A GPU-based Implementation of an Enhanced GEP Algorithm

Shuai Shao, Xiyang Liu
Software Engineering Institute
Xidian University
Xi'an, Shaanxi 710071, China
{ShaoShuaiXD, XiyangLiu}
@gmail.com

Mingyuan Zhou, Jiguo Zhan
Software Engineering Institute
Xidian University
Xi'an, Shaanxi 710071, China
{myzhou.xd, ZhanjiGuoXD}
@gmail.com

Xin Liu, Yanli Chu, Hao Chen
Software Engineering Institute
Xidian University
Xi'an, Shaanxi 710071, China
{xinliuxd, yanli.chu,
chenhao9255}@gmail.com

ABSTRACT
Gene expression programming (GEP) is a functional genotype/phenotype system. The separation scheme increases the efficiency and reliability of GEP. However, the computational cost increases considerably with the expansion of the scale of problems. In this paper, we introduce a GPU-accelerated hybrid variant of GEP named pGEP (parallel GEP). In order to find the optimal constant coefficients locally on the fixed function structure, the Method of Least Square (MLS) has been embedded into the GEP evolutionary process. We tested pGEP using a broad problem set with a varying number of instances. In the performance experiment, the GPU-based GEP, when compared with the traditional GEP version, increased speeds by approximately 250 times. We compared pGEP with other well-known constant creation methods in terms of accuracy, demonstrating MLS performs at several orders of magnitude higher in terms of both the best residuals and average residuals.

Categories and Subject Descriptors
I.2.6 [**Artificial Intelligence**]: Learning – *Knowledge acquisition*; D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*

General Terms
Algorithms, Performance, Experimentation

Keywords
Symbolic Regression, GEP, Constant Creation, GPU, CUDA, MLS

1. INTRODUCTION
Gene expression programming (GEP) is a popular evolutionary algorithm launched in 2001 [1]. It has been widely applied in many fields, and in the era of data mining, appears to be a promising research area for symbolic regression [2]. Without any prior knowledge about physics, kinematics, or geometry, Michael Schmidt and Hod Lipson [3] [4] [5] [6] [7] used GP, the ancestor of GEP, to discover Hamiltonians, Lagrangians, and other laws of geometric and momentum conservation. GEP is differentiated from its predecessors, genetic algorithms (GAs) [8] and genetic programming (GP) [2] in individual expressions. The individuals are represented by symbolic strings of fixed length (chromosomes) in GAs, and non-linear entities of different sizes and shapes (parse trees) in GP. The GEP individuals are encoded as fixed length linear chromosomes (genotype) which are then translated into different sized and shaped non-linear entities, Expression Trees (ETs) (phenotype). Any genetic operation upon chromosomes always results in syntactically correct ETs. In this genotype/phenotype system, the search space is separated from the solution space, enabling unrestricted searches and effective solutions. Unfortunately the fitness evaluation process of GEP is computationally expensive. In order to effectively apply the GEP algorithm for large scale data applications, the computational challenge has to be resolved.

Recent advances in Graphics Processing Units (GPUs) opened a new era of GPU computing [9]. Traditionally, using GPU for non-graphic applications was considered to be very difficult. However, the NVIDIA CUDA programming model made the development of non-graphic applications on GPU easier [10] [11]. In CUDA model, the GPU becomes a dedicated coprocessor to the host CPU, which uses the principle of Single Program Multiple Data (SPMD) where multiple threads based on the same code can run simultaneously.

Motivated by observing the computational requirement of evaluation process, in this paper we present a method, pGEP (parallel GEP), to optimize the GEP with GPU, accelerating the search in the expression space. In addition to using GEP to improve performance, we embedded MLS [12] (Method of Least Square) as a local optimizer into GEP. When the structure of a function expression is set, the coefficients best satisfying the experiment data can be obtained efficiently. The GEP algorithm is responsible for evolving individual structures and the MLS for obtaining the most satisfying coefficients based on the fixed structure. Results show that pGEP is significantly more accurate and efficient.

The rest of this paper is organized as follows. Section 2 provides a brief overview of GEP, GPU architecture, CUDA programming model and some existing constant creation methods. Section 3 describes the design of CUDA-based algorithm pGEP, which embraces MLS for constant creation. Section 4 provides experiment results and evaluations. Finally, Section 5 concludes this paper and indicates future work directions.

2. BACKGROUND MATERIAL
2.1 An Overview of GEP
The GEP system is a genotype/phenotype system, with linear chromosomes functioning as genotype, and ETs as phenotype. In GEP, individuals are encoded as linear strings of fixed length (chromosomes) which are afterwards expressed as nonlinear

GECCO'12, July 7–11, 2012, Philadelphia, Pennsylvania, USA.

entities of differing sized and shaped ETs. Chromosomes are linear, compact and easy to manipulate genetically. Selection processes act on chromosomes and, according to fitness, they can be selected to reproduce with modification. GEP is simple in encoding, and any modification made in the chromosome results in syntactically correct ETs.

