Parallel Particle Swarm Optimization on Graphical Processing Unit for Pose Estimation

VINCENT ROBERGE, MOHAMMED TARBOUCHI
Electrical and Computer Engineering
Royal Military College of Canada
PO Box 17000, Station Forces, Kingston, Ontario, K7K 7B4
CANADA
vincent.roberge@rmc.ca, tarbouchi-m@rmc.ca

Abstract: - In this paper, we present a parallel implementation of the Particle Swarm Optimization (PSO) on GPU using CUDA. By fully utilizing the processing power of graphic processors, our implementation provides a speedup of 215x compared to a sequential implementation on CPU. This speedup is significantly superior to what has been reported in recent papers and is achieved by a few simple optimizations we made to better adapt the parallel algorithm to the specific architecture of the NVIDIA GPU. Next, we apply our parallel PSO to the problem of 3D pose estimation of a bomb in free fall. We reduce the computation time of the analysis of 120 images to about 1 s, representing a speedup of 140x compared to the sequential version on CPU.

Key-Words: - CUDA, graphic processing units, particle swarm optimization, parallel implementation, 3D pose estimation

1 Introduction

The particle swarm optimization (PSO) is a population based non-deterministic optimization algorithm [1]. Since it was first proposed in 1995, the PSO has been extensively used to optimize very complex functions in a wide range of applications. Its implementation is simple, its performance is very competitive and it is not required to derive the function to be optimized. However, because the algorithm simulates the movement of a swarm of candidate solutions over a very large number of iterations, it has the disadvantage to require significant processing power.

In order to reduce the computation time, we propose a parallel implementation of the PSO on GPU in CUDA [2]. Our approach allows the execution of the PSO with a very large number of particles and iterations in a minimal time. This paper provides three main contributions. First, it presents an innovative implementation of the original PSO on GPU. Second, it proposes four simple optimizations to fine tune the algorithm for the NVIDIA GPUs, resulting in a very impressive speedup compared to what has been previously published in [3], [4] and [5]. Finally, we demonstrate how our parallel PSO can be applied to the 3D pose estimation of a bomb in free fall allowing for very high accuracy and an extremely short execution time.

The remainder of this document is organized in sections. Sections 2 and 3 briefly discuss the latest architecture of the NVIDIA GPU and provide an overview of the PSO algorithm. Section 4 summarizes some of the previous work done in the field. We present our parallel implementation of the PSO in section 5 and propose four optimizations in section 6. The performance results of our implementation are published in section 7. Finally, we apply our algorithm to the problem of 3D pose estimation of a bomb in free fall in section 8.

2 GPU Architecture

The graphic card used in our experiments is an EVGA NVIDIA GTX560Ti [6]. It is equipped with the NVIDIA graphic processor Fermi GF116 composed of 384 cores. This chip is currently available with a maximum of 16 multi-processors blocks, each equipped with 32 processors. In the case of the GTX560Ti, it is equipped with 8 blocks of 48 processors. We could discuss here all the details of the GPU architectures. These processors are indeed significantly different from CPUs. However, we will focus on four characteristics of the GPUs that we deem important. We actually optimized our implementation of the PSO based on those characteristics.

2.1. PCI Express Bus

In today's personal computers, graphic cards are usually connected to the motherboard through a PCI Express bus. Although rated to 16 GB/s for the PCIe 2 bus, the actual throughput achieved when transferring data from the main memory to the graphic card is usually lower and depends on the hardware used. As an example, the NVIDIA GTX throughput is about 5.11 GB/s [4]. This bandwidth is slower than the memory bandwidth of recent CPUs and very much slower than the memory bandwidth of recent GPUs [7]. When designing a CUDA application, it is therefore important to consider the bottleneck caused by the PCIe bus and to minimize the data transfers between the CPU and the GPU. This slow transfer could easily cancel the performance gain of a parallel implementation.

2.2. Memory architecture

The memory of the NVIDIA Fermi GPU is divided into global memory, constant memory, texture memory, shared memory and registers [8]. The constant and texture memory can be bothersome to use and are less attractive for scientific computing. The global memory resides off chip, can be a few GB in size and has a large data bus resulting in a very high bandwidth (up to 192.4 GB/s [9]). It is used to store large amounts of data and is persistent for the entire length of the application. The shared memory has a maximum size of 48 KB and resides on chip. This memory is shared between all the processors of a multi-processor block and is persistent for the duration of a parallel function (called a kernel in CUDA). The latency of this memory is about 20 to 40 times shorter than for the global memory [8]. The registers are even smaller than the shared memory, but offer a slightly faster access. Their scope is also limited to the life of the kernel. The maximum use of shared memory and registers is therefore recommended to improve the efficiency of a CUDA application.

2.3. Single-instruction, multiple threads

Unlike multicore systems (such as dual or quad-core Intel and AMD CPUs), graphic processors can contain hundreds of cores and are considered "manycore" systems. This type of system allows the parallel executions of many threads on a single multi-processor block. It has the advantage of providing superior computing power, but the disadvantage that all threads within the block must execute the same instruction at the same time. These systems have been categorized as "single-instruction, multiple-threads" (SIMT) by NVIDIA [8]. Their functioning is directly linked to the hardware architecture of the multi-processor blocks which use a limited number of control units (instruction caches, schedulers, dispatch units) and a large number of cores. As a consequence, when a parallel program contains conditional statements (*if*, *elseif*), the multi-processor block will sequentially execute the possible paths regardless the number of active cores. This phenomenon is called thread divergence and should be avoided to maximize performance [7].

