GPU-Based Asynchronous Global Optimization with Particle Swarm

This content has been downloaded from IOPscience. Please scroll down to see the full text.


(http://iopscience.iop.org/1742-6596/385/1/012012)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 52.27.129.95
This content was downloaded on 13/07/2015 at 21:35

Please note that terms and conditions apply.
GPU-Based Asynchronous Global Optimization with Particle Swarm

M P Wachowiak and A E Lambe Foster
Department of Computer Science and Mathematics
Nipissing University, North Bay, ON, Canada, P1B 8L7

E-mail: markw@nipissingu.ca

Abstract. The recent upsurge in research into general-purpose applications for graphics processing units (GPUs) has made low cost high-performance computing increasingly more accessible. Many global optimization algorithms that have previously benefited from parallel computation are now poised to take advantage of general-purpose GPU computing as well. In this paper, a global parallel asynchronous particle swarm optimization (PSO) approach is employed to solve three relatively complex, realistic parameter estimation problems in which each processor performs significant computation. Although PSO is readily parallelizable, memory bandwidth limitations with GPUs must be addressed, which is accomplished by minimizing communication among individual population members though asynchronous operations. The effect of asynchronous PSO on robustness and efficiency is assessed as a function of problem and population size. Experiments were performed with different population sizes on NVIDIA GPUs and on single-core CPUs. Results for successful trials exhibit marked speedup increases with the population size, indicating that more particles may be used to improve algorithm robustness while maintaining nearly constant time. This work also suggests that asynchronous operations on the GPU may be viable in stochastic population-based algorithms to increase efficiency without sacrificing the quality of the solutions.

1. Introduction
Particle Swarm Optimization (PSO) is a relatively new stochastic global optimization algorithm that is used in a variety of problems, including model calibration and parameter estimation [1–6]. Similar to genetic algorithms and evolutionary strategies, PSO is an iterative, stochastic population-based technique. However, in contrast to genetic algorithms, which model the competitive characteristics of biological evolution, PSO simulates cooperative and social behaviour, such as fish schooling, birds flocking, or insects swarming. A diffuse population of P individuals, now termed as particles, explores the search space, gradually forming smaller swarms in optimal regions. PSO has been successful in many applications, including difficult simulation-based optimization problems [7]. Investigation into the theoretical and convergence properties of PSO is an active, on-going research pursuit [8–10].

PSO offers many advantages. It is relatively easy to conceptualize and to implement, requires few parameters to tune, and, in many cases, demonstrates robust convergence to high-quality solutions. Numerous adaptations and enhancements have been made to PSO, many of which were designed to solve specific optimization problems, or to address shortcomings to which PSO is susceptible, such as...
slow convergence and possible entrapment in local optima [8–10]. A particularly useful benefit is that PSO exhibits a high degree of data-level (fine-grained) and task-level (coarse-grained) parallelism. It is therefore conducive to implementation on high-performance computing hardware, and many investigators have explored the potential of parallel PSO on both distributed-memory clusters and on shared-memory supercomputers [11–13].

Within the last ten years, novel parallel architectures have been gaining prominence in high-performance computing. Because processor performance cannot continue to improve indefinitely (due to the inability to reduce the operating voltage of chips as sharply as in the past with increasing numbers of transistors), performance gains are increasingly spearheaded by multicore, graphics processing units (GPUs), and hybrid technologies. GPUs are highly parallel processors that were initially designed to render graphics through a fixed function graphics pipeline. The power of general purpose GPU computation became evident in the mid 2000s, and, consequently, frameworks for GPU programming have been developed, each with unique benefits. Popular GPU environments include CUDA (Compute Unified Device Architecture) by NVIDIA [14], BROOK+ by AMD, and OpenCL by the Khronos group. OpenCL is an open standard for parallel programming of heterogeneous systems that can be used to program the CPU, GPU, Cell architecture, clusters, and even mobile devices, and which supports both data and task-based parallel programming models [15].

