EEE3096S 2023 Practical 1

1st Travimadox Webb
Department of Electrical Engineering
University of Cape Town
Cape Town, South Africa
WBBTRA001

2nd Rumbidzai Mashumba
Department of Electrical Engineering
University of Cape Town
Cape Town, South Africa
MSHRUM006

***Abstract*—This practical critically evaluates the performance differential between C and Python programming languages within the context of embedded systems. Through the application of parallelization, compiler flags, bit widths, and specific hardware characteristics of the ARM processor, a comprehensive benchmarking analysis of the two languages was conducted. The data obtained reinforces the distinct efficiency advantages of C in contrast to Python, particularly concerning the speed-up ratio. This underscores the significance of strategic code optimization in embedded system design.**

***Index Terms*—Embedded Systems, Qemu, Emulation, Benchmarking, C, Python, Compiler Flags, Bit Widths, Parallelization, ARM Processor.**

I. Introduction

In this practical, a rigorous examination of performance dynamics between C and Python languages is undertaken, emphasizing their roles in embedded systems development. The study commenced with the deployment of a Python-oriented program, identified as the 'Golden Measure', and its subsequent performance metrics were contrasted against a parallel C implementation. Thereafter, the C code was subjected to an in-depth optimization process, encompassing various bit widths, compiler flags, inherent hardware features of the ARM processor, and threading methodologies. While the path to optimization appears infinite, its exploration remains paramount. The findings from this study not only provide a tangible performance differentiation between the two languages but also clearly highlight the enhanced speed-up ratio of C relative to Python.

II. Methodology

A. Experimental Setup

Our experimental process was anchored on a Qemu-emulated Raspberry Pi environment, instantiated on an Ubuntu Virtual Machine. To maintain consistency throughout our testing process, we fetched the necessary resources from a specified GitHub repository.

B. Establishing the Golden Measure

We earmarked a Python program to be our 'Golden Measure.' The execution time and output parameters of this program were diligently recorded to serve as our benchmark.

C. Benchmarking

An equivalent program, but written in C, was compiled, and executed. The primary objective of this step was to assess the performance of the C program, especially its speed and accuracy when utilizing 64-bit floats, in juxtaposition with the established Python golden measure.

D. Parallelisation Analysis

With the intent of discerning the role of threading in enhancing program efficiency, we transitioned to a threaded C program variant. Its execution was replicated across a spectrum of thread counts: 2, 4, 8, 16, and 32. We meticulously documented the performance variances triggered by these diverse thread counts.

E. Compiler Flags Investigation

Our methodology employed a succession of compiler flags, which included -O0, -O1, -O2, -O3, -Ofast,-Os, and -Og, with an ancillary flag (-funroll-loops). For each flag, we scrutinized its influence on the program's speed and overall behavior.

F. Bit Widths Assessment

An integral component of our experiment involved evaluating the C code's performance across three distinct bit-width configurations: double, float, and fp16. We documented the emergent variations in terms of speed and precision.

G. Combinatorial Optimization

Our final phase championed a holistic approach, wherein we synergistically fused different bit-widths and compiler flags. The objective was to pinpoint the optimal configuration that yielded the most pronounced speedup in relation to the golden measure.

III. Results and Discussion

A. Execution Time Comparison

The primary metric for comparison was the execution time of the programs written in Python and C. Both programs were executed on two separate machines, with the results averaged over multiple runs to ensure accuracy.The table below summarizes the average execution times:

From the data, it is evident that:

TABLE I
MACHINE PERFORMANCE

Language	Machine 1 (seconds)	Machine 2 (seconds)
Python	0.2854	0.3014
C	0.00936	0.01649

1) The C program outperforms its Python counterpart on both machines.
2) The C program's execution on Machine 1 is approximately 30 times faster than the Python version.
3) On Machine 2, the C program exhibits an approximate 18 times speed advantage over the Python version.

This initial analysis establishes a clear performance disparity between the two languages, with C exhibiting a notable advantage in execution speed.

B. Performance Optimization Through Multithreading in C

Utilizing multi-threading capabilities offers a pathway to optimize the execution speed of programs. This section focuses on understanding the impact of multi-threading on the performance of the C program. The graph below depicts the speed-up of the program with different thread counts

Fig. 1. Speed-up graph comparing the execution times of the C++ program with varying thread counts across the two machines using 0 threads as baseline

1) Discussion: From the results:

1) Thread Count Influence: The program's performance improves as the number of threads increases from 2 to 8. Beyond this point, the gains plateau, suggesting an optimal level of parallelism for this specific workload.
2) Optimal Thread Count: The best performance across both machines is observed with 8 threads. This indicates efficient workload distribution without the overheads associated with higher thread counts.
3) Performance Ceiling: Increasing the thread count beyond 8 does not provide significant performance benefits and can even degrade performance slightly. This could be attributed to the overhead of managing more threads or hardware constraints.
4) Consistency Across Machines: The trends are consistent across both test machines, underlining the general applicability of the observations.

C. Compiler Flag Optimization Analysis

Compiler flags offer a suite of optimization techniques that can substantially influence a program's execution speed. In this section, we explore the impact of various compiler flags on the performance of our C program.

