Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«Code Optimizations - II»

Παναγιώτης Χατζηδούκας

Outline

- Data and control dependencies
- Data dependencies in loops
- Advanced code transformations
  - Loop merge (fusion)
  - Loop distribution (fission)
  - Loop reversal
  - Loop peeling
  - Loop bump
  - Array copy transformation
  - Software prefetching
  - Loop tiling/blocking

Dependencies in programs (1)

- Data dependencies
  - statement S3 cannot be moved before either S1 or S2 without producing incorrect values

```
S1: PI=3.14;
S2: R=5.0;
S3: AREA=2 * PI * R
```

- Control dependencies
  - statement S2 cannot be executed before S1 in a correctly transformed program, because the execution of S2 is conditional upon the execution of the branch in S1
  - Statement S3 cannot be executed before S2

```
S1: if (temp==0)
S2:     a=5.0;
S3: a=3.0;
```

Dependencies in programs (2)

- Definition: There is a data dependence from statement S1 to statement S2 (statement S2 depends on statement S1) if and only if
  - both statements access the same memory location and at least
  - one of them stores into it and
  - there is a feasible run-time execution path from S1 to S2.

Data Dependencies – classification

- Data dependencies reside into 3 categories
  - Read after Write (RAW) or true dependence
  - Write after Read (WAR) or anti-dependence
  - Write after Write (WAW) or output dependence

T=…
…=T

…=T
T=…

T=…
T=…

A:
```
S1:  PI=3.14;
S2:  R=2;
S3:  S=2 x PI x R    //S3 cannot be executed before S1, S2 – true dependence
```

B:
```
S1:  T1=R1;      //S3 cannot be executed before or in parallel with S1 – anti-
S2:  R2=PI-T1;   //dependence. But it can be eliminated by applying register
S3:  R1=PI+S;    //renaming – this is why it is called 'anti' dependence
```

⇩

```
S1:  T1=R1;
S2:  R2=PI-T1;
S3:  R3=PI+S;
```

C:
```
S1:  T1=R1;          ⇨    S1:  T1=R1;
S2:  T1=R2+5;             S2:  T2=R2+5;
```

WAW dependence is eliminated by applying register renaming

Data Dependencies – Terminology

- Data dependencies
  - Read after Write (RAW) or true dependence

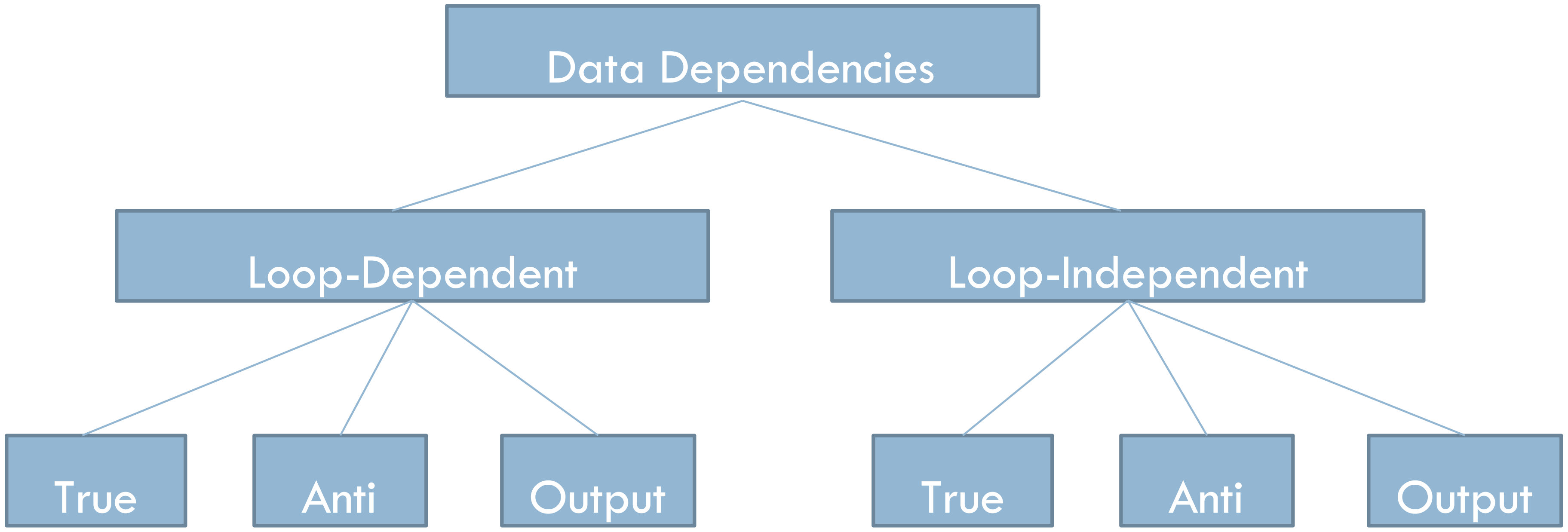
    $S1 \xrightarrow{\delta^1} S2$ *OR* $S1 \xrightarrow{\delta} S2$
  - Write after Read (WAR) or anti-dependence

    $S1 \xrightarrow{\delta^{-1}} S2$
  - Write after Write (WAW) or output dependence

    $S1 \xrightarrow{\delta^0} S2$

- The convention for graphically displaying dependence is to depict the edge as flowing from the statement that executes first (the source) to the one that executes later (the sink).
  - Here S2 depends on S1

Data Dependencies – classification



Data Dependencies in loops

- Loop dependent dependencies
  - the statement S1 on any loop iteration depends on the instance of itself from the previous iteration.
  - A true dependence occurs for each different colour
  - The program writes in iteration i and reads in iteration i+1
  - The iterations cannot be executed in parallel

```
    for (i = 1; i<N i++)
S1:     A(i+1) = A(i) + B(i)

              ⇩

i=1 : A[2] = A[1] + B[1]
i=2 : A[3] = A[2] + B[1]
i=3 : A[4] = A[3] + B[3]
i=4 : A[5] = A[4] + B[4]
i=5 : A[6] = A[5] + B[5]
                  ...
```

Loop dependent dependencies: terminology

- On the right, there is a loop-dependent true dependence

$$\begin{tikzcd} S1 \arrow[r, "\delta_1^1"] & S1 \end{tikzcd}$$

True, Anti, Output

$$\delta_n^{1,\,-1,\,0}$$

Nesting level value for loop-dependent dependencies or '∞' for loop-independent dependencies

```
for (i = 1; i<N i++)
S1:    A(i+1) = A(i) + B(i)
```

```
i=1 : A[2] = A[1] + B[1]
i=2 : A[3] = A[2] + B[1]
i=3 : A[4] = A[3] + B[3]
i=4 : A[5] = A[4] + B[4]
i=5 : A[6] = A[5] + B[5]
             ...
```

Loop dependent dependencies: example

- Now, the distance of the dependence is 2
- Therefore i=1 and i=2 can be executed in parallel – no dependence exists

