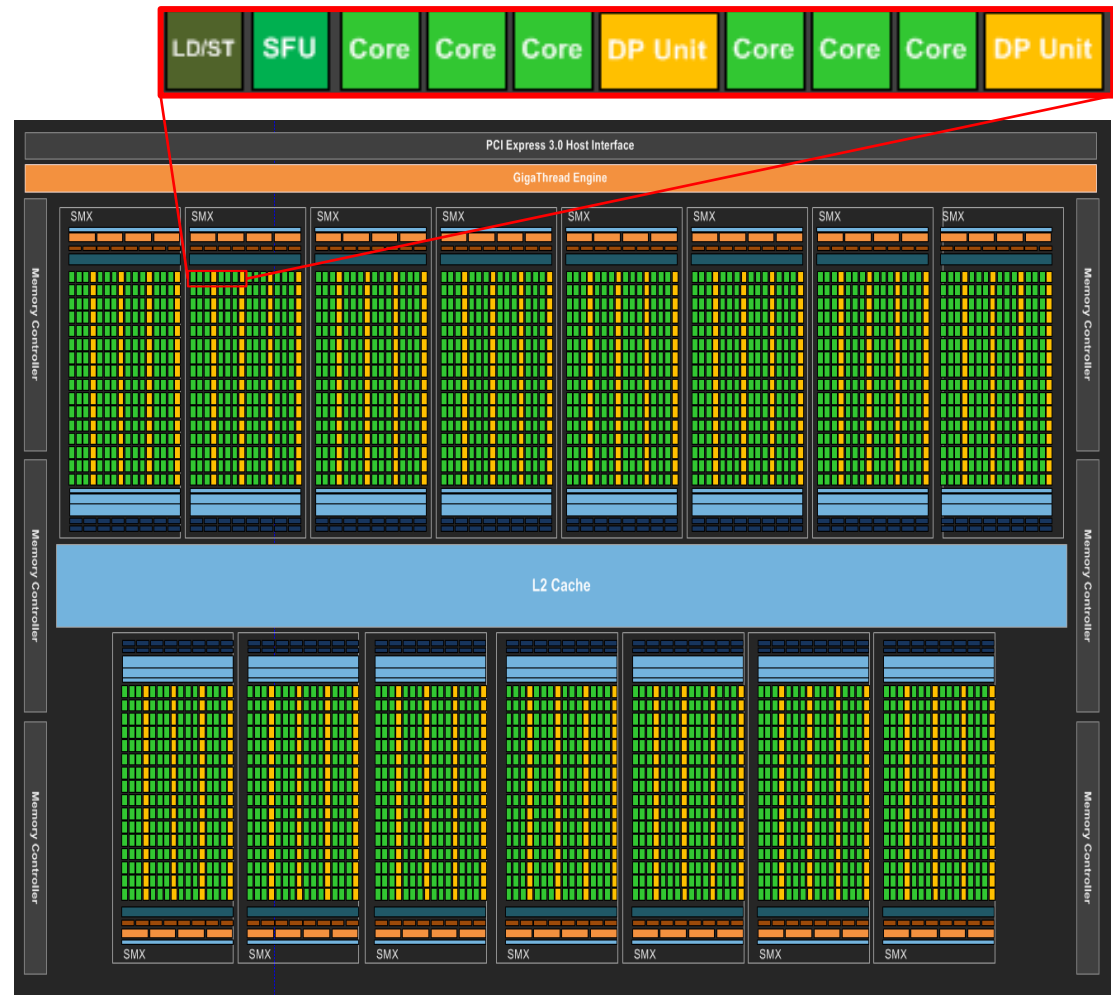




# NVIDIA Kepler GK110 Block Diagram

## Architecture

- 7.1B Transistors
- 15 “SMX” units
  - 192 (SP) “cores” each
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3
  
- 3:1 SP:DP performance

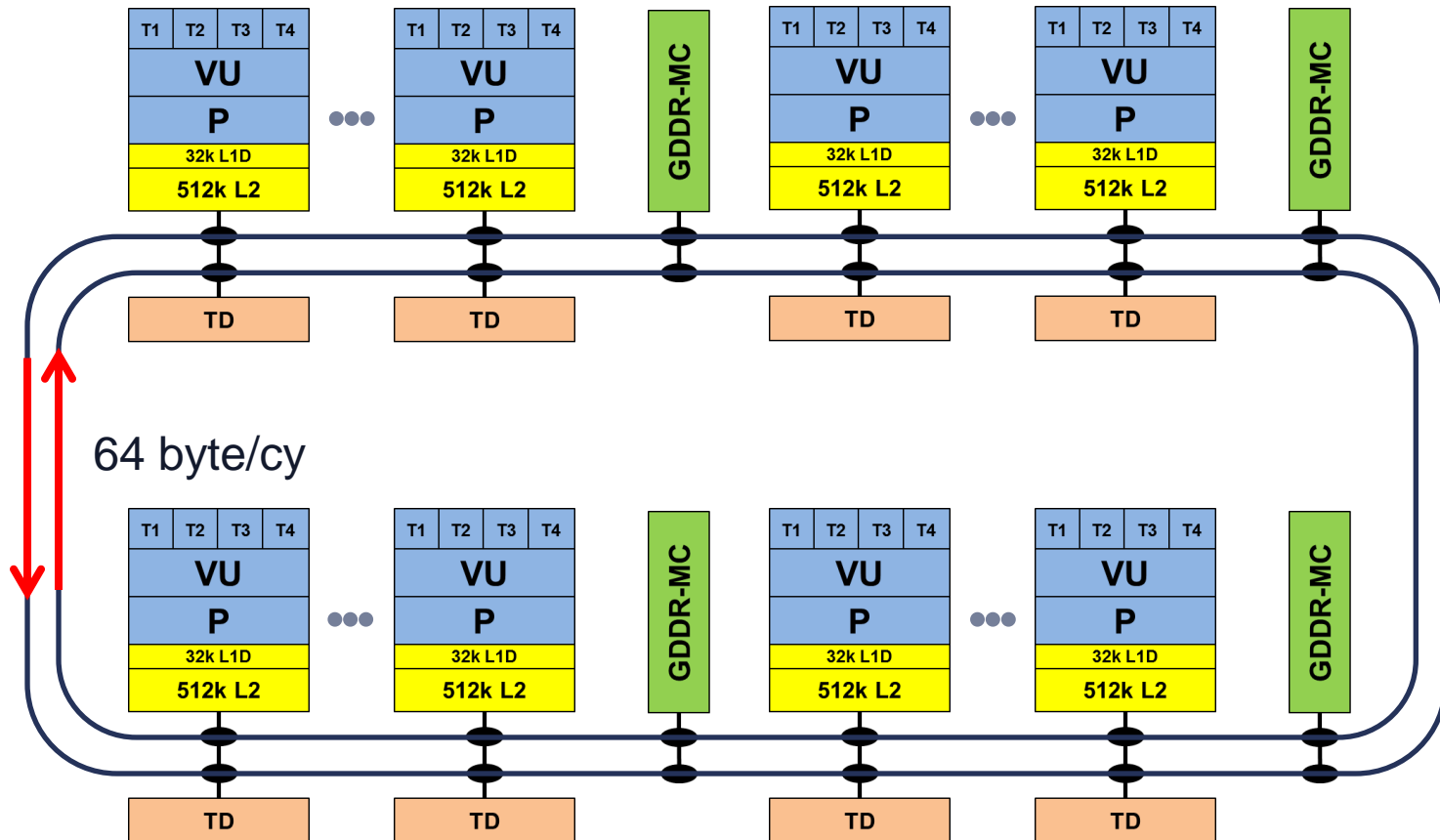


© NVIDIA Corp. Used with permission.

# Intel Xeon Phi block diagram

## Architecture

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



# Comparing accelerators

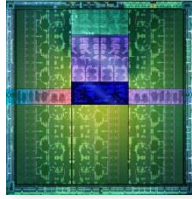
## Intel Xeon Phi

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



## NVIDIA Kepler K20

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



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

**TOP500  
rankings  
Nov 2012**

- TOP1: “Titan” at Oak Ridge National Laboratory

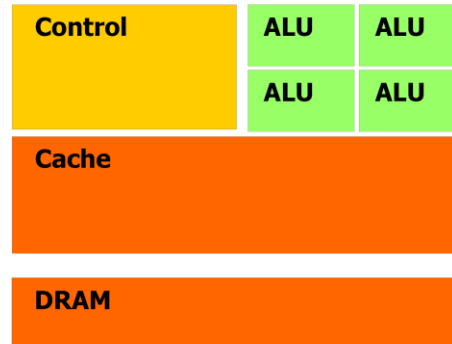
# Trading single thread performance for parallelism:

## GPGPUs vs. CPUs

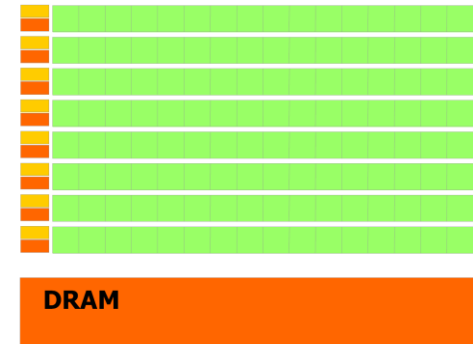
### GPU vs. CPU

light speed estimate:

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



CPU



GPU

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

+ Single Precision

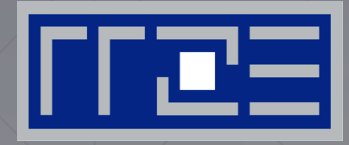
\* Includes on-chip GPU and PCI-Express

Complete compute device



# 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
  - `hwloc` is another option



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

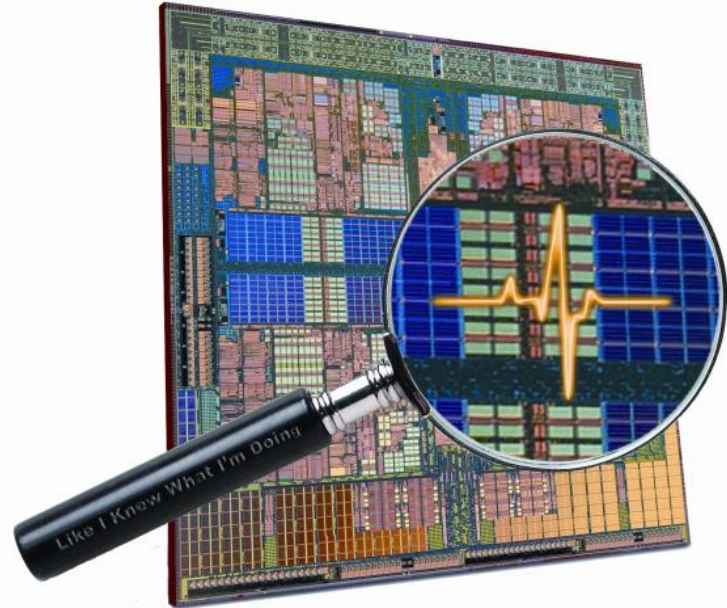
# How do we figure out the node topology?

LIKWID tool suite:

Like  
I  
Knew  
What  
I'm  
Doing

Open source tool collection  
(developed at RRZE):

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



J. Treibig, G. Hager, G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments*. PSTI2010, Sep 13-16, 2010, San Diego, CA  
▪ <http://arxiv.org/abs/1004.4431>

# Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux kernel
  - simple and clear to use
  - supports Intel and AMD CPUs
- Current tools:
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr**: Measure performance counters
  - **likwid-powermeter**: Query turbo mode steps. Measure ETS.
  - **likwid-bench**: Low-level bandwidth benchmark generator tool





# Output of `likwid-topology -g`

*on one node of Cray XE6 "Hermit"*

-----  
CPU type: AMD Interlagos processor

\*\*\*\*\*

## Hardware Thread Topology

\*\*\*\*\*

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

-----

HWTThread	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:

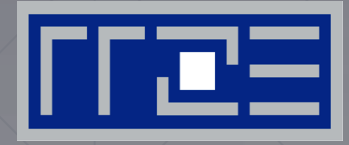
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB	16kB
2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB
6MB								6MB							

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	2MB	2MB	2MB	2MB	2MB	2MB	2MB	2MB
6MB								6MB							



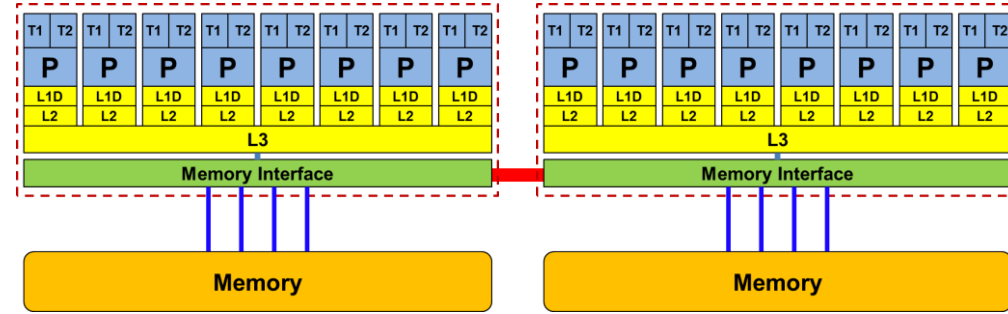
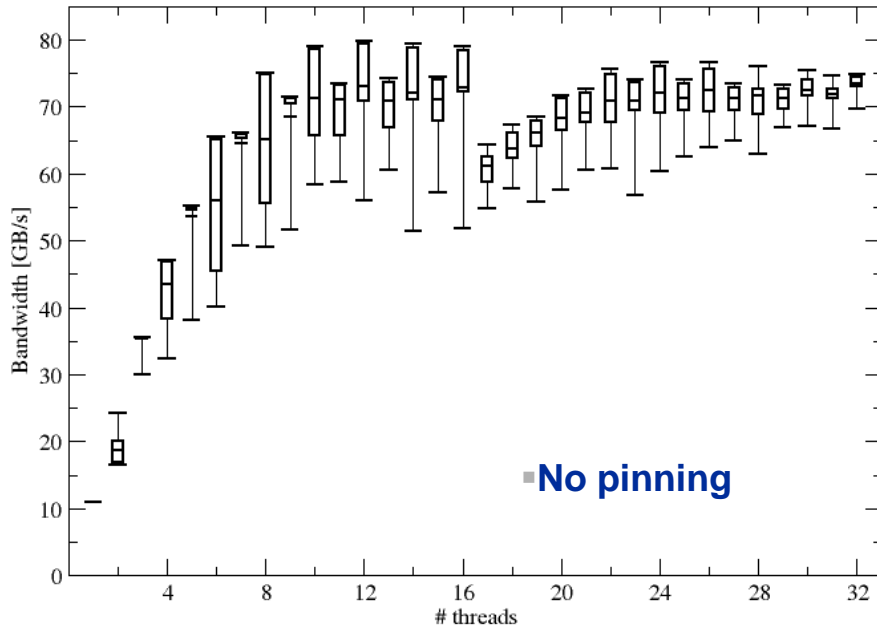
# ENFORCING THREAD/PROCESS-CORE AFFINITY UNDER THE LINUX OS



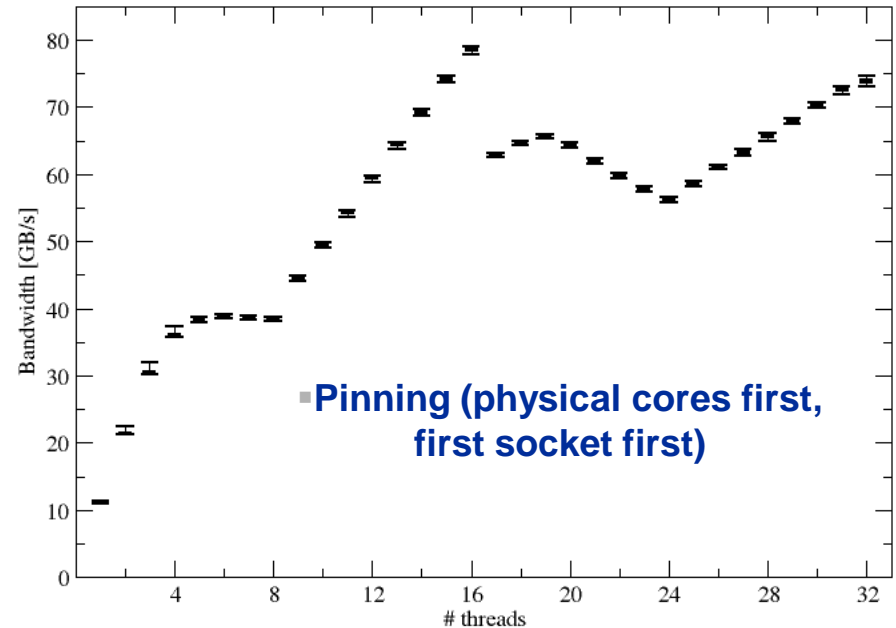
- Standard tools and OS affinity facilities under program control
- `likwid-pin`

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

## Anarchy vs. thread pinning



- There are several reasons for caring about affinity:
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention
- Benchmark how code reacts to variations



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

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`
    - Windows: `SetThreadAffinityMask()`
  - OpenMPI: hwloc library
- Support for “semi-automatic” pinning in some compilers/environments
  - All modern compilers with OpenMP support
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
  - OpenMP 4.0
  - 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:
  - `likwid-pin -c 0,2,4-6 ./myApp parameters`
  - `likwid-pin -c S0:0-3 ./myApp parameters`

