GPU-based Acceleration of the Genetic Algorithm

Petr Pospichal
Brno University of Technology
Bozetechova 2
612 66 Brno, Czech Republic
xpospi45@stud.fit.vutbr.cz

Jiri Jaros
Brno University of Technology
Bozetechova 2
612 66 Brno, Czech Republic
jarosjir@fit.vutbr.cz

Abstract

This paper presents implementation details of GPU-based genetic algorithm submitted to GPUs for Genetic and Evolutionary Computation competition taking place at GECCO'09.

1 Introduction

Genetic algorithm (GA) is a stochastic optimization method inspired by nature evolution. Because of their parallel nature, they have been parallelized many times.

Graphic Processing Units (GPU) were originally targeted for rasterization of graphics primitives. Today GPU's are more likely fast multicore processors capable of performing complex mathematical tasks.

There are many ways how to exploit GPU's potential for general purpose computation (GPGPU). One option is to employ Compute Unified Device Architecture (CUDA) framework.

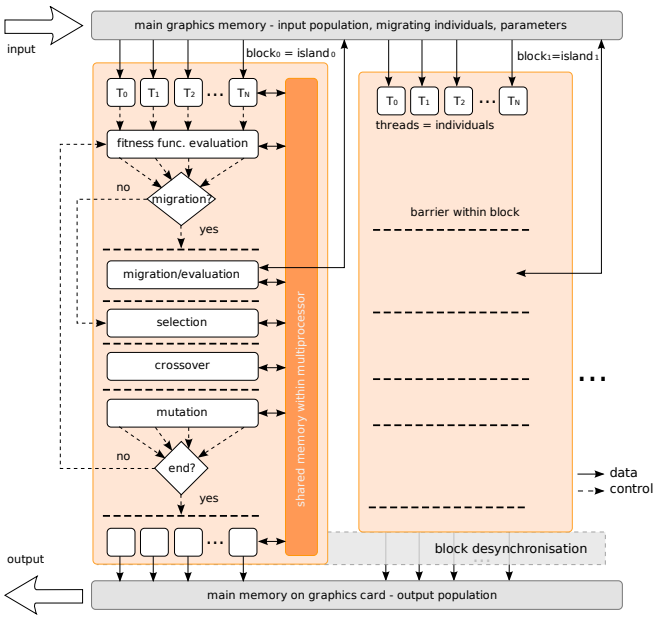

Figure 1: *Mapping of GA to CUDA software model. Every thread controls one individual, every block represents one independent island. The fast shared memory is used for local island population that leads to the computationally intensive execution with high degree of paralelism. Migration is performed through the main (global) memory.*

2 GPU-Based Genetic Algorithm

GPU's are optimised especially for SIMD-type processing with massive paralelism. Hence the proposed optimization tool should exploit a fine-grained island-based GA containing as few branches as possible. Existing CUDA applications also benefit from usage of the fast shared memory within GPU multiprocessors. Data consistency in blocks can be achieved using lightweight barrier `syncthreads()` [2]. Unfortunately, blocks themselves cannot be synchronised easily without performance loss.

Fig. 1 shows the GA mapping to blocks and threads. The entire GA is executed on GPU employing shared memory within processor to maintain the population and achieve the best performance. In addition, asynchronous migration between islands is used (see Fig. 3), which turned out to be efficient way how to improve convergence.

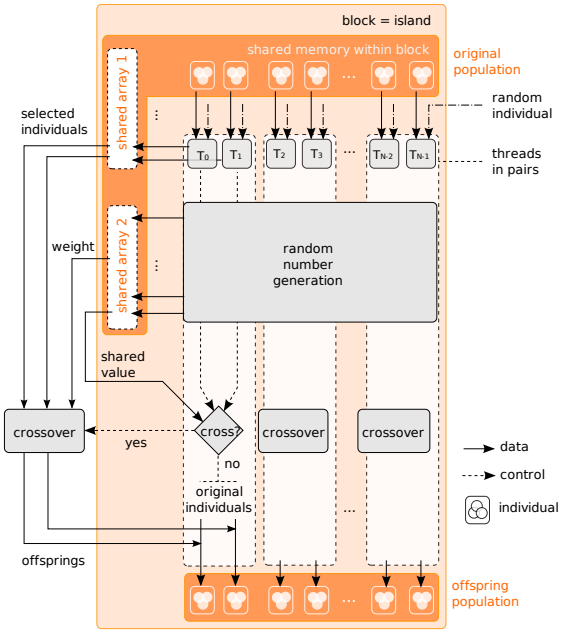

Figure 2: *Scheme of tournament selection with crossover. First, every thread compares its own individual with randomly chosen one, and writes index of better one to the shared memory to notify neighbors of a more suitable partner to crossover. Next, the parallel random numbers generation is performed – first half of generated array is used by thread pairs for decision whether to perform the crossover or not; second half is used as weights for arithmetic crossovers. This wastes the selection in the case that crossover is not finally made, but it is 0.1–2% faster, because of SIMD GPU optimization.*

Tournament selection and arithmetic crossover are tighly connected, as it is evident from Fig. 2, so that limited shared memory is used efficiently. Mutation and fitness evaluation is performed in parallel for each thread (individual).

CUDA currently does not support the function pointers. Moreover, much of the code speed optimization (cycle unrolls to static code, allocation of the shared memory, elimination of some conditions,...) is done by a compiler. Hence, compiler preprocessor

directives are used for the setting of parameters instead of the standard runtime parametrization. That turned out to be highly beneficial for maximum speedup.

Our implementation also utilizes Bitonic-Merge sort algorithm [3], and both uniform and Gaussian PRNG's [1] described in nVidia GPU GEMS book series.

