Gacuda - a Genetic Algorithm Framework over GPUs

Bruno Miguel Morgado Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Carlos Alves Pereira Monteiro

Examination Committee
Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Nuno Cavaco Gomes Horta

May 2016
Resumo

Os processadores gráficos (GPUs) têm vindo a evoluir rapidamente desde o seu aparecimento. Inicialmente apenas destinados a providenciar aceleração por hardware nos cálculos de gráficos 2D e 3D, estes servem agora também para providenciar aceleração aos cálculos genéricos tradicionalmente processados pelo processador central (CPU) do computador. No entanto, a programação sobre GPUs requer o conhecimento da arquitetura e do modelo de programação destes. O tempo necessário para obter estes conhecimentos pode fazer com que muitas vezes o GPU não seja a plataforma escolhida, mesmo em problemas altamente paralelizáveis onde teria melhor desempenho do que correndo no CPU.

Normalmente, devido ao elevado número de unidades de processamento dos GPUs, em problemas onde é necessário executar a mesma operação várias vezes em dados diferentes têm um desempenho muito superior quando executados no GPU em relação à sua execução no CPU. Um exemplo de tais problemas são os algoritmos genéticos, problemas visados por este trabalho.

Nesta dissertação apresenta-se Gacuda, um motor de algoritmos genéticos que corre sobre GPUs. Este possibilita a utilização do GPU para executar algoritmos genéticos de uma maneira mais simples, escondendo a complexidade da programação do GPU para o utilizador do motor através da API fornecida mas ao mesmo tempo mantendo os benefícios em termos de desempenho. Em termos de código, apenas é necessário desenvolver um pequeno conjunto de funções na linguagem de programação C, sem detalhes sobre o GPU.

Palavras-chave: GPU, algoritmo genético, CUDA, motor
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 thesis 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
Contents

Resumo ................................................................. iii
Abstract .............................................................. v
List of Tables ......................................................... xi
List of Figures ......................................................... xiii
Glossary .............................................................. xv

1 Introduction ......................................................... 1
  1.1 Motivation ..................................................... 2
  1.2 Objectives ..................................................... 2
  1.3 Thesis Outline ............................................... 2

2 Genetic Algorithms ............................................... 5
  2.1 Overview ..................................................... 5
  2.2 Encoding ..................................................... 6
    2.2.1 Binary Encoding ........................................ 6
    2.2.2 Value Encoding ......................................... 7
    2.2.3 Permutation Encoding .................................. 7
    2.2.4 Tree Encoding ......................................... 7
  2.3 Fitness ....................................................... 8
  2.4 Selection ..................................................... 8
    2.4.1 Proportionate Reproduction ............................ 9
    2.4.2 Rank Selection ......................................... 9
    2.4.3 Tournament Selection .................................. 9
    2.4.4 Steady State .......................................... 10
    2.4.5 Elitism ................................................ 10
  2.5 Crossover .................................................... 10
    2.5.1 One-Point Crossover ................................... 10
    2.5.2 Two-Point Crossover .................................. 11
    2.5.3 Uniform Crossover ..................................... 12
    2.5.4 Crossover for Ordered Genomes ...................... 12
  2.6 Mutation ..................................................... 12
2.6.1 Flip Bit ................................................. 13
2.6.2 Uniform .............................................. 13
2.6.3 Swap ................................................... 13
2.7 Termination Criteria .................................... 14
2.8 Parallel Genetic Algorithms ............................. 14

3 General-Purpose Computing On Graphics Processing Units 17
   3.1 NVIDIA GPU Architecture ............................ 19
   3.2 CUDA .................................................. 20
      3.2.1 CUDA Runtime ................................... 20
      3.2.2 Workflow .......................................... 20
      3.2.3 Kernel .............................................. 21
      3.2.4 Kernel Scheduling ................................. 22
      3.2.5 Memory Model ..................................... 23
      3.2.6 Compute Capabilities ............................. 23

4 Implementation 25
   4.1 Architecture .......................................... 25
      4.1.1 Execution Flow .................................... 26
      4.1.2 GPU Mapping ...................................... 29
      4.1.3 Shared Memory ................................... 30
      4.1.4 Migration .......................................... 30
      4.1.5 Multi-GPU / Inter-Migration ....................... 30
      4.1.6 Pseudo Random Number Generation ............... 31
      4.1.7 Elitism ............................................ 32
      4.1.8 Statistics ......................................... 32
      4.1.9 Macros ............................................. 33
   4.2 Creating an Application ............................... 34
      4.2.1 Code ............................................... 34
      4.2.2 Configuration File ................................ 37
      4.2.3 Compilation ....................................... 39
      4.2.4 Execution ......................................... 39

5 Results 41
   5.1 Test Setup ............................................. 41
   5.2 Case Study 1: Numerical Optimization ............... 41
      5.2.1 Griewank ......................................... 42
      5.2.2 Rosenbrock ....................................... 43
   5.3 Case Study 2: 0-1 Knapsack ........................... 44
   5.4 Case Study 3: Travelling Salesman Problem .......... 45
5.5 Fitness Evolution ............................................. 47
5.6 Speedup ..................................................... 48
5.7 Multi-GPU Speedup ......................................... 50
5.8 GPU Occupancy ............................................. 50

6 Conclusions .................................................... 53
6.1 Future Work ................................................... 53

Bibliography ...................................................... 55

A Gacuda User Manual ......................................... 59
A.1 Getting Started ............................................... 59
A.2 Development .................................................. 60
  A.2.1 Headers ................................................... 60
  A.2.2 Macros .................................................... 60
  A.2.3 Functions .................................................. 61
  A.2.4 Configuration .............................................. 62
A.3 Building ....................................................... 64
A.4 Executing ....................................................... 64
A.5 Example Source-Code ....................................... 65
  A.5.1 Griewank .................................................... 65
List of Tables

2.1 Binary encoding examples. ................................................. 6
2.2 Value encoding examples. ................................................. 7
2.3 Permutation encoding examples. ....................................... 7
2.4 Tree encoding examples. .................................................. 8

5.1 Griewank single precision execution times. ......................... 43
5.2 Griewank double precision execution times. ....................... 43
5.3 Rosenbrock single precision execution times. ....................... 44
5.4 Knapsack execution results. .............................................. 45
5.5 The effects of the thread block size parameter on TSP results. .... 46
5.6 The effects of the number of iterations parameter on TSP results. . 46
# List of Figures

2.1 Genetic algorithm flowchart. ................................................................. 6
2.2 Visual representation of the encoded example trees. ......................... 8
2.3 Roulette-wheel and rank selection comparison. ................................. 9
2.4 One-point crossover. ................................................................. 11
2.5 Two-point crossover. ................................................................. 11
2.6 OX1 crossover. ................................................................. 12
2.7 Flip bit mutation. ................................................................. 13
2.8 Uniform mutation. ................................................................. 13
2.9 Swap mutation. ................................................................. 14

3.1 Comparison between CPU and GPU transistors dedicated to processing. 18
3.2 Floating-Point Operations per Second comparison between CPU and GPU. 18
3.3 High-level view of the NVIDIA GPU architecture. ............................ 19
3.4 CUDA Runtime workflow. ................................................................. 21
3.5 Thread organization in CUDA. ................................................................. 22

4.1 High level flowchart of Gacuda. ......................................................... 27
4.2 Gacuda GPU mapping. ................................................................. 29
4.3 Gacuda Migration. ................................................................. 31
4.4 Gacuda travelling salesman problem instance. .................................... 33

