Introduction to Parallel Programming for Multicore/Manycore Clusters

Part IV: Introduction to Tuning

Kengo Nakajima
Information Technology Center
The University of Tokyo

- What is “tuning/optimization” ?
- Vector/Scalar Processors
- Example: Scalar Processors

What is Tuning/Optimization

Optimization of Performance

- Purpose
  - Reduction of computation time
  - Optimization

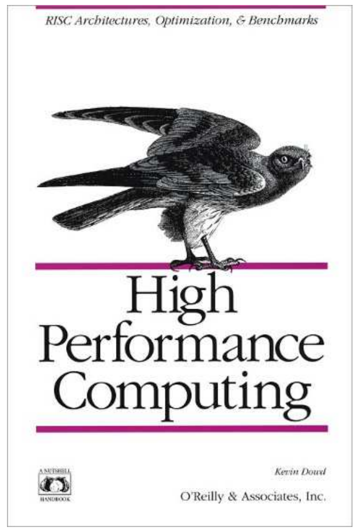
- How to tune/optimize codes
  - Applying new algorithms
  - Modification/optimization according to property/parameters of H/W
  - Tuning/optimization should not affect results themselves.
    - or you have to recognize that results may change due to tuning/optimization

How do we tune/optimize codes ?

- When do we apply tuning ?
  - It's difficult if you apply tuning to the codes with $O(10^4)$ lines ...
- You have to be careful to write "efficient" codes.
  - Several tips.
- Simple, Readable codes
  - codes with few bugs
- Using optimized libraries
- Good parallel program = good serial program
- Tuning/optimization – faster computation – efficient research ...

Some References (in Japanese)

- スカラープロセッサ
  - 寒川「RISC超高速化プログラミング技法」, 共立出版, 1995.
  - Dowd（久良知訳）「ハイ・パフォーマンス・コンピューティング-RISCワークステーションで最高のパフォーマンスを引き出すための方法」, トムソン, 1994. （Dowd, High Performance Computing, O'Reilly）
  - Goedecker, Hoisie "Performance Optimization for Numerically Intensive Codes", SIAM, 2001.
- 自動チューニング
  - 片桐「ソフトウェア自動チューニング」, 慧文社, 2004.
- ベクトルプロセッサ
  - 長島, 田中「スーパーコンピュータ」, オーム社, 1992.
  - 奥田, 中島「並列有限要素解析」, 培風館, 2004.



Tips for Tuning/Optimization

- Be careful about memory access patterns
- Avoid calling functions in innermost loops
  - Inline expansion of modern compilers might not work efficiently.
  - Avoid "if-clauses" in innermost loops.
- Avoid "too-multi-nested" loops
- Avoid many division operations, calling built-in-functions
- Avoid redundant operations
  - Storing in memory
  - Trade-off with memory capacity
- **Unfortunately, dependency on compilers and H/W is very significant !**
  - **Optimum options/directives through empirical studies**
  - **Today's content is very general remedy.**

Example: Multi-Nested Loops

- Overhead for initialization of loop-counter occurs at every do-loop.
  - In the lower-left example (blue), innermost loop is reached 10^6 times. Therefore, 10^6 times initialization of loop-counter occurs.
  - In the lower-right example (yellow) with loop expansion, only one initialization of loop-counter occurs.

```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  do i= 1, 3
    A(i,j)= A(i,j) + 1.0
  enddo
enddo
. . .
```

→

```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  A(1,j)= A(1,j) + 1.0
  A(2,j)= A(2,j) + 1.0
  A(3,j)= A(3,j) + 1.0
enddo
. . .
```

Simple ways for measuring performance

- “time” command
- “timer” subroutines/functions
- Tools for “profiling”
  - Detection of “hot spots”
  - gprof (UNIX)
  - Tools for compilers/systems
    - pgprof: PGI
    - Vtune: Intel
    - Special Profiler: e.g. Fujitsu PRIMEHPC FX10

Files on OBCX

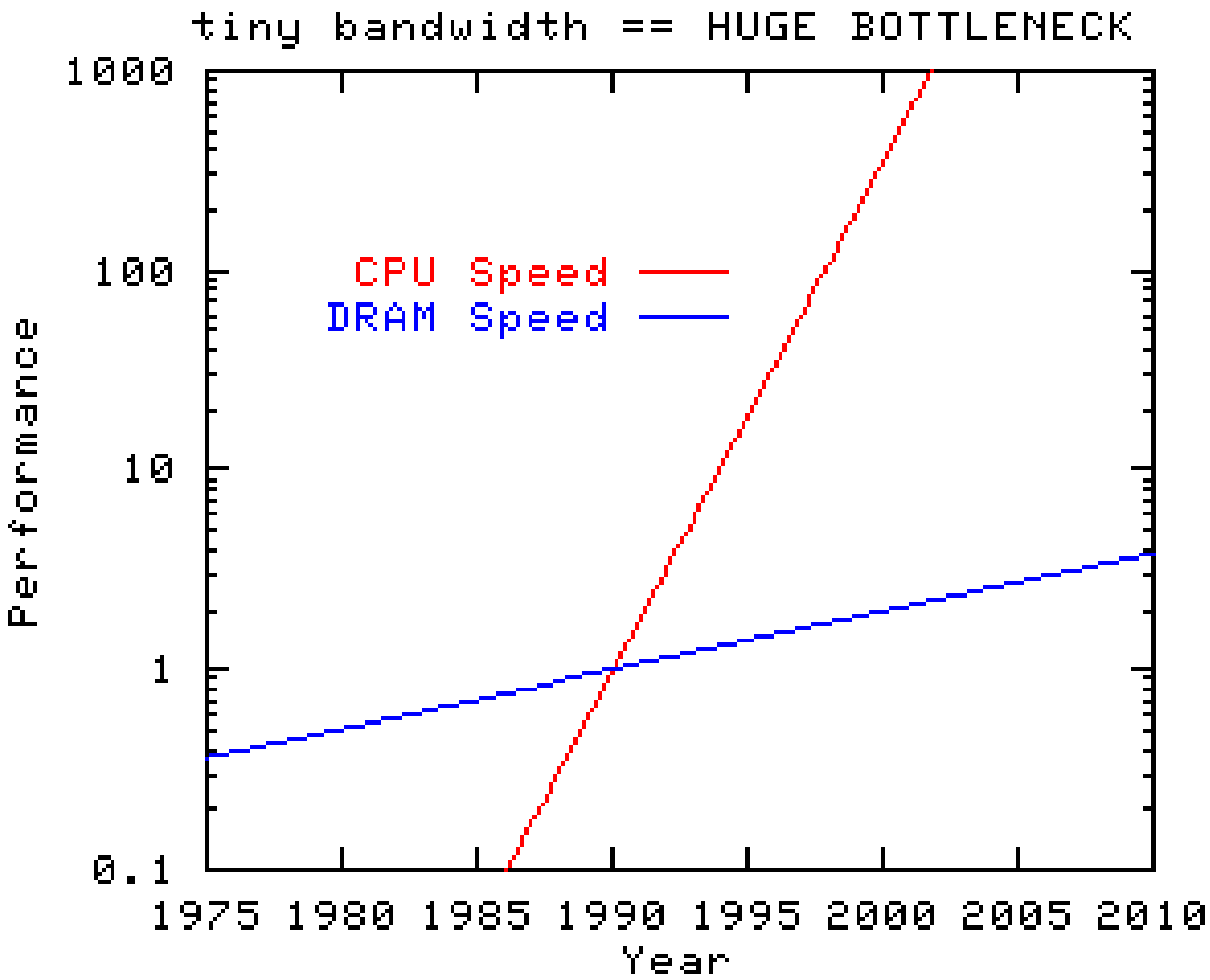
```
>$ cd /work/gt69/t69XXX

>$ cp /work/gt69/z30088/omp/omp-f4.tar .

>$ tar xvf omp-f4.tar

>$ cd multicore/omp   <$O-omp>
>$ ls

dense run src20
```

- What is “tuning/optimization” ?
- **Vector/Scalar Processors**
- Example: Scalar Processors

Scalar/Vector Processors

- Scalar Processors
  - Gap between clock rate and memory bandwidth.
    - getting better, but multi/many-core architectures appear
  - Low Peak-Performance Ratio
    - Ex.: IBM Power3/Power4, 5-8% in FEM applications
- "Traditional" Vector Processors
  - High Peak-Performance Ratio
    - Ex.: Earth Simulator, 35% in FEM applications
  - requires ...
    - very special tuning for vector processors
    - sufficiently long loop (problem size)
  - Appropriate for rather simpler problems
  - GPU, Intel Xeon Phi, Intel Xeon, A64FX on Fugaku

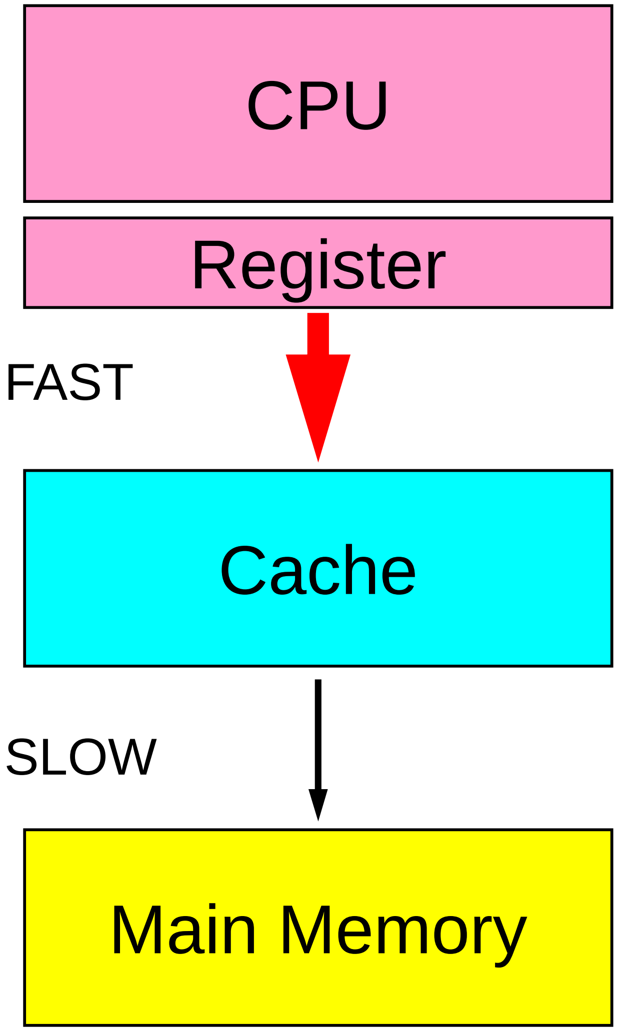
Gap between performance of CPU and Memory



http://www.streambench.org/

Scalar Processors

CPU-Cache: Hierarchical Structure



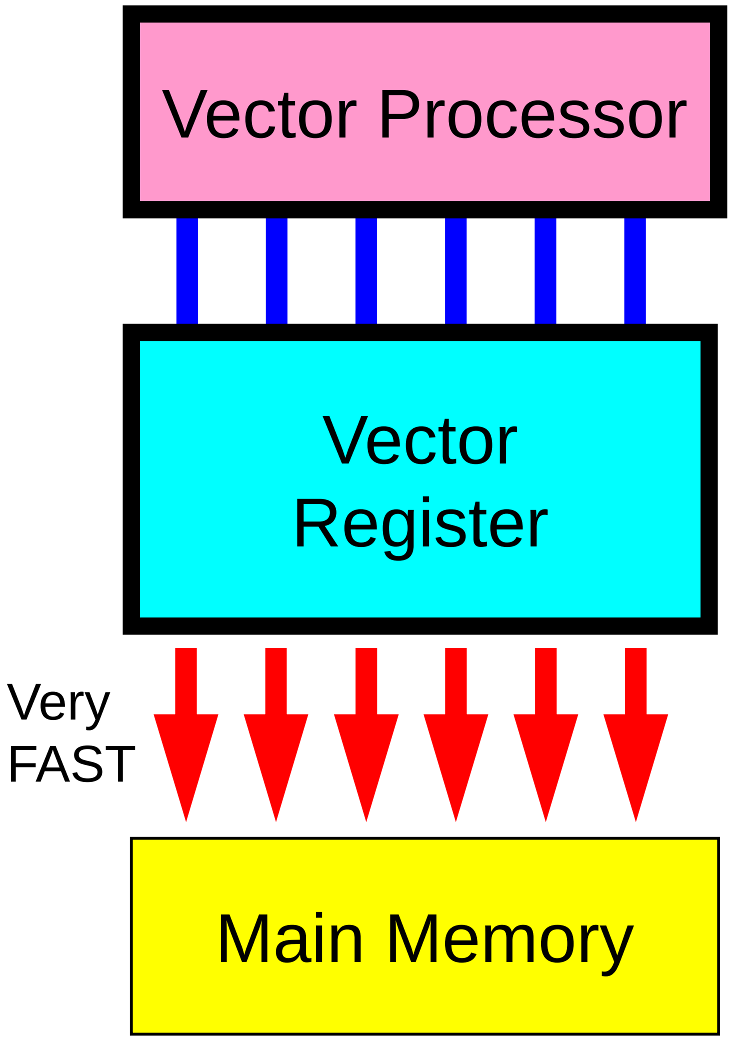
Small (MB)
Expensive
Large (with $O(10^8)$-$O(10^9)$ transistors)
Instruction/Data Cache
Hierarchy: L1, L2, L3 (Last-Level)

Large (GB)
Cheap

"Traditional" Vector Processors

Vector Registers/Fast Memory

Size of Vector Register: 128, 256 …



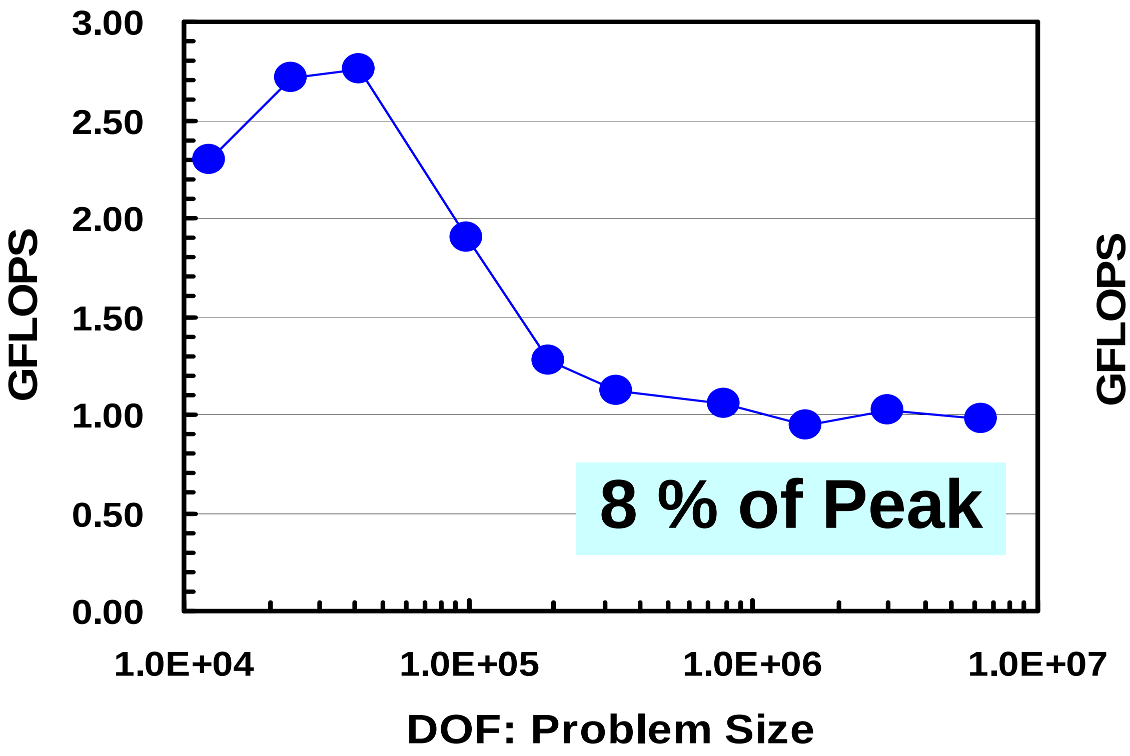
- Parallel operations for simple do-loops
- Good for large-scale simple problems

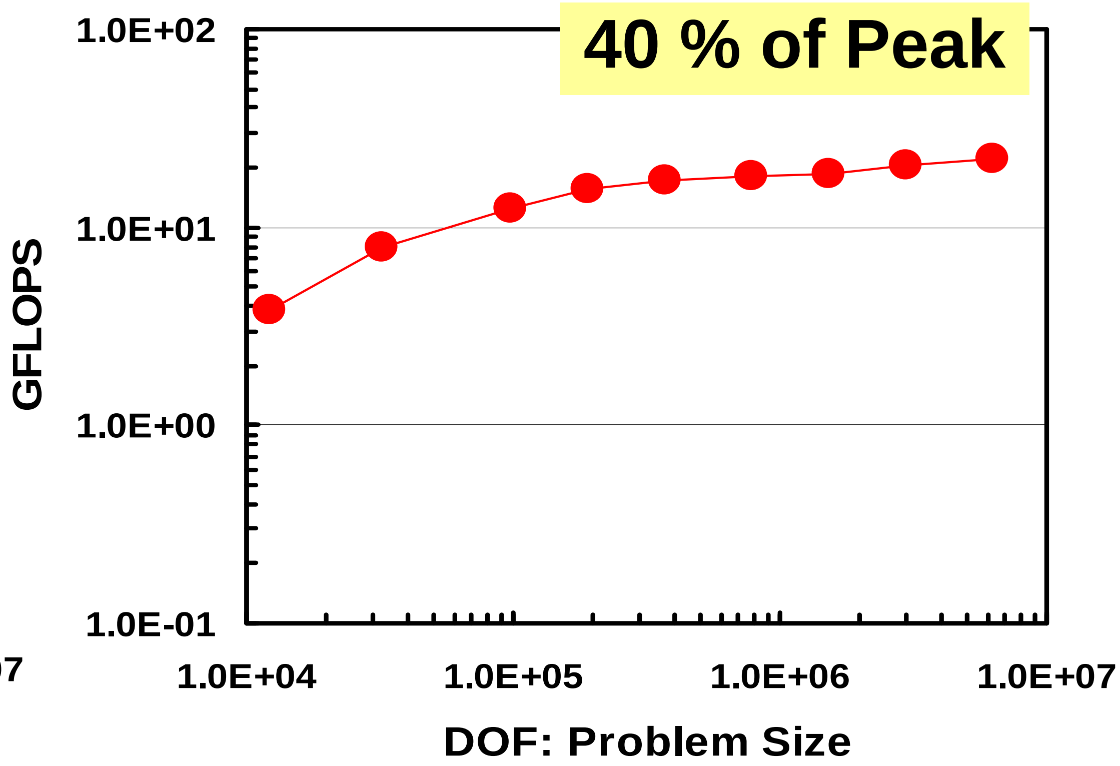
```
do i= 1, N
  A(i)= B(i) + C(i)
enddo
```

