Accelerated Anticor Online Portfolio Selection on Multi-core CPUs and GPU with OpenCL

Amril Nazir

***Abstract*—We present an efficient financial portfolio selection and optimization implementation of Anticor's algorithm. Our solution utilizes the OpenCL framework to offer the most optimal speedups on heterogeneous hardware platforms that take advantages of multi-core CPU and many-core GPU architectures. To our knowledge, this work is the first accelerated Anticor portfolio selection implementation that solves computationally intensive portfolio optimization problems across heterogeneous platforms using both multi-core CPUs and GPU.**

***Index Terms*—Big Data Portfolio Simulation, Portfolio Selection, Online Portfolio Selection and Optimization, Accelerated Portfolio Optimization, Anticor, GPGPU.**

I. Introduction

FUND and portfolio managers often need to analyze large amount of financial data to make investment decisions. The most common decision they have to make is on how to optimize the allocation of investors' wealth across a set of assets. In achieving this, the portfolio selection and optimization method is often used in their daily work routines. Most recently, many modern portfolio optimization algorithms have been developed to optimize the allocation of a set of assets from existing portfolio selection. These algorithms rely on large amount of historical data to make accurate decisions.

In 2004, [1] published an influential paper titled "Can We Learn to Beat the Best Stock" in the Journal of Artificial Intelligent Research", demonstrating how their proposed Anticor portfolio optimization algorithm was able to consistently outperform the best stocks in NYSE, TSX, S&P500, and DJIA stock markets for 9 years from 1994 to 2003. Interestingly, the proposed algorithm was able to significantly outperform existing algorithms in the literature. However, the Anticor algorithm is computationally intensive algorithm due to three main reasons: (i) the algorithm needs to compute the cross-correlation between all possible asset pairs, (ii) for each asset, the algorithm needs to compute the cross-correlation between all different window sizes to find the most optimal configuration, and (iii) the algorithm requires large repetitive computations based on the positive and negative correlations during the several phases of its execution.

Recently, general purpose computation on graphics processing units (GP-GPUs) have emerged as a competitive parallel computing platform for computationally expensive and demanding tasks, which offers significant speedup compared to the central processing unit (CPU). GP-GPUs are especially well-suited for computing the Anticor algorihm because it has several hundreds/thousands of streaming processors that would enable computationally intensive computations to be performed independently in parallel. Further, the graphic card has became so widespread to the point that they are now embedded in most recent and modern consumer laptops. Since fund/portfolio managers often need to travel to meet their clients, this provides a significant advantage to the fund/portfolio managers as it offers them the ability to perform complex financial analysis while travelling.

The objective of our work is to tailor an Anticor algorithm for GPU devices, enabling massive computations of cross-corelations and sorting operations required for the algorithm phases. Acceleration of the algorithm using GPUs would certainly be of benefit to the financial community, but, there are many challenges involved in achieving this. First of all, the number of historical prices is commonly not fixed for each asset. This makes GPU optimization difficult because we will not be able to provide a generic optimization procedure. Effective scheduling is required at the host level in which the kernel will need to adapt different historical sizes. Second, financial optimization calculations rely on many dependent parameters such as the historical window sizes and the total number of assets. For example, the calculation for the moving average reversion in the Anticor algorithm largely depends on the lookup up window size. There is no reliable way to determine the optimal historical window size before the algorithm execution.

Third, the Anticor algorithm will need to compute the moving average reversion with different window sizes until the accuracy of the direction movement reaches the profit and/or risk threshold. Fourth, to complicate matter, the optimal window size may not be the same for different types of assets. Hence, there is a need to determine the threshold based on the historical data for each stock, which is only known after several iterations of the cross-correlation phases.

The above issues pose a serious challenge for Single Instruction, Multiple Thread (SIMT) architectures like the GPU, where context switching between groups of threads is used to hide memory latency. Threads are dispatched as work items, and are grouped into a set of workgroups, with threads performing the same task as their workgroup peers, but on different data items. There is a need to ensure that all these threads compute the same amount of computations concurrently, as to avoid redundant operations when some some threads are assigned with less work.

Finally, the Anticor algorithm has a moving average reversion computation which may finish soon if the optimal window size is detected early, or the computation may take significantly longer if the optimal window size threshold is large. Hence, there is a need to constantly balance the workloads based on the window sizes at run-time to avoid any redundant operations from threads which are assigned with less work.

In this paper, we present an efficient financial portfolio

A. Nazir is with the Department of Computer Science, Taif University , Taif, Makkah, 21974 Saudi Arabia e-mail: amril@tu.edu.sa.

selection and optimization of Anticor's algorithm. Our solution utilizes the OpenCL to offer the best speedups, taking advantages of the GPU architectures on heterogeneous hardware platforms. We also implemented an optimized multi-core CPU implementation to provide fair comparison of the acceleration benefits. To our knowledge, this work is the first Anticor implementation that uses the OpenCL programming framework to optimize the portfolio optimization across both multi-core CPUs and GPU.

Section II reviews background and related work on portfolio selection and optimization algorithms, the Anticor algorithm, as well as related work on portfolio optimization on the GP-GPU. An overview of the GPU architecture is given is Section III. Section IV describes our implementation details including specific GPU optimization techniques, and a cost benefit analysis. Section V provides information on the experimental setup and parameters. Benchmarks, empirical comparisons and discussions are provided in Section VI. Section VII concludes our paper.

II. Background and Related Work

A. Portfolio Selection and Optimization Algorithms

There are two different categories of portfolio selection and optimization algorithms. The first category is based on theoretically grounded algorithms, while the other category is based on heuristics. Earlier portfolio optimization algorithms were developed based on the theoretical guarantee of exponential growth, aiming to achieve as much wealth as the best constant rebalanced portfolio. The concept is to allocate a proportionate amount of investment to a set of individual stocks so that the wealth can accumulated at exponential rate until the end of investment period. Such algorithms include Universal Portfolio [2], Exponential Gradient [3], and Online Newton Step [4]. These algorithms aim to accumulate the wealth a through sequential rebalancing strategy possessing explicit lower bounds given a sufficiently long period of time. Although very elegant in terms of their mathematical formulation, they have displayed very disappointing performance in practical applications [1], [5].

More recent algorithms have employed heuristic strategies to maximise the total wealth as well as minimizing investment risk. They have been shown to outperform all theoretical algorithms in empirical studies. However, there are only a handful of heuristic strategies that have been promising recently. These include Anticor [1], Kalman Filtering [6] and OLMAR [5], [7]. Anticor is the first algorithm which was shown to outperform all theoretical-based algorithms, including Nonparametric Nearest Neighbor [8], Nonparametric Nearest Neighbor Log-optimal [9], Exponential Gradient [3], Exponential Gradient [3], Universal Portfolio [2], Online Moving Average Reversion [5], Robust Median Reversion [10], and Confidence Weighted Mean Reversion [11], [12]. However, the latest empirical studies have shown that the OLMAR outperformed both Anticor and all other algorithms in the literature on three major historical datasets: NYSE, S&P500, and TSX markets. Independent studies conducted by Paul Perry ([13]) on more recent ETF datasets have also validated the superior of OLMAR algorithm over other existing algorithms. Interestingly, the OLMAR algorithm is based on the original concept of Anticor's price mean reversal. However, the difference is that Anticor is based on a single-period price reversal, while OLMAR exploits the multi-period price reversal correlation to further increase the accuracy of the prediction. Recently, [6] also claims that its proposed algorithm gives better profitability than Anticor, but the algorithm has not been validated extensively for consideration as a serious contender.

B. Anticor Algorithm

In 2004, [1] published a novel online portfolio selection algorithm that has been demonstrated to outperform all other existing portfolio selection and optimization algorithms in the literature. While traditional universal algorithms and technical trading heuristics attempt to predict winners or trends, their approach, known as the Anticor algorithm, relies on predictable statistical relations among all pairs of stocks in the portfolio. The principle of the Anticor (AC) algorithm is to evaluate changes in overall stocks' performance by dividing the historical sequence of past returns series into equal-size periods known as windows, each with a length of w days, where w is an adjustable parameter. Following the mean reversion principle, the algorithm will then transfer the wealth from recently high-performing stocks to anti-correlated low-performing stocks. The idea is that low-performing stocks will eventually increase to the prices' mean. Initially, Anticor captures a short stock market history between two consecutive windows LX_1 and LX_1, each of w trading days [14], [1]:

$$LX_1 = log(x_{t-2w+1}), ..., log(x_{t-w})^T$$
$$LX_2 = log(x_{t-w+1}), ..., log(x_t)^T$$

The LX_1 and LX_2 are the two vector sequences constructed by taking the logarithm over market subsequences corresponding to the time windows $[t-2w+1; t-w]$ and $[t-w+1; t]$, respectively. Further, window size w is chosen based on historical performance. In [1]'s empirical studies, the chosen value is $w = 30$ for the best performance. However, empirical studies have also shown that the value will need to be modified based on the historical data to achieve the the best result. Next, the cross-correlation matrix between column vectors in LX_1 and LX_2 is calculated as follows:

$$M_{cov}(i,j) = (LX_{1i} - \mu_{1i})^T (LX_{2j} - \mu_{2j})$$
$$M_{cov}(i,j) = \begin{cases} \frac{M_{cov(i,j)}}{\sigma_{1i}\sigma_{2j}} & \sigma_{1i}, \sigma_{2j} \neq 0 \\ 0 & otherwise. \end{cases}$$

The strategy of the algorithm is to generate signal based on two important conditions. The first condition is when it detects that stock i has outperformed stock j during the last window. The second condition is when the stock i's performance in the last window is anti-correlated to stock j's performance in the second last window $[\mu_2(i) \geq \mu_2(j) \wedge Mcorr(i,j) > 0]$. If both criteria are met, the algorithm then transfer weigh allocation from stock i to stock j in the hope that stock j will increase, leading to higher profits gained. Despite the algorithm's simplicity, the empirical results for four major market indices (NYSE, S&P500, DJIA, and TSX) from July 1962 to April 2013 have provided strong evidence that the Anticor algorithm is able to significantly beat the market [7].

