Friedrich-Alexander-Universität Erlangen-Nürnberg

Performance Engineering
Case study: “Jacobi” stencil

The basics in two dimensions (2D)

Layer condition in 2D

From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a]
[a] Erlangen National High Performance Computing Center (NHR@FAU)
[b] Department für Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2025

Stencil schemes

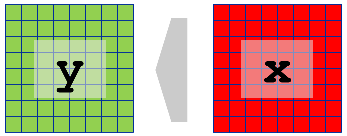
- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- but the regular access structure allows for matrix free coding

```
do iter = 1, maxit

      Perform sweep over regular grid: y ← x

      Swap y ←→ x

enddo
```



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

Case study: Jacobi stencil

The basics in two dimensions (2D)

Layer condition in 2D

From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

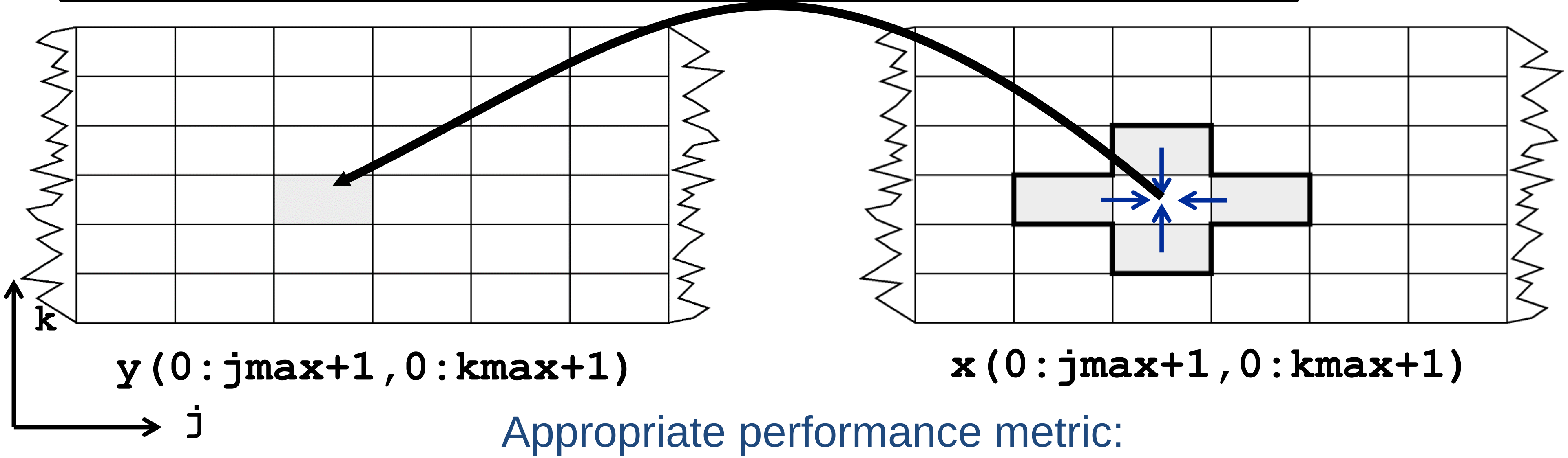
Jacobi-type 5-pt stencil in 2D

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)

2 arrays



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

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)

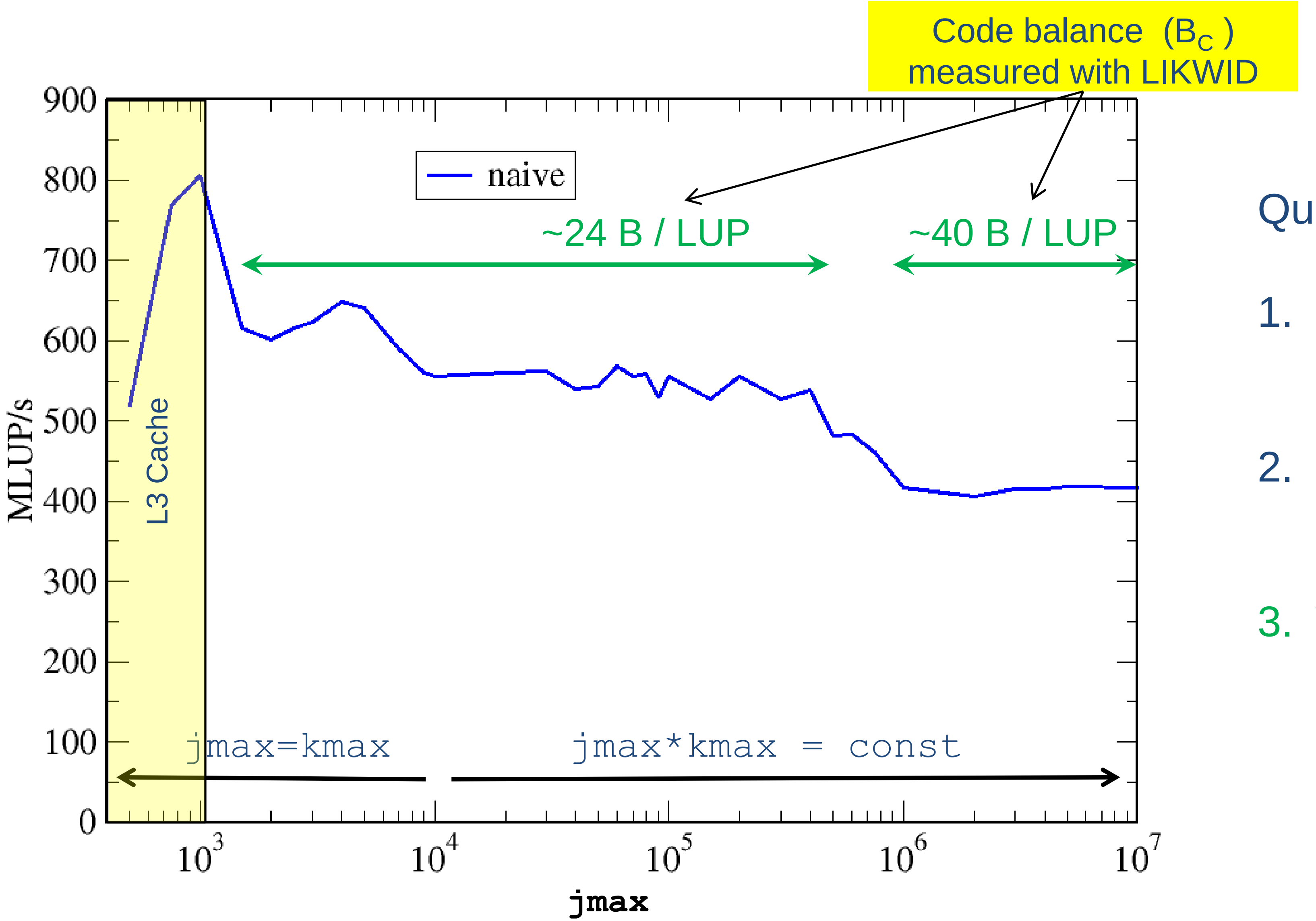
LD x(j,k+1)

Naive balance (incl. write allocate):

x(:, :) : 3 LD +
y(:, :) : 1 ST+ 1LD

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

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 > 10⁴`** ?
3. Why 24 B/LUP anyway???

Intel Compiler: ifort V13.1

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

Case study: Jacobi stencil

The basics in two dimensions (2D)

Layer condition in 2D

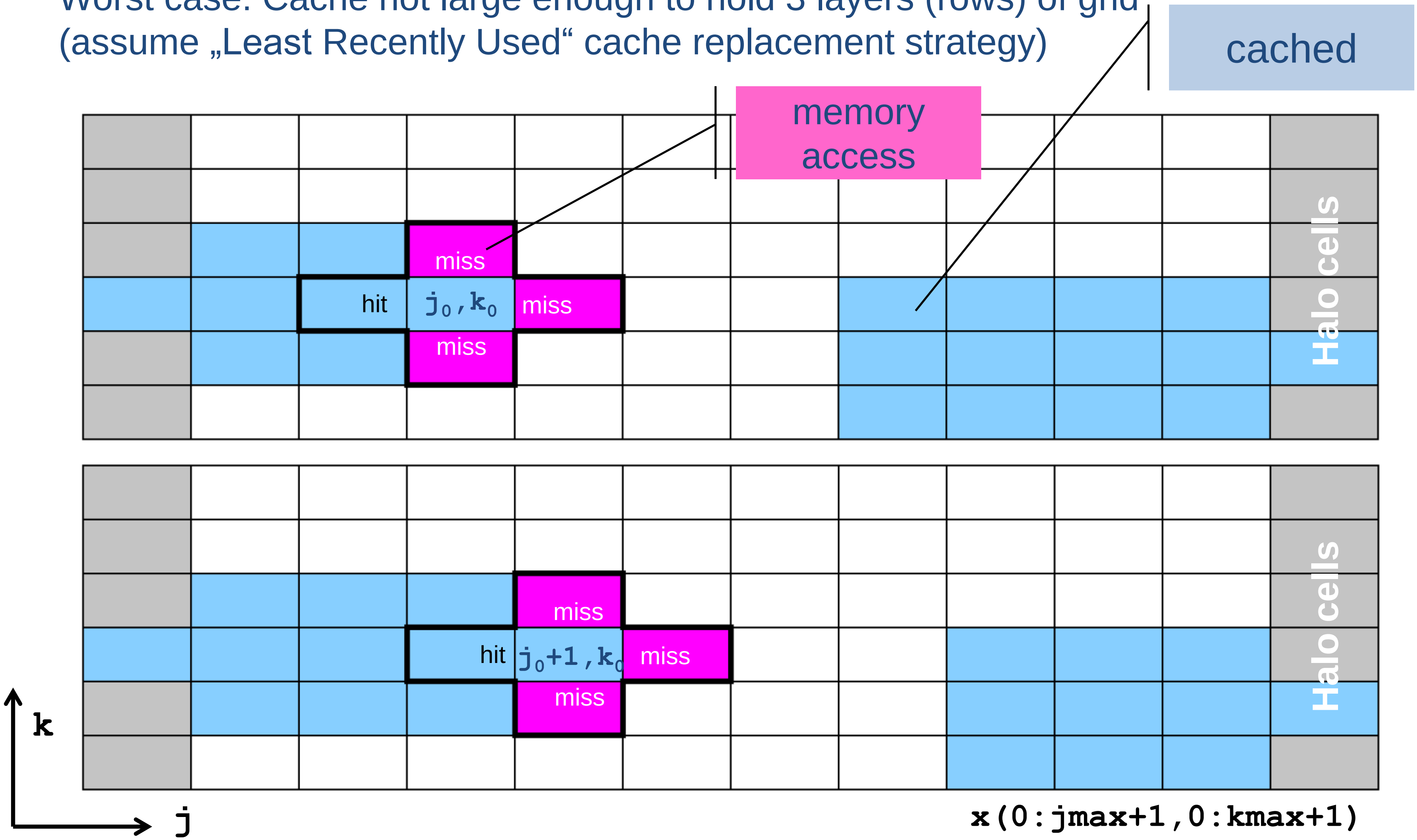
From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

