The first implementation of an optimized HFT algorithm on an ARM chip

Supervisor: Dr. Ruibang Luo

Siddhant Ningoo

18\textsuperscript{th} April 2023
# Table of Contents

Abstract .............................................................................................................................................. 4

1.  Introduction & Background ........................................................................................................ 5
    The current situation .................................................................................................................. 5
    The advancement in technology ............................................................................................. 5
    The gap ....................................................................................................................................... 6

2.  Objective ..................................................................................................................................... 7

3.  Project Overview ....................................................................................................................... 8
    Part 1: Literature review and research on ARM architecture .................................................. 8
    Part 2: Implementation .............................................................................................................. 8
    Part 3: Appraisal of performance ............................................................................................ 9

4.  The architecture ....................................................................................................................... 10
    4.1 RISC vs CISC .................................................................................................................... 10
    4.2 ARM Architecture specific optimization ............................................................................ 12
        4.2.1 SIMD ....................................................................................................................... 12
        4.2.2 NEON ....................................................................................................................... 12

5.  Vectorization ............................................................................................................................ 13
    5.1 Auto-vectorization ............................................................................................................... 13
    5.2 NEON Intrinsics .................................................................................................................. 18

6.  Project Formulation & Experimenting .................................................................................... 20
    6.1 Project Formulation .............................................................................................................. 20
    6.2 Experimentation ................................................................................................................... 21

7.  Implementation .......................................................................................................................... 22
7.1 Black Scholes Model ...........................................................................................................23
7.2 Monte Carlo Simulation Code ...........................................................................................27
7.3 Pricing Derivatives .............................................................................................................31
7.4 Results ...............................................................................................................................32
   Black Scholes Model...........................................................................................................32
   Monte Carlo Simulation model..........................................................................................34
   Analysis on x86 ...................................................................................................................36
8. Evaluation & Recommendations .........................................................................................37
   8.1 Project Evaluation ..........................................................................................................37
   8.2 Future Work & Recommendation ..................................................................................38
References ...............................................................................................................................39
Abstract

With the advancement in ARM based architecture, there has been an increase in the prominence of NEON instruction set based chips in more and more commercial solutions. This project looks toward leveraging ARM based chips’ potential to parallelize computing in the form of vectorization, specific to the High Frequency Trading (HFT) field where high-performance computing is pivotal to successful implementation of HFT algorithms. While the majority of the results are pointing towards ARM based vectorization bringing out better performance and lower latency, more research is required in this area to be 100% certain. However, in order to capitalize on the first-mover advantage it is recommended that low-latency based trading shops focus on developing their ARM based architecture and look to further capitalize on the RISC benefits.
1. Introduction & Background

Technology, specifically Algorithmic Trading (AT), has revolutionized the way people trade in the markets, with it now accounting for 60-73% of overall US equity trading as per a market research by Mordor Intelligence LLP (2022), with them predicting a further surge of 10.5% over the coming 5 years. High Frequency Trading (HFT) is a form of algorithmic trading that leverages high computational speeds to automate establishment and liquidation of securities in a small-time frame (SEC, 2010).

The current situation

It is estimated that as of 2020, about 50% of the trading volume in US Equity Markets is generated from HFT (Breckenfelder, 2020). HFT is a form of AT that heavily relies on low-latency and the efficiency of the algorithms, in order to achieve high speed and high frequency trading. Apart from the hawkish regulators, firms engaging in HFT need to constantly be aware of the competition from their competitors in the race to produce an algorithm that can be faster by milliseconds ("An Introduction To The HFT Industry And Its Key Players", n.d.). Buchanan (2015) mentions that “as technology advances, trading speed is increasingly limited only by fundamental physics.”. However, De Prado (2018) has outlined in his book that while many investment managers think complex machine learning algorithms is key to success for lower latency, it is in fact a mixture of multiple factors, ranging from enhanced data manipulation, vectorization, multiprocessing, etc., which often most firms engaged in HFT fail to leverage.

The advancement in technology

Spanning across decades, x86-64 was the predominant architecture used in PCs while the ARM based architecture was mainly used to power smartphones. However, in recent development, ARM based architecture with the NEON instruction set has been entering the PC space and even the cloud computing space with the likes of Apple M series chips.
and AWS’s AWS Graviton2 respectively. It has been claimed that customers of AWS (such as Snap, Twitter, Netflix, etc.) have seen improved performance and lowered costs after switching to the ARM based architecture ("AWS & ARM - Partners - ARM", n.d.). Similarly, there are many studies done to ascertain the positioning of ARM processors in this space and many come to a similar conclusion of ARM performing better energy-to-solution (Gupta & Sharma, 2021; Mantovani et al., 2020; Criado et al., 2020) While Mantovani et al. (2020) even go on to mention ARM based chips could power the next generation of High Performance Computing (HPC) systems.

The gap

When looking at the type of computer architecture on which HFT algorithms run on, the x86-64 based processor segment had the highest revenue share and is estimated to further grow it’s foothold in this space from 2021 to 2028 (Bloomberg, 2021). Coming from the positive signs of using an ARM based architecture in HPCs to the wide praise it has received, it would hence be worth exploring how HFT algorithms would perform on an ARM based chip, especially when firms are looking in every direction to shed a few milliseconds. Despite it’s efficient architecture, there is a current issue of the algorithms still being not as efficient and utilizing the ARM architecture to the maximum.
2. Objective

While there are tons of research done on ARM based HPC, for instance Project Mont Blanc (n.d.), Fugaku - another ARM based supercomputer, there is no research on optimizing and evaluating HFT algorithms specific to an ARM chip, hence this project aims to fill this gap to optimize & analyze algorithms for ARM based CPUs for industrial purposes (eg. Apple M series chips or AWS Graviton2)

Owing to the productivity of the newer ARM based chips, it is worthwhile to study the performance of HFT algorithms on the newer ARM based chips, with algorithms and the AI models further streamlined to best optimize the total output capabilities of the newer ARM chips. With a focus on Vectorization, the objective of this project is to make such modifications to the HFT models to better fit the ARM processor which in turn makes the algorithm more efficient, in.

To truly assess if it has any impact and if these modifications bring about deviant performance changes this project proposes to analyze the performance of the ARM based algorithm in comparison to pre-existing models running on x86-64 processors:

1. it’s latency in order placement

2. the total portfolio performance

From this, the main objectives of the project are to:

- Provide the first fintech application of HFT that leverages the new ARM based architecture to its fullest
- Evaluate and provide recommendations to corporates and funds that engage in HFT to enhance their algorithms’ performance for higher efficiency and lower latency.
3. Project Overview

The project will be divided in roughly 3 parts, with Part 1 focusing on research of the ARM architecture, Part 2 focusing on the implementation and Part 3 focusing on the analysis.

Part 1: Literature review and research on ARM architecture

In order to better optimize the models it is imperative to understand the ARMv8-A architecture, the NEON instruction set, based on a Reduced Instruction Set Computer (RISC), while the current implementations of x86-64 are all based on a Complex Instruction Set Computer (CISC) and how the NEON instruction set can be utilized to it’s fullest. The literature review will pave a way to the project objective to make the optimization more unique to ARM chips.