C. Portfolio Optimization on GP-GPU

There are a few attempts in the literature to develop parallel algorithms for portfolio selection and optimization. Recent approaches to the use of CPU and GPU to speed up the portfolio selection execution include [15] who accelerated the genetic algorithm on the GPU to compute value-at-risk (VaR), which is a risk measure of potential loss on a specific portfolio. The main uses of VaR are in risk management and financial reporting. [16] further optimizes the GPU code for VaR using three techniques: using problem reformulation, module selection, and kernel merging. Based on these optimization techniques, they were able to achieve 538× speedup over the sequential VaR implementation.

[17] implemented a different portfolio optimization technique, which accelerates a Numerical Particle swarm optimization (NPSO) algorithm on the GPU to price complex option pricing. [18] also accelerated a simulated annealing algorithm on the GPU to derive the optimal portfolio from randomly generated portfolios. The author indicated that he was able to achieve 4× speedup over the sequential implementation. [19] further accelerated the Critical Line Algorithm to optimize portfolios using Markowitz's Efficient Frontier, in which they were able to achieve 8× speedup over the sequential implementation. To our knowledge, there has been no previous attempt to accelerate the Anticor algorithm on a GPU.

III. The GPU Architecture

A number of frameworks have emerged for general programming of GPUs in recent years. There have been quite remarkable efforts on the GPU implementations from the academia [20], [21], [22], [23], [24]. The implementations are more geared towards NVDIA's CUDA frameworks. However, the CUDA framework is only able to run on NVDIA devices only. In this paper, we present the first heterogeneous implementation of Anticor algorithm. The OpenCL framework will be used to achieve this goal. OpenCL is an open standard for parallel programming on heterogeneous architectures, which makes it possible to express parallelism in a portable way so that applications written in OpenCL can run on different architectures without code modification. This is particularly beneficial for financial analysts as it will provide them with the abilities to conduct analysis independently of the hardware platforms.

A. Parallel Execution

At the highest level, OpenCL divides tasks between a CPU host and GPU device. The kernel functions are invoked from the host, but executed on the device by multiple threads concurrently. Threads are generally known as work items, and they can be dispatched into 1 dimension (1D), 2 dimension (2D), or 3 dimension. From the kernel, the work items can be identified by their local IDs, group IDs, and/or group IDs. From the host, it is possible to specify the number of work items that need to be dispatched, the dimension configuration (i.e., 1D, 2D, or 3D), and the local item size. The local item size (also known as the work group size) specifies how the work items should be grouped. For example, specifying the local size to be 64 with 128 work items imply that the work items should be partitioned into 2 groups, in which each group comprises 64 work items. A set of work items that belong to the same work group will be able to share their data with each other through a local memory of the GPU. However, the local memory size is typically very small (e.g., 32 KB). Hence, excessive use of the local memory may impede the performance of the GPU. Further, it is not possible for threads to share data between different work groups. Therefore, the final results from a work group must be written to the global memory so that the results can be fetched by other work items at later point of the algorithm execution. In OpenCL, the number of work items (i.e., local item size) that can be assigned to work group is limited by hardware capacity. For example, the AMD Radeon HD 8650G has a maximum capacity of 256 for a work group size. Moreover, the OpenCL has a limitation whereby the number of work items must be a multiple of the work group size. For example, dispatching 100 work items with 64 work group size will not be permitted. Such restriction has been removed in OpenCL 2.0. However, the OpenCL 2.0 is still considered a new development and most modern laptops do not support OpenCL 2.0 yet at the time of writing.

In hardware, the GPU comprises a number of compute units. The number of compute units vary from a device to another device. For example, the AMD Radeon HD 8650G has 6 compute units with processing clock of 720MHz. Each compute unit includes 4 separate SIMD units for vector processing. Each SIMD further comprises 16 ALUs. Each 16-wide SIMD processes will need to process one 64-wide wavefront over 4 clock cycles. This implies that a wavefront is comprised of 64 work items, and is issued to a SIMD, but it takes 4 cycles to execute operations for all 64 work items. To support massive parallelism, each SIMD simultaneously executes a single operation across 16 work items (on 16 ALUs), and each SIMD can execute different wavefronts.

B. Memory Management

The GPU has its own physical memory, which is separate from the host's main memory. The GPU comprises transistors of ALUs and registers. Registers store thread state and allow fast switching between work items. The memory hierarchy begins with global memory, which is capable of storing few gigabytes. While plentiful, it is approximately 100 times slower in comparison to on-chip memory. However, the global memory has the advantage with the ability to be read and written by the host and all work items on the device. GPUs also have caches which provide the ability to run multiple threads concurrently, interleaving to hide the latency. It provides a unified read/write caching system with virtual memory support and excellent atomic operation performance.

In order to optimize the algorithm efficiently, it is crucial to understand how the OpenCL memory model maps to the actual physical GPU model. In OpenCL, the memory management is explicit. The programmer must explicitly transfer data from host, to the global/constant memory, and further to the private, local, or back to the global memory. The private memory is accessible only by a work item. On the other hand, the local memory can be shared within a work group. The global and constant memory are visible

to all work groups and each work item. In OpenCL, these memory assignments can be explicitly assigned using such variables: *__global*, *__local*, *__constant* and *__private*. In order to ensure efficient access to memory, we may need to apply various techniques such as the memory coalescing, memory alignment, and employ careful allocation of private or local memory, in order to avoid high register usage and low GPU occupancy.

IV. Implementations

Algorithm 1 shows a sequential Anticor procedure for the execution of the Anticor algorithm. At the initial algorithm execution, all possible asset pairs are identified from a sequence of asset examples. Given two different asset i and j, the aim is to find a positive correlation between asset i during the second last window and asset j during the last window. Based on the window size, the algorithm computes the logarithms between two different time windows (line 7). Further, the algorithm computes the means from the two logarithms, and finally computes the cross correlation (lines 11-12). Based on the results of the cross correlation, the algorithm determines the weight allocation for the new portfolio. This is done by comparing the calculated means with the calculated cross correlations to determine the new weight allocation that is needed to be transferred from asset i to asset j, or vice versa (lines 19-29). Finally, the algorithm updates the allocation weight of current portfolio p to a new allocation weight as computed by the claims (lines 31-36).

We implemented the single-core CPU version of the algorithm as a single thread using C. The most computationally expensive part of the algorithm is the cross correlation computation. The computational complexity significantly increases as the number unique assets increases. For example, for 100 unique assets, a maximum 4950 cross-correlation computations are required. As the number of unique assets increase to 1000, the algorithm will need to compute 1,999,000 computations. We further optimize the cross-correlation code using matrix operations, rather than loop, as the for loop implementation will be extremely slow when the number of assets increases.

Next, our aim is to port and implement the Anticor algorithm for multi-core CPUs and GP-GPU using OpenCL. The immediate action is to translate the key Anticor procedures to OpenCL kernel code. However, this would require some insights on determining the most expensive operations that require accelerations. If we were able to identify the key performance bottleneck, we could effectively select and parallelize the most important code at the kernel. We conducted a preliminary CPU benchmark within each segment of the procedures to find time critical hotspots and diagnose performance issues. Figure 1 illustrates the performance breakdown for the different procedures of the Anticor algorithm execution under increasing number of assets.

We can observe that ComputeCrossCorrelation procedure incurs the most significant performance overhead. The overhead is in fact 90×-120× when compared to the other procedures. The ApplyClaims procedure incurs the second most significant performance overhead. However, its overhead is considered very much lower than the ComputeCrossCorrelation procedure. It can be observed that the performance overhead for ComputeCrossCorrelation

Algorithm 1 Sequential Anticor Algorithm

1: **Inputs**
2: A is a sequence of asset examples
3: ws is the window size
4:
5: **procedure** ANTICOR(A, ws)
6: **for** each asset pair i, j
7: Let $LX1$ and $LX2$ be the two vector sequences of the logarithm from time windows $[t-2w+1, t-w]$ and $[t-w+1, t]$
8: Let p be the portfolio weight allocation
9: $claim_{i\to j} \leftarrow 0$
10: $transfer_{i\to j} \leftarrow 0$
11: $\mu_1, \mu_2 \leftarrow ComputeMean(LX1, LX2)$
12: $M_{cor} = ComputeCrossCorrelation(LX1, LX2, \mu_1, \mu_2)$
13: $claim_{i\to j} \leftarrow$ APPLYCLAIMS(M_{cor}, μ_1, μ_2)
14: APPLYTRANSFER($p, \mu_1, \mu_2, claim_{i\to j}$)
15: **end for**
16: Sort a set of M_{corr} values to determine highest positive and highest negative correlations
17: **end procedure**
18:
19: **procedure** APPLYCLAIMS(M_{cor}, μ_1, μ_2)
20: **if** $\mu_2(i) \geq \mu_2(j) \wedge M_{cor}(i,j) > 0$ **then**
21: $claim_{i\to j} \leftarrow claim_{i\to j} + M_{cor}(i,j)$
22: **if** $M_{cor}(i,i) < 0$ **then**
23: $claim_{i\to j} \leftarrow claim_{i\to j} + M_{cor}(i,i)$
24: **end if**
25: **if** $M_{cor}(j,j) < 0$ **then**
26: $claim_{i\to j} \leftarrow claim_{i\to j} + M_{cor}(j,j)$
27: **end if**
28: **end if**
29: **end procedure**
30:
31: **procedure** APPLYTRANSFER($p, claim_{i\to j}$)
32: Let t be the index of last trading day
33: $transfer_{i\to j} = \frac{claim_{i\to j}}{\sum_j claim_{i\to j}}$
34: $b^{t+1} = b^{t+1} - transfer_{i\to j}$
35: $b^{t+1} = b^{t+1} - transfer_{j\to i}$
36: **end procedure**

