ee also the Online-Recording:
tp://www.hlrs.de/training/par-prog-ws/2014-nlp

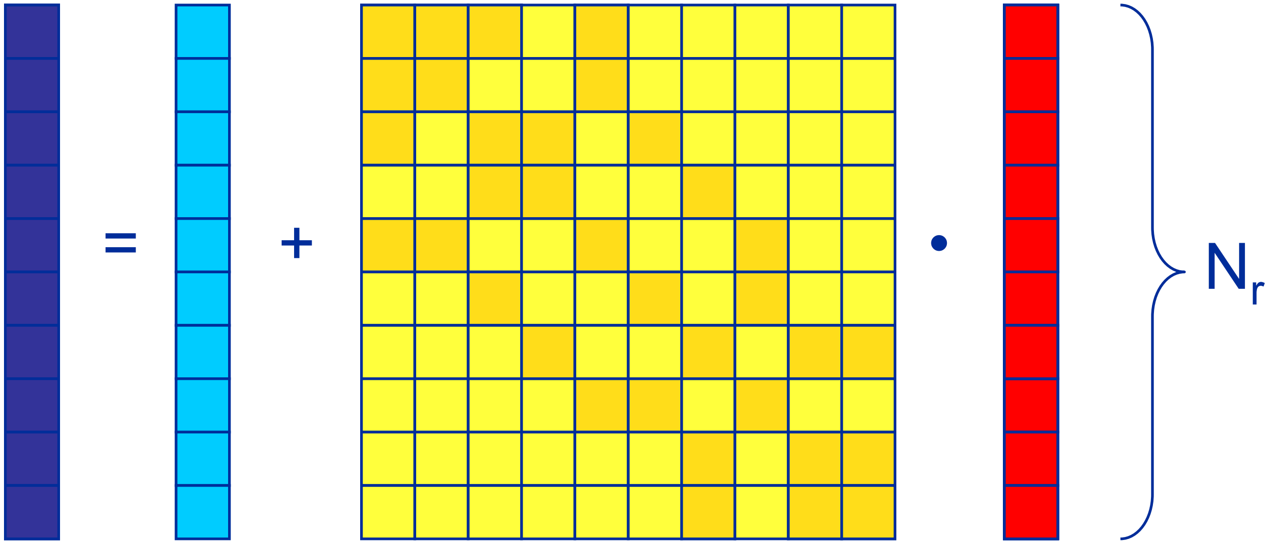
Case study: OpenMP-parallel sparse matrix-vector multiplication

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

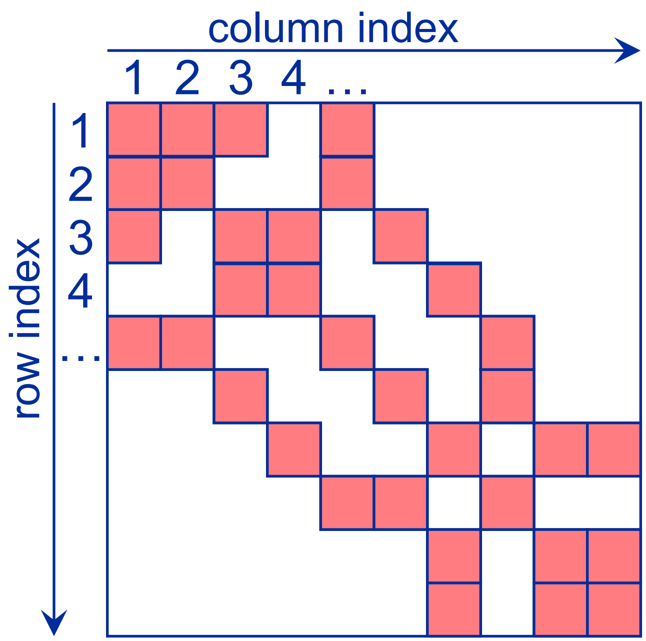
Sparse matrix-vector multiply (spMVM)

- **Key ingredient in some matrix diagonalization algorithms**
  - Lanczos, Davidson, Jacobi-Davidson

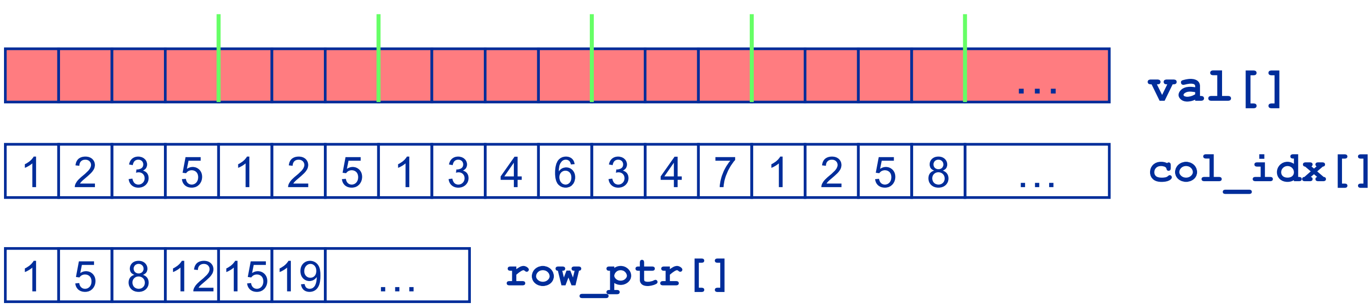
- **Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries**
- **"Sparse": $N_{nz} \sim N_r$**



CRS matrix storage scheme



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



Case study: Sparse matrix-vector multiply

- **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

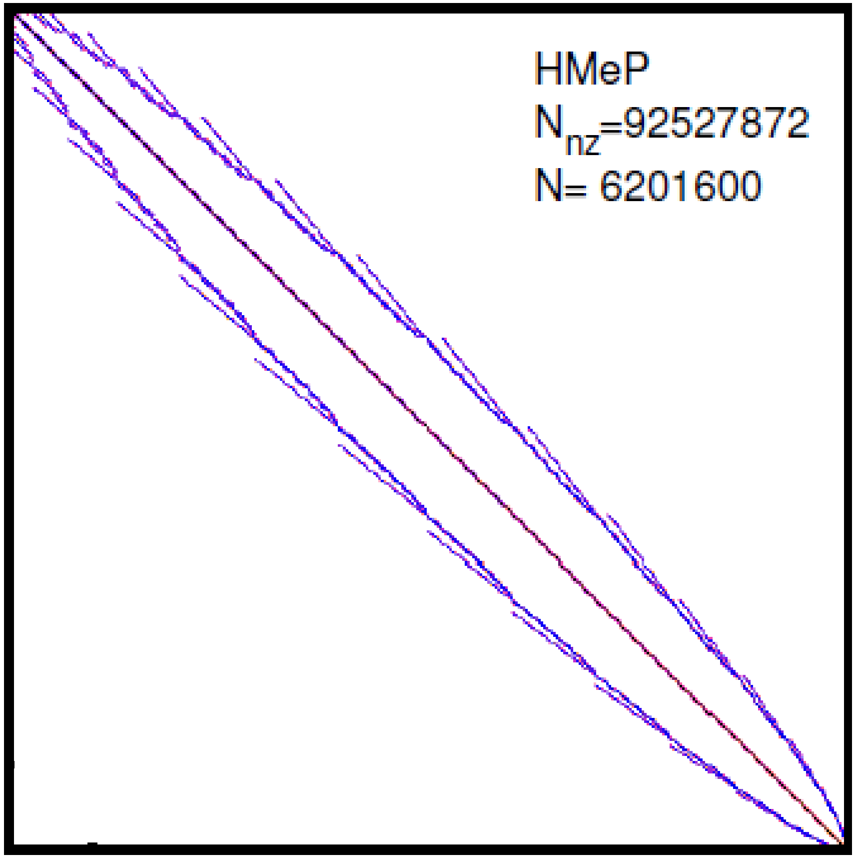
- **Following slides: Performance data on one 24-core AMD Magny Cours node**

Bandwidth-bound parallel algorithms:
Sparse MVM

- **Data storage format is crucial for performance properties**
  - Most useful general format: Compressed Row Storage (CRS)
  - SpMVM is easily parallelizable in shared and distributed memory

- **For large problems, spMVM is inevitably memory-bound**
  - Intra-LD saturation effect on modern multicores

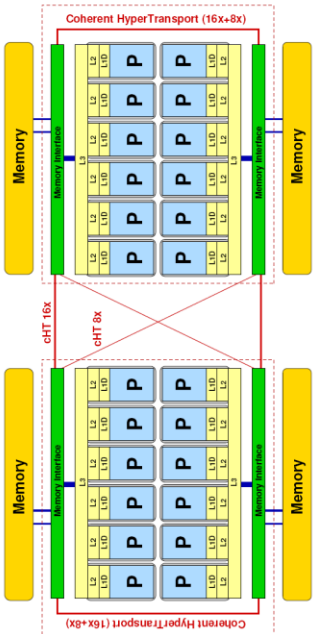
- **MPI-parallel spMVM is often communication-bound**
  - See later part for what we can do about this…



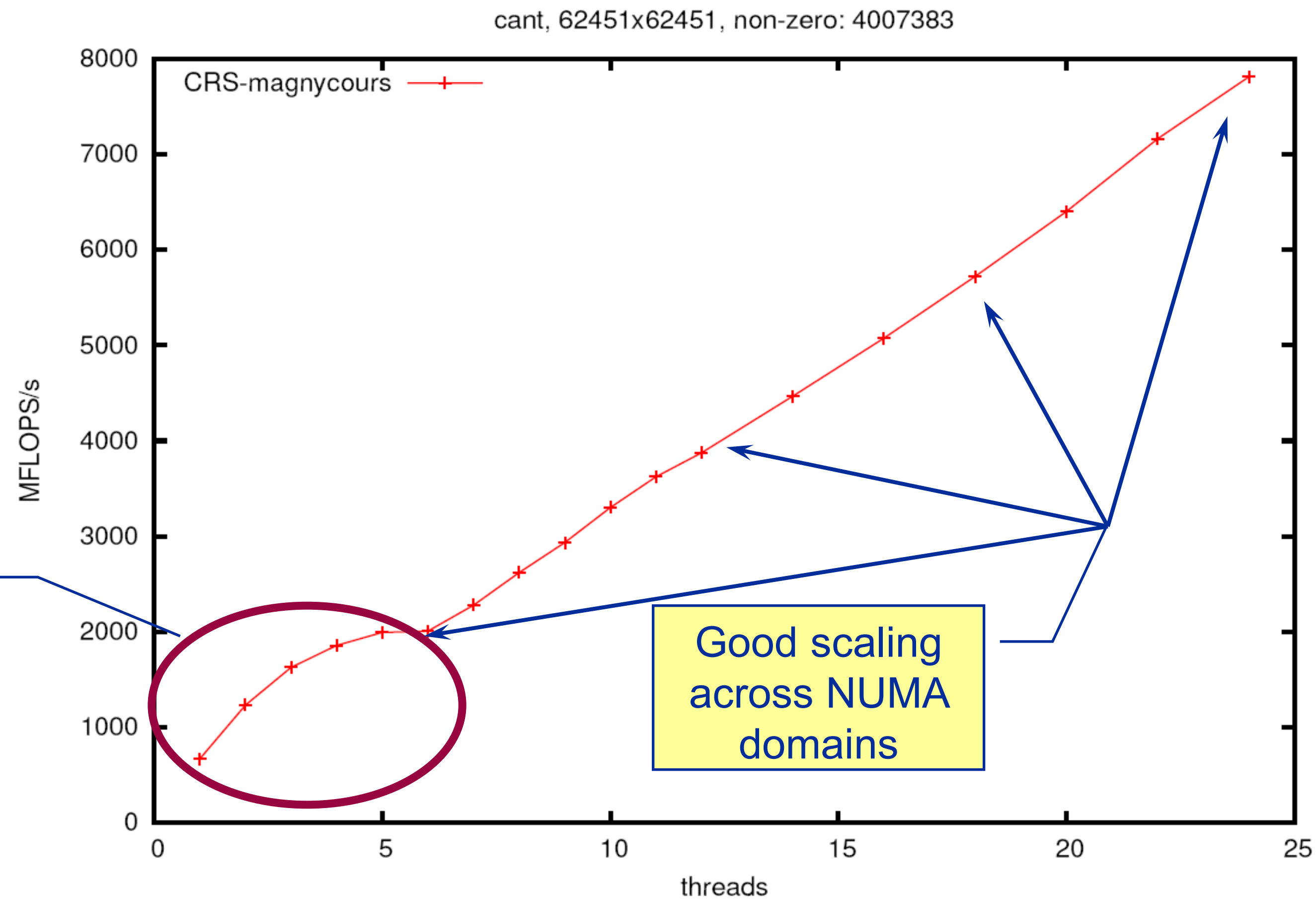
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- ## Case 1: Large matrix

