1 Prac 2 - Python vs. C

1.1 Overview

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, and comparing it to the exact same program, but written in C++. This shows us the importance of using a particular language. From there, we we'll try and improve the performance of the C code as much as possible, by using different bit widths, compiler flags, threading, and particular hardware available in the ARM Processor.

The quest for optimisation can lead one down a long, unending spiral. What is important to take away from this practical is awareness of concepts, 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.

1.2 Pre-prac Requirements

1.2.1 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, bit widths and hardware optimisations
- Makefiles
- Report writing

1.2.2 Videos

Some videos were made to help you with this practical. Here they are:

- An Introduction to Benchmarking
- Compilers, Toolchains and Makefiles
- Report Writing

1.2.3 Submissions for Pre-Prac

Before the practical begins, you need to submit a PDF on Vula of the IEEE report, with the Title, Authors and Introduction edited to reflect the details of this practical. A due date will be available on Vula.

1.3 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
- Instruction Sets
- Report Writing

1.4 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.

1.5 Hardware Required

- Raspberry Pi
- SD Card
- Ethernet Cable

1.6 Walkthrough

1.6.1 Overview

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

1.6.2 Detailed

1. Start by getting the resources off of GitHub

   ```
   $ cd ~/PracSource
   $ git pull origin master
   ```

   This updates the repository to ensure you have the Prac2 source files.

2. Read the README in the Prac2 directory. Here is a direct link.
3. Enter into the Prac 2 source files using the `cd` command. Run the Python code to establish a golden measure. Be sure to use proper testing methodology as explained in the pre-prac content.
4. Now, 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 [1].
5. How does the execution speed compare between Python and C when using floats of 64 bits? Record your results and comment on the differences.
6. Now let's optimise through using multi-threading.
   (a) You can compile the threaded version by running `make threaded`
   (b) The number of threads is defined in `Prac2_threaded.h`
   (c) Run the code for 2 threads, 4 threads, 8 threads, 16 threads and 32 threads.

[1] 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.

(d) Does the benchmark run faster every time? Record your results, and discuss the effects of threading in your report.

7. Record your results, taking note of the most performant one. What can you infer from the results?

8. Now let's optimise through some compiler flags

   (a) 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 dont 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

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

10. Now let's optimise using bit widths

    (a) The standard code runs using **float**

    (b) Start by finding out how many bits this is.

    (c) 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

11. Now let's optimise using hardware level support on the Raspberry Pi

    (a) The Pi has various instruction sets. See what they are by running "cat /proc/cpuinfo" in a terminal

    (b) You can set a floating point hardware accelerator with the "-mfpu=" flag

    (c) The important ones for you to try are:

Table II: Compiler Flags for the Floating Point Unit

mfpu Flag	**Description**
none specified	Default implementation
vfpv3	Version 3 of the floating point unit
vfpv3-fp16	Equivalent to VFPv3 but adds hfp16 support
fpv4	Version 4 of the floating point unit
neon-fp-armv8	Advanced SIMD with Floating point
neon-fp16	Advanced SIMD with support for half-precision
vfpv3xd	Single Precision floating point
vfpv3xd-fp16	Single precision floating point, plus support for fp16

12. Find the best combination of bit-width and compiler flags to give you the best possible speed up over your golden measure implementation.

13. 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?)

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

1.7 The Report

The report you need to complete needs to follow the IEEE Conference paper convention. This style you can find online. It is recommended you use Overleaf as an editor, as it allows collaboration and handles all packages and 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. 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.

1.8 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