GPU Extended Stock Market Software Architecture

Alisa Krstova(✉), Marjan Gusev, and Vladimir Zdraveski

Faculty of Computer Science and Engineering,
University Ss. Cyril and Methodius, Skopje, Macedonia
krstova.alisa@gmail.com

Abstract. We propose a stock market software architecture extended by a graphics processing unit, which employs parallel programming paradigm techniques to optimize long-running tasks like computing daily trends and performing statistical analysis of stock market data in real-time. The system uses the ability of Nvidia's CUDA parallel computation application programming interface (API) to integrate with traditional web development frameworks. The web application offers extensive statistics and stocks' information which is periodically recomputed through scheduled batch jobs or calculated in real-time. To illustrate the advantages of using many-core programming, we explore several use-cases and evaluate the improvement in performance and speedup obtained in comparison to the traditional approach of executing long-running jobs on a central processing unit (CPU).

Keywords: Stock market · GPU · Parallel programming · CUDA

1 Introduction

As more and more people become interested in getting familiar with and investing into the stock market, more research efforts are devoted for its analysis. The stock market is a complex platform which acts like an intermediary between the sellers of shares of stocks and the interested buyers. There are many details connected to the stock trading process that can be perplexing for the average investor or to a person who is just entering the market.

A good starting point to learn the intricacies of stock trading are web applications that simulate the stock market dynamics and offer an up-to-date overview of the stock market with all the relevant information (stock indexes, daily stock returns, volatility, Sharpe ratio etc.) being updated a few times per day. These web applications often act like a virtual stock market, where the users can learn how to build their investment portfolio, i.e. how to buy and sell shares in the most profitable way [1]. A crucial step in facilitating the process of making an

V. Poulkov (Ed.): FABULOUS 2019, LNICST 283, pp. 386–399, 2019.
https://doi.org/10.1007/978-3-030-23976-3_34

informed decision would be providing valuable insights about the situation on the stock market with the following use-cases:

- calculating daily/monthly/yearly stock returns,
- grouping together stocks that exhibit high correlation in returns and
- ranking stocks in terms of relevant metrics.

Note that some of these features have been integrated as part of the ByteWorx Contest, http://www.byteworx.eu/stock-contest/. Integrating such a module into the aforementioned virtual stock market system would help investors improve their risk management strategies.

Performing statistical analysis on a stock market dataset is different from applying these techniques in other fields, mostly due to the large amounts of collected data and the complex interactions between companies and individuals. This implies that constructing a web-based system that would offer all the relevant metrics re-computed in regular intervals from a stream of raw stock market data would come at a high computational cost. We aim to remedy this issue by proposing a prototype of a system which would harness the power of graphics processing units (GPUs) and the parallel programming paradigm in order to find patterns in a large stock market dataset obtained from Kaggle [2]. This dataset provides the full historical daily price and volume data for all US-based stocks and exchange-traded funds (ETFs) trading on the New York Stock Exchange (NYSE) and NASDAQ stock market and represents a good starting point for building our system.

The rest of the paper follows the following structure. Section 2 presents the related work. In Sect. 3 we describe the proposed solution to the problem of long-running tasks and the high computational cost that accompanies them. The parallelization approach this solution is based on is discussed in Sect. 4. The testing methodology used to validate the suggested concepts is described in Sect. 5 and an overview of the obtained results is demonstrated in Sect. 6. Finally, a summary of the evaluation process and concluding remarks are given in Sects. 7 and 8.

2 Related Work

The literature documents several attempts to analyze and extract knowledge from large stock market datasets using different techniques. Golan and Ziarko [3] employ a model based on the variable precision model of rough sets to acquire new knowledge from market data. There are also various stock market simulators which illustrate the principles of share trading in the form of interactive games, such as MarketWatch [4]. Many of these web applications support up to tens of thousands of users that can interact with the updated stock market data.

