Compilers for Embedded Systems

Integrated Systems of Hardware and Software

Lecture 2-3

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk
Website: https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Outline

- Code optimization
  - key problems
- Some **basic/simple code optimizations**/transformations and manually applied techniques:
  - Use the available Compiler Options
  - Reduce complex operations
  - Loop based strength reduction
  - Dead code elimination
  - Common subexpression elimination
  - Use the appropriate precision
  - Choose a better algorithm
  - Loop invariant code motion
  - Use table lookups
  - Function Inline
  - Loop unswitching
  - Loop unroll
  - Scalar replacement
- More advanced code transformations
  - Loop merge/distribution, loop tiling, register blocking, array copying, etc

Optimize What?

- Optimization in terms of
  - Execution time
  - Energy consumption
  - Space (Memory size)
    - Reduce code size
    - Reduce data size

How to optimize ?

- **Optimizing the easy way**
  - Use a faster programing language, e.g., C instead of Python
  - Use a better compiler
  - Manually enable specific compiler's options

  - Normally, the optimization gain is limited
  - No expertise is needed

- **Optimizing the hard way**
  - use a profiler to identify performance bottlenecks, normally loop kernels
  - Manually apply code optimizations
  - Re-write parts of the code from scratch

  - Needs expertise
  - Optimization gain is high

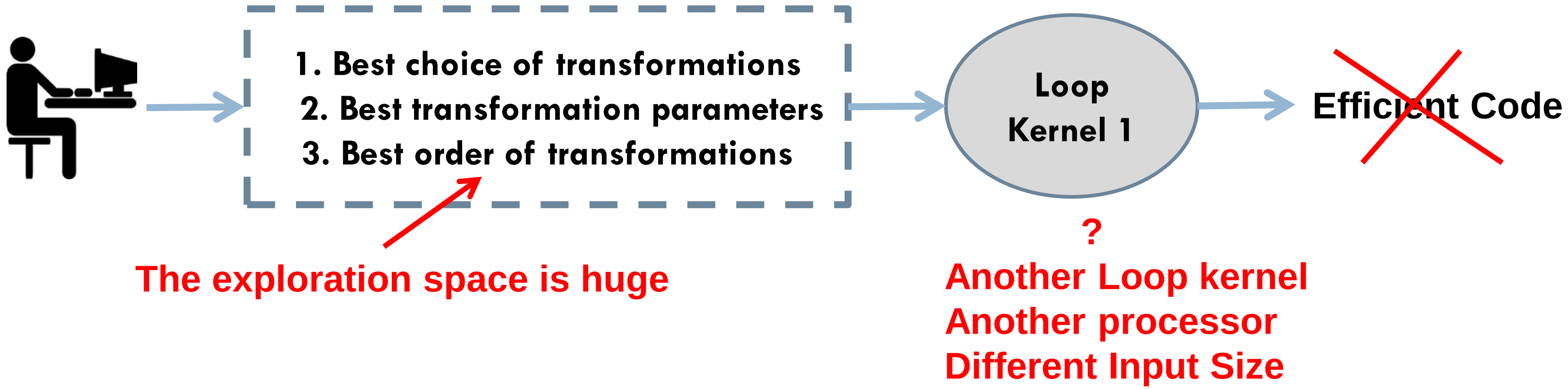
Introduction

- Loops represent the most computationally intensive part of a program.
- Improvements to loops will produce the most significant effect

- Loop optimization
  - **90% / 10% rule**
  - Normally, "90% of a program's execution time is spent in executing 10% of the code"
    - larger payoff to optimize the code within a loop

Which Compiler Options to use and when?

- Compilers offer a large number of transformation/optimization options
- This is a complex longstanding and unsolved problem for decades
  - Which compiler optimization/transformation to use?
  - Which parameters to use? Several optimizations include different parameters
  - In which order to apply them?



Optimizing SW - problem (1)

- **The key to optimizing software is the correct**
  - **Choice**
  - **Order**
  - **Parameters**

  of code optimizations

- One of the most used compilers is gcc
- You can find its options here https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html

- **But why optimizing SW is so hard?**
- Normally, the efficient optimizations for a specific code are not efficient for
  - another code
  - another processor
  - different hardware architecture details, e.g., cache line size
  - or even for a different input size

Optimizing SW – problem (2)

- Why compilers can't find the optimum choice, order and parameters of optimizations?
  1. Compilers are not smart enough to take into account
     - ✓ most of the hardware architecture details (e.g., cache size and associativity)
     - ✓ custom algorithm characteristics (e.g., data access patterns, data reuse, algorithm symmetries)
     - Even experienced programmers
       - Do not understand how software runs on the target hardware
       - Treat threads as black boxes
       - Blindly apply loop transformations

     - ➢ Peak performance demands going low level
       - Understand the hardware, compilers, ISA

Optimizing SW – problem (3)

- Why compilers can't find the optimum choice, order and parameters of optimizations?
  2. The compilation sub-problems depend on each other which makes the problem extremely difficult
     - ✓ these dependencies require that all the problems should be optimized together as one problem and not separately

     - Toward this much research has been done
       - Iterative compilation techniques
       - Methodologies that simultaneously optimize only two problems
       - Searching and empirical methods
       - Heuristics
       - But …
         - They are partially applicable
         - They cannot give the best solution

Optimizing SW – problem (4)

- Why compilers can't find the optimum choice, order and parameters of optimizations?
  3. The exploration space (all different implementations/binaries) is so big that it cannot be searched; researchers try to decrease the space by using
     - machine learning compilation techniques
     - genetic algorithms
     - statistical techniques
     - exploration prediction models focusing on beneficial areas of optimization search space

     - however, the search space is still so big that it cannot be searched, even by using modern supercomputers

Basic and Simple techniques that will improve your code

- Use the available Compiler Options
- Reduce complex operations
- Loop based strength reduction
- Dead code elimination
- Common subexpression elimination
- Use the appropriate precision
- Choose a better algorithm
- Loop invariant code motion
- Use table lookups
- Function Inline
- Loop unswitching
- Loop unroll
- Scalar replacement

Use the available compiler options

- **The most used optimization flags/options are the following**
  - **' –O0'** - Disables all optimizations, but the compilation time is very low
  - **' –O1'** - Enables basic optimizations
  - **' –O2'** - Enables more optimizations
  - **' –O3'** - turns on all optimizations specified by -O2 and enables more aggressive loop transformations such as register blocking, loop interchange etc
  - **'-Ofast' option - be careful:** it is not always safe for codes using floating point arithmetic
  - **'Osize' option –** Optimizes for code size

- In VS, go to Project tab -> properties -> C/C++ -> Optimization

- *gcc options can be found here:*

https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html

Loop unroll transformation (1)

- Creates additional copies of loop body
- Always safe

```
//C-code1
for (i=0; i < 100; i++)
  A[i] = B[i];
```

⇨

```
//C-code2
for (i=0; i < 100; i+=4) {
  A[i] = B[i];
  A[i+1] = B[i+1];
  A[i+2] = B[i+2];
  A[i+3] = B[i+3];
}
```

Pros:

✓ Reduces the number of instructions

✓ Increase instruction parallelism

Cons:

– Increases code size

– Increases register pressure

Loop unroll transformation (2)


```
// C code1
for (i=0; i<100; i++) {
...
}
// assembly code1
loop_i ...
...                              A[i] = B[i];
inc i          // increment i
cmp i, 100     // compare i to 100
jl loop_i      // jump if  i lower to 100
```

100 times

```
// C code2
for (i=0; i<100; i+=4) {
...
}
// assembly code2
loop_i ...
...              A[i] = B[i];
...              A[i+1] = B[i+1];
...              A[i+2] = B[i+2];
...              A[i+3] = B[i+3];
inc i          // increment i
cmp i, 100     // compare i to 100
jl loop_i      // jump if lower
```

100/4 times

✓ The number of arithmetical instructions is reduced

1. Less add instructions for i, i.e., i=i+4 instead of i=i+1
2. Less compare instructions, i.e., i==100 ?
3. Less jump instructions

Scalar replacement transformation

- Converts array reference to scalar reference
- Most compilers will do this for you automatically by specifying '-O2' option
- Always safe

```
//Code-1
for (i=0; i < 100; i++){
  A[i] = … + B[i];
  C[i] = … + B[i];
  D[i] = … + B[i];
}
```