Each GEP chromosome is composed of genes of equal length, and each gene is further composed of a head and a tail. The head contains symbols chosen randomly from function set F and terminal set T, whereas the tail contains only terminals. For algebraic sub-ETs in general symbolic regression problems, an addition or multiplication operator is often employed to link the sub-ETs into a final, multi-subunit ET.

The GEP algorithm begins with the random generation of the chromosomes of the initial population. The chromosomes are expressed in ETs and the fitness of each individual is assessed. The individuals are then selected according to fitness to reproduce with genetic operators. The new individuals are, in their turn, subjected to the same evolution process as mentioned above. This process is repeated for a specified number of generations or until a solution has been found.

2.2 GPU and CUDA Architecture

GPUs are dedicated hardware for manipulating computer graphics. In 2006, one of the major GPU vendors, NVIDIA, announced their new general-purpose parallel programming model, Compute Unified Device Architecture (CUDA) [10] [11]. CUDA extends the C programming language for general-purpose application development. It has been used for speeding up a large number of applications.

The NVIDIA Tesla C2050 GPU has 14 Streaming Multiprocessors (SMs), and each SM has 32 Scalar Processors (SPs), giving a total of 448 processor cores. The SMs are designed as Single-Instruction Multiple-Data (SIMD) architecture: At any given clock cycle, each SP executes the same instruction, but operates on different data. Each SP can support 32-bit single-precision floating-point arithmetic as well as 32-bit integer arithmetic.

Each SM has several different forms of on-chip memories, e.g. register, shared memory, constant cache, texture cache and so on. Shared memory is almost as fast as register. Constant cache and texture cache are both read-only memories shared by all SPs, but with very limited size. Off-chip memories, such as local memory and global memory, have relatively long access latency, usually 400 to 600 clock cycles [11]. The properties of the different types of memories have been summarized in [11] [16]. In general, the limited shared memory should be utilized carefully to amortize the global memory latency cost. Shared memory is divided into equally-sized banks, which can be accessed simultaneously. Any memory request falling in distinct memory banks can be serviced simultaneously. If two addresses of a memory request fall into the same bank, it is referred to as bank conflict, and the access has to be serialized [11]. For Tesla C2050, each SM has 32768 32-bit registers and 48 Kilobytes of shared memory. For details on the CUDA programming model refer to [15].

2.3 Constant Creation

Research and discussion on constant creation issues have continued for some time in GP research circles. In GP, both the entire form of the model and its numerical parameters can be found simultaneously, but GP has difficulty in discovering satisfactory constants. As John Koza mentioned [16], the area of constant creation research requires more investigation. Constant creation has become one of the major obstacles that obstruct achieving greater efficiency for complex GP applications [13] [14].

Several constant creation approaches in GEP have been proposed. Ferreira introduced two approaches [17]. One approach does not include any constants in the terminal set, generating constants spontaneously in GEP evolutionary process, whereas the other appends a random constant domain Dc at the end of the chromosome, explicitly manipulating the numerical constants. Experiments demonstrate that the first algorithm is more efficient in terms of both accuracy of the evolved models and computational cost of the search process. Li et al. proposed five GEP constant creation approaches [18], based on two basic approaches similar to creep and random mutation but in a greedy fashion. Although these constant enhancement strategies perform better for average of fitness of the best solutions than the original methods, all of them require extra computation. Zhang et al. introduced a new approach to constant generation using Differential Evolution (DE) [19], an example of a global, derivative-free optimization algorithm that combines a multipoint-based search with a generate-and-test paradigm. Their experimental results show that DE-GEP is more likely to find an optimal constant than the best method in [20]. However, the extra computation requirements of DE are considerable. In this paper, we improve the process of constant creation in GEP.

3. ALGORITHM DESIGN

3.1 CUDA-based Algorithm

GEP inherits the manipulative and expressive advantages of GAs and GP. However, traditional GEP is inefficient, with long run time especially in large evolutionary generations, population size and training data. In this section, we propose an improved version of GEP, the pGEP. We accelerate GEP with GPU. The basic flowchart of pGEP is shown in Figure 1.

There are three evaluation phases. In Evalution-1 phase, CUDA environment is initialized and CPU transmits training data to global memory on GPU. The evolutionary process begins with the random generation of the chromosomes in the initial population. Then the chromosomes are expressed as ETs in the form of post-order sequence. Finally, these linear structures are transferred into the constant memory in GPU.

In Evaluation-2 phase, the training data on the global memory is mapped to each thread, which calculates values and absolute errors according to the post-order sequence on constant memory. Then the total absolute errors are obtained by reduction algorithm [21]. Finally, the calculated fitness of each individual is transmitted to host memory on CPU.

In Evalution-3 phase, the individuals are selected according to fitness to reproduce with genetic operators. In this phase, individuals with better fitness can be replicated into the next generation.

The process is cycled for a predetermined number of generations. Finally, some of the best individuals are stored.