Part 2: Implementation

As mentioned, there are several open-source pre-existing HFT models, this project will use these open source projects as the starting point and will build from there. Many of such projects are written in either Python or C++, however, NEON architecture provides vectorizing compiler support for C/C++ to enable higher levels of parallelism for higher performance, therefore C++ would be used for the optimization.

The primary focus of the optimization will be on Vectorizing the existing code, where the operation will be applied at once to the entire data. Vectorization entails the replacement of all the explicit iterators (ie. for loops) with more flexible solutions such as matrix algebra operations, iterators or generators. With vectorization, the process could be parallelized with faster running times, potentially leading to faster HFT algorithms.
Part 3: Appraisal of performance

As outlined in the objectives, following the complete optimization of the HFT algorithms, including the vectorization, and other HFT model enhancements, a performance review would be carried out.

The first method of reviewing the performance involves comparing the latency of order placement between the ARM chips and the x86-64 chips, where even a difference of milliseconds could make a difference, hence this being a very important test to review.

The second method of reviewing the performance of the optimized model is by evaluating the entire portfolio performance. The HFT algorithm will be connected with a paper trading account, where no real money is at stake, but all other factors emulate the real-world performance, and hence observing real-time changes made to the model. Here the Trader WorkStation (TWS) API from a brokerage firm, Interactive Brokers, that allows paper trading will be used to relay the automated trades to be placed and executed, additionally it provides access to market data, current portfolio, and trades.

From these metrics, there would be a clearer picture of how effective the ARM architecture is to process optimized & parallelized HFT algorithms.
4. The architecture

4.1 RISC vs CISC

All ARM chips are based on a RISC, while all the x86-64 based chips utilize CISC. When coming up with CISC a common philosophy was to write an assembly code that is as short as possible. The CISC processor has several “complex instructions” to choose from to run a piece of code, while the RISC has fewer, more “simple instructions” that it can utilize. Accordingly, due to a smaller instruction set to use from, RISC processors often require less memory for storing the instruction set. Following that, RISC processors have more memory allocated for general purpose registers, and its more Complex counterpart will have less memory for general purpose registers, and often times RISC assembly code did tend to use more memory than CISC. Back then, cost of memory used to be very high, hence it was not reasonable enough to be using RISC. However, in the current age, the cost of memory has dropped significantly hence it is worth exploring RISC based processors, which thus explains a shift of focus from x86 processors to ARM processors.

Another key feature which could be one of the main reasons why RISC processors are staring to make a comeback, and can be seen in HPC is the fact that most of the instructions on a RISC processor only take one clock cycle to be executed, while CISC instructions could vary in their execution time, and thus allowing for pipelining in RISC. For instance, a program to multiply 2 integers on a CISC processor would decompose it to just one instruction, while a RISC would decompose it into 4 instructions, however, both of them would execute the multiplication program in approximately the same time.

Below is an example that we can look at for comparing how a high level implemented C++ code would be compiled into for a RISC and a CISC based architecture.

C++ code for finding the cube for a number passed to the function
The first implementation of an optimized HFT algorithm on an ARM chip

```c
int cube(int num) {
    return num*num*num;
}

int main(int argc, char** argv) {
    return cube(argc);
}
```

**Code Snippet 1: Cube Calculator**

<table>
<thead>
<tr>
<th>ARMv8-a clang 15.0.0</th>
<th>X86-64 clang 15.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cube(int):</td>
<td>1 cube(int):</td>
</tr>
<tr>
<td>2 sub sp, sp, #16</td>
<td>2 push rbp</td>
</tr>
<tr>
<td>3 str w0, [sp, #12]</td>
<td>3 mov rbp, rsp</td>
</tr>
<tr>
<td>4 ldr w8, [sp, #12]</td>
<td>4 mov dword ptr [rbp - 4], edi</td>
</tr>
<tr>
<td>5 ldr w9, [sp, #12]</td>
<td>5 mov eax, dword ptr [rbp - 4]</td>
</tr>
<tr>
<td>6 mul w8, w8, w9</td>
<td>6 imul eax, dword ptr [rbp - 4]</td>
</tr>
<tr>
<td>7 ldr w9, [sp, #12]</td>
<td>7 imul eax, dword ptr [rbp - 4]</td>
</tr>
<tr>
<td>8 mul w0, w8, w9</td>
<td>8 pop rbp</td>
</tr>
<tr>
<td>9 add sp, sp, #16</td>
<td>9 ret</td>
</tr>
<tr>
<td>10 ret</td>
<td></td>
</tr>
<tr>
<td>11 main:</td>
<td>11 push rbp</td>
</tr>
<tr>
<td>12 sub sp, sp, #32</td>
<td>12 mov rbp, rsp</td>
</tr>
<tr>
<td>13 stp x29, x30, [sp, #16]</td>
<td>13 sub rbp, 16</td>
</tr>
<tr>
<td>14 add x29, sp, #16</td>
<td>14 mov dword ptr [rbp - 4], 0</td>
</tr>
<tr>
<td>15 stur wzr, [x29, #4]</td>
<td>15 mov dword ptr [rbp - 8], edi</td>
</tr>
<tr>
<td>16 str w0, [sp, #8]</td>
<td>16 mov qword ptr [rbp - 16], rsi</td>
</tr>
<tr>
<td>17 str x1, [sp]</td>
<td>17 mov edi, dword ptr [rbp - 8]</td>
</tr>
<tr>
<td>18 ldr w0, [sp, #8]</td>
<td>18 call cube(int)</td>
</tr>
<tr>
<td>19 bl cube(int)</td>
<td>19 add rbp, 16</td>
</tr>
<tr>
<td>20 ldp x29, x30, [sp, #16]</td>
<td>20 pop rbp</td>
</tr>
<tr>
<td>21 add sp, sp, #32</td>
<td>21 ret</td>
</tr>
<tr>
<td>22 ret</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: Assembly Code comparison**

From this, we can see that the ARM based assembly would carry out 22 lines of code, while the x86-64 would carry out 21, and since we know RISC instructions are one-clock-cycle long while CISC could be longer, the execution of finding a cube of an integer will be faster on ARM than on x86-64, if not, at least it would shave off certain few microseconds, which can be pivotal for HFT.
4.2 ARM Architecture specific optimization

However, simply switching from x86-64 to ARM is not enough for HFT, and it requires optimization of the algorithms to ARM.

One of the key methods of optimizing an algorithm is via vectorization. Hence, when looking to optimize an algorithm on ARM architecture the following areas are of great interest.

4.2.1 SIMD

Single instruction multiple data (SIMD) is a special processor instruction that uses a single instruction to perform the same operation in parallel on multiple data elements of the same type and size such that the available resources are used more efficiently. This way, an addition of 2 32-bit values would be instead performed as a parallel addition of 4 8-bit values in the same amount of time. (ARM, 2009).

