© 2015, Alok Sawant. All Rights Reserved.

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

EXPLOITING GPGPU FOR AI GRAPH ALGORITHMS

BY

Alok Sawant

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
awarded by DigiPen Institute of Technology
Redmond, Washington
United States of America

July
2015

Thesis Advisor: Dmitri Volper

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS TITLED

Exploiting GPGPU for AI Graph Algorithms

BY

Alok Sawant

HAS BEEN SUCCESSFULLY COMPLETED ON July 20th, 2015.

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

APPROVED:

Dmitri Volper date Graduate Program Director	Xin Li date Dean of Faculty
Dmitri Volper date Department Chair, Computer Science	Claude Comair date President

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

THESIS APPROVAL

DATE: July 20th, 2015

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Alok Sawant

ENTITLED

Exploiting GPGPU for AI Graph Algorithms

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Dmitri Volper date
Thesis Committee Chair

Pushpak Karnick date
Thesis Committee Member

Erik Mohrmann date
Thesis Committee Member

Gary Herron date
Thesis Committee Member

ABSTRACT

Breadth-First Search (BFS) and Depth-First Search (DFS) are primitives for graph traversal and are used as a basis in many search problems. Recent work has demonstrated various methods on parallelizing the BFS on the Graphics Processing Unit (GPU), but there has been no focus on DFS as it tends to be problem specific to parallelize. Graph Plan is an Action Planner that makes use of a DFS style search to find a solution. This research presents a DFS parallelization algorithm to exploit the processing power of the GPU for this Planner's search step. This new DFS parallelization, while applied to a Graph Plan, is generic and can be applied to other algorithms with similar search requirements. The implementation delivers performance improvements on diverse planner problems.

This thesis is dedicated to my parents who have always believed in me and provided me with this opportunity of a Master's degree from a well-recognized institution.

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Dmitri Volper and Dr. Pushpak Karnick, for their valuable and insightful recommendations in regards to this research and their advice during our several meetings. I would like to thank Dr. Erik Mohrmann and Dr. Gary Herron for being members on my committee. I would also like to thank my Game Project teammates for their suggestions and support. Lastly, I would like to thank Alex Champandard for providing the inspiration for researching on exploiting the GPU through his articles.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . xi

LIST OF FIGURES . . . xii

LIST OF ALGORITHMS . . . xiv

CHAPTER 1 Introduction . . . 1

1. Research Overview . . . 1
2. Intended Outcomes . . . 3
3. Thesis Overview . . . 3

CHAPTER 2 Background . . . 5

1. GPU Programming . . . 5
2. Compute Languages and CUDA . . . 9
3. Prefix Sum . . . 10

CHAPTER 3 Motivation . . . 11

CHAPTER 4 Graph Traversal . . . 13

1. Graph Representation . . . 13
2. GPU BFS . . . 14
3. GPU SSSP . . . 16
4. Summary . . . 18

Page

CHAPTER 5 Monte-Carlo Tree Search 21

1. Algorithm 22
2. Parallel MCTS 25
3. Parallel MCTS on GPU 26
4. Summary 28

CHAPTER 6 Action Planning 30

1. Classical Planning and Games 30
2. Parallel Planning 34

CHAPTER 7 Graph Plan 35

1. Algorithm 36
2. Backward Search Summary 37
3. My Parallel Approach 46
   - 3.1. GPU BFS Search 46
   - 3.2. GPU DFS Search 50

CHAPTER 8 Implementation Details 54

1. SSSP 54
   - 1.1. Grid SSSP 54
   - 1.2. Baseline SSSP 55
2. Graph Plan 56
   - 2.1. Basic version 56

Page

2.2. PDDL Planner 57

3. My GPU BFS Search Approach 58

4. My GPU DFS Search Approach 58

CHAPTER 9 Experiments and Results 62

1. Evaluation methods 62

2. Evaluation Setup 63

3. Logistics Planning Problem 64

4. Results 66

5. Possible CPU+GPU Approach 70

CHAPTER 10 Conclusions 73

1. Future Work 75

2. Final Conclusion 77

REFERENCES 80

LIST OF TABLES

Table Page

1. Graph Plan search tree size for few Gripper problems 48

2. Graph Plan search tree size for few Logistics problems 66

3. Gripper Problem: Performance Ratio of CPU to GPU 70

4. Logistics Problem: Performance Ratio of CPU to GPU 71

5. Gripper Problem: Percentage of the search time taken when number of *openNodes* is less than 4,000 71

6. Logistics Problem: Percentage of the search time taken when number of *openNodes* is less than 4,000 72

LIST OF FIGURES

Figure Page

1. Grid of Thread Blocks 6

2. Memory Hierarchy 8

3. CSR Example: Adjacency matrix A represented as vertex array R and edge array C. 14

4. Different BFS improvement speedups on various graphs compared to the CPU version. 20

5. Outline of a Monte-Carlo Tree Search 22

6. MCTS Example 24

7. (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex (d) and with local mutexes 25

8. Blocks-world planning domain expressed in PDDL. 31

9. Blocks-world planning problem expressed in PDDL. 32

10. Plan to solve problem described in Figure 9 32

11. Graph Plan: Constructed graph example 41

12. Graph Plan: Successful Backward Search 42

13. Graph Plan: Constructed graph for Blocks problem defined in Figure 9 43

14. Graph Plan: Fraction of the Backward-search on Figure 13 44

15. Gripper planning domain expressed in PDDL. 45

16. Graph Plan: Constructed graph for an instance of the Gripper problem 46

17. Graph Plan: Fraction of the Backward-search on Figure 16 47

Figure Page

18. GPU DFS Example 50

19. PC Specifications 64

20. Logistics planning domain expressed in PDDL. 65

21. Gripper Problem: Time Complexity 67

22. Gripper Problem: Nodes visited 68

23. Logistics Problem: Time Complexity 69

24. Logistics Problem: Nodes visited 69

LIST OF ALGORITHMS

1 BFS on the CPU 15

2 BFS on the GPU 17

3 Single Source Shortest Path on GPU 19

4 Graph Plan - Main loop 38

5 Graph Plan - Expand Graph 39

6 Graph Plan - Search for Plan 40

7 Graph Plan - BFS Search for Plan executed on the GPU 49

8 DFS Search for Graph Plan executed on the GPU 51

9 Partial DFS executed by each GPU thread 52

CHAPTER 1

Introduction

1. Research Overview

There is currently a lot of hype surrounding massively parallel GPUs. Consoles and graphics cards currently have massive computational resource at the developers disposal. GPUs though can basically do the same thing a CPU can, just faster. This computation power can, therefore, be used not only for graphics but for other fields too, meaning GPGPUs.

Over the years the clock frequency of CPUs has plateaued out at around 3 GHz. By using turbo boosts it is possible to exceed this limit, but only for a short duration. Thus, over the past 10 years, manufacturers have focused more on parallelism by adding more cores. The big reason for that is power, while performance also plays a major role. More power, allows for better performance and vice versa. Power is still very much an issue now on smartphones. All this points in the direction of GPGPUs for parallelism. GPUs are more efficient than CPUs in terms of power and performance, this is because a GPU provides more control over data management (E.g. data on local memory consumes less power, if not the data can be moved to

shared memory that consumes a little bit more power, or finally to the device/system memory). Thus allowing for less power consumption and better performance. GPU code is also abstracted from how the hardware perceives it, thus allowing for more innovations and no assembly code. The GPU market is also thriving because of the video-game industry and mobile innovations like retina displays. On smartphones, GPU power available is significantly higher than the CPUs, and this is the same trend on desktops.

Current AAA games have very little GPU resource available outside of graphics for physics or AI. GPUs should be thought of for not just completing the task at hand, as they can perform more than that. Even with the downsides to running code on the GPU (i.e., algorithms that will run faster on the CPU), a GPGPU allows us to perform more computations in the same amount of time. Hence, while solving just 1 task could be slower on GPU, solving many tasks will be faster. Therefore, GPU algorithms need to be thought differently as they can process more data and provide more data as output. This is something that is not easily possible on the CPU. Whereas, on the GPU this opens up new opportunities.

There have already been various algorithmic methods explored to be parallelized on the GPU [13]. Even AI techniques which normally tend to be sequential, can be parallelized based on the couple of mentioned points. In the future, based on current trends, more emphasis will need to be put into parallelism using data-decomposition during software development. This means that algorithms will need to be updated to make this possible. This paper will focus on some AI algorithms

running on the GPU.

2. Intended Outcomes

As discussed in Section 1, this research intends to look into the various AI search applications running on the GPGPU and compare the performance results. This research aims to specifically address the following questions:

1. *How to port Graph search algorithms to the GPU?*
2. *How these search algorithms can be used in AI techniques to parallelize on the GPU?*
3. *How do GPU search algorithms compare to their CPU counterparts?*

3. Thesis Overview

This thesis aims to answer the questions put forward in Section 2.

In Chapter 2, we look into the background of GPGPU programming and CUDA. Here, we understand the GPU architecture and its massive parallel computing power. This will clarify on how it is different from parallelism on the CPU. We also explore some of the related fields where the parallel computing power of the GPU can and has been utilized. Lastly, prefix sum, similar to the BFS, has been researched to be parallelized on the GPU as well.

In Chapter 4, we investigate the Graph Traversal on a simple graph and a weighted graph. Here, BFS is explored for the GPU and how it can be extended to work on a weighted graph.

In Chapter 5, Monte-Carlo Tree Search, an AI algorithm, that uses graph traversal is investigated. The algorithm is outlined followed by how it can be parallelized for the CPU and the GPU.

In Chapter 6, we look into what Classical planning is and its current state in Game AI.

In Chapter 7, Graph Plan, an Action Planner algorithm, is described. First, an existing method of Graph plan is examined followed by an analysis into the search step. Two approaches are mentioned that can parallelize it on the GPU.

In Chapter 8, the various GPU algorithms implemented in CUDA are detailed. This includes the new DFS parallelization's implementation details.

In Chapter 9, the experiments and results of the CUDA and CPU version of the Graph plan search algorithm are analyzed.

Finally, in Chapter 10, all the findings are summarized along with the outline of the overall thesis goals. It ends by delivering the areas for future work.

CHAPTER 2

Background

1. GPU Programming

Graphics Processing Units (GPUs) are made up of hundreds of small processing elements compared to a small number of cores on a CPU. GeForce GTX 980, the most recent NVIDIA graphics card, contains 2048 processing elements also known as the CUDA cores. GPUs are very good at floating-point computations, managing large quantities of information and well suited at handling large arrays or images, with special hardware for various image processing operations. Thus, any algorithm that falls under any of these categories, has the potential of being ported to the GPU [27].

