NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Performance Engineering Case Studies

Analytic performance modeling and its use

in scientific computing

Georg Hager, NHR@FAU

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

PE Case Study: A Jacobi Smoother

The basics in two dimensions

Stencil schemes

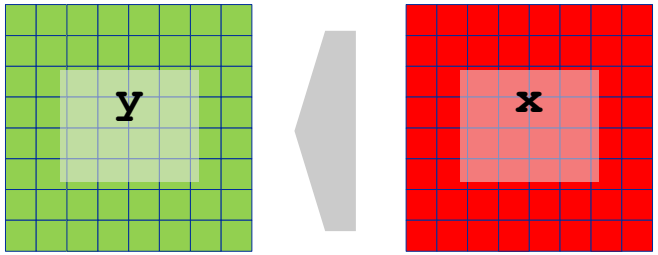
- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- … but the regular access structure allows for matrix-free coding

```
do iter = 1, max_iterations

  Perform sweep over regular grid: y(:) ← x(:)

  Swap y ←→ x

enddo
```



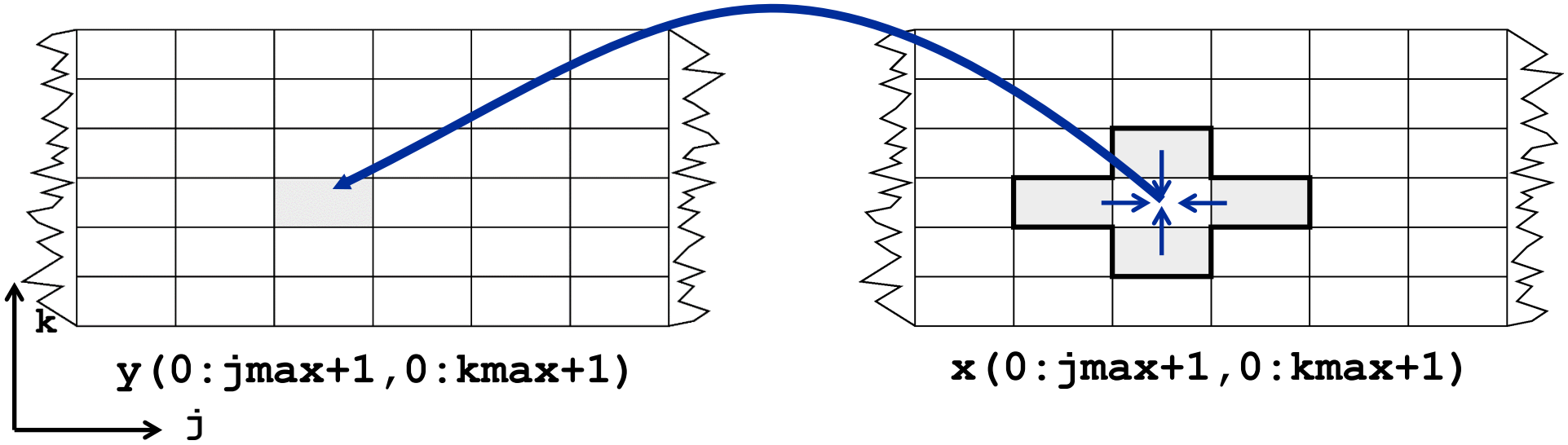
- Complexity of implementation and performance depends on
  - stencil operator, e.g. Jacobi-type, Gauss-Seidel-type, …
  - discretization, e.g. 7-pt or 27-pt in 3D,…

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
```

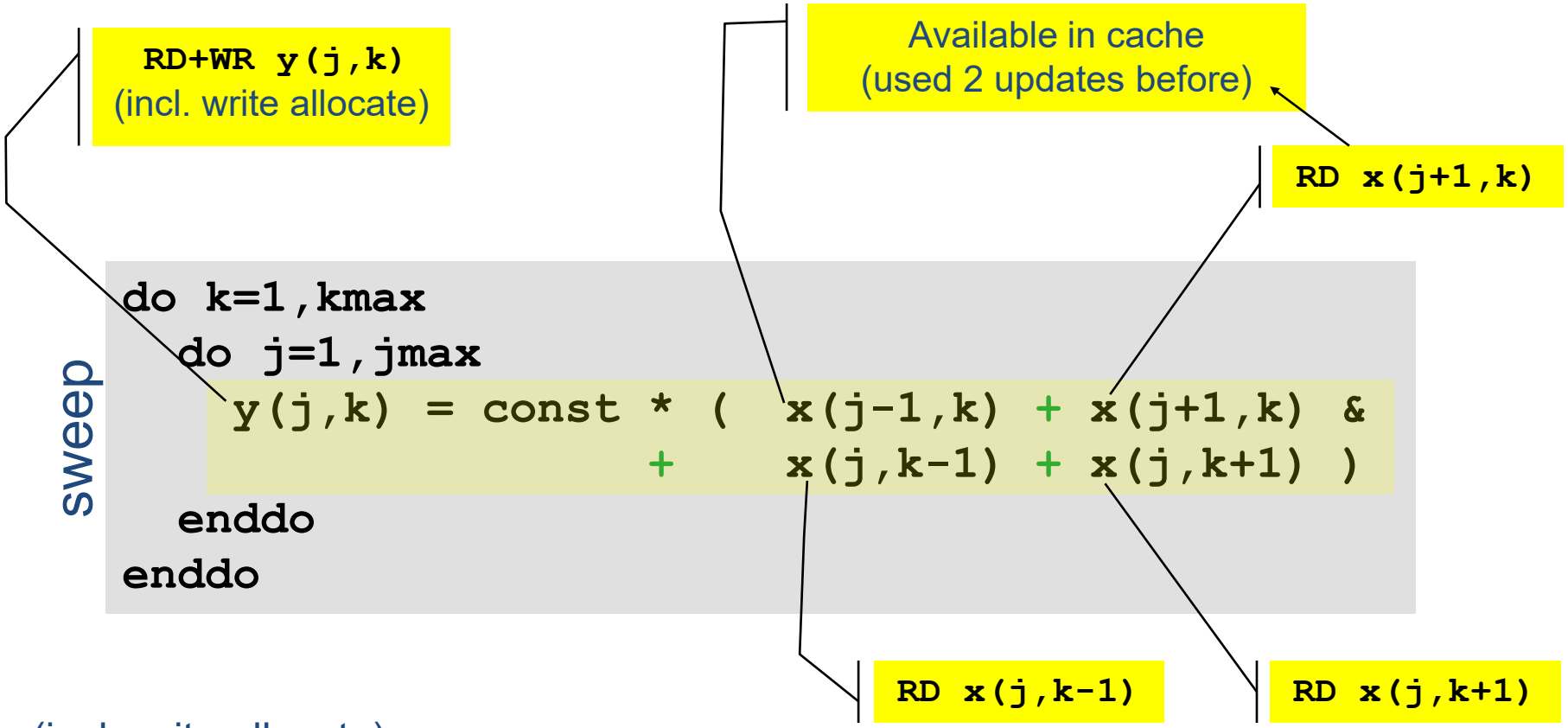
Lattice site update (LUP)



Appropriate performance metric: "Lattice site updates per second" [LUP/s]

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

Jacobi 5-pt stencil 2D: data transfer analysis

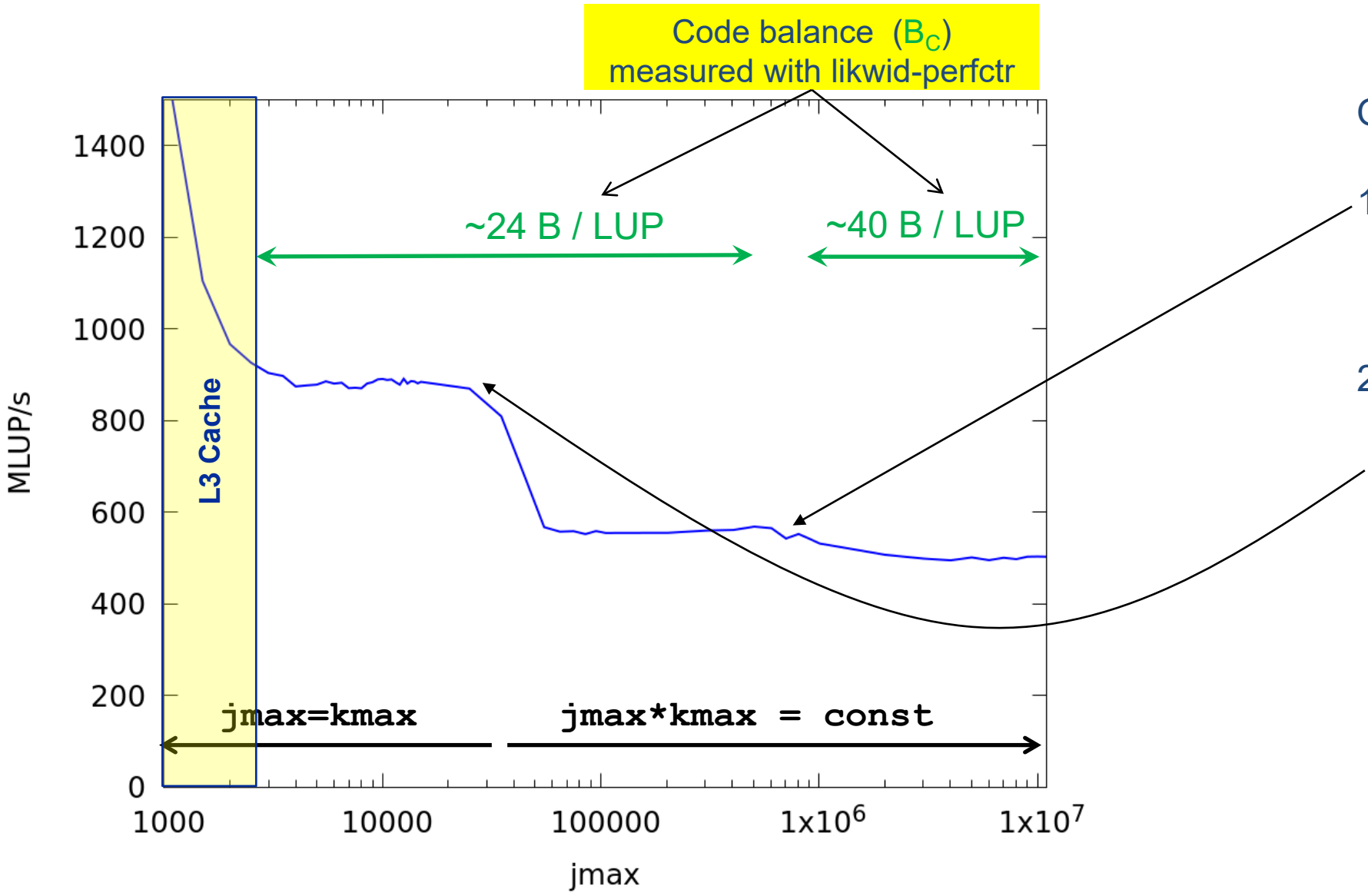


Naive balance (incl. write allocate):

`x( :, :)` : 3 RD +
`y( :, :)` : 1 WR+ 1 RD

→ B_C = 5 Words / LUP = 40 B / LUP (assuming double precision)

Jacobi 5-pt stencil 2D: Single core performance



Questions:

1. How to achieve 24 B/LUP also for large **jmax**?
2. How to sustain >800 MLUP/s for **jmax > 10^4** ?

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

NHR FAU

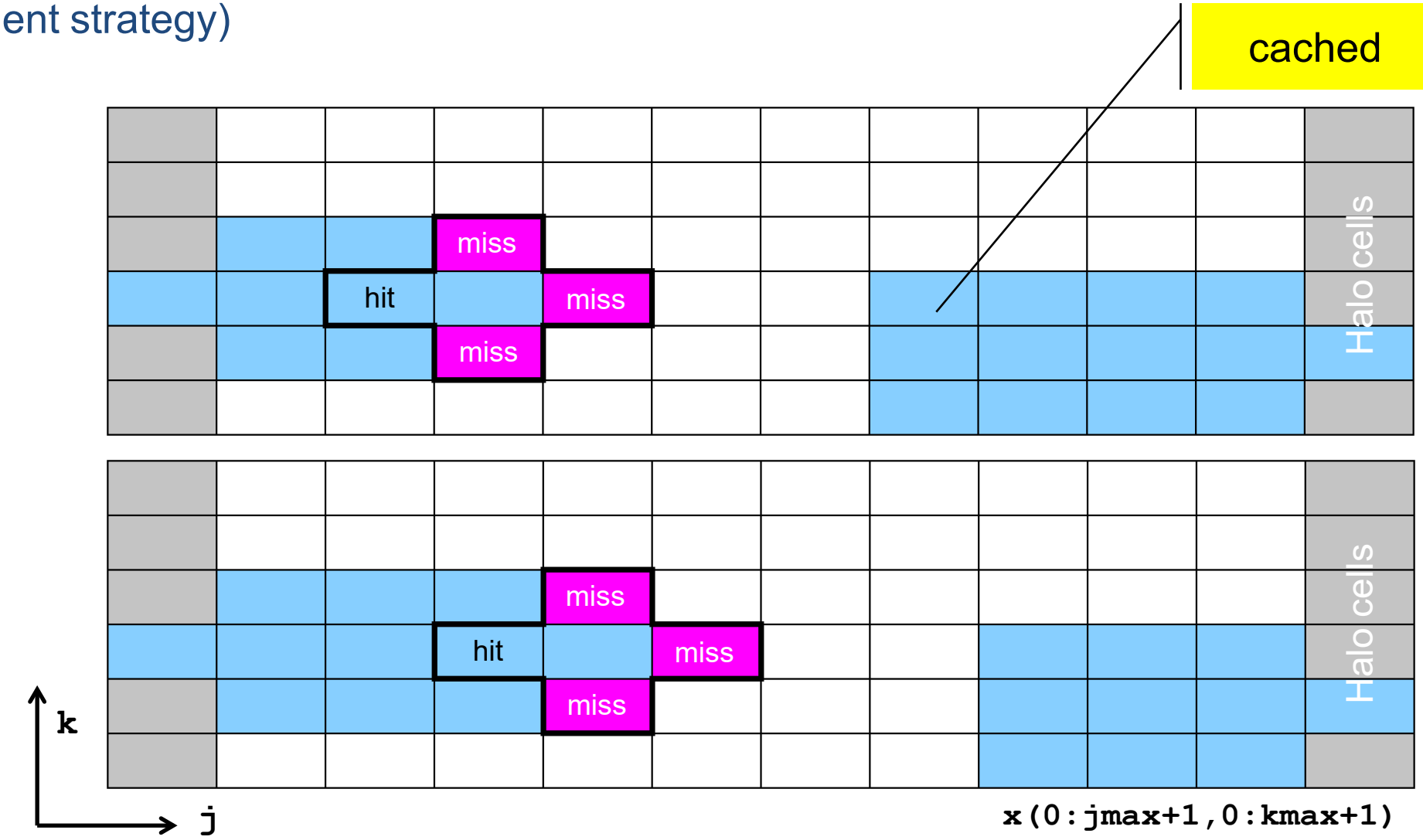
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: A Jacobi smoother

Layer conditions

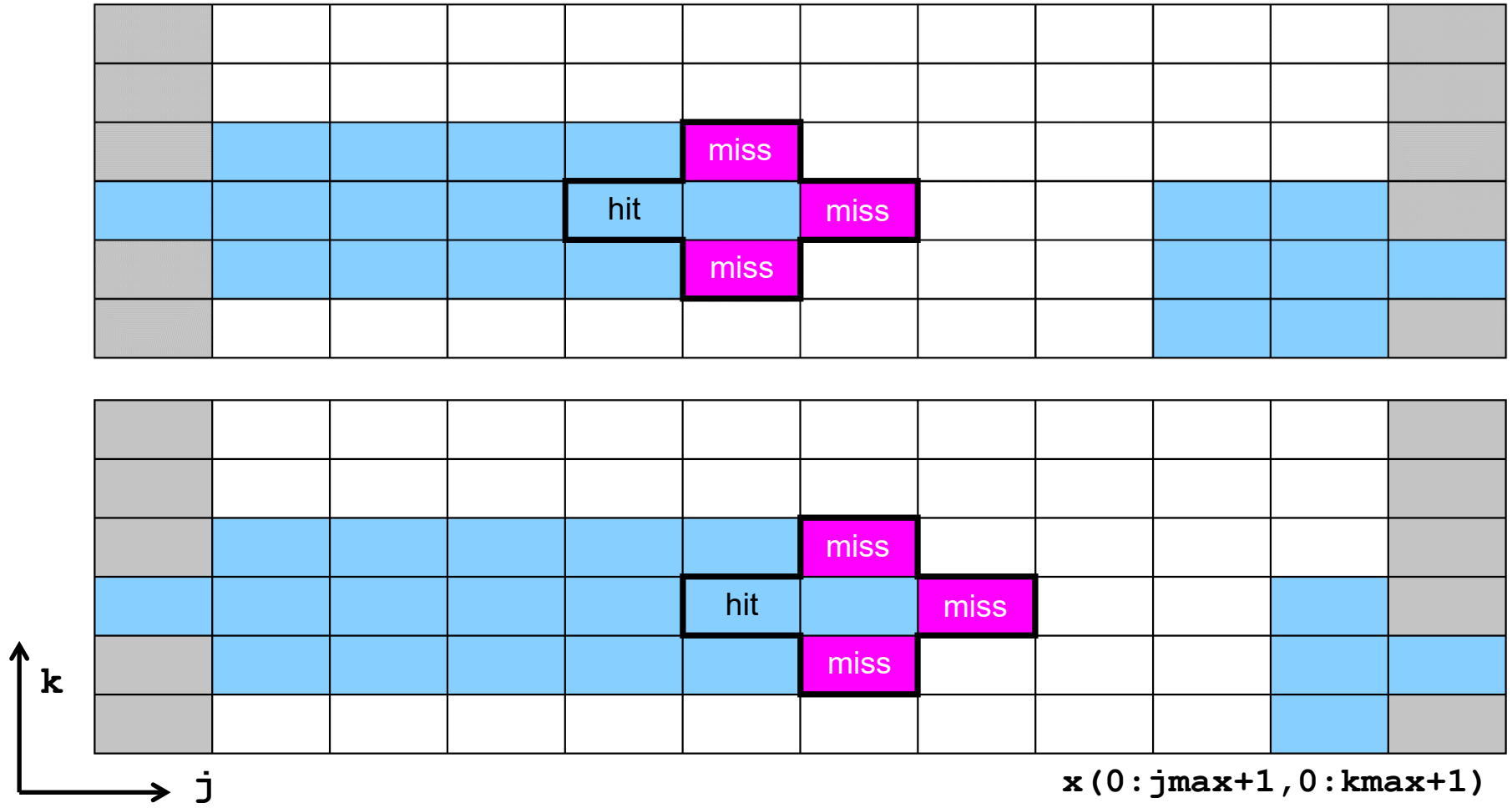
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume “Least Recently Used” replacement strategy)



Analyzing the data flow

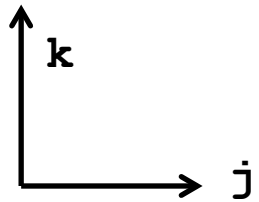
Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume „Least Recently Used“ replacement strategy)

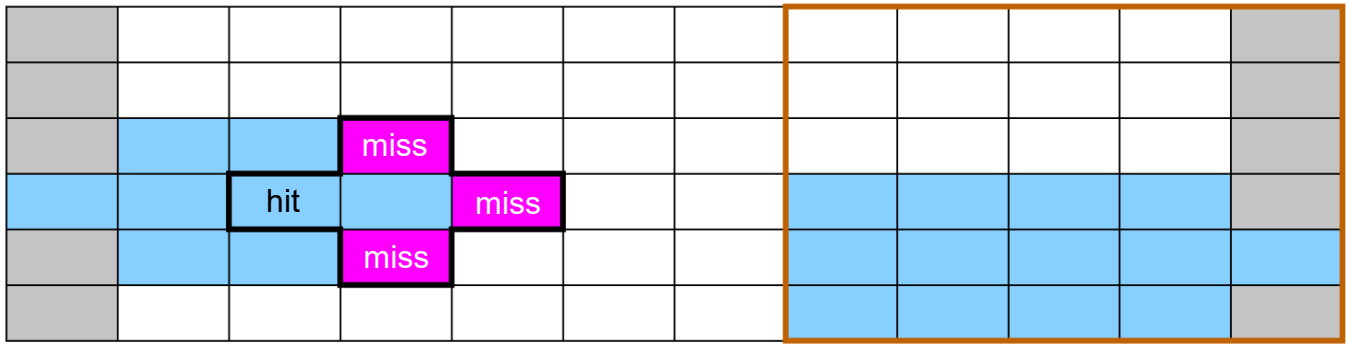


Analyzing the data flow

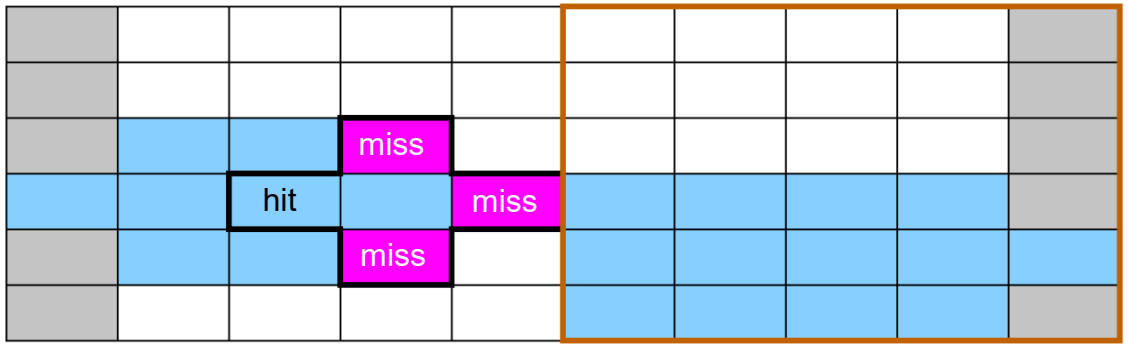
Reduce inner (j-) loop dimension successively

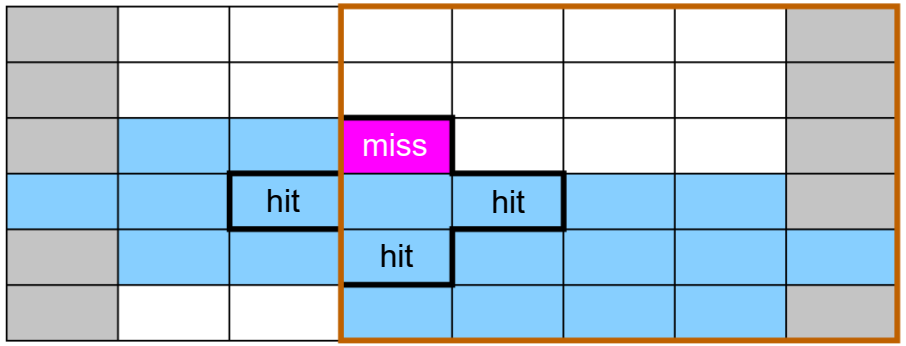
Best case: 3 “layers” of grid fit into the cache!





`x(0:jmax1+1,0:kmax+1)`

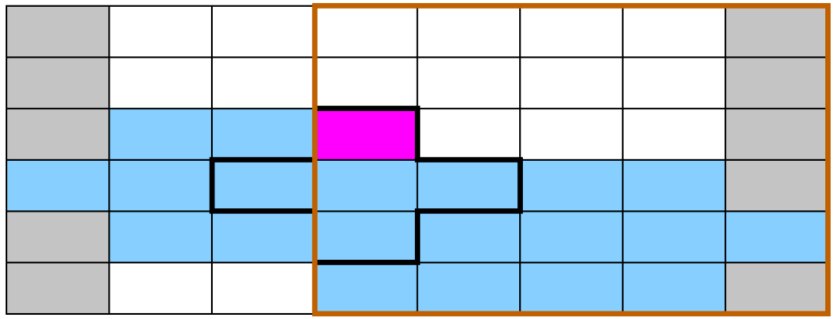




`x(0:jmax2+1,0:kmax+1)`

Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil, cache size C