When writing vectorized code, there are a few ways to do so, and using SIMD assembly instructions is one way, which is a form of manual vectorization.

In previous generations, ARM chips, had supported a small set of SIMD instructions, however in the more newer ARM chips, starting from ARMv7, a more advanced set of SIMD instructions is being supported. This implementation of the advanced set of SIMD instructions on the ARM processor are called NEON instruction set (ARM, 2009).

4.2.2 NEON

The NEON instruction set that is included in the newer ARM chips includes tremendous amounts of features, starting from auto-vectorization, NEON intrinsics, NEON optimized libraries and even support for assembly code. Many of the features provide support to compilers for vectorizing code from C/C++, and hence significantly
accelerate repetitive operations on large data sets. It is pivotal to provide the compiler with keywords that indicate the requirement of parallelization as the C language does not specify parallelizing behavior. Similarly there are also some additional code optimization tips that will be helpful when writing code specific to optimizing on ARM devices that as seen on a discussion forum on Stack Overflow (2012), such as:

1. Avoiding high-cost instructions, (eg. Division)
   Instead logical shifts or multiplication by inverse are some work arounds
2. Optimizing inner nested for loops by avoiding high-cost instructions or calculations

Applying these tips to a code and then letting the compiler handle the heavy lifting is one of the few approaches to optimizing on an ARM chip

5. Vectorization

5.1 Auto-vectorization

GCC

GCC compiler comes with a set of flags that allow for various levels of optimization.

GCC command line input for O2 optimization:

```
g++ -std=c++11 -O2 -o GSP1_O2 genSortPart1.cpp
```

The overall compiler optimization level is controlled by the command line option -On, where n is the required optimization level, with each optimization level with different objectives in mind, as follows:

- -O0. (default). No optimization is performed.
• -O1. Enables most common forms of optimization that requires no size versus speed decisions, including function inlining.

• -O2. Enables additional optimizations where no size versus speed decision based flags will be enabled. Some examples includes instruction scheduling.

• -O3. Enables further optimizations including those where speed versus size decisions would be taken, such as aggressive function inlining. This level of optimization will also enable -ffree-vectorize, which allows the compiler to generate NEON code directly.

• -Os. Optimizes for size, essentially -O2 with some flags that lead to an increase in code size turned off.

• -Ofast. Enables all -O3 flags and additionally more flags that might cause the code to misbehave as it disregards various compliance standards.

• -Oz. Enables further optimizations compared to -Os, again aggressively optimizing for size, where it enables most -O2 flags apart flags that increase the code size of the executable.

In order to measure the performance difference, an implementation that could simulate the speed of HFT implementation tries to leverage the auto-optimization features. This program sorts 10000 numbers stored in a vector for 1000 different times.

```cpp
#include <algorithm>
#include <chrono>
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;
using namespace std::chrono;

void genAndSortVector(){
    vector<int> values(10000);
}
11 auto f = []() -> int { return rand() % 10000; };
12 generate(values.begin(), values.end(), f);
13 sort(values.begin(), values.end());
14 }

16 int main(int argc, char** argv){
17 auto startMain = high_resolution_clock::now();
18 int durationArray [1001];
19 int sum = 0;
20 for(int i=1; i<1001; i++){
21 auto start = high_resolution_clock::now();
22 genAndSortVector();
23 auto stop = high_resolution_clock::now();
24 auto duration = duration_cast<microseconds>(stop-start);
25 sum += duration.count();
26 durationArray[i] = duration.count();
27 }
28 auto stopMain = high_resolution_clock::now();
29 auto durationMain = duration_cast<milliseconds>(stopMain-startMain);
30 cout << "--------------------------------------------------------------" << endl;
31 cout << "Time taken by entire program: " << durationMain.count() << " milliseconds" << endl;
32
33 float mean = (float(sum)/1000);
34 float var = 0;
35 for( int n = 0; n < 1000; n++ ){
36 var += (durationArray[n] - mean) * (durationArray[n] - mean);
37 }
38 var /= 1000;
39 float sd = sqrt(var);
40 cout << "Mean time taken by single for loop: " << mean << " microseconds" << endl;
41 cout << "Standard Deviation: " << sd << " microseconds" << endl;
42 }

Code 2: Sort 1000 vectors consecutively containing 10,000 numbers per vector

Upon compiling the code with different levels of Optimization from O0 to O3, on both x86 and ARM below were the results that were recorded
The first implementation of an optimized HFT algorithm on an ARM chip

**Final Report**

---

The results show that the same code with the same compiling flags lead to a huge discrepancy when comparing the performance on a x86 machine compared to an ARM machine. This validates the hypothesis that was derived from when the assembly code of x86 and ARM was cross compared for a smaller impact implementation.

Further, when trying to specify the compiler to use optimization attuned to ARM architecture by specifying the architecture and to allow the compiler to generate NEON code directly:

```
g++ -march=armv8-a -mfpu=neon -mfloat_abi=hard -ftree-vectorize -std=c++11 -O2 -o GSP1ARM_O2 genSortPart1.cpp
```

**Figure 1: Performance comparison: ARM & x86**

The above performance comparison with the specified additional flags highlight that there is not much discrepancy between passing a longer list of flags in comparison to O1/2/3. In actuality, a few of the “additional” flags, such as -ftree-vectorize is turned on in O3 optimization, hence rendering passing additional flags useless in this case.

---

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
<td>O1</td>
</tr>
<tr>
<td>Total time taken for entire program</td>
<td>495 Milliseconds</td>
<td>427 Milliseconds</td>
</tr>
<tr>
<td>Mean time per loop</td>
<td>494.515 Microseconds</td>
<td>426.65 Microseconds</td>
</tr>
<tr>
<td>Standard deviation of time per loop</td>
<td>83.64 Microseconds</td>
<td>1.49505e+07 Microseconds</td>
</tr>
</tbody>
</table>

**Figure 2: Performance Comparison: ARM**
One more observation that can be seen is optimization level O2 seems to perform the best in this case.

Clang

Similar to GCC, Clang compiler also supports flags to enable different optimization levels, with the same convention of -On.

