Acceleration of Grammatical Evolution Using Graphics Processing Units

Computational Intelligence on Consumer Games and Graphics Hardware

Petr Pospichal
Faculty of Information Technology
Brno University of Technology
Czech Republic
ipospichal@fit.vutbr.cz

Eoin Murphy
Natural Computing Research and Applications Group
University College Dublin
Ireland
eoin.murphy@ucd.ie

Michael O'Neill
Natural Computing Research and Applications Group
University College Dublin
Ireland
m.oneill@ucd.ie

Josef Schwarz
Faculty of Information Technology
Brno University of Technology
Czech Republic
schwarz@fit.vutbr.cz

Jiri Jaros
Faculty of Information Technology
Brno University of Technology
Czech Republic
jarosjir@fit.vutbr.cz

ABSTRACT

Several papers show that symbolic regression is suitable for data analysis and prediction in financial markets. Grammatical Evolution (GE), a grammar-based form of Genetic Programming (GP), has been successfully applied in solving various tasks including symbolic regression. However, often the computational effort to calculate the fitness of a solution in GP can limit the area of possible application and/or the extent of experimentation undertaken. This paper deals with utilizing mainstream graphics processing units (GPU) for acceleration of GE solving symbolic regression. GPU optimization details are discussed and the NVCC compiler is analyzed. We design an effective mapping of the algorithm to the CUDA framework, and in so doing must tackle constraints of the GPU approach, such as the PCI-express bottleneck and main memory transactions.

This is the first occasion GE has been adapted for running on a GPU. We measure our implementation running on one core of CPU Core i7 and GPU GTX 480 together with a GE library written in JAVA, GEVA.

Results indicate that our algorithm offers the same convergence, and it is suitable for a larger number of regression points where GPU is able to reach speedups of up to 39 times faster when compared to GEVA on a serial CPU code written in C. In conclusion, properly utilized, GPU can offer an interesting performance boost for GE tackling symbolic regression.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms

Keywords

CUDA, grammatical evolution, graphics chips, GPU, GPGPU, speedup, symbolic regression

1. INTRODUCTION

Problems of symbolic regression require finding a function, in symbolic form that fits a given finite sampling of data points [11]. Areas of applications include econometric modeling and forecasting, image compression and others.

Grammatical evolution [20, 4], a grammar-based form of Genetic Programming [13], is a promising tool based on the fusion of evolutionary operators and formal grammars. It has been successfully applied to various problems including symbolic regression [17, 1]. Although GE is very effective in solving many practical problems, like all GP methods, its execution time can become a limiting factor for computationally intensive problems, as a lot of candidate solutions must be evaluated, and each evaluation is expensive.

Driven by ever increasing requirements from the video game industry, graphics chips (GPUs) have evolved into very powerful and flexible processors, while their price has remained in the range of the consumer market. They now offer floating-point calculations much faster than today's CPU and, beyond graphics applications; they are very well suited to address general problems that can be expressed as data-parallel computations (i.e., the same code is executed on many different data elements)[9].

In this paper, we explore the possibility of using consumer-level a GPU for acceleration of the grammatical evolution solving symbolic regression problems.

The remainder of the paper is organized as follows. The next section describes the Grammatical Evolution and possible areas of its application. Section 3 deals with GPUs in the context of CUDA general purpose computations with special focus on optimization techniques used in recent papers. The focus of section 4 is a GPU utilization analysis, where the CUDA compiler is discussed in detail. A mapping of the grammatical evolution algorithm to the GPU hardware is described in the next section. After that, performance and convergence of the proposed algorithm is measured and compared with the GEVA GE library. The paper concludes with section 7.

2. GRAMMATICAL EVOLUTION

The basic motivation behind Grammatical Evolution (GE) [20, 4, 19] is to "evolve complete programs in an arbitrary language using variable length binary strings" [19]. It is being used for various tasks including financial modelling [17], 3D Design [2] and game strategies [5].

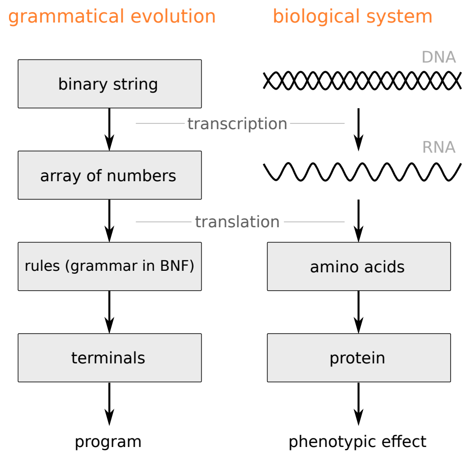


Figure 1: Grammatical evolution

As it is shown in Fig. 1, the process of Grammatical Evolution is inspired by biological systems. Whereas in living organisms, the transformation of genotype to phenotype is performed by transcribing DNA into intermediate RNA and then translated into amino acids forming proteins, GE resembles this process by using an intermediate representation and building rules in a form of a grammar.