As we have already mentioned, a high number of users and large data volumes introduce the need for more compute-intensive operations, such as calculating user statistics, stock market indicators or identifying clusters of related stocks.

Much effort has been invested into attempting to discover meaningful relationships in data of such nature - the research presented by Gariney [5] describes several statistical measures whose integration into our system would accelerate the overall process of understanding stock market data.

Within the context of developing models for statistical analysis of data using a GPU/CUDA approach, some of the most interesting approaches include implementing the computation of pairwise Manhattan distance and Pearson correlation coefficient between data points presented in the work of Chang et al. [6]. Although not referring specifically to stock market data, the authors show that it is possible to obtain a speedup of up to 38 times when calculating this metric in comparison to the central processing unit implementation. We aim to test this claim in a stock market environment.

The hybrid approach of combining CUDA and the Message Parsing Interface (MPI) to compute the Pearson correlation coefficient described by Kijsipongse et al. in [7] offers valuable guidelines for implementing this kind of module in a distributed, possibly web-based environment. The research we propose, however, goes one step further by identifying relevant use cases for the integration of parallel computation in modern web development and testing the feasibility of this objective.

3 Proposed Solution

We propose a new web system architecture in order to decrease the load time of the virtual stock market web application by speeding up the underlying computations of relevant metrics. This section describes the extension of the traditional web architecture and the advanced use-cases that the improved web application model can be efficient for.

3.1 GPU Extended System Architecture

With large amounts of stock data being collected every day, the size of the problem at hand is scaled up, mostly due to the increased demand for web applications that deliver fast performance in analyzing this data. Web application speed is becoming more and more important for providing the impression of a fluid website experience and ultimately increasing user conversion rate. Handling thousands, sometimes even millions of records of stored or streamed stock market data to provide near real-time answers to user queries is challenging due to several factors, such as network strength, load distribution, traffic size and the nature of the computation itself. While the first factors are performance-indifferent, the problems we are trying to solve are susceptible to parallelization and thus allow room for performance improvements.

We propose extending the traditional monolithic web application architecture where the entire application is deployed onto several servers/containers by equipping each of these servers with a GPU.

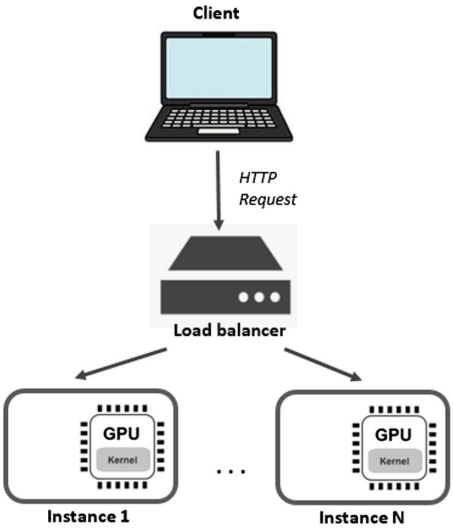

Fig. 1. Traditional monolithic architecture extended with GPU

Figure 1 presents how a HTTP request initiated by the client is first intercepted by a load balancer, which directs it to one of the available web server instances (e.g. Tomcat). This instance performs the requested computation using a CUDA kernel function which employs massively parallel computing on the built-in GPU and returns the result to the client.

3.2 Advanced Use-Cases

The aim of the proposed prototype of GPU-extended stock trading software architecture is to provide the user with an advanced, accurate and clean overview of her portfolio as well as the situation on the stock exchange of interest. The system contains several modules (functionalities):

- *Performance overview* - keeps track of the percentage change of the portfolio value for the current day, the total portfolio cost and value;
- *Transaction management* - tracks individual buy and sell transactions;
- *Visualization module* - provides stock charts for a chosen company illustrating price and volume trends for a given period of time (1 day, 1 week, 1 month etc.) and
- *Additional metrics* - determines volatility and Sharpe ratio for a given portfolio; calculates correlation coefficient of chosen company with any other company on the stock market based on past data.

