Technical University of Crete
Department of Electrical & Computer Engineering

Hayashi-Yoshida coefficient estimator implementnation in CUDA

Spyros Fotopoulos

Submitted in part fulfillment of the requirements for the degree of
Diploma in Electrical and Computer Engineering of the Technical University of Crete,
October 2017

Abstract

The increased need for fast computational systems that can process large streaming inputs in real-time, combined with the limits in parallelism of the CPU's and the difficulties of FPGA based designs have led the hardware community in the search for alternatives. In this thesis, we present a design that leverages the vast parallel capabilities of current GPU's that contain thousands of cores. We implement on a CUDA-enabled GPU two designs of the Hayashi-Yoshida correlation coefficient estimator algorithm, with and without window, for large inputs and we can yield an impressive fifteen-fold and twenty-fold decrease in overall computing latency respectively, coupled with the ability to handle simultaneously up to 18000 inputs.

Acknowledgements

I would like to thank my supervisor Professor Apostolos Dollas for the assignment of this diploma thesis and the guidance provided during its accomplishment. I would also thank Professor Minos Garofalakis and Associate Professor Yannis Papaefstathiou for being members of my diploma thesis committee.

I would like to express my sincere gratitude to Pavlos Malakonakis for the continuous support, for his patience, motivation, guidance and useful consults throughout the thesis stages. His help was crucial to the completion of this thesis.

I would also like to acknowledge Grigoris Chrysos for his insightful comments and help provided whenever needed.

Finally, I want to thank my family and my friends who stood by me through my studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General

Many industries are Big-Data driven, most notably, the financial industry. Today the use of Big-Data is widespread at various levels of this field, ranging from financial services to capital markets. The ability to efficiently handle and in minimum time process the available Big-Data is an essential topic in computer research.

Our work focuses on a specific metric used extensively by the financial sector, the correlation coefficient. The goal is to present a computational platform that can calculate in real time the correlation coefficient for Big-Data streaming inputs, using the Hayashi-Yoshida(HY) correlation coefficient estimator, implemented in parallel computing platform on a Graphics Processing Unit. By leveraging the vast parallel computational capabilities of current GPUs and their orientation in floating point operations, we aim to create a design that can offer significant improvement comparing to current solutions.

The application of our proposed implementation are mainly in the financial sector, and specifically in stock markets. Nevertheless, this work can be used to calculate the Hayashi-Yoshida correlation estimator of any large streaming inputs.

1.2 Thesis Organization

This thesis is organized in 6 chapters. In the first three, we introduce the fundamental concepts upon which we base our work and the in the following three we detail our proposed implementation.

Chapter 2 presents the importance of correlation and covariance in economics, the most common types of correlations used and a detailed presentation of the Hayashi-Yoshida correlation coefficient estimator.

In the following Chapter 3, we introduce the CUDA platform. The goal of this chapter is to explain to the reader how a CUDA device fundamentally works, the basic units it includes and their purpose as well as the basic terminology used throughout the text.

Chapter 4 gives a detailed presentation of our proposed design. We first present a CPU implementation that will be used as a performance reference. Then we present the additive formulation of the HY algorithm and the mapping we will use in all our versions. Finally, we present all the major versions we implemented and the reasoning behind the changes.

In Chapter 5, we illustrate the performance of each version with various metrics and compare them to previous versions as well as the CPU implementation. We discuss the effectiveness of the changes in each version and the limits of the designs.

In the last Chapter 6, we analyze the conclusions gathered from our work, and we propose some thoughts about future extensions of the design to both achieve better performance and implement a complete deployable system.

1.3 Related Work

This section presents existing research approaches related to hardware implementations of correlation coefficient algorithms. First, we present a hybrid FPGA-based platform that calculates the HY correlation coefficients. Next, we present three CUDA-based implementations of the

Pearson correlation coefficient which bears similarities on the mathematical structure of the HY algorithm and a framework for the calculation of the correlation on streaming data.

1.3.1 Hayashi-Yoshida Hardware Implementations

The work most related to ours is a part of the EU-funded project Qualimaster[18]. Particularly the publication Leveraging Reconfigurable Computing in Distributed Real-time Computation Systems[16] presents among others, an implementation of the HY correlation mapped on a Maxeler MPC-C Series reconfigurable high-end server into the Apache Storm, a well-known distributed platform. Particularly they implemented a sliding window version of the HY algorithm (more details about the specifics of the window implementation are provided at the following chapter). They achieved a performance calculating the windowed HY correlation coefficients for 5000 stocks in 0.869sec, although it is worth noting that a significant part of that time was used to process the I/O packets from and to the network.

1.3.2 Correlation Coefficients Hardware Implementations

Kijsipongse et al. [10] proposes parallel computing solution based on the hybrid MPI/CUDA programming for fast and efficient Pearson correlation matrix calculation on GPU clusters. This implementation uses a cluster of CUDA-enabled GPUs using MPI for load balancing, consisting of the Nvidia cards 8400GS, GTS250 and Tesla 1060C. The platform was tested under the load of 4000 inputs spread into three tiles of 2000x2000 elements and was able to achieve significant speedup compared to a single core Intel Quad Core 2.33 GHz, at 119.72sec, 6.47sec and 3.96sec for each card respectively.

Muyan-Ozcelik et al. [13] implements the Demons algorithm on CUDA. Demons is a widely used deformable image registration algorithm widely used to match medical volumes, and a part of the algorithm contains the Pearson correlation calculation. The results showed a significant improvement in the overall execution of the algorithm compered to an Intel Core2 6600s single and multi-threading solutions, although the correlation computation was only part and the input dimensions relatively small.

Chang et al. [1] presents a CUDA implementation of pairwise Manhattan distance and Pearson correlation coefficient, for genes analysis, on a Nvidia Tesla C870 GPU. They tested their implementation under various input size configurations and achieved speedup between x28.6 and 38.9 compared to an Intel Pentium D 3Ghz CPU with overall execution time between 90ms and 1500ms for the various configurations.

Green and Birk [3] presents a novel computation algorithm for the 2D covariance method, which dramatically reduces the computational complexity, both ALU operations and memory access, relative to previous algorithms. They propose an algorithm where every product is computed exactly once, and is then added to all the output-matrix indices to which it contributes. They implemented their algorithm on C and used OpenMP on powerfull multi-CPU systems and achieved significant speedup over the traditional algorithm.

Lastly, the Statstream[21] tool maintains multi-stream and time-delayed statistics in a continuous online fashion. This tool solves the correlation calculation problem in a scalable way and it gives a guaranteed response time with high accuracy.

Chapter 2

Covariance and Correlation

2.1 Covariance and Correlation in Modern Portfolio Theory

In 1952, Harry Markowitch presented his work on a theory of portfolio management[12], for which he was awarded a Nobel Prize in 1990. His work significantly changed the asset management industry and is still considered as cutting edge in portfolio management. He propounded that two main concepts should be considered in portfolio management, risk and expected return and diversification. He looked at the assets and their contributions to the overall portfolio risk, and not just the individual risk of each asset and noted that investors must be compensated for the risk that cannot be diversified away, but not for the risk that was diversifiable.

The main tools used to achieve diversification in portfolio management is the calculation of covariance and correlation of an individual asset before its inclusion in a portfolio and over the time that remains in it. In probability theory and statistics, covariance is a measure of the joint variability of two random variables. It measures how two random variables move together, in the same direction (a positive covariance) or opposite directions (a negative covariance). In the construction of a portfolio, it is important to attempt to reduce the overall risk by including assets that have a negative covariance with each other. By diversifying the assets based on negative covariance the overall risk of a portfolio is minimized.

Finding that two assets have a positive or negative covariance might not be a useful metric on its own since covariance can only measure the directional relationship between two assets. It cannot show the strength of the relationship. To determine the strength of that relationship we need to calculate the correlation. Correlation is a statistic that measures the degree to which two random variables move together. It follows the same sign as covariance and takes values between -1 and +1. A negative correlation shows that the two variables tend to move in different directions, a positive correlation shows that the two variables have a tendency to move in the same direction, and finally a correlation equal to 0 indicates that the two variables movement is not related. The strength of the relationship in each direction is demonstrated by the value it self, meaning that the closer the value is to the boundaries, the stronger the dependency. The correlation is an industry-standard technique in the financial domain that is widely known and well accepted.

2.2 Common Correlation Coefficients

Our work focuses on a specific type of coefficient, the Hayashi-Yoshida correlation coefficient estimator. However, there are several other types of correlation coefficients used in statistical analysis, depending on the input data and the types of scales.

2.2.1 Pearson Correlation Coefficient

The single most common type of correlation is the Pearson Product-Moment Correlation Coefficient, which measures the degree of relationship between two continuous, linearly related, normally distributed variables. A continuous variable is a variable, which can be measured along a line scale. The price of a stock for example is continuous because it can range along a line scale from about 1 euro to about 1000 euros a share. Linearity assumes a straight line realationship between each of the variables in and normal distribution assumes a bell-shaped curve.

It is defined as :

$$\rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y}$$

and the estimate :

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 (y_i - \overline{y})^2}}$$

2.2.2 Spearman Correlation Coefficient

Spearman rank correlation coefficient measures the degree of relationship for ranked-ordered data. It does not assume any assumptions about the distribution of the data and is the appropriate correlation analysis when the variables are measured on a scale that is at ordinal. Spearmans correlation is usually used in situations where there are a number of variables and they are all ranked by an independent continuous variable. It is defined as :

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

where d is the pairwise distances of the ranks of the variables xi and yi and n the number of samples.

2.2.3 Kendall Correlation Coefficient

Kendall's coefficient is a correlation coefficient that can be used as an alternative to Spearman's for data in the form of ranks. It is a simple function of the minimum number of neighbour swaps needed to produce one ordering from another.The main advantages of using Kendall's coefficient are the fact that its distribution has slightly better statistical properties, and that there is a direct interpretation of this statistics in terms of probabilities of observing concordant and discordant pairs[8]. It is defined as :

$$\tau = \frac{c-d}{c+d} = \frac{S}{\begin{pmatrix} n \\ 2 \end{pmatrix}} = \frac{2S}{n(n-1)}$$

where c is the number of concordant pairs and d is the number of discordant pairs.

2.2.4 Point-biserial Correlation coefficient

The point-biserial correlation gives an estimate of the degree of relationship between a dichotomous variable and a continuous variable. The variable can be either "naturally" dichotomous, for example whether a coin lands heads or tails, or an artificially dichotomized variable usually using the median value. Mathematically is equivalent to the Pearson Correlation. It is defined as :

$$r_{pb} = \frac{M_1 - M_2}{s_n}\sqrt{pq}$$

where M_1, M_2 are the mean of each group respectively, s_n the standard deviation for the entire test and p,q the proportion of cases in each group.

2.2.5 Coefficient of Determination

The coefficient of determination is the proportion of the variance in the dependent variable that is predictable from the independent variable(s). It is the square of the correlation between the predicted values of a variable and the actual values of the variable. It is a measure used in the statistical analysis that assesses how well a model explains the predicts future outcomes and can be thought as a percentage.

2.3 Hayashi-Yoshida Correlation Coefficient Estimator

2.3.1 General

The Hayashi-Yoshida correlation coefficient estimator[9] is a measurement of the linear correlation between two asynchronous diffusive processes, e.g., stock market transactions. This method is used to calculate the correlation of high-frequency streaming financial assets. It can correlate the share prices of markets in real time, which can be used for forecasting the variation of the corresponding values. This method does not require any prior synchronization of the transaction-based data hence free from any problems caused by it. The estimator is shown to have consistency as the observation frequency tends to infinity.