→

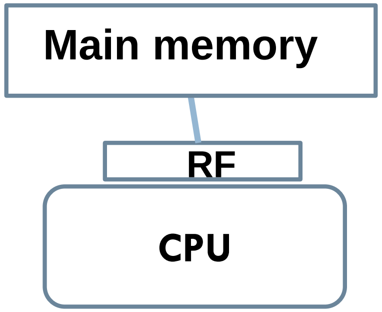
```
//Code-2
for (i=0; i < 100; i++){
  t=B[i];
  A[i] = … + t;
  C[i] = … + t;
  D[i] = … + t;
}
```

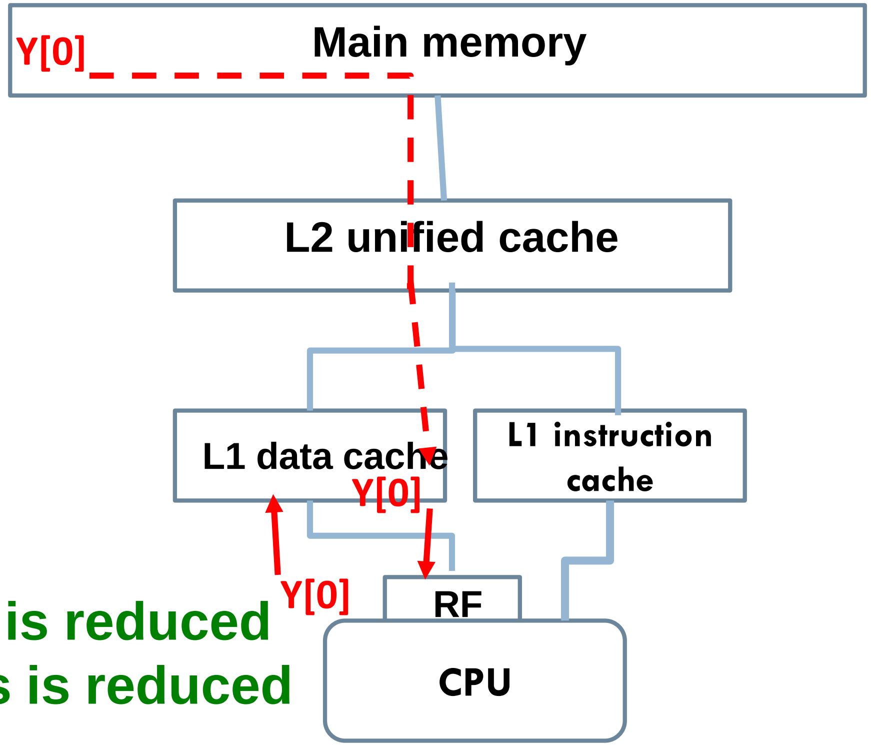
✓ Reduces the number of L/S instructions
✓ Reduces the number of memory accesses

Scalar Replacement Transformation example (1)


```
// C-code1
for (i=0; i<300; i++)
  for (j=0; j<300; j++)
    Y[i] += A[i][j] * X[j];
```

```
// C-code2
for (i=0; i<300; i++) {
   tmp=Y[i];
  for (j=0; j<300; j++) {
   tmp+= A[i][j] * X[j];
   }
  Y[i]=tmp;
}
```





- Y[i] is not affected by j loop
- For every j, Y[i] is redundantly loaded/stored from/to memory
- A load/store instruction needs 1-3 CPU cycles

✓ the number of L/S instructions is reduced

✓ the number of L1 data accesses is reduced

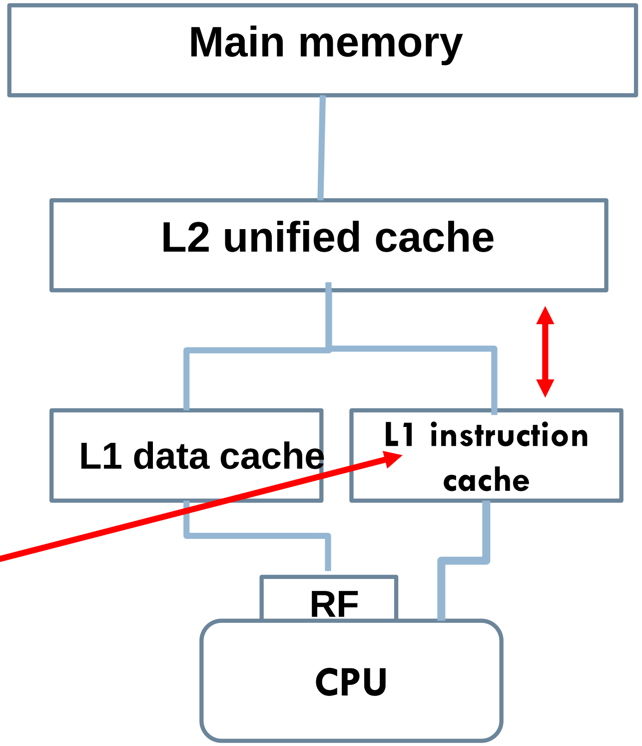
You have learned that the largest the loop unroll factor, the largest the gain in instructions, but is it always efficient?

- When code2 is faster than code1?
  - a) Always
  - b) Never
  - c) It depends on the hardware architecture
  - d) It is impossible to know

When the code2 size becomes larger than L1 instruction cache size, code2 is no longer efficient

```
//code1
N=1000000;
for (i=0; i < N; i++)
  A[i] = B[i];
```

```
//code2
N=1000000;
for (i=0; i < N; i+=10000) {
  A[i] = B[i];
  A[i+1] = B[i+1];
  A[i+2] = B[i+2];
  A[i+3] = B[i+3];
        ...
  A[i+9999] = B[i+9999];
}
```



Use as less complex operations as possible (1)

- **Division is expensive**
  - On most CPUs the division operator is significantly more expensive (i.e. takes many more clock cycles) than all other operators. When possible, refactor your code to not use division.
  - Use multiplication instead
  - For example, change ***‘ / 5.0 ‘*** to ***‘ * 0.2 ‘***

- Use shift operations instead of multiplication and division
  - Only for multiplications and division with powers of 2
  - Compilers will do that for you though

Use as less complex operations as possible (2)

- **Functions such as pow(), sqrt()** etc are expensive, so avoid them when possible
  - E.g., avoid calling functions such as strlen() all the time, call it once (x=strlen()) and then x++ or x-- when you add or remove a character.
- **Avoid Standard Library Functions**
  - Many of them are expensive only because they try to handle all possible cases
  - Think of writing your own simplified version of a function, if possible, tailored to your application
  - E.g., pow(a, b) function where b is an integer and b=[1,10]

Strength Reduction (1)

- Strength reduction is the replacement of an expression by a different expression that yields the same value but is cheaper to compute
- Most compilers will do this for you automatically by specifying '-O1' option

```
do i = 1, n
  a[i] = a[i] + c*i
end do
```

(a) original loop

- Normally, addition needs less CPU cycles than multiplication
- In each iteration c is added to T

```
T = c
do i = 1, n
  a[i] = a[i] + T
  T = T + c
end do
```

(b) after strength reduction

Loop-Invariant Code Motion

- Any part of a computation that does not depend on the loop variable and which is not subject to side effects can be moved out of the loop entirely
- **Most compilers will do this for you automatically by specifying '-O1' option**

```
do i = 1,n
  a[i] = a[i] + sqrt(x)
end do
```

(a) original loop

```
if (n > 0) C = sqrt(x)
do i = 1,n
  a[i] = a[i] + C
end do
```

(b) after code motion

- The value of sqrt(x) is not affected by the loop
- Therefore, its value is computed just once, outside of the loop
- If n<1, the loop is not executed and therefore C must not be assigned with the sqrt(x) value

Function Inline

- Replace a function call with the body of the function
- It can be applied in many different ways
  - Either manually or automatically
  - ‘-O1’ applies function inline
  - In C, a good option is to use macros instead (if possible)
- **Pros :-**

  1. It speeds up your program by avoiding function calling overhead
  2. It saves the overhead of pushing/poping on the stack
  3. It saves overhead of return call from a function
  4. It increases locality of reference by utilizing instruction cache
- **Cons**
  - The main drawback is that it increases the code size

Loop Unswitching

- A loop containing a loop-invariant IF statement can be transformed into an IF statement containing two loops
- After unswitching, the IF expression is only executed once, thus improving run-time performance
- After unswitching, the loop body does not contain an IF condition and therefore it can be better optimized by the compiler
- **Most compilers will do this for you automatically by specifying '-O3' option**

```
for (i = 0; i < N; i++) {
  if (x<0)
    a[i] = 0;
  else
    b[i] = 0;
}
```

```
if (x<0)
  for (i = 0; i < N; i++) {
    a[i] = 0;
  }
else
  for (i = 0; i < N; i++) {
    b[i] = 0;
  }
```