```
clang++ -std=c++11 -O2 genSortPart1.cpp -o GSP1_ClangO2
```

<table>
<thead>
<tr>
<th></th>
<th>ARM: GCC</th>
<th>ARM: Clang</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
<td>O1</td>
</tr>
<tr>
<td>Total time taken for entire program (Milliseconds)</td>
<td>495</td>
<td>427</td>
</tr>
<tr>
<td>Mean time per loop (Microseconds)</td>
<td>494.515</td>
<td>426.65</td>
</tr>
<tr>
<td>Standard deviation of time per loop (Microseconds)</td>
<td>83.64</td>
<td>1.49505e+07</td>
</tr>
</tbody>
</table>

Figure 3: Performance Comparison: GCC & Clang

For the implementation of a HFT algorithm, having a lower standard deviation of time per loop tends to be a better metric than the mean time per loop, as in HFT applications having a lower consistent latency is far more important. The results across the 3 different experiments show that the O2 level of optimization is the most optimal. However, putting this in context to our application is important too, and hence it is too early to tell.

It is also worth highlighting that simply relying on the compiler to generate a code that is efficient is not enough, and we need to rely on more SIMD-like instructions that had been introduced earlier.
5.2 NEON Intrinsics

The other way to optimize code is by utilizing NEON intrinsics within the code, hence allowing the programmer to be able to interact with machine code, in a low-level behaviour, all the while doing this from a high-level language (ie. C/C++). NEON intrinsics usually appear as function calls in C/C++ that optimize the code and improves its performance.

This project will look to leverage NEON intrinsics heavily to interact with assembly level code directly from a higher-level code to better optimize the implementation in order to achieve vectorization.

To begin with, a small instance case of the cube calculator of 1000 randomly generated number in scalar operations and comparing that with NEON intrinsics code was performed. Below is the main logic performing code snippet:

```
int cube(int num){
    return num*num*num;
}

void cubeHandler (){}

int A = rand();
float32x4_t A = {{(float)rand()},
(float)rand(), (float)rand(),
(float)rand()};

int B = 10;
float32x4_t B = vdupq_n_f32(10.0f);

int C = A-B*(A/B); //In order to avoid costly operation A%B
float32x4_t C = vsubq_f32(A,
vmulq_f32(B, vdivq_f32(A, B)));

cube(C);
}
```

Code 3: Code Snippet: Scaler vs NEON Vectorized
When calling the CubeHandler() function, the cube handler would create a float32x4_t data type, which assigns 4 values to the 4 registers/lanes that are SIMD enables. It would then perform the calculations on all 4 lanes simultaneously (SIMD) and would then call Cube(), while passing all the 4 lanes of data to cube so again performing SIMD operations. All the while ensuring each for loop in the NEON version (which in theory calculates 4 different cube values), is approximately double in duration than the scalar code’s for loop (which only calculates a single for loop). In retrospect, a vectorized for loop taking twice as long, but outputting 4 times more values, certainly is extremely efficient.

With improvement of upto 82% faster code when using NEON intrinsics to perform the cubing in a SIMD manner, it’s a clear indicator that converting a scalar code into vector using NEON intrinsics certainly has significantly made the code faster, which again supports the original hypothesis.

<table>
<thead>
<tr>
<th></th>
<th>Time Taken</th>
<th>Q0</th>
<th>Q2</th>
<th>Q3</th>
<th>Qs</th>
<th>O2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total (µs)</td>
<td>202</td>
<td>53</td>
<td>142</td>
<td>74</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Avg. for loop (ns)</td>
<td>55.5430</td>
<td>27.8740</td>
<td>72.6300</td>
<td>37.9160</td>
<td>27.7900</td>
</tr>
<tr>
<td></td>
<td>S.D. for loop (ns)</td>
<td>22.6633</td>
<td>26.4754</td>
<td>19.8180</td>
<td>13.3703</td>
<td>22.7696</td>
</tr>
<tr>
<td>NEON</td>
<td>Total (µs)</td>
<td>70</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Avg. for loop (ns)</td>
<td>136.66400</td>
<td>60.30800</td>
<td>61.51600</td>
<td>59.84000</td>
<td>59.32800</td>
</tr>
<tr>
<td></td>
<td>S.D. for loop (ns)</td>
<td>22.25230</td>
<td>22.63080</td>
<td>35.22130</td>
<td>22.59790</td>
<td>23.72150</td>
</tr>
</tbody>
</table>

Figure 3: Performance Comparison: Random Cube

Percentage change in total time when applying NEON

-0.65346535 -0.5283019 -0.8239437 -0.6621622 -0.5283019

The first implementation of an optimized HFT algorithm on an ARM chip

Final Report

Page 19 of 41
6. Project Formulation & Experimenting

6.1 Project Formulation

Over the span of the Final Year Project, a lot of tweaking had been performed in order to come to the current topic. Initially, the topic was meant to be increasing the performance of HFT algorithms from the perspective of making more profit via increased Machine Learning Algorithms, Sentiment Analysis & Quantitative methods. However, from there, a change was made to instead focus on improving the performance of HFT in terms of latency. This also made me think along the lines of scheduling issues, and multithreading.

As mentioned in the introduction, there has been a steady rise in RISC based processors, with heavy commercialization of Apple Silicon recently, it was worth looking into how much more efficient can ARM processor be. While currently, most HFT firms still are on x86 architecture, and while they are using libraries such as Boost for Intel, they hadn’t given a thought to trying optimization on ARM processors.

From there, my objective was to take pre-existing HFT algorithms and apply NEON intrinsics such that it could be completed in a SIMD manner. Using this, a more concrete outcome was to come out with paper testing the vectorized algorithm in real time data. And to then compare the results of the code against scalar code and x86 platform scalar code.

While doing further research, it came to my attention that in fact x86 systems also do support SIMD, with Intel providing SSE Intrinsics to allow x86 based users to optimize code for SIMD operations.

Hence allowing for comparison of ARM based vectorized code against Intel SSE based vectorized code as well.
6.2 Experimentation

It is worth mentioning that most, profit making, usable code would either be proprietary or the author of the code wouldn’t want the algorithm to be available publicly. Additionally, I had a self-imposed restriction to use only C/C++ based code in order to use NEON intrinsics, while many projects had employed Python. Hence, most of the available existing projects in C/C++ online always came with a catch, where some might have multiple dependency issues, incompatible operating platforms, poor documentation, or simply unusable for NEON intrinsics implementation. Owing to which, I was looking at various types of and trying to focus on different parts of HFT algorithms. Some of the implementations that I had tried to work with but eventually had to scrap were:

6.2.1 Ultra-Fast Matching Engine & Order Book

Retrieved from: https://github.com/chronoxor/CppTrader

This repository focuses on creating an engine that matches buy order and sell order with each other, and at the same time builds an order book. While this project would have been a perfect candidate for optimizing using NEON intrinsics to promote SIMD operations, much of it has been written with high complexity, and the project being at a larger scale hence making it not a good candidate for implementing NEON intrinsics.

6.2.2 Front to back algorithm connected to Alpaca Trading API

Retrieved from: https://github.com/marpaia/alpaca-trade-api-cpp

This implementation was quite close to what I had in mind from the get go, a full front to end algorithm that could be linked to a paper brokerage firm. While there are certainly many different brokerages, many do also have restrictions as to the amount of API interactions, or lack of paper trading. Alpaca is a trading platform that doesn’t
have any sort of restrictions and as is pro-algorithmic trading hence allowing for its users to live test their code in a simulated environment.

While Alpaca doesn’t provide C++ trading API support on their own, the Alapaca community did come up with one. Upon setting up, the trade API works well and had been able to create trades using the Trade API, however the community based C++ API is outdated and lacks Market Data APIv2 support. As Alpaca servers would only connect & communicate with Market Data API v2, using this community API doesn’t work. Delving further in, there were other such libraries that did support Market Data APIv2 but had tons of dependency issues. Hence meaning this too had to be scrapped.

6.2.3 HFT Forex Bot

Retrieved from: https://github.com/mpearl1020/HFT-Forex-Bot

This repository is described to be a HFT bot that uses Bellman-Ford algorithm to act on arbitrage. However, this repository didn’t contain proper documentation for installation.

Owing to some of the many different projects I tried to use as above but wasn’t able to properly implement vectorization, I had to deviate a little to apply vectorization using NEON to optimize Derivatives Pricing instead of High Frequency Trading. And hence am not able to output a complete front to back algorithm that can be linked with a paper trading brokerage account to test the performance of the algorithm in profit making terms nor in order latency. However, have been able to compare the performance of the algorithm against the scalar implementations in terms of time taken to execute the code.

7. Implementation

More on NEON data type & methods
Most of the data types across the implementations have been in Float data type in the scalar version and in float32x4_t in the vector implementations. While NEON intrinsics does support Double data type processing as well, it certainly wouldn’t be efficient. Double data type occupies 64-bits in computer memory, while Float Data type occupies 32-bits, and NEON Intrinsics allows for a maximum of 128-bits of memory per data type. Hence keeping SIMD, multiple lanes carrying data in mind, we could have a data type of 64-bits (Double) across 2 lanes (ie. float64x2_t), or we could have a data type of slightly smaller precision of 32-bits(float) across 4 lanes (ie. float32x4_t). This again boils down to a speed vs. precision issue, where float32x4_t in comparison would allow us to process two times more data at the expense of slightly smaller precision. As a very small difference in precision (> 0.01%) was measured, it could be considered too insignificant of an error to disallow the conversion of data type to float32x4_t.

This consistency was important to set out clear before beginning as all the mathematical & logical methods in NEON are specific to that data type. For instance, a vmulq_f32() is a method for multiplying two float32x4 data types against each other, while a vmul_f32() is a method for multiplying two float32x2 data types. If there is a mismatch between the data type and the type of the method used the compiler would generate errors. Hence setting out the data types from the get go was a key procedure in the implementation.

7.1 Black Scholes Model


The above Black Scholes model for European vanilla option pricing would take parameters such as the Strike Price, Spot Price, Volatility, Time to Maturity, Risk-free Rate and would output the Call & Put prices for a single set of instruction. Hence, minor tweaks had to be made so that it instead performs the calculations for 1,000,000 different situations instead of just one, however for simplicity the tweaked version would
simply reuse the same values to calculate the Call & Put prices. As the main purpose is
to find a measure of the performance of the scalar implementation, we only keep record
of the time taken by the entire program, and simple statistics for each for loop.

Once the scalar code is up and running, NEON intrinsics were then applied to the code
to allow for each for loop to handle the processing of 4 batches of data, all of which
using the float32x4_t data type. This required all the functions to be re-written using
NEON intrinsics to perform calculations simultaneously, on all the lanes, instead of
sequentially as it would in the scalar implementation. Due to the limitations of the
NEON intrinsics library, some of the code also needed to be re-written in another where
the intrinsics could be applied instead.

A small code snippet from both the scalar and vector implementations is as below:

```c
float d_j(const int& j, const float& S, const float& K, const float& r, const
float& v, const float& T) {
    float numerator = log(S/K) + (r + (pow(-1,j-1))*0.5*v*v)*T;
    float denominator = v*(pow(T,0.5));
    return numerator / denominator;
}
```

```c
float32x4_t d_j(const int& j, const float32x4_t& S, const float32x4_t& K, const
float32x4_t& r, const float32x4_t& v, const float32x4_t& T) {
    float32x4_t powTerm = vdupq_n_f32((float)1);
    if (j==2) {
        powTerm = vnegq_f32(powTerm);
    }
    float32x4_t vvHalf = vmulq_n_f32(vmulq_f32(v, v), (float)0.5);
    //vmulq_n_f32: multiply 32x4 by scalar; vmulq_f32: multiply 32x4 by 32x4
    //0.5*v*v
    float32x4_t powVVHalf = vmulq_f32(powTerm, vvHalf); //pow(-1,j-1)*0.5*v*v
    float32x4_t summedPow = vaddq_f32(r, powVVHalf); //r + (pow(-1,j-1)*0.5*v*v
    //vaddq_f32: add 2 32x4 to each other lane by lane
```
The first implementation of an optimized HFT algorithm on an ARM chip

The \textit{d} \_j function is a function that is directly related to the pricing of the options, with a pre-defined formula for \textit{d}_j where the value of \textit{j} is passed to be either 1 or 2 as the very first argument when calling \textit{d}_j() (Quantstart, n.d.).

\[
\begin{align*}
\text{d}_1 &= \frac{\log\left( \frac{S}{K} \right) + \left( r + \left( \text{pow}(-1, j - 1) \right) \right) \times \left( 0.5 \times v \times v \right) \times T}{\sigma \sqrt{T}} \\
\text{d}_2 &= \text{d}_1 - \sigma \sqrt{T}
\end{align*}
\]

Therefore the coded scalar formula for \textit{d} \_j is given to be (Quantstart, n.d.):

\[
\frac{\log\left( \frac{S}{K} \right) + \left( r + \text{pow}(-1, j - 1) \right) \times 0.5 \times v \times v \times T}{v \times \text{pow}(T, 0.5)}
\]

Some example intrinsics that were used to calculate the final \textit{d} \_j value, that are also more commonly used in the rest of the code are:

\texttt{vaddq_f32}: Used to add two float32x4 \_t vectors up and the output would be a single float32x4 \_t vector. Eg. [a, b, c, d], [1, 2, 3, 4] = [a+1, b+2, c+3, d+4]
vmulq_f32: Used to multiply two float32x4_t vectors against each other and the output would be a single float32x4_t vector. Eg. [a, b, c, d], [1, 2, 3, 4] = [a1, b2, c3, d4]

vmulq_n_f32: Used to multiply a float32x4_t vector against a scalar value each and the output would be a single float32x4_t vector. Eg. [a, b, c, d], 7 = [a7, b7, c7, d7].

vdivq_f32 Used to divide a float32x4_t vector against another of same type and the output would be a single float32x4_t vector. Eg. [a, b, c, d], [e, f, g, h] = [a/e, b/f, c/g, d/h].

While the above listed intrinsics are the more commonly used intrinsics across the Black Scholes Model, there certainly are some which are more unique. Additionally, the ARM specified NEON intrinsics library isn’t well equipped for slightly more mathematically heavy calculations such as calculating log(), pow(), exp(), etc. For this, a separate library that has already implemented such methods using NEON intrinsics was used.

neon_mathfun.h, retrieved from: http://gruntthepeon.free.fr/ssemath/neon_mathfun.html, is a library that provides programmers with optimized functions for log_ps(), exp_ps() and 3 more trigonometric calculations. The scalar code for the Black Scholes pricing methods only calls for log() & pow(). While neon_mathfun.h doesn’t support a pow() method directly, pow(x,y) could in fact be rewritten as exp(y * log (x)).

While rewriting pow(T, 0.5) [line 3 on Scalar code] to exp_ps(vmulq_n_f32(log_ps(T), 0.5)) [line 13 on Neon code] seems to make sense. When trying to rewrite pow(-1, j-1) [line 2 on Scalar code] would essentially mean rewriting it to exp_ps(vmulq_n_f32(j-1, log(-1))). But the issue here is log(-1) would return a nan value for a floating point variable. Hence requiring a more logic taken approach to deal with this term. As this was just a term to calculate if d_j would calculate for d1 or d2, this was tackled using conditional statements.
7.2 Monte Carlo Simulation Code


The above Monte Carlo Simulation using the Gaussian Box Muller algorithm to generate large random numbers that would approximate the expectation would take parameters such as the Strike Price, Spot Price, Volatility, Time to Maturity, Risk-free Rate and would output the Call & Put prices based on the random simulations. This code requires each batch of inputs to be simulated over 1,000,000 times. Hence for this, the NEON SIMD operation can be applied on the step that is responsible for calling the simulations for the 1,000,000 times, and accordingly all the functions that are called from there on. Therefore, here too, a large part of the code needs to be rewrote in order to perform SIMD operations for each loop.

The original code constantly calls functions multiple times hence duplicating it’s efforts, where in fact it would have saved much time by simply reusing the same variables, that’s why a faster scalar version was also created such that the methods reuse the variables. And then also implemented these changes along with a couple more performance changes in the vectorized code.

