March 2019

Parallelization of Genetic Algorithm to Solve MAX-3SAT Problem on GPUs

Prakruthi Shivram
University of South Florida, prakruthis@mail.usf.edu

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation
Shivram, Prakruthi, "Parallelization of Genetic Algorithm to Solve MAX-3SAT Problem on GPUs" (2019). Graduate Theses and Dissertations. https://scholarcommons.usf.edu/etd/7916

This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Parallelization of Genetic Algorithm to Solve MAX-3SAT Problem on GPUs

by

Prakruthi Shivram

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Srinivas Katkoori, Ph.D.
Yicheng Tu, Ph.D.
Hao Zheng, Ph.D.

Date of Approval:
March 1, 2019

Keywords: Parallel Computing, CUDA, Combinatorial Optimization

Copyright © 2019, Prakruthi Shivram
DEDICATION

This study is dedicated to my beloved parents, who have been my source of inspiration, who continually provide their moral, spiritual, emotional, and financial support.

To my brother, relatives, mentor, friends, and colleagues who shared their words of advice and encouragement to finish this study.

And lastly, I dedicated this book to the Almighty God, thank you for the guidance, strength, power of mind, skills and for giving me a healthy life.
ACKNOWLEDGMENTS

I would first like to formally acknowledge Dr. Srinivas Katkoori for providing me the opportunity to work on this project. I would like to thank him for his constant help and guidance. I am forever grateful to him. I would also like to thank Dr. Yicheng Tu for his guidance, assistance, and feedback throughout this work. I give thanks to Dr. Hao Zheng for volunteering his precious time to serve as a member on my thesis committee. I would like to thank my family for their love and support. I would like to thank Hao Li for providing assistance related to the GPU cluster access. I would also like to thank my roommates, friends and the colleagues I have acquainted for their help and association.
# TABLE OF CONTENTS

LIST OF TABLES................................................................................................. iii

LIST OF FIGURES................................................................................................ iv

ABSTRACT .......................................................................................................... v

CHAPTER 1: INTRODUCTION AND MOTIVATION........................................ 1
1.1 Advantages of GPU .................................................................................. 3
1.2 Applications of MAX-3SAT .................................................................... 3
1.3 Goals ......................................................................................................... 3
1.4 Thesis Organization .................................................................................... 4

CHAPTER 2: BACKGROUND AND RELATED WORK .................................. 5
2.1 Conjunctive Normal Form ...................................................................... 5
2.2 SAT .......................................................................................................... 5
2.3 Heuristics ................................................................................................ 6
2.4 Genetic Algorithm .................................................................................... 7
2.5 Genetic Operators .................................................................................... 8
  2.5.1 Fitness Function .................................................................................. 8
  2.5.2 Selection ............................................................................................. 8
  2.5.3 Crossover ........................................................................................... 10
  2.5.4 Mutation ........................................................................................... 10
  2.5.5 Termination ......................................................................................... 11
2.6 Parallel Genetic Algorithms .................................................................. 11
  2.6.1 Single Population Master Slave Parallel Genetic Algorithm ............ 11
  2.6.2 Coarse Grained Parallel Genetic Algorithm ..................................... 12
  2.6.3 Fine Grained Parallel Genetic Algorithm .......................................... 12
2.7 Serial Genetic Algorithm ........................................................................ 13
2.8 GPU ......................................................................................................... 15
  2.8.1 What Is a GPU? .................................................................................. 17
  2.8.2 What are CPUs and GPUs? ............................................................... 18
  2.8.3 What is the Difference? ..................................................................... 18
2.9 CUDA ....................................................................................................... 19
  2.9.1 Memory Hierarchy ............................................................................ 20
2.9.2 CUDA Memory and Cache Architecture ................................ 21
2.10 Chapter Summary .............................................................. 22

CHAPTER 3: PROPOSED PARALLEL GA ON GPU................................. 23
3.1 Proposed Implementation ...................................................... 23
   3.1.1 CPU and GPU Relationship ............................................ 24
3.2 Data Organization ............................................................... 25
3.3 GPU Implementation of Genetic Algorithm to Solve MAX-3SAT Problem ...... 25
3.4 Chapter Summary ............................................................... 29

CHAPTER 4: EXPERIMENTAL RESULTS................................................. 30
4.1 Chapter Summary ............................................................... 32

CHAPTER 5: CONCLUSION AND FUTURE WORK....................................... 33
5.1 Future Enhancements............................................................ 33

REFERENCES .............................................................................. 34
LIST OF TABLES

Table 4.1 Nvidia GeForce GTX Titan X Card Specifications ................................ 30
Table 4.2 Example Instances of SAT Benchmarks ................................................ 31
Table 4.3 Fitness Results for Serial and CUDA Versions of Genetic Algorithm ........ 31
LIST OF FIGURES

