Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Case study: A Jacobi smoother

The basics in two dimensions

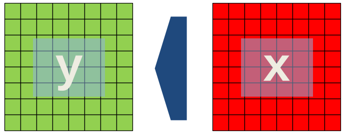
Layer conditions

Optimization by spatial blocking

Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- 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
```

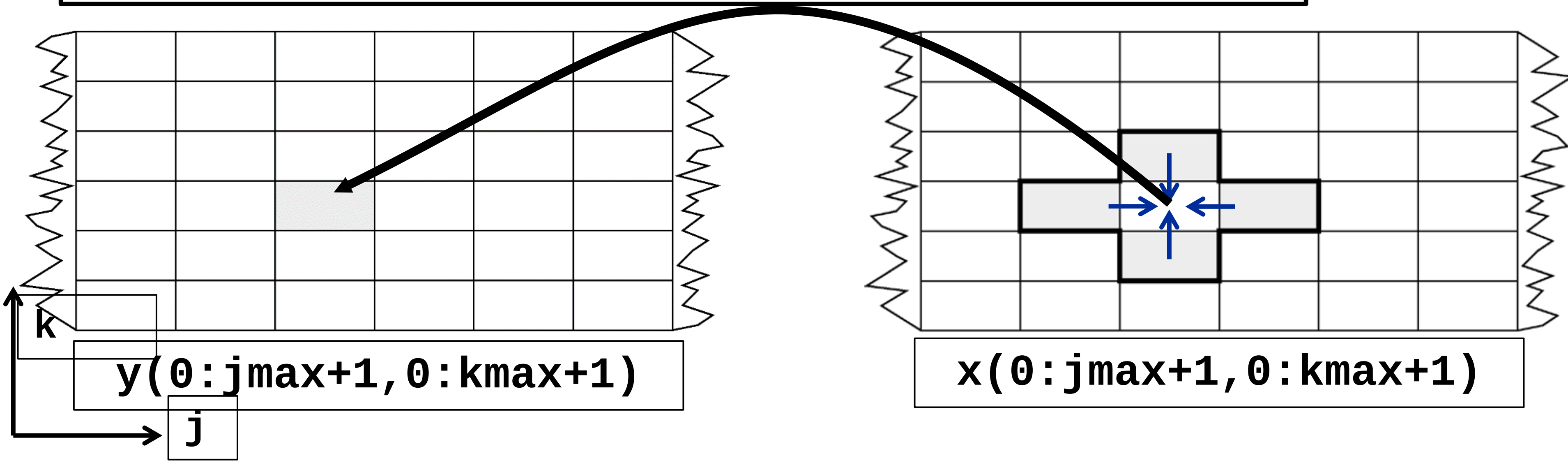


- Complexity of implementation and performance depends on
  - update scheme, e.g. Jacobi-type, Gauss-Seidel-type, …
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…

Jacobi-type 5-pt stencil in 2D

```
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
```

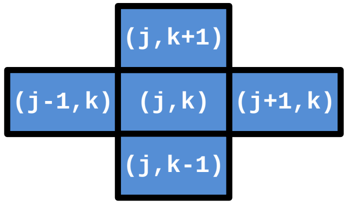
Lattice site Update (LUP)



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 in 2D: data transfer analysis



Available in cache (used 2 updates before)

RD+WR y(j,k) (incl. write allocate)

RD 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
```

RD x(j,k-1)

RD x(j,k+1)

Naive balance (incl. write allocate):

