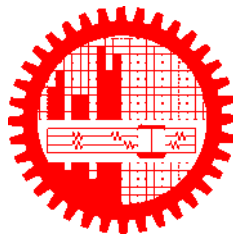


M.Sc. Engg. (CSE) Thesis

# **A NEW PARALLEL ALGORITHM FOR RECURSIVE BEST FIRST SEARCH ON A GPU**

Submitted by  
Sifat E Jahan  
1015052084

Supervised by  
Dr. Rifat Shahriyar



Submitted to  
**Department of Computer Science and Engineering**  
**Bangladesh University of Engineering and Technology**  
Dhaka, Bangladesh

in partial fulfillment of the requirements for the degree of  
Master of Science in Computer Science and Engineering

March 2022

*Dedicated to my parents*

## Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "A NEW PARALLEL ALGORITHM FOR RECURSIVE BEST FIRST SEARCH ON A GPU", is the outcome of the investigation and research carried out by me under the supervision of Dr. Rifat Shahriyar, Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

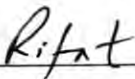
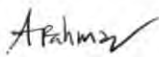
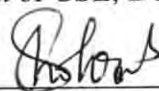


  
\_\_\_\_\_

Sifat E. Jahan

1015052084

The thesis titled “A NEW PARALLEL ALGORITHM FOR RECURSIVE BEST FIRST SEARCH ON A GPU”, submitted by Sifat E Jahan, Student ID 1015052084, Session October 2015, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on March 9, 2022.

### Board of Examiners

1.   
\_\_\_\_\_  
Dr. Rifat Shahriyar  
Professor  
Department of CSE, BUET, Dhaka  
Chairman  
(Supervisor)
2.   
\_\_\_\_\_  
Dr. A. K. M. Ashikur Rahman  
Professor and Head  
Department of CSE, BUET, Dhaka  
Member  
(Ex-Officio)
3.   
\_\_\_\_\_  
Dr. Md. Shohrab Hossain  
Professor  
Department of CSE, BUET, Dhaka  
Member
4.   
\_\_\_\_\_  
Dr. A. B. M. Alim Al Islam  
Professor  
Department of CSE, BUET, Dhaka  
Member
5.   
\_\_\_\_\_  
Dr. Md. Golam Rabiul Alam  
Associate Professor  
Department of CSE  
BRAC University, Dhaka  
Member  
(External)

## **Acknowledgement**

I am thankful to my thesis supervisor Dr. Rifat Shahriyar for his constant guidance and support. This thesis would not have been possible without his immense support, knowledge, guidance and patience. His constant motivation helped me a lot to complete this thesis.

I would also like to thank my board of examiners Dr. A. K. M. Ashikur Rahman, Dr. Md. Shohrab Hossain, Dr. A. B. M. Alim Islam, and Dr. Golam Rabiul Alam for helping me throughout my thesis. Their informative discussions, feedback, and cooperative nature have motivated and guided me.

Dhaka  
March 9, 2022

Sifat E Jahan  
1015052084

# Contents

<b>Candidate’s Declaration</b>	<b>i</b>
<b>Board of Examiners</b>	<b>ii</b>
<b>Acknowledgement</b>	<b>iii</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>
<b>List of Algorithms</b>	<b>ix</b>
<b>Abstract</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	2
1.2 Motivation . . . . .	2
1.3 Scope and Contributions . . . . .	3
1.3.1 Major Contributions . . . . .	3
1.4 Objectives and Outcomes . . . . .	4
1.5 Thesis Outline . . . . .	4
<b>2 Literature Review</b>	<b>5</b>
2.1 Background . . . . .	5
2.1.1 Searching in Artificial Intelligence . . . . .	5
2.1.2 Different Searching Techniques . . . . .	6
2.1.3 Recursive Best First Search (RBFS) . . . . .	7
2.1.4 A* search and RBFS algorithm . . . . .	8
2.1.5 Multi-threading . . . . .	9
2.1.6 Sequential and Parallel Algorithms . . . . .	10
2.1.7 Central Processing Unit (CPU) and Graphics Processing Unit (GPU) . . . . .	11
2.1.8 Computation Offloading . . . . .	12
2.1.9 CUDA and OpenCL . . . . .	13

2.1.10	Aparapi Framework . . . . .	13
2.1.11	Tornado VM . . . . .	14
2.1.12	JOCL . . . . .	15
2.2	Related Works . . . . .	15
<b>3</b>	<b>Proposed Model</b>	<b>18</b>
3.1	Problem Formulation . . . . .	18
3.2	System Model . . . . .	19
3.3	Methodology . . . . .	19
3.3.1	Parallel Computation of the Heuristics Functions . . . . .	19
3.3.2	Pruning Duplicate States . . . . .	20
3.3.3	Choosing Best Fitted Child States . . . . .	22
3.3.4	Parallel RBFS Execution . . . . .	24
3.3.5	Algorithm: GRBFS . . . . .	25
<b>4</b>	<b>Simulation Results</b>	<b>28</b>
4.1	Sliding Puzzle . . . . .	28
4.2	Experimental Setup . . . . .	30
4.3	Experimental Results . . . . .	31
4.3.1	Sequential RBFS Algorithm Performance . . . . .	31
4.3.2	Parallel RBFS Algorithm without GPU Performance . . . . .	33
4.3.3	Parallel RBFS Algorithm with GPU Performance . . . . .	33
4.3.4	States Expanded in Sequential and Parallel RBFS Algorithm . . . . .	35
4.3.5	Aparapi Performance on Parallel RBFS . . . . .	35
4.4	Analysis and Comparison of results . . . . .	36
4.5	Performance Evaluation . . . . .	43
4.6	Discussion . . . . .	45
<b>5</b>	<b>Conclusion and Future Works</b>	<b>46</b>
5.1	Conclusion . . . . .	46
5.2	Future Directions of Further Research . . . . .	46
	<b>References</b>	<b>48</b>
	<b>Index</b>	<b>52</b>
<b>A</b>	<b>PseudoCodes</b>	<b>53</b>
A.1	Sequential RBFS pseudocode . . . . .	53
<b>B</b>	<b>Sample GPU Codes</b>	<b>55</b>
B.1	Matrix multiplication with GPU . . . . .	55

B.2	Array Summation using Aparapi . . . . .	60
B.3	Matrix multiplication using Tornado VM . . . . .	60
B.4	Array Summation using JOCL API . . . . .	61



# List of Figures

2.1	Search Problem	6
2.2	RBFS Algorithm Mechanism	8
2.3	Sequential and Parallel execution	9
2.4	Single Threaded and Multiple Threaded Execution	10
2.5	A basic architecture diagram of a multi-core processor	10
2.6	CPU and GPU Architecture	12
2.7	CUDA Execution Mode	14
2.8	Tornado Execution	14
3.1	Flow Diagram of the Proposed System Model.	20
3.2	Parallel Fitness Calculation	21
3.3	Exponential Search Space	21
3.4	Pruning Duplicate States	22
3.5	Best-fitted states	24
3.6	Parallel GRBFS sources	25
3.7	Parallel Approach	25
4.1	8-Puzzle State	29
4.2	15-Puzzle State	29
4.3	8-Puzzle Heuristics	30
4.4	8-Puzzle Solution	31
4.5	Computation Time of 8-Puzzle instances with small optimal steps	37
4.6	Computation Time of 8-Puzzle instances with larger optimal steps	39
4.7	Computation Time of 15-Puzzle instances	39
4.8	Memory requirement of 8-Puzzle instances	40
4.9	Memory requirement of 15-Puzzle instances	41
4.10	8 Puzzle States expanded in RBFS	41
4.11	15 Puzzle States expanded in RBFS	42
4.12	Aparapi computational time	42
4.13	Aparapi memory requirement	43
4.14	8 puzzle Performance Evaluation CPU-based RBFS with GPU-based RBFS	44
4.15	15 puzzle Performance Evaluation CPU-based RBFS with GPU-based RBFS	44

# List of Tables

4.1	Sequential 8 Puzzle RBFS results . . . . .	32
4.2	Sequential 15 Puzzle RBFS results . . . . .	32
4.3	Parallel 8 Puzzle RBFS without GPU . . . . .	33
4.4	Parallel 15 Puzzle RBFS without GPU . . . . .	34
4.5	Parallel 8 Puzzle RBFS in GPU . . . . .	34
4.6	Parallel 15 Puzzle RBFS in GPU . . . . .	35
4.7	8 Puzzle States expansion . . . . .	36
4.8	15 Puzzle States expansion . . . . .	36
4.9	Aparapi Performace . . . . .	37
4.10	Computation Time Analysis . . . . .	38
4.11	Memory Requirement of RBFS . . . . .	40

# List of Algorithms

1	Parallel RBFS on GPU	27
---	----------------------	----

# Abstract

Parallel programming has emerged preeminence as an efficient paradigm for designing and solving complex problems. In recent years, the usage of Graphics Processing Units (GPUs) with parallel approaches has scaled up the computational speed of the traditional CPU-based algorithms. Therefore, developing a parallel adaptation of the fundamental algorithms has become essential to harness the advantages of modern multi-core processors. This thesis describes the first variant, which exploits parallelism from a well-known Artificial Intelligence (AI) searching algorithm (Recursive Best First Search, RBFS) using GPU. It proposes a methodology to convert a sequential RBFS algorithm to a parallel algorithm that provides enhanced solutions. Furthermore, the proposed algorithm has been studied thoroughly in centralized and distributed optimization contexts. The performance analysis on sliding puzzles illustrates the superiority gained by using GPU, resulting in excelled performances and scalability. The proposed parallel GPU-based RBFS can achieve significant computational speed-up for large-scale search problems compared to the traditional sequential CPU-based RBFS. Moreover, different approaches of GPU programming have been considered, showing the impact of using a naïve framework (Aparapi) for Java based implementation and CUDA based implementation using Python. In addition, the proposed model will be effective in any non-GPU based parallel system as it is not solely focused to enhance performance on GPU based architecture.

# Chapter 1

## Introduction

It is widely foreseen that in future, the modern microprocessors will have more computing cores; hence, the computation tasks without parallel adaptation will become less effective according to the system performance. At present, the world is evolving around modern Artificial Intelligence advances and heuristic search is one of the most widely used problem solving pathways.

Recursive Best First Search (RBFS) is a well-known heuristic based searching algorithm in Artificial Intelligence, which resembles the best first search algorithm using linear search space. It performs similar to recursive depth-first search (DFS), however instead of going down to the current path; it remembers the best alternative path from the ancestors of the current path. One of the outstanding parts of this algorithm is that it solves the memory issues of A\* search algorithm. Intrinsically, RBFS is a recursive algorithm, which efficient parallelization is quite challenging. The central challenge in parallelizing recursive best-first search is avoiding contention between threads when accessing the function several times due to its recursive approach. There are very few studies focused on this algorithm and only few possible approaches have been shown.

Recently, the Graphics Processing Unit (GPU) has been used to accelerate various traditional CPU-based computational tasks. GPUs manage tasks differently than CPUs; usually CPUs perform well in sequential execution whereas GPUs perform better handling parallel executions. This thesis aims at inheriting the advantages of both GPUs' powerfulness with the recursive best-first search scheme. It proposes the first parallel variant of RBFS, which is able to execute in GPU in an immense parallel manner that surpasses the computational speed of the traditional sequential RBFS algorithm. Moreover, the proposed parallel RBFS algorithm can be implemented in any device without GPU that supports parallelization techniques and will provide compelling solutions.

## 1.1 Problem Statement

In recent years, there has been a developing interest with GPU programming to accelerate the computational speed of the traditional algorithms. Typically, a GPU contains thousands of cores that can handle different data at the same time very efficiently, whereas a CPU contains advanced cores to handle sequential execution. There have been many different approaches of GPU in different algorithms to accelerate its computational speed showing that efficient re-design of traditional sequential algorithms to parallel algorithms can enhance performance using GPUs.

Recursive best-first search (RBFS) is one of the heuristic search algorithms, which belongs to Artificial Intelligence algorithms. In this algorithm, the nodes are expanded in best-first order, which is chosen based on the specific environment of the problem and uses a backtracking condition to expand only the required node based on a cost function finding the most optimal path with least expansion possible.

GPU has been used for developing different Artificial Intelligence algorithms such as Genetic programming, A\* search algorithm or best-first search as well as traditional algorithms such as breadth first search and shows a notable increase in performance. There have been several studies on performance of GPUs to understand its performance over CPUs, which shows significant results. However, the integration of GPU hardware with algorithms remains challenging and only efficient remodel of the sequential algorithms can provide excellence results.

We claim that designing an efficient parallel version of the sequential RBFS algorithm will speed up its computational speed significantly employing GPUs for large-scale search problems.

## 1.2 Motivation

Parallel programming is blooming as a new technology nowadays and with the availability of multiple cores in the modern devices, GPU based computing is denominating the current world. The computational power of GPU is an impressive driver for further progress in different sectors such as data science or machine learning.

However, designing parallel algorithms that will work on GPUs efficiently has remained a challenge. Hence, the study of design of the traditional sequential algorithms to parallel algorithms for GPU hardware has immense significance.

There have been several studies about different traditional searching algorithms such as A\* search, breadth first search, genetic algorithms, etc. However, none of the studies has focused on parallelization of RBFS to increase its computation speed using GPU architecture.

RBFS is a recursive algorithm, which parallelization is quite challenging due to its recursive nature. There are not enough studies that have been focused on extensive analysis of the performance of this algorithm. Moreover, there have been very limited studies of the

parallelization of this algorithm that makes it an interesting topic to study.

Considering all the possible outcomes of GPU programming, this study has focused its interest on integrating the modern technology advances with a well-known algorithm (RBFS) that will provide insight on the design of parallel algorithms, which will enhance the computational speed of the sequential algorithms. This research will provide a rigorous study on the behavior of the parallel algorithms on heuristic based search problems.

## 1.3 Scope and Contributions

This research work focuses on the design of the parallel approach of RBFS algorithm, which will increase the computational speed of the sequential RBFS algorithm using the latest technology employing GPUs. Researchers have proposed very few approaches of parallelization of the RBFS algorithm and there are very limited studies about increasing its computational speed. Most prior studies only focus on improvement of the sequential RBFS algorithm but none of those concentrates on parallelization of it using GPU. The proposed parallel RBFS algorithm increases significantly the computational speed compared to the sequential RBFS algorithm.

It proposes a parallel architecture for remodeling the sequential algorithms that can perform efficient on the GPU hardware.

An experimental analysis on CPU platform as well as GPU platform in a large-scale optimization problem has shown on both sequential and parallel RBFS algorithm.

Furthermore, this research paper studied several approaches of GPU programming using two high-level languages (Java and Python) and trade-off their computational speed in large-scale optimization problems.

### 1.3.1 Major Contributions

The contributions made by this thesis work are as follows:

- This thesis proposes a new parallel approach that can parallelize sequential algorithms.
- This research study proposes a new parallel approach to parallelize recursive algorithms.
- There are no parallel approaches of RBFS algorithm until this thesis, which shows a naïve way to implement parallel algorithm for RBFS.
- Prior studies have not focused on RBFS algorithm on a GPU platform; hence, this study illustrates an extensive analysis of RBFS on a GPU device.
- The proposed algorithm enhances the performance of the RBFS algorithm trading off the constraint between time and memory.

- The proposed model shows that the computational speed of the RBFS algorithm on GPU decrease while increasing the difficulty level of the problem compare to the traditional RBFS approach.
- This study also demonstrates several possible GPU implementations. It determines the unsuitability of the Aparapi framework for the proposed model and it shows that Aparapi does not perform well for any algorithm.

## 1.4 Objectives and Outcomes

The objectives of this thesis are as follows:

1. To design a parallel version of the RBFS algorithm, which provides a faster solution than the existing sequential algorithm.
2. To implement the parallelized version of the RBFS algorithm on a server and a GPU.
3. To study the comparative performance of the sequential and parallel versions of the RBFS algorithm on a server and a GPU for different heuristic-based search problems.

The possible outcomes of this thesis are as follows:

1. A new parallel version of RBFS algorithm, which provides a faster solution than the existing sequential algorithm by minimizing the computation time.
2. Two separate implementations of the parallel RBFS algorithm (server and GPU).
3. Comprehensive performance analysis of the sequential and parallel versions of RBFS algorithm in different heuristic-based search problems.

## 1.5 Thesis Outline

This thesis has been structured in four critical sections outlined below. Chapter 2 provides an overview on different artificial intelligence searching algorithms, including uninformed searches and informed searches. As well as, different parallelization techniques with GPUs are mentioned. In addition, a detailed background on previous studies on parallelization of searching algorithms is given. Chapter 3 insights the experimental methodology of the proposed parallel RBFS algorithm including the algorithm as well. Chapter 4 discusses the simulation results of the proposed algorithm. Finally, Chapter 5 presents the contributions of the research work addressing the challenges faced in order to parallelize a recursive algorithm and achieve high performance from it using GPUs. Furthermore, it identifies the future directions of this research work.



# Chapter 2

## Literature Review

This chapter provides background information on the Artificial Intelligence searching techniques and the basics of parallel programming and GPU architecture to place the research contributions in context. This chapter starts with a brief introduction to different searching techniques. Section 2.1 describes the background details, including basic parallel programming concepts CPU and GPU architecture, as well as outlines the sequential RBFS algorithm. Section 2.3 provides detailed background on studies of parallel approaches and RBFS algorithm performance analysis.

### 2.1 Background

#### 2.1.1 Searching in Artificial Intelligence

Artificial Intelligence (AI) is a computer science branch. Mainly concerned with providing human-mind behaviors in machines, i.e., it focuses on making intelligent machines that can perform similar functions to human beings.

One of the most prominent areas of AI is search. An example of a search problem containing several states, stating the start point to the target point, is shown in Figure 2.1. The solution to search problems needs to be determined in a systematic way (trial-and-error approach). There are many problem variations such as chess, sliding puzzle, and even medical diagnosis where searching is required [1]. Any searching algorithm consists of the following parts:

- **Problem search space/state space** refers to the environment in which the searching will take place.
- **Initial state** referring to the state where the searching begins.
- **Goal State** refers to the state where the searching should stop.

- A **solution** will be achieved by using the search algorithms, which will follow some defined steps to get the result if possible, otherwise will return failure.

### 2.1.2 Different Searching Techniques

There are several search algorithms in AI. In this below section, the most fundamental search algorithms will be discussed. These can be categorized broadly into two types:

- **Uninformed Search** - often called brute-force search or blind search. These types of algorithms do not have or need any additional information about the search space. An uninformed search only knows how to traverse the search space identify the goal state and leaf states. Therefore, it will examine all the states of the search tree until the solution is found if it exists. There are mainly five types:
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Uniform cost search
  - Iterative deepening depth-first search
  - Bidirectional search
- **Informed Search** - also called **heuristic search** [2]. These types of algorithms have domain-specific knowledge to increase the performance of the techniques. These algorithms have information about the closeness of the goal state from a given state,

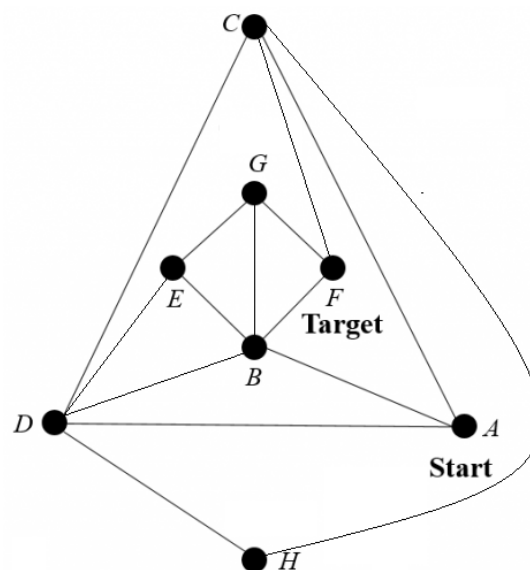


Figure 2.1: Search Problem

which is defined by a property known as heuristic. A heuristic is a function defined as  $h(n)$ , which determines the closeness of the goal state. Each state is selected based on the value of the heuristic function. There are various types of informed search:

- Greedy Best-first search
- A\* search
- Iterative Deepening A\* (IDA\*) search
- Recursive Best First Search (RBFS)

There is another type of searching approach in AI known as **local search algorithms**, which starts from an acceptable solution and then moves to another solution (looking at the neighboring solution). These types of algorithms can give an approximation solution anytime even if these are interrupted before ends; those also may lead to a not optimal solution. Most commonly known as hill-climbing search and simulated annealing search.

