Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«Code Optimizations»

Παναγιώτης Χατζηδούκας

Outline

- Code optimization: key problems
- Some basic/simple code optimizations/transformations and manually applied techniques:
  - Use the available Compiler Options
  - Use the appropriate precision
  - Choose a better algorithm
  - Reduce complex operations
  - Loop-based strength reduction
  - Dead code elimination
  - Common subexpression elimination
  - 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 programming 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 several code transformation and 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?

Software Optimization Problem

- The key to optimizing software is the correct choice, order, and parameters
  of code optimizations
- But why optimizing software 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
- The compilers cannot find the optimum choice, order, and parameters of optimizations

Why compilers fail

- 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, instruction set

Why compilers fail

- 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

Why compilers fail

- 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

Data handling techniques

- Use Variables of the Same Type for Processing
- Use of Unsigned Type
- Float and Double
- Data Declaration - Constant
- Data Declaration - Volatile
- Data Initialization at Declaration
- Data Definition - Arrangement and Packing
- Passing Reference as Parameters
- Return Value
- Better Data Structure and Representation
- Array and Structure Initialization

Data handling techniques

- Use Variables of the Same Type for Processing
  - Programmers should plan to use the same type of variables for processing. Type conversion must be avoided.
  - Otherwise, precious cycles will be wasted to convert one type to another (Unsigned and signed variables are considered as different types).
- Use of Unsigned Type
  - All variables must be defined as “unsigned” unless mathematical calculation for the signed bit is necessary. The “signed-bit” may create complication, unwanted failure, slower processing and extra ROM size.
- Float and Double
  - Maximum value of Float = 0x7F7F FFFF
  - Maximum value of Double = 0x7F7F FFFF FFFF FFFF
  - To avoid unnecessary type conversion or confusion, programmers can assign the letter “f” following the numeric value.
  - x = y + 0.2f;

Data handling techniques

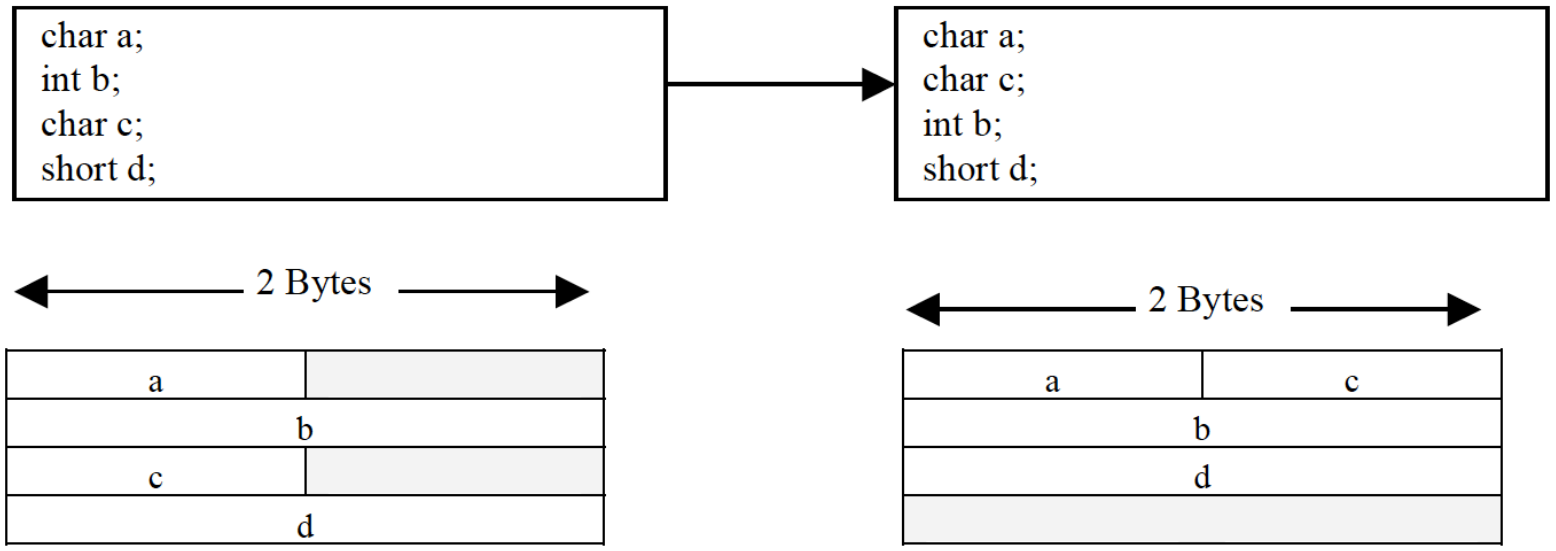
- Data Declaration - Constant
  - The “const” keyword is to define the data as a constant, which will allocate it in the ROM space (section C). Otherwise, a RAM space will also be reserved for this data. This is unnecessary as the constant data is supposed to be read-only.
- Data Declaration - Volatile
  - “Volatile” keyword will forbid the compiler from performing any optimization on the variable. This is usually used on IO registers and variables that will be altered by interrupts. This is necessary as the value of these variables can be asynchronously accessed.
- Data Initialization at Declaration
  - Data should be initialized at declaration.

```
int a;
void main(void)
{ a = 1;
…
```

```
int a = 1;
void main(void)
{
…
```

