The Practitioner's Cookbook for Good Parallel Performance on Multi- and Many-Core Systems

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

SC13 full-day tutorial
June 18, 2013
Denver, CO

Agenda

- **Preliminaries** — GHa — 08:30
- **Introduction to multicore architecture** — GW
  - Cores, caches, chips, sockets, ccNUMA, SIMD

10:00

10:30

- **LIKWID tools** — JT
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation effects and OpenMP overhead

12:00

- **Lunch break**

13:30

- **Node-level performance modeling** — GHa
  - The Roofline Model
  - Case study: 3D Jacobi solver and model-guided optimization

15:00

15:30

- **Optimal resource utilization** — JT, GW
  - SIMD parallelism
  - ccNUMA
  - Simultaneous multi-threading (SMT)

17:00

- **Optional: The ECM multicore performance model** — GHa

Agenda

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

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
          L DO
```

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

T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2
P P P P P P P P
L1D L1D L1D L1D L1D L1D L1D L1D
L2 L2 L2 L2 L2 L2 L2 L2
L3
Memory Interface
Memory

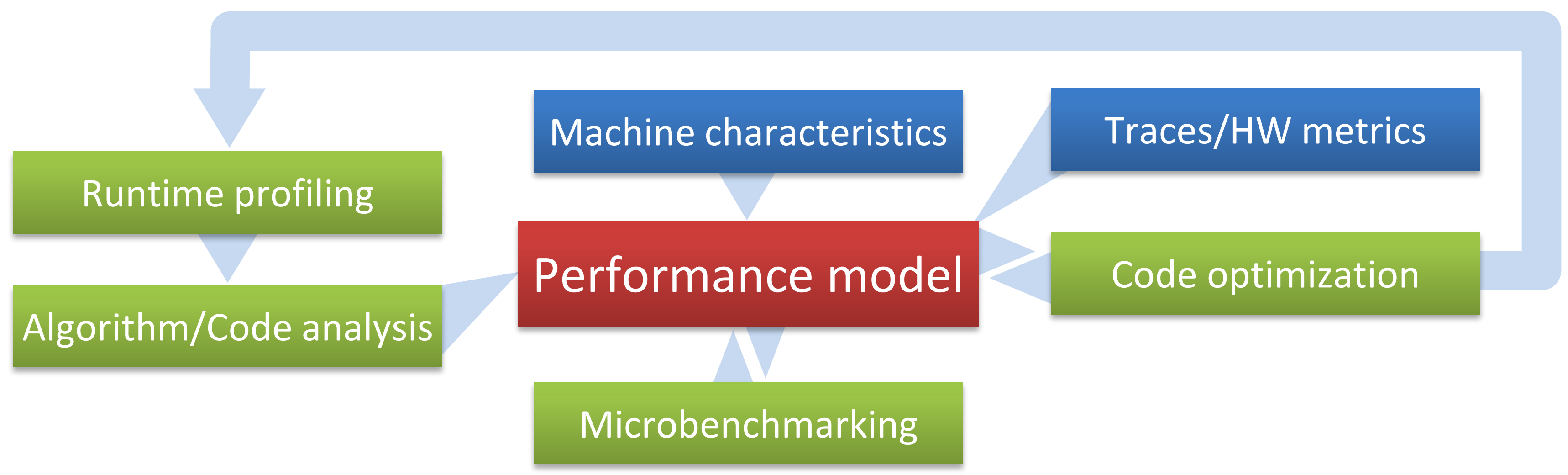
Performance [MLUP/s]
1500
1000
500
0
1 2 3 4 5 6 7 8
#cores
Version 1
Version 2
3D Stencil Update ("Jacobi")

Single core/socket efficiency is key issue!

Questions to ask in high performance computing

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

The Performance Engineering (PE) process:



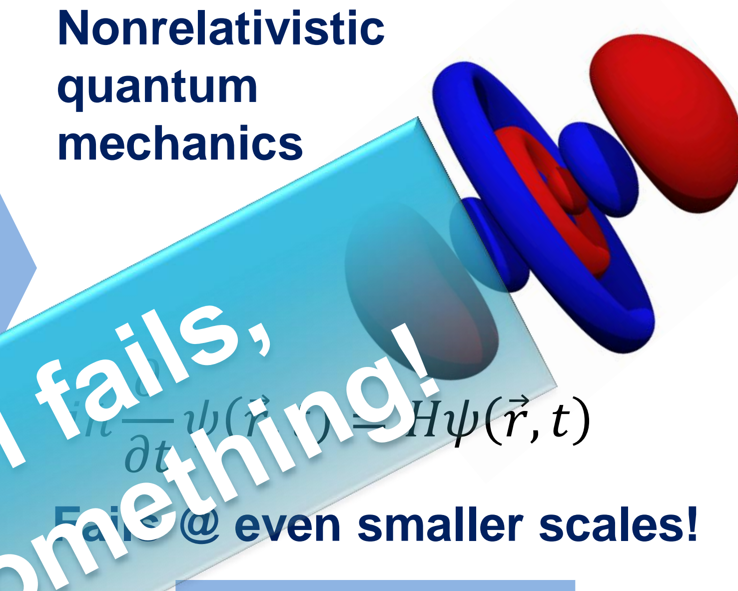
The performance model is the central component – if the model fails to predict the measurement, you learn something!

The analysis has to be done for every loop / basic block!

- **White Box Performance Model**
- **Simple enough to do on paper**
- **Catching the important influences**

Newtonian mechanics

Nonrelativistic quantum mechanics



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

Fails @ even smaller scales!

If a model fails, we learn something!

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

Fails @ small scales!

Relativistic quantum field theory

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

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

Agenda

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

Introduction: Modern node architecture

Multi- and manycore chips and nodes

A glance at basic core features

Caches and data transfers through the memory hierarchy

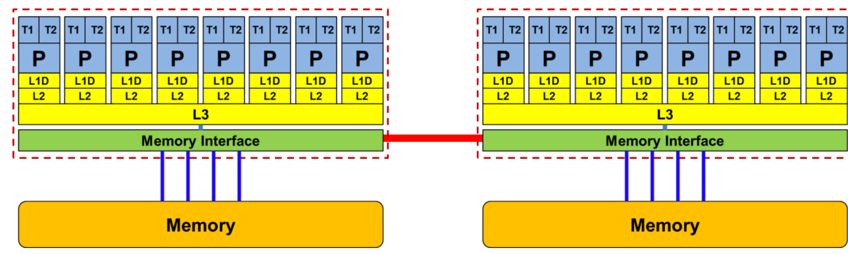
Memory organization

Accelerators

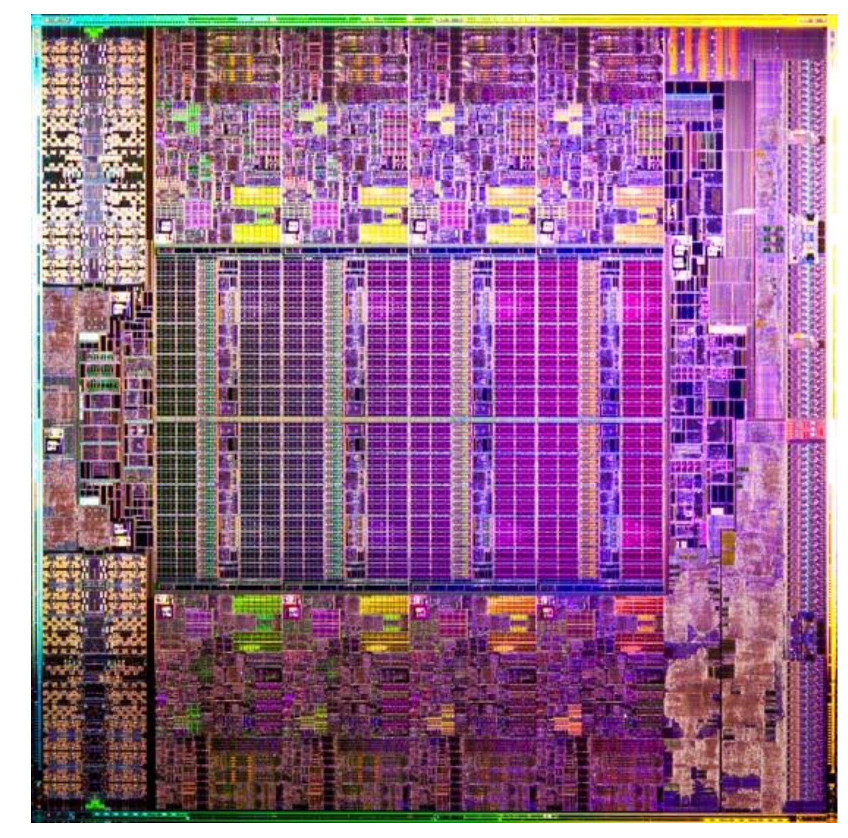
Programming models

Multi-Core: Intel Xeon 2600 (2012)

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



2-socket server



General-purpose cache based microprocessor core

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

Main memory

L2 unified cache

L1 data cache

Memory queue

INT reg. file

shift mask

INT op

LD

ST

Memory interface

L1 instr. cache

INT/FP queue

FP reg. file

FP mult

FP add

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

Pipelining of arithmetic/functional units

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

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

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

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

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

	1	2	3	4	5	...	N	N+1	N+2	N+3	N+4
Separate mant./exp.	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	B(N) C(N)	Wind–down			
Multiply mantissas		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	B(N–1) C(N–1)	B(N) C(N)			
Add exponents			B(1) C(1)	B(2) C(2)	B(3) C(3)	...	B(N–2) C(N–2)	B(N–1) C(N–1)	B(N) C(N)		
Normalize result				A(1)	A(2)	...	A (N–3)	A (N–2)	A (N–1)	A(N)	
Insert sign	Wind–up				A(1)	...	A (N–4)	A (N–3)	A (N–2)	A (N–1)	A(N)

Cycle

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

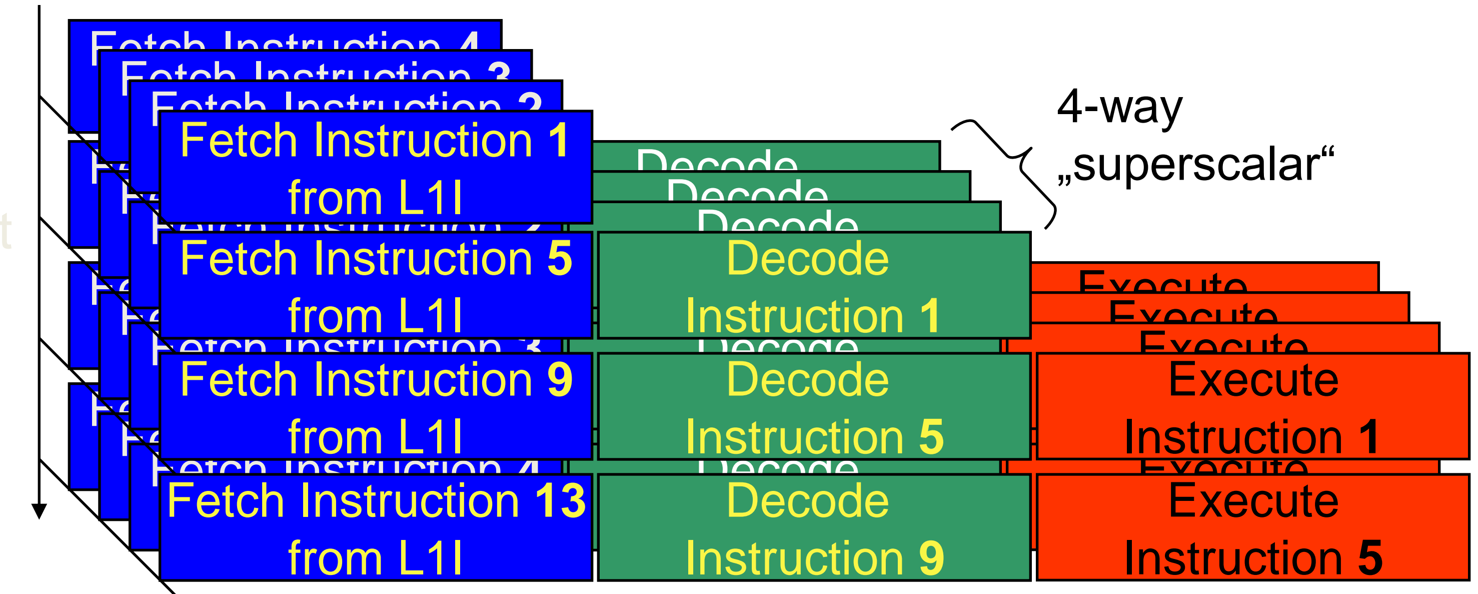
Wind-up/-down phases: Empty pipeline stages

Pipelining: The Instruction pipeline

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

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

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



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

SMT principle (2-way example):

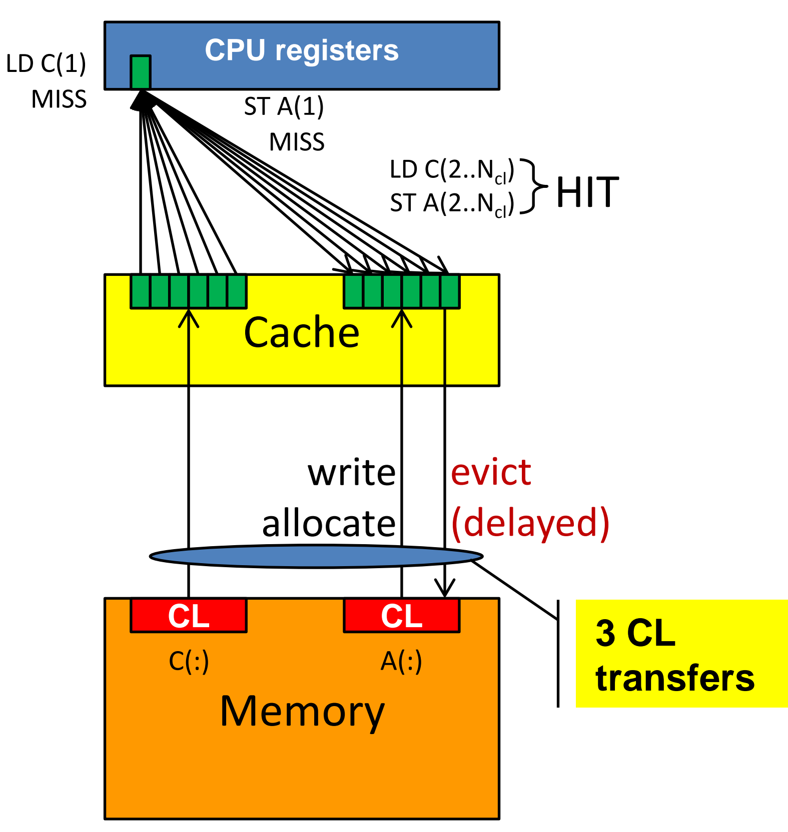
Core details: SIMD processing

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

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

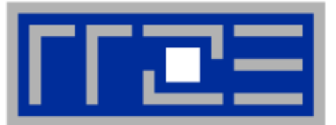
- Remember: Caches are organized in cache lines (e.g., 64 bytes)
- Only complete cache lines are transferred between memory hierarchy levels (except registers)
- MISS: Load or store instruction does not find the data in a cache level → CL transfer required

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

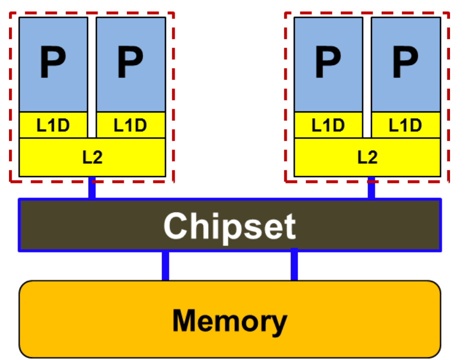


From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes



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

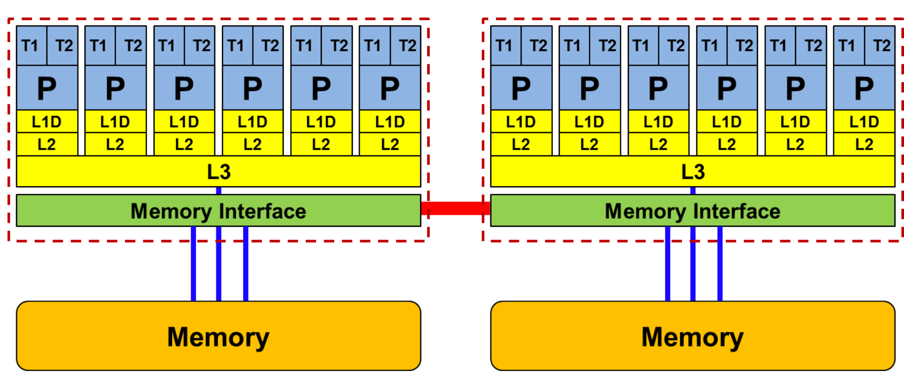


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system “anisotropy”

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



Cache-coherent Non-Uniform Memory Architecture (ccNUMA)

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

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

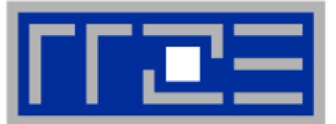
- **Up to 16 cores (8 Bulldozer modules) in a single socket**
- **Max. 2.6 GHz (+ Turbo Core)**
- **P_{max} = (2.6 x 8 x 8) GF/s = 166.4 GF/s**

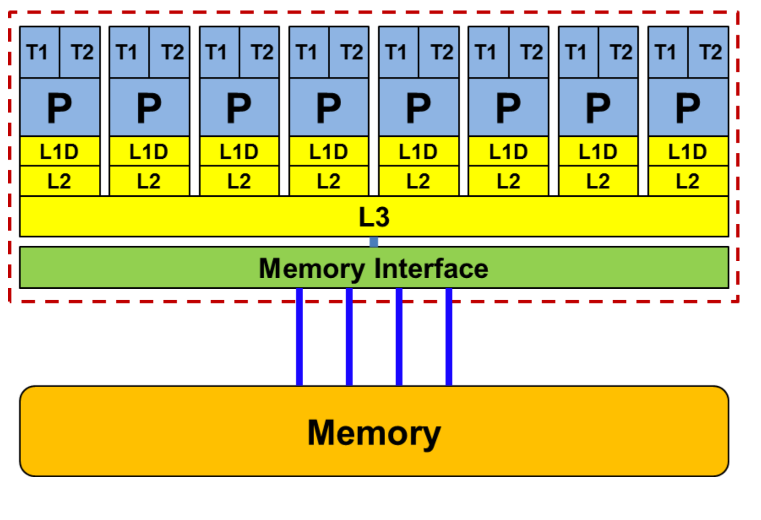
Each Bulldozer module:

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

2 DDR3 (shared) memory channels > 15 GB/s

There is no single driving force for chip performance!





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

Floating Point (FP) Performance:

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

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

TOP500 rank 1 (mid-90s)

P = 173 GF/s (dp) / 346 GF/s (sp)

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

Challenges with modern compute nodes

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

Core:
SIMD vectorization
SMT

Socket:
Parallelization
Shared Resources

GPU #1

Other I/O

PCIe link

GPU #2

T1 T2 T1 T2 T1 T2 T1 T2 T1 T2 T1 T2

P P P P P P

L1D L1D L1D L1D L1D L1D

L2 L2 L2 L2 L2 L2

L3

Memory Interface

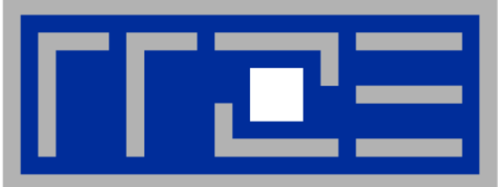
Memory

Memory

Where is the data?

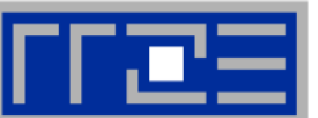
Node:
ccNUMA/data locality

Accelerators:
Data transfer to/from host



Interlude:
A glance at current accelerator technology

NVIDIA Kepler GK110 Block Diagram



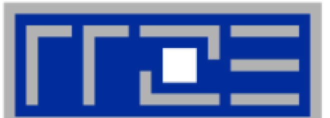
Architecture

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

- **3:1 SP:DP performance**

© NVIDIA Corp. Used with permission.

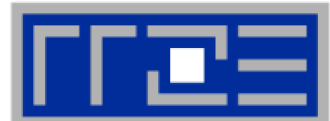
Intel Xeon Phi block diagram



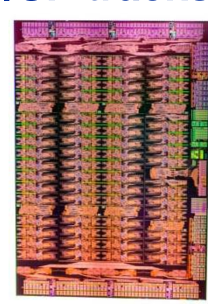
Architecture

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

Comparing accelerators

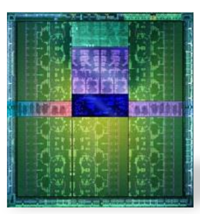


- **Intel Xeon Phi**
  - **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **480/960 SIMD DP/SP tracks**



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

- **NVIDIA Kepler K20**

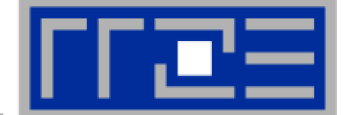


  - 15 SMX units each with 192 "cores" → **960/2880 DP/SP "cores"**
  - Clock Speed: ~700 MHz
  - Transistor count: 7.1 B (28nm)
  - Power consumption: ~250 W
  - Peak Performance (DP): ~ 1.3 TF/s
  - Memory BW: ~ 250 GB/s (GDDR5)
  - Threads to execute: 10,000+
  - Programming: CUDA, OpenCL, (OpenACC)

TOP500 rankings Nov 2012

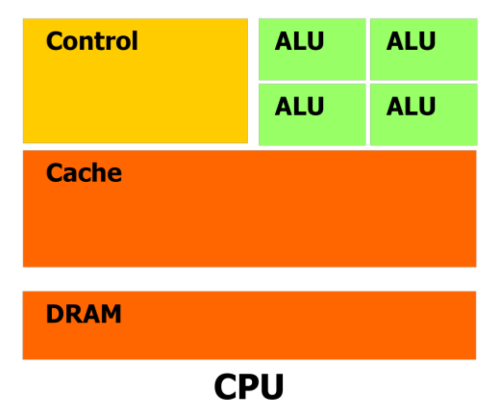
- Top7: "Stampede" at Texas Center for Advanced Computing
- Top1: "Titan" at Oak Ridge National Laboratory

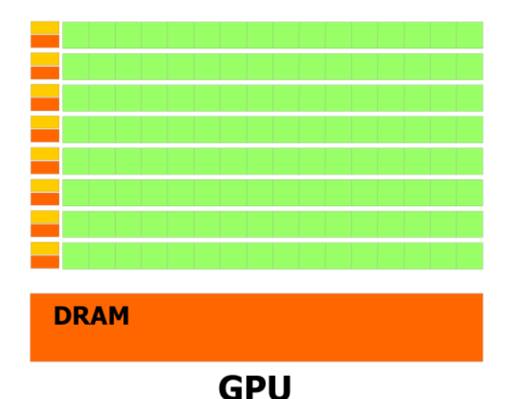
Trading single thread performance for parallelism: *GPGPUs vs. CPUs*



GPU vs. CPU
light speed estimate:

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





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

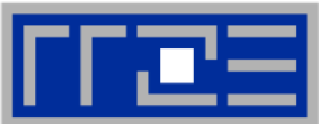
+ *Single Precision*

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

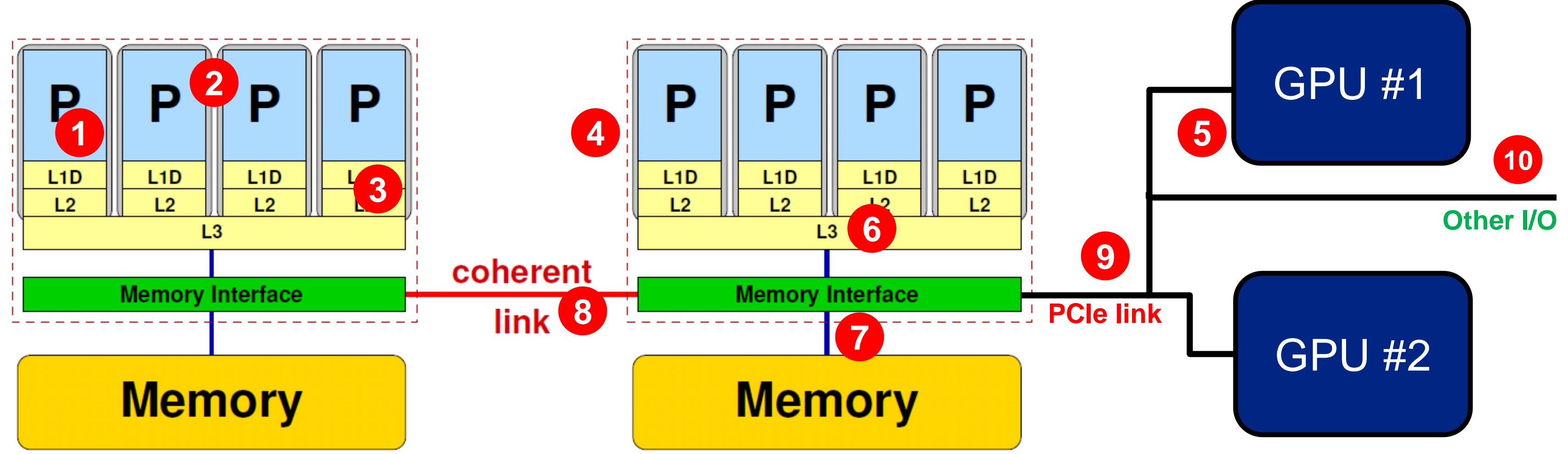
Complete compute device

Node topology and programming models

Parallelism in a modern compute node



- Parallel and shared resources within a shared-memory node



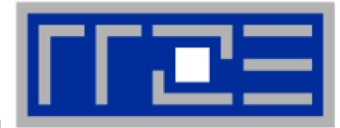
Parallel resources:

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

Shared resources:

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

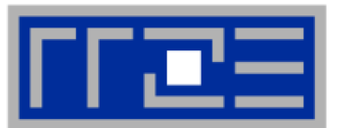
How does your application react to all of those details?



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

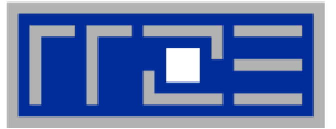
All models require awareness of *topology* and *affinity* issues for getting best performance out of the machine!

Parallel programming models:
Pure MPI

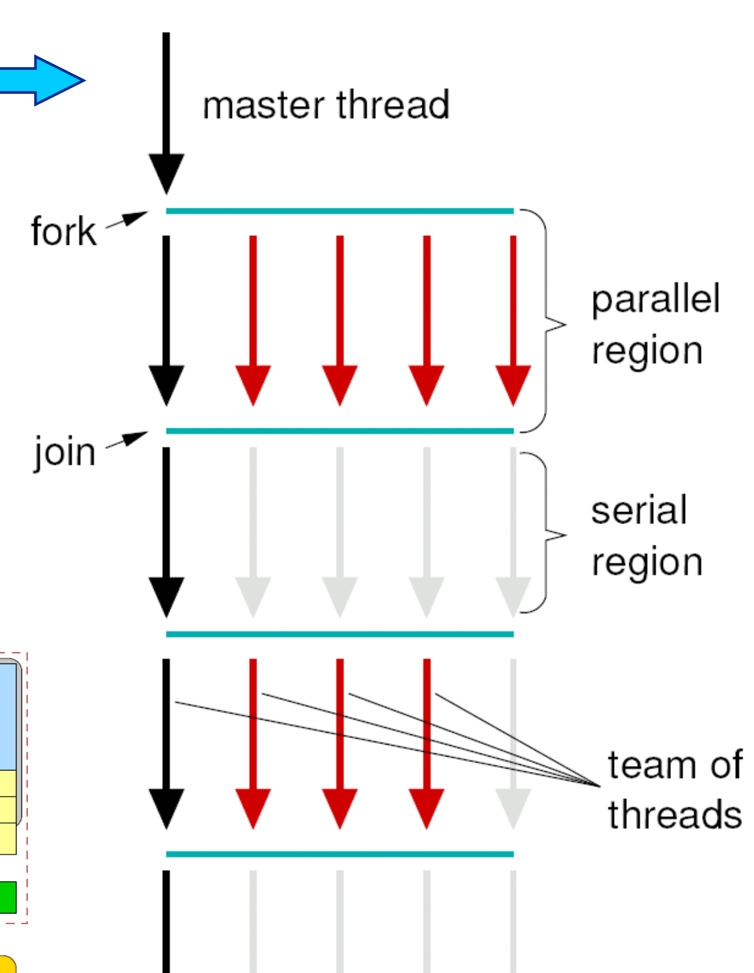


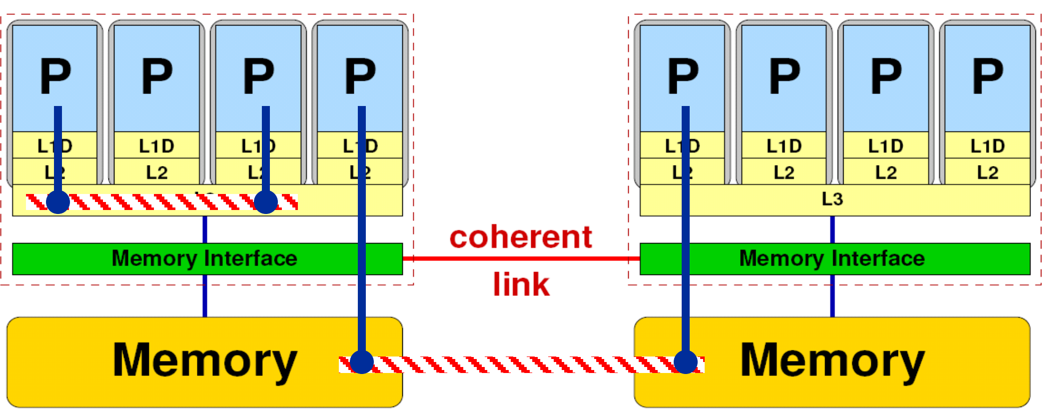
- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI "knows what to do"!?
- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

Parallel programming models:
Pure threading on the node



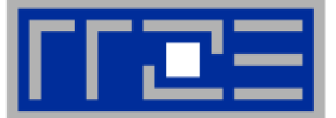
- **Machine structure is invisible to user**
  - → Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,…) should know about the details
- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology





Parallel programming models: Lots of choices

Hybrid MPI+OpenMP on a multicore multisocket cluster



One MPI process / node

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

OpenMP threads pinned **"round robin"** across cores in node

Two MPI processes / socket
OpenMP threads on same socket

Conclusions about architecture

- **Modern computer architecture has a rich "topology"**

- **Node-level hardware parallelism takes many forms**
  - Sockets/devices – CPU: 1-8, GPGPU: 1-6
  - Cores – moderate (CPU: 4-16) to massive (GPGPU: 1000's)
  - SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10's-100's)
  - Superscalarity (CPU: 2-6)

- **Exploiting performance: parallelism + bottleneck awareness**
  - **"High Performance Computing" == computing at a bottleneck**

- **Performance of programs is sensitive to architecture**
  - Topology/affinity influences overheads of popular programming models
  - Standards do not contain (many) topology-aware features
    - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
  - Apart from overheads, performance features are largely independent of the programming model

Agenda

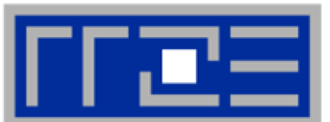
- **Preliminaries**
- **Introduction to multicore architecture**
  - Cores, caches, chips, sockets, ccNUMA, SIMD
- **LIKWID tools**
- **Microbenchmarking for architectural exploration**
  - Streaming benchmarks: throughput mode
  - Streaming benchmarks: work sharing
  - Roadblocks for scalability: Saturation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

Multicore Performance and Tools

Probing node topology

- Standard tools
- likwid-topology

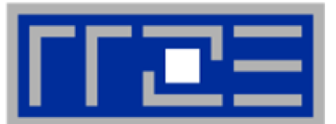
How do we figure out the node topology?



- **Topology =**
  - Where in the machine does core #n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?
- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers may change across kernels and BIOSes even on identical hardware
  - `numactl --hardware` prints ccNUMA node information →
  - Information on caches is harder to obtain

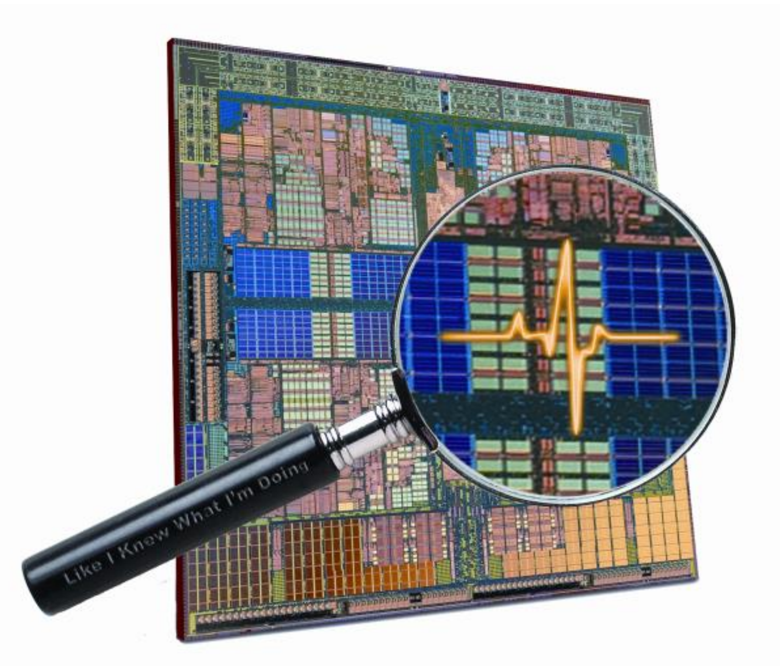
```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: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