5.1 Griewank function plot. ................................................................. 42
5.2 Rosenbrock function plot. ................................................................. 44
5.3 Gacuda solution. ................................................................. 47
5.4 Optimal solution. ................................................................. 47
5.5 TSP fitness for a given number of generations. ................................. 47
5.6 TSP fitness for a given number of generations with the elitism feature. 48
5.7 Griewank execution time for a given number of iterations. ................. 49
5.8 Griewank execution time for a given population size. ..................... 49
5.9 Execution time comparison between single and dual GPUs setup. ...... 50
5.10 Gacuda execution timeline of CUDA API calls. ............................ 51
5.11 Gacuda occupancy analysis in Nsight Visual Studio. ..................... 51
# Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit.</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit.</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture.</td>
</tr>
<tr>
<td>CX</td>
<td>Cycle Crossover.</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access.</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random-Access Memory.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor.</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array.</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-Point Unit.</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm.</td>
</tr>
<tr>
<td>GDDR</td>
<td>Graphics Double Data Rate.</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit.</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-Purpose computing on Graphics Processing Units.</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment.</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial time.</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language.</td>
</tr>
<tr>
<td>OX</td>
<td>Order Crossover.</td>
</tr>
<tr>
<td>PMX</td>
<td>Partially Matched Crossover.</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data.</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction, Multiple Thread.</td>
</tr>
<tr>
<td>SM</td>
<td>Streaming Multiprocessor.</td>
</tr>
<tr>
<td>TSP</td>
<td>Traveling Salesman Problem.</td>
</tr>
</tbody>
</table>
Chapter 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 and 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.

1.1 Motivation

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 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.

1.2 Objectives

The main objective of this thesis 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 thesis 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.

1.3 Thesis Outline

This thesis describes how the solution Gacuda, a framework for developing genetic algorithms running over GPUs, was implemented in detail and how it can be used to solve a set of problems through
explaining the developed applications using the framework. This work also presents all the background information behind the solution developed.

This thesis is organized as follows: Chapter 2 introduces genetic algorithms; Chapter 3 explains the GPU architecture and the CUDA programming model used in the solution; Chapter 4 details the implementation and use of the software developed; Chapter 5 displays the results of the example problems implemented; finally, Chapter 6 ends the thesis with conclusions and possible future improvements to the solution.
Chapter 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.

A high level view of the execution flow of a genetic algorithm can be seen in Figure 2.1.
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 as is demonstrated next, it is up to the developer to choose the most fitting encoding to the problem at hand.

2.2.1 Binary Encoding

The first genetic algorithms used binary encoding [6]. In this simple type of encoding each individual genome is composed of bits. Each bit with the value one marks the presence of a characteristic in that individual while each bit with the value zero marks otherwise. The 0-1 Knapsack is an example of a problem that can be naturally represented with binary encoding [7]. In this case, a bit with the value one would indicate that a certain item would be taken in the knapsack while a bit with the value zero would indicate that the item would not be taken. A representation of genomes encoded in binary can be seen in Table 2.1.

<table>
<thead>
<tr>
<th>Genomes</th>
<th>Genes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome 1</td>
<td>1 0 1 1 1 0 0 1 0 1 0 1</td>
</tr>
<tr>
<td>Genome 2</td>
<td>0 0 1 0 0 1 0 1 0 1 0 0</td>
</tr>
</tbody>
</table>
2.2.2 Value Encoding

In value encoding [8], each genome contains one or more values. These values can be anything, real numbers, strings or other structured objects. Numerical optimization problems use this type of encoding where the genes are actually real numbers. A representation of two different types of genomes encoded with value encoding can be seen in Table 2.2.

Table 2.2: Value encoding examples.

<table>
<thead>
<tr>
<th>Genes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome 1</td>
</tr>
<tr>
<td>10.0</td>
</tr>
<tr>
<td>Genome 2</td>
</tr>
<tr>
<td>ABC</td>
</tr>
</tbody>
</table>

2.2.3 Permutation Encoding

In permutation encoding [8], the genome of each individual encodes an order. The genome is an ordered sequence of numbers or other objects that the fitness function will use to evaluate each individual. This type of encoding is used in ordering problems. One example of such a problem is the Travelling Salesman Problem (TSP), consisting 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 common, when using this type of encoding, to use special cross-over and mutation operators so that the consistency of the solution is kept. For example, if a mutation operator would just change a gene to a duplicate value of the sequence, the solution could become invalid depending on the problem. A representation of two different types of genomes encoded with permutation encoding can be seen in Table 2.3.

Table 2.3: Permutation encoding examples.

<table>
<thead>
<tr>
<th>Genes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome 1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Genome 2</td>
</tr>
<tr>
<td>(1, 3)</td>
</tr>
</tbody>
</table>

2.2.4 Tree Encoding

It is also possible to encode the genes of an individual in a tree like structure [8] as seen in Table 2.4. In the first example, the genome 1 encodes the expression \((2 \times b) + (3 - c)\) in a tree. This type of encoding is particularly useful to encode expressions and programs. This representation could be used, for example, to make genetic algorithms evolve programs in the Lisp programming language. Lisp is used as an example because its functions are defined through tree structures, mapping naturally to this encoding.

The Figure 2.2 represents visually the genomes presented in the Table 2.4.
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 [9].

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). The most commonly used selection operators are described next.
2.4.1 Proportionate Reproduction

Proportionate reproduction is the name given to a class of selection methods that chooses individuals directly based on their fitness. The roulette-wheel selection [10] (also known as the fitness proportionate selection) is one of the most relevant.

For the roulette-wheel selection method the first step is to attribute to each individual a range to be selected depending on its fitness. The fitter the individual is, the bigger its range is and bigger its probability of being selected. Finally, to select an individual, a random number between the total range is generated. This operation is repeated several times until all the individuals that will make it to the next generation are selected.

Other methods include the stochastic remainder [11] and the stochastic universal selection [12].

2.4.2 Rank Selection

The roulette-wheel selection can be too harsh for the less fit individuals, giving them a very low probability to propagate their genes and therefore making the genetic algorithm population converge too rapidly. An alternative solution that is more benevolent to the worst genomes is the rank selection [13]. In this method of selection, each individual is ordered by its fitness, from worst to best. Then, for all individuals, a new fitness value is attributed with the worst individual having the value 1, the second worst the value 2 and so on, ending with the best solution having a fitness value equal to the size of the population. This will make the probability of selecting an individual directly proportional to its order among all individuals. With the new fitness values calculated the regular round-wheel selection is executed. In the Figure 2.3 it is displayed the impact on the probability of selection of an individual caused by the use of different selection algorithms for the same exact population of four individuals.

2.4.3 Tournament Selection

Tournament selection [14] 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. The more individuals participate in a tournament, the less are the odds of low fitness participants winning the tournament and being selected.

2.4.4 Steady State

The class of steady state contains selection methods that do not act on the whole population, but only pick some of the best individuals to cross and some of the worst to be replaced. Most of the population goes unchanged to the next generation. The Genitor [12, 15] selection method is an example of a steady-state selection algorithm.

2.4.5 Elitism

Elitism [16] 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. The probability of occurrence of crossover between a pair of individuals is usually set up a high value, between 60% and 90%. 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. A probability of 0% means the population is unaffected by crossover and passes on unchanged by crossover to the next iteration of the algorithm. Setting the probability below 100% allows for some of the individuals to pass on unaffected to the next generation.

Many crossover operators exist. They impact the performance of the algorithm greatly depending on the problem and encoding used. Some of the most commonly used crossover operators will be presented next.

