LABORATORY TUTORIAL #2

ANALYSIS AND PROFILING OF APPLICATIONS

V1.0, Sept. 2023; V1.1, 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, identify its main bottlenecks and potential for improvement

- understand some of the tools that can be used to support such analysis

- understand the importance of application profiling and the Roofline model

INVOLVES

- experiments with a provided C program, the use of the GNU **gcc** compiler [8] and GNU **gprof** [13] and **gcov** [14]

- experiments using Intel VTune Profiler [11]and Intel Advisor [12]

PLANNED TIME:

Around 2 hours

EXPERIMENTS AND RESULTS

For these experiments, we will use a simple C program that detects the edges of an input image[1]. The original program is from the UTDSP benchmark suite [18]. The version used here has been modified in terms of code structure and consists of 7 functions (2 of them to load/store image data from/to files).

PROFILING WITH GNU GPROF

GNU gprof is a simple profiling tool [13]. The options of the gprof tool can be seen using:

```
gprof –help or gprof –h
```

Compile the code for profiling but without any specific gcc flag optimization:

[1] Edge detection, https://en.wikipedia.org/wiki/Edge_detection

```
gcc -c -pg IO.c

gcc edge_detect.c IO.o -pg -o edge_detect
```

Execute the program: ./edge_detect.exe

Run gprof[2]: gprof ./edge_detect.exe

Observe the output of gprof, first the flat profiling and then the call graph. Note that gprof does not provide any call graph representation to be seen by a visualization tool; gprof2dot [17] is a script that can be used to generate a dot representation [16] for GraphViz [15] or other visualization tool that accepts graphs in dot format as input.

Table 1. Flat profiling for edge_detection with program compiled with default gcc options.

% time	Cumulative seconds	Self seconds	calls	Self ms/call	Total ms/call	Name

a) What is the total execution time reported by gprof?
b) Which is the main hotspot function in the code? How much time does it take to execute?
c) What will be the theoretical bound of the speedup if we have a computing unit that could execute the hotspot with negligible execution and communication time (consider 0 sec.)?
d) Draw the call graph and decorate it with the information you think can be used for analysis and decisions regarding improvements (e.g., via code optimizations and/or use of accelerators).
e) Draw a task graph for edge_detect and decorate it with the information you think can be used for analysis and decisions regarding improvements (e.g., via code optimizations and/or use of accelerators).
f) Compile the code with -O3 and repeat the steps and answers to the above questions.

Table 2. Flat profiling for edge_detection with program compiled with -O3 gcc.

% time	Cumulative seconds	Self seconds	Calls	Self ms/call	Total ms/call	Name

[2] Option --brief only outputs the flat profile and call graph data.

COVERAGE DATA WITH GNU GCOV

GNU gcov is a test coverage tool. Let's now use GNU gcov in order to understand more about the code.

Options of the gcov tool:

```
gcov –help or gcov -g
```

First compile for gcov:

```
gcc -c -coverage IO.c

gcc  edge_detect.c IO.o -coverage -o edge_detect
```

Then execute the program: ./edge_detect.exe

And finally run gcov: gcov ./edge_detect.exe

Check the output of gcov and see the reported information.

Use gcov for acquiring data regarding the number of times each statement (basic block) was executed for a single run of the program. Observe the results obtained using -b, -a, -f, individually and globally.

INTEL VTUNE PROFILER AND INTEL ADVISOR

The Intel VTune Profiler [11] and the Intel Advisor [12] are two very useful tools for analyzing and improving applications when targeting Intel machines. They have standalone versions, or they can be integrated in Microsoft Visual Studio [7]and used from the IDE. In these experiments we are going to explore these two tools and visualize the position in the Roofline model of the different versions of an application.

ANALYSIS OF EDGE_DETECT USING INTEL VTUNE PROFILER AND INTEL ADVISOR

a) Start by using VTune Profiler and observe the profiling information reported for the *edge_detect* program.
b) Then use Intel Advisor to collect useful information and/or optimization hints reported by the tool and the Roofline model for the *edge_detect* program.

