Fast Knowledge Discovery in Time Series with GPGPU on Genetic Programming

Sungjoo Ha
School of Computer Science & Engineering
Seoul National University
1 Gwanak-ro, Gwanak-gu, Seoul, 151-744 Korea
shurain@soar.snu.ac.kr

Byung-Ro Moon
School of Computer Science & Engineering
Seoul National University
1 Gwanak-ro, Gwanak-gu, Seoul, 151-744 Korea
moon@snu.ac.kr

ABSTRACT

We tackle the problem of knowledge discovery in time series data using genetic programming and GPGPUs. Using genetic programming, various precursor patterns that have certain attractive qualities are evolved to predict the events of interest. Unfortunately, evolving a set of diverse patterns typically takes huge execution time, sometimes longer than one month for this case. In this paper, we address this problem by proposing a parallel GP framework using GPGPUs, particularly in the context of big financial data. By maximally exploiting the structure of the nVidia GPGPU platform on stock market time series data, we were able see more than 250-fold reduction in the running time.

Categories and Subject Descriptors

C.1.2 [**Processor architectures**]: Multiple Data Stream Architectures—*Interconnected architectures; Parallel processors; multiple-data-stream processors (SIMD)*; J.m. [**Computer Applications**]: Miscellaneous;

Keywords

patterns; technical patterns; GPU-based acceleration; multi-GPU systems; time series data; genetic programming; parallelization; speedup technique

1. INTRODUCTION

Processing and storing time series data has a long history. It is drawing ever stronger attention, particularly with the advent of big data. An increasing number of systems produce a massive amount of time series data. Traditional sources include stock markets and various scientific computing environments. More recent sources include monitoring metrics produced by millions of nodes in cloud computing infrastructures and various sensor devices. The vast amount of time series data gathered across various fields suggests that the importance of knowledge discovery in time series will be emphasized even further.

GECCO '15, July 11 - 16, 2015, Madrid, Spain

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3472-3/15/07...$15.00

DOI: http://dx.doi.org/10.1145/2739480.2754669

Discovering knowledge in time series can be decomposed into two components. A discovery engine proposes a candidate pattern that is believed to possess some characteristics that can be interpreted as knowledge. A query engine takes this pattern and matches it against the time series database. Querying on a time series database has different aspects from querying on a database without the dimension of time. In traditional databases each record is independent of other records but in time series database, some of the attributes are related to one another by the time dimension.

There are a notable amount of research regarding the time series database (TSDB). Last *et al.* [12] described a general methodology for knowledge discovery in TSDB. Przymus and Kaczmarski [22] proposed query engine for time series data based on GPGPU and NoSQL databases. While such researches are promising, not enough mature implementations of GPGPUs are available. We exploit the problem-dependent structure with respect to GPGPU to further enhance the performance of such a system.

There are various instances of using evolutionary algorithms (EAs) for pattern mining [27][10]. Povinelli [21] proposed a genetic algorithm (GA) to find temporal patterns indicative of time series events. There are numerous researches in finding technical patterns in stock markets [20][14]. One such example is illustrated in Lee and Moon [13] where they applied genetic programming (GP) [8] for mining attractive technical patterns using a modular GP.

Extensive research has been done on parallelization of EA and local search algorithms [1][11][15][24][25][23]. The simplest form of parallelization is the master-slave model. In a master-slave model, a single population is maintained and fitness evaluation is distributed over multiple slaves; sometimes different genetic operators are assigned over them. This approach has the benefit of being simple and effective, especially when a memetic GA is used. Jaros and Tyrala [7] used a master-slave model to parallelize the evolution of communication schedules by offloading fitness evaluation to GPGPUs. Instead of offloading only the fitness function evaluation, Hidalgo *et al.* [5] parallelized the whole evolutionary process using GPGPUs. More careful design of EA to take advantage of GPU execution are also possible. Mussi *et al.* [18] organized a particle swarm optimization (PSO) algorithm that assigned a particle per block to get rid of explicit synchronization points. In a similar manner, Krömer *et al.* [9] suggested a many-threaded implementation of differential evolution where a candidate solution is assigned to each block and a vector coordinate is processed by each thread.

Another popular choice of parallelization scheme is the island model [19][16]. For an island model, multiple subpopulations are maintained; they are relatively independent from one another. Once in a while, these subpopulations exchange individuals by migration. Hrbacek and Sekanina [6] tried parallelization for Cartesian genetic programming in the domain of evolutionary circuit design under the island model.

There are also variants of GAs that employ a population that has a special topology whose spatial structure can be exploited to achieve parallelism [3]. Vidal and Alba [28] implemented a cellular genetic algorithm on multiple GPUs exploiting the toroidal structure of the population. Of course, there are approaches that combine multiple approaches together in order to further reduce the execution time [30].