We seek to optimize the computations which constitute the last module for displaying additional metrics.

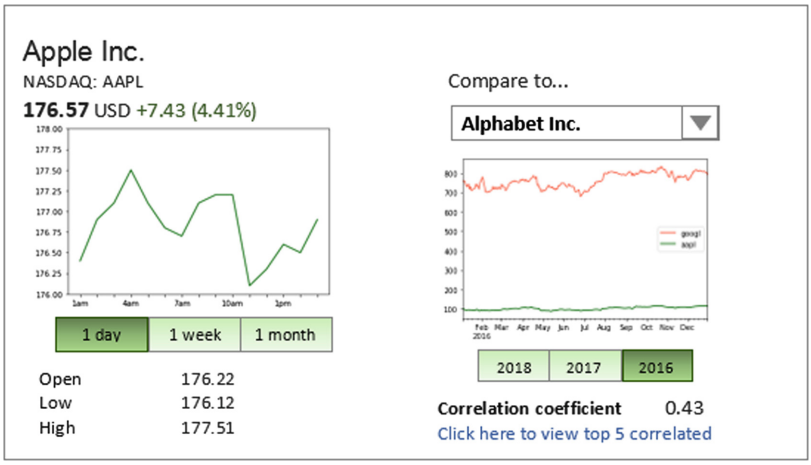

Fig. 2. User interface of the company comparison module

Figure 2 displays all relevant statistics the company view offers to the users, such as opening, closing, high and low prices for the chosen time period. The right side of the view enables the user to choose another company to compare to - a chart illustrating the closing prices across a given period for the two companies is given and the correlation coefficient is computed. The link at the bottom of the screen gives a list of the 5 companies with the closest correlation coefficient to the base choice company. The described concepts can be expanded to include other relevant metrics.

4 Parallelization Approach

We identify three scenarios related to stock market analysis that can be included as part of the *Additional Metrics* functionality illustrated in Sect. 3. More precisely, we propose computing metrics such as the Pearson correlation coefficient between stocks and the Sharpe ratio by exploiting the highly parallelizable nature of these problems. In addition, we describe a parallel CUDA approach to ranking/sorting stocks based on a metric like the Pearson correlation coefficient.

4.1 Identifying Related Stocks Using the Pearson Correlation Coefficient

From a user's perspective, the ultimate goal of buying shares is to make profit by buying stocks in companies that are expected to do well on the market, i.e. whose share price would rise. Upon inspecting current and past trends of the performance of a specific company, it can be useful to see whether another company exhibits similar or different behavior. One way to do this is by calculating the Pearson correlation coefficient between two variables, in this case two

populations of stock market data for two companies. This coefficient can help to determine how well a mutual fund is behaving compared to its benchmark index, or how a mutual behaves in relation to another fund or asset class. It is also a useful tool for building a portfolio and mitigating risk - by adding a low or negatively correlated mutual fund to an existing portfolio, diversification is increased.

Assume X and Y hold the closing prices of CompanyX and CompanyY. A Pearson correlation coefficient $r_{X,Y}$ is defined by (1), where n is the number of samples, $\bar{X}$ and $\bar{Y}$ are the means of X and Y, respectively and σ_X and σ_Y are their standard deviations.

$$r_{X,Y} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sigma_X \sigma_Y} \tag{1}$$

The value for the correlation coefficient can range from -1.0 to 1.0, where -1.0 means perfect negative correlation, whereas 1.0 indicates perfect positive correlation.

Parallel reduction is used as a common data parallel primitive to speed up the computation of the mean and standard deviation. Although perhaps not evident at first sight, according to (2), computing the standard deviation can be treated partially as a reduction problem - computing sum of squares in the numerator can be done in parallel using reduction, which is one of the basic data parallel primitives. Also, calculating the mean of the input array can be implemented using parallel summation followed by division by the length of the array, n.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} X_i^2}{n} - \bar{X}^2} \tag{2}$$

Processing large arrays which can have up to millions of elements means that multiple thread blocks must be used. The PyCUDA implementation [8] uses interleaved addressing to avoid bank conflicts and the shared memory to reuse intermediate results and data that has already been pulled from global memory.