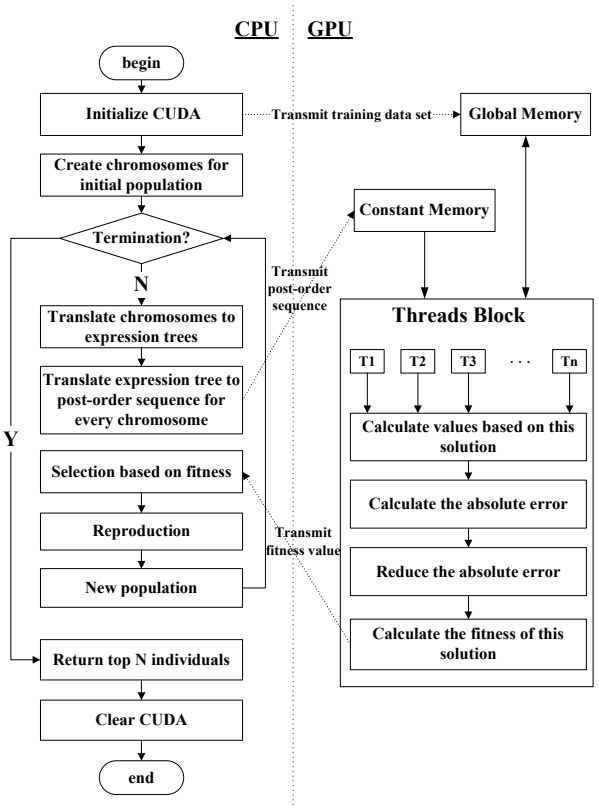

Figure 1. The basic flowchart of pGEP.

3.2 Evaluation Process

As mentioned in the introduction, the fitness calculation is one of major bottlenecks in GEP, because it involves calculating all the results with all the instances in the training set. Our goal is to parallelize the calculation of the chromosome's fitness value. This CUDA-based fitness calculation involves two major stages. The first stage obtains the computation results after the training data is substituted into each chromosome's phenotype. The second stage calculates a summation of the results that were previously obtained in the first stage for each chromosome.

For the first stage, we first transfer the phenotypes from the host (CPU) to the device (GPU) before we calculate the fitness value. In the host, GEP chromosomes are composed of one or more genes, and obviously the encoded individuals result in different degrees of complexity. The complex individual is encoded in several genes, and the phenotype is multi-subunit expression tree, in which the different sub-ETs are linked together by a particular function. So each phenotype's structure is a type of nonlinear structure. A nonlinear structure can be easily achieved in CPU, but it must be converted to linear structure when it is achieved in GPU. In this paper, we convert this multi-subunit expression tree into binary tree, then we use post-order recursive traversal algorithm to traverse this binary tree. The result is a linear suffix expression, easily stored and read by threads in GPU. Figure 2 gives an example.

Because the suffix expression can only be read in GPU and never be changed in the future, we put this linear structure in the constant memory space which resides in GPU device memory and is cached in the constant cache. Here it promotes the threads' reading speed and data throughput. The experimental data has been transferred to the global memory before the evolution process. In Figure 3, each thread is in charge of computing a case of experimental data, e.g. sample data (X_n,Y_n,Z_n) will be substituted into the suffix expression to obtain the function result. X_n is a data set including $\{x_0, x_1, \ldots, x_n\}$, of which each data x_i is managed by a thread in the same block. The meanings of Y_n and Z_n are in a similar way. An instruction that accesses addressable memory might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps [11]. When a warp executes an instruction that accesses global memory, it coalesces the memory accessions of the threads within the warp. We split a case of experimental data (X_n,Y_n,Z_n) into three consecutive sequence of storage areas, which causes the successive threads to access the adjacent memory locations and upgrades the performance in GPU.

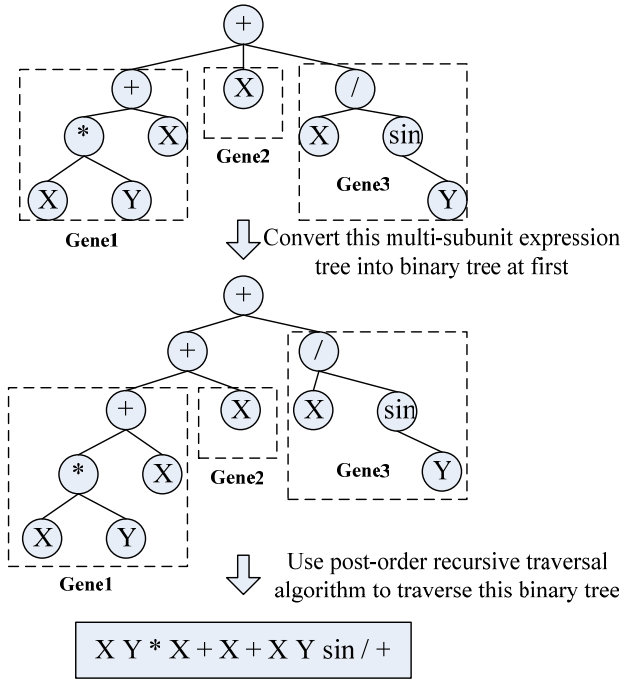

Figure 2. The changing representation of a multi-gene chromosome.