The range of performance gain differs considerably depending on the domain. While some report more than a thousand-fold increase in the performance [19], most of the results fall somewhere between a few times to a few hundred times reduction in the execution time. In an application of parallel multi-swarm PSO to a task matching problem, Solomon *et al.* [26] achieved up to a 37-fold speedup over a sequential algorithm. Maitre *et al.* [17] tested coarse grain parallelization using the EASEA language [2] where they achieved a 105-fold speedup for the Weierstrass benchmark and a 60-fold reduction in the running time for the real world application of atomic model matching for chemical structures. Rocki and Suda [24] implemented the 2-opt and 3-opt local search method on GPUs to solve the traveling salesman problem (TSP). They limited the problem size and extensively used the shared memory to achieve over 500-fold reduction in running time against a sequential algorithm.

In this paper, we propose a parallel framework for GP using the CUDA platform on time series data. We describe the implementation details for this parallelization framework. We achieved a significant reduction of running time in finding interesting patterns from the time series data of Korean stock market.

The remainder of the paper is organized as follows. Section 2 formulates the problem. In Section 3 we explain the proposed parallelization framework in details. The evolution of precursor patterns using GP is described in Section 4. We present the experimental results in Section 5. Section 6 concludes the paper.

2. PROBLEM STATEMENT

A time series data set is a sequence of records each containing a set of attributes and a timestamp. Since we are dealing with time series data, it is natural to sort them on timestamps. Although the time step between two data points may not be discrete, there is always a finite number of points and we can assign an integer to refer to a single data point. If we need to work with the exact time differences between two data points, this value can be added into the record as an attribute. For example, in stock markets, we have various prices such as the opening price and the closing price being provided with a timestamp. Assuming the current record is indexed by an integer i, we can refer to the previous record with the index $i-1$. Time series data may be collected from multiple sources that share common characteristics. An example of such a case is where we have multiple companies in a given stock market. Each company stock may span a different range of time but they have the same attributes as other company stocks in the market.

Each attribute may contain categorical or real values but here we limit our discussion to real or Boolean valued attributes. We define a pattern as a conjunction of Boolean expressions. An expression is a combination of constants, comparison operators, arithmetic operators, logical operators, and attributes. Constants refer to real numbered values. Comparison operators, arithmetic operators, and logical operators may consist of operators such as $\{<,>,=,\leq,\geq\}$, $\{+,-,\times,/\}$, and $\{\wedge,\vee,\neg\}$, respectively. Depending on the domain of the time series data, we may choose to include whatever functions, in terms of the attributes, that seem appropriate. In the context of discovering attractive technical patterns for stock markets, a technical pattern may look something like "$1.1\times p_o(-1) < p_c(0) \wedge MA_{20}(0) > MA_{60}(0)$" which means that 1.1 times the opening price of the previous trading day is less than the closing price of today and the 20-day moving average of price is greater than that of 60-day moving average. The variables such as p_o, p_c, and MA are attributes or higher-level functions.

Knowledge discovery is a process of finding interesting patterns or structures from the given data. In time series data, we are interested in finding precursors to some events of interest.

Given a pattern, a time series index, and time series data, we say that an event occurred if a pattern matched at a given time series index returns *true*. Not every pattern will yield *true* all the time. Therefore there are inherent uncertainties associated with events. These uncertainties can be handled by taking the expectation of the events. This naturally extends to the case where we are interested in calculating some other values that are related to an event. For example, in a technical pattern analysis, we are interested in finding attractive technical patterns that are indicative of high profitability. We may calculate 10-day earning rates of the events associated with a technical pattern.

3. GPU ACCELERATION

3.1 CUDA Memory and Execution Model

In the CUDA platform, we define functions, called kernels, which are executed in parallel by multiple CUDA threads. Threads are organized into blocks which, in turn, are organized into grids. This is illustrated in Figure 1. The execution of threads is conducted in parallel with each unit of execution being a half warp. A warp consists of 32 threads that execute exactly the same instruction. If there are branches of instructions within a warp, the instructions are executed multiple times for different branches.

Threads within a block can cooperate with other threads within the same block using the shared memory. Thread blocks are required to execute independently which implies that they can be executed in any order. Such a design allows us to scale with the number of cores. But this also means that we cannot explicitly synchronize between blocks. For such a case where we need global synchronization, we adopt the divide-and-conquer strategy and decompose the problem into multiple kernels. A kernel launch serves as a global synchronization point.

