

# **Performance Engineering on Multi- and Manycores**

**Georg Hager, Gerhard Wellein**

HPC Services, Erlangen Regional Computing Center (RRZE)

**Tutorial @ SAHPC 2012**

**December 1-3, 2012**

**KAUST, Thuwal**

**Saudi Arabia**

- Where can I find those *gorgeous* slides?

<http://goo.gl/cTSKL>

or:

<http://blogs.fau.de/hager/tutorials/sahpc-2012/>

- Is there a book or anything?

Georg Hager and Gerhard Wellein:

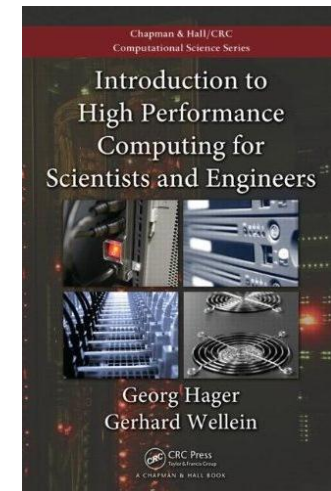
*Introduction to High Performance Computing for Scientists and Engineers*

CRC Press, 2010

ISBN 978-1439811924

356 pages

- Fun and facts for HPC: <http://blogs.fau.de/hager/>

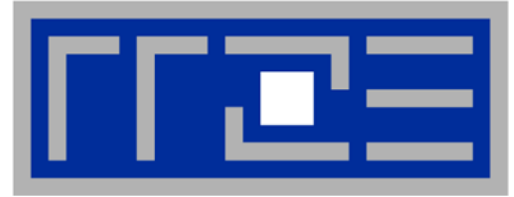




- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



**Motivation 1:  
Scalability 4 the win!**



## **Lore 1**

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

## **Lore 2**

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

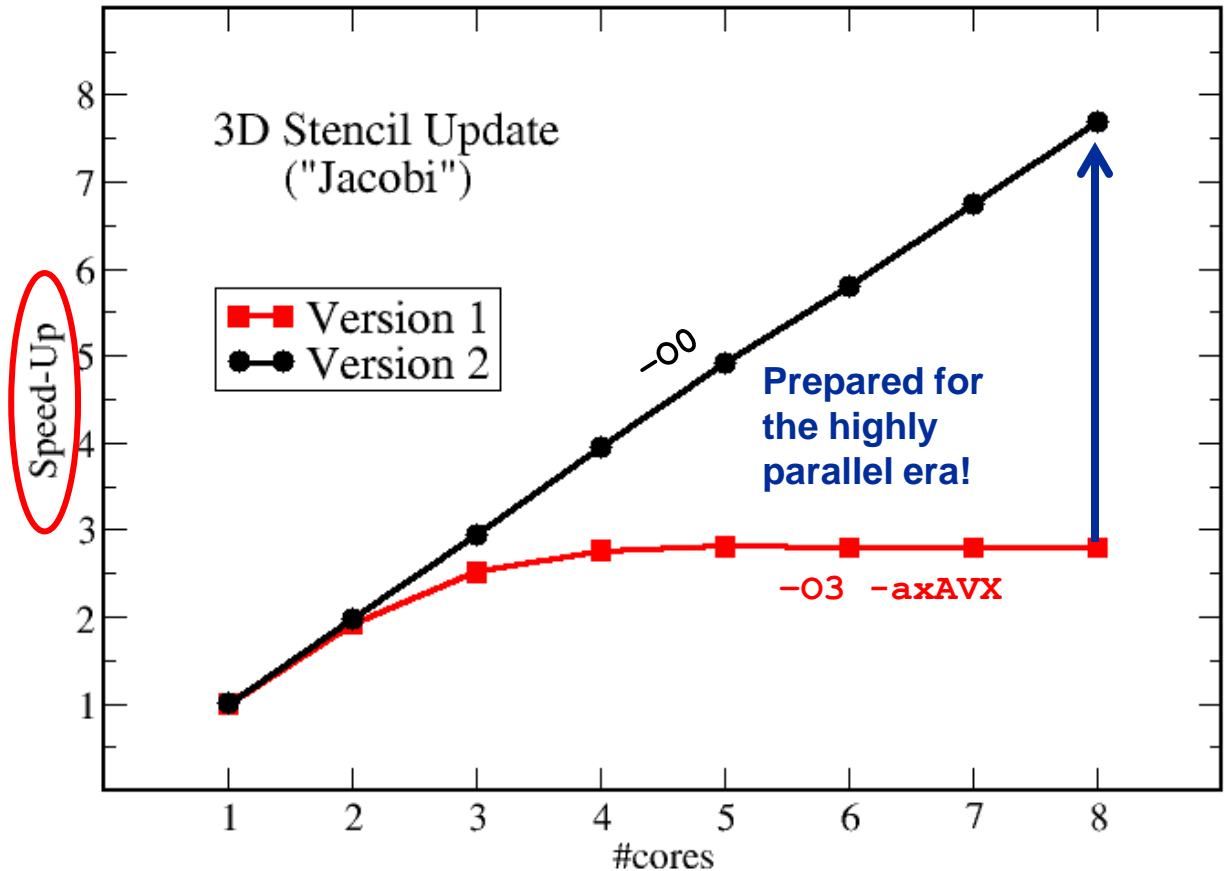
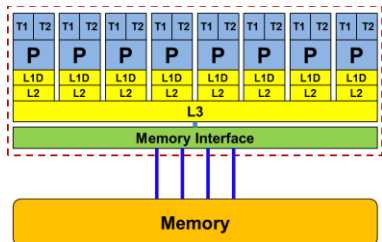
# Scalability Myth: Code scalability is the key issue



```

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

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



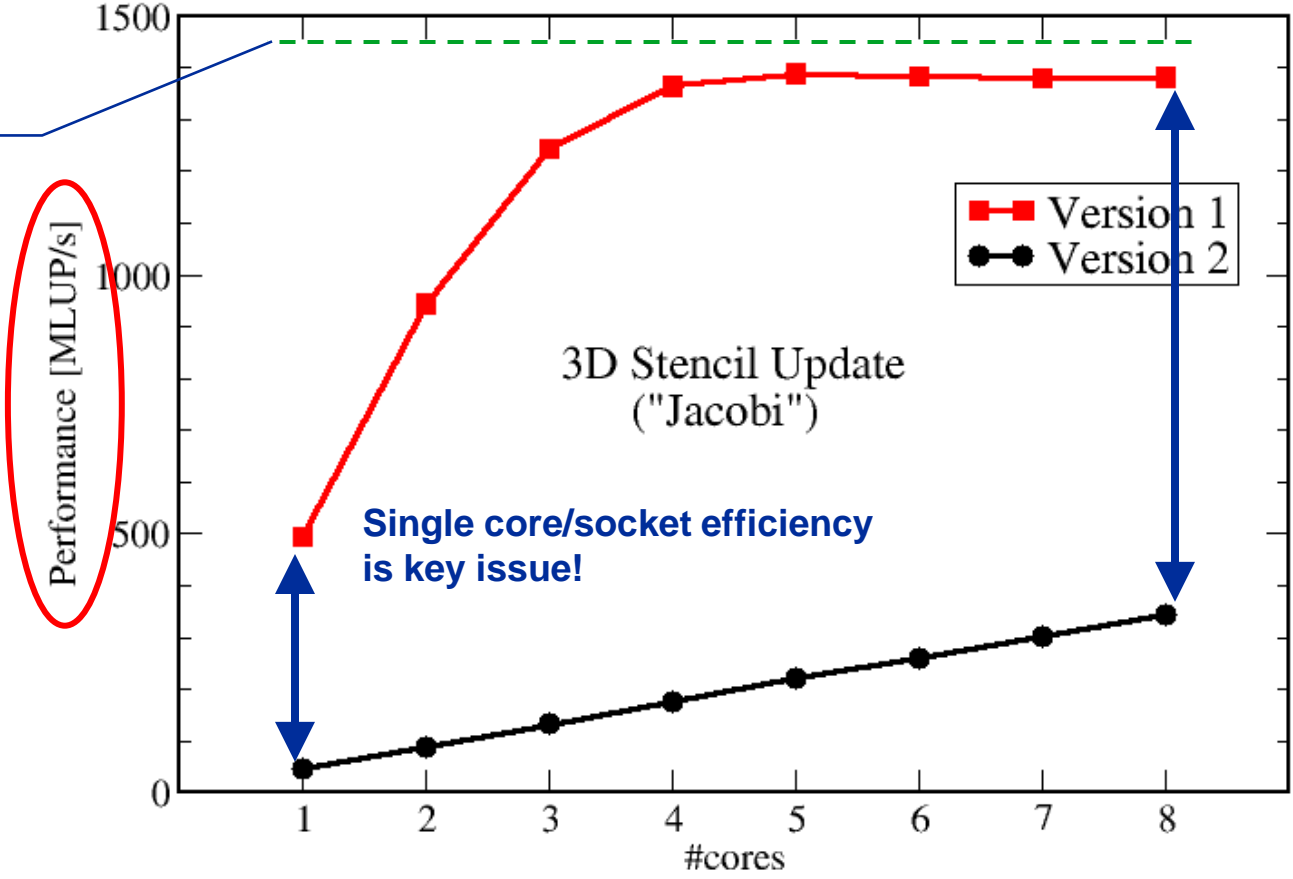
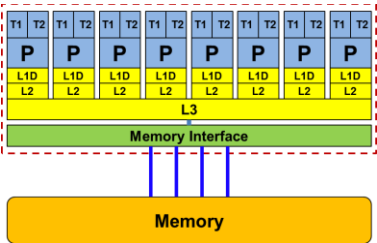
# Scalability Myth: Code scalability is the key issue



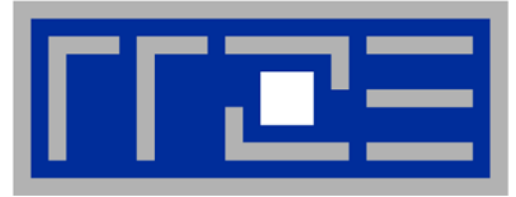
```

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

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







**Motivation 2:**  
**The 200x GPGPU speedup story**

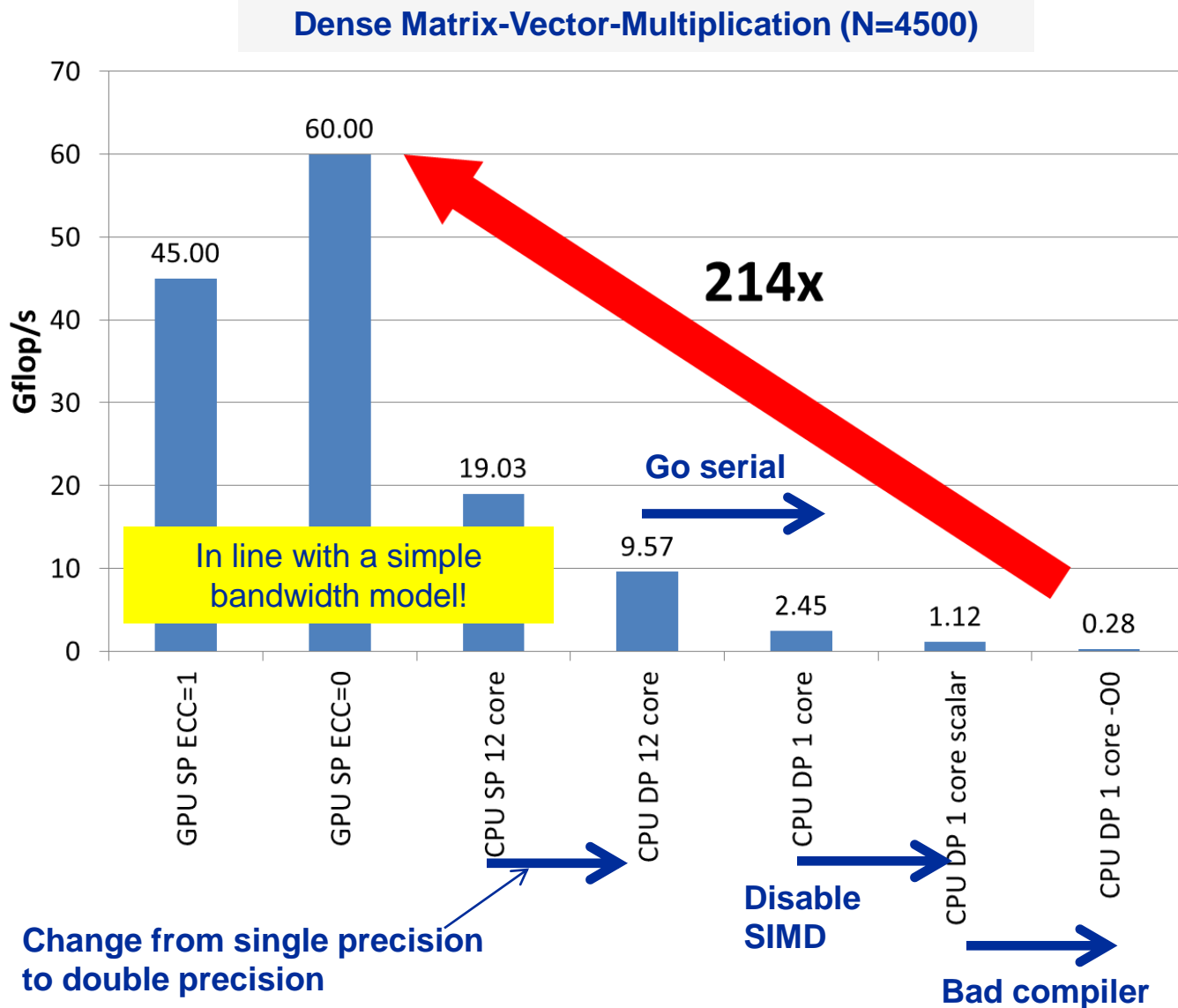
# Accelerator myth: The 200x speedup story...



NVIDIA Tesla C2050

vs.

2x Intel Xeon 5650  
(6-core)



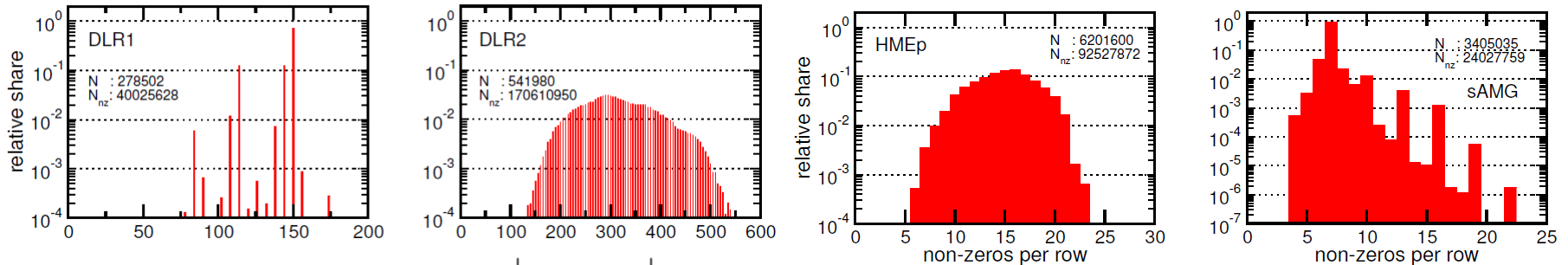


## Sparse matrix-vector multiply

M. Kreutzer et al., LSP12

[DOI: 10.1109/IPDPSW.2012.211](https://doi.org/10.1109/IPDPSW.2012.211)

Matrix structure of test cases



|                    |           | DLR1 | DLR2 | HMEp | sAMG |
|--------------------|-----------|------|------|------|------|
| data reduction [%] |           | 17.5 | 48.0 | 36.0 | 68.4 |
| SP ECC=0           | ELLPACK-R | 22.1 | 15.2 | 15.8 | 14.6 |
|                    | pJDS      | 27.6 | 18.7 | 18.9 | 19.5 |
| SP ECC=1           | ELLPACK-R | 18.0 | 13.2 | 12.1 | 11.6 |
|                    | pJDS      | 19.1 | 12.1 | 11.6 | 12.6 |
| DP ECC=0           | ELLPACK-R | 18.7 | 11.7 | 12.3 | 11.1 |
|                    | pJDS      | 18.3 | 14.6 | 12.2 | 13.0 |
| DP ECC=1           | ELLPACK-R | 12.9 | 9.6  | 7.9  | 7.8  |
|                    | pJDS      | 12.9 | 9.5  | 7.5  | 8.5  |
| Westmere EP        | CRS (DP)  | 5.7  | 5.8  | 3.9  | 4.1  |

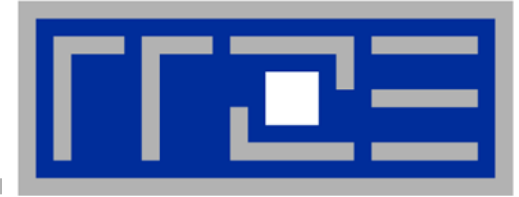
NVIDIA Tesla C2070  
performance in GF/s

2-way Intel Xeon 5650 node

- **GPGPU speedup: 1.6x,...,2.1x (no PCIe data transfer!)**



- Motivation
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



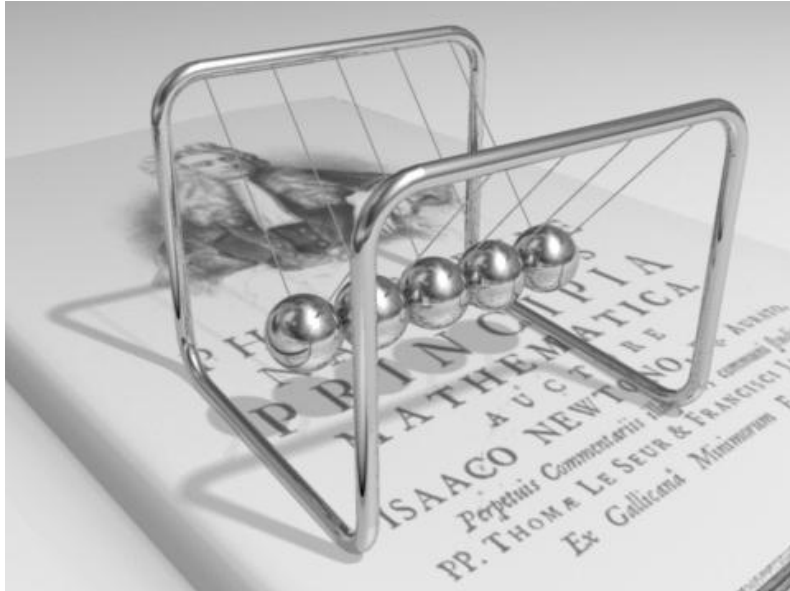
# **The Performance Engineering process**

**Model building**

**Our definition**



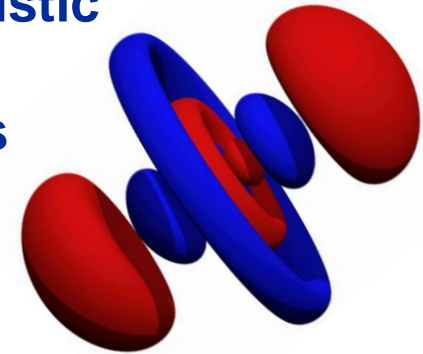
## Newtonian mechanics



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

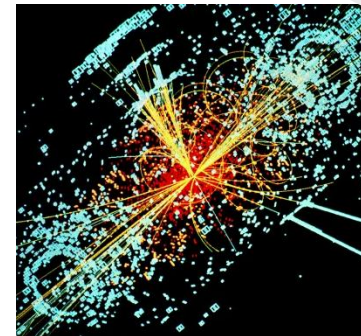
**Fails @ small scales!**

## Nonrelativistic quantum mechanics



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

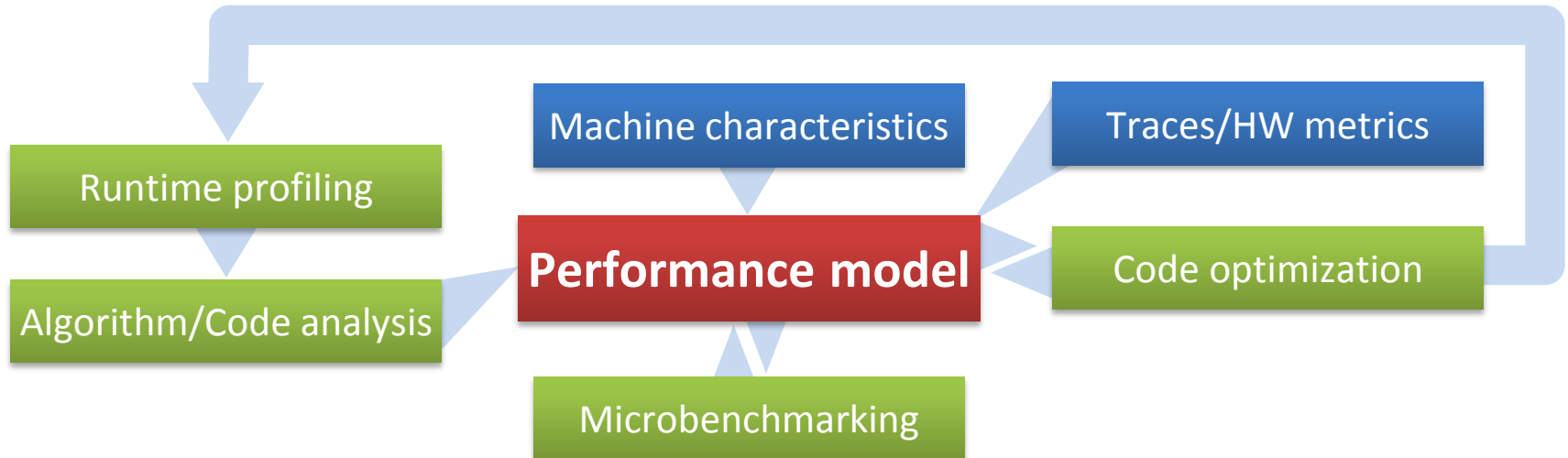
**Fails @ even smaller scales!**



## Relativistic quantum field theory

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

## The Performance Engineering (PE) process:



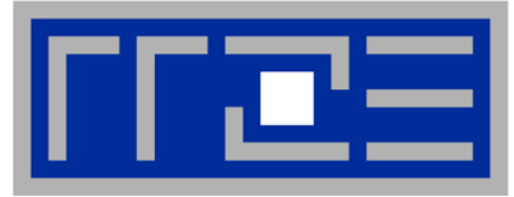
**The performance model is the central component – if the model fails to predict the measurement, you learn something!**

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



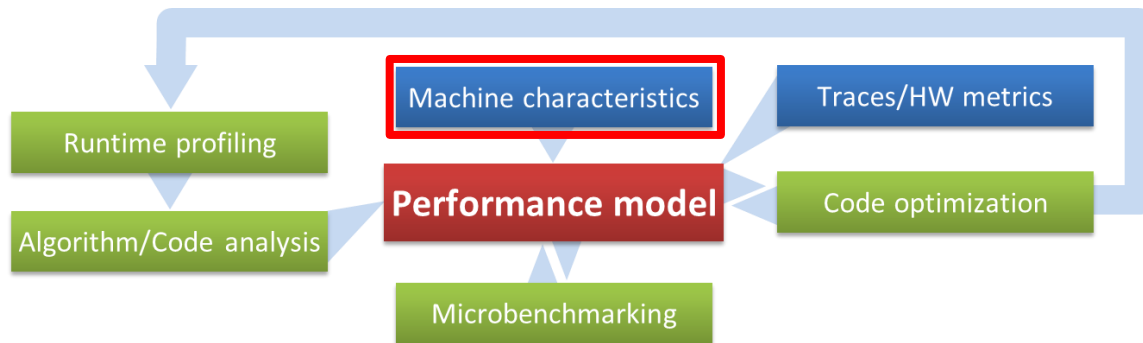
- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**





# Multicore processor and system architecture

## Basics of machine characteristics

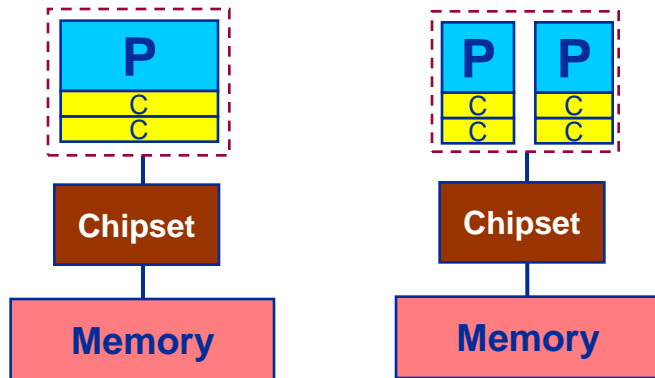


# The x86 multicore evolution so far

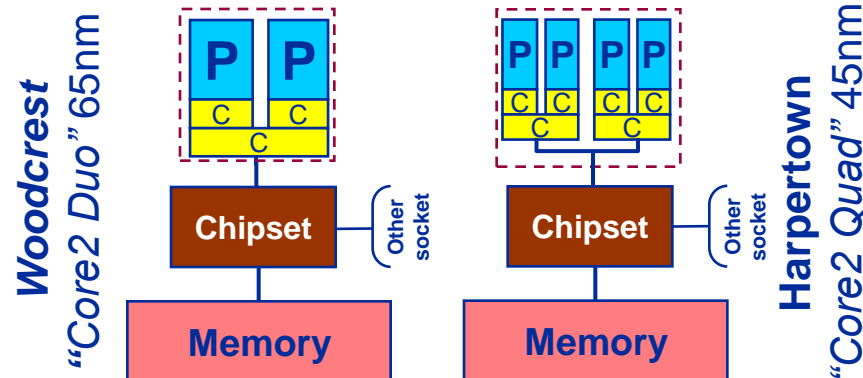
Intel Single-/Dual-/.../Octo-Cores (one-socket view)



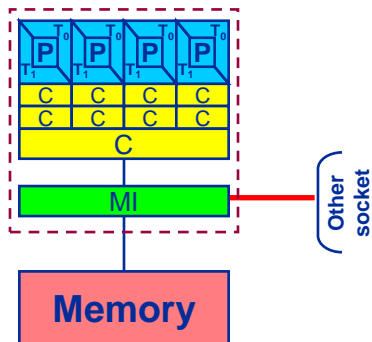
2005: "Fake" dual-core



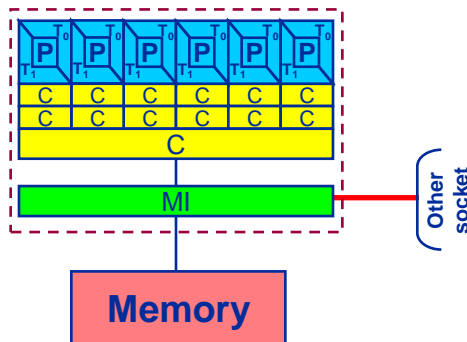
2006: True dual-core



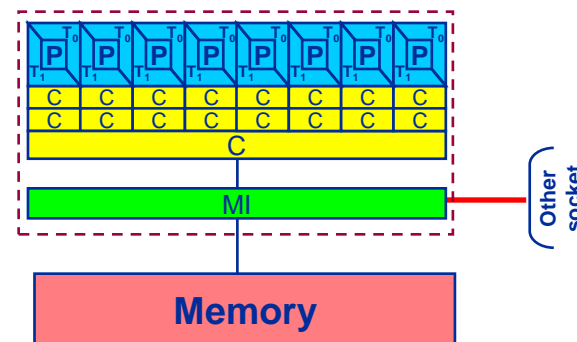
2008: Simultaneous Multi Threading (SMT)



2010: 6-core chip



2012: Wider SIMD units  
AVX: 256 Bit



**Nehalem EP**  
"Core i7"  
45nm

**Westmere EP**  
"Core i7"  
32nm

**Sandy Bridge EP**  
"Core i7"  
32nm

# There is no single driving force for chip performance!



## Floating Point (FP) Performance:

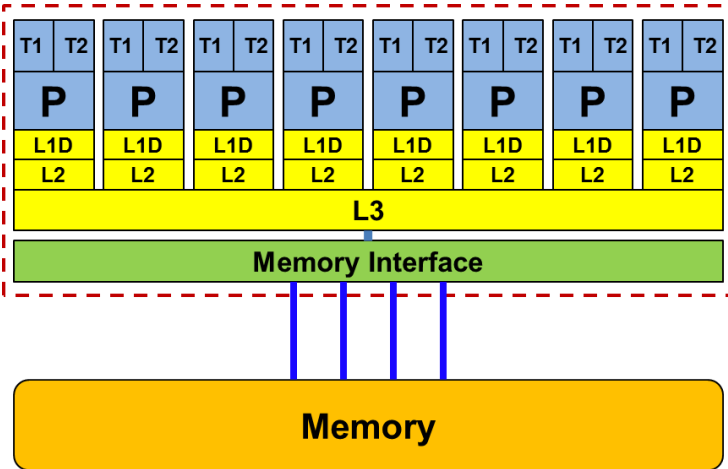
$$P = n_{\text{core}} * F * S * v$$

$n_{\text{core}}$  number of cores: 8

$F$  FP instructions per cycle: 2  
(1 MULT and 1 ADD)

$S$  FP ops / instruction: 4 (dp) / 8 (sp)  
(256 Bit SIMD registers – “AVX”)

$v$  Clock speed : ~2.7 GHz



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

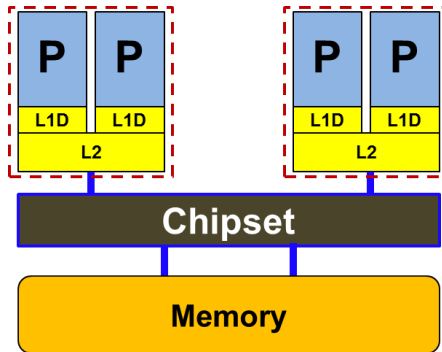
TOP500 rank 1 (1995)

$$P = 173 \text{ GF/s (dp)} / 346 \text{ GF/s (sp)}$$

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



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

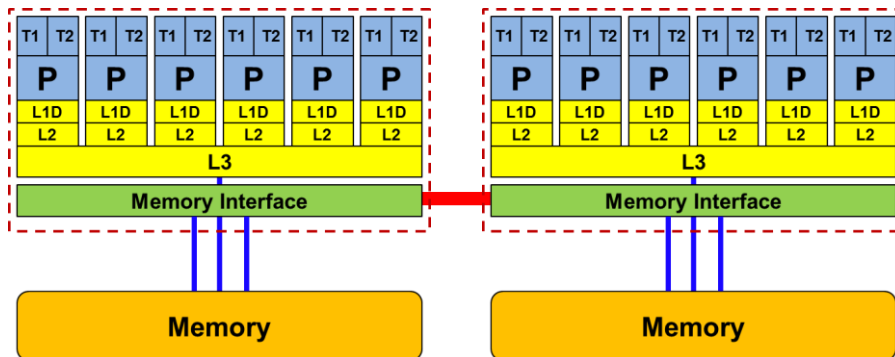


Uniform Memory Architecture (UMA)