Data handling techniques

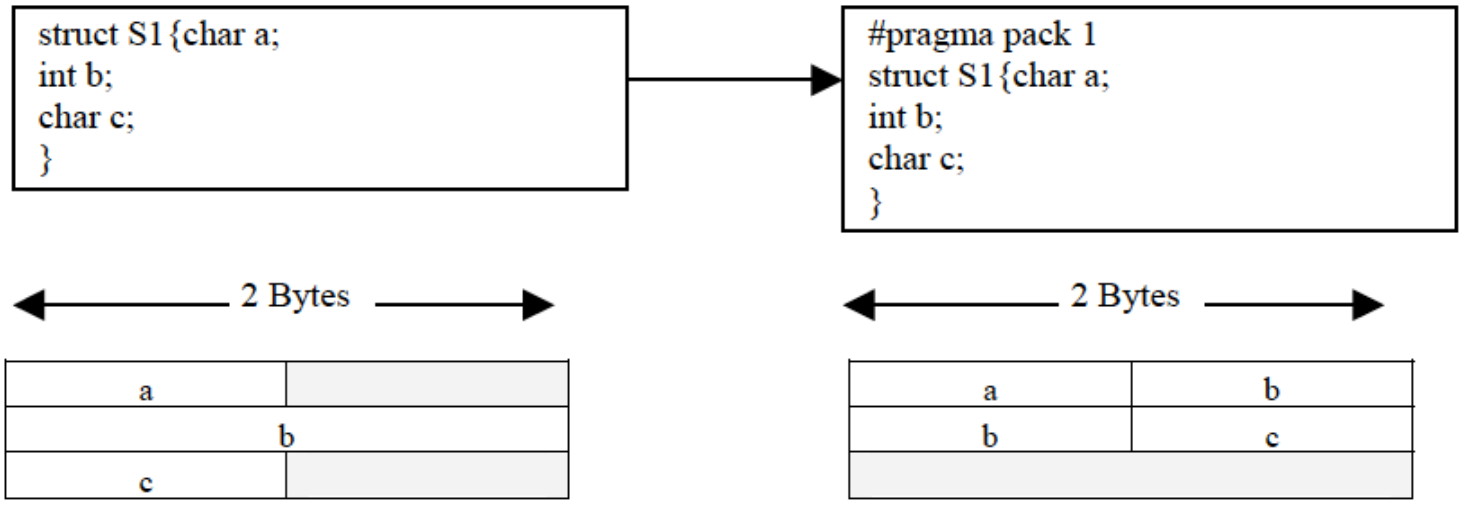
- Data Definition - Arrangement and Packing
  - The declaration of the components in a structure will determine how the components are being stored.
  - Due to the memory alignment, it is possible to have dummy area within the structure. It is advised to place all similar size variables in the same group.



Example for a specific embedded system where sizeof(int)=2

Data handling techniques

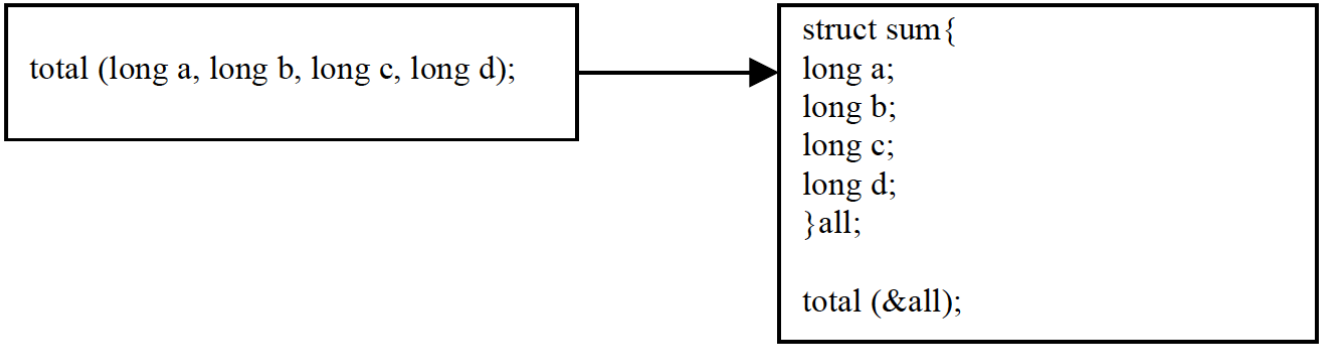
- Data Definition - Arrangement and Packing
  - As the structure is packed, integer b will not be aligned. This will improve the RAM size but operational speed will be degraded, as the access of ‘b’ will take up two cycles.