S1: for (i = 1; i<N i++)
S2: A(i+2) = A(i) + B(i)

$$S2 \xrightarrow{\delta_1^{1}} S2$$

i=1 : A[3] = A[1] + B[1]
i=2 : A[4] = A[2] + B[1]
i=3 : A[5] = A[3] + B[3]
i=4 : A[6] = A[4] + B[4]
i=5 : A[7] = A[5] + B[5]
i=6 : A[8] = A[6] + B[5]
…

- No dependence exists between 2 iterations
- they can be executed in parallel or vectorized

Distance Vector & Direction Vector

- It is convenient to characterize dependences by the distance between the source and sink of a dependence in the iteration space
- We express this in terms of distance vectors and direction vectors
- Distance Vector
  - Suppose that there is a dependence from S1 on iteration **i** (of a loop nest of n loops) to S2 on iteration **j**, then the dependence distance vector **d(i,j)** is defined as a vector of length **n** such that $\mathbf{d(i,j)_k = j_k - i_k}$
- Direction Vector: is defined as a vector of length **n** such that

$$D(i,j)_k = \begin{array}{l} \text{“}<\text{” if } d(i,j)_k > 0 \\ \text{“}=\text{” if } d(i,j)_k = 0 \\ \text{“}>\text{” if } d(i,j)_k < 0 \end{array}$$

Data Dependencies: example

```
for (i = 1; i<10; i++)
  for (j = 0; j<20; j++)
    for (k = 0; k<100; k++)
      for (n = 2; n<80; n++)
S1:   A(i, j+2, k, n) = A(i, j, k, n+1) + temp;
```

$$S1 \xrightarrow{\delta_2^1} S1$$

- Distance vector: $d(i, j, k, n) = (0, \underset{\uparrow\ \delta_2^1}{2}, 0, -1)$

- Direction vector: D(i, j, k, n) = (=, <, =, >)
- The dependence is always given by the leftmost non '=' symbol

Loop Merge / Loop Fusion (1)

- Loop Merge is a transformation that combines 2 independent loop kernels that have the same loop bounds and number of iterations
- This transformation is not always safe
  - data dependencies must be preserved

```
for (i=1; i<N; i++)
 A[ i ] = B[ i ];

for (i=1; i<N; i++)
 B[ i ] = A[ i-1 ];
```

➡

```
for (i=1; i<N; i++){
 A[ i ] = B[ i ];
 B[ i ] = A[ i-1 ];
}
```

Loop Merge / Loop Fusion (2)

- Benefits
  - Reduces the number of arithmetical instructions
    - Remember each loop is transformed into an add, compare and jump assembly instruction
  - May improve data reuse
  - May enable other loop transformations
- Drawbacks:
  - May increase register pressure
  - May hurt data locality (extra cache misses)
  - May hurt instruction cache performance

```
for (i=1; i<N; i++)
 A[ i ] = B[ i ];

for (i=1; i<N; i++)
 B[ i ] = A[ i-1 ];
```

$\Rightarrow$

```
for (i=1; i<N; i++){
 A[ i ] = B[ i ];
 B[ i ] = A[ i-1 ];
}
```

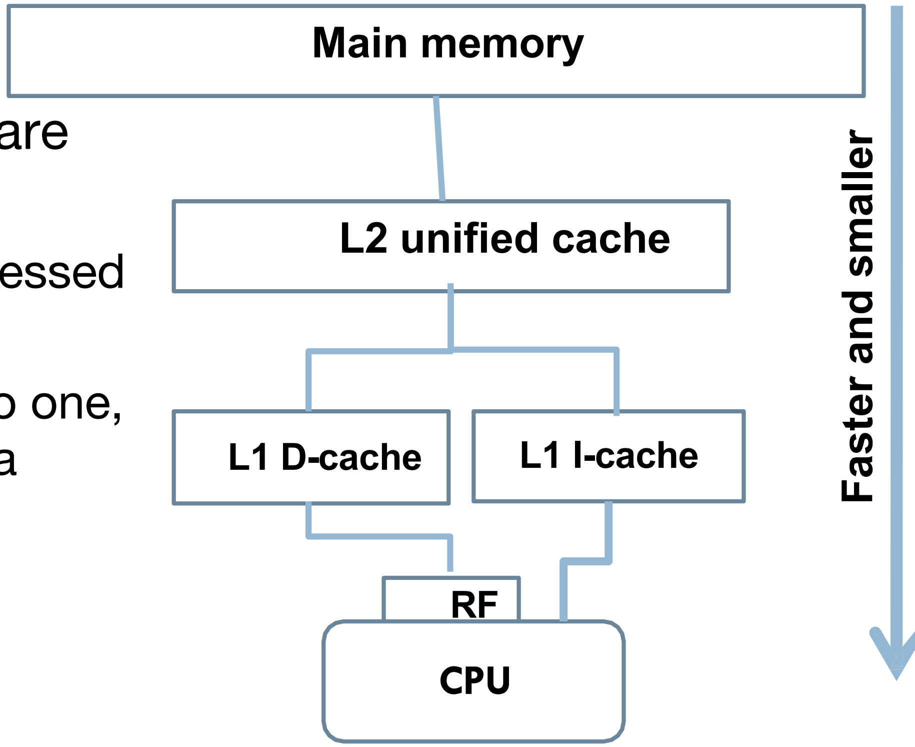
Loop Merge / Loop Fusion (3)

```
for (i=1; i<N; i++)
 A[ i ] = B[ i ];

for (i=1; i<N; i++)
 B[ i ] = A[ i-1 ];
```

$\Rightarrow$

```
for (i=1; i<N; i++){
 A[ i ] = B[ i ];
 B[ i ] = A[ i-1 ];
}
```

- Consider the case where the arrays are bigger than the L1 data cache, then
  - In the first case, both arrays are accessed from L2 and/or main memory twice
  - By merging the two loop kernels into one, the arrays are loaded once, and data locality is achieved



Loop Merge not always safe

- Is the following transformation correct?
  - NO – Data dependencies are not preserved

i=1: A[1] = B[1]
i=2: A[2] = B[2]
i=3: A[3] = B[3]
…

```
for (i=1; i<N; i++)
 A[ i ] = B[ i ];
```

```
for (i=1; i<N; i++)
 B[ i ] = A[ i+1 ];
```

i=1: B[1] = A[2]
i=2: B[2] = A[3]
i=3: B[3] = A[4]
…

$\Rightarrow$

~~*for (i=1; i<N; i++){*~~
~~*A[i] = B[i];*~~
~~*B[i] = A[i+1];*~~
~~*}*~~

i=1: A[1] = B[1]
B[1] = A[2]
i=2: A[2] = B[2]
B[2] = A[3]
i=3:
…

On the left,
we write in A [ ] and
then read from A[ ]

On the right,
we read from A [ ]
and then write to A[ ] (wrong)

Loop Merge not always safe

- The following transformation is not correct
  - Data dependencies are not preserved
- How to be sure
  - The top subscript must be larger or equal to the bottom subscript
  - Here, i >= i+1 is not true, thus loop merge is not safe

```
for (i=1; i<N; i++)
 A[ i ] = B[ i ];


for (i=1; i<N; i++)
 B[ i ] = A[ i+1 ];
```

Loop Distribution / Loop Fission (1)