A grammar in the Backus-Naur form allows users to define constraints as well as potential building blocks for the problem at a hand. Embedded domain knowledge in this manner can be beneficial in some applications [19, 21]. GE is a population-based, iterative, stochastic algorithm that uses genetic operators known from other methods of the evolutionary computation [8]. It is formed by the following steps:

selection ensures propagation of fitter individuals to the next generation, which allows convergence towards a solution

crossover is responsible for mixing good features of individuals together [7]

mutation includes small changes into genotype which has a positive effect for overcoming local extremes [2, 3]

genotype-phenotype mapping is a process of rewriting a genotype (binary string or array of integers) into a phenotype (evolved program) using the user-defined grammar

evaluation is a problem-dependent process of getting the fitness value of individual's phenotype. In this paper, we focus on symbolic regression, which can be defined as a sum of local differences in a set of data points:

$$fitness = \sum_{i=0}^{n} |x[i] - f[i]| \tag{1}$$

where $x[i]$ is the value of individuals phenotype, $f[i]$ is the value of the desired solution and n is the number of regression points.

Additional information and examples are available in [18].

3. GRAPHICS PROCESSING UNITS (GPUS)

Historically, GPUs were used exclusively for fast rasterization of graphics primitives such as lines, polygons and ellipses. These chips had a strictly fixed functionality. Over time, a growing gaming market and increasing game complexity won GPUs limited programmable functionality. This turned out to be very beneficial, so their capabilities quickly developed up to a milestone, unified shader units. This hardware and software model has given birth to the nVidia Compute Unified Device Architecture (CUDA) framework [16], which is now often used for General Purpose Computation on these GPUs (GPGPU) with interesting results [23, 24].

3.1 CUDA

The CUDA hardware model is shown in Fig. 2: the graphics card is divided into a graphics chip (GPU) and main memory. Main memory, acting as an interface between the host CPU and GPU, is connected to the host system using a PCI-Express bus. This bus has a very high latency and low transfer rates in comparison to inter-GPU memory transfers [23]. Main memory is optimized for stream processing and block transactions as it has low bandwidth compared to the GPU on-chip memory. Actual GPUs consist of several independent Single Instruction, Multiple Data (SIMD) engines called stream multiprocessors (SM) in nVidia's terminology. Simple processors (CUDA cores) within these multiprocessors share an instruction unit and a hardware scheduler so they are unable to execute different codes in parallel but, on the other hand, can be synchronized quickly in order to maintain data consistency. Multiprocessors also possess a small amount (16-48KB) of very fast, shared memory and a read-only cache for code and constant data. Newer, DirectX 11 GPUs have also a read-write L1 cache and some of them have a L2 cache as well.

The CUDA software model maps all mentioned GPU features to actual user programs. The programmer's job is to perform this mapping efficiently to fully utilize the capabilities of the GPU.

The main advantage of GPUs is very high raw floating point performance resulting from a moderate degree of parallelism. Proper usage of this hardware can lead to a speedup up to hundred times compared to general CPUs. But in order to utilize such power, a programmer must consider a variety of restrictions:

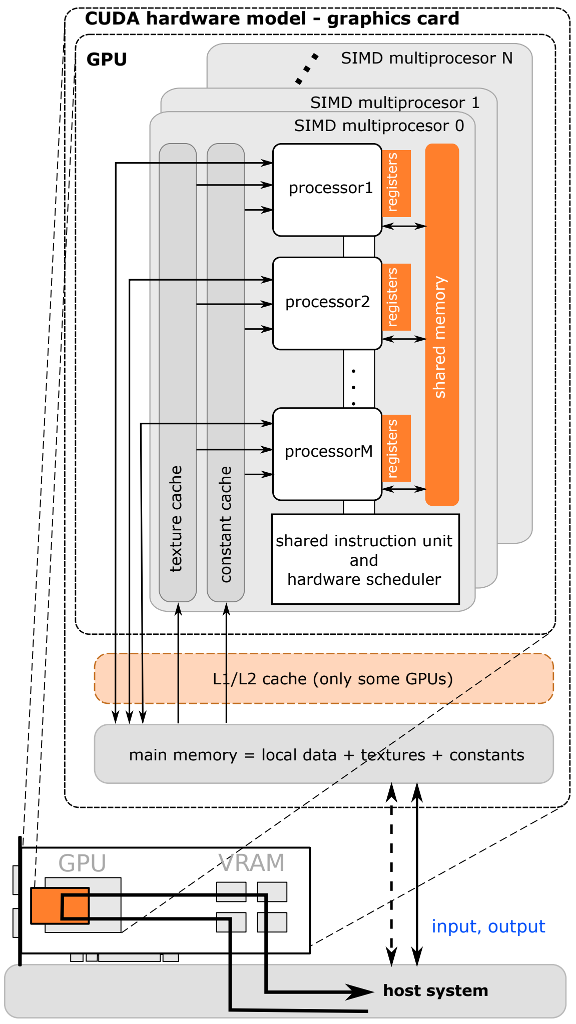


Figure 2: CUDA hardware model

- GPUs require massive parallelism in order do be fully utilized. Applications must be therefore decomposable into thousands of relatively independent tasks.
- GPUs are optimized for the SIMD type processing, meaning that the target application must be data parallel otherwise the performance is significantly decreased.
- A graphics card is connected to the host system via a PCI-Express bus, which is, compared to GPU memory, very slow (80x for GTX285).

