Genetic algorithms – CPU vs GPU implementation discussion

Metaheuristics course report

Adrian Horga

Introduction

Since their inception in the 1970s, Genetic Algorithms' uses have switched from the need to understand adaptive processes of the natural systems to being used for optimization and machine learning ([1]).

Since these types of algorithms require a high degree of computations, it is interesting to see how a normal implementation done using a Central Processing Unit (CPU) works and how long it takes. Furthermore, there are other hardware that can be used to run such a Genetic Algorithm implementation. One example of such hardware are the Graphic Processing Units (GPU).

In this report I will discuss a little bit about the difference about CPUs and GPUs, what tools one can use to develop a Genetic Algorithm and what are the strengths and weaknesses of each hardware when it comes to developing such an algorithm.

I have implemented a Genetic Algorithm example on the CPU and on the GPU and I will discuss the results and the conclusions in the later chapters of the report.

CPUs and GPUs

CPUs are, as the name says it, the central processing units of a usual computer. They are the ones that control and compute the data provided by the other devices in a computer. Since CPU need to control various types of hardware and data, they are built in a complex manner, and they focus on doing most general operation as fast as possible. Therefore CPUs are built for fast response time (low latency), by having a big cache and complex Arithmetic Logical Units (ALUs). Since the global memory is slow to read and write, caches are used as an intermediary hardware to speeed up this task. When a request is done to the global memory, if the data is already in the cache it is read from there much faster. If it is not, it will be first read from global memory, then written into the cache and then given to the requesting logical unit.

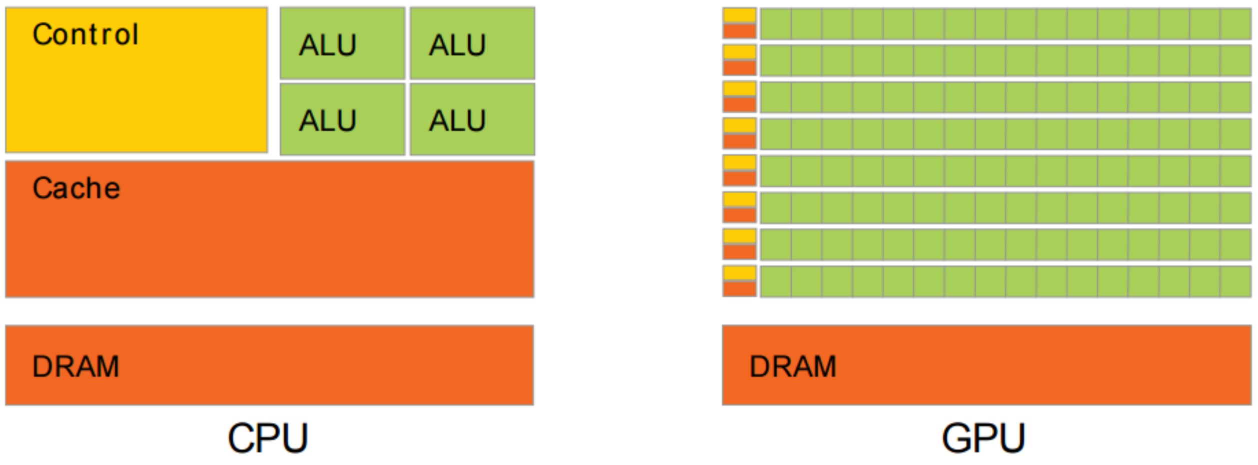

Figure 1: CPU vs. GPU [3]

The GPUs are designed differently than CPUs, as seen in Figure 1. They do not focus on low latency, but on high throughput. Their initial goal was to render images, more specific the pixels of an image. Since each pixel's value can be computed independently from the others, the GPUs focus on calculating as much data as possible. So, they have more hardware allocated for computation and less space for faster memories like caches.

In the past, GPUs were used to render images, but now one can make use of their high computing capabilities in general purpose computations. The main requirement for these is that they should be highly parallelizable. By parallelizable I mean that the elements that are calculated are independent of each other, so an element should not need to wait for another to have its value calculated before it can calculate its value.

CPU code is written mostly for sequential run (on one thread) and a common programming language is C/C++ ([2]). For GPU programming there are available extensions to this programming language or even others like (Java, Python etc.) in which programmers could write and run their code. These programming platforms are NVIDIA's CUDA [3] or OpenCL [4].

There are other programming paradigms available, but I will discuss the Genetic Algorithm usage by using an example written in C++ for CPU and in CUDA for GPU.

