# CUDAGRN: Parallel Speedup of Inferring Large Gene Regulatory Networks from Expression Data Using Random Forest

Seyed Ziaeddin Alborzi[1], D.A.K. Maduranga[1], Rui Fan[1], Jagath C. Rajapakse[1,2], and Jie Zheng[1,3]

[1] Bioinformatics Research Centre, School of Computer Engineering, Nanyang Technological University, Singapore 639798
{seyed1,Kasun1}@e.ntu.edu.sg,
{FanRui,ASJagath,ZhengJie}@ntu.edu.sg
[2] Department of Biological Engineering, Massachusetts Institute of Technology, USA
[3] Genome Institute of Singapore, A*STAR (Agency for Science, Technology, and Re-search), Bio-polis Street, Singapore 138672

**Abstract.** Reverse engineering of the Gene Regulatory Networks (GRNs) from high-throughput gene expression data is one of the most pressing challenges of computational biology. In this paper a method for parallelization of the Gene Regulatory Network inference algorithm, GENIE3, based on GPU by exploiting the compute unified device architecture (CUDA) programming model is designed and implemented. GENIE3 solves regulatory network prediction by developing tree based ensemble of Random forests. Our proposed method significantly improves the computational efficiency of GENIE3 by constructing the forest on the GPU in parallel. Our experiments on real and synthetic datasets show that, CUDA implementation outperforms sequential implementation by achieving a speed-up of 15 times (real data) and 14 to 18 times (synthetic data) respectively.

**Keywords:** Gene regulatory network, Random forests, GPU, compute unified device architecture (CUDA).

## 1 Introduction

A set of DNA portions which collaborate together and with other objects control RNA and proteins expression levels in a cell is called a Gene Regulatory Network (GRN). Predicting GRN is critical for perceiving the functioning and development of biological organisms [1]. Due to progresses in high-throughput gene expression patterns profiling with DNA microarrays and prevalence of expression data, reverse engineering of GRN from biological data is now widely used for understanding the underlying mechanisms. However it is still one of the most challenging tasks in bioinformatics and systems biology. The ability of GRN models to precisely predict gene expressions would help find interrelated genes in a biological process in addition to exploring how a system of genes is influenced by drugs. There are several different methods to predict GRN, including relevance networks [2], empirical Bayesian networks [3], Boolean networks [4,5], Bayesian networks [6,7], and neural network [8]. In spite of intense studies, GRN inference approaches still suffer from low performance. The two main reasons are their incapability of modelling inherent complexities of biological processes and the difficulty to handle high dimensional data (which include expressions of thousands of genes). On account of recent advancement in high-throughput technologies, large datasets are frequently available, thus algorithms and software of high-performance computing for GRN inference with high accuracy is becoming more important for the current research in systems biology.

Within this context, Huynh et al. has applied Random forests to GRN inference in order to tackle the above difficulties [9], because the Random forests method has become popular in handling large datasets as well as high dimensional data [10,11]. Their method, namely GENIE3, was one of the best performers in the DREAM4 in Silico challenge for GRN reverse engineering [12]. Even though it infers GRN with a higher accuracy than other similar methods, it still takes a significant amount of time even for a dataset of moderate size (e.g. less than 50 genes).

In this paper, we present a novel method to accelerate the GENIE3 algorithm based on the model of CUDA programming. In order to increase the speed of GENIE3, for each forest, trees grow in parallel inside GPU. Also, for gaining efficiency, shared memory for fast I/O is exploited. We evaluate our approach for several simulated datasets and one real dataset. Our parallelized approach (named CUDAGRN) is able to achieve a speed-up of 15 times on the real dataset on NVidia Quadra 600 in comparison to the sequential algorithm of GENIE3.

Several methods in computational science and technology have been implemented to run on a GPU in CUDA environment. For instance, GPU implementations have been reported for Smith-Waterman algorithm for sequence alignment [13], robotic multisensory perception [14], structured Bayesian mixture [15], image processing methods [16], mutual information estimation algorithm [17], a PoissonBoltzmann equation solver [18] and biomolecules Del-Phi [19]. To our knowledge, CUDAGRN is one of the first few attempts to parallelize a GRN inference algorithm, which will find applications in many biological problems involving high-throughput data and large regulatory networks.