4.2 Ranking Stocks Based on Correlation Coefficient

In addition to being able to find the degree to which two companies' shares movements are associated, another practical use-case would be to offer the user a list of most or least correlated companies with the one she is currently analyzing. For this purpose, we need to compute the Pearson correlation coefficient (1) between all pairs of company stock market data. This can be executed as a scheduled batch-job in a predefined time period, for example once a day.

The proposed approach would allow the resulting array of correlation coefficients for a given company to be constructed faster compared to a serial approach.

Upon a user request to display the most or least correlated companies for a certain company, a sorted array of correlation coefficients using the bitonic sort

algorithm is implemented in CUDA [9,10]. We opt for this algorithm because it is highly parallelizable, i.e. the data to be sorted can be efficiently distributed among the threads in the GPU.

4.3 Parallel Computation of Sharpe Ratio

The Sharpe ratio is the average return earned in excess of the risk-free rate per unit of volatility or total risk. One way to better understand this metric is by observing a "zero-risk" portfolio which has a Sharpe ratio of exactly zero. The greater the value, the more attractive the risk-adjusted return. This metric can be computed by (3), where n is the number of business days used in the calculation for and d is the daily return as a vector for the given period.

$$Sharpe = \frac{\sqrt{n} \cdot \bar{d}}{\sigma_d} \tag{3}$$

We make use of the Pandas library in Python to compute the daily returns on a closing price series for a given portfolio and reuse the aforementioned parallelized code for determining the mean and standard deviation of the returns vector.

5 Testing Methodology

In order to illustrate the discussed concepts and identify the scenarios which might result in an improvement over the traditional methods of computation, we perform several experiments to compare the proposed parallelized approach with a serial solution.

The testing environment is a desktop computer using an Intel Core i5-4200M 2.5 GHz CPU for testing the sequential implementation of the programs. An NVIDIA GeForce GT 820M graphics card is used for the CUDA-based testing. This GPU configuration allows the utilization of CUDA with compute capability of 2.1, which implements concepts like atomic functions, 3D grids of thread blocks, surface functions etc. [11].

The software tools used as part of the testing methodology were Numpy, one of the most powerful and fast libraries for scientific computing with Python for the serial CPU implementation and PyCUDA, a fast Python wrapper for the CUDA parallel computation API, which integrates seamlessly with the Flask micro web framework. We chose PyCUDA for its robustness, automatic memory management and error checking, and near-zero wrapping overhead. Wise exploitation of these concepts, as well as the massively parallel hardware offered by the GPUs can lead to the time needed for a user to receive an HTTP response being determined only by the speed of the communication channel rather than the complexity of the request.

Kaggle dataset [2] is used in the experiments. Figure 3 gives an overview of how the stock market data is distributed over the years. The x-axis represents the year the data is collected for and the y-axis gives the corresponding number

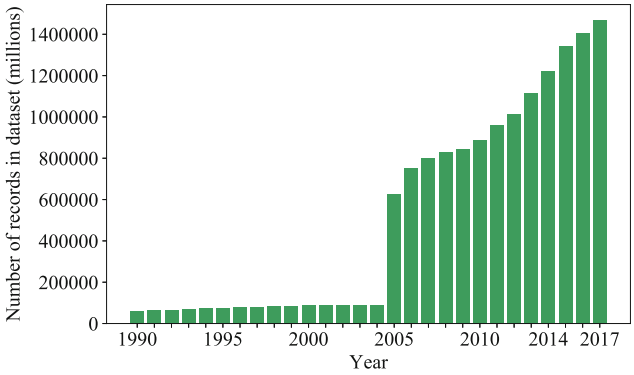

Fig. 3. Distribution of part of the records in the available dataset over the years

of available records for that year. The dates of the records in the dataset range from the beginning of 1962 until October 2017 and it can be seen that the amount of collected data has grown significantly in the period between 2005 and 2017[1], reaching a maximum of 1405977 records for 2017. The records are distributed unevenly among 7197 companies on the US stock market and provide enough data to experiment with finding correlations between older and newer data.