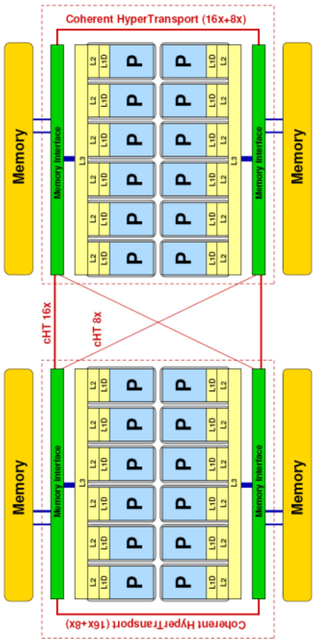


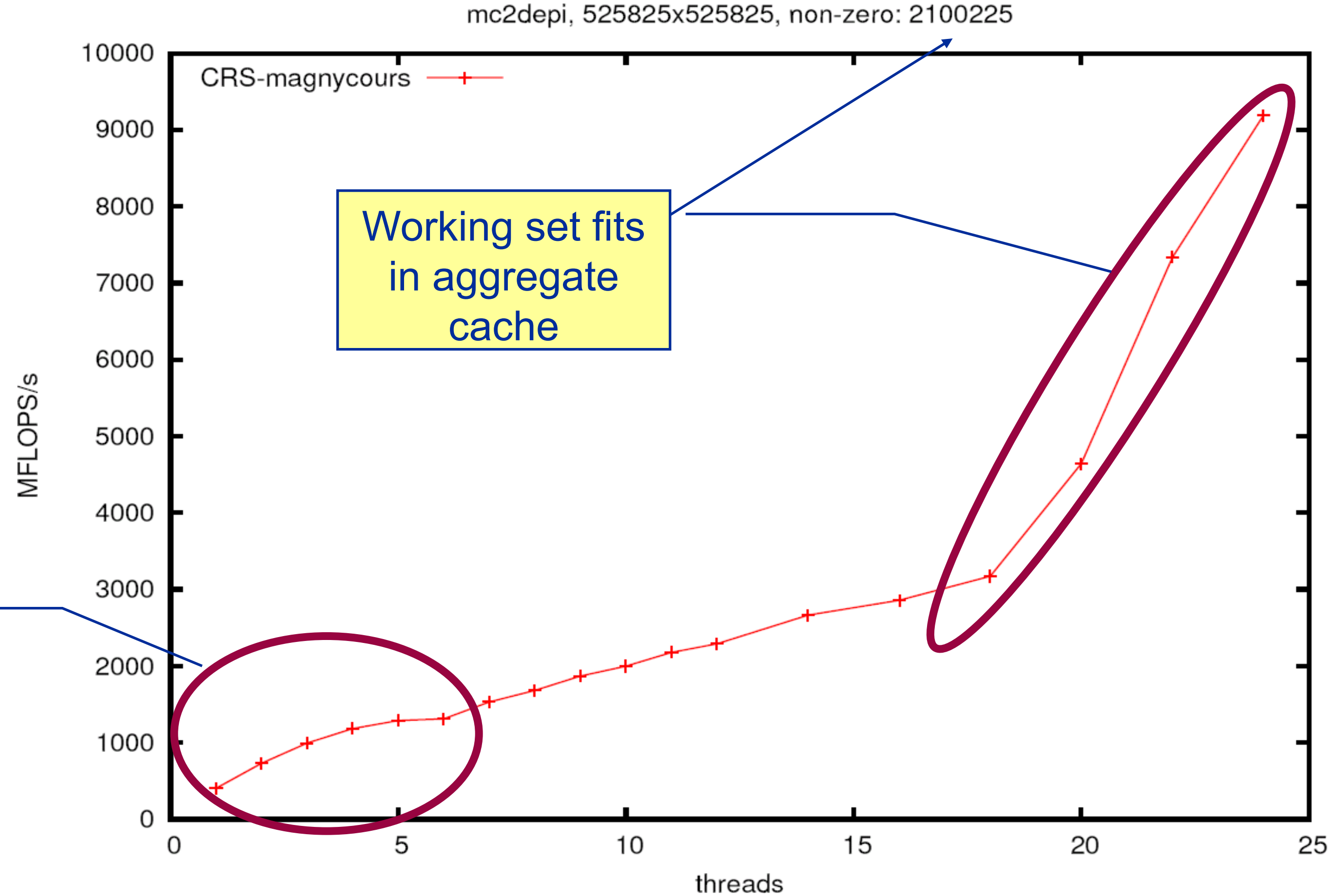
Intrasocket bandwidth bottleneck



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

- ## Case 2: Medium size

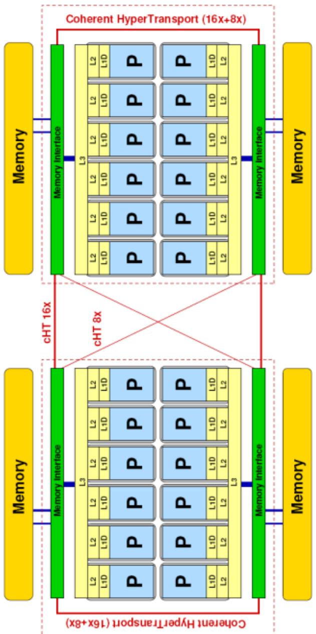




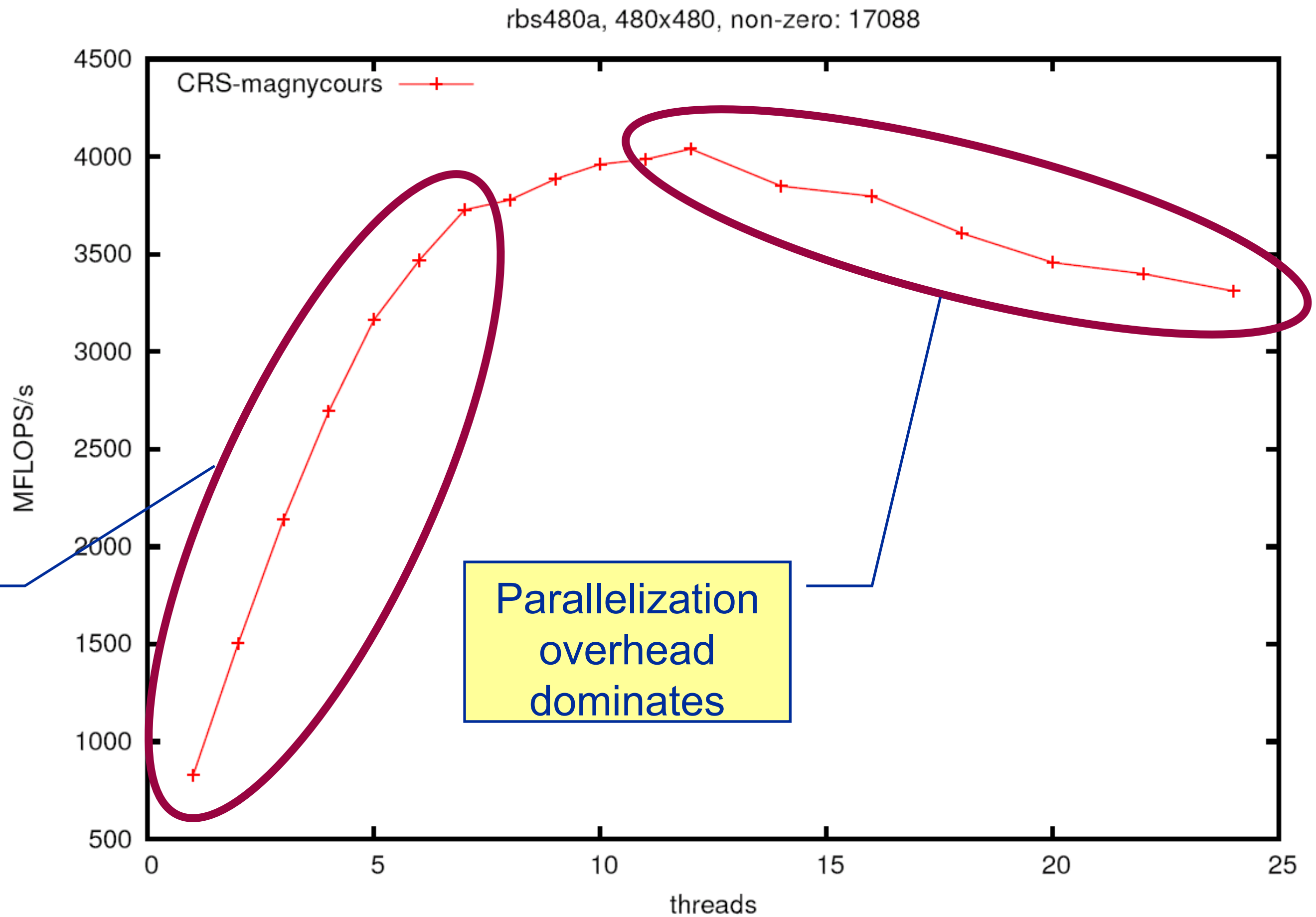
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 3: Small size**



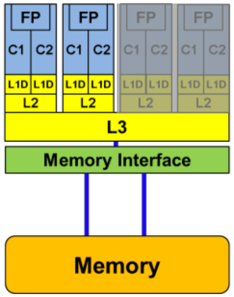
No bandwidth bottleneck



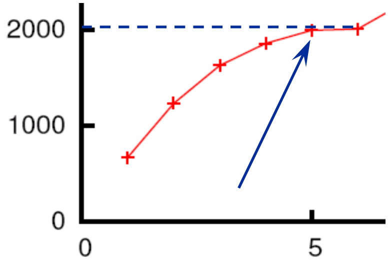
Parallelization overhead dominates

Conclusions from the spMVM benchmarks

- **If the problem is "large", bandwidth saturation on the socket is a reality**
  - → There are "spare cores"
  - Very common performance pattern
- **What to do with spare cores?**
  - Let them idle → saves energy with minor loss in time to solution
  - Use them for other tasks, such as MPI communication



- **Can we predict the saturated performance?**
  - Bandwidth-based performance modeling!
  - What is the significance of the indirect access? Can it be modeled?
- **Can we predict the saturation point?**
  - … and why is this important?