2.5.1 One-Point Crossover

For this type of crossover, one randomly selected point in the genome of both parents is selected (this crossover point will be randomly selected for every crossover operation). Then the first part of the offspring’s genome receives the genetic code of the first parent while the second part of the offspring’
genome takes the matching genetic code of the second parent. Figure 2.4 shows an example of two individuals using one-point crossover to generate offspring, where the randomly selected crossover point is between the third and fourth gene. The generated individual will have the first three genes taken from the first parent and the rest of the genes from the second.

![Figure 2.4: One-point crossover.](image)

### 2.5.2 Two-Point Crossover

Two-point crossover works similarly to one-point crossover, but two random points are selected instead of just one. The genetic code of the offspring is then constructed with the genes of the parents alternated between the selected points. Figure 2.5 shows an example of two-point crossover with the first crossover point between the fourth and fifth gene. The second crossover point was randomly selected to be between the eighth and ninth genes. The first and last part of the generated individual take the genes from the first parent while the part in the middle takes the genes from the second parent.

![Figure 2.5: Two-point crossover.](image)
2.5.3 Uniform Crossover

This crossover method is applied gene by gene until the whole offspring genome is generated. Each gene has an equal probability to come from either of his parents.

2.5.4 Crossover for Ordered Genomes

Ordered genomes are those that encode an order (using permutation encoding as seen in 2.2.3). For ordered genomes special care has to be taken so that the solution does not become invalid. There are multiple ordered crossover operators such as the partially matched crossover (PMX), cycle crossover (CX), order crossover operator (OX1), among others [17]. OX1 crossover will be presented with more detail.

Just like in one point cross over, the first part of the offspring receives the corresponding genes from the first parent. But as for the correspondent genes transferred from the second progenitor, in this method, the genes cannot be blindly used. This is due to the fact it could result in the repetition of genes already set from the first parent and therefore lead to an invalidation of the solution. So, in this case, the genes that were not previously used are transferred from the second parent by order.

![OX1 crossover](image)

**Figure 2.6: OX1 crossover.**

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.

The probability of mutation of an individual is usually set to a very small value between 0% and 10%. A very large value would frustrate the crossover and basically turn the genetic algorithm to a blind search.
The mutation operator that can be used depends on the encoding selected. A short list of available mutators are discussed next.

### 2.6.1 Flip Bit

Suitable for binary encoding problems, this mutation method chooses a random position in the genome and inverts the bit of the selected gene. In Figure 2.7 an example of a flip bit mutation can be seen. The fourth bit was randomly selected to be mutated and it changed from the value 0 to the value 1.

![Figure 2.7: Flip bit mutation.](image)

### 2.6.2 Uniform

This operator selects one gene randomly and then randomly selects a new value for the gene based on the allowed range of values. The probability of selecting any value is uniform among the allowed bounds. In Figure 2.8 an example of uniform mutation can be seen. Being randomly selected, the fourth gene is mutated to a value within the acceptable range of values for that particular problem. In that particular case it changed from the value four to the value five.

![Figure 2.8: Uniform mutation.](image)

### 2.6.3 Swap

In this mutation method two randomly selected genes have their values swapped. Swap mutation is particularly useful for problems using ordered encodings such as the Permutation Encoding. With ordered encodings using other mutators could make an individual no longer represent an order between
all elements. Since other mutators blindly change genes values, the original problem genes would be lost and the encoded solution would become invalid.

In Figure 2.9 both the second and fourth genes on the original individual are randomly selected. After this, they are swapped and the genome of the mutated obtained.

<table>
<thead>
<tr>
<th>Original</th>
<th>Mutated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 0</td>
<td>1 4 3 2 5 6 7 8 9 0</td>
</tr>
</tbody>
</table>

Figure 2.9: Swap mutation.

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. The most traditionally employed termination criteria are:

- Pre-determined number of iterations complete. After iterating the set number of times the algorithm ends.

- Reaching a pre-determined target fitness. Here, a target fitness value is set to be reached. When any individual fitness value becomes larger than the fitness set the algorithm stops. This condition requires previous knowledge of the fitness values of the problem at hand.

- No improvement after a pre-determined number of iterations. In this case, after every iteration a test is made to check if the fitness of the most fit individual has improved. If no fitter individual is found after a pre-defined number of generations then the algorithm ends. Otherwise, if a more fit individual is found then the termination condition counter resets.

- User-Input. In this condition the algorithm is constantly running, only stopping when the user decides to do so and makes it stop directly through some input.

2.8 Parallel Genetic Algorithms

Making genetic algorithms run in parallel can be achieved through various methods [18]:

- Master-slave: In this method the population is not split, it is processed fully by one processor (the master) with the exception of the fitness function, whose evaluation is delegated to other processors (the slaves).
• Fine-grained: Fit for massively parallel computers, in this case the population is arranged into a structure, divided in a large amount of areas with few individuals. Each part of the population is processed individually but selection and crossing can use individuals from the close neighbourhood.

• Coarse-grained: In this method the population is divided evenly among the available processors (few areas with lots of individuals) and each processor computes its respective population in isolation. The only exception is that every number of iterations a migration occurs and processors trade individuals with each other in a ring topology. This method of dividing the population and processing it in isolation is also referred to as the Island Model [19].
Chapter 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 parallelization are met [20]. 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 [21]. Figure 3.1 illustrates this situation well.

It is expected that GPUs will continue to outperform CPUs in these conditions for the close future, according to the current trends, as seen in Figure 3.2 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 [22] and CUDA [5].

OpenCL is a framework that allows developing applications for various types of platforms such as CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and others. It supports development of applications for the Playstation 3 accelerators as an example [23]. 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.
Figure 3.1: Comparison between CPU and GPU transistors dedicated to processing. (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

Figure 3.2: Floating-Point Operations per Second comparison between the CPU and GPU. (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)
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.
Looking back to 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. It is widely used, specially to develop GPU applications on NVIDIA devices. It supports Microsoft Windows, Linux, Apple’s MacOSX and various Integrated Development Environments (IDEs) through dedicated integration plug-ins. It also has a considerable amount of documentation and libraries available online. One problem with CUDA is being vendor locked to NVIDIA and not available for other 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.

3.2.1 CUDA Runtime

CUDA Runtime is the high level API provided by NVIDIA that allows the user application to set up and schedule the execution of code in the GPU. It supports C, C++ and Fortran programming languages, with other languages being available through third party bindings. This API provides the functions, for example, to allocate GPU memory and to schedule the execution of code in the GPU. The code that runs in the GPU is called a kernel. CUDA kernels are written in a language similar to the C language but with CUDA extensions.

3.2.2 Workflow

From a developer point of view, a typical CUDA Runtime application consists in the following operations:

1. Copying the data to process from the (CPU) main memory to the GPU memory.

2. Instructing the GPU how to schedule processing, indicating the number of blocks and how many threads which block will process. This step will start the execution on the GPU. (The GPU code is copied automatically to the GPU on this step).

3. Waiting for the completion of the GPU task. The GPU will be processing the data in parallel.

4. Copying the processed data from the GPU memory to the main memory.
While the GPU is executing code, the CPU is free to execute other tasks normally or can be set to wait for the GPU to end its work. This workflow is represented in Figure 3.4.

![CUDA Runtime workflow](image)

**Figure 3.4: CUDA Runtime workflow.**

### 3.2.3 Kernel

A kernel in CUDA is the function that is launched to run in the GPU, and can be developed in the C programming language with CUDA extensions. A simple kernel that performs the square operation to every float in one array can be seen in Listing A.16.

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

Listing 3.1: Simple CUDA kernel.

In this kernel every thread will be responsible for modifying only one float of the array. The kernel will use the CUDA provided variables blockIdx, blockDim and threadIdx to distinguish between threads.