## 2 Method

### 2.1 Sequential Algorithm of GRN Inference

There is an assumption of GENIE3, apart from random noises of the regulatory network, that the gene expression of an individual gene is a function of the expression levels of all other genes. It is assumed that the function defining the expression of gene i can be written in the following formula:

$$Y_j^i = f_i(Y_j^{i*}) + \epsilon_j, \forall_j \in \mathit{All\ experiments} \tag{1}$$

where $Y_j^{i*} = [(Y_j^1, ..., Y_j^{i-1}, Y_j^{i+1}, ..., Y_j^p]$, is the list of input samples, containing values of expression in the $j^{th}$ experiment of all genes excluding gene $i$ and $\epsilon_j$ is a random fluctuation with mean of 0. Moreover, GENIE3 algorithm assumes that the function $f_i$ only uses the expression of the genes in $Y^{i*}$ that regulates gene $i$ directly. These are the genes with an edge linked to gene $i$ in the final output network. Constructing regulatory edges connecting to gene $i$ will be finding genes whose expression levels are predictive of the expression of gene $i$. In terminology of machine learning, the problem is a feature selection problem in regression [20].

Each function of $f_i$ is nonlinear [9] and it has to take into account the expression of a number of genes. Hence, it is required to be fast. Generally, tree-based ensemble approaches, particularly Random forests, are methods of choice to fulfil this purpose. Random forests method is scalable, fast and does not assume the nature of the functions. Also, it can cope with a higher number of features and nonlinear functions [21].

In 2001, Brieman introduced the method of Random forests [22]. From the same dataset, it constructs several decision trees using randomly sampled variables and bootstrapping to generate variant trees to work as an ensemble classifier. In bootstrapping, for each tree new datasets are created uniformly by sampling with replacement cases from the training dataset. Then, these produced bootstraps are used for building trees which are finally aggregated into a forest. It has been demonstrated to be efficacious for datasets which are large and have missing attributes values [22,23]. Two parameters can be configured during the Random Forest training. One is the number of trees which can be adjusted by the user, and the other one is the number of attributes to consider in each split (denoted by $K$). By tweaking the two parameters, the result can be optimized. Building many decision trees is inefficacious when the trees need to be constructed independently from each other. When the number of trees in the forest is large, a parallel implementation of random forest has the potential to achieve considerable speed-up. On the other hand, it might be an ineffective approach only a small number of trees in the forest.

The GENIE3 algorithm uses the tree-based random forest method to predict a regulatory network. The main idea of the random forest method for inferring a network is to break the problem of constructing a network with $p$ genes into $p$ independent sub-problems. Each sub-problem is defined by a unique learning sample consisting of a pair of input-output sets of the $i^{th}$ gene (denoted by $LS$) from which the network can be inferred. For instance, the learning sample of gene $i$ is as follows:

$$LS_j^i = (Y_j^{i*}, Y_j^i), j = 1, 2, ..., N \quad (2)$$

where $N$ is total number of samples for each gene, $Y_j^{i*}$ is the set of all samples of input genes and $Y_j^i$ is the set of all samples of output gene $i$. Taking this learning sample as input, the objective of a GRN inference algorithm is to predict the regulatory links among genes such that it works by first ranking all possible regulatory links from the most significant to the least significant links. Recovery of a network is then achieved by pruning the ranked list of links using a threshold.

In this paper, our focus is mainly on the first one. As such, inference algorithm is introduced here as a process that uses $LS$ to allocate weights to candidate regulatory links from any gene to any other gene, such that edges corresponding to real interactions in the regulatory network would be given higher weights. Each sub-problem which is determined by $LS^i$, is a regression problem which tries to find a function of $f_i$ to minimize the error in (3):

$$\Sigma_{j=1}^{N} = (Y_j^i - f_i(Y_j^{i*}))^2 \tag{3}$$

In random forest, regression trees [24] solve the above problem. The main idea is to split the learning sample iteratively with binary tests in accordance with one input variable ($Y^{i*}$) and strive to deduct the output variable variance ($Y^i$) in the resulting subsets of samples. Candidates are split for variables by comparing with the threshold which is defined as long as the tree grows with the values of input variables. In the method, each tree is constructed on a bootstrap learning sample from the original one, and at each test node, before defining the best split, $K$ attributes are chosen randomly from all attributes which become candidates.