  The OS driver transfer overhead is also very performance-choking for small tasks, so applications should minimize the number of data transfers between CPU and GPU. The GPU must also be utilized for a sufficiently long time in order to obtain meaningful speedup.
- A properly designed application should also take into account the memory architecture. Transactions to main GPU memory are up to 500x slower in comparison with on-chip transfers.
- GPUs are optimized for `float` data type, `double` is usually very slow.

3.2 CUDA compiler

CUDA allows the compilation of the C source code to an intermediate PTX assembly language as well as to a binary CUBIN package. PTX has the advantage of being compiled at runtime to a specific GPU and allows user modification without a need to run the whole NVCC package. On the other hand, module load time is, in the case of PTX, much higher, which brings an unpleasant overhead upon GPU invocation.

CUDA allows the caching of the source code on the graphics memory so that code doesn't have to be transferred before each execution.

4. GPU UTILIZATION ANALYSIS

As mentioned earlier, GE consists of several independent steps: selection, crossover, mutation, genotype-phenotype mapping and evaluation. The most straightforward way to utilize the GPU for acceleration of GE is to outsource the most time-consuming part, evaluation, to the GPU. This approach shows benefit of the least programming effort but it also has a major limitation: due to CPU-GPU connection, data has to be transferred to and from GPU every generation, which is a serious performance bottleneck. In our work, we have chosen the approach of running the whole Grammatical Evolution algorithm on a GPU.

Experiments implemented by Pospichal and Jaros [23, 24] indicate that avoiding conditions in source code by compiling algorithm parameters directly into GPU code can lead to significant speedup. In this paper we have applied the same approach.

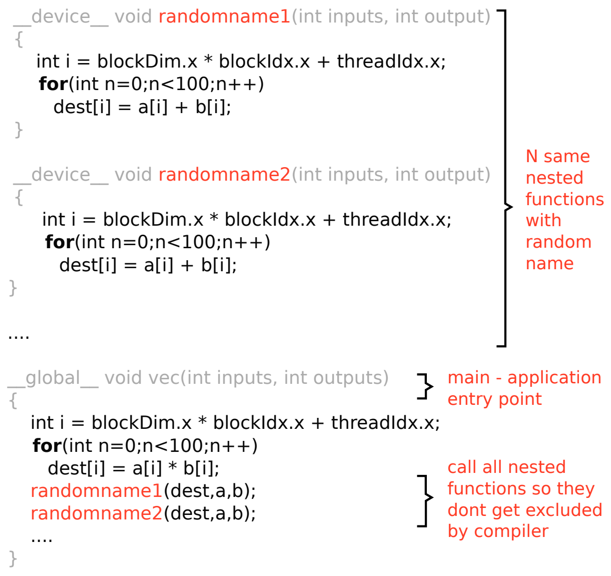
GPU performance is very sensitive to divergent branching in the code path. Therefore, we considered an option of generating optimized source code for evaluating the population of individuals with respect to the current population characteristics. This source code would be compiled and uploaded to the GPU upon every iteration of GE so that evaluation is effective.

In order to do this, we would have needed either a fast compiler or the option of modifying GPU code directly so that the GPU part would not be slowed down by this operation. We have investigated this option with a simple experiment illustrated in figures 3. We simulated rising compiler input source code complexity as shown in Fig. 3(a). Every generated source code was compiled and run 10× . Fig. 3(b) shows that both PTX and CUBIN versions of compilation took at least 350ms. This is too much as one generation of GE often takes no more than a second (depending on parameters). Because of these results, our implementation doesn't compile every population of individuals each generation.

On the other hand, module load times are very different (see Fig. 3(c) and 3(d)). The CUBIN module is loaded in less than 0.12ms while PTX takes as much as 80ms. As a result, we chose CUBIN and compromise between optimization and compiler invocation overhead – GPU source code is compiled once in the beginning of the GE run. This gets rid of 350ms of compilation time every generation and offers good optimization based on GE parameters [6].

