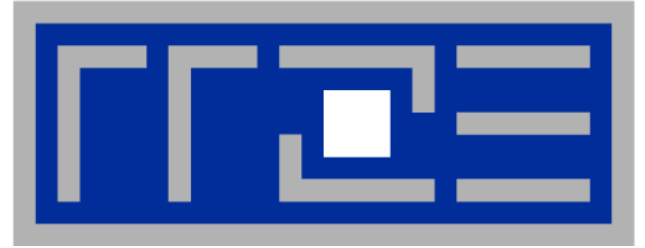


# **Node-Level Architecture and Performance Engineering**

**Georg Hager, Gerhard Wellein  
Erlangen Regional Computing Center  
University of Erlangen-Nuremberg**

**Guest Lecture  
University of Wuppertal  
February 3, 2014**



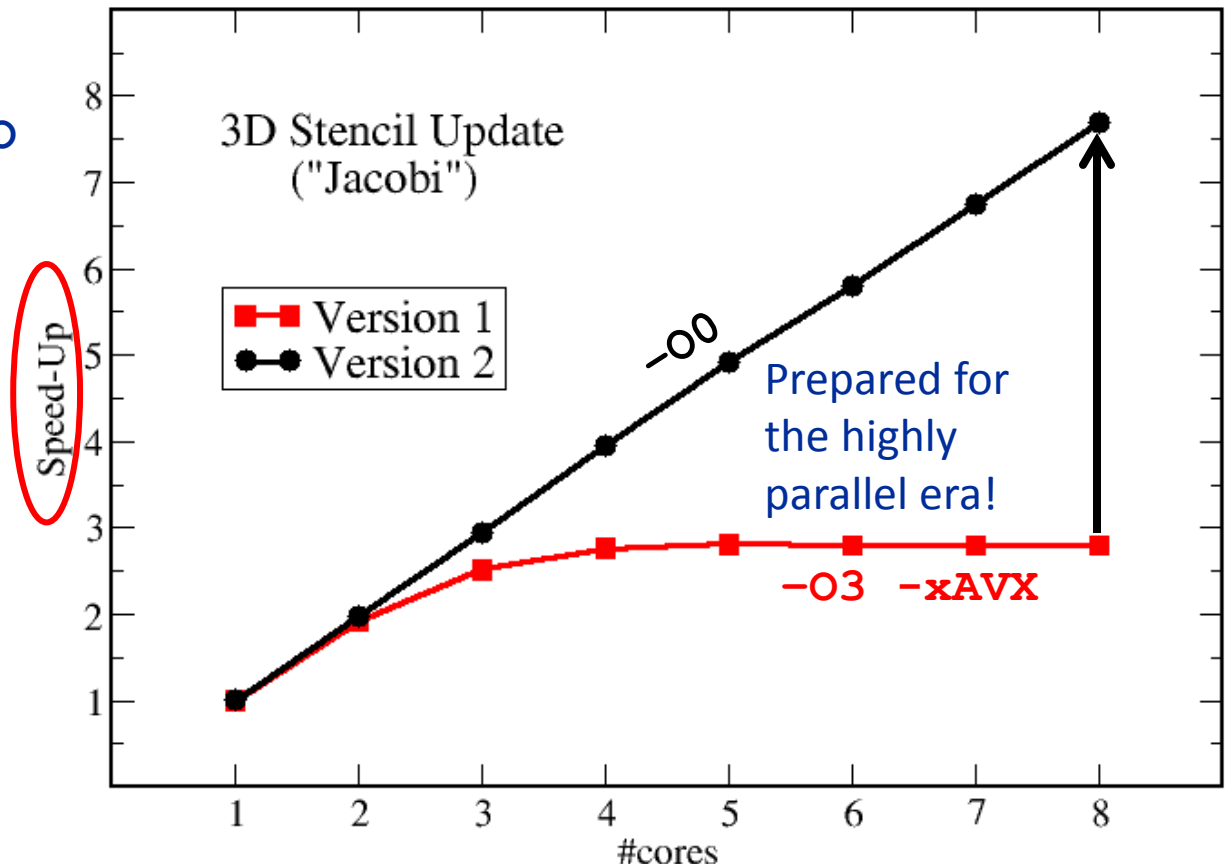
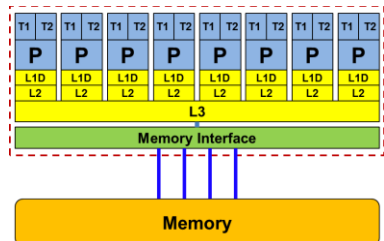
**Prelude:**  
**Scalability 4 the win!**

# 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 a compile option makes this code scalable on an 8-core chip



# Scalability Myth: Code scalability is the key issue



```
!$OMP PARALLEL DO
```

```
do k = 1 , Nk
```

```
do j = 1 , Nj; do i = 1 , Ni
```

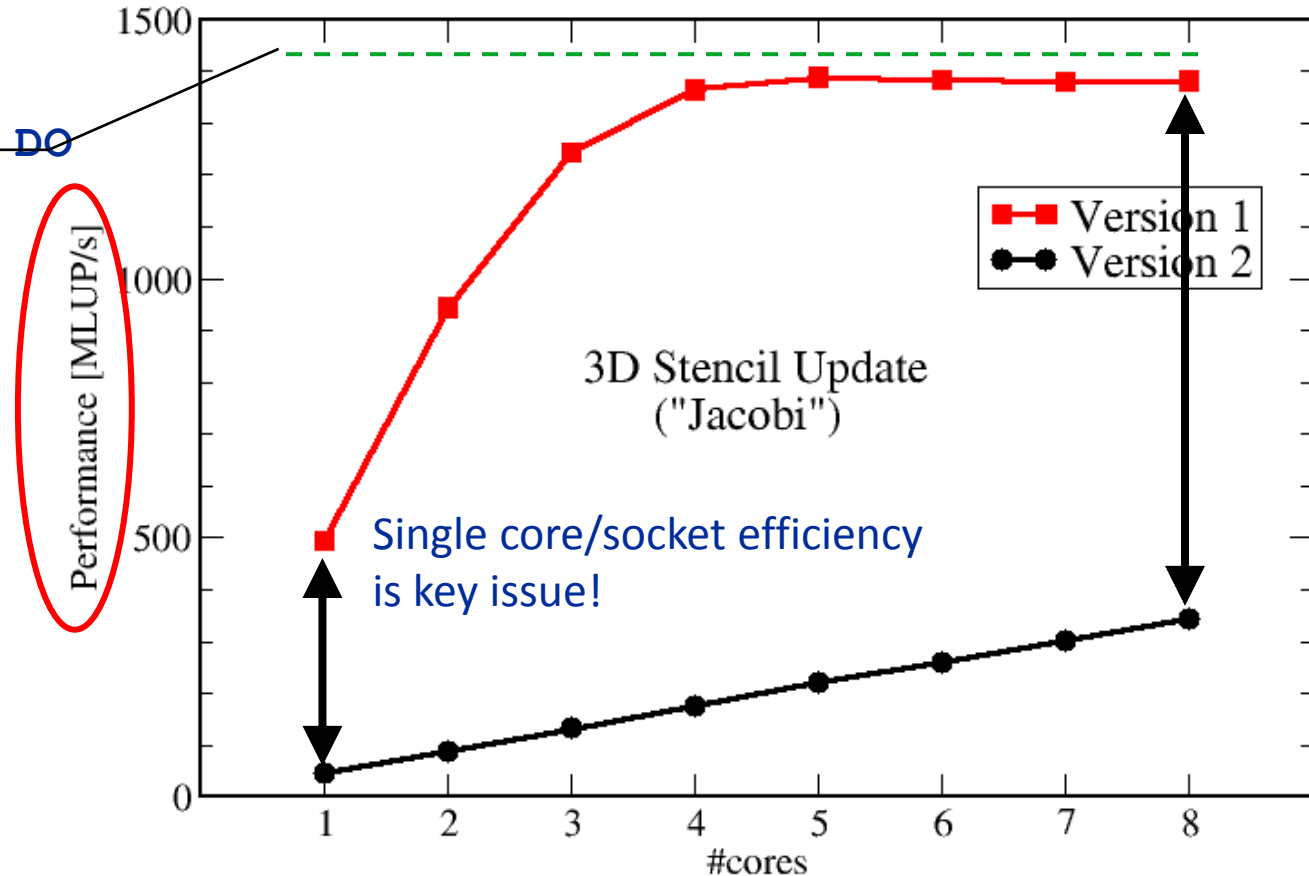
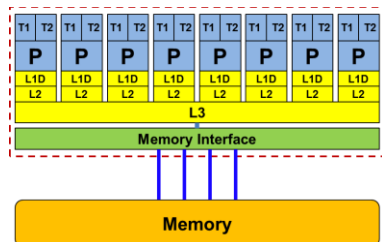
```
y(i,j,k) = b*( x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+  
x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1) )
```

```
enddo; enddo
```

```
enddo
```