A thread may access multiple memory spaces for data access and storage. Each thread has a private local memory space. A local memory space consists of thread local global

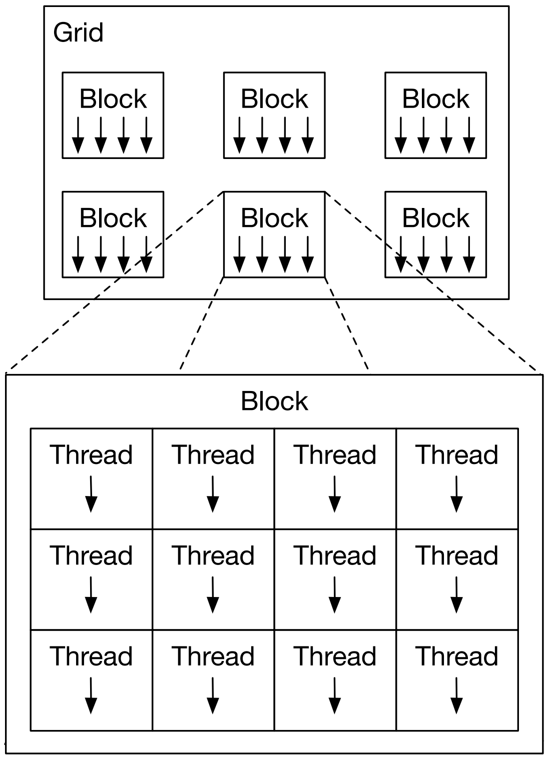


Figure 1: Grid of thread blocks

memory, which is often just called a local memory, and registers. While registers are very fast, they are scarce resources. If we allocate too many registers per thread, some of the data are spilled into the global memory which is called local memory in such a context. Each thread block has a shared memory that is shared and accessed by all the threads within the block. All threads in different blocks have access to the global memory. There are two additional read-only memory spaces: the constant memory and the texture memory. These two memory spaces are essentially a global memory optimized for different memory usages. The memory hierarchy is illustrated in Figure 2.

3.2 Parallelization Model

There are several ways to parallelize a GP program with GPGPUs. While it is possible to implement the whole GP framework on top of GPGPU, this approach has several downsides. The inherent complexity of programming on GPGPU discourages implementing the whole GP framework on GPGPU. For example, most EAs demand some form of randomness usually implemented by random number generation. There are multiple ways of providing random numbers and this imposes a design decision that has to be made. Although the current CUDA framework provides *Curand()* function for random number generation, this is not adequate for some problems [5]. Also, for hybrid GPs, overwhelming majority of the execution time is spent during the fitness evaluation of individuals. In Section 5 we show that this is the case for this work and more than 97% of the GP execution time is spent during the fitness evaluation. Therefore, a simpler approach is to only parallelize the fitness evaluation part. Not only is this approach much simpler, it also offers us the flexibility in the design of the GP framework.

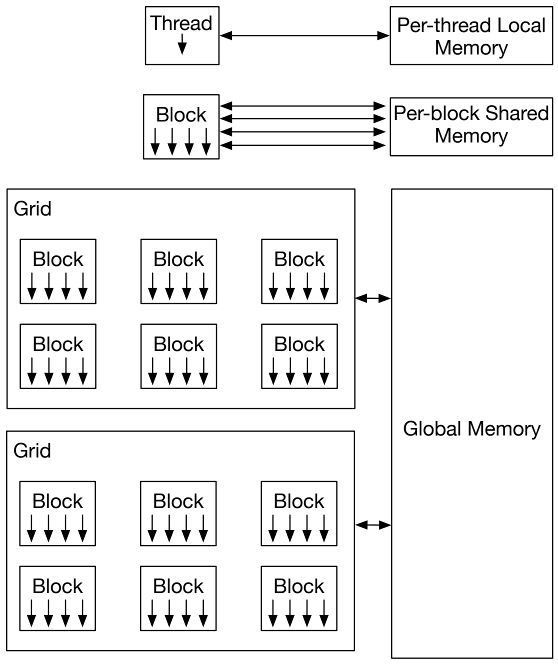


Figure 2: Memory hierarchy of CUDA framework

3.3 Implementation Details

3.3.1 Structure of Parallel Framework

We divide the program into two parts: the evaluation part which evaluates the fitness of a given pattern and the rest of the genetic programming.

A pattern is expressed using a postfix notation and passed to the GPU in a form of an array. Such postfix trees can be easily evaluated with a stack. An obvious way to pass the pattern tree to GPU is through the constant memory. A pattern tree is small, relative to the whole data, and needs to be broadcasted to all the threads in exactly the same way which fits the memory access assumptions imposed on the constant memory.

We can extend the framework by using multiple GPUs. The time series data set is partitioned into different chunks and distributed over multiple GPUs. We can load the data once and perform asynchronous memory copy only for the newly created individuals and their partial aggregate results. A global synchronization barrier collects the partial evaluation results from GPU devices and aggregates the results as a whole in the host.

Since we are matching a pattern against the time series data and collecting the calculated aggregates, only a small amount of data is transferred between the host and the GPUs. This is desirable since the data transfer between the host and the GPUs is the top bottleneck candidate for the whole process. Partial aggregates from each kernel are passed to host using streams.