3. Particle Swarm Optimization

The PSO is a population based non-deterministic optimization method that was proposed by Kennedy and Eberhart in 1995 [1]. The algorithm simulates the movement of a swarm of particles in a multidimensional search space progressing towards an optimal solution. The position of each particle represents a candidate solution (a complete trajectory encoded in a single vector) and is randomly initiated. At every step of the iterative process, the velocity of each particle is individually updated based on the previous velocity of the particle, the best position ever occupied by the particle (personal influence) and the best position ever occupied by any particle of the swarm (social influence). As outlined in [10], the equations used to compute the velocity and position of a single particle at iteration t are as follows:

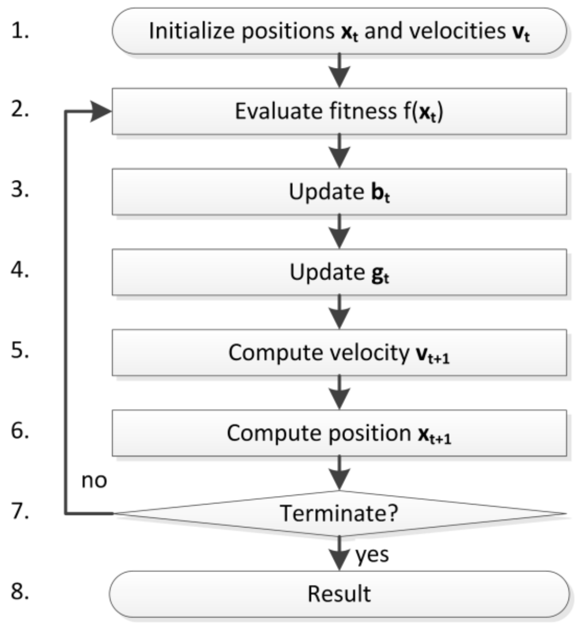
$$\boldsymbol{v}_{t+1} = \omega \boldsymbol{v}_t + c_1 \boldsymbol{r}_1 .* (\boldsymbol{b}_t - \boldsymbol{x}_t) + c_2 \boldsymbol{r}_2 .* (\boldsymbol{g}_t - \boldsymbol{x}_t) \quad (1)$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{v}_{t+1} \quad (2)$$

where variables in bold are vectors; $\boldsymbol{v}$ is the velocity of the particle; $\boldsymbol{x}$ is its position; $\boldsymbol{b}$ is the best position previously occupied by the particle; $\boldsymbol{g}$ is the best position previously occupied by any particle of the swarm; $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ are vectors of random values between 0 and 1; and ω, c_1 and c_2 are the inertia, the personal influence and the social influence parameters. Still based on [10], the flow diagram of the PSO is displayed in Fig. 1.

4. Related Works

Scientific computing on GPUs is a relatively new field of research. The CUDA SDK was actually released in 2007 [7]. For this reason, there are only a few publications available that discuss the implementation of the PSO on GPU. In [3], Zhou and Tan implement in CUDA a local variant of the PSO. Instead of using a global communication scheme as in the original PSO [1], they restrict the communication of a particle to its two closest neighbors (based on the particles' index, not the actual location in the search space). This approach does not require a parallel search for the best particle g and limits the communication between the threads. However, it also modifies the behavior and affects the effectiveness of the PSO. They achieve a speedup of 11.4x. The authors of [11] implement and compare three different variants of the PSO on GPU, but only parallelize the evaluation of the cost function. Although they report a maximum speedup of 27x, we believe that their approach is not scalable to larger swarm sizes since the initialization, the search for the best particle, and the velocity and position updates are performed sequentially. The authors of [4] discuss the impact of different thread-block sizes over the performance of the PSO in a GPU implementation. They also limit their parallelization to the evaluation of the cost function. They report a speedup of 43x, but their results provide insight with regard to the specific cost function used, but not for the PSO. More recently, the authors of [12] proposed a GPU-based asynchronous PSO that uses a single CUDA kernel and simultaneously runs independent swarms on the different multi-processor blocks. Their implementation eliminates the communication between the thread blocks, but modifies the behaviors of the original PSO. Their approach also has the disadvantage of forcing a single thread block size for all steps of the PSO including the fitness evaluation. This can significantly reduce the maximum speedup achieved for complex fitness functions [4].



1.
Initialize positions $\mathbf{x}_t$ and velocities $\mathbf{v}_t$
2.
Evaluate fitness $f(\mathbf{x}_t)$
3.
Update $\mathbf{b}_t$
4.
Update $\mathbf{g}_t$
5.
Compute velocity $\mathbf{v}_{t+1}$
6.
Compute position $\mathbf{x}_{t+1}$
no
7.
Terminate?
yes
8.
Result

Fig. 1. Flowchart of the particle swarm optimization algorithm

They achieved a speedup of 30x for the Rosenbrock function. Finally, the authors of [13] proposed a similar approach using multiple swarms, but did not limit their implementation to a single kernel and allowed migrations of the particles between the swarms. They achieved a speedup of 37x. In this paper, we parallelize every step of the original PSO and achieve a 215x speedup for the Rosenbrock function while maintaining the behavior of the original PSO algorithm.

5. CUDA Implementation of the PSO

5.1. Overall design

The flowchart of our parallel implementation of the PSO in CUDA is presented in Fig. 15 at the end of this document. The program starts on the CPU and copies the configuration parameters, such as the swarm size and the number of iterations, to the global memory of the GPU. The CPU then launches the execution of the GPU. The particle swarm is randomly initiated and its movement is simulated on the GPU. In between each parallel kernel (a parallel function in CUDA), the state of the swarm is saved in the global memory of the GPU. Once the specified number of iterations has been reached, the control is given back to the CPU who then copies the optimal solution from the GPU to the CPU. It is important to note that a block of threads is executed by a single multi-processor block and there is no means to synchronize or communicate between blocks other than terminating and launching another kernel. Moreover, the CPU and the GPU do not share the same memory and special CUDA functions must be used to transfer the data.