# Likwid-pin

Example: Intel OpenMP

Running the STREAM benchmark with likwid-pin:

```
$ 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

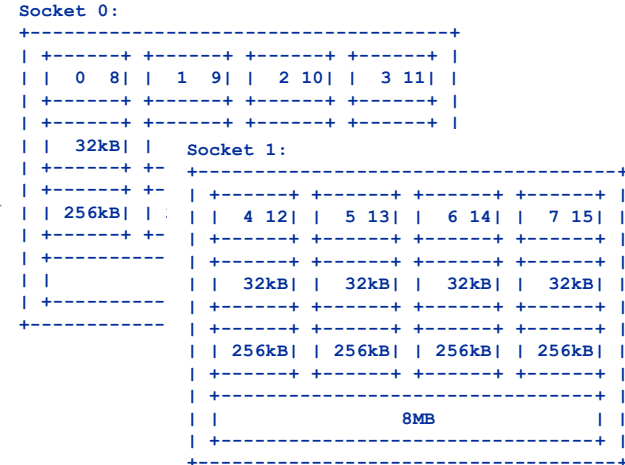
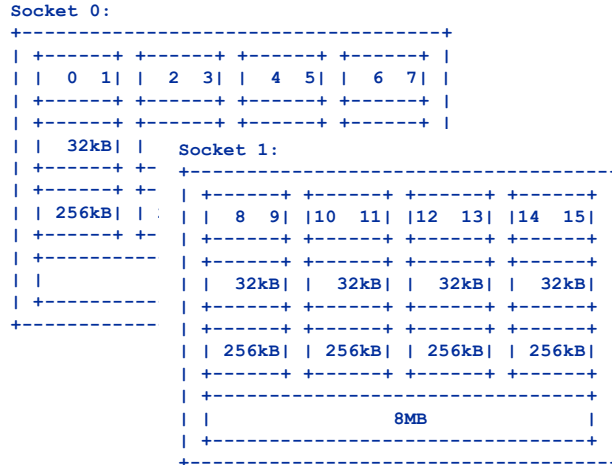
Pin all spawned  
threads in turn



# Likwid-pin

## Using logical core numbering

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



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

# Likwid-pin

## Using logical core numbering

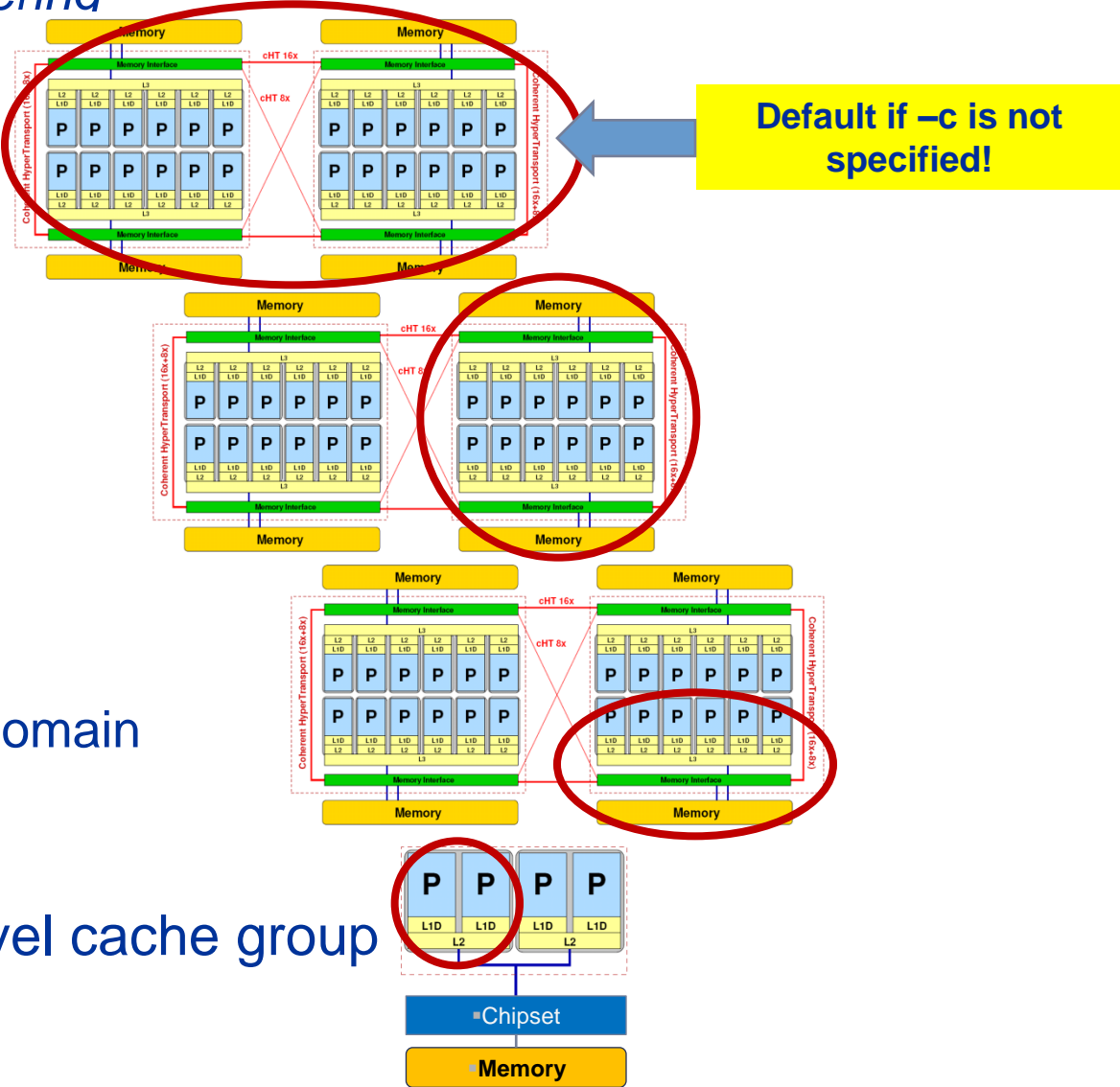
- Possible unit prefixes

N node

S socket

M NUMA domain

C outer level cache group



# Advanced options for pinning: Expressions

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

## Compact pinning:

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

## Scattered pinning across all domains of the designated type :

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

- **Examples:**

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

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

- **Scatter across all NUMA domains:**

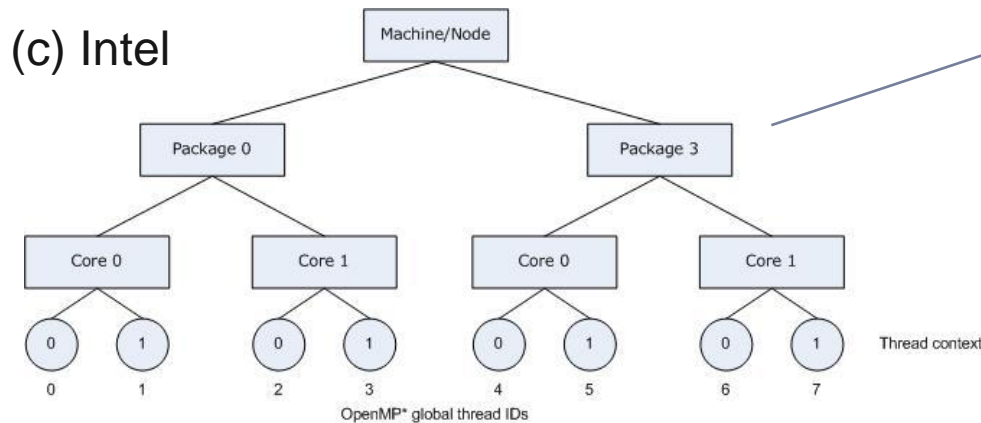
```
likwid-pin -c M:scatter
```

# Intel KMP\_AFFINITY environment variable

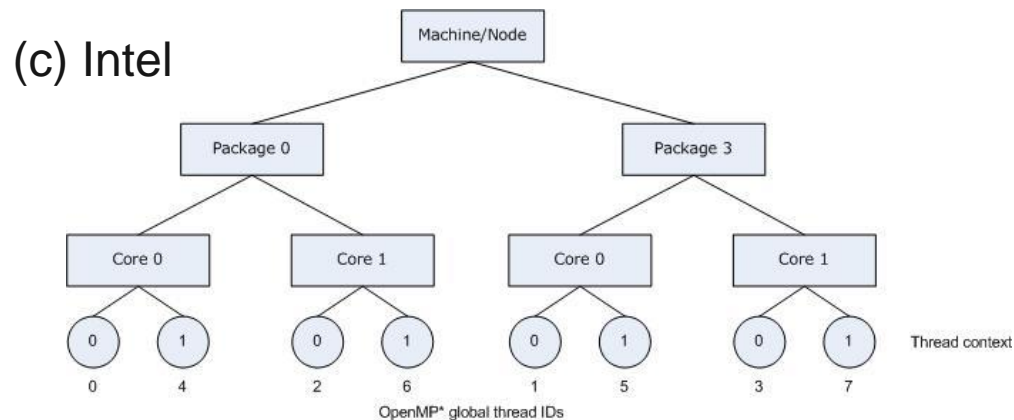
- `KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]`
  - **modifier**
    - `granularity=<specifier>` takes the following specifiers: fine, thread, and core
    - `norespect`
    - `noverbose`
    - `proclist={<proc-list>}`
    - `respect`
    - `verbose`
  - **type (required)**
    - `compact`
    - `disabled`
    - `explicit` (`GOMP_CPU_AFFINITY`)
    - `none`
    - `scatter`
  - **Default:**  
`noverbose, respect, granularity=core`
  - `KMP_AFFINITY=verbose, none` to list machine topology map
- 
- The diagram consists of two yellow rectangular boxes with blue text. The top box is labeled 'OS processor IDs' and has a line connecting it to the 'proclist={<proc-list>}' modifier in the list. The bottom box is labeled 'Respect an OS affinity mask in place' and has a line connecting it to the 'respect' modifier in the list.

# Intel KMP\_AFFINITY examples

- KMP\_AFFINITY=granularity=fine,compact

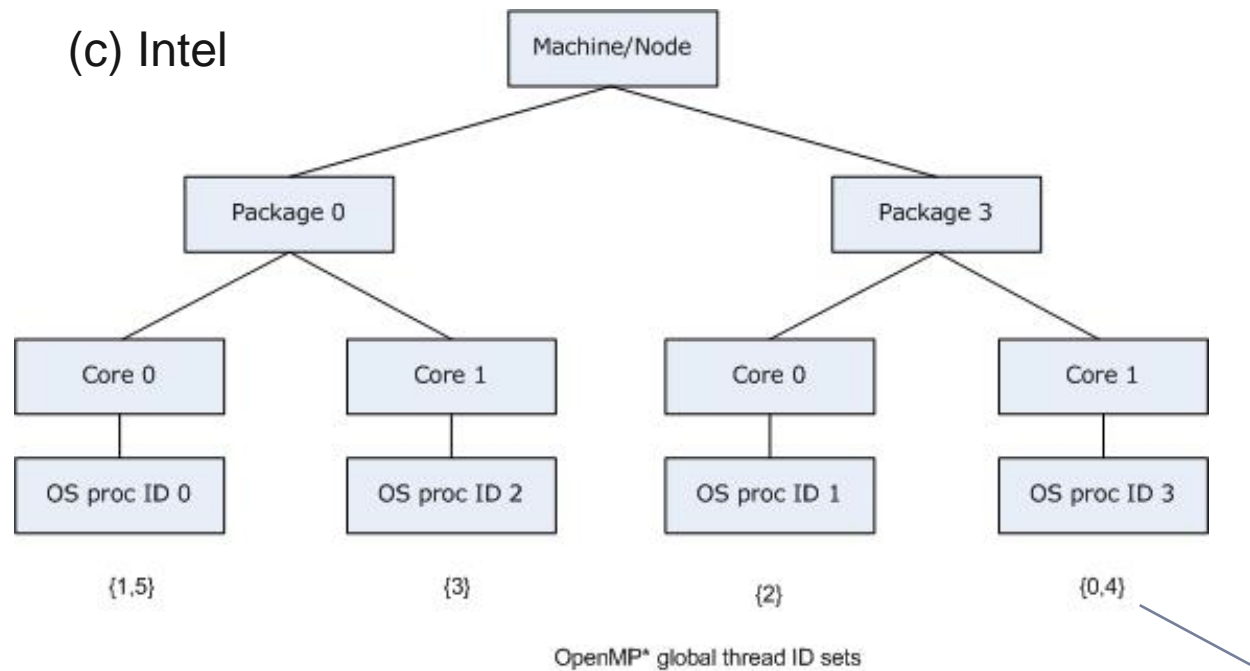


- KMP\_AFFINITY=granularity=fine,scatter



# GNU GOMP\_AFFINITY

- `GOMP_AFFINITY=3, 0-2` used with 6 threads

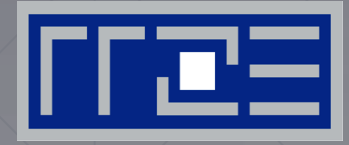


**Round robin  
oversubscription**

- Always operates with OS processor IDs



# 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

Always  
measured

Configured metrics  
(this group)

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

Derived  
metrics

# likwid-perfctr

## Marker API

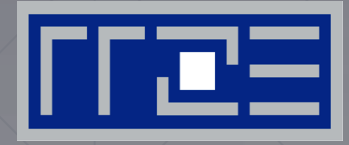
- A marker API is available to restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr
- Multiple named regions support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid.h>
. . .
LIKWID_MARKER_INIT;           // must be called from serial region
#pragma omp parallel
{
    LIKWID_MARKER_THREADINIT; // only reqd. if measuring multiple threads
}
. . .
LIKWID_MARKER_START("Compute");
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .
LIKWID_MARKER_CLOSE;        // must be called from serial region
```

Activate macros with  
-DLIKWID\_PERFMON



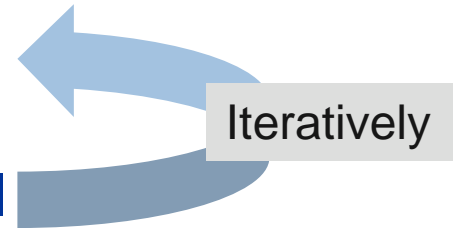
# PATTERN-DRIVEN PERFORMANCE ENGINEERING PROCESS



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



## 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 variants: 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)

# Thinking in Bottlenecks

- A bottleneck is a performance limiting setting
- Microarchitectures expose numerous bottlenecks

## Observation 1:

Most applications face a single bottleneck at a time!

## Observation 2:

There is a limited number of relevant bottlenecks!



# Process vs. Tool

Reduce complexity!

We propose a human driven process to enable a systematic way to success!

- Executed by humans.
- Uses tools by means of data acquisition only.

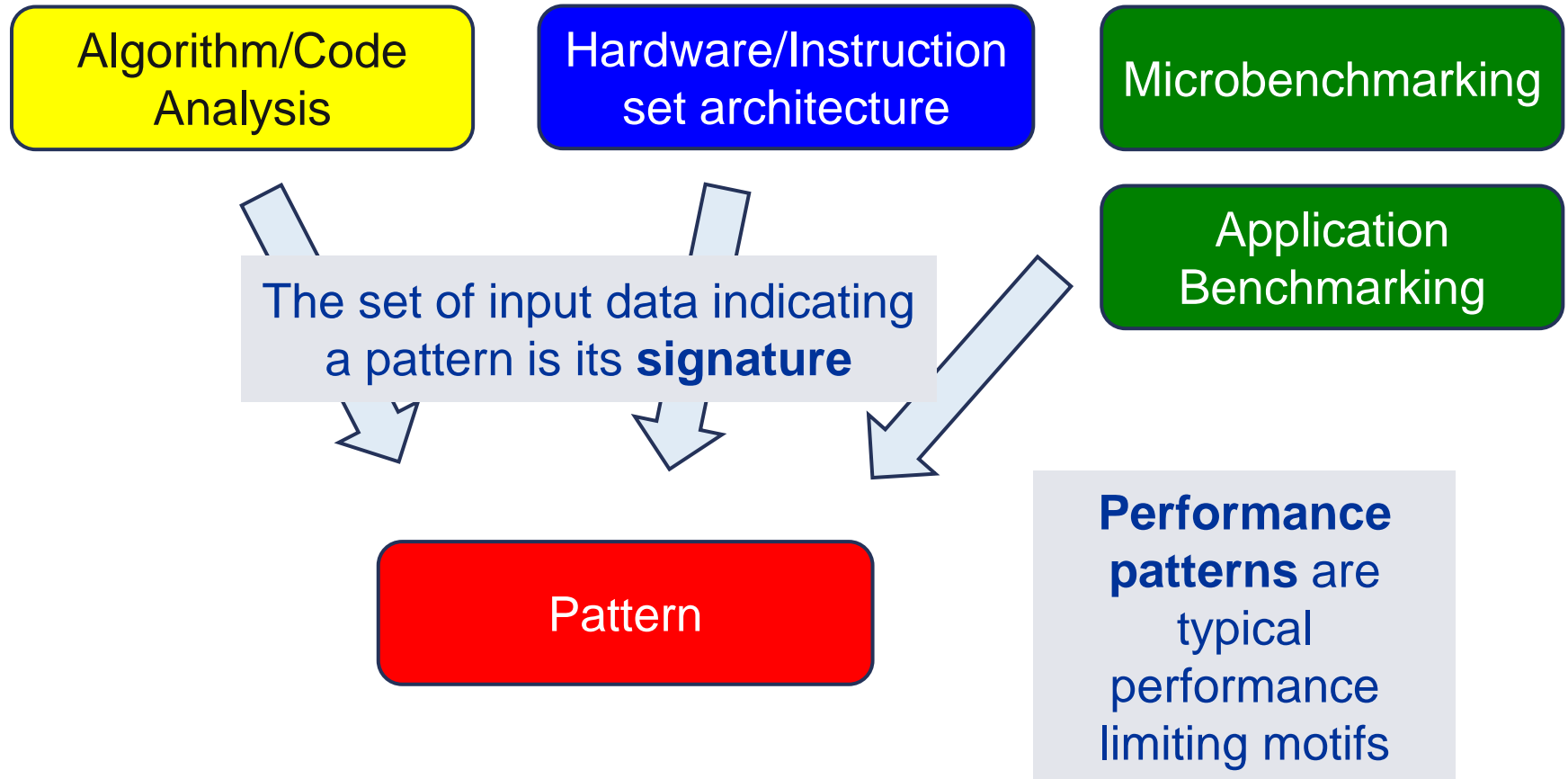
Uses one of the most powerful tools available:

**Your brain !**

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



# Performance Engineering Process: Analysis



Step 1 **Analysis**: Understanding observed performance

# Performance analysis phase

Understand observed performance: **Where am I?**

Input:

- Static code analysis
- HPM data
- Scaling data set size
- Scaling number of used cores
- Microbenchmarking

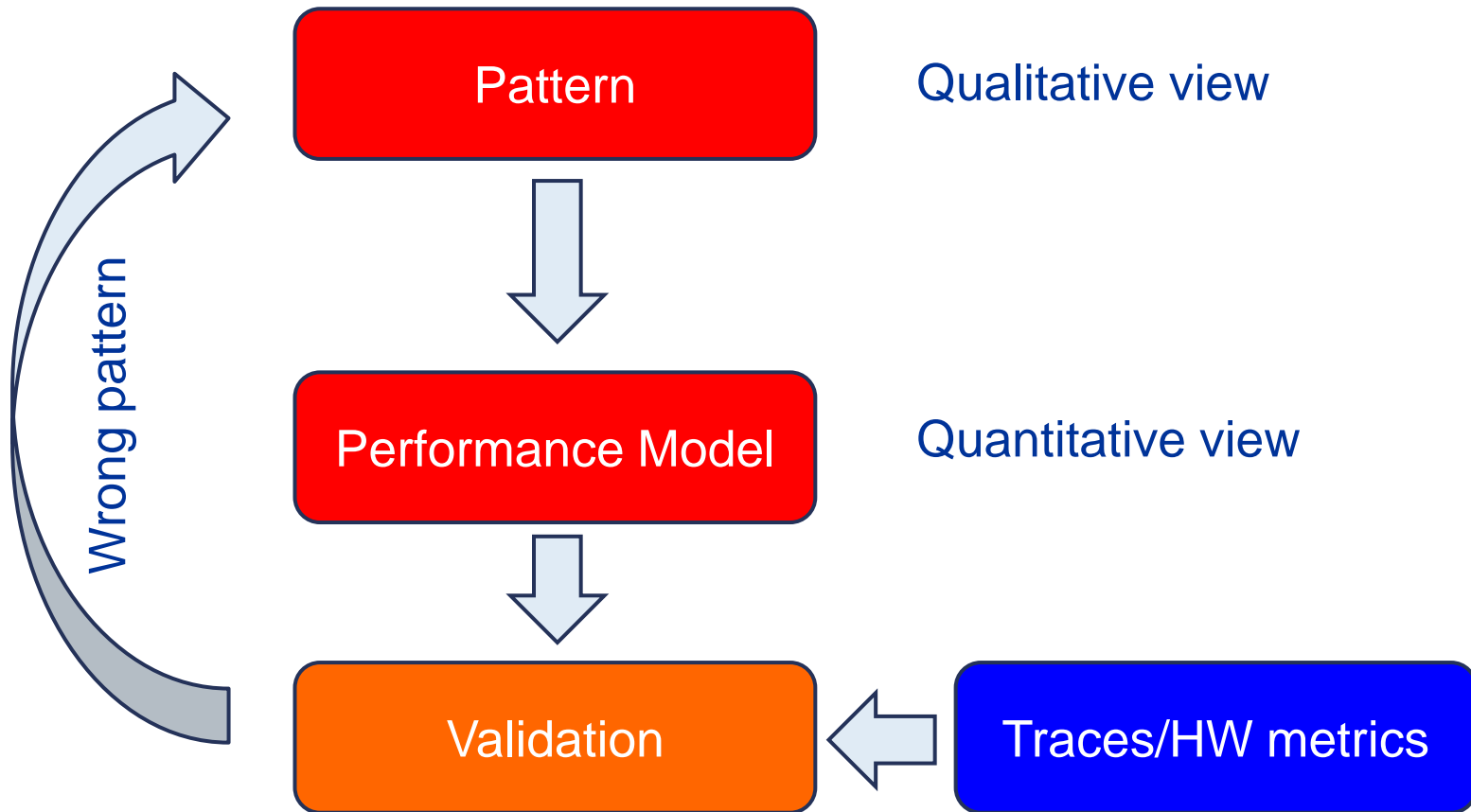


Signature

**Pattern**

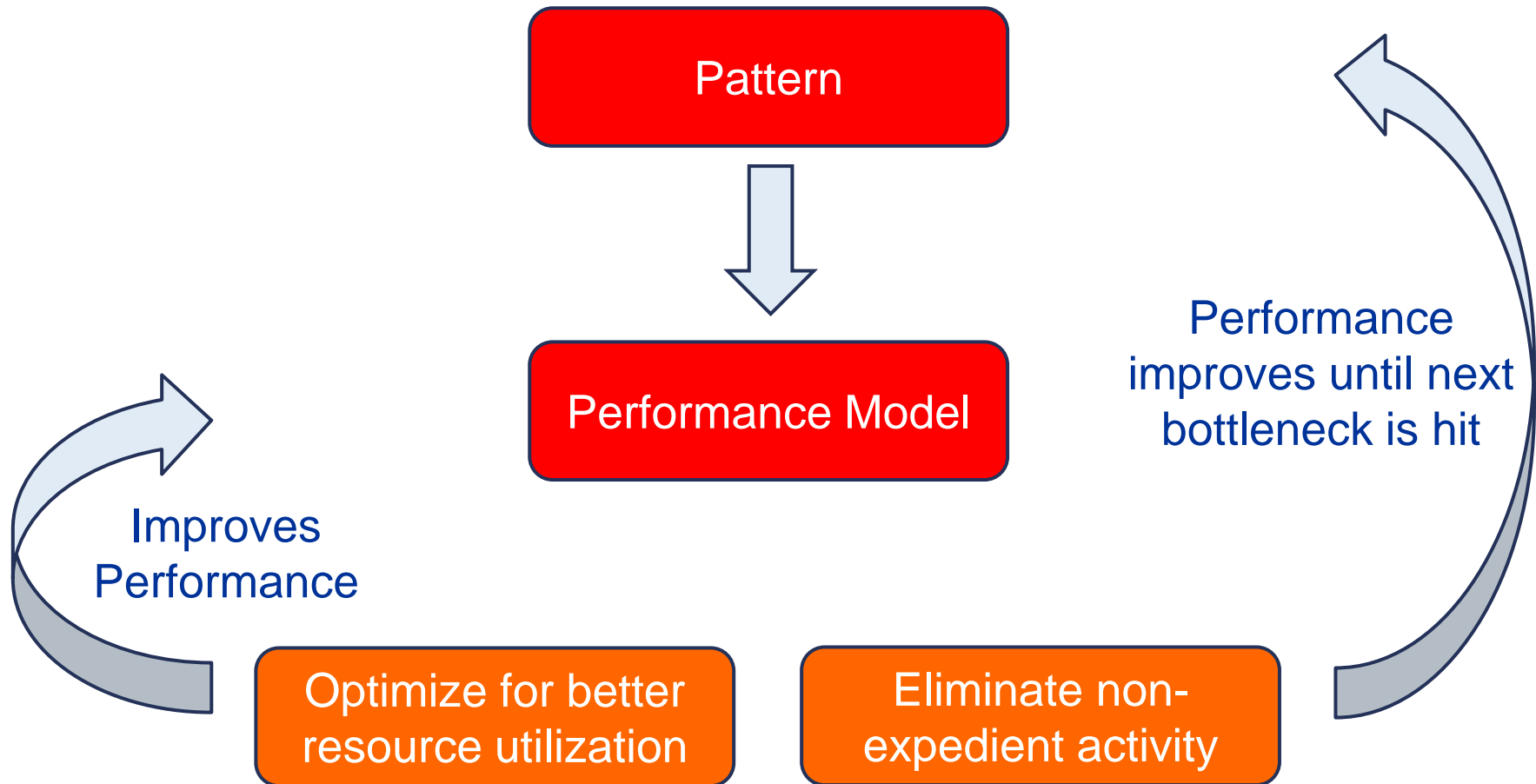
**Performance patterns** are typical performance limiting motives.  
The set of input data indicating a pattern is its **signature**.

# Performance Engineering Process: Modelling



**Step 2 Formulate Model:** Validate pattern and get quantitative insight.

# Performance Engineering Process: Optimization



Step 3 **Optimization**: Improve utilization of offered resources.