In this paper, an asynchronous GPU-based approach to parameter estimation using PSO is presented. Although individual particles simultaneously explore many different regions of the search space (data parallel/fine-grained), because of the complexity of the cost function, and because the cost function often requires observational data, each particle performs significant computation (task parallel/coarse-grained). Consequently, the approach presented here can be termed “medium-grained”, where GPU-specific memory bandwidth issues in CPU-GPU communication are addressed. The approach taken in this paper is similar to that taken by other investigators working in parallel PSO on GPUs [16–17]. However, these previous investigations mostly report results on standard global optimization test functions (although some have quite high dimensionality), whereas the current study considers relatively realistic, compute- and data-intensive problems.

2. The Particle Swarm Algorithm

At each PSO iteration, a $d$-dimensional particle $\mathbf{p}_i$, $i = 1, \ldots, P$, representing a point in the search space, evaluates the cost function $f(\mathbf{p}_i)$. The particle then probes the search space by modifying its position according to a velocity vector, which is a function of the best position found by that particle – its personal best ($\mathbf{p}_i^{\text{best}}$) – and of the best position – the global best ($\mathbf{g}^{\text{best}}$) – found so far among all particles. In this way, particles tend to congregate (swarm) around good positions. As PSO is stochastic, particles may move on a trajectory in which a better response is encountered. The other particles subsequently adjust their movements to swarm around the better position. A particle’s movement is influenced not only by the best global position, but also by its personal best position and by stochastic perturbations. Of the few parameters that PSO does require, the inertial weight $w$ indicates the relative influence of the velocity in the previous iteration and determines the degree to which the particle should move in the same direction as in the last iteration. It can be adjusted adaptively depending upon the progress of the search. This parameter is very important because it regulates the balance between global expansion (large $w$ values) and local refinement [6, 18]. Two other parameters, $C_1$ (self) and $C_2$ (swarm) are not critical for the operation of PSO, but tuning these parameters may result in better convergence and avoidance of local optima.

With these parameters, the basic velocity update equation for particle $i$ is given as:

$$
\mathbf{v}_i(t+1) = w(t)\mathbf{v}_i(t) + C_1\phi_1(\mathbf{p}_i^{\text{best}}(t) - \mathbf{x}_i(t)) + C_2\phi_2(\mathbf{g}^{\text{best}}(t) - \mathbf{x}_i(t))
$$

(1)
where $v_i(t)$ is the velocity of a particle $i$ at time $t$, $v_i(t+1)$ is the velocity in the next iteration $t+1$, $p_i^{\text{best}}$ denotes the best position of $p_i$ thus far in the search, $x_i(t)$ denotes the particle’s current position at time $t$, and random numbers $\varphi_1, \varphi_2 \sim U(0, 1)$ add stochasticity to the update and diversity to the swarm population. After the new velocity is computed, the position of the $i$-th particle is updated as:

$$x_i(t+1) = x_i(t) + v_i(t+1).$$

(2)

3. Methods

3.1. Parallel implementation of PSO

Investigators have previously reported success in exploiting the inherent parallelism of PSO on GPU architectures [16–17]. In this paper, a similar “medium-grained” parallel approach was taken wherein each particle is represented by an individual thread running on the GPU. Separate implementations were developed for single-thread CPU and parallel GPU execution. The CPU and GPU codes are nearly identical, but each particle is given its own individual thread in the GPU implementation.

3.2. Asynchronicity in the GPU implementation

To reduce communication time between GPU cores, asynchronicity was introduced into PSO. Asynchronous parallel PSO has been explored by other investigators [12–13]. In this method, the GPU cores run entirely independently for a specified number of iterations, after which they are re-synchronized. As reported below, it was observed that asynchronicity does not have an adverse effect on the quality of the solutions.

3.3. Adaptive inertial constant

A standard PSO modification allows the inertial parameter of the velocity function $w$ to be modified based upon the number of iterations. In the current work, $w$ was adjusted based on the population’s performance. If stagnation is detected (50 iterations have been expended with no improvement in the global best value), the inertial parameter is either increased (to encourage more global exploration) or decreased (to refine existing good solutions). This adjustment requires additional computation time, but has been shown to avoid entrapment in local minima [6, 18].