Register Blocking
also known as Loop unroll and jam (1)

- Register blocking is primarily intended to
  - **increase register exploitation (data reuse)**
  - **reduce the number of L/S instructions**
  - **reduce the number of memory accesses**

- Register blocking involves two transformations
  - Loop unroll
  - Scalar replacement
- Register blocking is included in '-O3' optimization option
  - In gcc you must enable this option : -floop-unroll-and-jam
  - However, an experienced developer can achieve better results

Register Blocking
also known as Loop unroll and jam (2)

- **The steps are:**
  1. One or more loops (not the innermost) **are partially unrolled** and as a consequence common array references are exposed in the loop body (data reuse)
  2. Then, the array references are **replaced by variables (scalar replacement transformation)** and thus the number of L/S instructions is reduced

```
// C code of MMM
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];
```

Step1

```
// C code of MMM
for (i=0; i<N; i++)
  for (j=0; j<N; j+=2) {
    for (k=0; k<N; k++) {
C[i][j]    += A[i][k] * B[k][j];
C[i][j+1] += A[i][k] * B[k][j+1];
} }
```

C[i][j] does not depend on the innermost loop Get it out and use register

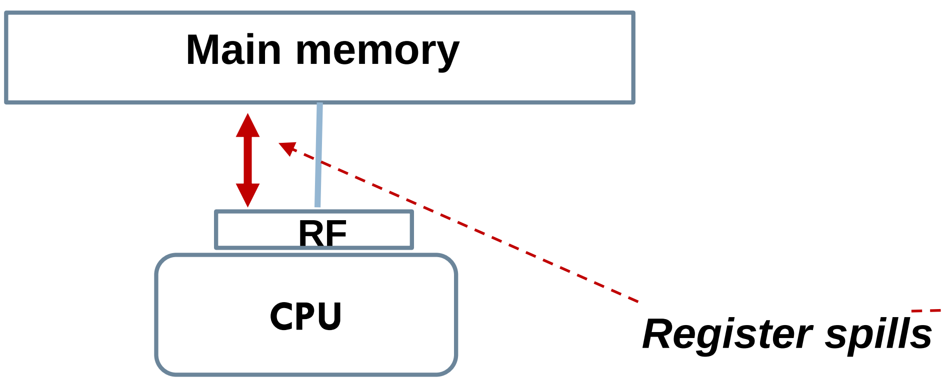
Common reference, use a register

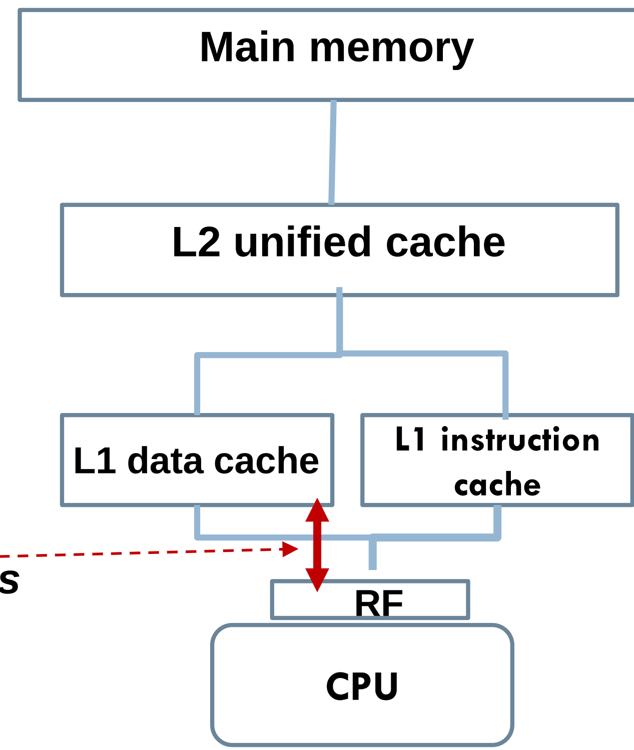
Step2

```
// C code of MMM
for (i=0; i<N; i++)
  for (j=0; j<N; j+=2) {
  c0=C[i][j];
  c1=C[i][j+1];

   for (k=0; k<N; k++) {
   a0=A[i][k];
   c0 += a0 * B[k][j];
   c1 += a0 * B[k][j+1];
        }
C[i][j]=c0;
C[i][j+1]=c1;
}
```

Register Blocking also known as Loop unroll and jam (3)

- Key Point:
  - **The number of the variables in the loop kernel must be lower or equal to the number of the available registers**
  - Otherwise, some of the variables cannot remain in the registers and they are loaded many times from L1 data cache (dL1), degrading performance
    - This is also known as **register spills**





Register Blocking (4)
Another example

- *A[i][k] is loaded and then used 4 times (data reuse)*
- *Therefore, A[i][k] is loaded 4 times less than before*
- *Every load from dL1 costs 1-3 cycles*

- *In the first case, C[i][j] is loaded/stored N^3 times, i.e., (N times for k loop x N times for j x N times for i loop)*
- *Now, registers are used to hold the intermediate results and therefore they are loaded/stored from/to registers not dL1*
- *Using registers is much faster*
- *Now, C array references are outside k loop and therefore it is loaded/stored N^2 times only*

```
// C code of MMM
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];
```

Step1

```
// C code of MMM
for (i=0; i<N; i++)
 for (j=0; j<N; j+=4) {
  for (k=0; k<N; k++) {
C[i][j] += A[i][k] * B[k][j];
C[i][j+1] += A[i][k] * B[k][j+1];
C[i][j+2] += A[i][k] * B[k][j+2];
C[i][j+3] += A[i][k] * B[k][j+3];
          } }
```

Step2

```
// C code of MMM
for (i=0; i<N; i++)
 for (j=0; j<N; j+=4) {
 c0=C[i][j];
 c1=C[i][j+1];
 c2=C[i][j+2];
 c3=C[i][j+3];

  for (k=0; k<N; k++) {
  a0=A[i][k];
  c0 += a0 * B[k][j];
  c1 += a0 * B[k][j+1];
  c2 += a0 * B[k][j+2];
  c3 += a0 * B[k][j+3];
          }
C[i][j]=c0;
C[i][j+1]=c1;
C[i][j+2]=c2;
C[i][j+3]=c3;   }
```

Register Blocking (5)
An example

- The number of L/S instructions is reduced and as a consequence the number of memory accesses
- **The number of arithmetical instructions is reduced too** as there are less address computations for C[i][j] and A[i][k]
  - **In the first case a different memory address is used for each load/store of *A[][]***
  - **Now, registers are used instead and therefore less memory addresses are computed**

// C code of MMM

```
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];
```

→

Step1

// C code of MMM

```
for (i=0; i<N; i++)
 for (j=0; j<N; j+=4) {
  for (k=0; k<N; k++) {
C[i][j] += A[i][k] * B[k][j];
C[i][j+1] += A[i][k] * B[k][j+1];
C[i][j+2] += A[i][k] * B[k][j+2];
C[i][j+3] += A[i][k] * B[k][j+3];
          } }
```

→

Step2

// C code of MMM

```
for (i=0; i<N; i++)
 for (j=0; j<N; j+=4) {
 c0=C[i][j];
 c1=C[i][j+1];
 c2=C[i][j+2];
 c3=C[i][j+3];

  for (k=0; k<N; k++) {
  a0=A[i][k];
  c0 += a0 * B[k][j];
  c1 += a0 * B[k][j+1];
  c2 += a0 * B[k][j+2];
  c3 += a0 * B[k][j+3];
          }
C[i][j]=c0;
C[i][j+1]=c1;
C[i][j+2]=c2;
C[i][j+3]=c3;   }
```

Register Blocking (6)

Activity


```
// C code of MMM
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];
```

→

```
// C code of MMM
for (i=0; i<N; i+=2)
  for (j=0; j<N; j+=2) {
    for (k=0; k<N; k++) {
                ...
                } }
```

Loop interchange

- The loop interchange transformation **switches the order of the loops** in order to improve data locality or increase parallelism
- **Not always safe,** only when data dependencies allow it
- In C/C++, accessing arrays column wise is inefficient (see next)