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

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

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

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

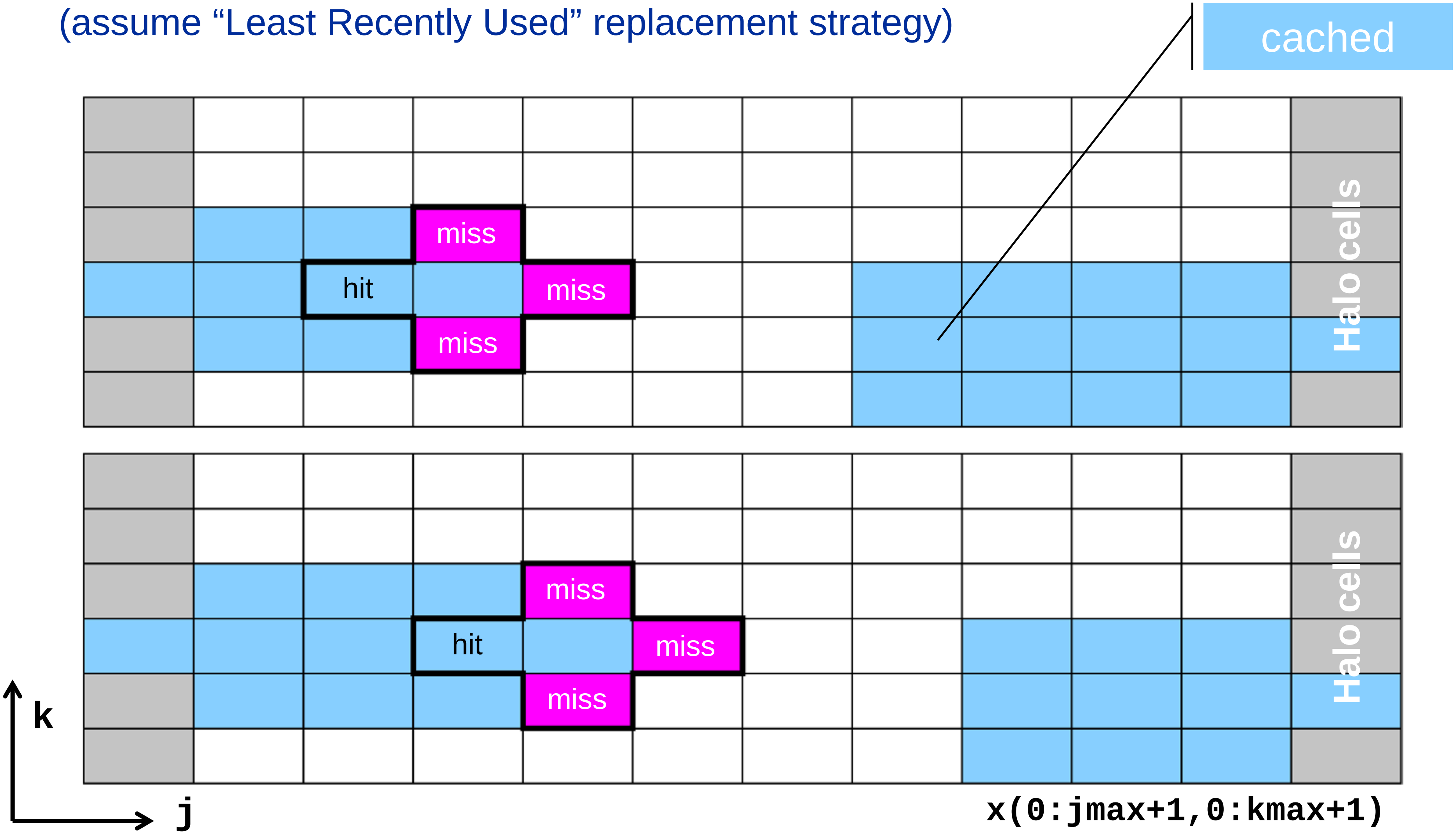
Case study: A Jacobi smoother

The basics in two dimensions