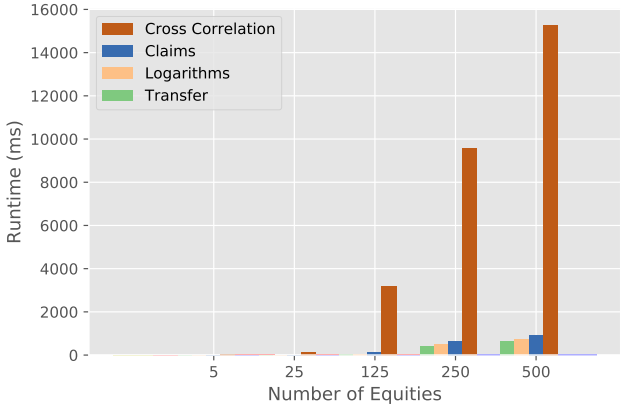

Fig. 1. The runtimes for the different procedures of the Anticor algorithm under increasing number of assets. The Cross correlation procedure is clearly the most computationally intensive when compared to other procedures.

overly suffers with an increase number of assets. On the other hand, the APPLYCLAIMS only incurs very low overhead with the increase of asset size. For the remaining procedures, there is only a slight decline in performance with increase of asset size. From this observation, we conclude that the major bottleneck of the Anticor algorithm is due to the cross-correlation procedure. Hence, this procedure would be the focus of our optimization for both multi-core CPU and GPU implementations.

Our initial implementation of the algorithm on OpenCL does not involve any optimization techniques (Algorithm 1). We simply ported the same CROSSCORRELATION procedure as a kernel function in OpenCL. Each asset pair i and j is identified from the host, and they are sent to the kernel in a batch. From the kernel, each pair of cross-correlation computation is assigned as a single work item. The aim is to maximize the number of cross-correlation throughput by maximizing the number of work items dispatched on the device. Since the number of work items have to be multiple of the work group size in OpenCL 1.2, we will need to estimate the number of work items required from the host. This involves identifying the number of m possible cross-correlation iterations from a set of n assets. In theory, this can be calculated as follows:

$$m = \frac{n!}{2!(n-2)!} \tag{1}$$

However, the combinatorial computation can get very large as the asset size n increases (e.g., $\geq$ 200). Furthermore, for each cross-correlation task, we will need to assign a work item with a unique beginning and ending indexes of the asset data items. To address both issues, Algorithm 2 shows a simple method in which both m, the beginning and ending indexes of the data items can be determined at the host for each work item. Initially, we will need to retrieve the asset datasets from the storage into the host memory (lines 2-3). Next, the Anticor algorithm will need to determine every possible asset pairs between two different assets i and j. Hence, lines 10-25 iterates every single pair of each unique asset, and it keeps tracks the starting and ending index positions for each cross-correlation computation during the iteration (lines 14-15). Once all the data items have been assigned to their respective work items, we initiate data transfer of the $indexA$, $indexB$, and $total$ count to the device (line 26). In line 36, the main Anticor procedure invokes the ASSIGNDATAITEMS kernel iteratively for each asset pair i and j. The APPLYREDUCTION kernel is then invoked after every ASSIGNDATAITEMS kernel completion (line 37). The APPLYCLAIMS and APPLYTRANSFER kernels are further invoked to update the new portfolio weight allocation (lines 38-39). Finally, the final result is retrieved from the GPU global memory (line 41).

Once the data transfer has been initiated, we will need to determine the number of mw work items to be dispatched on the device, which is calculated as follows:

Algorithm 2 Method to compute the number of cross correlation iterations, the beginning and ending indexes of the asset data items for each work item. Further, the the final reduction phase is performed before APPLYCLAIMS and APPLYTRANSFER kernels are invoked.

1: **Inputs**
2: indexA is a sequence of index values for asset i
3: indexB is a a sequence of index values for asset j
4: ws is the window size
5: **procedure** ASSIGNDATAITEMS($indexA, indexB, ws$)
6: Let $msize$ be the total number of data items
7: $start \leftarrow \frac{msize}{ws-1}$ ▷ index of starting point
8: $incrementB \leftarrow 1$
9: $total \leftarrow 0$
10: **for** $k = start$ such that $k \geq 1$
11: $counterB \leftarrow incrementB * ws$
12: **for** $m = 1$ $start$ such that $m \leq start$
13: $counterA \leftarrow incrementA * ws$
14: $indexA(total) \leftarrow counterA$
15: $indexB(total) \leftarrow counterB$
16: $counterA \leftarrow counterA + 1$
17: $counterB \leftarrow counterB + 1$
18: $total \leftarrow total + 1$
19: $m \leftarrow m + 1$
20: **end for**
21: $incrementA \leftarrow incrementA + 1$
22: $incrementB \leftarrow incrementB + 1$
23: $start \leftarrow start - 1$
24: $k \leftarrow k - 1$
25: **end for**
26: Initiate data transfer of $indexA$, $indexB$, and $total$ count to the device
27: **end procedure**
28:
29: **procedure** ANTICOR(A, ws)
30: **for** each asset pair i, j
31: Let $LX1$ and $LX2$ be the two vector sequences of the logarithm from time windows $[t-2w+1, t-w]$ and $[t-w+1, t]$
32: Let p be the portfolio weight allocation
33: $claim_{i \to j} \leftarrow 0$
34: $transfer_{i \to j} \leftarrow 0$
35: $\mu_1, \mu_2 \leftarrow ComputeMean(LX1, LX2)$
36: M_{cor} = ASSIGNDATAITEMS($LX1, LX2, \mu_1, \mu_2$)
37: $M_{cor} = ApplyReduction(M_{corr})$
38: $claim_{i \to j} \leftarrow$ APPLYCLAIMS(M_{cor}, μ_1, μ_2)
39: APPLYTRANSFER($p, \mu_1, \mu_2, claim_{i \to j}$)
40: **end for**
41: Retrieve final result from the GPU global memory
42: **end procedure**

$$mw = \frac{m}{lw} \tag{2}$$

$$G_s = mw * lw \tag{3}$$

$$L_s = \begin{cases} G_s & if\ G_s < lw \\ lw & otherwise \end{cases} \tag{4}$$

where m represents the amount of data items required to compute cross-correlation for all possible pairs of unique assets, and lw is the local item/work group size. The G_s is the finalized global work size and L_s is the new ideal work local size which is re-calculated based on the global work size. The lw is modified differently based on the implementation. For multi-core CPU, we discovered that assigning larger work items (i.e., 1024) have a positive impact on the performance, while the GPU implementation performs better within 64 to 256 work items per work group. Furthermore, the CPU

implementation achieves better performance when the cross correlation procedure is merged as a single kernel, whereas merging such a procedure as a single kernel has significant performance degradation on the GPU. Hence, we optimize the multi-core and GPU implementations differently based on the compute device architectures.

For the GPU implementation, identifying the data item indexes for each work item from the host has the advantage of simplifying the kernel code as well as reducing the amount of register usage within the kernel. However, it imposes a slight memory transfer overhead to the GPU, but such overhead does not have a significant impact on the performance since a global GPU memory with 2GB is typically sufficient to transfer a large dataset just in a single trip memory transfer (e.g., the MSCI World index dataset which is currently the largest stock market index with 1,653 assets only uses a maximum of 255MB of RAM consumption on the GPU global memory).

However, our initial naive GPU implementation has serious performance bottleneck. The top Figure 2 shows the run times for both AMD APU (i.e., multi-core CPUs) and AMD GPU implementations with increasing number of assets. With the current naive implementation, the multi-core CPUs implementation performs better than the GPU by an average of $2\times$ speedup when experimented on a wide range of asset sizes. This somewhat indicates a serious inefficiency in our GPU implementation, and therefore code optimization is crucial to deliver optimal performance on the GPU. To analyze this further, the bottom Figure 2 plots the performance breakdown of hotspot analysis within each OpenCL kernel. Interestingly, the GPU almost outperforms the multi-core CPU in all procedures except the cross correlation. For cross correlation, we notice the GPU is relatively slower by 120 percent when compared to the AMD APU. We suspect this performance degradation is due to the overuse of GPU registers, and ultimately causes the register spilling.

To address this issue, we present a solution that partitions the cross-correlation computations into three different kernels. The first kernel involves in computing the logarithms of each asset pair from two different time windows, as well as computing the mean of the two series (Algorithm 3). The second kernel (Algorithm 4) calculates the denominator, which will be used to normalize the cross correlation coefficient at every point of time series. We use the square root the OpenCL's built-in fast math option to calculate the denominator value. Finally, the third kernel (Algorithm 5) performs the final correlation coefficient as required.