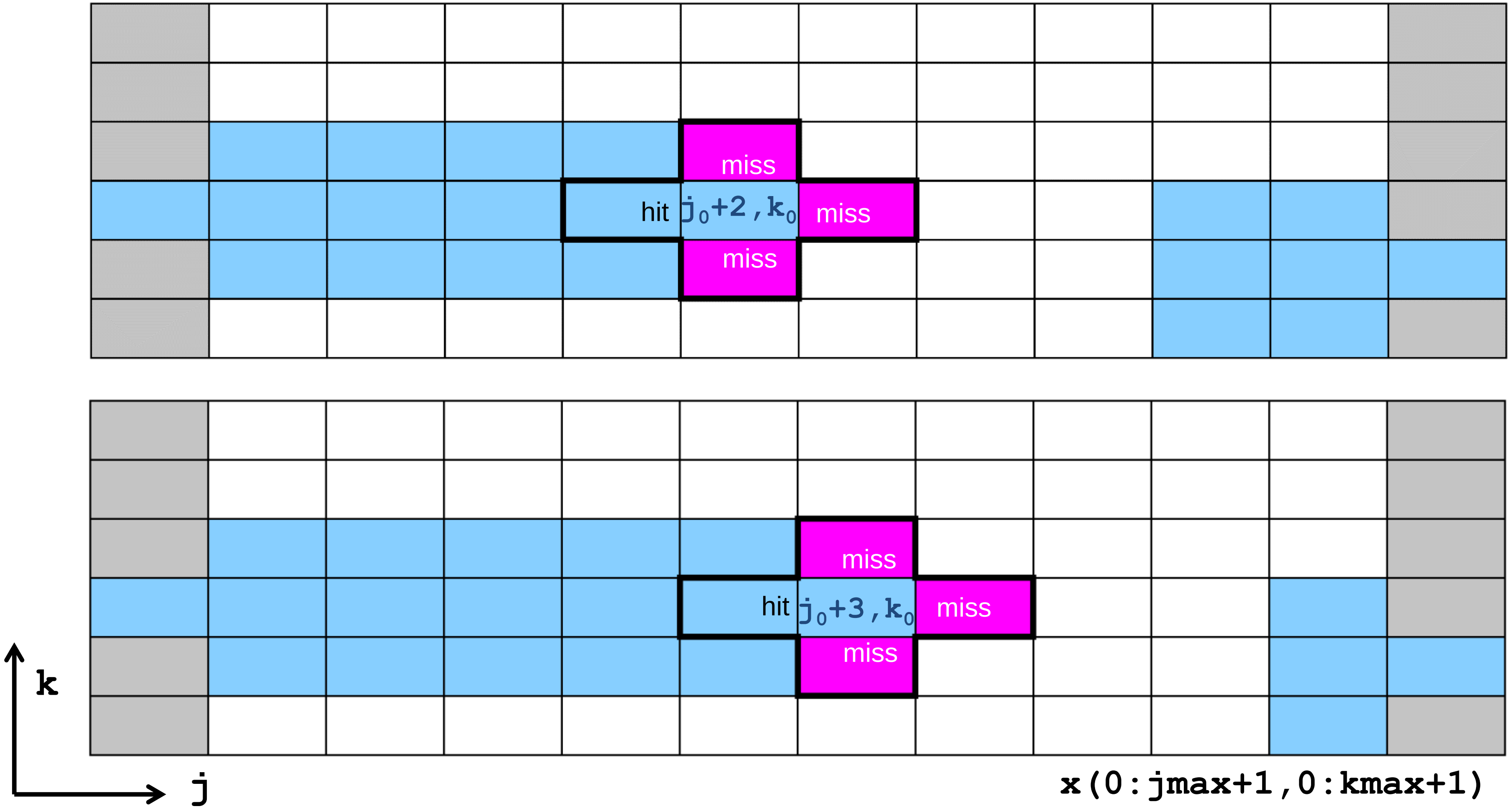
Analyzing the data flow

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



Analyzing the data flow

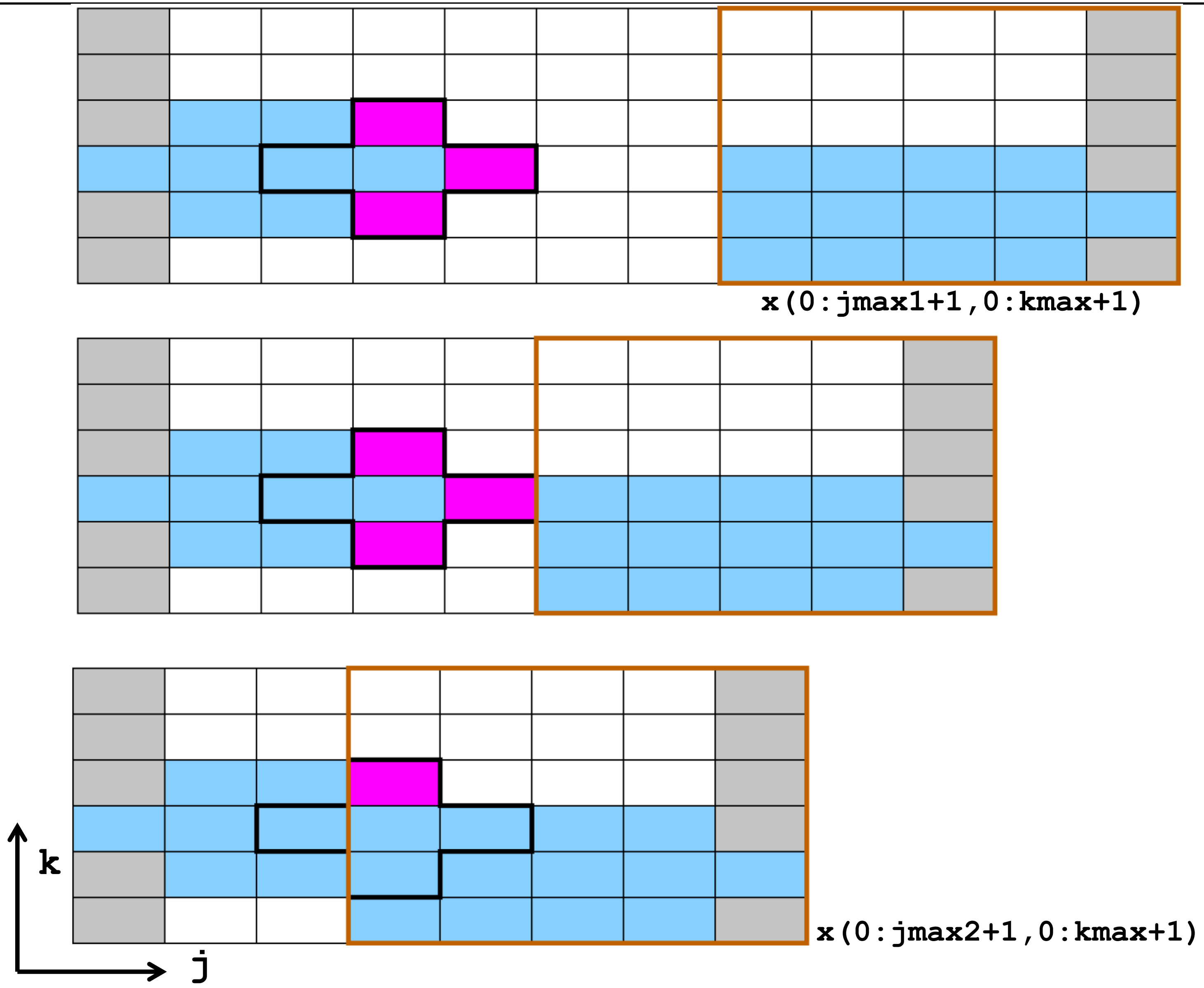
Worst case: Cache not large enough to hold 3 layers (rows) of grid
(+assume „Least Recently Used“ replacement strategy)



Analyzing the data flow

Reduce inner (j-) loop dimension successively

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

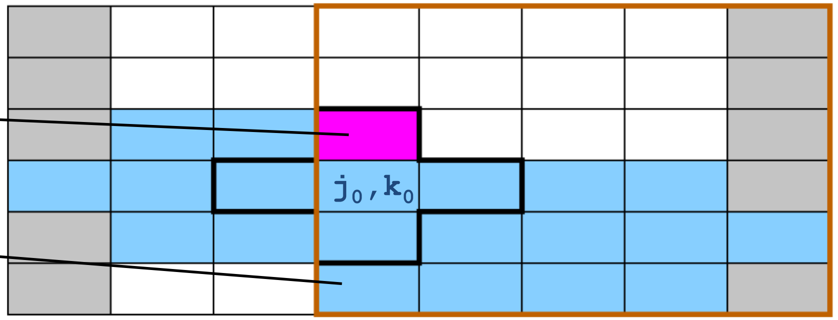


Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

$x(j_0,k_0+1)$

$x(j_0,k_0-2)$




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

3*jmax * 8B < CacheSize/2

"Layer condition"

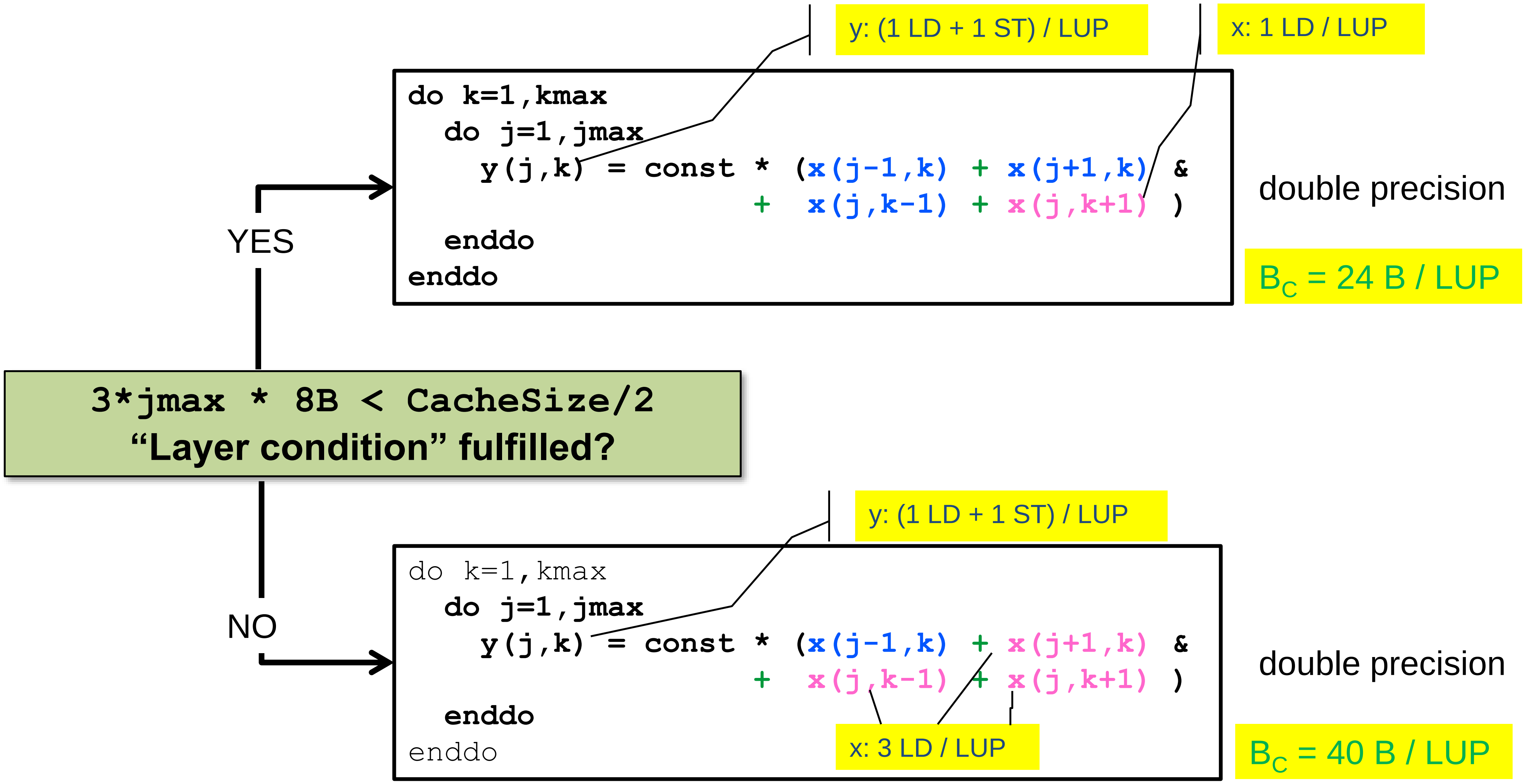
3 rows of **jmax**

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)
- Needs to be adapted for other stencils, e.g., in 3D, long-range, multi-array,…

