Performance Comparison and Analysis of Various C++ Optimisation Techniques for Embedded Systems Software Development

Hendri Vermeulen[†] and Kyle du Plessis[‡]
EEE3095S Class of 2019
University of Cape Town
South Africa
[†]VRMHEN004 [‡]DPLKYL002

***Abstract*—This paper firstly devises and structures a comparison of a Python program implementation to a C++ program implementation in terms of run-time speed and accuracy. It then demonstrates the optimisation of the run-time performance of the C++ code through the use of various C++ optimisation techniques which include parallelisation, compiler flags, using different bit widths and hardware-level features available on the Raspberry Pi. Finally, it takes a step toward the optimisation of the C++ code using a combination of parallelisation, compiler flags, using different bit widths and hardware-level features. Principally, we showed that (1) a Python program is usually much less efficient than the C/C++ equivalent and (2) the optimised C++ program implementation run with the 32 threads, 16 bit width, -O2 -funroll-loops compiler flag option and neon-fp16 mfpu flag option resulted in the lowest average run-time speed - this particular combination of optimisation techniques gives the best possible speed up over the golden measure implementation.**

I. Introduction

The C/C++ programming languages largely predominates the embedded systems space as it creates more compact, faster and efficient run-time code [1]. The C/C++ languages allows for direct low-level machine hardware control and manipulation, and is able to utilise the machine hardware to its full potential. Thus when it comes to speed and efficiency, it has become the language of choice for the vast majority of today's embedded system code [2]. Despite introducing substantial run-time performance overhead, Python is well-known for its ease-of-programming, error reduction as well as readability and this makes development much more efficient. For this reason, Python is currently becoming an increasingly popular choice for embedded systems development [3]. Different programming languages support different optimisation features and it can be proven that the implementation of these features also have a significant effect on program speed-up. It is evident that there is an increasing push for parallelism in computer architecture due to the Power Wall, the Memory Wall and the ILP (Instruction Level Parallelism) Wall. This makes multi-threading more relevant than ever and one of the foremost methods of improving processor performance [4]. Using particular hardware available in the ARM Processor allows specific hardware-level optimisations to be applied to increase program speed. It also becomes important to consider the bit width used to represent variables, as shortening bit widths leads to faster execution but less precision. Since C/C++ are compiled languages, it has a lot of compiler optimisation techniques built-in which may also further increase program run-time performance. Hence through the use of these different optimisation techniques, either together or on their own, an increase in program speed and perhaps accuracy can be achieved. It is essential to explore the effect of implementing these different optimisation techniques when developing embedded systems, especially if they are meant for real-time or time-critical applications.

II. Methodology

A. Hardware

The entire experiment had been run on a Raspberry Pi 3 B+ Model which uses a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz CPU, has 1GB LPDDR2 SDRAM. The Raspberry Pi runs Raspbian Strect (08/04/2019) Operating System on a Sandisk 16GB SD Card and makes use of an ethernet cable to connected to a PC.

B. Implementation

Cache warming involves running a program a few times to ensure the program and its data is cached. This ensures all program run-time speed measurements are in the same uncertainty and that there are no outliers. A bash script used to employ proper testing methodology had been created. It firstly warms up the cache and then records 10 run-time speed measurements of the C++ program implementations. It then saves the result into a csv file. This had been slightly modified and adapted for the Python program implementation. The average of the 10 run-time speed measurements had been computed to best represent run-time performance. Furthermore, only run-time speed measurements of the critical sections of the programs had been recorded. Bash script for cache warming and recording run-time speed measurements:

```
echo "Warming up"
i=1
while [ $i -le 300 ]
```

```
do
        make run > /dev/null
        ((i++))
done
i=1
echo "Running Tests"
while [ $i -le 10 ]
do
        make run | grep -oP '\d+.+d* +ms' | sed 's/ ms/, /g' >> Output.csv
        ((i++))
done
```

The following source code was used for the golden measure Python program implementation:

```
def main():
    print("using_type_{}".format(type(data[0])))
    Timing.startlog()
    for i in range(len(c)):
        result.append(c[i] * d[i])
    Timing.endlog()
```

The following source code was used for the unoptimised C++ program implementation:

```
extern float data [SAMPLE_COUNT];
extern float carrier[SAMPLE_COUNT];

float result [SAMPLE_COUNT];

int main(int argc, char**argv){
    printf("Running Unthreaded Test\n");
    printf("Precision sizeof %d\n", sizeof(float));

    tic(); // start the timer
    for (int i = 0;i<SAMPLE_COUNT;i++ ){
        result[i] = data[i] * carrier[i];
    }
    double t = toc();
    printf("Time: %lf ms\n",t/1e-3);
    printf("End Unthreaded Test\n");
    return 0;
}
```