From lines 7-9 of Algorithm 3, the kernel retrieves both the start index values of the asset i and asset j from the global memory. These index values are obtained from the global memory, which was copied from the host to the GPU global memory. Within the kernel, line 10 verifies whether the index value is a valid index (non positive index values do not have any data items). Hence, a work item with a non positive index will not execute any further code and the ALU will be immediately freed for other queue task/kernel. On the other hand, the index values will determine the workspace area in the global memory that it needs to compute. In lines 13-18, all the work items are processed sequentially. Next, the means are calculated in lines 19-20, and finally, the means are updated atomically into the global memory as two separate

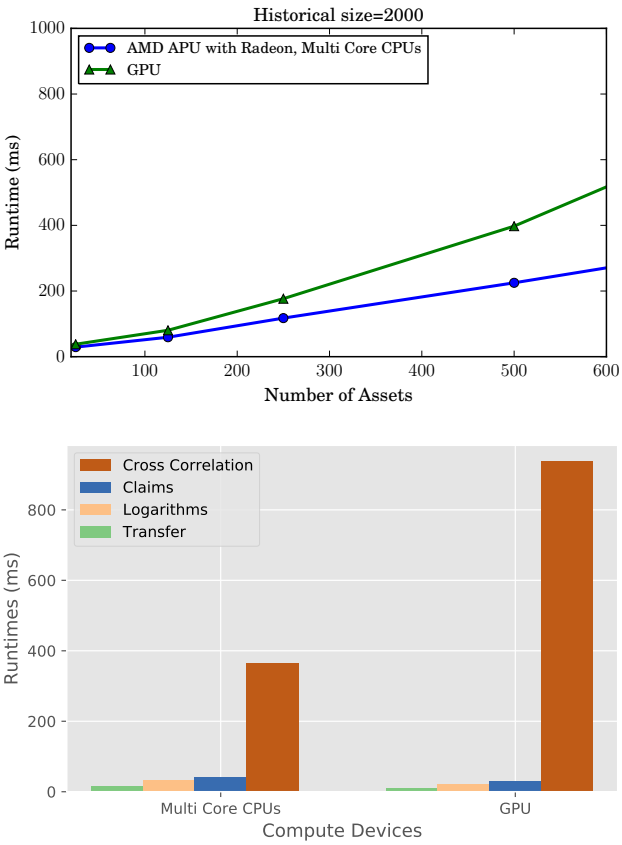

Fig. 2. The top figure plots the runtimes for both AMD APU (i.e., multi-core CPUs) and AMD GPU implementations with increasing number of assets, while the bottom figure shows the performance breakdown of hotspot analysis within the kernel for 1000 asset size with historical size of 2000.

array pointers.

Similarly, the second kernel retrieves both the start index values from asset i and asset j from the global memory in lines 7-9 of Algorithm 4. Also, the index values of asset i and asset j pair are verified (line 11). Next, the kernel retrieves the mx and my values from the global memory, which were calculated previously in Algorithm 3. The kernel performs the necessary denominator computations in lines 14-21. The computations are performed sequentially until it reaches the window size threshold. In line 21, the square root function uses a built-in fast-math optimization of the GPU. Finally, the finalized denominator value is updated to the global memory as an array pointer (line 22).

The Algorithm 5 shows how the third kernel retrieves the calculated denominator value from the global memory and performs the final correlation coefficient. Similarly, the mx and my values are also obtained from the global memory (in lines 15-16). The maximum lagging is specified between the two time windows $t-2w+1, t-w$ and $t-w+1, t$. The correlation series are calculated from the first time window until the second time window (lines 17-27). Given the correlation series, the correlation coefficient is computed based on the denominator value (line 26). The correlation coefficient is updated to the global memory (line 28). Note that for each write update to the global memory in all the three kernels, we use the atomic built-in functions to avoid any serious race conditions. Finally, the APPLYREDUCTION kernel of Algorithm 2 (line 37) will perform the final reduction phase,

Algorithm 3 Task Cross Corrrelation: Computing Logarithms and Means

```
1: Inputs
2: A      is a sequence of asset examples
3: B      is a pre-allocated global memory to store mx
4: C      is a pre-allocated global memory to store my
5: ws     is the window size
6: procedure MeanCrossCorrelation(A, B, C, ws)
7:     Let gid be the work item ID
8:     Let startA be the index value of the global memory where the cross-correlation should start
9:     Let startB be the second index value of the global memorywhere the cross-correlation should start
10:    if StartA ≥ 0 || StartB ≥ 0 then
11:        startA ← SA(gi)        ▷ Get start index value from global memory
12:        startB ← SB(gi)        ▷ Get start end value from global memory
13:        while r < ws do
14:            mx ← mx + A(startA)
15:            my ← my + A(startB)
16:            startA ← startA + 1
17:            startB ← startB + 1
18:        end while
19:        mx ← mx/ws
20:        my ← my/ws
21:        B(gid) = mx
22:        C(gid) = my
23:    end if
24: end procedure
```

Algorithm 4 Task Cross Correlation: Compute Denominator

```
1: Inputs
2: A      is a sequence of asset examples
3: B      is a pre-allocated global memory to store mx
4: C      is a pre-allocated global memory to store my
5: ws     is the window size
6: procedure CorrelationDenominator(A, B, C, ws)
7:     Let gid be the work item ID
8:     Let startA be the index value of the global memory where the cross-correlation should start
9:     Let startB be the second index value of the global memory where the cross-correlation should start
10:    Let sx and sy be the temporary variables used to compute the denominator of the x and y series
11:    if StartA ≥ 0 || StartB ≥ 0 then
12:        mx ← B(gi)        ▷ Get mx from global memory
13:        my ← C(gi)        ▷ Get my from global memory
14:        while r < ws do
15:            sx ← sx+[(A(startA)−mx)∗A(startA)−mx]
16:            sx ← sy+[(A(startA)−my)∗A(startA)−my]
17:            startA ← startA + 1
18:            startB ← startB + 1
19:            r ← r + 1
20:        end while
21:        denom ← sqrt(sx ∗ sy)
22:        D(gid) = denom
23:    end if
24: end procedure
```

Algorithm 5 Task Cross Correlation: Compute Final Correlation Coefficient

```
1: Inputs
2: A      is a sequence of asset examples
3: B      is a pre-allocated global memory to store mx
4: C      is a pre-allocated global memory to store my
5: D      is a pre-allocated global memory to store denom
6: E      is a pre-allocated global memory to store correlation coefficient
7: ws     is the window size
8: procedure CorrelationSeries(B, C, D, E, ws)
9:     Let gid be the work item ID
10:    Let startA be the index value of the global memory where the cross-correlation should start
11:    Let startB be the second index value of the global memory where the cross-correlation should start
12:    Let maxd be the maximum lagging between two series
13:    Let begin be the start index on cross-corelation series
14:    if StartA ≥ 0 || StartB ≥ 0 then
15:        mx ← B(gi)        ▷ Get mx from global memory
16:        my ← C(gi)        ▷ Get my from global memory
17:        for d = −maxd such that d < maxd
18:            sxy ← 0
19:            while r < ws do
20:                begin ← r + d
21:                if begin > 0 ∧ begin ≤ ws then
22:                    sxy ← sxy + [(A(startA) − mx) ∗ A(startB) − my]
23:                end if
24:                r ← r + 1
25:            end while
26:            r ← sxy/D(gid)
27:        end for
28:        E(gid) = r
29:    end if
30: end procedure
```

before the host reads the result from the GPU global memory (line 41 of Algorithm 2).

A. A Cost-Benefit Analysis

The major source of overhead derives from the cross correlations procedures of the multiple kernels. This is due to the overheads in writing intermediary results to the global memory, and the overheads from the frequent global memory read accesses. Nonetheless, we can hide these overheads if our implementation can ensure a balanced load on each streamed GPU processing element.

Let assume N CPU processors are used to execute W workload of the Anticor algorithm. On the GPU, we would expect the number GPU stream processor units N' to be a lot greater than the number of N CPU processors (i.e., $N' > N$).

Since the number of N' GPU stream processor units is higher than the CPU processors, we can increase our original workload W to workload W', while maintaining the average speed:

$$\frac{W}{NT_N} = \frac{W'}{N'T_{N'}} \tag{5}$$

$$\frac{W}{N(C_{tN} + S_{tN})} = \frac{W'}{N'(C_{tN'} + S_{tN'})} \tag{6}$$

where T_N is the execution time on N, C_{tN} is the computational time overhead, and S_{tN} is the synchronization communication overhead. Assuming that all tasks are distributed evenly among all processor units, we have:

$$\frac{W}{N(\frac{W}{N\mu(W/N)}+S_{tN})}=\frac{W'}{N'(\frac{W'}{N'\mu(W'/N')}+S_{tN'})} \tag{7}$$

$$\left(\frac{1}{\mu(W/N}-\frac{1}{\mu(W'/N'}\right)+\frac{S_{tN}N}{W}=\frac{S_{tN'}N'}{W'} \tag{8}$$

$$\left(\frac{1}{\mu(W/N}-\frac{1}{\mu(W'/N'}\right)+\frac{S_{tN}N}{W}>0 \tag{9}$$

and the amount of work W' required to maintain the average unit speed is:

$$W' = \frac{S_{tN}N'}{\frac{1}{\mu(W/N)}-\frac{1}{\mu(W'/N')}+\frac{S_{tN}N}{W}} \tag{10}$$

From this, we can estimate the scalability from multi-core CPU of N processors to GPU system with N' stream processors, as follows:

$$\frac{N'W}{NW'} = \frac{W[(\frac{1}{\mu(W/N)}-\frac{1}{\mu(W'/N')})+\frac{S_{tN}N}{W}]}{S_{tN}N} \tag{11}$$

$$= \frac{(\frac{W}{\mu(W/N)}-\frac{W}{\mu(W'/N')})+S_{tN}N}{S_{tN}N} \tag{12}$$

$$1 \geq \frac{S_{tN}}{S_{tN'}}+\frac{1}{S_{tN'}}\left[C_{tN}-C_{tN}\frac{\mu(W/N)}{\mu(W'/N')}\right] \tag{13}$$

, which shows that $\frac{N'W}{NW'}$ has only a slight overhead cost (i.e., $\frac{N'W}{NW'}<1$). Furthermore, since $W'>\frac{N'W}{N}$, this indicates that the overheads of data transfer to the GPU, and the synchronization cost between kernels, have very little impact on the overall performance of algorithm execution, provided a sufficiently high amount of task concurrency (i.e., high GPU occupancy) and the tasks are equally distributed and dispatched on each GPU stream processor.