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



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

- How to establish the 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 !      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
```

New layer condition (blocking)

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

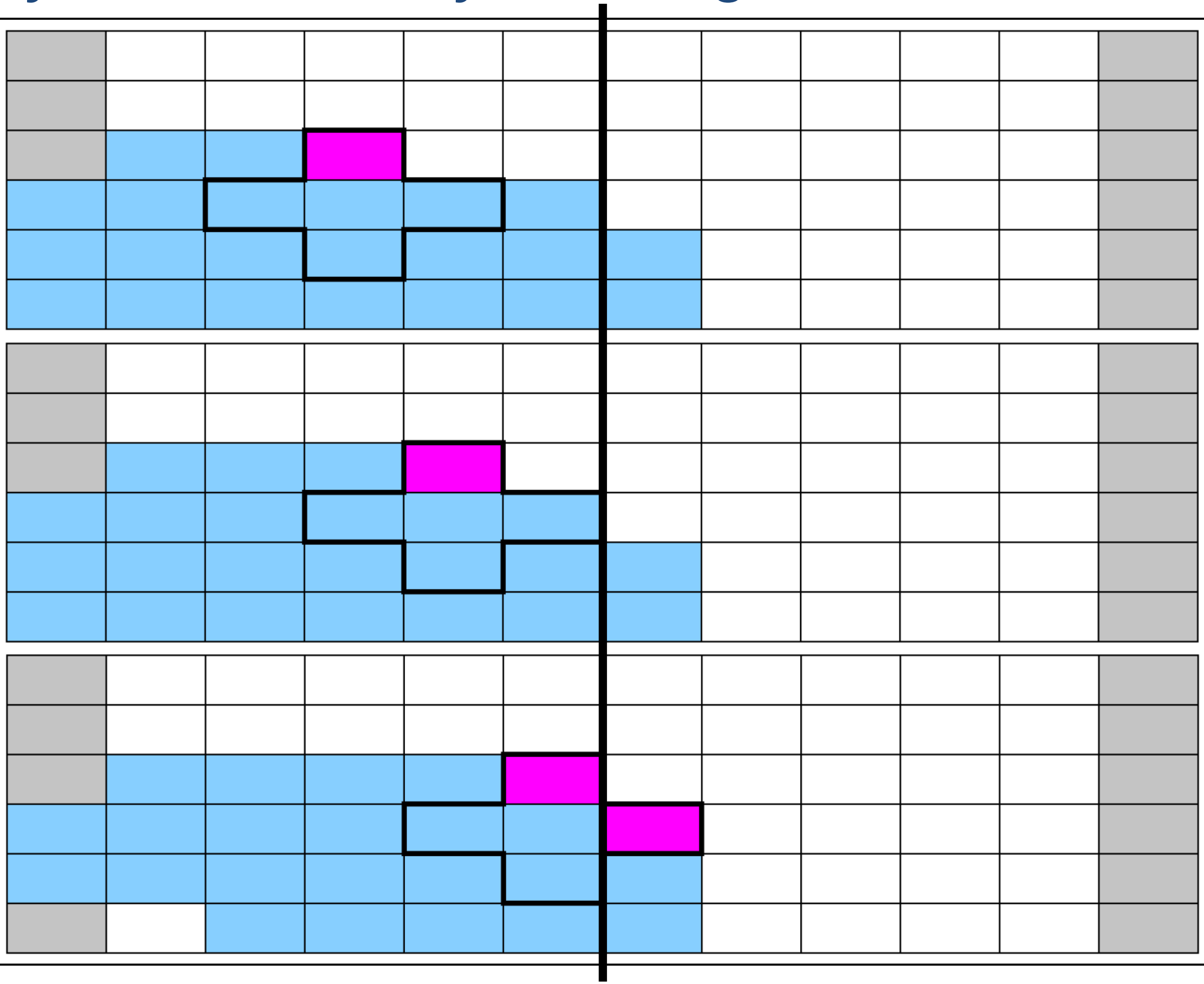
→ Determine for given **`CacheSize`** an appropriate **`jblock`** value:

`jblock < CacheSize / 48 B`

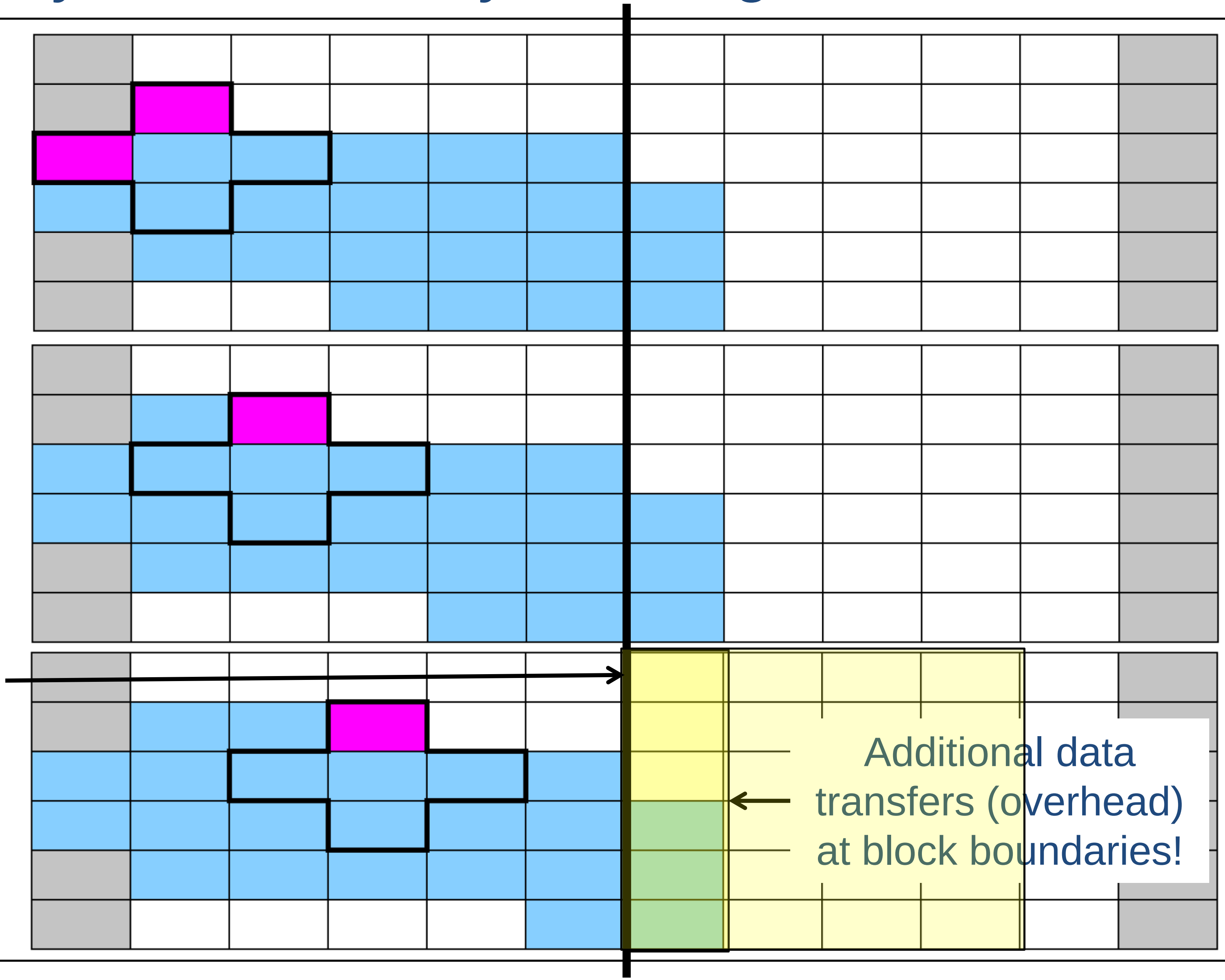
Establish the layer condition by blocking

Split domain into subblocks

e.g. block size = 5



Establish the layer condition by blocking



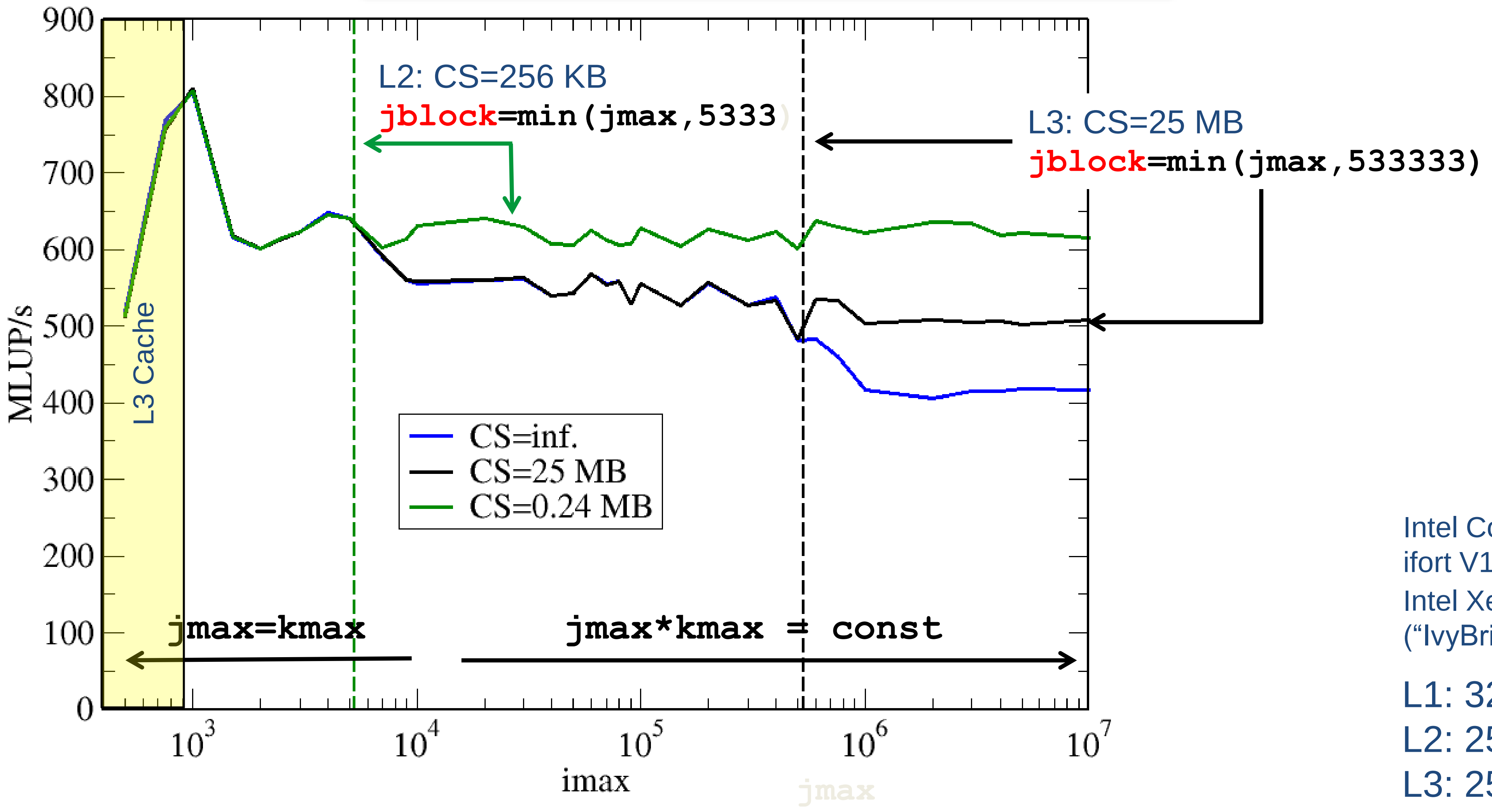
Short inner loop (jblock) may fool/trigger prefetchers

Additional data transfers (overhead) at block boundaries!

Establish layer condition by spatial blocking

`jblock < CacheSize / 48 B`

Which Cache to block for?

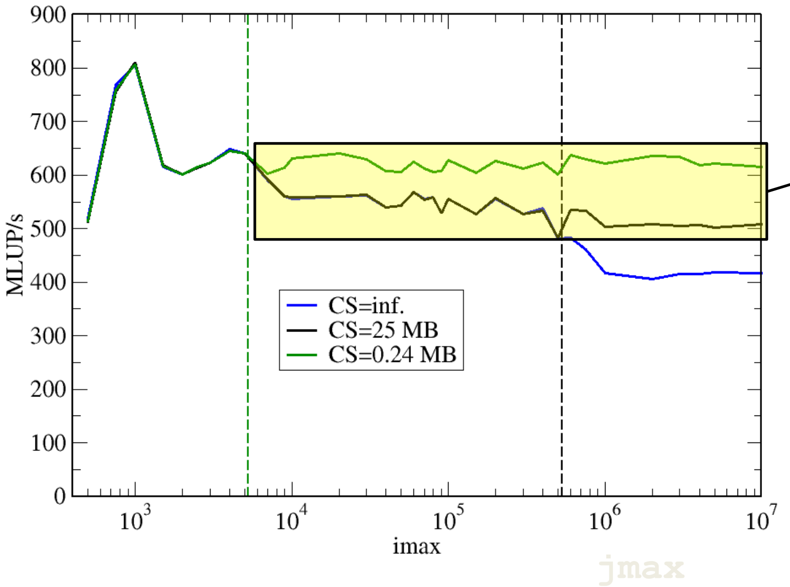


Intel Compiler
ifort V13.1

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

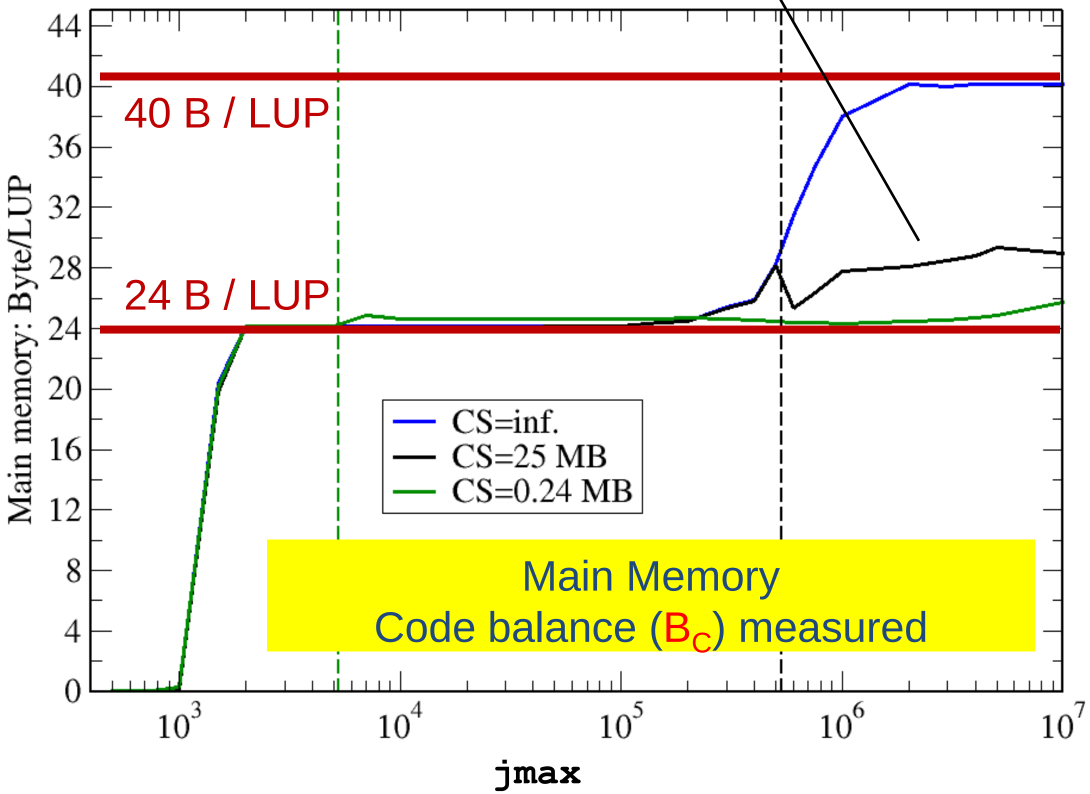
L1: 32 KB
L2: 256 KB
L3: 25 MB

Layer condition & spatial blocking: Memory Balance



Main memory access is not reason for different performance

Blocking factor (CS=25 MB) too large

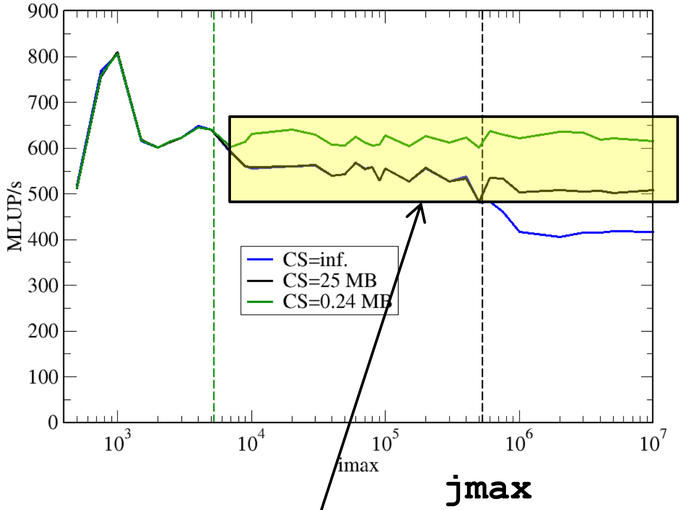


Main Memory
Code balance (B_C) measured

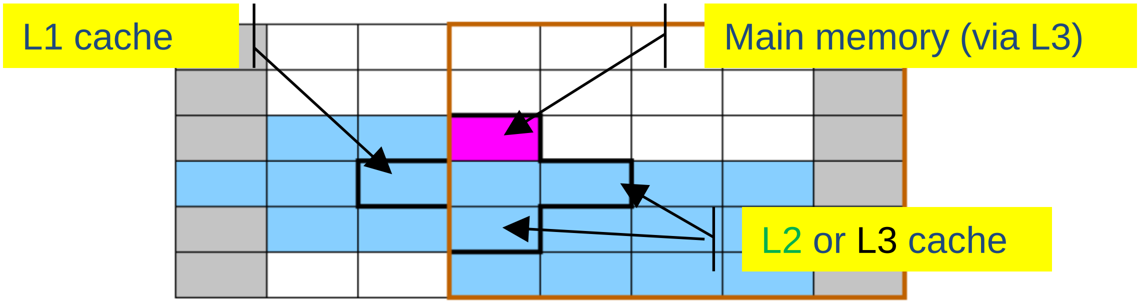
Intel Compiler
ifort V13.1

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

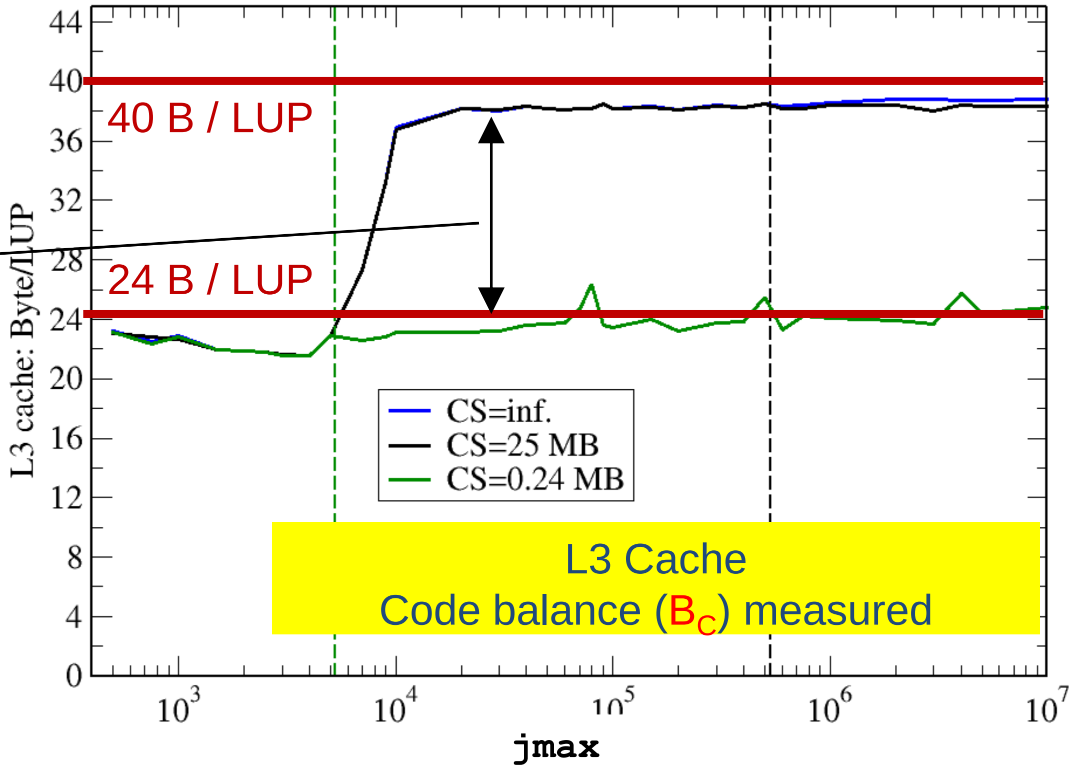
Layer condition & spatial blocking: L3 cache balance



Data accesses to L3 cache (blocking)



Impact of total L3 traffic:
24 B/LUP vs. 40 B/LUP

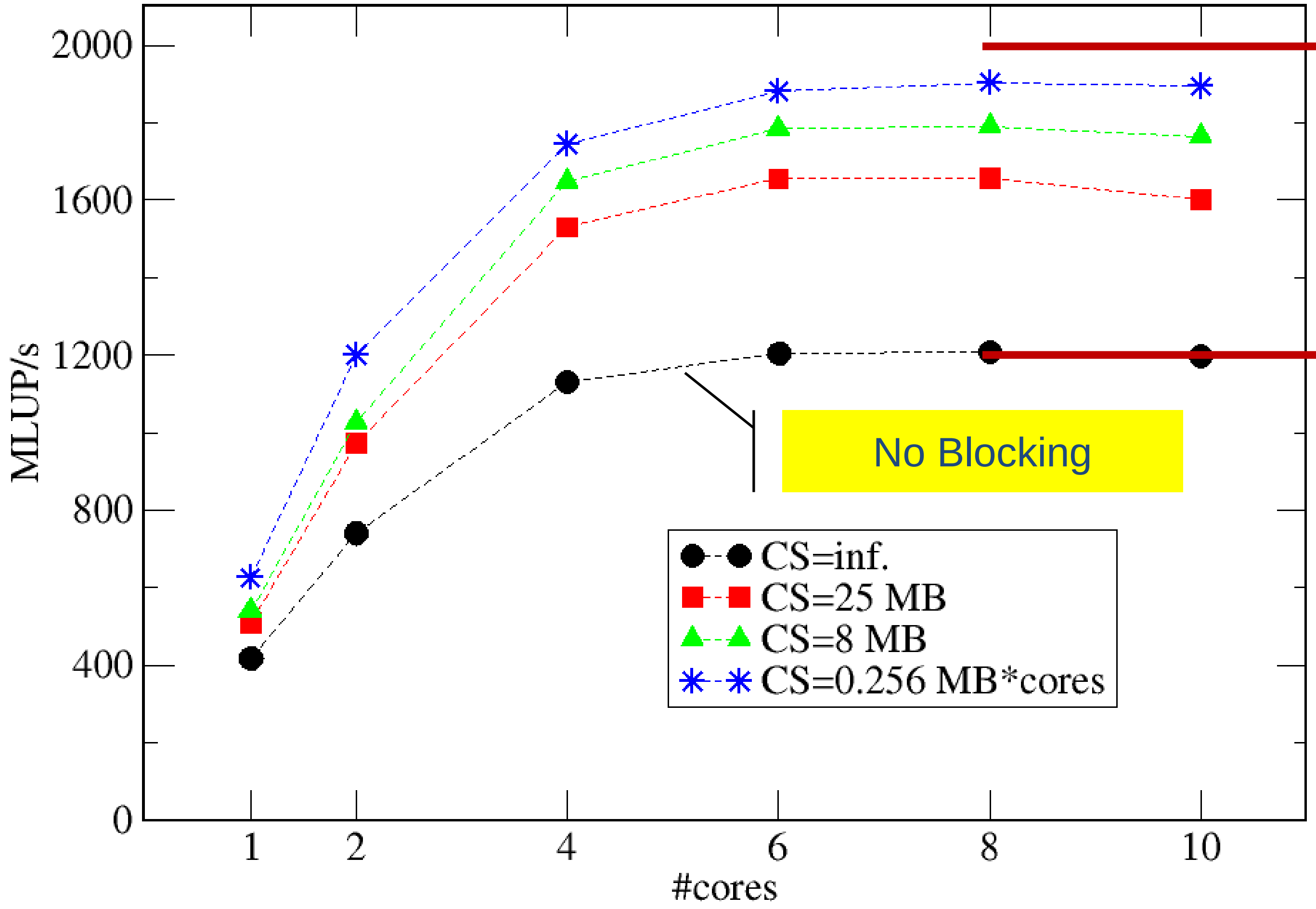


Intel Compiler
ifort V13.1

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

Socket scaling – Validate Roofline model

Full socket Roofline model: $P = \frac{b_S}{B_C^{mem}} = I * b_S$



B_C= 24 B/LUP

B_C= 40 B/LUP

Intel Compiler
ifort V13.1
OpenMP Parallel

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

$$\boldsymbol{b_S = 48\ \mathrm{GB/s}}$$

Layer condition changes for #cores>1 (see later)

Case study: Jacobi stencil

The basics in two dimensions (2D)

Layer condition in 2D

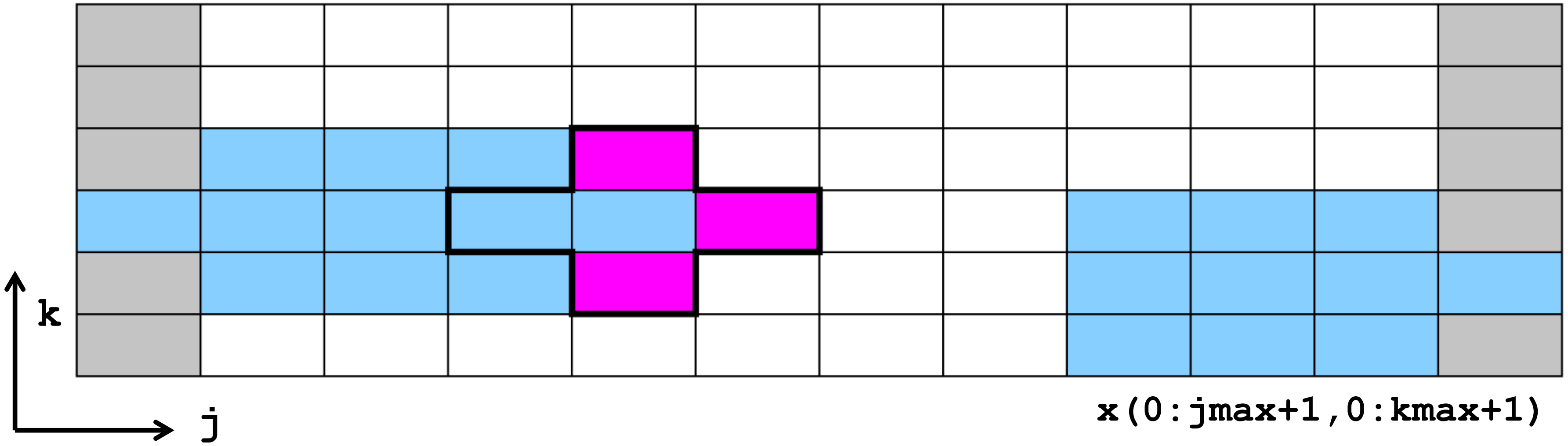
From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

From 2D to 3D

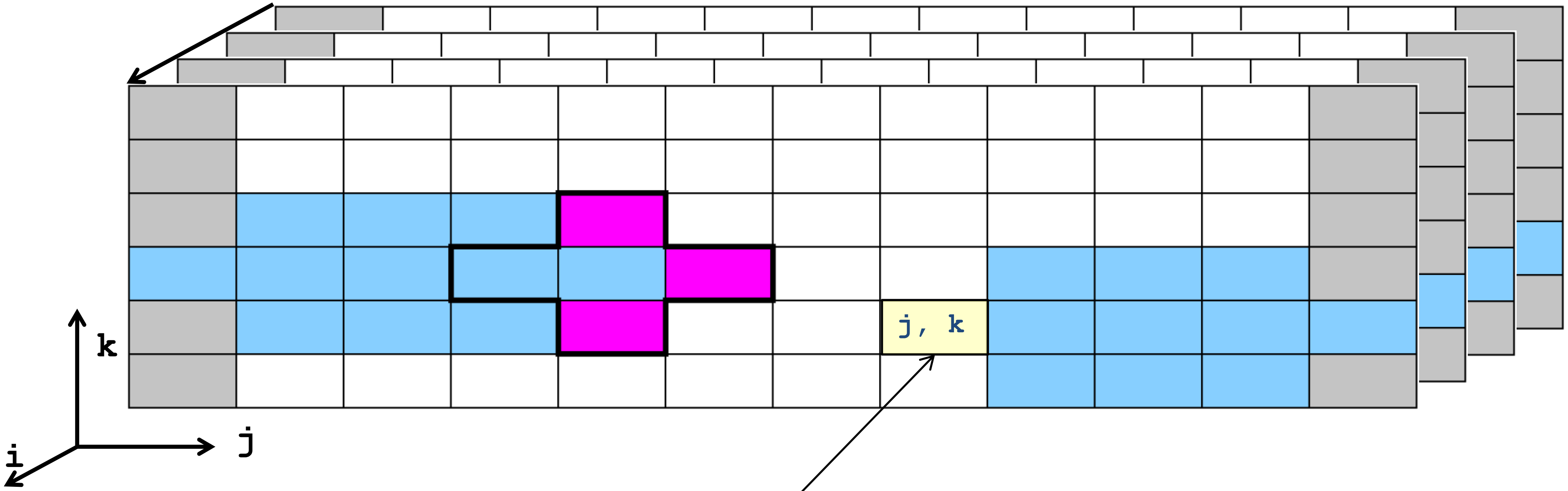
- 2D



- 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 slab 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
```