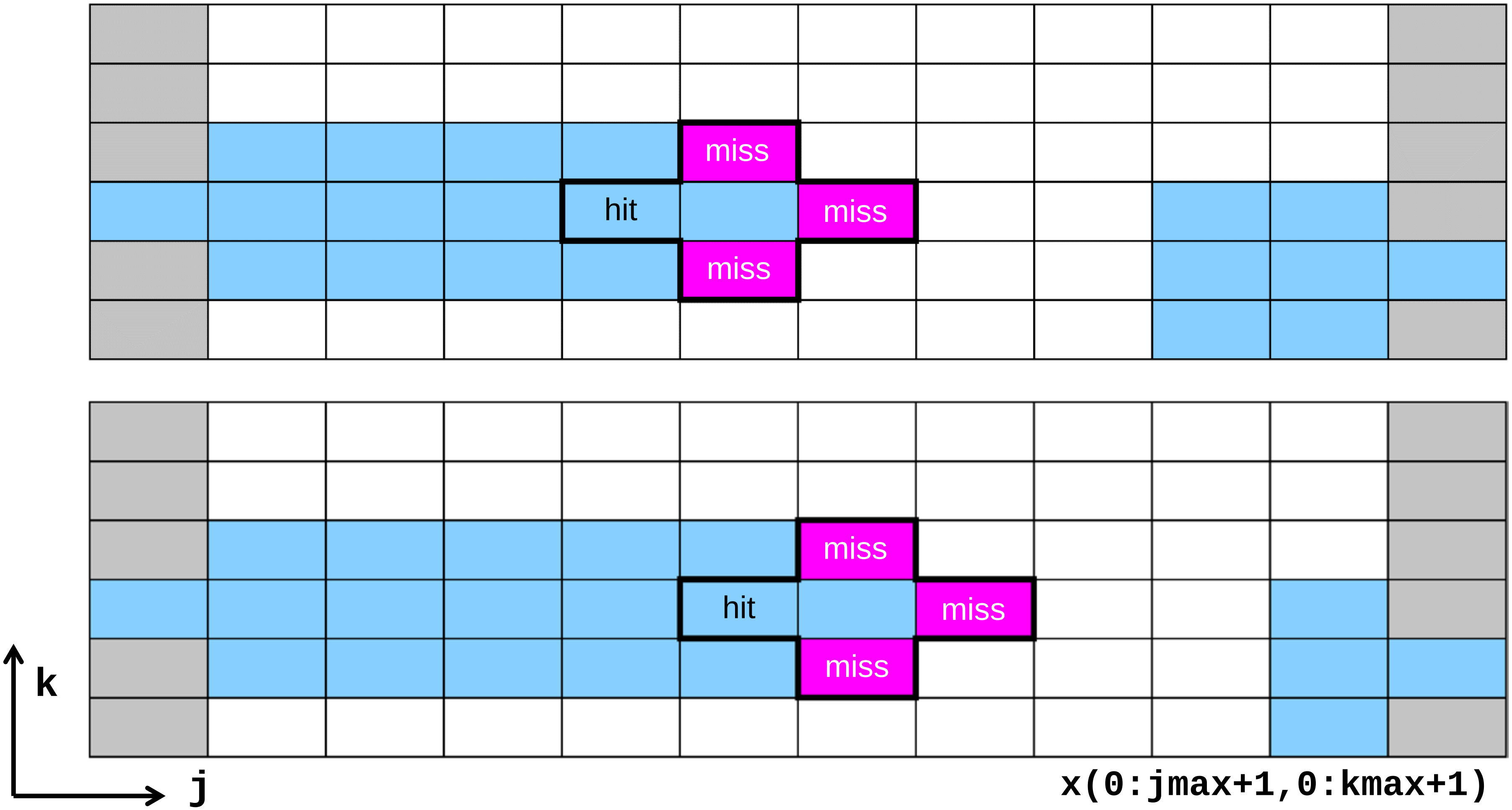
Layer conditions

Optimization by spatial blocking

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

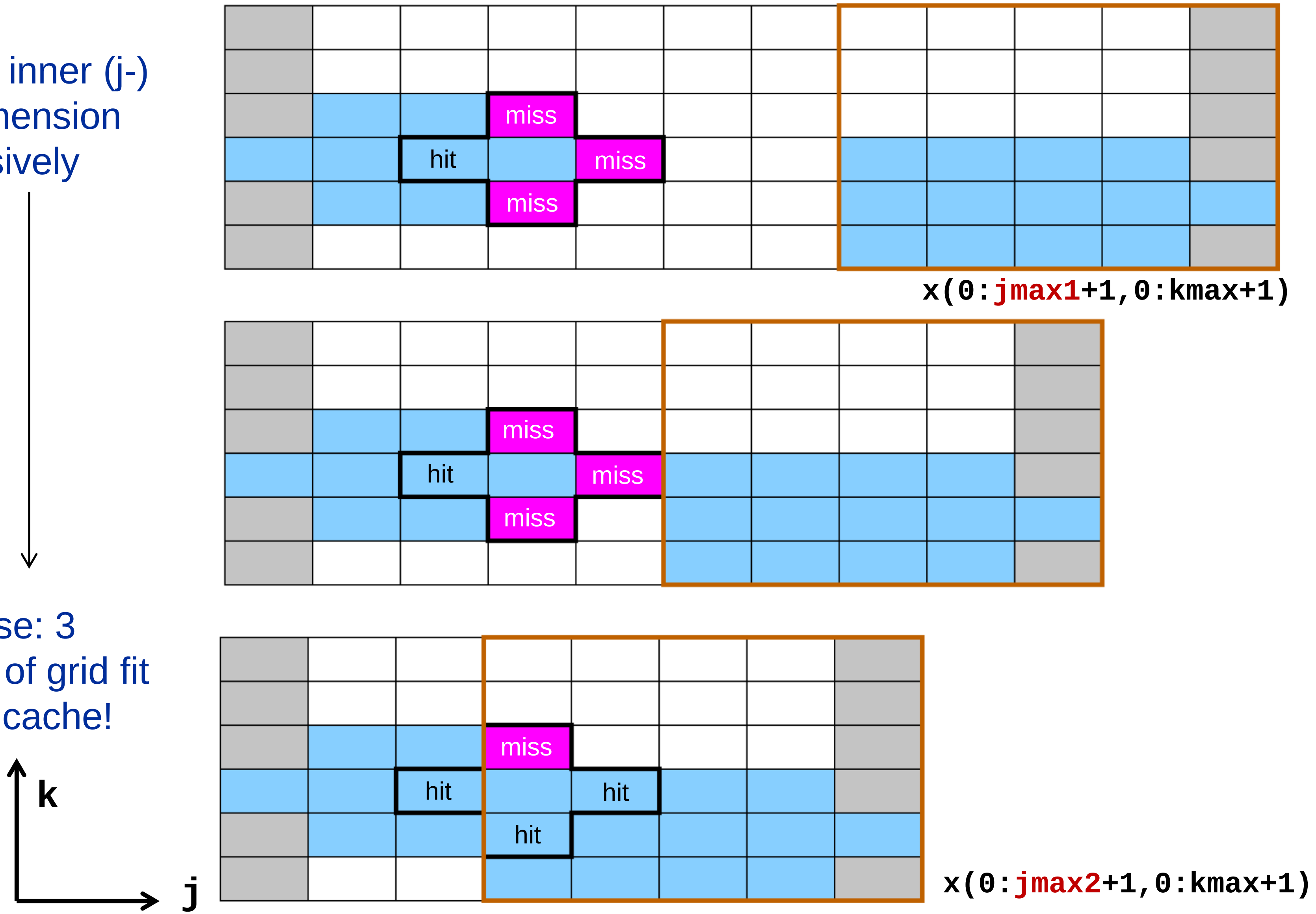


Worst case: Cache not large enough to hold 3 layers (rows) of grid
(+assume „Least Recently Used“ replacement strategy)