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!**

# Another flavor of "SMT"

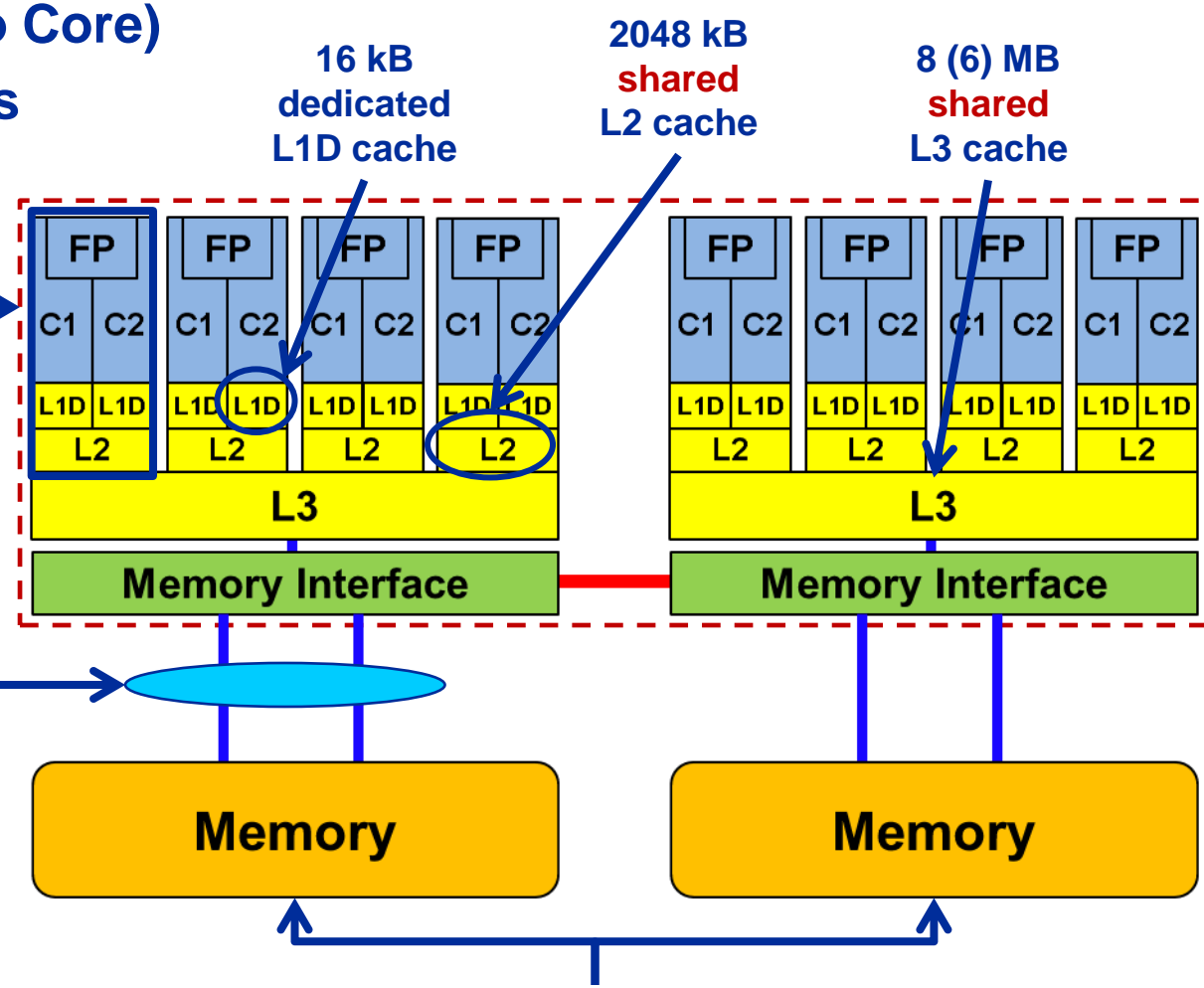
## AMD Interlagos / Bulldozer



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

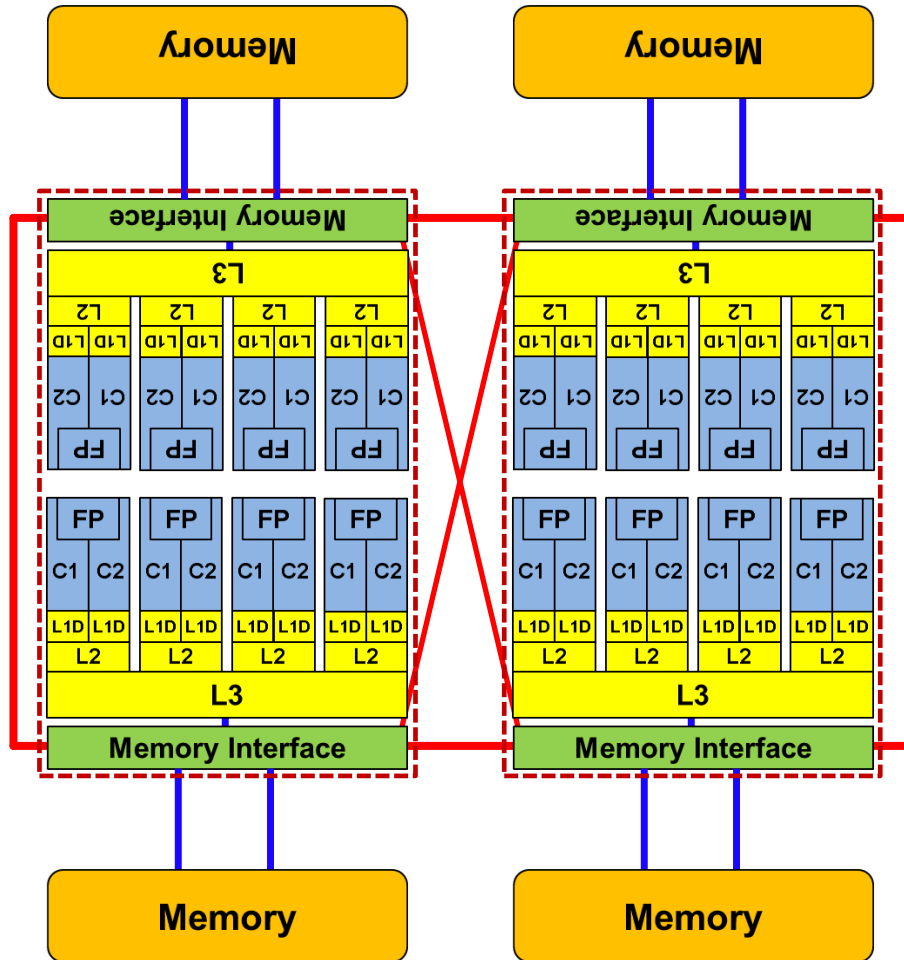
### Each Bulldozer module:

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

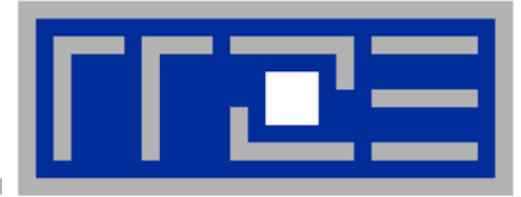


2 DDR3 (shared) memory channel > 15 GB/s

2 NUMA domains per socket



- **Two 8- (integer-) core chips per socket @ 2.3 GHz (3.3 @ turbo)**
- **Separate DDR3 memory interface per chip**
  - ccNUMA on the socket!
- **Shared FP unit per pair of integer cores (“module”)**
  - “256-bit” FP unit
  - SSE4.2, AVX, FMA4
- **16 kB L1 data cache per core**
- **2 MB L2 cache per module**
- **8 MB L3 cache per chip (6 MB usable)**



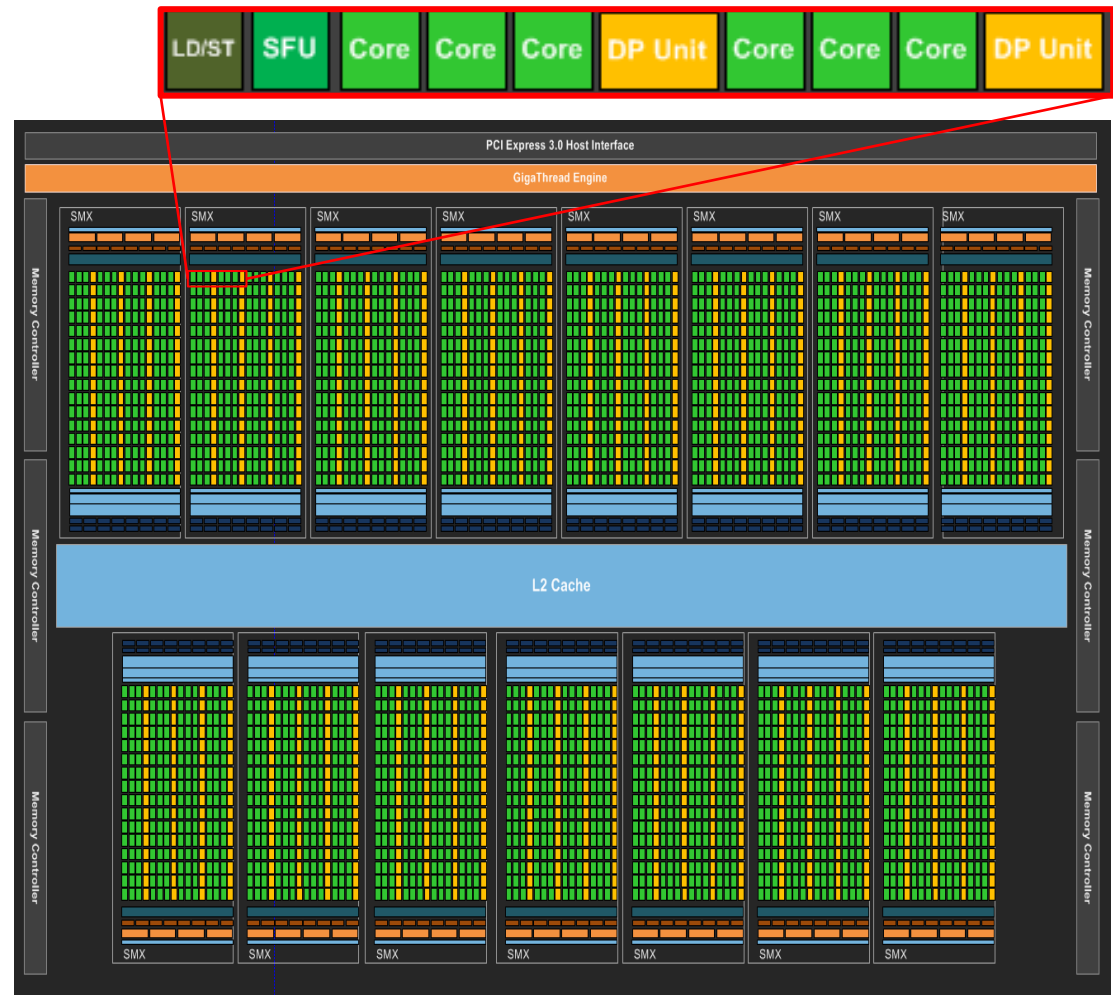
**Interlude:**  
**A glance at current accelerator technology**

# NVIDIA Kepler GK110 Block Diagram



## Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3
- **3:1 SP:DP performance**



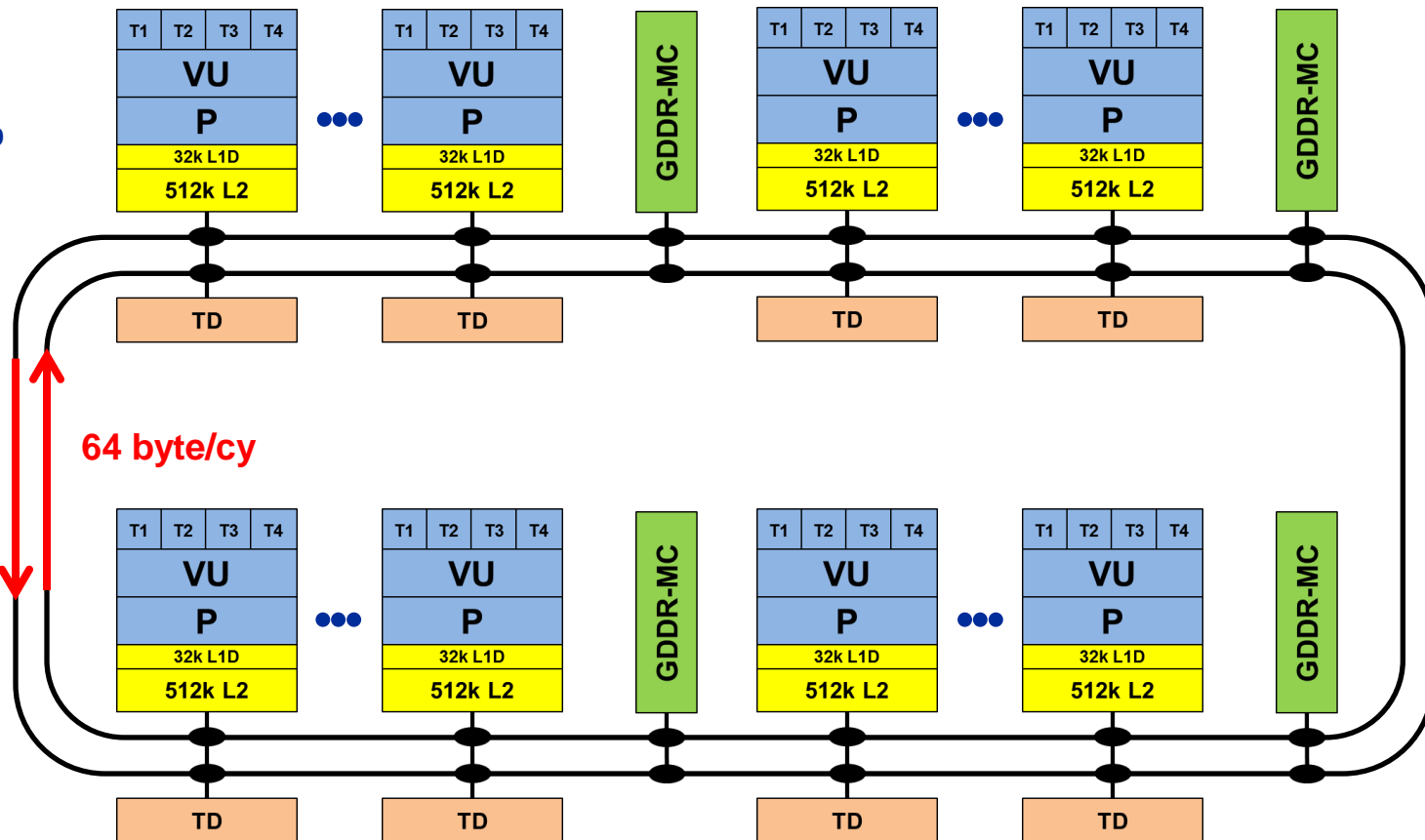
© NVIDIA Corp. Used with permission.





## Architecture

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





## ■ Intel Xeon Phi

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

- Clock Speed: ~1000 MHz
- Transistor count: ~3 B (22nm)
- Power consumption: ~250 W



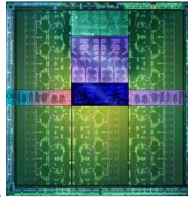
- Peak Performance (DP): ~ 1 TF/s
- Memory BW: ~250 GB/s (GDDR5)

- Threads to execute: 60-240+
- Programming:  
Fortran/C/C++ +OpenMP + SIMD

## ■ NVIDIA Kepler K20

- 15 SMX units each with 192 “cores” → **960/2880 DP/SP “cores”** in total

- Clock Speed: ~700 MHz
- Transistor count: 7.1 B (28nm)
- Power consumption: ~250 W



- Peak Performance (DP): ~ 1.3 TF/s
- Memory BW: ~ 250 GB/s (GDDR5)

- Threads to execute: 10.000+
- Programming:  
CUDA, OpenCL, (OpenACC)

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

**TOP500 rankings**

- TOP1: “Titan” at Oak Ridge National Laboratory

# Trading single thread performance for parallelism:

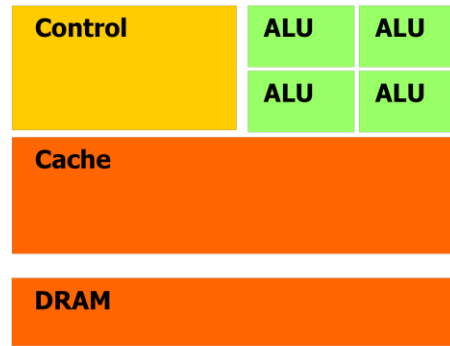
## GPGPUs vs. CPUs



### GPU vs. CPU

light speed estimate:

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



CPU



GPU

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

<sup>+</sup> Single Precision

\* Includes on-chip GPU and PCI-Express

**Complete compute device**



- **Shared-memory (intra-node)**
  - **Good old MPI** (current standard: 2.2)
  - **OpenMP** (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs,... you name it
  
- **Distributed-memory (inter-node)**
  - **MPI** (current standard: 2.2)
  - 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

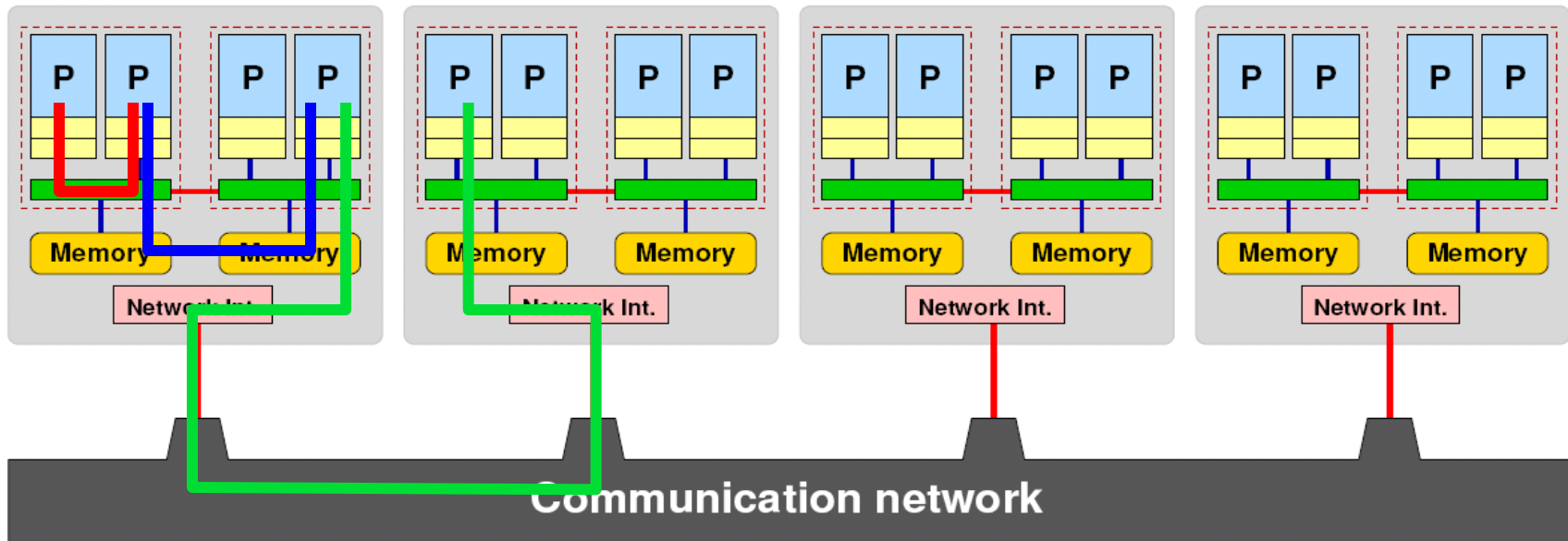
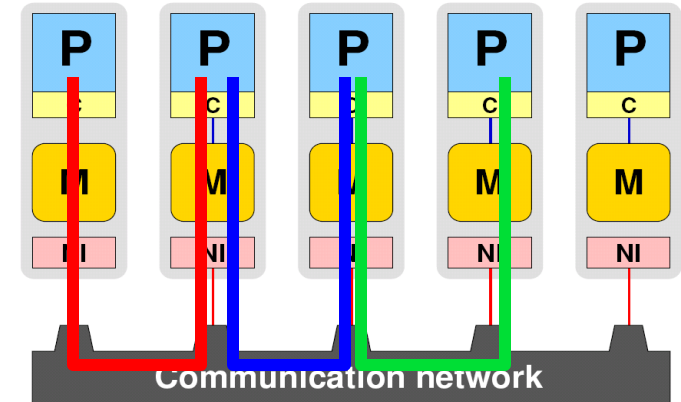


- Machine structure is invisible to user:

- 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

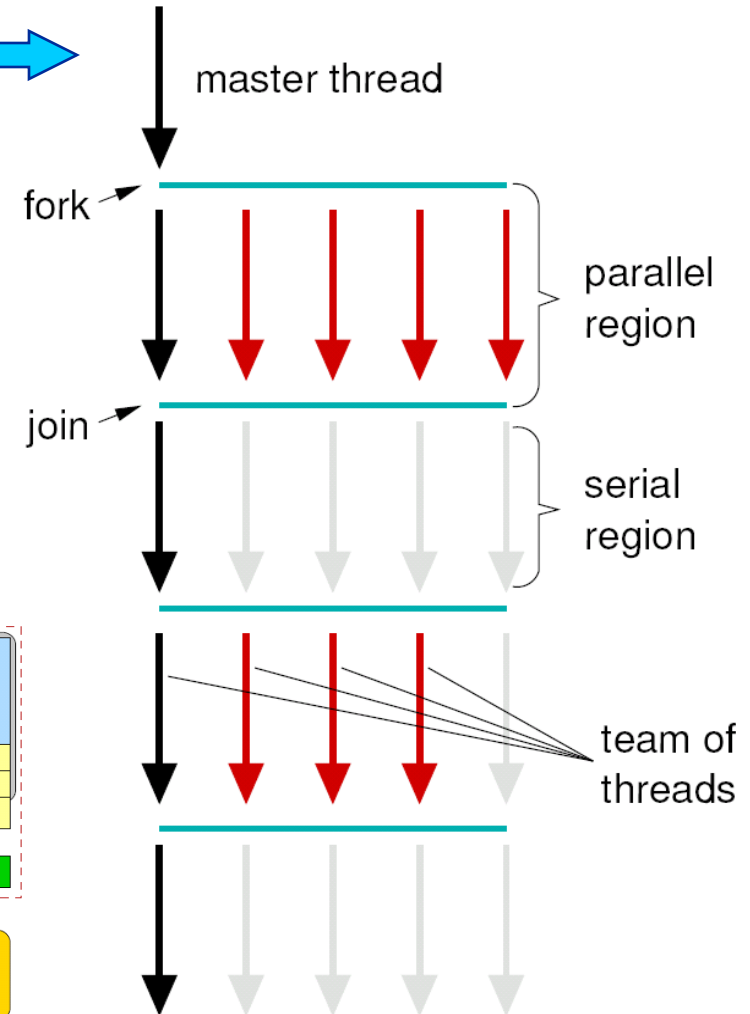
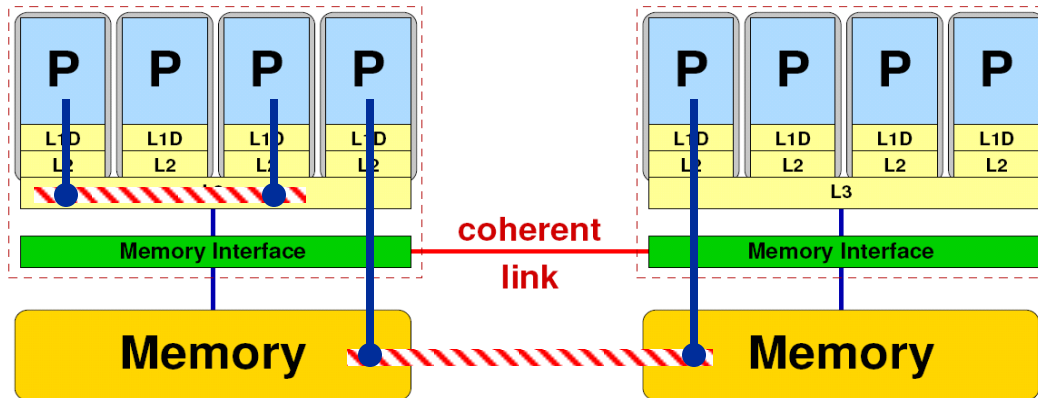


- Machine structure is invisible to user

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

- Performance issues

- Synchronization overhead
- Memory access
- Node topology

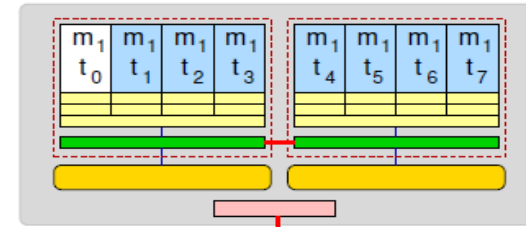
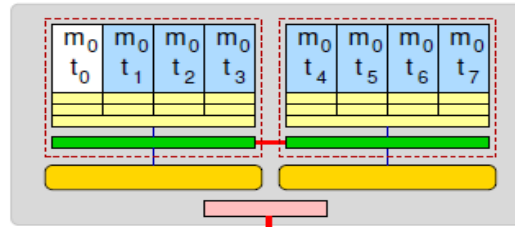


# Parallel programming models:

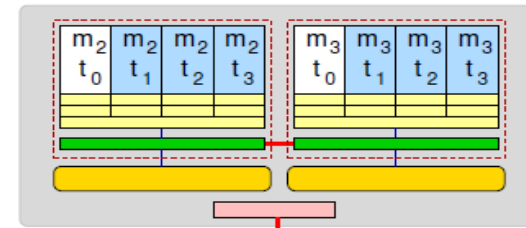
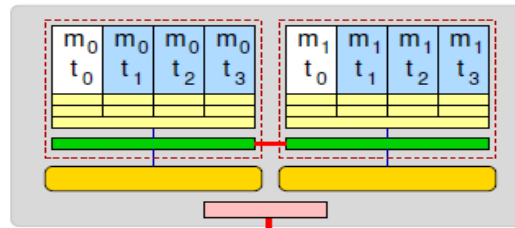
Hybrid MPI+OpenMP on a multicore multisocket cluster



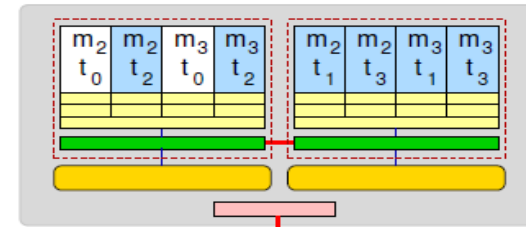
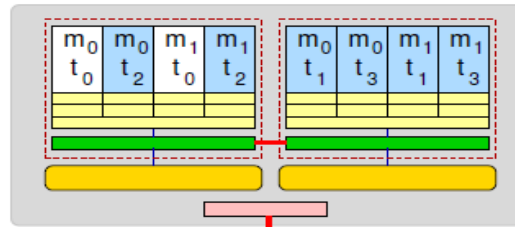
One MPI process / node



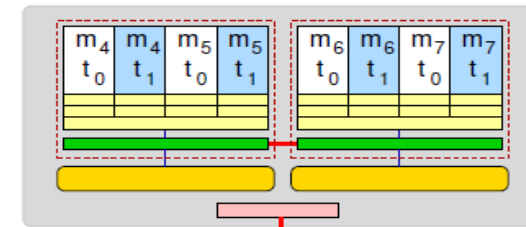
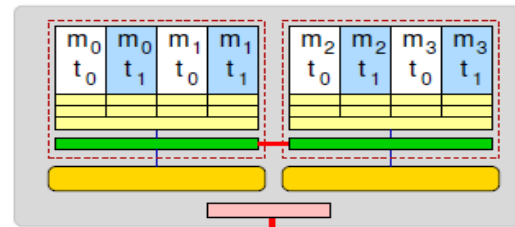
One MPI process / socket:  
OpenMP threads on same  
socket: “**blockwise**”



OpenMP threads pinned  
“**round robin**” across  
cores in node



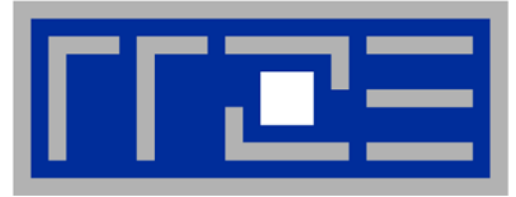
Two MPI processes / socket  
OpenMP threads  
on same socket





- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



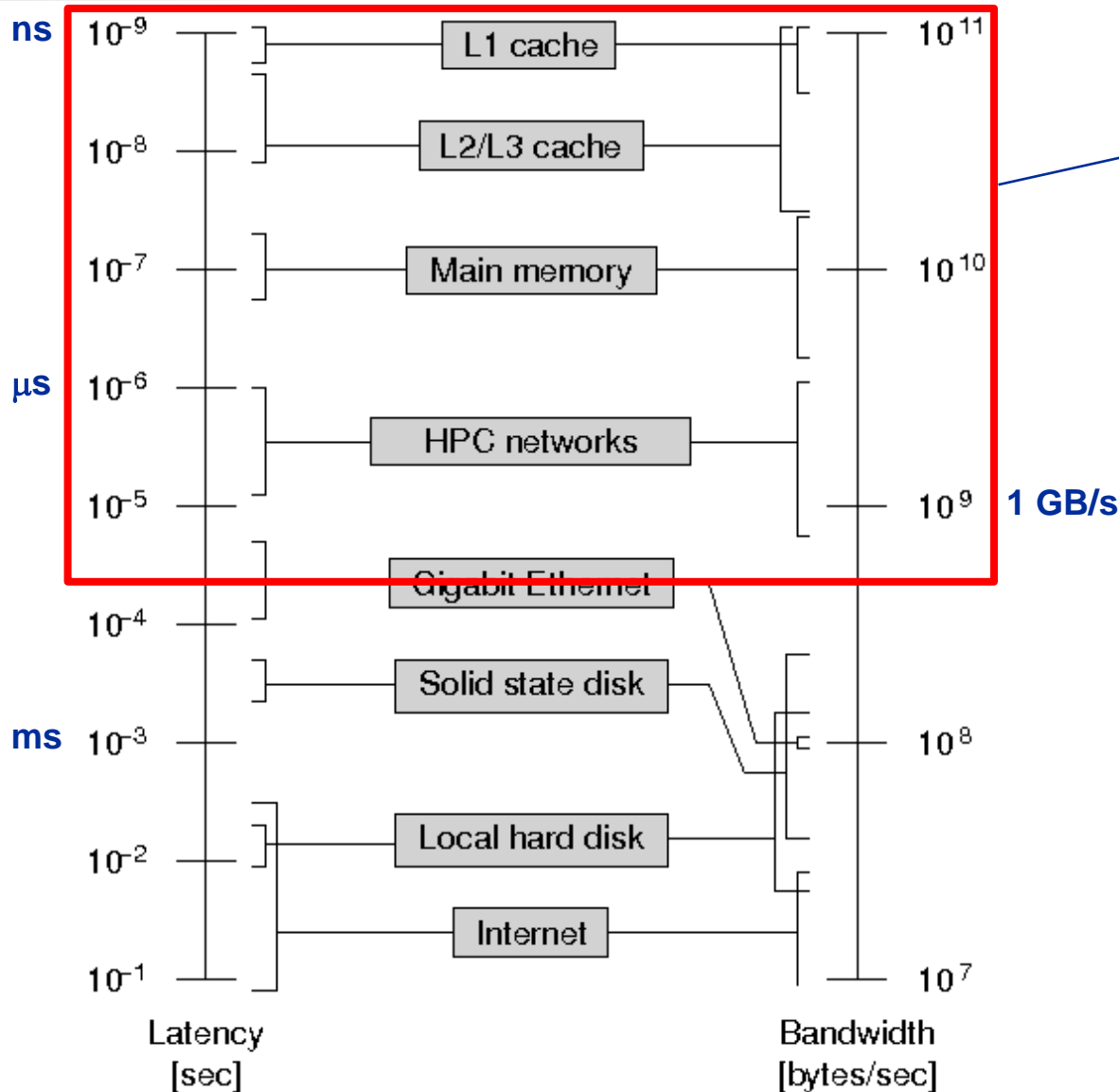


## **Data access on modern processors**

**Characterization of memory hierarchies**

**General performance properties of multicore processors**

# Latency and bandwidth in modern computer environments



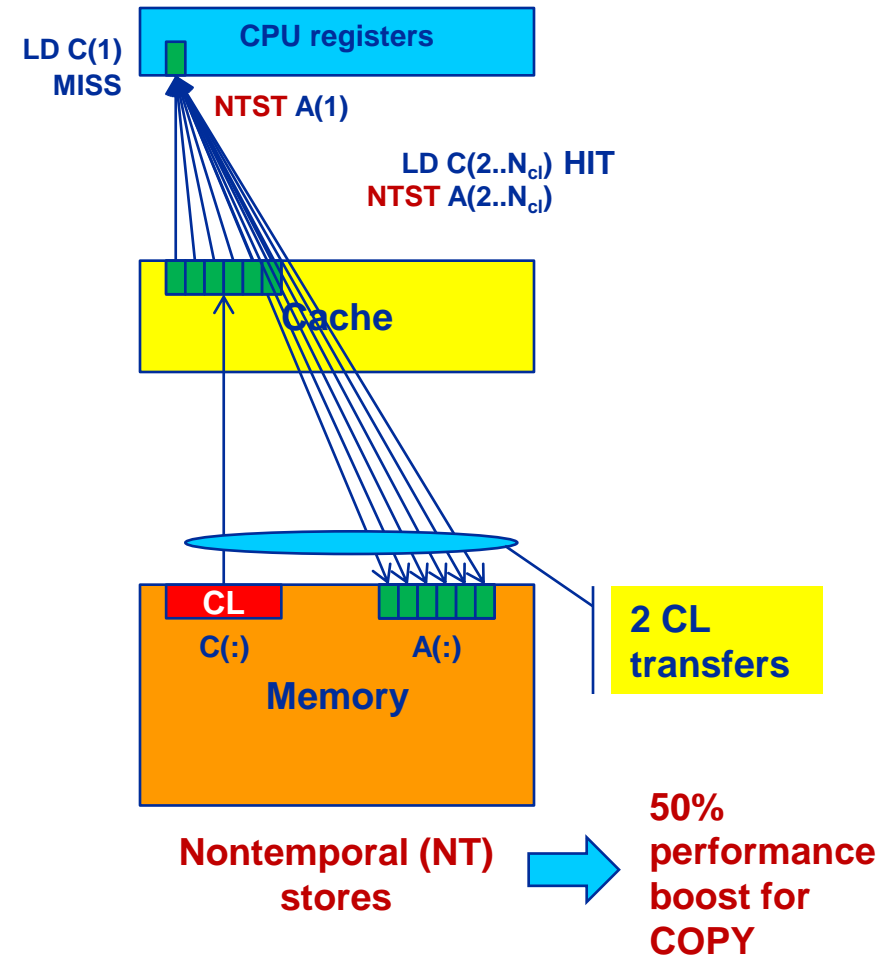
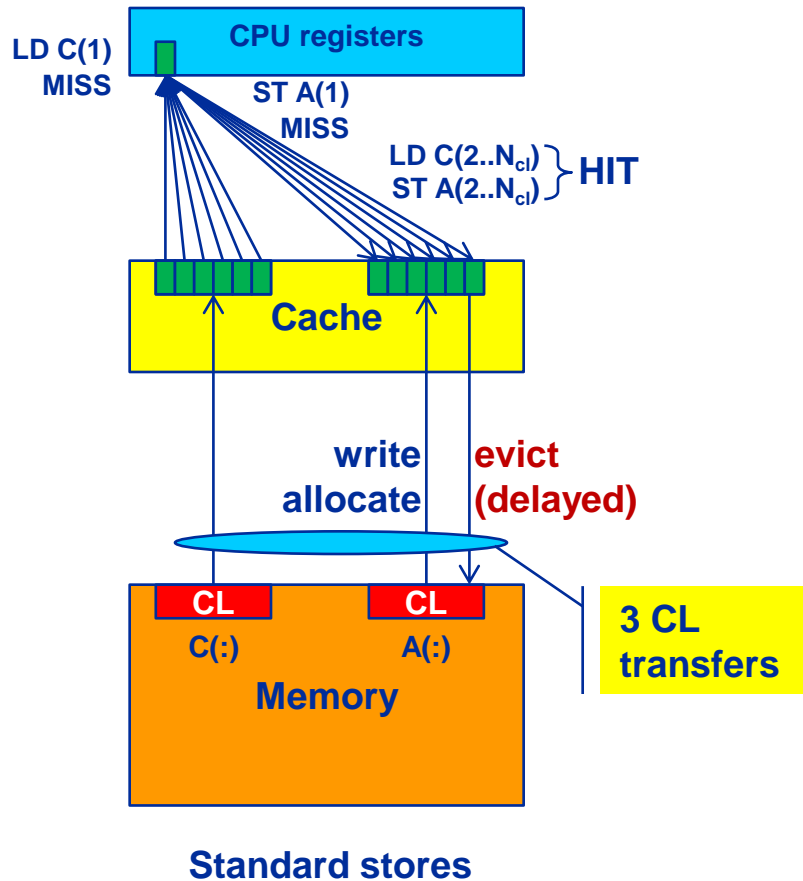
HPC plays here