2.3.2 Hayashi-Yoshida Correlation and Covariance Coefficient Equations

The Hayashi-Yoshida covariance estimator is defined as follows:

$$HYcov = \sum (P^1_{t_i} - P^1_{t_{i-1}}) * (P^2_{t_j} - P^2_{t_{j-1}}) * 1_{\{[t_i, t_{i-1}] \cap [t_j, t_{j-1} \neq 0]\}} \tag{2.1}$$

where $P^1_{t_i}, P^1_{t_{i-1}}$*, the values of variable 1 at times* t_i, t_{i-1}*,respectively*

and $P^2_{t_j}, P^2_{t_{j-1}}$*, the values of variable 2 at times* t_j, t_{j-1}*,respectively*

This equation Eq(2.1) uses the product of any pair of values that will contribute to the sum only when the respective observation intervals are overlapping with each other. The proposed non-synchronous correlation estimator is computed by the Eq(2.2), which uses the covariance results as described in Eq(2.1).

$$HY cor = \frac{\sum (P^1_{t_i} - P^1_{t_{i-1}}) * (P^2_{t_j} - P^2_{t_{j-1}}) * 1_{\{[t_i, t_{i-1}] \cap [t_j, t_{j-1} \neq 0]\}}}{\sqrt{\sum_i (P^1_{t_i} - P^1_{t_{i-1}})^2 * \sum_j (P^2_{t_j} - P^2_{t_{j-1}})^2}} \tag{2.2}$$

where $P^1_{t_i}, P^1_{t_{i-1}}$, *the values of variable 1 at times* t_i, t_{i-1},*respectively*

and $P^2_{t_j}, P^2_{t_{j-1}}$, *the values of variable 2 at times* t_j, t_{j-1},*respectively*

The quantities at the denominator represent the realized volatilities calculated using raw data. As shown in Eq(2.2), the calculation of the Hayashi-Yoshida Correlation Estimator can be parallelized. In more details, the parallelization level of the of the formula in Eq(2.2) is hiding behind the parallel calculation of the quantities at the denominator and the nominator, as they refer to the transactions that take place in different time intervals. The problem is inherently parallel as the computation of correlation metrics for different pairs of stocks is entirely independent; thus they can be computed in parallel.

Chapter 3

CUDA

3.1 Intodoction

In the recent years, semiconductors and computer manufacture shifted their designs from single core processing units with high clock speeds to multi-core processing units. This change was implemented to overcome the increased fundamental limitations of fabricating integrated circuits which relied on frequency scaling as a means for extracting the additional needed performance. The most notable example of this turn in multi-core processing units is the GPU (Graphics Processor Units), where every architecture nowadays has thousands of cores. This vast, and relatively cheap, processing power that now exists in almost every system can be used for parallel processing in the GPU instead of the CPU(Central Processor Unit), with results that are many times better comparing to CPU implementations, especially algorithms that can be heavily parallelized.

CUDA is a parallel computing platform and programming model, consisting of the CUDA architecture on the GPU and a programming API, designed by NVIDIA and introduced in 2006, that allows software developers and engineers to use CUDA-enabled GPUs to perform computations in applications traditionally handled by the CPU. CUDA made the access to the processing power of the GPU easier to the parallel programmer by not requiring advanced skills in graphics programming and by using programming languages widely used such as C/C++, Fortran, and Python.

CUDA-enabled GPUs have differences in the fundamental architecture design comparing to CPUs since their goal is different. The design of a CPU is optimized for sequential code performance, by using sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of a sequential execution. Moreover, large cache memories are provided to reduce latencies of large complex applications, with the goal of minimizing the execution time of a single thread. Neither control logic nor memories contribute to the peak calculation speed.

The design of GPUs is driven by the fast-growing video game industry that demands cards that can perform a massive number of floating-point calculations per video frame. The designs focus on the execution throughput of massive numbers of threads. This way the hardware overlaps the time needed by some threads waiting for long-latency memory accesses with threads that can do calculations. Small cache memories are provided so that multiple threads that access the same memory data do not need to all go to the DRAM. This design is referred to as throughput-oriented since it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute.

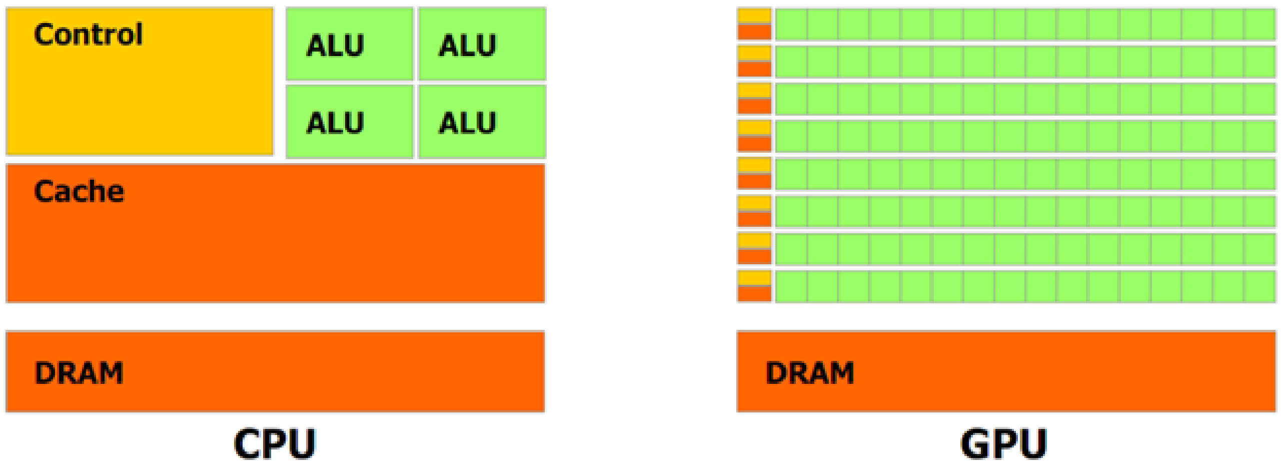

Figure 3.1: Different design philosophies

3.2 CUDA Terminology

Nvidia uses the following terminology which will be followed by this thesis. [1]

- **Device** is the CUDA-enabled GPU
- **Host** is the CPU of the system in which the GPU is installed
- **Device Memory/Global Memory** is the GDDR memory located on the GPU
- **Host Memory** is the RAM of the host system
- **Kernel** is a function executed in the CUDA-enabled GPU
- **Thread** is the fundamental set of instructions with independent set of values
- **Block/ThreadBlock** is group of threads, the maximum number of which is limited by the architecture
- **Grid** is the number of blocks in a kernel, the maximum number of which is limited by the architecture
- **Shared Memor**y is the memory allocated per block and accessible only by the threads of that block
- **Local Memory** is the memory used for everything that does not fit in the thread registers
- **Constant Memory** is where constants and kernel arguments are stored
- **Wrap** is a group of 32 threads that execute in parallel on hardware level

[1] We are using Compute Compatibility 5.x and Maxwell Architecture on a GTX 980

3.3 CUDA Architecture

3.3.1 Overview

CUDA GPUs are based on a scalable array of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMMs), and memory controllers. A GM204(GTX 980 core chip) consists of four GPCs, 16 Maxwell SMM, and four memory controllers. Maxwell GPUs feature from 6 to 16 SMM depending on the exact model and GTX 980 uses the full complement of these architectural components. In GeForce GTX 980, each GPC ships with a dedicated raster engine and four SMMs. Each SMM has 128 CUDA cores, a PolyMorph Engine, and eight texture units. With 16 SMMs, the GeForce GTX 980 ships with a total of 2048 CUDA cores and 128 texture units.CPU commands are read by the GPU via the Host Interface. The GigaThread Engine fetches the specified data from system memory and copies them to the frame buffer.

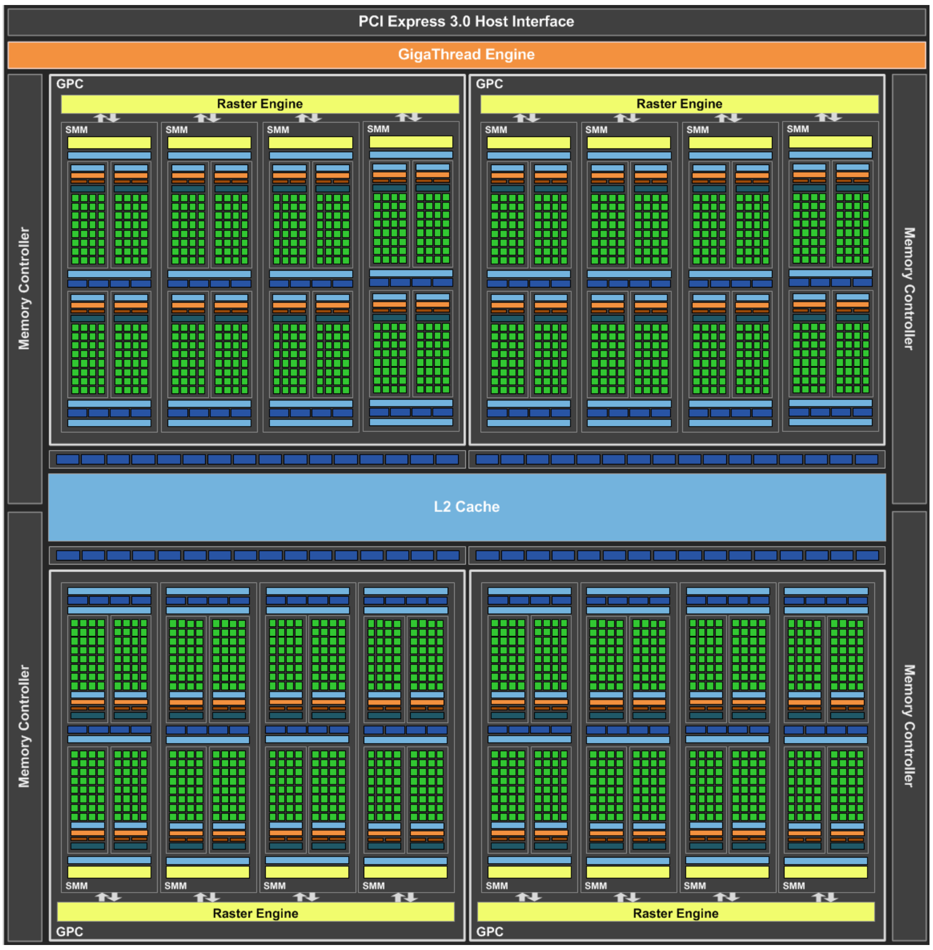

Figure 3.2: GM204 Full-chip block diagram

3.3.2 Graphics Processing Clusters

A GPC contains a raster engine and up to four SMMs, and as the name indicates it encapsulates all the key graphics processing units and can be thought as a self-contained GPU.

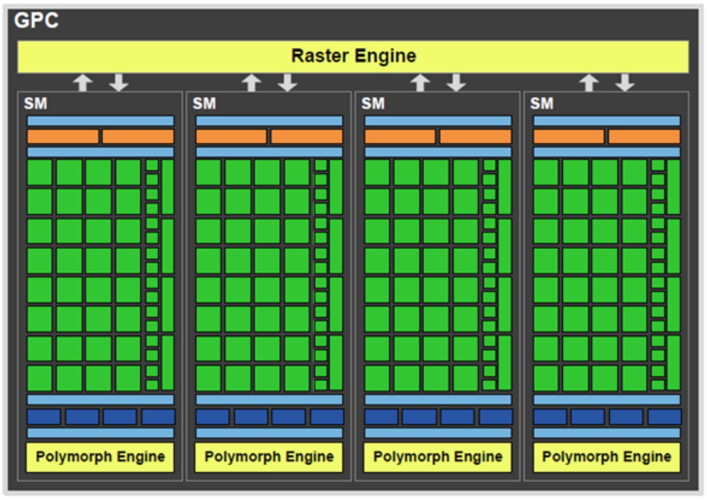

Figure 3.3: GM204 Graphics Processing Cluster (GPC)

The PolyMorph Engine has five stages: Vertex Fetch, Tessellation, Viewport Transform, Attribute Setup, and Stream Output. Results calculated in each stage are passed to an SM. The SM executes the games shader, returning the results to the next stage in the PolyMorph Engine. After all stages are complete, the results are forwarded to the Raster Engines After

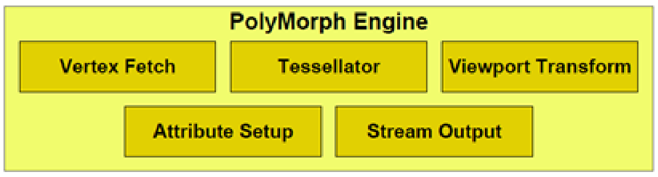

Figure 3.4: Polymorph Engin

primitives are processed by the PolyMorph Engine, they are sent to the Raster Engines. To achieve high triangle throughput, GTX980 uses four Raster Engines in parallel.

