EFFECTIVE MAPPING OF GRAMMATICAL EVOLUTION TO CUDA HARDWARE MODEL

Petr Pospichal
Doctoral Degree Programme (2), FIT BUT
E-mail: ipospichal@fit.vutbr.cz

Supervised by: Josef Schwarz
E-mail: schwarz@fit.vutbr.cz

Abstract: Several papers have shown that symbolic regression is suitable for data analysis and prediction in finance markets. The Grammatical Evolution (GE) has been successfully applied in solving various tasks including symbolic regression. However, performance of this method can limit the area of possible applications. This paper deals with utilizing mainstream graphics processing unit (GPU) for acceleration of GE solving symbolic regression. With respect to various mentioned constrains, such as PCI-Express and main memory bandwidth bottleneck, we have designed effective mapping of the algorithm to the CUDA framework. Results indicate that for larger number of regression points can our algorithm run 636 or 39 times faster than GEVA library routine or a sequential C code, respectively. As a result, the ordinary GPU, if used properly, can offer interesting performance boost for solution the symbolic regression by the GE.

Keywords: GPU, Graphics Processing Units, Grammatical Evolution, CUDA, Symbolic Regression, Speedup, C

1 INTRODUCTION

The Grammatical evolution is a promising technique based on the fusion of evolutionary operators and formal grammars. It was firstly introduced by O'Neil[2] in 1998 and it is still under active development nowadays [3]. Basic motivation is to "evolve complete programs in an arbitrary language using variable length binary strings". It is being used for various tasks including 3D-design, game strategies and parcitulary efficiently in the area of symbolic regression for financial modelling.

Although GE is very effective in solving many practical problems, its execution time can become a limiting factor for some huge problems, because a lot of candidate solutions must be evaluated.

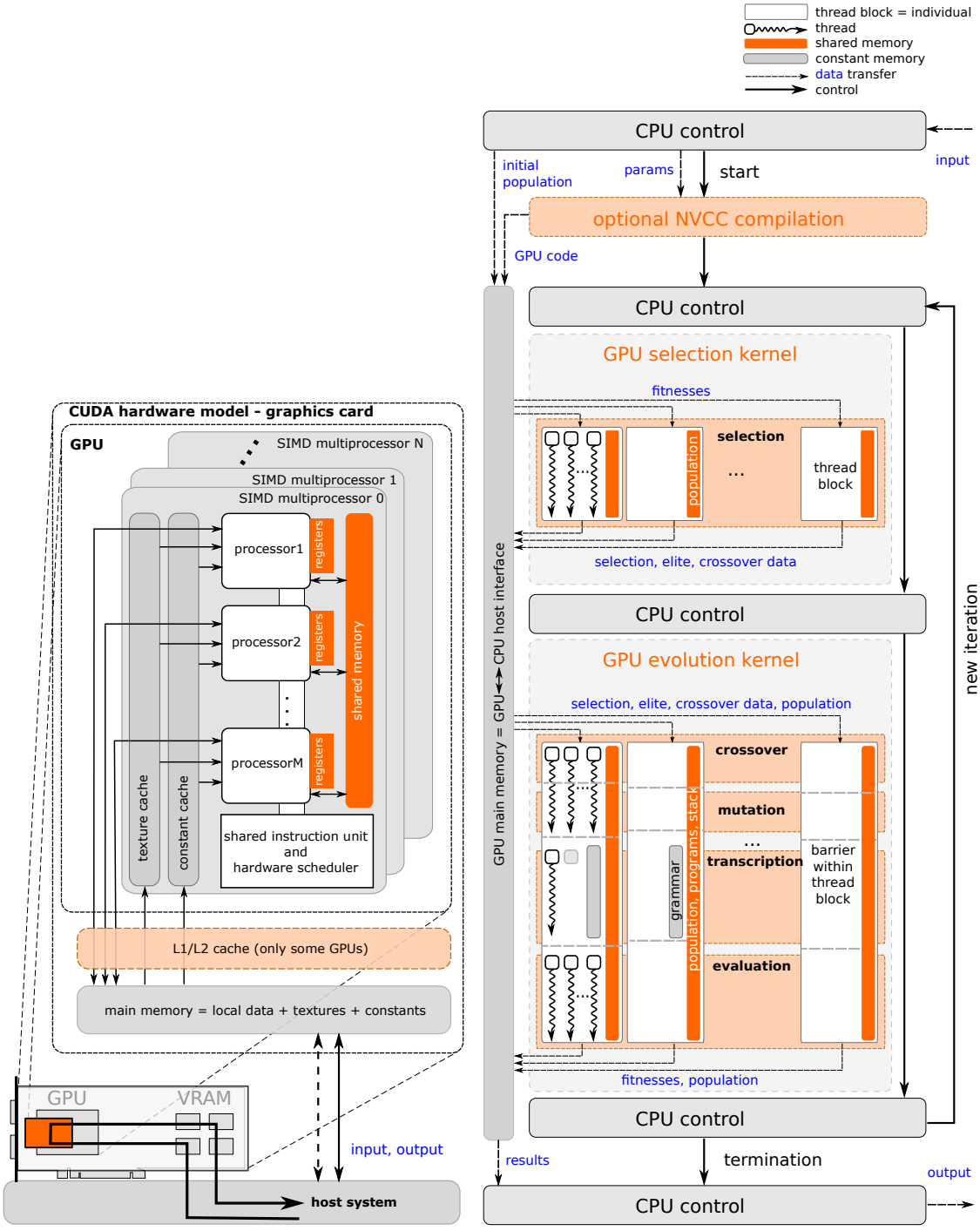
Historically, GPUs were used exclusively for fast rasterization of graphics primitives such as lines, polygons and ellipses. OAs time went, the growing game market and more sophisticated games pushed GPUs to higher functionality and easier programmability. This turned out to be very beneficial, so their 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, which is now often used for General Purpose Computation on the GPUs (GPGPU) with interesting results [1].