- Loop Distribution is a transformation where a loop kernel is broken into multiple loop kernels over the same index range with each taking only a part of the original loop's body
- This transformation is not always safe
    - data dependencies must be preserved
    - The top subscript must be larger or equal to the bottom subscript

```
for (i=1; i<N; i++){
  A[ i ] = B[ i ];
  B[ i ] = A[ i-1 ];
}
```

⇨

```
for (i=1; i<N; i++)
  A[ i ] = B[ i ];

for (i=1; i<N; i++)
  B[ i ] = A[ i-1 ];
```

Loop Distribution / Loop Fission (2)

- Benefits:
  - May enable partial/full parallelization
  - This optimization is most efficient in multi/many core processors that can split a task into multiple tasks for each processor
  - May reduce register pressure
  - May improve data locality (cache misses)
  - May enable other loop transformations
- Drawbacks:
  - Increases the number of arithmetical instructions
  - May hurt data locality

```
for (i=1; i<N; i++){
  A[ i ] = B[ i ];
  B[ i ] = A[ i-1 ];
}
```

$\Rightarrow$

```
for (i=1; i<N; i++)
  A[ i ] = B[ i ];

for (i=1; i<N; i++)
  B[ i ] = A[ i-1 ];
```

Activity

- Should we apply loop merge or not?

```
// A
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      y[i] = y[i] + beta * A[i][j] * x[j];

for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      w[i] = w[i] + alpha * A[i][j] ;
```

```
//B
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
       y[i]+=A[i][j] * x[j]

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
       y2[i]+=A2[i][j] * x2[j]
```

Loop Reversal (1)

```
for (i=start; i<=end; i++)
        A[i] = … ;
```

$\rightarrow$

```
for (i=end; i>=start; i--)
        A[i] = … ;
```

OR

```
for (i=start; i<=end; i++)
  A[end - (i - start)] = … ;
```

- Loop reversal is a transformation that reverses the order of the iterations of a given loop
- It is not always safe
    - Remember, in the direction vector, the leftmost non '=' symbol has to be the same as before
    - Loop reversal has no effect on a loop-independent dependence.

Loop Reversal (2)

```
for (i=0; i<N; i++)
  for (j=0; j<P; j++)
    A[j][i] = A[j+1][i-1] + temp;
```

$\Rightarrow$

$d(i, j) = (1, -1)$

$D(i, j) = (\underset{\uparrow \text{ Dependence}}{<}, >)$

- Loop reversal cannot be applied to i loop
  - In this case D(i, j) = (>, >) and therefore the leftmost non '=' symbol changes, violating data dependencies
- Loop reversal can be applied to j loop though
  - In this case D(i, j) = (<, <) and therefore the leftmost non '=' symbol does not change

Loop Reversal (3)

- Main Benefits
  - Increase parallelism
    - In loop nests, loop reversal is used to uncover parallelism and move it to the outermost iterator possible
  - Enable other transformations

Loop Reversal: example 1

```
for (i=0; i<N; i++)
  for (j=0; j<P; j++)
    A[j][i] = A[j+1][i-1] + temp;
```

Dependence

$$\underset{\uparrow\ \text{Dependence}}{}D(i, j) = (<, >)$$

- Problem: The array is accessed column-wise; this gives
  - Low performance
  - High energy consumption
- Potential Solution: Apply loop interchange
  - However, loop interchange gives D(j, i) = (>, <), violating data dependencies
- Solution: apply loop reversal to the j loop which gives D(i, j) = (<, <)
  - Then, loop interchange is valid as it gives D(j, i) = (<, <)

Loop Reversal: example 1

```
for (i=0; i<N; i++)
  for (j=0; j<P; j++)
    A[j][i] = A[j+1][i-1] + temp;
```

Dependence

D(i, j) = (<, >)

column-wise array accesses (inefficient)

loop reversal

```
for (i=0; i<N; i++)
  for (j=P-1; j>=0; j--)
    A[j][i] = A[j+1][i-1] + temp;
```

Dependence

D(i, j) = (<, <)

loop interchange

Dependence

D(j, i) = (<, <)

```
for (j=P-1; j>=0; j--)
  for (i=0; i<N; i++)
    A[j][i] = A[j+1][i-1] + temp;
```

row-wise array accesses (efficient)

Loop Reversal: example 2

```
for (i=0; i<=N; i++)
   B[i] = A[i] + …;

for (i=0; i<=N; i++)
   C[i] = B[N-i] - …;
```

Loop merge not possible
i >= N - i, not true

Apply loop reversal to the 2nd loop kernel

```
for (i=0; i<=N; i++)
   B[i] = A[i] + …;

for (i=0; i<=N; i++)
   C[N-i] = B[N-(N-i)] - …;
```

Loop merge is now possible
as i >= i

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

Loop Peeling

- Separate special cases at either end
    - Always safe

```
for (i=0; i<100; i++)
  A[i] = A[0] + B[i];
```

$\Rightarrow$

```
A[0] = A[0] + B[0];

for (i=1; i<100; i++)
 A[i] = A[0] + B[i];
```

Loop carried dependence - The compiler cannot parallelize it

No dependence - The compiler can parallelize it or vectorise it

Loop Peeling: example

```
for (i=2; i<=N; i++)
  B[i] = A[i] + temp;

for (i=3; i<=N; i++)
  C[i] = A[i] + D[i];
```

Loop merge not possible

Apply loop peeling to the 1st loop kernel

```
If (N>=2)
  B[2] = A[2] + temp;

for (i=3; i<=N; i++)
  B[i] = A[i] + temp;

for (i=3; i<=N; i++)
  C[i] = A[i] + D[i];
```

Loop merge is now possible

```
If (N>=2)
  B[2] = A[2] + temp;

for (i=3; i<N; i++) {
  B[i] = A[i] + temp;
  C[i] = A[i] + D[i];
}
```

Loop Bump

```
for (i=start; i<end; i++)          for (i=start + N; i<end + N; i++)
       A[i] = …              ⇨              A[i - N] = …
```

- Changes the loop bounds
- It is always safe

- Benefits:
  - It can enable other transformations
  - It can increase parallelism

Loop Bump: example 1

```
for (i=2; i<N; i++)
  B[i] = A[i] + …;

for (i=0; i<N-2; i++)
  C[i] = B[i+2] + …;
```

Loop merge not possible i >= i+2, not true

Apply loop bump to the 2nd loop kernel

```
for (i=2; i<N; i++)
  B[i] = A[i] + …;

for (i=0+2; i<N-2+2; i++)
  C[i-2] = B[i+2-2] + …;
```

Loop merge is now possible as i >= i

```
for (i=2; i<N; i++) {
  B[i] = A[i] + …;
  C[i-2] = B[i] + …;
}
```

Array copying transformation (1)

- Copies the array's elements into a new array before computation
  - The new array's elements will be written in consecutive main memory locations
- Always safe but incurs high cost

```
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
  for (k=0;k!=N;k++)
   C[i][j]+=A[i][k] * B[k][j];
```

Vectorization is extremely pure

```
//array copying
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
  B_transpose[i][j]=B[j][i];

for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
 for (k=0;k!=N;k++)
  C[i][j]+=A[i][k] * B_transpose[j][k];
```