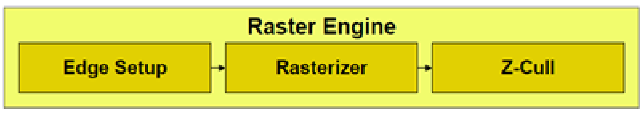

Figure 3.5: Raster Engine

3.3.3 Maxwell Streaming Multiprocessor

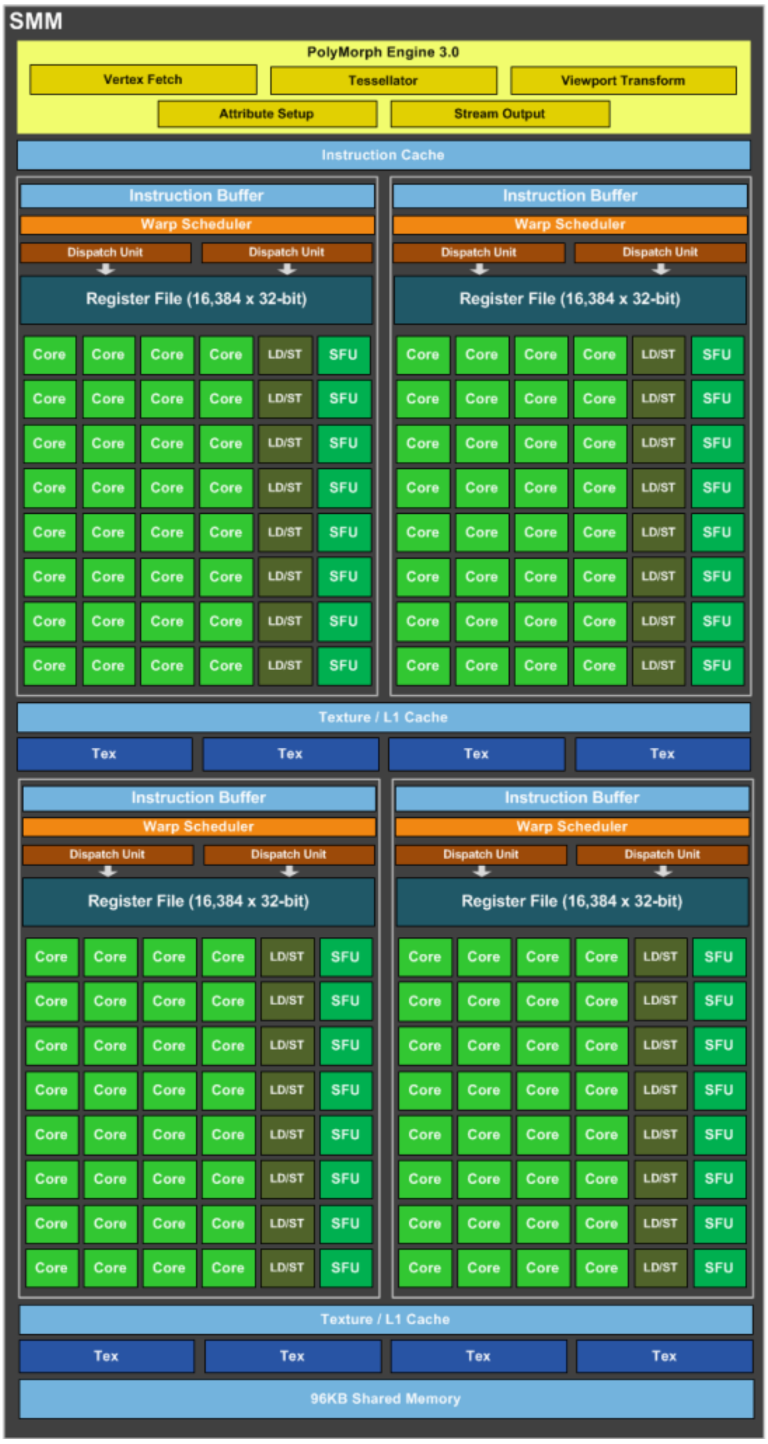

Figure 3.6: CUDA Streaming Multiprocessor

Each SMM features 128 single-precision CUDA cores partitioned into 32-CUDA core processing blocks, each with its own dedicated resources for scheduling and instruction buffering. Additionally every SMM contains 32 Special Function Units (SFU) and 32 load/ store units. Each CUDA core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Load/Store units allowing source and destination addresses to be calculated for sixteen threads per clock. SPU execute transcendental instructions such as sin, cosine, reciprocal, and square root but also graphics interpolation instructions. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

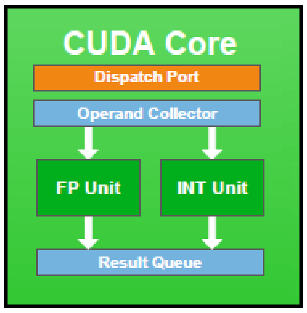

Figure 3.7: CUDA Core

3.3.4 Wrap Scheduler

The SMM creates, manages and executes threads in groups of 32 called wraps. Every thread in a wrap starts executing the same instruction but using independent instruction counter and registers thus every thread can deviate from the original path and continue to execute independently. This execution model is named Single Instruction Multiple Threads (SIMT), and it was introduced by Nvidia. Each SMM features four wrap schedulers and eight instructions dispatch units, allowing four warps to be issued and executed concurrently. The quad wrap schedulers select four warps, and two independent instructions per warp can be dispatched each cycle.

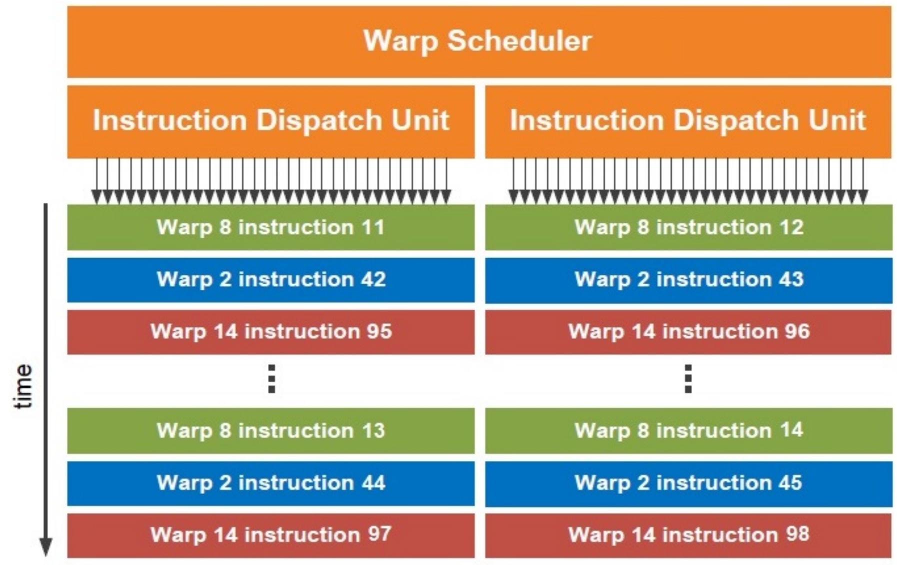

Figure 3.8: A single Warp Scheduler Unit

To achieve maximum throughput every thread in a warp must follow the same execution path. In case we have branching during the execution, the threads split into the ones that follow on path and continue to execute in serial while the threads on the other part of the warp wait until the branch execution finishes. The execution in serial and not parallel degrades the performance of the kernel. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. The threads of a warp that are on that warp's current execution path are called the active threads, whereas threads not on the current path are inactive (disabled). Threads can be inactive because they have exited earlier than other threads of their warp, or because they are on a different branch path than the branch path currently executed by the warp, or because they are the last threads of a block whose number of threads is not a multiple of the warp size.

The execution context (program counters, registers, etc.) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the active threads of the warp) and issues the instruction to those threads. In particular, each multiprocessor has a set of 32-bit registers that are partitioned among the warps, and a parallel data cache or shared memory that is partitioned among the thread blocks. The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor. These limits as well the amount of registers and shared memory available on the multiprocessor are a function of the compute capability of the device.

At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the latency, and full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely "hidden".

The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not available yet. If all input operands are registers, latency is caused by register dependencies, i.e., some of the input operands are written by some previous instruction(s) whose execution has not completed yet. In the case of a back-to-back register dependency (i.e., some input operand is written by the previous instruction), the latency is equal to the execution time of the previous instruction and the warp schedulers must schedule instructions for different warps during that time. If some input operand resides in off-chip memory, the latency is much higher (x100).Finally another reason a warp is not ready to execute its next instruction is that it is waiting at some synchronization point A synchronization point can force the multiprocessor to idle as more and more warps wait for other warps in the same block to complete execution of instructions prior to the synchronization point.

3.3.5 Thread Hierarchy

As stated previously, Grid is a group of blocks and Block is a group of threads. Grids,Blocks and Threads can be organized into a one-dimensional, two-dimensional, or three-dimensional structures. Each thread and each block have a unique ID that distinguishes them. The number of threads per block, the number of blocks per grid and the grid dimensions is specified in the kernel execution. Thread blocks are required to execute independently, and this independence requirement allows thread blocks to be scheduled in any order across any number of cores. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.

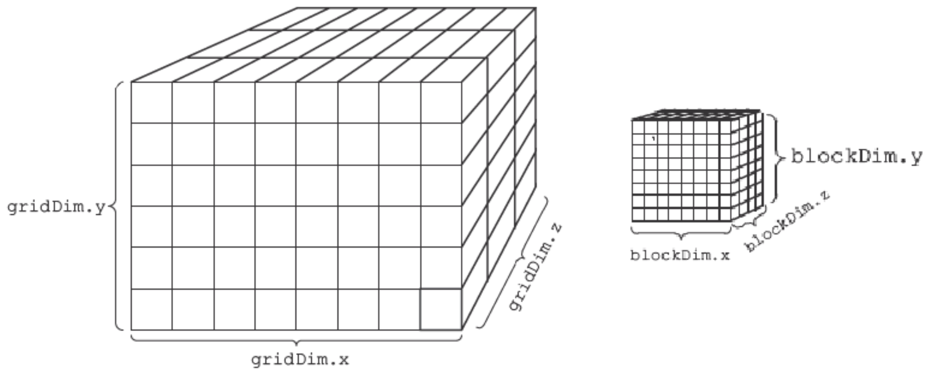

Figure 3.9: Grid of Thread Blocks

3.3.6 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory, shared memory visible to all threads of the block and with the same lifetime as the block, and access to the same global memory.There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages.

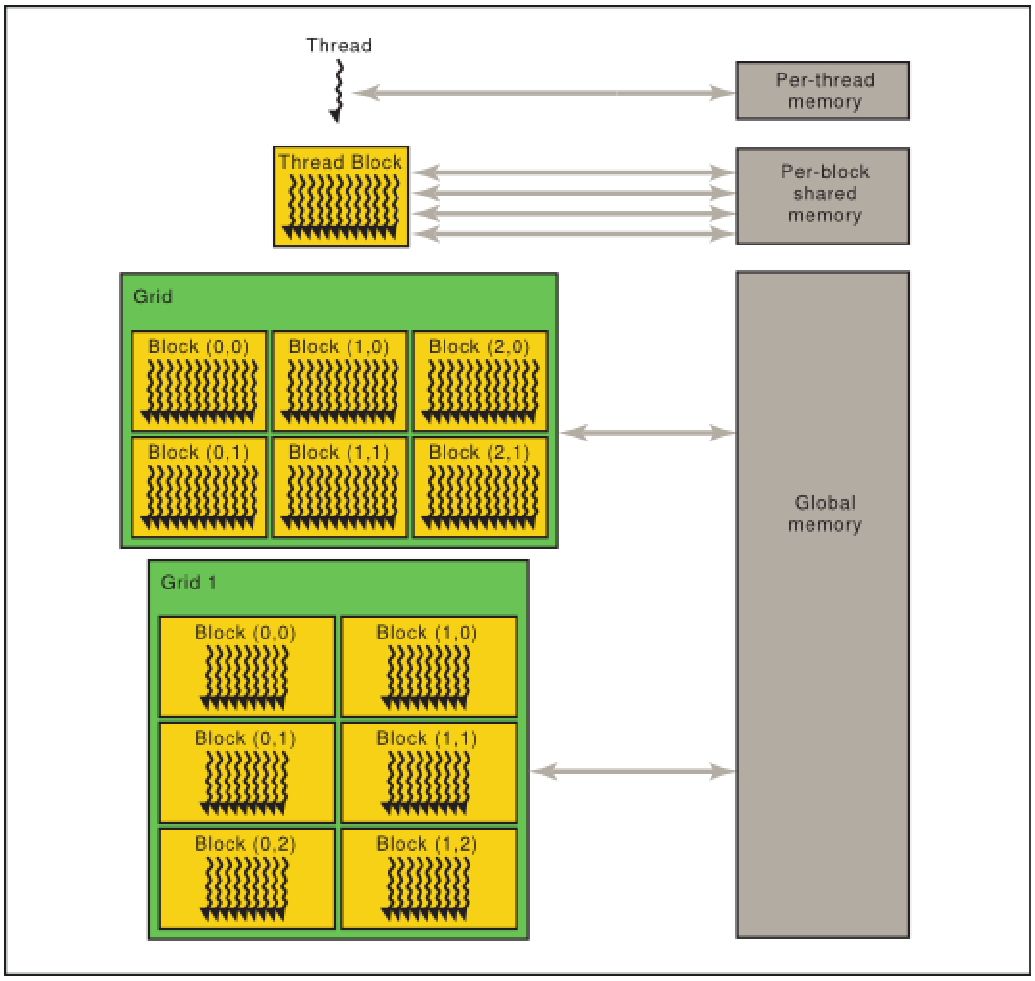

Figure 3.10: Memory Hierarchy

Chapter 4

Design and Implementation

4.1 General

The goal of this project was to design and implement a fully functional program that calculates the HY correlation coefficients estimator for a large number of stocks. In the development and testing process, various versions were created while trying to improve the functionality and performance of the program. Although the final version is the fastest, previous versions can provide useful information in understanding the way CUDA implemented design work and how they can be improved. Additionally we will present two variations of the algorithm, in order to have comparable results with previous works[16]. One implementation will calculate the coefficients for indefinite streaming data and the second will be a modified version where we only calculate the coefficients considering price changes in a selected size sliding time window.