All these processing elements, also called threads, act as mini-processors. The big difference is that they are not standalone processors, they work in groups also known as blocks [4]. All blocks in turn belong to a grid. Creating and using these light-weight threads have very little overhead compared to on a CPU. Thus, threads in a block work together at the same time by executing concurrently on one multiprocessor. From the CPU, i.e., a host, a kernel can be executed which is a function that runs on the GPU, i.e., a device. Compared to a C/C++ function, the

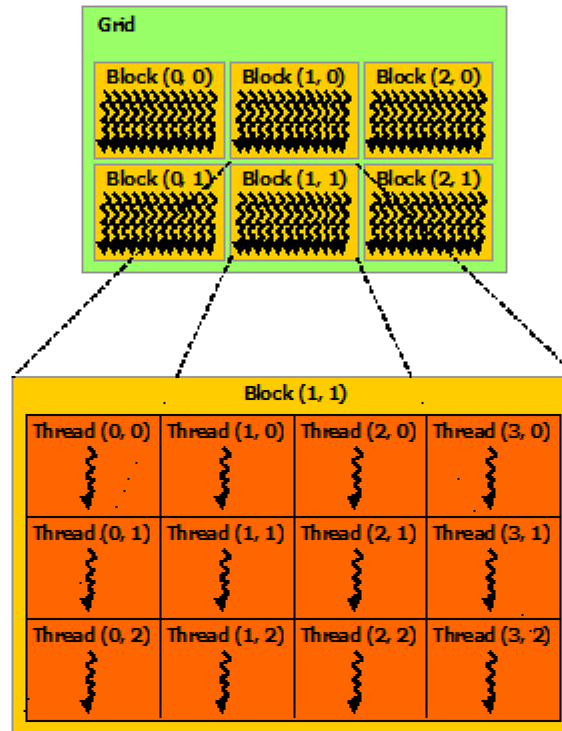


Figure 1. Grid of Thread Blocks

kernel is executed N times in parallel by N threads.

The number of threads and block dimensions are specified during the execution of the kernel, thus allowing for dynamic parallelism. The thread or block dimensions can be specified as one-dimensional, two-dimensional, or three-dimensional vectors for convenience. In Figure 1, each block has 12 threads and they are performing the same execution. Here, if 1 thread goes down a branch of an if condition, all 12 threads will go down that path, the same applies for the else condition. This lock step execution is one of the downsides of coding on the GPU and one has to be careful of this. From a software point of view, each thread is assigned 1 work item by the driver based on the parameters that decide how many blocks to use and how many threads per block.

Figure 1, shows the usage of some threads and blocks to perform a task; instead, the grid and blocks could be saturated with the maximum number of supported threads to enable more computational resource.

As mentioned in the Introduction, a GPU provides more control over data management similar to the thread-block-grid hierarchy illustrated in Figure 2. Accessing a core's on-chip registers is the fastest, followed by the local memory. Threads in a block have access to the same shared memory with the same lifetime as the block. This memory is used by the threads for cooperation. Global memory is accessible across all threads and blocks, at a speed similar to RAM. The host (CPU) uses the global, constant or the texture memory to communicate with the kernel/device (GPU). The local and shared memory have the same lifetime as the thread blocks.

The major downsides to GPU programming compared to CPU are, firstly, that it cannot manage dynamic tasks, secondly, it cannot do random access in memory, and lastly, the code has a lot of branching. As accessing the global memory space can usually be the main bottleneck, the trick, here, is to optimize GPGPU code to minimize this memory latency by making use of local and shared memory wherever possible. One of the many GPU optimizations is to optimally use the threads across available cores. Thus, more threads and blocks are used than the number of streaming processors on the GPU.

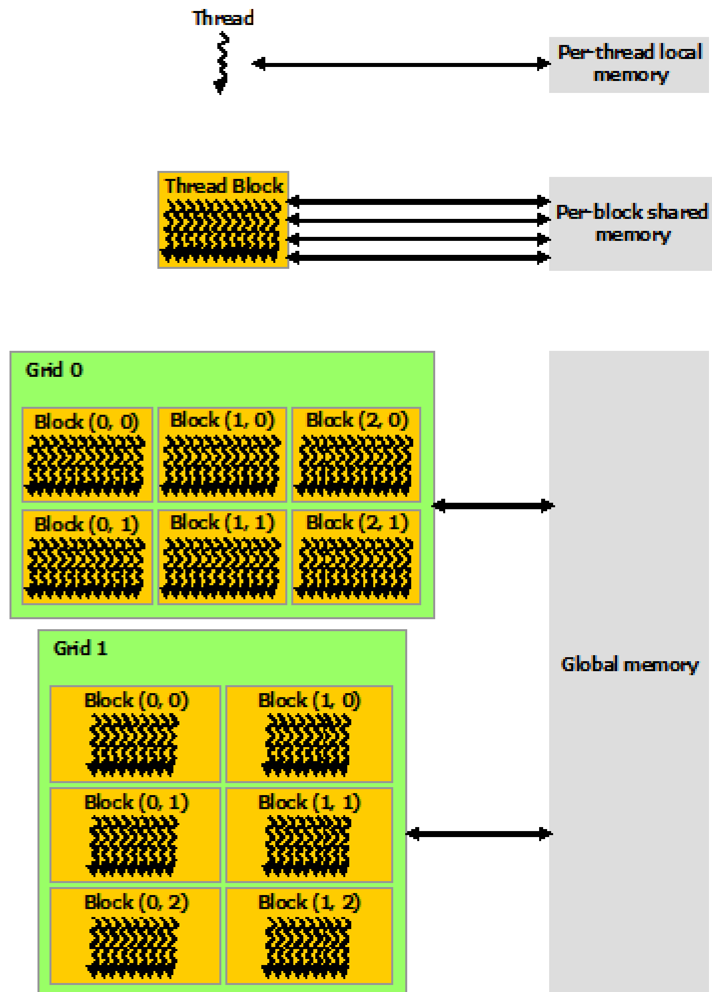


Figure 2. Memory Hierarchy

2. Compute Languages and CUDA

These above terms vary based on which computing language is being used. While OpenCL is trying to be a standard by working on any graphics card, for this paper I'll be using CUDA which runs only on the NVIDIA graphics cards. CUDA allows the developers to use C/C++ as the default programming language to write GPGPU algorithms without the need to learn a new language. CUDA also supports other languages, application programming interfaces, and directives-based approaches such as FORTRAN, DirectCompute, and OpenACC respectively. CUDA enables cooperation for threads in a block, through shared memory and synchronization points, in the kernel. These synchronization points can be used as light-weight barriers forcing all threads in a block to wait before any are allowed to proceed. While this is helpful, unnecessary usage can lead to a performance degradation. The CUDA package also contains the CUDA Thrust library, developed separately, which is a C++ template library similar to STL, using the GPU instead of the CPU. For the implementations discussed in Chapter 8, the Thrust library is used to simplify some of the primitive methods.

There are many graphics samples that use this computation resource to perform many calculations on a per pixel basis for high quality CGI. Algorithms running on the GPU can receive data from the CPU, perform the computation, and then the CPU reads actions to be performed from the GPU. Thereby, allowing us to build more interesting applications. Adobe Premier Pro CS5 onwards, by using CUDA and OpenCL, real-time processing of images/videos can be done while also

applying filters [21]. Civilization 5 for Mac Pro uses OpenCL for better visuals and 4K display support [3]. Avalanche, a game studio in Sweden, uses OpenCL in their tools pipeline for landscape processing for the ”Mad Max” game.

3. Prefix Sum

From a given list of input elements, prefix sum (commonly known as scan) produces an output list where each element is computed from elements occurring prior to this in the input list [20]. In general, it converts a sequential operation into parallel operations. Prefix sum is particularly useful for parallel threads for either totaling data that was individually computed by the threads or to cooperate in accessing shared data structures.

CHAPTER 3

Motivation

As stated in the Introduction, there has been a lot of research done in the academic side of AI to harness GPU resources. This paper summarizes the various GPU graph traversal papers out there. Monte-Carlo-Tree-Search, being a recent research topic, has not been explored in depth on the GPU. Action planning, on the other hand, has had fewer attempts at being massively parallelized. This paper explores some of the approaches on how this, AI to harness GPU resources, can be achieved. Graph plan's parallelization can be utilized in other similar planners.

Game AI, on the other hand, currently uses only the CPU to perform computations. It also needs to compete with other systems, like gameplay and physics, for CPU resources. Due to this, game AI has always had to cut through corners to perform efficiently. There is also this notion that, game AI only has to be good enough and it is, therefore, turned out to be an expectation from players. There have been games that perform exceptionally in AI.

As of now graphics systems primarily use the GPU followed by physics to a small extent. The current GPUs have potential of sharing this resource a little with

other systems, like AI. With the rapid growth of GPUs this may increase in the future, and thus, this thesis provides an overview of some AI algorithms that have been parallelized to execute on the GPU.

AAA games may have very little GPU bandwidth available outside of the graphics systems, other games do not have this restriction. These games can utilize the player's GPU resources to perform a wide range of computations. Nowadays, there are even server machines with GPUs. Networked games, that are anyways performing gameplay/AI computations on these servers, can harness their GPUs. Graph traversals are common queries performed by gameplay logic and AI systems. Instead of performing multiple queries, a GPU graph traversal can massively parallelize them to generate more data to further improve game AI.

CHAPTER 4

Graph Traversal

A graph, a powerful data representation, is described as a set of nodes connected by edges. Graph traversal algorithms are primitive algorithms that serve as key components in many computational domains. Breadth-first Search (BFS), a common graph algorithm, finds the minimum edges to reach a goal node from the root node. Since, the BFS can be represented as a parallel computation, recent researches have shown many promising results of the BFS being accelerated on GPUs.

1. Graph Representation

A graph is commonly expressed as an adjacency matrix, but for sparse graphs using it results in substantial amounts of wasted memory. An adjacency list, on the other hand, is a more compact representation, and for the GPU a single large array containing all adjacency lists works better. This is well-known as a compressed sparse row (CSR) matrix format. Thus, for a graph $G = (V, E)$ where V is a set of n vertices and E is a set of m directed edges, the vertices can be represented as an array R of size n+1 and the edges as an array C of size m. Each entry in R is an

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad C = [0,2,1,2,0,2,3,1,3] \quad R = [0,2,4,7,9]$$

Figure 3. CSR Example: Adjacency matrix A represented as vertex array R and edge array C.

index in C, as illustrated in Figure 3. Such a graph can be traversed by an efficient sequential algorithm with $O(n + m)$ work complexity as described in Algorithm 1. In my implementation, the graph is stored in the order that the edges are defined, techniques that improve locality or load balancing are not performed.