Example: SpMVM chip performance model

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

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

- **DP CRS comp. intensity**

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

  - α quantifies traffic for loading RHS
    - $\alpha = 0$ → RHS is in cache
    - $\alpha = 1/N_{nzr}$ → RHS loaded once
    - $\alpha = 1$ → no cache
    - $\alpha > 1$ → Houston, we have a problem!
  - "Expected" performance = b_S x I_{CRS}
  - Determine α by measuring performance and actual memory traffic
    - Maximum memory BW may not be achieved with spMVM

Determine RHS traffic

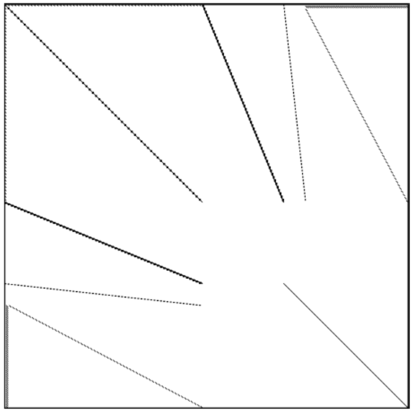
$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \frac{\text{flops}}{\text{byte}} = \frac{N_{nz} \cdot 2\ \text{flops}}{V_{meas}}$$

- V_{meas} **is the measured overall memory data traffic (using, e.g., likwid-perfctr)**
- **Solve for α:**

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

- **Example: kkt_power matrix from the UoF collection on one Intel SNB socket**
  - $N_{nz} = 14.6 \cdot 10^6$, $N_{nzr} = 7.1$
  - $V_{meas} \approx 258\ \text{MB}$
  - → $\alpha = 0.43$, $\alpha N_{nzr} = 3.1$
  - → RHS is loaded 3.1 times from memory
  - and: $\frac{I_{CRS}^{DP}(1/N_{nzr})}{I_{CRS}^{DP}(\alpha)} = 1.15$

15% extra traffic → optimization potential!



Roofline analysis for spMVM

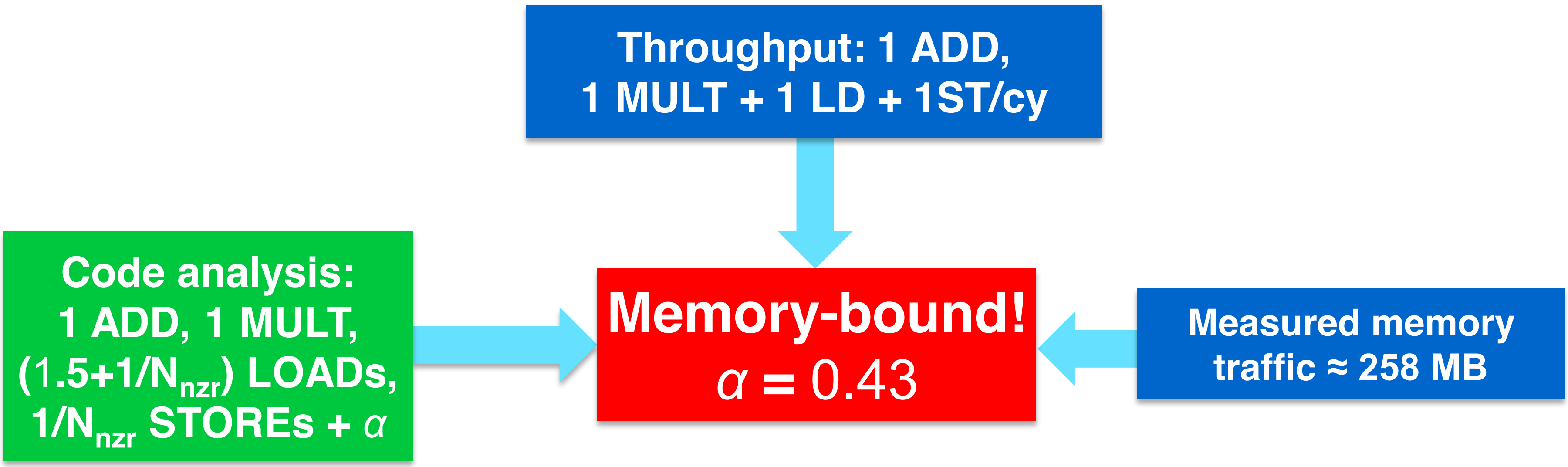
- **Conclusion**
  - The roofline model does not work 100% for spMVM due to the RHS traffic uncertainties
  - We have "turned the model around" and measured the actual memory traffic to determine the RHS overhead
  - Result indicates:
    1. how much actual traffic the RHS generates
    2. how efficient 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!**

Input to the roofline model

optional

… on the example of spMVM with kkt_power matrix



optional

A word on sparse matrix storage formats

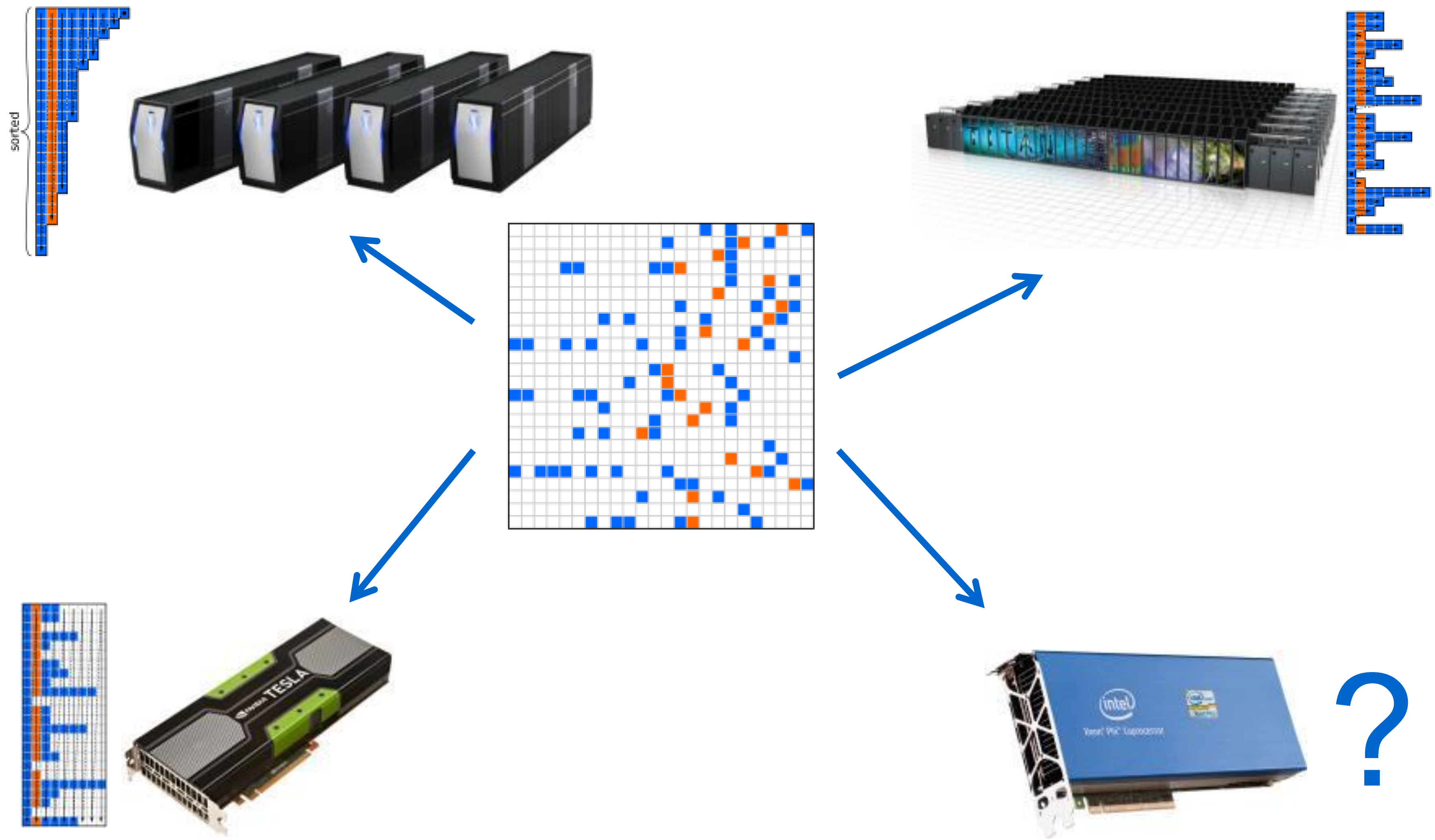
CRS

Sliced ELLPACK

SELL-C-σ

Sparse Matrix Format Jungle

optional



SpMVM in the Heterogeneous Era

optional

- **Compute clusters are getting more and more heterogeneous**

- **A special format per compute architecture**
  1. hampers runtime exchange of matrix data
  2. complicates library interfaces

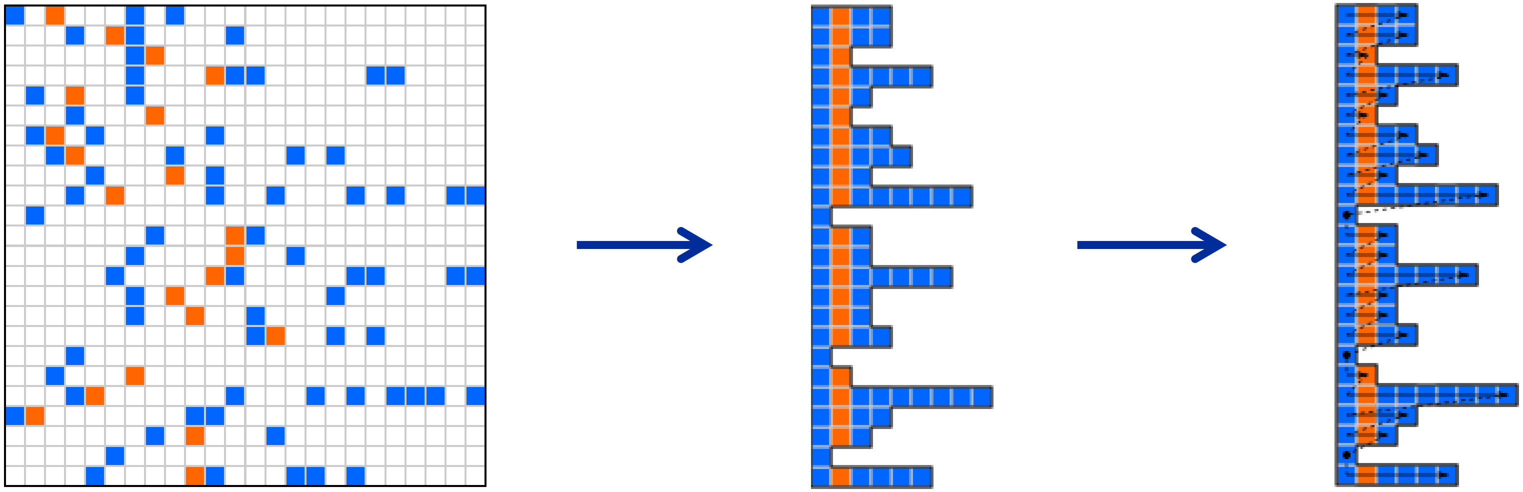
- **CRS (CPU standard format) may be problematic (cf. next slide)**
  - Vectorization along matrix rows
  - Bad utilization for short rows and wide SIMD units (Intel MIC: 512 bit)

➔ We want to have a unified, SIMD-friendly, and high-performance sparse matrix storage format.