V. Experiments

The experimental environment for our evaluation is shown in Table I. Our evaluation is conducted on both CPUs and GPU. CPU experiments are carried out on a Lenovo laptop with a single quad-core 2.5 GHz AMD A10-5750M APU of 8,320KB cache and 8 GB of RAM memory at its disposal. GPU experiments are run on AMD Radeon HD 8650G at 720 Mhz hosted by the same CPU. This particular graphics card has 384 multiprocesors, 2 GB of global memory and supports OpenCL 1.2. All implementations are in C, on both host and compute devices. Both CPU and GPU implementations are compiled using Visual Studio Community 2013 version 12.0.4 Update 5. For both multi-core CPUs and GPU compilations, the AMD APP SDK v3.0 for 64-bit Windows is used.

We developed three separate implementations: a single-core CPU implementation, multi-CPUs implementation, and a GPU implementation. A single-core CPU implementation is implemented solely in C without OpenCL. However, both the multi-threaded CPU and GPU implementation are implemented using OpenCL 1.2. We use the CL_DEVICE_TYPE_GPU or the CL_DEVICE_TYPE_CPU values for the device type parameter to switch between the multi-core CPU and GPU implementations. All implementations are written for a single precision computation. This is ideal since most GPU units on portable devices (i.e., laptops) mostly run with single precision.

In order to measure the performance of the algorithm, we measure the performance from two aspects: (1) the total running time for the algorithm, and (2) average runtimes for specific operations. The total running time comprises of several phases including: host processing time, data transfer time to and from the GPU, and the run times for all kernels. We measure the average runtimes it takes to perform a specific kernel operation (e.g., cross-correlation, claims, transfer etc.). We run the same experiment 20 times and calculates the average before finalizing the result. Each experiment is conducted at 20 runs to ensure that the standard devations are less than 1 percent, before we calculate the average to finalize the result. The AMD CodeXL is used to measure the performance statistics across different number of assets and increasing number of window size parameters, to assess scalability across the parallelization. Memory requirements are linear with respect to the increase number of assets and window size parameters. More than sufficient physical memory is available on both the CPU and GPU, thus, this is not a limiting factor on performance in either case.

TABLE I
Experimental Environment

CPUs	AMD A10-5750M APU
Cores	1 Quad-core, 4 single precision FP
Caches (L1/L2)	320KB/8000MB
Core frequency	2.5 GHz
DRAM	8GB
GPUs	AMD Radeon HD 8650G
# SMs	384
Caches (Local/Constant)	32KB/64KB
Shader Clock Frequency	720 Mhz
O/S	Windows 8.1 64-bit
Platform	AMD APP SDK v3.0 for 64-bit Windows
Compiler	Visual Studio Community 2013 version 12.0.4 Update 5

For each experiment, we define a number of parameters to control the computational workloads. The most important parameters are the size of data items (M), historical size (HS), and window size (ws). For the purpose of benchmarking, the optimal threshold for window size (k) is set to 180 days ($k=180$). we set the historical sizes into four groups: $HS=200$, $HS=400$, $HS=800$, and $HS=2000$. To validate our implementations on different types of workloads, we perform multiple experiments of the same historical size setting with increasing number of assets. To tune the M parameter for the number of assets, we modify the size of data items accordingly.

By determining the size of data items and the historical size, we calculate the number of work items to be dispatched by a kernel from the host. Based on the historical size, we determine the local item size as follows:

$$lw=\begin{cases} ms & if\ HS \geq ms \\ HS & otherwise \end{cases} \tag{14}$$

where HS is the historical size, and ms is the maximum work group size that the compute device is able to support. This information can be discovered via the OpenCL's CL_DEVICE_MAX_WORK_GROUP_SIZE parameter. We do not highly optimize the local work item size since the performance impact of this parameter varies on different graphic cards.

VI. Discussion

For each implementation (i.e., a single-core CPU, multi-core CPUs, and GPU), we conducted extensive experiments to measure their performance on both AMD Quad-core processor and AMD Radeon graphics card. We compare their performance in terms of the total runtimes required to complete the tasks. Benchmark results for each configuration are given in Table II. Our initial observation is that the multi-core implementation significantly outperforming a single-core CPU implementation by a sheer margin. As can be observed, the speedup gain is higher when the historical size grows. With historical size of 2000, the multi-core implementation was able to achieve amazingly $5800.11\times$ speedup when compared to $202.53\times$ speedup for $HS=200$. Same observation is made for both asset size 2^6 and 2^9. This was to be expected as our single-core implementation is unoptimized. On the other hand, the multi-core implementation was developed using OpenCL, which is optimized to exploit the instruction level parallelism.

The most interesting discussion is related to the performance of multi-core and GPU implementations because this will provide

fair insights into the acceleration benefits of the GPU. With this regards, our observation is that the speedups between various implementation configurations are largely influenced by the asset and historical size. One observation is that the speedup seems to be higher when the asset size is smaller. For example, an asset size of 2^6 achieves higher speedup at 202.53× when compared to the speedup of 196.50× for multi-core CPUs vs. single-core CPU configuration with $HS = 200$. Similarly, the GPU achieves a higher speed up at 20.37× for an asset of 2^6, while it only achieves 10.45× speedup for an asset size of 2^9.

However, the same observation does not apply for historical size, as increasing the history size does not have incremental/decremental pattern on the speedup for the GPU. The speedup pattern of multi-core over a single core CPU is clearly visible; as the historical size increases, the speedup also increases. Moreover, it can be observed that there is an increase in speedup when the asset size grows. Interestingly, GPU has no visible pattern when the historical size grows. For example, the GPU achieves the best speedup for 2^6 assets when the historical size is 400. On the other hand, the GPU achieves the best speedup for 2^9 assets when the historical size is 200. Despite having no clear patterns, we notice a few interesting observations. For example, both asset sizes of 2^6 and 2^9 incur the worst speedups when the historical size is defined to 800.

The Figure 3 further analyzes our results in detail by illustrating the average runtimes of completing the Anticor algorithm across different implementation and hardware configurations for $3000 \leq HS \geq 200$ with increasing number of assets. Increasing the number of assets has the effect of overloading the compute devices with more computations, and therefore increases the resource utilization.

For $HS = 200$, we notice that the speedup is generally insignificant for smaller size of assets. For example, the speedups were only between 1× to 3× for when the number of assets is less than 25. As the size of assets increases beyond 25, we started to notice an increase of speedup between a single thread CPU version versus multi-threaded AMD APU version. For example, the speedup achieved by the multi-core CPU is 67× for an asset size of 25. This difference is considered significant when we compared to the speedup between 2× and 3× for small amount of asset size. Seemingly, the GPU outperforms a single CPU implementation by an impressive speedup of 1118.3×. When the asset size increases beyond 2^9, the GPU amazingly achieves a massive 4004x speedup. When compared to the AMD APU, the GPU was able to achieve and an average speedup of 6.52×. We notice that the speedup is low (within 2 to 3×) when the asset size is smaller (i.e., ≤120). However, as the number of assets increases to more than 120, we started to see a significant increase of the speedup from 10× to 20×.

For $HS = 800$, the GPU achieves an average of 6.85× speedup, which is approximately similar to the speedup achieved at $HS = 200$. This poses an interesting question whether the speedup has achieved its optimal limit? To answer this question, we will need to examine whether the GPU can offer better speedup when we impose the historical size to be very large. Hence, we stress the historical size to 2000 ($HS = 2000$) in our next experiment to determine whether it can offer better speedup under very high utilization. Figure 3 plots the results. From the results, the GPU meets our expectation by outperforming the AMD APU with a better speedup when compared to our earlier results with historical size of 800. The results show that the GPU achieves approximately 8.75× speedup over the AMD APU. This is approximately two times higher speedup when compared to the 6.85× speedup achieved for $HS = 800$. From this observation, we are convinced that the results illustrate the importance of utilizing massive parallelism in GPU, as the GPU has an upper hand in performance advantage over the AMD APU when it has higher occupancy.

Based on the results, we can observe that the GPU offers a greater benefit when their capacity is fully maximized. Seemingly, the GPU does not give good speedups when the capacity of the GPU is not exhausted. As the capacity is fully utilized with increasing number of assets, we started to observe a significant difference in speedups. The explanations are due to the high number of work items which are dispatched based on the number of historical sizes. When the amount of historical size is small, less work items are being dispatched to the OpenCL kernel, which results in low occupancy of the GPU. As a result, the performance offered by the GPU is similar to the AMD CPU performance. However, as the historical size increases, the amount of data items increases too. As a result, a large number of work items will need to be dispatched concurrently, which in turn increase the GPU occupancy.