There are special keywords in CUDA that can be applied to function declarations to modify their meaning. The effect of these qualifiers is the following:

- `__global__`: Means the function being declared is a GPU kernel that is launched by the CPU. This function must return void.
• _device_: Means that the function is GPU code. This type of functions can only be called by GPU code.

• _host_: This qualifier means that the function being declared is to be executed by the CPU.

A function can have both _host_ and _device_ declared, so that is capable of running in both the CPU and GPU.

### 3.2.4 Kernel Scheduling

In CUDA, when ordering a kernel to execute, the user must provide the information of how many threads there will be and how they are organized. This is done adding meta information between brackets in a otherwise regular C function call as seen in Listing 3.2.

```c
dim3 number_blocks(3,2,1);
dim3 threads_per_block(4,3,1);
test_kernel <<< number_blocks , threads_per_block >>>(array);
```

Listing 3.2: CUDA kernel call example.

In the example shown, the kernel `test_kernel` is being executed after variables with information of how to organize the threads were defined. The first argument inside the brackets stands for the number of blocks of threads that will be created to execute the GPU code. Each of these blocks will then have the quantity of threads specified on the second argument inside the brackets. In the case listed there will be 6 blocks of 12 threads each for a total of 72 threads.

![Thread organization in CUDA.](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html)

CUDA organizes threads in a multidimensional grid. A CUDA grid is composed by a variable number of groups of threads, called blocks, organized in up to three dimensions. Each block will be fully
processed only by one SM. Going deeper, each block is itself composed of threads also organized mul-
tidimensionally. Although CUDA grids support three dimensions, developers can choose to use less
dimensions. Figure 3.5 shows the 2-dimensional grid resulting from the call issued in the Listing 3.2.

3.2.5 Memory Model

Memory in CUDA can be divided in the following classes:

- Global memory: High latency memory but available in high amounts. Corresponds to the GDDR5 memory of the GPU. Accesses to this memory should be minimized for maximum performance.

- Constant Memory: Low latency memory when all thread access the same memory location. This memory is also, like global memory, placed in the GDDR5 memory portion of the GPU but it is always cached for fast accesses in the L2 cache. The GPU kernel only has read-only access to this memory.

- Texture Memory: Similar to constant memory, it is read-only too. It has high throughput when all threads in a warp access physically adjacent memory positions.

- Shared Memory: Low latency memory. It is shared between all threads within a SM. Ideally all the global memory accessed multiple times would be accessed only once and copied to shared memory to reduce access times. Unfortunately the shared memory more limited size does not allow that for every application.

- Register: The memory with the lowest latency. Usable only within the execution of a single thread. Only a small amount of registers are available per thread. Exhaustion of registers causes register spilling, meaning the compiler has to resort to other memories.

3.2.6 Compute Capabilities

Depending on the model, a compute capability number is attributed to every CUDA device. Each device with a certain compute capability number will fulfill a list of requirements either in supported features (atomic functions, GPU to GPU DMA) or in hardware specifications (such as the amount of shared memory per SM). When compiling for CUDA it is possible to choose a compute capability. This can be useful since it is possible for any GPU to execute a CUDA program compiled for an inferior compute capability number.
Chapter 4

Implementation

In this chapter, 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 through an example application of a simple string problem.

4.1 Architecture

Gacuda is the framework created for this thesis 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. In addition, this function also serves to define the INPUT array that the framework provides. This array is used to hold support data for the user application. It allows, for example, the user to read data from files and then use that data in the fitness function. This way there is no need for this data to be hard coded in the fitness function. This feature will be useful for some problems such as the 0-1 knapsack (see Section 5.3) for loading the weights and values that help define the problem.

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 [24]. 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. The speedup of 7000X in relation to the CPU execution presented in the paper shows exactly that [25] (although part of this extraordinarily high speedup can be attributed to the slow library used in the implementation of the CPU version used in the comparison). 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, the framework code will run and will eventually use the user-defined functions when appropriate.

### 4.1.1 Execution Flow

A detailed description of Gacuda execution flow will now be provided. A visual representation of it can be seen in Figure 4.1.

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
Figure 4.1: High level flowchart of Gacuda.
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 ($2.88 \times 10^{17}$ threads for compute capability 2.0).

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 undergo 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.
4.1.2 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 \textit{population.size} and \textit{threads.per.block} parameters are specified in the configuration file. The formula for calculating the number of blocks parameter to send to the GPU is the following:

\[
\text{number_blocks} = \frac{\text{population.size}/\text{threads.per.block}}{\text{number.GPU.devices}}
\]

In Figure 4.2 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.

![Gacuda GPU mapping diagram](image-url)
4.1.3 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 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, then CUDA API `cudaFuncSetCacheConfig` function is called to set CUDA to use more of the SM memory as shared memory in detriment of the L1 cache size. If the population does not fit the shared memory then the same function is called but, this time, to order CUDA to allocate more L1 cache size. This way, every access to a gene requires accessing global memory but the odds of this access being cached are better.

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.

The need to store two whole populations exists because the kernel code uses double buffering to save the population, keeping both the old and new populations. This allows minimising the use of synchronization inside the kernel, since with only one population there would be problems with a newly generated individual being used as a parent for another selected individual in the same algorithm step.

4.1.4 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 [26]. 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 4.3.

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.

4.1.5 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 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.

4.1.6 Pseudo Random Number Generation

CUDA kernel functions cannot 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 [27] 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.

The implemented algorithm is seeded with a random number from the Standard Library `rand()` function and then every time a number is generated by a thread, the seed in global memory is updated. Random race conditions occur when reading and writing the volatile seed variable but they are not problematic since they only affect the generated number that is already being randomly generated.

The random number generation function is displayed in Listing 4.1.

A random number generator macro function is provided for the user to apply when programming the crossover and mutation functions. This macro calls the `rand()` function if executed on the CPU or the `rng()` function if called on the GPU. The macro is showed in Listing 4.2.
4.1.7 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. Other works [25], for example, execute the bitonic merge sort algorithm [28] in the GPU to order the fitness values allowing for a variable number of the best individuals to pass on to the next generation.

The elitism feature is very useful, as it will be shown later, making the algorithm converge to better solutions in a shorter amount of time. It prevents losing the best solution in each iteration and makes it so that the genetic algorithm can not move backwards, that is, the maximum fitness value is always increasing and never recedes.

4.1.8 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\ atom\text{icMax},\ atom\text{icMin} and\ atom\text{icAdd}, respectively.

In Figure 4.4 it is possible to see one way of treating the data output of the statistics kernel. A simple script was developed in the Python programming language using an open-source graph library called NetworkX. The script polls the output file with the best current population generated by Gacuda every couple of iterations. After detecting changes, the script plots an image with a visual representation of the problem, allowing for the visualization of the progress of the algorithm in real time.

![Figure 4.4: Gacuda travelling salesman problem instance.](image)

4.1.9 Macros

The configurations set through the configuration file 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. A list of available macros is displayed next.

- \text{POPULATION\_SIZE}: The total number of individuals in the population.
- \text{GENE\_SIZE}: The number of bytes in a gene.
- \text{NUMBER\_GENES}: The number of genes of an individual.
- \text{GENOME\_SIZE}: The total size in bytes of an individual. Equals \text{GENE\_SIZE} times \text{NUMBER\_GENES}.
- \text{INPUT\_SIZE}: The size of the auxiliary read-only data block used to store problem specific input data.
Additionally, a pseudo random number generation function macro working for both CPU and GPU code is available through `RAND()`. This allows the user to apply always the same macro when generating random numbers with no distinction between GPU or CPU code.