Our experiments aim at measuring the execution time of three different metrics. We consider the implementation, computation and evaluation of each of these metrics as three separate logical units, i.e. modules (M) denoted by:

- *M1* to calculate the Pearson correlation,
- *M2* to calculate the Sharpe ratio, and
- *M3* to calculate the stocks ranking.

The response time for the sequential execution is denoted by $T_s(M)$ and for parallel $T_p(M)$, where M refers to either $M1, M2$ or $M3$.

Afterwards, we evaluate the possible speedup obtained by the parallelization approach for each identified module, calculated by (4).

$$S(M) = \frac{T_s(M)}{T_p(M)} \tag{4}$$

The speedup is defined as the ratio of the sequential execution time to the parallel execution time, i.e. it measures the improvement in speed of execution of the specific task when using a parallel as opposed to a sequential method of processing. The larger the value of $S(M)$, the more significant the difference between the two measured times. The advantages of using this evaluation approach are several:

[1] The author of the dataset does not provide reasons for the very sharp increase in collected data between 2004 and 2005.

- Speedup illustrates well the relative performance of two systems processing the same problem and it is most commonly used in the parallel programming world.
- Speedup can also be a base for further evaluation measurements like parallel efficiency (ratio of speedup to the number of processors) which provides information on how well the available resources are used.
- We can generate informative plots to understand the behavior of the parallelized code.
- By computing the speedup for different block and grid configurations for a problem of fixed size (for example, a vector of fixed length) we can identify the optimal GPU setup which would significantly outperform sequential processing.

The experiments consist of test cases with different block sizes - we compare the efficiency of the program when using 32, 64, 128, 256, 512 and 1024 threads per block (block sizes). The blocks are launched in a grid of blocks with dynamically determined dimensions, depending on the size of the array.

Different aspects of the dataset are taken into consideration for modules $M1$ to $M3$. For $M1$, as input (the x-axis on Fig. 4) we take two arrays of equal length consisting of closing prices for two companies over the same period of time. For $M2$, the computations are also performed over a single array of closing prices, however the daily returns are calculated first. Finally, for $M3$ as input we take an array of values for the Pearson correlation coefficient between N pairs of companies.

The size of the input also varies in each module. In $M1$ for a given company, the number of available information about the closing prices ranges from 16 thousand to 1.6 million. For $M2$, in order to obtain a noticeable improvement in performance, a bigger dataset was needed and the experiments were conducted on several dataset sizes ranging from 16 thousand to 11.6 million records. Finally, for $M3$ the number of values to be sorted goes up to 1.4 million.

6 Evaluation of Results

This section describes the obtained results and evaluates the performance of the proposed parallelized approach for calculating metrics as opposed to using standard methods and libraries.

To illustrate the benefits of parallel computing we artificially extend the available dataset for the query companies by replicating and applying minor transformations to the closing prices. More specifically, we replicate the original vector to reach the desired size and add a random number between 0 and 2 to every value except the original ones. The random number added to the replicated closing prices conforms with the volatile nature of the stock market (if we exclude major political or economic events, the closing prices usually do not vary dramatically from day to day).

