PARALLEL METAHEURISTIC ALGORITHM FOR ROUTE PLANNING USING CUDA

BY

DANIEL LOOI JUN JIE

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2025

COPYRIGHT STATEMENT

This Final Year Project proposal is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project proposal represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project proposal may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Ts Wong Chee Siang who has given me this amazing opportunity to engage in a parallel computing project using CUDA. It is my first step in exploring the field of CUDA programming, equipping me with the proper skills to perform parallel programming.

I must also say thanks to my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

This research will be focusing on developing a Parallel Metaheuristic Algorithm for Route Planning using CUDA to improve the efficiency and performance of route planning. The increasing demand for more efficient route planning approaches, which includes several factors such as cost savings, timely deliveries and reduced carbon emissions, has led to a surge in demand for more advanced route planning algorithms in search of more efficient solutions.

The problem that this research will be tackling is the Travelling Salesman Problem (TSP), which is a specific type of route planning problem where the main objective of it is to find out the optimal set of routes for a given number of vehicles to transport goods to a defined set of destinations. TSPs are known to be NP-hard problems[1] where an increase in the number of vehicles and destinations will significantly increase the computational time required to obtain an optimal solution. Existing works that utilized metaheuristic algorithms have shown their flexibility in solving multiple TSP variants and their capabilities in obtaining near-optimal solutions within a reasonable amount of time. However, due to the limitations of CPUs in terms of parallelization, these algorithms do not perform well as they are highly iterative.

The proposed approach will be utilizing the Compute Unified Device Architecture (CUDA) to enhance the performance and efficiency of metaheuristic algorithms in finding optimal solutions for the TSP by leveraging the parallel processing capabilities of Nvidia Graphics Processing Units (GPUs). This research aims to significantly speed up solution searching for the TSP by using GPUs compared to CPUs. Besides, this research strives to provide a foundation for future research on parallel metaheuristic algorithms, and to further encourage their implementations in real-world instances of route planning.

Area of Study: Massively Parallel Computing, Combinatorial Optimization

Keywords: Parallel Metaheuristic Algorithm, Travelling Salesman Problem, CUDA, GPU, Genetic Algorithm

TABLE OF CONTENTS

TITLE PAGE i

COPYRIGHT STATEMENT ii

ACKNOWLEDGEMENTS iii

ABSTRACT iv

TABLE OF CONTENTS v

LIST OF FIGURES viii

LIST OF TABLES ix

LIST OF ABBREVIATIONS x

CHAPTER 1 Introduction 1

1.1 Problem Statement and Motivation 1

1.2 Research Objectives 2

1.3 Research Scope and Direction 4

1.4 Significance and Contributions 5

1.5 Report Organization 6

CHAPTER 2 Literature Review 7

2.1 An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems [5] 7

2.2 Parallel Implementation of Ant Colony Optimization for Travelling Salesman Problem [6] 8

2.3 CUDA-Based Genetic Algorithm on Traveling Salesman Problem [7] 9

2.4 A Genetic Algorithm for the Split Delivery Vehicle Routing Problem [8] 9

2.5 Genetic Algorithm [9] 10

2.6 GPGPU Processing in CUDA Architecture [10] 11

2.7 Parallelizing a Genetic Operator for GPUs [12] 12

CHAPTER 3 System Model 13

3.1 Equations 13

3.2 System Architecture and Design 14
3.2.1 Sequential Single-threaded CPU Implementation 14
3.2.2 Parallel Multi-threaded CPU Implementation 15
3.2.3 Parallel Multi-threaded GPU Implementation 15
3.3 Nvidia GPU Architecture 16
3.4 Genetic Algorithm Design 19
3.4.1 Chromosome Representation 19
3.4.2 Evolution 20
3.4.3 Elitism 20
3.4.4 Selection 21
3.4.5 Crossover 21
3.4.6 Mutation 21
3.4.7 Fitness Evaluation 22
3.5 Parallel Optimizations for the Genetic Algorithm 22
3.5.1 CPU-side Optimizations 22
3.5.2 GPU-side Optimizations 22
CHAPTER 4 Experiment 25
4.1 Hardware Setup 25
4.2 Software Setup 25
4.3 Settings and Configurations 25
4.3.1 Parameters for Genetic Algorithm 26
4.4 Test Set 26
4.5 Implementation Issues and Challenges 27
4.6 System Operation 28
4.6.1 Single-threaded CPU Implementation 28
4.6.2 Multi-threaded CPU Implementation 28
4.6.3 Block-level Sorting and Elitism – GPU Implementation 29

4.6.4 Sorting with Thrust Library – GPU Implementation 29
4.7 Concluding Remarks 29
CHAPTER 5 System Evaluation and Discussion 30
5.1 System Testing and Performance Metrics 30
5.2 Experimental Results 30
5.2.1 Results of Various Implementations Across Multiple Problems 31
5.2.2 Average Speedup of Parallel Implementations on Each Problem 32
5.2.3 Comparison of Average Solution Quality against Average Speedup 34
5.3 Result Analysis 36
CHAPTER 6 Conclusion **39**
REFERENCES **41**
POSTER **43**

LIST OF FIGURES

Figure Number	Title	Page
Figure 3.1	Nvidia's Ampere SM Architecture	16
Figure 3.2	Memory Hierarchy in Nvidia GPUs	17
Figure 3.3	Nvidia CUDA Thread Architecture	18
Figure 3.4	General Flow of Genetic Algorithm	19
Figure 4.1	Single-threaded CPU Implementation	28
Figure 4.2	Multi-threaded CPU Implementation	28
Figure 4.3	Block-level Sorting and Elitism – GPU Implementation	29
Figure 4.4	Sorting with Thrust Library – GPU Implementation	29
Figure 5.1	Graph of Algorithm Speedup Across Multiple Implementations against Multiple Problem Instances	37

LIST OF TABLES

Table Number	Title	Page
Table 4.1	Specifications of Personal Computer	25
Table 5.1	Results of Single-threaded CPU Implementation	31
Table 5.2	Results of Multi-threaded CPU Implementation	31
Table 5.3	Results of Block-level Sorting and Elitism - GPU Implementation	32
Table 5.4	Results of Sorting with Thrust Library - GPU Implementation	32
Table 5.5	Average Speedup of Parallel Implementations on berlin52	32
Table 5.6	Average Speedup of Parallel Implementations on pr107	33
Table 5.7	Average Speedup of Parallel Implementations on pr152	33
Table 5.8	Average Speedup of Parallel Implementations on pr264	33
Table 5.9	Average Speedup of Parallel Implementations on pr439	34
Table 5.10	Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on berlin52	34
Table 5.11	Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr107	34
Table 5.12	Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr152	35
Table 5.13	Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr264	35
Table 5.14	Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr439	35

LIST OF ABBREVIATIONS

ACO	Ant Colony Optimization
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
MPI	Message Passing Interface
OX	Order Crossover
RAM	Random Access Memory
SDVRP	Split-Delivery Vehicle Routing Problem
SIMD	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
SSD	Solid-State Drive
TBB	Threading Building Blocks
TSP	Travelling Salesman Problem
VRAM	Video Random Access Memory
VRP	Vehicle Routing Problem

CHAPTER 1
Introduction

In this chapter, we will present the background, problem and motivation of our research, our contributions to the field, and the outline of the thesis.

The research will be further introduced in this chapter, where the problem definition, research objectives, scope and directions, and contributions to the fields of computational optimization, parallel metaheuristic algorithms, and logistics will be described in the following parts.

1.1 Problem Statement and Motivation

The Travelling Salesman Problem (TSP) is a well-known problem that falls under the combinatorial optimization category, whereby the objective is to search for the most efficient routes, which is the minimal total distance for a traveler to visit each node exactly once. However, TSPs are infamous NP-hard problems[1], where computational demands such as time and power are substantially increased as the number of vehicles and destinations increase, to find an optimal solution for the given scenario. Due to the complex nature of these problems, they require more advanced and sophisticated optimization techniques that are highly efficient to obtain optimal solutions within a reasonable timeframe. Therefore, it is necessary to tackle these challenges to optimally handle an extensive amount of data and complex constraints that are generated from the problems.

The motivation of this research stems from the vital need to develop route planning solutions that are efficient, well-performing and scalable. These solutions must be able to meet the demands of the world today. With the development of logistics networks each day and the growth of the e-commerce sector [2], route planning will get more tedious and complex, and we cannot simply rely only on humans to plan routes as humans tend to have biases and are not efficient enough to obtain solutions that are near-optimal all the time. Additionally, as businesses, organizations and even in a larger scale, countries, take the initiative to transition towards being carbon-neutral [3], efficient route planning becomes even more crucial because it is a step forward towards

less carbon emission. Not only that, but it is also possible for a significant room for cost, time and fuel savings.

Furthermore, acknowledging the fact that the GPU architecture is designed for parallel processing [4], it has given us much motivation for the implementation of our proposed method. GPUs offer thousands of cores that work in parallel, which provides them with the capability to handle a large number of tasks simultaneously. Furthermore, GPUs also have higher memory bandwidth compared to CPUs, which paves way for quicker data transfer and access. This offers several advantages over CPUs when it comes to parallel computing as GPUs are able to perform parallel computations much quicker compared to CPUs with a limited number of cores. Besides, these strengths also improve their capability in handling complex calculations and high-throughput computations. This allows GPUs to have a high efficiency in performing metaheuristic algorithms as their design is well-suited for the parallelizable components of the algorithms. The existence of CUDA which allows the use of Nvidia GPUs for general-purpose computing provides developers like us with more accessibility in utilizing GPUs effectively and efficiently.