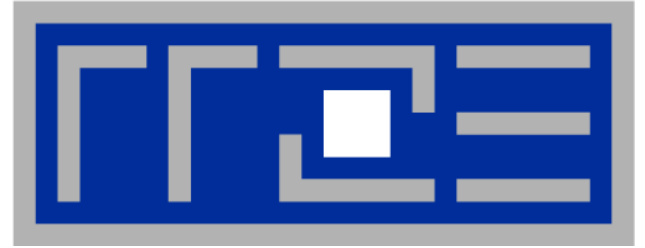
Upper limit from simple  
performance model:  
35 GB/s & 24 Byte/update

L DO





- **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



# **A little information on modern computer architecture**

**Core architecture**

**SIMD**

**Data transfers and caches**

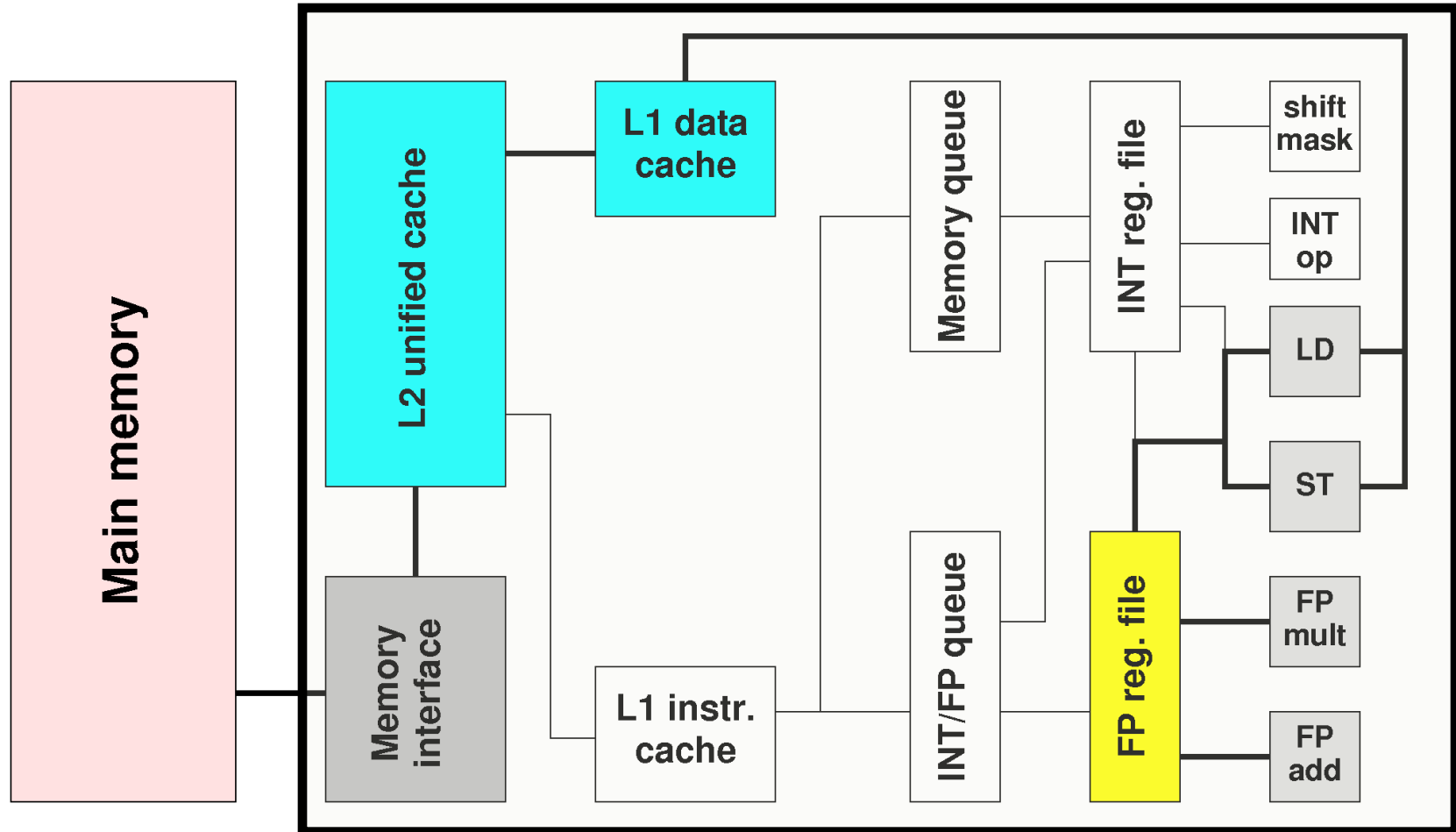
**Memory organization**

**Performance composition**

**Topology**



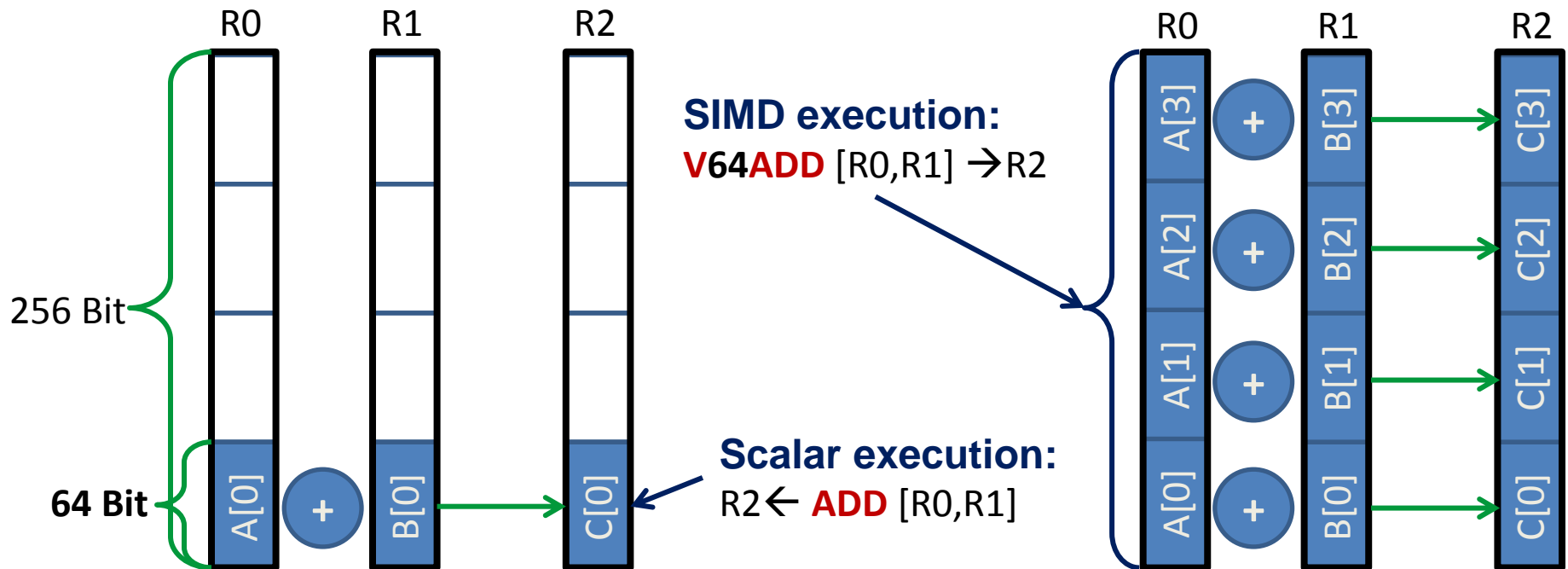
- (Almost) the same basic design in all modern systems



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

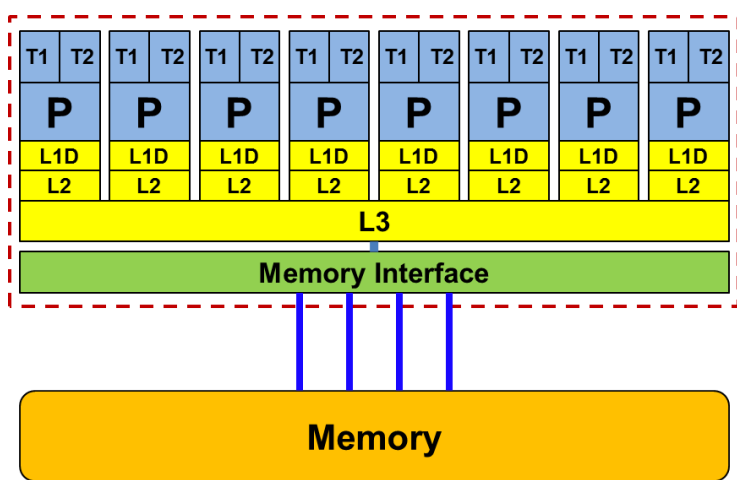


- **Single Instruction Multiple Data (SIMD)** operations allow the **concurrent execution of the same operation** on “wide” registers
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating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