5. GE RUNNING ON GPU

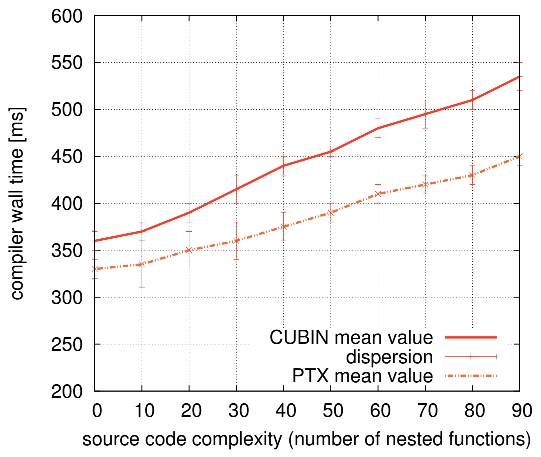
The CUDA software model requires programmer to identify application parallelism on three levels of abstraction: kernels, thread blocks and threads within these blocks [25,



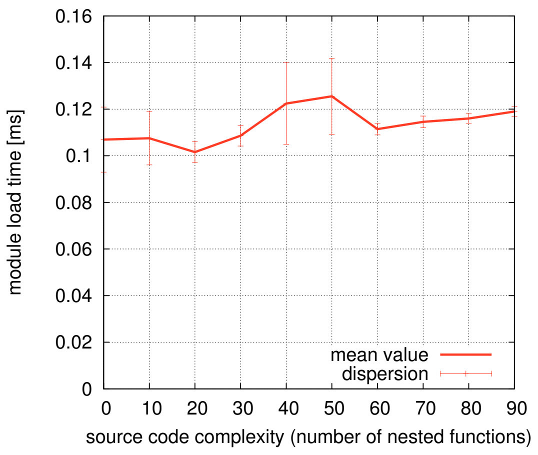
```
__device__ void randomname1(int inputs, int output)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   for(int n=0;n<100;n++)
     dest[i] = a[i] + b[i];
}

__device__ void randomname2(int inputs, int output)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    for(int n=0;n<100;n++)
      dest[i] = a[i] + b[i];
}

....

__global__ void vec(int inputs, int outputs)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   for(int n=0;n<100;n++)
     dest[i] = a[i] * b[i];
   randomname1(dest,a,b);
   randomname2(dest,a,b);
   ....
}
```

N same nested functions with random name
main - application entry point
call all nested functions so they dont get excluded by compiler

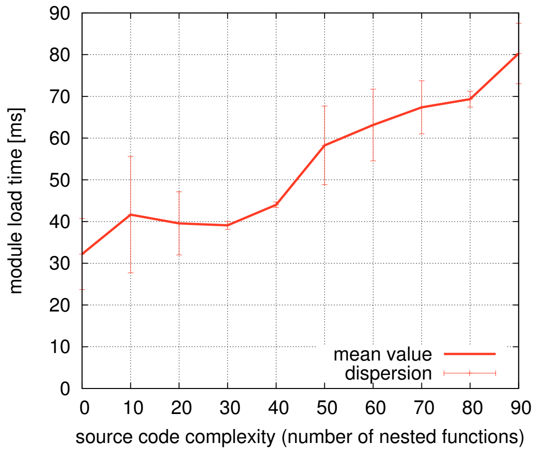
(a) source code of nested functions experiment



(b) comparision of CUBIN and PTX compiler times depending on source code complexity



(c) CUBIN module load times



(d) PTX module load times

Figure 3: CUDA NVCC compiler analysis

15, 16, 14, 22]. Kernels are complete programs executed independently on the GPU hardware. The GPU hardware scheduler dynamically maps blocks to SIMD multiprocessors and threads to processors within them during runtime. Because of this and the CUDA hardware model, threads should execute the same code over different data and use their shared memory extensively to avoid main memory transactions [24].

In some research [23, 24], this mapping has been performed by running individuals by threads and independent islands by blocks. However for GE, it may not be the best option for these reasons:

- Individuals are running potentially different code (interpreting the symbolic regression problem)
- The most time-consuming part is the fitness function evaluation, which can be parallelized for many problems
- Individuals use larger amounts of shared memory, so island populations would be very limited

Therefore, we have decided to alter the mapping to the CUDA software model so that individuals are maintained by thread blocks and threads are used to manipulate individuals (i.e., the number of thread blocks equals the number of individuals in the population). This brings massive parallelism needed for GPU to reach its full potential as well as one major issue: according to the nVidia documentation, thread blocks cannot be mutually synchronized. This is necessary for data consistency, as selection should not happen before evaluation, evaluation before genotype-phenotype mapping and so on. To overcome this, we separated evolution into two independent kernels which ensures synchronization between blocks upon kernel completion.