How do we figure out the node topology?



- **LIKWID tool suite:**

  Like
  I
  Knew
  What
  I'm
  Doing

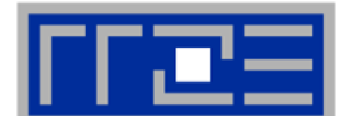


- **Open source tool collection (developed at RRZE):**

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

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

Likwid Tool Suite

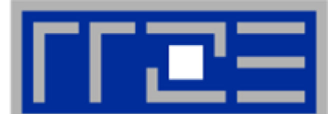


- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

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

Output of `likwid-topology -g`

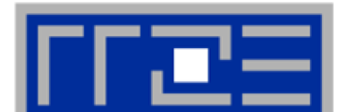
on one node of Cray XE6 “Hermit”



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

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

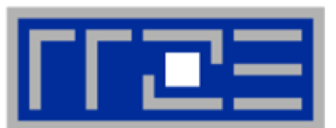
Output of likwid-topology continued



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

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

Output of likwid-topology continued

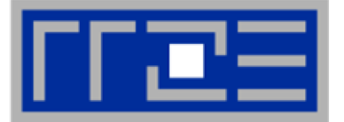


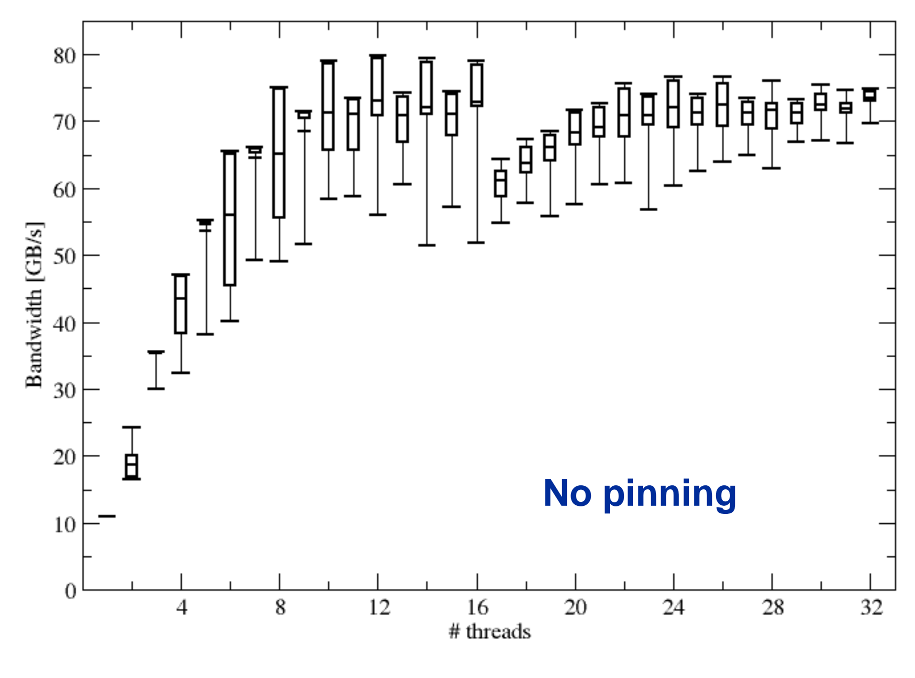
```
*************************************************************
Graphical:
*************************************************************
Socket 0:
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------+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

Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
- aprun (Cray)

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





Memory Interface
L3
Memory

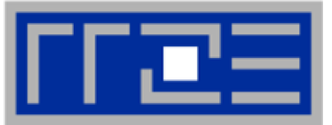
Pinning (physical cores first, first socket first)

There are several reasons for caring about affinity:

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

Generic thread/process-core affinity under Linux

Overview



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

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

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

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

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

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

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

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

Generic thread/process-core affinity under Linux

- Complementary tool: `numactl`

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

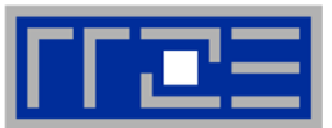
  Example: `numactl --cpunodebind=1 command [args]`
  Restricts process to specified ccNUMA node(s)

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

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

- Again, this is not suitable for a shared machine

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

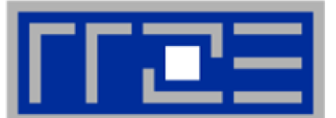


- **Highly OS-dependent system calls**
  - But available on all systems

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

Likwid-pin

Overview

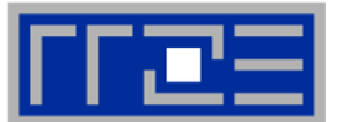


- **Pins processes and threads to specific cores without touching code**
- **Directly supports pthreads, gcc OpenMP, Intel OpenMP**
- **Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!**
- **Can also be used as a superior replacement for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

- **Usage examples:**
  - Physical numbering (as given by likwid-topology):

    ```
    likwid-pin -c 0,2,4-6 ./myApp parameters
    ```
  - Logical numbering by topological entities:

    ```
    likwid-pin -c S0:0-3 ./myApp parameters
    ```



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

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

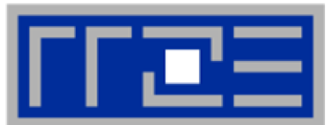
Main PID always pinned

Skip shepherd thread

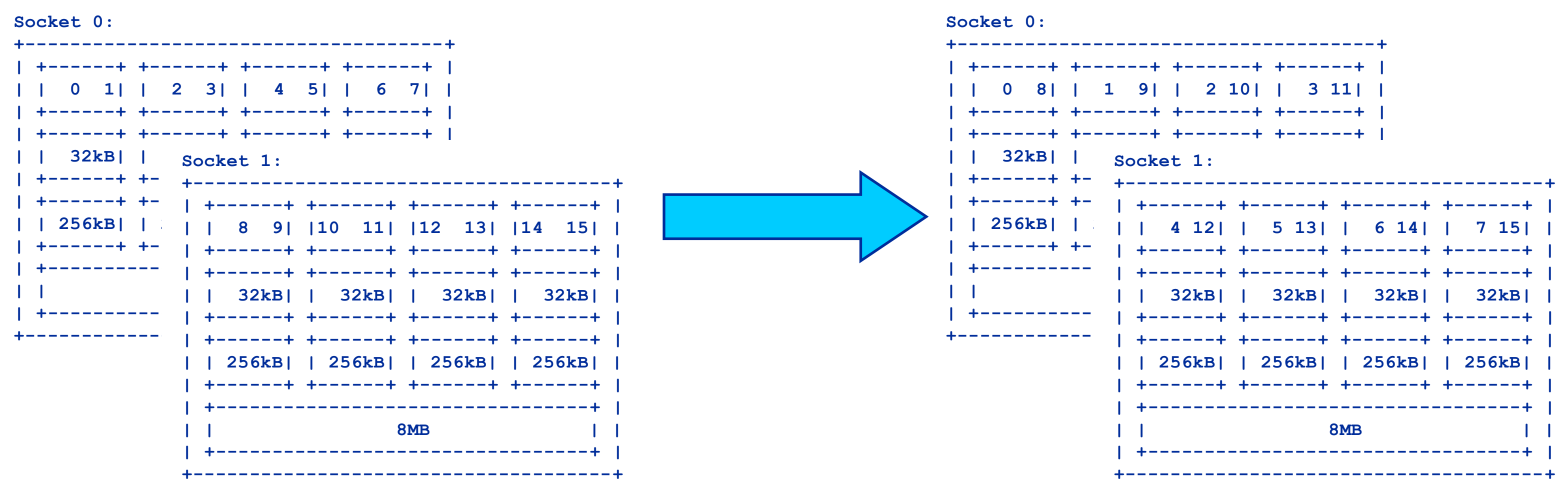
Pin all spawned threads in turn

Likwid-pin

Using logical core numbering



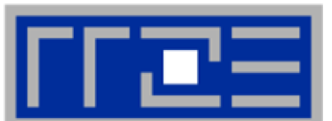
- **Core numbering may vary from system to system even with identical hardware**
  - Likwid-topology delivers this information, which can then be fed into likwid-pin
- **Alternatively, likwid-pin can abstract this variation and provide a purely logical numbering (physical cores first)**



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

Likwid-pin

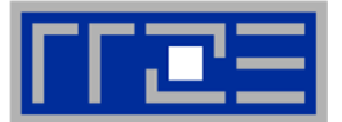
Using logical core numbering



- **Possible unit prefixes**

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

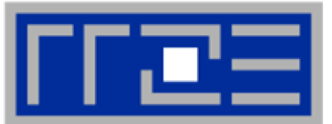
Default if -c is not specified!



DEMO

Multicore performance tools: Probing performance behavior

likwid-perfctr

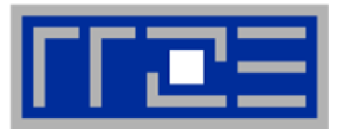


1. Runtime profile / Call graph (gprof)
2. Instrument those parts which consume a significant part of runtime
3. Find performance signatures

Possible signatures:

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

Probing performance behavior

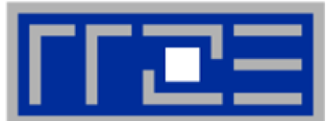


- **How do we find out about the performance properties and requirements of a parallel code?**
  - Profiling via advanced tools is often overkill
- **A coarse overview is often sufficient**
  - likwid-perfctr (similar to "perfex" on IRIX, "hpmcount" on AIX, "lipfpm" on Linux/Altix)
  - Simple end-to-end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

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

likwid-perfctr

Example usage with preconfigured metric group

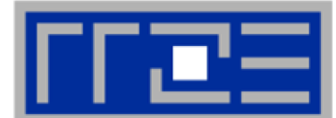