As used in [14], [15] and [16], a common technique to parallelize the PSO consists of executing a local swarm on each processor while minimizing the communication between the swarms. This approach can be used on current multicore processors, but does not provide the level of parallelism needed for an efficient implementation on GPU. In our implementation, we launch one independent thread for each particle. For a complex problem, the number or particles used may be very large resulting in more threads than GPU cores. Although it may seem undesirable, it is actually advantageous to create more threads than cores. In an NVIDIA GPU, the creation, scheduling and destruction of threads are done in hardware and require minimal time. Moreover, using a lot of threads will give flexibility to the GPU to better schedule the computation and hide the memory access latency. A parallel program which uses a lot of threads will better scale to future GPUs. For all these reasons, it is recommended to develop parallel algorithms that will provide sufficient parallelism to maximize the number of threads created [7]. The NVIDIA Fermi GPUs allow the creation of 2^{16} blocks of 2^{10} threads (for a total of 2^{26} threads) [8].

5.2. Implementation details

In this section, we present the implementation details of each kernel shown in Fig. 15.

5.2.1 Initialize particles' position and velocity

This kernel launches one thread for each particle of the swarm. It randomly initializes the position of the particles within the search space using the *curand_uniform()* function from the CUDA CURAND library [17] and sets the particles' velocity to 0. Before terminating, the kernel saves the particles' position and velocity in the GPU global memory so the data is available to the next kernel.

5.2.2 Compute costs and update local bests

This kernel launches a thread for each particle. This thread loads the position of the particle and computes the associated cost by evaluating the function to be optimized. If necessary, the thread updates the best cost and best position b ever occupied by the particle (local influence). The thread saves this information back to the global memory.

5.2.3 Find the index of the best particle

This kernel launches one thread per particle and searches for the index of the best particle using a parallel tree-based reduction [18] (shown on Fig. 2). This type of reduction is a well-known pattern and requires *$log_2(N)$* steps. Because CUDA thread-blocks cannot communicate between them, we divide the parallel reduction into two kernels.

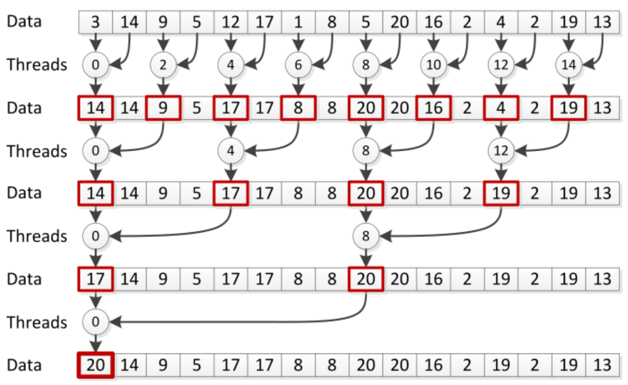


Fig. 2. Tree-based parallel reduction

The first kernel finds the best particle within each block and the second, the best particle within the entire swarm. The first kernel is then launched with a number of threads equal to the swarm size and the second, with a number of threads equal to the number of thread-blocks of the first kernel.

5.2.4 Update the global best

This kernel is launched with a number of threads equal to the dimension of the function to optimize. Each thread simply updates one element of the global best when the best particle found at this iteration is of better quality (lower cost), ensuring a parallel access to the memory.

5.2.5 Compute the particles' new velocity and position

This kernel launches one thread for each particle. This thread first reads the current velocity of a particle, its current position, its best previous position and the swarm's best position and updates its velocity and position using equations (1) and (2). The new velocity and position are saved in global memory.

This entire process is repeated several times until the specified number of iterations is reached. At that time, the position of the best particle of the swarm represents the optimal solution returned to the CPU. Our implementation limits the amount of data transferred between the CPU and the GPU and fully parallelizes all the steps of the PSO algorithm. The speedup achieved is therefore significant.

6. Optimizations

Our parallel PSO exhibits the level of parallelism necessary to fully use the GPU hardware. However, it is still too general and needs to be fine-tuned to the specific architecture of the NVIDIA GPUs as explained in section 2. In this section, we discuss four optimizations we made to our program that are specific to a CUDA implementation. These optimizations further improve the achieved speedup by a factor of 10x.

6.1 Maximum use of shared memory

Each multi-processor block has 48 KB of shared memory physically collocated on the chip. As explained earlier, this memory has a very fast access time. For this reason, we modified all our kernels to always load the data in shared memory before performing the calculations. As an example, in the first kernel, each thread calls the *curand_uniform()* function multiple times to initialize each dimension of the particle's position. Our initial implementation loaded and stored the state of the random generator (40 bytes per thread) from global memory every time. In our optimized version, the states are first loaded in shared memory and all future memory accesses are done to the shared memory. Once completed, the kernel stores all the variables back to the global memory to ensure persistence of the data.

6.2 Coalescing memory access

NVIDIA GPUs are equipped with a very large global memory bus (up to 384-bit) resulting in a very high bandwidth (up to 192.4 GB/s) [9]. However, this performance can only be achieved when the memory access pattern is fully coalesced [8]. One of the main factors for this condition is that each thread involved in a parallel load operation accesses memory cells that are collocated. In our initial implementation, we stored the positions and velocities of the particles in global memory in a contiguous manner as shown in Fig. 3. When the threads load the data from global memory to shared memory, they simultaneously fetch one dimension at the time (remember the "single-instruction, multiple-threads" model). The throughput achieved is therefore *1/D* of the rated bandwidth (where *D* is the dimension of the particles). To address this limitation, we re-organized our data in an interleave configuration, as in Fig. 4, to ensure a coalescing memory access pattern, resulting in an improved throughput.