3.4. CUDA implementation

CUDA is a framework for GPU programming on NVIDIA graphics cards that supports C-like language constructs. It contains functions to simplify the use of the GPU for general purpose processing. At the lowest level, parallelization is achieved via threads, which are analogous to threads in serial computing. GPU programs, called kernels, run in these threads. Blocks, at the next higher level, are simply large arrays of threads. At the highest level, grids are arrays of blocks. CUDA employs symmetric processing: thread tasks are determined according to the block and grid in which the threads reside, as well as their position in these blocks and grids. CUDA also simplifies the transfer of information between RAM and the GPU-specific RAM.

The basic schema for CUDA parallelization consists of three general steps. First, the data to be processed are transferred from the CPU to the GPU RAM. Next, the kernel is run to carry out the processing using the thread number to determine which segment of the data is to be processed by that thread. Finally, the processed data are copied back to the CPU for use and further processing. Additional details of the GPU implementation of PSO are found in [17].

3.5. Cost functions

Three cost functions, representing realistic problems, were explored. Ground truth values for these functions were known.

3.5.1. Toy protein folding. In the toy protein folding problem, intra-molecular potential energy $\Phi$ is to be minimized [19]. Each monomer (amino acid) is connected to the next monomer in a
protein chain through a bond angle $\theta \in (-\pi/2, \pi/2]$. For a protein of $n$ amino acids, there are $d = n - 2$ such angles. The “toy” model is a simplification of the real protein folding problem in that all monomers are constrained to be co-planar, the bond distances are equal, and only two monomers, designated as A and B, are used. Monomer pairs are restricted to AA, AB/BA, and BB. PSO for this model has been studied by other investigators [20]; however, to the authors’ knowledge, the work presented here is the first GPU implementation. The cost function is given as a function of $\Theta$:

$$\Phi(\Theta) = \sum_{i=2}^{d} V_{i}(\theta_{i}) + \sum_{i=1}^{2} \sum_{j=i+2}^{d} V_{ij}(\xi_{i}, \xi_{j}),$$

where

$$r_{ij} = \left[1 + \left(\sum_{k=1}^{i} \cos \left(\sum_{l=1}^{k} \theta_{l}\right)\right)^{2} + \left(\sum_{k=1}^{i} \sin \left(\sum_{l=1}^{k} \theta_{l}\right)\right)^{2}\right]^{1/2},$$

$$V_{i}(\theta_{i}) = \frac{1}{4} \left(1 - \cos \theta_{i}\right),$$

$$V_{ij}(\xi_{i}, \xi_{j}) = 4\left[\frac{1}{r_{ij}} - C(\xi_{i}, \xi_{j})\right].$$

and

$$C(\xi_{i}, \xi_{j}) = \left\{ \begin{array}{ll}
1 & \text{for AA, BB, AB pairs.}
\end{array} \right.$$
3.5.3. Disequilibrium problem. The disequilibrium problem in econometrics concerns determining the supply and demand components of a time series of transacted quantities \( Q_t \). This problem has been studied in the context of global optimization methods [6, 22]. For time period \( t \), the quantity transacted \( Q_t \), supply \( S_t \), and demand \( D_t \) are related by:

\[
D_t = \alpha_0 + \alpha_1 X_{1t} + \alpha_2 P_{t-1} + u_t, \quad S_t = \beta_0 + \beta_1 X_{2t} + \beta_2 P_{t-1} + v_t, \quad Q_t = \min(D_t, S_t).
\]

\( P_t \) is the price in period \( t \), \( X_{1t} \) and \( X_{2t} \) are explanatory variables for period \( t \), and \( u_t \sim N(0, \sigma_1) \), and \( v_t \sim N(0, \sigma_2) \). The goal is to determine \( \theta = [\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1, \beta_2, \sigma_1, \sigma_2] \) by maximizing the log likelihood:

\[
L(\theta) = \sum_{i=1}^{T} \log(f_{i1}f_{i2}^2 + f_{i2}^2f_{iu})
\]

where

\[
f_{iu} = \frac{1}{\sqrt{2\pi}\sigma_i} \exp(-h_{iu}^2/2), \quad f_{i2} = \frac{1}{\sqrt{2\pi}\sigma_2} \exp(-h_{i2}^2/2),
\]

\[
F_{ij} = \left[ \frac{1}{h_i\sqrt{2\pi}} \exp(-u^2/2) \right]_{u=0}^{u=-h_i} = \left[ \frac{1}{2(1-\text{erf}(h_i/\sqrt{2}))} \right]_{i=1,2}
\]

\[
h_{iu} = \frac{Q_i - X_{1i}\beta_1}{\sigma_1}, \quad h_{i2} = \frac{Q_i - X_{2i}\beta_2}{\sigma_2}.
\]