Therefore, we see that leveraging CUDA to utilize GPUs for the implementation of parallel metaheuristic algorithms is another big step forward towards a significant improvement of the efficiency of metaheuristic algorithms on solving the TSP, as there is much potential to improve computational speed and scalability of the algorithms, which can also lead to improvements in solution quality. This research seeks to develop a robust framework that has the capabilities to solve complex route planning problems, ultimately contributing to the improvement of logistics and transportation systems.

1.2 Research Objectives

The primary objective of this research is to demonstrate the capability of parallelized metaheuristic algorithms in significantly increasing the efficiency and performance of these algorithms in solving the TSP. This research aims to obtain significant improvements in efficiency, computation time, scalability and solution quality for the TSP by leveraging CUDA. The specific objectives of the project will be further discussed below.

Throughout this research, we aim to design and implement a massively parallel GA that is highly efficient. The design and implementation of the proposed algorithm will be focused on parallelizing the key components of GA to maximize GPU utilization and performance. By this, we strive not to waste any available resources and to reduce time efficiencies. We will be looking deeper into the components of the GA and the architecture of the GPU to ensure maximum and efficient parallelization of the GA components wherever possible. This is also important to reduce overhead caused by communication and synchronization of information between the GPU and the CPU, and also between threads.

This research also aims to achieve a significant speedup in solving TSP instances using the parallelized GA strategies, as we look forward to reducing the overall computation time compared to CPU-based implementations. As multiple components of the GA are highly parallelizable, it becomes less efficient to run the GA sequentially as the number of loops will significantly increase as the TSP instance gets larger due to its iterative nature.

Furthermore, we would also like to ensure substantial improvements in terms of the scalability of the developed algorithm to improve its capability in handling large-scale TSP instances. As demands in real-life scenarios continue to increase, it is crucial for the algorithm to have the ability to perform optimization for large instances of the TSP, as it better reflects the scale of demands in real-life applications. The improvements in terms of scalability will help us to demonstrate the practical applicability of the algorithm in real-world logistics and transportation systems.

Another objective of this research is to present a detailed comparison between single-threaded CPU, multi-threaded CPU and GPU-based implementations of the GA to highlight the effectiveness of our proposed algorithm that is achieved through GPU acceleration. Comprehensive benchmarking and performance analysis will be conducted to assess the different implementations in terms of efficiency, scalability and solution quality to properly evaluate the algorithm’s effectiveness.

One more objective of this research is to be able to contribute towards a massively parallel implementation of metaheuristic algorithm on GPUs to solve the TSP. This field of research is very much underexplored; therefore, we look forward to

providing more examples and demonstrations of this implementation to further encourage research in this field and the application of it in real-life scenarios.

1.3 Research Scope and Direction

The aim of this research is to develop and evaluate a parallelized metaheuristic algorithm to solve the TSP by using CUDA. The scope of this project encompasses a few key areas which are stated below.

This research will encompass algorithm design and development. We will be implementing GAs with multiple approaches, including a sequential and parallel GA on the CPU, and two parallel GAs on the GPU. The main focus of this research will be on the parallel algorithm, as we would like to demonstrate the effectiveness of parallelism in enhancing the performance of GAs in solving the TSP. For the proposed algorithms, parallel strategies will be designed to leverage CUDA to accelerate the GA using GPU, which include components of the GA such as fitness evaluation, elitism, crossover and mutation operations. Not only that, as every generation of the GA has multiple instances, these instances will also be parallelized instead of being processed sequentially. Then, the parallel GAs will be developed using CUDA C++ and kernel functions will be optimized to maximize GPU utilization and performance. Furthermore, we will also be optimizing the parallelization strategy to reduce communication and information synchronization overhead between parallel threads, and also between CPU and GPU memory.

Furthermore, our research will also be benchmarking the performance of the proposed algorithms against CPU implementations of GA. Multiple performance metrics such as computation time, speedup, scalability and solution quality will be defined to assess the effectiveness, efficiency and performance of our proposed algorithm. A comparative analysis will be conducted between the proposed algorithms and the CPU-based implementations to identify the improvements in terms of efficiency, performance and scalability.

1.4 Significance and Contributions

This research titled “Parallel Metaheuristic Algorithm for Route Planning using CUDA” aims to make several substantial contributions to the field of computational optimization, parallel metaheuristic algorithms and logistics. Our experiments and analysis have confirmed the feasibility of our proposed method in improving the efficiency and performance of route planning. The primary significance and contributions of this study will be explained below.

We hope that our research could help and impact the development of parallel metaheuristic frameworks for solving route planning problems so that this field of research could be utilized optimally in the future. In this research, we will design and implement GAs that run in parallel on a GPU to solve the TSP by using CUDA. This framework is expected to significantly reduce computation time while also improving the scalability of metaheuristic algorithms. We aim to optimize the process of the metaheuristic algorithms running on the GPU to reduce inefficiencies such as overhead due to communication and information synchronization between parallel threads, and between CPU and GPU memory. With proper optimization, the program will be able to run efficiently, which will ensure its performance, showing a greater potential in real-world applications.

This research will also bring forward a thorough benchmarking and performance analysis in order to compare the proposed implementation of parallel metaheuristic algorithms using GPU against traditional CPU-based implementations. This research will be providing insights into the improvements in computing time, scalability and solution quality accomplished through GPU acceleration. From the benchmarks, we also aim to assess the practical applicability of the proposed implementation to bridge the gap between theoretical research and real-world implementation in logistics and transportation systems.

Last but not least, this research aims to develop a foundation for future research on the field of computational optimization, parallel metaheuristic algorithms and logistics. Our research will address the computational challenges and difficulties related to TSPs and demonstrate the effectiveness of leveraging CUDA to enhance metaheuristic algorithms. We aim for this research to lay the groundwork for future studies on this topic. We believe that our research will open new avenues for exploring

parallel computing techniques that utilize CUDA in optimization problems other than route planning.

We expect to advance the latest developments and improvements in route planning by proposing our research on this topic to further contribute to the related fields. The outcomes of this research have immense potential in developing and transforming logistics and transportation systems, which could provide various benefits that are practical in terms of cost and time efficiency and environmental sustainability.

1.5 Report Organization

This report is organized into 6 chapters: Chapter 1 Introduction, Chapter 2 Literature Review, Chapter 3 System Model, Chapter 4 Experiment, Chapter 5 System Evaluation and Discussion, and Chapter 6 Conclusion. The first chapter is the introduction of this research which includes the problem statement, objectives, scope, direction, significance and contributions, and also report organization. The second chapter covers the literature review carried out on past works on solving routing problems to evaluate the strengths and weaknesses of each approach, and also regarding genetic algorithm and the CUDA architecture. In the third chapter, we will discuss about the system models in this research. The fourth chapter will cover the experiment setup. The fifth chapter reports the results obtained from the experiment, and discussions about the results obtained. The sixth chapter will be our conclusion and recommendations for further research.

CHAPTER 2
Literature Review

2.1 An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems [5]

This paper presents a parallelization method for GAs to accelerate the algorithm in finding solutions for the VRP [5]. The authors experimented the parallel GA on the Travelling Salesman Problem (TSP) to showcase how effective the proposed method is in solving equivalent problems such as the TSP. The authors have done the experiments on multi-core and many-core systems [5]. Furthermore, the authors also proposed to integrate the system with vehicular cloud computing, where participating vehicles communicate with the cloud service to provide vehicle position and obtain routing information and optimum routes for the vehicles.

The authors utilized CUDA to parallelize the GA on a GPU. They organized three different kernels to parallelize three different components of GA. The three kernels will calculate the primary population's fitness, perform crossover, mutation and fitness functions, and the selection process, respectively. Since the computation results are stored in the GPU's global memory, there is no data switching between the host and the device while the kernel is switched. Therefore, there is very little overhead in terms of kernel-switching time.

The proposed method is tested with 3 sets of data that is derived from VLSI data for TSP implementation, each having 380, 737 and 984 cities, and is compared against sequential CPU implementation and Threading Building Blocks (TBB) multi-core CPU implementations. It is found that the CUDA implementation yielded the best results when the population size is large due to CUDA enabling many threads to be defined but falls short to the TBB implementations in smaller populations. Therefore, this showed that the CUDA implementation can significantly speedup GA operations for medium to large instances of the TSP, provided that a maximum number of threads and a large population size is defined.

The effectiveness of the implementation and its improvements of the parallel GA does not necessarily reflect the exact scale of improvements when it comes to the VRP, as the TSP and VRP have their differences. However, since the paper discussed

the effectiveness of a parallel GA on the TSP using CUDA, it serves as another confirmation towards our direction of research.

2.2 Parallel Implementation of Ant Colony Optimization for Travelling Salesman Problem [6]

This paper presents a parallel ant colony optimization (ACO) algorithm for the TSP. The authors leveraged a Message Passing Interface (MPI) framework to parallelize the algorithm.

The authors noted multiple characteristics of sequential algorithms that can be effectively parallelized with MPI, including the ability of the algorithms to run in cycles, and that the work should be able to be divided between several processing units. Furthermore, the architecture of the machine is also crucial as it determines how efficient the parallel implementation can be.

The way this MPI works follows a master-slave architecture, where the master obtains the problem instance and broadcasts it to all slaves, and then passes control to the slaves for them to work on the algorithm. Then the master waits for them to update the data. Every individual slave will be working on an independent instance of the sequential algorithm.

To implement the MPI framework with the ACO, the authors decided that it is necessary for each slave or ant to have a local copy of pheromone matrix that is synchronised with the master's global pheromone matrix after every ant cycle completes.

The parallel implementation is successful that the authors were able to achieve a linear speedup for up to 26 processors, and there is degrading beyond this number of processes which the authors believe that it is due to communication overhead and idle time of the processors.

