LABORATORY TUTORIAL #2 – PART-II

ROOFLINE MODEL

V1.0, 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

SUMMARY

In these lab experiments, we address typical analysis techniques to help you understand the performance of an input application, its main bottlenecks, and the potential for improvement.

MAIN GOALS

- analyze an input application, and given the characteristics of the target machine, build a Roofline model

- understand the memory-bound vs compute-bound regions and what may possibly make an application to move in the Roofline Model regions

- understand the importance of some loop optimizations for increasing performance and how they reflect in the Roofline Model

INVOLVES

- latency estimation of a region of code (kernel) based on simple machine models, and representing the kernel in a Roofline Model

- experiments using Intel Advisor [2] for building a Roofline Model

- experiments with a provided C program, the use of the GNU gcc compiler [1] considering a set of compiler optimizations and the impact seen on the Roofline model

PLANNED TIME:

Around 1 hour

EXPERIMENTS AND RESULTS

Let us consider a simple machine model with only one level of memory and with the cost per operation shown in Table I. When estimating the number of clock cycles to execute the code we will consider insignificant the cost for the other operations of the examples considered in the exercises. Consider that the target machine can provide **8 bytes/cycle** and **2 FLOPS/cycle**.

Table I. Cost of operations in the target machine.

Operation	Cost in #clock cycles (ccs)
Load (maximum of 64-bit)	1
Store (maximum of 64-bit)	1
double-precision (64-bit) arithmetic operations (e.g., add, sub, mul)	2
single-precision (32-bit) arithmetic operations (e.g., add, sub, mul)	1
double-precision (64-bit) mac (a × b + c)	3
single-precision (32-bit) mac (a × b + c)	1

Consider the two simple code examples presented in Figure 1 and Figure 2.

```
A, B, C: floating-point arrays
N = 2048

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

Figure 1. addvec example.