# 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)

# Patterns (I): Bottlenecks & hazards

Pattern		Performance behavior	Metric signature, LIKWID performance group(s)
<b>Bandwidth saturation</b>		Saturating speedup across cores sharing a data path	Bandwidth meets BW of suitable streaming benchmark (MEM, L3)
<b>ALU saturation</b>		Throughput at design limit(s)	Good (low) CPI, integral ratio of cycles to specific instruction count(s) (FLOPS_*, DATA, CPI)
<b>Inefficient data access</b>	<b>Excess data volume</b>	Simple bandwidth performance model much too optimistic	Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)
	<b>Latency-bound access</b>		
<b>Micro-architectural anomalies</b>		Large discrepancy from simple performance model based on LD/ST and arithmetic throughput	Relevant events are very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events

# Patterns (II): Hazards

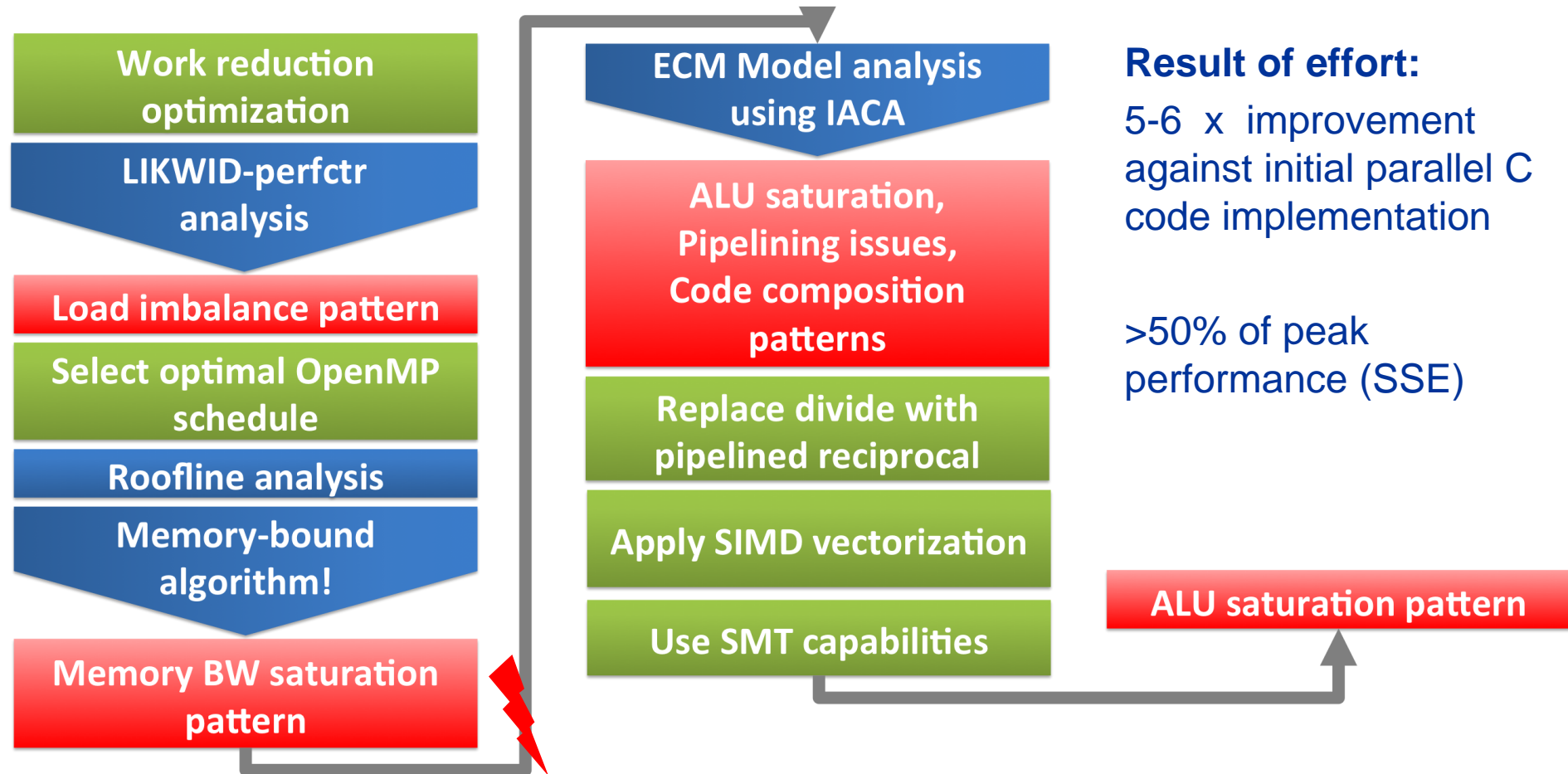
Pattern	Performance behavior	Metric signature, LIKWID performance group(s)
<b>False sharing of cache lines</b>	Large discrepancy from performance model in parallel case, bad scalability	Frequent (remote) CL evicts (CACHE)
<b>Bad ccNUMA page placement</b>	Bad or no scaling across NUMA domains, performance improves with interleaved page placement	Unbalanced bandwidth on memory interfaces / High remote traffic (MEM)
<b>Pipelining issues</b>	In-core throughput far from design limit, performance insensitive to data set size	(Large) integral ratio of cycles to specific instruction count(s), bad (high) CPI (FLOPS_*, DATA, CPI)
<b>Control flow issues</b>	See above	High branch rate and branch miss ratio (BRANCH)



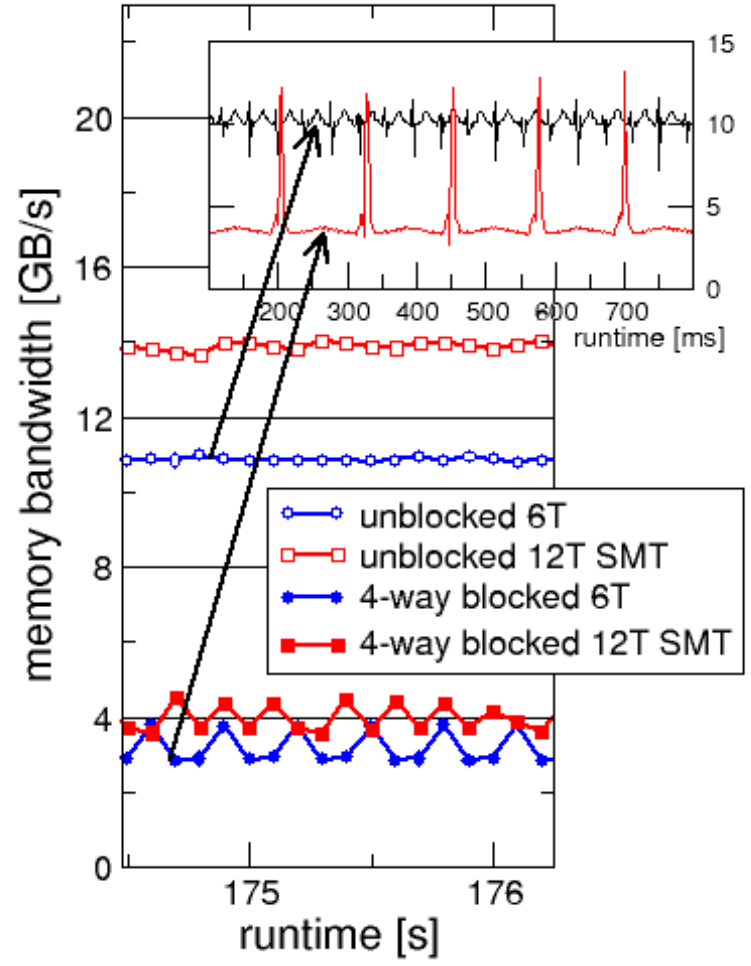
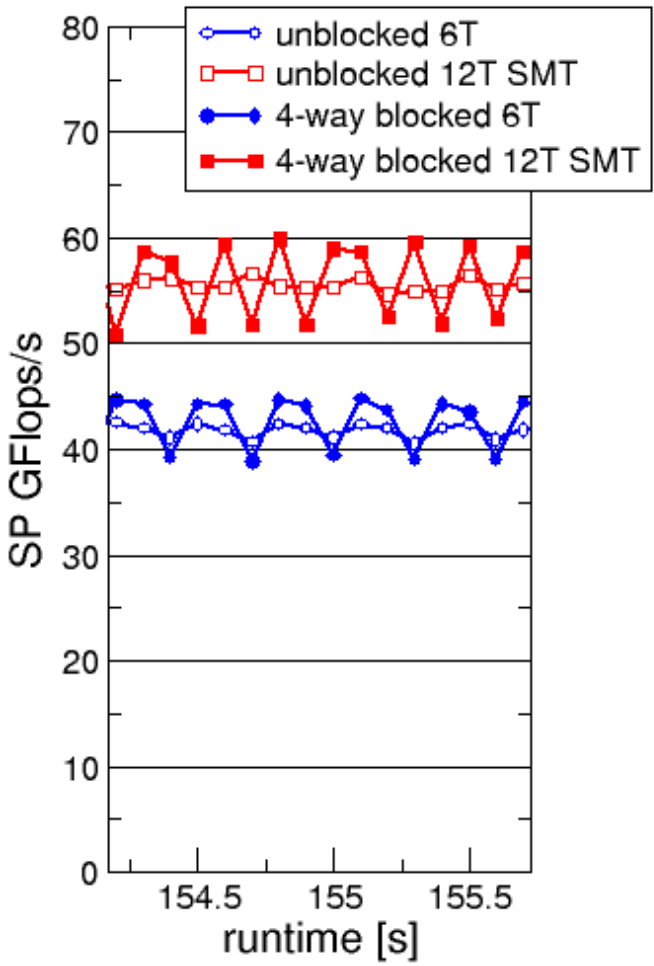
# Patterns (III): Work-related

Pattern		Performance behavior	Metric signature, LIKWID performance group(s)
Load imbalance / serial fraction		Saturating/sub-linear speedup	Different amount of “work” on the cores (FLOPS_*); note that instruction count is not reliable!
Synchronization overhead		Speedup going down as more cores are added / No speedup with small problem sizes / Cores busy but low FP performance	Large non-FP instruction count (growing with number of cores used) / Low CPI (FLOPS_*, CPI)
Instruction overhead		Low application performance, good scaling across cores, performance insensitive to problem size	Low CPI near theoretical limit / Large non-FP instruction count (constant vs. number of cores) (FLOPS_*, DATA, CPI)
Code composition	Expensive instructions	Similar to instruction overhead	Many cycles per instruction (CPI) if the problem is large-latency arithmetic
	Ineffective instructions		Scalar instructions dominating in data-parallel loops (FLOPS_*, CPI)

# Example rabbitCT



# Optimization without knowledge about bottleneck



# Where to start

Look at the code and understand what it is doing!

Scaling runs:

- Scale #cores inside ccNUMA domain
- Scale across ccNUMA domains
- Scale working set size (if possible)

HPM measurements:

- Memory Bandwidth
- Instruction decomposition: Arithmetic, data, branch, other
- SIMD vectorized fraction
- Data volumes inside memory hierarchy
- CPI

# Most frequent patterns (seen with scientific computing glasses)

## Data transfer related:

- Memory bandwidth saturation
- Bad ccNUMA page placement

## Parallelization

- Load imbalance
- Serial fraction

## Code composition:

- Instruction overhead
- Ineffective instructions
- Expensive instructions

## Overhead:

- Synchronization overhead

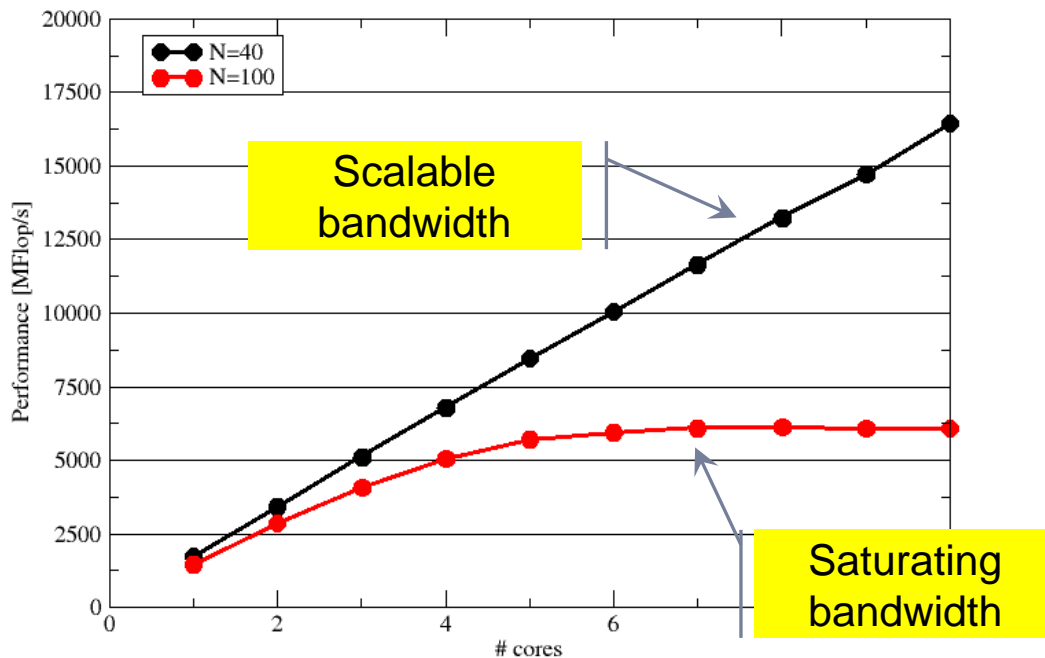
## Excess work:

- Data volume reduction over slow data paths
- Reduction of algorithmic work

# Pattern: Bandwidth Saturation

Always check this first!

1. Perform scaling run inside ccNUMA domain
2. Measure memory bandwidth with HPM
3. Compare to micro benchmark with similar data access pattern



Measured bandwidth spmv:

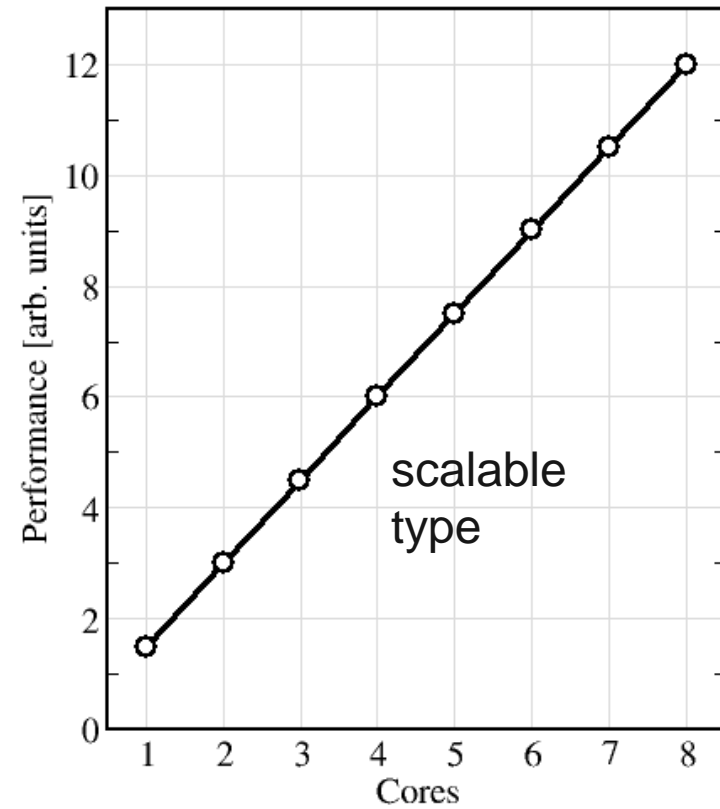
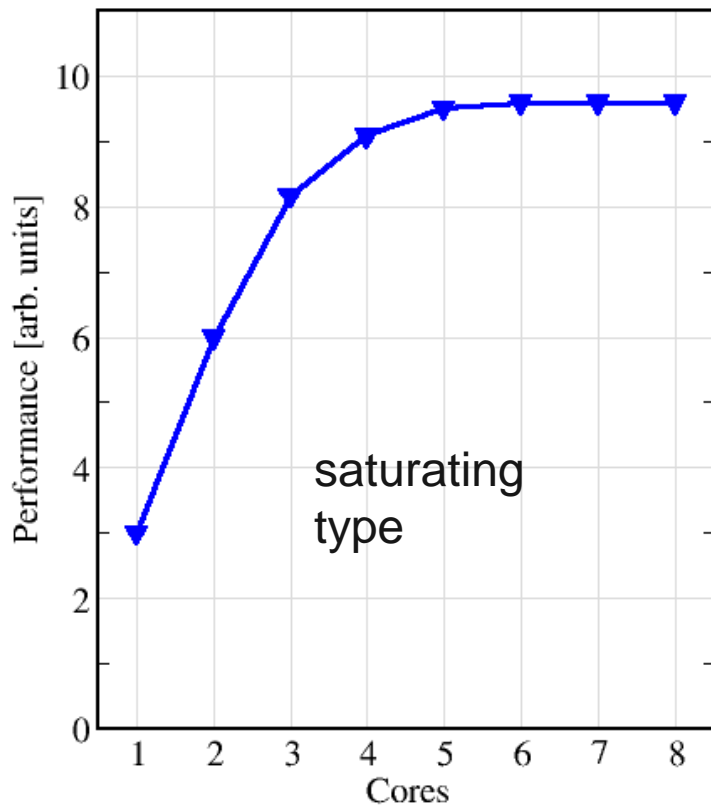
45964 MB/s

Synthetic load benchmark:

47022 MB/s

# Consequences from the saturation pattern

Clearly distinguish between “saturating” and “scalable” performance on the chip level



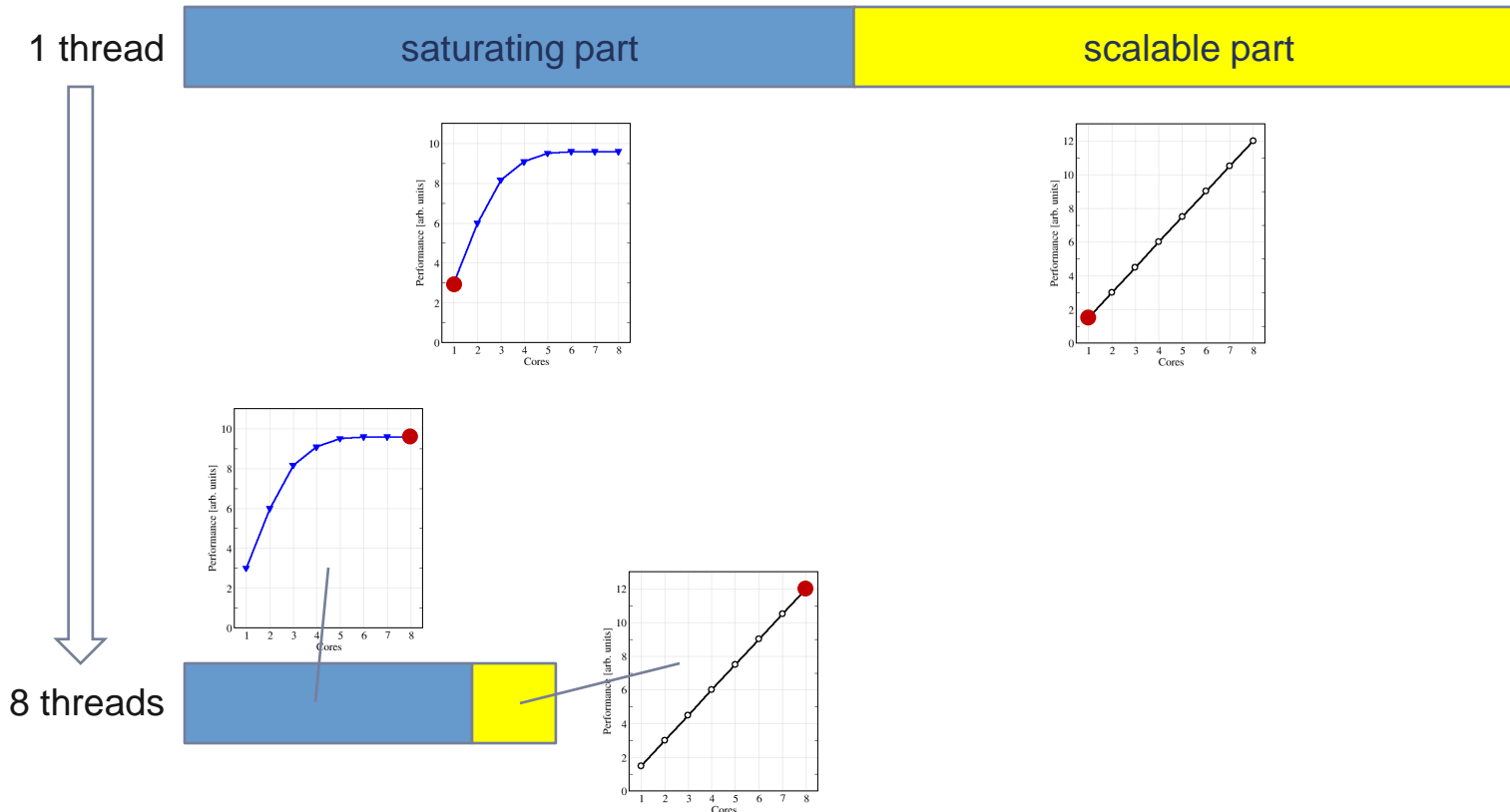
# Consequences from the saturation pattern

There is no clear bottleneck for single-core execution

Code profile for single thread  $\neq$  code profile for multiple threads

→ Single-threaded profiling may be misleading

runtime





# Pattern: Load imbalance

1. Check HPM instruction count distribution across cores
  - 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

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
    
```

# Example for a load balanced code

```
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

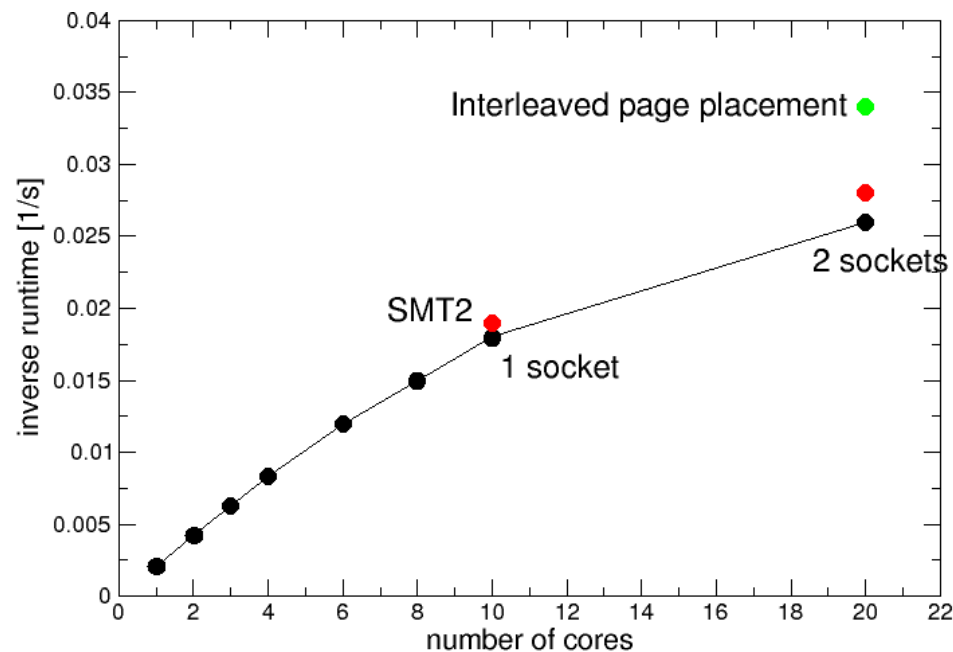
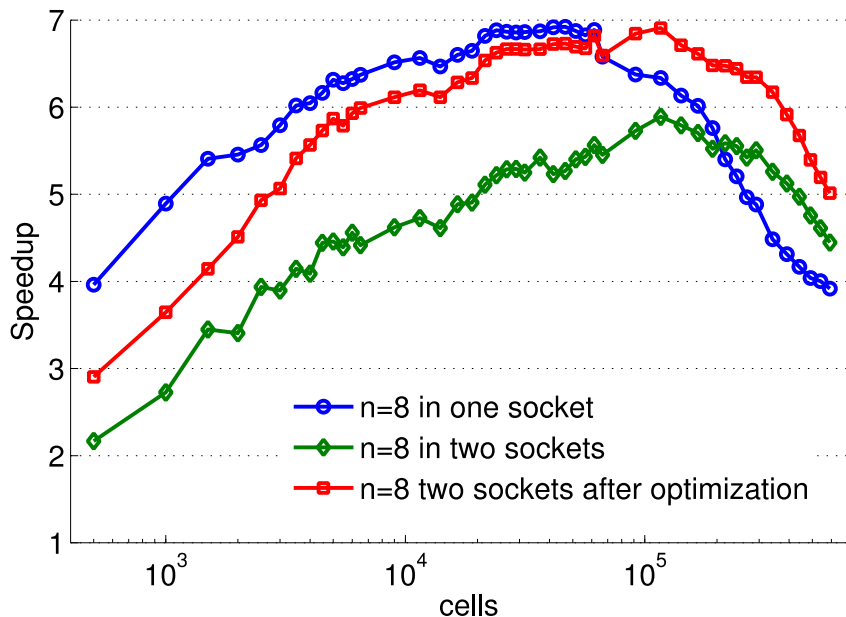
Higher CPI but better performance