The concept of our system is shown in Fig. 4. The application is started on the CPU with defined GE parameters and the NVCC compiler is invoked to compile CUDA kernels with defined macros as these parameters. GPU code compilation and transfer is actually performed only if the GPU has never run a program with the same parameters before, as NVCC caches previous programs in the GPU. Next, the

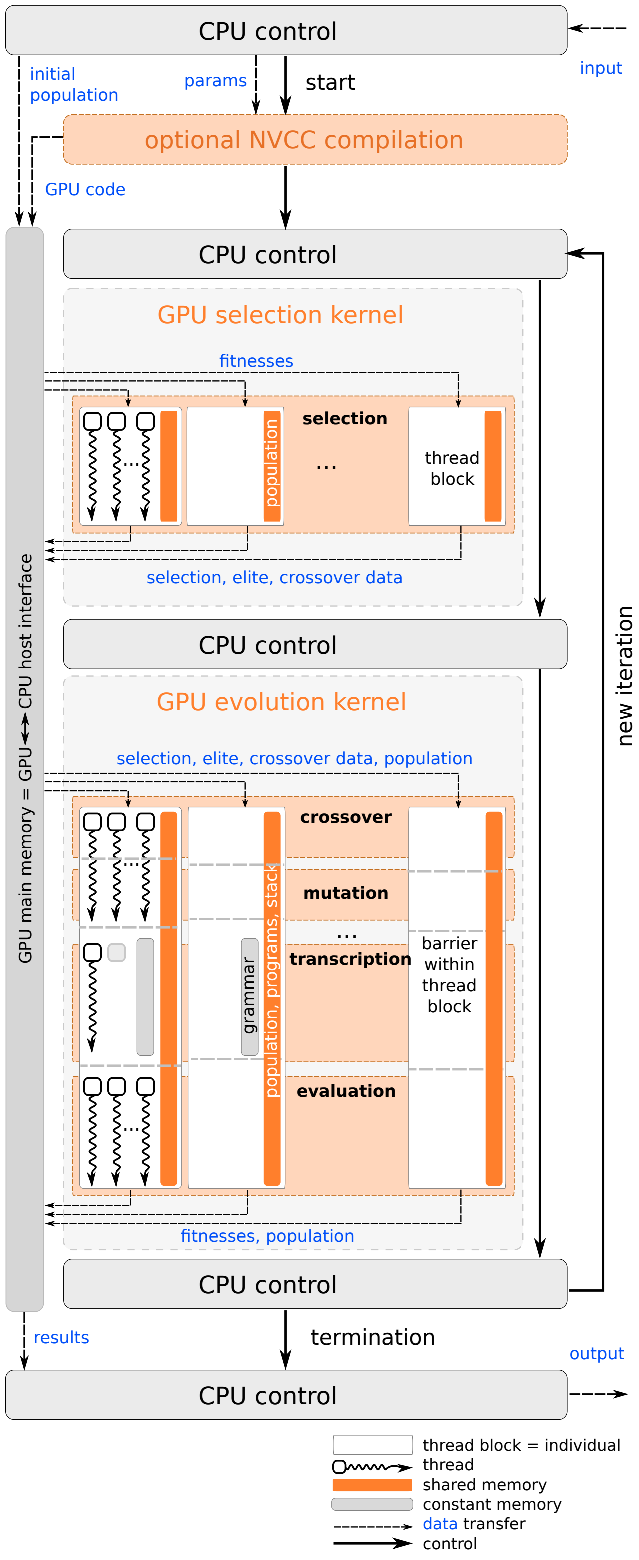


Figure 4: Grammatical evolution running on GPU

GPU is initialized with a random population and other initial values. Following this an iterative algorithm of evolution consisting of two successive kernels is executed.

The first GPU kernel performs selection while the second is used for the rest of the evolutionary process.

All kernels use the same data pointers to main memory, so the CPU doesn't need to copy data back and forth every iteration of GE, therefore the PCI-Express bottleneck is avoided. Kernels also copy just the minimum amount of data between shared and main memory with respect to effective block transfers. This tight interoperability eliminates most of the main memory transactions as the population is kept in fast-shared memory through the whole process. Constant cache is used for grammar data, and the implementation uses an effective PRNG GPU generator from [22, 14].

As noted earlier, GE is mapped to the GPU so that thread blocks running in parallel on SIMD multiprocessors are maintaining individuals. Threads within these blocks, on the other hand, are running on processors (CUDA cores) in SIMD. Thus, there are two levels of parallelism: 1) individuals are evaluated in parallel and 2) data within individuals (genes, crossover points, mutations, fitness points, etc.) are maintained by parallel access as well. GE phases are performed as follows:

selection begins with a parallel copy of fitness values of all individuals to the shared memory. Elitism is performed afterwards by a single thread walking the array of fitness values. Then, the selection is performed: threads up to tournament_size index (i.e. tournament_size = 3, so threads 0,1,2) perform random number generation as Tournament selection. The fittest individual index is then chosen as the tournament winner and written as an index to main memory. Random crossover points and probabilities for the following kernel are also generated and stored by each thread. These values are later used by the second kernel. As a consequence, there is no need for inter block synchronization after the second kernel is launched.

crossover is mixing genes of two individuals together and therefore requires inter-individual (inter-thread block) communication, so shared memory cannot be used. This is solved by using main-memory data gathered by the previous kernel. From now on, each thread block stores its individual in fast, shared memory. In the case of elitism, the first thread block just reads elite individuals. Other thread blocks are working in pairs (i.e., block 1 and 2, 3 and 4,...) performing crossover if the probability to read from main memory is higher than the crossover_rate. Crossover itself is performed by reading corresponding parts of chromosomes to the shared memory.