One of the key strengths of the Random forests method is its ability to calculate a variables importance from a tree which allows to rank the input features based on their pertinence for predicting the output [23]. In the Random Forest technique several ways to measure the importance of variables have been recommended. Here, we adopt a measure such that in every test node $Z$, we can calculate the whole reduction of the variance of the output variable because of the split [25]

$$R(Z) = |S|\sigma^2(S) - |S_t|\sigma^2(S_t) - |S_f|\sigma^2(S_f) \tag{4}$$

where $S$ denotes the set of samples which reach node $Z$, $S_t$ is its subset for which the test is true, $S_f$ is the subset for which the test is false, $\sigma^2$ $(S)$ is the variance of the output variable in a subset, and $|S|$ denotes the cardinality of a set of samples. For an individual tree, the total importance of one variable is calculated by adding resulting values of nodes in the entire tree where this variable is used to split. Those attributes that are never chosen, receive a value of 0 for their importance, and those attributes that are chosen near the root node generally get high scores. Measures of attribute importance can be extended to ensembles, simply by averaging importance scores over all trees in the ensemble.

The computational complexity of the Random Forests is $O(TKN\log(N))$, where $N$ is the size of the learning sample, $K$ is the number of attributes and $T$ is the number of trees. Therefore, our method has a time complexity of $O(pTKN\log(N))$ as it needs to recover trees in the forest for every $p$ genes. Thus, the computational complexity is log-linear with reference to the number of measurements. In the worst case scenario, it is quadratic in reference to the number of genes since $K = p - 1$. In the next section we will describe how the approach can be parallelized since the $p$ problems, and the generation of $T$ trees, in Random Forest are executed independently from each other.

The final results are directed graphs (i.e. networks) each with $p$ nodes and each node indicates a gene. In each graph, a directed edge from one gene $i$ to another gene $j$ represents that gene $i$ regulates, i.e. represses or activates, the expression of gene $j$. The objective of inferring the GRN is to find a graph only by analysis of the genes expression in diverse situations. By taking into account the dynamic and combinatorial regulatory relations among genes, the expression levels of individual genes can be predicted. Since the overall procedure of the inference tends to be time-consuming, sometimes taking several days even for datasets of moderate sizes, our goal is to implement parallel versions of GRN inference using CPU cluster and GPU.

## 2.2 CUDA Programming Model

The general architecture of NVIDIA GPUs with the support of CUDA is illustrated in Figure 1. The GPU has a number of CUDA cores known as shader processors (SP). Each SP has an immense number of registers and a private local memory (LM). Eight SPs together form a streaming multiprocessor (SM). Each SM also contains a particular memory region that is shared among the SPs within the same SM. By combining a number of SMs the GPU is constructed. GPUs also have some additional memories, for instance the global device memory which is accessible from all SPs. The GPU used for the development of our approach and experimental evaluation is the NVidia Quadra 600.