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

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

# Pattern: Bad ccNUMA page placement

1. Benchmark scaling across ccNUMA domains
2. Is performance sensitive to interleaved page placement
3. Measure inter-socket traffic with HPM



# Pattern: Instruction Overhead

1. Perform a HPM instruction decomposition analysis
2. Measure resource utilization
3. Static code analysis

Instruction decomposition	Inlining failed	Inefficient data structures
Arithmetic FP	12%	21%
Load/Store	30%	50%
Branch	24%	10%
Other	34%	19%

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
- Low-ish bandwidth
- Low # of floating-point instructions vs. other instructions

# Pattern: Inefficient Instructions

1. HPM measurement: Relation packed vs. scalar instructions
2. Static assembly code analysis: Search for scalar loads

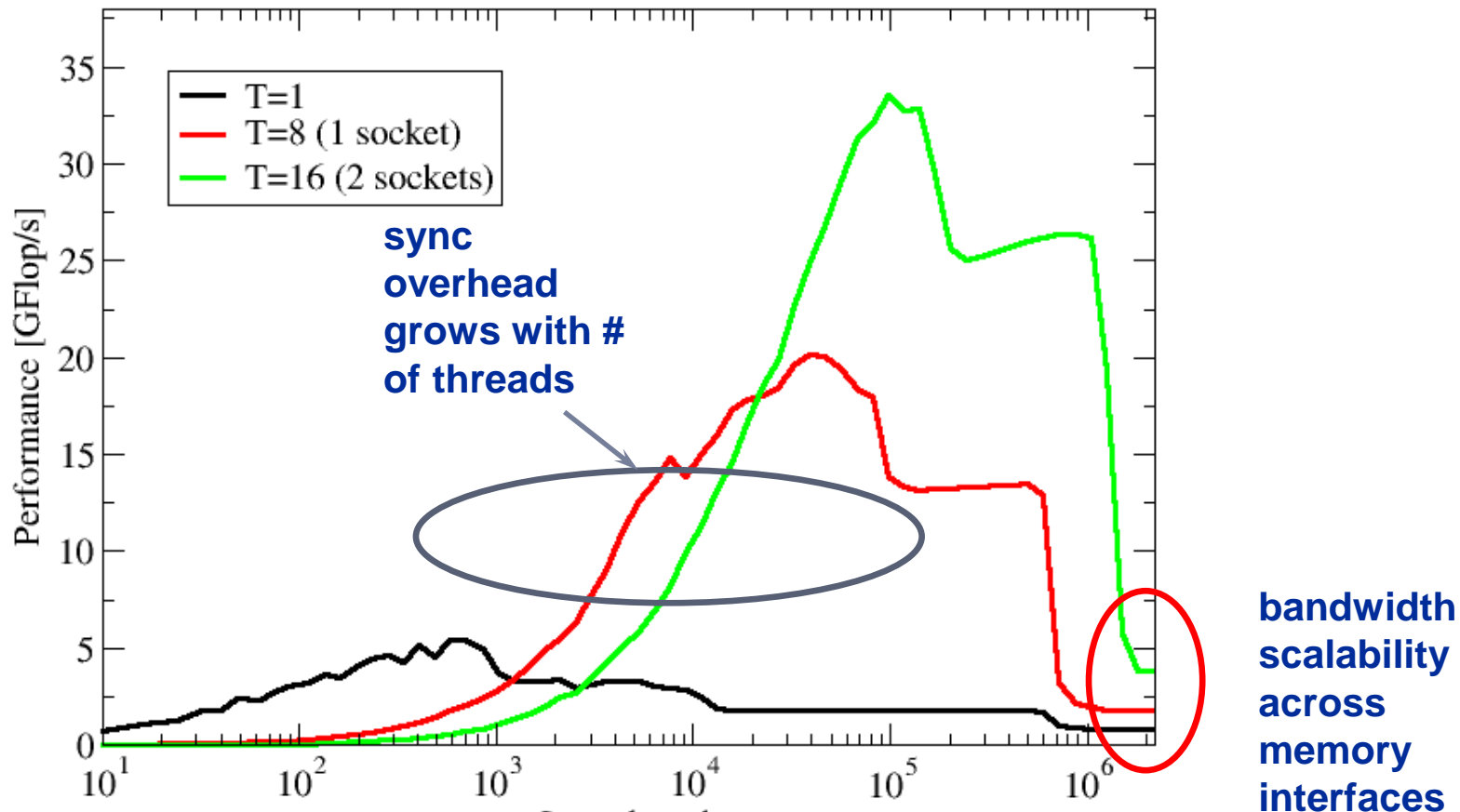
	core 0	core 1	core 2	core 3	
					No AVX
INSTR_RETIRED_ANY	2.19445e+11	1.7674e+11	1.76255e+11	1.75728e+11	1.75578e+11
CPU_CLK_UNHALTED_CORE	1.4396e+11	1.28759e+11	1.28846e+11	1.28898e+11	1.28905e+11
CPU_CLK_UNHALTED_REF	1.20204e+11	1.0895e+11	1.09024e+11	1.09067e+11	1.09074e+11
FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE	1.1169e+09	1.09639e+09	1.09739e+09	1.10112e+09	1.10031e+09
FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE	3.62746e+10	3.45789e+10	3.45446e+10	3.44553e+10	3.44829e+10
SIMD_FP_256_PACKED_DOUBLE	0	0	0	0	0

- There is usually no counter for packed vs scalar (SIMD) loads and stores.
- Also the compiler usually does not distinguish!

Only solution: Inspect code at assembly level.

# Pattern: Synchronization overhead

1. Performance is decreasing with growing core counts
2. Performance is sensitive to topology
3. Static code analysis: Estimate work vs. barrier cost.



# Thread synchronization overhead on SandyBridge-EP

*Barrier overhead in CPU cycles*

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



Gcc not very competitive

Intel compiler



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

# Thread synchronization overhead on AMD Interlagos

*Barrier overhead in CPU cycles*

2 Threads	Cray 8.03	GCC 4.6.2	PGI 11.8	Intel 12.1.3
Shared L2	258	3995	1503	128623
Shared L3	698	2853	1076	128611
Same socket	879	2785	1297	128695
Other socket	<b>940</b>	2740 / 4222	1284 / 1325	<b>128718</b>



Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler 

Full domain	Cray 8.03	GCC 4.6.2	PGI 11.8	Intel 12.1.3
Shared L3	2272	27916	5981	151939
Socket	3783	49947	7479	163561
Node	7663	167646	9526	178892



# Thread synchronization overhead on Intel Xeon Phi

*Barrier overhead in CPU cycles*

2 threads on  
distinct cores:  
1936

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

That does not look bad for 240 threads!

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

3.75 x cores (16 vs 60) on Phi

2 x more operations per cycle on Phi

2.7 x more barrier penalty (cycles) on Phi



7.5 x more work done on Xeon Phi per cycle

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

# SpMV kernel: Data set size and thread count influence on limiting pattern

## Strongly memory-bound for large data sets

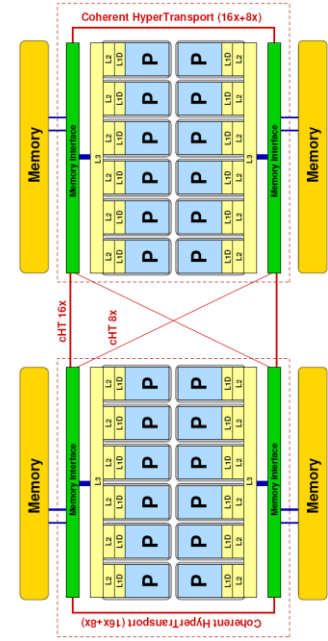
- Streaming, with partially indirect access:

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

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

# Application: Sparse matrix-vector multiply

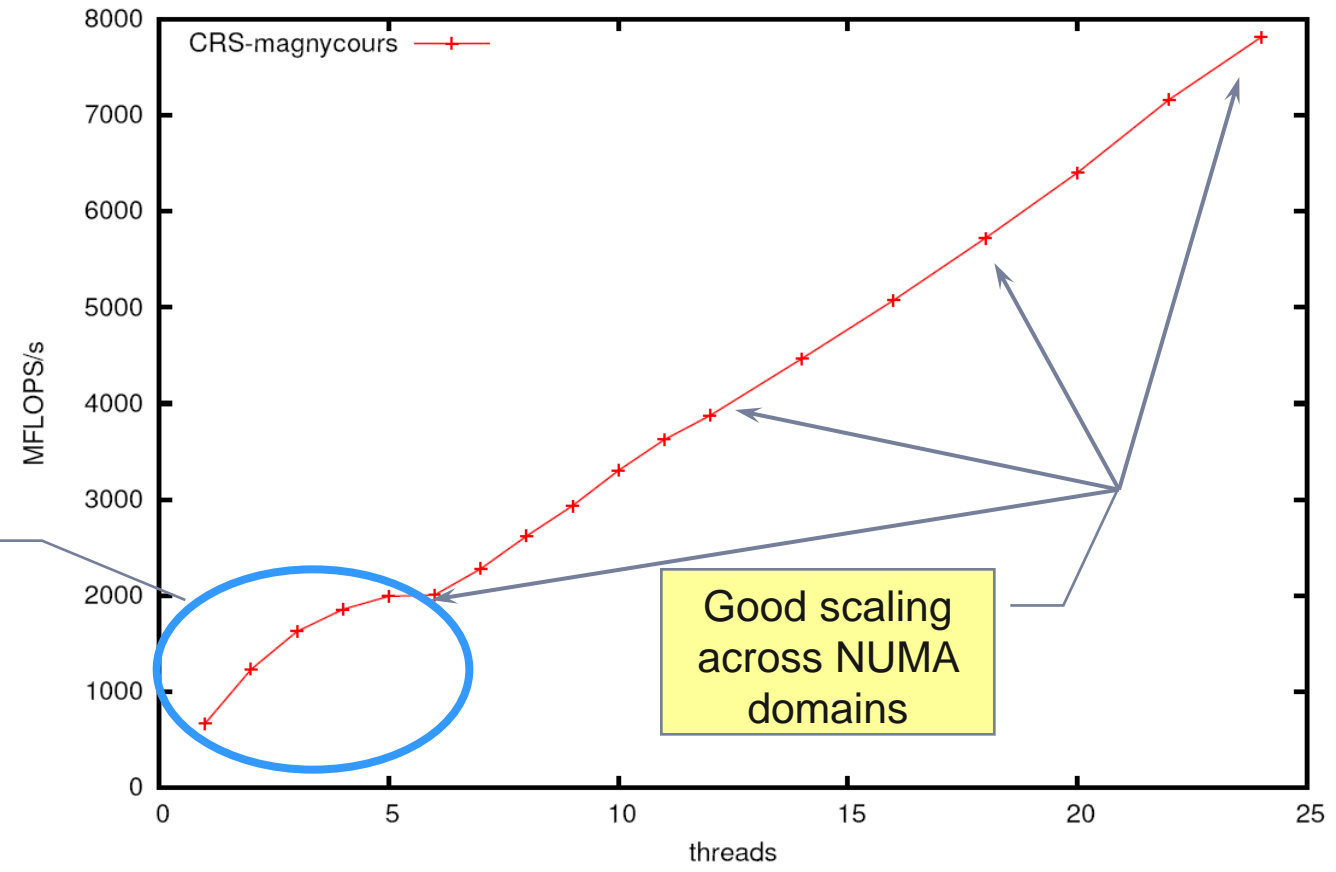
Strong scaling on one XE6 Magny-Cours node



Large matrix

Pattern: Bandwidth saturation

cant, 62451x62451, non-zero: 4007383



# Application: Sparse matrix-vector multiply

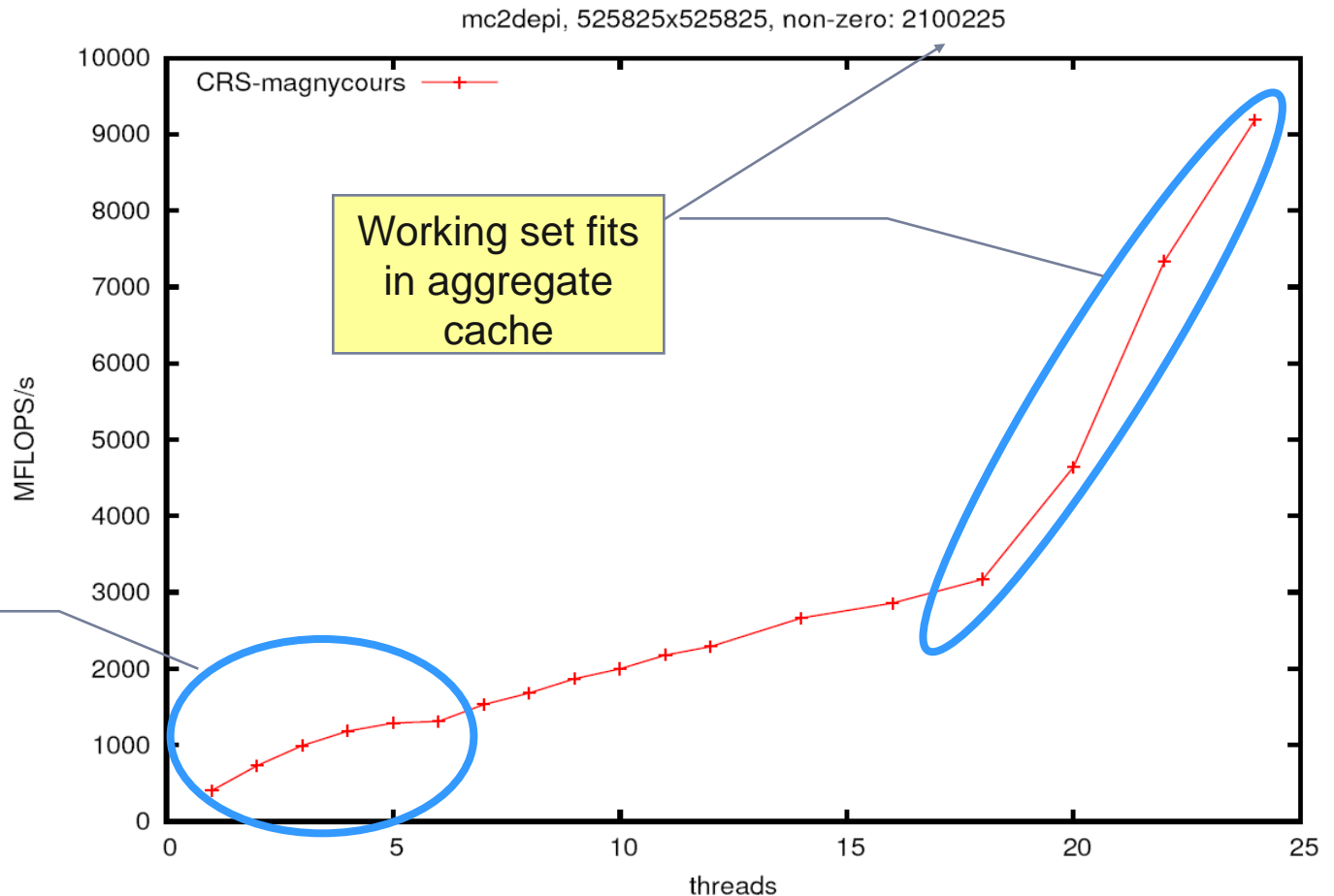
Strong scaling on one XE6 Magny-Cours node