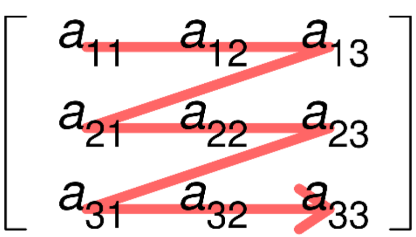
Column-wise (bad)

```
….
int i, j, N=1000;
int A[N][N];

for (j=0; j<N; j++)
 for (i=0; i<N; i++)
  A[ i ][ j ] = i+j;

….
```

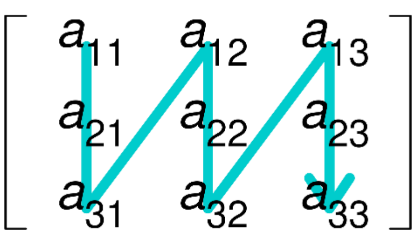
Row-wise (good)

```
….
int i, j, N=1000;
int A[N][N];

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  A[ i ][ j ] = i+j;

….
```

Row-major order



Column-major order



Loop interchange
A more complicated example

- **Which one is more efficient and why?**

```
for (j=0; j<N; j++)
 for (i=0; i<N; i++)
   total [ i ] = total [ i ] + A [ i ] [ j ];
```

loop interchange

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
   total [ i ] = total [ i ] + A [ i ] [ j ];
```

Loop interchange
A more complicated example

- *total [ ] is loaded and stored N^2 times*
- *all the intermediate results are loaded/stored from/to dL1*

  - *total[i] is invariant with respect to the inner loop and therefore it can be replaced by a register, yielding better data locality*
  - *This can be applied either manually or* ***automatically by compiling with '-O3'***

```
for (j=0; j<N; j++)
 for (i=0; i<N; i++)
  total [ i ] = total [ i ] + A [ i ] [ j ];
```

loop interchange

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  total [ i ] = total [ i ] + A [ i ] [ j ];
```

Scalar replacement

```
for (i=0; i<N; i++) {
  t = total [ i ];
 for (j=0; j<N; j++) {
  t = t + A [ i ] [ j ];
}
total [ i ] = t;  }
```

- *A[ ][ ] is accessed column-wise*
- *A[ ][ ] is accessed row-wise*

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
  1. both statements access the same memory location and at least one of them stores into it and
  2. there is a feasible run-time execution path from S1 to S2.

Data Dependencies – classification

- **Data dependencies reside into 3 categories**
  A. **Read after Write (RAW) or true dependence** → T=… / …=T
  B. **Write after Read (WAR) or anti-dependence** → …=T / T=…
  C. **Write after Write (WAW) or output dependence** → 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;
S2: T1=R2+5;

⇨

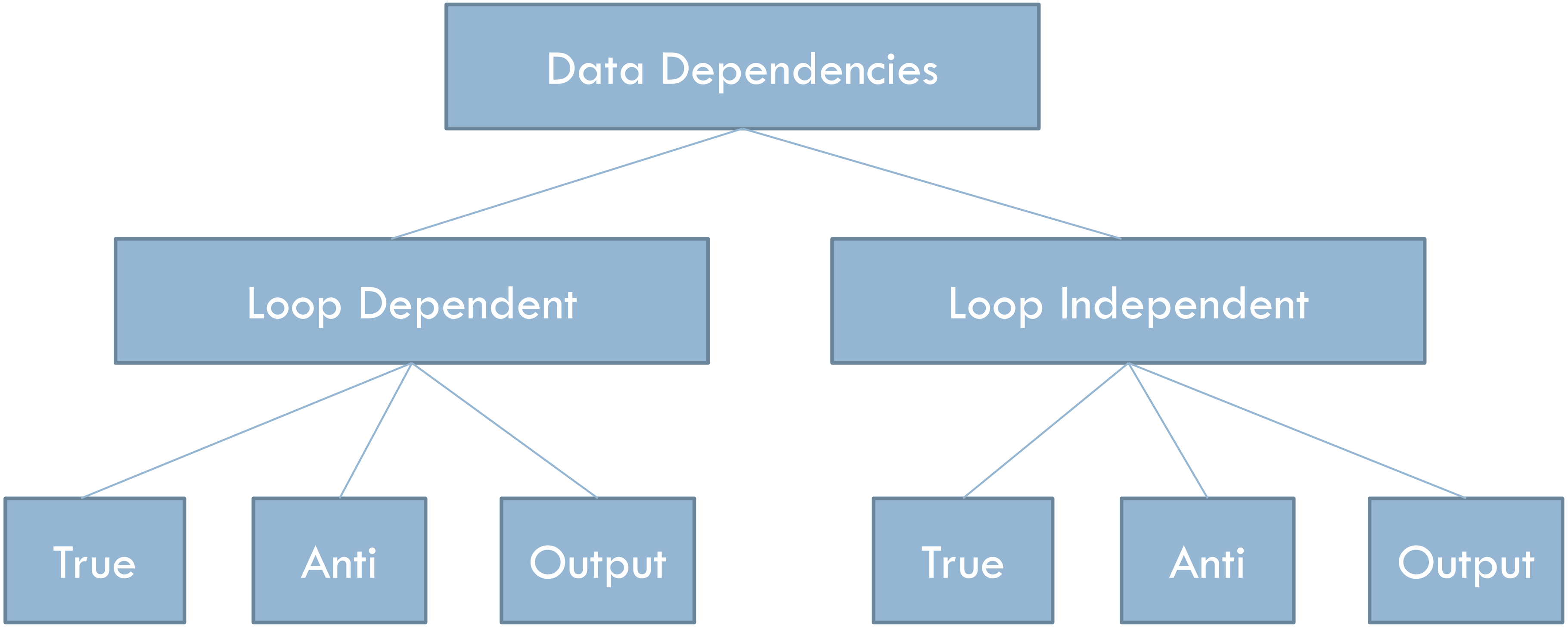
S1: T1=R1;
S2: T2=R2+5;

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

- **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 $S1 \xrightarrow{\delta_1^1} S1$

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]
…

True, Anti, Output

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

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

Loop dependent dependencies another 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 vectorised (see later on)

Data Dependencies
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 *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* $\mathbf{d}(\boldsymbol{i},\boldsymbol{j})$ is defined as a vector of length n such that $\mathbf{d}(\boldsymbol{i},\boldsymbol{j})_k = \boldsymbol{j}_k - \mathbf{i}_k$

- **Direction Vector:** is defined as a vector of length *n* such that

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

Data Dependencies
An 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++)

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

S1: A(i, j+2, k, n) = A(i, j, k, n+1) + temp;

- **Distance vector: d(i, j, k, n) = (0, 2, 0, -1)**
- **Direction vector: D(i, j, k, n) = (=, <, =, >)**

δ_2^1

- **The dependence is always given by the leftmost non '=' symbol**

Loop Merge also known as 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
also known as 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 ];
```

⇨

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

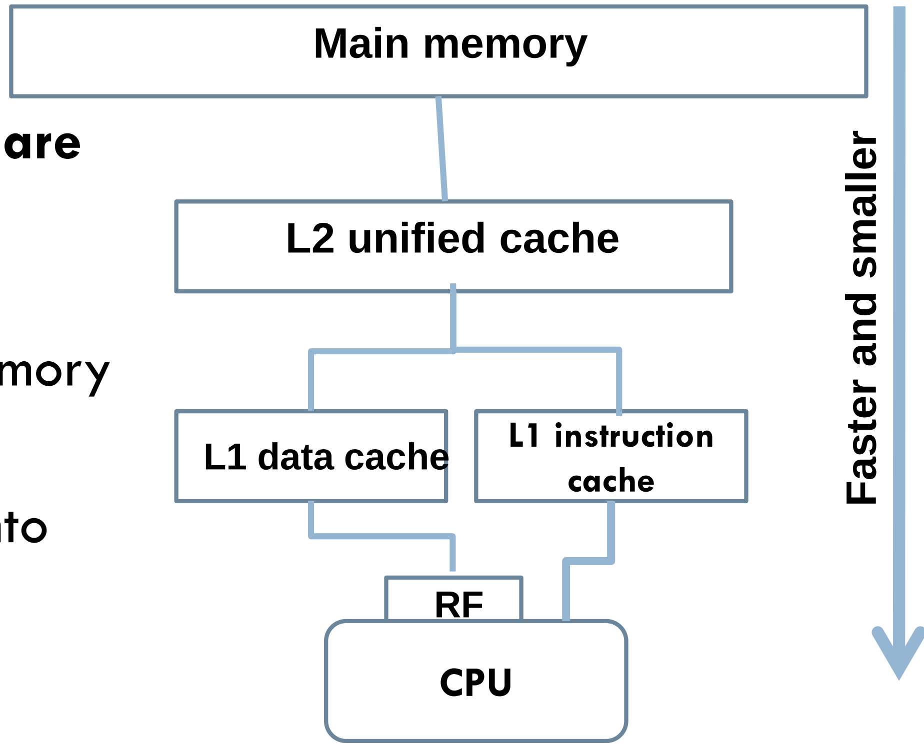
Loop Merge also known as Loop Fusion (3)