The experimental data is stored in the first three global memory areas in Figure 3. At the beginning of pGEP, two different pointers are defined and initialized in GPU. The pointer P1 indicates the beginning location from which the computation results are saved in the global memory. Pointer P2 indicates the first symbol of the suffix expression which is transferred into the constant memory. The computation rule is described as follows: if the symbol that P2 points at is one of the terminals (e.g. X or Y), each thread moves its corresponding experimental data to the storage area that P1 points to in the global memory, then P1 moves to its following memory area. However, if the symbol is a function (e.g. '+' or 'sin'), each thread will use the data stored in previous area P1 indicated and chooses the corresponding function operator to gain the calculation results. On completion in either case, P2 is moved to point at the next symbol. The process is repeated until P2 points to the end of the suffix expression. We present the circumstance when P2 finishes traversing the suffix

expression in Figure 3. The final function results are calculated in the $Z_0^{'}$, $Z_1^{'}$ ... $Z_n^{'}$ by each thread.

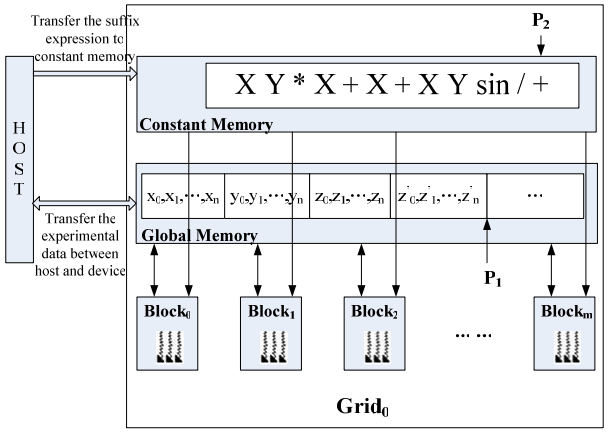

Figure 3. The data storage and each thread's task allocation in the GPU.

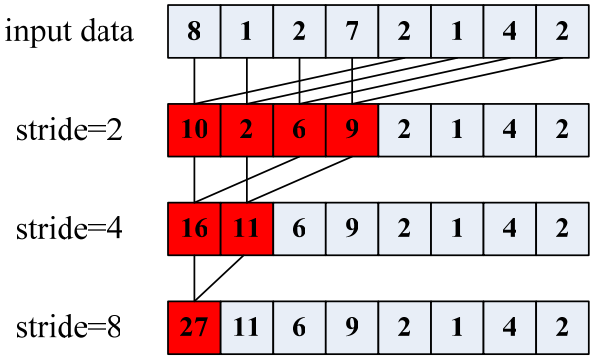

Figure 4. Single block parallel reduction.

Mathematically, the fitness f_i of an individual program i is expressed by equation (1), where M is the range of selection. $Z^{'}_{(i,j)}$ is the value calculated by the individual chromosome i for fitness case j, and Z_j is the target value for the fitness case j. Then we reduce this relative error in the second stage to get the fitness value of every chromosome.

$$f_i = \sum_{j=1}^{n} (M - |\frac{Z^{'}_{(i,j)} - Z_j}{Z_j} * 100|) \quad (1)$$

In the second stage, we sum up the information previously calculated iteratively. This stage performs the CUDA parallel reduction algorithm [21]. Multiple thread blocks are used to process very large arrays and keep all multiprocessors in the GPU engaged. Figure 4 shows an example of the reduction process. Each thread block reduces a portion of the large array. In order to avoid the shared memory bank conflicts, neighboring threads operate on adjacent elements. After each block reduces its corresponding portion of the array to one value, the value is copied back to host memory using a single memory copy operation.

3.3 MLS for GEP Constant Creation

In pGEP, we embed the Method of Least Square (MLS) [12] into the GEP evolutionary process. GEP uses genetic operations to create an evolution of the structure of function expression, while MLS finds the optimal constant coefficients locally on the fixed function structure.

The general purpose of symbolic regression is to find an optimal symbol expression that best fits the target sample set, $S = (x_i, y_i)$, i=0,1,…m, which contains one or more independent variables and dependent variables. Since we can easily extend one input variable to n input variables by adding n variables in the terminal set [12], we assume that the formula would have only one independent variable (x) and one dependent variable (y) in the following equations. Given m+1 data set $\{(x_0,y_0),...,(x_m,y_m)\}$, we define the mean error of the function expression by equation (2).

$$MSE(f,S) = \frac{1}{m+1}\sum_{i=0}^{m}\left(f(x_i) - y_i\right)^2 \quad (2)$$

In this paper, we adopt a multi-gene structure (c_0, c_1,…, c_n) to represent individuals in GEP. Genes are connected by an addition operator and treated as mutually independent function terms. Combining with corresponding coefficients, genes form the final symbolic expression $f(x)$ (see equation (3)). In MLS, genes care about only function structures without any consideration of constant coefficients, such as forms like $x^2+\sin(x)$, $x-\log x$, $\sin(\cos x)$. In short, the function structures evolve independently in pGEP, while their corresponding optimum coefficients are calculated by MLS.

$$f(x) = \sum_{k=0}^{n} a_k c_k(x) \quad (3)$$

Substituting equation (3) into equation (2), we find equation (4).

$$MSE(f,S) = \frac{1}{m+1}\sum_{i=0}^{m}\left(\sum_{k=0}^{n} a_k c_{k,i} - y_i\right)^2 \quad (4)$$