2. GPU BFS

Harish *et al.* [15] presented a vertex-oriented version that performs quadratic parallelization, as illustrated in Algorithm 2. The graph was represented as CSR matrix format, allowing for quick memory lookup. His BFS approach performs a level synchronization by traversing the graph in levels. The vertex-oriented version basically allocates a thread for each vertex in the graph. Each thread checks if it needs to be visited and if so updates its neighbors to be visited in the next iteration. The paper also extends this to solve the Single Source Shortest Path (SSSP) and the All Pairs Shortest Path (APSP) problems. Both SSSP and APSP algorithms make use of 2 kernels because already visited nodes can be revisited. This resulted in a

Algorithm 1: BFS on the CPU

```
1  Q ← ø
2  for i in V do  d[i] ← ∞
3  d[s] ← 0
4  Q.push(s)
5  while Q ≠ ø do
6      i ← Q.pop()
7      for offset ← V[i] to V[i + 1] − 1 do
8          j ← E[offset]
9          if d[j] = ∞ then
10             d[j] ← d[i] + 1
11             Q.push(j)
12         end
13     end
14 end
```

decent speedup compared to other multi-threaded CPU graph implementations, but still not enough primarily due to large memory bandwidth. Kemp [19] also presents various techniques to parallelize the APSP problem.

Real-world graphs tend to be highly irregular which can result in poor performance on the GPU because of irregular memory access and work distribution. Recent researches have proposed efficient methods to improve performance of applications with these heavily imbalanced workloads. A BFS designed for a Parallel Random Access Machine (PRAM) has resulted in a performance gap between GPUs and other multi-threaded CPU algorithms which is primarily due to the large difference in memory bandwidth. The GPU architecture (Section 2) illustrates the low performance despite the available parallel computation resources. Hong *et al.* [17] describes a warp-centric model where each thread performs chunks of work instead of the normal serial approach. This resulted in an impressive speedup. While graphs with a small average edge degree may result in a performance degradation, GPU executions of these graph algorithms outperform CPU versions as the graphs get bigger.

3. GPU SSSP

Champandard *et al.* [8] uses the Bellman-Ford-Moore algorithm on the GPU using OpenCL to perform real-time SSSP along with other game AI calculations. Each thread represents a vertex on a grid, and threads in a group/block cooperate to perform multiple iterations. This cooperation allows for reduced number of kernel

Algorithm 2: BFS on the GPU

```
1  parallel for i in V do
2  |   d[i] ← ∞
3  end
4  d[s] ← 0
5  done ← false
6  level ← 0
7  while !done do
8  |   done ← true
9  |   parallel for i in V do
10 |   |   if d[i] = level then
11 |   |   |   done ← false
12 |   |   |   for offset ← V[i] to V[i + 1] − 1 do
13 |   |   |   |   j ← E[offset]
14 |   |   |   |   d[j] ← level + 1
15 |   |   |   end
16 |   |   end
17 |   end
18 |   level ← level + 1
19 end
```

calls from the host, speeding up the algorithm significantly. Their graph is also stored as a grid and thus enabling an additional optimization by copying a thread block's partial grid to its shared memory. When the shortest path calculations can no longer be permuted for the vertices in a block, the updated shared memory is copied back to the global memory. The grid representation as a 2D array also enables similar memory lookups for adjacent threads.

My GPU version implemented in CUDA uses this same approach, but instead of a grid it uses the CSR format to represent the graph, as illustrated in Algorithm 3. These approaches are also different from the sequential BFS stated above. Here, instead of expanding and updating neighboring nodes, the node is updated iff any of its neighbors have been updated.

4. Summary

While the GPU Bellman-Ford works really well for a grid data structure, the CSR sparse matrix format doesn't match a CPU Dijkstra algorithm. This as explained above is primarily because of the large memory bandwidth. A warp-centric approach can provide the necessary speedup as stated previously. Figure 4 illustrates this improvement over the baseline implementation alongside other possible improvements.

All the above algorithms while having a better time complexity, perform quadratic parallelization resulting in $O(n^2+m)$ work complexity. Merrill *et al.* [22, 23] stated that a parallel BFS algorithm should have $O(n+m)$ work complexity. This can

Algorithm 3: Single Source Shortest Path on GPU

```
1  parallel for i in V do d[i] ← ∞
2  d[s] ← 0; done ← false
3  while !done do
4      done ← true
5      parallel for i in V do
6          shared loop ← true
7          while loop do
8              cost ← d[index]
9              for offset ← V[i] to V[i + 1] − 1 do
10                 neighborCost ← d[E[offset]]
11                 newCost ← neighborCost + weights[offset]
12                 if newCost < cost then  cost ← newCost
13             end
14             SyncThreads; loop ← false; SyncThreads
15             if cost ≠ d[index] then
16                 loop ← true; done ← false
17                 d[index] ← cost
18             end
19             SyncThreads
20         end
21     end
22 end
```

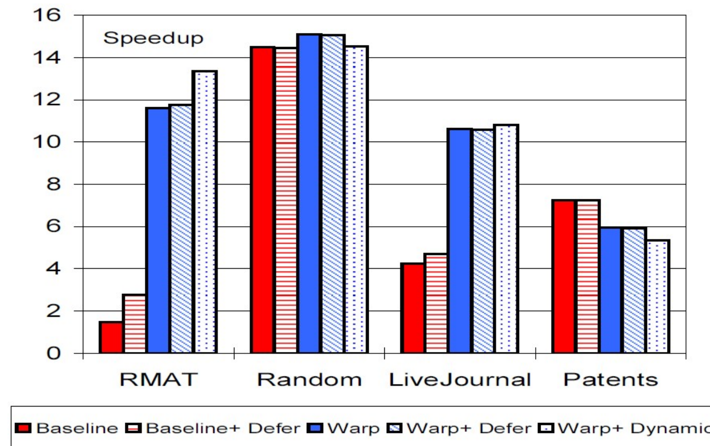


Figure 4. Different BFS improvement speedups on various graphs compared to the CPU version.

be achieved for the above methods by using a parallel queue that contains next set of vertices/edges to open which is similar to a sequential CPU BFS. In his paper he stated that serial-expansion and warp-centric data structures under-utilize the GPU compared to sparse graph datasets. He then proposes a hybrid approach which is a combination of scan, warp, and cooperative thread array gathering techniques. While this hybrid method works even for out-of-core, this optimization will further speedup both the BFS and SSSP problems.

CHAPTER 5

Monte-Carlo Tree Search

Normally implementing AI for computer games requires an in-depth knowledge of the game to design, i.e., an evaluation function that plays an important role in estimating the quality of a game state. This can turn out to be a complex and domain-dependent task. This is the primary reason as to why in a dynamic, complex game world, it is hard to achieve a strong AI even after using domain-knowledge based heuristics. Recent research into the Monte-Carlo Tree Search (MCTS) has led to staggering improvements into game AI for board games [10]. Chaslot *et al.* states that the MCTS makes this easier and with less domain knowledge required. The paper explains the reason behind why the MCTS is effective for modern board games and even RTS games. Recent breakthrough research in using it for Computer Go has risen a lot of interest into other potential applications. Since it can be applied to any game of finite length, theoretically it can, thus, be applied to any domain that can described as $\{state,\ action\}$ pairs and can be simulated [1].

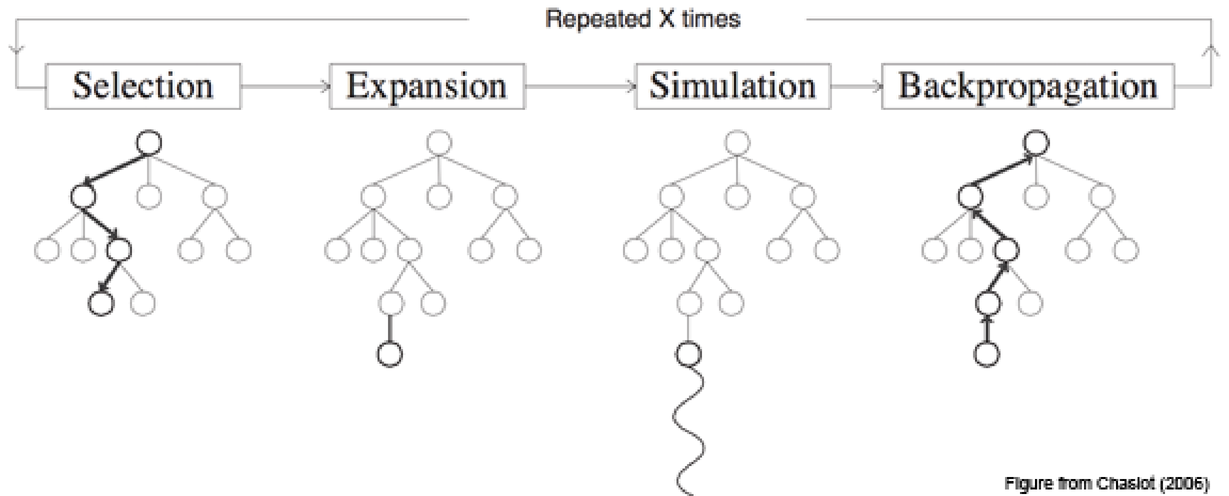


Figure 5. Outline of a Monte-Carlo Tree Search

1. Algorithm

The MCTS basically simulates the game where all players, human and AI, play random/pseudo-random moves. From a single iteration, very little is learned. But from multiple simulated random games, a good strategy can be inferred. Figure 5 illustrates the basic algorithm that generates a tree of possible game states by the following methods:

Selection From a given state, the next action is selected by balancing between exploitation and exploration. Here, exploitation is the task of selecting an action that leads to the best result. Whereas, exploration is using a less promising action that has to be explored cause of the uncertainty of the evaluation.

Expansion From the selected node in the previous stage, one or more of its child nodes are created and added to the tree. This expands the tree for each

simulation.

Simulation From this point onwards, actions are selected at random till a leaf node (end of the game) is reached. Adequate weights/evaluation functions are used for action selection, instead of making all legal actions with equal probability, as otherwise the Monte-Carlo simulation is suboptimal. Heuristic knowledge/evaluation function is used to update action weights, thus promising actions with larger weights.

Back-propagation After completion of the simulation, each node that was traversed is updated. Each node's visit count is increased and win/loss ratio is modified according to the outcome.

Thus, each node contains 2 important variables: the estimated win/loss ratio based on simulation results and the node's visit count. The actual action executed by the program corresponds to the most explored child.