```
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP  ./stream.exe
-------------------------------------------------------------
CPU type:       Intel Core Lynnfield processor
CPU clock:      2.93 GHz
-------------------------------------------------------------
Measuring group FLOPS_DP
-------------------------------------------------------------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

Always measured

Configured metrics (this group)

Derived metrics

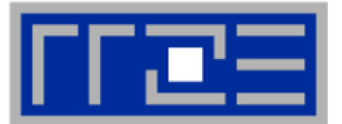


Things to look at (in roughly this order)

- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Shared cache BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions, branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are overrated
  - If I really know my code, I can often *calculate* the misses
  - Runtime and resource utilization is much more important



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

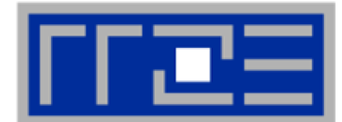
Event	core 0	core 1	core 2	core 3	core 4	core 5
INSTR_RETIRED_ANY	2.10045e+10	1.90983e+10	1.729e+10	1.60898e+10	1.67958e+10	1.84689e+10
CPU_CLK_UNHALTED_CORE	1.82569e+10	1.81203e+10	1.81802e+10	1.82084e+10	1.82334e+10	1.82484e+10
CPU_CLK_UNHALTED_REF	1.66053e+10	1.6473e+10	1.65274e+10	1.65531e+10	1.65758e+10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

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

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

likwid-perfctr

… and load-balanced codes



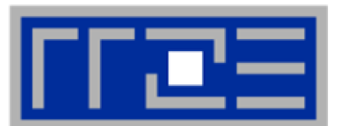
```
env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out
```

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

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

Higher CPI but better performance

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



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

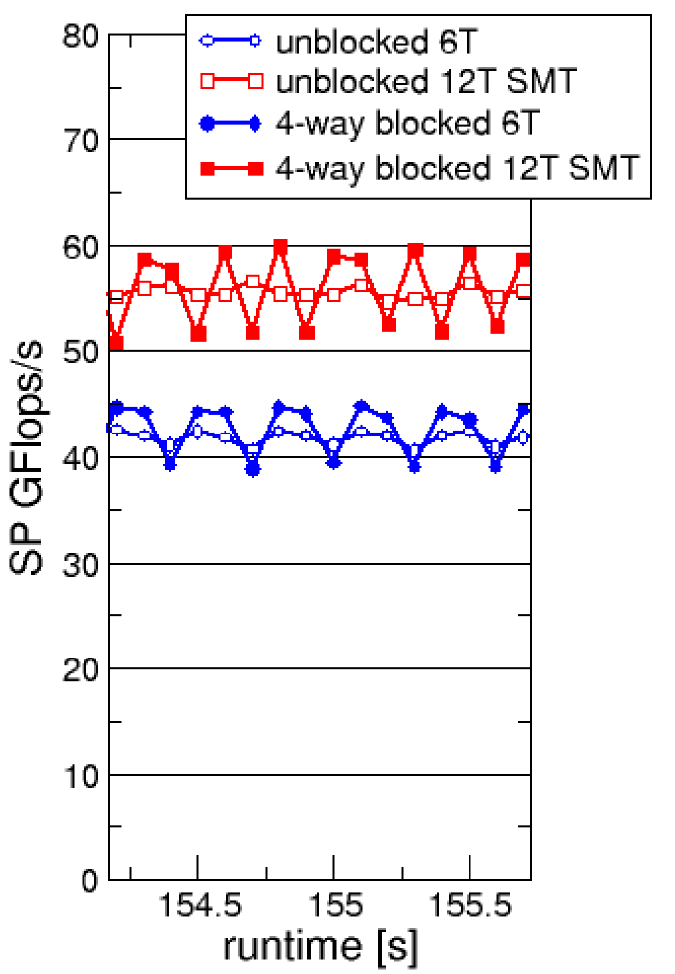
  This enables to listen on what currently happens without any overhead:

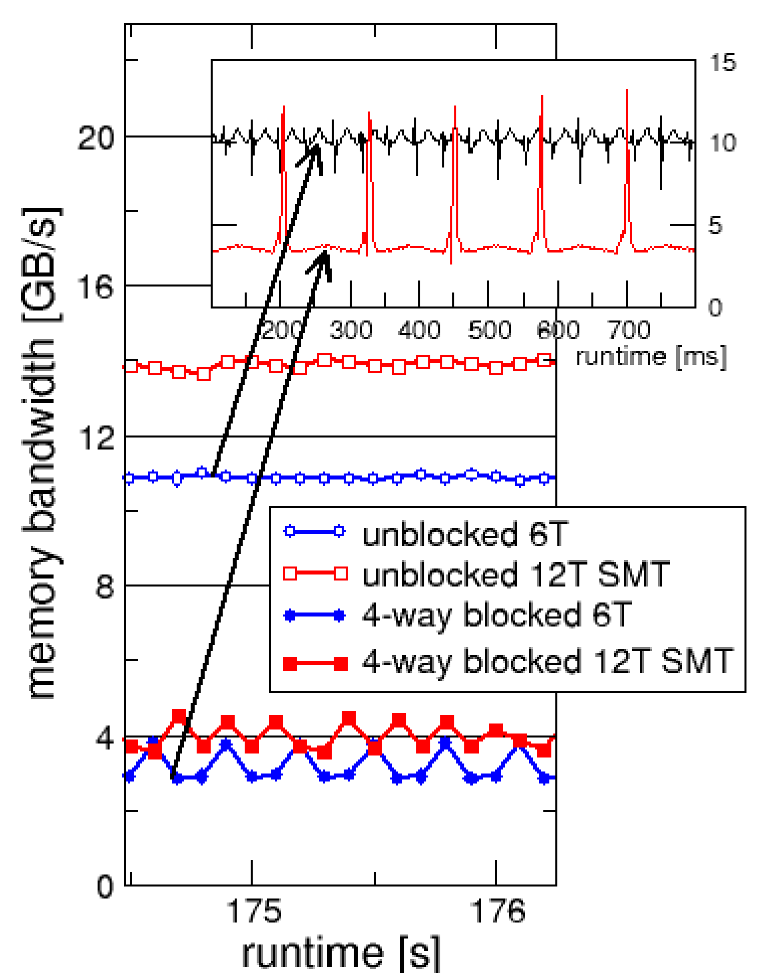
```
likwid-perfctr -c N:0-11 -g FLOPS_DP  -s 10
```

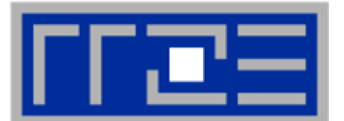
- **It can be used as cluster/server monitoring tool**
- **A frequent use is to measure a certain part of a long running parallel application from outside**

- **likwid-perfctr supports time resolved measurements of full node:**

```
likwid-perfctr -c N:0-11 -g MEM -d 50ms  > out.txt
```







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

```
likwid_markerInit();  // must be called from serial region

likwid_markerStartRegion("Compute");
. . .
likwid_markerStopRegion("Compute");


likwid_markerStartRegion("postprocess");
. . .
likwid_markerStopRegion("postprocess");


likwid_markerClose();  // must be called from serial region
```

likwid-perfctr

Group files

optional

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

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

Measuring energy consumption with LIKWID

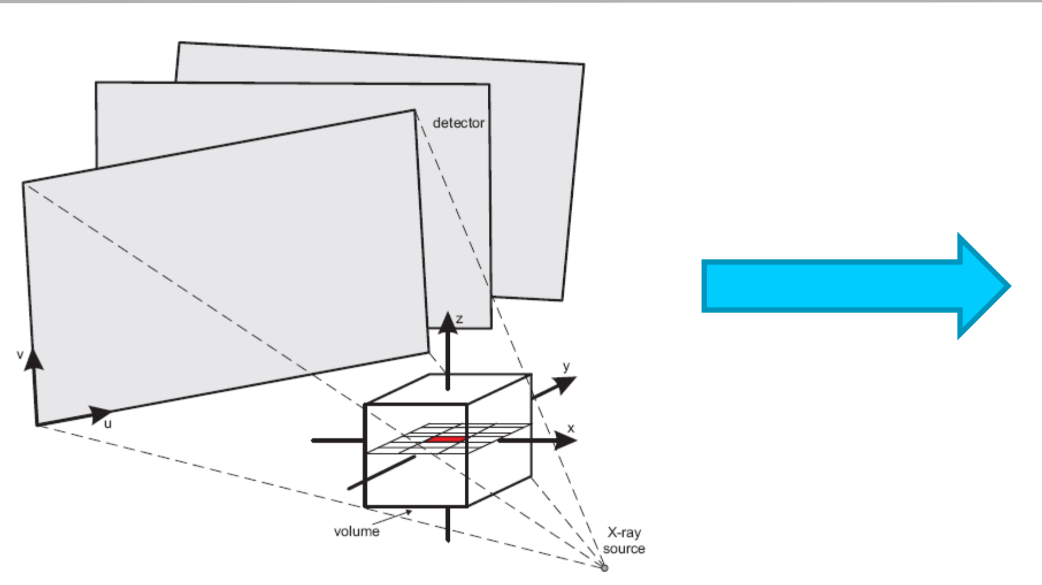
- **Implements Intel RAPL interface (Sandy Bridge)**
- **RAPL = “Running average power limit”**

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

Example:

A medical image reconstruction code on Sandy Bridge

optional

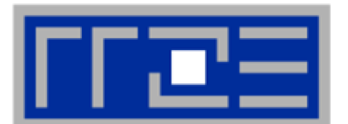


Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

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

Faster code → less energy

Agenda



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

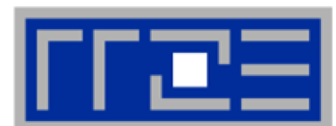
Microbenchmarking for architectural exploration

Probing of the memory hierarchy

Saturation effects in cache and memory

Typical OpenMP overheads

Latency and bandwidth in modern computer environments



ns 10^{-9} | L1 cache | 10^{11}

10^{-8} | L2/L3 cache

10^{-7} | Main memory | 10^{10}

μs 10^{-6} | HPC networks

10^{-5} | 10^{9} 1 GB/s

Gigabit Ethernet

10^{-4} | Solid state disk

ms 10^{-3} | 10^{8}

10^{-2} | Local hard disk

Internet

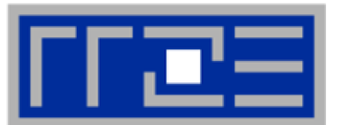
10^{-1} | 10^{7}

Latency [sec]

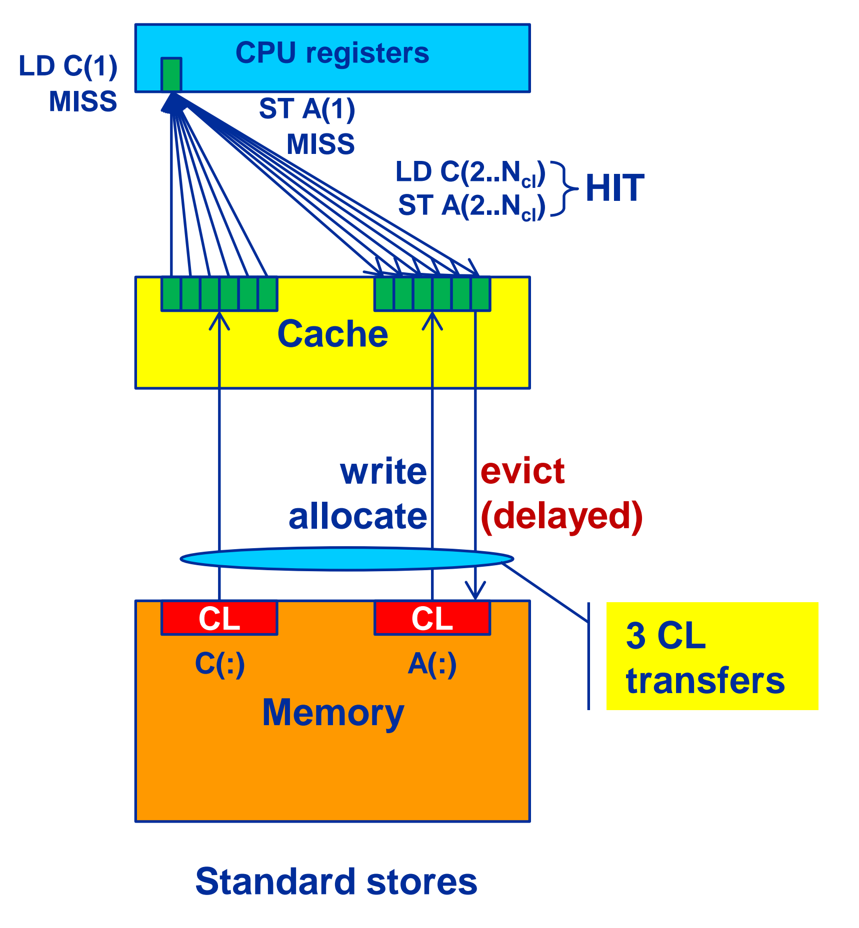
Bandwidth [bytes/sec]

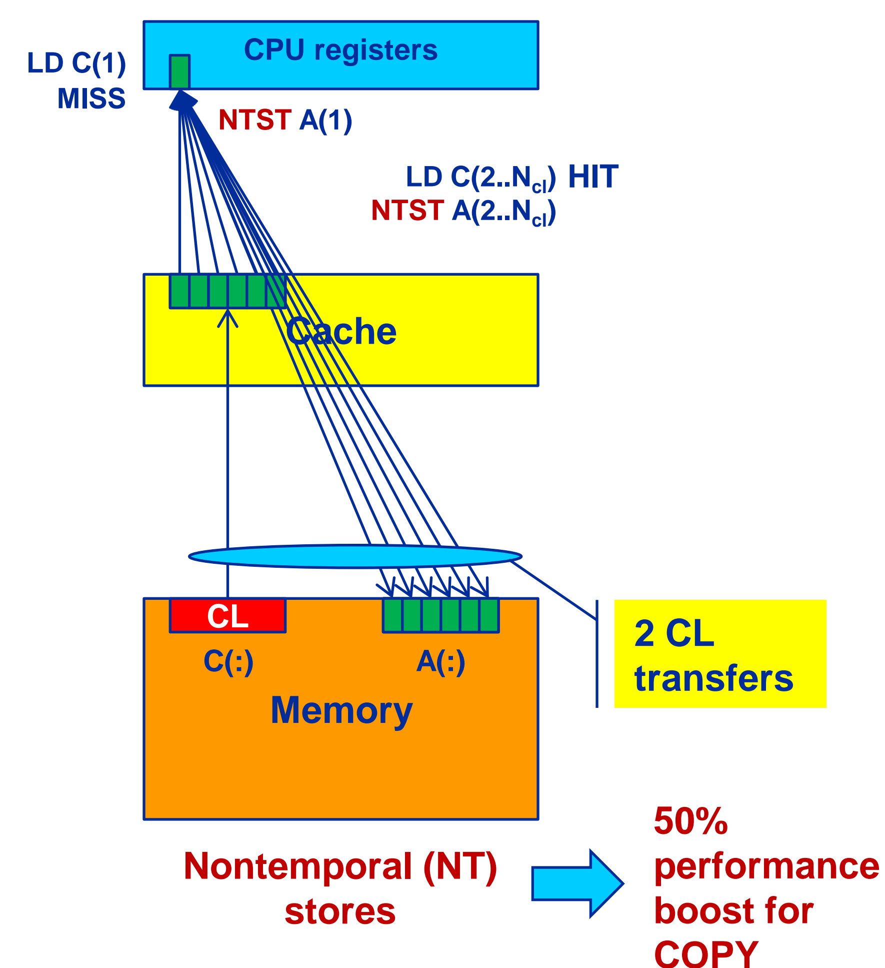
HPC plays here

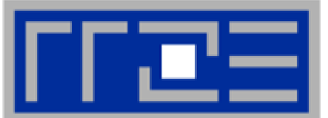
Avoiding slow data paths is the key to most performance optimizations!



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







Simple streaming benchmark:

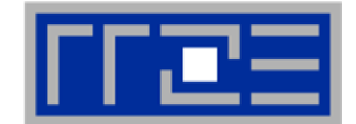
```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

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

Prevents smarty-pants compilers from doing “clever” stuff

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!

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



Performance [MFlops/s]

9000
8000
7000
6000
5000
4000
3000
2000
1000
0

10^2 10^3 10^4 10^5 10^6

Loop length

AVX
scalar

Theoretical limit

4 W / iteration
→ 128 GB/s

L1D cache (32k)

L2 cache (256k)

L3 cache (20M)

Memory

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

What about multiple cores?

Do the bandwidths scale?

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

Theoretical limit

4 W / iteration → 128 GB/s

AVX
scalar

2.66x SIMD impact

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

Data far away →smaller SIMD impact

Theoretical limit

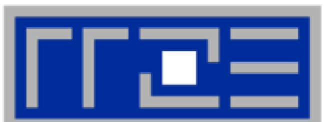
4 W / iteration → 48 GB/s

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

Performance [MFlops/s]: 0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000

Loop length: 10^2, 10^3, 10^4, 10^5, 10^6

See later for more on SIMD benefits



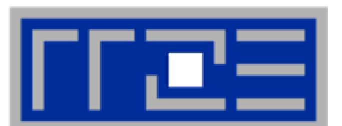
Every core runs its own, independent triad benchmark

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

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

→ **pure hardware probing, no impact from OpenMP overhead**

Throughput vector triad on Sandy Bridge socket (3 GHz)



Performance [GFlops/s]

Loop length

T=1
T=2
T=4
T=8

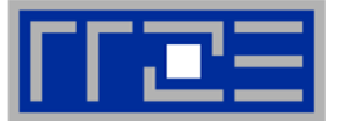
Saturation effect in memory

Scalable BW in L1, L2, L3 cache

L1D L2 L3 Memory Interface Memory

Bandwidth limitations: Main Memory

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



Saturation with 3 threads

Saturation with 2 threads

1 thread cannot saturate bandwidth

Saturation with 4 threads

Westmere
Sandy Bridge
Interlagos

Bandwidth [GByte/s]

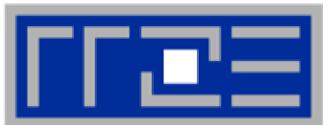
50
40
30
20
10
0

0
4
8
12
16

cores

1 NUMA domain
2 NUMA domains

Attainable memory bandwidth: Comparing architectures



Intel Sandy Bridge

AMD Interlagos

ECC=on

Intel Xeon Phi 5110P

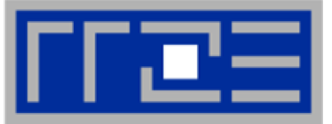
ECC=on

NVIDIA K20

2-socket CPU node

Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache



Bandwidth [GByte/s]

300
250
200
150
100
50
0

Westmere (L3)
Sandy Bridge (L3)
Interlagos (L3)
Interlagos (L2)

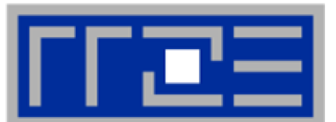
Intel SB:
New scalable
L3 design

AMD:
Optimize for L2 cache!

1 NUMA domain
2 NUMA domains

0 4 8 12 16

cores



OpenMP work sharing in the benchmark loop

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

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

Implicit barrier

OpenMP vector triad on Sandy Bridge socket (3 GHz)

Performance [GFlop/s]

Loop length

- T=1
- T=8 (1 socket)
- T=16 (2 sockets)

L1 core limit

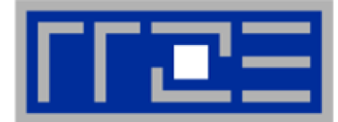
sync overhead grows with # of threads

bandwidth scalability across memory interfaces

OpenMP performance issues on multicore

Synchronization (barrier) overhead

Welcome to the multi-/many-core era
Synchronization of threads may be expensive!



```
!$OMP PARALLEL …
…
!$OMP BARRIER
!$OMP DO
…
!$OMP ENDDO
!$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP progams.

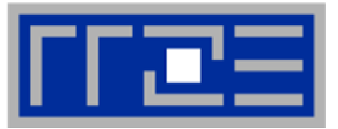
Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization!