The following source code was used for the multi-threaded-optimised C++ program implementation:

```
    tic();
    for(j = 0; j < Thread_Count; j++){
        if(pthread_join(Thread[j], 0)){
            pthread_mutex_lock(&Mutex);
            printf("Problem joining thread %d\n", j);
            pthread_mutex_unlock(&Mutex);
        }
    }
    printf("All threads have quit\n");
    printf("Time taken for threads to run = %lg ms\n", toc()/1e-3);
```

C. Experiment Procedure

The experiment had been carried out in six parts. The first part of the experiment involved running and comparing the Python program implementation to the unoptimised C++ program implementation in terms of run-time speed. In this part of the experiment, the golden measure is the Python program implementation. Results had then been recorded. This experiment was done to prove that a Python program is usually much less efficient than the C++ equivalent. For the next few parts of the experiment which involved optimising the C++ program implementation using various techniques, the unoptimised C++ program implementation used in the C++ vs Python part of the experiment will be used as the golden measure. The second part of the experiment involved optimising the C++ program implementation through multi-threading. The C++ multi-threaded program implementation had been run with 2, 4, 8, 16 and 32 threads respectively. Results had then been recorded. This experiment was done to prove that parallelisation and the use of more threads increases program run-time performance up to a certain extent. The third part of the experiment involved optimising the C++ program implementation through various compiler flags. The C++ program implementation had been run with -O0, -O1, -O2, -O3, -Ofast, -Os, -Og and -funroll-loops compiler flag options respectively. Results had then been recorded. This experiment was done to prove that the use of different compiler flags increases program run-time performance. The fourth part of the experiment involved optimising the C++ program implementation through various bit widths. The C++ program implementation had been run with 64 bit, 32 bit and 16 bit bit-widths respectively. Results had then been recorded. This experiment was done to prove that shortening bit widths leads to faster program execution but less precision. The fifth part of the experiment involved optimising the C++ program implementation through hardware level support on the Raspberry Pi. The C++ program implementation had been run with none specified, vfpv3, vfpv3-fp16, vfpv4, neon-fp-armv8, neon-fp16, vfpv3xd and vfpv3xd-fp16 mfpu flag options respectively. Results had then been recorded. This experiment was done to prove that the use of different mfpu flags (hardware acceleration) increases program run-time performance. The last part of the experiment involved optimising the C++ program implementation through a combination of parallelisation, compiler flags, using different bit widths and hardware-level features. The C++ program implementation had been run using combinations of optimisation techniques. Results had then been recorded. This experiment was done to prove that there is a particular combination of optimisation techniques which gives the best possible speed up over the golden measure implementation. It had been noted that of the two Raspberry Pis available used to run the experiment and record results - one gives significantly slower run-time speed measurements for all parts of the experiment using the exact same testing methodology, benchmarking and experimental setup. This might be due to a hardware issue, corrupted installation of Raspbian or some other unforeseen issue. The experiment had thus been conducted and results recorded using the faster Raspberry Pi.

III. Results

The golden measure Python program implementation resulted in an average run-time speed of 5.9226575ms as shown in table I, while the unoptimised C++ program implementation resulted in a significantly lower average run-time speed of 3.1605231ms as shown in table II. It can be seen that the Python program implementation is much less efficient than the C++ equivalent due to C++ being a compiled language. The multi-threaded-optimised C++ program implementation run with 32 threads resulted in the lowest average run-time speed of 1.1105879ms as shown in table III. It can be seen that as the thread count increases, the average run-time speed decreases but not very significantly. Thread count 4 and 8 as well as 16 and 32 showed very similar average run-time speeds which proves that the use of more threads increases program run-time performance up to a certain extent as there is performance

overhead associated with creating and merging threads. The compiler-flag-optimised C++ program implementation run with the -02 and -funroll-loops compiler flag options resulted in the lowest average run-time speed of 1.0425951ms as shown in table IV. It can be seen that the use of different compiler flags increases program run-time performance, as the average run-time speeds decreases significantly. The -funroll-loops compiler flag option always brings additional average run-time speedup when combined with other compiler flag options. The bit-width-optimised C++ program implementation run with the 32 bit width resulted in the lowest average run-time speed of 3.1433322ms as shown in table V. This had most likely to do with the CPU's architecture being optimised for 32 bit processes. The 16 bit width was expected to have the lowest average run-time speed, as shortening bit widths leads to faster program execution but less precision. However, this was not the case as seen from the results as shown in table V. The hardware-level-feature-optimised C++ program implementation run with the vfpv4 mfpu flag option resulted in the lowest average run-time speed of 3.0120218ms as shown in table VI. It can be seen that the use of different mfpu flags (hardware acceleration) does not increase program run-time performance since all the average run-time speeds are very similar to each other and the golden measure. The combination-of-optimisation-techniques-optimised C++ program implementation run with the 32 threads, 16 bit width, -O2 -funroll-loops compiler flag option and neon-fp16 mfpu flag option resulted in the lowest average run-time speed of 0.5117221ms as shown in table VII.This proves that this particular combination of optimisation techniques gives the best possible speed up over the golden measure implementation. Here it can be seen that the 16 bit width optimisation does lead to a small program run-time performance increase when using it with other optimisations that support it.

