LABORATORY TUTORIAL #3

CODE TRANSFORMATIONS AND OPTIMIZATIONS

V1.2, Oct. 2024

Course: “Efficient Heterogeneous Computing”
Master Program in Informatics and Computing Engineering (M.EIC)

Faculty of Engineering of the University of Porto (FEUP), Porto, Portugal

GOALS:

- be aware of some compiler options and their impact on the execution time of applications

- experiment with compiler optimizations, such as auto-vectorization and loop tiling

REQUIRES:

- experiments on a desktop or laptop with C programs

- execution time measurements

PLANNED TIME:

2x2:00 (hours)

EXPERIMENTS

For the experiments to be done, note that tools such as the Godbolt compiler explorer [7][1] are very useful for checking the code generated by compilers when targeting CPUs.

AUTO-VECTORIZATION

Consider the codes in Figure 1 and use the Godbolt compiler explorer [1][7] to help to analyze the code generated, including their Control Flow Graphs (CFGs). Use -O3 and -O3 -march=native flags for compilation with a `gcc` version targeting x86-64 ISA and verify the differences.

Note: Information about `gcc` auto-vectorization is presented in [2].

```c
void muladdvec(float *A, float *B,
float *C, float *D, int N) {
    for(int i=0; i<N; i++) {
        D[i] = A[i]*B[i]+C[i];
    }
}
```
(a)

```c
void muladdvec(float *A, float *B,
float *C, float *D) {
    for(int i=0; i<1000; i++) {
        D[i] = A[i]*B[i]+C[i];
    }
}
```
(b)

```c
void muladdvec(float * __restrict A,
float * __restrict B, float *
__restrict C, float * __restrict D,
int M) {
    for(int i=0; i<M; i++) {
        D[i] = A[i]*B[i]+C[i];
    }
}
```
(c)

```c
void muladdvec(float * __restrict A,
float * __restrict B, float *
__restrict C, float * __restrict D) {
    for(int i=0; i<1000; i++) {
        D[i] = A[i]*B[i]+C[i];
    }
}
```
(d)

Figure 1. Versions of a vector dot product and add function.

Regarding the code generated when compiling with -O3 -march=native, fill Table I. Inspect the assembly code generated for the function and use compiler `gcc` flags to report the loops vectorized by the compiler.

Table I. Summary of the application of auto-vectorization to the code in Figure 1.

Version	Autovectorized (yes/no)?	If not, why?	Multiversioning (yes/no)?	If yes, why?
(a)				
(b)				
(c)				
(d)				

Now let us consider the codes in Figure 2. Do you expect the same generated code for the 4 different versions of the function? Use -O3 -march=native for compilation with a `gcc` version targeting x86-64 ISA and verify the differences.

```c
void mulvecsel(float __restrict  *A,
float __restrict  *B, float __restrict
*C, int *opt) {
  for(int i=0; i<1000; i++) {
    if(opt[i]) {
      C[i] = A[i]*B[i];
    }
  }
}
```
(a)

```c
void mulvecsel(float __restrict  *A,
float __restrict  *B, float __restrict
*C, int *opt) {
  for(int i=0; i<1000; i++) {
    if(opt[i]==1) {
      C[i] = A[i]*B[i];
    }
  }
}
```
(b)

```c
void mulvecsel(float __restrict  *A,
float __restrict  *B, float __restrict
*C, int *opt) {
  for(int i=0; i<1000; i++) {
    C[i] = opt[i]?A[i]*B[i]:C[i];
  }
}
```
(c)

```c
void mulvecsel(float __restrict  *A,
float __restrict  *B, float __restrict
*C, int *opt) {
  for(int i=0; i<1000; i++) {
    C[i] = (opt[i]==1)?A[i]*B[i]:C[i];
  }
}
```
(d)

Figure 2. Versions of a function that calculates the dot product of two vectors for the vector elements identified by the opt vector parameter.

Table II. Summary of the application of auto-vectorization to the code in Figure 2.