Node Selection is achieved by choosing the node that maximizes a quantity. Typically, an Upper Confidence Bounds (UCB) equation is used:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{5.1}$$

where v_i is the estimated win/loss ratio of the node, n_i is the node's visit count and N is its parent's visit count. Here, C is a constant tunable parameter. The UCB equation balances exploitation of visited nodes with exploration of unvisited nodes. Reward estimates for a node are updated in every random simulation, thus making

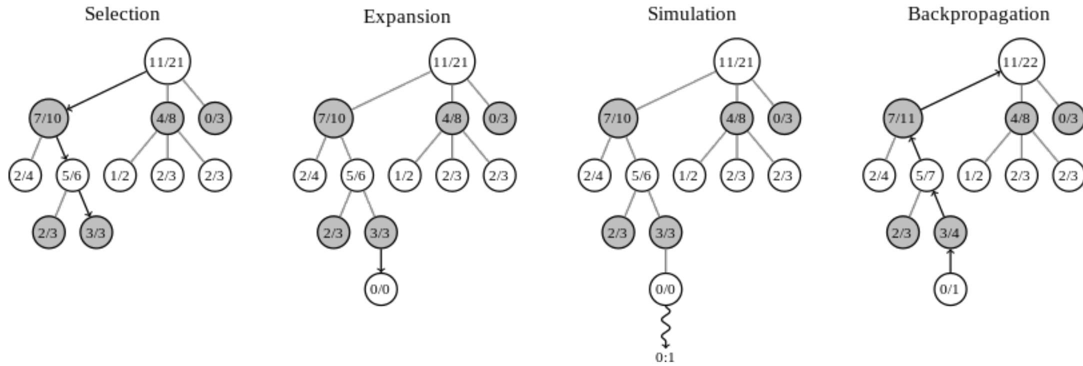


Figure 6. MCTS Example

its estimates more reliable. The MCTS estimates start with being unreliable, but after sufficient searches the estimates are reliable and an infinite search will result in perfect estimates. This search algorithm was first formalized by extending the UCB to a minimax tree search and named as the Upper Confidence Bounds for Trees (UCT) method. The UCT can also be described as a special case of the MCTS. UCT = MCTS + UCB. The basic improvement to this MCTS algorithm is to use the domain knowledge that can be used to filter the implausible moves or increase the weights of moves similar to what is expected by a human. Thus, the nodes will require fewer iterations to generate good estimates.

Figure 6 illustrates a simple example of how the tree looks after a few iterations. Each node has a win/loss ratio. During the selection method, node (3/3) is selected and then its child is expanded and added to the tree as the node (0/0). From this new node, a random simulation is performed till a result is achieved. In this case it is a failure and this result is back-propagated, updating all the visited nodes during this iteration.

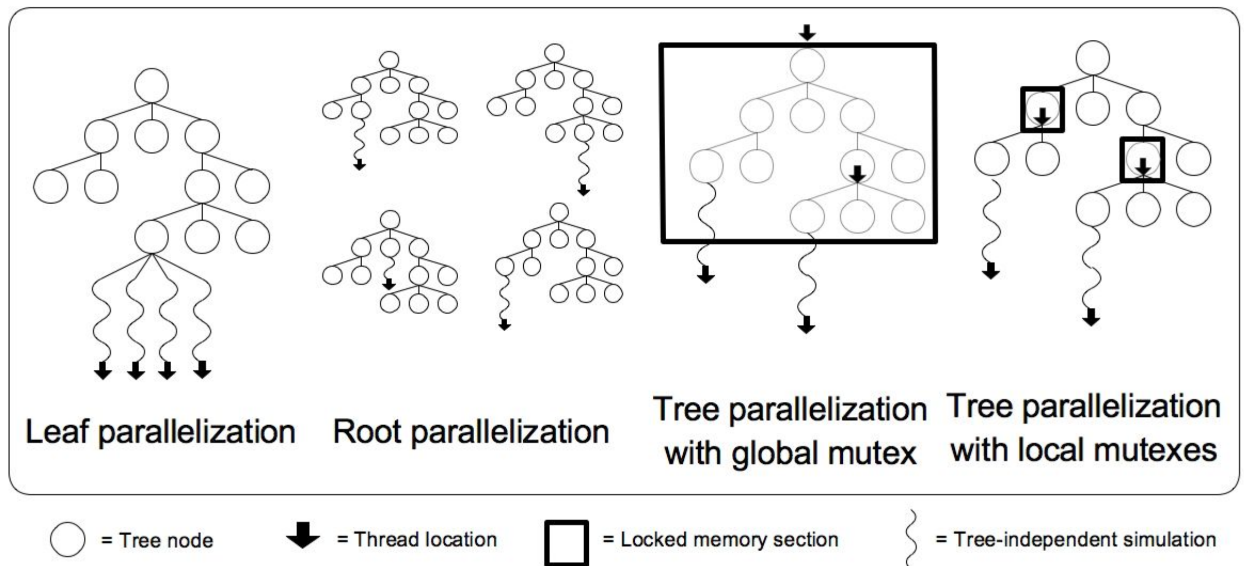


Figure 7. (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex (d) and with local mutexes

2. Parallel MCTS

There are 3 techniques on how to parallelize the MCTS: Leaf, Root and Tree parallelization, illustrated in Figure 7 [11].

Leaf Parallelization This is the easiest, all threads are completely independent as they run independent simulated games. When all threads are done, the results are propagated backwards through the tree on a single thread. Easy to implement and does not require any mutexes. There are 2 problems in this approach. First, game simulation time is random and the program needs to wait for the longest simulated game which is longer than the average simulated game time. Second, no information is shared. If all threads result in losses then

the program might deduce that most games will lead to a loss. There are a few tricks around this, but they can lead to some threads being idle.

Root Parallelization This method consists of building multiple MCTS trees in parallel, one tree per thread. After a certain period of time, all root children of the separate trees are merged. Similar to a leaf technique, minimal communication is needed between threads. This is also easy to implement, even on a cluster.

Tree Parallelization This method builds on the same shared tree with games played simultaneously. Each thread modifies the tree, thus mutexes are needed in some cases to avoid data corruption.

3. Parallel MCTS on GPU

There has been research done to perform these parallelization techniques on the GPU [31, 26]. Tree parallelization is not that viable on the GPU because of the high dependency between threads. As the GPU does not handle complex dependencies that well, this approach has not been explored. Since MCTS is a best-first search algorithm, it cannot utilize a BFS style parallelization as mentioned in Section 4. The leaf parallelization and, to some extent, the root parallelization have potential and have been explored on the GPU. Both these perform something similar to a parallel DFS on the GPU.

Node selection and expansion steps are performed on the CPU. Parallelization happens at the lead node i.e. only the simulation step is performed on the GPU.

Initial board state is copied from the CPU memory to the GPU and executed by thousands of threads in parallel. Each thread manages its own board state in the global memory. After a thread terminates, the score is stored in a preallocated array in the global memory. The threads are synchronized to ensure termination. The scores are accumulated using the prefix sum function in the CUDA Thrust library. The results are copied back to the CPU, where the back-propagation step is executed. After that the next possible nodes to be explored are handed to the GPU. The search tree is, thus, iteratively constructed. After each game, the result is saved by the thread at the respective index in a global array. This array is accumulated by a fast inclusive scan function to generate the total number of wins. Common lookup data is stored in the shared memory. Transferring back to the CPU cannot be done carelessly, e.g., by accumulating all results in a single global variable with atomic lock would create delays for a non-trivial task. Instead, transferring the result into an array accessed using the thread's ID index is faster.

If only a limited number of the CUDA threads are executed in parallel, then the GPUs computational power is under-utilized. On the contrary, too many threads will result in divergent execution paths and unmanaged memory access patterns. Zhuo in his thesis [31] focuses on the variety of configurations to illustrate each setup. The application finds the optimal configuration based on the results. Various default configurations were tried with varying performance. Finally, a default grid size of 512 blocks and each block containing 512 threads was chosen.

4. Summary

Alpha-beta pruning works well under 2 conditions: an adequate evaluation function exists and the game has a low branching factor. These conditions are lacking in many games. On the other hand, the MCTS without any expert knowledge can still achieve a reasonable level of play. The MCTS offers the following advantages:

Aheuristics The MCTS does not require any domain knowledge to make reasonable decisions.

Asymmetric The MCTS performs asymmetric tree growth, i.e., it visits more interesting nodes more often, thereby focusing search time in more relevant parts of the tree. Thus for a 19x19 Go, the MCTS is perfect whereas it causes issues for the standard DFS or BFS.

Anytime The algorithm can be paused/terminated any time to return the current best estimation. The tree built so far can either be discarded or reused for the next iteration.

Elegant It is simple to implement.

There are a few drawbacks to the MCTS that if not taken into consideration can be major:

Playing strength The MCTS, in its basic form, can fail to find a reasonable solution for medium complexity games within a reasonable time frame, mostly, due to the large search space and the key nodes not getting visited as often.

Speed The MCTS can take many iterations to find a good solution. Luckily, there are methods to improve the performance of the algorithm.

Modern board games, like the Settlers of Catan, are discrete, turn-based, incorporate randomness, have hidden information, multiple players and a variable initial state. In Settlers of Catan, the AI implementations are typically rule-based in design, but can still be easily defeated by an experienced player. Machine Learning techniques have gotten close. Chaslot *et al.* implemented an AI for the game using the MCTS [30].

Applications that utilize the GPU get more speed if heavy arithmetic operations are performed. For the MCTS, most operations are branching and memory loading, thus improvements are not drastic as other applications. Although the GPU has more cores than the CPU, a CPU core is much faster than a GPU one. Proper thread configuration and GPU specific optimizations can provide a decent speed-up.