P0	P0	P0	P0	P1	P1	P1	P1	P2	P2	P2	P2	P3	P3	P3	P3

Fig. 3. Sub-optimal memory storage layout for 4 particles of 4D resulting in a non-coalescing memory access pattern to global memory

P0	P1	P2	P3	P0	P1	P2	P3	P0	P1	P2	P3	P0	P1	P2	P3

Fig. 4. Optimal memory storage layout for 4 particles of 4D resulting in a coalescing memory access pattern to global memory

6.3 Thread divergence

As explained earlier, when programming a "single-instruction, multiple-threads" system, it is important to minimize the different execution paths to avoid their sequential execution. In CUDA, thread divergence is actually only applicable to groups of 32 threads (called warp). In other words, peak performance will only be achieved when all the threads within the same warp follow the same execution path. In order to minimize thread divergence, we modified our parallel reduction kernel following the approach presented in [19]. In our original implementation, every warp suffered thread divergence while only one warp exhibited the divergence in our optimized version shown in Fig. 5.

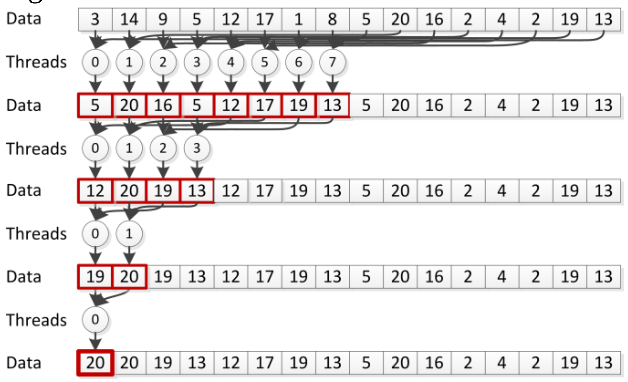


Fig. 5. Optimized version of the tree-based parallel reduction minimizing thread divergence

6.4 Block size

When launching a CUDA kernel, different thread-block configurations can be chosen. A programmer might decide to create a few blocks, each containing many threads or many blocks, each containing a few threads. As explained in [4], this design decision will influence the overall performance of the application. Although one could mathematically compute the best configuration, it is usually suggested to experimentally test multiple configurations [7]. This test should be repeated when using different application parameters (such as the PSO swam size) or hardware (such as a newer, more powerful GPU). In our case, we ran our parallel PSO using different numbers of particles and block sizes. As shown in Fig. 6, we found that smaller block sizes generally deliver better performance for a smaller problem size. For 16 384 particles, a block size of 128 threads provides the best performance, 58% more than with a block size of 32 threads.

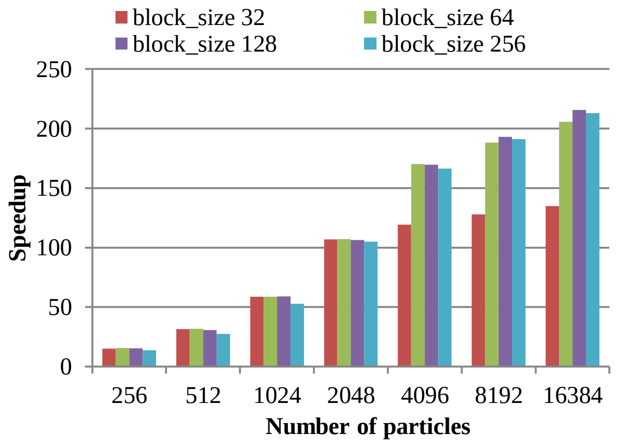


Fig. 6. Speedup of our parallel PSO for different number of particles and block sizes

7. Performance Results

To test the performance of our parallel PSO, we optimize the following benchmark function which is based on the Rosenbrock function and illustrated for 2D in Fig. 7.

$$f_{Ro}(\boldsymbol{x}) = \frac{1}{\sum_{d=1}^{N-1}\left(\left(100 * (x_{d-1} - x_d^2)\right)^2 + (x_d - 1)^2\right)} \qquad (3)$$

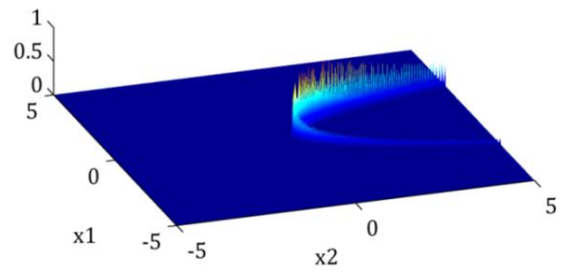


Fig. 7. Benchmark function based on the Rosenbrock function

For our performance evaluation, we ran our sequential and parallel PSO to optimize equation (3) with 20 dimensions, 2000 iterations and different numbers of particles. The characteristics of the system used are listed in Table 1 and the results obtained are shown in Fig. 8 and Fig. 9. The maximum speedup achieved is 215.6x, which is significantly higher than what was achieved by other CUDA PSO implementations proposed in recent papers (11.4x in [3], 17.2x in [5] and 43.9x in [4]). We believe that our better performance is related to the parallelization of the entire PSO and the four optimizations we discussed in section 6.