# 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:

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

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

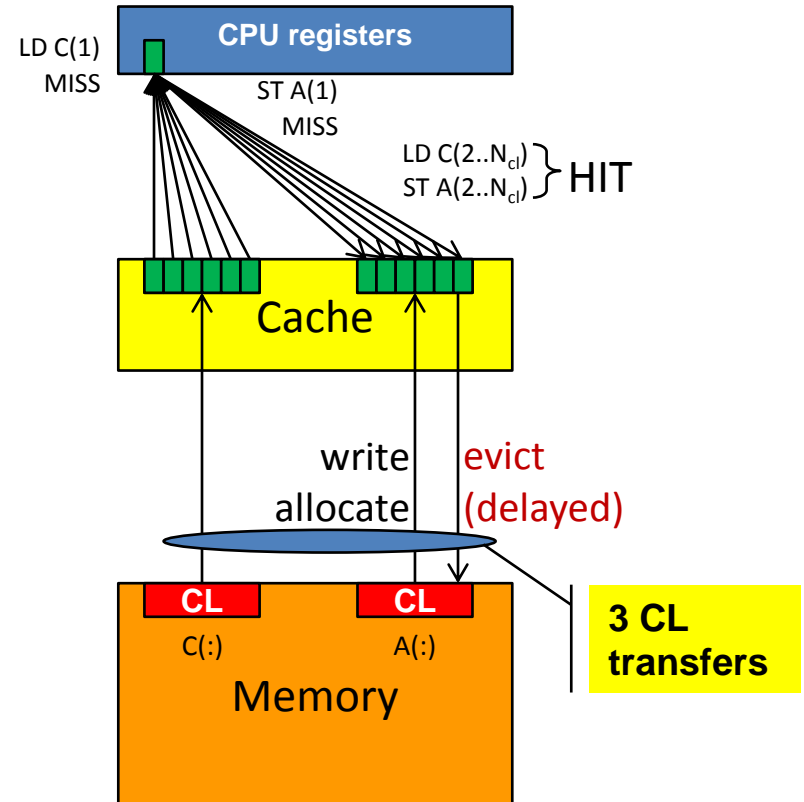
TOP500 rank 1 (mid-90s)

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

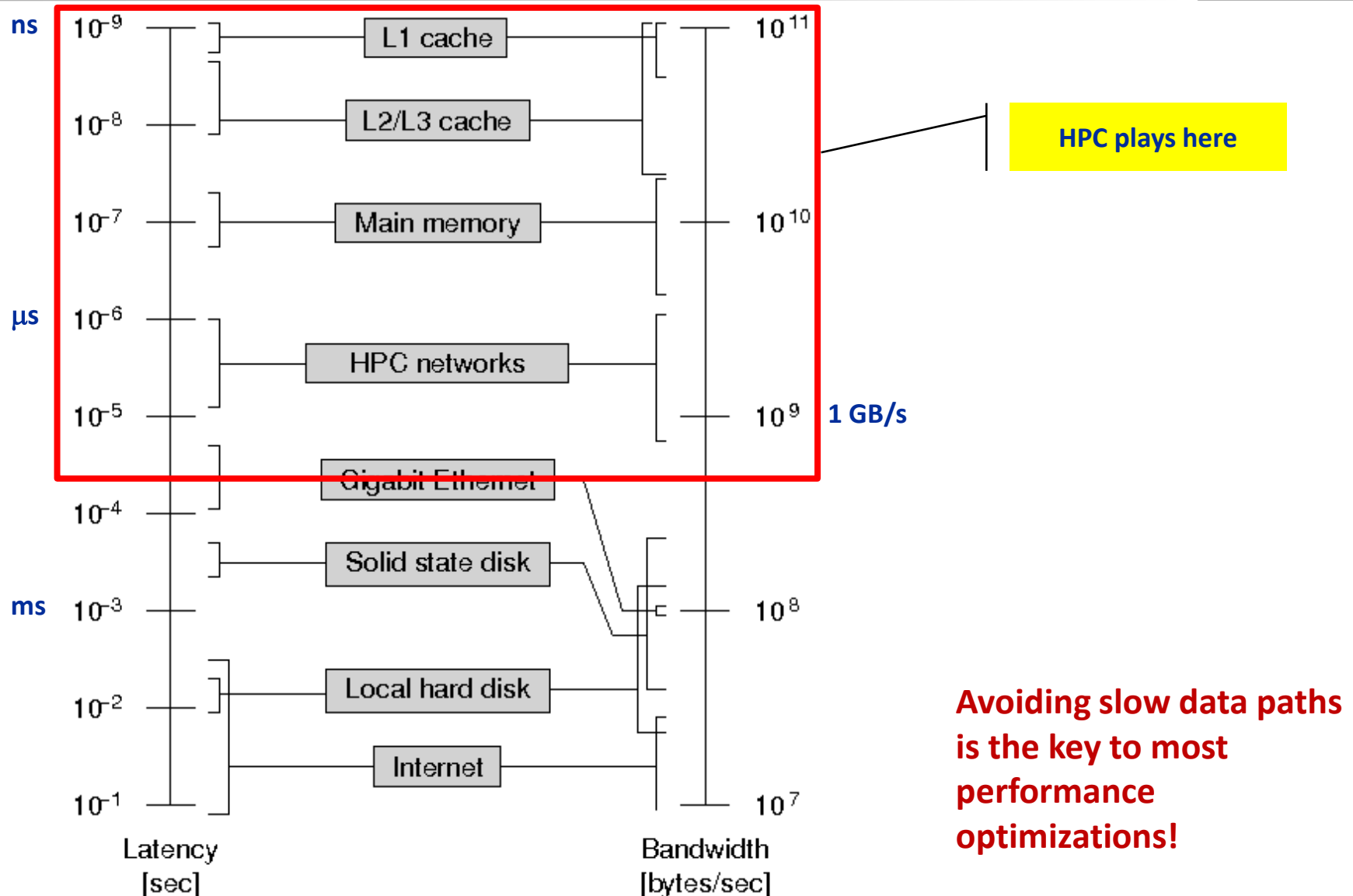
**But:  $P=5.4 \text{ GF/s}$  for serial, non-SIMD code**



- 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  $A(:) = C(:)$

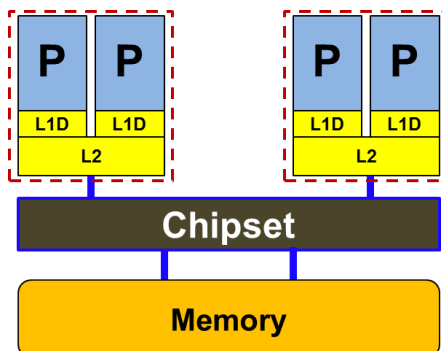


# Latency and bandwidth in modern computer environments





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

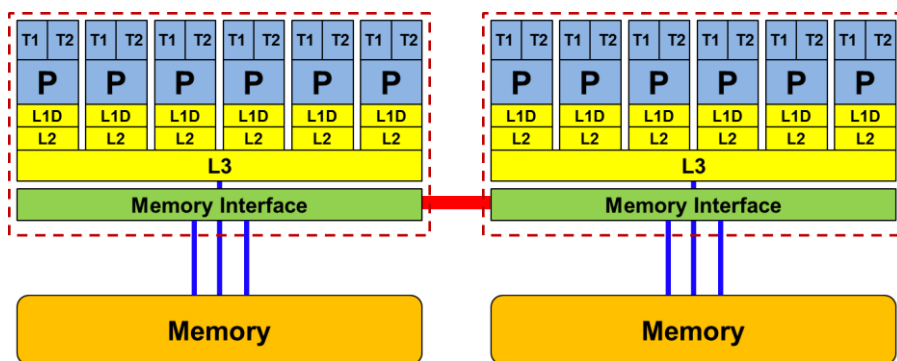


Uniform Memory Architecture (UMA)

Flat memory ; symmetric MPs

But: system “anisotropy”