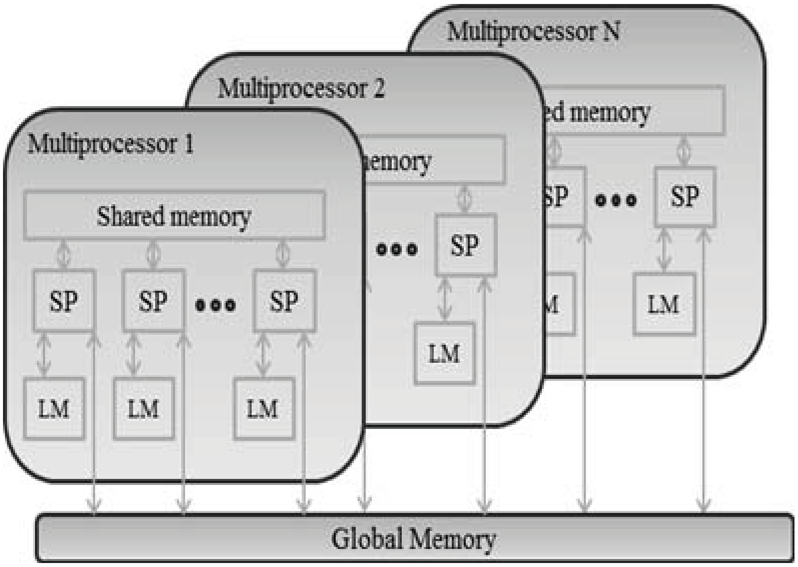

**Fig. 1.** The GPU architecture assumed by CUDA

The important features of utilized GPU are described in Table 1. For computation in GPU, all data need to be transferred to the GPU memory from the host memory. Therefore the bottleneck of the system is the latency between the CPU and the GPU.

**Table 1.** The main characteristics for the NVIDIA Quadro 600 graphics card

| GPU Property | Values |
|---|---|
| CUDA cores | 98 |
| Compute capability | 2.1 |
| GPU / Memory clock rate | 1280 Mhz / 800 Mhz |
| Total amount of memory | 1024 MB |
| Memory interface | 128-bit DDR3, 25.6 GB/s |

## 2.3 CUDAGRN Inference Algorithm

All CUDA programs are separated into two sections: (1) sequential codes which are executed on the host CPU and (2) CUDA functions, or kernels, which are launched from the host and executed on the GPU. Before launching a kernel, required data must be transferred to the device memory from the host memory. Since there are several different memories with different sizes accessible by GPU, data transferred into GPU need to be managed, which also can be a bottleneck. In our algorithm, data are arranged based on usage frequency. If data are regularly used, they are moved to the fastest accessible memory, i.e. shared memory. Otherwise, they are stored in the global memory. By placing data in the GPU memory, in a similar way as calling a regular C function, the CUDA kernel is launched. During the execution of a kernel, several CUDA threads are generated and each thread executes an instance of it. Threads are arranged systematically into blocks, and blocks are arranged into grids.

In this paper, we address the problem of parallel constructing Gene Regulatory Networks from gene expression data using the computational power of the GPU. To parallelize the described method in the CUDA environment, some algorithm sections are sent to GPU and executed by GPU threads. The proposed algorithm is highly parallelizable, since all of the $p$ problems of feature selection, solvable by Random Forest, are independent of each other. In addition, different trees in a forest grow independently. Thus, to implement the program in CUDA, forests construction of feature selection problems is achieved in GPU. Because of the memory constraints of device, our approach computes only one problem at a time and for each problem, all of the trees in the forest grow in parallel. As such, it needs a loop of $p$ iterations to accomplish the calculation and come up with the final network. Figure 2 depicts the overall procedure.

As the figure illustrates, we divide the network recovery problem into $p$ isolated sub-problems and iterate $p$ times, where the $p$ is the number of genes, to solve the overall problem. In each loop, constructions of all $T$ trees are parallelized. Furthermore, there is not any straightforward way to recover each single tree in parallel, so we exploit one thread of CUDA to construct a complete tree in the forest. Therefore, our algorithm performs better with a larger number of trees.

