Particle Swarm Optimization within the CUDA Architecture

Luca Mussi, Stefano Cagnoni
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Viale G. Usberti 181a, I-43124 Parma, Italy
{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; at the beginning they were using textures rendering for the encoding and evaluation of individuals and most of the times tasks like pseudo random numbers generation and other evolutionary operations were executed on CPU. This project presents an approach for the implementation of PSO algoritms on GPUs which, by means of the nVIDIA CUDA™ environment, 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

In this work we chose to implement the 2006 standard PSO on GPU: its main peculiarity stands in the use of a random neighborhood topology for the determination of particles' local best. The rationale behind this choice is the absolute inadequacy of a PSO with synchronous bests update (as it would be for a CUDAPSO) and global best topology for the optimization of multi-modal problems.

To effectively fully exploit the impressive computation capabilities offered by CUDA to develop a parallel PSO, the best approach is probably to think about the main phases of the algorithm as separate tasks, each one to be parallelized separately: this way each phase can be accomplished by means of a different kernel and the whole optimization process can be achieved by scheduling many times in a row all the basic kernels needed to perform one generational update of the swarm.

Since the only way CUDA offers to share data among different kernels is to keep it in global memory, the current status of our PSO must be saved there: one of the first problems to takle is therefore how to organize the PSO data to exploit the GPU read/write coalescing capability.

Figure 1 shows the structures we adopted for the arrays involved in our design. For instance, in the array which stores the current "positions" the D elements used to encode one particle are always a multiple of sixteen. This way, one block of threads that needs to load the current position of a particle (i.e. for fitness evaluation) can always perform a unique coalesced read operation which satisfy the requirements for byte alignment (since we use the *floats*, sixteen elements correspond to sixty-four bytes). Additionally, this organization permits to run several swarms at the same time simply by playing with the threads' indexes.

As for the generation of pseudo-random numbers on GPU, we use the Mersenne Twister kernel provided within the CUDA SDK. A brief description of the others kernels follows.

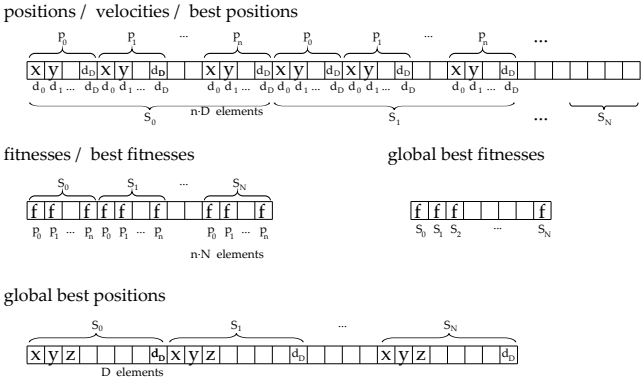

Figure 1: CUDAPSO: Data Organization.

2.1 Bests Update

For each running swarm a thread block is scheduled with a number of threads equal to the number of particles in the swarm. Firstly, each thread loads in shared memory both the current and the best fitness values of its corresponding particle. Then, the need to update the best fitness value is checked and, when this is the case, a flag for the particle's best position to be updated is set. Successively the current best fitness value in the swarm, determined by means of parallel reduction, is checked against the best value found so far (to be possibly updated) and, in case there is no improvement, the random topology of the swarm is re-initialized. Before the end, each thread/particle scans the neighborhood map (implemented with a bitwise-matrix) and controls the fitness values of the other particles to find its local best.

2.2 Position Update

A computational grid divided into block of 32 threads has the duty to update the position of all particles being simulated. Each thread manages the update of one element of the positions and velocities arrays, irrespective of to which particle (or to which dimension) it corresponds. At the beginning the current position, best position, velocity and best-position-update flag are loaded. In case the update flag results set, the best position is updated and successively the classical PSO equations are applied to the loaded values. At the end all updated values are stored back to global memory to be used by the fitness evaluation kernel.

2.3 Fitnesses Evaluation

The fitness function we implemented for this work is the generalized Rastrigin's function which is characterized by a pretty high computational complexity and, at the same time, by a high degree of parallelization. This kernel is scheduled with a computational grid composed by one block for each

particle being simulated (irrespective of to which swarm it belongs), each block being composed by a number of threads equal to the problem dimensions. Each thread loads one coordinate of the considered particle and calculates one partial addend of the Rastrigin's formula. The final sum is computed, once again, by means of parallel reduction.