### 2.1.3 Recursive Best First Search (RBFS)

A straightforward algorithm that imitates the technique of best-first search. The highlighted feature of this algorithm is that it uses simply linear space while performing the search.

The evaluation function  $f(n)$  used is

$$f(n) = g(n) + h(n) \quad (2.1)$$

where  $g(n)$  is the path cost from the source state to the current state and  $h(n)$  is the estimated cheapest cost from the current state to the goal state.

Hence,  $f(n)$  defines the estimated cheapest solution to the goal state traversing through the current state ( $n$ ). In each step, it calculates  $f(n)$  and decides which state to choose.

The approach is quite similar to Recursive DFS; however, instead of going to the maximum depth of the current state, it keeps track of the fitness value (commonly known as f-value) of the best alternate path. There is a limit set while choosing a state; if the current state surpasses the limit, the recursion backtracks and determines the alternate best state. The f-values of every state of the current path change with the best f-value of its children states while backtracking. By updating f-values, the RBFS algorithm can decide if it is worth re-expanding the forgotten sub-tree sometime afterward. It uses the given evaluation function in 2.1, which refers to the path cost from the source state to the current state and refers to the estimated cheapest cost from the current state to the goal state to expand the state or not.

In Figure 2.2 the state expansion of the RBFS algorithm with backtracking is shown. Starting from the source state S1, the nodes are expanded according to the lowest fitness values of the

successors, while a state expanding it remembers the second-best fitness value of the successors. From the state S1, it expands to S3, then S7 till S8. When the expanded node  $n$  sees its child states' fitness are not better than the backup state's fitness value, then it does not expand further and start backtracking to the backup state S5 denominating as  $n'$  from where the algorithm continues the search to the goal state. The backup stored parent is defined using  $k$ , whereas  $k'$  is used to represent the discarded sub-tree whose fitness value gets updated with fitness value to the best successor state of  $k'$ .

The space complexity of the algorithm is  $O(bd)$ . The time complexity is difficult to measure as it depends on the accuracy of the heuristic function and the frequency of the change in paths. However, in the worst case, an asymptotic time complexity can be  $O(b^d)$ . Here,  $b$  is the branching factor, and  $d$  is the maximum search depth. The pseudocode of the traditional sequential RBFS algorithm is shown in given in Section A of the Appendix.

### 2.1.4 A\* search and RBFS algorithm

A\* search is a widely renown heuristic algorithm closely related to RBFS. It starts from the source state and finds the route to the goal state having the minimal cost possible. It uses the same fitness function as RBFS and maintains a tree of the paths that it has expanded. While implementing this algorithm, it considers two different lists to store the expanded nodes: an open list, which contains the states to expand, and a closed list, which stores the visited states to prevent unnecessary re-expansion of the same state.

The main advantage of RBFS over the A\* search algorithm is the memory usage. The RBFS algorithm remembers only the next best favorable state, whereas, in A\* search, it keeps track of all the visited states. Consequently, A\* search becomes impractical for large-scale problems due to enormous memory requirements.

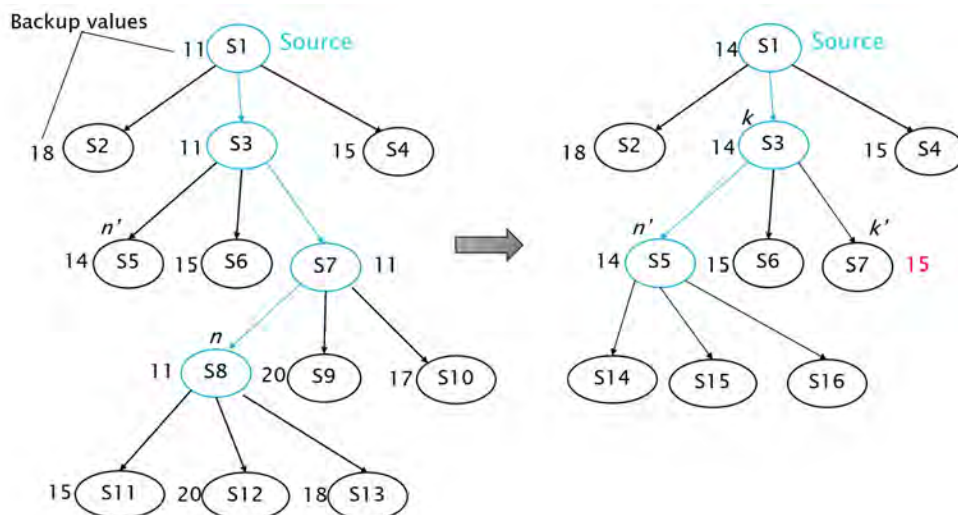


Figure 2.2: RBFS Algorithm Mechanism

Both Algorithms work well in finding solutions to searching problems. These are complete and optimal algorithms if the heuristics are admissible and consistent. However, RBFS has a linear space complexity, whereas A\* search is exponential like any graph-search algorithm.

### 2.1.5 Multi-threading

A thread is the smallest single sequential instruction within a program that the processor can manage independently. Multiple threads operating in a single process are known as multi-threading. The difference in the execution mode of single-threaded and multi-threaded applications is shown in Figure 2.4. Threads enhance the possibility of managing a program more efficiently by executing multiple tasks simultaneously. These threads share the resources and run concurrently, making them suitable for parallel processing.

Multi-threaded processors can execute several threads concurrently by controlling those in a single pipeline. To fulfill this operation, these processors must be able to control various threads in parallel by using independent program counters, internal tagging mechanisms that differentiate different threads instructions, and context-switching capability. The overhead of context switching must be minimal to gain higher performance [3].

The advantages of multi-threading is that it enables the processor to utilize effectively its resources which enhance performance within the limitation of the processor. Multi-threading applications demand has increased over the years due to the expansion of multi-core processors, however developing such programs is still challenging. The basic multi-core processor architecture is shown in Figure 2.5. Several issues arise while designing a concurrent application such as concurrency errors that include deadlocks and race conditions. As the execution of threads are non-deterministic, several executions of the same multi-threaded application can lead to different results [4].

Different high-level languages support thread implementations and provide the necessary mechanism to develop multi-threaded applications efficiently. As there are possibilities of concurrency problems several lock and release mechanisms are available now to enhance parallel processing. Currently, several programming languages such as Java's ExecutorService



Figure 2.3: Sequential and Parallel execution

or Python's Threading module, provides an easier platform to handle threads.

### 2.1.6 Sequential and Parallel Algorithms

Sequential algorithms execute the instructions steps in consecutive order to solve the problem whereas parallel algorithms divide the problem into smaller parts that are executed in parallel producing individual output that are merged together to produce the final result. The difference in execution mode is shown in Figure 2.3.

Dividing large-scale problems in smaller sub problems is difficult as they might have data dependency. Parallelization of sequential algorithms is sometimes challenging as many sequential algorithms depend on some calculations done in the previous step, which needs to be remembered to process further. Any sequential algorithm's structure modifies a lot while

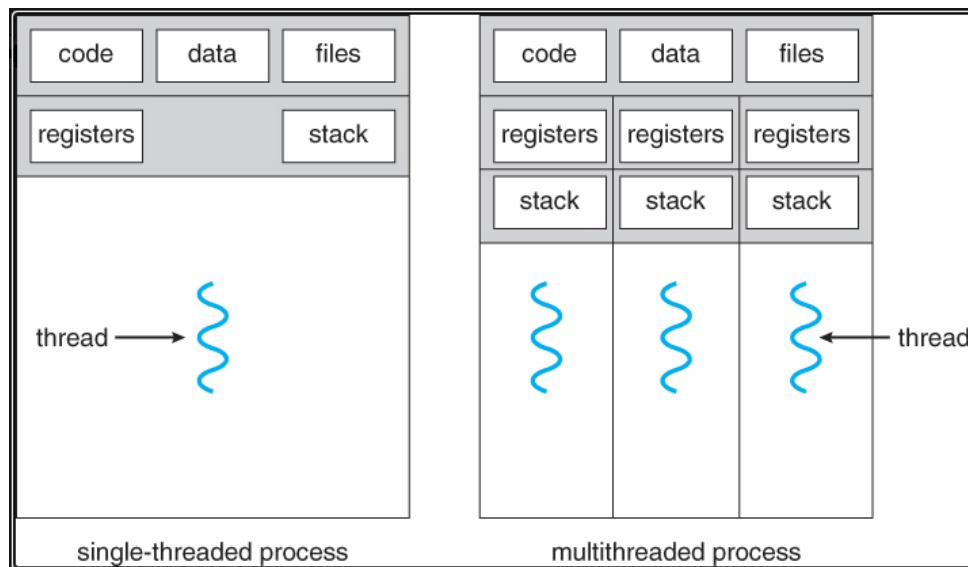


Figure 2.4: Single Threaded and Multiple Threaded Execution

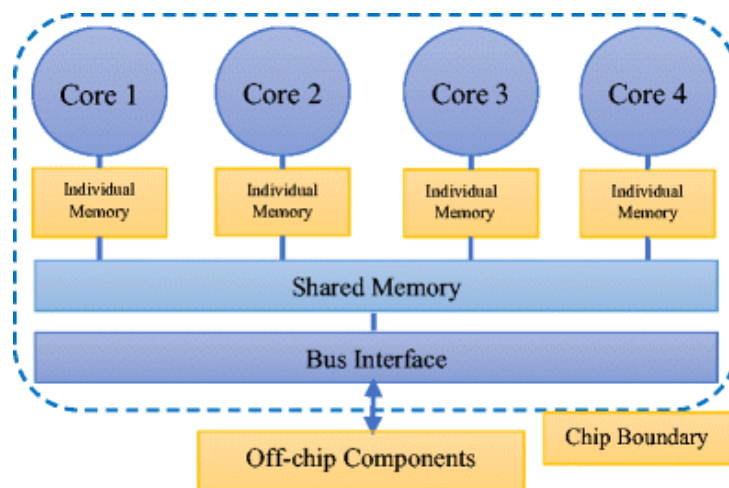


Figure 2.5: A basic architecture diagram of a multi-core processor

applying parallelization techniques. However, without parallelization of algorithms, the benefits of using resources of Graphics Processor Units (GPUs) cannot be achieved. Some parallelization techniques have shown in [5]. An example of parallel matrix multiplication has been shown to demonstrate how parallelization can be done in a sequential algorithm. Besides, the experimental results have presented significant improvement over the sequential approach.

Usually the complexity of the algorithms is measured by its execution time and memory allocations. However, in parallel programming there are two other concerns:

- **Communication:** As parallel programming involves several processors, the communication between the processors need to be optimized to make an efficient parallel design. There are two ways of communication: shared memory (uses locks on the data to ensure correct message transference between processors) and message passing (uses channels and message boxes, which occupy additional memory) Both mechanisms involve extra overhead that need to be addressed carefully in order to get a good parallel algorithm.
- **Load Balancing:** the overall work/load needs to be balanced, so that all processors involved can get the almost the same amount of work without burdening some processors with excessive processing whereas making others sit idle.

### 2.1.7 Central Processing Unit (CPU) and Graphics Processing Unit (GPU)

Graphics Processor Units (GPUs) are many-core processors that can process large-scale data quite efficiently [6]. It is designed especially for executing data in a parallel manner very efficiently. Thus, using GPUs may reduce the computational time of many algorithms and can improve the performance given by the traditional Computer Processor Units (CPUs) [7]. Nowadays, practically every computer or laptop device has integrated GPUs to enhance user usability, which influences its usage due to its higher performance. Previously, it was used for accelerating the performance of video games that needed a high level of computational tasks. Observing the possibility that the GPUs provide, more researchers have been interested in it in order to improve the performance of the traditional algorithms. There have been quite a number of research works where GPUs have been used to increase the efficiency of the traditional algorithms in recent years. There are several frameworks as well to facilitate the GPU implementation such as Aparapi [8] or Tornado VM for Java language binding to OpenCL [9].

CPU (Central Processing Unit) is a hardware device that acts as the brain structure of any embedded system. It contains an ALU (Arithmetic Logic Unit) for storing data temporarily, performing operations, and for sequencing and branching instructions it has a CU (Control Unit). CPU also interacts with memory, input and output units for executions. Usually it contains several cores, which are highly optimized for execution of sequential instructions. CPU is dedicated for the performance of the operating systems and applications. It emphasizes low



latency. CPU performs more effectively if sequential instructions are involved and requires more memory.

GPU (Graphics Processing Unit) is hardware designed to compute using displays. It contains thousands of cores that are more efficient than the CPU's cores, which can handle various data at the same time. GPU mainly handled display related data operations; used previously to render images for video games. It emphasizes on high throughputs and performs more effectively with parallel instructions; as well as the memory requirement is low.

In comparison with the small number of cores available in the CPUs, the thousands of cores in GPUs provide an enormous capability of parallel programming for scientific computation.

The architecture of CPU and GPU is shown in Figure 2.6.

### 2.1.8 Computation Offloading

Computation offloading is an emerging research area, which mainly focuses on addressing the limitations of the resources. The basic idea is to transfer highly intensive computational tasks to a different processor such as a hardware accelerator (GPU) or external platform (cluster, grid or cloud). There are several benefits of using offloading such as efficient power management, less storage requirements and improvement on application performance.

CPUs computational tasks are executed using elementary arithmetic, control logic and input/output operations and their performance is dependent on the instructions per seconds. GPUs contain numerous low-performance cores and it is highly efficient than CPUs, which contain a limited amount of cores. Keeping the CPU or main memory available to perform different tasks without loading with highly intensive tasks and executing those in the secondary memory or GPU have shown acceleration in the performance. There are many applications of offloading nowadays in mobile computing, video games and cloud servicing providing enhanced performance.

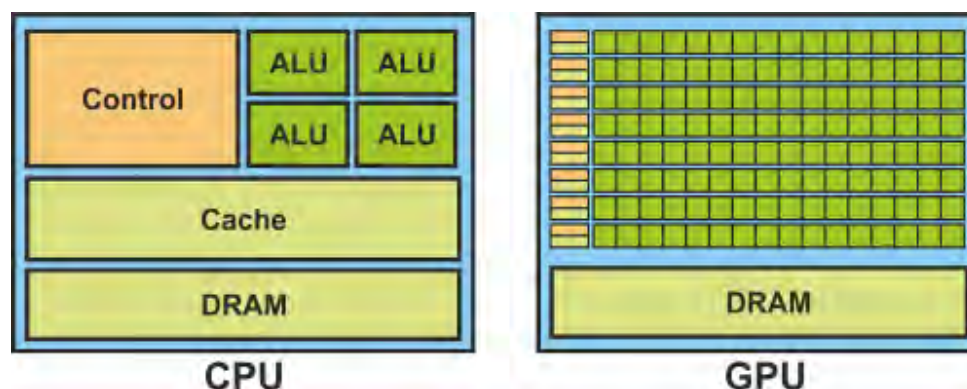


Figure 2.6: CPU and GPU Architecture



### 2.1.9 CUDA and OpenCL

CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are software frameworks that allow performing general purpose computations on GPUs.

CUDA is a parallel computing platform developed by NVIDIA for programming on graphical processing units (GPUs). Mainly, it is a programming model that utilizes GPU to speed up different computational tasks. It provides the developers the ability to speed up highly intensive computer programs by harnessing GPUs' power. It is a recent technology, which was launched in 2006 by NVIDIA developers for commercial use. It has shown significant boost up of several applications over the years. The execution model is a GPU device is shown in Figure 2.7.

Its close competitor is OpenCL, which was launched in 2009 by Apple and the Khronos Group. However, CUDA is still leading the performance of GPUs on highly extensive computer applications. It can give up to 50x performance improvements over CPUs applications.

OpenCL is a cross-platform solution; OpenCl codes can be executed in any operating system whereas CUDA codes can only be executed in a NVIDIA hardware.

At present NVIDIA technologies are stronger in the market than OpenCL. NVIDIA has provided several resources to developers such as toolkits, libraries, etc. that make the integration of CUDA to the programming environment more efficient. Whereas the resources of OpenCL are quite limited compared to the existing CUDA resources.

CUDA with NVIDIA GPUs are reaching different sectors due to its high performance. Areas such as computational finance, climate modeling, data science, deep learning and machine learning, defense and intelligence, medical imaging are few of the significant sectors where notable performance has achieved due to GPU usages.

Numba is a very popular library of Python that enables developers to accelerate their code and provide features to exploit GPU architecture for parallel processing. A sample code using Numba library's CUDA feature is shown in in the Section B.1 of the Appendix.

### 2.1.10 Aparapi Framework

Aparapi ("A PARallel API") is a parallel API build for Java language, which provides Java binding to OpenCL. The developers of Aparapi provide a high-level API to translate parallel works in Java without being concerned with GPU implementation details required in the backend. However, it requires having some basic notions of GPU hardware in order to gain performance with this API, as well as there is no need to know OpenCL language. This API will translate the Java bytecode to OpenCL kernel dynamically at runtime that will run on GPU. Aparapi's source code was released with a GPL license and supported by AMD. A sample Aparapi Code for summation of two arrays in given in n in the Section B.2 of the Appendix. There are several studies that insights the performance enhancements achieved using Aparapi [10, 11].

### 2.1.11 Tornado VM

Tornado VM [12] is a virtual machine for Java applications that exploits high-performance heterogeneous hardware. It is a programming and execution framework that enhances java based applications performance using multi-core CPUs, GPUs and FPGAs. It is a backend platform for OpenCL and extends Graal JIT compiler. It allows developers to increase Java applications performance while decrementing the consumption of energy. The idea of Tornado VM was opted to find solutions of contemporary development problems and started as a research project of University of Manchester. It allows Java developers execute their applications on heterogeneous hardware without the necessity of having knowledge of parallel computing or heterogeneous programming models. It is an open-source framework intended to support developers to accelerate their Java programs and get the benefit of GPU hardware. The overall execution mode is shown in Figure 2.8 and a sample code of matrix multiplication is shown in in the Section B.3 of the Appendix.