NO cache

Modern vector processors have cache
GPU, Xeon Phi: short vectors (AVX512: 8 for Double Precision (512-bit))

Typical Behaviors (2002) Sparse Linear Solvers for FEM





IBM-SP3

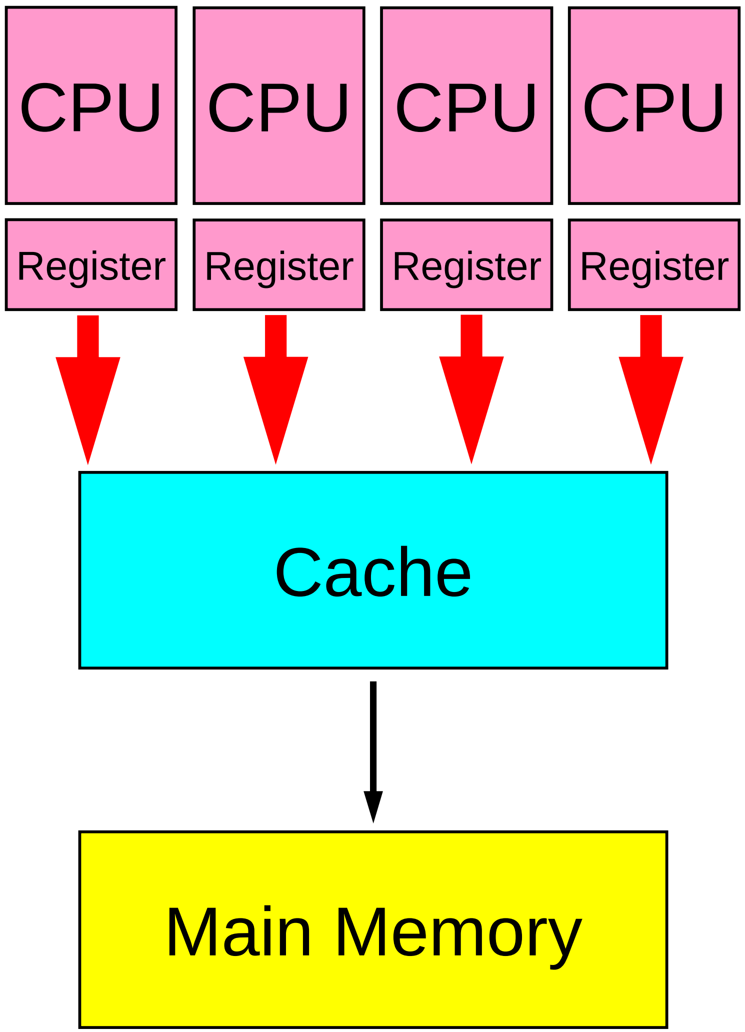
Higher performance for small problems, effect of cache

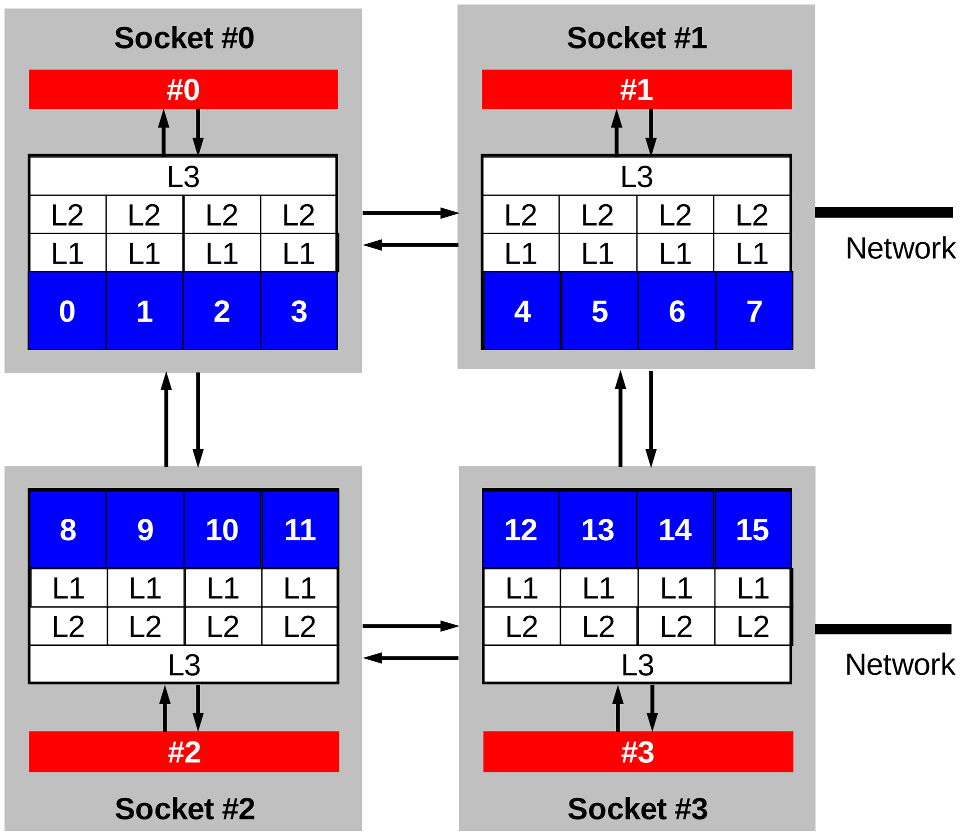
Earth Simulator

Higher performance for large-scale problems with longer loops

Multicores/Manycores

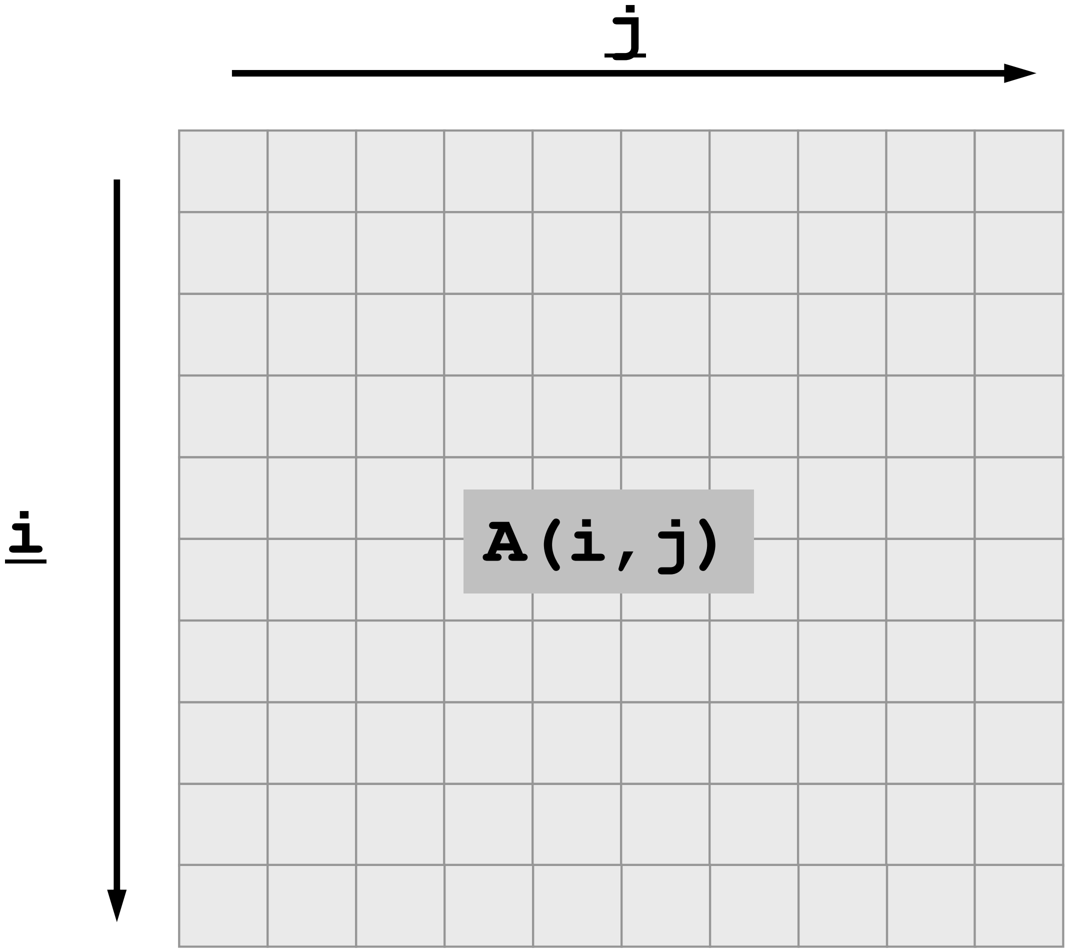
Multiple cores share memory/cache
-> Saturation of Memory





How to get optimum performance by Tuning ?

- = Optimization of memory access



How to get optimum performance by Tuning ? (cont.)

- “Traditional” Vector Processors
  – Long loops (128-256)
- Scalar Processors
  – Utilization of cache, small chunks of data
- Common Issues
  – Continuous memory access
  – Localization
  – Changing sequence of computations might provide change of results.

- What is “tuning/optimization” ?
- Vector/Scalar Processors
- **Example: Scalar Processors**

Typical Methods of Tuning for Scalar Processors

- Loop Unrolling
  - loop overhead
  - loading/storing

- Loop Exchange
- Blocking/Cache Blocking

- All evaluations are done by using a Single Core

Scalar

BLAS: Basic Linear Algebra Subprograms

- Library API for fundamental operations of vectors and (dense) matrices)

- Level 1: Vectors: dot products, DAXPY
- Level 2: Matrix x Vector
- Level 3: Matrix x Matrix

- LINPACK/HPL
  - DGEMM: Level 3 BLAS

Scalar

Loop Unrolling

reduction of loading/storing

- Ratio of computation increases

```
N= 10000

do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
  enddo
enddo

do j= 1, N-1, 2
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    A(i)= A(i) + B(i)*C(i,j+1)
  enddo
enddo

do j= 1, N-3, 4
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    A(i)= A(i) + B(i)*C(i,j+1)
    A(i)= A(i) + B(i)*C(i,j+2)
    A(i)= A(i) + B(i)*C(i,j+3)
  enddo
enddo
```

Original

Interval=2

Interval=3

Scalar

Loop Unrolling

reduction of loading/storing

```
$> cd /wotk/gt69/t69XXX
$> cd multicore/omp/dense
$> mpiifort -align array64byte -O3 -axCORE-AVX512 t2.f -o t2
```

Please add the option "-no-multibyte-chars", if you have any problems in compiling

```
<modify "t2.sh">
$> pjsub t2.sh (1 process)

$> cat t2.lst
    6.376914E-02        Original
    5.503920E-02        Interval=2
    5.233580E-02        Interval=4
```

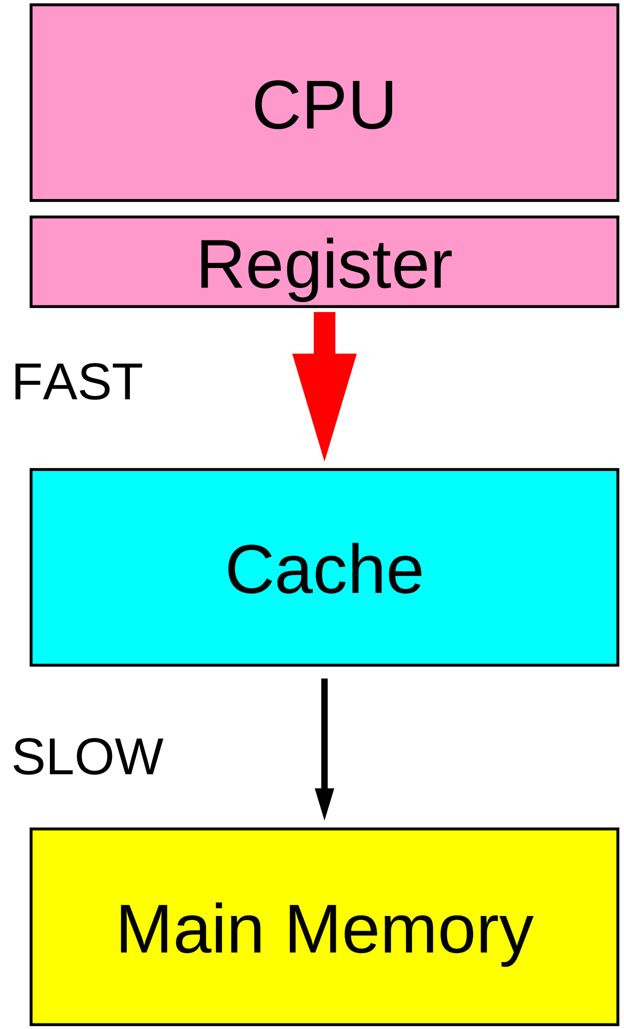
t2.sh for a single core

```
#!/bin/sh
#PJM -N   "t2"
#PJM -L   rscgrp=lecture9
#PJM -L   node=1
#PJM --mpiproc=1
#PJM -L   elapse=00:15:00
#PJM -g   gt69
#PJM -j
#PJM -e   err
#PJM -o   t2.lst

mpiexec.hydra  -n   ${PJM_MPI_PROC}./t2
```

Loop Unrolling

reduction of loading/storing (2/4)

- **Load** : Memory-Cache-Register
- **Store**: Register-Cache-Memory
- Fewer loading/storing ⇒ better performance



Scalar

Loop Unrolling

reduction of loading/storing (3/4)

```
do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
   Store  Load   Load Load
  enddo
enddo
```

- Loading/Storing for A(i), B(i), C(i,j) occurs in each loop.
- 1*S, 3*L: 2*C (4:2)

Scalar

Loop Unrolling

reduction of loading/storing (4/4)

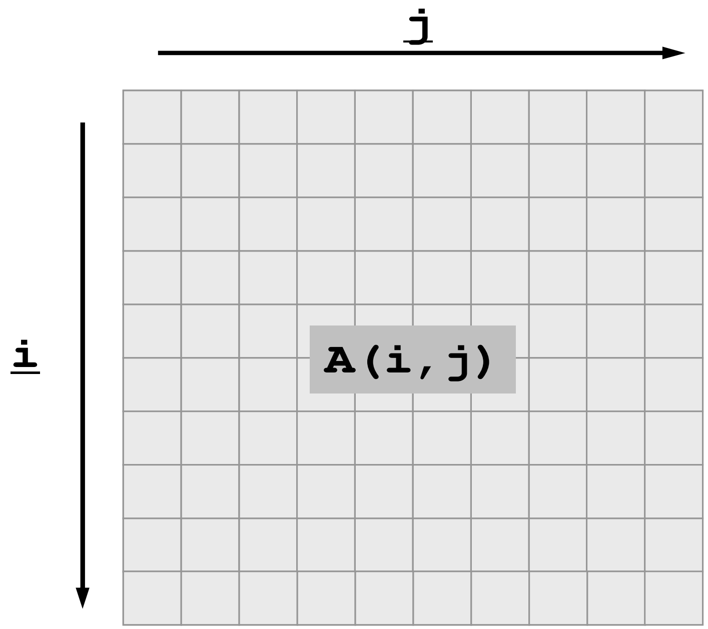
```
do j= 1, N-3, 4
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
          Load   Load Load
    A(i)= A(i) + B(i)*C(i,j+1) Load
    A(i)= A(i) + B(i)*C(i,j+2) Load
    A(i)= A(i) + B(i)*C(i,j+3) Load
    Store
  enddo
enddo
```

- Values of arrays are kept on register during each loop. Storing occurs only at the end of the loop.
- Ratio of memory access (loading/storing) to computation can be reduced (1*S, 6*L: 8*C (7:8))
- Be careful about sequence of computations.

Scalar

Loop Exchanging (1/3)

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo

TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```



- In Fortran, component of A(i,j) is aligned in the following way: A(1,1), A(2,1), A(3,1),…, A(N,1), A(1,2), A(2,2),…, A(1,N), A(2,N),…, A(N,N)
  - In C: A[0][0], A[0][1], A[0][2], …, A[N-1][0], A[N-1][1],…,A[N-1][N-1]
- Access must be according to this alignment for higher performance.