Compressed Row Storage (CRS)

optional

- Standard format for CPUs



- Entries and column indices stored row-wise

CRS Vectorization

optional

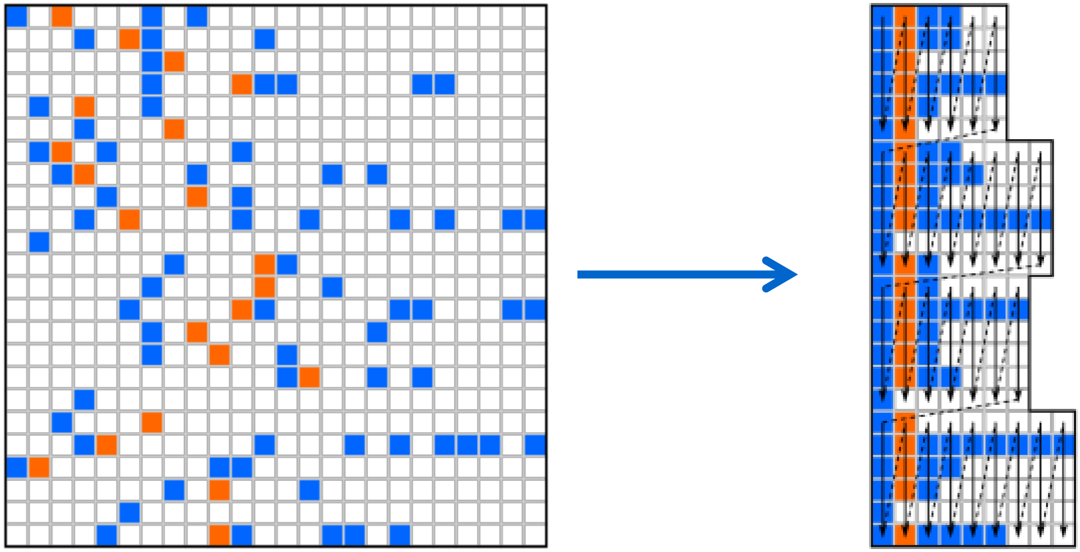
```
unsigned int i, j;
double tmp;

#pragma omp parallel for schedule(runtime) private (tmp1, tmp2, j)
for (i=0; i<nrows; i++){
    tmp1 = 0.0;
    tmp2 = 0.0;
    for (j=rpt[i]; j<rpt[i+1]; j=j+2){
        tmp1 += val[j] *   rhs[col[j]];
        tmp2 += val[j+1] * rhs[col[j+1]];
    }
    lhs[i] += tmp1+tmp2;
}
```

SSE vectorization

- **Potential problem: Long vector registers on modern CPUs (e.g., 512 bit on Xeon Phi)**
  - 512 bit → 8 doubles or 16 integers in a single vector
  - j-loop:16-way unrolling ➔ problem for short rows

Sliced ELLPACK

- **Well-known sparse matrix format for GPUs**



- **Entries and column indices stored column-wise in chunks**
- **One parameter:**
  1. C: Chunk height

Sliced ELLPACK

optional

Potential problem:
Depending on the variation in the row length, a more or less significant amount of zeros will be loaded and processed, quantified by β („chunk occupancy“):

$$\beta = \frac{N_{\mathrm{nz}}}{\sum_{i=0}^{N_c} C \cdot \mathtt{cl[i]}}$$

C............ chunk height
N_c......... number of chunks
cl[k]......... max. row length in k-th chunk

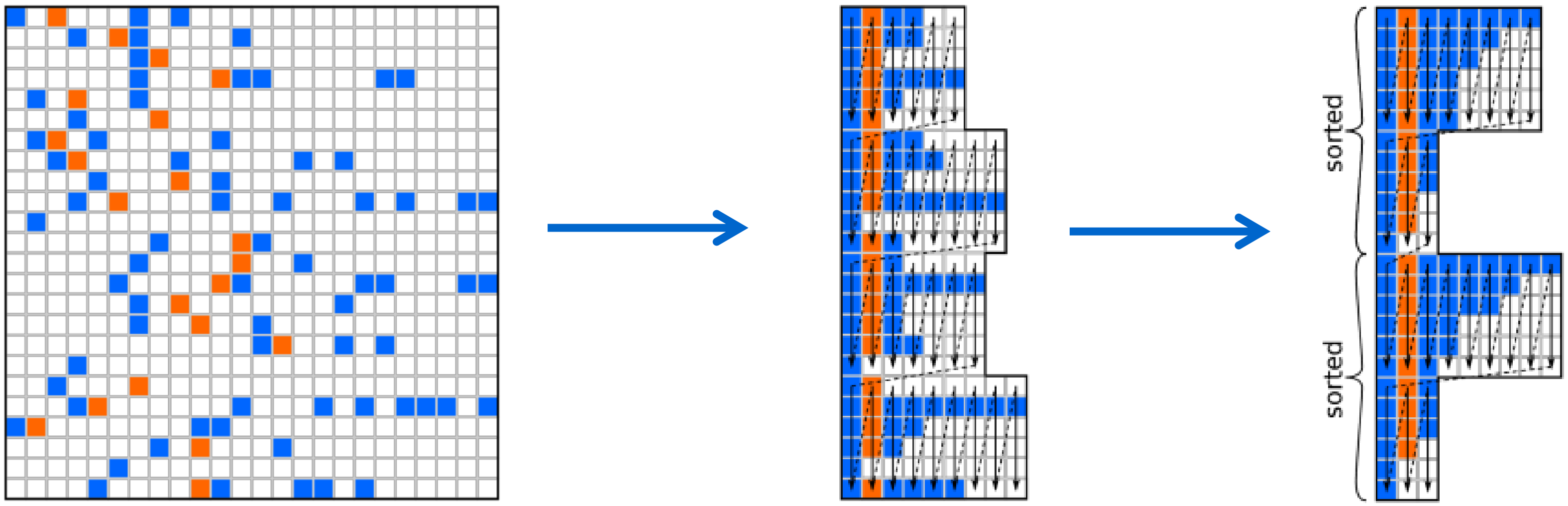
1/C ≤ β ≤ 1

β = 1/C ➔ maximum overhead
β = 1 ➔ no overhead at all
(row length is constant in a chunk)

Minimizing the storage overhead ➔ SELL-C-σ

optional

- **Sort rows within a range σ to minimize the overhead**
  - σ should not be too large in order to not worsen the RHS vector access pattern

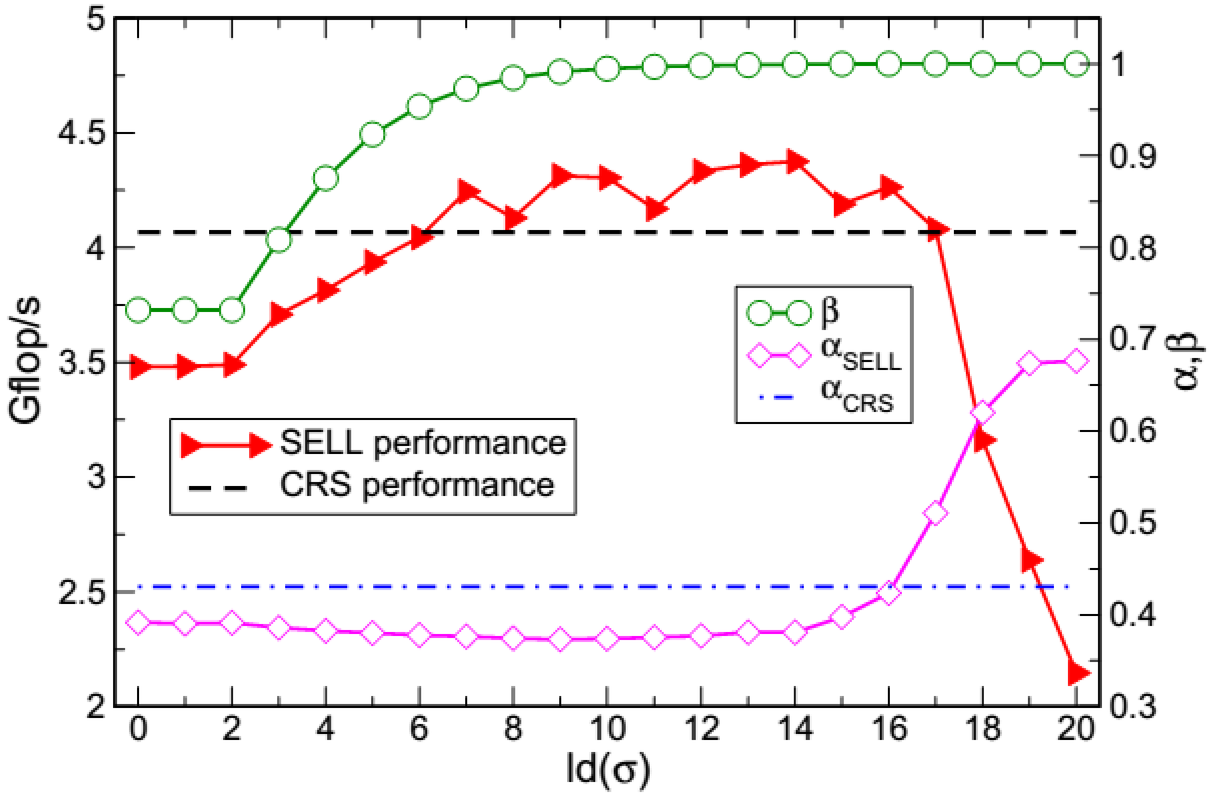


- **Two parameters:**

1. C: Chunk height
2. σ: Sorting scope

Choosing the Sorting Scope σ

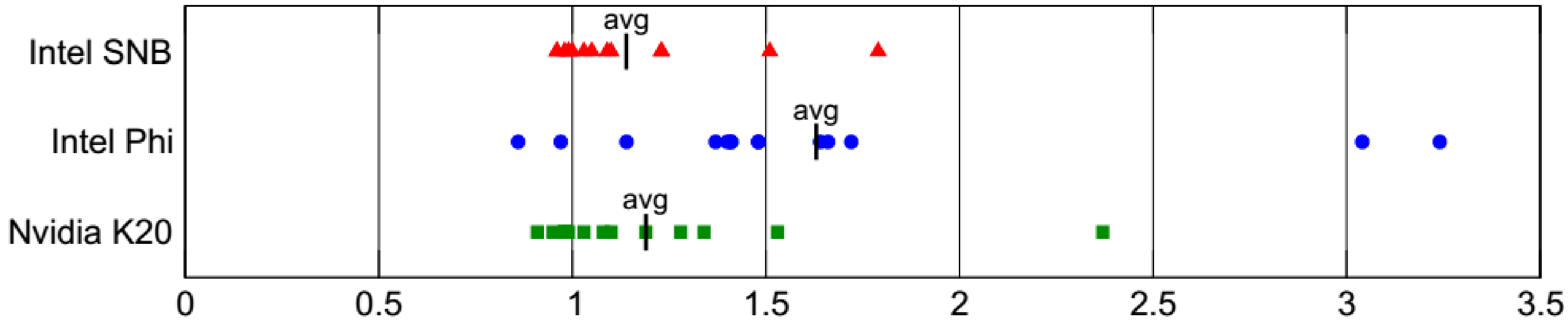
- The larger the sorting scope, the lower the storage overhead
- But what happens if the sorting scope gets too large?



SELL-C-σ Performance

optional

Using a unified storage format comes with little performance penalty in the worst case and up to a 3x performance gain in the best case for a wide range of test matrices.