Intrasocket bandwidth bottleneck

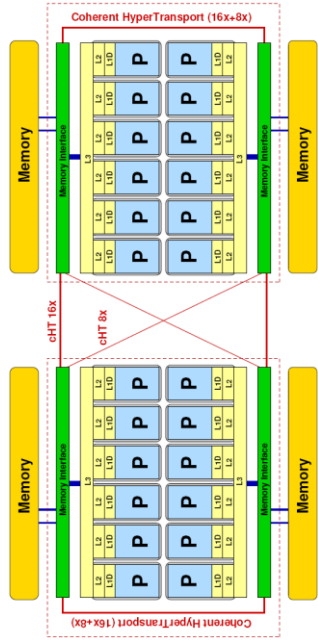
Medium size

Pattern: Work reduction. Less data volume over slow data paths



# Application: Sparse matrix-vector multiply

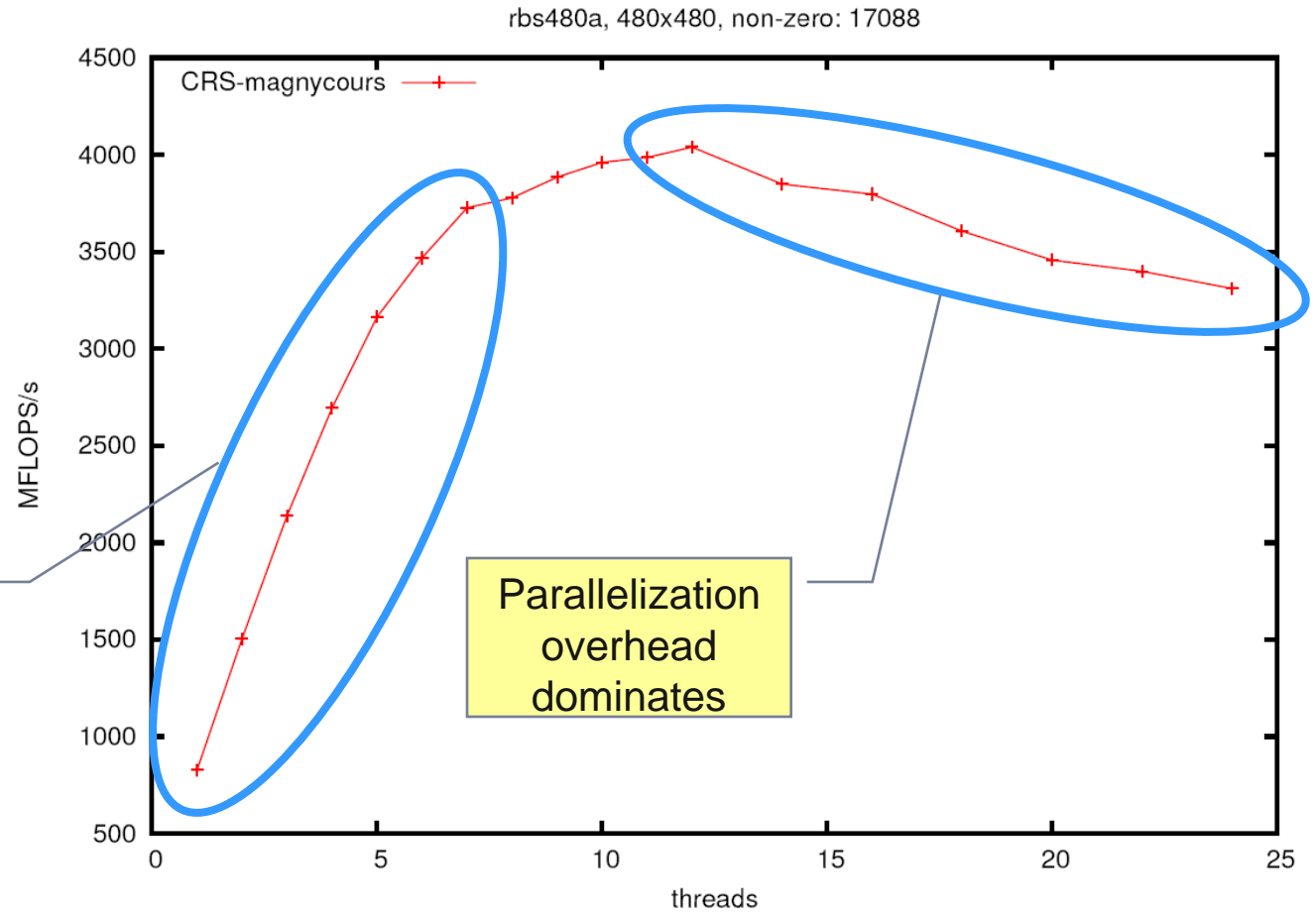
Strong scaling on one Magny-Cours node



No bandwidth bottleneck

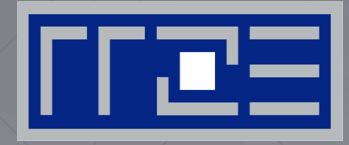
Small size

Pattern: Synchronization overhead





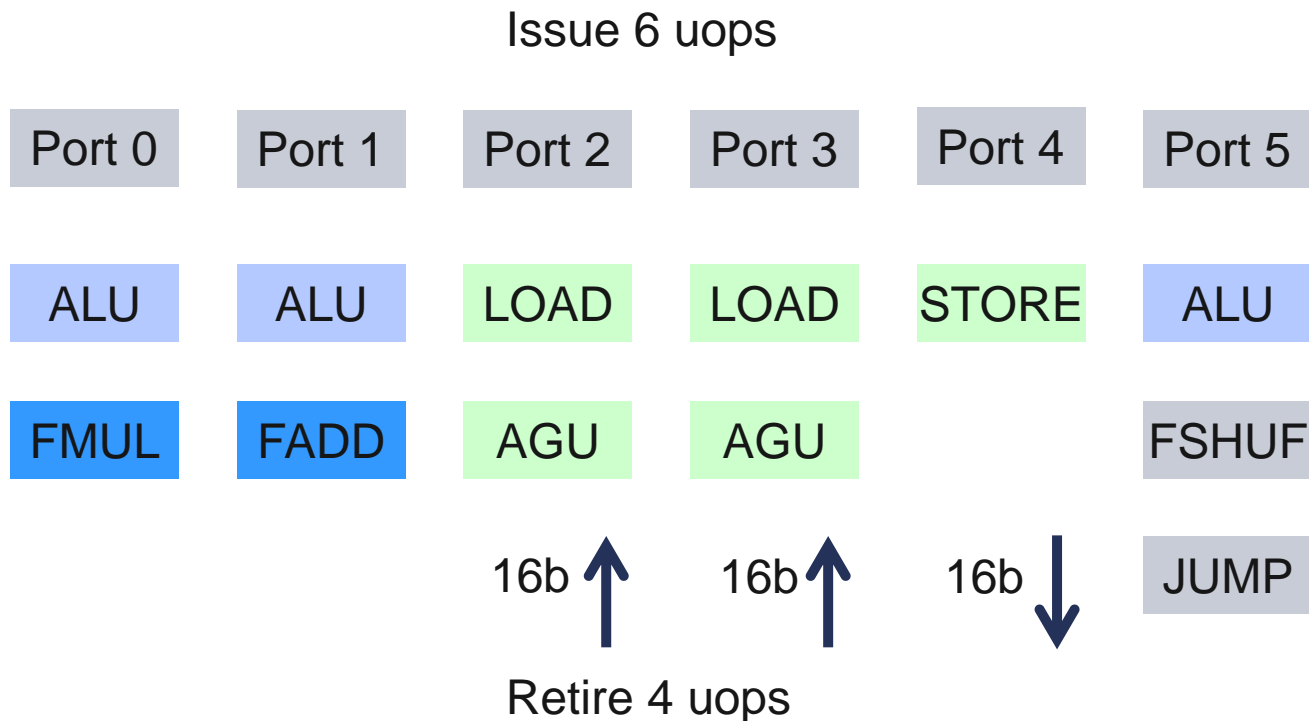
# “SIMPLE” PERFORMANCE MODELING: THE ROOFLINE MODEL



Loop-based performance modeling:  
Execution vs. data transfer

# Preliminary: Estimating Instruction throughput

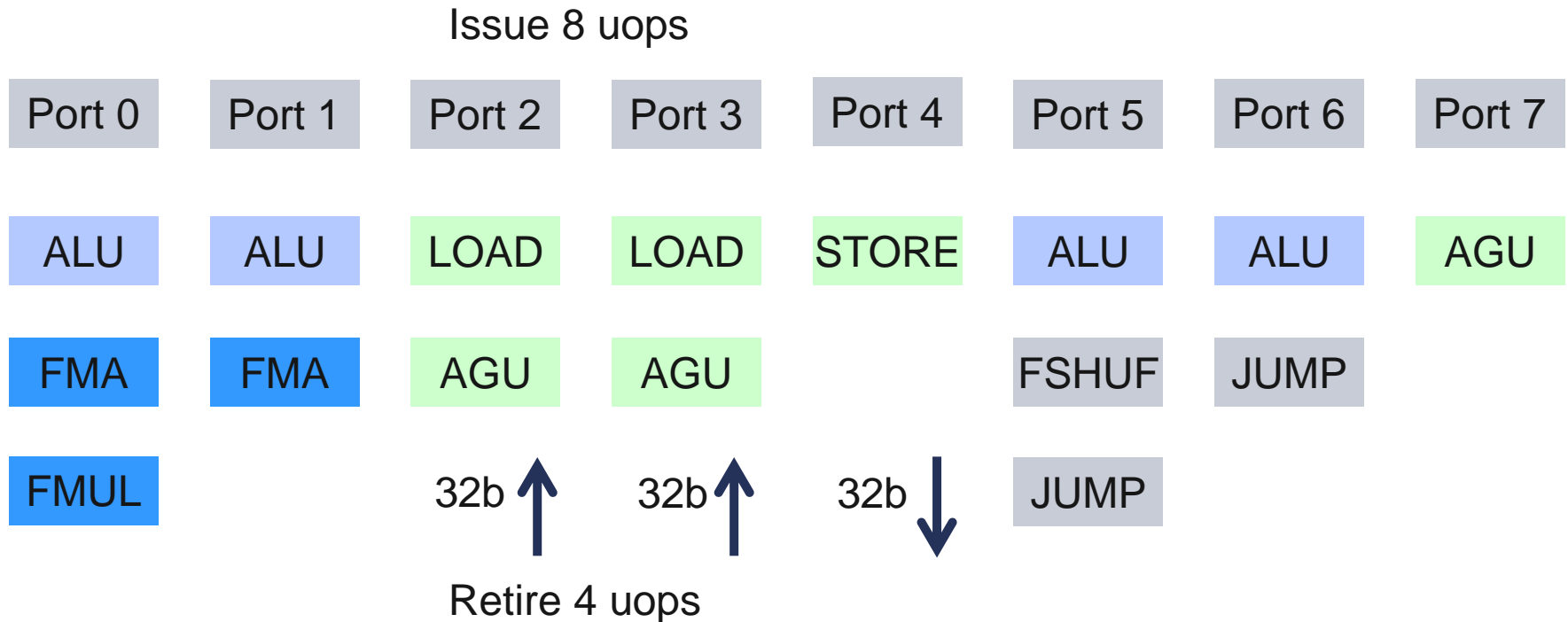
How to perform a instruction throughput analysis on the example of Intel's port based scheduler model.



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.



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  $\frac{1}{2}$  store and mult and add

Cycle 2: load and  $\frac{1}{2}$  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  $4 \times 2 = 8$  flops.

$(3 \text{ GHz} / 3 \text{ cycles}) * 4 \text{ updates} * 2 \text{ flops/update} = \mathbf{8 \text{ GFlops/s}}$

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

# The Roofline Model<sup>1,2</sup>

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

Expected performance:

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

The diagram shows the equation  $P = \min(P_{\max}, I b_S)$ . Above the term  $P_{\max}$  is a yellow box containing the units  $[F/B]$ . Above the term  $b_S$  is a yellow box containing the units  $[B/s]$ . Lines connect these boxes to their respective terms in the equation.

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

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

# “Simple” Roofline: The vector triad

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

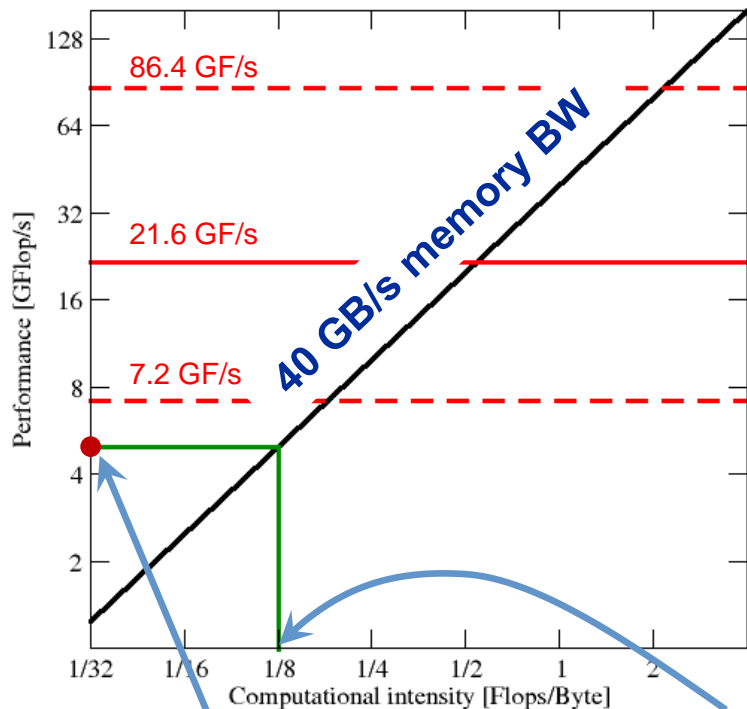
- $b_S = 40 \text{ GB/s}$
- $B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F}$  (including write allocate)
  - $I = 0.4 \text{ F/W} = 0.05 \text{ F/B}$
  - $I \cdot b_S = 2.0 \text{ GF/s}$  (1.2 % of peak performance)
- $P_{\text{peak}} = 173 \text{ GFlop/s}$  (8 FP units x (4+4) Flops/cy x 2.7 GHz)
- $P_{\text{max}}?$  → Observe LD/ST throughput maximum of 1 AVX Load and ½ AVX store per cycle → 3 cy / 8 Flops
  - $P_{\text{max}} = 57.6 \text{ GFlop/s}$  (33% peak)

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

# A not so simple Roofline example

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

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



**P = 5 Gflop/s**

**I = 1 Flop / 8 byte (in DP)**

ADD peak  
(best possible code)

no SIMD

3-cycle latency per ADD  
if not unrolled

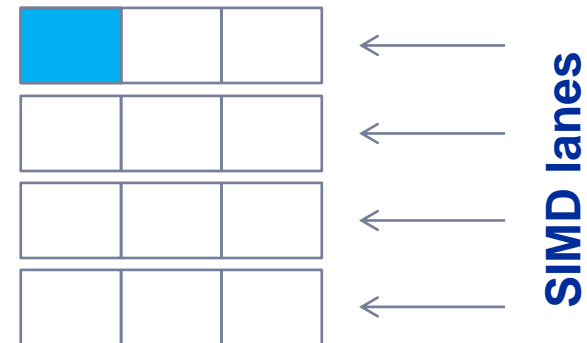
How do we get  
these?  
→ See next!

# 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:



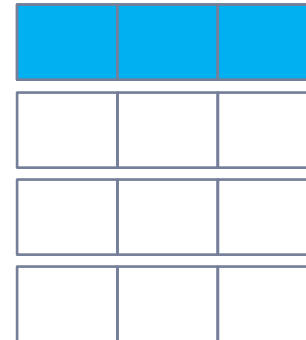
→ 1/12 of ADD peak

# 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
```

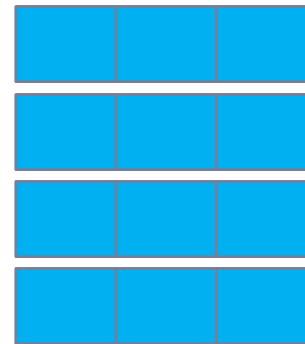
```
ADD r2 ← r2+r5
```

```
ADD r3 ← r3+r6
```

```
i+=12 →? loop
```

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

ADD pipes utilization:



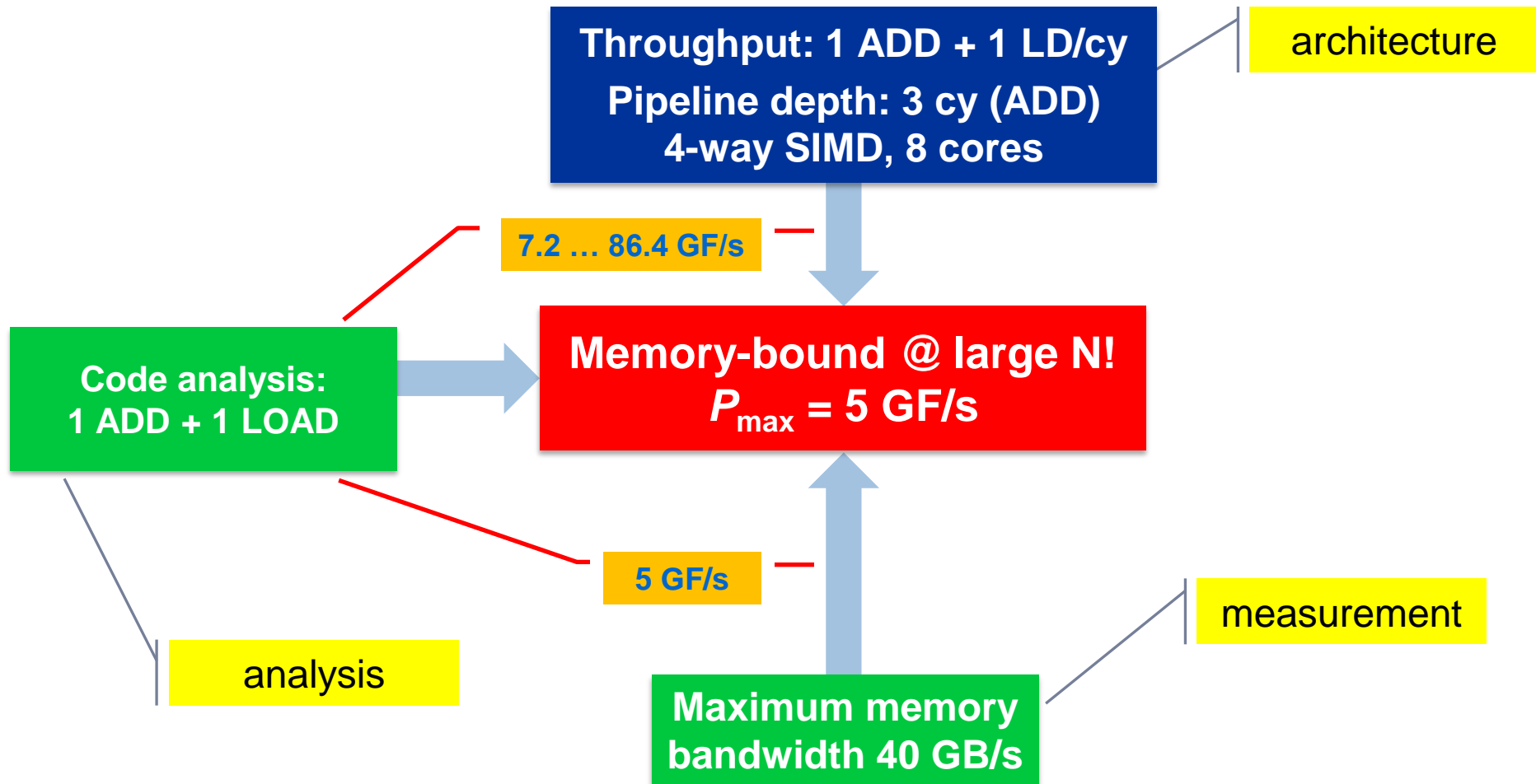
→ ADD peak



# Input to the roofline model

... on the example of

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



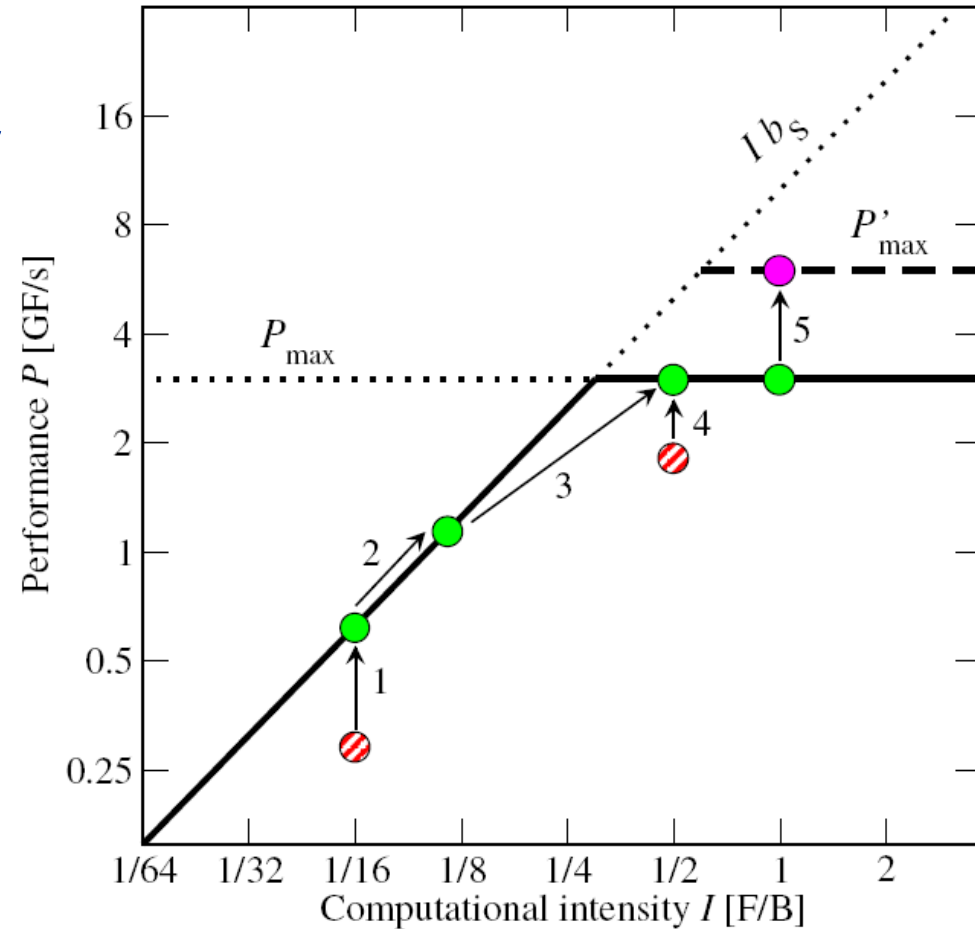
# Assumptions for the Roofline Model

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**

# Typical code optimizations in the Roofline Model

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

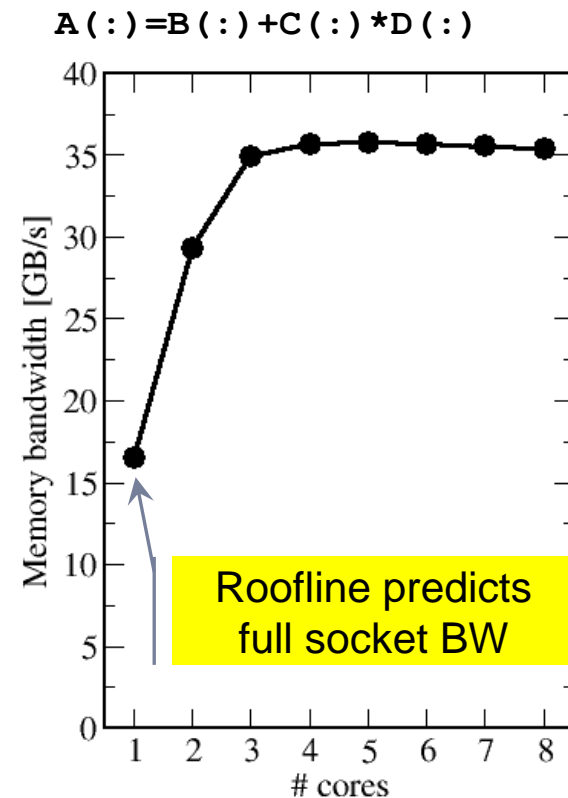


# Shortcomings of the roofline model

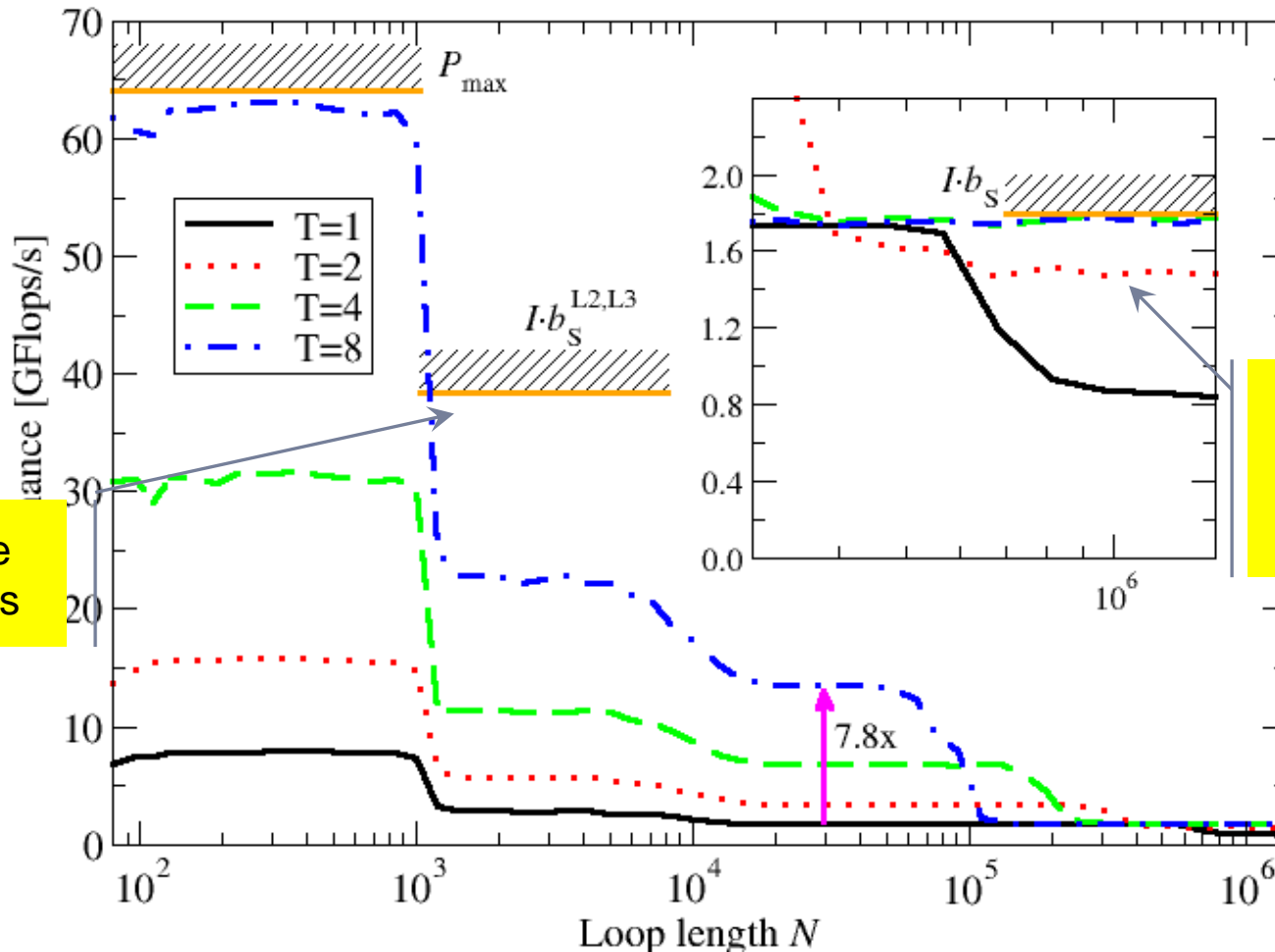
**Saturation effects** in multicore chips are not explained

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

**ECM model** gives more insight



# Where the roofline model fails



In cache situations

In memory performance below saturation point

# 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

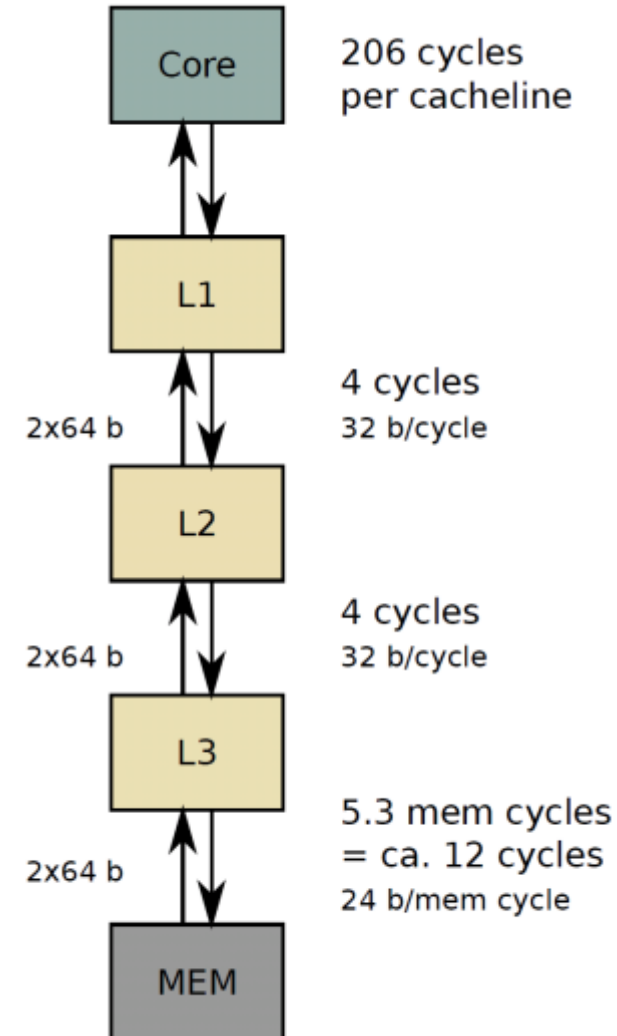
Data transfers may or may not overlap with each other or with in-core execution

Scaling is linear until the relevant bottleneck is reached

## Input:

Same as for Roofline

+ **data transfer times** in hierarchy



# Introduction to ECM model

ECM = “Execution-Cache-Memory”

- Analytical performance model
- Focus on resource utilization
  - Instruction Execution
  - Data Movement
- Lightspeed assumption:
  - Optimal instruction throughput
  - Always bandwidth bound

The RULES™

1. Single-core execution time is composed of
  1. In-core execution
  2. Data transfers in the memory hierarchy
2. All timings are in units of one CL
3. LOADS in the L1 cache do not overlap with any other data transfer
4. Scaling across cores is linear until a shared bottleneck is hit

# Vector dot product: Code characteristics

