EEE3096S Practical 2

David Da Costa[†] and Theodore Psillos[‡]
EEE3096S Class of 2021
University of Cape Town
South Africa
[†]DCSDAV001 [‡]PSLTHE001

***Abstract*—This report consists of an assessment of a heterodyning algorithm efficiency when utilizing various methods of optimization. It comprises of a comparison of the Python and C form of the same algorithm, a comparison of bit size and memory allocation's effect on efficiency, assessments of the speedup of threaded forms of the algorithm as well as an assessment on the effect of various compiler flags on time efficiency.**

I. Introduction

This practical is an assessment of of execution speed of a program when utilizing different variations of primitive data types, sequential execution, parallel execution and different forms of compiler flags. The assessment will be performed on the efficiency of a heterodyning algorithm. The primitive data types experimented on are in the form of float, double and fp16. A float is a single precision 32- bit floating point where a double is a double precision 64 bit floating point and fp16 is a 16 bit floating point, [1]. Within the practical is an assessment of parallel execution speed. Parallel computing is a form of computing where the task at hand is split up into multiple threads across different cores. Parallel computing if implemented correctly results in a speed up of execution time of the algorithm. The final tests are done on different compiler flags. Compiler flags are options passed to the compiler. Different compiler flags offer different advantages and disadvantages to the compilation and execution efficiency and accuracy. The compiler flags used in this practical as well as their benefits and losses are summarized in this link.

II. Methodology

In this section you should describe the method of the experiment.

A. Hardware

The hardware used in this practical is a single core PiZero with specs shown in the following website.

B. Implementation

The source code used in the practical can be found here. The changes made to the original code were adjustments to the end time of the threaded algorithm, changes to the makefile, changes of integer size in the CHeterodyning and globals as well as the number of threads in CHeterodyningThreaded. the reasoning for these changes are discussed below.

Makefile Changes:
The makefile was adjusted in order to change the compilation flags for the various experiments. It was also adjusted in order to allow for 16 bit floating points. These changes resulted in different run/compilation time which will be used as variables during experimentation

Integer Size Changes:
The size of the memory dedicated to each number was changed in order to see execution time changes depending on memory size allocation.

Changes to end time of CHeterodyningThreaded:
One of the key factors influencing the outcome of this practical is fair, valuable and reliable results to timing experiments as explained in the bench marking Video. After assessing the given CHeterodyningThreaded algorithm there were many oddities found. The first issue found was that the end timer occurred after the final print statements. This would result in incorrect timing of the parallel method as the print statements were not part of the core functionality of the code which should be timed. In order to fix this issue the end time was changed to be called before the print statement. The following is the original relevant code followed by the changed/current code:

Old code:

```
printf("All threads have quit\n");
printf("Time taken for threads to run = %lg ms\n", toc()/1e-3);
```

Current code

```
double x=toc()/1e-3;
printf("All threads have quit\n");
printf("Time taken for threads to run = %lg ms\n", x);

Other relevant changes/ additions to code:
```

A Python Automation program was also created in order to simplify the process of executing the program multiple times:

```
import  os
for i in range(15):
        os.system("python3  PythonHeterodyning.py >>outputTime.txt")
```

The rest of the changes include changing floats to double and fp16 as well as adding compiler flags to the makefile. These changes were small enough to exclude from this report but can be found here.

C. Experiment Procedure

The goal of this experiment was to obtain reliable timings of the various forms of the algorithm for comparison. In order to do so one has to set up a benchmark time. This was done by timing the python heterodyning algorithm. This time was used as a comparison factor. Following this was a time experiment on bit sizes. This consists of timings done

on the C algorithm for bit sizes of 16, 32 and 64 bits. Then a comparison of the parallel threaded algorithm was done. This was conducted with threads of sizes 2, 4, 8, 16, 32. This was followed by a comparison of compiler flag combinations. The flags used were O0, O1, O2, O3, Ofast, Os, Og, funroll -loops. All timings were done by taking the average of 5 reliable executions. In order to get a reliable execution each algorithm was executed a minimum of 10 times for cache warming before the times were taken.

III. Results and Discussion

The Python Code was run to determine its execution speed. This speed, which is referred to throughout the practical as the Golden Measure, was 529.88 ms.