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

2 THE PROPOSED SOLUTION

Grammatical evolution consists of several independet steps: selection, crossover, mutation, transcription and evaluation. The most straightforward way how to utilize GPU for acceleration of GE is to

outsource the most time-consuming part, evaluation, to the GPU. This approach has the 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 the GPU every generation, which is a serious performance bottleneck. We have chosen the different approach: running the whole Grammatical Evolution on GPU [1].



(a) CUDA hardware model in general (b) CUDA software model for Grammatical Evolution

Figure 1: The mapping of Grammatical Evolution to CUDA

The CUDA hardware model is shown in Fig. 1(a): the graphics card is divided into the graphics chip (GPU) and the main memory. The main memory, acting as an interface between the host CPU and

the GPU, is connected to the host system using the PCI-Express. This bus has a very high latency and low transfer rates in comparison to inter-GPU memory transfers[1]. The 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 code in parallel. On the other hand CUDA cores 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 read-write L1 cache and some of them have L2 cache as well.

The CUDA software model maps all mentioned GPU features to actual user programs. The programmer's job is to do this mapping properly to fully utilize GPU.

The CUDA software model requires programmer to identify the application parallelism on three levels of abstraction: kernels, thread blocks and threads within these blocks [4]. Kernels are complete programs executed independently on GPU hardware. A GPU hardware scheduler dynamically maps thread blocks to SIMD multiprocessors and individual threads to processors within them during runtime. Because of this and CUDA hardware model, threads should execute the same code over different data and use their shared memory extensively to avoid main memory transactions

The concept of our system is shown on Fig. 1. Application is started on CPU with defined GE parameters and the NVCC compiler is invoked to compile CUDA kernels with defined macros as these parameters. The GPU code compilation and transfer is actually performed only if the GPU has never run program with the same parameters before, as NVCC caches previous programs in GPU. Next, GPU is initialized with a random population and other initial values. After that, 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 evolution. Kernels use shared data pointers to main memory, so 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 main memory transactions as population is kept in the fast-shared memory throughout whole process. Read-only cache is used for grammar data.

As it has been said before, the 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 mode. 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 accesses as well.

3 RESULTS

3.1 TESTING ENVIRONMENT

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

CPU_G is implemented using newest GEVA framework (JAVA language)

GPU is previously described parallel GPU implementation (C language) running on nVidia Geforce GTX 480 GPU running at 1400 Mhz

CPU_C is serial (single-threaded) CPU version of described GPU implementation (C language) where threads as well as thread blocks are simulated using `for` cycles. All source codes were tested on CPU Core i7 at 3.2 GHz.

