Exploring the Impact of Programming Languages and Optimisation Techniques on Embedded System Performance Using a Python and C++ Benchmarking Study

Thiyashan Pillay[†] and Danté Webber[‡]
EEE3096F Class of 2023
University of Cape Town
South Africa
[†]PLLTHI032 [‡]WBBDAN003

***Abstract*—This report represents a detailed investigation into the effect of programming languages and optimisation techniques on the performance of embedded systems. The objective of this study is to compare the execution speeds of the program coded in Python, which will be the "golden measure", and coded in C++ using various enhancements. The results will be used to evaluate execution speed of the two languages, as well as the trade-off between speed and accuracy. The practical exercise involves benchmarking while experimenting with different bit widths, compiler flags, and parallelisation.**

I. Introduction

Welcome to the world of embedded systems, where milliseconds define success. This study dives into the effects that coding languages, efficient code, and automatic code optimisation have on benchmarking.

The objective of this experiment is to:

- Compare the execution speed of the same program implemented in Python, the "golden measure" and C++.
- Explore different optimisation techniques to enhance the performance of a program in C++ and compare execution times to the "golden measure".
- Evaluate the trade-off between speed and accuracy.

Several factors contribute to reduced benchmark test performance:

- Hardware limitations: Based on the two different PCs used.
- Thermal throttling: Overheating of the hardware components does lead to reduced benchmarking performances.
- Background processes: Applications running in the background consume resources, leading to reduced benchmark performance.
- Power settings: The machine might be equipped with power saving settings to conserve energy, limiting resources
- Virtualisation: Running a benchmark in a virtual machine introduces added performance overhead
- Caching: reduces benchmark performance by hiding the true execution times through the reuse of data that was accessed previously.

With regards to bit widths, accuracy and speed varies from 16-, 32-, 64- bits.

- 16-bit (half precision): (Accuracy) Provides a limited range of precision in comparison to the other options. Not suitable for tasks of higher accuracy. (Performance) Fastest due to smaller data size.
- 32-bit (single precision): (Accuracy) Reasonable accuracy for most applications. The balance between precision and performance. (Performance) Faster than 64 bits, but slower than 16 bits.
- 64-bit (double precision): (Accuracy) Provides the best accuracy. Best in situations where precise numerical results are needed. (Performance) Generally slower than lower precision formats due to greater data size.

With regards to parallelisation, A gradual speedup as the number of threads increased from 2 to 32 threads is expected. There are two big problems that lead to a deviation from this expected result.

- diminishing returns: When adding more threads doesn't yield the same performance gain. The performance per thread decreases as a result of the overhead linked to managing and coordinating the threads.
- Potential for bottlenecks: This is the point where the performance is limited by the hardware. Adding moe threads will not yield a better performance.

With regards to compiler flags, Compiler flags are expected to get faster as the order is increased (from -O0 to -Og), but this is not guaranteed. There is an intricate interplay between the compiler optimisation and code construction. There may be faster execution times with one flag, but there is a possibility that unintended side effects get introduced leading to inaccuracies.

II. Methodology

To begin the study, a new virtual machine using VirtualBox was set up on Thiyashan's computer (Machine 1), with Linux (Ubuntu) as a guest operating system. On Dante's computer (machine 2), running Linux (Ubuntu) as the main OS, had no need for a virtual machine. On both systems, Qemu was installed, and thereafter Raspberry Pi OS was loaded on. The Git repository containing benchmark code was cloned into the designated folder within Qemu.

For both programming languages (Python and C++), the cache on Qemu was 'warmed up' by running each program between 15 and 20 times before the first recorded execution. After the 'warm up', each program was run 10 times, and the time taken for each execution was recorded. When running computer programs, the speed at which they are executed can vary greatly based on external factors presented in the introduction of this report. These real-world factors vary themselves and hence the inaccuracy they produce in the results is largely unpredictable. Therefore, to ensure integrity of results, the average over the 10 results for each program was taken. Furthermore, outliers in the result dataset are extremely common, especially when the C program is parallelised. This is because the program has no control over when the threads that it creates will be executed, once the threads are passed to the OS's event scheduler, the scheduler could allow another process to be executed before the program, and the timer coded into the program could end up timing this unwanted process in the middle. Hence the outliers in the dataset are most likely somewhat corrupted representations of the execution time of the program. To account for this error, the median of the results is also taken, as taking the median of a dataset removes dependence on outliers in the data.