```
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 ];
}
```

- **Consider the case where the arrays are bigger than 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
    - 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 ];
```

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

```
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 ];
}
```

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

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

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

Loop Merge
not always safe

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

- **How can we 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 also known as 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 also known as 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 ];
}
```

⇨

```
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] = … ;
```

```
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;
```

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

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

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 – 1st example (1)


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

Dependence

↓

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

- **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 j** loop which gives $D(i, j) = (<, <)$
  - Then, loop interchange is valid as it gives $D(j, i) = (<, <)$

Loop Reversal – 1st example (2)


```
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 – 2nd example


```
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];
```

Loop carried dependence - The compiler cannot parallelize it

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

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

No dependence - The compiler can parallelize it or vectorise it

Loop Peeling
An 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++)
   A[i] = …
```

→

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

- Changes the loop bounds
- It is always safe

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

Loop Bump
1st example


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

→

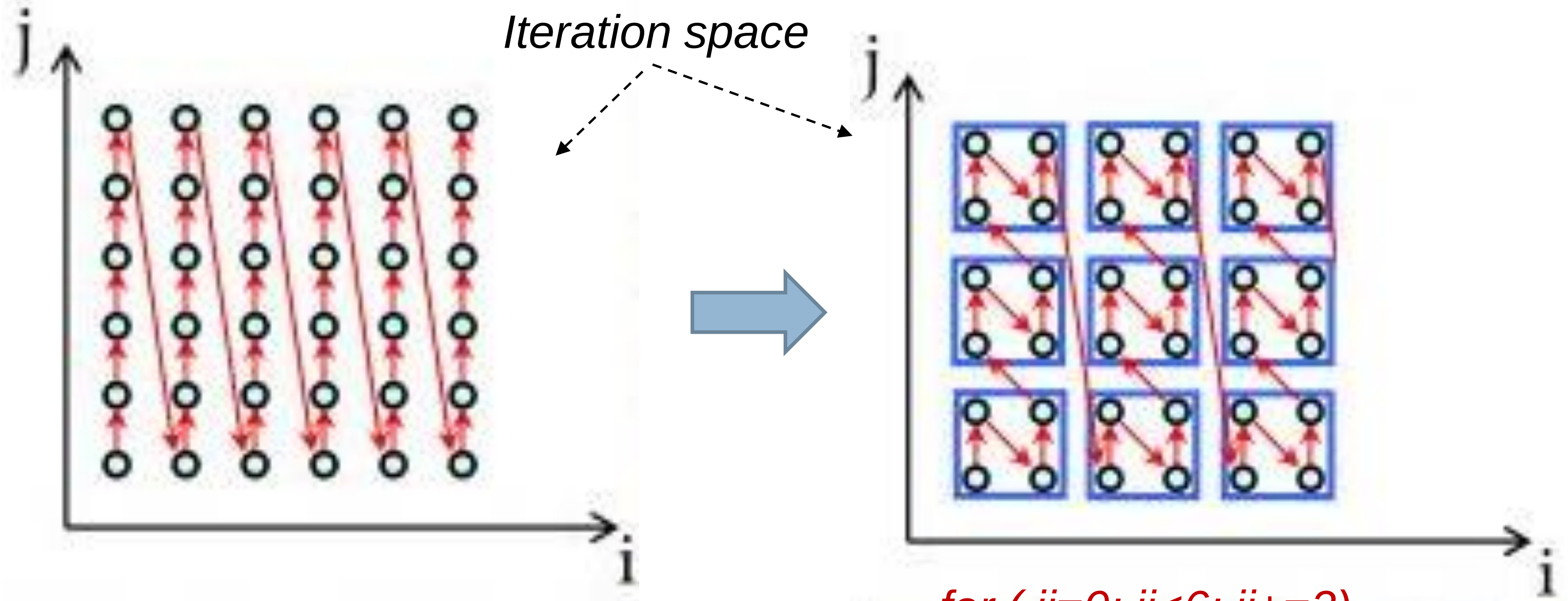
```
//array copying
 for (i=0;i!=N;i++)
  for (j=0;j!=N;j++)
   B_transpose[i][j]=B[j][i];

 for (i=0;i!=M;i++)
  for (j=0;j!=M;j++)
   for (k=0;k!=M;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
- Next week, we will learn how to use SSE/AVX x86-64 intrinsics.
  - These include prefetch instructions.
  - All the prefetch instructions supported for x86-64 architectures can be found here https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand=173,5533,3505,1449,3505,2940,2024&text=prefetch.
  - An example of a software prefetch instruction is shown below