Figure 2.1 Different Crossover Operations ................................................. 10
Figure 2.2 Mutation Operation ................................................................. 11
Figure 2.3 Master Slave Genetic Algorithm ................................................. 12
Figure 2.4 Serial Genetic Algorithm ............................................................. 14
Figure 2.5 Bit Representation of Genetic Algorithm ........................................ 15
Figure 2.6 GPU Architecture ....................................................................... 17
Figure 2.7 Difference in CPU and GPU Architecture ...................................... 18
Figure 2.8 Thread Hierarchy ......................................................................... 20
Figure 2.9 Cache Hierarchy .......................................................................... 21
Figure 3.1 Flow of Execution in CUDA Program on GPU ................................. 24
Figure 4.1 Performance Comparison for Instances with 20 Variables ................. 31
Figure 4.2 Performance Comparison for Instances with 50 Variables ................. 32
Figure 4.3 Performance Comparison for Instances with 250 Variables ............... 32
ABSTRACT

There are many combinatorial optimization problems such as flow shop scheduling, quadratic-assignment problem, traveling salesman problem, that are computationally intractable. Genetic Algorithm is a heuristic algorithm used to find an answer to combinatorial optimization problems. MAX-3SAT is an example of combinatorial optimization problem which has wide range of applications as many real world problems can be translated to MAX-3SAT problem. Genetic algorithms are suitable to solve MAX-3SAT problems but usually undergo premature convergence. To prevent this convergence and maintain diversity, one possible solution is to use large population size. This increases computation cost and time. Since Genetic Algorithms compute the same fitness function on large data (population), it provides data and instruction parallelism. Hence Genetic algorithm can be scaled on to GPU architecture. GPUs are affordable, efficient parallel computing hardware. Hence in this thesis, we use CUDA framework to implement a parallel version of Genetic Algorithm on GPU. We use the MAX-3SAT problem to verify our algorithm. Compared to the CPU implementation with similar workload, the proposed GPU implementation is upto four times faster and often finds better results.
CHAPTER 1: INTRODUCTION AND MOTIVATION

Combinatorial optimization problem is a problem of finding an optimal object from a fine set of objects. Many combinatorial optimization problems are computationally hard. The methodologies used to solve such problems are classified as exact and heuristic algorithms. Combinatorial optimization algorithms can be further classified into two categories which are exact and heuristic algorithms.

• Exact algorithm ensures optimality for all the instances of the problem and also ensures finding an optimal solution. The run-time is directly proportional to size of the problem instance and it increases exponentially with increase in population size. Instances which are small or medium sized can only be solved in this algorithm. Exact method requires more considerable implementation resources.

• On the other hand, heuristic algorithms are considered for larger instances and they ensure near optimal solution in a reasonable run-time. But it compromises the solution quality which is guaranteed in the exact algorithm. So heuristic algorithms are used when we need to solve the problem faster but by sacrificing accuracy, completion, and optimality.

Boolean satisfiability (SAT) is a well-known computationally hard problem. The problem is to find an assignment of binary values which satisfy a given boolean formula. We can say that a Boolean formula is satisfied if all clauses are fulfilled. SAT problem can be applied to many problems in artificial intelligence, design logic, theoretical computer science, and numerous other problems.
The interest in this problem is motivated by a wide spectrum of implementations. The maximum satisfiability (MAX-SAT) problem is a variation of the SAT problem where, for a given Boolean assignment, we need to find the maximum number of clauses. MAX-3SAT is a SAT problem where each clause has exactly three literals making it an NP-hard problem which can be solved using combinatorial optimization algorithms. In this work, we consider a heuristic population-based method to solve the MAX-3SAT problem, we use genetic algorithm (GA) approach to solve our problem [1].

MAX-SAT problems are frequently solved by using Genetic Algorithms, but simple genetic algorithms are usually prone to premature convergence due to lack of diversity, thus reducing their efficiency. To prevent them from converging prematurely, different methods are used, for example, using bigger population size, using sub-population based algorithm, using local search and niching. All these implementations increase the computational cost. To reduce this cost, we can employ parallel computing and hence in our thesis we implement the algorithm on GPU architecture to achieve maximum parallelism.

GPUs are less expensive and low energy consuming parallel computing resources. GPUs were originally designed for computer graphics and hence are highly parallel, multithreaded, and multi-core processors. General Purpose processing using GPU (GPGPU) is the use of GPU for non-graphics applications. GPUs have less general processing compared to CPUs, this is because GPUs have an architecture designed to give more importance to data processing relatively than control flow and data caching.

CUDA is Compute Unified Device Architecture a programming model by Nvidia to support parallel capabilities of GPUs. The use of General Purpose GPUs in a wide range of problems has
increased drastically since the release of CUDA. The reason being that CUDA framework is well
documented now. The advantage of CUDA is that it is easy to understand because it employs the
practices of C programming language.

1.1 Advantages of GPU

With the advent of General Purpose GPUs, the use of GPUs is now well documented. GPU
is convenient to tackle those problems which can be represented as parallel data computations
and which have high arithmetic intensity. GPUs can offer huge performance gain when used for
applications which process large data sets [2].