4.2 Software&Hardware Used

For our software implementations and as a Host to our CUDA implementations we tested two systems:

1. System 1

- CPU: Quad Core Intel Core i7 3770 @ 3.4Ghz
- RAM: 16GB DDR3 1600Mhz
- M/B: Asus P8H77-V LE
- GPU: Nvidia GTX960 4GB
- OS: CentOS 7.3.1611

2. System 2

- CPU: Quad Core Intel Core i5 3570 @ 3.4Ghz
- RAM: 8GB DDR3 1600Mhz
- M/B: Asus P8Z77-V
- GPU: Nvidia GTX960 2GB
- OS: Windows 10

Our CUDA design are compiled using CUDA Toolkit 8.0 and the compute compatibility of our GPU's is 5.2.

	GTX960	GTX980
CUDA Cores	1024	2048
SMMs	8	16
Core Clock	1127Mhz	1126Mhz
L2 Cache	1024KB	2048KB
Memery	2GB GDDR5	4GB GDDR5
Memory Interface Width	128bit	256bit
Memory BandWidth	112GB/sec	224GB/sec
TDP	120W	165W

Table 4.1: GPU Specifications

4.3 Algorithm Analysis

As explained previously the HY estimator works for streaming financial data over time and in order visualize the algorithm we will use candid of 10 stocks in any random moment.

If now is the time moment t, we have the current stock prices and the previous $(t-1)$ prices for each of the 10 stocks as inputs. First, we must calculate the price difference between the current price and the price in the previous time interval for every stock. These differences will be used to initially calculate the HY covariance for each pair of stocks and the standard deviations of each stock. Finally, we will use both to calculate each pair's HY correlation. Both correlation and covariance values of each pair of stocks are updated with each cycle (new time interval) of the execution.

The result for the HYcov and HYcor can be visualized as two matrices of 10 by 10 dimensions where each field contains the correspondent result for each pair of stocks. There is no point in calculating the values of the whole matrix since the $\mathrm{Cov}_{i,j}=\mathrm{Cov}_{j,i}$ and $\mathrm{Cor}_{i,j}=\mathrm{Cor}_{j,i}$. Additionally, we can skip the calculation of the diagonal, considering that $\mathrm{Cor}_{i,i}$ and $\mathrm{Cov}_{i,i}$ are equal to 1 always. Therefore we can visualize the matrices as follows :

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S_0	–	$hy_{0,1}$	$hy_{0,2}$	$hy_{0,3}$	$hy_{0,4}$	$hy_{0,5}$	$hy_{0,6}$	$hy_{0,7}$	$hy_{0,8}$	$hy_{0,9}$
S_1		–	$hy_{1,2}$	$hy_{1,3}$	$hy_{1,4}$	$hy_{1,5}$	$hy_{1,6}$	$hy_{1,7}$	$hy_{1,8}$	$hy_{1,9}$
S_2			–	$hy_{2,3}$	$hy_{2,4}$	$hy_{2,5}$	$hy_{2,6}$	$hy_{2,7}$	$hy_{2,8}$	$hy_{2,9}$
S_3				–	$hy_{3,4}$	$hy_{3,5}$	$hy_{3,6}$	$hy_{3,7}$	$hy_{3,8}$	$hy_{3,9}$
S_4					–	$hy_{4,5}$	$hy_{4,6}$	$hy_{4,7}$	$hy_{4,8}$	$hy_{4,9}$
S_5						–	$hy_{5,6}$	$hy_{5,7}$	$hy_{5,8}$	$hy_{5,9}$
S_6							–	$hy_{6,7}$	$hy_{6,8}$	$hy_{6,9}$
S_7								–	$hy_{7,8}$	$hy_{7,9}$
S_8									–	$hy_{8,9}$
S_9										–

Table 4.2: HY Matrix for Covariance/Correlation

The upper (or the lower) triangle of a matrix(nxn) contains $\frac{(n*n)}{2}$ values, and by abolishing the diagonal, the values are:

$$\frac{n*(n-1)}{2}$$

Thus with our initial arguments in every new time interval, we must calculate 10 price differences, 10 standard deviations, 45 HY covariance coefficients and 45 HY correlation coefficients. As we can see the algorithm calculations is split into two different sets with different computational complexities, one set follows the O(n) and the other set follows the $O(\frac{n*(n-1)}{2})$. It

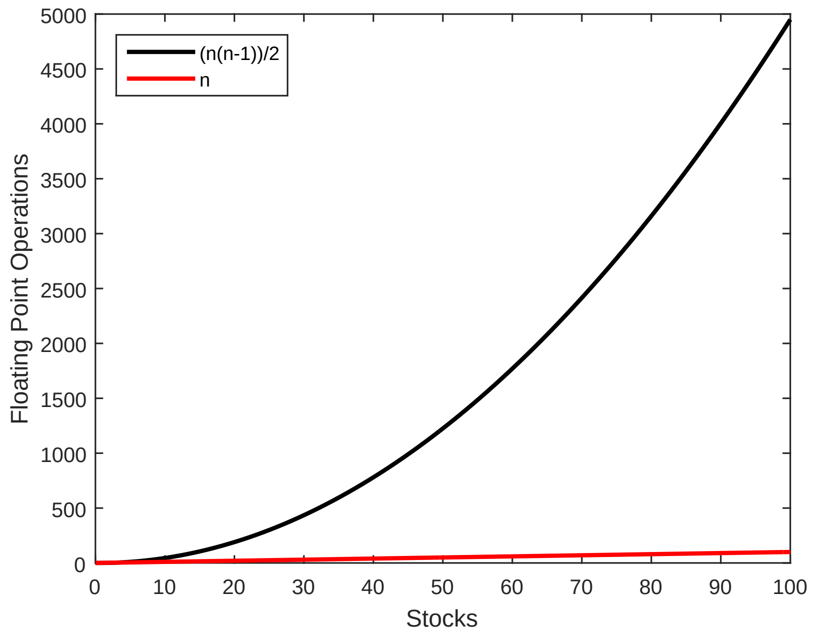

Figure 4.1: Big-O Complexity Comparison

is evident that a design completely executed on CUDA Device it would require two separate kernels, one for each set. Such a design would be inefficient since the first set of calculations is relatively small compared to the second, especially for a large number of stocks[1], and can be executed by the Host CPU very fast comparing the second set. Launching a separate kernel for those calculations would not only increase the overall execution time (our test showed that it requires more time to transfer the data to the device than the execution of the kernel itself), but could not provide any significant speed up considering that for a large number of stocks, the Host-side calculations account for an insignificant fracture of the overall execution time.

The design we will implement, in every new timer interval, calculates the first set in the Host CPU (differences and standard deviations) and then passes those values to the CUDA device, which calculates the matrix values and then return the results to HOST.The calculations of standard deviation and price difference will be calculated only one time in the host CPU and then re-used for all the corresponding pairs by the CUDA Device.

[1]To get a perspective of the difference in complexity for 5.000 stocks the first set is 5.000 floating point operations for price difference and standard deviation and 12.497.500 floating point operations for the the two matrices

4.4 Versions

4.4.1 Software Implementation

Initially, we designed and implemented a software implementation of the algorithm executed solely by the CPU, to fully understand how the algorithm works. This version was additionally used to confirm the accuracy of the following implementations in CUDA and can provide base results to compare them, acting as a proof of concept. Moreover, we executed a version of the software implementation using OpenMP(Open Multi-Processing) API to utilize the multi-thread capabilities of our CPU and understand the parallelization of the algorithm. First of all, we created the needed memory structures both in the Host Memory. We created two *nxn* matrices that will store the HY covariance and HY correlation pairs and *n* size matrices to store the standard deviation and price difference of the stocks.

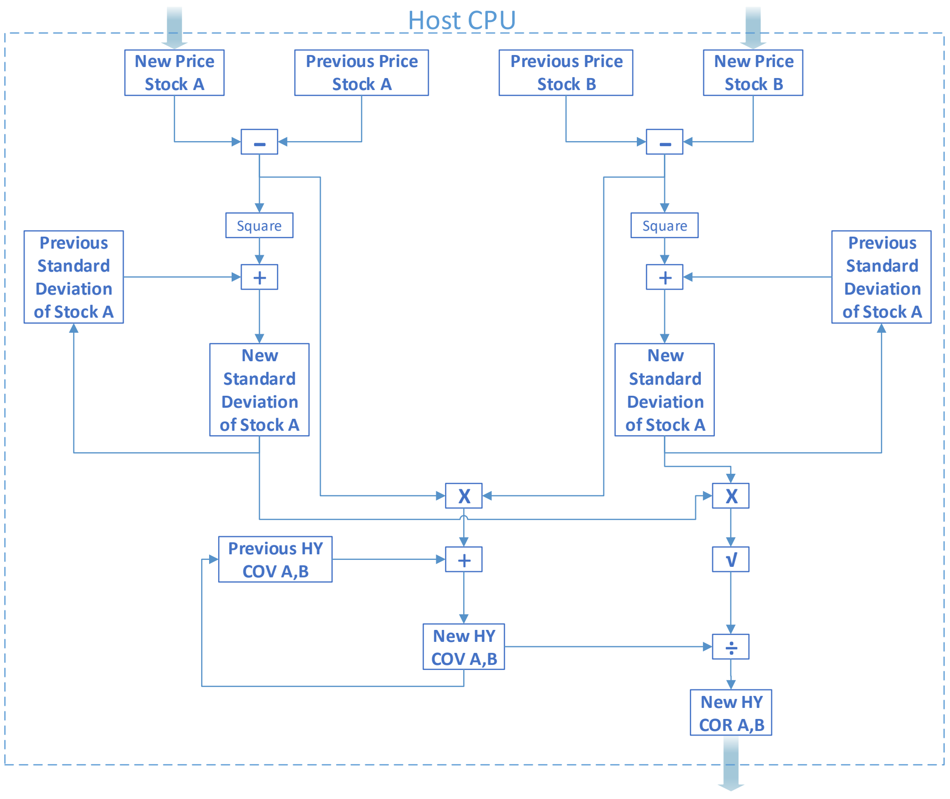

Figure 4.2: Software Architecture for two stocks

	Threads			
Stocks	1	4	8	16
1024	6.48	3.25	3.49	22.65
2048	23.81	3.25	10.21	48.45
5000	140.41	50.06	50.2	143.73
8000	358.71	127.21	122.59	278.47
10000	560	193.61	188.87	387.98
15000	1258.89	434.17	411.49	725.21
18000	1812.92	617.04	591.39	969.74

Table 4.3: Average Software calculations time in ms
Intel Quad Core i7-3770@3.4Ghz

4.4.2 CUDA Version 1

This implementation of the HY algorithm on CUDA is the simplest approach to the problem. We did not try to optimize the code, as our main goal was to get a working prototype implementation in CUDA that will act as a guide for final improved versions.