A. Figures

Figure 1 shows that an increase in the number of threads will not necessarily decrease the execution time of the algorithm. This can be explained by the overhead as result of multiple threading. The most optimal thread number was 16 having the fastest execution time of 1.5 ms. This execution time of the golden measure, 529.88 ms, pales in comparison to the fastest execution time as the execution time is decreased by more than 300%. The Pi Zero has a single core as shown at the following website. The speedup ratio between the fastest sequential no flags algorithm(9.4ms) and fastest parallel algorithm(1.5ms) is 6.267. This speedup ratio is much higher than the expected speedup as the ideal possible speedup is the sequential speed divided by the number of hyper threads. In the case of the pi-Zero there is only a single core which means that this speedup ratio is practically impossible. After further assessment we noticed that the timing and threading of the threaded algorithm was somewhat deceiving and could be considered incorrect. The program does not properly time the actual run time of the algorithm. Please see III-C for a more thorough discussion.

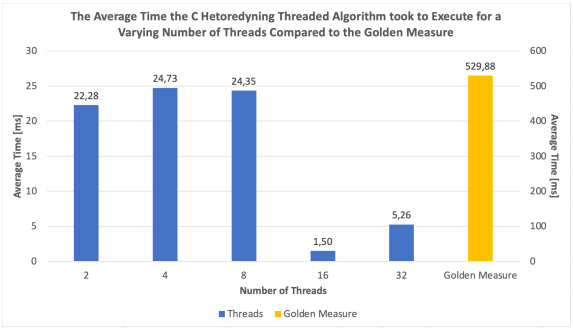

Fig. 1: Optimisation of the C code through Parallelization

Figure 2 represents the execution time of different combinations of compiler flags for a bit width of 32 bits. Up to 20 combinations of all 8 compiler flags were tried and tested, with the top 5 fastest execution times plotted on figure 2. Combination that produced the fastest execution time of 6.08 ms was O1 O2 O3 Ofast. All top 5, and as a matter of fact all 20 combinations, decreased the execution time by at most 90% and at least 50%. Although the Ofast compilation flag increases the chance of the code to behave unexpectedly, the fastest combination was very accurate as all values lay within one standard deviation of the average value.

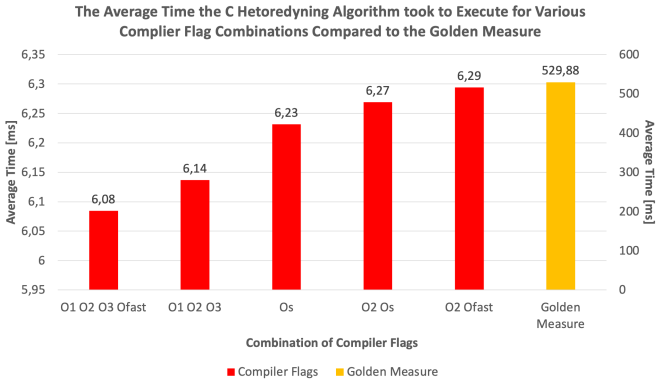

Fig. 2: Optimisation of the C code through Compiler Flags

Figure 3 depicts the execution time of different bit widths. The bit width defines how much memory is allocated for each execution. The PiZero has 32 bit registers and so the bit width will determine the number of registers in memory required for execution. All width bits had faster execution times than the golden measure. However, increasing or decreasing the bit width from 32 bits to either 16 or 64 bits did not necessarily lead to a decreased execution time. When the 64 bit width was used, more memory registers were required for execution leading to a slower execution time. On the other hand, a bit width of 16 bits did not result in a decrease in execution time either as 32 bit registers were used to compile 2 sets of __16bit code. The float bit width was the most accurate will the __16bit showed some inconsistencies as values varied making the average execution time much higher.

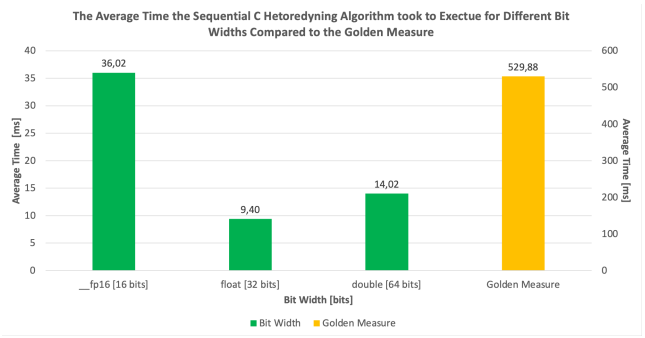

Fig. 3: Optimisation of the C code through Different Bit Widths

Figure 4 compares all the combinations of the best performing optimisations. Evidently the use of 16 threads was produced the fastest execution time of 1.5 ms. As mentioned in the previous figures all optimisations where faster than the golden measure.