1.2 Applications of MAX-3SAT

Many real world problems are optimization based, and so, are more suitable to be trans-
lated to Max-SAT, e.g., FPGA Routing [3], Design Debugging [4], Bioinformatics, Scheduling, and
Probabilistic Reasoning [5]. Max-Sat has been applied to many fields, few examples are given below:

- Employee schedules can be defined as a Max-SAT problem, as we need to find more efficient
  schedules.

- Max-SAT can be used to help in automation of the detection of faults in energy consumption
  behaviors, this acts as a way to remove energy bugs in mobile devices.

1.3 Goals

In this thesis, we propose a genetic algorithm on GPU to solve the MAX-3SAT problem.
The main goals for this thesis can be described as:
1. To develop a CPU only serial Genetic algorithm program in C to solve MAX-3SAT.

2. To develop a GPU-CPU genetic algorithm program in CUDA to solve MAX-3SAT.

3. To compare the performance of the two programs.

4. Explore and implement further variations of the CUDA program.

5. Perform a critical analysis of the results.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 presents the concepts of Max-3SAT, Genetic algorithm, GPU, and CUDA as well as related work on parallelization of Genetic Algorithm. Chapter 3 proposes a parallel Genetic algorithm on GPU. Chapter 4 reports the results and performance of the parallel version when compared to the serial version. Chapter 5 concludes the work done and outlines directions for further enhancement.
CHAPTER 2: BACKGROUND AND RELATED WORK

This chapter gives brief description of the basic concepts involved in this thesis. This includes definition of the 3SAT problem, description of the genetic algorithm used and the technology used to solve the 3SAT problem.

2.1 Conjunctive Normal Form

The representation of a logic formula where the formula is a conjunction of clauses and each clause is a disjunction of literals is called Conjunctive Normal Form (CNF). Each literal can be either a variable or its negation. Example, given the variables x1 and x2, we can have a possible clause as follows:

\[(x_1 \lor x_2 \lor \neg x_1)\] (2.1)

where \(x_1\) is a Boolean variable and \(\neg x_1\) is the negation of the Boolean variable \(x_1\). The following example shows how an expression is represented in CNF. It consists of multiple clauses:

\[
(x_1 \lor \neg x_2) \land (\neg x_3 \lor x_2) \land (\neg x_1 \lor x_3) 
\] (2.2)

2.2 SAT

The boolean satisfiability problem (SAT) is defined as a problem of finding an assignment for each of the variables, given a logic expression in Conjunctive Normal Form (CNF) such that it
satisfies the Boolean expression or stating that no such assignment exists. The Boolean variables can have either value TRUE or FALSE. The Boolean expression is said to be satisfied if the formula calculates to value TRUE. This is possible only if each clause in the expression estimates to TRUE. The 3SAT problem is a Boolean expression reduced to 3CNF. Here the number of literals in a clause is limited to at most three literals. An example is give below:

\[(x_1 \lor \neg x_2 \lor x_6) \land (x_3 \lor x_7 \lor x_2) \land (x_1 \lor x_3 \lor x_2) \land (x_3 \lor x_5 \lor x_9) \land (x_{10} \lor x_7 \lor x_3)\] (2.3)

The MAX-SAT problem is defined as a problem of finding an assignment for each of the variables, given a logic expression in conjunctive normal form (CNF) such that we have the maximum possible number of true clauses.

2.3 Heuristics

It is a common procedure to try speeding up the performance of search based algorithms by predicting results during execution. These predictions in most cases, cause significant increase in performance and decrease in need for resources. Such improvement techniques are referred to as heuristics. Particularly with incomplete or imperfect information or limited computation capacity, a high-level procedure or a heuristic is designed to search, generate, or choose a heuristic (restricted search algorithm) called metaheuristic which can offer a sufficiently good answer to an optimization problem [6].

In order to be usable for a variety of problems Metaheuristics may make few assumptions about the optimization problem being solved. There are many types of metaheuristic algorithms. They can be classified as follows:
• **Local Search** and **Global Search**: simulated annealing, tabu search etc., are metaheuristics that improve local search metaheuristics. Population based metaheuristics like particle swarm optimization, evolutionary computation, ant colony optimization, etc are few examples of global search metaheuristics.

• **Single-solution** and **Population-based**: In single-solution metaheuristic we concentrate on transforming and enhancing a lone candidate solution. Example, simulated annealing, iterated local search and guided local search. Whereas population-based metaheuristic concentrates on transforming and enhancing multiple candidate solutions. Examples contain swarm optimization, genetic algorithms, and evolutionary computation.

• **Hybridization** and **Memetic algorithms**: Hybridization heuristic is a combination of a general metaheuristic along with other optimization approaches like machine learning. Memetic heuristic is a combination of local search algorithm and evolutionary algorithm.

• **Parallel metaheuristics**: This heuristic runs multiple metaheuristic simultaneously by using parallel programming methods. Examples can be simultaneous search runs to enhance the final solution or easy distributed schemes.