### Today: Dual-socket Intel node: (Nehalem and later)

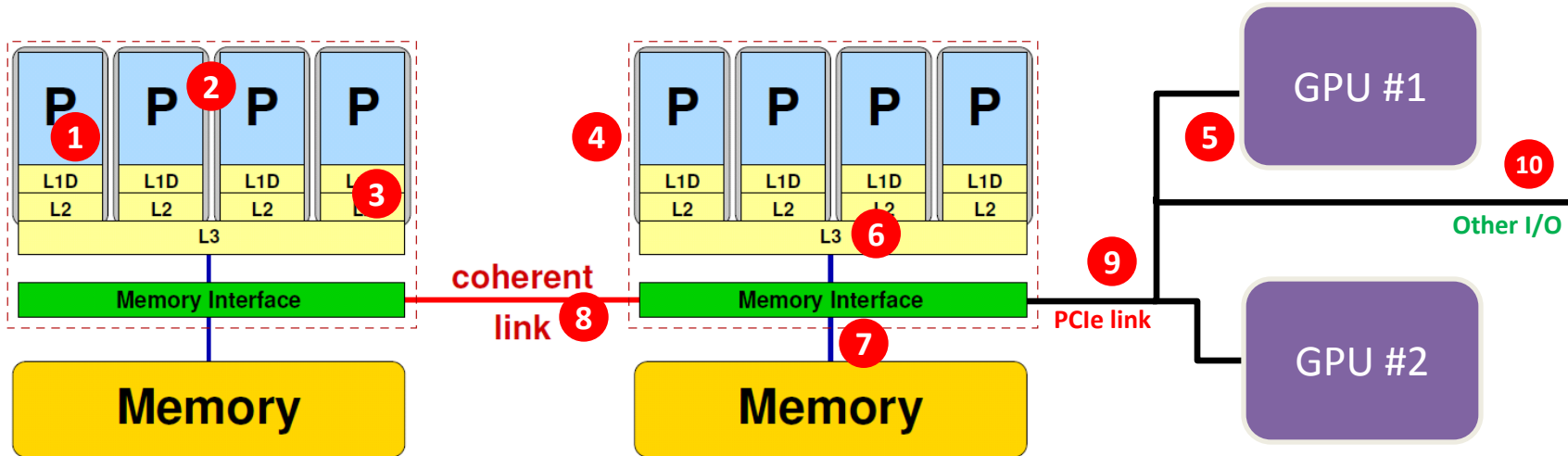


Cache-coherent Non-Uniform Memory Architecture (**ccNUMA**)

**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!**

## ■ Parallel and shared resources within a shared-memory node



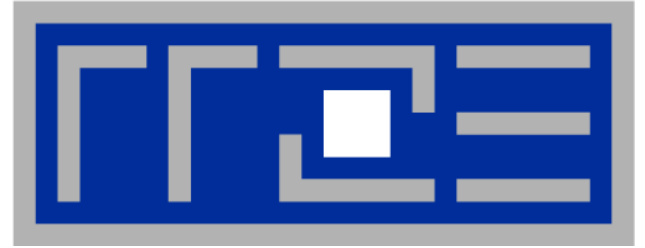
### Parallel resources:

- 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

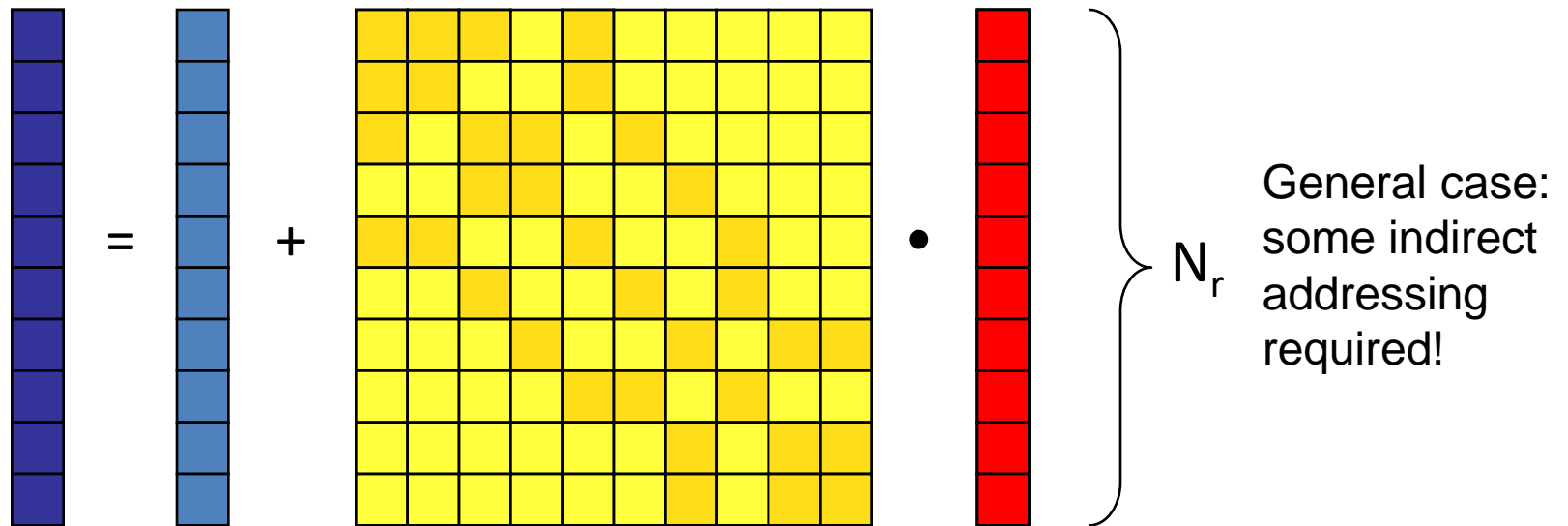
**How does your application react to all of those details?**

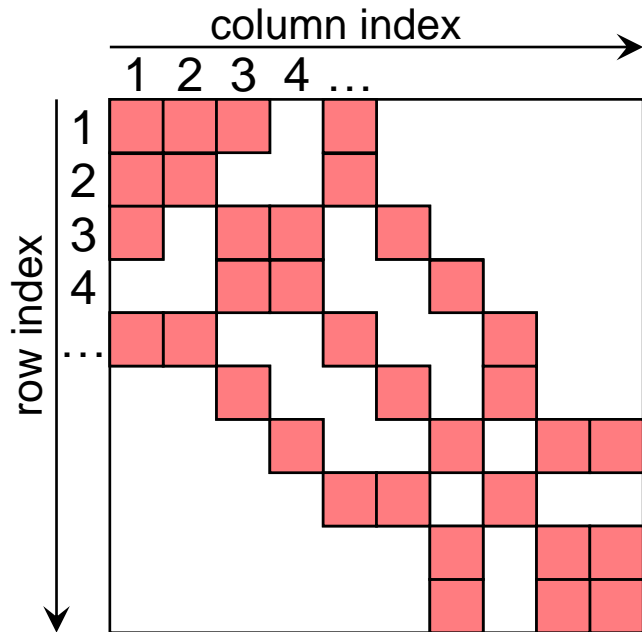


## **Case study: OpenMP-parallel sparse matrix-vector multiplication (part 1)**

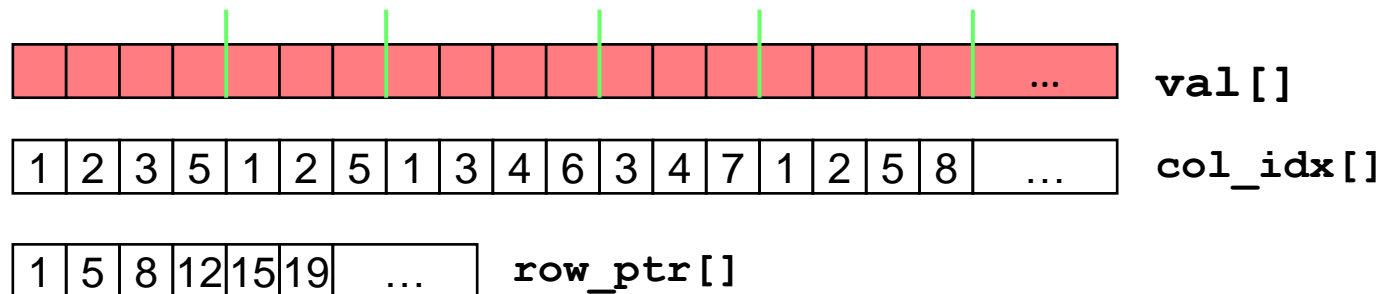
**A simple (but sometimes not-so-simple) example for  
bandwidth-bound code and saturation effects in memory**

- 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:

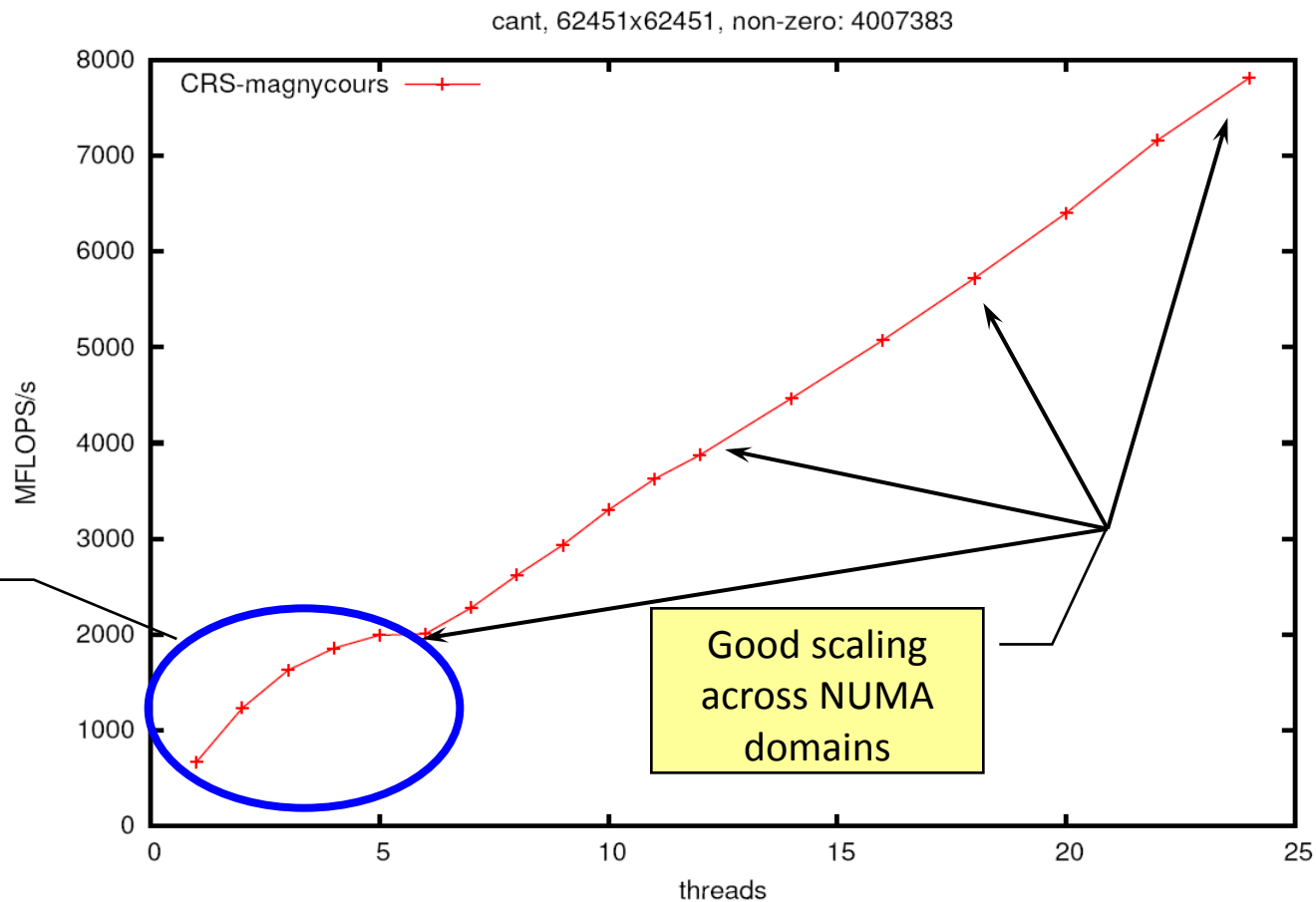
```
!$OMP parallel do
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    c(i) = c(i) + val(j) * b(col_idx(j))
  enddo
enddo
!$OMP end parallel do
```