Case study:
A 3D Jacobi smoother

The basics in two dimensions

Layer conditions

Validating the model

Optimization by spatial blocking

Case study:
A 3D Jacobi smoother

The basics in two dimensions

Layer conditions

Validating the model

Optimization by spatial blocking

Intel® Xeon® Processor E5-2690 v2
10 cores@3 GHz

```
L3 CacheSize          = 25 MB
Memory Bandwidth      = 48 GB/s
```

Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- but the regular access structure allows for matrix free coding

```
do iter = 1, maxit

      Perform sweep over regular grid: y ← x

      Swap y ←→ x

enddo
```

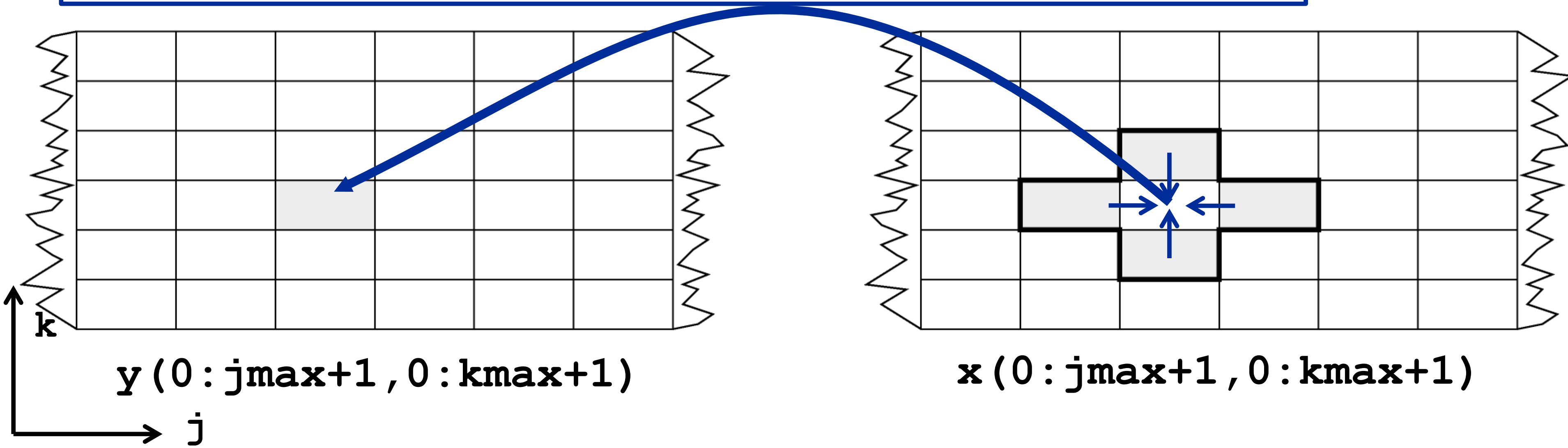
- Complexity of implementation and performance depends on
  - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type,…
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…

Jacobi-type 5-pt stencil in 2D

sweep

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (  x(j-1,k) + x(j+1,k) &
                     +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

Lattice Update (LUP)



Appropriate performance metric: “**Lattice Updates per second**” **[LUP/s]**

(here: Multiply by 4 FLOP/LUP to get FLOP/s rate)

Jacobi 5-pt stencil in 2D: data transfer analysis

LD+ST y(j,k) (incl. write allocate)

Available in cache (used 2 iterations before)

LD x(j+1,k)

SWEEP

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (  x(j-1,k) + x(j+1,k) &
                     +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

LD x(j,k-1)

LD x(j,k+1)

Naive balance (incl. write allocate):

`x( :, :)` : 3 LD +

`y( :, :)` : 1 ST+ 1LD

B_C = 40 B / LUP (assuming double precision)

Case study:
A 3D Jacobi smoother

The basics in two dimensions

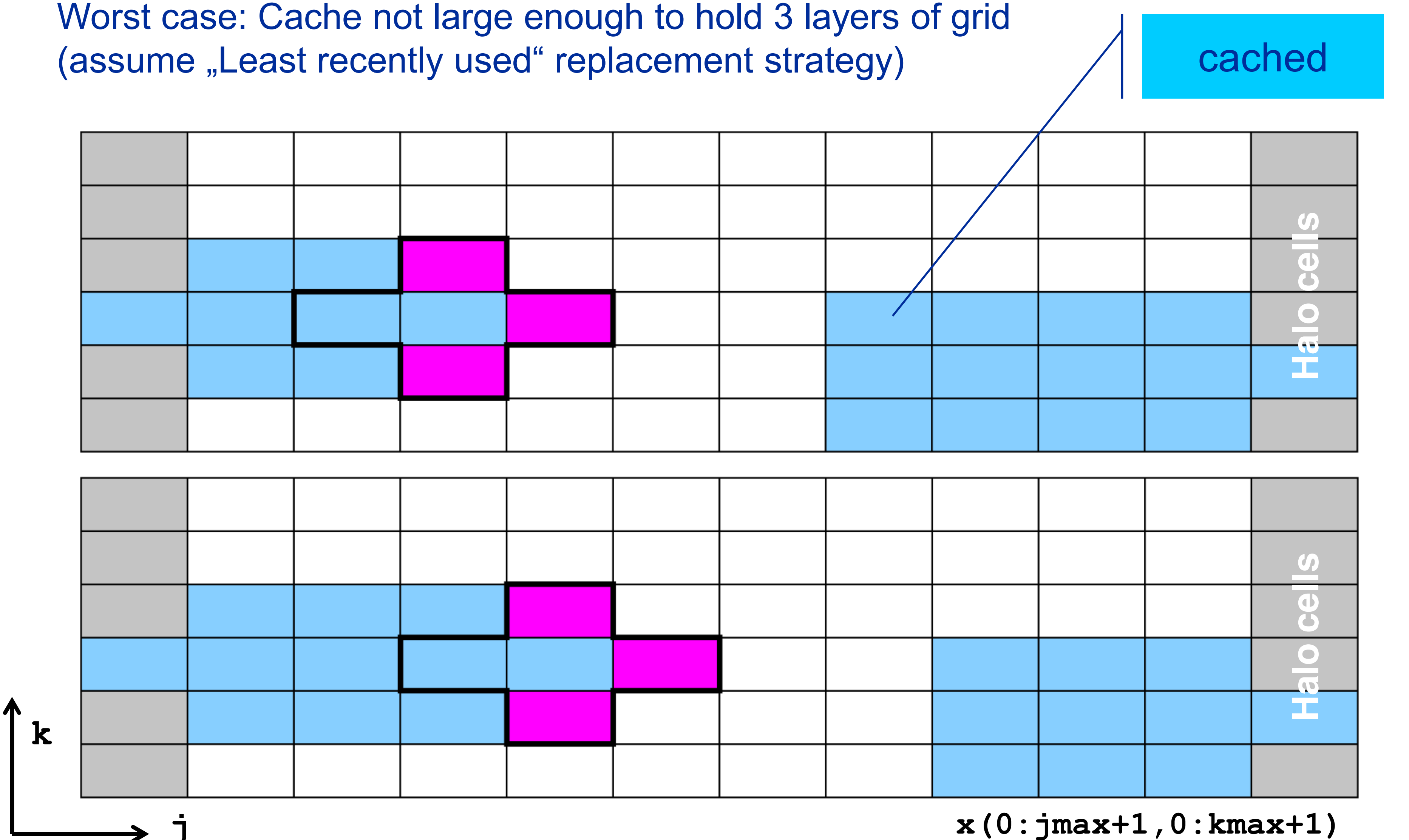
Layer conditions

Validating the model

Optimization by spatial blocking

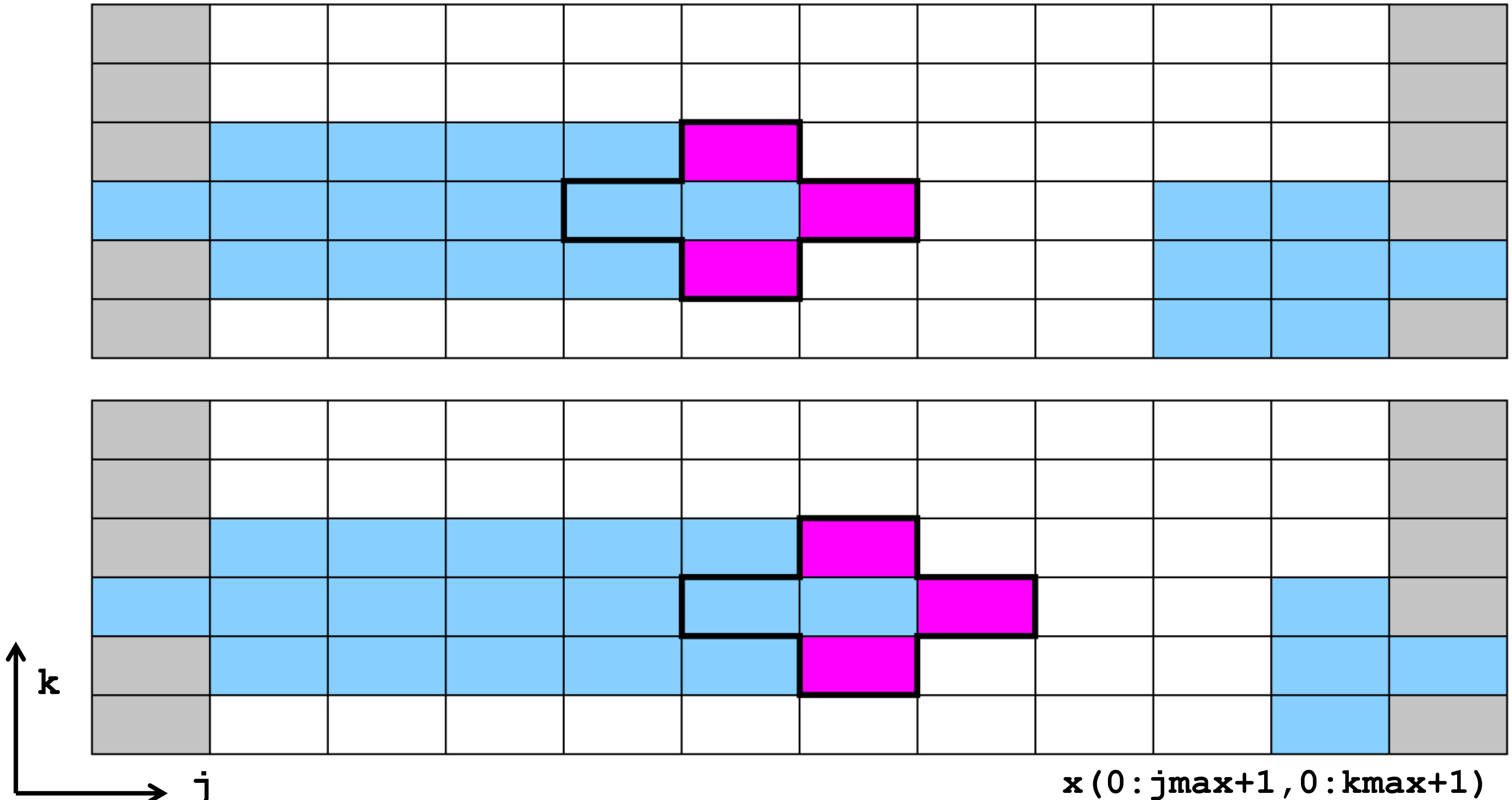
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers of grid
(assume „Least recently used“ replacement strategy)