Vectorization can be applied effectively

Array copying transformation (2)

- When should we apply array copying?
  - When the number of cache misses is high and multi-dimensional arrays exist
  - In vectorization, as vectorization needs consecutive memory locations

```
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
  for (k=0;k!=N;k++)
   C[i][j]+=A[i][k] * B[k][j];
```

$\Rightarrow$

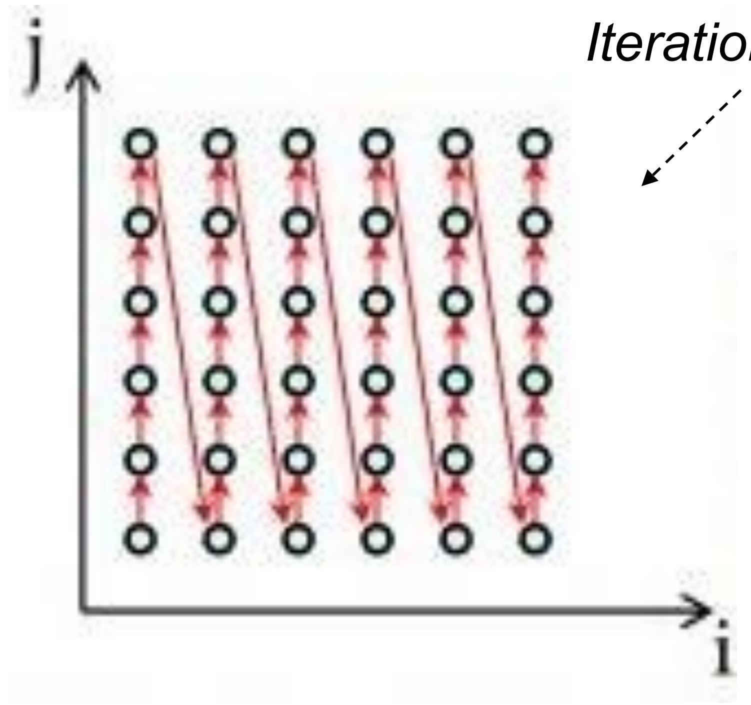
```
//array copying
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
  B_transpose[i][j]=B[j][i];

for (i=0;i!=N;i++)
 for (j=0;j!=N;j++)
 for (k=0;k!=N;k++)
  C[i][j]+=A[i][k] * B_transpose[j][k];
```

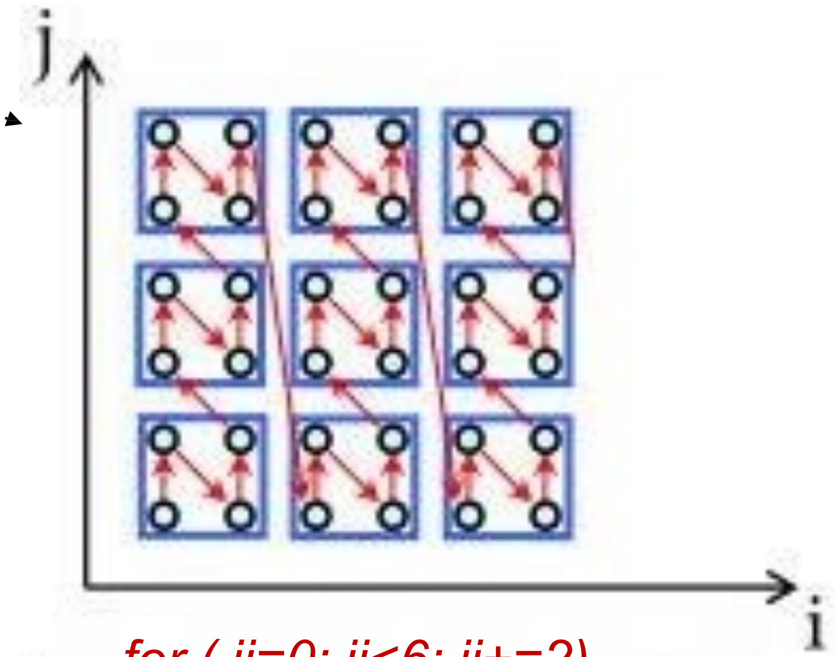
Software Prefetching

- This is an advanced topic and it is not going to be studied
- The SSE/AVX x86-64 intrinsics include prefetch instructions.
- Example of a software prefetch instruction:
  _mm_prefetch(&C[i][j], _MM_HINT_T0);
    - The instruction above pre-fetches the cache line containing C[i][j] from DDR.
    - No value is written back to a register and we do not have to wait for the instruction to complete.
    - The cache line is loaded in the background.

Loop Tiling / blocking (1)

Iteration space





```
for ( i=0; i<6; i++)
  for ( j=0; j<6; j++)
  S1[i][j]=…;
```

```
for ( ii=0; ii<6; ii+=2)
  for ( jj=0; jj<6; jj+=2)

    for ( i=ii; i<ii+2; i++)
     for ( j=jj; j<jj+2; j++)
     S1[i][j]=…;
```

Loop Tiling / blocking (2)

- Loop tiling partitions a loop's iteration space into smaller chunks or blocks, to help data remain in the cache (data reuse)
- The partitioning of loop iteration space leads to the partitioning of large arrays into smaller blocks (tiles), thus fitting accessed array elements into the cache, enhancing cache reuse, and reducing cache misses
- Loop tiling can be applied to each iterator multiple times, e.g., it is applied to the j and I iterators in the previous example
- Loop tiling is one of the most performance-critical transformations for data-dominant algorithms

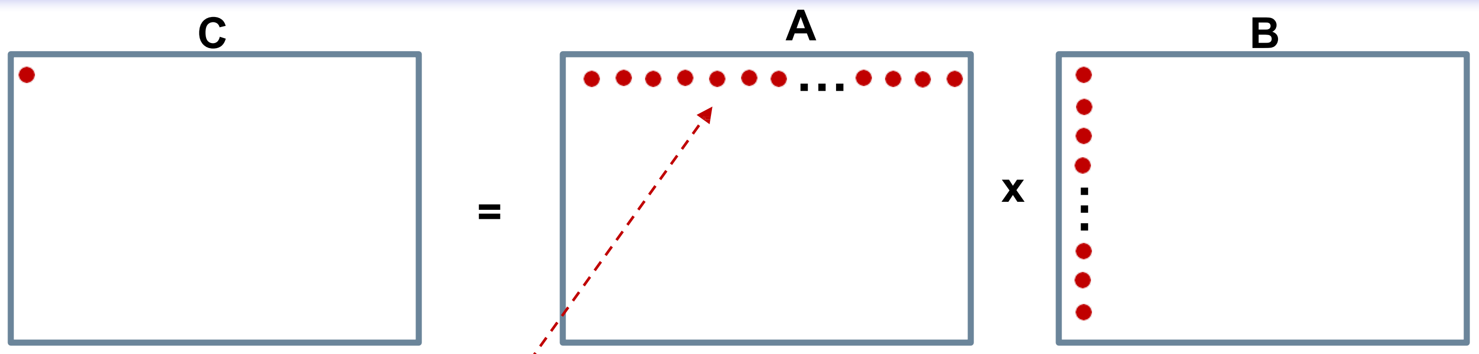
Loop Tiling / blocking (3)