Table 1. Details of our experimental setup

Host PC used to run the sequential version:
• AMD Phenom II X6, 2.80 GHz • 8 GB DDR3, 1333 MHz • Windows 7 SP1 64 bit Enterprise • Visual Studio 2010 SP1
Device GPU used to run the parallel version in CUDA:
• EVGA NVIDIA GTX 560 Ti • 384 CUDA cores, 1.701 GHz • 1 GB DDR5, 2052 MHz • Windows 7 SP1 64 bit Enterprise • Visual Studio 2010 SP1 • CUDA Toolkit 4.0 • CUDA SDK 4.0

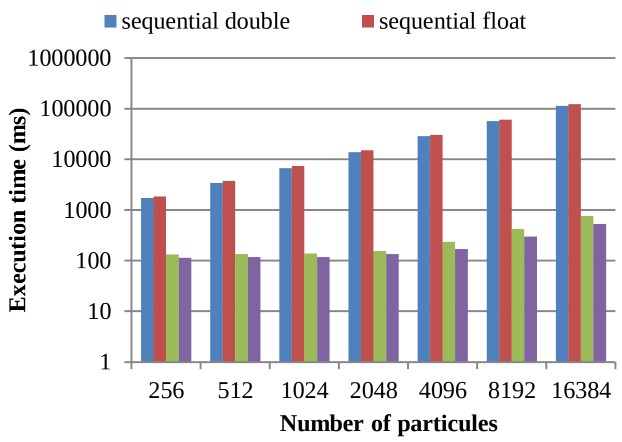


Fig. 8. Execution time of our sequential and parallel PSO for different precisions and number of particles (20 dim, 2000 iterations, avg of 20 trials)

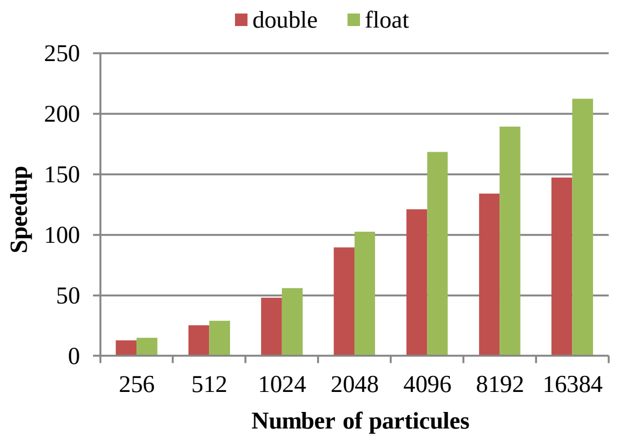


Fig. 9. Speedup of our parallel PSO for different precisions and number of particles (20 dimensions, 2000 iterations, average of 20 trials)

8. Application to 6DOF

In this section, we use our parallel PSO to accelerate the computation of the 6 degrees of freedom (6DOF) *(x, y, z, yaw, pitch, and roll)* of a bomb in free fall. In modern military aviation, the accreditation of new ordnances usually requires a safe separation test. This test is used to define the safe envelope of operation to drop the bomb. One approach consists of installing a high speed camera on the aircraft and recording multiple drops at different speeds and orientations. The 2D video images are then processed to produce a 3D animation which is visualized under different angles by experts to assess the safety of the separation between the bomb and the aircraft. This approach requires the accurate computation of the 3D position of the bomb from a 2D image. To help with the tracking of the bomb, markers are precisely painted in many locations on the bomb.

We previously developed software to compute the 6DOF from 2D images using an exhaustive search (see Fig. 10) [20]. This approach is extremely slow and not very precise. There exist other methods to tackle the problem such as the direct mathematic method presented in [21]. After experimentation, we concluded that this approach is very fast, but provides bad results for noisy inputs (which is obviously the case for the safe separation test). In this section, we propose to increase the accuracy of the results and minimize the execution time by using our parallel PSO to compute the 6DOF on a GPU.

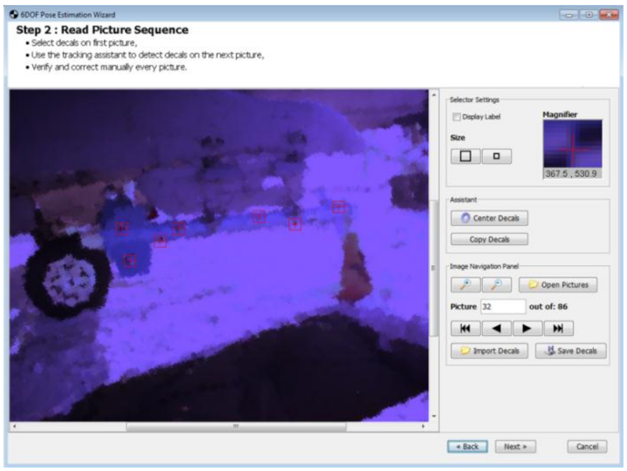

6DOF Pose Estimation Wizard
Step 2 : Read Picture Sequence
• Select decals on first picture,
• Use the tracking assistant to detect decals on the next picture,
• Verify and correct manually every picture.
Selector Settings
Display Label
Size
Magnifier
367.5 , 530.9
Assistant
Center Decals
Copy Decals
Image Navigation Panel
Open Pictures
Picture 32 out of: 86
Import Decals
Save Decals
« Back
Next »
Cancel

Fig. 10. Screenshot of our software showing the bomb being released from the aircraft.

8.1 Cost Function

As discussed earlier, the PSO is a non-deterministic algorithm that allows the optimization of a function without the need to derive it. It other words, the PSO can be used to optimize a problem where the optimal solution is not practically computable (such as the 6DOF problem with noisy input), but a candidate solution is easily evaluated.

In our implementation, we use the PSO to find the 3D position of the bomb that, when projected to the 2D video image, minimizes the average distances between the original markers and the projected markers. For a candidate solution *(t_x, t_y, t_z, yaw, pitch, and roll)*, we define our cost function as follows:

1) Place a virtual bomb centered and oriented in the same direction as the camera;
2) Select the *(x, y, z)* coordinates of one of the marker of the virtual bomb;
3) Rotate this virtual marker using the following equation:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \boldsymbol{R} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4}$$

where *R(α, β, γ)* is the rotation matrix and is defined as follows [22]:

$$\begin{bmatrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\beta) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\beta) \end{bmatrix} \tag{5}$$

where α is the yaw angle, β is the pitch angle and γ is the roll angle of the candidate solution. It is important to note that this matrix performs the three rotations in the roll-pitch-yaw order and would be different if the order was altered.

4) Translate the virtual marker based on the t_x, t_y and t_z of the candidate solution:

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{6}$$

5) Project the virtual marker on the 2D images using:

$$x''' = f_x * \left(\frac{x''}{z''} + \alpha_c \frac{y''}{z''} \right) + c_x \tag{7}$$

$$y''' = f_y * \frac{y''}{z''} + c_y \tag{8}$$

where the vector *(f_x, f_y)* represents the focal distance of the lens in pixels, the vector *(c_x, c_y)* represents the coordinates of the principal point of the lens in pixels, and α_c is the skew coefficient (angle between the *x* and *y* pixels) [23].

6) Finally, compute the distance between the virtual marker projected on the 2D image and the actual marker identified on the 2D image using:

$$error = \sqrt{(x - x''')^2 + (y - y''')^2} \tag{9}$$

where *(x, y)* is the position in pixels of the actual marker on the 2D image and *(x''', y''')* is the position in pixels of the virtual marker projected on the 2D image.

7) Repeat these steps for all markers visible on the 2D video image and compute the average error per marker in pixels.

Throughout the iterative process of the PSO, the swarm of candidate solutions will move towards a solution that minimizes the above cost function. This final solution represents the 3D position of the bomb for that 2D image. The process must be repeated for every image of the video sequence.

This approach demonstrates a very high level of parallelism and can be accelerated on GPUs using CUDA.

8.2 CUDA implementation

Our parallel implementation in CUDA of the PSO applied to the problem of pose estimation of a bomb in free fall is almost identical to the one we previously described in section 4. As shown in Fig. 11, the algorithm creates an independent swarm of particles for each 2D image and still uses one thread per particle. The positions of the bombs are therefore computed simultaneously for all images using an independent PSO algorithm. For 128 images and 256 particles, our software initially launches 32 768 threads. The cost function is replaced with the one described at the previous section. The parallel reduction is now implemented with a single kernel since each thread-block implements an independent PSO. The rest of the algorithm remains identical.

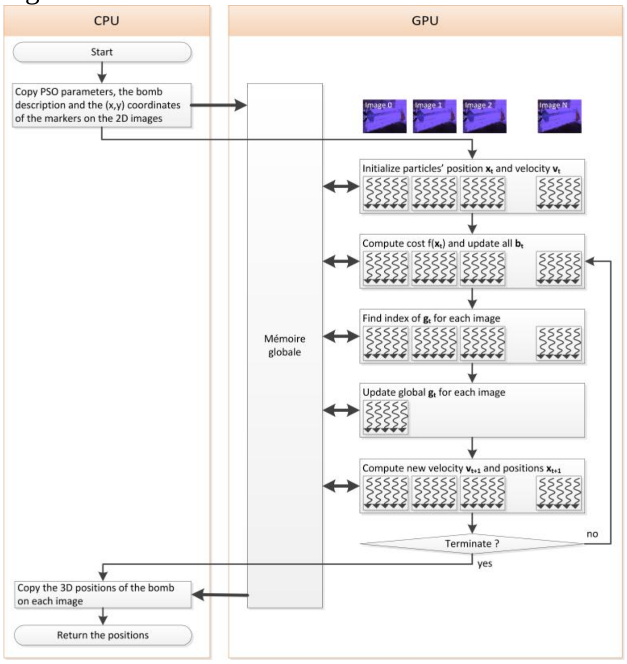

CPU
GPU
Start
Copy PSO parameters, the bomb description and the (x,y) coordinates of the markers on the 2D images
Image 0
Image 1
Image 2
Image N
Initialize particles' position x_t and velocity v_t
Compute cost $f(x_t)$ and update all b_t
Find index of g_t for each image
Mémoire globale
Update global g_t for each image
Compute new velocity v_{t+1} and positions x_{t+1}
Terminate ?
no
yes
Copy the 3D positions of the bomb on each image
Return the positions

Fig. 11. Flowchart of our implementation of the parallel PSO in CUDA applied to the 6DOF problem

8.3 Performance analysis

To assess the performance of our parallel algorithm, we developed a sequential version for the CPU. We used the system described in Table 1 and executed our algorithms to analyse 128 video images with 2000 iterations and different swarm sizes. The execution times and speedups achieved are shown in Fig. 12 and Fig. 13. The maximum speedup reached is 140.3x with an execution time of 1.4 s. To demonstrate the precision of our approach, we also show the average error in pixels for each image on Fig. 14. The average error for all images is of 0.286 pixels per marker which is 21% more precise than the original exhaustive search method we developed [20]. It is important to note that an error of 0 would be impossible since it would involve that the marker detection was 100% accurate with an extreme sub-pixel precision and that the camera calibration was also perfect. An error of 0.286 pixels is excellent and truly demonstrates the accuracy of our PSO algorithm.

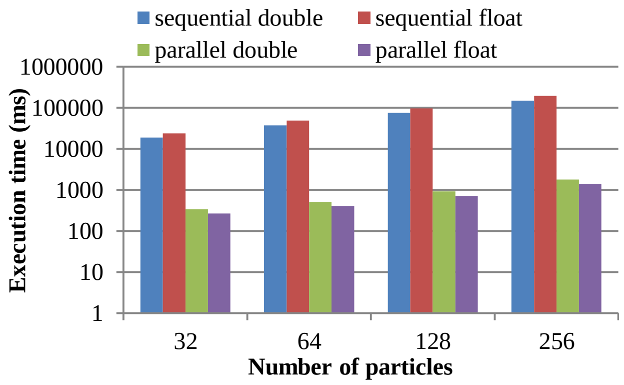


Fig. 12. Execution of our sequential and parallel PSO for different precisions and number of particles (6DOF problem, 2000 iterations, average of 20 trials)