This work shows that there is significant room for metaheuristic algorithms to be parallelized, as many metaheuristic algorithms work for many cycles, and at most times, each instance is independent from the other instances, where most of the communications occur only after each iteration is completed.

2.3 CUDA-Based Genetic Algorithm on Traveling Salesman Problem [7]

In this piece of work, the authors designed a genetic algorithm that can run on CUDA to leverage the many processing units offered by GPUs. The authors parallelized the crossover and mutation processes as each thread's process does not interfere with other threads. However, they noted that the population cannot be directly replaced by the new population as it will overwrite the information of the old population that were being used to generate the new population. Therefore, the authors have to make another temporary copy of the target chromosome to work on.

The authors also noted other challenges of a parallel implementation of GA like this. At the time of this implementation, CUDA does not provide random number generation methods yet, therefore the authors have to simulate a pseudo random number generator for the use of GA. They generated random seeds from the CPU for every thread of the GPU so that each thread can have a local random number generator function that does not interfere with each other.

Furthermore, due to the limits of shared memory, the authors also did not use a distance table but instead used coordinate arrays as they are small enough to fit in the shared memory of the GPU.

The authors ran the experiment on only one streaming multiprocessor (SM) by generating only one block, which they noted that they only used about 1/16 of the computing resource on their GPU as their GPU has 16 SMs. From this experiment, they are still able to obtain around 50% of speed up compared to CPU.

As we can see that with only one SM active, the GPU can already beat CPU in terms of speed, this poses a big opportunity to further speed up GAs when the whole GPU is being utilized.

2.4 A Genetic Algorithm for the Split Delivery Vehicle Routing Problem [8]

The authors of this paper propose two hybrid genetic algorithms to solve the split delivery vehicle routing problem (SDVRP) [8], where in this variant of VRP, customers are able to be visited more than once. The authors propose the hybrid genetic algorithms to allow genetic global search procedure and guarantees feasibility because a pure genetic algorithm implementation is unable to guarantee feasible solutions.

The two hybrid GAs each utilize a different fitness approach. The first fitness approach is based on the shortest route. The feasible routes are sorted based on travel distance, from the shortest to the longest, and each route has a fixed probability of being added to the current solution. The shorter routes that meet a certain capacity threshold has a higher chance to be selected compared to longer routes. If no other routes are feasible, the solution is completed with a construction heuristic to ensure feasibility. On the other hand, the second approach is quite similar to the first approach but is based on the ratio of demand unit versus distance unit. The larger the ratio, the route will be sorted in front, therefore having a larger probability of being selected.

From the results, it is shown that both hybrid GAs have significantly reduced computation times compared to Chen et al's two-phase method. and Jin et al's column generation method in most instances, but the solution quality does not always match the two compared methods. The authors found that neither approach is faster than the other, but the second approach yields better improvement in most cases.

This paper has shown the effectiveness of implementing a hybrid GA to solve the SDVRP as it has achieved significant speedup in obtaining near-optimal solutions, and in larger instances of the problem, the algorithm is also able to find better solutions. Therefore, this paper showcased that GA is versatile enough to handle route optimization problems other than the TSP and is able to combine with other heuristics to form hybrid metaheuristics that could potentially perform better.

2.5 Genetic Algorithm [9]

In this paper, the author goes in depth about the genetic algorithm (GA). The author mentions that GAs have been applied to solve a broad range of problems. GAs follow the concept of natural selection as the algorithm evolves into more successful chromosomes. The fitness function determines the fitness of each chromosome, which guides the search process by determining the quality of solutions. Chromosomes with better fitness will have a higher chance of being selected for reproduction.

The three main genetic operators are reproduction, crossover and mutation, where these genetic operators help to create new solution vectors from current solution vectors through selection, combination or alteration. This helps in exploitation and exploration of the search space.

Compared with traditional optimization methods, GAs are quite different. The nature of GA which uses a string-coding of variables can effectively discretize the search space. Furthermore, due to the fact that GAs work with multiple solutions, good information from previous generations can be passed down to later generations, which helps the algorithm to reach optimal solutions.

Therefore, this paper tells us that GAs are versatile as they can be used to optimize various types of problems. It is even more powerful when the problem's search space is large and complex, which route optimization problems such as the TSP also falls into this category.

2.6 GPGPU Processing in CUDA Architecture [10]

This article highlights the strengths of GPUs in data parallelism when compared against CPUs [10] and describes in detail about CUDA, the parallel computing architecture developed by Nvidia.

It is highlighted that CPUs and GPUs both excel in different kinds of tasks. CPUs are optimised for better performance on sequential tasks, while GPUs are optimised for parallelizing a greater number of arithmetic operations, especially floating point operations. This is due to the different architectures between the CPU and the GPU.

CPUs typically have high performance on a single thread, and have immensely high-speed caches, which are good for data reuse [10]. This allows CPUs to handle multiple different processes or threads at a single time, making them more efficient in tackling operating system tasks including memory management and job scheduling.

GPUs consist of a large number of math units [10], which gave GPUs the ability to have high throughput when it comes to performing operations in parallel. The GPU architecture allows low access times to onboard memory, and programs were run independently on each compute unit within the GPU. Therefore, the GPU has been used to parallelize complex mathematical operations, and this use of GPU for non-graphical workloads is known as General Purpose GPU (GPGPU). This can be done by using CUDA, which is Nvidia's GPU architecture for general purpose computing using GPUs.

The CUDA architecture splits the GPU into grids, blocks and threads hierarchically [10]. Grids are made up of blocks, and blocks are made up of threads.

Not only that, but there are also different memory types for different purposes and with different memory access times, such as global, shared, local, texture and constant memory.

The author highlighted that application development for CUDA can be done easily by using programming languages such as C, C++, Java, Python and more [10]. This allows developers to harness the computing capabilities of the GPU. The compiled code can be run directly on the device. There are also fully GPU accelerated libraries that are currently available for developers to use, and one can easily analyze the GPU's performance with CUDA's visual profiler [11]. Therefore, CUDA programming is not complex to perform. However, there are also limitations on CUDA, such as the latency of communication between the CPU and GPU, and CUDA support is only limited for Nvidia GPUs.

2.7 Parallelizing a Genetic Operator for GPUs [12]

The authors proposed a method to parallelize a genetic operator, which is the order crossover (OX) operator on the GPU. In the proposed method, the authors want to parallelize the process of OX operator by allowing one whole thread block to process one individual. This is done by randomly selecting a segment from one parent and copying the elements to the child in parallel. Then, the generalized prefix-sums were computed in parallel. This result is converted to the destination indices in parallel, and according to the indices, segments from the other parent are copied into the child in parallel. In this model, only one child is generated from two parents.

The proposed method is able to achieve significant speedup compared to sequential CPU implementation when implemented in GA. Initially, when the process is directly ported onto the GPU, it fails to run faster in certain cases. However, the parallelized OX operator turned things around, making the GA faster on GPU than the CPU by more than an order of magnitude. Therefore, this approach has proven to be effective on CUDA architecture. This shows that there is room of optimization for genetic operators within the GA itself.

CHAPTER 3
System Model

3.1 Equations

The TSP seeks to determine the most efficient route for a single traveller visiting a set of cities before returning to the starting point. The TSP is focused on a single tour that covers all cities with minimal costs (distance).

The related equations are as follows:

Let:

N be the number of cities (or locations to be visited),

X be the set of N cities, where $X = \{ x_i \mid i = 1, \dots, N \}$

l_i be the location of city x_i,

d_{ij} be the cost (distance) of going from city x_i to x_j, and,

x_{ij} be a binary decision variable, where its value is 1 if travel occurs from city x_i to x_j, and 0 otherwise.

The objective of solving the TSP is to minimize the total cost, which in this case, the total travelling distance, while ensuring that every city is visited once and only once before returning to the origin city, which is the starting city. This objective can be denoted by the equation below:

$$min \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} d_{ij} x_{ij}$$

There are a few constraints to solving the TSP, which are shown below:

(a) Each city must be visited exactly once:

$$\sum_{j=2}^{N} x_{1j} = 1$$

This ensures that each city is visited once and only once, preventing duplicate visits.

(b) The route starts with the first city:

$$\sum_{j=2}^{N} x_{1j} = 1$$

This ensures that travel initiates from the first city.

(c) The route must return to the first city:

$$\sum_{i=2}^{N} x_{i1}^{k} = 1$$

This ensures that the final city visited will be linked back to the starting city.

(d) If a city is entered, it must be exited:

$$\sum_{i=1}^{N} x_{ij} = \sum_{j=1}^{N} x_{ji} = 1, \forall\, i = 1, \ldots, N$$

This ensures that the tour is a valid closed loop.

We will be working on the Symmetric TSP, so $d_{ij} = d_{ji}$.

Therefore, the process of solution searching for the TSP must minimize the total distance travelled while adhering to the constraints listed above.

3.2 System Architecture and Design

The research will consider three distinct approaches for implementing the GA:

(a) A sequential, single-threaded implementation on CPU,
(b) A parallel, multi-threaded implementation on CPU, and
(c) A massively parallel, many-core implementation on the GPU.

This allows us to benchmark and compare the implementations in terms of scalability and overall computational performance.

3.2.1 Sequential Single-threaded CPU Implementation

The sequential, single-threaded CPU implementation of the GA will be used to reflect the operations of GA without parallelization, as it is the most fundamental and

straightforward approach. In the sequential implementation, all operations within a single iteration are repeated on every single chromosome in the population. This forms two nested loops that are repeated, which is highly inefficient and redundant, especially for large population sizes. Due to sequential execution, the runtime is heavily constrained by the processor's clock speed and instruction execution efficiency.

This sequential approach serves as a baseline for comparison and provides a clear understanding of how GA functions at its fundamental level before leveraging parallel computing techniques.