### 2.4 Genetic Algorithm

Genetic algorithm (GA) is a metaheuristic based on the process of natural selection and is affiliated to the larger class of evolutionary algorithms (EA). Genetic algorithms are frequently used to generate extremely good solutions for various problems such as Travelling Salesman problem, Satisfiability problem, machine learning, function optimization, scheduling etc. In Genetic Algorithms, a generation represents individuals participating in the evolution process. A population is a set of
chromosomes, on which genetic operations are performed. In the beginning of the algorithm, fitness function is used to evaluate current population. Then genetic operations are performed on this population to generate new population, the new individual created after applying the operations is called offspring or child. Genetic Algorithm consists of the following terminologies:

- Initial Population: It is the initial set of possible solutions (individuals) for a given problem. It consists of set of unique chromosomes at that point in time.

- Chromosome: An individual is defined by a set of parameters joined together to form a chromosome (solution).

Genetic Algorithms mostly contain Genetic Operators to perform the required evolution.

2.5 Genetic Operators

2.5.1 Fitness Function

It is used to calculate fitness of each individual in a population. Fitness represents the quality of an individual. The individuals with higher fitness value are more likely to be passed on to next generation. The fitness function depends on the problem given. It takes as input a single chromosome and generates a value as a measure of the fitness of that chromosome.

2.5.2 Selection

This phase determines the top fittest individuals in order to pass them onto the next generation. The selected individuals are called parents. There are many types of selection operations:
• Roulette Wheel Selection

This method assigns a value for each individual which determines the probability of its selection. This value is the ratio of fitness of given individual over the total fitness of all individuals in the given generation. The parent individuals are selected randomly based on their probability of selection. This type of selection prevents premature convergence by maintaining diversity within the population.

• Tournament Selection

This method involves running many tournaments among tournament subset chromosomes [7]. The individual with best fitness inside this tournament subset is chosen as the parent chromosome for further genetic operations like crossover and mutation. This method has an advantage of applying additional selective pressure compared to roulette selection. The selection pressure is the probability that a chromosome will participate in the tournament and is based on the participant selection pool size. When tournament size is one, the method is equivalent to random selection.

• Elitism

This method ensures that the chromosomes of the best solution are passed on to the new population. Usually the best chromosomes are lost after crossover and mutation operations are performed. To prevent loss of the fittest chromosome during new population creation, this method initially replicates the best chromosomes to the new population. The crossover and mutation operations are then performed as usual.
2.5.3 Crossover

This phase selects a random crossover point for each pair of selected parents and exchanges the bits of the parents to create child individuals. Crossover is performed to pass characteristics of previous generations to future generations. Two parent individuals are combined to form two off-springs (child individuals). The common types of crossover are single-point crossover, two-point crossover and uniform crossover [7]. Figure 2.1 illustrates these three types of crossover operations.

![Different Crossover Operations](image)

**Figure 2.1: Different Crossover Operations.**

2.5.4 Mutation

Some bits of a randomly chosen child individual is inverted to maintain diversity and prevent the program from getting stuck at the local optimum. Figure 2.2 gives an illustrative example.
The other genetic operations are performed on the given population until a termination condition is met. This may be when a solution that satisfies the problem is found or when the fixed number of iterations is reached or when the fitness of the fittest individual doesn’t increase for many successive generations or when the program runs out of resources (time).

2.6 Parallel Genetic Algorithms

There are three common types of parallel genetic algorithms.

2.6.1 Single Population Master Slave Parallel Genetic Algorithm

In this type of parallel algorithm the fitness calculation is distributed among several processors. Hence this is called global parallel genetic algorithm [7]. Selection and crossover operations play a big role in this type of implementation. In this algorithm, a single large population acts as a master and distributes fitness calculation to different computers as shown in Figure 2.3. This method is used to increases the speed and calculation power of the genetic algorithm. But, the disadvantage of this method is that the speed is limited by the slowest node as the master has to wait for the results from the slave nodes.
2.6.2 Coarse Grained Parallel Genetic Algorithm

This type of Genetic Algorithm is the basic requirement in order to implement Genetic algorithm on GPUs with large amount of threads. The population is divided into sub populations which are operated on separately by different processors. The advantage in this type of algorithm is that for small populations, the individuals are spread faster within the population. However, when compared to a single large population, the increase in fitness stops at a lower value. When the migration rate between the sub-populations is lowered, the final quality of the result is lowered [8]. Coarse grained parallel genetic algorithm is faster compared to the master-slave parallel algorithm and it also has more diversity of the population.

2.6.3 Fine Grained Parallel Genetic Algorithm

This type is better than the previous coarse grained algorithm as it has better population diversity. It also prevents premature convergence and has optimal parallelism with better results, making it better than all other simple genetic algorithms. This type of algorithm prevents population
from communicating with its direct neighbors but these neighborhoods can overlap. For this type of parallel algorithm, as the population size increases, the performance decreases [9]. This type of algorithm model has best performance with good results when implemented on massively parallel computers.

