EEE3096S - Practical 1

2022

2 Emulation and Bench-marking

Note to the reader: You may be confused by the numbering system - just know that it is solely due to me not wanting to create separate LaTeX files. This is Practical 1 but it's the second overall assessment and consequently chapter in the complete manual which I am working in (this will be on Vula as soon as the details for all the assessments have been worked out). The rest of the tutorials and practicals will be numbered similarly. Just remember that even numbered assessments are practicals and odd ones are tutorials.

2.1 Overview

The completion of the pre-prac requirements are ABSOLUTELY REQUIRED for this practical. The practical is fairly straightforward, but there is a lot of information to take in and you will likely not be able to complete it correctly if you do not understand the concepts.

This practical is designed to teach you core concepts which are fundamental to developing embedded systems in industry. It will show you the importance of C or C++ for embedded systems development. We'll start by running a program in Python - which will be our 'Golden Measure' - and comparing its execution speed to the exact same program written in C++. This will show us how using different programming languages can impact the performance of the program. From there, we'll try and improve the performance of the C code as much as possible, by using different bit widths, compiler flags and threading.

The quest for optimisation can lead one down a long, unending spiral. What is important to take away from this practical is the awareness of optimisation concepts and the ability to use good practice when benchmarking.

There's a considerable flaw in this investigation in that there is no comparison of results to our golden measure. Meaning, by the end of the practical, you will (ideally) have considerably faster execution times - but your speed-up could be entirely useless if the result you're getting is not accurate. Comparison of accuracy is left as a task for bonus marks.

Previously, this practical would have been done on the Raspberry Pi. Since we don't have those, we will make use of Qemu. Like VirtualBox, Qemu (Quick EMUlator) is capable of vitualisation but it can also be used for emulation. Emulation is the process of creating a "guest" environment that emulates the a hardware or software system inside of a different "host" system. The "guest" code is not executed directly. Instead, an interpreter interprets the ”guest” code and executes equivalent operations on the ”host”. Emulators are commonly used to run video games whose hardware have become obsolete.

2.2 Pre-prac Requirements

2.2.1 Knowledge areas

This section covers what you will need to know before starting the practical. This content will be covered in the pre-prac videos.

- Introduction to benchmarking concepts
- Introduction to cache warming and good testing methodology (multiple runs, wall clock time, speed up)
- Instruction set architecture and bit widths
- Report writing

2.2.2 Videos

Some videos were made to assist students with the 2019 version of this practical. They'll still be quite helpful, so here they are:

- A quick intro to compilers, flags and makefiles
- An Introduction to Benchmarking
- Report Writing

2.3 Outcomes

By the end of this practical you will have an appreciation for the importance of benchmarking, lower level languages, ISA and integrated hardware.

- Benchmarking
- Latex and Overleaf
- Bit widths
- Compiler Flags
- Emulation
- Report Writing

2.4 Deliverables

At the end of this practical, you must:

- Submit a report no longer than 3 (three) pages detailing your investigation. You must use IEEE Conference style. You must cite relevant literature. Note that being able to convey your findings clearly and concisely is a skill and you may incur penalties if you go over the page limit

2.5 Hardware Required

- Linux machine or Windows/MacOS machine with VirtualBox installed

2.6 Walkthrough

2.6.1 Overview

1. Emulate Raspberry Pi using Qemu
2. Establish a golden measure in Python
3. Compare Python implementation to C++ implementation, in terms of accuracy and speed
4. Optimise the C++ code through parallelization and compare to golden measure
5. Optimise the C++ code through compiler flags and compare to golden measure
6. Optimise the C++ code through different bit widths, ensuring that the changes in accuracy are accounted for and compare to golden measure
7. Optimise the C++ code using a combination of parallelization, compiler flags, and bit widths and compare to golden measure

2.6.2 Detailed

1. Launch your Ubuntu VM and open up a terminal.
2. create a new directory called rpi and navigate into it:
   ```
   $ mkdir rpi && cd rpi
   ```
3. Install Qemu:
   ```
   $ sudo apt install qemu-system
   ```
4. Clone the Qemu Raspberry Pi kernel repo:
   ```
   $ git clone https://github.com/dhruvvyas90/qemu-rpi-kernel.git
   ```
5. Download the latest version of Raspbian (Raspberry Pi OS):
   ```
   $ wget https://downloads.raspberrypi.org/raspbian_latest
   ```
   (if this keeps failing, open the link in Mozilla Firefox)
6. Unzip the .zip folder you just downloaded to `~/rpi`. Rename the .img file to something simpler (like rpi.img).

7. Run the following command from within the `~/rpi` directory (Note that this is all one command and can be typed without the backslashes) :

```
$ qemu-system-arm \
-M versatilepb -cpu arm1176 \
-m 256 \
-kernel qemu-rpi-kernel/kernel-qemu-4.19.50-buster \
-hda rpi.img \
-append ”dwc_otg.lpm_enable=0 root=/dev/sda2 console=tty1 rootfstype=ext4
elevator=deadline rootwait” \
-dtb qemu-rpi-kernel/versatile-pb-buster.dtb \
-no-reboot \
-serial stdio
```

   The emulated Raspberry Pi should now start up.