```
A, C: floating-point arrays
b,d: floating-point scalars
N = 2048

for (i = 0; i < N; i++) {
    C[i] = b*A[i] + d;
}
```

Figure 2. maddvec example.

ROOFLINE MODEL – W/O OPTIMIZATIONS

For estimating the number of clock cycles required to execute the examples, we simplify and use (1) for *addvec*, and (2) and (3) for *maddvec*. These simple analytical models of latency do not consider:

- software pipelining
- more than one load/store operation at the same time
- the cost of other operations involved such as the operations for calculating the memory addresses
- the cost of the operations associated with controlling the iterations of the loop

$$Latency\ (ccs) = N \times (2 \times Cost(Load) + Cost(add) + Cost(Store)) \quad (1)$$

$$Latency\ (ccs) = N \times (Cost(Load) + Cost(mac) + Cost(Store)) \quad (2)$$

$$Latency\ (ccs) = N \times (Cost(Load) + Cost(mul) + Cost(add) + Cost(Store)) \quad (3)$$

For the Roofline Models (see sketch in Figure 3) requested below, and based on estimations, consider for axis FLOPS/cycle and Bytes/cycle (i.e., without considering the clock frequency of the machine), and β = 2 cycles/byte and π=2 FLOPS/cycle.

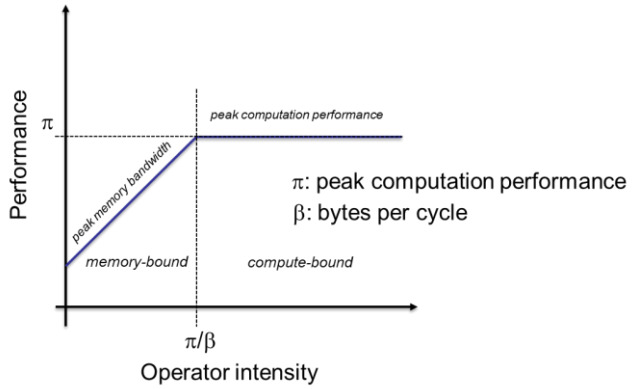

Figure 3. Sketch of a Roofline Model.

Based on this information and the two code examples, answer the following:

a) In terms of the Roofline Model region, where do you think each example is located (i.e., memory-bound region or compute-bound region)?
b) Draw the Roofline Model for the two code examples and consider *float* (single floating-point precision) and *double* (double floating-point precision) data type variants for each example (i.e., *addvec-float*, *addvec-double*, *maddvec-float*, and *maddvec-double*).
c) Include in the Roofline Model the *maddvec* examples when not using *mac* operations and, instead, the operations are done by a multiplication followed by an addition.

ROOFLINE MODEL – W/ OPTIMIZATIONS

We now analyze the impact of some optimizations:

A. Software pipelining, but without pipelining of multicycle operations and without considering simultaneous execution of similar operations (i.e., only one functional unit and only one load/store unit). In this case, each clock cycle, one load or one store can be executed in parallel with an addition.
B. Loop vectorization and use of SIMD[1] and/or FMA[2] units. In this case, consider that the machine can load/store 512-bit of contiguous memory data (vector length) in a clock cycle, includes two input vectors and one output vector, and the SIMD and FMA units are each one able to process the following number of operations simultaneously (note that a load operation is able to load memory data to an input vector, and one store operation is able to store the data in the output vector to the memory) with the number of clock cycles shown:

Unit	#float operations at the same time	#ccs w/ float	#double operations at the same time	#ccs w/ double
SIMD (multiple *add* and *mul* operations)	16	1	8	2
FMA (multiple *madd* operations)	16	1	8	3

Answer the following:

[1] SIMD: Single Instruction Multiple Data.
[2] FMA: Fused Multiply-Add.

d) Include in a Roofline Model the two code examples and consider *float* and *double* precision variants for each example (i.e., *addvec-float*, *addvec-double*, *maddvec-float*, and *maddvec-double*), and for each variant provide two subvariants: one with optimization A and another one with optimization B.

Use the same examples (code in Appendix below), N = 1048576, and *float* and *double* variants to provide a Roofline Model using Intel Advisor [2] (you can follow the main guidelines in [3]):

e) By compiling with gcc -O2 -march=native (e.g., gcc -O2 -march=native-DVARIANT=1 -DEXAMPLE=2 examples.c -o maddvec-O2-double);
f) By compiling with gcc -O2 -march=native and the use of the restrict keyword (e.g., gcc -O2 -march=native -DVARIANT=1 -DEXAMPLE=2 -DNONOVERLAP=1 examples.c -o maddvec-O2-double-vect);
g) By compiling with gcc -march=native -O2 -fopenmp, PARALLEL=1, and the use of the restrict keyword (e.g., gcc -O2 -march=native -fopenmp -DVARIANT=1 -DPARALLEL=1 -DEXAMPLE=2 examples.c -o maddvec-O2-double-vect-par).

REFERENCES

[1] "GCC, the GNU Compiler Collection," https://gcc.gnu.org/
[2] Intel Advisor, https://www.intel.com/content/www/us/en/developer/articles/tool/oneapi-standalone-components.html#advisor
[3] https://www.intel.com/content/www/us/en/developer/articles/technical/memory-level-roofline-model-with-advisor.html
[4] https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/analyze-cpu-roofline.html
[5] https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/run-cpu-roofline-perspective.html
[6] Explore CPU/Memory Roofline Results, https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/explore-cpu-memory-roofline-results.html
[7] CPU Roofline Report Overview, https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/cpu-roofline-report-overview.html
[8] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato and M. Püschel, "Applying the roofline model," 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 76-85, doi: 10.1109/ISPASS.2014.6844463.

APPENDIX

Code in file *example.c* for the Intel Advisor experiments:

```
#include <stdio.h>

#ifndef PARALLEL
#define PARALLEL 0
#endif

#if PARALLEL == 1
#include <omp.h>
#endif

#ifndef VECT
#define VECT 0 // y default no use of restrict
#endif
```

```
#if VECT == 1
#define NONOVERLAP
#else
#define NONOVERLAP __restrict
#endif

#ifndef EXAMPLE
#define EXAMPLE 1 // by default use addvec
#endif

#ifndef VARIANT
#define VARIANT 1 // by default use double precision
#endif

#if VARIANT == 1
    typedef double DATATYPE;
#elif VARIANT == 2
    typedef float DATATYPE;
#endif

#define N 1048576

DATATYPE A[N];
DATATYPE B[N];
DATATYPE C[N];

void addvec(DATATYPE * NONOVERLAP A, DATATYPE * NONOVERLAP B, DATATYPE *
NONOVERLAP C) {
      int i;
      #if PARALLEL == 1
      #pragma omp parallel for
      #endif
    for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}

void maddvec(DATATYPE * NONOVERLAP A, DATATYPE * NONOVERLAP C, DATATYPE b,
DATATYPE d) {
      int i;
      #if PARALLEL == 1
      #pragma omp parallel for
      #endif
    for (i = 0; i < N; i++) {
        C[i] = b*A[i] + d;
    }
}

int main() {
    printf("Executing example...\n");

    // put here initialization code if needed

#if EXAMPLE == 1
    addvec(A, B, C);
#elif EXAMPLE == 2
```

```
    DATATYPE b = (DATATYPE) 2.0;
    DATATYPE d = (DATATYPE) 3.0;
    maddvec(A, C, b, d);
#endif

    printf("Value C[%i] = %e\n", N / 2, C[N / 2]);

    return 0;
}
```