- In data dominant algorithms, loop tiling is applied to exploit data locality in each memory, including register file
- Register blocking can be considered as loop tiling for the register file memory
- By applying loop tiling to L_i cache memory, the number of L_i cache misses is reduced
- The number of Li cache misses is equal to the number of L_{i+1} accesses

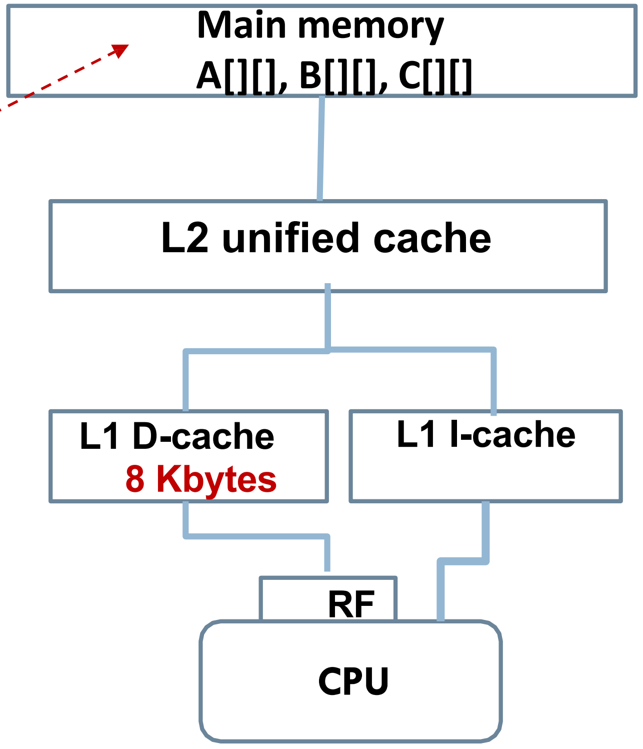
Loop Tiling / blocking (4)

- Loop tiling reduces the number of cache misses
  - This doesn’t always entail performance improvement
    - Performance depends on other parameters too, e.g., the number of instructions
- Key problems:
  - Selection of the tile size
  - Loops/iterators to be applied to
  - How many levels of tiling to apply (multi-level cache hierarchy)
- Pros: may increase locality (reduce cache misses)
- Cons: increases the number of instructions (adds extra loop)

Loop tiling: MMM Problem



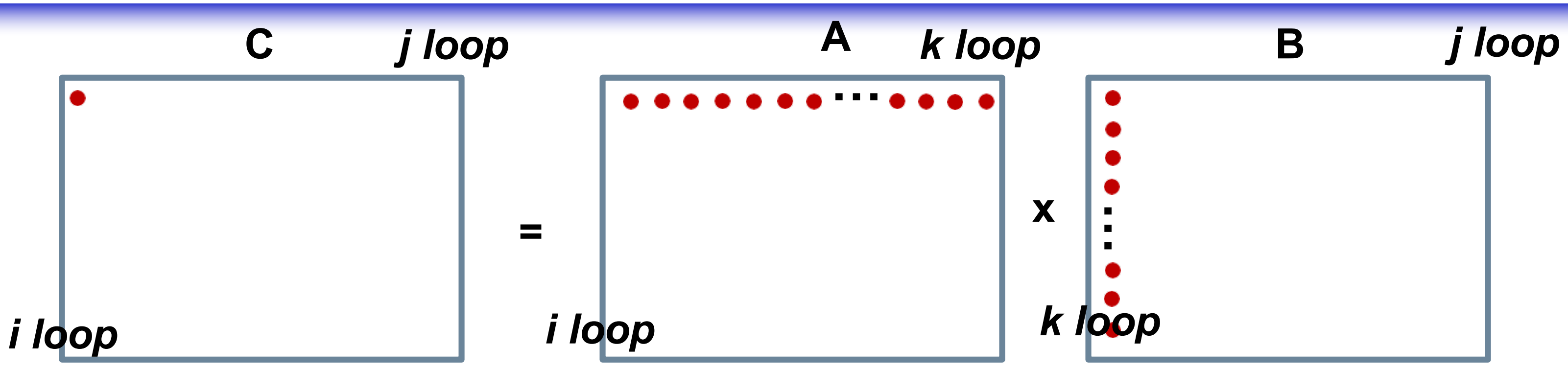
The size of each row of A is 8 kbytes



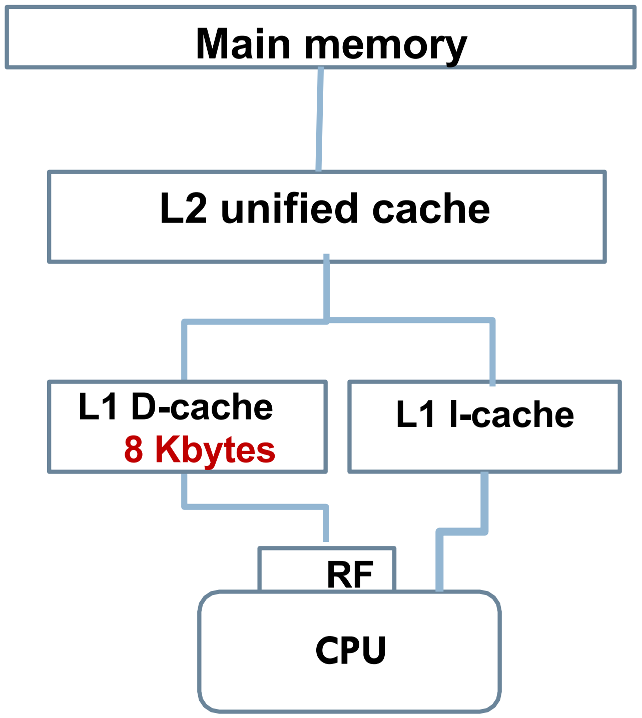
float C[2048][2048], A[2048][2048], B[2048][2048];

```
for (i=0; i<2048; i++)
  for (j=0; j<2048; j++)
   for (k=0; k<2048; k++)
     C[i][j] += A[i][k] * B[k][j];
```

