CUDA and OpenCL-based asynchronous PSO

Youssef S. G. Nashed, Alessandro Bacchini, and Stefano Cagnoni
Dept. of Information Engineering
University of Parma - Italy
{nashed,cagnoni}@ce.unipr.it
alessandro.bacchini@studenti.unipr.it

Luca Mussi
Henesis s.r.l.
Dept. of Information Engineering
University of Parma - Italy
luca.mussi@henesis.eu

1. GPU-BASED PSO PARALLELIZATION

In 'synchronous' PSO, positions and velocities of all particles are updated in turn in each 'generation', after which each particle's new fitness is evaluated. The value of the social attractor is only updated at the end of each generation, when the fitness values of all particles are known.

The 'asynchronous' version of PSO, instead, allows the social attractors to be updated immediately after evaluating each particle's fitness, which causes the swarm to move more promptly towards newly-found optima. In asynchronous PSO, the velocity and position update equations can be applied to any particle at any time, in no specific order.

The most common GPU implementations of PSO assign one thread per particle and do not take full advantage of the GPU power in evaluating the fitness function in parallel. Parallelization only occurs on the number of particles of a swarm and ignores the dimensions of the function.

In our parallel implementations: (i) we designed the thread parallelization to be as fine-grained as possible, considering that, in PSO, velocity and position update occur independently over each dimension; (ii) we implemented an 'asynchronous' PSO which, despite updating all particles in parallel, allows each of them to update the social attractor without waiting for all other particles' fitness values to be evaluated. A block diagram representing the GPU execution of our parallel asynchronous PSO is shown in Figure 1.

1.1 Synchronous CUDA-PSO

The synchronous implementation [4] we previously developed using CUDA comprises three stages (kernels), namely: positions update, fitness evaluation, and bests update, implemented as three CUDA kernels which must be executed sequentially; synchronization occurs at the end of each kernel run. While allowing for virtually any swarm size, this implementation requires synchronization points where all the particles' data has to be saved into slow global memory to be read by the next kernel. This frequent access to global memory has been the main justification behind the asynchronous implementation.

1.2 Asynchronous CUDA-PSO

Like the synchronous version, the asynchronous PSO allocates a thread block per particle with a thread per problem dimension, but it is implemented as just one kernel. Removing the synchronization constraint allows each particle to keep all its data in fast-access registers and shared memory, and removes the need to share data in global memory within a generation. In practice, every particle checks its neighbors' personal best fitnesses, then updates its own personal best in global memory only if it is better than the previously

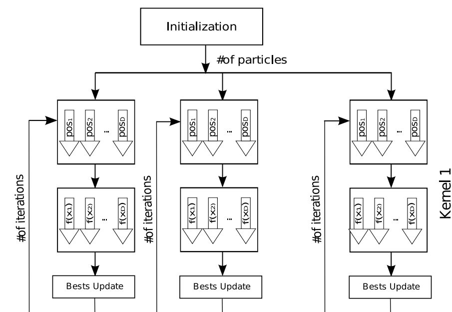

Figure 1: Asynchronous CUDA-PSO: particles run in parallel independently. Blocks represent particles and thick white arrows represent threads for each dimension of the search space.

found personal best fitness. This speeds up execution time dramatically, particularly when the fitness function itself is highly parallelizable. The price to be paid is a limitation in the number of particles in a swarm which must match the maximum number of thread blocks that a certain GPU can execute in parallel. Using different mid-priced graphics cards we could run swarms of up to 27 or 32 particles.

A fully-detailed description of our asynchronous parallel PSO implementation is given in [5].

1.3 OpenCL-PSO

The OpenCL version is substantially identical to the CUDA-based asynchronous implementation. An effective random number generator[1] is embedded in the code, since OpenCL does not provide any. The same algorithm was also used in the CUDA versions since it is simpler and faster than the one provided by CUDA. Comparison of results obtained with the different random functions showed that the limited length of the sequence generated by the one we used (2^{63}) does not affect performances significantly.

2. RESULTS

We compared the different versions of our parallel PSO implementation and the sequential SPSO on a 'classical'