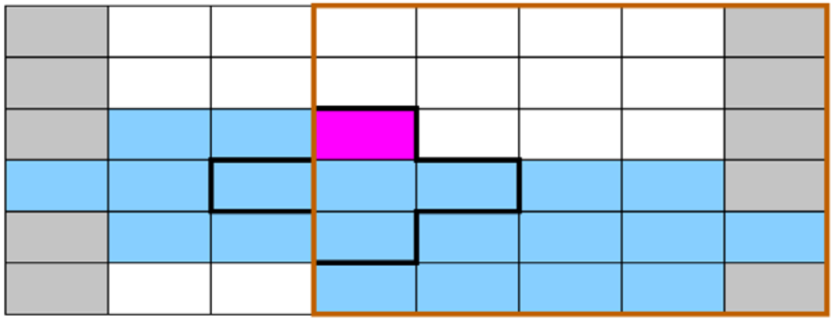


Reduce inner (j-) loop dimension successively

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

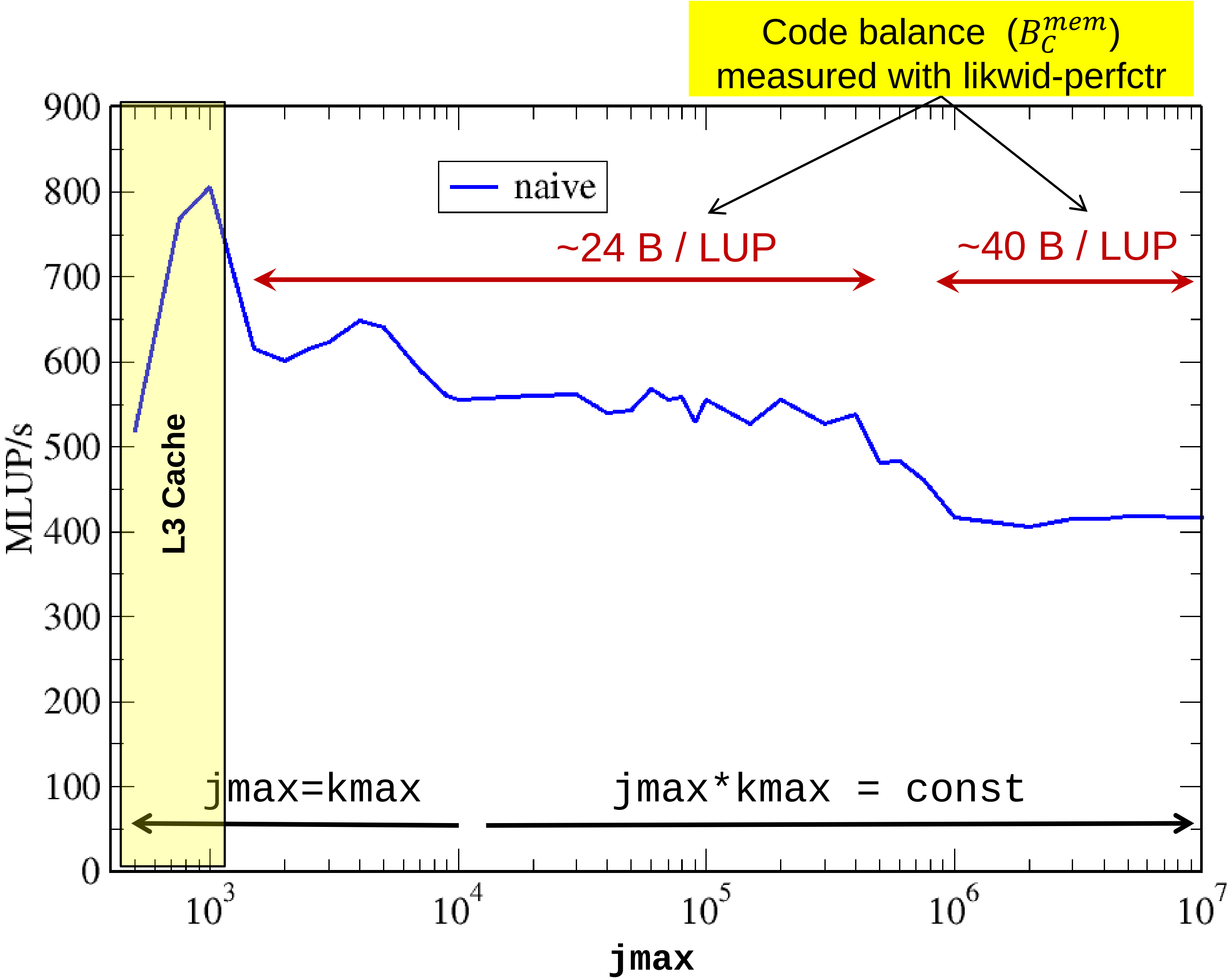
"Layer condition"

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

Layer condition:

- 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., 3D 7-pt stencil)

Jacobi 5-pt stencil in 2D: Single core performance



Questions:

1. How to achieve 24 B/LUP also for large **`jmax`**?
2. How to sustain >600 MLUP/s for **`jmax`** > 10^4 ?

Intel Compiler
ifort V13.1

Intel Xeon E5-2690 v2 (“IvyBridge”@3 GHz)

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Case study: A Jacobi smoother

The basics in two dimensions

Layer conditions

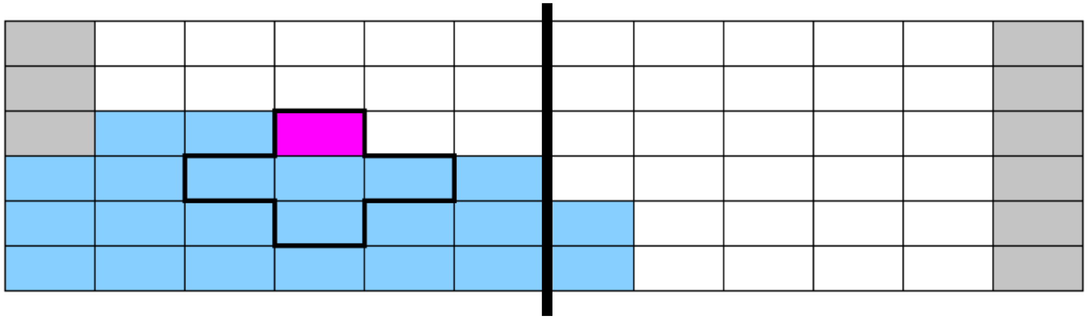
Optimization by spatial blocking

Why 24 byte/LUP?

- High-level view of sweep: read **x(:)**, write **y(:)**
  → if maximum reuse with sweep is possible, 24 byte/LUP should be achievable

But how to establish the layer condition for all domain sizes?

