Parallelizing PSO with CUDA

Boris Kravchenko[1]

***Abstract*— An implementation of the Particle Swarm Optimization algorithm using Nvidia CUDA. Run times of single-threaded implementation, multi threaded implementation and CUDA are compared. GPU implementation runs 12x faster than single-threaded CPU implementation in best case. Shows that GPU can provide excellent improvements in runtime for floating point intensive optimization problems.**

***Index Terms*— CUDA, PSO, Multithreading, C++11**

I. Introduction

Presented originally by James Kennedy and Russell Eberhart in 1995, Particle swarm optimization (PSO) [1] is a simple optimization algorithm that simulates a swarm of agents over a search space to find an optimal solution to a continuous non-linear function. Originally motivated by the behavior of flocking birds, this algorithm has been widely studied and expanded upon. Since every agent in the swarm is independent, this algorithm can quite naturally be executed in a concurrent fashion.

Compute Unified Device Architecture (CUDA) [2] is a parallel computing platform created by Nvidia which allows programmers to leverage the parallelism offered by desktop graphic processing units (GPUs) in programming languages such as C++.

In this paper, several variations of PSO are implemented and compared, including the original version as described in the 1995 paper, a multi threaded version and a CUDA version. Lessons that were learned during the CUDA implementation are discussed.

II. Particle Swarm Optimization

Particle Swarm Optimization is an algorithm that runs for a preset number of iterations. Upon initialization, N particles are placed within an M dimensional search space, each assigned a random position P and velocity V. The goal of the algorithm is to find an optimal numerical solution to a given fitness function F. Every iteration t, the following equations are executed:

$$\begin{aligned}\mathcal{V}_{id}(t+1) = V_{id}(t) + c_1 r1(L_{id} - P_{id}(t)) \\ + c_2 r_2(G_d - P_{id}(t))\end{aligned} \tag{1}$$

$$\mathcal{P}_{id}(t+1) = P_{id}(t) + V_{id}(t) \tag{2}$$

Where i = 1...N, d = 1...M, L best position a particle has found so far, G is the best position a swarm has found so far, c1 and c2 are constants, r1 and r2 are random floating point values between 0 and 1.

Following the computation of above equations, each particle calculates the fitness of its current position and updates G_d and L_{id} accordingly. Upon termination, the algorithm outputs G_d for all dimensions, which represents the best value the swarm has found.

III. Compute Unified Device Architecture

A. Introduction to CUDA

In graphics rendering, large sets of pixels and vertices are mapped to parallel threads which execute vertex and pixel shader programs in order to rasterize an image. Shaders consist entirely of floating point calculations. These threads are run on several Graphic Processing Clusters (GPCs). Each GPC contains several Streaming Multiprocessors (SMs). A set of threads, organized as a block, is executed on each SM, following the Single Instruction Multiple Data (SIMD) model of parallel computing. [3] The Nvidia CUDA API allows programmers to leverage this parallelism and run general-purpose floating point algorithms on the SMs. Due to the nature of graphics rendering, the GPU has many more transistors dedicated to floating point computation than the CPU, thus making it advantageous for solving parellelizable floating point problems. [4] Figure 1 depicts Nvidia's latest arcitecture, GM204, which is implemented in the GTX970 and GTX980 graphics cards. It has 4 GPCs with 4 SMs in each GPC.

Fig. 1. GM204 Architecture Diagram [5]

[1] Boris Kravchenko is with the School of Computer Science, University of Waterloo, 200 University Avenue, Waterloo, Ontario, Canada N2L 3G1.

B. CUDA Architecture

The CUDA API provides the programmer with a C/C++ API and a complier. The GPU executes C++ code in blocks of threads, similar to how shader code is executed. Blocks are scheduled to run on different SMs by the GigaThread engine. Each SM is capable of running 32 threads in parallel. These threads share a single program counter. [6] Figure 2 depicts a single SMM (Streaming Multiprocessor Maxwell) unit in the GM204 architecture.

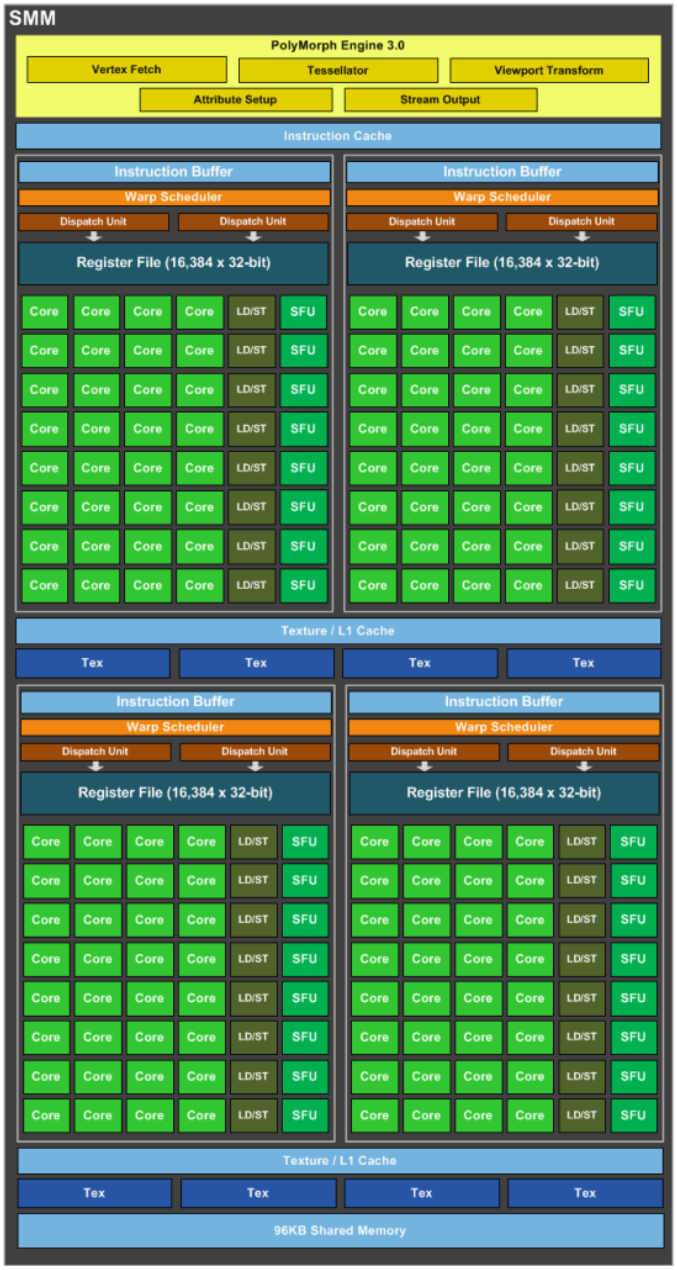

Fig. 2. SMM architecture diagram [7]

C. Structure of a CUDA program

The following section describes the general structure of a CUDA program. CUDA introduces the following terminology:

- **Host** - CPU and RAM
- **Device** - GPU and VRAM
- **Kernel** - Function that runs on the device
- **Warp** - Group of 32 threads executed in parallel on a single SM

The following is a simple CUDA kernel [8] that adds two vectors a and b and saves the result in vector c. Here, each thread is responsible for adding a single item in the array. The index is the thread id in the set of running threads.

```
__global__ void add(int *a,int *b,int *c)
{
  int index = threadIdx.x + blockIdx.x
            * blockDim.x;
  c[index] = a[index] + b[index];
}
```

The following demonstrates how to execute this kernel on the GPU:

1. The input and output parameters of the function are allocated in VRAM.

```
cudaMalloc((void**)&d_a, size)
cudaMalloc((void**)&d_b, size)
cudaMalloc((void**)&d_c, size)
```

2. Data is copied from host memory to device memory. The value of setting here is either CudaMemcpyHostToDevice or CudaMemcpyDeviceToHost, depending on mode of operation needed.

```
cudaMemcpy(d_a, &a, size, setting)
cudaMemcpy(d_b, &b, size, setting)
cudaMemcpy(d_c, &c, size, setting)
```

3. The add kernel is launched on X blocks with Y threads in each block. Threads have shared memory and can cooperate since they all run on the same SM. Blocks do not share memory, and thus cannot cooperate. These parameters are determined by problem size and data dependencies imposed by problem specifications.

```
add<<< X,Y >>>(d_a,d_b,d_c)
```

4. Output data is copied back from the device to the host.

```
cudaMemcpy(&c, d_c, size, setting)
```

5. Memory is freed on the device

```
cudaFree(d_a)
cudaFree(d_b)
cudaFree(d_c)
```

D. Performance Guidelines

The following is a short list of performance guidelines for implementing a CUDA kernel. These were all validated during the implementation of CUDA PSO.

1. Data between the CPU and GPU is transferred via PCIE at around 8 GB/s. This is very slow compared to the memory

bandwidth between GPU and VRAM (100-200 GB/s). [9] Therefore, data should be copied as infrequently as possible. It was found that constantly copying data to GPU, executing the kernel, and copying the result back is a very ineffective way to use CUDA.

2. Due to the SMID model, any control flow in the kernel code causes thread divergence. This causes different execution paths to be serialized since all threads in a warp share the same program counter. The same applies to mutual exclusion. CUDA does not provide mutual exclusion primitives, and while constructing locks out of atomic primitives is possible, using such constructs cripples performance due to thread divergence. Therefore, to minimize thread divergence, it's best to eliminate all control flow and data dependencies inside a CUDA kernel.

3. It's best to reformulate the problem so it can run on multiple blocks. This requires ensuring that threads do not need to cooperate within the kernel. For instance, GTX970 contains 13 SMs [5], and spawning a kernel that only uses a single thread block ends up using only 7.6% of the GPU's processing power, due to the fact that only a single SM is used.

IV. Algorithm Implementation

A. Single threaded PSO

The PSO algorithm was implemented in C++ exactly as defined in section II. This was used as a baseline to test multi threaded solutions against.

B. Multi threaded PSO

For the multi threaded version of the PSO algorithm, the C++ 11 native thread library was used. In this version, T worker threads are spawned, where T = number of physical cores on machine. This version can therefore run at most T times faster than Single threaded PSO. The following pseudo code is a high level overview of the implementation.

```
conditionVariable cv1, cv2
bool ready[T] {false}
atomic int counter;

doWork(){
  while (running){
    while(!ready[threadID]) wait on cv1
    run PSO on 1/T of the particles
    ready[threadID] = false

    if (++counter == T){
      counter = 0
      wake up cv2
    }
  }
}
```

```
int main(){
  spawn T threads with doWork()

  for i in numberOfIterations {
    ready[0...T-1] = true
    wake up all on cv1

    wait on cv2
  }
}
```

C. CUDA Implementation

A part of the PSO algorithm was evolved to use CUDA. Specifically, the fitness function calculation for all particles was batched to run on the GPU every iteration. The justification behind this was that the fitness particle computation is the most floating point intensive part of the algorithm. The following is pseudo code for the kernel that was used:

```
__global__ void calculateFitness(...){
  int particle = blockIdx.x * blockDim.x
               + threadIdx.x;

  for i in dimensions {
    Calculate fitness of
    particle in dimension i

  }

  set fitness of particle

}
```

V. Evaluation

A. Methodology and Results

All the three implementations were evaluated on a machine with the following specifications: Nvidia GTX970, Intel Core i5-4690k @ 3.5Ghz (Quad Core, no Hyper Threading), 16GB of RAM, Windows 8.1, Visual Studio 2013 with CUDA toolkit 7.0

Table I shows fitness functions that were tested. f_1 is the Sphere function, f_2 is the Rosenbrock function [10], f_3 is the Ackley function [10] and f_4 is a floating-point intensive benchmark function. Table II and Table III show the runtimes of all test cases, averaged over 3 runs. Figure 3 depicts the runtime differences in a graph. The multi threaded algorithm was executed on 4 threads in each test.

B. Analysis

Given the strengths and weaknesses of the CUDA architecture, the results were as expected. Executing f_1, f_2 and f_3 with 3 dimensions on the GPU does not yield much speedup, due to the fact that very little time is spent inside the kernel. This is because each thread running the kernel performs the fitness calculation only three times. Thus, most of the

TABLE I

TEST FITNESS FUNCTIONS

f_1	$f(x) = \sum_{i=1}^{d} x_i^2$
f_2	$f(x) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$
f_3	$f(x) = -20exp\left[-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right] - exp\left[\frac{1}{d}\sum_{i=1}^{d} cos(cx_i)\right] + 20 + exp(1)$
f_4	$f(x) = \sum_{i=1}^{d} 2x^2 * sin(exp(sin(exp(sin(x)))))$

TABLE II

TEST 1: 10000 PARTICLES, 3 DIMENSIONS, 1000 RUNS

Function	ST	MT	Speedup	CUDA	Speedup
f_1	$3.41s$	$1.09s$	**2.88 x**	$2.52s$	**1.24 x**
f_2	$4.77s$	$1.46s$	**3.2 x**	$2.59s$	**1.84 x**
f_3	$7.24s$	$2.12s$	**3.4 x**	$2.71s$	**2.6 x**

TABLE III

TEST 2: 1000 PARTICLES, 1000 DIMENSIONS, 1000 RUNS

Function	ST	MT	Speedup	CUDA	Speedup
f_3	156.56s	43.39s	**3.6 x**	34.36s	**4.5 x**
f_4	348.89s	92.35s	**3.7 x**	30.11s	**11.59 x**

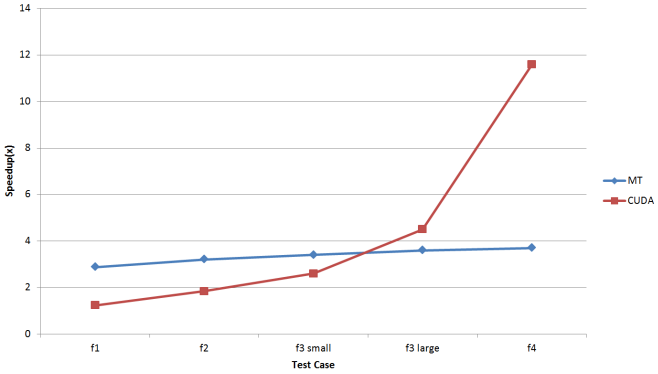

Fig. 3. Speedup of Multi threading solution vs CUDA solution

time is spent copying data to and from the GPU. However, executing f_3 with a very large number of dimensions yields more time spent in the kernel, thus giving a slight speed up over the multi threaded solution. Finally, running f_4 yields the greatest speedup, due to the large number of floating point operations involved in this computation.

VI. CONCLUSION

This paper presented an implementation of PSO on the GPU, using the CUDA API developed by Nvidia. The implementation had the following features:

- For problems with a small number of dimensions and simple fitness functions, the CUDA implementation showed modest performance gains. However, due to the overhead of copying data to and from the GPU, the multi threaded solution outperformed the CUDA solution.
- For problems with a large number of dimensions and more complicated fitness functions, the CUDA implementations showed incredible performance gains, outperforming the multi threaded solution.

Because of these characteristics, there is much potential in using similar GPU based approaches in solving complicated optimization problems.

VII. DISCUSSION

Due to time constraints, this work had the following limitations:

- The accuracy of the three algorithms was not presented. Due to the random nature of the PSO algorithm, the outputs were different every time, and sometimes the algorithm would fail to find the correct minimum or maximum. However, since the core algorithm was mostly unchanged across implementations, it can be argued that the speedup results are valid.
- The velocity recalculation portion of the PSO algorithm could have also been ran on a CUDA kernel, which would have provided an even better speedup. This could be done in a future enhancement to the algorithm.

VIII. RELATED WORKS

In 2009, Veronese et all developed the first variant of the PSO algorithm that was accelerated with CUDA. [11] The same year, Zhou and Tan developed a novel approach to accelerate standard PSO using CUDA. [12]. In 2011, Mussi et all developed an asynchronous version of CUDA PSO. [13]

REFERENCES

[1] James Kennedy, "Particle swarm optimization," in *Encyclopedia of Machine Learning*, pp. 760–766. Springer, 2010.

[2] Nvidia, "Cuda toolkit documentation - a general-purpose parallel computing platform and programming model," March 2015.

[3] Christoph Kubisch, "Life of a triangle - nvidia's logical pipeline," March 2015.

[4] Nvidia, "Cuda toolkit documentation - from graphics processing to general purpose parallel computing," March 2015.

[5] Nvidia, "Whitepaper - nvidia geforce gtx 980," 2014.

[6] Nvidia, "Cuda toolkit documentation - thread hierarchy," March 2015.

[7] Mark Harris, "Maxwell: The most advanced cuda gpu ever made," September 2014.

[8] Cyril Zeller, "Cuda c/c++ basics," 2011.

[9] Nvidia, "Cuda toolkit documentation - best practices guide - memory optimizations," March 2015.

[10] Guopu Zhu and Sam Kwong, "Gbest-guided artificial bee colony algorithm for numerical function optimization," *Applied Mathematics and Computation*, vol. 217, no. 7, pp. 3166–3173, 2010.

[11] L de P Veronese and Renato A Krohling, "Swarm's flight: accelerating the particles using c-cuda," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE, 2009, pp. 3264–3270.

[12] You Zhou and Ying Tan, "Gpu-based parallel particle swarm optimization," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE, 2009, pp. 1493–1500.

[13] Luca Mussi, Youssef SG Nashed, and Stefano Cagnoni, "Gpu-based asynchronous particle swarm optimization," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 1555–1562.