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

# Interlude: Data transfers in a memory hierarchy



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





### Simple streaming benchmark:

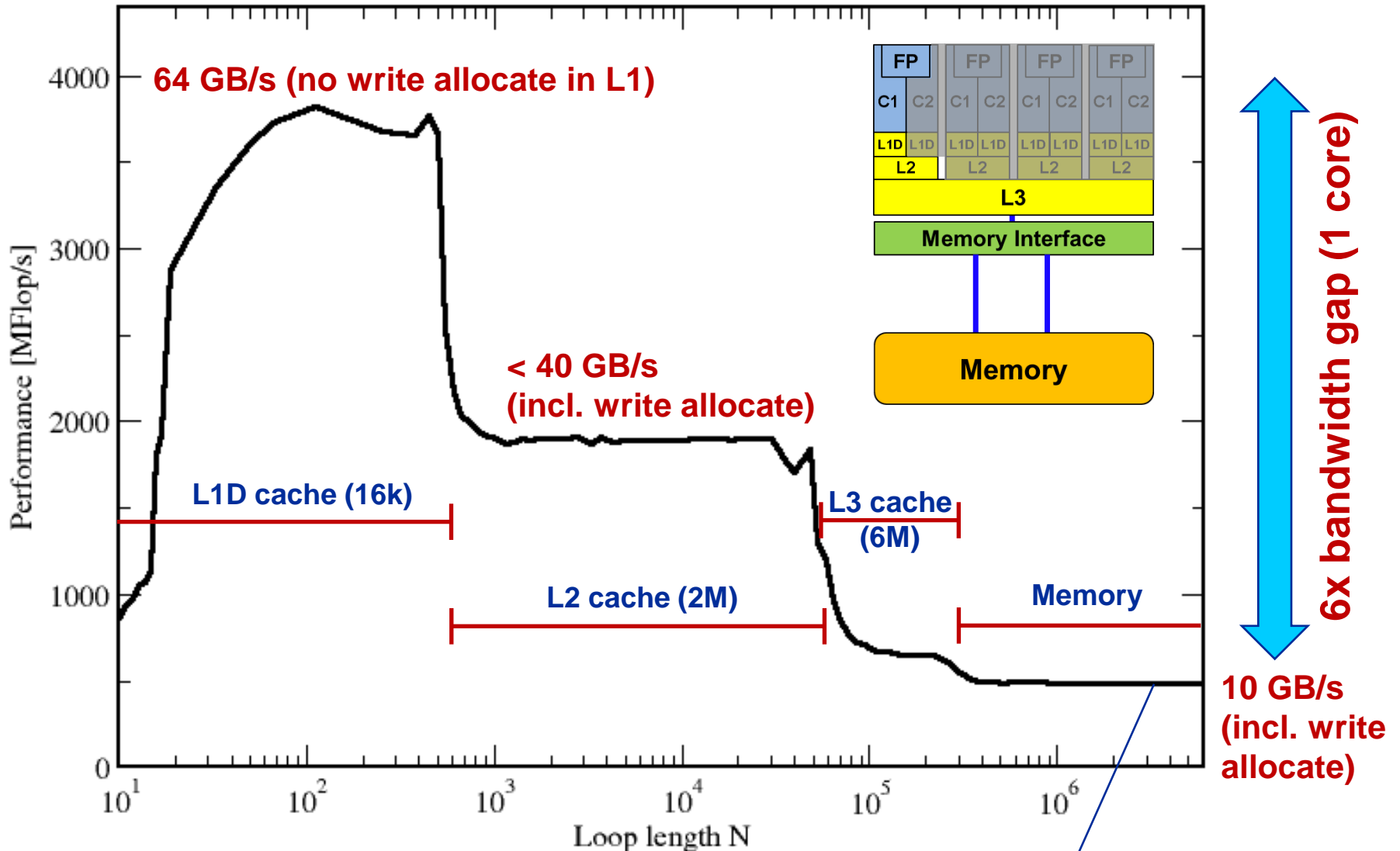
```
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A
```

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

Prevents smarty-pants compilers from doing “clever” stuff

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

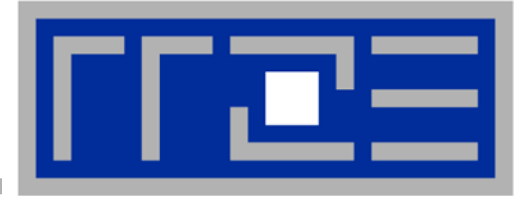
# A(:)=B(:)+C(:)\*D(:) on one Interlagos core



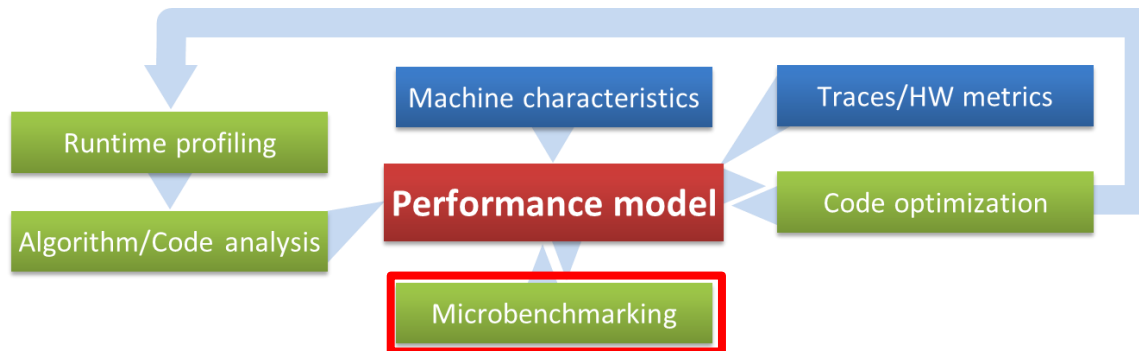
Is this the limit???



- Motivation
- Performance Engineering
  - Performance modeling
  - The Performance Engineering process
- Modern architectures
  - Multicore
  - Accelerators
  - Programming models
- Data access
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- Case study: OpenMP-parallel sparse MVM
- Basic performance modeling: Roofline
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- Conclusions

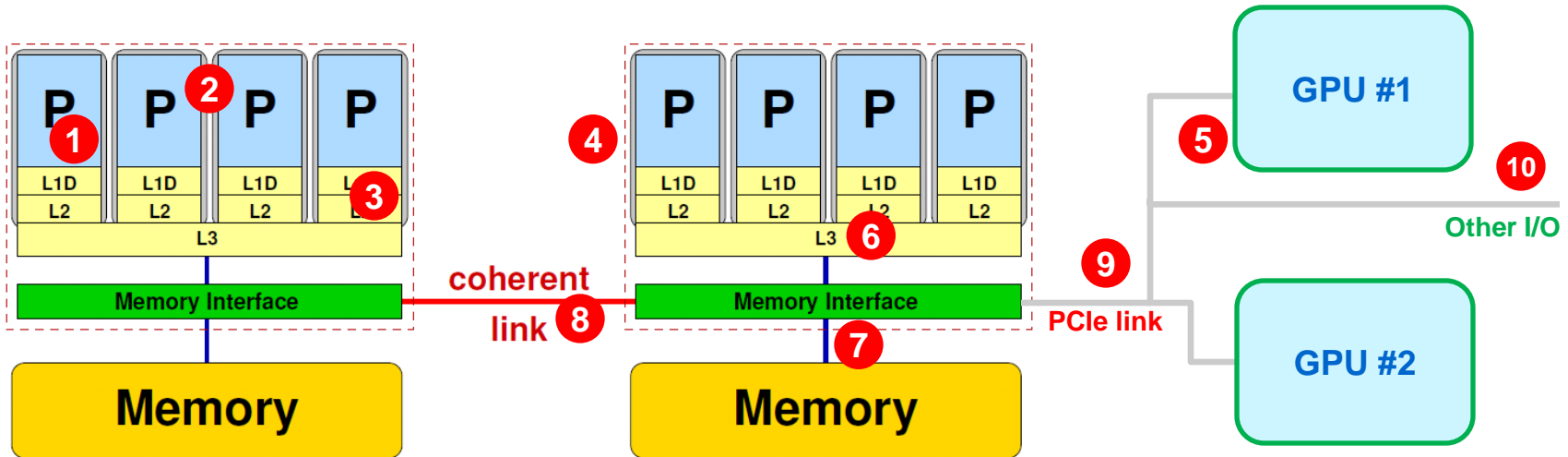


## General remarks on the performance properties of multicore multi-socket systems





- Parallel and shared resources within a shared-memory node



## Parallel resources:

- Execution/SIMD units (1)
- Cores (2)
- Inner cache levels (3)
- Sockets / memory 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?**



# The parallel vector triad benchmark

(Near-)Optimal code on (Cray) x86 machines



```
call get_walltime(S)
!$OMP parallel private(j)
do j=1,R
  if(N.ge.CACHE_LIMIT) then
!DIR$ LOOP_INFO cache_nt(A)
!$OMP parallel do
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP end parallel do
  else
!DIR$ LOOP_INFO cache(A)
!$OMP parallel do
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP end parallel do
  endif
  ! prevent loop interchange
  if(A(N2).lt.0) call dummy(A,B,C,D)
enddo
!$OMP end parallel

call get_walltime(E)
```

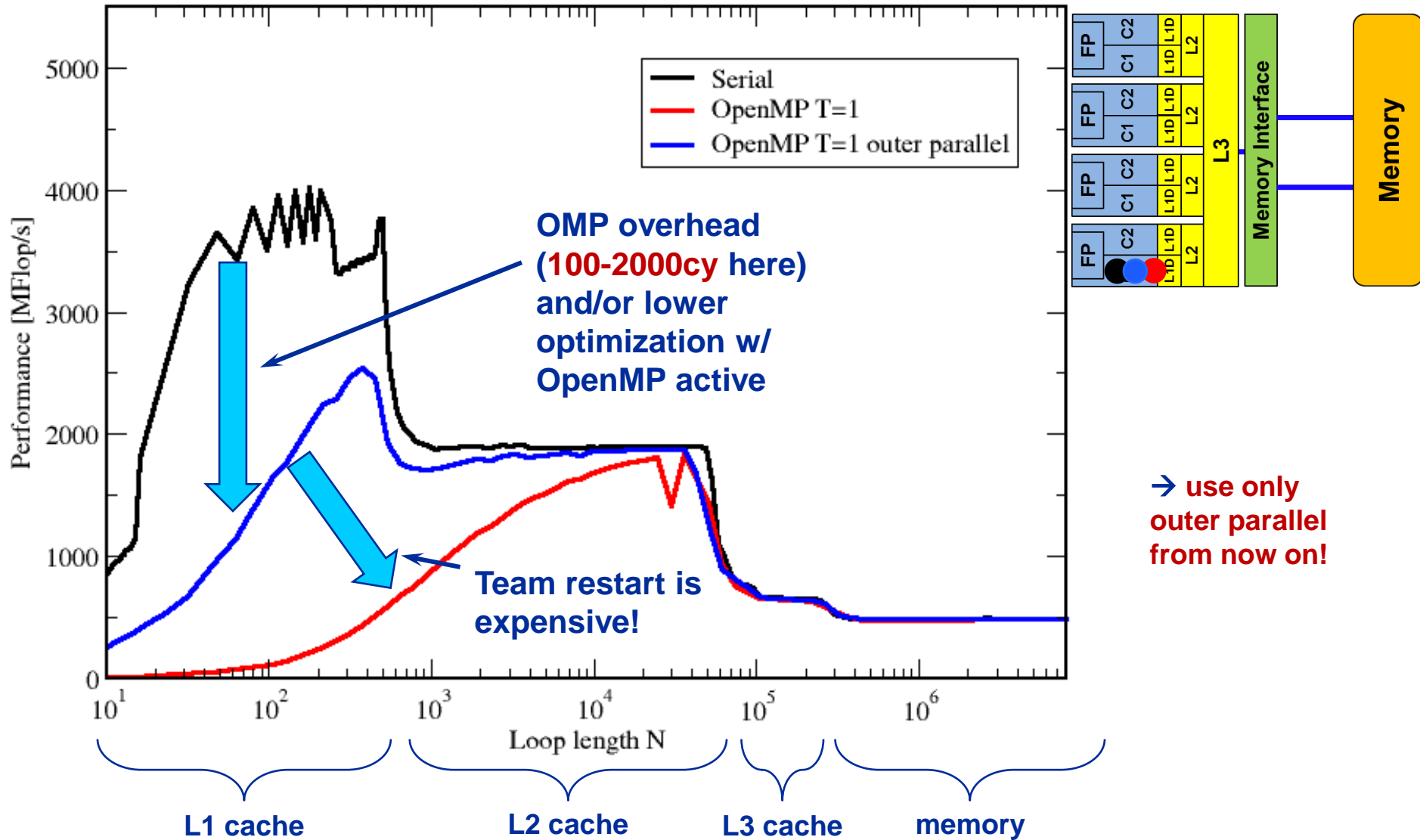
“outer parallel”: Avoid thread team restart at every workshared loop

Large-N version  
(nontemporal stores)

Small-N version  
(standard stores)

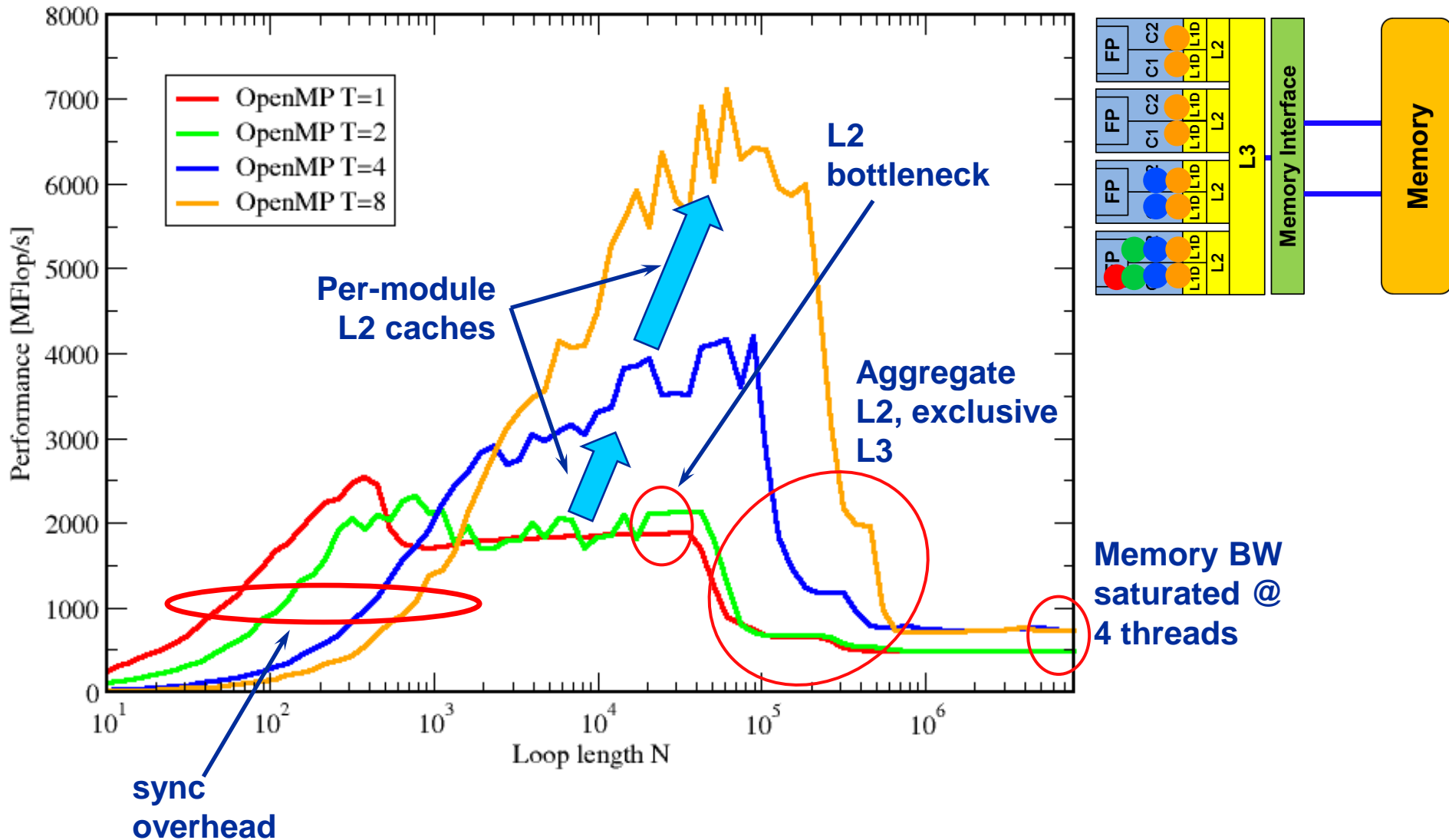
# The parallel vector triad benchmark

Single thread on Cray XE6 Interlagos node



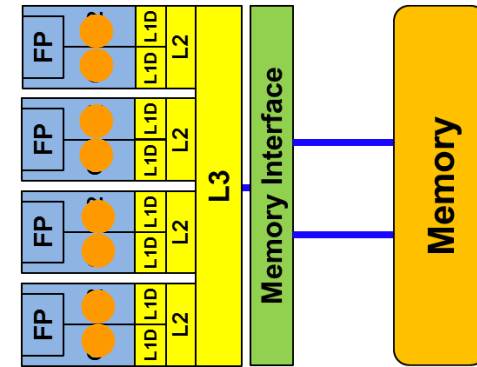
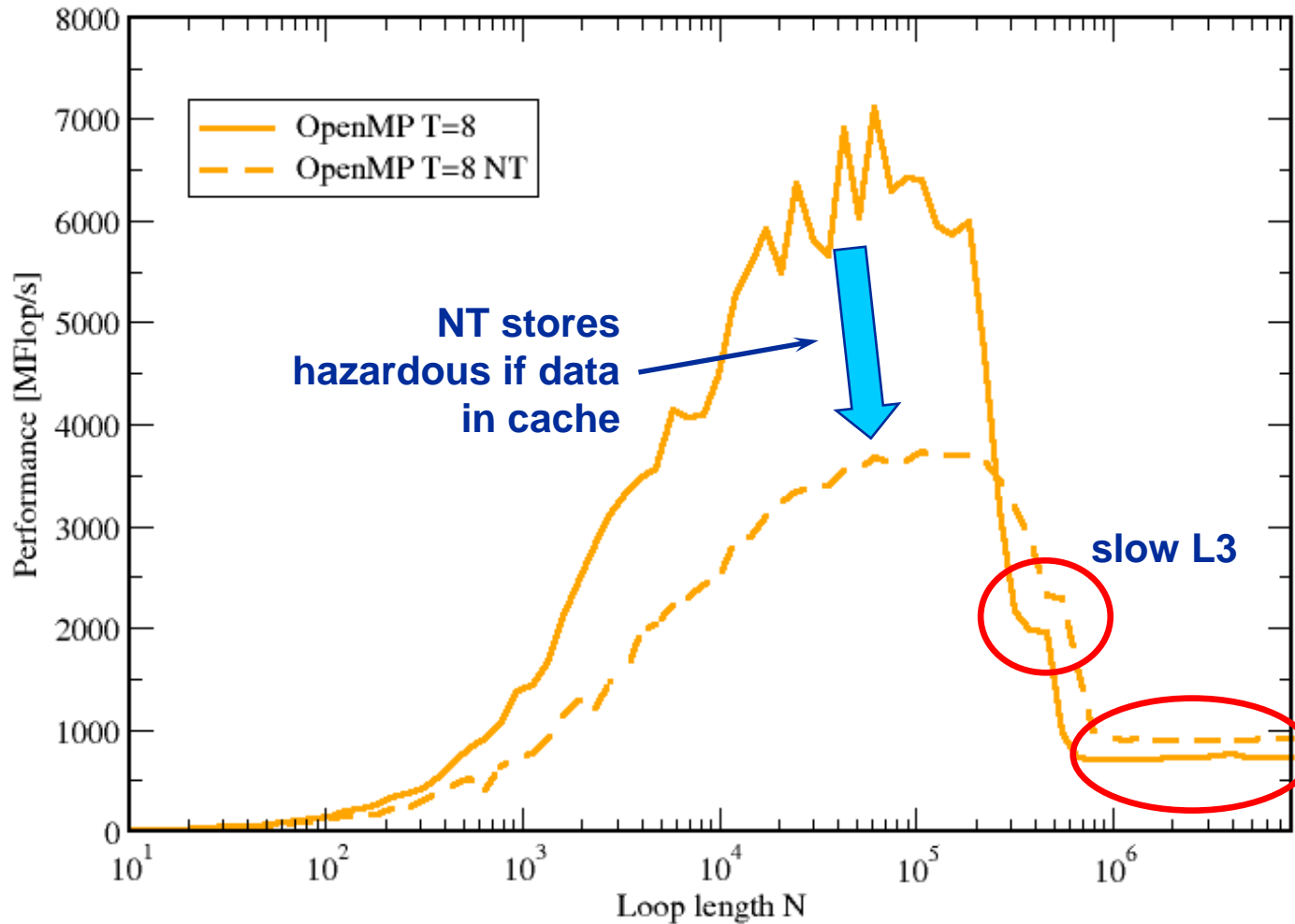
# The parallel vector triad benchmark

## Intra-chip scaling on Cray XE6 Interlagos node



# The parallel vector triad benchmark

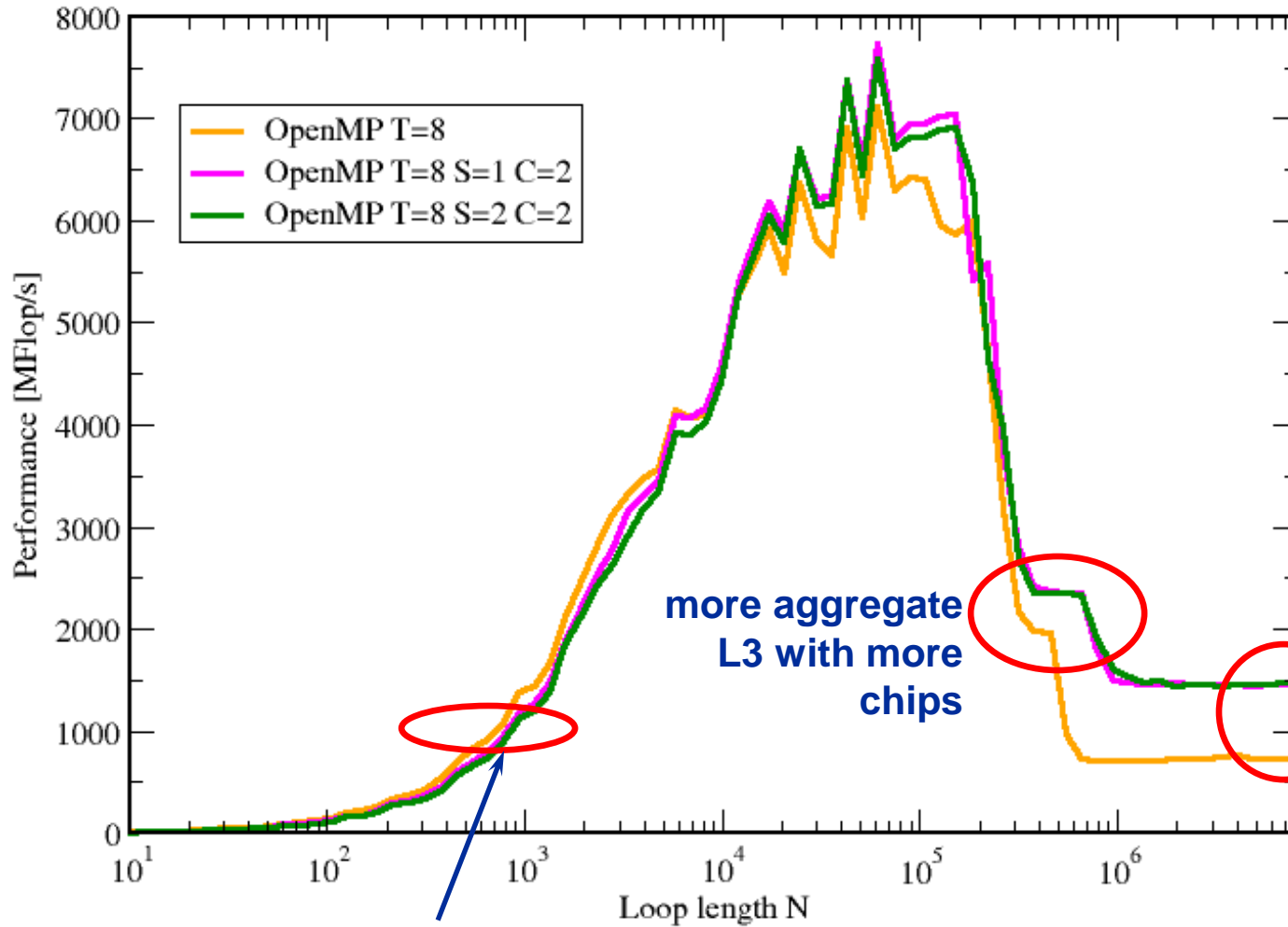
*Nontemporal stores on Cray XE6 Interlagos node*



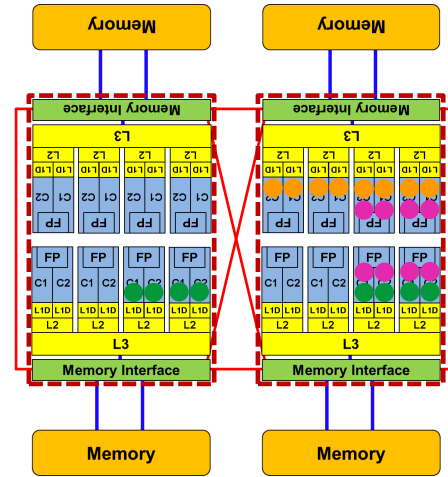
**25% speedup for vector triad in memory via NT stores**

# The parallel vector triad benchmark

Topology dependence on Cray XE6 Interlagos node



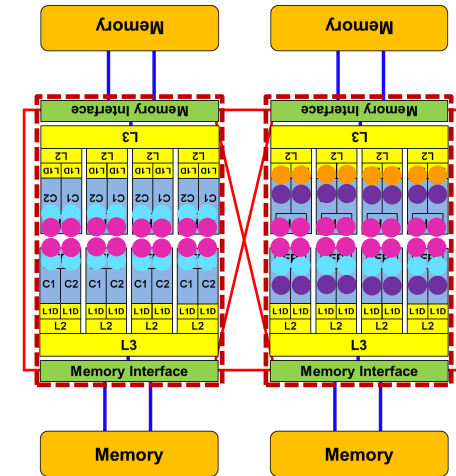
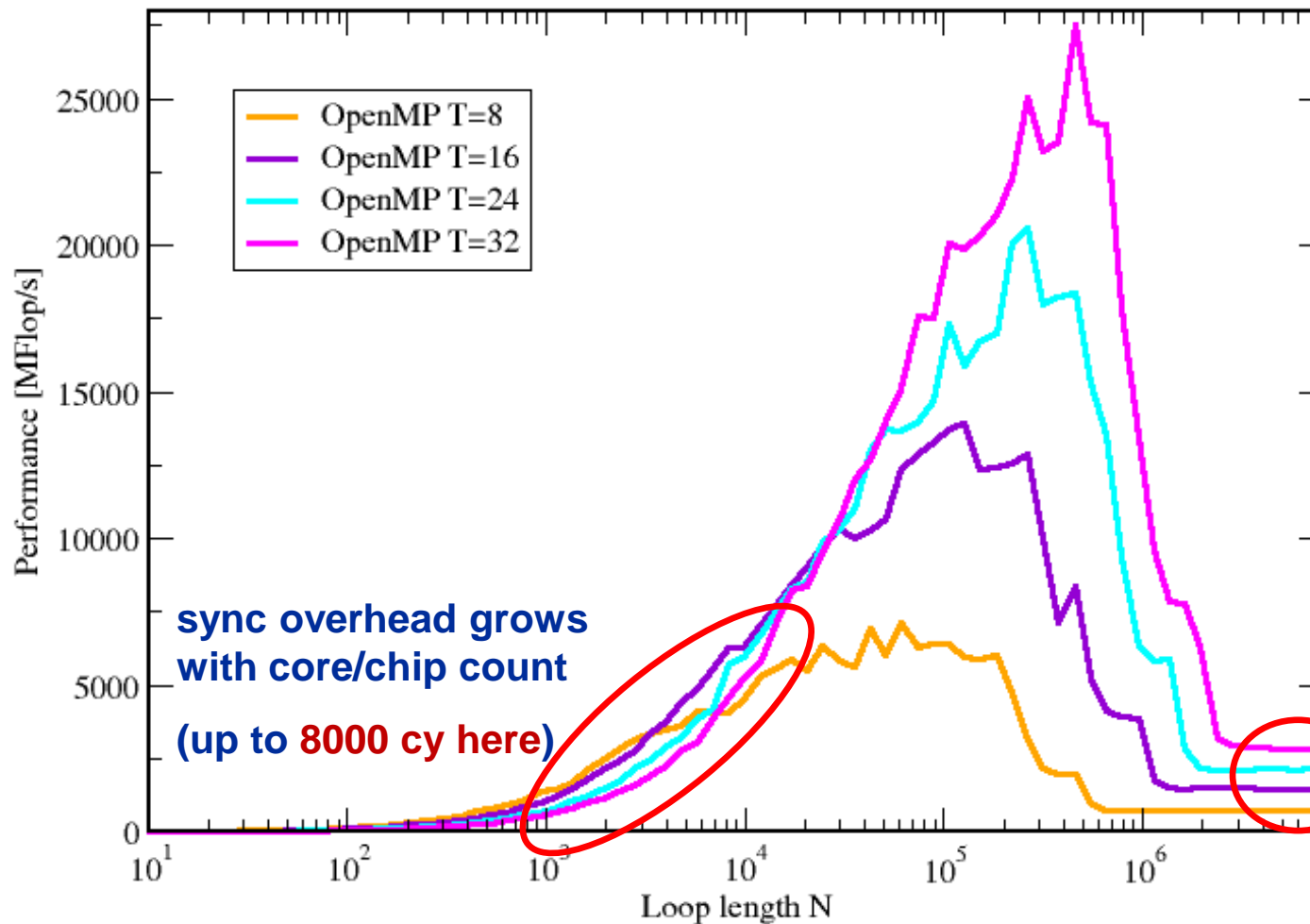
sync overhead nearly topology-independent @ constant thread count



bandwidth scalability across memory interfaces

# The parallel vector triad benchmark

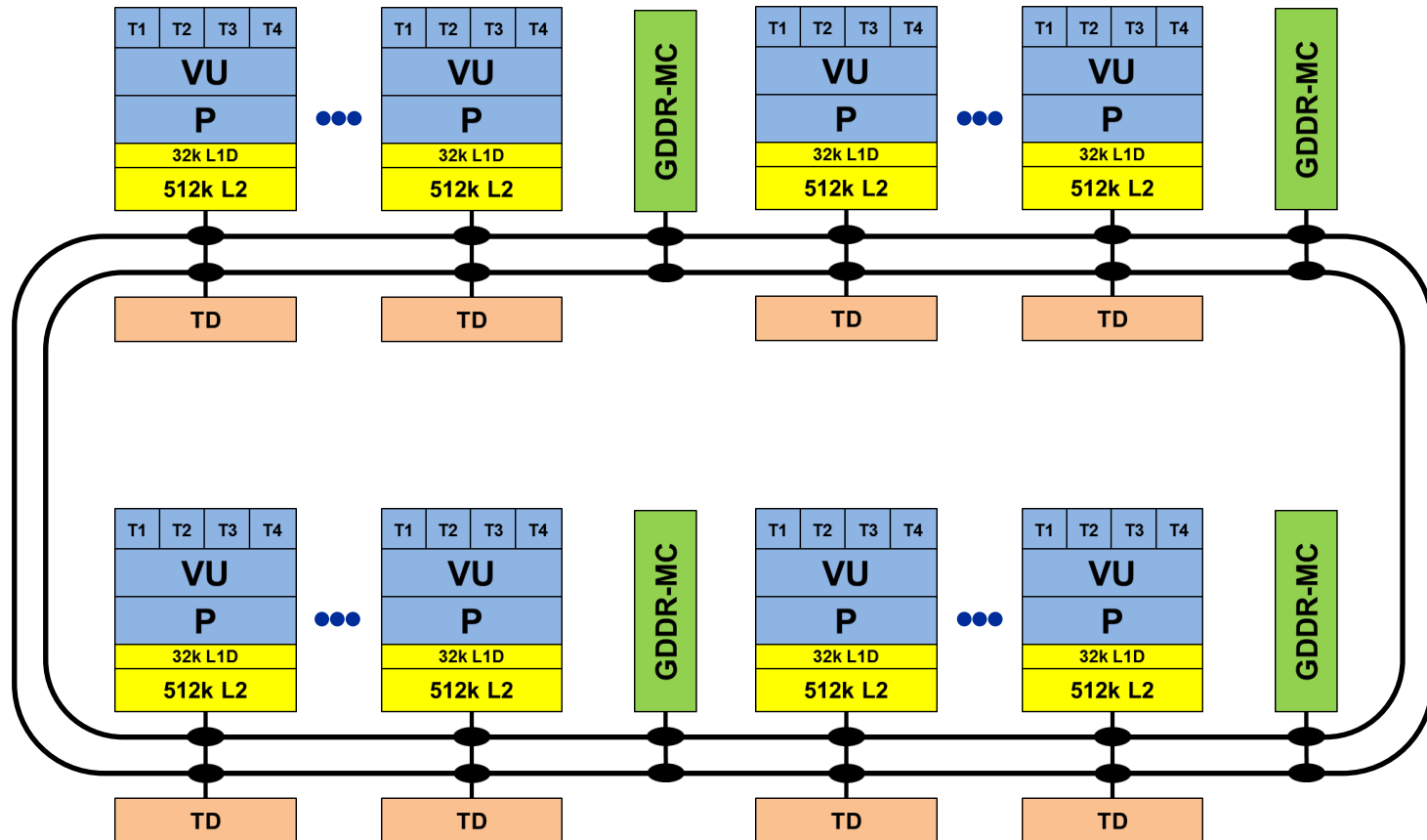
Inter-chip scaling on Cray XE6 Interlagos node

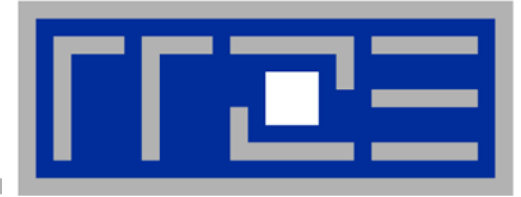


# What will it look like on many-cores?



Go figure.