### 4.2 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 section where a simple sample application will be created. The objective this application has is reaching the binary string ’10101’. For that purpose, value encoding will be used. Each individual will consist of five characters, with the whole genome of an individual representing a five letter string. In short, the genetic algorithm will start with a population of strings with five random characters each and will try to obtain a string matching ’10101’.

#### 4.2.1 Code

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.

**Initialization**

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. These functions run at the CPU level so that files can be open, for example. Since initialization takes such a short time in comparison to the whole execution time of the algorithm, running it in the GPU would not provide any worthwhile reduction of the total execution time.

In Listing 4.3 a cast to the correct data type of the gene is made. This is a typical operation done at the start of every Gacuda function. Afterwards, the whole individual genome is iterated gene by gene, with each gene being initialized with a random value between a fixed range.

In this example application, the function `initialize_population` was not used and that is why the function only instruction is a return, as seen in Listing 4.4.

Nevertheless, this function is useful in problems where input data must be read from files, since this way the file is only read once. This, in opposition to executing the `initialize_gene` multiple times, opening the input file to initialize a gene at every call. An example of a problem of this kind includes the 0-1 Knapsack problem where the maximum weight limit and existing items weights and values must be known *a priori*, before the problem starts being solved.
**__host__** void initialize_gene ( void * genome )
{
    char *gene = ( char *) genome ;
    for ( int i = 0; i < NUMBER_GENES ; i++)
        gene[i] = 'A' + ( RAND() % ('z' - 'A' + 1));
    return ;
}

Listing 4.3: Initialize gene function *initialize_gene*.

**__host__** void initialize_population ( void * population )
{
    return ;
}

Listing 4.4: Initialize population function *initialize_population*.

**Fitness**

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 fitness. In the example in study, as seen in Listing 4.5, every gene of the measured individual is being evaluated to how close it is to the desired solution value. The desired value for a gene is either the character '1' (if it is located in an even position) or the character '0' (if in an odd position).

```c
__host__ __device__ float fitness ( void * genome )
{
    char *gene = ( char *) genome ;
    float score = 0;

    for ( int i = 0; i < NUMBER_GENES ; i++)
    {
        if (i % 2 == 0)
            score += 255 - abs(gene[i] - '1');
        else
            score += 255 - abs(gene[i] - '0');
    }
    return score;
}
```

Listing 4.5: Fitness function.

The keyword **__host__** is used in the signature of the fitness function so that the function can be executed on the CPU too. The fitness function is called on the CPU side by Gacuda when the algorithm ends and the user opted (through the configuration file) to output the final fitness values and identify the
best individual.

**Crossover**

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. For the sample application in study, a single point crossover was implemented as it can be seen in Listing 4.6. The crossover point is fixed in the middle of the genome size. The first part of the offspring receives genes from the first parent and the second part receives genes from the other progenitor.

```
__device__ void crossover (void *dest, void *genome1, void *genome2)
{
    char *gene1 = (char *) genome1;
    char *gene2 = (char *) genome2;
    char *gene_dest = (char *) dest;
    for (int i = 0; i < NUMBER_GENES / 2; i++)
        gene_dest[i] = gene1[i];

    for (int j = NUMBER_GENES / 2; j < NUMBER_GENES; j++)
        gene_dest[j] = gene2[j];
    return;
}
```

Listing 4.6: Crossover function.

**Mutation**

The mutation function receives a pointer to an individual's genes. Afterwards, the genes are to be mutated according to chance. In the example presented in Listing 4.7, a random gene is selected and its value is either incremented or decremented randomly. It is worth noting that the rate at which this function is called for each individual is set in the configuration file.

```
__device__ void mutation (void *genome)
{
    char *gene = (char *) genome;
    if (RAND() % 2 == 0)
        gene[RAND() % NUMBER_GENES]++;
    else
        gene[RAND() % NUMBER_GENES]--;
    return;
}
```

Listing 4.7: Mutation function.
Output

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.

This function receives a pointer to an individual’s genetic code and a pointer to the output string. The example presented in Listing 4.8 iterates the individual’s genes one by one and copies their value directly to the output string since the genes in this example actually correspond to a character.

```
__host__ void output (void *genome, char *result)
{
    char *gene = (char *) genome;

    for (int i = 0; i < NUMBER_GENES; i++)
        result[i] = gene[i];

    result[NUMBER_GENES] = '\0';

    return;
}
```

Listing 4.8: Output function.

4.2.2 Configuration File

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 most important configuration entries are:

- **number_runs**: Specifies the number of iterations to be executed before halting the algorithm.

- **population_size**: Specifies how many individuals exist in the population. This number should be large enough to guarantee that all GPU resources are utilized. It corresponds to the number of GPU threads that will be executed per iteration.

- **number_genes**: Specifies the number of genes in each individual.

- **gene_size**: Specifies the size in bytes of each gene.

- **threads_per_block**: Specifies the population size of each island. The best value to use here must be tuned to the specific problem since it depends on the resources used that are limited by hardware constraints. It is recommended the use of the CUDA GPU Occupancy Calculator excel spreadsheet to obtain the best value to apply.
The configuration file used for this application is presented in Listing 4.9.

```plaintext
number_runs = 500
population_size = 24
number_genes = 5
gene_size = 1
threads_per_block = 8

crossover_rate = 0.8
mutation_rate = 0.2
elitism = true

migration = false
migration_rate = 10
migration_size = 2

intra_migration = true
intra_migration_rate = 5
intra_migration_size = 2

# can take multiple devices "0;1;2;3"
device_list = 0

output_input = true

output_scores = true
output_scores_file = output/scores.dat

output_individuals = true
output_individuals_file = output/genes.dat

output_best = true

input_size = 0

trace_best = false
statistics = true
```

Listing 4.9: Configuration file.
4.2.3 Compilation

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. A sample Makefile for the knapsack application is presented in the Listing 4.10.

```makefile
APP = knapsack
NVCC = /usr/local/cuda/bin/nvcc
OPTIONS = -gencode=arch=compute_20,code="sm_20,compute_20" \ 
  --ptxas-options=-v -use_fast_math -lm -maxrregcount 32

# END OF CONFIGURABLE ZONE

all: $(APP)

$(APP): $(APP:.cu) gacuda.h loader.h gacuda_code.h
  $(NVCC) -g -G -o $(APP) $(APP:.cu) $(OPTIONS)

clean:
  rm $(APP)

Listing 4.10: Gacuda Makefile.
```

4.2.4 Execution

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 as it can be seen in Listing 4.11.

```
alucart@unix ~/gacuda.git/src $ ./binary conf-binary.conf
...
GACUDA started.
...
GA Execution ended.
Outputing scores to output/scores.dat.
Outputing individuals to output/genes.dat
Best score: 1275.0000000000000000000
Best individual: 10101
All done.
```

Listing 4.11: Executing the application.
Chapter 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 Test Setup

The tests were carried out in a computer equipped with an Intel(R) Xeon(R) CPU E5-2620 v2, clocked at 2.10 GHz, with 32 GB of memory. The GPU installed on that computer was the NVIDIA Tesla K40c with 15 SMs, 2880 CUDA cores and 12 GB of memory. The version 7.0 of CUDA was used. All measurements were repeated seven times, with the best and worst measurements being discarded.

Regarding the configuration file, the following settings were used, unless stated otherwise:

- Threads per block: 64
- Crossover rate: 80%
- Mutation rate: 10%
- Elitism feature: Active

5.2 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 [29] and Griewank [30].
5.2.1 Griewank

This function has a multitude of local minima present, as seen in Figure 5.1, that might trick some algorithms in limiting their search space quickly.