```
double sum = 0.0;

for (int i=0; i<N; i++){
    sum += a[i]*b[i];
}
```

Naive

```
double sum = 0.0;
double c = 0.0;

for (int i=0; i<N; i++) {
    double prod = a[i]*b[i];
    double y = prod-c;
    double t = sum+y;
    c = (t-sum)-y;
    sum = t;
}
```

Kahan

	naive	kahan
loads	2	2
mul	1	1
add	1	4

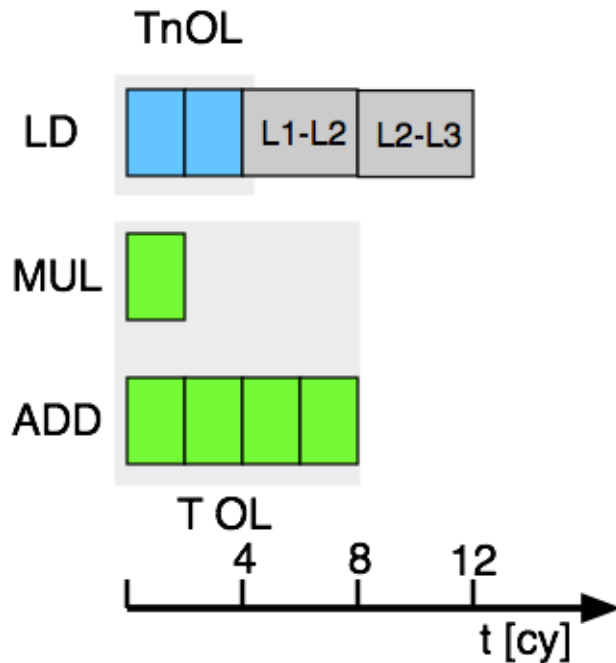
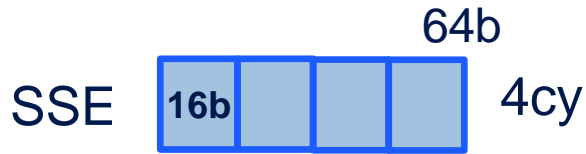


# Machine Model

	SandyBridge-EP	IvyBridge-EP	Haswell-EP
Type	Xeon E5-2680	Xeon E5-2690 v2	Xeon E5-2695 v3
# cores	8 cores @ 2.7GHz	10 cores @ 3.0GHz	14 cores @ 2.3GHz
Load / Store	2 L + 1 S per cy	2 L + 1 S per cy	2 L + 1 S per cy
L1 Port Width	16b	16b	32b
Add	1 per cy	1 per cy	1 per cy
Mul	1 per cy	1 per cy	2 per cy
FMA	n/a	n/a	2 per cy
SIMD width	32b	32b	32b

	SandyBridge-EP	IvyBridge-EP	Haswell-EP
L1 – L2	32b/cy 2cy/CL	32b/cy 2cy/CL	64b/cy 1cy/CL
L2 – L3	32b/cy 2cy/CL	32b/cy 2cy/CL	32b/cy 2cy/CL
L3 - MEM	4.0cy/CL	3.5cy/CL	2.5cy/CL

# Example Kahan (AVX) on IvyBridge-EP



Shorthand notation:

$$T_{core} = \max(T_{nOL}, T_{OL})$$

$$T_{ECM} = \max(T_{nOL} + T_{data}, T_{OL})$$

Contributions:

$$\{T_{OL} \parallel T_{nOL} \mid T_{L1/L2} \mid T_{L2/L3} \mid T_{L3/MEM}\}$$

Kahan (AVX)  $\{8 \parallel 4 \mid 4 \mid 4\}cy$

Prediction  $\{8 \setminus 8 \setminus 12\}cy$

# ECM Model IvyBridge-EP

## Model

Naïve (AVX):  $\{4 \parallel 4 \mid 4 \mid 4 \mid 7\}cy$        $\{4 \setminus 8 \setminus 12 \setminus 19\}cy$   
Kahan (scalar):  $\{32 \parallel 8 \mid 4 \mid 4 \mid 7\}cy$        $\{32 \setminus 32 \setminus 32 \setminus 32\}cy$   
Kahan (AVX):  $\{8 \parallel 4 \mid 4 \mid 4 \mid 7\}cy$        $\{8 \setminus 8 \setminus 12 \setminus 19\}cy$

## Measurement

Naïve (AVX): 4.1 \ 8.7 \ 13.0 \ 24.9cy  
Kahan (scalar): 32.5 \ 32.4 \ 3248 \ 37.9cy  
Kahan (AVX): 8.4 \ 10.2 \ 13.7 \ 23.8cy

# Multicore scaling in the ECM model

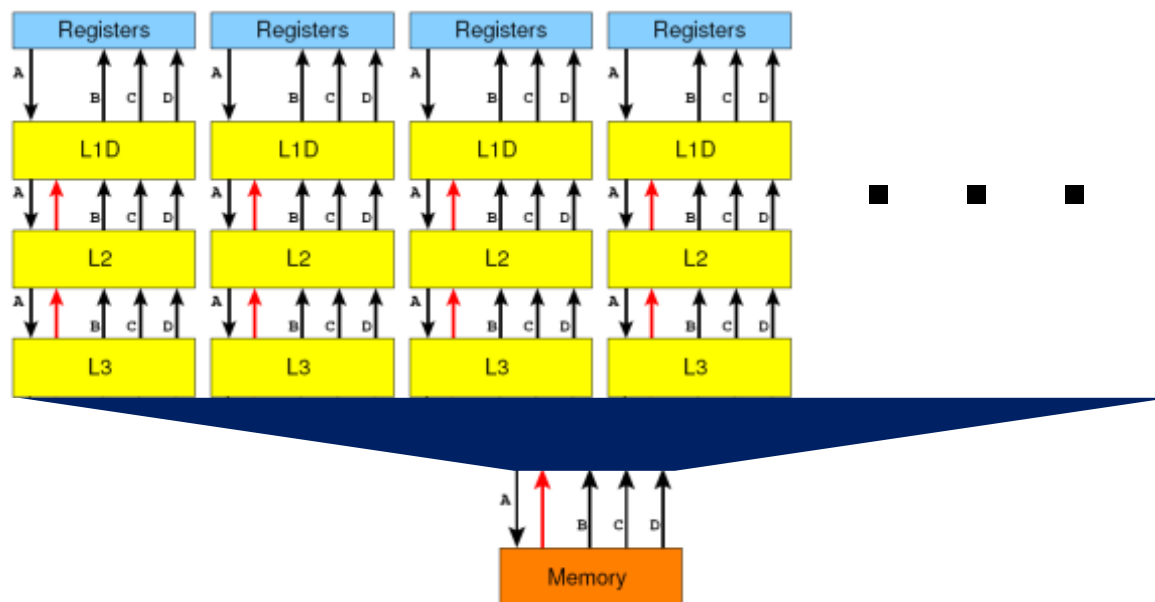
Identify relevant **bandwidth bottlenecks**

- L3 cache
- Memory interface

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

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

Example:  
Scalable L3  
on Sandy  
Bridge



# ECM Model IvyBridge-EP

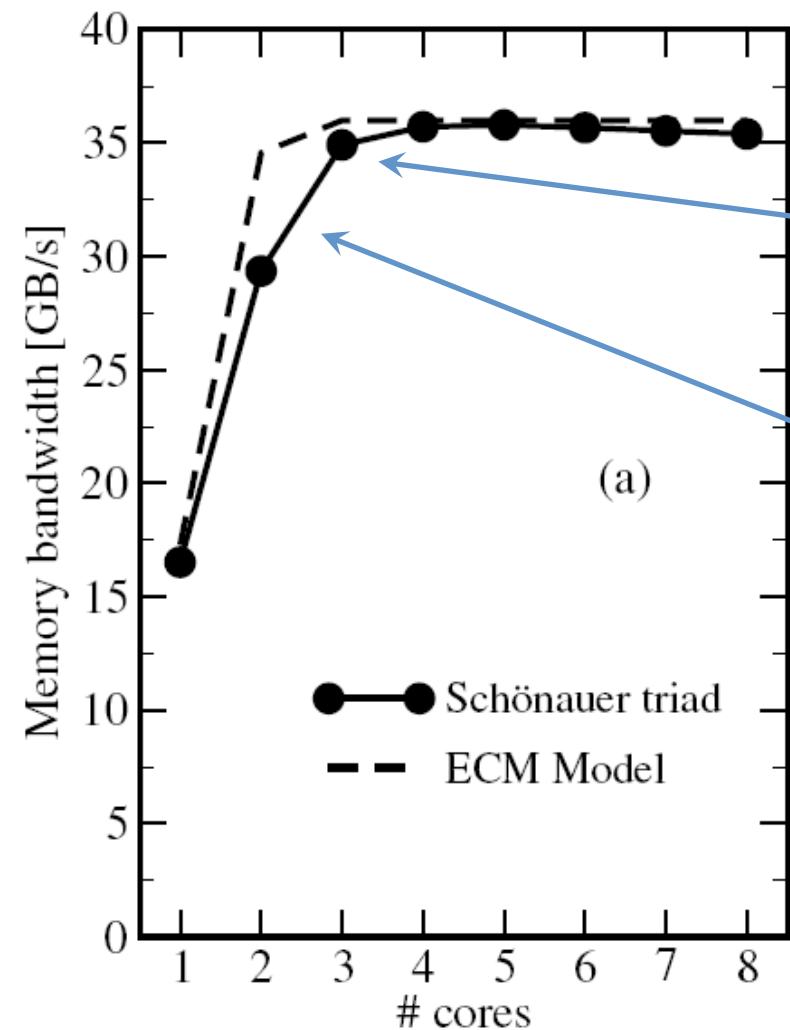
## Model

Naïve (AVX):  $\{4 \parallel 4 \mid 4 \mid 4 \mid 7\}cy$        $\{4 \setminus 8 \setminus 12 \setminus 19\}cy$   
Kahan (scalar):  $\{32 \parallel 8 \mid 4 \mid 4 \mid 7\}cy$        $\{32 \setminus 32 \setminus 32 \setminus 32\}cy$   
Kahan (AVX):  $\{8 \parallel 4 \mid 4 \mid 4 \mid 7\}cy$        $\{8 \setminus 8 \setminus 12 \setminus 19\}cy$

## Measurement

Naïve (AVX): 4.1 \ 8.7 \ 13.0 \ 24.9cy  
Kahan (scalar): 32.5 \ 32.4 \ 3248 \ 37.9cy  
Kahan (AVX): 8.4 \ 10.2 \ 13.7 \ 23.8cy

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



Model: Scales until saturation sets in

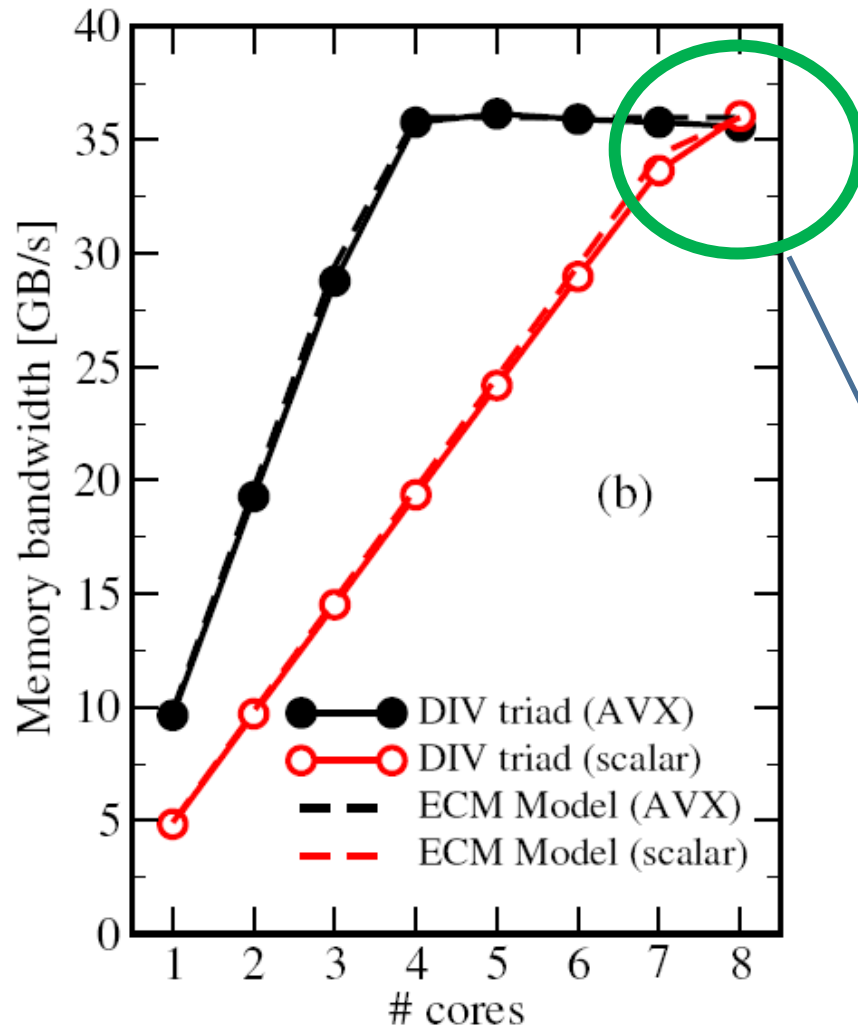
Saturation point (# cores) well predicted

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!

# 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
401d0d:  48 83 c0 01
401d11:  39 c7
401d13:  77 f3
```

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

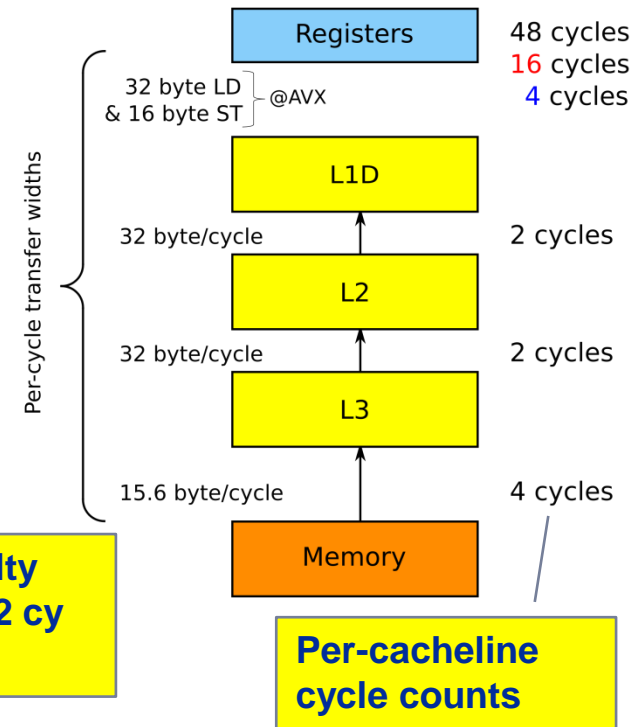
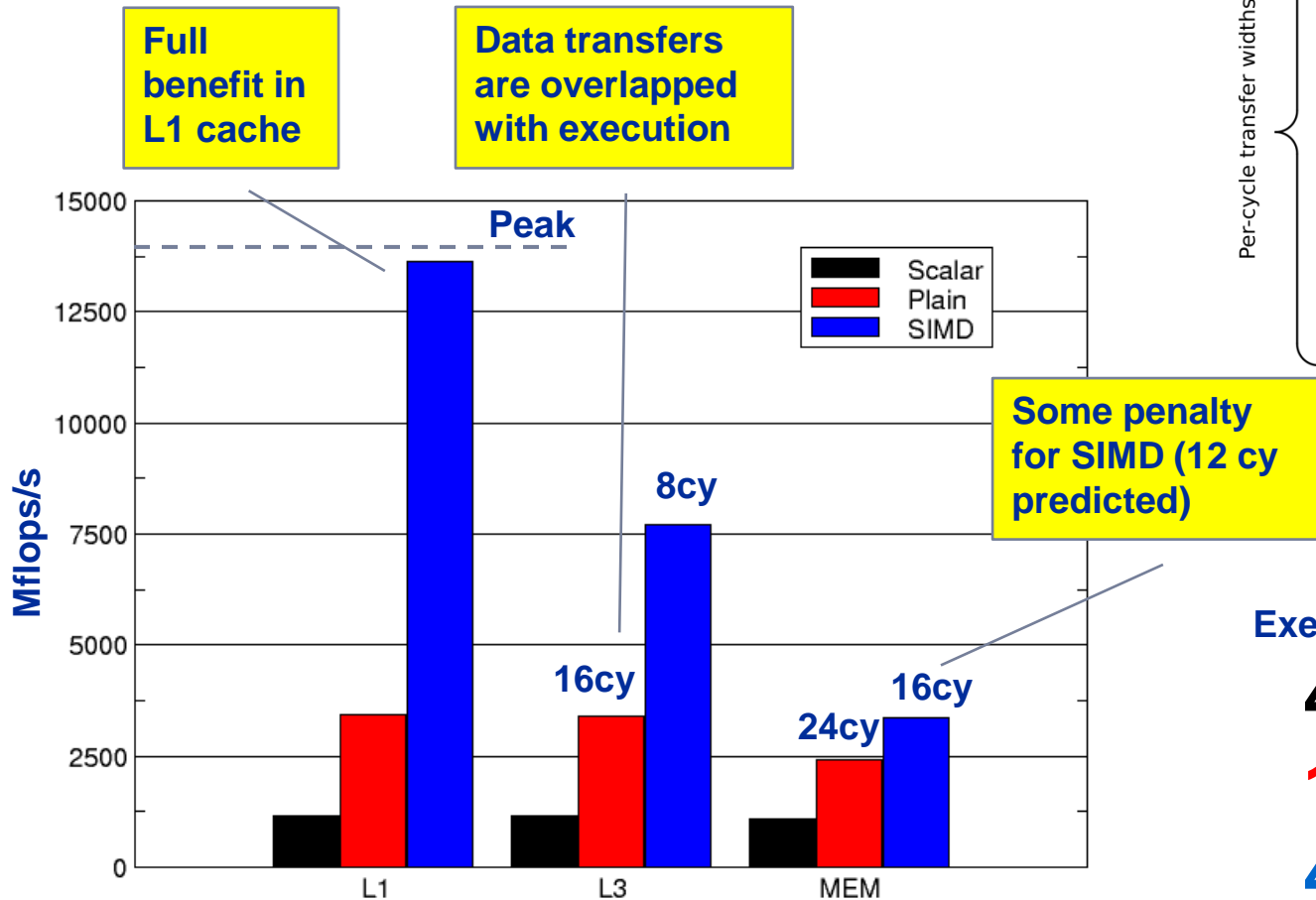
```
1:  
addss xmm0, [rsi + rax * 4]  
addss xmm1, [rsi + rax * 4 + 4]  
addss xmm2, [rsi + rax * 4 + 8]  
addss xmm3, [rsi + rax * 4 + 12]  
add    rax, 4  
cmp    eax,edi  
js 1b
```

```
1:  
addps xmm0, [rsi + rax * 4]  
addps xmm1, [rsi + rax * 4 + 16]  
addps xmm2, [rsi + rax * 4 + 32]  
addps xmm3, [rsi + rax * 4 + 48]  
add    rax, 16  
cmp    eax,edi  
js 1b
```