IV. Conclusion

Experimentally, we found that (1) a Python program is usually much less efficient than the C/C++ equivalent and (2) the optimised C++ program implementation run with the 32 threads, 16 bit width, -O2 -funroll-loops compiler flag option and neon-fp16 mfpu flag option resulted in the lowest average run-time speed - this particular combination of optimisation techniques gives the best possible speed up over the golden measure implementation.

References

[1] T. Radcliffe, "Python vs. c/c in embedded systems - dzone iot," Sep 2018. [Online]. Available: https://dzone.com/articles/python-vs-cc-in-embedded-systems

[2] W. G. Wong, "Is c the best embedded programming language?" Feb 2018. [Online]. Available: https://www.electronicdesign.com/embedded-revolution/c-best-embedded-programming-language

[3] "The rise of python for embedded systems continues," Dec 2018. [Online]. Available: https://www.zerynth.com/blog/the-rise-of-python-for-embedded-systems-continues/

[4] L. Jelenkovi and G. Omren-eko, "Experiments with multithreading in parallel computing." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.7775&rep=rep1&type=pdf

TABLE I

Golden measure Python program implementation results

Run-time speed measurements in seconds
5.904379
5.823565
5.973295
5.929214
5.926879
5.929543
5.877178
5.891979
6.032198
5.938345

TABLE II

Unoptimised C++ program implementation results

Run-time speed measurements in ms
3.1647
3.257824
3.131888
3.162617
3.109701
3.100847
3.217981
3.085743
3.164439
3.209491

TABLE III

Multi-threaded-optimised C++ program implementation results

Thread Count	Avg run-time speed (ms)
2	2.752265
4	2.039762
8	2.049099
16	1.307292
32	1.1105879

TABLE IV

Compiler-flag-optimised C++ program implementation results

Compiler Flag	Avg run-time speed (ms)
-O0	3.1433322
-O1	1.3250988
-O2	1.2995908
-O3	1.3125397
-Ofast	1.3109214
-Os	1.3250535
-Og	2.3877138
-O0 -funroll-loops	3.1100176
-O1 -funroll-loops	1.2737851
-02 -funroll-loops	1.0425951
-O3 -funroll-loops	1.0518987
-Ofast -funroll-loops	1.0533573
-Os -funroll-loops	1.3205563

TABLE V

BIT-WIDTH-OPTIMISED C++ PROGRAM IMPLEMENTATION RESULTS

Bit Width	Avg run-time speed (ms)
16 bit (__fp16)	8.543334
32 bit (float)	3.1433322
64 bit (double)	3.621875

TABLE VI

HARDWARE-LEVEL-FEATURE-OPTIMISED C++ PROGRAM IMPLEMENTATION RESULTS

mfpu Flag	Bit Width	Avg run-time speed (ms)
None	32 bit	3.1433322
vfpv3	32 bit	3.1233128
vfpv3-fp16	32 bit	3.1608857
vfpv4	32 bit	3.0120218
neon-fp-armv8	32 bit	3.1074637
neon-fp16	32 bit	3.1379743
vfpv3xd	32 bit	3.1704848
vfpv3xd-fp16	32 bit	3.1277139
vfpv4	16 bit	3.0120218
vfpv3-fp16	16 bit	3.3209741
neon-fp16	16 bit	3.24774
vfpv3xd-fp16	16 bit	3.3214638

TABLE VII

COMBINATION-OF-OPTIMISATION-TECHNIQUES-OPTIMISED C++ PROGRAM IMPLEMENTATION RESULTS

Thread Count and Flag(s)	Bit Width	Avg run-time speed (ms)
16 Threads -O2	32 bit	0.7330316
32 Threads -O2	32 bit	0.6416856
16 Threads -O2 -funroll-loops	32 bit	0.6145778
32 Threads -O2 -funroll-loops	32 bit	0.5369961
32 Threads -O2 -funroll-loops	16 bit	0.8852878
32 Threads -O2 -funroll-loops vfpv4	32 bit	0.5320755
32 Threads -O2 -funroll-loops vfpv3-fp16	16 bit	0.5558464
32 Threads -O2 -funroll-loops neon-fp16	16 bit	0.5117221
1 Thread -O2 -funroll-loops neon-fp16	16 bit	0.798233