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

### ■ Case 1: Large matrix



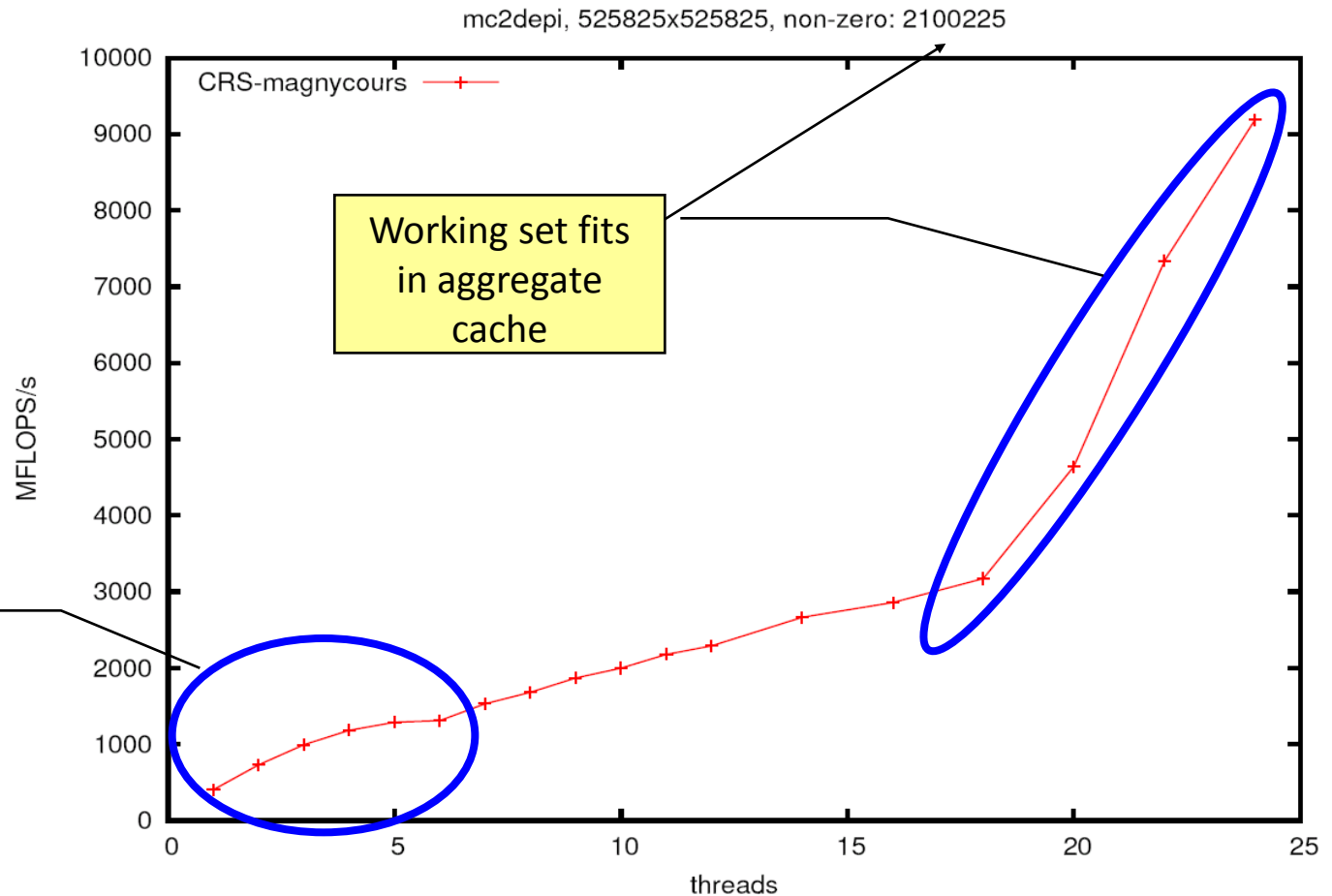
Intrasocket  
bandwidth  
bottleneck



### ■ Case 2: Medium size

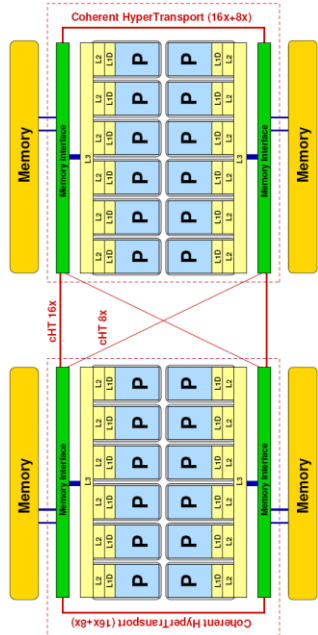


Intrasocket  
bandwidth  
bottleneck

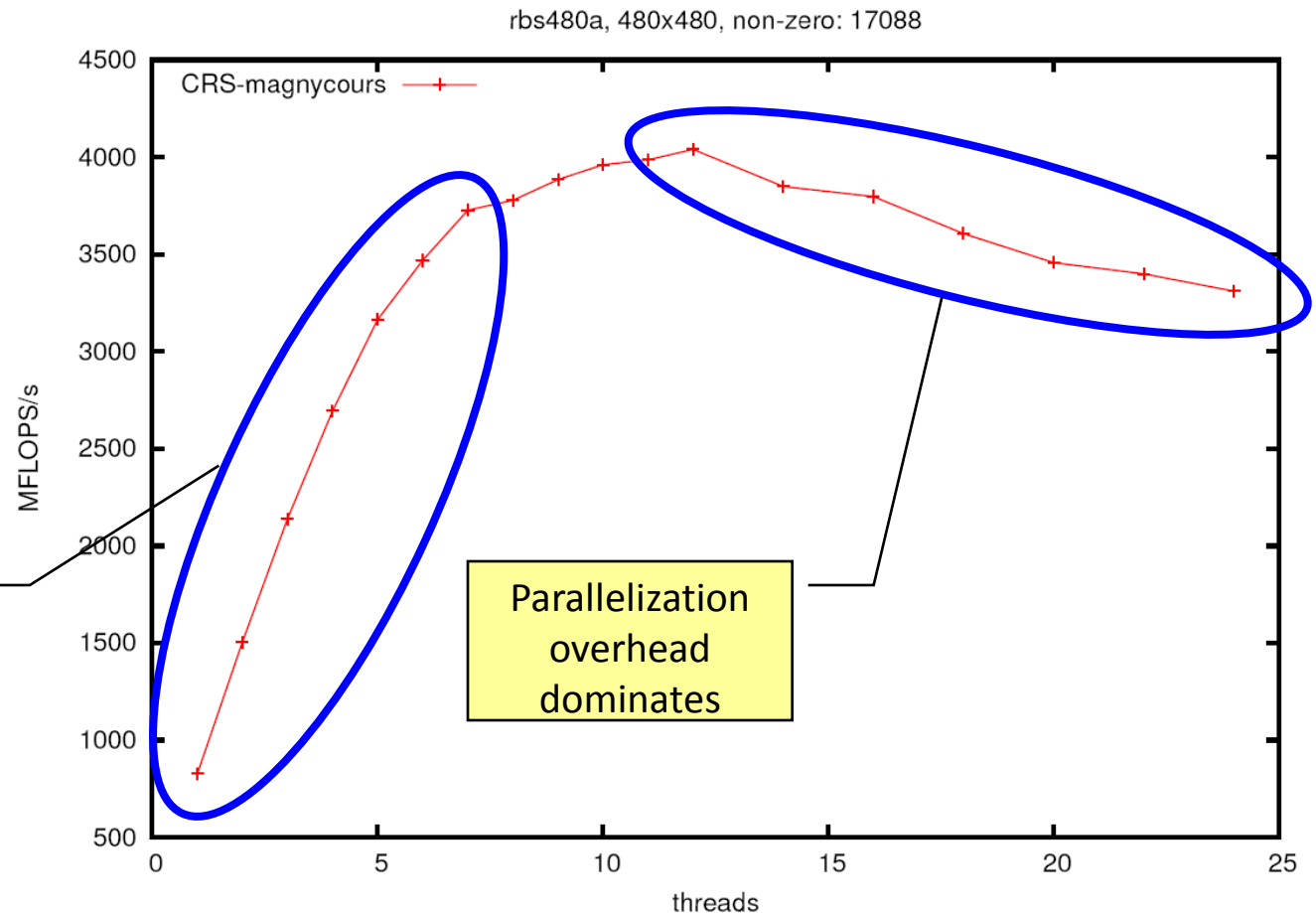




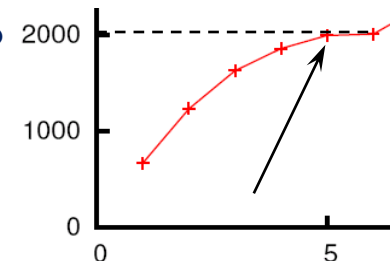
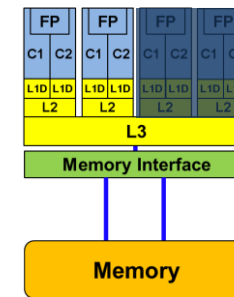
### ■ Case 3: Small size



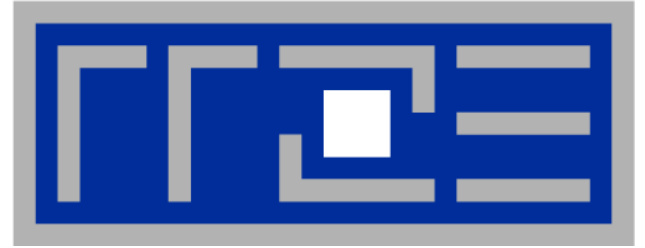
No bandwidth bottleneck



- 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?



See later for  
answers!



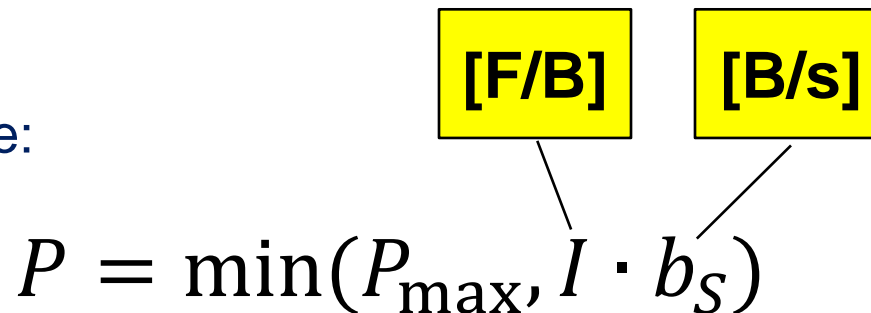
# **“Simple” performance modeling: The Roofline Model**

**Loop-based performance modeling: Execution vs. data transfer**  
**Example: array summation**



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

Expected performance:



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

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

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

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



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

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

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



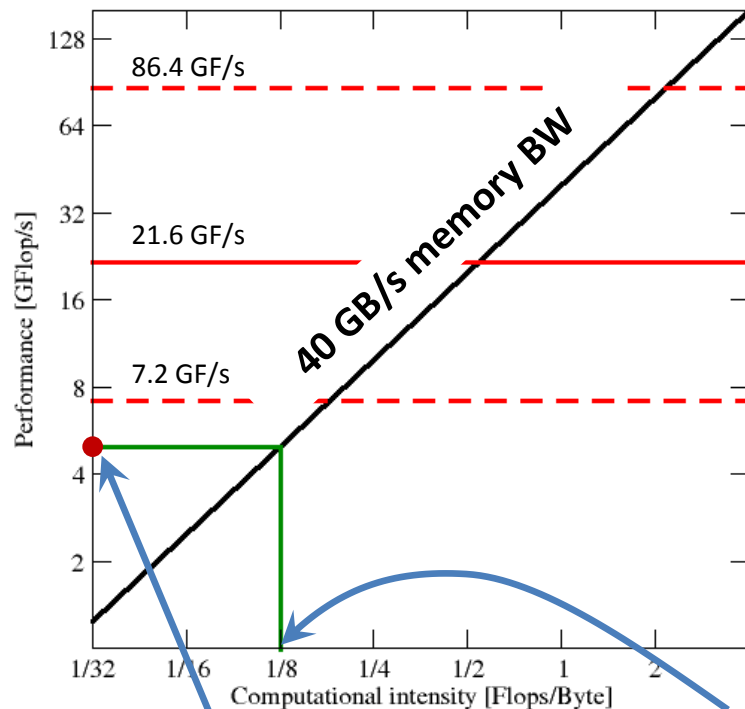


Example: **Vector triad**  $A(:) = B(:) + C(:) * D(:)$   
on a 1.05 GHz 60-core Intel Xeon Phi chip (vectorized)

- $b_S = 160 \text{ GB/s}$
- $B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F (including write allocate)}$   
 $\rightarrow I = 0.4 \text{ F/W} = 0.05 \text{ F/B}$
- $\rightarrow I \cdot b_S = 8.0 \text{ GF/s (0.8 \% of peak performance)}$
- $P_{\text{peak}} = 1008 \text{ Gflop/s}$  (60 FP units x (8+8) Flops/cy x 1.05 GHz)
- $P_{\text{max}}?$   $\rightarrow$  Observe LD/ST throughput maximum of 1 Load or 1 Store per cycle  $\rightarrow 4 \text{ cy} / 16 \text{ Flops} \rightarrow P_{\text{max}} = 252 \text{ Gflop/s (25\% of peak)}$

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

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



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

- ADD peak (best possible code)
- no SIMD

- 3-cycle latency per ADD if not unrolled

How do we get these?  
→ See next!