Example for a specific embedded system where sizeof(int)=2

Data handling techniques

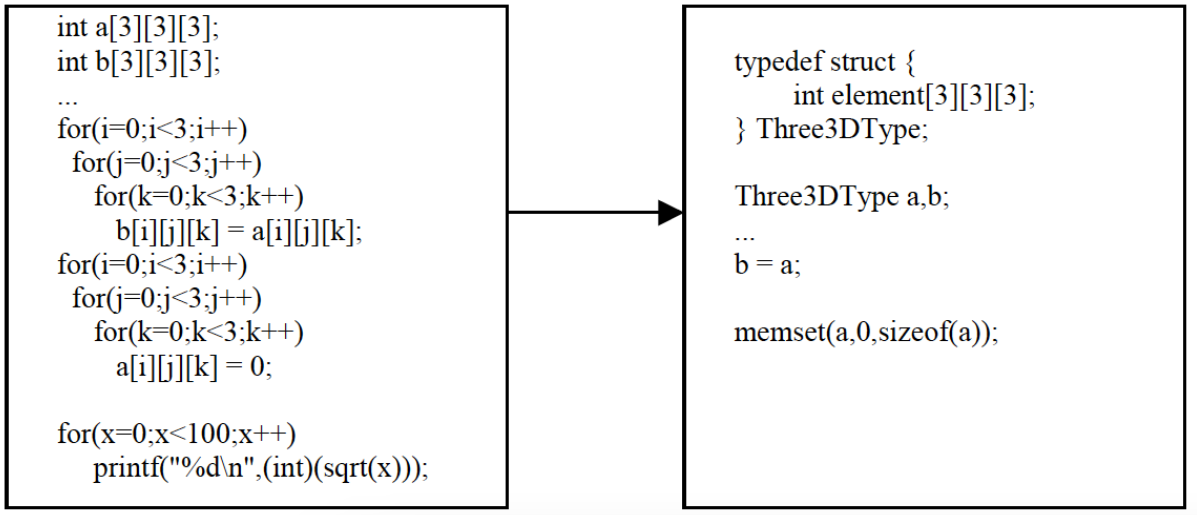
- Passing Reference as Parameters
  - Larger numbers of parameters may be costly due to the number of pushing and popping actions on each function call.
  - It is more efficient to pass structure reference as parameters to reduce this overhead



- Return Value
  - The return value of a function will be stored in a register. If this return data has no intended usage, time and space are wasted in storing this information.
  - Programmer should define the function as “void” to minimize the extra handling in the function.

Data handling techniques

- Better Data Structure and Representation
  - Proper data structure consideration can improve the program.
  - Example: Use computation to regenerate a large junk of data (compression, technique), this will reduce the space usage. However, the computation process may slow down the operation.
    - An array of [0,0,0,0,0,0,1,1,2,2,2,3,3,3,3…], this can be replaced with [6, 2, 3, 4…], which signifies 6x'0', 2x'1', 3x'2', 4x'3'…
- Array and Structure Initialization
  - A simple illustration of implementation:



```
int a[3][3][3];
int b[3][3][3];
...
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      b[i][j][k] = a[i][j][k];
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      a[i][j][k] = 0;

for(x=0;x<100;x++)
    printf("%d\n",(int)(sqrt(x)));
```

```
typedef struct {
      int element[3][3][3];
} Three3DType;

Three3DType a,b;
...
b = a;

memset(a,0,sizeof(a));
```

Other handling techniques

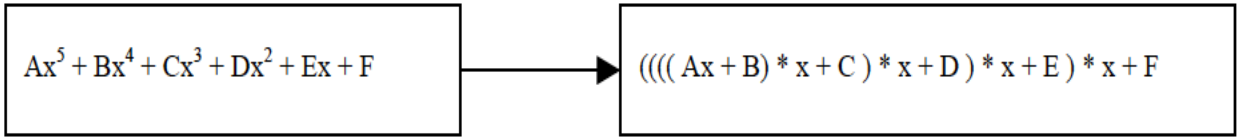
- Lookup Table and Calculation
- Fixed-point and Floating-point Arithmetic
- Horner's Rule of Polynomial Evaluation
- Factorization
- Modula
- Division and Multiplication
- Constant in Shift Operations
- Use Formula
- Simplify Condition
- Absolute Value