```c
1 float gaussian_box_muller() { .. }

2

3 float monte_carlo_call_price(const int& num_sims, const float& S, const float& K, const float& r, const float& v, const float& T) {
4     float S_adjust = S * exp(T*(r-0.5*v*v));
5     float S_cur = 0.0;
6     float payoff_sum = 0.0;
7     float zero = 0.0;
8
9     for (int i=0; i<num_sims; i++) {
10         float gauss_bm = gaussian_box_muller();
11         S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
12     }
13     return payoff_sum / num_sims;
14 }
```
The first implementation of an optimized HFT algorithm on an ARM chip

12    payoff_sum += std::max(S_cur - K, zero);
13 }
14    return (payoff_sum / static_cast<float>(num_sims)) * exp(-r*T);
15 }
16
17 float monte_carlo_put_price(const int& num_sims, const float& S, const float& K,
18                            const float& r, const float& v, const float& T) {
19    float S_adjust = S * exp(T*(r-0.5*v*v));
20    float S_cur = 0.0;
21    float payoff_sum = 0.0;
22    float zero = 0.0;
23
24    for (int i=0; i<num_sims; i++) {
25        float gauss_bm = gaussian_box_muller();
26        S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
27        payoff_sum += std::max(K - S_cur, zero);
28    }
29
30    return (payoff_sum / static_cast<float>(num_sims)) * exp(-r*T);
31 }
32
33 int main(int argc, char **argv) {
34    ...
35    float call = monte_carlo_call_price(num_sims, S, K, r, v, T);
36    float put = monte_carlo_put_price(num_sims, S, K, r, v, T);
37    ...
38
39 Code 4: Monte Carlo Simulation Scalar v1

1 float gaussian_box_muller() {...
2 std::pair<float, float> monte_carlo_call_put_price(const int& num_sims, const
3    float& S, const float& K, const float& r, const float& v, const float& T) {
4    float S_adjust = S * exp(T*(r-0.5*v*v));
5    float S_cur = 0.0;
6    float payoff_sum_call = 0.0;
7    float payoff_sum_put = 0.0;
8    float zero = 0.0;
9    double sqrtVariable = sqrt(v*v*T);
10   for (int i=0; i<num_sims; i++) {
11       float gauss_bm = gaussian_box_muller();
The first implementation of an optimized HFT algorithm on an ARM chip

The Gaussian Box-Muller, and the static scalar calculations would be called twice as per Code 6: v1, once from the `monte_carlo_call_price()` & once from the `monte_carlo_put_price()`. By resusing these values in Code 7: v2, in the `monte_carlo_call_put_price()` function in v2, it cuts down an entire function hence reduces the time taken by roughly 50%. And when passing such a streamlined code for vectorization, a more efficient outcome could be anticipated.

```cpp
1 std::pair<float, float> monte_carlo_call_put_price(const int& num_sims, const float& S, const float& K, const float& r, const float& v, const float& T) {
2    float S_adjust = S * exp(T*(r-0.5*v*v));
3    float32x4_t S_cur;
4    float payoff_sum_call = 0.0;
5    float payoff_sum_put = 0.0;
6    float zero = 0.0;
7    float sqrtVariable = sqrt(v*v*T);
8    float finalExp = exp(-r*T);
9    
10   for (int i=0; i<num_sims; i+= 4) {
11      float32x4_t gauss_bm = gaussian_box_muller();
12      S_cur = vmlq_n_f32(exp_ps(vmlq_n_f32(gauss_bm, sqrtVariable)), S_adjust);
13      payoff_sum_call += std::max(S_cur - K, zero);
14      payoff_sum_put += std::max(K - S_cur, zero);
15   }
16   return {(payoff_sum_call / static_cast<float>(num_sims)) * exp(-r*T),
17            (payoff_sum_put / static_cast<float>(num_sims)) * exp(-r*T)};
18 }
19
20 int main(int argc, char **argv) {
21   ...
22   std::pair<float, float> callPut = monte_carlo_call_put_price(num_sims, S, K, r, v, T);
23 ...
```

**Code 5: Monte Carlo Simulation Scalar v2**

The guassianBoxMuller, and the static scalar calculations would be called twice as per Code 6: v1, once from the `monte_carlo_call_price()` & once from the `monte_carlo_put_price()`. By resusing these values in Code 7: v2, in the `monte_carlo_call_put_price()` function in v2, it cuts down an entire function hence reduces the time taken by roughly 50%. And when passing such a streamlined code for vectorization, a more efficient outcome could be anticipated.
The first implementation of an optimized HFT algorithm on an ARM chip

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td><code>payoff_sum_call += vaddvq_f32(vmaxq_f32(vsubq_f32(S_cur, vdupq_n_f32(K)), vdupq_n_f32(0))))</code></td>
</tr>
<tr>
<td>14</td>
<td><code>payoff_sum_put += vaddvq_f32(vmaxq_f32(vsubq_f32(vdupq_n_f32(K), S_cur), vdupq_n_f32(0))))</code></td>
</tr>
<tr>
<td>15</td>
<td><code>}</code></td>
</tr>
<tr>
<td>16</td>
<td><code>return {(payoff_sum_call / static_cast&lt;float&gt;(num_sims)) * finalExp, (payoff_sum_put / static_cast&lt;float&gt;(num_sims)) * finalExp};</code></td>
</tr>
<tr>
<td>17</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

*Code 6: Monte Carlo Simulation NEON*

Again, some of the many intrinsics that have been used here in this code includes:

**vaddvq_f32**: Used to sum up all the values in a float32x4_t vector and would then return a float32 value. Eg. \([a, b, c, d] = a + b + c + d\).

**vmaxq_f32** Used to find the max value in a pair by pair comparison between two float32x4_t vector, and would then return a single float32x4_t vector. Eg. \([3, 7, 2, 5], [1, 9, 3, 2] = [3, 9, 3, 5]\).

**vsubq_f32** Used to subtract one float32x4_t vector from another, again in a per element manner. The method would then return a single float32x4_t vector. Eg. \([3, 7, 2, 5] , [1, 9, 3, 2] = [2, -2, -1, 3]\).

**vdupq_n_f32** Used to make a float32x4_t vector from a given float32 number, return type is float32x4_t. In other words the give float value is copied and assigned to the other lanes. Eg. \((0.0f) = [0.0, 0.0, 0.0, 0.0]\).

**vgetq_lane_f32** Used to retrieve the float value in a particular lane of a float32x4_t vector. Eg. \([a, b, c, d], 3 = d\)

However it is worth noting that there were some scalar operations that were still applied in the NEON code [line 2, 7, 8] as none of these values were dependent on SIMD passed variables. SIMD only begins in the Monte Carlo NEON code starting from the calling of gaussian box muller function in the for loop that increments by 4 every time (owing to 4 lanes being used per iteration to store the float data).
7.3 Pricing Derivatives

Retrieved from: https://github.com/welien/pricing-derivatives

This repository is a project which reads market data from a CSV and would then pass it to the models to calculate the options pricing as per different models and would output the final profit. After making some changes to the code base, I was able to make the repository work with my code base in a SIMD manner, however am unable to get proper results with this implementation.

As the Monte Carlo Simulation code can directly perform SIMD within its own code, I began with including the Monte Carlos Code (gausBox.cpp) into the pricing-derivatives repository and included the linkers and project set up to get it working.