The goal of MLS is to find values of vector a that minimize the error in equation (5).

$$I = \sum_{i=0}^{m}\left(\sum_{k=0}^{n} a_k c_{k,i} - y_i\right)^2 \quad (5)$$

Differentiating $I(a_0,a_1,...,a_n)$ and setting $\partial I/\partial a = 0$ yields in equation (6).

$$\frac{\partial I}{\partial a_j} = 2\sum_{i=0}^{m}\left(\sum_{k=0}^{n} a_k c_{k,i} - y_i\right)c_{j,i} = 0 \quad j = 0,1,...,n \quad (6)$$

After simplifying equation (6), we get equation (7).

$$\begin{bmatrix} \sum_{i=0}^{m} c_{0,i}c_{0,i} & \sum_{i=0}^{m} c_{1,i}c_{0,i} & \cdots \sum_{i=0}^{m} c_{n,i}c_{0,i} \\ \sum_{i=0}^{m} c_{0,i}c_{1,i} & \sum_{i=0}^{m} c_{1,i}c_{1,i} & \cdots \sum_{i=0}^{m} c_{n,i}c_{1,i} \\ \cdots & & \\ \sum_{i=0}^{m} c_{0,i}c_{n,i} & \sum_{i=0}^{m} c_{1,i}c_{n,i} & \cdots \sum_{i=0}^{m} c_{n,i}c_{n,i} \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ \cdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{m} y_i c_{0,i} \\ \sum_{i=0}^{m} y_i c_{1,i} \\ \cdots \\ \sum_{i=0}^{m} y_i c_{n,i} \end{bmatrix} \quad (7)$$

$$a = A^{-1}B \quad (8)$$

We denote coefficient matrix in equation (7) as A, and constants matrix as B. If a coefficient matrix is a symmetric positive definite matrix, it can lead to only one solution. When A is a symmetric

positive definite matrix, we get vector a as equation (8). If the coefficient matrix is nonsingular, we set the vector $a\ \{a_0, a_1, ...a_n\}$ to {1} by default.

So far, we have used MLS to find the optimal weights a_k (k=0,1,…n) using the fixed function structures to minimize mean square error.

In pGEP, to find a good approximation, GEP and MLS are mutually dependent in the evolutionary process. A good GEP formula structure has the potential to use the power of MLS for finding appropriate constants to best fit the sample set. If GEP fails to find an appropriate formula structure through its evolution, the power of MLS cannot be fully utilized in optimization, even though it can find the best constants given a poorly structured formula. The extent MLS can play a role largely depends on the function structure evolved by GEP. MLS generates the optimal corresponding constants upon the fixed structures in order to improve the individual's fitness. Individuals who have better structures are more likely to survive.

4. EXPERIMENTS AND RESULTS

To test the performance of our implementation pGEP, two groups of experiments have been designed. First we evaluated the speedup of our pGEP compared with the traditional GEP. Secondly, we conducted experiments to verify the effect of the pGEP algorithm on constant creation. In this section, thirty problems are selected as test cases. These problems appeared in the published literature on constant creation issues in GP or GEP.

4.1 Experimental Settings

Parameter settings in the experiments are summarized in Table 1. Note that for simple polynomial problems, F= {+,-,*, /}; for problems with complex structures, F = {+,-,*, /, exp, sin, cos, log, sqrt}. The residual precision value varies for different problems.

Table 1. Experiment parameters for the problems.

Number of runs	100
Number of generations	100
Population size	200
Number of fitness cases	21~50
Function set (F)	{+,-,*,/,cos,sin,sqrt,exp,log}
Terminal set (T)	{x,y,z}
Head length	3~10
Number of genes	1~10
Linking function	{+,*}
Chromosome length	7~210
Mutation rate	0.051
One-point recombination rate	0.2
Two-point recombination rate	0.5
Gene recombination rate	0.1
IS transposition rate	0.1
RIS transposition rate	0.1
Gene transposition rate	0.1

Note that the roulette-wheel with elitism is chosen as the selection method and the fitness function is used by equation (1) in [1].

We conducted all of our experiments on a workstation HP xw8600 with an NVIDIA Tesla C2050 GPU with 448 processing cores. The computer system has an Intel Xeon CPU E5440 2.83GHz and 8G RAM in 64-bit Redhat Linux.

Table 2. Speed-up Ratio.