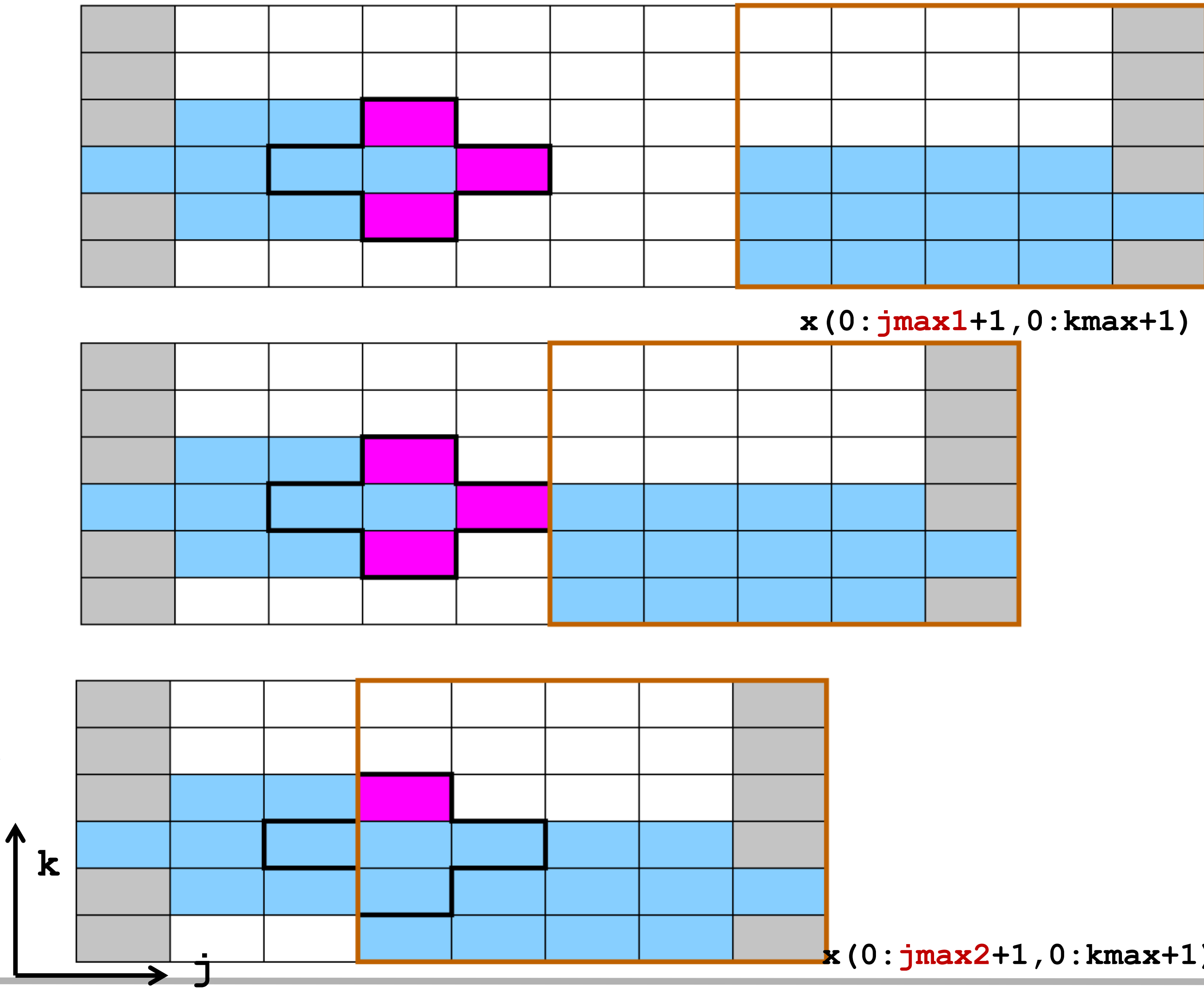
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers of grid
(+assume „Least recently used“ replacement strategy)



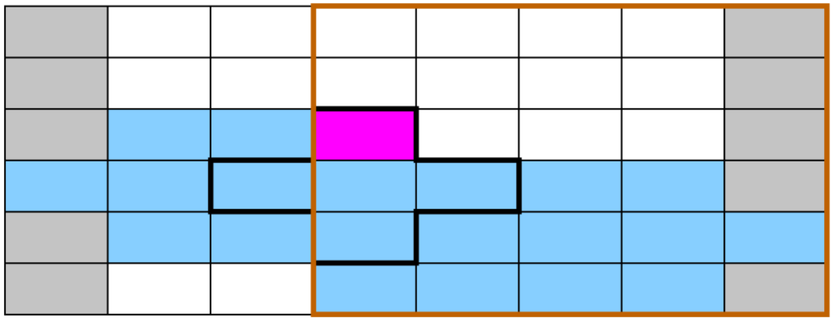
Analyzing the data flow

Reduce inner (j) loop dimension successively



Best case: 3 "layers" of grid fit into the cache!

Analyzing the data flow: Layer condition



```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                   +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

3 * jmax * 8B < CacheSize/2

"Layer condition"

3 rows of jmax | **double precision** | **Safety margin (Rule of thumb)**

Layer condition:

- No impact of outer loop length (kmax)
- No strict guideline (cache associativity – data traffic for y not included)
- Need to be adapted for other stencils, e.g. 3D 7-pt stencil

Analyzing the data flow: Layer condition

YES

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                   +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

B_C = 24 B / LUP

```
3 * jmax * 8B < CacheSize/2
```

“Layer condition” fulfilled?

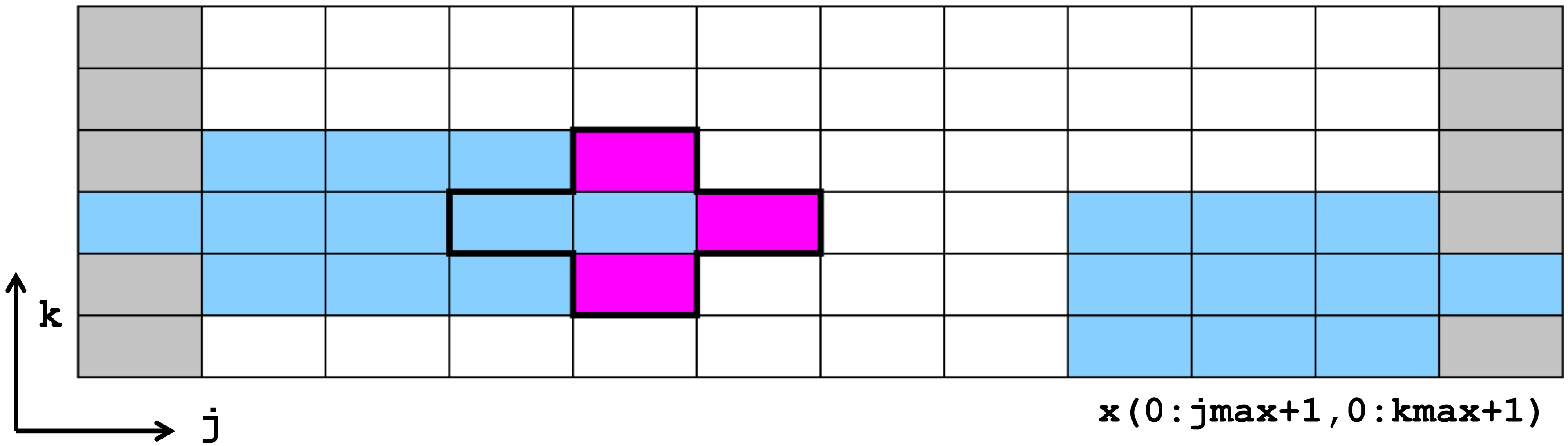
NO

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                   +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

B_C = 40 B / LUP

From 2D to 3D

2D:

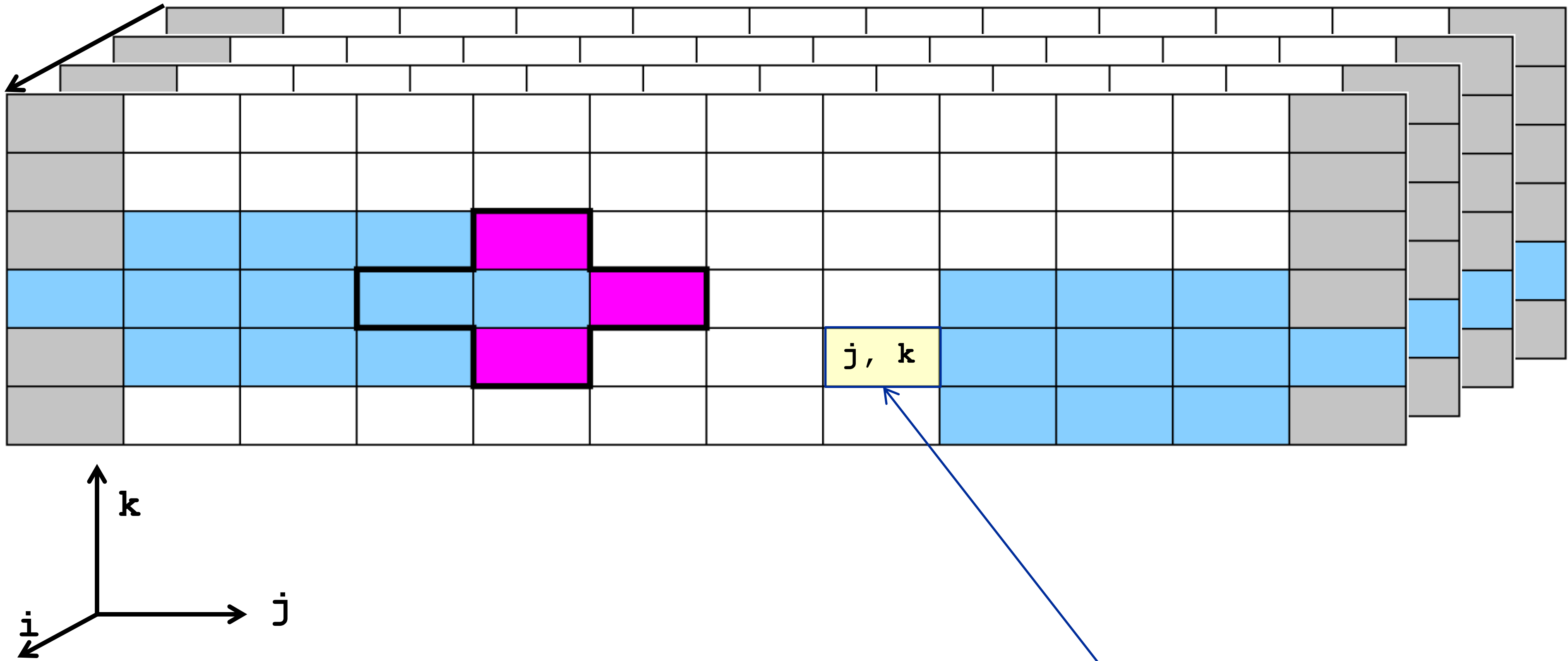


Towards 3D understanding

- Picture can be considered as 2D cut of 3D domain for (new) fixed `i`-coordinate:

`x(0:jmax+1,0:kmax+1)` → `x(i, 0:jmax+1,0:kmax+1)`

From 2D to 3D



- `x(0:imax+1, 0:jmax+1,0:kmax+1)` – **Assume i-direction contiguous in main memory (Fortran notation)**
- **Stay at 2D picture and consider one cell of `j-k` plane as a contiguous row of elements in i-direction: `x(0:imax,j,k)`**

Layer condition: From 2D 5-pt to 3D 7-pt Jacobi-type stencil