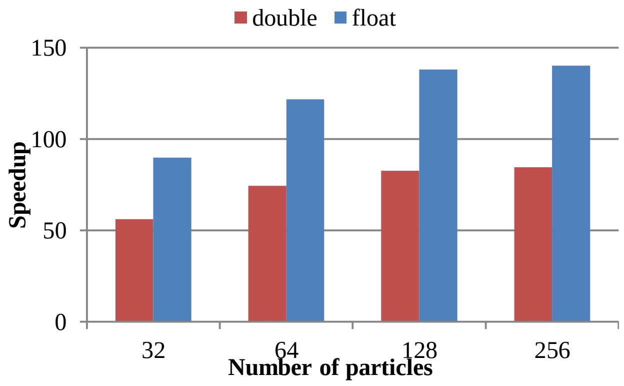


Fig. 13. Speedup of our parallel PSO for different precisions and number of particles (6DOF problem, 2000 iterations, average of 20 trials)

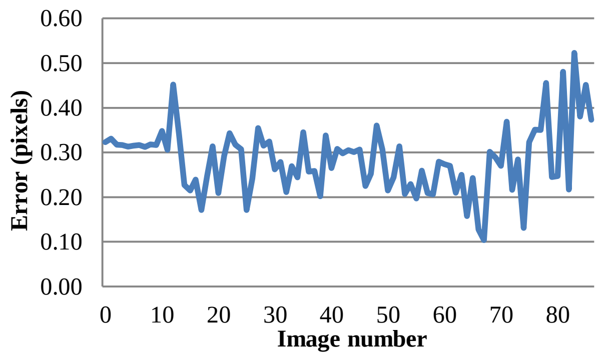


Fig. 14. Average error in pixels for each image

9. Conclusion

In this paper, we presented a brief description of the latest NVIDIA GPU architecture and its limitations. We provided an overview of the PSO algorithm and proposed a parallel implementation on GPU using CUDA. Unlike multicore CPU, GPU are built of hundreds of cores. The CUDA program must exploit a very high level of parallelism to fully use the GPU hardware. Unlike other implementations presented in the literature, we exploited this level of parallelism and tuned our algorithm to achieve a speedup of 215x on NVIDIA GPU. Finally, we applied our algorithm to accelerate the computation of the 6 degrees of freedom of a bomb in free fall. This allowed an increase in the accuracy of the results and a speedup of 140x. This paper represents a significant contribution since it provides a parallel implementation of the PSO with a speedup higher than any previously proposed implementations. Our approach can be used at the core of many optimization solvers to significantly speedup the computation using a GPU. In the future, we intend to implement other optimization algorithms on GPUs using CUDA.

References:

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," Proceedings of: *IEEE International Conference on Neural Networks*, Perth, WA, Australia, 1995, pp. 1942-1948.

[2] "CUDA" Available: http://www.nvidia.com/ object/cuda_home_new.html.

[3] You Zhou and Ying Tan, "GPU-based parallel particle swarm optimization," Proceedings of: *2009 IEEE Congress on Evolutionary Computation*, Piscataway, NJ, USA, 2009, pp. 1493-500.

[4] M. Cárdenas-Montes, M. Vega-Rodríguez, J. Rodríguez-Vázquez, and A. Gómez-Iglesias, "Effect of the Block Occupancy in GPGPU over the Performance of Particle Swarm Algorithm," *Adaptive and Natural Computing Algorithms*, Springer Berlin / Heidelberg, 2011, pp. 310-319.

[5] Y. Tan and Y. Zhou, "Parallel Particle Swarm Optimization Algorithm Based on Graphic Processing Units," *Handbook of Swarm Intelligence*, Springer Berlin Heidelberg, 2010, pp. 133-154.

[6] "EVGA | Intelligent Innovation" Available: http://www.evga.com/.

[7] D.B. Kirk and W. Mei W. Hwu, *Programming Massively Parallel Processors, A Hands-on Approach*, Burlington, MA: Elsevier, Morgan Kaufmann, 2010.

[8] NVIDIA CUDA C Programming Guide, version 3.2, Santa Clara, CA, NVIDIA Corporation, 2010.

[9] NVIDIA Corporation, "GeForce GTX 580" Available: http://www.nvidia.com/object/product-geforce-gtx-580-us.html.

[10] M. Clerc, *Particle Swarm Optimization*, France, Lavoisier, 2005.

[11] G.A. Laguna-Sánchez, M. Olguín-Carbajal, N. Cruz-Cortés, R. Barrón- Fernández, and J. Álvarez-Cedillo, "Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU," *Journal of Applied Research and Technology*, Vol. 7, 2010, pp. 292-309.

[12] L. Mussi, Y.S.G. Nashed, and S. Cagnoni, "GPU-based asynchronous particle swarm optimization," Proceedings of: *13th annual conference on Genetic and evolutionary computation*, Dublin, Ireland, ACM, 2011, pp. 1555-1562.

[13] S. Solomon, P. Thulasiraman, and R. Thulasiram, "Collaborative multi-swarm PSO for task matching using graphics processing units," Proceedings of: *the 13th annual conference on Genetic and evolutionary computation*, Dublin, Ireland: ACM, 2011, pp. 1563-1570.

[14] V.R. Roberge, "*Contributions à la conception d'un système opérationnel de planification de trajectoires en temps réel pour les drones*," M.S. thesis, Collège Militaire Royal du Canada, 2011.

[15] F. Parra, S. Galan, A. Yuste, R. Prado, and J. Muñoz, "A Method to Minimize Distributed PSO Algorithm Execution Time in Grid Computer Environment," *Bioinspired Applications in Artificial and Natural Computation*, Springer Berlin / Heidelberg, 2009, pp. 478-487.