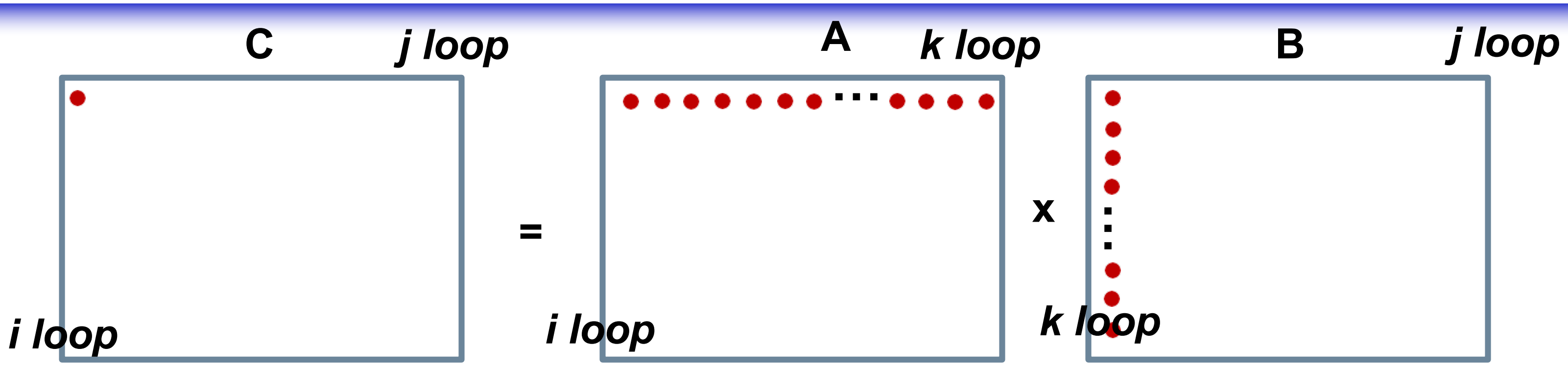
Loop tiling: MMM Problem



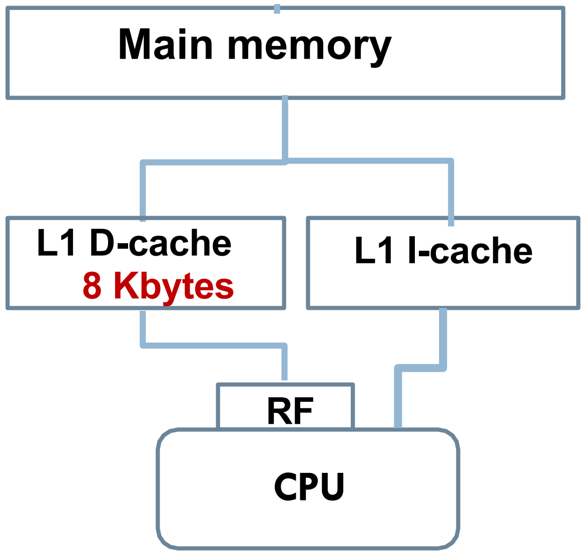
- Each row of A is multiplied by all the columns of B, thus:
  - Each row of A is loaded from memory 2048 times
  - If the row of A cannot remain in L1D, it will be loaded 2048 times from L2
  - If the row of A cannot remain in L2, it will be loaded 2048 times from the main memory
- The whole B array is multiplied by each row of A, thus:
  - B array is loaded 2048 times from memory
  - If B cannot remain in L1D, it will be loaded 2048 times from L2
  - If B cannot remain in L2, it will be loaded 2048 times from main memory



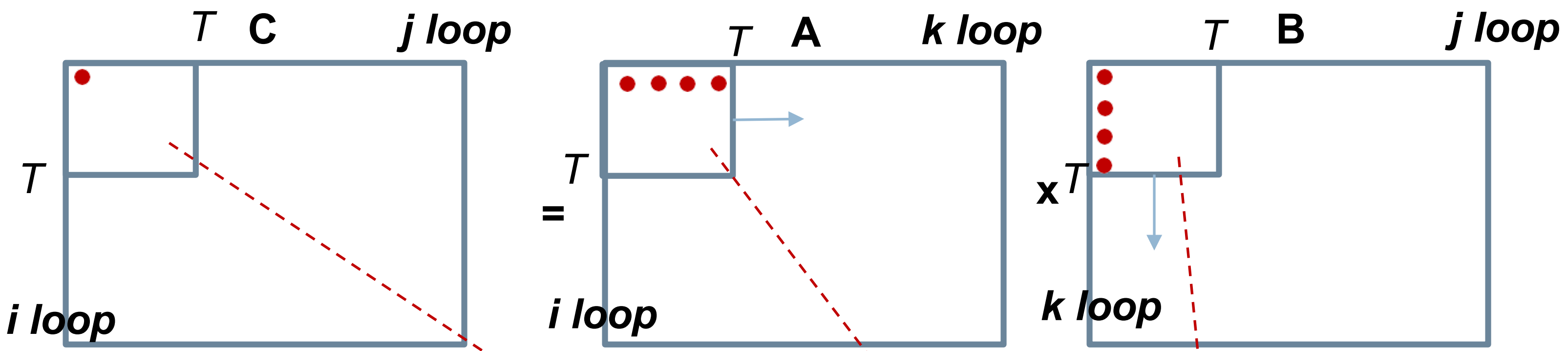
Loop tiling: MMM Problem



- Consider a single level of cache.
- A is loaded 2048 times from main memory, 2048^3 loads
- B is loaded 2048 times from main memory, 2048^3 loads
- C is stored just once, 2048^2 stores



Loop tiling & MMM – 1 level of cache (1)



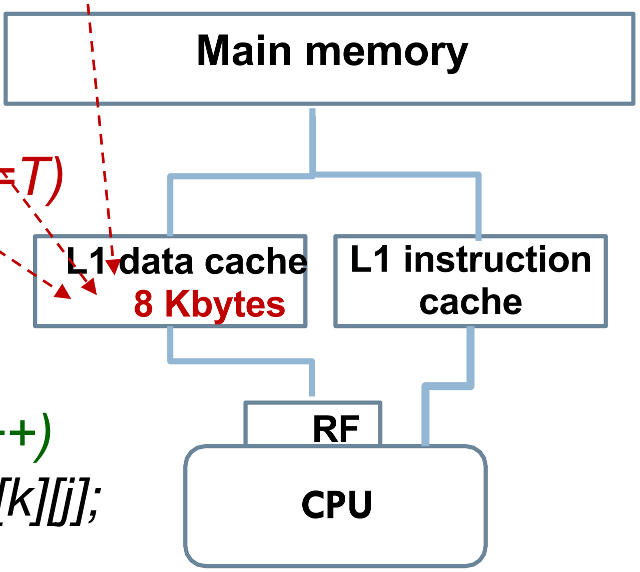
These loops specify which tiles to multiply

```
for (i=0; i<2048; i++)
  for (j=0; j<2048; j++)
    for (k=0; k<2048; k++)
      C[i][j] += A[i][k] * B[k][j];
```