3.2.2 Parallel Multi-threaded CPU Implementation

The parallel, multi-threaded implementation on CPU will show the utilization of hardware based on today's specifications, as CPUs these days are designed with multi-core architectures, it is proper to demonstrate the GA implementation which fully utilizes CPUs available today to better reflect the performance of GAs on the CPU.

3.2.3 Parallel Multi-threaded GPU Implementation

On the other hand, the GPU implementations of the GA will be massively parallel implementations across thousands of cores within the GPU, where parallelization happens on a large scale, and it is to demonstrate the effectiveness and efficiency of an implementation like this on today's GPUs. Furthermore, GPUs are optimized for high-throughput parallel computation, making them ideal for handling large-scale optimization problems.

In the parallel implementations, multiple threads perform repeated operations on different chromosomes within the population, as the tasks are being spread out to multiple different processing units. This approach ensures better utilisation of available computational resources. Therefore, this reduces the number of loops to go through each chromosome, which theoretically will significantly reduce the amount of computation time.

We will be implementing two different approaches for the GPU implementation. These two approaches differ in the sorting and elitism approach. This will be discussed later in section 3.5.2 (c).

3.3 Nvidia GPU Architecture

To fully leverage the massive parallelism offered by GPUs, components of the GA were restructured for more efficient execution on the GPUs. It is important to understand the GPU architecture to properly optimize algorithms for the GPU.

An Nvidia GPU is typically composed of multiple streaming multiprocessors (SM). The following is a diagram from [13] that describes Nvidia's Ampere SM architecture, which is the architecture our GPU, the RTX 3080 is built on:

Figure 3.1: Nvidia's Ampere SM Architecture

As we see from the diagram above, each SM consists of multiple CUDA cores. These cores execute Single Instruction, Multiple Data (SIMD) operations, meaning multiple cores process the same instruction simultaneously. Within each SM, there are 4 warp schedulers, which manages 32 threads each. All 32 threads managed by the warp scheduler will process the same instruction. For every block spawned, it will be distributed to one SM, where the threads in the block will be scheduled by the warps.

Nvidia GPUs also have a memory hierarchy as described in [14] which is shown in the figure below:

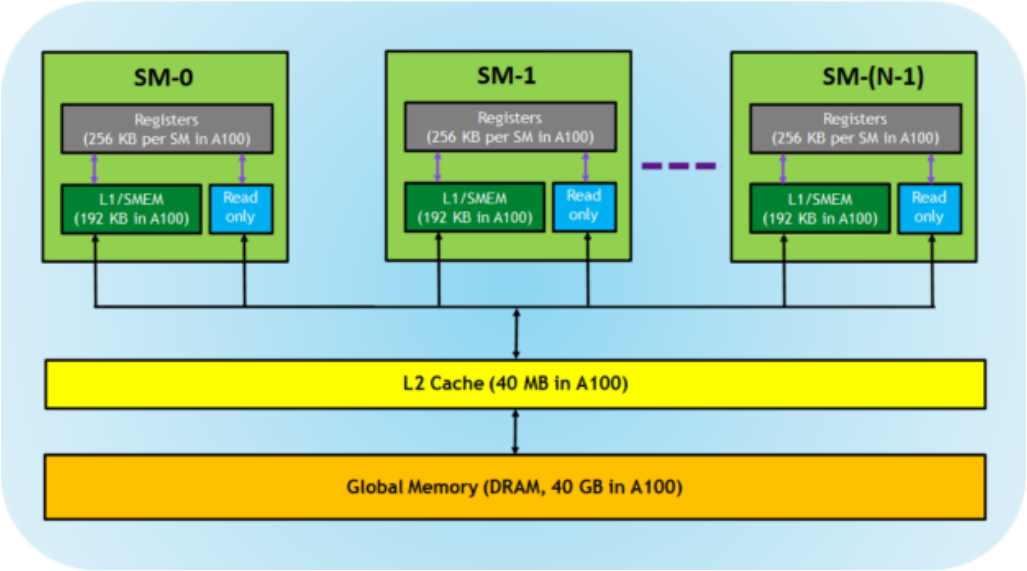


Figure 3.2: Memory Hierarchy in Nvidia GPUs

The following is a breakdown of the memory hierarchy shown above:

(a) Registers

The fastest memory available to a GPU thread with instantaneous access, where they store temporary variables used during computation.

(b) L1 Cache/Shared Memory (SMEM)

This is a fast, small on-chip memory located inside a SM, where it is shared within the same block with all threads.

(c) Read-only Memory

Consists of multiple components such as constant memory, texture memory, instruction cache and read-only cache, and these are read-only to kernel code.

(d) L2 Cache

Slower than L1 cache, but faster than global memory. Every SM can access it, which means that all threads across different blocks can access the data in it.

(e) Global Memory

The largest memory available, which is the VRAM of the GPU. Has the slowest memory access speeds due to access requiring going off-chip.

When a GPU kernel is launched, each kernel is executed as an array of threads [15] A grid consists of multiple blocks, and each block is made up of multiple threads that are guaranteed to execute instructions simultaneously. Below is a diagram of the CUDA thread architecture described in [15].

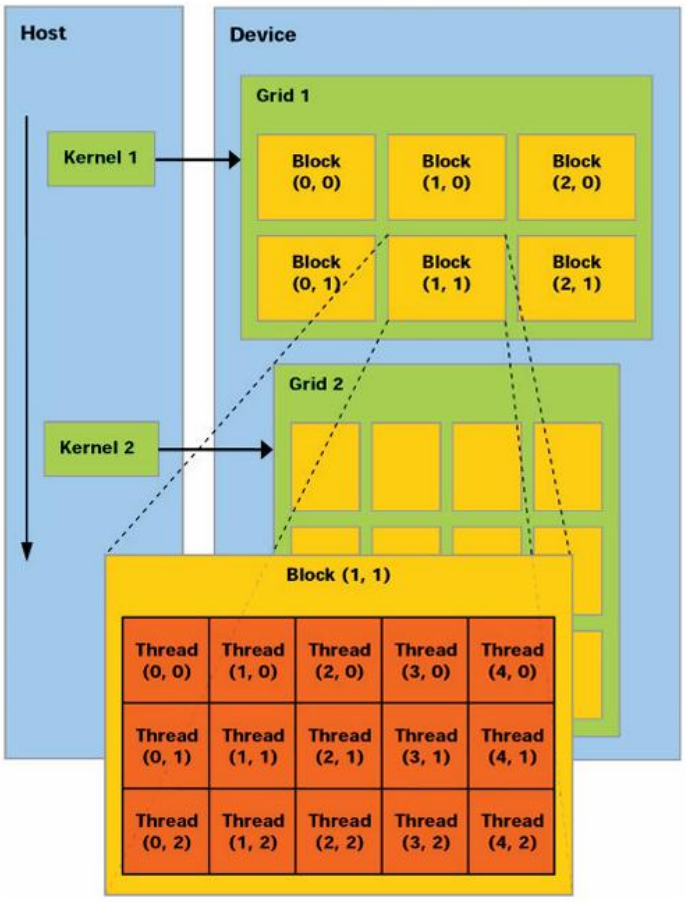


Figure 3.3: Nvidia CUDA Thread Architecture [15]

3.4 Genetic Algorithm Design

The proposed system implements the GA to solve the TSP. The following diagram presents the general flow of our GA, which is similar to [16]:

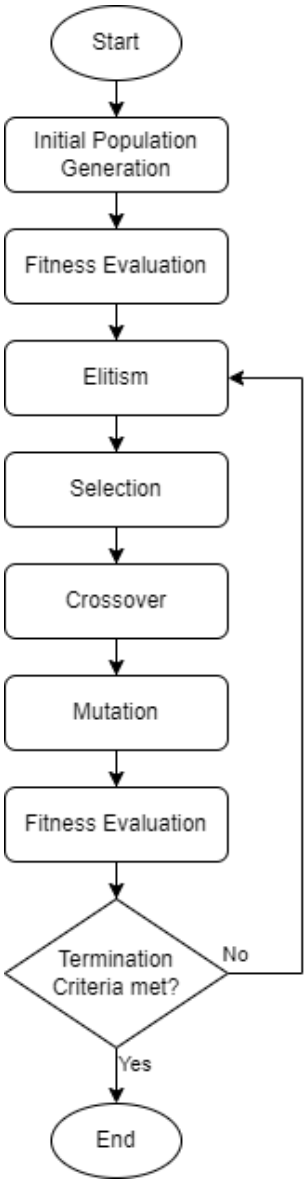


Figure 3.4: General Flow of Genetic Algorithm [16]

The details of the operations across the population are described below.

3.4.1 Chromosome Representation

Each individual in the population, known as a chromosome, represents a possible TSP tour. The chromosome is implemented as a permutation of city indices stored in an array. The order of these indices represents the path taken to visit each city once and only once, and then return to the origin. Additionally, a fitness value is

associated with each chromosome which corresponds to the total tour distance computed during the evaluation phase.

3.4.2 Evolution

Evolution refers to the process of generating the next generation of the population. There will be a predefined number of chromosomes in the population. In our GA, each individual evolution process will generate one new chromosome for the next generation based on genetic operations. For the evolution process, the population will be divided into several groups to perform different roles. The roles are described as follows:

(a) Elitism

In this group, 12.5% of the population with the best fitness will be directly selected to be placed in the new population. This ensures that the best solutions are retained across generations.

(b) Elitism with Swap Mutation

In this group, chromosome from the elite subset is selected, and swap mutation is performed on each chromosome. This makes up another 12.5% of the new population.

(c) Elitism with 2-opt Mutation

In this group, chromosome from the elite subset is selected, and 2-opt mutation is performed on each chromosome. This makes up another 12.5% of the new population.