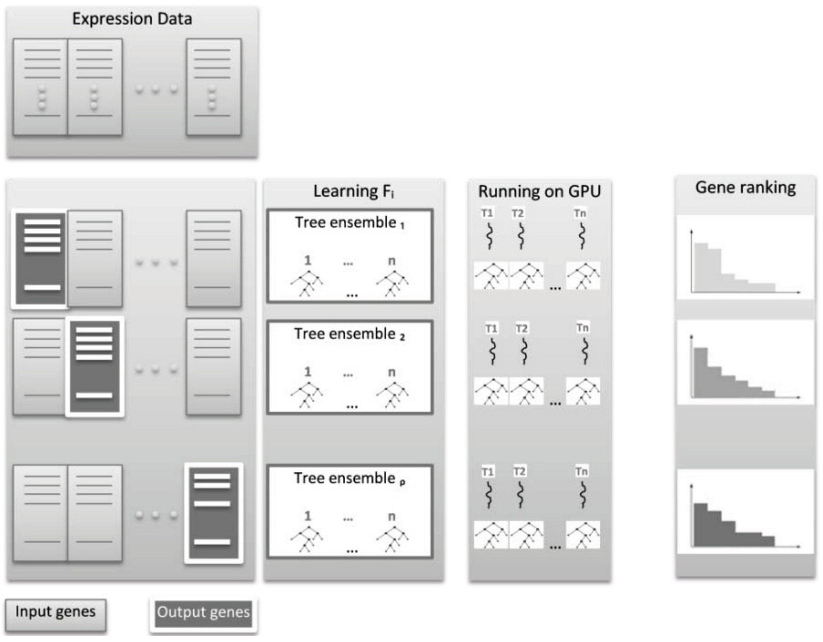

**Fig. 2.** Parallel procedure for execution of Random Forest algorithm for GRN inference on GPU

Several algorithms for decision tree are developed by recursion. However, using a recursion is not possible for algorithms implemented in CUDA since kernels running on graphic device do not support recursion. Hence, it was necessary to design an algorithm which generates trees iteratively. Algorithm 1 shows the pseudo-code of GENIE3 and algorithm 2 describes the steps in our parallelization of the random forest algorithm for GRN inference.

We have described the details of how trees are built during the training phase. The rest of our approach is similar to the sequential implementation. That is, each tree in the forest is sequentially built by using one thread per tree during the training phase. If $N$ threads are run, then $N$ trees are generated in parallel. Thus, our system works best for an immense number of trees. At each level in a tree, the best attribute to use in order to split a node is picked from a pool of $K$ attributes that are randomly chosen. While all trees are built, they are sent to the host memory for use during the phase of computing variable importance.

**Algorithm 1.** GENIE3 [9] Pseudo Code

```
1: procedure GRN-Inference Sequentially
2:     Data ← DataReader()
3:     For each sub problem ($i$ = 1 to $p$):
4:         LearningSamplesGenerator(Data)
5:         FeatureSelectionApproach()
6:         Result ← CalculateLevelOfConfidence()
7:         Normalization(Result)
8:     End For
9:     RankResult(Result)
10: end procedure
```

**Algorithm 2.** CUDAGRN Pseudo Code

```
1: procedure GRN-Inference In Parallel Using GPU
2:     HostMemory ← DataReader()
3:     For each sub problem ($i$ = 1 to $p$):
4:         LearningSamplesGenerator()
5:         DeviceMemory ← $CUDAMemCpy$(HostMemory)
6:         KernelLaunch()
7:         For each Tree ($j$=0 to NumberOfTreesInForest)
8:             OpenNodeInStack ← FirstNodeOfATreeGeneration()
9:             While(OpenNodeInStack)
10:                Node ← OpenNodeInStack[head]
11:                If(StopSplitFunction(Node))
12:                    Leaf(Node)
13:                Else
14:                    FindSplit(Node)
15:                    Split(Node)
16:            End While
17:            Trees ← SaveTree()
18:        End For
19:        HostMemory ← $CUDAMemCpy$(Trees)
20:        $GPUMemoryCleanUp$()
21:        Result ← VariableImportanceCalculator($Trees$)
22:        Normalization(Result)
23:    End For
24:    RankResult(Result)
25: end procedure
```

## 2.4 Parallel GRN Inference on CPU

Message Passing Interface (MPI) is a portable and standardized message-passing system designed to run on a diverse range of parallel computers [26]. Furthermore, OpenMP is an API which supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems [27]. In this section the CPU parallelization of GENIE3, using MPI and OpenMP are presented. Assuming there is enough memory, there are two ways to parallelize GENIE3. In the first way, the algorithm can be parallelized at the tree construction level. For each sub-problem, each CPU thread corresponds to constructing a tree since trees grow independently. Thus, for each sub-problem, a loop with $W$ interactions is executed, where $W = T/thr$, $T$ is the number of trees and thr is the number of active threads. Overall $W * p$ iterations are needed to solve the whole problem. In the second way of parallelizing, each sub-problem must be dealt with in one CPU thread. Therefore, in order to find an answer for a sub-problem, a loop with $T$ iterations is required, and $T * V$, where $V = p/thr$, iterations are required to find a result for all sub-problems.