2.7 Serial Genetic Algorithm

Here, the final solution is obtained by modifying and evolving the initial solutions. This evolution follows the genetic algorithm as illustrated in Figure 2.4. In this algorithm, first a set of initial solutions known as population is created by randomly initializing a set of individuals known as chromosomes.

In this thesis, a chromosome is bit-array of zeros and ones, where nth bit represents the value of the nth variable in the expression. Then we calculate the fitness value for each individual. For our problem the fitness is the sum of truthful clauses for a given assignment of chromosome. This step is called as population evaluation. Next we perform the process of evolution. The genetic operator functions like selection, crossover and mutation are performed on the initial population until the termination condition is met. The bit representation of this is illustrated in Figure 2.5.

When a possible solution for the problem is found or if the maximum number of generations was reached then the termination condition is met. In the selection phase two parent chromosomes are selected randomly in order to produce new population chromosomes. In the crossover stage a random number is generated which defines the crossover point and the bits of the parent chromosomes are exchange with the crossover point as pivot. The new chromosomes formed are added to the new population if their fitness is greater than the parent fitness.
Next, mutation operation is performed depending on the mutation rate. In mutation function some bits of a randomly chosen chromosome is flipped, i.e if the nth bit is zero it is flipped to one and vice versa. Now the new population is evaluated by calculating their fitness value and the termination condition is checked. The above three processes are repeated again if the ending condition is not met. After the termination condition is met the final result is selected.
condition is not met then. The chromosome with maximum fitness value is returned if stop condition
is satisfied.

2.8 GPU

A Graphic Processing Unit (GPU) is a special circuit consisting of single-chip processor. GPU’s
primary focus is to control and amplify the performance of the graphics and videos. The wide range
of features contain:

- Supporting high-intensity graphics software
- Hardware overlays
• 2-D or 3-D graphics
• Texture mapping

In order to reduce the CPUs work and produce better videos and graphics, the above-mentioned features are carefully designed. Along with its use in PC on a motherboard or video card, GPU is used in workstations, display adapters, mobile phones, game consoles and many more. It is also called as Visual Processing Unit (VPU) In 1999, Nvidia developed the first GPU, a single-chip processor and called it as GeForce 256. It included features like triangle setup/clipping and rendering engines, lighting effects, integrated transform, drawing and BitBLT support. The model had more than 22 million transistors and could perform 10 million polygons per second.

The popularity of GPUs increased drastically with the increase in the demand for more graphic applications. Ultimately, they were no more used just for the enhancements of the graphic features but became a necessity for optimum performance of a computer. In the general set-up, the GPU and CPU are connected, but GPU is separated from motherboard. There are some GPUs where they make use of the main memory as a digital storage area as they are combined into the northbridge on the motherboard. Due to this, the GPUs speed is reduced resulting in poor performance.

The maximum percentage of GPUs make use of their transistors for 3-D computer graphics. However, there are some GPUs like Geographic Information System (GIS) applications which have accelerated memory for mapping vertices. Most of the engineers and scientists make use of GPUs for studying and solving the complex problems exploiting vector and matrix features.

The Central Processing Unit (CPU) is termed as the brain of the computer, but the brain is boosted up by another part of the system called as GPU. GPU’s architecture is shown in Figure2.6
as explained in [10]. It is a powerful programmable and computational device on its own and way ahead compared to basic graphics controller functions. The computer system consists of chips required for rendering the images to monitor, but the chips are not equally created. Intels integrated graphics controller offers primitive graphics which can only display productivity applications like gaming applications, less quality videos, and Microsoft PowerPoint.

### 2.8.1 What Is a GPU?

The advance capabilities of GPU were initially used for 3D game rendering. In recent years they are more broadly used to accelerate computational workloads in areas such as oil and gas exploration, state-of-art scientific research, and finance modeling. GPUs and CPUs have a lot in common. Both are made up of silicon-based microprocessors. However, they are noticeably different and are used for different roles. Their difference is shown in Figure 2.7.
2.8.2 What are CPUs and GPUs?

The Central Processing Unit (CPU) is usually terms as the brain or heart of a computer system. It helps in running most of the engineering and office software. The CPU will be overpowered if the number of tasks increases. At this stage GPU play an important role in computations. The Graphic Processing Unit (GPU) is a specific kind of microprocessor mainly designed for fast image interpretation. They acted like a reply to applications which demands high graphically intensity and thus reduce the burden on CPU and helped in maintaining the computers performance.

They were one of the methods to unload the tasks from CPU until the modern graphics processors which are powerful enough to perform fast mathematical calculations for many other applications apart from image interpretation.

2.8.3 What is the Difference?

The GPUs and CPUs way of processing tasks is different. They are compared with the brain and brawn of the human system. The Central Processing Unit (CPU) which is called as the brain performs variety of calculations, where as the Graphic Processing Unit (GPU) which is the brawn
focuses on computational capabilities of a specific task. This is because the CPU has few cores (up to 24) specialized for sequential serial processing and it increases the performance of the single task within the list. However, there are wide range of tasks. On the contrary, GPU has thousands of small and efficient cores specialized in parallel processing which can handle multiple tasks at the same time. Modern GPUs are 50-100 times faster when it comes to tasks which need parallel processing like machine learning and big data analysis. They offer high processing power, memory bandwidth and efficiency over their CPU counterparts.