CHAPTER 6

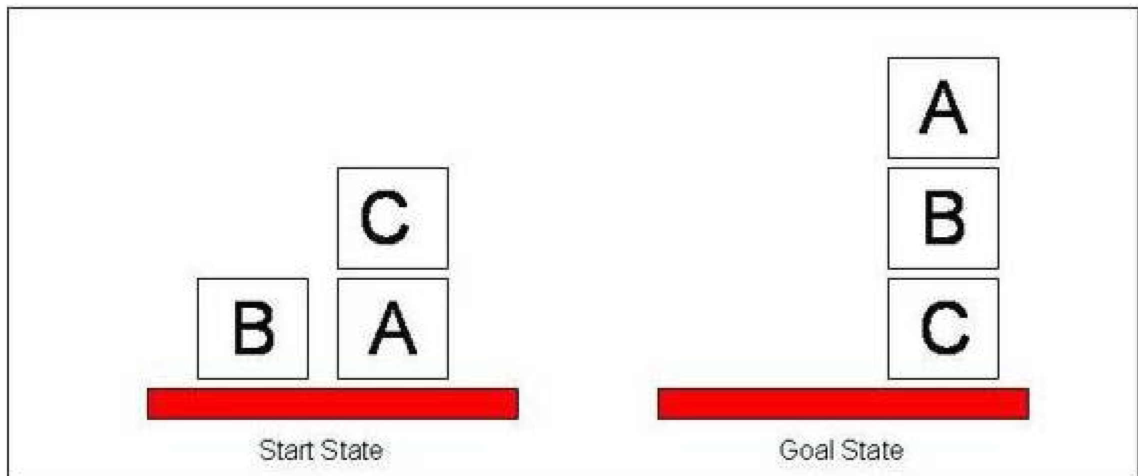
Action Planning

1. Classical Planning and Games

Over the years there have been many advancements in planning systems for development and research purposes. Planners generate sequence of actions to satisfy goals based on given initial state of the world. A classical planning problem consists of the current state of the world represented as a set of predicates. Actions, also called operators, perform changes to this current state. Each action is defined by preconditions and effects, which are also represented as predicates. The goal condition, also a set of predicates, is used as input to a classical planner to generate a plan of actions based on the initial state. PDDL is a standard planning language used to represent the domain and planning problem [2]. For a Blocks-World planning problem, Figure 8 illustrates the domain definition, Figure 9 defines a problem instance and Figure 10 shows the plan to solve it. In the past 10 years, there have been many games that have used planning techniques for decision making. In theory, they can be used to build more compelling, intelligent and entertaining NPCs (non-player characters) or as game directors to craft the perfect player experience [6].

```
(define (domain blocks-world)
  (:predicates
    (on ?x ?y)    ;; Block ?x is on the top of Block ?y
    (clear ?x)    ;; Nothing is on the top of Block ?x
    (onTable ?x)) ;; block ?x is on table
  ; Block ?x is on table and is moved to on top of block ?y
  (:action PutFromTable
    :parameters (?x ?y)
    :precondition (and (onTable ?x) (clear ?x) (clear ?y))
    :effect (and (not (onTable ?x)) (not (clear ?y)) (on ?x ?y)))
  ; Block ?x is on top of block ?y and is move to on table
  (:action PutToTable
    :parameters (?x ?y)
    :precondition (and (clear ?x) (on ?x ?y))
    :effect (and (not (on ?x ?y)) (clear ?y) (onTable ?x)))
  ; Block ?x is on top of block ?y and is move to be on top of block ?z
  (:action PutOn
    :parameters (?x ?y ?z)
    :precondition (and (clear ?x) (clear ?z) (on ?x ?y))
    :effect (and (not (on ?x ?y)) (not (clear ?z)) (clear ?y) (on ?x ?z))))
```

Figure 8. Blocks-world planning domain expressed in PDDL.




```
(define (problem sussman-anomaly)
  (:domain blocks-world)
  (:objects a b c)
  (:init (clear c) (on c a) (onTable a) (clear b) (onTable b))
  (:goal (and (on a b) (on b c))))
```

Figure 9. Blocks-world planning problem expressed in PDDL.

```
1: PutToTable(c, a)
2: PutFromTable(b, c)
3: PutFromTable(a, b)
```

Figure 10. Plan to solve problem described in Figure 9

STRIPS is a state-space planning algorithm that performs a backward search from the goal state by applying actions to satisfy the initial state [14]. The A-star algorithm has been used in the past with a simple heuristic to make it efficient. F.E.A.R. was the first game to use a planner for enemy AI [24]. The enemy AI uses STRIPS to find possible actions it can perform based on the current world state. Monolith's other titles have also used STRIPS-style planning and the AI in those games have also been well received by the players and reviewers.

Hierarchical Task Network (HTN) planners, a technique similar to behavior trees, was later adapted in the video-game industry. There are various HTN algorithms that take a different approach to expanding the plan. SHOP/SHOP2, an ordered HTN planner, was later popularized by Guerrilla Games for their KILLZONE series. High Moon also switched from STRIPS to an HTN planner for TRANSFORMERS: WAR FOR CYBERTRON.

Planning has generally been popular in open world games or with emergent gameplay. Eric Jacopin's analytics provide a very useful insight to these planning techniques for 3 of these games [18]. On average, F.E.A.R. plans are 1-2 actions long from a pool of 26 actions and generates 0.5 plans per second. Whereas, KILLZONE 3 plans are 2-3 actions long from a pool of 44 actions and generates 3 plans per second. Lastly, Transformer 3: Fall of Cybertron plans are 2 actions long from a pool of 137 actions and generates 4 plans per second, though compared to the previous 2 has plans no longer than 12 actions.

2. Parallel Planning

Success of a planner is dependent on the amount of available computational resources. In the last few years there have been many contributions to parallelize planning but mainly on the CPU. Damian Sulewski's domain-independent PDDL planner utilized the processing power of the GPU [29, 28]. Two steps are performed on the GPU, checking for actions that can be performed and generating its possible successors. A delayed duplicate detection step is executed on the CPU, which also uses multiple cores to avoid slow access of the global memory. A lock-free hash table also enables to improve the processing speed. Some of these ideas have been augmented into my GPU Graph Planner.

CHAPTER 7

Graph Plan

Graph Plan is an automated planning algorithm developed by Avrim Blum and Merrick Furst [5]. It takes a planning problem expressed in STRIPS and produces, if achievable, a sequence of actions of achieving a goal state from an initial state. As the name suggests it plans the graph to reduce the amount of search needed for a typical forward-chaining planner. The constructed graph contains constraints on possible plans. If a plan exists, it is a subgraph of the planning graph. These planning graphs can be constructed in polynomial time and the result is the shortest parallel plan. A Graph Planner is sound, complete, and terminates if no plan is present. For a typical state-space planner, the nodes are possible states and the edges are actions performed to achieve the child nodes. In comparison, for a Graph Plan, the nodes are atomic facts and actions, while the edges are preconditions and effects between facts and actions. When searching for a plan, multiple actions can be selected to satisfy sub-goals of that level, meaning that these actions can be executed in parallel. The graph is constructed in the forward pass, and search in the backward pass for a plan. The backward pass is a graph traversal and thus can be performed on the GPU, as

illustrated in Chapter 4.

1. Algorithm

Algorithm 4 explains the basic idea of the Graph Plan. A layered graph is constructed by relaxing a few planning rules. During each iteration, the graph is grown by 1 layer as illustrated in Algorithm 5. Once expanded, the last layer is checked on whether it contains the goal facts. If so, a search, i.e. the backward-pass, described in Algorithm 6 is performed to find the initial state. If the search can reach the first layer, then the goal is achievable and the visited nodes during the search form the plan. If the goal facts are not found, another iteration is performed.

The initial state acts as the first layer of the graph. During the forward-pass, all the actions, having their preconditions present in the current layer, are added. Its effects are added as propositions in the next layer. Even the propositions in the current layer are added to the next layer connected by a no-op action. The actions that can not be applied in parallel are marked as mutually exclusive. The rules to mark actions in mutex are: inconsistent effect, interference and competing needs. Negation and inconsistent support are the rules used to check if propositions in next layer are also in mutex. This is explained in more detail in algorithm 5. Compared to the previous layer, the graph construction is terminated if no more mutex pairs can be removed. If no solution is present at this level, the algorithm concludes that the problem has no solution. Figure 11 illustrates a simple example on how a constructed graph looks like after 3 iterations. In the figure, the red and pink lines signify pair of

actions and propositions that are mutually exclusive. Based on this graph, at level 1, *cook* and *carry* cannot be performed in parallel, but *cook* and *wrap* can be.

The backward-pass performs a search to find a possible solution in this graph. First, the goal facts that are found in the last proposition layer are checked for not being in mutex. This is a recursive step where the goals are marked as subgoals at the deepest layer. Actions from the previous layer are selected that satisfy these subgoals and are not in mutex. If an action set is not found, then the action set from the previous layer selected needs to be changed. Once an action set which is not in mutex is found, then its preconditions are marked as subgoals for the previous layer. This is continued till we reach the first layer, meaning that the actions can be performed from the initial state to reach the goal state. This is explained in more detail in Algorithm 6. Figure 12 gives an example on how this search is performed on the constructed graph.

2. Backward Search Summary

This search is a DFS, where the graph's depth is known and any node at the max depth is a goal node. When thinking this as a DFS, subgoals at a particular layer act as nodes and a set of actions to satisfy these subgoals acts as an edge from this node to a new node i.e. the preconditions of the action set. For the Blocks problem described in Figure 9, the constructed graph is illustrated in Figure 13. In this graph only propositions and actions are depicted, not mutexes as there are quite a lot of pairs. Figure 14 depicts how the backward search looks like. The root node is the

Algorithm 4: Graph Plan - Main loop

```
1  k ← 0
2  level_k.propositions ← initialState
3  while true do
4      plan ← SearchForPlan(k)
5      if plan ≠ ∅ then  return plan
6      if level_{k−1} = level_k then // Terminate if graph has plateaued
7          return ∅
8      end
9      ExpandGraph (k)
10     k ← k + 1
11 end
```

Algorithm 5: Graph Plan - Expand Graph

```
1  foreach Proposition P in level_k do
2  |   level_k.AddNoopAction(P); level_{k+1}.AddProposition(P)
3  end
4  foreach action A in domain do
5  |   if level_k.Contains(A.preconditions) then
6  |   |   level_k.AddAction(A); level_{k+1}.AddPropositions(A.effects)
7  |   end
8  end
9  foreach action pair A_{i,j} in level_k do
10 |   if any effect from A_i negates an effect from A_j then
   |    level_k.MarkMutex(A_i, A_j)
11 |   if any effect from A_i deletes a precondition from A_j then
   |    level_k.MarkMutex(A_i, A_j)
12 |   if any precondition from A_i is in mutex with a precondition from A_j then
   |    level_k.MarkMutex(A_i, A_j)
13 end
14 foreach proposition pair P_{i,j} in level_{k+1} do
15 |   if P_i negates P_j then  level_{k+1}.MarkMutex(P_i, P_j)
16 |   if all support actions of P_i and P_j are in mutex at level_k then
   |    level_{k+1}.MarkMutex(P_i, P_j)
17 end
```

Algorithm 6: Graph Plan - Search for Plan

```
1  if !level_k.Contains(goals) then return ∅
2  if any goal pair in level_k are in mutex then return ∅
3  plan ← ∅
4  subgoals ← goals
5  i ← k
6  while i > 0 do
7      actions ← new set of support actions that satisfies subgoals at level_i
8      if actions = ∅ then
9          i ← i + 1
10         if i > k then return ∅
11         continue
12     end
13     if any action pair in actions are in mutex at level_i then continue
14     plan_i ← actions
15     subgoals ← actions.preconditions
16     i ← i − 1
17 end
18 return plan
```