double precision

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
```

double precision

B_C = 24 B / LUP

3D 7-pt Jacobi-type 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?

Case study: Jacobi stencil

The basics in two dimensions (2D)

Layer condition in 2D

From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

Jacobi Stencil – OpenMP parallelization (I)

```
!$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
```

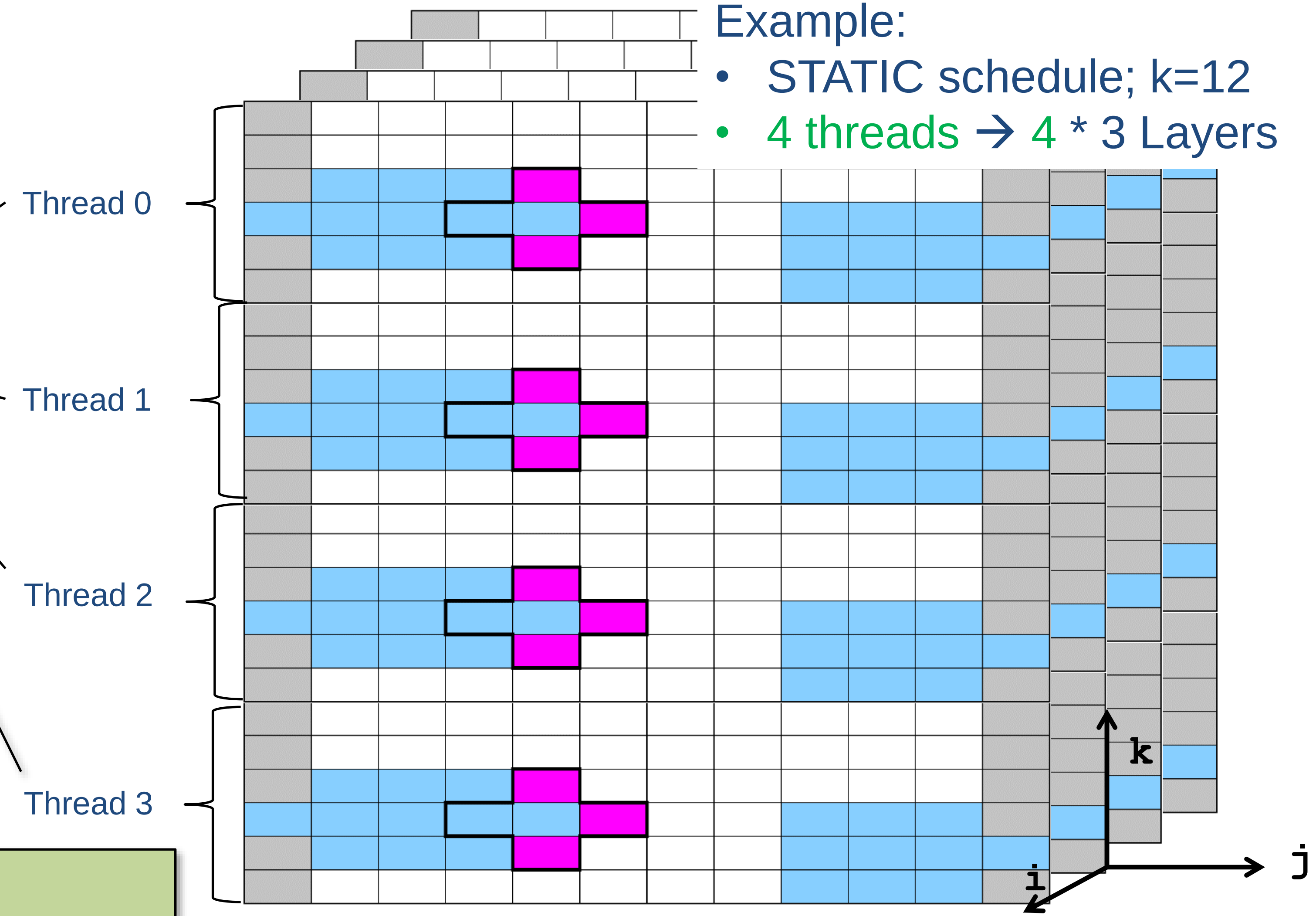
Basic guideline:
Parallelize outermost loop

SCHEDULE(STATIC)
Equally large chunks in k-direction

Jacobi Stencil – OpenMP parallelization (I)

Equally large chunks per thread in k-direction

Layer condition" to be fulfilled by each thread



Example:

- STATIC schedule; k=12
- 4 threads → 4 * 3 Layers

Note: Threads do not need to be in sync – they only sync at the end of the sweep

"Layer condition":

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

Jacobi Stencil – OpenMP parallelization (I)

Impact of parallelizationon **CacheSize**: core-local vs. shared caches

- CS_t is the available cache size for holding data when using `nthreads` (=cores)

  "Layer condition" (LC): `nthreads *3*jmax*imax*8B < CSt/2`

- If CS is **shared cache** (e.g. L3): **CS_t = constant**
  → LC more stringent at higher thread counts

  `L3-CacheSize=25 MB`
  1 thread: `imax=jmax < 720`
  10 threads: `imax=jmax < 230`

- If CS is core-local cache (e.g. L2): **CS_t = constant *** `nthreads`
  → LC independent of thread count

  `L2-CacheSize=256 KB`
  1 thread: `imax=jmax < 70`
  10 threads: `imax=jmax < 70`

Jacobi Stencil – OpenMP parallelization (I)

“Layer condition”: `nthreads *3*jmax*imax*8B < CS_t/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
```