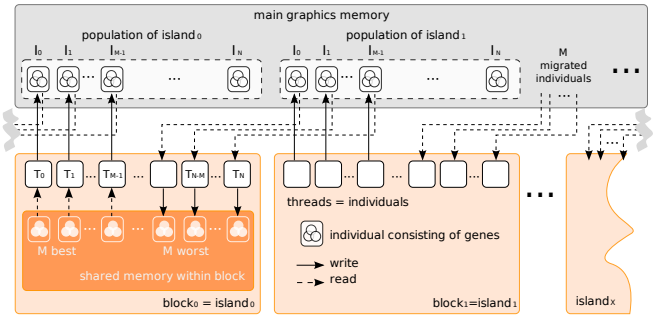

Figure 3: *Scheme of migrations between islands. First, the local island population is sorted according its fitness using Bitonic-Merge sort. Then, M threads write the best chromosomes to the main memory and other M threads read (in parallel) the chromosomes from a neighbour island, overwriting thus M worst individuals.*

3 Results

The speedup of our implementation was investigated using 2-year old consumer-level graphics card GeForce 8800 GTX (128 cores) and a single-threaded, optimised program (approx. 20% faster than GALib) running on the latest processor Core i7 920 at 3.2 GHz. Our innovative mapping of the genetic algorithm to CUDA software model leads to speedups up to 2600 times (see Fig. 4), thus allowing the tasks that took hours to be solved in second-order time. Speed and convergence were tested using Griewank's, Michalewicz's and Rosenbrock's functions. In all cases, 95% of maximum speedup is reached using the population of only 32 individuals per island. The quality of results produced by GPU island is practically the same as for CPU, in addition, the migration significally increases convergence to a global optimum.

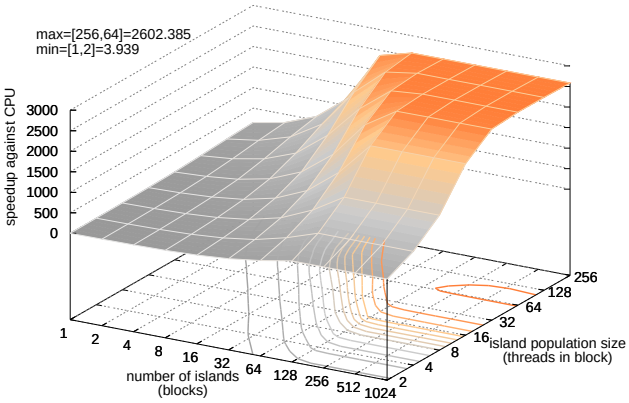

Figure 4: *The GPU speedup on Griewank's function depending on GA parameters. Peak performance is achieved with population sizes from 32 to 256 individuals on 256, 512 and 1024 islands. The islands can be independently simulated on multiple GPU's/graphics cards, the optimization is thereby highly scalable.*

The proposed technique is expected to scale very well to future graphics units and multiple graphics cards allowing furher speedup. GPU's also excel in the computationally-intense applications, i.e. complex problem-fitness functions.

GPU implementation achieves approximately 230-times better power-to-watt ratio then CPU, thus electrical energy is saved during the computation. Furthermore, the graphics card used is hundreds times cheaper than any CPU grid running at the same speed.

Permitting of compiler parameter `-use_fast_math` leads to the significant speedup while maintaing good quality of results.

Table 1: *Overall performance of GA running on CPU and GPU. The performance unit is chosen to be population-size independent IIGG (Island poulation size * #Islands * Genotype length * #Generations) per second. GPU performance highly varies according to degree of paralelism, but it always outperform any CPU, even for only 2 individuals.*

arch.	fitness function	IIGG·10^6 per second
CPU	Rosenbrock's function	5.4 to 5.9
	Michalewicz's function	2.9 to 3.3
	Griewank's function	3.9 to 4.0
GPU	Rosenbrock's function	14.2 to 8877
	Rosenbrock's function – fastmath	18.5 to 11914
	Michalewicz's function	6.9 to 5893
	Michalewicz's function – fastmath	11.7 to 9894
	Griewank's function	9.6 to 7108
	Griewank's function – fastmath	15.9 to 10507

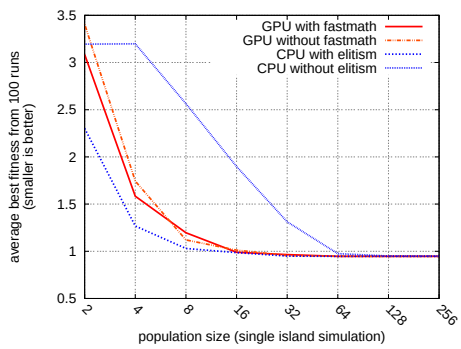

Figure 5: *Comparsion of the results quality after 50 generations on Griewank's function. The GPU results are similar to CPU version with elitism. Compiler parameter -use_fast_math has no negative effect on the quality.*

4 Conclusions

GPU's have proven their abilities for acceleration of genetic algorithms. Not only impressive speedups were achieved, but also high quality solutions were met.

The area of applications is very large – automatic design of new/patentable/innovative solutions, optimisation of design/ production/scheduling, fast solutions to NP problems, search for extremes in complex numerical functions, huge speedup and lower computational costs of existing GA applications and many others.

Our solution can be executed at any nVidia GPU supporting ShaderModel 4.0 and Linux/Windows platform.

References

[1] H. Nguyen. *Gpu gems 3.* Addison-Wesley Professional, 2007.

[2] nVidia Corporation. Nvidia cuda programming guide 2.0.

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