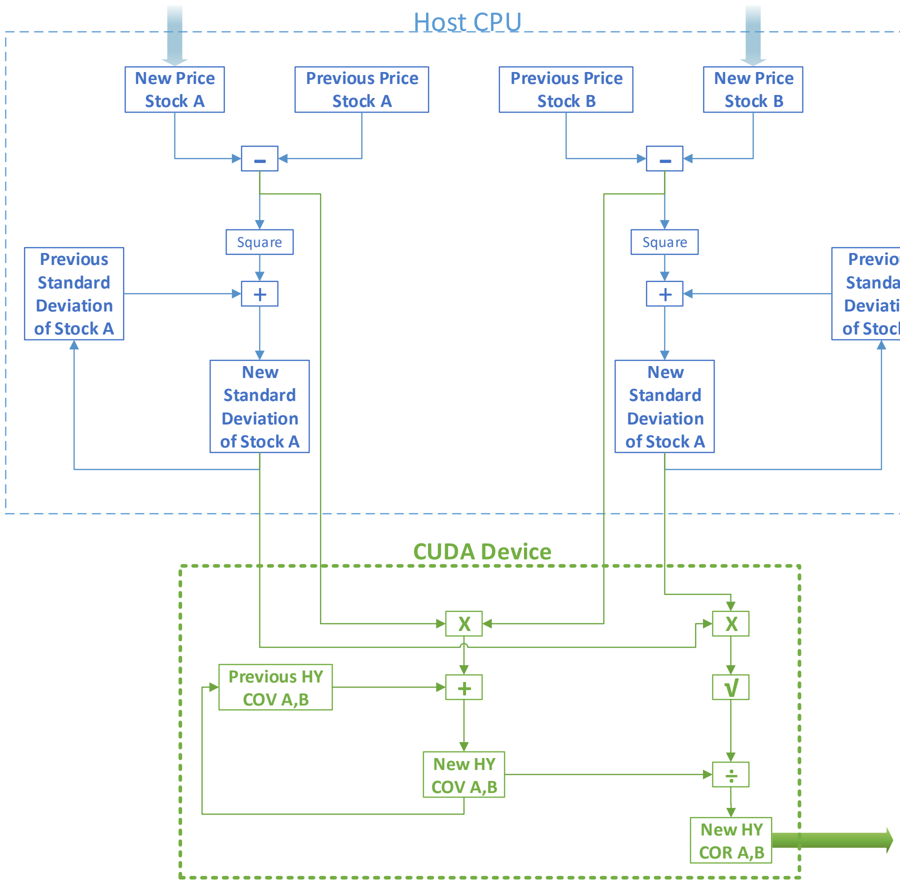

Figure 4.3: CUDA Design Architecture Version 1 for two stocks

At first, we created the Device Memory counterparts of the memory structures we created in the software version. Two *nxn* matrices for HY covariance and HY correlation pairs and *n* size matrices to store the standard deviation and price difference of the stocks were allocated in the Global Memory.

In every new time interval, the new stock prices are first processed by the Host, to calculate the standard deviation and the difference of each stock and then transferred to their counterparts on the Device. After copying the values to the Device, it calculates the values of the two matrices but only returns the values of the HY correlation matrix by copying it from the Device memory to the Host memory. This round-trip from the Host to the Device will repeat in every new time interval as long as data are provided and the time passed for each round-trip is the overall execution time that we will measure.

Inside the kernel, we designed the algorithm so that every block contains threads that calculate the HY coefficients of the same line of the matrices in memory. This approach is used so that every thread in a block can take advantage of the shared memory provided in each block, by using the memory values that are common for every pair calculation of the same line. The calculation of pairs of the same line in the matrix will have always common memory values needed as we know from the HY equations.

For example, in order to calculate the HY coefficients of pair(1,2) and pair(1,5) we use both times the values of price difference of stock 1 and standard deviation of stock 1. By making available in the shared memory of the block, we save for the calculations each thread the 100 cycles needed to access the Global Memory for each of those values.

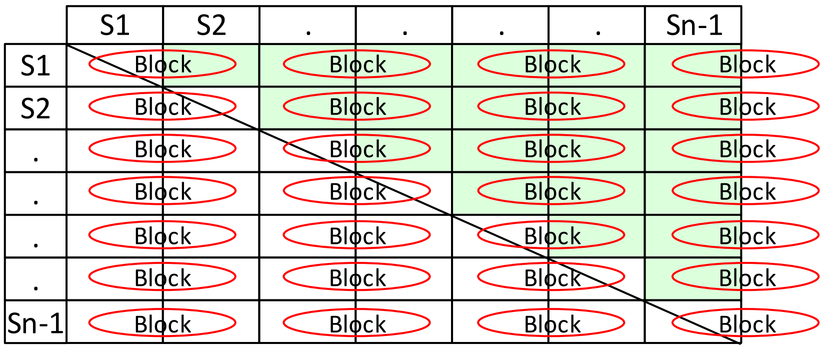

Figure 4.4: CUDA Block Architecture Version 1 for two stocks

For the block size, we used the maximum given by the architecture of the Device, 1024 threads per block. To calculate the number of blocks we need, meaning the grid size, we multiplied the number of blocks that will be needed for each line of the matrices with the number of stocks we have.

The following times were measured[2]

	GTX 960		
Stocks	First Result	Average	Maximum
1024	2.26	0.98	1.53
2048	7.49	3.75	4.46
5000	39.87	21.71	25.71
8000	102.65	54.05	57.41
10000	157.52	84.63	90.08

	GTX 980		
Stocks	First Result	Average	Maximum
1024	1.93	0.74	0.77
2048	6.10	2.77	2.83
5000	30.51	16.03	16.71
8000	76.68	40.76	43.97
10000	117.75	62.73	63.44
15000	266.93	142.50	143.02
18000	385.67	204.56	205.44

Table 4.4: Version 1 calculations time in ms

In this version the are not any pre-calculations required, and the first result is delayed because of the initial memory copies of the matrices to the device. The GTX980 can process inputs up to 18000 stocks thanks to the double sized device memory compared to the GTX960.

4.4.3 CUDA Version 2

The first design, as we explained, was using the same number of blocks for every line of matrices, without considering the fact that our goal is to calculate the coefficients of only one

[2]To measure the execution time of each implementation we use synthetic data inputs and the key times measured are from samples of 100 time intervals.

rectangular of the matrix. Even though the blocks corresponded to pairs that did not execute any calculations, they contained threads that had to be mapped and scheduled thus increased the initialization overhead of each kernel and the workload of the wrap scheduler, resulting in increased overall execution time. Hence our main goal in this implementation is to reduce the number of blocks used to the minimum required by removing any blocks that did not align with any coefficient pair.

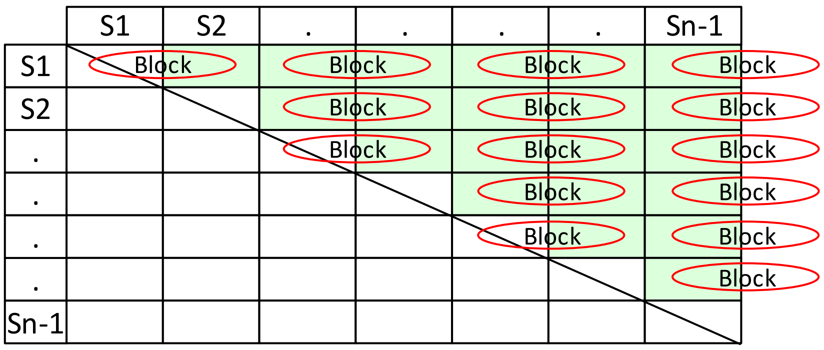

Figure 4.5: CUDA Design Architecture Version 2 for two stocks

The main problem what we had to overcome in this design is that, if the number of blocks in every line is not constant, we must add branch logic to the kernel to determine which block executed corresponds to which pairs of the matrices. Adding branch logic in CUDA even though is possible, it is a bad design philosophy because it destroys the parallelization of the load.

Our solution is to implement on the Host, before the call of the kernel for the first time, a function that determines how many blocks correspond to every line of the matrix factoring the input arguments(number of stocks, the maximum number of threads per block). The results of the function are then copied to the device and inside the kernel are used to calculate the correspondence of every block without any additional branching.

The function used takes a respectively large time to finish, creating a significant delay to the initialization time of the implementation, but this delay is paid only once[3]. Additionally, since we increase the amount of data we copy for the first call of the kernel the first result, this adds to the time needed for the first result.

[3]Pre-calculation time is host CPU dependant and the GTX 960's host system is inferior.

	GTX 960			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	131143	3.73	0.96	1.36
2048	131348	9.91	3.61	4.01
5000	132070	49.71	20.71	21.50
8000	556183	128.26	52.72	54.39
10000	556183	202.75	84.47	92.55

	GTX 980			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	379	1.88	0.73	0.90
2048	1314	6.83	2.71	2.87
5000	7859	37.38	15.48	15.57
8000	77226	97.69	39.81	40.82
10000	118280	149.86	61.96	62.73
15000	78130	325.51	138.98	140.64
18000	119010	466.62	199.24	204.75

Table 4.5: Version 2 calculations time in ms

4.4.4 CUDA Version 3

In this implementation, we wanted to test a technique that optimizes data transfers in CUDA, pinned memory, and possibly improve the overall execution time knowing that a big part of it is caused by data transferred between the host system and the device.

Host CPU data allocations are pageable by default, and the device cannot access data directly from pageable host memory.When a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or pinned, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory. Pageable memory is stored not only in RAM but can be swaped to the hard drive (virtual memory), if the O/S needs more RAM available. However, with current technology of memories the use of virtual memory is no longer necessary in most cases, since the size of RAM available usually sufficient.

As you can see in the figure bellow, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

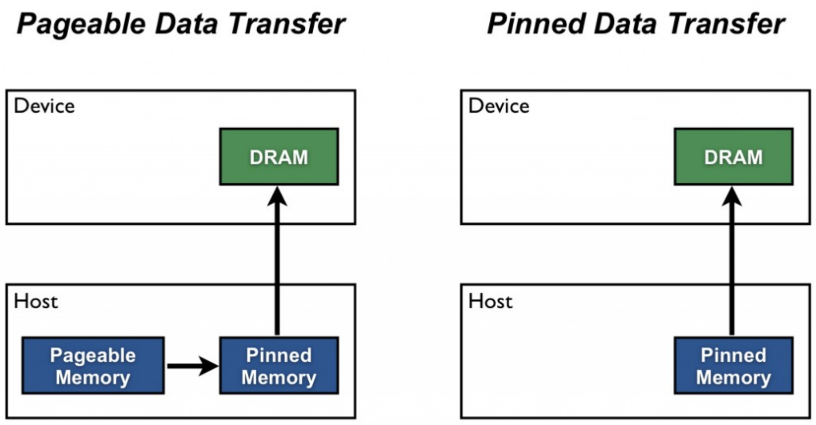

Figure 4.6: Pageable and Pinned Memory

The use of pinned memory when allocating the necessary memory on the host, enables the design to save the time needed for the transfer of the data from a pageable memory area to a pinned area.

	GTX 960			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	131107	3.33	0.57	1.05
2048	131203	6.51	1.90	2.41
5000	132237	25.07	10.68	11.08
8000	523533	61.33	26.93	27.16
10000	538026	91.16	41.74	41.95

	GTX 980			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	373	1.04	0.42	0.43
2048	1302	3.74	1.57	1.64
5000	7984	21.36	9.12	9.20
8000	77188	57.98	23.50	23.75
10000	118600	89.28	37.15	37.53
15000	259848	194.24	81.97	82.42
18000	372155	278.04	117.97	118.67

Table 4.6: Version 3 calculations time in ms

4.4.5 CUDA Version 3 Window Implementation

Our final implementation is based on the previous version but with a differentiation of how we implement the time intervals of the algorithm. The HY correlation coefficients can provide information about the strength of the relationship between two assets for an indefinite time, but can also give us information about how this relationship is fluctuates considering only shorts periods of time using sliding time windows.

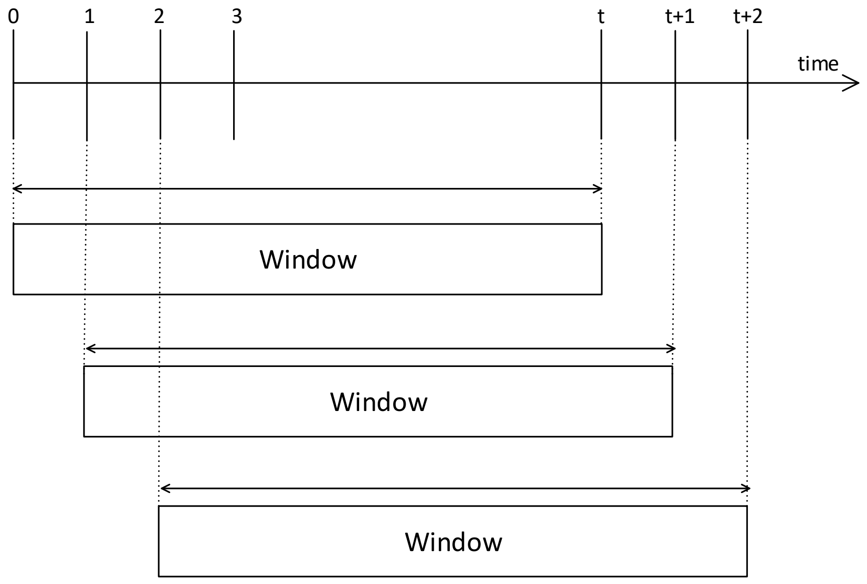

Figure 4.7: Sliding Window

At the start of the execution, we define the desired size of the window which slides forward in every new time interval. The size of the window determines the count of previous time instants that will be considered in the calculation of the HY coefficients. The new values that in the forward time movement come as inputs, are used to calculate the current price difference and then get stored in a FIFO structure in memory. The FIFO structure created can be modeled as an array, that can store the as many previous price differences of each stock as the size of the window. The differences that in forwarding time movement slide off the window are considered expired, and we revise their contribution to the estimator, by removing their effect on the corresponding sums, and we discard all relevant to it information.