(d) Order Crossover (OX) with Chance of Swap Mutation or 2-opt Mutation

This section implements the OX which generates one new child. After crossover, a mutation, either swap or 2-opt mutation, is applied to each offspring based on a predefined mutation rate.

3.4.3 Elitism

The whole population will be sorted in ascending order based on their fitness. 12.5% of the population with the best fitness, which has the lowest cost, will be the elite population.

The sorting process will be done globally on the CPU implementations, however on the GPU side, we defined two approaches to the sorting and elitism process,

which is the block-level sorting and elitism, and sorting with Thrust library. These two approaches on the GPU will be further discussed in section 3.4.2 later.

3.4.4 Selection

The selection process is very straightforward, where two parents are randomly selected from the elite population.

In the GPU implementations however, since there are two strategies for sorting and elitism, the locations of the elite population will be different in the global memory, which requires a slightly more sophisticated selection clause. This will be further discussed in section 3.5.2 (c).

3.4.5 Crossover

The crossover method used is Order Crossover (OX). This crossover works as described below:

1. Two parents are randomly selected from the elite population.
2. Random subsequence from one parent is preserved in its original position.
3. The remaining genes are filled from the second parent in order, skipping already used cities to ensure a valid permutation.

This maintains the relative order and precedence of genes, which is beneficial for TSP problems. This enables the exploitation of good sub paths from one parent, and at the same time explores alternative city orders from the other parent.

3.4.6 Mutation

Two types of mutation are employed to diversify the search:

(a) Swap Mutation: Randomly selects 2 positions and swaps the cities.
(b) 2-opt Mutation: Chooses a random segment and reverses it to create a shorter route.

By incorporating two different mutations strategies in the algorithm, we can balance out exploration and exploitation because swap mutation is a simple, random change that makes small changes, while 2-opt mutation is a structured local search, which could potentially greatly optimize existing routes. This not only allows us to discover new areas of the search space, but also fine tune existing good solutions.

3.4.7 Fitness Evaluation

The fitness of each chromosome is evaluated after the initialization of each chromosome and also after each iteration of genetic operations. Starting from the first customer, the distances between each successive city are computed. After reaching the final city the return distance to the starting note is added to complete the tour. Then, the total distance is stored in the fitness field of the chromosome.

3.5 Parallel Optimizations for the Genetic Algorithm

This subsection outlines the parallelization strategies applied to both multi-threaded CPU and GPU implementations of the GA.

3.5.1 CPU-side Optimizations

To accelerate preprocessing and CPU bound tasks, OpenMP is used for lightweight multi-threading on the CPU. Listed below are a few key areas of the GA that is being optimized:

(a) Genetic Operations: Since the whole chain of genetic operations (selection, crossover, mutation) is embarrassingly parallel, the tasks are divided equally to each logical thread of the CPU to be processed in parallel.
(b) Fitness Evaluation: The process of evaluating chromosome fitness is the same for every chromosome. Therefore, this process is also parallelized.

3.5.2 GPU-side Optimizations

Certain components of the GA were restructured for more efficient execution on the GPUs. Multiple strategies were employed to ensure that the GPU could effectively handle the large population and chromosome sizes while minimizing latency and maximizing throughput.

(a) Genetic Operations:

The whole chain of genetic operations is contained within a kernel. In this kernel, each thread will be handling the operations of generating a new child. This kernel will fully populate all cores of the GPU to prevent underutilization.

As we described before that different groups of threads will perform different operations to generate the new population, the processes are grouped in a way

such that all threads within a warp perform the same set of instructions. This is why we set our block size to 256 and the groups to be 1/8 or 12.5% in size, as 12.5% of 256 is 32, which is exactly the number of threads a warp scheduler manages. This prevents warp divergence where threads within a warp perform different instructions, causing certain threads to wait for divergent paths to complete.

(b) Fitness Evaluation:

The process of fitness evaluation is contained within another kernel, where every thread spawned will be calculating the fitness for one chromosome.

Instead of computing the distances between each node to form a distance matrix, we decided to compute the distances every single iteration. This is mainly due to the limited amount of shared memory, which is not enough to store large distance matrices. Therefore, instead of storing the distance matrix, the kernel only stores the coordinates of each city in the shared memory, which could at least help to reduce global memory traffic and latency.

(c) Sorting and Elitism:

We implemented two approaches for sorting and elitism to test out their feasibility and effectiveness:

(i) Per-block sorting and elitism

Sorting a huge population is an extremely time-consuming task, as the time taken to sort scales with the number of items to sort. Therefore, instead of sorting the whole population, this approach only sorts the chromosomes within each block. This process is done by obtaining the fitness values of each chromosome in the block and putting them into a new temporary array, generating another array of indices, and then sort the indices according to how the fitness values are sorted. This is done primarily to minimize global memory accesses, as moving around chromosomes can be very memory intensive due to the size of the chromosome structure. Then, the whole block of chromosomes is rearranged according to the order of the indices.

Therefore, in this approach, the elite population will be located at the first 12.5% of chromosomes for every block of chromosomes, which is then replicated across the whole population. This is to make the selection process slightly more efficient as parents can be randomly selected across the global population without the need for conditional operators to limit the selection scope.

(ii) Sorting using the Thrust library and obtain the best 12.5% of population for elitism

The Thrust library provides a high-level interface for parallel algorithms such as sorting, reducing and copying, and data structures such as host vectors and device vectors. These implementations are highly optimized for the GPU as they can be very tricky to implement and optimized for GPU usage.

By leveraging thrust, we will be using device vectors to store our chromosomes. For sorting, we utilize the `thrust::sort_by_key` function as it is expensive to directly move large chromosomes around, which means that we will be creating a list of indices, and then sort the fitness values obtained together with the indices. After the sorting process is done, the chromosomes will then be arranged according to the arrangement of the indices by using `thrust::gather`. Then, the best 12.5% of the population is chosen for elitism. Therefore, in this approach, the global elites will be placed at the first 12.5% of the global population. Then, during the selection process, the parents will be randomly selected from this group only.

These two approaches have their own benefits and shortcomings:

(i) Per-block sorting and elitism:

This greatly reduces the number of chromosomes to be sorted in one whole operation by perform parallel sorting within each block. This approach will theoretically be faster than global sorting.

However, the quality of the elite population might be compromised, as it is uncertain that the best 12.5% of the chromosomes in each block are among the best 12.5% of chromosomes globally. This could potentially affect the convergence quality of the algorithm.

(ii) Sorting using the Thrust library and obtain the best 12.5% of population for elitism

This approach will fare better in terms of the quality of the elite population as it can ensure that the global best 12.5% of chromosomes are chosen as the elites. However, due to global sorting, this approach will be slower than the block-level sorting approach but can still perform better than custom kernels for sorting due to the amount of optimizations done in the Thrust library.

CHAPTER 4
Experiment

4.1 Hardware Setup

The hardware involved in this research is a personal computer. This computer will have an NVIDIA GPU installed for the implementation of GA for TSP using CUDA. Specifications of the computer is listed below:

Description	Specifications
Processor	Intel ® Core ™ i7-13700K CPU @ 5.40GHz, 16 Cores, 30MB Cache
Operating System	Windows® 11
Graphics	NVIDIA® GeForce RTX 3080 10GB GDDR6X
Memory	64GB 6400MT/s DDR5 RAM
Storage	3x 2TB PCIe NVMe M.2 SSD

Table 4.1: Specifications of Personal Computer

An Nvidia GPU is required for this research as only Nvidia GPUs are CUDA-enabled.

4.2 Software Setup

The following software components were installed:

1. Visual Studio 2022 Enterprise Edition v17.12.3
2. Nvidia Graphics Driver version 566.36 (Required to install and run CUDA Toolkit)
3. CUDA Toolkit 12.6

In Visual Studio, the option for OpenMP support is turned on to allow the use of OpenMP 2.0 language extensions.

4.3 Settings and Configurations

There are four different approaches to the implementation of the GA for the multi-vehicle TSP, which are single-threaded CPU, multi-threaded CPU and two GPU

implementations. We will be implementing all four approaches to study the feasibility and effectiveness of using parallel methods to speed up the GA processes.

4.3.1 Parameters for Genetic Algorithm

For all three implementations, the parameters of the program are set as follows:

- Population size: 34816
- Mutation rate: 0.10
- The algorithm ends after 500 iterations without solution improvement.

For the GPU implementation, there exists block size and grid size.

- Block size (Threads per block): 256
- Grid size: Population size/Block size = 34816/256 = 136

The grid size is set to 136, which is double the number of SMs of the GPU, where the Nvidia GeForce RTX 3080 has 68 SMs. This way, we allow overlapping execution to occur, which can ensure that each SM in the GPU is utilized, and the workload is evenly balanced across each block. Furthermore, this allows the GPU to execute other warps when some are stalled due to memory access or synchronization points, which helps to maximize GPU usage by preventing cores from being idle.

By setting a block size that is a multiple of 32 (size of a warp), it ensures optimal computing efficiency as it prevents under-populated warps which can waste computation [11]. And in this case, we set a block size of 256, where each of the 128 cores in a SM will be handling 2 threads.

Therefore, with a block size of 256 and a grid size of 136, we get a total number of 34816 threads, which resembles the population size, meaning that each chromosome has its own dedicated processing thread on the GPU.

4.4 Test Set

We obtained our problem instances from TSPLIB, which is a library of sample problems for the TSP from multiple sources and of various types [17]. We will be selecting a few problems from the whole set of symmetric TSP problems to use in our experiments.

4.5 Implementation Issues and Challenges

There are multiple issues and challenges that can arise when implementing the different approaches of the GA.

The crossover method of the GA, which is OX, is quite a sequential task due to the way it requires checking used values within the new child. Therefore, this might limit the amount of optimization we can perform on the GA as the GPU prefers parallel tasks compared to sequential ones.