SSE SIMD vectorization

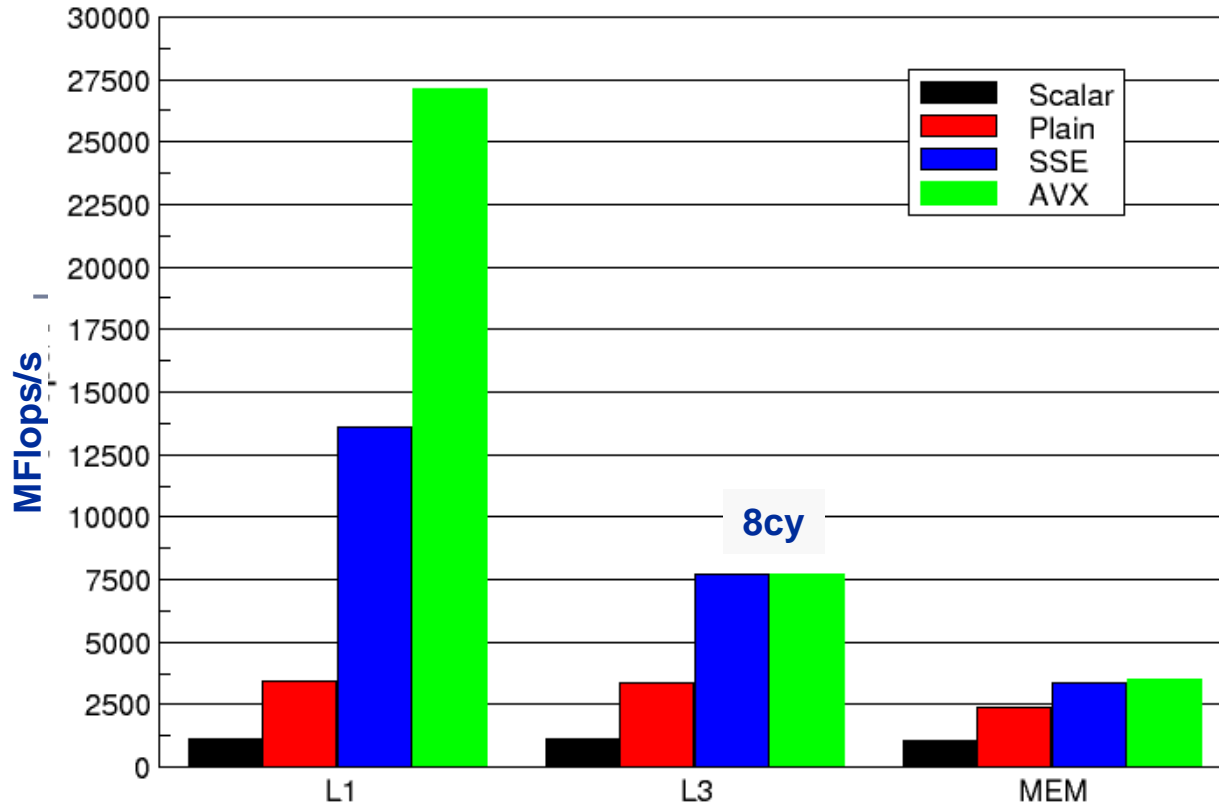
# SIMD processing – single-threaded

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



Execution	Cache	Memory
48		
16	4	4
4		

# And with AVX?



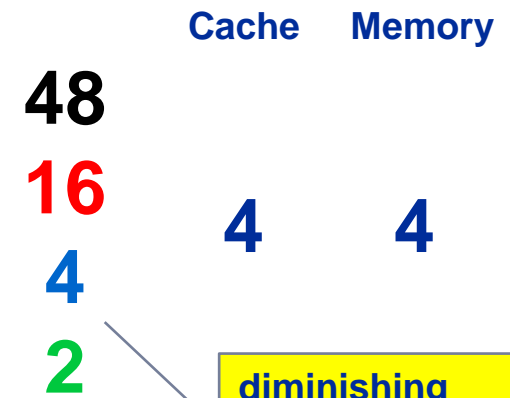
**L3 Cache**

**SSE** 8 cycles

**AVX** 6 cycles

**With preloading:**

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



# SIMD processing – Full chip (all cores)

## Influence of SMT

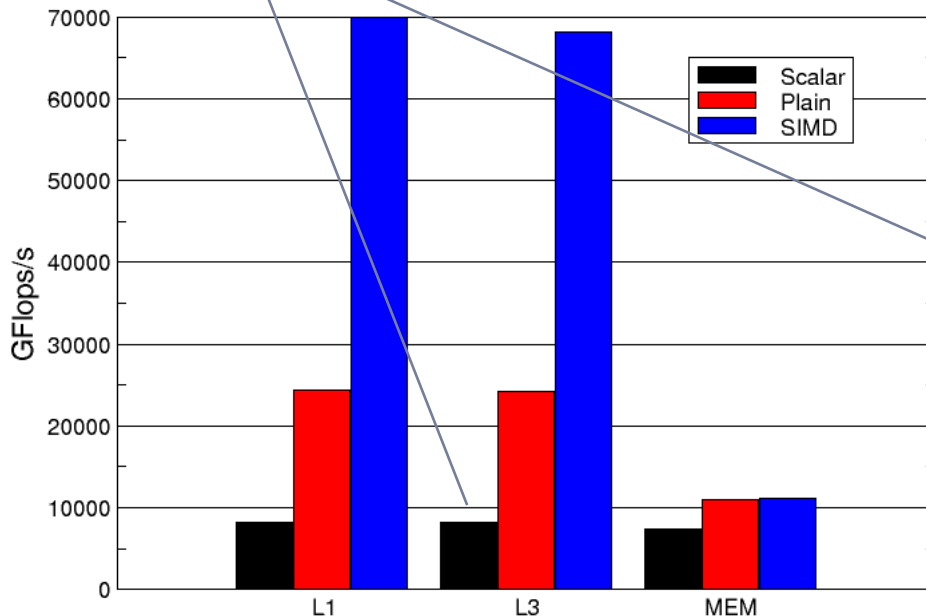
Bandwidth saturation is the primary performance limitation on the chip level!

Full scaling using SMT due to bubbles in pipeline

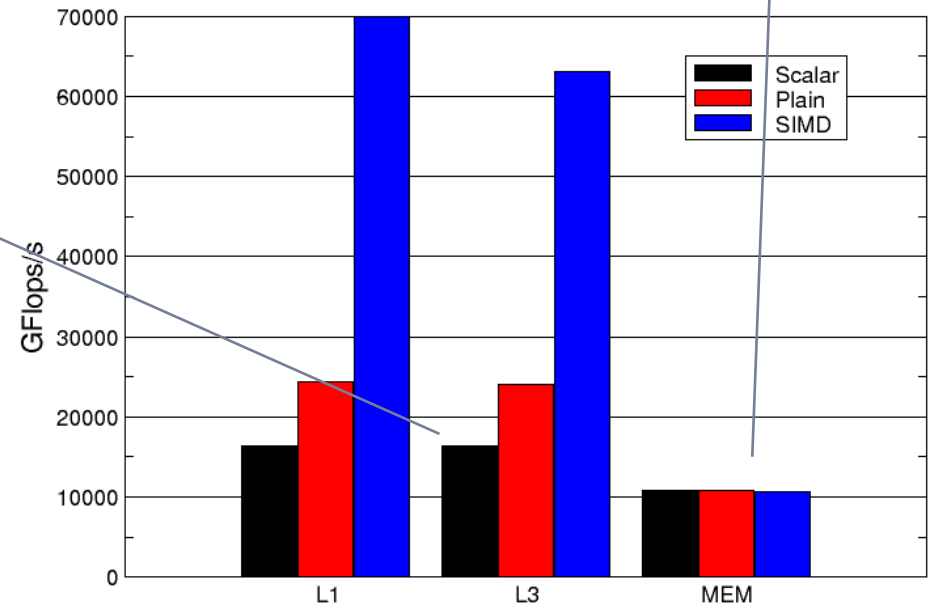
**Conclusion:** If the code saturates the bottleneck, all variants are acceptable!

All variants saturate the memory bandwidth

8 threads on physical cores



16 threads using SMT



# Summary: The ECM Model

- The ECM model is a simple analysis tool to get insight into:
  - Runtime contributions
  - Bottleneck identification
  - Runtime overlap

It can predict single core performance for any memory hierarchy level and get an estimate of multicore chip scalability.

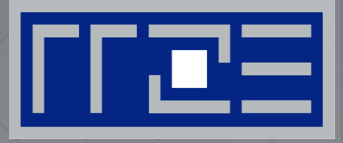
## **ECM correctly describes several effects**

- Saturation for memory-bound loops
- Diminishing returns of in-core optimizations for far-away data

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



# 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

$2 * 8b L + 1 * 8b S = 24b$

$2.2GHz/4c * 24flops =$

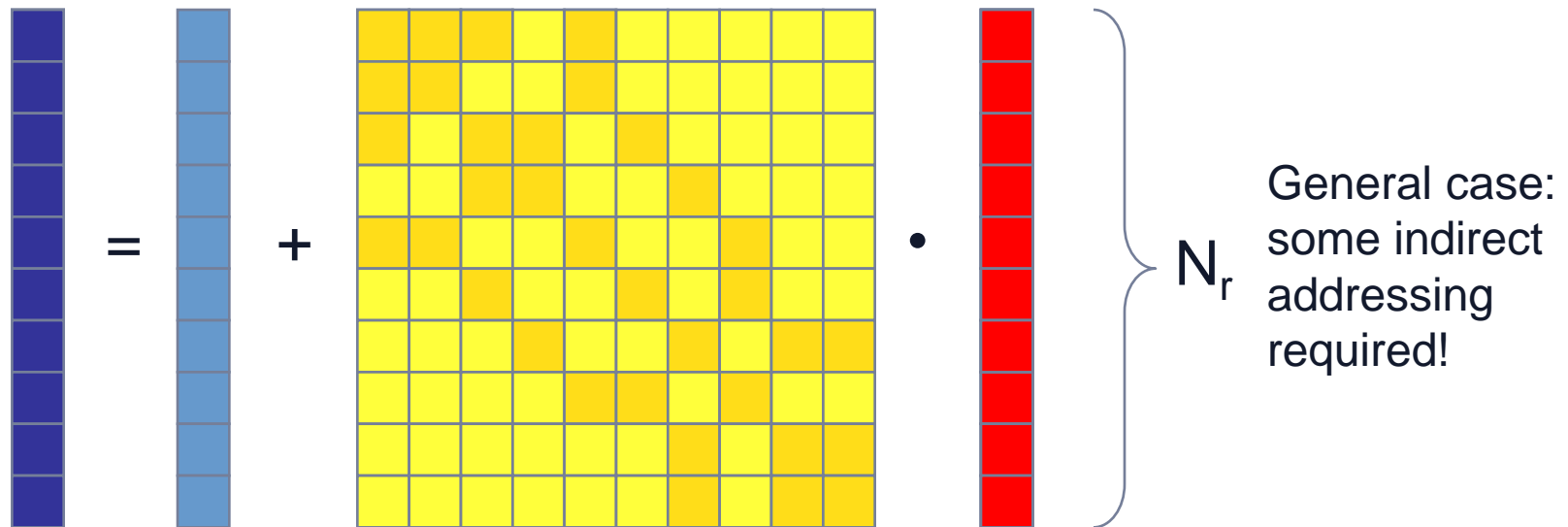
13.2 GFlops/s or

106GB/s L1 or 47GB/s L2

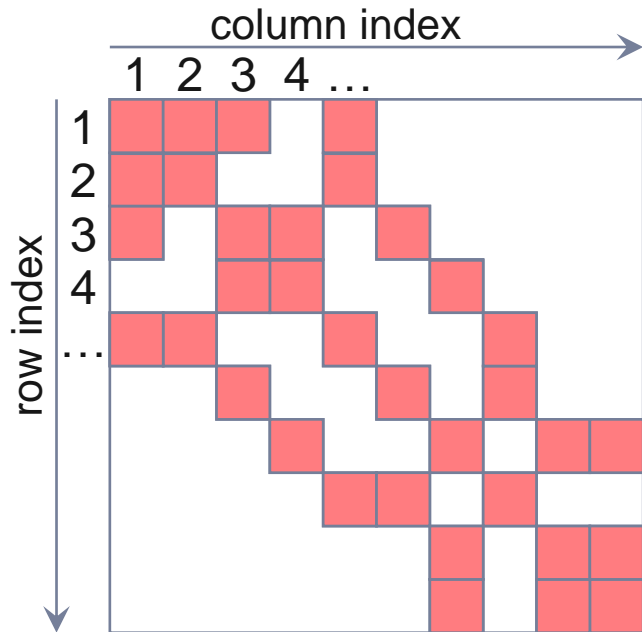


# Sparse matrix-vector multiply (spMVM)

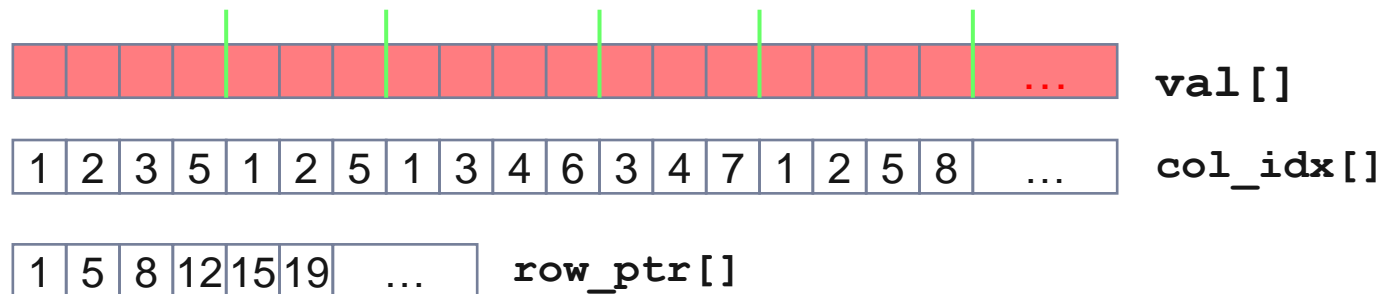
- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- **Store only  $N_{nz}$  nonzero elements** of matrix and RHS, LHS vectors with  $N_r$  (number of matrix rows) entries
- **“Sparse”**:  $N_{nz} \sim N_r$



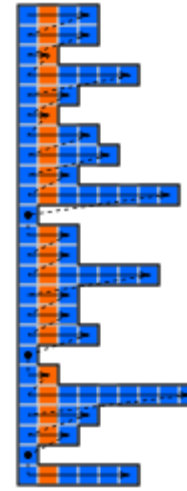
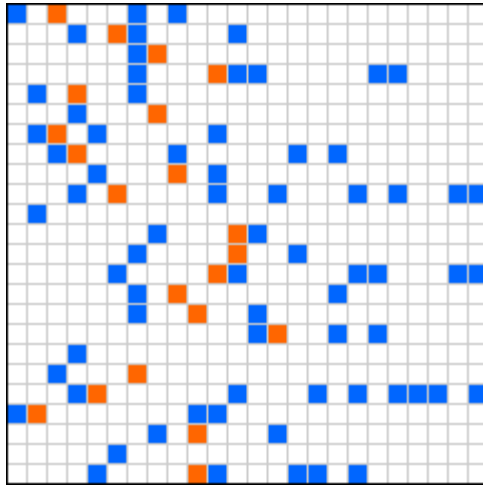
# CRS matrix storage scheme



- **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:  $N_r$ )



# 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<sup>3</sup>)

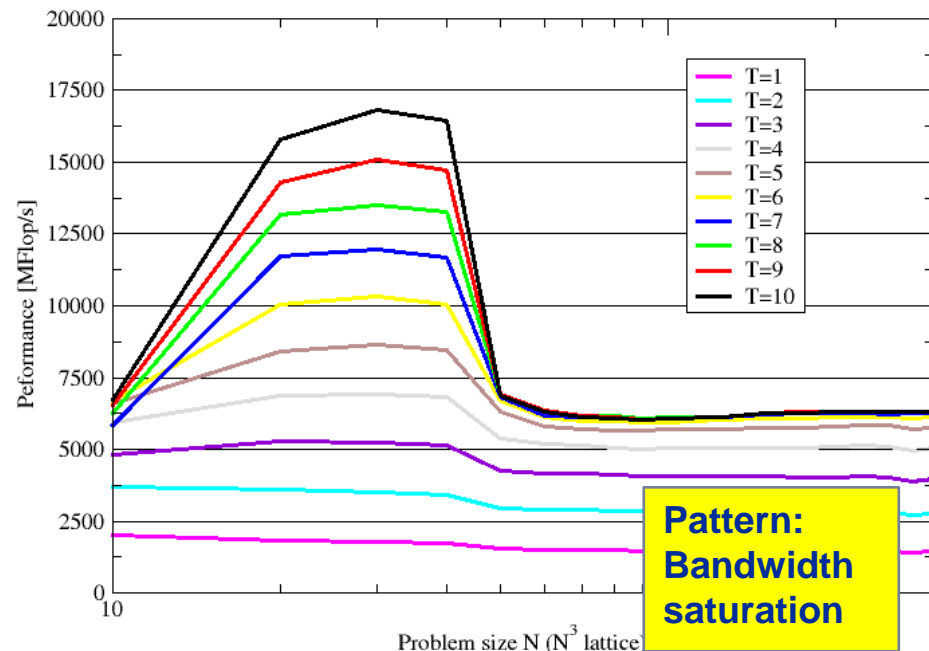
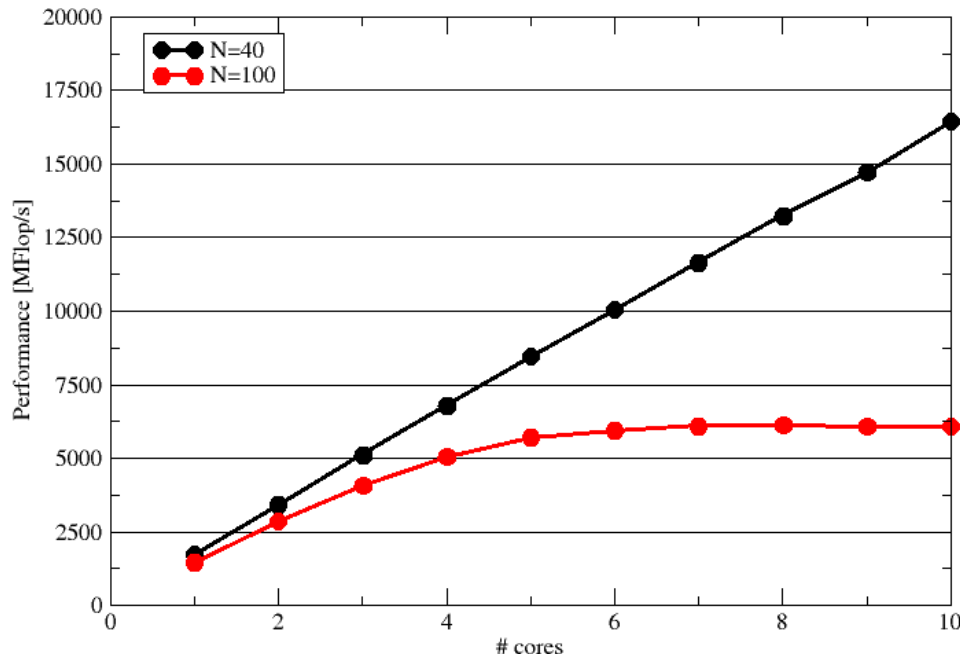
Intel IvyBridge-EP (2.2GHz, 10 cores/chip)

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<sup>3</sup>)

## Performance [GFlops/s]

Routine	Socket	Node
ddot	6726	14547
waxby	3642	6123
spmv	6374	<b>6320</b>
Total	5973	6531

## Memory Bandwidth measured [GB/s]

Routine	Socket 1	Socket 2	Total
ddot	44020	47342	91362
waxby	39795	28424	68219
spmv	43109	<b>2863</b>	45972

Pattern: Bad  
ccNUMA page  
placement

# Optimization: Apply correct data placement

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

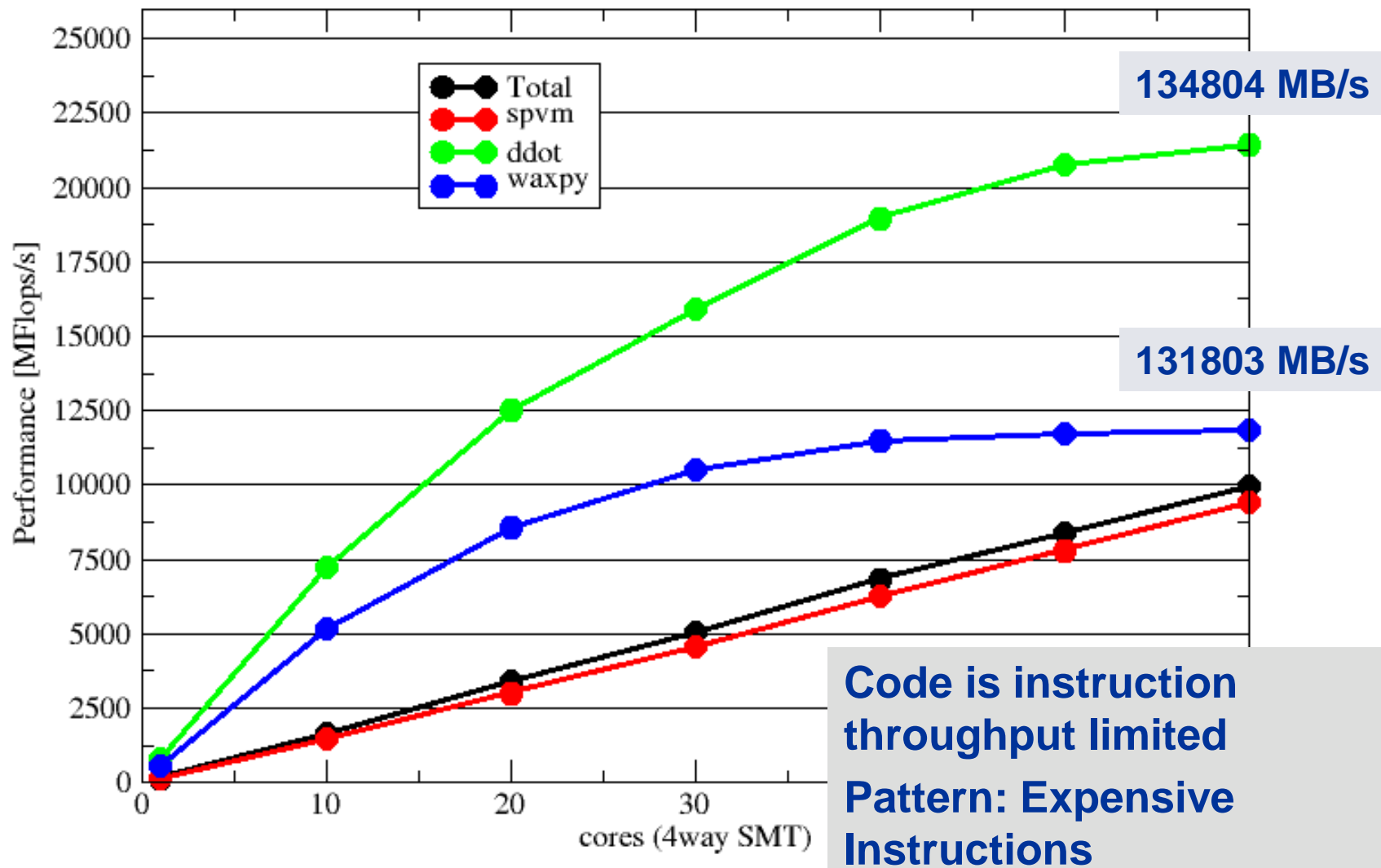
```
#pragma omp parallel for
  for (int i=0; i< local_nrow; i++){
    for (int j=0; j< 27; j++) {
      curvalptr[i*27 + j] = 0.0;
      curindptr[i*27 + j] = 0;
    }
  }
```