Figure 4 compares the execution time needed to calculate the Pearson correlation coefficient on vectors with variable number of elements. For a given

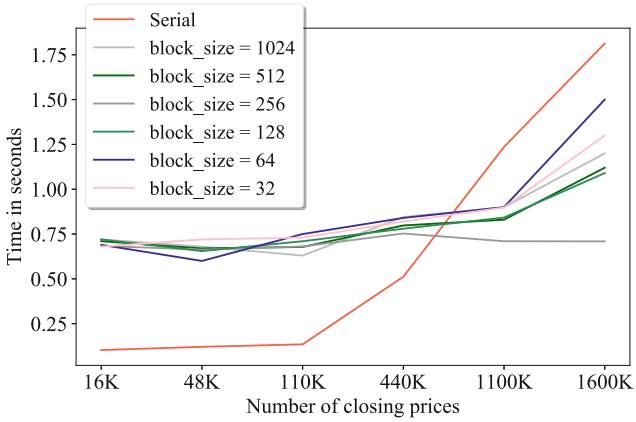

Fig. 4. Performance evaluation of $M1$ module for Pearson correlation coefficient calculation.

company, the number of available information about the closing prices ranges from 16 thousand to 1.6 million (as shown on the x-axis is a logarithmic scale of the number of closing prices, and y-axis the execution time (in seconds) of computing the $M1$ module of Pearson correlation coefficient.

One can observe that for vectors with a relatively small number of instances, the CPU version significantly outperforms our PyCUDA implementation. This is because a lot of time is lost on initializing the kernel function, copying the input vectors to and from the GPU etc. The benefit of using the parallel approach becomes evident for arrays larger than 400 thousand elements - the relatively constant CUDA execution time (around 0.7 s) is a better result than the growing value for Numpy's execution time.

After performing experiments with several different block sizes, as explained in Sect. 5, we can see that most block sizes yield similar results in performance. The block size that stands out with lowest execution time is 256 threads per block, as shown in Fig. 4.

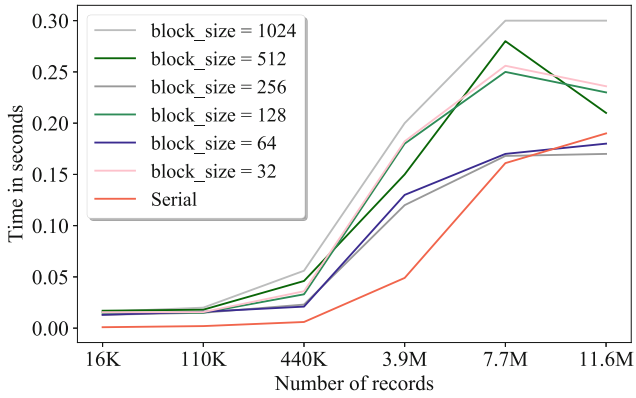

Fig. 5. Performance evaluation of $M2$ module for Sharpe ratio calculation.

A similar conclusion can be drawn for the Sharpe ratio calculation - the standard way is efficient enough for handling moderate to large-size arrays (Fig. 5). The performance starts to decline after hitting the 7-million-elements mark - this is when PyCUDA becomes more efficient. This leads us to the idea that we can combine data for several companies/portfolios to calculate the respective values for the Sharpe ratio to obtain a more significant speedup. We have again run several experiments to test the impact of the block size and conclude that the most optimal performance is achieved with 256 threads per block, with 64 threads per block performing insignificantly worse.

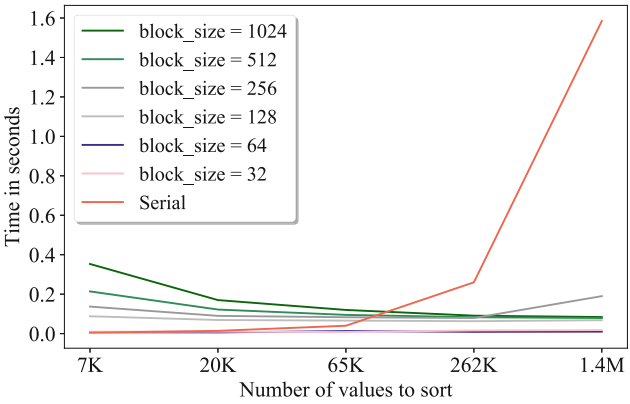

Fig. 6. Performance evaluation of $M3$ module for sorting of correlation.