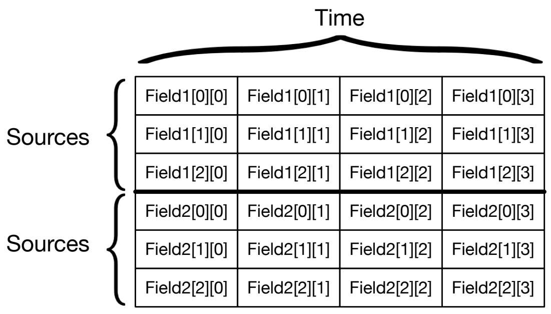


Figure 3: Layout of the time series data in the global memory. The whole data is divided into several chunks. Each chunk contains only a single attribute of the whole time series data. An attribute of a single input source is stored in a contiguous memory space. This spans multiple input sources to form a conceptual 2-dimensional array.

3.3.2 Memory Utilization

Each time series source and its related fields are arranged as a 1-dimensional array per field. This is illustrated in Figure 3. A warp accesses the same attribute of the same input source for different time indices. This suggests that we had better place these values in consecutive slots. When a single input source is processed, the warp jumps to the next input source and this behavior is naturally exploited by placing input sources consecutively.

We will match a pattern by moving along with time per source at a time; this leads to a better memory access pattern if we arrange the data correctly. Each source can span a different range of time, and in such cases we add dummy data to pad each source which reduces the complexity of the program. All of the time series data is stored in the global memory and passed to GPUs. When using multiple GPU devices, we can divide the data into an equal number of input sources and assign the divided data to multiple devices. A pattern is passed asynchronously to multiple devices using streams.

Each thread will accumulate the result of its calculation into the shared memory. This is later accumulated into a single number per block by parallel reduction. The partially accumulated values computed in blocks are returned to the host and a sequential reduction at the host is performed to compute the final result of the pattern match. If some calculation depends on multiple input sources, we accumulate the temporary values in the global memory and launch a different kernel for reduction. This does not cause a notable performance loss because hardware overhead of a kernel launch is negligible and software overhead is low [4].

The stack structure maintained by each thread is another aspect that needs deliberation. If we limit the size of the tree to a reasonable size, the stack part can position at the shared memory which is much faster than the global one. But the limitation on the size of the tree may be too much of a constraint for some domains. Not only that, in some cases, it is possible to achieve better performance by using a thread local global memory. Usually, the shared memory is much faster than the local memory but the actual performance may vary depending on the memory access pattern and the resource constraints imposed by the hardware. If the usage of the shared memory hinders the utilization of a large number of threads, it is better to use the local memory instead.

3.3.3 Single vs. Double Precision Floating Point

One decision to make is the choice between single and double precision for floating point numbers. While in CPU we don't suffer a notable speed loss by choosing double precision, in GPU it is not the case. When we experimented on our machine, we saw roughly 2-fold speed differences. Not only that, a double precision floating point number will use twice the number of registers compared to a single precision floating point. Therefore, it is reasonable to work with single precision floating point numbers unless it is absolutely necessary to have high precision. On CPU, accumulating many small numbers may lead to a significant numerical error. This happens because at the later stages of accumulation, we add large numbers with small numbers. On GPU this effect is somewhat mitigated because we perform parallel reduction and add numbers of comparable sizes.

4. EVOLUTION OF PATTERNS

A steady state genetic programming is used to evolve precursor patterns. A total of 500 individuals are evolved, where each individual represents a precursor pattern. Each individual is randomly initialized to a tree of depth 3.

The crossover operator chooses a random subtree of a parent and swaps it with a random subtree of the other parent to create a new offspring. For mutation, we choose a random subtree and replace it with a randomly created subtree.

Once an offspring is created, local optimization is applied until there is no improvement by corresponding local search operators. The local optimization traverses each node and changes the constants according to their types.

The fitness of an individual should reflect our notion of how interesting are the events associated with the precursor pattern. Here, we test with the problem of finding attractive technical patterns in a stock market. We assume that a technical pattern is attractive if it is profitable and statistically frequent.