Record Number	Data Size	Average Time of Traditional GEP(sec)	Average Time of pGEP(sec)	Speed-up Ratio
1	1000	0.025300	0.004990	5.07
2	2000	0.049860	0.005030	9.91
3	3000	0.075410	0.005100	14.79
4	4000	0.099580	0.005110	19.49
5	5000	0.124520	0.005180	24.04
6	6000	0.149900	0.005190	28.88
7	7000	0.173160	0.005270	32.86
8	8000	0.198500	0.005390	36.83
9	9000	0.224890	0.005500	40.89
10	10000	0.248110	0.005560	44.62
11	20000	0.503070	0.006170	81.53
12	40000	1.016660	0.008660	117.40
13	60000	1.479000	0.010340	143.04
14	80000	2.047500	0.012300	166.46
15	100000	2.502000	0.014300	174.97
16	120000	2.972000	0.016620	178.82
17	140000	3.627000	0.018630	194.69
18	160000	3.931000	0.020010	196.45
19	180000	4.579000	0.021720	210.82
20	200000	5.019000	0.023530	213.30
21	400000	11.370840	0.044830	253.64
22	600000	18.852000	0.060530	311.45
23	800000	26.146000	0.082350	317.50
24	1000000	34.687060	0.097460	355.91
25	2000000	68.004000	0.189450	358.95
26	4000000	140.325000	0.381410	367.91
27	6000000	206.845000	0.559200	369.89
28	8000000	281.588000	0.836220	336.74
29	10000000	378.949000	0.927610	408.52
30	12000000	432.017000	1.112480	388.34
31	14000000	498.112000	1.302310	382.48
32	16000000	578.815000	1.570410	368.58

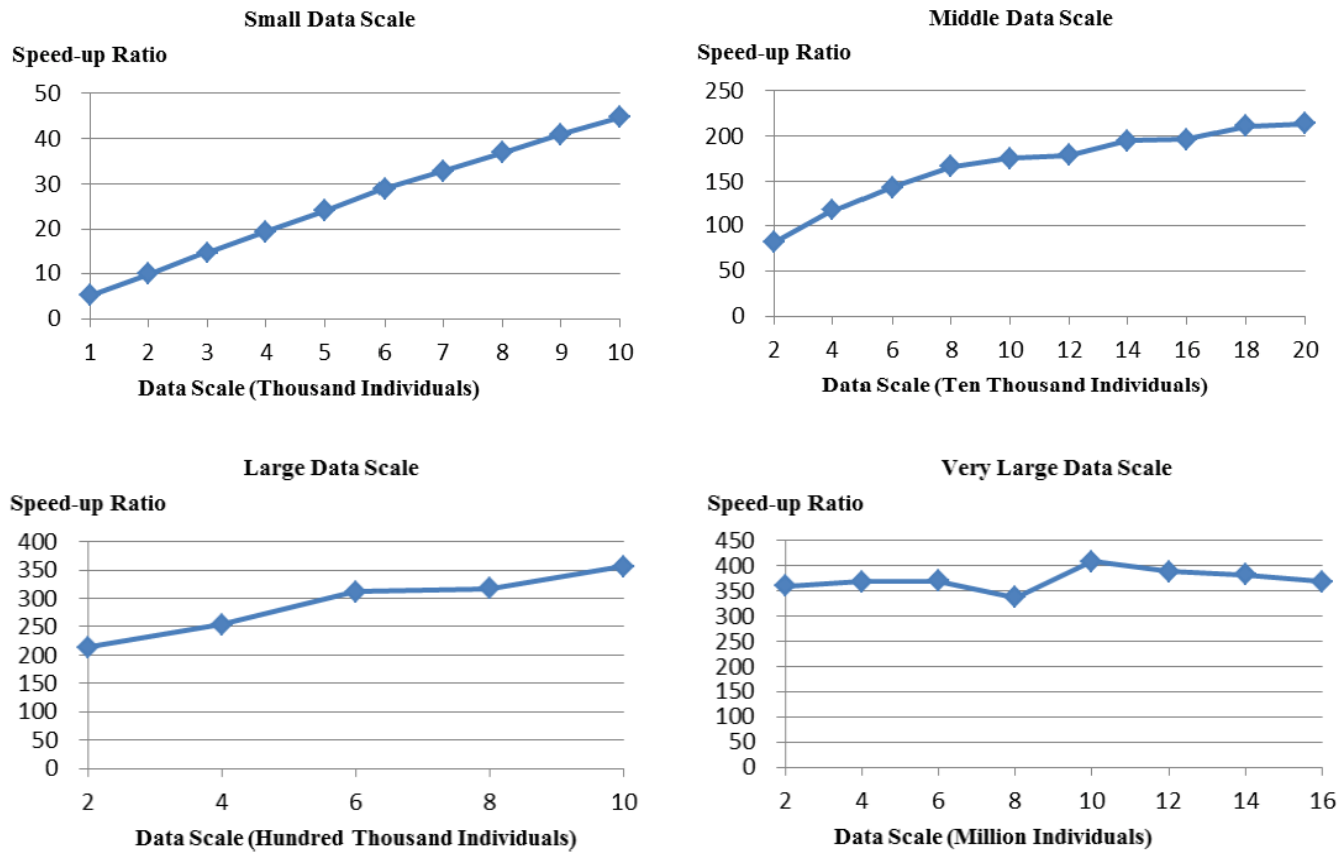

Figure 5. Speed-up ratios of different data scales.

4.2 Experimental Results and Analysis

In this section, we present the results of both groups of experiments. The speed up of the CUDA evaluation process against the traditional serial algorithm is reported in order to determine the advantage of using this parallel architecture. And then the accuracy of pGEP with MLS is compared with other constant creation methods.