[16] Z.-hui Zhan and J. Zhang, "Parallel Particle Swarm Optimization with Adaptive Asynchronous Migration Strategy," *Algorithms and Architectures for Parallel Processing*, Springer Berlin / Heidelberg, 2009, pp. 490-501.

[17] NVIDIA, "CUDA CURAND Library," Aug. 2010.

[18] T.G. Mattson, B.A. Sanders, and B.L. Massingill, *Patterns for Parallel Programming*, Addison Wesley, 2004.

[19] J. Hoberock and D. Tarjan, “NVIDIA Lecture Slides for Course CS193G, Standford University.”

[20] V. Roberge, G. Vigeant, and A. Forest, “*Document de conception détaillée pour le projet des six degrées de liberté (6DOF)”*, GEF455/457-DID-08, Royal Military College of Canada, Mar. 2005.

[21] A. Ansar and K. Daniilidis, “Linear pose estimation from points or lines,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 25, 2003, pp. 578-589.

[22] S. Lavalle, “*Yaw, pitch, and roll rotations*”, Planning Algorithms Available: http://planning.cs.uiuc.edu/node102.html.

[23] “Camera Calibration Toolbox for Matlab” Available: http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/parameters.html.

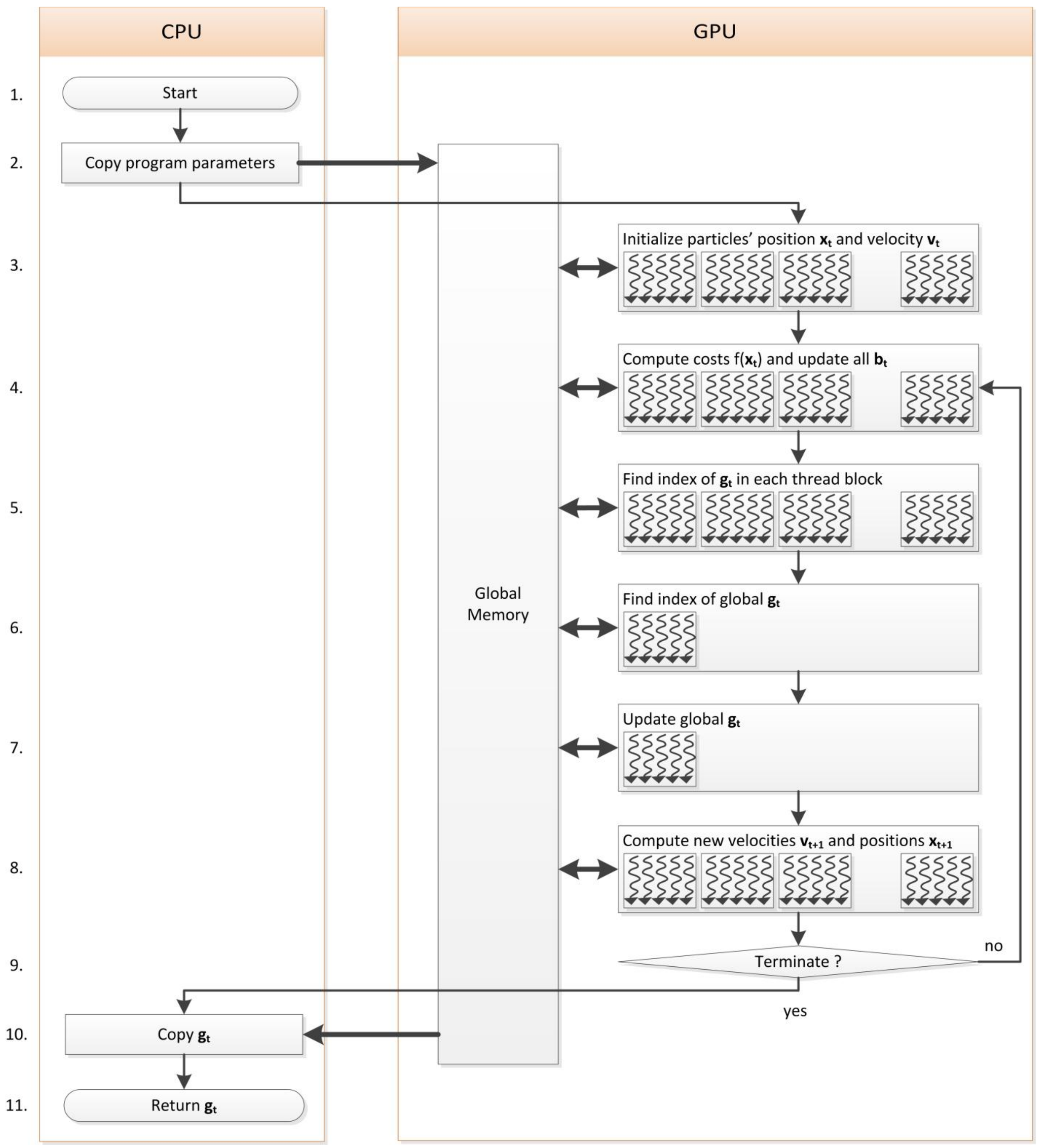

CPU
GPU
1.
Start
2.
Copy program parameters
3.
Initialize particles’ position $\mathbf{x}_t$ and velocity $\mathbf{v}_t$
4.
Compute costs f($\mathbf{x}_t$) and update all $\mathbf{b}_t$
5.
Find index of $\mathbf{g}_t$ in each thread block
6.
Global Memory
Find index of global $\mathbf{g}_t$
7.
Update global $\mathbf{g}_t$
8.
Compute new velocities $\mathbf{v}_{t+1}$ and positions $\mathbf{x}_{t+1}$
9.
Terminate ?
no
yes
10.
Copy $\mathbf{g}_t$
11.
Return $\mathbf{g}_t$

Fig. 15. Flowchart of our implementation of the parallel PSO in CUDA