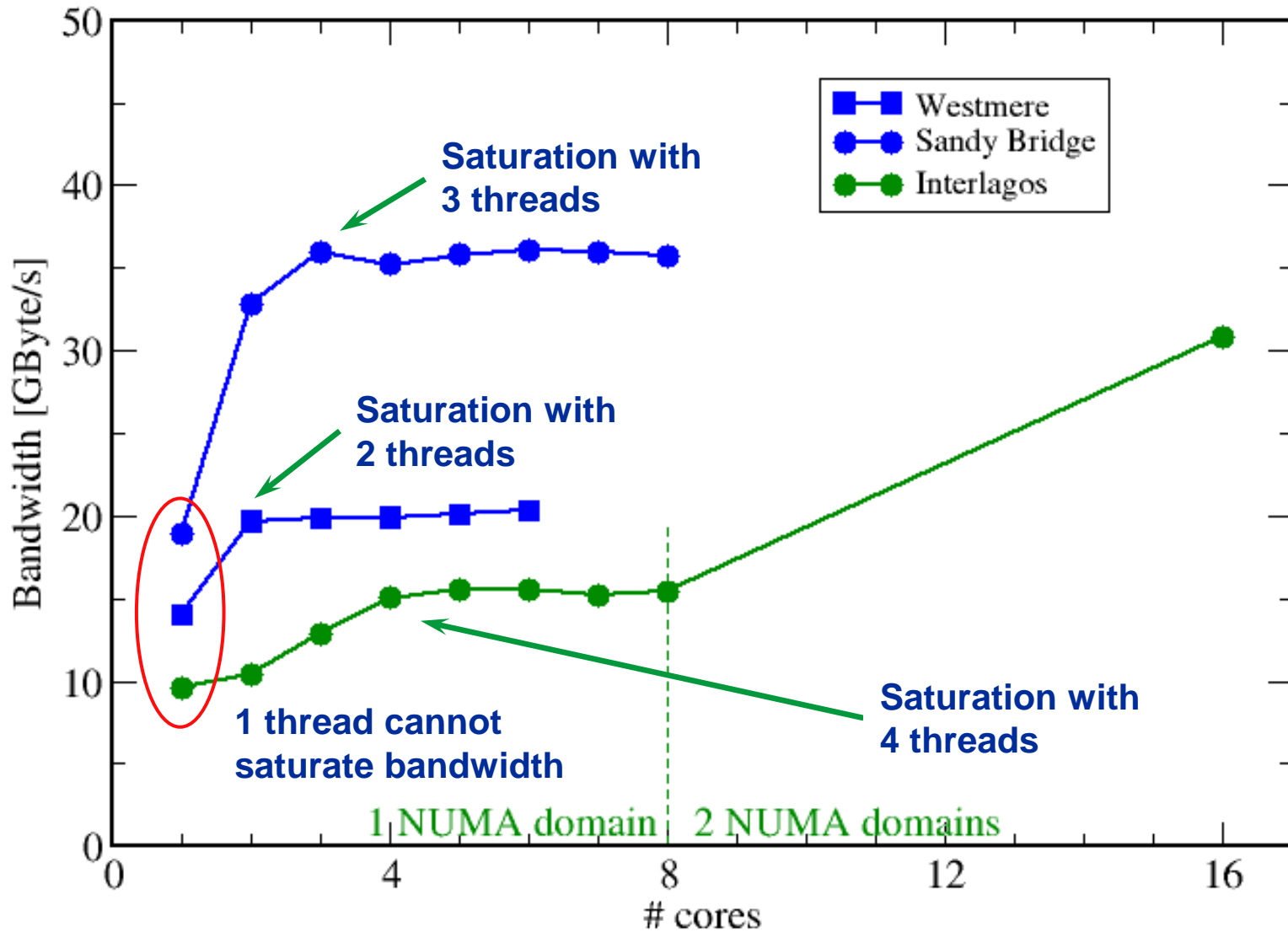
# **Bandwidth saturation effects in cache and memory**

**A look at different processors**



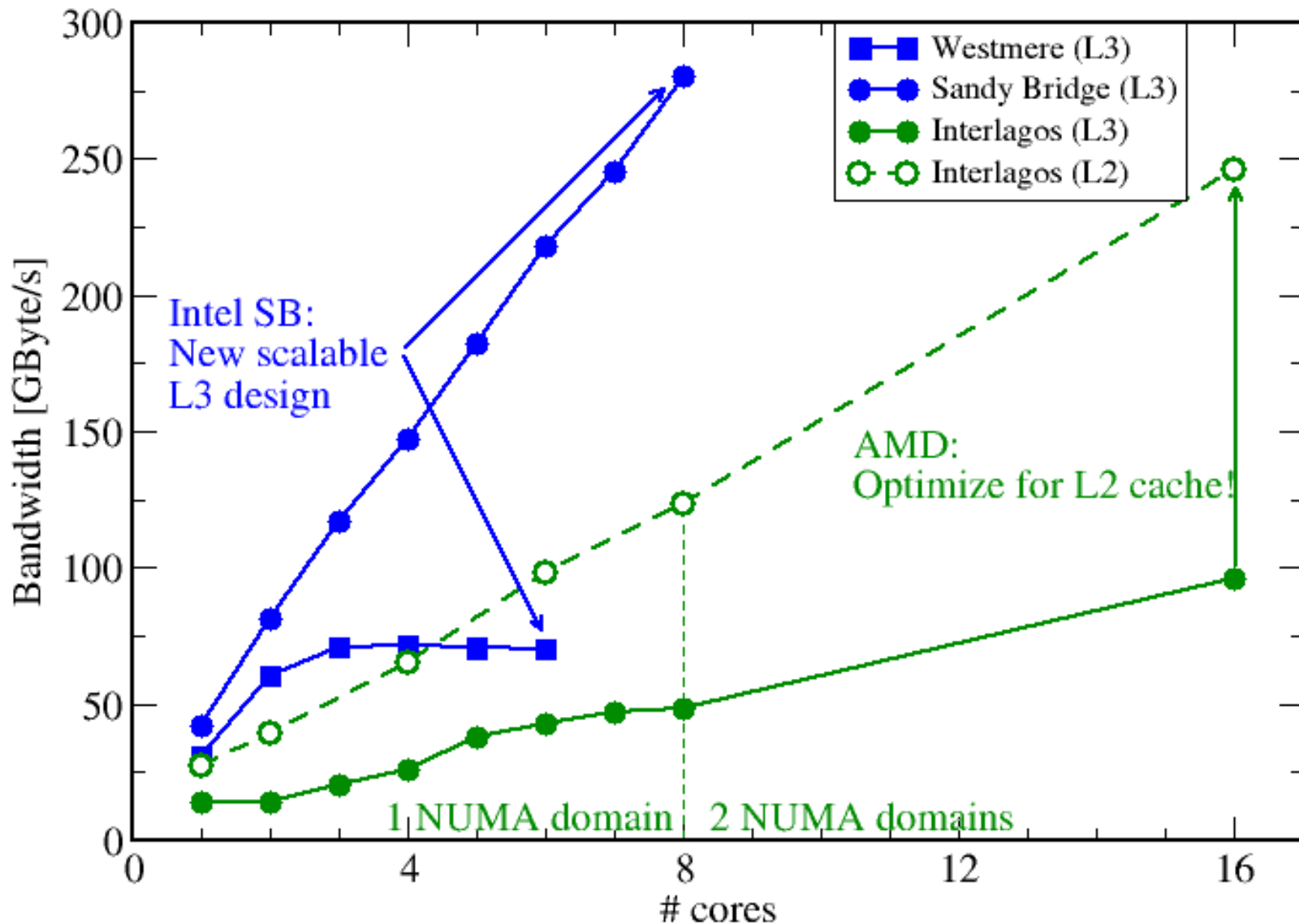
# Bandwidth limitations: Main Memory

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



# Bandwidth limitations: Outer-level cache

## Scalability of shared data paths in L3 cache





- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - **Know the topology of your machine**
    - **Know where your threads are running**
    - **Know where your data is**
  
- **Bandwidth bottlenecks are ubiquitous**
  - Bad scaling is not always a bad thing
  - Do you exhaust your bottlenecks?
  
- **Synchronization overhead may be an issue**
  - ... and also depends on affinity!



- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



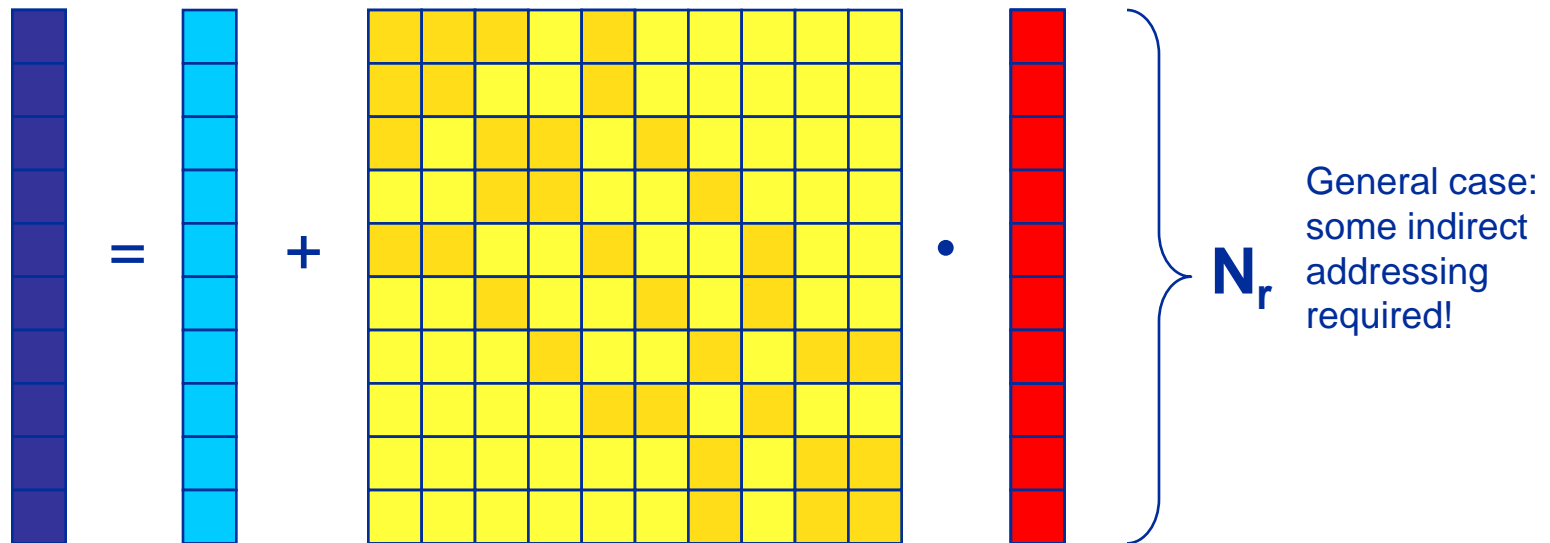
**Case study:  
OpenMP-parallel sparse matrix-vector  
multiplication**

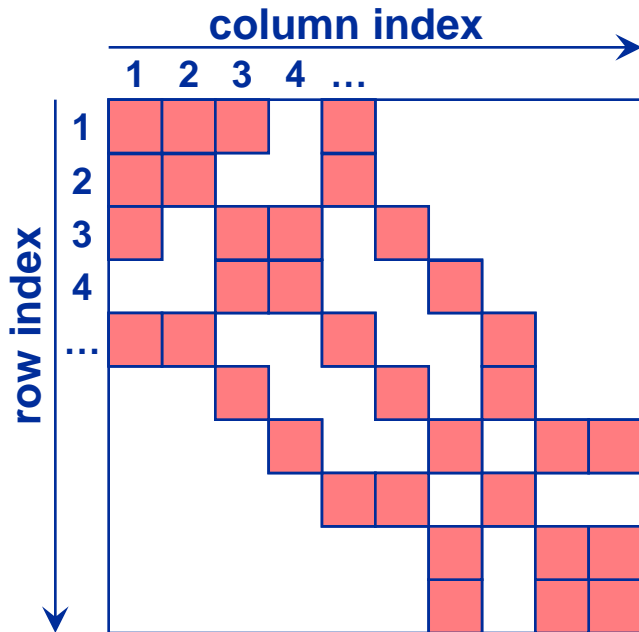
**A simple (but sometimes not-so-simple)  
example for bandwidth-bound code and  
saturation effects in memory**

# Sparse matrix-vector multiply (sMVM)

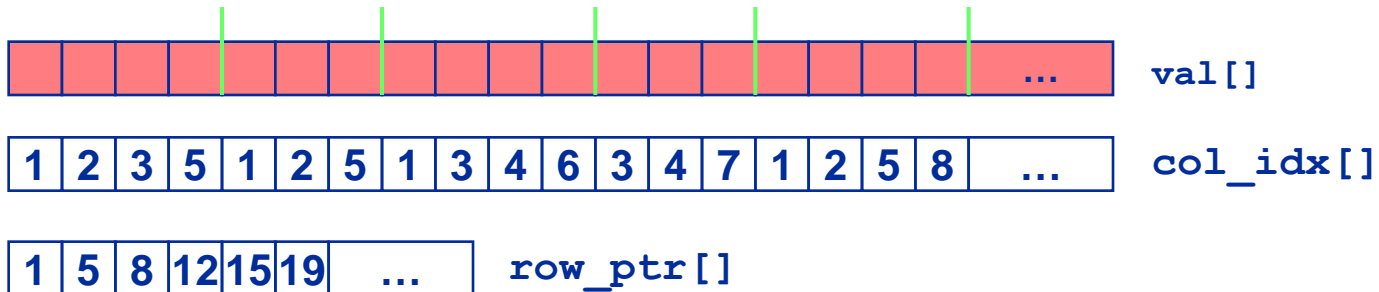


- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- Store only  $N_{nz}$  nonzero elements of matrix and RHS, LHS vectors with  $N_r$  (number of matrix rows) entries
- “Sparse”:  $N_{nz} \sim N_r$





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





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

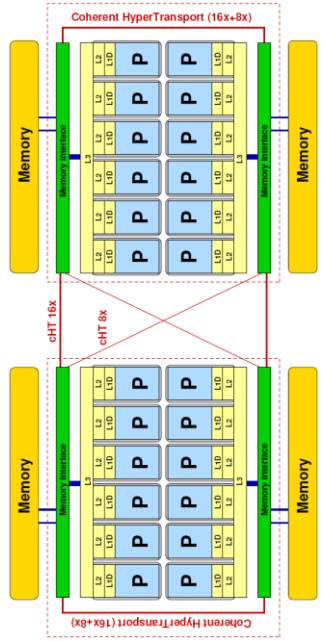
```
!$OMP parallel do
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    c(i) = c(i) + val(j) * b(col_idx(j))
  enddo
enddo
!$OMP end parallel do
```

- Usually many spMVMs required to solve a problem
  - MPI parallelization possible and well-studied
- **Following slides: Performance data on one 24-core AMD Magny Cours node**

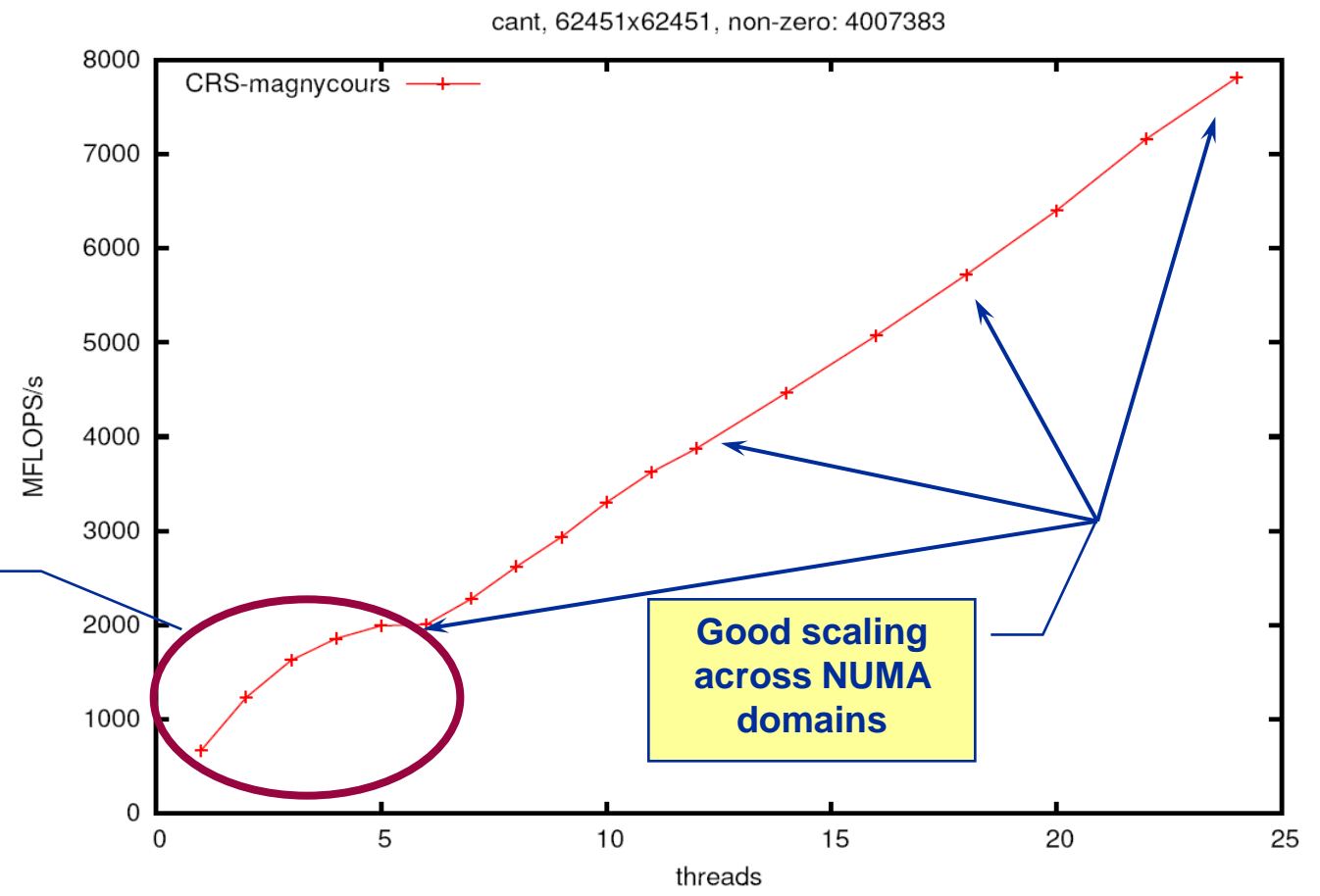




### Case 1: Large matrix



Intrasocket bandwidth bottleneck



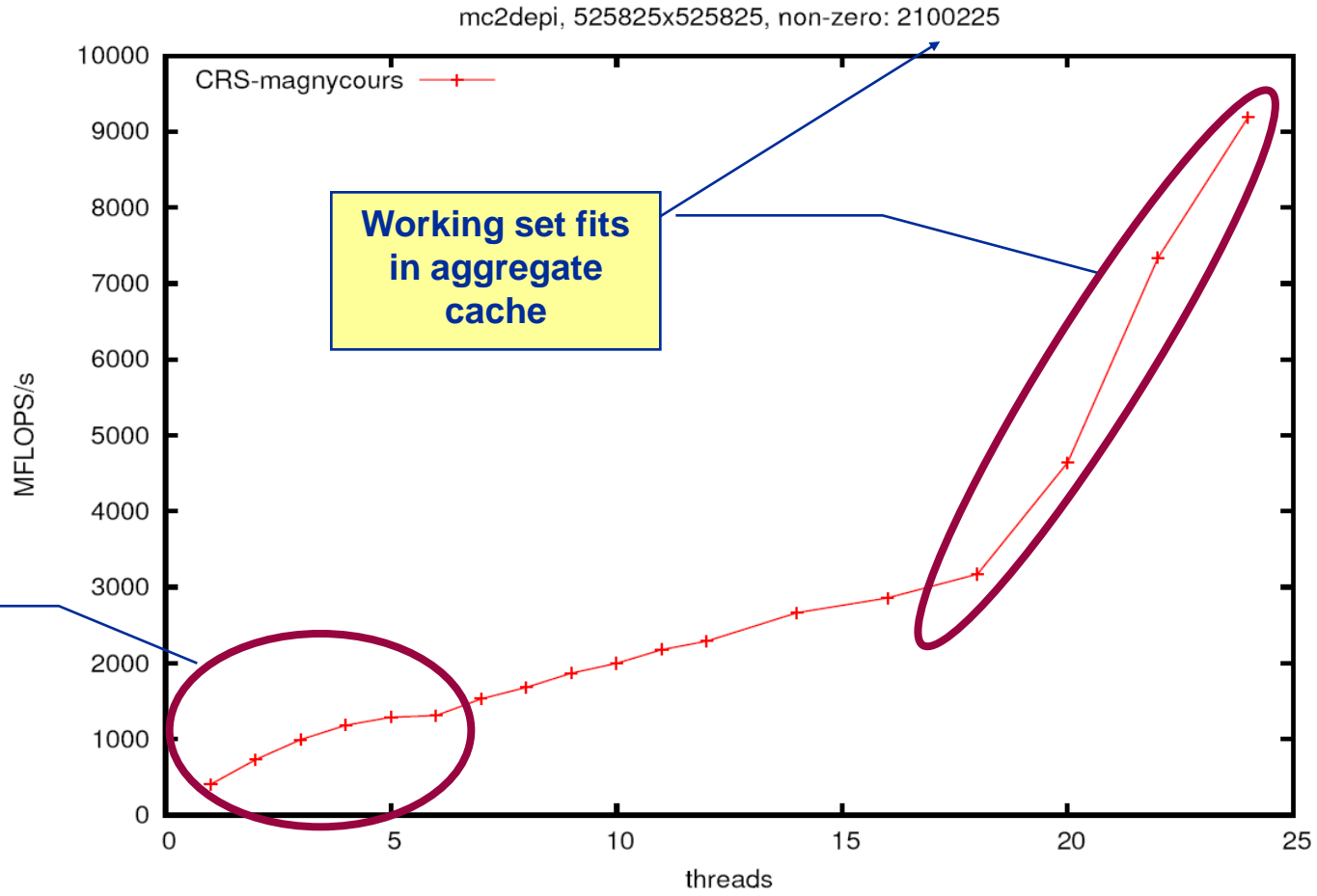
Good scaling across NUMA domains



### Case 2: Medium size



Intrasocket bandwidth bottleneck

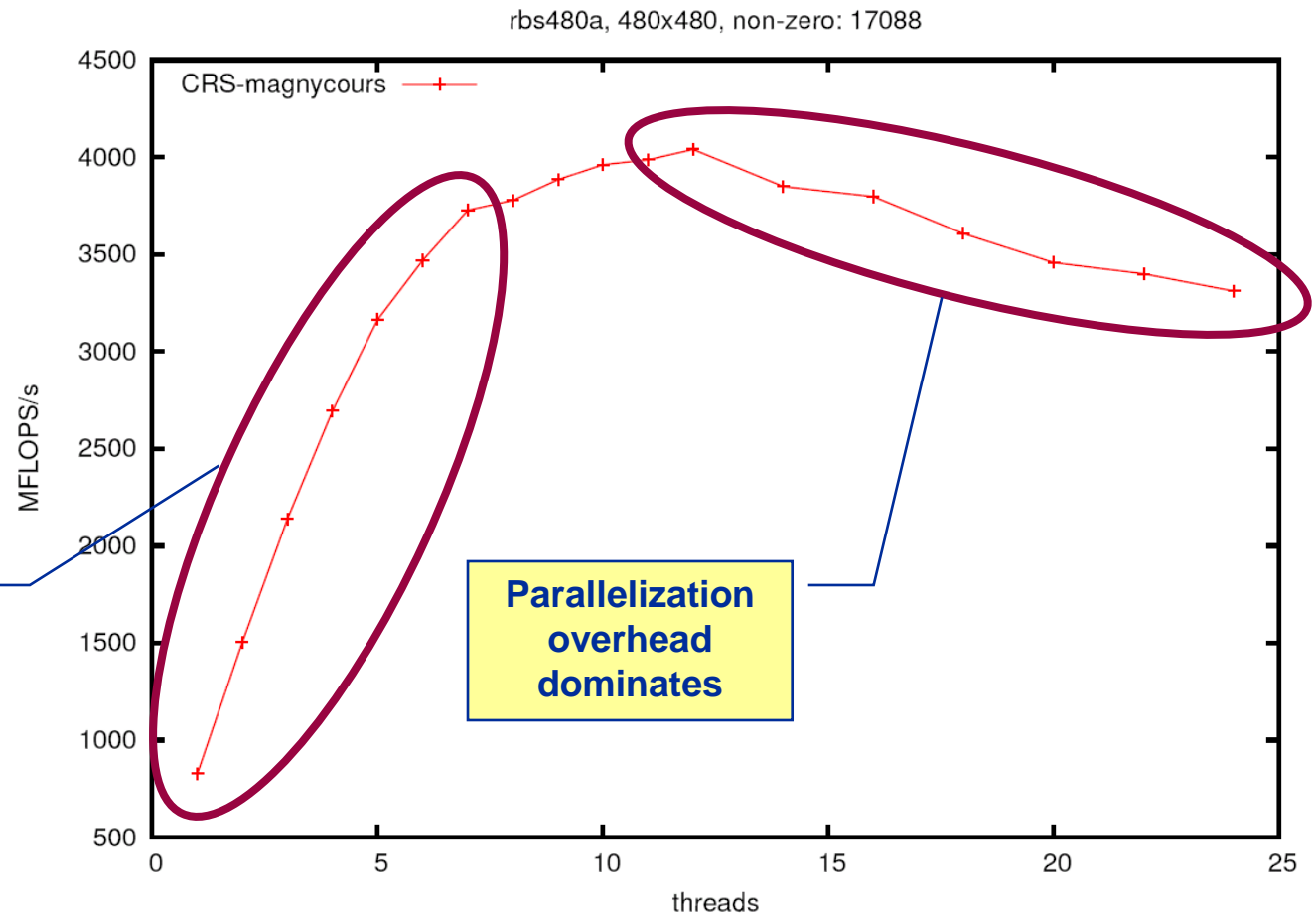




### Case 3: Small size



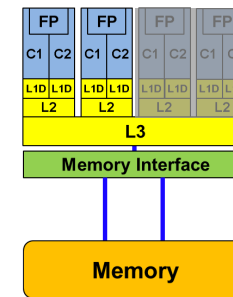
No bandwidth bottleneck



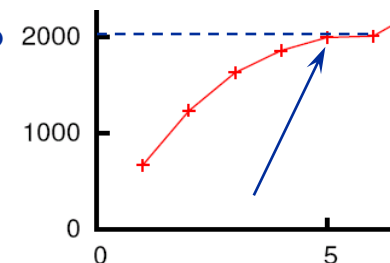
Parallelization overhead dominates



- 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?**
  - Use them for other tasks, such as **MPI communication**
  - Let them **idle** → **saves energy** with minor loss in time to solution
- 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?

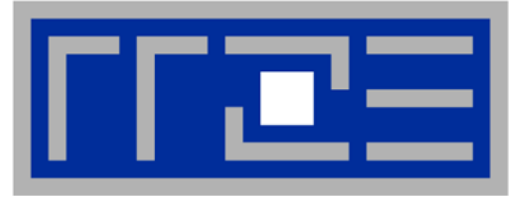


See later for answers!