A. Establishing the golden measure with Python,

The study began by creating the "golden measure" using Python, a benchmark against which the successive optimisations will be compared to. This is the baseline for performance comparison.

B. Repetitive refinement of the C++ implementation,

With the "golden measure" established, the C++ implementation is refined iteratively. The first test was run using floats, the same as the Python implementation. This is a direct comparison of execution times between the two programming languages.

C. Effect of bit widths on precision and performance,

Different bit widths were explored – float (32 bits), double (64 bits), and 16 bits – in the C++ program equivalent. The speeds will be compared, and the code evaluated to see which bit width would be best.

D. The power of compiler flags,

A range of flags were tested – -O0 to -Og – as well as adding -funroll-loops with the fast flag in the range to improve speed at cost of size. The impact each flag has on the execution speed is analysed.

E. Multithreading for parallel execution,

By varying the thread count – 2, 4, 8, 16, 32 threads – the influence this has on the execution speed is then evaluated.

F. Combining bit widths, compiler flags, and parallelization,

The pinnacle of this study involves combining the insights gained in previous stages. The best of the three optimisations is combined to reveal the ultimate performance enhancement.

G. Accuracy assessment and validation,

Producing an impressive speed up alone in the results does not support a valid argument without analysis of the accuracy of the results. To test the accuracy of the benchmark tests, a program will be coded to compare the result arrays of each benchmark test relative to the "Golden Measure" result array. The benchmark test populates 2 arrays and then multiplies each corresponding element together and places the results in a result array.

III. Results and Discussion

The 2 machines used have distinct hardware configurations for running the benchmarks. Directly comparing the results between the disparate machines may yield a false insight due to the inherent machine-specific nature of benchmarking. Instead, our focus is identifying prominent trends within the shared dataset, transcending the variating that is induced by the different setups. The findings from the experiment serve as proof to reveal interesting trends in the different optimisations. Upon closer examination of the benchmarking code files, it is evident that there are large arrays involved. Accelerated processing times are desired, so it is important to remain mindful of potential payoffs that may be expected. Faster execution speeds could indirectly entail the sacrifice of data fidelity during the optimisation process. It is essential to strike an equilibrium between performance and to keep the integrity of the data. See table I below for a summary of the results,

Test Type	**Machine 1**	**Machine 2**
	Speed-Up (Med)	**Speed-Up (Med)**
Python	1	1
C-default (Float)	25.938	37.222
C (64-bit float)	14.411	15.541
C (16-bit float)	5.565	5.810
C-default (O0)	25.851	41.820
C-default (O1)	37.058	51.066
C-default (O2)	53.488	51.172
C-default (O3)	28.778	50.479
C-default (Ofast)	32.504	52.242
C-default (Os)	54.317	50.511
C-default (Og)	21.250	46.555
C-Threaded (2)	19.383	26.075
C-Threaded (4)	109.928	30.227
C-Threaded (8)	88.162	87.019
C-Threaded (16)	44.997	136.181
C-Threaded (32)	29.335	96.023

TABLE I: Table showing the speed-up of the median of each data-set relative to the python benchmark code. Take from the day which showed the most consistent results.

A. The baseline for comparison,

The Python “golden measure” exhibited slower execution speeds than the C++ code implementation. This discrepancy is expected, given the underlying dissimilarities in execution models, run-time environments, and levels of optimisations between the two programming languages. C++ boasts a more robust memory management that minimises the effect of the run-time overhead. Upon comparing Python and C++ utilising a 64-bit framework, the data in table 1 shows that the default (non-optimised) C code has between 25.9 and 37.2 speed-up.