Finally, we evaluate the performance of the bitonic sort algorithm on the Pearson correlation coefficient vectors computed using the parallel approach. Once again we compare the time of execution of the PyCUDA implementation and the standard Python sorting method. There is a negligible difference between the two approaches in processing vectors of moderate size. The gap in performance starts to grow rapidly on vectors of more than 20000 values - the parallel bitonic sort performs more than 4 times faster in comparison to sorting on the CPU. Figure 6 illustrates the negligible difference in execution time when running the bitonic sort with a different number of threads per block. Although the concept of implementing bitonic sort with PyCUDA has been tried out on sorting correlation coefficient values, the same technique can be applied to any other stock market metric that requires sorting.

7 Discussion

As the previous section demonstrates, GPUs can provide good performance at low computational cost (measured in both power consumption and execution time) provided there is a good utilization of the available resources. Thread

block size is a key factor in determining kernel occupancy. Kernel occupancy can be defined as the ratio of active warps on a Streaming Multiprocessor (SM) to the maximum number of active warps supported by the SM [12]. This metric is important as it provides information as to how well the parallel kernel is using the allocated GPU resources.

Multiple grid and thread block sizes can provide high kernel occupancy, however different configurations can lead to differences in execution time. We seek to explore the effect of specifying different combinations of grid and block sizes to optimize the parallelized approach for each module. In particular, we are interested to apply the hypothesis that larger block sizes lead to better results, as noted by Connors and Qasem [13].

Our experiments show that different configurations affect the performance of the CUDA implementation for the three discussed problems differently. For instance, the most optimal thread block size for computing the Pearson correlation coefficient has been shown to be 256 threads per block, leading to a speedup of 2.6 times compared to the serial implementation. The same holds for the calculation of the Sharpe ratio for a vector of stock market records - performance is best when we use 256 threads per block, yielding a speedup of up to 1.2 times.

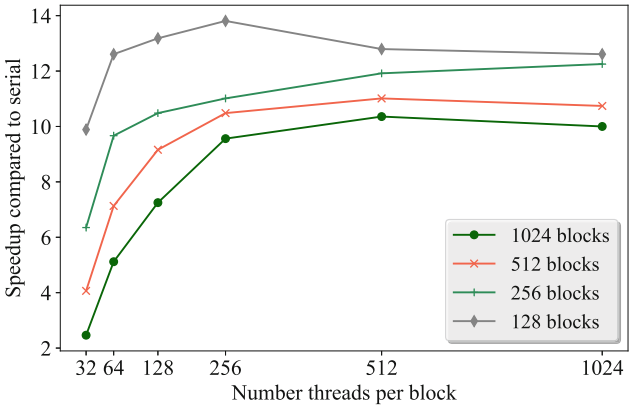

Fig. 7. Speedup diagram - bitonic sort

We conduct additional trials for the last use-case, that is sorting records using the bitonic sort algorithm. Figure 7 illustrates the dependency between using different grid and block sizes and the resulting speedup. We perform 4 series of experiments with 4 different grid sizes, i.e. 128, 256, 512 and 1024 blocks. The respective block sizes (threads per block) are given on the x-axis and the resultant speedup is shown on the y-axis. We conclude that it is best to use a grid size of 128 blocks and 256 threads per block; this allows the sorting task to execute up to 13.8 times faster than the sequential version of the program. As can be seen from the plot, for grid sizes larger than 256, a bigger number of threads in each block leads to better performance compared to using less threads for the same grid size.

It can be noticed that 256 threads per block has proven to be optimal in all three cases, which means that larger block sizes do lead to satisfactory results.