Other handling techniques

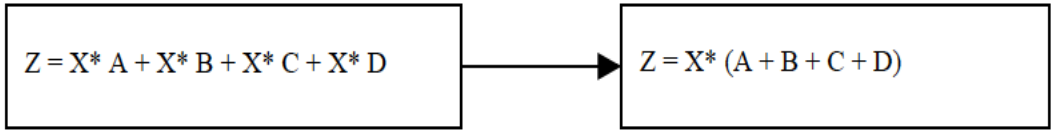
- Lookup Table and Calculation
  - In lower operation frequency of MCU, lookup table may be faster than recalculation methods. However, programmers must make their judgment on the complexity and speed requirement.
  - Example:
    - A function like $y = ax + bx^2$ will already take up significant CPU processing time.
    - However, the function $y = 2x$ can be implemented with a shift instruction (2 cycles). Thus this function is preferred to be implemented with re-calculation method than a lookup table method.
- Fixed-point and Floating-point Arithmetic
  - It takes up much processing power to perform floating-point arithmetic in a non-floating-point processor.
  - If accuracy is not a requirement, programmers should use fixed-point calculation instead. Otherwise, the calculation can be re-implemented in a cheaper means.
  - Example:
    - 123.45 + 678.89 is equivalent to (12345 + 67889) with a decimal point placed at the correct place.

Other handling techniques

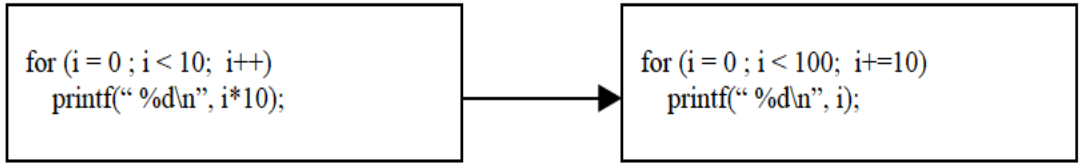
- Horner's Rule of Polynomial Evaluation
  - The rules state that a polynomial can be rewritten as a nested factorization. The reduced arithmetic operations will have better ROM efficiency and execution speed.



- Factorization
  - The compiler may be able to perform better when the formula



- Use Finite Differences to Avoid Multiplies



Other handling techniques

- Modula

```
x = y %32 ;
```

→

```
x = y &31;
```

- Division and Multiplication

```
x = y * 8 ;
x= y / 8 ;
```

→

```
x = y <<3;
x = y >>3;
```

- Constant in Shift Operations

```
int shift=8;

data = data <<shift;
```

→

```
#define SHIFT 8

data = data << SHIFT;
```

- Use Formula

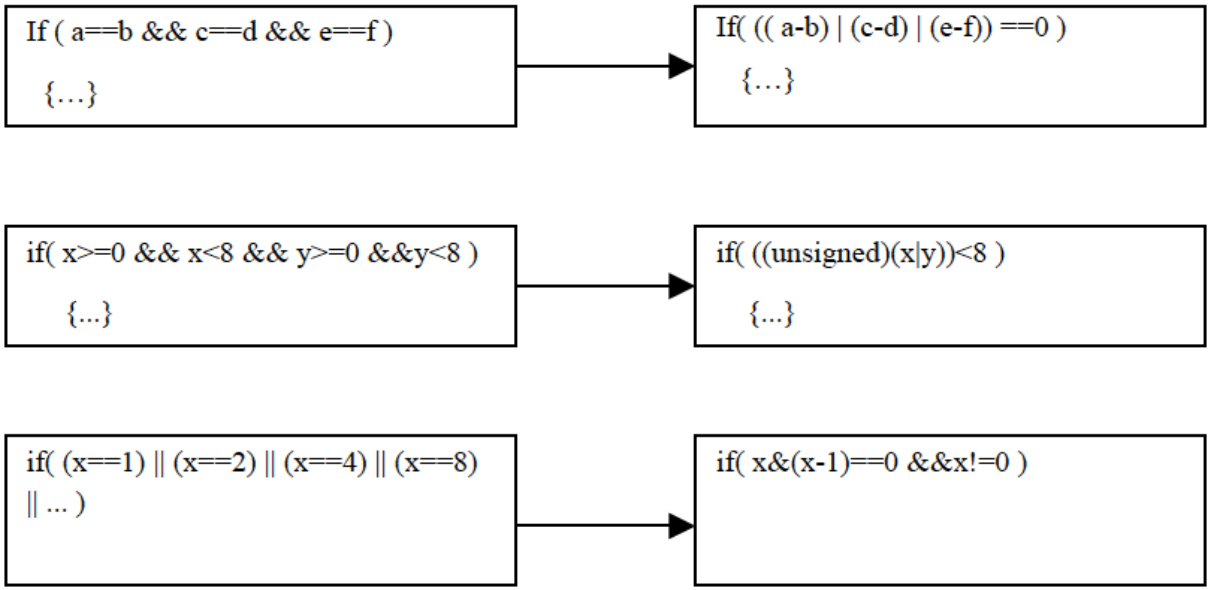
```
n=100;
for (x=0, y=1; y<=n; y++)
        x +=y;
```