In CUDA a programmer will write a function named kernel which will run on the GPU. The difference from the CPU is that the respective kernel will be executed by thousands or even millions of threads. The programmer can specify the number of blocks and the number of threads per block that he want to be run. The threads in a block will run all run on the same core. There is also the concept of a warp, in which 32 threads run in lockstep. The warp is important because 32 threads will be fed the same instruction but with different data (corresponding to each thread), which speeds up the running process. The issues here is that if the instruction is an "if" statement and some threads in a warp need to take one branch in the execution path the the others take another, the instructions will be fed sequentially, meaning some threads in the warp will wait for the others to finish their branch and only after that they will execute their part of the branch.

The most used type of memory in the GPU is the global memory (DRAM in Figure 1.), that is the largest memory available in the GPU and where the data is being stored usually. The problem is that reading from global memory take a lot of time. GPUs have a small amount of shared memory available to each streaming multiprocessor (SM) and is orders of magnitude faster than the global memory. The issues with it are the reduced size and the fact that the data needs to be stored from global memory there and the threads in a block need to synchronize in order to be able to read coherently from it. I will have two versions of the GA algorithm, one without and one with shared memory.

Another concept introduced from CUDA 6.0 is the Unified Virtual Addressing (UVA) which means that from this version the CPU and the GPU can see all the memory available as a unified memory and they can access it without any issues. Before this, the programmer needed to allocate memory on the CPU, allocate memory on the GPU, transfer it to the GPU, compute it and finally transfer it back to the CPU for printing. With UVA , one only needs to allocate the memory as a general data, and the system will know when the CPU is using it and when the GPU is using it. I have used UVA to eliminate all the copy parts from our two versions

Genetic algorithm example

In this section I will talk a little bit about the GA example I have developed on the CPU and on the GPU. The basic form of a GA is as in Figure 2. I have an example from [5] which I have modified in order to work with pointers to characters ("char*") rather than strings as it was initially.

I have done this in order to prepare the algorithm to be ported to the GPU with CUDA since there are no provided libraries for string manipulation in CUDA.

The algorithm takes a target string and it will generate a random initial population each individual being a random string of the same length as the target string. The point of the algorithm is to get an at least one individual that has the same string as the target. The fitness is actually the sum of the absolute distances between the individual's character values and the targets characters values. The initial parameters that can be set for this example are the target string, population size, maximum number of iterations, elitism rate and mutation rate.

After the fitness is calculated for each individual, the list is sorted from the best to the worst. Based on the elitism rate set at the start of the run, the algorithm picks that percentage of individuals starting from the start of the sorted list.

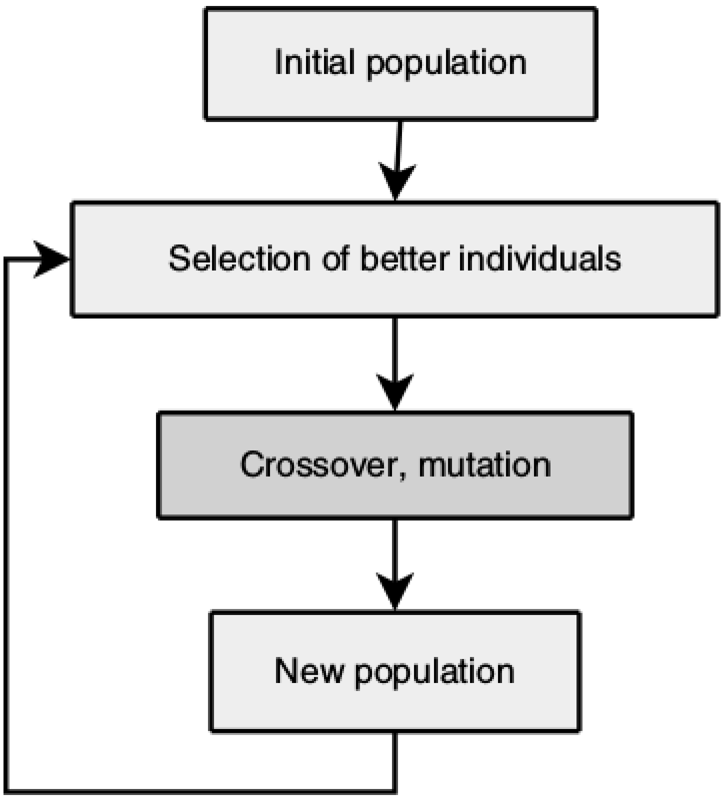

Figure 2: Genetic Algorithm structure [1]

The rest of the list is populated by doing crossover from two parents and the result is stored in the population list. Based on a random chance (set at the start), the new individual will also be mutated. The mutation will be done by changing the value of one of the individual's characters from the string with another random one.