Version	Autovectorized (yes/no)?	If not, why?
(a)		
(b)		
(c)		
(d)		

OTHER COMPILER OPTIMIZATIONS

Consult the information about `gcc` optimizations [3][4] and describe each one of the compiler options presented in Table III.

Table III. Examples of gcc compiler optimizations.

Gcc optimization	Brief description	Possible parameters? (if yes, which ones?)	Identify in which optimization option, i.e., -=, -O1, -O2, -O3, -Ofast, -Ox, the optimization is included (e.g., check via gcc -Q --help=optimizers)
-floop-unroll-and-jam			
-ftree-loop-distribution			
-floop-interchange			
-funroll-loops			
-funroll-all-loops			
-ftree-loop-vectorize			
-fno-tree-loop-optimise			
-fmove-loop-invariants			
-ffast-math			
-funsafe-math-optimizations			
-fcrossjumping flag			
-fcse-follow-jumps			
-fguess-branch-probability			
-fno-guess-branch-probability			
-ftree-ccp			
-ftree-bit-ccp			
-finline-functions			
-fno-inline-functions			
-fipa-icf			
-fipa-vrp			
-fipa-cp			

LOOP INTERCHANGE AND LOOP TILING

Considering the code of a matrix multiplication function shown in Figure 3, do the following steps:

1. Integrate the code in a C program to test and evaluate the function and consider the initialization of the b and c matrices with random values.
2. Compile the code with gcc -O3 and measure its execution time.
3. Interchange loops i, j, k to have them in the order k, i, j and compile the resultant code with gcc -O3 and measure its execution time. Did you notice significant execution time differences with respect to the original version? If so, justify those differences.
4. Verify the impact when you compile the code version with the loops ordered as k, i, and j and with gcc -O3.
5. Apply loop tiling (also known as loop blocking) to the original code and analyze its impact on execution time.

```
#define N 1000
void matmul(float a[N][N], float b[N][N], float c[N][N], float d[N][N]) {
  int i,j,k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      for (k = 0; k < N; k++){
        a[i][j] += b[i][k]*c[k][j];
      }
    }
  }
}
```

Figure 3. Simple code for a matrix multiplication function.

ARRAYS OF STRUCTURES (AOS) TO STRUCTURES OF ARRAYS (SOA)

The code transformation from Arrays of Structures (AoS) to Structures of Arrays (SoA) is one important code transformation that may result in performance improvements, power dissipation and/or energy consumption reductions. Overall, the transformations may require substantial code changes, and the typical compilers do not include an automatic AoS to SoA in their optimizations portfolio.

1. Analyze the potential of AoS to SoA for the code of the project.
2. Apply AoS to SoA if you think this may improve execution time and reduce power dissipation and/or energy consumption.
3. Measure the execution time for the new code and compare it to the execution time of the baseline code. Calculate the speedup.
4. Measure the power dissipation and energy consumption for the new code and compare it to the execution time of the baseline code. Calculate the power and energy reductions.

Note: AoS to SoA is sometimes, and depending on the code, an AoS to Arrays, i.e., a transformation of arrays of structures to arrays.

REFERENCES

[1] Matt Godbolt, “Optimizations in C++ compilers,” in Commun. ACM 63, 2 (February 2020), 41–49. https://doi.org/10.1145/3369754

[2] “Auto-vectorization in GCC,” https://gcc.gnu.org/projects/tree-ssa/vectorization.html

[3] GNU gcc, “3.1 Option Summary,” https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

[4] GNU gcc “3.11 Options That Control Optimization,” https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[5] GNU gcc, “3.19.54 x86 Options,” https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html

[6] Intel Corp., “Intel® Architecture Instruction Set Extensions Programming Reference,” ID 790021, 2023-09-30. https://www.intel.com/content/www/us/en/content-details/790021/intel-architecture-instruction-set-extensions-programming-reference.html

USEFUL TOOLS

[7] Godbolt, M., “Compiler Explorer,” https://www.godbolt.org/ [online tool]