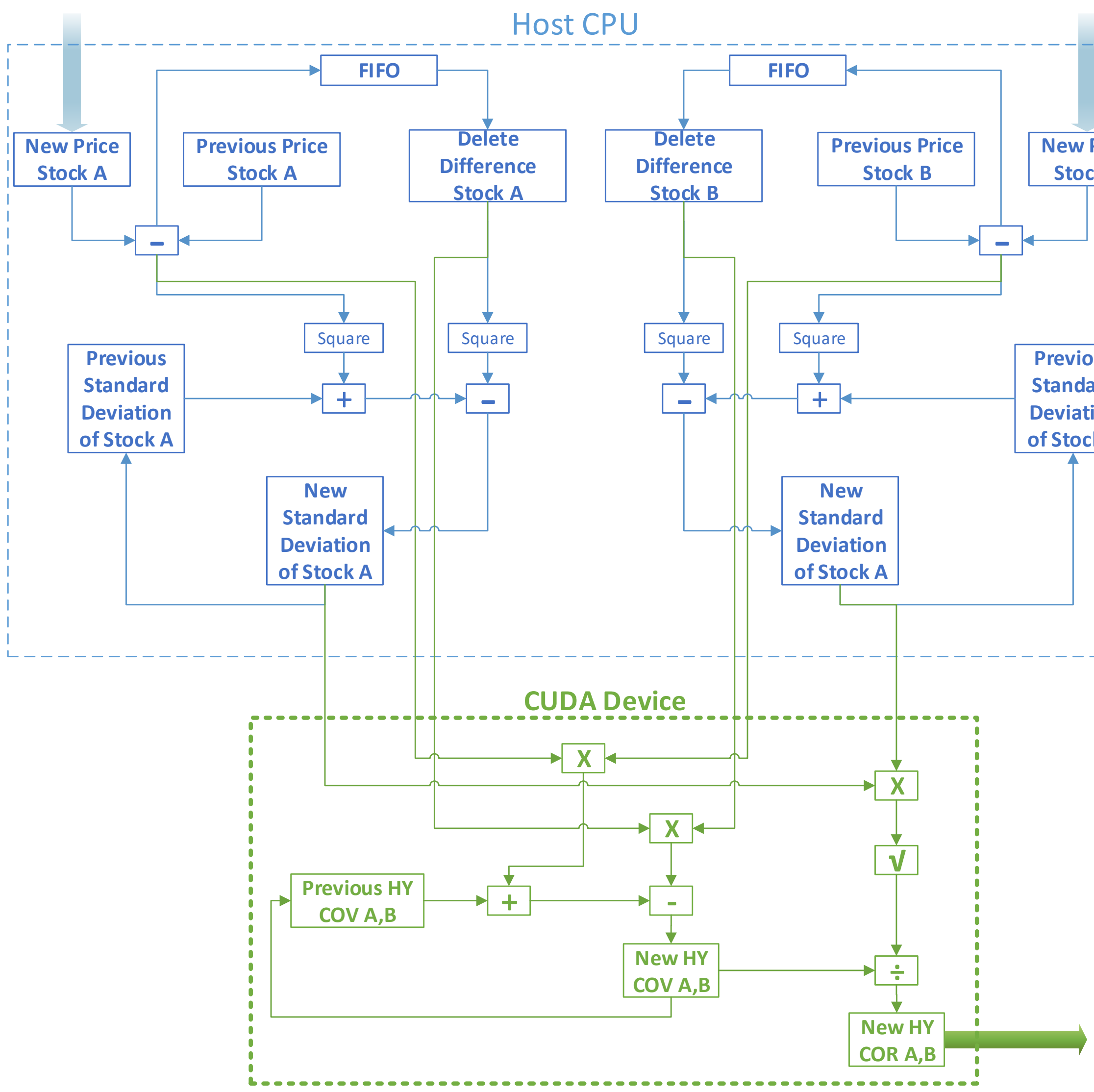
Host CPU
FIFO
FIFO
New Price Stock A
Previous Price Stock A
Delete Difference Stock A
Delete Difference Stock B
Previous Price Stock B
New Price Stock B
Square
Square
Square
Square
Previous Standard Deviation of Stock A
Previous Standard Deviation of Stock A
New Standard Deviation of Stock A
New Standard Deviation of Stock A
CUDA Device
X
X
X
√
Previous HY COV A,B
+
-
New HY COV A,B
÷
New HY COR A,B

Figure 4.8: CUDA Design Architecture Version 3 with Window for two stocks

	GTX 960			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	131281	3.37	0.67	1.17
2048	131411	6.53	2.04	2.48
5000	132318	25.11	10.97	11.36
8000	533892	65.37	27.57	27.86
10000	538178	94.62	42.95	43.30

	GTX 980			
Stocks	Pre-Calculations	First Result	Average	Maximum
1024	376	1.11	0.47	0.48
2048	1313	3.80	1.67	1.90
5000	7858	21.57	9.34	9.55
8000	77187	58.47	23.96	24.28
10000	118600	89.28	37.15	37.53
15000	260267	196.30	82.95	83.73
18000	374025	279.32	119.25	119.72

Table 4.7: Version 3 Window calculations time in ms

For comparison we implemented a software version of the window version of the HY algorithmn and we measured the following times:

	Threads			
Stocks	1	4	8	16
1024	8.11	4.25	4.34	23.52
2048	30.69	13.73	12.27	51.10
5000	181.17	75.74	64.85	161.51
8000	463.22	186.56	156.13	319.68
10000	722.94	291.46	242.46	452.72
15000	1627.18	645.37	557.10	882.42
18000	2340.66	928.82	760.89	1205.55

Table 4.8: Average Software Window calculations time in ms
Intel Quad Core i7-3770@3.4Ghz

Chapter 5

Results and Performance Analysis

In this chapter we are going to present an analysis and comparison of our results for all the versions of our implementation, explain the reason behind the changes in average execution time of each version and highlight on key performance factors.

5.0.1 First and Second Version

Our first version, even with the wasteful manner it deploys blocks, shows significant improvement compared to both single and multi-thread implementations in CPU. These results confirmed our initial hypothesis that an algorithm that can be heavily parallelized such as the HY correlation coefficient estimator can gain significant performance when implemented in CUDA.

The appropriate use of block resources in the second versions showed only a minor improvement in total execution time, and even though the gains increase with the increase of inputs calculated, it only accounts for a small part of the total execution time. Nevertheless, it allowed as to eliminate one of the obvious slowing factors and proved that allocating and mapping excessive thread resources that in theory should not slow the design since they are empty threads, can hurt the performance of the design even if in our case it concerns only a small part of the total latency.

The amount of threads allocated has an impact on the time required to initialize a kernel and during the kernel execution every thread, even without a computational content, consumes GPU cycles for its schedule in and schedule out procedures. The reason why the gains in total execution time are so small in this version is that from the total latency in calculating the HY correlations, the kernel initialization overhead and the kernel execution time proved to be relatively small part of the total thus optimizing it can only yield low results.

As we can see in the graph bellow, even though there is significant difference between the CUDA implementation and software implementation, the gains from the second version are so small that are difficult to represent graphically.

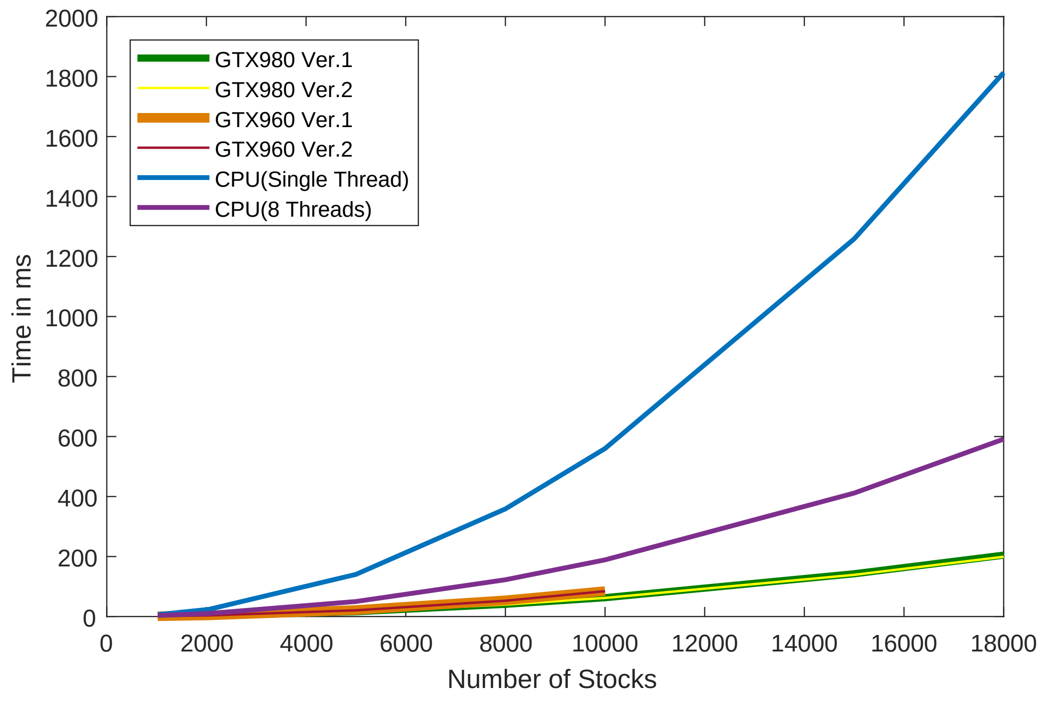

Figure 5.1: Versions 1&2 Comparison

5.0.2 Pinned and Pageable Memory

The difference between the second and the third version is the use of pinned memory. This change was implemented because we noticed that our implementation,when executed on both the 980GTX and the 960GTX, is memory bound, with the memory copy of the results from the device back to the host, taking most of the overall execution time. In CUDA implementations that due to the nature of the algorithm we cannot parallelize the memory transfers, the only tool provided to reduce the memory transfer's latency is the use of pinned memory. This way we can take advantage of the bandwidth provided by the host system's PCI-e ver.3.0 x16 interface achieving a bandwidth that is close to the maximum achievable on of this device.

The graph below represents the increase in bandwidth on the memory copy of the results from the device to the host for 5000 stocks, which is 95.4MB large. The initial transfer to the device at the kernel launch reach a much lower bandwidth due to their small size of 19.5KB, therefore, they do not capitalize on the use of pinned memory, but they do not add any significant latency either.

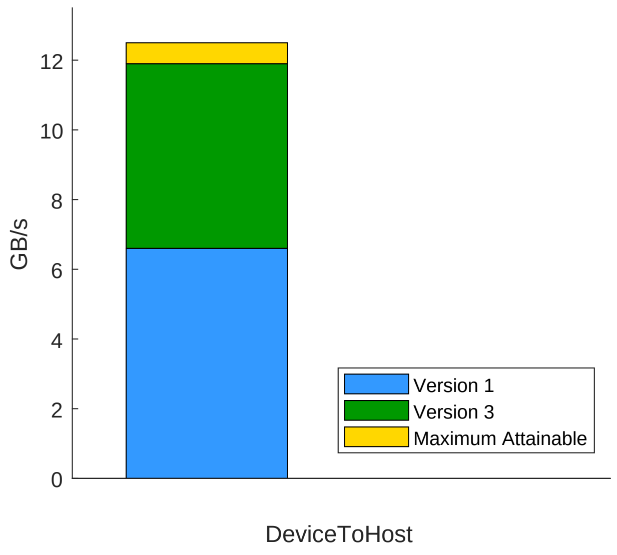

Figure 5.2: Achieved Bandwidth DeviceToHost

The memory copy of the results from the device to the host still is the bound of our implementation but now it represents 7% less of the total execution time of the kernel.