mutation is performed on every gene with a defined probability upon writing to the population in shared memory. This is achieved by threads generating random values in parallel and then mutating if necessary.

genotype-phenotype mapping is a serial process as the rewritten nonterminal from a final string depends on the previous grammar rule applied. As a consequence, mapping is the only part of GE which is performed on GPU by a single thread in a thread block (individual). In addition to shared memory used to store the chromosome and generated programs, grammar rules are kept in constant data cache while rules are encoded in a form of array as shown in Fig. 5. A GEVA grammar

```
// GEVA grammar definition
<expr> ::= ( <op> <expr> <expr> ) | <var>
<op>   ::= +  | -  | *
<var>  ::= x0 | 1.0


// GPU grammar definition
__constant__ const int grammar_metadata[][2]=
{
//  # of rules , beginning index
  {2, 0},  // EXPR
  {3, 2},  // OP
  {2, 5}   // VAR
};

// constant precalculated table grammar_rules
__constant__ const int grammar_rules[][5]=
{
//  # of NT , # of symbols , rule symbols
  {3,3, EXPR,     EXPR,   OP },// <expr> ::=
                               //  <expr> <expr> <op>
  {1,1, VAR,      EMPTY,  EMPTY},// <expr> ::= <var>
  {0,1, OPPLUS,   EMPTY,  EMPTY},// <op>    ::= +
  {0,1, OPMINUS,EMPTY,  EMPTY},// <op>    ::= -
  {0,1, OPMUL,    EMPTY,  EMPTY},// <op>    ::= *
  {0,1, VAR1,     EMPTY,  EMPTY},// <var>   ::= x
  {0,1, ONE,      EMPTY,  EMPTY} // <var>   ::= 1.0
};
```

Figure 5: Grammar rules in GEVA and CUDA

definition requires further decoding while our CUDA grammar uses preprocessor integer constants EXPR, OP, VAR, etc. for each symbol. An individual's string is rewritten during mapping by accessing arrays grammar_metadata and grammar_rules very quickly in the GPU constant cache. This completely eliminates slow main-memory transfers. More information about the process of mapping can be found in [20].

evaluation is implemented by all threads in thread blocks in parallel: symbolic regression points are simulated using the GPU stack described by Langdon [12] and mapped strings are compared with the desired solution value in each point computed on-fly from the predefined target expression. Finally the sum of differences in these points are computed serially by a single thread.

GPU code written in lowlevel C is maintained by the Python CUDA wrapper PyCUDA[10]. According to the Klockner, actual CUDA calls are done in C++ so GPU measurements have no additional overhead.

5.1 Limitations

Our implementation uses shared memory transactions. The size of this memory is limited to 16, resp. 48 KB per SIMD multiprocessor (individual) in case of pre-Fermi resp. Fermi GPUs. There is a possibility to use main memory (size varies around 1GB), but these transactions are much slower so they would degrade speed (actual impact depends on how often is main memory used).

6. RESULTS

In the following sections, we compare three implementations of grammatical evolution:

CPU_G is implemented using the GEVA library (JAVA language)

Table 1: Algorithm parameters

parameter	value	
	convergence	performance
mutation rate	5%	
crossover rate	90%	
mutation operator	Integer flip	
selection operator	Tournament	
replacement policy	Generational	
elitist size	1	
tournament size	3	
initialization	random	
chromosomze size	128	
symbolic reg. problem	$x + x^2 + x^3 + x^4$	
symbolic reg. terminals	$\{+, -, *, x, 1\}$	
symbolic reg. interval	$\langle 0; 10 \rangle$	
symbolic regr. points	128	$\{128, 256, 1280, 2560\}$
population size	32	$\{2, 4, 8, 16, 32, 64\}$
generations	100	1000

GPU is the previously described parallel GPU implementation (C language) running on nVidia Geforce GTX 480 GPU

CPU_C is a serial (single-threaded) CPU version of the described GPU implementation (C language) where threads as well as thread blocks are simulated using `for` cycles

Our primary focus was to compare performance. In addition we performed a convergence test as well to see if all algorithms were able to optimize the selected problem. For testing purposes, we used settings shown in table 1 together with hardware and software described in table 3.

6.1 Algorithms convergence

As a convergence test, we measured the success rate of 100 runs with the random population initialization. The success was defined as fitness of the best individual in the last generation has zero value (i.e. solution is found). We observed 77% success rate in the case of CPU_C and GPU implementations and 74% in case of CPU_G. Just 3% difference indicating that all algorithms are able to optimize the examined problem similarly.