However, the speed up factor suffers when the number of historical size increases beyond 2000. The bottom right of Figure 3 plots the runtimes for $HS = 3000$. We can clearly observed that the speedup of the GPU, in average, never exceeds 5.5× speedup. The best speedups achieved were in the range between 3.68× and 5.24× when the number of assets were below 2^6. This shows the the GPU has reached its point where the number of registers used exceeded the limit of the GPU device, which exhausts register space. When this occurs, computation values then have to be temporarily stored to and read from the global memory, which cause the performance degradation.

Next, we examine the impact of introducing the window size in our Anticor financial portfolio implementation. The window size is a crucial part of the Anticor algorithm to determine the most optimal window size range before computing the claim and transfer weights. Four different configurations with different historical sizes are prepared: $HS = 200$, $HS = 400$, $HS = 800$, and $HS = 2000$. For each historical size, we set the number of data items to a pre-determined value so that the asset size is fixed to 2^9. The aim is to examine the impact of increasing the window size when number of assets remains constant. For each window size parameter, the search range increases with 10 timesteps (i.e., 10 days) for each iteration until the number of threshold has been reached.

Figure 4 plots the average run time executions by all multi-core CPU and GPU implementations. We did not include the results of a single CPU thread implementation as the runtimes are significantly higher, which is not even worth analyzing. For each experiment, we increase the window size to examine the performance under additional computational workloads.

For all historical sizes, we can observe that the speedup decreases as the window size increases. This is to be expected as the window increases, the compute device has additional workloads to perform. For example, the performance overhead incurred is more significant when the window size reaches to more than 100 assets (4950 total pairs), as observed in the plot. However, we notice that the scale of the speedups change for different historical sizes. For window size of 10 and $HS = 200$, the speedup achieved is much higher at 43.93× when compared to the 17.99× speedup of $HS = 400$. Similarly, only a speedup of 3.04× is achieved when $HS = 2000$. This shows that the historical size has a large influence on the overall performance of algorithm execution.

Specifically, for $HS = 200$ with a window size of 25, the average runtime increases by a factor of 16.12 when compared to the multi-core CPU of the same window size in our previus results. However, we were able to achieve an amazing speedup of 41.24× over the multi-core CPU for window size of 25. As the window size increases, the speedup decreases linearly. The speedup declines by 106 to 141 percent as the window size doubles. However, the percentage of decline increases to 215 to 262 percent when the window size increases beyond 250. Overall, the GPU achieves an average speedup of 26.61× over multi-core implementation.

We notice that for smaller range of window size ($W < 190$), the speedup is very similar to the previous experiment (without window size) under $HS = 200$. However, we observe a significant decline in speedup of almost two times (the average speedup was 3.51× for the $HS = 200, W < 190$ in the previous experiment). This indicates an additional overhead is imposed when window size parameter is introduced. Introducing the window size parameter has the additional overhead of task scheduling, as we now have two different kernels computing to perform the cross correlations concurrently.

The overall speedup for $HS = 400$ is lower than $HS = 200$ by an average factor of 1.59. The speedup difference between the two different historical size configurations (i.e., $HS = 200$ vs. $HS = 400$ and $HS = 400$ vs. $HS = 800$) somewhat varies in

TABLE II
BENCHMARK PERFORMANCE FOR VARIOUS CONFIGURATIONS (MULTI-CORE CPUS VS. SINGLE-CORE CPU, AND GPU VS. MULTI-CORE CPUS), USING $k = 180$

Asset Size	Configurations	Historical Size	Total Runtime (ms)	Speed-up
2^6	Multi-core CPUs vs. single-core CPU	200	15.78	202.53
2^6	Multi-core CPUs vs. single-core CPU	400	26.16	479.74
2^6	Multi-core CPUs vs. single-core CPU	800	24.5	2521.22
2^6	Multi-core CPUs vs. single-core CPU	2000	70.94	5800.11
2^9	Multi-core CPUs vs. single-core CPU	200	281.21	196.50
2^9	Multi-core CPUs vs. single-core CPU	400	304.7	771.90
2^9	Multi-core CPUs vs. single-core CPU	800	447.9	2132.90
2^9	Multi-core CPUs vs. single-core CPU	2000	1507.9	4397.63
2^6	GPU vs. multi-core CPUs	200	1.51	10.45
2^6	GPU vs. multi-core CPUs	400	2.04	12.82
2^6	GPU vs. multi-core CPUs	800	4.89	5.01
2^6	GPU vs. multi-core CPUs	2000	7.54	9.40
2^9	GPU vs. multi-core CPUs	200	13.8	20.37
2^9	GPU vs. multi-core CPUs	400	19.52	15.60
2^9	GPU vs. multi-core CPUs	800	36.48	12.27
2^9	GPU vs. multi-core CPUs	2000	97.94	15.39

between 1.22 to 1.84 for a wide configuration of historical sizes ($HS < 2000$), but the scale factor never goes beyond two. This shows that the GPU scales considerably as we increase the window size. Interestingly, we notice that speedup is greatly reduced by 240 to 246 percent under $HS = 400$ when the window size is small ($ws < 125$). On the other hand, the speedup achieved is almost comparable at large window size ($ws \geq 250$), with only 59 to 81 percent decline. This demonstrates that the GPU scales better for large window size.

The improvement in speedup for large window size may be due to the sudden increase of the GPU occupancy, as the amount of data items dispatched to the GPU was increased from 100 thousands to a 200 thousand in order to maintain the asset size of 500. The AMD CodeXL profiling tool further confirmed our hypothesis as we observe a significant increase in GPU occupancy (from 55 to 95 percent for $HS = 400$) when the window size exceeds 250. This illustrates that the GPU can achieve significantly a better performance when the number of work items is maximized. When the GPU is fully exhausted, we notice the GPU maintains good speedups, even when the window size increases. It can be observed that the GPU continuously maintains a speedup between 7.38× and 9.03× even after the window size increases to 600.

The bottom left of the Figure 4 illustrates the results when we further increase the historical size to 800 while keeping the asset size constant to 500. This is done by increasing the amount of data items for 200 thousands to 400 thousands. We would expect the speedup to decline by 90 to 120 percent only, but instead, it declines by approximately 184 percent. This may be potentially due to the additional synchronization overheads between multiple workgroups for final reduction phase. Since 800 historical sizes now require at least 3 work groups (i.e., $\frac{800}{256}$), the reduction phase requires additional read operations from the global memory and finally merge the results, which cause the degradation in performance. Due to this overhead, the GPU for $HS = 800$ only achieves an average speedup of 9.03× when compared to the average speedup of 16.64× for the same historical size.

To further validate the performance impact of performance degradation, we increase the historical size to 2000. We anticipate the average speedup would further detoriarate as we stress the work groups to be large. The worst speedup was observed as it never reaches beyond 4× even for smaller window size. The speedup increases slightly when the window size is in between 2^7 to 2^8. This signifies the point where the GPU occupancy is in between 90 percent to 95 percent. However, as the GPU occupancy reaches its peak, the work group synchronization overhead seems to resurface. Hence, the final average speedup is only 7.38×, when compared to the 9.03× for $HS = 800$.

TABLE III
DMA-BI RESULTS IN TERMS OF THE TOTAL WEALTH, SHARPE RATIO, AND ANNUALIZED RETURNS BASED ON THE HISTORICAL DATASETS FOR 4 MARKET INDICES.

Strategy	NYSE	S&P500	DJIA	TSX
Anticor	1.08 0.03 / 2.41%	6.20 0.19 / 12.36%	5.18 0.21 / 10.95%	0.96 -0.01 / -0.24%

Overall, we notice that the historical size range between 200 and 400 achieves the best performance for the GPU over the multi-core CPU from a series of our experiments. Within this range, the GPU achieves a minimum average speed up of 16.64× when compared to an average speedup of 7.38× for $HS = 2000$.

To validate the accuracy of our Anticor algorithm execution on the GPU, we present an experimental study of the Anticor algorithm. Four main historical datasets are used, each from different market. The first NYSE dataset comprises a selection of stocks from the NYSE market during the period 2000 to 2017. The stocks are chosen based on a number of criteria. The NYSE comprises 100 top stocks with the largest market capitalization. To avoid data-snooping bias, market capitalization was selected based on its listing at the year 2000. Hence, this represents a realistic scenario since the strategy does not know whether the same 100 stocks will continue to remain in the largest market capitalization category in the next 15 years. Similarly, the top 100 stocks (by largest market capitalization) will also be selected for other market indices such as the SP100, and TSX indices. The only exception is the DJIA since the index only comprises 30 stocks at a maximum. Hence, all 30 stocks from the DJIA will be included in the DJIA datasets. The benchmark index is chosen by the top 20 stocks (by largest market capitalization) for all markets (i.e., NYSE, S&P500, DJIA and TSX). To facilitate comparisons, all datasets will begin from Jan 2000 and ends at Dec 2017 (as of today's date). Hence, we will evaluate the performance of the Anticor algorithm during the last 15 year period.

Table III provides a performance summary of the Anticor algorithm on four different markets i.e., NYSE, S&P500, DJIA, and TSX. The performance is shown in terms of the total wealth, Sharpe ratio, as well as the annualized return. Overall, the S&P500 market generates higher returns with reasonable risks when compared to the other markets. In particular, the algorithm produces excellent and fantastic returns on the DJIA and S&P markets with the total wealth of 5.18, and 6.20, respectively. These returns are very impressive.