- **Next slide: Test OpenMP Barrier performance…**
- **for different compilers**
- **and different topologies:**
  - **shared cache**
  - **shared socket**
  - **between sockets**
- **and different thread counts**
  - **2 threads**
  - **full domain (chip, socket, node)**

Thread synchronization overhead on SandyBridge-EP

Barrier overhead in CPU cycles

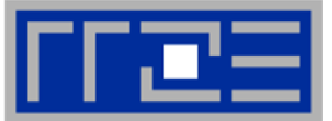


2 Threads	Intel 13.1.0	GCC 4.7.0	GCC 4.6.1
Shared L3	**384**	5242	4616
SMT threads	**2509**	3726	3399
Other socket	1375	5959	4909

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

Thread synchronization overhead on Intel Xeon Phi

Barrier overhead in CPU cycles



2 threads on distinct cores: 1936

	SMT1	SMT2	SMT3	SMT4
One core	n/a	**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

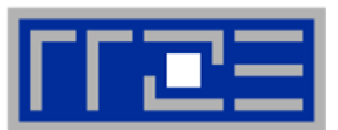
2 x more operations per cycle on Phi

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 x 7.5 = 20x more pain ☺.

Conclusions from the microbenchmarks



- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is

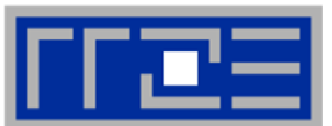
- **Bandwidth bottlenecks are ubiquitous**

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

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

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

Case study: Sparse matrix-vector multiply



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

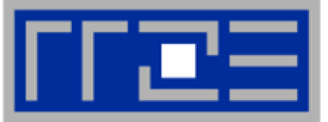
```
!$OMP parallel do
do i = 1,Nr
 do j = row_ptr(i), row_ptr(i+1) - 1
  c(i) = c(i) + val(j) * b(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

  - Usually many spMVMs required to solve a problem

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

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node



- **Case 1: Large matrix**

Coherent HyperTransport (16x+8x)

cHT 16x

cHT 8x

Memory

Memory Interface

P

cant, 62451x62451, non-zero: 4007383

CRS-magnycours

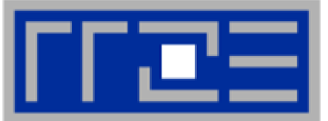
MFLOPS/s

threads

Intrasocket bandwidth bottleneck

Good scaling across NUMA domains

Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

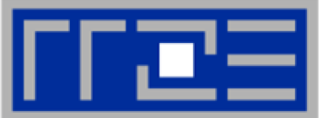


- **Case 2: Medium size**

Intrasocket bandwidth bottleneck

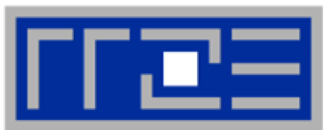
Working set fits in aggregate cache

mc2depi, 525825x525825, non-zero: 2100225

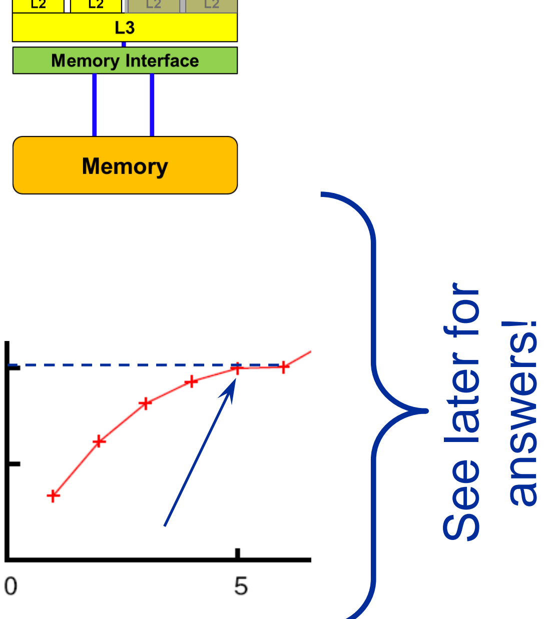


- **Case 3: Small size**

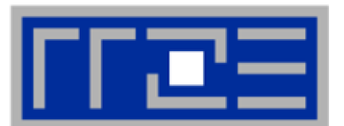
Conclusions from the spMVM benchmarks



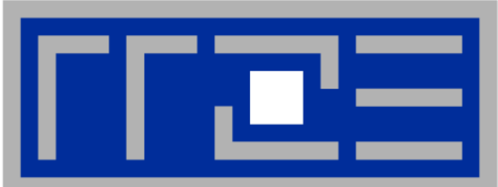
- **If the problem is "large", bandwidth saturation on the socket is a reality**
  - → There are "spare cores"
  - Very common performance pattern
- **What to do with spare cores?**
  - Let them idle → saves energy with minor loss in time to solution
  - Use them for other tasks, such as MPI communication
- **Can we predict the saturated performance?**
  - Bandwidth-based performance modeling!
  - What is the significance of the indirect access? Can it be modeled?
- **Can we predict the saturation point?**
  - … and why is this important?



Agenda



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



“Simple” performance modeling: The Roofline Model

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

The Roofline Model[1,2]

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

2. I = **Computational intensity ("work" per byte transferred)** over the slowest data path utilized ("the bottleneck")
   - Code balance $B_C = I^{-1}$

3. b_S = **Applicable peak bandwidth** of the slowest data path utilized

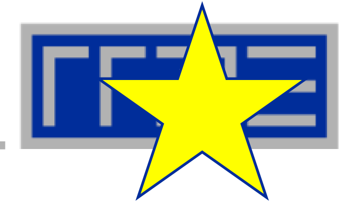
Expected performance:

[F/B] [B/s]

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

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

“Simple” Roofline: The vector triad



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

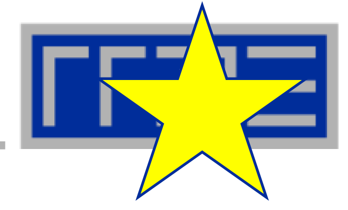
- b_S = **40 GB/s**
- B_c = (4+1) Words / 2 Flops = 2.5 W/F (including write allocate)

  → ***I* = 0.4 F/W = 0.05 F/B**

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

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

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



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

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

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

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

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

A not so simple Roofline example

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

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

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

ADD peak (best possible code)

no SIMD

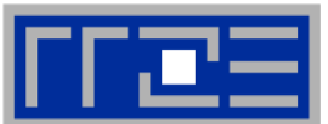
3-cycle latency per ADD if not unrolled

How do we get these? → See next!

I = 1 Flop / 8 byte (in DP)

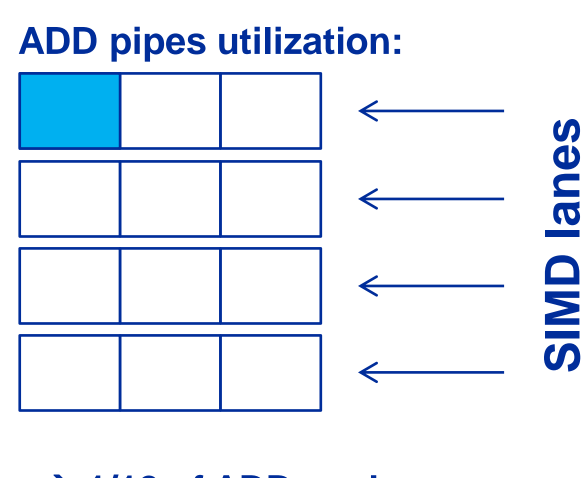
***P* = 5 Gflop/s**

Applicable peak for the summation loop

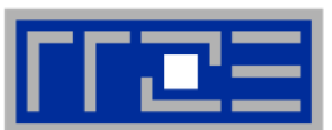


Plain scalar code, no SIMD

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



→ 1/12 of ADD peak



Scalar code, 3-way unrolling

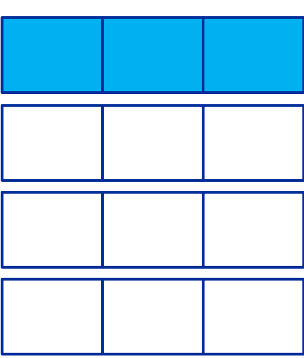
```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1

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

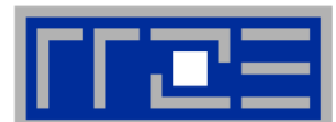
  i+=3 →? loop
result ← r1.0+r2.0+r3.0
```

ADD pipes utilization:



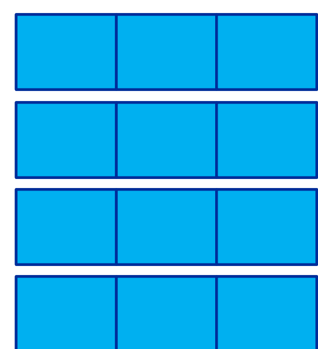
→ 1/4 of ADD peak

Applicable peak for the summation loop



SIMD-vectorized, 3-way unrolled

ADD pipes utilization:



→ ADD peak

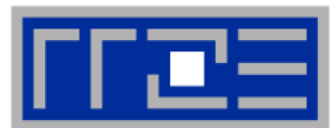
```
LOAD [r1.0,…,r1.3] ← [0,0]
LOAD [r2.0,…,r2.3] ← [0,0]
LOAD [r3.0,…,r3.3] ← [0,0]
i ← 1

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
```

Input to the roofline model



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

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

architecture

7.2 … 86.4 GF/s

Code analysis:
1 ADD + 1 LOAD

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

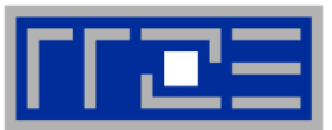
5 GF/s

measurement

analysis

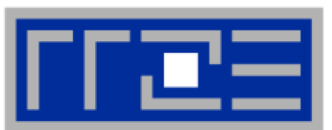
Maximum memory bandwidth 40 GB/s

Assumptions for the Roofline Model



- **The roofline formalism is based on some (crucial) assumptions:**
  - There is a clear concept of "work" vs. "traffic"
    - "work" = flops, updates, iterations…
    - "traffic" = required data to do "work"
  - Attainable bandwidth of code = input parameter! Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications
  - Data transfer and core execution overlap perfectly!
  - Slowest data path is modeled only; all others are assumed to be infinitely fast
  - If data transfer is the limiting factor, the bandwidth of the slowest data path can be utilized to 100% ("saturation")
  - Latency effects are ignored, i.e. perfect streaming mode

Factors to consider in the roofline model

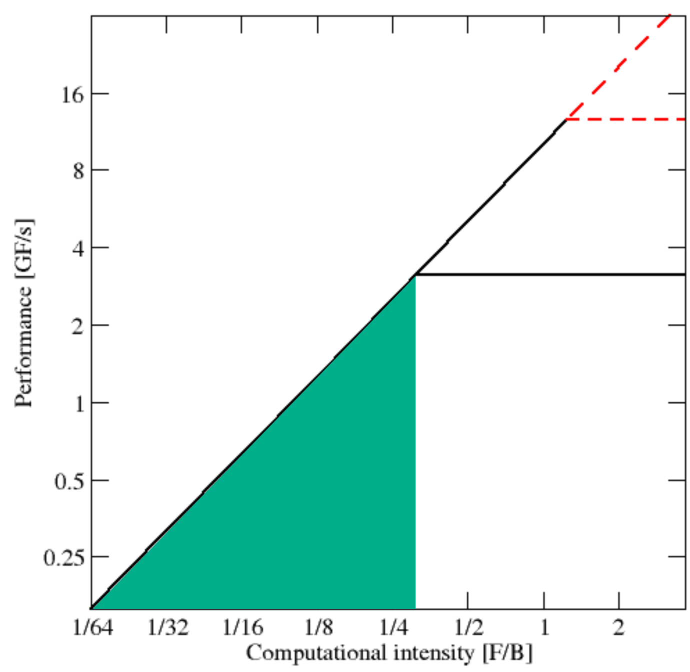


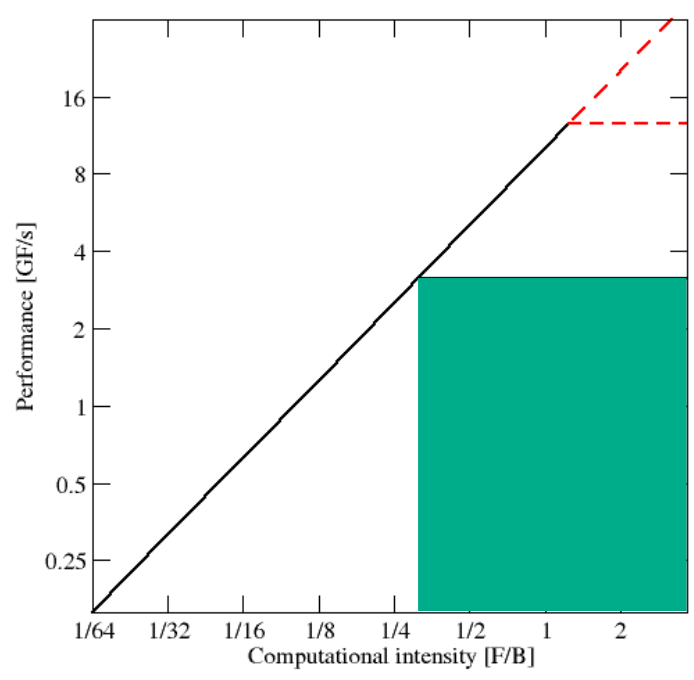
Bandwidth-bound (simple case)

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

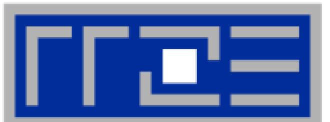
Core-bound (may be complex)

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





Complexities of in-core execution

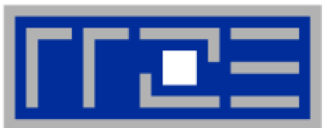


Multiple bottlenecks:

- L1 Icache (LD/ST) bandwidth
- Decode/retirement throughput
- Port contention (direct or indirect)
- Arithmetic pipeline stalls (dependencies)
- Overall pipeline stalls (branching)
- L1 Dcache bandwidth (LD/ST throughput)
- Scalar vs. SIMD execution
- …

- Register pressure
- Alignment issues

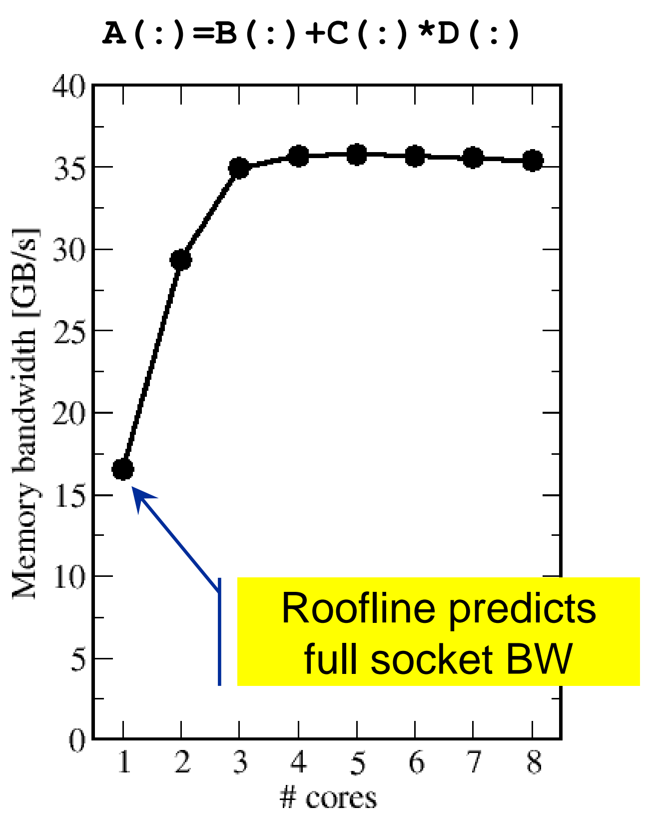
L1 Icache
Decoder
Decoder
Decoder
Decoder
Register file
Reorder buffer / Register renaming
Scheduler
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
ALU
MUL
DIV
ALU
ADD
LOAD
LOAD
ADRS
STORE
ADRS
ALU
JMP
L1 Dcache
Memory control
Data flow
Control flow
Pot. bottleneck

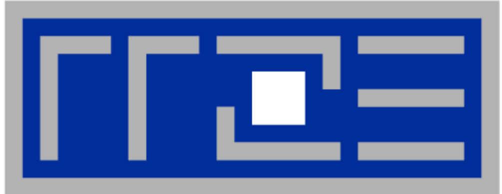


Shortcomings of the roofline model

- **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 (see later)**

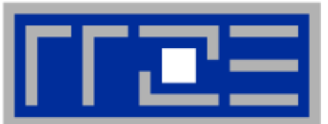




Case study: OpenMP-parallel sparse matrix-vector multiplication (part 2)

Putting Roofline to use where it should not work

Example: SpMVM node performance model



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

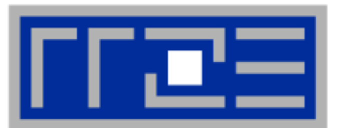
```
do i = 1,Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

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

$$I_{\mathrm{CRS}} = \frac{2}{12 + 24/N_{\mathrm{nzr}} + \kappa} \frac{\mathrm{Flops}}{\mathrm{Byte}} = \left(6 + \frac{12}{N_{\mathrm{nzr}}} + \frac{\kappa}{2}\right)^{-1} \frac{\mathrm{Flops}}{\mathrm{Byte}}$$

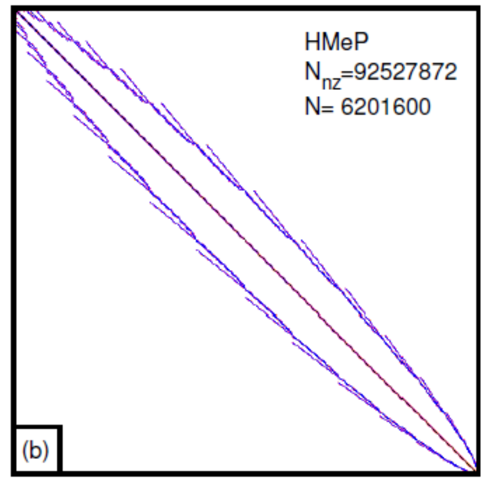
  - Expected performance = b_S x I_{CRS}
  - Determine κ by measuring performance and actual memory bandwidth
    - Maximum memory BW may not be achieved with spMVM

Roofline analysis for spMVM



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

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



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

… on the example of spMVM with HMeP matrix

Throughput: 1 ADD, 1 MULT + 1 LD + 1ST/cy

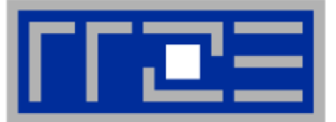
Code analysis: 1 ADD, 1 MULT, $(2.5+2/N_{nzr})$ LOADs, $1/N_{nzr}$ STOREs + κ

Memory-bound!
$\kappa = 2.5$

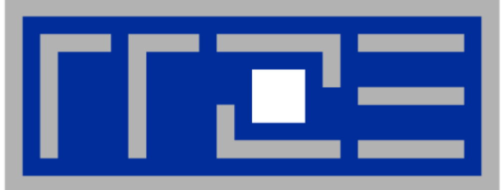
Measured memory BW for spMVM 18.1 GB/s

Measured performance for spMVM 2.25 GF/s

Maximum memory bandwidth 20 GB/s



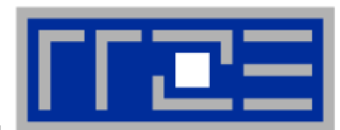
DEMO



Case study: A 3D Jacobi smoother

The basics in two dimensions

Roofline performance analysis and modeling



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

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

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

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

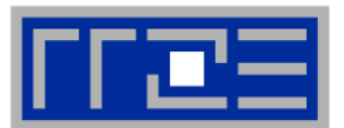
WRITE ALLOCATE:
LD + ST `phi(i,k,t1)`

Naive balance (incl. write allocate):

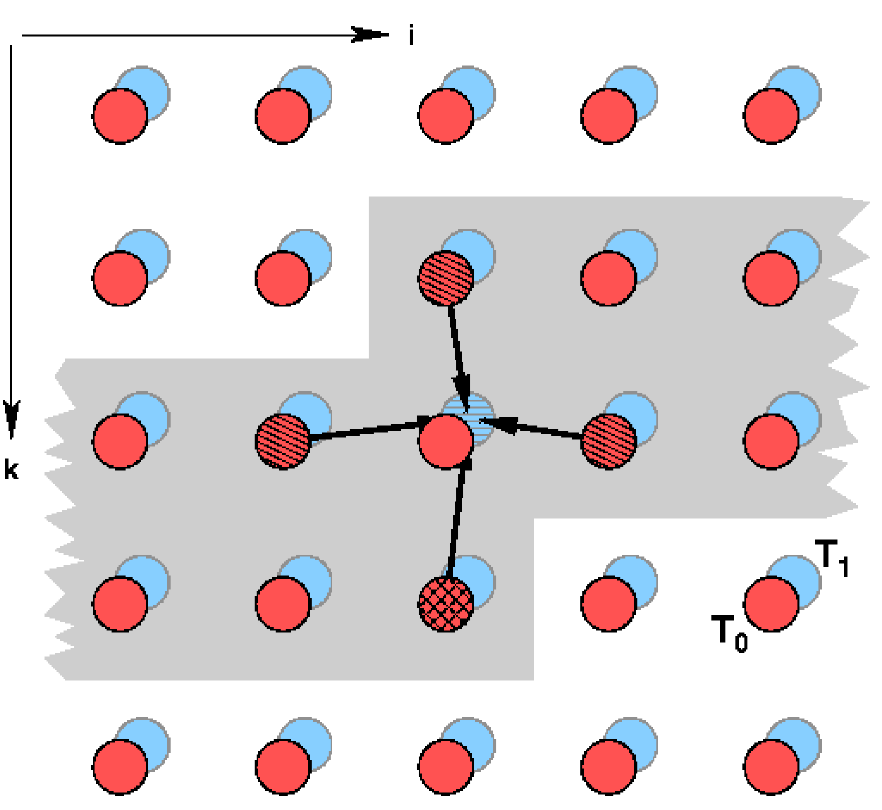
`phi(:,:,t0)`: 3 LD +
`phi(:,:,t1)`: 1 ST+ 1LD