    _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)




```
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, so as to help data remain in the cache (data reuse)

- The partitioning of loop iteration space leads to partitioning of large arrays into smaller blocks (tiles), thus fitting accessed array elements into 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 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 Li cache memory, the number of Li cache misses is reduced**
  - **The number of Li cache misses equals to the number of Li+1 accesses**

Loop Tiling / blocking (4)

- **Loop tiling reduces the number of cache misses**
  - **This doesn't entail performance improvement at all times – performance depends on other parameters too, e.g., 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)

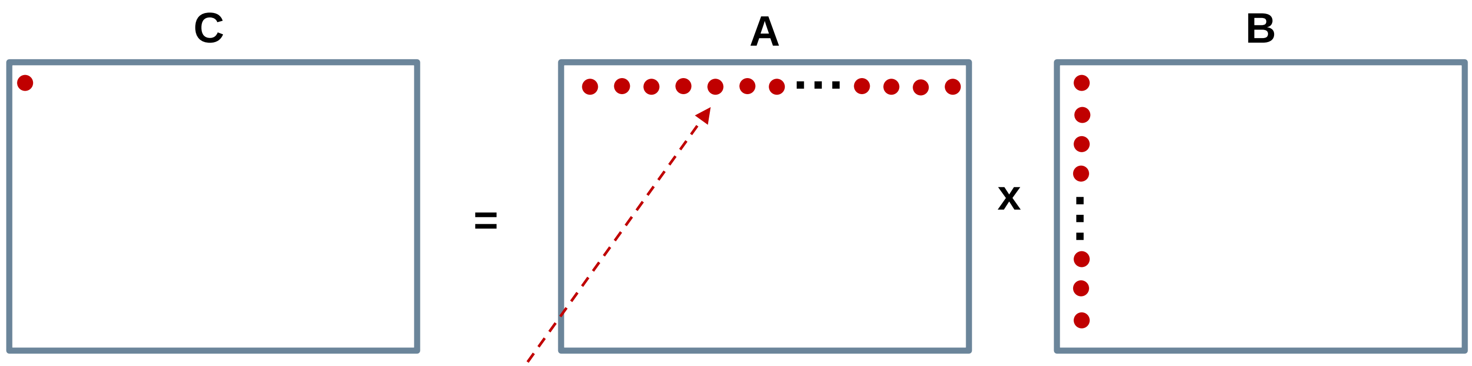
Pros:

- May increase locality (reduce cache misses)

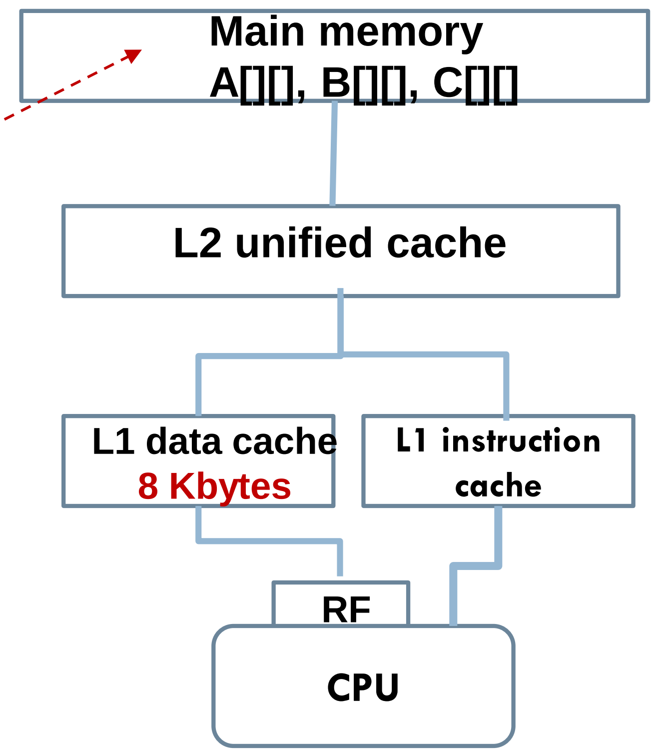
Cons:

- Increases the number of instructions (adds extra loops)

Loop tiling - Case Study Matrix-Matrix Multiplication Problem



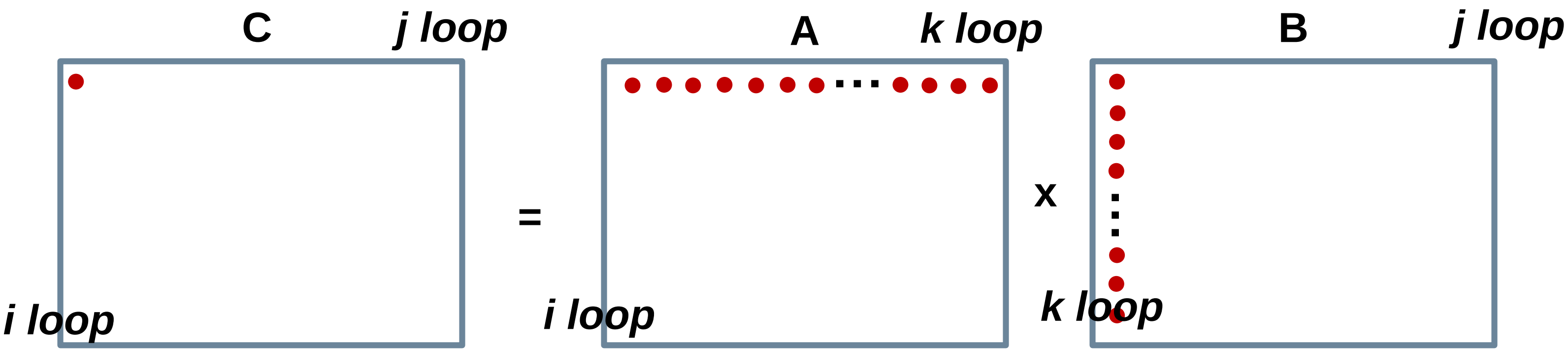
The size of each row of A is 8 kbytes



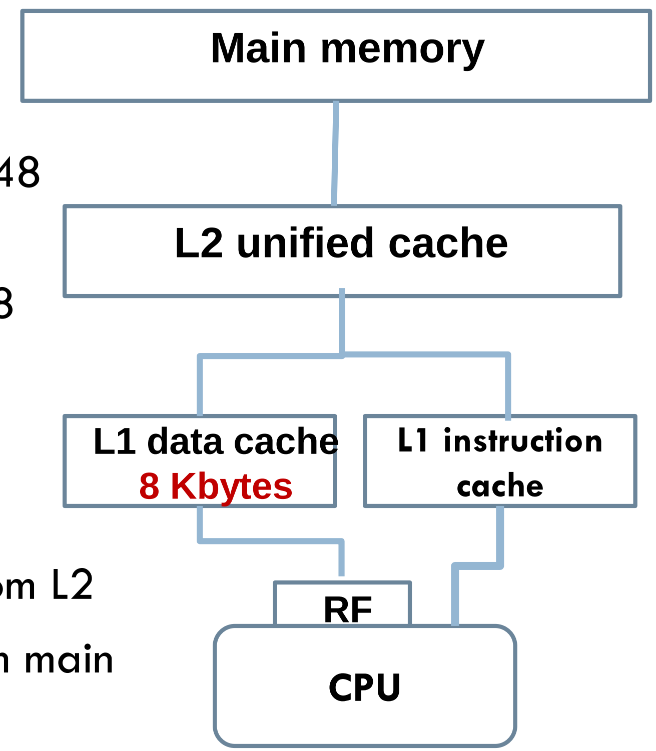