The profitability of a technical pattern r is defined to be the expected earning rate of r after k trading days:

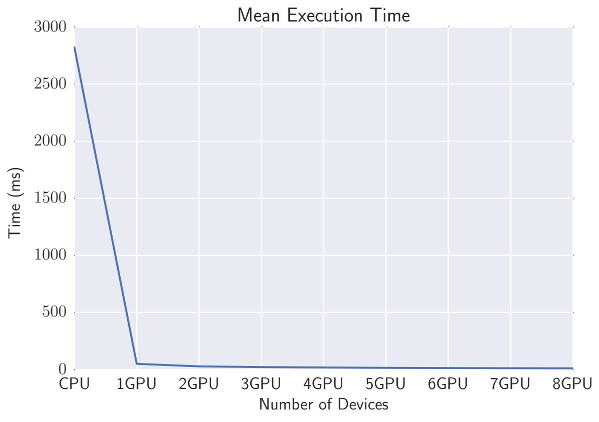
$$E_k[r] = \frac{1}{|R(r)|} \sum_{(i,j)\in R(r)} \frac{p_c(i,j+k)}{p_c(i,j)} \qquad (1)$$

where $R(r) = \{(i,j)|r \text{ matches company } i \text{ on trading day } j\}$ and $p_c(i,j)$ is the closing price of company i at trading day j.

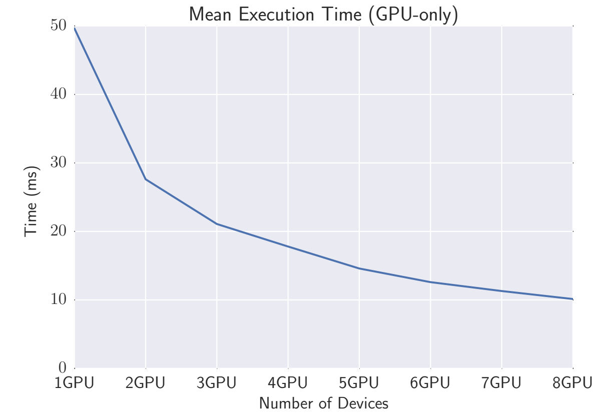
To account for the frequency of a given pattern, we define the attractiveness of a technical pattern as follows:

$$f(r) = \begin{cases} E_k[r] & \text{if } |R(r)| \geq k \\ 0 & \text{otherwise.} \end{cases} \qquad (2)$$

The constant k is a predefined threshold value to determine whether the given pattern is frequent or not in a statistical point of view. The fitness of an individual is computed by Equation 2.

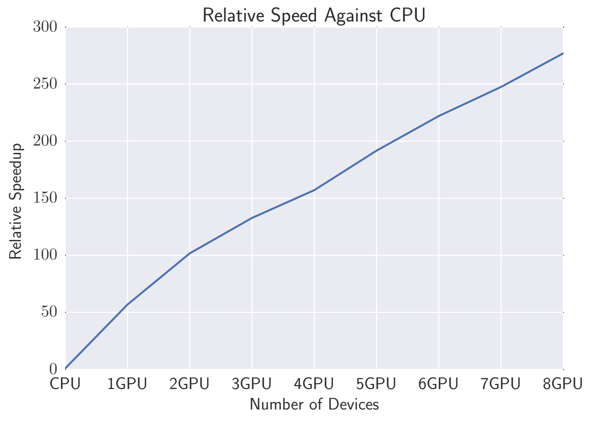


(a)

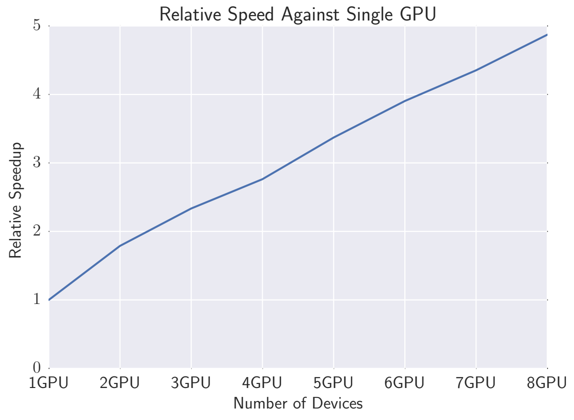


(b)

Figure 4: Mean execution time of a fitness evaluation function for different numbers of devices. (a) Comparison of the execution time among CPU and various number of GPU devices. (b) A close look at the difference over the numbers of GPU devices.



(a)



(b)

Figure 5: Relative speedup with different numbers of devices. (a) Relative speed against the CPU-only version. The speed of CPU is 1. (b) Comparison among multiple GPUs.

5. EXPERIMENTAL RESULTS

The experiment is conducted on the Korean stock market ranging from 2000 to 2014.

We run the sequential and CPU portion of the program on Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz on Linux 3.2.0. For GPU parallelization, four GeForce GTX 690 cards, where each card has two devices leading to a total of eight GPU devices, were used. Programs were compiled using clang compiler and nvcc compiler.

A single GP generation, even though it is a steady-state GP, takes fairly long with the CPU version of the code. Therefore, we report the time it took to evaluate various pattern trees created during a typical GP run. We test a sequential version of the program implemented in CPU, a parallelized version of the program using a single GPU device, and one using multiple GPU devices.