```
3 * jmax * 8B < CacheSize/2
```

```
do k=1,kmax
  do j=1,jmax
    y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                   +  x(j,k-1) + x(j,k+1) )
  enddo
enddo
```

B_C = 24 B / LUP

```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = const * (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
```

```
3 * jmax * imax * 8B < CacheSize/2
```

B_C = 24 B / LUP

3D 7-pt Jacobi stencil (sequential)

"Layer condition"
`3*jmax*imax*8B < CS/2`

**"Layer condition" OK →
5 accesses to `x()` served by cache**

```
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) =const. *(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
```

Question:
Does parallelization/multi-threading change the layer condition?

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = 1/6.    *(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
```

Equal chunks in k-direction
→ Layer condition for each thread

```
nthreads *3*jmax*imax*8B < CS/2
```

Layer condition (cubic domain; `CS = 25 MB`)
1 thread: `imax=jmax < 720` → 10 threads: `imax=jmax < 230`

Jacobi stencil – OpenMP outer loop parallelization (II)

Layer condition OK: `nthreads * 3 * jmax * imax * 8B < CS/2`

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = 1/6. *(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
```

$B_C = 24$ B / LUP

Roofline model:

$$P = b_S / B_C$$

Intel® Xeon® Processor E5-2690 v2

10 cores@3 GHz

CS = 25 MB (L3)

b_S = 48 GB/s

$$\rightarrow P = 2000\ \mathrm{MLUP/s}$$

Case study:
A 3D Jacobi smoother

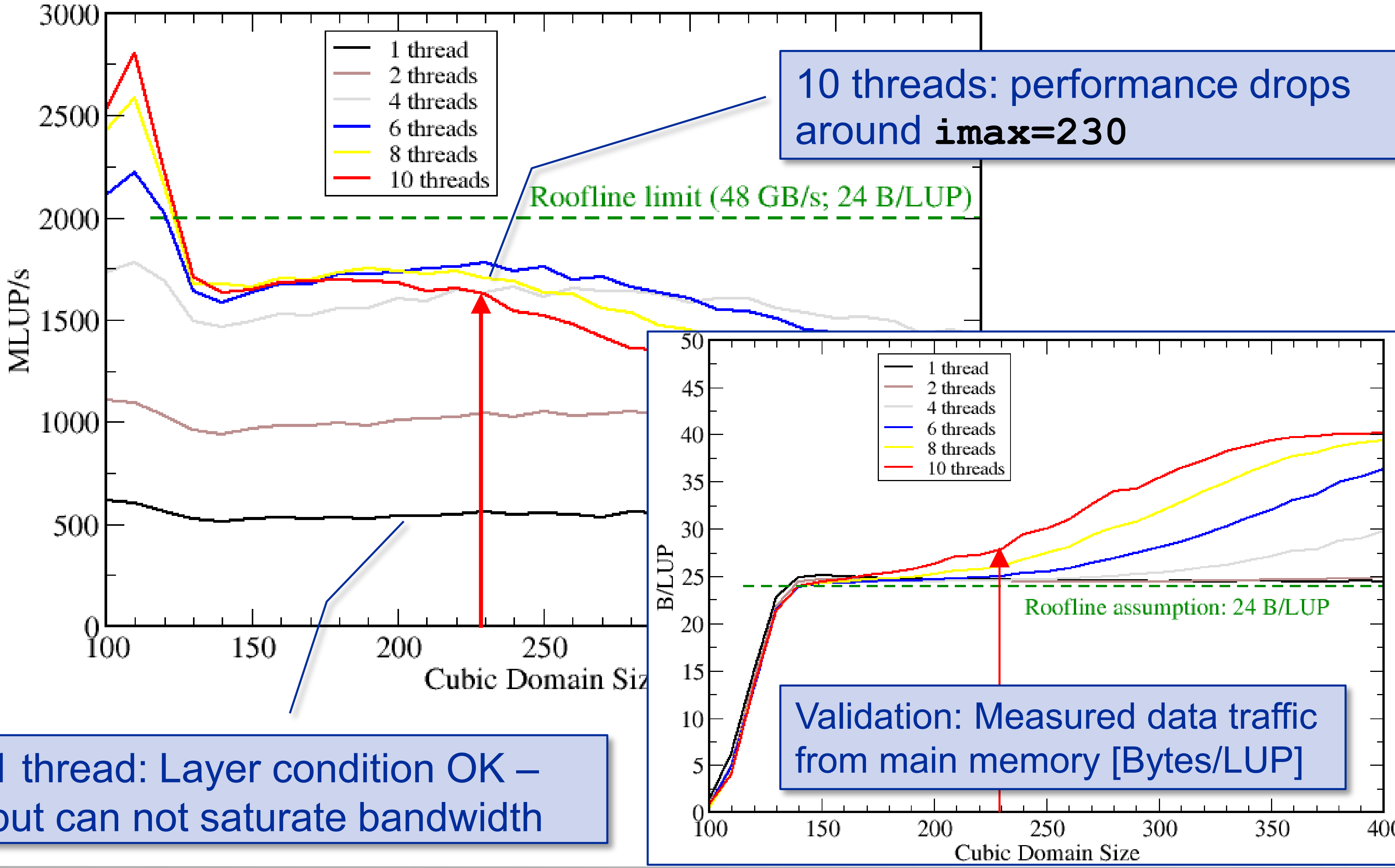
The basics in two dimensions

Layer conditions

Validating the model

Optimization by spatial blocking

3D OpenMP Jacobi Stencil – model validation



10 threads: performance drops around **`imax=230`**

1 thread: Layer condition OK – but can not saturate bandwidth

Validation: Measured data traffic from main memory [Bytes/LUP]

Jacobi Stencil – violated layer condition

Layer condition not OK: `nthreads*3*jmax*imax*8B > CS/2`

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    do i=1,imax
      y(i,j,k) = 1/6.    *(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
```

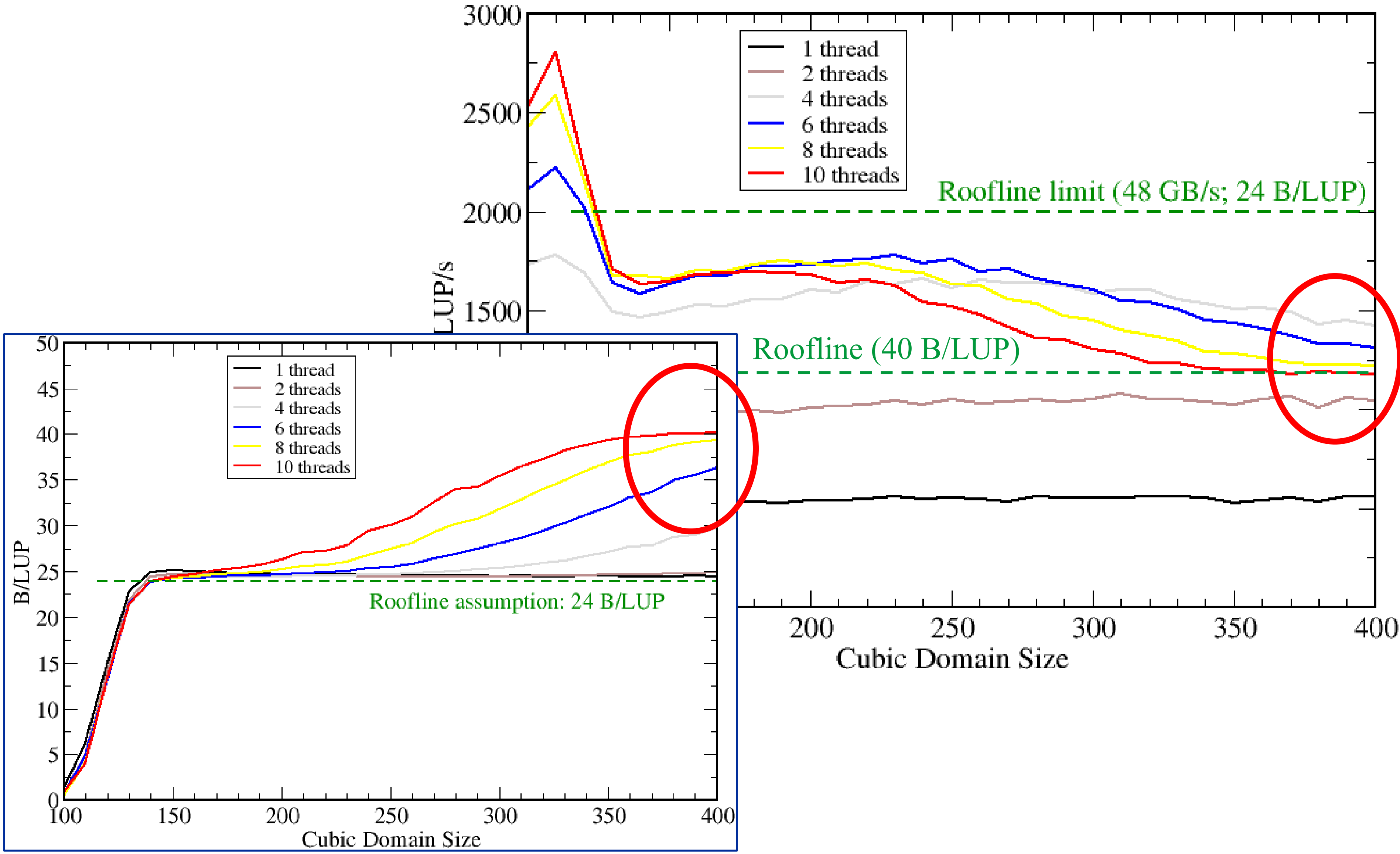
But assume: `nthreads*3*imax*8B < CS/2`

```
 (8+8) B/LUP for y() (ST+WA)
+    8 B/LUP for x(i,j,k+1)
+    8 B/LUP for x(i,j+1,k)
+    8 B/LUP for x(i,j,k-1)
```