double precision

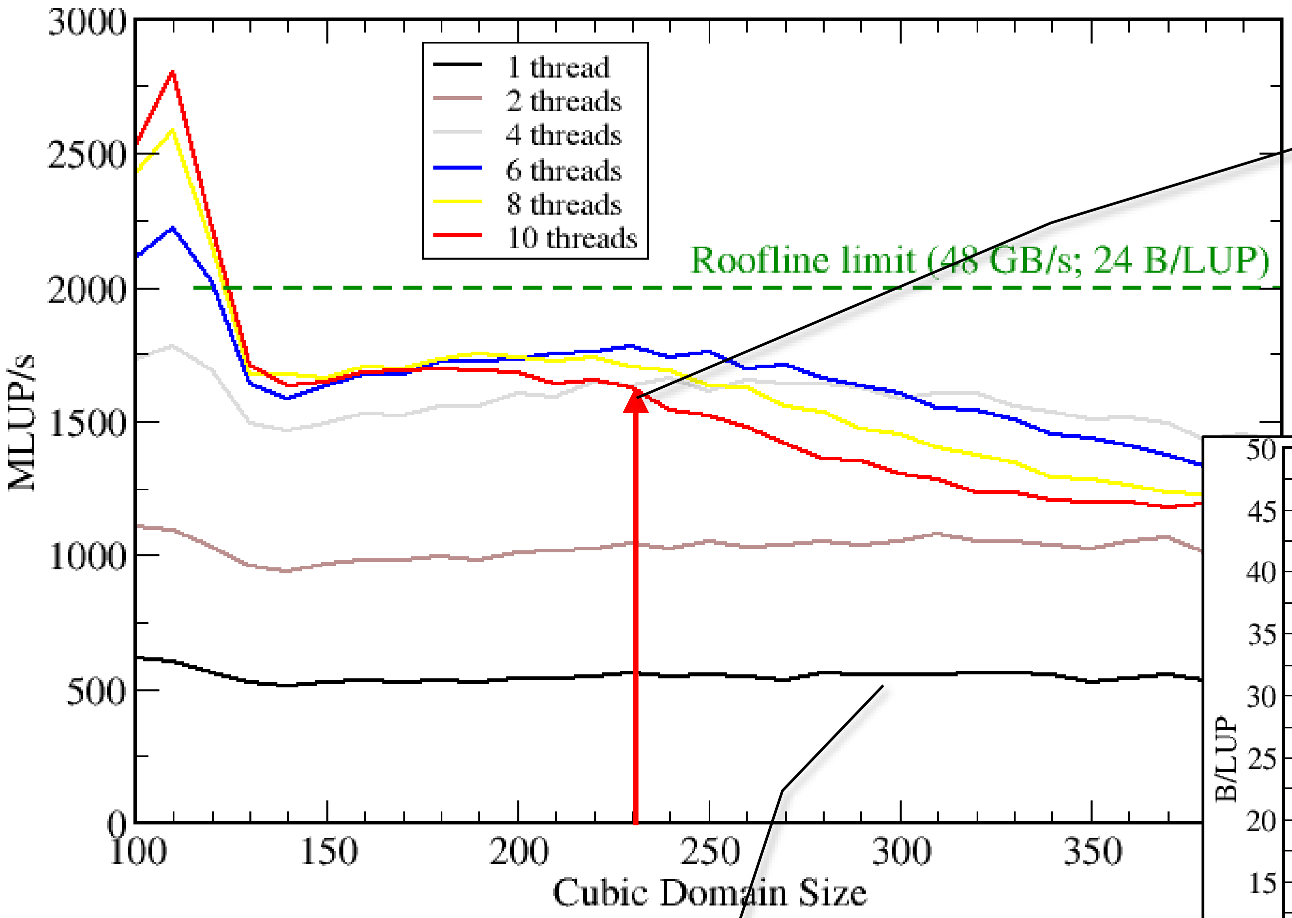
B_C = 24 B / LUP

Roofline model:
$P = b_S / B_C^{mem}$

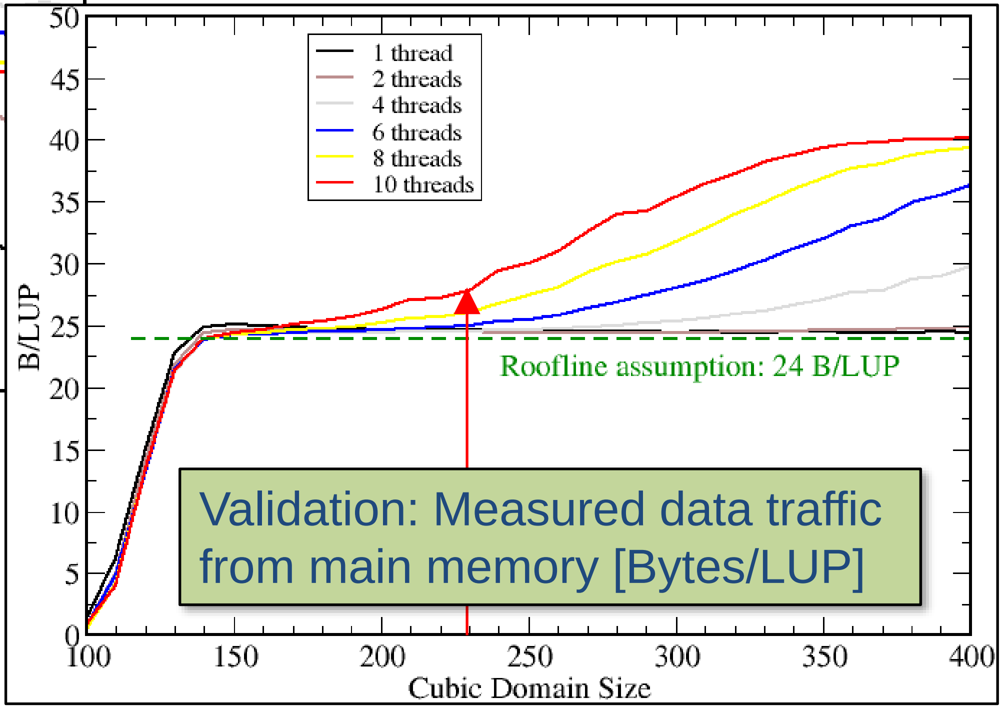
Intel® Xeon® Processor E5-2690 v2
10 cores@3 GHz
CacheSize = 25 MB (L3; shared)
$b_S = 48\ \mathrm{GB/s}$

Best performance:
$P = 2000\ \mathrm{MLUP/s}$

Jacobi Stencil – OpenMP parallelization (I)



10 threads: performance drops around `imax=230`



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

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

Jacobi Stencil – OpenMP parallelization (I)