→

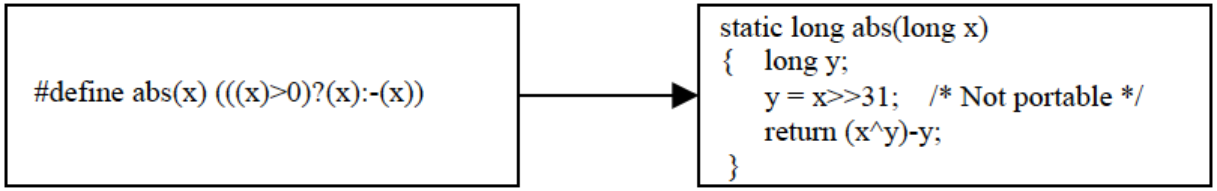
```
n=100;
x = n* (n >>1);  //n^2 /2
```

Other handling techniques

- Simplify Condition



- Absolute Value



Basic code improvement techniques

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

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: it is not always safe for codes using floating point arithmetic
  - -Osize: optimizes for code size

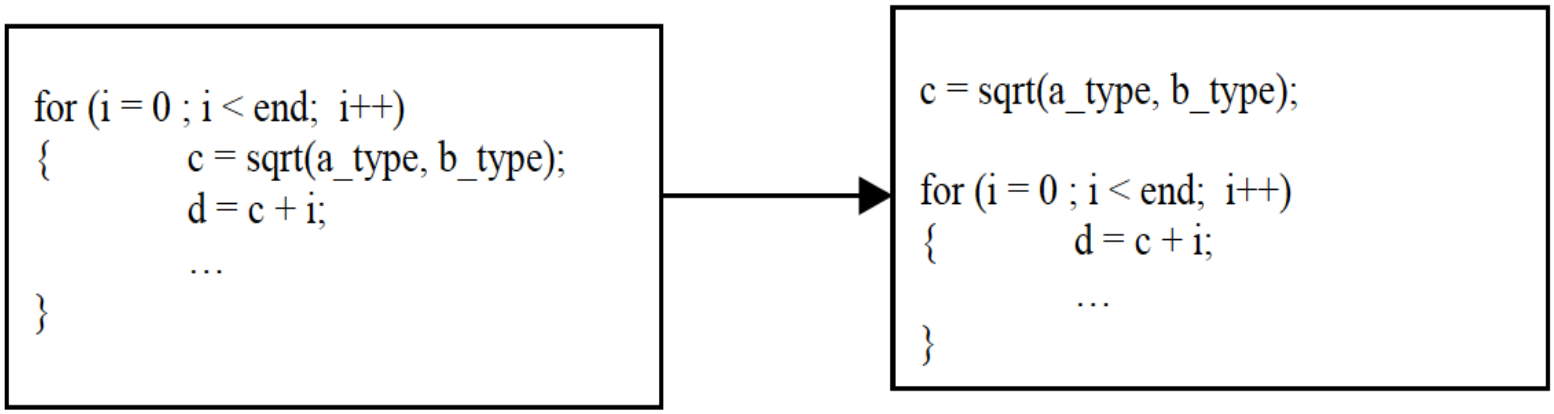
Dead code elimination

- Compiler optimization that removes dead code (code that does not affect the program results).
- Benefits:
  - it shrinks program size, an important consideration in some contexts
  - it allows the running program to avoid executing irrelevant operations, which reduces its running time

```
int foo(void) {
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

Common subexpression elimination

- Common expression should be calculated once or earlier
  - A parameter can be calculated at earlier stages, such as the power-up initialization stage instead of the actual execution stage.
  - This will help to speed up the processing.



Use LookUp Table (LUT)

- A LUT is an initialized array that contains precalculated information.
- They are typically used to avoid performing complex (and hence time-consuming) calculations.
- Example
  - In a data transmission system, a Packet Error Checking (PEC) byte can be appended at the end of each transaction as an error-detecting code. The PEC byte is calculated based on a CRC-8 byte represented by the polynomial $C(X) = X^8 + X^2 + X^1 + 1$.
  - It is well known that the speed of CRC calculations may be significantly increased by the use of a lookup table.
  - A suitable lookup table for computing the CRC (or PEC) used in System Management Bus (SMBUS) protocol calculations is shown next.

LUT Example