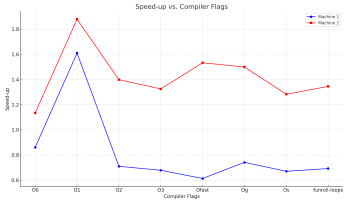

Fig. 2. Speed-up graph comparing the execution times of the C++ program with varying compiler optimization flags across two machines using O0 as reference

From our experiments, the O1 optimization flag consistently delivered the best performance on both machines. This level of optimization strikes a balance, eliminating code inefficiencies without introducing the complexities and potential pitfalls of higher optimization levels. On the other hand, O2 and O3, despite being more aggressive optimization levels, didn't yield a better performance than O1. This highlights the importance of tailoring compiler optimizations to the specific nature and structure of the code in question. The Ofast flag, which prioritizes speed over standard compliance, showcased intriguing results. While it outperformed other flags on Machine 2, its benefits were less evident on Machine 1. Such disparities emphasize the role of hardware and system characteristics in determining optimization outcomes.

D. Impact of Bitwidth on Execution Time

Bitwidth optimization, an essential technique in computational efficiency, entails adjusting the precision of calculations. Different bitwidths can have varied impacts on both the execution speed and accuracy of a program. In this section, we investigate the performance of our C program when using three distinct bitwidths: float32 bits, double 64 bits, and floating point 16 bits (fp16).

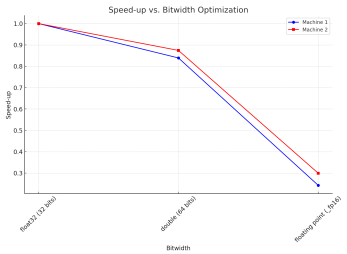

Fig. 3. Graph depicting the speed-up associated with each bitwidth optimization, using the fastest setting (float32, 32 bits) as the reference.

From the data, it's evident that bitwidth settings significantly influence the execution speed:

1) Float32 (32 bits): This bitwidth yielded the most efficient results, with the shortest execution times on both machines. This could be attributed to the reduced precision, which allows for faster computations.
2) Double (64 bits): With a larger bitwidth and increased precision, execution times were moderately longer than the 32-bit counterpart. The enhanced precision may

introduce additional computational overhead, leading to slightly prolonged execution times.

3) Floating Point (_fp16, 16 bits): Surprisingly, this setting resulted in the longest execution times, considerably underperforming compared to the 32-bit and 64-bit configurations. The reduced precision of 16 bits, while expected to speed up computations, might be introducing other computational challenges or inefficiencies in the execution environment.

While the 32-bit configuration offers the best speed, it's essential to consider the trade-off between speed and precision. A reduced bitwidth can accelerate computations but might sacrifice accuracy. Conversely, a larger bitwidth, while ensuring greater precision, can be more computationally intensive.

E. Bit Accuracy and Compiler Optimization for Improved Performance

In the field of computational efficiency, both the accuracy of calculations, indicated by the number of bits used, and the optimization techniques used by compilers, are crucial. These two elements, when effectively combined, can significantly boost performance. In this section, we explore how the number of bits and compiler optimizations affect the execution speed of our C program.

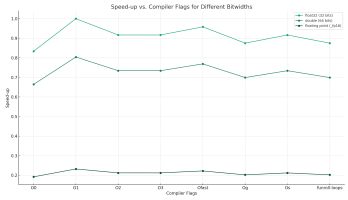

Fig. 4. Speed-up graph comparing the execution times of the C program with varying compiler flags across different bitwidth optimizations using the fastest combination (float32 (32 bits) with O1 compiler flag) as the reference.

From the results:

1) Performance with `float32 (32 bits)` & `O1`: This setup resulted in the fastest execution time. The `O1` flag, which is known for providing a balance between various optimizations, in conjunction with 32-bit precision, achieved a good balance between computational speed and accuracy.
2) Effect of Bit Accuracy on Compiler Optimizations: The influence of compiler optimizations was notably higher with `float32 (32 bits)` compared to other precisions. This may indicate that the program is particularly suited for optimizations at this accuracy level.
3) Issues with `_fp16`: Even though `floating point (_fp16)` has less precision, which often means faster computations, it showed extended execution times across all optimization levels. This could be due to challenges or inefficiencies when operating at this level of precision.

The combination of bit accuracy and compiler optimizations provides a significant potential for improving performance. While both factors are influential on their own, their joint effect can result in notable performance differences. For instance, the pairing of `float32 (32 bits)` & `O1` showcased optimum results. However, it's important to note that the best settings can vary based on the specific program. Detailed testing and continuous adjustments are essential to determine the ideal setup for each application.

IV. Conclusion

In this report, we thoroughly compared the performance of Python and C programming implementations, focusing on optimization techniques within the C environment. Our methodical study covered variations in the number of bits used for calculations, the complexities of running multiple threads simultaneously, and the combined effect of compiler settings. The results clearly showed that the best performance was achieved when the program was coded using a 32-bit format, specifically 'float32 (32 bits)', and optimized with the 'O1' compiler setting. This key finding highlights the importance of careful optimization, emphasizing the need to find the right balance between computational speed and the accuracy of the results.

Appendix

Supplementary Experimental Data

For those seeking a deeper examination of our findings, the raw experimental data utilized throughout this practical is available. This data serves as a comprehensive foundation for our conclusions and is provided to encourage transparency and further exploration. It is accessible It is accessible here.

Bibliography

References

[1] Y. Martin, *Emulation and Benchmarking*, EEE3096S Practical 1 2023, University of Cape Town, Cape Town, 2023. [Online]. Available: https://amathuba.uct.ac.za/d2l/le/lessons/14467/topics/1277489. [Accessed: 08 Aug 2023].