	Version	
	Version 1	Version 3
Host to Device	<1%	<1%
Kernel	8%	15%
Device to Host	92%	85%

This bandwidth increase significantly reduces as expected the overall execution time, for both GPU's, and the reduction is larger as the number of inputs increases since the results matrix is getting a squared increase in size.

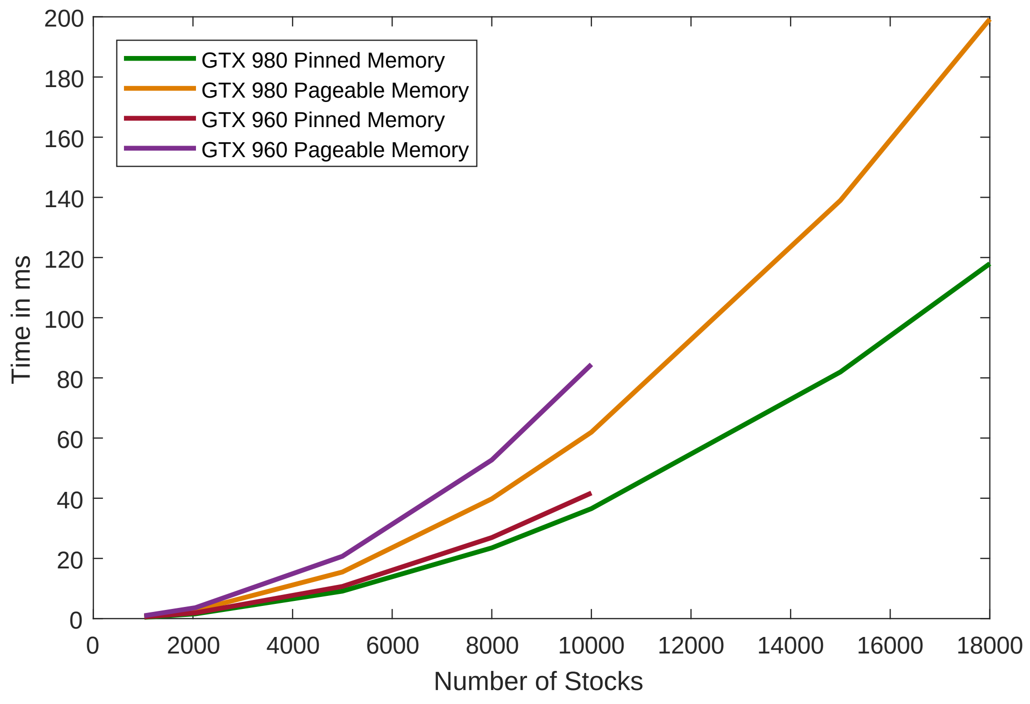

Figure 5.3: Pageable vs Pinned Implementations

5.0.3 No Window Implementations Comparison

The final version of our implementation on the GTX960 and the GTX980 provided close results up to inputs of 5000 stocks, and then until the GTX960 cannot go any further, due to it's memory size, we see that the gap is growing. This is the load point where the differences in specifications and compute capabilities of each card have a significant impact on the measured performance. It is almost sure that if we were able to test a GTX960 4GB version, this trend would continue and the gap would only get larger but still the GTX960 would be faster than the software implementation.

The latency difference between the software and CUDA implementations is exponentially growing as we increase the number of inputs to the limits of each GPU's memory. Even though the multithreaded software version showed significant improvement over the single thread execution, the CUDA implementation on both GPU's proved to be significantly faster when handling small and large loads.

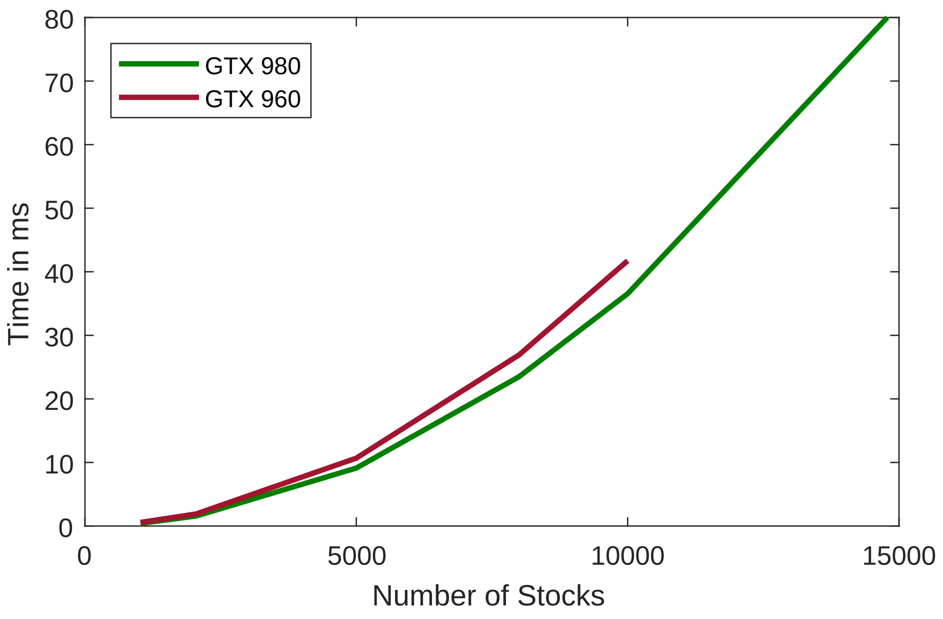

Figure 5.4: GTX980 compared to GTX960

	GTX 960 vs i7 3770		
Stocks	1 Thread	4 Threads	8 Threads
1024	x11.4	x5.7	x6.1
2048	x12.5	x5.5	x5.4
5000	x13.1	x4.7	x4.7
8000	x13.3	x4.7	x4.6
10000	x13.4	x4.6	x4.5

	GTX 980 vs i7 3770		
Stocks	1 Thread	4 Threads	8 Threads
1024	x15.3	x7.7	x8.3
2048	x15.2	x6.6	x6.5
5000	x15.4	x5.5	x5.5
8000	x15.3	x5.4	x5.2
10000	x15.3	x5.3	x5.2
15000	x15.4	x5.3	x5
18000	x15.2	x5.2	x5

Table 5.1: Speed Up using the Final Version

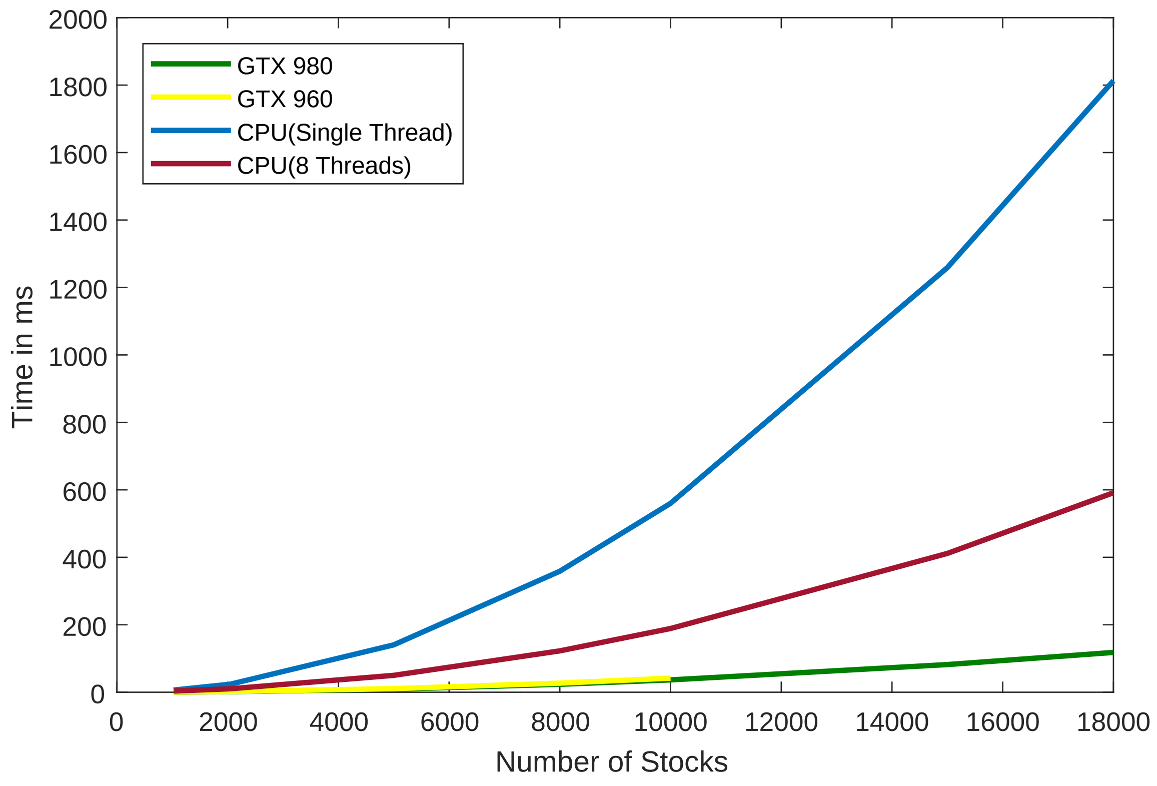

Figure 5.5: Final CUDA Version to Software Comparison

If we extract the latency added from the kernel initialization adn the memory transfers, to and from the device, we can compare the strictly computational latency of the GPU implementation compared to the software implementation.

	GTX 980 vs i7 3770		
	GTX 980	1 Thread	8 Threads
Latency in ms	1.263	50.06	50.2

Table 5.2: Computational latency for 5000 inputs

Finally we can compare the achieved throughput of both the software and the CUDA implementations, compared to their theoretical values.

	Theoretical	Achieved
GTX 980	4612 GFLOP/s	281 GFLOP/s
CPU (1 Thread)	108.8 GFLOP/s	0.71 GFLOP/s
CPU (8 Threads)	108.8 GFLOP/s	2 GFLOP/s

Table 5.3: Throughput GTX 980 for 5000 stocks

5.0.4 Window Implementation Comparison

The window version of the algorithm increases the number of floating-point operations of each time interval greatly. That proved to add a significant latency to the software implementations, single and multi-threaded compared to non-window implementations in software, but the latency of the CUDA implementation was only increased by a fraction. As a result even though we increased the total amount of floating point operations, the throughput increased.

Those results show that in a CUDA implementation, the execution time is usually bounded by the overheads of the data transfers from and to the host, accesses to the device memory, logic branching and initialization overhead. The actual quantity of floating point operations executed inside each CUDA core, provided that we can parallelise the calculations, accounts for only a fracture of the overall execution time.

GTX 980	No Window	Window	Difference(%)
SP Floating Point Operations	351,164,340	376,159,340	+6.87%
Throughput	281 GFLOP/s	297 GFLOP/s	+5.54%

Table 5.4: Throughput GTX 980 for 5000 stocks

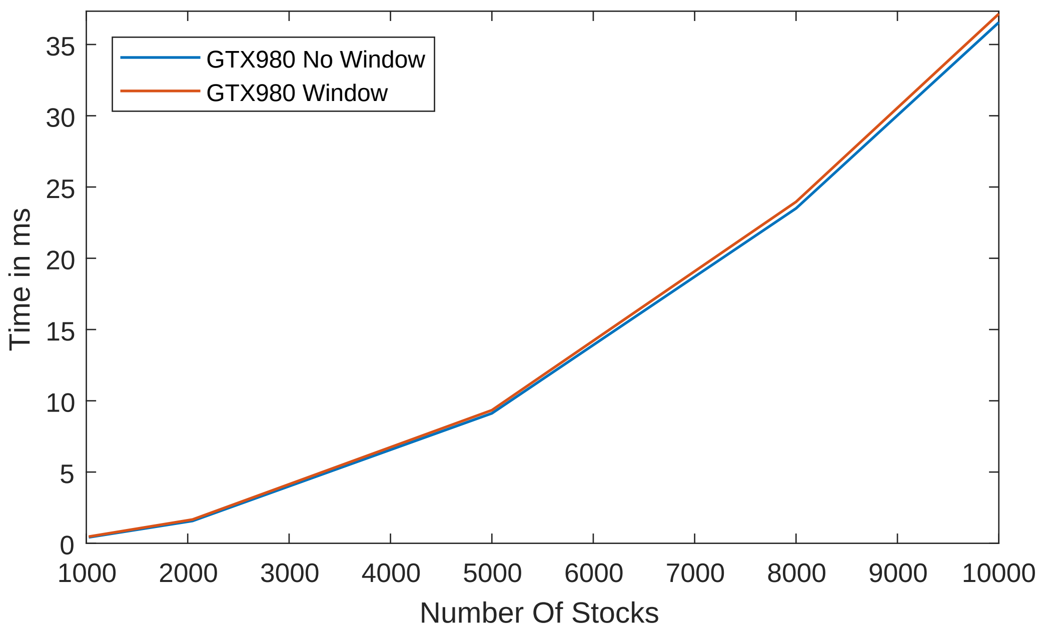

Figure 5.6: Window and non-Window Performance Comparison

On the contrary, the CPU's performance decreases significantly under the increased amount of calculations,hence the CUDA implementation of the HY algorithm with sliding window provides a bigger speedup.