B. Bit widths,

Floats displayed the swiftest execution times. This does not agree with the research that was presented in the introduction showing the comparison between different bit widths (float, double, 16-bit). The research suggested that 16-bit should have bee the fastest in terms of execution speeds, while 64 bit should have been the most accurate. In the test, the 16-bit proved to have the worst speed-up of around 5.5, compared with the 32-bit speed-up of 30 and 64-bit around 15. These results seem to agree with the idea that 32-bit floats are the optimal bit-width for this benchmark test.

C. Compiler flags,

On the two days of the test, a different compiler flag was fastest. Both the -Os and -Ofast times were very similar on both days for both machines. The fastest time yielded from compiler flags alone on the Virtual Machine (VM) was with the -Os flag, with a median value of 4.468ms on day 2. Whereas on machine 2 (Non-VM), -Ofast was the fastest, at 8.077ms. Considering the approach of each flag, -Os seems to be the best option. With -Ofast, while the execution times are said to be a lot quicker, it may be too aggressive. Many corners are cut in a bid to yield the fastest times. -Os on the other hand is focused more on minimising the code size of the compiled program. Reduced memory usage and better cache performance is prioritised. This makes it suitable for embedded systems, where memory and storage space are limited.

Aside from the -Os and -Ofast compiler flags, all of the other flags responded quite erratically, and hence the data is not worth analysing in depth.

D. Multithreading (also known as parallelization),

On both days, each machine followed its own trend. For machine 1 (VM), the speed-up gets progressively better, starting at 2 threads and reaches its peak at 16 threads, after which the speed-up drops quite dramatically for 32 threads. On machine 2 the results were less predictable although the speed up seemed to increase until maximising at 8 threads, after which it dropped down. Based on our knowledge, we were expecting for there to be a gradual speed-up as the number of threads increased from 2, until reaching the number of threads available for use on the machine. However, the data does not seem to support this entirely, instead of machine 1 (VM) having the highest speed-up with 8 threads, the highest speed-up was with 16 threads. And machine 2 should have performed best at 4 threads, but instead performed best with 8 threads. While this is not as expected, it seems to follow a common trend, that double the number of available threads yields the highest speed-up.

E. Combined optimisation strategy,

This analysis was performed on the final day of testing, when the best possible combination was realised. The method used was described previously. The optimal combinations for machine 1 are:

- Bit width: Float (32-bit)
- Thread count: 16 threads
- Compiler flag: -Os

The optimal combination for machine 2

- Bit width: Float (32-bit)
- Thread count: 8 threads
- Compiler flag: -Os

On some days, perhaps another optimisation type would seem faster, but the combinations listed for each machine is the one that struck the best balance between speed and accuracy.

F. Accuracy assessment,

The accuracy test program was run on the highest optimised C program and the Python benchmark script. The results showed that the only differences in the results were rounding differences in the 6th decimal place. This is because the python does not use a float (32-bit) sized variables in the calculations, but much larger sized variable, and the Accuracy Test program rounded off the values from the Python code to compare them with the C code float variables which caused some rounding errors in the output. The code used is available here: https://gitlab.com/bright-ideas/eee3096s.

G. Future considerations,

- Improved algorithm: Looking at alternative algorithms may lead to better benchmarking performance while maintaining accuracy.
- Hardware specific optimisations: better optimisations tailor suited to the hardware of the machine can be explored to unlock more performance.

IV. Conclusion

The study of optimisation and benchmarking has revealed a dynamic interplay between the execution speed and accuracy. From different programming languages, bit widths, compiler flags and multithreading, the study shows the intricate trade-offs for gains in performance. The Python “golden measure” set the benchmark for the other optimisations to be compared to. C++ was significantly faster with the same code implementation. Floats emerged victorious amongst the choices for the best bit widths. Compiler flags introduced variability across both machines, with -Os being the best choice in the end. Multithreading was an interesting one, with the best performing thread count being the one that was double the thread count of the CPU of the machine. The combined strategy achieved at the end of the study is a harmonious blend of speed and accuracy.