Scalar

Loop Exchanging (2/3)

- TYPE-B provides continuous memory access
- Generally, TYPE-B is faster than TYPE-A

TYPE-A
```
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

TYPE-B
```
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

TYPE-A
```
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

TYPE-B
```
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

Loop Exchanging (3/3): Comp. Time

mpiifort -align array64byte -O3 -axCORE-AVX512 2d-1.f -o 2d-1

```
$> cd /work/gt69/t69XXX
$> cd multicore/omp/dense
$> mpiifort …
$> pjsub 2d-1.sh

$> cat 2d-1.lst
### N ###      500
WORSE       1.521902E-04 TYPE-A
BETTER      1.496547E-04 TYPE-B
### N ###     1000
WORSE       6.083571E-04
BETTER      6.320321E-04
### N ###     1500
WORSE       1.633597E-03
BETTER      1.578360E-03
### N ###     2000
WORSE       3.586689E-03
BETTER      3.616580E-03
### N ###     2500
WORSE       6.087084E-03
BETTER      6.115762E-03
### N ###     3000
WORSE       9.118990E-03
BETTER      9.118399E-03
### N ###     3500
WORSE       1.249523E-02
BETTER      1.251535E-02
### N ###     4000
WORSE       1.669729E-02
BETTER      1.657084E-02
```

2d-1.sh

```
#!/bin/sh
#PJM -L rscgrp=lecture9
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt69
#PJM -j
#PJM -e err
#PJM -o 2d-1.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./2d-1
```

No difference between TYPE-A and TYPE-B on OBCX

Scalar

Scalar

Loop Exchanging

No Difference between TYPE-A & TYPE-B

2d-1.f, 2d-1.sh, Small

```
### N ###      500
WORSE        1.521902E-04  TYPE-A
BETTER       1.496547E-04  TYPE-B
### N ###     1000
WORSE        6.083571E-04
BETTER       6.320321E-04
### N ###     1500
WORSE        1.633597E-03
BETTER       1.578360E-03
### N ###     2000
WORSE        3.586689E-03
BETTER       3.616580E-03
### N ###     2500
WORSE        6.087084E-03
BETTER       6.115762E-03
### N ###     3000
WORSE        9.118990E-03
BETTER       9.118399E-03
### N ###     3500
WORSE        1.249523E-02
BETTER       1.251535E-02
### N ###     4000
WORSE        1.669729E-02
BETTER       1.657084E-02
```

2d-1x.f, 2d-1x.sh, Large

```
### N ###     2000
WORSE        3.734894E-03  TYPE-A
BETTER       3.897774E-03  TYPE-B
### N ###     4000
WORSE        1.701343E-02
BETTER       1.697535E-02
### N ###     6000
WORSE        3.863532E-02
BETTER       3.869803E-02
### N ###     8000
WORSE        6.898103E-02
BETTER       6.909023E-02
### N ###    10000
WORSE        1.075633E-01
BETTER       1.078388E-01
### N ###    12000
WORSE        1.552545E-01
BETTER       1.553766E-01
### N ###    14000
WORSE        2.113688E-01
BETTER       2.112531E-01
### N ###    16000
WORSE        2.754752E-01
BETTER       2.754681E-01
```

Intel 2018 or 2019 or 2020

```
$> cd /work/gt69/t69XXX
$> cd multicore/omp/dense

$> module avail intel

-- /home/opt/local/modulefiles/L/compiler/intel/2019.5.281 -----------
intelpython/2.7          intelpython/3.6(default)

-- /home/opt/local/modulefiles/L/core ----------------------
intel/2017.4.196   intel/2019.4.243    intel/2020.2.254
intel/2018.3.222   intel/2019.5.281    intel/2020.4.304(default)
intel/2019.3.199   intel/2020.1.217


$> module unload intel/2020.4.304
$> module load intel/2018.3.222


$> mpiifort -align array64byte -O3 -axCORE-AVX512 2d-1.f -o 2d-1b
$> pjsub 2d-1b.sh

$> mpiifort -align array64byte -O3 -axCORE-AVX512 2d-1x.f -o 2d-1xb
$> pjsub 2d-1xb.sh
```

2d-1b.sh: 2020⇒2018

```
#!/bin/sh
#PJM -L rscgrp=lecture9
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt69
#PJM -j
#PJM -e err
#PJM -o 2d-1b.lst

module unload intel/2020.4.304
module load intel/2018.3.222
mpiexec.hydra -n ${PJM_MPI_PROC} ./2d-1b
```

Scalar

Loop Exchanging (Small)

2019: 2d-1.lst

```
### N ###       500
WORSE       1.521902E-04 TYPE-A
BETTER      1.496547E-04 TYPE-B
### N ###      1000
WORSE       6.083571E-04
BETTER      6.320321E-04
##
WO
BE
##
WO
BE
##
WO
BE
### N ###      3000
WORSE       9.118990E-03
BETTER      9.118399E-03
### N ###      3500
WORSE       1.249523E-02
BETTER      1.251535E-02
### N ###      4000
WORSE       1.669729E-02
BETTER      1.657084E-02
```

2018: 2d-1b.lst

```
### N ###       500
WORSE       1.609325E-04 TYPE-A
BETTER      1.499653E-04 TYPE-B
### N ###      1000
WORSE       6.151199E-04
BETTER      6.170273E-04
### N ###      3000
WORSE       1.769781E-02
BETTER      9.181976E-03
### N ###      3500
WORSE       2.958822E-02
BETTER      1.265812E-02
### N ###      4000
WORSE       3.591013E-02
BETTER      1.658821E-02
```

- **Significant difference between TYPE-A & TYPE-B for Intel 2018**
- **No difference for Intel 2019**
- **2020~2019: 2020 is slightly better for larger problems**

Scalar

Loop Exchanging (Large)

2019: 2d-1x.lst, 2020~2019

```
### N ###      2000
WORSE        3.734894E-03 TYPE-A
BETTER       3.897774E-03 TYPE-B
### N ###      4000
WORSE        1.701343E-02
BETTER       1.697535E-02
### N ###      6000
WORSE        3.863532E-02
BETTER       3.869803E-02
### N ###      8000
WORSE        6.898103E-02
BETTER       6.909023E-02
### N ###     10000
WORSE        1.075633E-01
BETTER       1.078388E-01
### N ###     12000
WORSE        1.552545E-01
BETTER       1.553766E-01
### N ###     14000
WORSE        2.113688E-01
BETTER       2.112531E-01
### N ###     16000
WORSE        2.754752E-01
BETTER       2.754681E-01
```

2018: 2d-1xb.lst

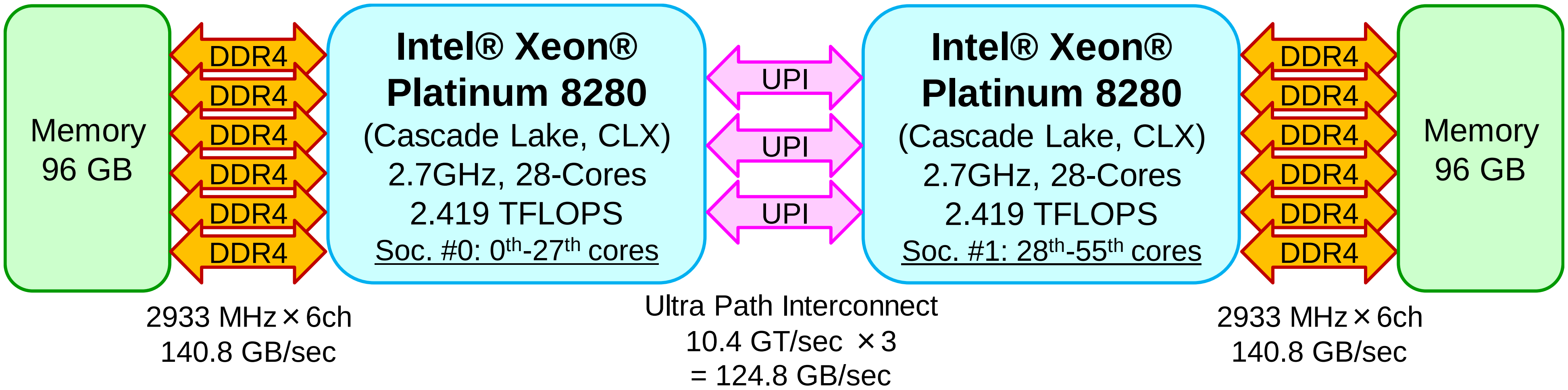
```
### N ###      2000
WORSE        6.095886E-03
BETTER       3.753901E-03
### N ###      4000
WORSE        3.577900E-02
BETTER       1.659989E-02
### N ###      6000
WORSE        8.446097E-02
BETTER       3.776813E-02
### N ###      8000
WORSE        1.626399E-01
BETTER       6.745481E-02
### N ###     10000
WORSE        2.308340E-01
BETTER       1.053209E-01
### N ###     12000
WORSE        3.729970E-01
BETTER       1.517632E-01
### N ###     14000
WORSE        4.552810E-01
BETTER       2.069180E-01
### N ###     16000
WORSE        6.461918E-01
BETTER       2.696209E-01
```

Cache Blocking (1/8)