Original “Layer condition” does not hold: `nthreads*3*jmax*imax*8B > CS_t/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)
```

→ $B_C = 40\ \mathrm{B/LUP}$

→ $P = 1200\ \mathrm{MLUP/s}$

Jacobi Stencil – OpenMP & 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))

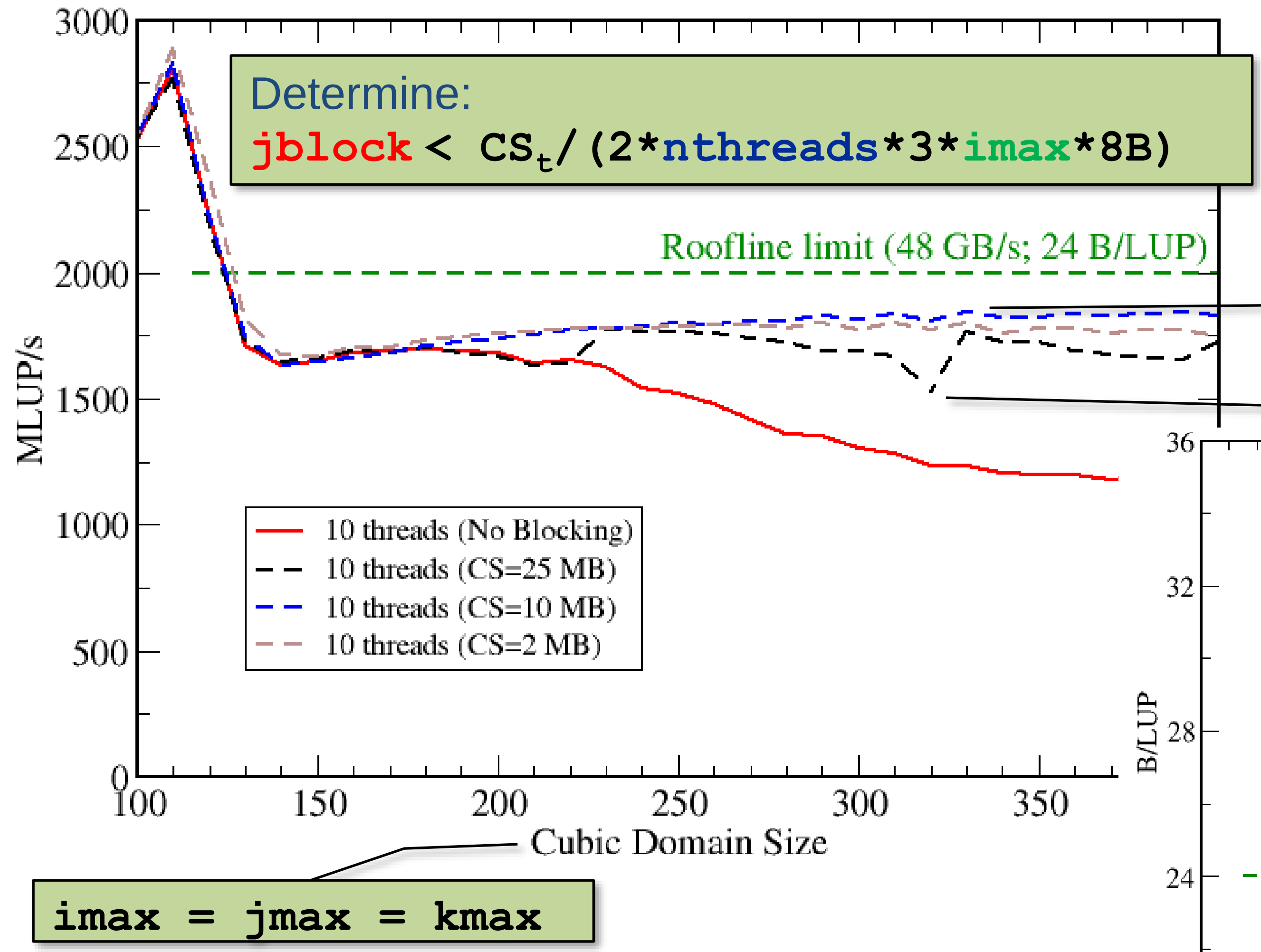
$nthreads*3*jblock*imax*8B < CS_t/2$

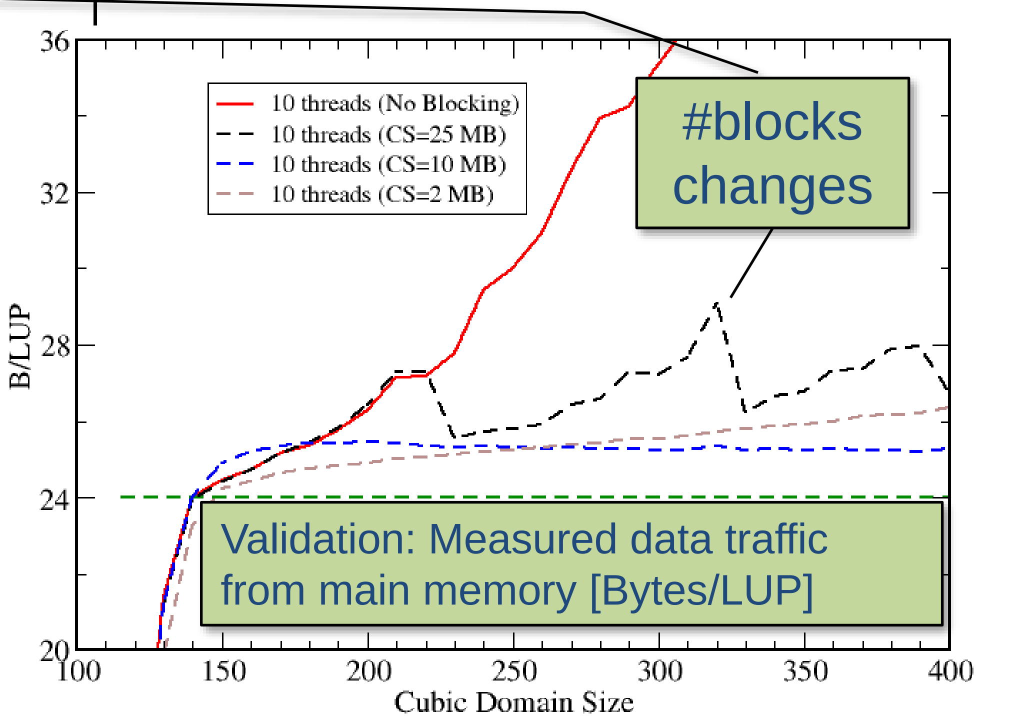
Ensure layer condition by choosing jblock approriately (Cubic Domains):

$jblock < CS_t/(imax * nthreads * 48B)$

→ Back to prediction of $P = 2000\ MLUP/s$

Jacobi Stencil – OpenMP & spatial blocking





Impact of blocking factor jblock

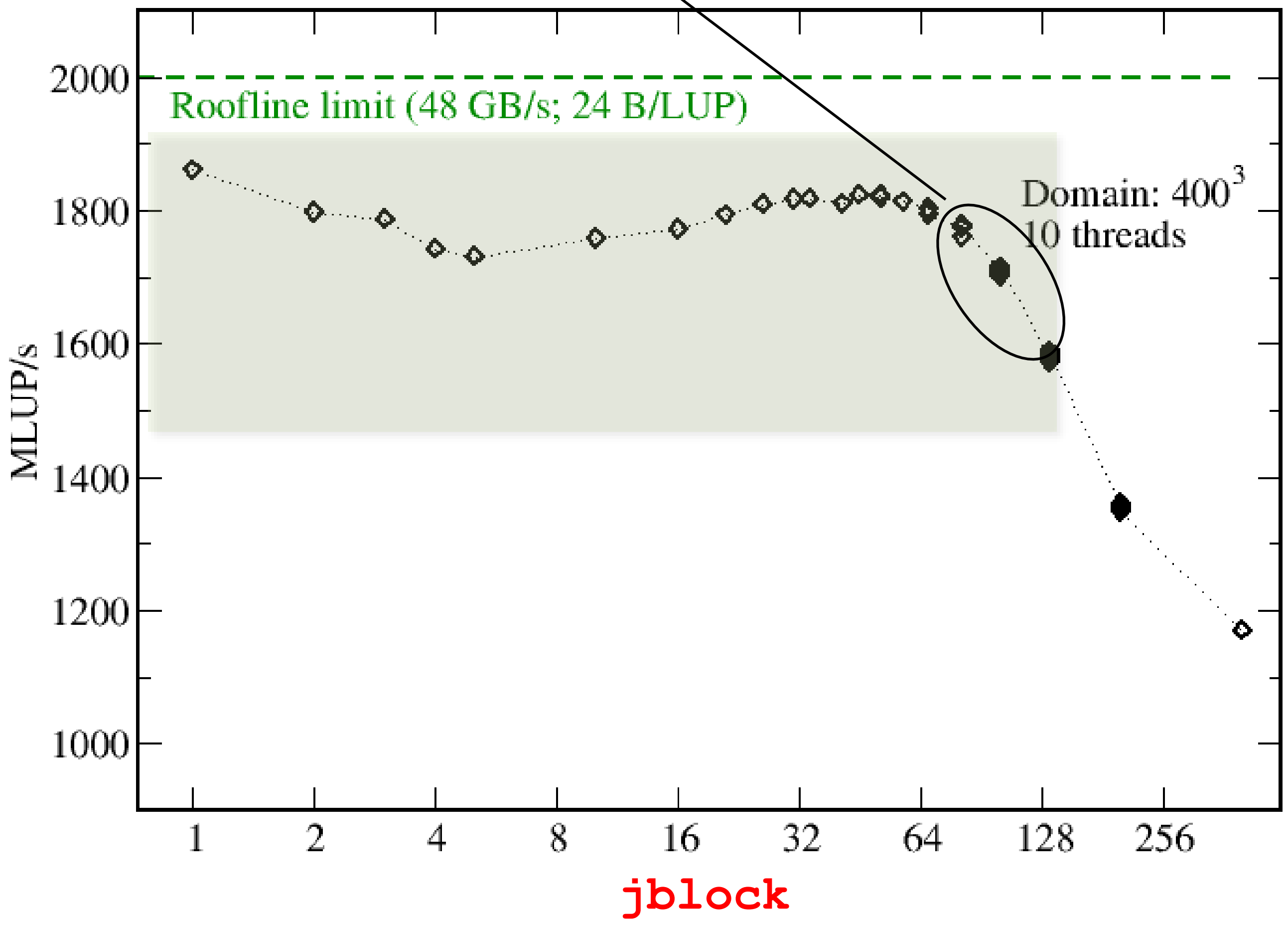
Layer condition estimates appropriate jblock:
jblock < CS_t/(2*nthreads*3*imax*8B)

CS_t=25 MB
nthreads=10
imax=400

jblock < 130

jblock=32,…,64
useful choices

Layer condition a bit too optimistic

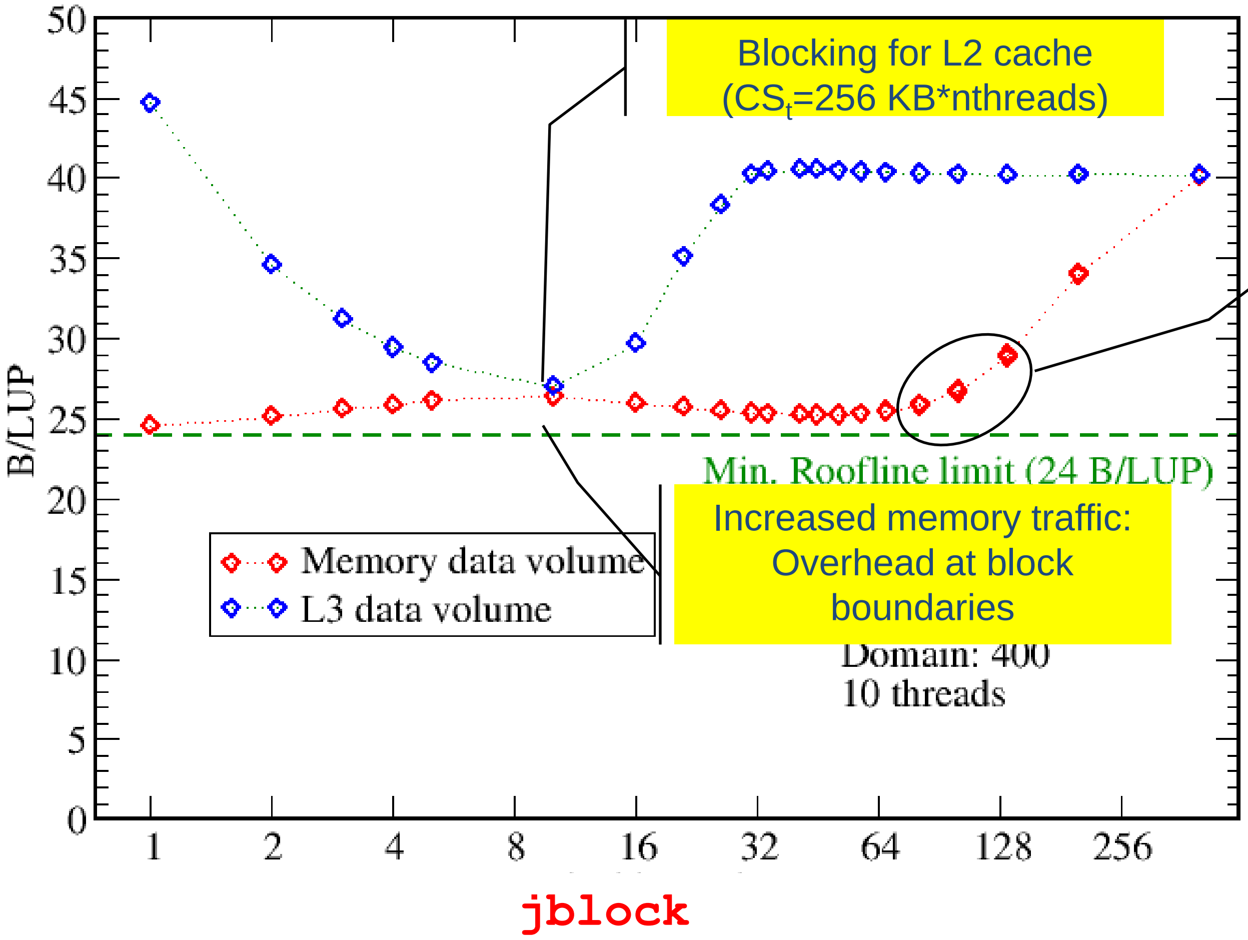


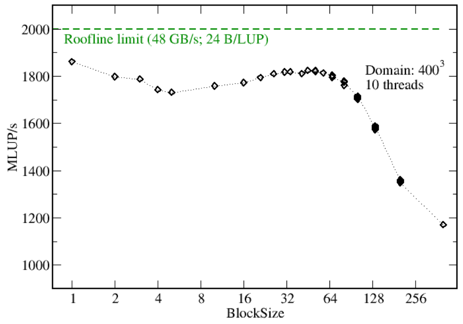
Impact of jblock: Data traffic from L3 & main memory

Layer condition estimates appropriate jblock:

jblock < CS_t/(2*nthreads*3*imax*8B)

jblock < 130





Jacobi Stencil – what about j-loop parallelization

```
!$OMP PARALLEL PRIVATE(k)
do k=1,kmax
!$OMP DO SCHEDULE(STATIC)
  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
!$OMP END DO
enddo
!$OMP END PARALLEL
```

All threads work on the same 3 k-layers at the same time

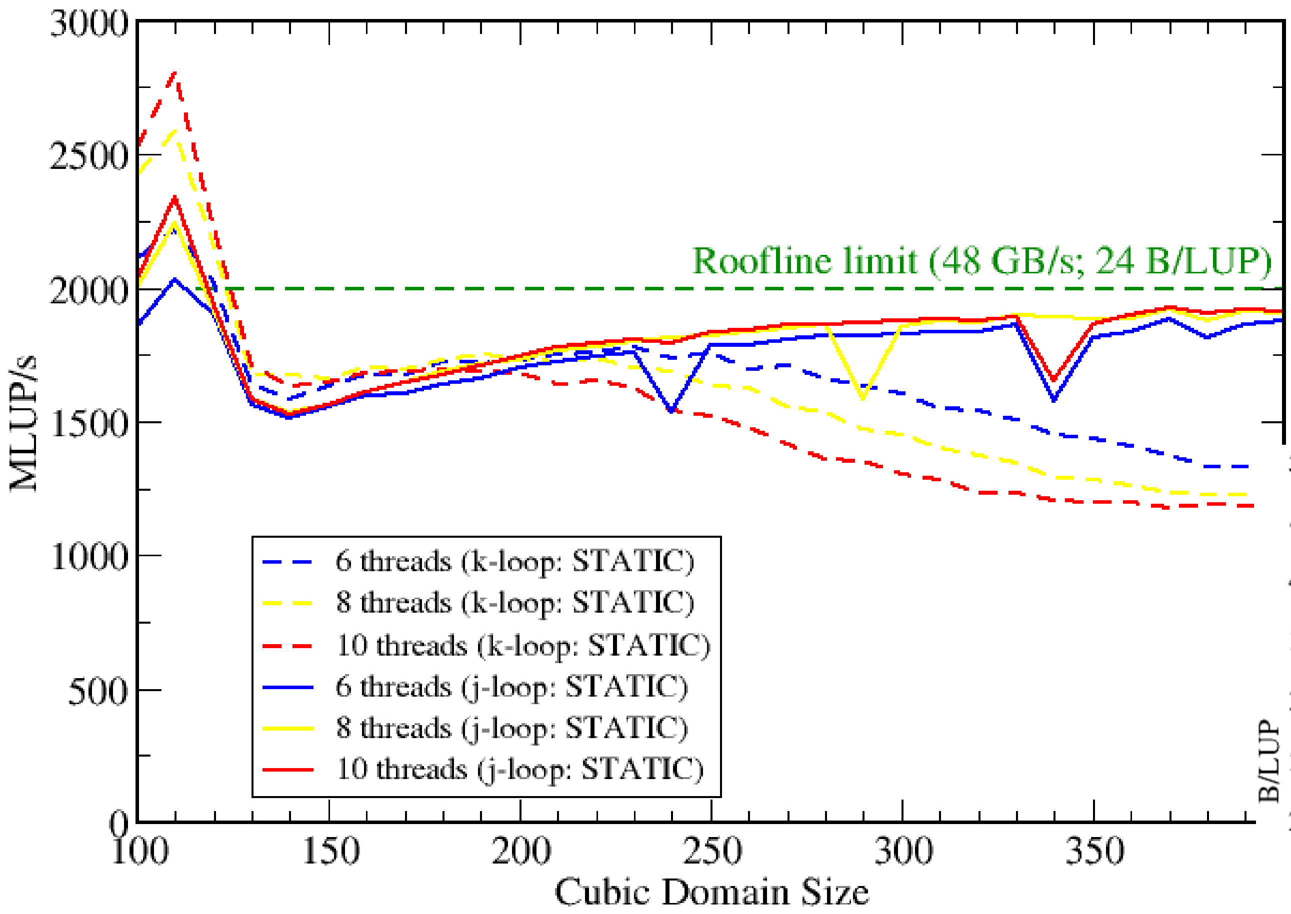
"Layer condition" (j-loop parallel; schedule=static)
`3*jmax*imax*8B < CS_t/2`

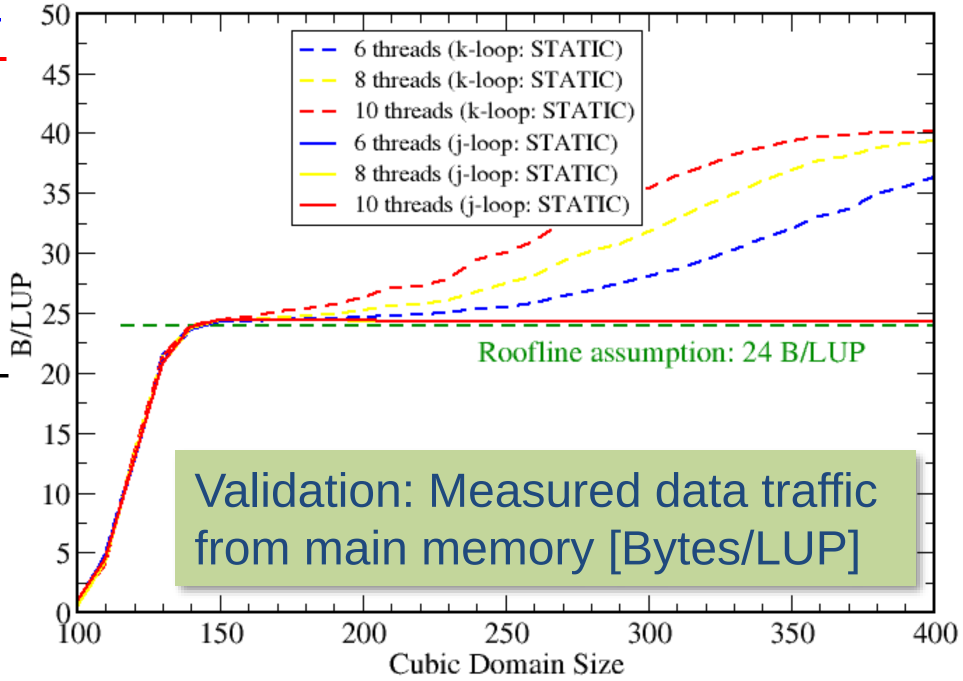
Layer condition is independent of number of threads (`imax<720` for test system)

Impact of parallelization overhead?!
- 1 barrier each `jmax * imax` LUPs
- Costs / barrier: approx. `1 500 cycles`
- Computational time / barrier: (`jmax * imax LUPs)/maxMLUPs`

  `e.g. jmax=imax=200 & maxMLUPs=2000 MLUPs` → $20 * 10^{-6}$ s
  → `60 000 cycles`

Jacobi Stencil – what about j-loop parallelization





Performance will drop at domain sizes ~500,..,700

Case study: Jacobi stencil

The basics in two dimensions (2D)

Layer condition in 2D

From 2D to 3D

OpenMP parallelization strategies and layer condition in 3D

NT stores

Jacobi Stencil – can we further improve?