However, we can also observe that the Anticor algorithm fails to outperform on the TSX market. To understand this lack of performance, it is necessary to examine the overall index perfor-

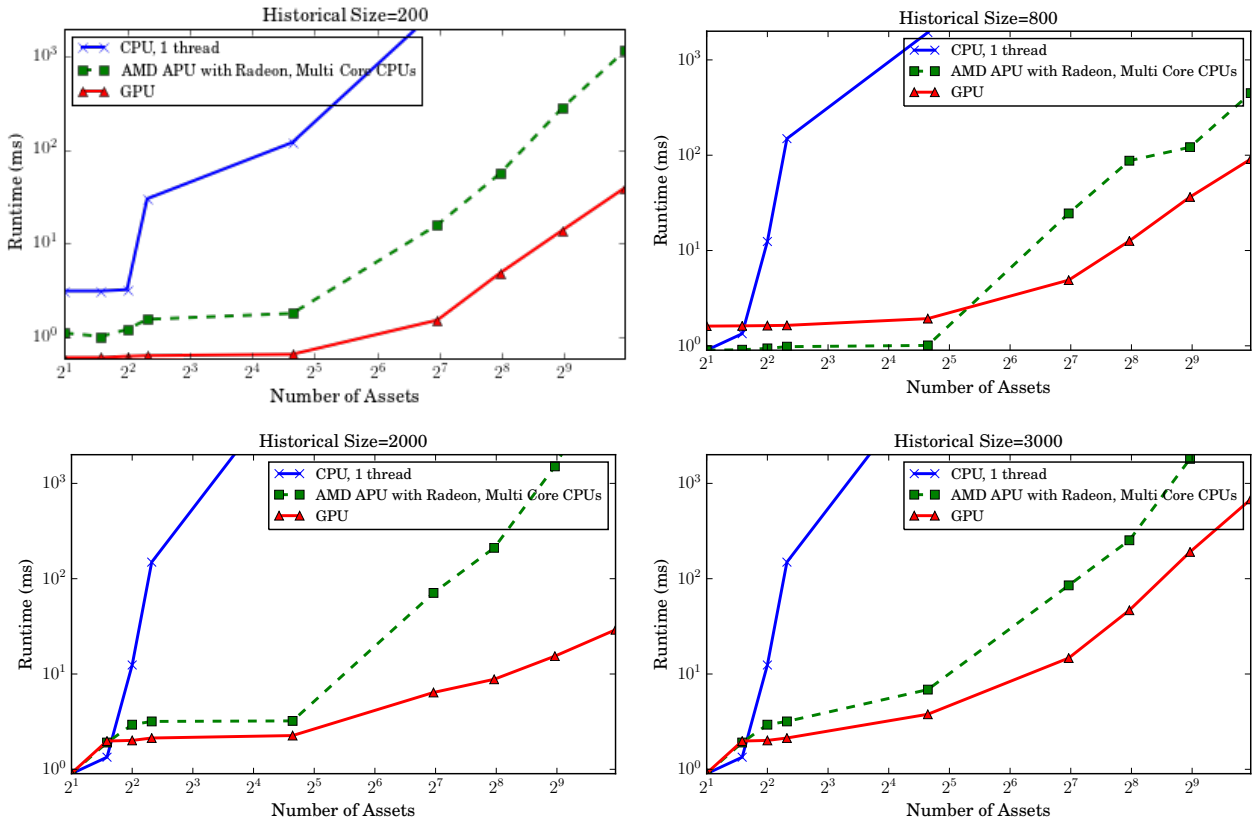

Fig. 3. The impact of runtimes across increasing number of assets. As the number of assets increases, the number of asset pairs also increase significantly, which require additional computations to compute correlations between unique combinations of all assets. The historical size is kept fixed while the number of asset size increases with each experiment. Results represent the mean of 20 runs, standard deviations are less than 1 percent.

TABLE IV
AVERAGE TOTAL WEALTH ON FOUR DIFFERENT STOCK PORFOLIOS DURING 200-2017 INVESTMENT PERIOD.

Average Total Wealth			
NYSE	S&P	DJIA	TSX
5.25	4.51	5.05	2.64

TABLE V
MAXIMUM TRADING DAILY GAINS AND LOSSES INCURRED BY THE DMA-BI AND ANTICOR ON THE DJIA STOCK MARKET PORTFOLIO.

Anticor's Top 5 Gains/Losses	
Gains (%)	Losses (%)
+45.6	-28.7
+20.6	-25.1
+20.3	-22.2
+17.6	-18.0
+16.7	-17.5

mance of both the all the four markets, especially the TSX market. Table IV shows the average performance of our portfolio on all four markets. Via close examination, we can observe that the TSX portfolios perform very poorly when compared to the NYSE, S&P and DJIA portfolios. For the last 15 years, the TSX only returned a total wealth of 2.64 on average, whereas all other stock index portfolios return above 4.5. This explains the reason for the lack of performance for the Anticor algorithm on the TSX portfolio. Nonetheless, the Anticor algorithm still achieves higher total wealth of 2.64, which is higher than the standard's S&P index average return of a total wealth of 2.34. This indicates that superiority of the Anticor algorithm to outperform the market benchmark even the index itself performs poorly.

For the DJIA dataset, the Anticor is able to achieve a total wealth of 5.18, which is almost double to that of the DJIA's standard index. This is quite impressive but since the Anticor algorithm has repeatedly given very poor performance on other 3 datasets, it is interesting to examine why Anticor gives spectacular return in this case. To examine this further, Table V shows the breakdown of the largest gains achieved by both strategies during the investment period. It can be seen that the Anticor strategy generates a very high return of 46% in a single day. It can be observed that the Anticor incurred three daily large negative returns with losses more than 20% in a single day. Further examination shows that the Anticor algorithm has a sharp ratio of 0.21. It can be clearly seen that the Anticor algorithm only incurs a few large losses in the most volatile DJIA's stock market, which do not have a large impact on the overall performance.

We conclude that both OpenCL's multi core CPU and GPU implementations significantly outperform a single thread implementation by a significant margin. This shows a clear benefit in parallelizing the Anticor portfolio selection algorithm on multi core CPUs and GPU. Furthermore, the optimized GPU implementations also clearly perform and scale better than the multi core CPUs across increasing number of assets and window size parameters. For serious performance gains, the GPU was able to achieve an average of eight- to nine-fold improvement in performance for large window size, while it was able to achieve an average of seven- to eight speedup for large asset size.

Note that the performance gain on the GPU was only achieved after code optimization. The naive GPU implementation perform poorly over a multi-core CPU by 120 percent without any code optimization. By optimizing the cross correlation procedures into multiple small kernels, this enables the creation of very large, efficient workgroups resulting in an average increase in speed-up by ten fold. The creation of efficient worksgroups also have an impact of increasing the GPU occupancy from 55 to 95 percent. There are

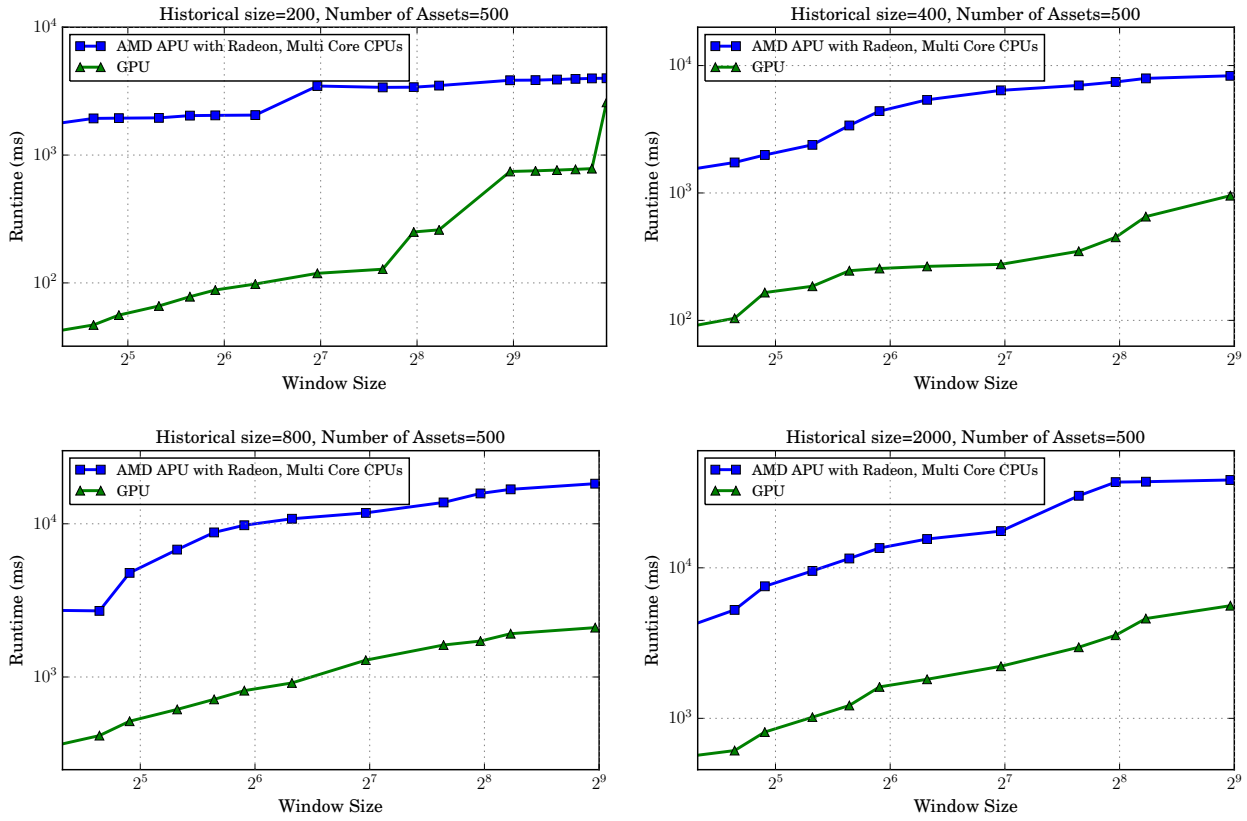

Fig. 4. The impact of average runtimes across increasing number of parameters based on 500 asset size. As the number of window size parameter increases, both the CPU and GPU has to perform extra work to compute additional correlations between the N different periods for each asset. The number of assets size is fixed to 500 while the number of parameter increases by 10 (w=+10) with each experiment. For each window size parameter, the search range increases with 10 timesteps (i.e., 10 days) until $k = 180$ (the optimal threshold). Results represent the mean of 20 runs, standard deviations are less than 1 percent.