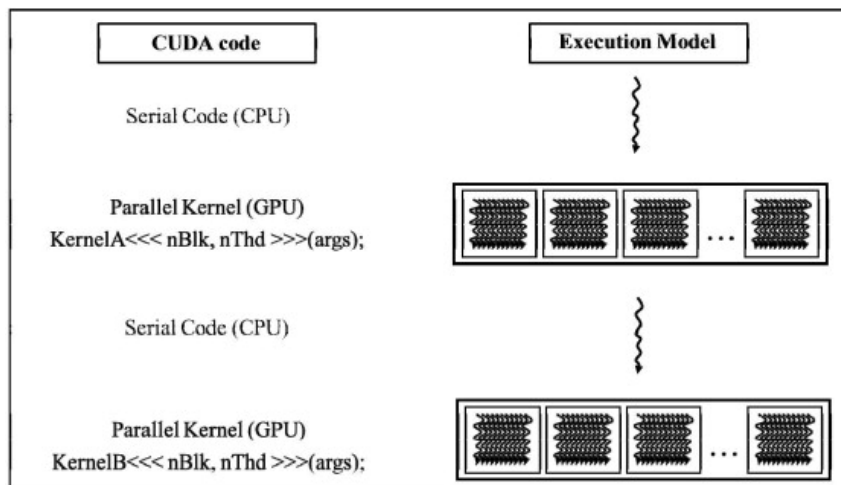


Figure 2.7: CUDA Execution Mode

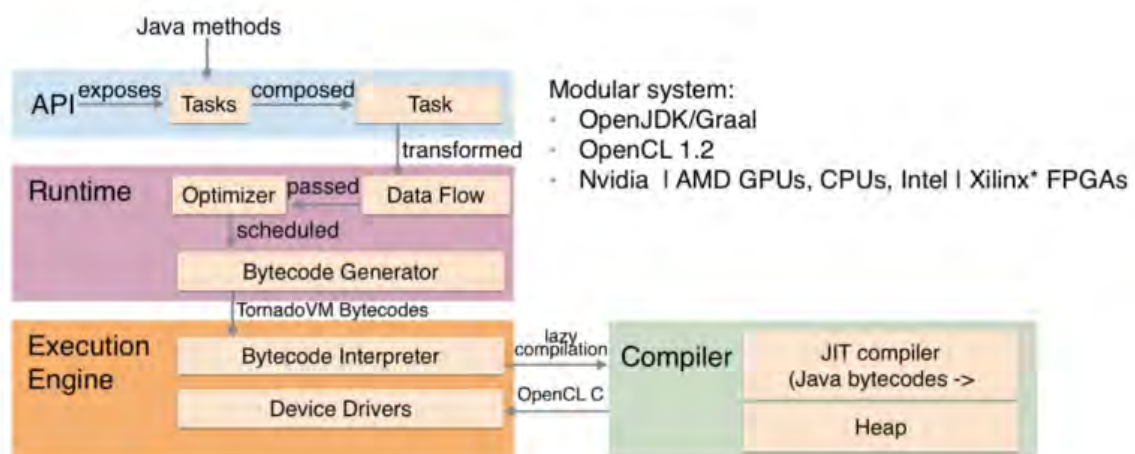


Figure 2.8: Tornado Execution

### 2.1.12 JOCL

JOCL is an API that provides Java binding for OpenCL that enables developers to execute Java applications in heterogeneous platforms containing multi-core CPUs or GPUs. It is quite similar to the original OpenCL API. A sample JOCL code for summation of two arrays is shown in the Section B.2 of the Appendix, where the syntax has similarity with C language as JOCL is just the binding to OpenCL and do not provide high level language implementation like Aparapi or Tornado VM.

## 2.2 Related Works

Parallel search approaches for combinatorial optimization problems have been thoroughly studied in [13]. In this research, different algorithms such as genetic algorithms, simulated annealing, tabu search, and greedy randomized adaptive search procedures have been explored. Parallel approaches as well as new heuristics for combinatorial optimization problems have been shown that can be implemented in multi-core processors, which will reduce the computational time of NP-hard problems.

Heuristic search algorithms have been emerging over recent years and the application of these searches has established new areas [14]. However, some associated issues need to be dealt with in order to expand and re-expand the states. The first problem is that the heuristics must be admissible and consistent to minimize the expanded states. The second issue is that the tree search algorithms grow exponentially with increasing search depth. Concerned with these issues, an improvement framework called IBEX has been proposed in [15], which can tackle both issues and has shown competent outcomes for A\* search. There have been other works in order to solve memory issues of A\* search and proving the inconsistency of heuristic functions as well as a proposed approach of a memory bounded A\* search algorithm, which shows significant improvement [16, 17].

A\* search has been implemented using GPU and the results have shown a significant improvement. The parallelization of this technique is achieved by using parallel queues and for storing the states, the Cuckoo Hashing scheme has been applied. In this study, they have shown the comparison of the sequential and parallel versions of A\* search. In addition, they have covered different areas of search such as path finding or protein design [18]. Moreover, there are different techniques to parallelize A\* search queues which have been studied in [19]. Recursive Best First Search (RBFS) has focused on in recent years. There are several works related to RBFS. The memory limitation of the Best-first search can be solved using RBFS and there are few different approaches related to it. Using an AND/OR search, the performance of the traditional RBFS can be improved [20]. Recursive Best First AND/OR Search with Overestimation (RBFAOO) defines such a possible combination in order to improve the sequential

RBFS approach. In these algorithms, the re-expansion of internal states has been improved using a caching scheme to avoid expanding entirely previously visited states instead of reuse partially visited states' information.

RBFS with Controlled Re-expansion is another approach to control the regeneration of the state of RBFS by using a histogram to store the currently explored path for every state, which is also propagated to the parent states while backtracking [21]. Searching for the nearest neighbor in a query set is one of the major concerns for data analytic studies due to the large-scale data present in the query set. Therefore, a massive parallel calculation may accelerate this process. Having this concern, in [22] some researchers have studied the use of GPUs for finding nearest neighbor queries. Their approach consists of combining k-d trees and GPUs for searching the nearest neighbor; the basic idea is to process the same leaf queries in batches. The results of this study have shown improvement in computational time besides their proposed methodology can be applied for different type tree structures as well. Using heap structure for k-nearest neighbor (kNN) search has been also studied in [23] by using both CPU and GPU architectures which have shown significant performance. Another study on kNN using the brute-force approach has been shown in [24].

Uninformed searches also benefit from using parallel techniques. The scalability of random graphs with more than one billion edges and vertices has been shown using parallel distributed breadth-first search (BFS). This approach uses 2D (edge) partitioning of the graph and has been tested in IBM BlueGene/L, which has demonstrated extensibility in very large-scale random graphs [25, 26].

Constraints Optimization Problems using Dynamic Programming exploring massive in a parallel manner have been studied in paper [27]. In this study, the constraints optimization problems have been solved by combining Bucket Elimination and Distributed Constraint Optimization Problems to achieve parallelism such that the use of GPU resources can be maximized. The experimental results have shown that using GPUs shows notable advantages in computational time. Another constraint optimization problem, Large Neighborhood Search using GPUs has been studied in [28]. In this study, both CPU and GPU versions of four problems (the transportation problem, the traveling salesman problem, the knapsack problem, and the coins grid problem) have been thoroughly explored, which have shown a notable speed up in the computational time while increasing the input size of the problem.

In paper [29], the researchers have compared CPU implementation with GPU implementation of traveling salesman problem using 2-opt and 3-opt local search heuristics. Their study shows a notable increment in the computation of the searching problem using GPU resources. Also, it demonstrated a different level of parallelization techniques in order to maximize the use of GPU resources.

Genetic Programming is another area where using GPUs has shown great results. Many genetic programming applications can use GPUs in order to solve real-life problems [30, 31]. The

crossover, mutation, and fitness calculation will be running in a parallel manner where the selection process will be the main executor, which will synchronize the rest of the running parallel executions. This scheme is known as the Master-slave model, where the selection process is the master process, and the rest are slave processes.

Game theory is another crucial area of AI, which mainly focuses on reinforcement learning in multi-agent systems. Parallel approaches can be applied in game theory as well. Alpha-beta algorithm is one of the oldest known AI algorithms used in search game trees. There have been various attempts to speed up the searching process of the Alpha-beta pruning algorithm, and it has shown that parallel approaches are more beneficial in strongly ordered trees. Using Principal Variation (PV) – splitting algorithm with Alpha-beta pruning has shown notable efficiency in searching among all the parallel versions [32].

There are many other applications of GPU in varieties of fields, such as a novel approach of Concurrent Constraint Programming on GPU has been shown in [33], which exploits the parallelism offered by GPUs for concurrent programming, which was principally designed for sequential computation. Moreover, IoT (Internet of Things) devices have also shown a performance gain using GPUs. In paper, [34], a solution for cryptographic operations for secure communications and authentications between edge computing nodes on embedded GPU devices has shown notable performance.

Another notable GPU computation achievement on domain propagation has been shown in the paper [35]. The proposed GPU-based algorithm for mixed-integer programming (MIP) problem for sparse matrices demonstrated significant performance results.

# Chapter 3

## Proposed Model

This section presents the proposed model, methodologies, and parallel algorithm used in this research work. In addition, some implementation details along with some properties of the parallel algorithm have been described.

### 3.1 Problem Formulation

To formulate the problem, we assume that the RBFS algorithm will have a source state  $S$  from which a goal state  $G$  needs to be reached. It can have multiple ways to reach the goal state but there will be only one optimal path. There will always be multiple states from where to choose to continue the search, however the selected one will always have the minimum fitness value possible. The fitness will be based on two different parameters: an estimated value from the current node to the goal state, known as  $h(n)$  and the current traversed path, known as  $g(n)$ , where  $n$  is the corresponding node. The amount of time required for the search depends on the state's displacements; the further the state's values are from its positions of the goal state values, the more time will be needed.

Considering the enormous powerfulness of GPUs, it has been one of the most pioneer technologies for modern science. GPUs are used nowadays to minimize the task load of several algorithms in different sectors. In this research work, the main objective is achieving the increment in the computational speed of the RBFS algorithm using parallel computation through the GPU.

The proposed model is a shared memory based algorithm and it uses multithreading in the parallel approach for non-GPU based architectures. Multithreading is better than multiprocessing because threads have light overhead which runs faster than processes which makes those more efficient. RBFS is an algorithm that relies on the searching path states values and these values needs to be taken in count in order to progress the search, which means it requires a sort of communication using some kind of shared memory. In order to gain performance, it must

ensure that the selected nodes for exploration are not determined without checking the previous explored states. Overall, RBFS is an algorithm that cannot run independently without having the knowledge of the previous states; it cannot choose the next one and requires knowledge from the environment for successful completion. Therefore, without independent parts a distributed algorithm will not be a great approach to consider.

## 3.2 System Model

The flow chart of the proposed system is shown in Figure 3.1. The proposed model is designed in such a way that the computational speed of the RBFS algorithm can be increased. In addition, it proposes a parallel approach of designing a sequential algorithm.

At the beginning, the RBFS function is called with a particular source state, goal state and a bound value. Subsequently, children of the source state will be generated, whose fitness value will be calculated using parallel processing. That is, different threads will be generated according to the number of branches a particular source state has (i.e., considering each child node as a new source state of the problem to reach the goal state).

Consequently, median value will be calculated based on the fitness values of the children as well as duplicate nodes will be removed from the children list. Among all the children, those who are most fitted according to the median value will be selected for being the new source and the RBFS algorithm will be called again in a parallel way using most fitted children as sources.

## 3.3 Methodology

This section outlines the methodology to design the proposed parallel RBFS has been explained in detail. Overall, the proposed parallel RBFS consists of four essential parts:

- Parallel Heuristic Calculation
- Duplicate State Pruning
- Choosing Best-Fitted States
- Parallel RBFS Calling

### 3.3.1 Parallel Computation of the Heuristics Functions

Calculating the heuristic functions is sometimes expensive in different applications and becomes the overall barrier for the algorithm performance. Independent calculation of the heuristic functions can overcome this issue. Therefore, for parallelizing the RBFS algorithm and being



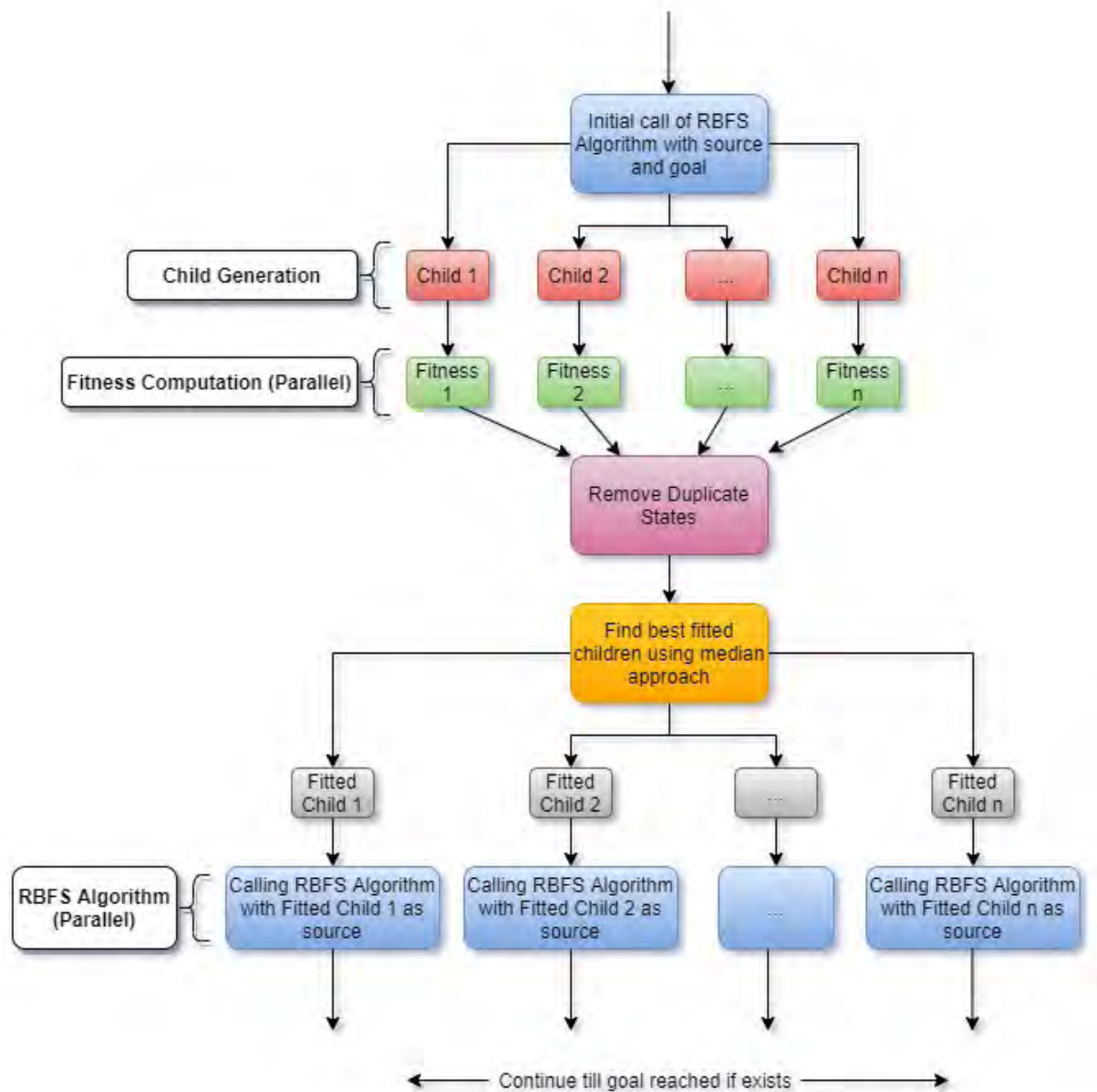


Figure 3.1: Flow Diagram of the Proposed System Model.

able to use GPU resources, the first step to consider is the parallel calculation of the heuristic function. As the calculation of the heuristic function is mutually independent for each state of the problem, its parallelization is unequivocal. In Figure 3.2 shows the fitness calculation of a source state having three child states in which their heuristic values are calculated in parallel manner.

### 3.3.2 Pruning Duplicate States

In the parallel RBFS algorithm, it is possible that it will try to visit the same state several times. If the duplicate state does not have better fitness value than the backup value then it is not worthy



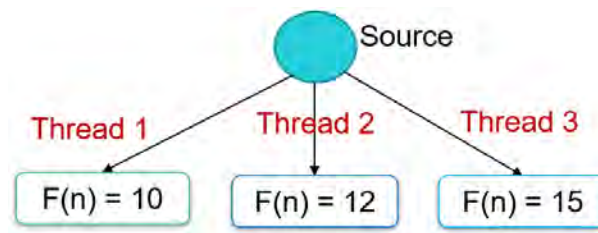


Figure 3.2: Parallel Fitness Calculation

to expand the state. To avoid the expansion of already explored states, these are pruned in such a way that the algorithm can speed up its computation.

Depending on different applications of the algorithm, the node duplication techniques might vary. However, in RBFS algorithm all the explored nodes are not necessarily kept in memory like A\* search instead of that only the next best state fitness value is remembered as backup value, which is passed along the explored tree path and unwinds to the backup state if the limit condition breaks.

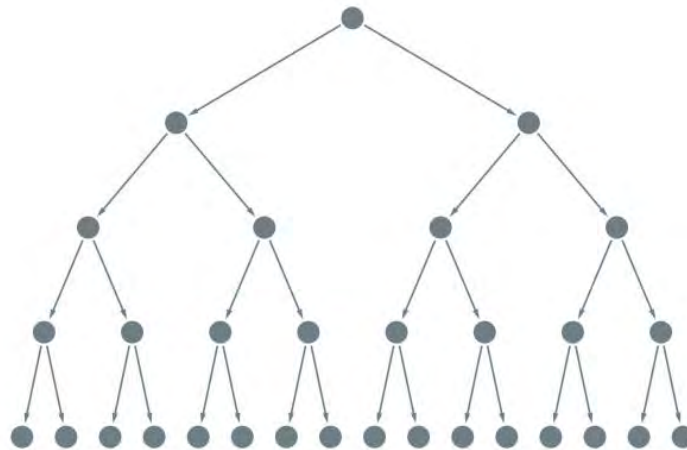


Figure 3.3: Exponential Search Space

Running the RBFS in a parallel way in GPU is required to call the RBFS function several times which may lead to try to expand the same state more than once, that is, leading to unnecessary expansions. Therefore, it is necessary to identify the states already explored, which can be easily achieved by storing the states in a data structure. However, for problems with exponential search space shown in Figure 3.3 which will not be feasible to have a pre-allocated structure.

Node duplication technique requires having a data structure that will support both insert, search and delete operations. The insert operation appends a state to the data structure. The search operation finds out if a particular state has been stored or not in the data structure, it returns true if the state exists in the data structures, otherwise returns false. The delete operation simply removes a state from the data structure when its fitness value becomes infinity, which indicates an unreachable state condition.

As the RBFS algorithm does not store all the possible nodes to expand in memory, there is no issue with the memory constraint on this algorithm. The duplicate nodes can be managed simply by using an array list or hash structure. However, handling insert, delete and search operations in GPU for parallel processing can be difficult. Since RBFS algorithm does not suffer of memory constraint like A\* search algorithm, the parallelization of this operation is not needed for the majority of problems. However, if the parallelization is needed, Parallel Cuckoo Hashing can be used [Pagh and Rodler 2001]. This parallelization technique will ensure that all duplicate nodes are detected correctly and can be implemented in GPU even though its implementation is a bit difficult due to the architectures of GPU devices.

In Figure 3.4 few duplicate states pruning are shown. In the first diagram (a), S1 is a duplicate state from S4, therefore, this path is not considered and entire sub-tree after S4 is pruned. In the diagram (b), S9 has been already explored by S2. Hence, S9 will be pruned and it will not be expanded from S7. In the last diagram (c), S4 will be pruned from S5 as it has already been explored from S1.