Griewank mathematical definition is as follows:

\[
f(x_1, \ldots, x_n) = 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{5.1}\]

This function has a global minimum at:

\[
f(0.0, \ldots, 0.0) = 0.0 \tag{5.2}\]

The interval used for the test was:

\[-600 \leq x_i \leq 600 \tag{5.3}\]

The crossover operator implemented is the one point crossover and the mutation operator is a simple value mutation.

The execution time and quality of the obtained solutions in the performed test are presented in Table 5.1. It is possible to see that using a bigger population (with a higher number of individuals), although it increases the execution time, results in better solutions (The optimal solution is the value 0).

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 ver-
Table 5.1: Griewank single precision execution times.

<table>
<thead>
<tr>
<th>Population Size</th>
<th>Average Best Solution Found</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>$1.47 \times 10^{-3}$</td>
<td>1.561</td>
</tr>
<tr>
<td>2048</td>
<td>$9.51 \times 10^{-4}$</td>
<td>1.614</td>
</tr>
<tr>
<td>4096</td>
<td>$3.18 \times 10^{-4}$</td>
<td>2.251</td>
</tr>
<tr>
<td>8192</td>
<td>$1.89 \times 10^{-4}$</td>
<td>5.238</td>
</tr>
<tr>
<td>16384</td>
<td>$9.86 \times 10^{-5}$</td>
<td>9.007</td>
</tr>
<tr>
<td>32768</td>
<td>$2.65 \times 10^{-5}$</td>
<td>16.493</td>
</tr>
</tbody>
</table>

As it can be seen in Table 5.2, 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.

Table 5.2: Griewank double precision execution times.

<table>
<thead>
<tr>
<th>Population Size</th>
<th>Average Best Solution Found</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>$1.45 \times 10^{-3}$</td>
<td>4.202</td>
</tr>
<tr>
<td>2048</td>
<td>$8.93 \times 10^{-4}$</td>
<td>4.301</td>
</tr>
<tr>
<td>4096</td>
<td>$4.66 \times 10^{-4}$</td>
<td>5.872</td>
</tr>
<tr>
<td>8192</td>
<td>$2.50 \times 10^{-4}$</td>
<td>14.346</td>
</tr>
<tr>
<td>16384</td>
<td>$1.59 \times 10^{-4}$</td>
<td>24.685</td>
</tr>
<tr>
<td>32768</td>
<td>$9.60 \times 10^{-5}$</td>
<td>45.254</td>
</tr>
</tbody>
</table>

### 5.2.2 Rosenbrock

The Rosenbrock function (also known as Rosenbrock’s valley) is characterized by having a global minimum in a narrow long parabolic shape resembling a valley. The plot of the Rosenbrock function can be seen in Figure 5.2.

Rosenbrock mathematical definition is as follows:

$$f(x_1, ..., x_n) = \sum_{i=1}^{n-1} (100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2) \quad (5.4)$$

This function has a global minimum at:

$$f(1.0, ..., 1.0) = 0.0 \quad (5.5)$$

The interval used for the test was:

$$-5 \leq x_i \leq 10 \quad (5.6)$$
The crossover operator selected for the implemented Rosenbrock application was the one point crossover and the mutation operator was a simple value mutation. The encoding is the value encoding with each gene corresponding to a float.

For this test, the selected variable parameter was the number of input variables for the Rosenbrock function. To modify this parameter it is enough changing the value of the entry corresponding to the number of genes in the configuration file. The population size was fixed at 4096 individuals. As it can be seen in the Table 5.3, 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.

Table 5.3: Rosenbrock single precision execution times.

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>Average Best Solution Found</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$8.51 \times 10^{-5}$</td>
<td>2.024</td>
</tr>
<tr>
<td>4</td>
<td>$7.21 \times 10^{-4}$</td>
<td>3.181</td>
</tr>
<tr>
<td>8</td>
<td>$2.81 \times 10^{-3}$</td>
<td>5.495</td>
</tr>
<tr>
<td>16</td>
<td>$1.70 \times 10^{-1}$</td>
<td>10.100</td>
</tr>
<tr>
<td>32</td>
<td>$3.64 \times 10^{0}$</td>
<td>19.380</td>
</tr>
<tr>
<td>64</td>
<td>$4.99 \times 10^{1}$</td>
<td>38.006</td>
</tr>
</tbody>
</table>

5.3 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.

Maximize:

$$\sum_{i=1}^{n} v_i x_i$$  \hspace{1cm} (5.7)

Restricted to:

$$\sum_{i=1}^{n} w_i x_i \leq W$$  \hspace{1cm} (5.8)

where

$$x_i \in \{0, 1\}$$  \hspace{1cm} (5.9)

The value and weight of an object are represented by $v_i$ and $w_i$ respectively. $W$ is the maximum amount of weight that can be taken. If the object is taken in the knapsack, then $x_i$ is equal to 1. Otherwise $x_i$ will be equal to 0.

Again the crossover operator used was the single point crossover. For the mutation a simple value mutation operator was used. The knapsack application was executed five times for the same knapsack problem. The results are listed in Table 5.4. It is possible to see that the application is able to obtain the optimal solution consistently in the very short execution time. There were 21 items in the test performed and the optimal weight was 396 for a combined value of 1030.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Target Weight</th>
<th>Target Value</th>
<th>Calculated Weight</th>
<th>Calculated Value</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>396</td>
<td>1030</td>
<td>396</td>
<td>1030</td>
<td>0.363</td>
</tr>
<tr>
<td>2</td>
<td>396</td>
<td>1030</td>
<td>396</td>
<td>1030</td>
<td>0.405</td>
</tr>
<tr>
<td>3</td>
<td>396</td>
<td>1030</td>
<td>396</td>
<td>1030</td>
<td>0.399</td>
</tr>
<tr>
<td>4</td>
<td>396</td>
<td>1030</td>
<td>395</td>
<td>1002</td>
<td>0.367</td>
</tr>
<tr>
<td>5</td>
<td>396</td>
<td>1030</td>
<td>396</td>
<td>1030</td>
<td>0.364</td>
</tr>
</tbody>
</table>

5.4 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.

The application developed uses a OX1 ordered crossover operator. A modified swap mutation was also a chosen operator. For it, before the swap of values is actually made, the algorithm checks if the
change would result in an improvement. If so, then the swap is made and if not, then no value is altered. This mutation operator is called a 2-opt mutation operator [31]. In this case, the crossover and mutation rates used were both 100%. Such a high mutation rate is not problematic for this problem because the mutation only acts if there is some benefit to the fitness value with the change.

To test the algorithm, a set of problem instances were used. These belong to TSPLIB [32], 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. The results are displayed in the Table 5.5. It can 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 performance 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.

Table 5.5: The effects of the thread block size parameter on TSP results.

<table>
<thead>
<tr>
<th>TSPLIB Problem</th>
<th>Population Size</th>
<th>Threads Per Block</th>
<th>Iterations</th>
<th>Optimal Solution Difference</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>att48</td>
<td>2048</td>
<td>4</td>
<td>2000</td>
<td>0.323</td>
<td>84.108</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>8</td>
<td>2000</td>
<td>0.204</td>
<td>50.097</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>16</td>
<td>2000</td>
<td>0.160</td>
<td>31.534</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>32</td>
<td>2000</td>
<td>0.116</td>
<td>32.586</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>64</td>
<td>2000</td>
<td>0.129</td>
<td>34.872</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>128</td>
<td>2000</td>
<td>0.142</td>
<td>36.138</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>256</td>
<td>2000</td>
<td>0.176</td>
<td>36.825</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>512</td>
<td>2000</td>
<td>0.218</td>
<td>37.787</td>
</tr>
</tbody>
</table>

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

Table 5.6: The effects of the number of iterations parameter on TSP results.