Since there are overhead costs each time we start OpenMP and MPI, and this can slow down the method, we chose the second way of CPU parallelizing. As such, each CPU thread corresponds to one sub-problem and threads run the problems independently. Eventually, at the end of execution, the main thread accumulates the intermediate results which have been produced by all the threads and provides the final result.

## 3 Result

In this section, we compare the execution times of the proposed CUDAGRN (described in details in the Methods section) with its sequential and parallel CPU counterparts. The platform for our development was Microsoft Windows 7 along with CUDA version 2.3. Our software also used Core i7 Intel CPU, RAM DDR3 of 8GB as the hardware platform. The GPU was a Quadra 600 NVIDIA with memory of 1GB. Note that we have used a low end GPU versus a high end CPU, which suggests that the observed speedups achieved by CUDAGRN were mainly through the parallelization. Both real and synthetic datasets have been used in our experiments. The real dataset, with 130 experiments and over 6000 genes, was downloaded from http://rana.lbl.gov/EisenData.htm. To further test the scalability of CUDAGRN, we have additionally generated simulated datasets with various parameters, e.g. the numbers of experiments, genes, and trees in the Random forests. Simulated datasets were produced by the software of GeneNetWeaver [28].

Three different versions of the GENIE3 algorithm are implemented and compared in our evaluation, i.e. the sequential C++ program, the CPU-parallelization (by MPI and OpenMP), and the GPU-based version (CUDAGRN). In our experimental evaluation, we studied how the execution time would be influenced by varying different parameters, which include the number of trees to generate ($T$), the number of genes ($p$) and sample size. Based on empirical experiments done in [22,23], we configured the number of attributes to sample in each split ($K$) to its optimal value of $K = \sqrt{p-1}$. However, it is beyond the scope of this paper to prove which configuration of the parameters has the greatest impact on the performance.

Since there are three parameters to vary in the algorithm, we alter one and keep the other two constant. Measurements are collected by running the algorithms on synthetic datasets with the number of trees in a forest from 100 to 10000 and the number of genes from 10 to 500 and sample size (i.e. number of conditions or time points in microarray) from 100 to 5000. Table 2, shows the runtime improvement of CUDAGRN in comparison with other implementations of the GENIE3 algorithm running on different synthetic datasets. As shown in the table, CUDA implementation of the GENIE3 has a faster runtime than the other two implantations when the amount of computation increases. Moreover,

**Table 2.** Demo result of execution improvement with 1000 trees and 1000 experiments

| Number of Genes | Sequential(sec) | CPU 8-Threads(sec) | GPU(sec) |
|---|---|---|---|
| 10 | 30 | 6 | 2 |
| 20 | 84 | 13 | 6 |
| 50 | 274 | 38 | 18 |
| 100 | 753 | 98 | 51 |
| 200 | 2175 | 285 | 154 |
| 500 | 8361 | 1093 | 586 |

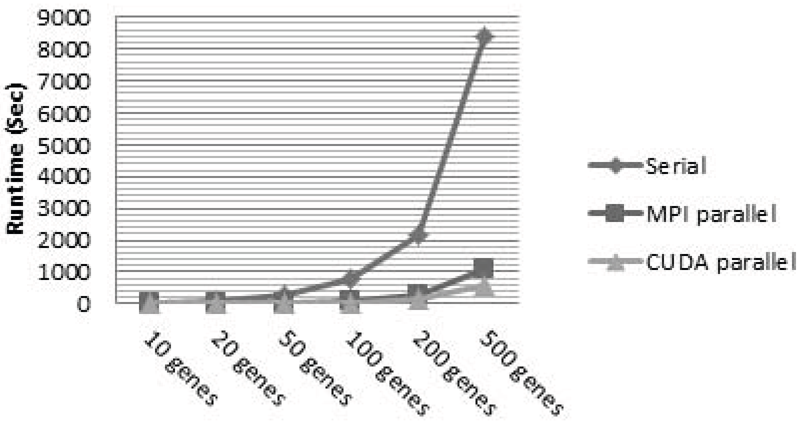

**Fig. 3.** Diagram of runtimes while the number of genes is varying

line charts of the three tables are shown in Figure 3. The diagram is depicted for a varying number of genes, while the numbers of experiments and trees are both equal to 1000. From the figure, we can see that CUDAGRN achieves the best performance in term of runtime.