Furthermore, optimizing memory access on the GPU is also a challenging task as the GPU memory hierarchy consists of multiple memory layers with different memory access times and memory sizes. This can increase the complexity of optimizing memory accesses, as we have to wisely manage memory accesses on the GPU.

An issue on GPU memory access is that each SM of the GPU has a limited amount of shared memory. Shared memory is crucial for efficient memory access as the GPU cores can communicate directly with the shared memory with very low latency, compared to accessing the global memory which typically requires hundreds of cycles per access. Due to our scale of operations in terms of population size and city count, our algorithm is unable to fully harness the benefits of shared memory, which further limits the amount of optimization we can perform.

Besides, synchronization might also lead to overheads, which can reduce the efficiency of the algorithms. These issues have to be taken care of to produce a more efficient algorithm.

One issue we face is the instability of Nvidia's graphics driver. At the time of writing, Nvidia has updated their graphics driver to version 576.28, and CUDA Toolkit also received its update to version 12.9. However, we noticed instability of the graphics driver beyond version 566.36, which caused random stuttering and crashing during daily use, although not frequent, but it impacted our process of developing the program for a few times. Therefore, we decided to roll back our driver to version 566.36, and to maintain the CUDA Toolkit version at 12.6 due to the later versions being not supported by the driver.

Another issue we face hardware related, which is having malfunctioned cores in our CPU. Intel’s microcode bug which caused the cores to draw too much power and

kill themselves has unfortunately handicapped certain cores of our CPU, resulting it to only have 5 performance, hyper-threading enabled cores and 8 efficient cores remaining, totalling to 18 logical processors.

4.6 System Operation

This section presents the sample output window for each implementation.

4.6.1 Single-threaded CPU Implementation

```
C:\Users\danlo\Downloads\CudaRuntime1\exe files>52cpu.exe
No improvements for the past 500 generations. Algorithm execution ended.

Node Count: 52
Population Size: 34816
Best overall solution: Generation 69
Solution: 1 22 31 18 3 17 21 42 7 2 30 23 20 50 29 16 46 48 24 6 4 25 12 28 27 2
6 47 13 14 52 11 51 33 43 10 9 8 41 19 45 32 49 36 35 34 39 40 38 15 5 37 44 0
Best overall distance: 7808
Time until best overall solution: 0.641135 seconds
Time elapsed: 3.94414 seconds


Algorithm execution ended, press any key to exit...
C:\Users\danlo\Downloads\CudaRuntime1\exe files>
```

Figure 4.1: Single-threaded CPU Implementation

4.6.2 Multi-threaded CPU Implementation

```
C:\Users\danlo\Downloads\CudaRuntime1\exe files>52mcpu.exe
No improvements for the past 500 generations. Algorithm execution ended.

Node Count: 52
Population Size: 34816
Best overall solution: Generation 65
Solution: 29 50 20 23 30 2 7 42 21 17 3 18 31 22 1 49 32 45 19 41 8 9 10 43 33 5
1 11 52 14 13 47 26 27 28 12 25 4 6 15 5 24 48 38 37 40 39 36 35 34 44 46 16 0
Best overall distance: 7542
Time until best overall solution: 0.268941 seconds
Time elapsed: 1.19708 seconds

C:\Users\danlo\Downloads\CudaRuntime1\exe files>
```

Figure 4.2: Multi-threaded CPU Implementation

4.6.3 Block-level Sorting and Elitism – GPU Implementation

```
C:\Users\danlo\Downloads\CudaRuntime1\exe files>52block.exe
No improvements for the past 500 generations. Algorithm execution ended.

Node Count: 52
Population Size: 34816
Best overall solution: Generation 165
Best overall solution: 28 27 26 47 13 14 52 11 51 12 25 4 6 15 5 24 48 38 40 37
34 35 36 39 43 33 10 9 8 41 19 45 32 49 1 22 31 18 3 17 21 42 7 2 30 23 20 50 29
 16 44 46
Best overall distance: 7940
Time until best overall solution: 0.034183 seconds
Time elapsed: 0.13106 seconds

C:\Users\danlo\Downloads\CudaRuntime1\exe files>
```

Figure 4.3: Block-level Sorting and Elitism – GPU Implementation

4.6.4 Sorting with Thrust Library – GPU Implementation

```
C:\Users\danlo\Downloads\CudaRuntime1\exe files>52thrust.exe
No improvements for the past 500 generations. Algorithm execution ended.

Node Count: 52
Population Size: 34816
Best overall solution: Generation 62
Best overall solution: 46 44 34 35 49 36 39 40 37 38 48 24 5 15 6 4 25 12 28 27
26 47 13 14 52 11 51 33 43 10 9 8 41 19 45 32 1 22 31 18 3 17 21 42 7 2 30 23 20
 50 29 16
Best overall distance: 7670
Time until best overall solution: 0.058201 seconds
Time elapsed: 0.556134 seconds

C:\Users\danlo\Downloads\CudaRuntime1\exe files>
```

Figure 4.4: Sorting with Thrust Library – GPU Implementation

4.7 Concluding Remarks

Despite all the challenges and issues mentioned above, the project still presents substantial opportunity for success in optimizing parallel GAs to solve the TSP. These challenges are significant in driving forward the development of more efficient and optimized parallel algorithms to harness the power of CUDA in accelerating parallel tasks.

CHAPTER 5
System Evaluation and Discussion

5.1 System Testing and Performance Metrics

There are multiple metrics that we will be measuring to benchmark and compare the different implementations. These metrics are shown as follows:

(a) Execution Time:

The time taken to reach the best route generated.

(b) Iteration Count:

The number of iterations the algorithm takes to reach the best route generated.

(c) Speedup:

The increase in convergence speed compared to sequential CPU implementation. This can be defined in two ways:

- Iteration speedup: Measures the reduction in computation time per iteration
- Effective speedup: Measures the reduction in total execution time to reach convergence

(d) Solution Quality:

How close the generated solution is to the known optimal solution

5.2 Experimental Results

The results calculated in this section adhere to the rounding function and computations described in the documentation provided by TSPLIB as follows [17]:

Rounding function: `nint(x) = (int)(x + 0.5)`

The distance between two points `i` and `j` is computed as follows:

Let `x[i]`, `y[i]`, and `z[i]` be the coordinates of node `i`.

```
xd = x[i]- x[j]

yd = y[i]- y[j]

dij = nint(sqrt(xd*xd + yd*yd))
```

For each of the implementations, their abbreviations are as follows:

(a) Single-threaded CPU Implementation (ST-CPU)
(b) Multi-threaded CPU Implementation (MT-CPU)
(c) Block-level Sorting and Elitism - GPU Implementation (BLOCK-GPU)
(d) Sorting with Thrust Library - GPU Implementation (THRUST-GPU)

5.2.1 Results of Various Implementations Across Multiple Problems

This section shows the results of each implementation across five chosen problems, which are berlin52, pr107, pr152, pr264 and pr439. The numbers in the problem name denotes how many cities are there in the problem. The optimal cost of each problem is obtained from [17].

For each problem instance and implementation pair, we ran 51 tests to obtain the average values. This is due to the nature of GA where it tries to find near-optimal solutions, and we might get a slightly different solution every iteration.

(a) Single-threaded CPU Implementation (ST-CPU)

Problem	Optimal Cost	Best Cost		Average Results	
		Cost	Time (s)	Cost	Time(s)
berlin52	7542	7542	0.4548	7775.13	0.5749
pr107	44303	45030	1.6814	48733.12	2.3140
pr152	73682	78576	4.8815	87588.31	5.6841
pr264	49135	65081	23.9938	73724.12	25.4448
pr439	107217	153848	126.1347	175716.4	123.2664

Table 5.1: Results of Single-threaded CPU Implementation

(b) Multi-threaded CPU Implementation (MT-CPU)

Problem	Optimal Cost	Best Cost		Average Results	
		Cost	Time (s)	Cost	Time(s)
berlin52	7542	7542	0.1989	7666.39	0.2534
pr107	44303	44347	0.8164	45614.41	0.9239
pr152	73682	75359	4.0483	77760.86	2.7716
pr264	49135	54895	9.7706	57972.51	9.4200
pr439	107217	122686	30.7701	134694.5	27.5964

Table 5.2: Results of Multi-threaded CPU Implementation

(c) Block-level Sorting and Elitism - GPU Implementation (BLOCK-GPU)

Problem	Optimal Cost	Best Cost		Average Results	
		Cost	Time (s)	Cost	Time(s)
berlin52	7542	7542	0.0289	7841.92	0.0330
pr107	44303	44581	0.1381	46093.88	0.1915
pr152	73682	73898	0.6147	76360.63	0.5196
pr264	49135	51544	2.1575	54174.02	2.1163
pr439	107217	112928	9.4270	117527.9	9.2499

Table 5.3: Results of Block-level Sorting and Elitism - GPU Implementation

(d) Sorting with Thrust Library - GPU Implementation (THRUST-GPU)

Problem	Optimal Cost	Best Cost		Average Results	
		Cost	Time (s)	Cost	Time(s)
berlin52	7542	7542	0.0519	7704.22	0.0722
pr107	44303	44474	0.1499	45558.9	0.2120
pr152	73682	74508	0.3820	76080.86	0.4473
pr264	49135	50813	2.4515	53738.24	1.6995
pr439	107217	112463	5.8458	117756.1	5.7646

Table 5.4: Results of Sorting with Thrust Library - GPU Implementation

5.2.2 Average Speedup of Parallel Implementations on Each Problem

This section shows the speedup of each parallel implementation on each problem computed against the sequential implementation on CPU.