→ B_C = 5 W / 4 FLOPs = 1.25 W / F

Balance metric: 2 D Jacobi



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



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

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

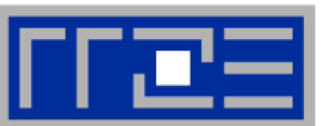
→B_C = 3 W / 4 F = 0.75 W / F

If cache is too small to hold one row:

`phi(:,:,t0)`: **2 LD +** `phi(:,:,t1)`: **1 ST+ 1LD**

→B_C = 5 W / 4 F = 1.25 W / F

Performance metrics: 2D Jacobi



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

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

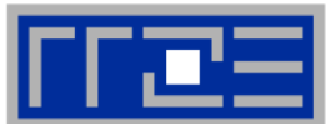
- **MFlops/sec increases by 7/4 but time to solution remains the same**

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

  2D Jacobi example: Compute LUPs/sec metric via

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

2D → 3D



- **3D sweep:**

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

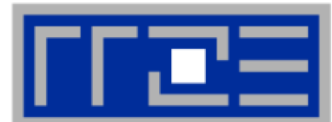
- **Best case balance:** 1 LD `phi(i,j,k+1,t0)`
  1 ST + 1 write allocate `phi(i,j,k,t1)`
  6 flops

  → B_C = 0.5 W/F (24 bytes/LUP)

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

3D Jacobi solver

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



Problem size: N^3

T=8 no blocking
T=1 no blocking

Performance [MLup/s]

6000
5000
4000
3000
2000
1000

cache

memory

2 layers of source array drop out of L3 cache

24 B/update model

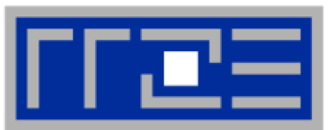
40 B/update model

Roofline inappropriate for unsaturated case

100
1000

Linear problem size [sites]

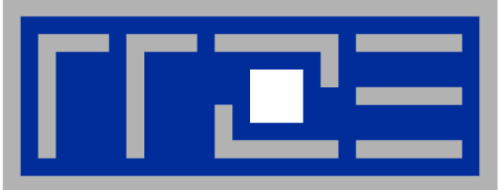
Conclusions from the Jacobi example



- **We have made sense of the memory-bound performance vs. problem size**
  - "Layer conditions" lead to predictions of code balance
  - Achievable memory bandwidth is input parameter

- **The model works only if the bandwidth is "saturated"**
  - In-cache modeling is more involved

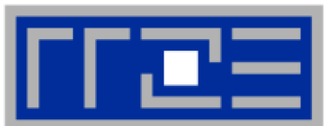
- **Optimization == reducing the code balance by code transformations**
  - See below



Data access optimizations

Case study: Optimizing the 3D Jacobi solver

Remember the 3D Jacobi solver on Sandy Bridge?



Problem size: N^3

T=8 no blocking

T=1 no blocking

Performance [MLup/s]

6000
5000
4000
3000
2000
1000
0

100
1000

Linear problem size [sites]

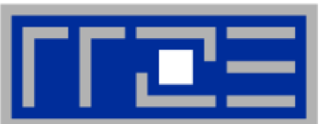
2 layers of source array drop out of L3 cache

→ Avoid through spatial blocking!

24 B/update model

40 B/update model

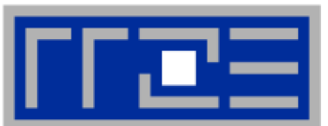
Jacobi iteration (2D): No spatial blocking



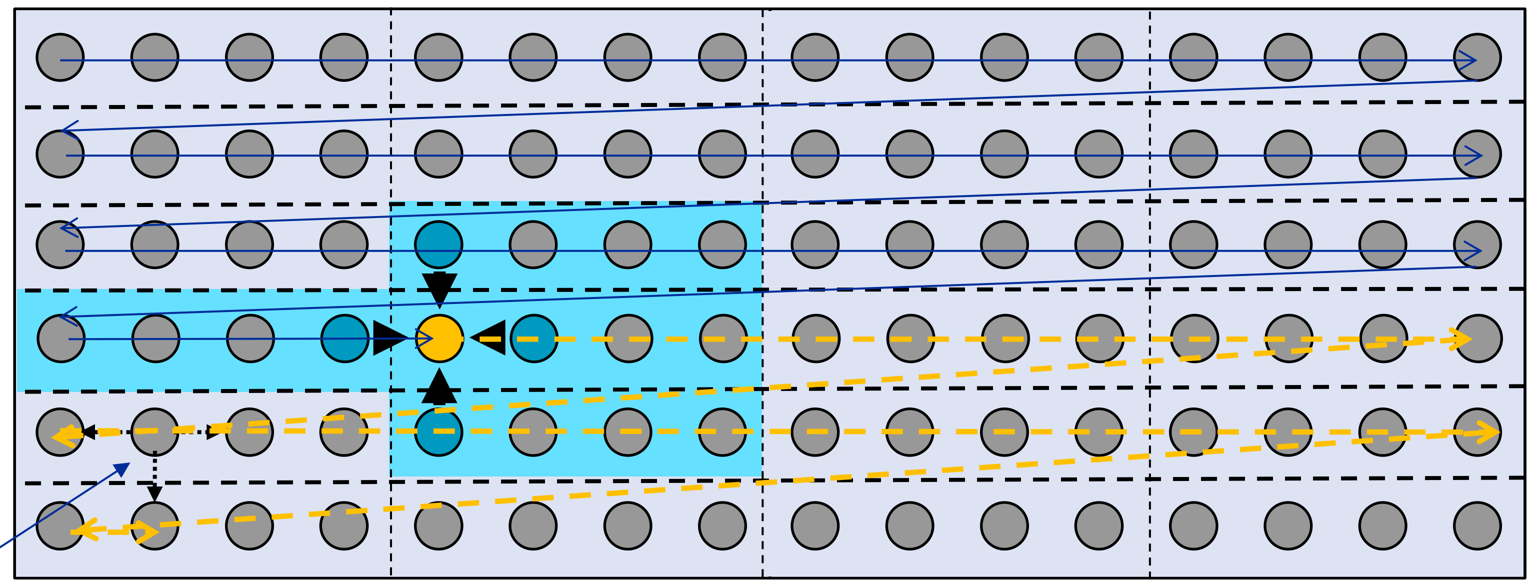
- **Assumptions:**
  - cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array

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

Jacobi iteration (2D): No spatial blocking

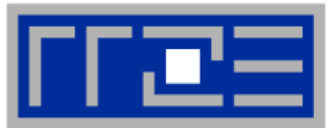


- **Assumptions:**
  - cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array

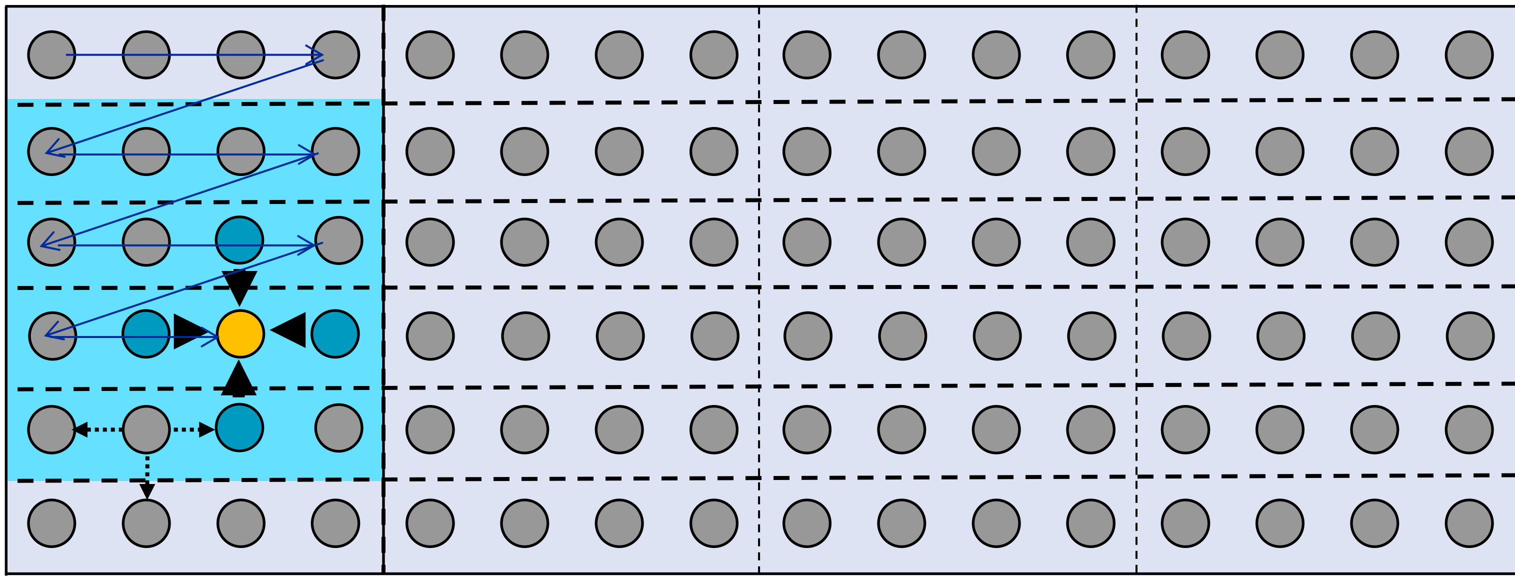


This element is needed for three more updates but has been evicted

Jacobi iteration (2D): Spatial blocking

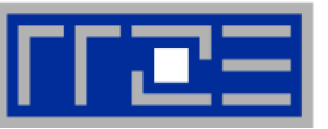


- Divide system into blocks
- Update block after block
- Same performance as if three complete rows of the systems fit into cache



- Some excess traffic at boundaries may be unavoidable

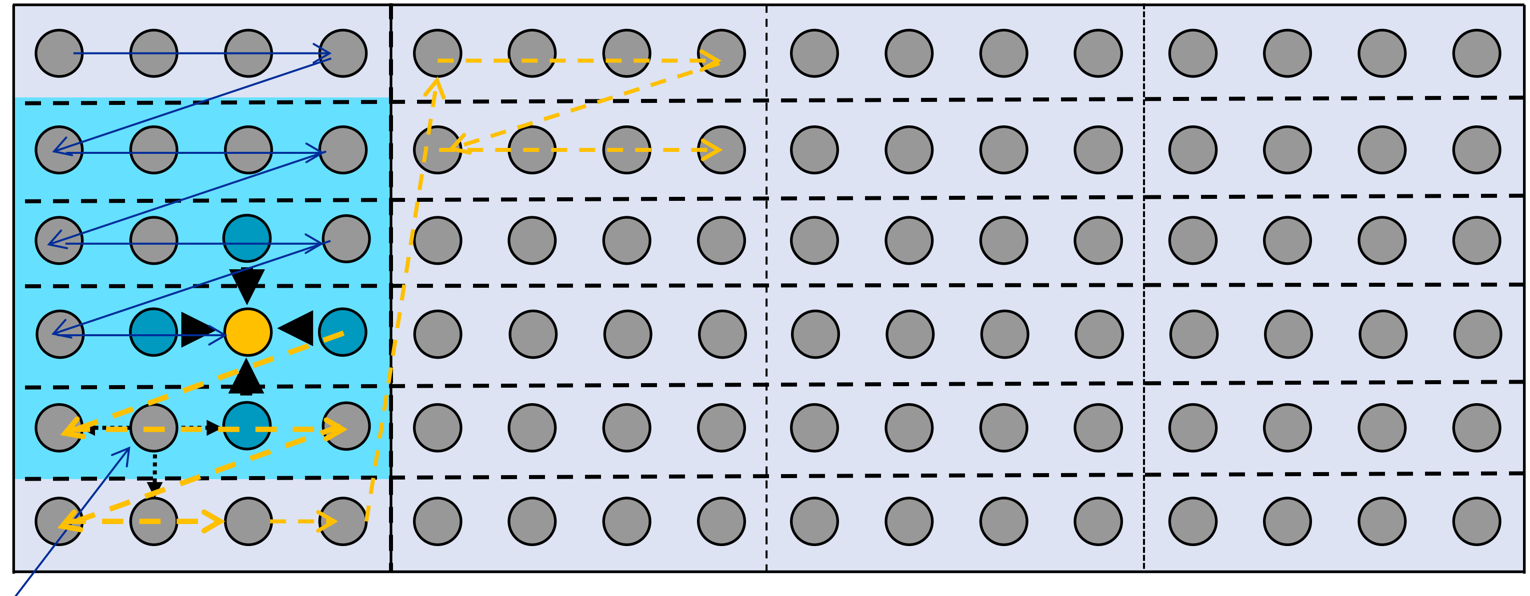
Jacobi iteration (2D): Spatial blocking



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

→Elements stay sufficiently long in cache to be fully reused

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



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

Jacobi iteration (3D): Spatial blocking

- **Implementation:**

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

loop over i-blocks

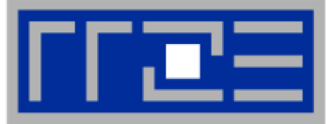
loop over j-blocks

2 · iblock · jblock · 8 byte · #cores < (cache size)/2

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

3D Jacobi solver (problem size 500^3)

Blocking different loop levels (8 cores Sandy Bridge)



optimum j block size

24B/update performance model

40B/update performance model

middle (j) loop blocking

inner (i) loop blocking

j blocking
i blocking
i & j blocking

Performance [MLup/s]

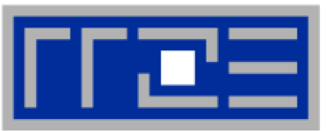
Blocking factor

0
200
400
600
800
1000
1200
1400

0
100
200
300
400
500

OpenMP parallelization?
Optimal block size?
k-loop blocking?
ccNUMA page placement?

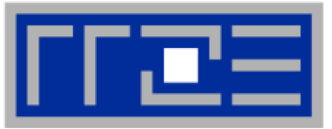
Jacobi iteration (3D): Nontemporal stores



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

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

Jacobi iteration (3D): Nontemporal stores for Intel



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

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

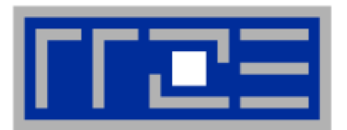
Main loop:

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

    enddo
  enddo
enddo
```

3D Jacobi solver

Spatial blocking + nontemporal stores



16 B/update perf. model

NT stores

blocking

Performance [MLup/s]

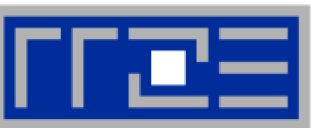
Linear problem size [sites]

T=8 no blocking

T=8 b_j=80

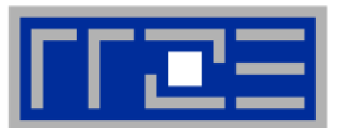
T=8 b_j=80 NT

Conclusions from the Jacobi optimization example



- **"What part of the data comes from where" is a crucial question**
- **Avoiding slow data paths == re-establishing the most favorable layer condition**
- **Improved code showed the speedup predicted by the model**
- **Optimal blocking factor can be estimated**
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of "optimization space"

Agenda



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

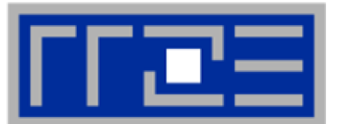
Optimal utilization of parallel resources

Exploiting SIMD parallelism and reading assembly code

Simultaneous multi-threading (SMT): facts & myths

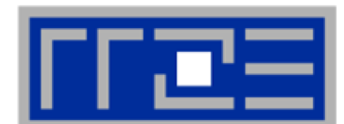
Programming for ccNUMA memory architecture

SIMD processing – Basics



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

SIMD processing – Basics



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

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

“Loop unrolling”

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

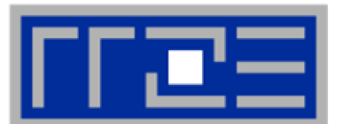
```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop handling
```

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

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

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

SIMD processing – Basics



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

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

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

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

  - C/C++ allows that `A` → `&C[-1]` and `B` → `&C[-2]`
    → `C[i] = C[i-1] + C[i-2]`: **dependency → No SIMD**

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

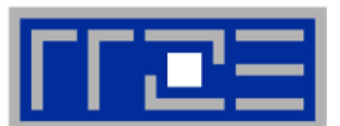
Reading x86 assembly code and exploting SIMD parallelism

Understanding SIMD execution by inspecting assembly code

SIMD vectorization how-to

Intel compiler options and features for SIMD

Sparse MVM part 3: SIMD-friendly data layouts



Why check the assembly code?

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

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

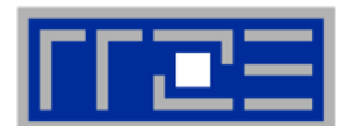
```
icc -S -O3  -xHost  triad.c  -o a.out
```

- **Disassemble Executable:**

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

The x86 ISA is documented in:

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



16 general Purpose Registers (64bit):

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

alias with eight 32 bit register set:

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

Floating Point SIMD Registers:

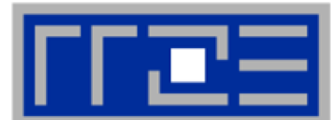
`xmm0-xmm15` **SSE (128bit) alias with 256-bit registers**

`ymm0-ymm15` **AVX (256bit)**

SIMD instructions are distinguished by:

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

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



```
float sum = 0.0;

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

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

Instruction code:

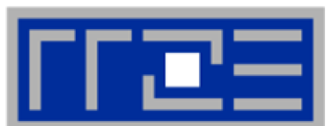
```
401d08:   f3 0f 58 04 82        addss   xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01           add     rax,1
401d11:   39 c7                 cmp     edi,eax
401d13:   77 f3                 ja      401d08
```

Instruction address

Opcodes

Assembly code

Summation code (single precision): Improvements



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

3 cycles add pipeline latency

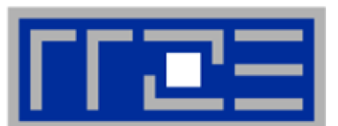
Unrolling with sub-sums to break up register dependency

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

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

AVX SIMD vectorization

How to leverage SIMD



Alternatives:

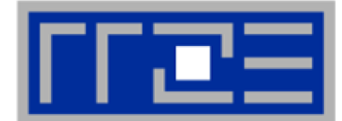
- The **compiler** does it for you (but: aliasing, alignment, language)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use intrinsics the following headers are available:

- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)

- `x86intrin.h` (all instruction set extensions)
- See next slide for an example

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



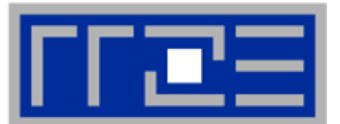
```
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 =  _mm_setzero_ps();
sum1 =  _mm_setzero_ps();
sum2 =  _mm_setzero_ps();
sum3 =  _mm_setzero_ps();

for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

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

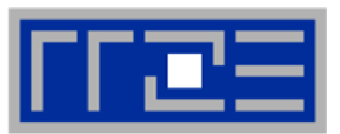
_mm_store_ss(&scalar_sum, sum0);
```

Example: array summation from intrinsics, instruction code



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

Loop body



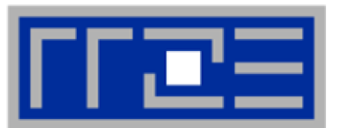
- **Intel compiler will try to use SIMD instructions when enabled to do so**
  - “Poor man’s vector computing”
  - Compiler can emit messages about vectorized loops (not by default):

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

  - Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
  - Some obstructions will prevent the compiler from applying vectorization even if it is possible

- **You can use source code directives to provide more information to the compiler**

Vectorization compiler options



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

  Available SIMD extensions:

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

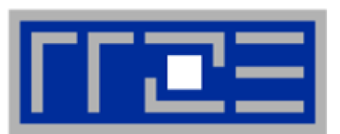
  - **`-xAVX`** on Sandy Bridge processors

  Recommended option:

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

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

Vectorization compiler options

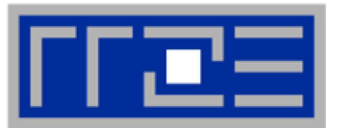


- **Controlling non-temporal stores (part of the SIMD extensions)**

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

`always`	use NT stores, assume application is memory bound (use with caution!)
`auto`	compiler decides when to use NT stores
`never`	do not use NT stores unless activated by source code directive

Rules for vectorizable loops



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

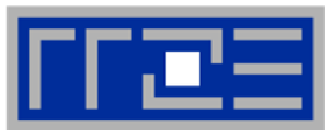
Better performance with:

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

Obstacles for vectorization:

- Non-contiguous memory access
- Data dependencies

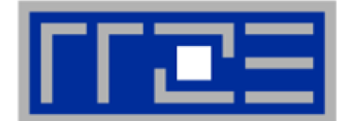
User mandated vectorization



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

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

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



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

- **Dynamic allocation of aligned memory (`align` = alignment boundary):**

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

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

Case study: OpenMP-parallel sparse matrix-vector multiplication (part 3)