In addition, CUDAGRN was faster than the other two implementations when executed on the aforementioned real dataset. CUDAGRN obtained the result of GRN inference approximately 15 times faster than the sequential program (Figure 4). In this section we conducted experiments on real and simulated datasets to show that CUDAGRN is able to infer gene regulatory networks from large and high-dimensional datasets faster than other implementations while maintaining nearly all the accuracy of inference.

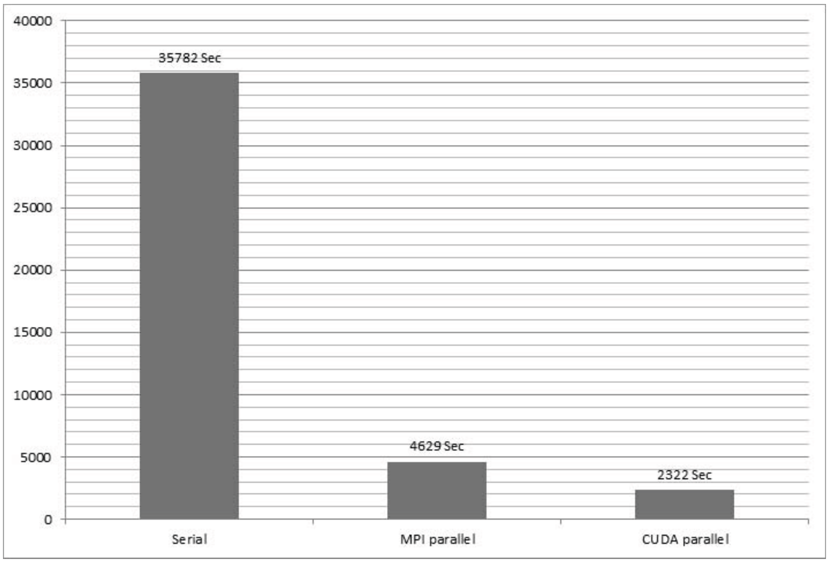

**Fig. 4.** Computational times of real dataset (6331 Genes, 131 Experiments, 1000 Trees)

## 4 Conclusion

We presented a novel parallel model of the GENIE3 algorithm, CUDAGRN, developed by exploiting the Compute Unified Device Architecture (CUDA). Comparing the performances of CUDAGRN, Sequential GRN inference and CPU multi-threaded implementations, we observed that CUDAGRN can outperform both the other competitors in term of computational time provided parallelization did not reduce the accuracy of inference.

Unlike the sequential approach and CPU parallelization, the CUDAGRN algorithm as proposed in this paper is executable on the GPUs. On ordinary PCs, the number of processing units (cores) in the CPUs is significantly less than the number of processing units on GPUs. In our approach, whereas the difference in regression performance, e.g. accuracy, among the different implementations is imperceptible (data not shown), it is clear that CUDAGRN is much more efficient in term of computational speed, particularly with large numbers of experiments and trees to build in the Random forest. Testing on real data shows that CUDAGRN is nearly 15 times faster than Sequential GRN, and about twice faster than multi-threaded CPU implementation.

In future, we will refine our implementation of CUDAGRN. In particular, we plan to append properties to make CUDAGRN more accessible to different kinds of applications and practical conditions. For instance, the present version of CUDAGRN is only able to operate on input attributes that are numeric and it cannot deal with missing values which will be addressed in the new version of our implementation.

**Acknowledgments.** This project has been supported by AcRF Tier 2 grant (ARC9/10, MOE2010-T2-1-056) of Ministry of Education, Singapore.