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo
```

- Consider this situation.

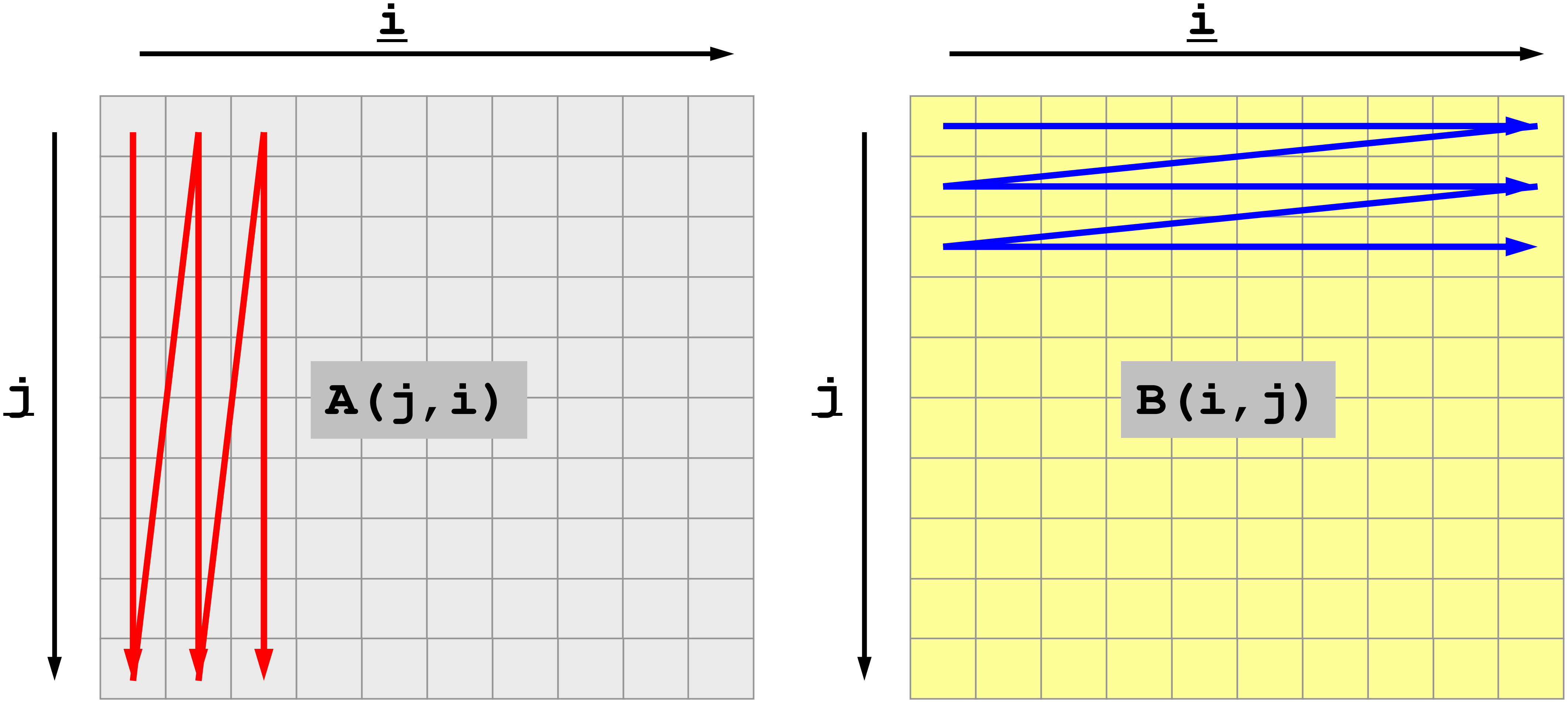
Category	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B



Scalar

Cache Blocking (2/8)

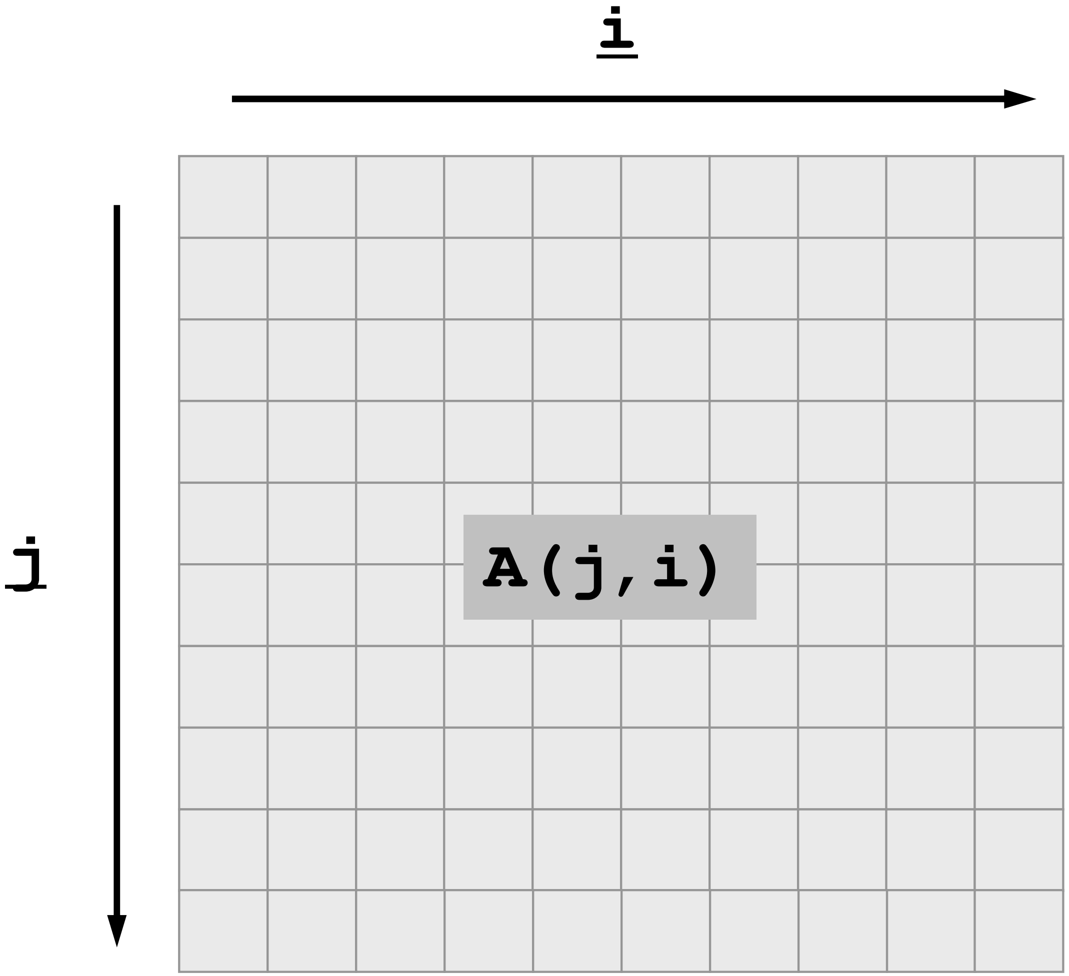
- Direction of optimum memory access for "A" is different from that of "B". Especially, not good for "B".



Scalar

Cache Blocking (3/8)

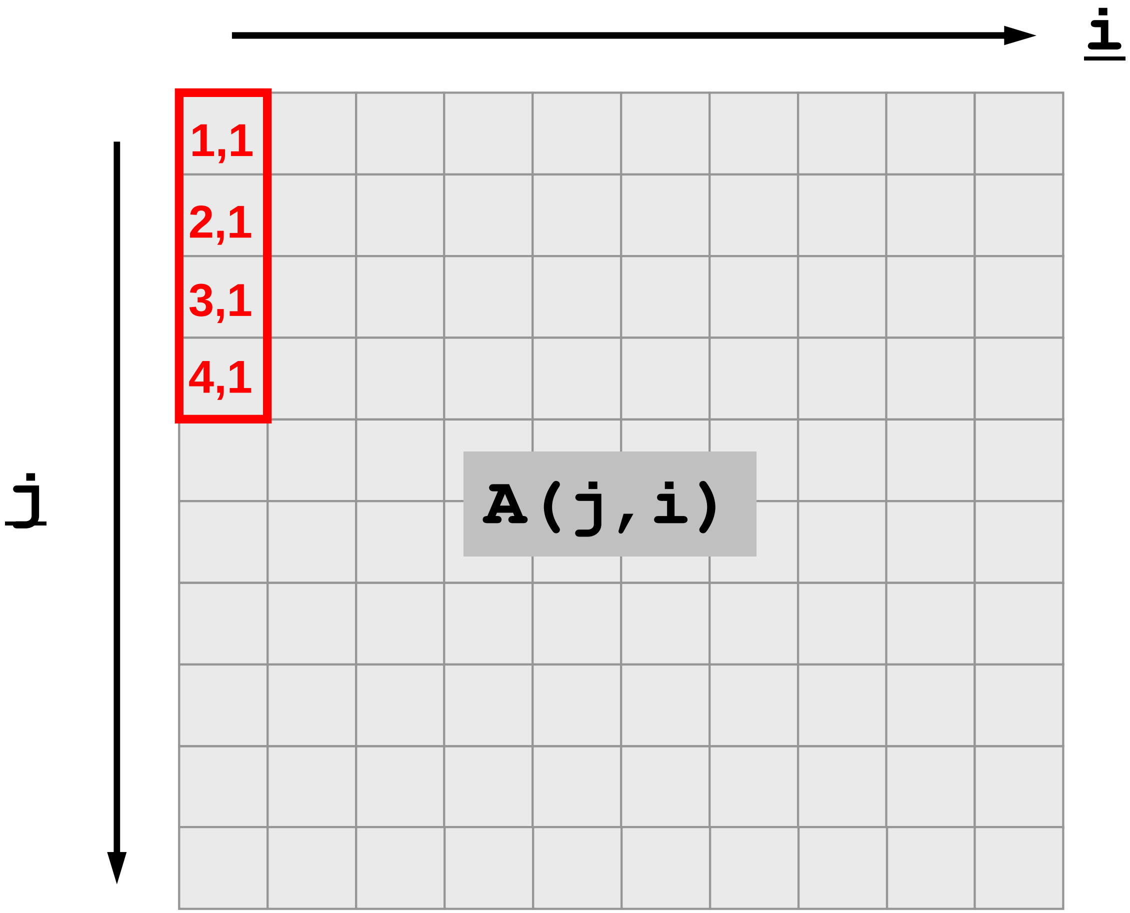
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:



Scalar

Cache Blocking (3/8)

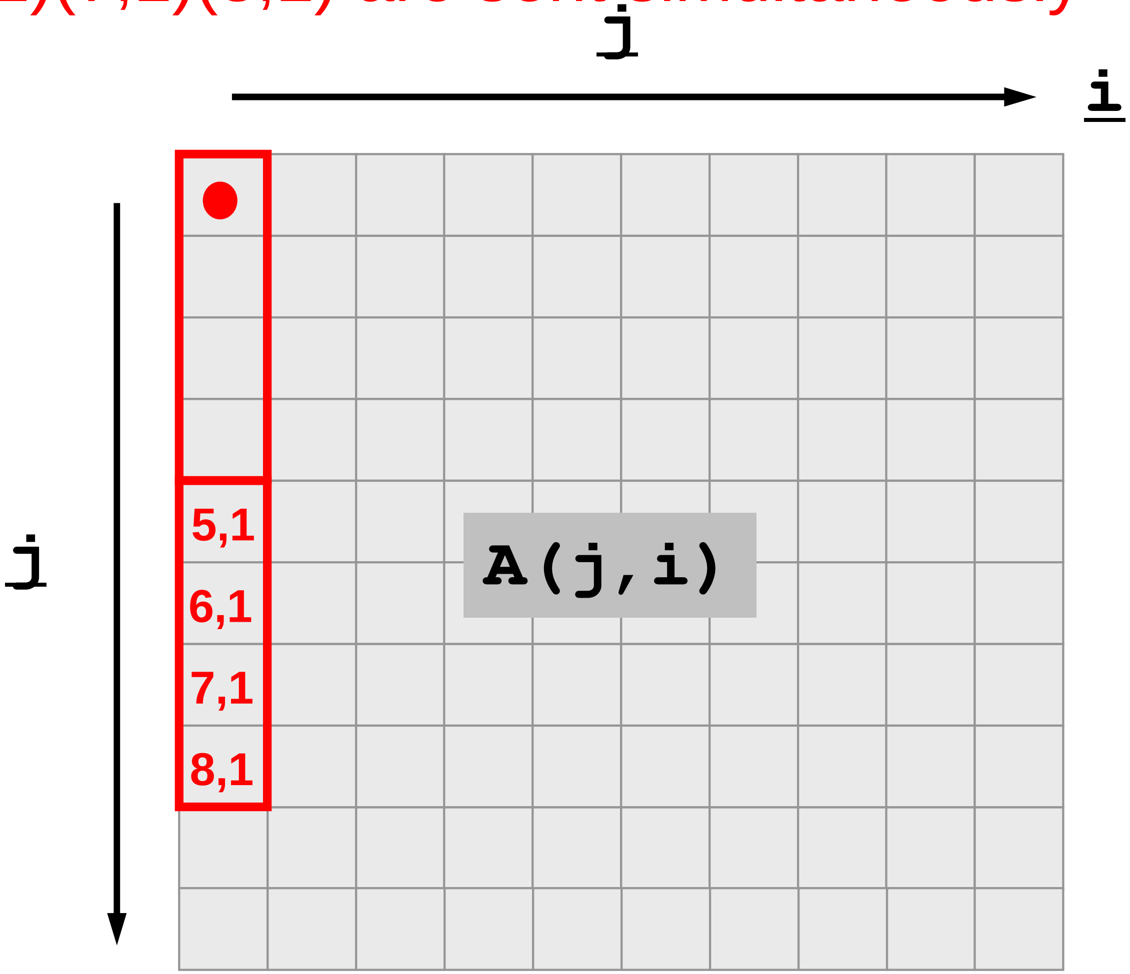
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:
  - A(1,1)(2,1)(3,1)(4,1) are sent simultaneously



Scalar

Cache Blocking (3/8)

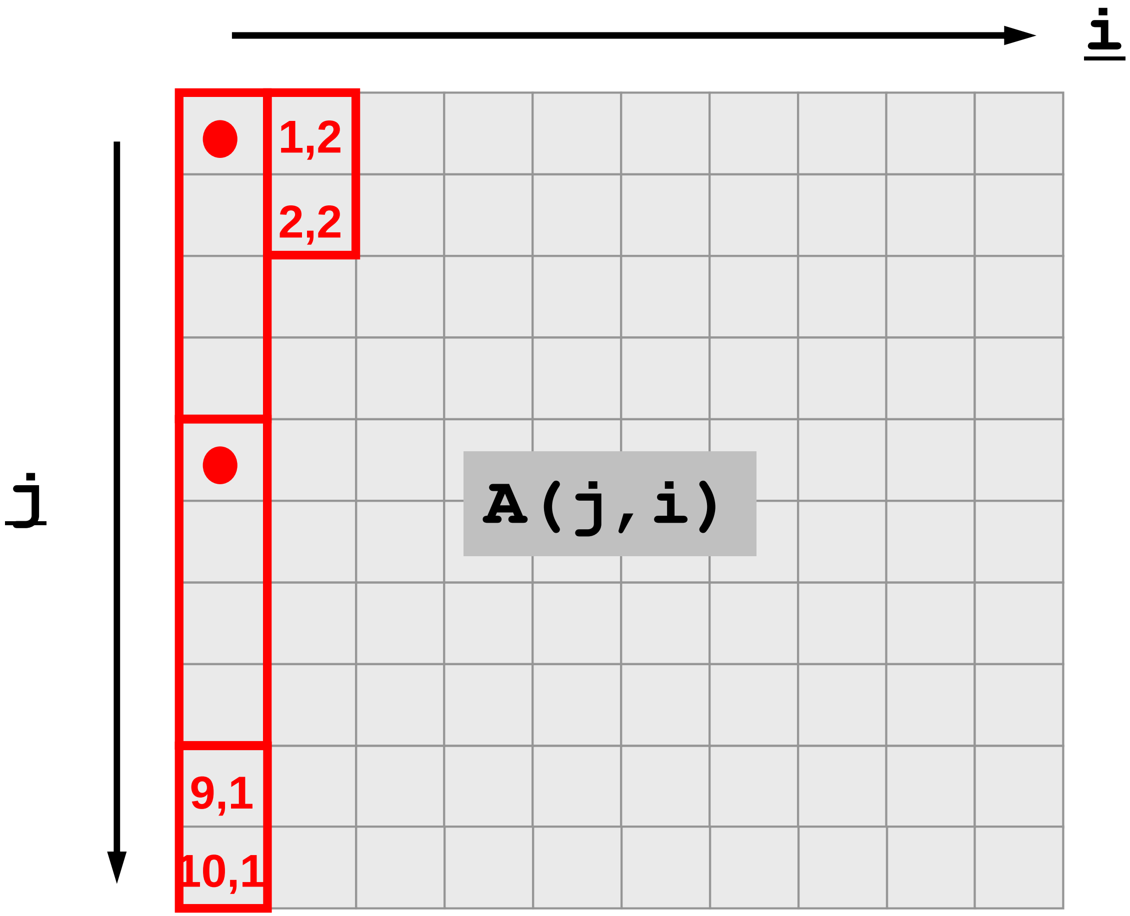
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:
  - A(5,1)(6,1)(7,1)(8,1) are sent simultaneously



Scalar

Cache Blocking (3/8)

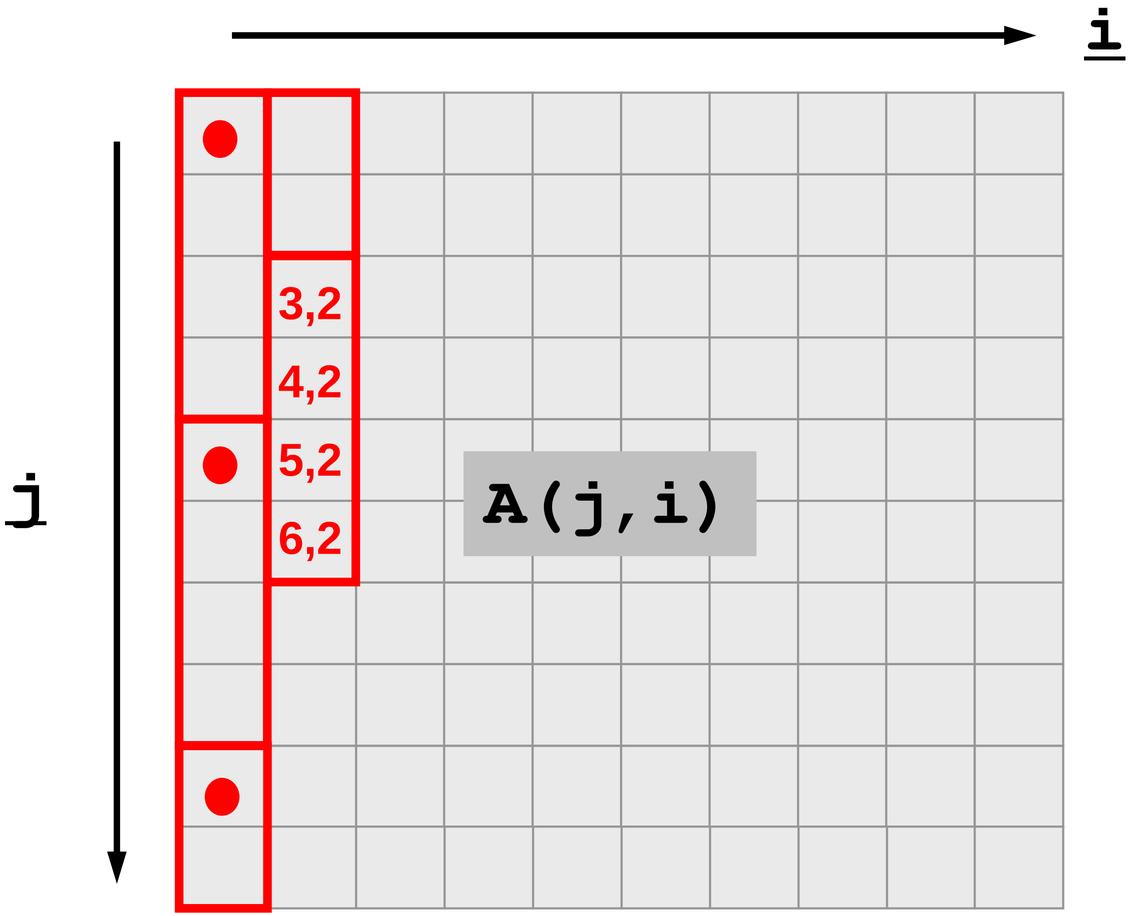
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:
  - A(9,1)(10,1)(1,2)(2,2) are sent simultaneously



Scalar

Cache Blocking (3/8)

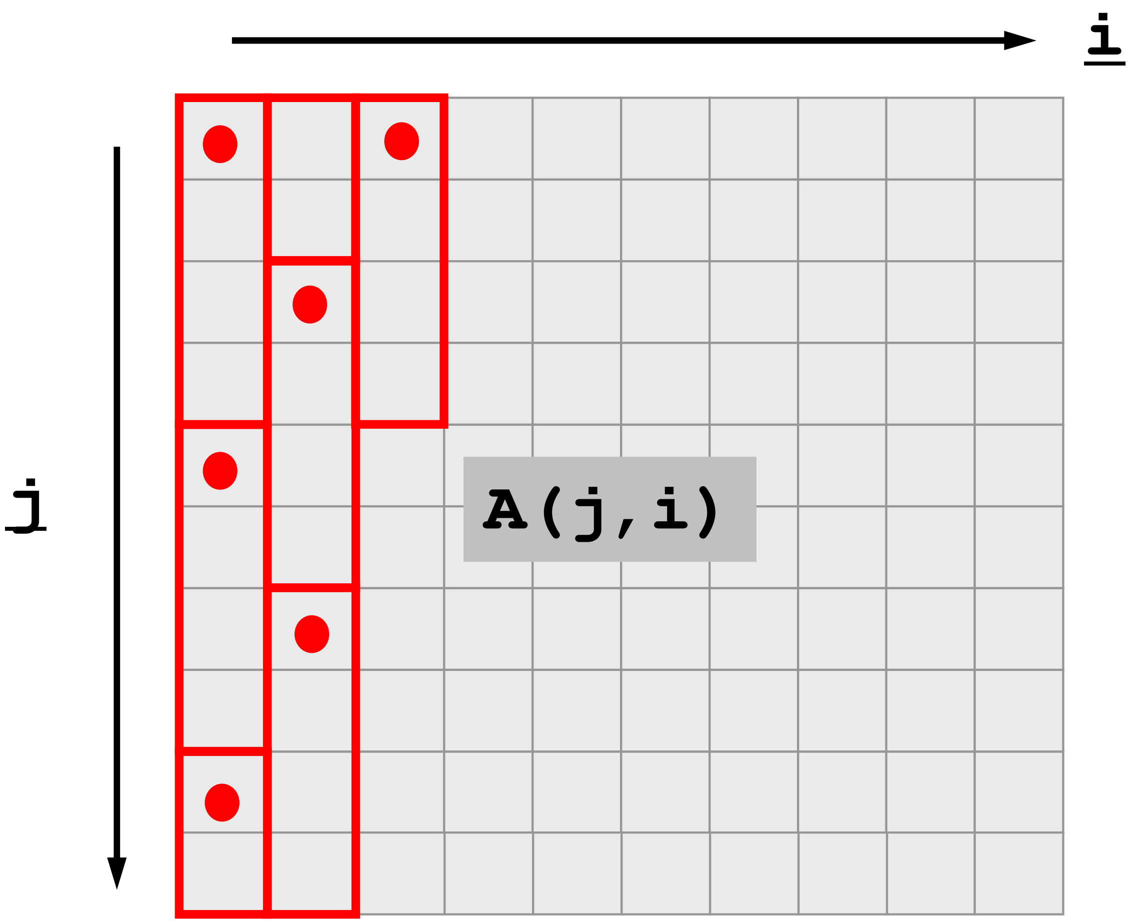
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:
  - A(3,2)(4,2)(5,2)(6,2) are sent simultaneously



Scalar

Cache Blocking (3/8)

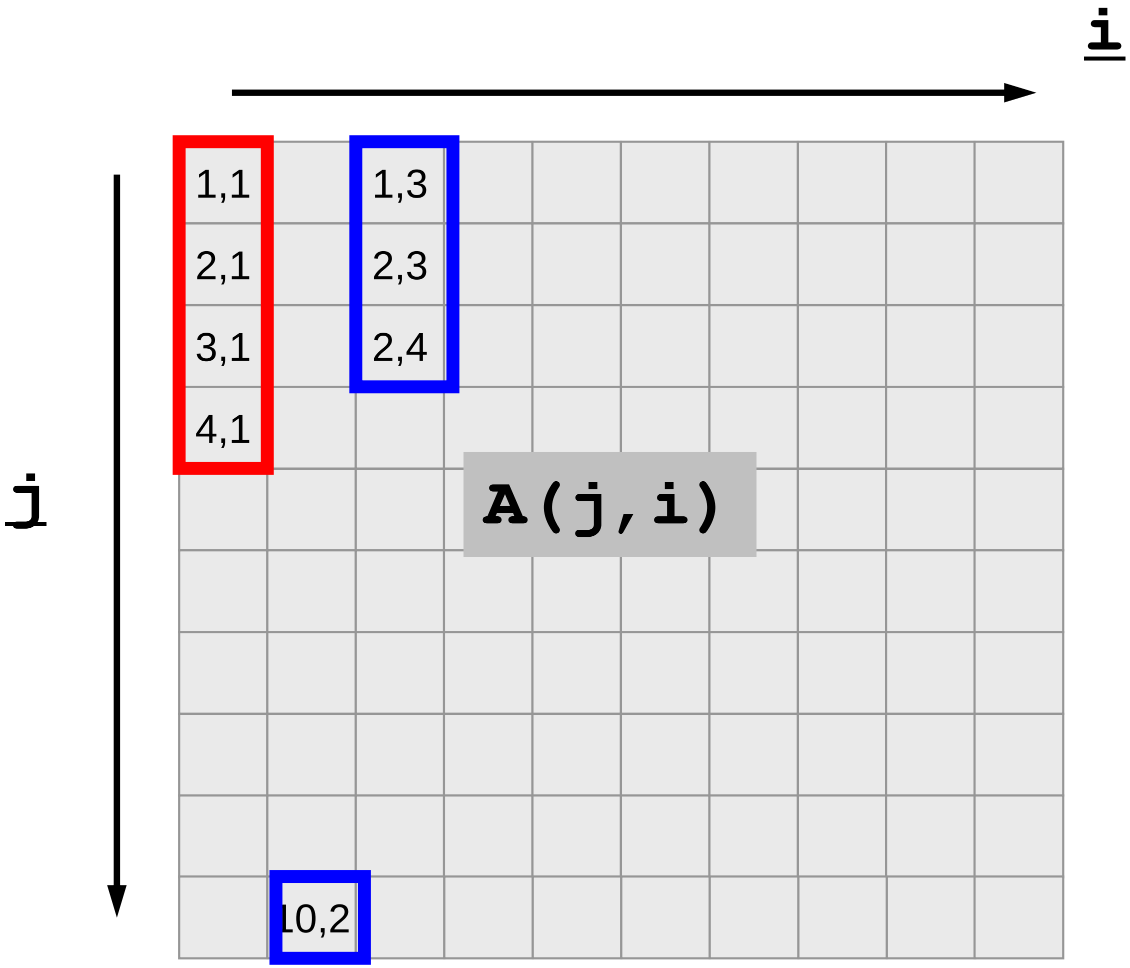
- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:



Scalar

Cache Blocking (4/8)