2.9 CUDA

Nvidia created a parallel computing platform and application programming interface (API) model. CUDA allows software developers and software engineers to use the CUDA-enabled graphics processing unit (GPU) for general purpose processing called the GPGPU [11]. The CUDA platform which is a software layer that provides direct access to the GPU virtual instruction set and parallel computational elements, can be used in order to execute the kernels. The programming languages like C, C++ and Fortran can be designed using the CUDA platform. CUDA is acronym for Compute Unified Device Architecture, which was introduced by Nvidia. Nvidias CUDA development platform from the past eleven years has been helpful for general purpose processing in a variety of applications which includes the high-performance computing (HPC), data center applications, and content ranging from the embedded systems to the cloud which are all benefited by the performance of GPUs.
2.9.1 Memory Hierarchy

A kernel is a function executed on the GPU as an array of threads in parallel. All the threads execute the same code which can take different paths. In order to select the input and output data and control decisions the thread has an ID. The CUDA threads are grouped into blocks and these are further divided into grid. A kernel is executed as a grid of blocks of threads. The CUDA memory hierarchy has threads which are subdivided into registers and local memory. The block of threads is called shared memory. All the blocks together form a global memory. The uses of shared memory are sharing data among the threads in a block and user-managed cache. The global memory hierarchy is accessible by all threads of any kernel. The different memories and thread hierarchy is illustrated in Figure 2.8. It provides a bandwidth of 156 GB/s [10].
2.9.2 CUDA Memory and Cache Architecture

It is important to understand the basic memory architecture of the system to create high performance programs. Majority of the desktop systems contain large sizes of system memory which is connected to a single CPU which has two to three levels or fully coherent cache as seen in Figure 2.9. It is necessary to understand why the memory and cache architecture to achieve high performance. A cache friendly process executes faster than a non-cache friendly function. Such an incident is also detected on GPUs when the cache is not used properly.

The main aim of this work is to develop a parallel version of generic algorithm meta-heuristic on Graphical Processing Unit (GPUs) which can be used to solve combinatorial optimization problems MAX-3SAT (Boolean Satisfiability) that are by nature NP-Complete.
2.10 Chapter Summary

In this chapter, the challenges involved in solving the combinatorial optimization problems and the approaches to solve them are explained. The motivation behind this work is also discussed. The overview of the thesis is presented.

This chapter also discussed existing search optimization technologies and so far what work has been done. Characteristics of GPU are discussed and are compared with CPU to better understand their behavior.
CHAPTER 3: PROPOSED PARALLEL GA ON GPU

Here, we discuss the proposed implementation of genetic algorithm on GPUs to solve 3-SAT problem.

3.1 Proposed Implementation

This section describes the implementation of the GA based algorithm on GPU using CUDA language. The first part gives an overview of the flow of a general program on GPU and the next part explains each section of our proposed algorithm explicitly in detail. The goal of this thesis is to gain maximum speedup without compromising the quality of the solution.

The execution of a program in CUDA involves two sections, the host part and the device part. The CPU represents the host part execution and the device part execution. The host plays the part of a controller and the GPU is responsible for providing the required computational resources.

In our implementation the host part involves operations like reading cnf file, population initialization, data transfer from host to device etc. The device part involves kernel operations which perform fitness calculation, population selection, crossover and mutation. The Figure 3.1 shows the flow of general CUDA program on GPU [7].
### CPU and GPU Relationship

GPU acts as extra computational device required by the CPU in order to gain faster execution of instructions. CUDA is a GPU specific language which has APIs providing a set of library functions which can be integrated along with C language. The CPU sees the GPU CUDA device as a multicore co-processor. The threads in the CUDA processor run parallely in sets of warp size. A block is a group of threads, and a set of blocks form a grid.

The first step in writing a cuda program is to allocate memory. To do this first pointers to the host data and device data are declared separately. Then these pointers are allocated with Malloc and CUDAMalloc. Next step involves initializing host data and then copying the data from host to device using CUDAMemcpy(HostToDevice). Now we need to determine the kernel configuration like number of threads and number of blocks required. Kernel code to be run on the device is defined before it is called. The program execution now shifts to GPU where the

<table>
<thead>
<tr>
<th>STEP</th>
<th>CPU TASK</th>
<th>GPU RELATED TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Declare pointers to host data</td>
<td>Declare pointers to device data</td>
</tr>
<tr>
<td>2</td>
<td>Allocate host pointers with Malloc</td>
<td>Allocate device pointers with CUDAMalloc</td>
</tr>
<tr>
<td>3</td>
<td>Populate input data pointers of host. Copy data from host to device using CUDAMemcpy (with parameter HostToDevice)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Specify number of blocks and number of threads (kernel configuration)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Specify kernel code (code to be run on device for each thread) and make a call to kernel code</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Perform processing on device</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Copy the result data from device to host using CUDAMemcpy (with parameter HostToDevice)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>If required post process the result on host and present the results to use</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Flow of Execution in CUDA Program on GPU.
code is processed on the device side. The result data is now copied from the device to host using CUDAMemcpy(DeviceToHost). The CPU program can now perform additional modifications to the result data and output the result.