```
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
```

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

Total data transfer / LUP:

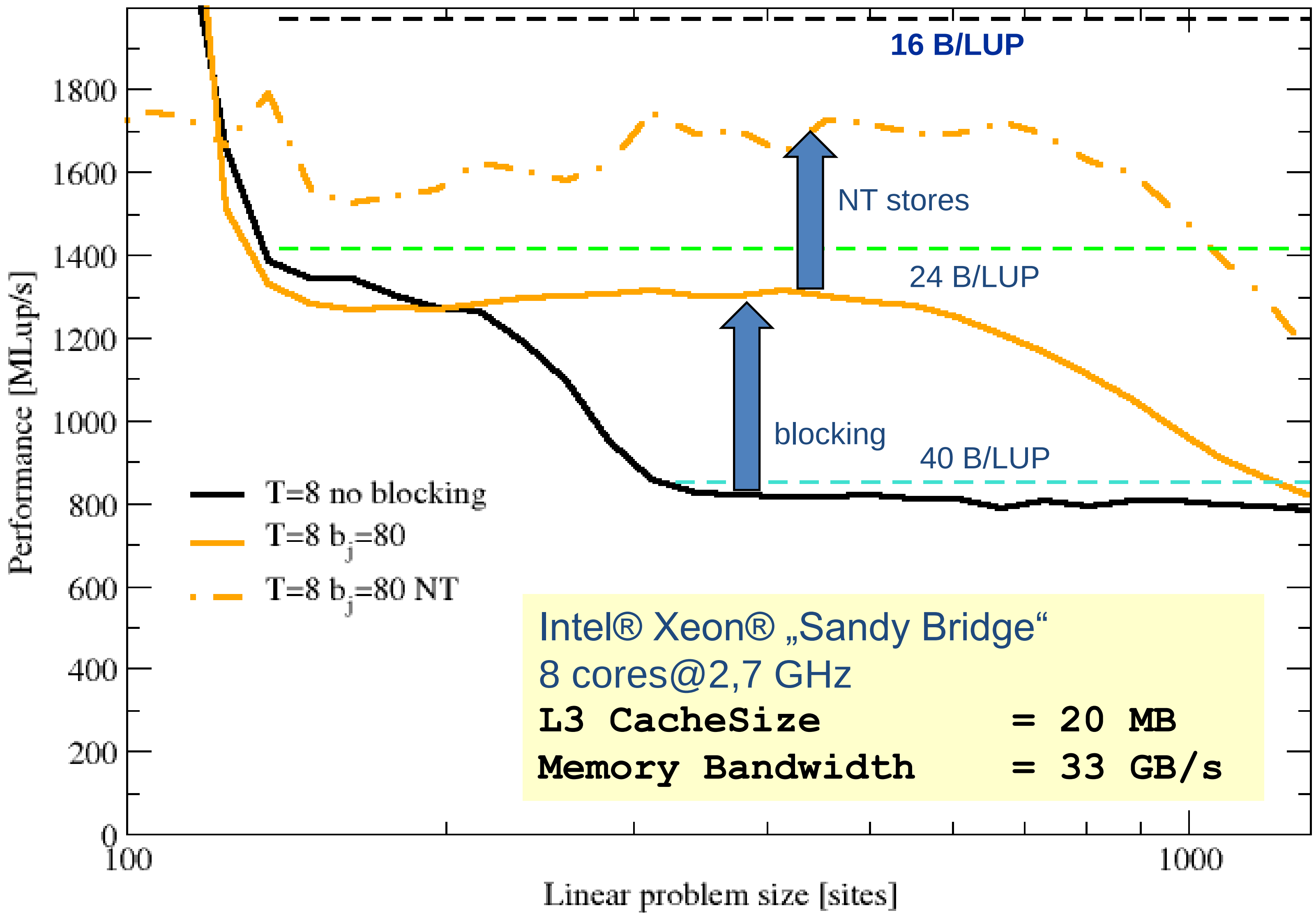
```
 (8+8) B/LUP for y()  (ST+Write Allocate)
+    8 B/LUP for x(i,j,k+1)
→ 24   B/LUP
```

Use NT-stores to avoid "Write Allocate"

Total data transfer / LUP:

```
    8  B/LUP for y()  (NT-STore)
+   8  B/LUP for x(i,j,k+1)
→ 16   B/LUP
```

Jacobi Stencil – Blocking + NT-stores



Conclusions from the Jacobi example

- Data transfer analysis is key to structured performance modeling and optimization
- Avoiding slow data paths == re-establishing the most favorable layer condition
- Optimal blocking factor for spatial blocking can be estimated
  - Be guided by the cache size the layer condition → does not depend on outer loop!
  - Blocking: overhead at block boundaries → Largest cache first choice
  - No need for exhaustive scan of “optimization space”
- Performance optimization by re-establishing the layer condition was guided and validated by (roofline-like) model
- Modeling and optimizing “Jacobi” stencil is blueprint for many stencil(-like) kernels.

Open topics – layer condition: 3D (outer loop parallel)

Derived for
`OMP_SCHEDULE=STATIC`
→ Does it change for other scheduling strategies?

nthreads *3*jmax*imax* 8B < CS$_t$/2

What about "square blocking":
`jmax` → **`jblock`** &
`imax` → **`iblock`** ?

What about i-blocking:
`imax` → **`iblock`** ?

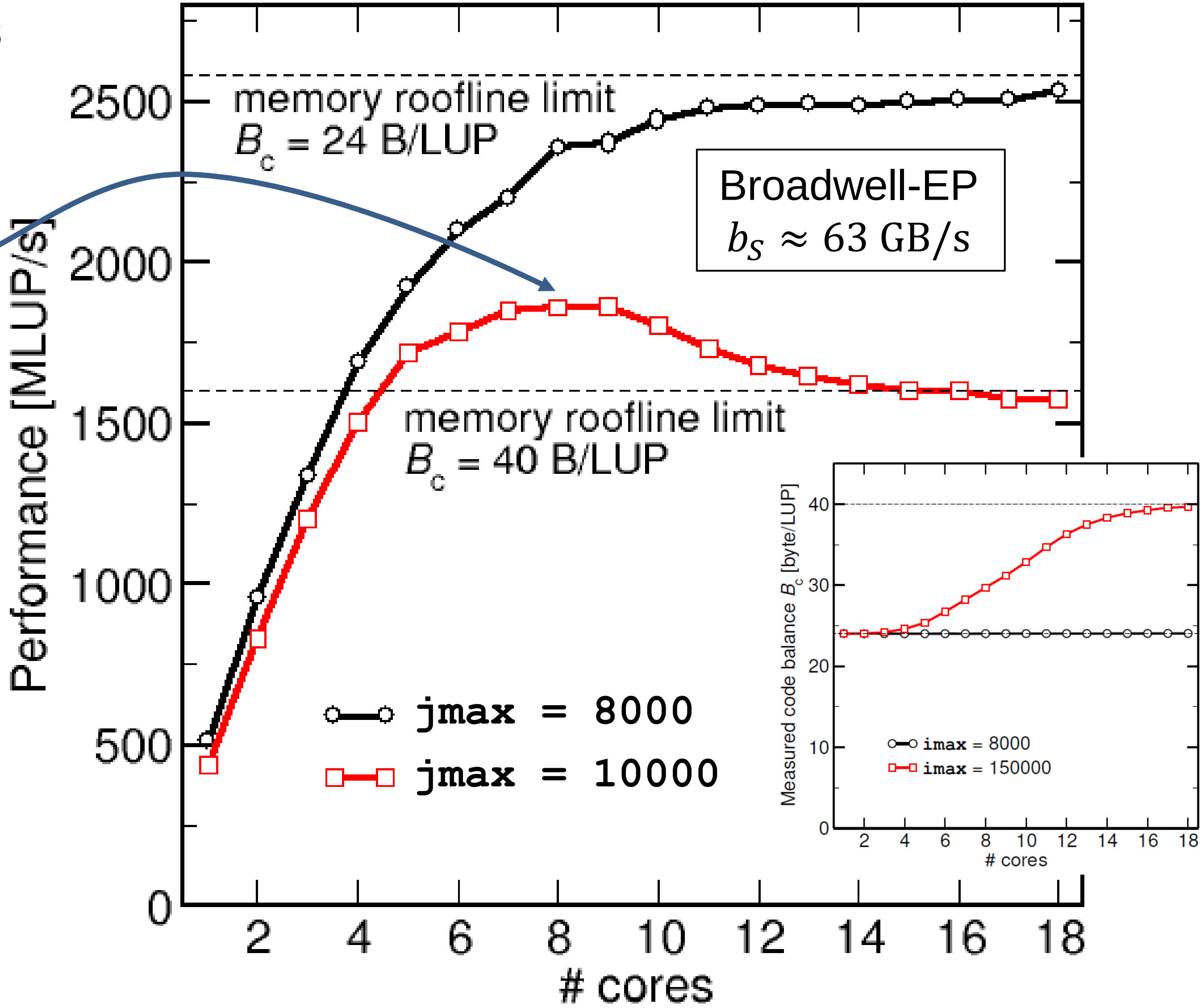
OpenMP parallelization and blocking for a shared cache

Layer conditions make for interesting effects

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

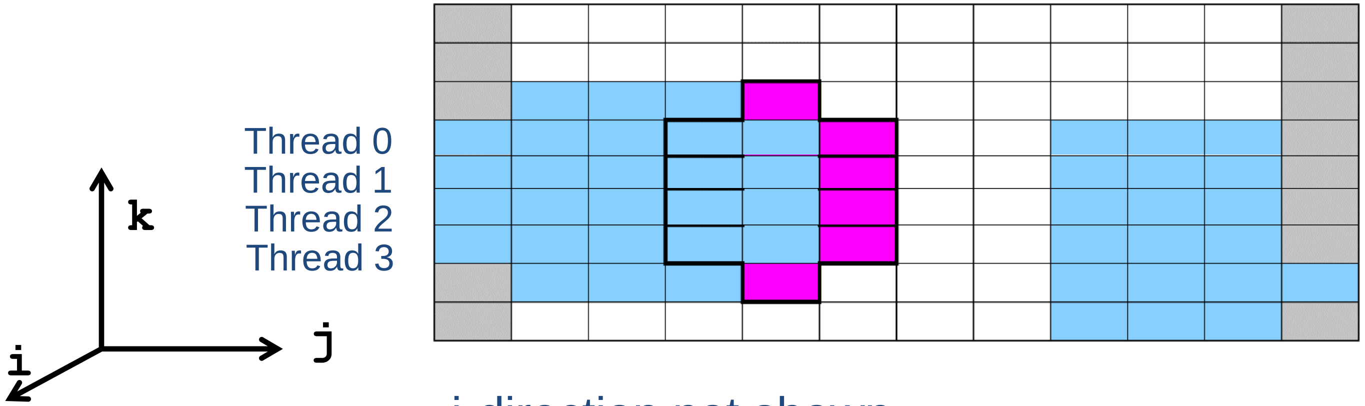
- Solutions
  1. Choose small enough block or domain size
     - Layers either small enough to fit in core-private caches *or*
     - Shared cache big enough to hold all layers for all threads
  2. Adaptive blocking for shared cache:

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



Open topics – layer condition & outer loop parallel w/ STATIC,1 schedule

```
!$OMP PARALLEL DO SCHEDULE(STATIC,1)
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
```



i-direction not shown

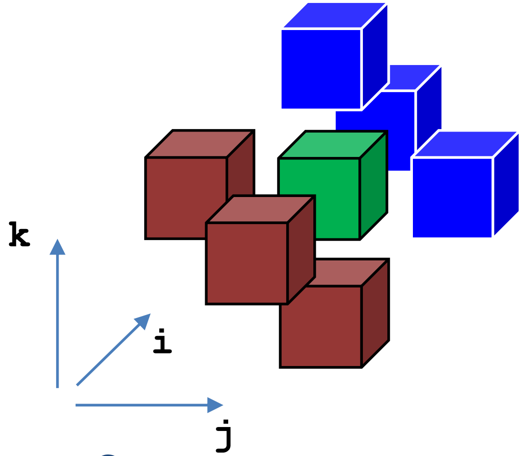
What is the relevant layer condition here???

Case study: Parallelizing a Gauss-Seidel Solver

3D matrix-free Gauss-Seidel smoother

- Matrix-free iterative solver for $Ax = b$

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)} + b_i\right)$$



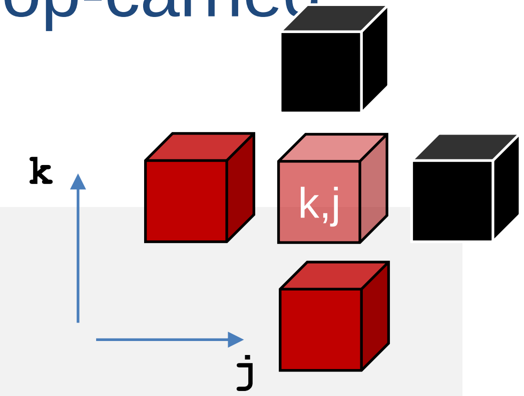
- Here used for Dirichlet boundary value (PDE) problem $\Delta x = 0$

