# Implementation of a Simple Genetic Algorithm within the CUDA Architecture

Stefano Debattisti, Nicola Marlat, Luca Mussi, Stefano Cagnoni
Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma
Viale G. Usberti 181a, I-43124 Parma, Italy
{nicola.marlat}{stefano.debattisti}@studenti.unipr.it
{mussi}{cagnoni}@ce.unipr.it

## 1. INTRODUCTION

The increasing interest of researchers in using low cost GPUs for applications requiring intensive parallel computing is due to the ability of these devices to solve parallelizable problems much faster than traditional sequential processors. The first applications of evolutionary algorithms (EAs) on GPUs have been developed to solve specific image processing problems; later, general purpose genetic algorithms (GA) have been implemented. However, those implementations used texture rendering for the encoding and evaluation of individuals, while, most of the times, still executing on the CPU tasks like pseudo-random number generation or selection. This project presents an implementation of a GA within the nVIDIA CUDA™ environment, which avoids the use of textures as data structures and performs all evolution on the GPU, reducing as much as possible the exchange of data with the CPU.

## 2. IMPLEMENTATION

The only process that this project requires to be executed by the CPU is data structure initialization on the GPU device. In particular, such a process allocates memory for the encoded population and initializes it with random bits. The core of this application is therefore virtually fully executed on the device side. We developed one kernel for each of the four typical stages of a GA: selection, crossover, mutation and fitness evaluation. During evolution, they are scheduled one after another, for a number of times equal to the total amount of desired generations, by the CPU, but no data need to be transferred between the two parts. Also pseudo-random numbers are generated GPU side with the Mersenne Twister kernel provided by the CUDA SDK.

While designing the kernels, we followed two main guidelines: maximization of the parallelism and minimization of the accesses to global memory (which is the main bottleneck of the CUDA architecture). At the same time, to optimize also the coalescence for device memory operations, we encoded the population of individuals with a single string of uchar (one byte for each bit of the genome) having care in allocating, in any case, multiples of sixty-four bytes for each individual in order to fulfill the requirements for byte alignment. This results in a waste of memory, but has clear benefits with respect to execution times.

### 2.1 Selection

We opted for a tournament selection strategy. A computational grid divided into block of 32 threads has the duty to compose the mating pool with each thread within a block dealing with the selection of one individual. The tournament size is a variable which must be chosen by the user. The *texture* interface has been exploited to mask fragmented accesses to the vector containing fitness values.

### 2.2 Crossover

Two-point Crossover was preferred to single-point since it offers better performances even if it is computationally more demanding. The grid of the crossover kernel depends on both the population size and the crossover probability, with each block performing the crossover of two individuals. To minimize the number of memory read accesses, each thread manages four bits at a time, accessing the population as a vector of *uchar4*: the size of each block is therefore approximately equal to one quarter of the genome length.

### 2.3 Mutation

We have chosen a classic mutation operator. Again, the population is seen as small blocks of four bits through *uchar4* pointers, each one mutated by a single thread by means of bitwise exclusive OR operations. Scheduling a grid with a sufficient number of 32-thread blocks is needed to mutate the whole population.

### 2.4 Fitness

The problem solved by this demo application is the classic ONEMAX: the fitness function is the sum of the bits of an individual and the goal is to obtain an individual encoded by as many '1' as possible. The kernel is launched with one block per individual: threads within each block read four bits of the genome at one time (once again through *uchar4* pointers) and load their sum into a vector in shared memory. The final value of each individual's fitness is then computed by means of a parallel reduction.

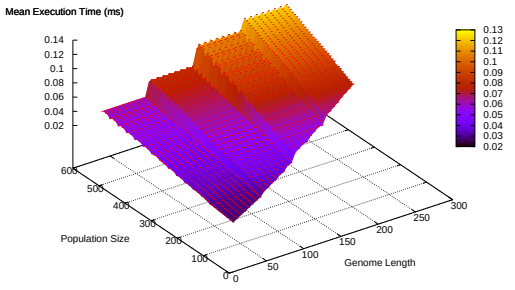


**Figure 1: GAGPU: Average time per generation.**

## 3. EXPERIMENTAL RESULTS

Experiments were run on a PC equipped with an Intel Core2Duo processor running at 1.86GHz with a GeForce 8800 GT video card from nVIDIA corporation equipped with 1Gb of video RAM. As a first set of results we report the

average execution time of GAGPU. In figure 1 we plot the average execution time for one GAGPU generation vs. both population size and genome length. Note that on our card, which is equipped with Streaming Multiprocessors (SMs) of computing capability 1.1, the design of our kernels limits these sizes to 512 and 256 respectively. As can be seen, execution time scale linearly with the number of individuals (with a low derivative), while step discontinuities appear with the growth of the genome size. Every 64 bits (recall that the actual size of an individual is always a multiple of 64 bytes) the load of the GPU (measured in number of blocks and number of threads per block to be executed) has a sharp increment. Anyhow, using a device having a higher number of available SMs (14 in our card) should flatten these steps with a consequent improvement on execution times.