3.2 Data Organization

The first phase of our CUDA based genetic algorithm consists of initialization. In this phase we first read the configuration file and allocate host and device memory. The 3SAT formula is stored in a dimacs file and represented as a 3CNF. We read this problem from a file and store it in an array. The number of variables in the formula and the number of clause in the formula are read from the file. These values are required to initialize the population and evaluate their fitness.

To obtain coalesced memory access the population is represented as a bit array of consecutive individual chromosomes where each chromosome is a series of zeros and ones with a size equal to number of variables in the problem. Once necessary memory is allocated both on the CPU and GPU device, the next step is to assign values to the population. This is done by a kernel call which uses CUDA curand library to randomly initialize the values of the population to either 0 or 1. Here each thread represents a single bit value of the chromosome.

3.3 GPU Implementation of Genetic Algorithm to Solve MAX-3SAT Problem

The initial population has to be evaluated. For this we call a global kernel function where each thread represents an individual chromosome. This function calculates the fitness value for each chromosome or individual in the population by calling a device function and passing arguments. For the given problem, the fitness value is calculated by finding the number of clauses which evaluate
to a positive number. The literals of the clause are dictated by the input array. The variable values are defined by the population array. Thus we need both the input array and initialized population array in order to calculate fitness value. The following three kernel calls are repeated until the termination condition is met. The selection kernel sorts the population in descending order and the top two chromosomes are taken as parent chromosomes. The crossover kernel performs the crossover operation on the chosen two random chromosomes, that is, it exchanges the bits of the two chromosomes around a crossover point based on the crossover rate. The mutation kernel generates a random number and flips a randomly chosen bit of the chromosome. The modified chromosomes are evaluated again and the chromosome with maximum fitness is updated. If the termination is not met, then the above kernel calls are called again. The pseudocode of the proposed algorithm is shown in Algorithm 1.

The Initialization kernel call has a thread size equal to the population size times number of variables. Here each thread accesses one bit of a chromosome or individual. The evaluation Kernel call has thread size equal to number of individuals in the population. Each Thread further calls the device function to calculate fitness. The Selection kernel is called with thread size equal to the number of individuals present in the population. This kernel sorts the individuals based on their fitness. The crossover and mutation kernel calls also have thread size equal to the population size.
Algorithm 1: Proposed Genetic algorithm

Data: CNF file containing problem instance

Result: Values allocated to the individual with best fitness

Initialize CPU data;

Read input problem instance into array A;

Copy CPU data onto GPU;

\( P \) is population size ;

\( N \) is chromosome length ;

Call initial Population Generation Kernel with \( P \times N \) threads;

Call Evaluate Population Kernel with \( P \) threads;

while \( i \leq \text{MaxGeneration} \) do

Call Selection Kernel with \( P \) threads ;

Call Crossover Kernel with \( P \) threads ;

Call Mutation Kernel with \( P \) threads ;

\( \text{fitness} = \text{Maximum fitness for this generation} ; \)

\( i \leftarrow i++ \) end

\( \text{fitness} = \text{Fitness of 1st chromosome} ; \)

The details of the program is given below in steps:

1. Read the specifications of the formula and store the cnf formula in integer array A

2. allocate memory on the CPU and GPU device for arrays representing population \( P \), fitness \( F \), array \( A \).

\text{cudaMalloc((void **)&gpupop, sizeof(int)*popsize*numvar)}
3. Copy array pointers from CPU to GPU using:

   cudaMemcpy(cpuvar, gpuvar, size, cudaMemcpyHostToDevice)

4. Use the cudaError library to check is error has occurred.

5. Call the init kernel function to initialize different states for generating random seeds and also pass the number of blocks and threads being used as their parameters.

   init <<< Blocks, Threads >>> (time(0), states);

   This is done to generate random numbers required for the program.

6. Call the kernel to initialize the current population. This assigns values zero or one randomly to the bit array gpuP. This kernel has arguments gpuP array, numberofvariable and states passed to it.

   Where states is the seed required to generate the random numbers. This kernels goal is to assign random values to gpupopulation array.

   Initpop<< Blocks, Threads >>> (gpuP, numvar, states);

7. Call the kernel to evaluate the fitness of each chromosome C in the population P. This kernel calls the device function to calculate fitness of a single chromosome for N number of times. Where N is the population size (number of chromosomes in a population). The fitness of each chromosome is stored in an integer array gpuF.

   Callfit <<< Blocks, Threads >>> (gpuP, gpuF, numvar, ...);

8. Until the Maximum number of iterations allowed or until best chromosome is found do the following kernel calls:
• Call Kernel to perform selection and sort the current population.

• Call Kernel to perform the cross-over operation on the population.