```
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
```

$$3 \times \mathbf{jmax} \times 8 \text{ byte} < \frac{C}{2}$$

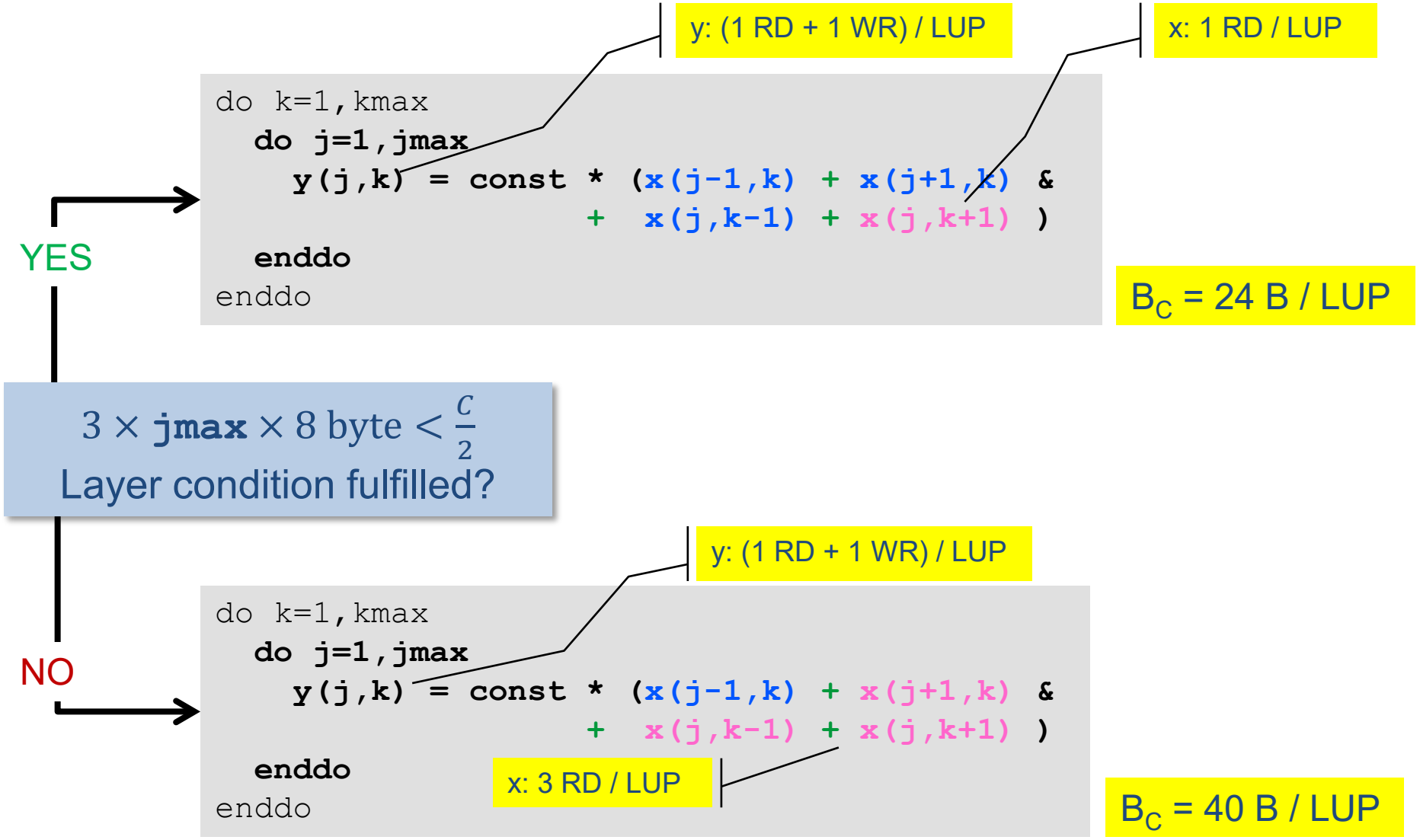
"Layer condition"

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

Layer condition:

- Does not depend on outer loop length (**kmax**)
- No strict guideline (cache associativity, data traffic for y not included)
- Needs to be adapted for other stencils (e.g., long-range stencils)

Analyzing the data flow: Layer condition (2D 5-pt Jacobi)



NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: A Jacobi smoother

Optimization by spatial blocking

Enforcing a layer condition (2D 5-pt Jacobi)

- How can we enforce a layer condition for all domain sizes ?
- Idea: Spatial blocking
  - Reuse elements of `x()` as long as they stay in cache
  - Sweep can be executed in any order, e.g., compute blocks in j direction

"Spatial Blocking" of j loop:

```
do jb=1,jmax,jblock !
  do k=1,kmax
    do j= jb, min(jb+jblock-1,jmax) !inner loop length jblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      +  x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

New layer condition (blocking)

$$3 \times \mathbf{jblock} \times 8\ \text{byte} < \frac{C}{2}$$

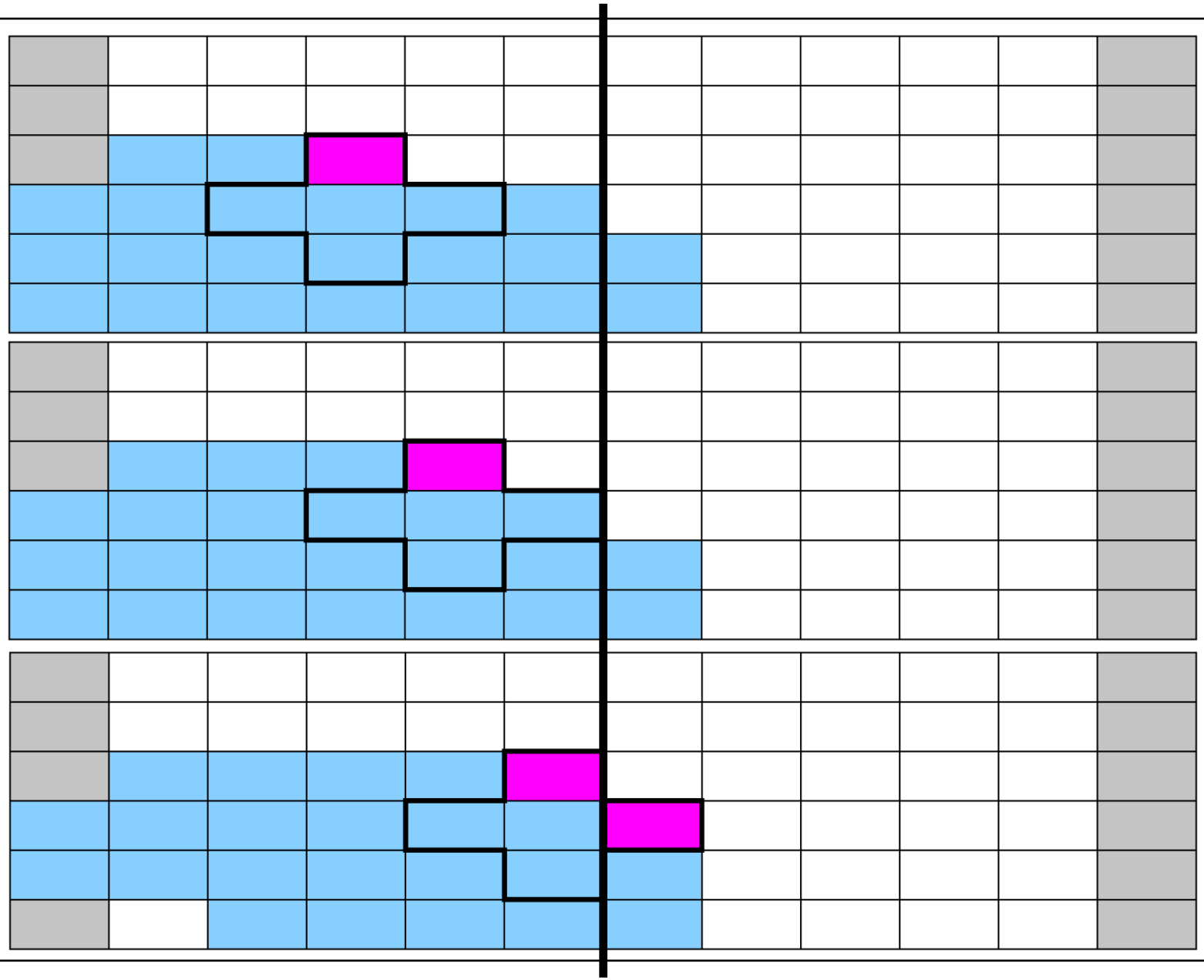
Determine for given C an appropriate `jblock` value:

$$\mathbf{jblock} < \frac{C}{48\ \text{byte}}$$

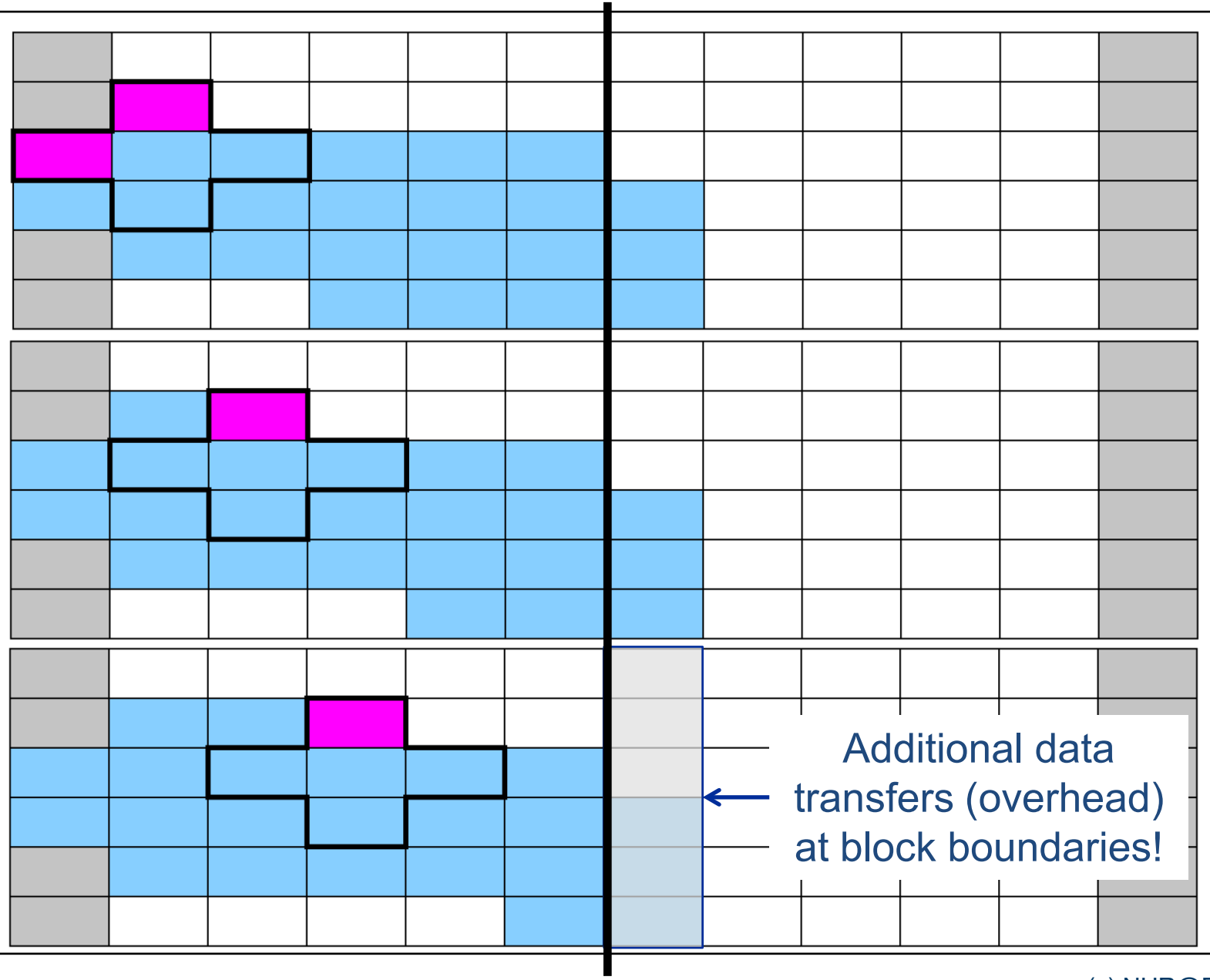
Establish the layer condition by blocking

Split domain into subblocks:

e.g. block size = 5



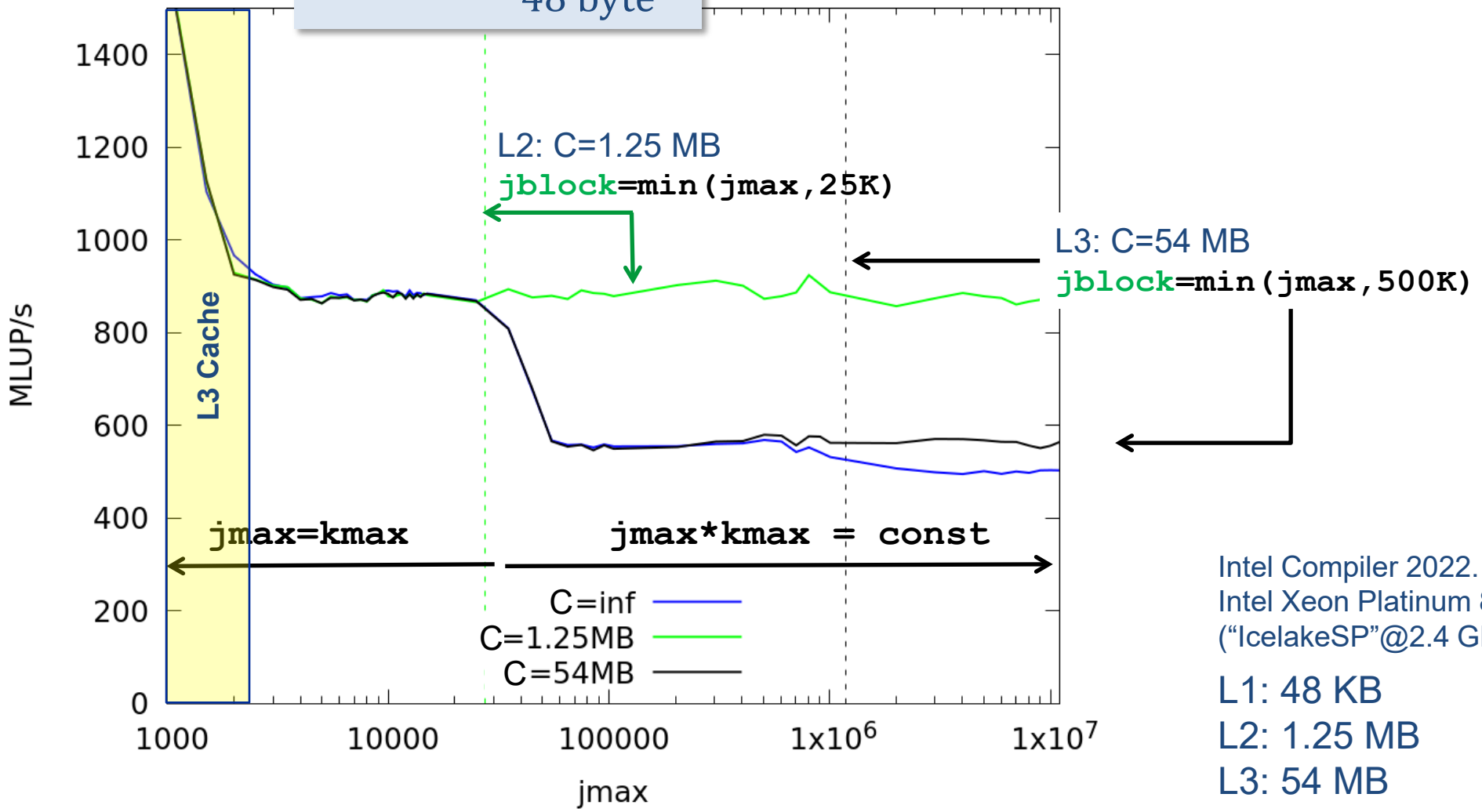
Establish the layer condition by blocking



Establish layer condition by spatial blocking

$$\texttt{jblock} < \frac{C}{48\ \text{byte}}$$

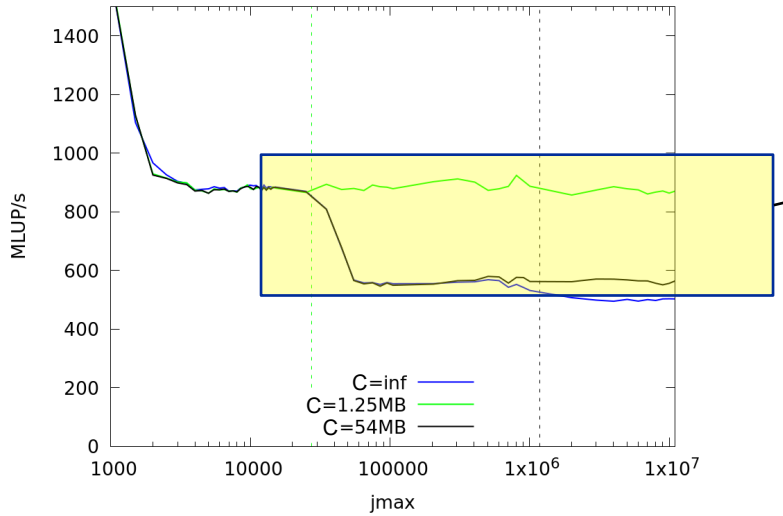
→ Which cache to block for?



Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

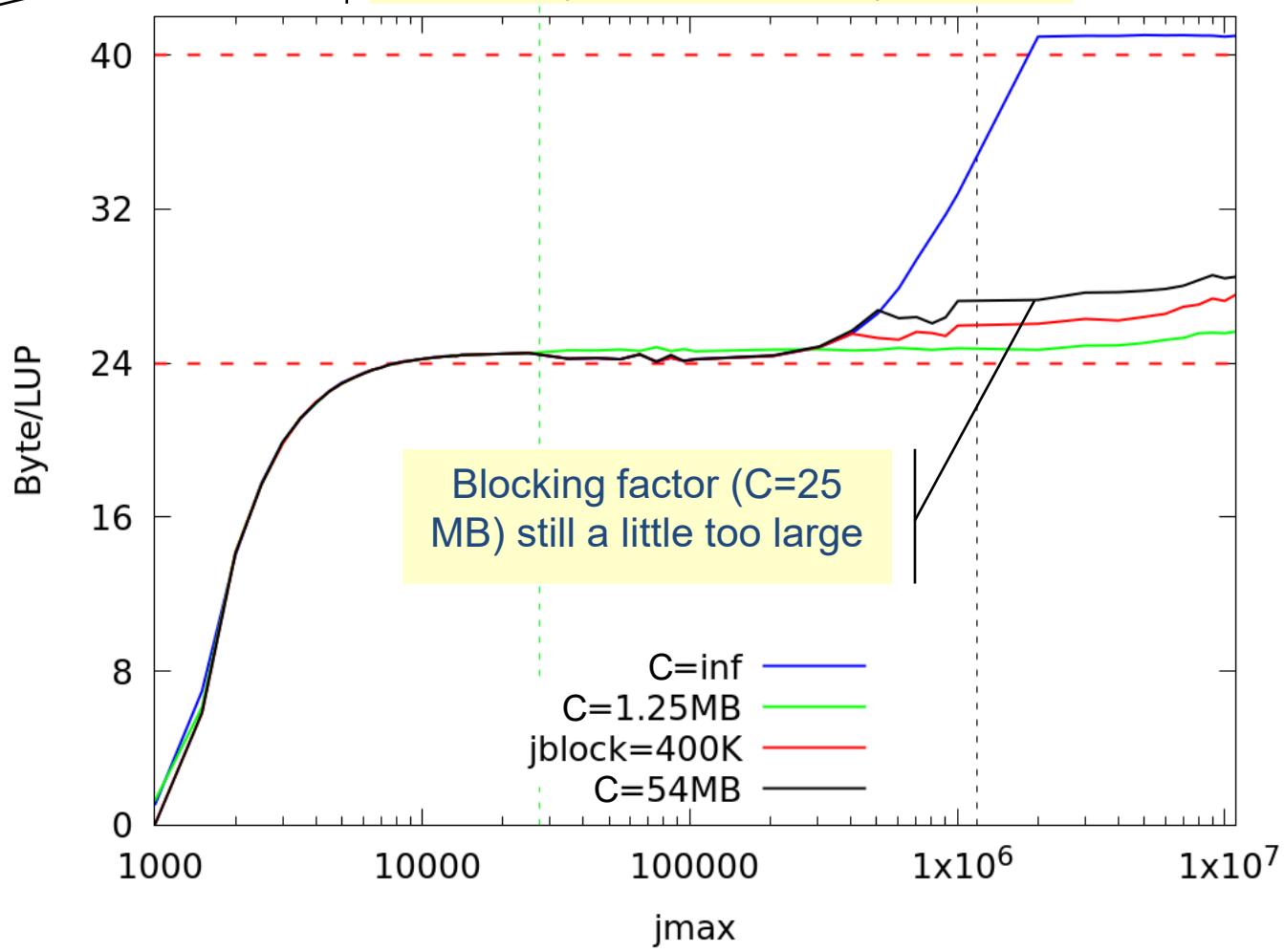
L1: 48 KB
L2: 1.25 MB
L3: 54 MB

Validating the model: Memory code balance



Main memory access is not reason for different performance (but L3 access is!)

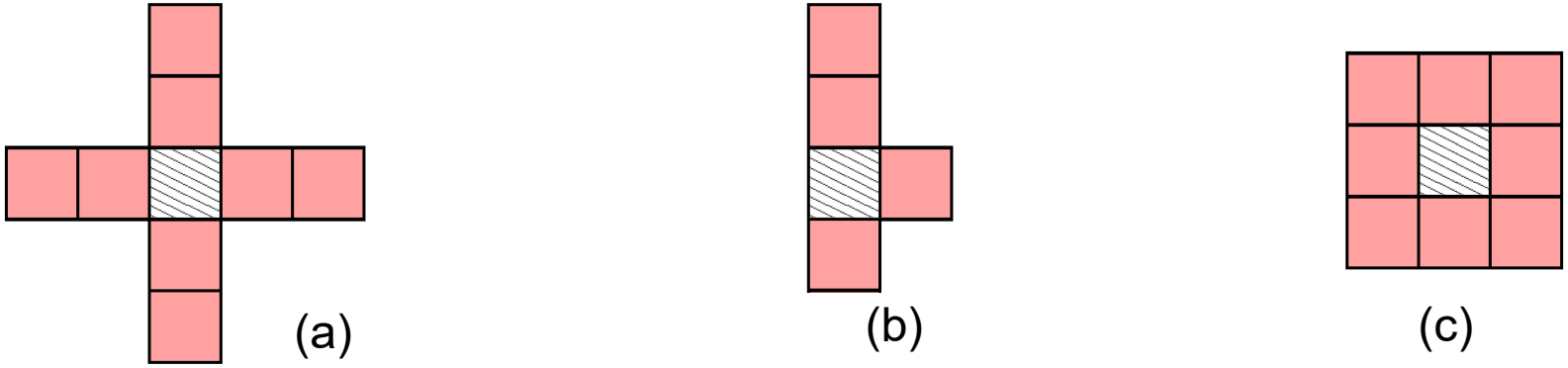
Measured main memory code balance (BC)



Blocking factor (C=25 MB) still a little too large

Intel Compiler 2022.1.0
Intel Xeon Platinum 8360Y
("IcelakeSP"@2.4 GHz)

Stencil shapes and layer conditions in 2D



a) Long-range $r = 2$: 5 layers $(2r + 1)$

b) Asymmetric: 4 layers

c) 2D box: 3 layers

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

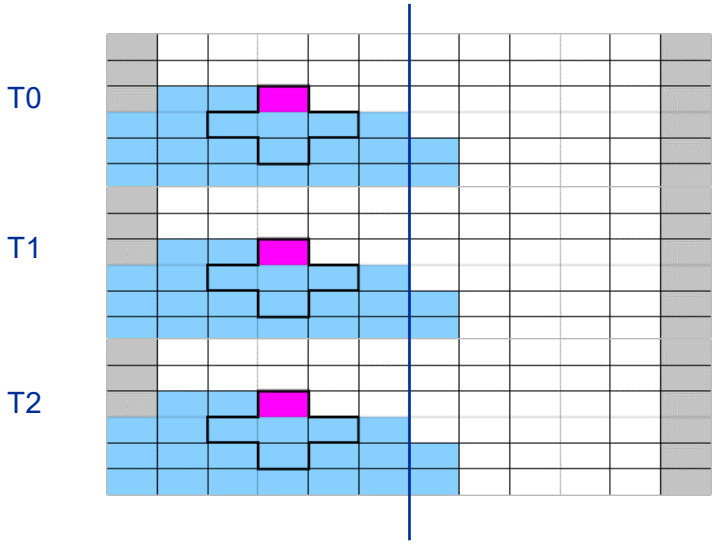
Case study: A Jacobi smoother

OpenMP parallelization

OpenMP parallelization of the blocked 2D stencil

Straightforward OpenMP work sharing:

```
do jb=1,jmax,jblock
!$OMP PARALLEL DO SCHEDULE(static)
  do k=1,kmax
    do j= jb, min(jb+jblock-1,jmax)
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                      +  x(j,k-1) + x(j,k+1) )
    enddo
  enddo
!$OMP END PARALLEL DO
enddo
```



Caveat: LC must be fulfilled per thread → shared cache causes smaller blocks!

Layer condition:

$$3 \times \texttt{jblock} \times 8 \text{ byte} < \frac{C_t}{2}$$

Cache size available per thread

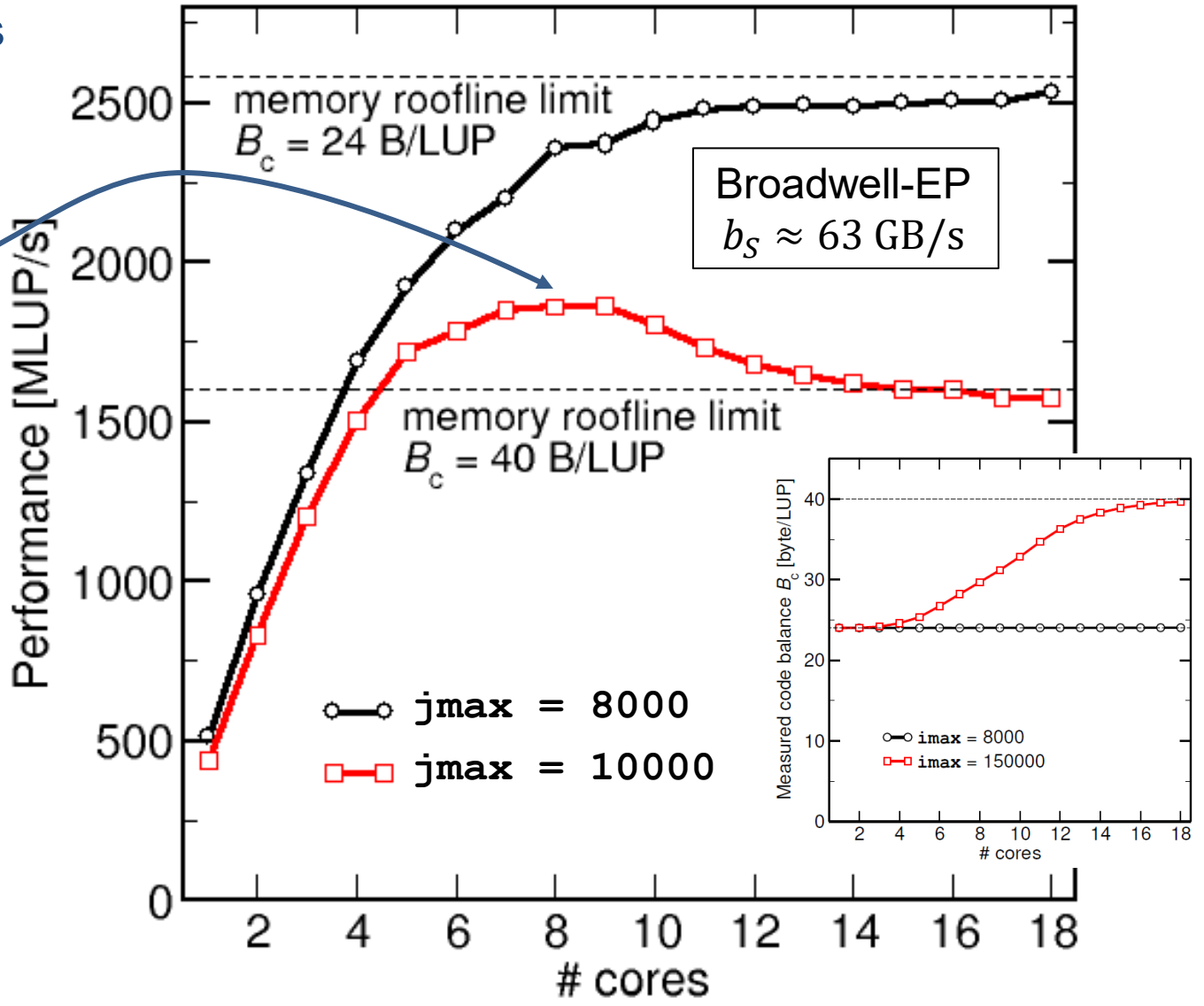
OpenMP parallelization and blocking for a shared cache

Layer conditions make for interesting effects

- Less and less shared cache available per thread as #threads goes up
- LC may break "along the way"

- Solutions
  1. Choose small enough block or domain size
     - Layers either small enough to fit in core-private caches *or*
     - Shared cache big enough to hold all layers for all threads
  2. Adaptive blocking for shared cache:

$$\texttt{jblock} = \frac{C}{\#threads \times 48\ \text{byte}}$$



NHR FAU

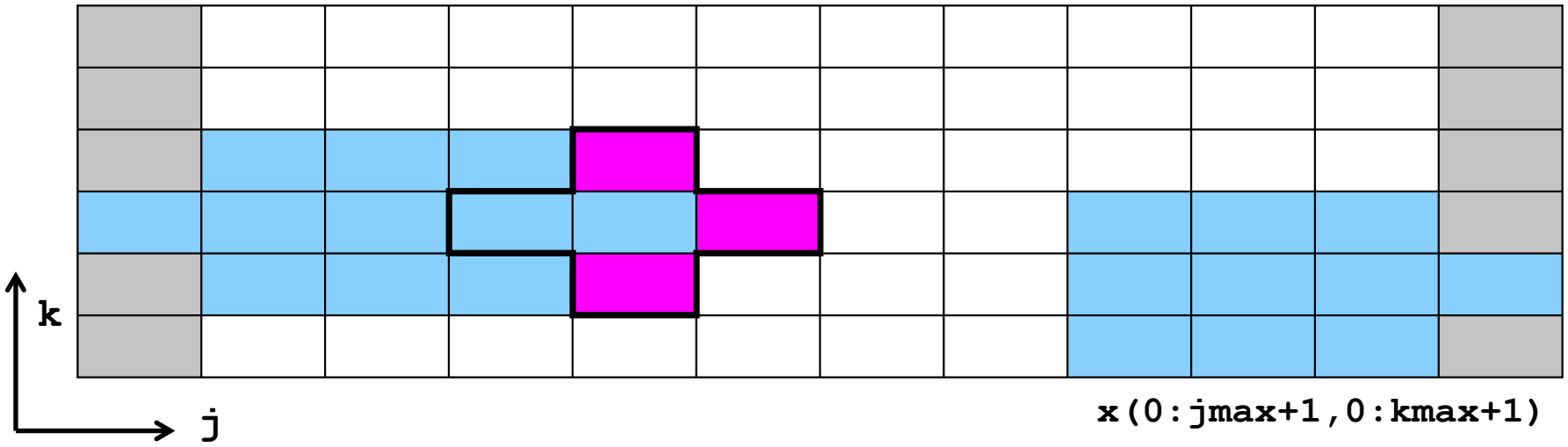
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: A Jacobi smoother