```
unsigned char pec_Update(unsigned char pec)
{
  static const unsigned char lookup[256] =
  {
    0x00U, 0x07U, 0x0EU, 0x09U, 0x1CU, 0x1BU, 0x12U, 0x15U,
    ...
    0xE6U, 0xE1U, 0xE8U, 0xEFU, 0xFAU, 0xFDU, 0xF4U, 0xF3U
  };

  pec = lookup[pec];
  return pec;
}
```

- Interesting points
  - The use of *static*: to avoid the allocation in the stack
  - The use of *const*: to avoid writing to the lookup table
  - The use of *__flash*: to force the array to be kept in flash memory (for embedded systems)
  - The use of a size-specific data type such as *unsigned char*: to reduce memory consumption
  - Avoidance of incomplete array declarations: explicitly declared array size to avoid errors due to missing values
  - Range Checking: useful but not necessary in this case

Use the appropriate precision / data type

- The use of correct data type is important in a recursive calculation or large array processing. The extra size, which is not required, is taking up much space and processing time.
- Example
- Speed concern:
  - Byte multiplication - MULXU .B R1L,R2L - take up 12 cycles
  - Word multiplication - MULXU.W R1,ER2 - take up 20 cycles
- Size concern:
  - char data_collect[100];
  - long data_collect[100]; -take up 4 times more spaces

Choose a better algorithm

- An example for data sorting algorithms

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Reduce complex operations (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

Reduce complex operations (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 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]

Loop unrolling

- Creates additional copies of the loop body
- Always safe to apply

//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
  - Increases instruction parallelism
- Cons:
  - Increases code size
  - Increases register pressure

Loop unrolling

- The number of arithmetical instructions is reduced
  - Less add instructions for i, i.e., i=i+4 instead of i=i+1
  - Less compare instructions, i.e., i==100 ?
  - Less jump instructions

//C-code1

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

```
add ip, r1, #800
.L2:
ldrd r2, [r1], #8
strd r2, [r0], #8
cmp r1, ip
bne .L2
```

//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];
}
```

```
add ip, r1, #800
.L5:
ldrd r2, [r1]
strd r2, [r0]
ldrd r2, [r1, #8]
strd r2, [r0, #8]
ldrd r2, [r1, #16]
strd r2, [r0, #16]
ldrd r2, [r1, #24]
strd r2, [r0, #24]
adds r1, r1, #32
adds r0, r0, #32
cmp r1, ip
bne .L5
```

https://godbolt.org/z/K9WEbrv7W

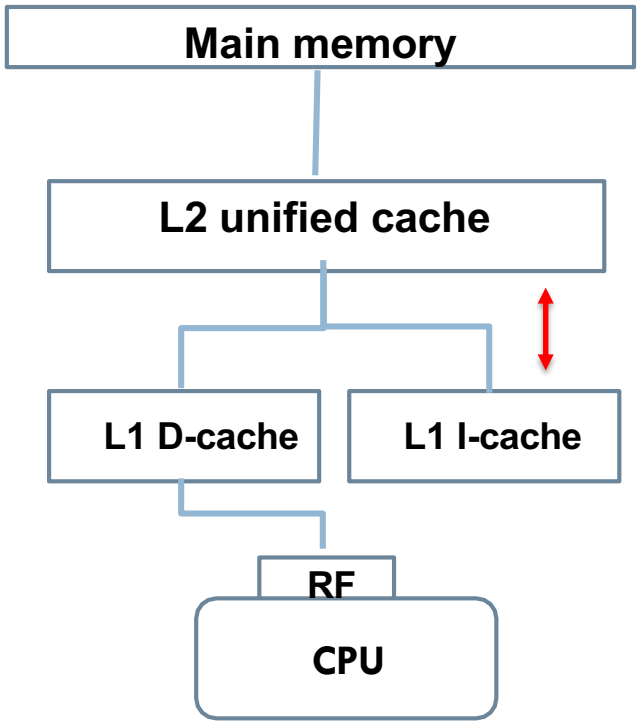
Limit in Loop unrolling

- A larger loop unrolling factor does not mean more efficient code

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

```
//C-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+99999] = B[i+9999];
}
```

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



Scalar replacement transformation

- Converts array reference to scalar reference
  - Most compilers will do this for you automatically by specifying the -O2 option
  - Always safe
  - Reduces the number of L/S instructions
  - Reduces the number of memory accesses