The maximum number of iterations is set so that if after that specific number of runs, the algorithm does not produce a fit individual (fitness zero), then the genetic algorithm will stop.

In order to implement a part of this algorithm in CUDA I have to find a large number of computations that can be computed independently. This is the case in the calculation of the fitness, since every individual must compare its char string to the target one and does not need to know about the other individuals.

The first implementation in CUDA uses this as a principle, each thread is assigned to compute the fitness of an individual. So we will start as many threads as the population size.

The second implementation uses shared memory. This implies that, in a block, all threads will work together at the start to load the necessary data in the shared memory. Since there is only a limited amount of shared memory, we can only load the data from one individual when a thread block is run. After the data is loaded, using the reduction operation. I have implemented this reduction based on the information from [6].

I have attached the code for all 3 versions (CPU, GPU, GPU_shared) to this report.

Discussion

I have tested to see what are the different running times for each implementation of the fitness function calculation. The tests were done on 64 bit Ubuntu 14.04 machine, with and i5 4210H CPU , 12 GB RAM, and a NVIDIA GTX 960M GPU with 2 GB RAM.

We can see the results of running the fitness function in Table 1. I have ran the GA with a population size of 16834, 100 maximum number of iterations, 0.1 (10%) elitism rate and a 0.25 (25%) mutation rate. And the target string was "Hello World! Hy!!".

	CPU (ms)	GPU (ms)	GPU_shared (ms)
Fitness function	76	8555	8220

Table 1: Fitness function execution

Even though this might seem unnatural that the CPU version is orders of magnitude faster than the GPU one, there are some explanations.

First of all, the first problem is the branch divergence, which I have mentioned previously. Since GPU cores are small and have low computational complexity, having "if" statements (in this case the "abs" function call) leads to the doubling of the execution time when it comes to that single instruction.

Another issue is the copying of the data from CPU to GPU and back. Even thoug I am using UVA, the copying is implicit and it is done by the CUDA framework behind the scenes.

The shared memory is decreasing the running time but not by much, this is because once the data is loaded into shared memory there are not so many computations that need to be done with it, so its potential is being wasted.

The next issue is the problem size. Since we don't have so much memory on the GPU to compute millions of individuals, therefore millions of threads, the potential provided by the GPU is again wasted. A simpler gene for the individual which takes less memory space would allow the increase in the population size from thousands to millions.

The last and most important issue with the GPU version is memory coalescing. The way GPUs work with threads is that if each thread can read the next data element from global memory space then the reading speed is optimized. This is not the case here since each thread will need to read the first charater of the gene, then the next, etc. This is a problem since the first characters of each string are not one after the other because of the way the memory is saved. I could invert it, but then I would need to invert it back each time I copy back to the CPU so that the sorting can be done.

Conclusions

Even though there is a possibility to improve the algorithm, and there exist versions of GA that are faster on the GPU than on the CPU ([8,9]), these have to be specifically tailored for the GPU hardware. The problem that needs to be solved needs to be as perfectly fitted for the GPU as it can be, and sometimes it comes at the expense of precision.

The problem with the GPUs is that they are not so flexible when it comes to memory patterns and complex computations involving branches. The GPUs work perfect for problems that have a lot of simple calculations per each element and few needs to access global memory.

References

[1] Metaheuristics: from design to implementation, E.-G. Talbi, Wiley, 2009

[2] http://www.cplusplus.com/

[3] http://docs.nvidia.com/cuda/

[4] https://www.khronos.org/opencl/

[5] http://www.generation5.org/content/2003/gahelloworld.asp

[6] http://developer.download.nvidia.com/compute/cuda/1.1/Beta/x86_website/projects/reduction/doc/reduction.pdf

[7] Petr Pospichal, Jiri Jaros, Josef Schwarz, "Parallel Genetic Algorithm on the CUDA Architecture", Aplications of Evolutionary Computations, Volume 6024 of the series Lecture Notes in Computer Science, p. 442-451, 2010

[8] Sifa Jang, Zhenming He, "Implementation of Parallel Genetic Algorithm Based on CUDA", Advances in Computation and Intelligence, Volume 5821 of the series Lecture Notes in Computer Science, p. 24-30, 2009