Department of Computer Science
University of Hong Kong
Final Year Project – Interim Report

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

Supervisor: Dr. Ruibang Luo

Siddhant Ningoo

22nd January 2023
# Table of Contents

Abstract ........................................................................................................................................ 3  
1. Introduction & Background ................................................................................................. 4  
   The current situation .............................................................................................................. 4  
   The advancement in technology .......................................................................................... 4  
   The gap .................................................................................................................................. 5  
2. Objective .................................................................................................................................. 5  
3. Project Overview .................................................................................................................... 6  
   Part 1: Literature review and research on ARM architecture ............................................. 6  
   Part 2: Implementation ......................................................................................................... 6  
   Part 3: Appraisal of performance ......................................................................................... 7  
4. The architecture .................................................................................................................... 8  
   4.1 RISC vs CISC .................................................................................................................. 8  
   4.2 ARM Architecture specific optimization ....................................................................... 10  
      4.2.1 Auto-vectorization .................................................................................................. 11  
      4.2.2 NEON Intrinsics ..................................................................................................... 14  
5. Schedule & Milestones ......................................................................................................... 14  
References .................................................................................................................................... 17
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 & multithreading, specific to the High Frequency Trading (HFT) field where high-performance computing is pivotal to successful implementation of HFT algorithms.
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 & Multithreading, 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. Similarly, Multithreading is also of great focus in this project to push the ARM chip to its max, especially when targeting vectorization and multithreading simultaneously in order to achieve two levels of parallelization.

Part 3: Appraisal of performance

As outlined in the objectives, following the complete optimization of the HFT algorithms, including the vectorization and multithreading, 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 requires 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

```cpp
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>10 main:</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 rsp, 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 wrz, [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 rsp, 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.

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).

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

4.2.1 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, 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 -ftree-vectorize, which allows the compiler to generate NEON code directly.

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

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

int main(int argc, char** argv){
    auto startMain = high_resolution_clock::now();
    int durationArray [1000];
    int sum = 0;
    for(int i=1; i<1001; i++){
        auto start = high_resolution_clock::now();
        genAndSortVector();
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop-start);
        sum += duration.count();
        durationArray[i] = duration.count();
    }
    auto stopMain = high_resolution_clock::now();
    auto durationMain = duration_cast<milliseconds>(stopMain-startMain);
    cout << "--------------------------------------------------------------" << endl;
    cout << "Time taken by entire program: " << durationMain.count() << " milliseconds" << endl;

    float mean = (float(sum)/1000);
    float var = 0;
    for( int n = 0; n < 1000; n++ ){
        var += (durationArray[n] - mean) * (durationArray[n] - mean);
    }
    var /= 1000;
    float sd = sqrt(var);
    cout << "Mean time taken by single for loop: " << mean << " microseconds" << endl;
    cout << "Standard Deviation: " << sd << " microseconds" << endl;
}

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

<table>
<thead>
<tr>
<th>ARM</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
</tr>
<tr>
<td>Total time taken for entire program</td>
<td>495 Milliseconds</td>
</tr>
<tr>
<td>Mean time per loop</td>
<td>494.515 Microseconds</td>
</tr>
<tr>
<td>Standard deviation of time per loop</td>
<td>83.6 Microseconds</td>
</tr>
</tbody>
</table>

*Figure 1: Performance comparison: ARM & x86*

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
```

<table>
<thead>
<tr>
<th>ARM</th>
<th>ARM: Additional flags (-mfpu=neon,-ftree-vectorize, etc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O0</td>
</tr>
<tr>
<td>Total time taken for entire program</td>
<td>495 Milliseconds</td>
</tr>
<tr>
<td>Mean time per loop</td>
<td>494.515 Microseconds</td>
</tr>
<tr>
<td>Standard deviation of time per loop</td>
<td>83.6 Microseconds</td>
</tr>
</tbody>
</table>

*Figure 2: Performance Comparison: ARM*

Clang

Similar to GCC, Clang compiler supports flags to enable different optimization levels, with the same convention of -On.
The first implementation of an optimized HFT algorithm on an ARM chip

**Interim Report**

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

![Figure 3: Performance Comparison: GCC & Clang](image.png)

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.

### 4.2.2 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.

### 5. Schedule & Milestones

This project entails the following deliverables by the end:
1. A fully working, optimized HFT bot that has been optimized, with automated order placement to a paper trading account.

2. Research findings on the architecture of ARM & that of x86-64 and the discrepancies amongst them.

Below is the amended schedule of the project along with the major milestones, along with a Gantt chart illustrating the timeline.

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Time period</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase 0: Ideation and research</strong></td>
<td><strong>August – September 2022</strong></td>
</tr>
<tr>
<td>• Background research on existing implementations of HFT</td>
<td></td>
</tr>
<tr>
<td>• Familiarization with NEON instruction set, key differences in x86-64 and in ARM chips for vectorization &amp; multithreading</td>
<td></td>
</tr>
<tr>
<td>Deliverable: Project Plan &amp; Project Website</td>
<td>2nd October, 2022</td>
</tr>
<tr>
<td><strong>Phase 1: Research</strong></td>
<td><strong>October – December 2022</strong></td>
</tr>
<tr>
<td>• Literature review on ARM architecture</td>
<td></td>
</tr>
<tr>
<td>• Documentation</td>
<td>22 January 2023</td>
</tr>
<tr>
<td>Deliverable: Interim Report; Prelim Implementation</td>
<td></td>
</tr>
<tr>
<td><strong>Phase 2: Development</strong></td>
<td><strong>January – February 2023</strong></td>
</tr>
<tr>
<td>• Implementation (100%)</td>
<td>18 April 2023</td>
</tr>
<tr>
<td>• Evaluation (20%)</td>
<td>17-21 April 2023</td>
</tr>
<tr>
<td>Deliverable: Final Implementation; Final Report</td>
<td>3 May 2023</td>
</tr>
<tr>
<td>Deliverable: Final Presentation</td>
<td></td>
</tr>
<tr>
<td>Deliverable: Project Exhibition</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Project Schedule
Figure 4: Gantt Chart illustrating project schedule
References