[1] `http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-mwc64x.html`

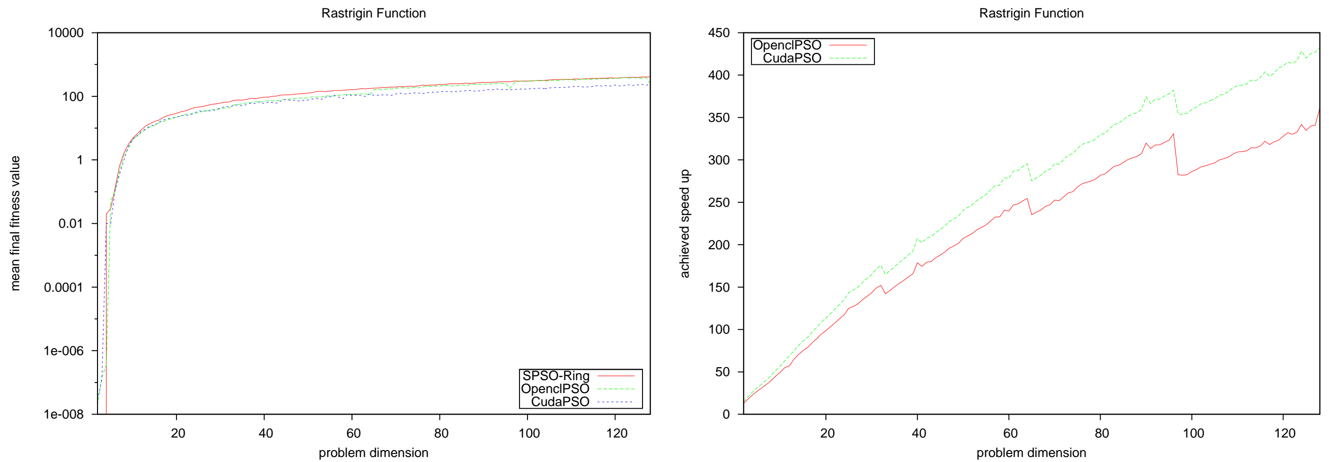

Figure 2: Comparison of sequential and asynchronous GPU-based PSO: final fitness values (left) and speed-up (right) vs. problem dimensions on the Rastrigin function running 32 particles for 10000 generations.

benchmark which comprised a set of functions which are often used to evaluate stochastic optimization algorithms. The different parallel PSO implementations were evaluated in terms of speed, while also checking that the quality of results had not been affected. All algorithm parameters were the same in all tests, set to the 'standard' values suggested in [2]: $w = 0.729844$ and $C_1 = C_2 = 1.49618$. Also, we adapted the sequential SPSO by substituting its original stochastic-star topology with the same ring topology adopted in the parallel versions and downgraded it to 'float' precision to match the GPU-based algorithms' precision.

Our parallel algorithms were developed using CUDA version 3.2 [6] and OpenCL 1.0 [3].

The following implementations of PSO have been compared: (1) the sequential SPSO version modified as described above; (2) *CUDA-PSO* implemented asynchronously with only 1 kernel; (3) CUDA-OpenCL following the same implementation as *CUDA-PSO*. Values were averaged over 100 runs for all values of problem dimension D between 2 and 128 running swarms of 32 particles for 10000 iterations.

We used the following test functions which can be found in the SPSO package [2] and in the Black Box Optimization Benchmark suite [1]: (a) Sphere, evaluated within the domain $[-100, 100]^D$, (b) High Conditioned Elliptic function ($[-100, 100]^D$), (c) Rastrigin ($[-5.12, 5.12]^D$), (d) Rosenbrock ($[-30, 30]^D$), and (e) Griewank ($[-600, 600]^D$).

The asynchronous version of our algorithm was able to significantly reduce execution time with respect not only to the sequential version but also to our previously-developed parallel versions. Depending on the degree of parallelization allowed by the fitness functions we considered, the asynchronous version of CUDA-PSO could reach speed-ups ranging from 50 (Rosenbrock) to over 400 (Griewank, Rastrigin) with respect to the sequential implementation, and often of more than one order of magnitude with respect to the corresponding GPU-based 3-kernel synchronous version.

Figure 2 compares average final fitness values and speed-ups obtained on the Rastrigin function: the GPU-based and the sequential versions produced very similar average best fitness values which shows that the speed-ups were not obtained at the expense of accuracy.

Regarding speed-ups, the best performances were obtained on the most complex functions. This is probably due to the fact that GPUs have internal *fast math* functions which can provide good computation speed at the cost of slightly lower accuracy (the error caused is of the same order of the mantissa's LSB), which causes no problems for most scientific problems.

As well the limit in the swarm size is not such a relevant shortcoming, as the number of particles needed by PSO for most applications is much lower than the population size usually required by evolutionary algorithms to solve high-dimensional problems. This makes the availability of swarms of virtually unlimited size less appealing than could appear at first sight. One should also consider that GPUs are being equipped with more processing cores with the introduction of every new model.

3. ACKNOWLEDGMENTS

Youssef S. G. Nashed is supported by the European Commission MIBISOC grant (Marie Curie Initial Training Network, FP7 PEOPLE-ITN-2008, GA n. 238819).

4. REFERENCES

[1] N. Hansen, A. Auger, R. Ros, S. Finck, and P. Pošík. Comparing results of 31 algorithms from the Black-Box Optimization Benchmarking BBOB-2009. In *Proc. GECCO2010 companion*, pages 1689–1696, New York, NY, USA,2010. ACM.

[2] J. Kennedy and M. Clerc, 2 2006. http://www.particleswarm.info/Standard_PSO_2006.c.

[3] Khronos Group. *The OpenCL Specification*, 1.0 edition, October 2009.

[4] L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences*, 2010, in press.

[5] L. Mussi, Y.S.G. Nashed, and S. Cagnoni. GPU-based Asynchronous Particle Swarm Optimization. *Proc. GECCO 2011*, 2011.

[6] nVIDIA Corporation. *nVIDIA CUDA programming guide v. 3.2*, October 2010.