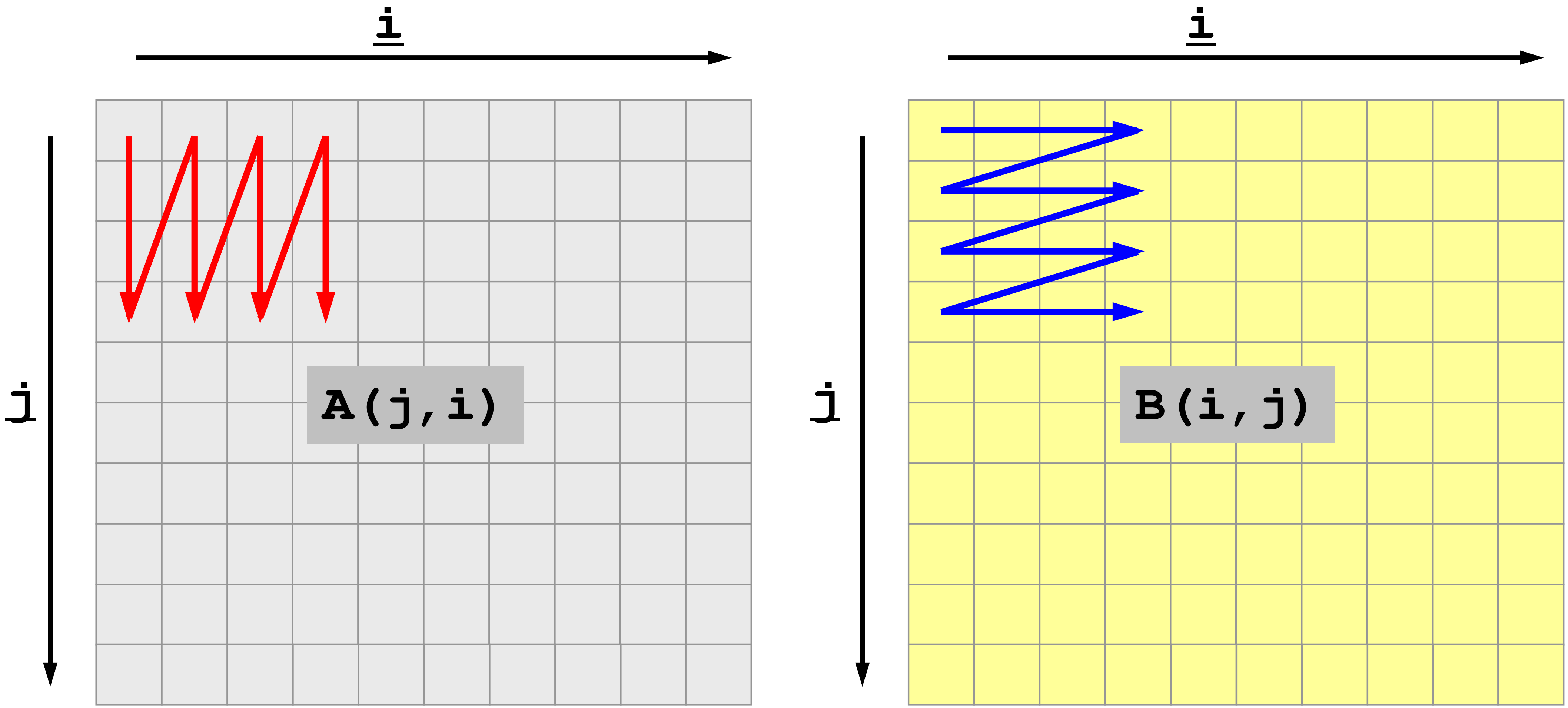
- If A(1,1) is touched, A(1,1), A(2,1), A(3,1), A(4,1) are on cache simultaneously. If A(10,2) is touched A(10,2), A(1,3), A(2,3), A(3,3) are on cache.



Scalar

Cache Blocking (5/8)

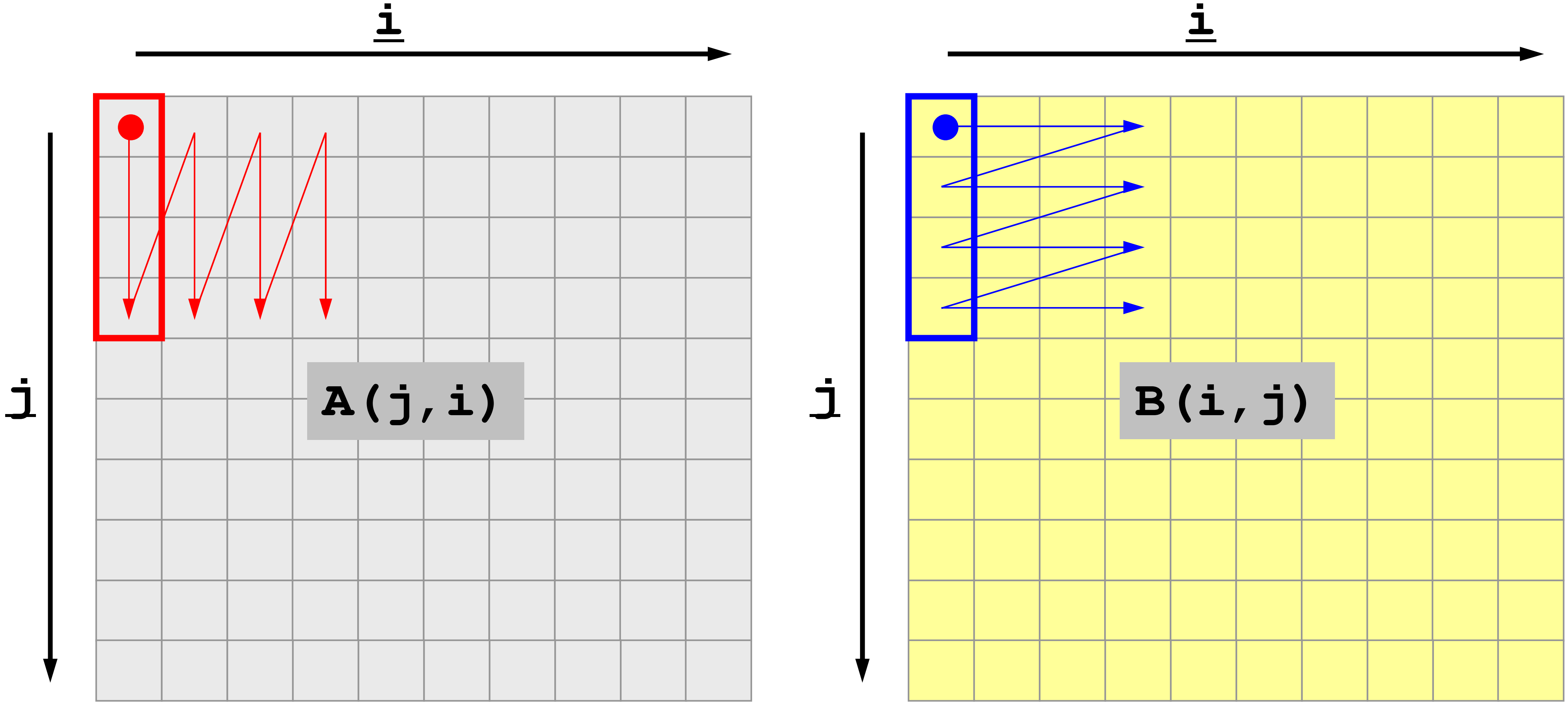
- Therefore, following block-wise access pattern utilizes cache efficiently.



Scalar

Cache Blocking (6/8)
A(1,1), B(1,1)

- □, □ are on cache.

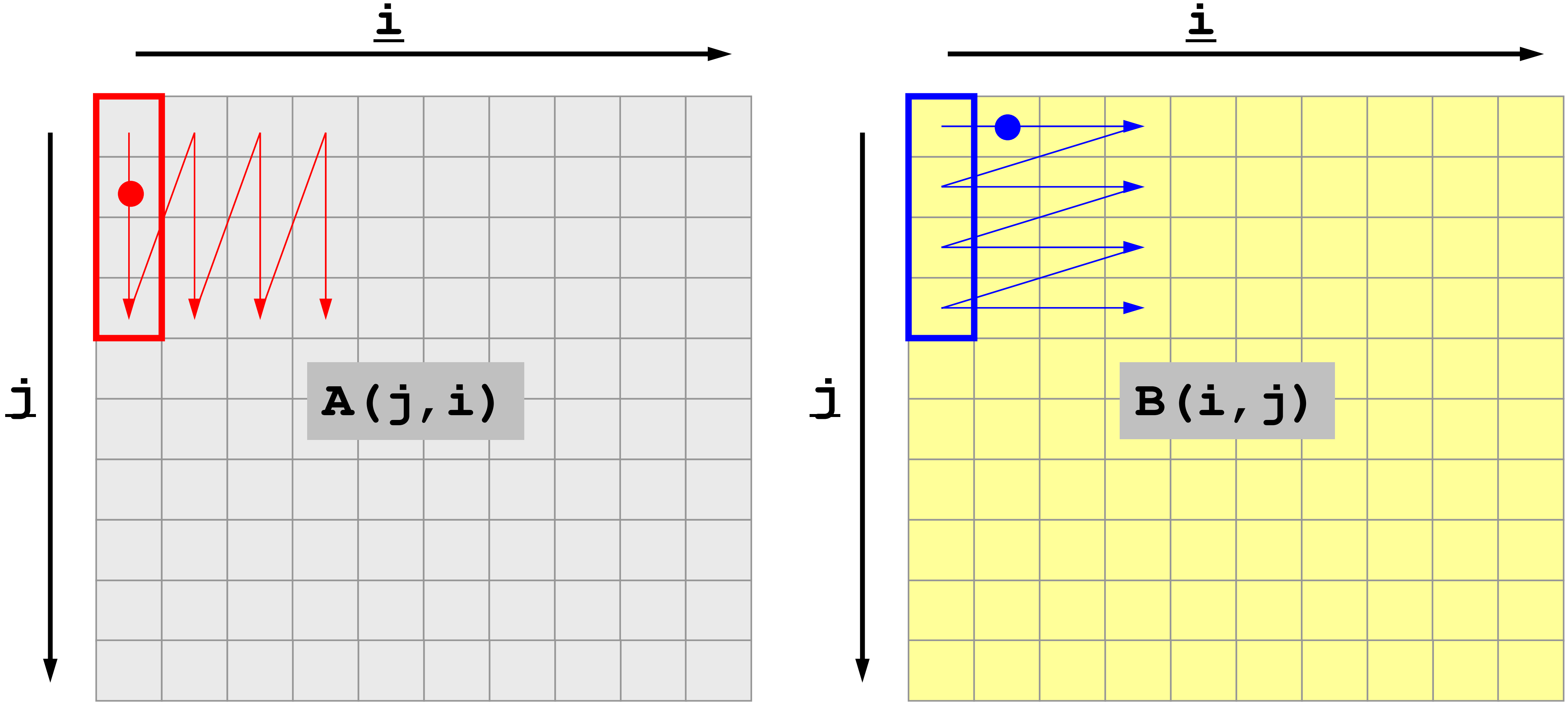


Scalar

Cache Blocking (6/8)
A(2,1), B(1,2)

B(1,2) is not on the cache

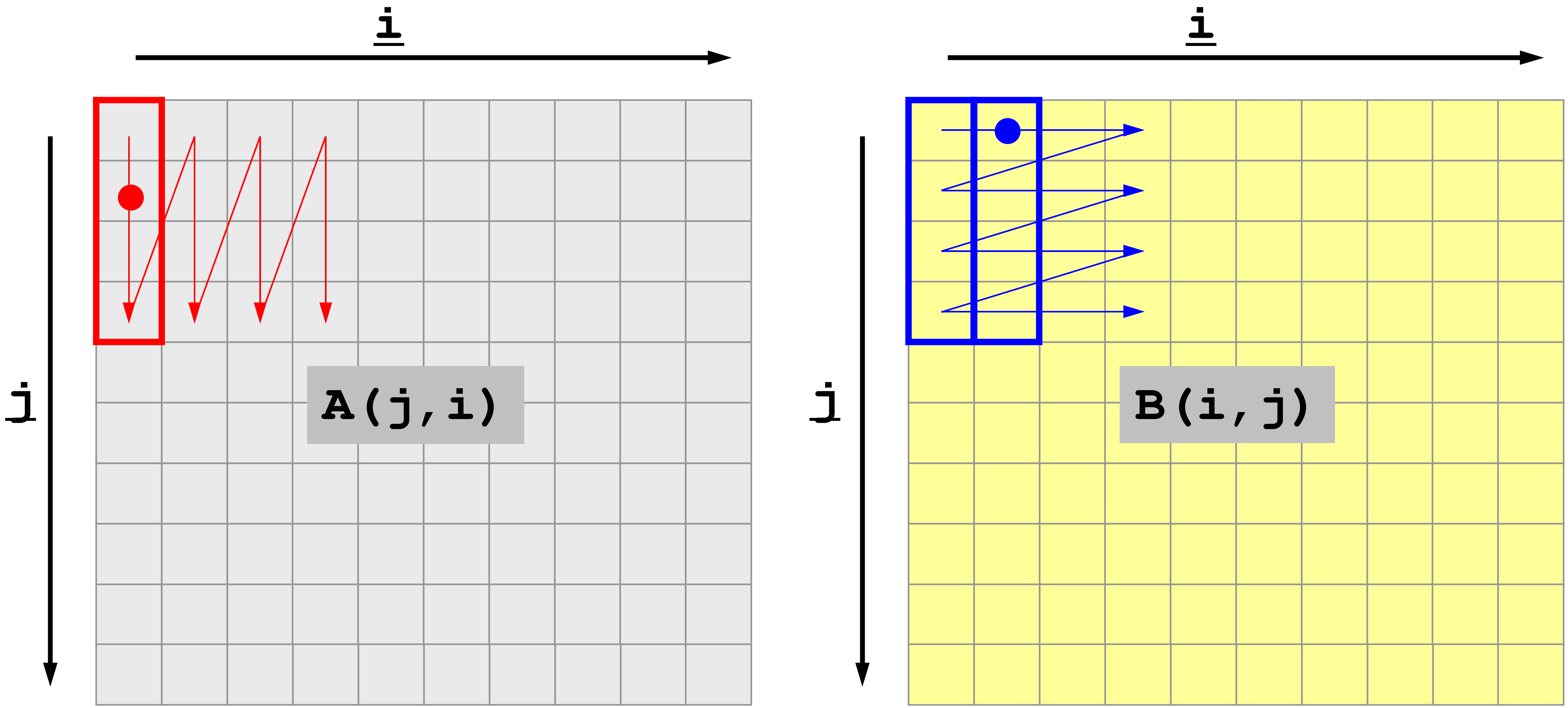
- □, □ are on cache.



Scalar

Cache Blocking (6/8)
A(2,1), B(1,2)

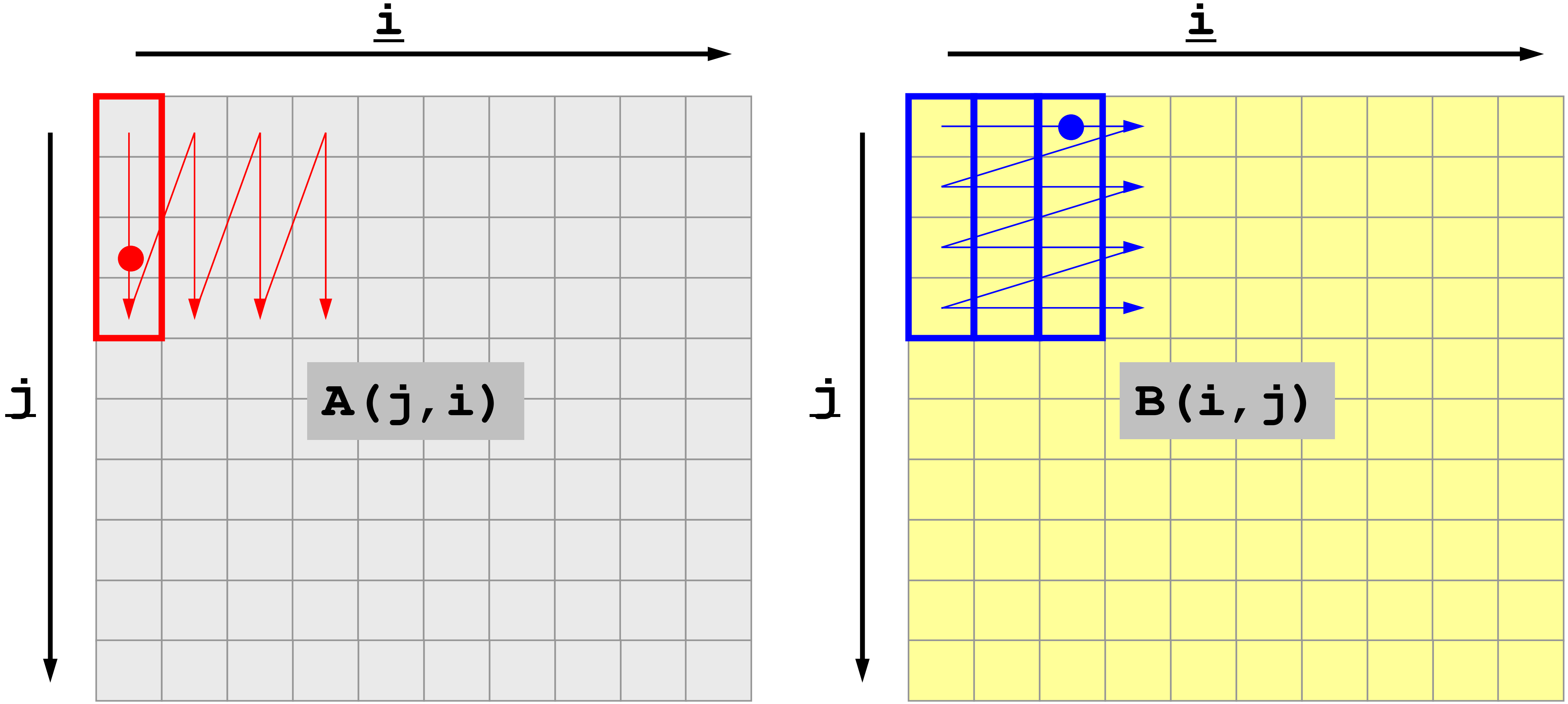
- □, □ are on cache.



Scalar

Cache Blocking (6/8)
A(3,1), B(1,3)

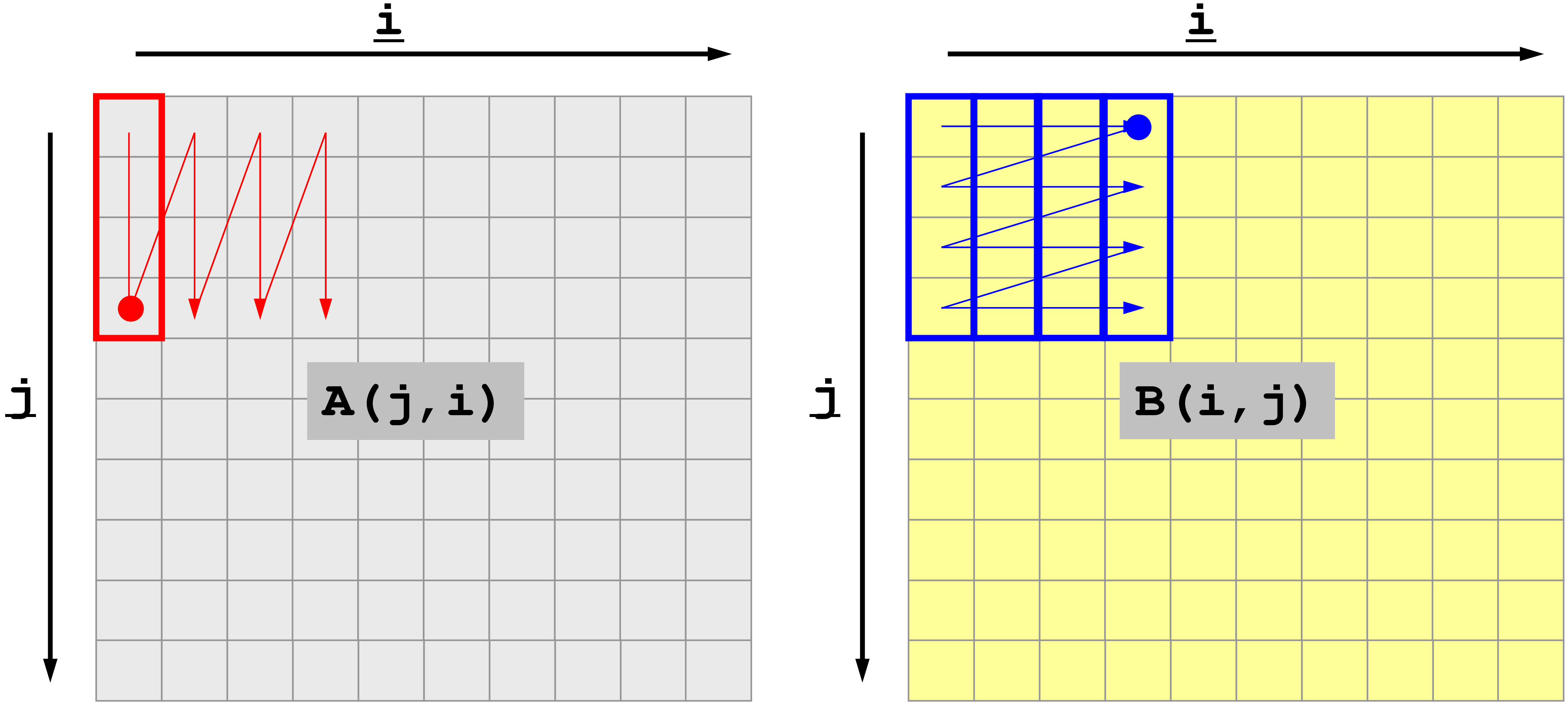
- □, □ are on cache.



Scalar

Cache Blocking (6/8)
A(4,1), B(1,4)

- ☐, ☐ are on cache.

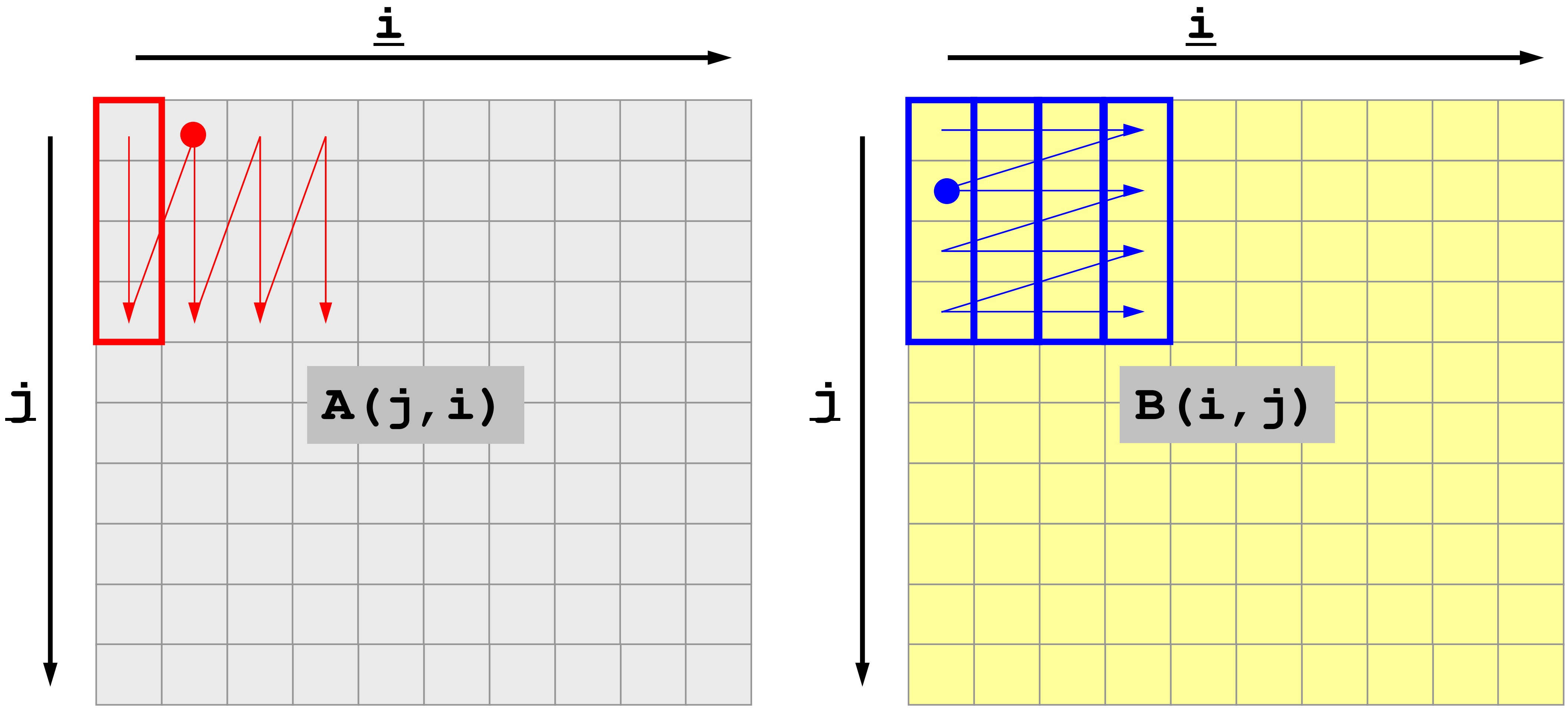


Scalar

Cache Blocking (6/8) A(4,1), B(1,4)

A(4,1) is not on the cache

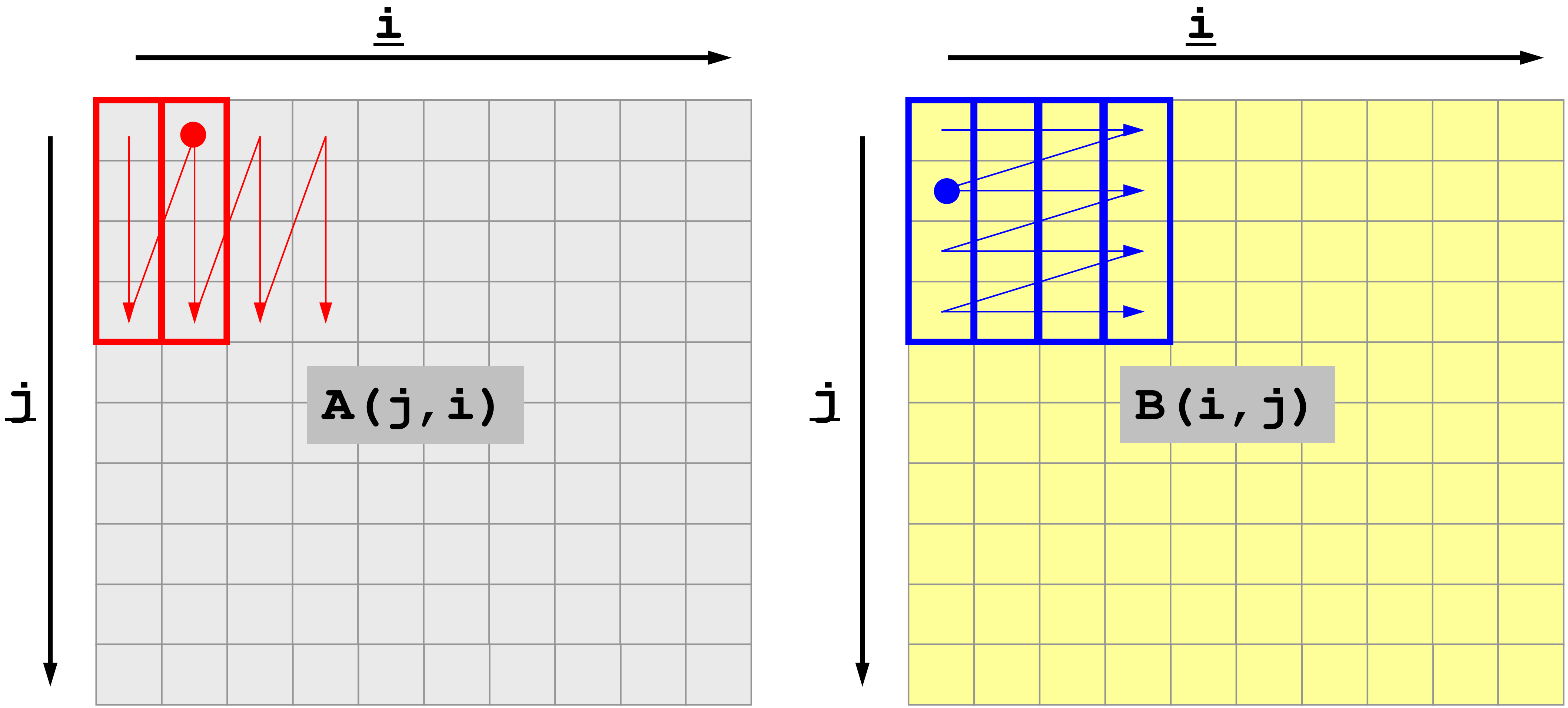
- □, □ are on cache.



Scalar

Cache Blocking (6/8) A(4,1), B(1,4)

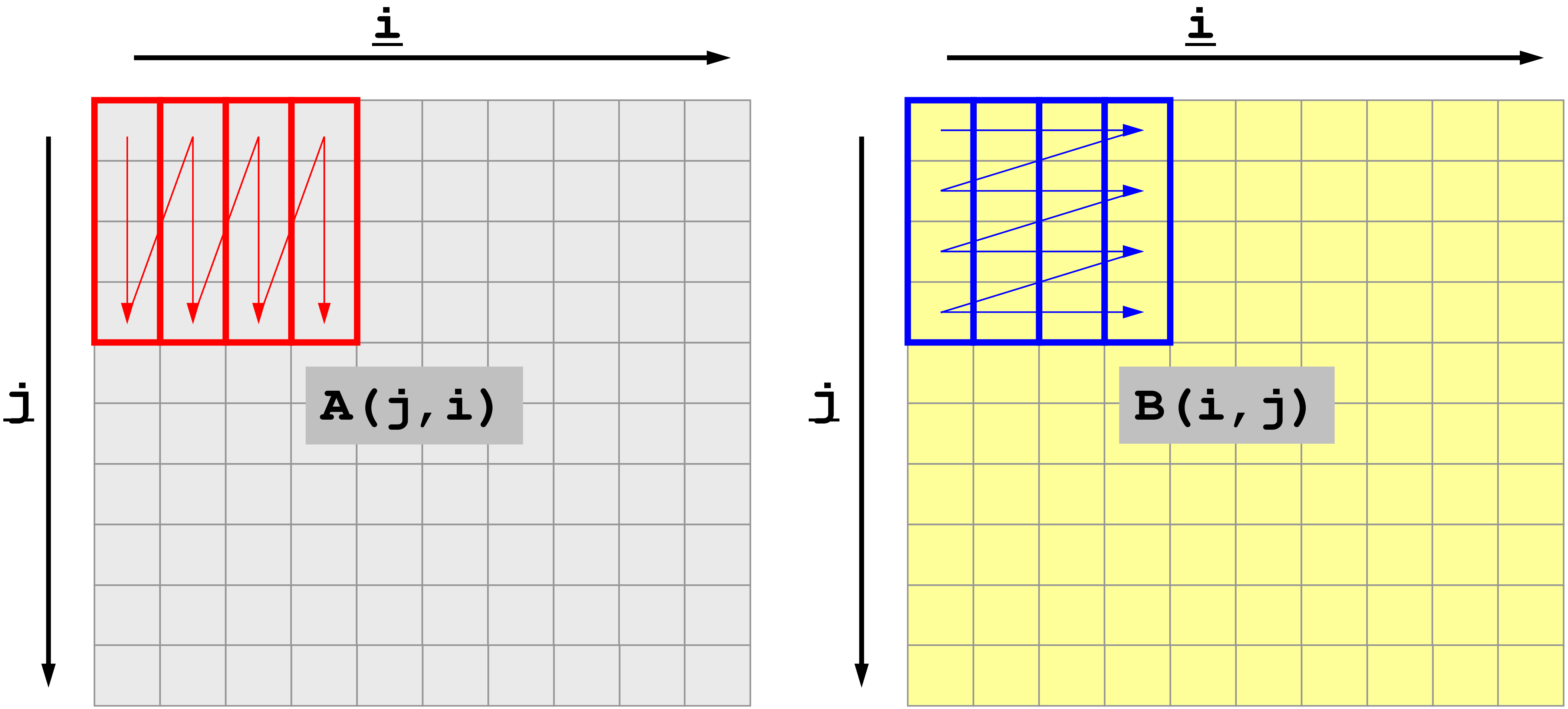
- □, □ are on cache.



Scalar

Cache Blocking (6/8)

- □, □ are on cache.



Cache Blocking (7/8)

- 2d-2.f
- 2x2 block

TLB: Translation Look-aside Buffer

Basic

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo
```