```
for(it=0; it<itmax; ++it) { // or convergence check
  for(k=1; k<kmax-1; ++k) {
    for(j=1; j<jmax-1; ++j) {
      for(i=1; i<imax-1; ++i) {
        x[k][j][i] = ( x[k][j][i-1] + x[k][j][i+1]
                     + x[k][j-1][i] + x[k][j+1][i]
                     + x[k-1][j][i] + x[k+1][j][i])/6.0;
} } } }
```

“new data” “old data”

OpenMP parallelization?

- Naïve OpenMP loop parallelzation impossible due to loop-carried dependency on all spatial loop levels

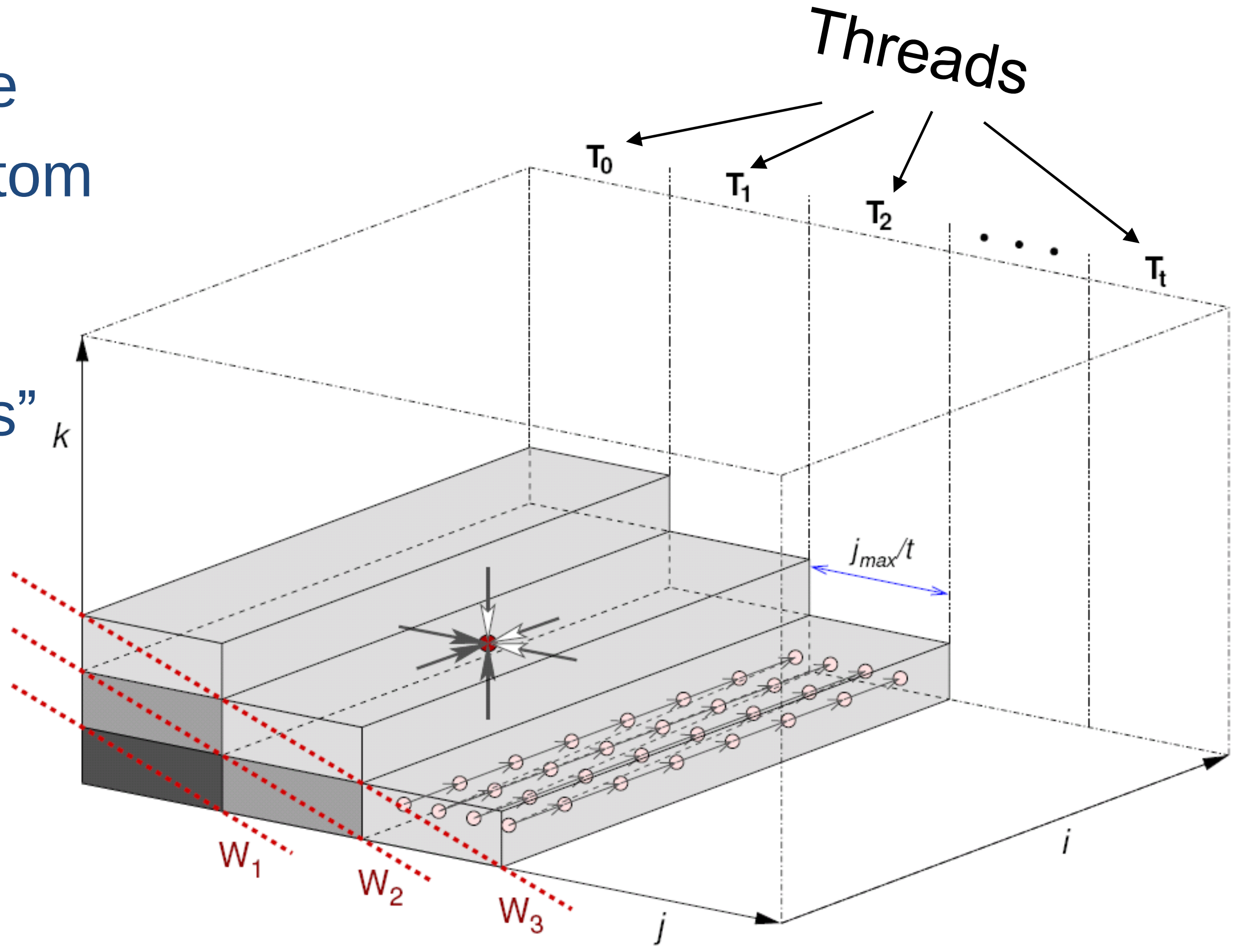



```
for(t=0; t<itmax; ++t) {
  for(k=1; k<kmax-1; ++k) {
    for(j=1; j<jmax-1; ++j) {
      for(i=1; i<imax-1; ++i) {
        x[k][j][i] = ( x[k][j][i-1] + x[k][j][i+1]
                     + x[k][j-1][i] + x[k][j+1][i]
                     + x[k-1][j][i] + x[k+1][j][i])/6.0;
} } } }
```

- Can we solve this in parallel but still keep the dependencies intact?

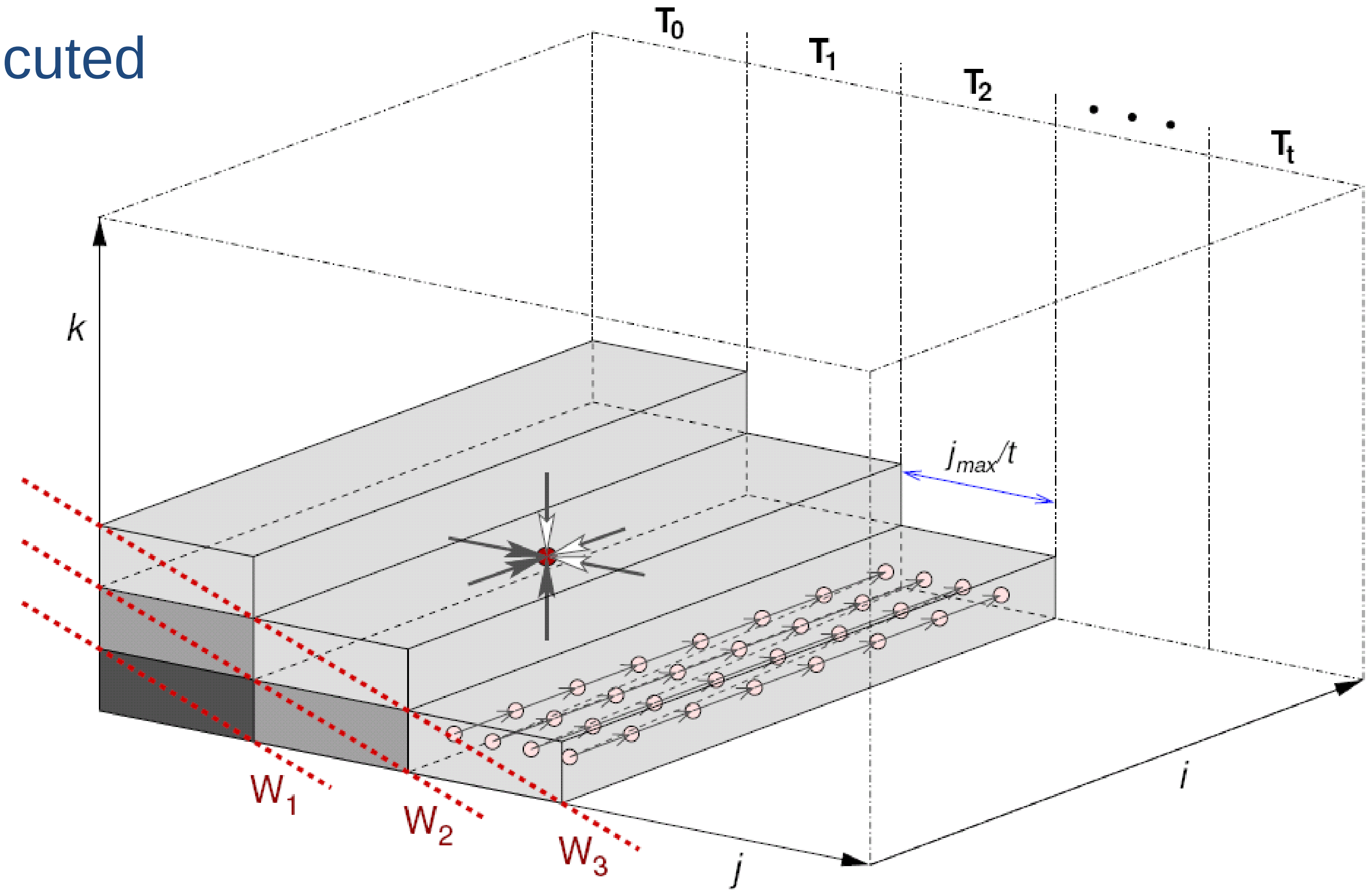
Idea: wavefront parallelization

- Parallelization approach
  - Middle (j) loop is parallel
  - Outer dimension: wavefront scheme
  - Each block can be updated iff if bottom neighbor (same threadID) and left neighbor (threadID-1) are done
  - W_i: independent blocks, “wavefronts”
  - After each wavefront: synchronization to maintain ordering



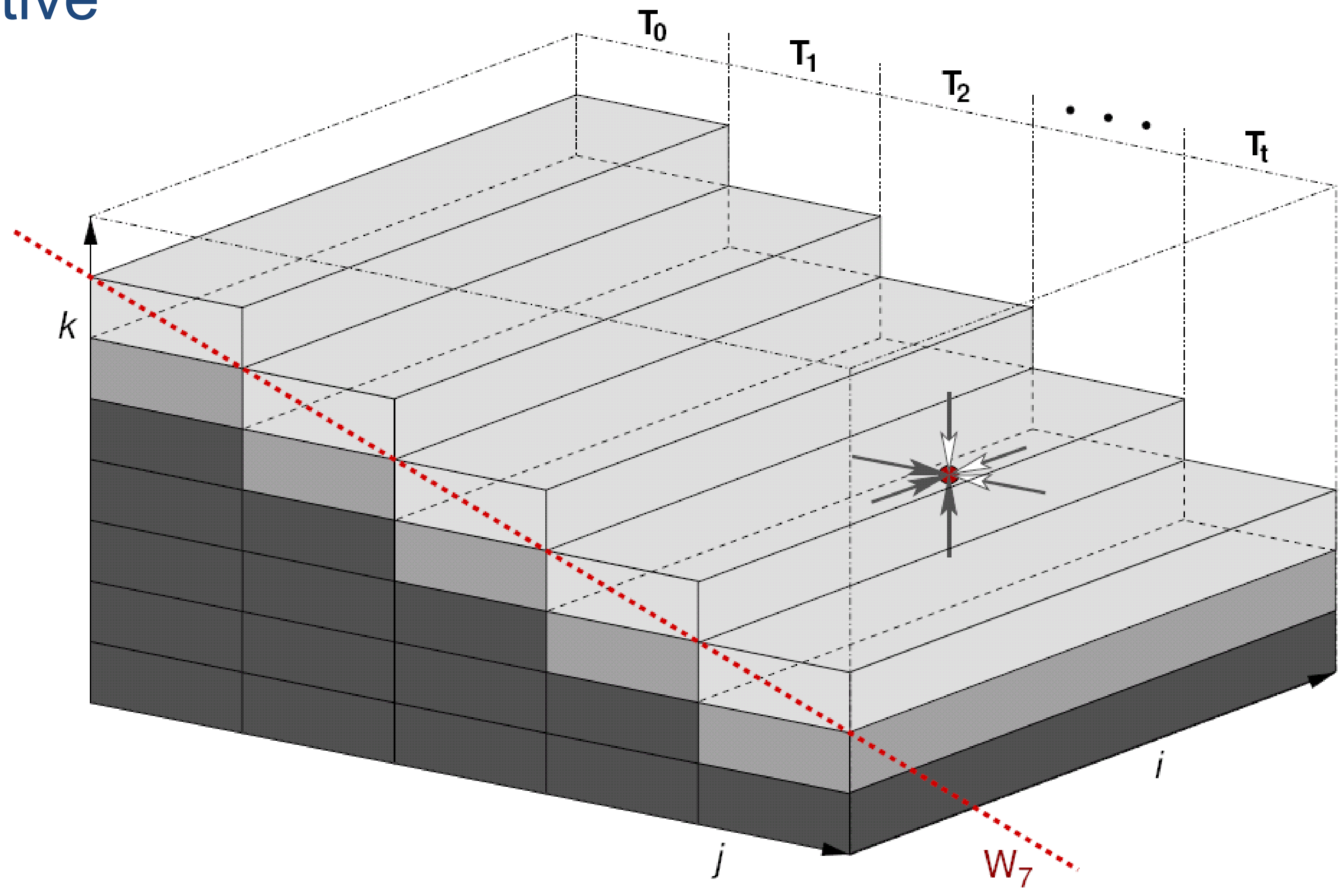
Wavefront parallelization

- Wind-up phase
  - Not all threads active
  - Each wavefront (W_i) is executed by i threads concurrently



Wavefront parallelization

- "Full pipeline": All threads active



- Wind-down phase starts after T_0 has completed its k loop (not shown)

Wavefront parallelization with OpenMP in 3D

```
#pragma omp parallel private(nth,istart,iend,tid,kk,it,k,i){
  nth = omp_get_num_threads();
  tid = omp_get_thread_num();
  jstart= (jmax-2)/nth * tid + 1;
  jend  = (tid==nth-1 ? jmax-2 : jstart+(jmax-2)/nth-1);
  for(t=0; t<itmax; ++t) {
    for(k=1; k<kmax-1+nth-1; ++k) {
      kk = k - tid;
      if(kk>=1 && kk<kmax-1) {
        for(j=jstart; j<=jend; ++j) {
          for(i=1; i<kmax-1; ++i) {
            x[kk][j][i] = ( x[kk][j][i-1] + x[kk][j][i+1]
                          + x[kk][j-1][i] + x[kk][j+1][i]
                          + x[kk-1][j][i] + x[kk+1][j][i])/6.0;
          }
        }
      }
      #pragma omp barrier
    }
  }
}
```

Chop **j** loop into **nthreads** chunks

Wind-up/-down

Wavefront sync

Wavefront parallelization – open questions

- Global barrier per middle loop sweep (i.e., **`kmax-2`** barriers overall)
  - Remedy?

Is there a global performance limit, i.e. minimum balance?

- Minimum data traffic:
  → Update whole array **`x(0:imax+1,0:jmax+1,0:kmax+1)`** once
  → 16 byte per element (read & write)
  → Minimum data traffic: **`16*imax*jmax*kmax`** byte
- Work:
  → **`imax*jmax*kmax`** Lattice Site Updates (LUPs)

→ B_C= 16 byte/LUP