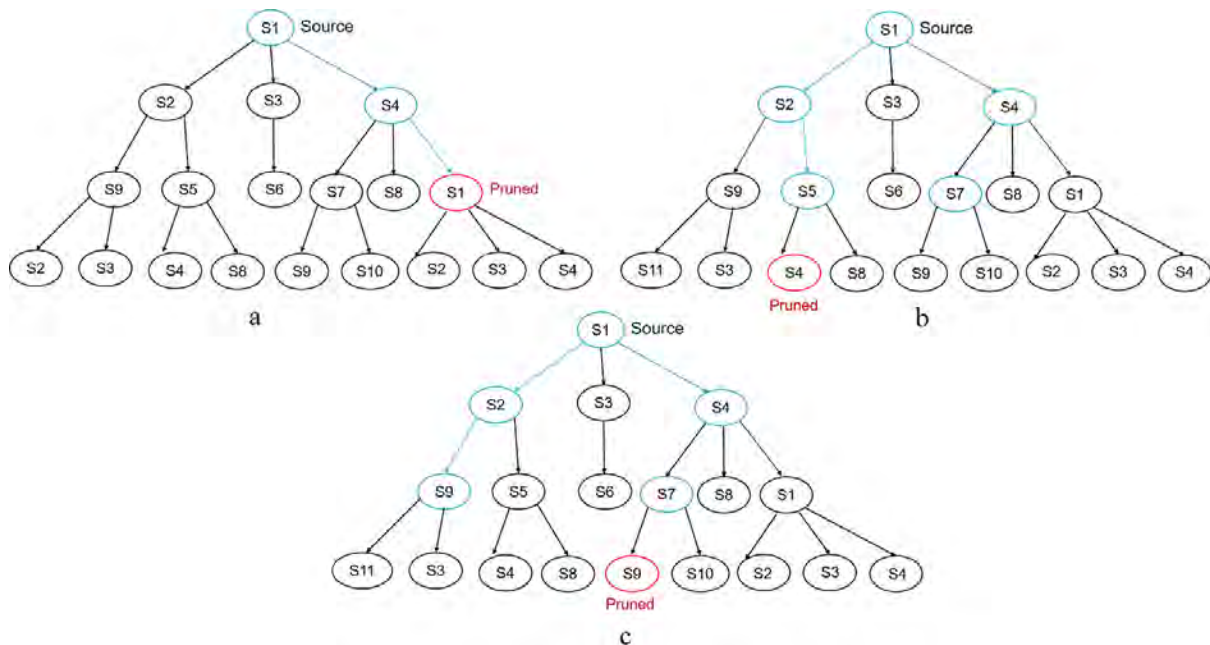


Figure 3.4: Pruning Duplicate States

### 3.3.3 Choosing Best Fitted Child States

Each source state generates several child states each having different paths that can lead to the desired goal state. However, not all the paths are suitable to follow and some may lead to a dead end, which will imply wastage of the resources. Choosing the best state is already difficult and where we have several options then it has become a puzzle for the algorithm. Finding the best ones among all the generated child states is very important, as it will be one of the performance

keys of the algorithm. In the traditional RBFS algorithm, the best state is the state that contains the lowest fitness value among all the child states. In the parallel approach, there is a need to choose more than one child to call RBFS several times concurrently. This approach needs to be handled carefully as it is possible to choose all the states in a parallel way, which may not need to be considered for expansion. There are several reasons that can be considered in terms of not choosing all the child states; some child states can be a misleading path and it cannot reach the goal state as well as some might have higher fitness value than the allowed limit. Moreover, the device resources will be unmanageable for larger search space of a problem in addition to the waste involved in exploring unreachable states.

Focusing on all these constraints, to choose the best-fitted child states median approach has been used. There are several reasons for using the median approach. Firstly, extreme outliers of the data do not affect the median value, thus, the states containing larger fitness values will not be selected automatically. Secondly, the median value ensures that at least one case will remain optimal and that state will lead to the goal state. In addition, it is a robust estimator that cannot lead to wrong selection as it will always provide a sub-optimal set of states.

There is a possibility to choose the best-fitted states using the mean value of the children's fitness values. However, there is a chance that it will choose more states than the median approach as the mean value is highly affected by outliers, thus, it might lead to unnecessary state expansions which might not lead to improve the performance significantly.

Some states have heuristic values that adding to the path values may give smaller fitness values while looking at the current state however in the searching process it may not lead to the goal state, thus, potential states must be also considered for expansion. The potential states among all the states can be chosen by using the median function. The median is calculated in such a way that only the potential states are chosen for expansion. For finding the median value, the fitness values of the children are sorted in ascending order and the median of these values are calculated. The child states that have fitness value less than the median value are considered to be the most fitted child, that is, the potential states to be explored.

Some of the best-fitted states selection are shown in the Figure 3.5. The states are chosen based on their fitness values and median value. If the fitness of the state is less than the median value then it is selected as best fitted state. The median value of the successors of S1 is 13 as only S2 has fitness lower value than the median value, only that particular state has chosen as the fitted one. The median value of the successors of S3 is 15.5, therefore S5, S6 and S7 are chosen as best fitted ones as they have lower value than the median value. Consequently, the median of S7 is 16, so states S11, S12, S15, S18 and S20 are chosen.

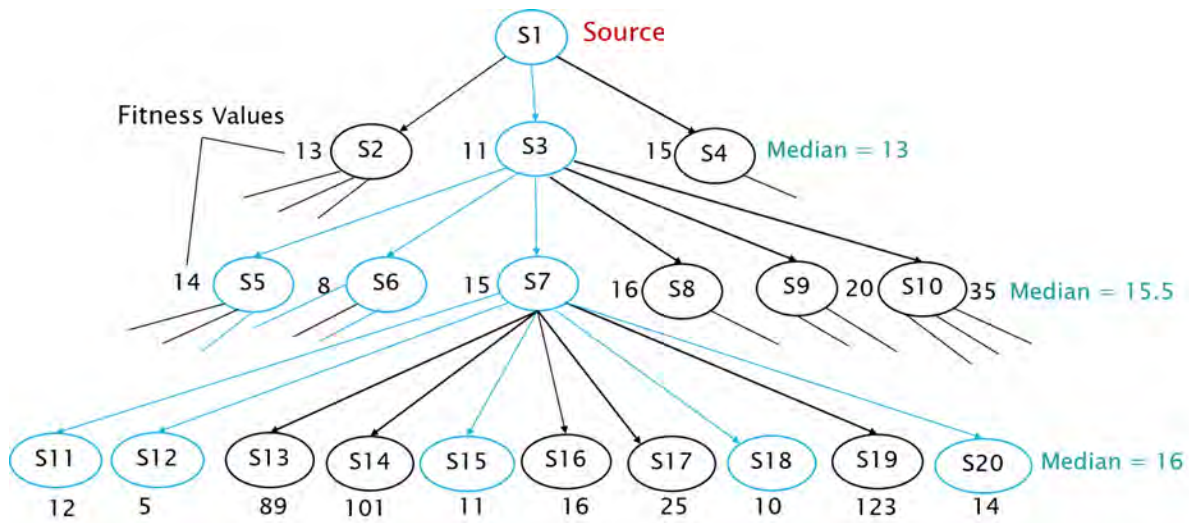


Figure 3.5: Best-fitted states

### 3.3.4 Parallel RBFS Execution

By incorporating parallel fitness calculation, a simple parallel algorithm is achieved, which does not significantly increase the computational speed of the RBFS algorithm. The main issue with the parallel fitness calculation is that it has a limitation as it depends on the outer degree of each state in the search tree. Moreover, in many applications the outer degree of states is quite low, thus fitness calculations, cannot be benefited by the GPU execution. As GPUs contain thousands of cores, by doing simple fitness calculations on GPU, its performance is inefficient. This simple algorithm has not been parallelized fully; there are too many sequential parts on it, which can break down the GPU architecture performance. Sequential operations are highly inefficient on GPU hardware and might perform worse in comparison with CPU executing single threaded applications. In order to increase the degree of parallelism in the RBFS algorithm, multiple calling of the recursive algorithm can be done using different sources in such a way that it will boost up its searching performance. By executing multiple states with different sources will increase the parallelism degree of the fitness calculation as multiple states will concurrently calculate the fitness value of their child states.

However, executing all the possible states may lead to unnecessary expansion creating wastage of the computational resources. Therefore, each time only the most fitted states will be used as new sources for parallel calling of the recursive algorithm. To gain the maximum performance, most promising states are selected according to the median value obtained from all the child states' fitness values. Each state will be treated as a new parallel execution of RBFS independently with limit value infinite as if it was the first call of the algorithm. Few states selection as new sources is shown in Figure 3.6.

In the proposed algorithm (GRBFS), each time multiple child states will be selected for being the new source of execution of the parallel GRBFS, thus, parallelizing the traditional algorithm.

In addition, it increases the number of states expansion, improving the degree of parallelism of fitness calculation of child states. The overall proposed model has been depicted in Figure 3.7, where initial three algorithm calling state has been taken in count to show the degree of parallelism of the model.

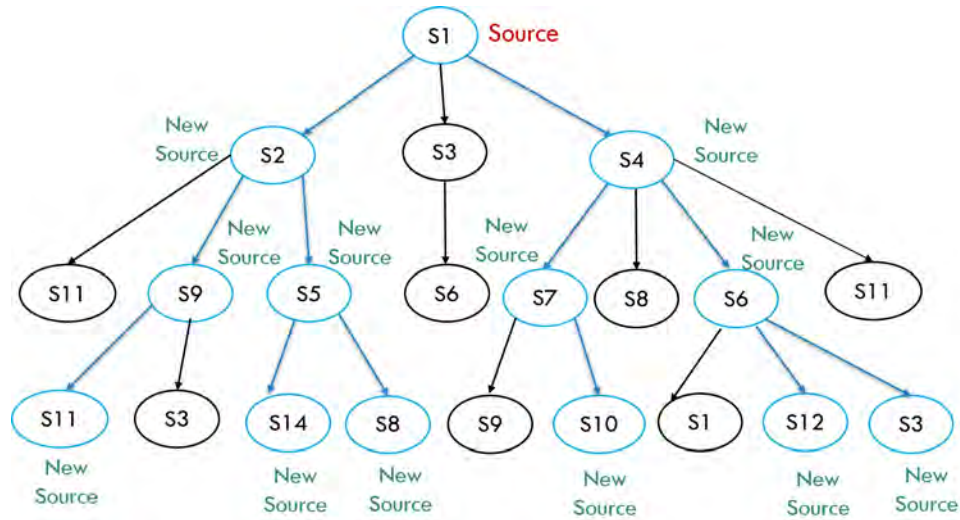


Figure 3.6: Parallel GRBFS sources

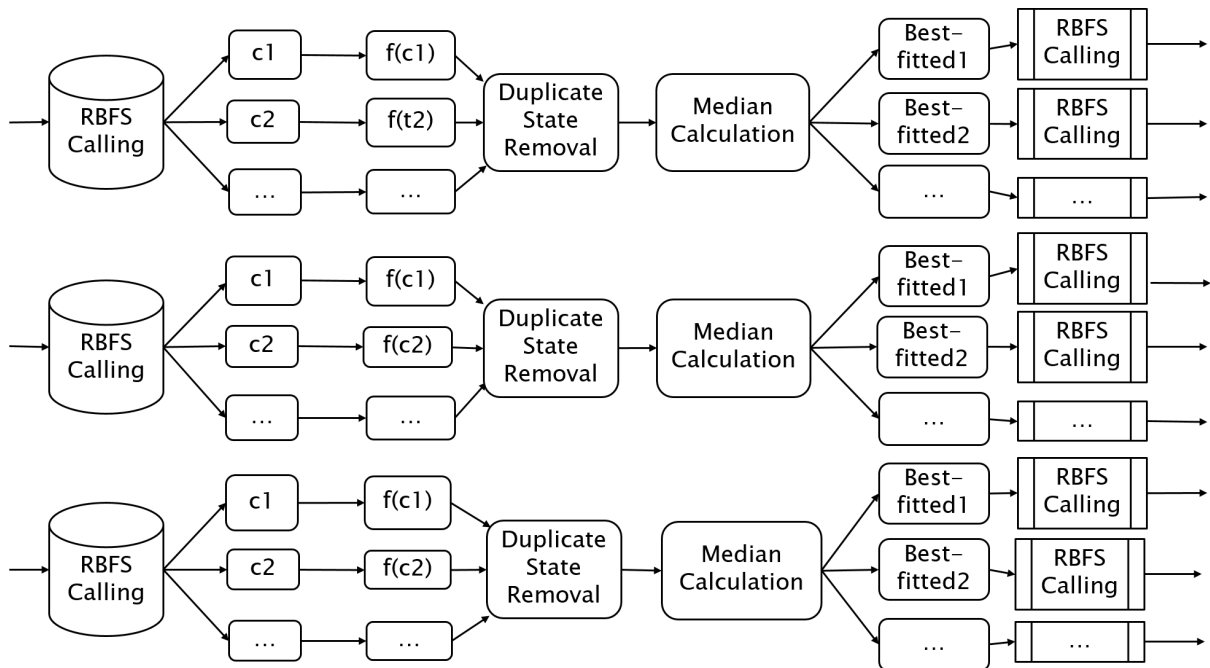


Figure 3.7: Parallel Approach

### 3.3.5 Algorithm: GRBFS

In this section the algorithm of the proposed parallel RBFS (GRBFS) model has been described; highlighting the parallel calculations that accelerate the performance of the model over the

traditional sequential RBFS algorithm. The overall space complexity of the algorithm remains the same as the sequential RBFS algorithm and time complexity with the parallel approach will depend on the number of parallel execution with branching factor  $b$  along with the maximum search depth  $d$ . The probabilistic time complexity of the proposed GRBFS is  $O(\log_{\text{number of parallel execution or threads}} b^d)$

**Algorithm 1** Parallel RBFS on GPU

---

```

1: Let tree be the data container containing the created states
2: Let vs be the data container containing the expanded states
3: procedure GRBFS( $s, g, b$ )      ▷ Find the shortest path from  $s$  to  $g$  with  $b$  bound value
4:   if  $s = g$  then return the path (solution)
5:   end if
6:   add source to the tree if it doesn't exist already
7:   create child states from the source state
8:   for each child of the source do
9:     add the child to the tree if it doesn't exist already
10:    if child already exists then
11:      Ignore
12:    end if
13:  end for
14:  Let  $F$  be the data container containing fitness values
15:  for each child of the source do
16:    calculate fitness value                                     ▷ Call in parallel
17:    store the values in  $F$ 
18:  end for
19:  median  $\leftarrow$  median value among all the fitness values
20:  for each child  $i$  of the source do
21:    if fitness value of the child  $i <$  median then
22:      if the child  $i$  is not in  $vs$  then                       ▷ Checking for duplicate states
23:        add it the  $vs$ 
24:        GRBFS( $i, g, \text{infinity}$ )                               ▷ Call in parallel (source is the child)
25:      end if
26:    end if
27:  end for
28:  loop
29:    if children size is  $\geq 2$  then
30:       $min \leftarrow$  best state from all the child states
31:       $first \leftarrow$  best child fitness value
32:       $second \leftarrow$  second best child fitness value
33:      if  $first >$  limit then
34:        source fitness value  $\leftarrow first$  return failure
35:      else
36:         $lim \leftarrow$  minimum value between second and limit
37:        GRBFS( $i, g, lim$ )
38:      end if
39:    else if children size is  $= 1$  then
40:      if fitness value of child  $>$  limit then
41:        source fitness value  $\leftarrow$  child fitness value return failure
42:      else
43:        GRBFS(child, goal, limit)
44:      end if
45:    else
46:      source fitness value  $\leftarrow \text{infinity}$  return failure
47:    end if
48:  end loop
49: end procedure

```

---



# Chapter 4

## Simulation Results

In this section, we have analyzed the proposed method from various aspects to evaluate its performance. At first, all the problem domain is defined along with the required constraints. Then, the simulation process is described. At last, the evaluation of the results are analyzed.

### 4.1 Sliding Puzzle

Sliding puzzle or N-puzzle problem is one of the classical problems that has been used to test search techniques. It has been used mostly to test the performance of heuristic search algorithms. There have been several researches focused on the evaluation of the sliding puzzle performance using traditional sequential algorithms as well as artificial intelligence algorithms such as breadth-first search and A\* search. The number of states in an N-puzzle is equal to the factorial of the number of tiles. Sliding puzzle consists of a square board with numbers in tiles. There are different types of N-puzzles: 8-puzzle, which consists of 3x3-board size with 1 to 8 numbers, 15-puzzle that consists of 4x4-board size with 1 to 15 numbers, and 24-puzzle, which consists of 5x5-board size. The board also contains a blank tile for sliding the tiles. In this puzzle, a tile is slid to the blank tile position at each step until it reaches to the goal state. The possible movements for the tiles are UP, DOWN, RIGHT and LEFT, that is, it can be moved horizontally or vertically to the blank tile position but not diagonally. Solving these puzzles manually is quite challenging and its state space is large. For 8-puzzles the state space is more than  $10^5$  nodes, for 15-puzzles is about  $10^{13}$  nodes and lastly, for 24-puzzles is about  $10^{25}$  nodes. The larger the board size turns into, the solution of the puzzle becomes more complicated and challenging. Figure 4.1 shows an arbitrary 8-puzzle instance and Figure 4.4 shows an arbitrary 8-puzzle instance solution, which can be found in 3 steps using an heuristic search algorithm. Figure 4.2 shows an arbitrary 15-puzzle problem instance with its corresponding goal state.

There are several heuristics used for solving N-puzzles, however the performance of the solution highly depends on the heuristic applied. The admissibility of the heuristic must also be accounted.



1	6	2
5	7	3
	4	8

Initial State

1	2	3
4	5	6
7	8	

Goal State

Figure 4.1: 8-Puzzle State

5	1	2	4
6	10	3	7
	14	12	8
9	13	11	15

Initial State

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal State

Figure 4.2: 15-Puzzle State

A heuristic function  $h(n)$  is admissible only if  $h(n)$  is never larger than  $h^*(n)$ , particularly  $h(n)$  is always less or equal to true minimum cost from any node  $n$  to the goal state.

Most of the implementations of sliding puzzles use misplaced tiles or hamming distance (count the number of tiles that are not in the correct position) as heuristic given in Equation 4.1. However, this heuristic failed to solve the puzzle in a reasonable amount of time, as it's execution is significantly slow.

$$h1(n) = \text{number of misplaced tiles} \quad (4.1)$$

The most sophisticated heuristic is Manhattan Distance (MD), which is the sum of the vertical and horizontal distance in absolute terms between the current tile to the goal state position given in Equation 4.2. This heuristic is far better than misplaced tiles because it provides greater execution speed as it can choose the states more accurately and in addition to that state exploration rate is less than hamming distance.

$$h2(n) = \text{abs}(x_{\text{value}} - x_{\text{goal}}) + \text{abs}(y_{\text{value}} - y_{\text{goal}}) \quad (4.2)$$

$$\mathcal{LC} = \begin{cases} 1, & \text{linear conflict exists} \\ 0, & \text{otherwise} \end{cases} \quad (4.3)$$

	2	1
7	4	5
6	3	8

	1	2
3	4	5
6	7	8

Misplaced Tiles = 4  
Manhattan Distance = 6  
Linear Conflict = 8  
Optimal Solution = 22 steps

Goal State

Figure 4.3: 8-Puzzle Heuristics

$$f_i = \sum_{i=1}^{N-1} MD(t_i) + 2 \times LC(t_i) \quad (4.4)$$

Even though Manhattan Distance heuristic is the most commonly used due to its admissibility. However, to gain speed-up in the searching performance it is combined with another permissible heuristic known as linear conflict heuristic [36]. This heuristic considers that if two tiles are misplaced in the same row and their positions in the goal state is the same row, it will add two moves to the Manhattan Distance between these two tiles; the function is given in Equation 4.3 and 4.4 where  $N$  refers to the size of the puzzle and  $t_i$  is  $i^{\text{th}}$  tile. An arbitrary 8-puzzle heuristic calculation according to given goal state is shown in Figure 4.3.