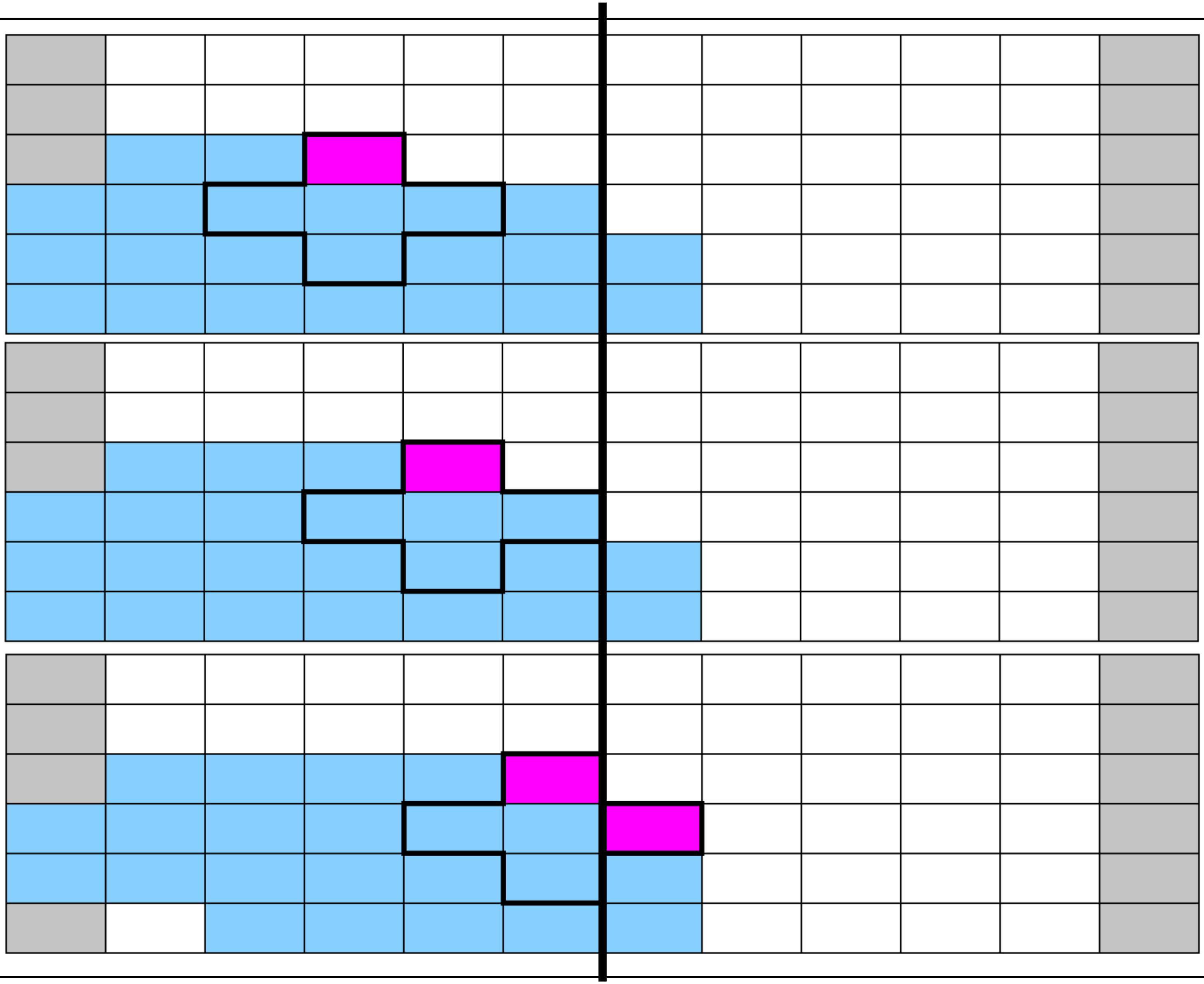
- Idea: Spatial blocking
  - Reuse elements of **x(:)** as long as they stay in cache
  - Main idea: Order of site updates does not matter
    → "reduce inner dimension" by cutting the inner loop short



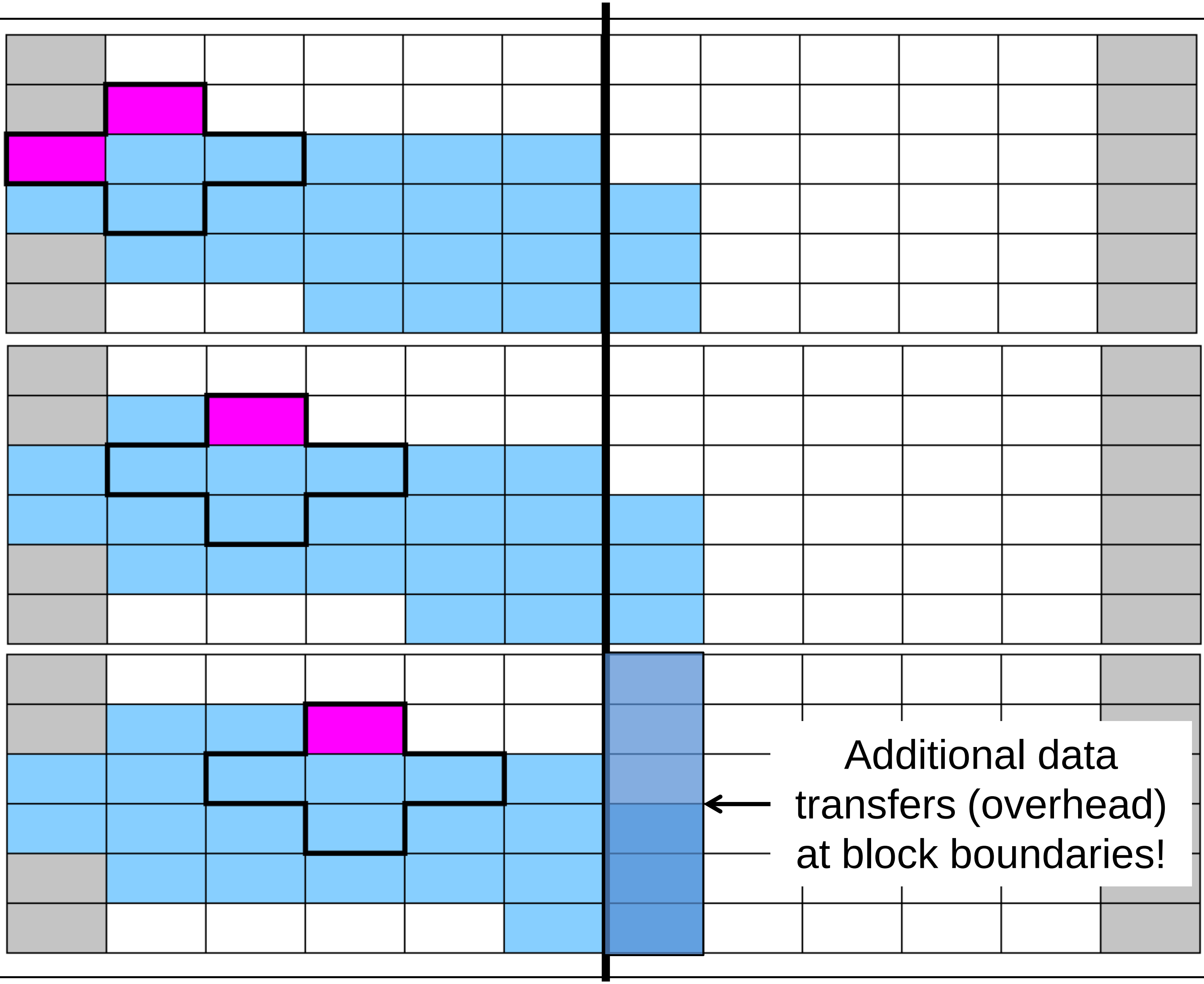
Idea: Enable data reuse by blocking!