A typical execution of GP for 50 generations (cut for this test) using 8 CUDA devices took around 120 $\sim$ 180 seconds to terminate. Roughly 17,000 to 21,000 fitness evaluations are performed during these 50 generations. Needless to say, the local optimization routine spent the majority of the execution time. For a typical GP run using 8 GPU devices, the local optimization took more than 97% of the time.

Appropriate thread/block/grid sizes are determined by cross validation. It is imperative that we choose these numbers carefully because they affect the occupancy of multiprocessors on GPUs. This, in turn, affects the performance of the whole system. It is not always better to allocate as many threads per block as possible due to the memory access pattern of the program and how instruction level parallelism is exploited. For this experiment, we used 512 threads per block and 256 blocks per grid. We only used a single grid.

5.1 Performance Comparison

We compare the execution time for evaluating a typical pattern tree and the relative speed over the numbers of GPU devices. Figure 4a compares the mean execution time of the fitness evaluation for a given tree. It is clear that parallelization using GPU leads to a dramatic performance gain. Switching from a sequential algorithm to a parallel version using a single GPU device resulted in 56-fold reduction of the execution time. The relative speeds against CPU with various numbers of GPU devices is shown in Figure 5a. We

# Devices	Relative Speed			
	Local		Shared	
	μ	σ	μ	σ
CPU	1.0	0	1.0	0
1 GPU	56.9	3.8	35.8	2.8
2 GPU	101.7	7.2	65.0	5.5
3 GPU	132.7	11.1	88.0	8.1
4 GPU	157.1	15.5	107.2	10.7
5 GPU	191.7	19.9	130.3	13.7
6 GPU	222.0	24.5	151.5	16.6
7 GPU	247.5	29.3	172.2	19.5
8 GPU	277.0	36.8	188.7	22.4

Table 1: Relative speeds against CPU with different numbers of GPU devices. Note that the thread local global memory is faster than the shared memory. The average relative speed is denoted by μ and σ represents its standard deviation.

see a strong linear relationship between the relative speed and the number of GPU devices. Fitness evaluation using 8 GPU devices was roughly 277 times faster than that of the CPU version of the program. Exact figures are provided in Table 1.

A close inspection of execution time shows a clearer picture of speed gains by additional introduction of GPU devices. Figure 4b shows that the execution time steadily decreases as GPU devices are increasingly added. The strong linear relationship shown in Figure 5b indicates that our parallelization framework scales well for this particular problem. While adding an additional device does not yield 100% performance gain possible, it roughly yields 80% gain per additional device. The strong scaling of the result suggests that there is room for additional performance gain by adding more GPUs.

5.2 Effect of Tree Sizes

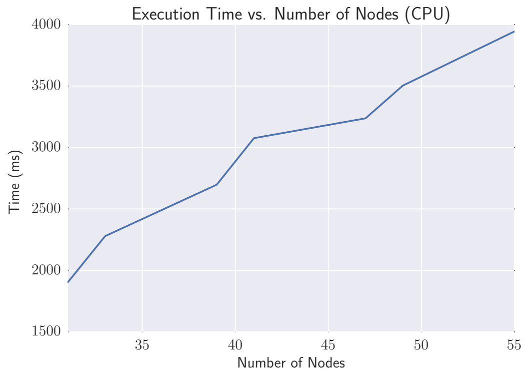
We analyze the effect of the tree size on the execution time. The size of a pattern tree corresponds to the number of nodes in the tree. A large tree is relatively more costly to evaluate than a smaller one. This behavior is clearly shown in Figure 6a. As the number of nodes per tree increases from 31 to 55, the time for a single fitness evaluation nearly doubles with the CPU version of the code. In Figure 6b we see that using more devices leads to a slower increase of the execution time as the tree grows bigger. Thus, parallelization allows us to build increasingly complex trees. Exact figures are given in Table 2.

While a simpler tree is ideal for human interpretation, it might not be able to capture complex underlying behavior that can only be expressed by a complex tree. If we wish to explore such representations, we need to be able to evolve increasingly complex trees. The experiment result suggests that by adopting the parallel GP framework, we can expect additional performance gain from this.

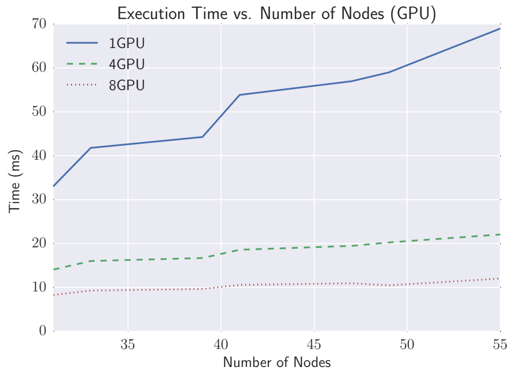
5.3 Local vs. Shared Memory

While it is widely known that the shared memory is fast and the thread local global memory is slow, relatively, it may not be the case all the time. In the CUDA platform, the shared memory is allocated to blocks which are executed on a single multiprocessor on GPU devices. Allocating too much shared memory per block prevents multiple blocks to be executed on a single multiprocessor and reduces the occupancy. When a high occupancy of the multiprocessor is attained, the high latency of the global memory is hidden by switching to other warps being not stalled.