- Motivation
- Performance Engineering
  - Performance modeling
  - The Performance Engineering process
- Modern architectures
  - Multicore
  - Accelerators
  - Programming models
- Data access
- Performance properties of multicore systems
  - Saturation
  - Scalability
  - Synchronization
- Case study: OpenMP-parallel sparse MVM
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- Some more architecture
  - Simultaneous multithreading (SMT)
  - ccNUMA
- Putting cores to good use
  - Asynchronous communication in spMVM
- A simple power model for multicore
  - Power-efficient code execution
- Conclusions



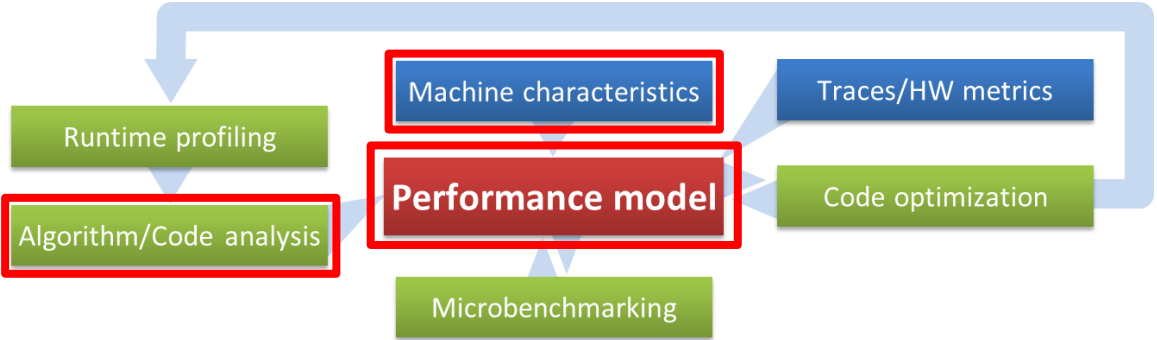
# **Basic performance modeling and “motivated optimizations”**

**The Roofline Model**

**Case study: The Jacobi smoother**

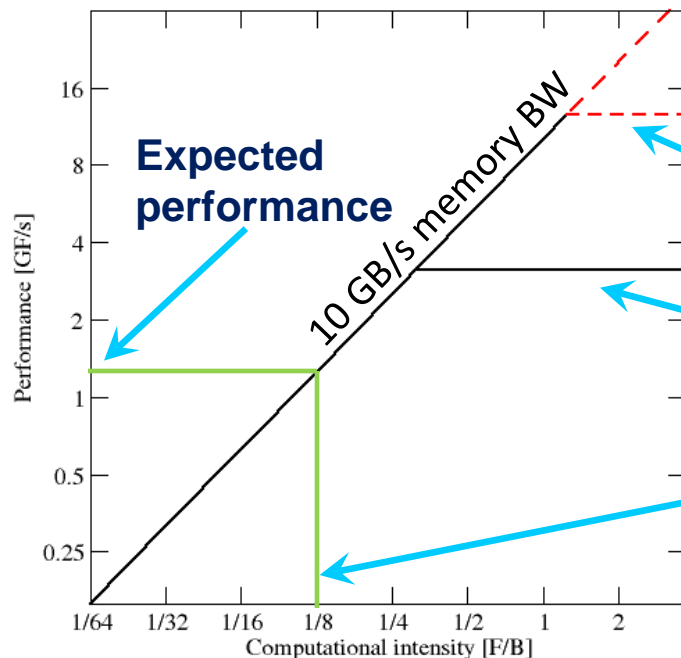


# The Roofline Model





1. Determine the **applicable peak performance** of a loop, assuming that data comes from L1 cache
2. Determine the **computational intensity (flops per byte transferred)** over the slowest data path utilized
3. Determine the **applicable peak bandwidth** of the slowest data path utilized



Example: `do i=1,N; s=s+a(i); enddo`  
in DP on hypothetical 3 GHz CPU, 4-way SIMD, N large

ADD peak (half of full peak)

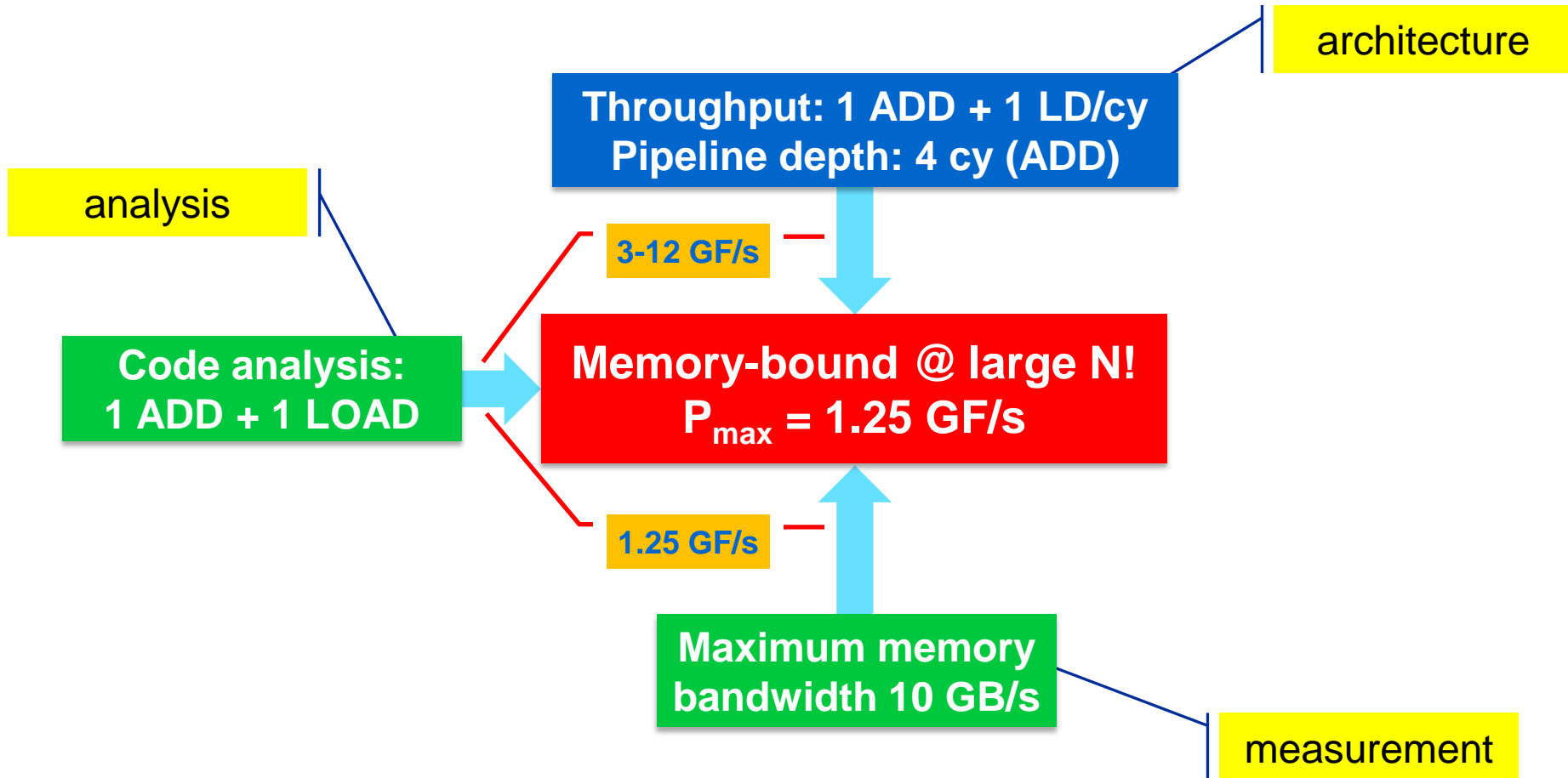
4-cycle latency per ADD if not unrolled

Computational intensity [Flops/byte]





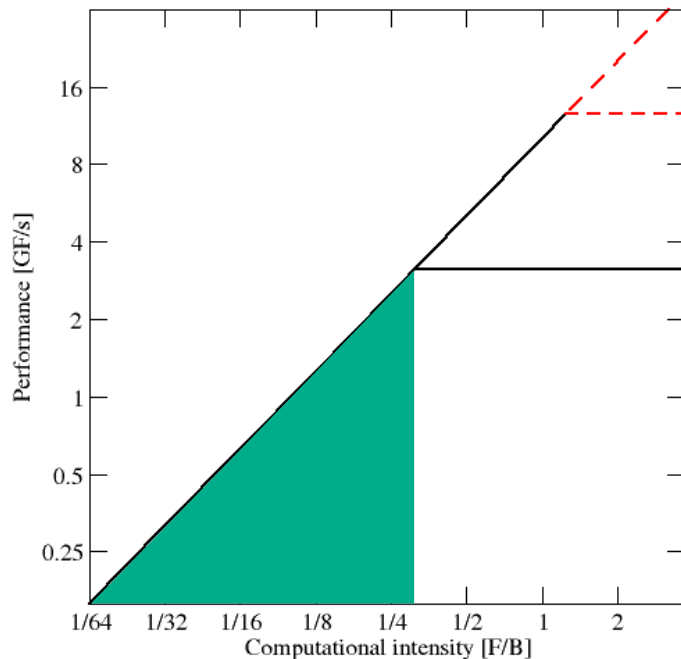
... on the example of `do i=1,N; s=s+a(i); enddo`





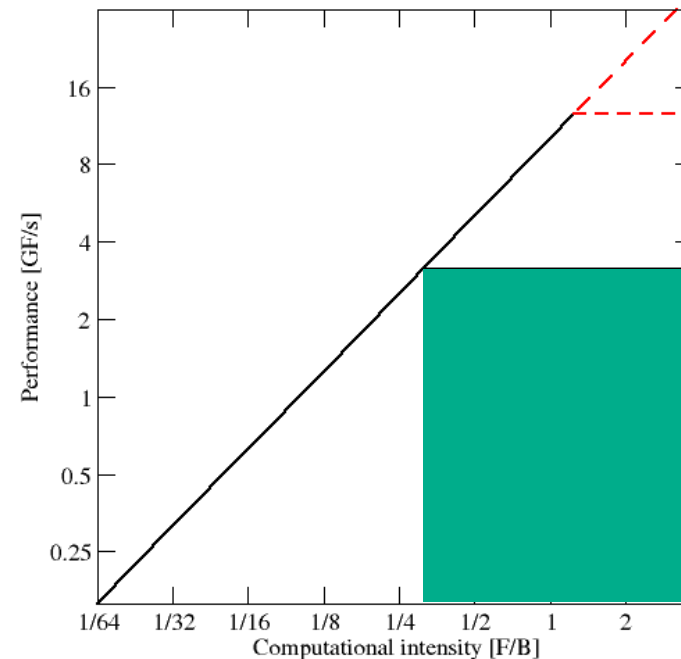
## Bandwidth-bound (simple case)

- Accurate traffic calculation (write-allocate, strided access, ...)
- Practical  $\neq$  theoretical BW limits
- **Erratic access patterns**



## Core-bound (may be complex)

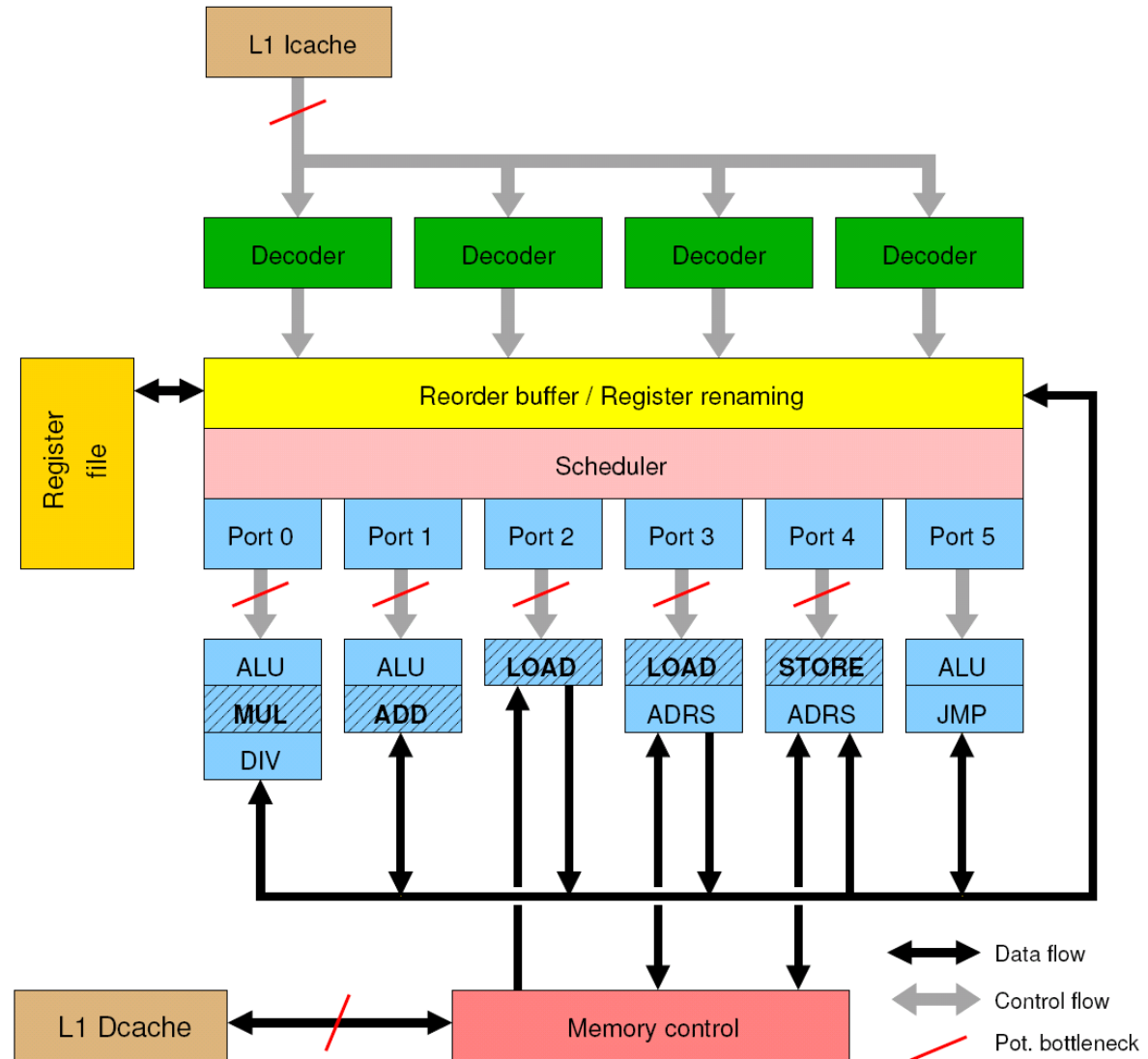
- **Multiple bottlenecks:** LD/ST, arithmetic, pipelines, SIMD, execution ports
- See next slide...





## Multiple bottlenecks:

- L1 Icache 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





- **Code balance ( $B_C$ )** quantifies the requirements of the code
  - Reciprocal of comp. intensity

$$B_C = \frac{\text{data transfer (LD/ST) [words]}}{\text{arithmetic operations [flops]}}$$

- **$b_S$  = achievable bandwidth over the slowest data path**
  - E.g., measured by suitable microbenchmark (STREAM, ...)

- **Lightspeed** for absolute performance: ( $P_{\max}$  : “applicable” peak performance)

$$P = \min\left(P_{\max}, \frac{b_S}{B_C}\right)$$

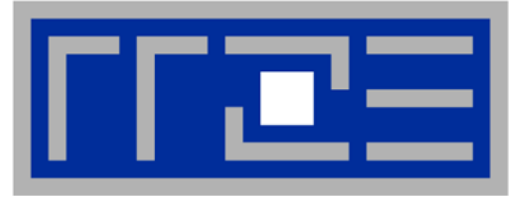
Newton's  
Second Law  
of  
performance  
modeling

- **Example: Vector triad  $A(:) = B(:) + C(:) * D(:)$  on 2.3 GHz Interlagos**
  - $B_C = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F}$  (including write allocate)

$$b_S/B_C = 1.7 \text{ GF/s (1.2 \% of peak performance)}$$



- **The balance metric 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**



## **Case study: A 3D Jacobi smoother**

**The basics in two dimensions  
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

```

Reuse when computing  $\text{phi}(i+2,k,t1)$

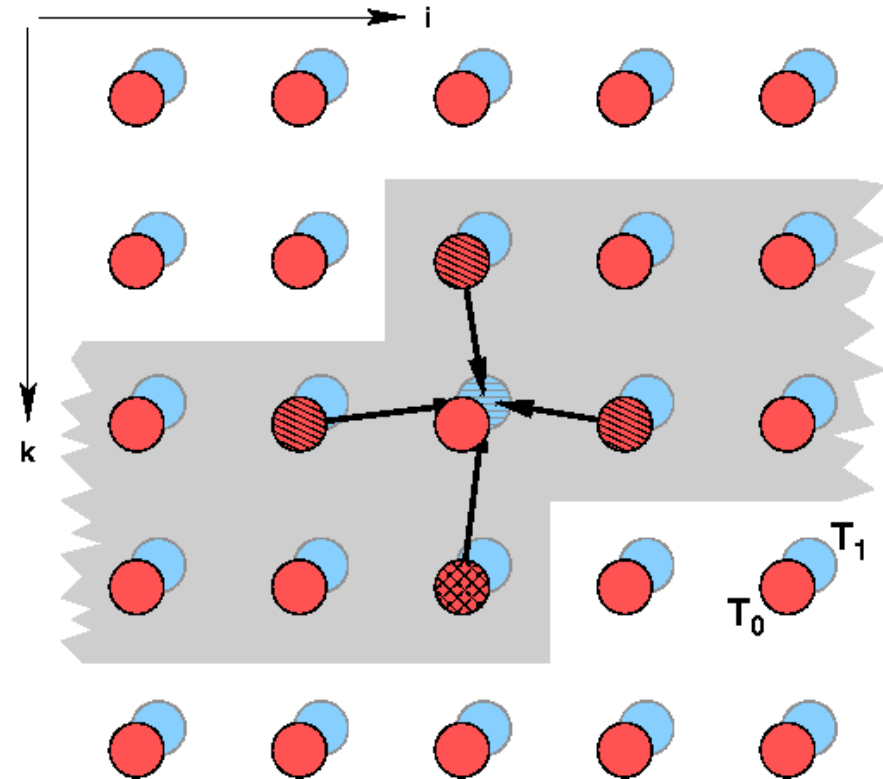
**Naive balance (incl. write allocate):**

**$\text{phi}(:, :, t0)$ : 3 LD +**  
 **$\text{phi}(:, :, t1)$ : 1 ST+ 1LD**

**$\rightarrow B_C = 5 W / 4 \text{ FLOPs} = 1.25 W / F$**

**WRITE ALLOCATE:**  
**LD + ST  $\text{phi}(i, k, t1)$**

- Modern cache subsystems may further reduce memory traffic



If cache is large enough to hold at least 2 rows (shaded region): Each  $\phi(i, k, t_0)$  is loaded once from main memory and re-used 3 times from cache:

$$\phi(i, k, t_0): 1 \text{ LD} + \phi(i, k, t_1): 1 \text{ ST} + 1 \text{ LD} \\ \rightarrow B_C = 3W / 4F = 0.75W / F$$

If cache is too small to hold one row:  
 $\phi(i, k, t_0): 2 \text{ LD} + \phi(i, k, t_1): 1 \text{ ST} + 1 \text{ LD} \\ \rightarrow B_C = 5W / 4F = 1.25W / F$





- **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**

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



- 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
```

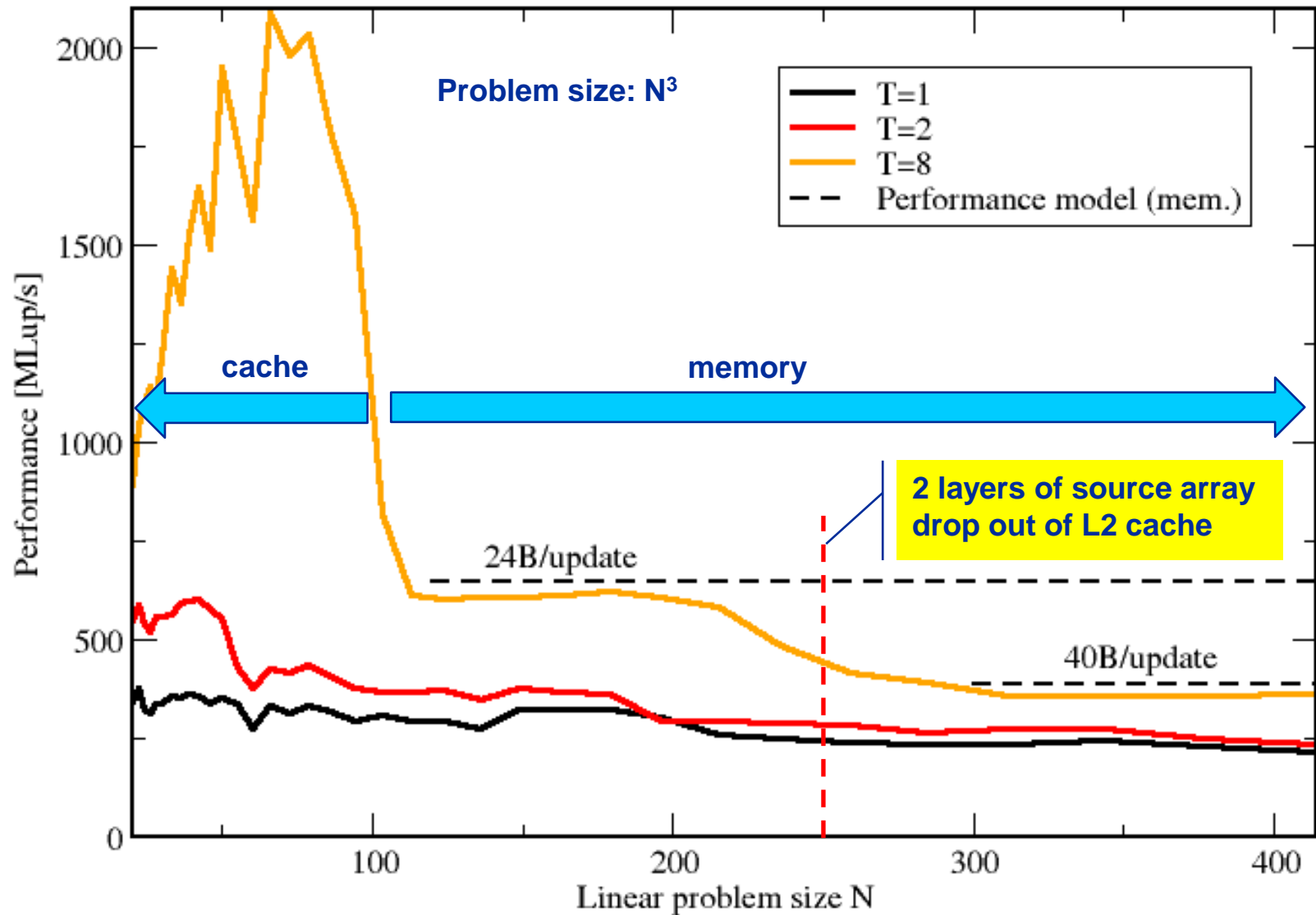
- Best case balance: 1 LD phi(i,j,k+1,t0)  
 1 ST + 1 write allocate phi(i,j,k,t1)  
 6 flops

→  $B_C = 0.5 W/F$  (24 bytes/update)

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

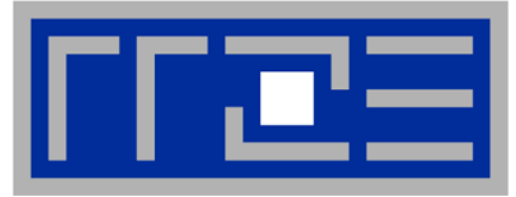
# 3D Jacobi solver

Performance of vanilla code on one Interlagos chip (8 cores)





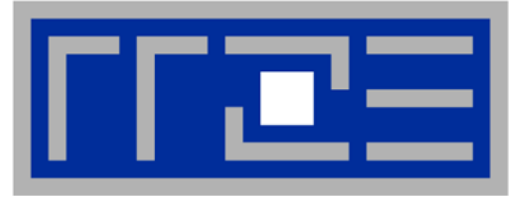
- We have **made sense** of the **memory-bound performance vs. problem size**
  - “Layer conditions” lead to **predictions of code balance**
  - Achievable memory bandwidth is input parameter
  
- **The model works only if the bandwidth is “saturated”**
  - In-cache modeling is more involved
  
- **Optimization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



# **Data access optimizations**

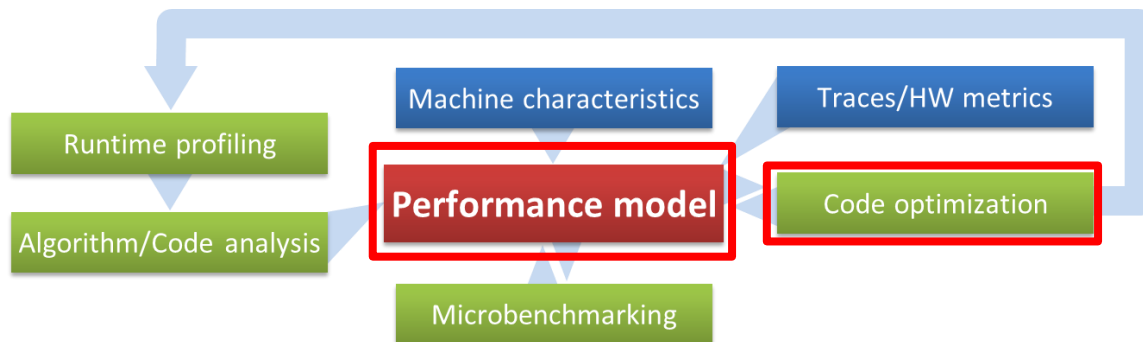
**Case study: Optimizing a Jacobi solver**

**Case study: Erratic RHS access for sparse MVM**

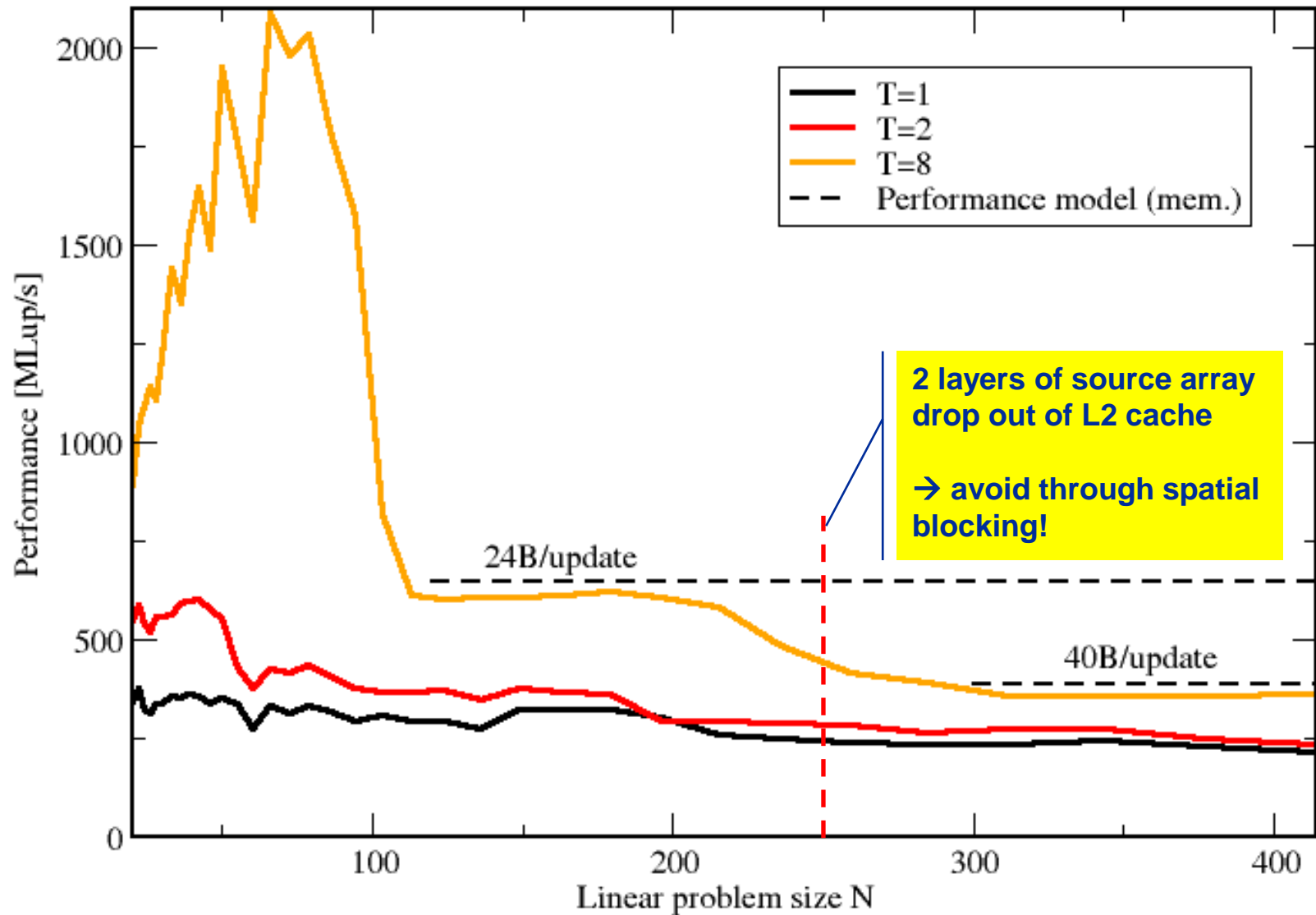


## Case study: 3D Jacobi solver

### Spatial blocking for improved cache re-use



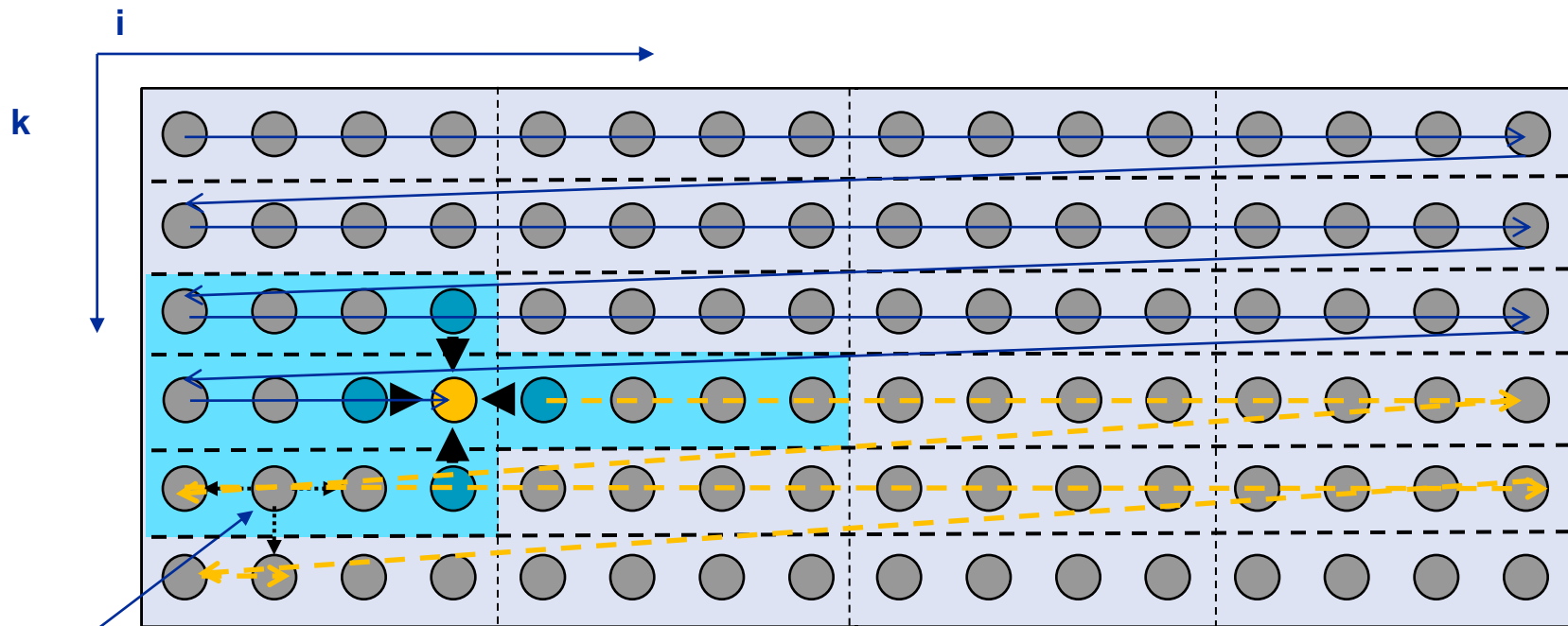
# Remember the 3D Jacobi solver on Interlagos?