Split domain into subblocks:

e.g. block size = 5



Establish the layer condition by blocking



→ "Spatial Blocking" of j-loop:

```
do jb=1,jmax,jblock !      Assume jmax is multiple of jblock
  do k=1,kmax
    do j= jb, (jb+jblock-1) ! Length of inner loop: jblock
      y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                     +  x(j,k-1) + x(j,k+1) )
    enddo
  enddo
enddo
```

→ Determine for given **CacheSize** an appropriate **jblock** value:

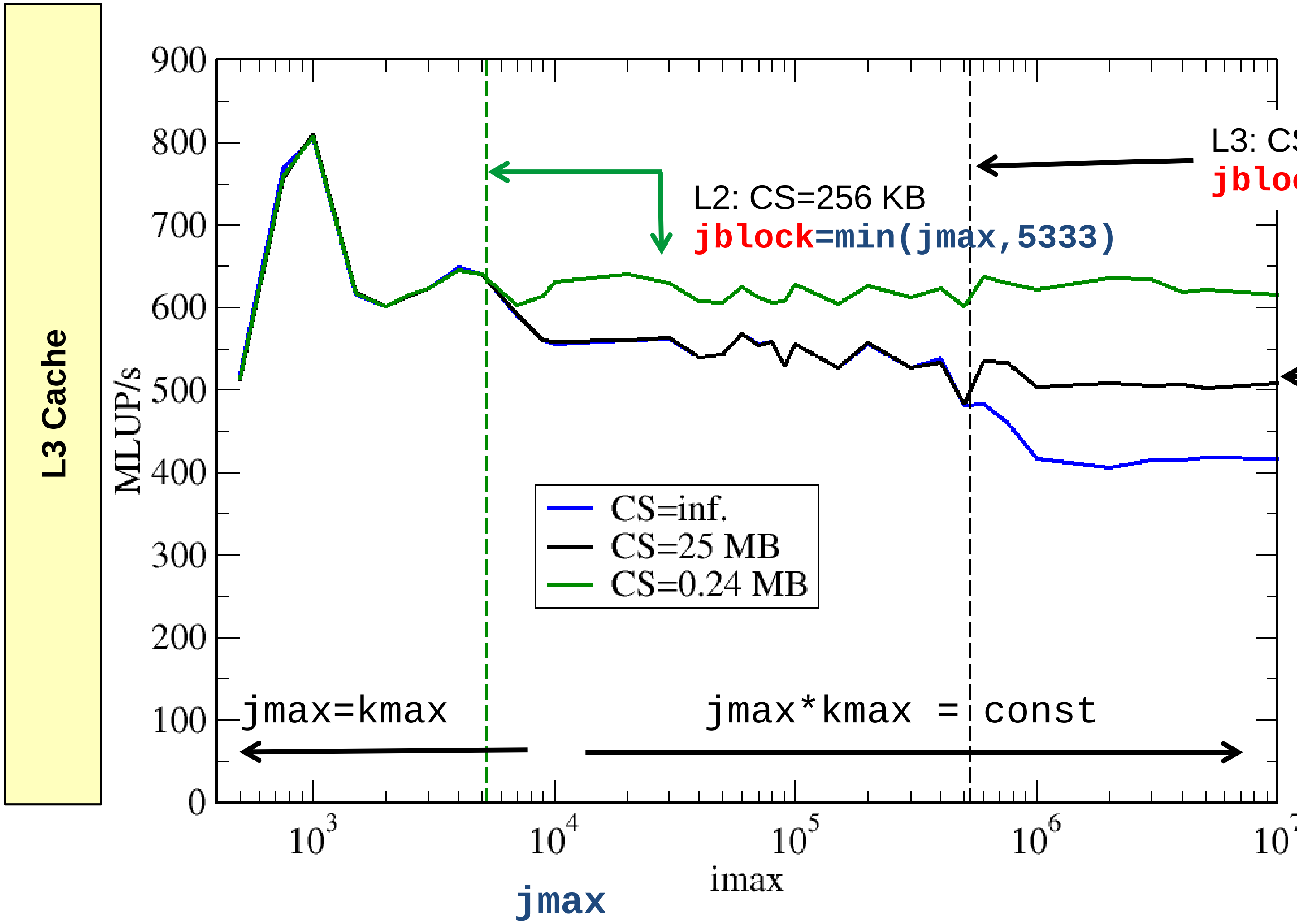
New layer condition (blocking)
3 * jblock * 8B < CacheSize/2

jblock < CacheSize / 48 B

Establish layer condition by spatial blocking

`jblock < CacheSize / 48 B`

Which cache to block for?