(a) Problem: berlin52

Implementation	Average Iteration Count	Average Time(s)	Average Iteration Speed (/s)	Average Iteration Speedup	Average Effective Speedup
ST-CPU	60.41	0.5749	105.08	1.00x	1.00x
MT-CPU	61.53	0.2534	242.82	2.31x	2.27x
BLOCK-GPU	157.27	0.0330	4765.75	45.35x	17.42x
THRUST-GPU	73.98	0.0722	1024.65	9.75x	7.96x

Table 5.5: Average Speedup of Parallel Implementations on berlin52

(b) Problem: pr107

Implementation	Average Iteration Count	Average Time(s)	Average Iteration Speed (/s)	Average Iteration Speedup	Average Effective Speedup
ST-CPU	131.6863	2.3140	56.91	1.00x	1.00x
MT-CPU	129.9608	0.9239	140.67	2.47x	2.50x
BLOCK-GPU	338.0392	0.1915	1765.22	31.02x	12.08x
THRUST-GPU	153.4706	0.2120	723.92	12.72x	10.92x

Table 5.6: Average Speedup of Parallel Implementations on pr107

(c) Problem: pr152

Implementation	Average Iteration Count	Average Time(s)	Average Iteration Speed (/s)	Average Iteration Speedup	Average Effective Speedup
ST-CPU	231.12	5.6841	40.66	1.00x	1.00x
MT-CPU	281.37	2.7716	101.52	2.50x	2.05x
BLOCK-GPU	568.17	0.5196	1093.48	26.89x	10.94x
THRUST-GPU	235.16	0.4473	525.73	12.93x	12.71x

Table 5.7: Average Speedup of Parallel Implementations on pr152

(d) Problem: pr264

Implementation	Average Iteration Count	Average Time(s)	Average Iteration Speed (/s)	Average Iteration Speedup	Average Effective Speedup
ST-CPU	625.27	25.4448	24.57	1.00x	1.00x
MT-CPU	548.92	9.4200	58.27	2.37x	2.70x
BLOCK-GPU	1161.51	2.1163	548.84	22.33x	12.02x
THRUST-GPU	601.78	1.6995	354.09	14.41x	14.97x

Table 5.8: Average Speedup of Parallel Implementations on pr264

(e) Problem: pr439

Implementation	Average Iteration Count	Average Time(s)	Average Iteration Speed (/s)	Average Iteration Speedup	Average Effective Speedup
ST-CPU	2004.69	123.2664	16.26	1.00x	1.00x
MT-CPU	1110.843	27.5964	40.25	2.48x	4.47x
BLOCK-GPU	2503.549	9.2499	270.66	16.64x	13.33x
THRUST-GPU	1236.02	5.7646	214.42	13.18x	21.38x

Table 5.9: Average Speedup of Parallel Implementations on pr439

5.2.3 Comparison of Average Solution Quality against Average Speedup

This section compares the average solution quality of each parallel implementation on each problem computed against the sequential implementation on CPU. Then, the average gap from known optimal cost to the average cost in percent is computed.

(a) Problem: berlin52, Optimal Cost: 7542

Implementation	Average Cost	Average Time(s)	Average Gap	Average Effective Speedup
ST-CPU	7775.13	0.5749	3.09%	1.00x
MT-CPU	7666.39	0.2534	1.65%	2.27x
BLOCK-GPU	7841.92	0.0330	3.98%	17.42x
THRUST-GPU	7704.22	0.0722	2.15%	7.96x

Table 5.10: Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on berlin52

(b) Problem: pr107, Optimal Cost: 44303

Implementation	Average Cost	Average Time(s)	Average Gap	Average Effective Speedup
ST-CPU	48733.12	2.3140	10.00%	1.00x
MT-CPU	45614.41	0.9239	2.96%	2.50x
BLOCK-GPU	46093.88	0.1915	4.04%	12.08x
THRUST-GPU	45558.9	0.2120	2.83%	10.92x

Table 5.11: Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr107

(c) Problem: pr152, Optimal Cost: 73682

Implementation	Average Cost	Average Time(s)	Average Gap	Average Effective Speedup
ST-CPU	87588.31	5.6841	18.87%	1.00x
MT-CPU	77760.86	2.7716	5.54%	2.05x
BLOCK-GPU	76360.63	0.5196	3.64%	10.94x
THRUST-GPU	76080.86	0.4473	3.26%	12.71x

Table 5.12: Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr152

(d) Problem: pr264, Optimal Cost: 49135

Implementation	Average Cost	Average Time(s)	Average Gap	Average Effective Speedup
ST-CPU	73724.12	25.4448	50.04%	1.00x
MT-CPU	57972.51	9.4200	17.99%	2.70x
BLOCK-GPU	54174.02	2.1163	10.26%	12.02x
THRUST-GPU	53738.24	1.6995	9.37%	14.97x

Table 5.13: Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr264

(e) Problem: pr439, Optimal Cost: 107217

Implementation	Average Cost	Average Time(s)	Average Gap	Average Effective Speedup
ST-CPU	175716.4	123.2664	63.89%	1.00x
MT-CPU	134694.5	27.5964	25.63%	4.47x
BLOCK-GPU	117527.9	9.2499	9.62%	13.33x
THRUST-GPU	117756.1	5.7646	9.83%	21.38x

Table 5.14: Comparison of Average Solution Quality against Average Speedup of Parallel Implementations on pr439

5.3 Result Analysis

From Section 5.2.2 above, we can see that our GPU implementations are able to converge faster compared to both CPU implementations across all five TSP instances chosen. It is shown that the BLOCK-GPU approach yields faster convergence in smaller problems, while the THRUST-GPU approach catches up and outperforms BLOCK-GPU in larger TSP instances. Starting from the 107-city instance, both GPU implementations are able to converge faster than the sequential CPU implementation by over an order of magnitude.

Furthermore, we observed that in terms of iteration count, BLOCK-GPU takes a lot more iterations to converge compared to other approaches. This might be due to the local nature of elite selection in the algorithm design. As we determine the elites by sorting the population locally within a block, therefore there is always chance for less fit chromosomes to be included within the elite population when we look at it globally, since there is no communication between blocks to synchronize and verify the elite population. As a result, less fit individuals from one block may be selected as elites, while globally superior individuals in other blocks may be discarded. This might cause beneficial traits to take longer to propagate through the population, and even the loss of certain potentially good solutions over time, which leads to slower convergence in terms of number of iterations.

It is important to note, however, that the raw speed of each iteration does not directly translate to faster overall convergence. The quality of convergence, that is, how effectively the algorithm moves toward an optimal or near-optimal solution, plays a critical role. Therefore, despite a faster iteration speed on the GPU, the BLOCK-GPU implementation may require more iterations to reach comparable solution quality, especially in more complex problems, due to its limited view of the global population structure.

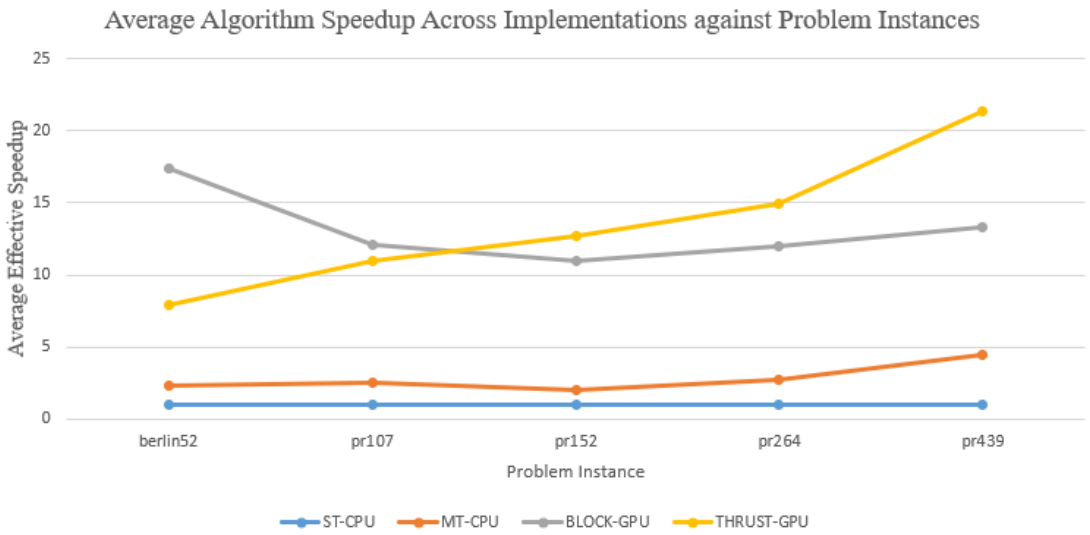


Figure 5.1: Graph of Algorithm Speedup Across Multiple Implementations against Multiple Problem Instances

Figure 5.1 above describes the average algorithm speedup across multiple implementations against the problem instances we chose. As we can see from the graph, BLOCK-GPU emerges as an efficient approach in smaller instances of the TSP as it begins with a high speedup on smaller problems such as the berlin52 problem. This is most likely due to its nature of block-level processing where no global synchronization is required, and that the sorting process involves a lot less individuals, which means that there is little overhead to perform the operations. Therefore, in smaller instances of problems where a smaller number of computations is required to reach convergence, this approach could appear quite efficient, even surpassing the THRUST-GPU approach.

However, BLOCK-GPU starts to lag behind THRUST-GPU from problem pr152 onwards. This situation will most likely be due to poorer convergence quality because of the massively increased problem complexity and lower quality of the elite population. Although during crossover, the algorithm picks parents randomly from the global elite population, the fact that there is no global best guarantee has negatively impacted the convergence speed of this algorithm.