→ $B_C = 40\ \mathrm{B/LUP}$

Roofline: $P = 1200\ \mathrm{MLUP/s}$

3D OpenMP Jacobi Stencil – model validation



Case study:
A 3D Jacobi smoother

The basics in two dimensions

Layer conditions

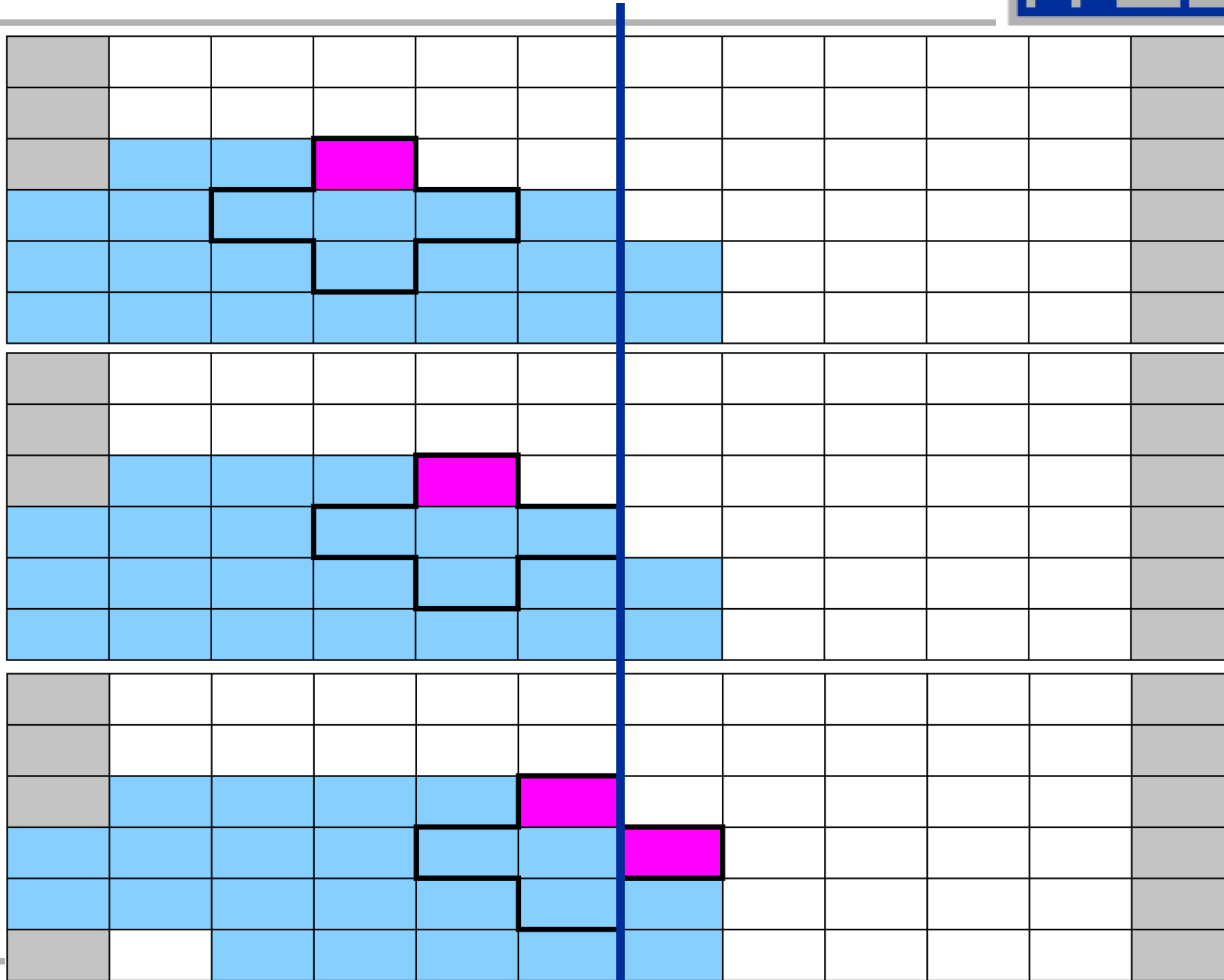
Validating the model

Optimization by spatial blocking

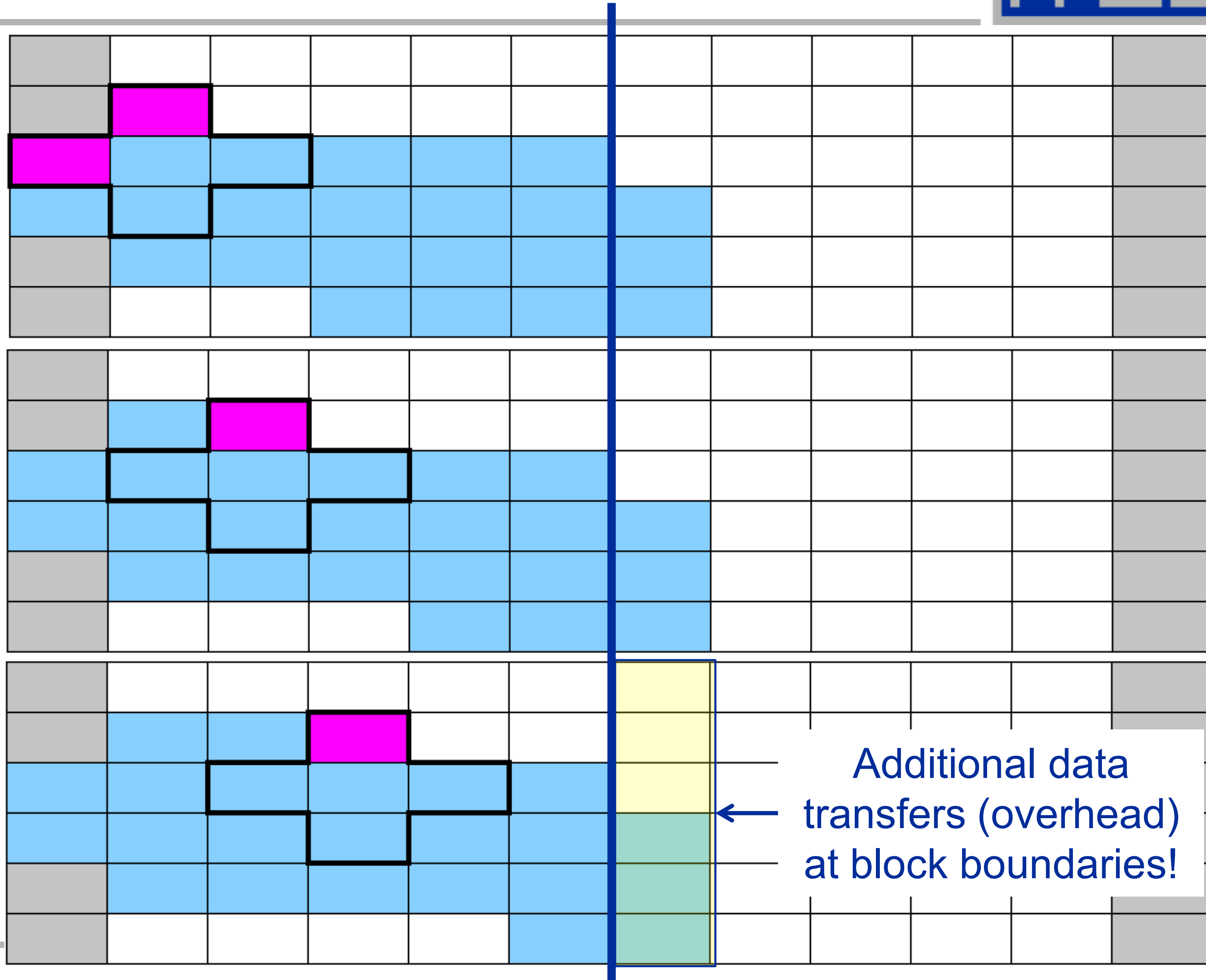
Enforcing the layer condition by blocking

Split up domain into subblocks.

e.g. block size = 5



Enforcing the layer condition by blocking



Jacobi Stencil – simple spatial blocking

```
do jb=1,jmax,jblock ! Assume jmax is multiple of jblock

!$OMP PARALLEL DO SCHEDULE(STATIC)
  do k=1,kmax
    do j=jb,(jb+jblock-1) !   Loop length jblock
      do i=1,imax
        y(i,j,k) = 1/6. *(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
enddo
```

Layer condition (j-Blocking)

`nthreads*3*jblock*imax*8B < CS/2`

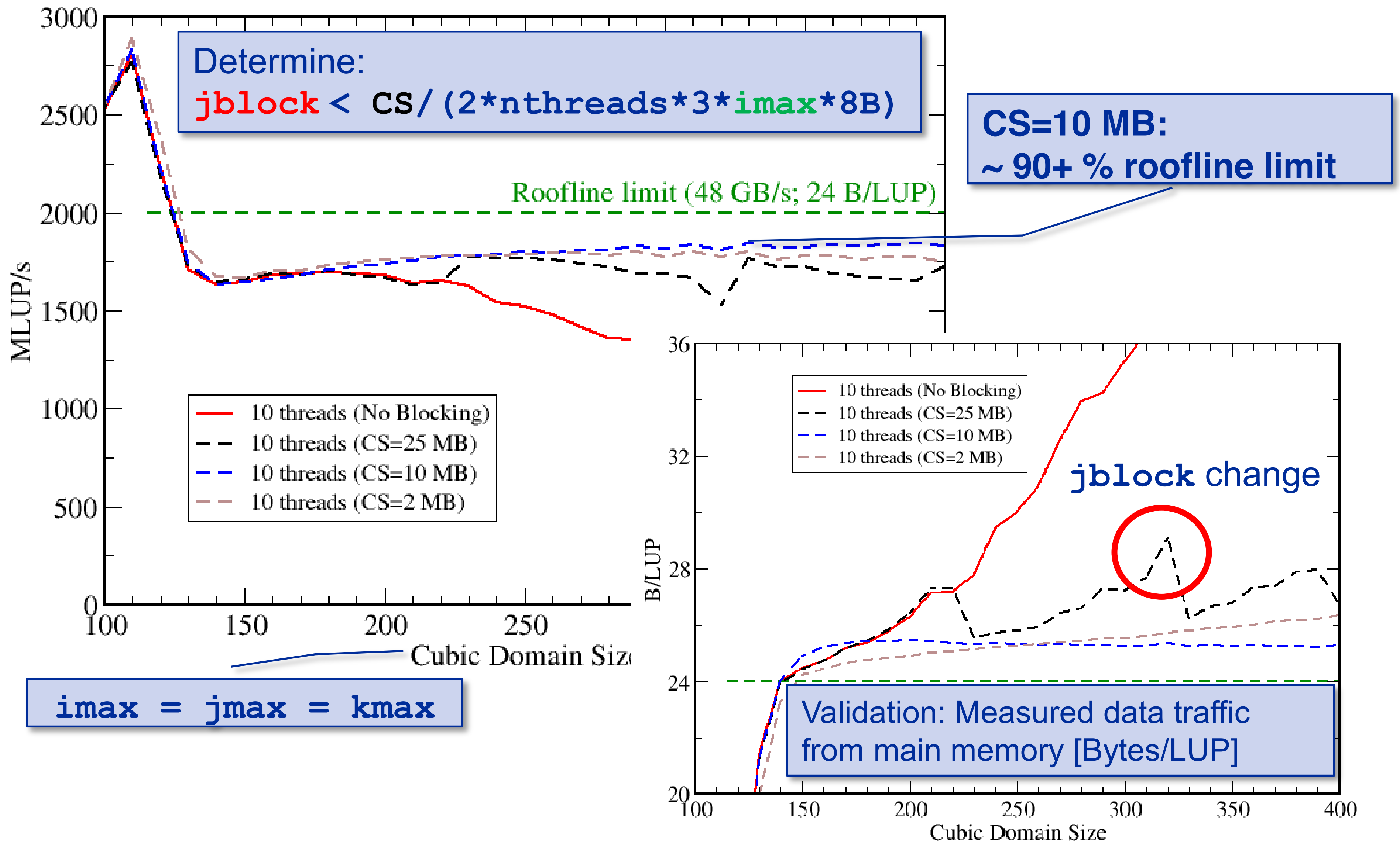
Ensure layer condition by choosing `jblock` approriately (Cubic Domains):

`jblock < CS/(imax* nthreads* 48B )`

Test system: Intel® Xeon® Processor E5-2690 v2 (10 cores / 3 GHz)

b_S = 48 GB/s, CS = 25 MB (L3) → $P = b_S/B_C = 2000\ \mathrm{MLUP/s}$

Jacobi Stencil – simple spatial blocking



Conclusions from the Jacobi example

- We have made sense of the memory-bound performance vs. problem size
  - "Layer conditions" lead to predictions of code balance
  - Achievable memory bandwidth is input parameter
- "What part of the data comes from where" is a crucial question
- The model works only if the bandwidth is "saturated"
  - In-cache modeling is more involved

- Avoiding slow data paths == re-establishing the most favorable layer condition

- Improved code showed the speedup predicted by the model
- Optimal blocking factor can be estimated
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of "optimization space"