2x2

```
do i= 1, NN-1, 2
  do j= 1, NN-1, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

Loop-Fission is also effective for reduction of cache/TLB miss's.

2x2-b

```
do i= 1, NN-1, 2
  do j= 1, NN/2, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
do i= 1, NN-1, 2
  do j= NN/2+1, NN-1, 2
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

Cache Blocking (8/8): Comp. Time

mpiifort -align array64byte -O3 -axCORE-AVX512 2d-2.f -o 2d-2

```
$> cd /work/gt69/t69XXX
$> cd multicore/omp/dense
$> mpiifort …
$> pjsub 2d-2.sh
$> cat 2d-2.lst

### N ###     500
BASIC     2.389066E-04
2x2       2.340041E-04
2x2-b     2.027079E-04
### N ###    1000
BASIC     8.907821E-04
2x2       8.303802E-04
2x2-b     8.222470E-04

...

### N ###    4000
BASIC     4.876246E-02
2x2       5.577422E-02
2x2-b     4.280486E-02
### N ###    4500
BASIC     4.549238E-02
2x2       5.131447E-02
2x2-b     5.149375E-02
### N ###    5000
BASIC     5.270619E-02
2x2       5.442439E-02
2x2-b     5.649631E-02
```

2d-2.sh

```
#!/bin/sh
#PJM -L rscgrp=lecture9
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt69
#PJM -j
#PJM -e err
#PJM -o 2d-2.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./2d-2
```

No Difference among 3 Implementations

Effect of Cache Blocking is small

Cache Blocking (2x2, Small & Large)

2d-2.f, 2d-2.sh, Small

```
### N ###      500
BASIC     2.389066E-04
2x2       2.340041E-04
2x2-b     2.027079E-04
### N ###     1000
BASIC     8.907821E-04
2x2       8.303802E-04
2x2-b     8.222470E-04
```

```
### N ###     4000
BASIC     4.876246E-02
2x2       5.577422E-02
2x2-b     4.280486E-02
### N ###     4500
BASIC     4.549238E-02
2x2       5.131447E-02
2x2-b     5.149375E-02
### N ###     5000
BASIC     5.270619E-02
2x2       5.442439E-02
2x2-b     5.649631E-02
```