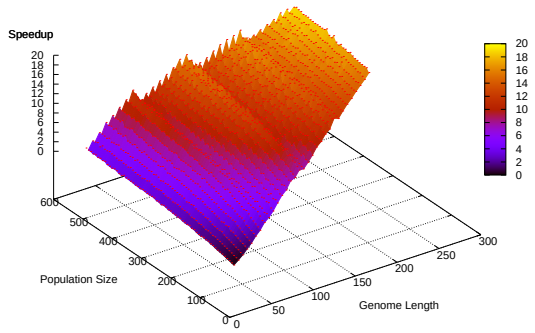


**Figure 2: GAGPU vs TinyGA: Speedup**

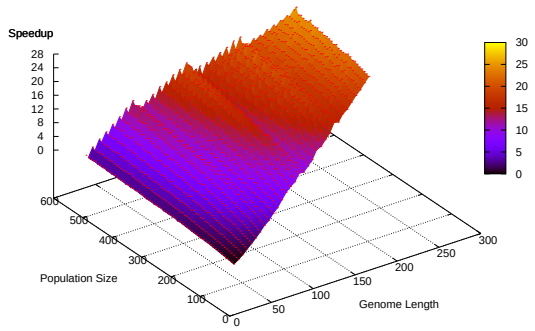


**Figure 3: GAGPU vs sequentialGAGPU: Speedup**

In a second test, the above performances were compared to those obtained with TinyGA, a sequential GA implementation by Riccardo Poli (which won the GECCO TinyGA competition in 2006) that can be considered as a lower limit, as concerns GA computational requirements on sequential architectures. Figure 2 shows the speedup achieved with respect to TinyGA: for small search space dimensions and population sizes, the two algorithms are nearly equivalent, but the more the problem dimension and the population increase, the more GAGPU gains advantage. The maximum speedup rate was 19 at population size 512 and genome length 256. Discontinuities in figure 1 are obviously present also in the speedup graph, which has a periodic trend. Instead, in figure 3 it is possible to see the speedup achieved by a well-coded fully equivalent sequential version of our GAGPU: in this case the speedup is close to 26 times.

**Table 1: Cuda Profiler Results**

| kernel | usec | ldu | ldc | stu | stc | it |
|---|---|---|---|---|---|---|
| *RGPU* | 1402 | 2355 | 0 | 0 | 117760 | 0.56 |
| *mut* | 6.86 | 0 | 219 | 0 | 292 | 0.2 |
| *fit* | 5.58 | 0 | 73 | 68 | 10 | 0.46 |
| *cpySel* | 4.95 | 34 | 75 | 0 | 293 | 0.27 |
| *xover* | 3.70 | 0 | 46 | 0 | 147 | 0.29 |
| *sel* | 3.05 | 0 | 4 | 0 | 8 | 0.02 |
| *stats* | 3.84 | 0 | 0 | 0 | 0 | 0 |

In table 1 we report some data about our GAGPU obtained by the CUDA Profiler. In particular, columns *ldu/ldc* and *stu/stc* highlight that there are no global uncoalesced read-write operations. Actually, there are just few in the *cpySel* and *fit* lines: this is due to the first thread in each block which reads/writes from/to an un-aligned vector. This does not slow the system at all, because there are no multiple operation to be serialized. It is worth noticing that the average execution time of each kernel is perfectly comparable with the one of a device-to-device *memcpy* operation of the same amount of processed data. This confirms that the computational intensiveness of a GA is not that high to fully exploit the power of a GPU (see the "it" column of table 1).

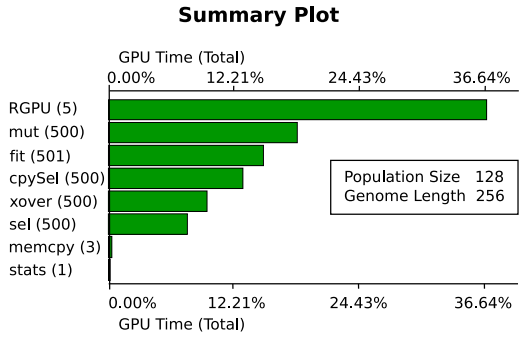


**Figure 4: GAGPU: kernels' occupancy**

The last figure 4 also reports data from the profiler. It shows, in decreasing order, the percentage of GPU occupation of all kernels. One can notice that they have very similar GPU occupation (a relevant slice of the cake is occupied by the pseudo-random number generator). As a consequence, no kernels slow down the system more than others. This is reflected in a good execution time scaling with problem dimension and population size (see figure 1).

## 4. CONCLUSIONS

All the results reported show the efficiency of our GAGPU as compared to a well-coded sequential version. A comparison with a generic GA library, such as GAlib, would result a much greater speedup. If we consider that the ONEMAX objective function utilized in our experiments is one of the less computationally intensive, it is reasonable to expect even better results on problems with complex parallelizable fitness functions.

Finally, we want to highlight that our system is versatile and modular and, even in its present version, can be general enough to be used to optimize a vast classes of problems, provided a proper fitness function kernel is developed. As well, it can be easily extended introducing other operators or fitness functions, as the different steps of evolution are split into independent kernels. We plan to introduce this enhancements in the near future, to make GAGPU a complete and extremely efficient environment for GA applications.