3.6. Experiments

Forty (40) experiments were run on each cost function: (1) Three 3D toy protein folding models with monomers AAAAA, AABBB, and BBBBB [19]; (2) The logistics function [21]; and (3) The disequilibrium problem [22]. Ground truth parameters and known global minima are found in [19], [21], and [22] for problems (1) – (3), respectively. All GPU experiments were run on clusters with NVIDIA Tesla S1070 GPUs (1.44 GHz clock speed) with 240 microprocessors supporting double precision computation, and two gigabytes of global memory. CPU results were obtained on a quad-core Intel E5430 Xeon processor with 8GB of FBD PC 5300 RAM. GPU code was developed with the NVIDIA CUDA framework. Code optimizations were similar in both implementations. For asynchronicity, each particle (population member) explored the search space independently and updated the rest of the flock every 50th iteration. Populations of 500, 1000, 2000, and 4000 particles.
were each budgeted 10000 iterations. The adaptive inertial constant $w$ was applied to reduce entrapment in local minima. Set to 1.0 initially, $w$ was incremented by 0.01 if there was no improvement in the cost function after 50 iterations, to a maximum of 1.03, to facilitate global exploration. If stagnation still occurred, $w$ was reduced by 0.01 to a minimum of 0.97 to refine good areas already identified [6].

4. Results

All protein folding optimizations resulted in correct results on both the GPU and CPU. The logistics function experiments failed with 500 particles, and therefore results for larger populations are presented (97.5% correct for CPU trials, 100% for GPU). The 8D disequilibrium function was the most difficult to optimize, because of the complexity of the cost function landscape and because of the inherent noisiness of the input data [6] (77.5% correct for CPU trials, 80.0% for GPU).

The mean GPU and CPU computation times are shown in Table 1. Using 500 particles as a baseline (1000 for the logistics function), it is seen that computation time for all cost functions increases linearly with the number of particles on the single threaded CPU, as expected – as the number of particles doubles, the computation time also approximately doubles. The computation times on the GPU, as well as being lower, also remain constant with the number of particles, as additional computing resources are not required. More resources on the CPU are required for larger populations, whereas on GPU hardware, existing processing units are better utilized.

Because the robustness of PSO generally increases with the number of particles, speedup was computed based on the population size. That is, for $P = 500, 1000, 2000, \text{ and } 4000$, \textit{Speedup}($P$) = CPU time ($P$) / GPU time ($P$). The results are shown in figure 3. The toy protein model was a simple calculation of an explicit formula (Eq. 3), and therefore, was very efficient on CPUs, lowering the CPU/GPU speedup. The logistic and disequilibrium functions require more computation and also make use of observed data. The 2D logistic function had the lowest computation time overall, but also the highest CPU/GPU speedup. The 8D disequilibrium problem required the most CPU time. However, if $f_1, f_2, h_1, \text{ or } h_2$ become unreasonably large (because of division by small $\sigma_1$ or $\sigma_2$), $L(\theta)$ (Eq. 6) is not considered valid, and the computation does not proceed. In this way, “bad” values are not used in the swarm, thereby increasing efficiency and the CPU/GPU speedup.

Table 1. Mean (± std. dev.) optimization time (seconds) for cost functions by number of particles. GPU results are shown in boldfaced type.