2d-2x.f, 2d-2x.sh, Large

```
### N ###    10000
BASIC     4.272723E-01
2x2       3.517476E-01
2x2-b     2.563077E-01
### N ###    12000
BASIC     7.726939E-01
2x2       6.727859E-01
2x2-b     6.148239E-01
```

```
### N ###    16000
BASIC     1.480378E+00
2x2       1.260129E+00
2x2-b     1.172744E+00
### N ###    18000
BASIC     1.624937E+00
2x2       1.305360E+00
2x2-b     1.130921E+00
### N ###    20000
BASIC     2.081350E+00
2x2       1.738748E+00
2x2-b     1.638656E+00
```

Effect of cache blocking is more significant for larger problems

Cache Blocking (4x4 Blocking) (1/3)

- 2d-3.f
- 4x4 block
- **4x4**

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo

do i= 1, NN-3, 4
  do j= 1, NN-3, 4
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j+2,i  )= A(j+2,i  ) + B(i  ,j+2)
    A(j+3,i  )= A(j+3,i  ) + B(i  ,j+3)
    A(j  ,i+1)= A(j  ,i+1) + B(i+1,j  )
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
    A(j+2,i+1)= A(j+2,i+1) + B(i+1,j+2)
    A(j+3,i+1)= A(j+3,i+1) + B(i+1,j+3)
    A(j  ,i+2)= A(j  ,i+2) + B(i+2,j  )
    A(j+1,i+2)= A(j+1,i+2) + B(i+2,j+1)
    A(j+2,i+2)= A(j+2,i+2) + B(i+2,j+2)
    A(j+3,i+2)= A(j+3,i+2) + B(i+2,j+3)
    A(j  ,i+3)= A(j  ,i+3) + B(i+3,j  )
    A(j+1,i+3)= A(j+1,i+3) + B(i+3,j+1)
    A(j+2,i+3)= A(j+2,i+3) + B(i+3,j+2)
    A(j+3,i+3)= A(j+3,i+3) + B(i+3,j+3)
  enddo
enddo
```

Cache Blocking (4x4 Blocking) (2/3)

- 2d-3.f
- 4x4 block
- **4x4-b**

```
do i= 1, NN-3, 4
  do j= 1, NN/2, 4
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j+2,i  )= A(j+2,i  ) + B(i  ,j+2)
    A(j+3,i  )= A(j+3,i  ) + B(i  ,j+3)
(…)
    A(j  ,i+3)= A(j  ,i+3) + B(i+3,j  )
    A(j+1,i+3)= A(j+1,i+3) + B(i+3,j+1)
    A(j+2,i+3)= A(j+2,i+3) + B(i+3,j+2)
    A(j+3,i+3)= A(j+3,i+3) + B(i+3,j+3)
  enddo
enddo

do i= 1, NN-3, 4
  do j= nn/2+1, NN-3, 4
    A(j  ,i  )= A(j  ,i  ) + B(i  ,j  )
    A(j+1,i  )= A(j+1,i  ) + B(i  ,j+1)
    A(j+2,i  )= A(j+2,i  ) + B(i  ,j+2)
    A(j+3,i  )= A(j+3,i  ) + B(i  ,j+3)
(…)
    A(j  ,i+3)= A(j  ,i+3) + B(i+3,j  )
    A(j+1,i+3)= A(j+1,i+3) + B(i+3,j+1)
    A(j+2,i+3)= A(j+2,i+3) + B(i+3,j+2)
    A(j+3,i+3)= A(j+3,i+3) + B(i+3,j+3)
  enddo
enddo
```

Loop-Fission is also effective for reduction of cache/TLB miss's.

Cache Blocking (4x4 Blocking) (3/3)

mpiifort -align array64byte -O3 -axCORE-AVX512 2d-3.f -o 2d-3

```
$> cd /work/gt69/t69XXX
$> cd multicore/omp/dense
$> mpiifort …
$> pjsub 2d-3.sh
$> cat 2d-3.lst

### N ###      500
BASIC     2.379557E-04
4x4       2.445420E-04
4x4-b     2.049869E-04
### N ###     1000
BASIC     9.401152E-04
4x4       8.805981E-04
4x4-b     7.804884E-04

...

### N ###     4000
BASIC     4.786995E-02
4x4       5.285953E-02
4x4-b     4.543237E-02
### N ###     4500
BASIC     4.510687E-02
4x4       5.675690E-02
4x4-b     5.392428E-02
### N ###     5000
BASIC     5.158540E-02
4x4       7.263093E-02
4x4-b     7.053960E-02
```

2d-3.sh

```
#!/bin/sh
#PJM -L rscgrp=lecture9
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt69
#PJM -j
#PJM -e err
#PJM -o 2d-3.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./2d-3
```

Performance decreases by 4x4 Cache Blocking

Cache Blocking (4x4, Small & Large)

2d-3.f, 2d-3.sh, Small

```
### N ###      500
BASIC     2.379557E-04
4x4       2.445420E-04
4x4-b     2.049869E-04
### N ###     1000
BASIC     9.401152E-04
4x4       8.805981E-04
4x4-b     7.804884E-04
```

2d-3x.f, 2d-3x.sh, Large

```
### N ###    10000
BASIC     4.184356E-01
4x4       3.233387E-01
4x4-b     2.545660E-01
### N ###    12000
BASIC     7.616602E-01
4x4       4.737681E-01
4x4-b     4.696510E-01
```

Effect of cache blocking is more significant for larger problems

```
### N ###     4000
BASIC     4.786995E-02
4x4       5.285953E-02
4x4-b     4.543237E-02
### N ###     4500
BASIC     4.510687E-02
4x4       5.675690E-02
4x4-b     5.392428E-02
### N ###     5000
BASIC     5.158540E-02
4x4       7.263093E-02
4x4-b     7.053960E-02
```

```
### N ###    16000
BASIC     1.469227E+00
4x4       8.759372E-01
4x4-b     8.426449E-01
### N ###    18000
BASIC     1.614159E+00
4x4       1.076962E+00
4x4-b     1.050305E+00
### N ###    20000
BASIC     2.064344E+00
4x4       1.330974E+00
4x4-b     1.330691E+00
```

Cache Blocking (2x2 vs. 4x4)

4x4 is much better for larger problems

Small 2x2

```
### N ###       500
BASIC     2.389066E-04
2x2       2.340041E-04
2x2-b     2.027079E-04
### N ###      1000
BASIC     8.907821E-04
2x2       8.303802E-04
2x2-b     8.222470E-04

...

### N ###      4000
BASIC     4.876246E-02
2x2       5.577422E-02
2x2-b     4.280486E-02
### N ###      4500
BASIC     4.549238E-02
2x2       5.131447E-02
2x2-b     5.149375E-02
### N ###      5000
BASIC     5.270619E-02
2x2       5.442439E-02
2x2-b     5.649631E-02
```

Small 4x4

```
### N ###       500
BASIC     2.379557E-04
4x4       2.445420E-04
4x4-b     2.049869E-04
### N ###      1000
BASIC     9.401152E-04
4x4       8.805981E-04
4x4-b     7.804884E-04

...

### N ###      4000
BASIC     4.786995E-02
4x4       5.285953E-02
4x4-b     4.543237E-02
### N ###      4500
BASIC     4.510687E-02
4x4       5.675690E-02
4x4-b     5.392428E-02
### N ###      5000
BASIC     5.158540E-02
4x4       7.263093E-02
4x4-b     7.053960E-02
```

Large 2x2

```
### N ###     10000
BASIC     4.272723E-01
2x2       3.517476E-01
2x2-b     2.563077E-01
### N ###     12000
BASIC     7.726939E-01
2x2       6.727859E-01
2x2-b     6.148239E-01

...

### N ###     16000
BASIC     1.480378E+00
2x2       1.260129E+00
2x2-b     1.172744E+00
### N ###     18000
BASIC     1.624937E+00
2x2       1.305360E+00
2x2-b     1.130921E+00
### N ###     20000
BASIC     2.081350E+00
2x2       1.738748E+00
2x2-b     1.638656E+00
```

Large 4x4

```
### N ###     10000
BASIC     4.184356E-01
4x4       3.233387E-01
4x4-b     2.545660E-01
### N ###     12000
BASIC     7.616602E-01
4x4       4.737681E-01
4x4-b     4.696510E-01

...

### N ###     16000
BASIC     1.469227E+00
4x4       8.759372E-01
4x4-b     8.426449E-01
### N ###     18000
BASIC     1.614159E+00
4x4       1.076962E+00
4x4-b     1.050305E+00
### N ###     20000
BASIC     2.064344E+00
4x4       1.330974E+00
4x4-b     1.330691E+00
```

- Effect of cache blocking is more significant for larger problems
- (4x4) Blocking is more effective than (2x2) for larger problems

Intel 2018 or 2019 or 2020

```
$> cd /work/gt69/t69XXX
$> cd multicore/omp/dense

$> module avail intel

-- /home/opt/local/modulefiles/L/compiler/intel/2019.5.281 ----------
intelpython/2.7          intelpython/3.6(default)

-- /home/opt/local/modulefiles/L/core ----------------------
intel/2017.4.196   intel/2019.4.243    intel/2020.2.254
intel/2018.3.222   intel/2019.5.281    intel/2020.4.304(default)
intel/2019.3.199   intel/2020.1.217


$> module unload intel/2020.4.304
$> module load intel/2018.3.222

$> mpiifort -align array64byte -O3 -axCORE-AVX512 2d-2.f -o 2d-2b
$> pjsub 2d-2b.sh

$> mpiifort -align array64byte -O3 -axCORE-AVX512 2d-2x.f -o 2d-2xb
$> pjsub 2d-2xb.sh
```

2d-2b.sh: 2020⇒2018

```
#!/bin/sh
#PJM -L rscgrp=lecture9
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt69
#PJM -j
#PJM -e err
#PJM -o 2d-2b.lst

module unload intel/2020.4.304
module load intel/2018.3.222
mpiexec.hydra -n ${PJM_MPI_PROC} ./2d-2b
```

Cache Blocking (Small)

2019: 2d-2.lst, 2020~2019

```
### N ###      500
BASIC     2.389066E-04
2x2       2.340041E-04
2x2-b     2.027079E-04
### N ###     1000
BASIC     8.907821E-04
2x2       8.303802E-04
2x2-b     8.222470E-04
...
### N ###     4000
BASIC     4.876246E-02
2x2       5.577422E-02
2x2-b     4.280486E-02
### N ###     4500
BASIC     4.549238E-02
2x2       5.131447E-02
2x2-b     5.149375E-02
### N ###     5000
BASIC     5.270619E-02
2x2       5.442439E-02
2x2-b     5.649631E-02
```

2018: 2d-2b.lst

```
### N ###      500
BASIC     2.110004E-04
2x2       1.940727E-04
2x2-b     1.769066E-04
### N ###     1000
BASIC     9.388924E-04
2x2       8.418560E-04
2x2-b     8.151531E-04
### N ###     4000
BASIC     4.798794E-02
2x2       5.487990E-02
2x2-b     4.266000E-02
### N ###     4500
BASIC     4.528904E-02
2x2       5.124688E-02
2x2-b     5.134392E-02
### N ###     5000
BASIC     5.185103E-02
2x2       5.468988E-02
2x2-b     5.686903E-02
```

- **No Difference between 2018 & 2019**
- **No Effect of Cache Blocking**

Cache Blocking (Large)

2019: 2d-2x.lst, 2019~2020

```
### N ###    10000
BASIC      4.272723E-01
2x2        3.517476E-01
2x2-b      2.563077E-01
### N ###    12000
BASIC      7.726939E-01
2x2        6.727859E-01
2x2-b      6.148239E-01
[illegible]
BASIC      1.480378E+00
2x2        1.260129E+00
2x2-b      1.172744E+00
### N ###    18000
BASIC      1.624937E+00
2x2        1.305360E+00
2x2-b      1.130921E+00
### N ###    20000
BASIC      2.081350E+00
2x2        1.738748E+00
2x2-b      1.638656E+00
```

2018: 2d-2xb.lst

```
### N ###    10000
BASIC      4.044471E-01
2x2        3.429871E-01
2x2-b      2.522290E-01
### N ###    12000
BASIC      7.317441E-01
2x2        6.534610E-01
2x2-b      5.961268E-01
[illegible]
BASIC      1.411986E+00
2x2        1.234417E+00
2x2-b      1.145437E+00
### N ###    18000
BASIC      1.543609E+00
2x2        1.275395E+00
2x2-b      1.107334E+00
### N ###    20000
BASIC      1.979401E+00
2x2        1.707699E+00
2x2-b      1.606713E+00
```

- No Difference between 2018 & 2019
- Effect of Cache Blockig for Larger Problems

Scalar

Summary: Results on OBCX

- Loop Exchange (i⇒j, or j⇒i)
  - Intel 2018: Significant difference between TYPE-A & TYPE-B
  - Intel 2019, Intel 2020: No difference
    - Compiler is more sophisticated in Intel 2019
    - Loops are automatically transformed

- Cache Blocking
  - No difference between 2018 & 2019/2020 (2018 is rather faster)
  - If the problem size is larger
    - Effect of Cache Blocking becomes more significant
    - 4x4 is much better than 2x2

Summary: Tuning

- Scalar Processor
- Dense Matrices: BLAS

- Optimization of operations for sparse matrices (which appear in this class) is much more difficult (still research topics)
  – Basic idea is same as that for dense matrices.
  – Optimum memory access.

Scalar

Sparse/Dense Matrices

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    Y(i)= Y(i) + AMAT(k)*X(item(k))
  enddo
enddo
```

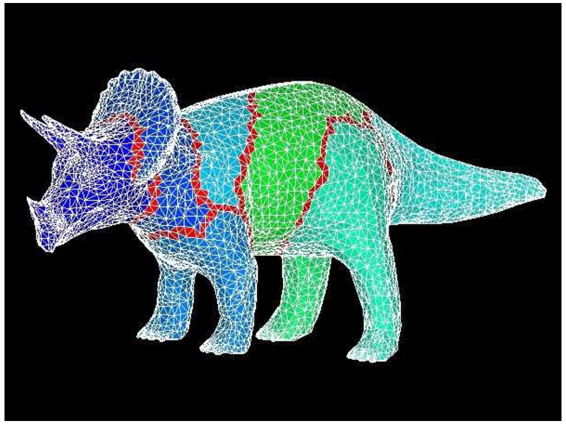
```
do j= 1, N
  do i= 1, N
    Y(j)= Y(j) + A(i,j)*X(i)
  enddo
enddo
```

- “X” in RHS
  – Dense: continuous on memory, easy to utilize cache
  – Sparse: continuity is not assured, difficult to utilize cache
    - more “memory-bound”
- Effective method for sparse matrices: Blocking
  – Reordering: provides “block” features
  – Utilizing physical features of matrices: multiple DOF on each element/node.

Scalar

Summary: Tuning (cont.)

- Sparse Matrices
  - Strategy for tuning may depend on alignment of data components.
  - Program may change according to alignment of data components.
- Dense Matrices
  - Structured, Regularity
  - Performance mainly depends on machine parameters & mat. size.
    - Effect of options of compilers
    - Automatic tuning (AT) is applicable.
  - Libraries
    - ATLAS (Automatic Tuning)
      - http://math-atlas.sourceforge.net/
    - GoToBLAS (Manual Tuning)
      - Kazushige Goto (Microsoft)



Reedbush-U & OBCX

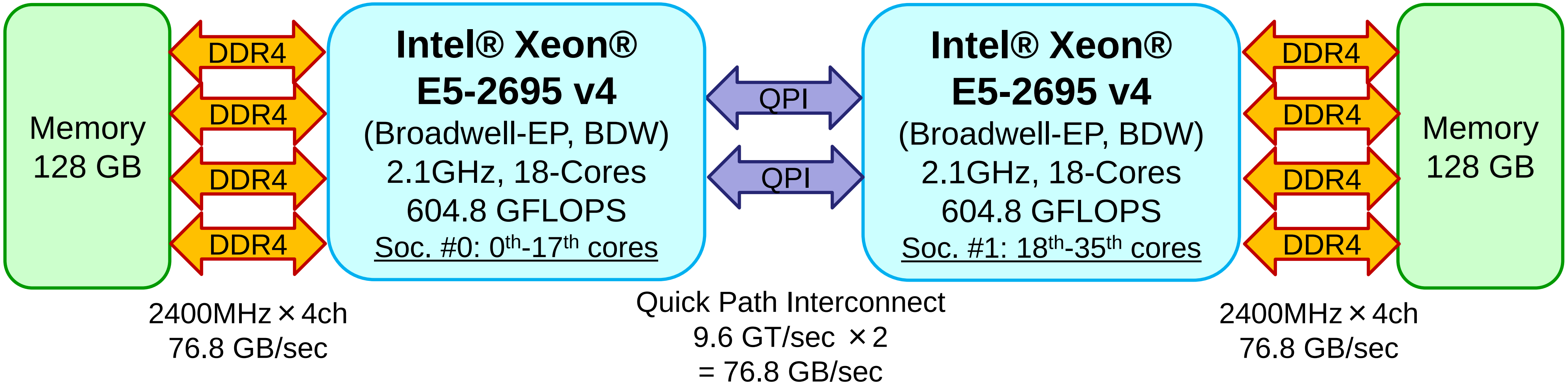
2,600+ users (55+% from outside of U.Tokyo)

- Reedbush (HPE, Intel BDW + NVIDIA P100 (Pascal))
  - Integrated Supercomputer System for Data Analyses & Scientific Simulations
    - Jul.2016-Mar.2021