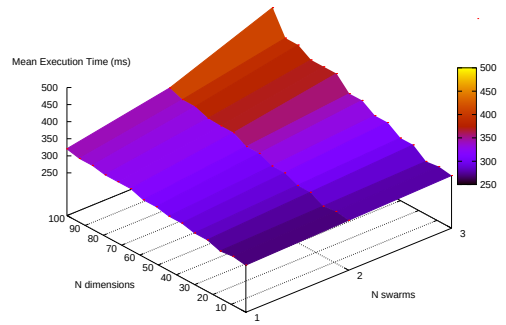

Figure 2: CUDAPSO: Mean execution time optimizing the generalized Rastrigin's function for 10000 generations. Times are in milliseconds.

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, in figure 2 we report the average execution time of CUDAPSO vs. both the number of swarms and the dimensions of the problem. As it can be seen, times scale linearly with the dimensions of the problem. The fact that, with only one swarm, the derivative is close to zero highlights the good parallel design of the whole system. Anyhow, using a device having a higher number of available SMs (14 in our card) should further flatten this graph with a consequent further improvement of execution times.

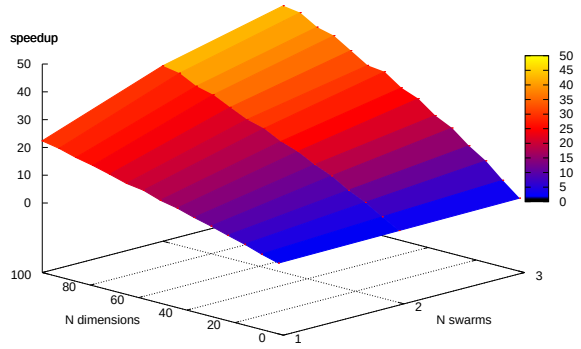

Figure 3: CUDAPSO vs sequentialPSO: Speedup

In a second test, the above performances were compared to those obtained with the original 2006 Standard PSO. Figure 3 shows the speedup achieved with its respect: even if, for small problem dimensions, the sequential version is twice as fast as the parallel one, the more the problem dimensions increase, the faster CUDAPSO runs with respect to the reference. In particular the achieved speedup running one swarm was of about 22 times (with Ndimensions = 100), while it was close to 50 times running three swarm at the same time.

Table 1: Cuda Profiler Results

kernel	usec	ldu	ldc	stu	stc	it
bestUp	10	0	1.57	0	4.57	0.01
posUp	2.95	8.57	23.42	0	36.6	0.11
fit	2.32	0	4.57	8.57	1.14	0.10
RGPU	4346	2560	0	0	400000	0.51
fBestUp	9.53	0	0	0	0	0
lBestUp	3.1	0	0	0	0	0
initPar	2.02	0	4	0	0	0

In table 1 we report some data about our CUDAPSO obtained by the CUDA Profiler. In particular, columns *ldu*, *ldc*, *stu* and *stc* highlight that there are no global uncoalesced read-write operations. Actually there are just few in the *posUp* 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. Looking at the "it" column of table 1 we can also note that the computational intensiveness of the PSO is not that high to saturate the resource usage of a GPU.

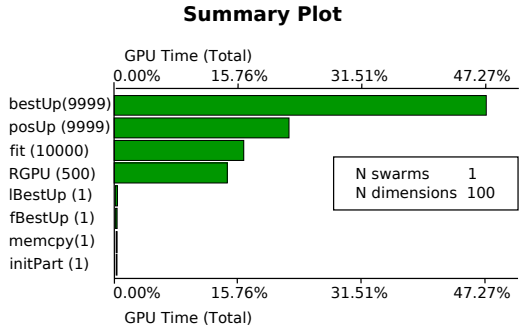

Figure 4: CUDAPSO: kernels' occupancy

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 the pseudo random numbers generation, being the most arithmetic intensive operation, takes a significant percentage of the time. Anyway, the *bestsUpdate* is the more demanding kernel, probably due to the implementation of the random topology as in 2006 standard PSO. As a matter of facts, the adoption of a global best topology would lead to better load balance between kernels and to an overall faster execution, but, at the same time, the algorithm effectiveness in solving multi-modal problems would certainly be affected.

4. CONCLUSIONS

All the results prove that our CUDAPSO is much more efficient compared to a well coded serial version. The possibility to run efficiently many swarms at the same time is another advantage that permits to improve the optimization success rate, compared to a single serial run with comparable execution time. Without any doubt, speedup times could be further improved on problems with complex fitness functions which permit a high degree of parallelization.

Finally, we want to remark that the proposed design is modular and versatile: evolutionary steps are split into independent kernels, which makes it easy to adapt them to different variants of the PSO algorithm. For example, the neighborhood topology could be changed by changing the *bestsUpdate* kernel. More simply, our CUDAPSO can be adapted to the optimization of other goals by changing the fitness evaluation kernel. These improvements will be part of our future work.