From 2D to 3D

From 2D to 3D

2D:

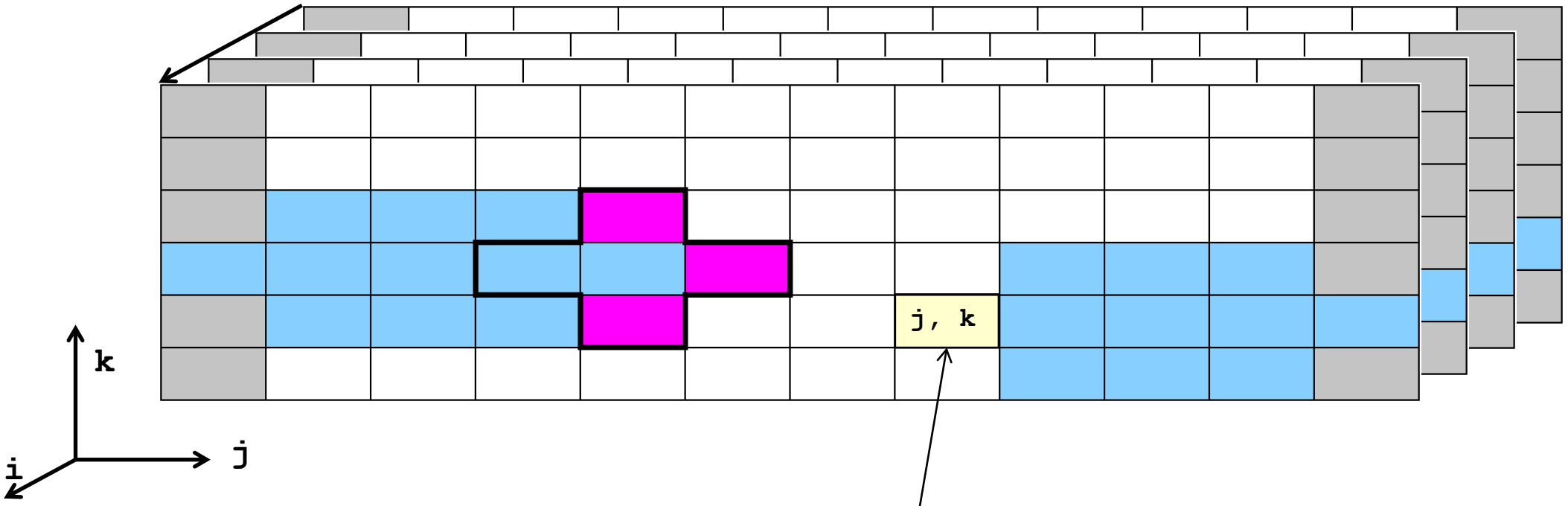


Towards 3D understanding

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

`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 slab of elements in `i` direction: `x(0:imax,j,k)`

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

```
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
```

2D

$$3 \times \mathbf{jmax} \times 8 \text{ byte} < \frac{C}{2}$$

Optimal 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
```

3D

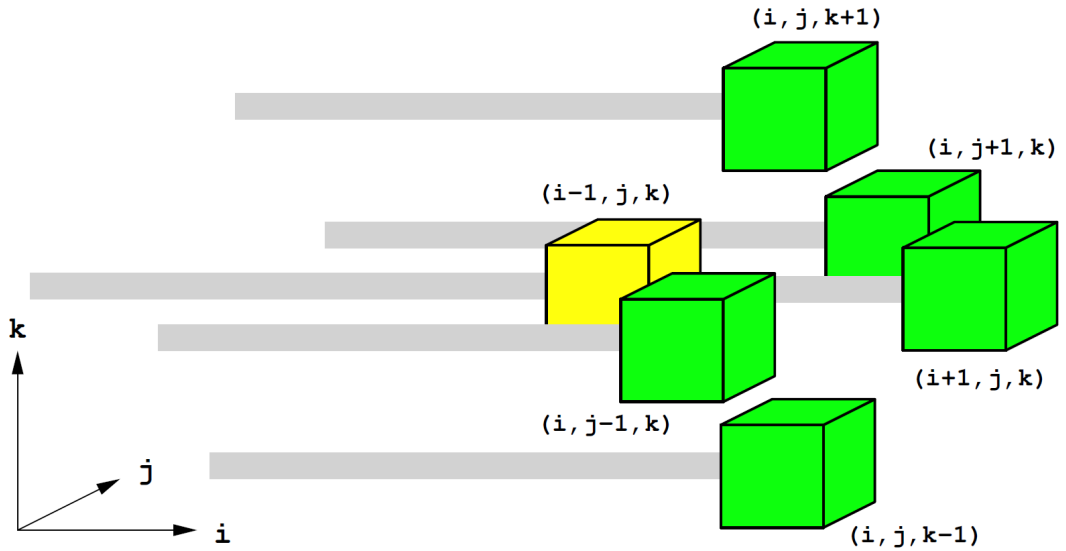
$$3 \times \mathbf{jmax} \times \mathbf{imax} \times 8 \text{ byte} < \frac{C}{2}$$

Optimal B_C = 24 B / LUP

3D 7-pt stencil

There is actually more than one layer condition in 3D

- "Outer" LC:
  `3 * imax * jmax`
- "Inner" LC:
  `3 * imax` in the central layer



→ Code balance of

- 24 B/LUP if outer LC fulfilled
- 40 B/LUP if outer LC broken but inner LC fulfilled
- 56 B/LUP if inner & outer LC broken

Online Layer Condition calculator: **`http://tiny.cc/LayerConditions`**

Conclusions from the stencil example

- We have made sense of the memory-bound performance vs. problem size
  - "Layer conditions" lead to predictions of code balance
  - "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"
- Food for thought
  - Multi-dimensional loop blocking – would it make sense?
  - Can we choose a "better" OpenMP loop schedule?
  - What about temporal blocking?

Stencil references

- T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes: Multi-dimensional intra-tile parallelization for memory-starved stencil computations. ACM Transactions on Parallel Computing **4**(3), 12:1-12:32 (2017). DOI: 10.1145/3155290, Preprint: arXiv:1510.04995

- J. Hammer, G. Hager, J. Eitzinger, and G. Wellein: Automatic Loop Kernel Analysis and Performance Modeling With Kerncraft. Proc. PMBS15, the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, in conjunction with ACM/IEEE Supercomputing 2015 (SC15), November 16, 2015, Austin, TX. DOI: 10.1145/2832087.2832092, Preprint: arXiv:1509.03778

- H. Stengel, J. Treibig, G. Hager, and G. Wellein: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. Proc. ICS15, DOI: 10.1145/2751205.2751240, Preprint: arXiv:1410.5010

- M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein: Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations. Concurrency and Computation: Practice and Experience (2015). DOI:10.1002/cpe.3489 Preprint: arXiv:1304.7664

- 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

- 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).

NHR FAU

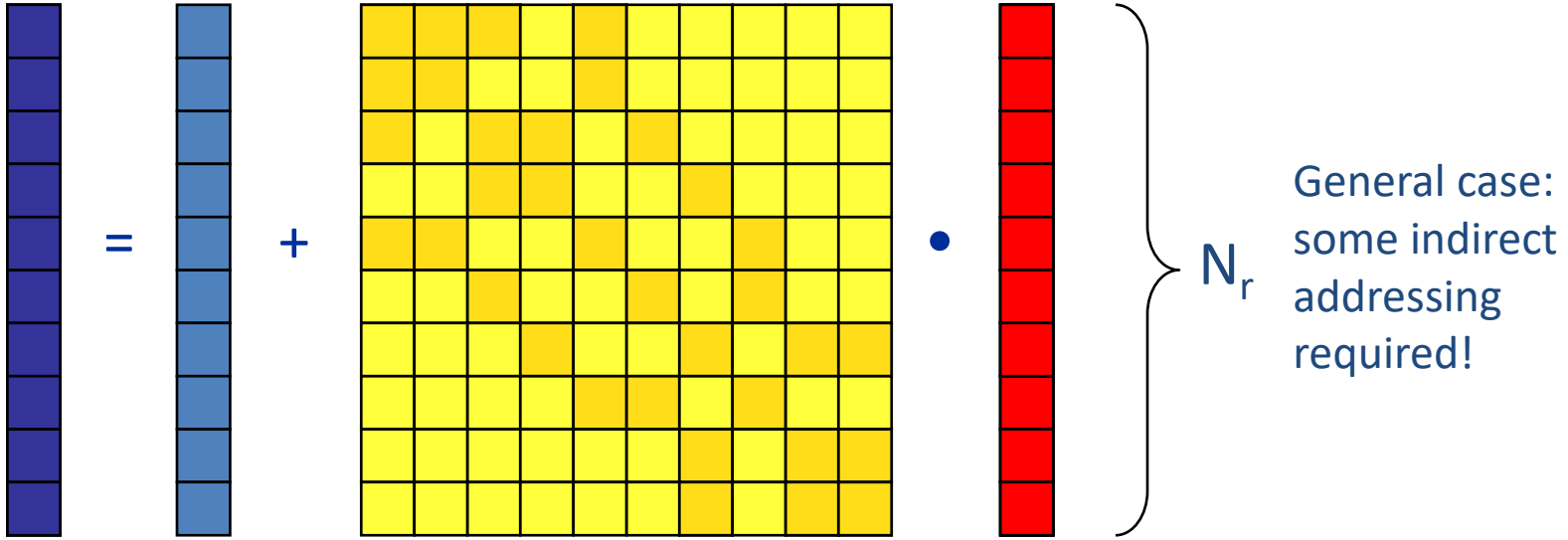
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Case study: Sparse Matrix-Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in numerous sparse linear algebra solvers

- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- "Sparse": $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzr} = N_{nz}/N_r$



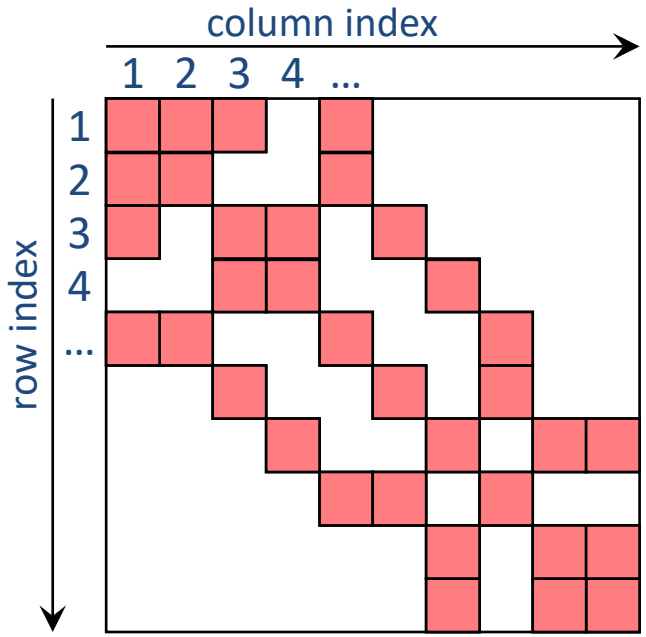
SpMVM characteristics

- For large problems, SpMV is inevitably memory-bound
  - Intra-socket saturation effect on modern multicores

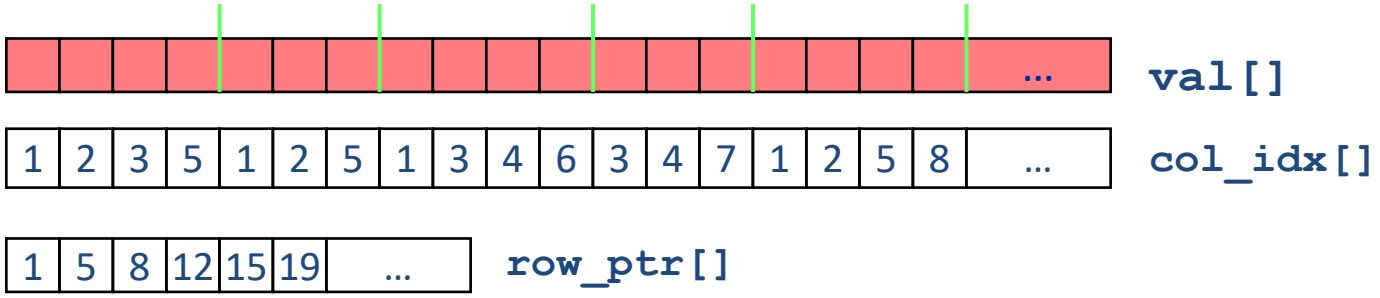
- SpMV is easily parallelizable in shared and distributed memory
  - Load balancing
  - Communication overhead

- Data storage format is crucial for performance properties
  - Most useful general format on CPUs:
    Compressed Row Storage (CRS)
  - Depending on compute architecture

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:

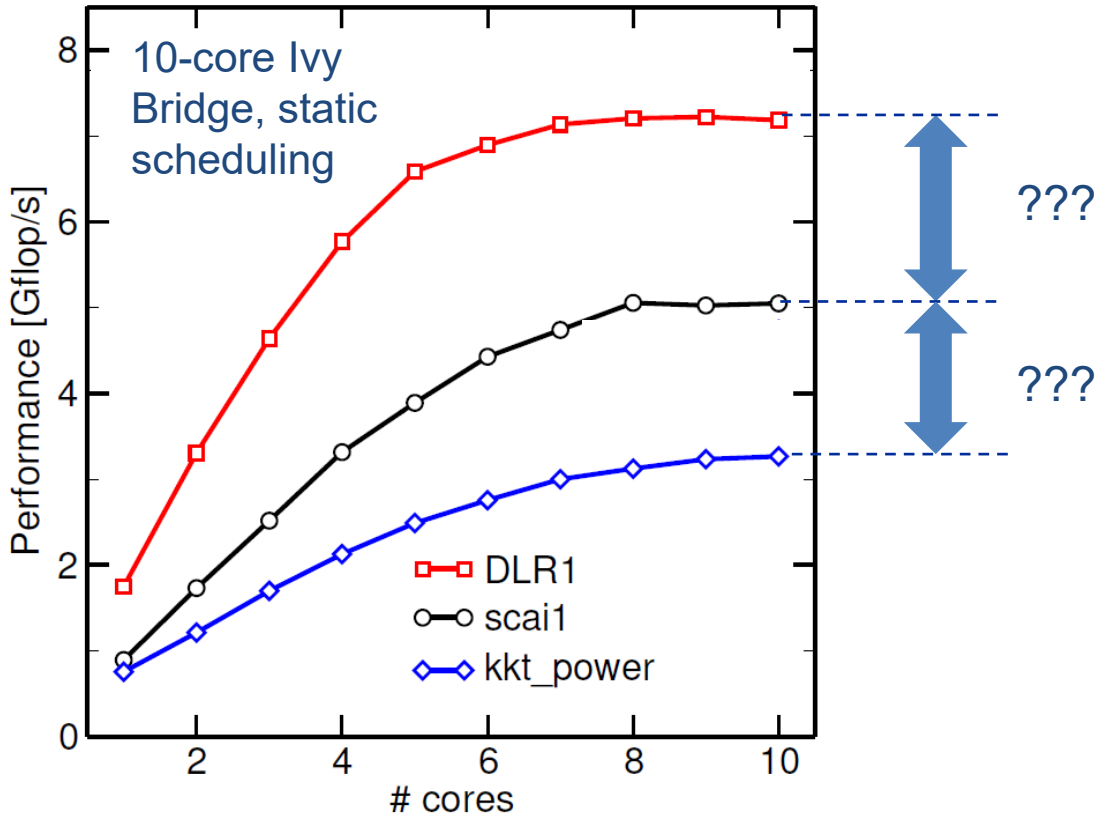
```
!$OMP parallel do schedule(???)
do i = 1,N_r
 do j = row_ptr(i), row_ptr(i+1) - 1
  C(i) = C(i) + val(j) * B(col_idx(j))
 enddo
enddo
!$OMP end parallel do
```

  - Usually many spMVMs required to solve a problem

- Now let's look at some performance measurements…

Performance characteristics

- Strongly memory-bound for large data sets → saturating performance across cores on the chip
- Performance seems to depend on the matrix
- Can we explain this?
- Is there a "light speed" for SpMV?
- Optimization?



SpMV node performance model – CRS (1)