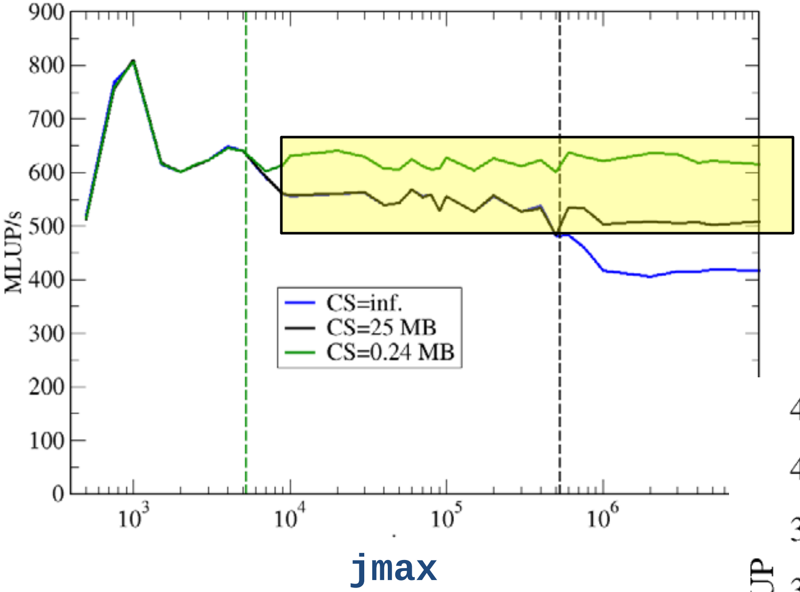


Intel Compiler
ifort V13.1

Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

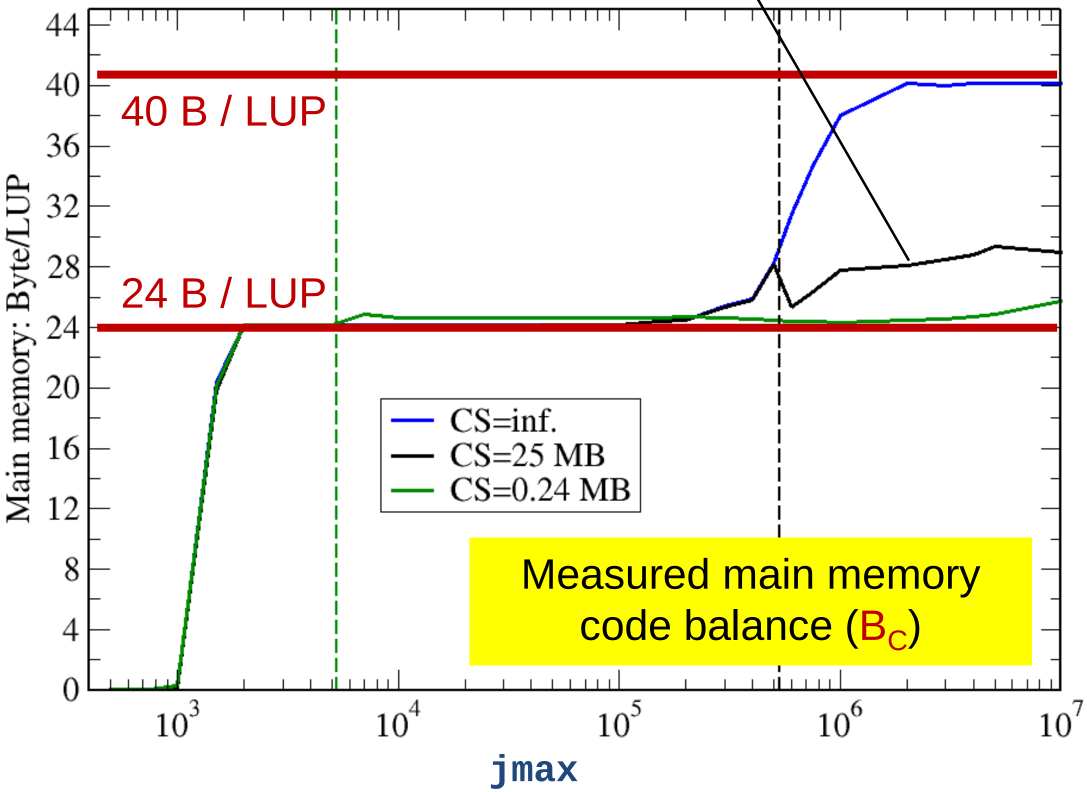
L1: 32 KB
L2: 256 KB
L3: 25 MB

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG



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

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

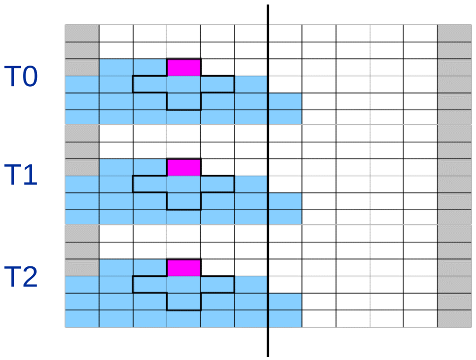


Measured main memory code balance (B_C)

Intel Compiler
ifort V13.1

Intel Xeon E5-2690 v2
("IvyBridge"@3 GHz)