## 4.2 Experimental Setup

The RBFS algorithm has been evaluated in 20 different instances of 8-puzzle using Manhattan Distance Heuristic and 13 different instances of 15-puzzle using Manhattan Distance with Linear Conflict Heuristics in three different approaches: sequential approach, parallel approach without GPU and parallel approach with GPU for their performance shown in the simulation results section. The puzzle instances have been collected from several research works [37, 38] and some online puzzle solvers.

The problem has been solved using two different languages: Java and Python. For parallel approach concurrent threads have been used in both languages. For GPU implementation, in Java Aparapi framework has been considered and for python implementation CUDA programming with JIT has been used from the Numba library.

Simulations of Java were tested on Intel Core i7 with 16 GB RAM NVIDIA GeForce GT 710 with 10 GB RAM 192 CUDA cores has been used.

Python simulations were executed on Intel(R) Xeon(R) CPU @ 2.20GHz with 13 GB RAM and NVIDIA Tesla K80 GPU with 13 GB RAM and 4992 cores has been used.

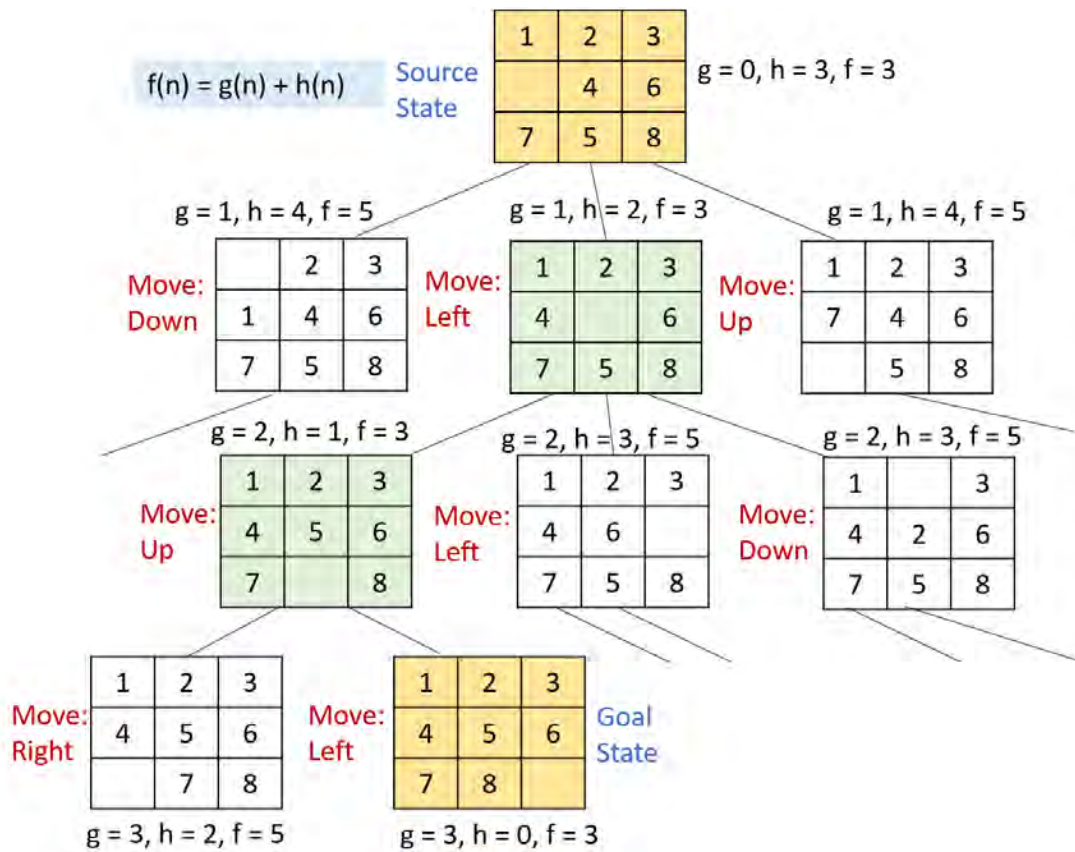


Figure 4.4: 8-Puzzle Solution

## 4.3 Experimental Results

In this section, we have analyzed the proposed method in the sliding puzzle using three different implementations: sequential or traditional RBFS, parallel RBFS without GPU and parallel RBFS with GPU and compare their performances.

In addition, GPU performance has been evaluated using two different approaches: CUDA for python based implementation and Aparapi framework for Java based implementation.

In these experiments, GPU architecture has been fully exploited due to multiple degrees of parallelism of the designed algorithm.

### 4.3.1 Sequential RBFS Algorithm Performance

Table 4.1 and Table 4.2 show the time and memory required for the traditional recursive best first search to reach the goal state from the given source state for 8-puzzle and 15-puzzle respectively. The time is shown in milliseconds (ms) and memory details are given in Megabytes (MB). In addition, the specific source and goal states are given along with the optimal steps required to solve these sliding puzzles. For all 15-puzzle instances the goal state used is given in the Figure 4.2.

Table 4.1: Sequential 8 Puzzle RBFS results

Sl.	Source State	Goal State	Number of Steps	Time (ms)	Memory Required (MB)
1	123046758	123456780	3	6.59	815
2	283104765	123804765	4	5.84	789
3	283164705	123804765	5	7.25	808
4	134862705	123804765	5	10.92	780
5	134805726	123804765	6	4.49	779
6	281043765	123804765	9	37.12	821
7	162573048	123456780	10	25.66	816
8	035428617	012345678	10	21.35	778
9	281463075	123804765	12	51.54	794
10	328451670	012345678	12	142.34	788
11	231708654	123804765	14	332.13	820
12	641302758	012345678	14	1245.30	788
13	725310648	012345678	15	854.41	818
14	231804765	123804765	16	3843.50	815
15	412087635	123456780	17	1526.34	816
16	102754863	012345678	23	212877.43	824
17	876105234	012345678	28	1196691.58	827
18	567408321	123804765	30	44543.67	832
19	806547231	012345678	30	5503295.74	841
20	867254301	123456780	31	4676010.36	860

Table 4.2: Sequential 15 Puzzle RBFS results

Sl.	Source State	Number of Steps	Time (ms)	Memory Required (MB)
1	1 2 3 4 5 6 7 8 0 10 11 12 9 13 14 15	4	6.39	813
2	1 2 3 4 5 10 6 7 9 11 0 8 13 14 15 12	6	24.7	823
3	1 3 4 8 6 2 7 0 5 10 11 12 9 13 14 15	10	87.37	812
4	1 3 4 8 6 2 0 7 5 10 11 12 9 13 14 15	11	165.51	811
5	1 3 4 8 6 0 2 7 5 10 11 12 9 13 14 15	12	227.97	814
6	1 3 4 8 6 10 2 7 5 0 11 12 9 13 14 15	13	263.88	813
7	5 1 2 4 6 10 3 7 0 14 12 8 9 13 11 15	14	263.72	813
8	1 7 2 0 9 5 3 4 6 10 12 8 13 14 11 15	15	510.87	813
9	0 2 11 3 1 6 7 4 5 9 12 8 13 10 14 15	16	621.69	813
10	1 2 6 0 5 10 4 3 14 7 11 8 9 13 15 12	17	1292.66	821
11	1 6 2 3 5 12 7 4 10 13 15 8 9 0 14 11	20	1038.96	817
12	5 1 4 8 2 6 7 3 13 9 11 0 10 14 15 12	21	143796.1	823
13	5 1 0 4 7 3 2 8 9 15 14 11 6 13 10 12	22	197998.38	822

Table 4.3: Parallel 8 Puzzle RBFS without GPU

Sl.	Source State	Goal State	Number of Steps	Time (ms)	Memory Required (MB)
1	123046758	123456780	3	11.13	787
2	283104765	123804765	4	8339.87	812
3	283164705	123804765	5	11.66	811
4	134862705	123804765	5	12.48	796
5	134805726	123804765	6	5.64	795
6	281043765	123804765	9	58.86	811
7	162573048	123456780	10	42.91	787
8	035428617	012345678	10	15.47	795
9	281463075	123804765	12	250.06	794
10	328451670	012345678	12	10585.37	808
11	231708654	123804765	14	508.35	812
12	641302758	012345678	14	4697.18	806
13	725310648	012345678	15	74.97	803
14	231804765	123804765	16	450.5	811
15	412087635	123456780	17	1658.18	790
16	102754863	012345678	23	4215.85	810
17	876105234	012345678	28	4786.15	811
18	567408321	123804765	30	35022.16	810
19	806547231	012345678	30	134470.85	816
20	867254301	123456780	31	7452.98	791

### 4.3.2 Parallel RBFS Algorithm without GPU Performance

The Table 4.3 and Table 4.4 show the time and memory required for the parallel recursive best first search to execute using multiple threads in a CPU platform to reach the goal state from the given source state for 8-puzzle and 15-puzzle respectively. The time is shown in milliseconds (ms) and memory details are given in Megabytes (MB). In addition, the specific source and goal states are given along with the optimal steps required to solve these sliding puzzles. For all 15-puzzle instances the goal state used is given in the Figure 4.2.

### 4.3.3 Parallel RBFS Algorithm with GPU Performance

The Table 4.5 and Table 4.6 show the time and memory required for the parallel recursive best first search to execute using multiple cores in a GPU platform to reach the goal state from the given source state for 8-puzzle and 15-puzzle respectively. The time is shown in milliseconds (ms) and memory details are given in Megabytes (MB). In addition, the specific source and goal states are given along with the optimal steps required to solve these sliding puzzles. For all 15-puzzle instances the goal state used is given in the Figure 4.2.

Table 4.4: Parallel 15 Puzzle RBFS without GPU

Sl.	Source State	Number of Steps	Time (ms)	Memory Required (MB)
1	1 2 3 4 5 6 7 8 0 10 11 12 9 13 14 15	4	478.9	786
2	1 2 3 4 5 10 6 7 9 11 0 8 13 14 15 12	6	647.72	794
3	1 3 4 8 6 2 7 0 5 10 11 12 9 13 14 15	10	178.67	790
4	1 3 4 8 6 2 0 7 5 10 11 12 9 13 14 15	11	835.2	801
5	1 3 4 8 6 0 2 7 5 10 11 12 9 13 14 15	12	2537.12	791
6	1 3 4 8 6 10 2 7 5 0 11 12 9 13 14 15	13	768.94	788
7	5 1 2 4 6 10 3 7 0 14 12 8 9 13 11 15	14	2894.4	789
8	1 7 2 0 9 5 3 4 6 10 12 8 13 14 11 15	15	495.21	795
9	0 2 11 3 1 6 7 4 5 9 12 8 13 10 14 15	16	136.56	787
10	1 2 6 0 5 10 4 3 14 7 11 8 9 13 15 12	17	1291.11	793
11	1 6 2 3 5 12 7 4 10 13 15 8 9 0 14 11	20	330.58	796
12	5 1 4 8 2 6 7 3 13 9 11 0 10 14 15 12	21	25605.76	798
13	5 1 0 4 7 3 2 8 9 15 14 11 6 13 10 12	22	12488.43	794

Table 4.5: Parallel 8 Puzzle RBFS in GPU

Sl.	Source State	Goal State	Number of Steps	Time (ms)	Memory Required (MB)
1	123046758	123456780	3	29.19	969
2	283104765	123804765	4	10970.8	969
3	283164705	123804765	5	43.7	975
4	134862705	123804765	5	49.13	960
5	134805726	123804765	6	26.06	960
6	281043765	123804765	9	158.17	976
7	162573048	123456780	10	124.54	969
8	035428617	012345678	10	58.47	946
9	281463075	123804765	12	553.44	961
10	328451670	012345678	12	13947.53	953
11	231708654	123804765	14	864.97	977
12	641302758	012345678	14	6732.16	953
13	725310648	012345678	15	174.37	965
14	231804765	123804765	16	770	976
15	412087635	123456780	17	2386.98	972
16	102754863	012345678	23	4967.63	972
17	876105234	012345678	28	6103.33	973
18	567408321	123804765	30	36716.5	955
19	806547231	012345678	30	137243.61	974
20	867254301	123456780	31	8750.51	987

Table 4.6: Parallel 15 Puzzle RBFS in GPU

Sl.	Source State	Number of Steps	Time (ms)	Memory Required (MB)
1	1 2 3 4 5 6 7 8 0 10 11 12 9 13 14 15	4	733.1	939
2	1 2 3 4 5 10 6 7 9 11 0 8 13 14 15 12	6	968.36	971
3	1 3 4 8 6 2 7 0 5 10 11 12 9 13 14 15	10	332.86	940
4	1 3 4 8 6 2 0 7 5 10 11 12 9 13 14 15	11	1215.26	938
5	1 3 4 8 6 0 2 7 5 10 11 12 9 13 14 15	12	3550.5	938
6	1 3 4 8 6 10 2 7 5 0 11 12 9 13 14 15	13	1162.67	939
7	5 1 2 4 6 10 3 7 0 14 12 8 9 13 11 15	14	3816.53	937
8	1 7 2 0 9 5 3 4 6 10 12 8 13 14 11 15	15	781.83	937
9	0 2 11 3 1 6 7 4 5 9 12 8 13 10 14 15	16	254.31	940
10	1 2 6 0 5 10 4 3 14 7 11 8 9 13 15 12	17	1846.21	950
11	1 6 2 3 5 12 7 4 10 13 15 8 9 0 14 11	20	552.61	947
12	5 1 4 8 2 6 7 3 13 9 11 0 10 14 15 12	21	28615.11	952
13	5 1 0 4 7 3 2 8 9 15 14 11 6 13 10 12	22	15107.66	970

#### 4.3.4 States Expanded in Sequential and Parallel RBFS Algorithm

In parallel approach, the number of nodes expanded is comparatively lesser than the sequential approach. The most potential states are explored in a parallel manner by selecting several states at the same time among which the most successful states are determined faster in the parallel version; whereas in the sequential approach after successive backtracking it go to the desired states and it requires more time and longer exploration as reaching to the successful state which can lead to the goal state as the selection varies depending on the heuristic function and explored path. Moreover, in a sequential approach only one state is explored at a time whereas in a parallel approach several states are considered simultaneously.

The Table 4.7 and Table 4.8 show the number of nodes expanded by the traditional recursive best first search and parallel recursive best first search to reach the given goal state from a specific source state as well as illustrates the optimal steps required to reach the goal state for 8-puzzle and 15-puzzle instances respectively.

#### 4.3.5 Aparapi Performance on Parallel RBFS

The Table 4.9 shows the time and memory required for the parallel recursive best first search execution in a CPU platform using Java and parallel recursive best first search using Aparapi framework to execute using multiple cores in a GPU platform to reach the goal state from the given source state. The time is shown in milliseconds (ms) and memory details are given in Megabytes. In addition, the specific source and goal states are given along with the optimal steps required to solve these sliding puzzles. Only 8-puzzle instances have been tested in Aparapi due to its slow performance no further testing was continued.

Table 4.7: 8 Puzzle States expansion

Sl.	Source State	Goal State	Number of Steps	Total Nodes (Sequential)	Total Nodes (Parallel)	% improvement
1	123046758	123456780	3	21	21	No improvement
2	283164705	123804765	5	20	26	-30%
3	134862705	123804765	5	19	30	-57.89%
4	134805726	123804765	6	16	16	No improvement
5	281043765	123804765	9	62	83	-33.87%
6	162573048	123456780	10	49	69	-40.82%
7	035428617	012345678	10	45	35	22.22%
8	281463075	123804765	12	87	202	-132.18%
9	231708654	123804765	14	218	279	-27.98%
10	725310648	012345678	15	319	94	70.53%
11	231804765	123804765	16	682	255	62.61%
12	102754863	012345678	23	5325	862	83.81%
13	876105234	012345678	28	13046	913	93%
14	567408321	123804765	30	2879	2401	16.6%
15	806547231	012345678	30	30848	4977	83.87%
16	867254301	123456780	31	30848	1224	96.03%

Table 4.8: 15 Puzzle States expansion

Sl.	Source State	Number of Steps	Total Nodes (Sequential)	Total Nodes (Parallel)	% improvement
1	1 2 3 4 5 6 7 8 0 10 11 12 9 13 14 15	4	25	229	-816%
2	1 2 3 4 5 10 6 7 9 11 0 8 13 14 15 12	6	75	270	-260%
3	1 3 4 8 6 2 7 0 5 10 11 12 9 13 14 15	10	125	123	1.6%
4	1 3 4 8 6 2 0 7 5 10 11 12 9 13 14 15	11	157	328	-108.92%
5	1 3 4 8 6 0 2 7 5 10 11 12 9 13 14 15	12	150	644	-329.33%
6	1 3 4 8 6 10 2 7 5 0 11 12 9 13 14 15	13	166	322	-93.98%
7	5 1 2 4 6 10 3 7 0 14 12 8 9 13 11 15	14	261	697	-167.1%
8	1 7 2 0 9 5 3 4 6 10 12 8 13 14 11 15	15	292	232	20.55%
9	0 2 11 3 1 6 7 4 5 9 12 8 13 10 14 15	16	440	99	77.5%
10	1 2 6 0 5 10 4 3 14 7 11 8 9 13 15 12	17	396	403	-1.77%
11	1 6 2 3 5 12 7 4 10 13 15 8 9 0 14 11	20	410	185	54.88%
12	5 1 4 8 2 6 7 3 13 9 11 0 10 14 15 12	21	6058	1988	67.18%
13	5 1 0 4 7 3 2 8 9 15 14 11 6 13 10 12	22	1994	1392	30.19%

## 4.4 Analysis and Comparison of results

To analyze and compare the effectiveness of the proposed model, it has been tested with state of art from various aspects. At the beginning, we compared the computational time of different sliding puzzles, incrementing gradually the required number of steps to solve those between three different implementations: sequential, parallel in CPU platform and parallel in GPU platform.



Table 4.9: Aparapi Performace

Sl.	Source State	Goal State	Number steps	Parallel		Aparapi	
				Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
1	123046758	123456780	3	3.74	0.6	810.36	3.36
2	283164705	123804765	5	4.56	0.61	831.8	3.36
3	725310648	012345678	15	6.92	0.68	31927	4.06
4	231804765	123804765	16	16.56	0.91	122736	5.46
5	876105234	012345678	28	355.15	2.79	928461.33	14.32
6	867254301	123456780	31	328.94	2.65	1663163.56	13.82

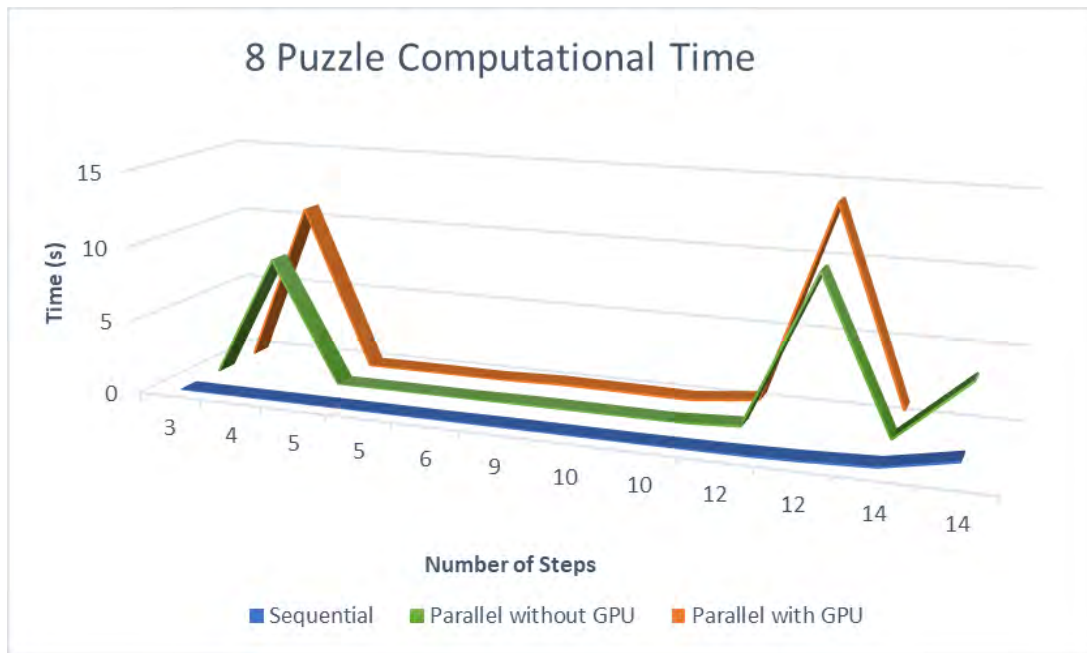


Figure 4.5: Computation Time of 8-Puzzle instances with small optimal steps

Even though, it is impractical to port an algorithm designed for CPU to a GPU platform, as GPU architecture is quite different and operates in a different way than CPU. The parallel approach in CPU will be similar to parallel approach in GPU even though GPUs require some initial setup, which consumes some time; however, for large-scale calculations it will definitely outperform the CPU performance.