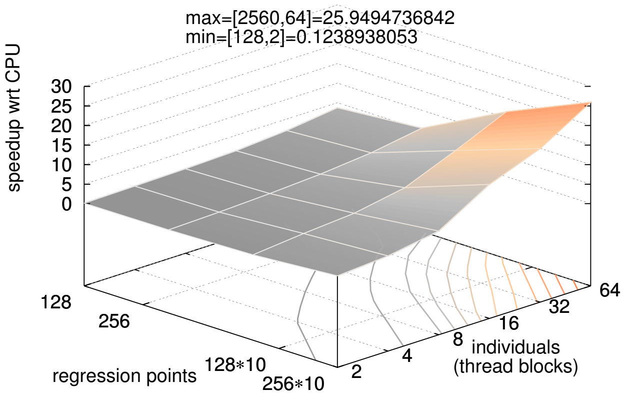
6.2 Performance

The execution time was measured using the Unix `time` utility in all cases, in addition for GPU, we measured kernels execution times as well. GPU run is thereby evaluated both with and without additional time overhead resulting from data copy to GPU, GPU initialization and `NVCC` compiler execution [6].

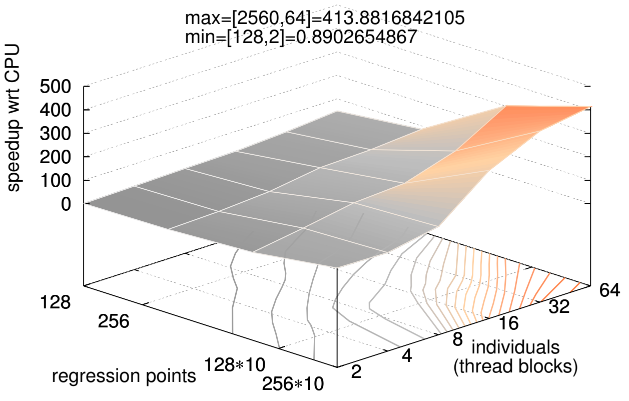
The overhead times are more or less constant so the less the program spends time utilizing GPU, the more overhead affects total timings. Thereby in general, we can say that for an infinite number of generations, times and speedup will be close to measurements excluding overhead.

In the following paragraphs, the GPU execution speed is compared with other two implementations of the GE, CPU_G and CPU_C.

Each GE implementation has been executed 10 times for all 24 combinations of input parameters showed in table 3. The averaged results rounded to 1 decimal digit are shown in table 2.



(a) CPU_C and GPU execution speed comparision



(b) CPU_G and GPU execution speed comparision

Figure 6: Speedup comparision including GPU overhead

Table 2: Speedup comparision

implementation	min	max	avg
execution time [s]			
GPU without overhead	0.2	1.6	0.5
GPU with overhead	0.9	2.4	1.3
CPU_G	1.0	982.9	175.2
CPU_C	0.1	61.6	9.3
***GPU* speedup including overhead**			
CPU_G	0.9×	413.9×	102.8×
CPU_C	0.1×	25.9×	5.3×
***GPU* speedup excluding overhead**			
CPU_G	5.4×	636.7×	215.4×
CPU_C	0.8×	39.0×	11.0×
CPU_C speedup			
CPU_G	7.2×	32.1×	20.6×

Table 3: Testing environment

hardware	
CPU	Core i7 3.3GHz
GPU	nVidia GeForce GTX 480
software	
OS	Ubuntu Linux 10.04, 64bit
CUDA	SDK v 3.2, driver 260.19.06-0ubuntu1
java	6.20dlj-0ubuntu1.9.10
Python	2.6 + PyCUDA 0.94.2
	GEVA v 1.2
GE	presented custom GE, serial CPU version
	presented custom GE, parallel GPU version

Evidentely execution times vary greatly. Obviously worst performance was observed with the GEVA implementation (CPU_G), which is on average for all runs 20.6× slower than serial CPU version written in C (CPU_C) and more than 400× resp. 600× slower than GPU including resp. excluding overhead times. However, the serial CPU version is faster than the *GPU* version in some cases. Examining surface plots shown in Fig. 6, we can observe that GPU performance is significantly better in tasks where there is enough data to exploit GPU's massively-parallel nature. Such situations can lead to speedups up to 25× (39× for sufficiently difficult problem) compared to CPU_C and several hundred times in comparision with the GEVA library. On the other hand, GPU is unsuitable for simple tasks where data transfer and compilation overhead take their toll. comparision

The GEVA library is written in a very general fashion. This allows programmers to easily experiment with virtually any part of the Grammatical Evolution algorithm, but at the same time it costs a lot of performance during execution. Furthermore, JAVA is an interpreted, high-level language. Table 2 shows that a well-designed, single-threaded version of the same algorithm written in C can perform on average 20× faster.

In general, we can say that GPU is very suitable for difficult data-parallel tasks. In the case of Grammatical Evolution used for symbolic regression problems, GPUs can offer very interesting performance boost up to 400× in comparision with GEVA. However, C code running on a CPU can perform better on simple problems as GPU utilization involves some overhead.