## Assumptions:

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



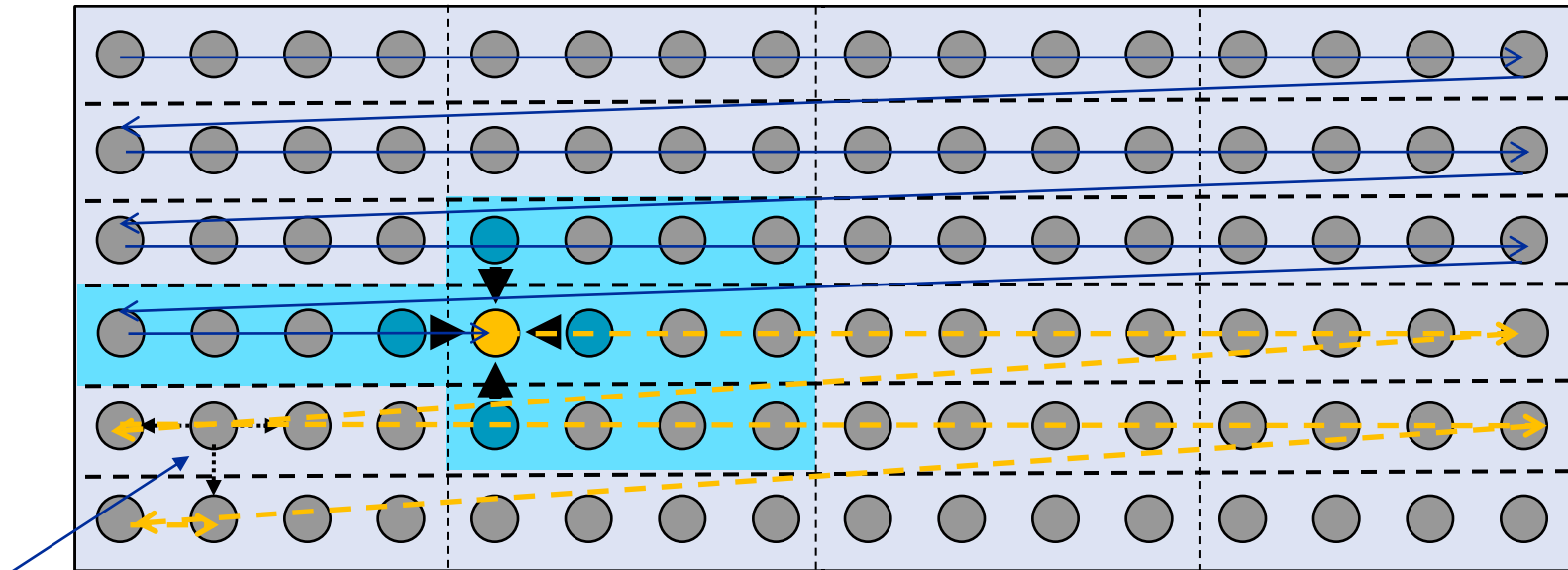
This element is needed for three more updates; but 29 updates happen before this element is used for the last time





## Assumptions:

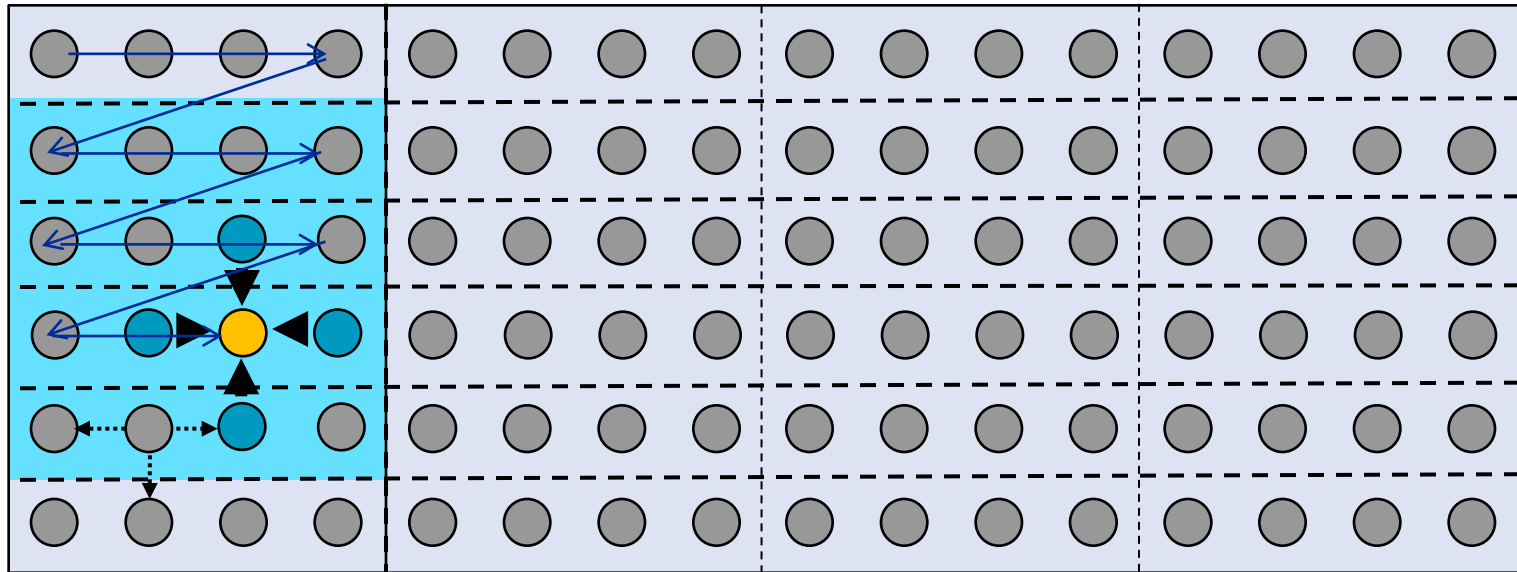
- cache can hold 32 elements (16 for each array)
- Cache line size is 4 elements
- Perfect eviction strategy for source array



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

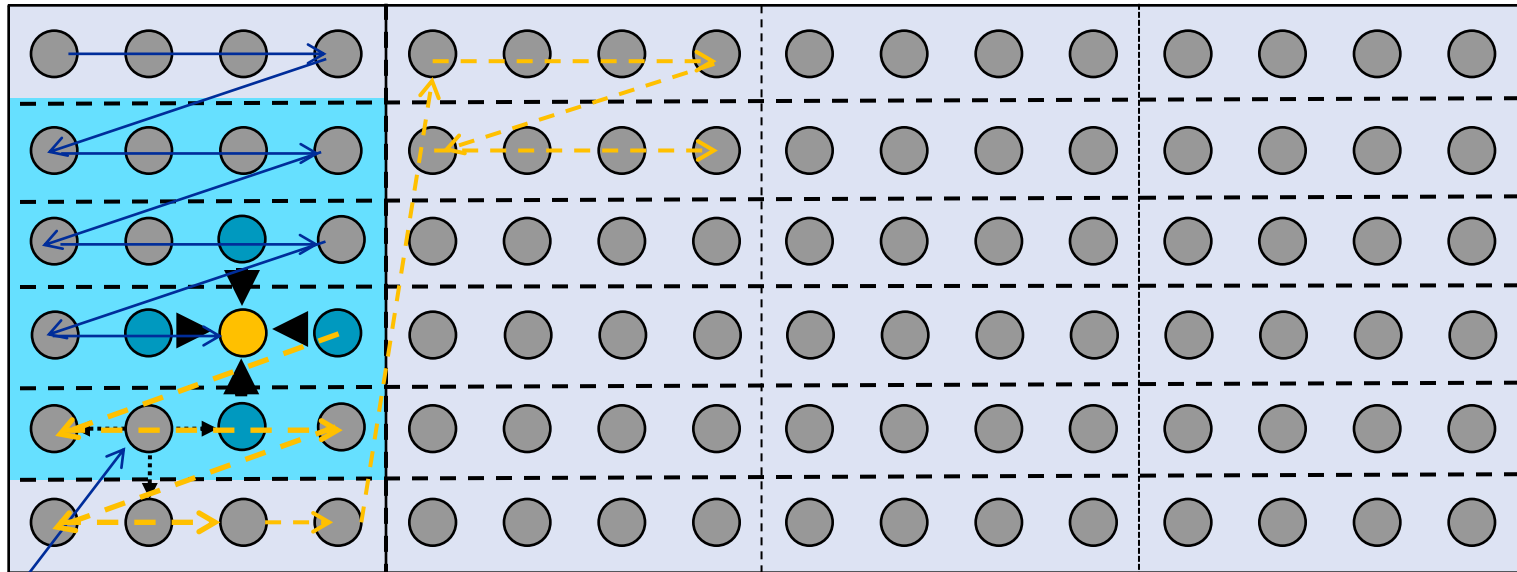


- divide system into blocks
- update block after block
- same performance as if three complete rows of the systems fit into cache





- **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)**



## 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
```

Diagram annotations:

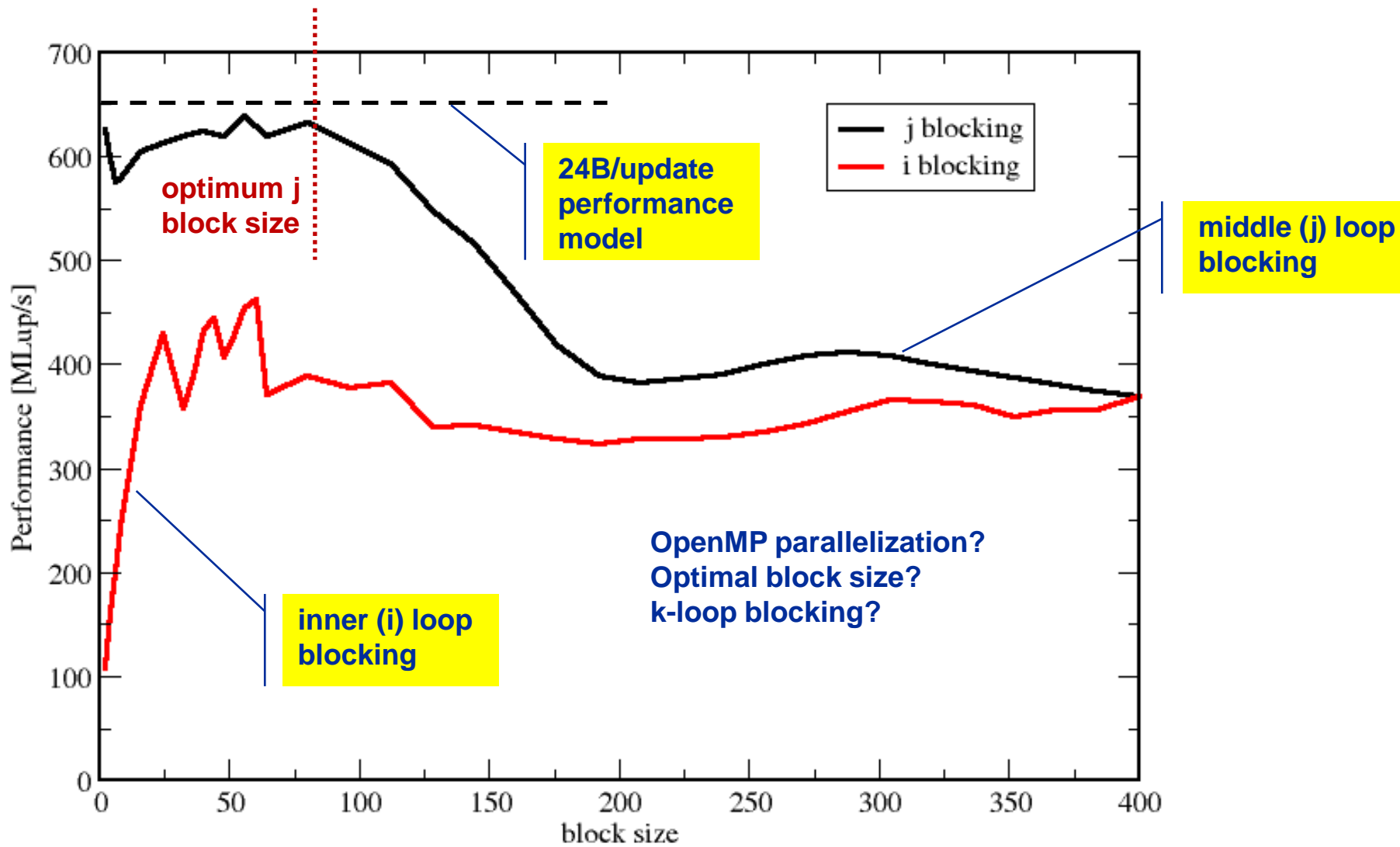
- A yellow box labeled "loop over i-blocks" has a line pointing to the `do ioffset=1,imax,iblock` line.
- A yellow box labeled "loop over j-blocks" has a line pointing to the `do joffset=1,jmax,jblock` line.

## 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 (see later)

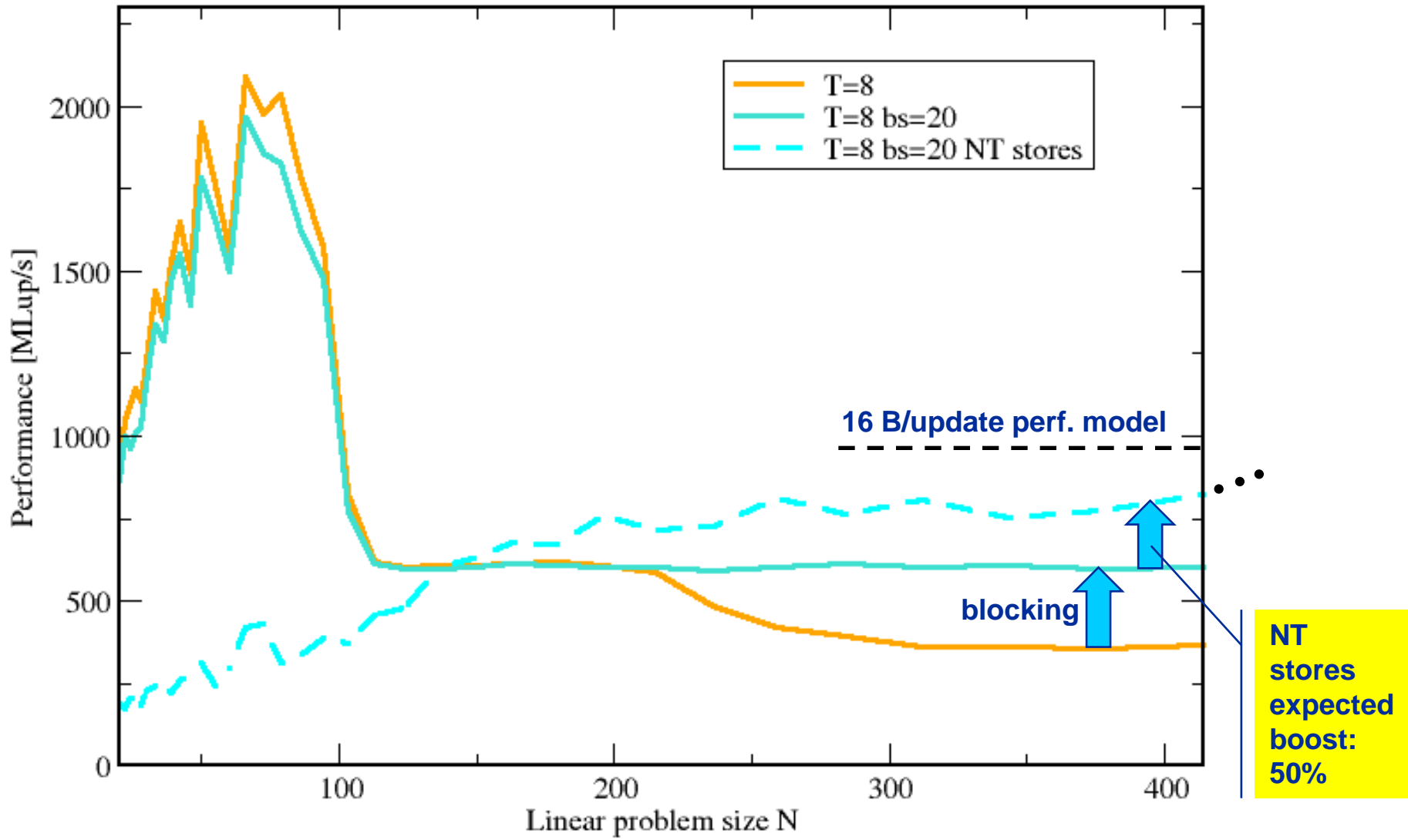
# 3D Jacobi solver (problem size $400^3$ )

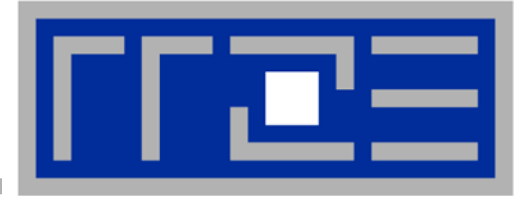
Blocking different loop levels (8 cores Interlagos)



# 3D Jacobi solver

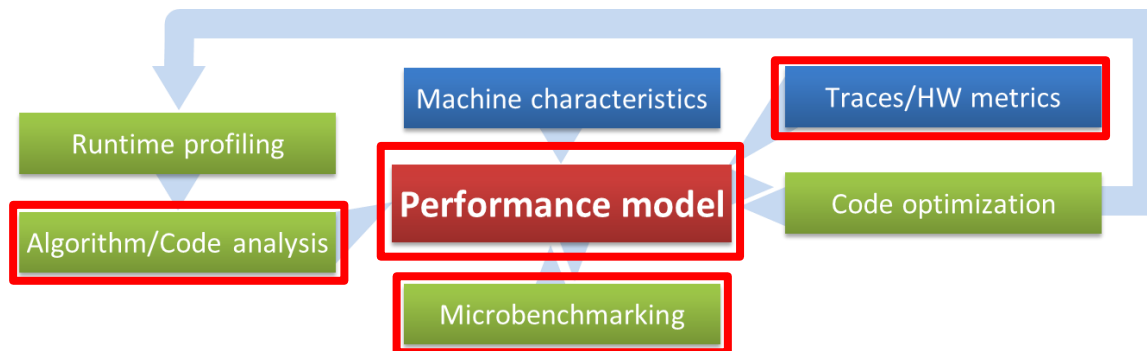
*Spatial blocking + nontemporal stores*





## Case study: Erratic RHS access in sparse MVM

“Modeling” indirect access





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

```
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 code balance**

- $\kappa$  quantifies extra traffic for loading RHS more than once

- Naive performance =  $b_s/B_{CRS}$

- Determine  $\kappa$  by measuring performance and **actual memory bandwidth**

$$B_{CRS} = \left( \frac{12 + 24/N_{nzs} + \kappa}{2} \right) \frac{\text{bytes}}{\text{flop}}$$

$$= \left( 6 + \frac{12}{N_{nzs}} + \frac{\kappa}{2} \right) \frac{\text{bytes}}{\text{flop}} .$$

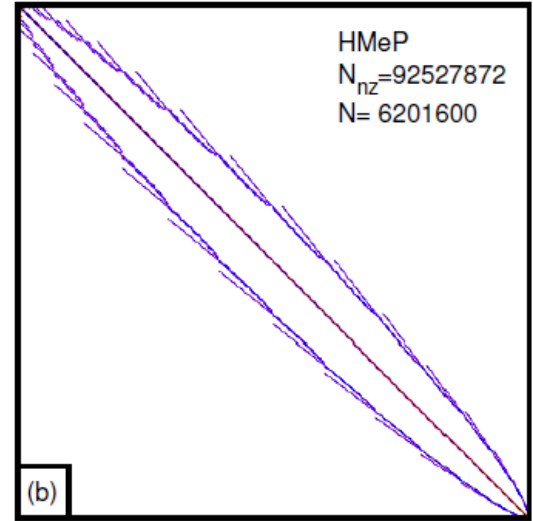
G. Schubert, G. Hager, H. Fehske and G. Wellein: *Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming*. Workshop on Large-Scale Parallel Processing (LSP 2011), May 20th, 2011, Anchorage, AK. [DOI:10.1109/IPDPS.2011.332](https://doi.org/10.1109/IPDPS.2011.332), Preprint: [arXiv:1101.0091](https://arxiv.org/abs/1101.0091)





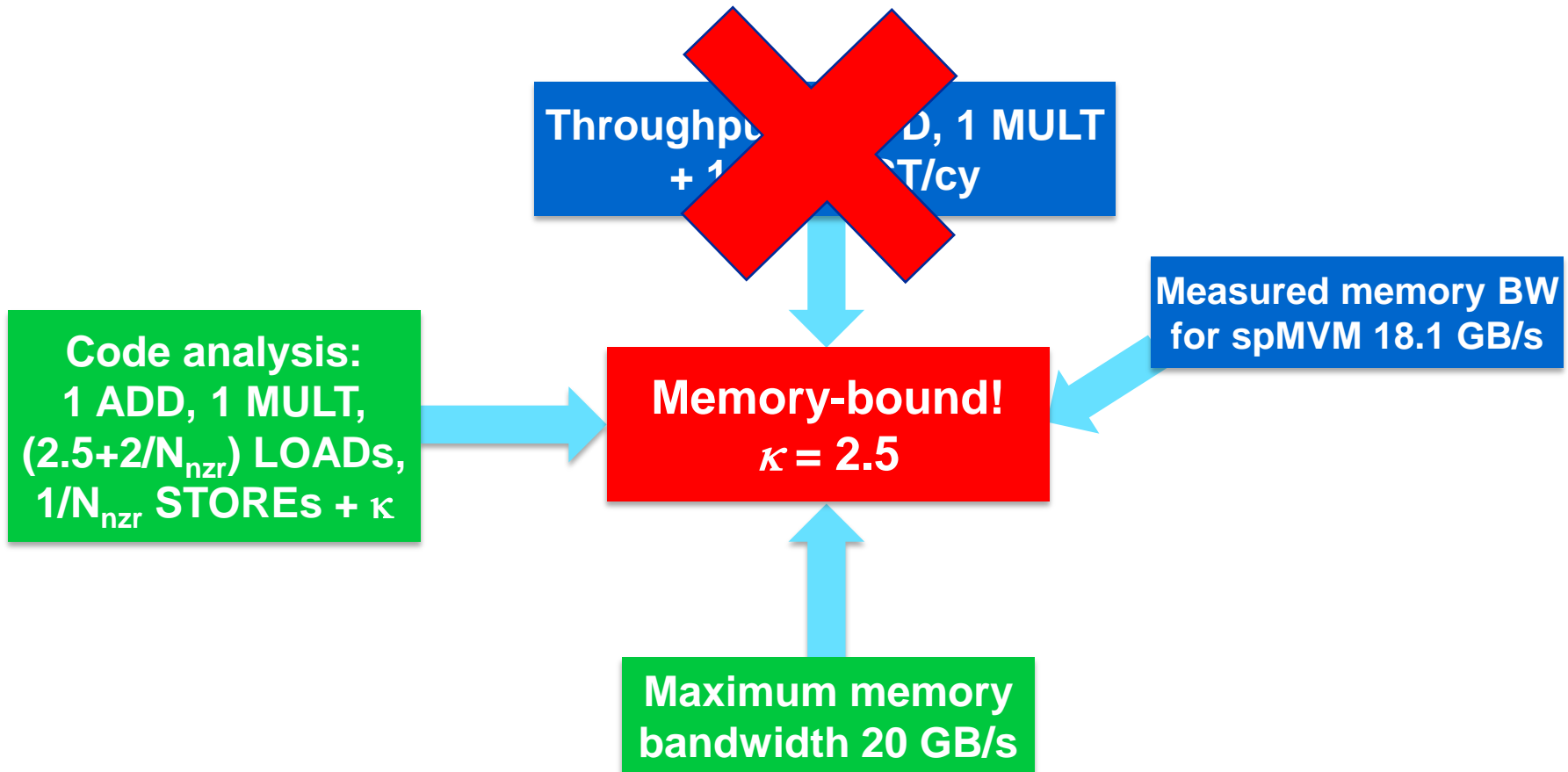
- **Analysis for HMeP matrix on Nehalem EP socket**

- BW used by spMVM kernel = 18.1 GB/s → should get  $\approx 2.66$  Gflop/s  
spMVM performance if  $\kappa = 0$
- Measured spMVM performance = 2.25 Gflop/s
- Solve  $2.25 \text{ Gflop/s} = b_S/B_{\text{CRS}}$  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



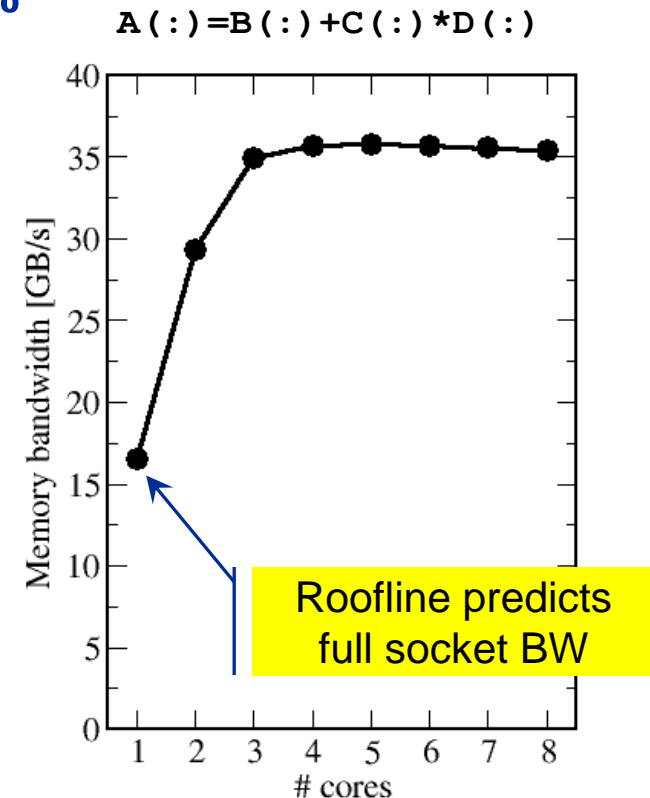
- **Conclusion:** Even if the roofline/bandwidth model does not work 100%, we can still learn something from the deviations
  - Optimization? Perhaps you can reorganize the matrix

... on the example of spMVM with HMeP matrix





- Assumes one of two bottlenecks
  - In-core execution
  - Bandwidth of a single hierarchy level
- Latency effects are not modeled → pure data streaming assumed
- Data transfer and in-core time overlap 100%
- In-core execution is sometimes hard to model
- Saturation effects in multicore chips are not explained**
  - ECM model gives more insight



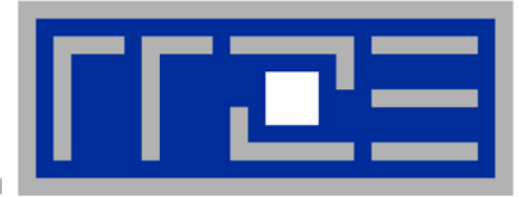
G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Submitted. Preprint: [arXiv:1208.2908](https://arxiv.org/abs/1208.2908)



- There is no substitute for knowing what's going on between your code and the hardware
- **Make sense of performance behavior** through sensible application of performance models
  - However, there is no “golden formula” to do it all for you automatically
  - If the model does not work properly, **you learn something new**
- **Model inputs:**
  - Code analysis/inspection
  - Hardware counter data
  - Microbenchmark analysis
  - Architectural features
- **Simple models work best; do not try to make it more complex than necessary**



- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - **Simultaneous multithreading (SMT)**
  - **ccNUMA**
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



# **Boosting core efficiency: Simultaneous multithreading (SMT)**

**Principles and performance impact**

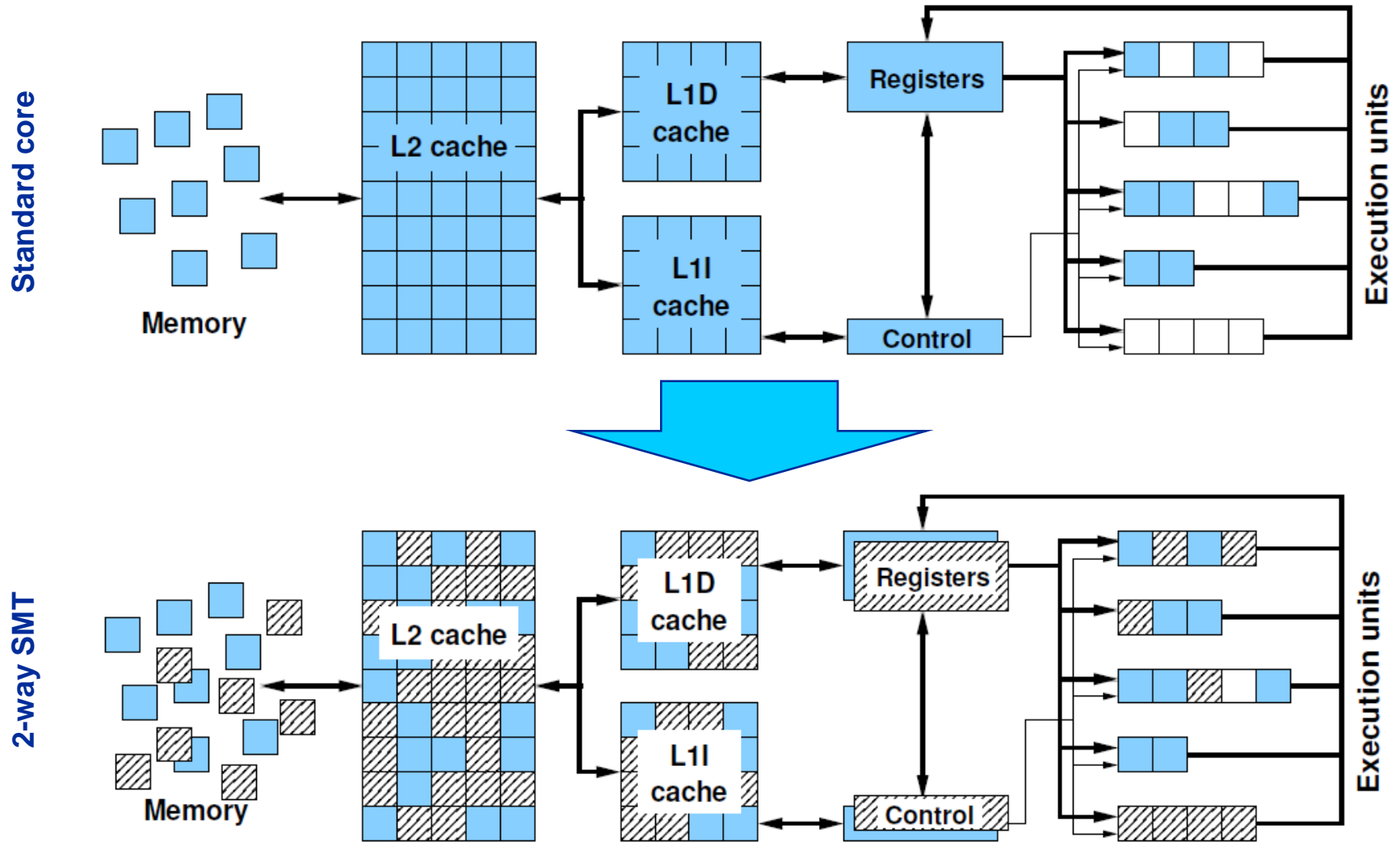
**SMT vs. independent instruction streams**

**Facts and fiction**

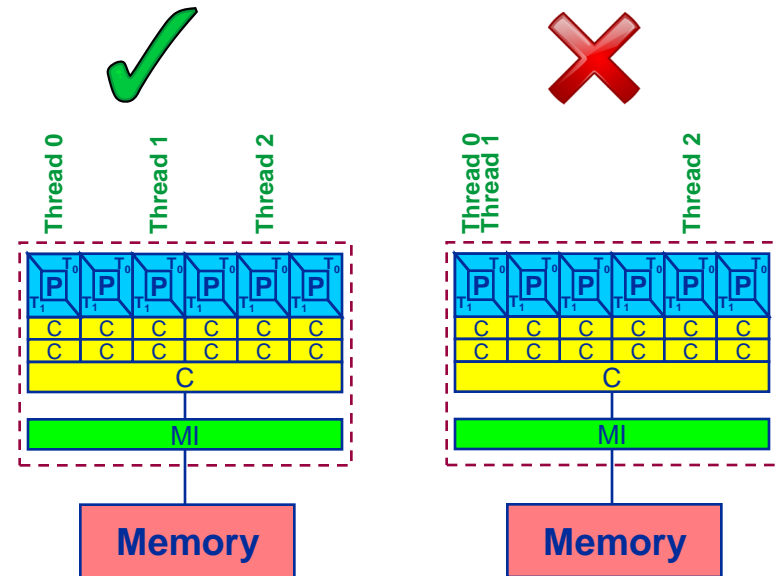
# SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently



- SMT principle (2-way example):



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

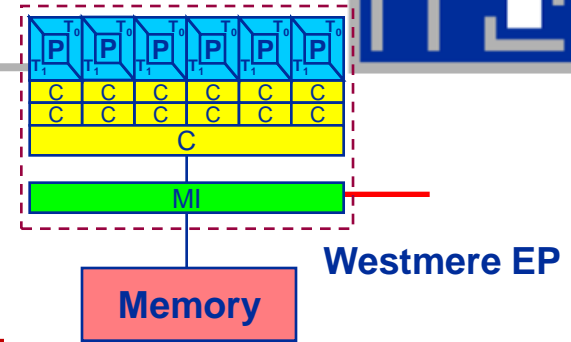




# SMT impact



- SMT adds **another layer of topology** (inside the physical core)
- **Caveat: SMT threads share all caches!**
- **Possible benefit: Better pipeline throughput**



- Filling otherwise unused pipelines
- Filling pipeline bubbles with other thread's executing instructions:

## Thread 0:

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

Dependency → pipeline stalls until previous MULT is over

## Thread 1:

```
do i=1,N
  b(i) = func(i)*d
enddo
```

Unrelated work in other thread can fill the pipeline bubbles

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

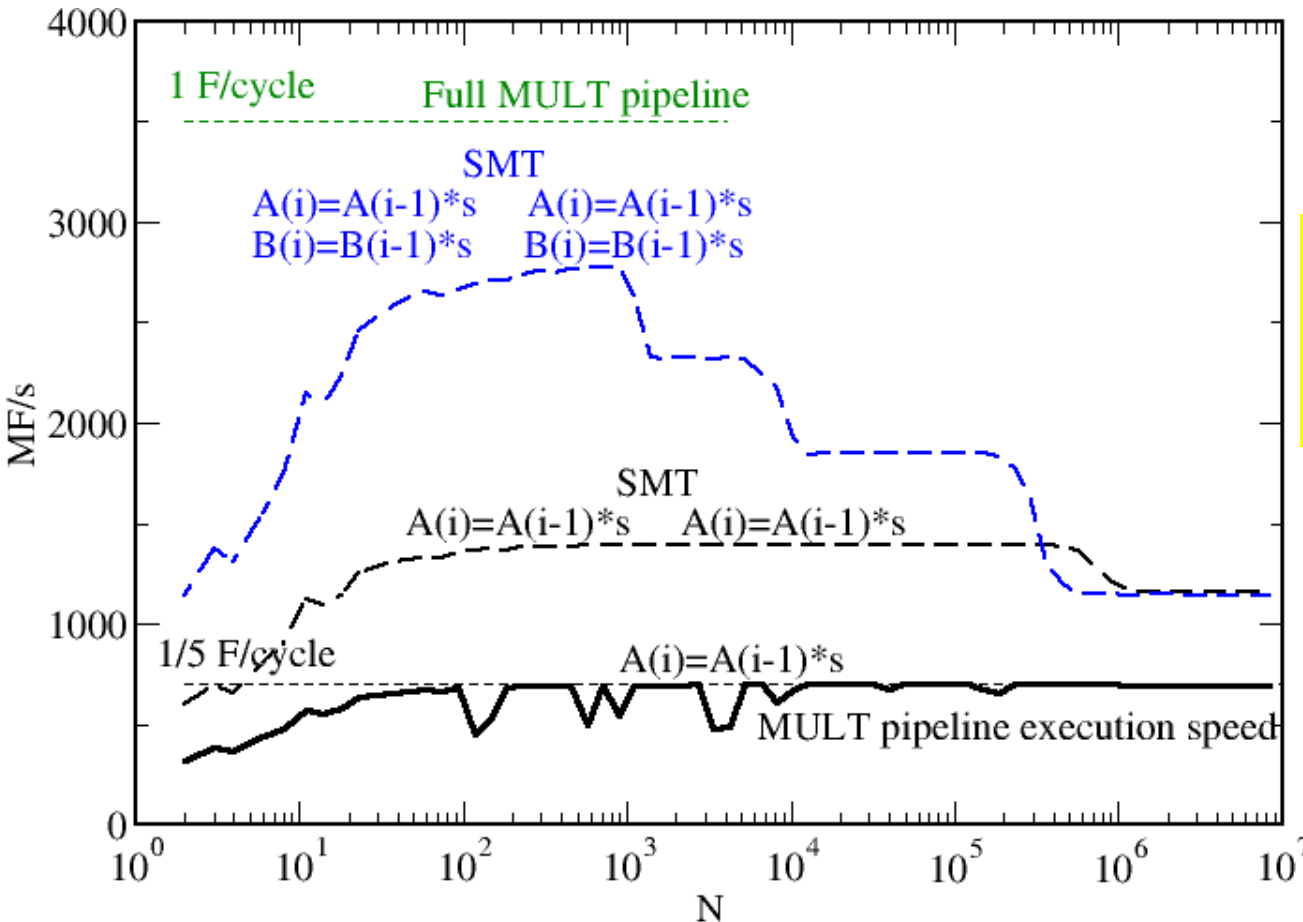
```
do i=1,N
  a(i) = a(i-1)*c
  b(i) = func(i)*d
enddo
```