The THRUST-GPU approach might appear slower in smaller instances of the TSP due to the fact we implement full global sorting using `thrust::sort_by_key` and `thrust::gather`. This approach, while precise and accurate for elitism, incurs more overhead – especially when the population is small. This is because we also need to take into consideration the costs of launching Thrust kernels, managing temporary

buffers, and performing global memory operations since the implementation of Thrust library is through a high-level interface[18]. These costs might dominate when there is little computational work to "hide" these costs, which in turn, outweigh the benefits of exact elitism, making the algorithm appear slower.

While the high-level abstraction of the Thrust library may carry some initial cost, as the problem size increases, the amount of computations eventually dominates over the costs. This is where the THRUST-GPU approach begins to demonstrate its full potential. As the overhead of kernel launches and Thrust functions gets "hidden", the cost becomes less impactful on overall performance. Not only that, as the full global sort provided by `thrust::sort_by_key` ensures the preservation of the true global best individuals, this leads to better convergence quality, which in turn also speeds up convergence. As a result, THRUST-GPU not only maintains robust convergence behaviour, but also becomes increasingly competitive or even superior in terms of solution quality and total runtime as the complexity of the problem grows. This can be seen from the results obtained above, where starting from problem pr152 with 152 cities, THRUST-GPU began to outperform BLOCK-GPU, and in more complex problems, proceeds to widen its performance gap with BLOCK-GPU. Therefore, THRUST-GPU has clearly demonstrated the best scalability and performance, with speedup increasing consistently across all problem instances.

Furthermore, it is important to note that both of our CUDA-accelerated implementations are not only able to be significantly faster than the CPU implementations but are also able to compete with the CPU implementations in terms of solution quality, and surprisingly, converge a lot better in larger TSP instances. This shows that our parallel implementations of the GA on the GPU have successfully leveraged the benefits of CUDA in an effective manner. Not only that, but the results also proved that our GPU implementations are very much scalable for larger instances of the TSP.

CHAPTER 6
Conclusion

This project explored the application of genetic algorithms (GAs) to solve the travelling salesman problem (TSP) with a particular focus on leveraging GPU parallelism to improve performance and scalability. By addressing the algorithmic complexity and highly iterative nature of GAs and the immense computational demands of the TSP, we demonstrated how modern GPU architectures can be utilized not only to accelerate genetic operators but also to enhance solution quality in large-scale optimization problems. Furthermore, we have also demonstrated that a CUDA-accelerated GA is able to achieve a performance of over an order of magnitude, as shown from the results we obtained from the best-performing approach, THRUST-GPU.

Through comparative analysis between the four strategies, namely single-threaded CPU, multi-threaded CPU, block-level sorting and elitism on GPU (BLOCK-GPU) and sorting with Thrust on GPU (THRUST-GPU), we identified the tradeoffs between raw performance and solution quality. While BLOCK-GPU excels in smaller problem instances due to minimal overhead, THRUST-GPU shines in larger problem instances with its ability to preserve elite solutions more effectively.

The results obtained from the experiment affirm that as GPUs become more accessible, integrating them into evolutionary algorithms such as GA is not only feasible but also highly beneficial for solving complex combinatorial problems such as the TSP as it is proven to be performant and scalable. Furthermore, with consistent development of the CUDA architecture, and its use for general purpose computing on GPUs, developing problem-solving methods to be applied on the GPU has become a lot less intimidating, and it is wise to leverage and utilize its benefits. This project reinforces the importance of thorough understanding of hardware architecture on algorithm design and opens pathways for extending such optimizations to brother classes of optimization problems such as routing, scheduling and logistics.

While this project demonstrates the significant benefits of GPU-accelerated GAs for solving the TSP, there are several promising avenues for further development, including global elitism enhancements that can combine the low-overhead advantage of BLOCK-GPU and the high throughput of THRUST-GPU, warp-level optimization

such as leveraging warp-level primitives for genetic operators, and also the integration of metaheuristics with local search to further refine solutions.

REFERENCES

[1] "The Traveling Salesman Problem," in *Combinatorial Optimization: Theory and Algorithms*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 527–562. doi: 10.1007/978-3-540-71844-4_21.

[2] "Delivering the Goods: E-commerce Logistics Transformation," 2018.

[3] F. Chen, W. Zhang, R. Chen, F. Jiang, J. Ma, and X. Zhu, "Adapting carbon neutrality: Tailoring advanced emission strategies for developing countries," *Appl Energy*, vol. 361, p. 122845, May 2024, doi: 10.1016/J.APENERGY.2024.122845.

[4] D. Luebke and G. Humphreys, "How GPUs work," *Computer (Long Beach Calif)*, vol. 40, no. 2, pp. 96–100, Feb. 2007, doi: 10.1109/MC.2007.59.

[5] M. Abbasi, M. Rafiee, M. R. Khosravi, A. Jolfaei, V. G. Menon, and J. M. Koushyar, "An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems," *Journal of Cloud Computing*, vol. 9, no. 1, Dec. 2020, doi: 10.1186/s13677-020-0157-4.

[6] L. Rudeanu and M. Craus, "Parallel implementation of ant colony optimization for travelling salesman problem," *WSEASs Transactions on Systems*, vol. 3, no. 3, pp. 1161–1166, 2004.

[7] S. Chen, S. Davis, H. Jiang, and A. Novobilski, "CUDA-based genetic algorithm on traveling salesman problem," in *Computer and Information Science 2011*, Springer, 2011, pp. 241–252.

[8] J. H. Wilck IV and T. M. Cavalier, "A Genetic Algorithm for the Split Delivery Vehicle Routing Problem," *American Journal of Operations Research*, vol. 02, no. 02, pp. 207–216, 2012, doi: 10.4236/ajor.2012.22024.

[9] T. V Mathew, "Genetic algorithm," *Report submitted at IIT Bombay*, vol. 53, pp. 18–19, 2012.

[10] J. Ghorpade, “GPGPU Processing in CUDA Architecture,” *Advanced Computing: An International Journal*, vol. 3, no. 1, pp. 105–120, Jan. 2012, doi: 10.5121/acij.2012.3109.

[11] “CUDA C++ Best Practices Guide.” Accessed: Dec. 05, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#thread-and-block-heuristics

[12] N. Fujimoto and S. Tsutsui, “Parallelizing a genetic operator for GPUs,” in *2013 IEEE Congress on Evolutionary Computation*, 2013, pp. 1271–1277. doi: 10.1109/CEC.2013.6557711.

[13] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, “NVIDIA Ampere Architecture In-Depth.” [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[14] P. Gupta, “CUDA Refresher: The CUDA Programming Model.” [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[15] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul, “Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs,” in *Proceedings - IEEE International Conference on Data Mining, ICDM*, May 2010, pp. 1001–1006. doi: 10.1109/ICDM.2010.21.

[16] M. Danish, S. Kumar, S. Kumar, and A. Qamareen, “Chemical Product and Process Modeling Optimal Solution of MINLP Problems Using Modified Genetic Algorithm,” *Chemical Product and Process Modeling*, vol. 1, pp. 1–41, May 2007, doi: 10.2202/1934-2659.1010.

[17] G. Reinelt, “TSPLIB 95.” [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[18] “1. Introduction — thrust 12.2 documentation,” Nvidia.com. Accessed: Apr. 28, 2025. [Online]. Available: https://docs.nvidia.com/cuda/archive/12.2.2/thrust

POSTER

UTAR
UNIVERSITI TUNKU ABDUL RAHMAN
Wholly owned by UTAR Education Foundation
(Co. No. 578227-M)

FACULTY OF INFORMATION COMMUNICATION AND TECHNOLOGY

Parallel Metaheuristic Algorithm for Route Planning using CUDA

Introduction

The Travelling Salesman Problem (TSP) is a combinatorial optimization problem that is NP-hard. Computational demands increase significantly as the problem gets more complex. This requires sophisticated design and optimization on algorithms and immensely powerful hardware to perform such computations. This research will be studying the feasibility of parallelizing a metaheuristic algorithm, the genetic algorithm (GA) using CUDA to enhance its performance.

Objective

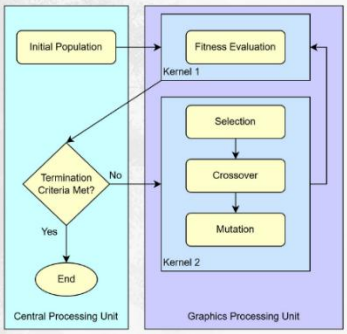
- To demonstrate the capability of parallelization in achieving significant speedup compared to sequential implementations.
- To improve scalability of parallel GA in handling large-scale VRP instances.
- To contribute towards the field of research of massively parallel metaheuristic algorithms

Method

- Development of a massively parallel GA on Nvidia GPUs using CUDA
- Parallelize the components within the GA and optimize them within the CUDA kernel
- Introduced two strategies to parallelize GA on GPU: block-level sorting and elitism, and utilizing Thrust library to sort

Results (Speedup Achieved)

Cities	ST-CPU	MT-CPU	BLOCK-GPU	THRUST-GPU
52	1.00x	2.27x	17.42x	7.96x
107	1.00x	2.50x	12.08x	10.92x
152	1.00x	2.05x	10.94x	12.71x
264	1.00x	2.70x	12.02x	14.97x
439	1.00x	4.47x	13.33x	21.38x



Analysis

Block-level sorting and elitism has low overhead, which excels at less computational work, but has reduced elite population quality, which hurts performance in larger TSP instances. Thrust library is applied through a high-level interface. High overhead costs dominate at small problems, but is hidden by large computational workload, thus very scalable.

Conclusion

This project demonstrated not only the feasibility but also the benefits of parallelizing GA to solve the TSP using CUDA in terms of performance and scalability.

Researcher
Daniel Looi Jun Jie

Research Supervisor
Ts Wong Chee Siang