Since we have to maintain a stack per thread, a relatively large amount of shared memory has to be allocated. The performance penalty of storing a large array in the shared memory is shown in Table 1. Although the thread local global memory has high latency, this is effectively hidden by high occupancy. When a warp from one block is stalled by slow memory access, the multiprocessor simply processes some other warp that is not stalled. This led to roughly 1.5-fold speedup.



(a)



(b)

Figure 6: Execution time for different sizes of pattern trees. (a) The increase in execution time as the pattern tree grows. (b) Comparison for different numbers of GPU devices. Increasing the number of devices led to relatively slower increase of execution time with respect to the number of nodes.

6. CONCLUSIONS

We proposed a parallel framework for knowledge discovery in time series data using GP and GPGPU. We experimented on Korean stock market from 2000 to 2014 to find attractive technical patterns using the parallel GP framework. The experimental results show that this framework scales well and by using 8 GPU devices, we could reduce the execution time up to roughly 270 times that of a sequential version of the program.

Tree Size	CPU	1 Device	2 Devices	3 Devices	4 Devices	5 Devices	6 Devices	7 Devices	8 Devices
31	1904.1	33.0	19.4	15.9	14.0	11.6	10.1	9.3	8.2
33	2278.5	41.7	23.9	18.6	16.0	13.1	11.4	10.3	9.2
39	2696.9	44.2	24.9	19.4	16.7	13.6	11.8	10.7	9.6
41	3075.4	53.8	29.4	22.1	18.5	15.2	13.0	11.6	10.5
47	3236.8	56.9	31.1	23.3	19.4	15.9	13.7	12.2	10.9
55	3943.4	68.9	36.7	27.0	22.0	17.9	15.2	13.4	12.0

Table 2: Mean execution time (ms) for various tree sizes

Tree Size	CPU	1 Device	2 Devices	3 Devices	4 Devices	5 Devices	6 Devices	7 Devices	8 Devices
31	184.37	0.32	0.20	0.11	0.07	0.25	0.25	0.23	0.72
33	103.16	2.53	1.13	0.77	0.60	0.62	0.56	0.44	0.72
39	55.55	1.18	0.52	0.31	0.24	0.31	0.29	0.39	0.80
41	68.27	2.48	1.09	0.68	0.50	0.53	0.49	0.48	0.81
47	156.64	3.23	1.29	0.87	0.63	0.54	0.50	0.43	0.68
55	73.37	3.00	1.05	0.59	0.44	0.49	0.52	0.32	0.71

Table 3: Standard deviation of execution time (ms) for various tree sizes

We described how the framework was implemented by using multiple GPUs in CUDA platform. A careful choice of the structure of parallel framework and the memory utilization method, which led to excellent speedup, was discussed in details. Specifically, by choosing the local memory over the shared memory for the storage of stack structure led to more than 1.4-fold speed gain.

Note that the acceleration attained by using multiple GPGPUs does not influence the quality of the resulting GP solutions because we only parallelize the fitness evaluation and the rest of the evolutionary technique is left unmodified. The faster execution of the fitness evaluation naturally allows the evolutionary algorithm to search for better solutions given a restricted time budget. For knowledge discovery in time series data, we can leverage this fact and evolve more diverse yet interesting individuals.

Future work would include extending our framework to exploit the population level parallelism by using island model. Parallelism obtained by our framework can easily be extended by exploiting the parallelism on population level using multiple machines. The evolution of the population based on island model will lead to rich diversity of individuals which is desirable for knowledge discovery. While the current framework mainly focuses on the thread level parallelism, to reach the maximum performance gain, additional attention to the instruction level parallelism is required [29].

7. ACKNOWLEDGMENTS

This work is the result of a commercial project conducted at Optus Investment Inc. The work was also partly supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609). The ICT at Seoul National University provided some research facilities for this study.

References

[1] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10, 1998.

[2] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. *Parallel Problem Solving from Nature - PPSN VI*, volume 1917, pages 891–901, 2000.

[3] M. J. Gibson, E. Keedwell, and D. A. Savic. Understanding the efficient parallelisation of cellular automata on CPU and GPGPU hardware. *Genetic and Evolutionary Computation Conference*, pages 171–172, 2013.

[4] M. Harris. Optimizing parallel reduction in CUDA. Technical report, nVidia, 2008.