**Node performance: spmv 11692, total 10912**

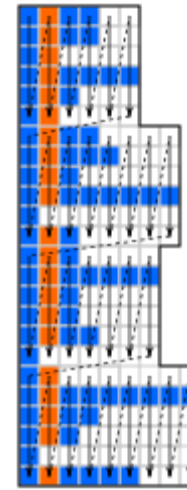
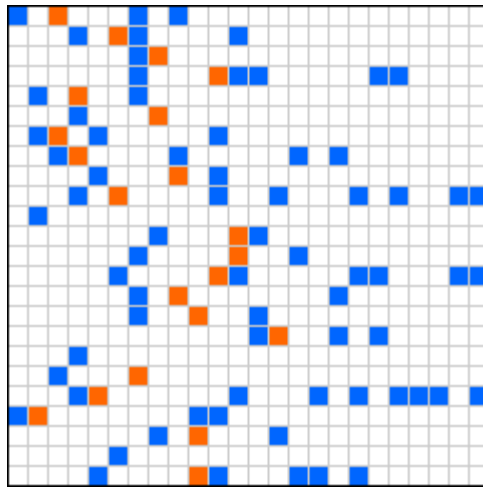
Routine	Socket 1	Socket 2	Total
ddot	46406	48193	94599
waxby	37113	24904	62017
spmv	45822	40935	86757



# Scaling behavior Intel Xeon Phi



# BJDS (Blocked JDS) – data format



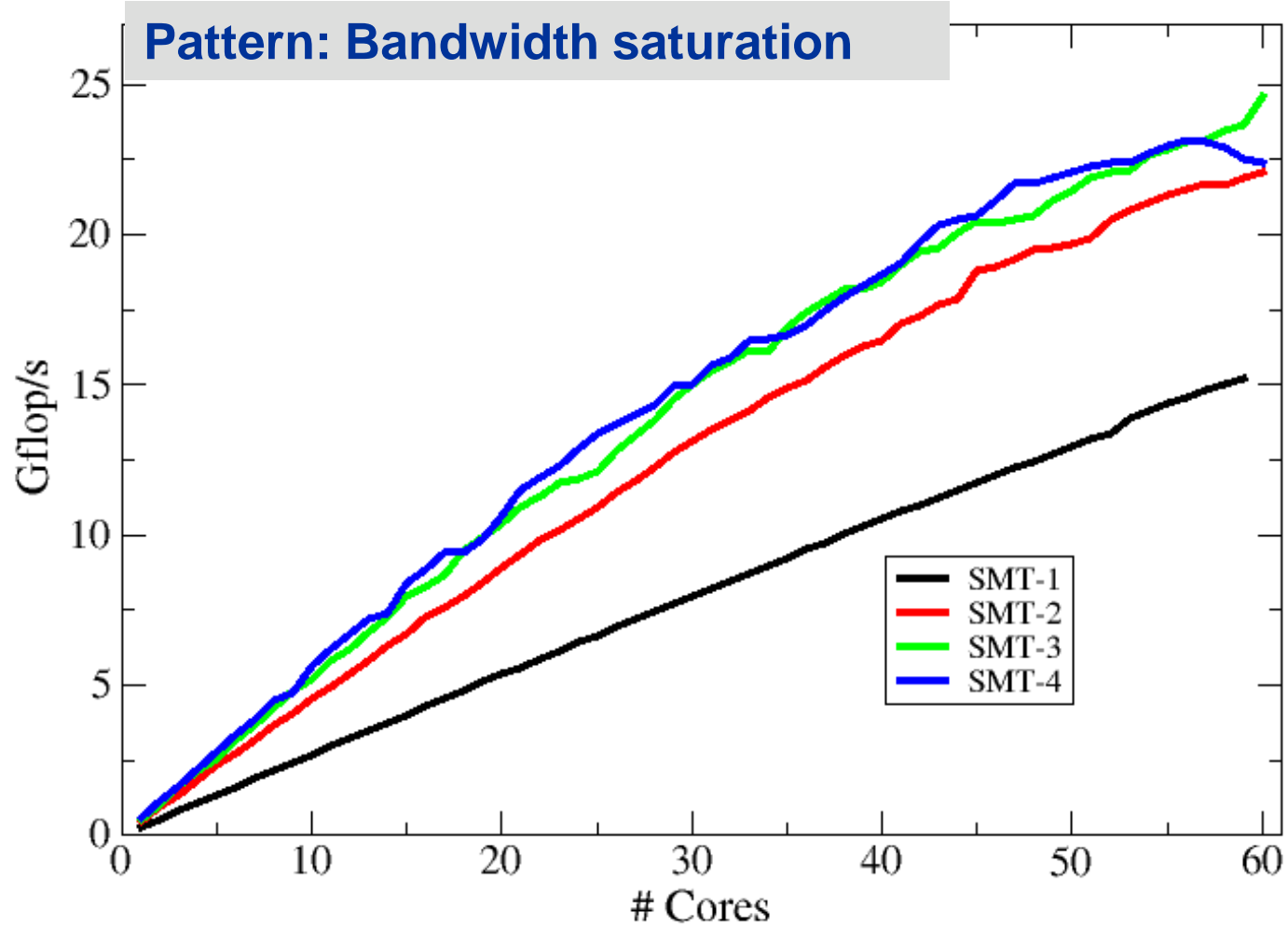
## Format creation

1. Shift nonzeros in each row to the left
2. Combine `chunkHeight` (multiple of vector length, here: 8) rows to one chunk
3. Pad all rows in chunk to the same length
4. Store matrix chunk by chunk and jagged-diagonal-wise within chunk

## Data arrays

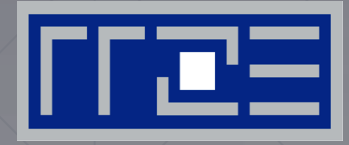
```
double val[]  
unsigned int col[]  
unsigned int chunkStart[]
```

# Optimized spmv data structure on Xeon Phi

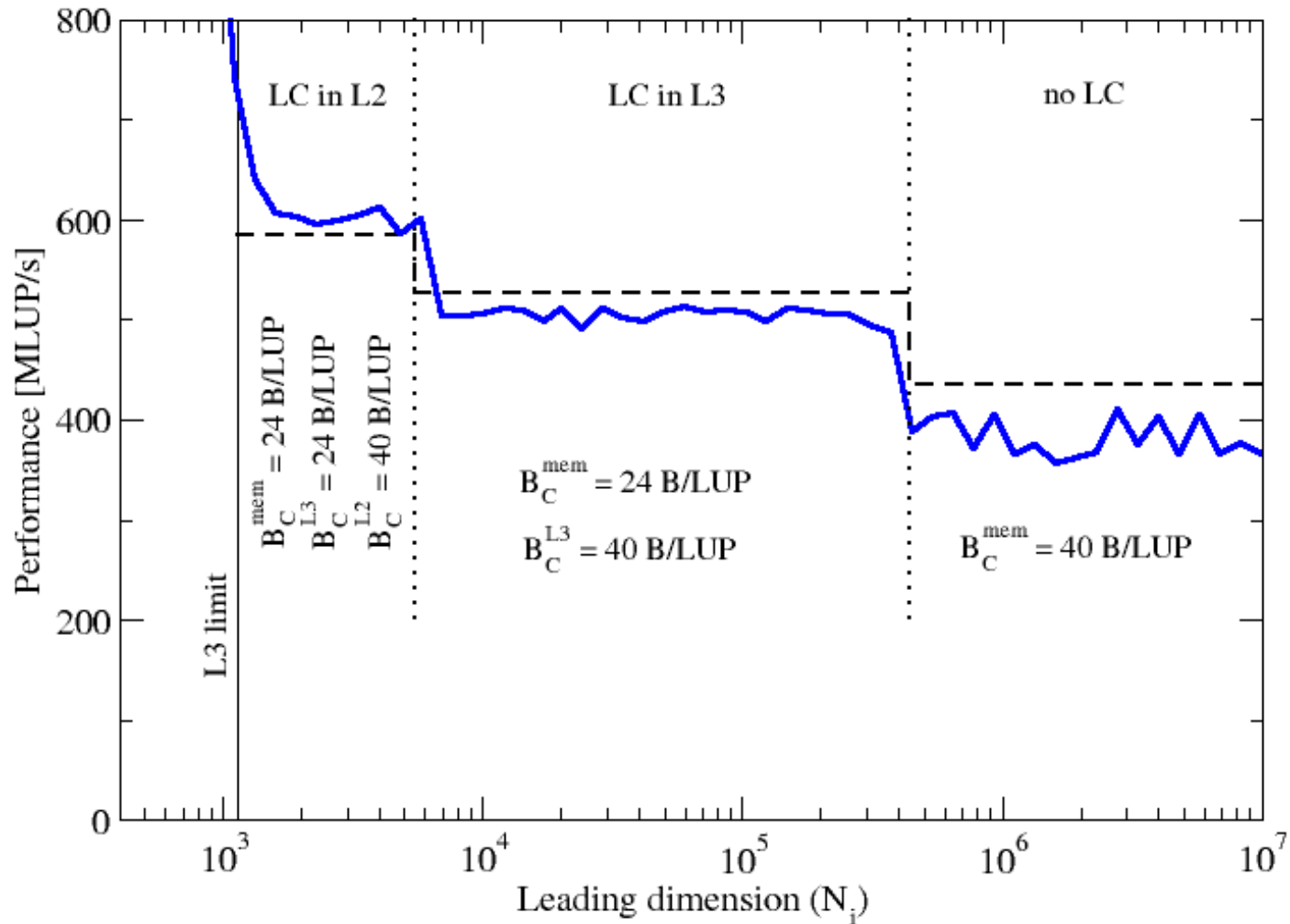




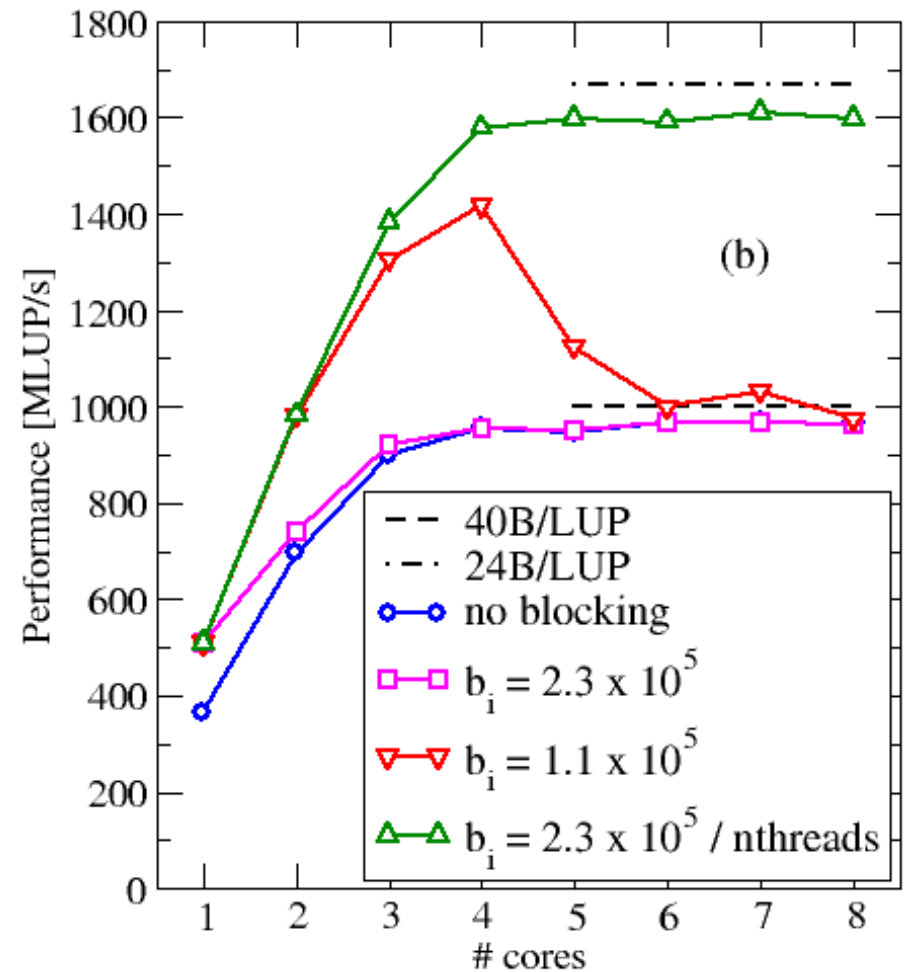
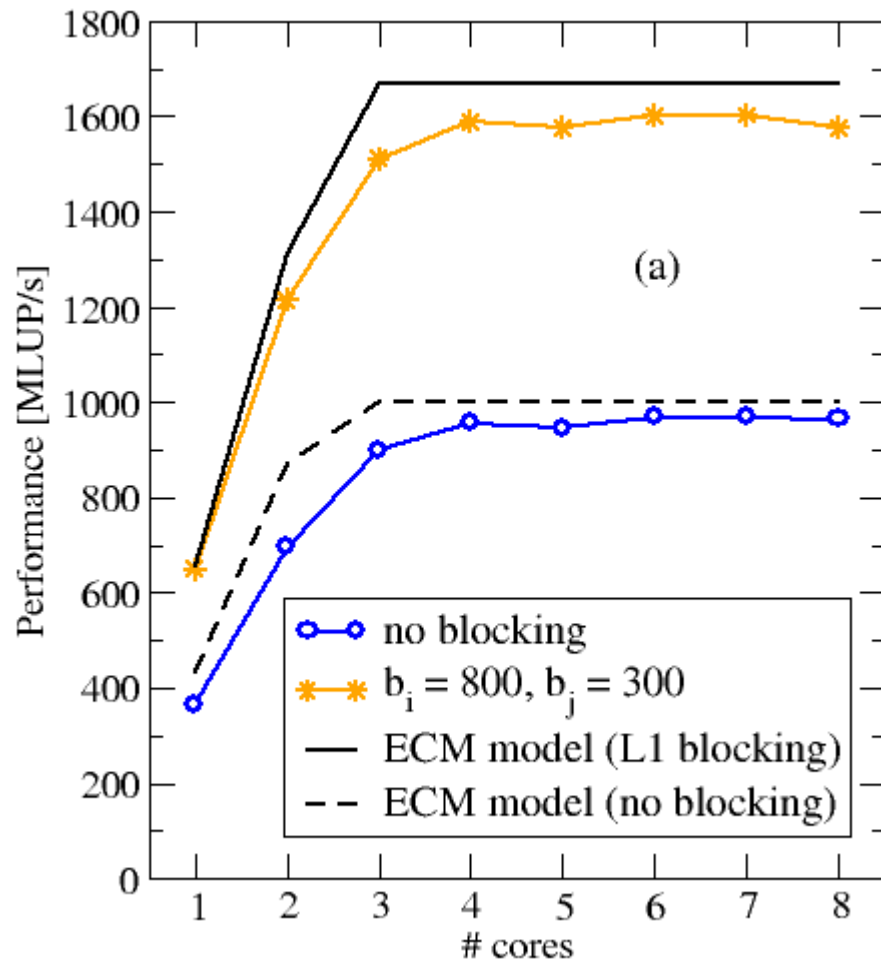
# EMPLOYING THE ECM MODEL ON STENCIL KERNELS



# 2D Jacobi Stencil: Layer condition



# J2D multicore chip scaling



# uxx stencil from earthquake propagation code

```
for(int k=2; k<=N-1; k++){
  for (int j=2; j<=N-1; j++){
    for (int i=2; i<=N-1; i++){
      d = 0.25*(d1[ k ][j][i] + d1[ k ][j-1][i]
                + d1[k-1][j][i] + d1[k-1][j-1][i]);
      u1[k][j][i] = u1[k][j][i] + (dth/d)
        *( c1*(xx[ k ][ j ][ i ]-xx[ k ][ j ][i-1])
          + c2*(xx[ k ][ j ][i+1]-xx[ k ][ j ][i-2])
          + c1*(xy[ k ][ j ][ i ]-xy[ k ][j-1][ i ])
          + c2*(xy[ k ][j+1][ i ]-xy[ k ][j-2][ i ])
          + c1*(xz[ k ][ j ][ i ]-xz[k-1][ j ][ i ])
          + c2*(xz[k+1][ j ][ i ]-xz[k-2][ j ][ i ]));
    }
  }
}
```

Expensive  
Divide!

**vdivpd**: 42 cycles throughput in double precision (SNB)

What about single precision?

# uxx kernel ECM model

Employing the Intel IACA tool for L1 throughput estimate.

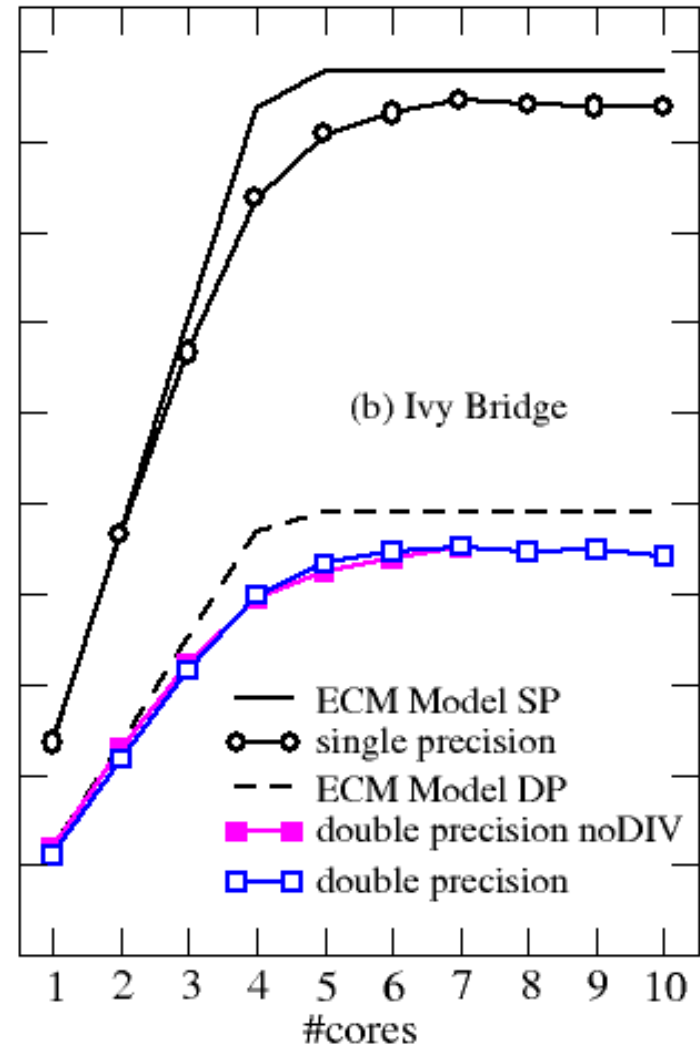
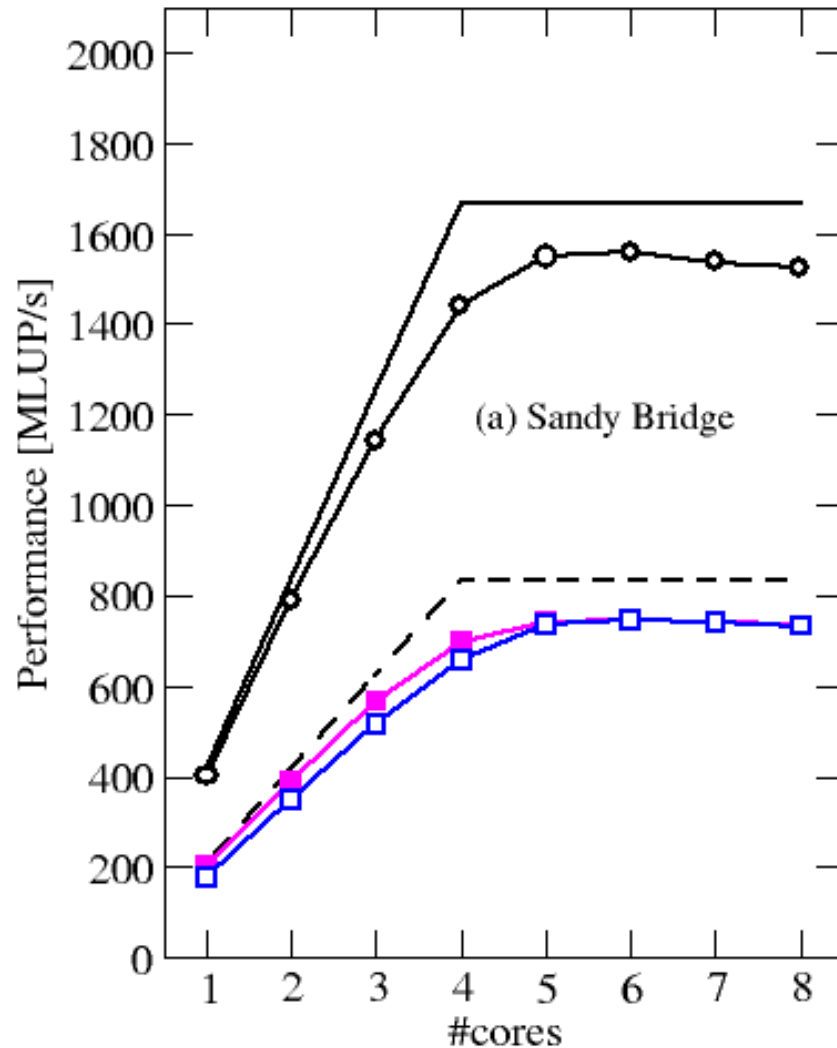
Version	ECM model	prediction
DP	{84    38   20   20   26} <i>cy</i>	{84 \ 84 \ 84 \ 104} <i>cy</i>
SP	{45    38   20   20   26} <i>cy</i>	{45 \ 58 \ 78 \ 104} <i>cy</i>
DP noDIV	{41    38   20   20   26} <i>cy</i>	{41 \ 58 \ 78 \ 104} <i>cy</i>

**Prediction for in Memory data set:**

1. SP is twice as fast as DP
2. All variants saturate at 4 cores
3. The presence of the DIV in DP makes no difference



# Comparison model vs. measurement



# uxx kernel: Optimization opportunities

- ECM model allows to predict upper limit for benefits from temporal blocking for the L3 cache:
  - Removes L3-MEM transfer time of 26cy
  - 24% speedup in DP (single core)
  - 33% speedup in SP (single core)
- Next bottleneck is the divide (DP) and L3 transfers (SP).
- True benefit: Both are **core-local** and therefore **scalable**.
- Expected performance in DP on chip level **2000 MLUP/s** instead of **800 MLUPS/s** (even with DIV)