4.2.1 Speed up of pGEP

In Table 2 we present the total speedup of the CUDA-based GEP against the serial version for the different scales of a training set. The total time of pGEP consists of data computation time and memory transfer time between CPU and GPU. The problem we tested is $y = 2x$. The training set scale is between 1000 and 16 million. Figure 5 shows the relationship between the training set scale and speed up. With increasing of data scale, the performance of pGEP gradually improves until the scale of the training data exceeds 1 million. We can obtain up to 350X speed up over a problem with 1 million training instances. We can draw the conclusions as follows:

(1) The superior performance of pGEP is an elegant solution to the problem of large scale data size.

(2) When the scale of data size reaches the limits set by the GPU's performance, the superior performance of pGEP will reach a steady state.

4.2.2 Accuracy of pGEP

We tested 29 experimental problems with Success Rate and the Average Time of Success as the metrics. In Table 3, the third column results demonstrate that there are 20 problems with a success rate of 100%, 5 problems with a success rate between 70% and 90%, and 4 problems with less than 40%. Of these, Problem-29 has the lowest success rate at 0%. Compared with the other problems tested, Problem-29 has the most complex structure, and has more coefficients in the nested function structure, where the inner structure $\log(4+2\sin x \sin(8x)) + e^{\cos(3x)}$ is extremely complex. MLS is incapable of calculating the coefficients inside the nested function structures. The statistical results of Average Time of Success show that the search time is very short, 22 problems are completed in less than one second, and 7 problems between one and two seconds. Results imply a linear relationship between Average Time of Success and Number of Genes. In M Schmidt's paper [3], it takes 30 seconds to 10 minutes to search the pendulum laws. While in our experiment, it takes approximately one second of computation.

In Table 4, we abstract Problem-1 and Problem-2 from Table 3 to compare pGEP with DE-GEP [19] and GEP with RM_INTV [18], processes which demonstrated impressive results in constant creation in earlier research papers.

(1) Problem-1 is a simple polynomial with coefficients of real numbers. In Table 3, the statistical results of Problem-1 show that pGEP algorithm has significantly better results in terms of Best residual and Avg. of best residuals. Compared to DE-GEP algorithm, pGEP provides results several orders of magnitude higher in both Best residual and Avg. of best residuals indicators.

Table 3. Experimental results of problems using pGEP.

ID	Problem	Success Rate[1]	Average Time of Success[2](sec)	Number of Genes[3]
1	$x^3 - 0.3x^2 - 0.4x - 0.6$	100%	1.01	4
2	$4.251a^2 + \log(a^2) + 7.243e^a$	85%	0.70	3
3	πr^2	100%	0.274	1
4	$\sin x$	100%	0.152	1
5	$v^2 - 6.04x^2$	100%	0.539	2
6	$a - 0.008v - 6.02x$	100%	0.796	3
7	$9.8\sin x + z$	100%	0.519	2
8	$-9.8\cos x + 0.5y^2$	100%	0.507	2
9	$\sin x + \cos y$	100%	0.503	2
10	$2x + 512y^2$	100%	0.608	2
11	$x^2 + y^3$	100%	0.683	2
12	$(x - 1.12) * \cos y$	100%	0.497	2
13	$0.91 * e^{y/z}$	100%	0.234	1
14	$1.37\omega^2 + 3.29\cos\theta$	100%	0.513	2
15	$2.71\alpha + 0.054\omega - 3.54\sin\theta$	100%	0.742	3
16	$114.28v^2 + 692.32x^2$	100%	0.509	2
17	$xy + x^2 - y^2 + 1$	100%	1.084	4
18	$xy + x^2 - y^2 + 1.234$	100%	1.06	4
19	$xy + x^2 - y^2$	100%	0.8	3
20	$x^2 + xy + y^2$	100%	0.785	3
21	$e^x + \log y$	100%	0.527	2
22	$3.2x^5 + 5.3x^3 + 4.7$	80%	0.952	3
23	$a^4 + a^3 + a^2 + a$	90%	1.02	4
24	$(x - 77.72)^2 + (y - 106.48)^2$	76%	1.688	5
25	$5a^4 + 4a^3 + 3a^2 + 2a + 1$	80%	1.28	5
26	$e^{x/3}\cos(3x)/2$	25%	0.312	1
27	$0.82xy\cos(x - y)$	36.7%	0.383	1
28	$4.771 * (3.714 - \omega^2) + \cos\theta + (3.714 - \omega^2)\cos\theta$	20%	1.212	3
29	$\log(4 + 2\sin x \sin(8x)) + e^{\cos(3x)}$	0%	0.558	2

[1]*Success Rate* represents the ratio of the number of experiments having successfully met the accuracy requirements to the total number of experiments.

[2]*Average Time of Success* is the average time taken to detect the target function.

[3]*Number of Genes* is the number of Genes required obtaining the highest *Success Rate.*

Table 4. Comparison of the experiment results on Problem-1 and Problem-2 for GEP with RM_INTV, DE-GEP, pGEP.

	Statistics	GEP with RM_INTV[1]	DE applied to Every Generation	MLS
Problem-1	**Best residual[2]**	0.157	5.901e(-6)	0
	Avg. of best residuals[3]	0.966	2.913e(-4)	4.36e(-9)
Problem-2	**Best residual[2]**	1.038	0.0089	9.328e(-4)
	Avg. of best residuals[3]	1.863	0.2572	0.0343

[1]GEP with RM_INTV is an algorithm for constant creation in [19].