```
Total result is 21903.6
Time taken by Black Scholes option pricing program: 54 milliseconds

Total result is 36504
Time taken by Gaus Box Model Option Pricing: 5471 milliseconds

Total result is 5631.85
Time taken by Binomial Model Option Pricing: 3358 milliseconds
```

*Figure 5: Added Monte Carlo Pricing to the code base*

In theory, the gausBox.cpp should output something similar to Black Scholes Model as mentioned in the source website for the Monte Carlos Simulation Code, however as the total result doesn’t match between the two models, I hypothesized perhaps there might be some error with the repository Black Scholes Model.

In order to use my version of the Black Scholes model, I had to rework a significant part of experiments.cpp such that the for loop and other parameters that would be passed to the Black Scholes Model function would also have float32x4_t data type for SIMD operations.
As the result shows, both the vectorized Black Scholes model and the vectorized Monte Carlo simulation code, when connected with this pricing-derivatives repository, doesn’t seem to work accurately, however when ran on their own, outside this repository, produce accurate results. A hypothesis as to why such results are occurring is due to the improper passing of parameters to these functions and a lack of understanding of the repository.

7.4 Results

All the various executables were obtained by compiling the codes with the same compiler settings, only tweaking the optimization level as shown below

```
clang++ -march=armv8-a -On inputFile -o outputFile
```

### Black Scholes Model

<table>
<thead>
<tr>
<th></th>
<th>Time Taken</th>
<th>O0</th>
<th>O2</th>
<th>O3</th>
<th>Oz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td>Total (μs)</td>
<td>358,093</td>
<td>94,757</td>
<td>88,139</td>
<td>183,181</td>
</tr>
<tr>
<td></td>
<td>Avg. for loop (ns)</td>
<td>284.22</td>
<td>56.62</td>
<td>52.67</td>
<td>149.12</td>
</tr>
<tr>
<td></td>
<td>S.D. for loop (ns)</td>
<td>182.98</td>
<td>55.85</td>
<td>41.58</td>
<td>115.66</td>
</tr>
<tr>
<td><strong>NEON</strong></td>
<td>Total (μs)</td>
<td>2,264,066</td>
<td>81,125</td>
<td>83,967</td>
<td>92,062</td>
</tr>
<tr>
<td></td>
<td>Avg. for loop (ns)</td>
<td>8,963.96</td>
<td>286.53</td>
<td>287.68</td>
<td>332.31</td>
</tr>
<tr>
<td></td>
<td>S.D. for loop (ns)</td>
<td>509.43</td>
<td>127.48</td>
<td>106.67</td>
<td>58.98</td>
</tr>
</tbody>
</table>

Percentage change in total time when applying NEON

|                          |                  | 5.32256425 | -0.14386272 | -0.04733432 | -0.49742604 |

**Figure 6: Converted parts of the code to perform SIMD**

**Figure 7: Performance Comparison: Black Scholes Pricing Model**
When looking at the performance of the Black Scholes model and comparing the scalar and vector implementations on ARM processors, the vectorized code is only faster than the scalar implementation if an optimization flag is applied. If none, the NEON candidate takes 5 times more time than the scalar implementation.

<table>
<thead>
<tr>
<th>Time Taken</th>
<th>O0</th>
<th>O2</th>
<th>O3</th>
<th>Os</th>
<th>Oz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td>202</td>
<td>53</td>
<td>142</td>
<td>74</td>
<td>53</td>
</tr>
<tr>
<td>Total (µs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. for loop (ns)</td>
<td>55.5430</td>
<td>27.8740</td>
<td>72.6300</td>
<td>37.9160</td>
<td>27.7900</td>
</tr>
<tr>
<td>S.D. for loop (ns)</td>
<td>22.6633</td>
<td>26.4754</td>
<td>19.8180</td>
<td>13.3703</td>
<td>22.7696</td>
</tr>
<tr>
<td><strong>NEON</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (µs)</td>
<td>70</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Avg. for loop (ns)</td>
<td>136.66400</td>
<td>60.30800</td>
<td>61.51600</td>
<td>59.84000</td>
<td>59.32800</td>
</tr>
<tr>
<td>S.D. for loop (ns)</td>
<td>22.25230</td>
<td>22.63080</td>
<td>35.22130</td>
<td>22.59790</td>
<td>23.72150</td>
</tr>
<tr>
<td>Percentage change in total time when applying NEON</td>
<td>-0.65346535</td>
<td>-0.5283019</td>
<td>-0.8239437</td>
<td>-0.6621622</td>
<td>-0.5283019</td>
</tr>
</tbody>
</table>

*Figure 4: Performance Comparison: Random Cube (as earlier)*

When comparing this with the initial test case that we had produced using the Random Cube, where we found a speedup even by using NEON intrinsics out of the box of about 60%, this in contrast to a 500% slowdown in the Black Scholes Pricing model.

Higher optimization flags in vectorized code in the test case perform at a reduction in time taken by an average of 62%, with the max reduction being 82%, while the Black Scholes Model performs at a reduction by an average of 22%, with the max reduction being 50% when using the -Oz flag. However, -O2 level optimization seems to produce the fastest candidate for vectorization, while -O3 for scalar candidate.
Monte Carlo Simulation model

<table>
<thead>
<tr>
<th></th>
<th>Time Taken Total (μs)</th>
<th>O0</th>
<th>O2</th>
<th>O3</th>
<th>Oz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1,185,211</td>
<td>799,358</td>
<td>799,231</td>
<td>859,665</td>
<td></td>
</tr>
<tr>
<td>NEON</td>
<td>3,560,171</td>
<td>731,273</td>
<td>732,045</td>
<td>741,427</td>
<td></td>
</tr>
</tbody>
</table>

Percentage change in total time when applying NEON

- 2.003828854
- 0.0851746
- 0.08406331
- 0.13753962

*Figure 8: Performance Comparison: Monte Carlo Simulation Model*

When looking at the performance of the Monte Carlo Simulation model and comparing the scalar and vector implementations on ARM processors, the vectorized code again is only faster than the scalar implementation if an optimization flag is applied. If none, the NEON candidate takes 2 times more time than the scalar implementation.

A similar story can be see with the Monte Carlo Simulation Code performance and Black Scholes Code performance, where both are significantly less faster than what we saw in the small test case. When looking closer, although the vectorized code does perform faster than the scalar implementation, its only with an average reduction in 10% of compute time, with the max being 13%, statistics which are much smaller than what we saw in the Black Scholes Model.

A hypothesis as to why such results were obtained

Usage of rand()