The Table 4.10 summarizes the different computational times of the same puzzle instances executing in various platforms. It can be visualized that increasing the number of steps required to solve the puzzle instances, the sequential approach required time is significantly higher than parallel approaches in both CPU and GPU platform even though for small steps the parallel approach do not perform well compare to the sequential approach.

In the Figure 4.5 shows the computational time required for different approaches to solve the 8-puzzle instances whose optimal required steps are quite low, thus the sequential approach is performing a way better than the parallel approaches.

Table 4.10: Computation Time Analysis

Sl.	Puzzle Size	Number of Steps	Sequential Time (ms)	Parallel in CPU Time (ms)	Parallel in GPU Time (ms)	speed-up of parallel in CPU over sequential	speed-up of parallel in GPU over sequential
1	8	3	6.59	11.13	29.19	0.6x	0.21x
2	8	10	25.66	42.91	124.54	0.21x	0.09x
3	8	12	51.54	250.06	553.44	0.21x	0.09x
4	8	14	332.13	508.35	864.97	0.65x	0.38x
5	8	15	854.54	74.97	174.37	11.4x	4.9x
6	8	16	3843.5	450.49	770	8.53x	4.99x
7	8	23	212877.43	4215.85	4967.63	50.49x	42.85x
8	8	28	1196691.58	4786.15	6103.33	250.03x	196.07x
9	8	31	4676010.36	7452.98	8750.51	627.40x	534.37x
10	15	4	6.39	478.9	733.1	0.01x	0.01x
11	15	10	87.37	178.67	332.86	0.49x	0.26x
12	15	11	165.51	835.20	1215.26	0.20x	0.14x
13	15	13	263.88	768.94	1162.67	0.34x	0.23x
14	15	15	510.87	495.21	781.83	1.03x	0.65x
15	15	17	1292.66	1291.11	1846.21	1x	0.7x
16	15	20	1038.96	330.58	552.61	3.14x	1.88x
17	15	21	143796.1	25605.7	28615.11	5.62x	5.03x
18	15	22	197998.38	12488.43	15107.66	15.85x	13.11x

In the Figure 4.6 shows the computational time required for different approaches to solve the 8-puzzle instances which optimal required steps are higher. Hence, the difficulty of the puzzle is increasing. Here, the sequential approach is performing worse than the parallel approaches on GPU and without GPU maintaining a steady computational time.

In the Figure 4.7 shows the computational time required for different approaches to solve the 5-puzzle instances which required steps are increasing. Hence, the difficulty of the puzzle is higher. Here, the sequential approach is performing worse than the parallel approaches on GPU and without GPU maintaining a steady computational time.

GPU devices trade off high performance with the memory requirement whereas CPU focuses on low latency. In Table 4.11 summarizes the different memory requirements of the same puzzle instances executing in various platforms. It can be visualized that by increasing the number of steps required to solve the puzzle instances, the sequential and parallel approach executing in a CPU requires less memory than the GPU platform. GPU requires to copy the entire data to the virtual memory in order to perform the parallel calculation that consumes some time; however, in the long run, it can be seen that CPU memory is increasing while increasing the difficulty level of the puzzles whereas GPU memory remains constant. In the Figure 4.8 and Figure 4.9, it can be visualized the memory requirements of some tested puzzle instances.

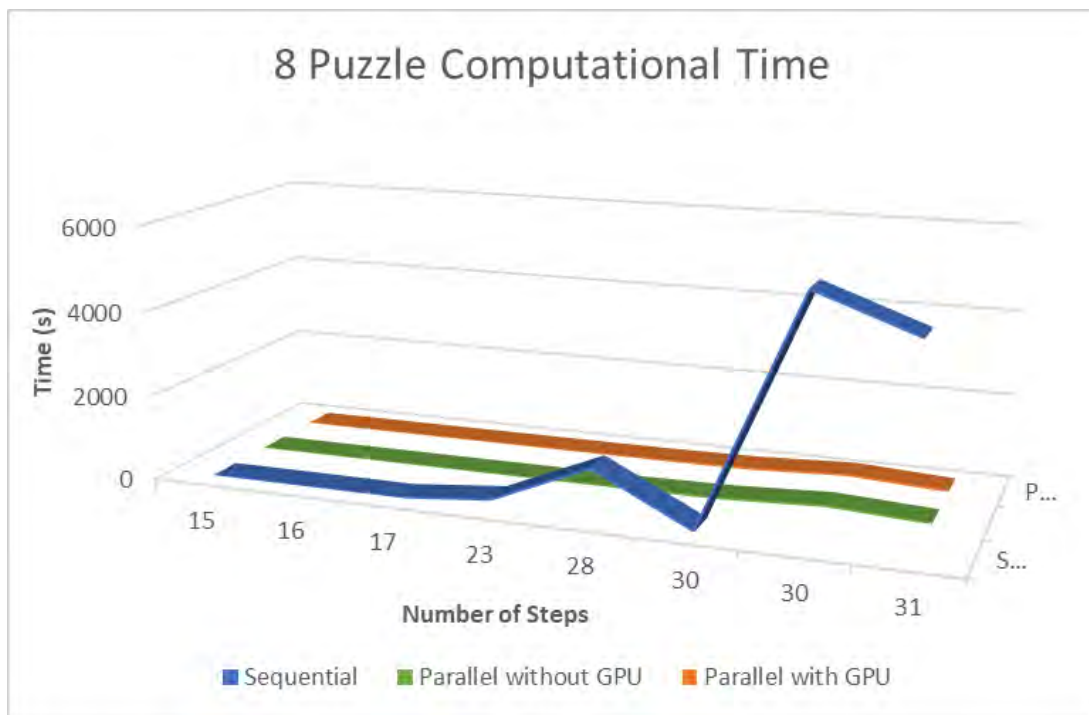


Figure 4.6: Computation Time of 8-Puzzle instances with larger optimal steps

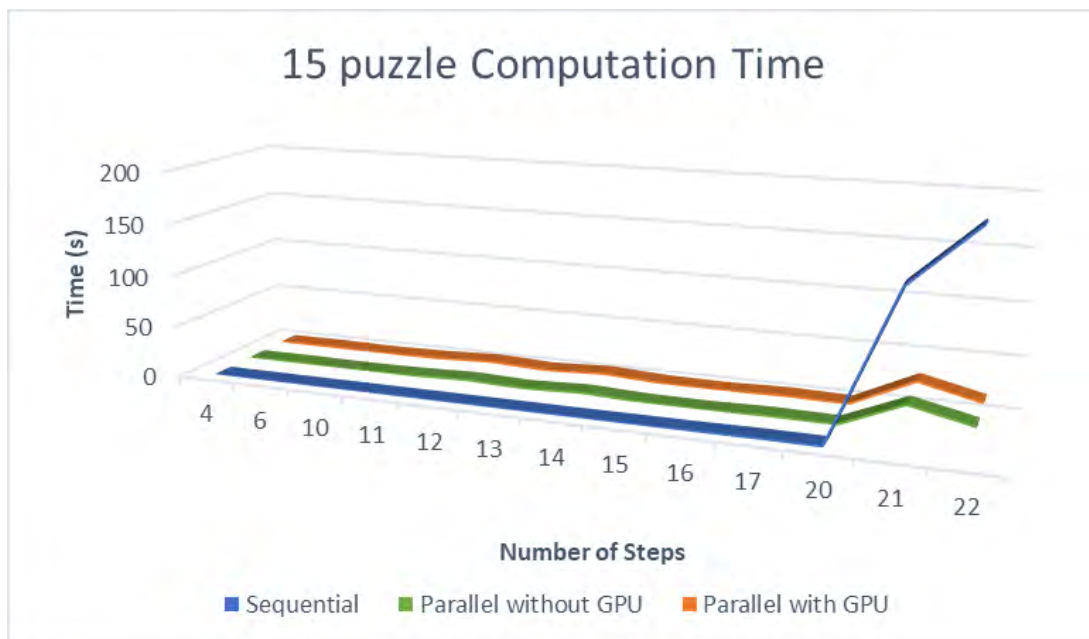


Figure 4.7: Computation Time of 15-Puzzle instances

In Table 4.7 and Table 4.8, the required number of expanded nodes between CPU approach and parallel approach is given. In Figure 4.10 and Figure 4.11, some puzzle instances of node expansion are visualized, it can be seen that the number of expansions are increasing with the difficulty level of the puzzles. The main reason behind is due to the less computational time as the parallel approach reaches the goal states faster than the sequential approach, the unnecessary nodes are not expanded and gradually the expansion rate decreases.

Table 4.11: Memory Requirement of RBFS

Sl.	Puzzle Size	Number of Steps	Sequential Memory (MB)	Parallel in CPU Memory (MB)	Parallel in GPU Memory (MB)
1	8	3	815	787	969
2	8	10	816	787	969
3	8	12	794	794	961
4	8	14	820	812	977
5	8	15	818	803	965
6	8	16	815	811	976
7	8	23	824	810	972
8	8	28	827	811	973
9	8	31	860	791	987
10	15	4	813	786	939
11	15	10	812	790	940
12	15	11	811	801	938
13	15	13	813	788	939
14	15	15	813	795	936
15	15	17	821	793	950
16	15	20	817	796	947
17	15	21	823	798	952
18	15	22	822	794	970

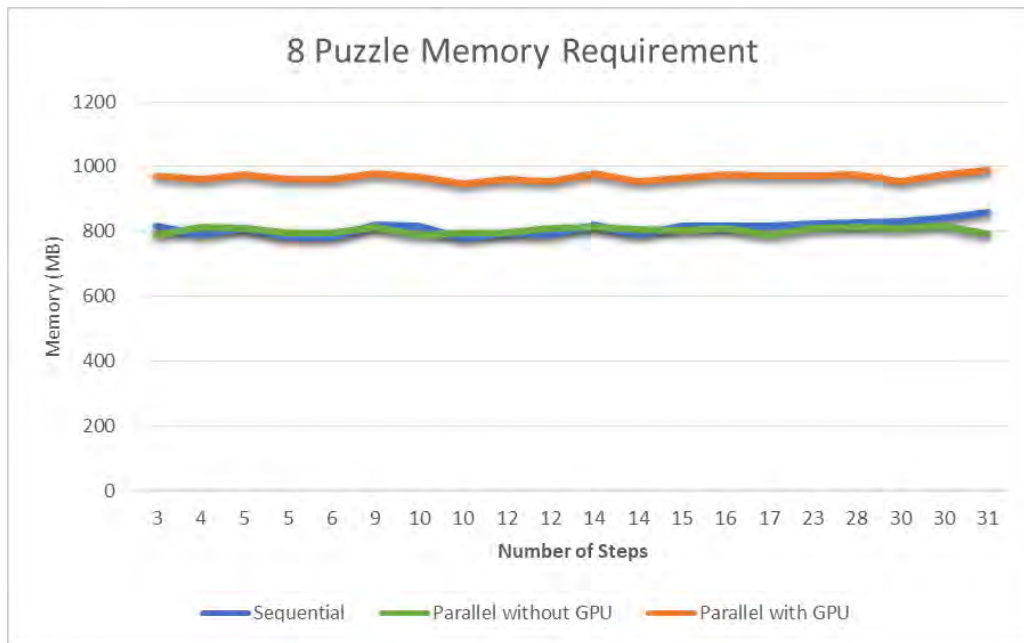


Figure 4.8: Memory requirement of 8-Puzzle instances

Aparapi is a very popular framework that facilitates the execution of Java based applications in GPU without the requirement of the knowledge of heterogeneous model and parallel computing concepts and has been demonstrated to show great results in different studies. Due to its popularity, we chose to evaluate our parallel GPU based algorithm in Aparapi. In Figure 4.12 and

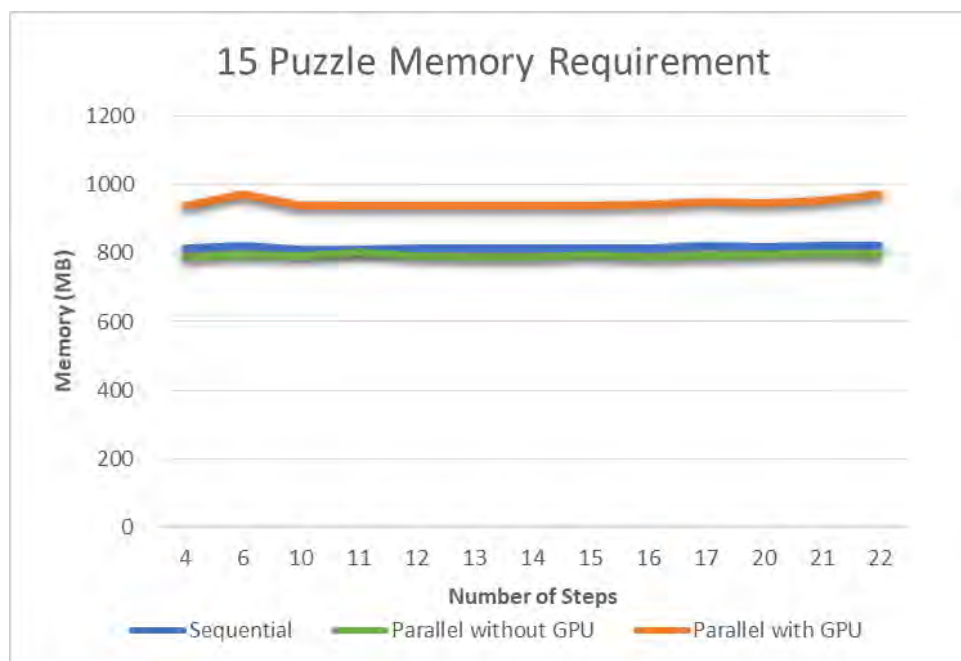


Figure 4.9: Memory requirement of 15-Puzzle instances

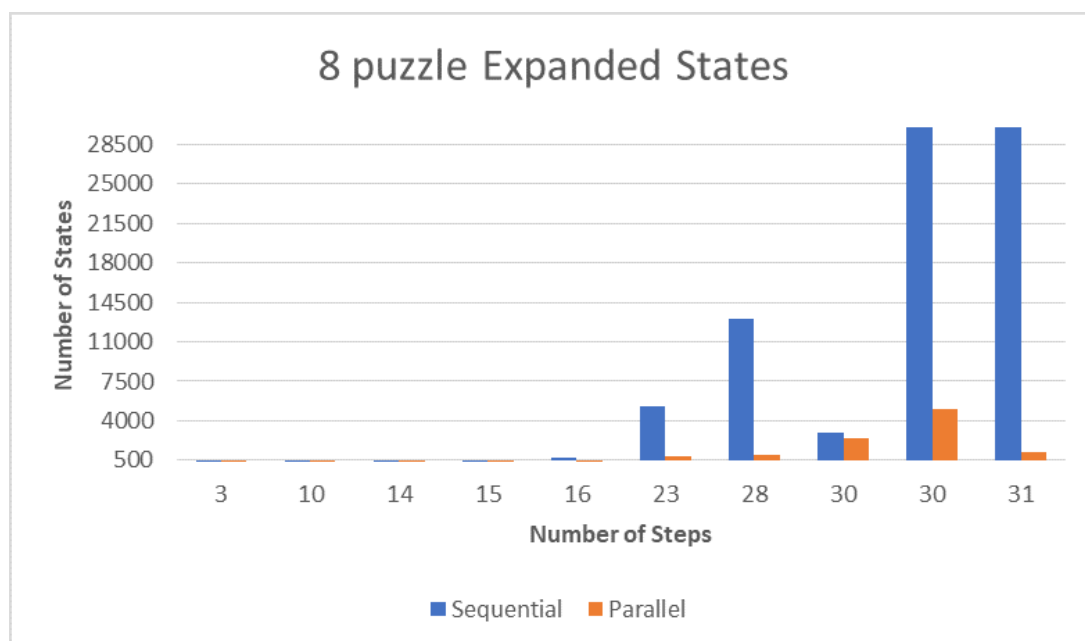


Figure 4.10: 8 Puzzle States expanded in RBFS

Figure 4.13, the comparison between CPU parallel approach and GPU parallel approach using Aparapi framework is shown. Both time and memory constraints using Aparapi are higher than the threaded parallel approach. Increasing the difficulty level of the puzzles, Aparapi performs a lot worse than the threaded parallel approach. The memory requirement of Aparapi is also significantly higher which makes it unsuitable to use for the GPU performance evaluation. This results notably indicate, this framework is not suitable for searching algorithms and does not always perform well for any algorithms even it might perform a way worse.