[2]*Best residual* represents the best residual error among the final best individuals after 100 runs.

[3]*Avg. of best residuals* is the average value of all 100 final best residual errors.

(2) Problem-2 is a "V" shaped function with complex structure and floating pointing coefficients. In Table 3, the data of Problem-2 demonstrate that pGEP algorithm has results several orders of magnitude higher than GEP with RM_INTV in terms of Best residual and Avg. of best residuals. The pGEP algorithm is also one order of magnitude higher than DE-GEP.

As mentioned in [19], the approach where DE is invoked in every generation is computationally expensive and takes significant additional time. In contrast, pGEP has the advantage of low time cost and can be applied to every generation. Table 2 and Table 3 clearly demonstrate that pGEP has a higher success rate, better accuracy and lower computational cost.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a parallel version of GEP, (pGEP). In every generation, GEP concentrated on evolving the structure of the function expression by essential genetic operations and MLS obtains optimal local constant coefficients from the fixed function structures. If GEP fails to evolve an appropriate formula structure, the power of MLS declines significantly. We modified the storage form of the ET-tree into linear structure and parallel GEP algorithm on GPU. The experimental results show that the CUDA-based implementation exploits the intrinsic parallelism in the evaluation process and achieves significant speedup. Then we compared pGEP with other existing well-known constant creation methods in terms of accuracy. The MLS is far less time consuming than other local optimization methods, and provides a significant improvement in performance.

As the GEP linear expression must be converted to an ET tree with pointer first enabling transformation into a post order traversal linear sequence, it consumes considerable time in the evaluation process. Future work we identified is to directly translate the GEP linear expression into post order traversal linear sequence without the process of constructing the ET tree. Further, we plan to test the addition of random constants in the terminal set to improve the ability of searching complex nest-structure expressions.

6. REFERENCES

[1] C. Ferreira. Gene expression programming. 2001. *A new adaptive algorithm for solving problems*. Complex Systems, 13(2):87–129.

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

[3] Michael Schmidt, Hod Lipson. 2009. *Distilling Free-Form Natural Laws from Experimental Data.* Vol 324, SCIENCE 2009.

[4] Michael Schmidt, Hod Lipson. 2009. *Solving Iterated Functions Using Genetic Programming.* GECCO'09, ACM 978-1-60558-505-5/09/07.

[5] Michael Schmidt, Hod Lipson. 2009. *Discovering a Domain Alphabet.* GECCO'09, ACM 1-58113-000-0/00/0004.

[6] Michael Schmidt, Hod Lipson. 2009. *Incorporating Expert Knowledge in Evolutionary Search: A Study of Seeding Methods.* GECCO'09, ACM 1-58113-000-0/00/0004.

[7] Michael Schmidt, Hod Lipson. 2010. *Symbolic Regression of Implicit Equations. Genetic Programming Theory and Practive VII, Genentic and Evolutionary Computation.* Springer Science + Business Media, LLC.

[8] M. Mitchell. 1996. *An Introduction to Genetic Algorithms.* MIT Press.

[9] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. 2008. GPU computing. IEEE Proceedings, May 2008, 879-899.

[10] NVIDIA CUDA. http://developer.nvidia.com/category/zone/cuda-zone.

[11] CUDA C Programming Guide 3.2, 2010.

[12] Steven J. Miller. 2006. *The Method of Least Squares.* Mathematics Department Brown University, pp. 1-7.Providence: Brown University.

[13] Matthew Evett, Thomas Fernandez. 1998. *Numeric Mutation Improves the Discovery of Numeric Constants in Genetic Programming.* Proceedings of the Third Annual Genetic Programming Conference. Madison, Wisconsin 66-71.

[14] Alexander Topchy, William F. Punch. 2001. *Faster Genetic Programming based on Local Gradient Search of Numeric Leaf Values.* Proceedings of the Genetic and Evolutionary Computation Conference. San Francisco, California 155-162

[15] D.B.Kirk and W.mei W.Hwu. 2010. *Programming Massively Parallel Processors.* Morgan Kaufmann, first edition.

[16] Koza, J. 1997. Tutorial on advanced genetic programming, at genetic programming.

[17] C. Ferreira. 2002. *Function finding and the creation of numerical constants in gene expression programming*. In: 7th Online World Conference on Soft Computing in Industrial Applications, September – October.

[18] X. Li, C. Zhou, P. C. Nelson, T. M. Tirpak. 2004. *Investigation of constant creation techniques in the context of gene expression programming.* In: Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA, July.

[19] Q.Zhang, C.Zhou, W.Xiao, Peter C.Nelson, X.Li. 2006. *Using Differential Evolution for GEP Constant Creation.* In: Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO 2006), Seattle, WA, USA.

[20] X. Li, C. Zhou, P. C. Nelson, T. M. Tirpak. 2004. *Investigation of constant creation techniques in the context of gene expression programming.* In: Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA, July.

[21] NVIDIA. Data-Parallel algorithms. http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html #reduction, December 2011