```
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
```

```
real*8     val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8     C(Nr)
real*8     B(Nc)
```

Min. load traffic [B]: $(8 + 4)\,N_{nz} + (4 + 8)N_r + 8\,N_c$

Min. store traffic [B]: $8\,N_r$

Total FLOP count [F]: $2\,N_{nz}$

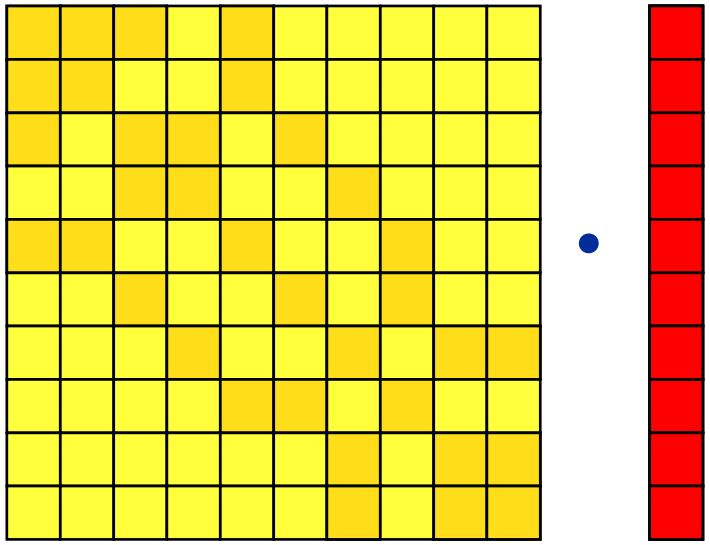
$$B_{C,min} = \frac{12\,N_{nz} + 20\,N_r + 8\,N_c}{2\,N_{nz}}\frac{B}{F} = \frac{12 + 20/N_{nzr} + 8/N_{nzc}}{2}\frac{B}{F}$$

Nonzeros per row ($N_{nzr} = {}^{N_{nz}}/_{N_r}$) or column ($N_{nzc} = {}^{N_{nz}}/_{N_c}$)

Lower bound for code balance: $B_{C,min} \geq 6\,\frac{\mathrm{B}}{\mathrm{F}}$ → $I_{\max} \leq \frac{1}{6}\,\frac{\mathrm{F}}{\mathrm{B}}$

SpMV node performance model – CRS (2)

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



$$B_{C,min} = \frac{12 + 20/N_{nzr} + 8/N_{nzc}}{2} \frac{B}{F}$$

$$B_C(\alpha) = \frac{12 + 20/N_{nzr} + \mathbf{8\,\alpha}}{2} \frac{B}{F}$$

Parameter (α) quantifies additional traffic for `B(:)` (irregular access):

$$\alpha \geq {}^{1}/_{N_{nzc}}$$

$$\alpha N_{nzc} \geq 1$$

Consider square matrices: $N_{nzc} = N_{nzr}$ and $N_c = N_r$

Note: $B_C\left({}^{1}/_{N_{nzr}}\right) = B_{C,min}$

The "α effect"

DP CRS code balance

- α quantifies the traffic for loading the 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!
- "Target" performance = b_S/B_c
- Caveat: Maximum memory BW may not be achieved with spMVM (see later)

$$B_C(\alpha) = \frac{12 + 20/N_{nzr} + 8\,\alpha}{2}\,\frac{B}{F}$$

$$= \left(6 + 4\,\alpha + \frac{10}{N_{nzr}}\right)\frac{B}{F}$$

Can we predict α?

- Not in general
- Simple cases (banded, block-structured): Similar to layer condition analysis

→ Determine α by measuring the actual memory traffic (→ measured code balance B_C^{meas})

Determine α (RHS traffic quantification)

$$B_C\,(\alpha) = \left(6+4\alpha+\frac{10}{N_{nzr}}\right)\frac{\mathrm{B}}{\mathrm{F}} = \frac{V_{meas}}{N_{nz}\cdot 2\ \mathrm{F}}\ \ (= B_C^{meas})$$

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

$$\alpha = \frac{1}{4}\left(\frac{V_{meas}}{N_{nz}\cdot 2\ \mathrm{bytes}} - 6 - \frac{10}{N_{nzr}}\right)$$

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

- $N_{nz} = 14.6\cdot 10^6,\ N_{nzr} = 7.1$
- $V_{meas} \approx 258\ \mathrm{MB}$

→ $\alpha = 0.36,\ \alpha N_{nzr} = 2.5$

→ RHS is loaded 2.5 times from memory

$$\frac{B_C\,(\alpha)}{B_{C,min}} = 1.11$$

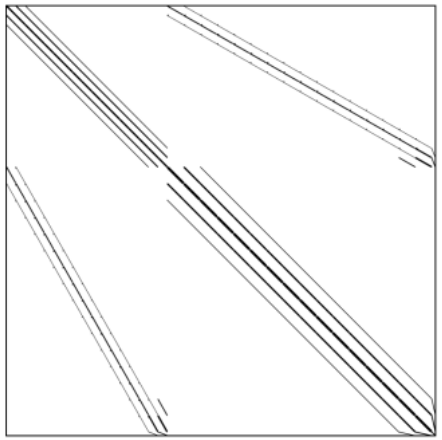
11% extra traffic → optimization potential!

Three different sparse matrices

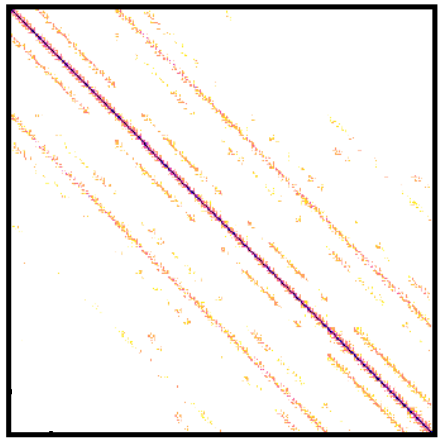
Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_S = 46.6\,\mathrm{GB/s}$

→ Roofline: $P_{opt} = {b_S}/{B_{C,min}}$

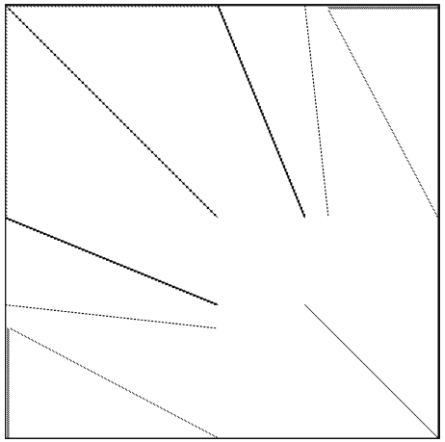
Matrix	N	N_{nzr}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

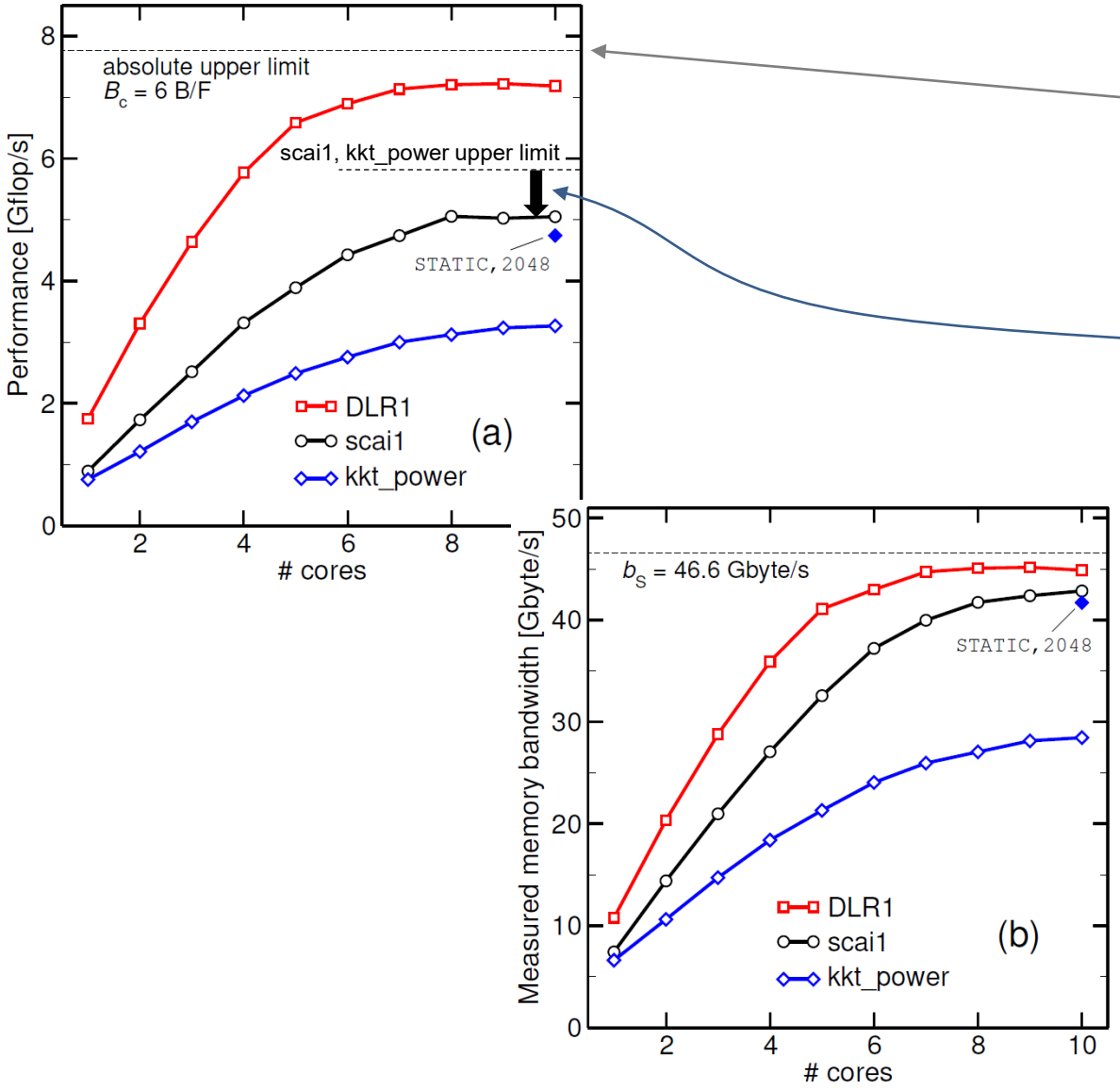


scai1



kkt_power

Now back to the start…



- $b_S = 46.6\,\mathrm{GB/s}, \quad B_c = 6\,\mathrm{B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8\,\mathrm{GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)

- **scai1** measured balance:

$B_c^{meas} \approx 8.5\,\mathrm{B/F} > B_{C,min}$ (6% higher than min)

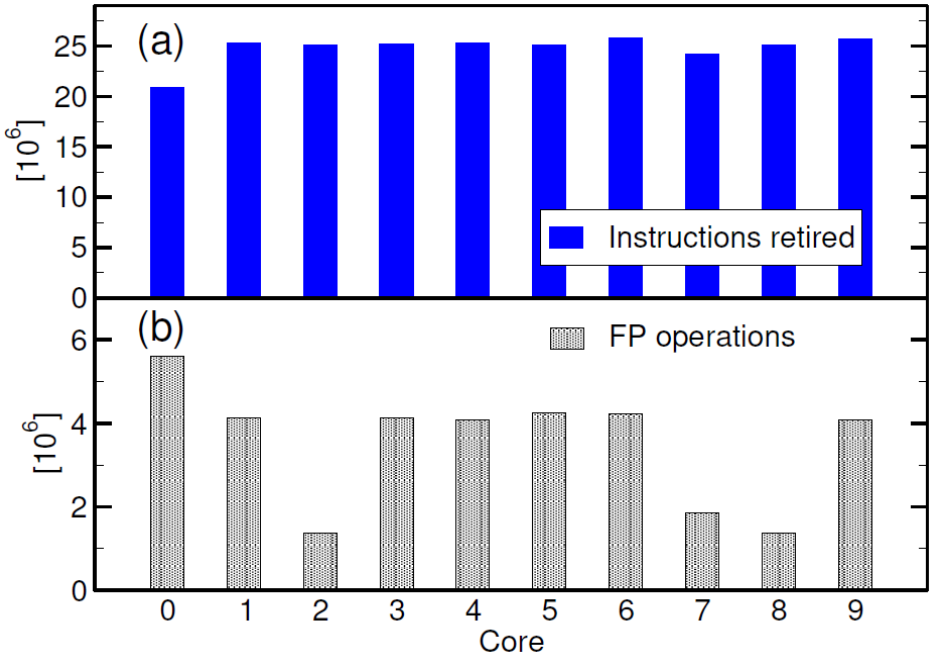
→ good BW utilization, slightly non-optimal α

- **kkt_power** measured balance:

$B_c^{meas} \approx 8.8\,\mathrm{B/F} > B_{C,min}$ (10% higher than min)

→ performance degraded by load imbalance, fix by block-cyclic schedule

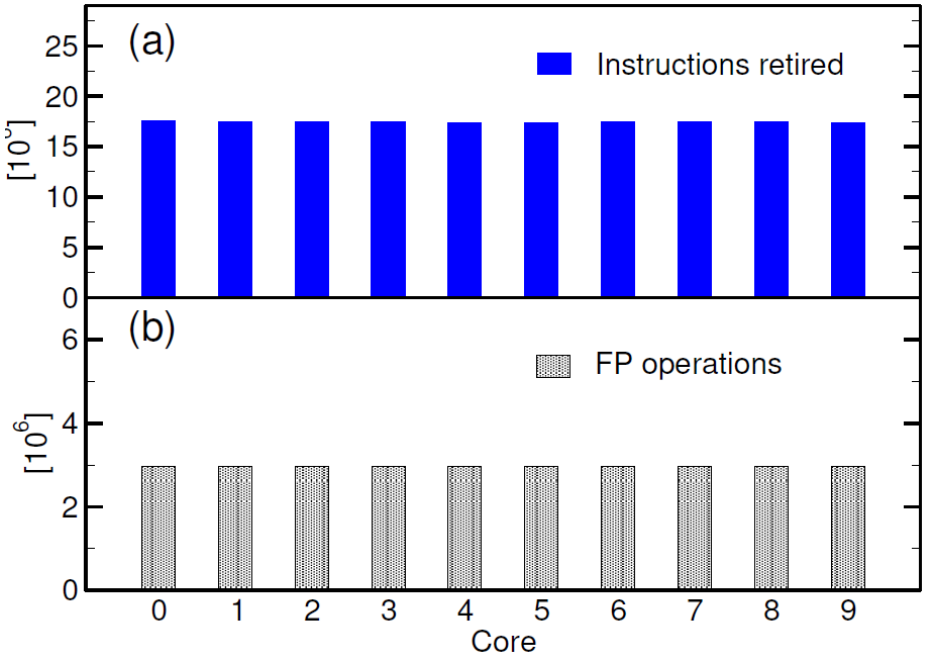
Investigating the load imbalance with kkt_power



Measurements with likwid-perfctr (MEM_DP group)

`static`

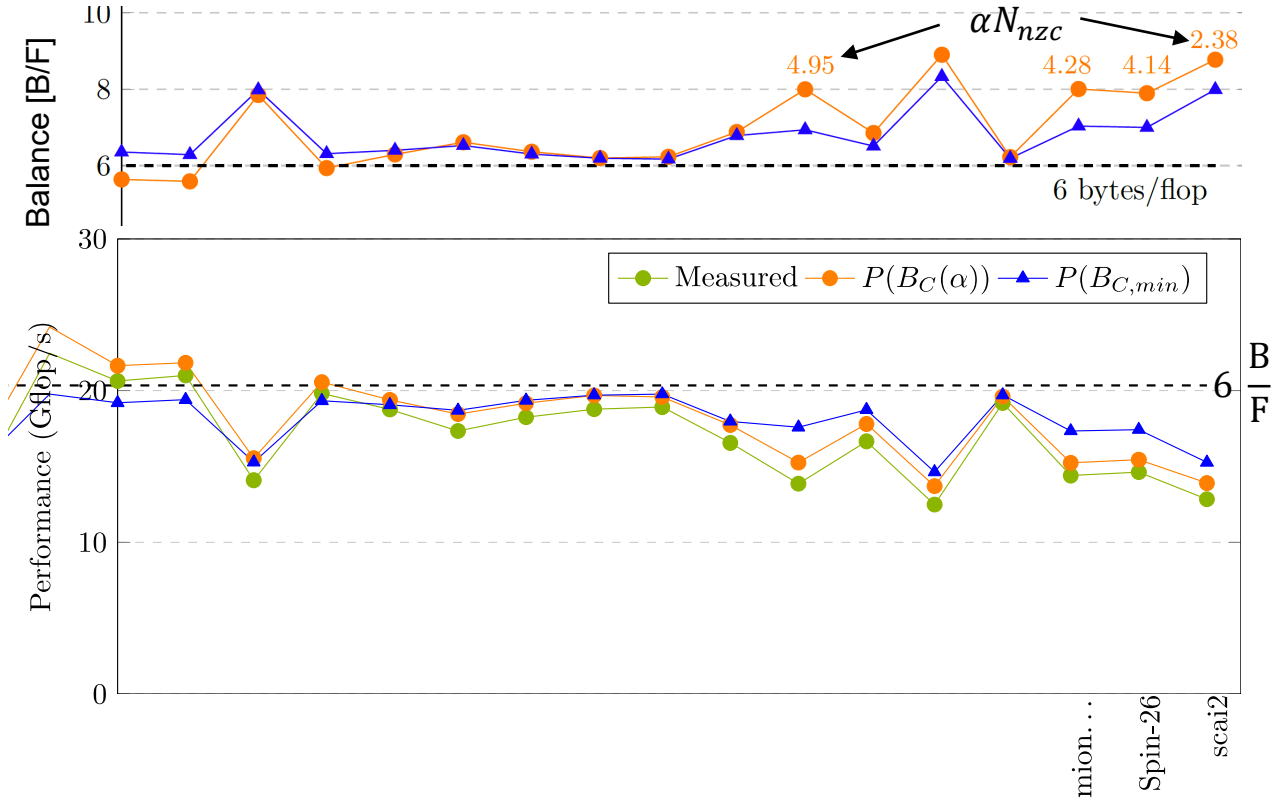
`static,2048`



→ Fewer overall instructions, (almost) BW saturation, 50% better performance with load balancing

→ CPI value unchanged!

SpMV node performance model – CPU



Intel Xeon Platinum 9242
24c@2.8GHz (turbo)
$b_S = 122\ GB/s$

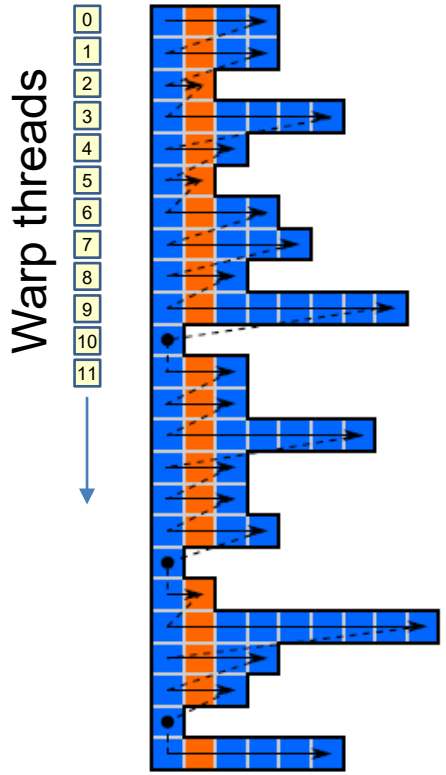
Matrices taken from: C. L. Alappat et al.: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.*
DOI: 10.1002/cpe.6512

What about GPUs?

- GPUs need
  - Enough work per kernel launch in order to leverage their parallelism
  - Coalesced access to memory (consecutive threads in a warp should access consecutive memory addresses)

- Plain CRS for SpMV on GPUs is not a good idea
  1. Short inner loop
  2. Different amount of work per thread
  3. Non-coalesced memory access

- Remedy: Use SIMD/SIMT-friendly storage format
  - ELLPACK, SELL-C-σ, DIA, ESB,…



CRS SpMV in CUDA (y = Ax)

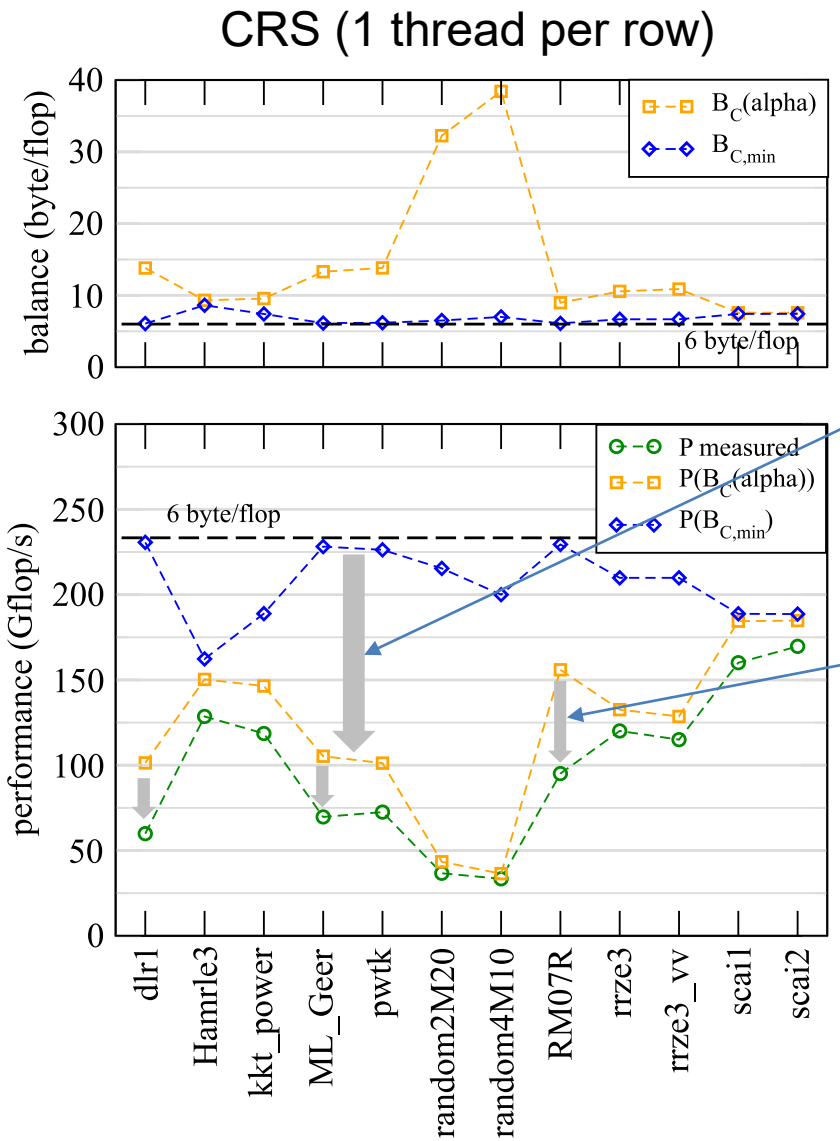
```
template <typename VT, typename IT>
__global__ static void
spmv_csr(const ST num_rows,
         const IT * RESTRICT row_ptrs, const IT * RESTRICT col_idxs,
         const VT * RESTRICT values,   const VT * RESTRICT x,
                                       VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x; // 1 thread per row