**$P = 5 \text{ Gflop/s}$**

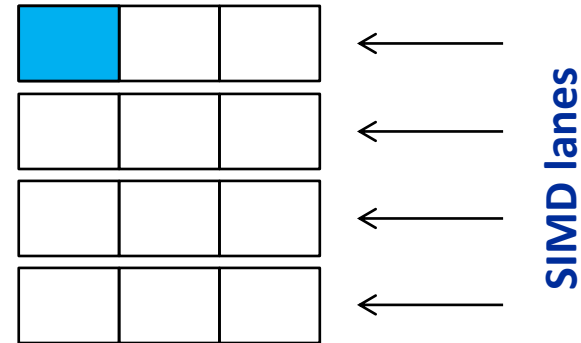
$l = 1 \text{ Flop} / 8 \text{ byte (in DP)}$



## Plain scalar code, no SIMD

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

ADD pipes utilization:



→ 1/12 of ADD peak



## Scalar code, 3-way unrolling

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

loop:

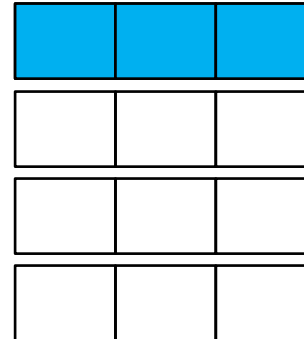
```
LOAD r4.0 ← a(i)
LOAD r5.0 ← a(i+1)
LOAD r6.0 ← a(i+2)
```

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

```
i+=3 →? loop
```

```
result ← r1.0+r2.0+r3.0
```

ADD pipes utilization:



→ 1/4 of ADD peak

# Applicable peak for the summation loop



## SIMD-vectorized, 3-way unrolled

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

loop:

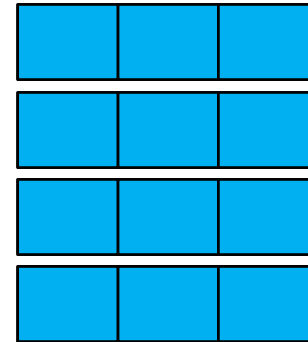
```
LOAD [r4.0,...,r4.3] ← [a(i),...,a(i+3)]
LOAD [r5.0,...,r5.3] ← [a(i+4),...,a(i+7)]
LOAD [r6.0,...,r6.3] ← [a(i+8),...,a(i+11)]
```

```
ADD r1 ← r1+r4
ADD r2 ← r2+r5
ADD r3 ← r3+r6
```

i+=12 →? loop

result ← r1.0+r1.1+...+r3.2+r3.3

ADD pipes utilization:



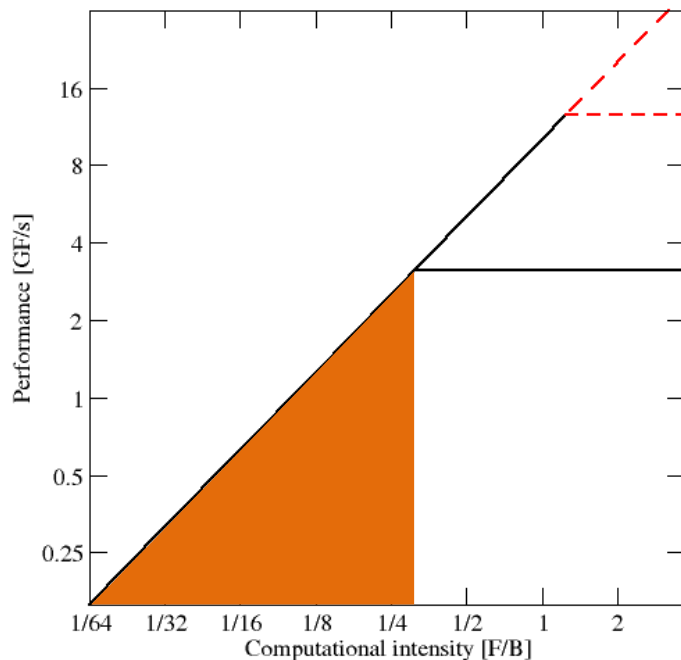
→ ADD peak



- **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**

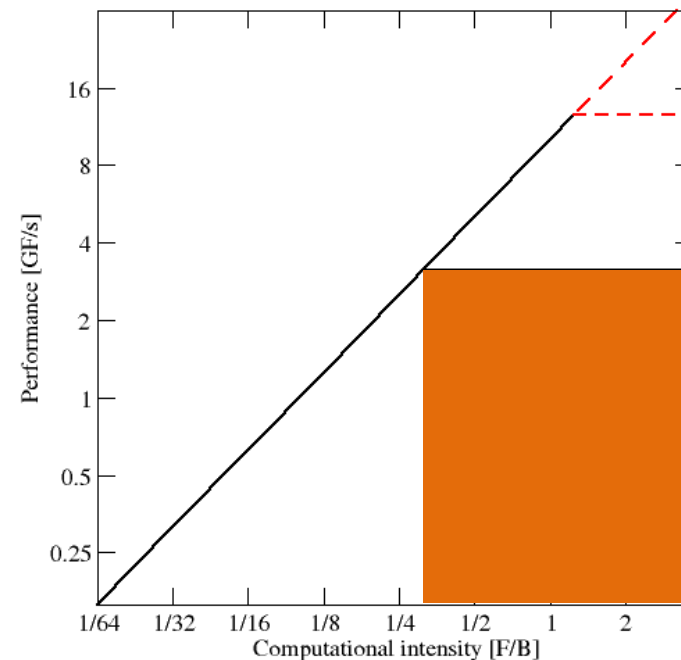
## Bandwidth-bound (simple case)

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



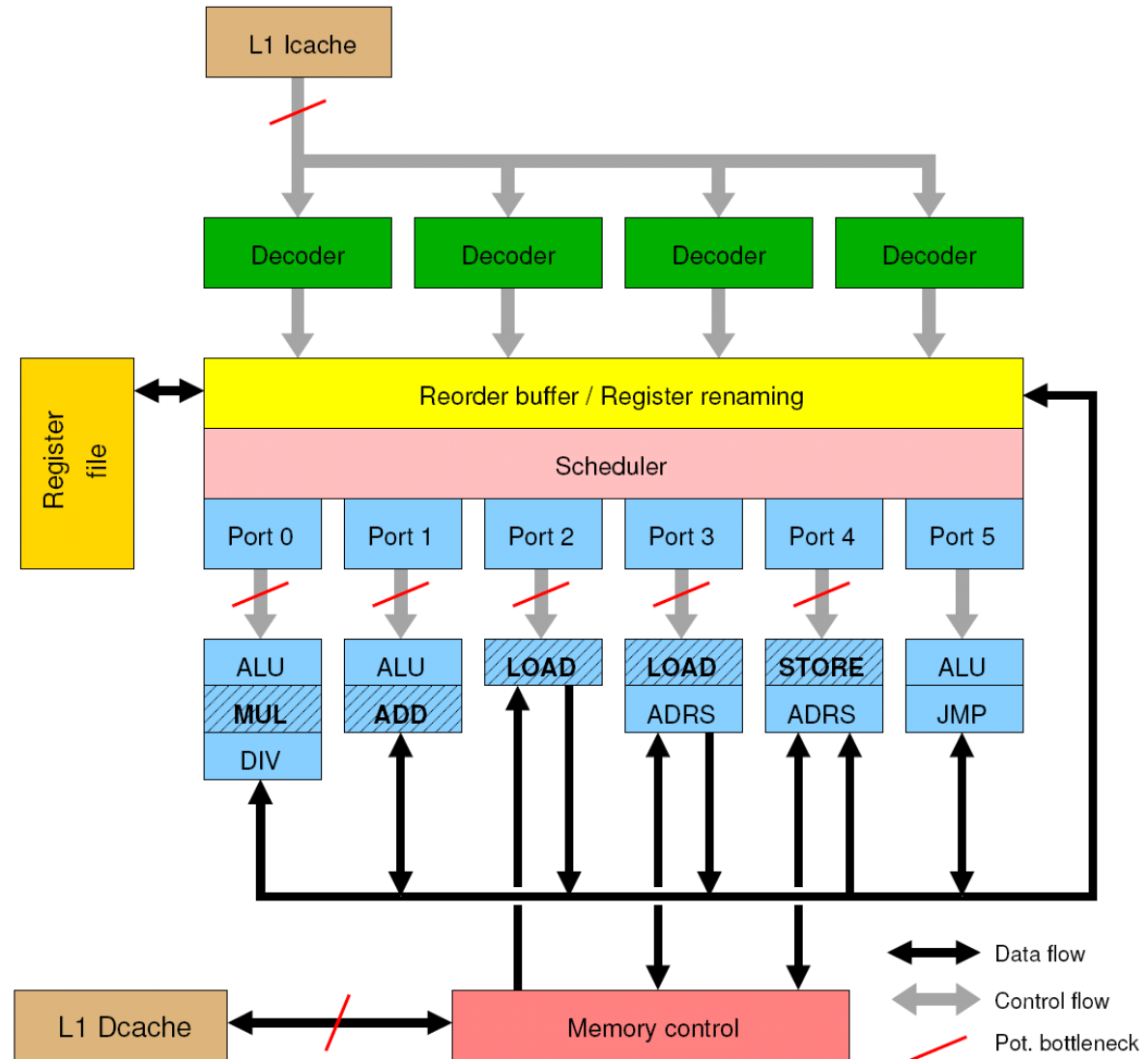
## Core-bound (may be complex)