7. CONCLUSIONS

In this paper, we have focused on the possibility of accelerating Grammatical Evolution (GE) using graphics processing units (GPUs). This is the first attempt to run GE entirely on a GPU.

We have briefly introduced GE and its ability to solve problems, followed by an overview of the architecture of modern nVidia GPUs. The architecture has been presented with special focus on performance optimization techniques used in the literature. In order to utilize GPU efficiently, we have analyzed the possibility of compiling GPU code every GE iteration and based on presented data made a decision to run the compiler at the beginning of evolution. This offers a good code optimization overhead tradeoff.

Section 5 has shown our innovative mapping of GE to the CUDA GPU hardware model featuring two levels of parallelism: individuals are evaluated on multiprocessors and threads are used to maintain individuals. This model is tested on both CPU and GPU and compared to standard

GEVA framework running on Core i7 3.3 Ghz with nVidia GeForce GTX 480 GPU.

It is shown that GPU is suitable especially for tasks with larger numbers of symbolic regression points (1280,2560) evaluated in parallel where it performs up to 636× resp. 39× faster compared to GEVA and serial CPU code, respectively. For smaller problems, the same lowlevel CPU code written in C can perform better as GPU utilization involves additional overhead of the code compilation, data transfer and initialization.

Overall, we have shown that properly utilized mainstream GPU is an interesting hardware platform for acceleration of grammatical evolution solving symbolic regression problems.

8. ACKNOWLEDGMENTS

This research has been carried out under the financial support of the research grants "Natural Computing on Unconventional Platforms", GP103/10/1517 (2010-2013) of the Czech Science Foundation, "Security-Oriented Research in Information Technology", MSM 0021630528 (2007-13), the BUT FIT grant FIT-S-11-1, and with financial support of GA CR 102/09/H042 and FR2641/2011/G1. This research is also based upon works supported by the Science Foundation Ireland under Grant No. 08/IN.1/I1868

9. REFERENCES

[1] A. Brabazon and M. O'Neill. *Biologically Inspired Algorithms for Financial Modelling*. Springer, 2006.

[2] J. Byrne, J. McDermott, E. Galvan-Lopez, and M. O'Neill. Implementing an intuitive mutation operator for interactive evolutionary 3d design. In *IEEE Congress on Evolutionary Computation*, 2010.

[3] J. Byrne, M. O'Neill, and A. Brabazon. Structural and nodal mutation in grammatical evolution. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1881–1882, Montreal, Québec, Canada, 8-12 July 2009. ACM.

[4] I. Dempsey, M. O'Neill, and A. Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*. Studies in Computational Intelligence. Springer, 2009.

[5] E. Galvan-Lopez, J. Swafford, and M. O'Neill. Evolving a ms.pac-man controller using grammatical evolution. In *EvoGAMES 2010 the 2nd European event on Bio-inspired Algorithms in Games*. Springer, 2010.

[6] S. Harding and W. Banzhaf. Fast genetic programming on gpus. In *Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101. Springer Berlin / Heidelberg, 2007.

[7] R. Harper and A. Blair. A structure preserving crossover in grammatical evolution. In *IEEE Congress on Evolutionary Computation*, pages 349–358, 2001.

[8] J. Jaros. Evolutionary optimization of multistage interconnection networks performance. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1537–1544, Montreal, Québec, Canada, 8-12 July 2009. ACM.

[9] D. Kirk and W. Whu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[10] A. Klockner, N. Pinto, Y. Lee, B. Catazaro, P. Ivanov, and A. Fasih. Pycuda: Gpu run-time code generation for high-performance computing.

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

[12] W. B. Langdon and W. Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *Proceedings of the 11th European conference on Genetic programming*, EuroGP'08, pages 73–85, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] R. McKay, X. Nguyen, P. Whigham, Y. Shan, and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–296, 2010.

[14] Nguyen and Hubert. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[15] nVidia. Cuda c best practices guide.

[16] nVidia. Cuda programming guide 3.0.

[17] M. O'Neill, T. Brabazon, C. Ryan, and J. Collins. Evolving market index trading rules using grammatical evolution. In *EvoIASP 2001*, 2001.

[18] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA: Grammatical evolution in java. *SIGEVOlution*, 3(2), 2008.

[19] M. O'Neill and C. Ryan. Grammatical evolution. In *IEEE Transactions on Evolutionary Computation*, pages 349–358, 2001.

[20] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*. Genetic programming. Kluwer Academic Publishers, 2003.

[21] M. O'Neill, J. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg. Shape grammars and grammatical evolution for evolutionary design. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1035–1042, Montreal, Québec, Canada, 8-12 July 2009. ACM.

[22] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[23] P. Pospichal, J. Schwarz, and J. Jaros. Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[24] P. Pospichal, J. Schwarz, and J. Jaros. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In *16th International Conference on Soft Computing MENDEL 2010*, pages 64–70. Brno University of Technology, 2010.

[25] D. Tarjan, K. Skadron, and P. Micikevicius. The art of performance tuning for cuda and manycore architectures.