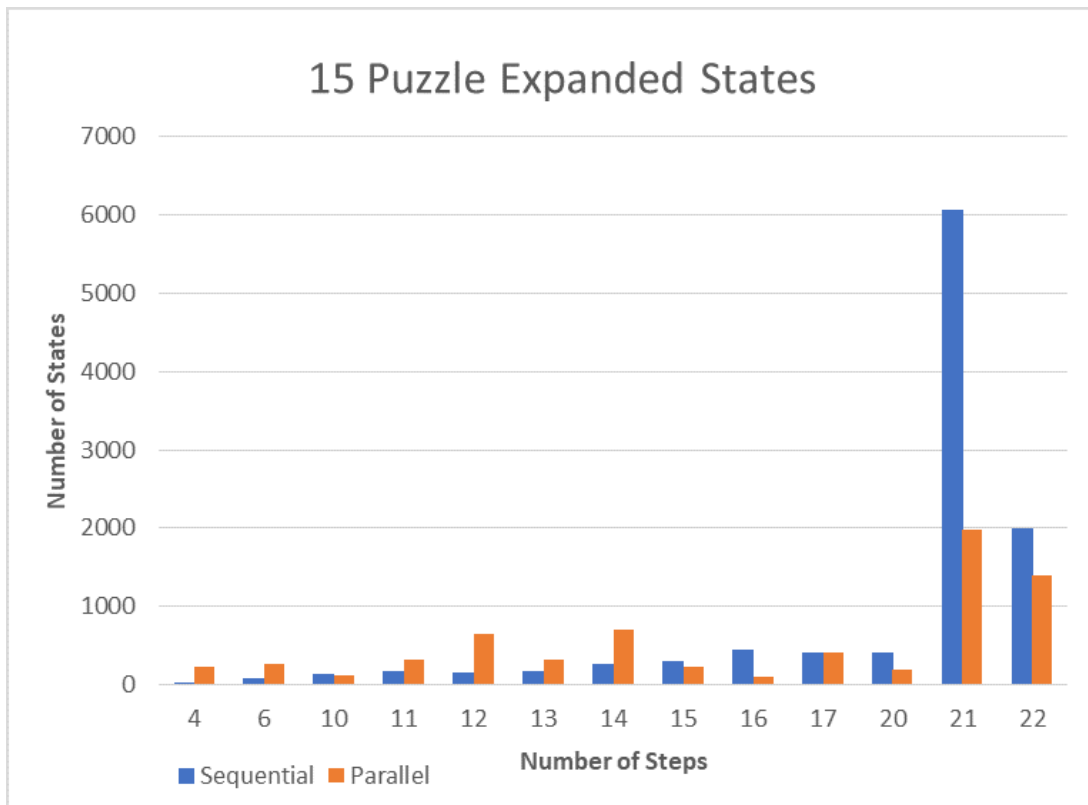


Figure 4.11: 15 Puzzle States expanded in RBFS

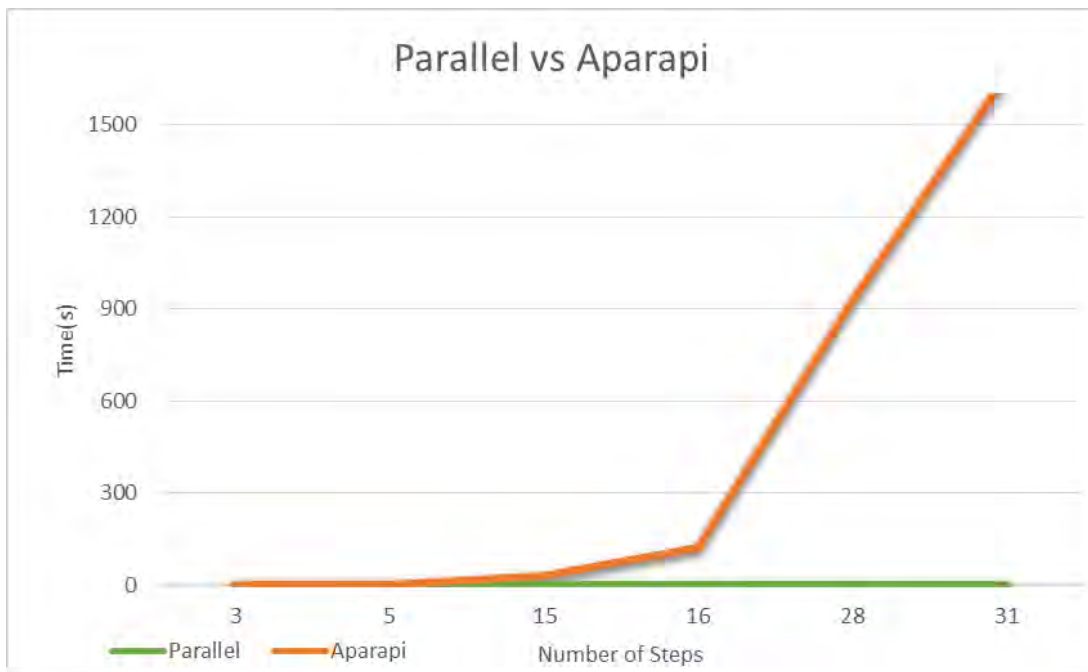


Figure 4.12: Aparapi computational time

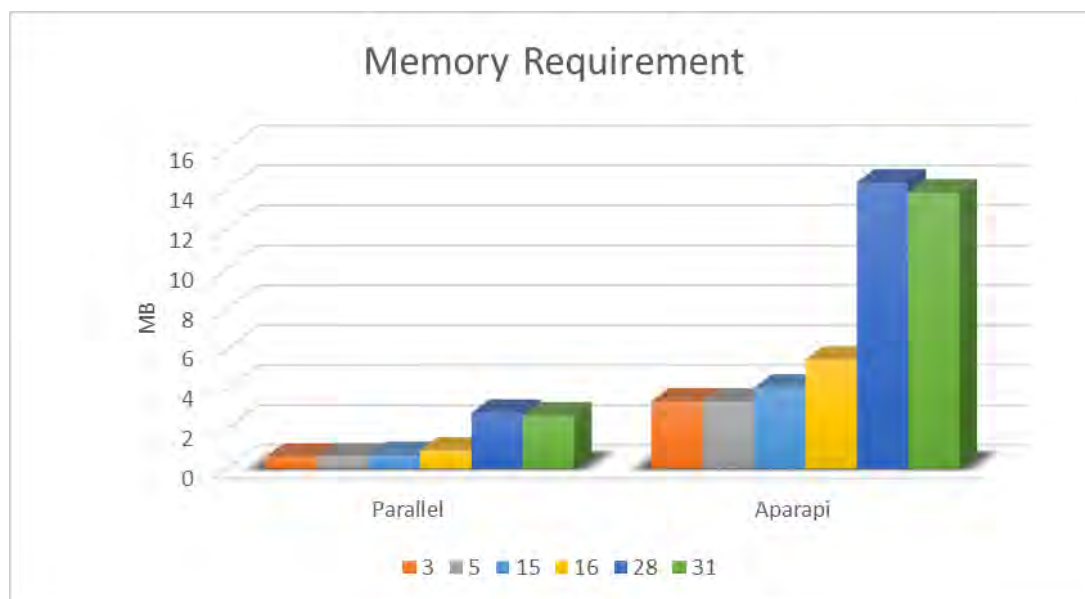


Figure 4.13: Aparapi memory requirement

## 4.5 Performance Evaluation

Prior studies have not compared or analyzed the recursive best first search in both CPU and GPU platforms. Moreover, no other parallel approaches of RBFS exist until the current study. The design of a parallel algorithm that outperforms the sequential algorithm of RBFS has shown in this study. In order to analyze the performance of the proposed algorithm, the sliding puzzle problem has been chosen as it has an exponential tree growth while increasing the puzzle difficulty level and it can be compared between different implementations showing notable results.

Because of exponential search space, the proposed GPU-based RBFS algorithm was much more efficient than the single-thread CPU-based RBFS algorithm in solving sliding puzzle problems. From the analysis of the previous section, it shows clearly that increasing the difficulty level of the sliding puzzles, CPUs computational time increases significantly whereas GPUs computational time decreases. Figure 4.14 and Figure 4.15 shows the comparison between CPU-based RBFS and GPU-based RBFS on different instances of 8-puzzle and 15-puzzle. The memory required is steady for both platforms even though CPUs show increment with the difficulty level whereas GPU is steadier. Moreover, the expansion of states is less in parallel approach than the sequential approach.

Our proposed model shows significant gain in performance; about 99.81% in 8-puzzles and 92.37% in 15-puzzles reduction in the computational speed. While performing the experiments, the number of states fluctuates in GPU, showing that expansion rate is lesser than the CPU approach. Around 67% in 15-puzzles and 96% in 8-puzzles reduction in the expansion rate in the parallel approach than sequential approach. All the experiment items are averaged values



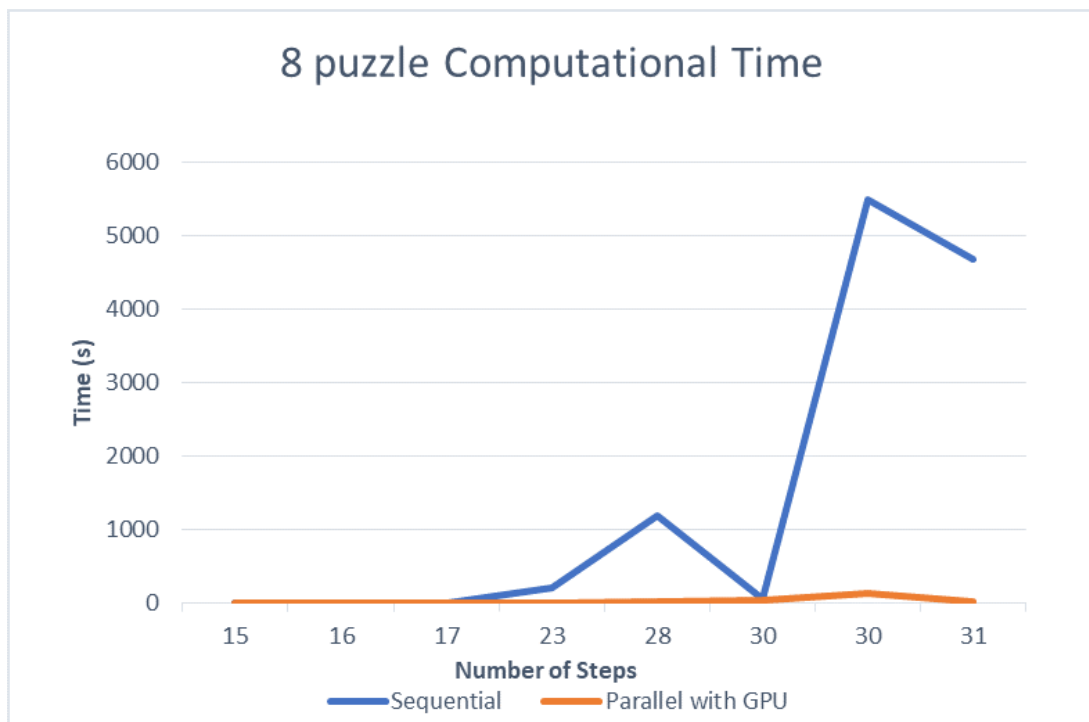


Figure 4.14: 8 puzzle Performance Evaluation CPU-based RBFS with GPU-based RBFS

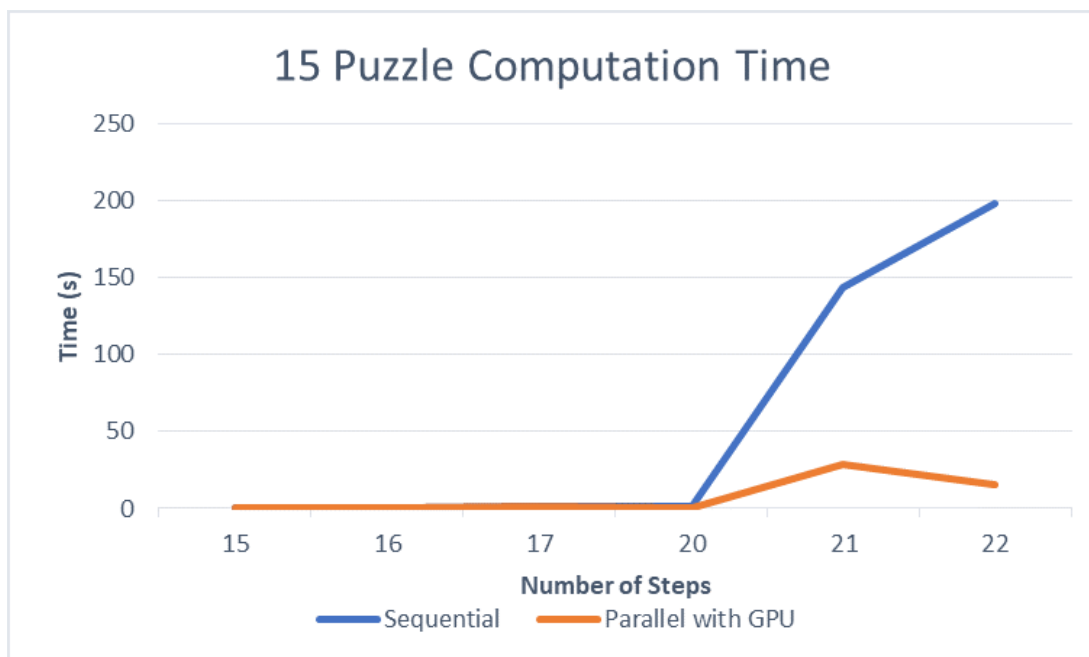


Figure 4.15: 15 puzzle Performance Evaluation CPU-based RBFS with GPU-based RBFS

over five runs in each inputted source and goal state of the sliding puzzle.



## 4.6 Discussion

Recursive best first search is a recursive algorithm that is hard to parallelize. There have been very few studies that attempted to explore the possibility of parallelizing the RBFS algorithm and almost none of studies has focused on GPU platforms to our knowledge. Because of that, the comparison of the RBFS algorithm cannot directly compare our work with state of the art.

In this paper, we propose a parallel technique to parallelize RBFS that has benefited by GPU execution. An innovative and new parallel approach is shown in this study that reduces the computational time of the RBFS algorithm for problems with large-scale search space. In addition, it shows a naïve technique to parallelize a sequential algorithm which is challenging due to its recursive nature.

Our main objective is to design a parallel technique that can parallelize the sequential RBFS algorithm, which can execute on a GPU machine in order to accelerate the performance of this searching technique by exploiting the GPU architecture.

Computation offloading is a technique to optimize resource usage. It has several advantages such as efficient power management, less storage utilization and improvement in the performance of the application. Therefore, offloading several tasks to GPU can substantially provide improvements in the computational speed of large-scale problems as GPU can execute tasks faster than CPU and enhance overall algorithm performance.

Our experimental results of the sliding puzzle demonstrates that GPU-based parallel RBFS algorithms can have a significant speedup compared to the traditional sequential RBFS algorithm with exponential search space and benefited by offloading.

In this paper, also the renowned framework Aparapi using Java for GPU execution has been studied and the results show that GPU execution of the parallel RBFS using Aparapi does not speedup the computational time of the parallel algorithm, instead its performance is quite disappointing compared to other GPU implementations. Even though, it can smoothly execute the different problem instances, its execution time is slower than the actual multi-threaded parallel RBFS algorithm without GPU as well as the parallel GPU based CUDA implementation in python.

# Chapter 5

## Conclusion and Future Works

In this chapter, we conclude the thesis work by detailing the significant contributions of this research work along with some future directions of this study.

### 5.1 Conclusion

This paper shows that RBFS is amenable to parallelization and can provide efficient execution on GPUs. The RBFS algorithm has never been parallelized, and its parallelization is quite challenging due to its recursive calling. None of the prior studies has offered any parallelization approach that can be efficient for RBFS. Neither has any studies focused on the speed-up of the computation execution of RBFS algorithm using GPUs. We are proposing the first variant of RBFS in a parallel manner that can be executed on GPUs devices as well in any other devices that can perform parallelism. Moreover, it shows a novel approach of parallelization technique for sequential algorithms, which can be applied to parallelize recursive algorithms. Extensive performance analyses of three different RBFS implementations (sequential RBFS, parallel RBFS without GPU, and parallel RBFS on GPU) have been shown. The experimental results show performance enhancement on parallel approaches of RBFS trading off the constraint between time and memory. It also demonstrates that the computational speed of the RBFS algorithm on GPU decreases while increasing the difficulty level of the problem compared to the sequential CPU-based RBFS approach. In addition, several different GPU implementations have been studied. A well-known framework for Java language for GPU computation, Aparapi was used for the proposed parallel approach. Its performance shows that it is not suitable for GPU computations of any parallel algorithm.

### 5.2 Future Directions of Further Research

In future, this thesis can have several directions listed below:

- 
- The parallel RBFS can be applied in different problem domains such as pathfinding, protein design, etc.
  - In this thesis, the fitness function uses Manhattan distance heuristic, which ensures an optimal solution of the problem and performs better significantly. In the future, better fitness functions can be used such as pattern databases to gain more efficiency.
  - In this study, an extensive performance analysis on 8-puzzle and 15-puzzle has been shown. It can be further experimented with larger scale puzzles such as 24-puzzle.
  - Moreover, the possibility of using parallel RBFS on a multi-core environment can be explored.

# References

- [1] J. Pearl and R. E. Korf, “Search techniques,” *Annual Review of Computer Science*, vol. 2, no. 1, pp. 451–467, 1987.
- [2] W. Zhang, R. Dechter, and R. E. Korf, “Heuristic search in artificial intelligence,” *Artificial Intelligence*, vol. 129, no. 1-2, pp. 1–4, 2001.
- [3] T. Ungerer, B. Robič, and J. Šilc, “A survey of processors with explicit multithreading,” *ACM Computing Surveys (CSUR)*, vol. 35, no. 1, pp. 29–63, 2003.
- [4] T. Liu, C. Curtsinger, and E. D. Berger, “Dthreads: efficient deterministic multithreading,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 327–336, 2011.
- [5] E. Rawashdeh, M. Qataweh, and H. A. Al Ofeishat, “A new parallel matrix multiplication algorithm on hex-cell network (pmmhc) using iman1 super computer,” *International Journal of Computer Science & Information Technology (IJCSIT) Vol*, vol. 9, 2017.
- [6] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, “Graphics processing unit (gpu) programming strategies and trends in gpu computing,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.
- [7] M. A. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, “Understanding gpu programming for statistical computation: Studies in massively parallel massive mixtures,” *Journal of computational and graphical statistics*, vol. 19, no. 2, pp. 419–438, 2010.
- [8] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, “Aparapi-ucore: A high level programming framework for unconventional cores,” in *2015 IEEE high performance extreme computing conference (HPEC)*, pp. 1–6, IEEE, 2015.
- [9] J. Fumero, M. Papadimitriou, F. S. Zakkak, M. Xekalaki, J. Clarkson, and C. Kotselidis, “Dynamic application reconfiguration on heterogeneous hardware,” in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 165–178, 2019.

- [10] J. Docampo, S. Ramos, G. L. Taboada, R. R. Expósito, J. Tourino, and R. Doallo, "Evaluation of java for general purpose gpu computing," in *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 1398–1404, IEEE, 2013.
- [11] K. G. Gupta, N. Agrawal, and S. K. Maity, "Performance analysis between aparapi (a parallel api) and java by implementing sobel edge detection algorithm," in *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pp. 1–5, IEEE, 2013.
- [12] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis, "Towards prototyping and acceleration of java programs onto intel fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 310–310, IEEE, 2019.
- [13] P. M. Pardalos, L. Pitsoulis, T. Mavridou, and M. G. Resende, "Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and grasp," in *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pp. 317–331, Springer, 1995.
- [14] N. R. Sturtevant, A. Felner, M. Likhachev, and W. Ruml, "Heuristic search comes of age," in *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.
- [15] M. Helmert, T. Lattimore, L. H. Lelis, L. Orseau, and N. R. Sturtevant, "Iterative budgeted exponential search," *arXiv preprint arXiv:1907.13062*, 2019.
- [16] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, vol. 39, pp. 689–743, 2010.
- [17] R. Zhou and E. A. Hansen, "Memory-bounded a\* graph search.," in *FLAIRS conference*, pp. 203–209, 2002.
- [18] Y. Zhou and J. Zeng, "Massively parallel a\* search on a gpu," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.
- [19] V. Kumar, K. Ramesh, and V. N. Rao, "Parallel best-first search of state-space graphs: A summary of results.," in *AAAI*, vol. 88, pp. 122–127, Citeseer, 1988.
- [20] A. Kishimoto, R. Marinescu, and A. Botea, "Parallel recursive best-first and/or search for exact map inference in graphical models," in *Advances in Neural Information Processing Systems*, pp. 928–936, Citeseer, 2015.
- [21] M. Hatem, S. Kiesel, and W. Ruml, "Recursive best-first search with bounded overhead," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

- [22] F. Gieseke, J. Heineremann, C. Oancea, and C. Igel, “Buffer kd trees: processing massive nearest neighbor queries on gpus,” in *International Conference on Machine Learning*, pp. 172–180, PMLR, 2014.
- [23] T. Matsumoto and M. L. Yiu, “Accelerating exact similarity search on cpu-gpu systems,” in *2015 IEEE International Conference on Data Mining*, pp. 320–329, IEEE, 2015.
- [24] S. Li and N. Amenta, “Brute-force k-nearest neighbors search on the gpu,” in *International Conference on Similarity Search and Applications*, pp. 259–270, Springer, 2015.
- [25] H. Wen and W. Zhang, “Improving parallelism of breadth first search (bfs) algorithm for accelerated performance on gpus,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2019.
- [26] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, “A scalable distributed parallel breadth-first search algorithm on bluegene/l,” in *SC’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pp. 25–25, IEEE, 2005.
- [27] F. Fioretto, T. Le, E. Pontelli, W. Yeoh, and T. C. Son, “Exploiting gpus in solving (distributed) constraint optimization problems with dynamic programming,” in *International Conference on Principles and Practice of Constraint Programming*, pp. 121–139, Springer, 2015.
- [28] F. Campeotto, A. Dovier, F. Fioretto, and E. Pontelli, “A gpu implementation of large neighborhood search for solving constraint optimization problems,” in *ECAI*, pp. 189–194, 2014.
- [29] G. Ermiş and B. Çatay, “Accelerating local search algorithms for the travelling salesman problem through the effective use of gpu,” *Transportation research procedia*, vol. 22, pp. 409–418, 2017.
- [30] J. R. Cheng and M. Gen, “Parallel genetic algorithms with gpu computing,” in *Industry 4.0-Impact on Intelligent Logistics and Manufacturing*, IntechOpen, 2020.
- [31] D. Robilliard, V. Marion, and C. Fonlupt, “High performance genetic programming on gpu,” in *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, pp. 85–94, 2009.
- [32] T. A. Marsland and M. Campbell, “Parallel search of strongly ordered game trees,” *ACM Computing Surveys (CSUR)*, vol. 14, no. 4, pp. 533–551, 1982.
- [33] P. Talbot, F. Pinel, and P. Bouvry, “A variant of concurrent constraint programming on gpu,” 2022.