<table>
<thead>
<tr>
<th>TSPLIB Problem</th>
<th>Population Size</th>
<th>Threads Per Block</th>
<th>Iterations</th>
<th>Optimal Solution Difference</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>att48</td>
<td>2048</td>
<td>64</td>
<td>1000</td>
<td>0.156</td>
<td>17.561</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>64</td>
<td>2000</td>
<td>0.128</td>
<td>34.909</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>64</td>
<td>3000</td>
<td>0.101</td>
<td>52.753</td>
</tr>
<tr>
<td>att48</td>
<td>2048</td>
<td>64</td>
<td>4000</td>
<td>0.064</td>
<td>69.715</td>
</tr>
</tbody>
</table>

In Figures 5.3 and 5.4 it is possible to compare a result obtained in 10000 generations from the developed TSP application versus the optimal solution for the same att48 TSPLIB problem. The difference of the total path distance sum between both is below 5%.
5.5 Fitness Evolution

In order to see how the fitness evolves over the number of generations the statistics feature in Gacuda was activated and the fitness values were recorded while executing the TSP application. Figure 5.5 presents the progress of the best fitness value along the number of generations.

![Figure 5.5: TSP fitness for a given number of generations.](image)

It can be seen that the fitness improves rapidly until it reaches a point where its progress slows down. This is the expected result from a genetic algorithm.

Another test was performed but this time with the elitism feature activated. This feature makes it so that the best individual of every island is kept unchanged on to the next generation. This results in less
fluctuations and in a better fitness value at the end of the algorithm, comparing with the same execution without the elitism feature activated. This can be seen in Figure 5.6.

![Figure 5.6: TSP fitness for a given number of generations with the elitism feature.](image)

## 5.6 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 5.7 and Figure 5.8 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 5.7: Griewank execution time for a given number of iterations.

Figure 5.8: Griewank execution time for a given population size.
5.7 Multi-GPU Speedup

To measure the framework performance gain of using multiple GPUs, additional tests were made. These tests were made on a machine with an Intel(R) Core(TM) i7 950 CPU, 12 GB and two NVIDIA Tesla C2050 GPUs.

Has it can be seen on Figure 5.9, when using two GPUs the execution time was reduced to almost half the time than when executing with only one GPU.

![Figure 5.9: Execution time comparison between single and dual GPUs setup.](image)

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.

5.8 GPU Occupancy

In order to assess how GPU's resources were utilized, the implemented Griewank application was chosen to be profiled. For these measurements the software used was the NVIDIA Visual Profiler and NVIDIA Nsight Visual Studio.

In Figure 5.10 it is possible to see a high level timeline of the execution of the application regarding CUDA API calls. The figure shows that the kernels are executed sequentially without great delay from either the statistics kernel or the migration memory transfers. It also reveals that the statistics kernel impact is so small that it does not even reach 0.1% of the total GPU execution time.

Regarding the performance of the GPU code itself, Figure 5.11 presents the data obtained from Nsight Visual Studio. It displays a list of executed kernels and the related performance information with
the most important columns being the last three. It is possible to see that the kernel code has good GPU occupancy with values varying between 93% and 95%. This means that in that percentage of time the GPU is always producing results. Regarding branch efficiency, good values were also obtained. This can be seen in the Branch Efficiency column. The value 98% means that most of the time the GPU threads in a warp are executing the same instructions. Only 2% of the time the flow diverges. Finally, we have the control flow execution efficiency value situated at 20%. This is the worst obtained value, which corresponds to how many threads are available for each instruction in a warp when the branching occurs. Great part of reason for such a small value can probably be attributed to the elitism feature which only uses one single thread per block.
Chapter 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 thesis 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 knapsack.

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.

6.1 Future Work

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 an 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.
Bibliography


[10] [genetic algorithms: A survey.


Appendix A

Gacuda User Manual

Welcome to the Gacuda User Manual. This manual will help you build and deploy your genetic algorithm applications on GPUs using the Gacuda framework.

This guide assumes GNU/Linux will be used but you are free to adapt this guide to other operating systems of your preference.

A.1 Getting Started

Before you start, you should confirm that the minimum requirements are satisfied:

- Computer with a Nvidia GPU that supports GPGPU.
- GNU/Linux with the C development tools installed.
- Version 7 (or greater) of Nvidia Cuda Toolkit installed.

After confirming all the requirements Gacuda framework can be downloaded and extracted. The resulting files will be:

- loader.h: Not to be modified. Contains Gacuda framework code responsible for loading the configuration file. (Header files containing code are necessary due to NVCC Cuda compiler limitations.)
- gacuda.h: Not to be modified. The actual include file where all the Gacuda framework external intricacies are declared, including the Gacuda provided macros.
- gacuda_code.h: Not to be modified. The Gacuda framework engine code and GPU kernels.
- app.cu: Sample code file, contains all the functions declared. The user only needs to add the functions’ body.
• app.conf: Sample configuration file, contains all the recommended configuration values, ready to be modified by the user.

Gacuda does not need to be built, only when you compile your application will Gacuda be built along with your application.

A.2 Development

Gacuda requires only the development of a few functions in C and the set up of a configuration file.

A.2.1 Headers

For the code, a single .cu file can be used. The first lines of it must include the Gacuda framework header files. The start of the .cu file should contain, at least, the following lines:

```c
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "gacuda.h"
#include "loader.h"
#include "gacuda_code.h"
```

Listing A.1: Application headers.

A.2.2 Macros

Including the file gacuda.h makes the Gacuda macros available. These are fundamental for the development of the necessary Gacuda functions. The macros available are:

- PopulationSize: Integer. The number of individuals in the whole population.
- NumberOfGenes: Integer. The number of genes in an individual.
- GenomeSize: Integer. The size (in bytes) of an individual’s genome. Equals to NumberOfGenes × GeneSize.
- Input: Pointer. Address of the common read-only memory array. This data can be accessed by all individuals.
- InputSize: Integer. The size (in bytes) of the common read-only memory array.
- Rand(): Integer. Function macro that allows to generate a random number in the functions that are executing in the GPU. Value between 0 and RNG_MAX.
- RNG_MAX: Integer. Contains the maximum value that is possible to be returned by Rand(). Equals to 2147483647.
A.2.3 Functions

With the headers declared and with the Gacuda Macros available, the required C functions can be developed. These are:

**initialize_gene**

Function to initialize the population, individual by individual. It is executed by the framework only once for each individual, at the start of the execution. The parameter `genome` allows access to the individual’s genes.

```c
__host__ void initialize_gene(void * genome);
```

Listing A.2: `initialize_gene` function declaration.

**initialize_population**

Function to initialize the population, all at once. It is executed by the framework only once, at the start of the execution after calling the function `initialize_gene` for all genes. Note that just one initialization function can be used. The parameter `population` allows access to the all the individuals’ genes.

```c
__host__ void initialize_population(void * population);
```

Listing A.3: `initialize_population` function signature.

**fitness**

The objective function. Should return a float indicating how fit the individual is. This function will be executed every iteration of the algorithm for every individual. The parameter `genome` allows access to the individual’s genes.

```c
__host__ __device__ float fitness(void * genome);
```

Listing A.4: `fitness` function signature.