    if (row < num_rows) {
        VT sum{};
        for (IT j = row_ptrs[row]; j < row_ptrs[row + 1]; ++j) {
            sum += values[j] * x[col_idxs[j]];
        }
        y[row] = sum;
    }
}
```

$$B_c(\alpha) = \left(6 + 4\,\alpha + \frac{6}{N_{nzr}}\right)\frac{B}{F}$$

No write-allocate on GPUs for consecutive stores

SpMV CRS performance on a GPU



NVIDIA Ampere A100
Memory bandwidth $b_S = 1400\ \mathrm{GB/s}$

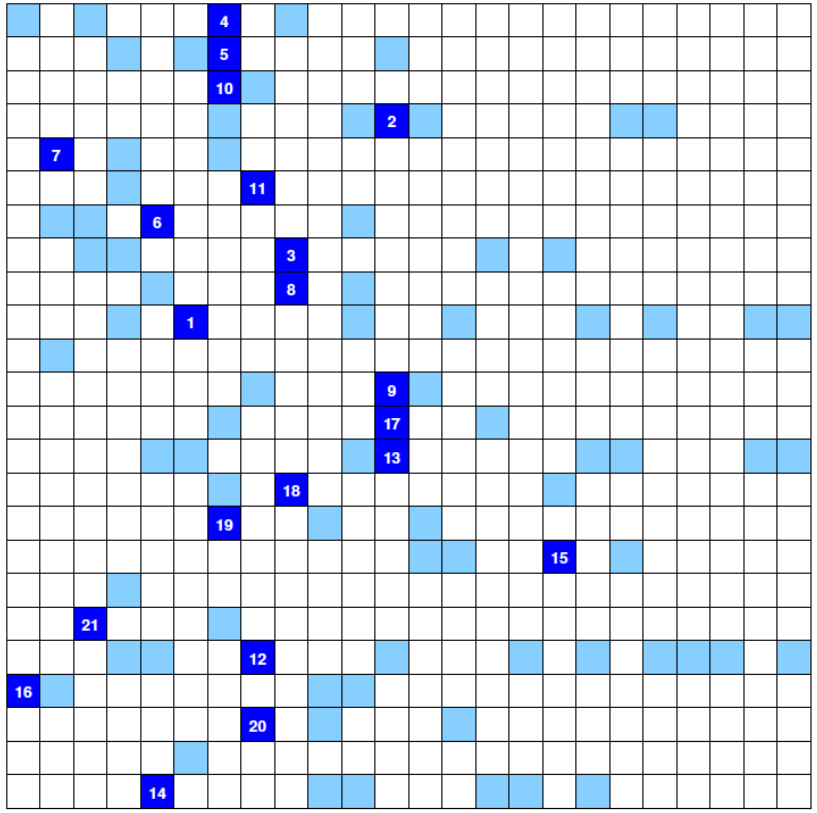
- Strong "α effect" – large deviation from optimal α for many matrices
  - Many cache lines touched b/c every thread handles one row → bad cache usage
- Mediocre memory bandwidth usage ($\ll 1400\ \mathrm{GB/s}$) in many cases
  - Non-coalesced memory access
  - Imbalance across rows/threads of warps

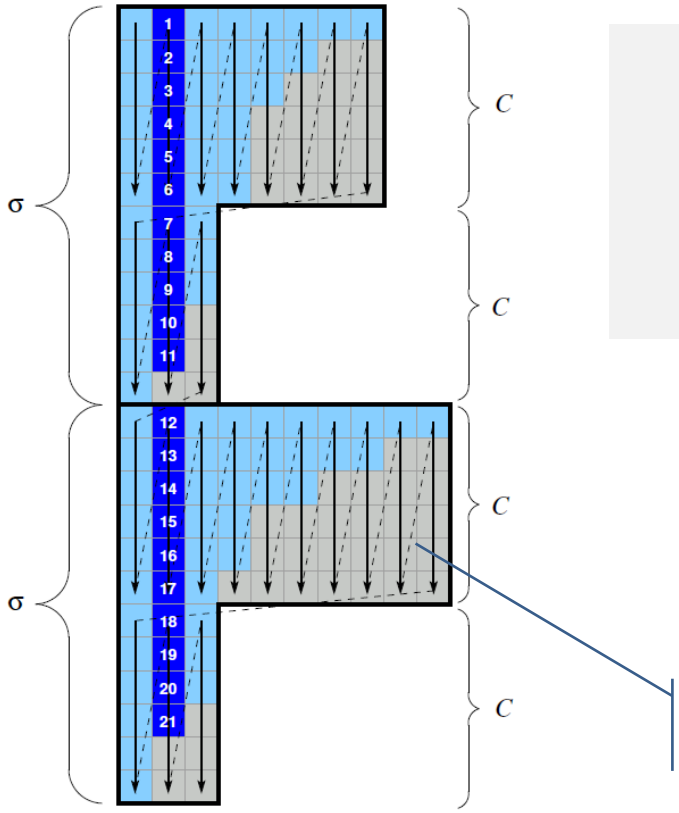
SELL-C-σ

M. Kreutzer et al.: *A Unified Sparse Matrix Data Format For Efficient General Sparse Matrix-vector Multiplication On Modern Processors With Wide SIMD Units*, SIAM SISC 2014, DOI: 10.1137/130930352

Idea

- Sort rows according to length within sorting scope σ
- Store nonzeros column-major in zero-padded chunks of height C





"Chunk occupancy":

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$

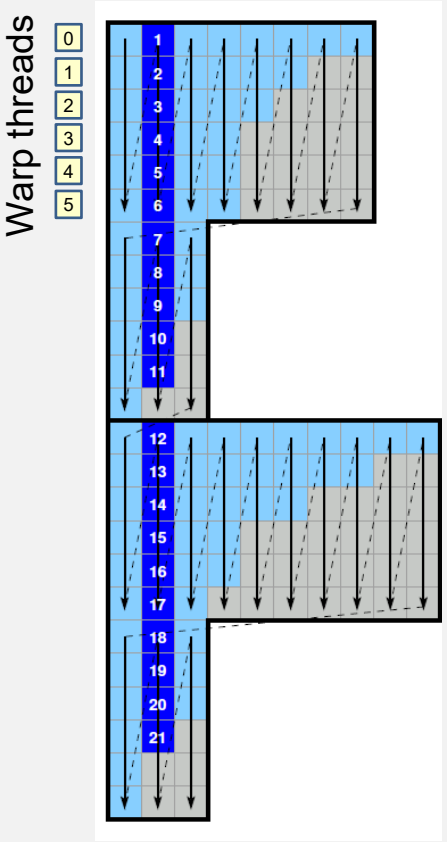
l_i: width of chunk i

SELL-C-σ SpMV in CUDA (y=Ax)