# Simultaneous recursive updates with SMT



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

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



Fill bubbles via:

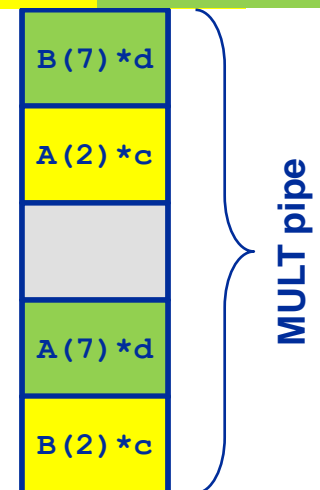
- SMT
- Multiple streams

**Thread 0:**

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

**Thread 1:**

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

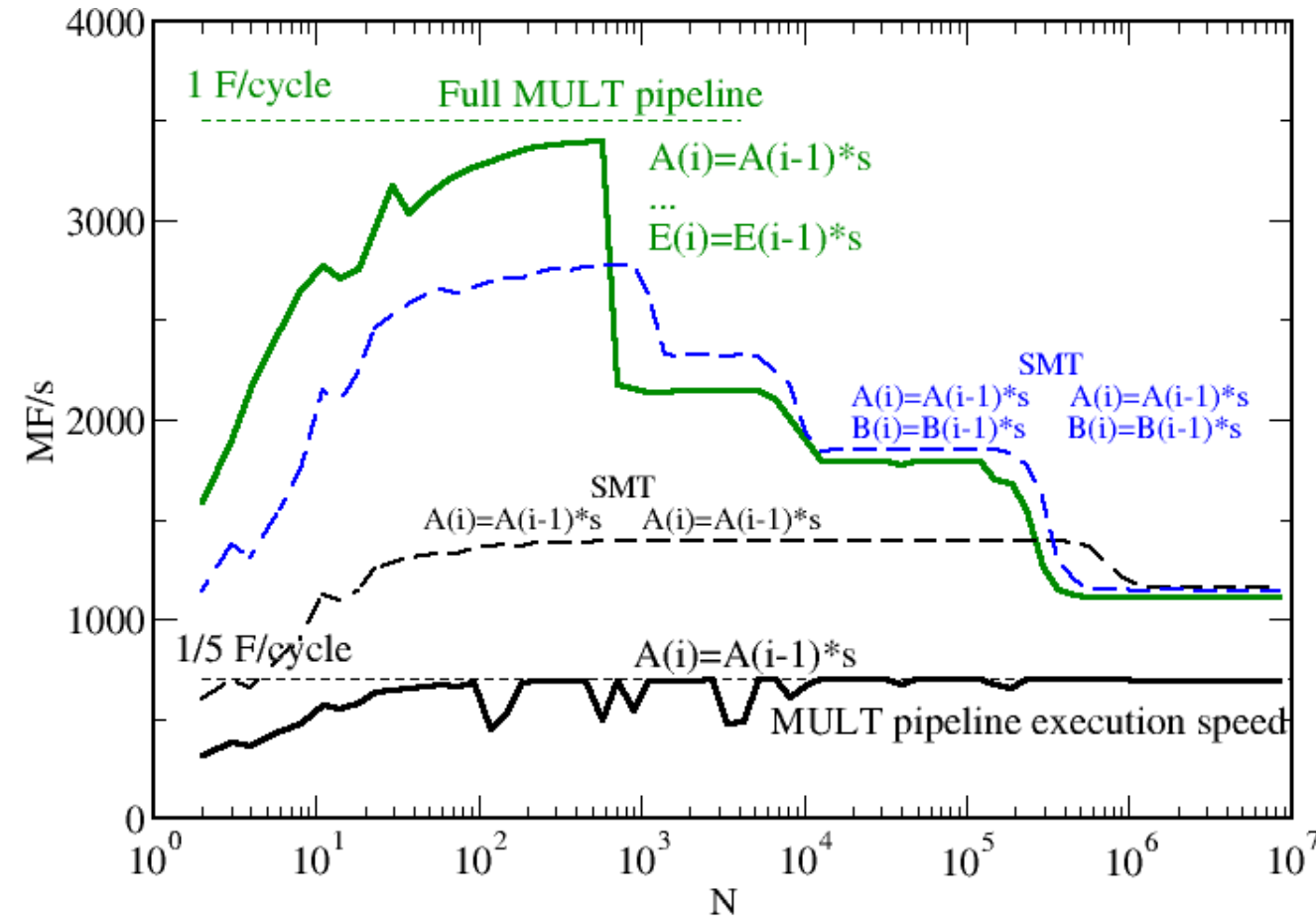


# Simultaneous recursive updates with SMT



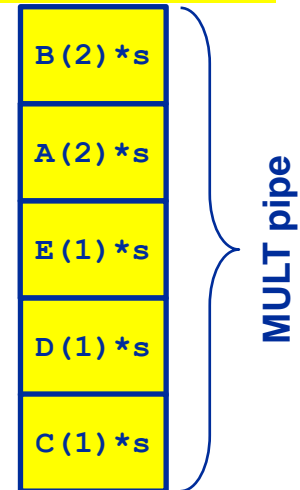
Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT

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



```

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



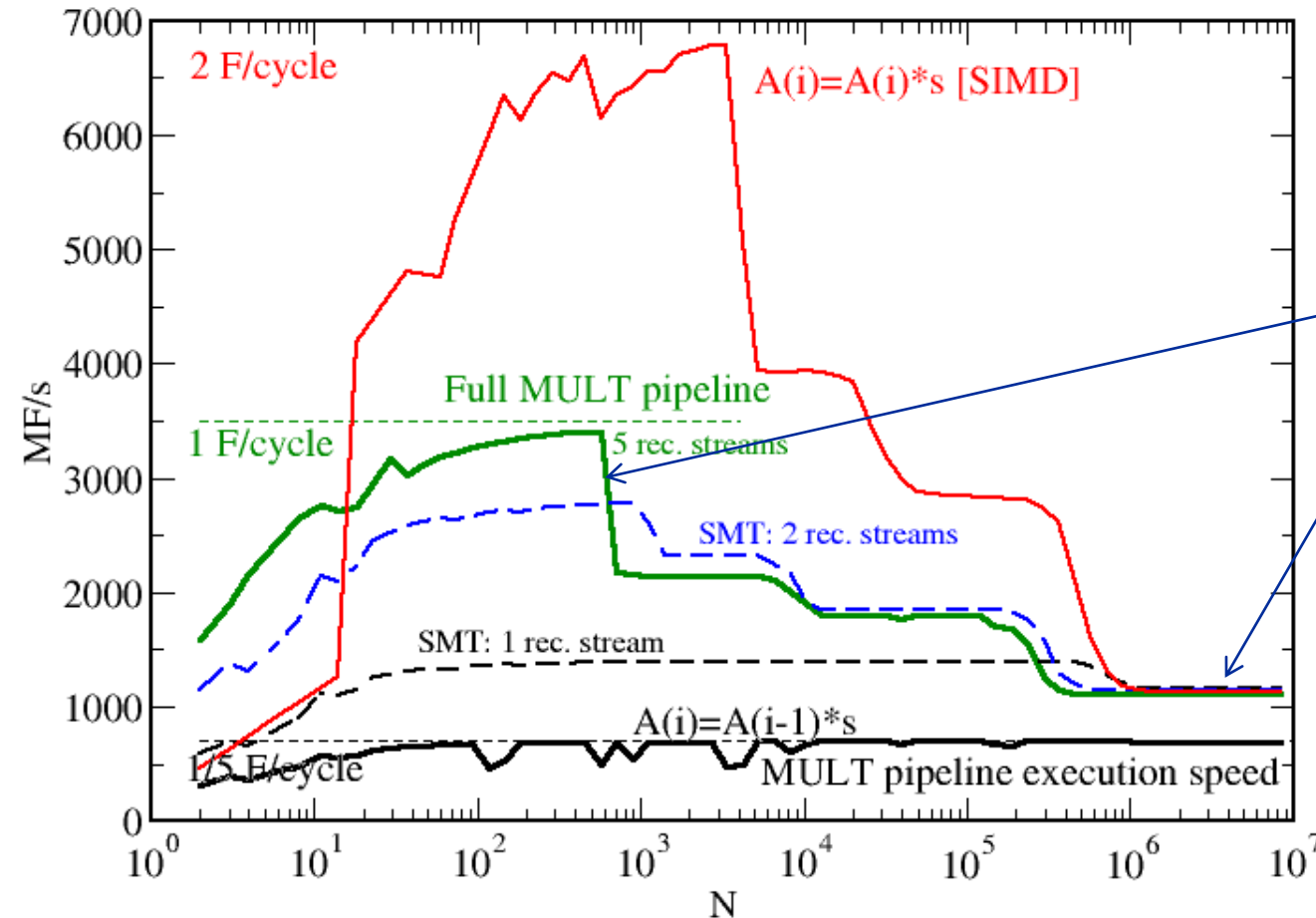
**5 independent updates on a single thread do the same job!**

# Simultaneous recursive updates with SMT



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

Pure update benchmark can be vectorized  $\rightarrow$  2 F / cycle (store limited)



## Recursive update:

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

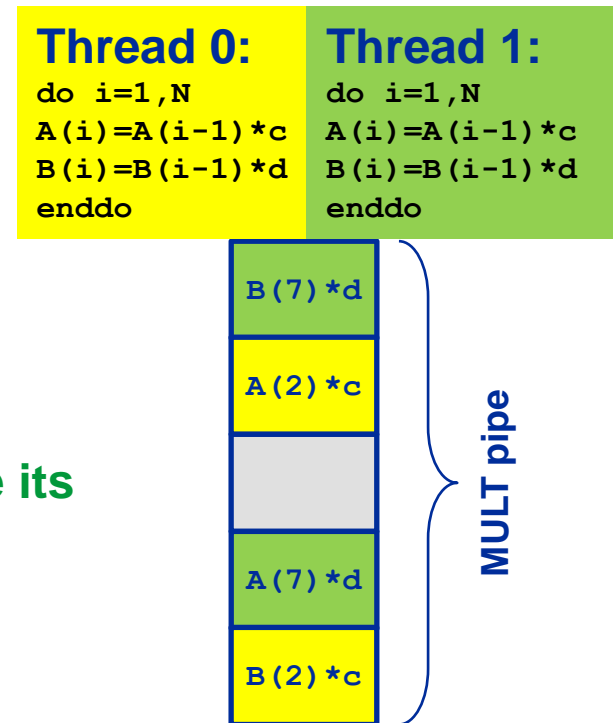
# SMT myths: Facts and fiction (1)



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

- Truth

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



# SMT myths: Facts and fiction (2)



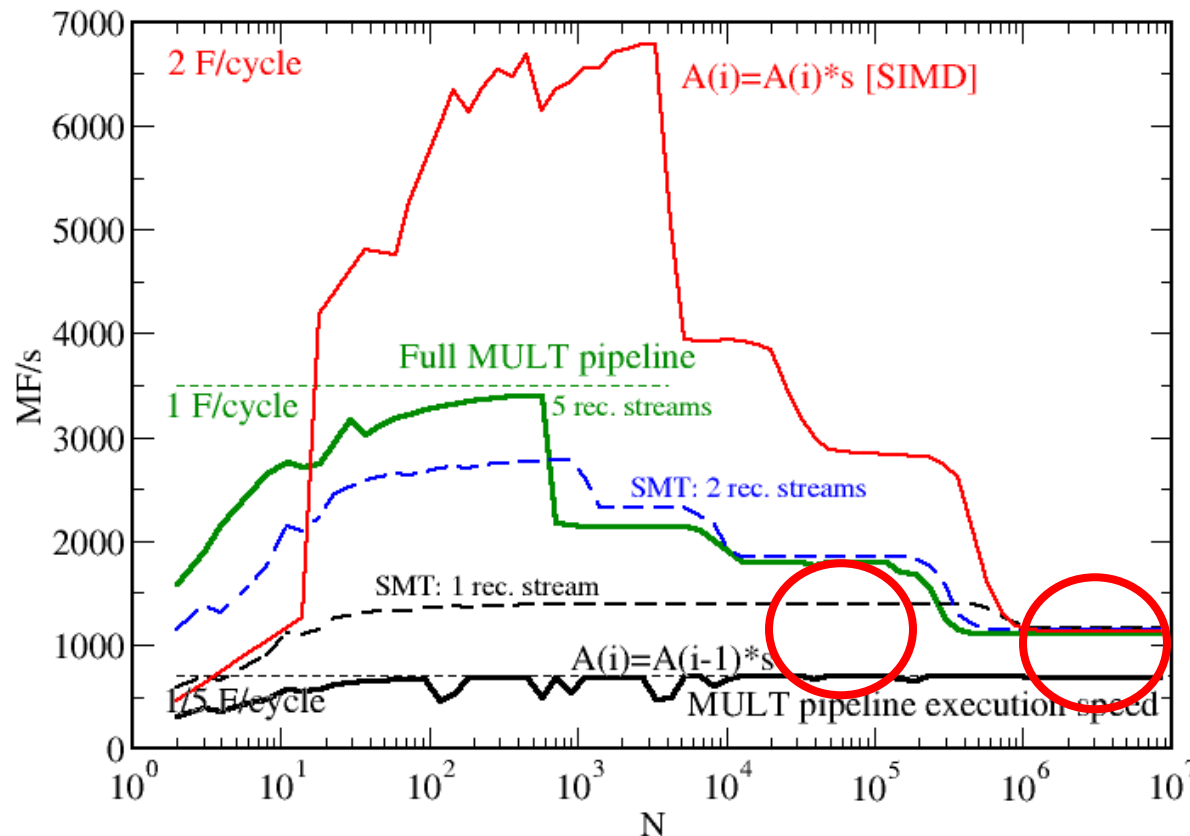
- **Myth:** “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

- **Truth:**

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!”**

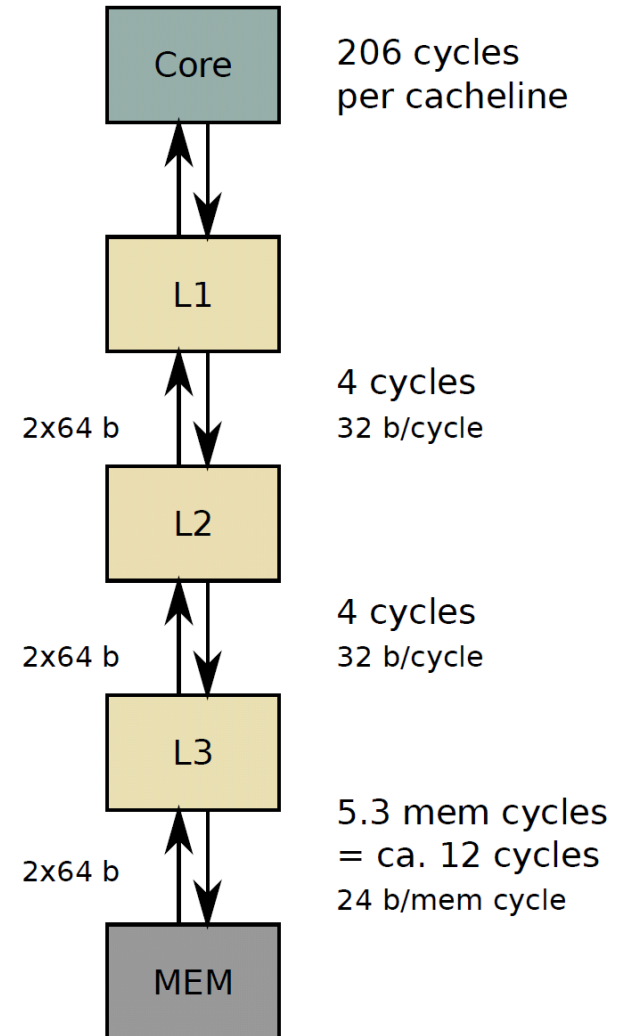


# SMT myths: Facts and fiction (3)











- **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.

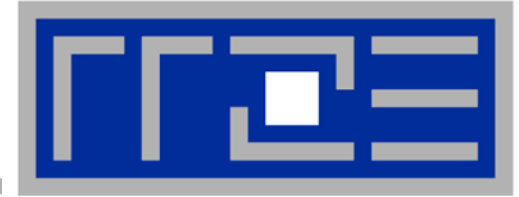
See also the “**ECM Performance Model**” later on.





|   |   |
|---|---|
| Functional parallelization                          |   |
| FP-only parallel loop code                          |   |
| Frequent thread synchronization                     |    |
| Code sensitive to cache size                        |    |
| Strongly memory-bound code                          |   |
| Independent pipeline-unfriendly instruction streams |    |





## **Beyond the chip boundary: Efficient parallel programming on ccNUMA nodes**

**Performance characteristics of ccNUMA nodes**

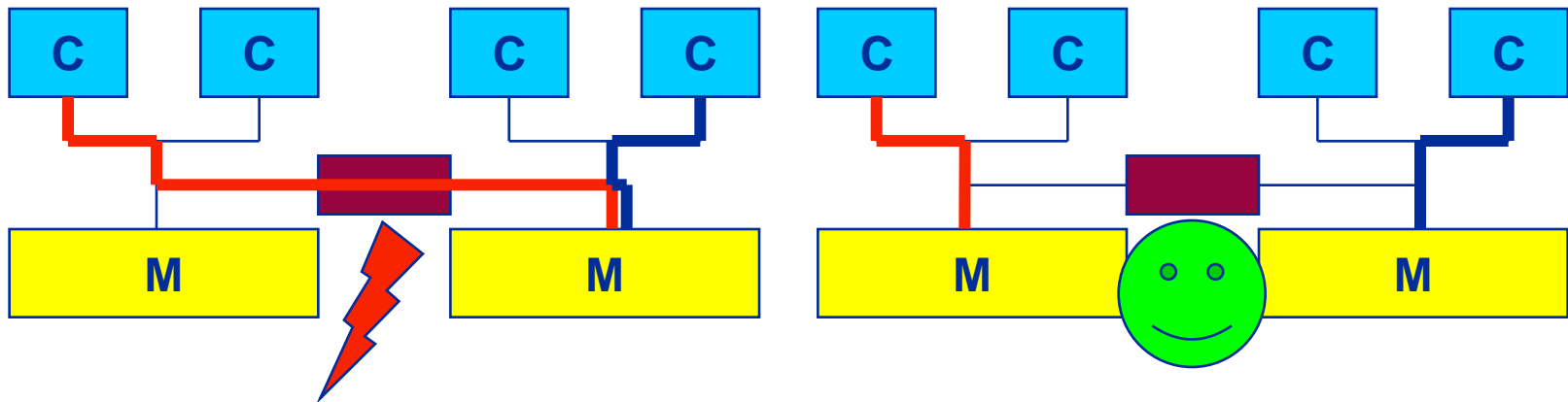
**First touch placement policy**

**ccNUMA locality and erratic access**



## ■ ccNUMA:

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



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

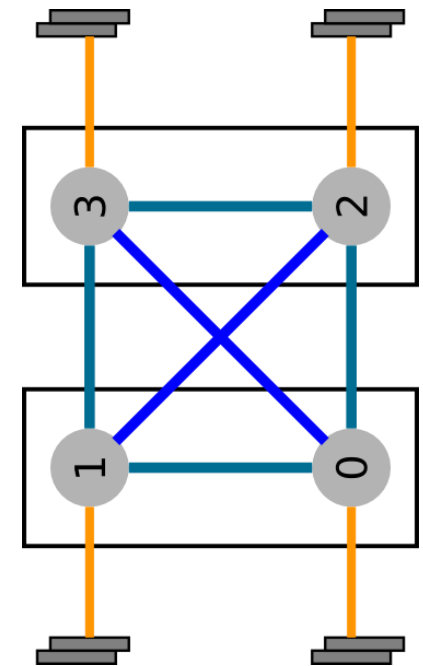
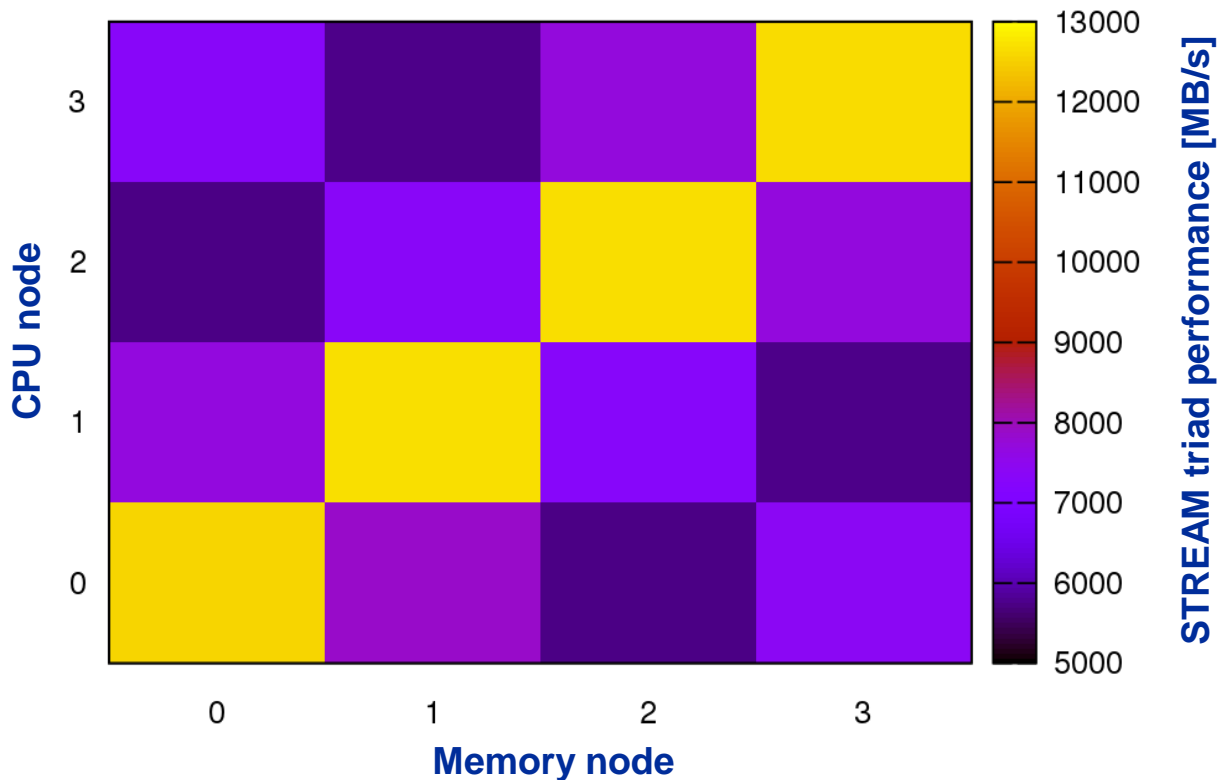
# Cray XE6 Interlagos node

4 chips, two sockets, 8 threads per ccNUMA domain



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

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





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

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

- **Examples:**

```
env OMP_NUM_THREADS=2 numactl --membind=0 --cpunodebind=1 ./stream
```

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

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



- "Golden Rule" of ccNUMA:

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

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

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

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

Memory not mapped here yet

Mapping takes place here

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



- Most simple case: explicit initialization

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

```
A=0.d0
```



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

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

```
!$OMP parallel  
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
    A(i)=0.d0
```

```
end do
```

```
!$OMP end do
```

```
...
```

```
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
    B(i) = function ( A(i) )
```

```
end do
```

```
!$OMP end do
```

```
!$OMP end parallel
```





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

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

```
READ(1000) A
```



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

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

```
!$OMP parallel
!$OMP do schedule(static)
```

```
do i = 1, N
    A(i)=0.d0
```

```
end do
```

```
!$OMP end do
```

```
!$OMP single
```

```
READ(1000) A
```

```
!$OMP end single
```

```
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
    B(i) = function ( A(i) )
```

```
end do
```

```
!$OMP end do
```

```
!$OMP end parallel
```





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





- 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...
- Try running with `numactl --interleave ...`
  - If performance goes up → ccNUMA problem!
- Consider using performance counters
  - `LIKWID-perfctr` can be used to measure nonlocal memory accesses
  - Example for Intel Nehalem (Core i7):