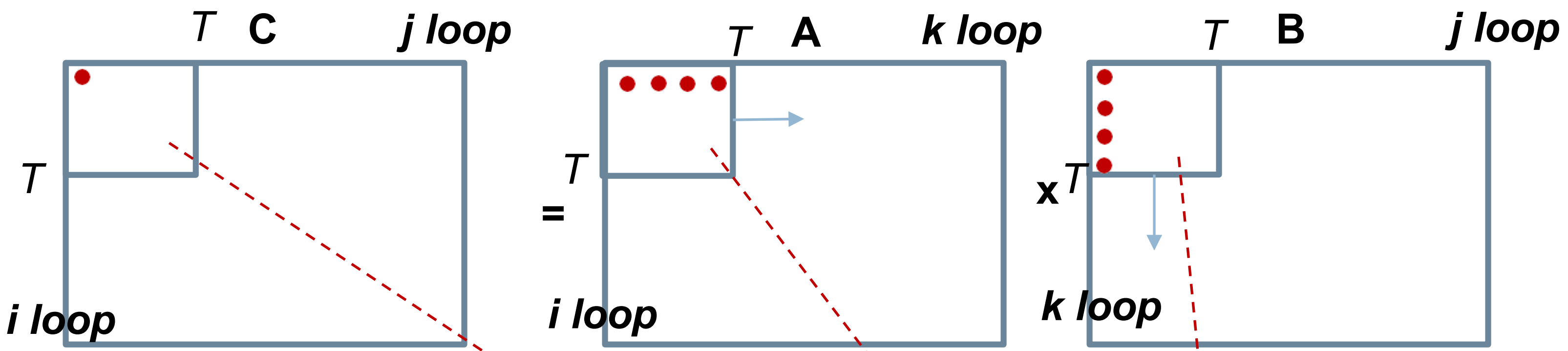
```
for (ii=0; ii<2048; ii+=T)
  for (jj=0; jj<2048; jj+=T)
    for (kk=0; kk<2048; kk+=T)
```

```
for (i=ii; i<ii+T; i++)
  for (j=jj; j<jj+T; j++)
    for (k=0; k<kk+T; k++)
      C[i][j] += A[i][k] * B[k][j];
```

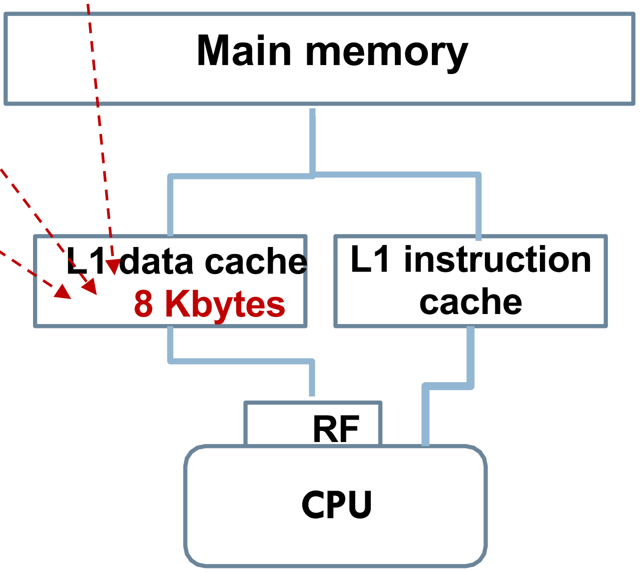
These loops specify which elements inside the tile to multiply



Loop tiling & MMM – 1 level of cache (2)

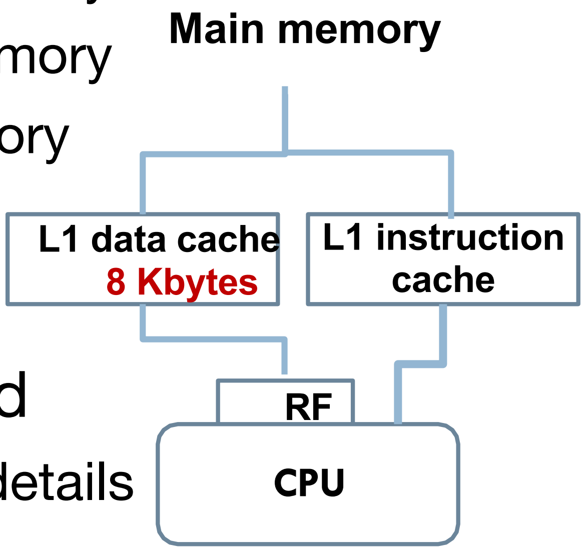


- The matrices are partitioned into smaller sub-matrices (TxT)
- Instead of multiplying A[][] by B[][], their tiles are multiplied
  - The tiles are small enough to fit in the cache
  - A is loaded 2048/T times from the main memory
  - B is loaded 2048/T times from the main memory
  - C is loaded and stored 2048/T times from/to the main memory



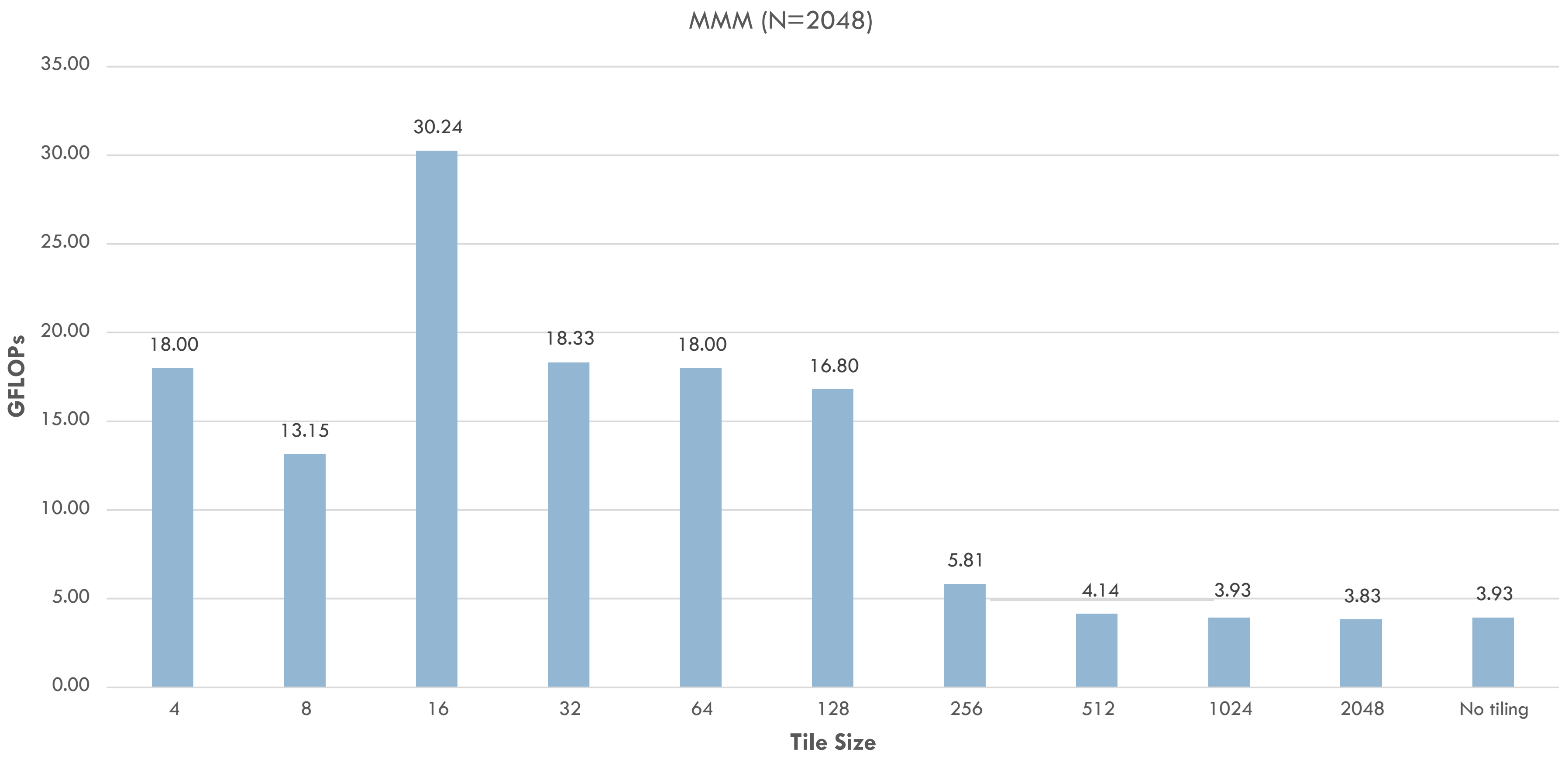
Loop tiling & MMM – 1 level of cache (3)