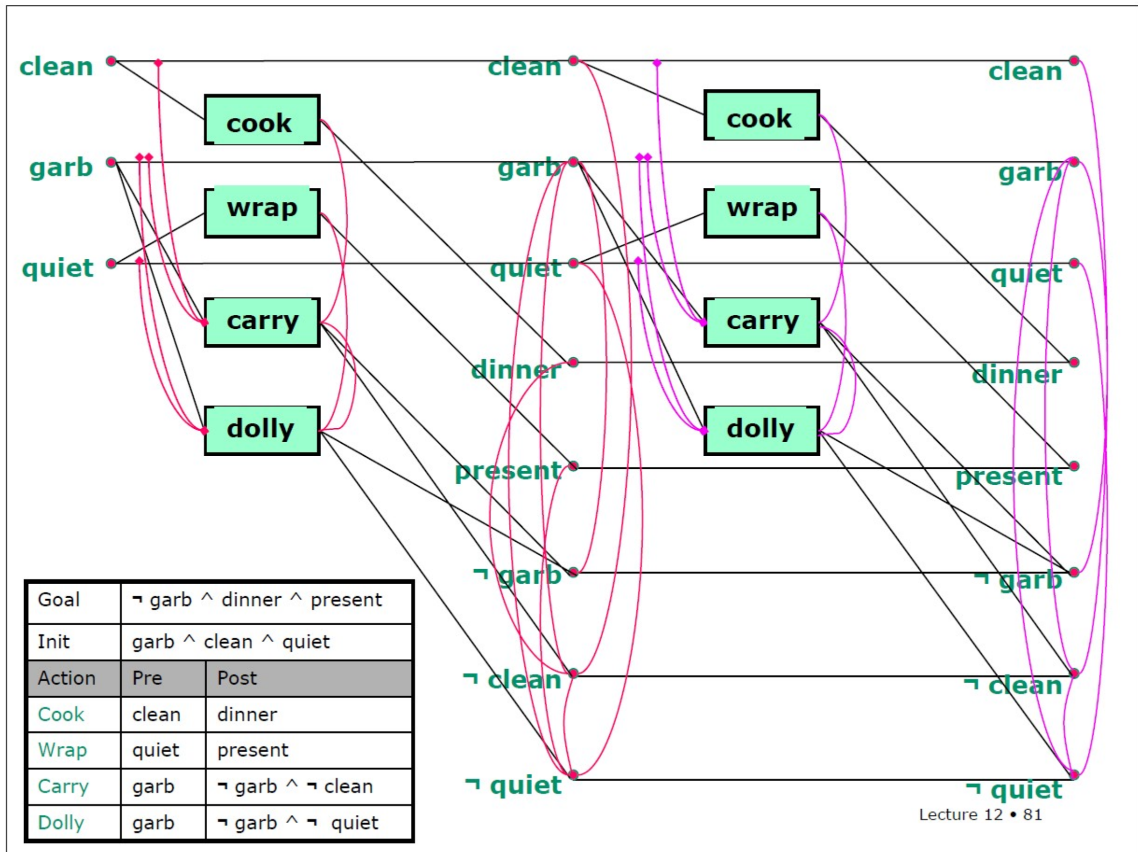


Figure 11. Graph Plan: Constructed graph example

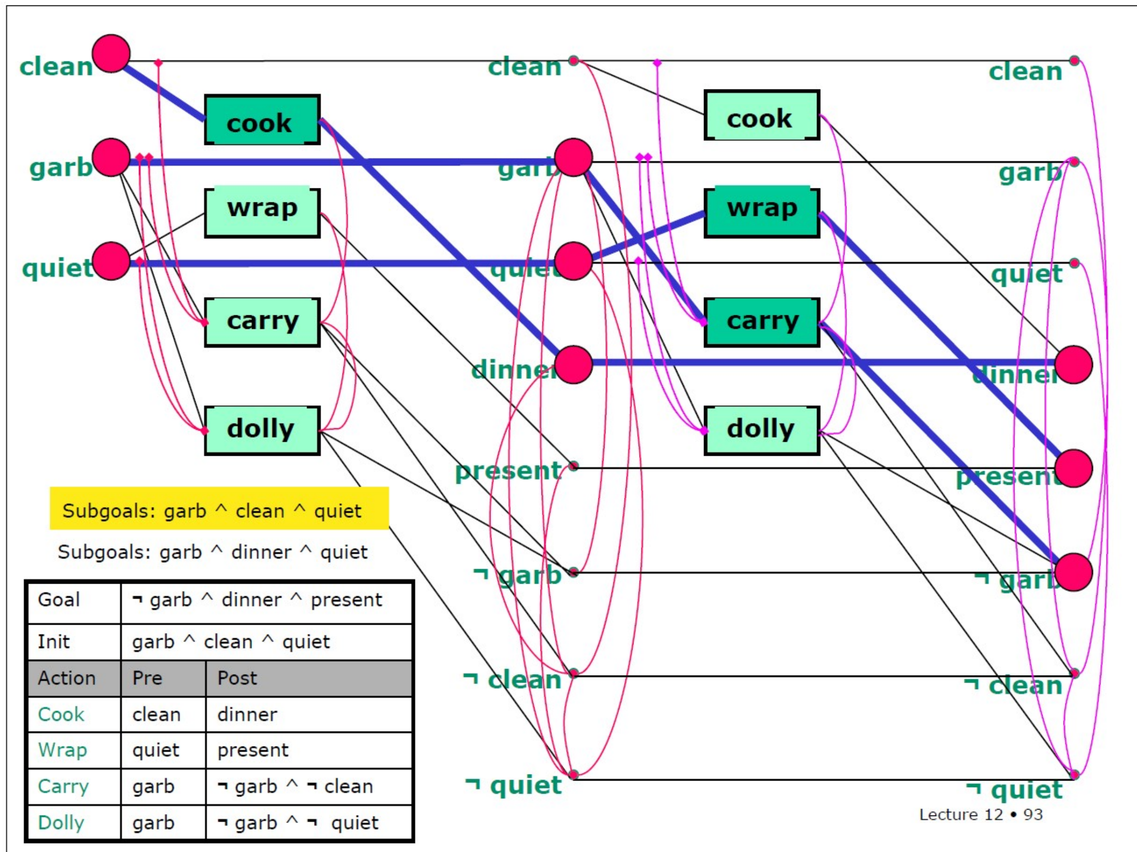


Figure 12. Graph Plan: Successful Backward Search

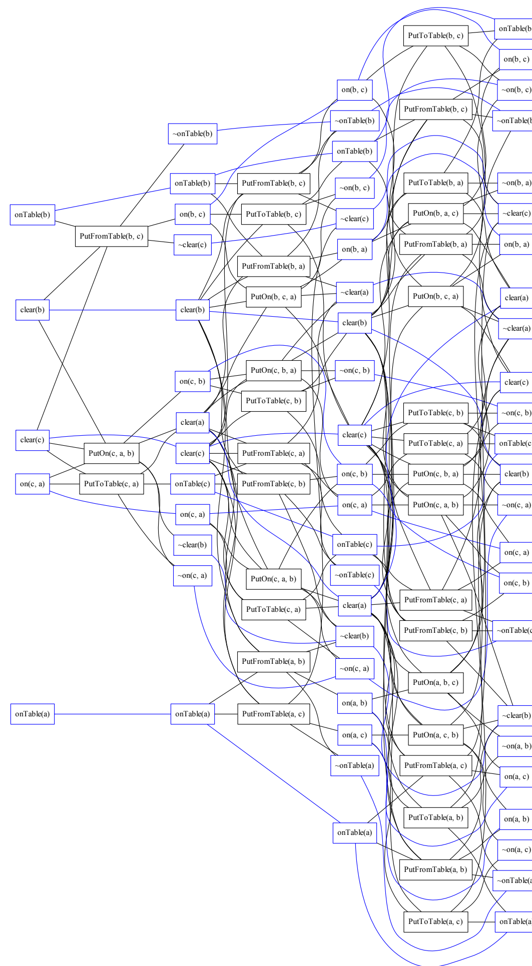


Figure 13. Graph Plan: Constructed graph for Blocks problem defined in Figure 9

goal state, followed by its edges being a set of valid actions with the goal state as effects that are not mutually exclusive. Each proposition in a layer can be satisfied by multiple actions from the previous layer. Therefore, the number of combinations to satisfy a set of propositions can result in a high branching factor for the search tree. For this simple Blocks problem, the backward search is straight forward and simple.

Figure 15 illustrates a common planning domain. All the problem instances

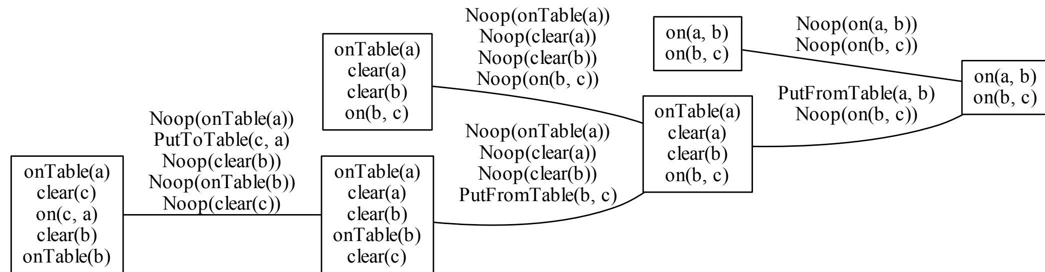


Figure 14. Graph Plan: Fraction of the Backward-search on Figure 13

are basically wherein a robot with select number of arms is to transport set number of balls from one room to another. Based on the number of arms available and number of balls, the plan length varies. This problem is known as the Gripper problem. Thus, this planning problem with such scalability acts as a good testing scenario which is explained in detail in Chapter 9. Figure 16 illustrates the constructed graph for a problem with 2 balls and 1 arm after 7 iterations. As you can see in this graph, after one point, the graph plateaus out once all possible actions and propositions are added. For this problem, Figure 17 depicts a fraction of the search tree. The branching factor for quite a lot of these nodes tends to be high. Thereby, resulting in a lot of time spent searching for a solution. Table 1 depicts the search tree sizes for different Gripper problem instances. These search trees do not have much depth, but have a high branching factor for many nodes.

```
(define (domain gripper-strips)
  (:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r)
    (at ?b ?r) (free ?g) (carry ?o ?g))
  (:action move
    :parameters  (?from ?to)
    :precondition (and  (room ?from) (room ?to) (at-robby ?from))
    :effect (and  (at-robby ?to) (not (at-robby ?from))))
  (:action pick
    :parameters (?obj ?room ?gripper)
    :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
      (at ?obj ?room) (at-robby ?room) (free ?gripper))
    :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
      (not (free ?gripper))))
  (:action drop
    :parameters  (?obj ?room ?gripper)
    :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
      (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room) (free ?gripper)
      (not (carry ?obj ?gripper)))))
```

Figure 15. Gripper planning domain expressed in PDDL.