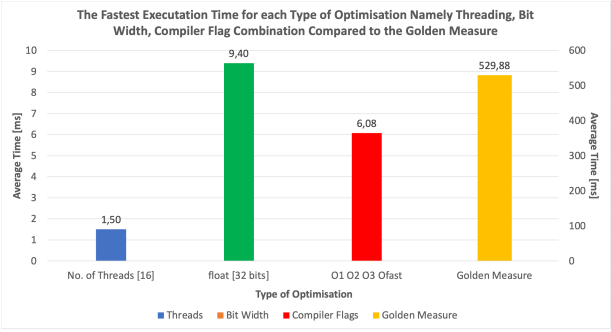

Fig. 4: Optimisation of the C code using a Combination of Parallelization, Compiler Flags, and Bit Widths

B. Tables

For simplicity sake, all tables relevant to this practical may be found here. A representation of the speed up ratio of the benchmark and the best performing optimisations are presented in tables I, II and III

TABLE I

Speed up Ratio of Parallelization Optimisation	
Number of Threads	**Speed Up Ratio**
2	0.42
4	0.38
8	0.39
16	6.27
32	1.79

The speed up ratio of the threading optimisation in table I is compared to the fastest sequential execution time (namely float). As discovered in graph 1, 16 threads produces the highest expected speed up ratio of 6.27. This alone is incredibly high for a single core processor.

TABLE II

Speed up Ratio through Compiler Flag Optimisation	
Compiler Flag Combination	**Speed Up Ratio**
O1 O2 O3 Ofast	87.15
O1 O2 O3	86.30
Os	85.05
O2 Os	84.51
O2 Ofast	84.24

In table II the speed up times were much higher as they were compared to the golden measure. As mentioned above, Python has a far higher execution time than C, regardless whether the C code is optimised or not. As a result, it is natural to expect that the addition of different combinations of compiler flags would have high speed up ratios. The highest being the combination of O1 O2 O3 Ofast producing a ratio of 87.15. This combination did have a slightly slower compile time but it the execution time is what was important.

TABLE III

Speed up through Bit Width Optimisation	
Bit Width	**Speed Up Ratio**
__16bit [16 bits]	14.71
float [32 bits]	56.37
double [64 bits]	37.79

Lastly, table III shows that for a bit width of 32 bits the speed up ratio was 56.37. Significantly higher than the other bit widths. This was expected as a 32 bit float has the same length as the memory register as of the PiZero meaning that when execution occurs no time is wasted fitting 64 bits or 16 bits into a 32 bit register.

C. Inefficiencies

Parallel timing: After further analysis of CHeterodyningThreaded one can see the placement of tic and toc is incorrect. If one looks at the simplified and shortened version of the main method of the parallel algorithm one can see that the timer starts at the beginning of the main method, then the timer is restarted again after a large piece of code was executed and then the timer ends at the end of the main method. By restarting the timer the the actual time of the algorithm is not accounted for but rather only a small portion of the algorithm is timed. This completely removes any correctness or reliability of the timing of the parallel algorithm. This is the provided code for the practical and was not meant to be changed. This algorithm provided firstly does not properly time the actual run time of the algorithm and secondly does not properly implement parallel programming as there is only a single core on the pi therefore no threads can run in parallel.

```
int main(int argc, char** argv){
 //some code
 tic();
 //removed a large amount of code including a for loop for simplicity sake
//removed a few print statements
tic();
//removed alot of code including a for loop
double x=toc()/1e-3;
printf("All threads have quit\n");
printf("Time taken for threads to run = %lg ms\n", x);
 return 0;
}
```

IV. Conclusion

The unoptimised C code executed significantly faster than the python code. The base C code had an execution time of 9.40 ms while the golden measure was 529.88 ms. After various different optimisation methods which included parrallelisation through threading, different compiler flag combinations and different bit width lengths, the fastest execution time was using parralelisation of 16 threads. It had an execution time of 1.5 ms. Additionally, it had a 6.27 speed up ratio to the sequential C implementation. However, it must be noted that dispareties were observed in that for a single core processor, such as the PiZero, it is impossible for multiple threads to run. As such improvements could be made to the code used in this practical to ensure more accurate and reliable results. Nevertheless, it proved that regardless the optimisation method applied to the C code, its execution time was significantly faster than the python code and is thus far superior when wanting to optimise your code.

References

[1] J. Cook, “Comparing bfloat16 Range and Precision to Other 16-bit Numbers,” Nov. 2018.