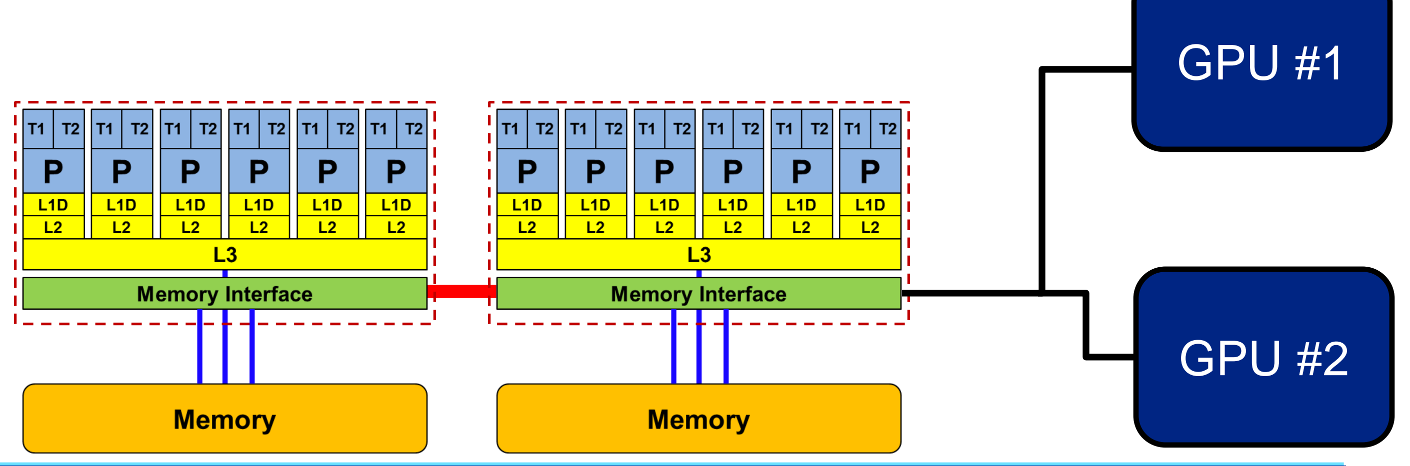
SIMD-friendly data layouts for sparse matrices

M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: A unified sparse matrix data format for modern processors with wide SIMD units. Submitted. Preprint: arXiv:1307.6209

Programming for heterogeneous systems: A unified code for CPU and Accelerators?

```
size_t i = get_global_id(0);
if (i < number_of_unknowns) {
   for(int j=row[i]; j<row[i+1]; ++j) {
      y[i] = y[i] + entry[j]*= x[column[j]];}}
```

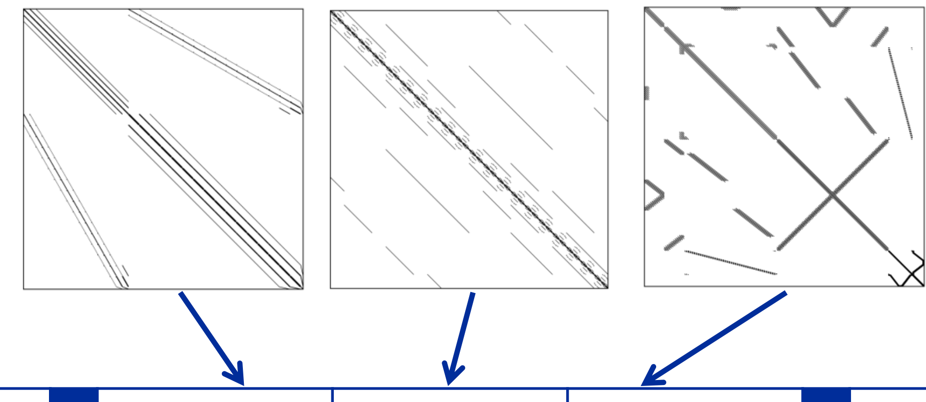
OpenCL

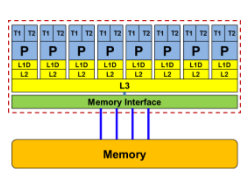



```
#pragma acc for
for(i = 0; i< number_of_unknowns; ++i){
    for(j = row(i); i < row(i+1);++j){
      y[i] =y[i] +entry[j] *x[column[j]];}}
```

Programming for heterogeneous systems: A unified code for CPU and Accelerators?

- **All kernels written in OpenCL / OpenMP**
- **Code portability is not the challenge!**



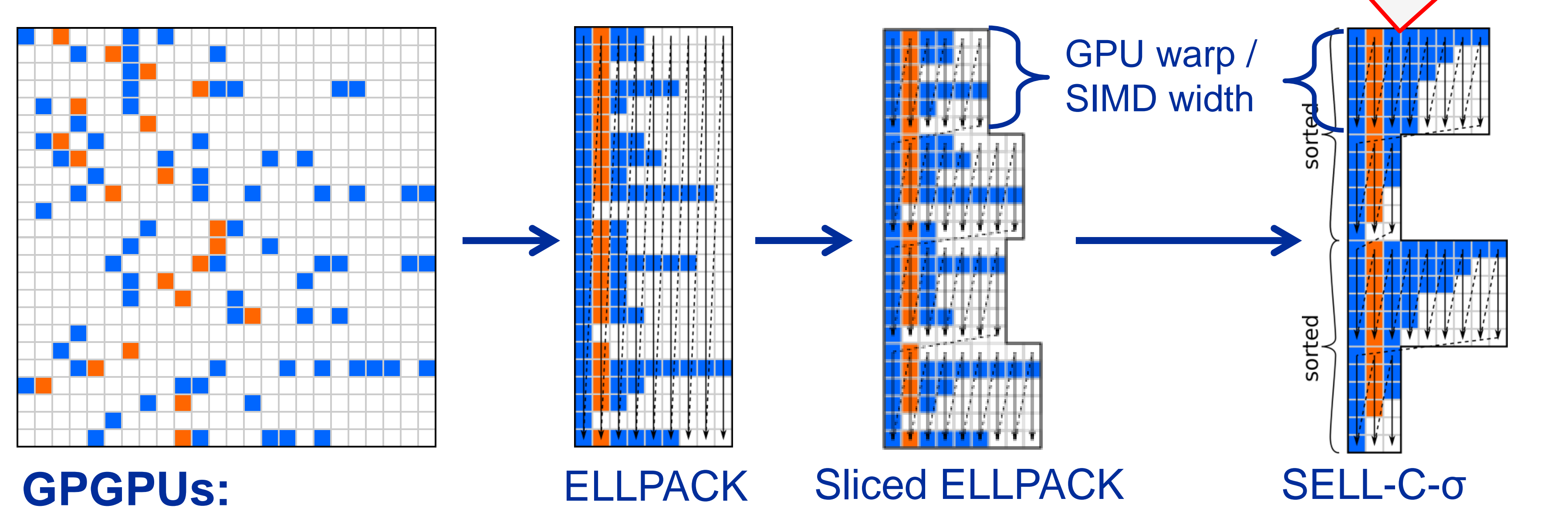


	Data format	dlr1	rrze3	RM07R	Rel. BW to 1 CPU
Intel Xeon E5-2690	CRS	**7.1 GF/s**	**5.3 GF/s**	**6.9 GF/s**	1
Tesla K20c (*Kepler*)	CRS	**1.3 GF/s**	**1.6 GF/s**	**1.8 GF/s**	**4**
Intel Xeon Phi 5110P	CRS	**18.9 GF/s**	**5.9 GF/s**	**16.9 GF/s**	**4**

Potential speed up based on memory bandwidth (BW)

The data format is the key to performance!

A unified data format for spMVM?!
GPGPU-friendly format (Sliced)ELLPACK



GPGPUs:

- Size of slices ~ warp sizes (`slice=32` rows)
- Padding of data structures for load coalescing
- Sort within blocks (multiple slices) according to nonzeros per row (JDS format – vector computers!) → reduce padding overhead

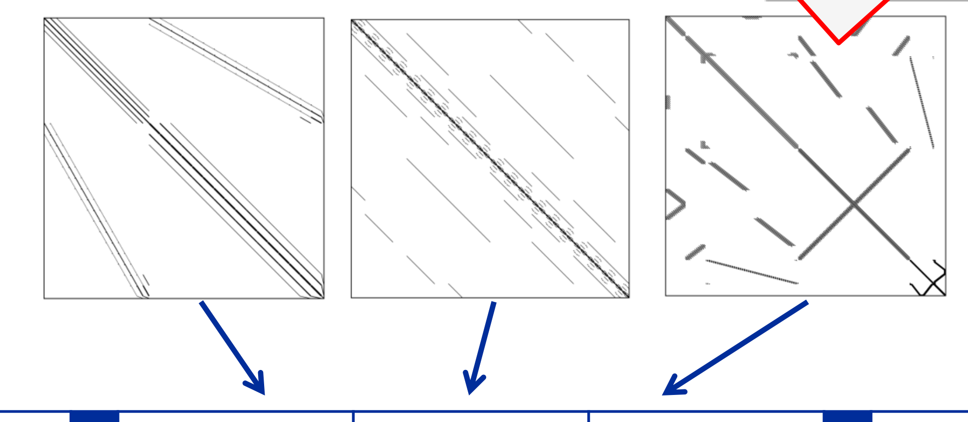
SIMD CPUs:

- Choose size of slices appropriately for x86 processors with SSE or AVX (`slice=4`) and Intel Xeon Phi (`slice=16`)

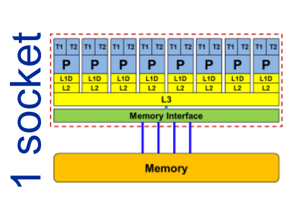
A unified data format CPU and Accelerators!

optional

- **Vectorizable code + vectorizable data structures are (often) beneficial for modern compute devices!**



1 socket

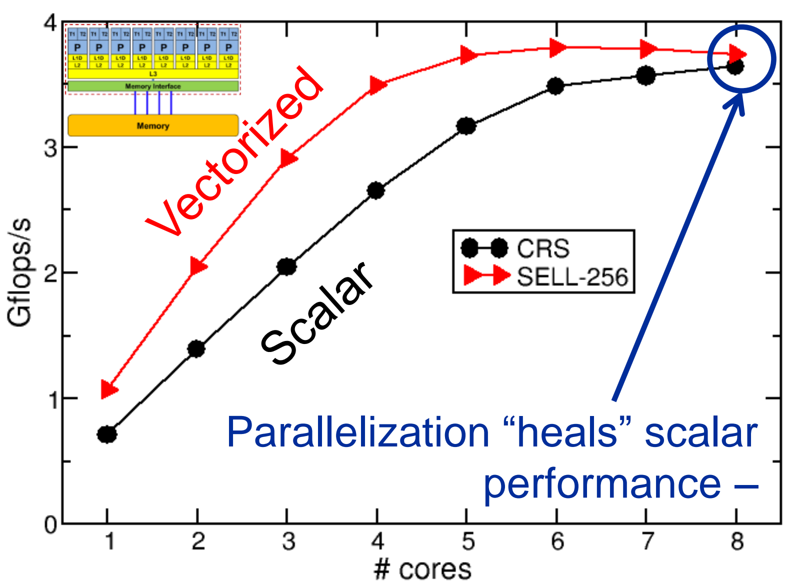


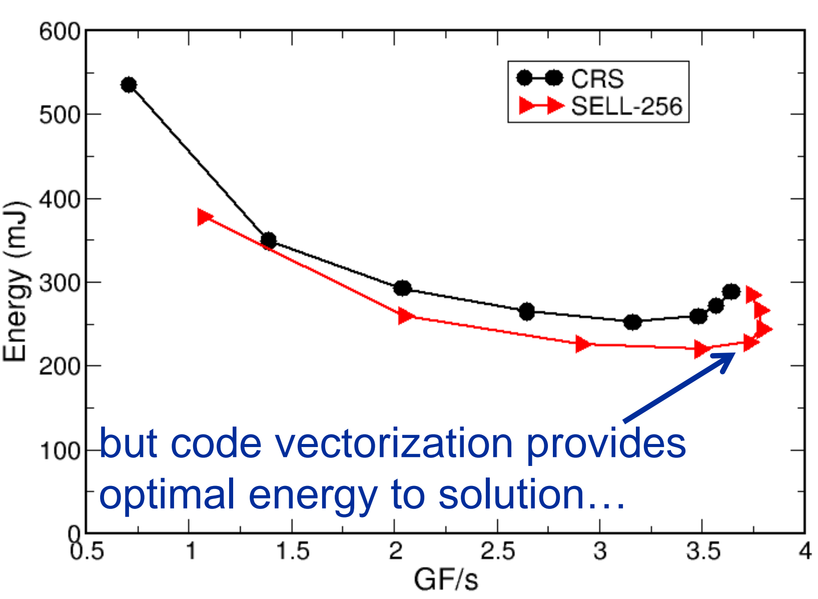
	Data format	dlr1	rrze3	RM07R	Rel. BW to 1 CPU
Intel Xeon E5-2690	CRS	7.1 GF/s	5.3 GF/s	6.9 GF/s	1
	SELL-256	**7.2 GF/s**	**5.3 GF/s**	**6.9 GF/s**	
NVIDIA Tesla K20	CRS	1.3 GF/s	1.6 GF/s	1.8 GF/s	**4**
	SELL-256	**23.0 GF/s**	**16.1 GF/s**	**21.0 GF/s**	
Intel Xeon Phi 5110P	CRS	18.9 GF/s	5.9 GF/s	16.9 GF/s	**4**
	SELL-256	**21.3 GF/s**	**13.5 GF/s**	**19.2 GF/s**	

→ **Speed-up of K20 or Phi vs. 2-socket CPU compute node ~ 1.5X**

Accelerators and SIMD CPUs:

- New frameworks / tools may provide code portability,…
- but portable performance will remain the challenge
- Back to the roots: Vectorized codes / data structures
- Memory bound codes: Vectorization ←→ Multicore parallel





Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes

First touch placement policy

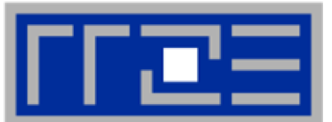
C++ issues

ccNUMA locality and dynamic scheduling

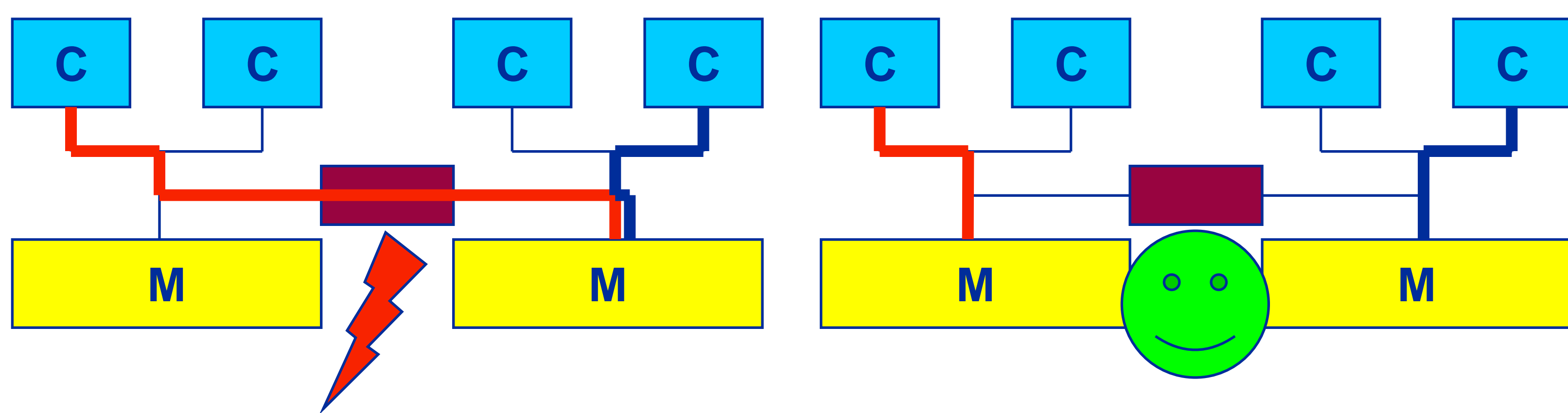
ccNUMA locality beyond first touch

ccNUMA performance problems

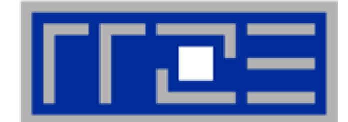
“The other affinity” to care about



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



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

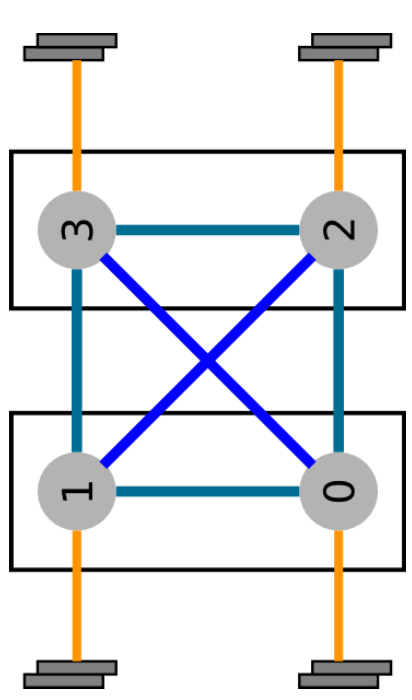


- **ccNUMA map: Bandwidth penalties for remote access**
  - Run 8 threads per ccNUMA domain (1 chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM triad benchmark using nontemporal stores

CPU node: 3, 2, 1, 0

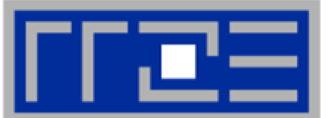
Memory node: 0, 1, 2, 3

STREAM triad performance [MB/s]: 13000, 12000, 11000, 10000, 9000, 8000, 7000, 6000, 5000



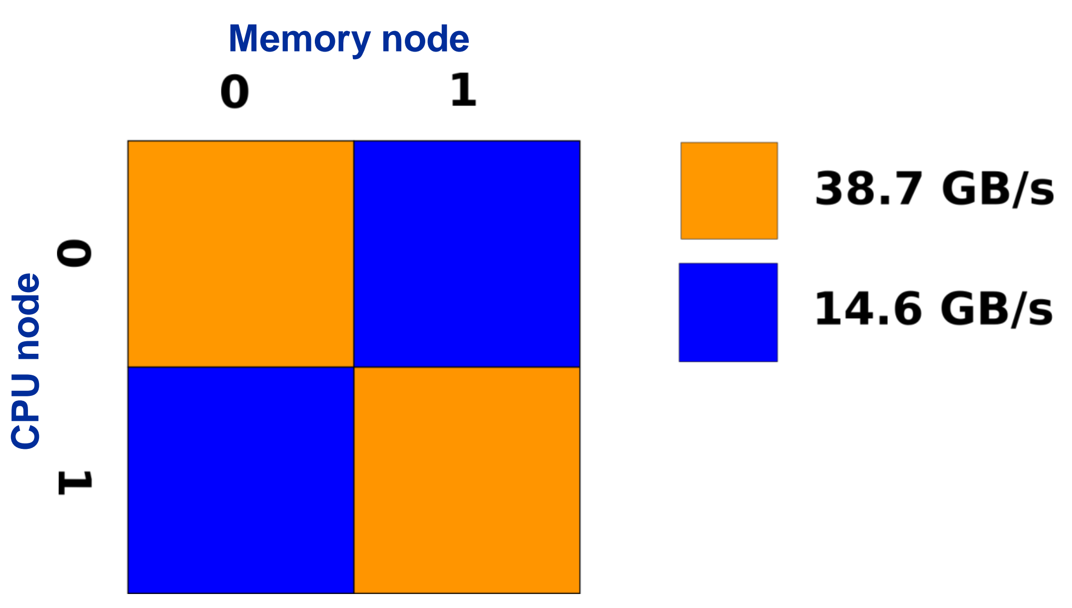
Intel Sandy Bridge 2-socket system

2 chips, 2 sockets, 8 threads per ccNUMA domain

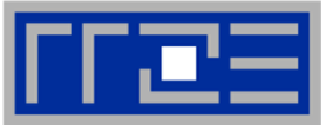


- **General rule:**

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



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



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

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

- Examples:

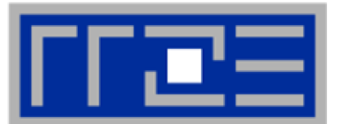
ccNUMA map scan

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

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

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

ccNUMA default memory locality



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

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

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

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

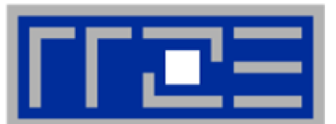
for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

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

Coding for ccNUMA data locality



- **Most simple case: explicit initialization**

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



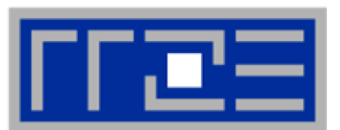

A=0.d0




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

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

Coding for ccNUMA data locality



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

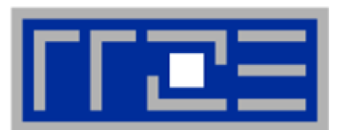
```
integer,parameter :: N=10000000
double precision A(N), B(N)

READ(1000) A

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

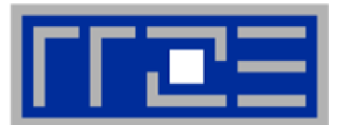
```
integer,parameter :: N=10000000
double precision A(N),B(N)
!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Coding for Data Locality



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

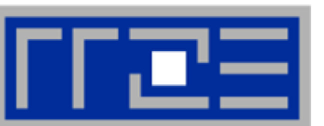
Diagnosing Bad Locality



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

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

Using performance counters for diagnosing bad ccNUMA access locality



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

Only one memory BW per socket ("Uncore")

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

Half of BW comes from other socket!