overheads on kernel launch invocations and global memory updates. However, since the cross correlation procedure is partitioned into only three kernels, the observed kernel launch overheads were less than 0.3 ms. Also, as long there is no branch divergence, the global memory updates also take less than 55 ms for all three kernels using the built-in atomic operations.

We also see the benefits of dispatching multiple data items for each work item, as long the historical size does not exceed 2000 (this is potentially due to the limit of 256KB per compute), in which the point where register spilling occurs to global memory and is very detrimental to performance. On the other hand, if the number of data is small (≤ 200), we are not able to maximize the performance gains of the GPU to more than four speedups.

When the number of data items is assigned to per work item, the items within a range between two indexes are assigned into the private memory, before any further computations within the kernel. Such changes seem to offer a remarkable increase in performance as the kernel avoids multiple reading operations from the global memory during the computations of the logarithms, the means, the denominator and the correlation coefficient.

We have not explored the use of local memory to improve the speed up for the cross correlation procedures. The local memory is only used during the sorting operation to store a set of values for fast comparisons (line 16 of Algorithm 1), which has been shown to offer $3\times$ speed when compared to the global memory version. However, the cross correlation operations do not require synchronization between the work items, and therefore, we opted out the option of using the local memory. Certainly, as the number of threads increases per compute unit, the availability of private memory may drop significantly, which can impede the performance. A broad applicable approach is to employ some local memory to balance the use of local and private memory, which may be worth considering in the future. In the future, we may also consider optimizing the code further using vectorization. Since SIMD instructions can perform computation on more than one data item at the same time, we can further pinpoint specific computations, and perform the computations in a vectorized fashion, so they can be computed concurrently.

From our analysis, we can conclude that only multi-core CPUs and GPU implementations seem to be viable for real-time portfolio optimization analysis. The performance of a single thread CPU implementation greatly suffers as the history size increases. Analysts often use 10 to 20 years of historical data to make any investment and risk management decisions. This would require an estimation of 2000 to more than 5000 daily prices. Given this situation, a single thread CPU will not be feasible for real-time analysis. While multi-core CPU implementation has also shown good performance, the GPU overall offers even a greater performance improvement.

Our results represent performance on mid-range AMD Radeon HD 8650G card, and it is true to say that GPU architectures are continuing to undergo some very fundamental changes, particularly with the recent release of Radeon Fury cards. Such changes are likely to impact the results reported here. However, the motivation of our research work is to assess the suitability of conducting financial portfolio optimization analysis on less powerful cards that are currently available on portable devices such such as laptops. More powerful graphic cards will not be likely to available on laptops in the near future due to their high power consumptions. More importantly, our work has demonstrated the applicability of achieving high speedups with only AMD's mid-range card.

VII. Conclusion

This paper presents the first multi-core and GP-GPU accelerated implementations of Anticor algorithm, which is designed to accelerate the highly intensive computations for conducting portfolio selection and optimization. Our motivation for this work is to

offer a cost-effective and a practical solution that would allow fund and portfolio managers to conduct power intensive portfolio selection/optimization tasks on handheld computers (e.g., laptops), at his/her convenience. The implementation of the solution proposed here is relatively simple, but it delivered excellent absolute and relative performance.

Specifically, our work has demonstrated substantial gains in the use of commodity mid-range GPU (i.e., AMD Radeon graphic card) for the implementation of Anticor algorithm. We have shown that the performance gains scale very well under increasing number of assets and window sizes. Due to the nature of GPU architectures, however, there is a need to optimize the kernel code to obtain the acceleration benefits. Despite this, we presented a number of optimization techniques, which have been shown to offer nontrivial performance gains.

REFERENCES

[1] A. Borodin, R. El-Yaniv, and V. Gogan, "Can we learn to beat the best stock," *Journal of Artificial Intelligence Research*, pp. 579–594, 2004.

[2] T. M. Cover, "Universal portfolios," *Mathematical Finance*, vol. 1, no. 1, pp. 1–29, 1991. [Online]. Available: http://dx.doi.org/10.1111/j.1467-9965.1991.tb00002.x

[3] D. P. Helmbold, R. E. Schapire, Y. Singer, and M. K. Warmuth, "On-line portfolio selection using multiplicative updates," *Mathematical Finance*, vol. 8, no. 4, pp. 325–347, 1998. [Online]. Available: http://www.magicbroom.info/Papers/HelmboldScSiWa98.pdf

[4] A. Agarwal, E. Hazan, S. Kale, and R. E. Schapire, "Algorithms for portfolio management based on the newton method," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 9–16.

[5] B. Li and S. C. H. Hoi, "On-line portfolio selection with moving average reversion," in *Proceedings of the International Conference on Machine Learning*, 2012. [Online]. Available: icml.cc/2012/papers/168.pdf

[6] R. Nkomo, A. Kabundi *et al.*, "Kalman filtering and online learning algorithms for portfolio selection," Tech. Rep., 2013.

[7] B. Li, S. C. Hoi, D. Sahoo, and Z.-Y. Liu, "Moving average reversion strategy for on-line portfolio selection," *Artificial Intelligence*, vol. 222, pp. 104–123, 2015.

[8] Győrfi, Lugosi, F. Udina, and H. Walk, "Nonparametric nearest neighbor based empirical portfolio selection strategies," *Statistics and Decisions*, vol. 26, no. 2, pp. 145–157, 2008. [Online]. Available: http://www.szit.bme.hu/ oti/portfolio/articles/NN.pdf

[9] Gyorfi, Lugosi, and F. Udina, "Nonparametric kernel-based sequential investment strategies," *Mathematical Finance*, vol. 16, no. 2, pp. 337–357, 2006. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9965.2006.00274.x/abstract

[10] D. Huang, J. Zhou, B. Li, S. C. H. Hoi, and S. Zhou, "Robust median reversion strategy for on-line portfolio selection," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, pp. 2006–2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2540128.2540416

[11] B. Li, S. C. Hoi, P. Zhao, and V. Gopalkrishnan, "Confidence weighted mean reversion strategy for on-line portfolio selection," in *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2011. [Online]. Available: http://jmlr.csail.mit.edu/proceedings/papers/v15/li11b/li11b.pdf

[12] ——, "Confidence weighted mean reversion strategy for on-line portfolio selection," *ACM Transactions on Knowledge Discovery from Data*, vol. NA, p. NA, 2013.

[13] P. Perry, "Comparing olps algorithms on a diversified set of etfs," 2015.

[14] A. Ventura, "Online Computational Algorithms for Financial Markets," Ph.D. dissertation, Università degli studi di Milano-Bicocca, 2006.

[15] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, "Computing value-at-risk using genetic algorithm," *The Journal of Risk Finance*, vol. 16, no. 2, pp. 170–189, 2015.

[16] M. Dixon, J. Chong, and K. Keutzer, "Accelerating value-at-risk estimation on highly parallel architectures," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 895–907, 2012.

[17] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, "Portfolio management using particle swarm optimization on gpu," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 2012, pp. 103–110.

[18] H. J, "Stock portfolio optimization using cuda gpu," 2009.

[19] R. H. Singh, L. Barford, and F. Harris, *Information Technology: New Generations: 13th International Conference on Information Technology*. Cham: Springer International Publishing, 2016, ch. Accelerating the Critical Line Algorithm for Portfolio Optimization Using GPUs, pp. 315–325.

[20] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 02, pp. 285–329, 2014.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃijger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," 2007.

[22] S. Mittal and J. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Computing Surveys*, 2015. [Online]. Available: https://goo.gl/fGYLJm

[23] G. M. Markus Gipp and R. Manner, "Haralick's texture features computed by gpus for biological applications," *IAENG International Journal of Computer Science*, vol. 36, no. 01, pp. 66–75, 2009.

[24] C.-U. L. NAN ZHANG and K. L. MAN, "Parallel binomial american option pricing on cpu-gpu hybrid platform," *IAENG Transactions on Electrical Engineering*, vol. 01, 2013.

Amril Nazir Amril Nazir, received the PhD degree in Computer Science from University College London (UCL) in 2011. His research interests include financial engineering, high performance computing, cloud computing, big data processing, machine learning, data mining, and embedded systems. He is an Assistant Professor at the Department of Computer Science, Taif University. Previously, he worked as a senior researcher at a government RD institute.