[5] J. I. Hidalgo, J. M. Colmenar, J. L. Risco-Martín, C. Sánchez-Lacruz, J. Lanchares, O. Garnica, and J. Díaz. Solving GA-hard problems with EMMRS and GPGPUs. *Genetic and Evolutionary Computation Conference*, pages 1007–1014, 2014.

[6] R. Hrbacek and L. Sekanina. Towards highly optimized cartesian genetic programming: From sequential via SIMD and thread to massive parallel implementation. *Genetic and Evolutionary Computation Conference*, pages 1015–1022, 2014.

[7] J. Jaros and R. Tyrala. GPU-accelerated evolutionary design of the complete exchange communication on wormhole networks. *Genetic and Evolutionary Computation Conference*, pages 1023–1030, 2014.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[9] P. Krömer, V. Snásel, J. Platos, and A. Abraham. Many-threaded implementation of differential evolution for the CUDA platform. *Genetic and Evolutionary Computation Conference*, pages 1595–1602, 2011.

[10] Y.-K. Kwon and B.-R. Moon. Personalized email marketing with a genetic programming circuit model. *Genetic and Evolutionary Computation Conference*, pages 1352–1358, 2001.

[11] W. B. Langdon. Graphics processing units and genetic programming: An overview. *Soft Computing*, 15(8): 1657–1669, 2011.

[12] M. Last, Y. Klein, A. Kandel, and A. K. Knowledge discovery in time series databases. *IEEE Transactions on Systems, Man, and Cybernetics*, 31:160–169, 2001.

[13] S.-K. Lee and B. R. Moon. A new modular genetic programming for finding attractive technical patterns in stock markets. *Genetic and Evolutionary Computation Conference*, pages 1219–1226, 2010.

[14] P. Lipinski. ECGA vs. BOA in discovering stock market trading experts. *Genetic and Evolutionary Computation Conference*, pages 531–538, 2007.

[15] T. V. Luong, N. Melab, and E.-G. Talbi. Parallel local search on GPU. Research Report RR-6915, 2009.

[16] T. V. Luong, N. Melab, and E.-G. Talbi. GPU-based island model for evolutionary algorithms. *Genetic and Evolutionary Computation Conference*, pages 1089–1096, 2010.

[17] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. *Genetic and Evolutionary Computation Conference*, pages 1403–1410, 2009.

[18] L. Mussi, Y. S. Nashed, and S. Cagnoni. GPU-based asynchronous particle swarm optimization. *Genetic and Evolutionary Computation Conference*, pages 1555–1562, 12-16 July 2011.

[19] P. Pospichal, J. Jaros, and J. Schwarz. Parallel genetic algorithm on the CUDA architecture. *Applications of Evolutionary Computation*, pages 442–451, 2010.

[20] J.-Y. Potvin, P. Soriano, and M. Vallée. Generating trading rules on the stock markets with genetic programming. *Computers and Operations Research*, 31 (7):1033–1047, June 2004.

[21] R. J. Povinelli. Using genetic algorithms to find temporal patterns indicative of time series events. *GECCO 2000 Workshop: Data Mining with Evolutionary Algorithms*, pages 80–84, 2000.

[22] P. Przymus and K. Kaczmarski. Time series queries processing with GPU support. *17th East European Conference on Advances in Databases and Information Systems*, pages 53–60, 2013.

[23] A. K. Qin, F. Raimondo, F. Forbes, and Y.-S. Ong. An improved CUDA-based implementation of differential evolution on GPU. *Genetic and Evolutionary Computation Conference*, pages 991–998, 2012.

[24] K. Rocki and R. Suda. Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem. *International Conference on High Performance Computing and Simulation*, pages 489–495, 2012.

[25] S. Shao, X. Liu, M. Zhou, J. Zhan, X. Liu, Y. Chu, and H. Chen. A GPU-based implementation of an enhanced GEP algorithm. T. Soule and J. H. Moore, editors, *Genetic and Evolutionary Computation Conference*, pages 999–1006, 2012.

[26] S. Solomon, P. Thulasiraman, and R. K. Thulasiram. Collaborative multi-swarm PSO for task matching using graphics processing units. *Genetic and Evolutionary Computation Conference*, pages 1563–1570, 2011.

[27] K. C. Tan, Q. Yu, and T. H. Lee. A distributed evolutionary classifier for knowledge discovery in data mining. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 35(2):131–142, 2005.

[28] P. Vidal and E. Alba. A multi-GPU implementation of a cellular genetic algorithm. *IEEE Congress on Evolutionary Computation*, pages 1–7, 2010.

[29] V. Volkov. Better performance at lower occupancy. *GPU Technology Conference*, 2010.

[30] S. Zhang and Z. He. Implementation of parallel genetic algorithm based on CUDA. *Advances in Computation and Intelligence*, volume 5821 of *Lecture Notes in Computer Science*, pages 24–30, 2009.