```c
1   float32x4_t gaussian_box_muller() {
2     float32x4_t x, y, euclid_sq;
3
4     do {
5         float32x4_t randx = {float(rand()), float(rand()), float(rand()), float(rand())};
6         float32x4_t randy = {float(rand()), float(rand()), float(rand()), float(rand())};
```
The first implementation of an optimized HFT algorithm on an ARM chip

Final Report

Page 35 of 41

In order to perform the Monte Carlo Simulations, a random number needs to be seeded, and on which mathematical computations would be performed as shown above in the gaussian_box_muller() code that is in the Monte Carlo Simulation Code. Further looking at lines 5 & 6, the rand() is being called for a total of 8 times, in comparison to only 2 in the scalar implementation. The reason for rand() to be called 4 times per vector is due to the 4 lanes of the float32x4_t data structure as mentioned earlier. While, rand() could be simply called once and the value could be duplicated across the different lanes, this would increase the error rate, and wouldn’t really make it “random”. And yet again there was limited support for NEON based libraries for SIMD random number generation, which is why rand() had to be called for a total of 8 times. Let’s try to experiment how it would perform if we do proceed to duplicate rand() across the 4 lanes:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>float32x4_t randx = vdupq_n_f32(float(rand())); //vdupq_n_f32: duplicate float 32 value across 4 lanes</td>
</tr>
<tr>
<td>6</td>
<td>float32x4_t randy = vdupq_n_f32(float(rand()));</td>
</tr>
</tbody>
</table>

Code 10: Updated Lines 5 & 6 from Gauss_Box_muller()
As suspected, lines 5&6 were the bottleneck of the code, as once it was replaced with `vdupq_n_f32(float(rand()))`, the vectorized code is performing 80% faster than the scalar code, though there is an issue with the precision, and to reiterate, the random values generated are only of 1/4 of the numbers, as 3 out of the 4 numbers in each iteration are copied from that one random number per loop.

Looking at the performance comparison, while it does look more akin to the test case Random Cube with roughly 80% average reduction in compute time, with max being 82%, however due to it’s precision and it’s pseudo(-pseudo-)random generation, it doesn’t satisfy the requirement to be a Monte Carlo Simulation.

**Analysis on x86**

In order to get a more coherent, full picture, comparing the performance of this code on x86 is also an important step, however NEON intrinsics are only ARM specific.

There is a library, NEON_2_SSE that allows the conversion of a NEON vectorized code to be ported to Intel SSE. (Intel, 2021; Intel, n.d.)
This library provides conversion support for all v7 Neon Intrinsics, however doesn’t support v8 or Arch 64 Intrinsics, which unfortunately meant further dependency issues and haven’t been able to benchmark my implementations on a x86 based processor.

8. Evaluation & Recommendations

8.1 Project Evaluation

The original project deliverables and outcome was meant to be a full front to back trading bot that would’ve been vectorized using NEON intrinsics such that it would be optimized specific to ARM processors, along with recommendations to corporates and funds that engage in HFT to enhance their algorithms’ performance for higher efficiency and lower latency. While this project wasn’t able to deliver on a complete front to back trading bot due to some issues mentioned above and the deviation that had to be taken, it certainly has lay some ground work as to the usage of SIMD libraries on the up & rising ARM based processors in order to vectorize code for higher efficiency and lower latency.

The results of this project do suggest that applying vectorization decreases the total compute time for most code, albeit at varying levels, however as we know even a nanosecond is all that matters in HFT. And from the results, vectorization specific to ARM architecture can shave of 15% compute time at the minimum and could go up to 82%. This highly depends on how the vectorization was performed, identifying bottlenecks in the code and working around such issues. Meaning with more research and work, the average reduction could further increase.

Many firms are still working with x86 based architecture but with efficiency libraries such as boost, SSE, etc. With the higher commercialization of ARM processors in notebooks and increased visibility of ARM based HPUs, exploring ARM based trading should definitely be something on their radar given the improvement in compute time taken by the Black Scholes model and the Monte Carlo Simulation, which is representative of codes in HFT algorithms. Granted most of HFT firms’ architecture is
The first implementation of an optimized HFT algorithm on an ARM chip

built around x86, further looking into NEON based vectorization should be something of interest to low latency based finance shops.

8.2 Future Work & Recommendation

Future Work

This project just scratches what is the world of SIMD operations and NEON intrinsics, and there certainly is tons of room for further work that I’ve come to realize over this project.

1. Building the system on your own and then applying SIMD

   Over this project, I’ve tried to “plug in” vectorization into a project that I could find, however, due to poor documentations and high complexity it is often difficult to simply “plug in” NEON intrinsics. In order to truly get the full potential of vectorization it would be best to use on a codebase that has been built with vectorization in mind. Things such as data processing, loop iterations, data storage all are important aspects that would be neglected when trying to build on top of existing projects

2. Comparison against vectorization on x86.

   While NEON based vectorization has shown it’s capabilities that it’s performing better than it’s scalar counter-part on the same processor, a big part of it is to actually compare a vectorized code, that is natively written for x86 using SSE4 (instead of a ported conversion that NEON_2_SSE does).

Recommendation

The results from this project shows that using NEON intrinsics to optimize code by vectorizing produces code that executes in a shorter time, hence making it perfect for low latency applications. Coming from this, it would be profitable for firms that look to apply low-latency code need to invest more in ARM architecture and NEON intrinsics engineers to further push the limits of SIMD NEON to obtain a highly vectorized code before more of their competitors do. This could mean developing their architecture to support ARM processors, team revamp to include SIMD SWE, putting out whitepapers, etc.
References


The first implementation of an optimized HFT algorithm on an ARM chip


The first implementation of an optimized HFT algorithm on an ARM chip