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



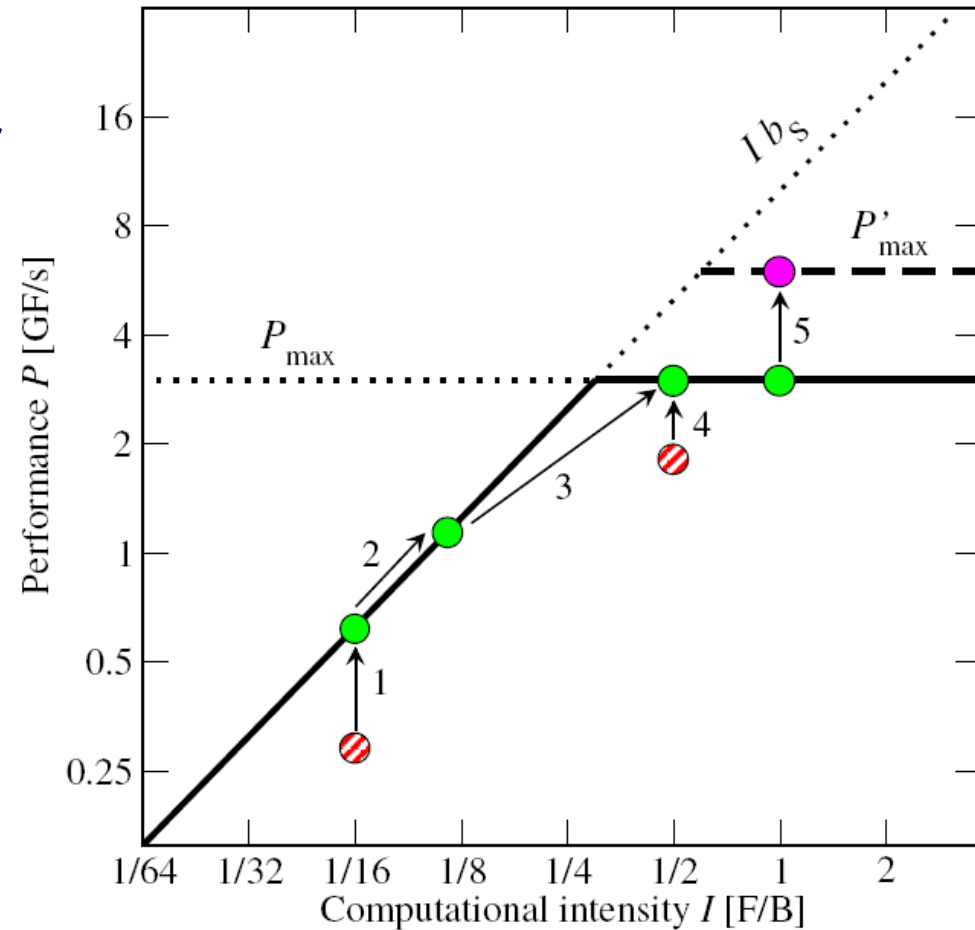
## Multiple bottlenecks:

- 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



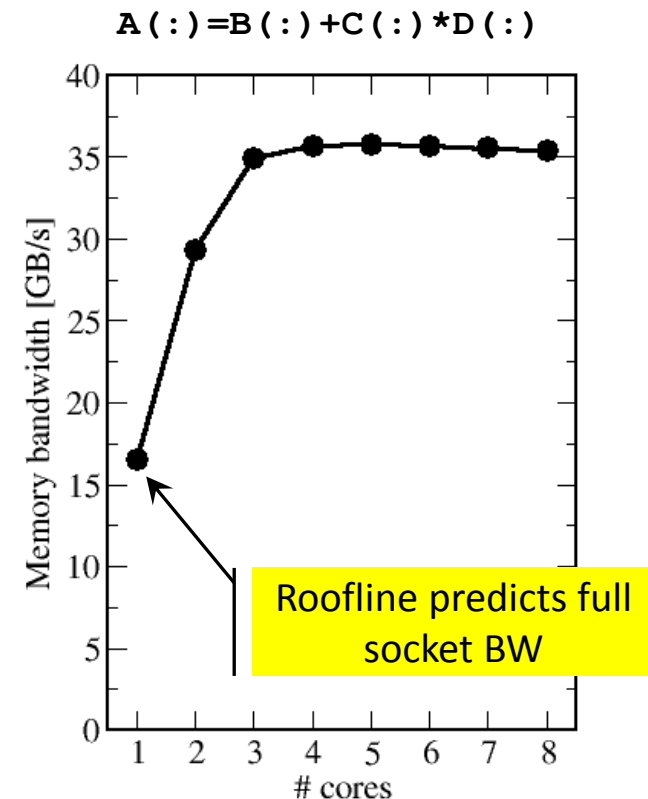


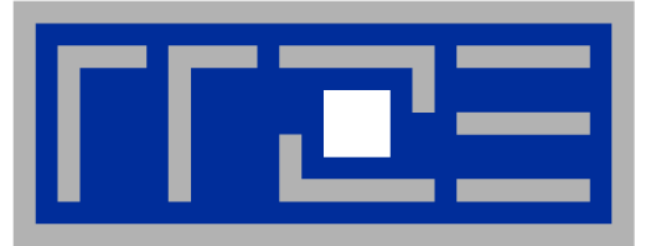
1. Hit the BW bottleneck by good serial code
2. Increase intensity to make better use of BW bottleneck
3. Increase intensity and go from memory-bound to core-bound
4. Hit the core bottleneck by good serial code
5. Shift  $P_{\max}$  by accessing additional hardware features or using a different algorithm/implementation



- **Saturation effects** in multicore chips are not explained
  - Reason: “**saturation assumption**”
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - Only increased “**pressure**” on the memory **interface** can saturate the bus  
→ need more cores!
- **ECM model** gives more insight

G. Hager, J. Treibig, J. Habich, and G. Wellein: **Exploring performance and power properties of modern multicore chips via simple machine models**. Concurrency and Computation: Practice and Experience.  
DOI: [10.1002/cpe.3180](https://doi.org/10.1002/cpe.3180) Preprint: [arXiv:1208.2908](https://arxiv.org/abs/1208.2908)





# Putting Roofline to use where it should not work

Sparse matrix-vector multiplication, part 2



- **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**

- $\alpha$  quantifies traffic for loading RHS

- $\alpha = 0 \rightarrow$  RHS is in cache
- $\alpha = 1/N_{nzs} \rightarrow$  RHS loaded once
- $\alpha = 1 \rightarrow$  no cache
- $\alpha > 1 \rightarrow$  Houston, we have a problem!

- “Expected” performance =  $b_s \times I_{CRS}$
- Determine  $\alpha$  by measuring the actual memory traffic
  - Maximum memory BW may not be achieved with spMVM

$$I_{CRS}^{DP} = \frac{2}{\boxed{8} + \boxed{4} + \boxed{8\alpha} + \boxed{16/N_{nzs}}} \frac{\text{flops}}{\text{byte}}$$

$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzs}} \frac{\text{flops}}{\text{byte}} = \frac{N_{nz} \cdot 2 \text{ flops}}{V_{meas}}$$

- $V_{meas}$  is the measured overall memory data traffic (using, e.g., `likwid-perfctr`)

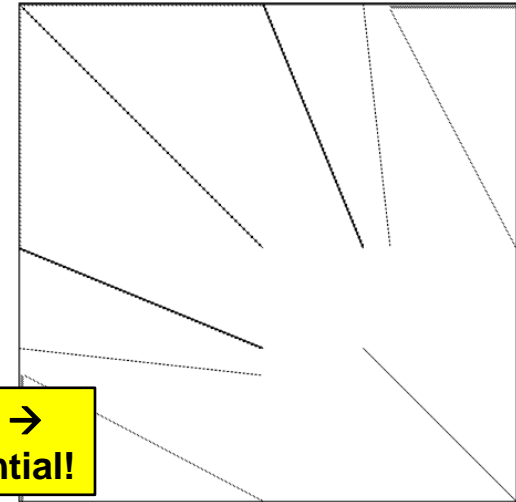
- Solve for  $\alpha$ : 
$$\alpha = \frac{1}{4} \left( \frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{8}{N_{nzs}} \right)$$

- Example: `kkt_power` matrix from the UoF collection on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6$ ,  $N_{nzs} = 7.1$
- $V_{meas} \approx 258 \text{ MB}$
- $\rightarrow \alpha = 0.43$ ,  $\alpha N_{nzs} = 3.1$
- $\rightarrow$  RHS is loaded 3.1 times from memory
- and:

$$\frac{I_{CRS}^{DP}(1/N_{nzs})}{I_{CRS}^{DP}(\alpha)} = 1.15$$

15% extra traffic  $\rightarrow$  optimization potential!





- **Conclusion from Roofline analysis**

- The roofline model does not work 100% for spMVM due to the RHS traffic uncertainties
- We have “turned the model around” and measured the actual memory traffic to determine the RHS overhead
- Result indicates:
  1. how much actual traffic the RHS generates
  2. how efficient the RHS access is (compare BW with max. BW)
  3. how much optimization potential we have with matrix reordering

- **Consequence: If the model does not work, we learn something!**