<table>
<thead>
<tr>
<th>Cost Function</th>
<th>Particles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500</td>
</tr>
<tr>
<td>Toy Protein AAAAA CPU</td>
<td>125.16 ± 3.06</td>
</tr>
<tr>
<td>Toy Protein AAAAA GPU</td>
<td>9.37 ± 0.13</td>
</tr>
<tr>
<td>Toy Protein BBBBB CPU</td>
<td>121.34 ± 2.20</td>
</tr>
<tr>
<td>Toy Protein BBBBB GPU</td>
<td>9.36 ± 0.01</td>
</tr>
<tr>
<td>Toy Protein AABBB CPU</td>
<td>111.95 ± 0.81</td>
</tr>
<tr>
<td>Toy Protein AABBB GPU</td>
<td>9.32 ± 0.02</td>
</tr>
<tr>
<td>Logistics (log likelihood) CPU</td>
<td>-</td>
</tr>
<tr>
<td>Logistics (log likelihood) GPU</td>
<td>-</td>
</tr>
<tr>
<td>Diseq. (log likelihood) CPU</td>
<td>133.27 ± 6.15</td>
</tr>
<tr>
<td>Diseq. (log likelihood) GPU</td>
<td>5.30 ± 0.00</td>
</tr>
</tbody>
</table>
5. Discussion

GPU computing takes advantage of data parallelism, and the current work is within the fine-grained paradigm by assigning one particle per thread. However, the logistics function and the disequilibrium problem require external experimental data – each patient’s age and CHD incidence for the former, and quantity transacted and explanatory variables for the latter. Each thread in both cases needed to access values from a large table of memory (originally stored entirely in global memory), which consequently increased the total execution time of the kernel. A more efficient use of the cache and of local memory reduced the slowdown, but longer relative execution times were still observed because memory overhead was added, reducing data parallelism. However, the massive parallelism of the GPU allowed more kernels (particles) to explore the search space simultaneously.

Previously, limiting the number of particles was seen as necessary to avoid slow convergence in serial implementations. With distributed-memory parallelism (e.g. MPI implementations), a reasonable number of particles was required to reduce communications overhead, and with SMP parallelism (e.g. with OpenMP), the number of available threads was an important consideration. With GPUs, however, increasing the number of particles (population size) is an easy way to make best use of this hardware. As seen from Table 1, the running times for each problem remained almost constant (zero slope) with an increasing population size, while the serial implementation, as expected, exhibited a linear increase in computation time. For instance, in the current work, 256 threads (particles) per block can be executed concurrently. There are 30 available processors on the GPU cluster, for a total of 7680 potentially concurrent threads. Therefore, a population of 4000 particles makes better use of resources than one with 1000 particles. Although speedup will decrease as the number of particles exceeds available concurrent threads, even smaller dual- and quad-core systems can take advantage of available GPU capabilities to execute a large number of concurrent threads.

6. Conclusion

Global optimization is one of the important target areas for general purpose GPU computing. Inherently parallel algorithms, including population-based stochastic methods such as PSO, are prime candidates for GPGPU implementation. The results presented in this paper demonstrate that parameter estimation problems requiring external data (contrasted with explicitly defined functions) are also possible on this new architecture without over-burdening the CPU-to-GPU memory bottleneck. Excellent scaling is observed with the population size, as more particles imply better use of available GPU compute resources. Furthermore, asynchronous execution has a minimal effect on the quality of the solutions. For larger parameter estimation problems, managing data and data transfer remain difficult, as larger observed and explanatory data are required. Employing hybrid
parallel architectures, such as multicore/many-core CPUs with GPU acceleration, will likely prove valuable, and may assist in overcoming the limits of GPU-only computing.

Although these results are encouraging, it should be noted that the cost functions addressed here were difficult, but relatively low-dimensional. PSO, while easy to code and requiring few tunable parameters, is less robust vis-à-vis other global approaches (such as genetic algorithms) for higher-dimensional problems [23]. Additionally, for realistic model calibration and simulation-based global optimization problems, other stochastic and deterministic approaches may be preferred [22–23]. Consequently, future work will include GPU and hybrid implementations to apply global optimization to solve more data intensive problems, and for realistic model calibration in practical applications.

Acknowledgments
The authors thank Dr. Renata Wachowiak-Smolíková, Jessica Tanner, Bryan Sarlo, and the anonymous reviewers for their helpful comments and criticisms. This work was supported by, and experiments run on the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca) and Compute/Calcul Canada. Funding was provided by an NSERC Discovery Grant.

References