• Call kernel to perform mutation operation on the population.

9. Return the Chromosome with maximum fitness. Copy the result to CPU.

cudamemcpy(cpuchromosome, gpuchromosome, numvar)

3.4 Chapter Summary

This Chapter discussed in detail the implementation of Genetic Algorithm to solve Max-3SAT problem on GPU including the data organization and the genetic operators parallelized.
CHAPTER 4: EXPERIMENTAL RESULTS

In this chapter, we report the experimental results performed on Nvidia GeForce GTX Titan X card which has 3072 CUDA cores. The specifications of this card is as shown in Table 4.1

Table 4.1: Nvidia GeForce GTX Titan X Card Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Nvidia GTX Titan X cards</td>
<td>4</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>5.2</td>
</tr>
<tr>
<td>Size of a single warp</td>
<td>32</td>
</tr>
<tr>
<td>Number of 32 bit registers per multiprocessor</td>
<td>64KB</td>
</tr>
<tr>
<td>Maximum number of 32 bit registers per thread block</td>
<td>64KB</td>
</tr>
<tr>
<td>Maximum number of 32 bit registers per thread block</td>
<td>64KB</td>
</tr>
<tr>
<td>Maximum number of 32 bit registers per thread</td>
<td>255</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>512KB</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum number of resident grids per device</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of blocks per multiprocessor</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of warps per multiprocessor</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor</td>
<td>2048</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>96KB</td>
</tr>
<tr>
<td>Maximum amount of shared memory per block</td>
<td>48KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>32</td>
</tr>
<tr>
<td>Number of multiprocessors</td>
<td>24</td>
</tr>
<tr>
<td>Total global memory</td>
<td>12GB</td>
</tr>
<tr>
<td>Maximum grid dimensions</td>
<td>2</td>
</tr>
</tbody>
</table>

The specifications for the GPU card are obtained by running a device query program on GPU. For the purpose of our experiments we consider test cases from [12]. Table 4.2 gives one instance of each variable size taken from the benchmark suite. We run 10 instances of 20 variables, 50 variables and 250 variables each and take the average time taken for each variable size.
Table 4.2: Example Instances of SAT Benchmarks

<table>
<thead>
<tr>
<th>Variables</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 variables</td>
<td>uf20-029.cnf</td>
</tr>
<tr>
<td>50 variables</td>
<td>uf50-010.cnf</td>
</tr>
<tr>
<td>250 variables</td>
<td>uf250-075.cnf</td>
</tr>
</tbody>
</table>

The Figure 4.1, Figure 4.2 and Figure 4.3 show the GPU and CPU execution time comparisons respectively for each variable size. We can see that the speedup increases with increase in number of generations and is also with the increase in problem size.

Figure 4.1: Performance Comparison for Instances with 20 Variables.

Table 4.3 shows the average Max Fitness value obtained for the three test cases, for multiple number of generations.

Table 4.3: Fitness Results for Serial and CUDA Versions of Genetic Algorithm

<table>
<thead>
<tr>
<th>Number of variables</th>
<th>20</th>
<th>20</th>
<th>50</th>
<th>50</th>
<th>250</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of generations</td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>32</td>
<td>85</td>
<td>89</td>
<td>199</td>
<td>201</td>
<td>950</td>
<td>969</td>
</tr>
<tr>
<td>64</td>
<td>85</td>
<td>89</td>
<td>199</td>
<td>204</td>
<td>950</td>
<td>971</td>
</tr>
<tr>
<td>1024</td>
<td>86</td>
<td>90</td>
<td>200</td>
<td>205</td>
<td>951</td>
<td>972</td>
</tr>
<tr>
<td>2048</td>
<td>87</td>
<td>91</td>
<td>201</td>
<td>206</td>
<td>951</td>
<td>976</td>
</tr>
<tr>
<td>5000</td>
<td>88</td>
<td>91</td>
<td>201</td>
<td>207</td>
<td>951</td>
<td>984</td>
</tr>
</tbody>
</table>
From the experimental results, we can conclude that the GPU version of genetic algorithm performs far better than the CPU version and is also time-efficient.

4.1 Chapter Summary

This chapter presented the experimental set up used to test the proposed parallel genetic algorithm. Results of each trial are provided in tabular form showing the run time for each version.
CHAPTER 5: CONCLUSION AND FUTURE WORK

This thesis dealt with developing a genetic algorithm to solve MAX-3SAT by utilizing GPU’s advantages. In summary, the parallel implementation of Genetic algorithm on GPU provides speed up of upto four times of magnitude when compared to the serial implementation on CPU with similar work load.

5.1 Future Enhancements

The following are suggestions for enhancing the proposed parallel genetic algorithm.

• Profiling the application in order to present the amount of GPU and memory used.

• Finding ways to increase the efficiency by increasing the number of threads which is possible by optimizing the other resources which occupy more space in memory.

• Implementing Local Search Algorithms along with Genetic algorithms on GPUs to achieve faster and more accurate results.
REFERENCES