If all fails…

optional

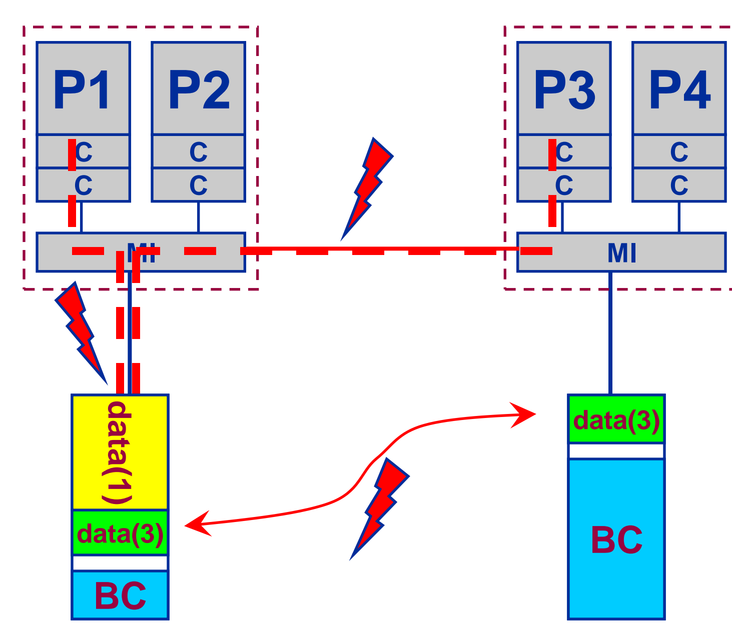
- **Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?**
  - Program has erratic access patters → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

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

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

ccNUMA problems beyond first touch:
Buffer cache

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



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

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

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

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

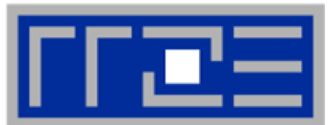
→ restrict to local domain if possible!

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

default placement
strictly local placement

Bandwidth [MByte/s]
File size [GByte]

ccNUMA placement and erratic access patterns



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

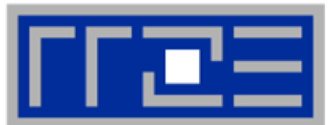
```
double precision :: r, a(M)
!$OMP parallel do private(r)
do i=1,N
   call RANDOM_NUMBER(r)
   ind = int(r * M) + 1
   res(i) = res(i) + a(ind)
enddo
!OMP end parallel do
```

- Or you have to use tasking/dynamic scheduling:

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

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

ccNUMA placement and erratic access patterns



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

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

Observe page alignment of array to get proper placement!

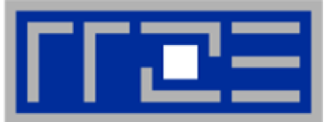
  2. Using global control via **`numactl`:**

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

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

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

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

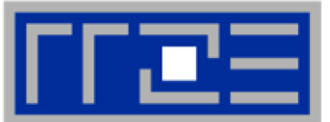


- **Parallel init: Correct parallel initialization**
- **LD0: Force data into LD0 via `numactl -m 0`**
- **Interleaved: `numactl --interleave <LD range>`**

Parallel placement
LD0 placement
Interleaved placement

Peformance [MBytes/s]
70000
60000
50000
40000
30000
20000
10000
0
1
2
3
4
# of locality domains

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



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

parallel init | LD0 | interleaved

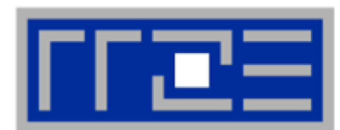
Bandwidth [Mbyte/s]

120000
100000
80000
60000
40000
20000
0

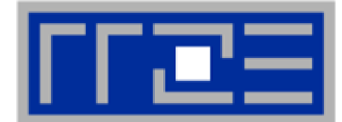
1 2 3 4 5 6 7 8

 # NUMA domains (6 threads per domain)

Summary on ccNUMA issues



- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with **`numactl --interleave`**
- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- **If dynamic scheduling cannot be avoided**
  - Consider round-robin placement
- **Buffer cache may impact proper placement**
  - Kick your admins
  - or apply sweeper code
  - If available, use runtime options to force local placement



DEMO

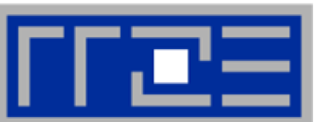
Simultaneous multithreading (SMT)

Principles and performance impact

SMT vs. independent instruction streams

Facts and fiction

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



- **SMT principle (2-way example):**

Standard core

Memory

L2 cache

L1D cache

L1I cache

Registers

Control

Execution units

2-way SMT

Memory

L2 cache

L1D cache

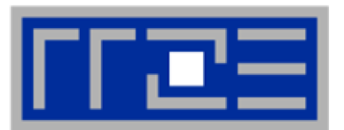
L1I cache

Registers

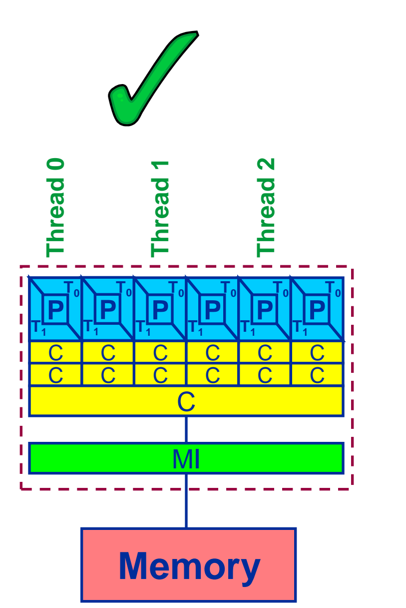
Control

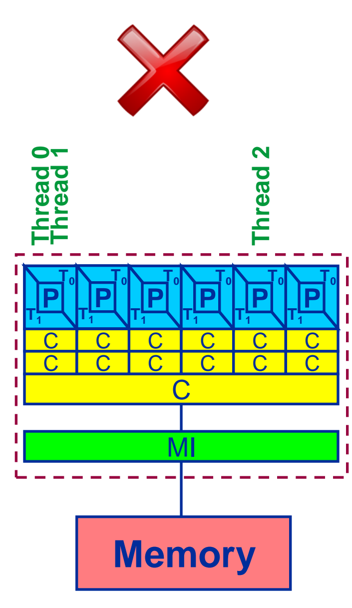
Execution units

SMT impact



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





SMT impact

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

Westmere EP

Memory

Thread 0:

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

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

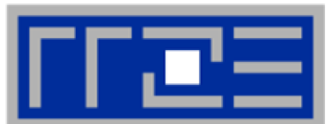
Dependency → pipeline stalls until previous MULT is over

Unrelated work in other thread can fill the pipeline bubbles

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

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

Simultaneous recursive updates with SMT



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

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

1 F/cycle
Full MULT pipeline
SMT
A(i)=A(i-1)*s A(i)=A(i-1)*s
B(i)=B(i-1)*s B(i)=B(i-1)*s
SMT
A(i)=A(i-1)*s A(i)=A(i-1)*s
1/5 F/cycle
A(i)=A(i-1)*s
MULT pipeline execution speed
MF/s
4000
3000
2000
1000
0
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7
N

Fill bubbles via:

- SMT
- Multiple streams

Thread 0:

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

Thread 1:

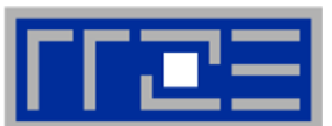
```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```

MULT pipe

B(7)*d
A(2)*c

A(7)*d
B(2)*c

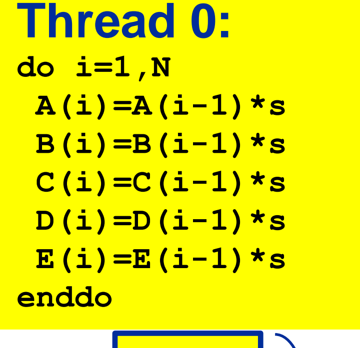
Simultaneous recursive updates with SMT

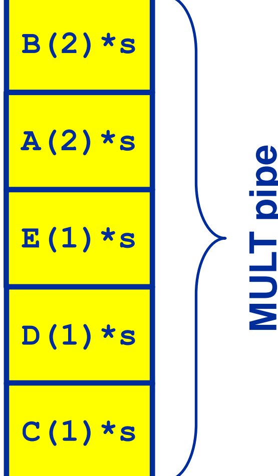


Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

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

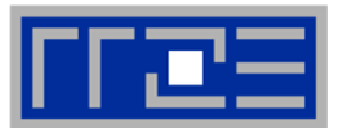
1 F/cycle
Full MULT pipeline
A(i)=A(i-1)*s
...
E(i)=E(i-1)*s
SMT
A(i)=A(i-1)*s A(i)=A(i-1)*s
B(i)=B(i-1)*s B(i)=B(i-1)*s
SMT
A(i)=A(i-1)*s A(i)=A(i-1)*s
1/5 F/cycle
A(i)=A(i-1)*s
MULT pipeline execution speed
MF/s
N
4000
3000
2000
1000
0
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7





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

Simultaneous recursive updates with SMT



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

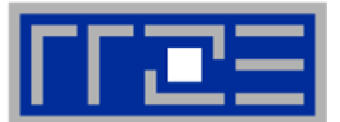
Pure update benchmark can be vectorized → 2 F / cycle (store limited)

7000
6000
5000
4000
3000
2000
1000
0
MF/s
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7
N
2 F/cycle
A(i)=A(i)*s [SIMD]
Full MULT pipeline
1 F/cycle
5 rec. streams
SMT: 2 rec. streams
SMT: 1 rec. stream
A(i)=A(i-1)*s
1/5 F/cycle
MULT pipeline execution speed

Recursive update:

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

SMT myths: Facts and fiction (1)



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

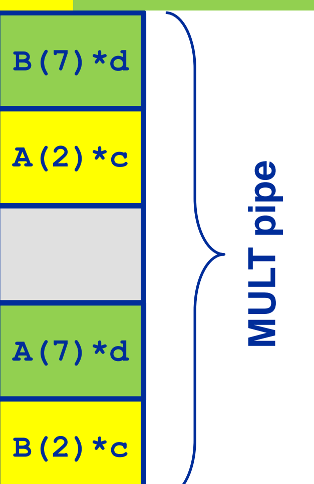
- **Truth**
  1. **A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.**
  2. **If a pipeline is already full, SMT will not improve its utilization**

Thread 0:

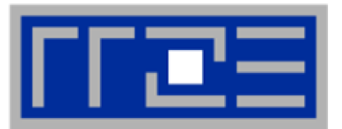
```
do i=1,N
A(i)=A(i-1)*c
B(i)=B(i-1)*d
enddo
```

Thread 1:

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



SMT myths: Facts and fiction (2)

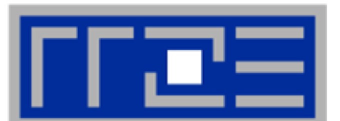


- **Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”**
- **Truth:**
  1. **If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.**
  2. **If the relevant bottleneck is not exhausted, SMT may help since it can fill bubbles in the LOAD pipeline.**

This applies also to other “relevant bottlenecks!”

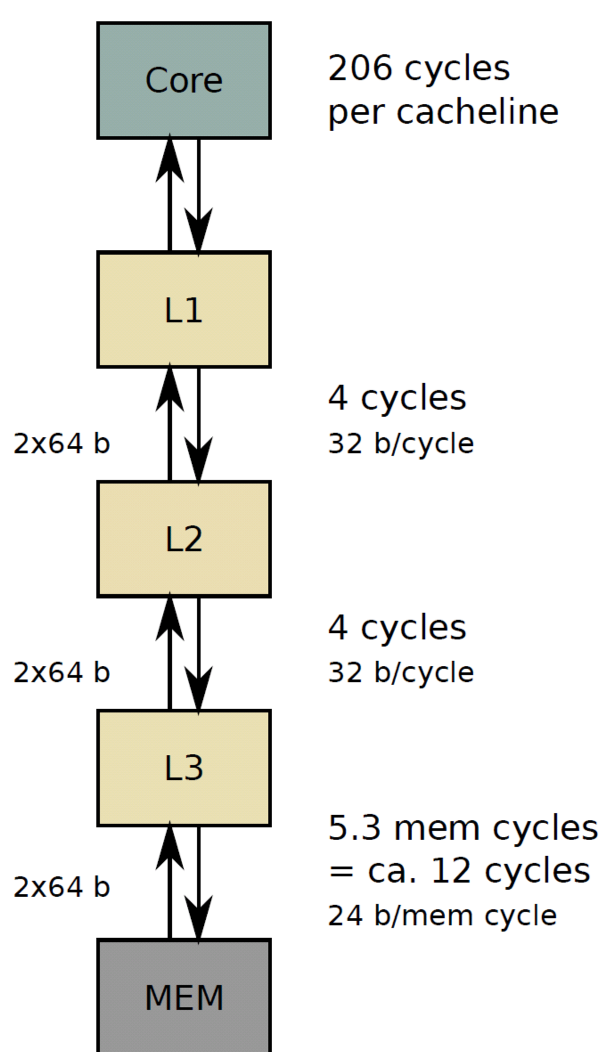
2 F/cycle
A(i)=A(i)*s [SIMD]
Full MULT pipeline
1 F/cycle
5 rec. streams
SMT: 2 rec. streams
SMT: 1 rec. stream
A(i)=A(i-1)*s
1/5 F/cycle
MULT pipeline execution speed
MF/s
N

SMT myths: Facts and fiction (3)

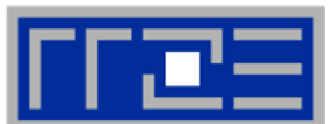


- **Myth: "SMT can help bridge the latency to memory (more outstanding references)."**

- **Truth:**
  Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets "wasted" in the cache hierarchy.



See also the "ECM Performance Model" later on.



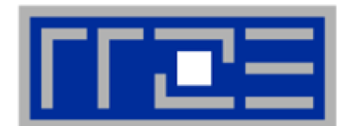
Goals for optimization:

1. Map your work to an instruction mix with highest throughput using the most effective instructions.

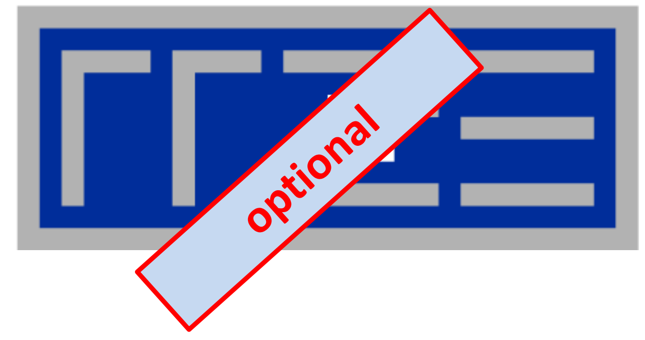
2. Reduce data volume over slow data paths fully utilizing available bandwidth.

3. Avoid possible hazards/overhead which prevent reaching goals one and two.

Agenda



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



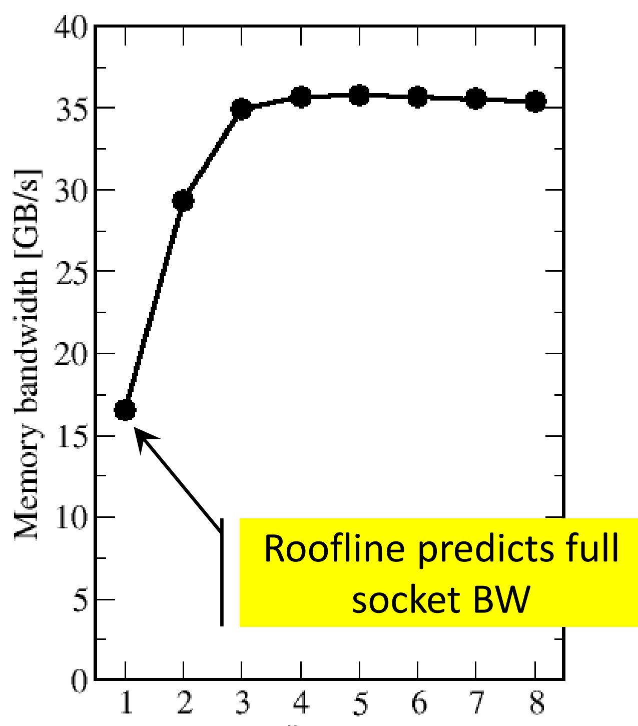
Multicore Scaling: The ECM Model

Improving the Roofline Model

See Poster →
"Pattern-Driven Node-Level Performance Engineering"
(Tomorrow 5:15pm – 7pm)

Assumptions and shortcomings of the roofline model

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

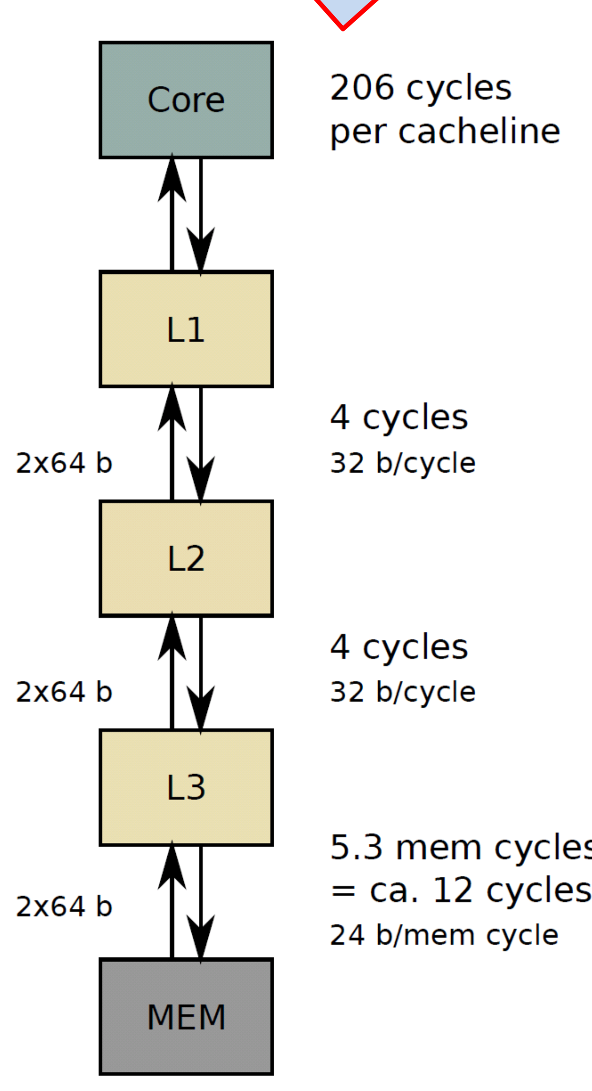


ECM Model

- **ECM = "Execution-Cache-Memory"**

- **Assumptions:**
- **Single-core execution time is composed of**
  1. In-core execution
  2. Data transfers in the memory hierarchy
- **Data transfers may or may not overlap with each other or with in-core execution**
- **Scaling is linear until the relevant bottleneck is reached**

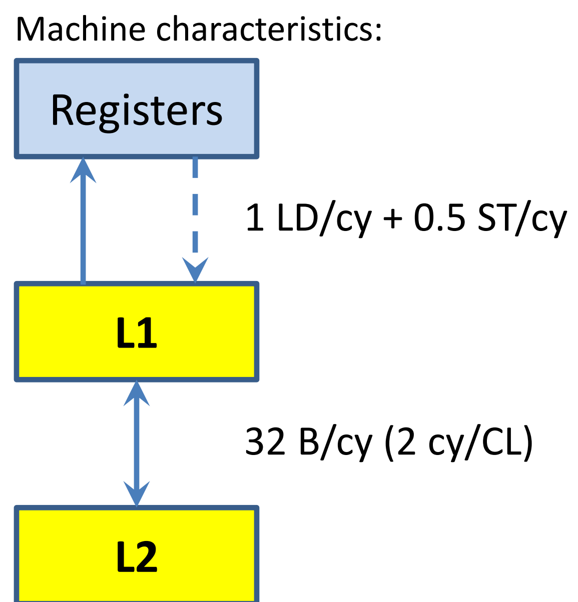
- **Input:**
- **Same as for Roofline**
- **+ data transfer times in hierarchy**

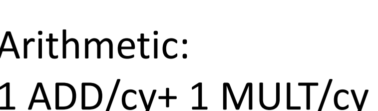


Example: Schönauer Vector Triad in L2 cache

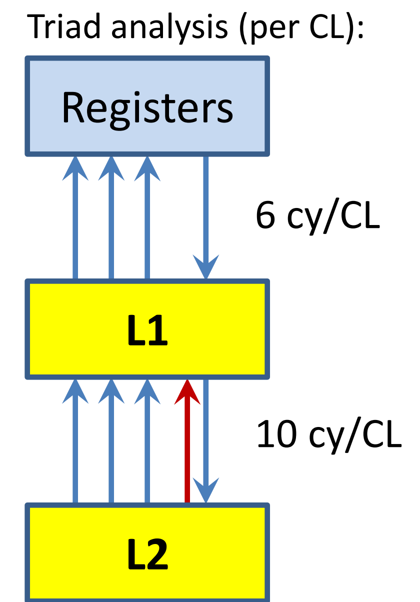
- REPEAT[`A(:) = B(:) + C(:) * D(:)`] @ double precision
- Analysis for Sandy Bridge core w/ AVX (unit of work: 1 cache line)

Machine characteristics:



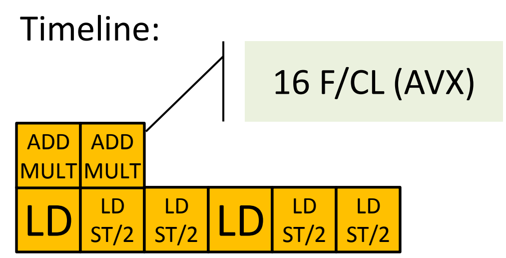


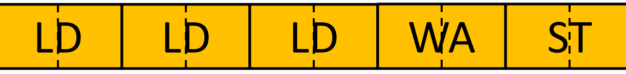
Triad analysis (per CL):



Arithmetic:
AVX: 2 cy/CL

Timeline:

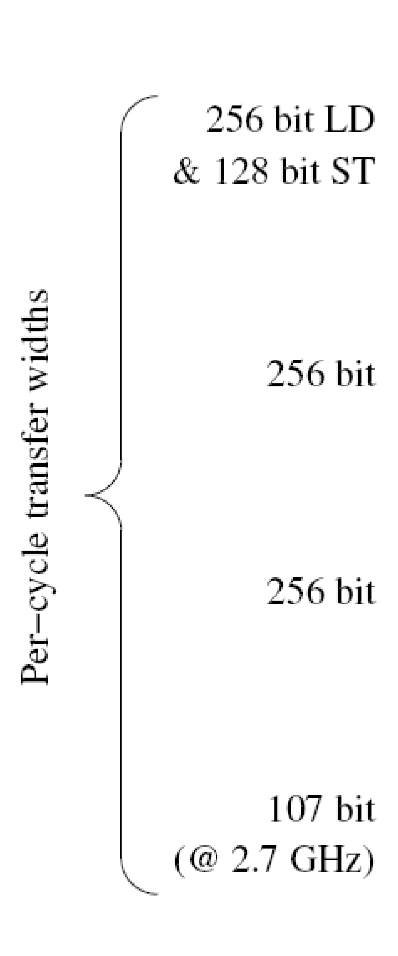


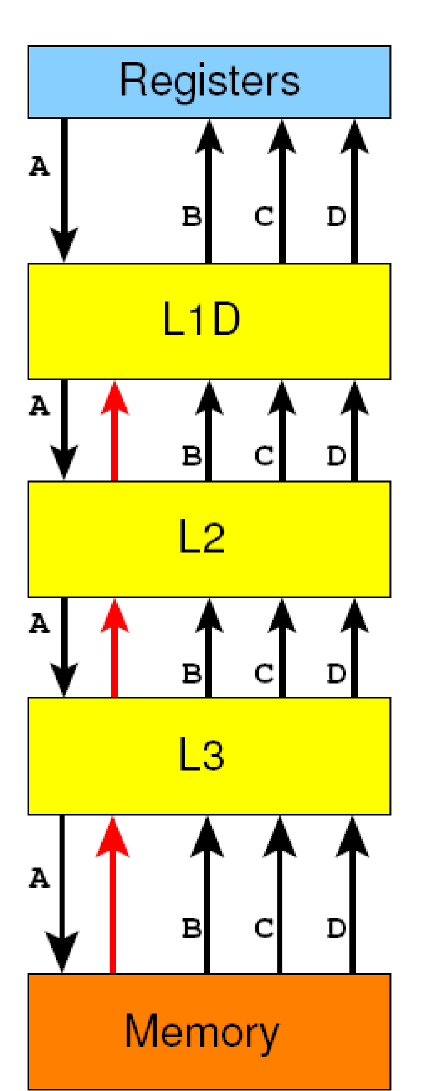


Roofline prediction: 16/10 F/cy

Measurement: 16F / ≈17cy

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





max(2(B) + 2(C) + 2(D), 4(A)) cy = 6 cy

(2(B) + 2(C) + 2(D) + 4(A)) cy = 10 cy

(2(B) + 2(C) + 2(D) + 4(A)) cy = 10 cy

(5 · 64 B · 2.7 Gcy/s) / (36 GB/s) = 24 cy

CL transfer

Write-allocate CL transfer

Full vs. partial vs. no overlap

No overlap | All caches single–ported | Full overlap beyond L2 | Measured

(a) No overlap: L1→R (0–6), L2→L1 (6–16), L3→L2 (16–26), Memory→L3 (26–50)

(b) All caches single–ported: L1→R, L3→L2, L2→L1, Memory→L3 (up to 34)

(c) Full overlap beyond L2: L1→R, L2→L1, L3→L2, Memory→L3 (up to 24)

cycles: 0, 6, 16, 20, 24, 26, 34, 50

(d) Measured: 6.04 L1, 17.2 L2, 26.3 L3, 52.3 Memory

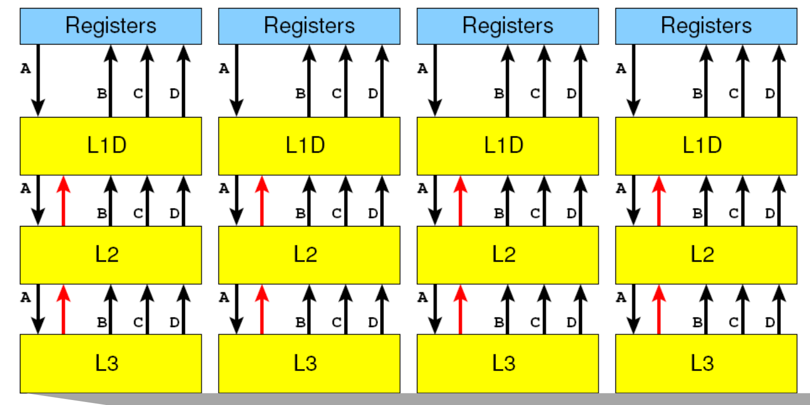
Results suggest no overlap!

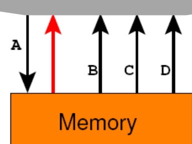
Multicore scaling in the ECM model

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

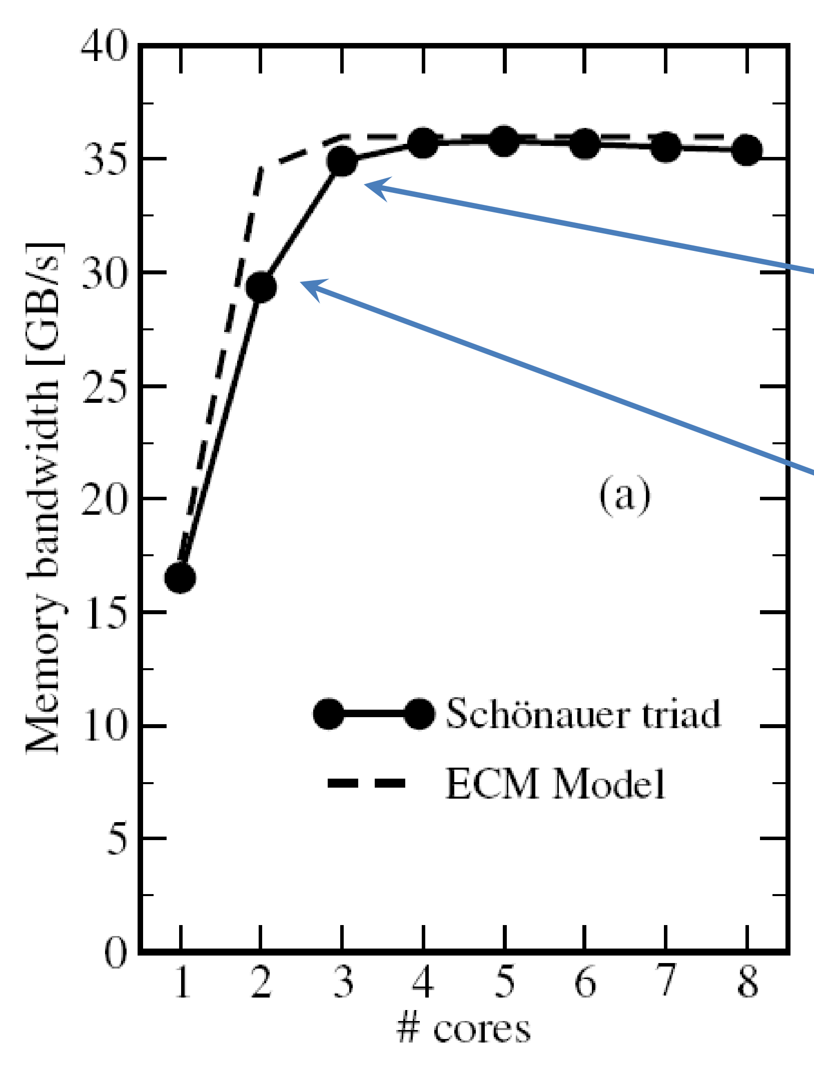
$$P(t) = \min(tP_0, P_{\text{roof}}), \text{ with } P_{\text{roof}} = \min(P_{\max}, I \cdot b_S)$$

Example: Scalable L3 on Sandy Bridge





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



Model: Scales until saturation sets in

Saturation point (# cores) well predicted

Measurement: scaling not perfect

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

Check: Use "overlappable" kernel code

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

Memory bandwidth [GB/s]

cores

(b)

DIV triad (AVX)
DIV triad (scalar)
ECM Model (AVX)
ECM Model (scalar)

In-core execution is dominated by divide operation

(44 cycles with AVX, 22 scalar)

→ Almost perfect agreement with ECM model

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

The impact of in-core optimizations

- **Remember the sequential vector triad?**

Performance [MFlops/s]

9000
8000
7000
6000
5000
4000
3000
2000
1000
0

10^2 10^3 10^4 10^5 10^6

Loop length

AVX — scalar

AVX: L1, L2, L3, Mem

scalar: L1, L2, L3, Mem

Less SIMD benefit
for far-away data
→ **"Amdahl's Law"!**

Summary: The ECM Model

optional

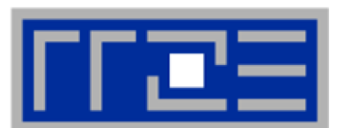
- **Saturation effects are ubiquitous; understanding them gives us opportunity to**
  - Find out about optimization opportunities
  - Save energy by letting cores idle → see power model later on
  - Putting idle cores to better use → asynchronous communication, functional parallelism

- **ECM correctly describes several effects**
  - Saturation for memory-bound loops
  - Diminishing returns of in-core optimizations for far-away data
  - Parallelism heals bad sequential code (sometimes…)

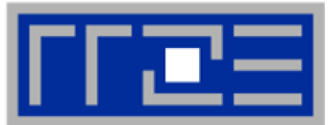
- **Simple models work best. Do not try to complicate things unless it is really necessary!**

- **Possible extensions to the ECM model**
  - Accommodate latency effects
  - Model simple "architectural hazards"

Tutorial conclusion



- **Multicore architecture == multiple complexities**
  - Affinity matters → pinning/binding is essential
  - Bandwidth bottlenecks → inefficiency is often made on the chip level
  - Topology dependence of performance features → know your hardware!
- **Put cores to good use**
  - Bandwidth bottlenecks → surplus cores → functional parallelism!?
  - Shared caches → fast communication/synchronization → better implementations/algorithms?
- **Simple modeling techniques help us**
  - … understand the limits of our code on the given hardware
  - … identify optimization opportunities
  - … learn more, especially when they do not work!
- **Simple tools get you 95% of the way**
  - e.g., LIKWID tool suite. Best tool: **your brain!**



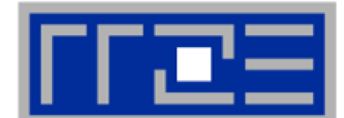
Moritz Kreutzer
Markus Wittmann
Thomas Zeiser
Michael Meier

OMI4papps
HSMB

THANK YOU.

hpcADD
FEPA
SKALB

Presenter Biographies

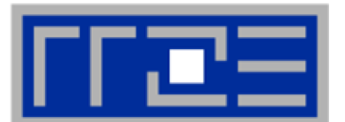


Georg Hager holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. See his blog at http://blogs.fau.de/hager for current activities, publications, and talks.

Jan Treibig holds a PhD in Computer Science from the University of Erlangen. He is now a postdoctoral researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE). His current research revolves around architecture-specific and low-level optimization for current processor architectures, performance modeling on processor and system levels, and programming tools. He is the developer of LIKWID, a collection of lightweight performance tools. In his daily work he is involved in all aspects of user support in High Performance Computing: training, code parallelization, profiling and optimization, and the evaluation of novel computer architectures.

Gerhard Wellein holds a PhD in solid state physics from the University of Bayreuth and is a professor at the Department for Computer Science at the University of Erlangen. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.

Abstract

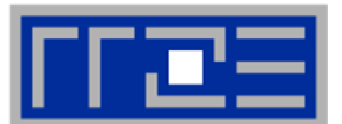


- **SC13 tutorial: The Practitioner's Cookbook for Good Parallel Performance on Multi- and Many-Core Systems**
- **Presenter(s): Georg Hager, Jan Treibig, Gerhard Wellein**

- **ABSTRACT:**

The advent of multi- and many-core chips has led to a further opening of the gap between peak and application performance for many scientific codes. This trend is accelerating as we move from petascale to exascale. Paradoxically, bad node-level performance helps to "efficiently" scale to massive parallelism, but at the price of increased overall time to solution. If the user cares about time to solution on any scale, optimal performance on the node level is often the key factor. Also, the potential of node-level improvements is widely underestimated, thus it is vital to understand the performance-limiting factors on modern hardware. We convey the architectural features of current processor chips, multiprocessor nodes, and accelerators, as well as the performance properties of the dominant MPI and OpenMP programming models, as far as they are relevant for the practitioner. Peculiarities like SIMD vectorization, shared vs. separate caches, bandwidth bottlenecks, and ccNUMA characteristics are introduced, and the influence of system topology and affinity on the performance of typical parallel programming constructs is demonstrated. Performance engineering is introduced as a powerful tool that helps the user assess the impact of possible code optimizations by establishing models for the interaction of the software with the hardware.

References



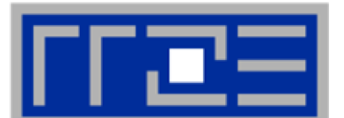
Books:

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

Papers:

- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: A unified sparse matrix data format for modern processors with wide SIMD units. Submitted. Preprint: arXiv:1307.6209
- G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Accepted for Computation and Concurrency: Practice and Experience. Preprint: arXiv:1208.2908
- J. Treibig, G. Hager and G. Wellein: Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. Preprint: arXiv:1206.3738
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann and A. R. Bishop: Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation. Workshop on Large-Scale Parallel Processing 2012 (LSPP12), DOI: 10.1109/IPDPSW.2012.211
- J. Treibig, G. Hager, H. Hofmann, J. Hornegger and G. Wellein: Pushing the limits for medical image reconstruction on recent standard multicore processors. International Journal of High Performance Computing Applications, (published online before print). DOI: 10.1177/1094342012442424

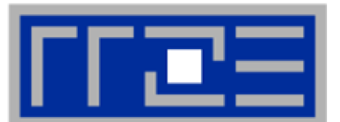
References



Papers continued:

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

References

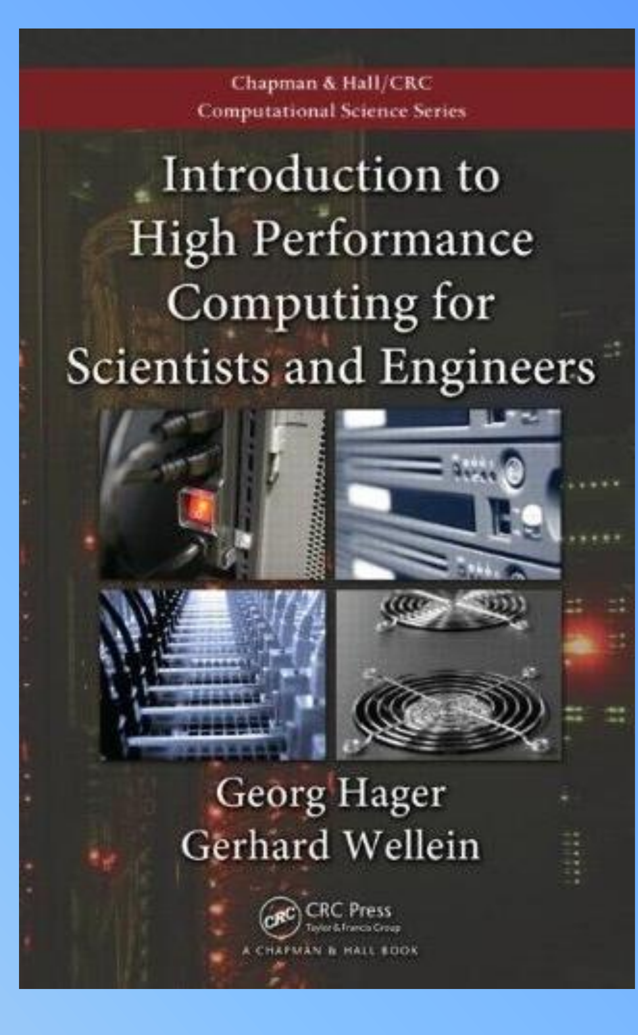


Papers continued:

- J. Habich, T. Zeiser, G. Hager and G. Wellein: Performance analysis and optimization strategies for a D3Q19 Lattice Boltzmann Kernel on nVIDIA GPUs using CUDA. Advances in Engineering Software and Computers & Structures 42 (5), 266–272 (2011). DOI: 10.1016/j.advengsoft.2010.10.007
- J. Treibig, G. Hager and G. Wellein: Multicore architectures: Complexities of performance prediction for Bandwidth-Limited Loop Kernels on Multi-Core Architectures. DOI: 10.1007/978-3-642-13872-0_1, Preprint: arXiv:0910.4865.
- G. Hager, G. Jost, and R. Rabenseifner: Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In: Proceedings of the Cray Users Group Conference 2009 (CUG 2009), Atlanta, GA, USA, May 4-7, 2009. PDF
- R. Rabenseifner and G. Wellein: Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. International Journal of High Performance Computing Applications **17**, 49-62, February 2003. DOI:10.1177/1094342003017001005

HPC textbook

Georg Hager and Gerhard Wellein:
Introduction to High Performance Computing for Scientists and Engineers



CRC Press, ISBN 978-1439811924
356 pages
July 2010

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

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

Georg Hager & Gerhard Wellein:

Introduction to High Performance Computing for Scientists and Engineers

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

Introduction to High Performance Computing for Scientists and Engineers

Contents

- **Modern Processors**
  - Stored-program computer architecture
  - General-purpose cache-based microprocessor architecture
  - Memory hierarchies
  - Multicore processors
  - Multithreaded processors
  - Vector processors
- **Basic Optimization Techniques for Serial Code**
  - Scalar profiling
  - Common sense optimizations
  - Simple measures, large impact
  - The role of compilers
  - C++ optimizations
- **Data Access Optimization**
  - Balance analysis and lightspeed estimates
  - Storage order
  - Case study: The Jacobi algorithm
  - Case study: Dense matrix transpose
  - Algorithm classification and access optimizations
  - Case study: Sparse matrix-vector multiply
- **Parallel Computers**
  - Taxonomy of parallel computing paradigms
  - Shared-memory computers
  - Distributed-memory computers
  - Hierarchical (hybrid) systems
  - Networks
- **Basics of Parallelization**
  - Why parallelize?
  - Parallelism
  - Parallel scalability
- **Shared-Memory Parallel Programming with OpenMP**
  - Short introduction to OpenMP
  - Case study: OpenMP-parallel Jacobi algorithm
  - Advanced OpenMP: Wavefront parallelization
- **Efficient OpenMP Programming**
  - Profiling OpenMP programs
  - Performance pitfalls
  - Case study: Parallel sparse matrix-vector multiply

Introduction to High Performance Computing for Scientists and Engineers

Contents continued

- **Locality Optimizations on ccNUMA Architectures**
  Locality of access on ccNUMA
  Case study: ccNUMA optimization of sparse MVM
  Placement pitfalls
  ccNUMA issues with C++
- **Distributed-Memory Parallel Programming with MPI**
  Message passing
  A short introduction to MPI
  Example: MPI parallelization of a Jacobi solver
- **Efficient MPI Programming**
  MPI performance tools
  Communication parameters
  Synchronization, serialization, contention
  Reducing communication overhead
  Understanding intranode point-to-point communication
- **Hybrid Parallelization with MPI and OpenMP**
  Basic MPI/OpenMP programming models
  MPI taxonomy of thread interoperability
  Hybrid decomposition and mapping
  Potential benefits and drawbacks of hybrid programming
- **Appendix A: Topology and Affinity in Multicore Environments**
- **Appendix B: Solutions to the Problems**
- **Bibliography**
- **Index**