float C[2048][2048], A[2048][2048], B[2048][2048];

```
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];
```

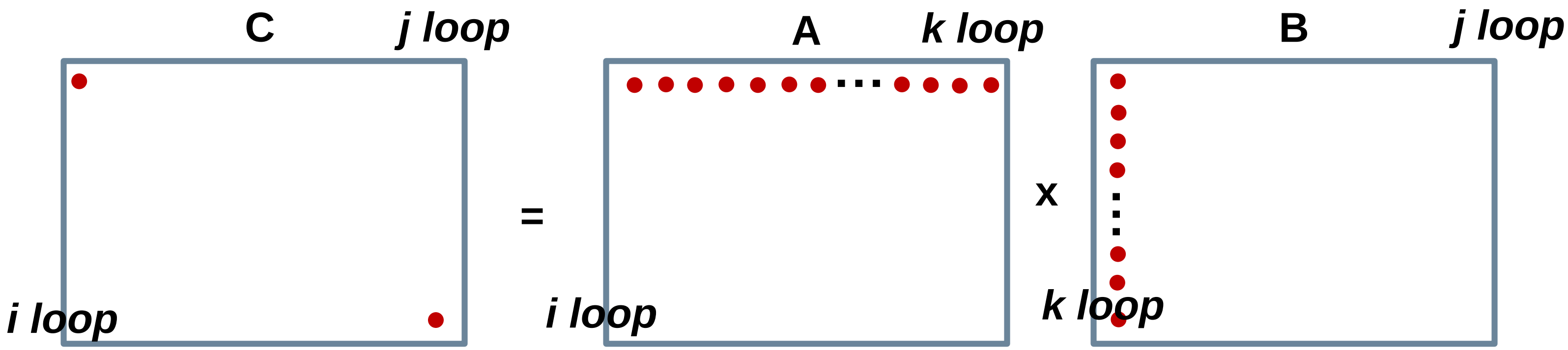
Loop tiling - Case Study Matrix-Matrix Multiplication Motivation



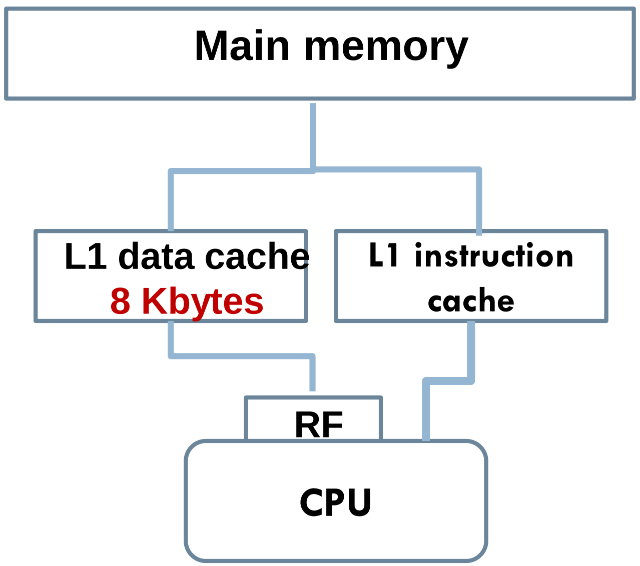
- Each row of A is multiplied by all the columns of B, thus:
  - **Each row of A is loaded from memory 2048 times**
  - If the row of A cannot remain in dL1, it will be loaded 2048 times from L2
  - If the row of A cannot remain in L2, it will be loaded 2048 times from main memory
- The whole B array is multiplied by each row of A, thus:
  - **B array is loaded 2048 times from memory**
  - If B cannot remain in dL1, 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 - Case Study
Matrix-Matrix Multiplication

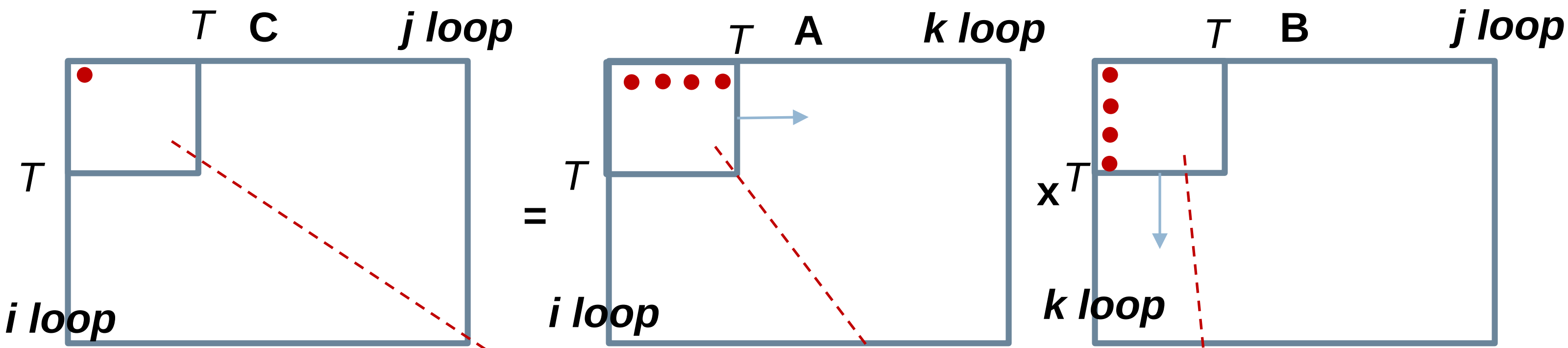


- Consider a single level of cache. In this case
  - A array is loaded 2048 times from main memory, **2048^3 loads**
  - B array is loaded 2048 times from main memory, **2048^3 loads**
  - C array is stored just once, **2048^2 stores**



Loop tiling - Case Study

Matrix-Matrix Multiplication – 1 level of cache (1)



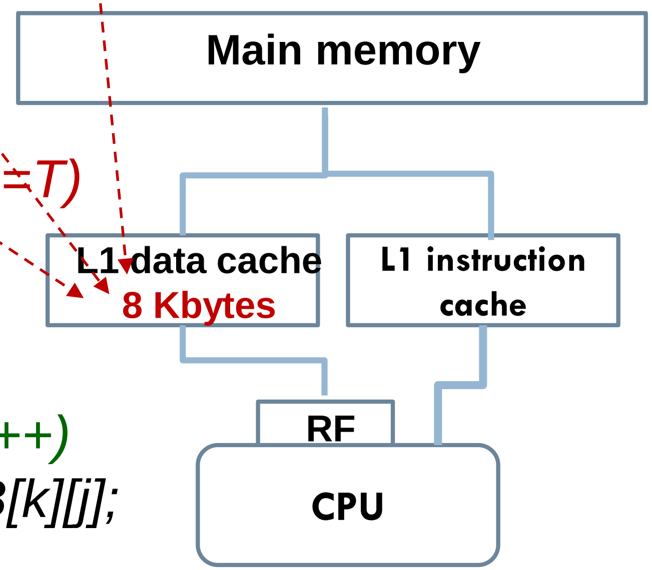