- 
- [34] J. Dong, F. Zheng, J. Lin, Z. Liu, F. Xiao, and G. Fan, “Ec-ecc: Accelerating elliptic curve cryptography for edge computing on embedded gpu tx2,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 2, pp. 1–25, 2022.
- [35] B. Sofranac, A. Gleixner, and S. Pokutta, “Accelerating domain propagation: An efficient gpu-parallel algorithm over sparse matrices,” *Parallel Computing*, vol. 109, p. 102874, 2022.
- [36] O. Hansson, A. Mayer, and M. Yung, “Criticizing solutions to relaxed models yields powerful admissible heuristics,” *Information Sciences*, vol. 63, no. 3, pp. 207–227, 1992.
- [37] N. Pillay, “Intelligent system design using hyper-heuristics,” *South African Computer Journal*, vol. 56, no. 1, pp. 107–119, 2015.
- [38] K. Igwe, N. Pillay, and C. Rae, “Solving the 8-puzzle problem using genetic programming,” in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, pp. 64–67, 2013.

# Index

- 15-puzzle, 28
- 8-puzzle, 28
- Aparapi, 13
- Best-fitted child, 23
- Computation Offloading, 12
- Computational speed, 18
- CPU (Central Processing Unit), 11
- CUDA, 13
- Fitness value, 18
- Graphics Processor Units (GPUs), 11
- Heuristic functions, 19
- Linear Conflict, 30
- Manhattan Distance, 29
- Median, 19
- Multi-threading, 9
- Node duplication, 21
- OpenCL, 13
- Parallel algorithm, 18
- Parallel computation, 18
- Parallel GRBFS, 24
- Parallel processing, 19
- RBFS algorithm, 18
- Recursive Best First Search (RBFS), 7
- Sequential algorithm, 19
- Sliding puzzle, 28



# Appendix A

## PseudoCodes

### A.1 Sequential RBFS pseudocode

This is the pseudocode of the traditional sequential RBFS

```
1 function RECURSIVE-BEST-FIRST-SEARCH(problem)
2 returns a solution, or failure
3
4     return RBFS(problem,MAKE-NODE(problem.INITIAL-STATE),infinity)
5
6 function RBFS(problem,node,f_limit) returns a solution,
7 or failure and a new f-cost limit
8
9     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
10    successors <- []
11    for each action in problem.ACTIONS(node.STATE) do
12        add CHILD-NODE(problem,node,action) into successors
13
14    if successors is empty then return failure,infinity
15    for each s in successors do
16        /* update f with value from previous search, if any */
17        s.f <- max(s.g + s.h, node.f)
18    loop do
19        best <- lowest f-value node in successors
20        if best.f > f_limit then
21            return failure,best.f
22        alternative <- the second-lowest f-value
23            among successors
```

---

```
24         result,best.f <- RBFS(problem,best,  
25                               min(f_limit,alternative))  
26     if result != failure then  
27         return result
```

# Appendix B

## Sample GPU Codes

### B.1 Matrix multiplication with GPU

This is the python implementation for matrix multiplication on GPU using Numba library

```
1 """
2 Matrix_multiplication_example_via_`cuda.jit`.
3 Reference: https://stackoverflow.com/a/64198479/13697228_by
4 @RobertCrovella
5 Contents_in_this_file_are_referenced_from_the_sphinx-generated_docs.
6 "magictoken" is used for markers as beginning and ending of example text.
7 """
8 import unittest
9 from numba.cuda.testing import CUDATestCase, skip_on_cudasim
10 from numba.tests.support import captured_stdout
11 @skip_on_cudasim("cudasim doesn't support cuda import at non-top-level")
12 class TestMatMul(CUDATestCase):
13     """
14     Text matrix multiplication using simple, shared memory/square,
15     and shared memory/nonsquare cases.
16     """
17     def setUp(self):
18         # Prevent output from this test showing up
19         # when running the test suite
20         self._captured_stdout = captured_stdout()
21         self._captured_stdout.__enter__()
22         super().setUp()
23
```

```
24     def tearDown(self):
25         # No exception type, value, or traceback
26         self._captured_stdout.__exit__(None, None, None)
27         super().tearDown()
28
29     def test_ex_matmul(self):
30         """Test_of_matrix_multiplication_on_various_cases."""
31         # magictoken.ex_import.begin
32         from numba import cuda, float32
33         import numpy as np
34         import math
35         # magictoken.ex_import.end
36
37         # magictoken.ex_matmul.begin
38         @cuda.jit
39         def matmul(A, B, C):
40             """Perform_square_matrix_multiplication_of_C=A*B."""
41             i, j = cuda.grid(2)
42             if i < C.shape[0] and j < C.shape[1]:
43                 tmp = 0.
44                 for k in range(A.shape[1]):
45                     tmp += A[i, k] * B[k, j]
46                 C[i, j] = tmp
47         # magictoken.ex_matmul.end
48
49         # magictoken.ex_run_matmul.begin
50         x_h = np.arange(16).reshape([4, 4])
51         y_h = np.ones([4, 4])
52         z_h = np.zeros([4, 4])
53
54         x_d = cuda.to_device(x_h)
55         y_d = cuda.to_device(y_h)
56         z_d = cuda.to_device(z_h)
57
58         threadsperblock = (16, 16)
59         blockspergrid_x = math.ceil(z_h.shape[0] / threadsperblock[0])
60         blockspergrid_y = math.ceil(z_h.shape[1] / threadsperblock[1])
61         blockspergrid = (blockspergrid_x, blockspergrid_y)
```

```
62
63     matmul[blockspergrid, threadsperblock](x_d, y_d, z_d)
64     z_h = z_d.copy_to_host()
65     print(z_h)
66     print(x_h @ y_h)
67     # magictoken.ex_run_matmul.end
68
69     # magictoken.ex_fast_matmul.begin
70     # Controls threads per block and shared memory usage.
71     # The computation will be done on blocks of TPBxTPB elements.
72     # TPB should not be larger than 32 in this example
73     TPB = 16
74
75     @cuda.jit
76     def fast_matmul(A, B, C):
77         """
78         Perform matrix multiplication of C=A*B
79         using CUDA shared memory.
80         Reference: https://stackoverflow.com/a/64198479/13697228
81         by @RobertCrovella
82         """
83         # Define an array in the shared memory
84         # The size and type of the arrays
85         # must be known at compile time
86         sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
87         sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
88
89         x, y = cuda.grid(2)
90
91         tx = cuda.threadIdx.x
92         ty = cuda.threadIdx.y
93         bpg = cuda.gridDim.x # blocks per grid
94
95         # Each thread computes one element in the result matrix.
96         # The dot product is chunked into
97         # dot products of TPB-long vectors.
98         tmp = float32(0.)
99         for i in range(bpg):
```

```
100         # Preload data into shared memory
101         sA[ty, tx] = 0
102         sB[ty, tx] = 0
103         if y < A.shape[0] and (tx + i * TPB) < A.shape[1]:
104             sA[ty, tx] = A[y, tx + i * TPB]
105         if x < B.shape[1] and (ty + i * TPB) < B.shape[0]:
106             sB[ty, tx] = B[ty + i * TPB, x]
107
108         # Wait until all threads finish preloading
109         cuda.syncthreads()
110
111         # Computes partial product on the shared memory
112         for j in range(TPB):
113             tmp += sA[ty, j] * sB[j, tx]
114
115         # Wait until all threads finish computing
116         cuda.syncthreads()
117         if y < C.shape[0] and x < C.shape[1]:
118             C[y, x] = tmp
119     # magictoken.ex_fast_matmul.end
120
121     # magictoken.ex_run_fast_matmul.begin
122     x_h = np.arange(16).reshape([4, 4])
123     y_h = np.ones([4, 4])
124     z_h = np.zeros([4, 4])
125
126     x_d = cuda.to_device(x_h)
127     y_d = cuda.to_device(y_h)
128     z_d = cuda.to_device(z_h)
129
130     threadsperblock = (TPB, TPB)
131     blockspergrid_x = math.ceil(z_h.shape[0] / threadsperblock[0])
132     blockspergrid_y = math.ceil(z_h.shape[1] / threadsperblock[1])
133     blockspergrid = (blockspergrid_x, blockspergrid_y)
134
135     fast_matmul[blockspergrid, threadsperblock](x_d, y_d, z_d)
136     z_h = z_d.copy_to_host()
137     print(z_h)
```

```
138     print(x_h @ y_h)
139     # magictoken.ex_run_fast_matmul.end
140
141     # fast_matmul test(s)
142     msg = "fast_matmul_incorrect_for_shared_memory,_square_case."
143     self.assertTrue(np.all(z_h == x_h @ y_h), msg=msg)
144
145     # magictoken.ex_run_nonsquare.begin
146     x_h = np.arange(115).reshape([5, 23])
147     y_h = np.ones([23, 7])
148     z_h = np.zeros([5, 7])
149
150     x_d = cuda.to_device(x_h)
151     y_d = cuda.to_device(y_h)
152     z_d = cuda.to_device(z_h)
153
154     threadsperblock = (TPB, TPB)
155     grid_y_max = max(x_h.shape[0], y_h.shape[0])
156     grid_x_max = max(x_h.shape[1], y_h.shape[1])
157     blockspergrid_x = math.ceil(grid_x_max / threadsperblock[0])
158     blockspergrid_y = math.ceil(grid_y_max / threadsperblock[1])
159     blockspergrid = (blockspergrid_x, blockspergrid_y)
160
161     fast_matmul[blockspergrid, threadsperblock](x_d, y_d, z_d)
162     z_h = z_d.copy_to_host()
163     print(z_h)
164     print(x_h @ y_h)
165     # magictoken.ex_run_nonsquare.end
166
167     # nonsquare fast_matmul test(s)
168     msg = "fast_matmul_incorrect_for_shared_memory,_non-square_case."
169     self.assertTrue(np.all(z_h == x_h @ y_h), msg=msg)
170
171 if __name__ == '__main__':
172     unittest.main()
```

## B.2 Array Summation using Aparapi

This is the java based implementation for array addition using Aparapi Framework

```
1 package com.aparapi.examples.add;
2 import com.aparapi.Kernel;
3 import com.aparapi.Range;
4
5 public class Main{
6     public static void main(String[] _args) {
7         final int size = 512;
8         final float[] a = new float[size];
9         final float[] b = new float[size];
10        for (int i = 0; i < size; i++) {
11            a[i] = (float) (Math.random() * 100);
12            b[i] = (float) (Math.random() * 100);
13        }
14        final float[] sum = new float[size];
15        Kernel kernel = new Kernel(){
16            @Override public void run() {
17                int gid = getGlobalId();
18                sum[gid] = a[gid] + b[gid];
19            }
20        };
21        kernel.execute(Range.create(size));
22        for (int i = 0; i < size; i++) {
23            System.out.printf("%6.2f_+_%6.2f_=_%8.2f\n", a[i], b[i], sum[i])
24        }
25        kernel.dispose();
26    }
27 }
```

## B.3 Matrix multiplication using Tornado VM

This is the java based implementation for matrix multiplication using Tornado VM Framework

```
1 class Compute {
2     public static void matrixMultiplication(final float[] A,
3         final float[] B, final float[] C, final int size)
```



```

4     for (@Parallel int i = 0; i < size; i++) {
5         for (@Parallel int j = 0; j < size; j++) {
6             float sum = 0.0f;
7             for (int k = 0; k < size; k++)
8                 sum += A[(i * size) + k] * B[(k * size) + j];
9             C[(i * size) + j] = sum;
10            }
11        }
12    }
13 }
14
15 //Creation of a task-schedule for the matrix-multiplication execution:
16 TaskSchedule t = new TaskSchedule("s0").task("t0",
17 Compute::matrixMultiplication, matrixA, matrixB,
18 result, size).streamOut(result);

```

## B.4 Array Summation using JOCL API

This is the java based implementation for array addition using JOCL API

```

1 public class ArrayGPU {
2     /**
3      * The source code of the OpenCL program
4      */
5     private static String programSource =
6         "__kernel_ void_ "+
7         "sampleKernel(__global_ const_ float_ *a, "+
8         "_____global_ const_ float_ *b, "+
9         "_____global_ float_ *c) "+
10        "{ "+
11        "____int_ gid_ = _get_global_id(0); "+
12        "____c[gid]_ = _a[gid]_ + _b[gid]; "+
13        "}" ";
14
15     public static void main(String args[])
16     {
17         int n = 10_000_000;
18         float srcArrayA[] = new float [n];

```

```
19     float srcArrayB[] = new float[n];
20     float dstArray[] = new float[n];
21     for (int i=0; i<n; i++)
22     {
23         srcArrayA[i] = i;
24         srcArrayB[i] = i;
25     }
26     Pointer srcA = Pointer.to(srcArrayA);
27     Pointer srcB = Pointer.to(srcArrayB);
28     Pointer dst = Pointer.to(dstArray);
29
30
31     // The platform, device type and device number
32     // that will be used
33     final int platformIndex = 0;
34     final long deviceType = CL.CL_DEVICE_TYPE_ALL;
35     final int deviceIndex = 0;
36
37     // Enable exceptions and subsequently
38         // omit error checks in this sample
39     CL.setExceptionsEnabled(true);
40
41     // Obtain the number of platforms
42     int numPlatformsArray[] = new int[1];
43     CL.clGetPlatformIDs(0, null, numPlatformsArray);
44     int numPlatforms = numPlatformsArray[0];
45
46     // Obtain a platform ID
47     cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
48     CL.clGetPlatformIDs(platforms.length, platforms, null);
49     cl_platform_id platform = platforms[platformIndex];
50
51     // Initialize the context properties
52     cl_context_properties contextProperties =
53         new cl_context_properties();
54     contextProperties.addProperty(CL.CL_CONTEXT_PLATFORM, platform);
55
56     // Obtain the number of devices for the platform
```

```
57     int numDevicesArray[] = new int[1];
58     CL.clGetDeviceIDs(platform, deviceType, 0, null, numDevicesArray)
59     int numDevices = numDevicesArray[0];
60
61     // Obtain a device ID
62     cl_device_id devices[] = new cl_device_id[numDevices];
63     CL.clGetDeviceIDs(platform, deviceType,
64                       numDevices, devices, null);
65     cl_device_id device = devices[deviceIndex];
66
67     // Create a context for the selected device
68     cl_context context = CL.clCreateContext(
69         contextProperties, 1, new cl_device_id[]{device},
70         null, null, null);
71
72     // Create a command-queue for the selected device
73     cl_command_queue commandQueue =
74         CL.clCreateCommandQueue(context, device, 0, null);
75
76     // Allocate the memory objects for the input and output data
77     cl_mem memObjects[] = new cl_mem[3];
78     memObjects[0] = CL.clCreateBuffer(context,
79         CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOST_PTR,
80         Sizeof.cl_float * n, srcA, null);
81     memObjects[1] = CL.clCreateBuffer(context,
82         CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOST_PTR,
83         Sizeof.cl_float * n, srcB, null);
84     memObjects[2] = CL.clCreateBuffer(context,
85         CL.CL_MEM_READ_WRITE,
86         Sizeof.cl_float * n, null, null);
87
88     // Create the program from the source code
89     cl_program program = CL.clCreateProgramWithSource(context,
90     1, new String[]{ programSource }, null, null);
91
92     // Build the program
93     CL.clBuildProgram(program, 0, null, null, null, null);
94
```

```
95     // Create the kernel
96     cl_kernel kernel = CL.clCreateKernel(program,
97                                         "sampleKernel", n
98
99     // Set the arguments for the kernel
100    CL.clSetKernelArg(kernel, 0,
101        Sizeof.cl_mem, Pointer.to(memObjects[0]));
102    CL.clSetKernelArg(kernel, 1,
103        Sizeof.cl_mem, Pointer.to(memObjects[1]));
104    CL.clSetKernelArg(kernel, 2,
105        Sizeof.cl_mem, Pointer.to(memObjects[2]));
106
107    // Set the work-item dimensions
108    long global_work_size[] = new long[]{n};
109    long local_work_size[] = new long[]{1};
110
111    // Execute the kernel
112    CL.clEnqueueNDRangeKernel(commandQueue, kernel, 1, null,
113        global_work_size, local_work_size, 0, null, null);
114
115    // Read the output data
116    CL.clEnqueueReadBuffer(commandQueue, memObjects[2], CL.CL_TRUE,
117        0, n * Sizeof.cl_float, dst, 0, null, null);
118
119    // Release kernel, program, and memory objects
120    CL.clReleaseMemObject(memObjects[0]);
121    CL.clReleaseMemObject(memObjects[1]);
122    CL.clReleaseMemObject(memObjects[2]);
123    CL.clReleaseKernel(kernel);
124    CL.clReleaseProgram(program);
125    CL.clReleaseCommandQueue(commandQueue);
126    CL.clReleaseContext(context);
127 }
128
129 private static String getString(cl_device_id device, int paramName) {
130     // Obtain the length of the string that will be queried
131     long size[] = new long[1];
132     CL.clGetDeviceInfo(device, paramName, 0, null, size);
```

```
133
134     // Create a buffer of the appropriate size and
135         // fill it with the info
136     byte buffer[] = new byte[(int)size[0]];
137     CL.clGetDeviceInfo(device, paramName, buffer.length,
138         Pointer.to(buffer), null);
139
140     // Create a string from the buffer (excluding the
141         // trailing \0 byte)
142     return new String(buffer, 0, buffer.length-1);
143 }
144 }
```

Generated using Postgraduate Thesis L<sup>A</sup>T<sub>E</sub>X Template, Version 1.03. Department of  
Computer Science and Engineering, Bangladesh University of Engineering and  
Technology, Dhaka, Bangladesh.

This thesis was generated on Tuesday 5<sup>th</sup> April, 2022 at 3:41pm.