- Before applying loop tiling
  - A: 2048 x (2048x2048) loads from main memory
  - B: 2048 x (2048x2048) loads from main memory
  - C: 1 x (2048x2048) stores to main memory
  - In total, $2*2048^3 + 2048^2$ main memory accesses
- After applying loop tiling
  - A: 2048/T x (2048x2048) loads from main memory
  - B: 2048/T x (2048x2048) loads from main memory
  - C: 2048/T x (2048x2048) stores to main memory
  - In total, $3*2048^3/T$ main memory accesses
- By increasing T, performance is increased
  - However, T is bound to the cache hardware details



MMM – Loop Tiling Performance Evaluation (1)

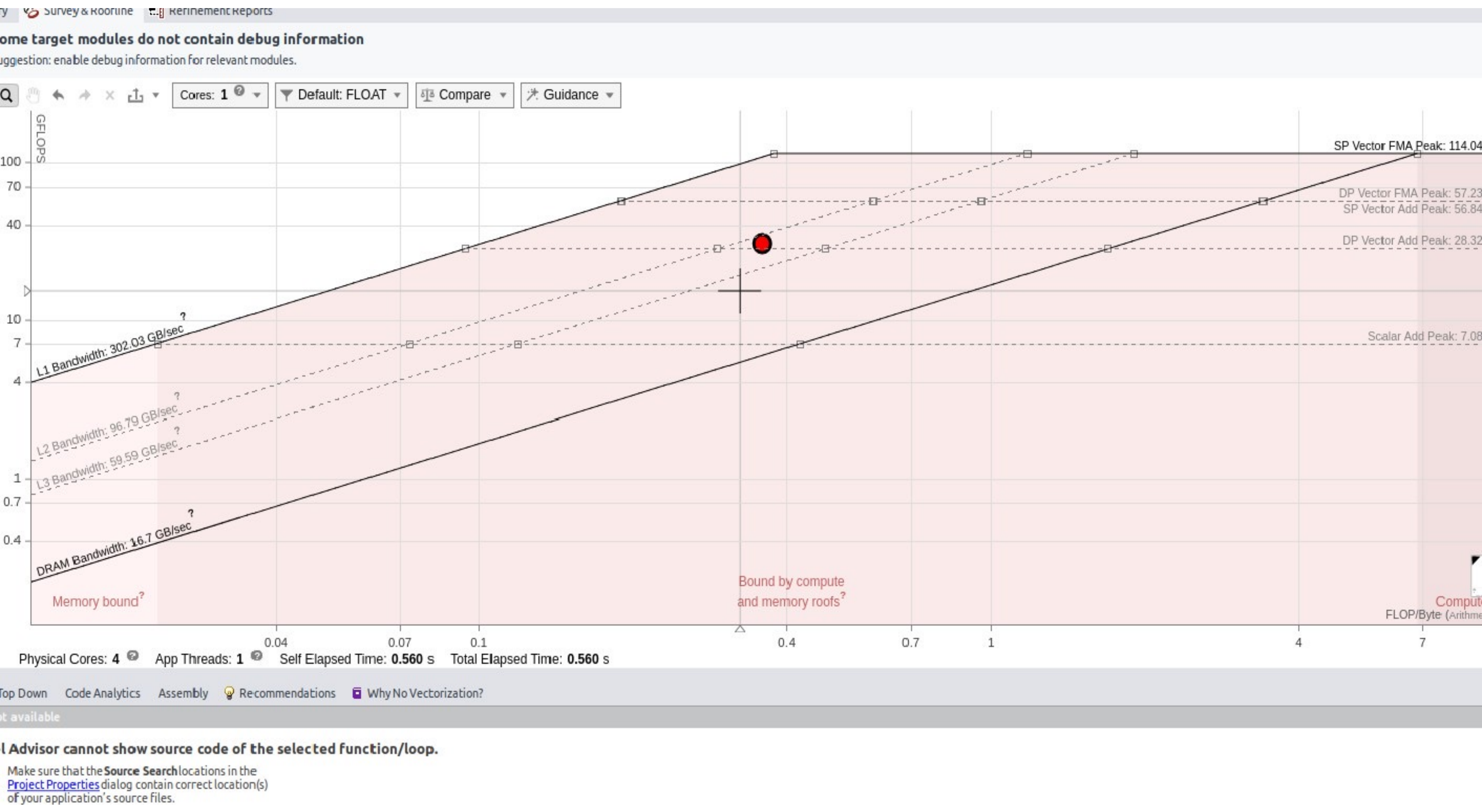
- Square Tile sizes are used Ti=Tj=Tk=T



Tile Size	GFLOPs
4	18.00
8	13.15
16	30.24
32	18.33
64	18.00
128	16.80
256	5.81
512	4.14
1024	3.93
2048	3.83
No tiling	3.93

MMM – Loop Tiling Performance Evaluation (2)

- Roofline analysis for T=16



y Survey & Roofline Refinement Reports

ome target modules do not contain debug information

uggestion: enable debug information for relevant modules.

Cores: 1 | Default: FLOAT | Compare | Guidance

GFLOPS
SP Vector FMA Peak: 114.04
DP Vector FMA Peak: 57.23
SP Vector Add Peak: 56.84
DP Vector Add Peak: 28.32
Scalar Add Peak: 7.08
L1 Bandwidth: 302.03 GB/sec
L2 Bandwidth: 96.79 GB/sec
L3 Bandwidth: 59.59 GB/sec
DRAM Bandwidth: 16.7 GB/sec
Memory bound
Bound by compute and memory roofs
Comput
FLOP/Byte (Arithme
100
70
40
10
7
4
1
0.7
0.4
0.04
0.07
0.1
0.4
0.7
1
4
7

Physical Cores: 4 App Threads: 1 Self Elapsed Time: **0.560** s Total Elapsed Time: **0.560** s

op Down Code Analytics Assembly Recommendations Why No Vectorization?

t available

l Advisor cannot show source code of the selected function/loop.

Make sure that the **Source Search** locations in the Project Properties dialog contain correct location(s) of your application's source files.

References

- Optimizing compilers for modern architectures: a dependence-based approach
  - https://dl.acm.org/doi/10.5555/502981
- Options That Control Optimization
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- V. Kelefouras, Compilers for Embedded Systems
  - https://eclass.upatras.gr/modules/document/?course=EE738