```
env OMP_NUM_THREADS=8 likwid-perfctr -g MEM -C N:0-7 ./a.out
```

# Using performance counters for diagnosing bad ccNUMA access locality



## Intel Nehalem EP node:

Uncore events only counted once per socket

| Event                         | core 0      | core 1      | core 2      | core 3      | core 4      | core 5      |
|-------------------------------|-------------|-------------|-------------|-------------|-------------|-------------|
| INSTR_RETIRED_ANY             | 5.20725e+08 | 5.24793e+08 | 5.21547e+08 | 5.23717e+08 | 5.28269e+08 | 5.29083e+08 |
| CPU_CLK_UNHALTED_CORE         | 1.90447e+09 | 1.90599e+09 | 1.90619e+09 | 1.90673e+09 | 1.90583e+09 | 1.90746e+09 |
| UNC_QMC_NORMAL_READS_ANY      | 8.17606e+07 | 0           | 0           | 0           | 8.07797e+07 | 0           |
| UNC_QMC_WRITES_FULL_ANY       | 5.53837e+07 | 0           | 0           | 0           | 5.51052e+07 | 0           |
| UNC_QHL_REQUESTS_REMOTE_READS | 6.84504e+07 | 0           | 0           | 0           | 6.8107e+07  | 0           |
| UNC_QHL_REQUESTS_LOCAL_READS  | 6.82751e+07 | 0           | 0           | 0           | 6.76274e+07 | 0           |

RDTSC timing: 0.827196 s

| Metric                      | core 0   | core 1   | core 2  | core 3   | core 4   | core 5   | core 6  | core 7  |
|-----------------------------|----------|----------|---------|----------|----------|----------|---------|---------|
| Runtime [s]                 | 0.714167 | 0.714733 | 0.71481 | 0.715013 | 0.714673 | 0.715286 | 0.71486 | 0.71515 |
| CPI                         | 3.65735  | 3.63188  | 3.65488 | 3.64076  | 3.60768  | 3.60521  | 3.59613 | 3.60184 |
| Memory bandwidth [MBytes/s] | 10610.8  | 0        | 0       | 0        | 10513.4  | 0        | 0       | 0       |
| Remote Read BW [MBytes/s]   | 5296     | 0        | 0       | 0        | 5269.43  | 0        | 0       | 0       |

Half of read BW comes from other socket!



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

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

- Or you have to use tasking/dynamic scheduling:

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

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



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

1. Explicit placement:

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

Observe page alignment of array to get proper placement!

2. Using global control via `numactl`:

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

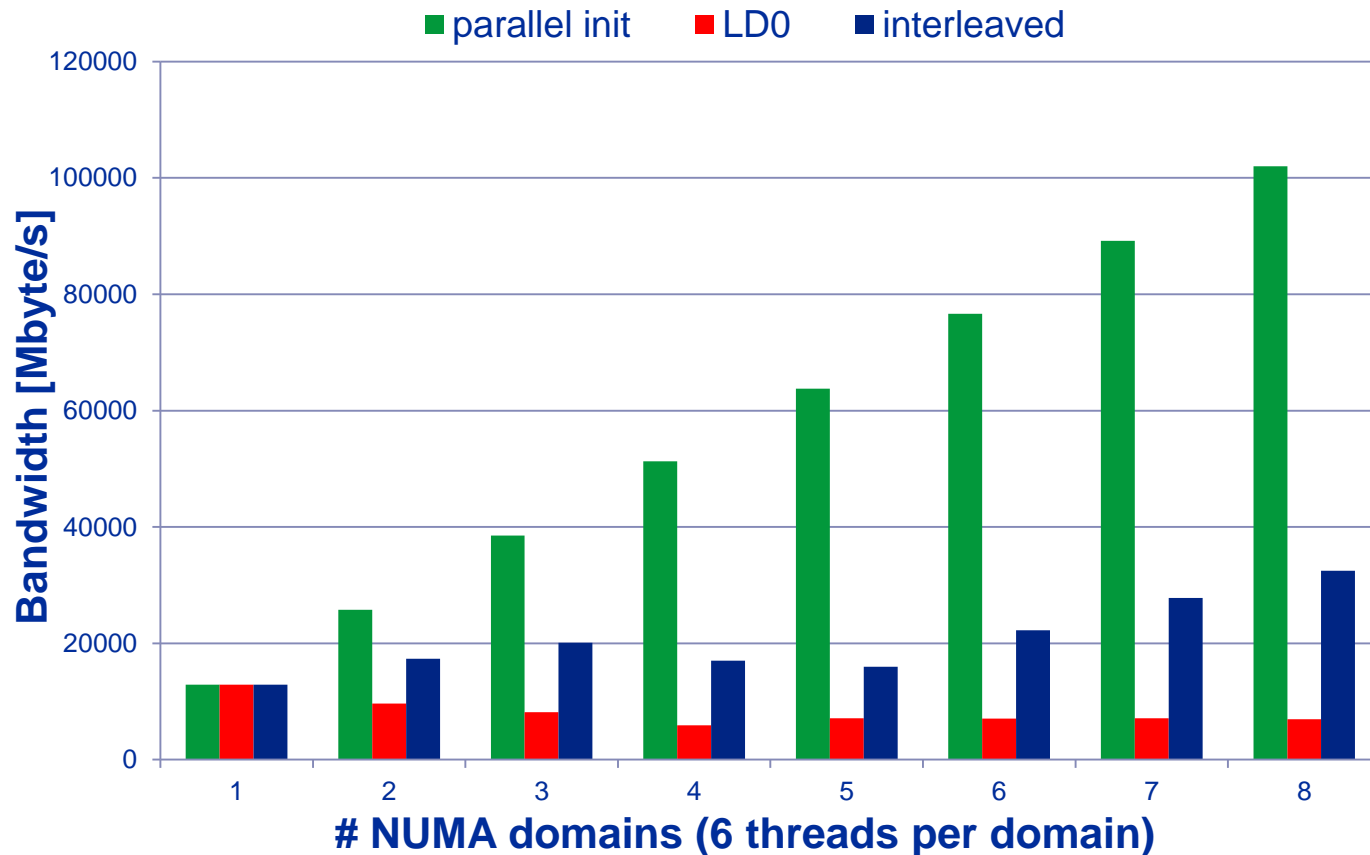
This is for **all** memory, not just the problematic arrays!

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

# The curse and blessing of interleaved placement: OpenMP STREAM 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>`

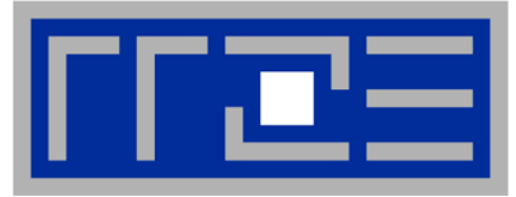




- ccNUMA is present on **all standard cluster architectures**
- With **pure MPI (and proper affinity control)** you **should be fine**
  - However, watch out for buffer cache
- With **threading**, you may be fine with **one process per ccNUMA domain**
- **Thread groups spanning more than one domain may cause problems**
  - Employ **first touch** placement (“**Golden Rule**”)
  - Experiment with round-robin placement
- **If access patterns are totally erratic, round-robin may be your only choice**
  - But there are advanced solutions (“locality queues”)



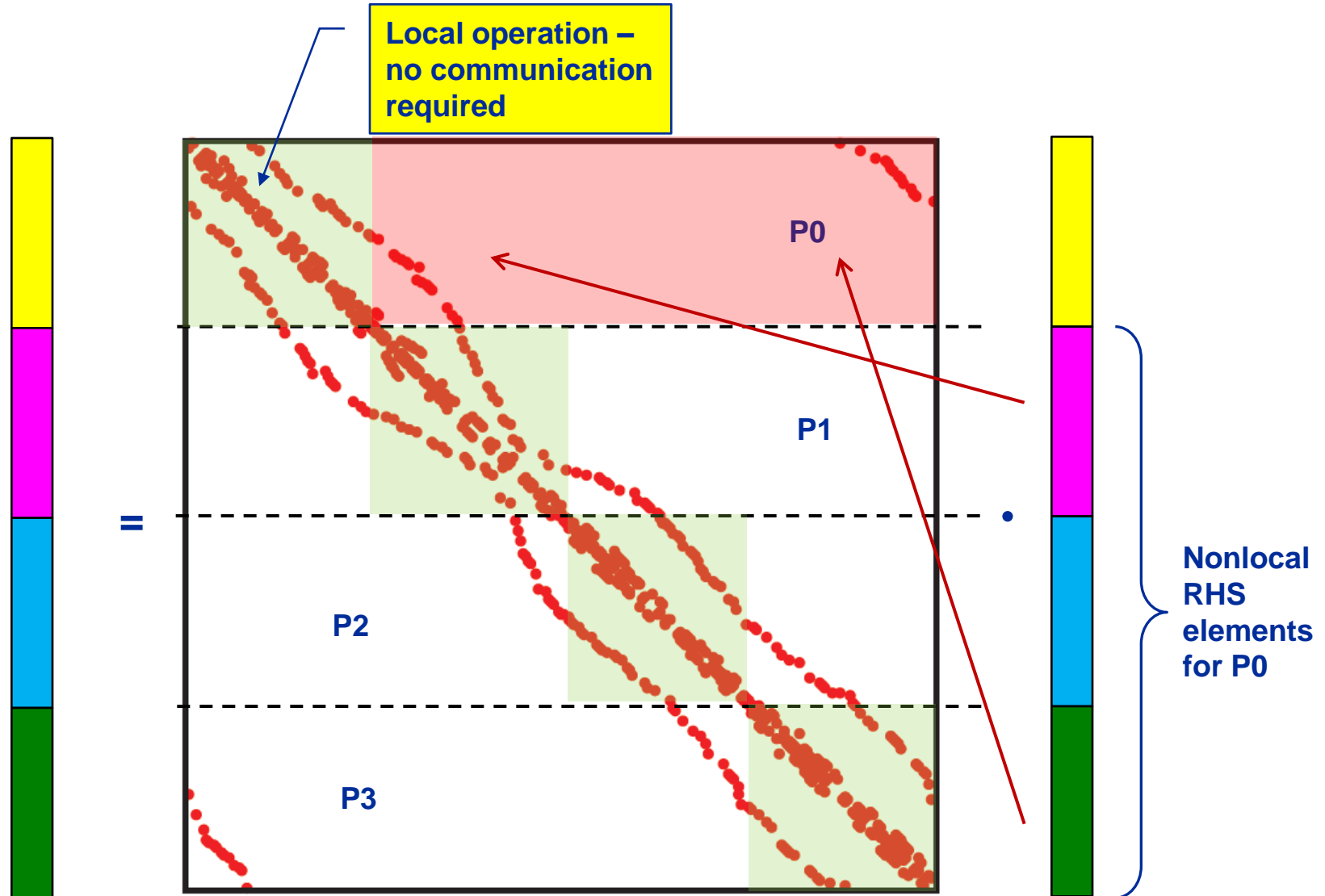
- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



## **Case study: Asynchronous MPI communication in sparse MVM**

**What to do with spare cores**

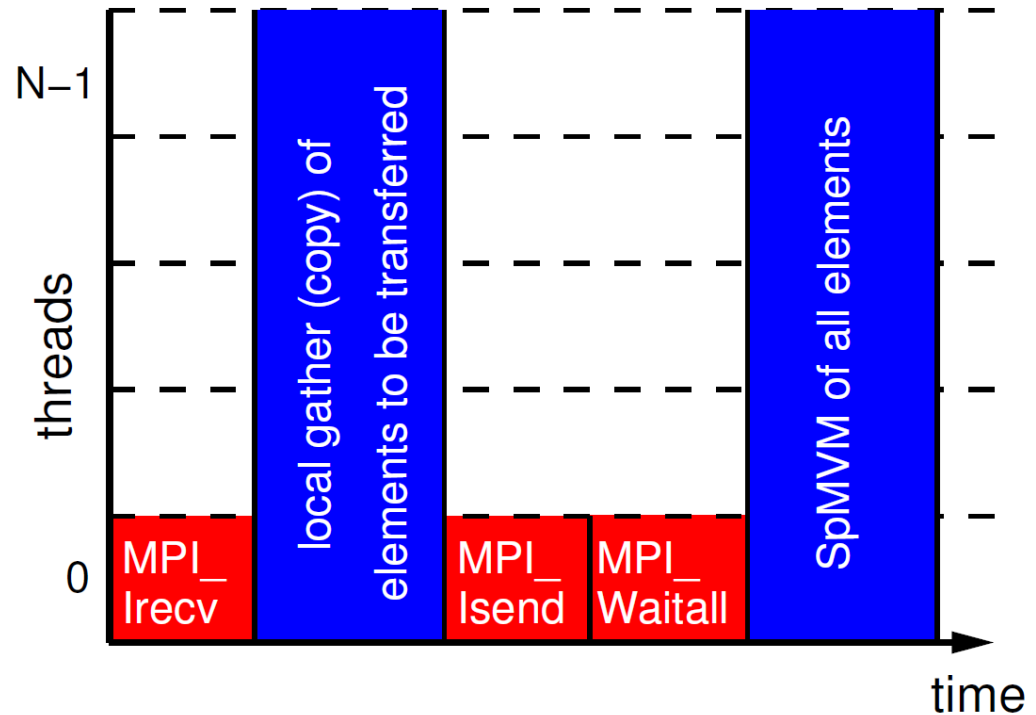






- **Variant 1: “Vector mode” without overlap**

- **Standard concept for “hybrid MPI+OpenMP”**
- **Multithreaded computation (all threads)**
- **Communication only outside of computation**

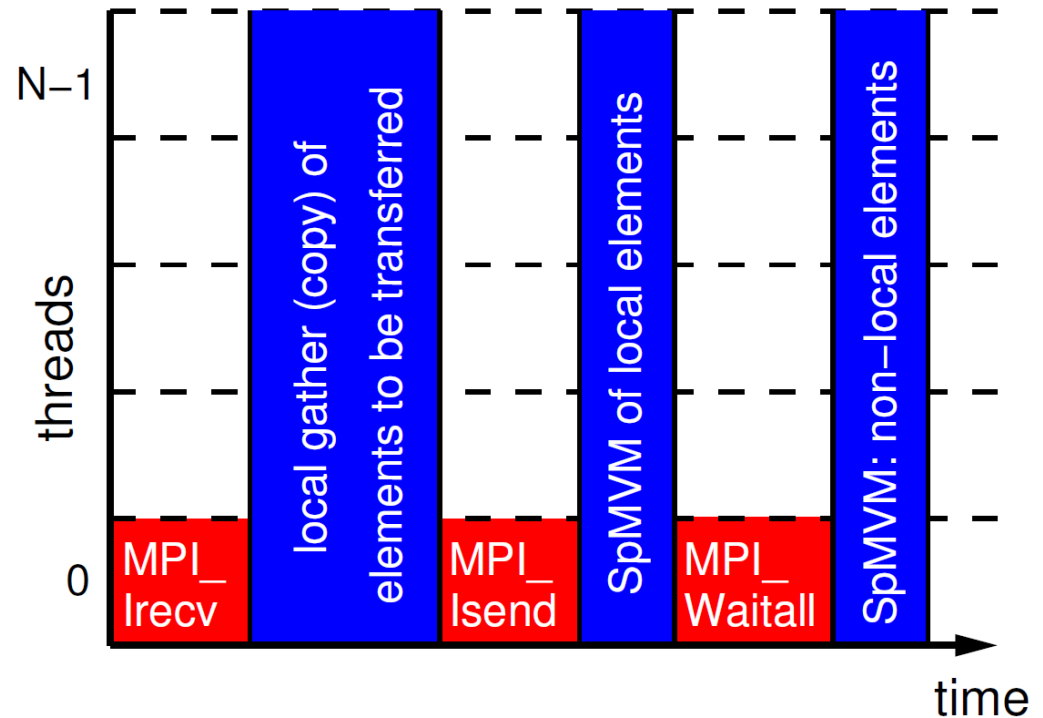


- **Benefit of threaded MPI process only due to message aggregation and (probably) better load balancing**

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](#)

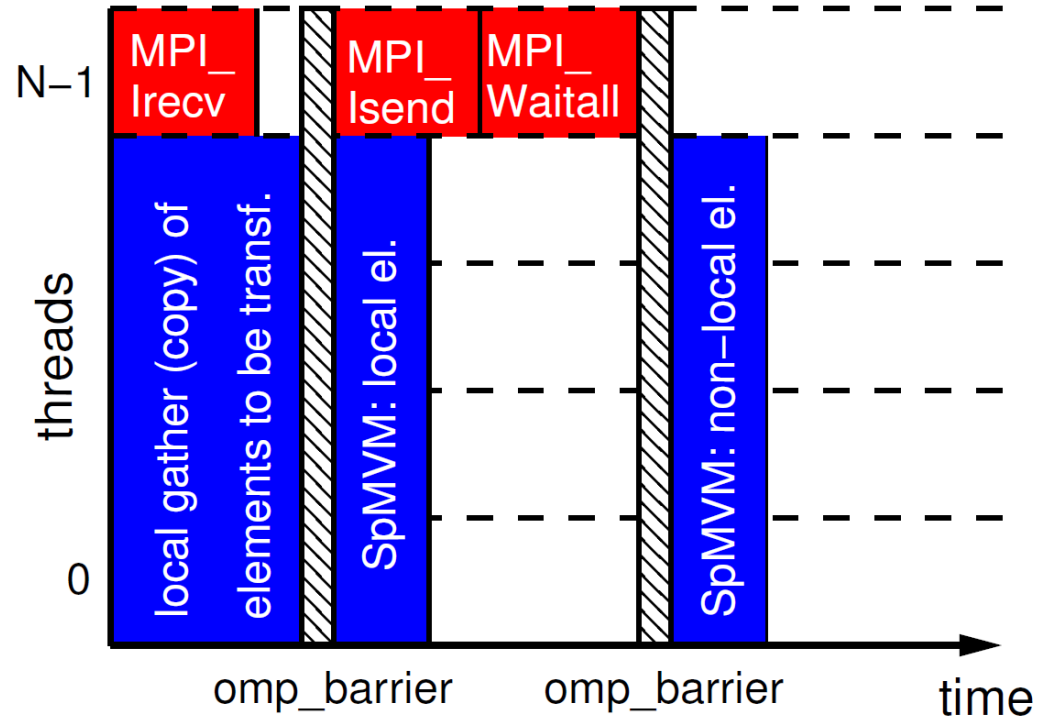


- **Variant 2: “Vector mode” with naïve overlap** (“good faith hybrid”)
- Relies on MPI to support async nonblocking PtP
- Multithreaded computation (all threads)
- Still simple programming
- **Drawback: Result vector is written twice to memory**
  - modified performance model



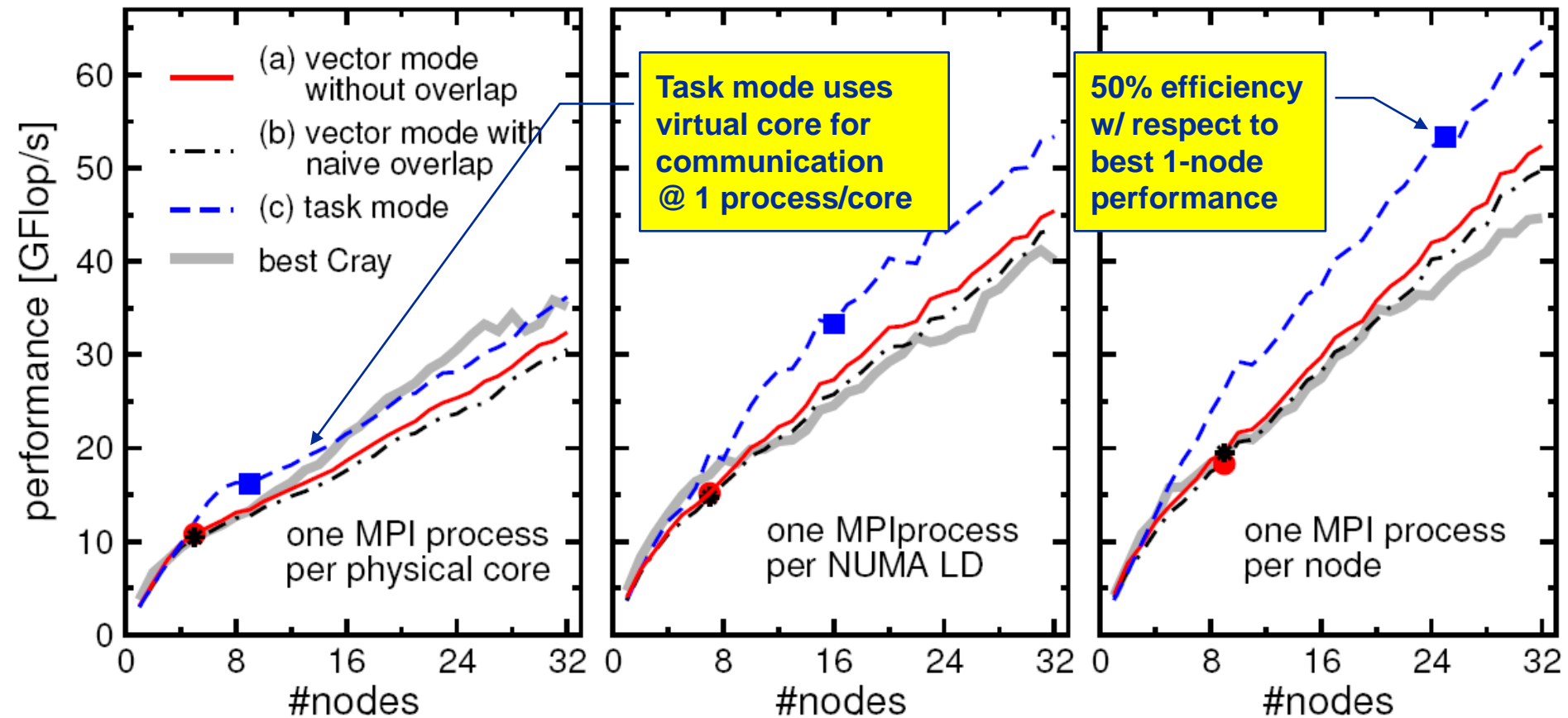


- **Variant 3: “Task mode” with dedicated communication thread**
- **Explicit overlap, more complex to implement**
- **One thread missing in team of compute threads**
  - But that doesn’t hurt here...
  - Using tasking seems simpler but may require some work on NUMA locality
- **Drawbacks**
  - Result vector is written twice to memory
  - No simple OpenMP worksharing (manual, tasking)



R. Rabenseifner and G. Wellein: *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures*. International Journal of High Performance Computing Applications **17**, 49-62, February 2003.  
[DOI:10.1177/1094342003017001005](https://doi.org/10.1177/1094342003017001005)

# Performance results for the HMeP matrix



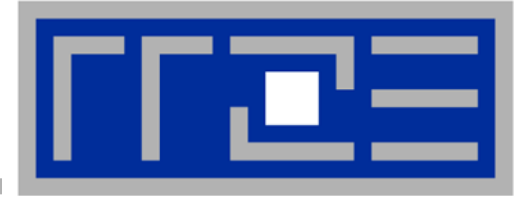
- **Dominated by communication (and some load imbalance for large #procs)**
- **Single-node Cray performance cannot be maintained beyond a few nodes**
- **Task mode pays off esp. with one process (12 threads) per node**
- **Task mode overlap (over-)compensates additional LHS traffic**



- Do not rely on asynchronous MPI progress
- Sparse MVM leaves resources (cores) free for use by **communication threads**
- Simple **“vector mode” hybrid MPI+OpenMP parallelization is not good enough** if communication is a real problem
- **“Task mode” hybrid can truly hide communication and overcompensate penalty** from additional memory traffic in spMVM
- Comm thread can share a core with comp thread via **SMT** and still be asynchronous
- **If pure MPI scales ok and maintains its node performance according to the node-level performance model, don't bother going hybrid**
- **Extension to multi-GPGPU is possible**
  - See references



- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



# A simple power model for the Sandy Bridge processor

Assumptions

Validation using simple benchmarks

G. Hager, J. Treibig, J. Habich and G. Wellein: Exploring performance and power properties of modern multicore chips via simple machine models. Submitted.  
Preprint: [arXiv:1208.2908](https://arxiv.org/abs/1208.2908)





- **Goal: Establish model for chip power and program energy consumption with respect to**
  - Clock speed
  - Number of cores used
  - Single-thread program performance
- **Choose different characteristic benchmark applications to measure a chip's power behavior**
  - **Matrix-matrix-multiply** (“DGEMM”): “Hot” code, well scalable
  - **Ray tracer**: Sensitive to SMT execution (15% speedup), well scalable
  - **2D Jacobi solver**: 4000x4000 grid, strong saturation on the chip
    - AVX variant
    - Scalar variant
- **Measure characteristics of those apps and establish a power model**



## Assumptions:

1. Power is a quadratic polynomial in the clock frequency
2. Dynamic power is linear in the number of active cores  $t$
3. Performance is linear in the number of cores until it hits a bottleneck ( $\leftarrow$  ECM model)
4. Performance is linear in the clock frequency unless it hits a bottleneck
5. **Energy to solution** is power dissipation divided by performance

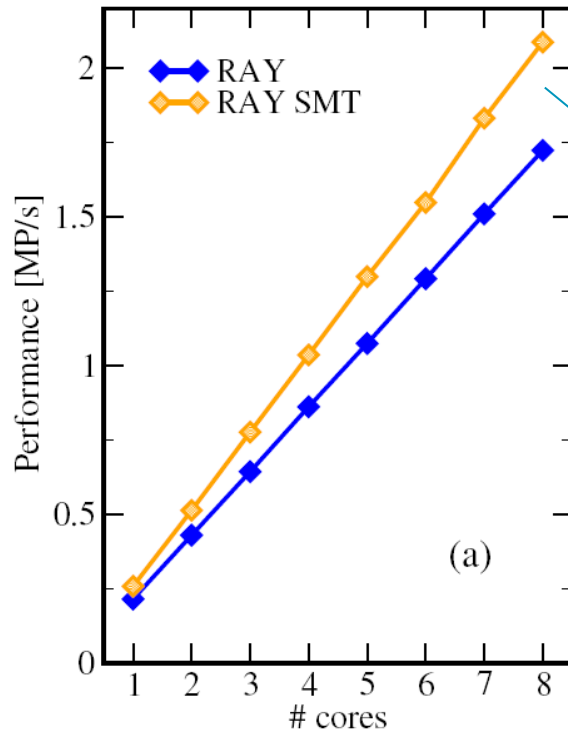
Model:

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{\min((1 + \Delta v)t P_0, P_{\max})}$$

where  $f = (1 + \Delta v)f_0$

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{\min((1 + \Delta v)tP_0, P_{\max})}$$

1. If there is **no saturation**, use **all available cores** to minimize  $E$



Minimum E here

$$\frac{\partial E}{\partial t} = -\frac{W_0}{(1 + \Delta v)t^2 P_0} < 0$$



$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{\min((1 + \Delta v)t P_0, P_{\max})}$$

2. There is an optimal frequency  $f_{\text{opt}}$  at which  $E$  is minimal in the **non-saturated** case, with

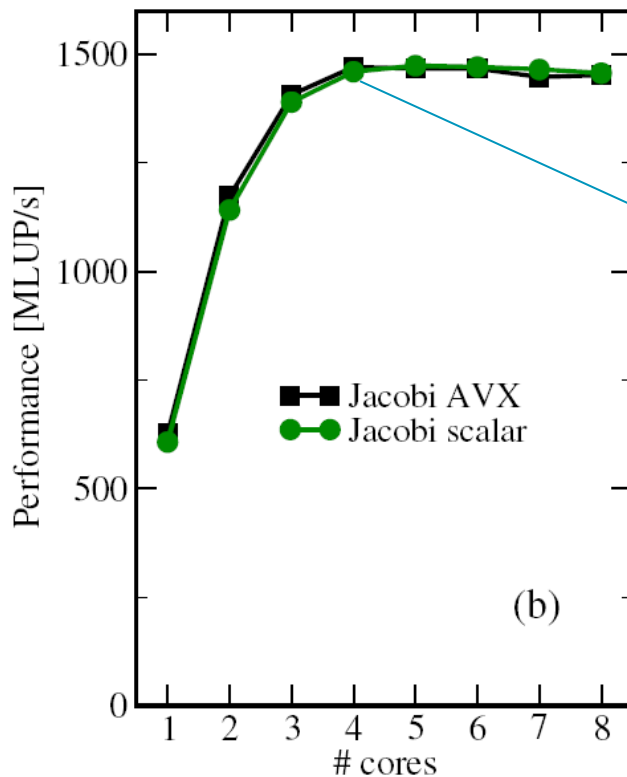
$$f_{\text{opt}} = \sqrt{\frac{W_0}{W_2 t}}, \text{ hence it depends on the baseline power}$$

- “**Clock race to idle**” if baseline accommodates whole system!
- May have to look at other metrics, e.g.,  $C = E/P$

$$\frac{\partial C}{\partial \Delta v} = -\frac{2W_0 + W_1 f t}{(f/f_0)^3 P_0^2} < 0$$

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{\min((1 + \Delta v)tP_0, P_{\max})}$$

3. If there is saturation, ***E*** is minimal at the saturation point

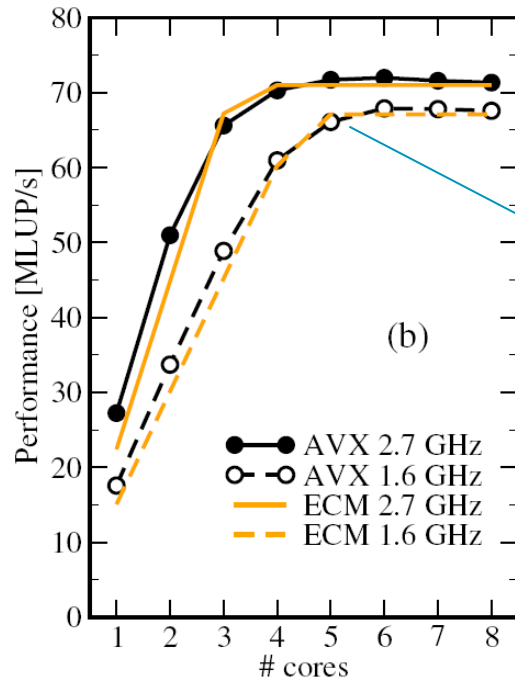


Minimum *E* here

$$t_s = \frac{P_{\max}}{(1 + \Delta v)P_0}$$

$$E = \frac{W_0 + (W_1 f + W_2 f^2) t}{\min((1 + \Delta v) t P_0, P_{\max})}$$

4. If there is saturation, **absolute minimum  $E$**  is reached if the **saturation point is at the number of available cores**

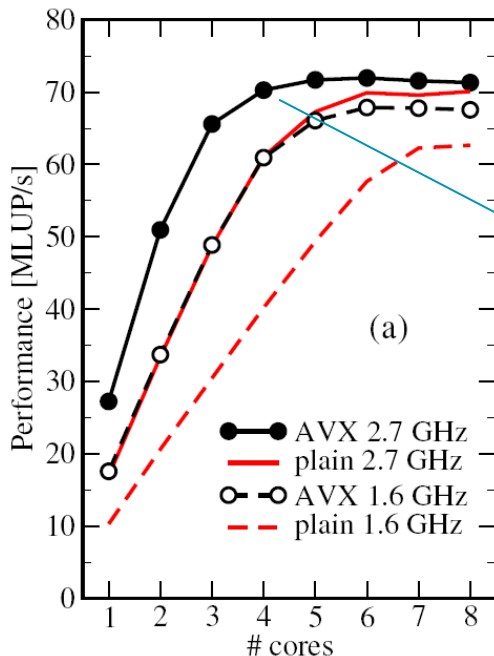


Slower clock  
 → more cores to saturation  
 → smaller  $E$

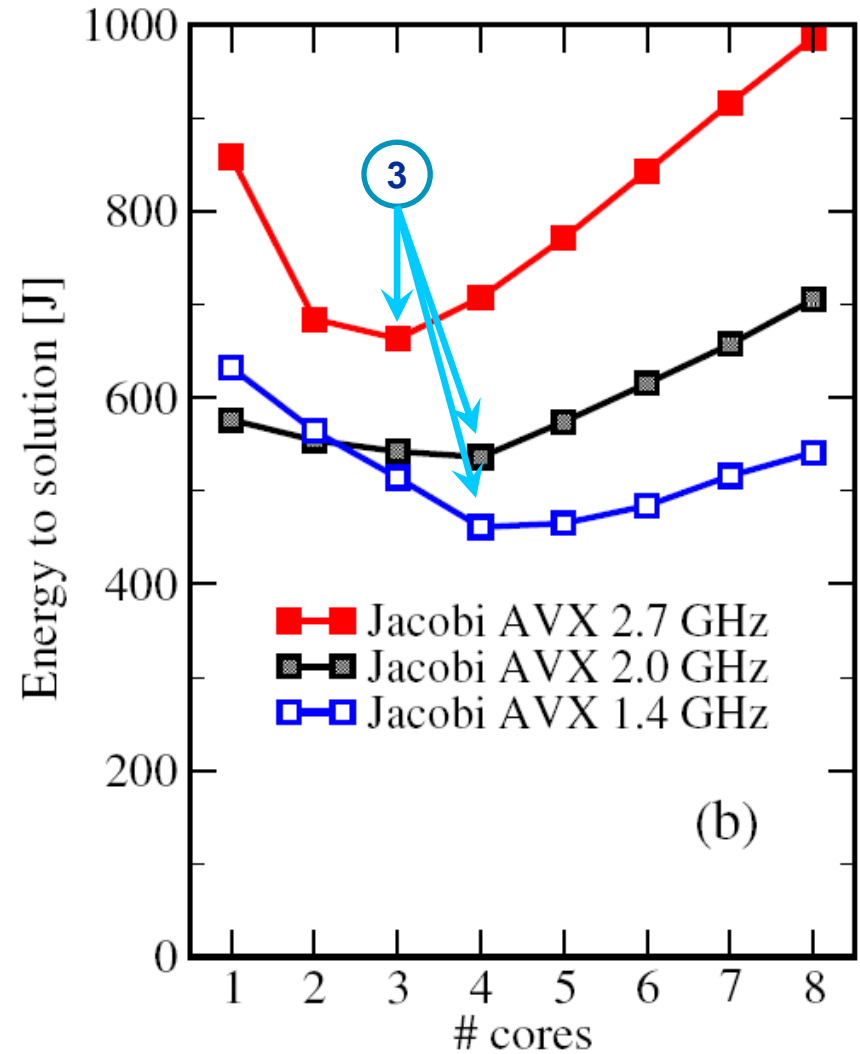
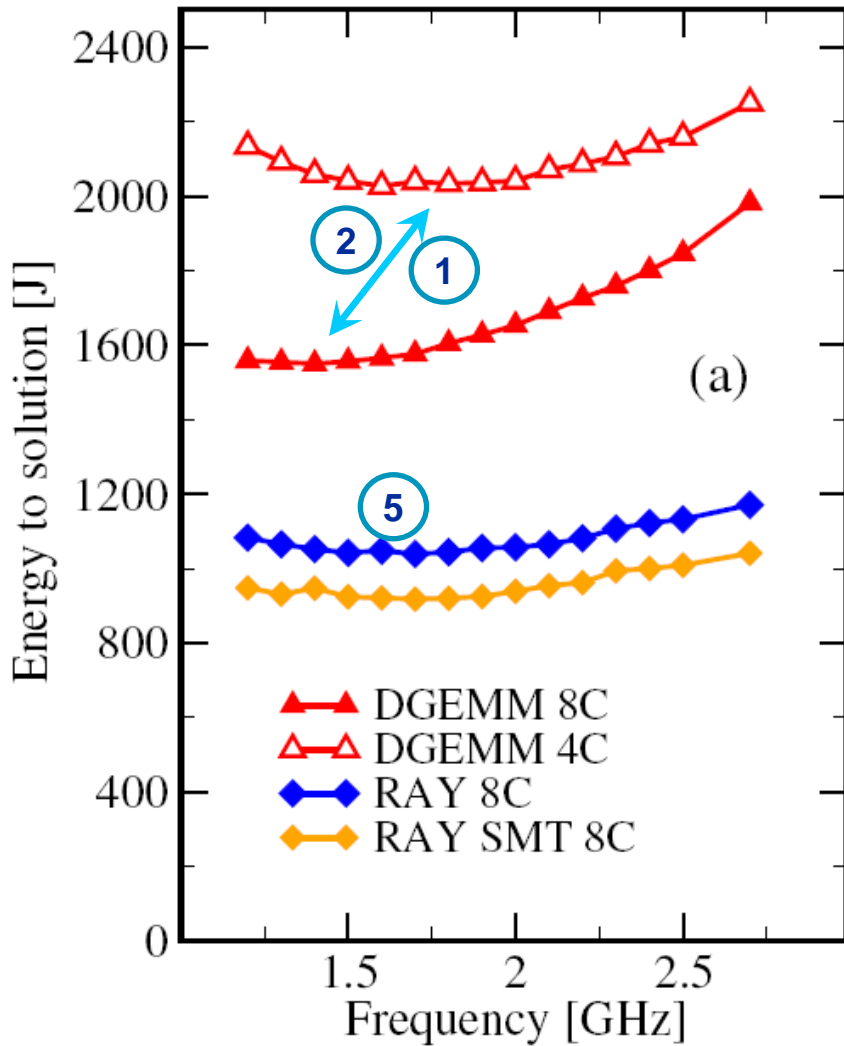
$$E = \frac{W_0 + (W_1 f + W_2 f^2) t}{\min((1 + \Delta v) t P_0, P_{\max})}$$

## 5. Making code execute faster on the core saves energy since

- The time to solution is smaller if the code scales (“Code race to idle”)
- We can use fewer cores to reach saturation if there is a bottleneck



Better code  
→ earlier saturation  
→ smaller E @ saturation







- **Simple assumptions lead to surprising conclusions**
- **Performance saturation** plays a key role
- **“Clock race to idle”** can be proven quantitatively
- **“Code race to idle”** (optimization saves energy) is a trivial result
  - Better: **“Optimization makes better use of the energy budget”**
- **Possible extensions to the power model**
  - Allow for per-core frequency setting (coming with Intel Haswell)
  - Accommodate load imbalance & sync overhead



- **Motivation**
- **Performance Engineering**
  - Performance modeling
  - The Performance Engineering process
- **Modern architectures**
  - Multicore
  - Accelerators
  - Programming models
- **Data access**
- **Performance properties of multicore systems**
  - Saturation
  - Scalability
  - Synchronization
- **Case study: OpenMP-parallel sparse MVM**
- **Basic performance modeling: Roofline**
  - Theory
  - Case study: 3D Jacobi solver and guided optimizations
  - Modeling erratic access
- **Some more architecture**
  - Simultaneous multithreading (SMT)
  - ccNUMA
- **Putting cores to good use**
  - Asynchronous communication in spMVM
- **A simple power model for multicore**
  - Power-efficient code execution
- **Conclusions**



- **LIKWID: Lightweight multicore performance tools**
  - <http://code.google.com/p/likwid>
- **Multicore-specific properties of MPI communication**
- **Sparse MVM on multiple GPGPUs: Performance modeling for viability analysis**
  - See references
- **Exploiting shared caches for temporal blocking of stencil codes**
- **Execution-Cache-Memory (ECM) model**
  - Predictive model for multicore scaling
  - Goes well with the power model
- **... and much more ☹**



- **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?
  - Leave surplus cores idle to save energy
- **Simple modeling techniques help us**
  - ... understand the limits of our code on the given hardware
  - ... identify optimization opportunities and hence save energy
  - ... learn more, especially when they do not work!



## Code:

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

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

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

**Optimal  
performance?**

**Jan Treibig  
Johannes Habich  
Moritz Kreutzer  
Markus Wittmann  
Thomas Zeiser  
Michael Meier  
Faisal Shahzad  
Gerald Schubert**

**THANK YOU.**

**KONWIHR**  
**II**  
OMI4papps  
HQS@HPC II



**Bundesministerium  
für Bildung  
und Forschung**

**hpcADD  
SKALB**



- **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.
- **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.





## Book:

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

## Papers:

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





Papers continued:

- G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske: **Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization**. Proc. COMPSAC 2009.  
[DOI: 10.1109/COMPSAC.2009.82](https://doi.org/10.1109/COMPSAC.2009.82)
- M. Wittmann, G. Hager, J. Treibig and G. Wellein: **Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters**. Parallel Processing Letters **20** (4), 359-376 (2010).  
[DOI: 10.1142/S0129626410000296](https://doi.org/10.1142/S0129626410000296). Preprint: [arXiv:1006.3148](https://arxiv.org/abs/1006.3148)
- J. Treibig, G. Hager and G. Wellein: **LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments**. Proc. [PSTI2010](https://www.psti2010.org/), the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010.  
[DOI: 10.1109/ICPPW.2010.38](https://doi.org/10.1109/ICPPW.2010.38). Preprint: [arXiv:1004.4431](https://arxiv.org/abs/1004.4431)
- G. Schubert, H. Fehske, G. Hager, and G. Wellein: **Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems**. Parallel Processing Letters 21(3), 339-358 (2011).  
[DOI: 10.1142/S0129626411000254](https://doi.org/10.1142/S0129626411000254)
- J. Treibig, G. Wellein and G. Hager: **Efficient multicore-aware parallelization strategies for iterative stencil computations**. Journal of Computational Science 2 (2), 130-137 (2011).  
[DOI 10.1016/j.jocs.2011.01.010](https://doi.org/10.1016/j.jocs.2011.01.010)



Papers continued:

- K. Iglberger, G. Hager, J. Treibig, and U. Rde: [Expression Templates Revisited: A Performance Analysis of Current ET Methodologies](#). *SIAM Journal on Scientific Computing* **34**(2), C42-C69 (2012). [DOI: 10.1137/110830125](#), Preprint: [arXiv:1104.1729](#)
- K. Iglberger, G. Hager, J. Treibig, and U. Rde: [High Performance Smart Expression Template Math Libraries](#). 2nd International Workshop on New Algorithms and Programming Models for the Manycore Era ([APMM 2012](#)) at [HPCS 2012](#), July 2-6, 2012, Madrid, Spain. [DOI: 10.1109/HPCSim.2012.6266939](#)
- 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](#)