In spite of CUDA's superior performance in processing very long sequences of data, there is a lower bound to the vector size for which the parallel approach becomes more efficient than the sequential. This lower bound is different for the three problems. For the Pearson correlation coefficient, the CPU is faster in processing sequences of up to 400 thousand elements. The Sharpe ratio has an even higher threshold - the GPU accelerated version of the computation starts to outperform for sequences longer than 7 million elements. This is because Python libraries like Numpy are specifically designed to utilize the CPU resources in the most optimal way. However, this result leads us to the conclusion that in order to make the most out of the GPU execution environment, we should either increase the complexity of the problem, while still allowing room for parallelization or combine multiple simple computations in one. We show that sorting is a good example of the first concept, where the GPU accelerated bitonic sort algorithm is superior to the CPU sorting technique even for moderately large arrays of 20 thousand elements. With an optimal resource distribution, the maximal speedup is 14 times compared to the sequential version.

8 Conclusion

In this paper, we described several use-cases for a stock market software system and proposed a parallel-programming approach to optimize the computations these use-cases include.

We proposed a system architecture which has the advantage over traditional web architectures in a way that it incorporates a GPU to speed up computations. The lower time needed for calculating the result requested from the user would lead to a decrease in response time. This means that if the relevant computations are fast enough, the overall response time will depend only on the speed of the communication channel. The paper also provided a view on the user interface which encompasses the use-cases. Finally, we evaluated the performance of the parallel approaches to compute the desired metrics compared to the traditional CPU methods.

Our current research efforts are directed towards applying the described concepts to other relevant metrics, such as ranking users by their performance, identifying related stocks by means of clustering analysis etc. We also seek to explore the approach of having a single GPU thread block allocated per user in order to speed up individual computations. Furthermore, we are interested in evaluating the effect of breaking down the monolithic architecture into dedicated microservices which would also incorporate GPU-accelerated computations and then comparing the two architectures.

References

1. Peachavanish, R.: Stock selection and trading based on cluster analysis of trend and momentum indicators. In: Proceedings of the International MultiConference of Engineers and Computer Scientists, vol. 1, pp. 317–321 (2016)
2. Marjanovic, B.: Huge stock market dataset. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs. Accessed 02 May 2018
3. Golan, R.H., Ziarko, W.: A methodology for stock market analysis utilizing rough set theory. In: Computational Intelligence for Financial Engineering: Proceedings of the IEEE/IAFE 1995, pp. 32–40. IEEE (1995)
4. Marketwatch - an online virtual stock market simulator. https://www.marketwatch.com/game. Accessed 02 May 2018
5. Gariney, V.: Statistical analysis for daily forecast of stock prices (2002)
6. Chang, D.-J., Desoky, A.H., Ouyang, M., Rouchka, E.C.: Compute pairwise Manhattan distance and Pearson correlation coefficient of data points with GPU. In: 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, pp. 501–506 (2009)
7. Kijsipongse, E., Suriya, U., Ngamphiw, C., Tongsima, S.: Efficient large Pearson correlation matrix computing using hybrid MPI/CUDA. In: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 237–241, May 2011
8. Klöckner, A., Pinto, N., Catanzaro, B., Lee, Y., Ivanov, P., Fasih, A.: GPU scripting and code generation with PyCUDA. In: GPU Computing Gems Jade Edition, pp. 373–385. Elsevier (2011)
9. Mu, Q., Cui, L., Song, Y.: The implementation and optimization of Bitonic sort algorithm based on CUDA, CoRR, vol. abs/1506.01446 (2015). http://arxiv.org/abs/1506.01446
10. Ionescu, M.F., Schauser, K.E.: Optimizing parallel Bitonic sort. In: Parallel Processing Symposium: Proceedings, 11th International, pp. 303–309. IEEE (1997)
11. NVIDIA Corporation: Compute capabilities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities. Accessed 12 May 2018
12. NVIDIA Corporation, Gameworks Documentation, "Achieved occupancy". https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm. Accessed 12 May 2018
13. Connors, T.A., Qasem, A.: Automatically selecting profitable thread block sizes for accelerated kernels. In: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 442–449, December 2017