IMPACT OF NUMBER OF THREADS

Let us now use once again the 2mm kernel from the PolyBench benchmark repository [3], and its OpenMP version [4] (used before in Lab #1). It includes 2 Matrix Multiplications (D=AxB; E=CxD) and computes the value of "alpha * A * B * C + beta * D". For these experiments, we will use the STANDARD_DATASET and the DATA_TYPE float.

In these experiments we intend to evaluate the impact of the number of threads on the performance of the program. For the available code version of *2mm*, the number of threads to be considered can be controlled at gcc compile time using -DNUM_THREADS=<number of threads>.

a) Collect results according to Table III below and conclude about the best options.
b) Use the Intel VTune Profiler to locate in the Roofline model the versions of 2mm according to the number of threads used.

Table III. Results including estimations for execution time, power dissipation and energy consumption.

Type of implementation	Option	GNU gcc compiler flags (add -march=native)	Execution time (s)	Power dissipation (W)	Energy consumption (J)	Speedup (vs. single-thread -O3)	Power dissipation reduction (%) (vs. single-thread -O3)	Energy consumption reduction (%) (vs. single-thread -O3)
Multi-thread	J1	-O3 -fopenmp -DNUM_THREADS=1				1	0%	0%
	J2	-O3 -fopenmp -DNUM_THREADS=2						
	J4	-O3 -fopenmp -DNUM_THREADS=4						
	J6	-O3 -fopenmp -DNUM_THREADS=6						
	J8	-O3 -fopenmp -DNUM_THREADS=8						
	J10	-O3 -fopenmp -DNUM_THREADS=10						
	J12	-O3 -fopenmp -DNUM_THREADS=12						

HOMEWORK

Use the Intel VTune Profiler and the Intel Advisor to understand more about the bottlenecks of the 2mm program and to compare via the Roofline model the object code generated by the different compiler options used (use the ones explored in the Lab. #1) and some variants with different number of threads.

Evaluate also the impact of the number of threads in power dissipation and energy consumption (e.g., using Intel PowerGadget) and compare it to the results you have obtained in Lab #1.

TOOLS AND BENCHMARKS USED

[1] Intel® Power Gadget, 2019. https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html

[2] GNU GCC compiler: "3.11 Options That Control Optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] Louis-Noel Pouchet, PolyBench/C, the Polyhedral Benchmark suite, © Ohio State University, USA. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[4] PolyBench-ACC, Copyright (c) 2012-2014 University of Delaware, USA. https://github.com/cavazos-lab/PolyBench-ACC

[5] "Powercfg command-line options." Microsoft. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/powercfg-command-line-options
[6] OPENMP API Specification: Version 5.0 November 2018, https://www.openmp.org/spec-html/5.0/openmp.html
[7] Microsoft Visual Studio 2022, https://learn.microsoft.com/en-us/cpp/?view=msvc-170
[8] "GCC, the GNU Compiler Collection," https://gcc.gnu.org/
[9] Compiler Explorer, https://godbolt.org/
[10] OneAPI (includes Intel Advisor, Intel Vtune Profiler, etc.), https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html
[11] Intel VTune Profiler, https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-download.html?operatingsystem=window
[12] Intel Advisor, https://www.intel.com/content/www/us/en/developer/articles/tool/oneapi-standalone-components.html#advisor
[13] Jay Fenlason and Richard Stallman, "GNU gprof: The GNU Profiler," https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
[14] "gcov—a Test Coverage Program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[15] "Graphviz," https://graphviz.org/
[16] "DOT Language," https://graphviz.org/doc/info/lang.html
[17] José Fonseca, "gprof2dot," https://github.com/jrfonseca/gprof2dot
[18] Corinna G. Lee, "UTDSP Benchmark Suite," University of Toronto, Canada. https://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html
[19] https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/analyze-cpu-roofline.html
[20] https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/run-cpu-roofline-perspective.html
[21] 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
[22] CPU Roofline Report Overview, https://www.intel.com/content/www/us/en/docs/advisor/user-guide/2023-0/cpu-roofline-report-overview.html

BIBLIOGRAPHY

[1] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz, Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation, Morgan Kaufmann (Elsevier), 2017. ISBN: 978-0-12-804189-5. https://doi.org/10.1016/C2015-0-00283-0 **[Chapter 2 and especially Section 2.6] Online resources: https://web.fe.up.pt/~jmpc/books/hpec/**