	GTX 960 vs i7 3770		
Stocks	1 Thread	4 Threads	8 Threads
1024	x14.2	x6.3	x7.6
2048	x15	x6.7	x6
5000	x16.5	x6.9	x5.9
8000	x16.8	x6.7	x5.7
10000	x16.8	x6.8	x5.7

	GTX 980 vs i7 3770		
Stocks	1 Thread	4 Threads	8 Threads
1024	x17.3	x9.0	x9.2
2048	x18.4	x8.2	x7.3
5000	x19.4	x8.1	x7
8000	x19.3	x7.8	x6.5
10000	x19.5	x7.8	x6.5
15000	x19.6	x7.8	x6.7
18000	x19.6	x7.8	x6.4

Table 5.5: Window Implementation Speed Up

We cam again extract the strictly computational latency of the design :

	GTX 980 vs i7 3770		
	GTX 980	1 Thread	8 Threads
Latency in ms	1.463	181.17	64.85

Table 5.6: Computational latency for 5000 inputs with sliding window

And the achieved throughput compared to theoretical :

	Theoretical	Achieved
GTX 980	4612 GFLOP/s	297 GFLOP/s
CPU (1 Thread)	108.8 GFLOP/s	0.696 GFLOP/s
CPU (8 Threads)	108.8 GFLOP/s	1.972 GFLOP/s

Table 5.7: Throughput GTX 980 for 5000 stocks

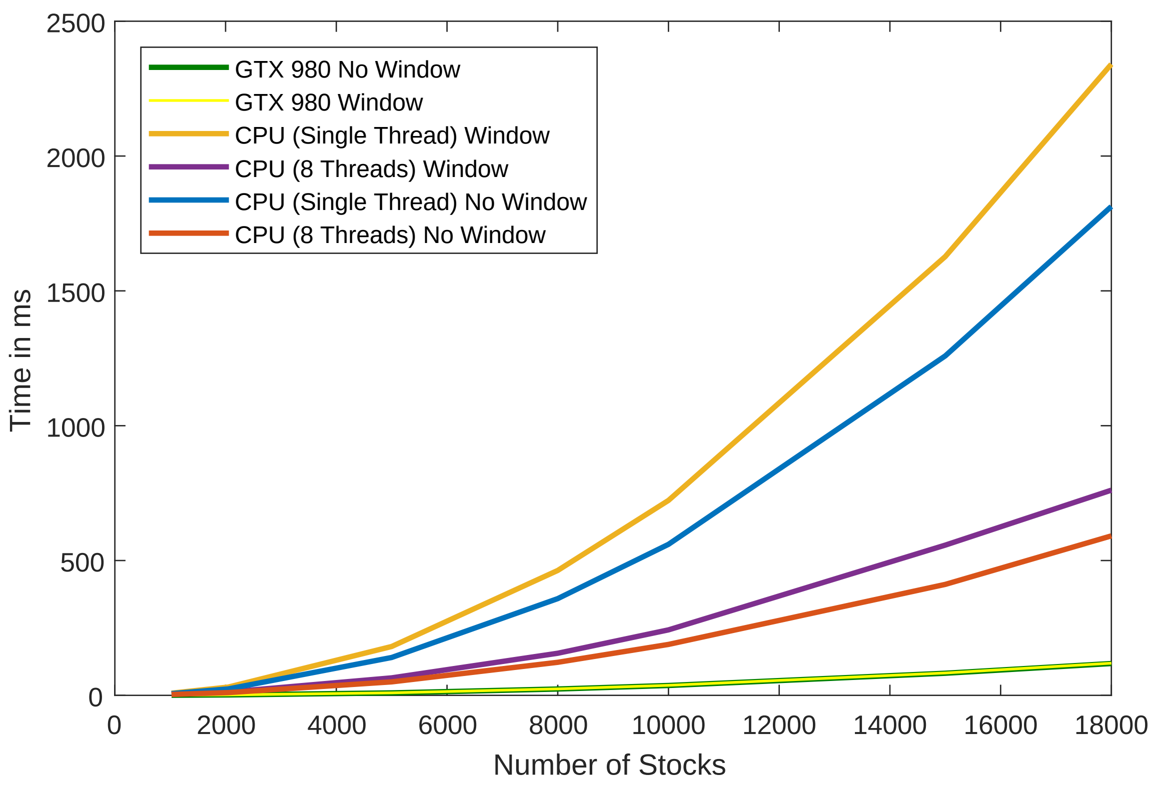

Figure 5.7: Window Implementation Comparison

This version was implemented as a direct comparison to previous an implementation on a Maxeler MPC-C Series FPGA[18]. Unfortunately, we cannot draw a comparison with that work because as we mentioned before, the processing times presented are not purely computational but include a significant latency for network package processing. Nevertheless, the authors estimate that the computational time needed in that implementation for 5000 inputs is approximately 160ms. Thus we conclude with that our proposed implementation performs significantly better, about x17 speedup compared to the corresponding FPGA-based implementation.

Chapter 6

Conclusions

6.1 Summary of Thesis Achievements

The continuous development of multi-core GPUs driven by the video gaming industry developed processing units with enormous capabilities in parallel floating point calculations. Relatively inexpensive devices found not only in professional environments but also inside the vast majority of personal computers, can become using a parallel computing platform as CUDA a secondary general purpose computing unit. GPUs are now a useful tool added to a category dominated by FPGAs, multi-CPU supercomputers and hybrid CPU-FPGA supercomputers, a tool already installed in millions of machines around the world.

In the specific problem of calculating the HY correlation coefficients for large streaming inputs, with or without sliding window, our CUDA implementation achieved a significant increase in performance on both the high-end and the mid-end devices compared to both single and multi-thread CPU executions. That performance gap grew even more in the window implementation where the GPU was able to handle the increased number of operations without any significant loss of performance. On the other hand the CPU's performance was notable diminished, and as a result, the overall speedup grew from x15 to x19 from the version without sliding window to the version with sliding window compared to a single thread CPU.

The performance of the 960GTX made a case about the cost of performance considering that it was retailed for less than half the price of a 980GTX. These results prove that even though the high-end CUDA GPUs are faster than their cheaper counterparts, we can still get positive results using cheaper CUDA devices, devices that a median personal computer uses.

Finally considering that the limits of our implementation are the size of the memory on the device concerning the size of inputs we can calculate, and the bandwidth of the host-device interface concerning the biggest part of the total execution time, we can expect this implementation to perform even faster and for larger inputs knowing that the new generation of CUDA GPUs have larger memories and the faster PCI-e ver. 4.0 has been already standardized and expected to get implemented in the near future GPUs and motherboards.

6.2 Applications

Nowadays, the major financial institutes have portfolios containing various assets, and in the global market of today, their value is interconnected with other assets from around the world. Those assets can be anything from land properties, commodities such as oil and gold, bonds, stocks, currencies or any other financial product. Tracking the streaming values of all these assets from global markets and calculating the essential metrics with minimum latency has created a need fast computational systems that can handle large loads of inputs and can process them in as little time as possible.

The HY correlation coefficients can be used to calculate the correlation between assets when their value is observed at discrete times in a non-synchronous manner. Our proposed design, implemented in a CUDA-enabled GPU outperforms CPU implementations while being able to handle loads up to 18,000 inputs, with the hardware used.

Considering that according to the World Federation of Exchanges[17], the number of public traded companies in the Europe, Africa and Middle East areas combined is around 14600 and in North America (Nasdaq,NYSE, TMX) is around 8600, the limit of 18,000 inputs of our design on the 980GTX should be sufficient in most cases. With inputs of the size, we can calculate the HY correlation coefficients of stocks from several stock exchanges around the world, in real time.

6.3 Future Work

While our work demonstrated the potential of performance computing floating operations for large streaming data sets, the vast capabilities of CUDA-enable GPU's can get exploited furthermore.

Firstly, with the use of current hardware or even better with the use of stream processing oriented hardware such as the Tesla GPU's, we can achieve an even better performance both in computational speed and in input limits without any changes in our design.

Nevertheless, the future goal is to expand our design and implement it into a complete system. We can use three GPU's that would work simultaneously, each calculating a quarter of the HY correlation and covariance matrix (the bottom left quarter as we explained does not require any calculations). All three cards are not required to be equally fast considering that the heaviest load is on the top right corner of the matrix. With this implementation will not only reduce the kernel execution time but more importantly reduce by 2/3 the time needed to transfer the results back to the host system which is currently the bound of our proposed design.

Finally, our last goal is to design a complete system where part of it would be the CUDA implementation. A system that takes as inputs the asset prices from the network, efficiently process the network packets into float values ready to get processed by the GPU's and forward the results back to the network, all in real time with a realistic limit of under 1 second. Such a system coupled with a fast serial processor as the IBM Power and NVidia Tesla GPU's could also take advantage of the new Nvidia technology NVLink, which allows for faster transfers between the GPUs and the host system than any current or at the moment future PCI-e interface[14].

Bibliography

[1] Dar-Jen Chang, Ahmed H Desoky, Ming Ouyang, and Eric C Rouchka. Compute pair-wise manhattan distance and pearson correlation coefficient of data points with gpu. In *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on*, pages 501–506. IEEE, 2009.

[2] CUDA Toolkit Documentation. v7. 5. *NVIDIA Corporation, September*, 2015.

[3] Oded Green and Yitzhak Birk. A computationally efficient algorithm for the 2d covariance method. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE, 2013.

[4] NVIDIA GeForce GTX. Gf100: Worlds fastest gpu delivering great gaming performance with true geometric realism. *White paper, NVIDIA Corporation*, 2010.

[5] NVIDIA GeForce GTX. 680: The fastest, most efficient gpu ever built, 2012.

[6] NVIDIA GeForce GTX. 980: Featuring maxwell, the most advanced gpu ever made. *White paper, NVIDIA Corporation*, 2014.

[7] NVIDIA GeForce GTX. Kepler gk110/210. *White paper, NVIDIA Corporation*, 2014.

[8] Jan Hauke and Tomasz Kossowski. Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. *Quaestiones geographicae*, 30(2):87, 2011.

[9] Takaki Hayashi, Nakahiro Yoshida, et al. On covariance estimation of non-synchronously observed diffusion processes. *Bernoulli*, 11(2):359–379, 2005.

[10] Ekasit Kijsipongse, U Suriya, Chumpol Ngamphiw, Sissades Tongsima, et al. Efficient large pearson correlation matrix computing using hybrid mpi/cuda. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 237–241. IEEE, 2011.

[11] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[12] Harry Markowitz. Portfolio selection. *The journal of finance*, 7(1):77–91, 1952.

[13] Pinar Muyan-Ozcelik, John D Owens, Junyi Xia, and Sanjiv S Samant. Fast deformable registration on the gpu: A cuda implementation of demons. In *Computational Sciences and Its Applications, 2008. ICCSA'08. International Conference on*, pages 223–233. IEEE, 2008.

[14] NVIDIA. Nvlink high-speed interconnect: Application performance. *White paper, NVIDIA Corporation*, 2014.

[15] NVIDIA. Summit and sierra supercomputers: An inside look at the u.s. department of energys new pre-exascale systems. *White paper, NVIDIA Corporation*, 2014.

[16] Apostolos Nydriotis, Pavlos Malakonakis, Nikos Pavlakis, Grigorios Chrysos, Ekaterini Ioannou, Euripides Sotiriades, Minos N Garofalakis, and Apostolos Dollas. Leveraging reconfigurable computing in distributed real-time computation systems. In *EDBT/ICDT Workshops*, 2016.

[17] World Federation of Exchanges, sep 2017.

[18] Qualimaster Project. Qualimaster. URL `http://qualimaster.eu/`.

[19] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[20] Taruna Seth and Vipin Chaudhary. Big data in finance., 2015.

[21] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.