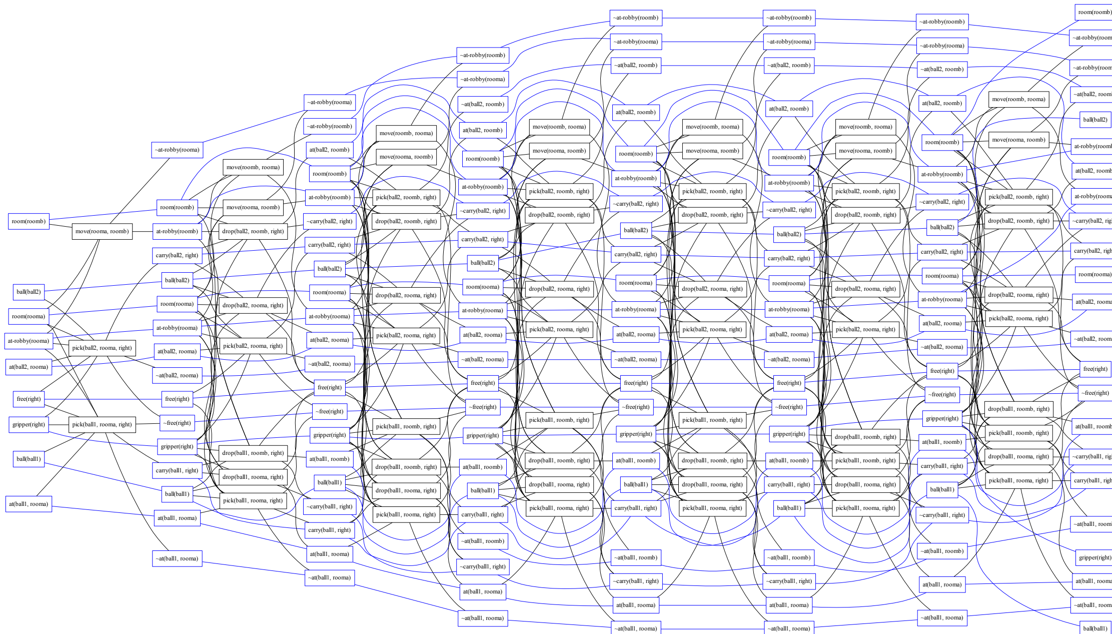


Figure 16. Graph Plan: Constructed graph for an instance of the Gripper problem

3. My Parallel Approach

The Graph Plan's backward search has many domain independent and dependent optimizations to improve it. While these improvements provide significant speedups, they do not change the basic nature of the algorithm i.e., it is still sequential. My approaches try to parallelize the backward search on the GPU.

3.1. GPU BFS Search. My initial approach was to convert the DFS search into a BFS and execute it on the GPU, thus supporting the massive parallelization that will be needed for this search. Algorithm 7 explains the basic approach to perform a parallel BFS style search for the backward-pass. The functions in bold are kernels that are to be executed on the GPU. Given a goal set of facts, different

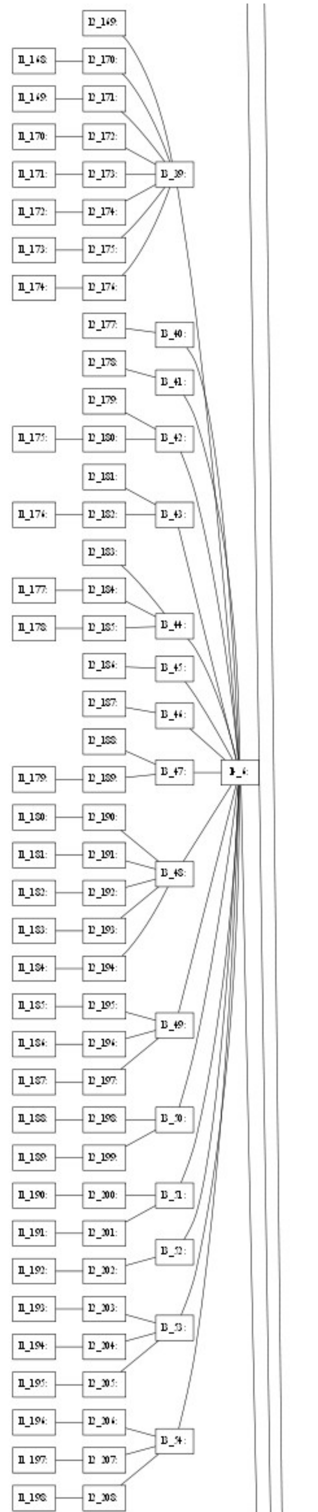


Figure 17. Graph Plan: Fraction of the Backward-search on Figure 16

Problem Index	**Plan Length**	**Total Node count**	**Goal Nodes**
1	3	13	1
2	7	21,011	2
3	7	350,231,947	36
4	8	435,736,183	36
5	11	379,551,789	6
6	12	422,810,229	6

Table 1. Graph Plan search tree size for few Gripper problems

combinations of actions act as edges to the next set of possible subgoal facts for the previous level. There will most likely be many edges that will not be valid and will be terminated. The goal of the BFS search will be to completely exhaust the tree and reach the $level_0$.

This approach initially sounds feasible on the GPU and not on the CPU, primarily because of the heavy branching factor of the graph. A possible drawback would be that the work complexity would be really high as the entire graph would be searched, in comparison to the CPU DFS that could terminate in only after visiting fewer nodes. The other problem is that after each layer, with a high branching factor, the number of nodes to be visited in the next iteration will grow exponentially. This may end up consuming a lot of memory. Thus, a parallel BFS approach is not a viable option for the backward-pass search in a Graph Plan. Another possible approach to this problem could be a semi-BFS that uses a heuristic to prioritize subgoal sets, which leads to my proposed solution of parallelizing this search.

Algorithm 7: Graph Plan - BFS Search for Plan executed on the GPU

```
1  if !level_k.Contains(goals) or level_k.InMutex(goals) then  return ∅
2  inQSize ← 1024
3  inQ.resize(inQSize)
4  edgeComboCntPerNode.resize(inQSize)
5  ResetQ(inQ, inQSize)
6  outQ ← ∅
7  outQSize ← 0
8  inQ_0 ← {goals}
9  openNodeCnt ← 1
10 while levelId > 0 and openNodeCnt > 0 do
11     CalcTotalEdgeCombos(inQ, edgeComboCntPerNode)
12     inclusive_scan(edgeComboCntPerNode)
13     outQSize ← edgeComboCntPerNode.back
14     outQ.resize(outQSize)
15     SearchLevel(inQ, outQ)
       // Swap Qs
16     inQ ← outQ
17     inQSize ← outQSize
18     edgeComboCntPerNode.resize(inQSize)
19 end
```

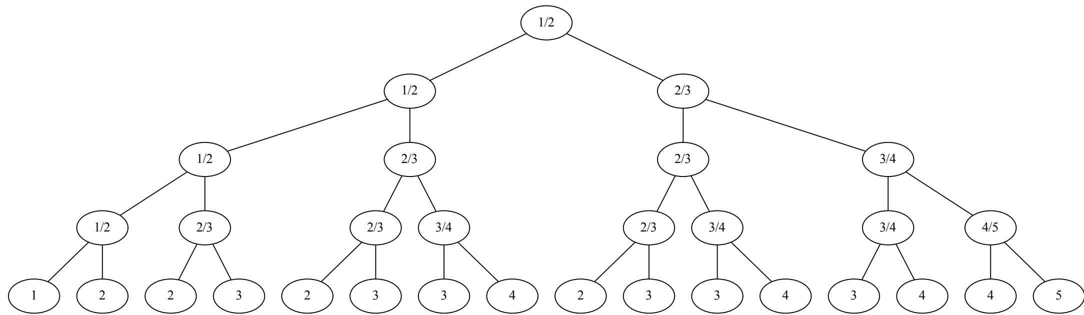


Figure 18. GPU DFS Example

3.2. GPU DFS Search. The backward-pass search of the Graph Plan in some cases is similar to the MCTS and, hence, similar leaf and tree parallelization techniques could be valid here. DFS searches are not easy to parallelize as they are heavily sequential [25]. Algorithm 8 is my approach to parallelize the DFS search. The basic outline is that initially only one node i.e. goal set is used and a partial-DFS execution till a leaf node is performed by the one thread. This partial search is illustrated in algorithm 9. $openNodes_1$ are the visited nodes (sub-goal sets) and the $openNodes_2$ contains the nodes for the next iteration. All visited nodes that have more action combinations left to try, are also saved in the $openNodes_2$. For the next iteration, relevant nodes from $openNodes_2$ are copied over to $openNodes_1$. In the next iteration, these saved visited nodes with new set of operator combinations are executed in parallel, thus slowly exploring the graph to find a solution/plan. This approach can also be regulated by limiting the number of threads and prioritizing the $openNodes$. The remaining nodes (sub-goal set) can be executed in the next iteration. Chapter 8 goes into more detail of how the algorithm works.

Algorithm 8: DFS Search for Graph Plan executed on the GPU

```
1  openNodes_1[0] ← {goals, k}; offsetList_1[0] ← k;
2  exclusive_scan(offsetList_1)
3  while openNodes_1[0].level > 0 do
4      openNodes_2.resize(offsetList_1.last)
5      parallel for i in childCnts_2 do childCnts_2[i] ← 0
6      parallel for i in openNodes_1 do
7          PartialDFS(i, openNodes_1, offsetList_1, openNodes_2, childCnts_2)
8      end
9      exclusive_scan(childCnts_2)
10     openNodes_1.resize(childCnts_2.last)
11     offsetList_1.resize(childCnts_2.last + 1)
12     childCnts_2.resize(offsetList_1.last + 1)
13     parallel for i in openNodes_1 do
14         for j ← 0 to childCnts_2[i + 1] − childCnts_2[i] do
15             openNodes_1[childCnts_2[i] + j] ← openNodes_2[offsetList_1[i] + j]
16         end
17         offsetList_1[i] ← openNodes_1[i].levelId + 1
18     end
19     Sort(openNodes_1, offsetList_1)
20     exclusive_scan(offsetList_1)
21 end
22 return GeneratePath(openNodes_1[0])
```

Algorithm 9: Partial DFS executed by each GPU thread

1 $node \leftarrow openNodes1[threadId]$

2 $openNodeIndex \leftarrow offsetList_1[threadId]$

3 $edgeComboIndex \leftarrow$ **GetEdgeComboIndex**$(node, levels[node.levelId])$