**crossover**

This function should combine the parents’s genomes into a new genome. It is executed, based on a configured probability, every generation for every individual. The parameter `dest` is where the newly combined individual genome must be written. The parameters `genome1` and `genome2` allow access to the parents’ genomes.

```c
__device__ void crossover(void *dest, void * genome1, void * genome2);
```

Listing A.5: `crossover` function signature.
mutation

This function should modify an individual genome slightly. It is executed, based on a configured probability, every generation for every individual. The parameter genome allows access to the individual’s genes.

```c
__device__ void mutation(void *genome);
```

Listing A.6: mutation function signature.

output

This function is optional. It is used to pretty print individuals. The parameter genome allows access to the individual’s genes. The parameter result is where the resulting string, representing the individual should be written. Memory should be allocated for this beforehand.

```c
__host__ void output(void *genome, char *result);
```

Listing A.7: output function signature.

A.2.4 Configuration

You also have to set up the Gacuda configuration file for your application. The entries of the configuration file are all shown in this section. For easier reference they were grouped in three different groups.

Main Configuration Entries

- **number_runs**: Decimal. The number of generations that will execute before ending the algorithm. This is the termination condition.
- **population_size**: Decimal. The total number of individuals in the whole population.
- **number_genes**: Decimal. The number of genes in each individual.
- **gene_size**: Decimal. The size (in bytes) of a gene.
- **threads_per_block**: Decimal. The number of threads that a Cuda block will have. Corresponds to the size of the islands. The population_size must be divisible by threads_per_block or execution will fail.
- **input_size**: Decimal. The size (in bytes) of the common read-only memory to allocate. Useful for storing shared read-only data between all individuals that does not need to make part of the genome.
- **device_list**: String. A list of the identifiers of the GPUs to run the algorithm. The identifiers must be separated by a semi-colon. The identifiers of the available GPUs can be seen with the Nvidia tool nvidia-smi.
• **crossover_rate**: Float. Probability of executing the crossover function, for one individual, in one generation. Value between 0 and 1.

• **mutation_rate**: Float. Probability of executing the mutation function, for one individual, in one generation. Value between 0 and 1.

• **elitism**: Boolean. Activates or deactivates the elitism functionality. Elitism keeps the best individual of a population alive for the next generation.

**Migration Configuration Entries**

• **migration**: Boolean. Activates or deactivates the (inter) migration functionality. Allows sharing genes between different GPUs.

• **migration_rate**: Decimal. The number of generations between a (inter) migration.

• **migration_size**: Decimal. The number of individuals to migrate per GPU.

• **intra_migration**: Boolean. Activates or deactivates the (intra) migration functionality. Allows sharing genes between different Cuda blocks.

• **intra_migration_rate**: Decimal. The number of generations between a (intra) migration.

• **intra_migration_size**: Decimal. The number of individuals to migrate per block.

**Diagnostics Configuration Entries**

• **output_input**: Boolean. Enables or disables the printing of the initial population to the console at the start of the algorithm.

• **output_score**: Boolean. Enables or disables the printing to file of the scores of every individual at the end of the algorithm.

• **output_score_file**: String. The name of the file to write all the scores.

• **output_individuals**: Boolean. Enables or disables the printing to file of all the genes of every individual at the end of the algorithm.

• **output_individuals_file**: String. The name of the file to write all the genes.

• **statistics**: Boolean. Enables or disables the statistics kernel. Allows printing information about the best, average and worst fitness values of the whole population while the algorithm is executing.

• **output_best**: Boolean. Enables or disables printing the best individual fitness value at the end of the algorithm. Requires running the fitness function on the CPU for all the population.

• **trace_best**: Boolean. Enables or disables printing the best individual fitness value for each block at every generation. Requires the elitism functionality. Might generate too much information.
A.3 Building

With the code ready, it is possible to compile the developed application. A Makefile will be used for this. All that is necessary is to run the command `make`. If you are using another source file then the first line of the Makefile should be modified accordingly.

```make
APP = app
NVCC = /usr/local/cuda/bin/nvcc
OPTIONS = -gencode=arch=compute_20,code=\"sm_20,compute_20\" \ 
    --ptxas-options=-v -use_fast_math -lm -maxrregcount 32

# END OF CONFIGURABLE ZONE

clean:
    rm $(APP)

all: $(APP)

$(APP): $(APP:=.cu) gacuda.h loader.h gacuda_code.h
    $(NVCC) -g -G -o $(APP) $(APP:=.cu) $(OPTIONS)

Listing A.8: Makefile for Griewank example.

A.4 Executing

After building your application, you can execute the genetic algorithm. The application takes one argument, the configuration file path.

```
alucart@unix ~/gacuda.git/src $ ./app app.conf
...
GACUDA started.
...
GA Execution ended.
Outputing scores to output/scores.dat.
Outputing individuals to output/genes.dat
All done.
```

Listing A.9: Executing the Griewank example.
A.5 Example Source-Code

An example of the use of the Gacuda framework for numerical optimization is provided in this section.

A.5.1 Griewank

Configuration File

```plaintext
number_runs = 1000
population_size = 262144
number_genes = 64
gene_size = 4
threads_per_block = 128
crossover_rate = 0.8
mutation_rate = 0.1
elitism = false
input_size = 0
migration = false
migration_rate = 100
#migration_rate = 2
migration_size = 0
intra_migration = false
intra_migration_rate = 10000
intra_migration_size = 4

device_list = 0

output_input = false
output_scores = false
output_scores_file = output/scores.dat
output_individuals = false
output_individuals_file = output/genes.dat
statistics = false
output_best = false
trace_best = false
```

Listing A.10: Configuration file for Griewank example.
**initialize_gene**

```c
__host__ void initialize_gene(void * genome) {
    float *gene = (float *) genome;

    for (int i = 0; i < NUMBER_GENES; i++) {
        gene[i] = (LIMIT_MAX - LIMIT_MIN+1) *((float) rand() / RAND_MAX) + LIMIT_MIN;
    }

    return;
}
```

Listing A.11: initialize_gene function for Griewank example.

**initialize_population**

```c
__host__ void initialize_population(void * population) {
    return;
}
```

Listing A.12: initialize_population function for Griewank example.

**mutation**

```c
__device__ void mutation(void * genome) {
    float *gene = (float *) genome;

    gene[RAND() % NUMBER_GENES] = (((float)rand()) / ((float)RNG_MAX) ) * (LIMIT_MAX - LIMIT_MIN)) + LIMIT_MIN;

    return;
}
```

Listing A.13: mutation function for Griewank example.
fitness

```c
__host__ __device__ float fitness(void *genome)
{
    float *gene = (float *)genome;
    float part1 = 0.0f;
    float part2 = 1.0f;

    for (int i = 0; i < NUMBER_GENES; i++)
        part1 += gene[i] * gene[i];

    for (int i = 0; i < NUMBER_GENES; i++)
        part2 *= cos((float)gene[i] / sqrt((float)i+1.0f));

    return (float)(-1.0f * (1.0f + ((float)part1)/4000.0f - ((float)part2)));
}
```

Listing A.14: fitness function for Griewank example.

crossover

```c
__device__ void crossover(void *dest, void *genome1, void *genome2)
{
    float *dest_gene = (float *)dest;
    float *gene1 = (float *)genome1;
    float *gene2 = (float *)genome2;

    for (int i = 0; i < NUMBER_GENES / 2; i++)
        dest_gene[i] = gene1[i];

    for (int j = NUMBER_GENES / 2; j < NUMBER_GENES; j++)
        dest_gene[j] = gene2[j];

    return;
}
```

Listing A.15: crossover function for Griewank example.
__host__ void output ( void * genome , char * result )
{
    char str [1024];
    float * gene = (float *) genome;

    str[0] = '\0';

    for ( int i = 0; i < NUMBER_GENES; i++ ) {
        sprintf (str , "%.15f ", gene[i]);
        strcat (result , str);
    }

    return;
}

Listing A.16: output function for Griewank example.