```
template <typename VT, typename IT> __global__ static void
spmv_scs(const ST C, const ST n_chunks,      const IT * RESTRICT chunk_ptrs,
         const IT * RESTRICT chunk_lengths, const IT * RESTRICT col_idxs,
         const VT * RESTRICT values, const VT * RESTRICT x, VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x;
    ST c   = row / C;  // the no. of the chunk
    ST idx = row % C;  // index inside the chunk

    if (row < n_chunks * C) {
        VT tmp{};
        IT cs = chunk_ptrs[c]; // points to start indices of chunks

        for (ST j = 0; j < chunk_lengths[c]; ++j) {
            tmp += values[cs + idx] * x[col_idxs[cs + idx]];
            cs += C;
        }
        y[row] = tmp;
    }
}
```



Code balance of SELL-C-σ (y=Ax)

Matrix data & column index

LHS update (write only)

chunk index

$$B_{SELL}(\alpha,\beta,N_{nzr}) = \left(\frac{1}{\beta}\left(\frac{8+4}{2}\right) + \frac{8\alpha + \beta(8+4/C)/N_{nzr}}{2}\right)\frac{\text{bytes}}{\text{flop}}$$

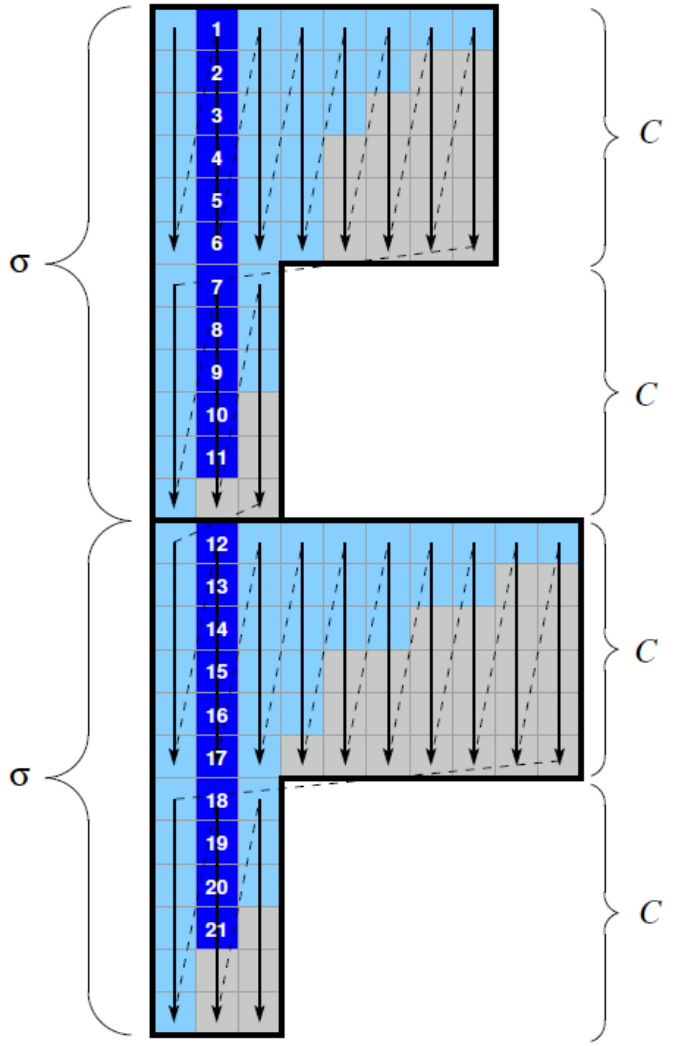
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzr}}\right)\frac{\text{bytes}}{\text{flop}}$$

Optimal $\alpha = \frac{\beta}{N_{nzr}}$

When measuring B_C^{meas}, take care to use the "useful" number of flops (excluding zero padding) for work

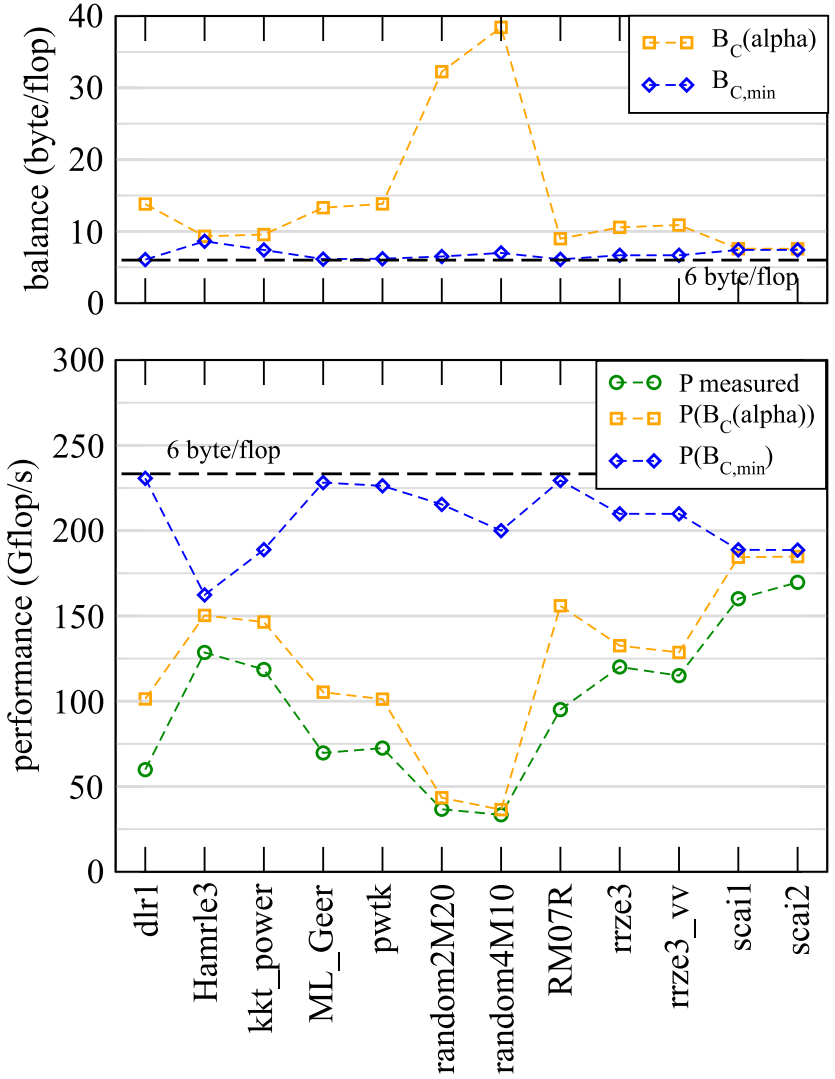
How to choose the parameters C and σ on GPUs?

- C
  - $n \times$ warp size to allow good utilization of GPU threads and cache lines
- σ
  - As small as possible, as large as necessary
  - Large σ reduces zero padding (brings β closer to 1)
  - Sorting alters RHS access pattern → α depends on σ



SpMV node performance model – GPU

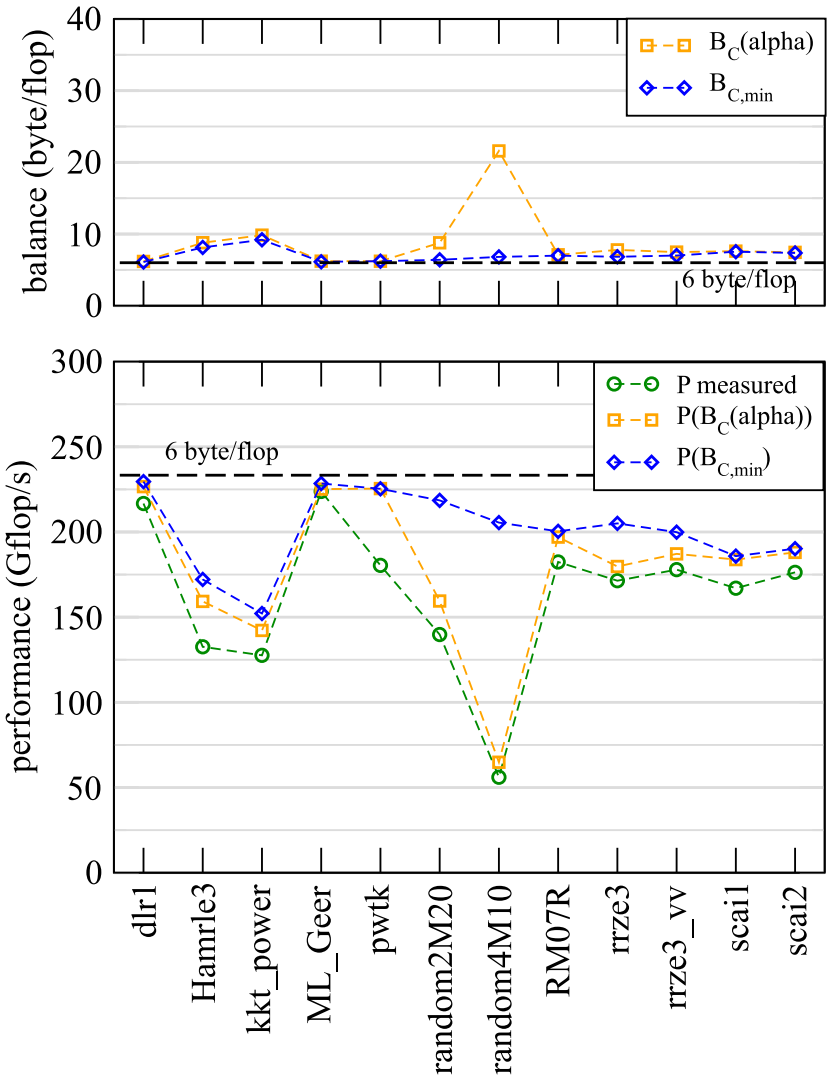
CRS (1 thread per row)



NVIDIA Ampere A100

$b_S = 1400$ GB/s

SELL-32-128



Roofline analysis for spMVM

- Conclusion from the Roofline analysis
  - The roofline model does not “work” 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
- Do not forget about load balancing!
- Sparse matrix times multiple vectors bears the potential of huge savings in data volume
- Consequence: Modeling is not always 100% predictive. It‘s all about *learning more* about performance properties!