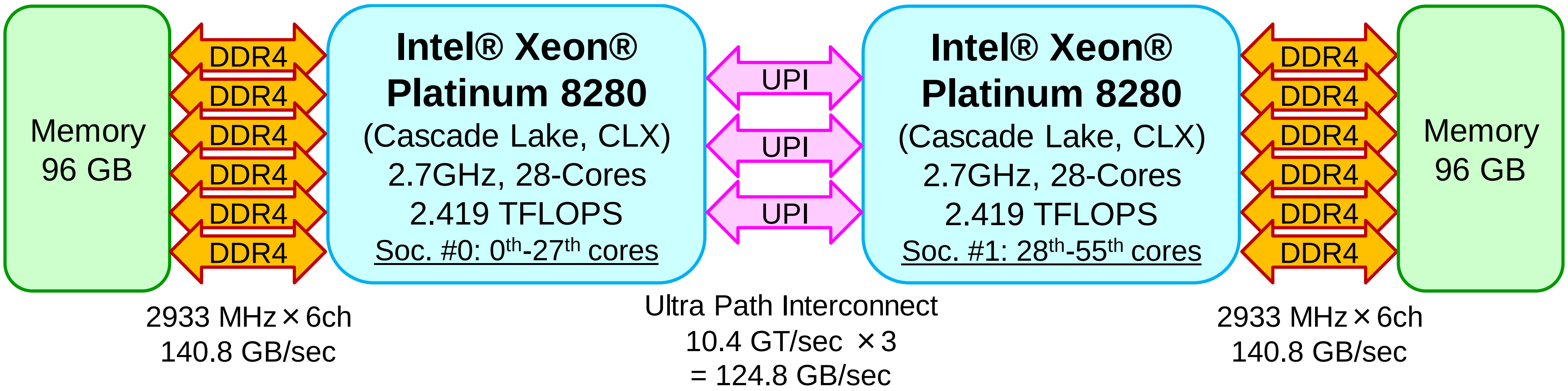
  - Our first GPU System, DDN IME (Burst Buffer)
  - **Reedbush-U: CPU only, 420 nodes, 508 TF (Retired on June 30, 2020)**
  - Reedbush-H: 120 nodes, 2 GPUs/node: 1.42 PF (Mar.2017)
  - Reedbush-L: 64 nodes, 4 GPUs/node: 1.43 PF (Oct.2017)
- Oakforest-PACS (OFP) (Fujitsu, Intel Xeon Phi (KNL))
  - JCAHPC (U.Tsukuba & U.Tokyo)
  - 25 PF, #19 in 55th TOP 500 (June 2020) (#3 in Japan)
  - Omni-Path Architecture, DDN IME (Burst Buffer)

- Oakbridge-CX (OBCX) (Fujitsu, Intel Xeon Platinum 8280, CLX)
  - Massively Parallel Supercomputer System
  - 6.61 PF, #60 in 55th TOP 500, July 2019-June 2023
  - SSD's are installed to 128 nodes (out of 1,368)

Reedbush-U: intel/18.1.163 (default)



Oakbridge-CX (OBCX): intel/2019.5.281 (default)



Reedbush-U Intel BDW	Capacity	X-Way Set Associative	Cache Line
L1 Data	32 KB/core	8-way	64 B
L1 Instruction	32 KB/core	8-way	64 B
L2	**256 KB/core**	**8-way**	**64 B**
L3	45 MB/socket	20-way	64 B

Oakbridge-CX Intel CLX	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	**1.00 MB/core**	**16-Way**	**64B**
L3	38.5 MB/socket	11-Way	64B

Scalar

Loop Exchanging

Reedbush-U (Intel 2018)

```
### N ###        500
WORSE         2.307892E-04  TYPE-A
BETTER        1.819134E-04  TYPE-B
### N ###       1000
WORSE         1.492023E-03
BETTER        7.221699E-04
### N ###       1500
WORSE         5.147934E-03
BETTER        1.756191E-03
### N ###       2000
WORSE         1.727295E-02
BETTER        5.671978E-03
### N ###       2500
WORSE         3.727102E-02
BETTER        1.051402E-02
### N ###       3000
WORSE         7.017899E-02
BETTER        1.530194E-02
### N ###       3500
WORSE         8.954906E-02
BETTER        2.009606E-02
### N ###       4000
WORSE         1.272562E-01
BETTER        2.680802E-02
```

OBCX (Intel 2019)

```
### N ###        500
WORSE         1.521902E-04
BETTER        1.496547E-04
### N ###       1000
WORSE         6.083571E-04
BETTER        6.320321E-04
### N ###       1500
WORSE         1.633597E-03
BETTER        1.578360E-03
### N ###       2000
WORSE         3.586689E-03
BETTER        3.616580E-03
### N ###       2500
WORSE         6.087084E-03
BETTER        6.115762E-03
### N ###       3000
WORSE         9.118990E-03
BETTER        9.118399E-03
### N ###       3500
WORSE         1.249523E-02
BETTER        1.251535E-02
### N ###       4000
WORSE         1.669729E-02
BETTER        1.657084E-02
```

Scalar

Loop Exchanging on RBU

Reedbush-U (Intel 2018)

```
### N ###      500
WORSE        2.307892E-04 TYPE-A
BETTER       1.819134E-04 TYPE-B
### N ###     1000
WORSE        1.492023E-03
BETTER       7.221699E-04
### N ###     2500
WORSE        3.727102E-02
BETTER       1.051402E-02
### N ###     3000
WORSE        7.017899E-02
BETTER       1.530194E-02
### N ###     3500
WORSE        8.954906E-02
BETTER       2.009606E-02
### N ###     4000
WORSE        1.272562E-01
BETTER       2.680802E-02
```

Reedbush-U (Intel 2019)

```
### N ###      500
WORSE        1.972821E-04
BETTER       1.896890E-04
### N ###     1000
WORSE        7.693437E-04
BETTER       7.542712E-04
### N ###     2500
WORSE        1.069489E-02
BETTER       1.096442E-02
### N ###     3000
WORSE        1.555246E-02
BETTER       1.570564E-02
### N ###     3500
WORSE        2.116195E-02
BETTER       2.129073E-02
### N ###     4000
WORSE        2.754035E-02
BETTER       2.761351E-02
```

- **Significant difference between TYPE-A & TYPE-B for Intel 2018**
- **No difference for Intel 2019**

Cache Blocking (Small)

Reedbush-U (Intel 2018)

```
### N ###       500
BASIC      3.120899E-04
2x2        2.901554E-04
2x2-b      2.548695E-04
### N ###      1000
BASIC      1.405001E-03
2x2        1.138926E-03
2x2-b      1.201868E-03
```

OBCX (Intel 2019)

```
### N ###       500
BASIC      2.389066E-04
2x2        2.340041E-04
2x2-b      2.027079E-04
### N ###      1000
BASIC      8.907821E-04
2x2        8.303802E-04
2x2-b      8.222470E-04
```

- **Effect of cache blocking is more significant on RBU**

Reedbush-U (Intel 2018)

```
### N ###      4000
BASIC      1.250241E-01
2x2        7.714486E-02
2x2-b      6.398797E-02
### N ###      4500
BASIC      1.862700E-01
2x2        1.056359E-01
2x2-b      8.264184E-02
### N ###      5000
BASIC      2.493391E-01
2x2        1.350009E-01
2x2-b      1.066360E-01
```

OBCX (Intel 2019)

```
### N ###      4000
BASIC      4.876246E-02
2x2        5.577422E-02
2x2-b      4.280486E-02
### N ###      4500
BASIC      4.549238E-02
2x2        5.131447E-02
2x2-b      5.149375E-02
### N ###      5000
BASIC      5.270619E-02
2x2        5.442439E-02
2x2-b      5.649631E-02
```

Cache Blocking (Large)

Reedbush-U (Intel 2018)

```
### N ###    10000
BASIC      1.326623E+00
2x2        6.905529E-01
2x2-b      6.224930E-01
### N ###    12000
BASIC      2.027832E+00
```

```
2x2        1.966102E+00
2x2-b      1.858198E+00
### N ###    18000
BASIC      4.660499E+00
2x2        2.537638E+00
2x2-b      2.425502E+00
### N ###    20000
BASIC      5.811255E+00
2x2        3.141790E+00
2x2-b      3.028385E+00
```

OBCX (Intel 2019)

```
### N ###    10000
BASIC      4.272723E-01
2x2        3.517476E-01
2x2-b      2.563077E-01
### N ###    12000
BASIC      7.726939E-01
```

```
2x2        1.260129E+00
2x2-b      1.172744E+00
### N ###    18000
BASIC      1.624937E+00
2x2        1.305360E+00
2x2-b      1.130921E+00
### N ###    20000
BASIC      2.081350E+00
2x2        1.738748E+00
2x2-b      1.638656E+00
```

Effect of cache blocking is:

- **more significant on RBU**
- **more significant for larger problems on both of RBU and OBCX**

Cache Blocking (Small)

Reedbush-U (Intel 2018)

```
### N ###       500
BASIC      3.120899E-04
2x2        2.901554E-04
2x2-b      2.548695E-04
### N ###      1000
BASIC      1.405001E-03
2x2        1.138926E-03
2x2-b      1.201868E-03

...

### N ###      4000
BASIC      1.250241E-01
2x2        7.714486E-02
2x2-b      6.398797E-02
### N ###      4500
BASIC      1.862700E-01
2x2        1.056359E-01
2x2-b      8.264184E-02
### N ###      5000
BASIC      2.493391E-01
2x2        1.350009E-01
2x2-b      1.066360E-01
```

Reedbush-U (Intel 2019)

```
### N ###       500
BASIC      3.232858E-04
2x2        2.933713E-04
2x2-b      4.021674E-04
### N ###      1000
BASIC      1.430064E-03
2x2        1.118408E-03
2x2-b      1.158118E-03

### N ###      4000
BASIC      1.320258E-01
2x2        8.029237E-02
2x2-b      6.571975E-02
### N ###      4500
BASIC      1.962729E-01
2x2        1.102015E-01
2x2-b      8.439950E-02
### N ###      5000
BASIC      2.627493E-01
2x2        1.435239E-01
2x2-b      1.113254E-01
```

- **No Difference between 2018 & 2019**

Cache Blocking (Large)

Reedbush-U (Intel 2018)

```
### N ###    10000
BASIC      1.326623E+00
2x2        6.905529E-01
2x2-b      6.224930E-01
### N ###    12000
BASIC      2.027832E+00
2x2        1.095561E+00
2x2-b      9.994931E-01

...

### N ###    16000
BASIC      3.627561E+00
2x2        1.966102E+00
2x2-b      1.858198E+00
### N ###    18000
BASIC      4.660499E+00
2x2        2.537638E+00
2x2-b      2.425502E+00
### N ###    20000
BASIC      5.811255E+00
2x2        3.141790E+00
2x2-b      3.028385E+00
```

Reedbush-U (Intel 2019)

```
### N ###    10000
BASIC      1.323399E+00
2x2        7.208769E-01
2x2-b      6.537885E-01
### N ###    12000
BASIC      2.020611E+00
2x2        1.102579E+00
2x2-b      1.022119E+00
### N ###    16000
BASIC      3.874089E+00
2x2        2.125244E+00
2x2-b      2.030426E+00
### N ###    18000
BASIC      4.997312E+00
2x2        2.739942E+00
2x2-b      2.654234E+00
### N ###    20000
BASIC      6.232887E+00
2x2        3.446736E+00
2x2-b      3.360425E+00
```

- **No Difference between 2018 & 2019**

Cache Blocking (Small)

Reedbush-U (Intel 2018)

```
### N ###        500
BASIC      3.120899E-04
2x2        2.901554E-04
2x2-b      2.548695E-04
### N ###       1000
BASIC      1.405001E-03
2x2        1.138926E-03
2x2-b      1.201868E-03

...

### N ###       4000
BASIC      1.250241E-01
2x2        7.714486E-02
2x2-b      6.398797E-02
### N ###       4500
BASIC      1.862700E-01
2x2        1.056359E-01
2x2-b      8.264184E-02
### N ###       5000
BASIC      2.493391E-01
2x2        1.350009E-01
2x2-b      1.066360E-01
```

```
### N ###        500
BASIC      3.161430E-04
4x4        2.541542E-04
4x4-b      4.870892E-04
### N ###       1000
BASIC      1.394033E-03
4x4        1.115084E-03
4x4-b      1.128912E-03

...

### N ###       4000
BASIC      1.254189E-01
4x4        5.854082E-02
4x4-b      5.547309E-02
### N ###       4500
BASIC      1.855319E-01
4x4        7.502913E-02
4x4-b      7.122779E-02
### N ###       5000
BASIC      2.488911E-01
4x4        9.344006E-02
4x4-b      9.031606E-02
```

OBCX (Intel 2019)

```
### N ###        500
BASIC      2.389066E-04
2x2        2.340041E-04
2x2-b      2.027079E-04
### N ###       1000
BASIC      8.907821E-04
2x2        8.303802E-04
2x2-b      8.222470E-04

...

### N ###       4000
BASIC      4.876246E-02
2x2        5.577422E-02
2x2-b      4.280486E-02
### N ###       4500
BASIC      4.549238E-02
2x2        5.131447E-02
2x2-b      5.149375E-02
### N ###       5000
BASIC      5.270619E-02
2x2        5.442439E-02
2x2-b      5.649631E-02
```

```
### N ###        500
BASIC      2.379557E-04
4x4        2.445420E-04
4x4-b      2.049869E-04
### N ###       1000
BASIC      9.401152E-04
4x4        8.805981E-04
4x4-b      7.804884E-04

...

### N ###       4000
BASIC      4.786995E-02
4x4        5.285953E-02
4x4-b      4.543237E-02
### N ###       4500
BASIC      4.510687E-02
4x4        5.675690E-02
4x4-b      5.392428E-02
### N ###       5000
BASIC      5.158540E-02
4x4        7.263093E-02
4x4-b      7.053960E-02
```

- Effect of cache blocking is more significant on RBU
- 4x4 is better on RBU, but it is worse on OBCX

Cache Blocking (Large)

Reedbush-U (Intel 2018)

```
### N ###    10000
BASIC     1.326623E+00
2x2       6.905529E-01
2x2-b     6.224930E-01
### N ###    12000
BASIC     2.027832E+00
2x2       1.095561E+00
2x2-b     9.994931E-01

...

### N ###    16000
BASIC     3.627561E+00
2x2       1.966102E+00
2x2-b     1.858198E+00
### N ###    18000
BASIC     4.660499E+00
2x2       2.537638E+00
2x2-b     2.425502E+00
### N ###    20000
BASIC     5.811255E+00
2x2       3.141790E+00
2x2-b     3.028385E+00
```

```
### N ###    10000
BASIC     1.337083E+00
4x4       4.136867E-01
4x4-b     4.006213E-01
### N ###    12000
BASIC     2.043310E+00
4x4       6.148252E-01
4x4-b     5.941429E-01

...

### N ###    16000
BASIC     3.874410E+00
4x4       1.126769E+00
4x4-b     1.092742E+00
### N ###    18000
BASIC     5.001099E+00
4x4       1.443131E+00
4x4-b     1.403199E+00
### N ###    20000
BASIC     6.229637E+00
4x4       1.794927E+00
4x4-b     1.752852E+00
```

OBCX (Intel 2019)

```
### N ###    10000
BASIC     4.272723E-01
2x2       3.517476E-01
2x2-b     2.563077E-01
### N ###    12000
BASIC     7.726939E-01
2x2       6.727859E-01
2x2-b     6.148239E-01

...

### N ###    16000
BASIC     1.480378E+00
2x2       1.260129E+00
2x2-b     1.172744E+00
### N ###    18000
BASIC     1.624937E+00
2x2       1.305360E+00
2x2-b     1.130921E+00
### N ###    20000
BASIC     2.081350E+00
2x2       1.738748E+00
2x2-b     1.638656E+00
```

```
### N ###    10000
BASIC     4.184356E-01
4x4       3.233387E-01
4x4-b     2.545660E-01
### N ###    12000
BASIC     7.616602E-01
4x4       4.737681E-01
4x4-b     4.696510E-01

...

### N ###    16000
BASIC     1.469227E+00
4x4       8.759372E-01
4x4-b     8.426449E-01
### N ###    18000
BASIC     1.614159E+00
4x4       1.076962E+00
4x4-b     1.050305E+00
### N ###    20000
BASIC     2.064344E+00
4x4       1.330974E+00
4x4-b     1.330691E+00
```

Effect of cache blocking is more significant

- on RBU
- for larger problems with 4x4

Summary: Results on OBCX/RBU

- Loop Exchange (i⇒j, or j⇒i)
  - Intel 2018: Significant difference between TYPE-A & TYPE-B
  - Intel 2019: No difference
    - Compiler is more sophisticated in Intel 2019
    - Loops are automatically transformed
- Cache Blocking
  - No difference between 2018 & 2019 (2018 is rather faster)
  - If the problem size is larger
    - Effect of Cache Blocking becomes more significant
    - 4x4 is much better than 2x2
  - **Effect of Cache Blocking is more significant for RBU, especially if the problem size is smaller**

Scalar

Difference of Behavior: OBCX/RBU

Reedbush-U Intel BDW	Capacity	X-Way Set Associative	Cache Line
L1 Data	32 KB/core	8-way	64 B
L1 Instruction	32 KB/core	8-way	64 B
L2	**256 KB/core**	**8-way**	**64 B**
L3	45 MB/socket	20-way	64 B

Oakbridge-CX Intel CLX	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	**1.00 MB/core**	**16-Way**	**64B**
L3	38.5 MB/socket	11-Way	64B

- Size of L2-Cache
  - RBU 256 KB/core
  - OBCX 1 MB/core
- Change of “Cache Policy”
  - RBU
    - Inclusive: copy of data-sets in L2-cache is also stored in L3-cache
  - OBCX
    - Non-Inclusive: copy of data-sets in L2-cache is NOT stored in L3-cache, larger L2, smaller L3
  - Non-inclusive policy was adopted at Skylake Generation (between Broadwell (RBU) and Cascadelake (OBCX))

Reedbush-U Intel BDW	Capacity	X-Way Set Associative	Cache Line
L1 Data	32 KB/core	8-way	64 B
L1 Instruction	32 KB/core	8-way	64 B
L2	**256 KB/core**	**8-way**	**64 B**
L3	45 MB/socket	20-way	64 B

Oakbridge-CX Intel CLX	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	**1.00 MB/core**	**16-Way**	**64B**
L3	38.5 MB/socket	11-Way	64B