## References

1. Bolouri, H.: Computational modeling of gene regulatory networks: a primer. Imperial College Press, London (2008)
2. Butte, A.J., Kohane, I.S.: Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In: Pacific Symposium on Biocomputing, vol. 5, pp. 418–429 (2000)
3. Schafer, J., Strimmer, K.: An empirical Bayes approach to inferring large-scale gene association networks. Bioinformatics 21(6), 754–764 (2005)
4. Liang, S., Fuhrman, S., Somogyi, R.: REVEAL, a general reverse engineering algorithm for inference of genetic network architectures. In: Pacific Symposium on Biocomputing, vol. 3(3), pp. 18–29 (1998)
5. Akutsu, T., Miyano, S., Kuhara, S.: Identification of genetic networks from a small number of gene expression patterns under the Boolean network model. In: Pacific Symposium on Biocomputing, vol. 4, pp. 17–28 (1999)
6. Friedman, N., Linial, M., Nachman, I., Pe'er, D.: Using Bayesian networks to analyze expression data. Journal of Computational Biology 7(3-4), 601–620 (2000)
7. Chen, H., Maduranga, D.A.K., Mundra, P.A., Zheng, J.: Integrating epigenetic prior in dynamic bayesian network for gene regulatory network inference. In: 2013 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), pp. 76–82 (2013)
8. Vohradsky, J.: Neural model of the genetic network. Journal of Biological Chemistry 276(39), 36168–36173 (2001)
9. Irrthum, A., Wehenkel, L., Geurts, P.: Inferring regulatory networks from expression data using tree-based methods. PloS One 5(9), e12776 (2010)
10. Li, X., Xu, R.: High-dimensional data analysis in cancer research. Springer (2009)
11. Maduranga, D.A.K., Zheng, J., Mundra, P.A., Rajapakse, J.C.: Inferring gene regulatory networks from time-series expressions using random forests ensemble. In: Ngom, A., Formenti, E., Hao, J.-K., Zhao, X.-M., van Laarhoven, T. (eds.) PRIB 2013. LNCS, vol. 7986, pp. 13–22. Springer, Heidelberg (2013)
12. The DREAM4 In Silico network challenge (2010), `http://wiki.c2b2.columbia.edu/dream`
13. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9(suppl. 2), S10 (2008)
14. Ferreira, J.F., Lobo, J., Dias, J.: Bayesian real-time perception algorithms on GPU. Journal of Real-Time Image Processing 6(3), 171–186 (2011)

15. Suchard, M.A., Wang, Q., Chan, C., Frelinger, J., Cron, A., West, M.: Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. Journal of Computational and Graphical Statistics 19(2), 419–438 (2010)
16. Park, I.K., Singhal, N., Lee, M.H., Cho, S., Kim, C.W.: Design and performance evaluation of image processing algorithms on GPUs. IEEE Transactions on Parallel and Distributed Systems 22(1), 91–104 (2011)
17. Shi, H., Schmidt, B., Liu, W., Müller-Wittig, W.: Parallel mutual information estimation for inferring gene regulatory networks on GPUs. BMC Research Notes 4(1), 189 (2011)
18. Colmenares, J., Ortiz, J., Rocchia, W.: GPU linear and non-linear Poisson Boltzmann solver module for DelPhi. Bioinformatics, btt699 (2013)
19. Li, L., Li, C., Sarkar, S., Zhang, J., Witham, S., Zhang, Z., Alexov, E.: DelPhi: a comprehensive suite for DelPhi software and associated resources. BMC Biophysics 5(1), 9 (2012)
20. Saeys, Y., Inza, I., Larrañaga, P.: A review of feature selection techniques in bioinformatics. Bioinformatics 23(19), 2507–2517 (2007)
21. Geurts, P., Irrthum, A., Wehenkel, L.: Supervised learning with decision tree-based methods in computational and systems biology. Molecular Biosystems 5(12), 1593–1605 (2009)
22. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
23. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC Press (1984)
24. Geurts, P., Ernst, D., Wehenkel, L.: Extremely randomized trees. Machine Learning 63(1), 3–42 (2006)
25. Strobl, C., Boulesteix, A.L., Zeileis, A., Hothorn, T.: Bias in random forest variable importance measures: Illustrations, sources and a solution. BMC Bioinformatics 8(1), 25 (2007)
26. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, vol. 1. MIT Press (1999)
27. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: portable shared memory parallel programming, vol. 10. MIT Press (2008)
28. Schaffter, T., Marbach, D., Floreano, D.: GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods. Bioinformatics 27(16), 2263–2270 (2011)