4 $childCnts_2[threadId] \leftarrow 0$

5 **if** $edgeComboIndex \neq -1$ **then**

6 $\quad openNodes2[openNodeIndex++] \leftarrow node$

7 $\quad ++childCnts_2[threadId]$

8 $\quad$ **while** $node.levelId > 0$ **do**

9 $\quad\quad subNode \leftarrow \{node.levelId - 1, node.partialPlan\}$

10 $\quad\quad level \leftarrow levels[subNode.levelId]$

11 $\quad\quad node.partialPlan.edgeCombo \leftarrow edgeComboIndex$

12 $\quad\quad$ **foreach** *Proposition subGoal in node.subGoals* **do**

13 $\quad\quad\quad edge \leftarrow level.$**GetEdge**$(edgeComboIndex, subGoal, node)$

14 $\quad\quad\quad subNode.subGoals.Add(edge.preconditions)$

15 $\quad\quad$ **end**

16 $\quad\quad edgeComboIndex \leftarrow$ **GetEdgeComboIndex**$(subNode, level)$

17 $\quad\quad$ **if** *edgeComboIndex = -1* **then** break

18 $\quad\quad openNodes2[openNodeIndex++] \leftarrow subNode$

19 $\quad\quad ++childCnts_2[threadId]$

20 $\quad$ **end**

21 **end**

Figure 18 illustrates an example of how the search works on the GPU. Compared to a standard DFS, during each iteration from a node, a partial-DFS is performed. A partial-DFS is where from a node, a DFS is run till a leaf node is visited. All nodes other than the child node are marked as future nodes for the partial-DFS in the next iteration. For the first iteration only the root node is present in this queue. In this example, after the first iteration, 4 new nodes are added to the queue. During the second iteration, from these 4 nodes partial-DFS is executed. All new nodes are added to the queue for the next iteration. All nodes in the queue will perform the searches in parallel and, thus, for the third iteration 6 nodes are then executed in parallel. From the previous iteration, nodes still with unvisited children are still added to the queue. Here, the numbers in a node signify the iterations during which that node is executed, meaning the entire tree is explored in 5 iterations. In case of a Graph Plan's backward search, from a node there are many possible action combinations. Thus, not all search edges from a node will be valid. Figure 18 displays only valid edges from a node. Also, for the backward search, the tree is constructed while it is being explored and, thus, the entire tree knowledge is not known at the start which is why algorithm 8 performs an iterative expansion of the tree.

CHAPTER 8

Implementation Details

This study required utilizing a platform to allow for usage of the GPU. Eberly's book [12] provided a good guidance to Compute Shaders and applications other than Graphics. Though, as previously mentioned, CUDA and OpenCL are the more easier languages to pickup. Also, with most researches conducted in CUDA, for the implementations during this study CUDA has been used. NVIDIA's NSight is a powerful tool also included in the CUDA package that enables debugging a kernel function and the GPU profiling features.

1. SSSP

Champandard [9] provides a good introduction to utilizing the GPU for performing a Single Source Shortest Path (SSSP) search. For this study, the initial implementations to understanding the GPU was implementing this SSSP using CUDA.

1.1. Grid SSSP. This SSSP search was conducted on a 2D grid to simplify the search. While initial comparison was against A-star on the same map, this wasn't

a fair comparison. The SSSP implementation was similar to Algorithm 3 where instead of the CSR format, the graph was represented as a 2D array. Thus, for a valid comparison the CPU version was changed to a Dijkstra search. This basic implementation, while it worked, was slower than the CPU version. As stated by Champandard, accessing the global memory frequently can affect the performance. Similar to his approaches, the implementation improvements made use of the shared memory. Each thread block stored a partial-grid in its shared memory, as the threads would be only accessing its neighbors. This copying step was performed each time at the start of the kernel since shared memory have the same lifetime as the thread block itself. Since this implementation was to only explore GPGPU programming, only basic time complexity analysis was conducted. For a grid graph, the GPU algorithm was generally faster than the CPU counterpart. According to Champandard's research [8], for this world representation, storing the graph as a 2D array provided a 10x speedup over the CSR representation.

1.2. Baseline SSSP. This study focuses on generic graphs that can also be irregular and, thus, the graph can not be represented as a 2D array. The follow-up to the above implementation was to use the CSR format. Since the CSR format allows for easy access to even irregular graphs, unless the entire graph is copied to the shared memory, this optimization can not be used. Thus, during each iteration the graph is accessed multiple times from the global memory space. Since the graphs were still 2D maps, the CPU provided better locality of reference. The 2D graphs used were not large enough and a significant change would be required to support

generic graphs available online to perform a better analysis. The proposed goal of this implementation was to grasp a search on the GPU. The mistake was to find a better solution from a published paper instead. Harish's approach [15] to use 2 kernels was tried instead of just 1 so far. One of the primary reasons for the slow performance was the workload imbalance [17]. My initial assumption was that Harish's algorithm could possibly reduce this imbalance. While it did remove the need for synchronization between threads, the algorithm was much slower than Champandard's version. This research was halted to be used as a stepping stone for parallelizing a Planner's search on the GPU.

2. Graph Plan

The Graph plan implementation on the CPU went though various iterations based on profiling needs.

2.1. Basic version. The initial Graph Plan implementation represented the state using literals. An Action's preconditions and effects were also represented using literals. Another mistake during this study was to start designing and implementing the backward search without giving much thought to analysis approaches. The basic version while was easy to debug and implement, the domain and problem representation did not follow any standards. This lead to fewer available problems and none were scalable.

2.2. PDDL Planner. The major re-factor to the implementation was to use the standard PDDL format for representing the domain and problem files [7]. The final CPU Graph Plan implementation was a domain-independent PDDL compliant. The directed, leveled graph contains facts as nodes which are also known as propositions. For this implementation, an edge is an Action abstraction that connects multiple propositions from current level to the propositions in the next level. These Action edges are connected to propositions in the current level that are its preconditions and effects propositions in the next level. No-op action edges signify transfer of all propositions to next level. $Level_0$ is basically the initial state, edges added to this are actions that can be performed on the initial state. A level information other than propositions and edges also contains proposition mutexes and edge mutexes for that level. After every iteration, the number of propositions and edges are greater than or equal to the previous level, but the number of mutexes will slowly reduce.

A few basic heuristic approaches were also implemented for the backward search that provided a decent speedup. These improvements were easy to implement on the CPU, though after careful consideration would be much harder to implement on the GPU. Instead, later these optimizations were removed to perform a fair comparison between the 2 search algorithms.

3. My GPU BFS Search Approach

The GPU BFS as stated previously was the first attempt at parallelizing the backward search. After the basic design, a few primitive algorithms like the prefix sum and sorting were explored for implementation. The CUDA Thrust library included in the CUDA package [16], was designed for this purpose, provided a low-level library similar to C++ STL. The mistake was to start implementation of this BFS algorithm without further analysis of the backward search on whether this is a viable approach. As stated previously, detailed analysis of the search tree provided valuable insight into the major drawbacks of this approach.

4. My GPU DFS Search Approach

It was the research into Monte-Carlo Tree Search, that sparked the GPU DFS approach. The CUDA Thrust library is used for this implementation. While initially experimented with the $thrust :: device_vector$ to create arrays on the GPU, the arrays were represented by a simple $thrust :: device_ptr$. The library not only simplifies the code for memory management on the GPU, but also provides few primitive methods that are executed on the GPU. Some of these are used in algorithm 8 and explained later. The graph level info used to perform the backward search needs to be stored in the GPU's global memory. Edges (the no-ops and the domain actions) are stored with only preconditions, as effects are not used during the backward search. This data is stored separately from proposition level data for simplification. This edge level data pertaining to edges contains the edge data for the domain actions and the

no-ops, along with edge mutex pair data. The proposition, internally represented as a predicate, is stored on the GPU as an ID referencing the predicate on the CPU's level data and also includes an array of IDs that are the support edges to it from the previous level. These edges have the mentioned propositions as effects. Similar to edge level data, this proposition level data contains all the propositions at the level on the CPU, but the proposition mutex pairs are also not stored on the device. During the graph construction, forward pass, for each level iteration, the level data is created and stored on the GPU.

For the backward search described in Algorithm 8, each node in the $openNodes$ arrays contains level ID, sub-goals for this node and an edge combination index. This edge combination index is a compact representation of the set of edges selected that satisfy the set of sub-goal propositions. In algorithm 9, the $GetEdgeComboIndex$ device function finds the next valid edge combination for a node. This is done by selecting the next set of edges from the current selection and then checking for mutex pairs similar to the CPU search. If no set of edges can satisfy a node, it is removed from the $openNodes$. Device function $Level :: GetEdge$ returns the edge data that satisfies a sub-goal proposition using similar logic as $GetEdgeComboIndex$. Partial plan contains the solution and, thus, if the search is successful, another kernel function copies the planning solution from the GPU to the CPU. The $offsetList_1$ array is used when referencing $openNodes_2$ array in the kernel as this is different from the $threadId$ that is used to refer $openNodes_1$. Similarly, $childCnts_2$ array while set in the partial-DFS kernel, is used for referencing the resized $openNodes_1$ when copying node

information from $openNodes_2$. Both of these arrays are calculated by performing an exclusive scan (the prefix sum) after populating the arrays. This scan is also executed on the GPU using a CUDA Thrust library method.

The edge mutexes are stored as an adjacency matrix instead of an adjacency list. The decent number of edge mutex pairs per level saves more memory as an adjacency matrix and improves the lookup time. The level data is queried multiple times in the partial-DFS search kernel and since it resides in the global memory, coying it to the shared memory has caused a decent speedup. While my implementation copies entire level data to shared memory, for future work if the level data is larger than the shared memory available, a comprise will need to be made. All threads in a thread block copy the level data in parallel to the shared memory and only then start their individual partial-DFS execution. Even the search-node from the $openNodes$ array is copied by each thread to their local memory instead of querying from the global memory. Similarly, the queries on the global memory are limited to improve the performance of the search kernel function. Another improvement has been on the parallelism for performing the partial-DFS searches. Due to the high branching factor in the search tree, the $openNodes$ arrays grow very fast and this lead to a stall in the kernel execution. This can be fixed by a threshold on the number of nodes/threads that are executed in parallel. In a few iterations, either a goal node will be found or the nodes are removed from the array, resulting in the nodes that were previously beyond the threshold to be executed in the next iteration. To further improve this, the $openNodes_1$ array is sorted to always execute nodes that are closer to the goal

depth. The code around the sort function in algorithm 8 performs this very task. The array being in the GPU and of a large size, the Thrust library's sort method is used which is again executed on the GPU.

CHAPTER 9

Experiments and Results

In this Section, we will look at the various results of running both the CPU and the GPU (CUDA) version of the Graph Plan's backward search for different configurations. