Gacuda - a Genetic Algorithm Framework over GPUs

Bruno Miguel Morgado Martins
bruno.martins@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2016

Abstract

Graphics processing units (GPUs) have been evolving rapidly since their first appearance. Initially only destined to provide hardware acceleration in 2D and 3D graphics calculations, they can now also be used to accelerate generic calculations, traditionally reserved to the central processing unit (CPU) of the computer. However, programming for GPUs requires knowledge of the architecture and their programming model. The effort required to obtain this skill set can mean that many times the GPU is not utilized in problems where it would obtain better performance than the CPU. Normally, due to the high number of processing units of the GPUs, problems where it is necessary to execute the same operation multiple times on different data have much higher performance when executed on a GPU than on a CPU. One example of such problems are the genetic algorithms, addressed in this work. This paper presents Gacuda, a framework for developing genetic algorithms running over GPUs. It allows the use of the GPU to execute the genetic algorithms in a simpler way, hiding the complexity of programming for the GPU to the developer, while at the same time maintaining the performance benefits of executing on the GPU. In terms of code, all that is required is the development of a few functions in completely regular C programming language, without GPU details.

Keywords: GPU, genetic algorithm, CUDA, framework

1. Introduction

Genetic algorithms [1] are a family of search heuristics based on Charles Darwin's theory of evolution, belonging to the broader class of evolutionary algorithms and machine learning algorithms. They are used in a large number of problems in various fields such as economics, molecular chemistry and computational physics.

Through genetic algorithms it is possible to reach non optimal solutions in computationally demanding problems where the time required to solve them and obtain the exact solution might not be acceptable. Yet, in many of those cases, a good but not perfect solution suffices. An example of the kind of problems is the broad class of NP problems [2] where the time required to find the optimal solution is superpolynomial to the size of the input.

These algorithms work by generating a group of random individuals known as the population. Each individual is represented by a single solution. These individuals encode the solutions in their genes or, as it is also known by, their genome. Having all the population generated, the algorithm works by making each individual evolve by crossing genes with other individuals and through gene mutation. Afterwards, a natural selection pressure is applied to this newly evolved population. This selection is done by means of a fitness function which evaluates each individual's genetic material. As a result, the fittest individuals are selected and the others are discarded. Due to the natural selection pressure the individuals will evolve towards the optimal solution of the problem.

Genetic algorithms require the fitness function to be evaluated for each individual of the population in every iteration of the algorithm and the fitness function calculation can be a very time consuming process. It makes sense to parallelize this process in order to obtain faster results. With the amount of multiprocessors in a GPU [3], each with SIMD (Single Instruction, Multiple Data) parallelism, executing the same operation on different locations of memory at the same time, this process should run faster on it than on a CPU.

In the past few years, general-purpose computing on graphics processing units (GPGPU) [4] became easier with the increase of support tools and quality of the APIs (Application Programming Interface) provided by the GPU makers themselves. An example of such API exists in the CUDA (Compute Unified Device Architecture) [5] platform provided by the company NVIDIA, famous for their high-

performance GPUs. CUDA was the software chosen to develop the framework presented on this thesis. Although GPGPU is much more accessible than before, it still requires the programmers to understand the GPU architecture and the programming model of the API used. The overhead of having to learn about it can ward off many programmers of harnessing the power of GPUs. Furthermore, in order for the GPU implementation to perform well, the applications to be developed must be constructed in a way to maximize all the GPU's cores usage while minimizing data transfers between the CPU and the GPU due to the high latency of such transfers.

Genetic algorithms are good candidates for having remarkable performance while processed in the GPU. This due to the fact that in genetic algorithms the same operation is executed multiple times independently for each individual of the population and this type of processing is exactly what the vectorial processing provided by the GPU excels at. Since a GPU can have a speedup multiple times greater in relation to CPUs, it makes sense to execute genetic algorithms in the former.

A greater performance in genetic algorithms translates to better solutions, as well as in a faster way in obtaining them. The main motivation for this work is therefore to harvest the power of the GPUs in every way possible in order to solve genetic algorithms faster and with better solutions. At the same time, it is interesting to allow people without knowledge of GPGPU programming to harness this power as well. This is the secondary goal of this project.

The main objective of this work was the presentation and development of Gacuda, a genetic algorithm framework running on the GPU using the CUDA API. This framework was to be completely generic, supporting a wide range of genetic algorithm problems by allowing the user to define the specific components (fitness, crossover and mutation functions as an example) of each problem. The framework should also be as time efficient as possible, using the most out of the GPU resources. The framework should also allow people without GPU developing knowledge to use it, hiding the complexity behind GPU programming through the developed API, which requires only the development of a few C functions, without GPU interaction, resembling traditional single threaded CPU programming.

This work also intends to demonstrate that the developed solution fulfils the proposed requirements and, as such, several example problems were implemented using the framework and the results measured and documented.

2. Genetic Algorithms

Genetic algorithms are generic search techniques based on natural selection and evolution. Solutions to the problem at hand are encoded into individuals and then these individuals are made to combine and mutate. This way, with the correct natural selection pressure, the individuals and solutions they represent will evolve towards the desired optimal solution of the target problem. Genetic algorithms can be used to solve hard to compute problems heuristically, obtaining good solutions to those problems albeit not guaranteed of reaching the optimal one.

2.1. Overview

A genetic algorithm starts by randomizing its initial population in a uniform fashion. The population is a group of individuals, each with its own genetic code. To the whole genetic code of an individual is given the name genome or, depending on the author, chromosome or genotype [6]. Each genome encodes a solution for the problem being treated by the genetic algorithm and is composed of a number of genes, each representing a parameter for the solution of the problem.

After the initialization, the whole population fitness is measured, individual by individual, through a fitness function, the objective function which the algorithm will try to maximize. With the fitness values obtained, natural selection is simulated through the execution of the selection code. Here, based on the fitness value of each individual, the individuals that will crossover and pass their genes to the next generation are selected. Crossover is akin to natural reproduction.

In order to introduce some genetic diversity through the generations, some randomly chosen individuals suffer mutations in their genetic code. This makes the algorithm explore new zones of the solution space and therefore obtain better solutions, avoiding local maxima.

With the new offspring generated replacing the old generation, the algorithm tests for the termination condition. If the termination condition is not verified then the algorithm goes back to calculating the fitness of every individual and generating a new population again. Otherwise, if the termination criteria is met, then the algorithm ends with the most fit individual containing the best solution to the problem.

2.2. Encoding

The first step to implement a genetic algorithm to solve a specific problem should be choosing the encoding used, that is, the amount of data stored for each individual and how this data is organized. There are multiple classic options to choose from, it is up to the developer to choose the most fitting encoding to the problem at hand.

2.3. Fitness

Having the encoding selected, it is necessary to develop a fitness function. The fitness function is the objective function which the algorithm will try to maximize. It is a function that will receive an individual as a parameter and, based on the individual genome, will return a numeric value that measures how fit the individual is to the problem being solved. The fitter an individual is, the better the solution to the problem will be. With the fitness function defined, the genetic algorithm will then try to maximise (or minimize, by simply reversing the sign of the fitness function results) the fitness of the population. This is done in order to obtain one single individual with the best fitness of the population and the best solution for the problem.

The fitness function will be executed once for each individual of the population in every iteration of the algorithm, meaning that speed of execution is a very desirable characteristic. For that reason, for some problems where computing the exact fitness of an individual takes an excessive amount of time, a fitness approximation function is used [7].

2.4. Selection

Selection is the phase in the genetic algorithm where the individuals that get their genes passed on to the next generation are chosen. It is based on the fitness value of each individual of the population. This is the phase of the algorithm where evolutionary pressure is exerted such that the individuals converge to the solution of the problem at hand. Passing on the genes to the next generation can happen through either passing the full individual genes unmodified or by passing part of the individual genes through crossover (see section 2.5).

Tournament selection [8] is fundamental for Gacuda. It starts by randomly selecting a number of individuals. Usually the number of selected individuals is two, making it a 2-tournament but other values can be parametrized. After having selected the random participants, their fitness values will be compared and the highest fitness individual will be selected. This process is repeated until all the desired individuals are selected. The number of participants in the tournament is pre-defined and will impact the speed of convergence of the solution.

Elitism [9] is not a selection method on its own. It is an additional selection method that guarantees that the best individuals in every iteration survive and make it to the next generation unchanged. This feature is often used in genetic algorithms because it speeds the convergence to better solutions and makes it so that the algorithm can not lose the best solutions and walk backwards.

2.5. Crossover

Crossover is an operation on genetic algorithms where the genetic code from multiple individuals is mixed to generate new offspring. It is analogous to biological reproduction. Two or more individuals are picked and then their genes are used to create a single individual. The crossover operator is the main responsible for the new individuals and respective solutions. This is one the main parameters of genetic algorithms and it has to be tuned according to the problem at hand and crossover function used.

2.6. Mutation

The mutation operator acts by modifying individuals genes. Its purpose is to introduce genetic diversity. This makes the algorithm converge more slowly but helps avoid local maxima by making the algorithm explore new solutions and, consequently, a wider solution space.

2.7. Termination Criteria

Termination criteria is the condition that is verified after each iteration of the genetic algorithm and that will determine if the algorithm has finished.

3. General-Purpose Computing On Graphics Processing Units

As processors evolve and transistor size diminishes, new problems related to heat dissipation arise. These problems are the reason why we no longer see the doubling of single core performance and frequencies clock of general purpose processors every couple of years any more. Engineers designing processors have to resort to different techniques to increase their performance. One of those techniques is adding additional cores to process data in parallel. Albeit having multiple cores, general purpose processors just cannot compete with GPUs in terms of floating point operations per second, when the right conditions for paralellization are met [10]. This is natural since general processors main focus is single core performance and GPUs are designed for massively parallel compute intensive applications. The GPU dedicates much more of its transistors to arithmetic calculations with the CPU focusing much of its die space on a more complex control unit and on bigger cache size [11].

It is expected that GPUs will continue to outperform CPUs in these conditions for the close future, according to the current trends, where the difference in floating point operations appears to be increasing, favouring GPUs. This makes GPUs attractive for solving intensive computational problems and that is what GPGPU allows.

GPGPU development is now much easier than it was in the past. Not only have the GPUs improved in order to facilitate this, but also documentation is

nowadays more widely available. In terms of tools there were great advances too, namely frameworks with debugging and detailed profiling support and various libraries with common tasks in the GPGPU world. Currently the most used frameworks are OpenCL [12] and CUDA [5].

Despite being an open standard, OpenCL does not have all the features that CUDA provides, specially in profiling, and that is why CUDA was picked for this work.

3.1. NVIDIA GPU Architecture

The GPU is composed of a variable number of computing devices depending on the GPU model. To each of these computing devices the name Streaming Multiprocessor (SM) is given. Besides the SMs, the GPU contains gigabytes of high latency GDDR5 DRAM memory referred as global memory. It also contains a smaller but lower latency L2 memory cache. There is also a scheduler denominated GigaThread global scheduler that distributes blocks of threads to process to the SM schedulers. The GPU communicates with the CPU through a PCI-Express interface.

Each SM contains a variable number of processing cores, usually 32, but this number can be higher depending on the model of the GPU. Each SM contains a low latency memory block of around 64KB (again depending on the model), to be split and used between L1 cache and shared memory. Any SM also contains a register file structure. This register file allows for the use of multiple registers that have very little latency and consequently very high bandwidth. The cores of a single SM share the registers between them. Exhausting all registers causes register spill, meaning further additional registers will not be used, instead data will be stored in the slower global memory.

The cores of a SM contain two important components, the arithmetic logic unit (ALU) and the floating point unit (FPU). The cores are responsible for processing the data scheduled by the SM in a synchronized fashion within the SM along with his companion cores. They do this through the single instruction, multiple thread (SIMT) model, meaning that one particular instruction is executed by a thread in every core of a SM at the same time. The SM schedules a group of 32 threads, called a warp, at a time.

Knowing the NVIDIA GPU architecture one can see that to perform well a GPU application should seek to make use of all the SMs and respective cores by scheduling a high enough number of threads. Also, memory accesses should be minimized or at least low latency memory use maximized, so that the cores can be working all the time instead of waiting for the data to be available. Finally, the code running in the GPU should minimize conditional instructions that can cause branch divergence between the threads in a warp. Since the threads in a warp must all execute the same instruction at the same, such branching impacts performance significantly.

3.2. CUDA

CUDA [5] is the platform provided by NVIDIA to develop GPGPU applications on their GPUs. To develop applications for CUDA the user has two options. To use the low level CUDA Driver API or the higher level CUDA Runtime API. For this work the latter was chosen for its simplicity and for being more commonly used and therefore having more available knowledge online.

4. Implementation

In this section, a detailed explanation is made about the implemented solution called Gacuda, a genetic algorithm framework that runs on GPUs. A detailed overview of the framework is given and the additional features implemented are described. Additionally, it is also explained how to develop for the framework.

4.1. Architecture

Gacuda is the framework created that allows the execution of genetic algorithms in the GPU. It was developed with the intent of supporting different genetic algorithms problems. With that intent, the minimum requirements needed to define a genetic algorithm problem were investigated and the conclusion was that the fitness function and the crossover and mutation operators were necessary. Taking that into account, in Gacuda, what is required from the user is the definition of the genetic algorithm operators (crossover and mutation) and the fitness function in completely regular C programming language. Besides that, the user only needs to set the genetic algorithm parameters in a configuration file. The user code will then run in the GPU orchestrated by Gacuda code.

Initialization of the genetic algorithm population could be made automatically, generating random genes for all the individuals but, for more complex cases, it may be important for the user to control the initialization of the genes of the population. For this reason, it is also necessary to define the initialization function when developing an application with Gacuda.

Selecting individuals to crossover is done automatically by the framework. The selection method implemented is the tournament selection. This method was chosen not only because it is easy to implement in parallel, but also because its implementation performs well in GPUs. Tournament selection requires only picking two random individuals from the population and comparing their fitness for

electing a parent. This process has to be repeated twice, once for each parent of the new individual.

In order to obtain the most out of the GPU, the results from several genetic algorithm GPU implementations were investigated. Two common approaches exist. One runs the algorithm sequentially in the CPU with the exception of the fitness function, which is commonly the most resource intensive operation of the whole execution. The fitness function is executed in parallel in the GPU. Unfortunately, this approach requires reading the whole population from the CPU memory every turn, resulting in poor performance [13]. The other GPU genetic algorithm approach executes the whole algorithm in the GPU in parallel. This way the CPU memory can be read only once at the beginning of the algorithm, with the whole population being copied to the shared memory of the GPU. This way further accesses to the population genes are extremely fast. Two main problems with this approach is to provide information to the user while the application is running and the population not fitting the shared memory. The latter situation is quite common since the amount of existing shared memory per SM is very small.

The Gacuda approach is somewhere in between the two approaches previously described. The code runs mostly in the GPU. Synchronization with the CPU occurs repeatedly every couple of steps (parametrizable). This allows displaying to the user the current best solution found until that point of the genetic algorithm. It offers the possibility, for example, to visually display the population while the application is running on the GPU. Another benefit of this stepped approach is that, in this way, it is possible to utilize multiple GPUs using the pause in the GPU execution to migrate individuals from one GPU population to other.

After defining the fitness and other necessary functions, the user needs to execute the build script to link the custom code with the framework code. This way, when the user executes the generated binary (his/her application), the framework code will run and will eventually use the user-defined functions when appropriate.

4.2. Execution Flow

When a Gacuda application is executed, it starts by running the main function code from the framework. This main function will read the configuration file for the application whose filename was passed on the command line as the first and only parameter any Gacuda application receives. This configuration file contains multiple settings for the execution of the application. These include the population size, memory size of each gene and crossover rates, among others. This configuration file also allows for the activation or deactivation of certain features of the framework.

After reading the configuration file, the framework executes the user-defined initialization functions where the user is supposed to generate a random population. This is performed on the CPU side for simplicity. Since initialization occurs only once, the benefit for running it on the GPU side would not be comparable to the benefit of executing the fitness function on the GPU.

With the initialization complete, the framework will schedule the execution of the genetic algorithm in the GPU for a configurable amount of steps. To do this, the framework must first copy the population from the computer main memory to the GPU global memory. Some settings read from the configuration file are also passed to the GPU. In order to optimize their access time, since these settings are read-only, they are put in the GPU constant memory.

The GPU kernel will then finally start executing. A thread will be created for every individual of the population. This means that in order to use most of the GPU resources, population size must be set to a number high enough, depending of the GPU model. Mapping one thread per individual makes it so that the maximum number of individuals is limited to CUDA's maximum number of threads. This number depends on the compute capabilities of the GPU device but it is generally quite high.

The first action of the kernel code is setting up the thread pointers to the corresponding individual, based on the thread and block identifiers. Also, if the population fits in the SM shared memory, then it will be copied from the global memory to the shared memory with each thread copying the respective individual. Otherwise, if the shared memory is not large enough, the individual is accessed from the global memory directly.

After initializing, the kernel will enter in a loop that will run a user definable number of times, referred to as steps. In this loop, the kernel will execute the basic genetic algorithm. It starts by executing the user-defined fitness function over the thread individual genes. Then, it executes the framework tournament selection twice for each thread, by picking four random individuals. The winners of the tournament will be the parents of the new thread individual if, depending on the crossover rate, it is randomly selected for a thread to overgo the user-defined crossover operation. Whether there was crossing of individuals or not, depending on mutation rate, it is decided if each thread individual goes through mutation or not. In an affirmative case, the user-defined mutation function is executed with the resulting individual passing on to the next generation. Afterwards, a test is made

to see if the desired number of steps were already completed. In a negative case, then the loop is repeated with the fitness function being executed once again. Finally, if all steps were executed and the whole population fits in the shared memory, then each thread copies its respective individual from the shared memory to the global memory ending the kernel execution.

With the end of the kernel execution, the algorithm checks for the termination condition of the algorithm that currently can only be the total number of generations. In case the algorithm has not ended, there is the opportunity to print some information back to the user about the current state of execution of the application. Such information includes the current best fitness and how many iterations of the algorithm been processed. If the algorithm has ended, then additional information will be provided to the user. If the configuration file is parametrized for such, Gacuda will write the whole population and each individual fitness to files. Of course, the best individual fitness is also printed and the best individual can be displayed through a user-definable function.

4.3. GPU Mapping

In Gacuda each thread is responsible for calculating the evolution of a single individual. This means that each thread run will start with an individual and at the end of its execution the result will be an evolved individual.

For mapping the threads in CUDA, Gacuda follows the Island Model. It divides the population evenly in a number of blocks. Each population block is processed in isolation by a single SM, with selection and crossover occurring only between individuals within the same block. The size of each block is configurable by the user through the configuration file. The number of existing blocks is determined from the ratio between the size of population and the size of each block, dividing by the number of existing GPU devices. Both the *population_size* and *threads_per_block* parameters are specified in the configuration file.

In Figure 1 it is possible to see a concrete case of the mapping of the Gacuda population to the GPU. In the example represented, the population size is 100 individuals. With a configured 25 threads per block, the number of blocks is calculated and this will result in 4 islands to process. Each island will then be processed exclusively by a single SM.

4.4. Shared Memory

Since accessing shared memory is much faster than using global memory, a mechanism to maximise the use of shared memory was developed for Gacuda. After initialization and before the first GPU kernel is scheduled, the framework does one test based on

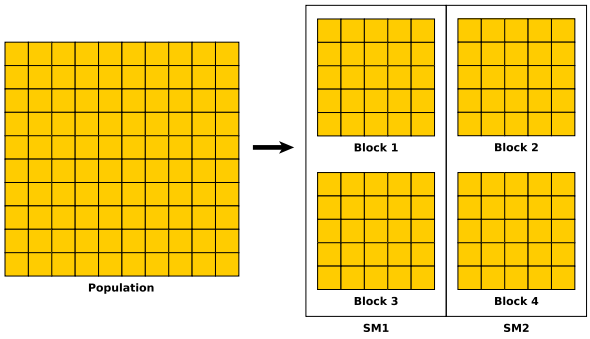

Figure 1: Gacuda GPU mapping.

the settings of the configuration file. The framework will determine if two whole populations (double-buffering) can fit on the shared memory of the GPU calculated through CUDA device information functions. If the population fits in shared memory then no global memory accesses in the CUDA kernel are made with the exception of the reading of the population at the beginning of the kernel and writing the population to global memory at the end when all iterations have been processed.

4.5. Migration

Trading genetic information between islands is done through a migration process. Migration of individuals between islands in parallel genetic algorithms results in better solutions [14]. So, since the population is divided in blocks when a kernel is scheduled to run and islands can not interact with other island's individuals efficiently through the use of shared memory, a way of trading genetic information between islands was developed for Gacuda. The developed mechanism occurs every few iterations (it is a number parametrizable in the configuration file). When bound to occur, the intra-migration process will basically copy individuals from one block to another. This occurs at the Gacuda kernel, where the first thread of a block will copy individuals from the current block shared memory to the global memory portion of the next block. The blocks are organized topologically in a ring for the purpose of migration, as seen in Figure 2.

In the same figure, it is also possible to see the effects of one migration step on the population and, in that case, how the first three individuals from the start of an island are copied to the next.

4.6. Multi-GPU / Inter-Migration

Gacuda supports multiple homogeneous GPUs. In such cases, following the island model, the total population is divided by the number of GPU devices with each device with an equal number of individuals. Instead of a single kernel call, the framework

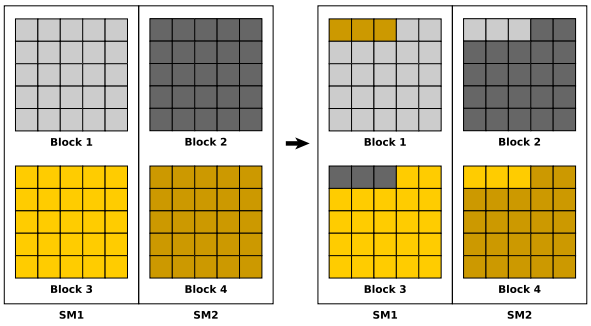

Figure 2: Gacuda Migration.

will issue one kernel call for each device and each kernel will process its corresponding part of individuals. At the end of each kernel execution, Direct Memory Access (DMA) is used to copy a portion of the beginning of one device population to the end of the next device population, in a ring topology. This is the only way that two different GPUs have, in Gacuda, to share genetic information between them.

4.7. Pseudo Random Number Generation

CUDA kernel functions can not invoke Standard Library *rand()* function to generate pseudo random numbers. It is possible, though, to generate pseudo random numbers through a CUDA library called cuRAND [15] although with relatively high computing cost. Nevertheless, for Gacuda a simple pseudo random number generator known as the linear congruential generator algorithm was implemented. This choice was made because this algorithm is very simple to implement and lightweight to compute. Although not suitable for cryptographic applications due to the predictability of the generated numbers, it is good enough for Gacuda genetic algorithm purposes.

A random number generator macro function is provided for the user to apply when programming the crossover and mutation functions.

4.8. Elitism

Elitism selection is guaranteeing that the individuals with the highest fitness remain on the population of the next iteration of the genetic algorithm. In Gacuda a similar feature was implemented that allows keeping the fittest individual of each block of threads. This was implemented through a block level reduction, comparing the previously calculated fitness values and copying the most fit individual to the first position of the array of each thread block. Since there is minimal synchronization and there is no need to sort the fitness values, as is done in other works, this method uses very little GPU resources at the expenses of only being able to save the best individual.

4.9. Statistics

Since the execution of a genetic algorithm can take quite some time, it would be interesting to have feedback of the progress of the application either with the current iteration out of the total of iterations or with the best fitness value. The former is easy to obtain and output while the latter requires comparing all the fitness values and storing that value. Gacuda supports displaying both in every few iterations, when the GPU ends executing the main kernel and the control returns to the CPU. This is done through the execution of a statistics gathering kernel in the GPU. This kernel is scheduled in the same way as the main kernel, with a thread responsible for a single individual of the population. The kernel performs three atomic operations in order to obtain the best, worst and average fitness values. The atomic operations are *atomicMax*, *atomicMin* and *atomicAdd*, respectively.

4.10. Macros

The configurations set through the configuration file (e.g. the population and genome size) are available to use in the user code through the use of C language macros. In order to use them, the user only needs to include the file *gacuda.h* in his code. This configuration data is stored in the GPU constant memory for faster access.

Additionally, a pseudo random number generation function macro working for both CPU and GPU code is available through *RAND()*.

4.11. Creating an Application

Creating an application using the Gacuda framework is simple, as it is only necessary to develop a few C functions and create a configuration file. This process is documented in this subsection.

With the encoding chosen, the user needs to implement a couple of C functions in a *.cu* file to develop an application using Gacuda. With the exception of the special CUDA keywords used at the function declaration, the code is straightforward C language code oblivious of any GPU and CUDA intricacies.

There are two functions that must be developed regarding initialization. Their purpose is the same, that is, to initialize the initial population but they work in different ways. The function *initialize_gene* is called for every gene and the user must only set the gene's data through the pointer received by the function. As for the other function *initialize_population*, the user must iterate through all the genes of the population and initialize them, since this function is called only once.

The fitness function is responsible for measuring how fit an individual is. In Gacuda, the fitness function receives a pointer to an individual's genes and is expected to return a float with the evaluated fit-

ness.

The crossover function receives two pointers to the genes of both parents and has to, using these, write the resulting genes of the crossing into a third pointer.

The mutation function receives a pointer to an individual's genes. Afterwards, the genes are to be mutated according to chance.

The output function is an optional function whose purpose is to allow representing an individual in a human perceptible way, so it can be displayed to the user through the whole execution of the application. It allows to display the best individual at the end of the genetic algorithm.

With the code developed, the next step is filling the necessary fields in the configuration file. This file allows the change of settings without the need to recompile the code.

The compilation process of a developed application is quite simple. The user needs only to modify the provided sample *Makefile* file, indicating the name of the target binary file to be generated. There must exist a corresponding user *.cu* file containing the Gacuda required developed functions. When executed, the *Makefile* will run the CUDA Compiler *nvcc*, which will compile the Gacuda framework using the code provided by the user into a single executable file.

After building the executable file from the compilation process, running the application only requires the user to specify the correct configuration file in the command line.

5. Results

Several applications were developed using Gacuda to demonstrate that the framework is truly generic and supports several different types of problems. In this chapter, it is made a short presentation of the problems implemented and the results of their execution. While presenting the various applications developed, the tests used are changed in order to present the effects of different configuration settings on the framework results. In the end, some tests results are presented to demonstrate the benefits of using Gacuda in the GPU versus a single-core CPU. It is also shown that GPU resources are fully utilized by the framework.

5.1. Case Study 1: Numerical Optimization

Two different numerical optimization applications were developed. Both are minimization problems and usually employed when testing a genetic algorithm numerical optimization prowess since their optimal solution is hard to obtain due to their intrinsic characteristics. The functions implemented are called Rosenbrock [16] and Griewank [17].

Two versions of the Griewank applications were developed. One working with single precision and other using double precision. The point of this was to compare the execution times between both versions. The execution times more than doubled for the same test conditions on the single precision application. This result is expected since the GPU's single precision processing power in relation to double precision is about the same ratio of the execution time differences presented.

For the Rosenbrock test, the selected variable parameter was the number of input variables for the Rosenbrock function. As for the results, increasing the number of variables, while keeping all the other settings constant, translates to higher execution times (since there is more to process) and to worst solutions (optimal solution is the value 0). This result makes sense and was predictable because increasing the number of variables is making the problem more arduous.

5.2. Case Study 2: 0-1 Knapsack

The 0-1 knapsack problem asks what is the combination of items, each having a certain value and weight, that should be chosen to fill a bag with a weight limit so that the most value is taken. It has applications in multiple fields and it is a NP-hard problem, having no known polynomial algorithm capable of solving it. This makes it a good problem to solve heuristically with, for example, genetic algorithms.

The knapsack application was executed five times for the same knapsack problem with 21 items. The application was able to obtain the optimal solution consistently in the very short execution time.

5.3. Case Study 3: Travelling Salesman Problem

The travelling salesman problem consists in trying to find the shortest route possible between a set of given cities, traversing any only once and ending at the starting point. It is an NP-hard problem, meaning that no algorithm running at polynomial time exists (so far) that can solve it optimally. Therefore, it is a good type of problem for genetic algorithms, which can solve it non-optimally in more acceptable times.

To test the algorithm, a set of problem instances were used. These belong to TSPLIB [18], a library of sample instances for the travelling salesman problem. In the first set of tests made regarding the TSP application, the parameter controlling the number of threads per block (and consequently the number of individuals per island) is increased starting from the value 4 up to the value 512. It could be seen that the optimal value to use for this parameter is not in the extremities. If set too low, the limit of concurrent number of blocks running in a SM will be reached and performance is lost. If set too high, the number of blocks can be low enough that not all SMs will be active, also punishing the perfor-

mance achieved. Changing the number of blocks and, therefore, the number of islands, also impacts the quality of the solutions. This is most likely due to the difference of the individuals' isolation according to the number of islands and their size.

For the second set of tests, the number of iterations simulated was varied. The execution time grown linearly with the increase of the number of generations.

5.4. Speedup

A version of the Griewank genetic algorithm application was developed to run exclusively on the CPU. This single-threaded application was then tested and the resulting execution times were compared with the matching Gacuda GPU application. The single threaded application simulates a single island with the size of the whole population and since it did not support elitism, the elitism feature was disabled in the GPU version of the application.

A speedup of around 29X was measured by the GPU application over the single-threaded CPU one, demonstrating the viability of a GPU genetic algorithm framework in terms of performance. Albeit a meaningful improvement, it is necessary to notice that the CPU version only used a single core and, therefore, the amount of performance gain could be reduced when comparing to a CPU parallel implementation.

The Figure 3 and Figure 4 present the results obtained when executing the Griewank applications on both the GPU and CPU, through a semi-logarithmic and a logarithmic graph, respectively, where the order of magnitude difference in execution times can be more easily seen. The GPU and CPU execution times evolve similarly, with the GPU performance improving slightly with more workload.

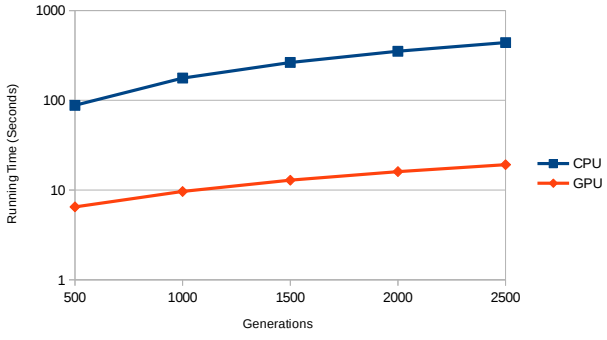

Figure 3: Griewank execution time for a given number of iterations.

5.5. Multi-GPU Speedup

To measure the framework performance gain of using multiple GPUs, additional tests were made. Has it can be seen on Figure 5, when using two GPUs

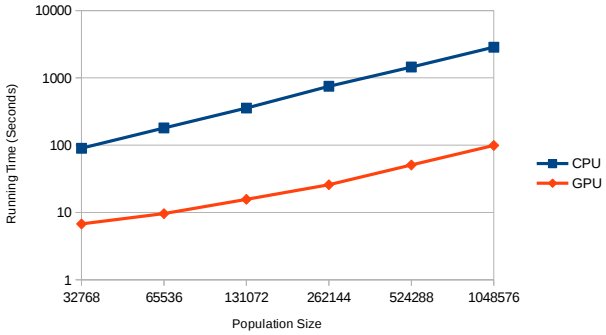

Figure 4: Griewank execution time for a given population size.

the execution time was reduced to almost half the time than when executing with only one GPU.

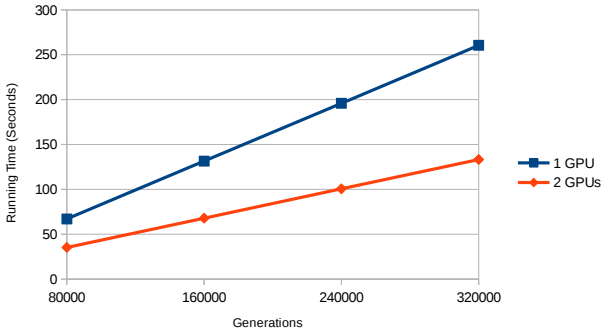

Figure 5: Execution time comparison between single and dual GPUs setup.

The speedup obtained was of 1.97X when executing with both GPUs against using only one GPU. With these results it can be concluded that the overhead of using two GPUs introduced by the framework is minimal.

6. Conclusions

This work presented Gacuda, a genetic algorithm framework over GPUs hiding the GPU complexity, requiring only for the user to write regular C code.

In terms of performance, this work shows that running genetic algorithms on the GPU through Gacuda is at least as viable as running them on the CPU, with the obtained speedup of up to 29X over the CPU single-core version (has seen by the results, this speedup can be almost doubled by adding an additional GPU). This means that the sampled Griewank implementation on Gacuda can obtain better solutions in a shorter time than by developing and running an application for the CPU.

Additionally, it was also shown that this framework can be generic. This is demonstrated through the development of three different types of genetic algorithms: numerical optimization of functions and two different combinatorial problems such as the travelling salesman problem and the 0-1 knap-

sack.

A performance analysis of the GPU code of the framework was also done, which showed that the GPU is kept busy most of the time, corresponding to over 99% of the execution time of the application. Besides that, the kernel code is also using all the SMs available a great part of the time, with approximately 94% of occupancy.

Finally, Gacuda confirms that it is possible to program applications using genetic algorithms for the GPU without almost any knowledge about GPGPU programming. The implemented sample applications use pure C language code in their entirety, with the exception of the reserved keywords in the functions declarations.

In the future, the framework performance could probably still improve after thorough profiling and consequent optimizations. For example, the elitism feature could be implemented through atomic operations instead of through a reduction operation, as is currently implemented.

Regarding hiding GPU details from the user, further improvements could also be made. For example, an automatism to detect the correct value, based on the computer hardware, for the number of threads per block to use. This would mean less one setting for the user to configure.

In terms of features more could be added too. For example, additional termination tests. As of now the framework only supports ending after a set number of generations. The other termination tests such as stopping at a pre-defined fitness level or halting after no improvement could be and interesting addition. Another feature that could be added would be overwriting the configuration settings through command line arguments. This way it would be faster to test several different settings since the configuration file would not have to be edited every time.

References

[1] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[2] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[3] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[4] Mark Harris. GPGPU: General-purpose computation on GPUs. *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[5] CUDA NVIDIA. Programming guide, 2008.

[6] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[7] David Beasley, RR Martin, and DR Bull. An overview of genetic algorithms: Part 1. Fundamentals. *University computing*, 15:58–58, 1993.

[8] Brad L Miller and David E Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.

[9] JA Vasconcelos, Jaime Arturo Ramirez, RHC Takahashi, and RR Saldanha. Improvements in genetic algorithms. *Magnetics, IEEE Transactions on*, 37(5):3414–3417, 2001.

[10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[11] John Owens. Streaming architectures and technology trends. In *ACM SIGGRAPH 2005 Courses*, page 9. ACM, 2005.

[12] Khronos OpenCL Working Group et al. The OpenCL specification. *version*, 1(29):8, 2008.

[13] Ogier Maitre, Laurent A Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410. ACM, 2009.

[14] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–48, 1999.

[15] CUDA NVIDIA. CURAND library. *NVIDIA Corporation, Santa Clara, CA*, 2010.

[16] HoHo Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, 1960.

[17] Andreas O Griewank. Generalized descent for global optimization. *Journal of optimization theory and applications*, 34(1):11–39, 1981.

[18] Gerhard Reinelt. TSPLIB - A traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.