Table 1: 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 against:**			
CPU_G	0.9×	413.9×	102.8×
CPU_C	0.1×	25.9×	5.3×
***GPU* speedup excluding overhead against:**			
CPU_G	5.4×	636.7×	215.4×
CPU_C	0.8×	39.0×	11.0×
CPU_C speedup against:			
CPU_G	7.2×	32.1×	20.6×

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 selected problem.

We tested optimization using default Grammatical Evolution parameters such as IntFlip mutation with 5% rate, SinglePoint crossover with rate 90%, single elitist individual, tournament selection for $N = 3$, generational replacement policy and 1000 generations. Tested problem was to find function $x + x^2 + x^3 + x^4$ for range $\langle 0; 10 \rangle$ using terminals x,+,-,* and 1. Fitness function is defined as sum of differences between desired and actual solution on regression points: $fitness = \sum_{i=0}^{n} |x[i] - f[i]|$ where $x[i]$ is value of individuals phenotype, $f[i]$ is the value of the desired solution and n is number of regression points.

During performance tests, number of regression points resp. population size varied from 128 to 2560 resp. 2 to 64 so that scalability is tested.

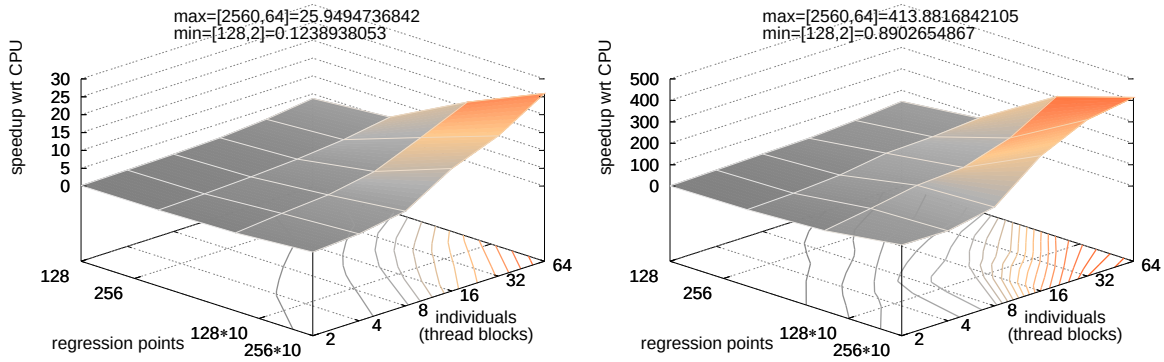
3.2 Convergence

As a convergence test, we measured success rate from 100 runs with random population initialization. Success was defined as fitness of the best individual in last generation reaching value 0 (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 indicate that all algorithms are able to optimize solved problem the same.

3.3 Performance

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. 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 infinite number of generations, times and speedup will be close to measurements excluding overhead.

Each GE implementation have been measured 10 times for all 24 combinations of input parameters. The averaged results rounded to one decimal digit are shown in table 1. As it is evident, execution



(a) CPU_C and *GPU* execution speed comparision

(b) CPU_G and *GPU* execution speed comparision

Figure 2: Speedup comparision including GPU overhead

times higly varies. Obviously worst performance has 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 serial CPU version is faster than *GPU* version in some cases. Put into surface plots shown in Fig. 2, 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 speedup up to 25× (39× for sufficiently difficult problem) compared to CPU_C and several hundred times in comparision with GEVA library. On the other hand, GPU is unsuitable for simple tasks where data transfer and compilation overhead take their toll.

4 CONCLUSIONS

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

Our results indicate that GPU is suitable especially for tasks with larger number of symbolic regression points (1280,2560) evaluated in parallel where it performs up to 636× resp. 39× faster compared to GEVA resp. serial CPU code. This significantly reduces processing time and allows to solve much complex tasks.

ACKNOWLEDGEMENT

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 Grant Agency of Czech Republic, "Security-Oriented Research in Information Technology", MSM 0021630528 (2007-13), the BUT FIT grant FIT-S-10-1, the research plan 0021630528 and with financial support of GA CR 102/09/H042 and FR2983/2011/G1.

REFERENCES

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

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

[3] Grammatical Evolution website: http://www.grammatical-evolution.org

[4] nVidia: CUDA programming guide 3.0.