8. (raspberrypi) Open up a terminal.

9. (raspberrypi) Start by getting the resources off of GitHub. You can clone it with:

```
$ sudo apt install git
$ git clone https://github.com/UCT-EE-OCW/EEE3096S-2022.git --depth=1
```

   This pulls all the code from EE3096S pracs (so far) into a folder called “EEE3096S-2022”.

10. (raspberrypi) Read the README in the WorkPackage1 directory. Here is a direct link.

11. (raspberrypi) Enter into the WorkPackage1 source file directory using the `cd` command. Enter the Python folder and run the Python code to establish a golden measure. Be sure to use proper testing methodology as explained in the pre-prac content.

```
$ cd ~/EEE3096S-2022/WorkPackage1/Python
$ python3 PythonHeterodyning.py
```

12. (raspberrypi) Now, navigate to the C folder and run the C code (don’t forget to compile it first, and every time you make a change to the source code!). This code has no optimisations in it, and also uses floats - just like the Python Implementation [2].

```
$ cd ../C
$ make
$ make run
```

13. (raspberrypi) How does the execution speed compare between Python and C when

[2] Floats in Python can get really weird - they don’t stick at a given 32 bits. But for the sake of this practical, we’re going to assume they do.

using floats of 64 bits? Record your results and comment on the differences.

14. (raspberrypi) Now let's optimise through using multi-threading.

    14.1 You can compile the threaded C version by running `make threaded` and run it using `make run_threaded`

    14.2 The number of threads is defined in `CHeterodyning_threaded.h`

    14.3 Run the code for 2 threads, 4 threads, 8 threads, 16 threads and 32 threads.

    14.4 Does the benchmark run faster every time? Record your results, and discuss the effects of threading in your report, making note of the number of cores on the Pi 0.

15. (raspberrypi) Record your results, taking note of the most performant one. What can you infer from the results?

16. (raspberrypi) Now let's optimise through some compiler flags

    16.1 Open the makefile, and in the `$(CFLAGS)` section, experiment with the following options:

Table I: Compiler Flags for optimisation

Flag	**Effect**
-O0	No optimisations, makes debugging logic easier. The default
-O1	Basic optimisations for speed and size, compiles a little slower but not much
-O2	More optimisations focused on speed
-O3	Many optimisations for speed. Compiled code may be larger than lower levels
-Ofast	Breaks a few rules to go much faster. Code might not behave as expected
-Os	Optimise for smaller compiled code size. Useful if you don't have much storage space
-Og	Optimise for debugging, with slower code
-funroll-loops	Can be added to any of the above, unrolls loops into repeated assembly in some cases to improve speed at cost of size

17. (raspberrypi) Record your results, taking note of the most performant one. Which combination of compiler flags offered the best speed up? Is it what you expected?

18. (raspberrypi) Now let's optimise using bit widths

    18.1 The standard code runs using **float**

    18.2 Start by finding out how many bits this is.

    18.3 Run the code using 3 bit-widths: double, float, and __fp16. How do they compare in terms of speed and accuracy? Note: for __fp16, you need to specify the flag "-mfp16-format=ieee" under your $CC flags in the makefile

19. (raspberrypi) Find the best combination of bit-width and compiler flags to give you the best possible speed up over your golden measure implementation. Bonus marks will be awarded for critical assessment of the performance and size using different combinations of bit widths and compiler flags.

20. (raspberrypi) Is there anything else you can think of that may increase performance? Perform your experiments and record your results. Bonus marks will be awarded for additional experimentation. (Hint: How can you test that the results of the Python code - our golden measure - and our fully optimised code produce the same results?)

21. Finally, record your methodology, results and conclusion in IEEE Conference format using Latex (Overleaf is recommended as an editor - See instructions here).

2.7 The Report

The report you need to complete needs to follow the IEEE Conference paper convention. You can find the conventions for this style online. It is recommended you use Overleaf as an editor, as it allows collaboration and handles all packages and the set up for you. In your report, you should cover the following:

- In your introduction, briefly discuss the objectives and core finding of your research
- In your methodology, discuss each experiment and how you plan to run it.
- In your results, record all your results in either tables or graphs. Be sure to include only what is relevant, but not miss out on anything that might be interesting. Remember to report on the **speedup** obtained and not just the execution speeds. Briefly justify why you got the results you did, and what each experiment did to affect the run time of your results.
- In your conclusion, mention what you did, and what your final findings are (e.g. "The program ran fastest when..."). This should only be a few sentences long.
- Look on Vula for report writing tips to avoid common mistakes and improve the readability of your report.

2.8 Marking

Your report will be marked according to the following:

- Following instructions
- Covering all aspects listed in this practical
- The quality of your report

Additional marks will be awarded for:

- Further experimentation
- Optimisation beyond what is described in this practical

Marks will be deducted for the following:

- Not using LaTeX
- IEEE Conference paper format not used
- Exceeding the page limit
- Incorrectly named submission