```
for (i=0; i<2048; i++)
  for (j=0; j<2048; j++)
    for (k=0; k<2048; k++)
      C[i][j] += A[i][k] * B[k][j];
```

These loops specify which tiles to multiply

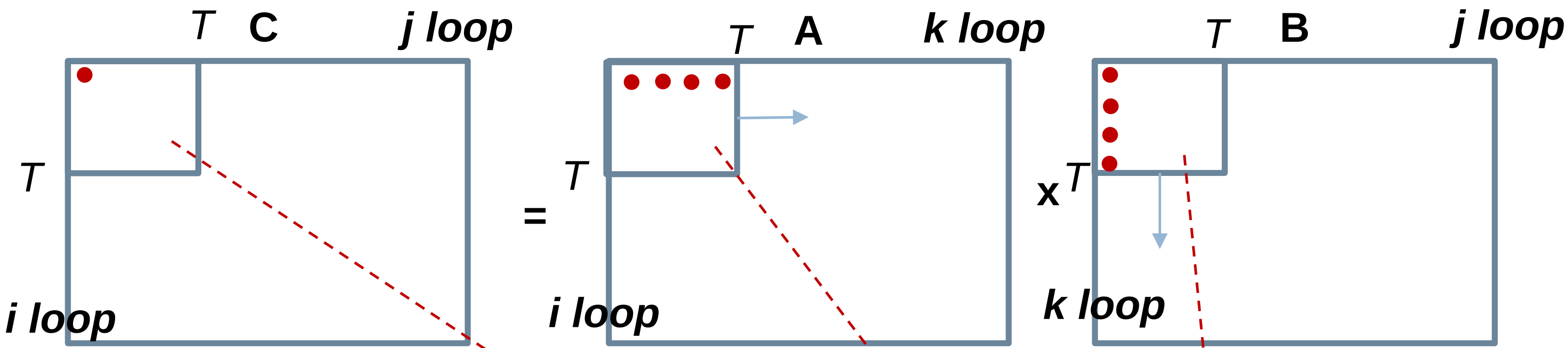
```
for (ii=0; ii<2048; ii+=T)
  for (jj=0; jj<2048; jj+=T)
    for (kk=0; kk<2048; kk+=T)
```

These loops specify which elements inside the tile to multiply

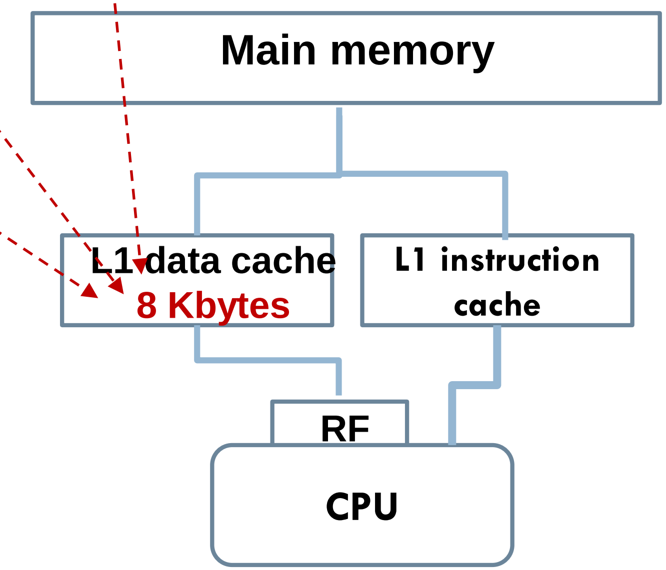
```
      for (i=ii; i<ii+T; i++)
        for (j=jj; j<jj+T; j++)
          for (k=0; k<kk+T; k++)
            C[i][j] += A[i][k] * B[k][j];
```



Loop tiling - Case Study
Matrix-Matrix Multiplication – 1 level of cache (2)

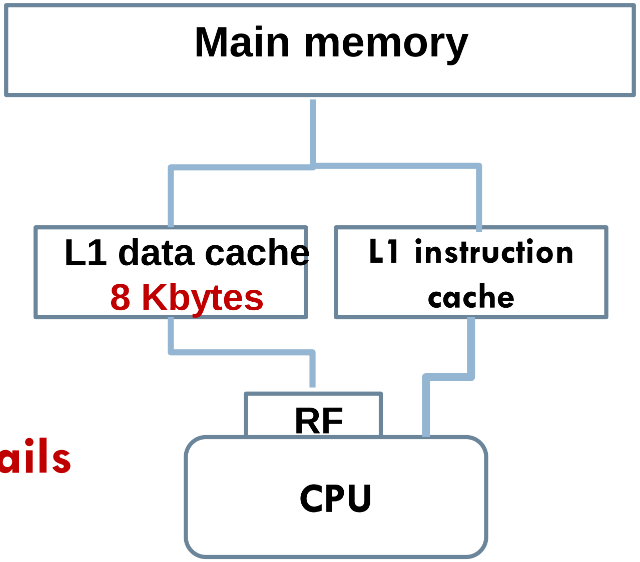


- The matrices are partitioned into smaller sub-matrices (TxT)
- Instead of multiplying A[][] by B[][], their tiles are multiplied
  - The tiles are small enough in order to fit in the cache
  - A array is loaded **2048/T** times from main memory
  - B array is loaded **2048/T** times from main memory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



Loop tiling - Case Study
Matrix-Matrix Multiplication – **1 level of cache (3)**

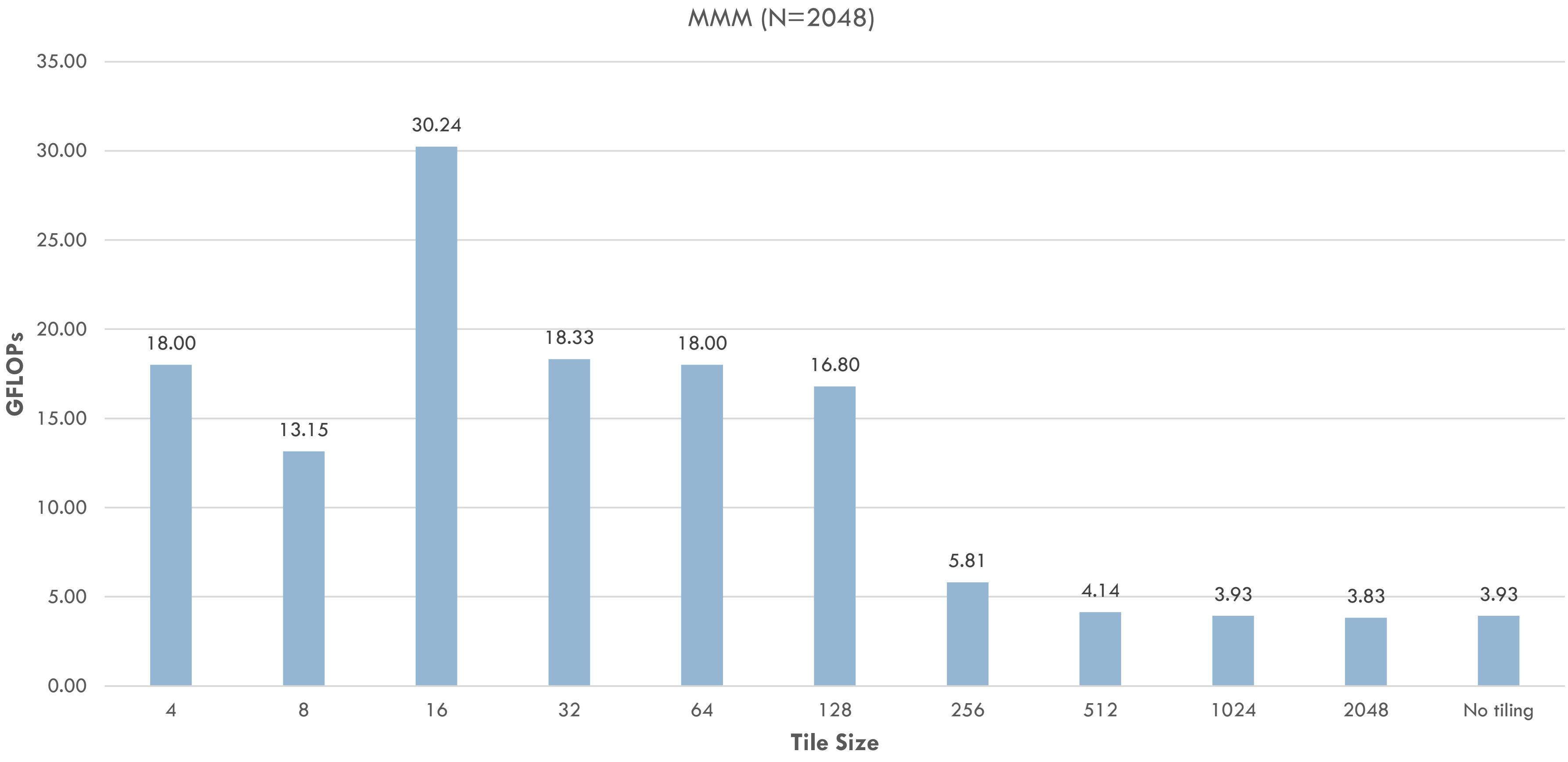
- Before applying loop tiling
  - A: 2048 x (2048x2048) loads from main memory
  - B: 2048 x (2048x2048) loads from main memory
  - C: 1 x (2048x2048) stores to main memory
  - **In total, $2*2048^3 + 2048^2$ main memory accesses**
- After applying loop tiling
  - A: 2048/T x (2048x2048) loads from main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

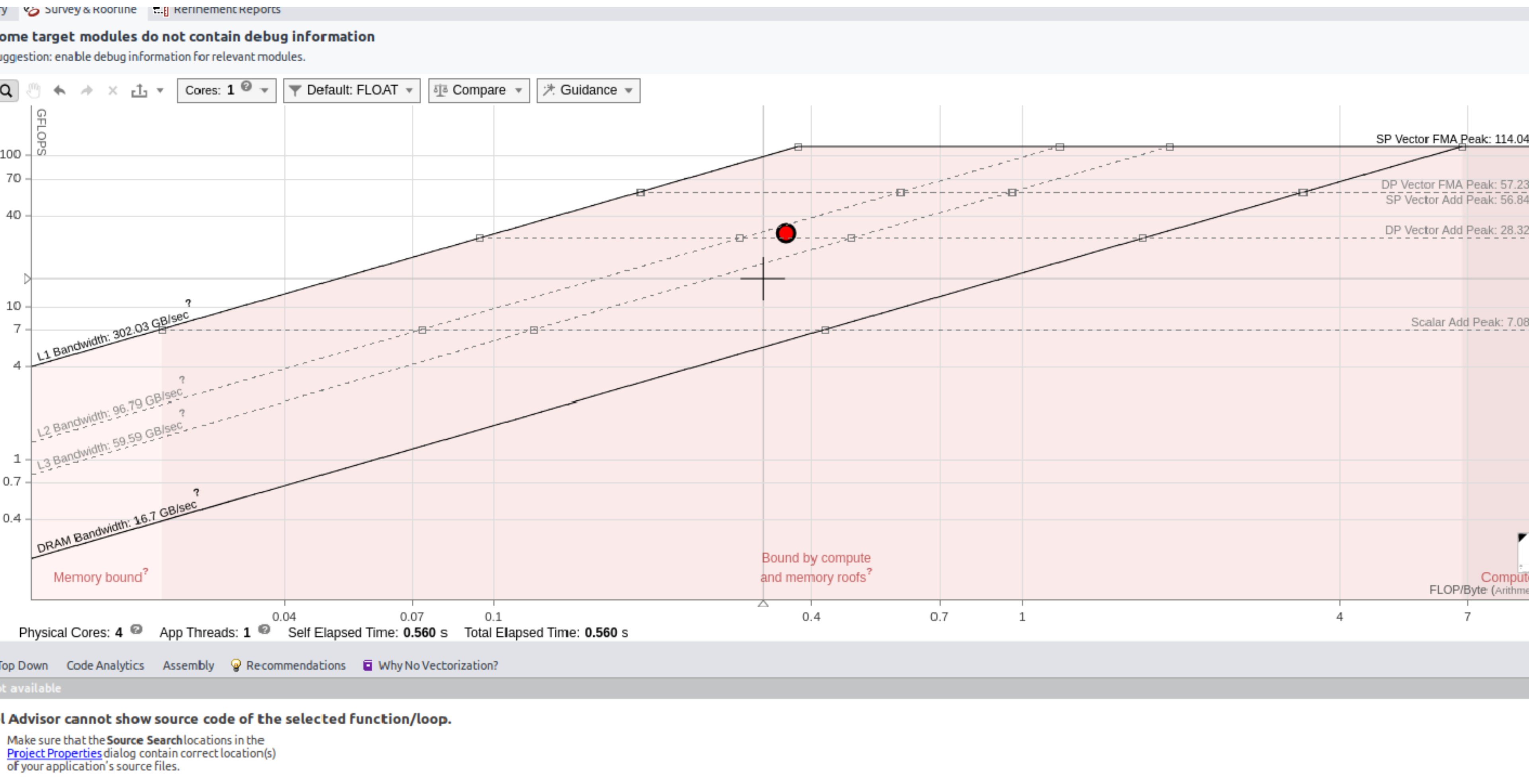
MMM – Loop Tiling Performance Evaluation

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



MMM – Loop Tiling Performance Evaluation (2)

- Roofline analysis for T=16



Further Reading

- Optimizing compilers for modern architectures: a dependence-based approach, book, available at https://liveplymouthac-my.sharepoint.com/:b:/g/personal/vasilios_kelefouras_plymouth_ac_uk/EVy4Laj_1W9Hr7D3W57CBuQBeohd0M9iVVT7x5n91PcDyg?e=RGnRCa
- Options That Control Optimization, available at
  https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html