//Code-1

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

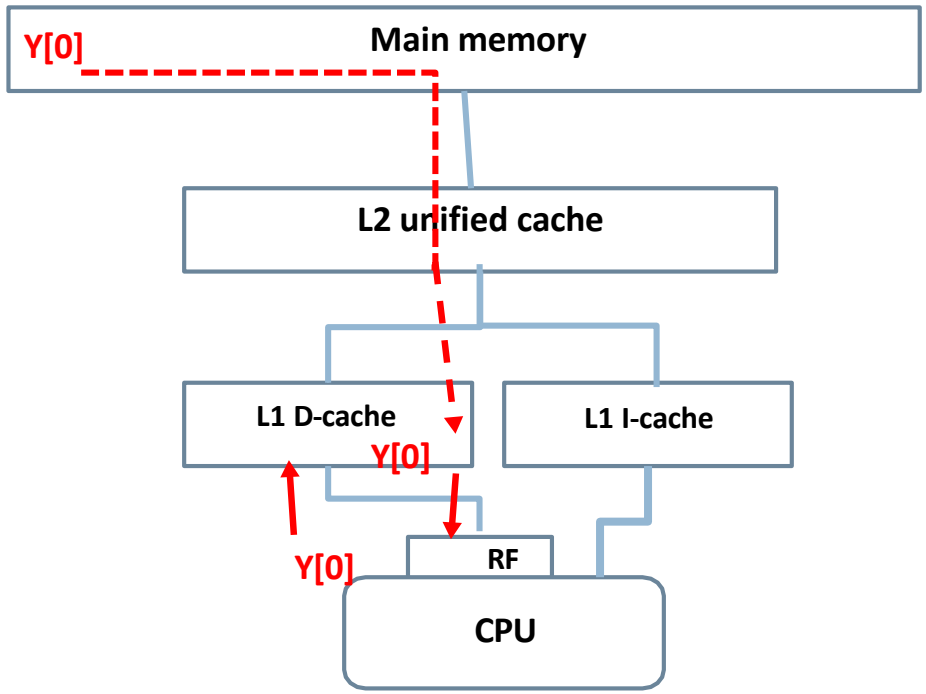
//Code-2

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

Scalar replacement: example

```
// 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 the j loop
- For every j, Y[i] is redundantly loaded/stored from/to memory
- A load/store instruction needs 1-3 CPU cycles
- The transformed code has fewer load/stores and L1 data accesses

Strength Reduction

- 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 the '-O1' optimization flag

//Code-1

```
for (i=0; i < n; i++){
  A[i] = A[i] + c*i;
}
```

//Code-2

```
T = 0;
for (i=0; i < n; i++){
  A[i] = A[i] + T;
  T = T + c;
}
```

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

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 the '-O1' optimization flag

//Code-1

```
for (i=0; i < n; i++){
  A[i] = A[i] + sqrt(x);
}
```

//Code-2

```
if (n>=0) C = sqrt(x);
for (i=0; i < n; i++){
  A[i] = A[i] + C;
}
```

- 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 ways
  - Either manually or automatically
  - '-O1' applies function inline
  - In C, a good option is to use macros instead (if possible)
- Pros
  - It speeds up your program by avoiding function-calling overhead
  - It saves the overhead of pushing/popping on the stack
  - It saves the overhead of a return call from a function
  - It increases the locality of reference by utilizing the instruction cache
- Cons
  - The main drawback is that it increases the code size

Function Inline

- The technique will cause the compiler to replace all calls to the function, with a copy of the function's code.
  - This will eliminate the runtime overhead associated with the function call.
  - This is most effective if the function is called frequently but contains only a few lines of code.

```
#pragma inline (sum)
…
int sum(int a, int b)
{           return (a+b);
}
…
routine()
{           …
            total  = sum (x,y);
            …
            sub_total = sum (cost_a, cost_b)
            …
}
```

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 the '-O3' optimization flag

```
//Code-1
for (i = 0; i < N; i++) {
  if (x<0)
    a[i] = 0;
  else
    b[i] = 0;
}
```

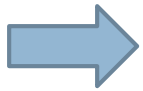
```
//Code-2
if (x<0)
  for (i = 0; i < N; i++) {
    a[i] = 0;
  }
else
  for (i = 0; i < N; i++) {
    b[i] = 0;
  }
```

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)

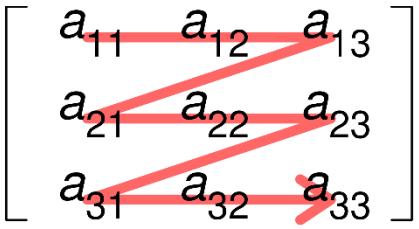
```
….
int i, j, N=1000;
int A[N][N];

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

….
```



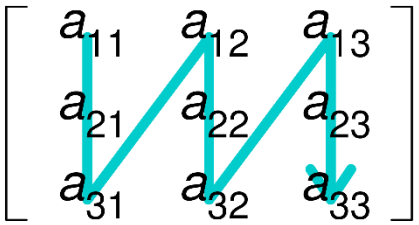
Row-wise (good)

```
….
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



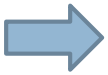
Column-major order



Loop interchange

- The following example is more complicated
- 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