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
```

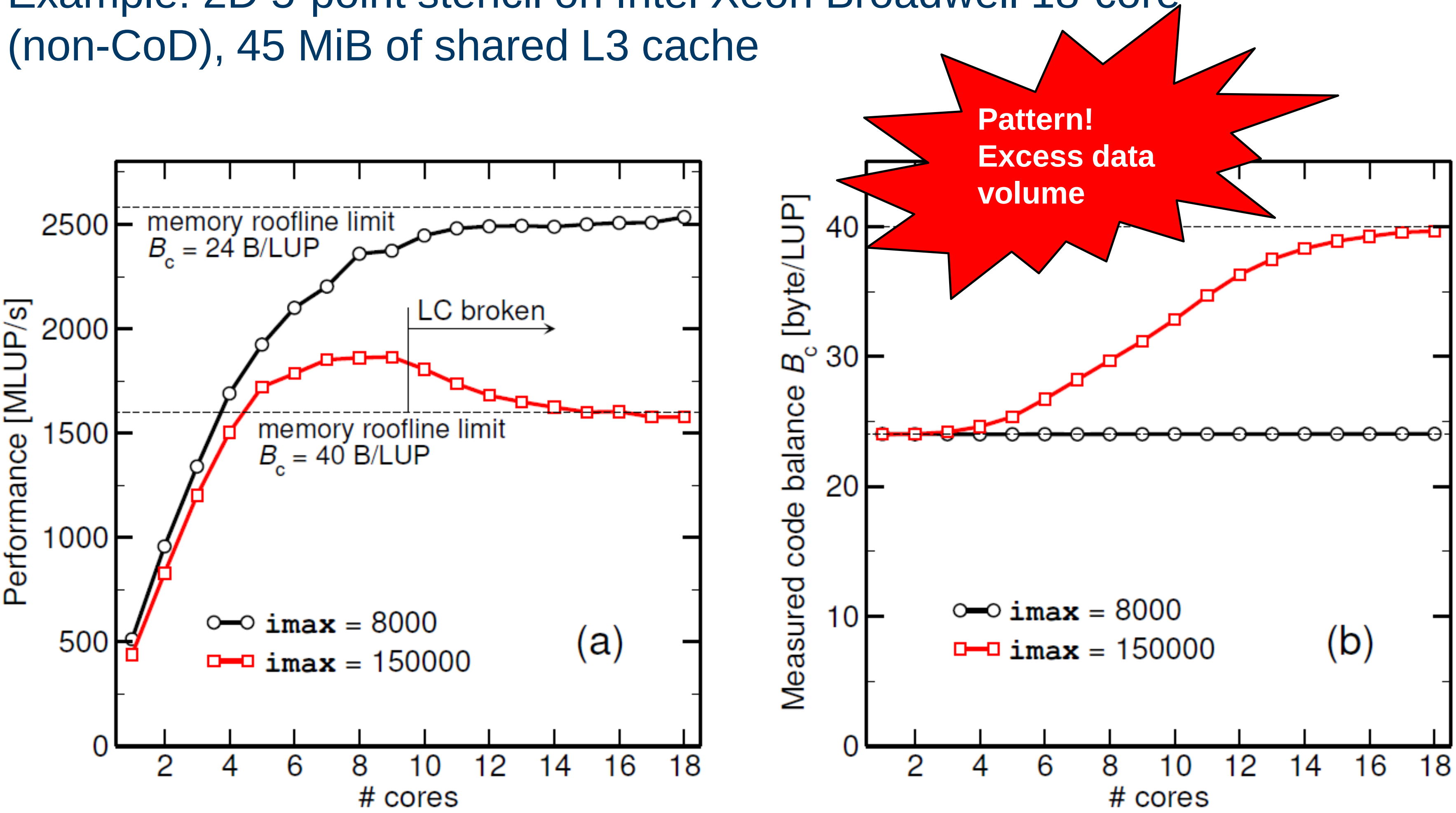
- Caveat: LC must be fulfilled per thread → shared cache causes smaller blocks!

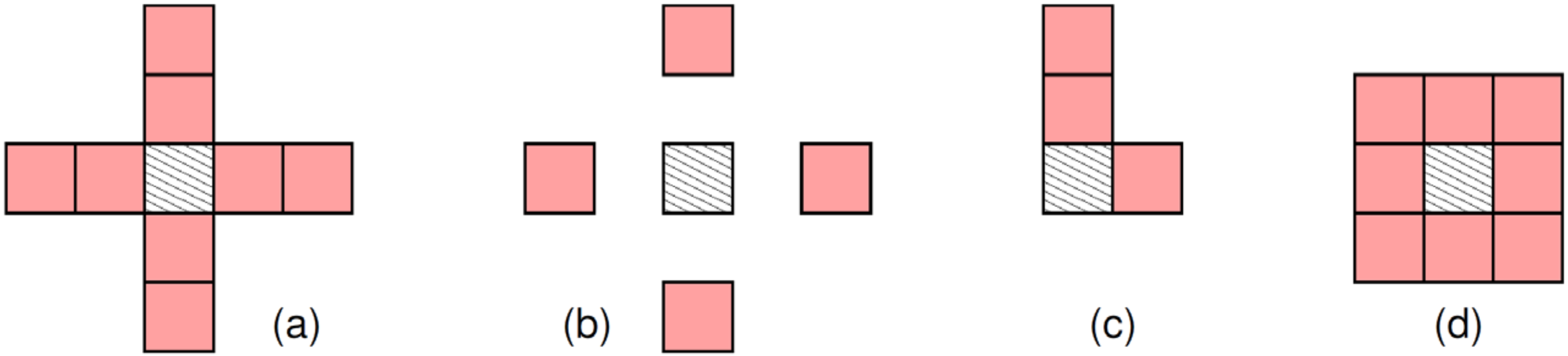
Layer condition:
3 * jblock * 8B < $CS_t/2$

Cache size available per thread

optional

Example: 2D 5-point stencil on Intel Xeon Broadwell 18-core (non-CoD), 45 MiB of shared L3 cache





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

b) Long-range $r = 2$ with gaps: 6 layers (2 per populated row)

c) Asymmetric: 3 layers

d) 2D box: 3 layers

Conclusions from the 2D 5pt 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 would change if we parallelized inner loops?