- *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 ];
```

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

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

loop interchange

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

Scalar replacement

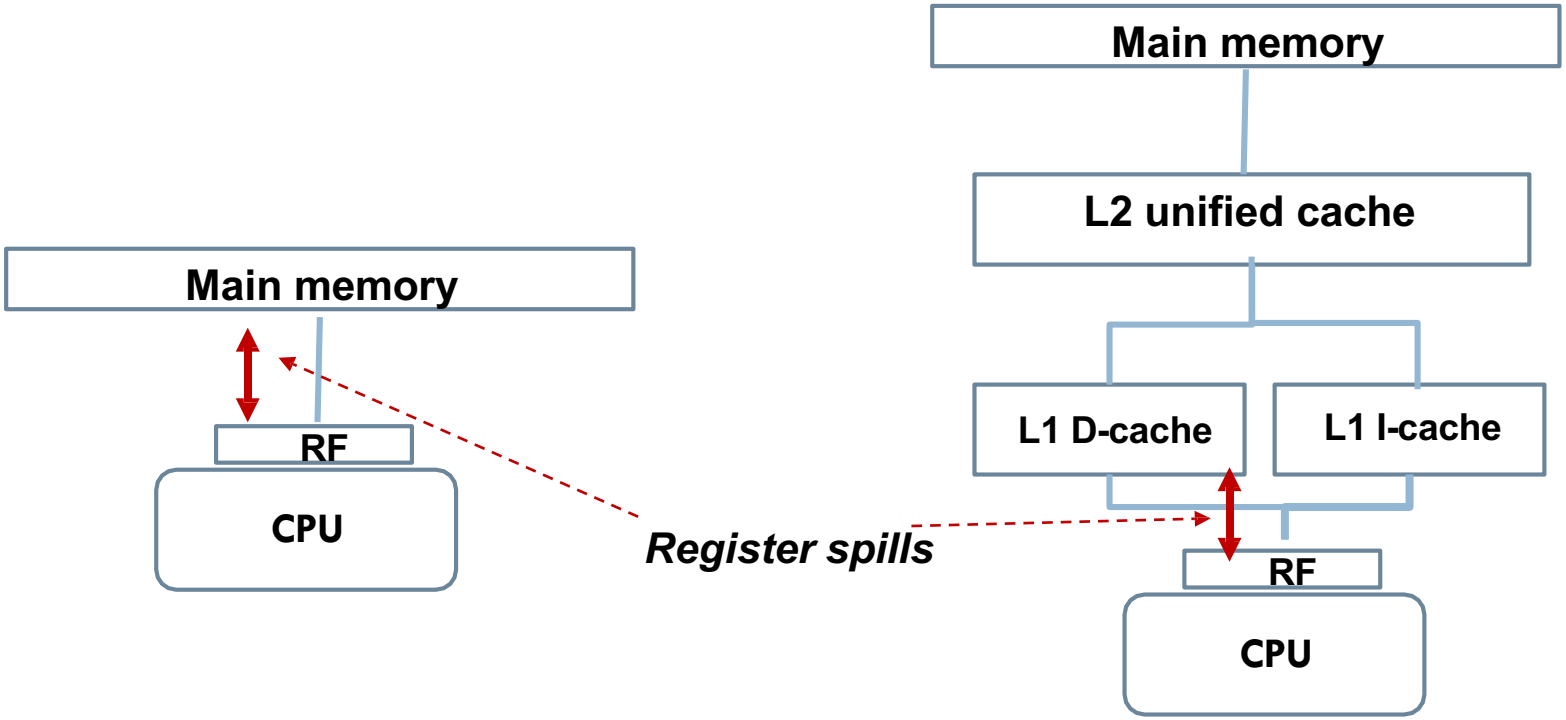
```
for (i=0; i<N; i++) {
  t = total [ i ];
  for (j=0; j<N; j++) {
    t = t + A [ i ] [ j ];
  }
  total [ i ] = t;
}
```

Register Blocking (1)

- Also known as “Loop unroll and jam”
- 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: 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



Two steps of Register Blocking

- 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)
- 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];
```

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

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];

  }
 }
```

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;
}
```

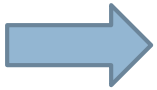
Register Blocking: example

- *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: (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

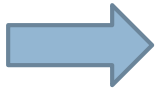
```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```



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

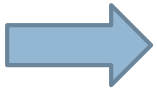
// C code of MMM

```
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: example

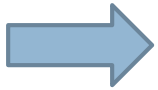
- The number of L/S instructions is reduced and as a consequence the number of memory accesses
- The number of arithmetical instructions is reduced too as there are fewer address computations for C[i][j] and A[i][k]
  - In the first case a different memory address is used for each load/store of A[][]
  - 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];
} }
```

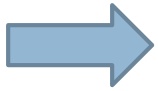


Step2

```
// 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: 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++) {
      …
     }
    }
```

References

- V. Kelefouras, Compilers for Embedded Systems
  - https://eclass.upatras.gr/modules/document/?course=EE738
- Renesas Electronics Corporation, Embedded C Programming III – Optimization
  - https://www.renesas.com/us/en/document/apn/embedded-programming-iii-ecprogramiiiopt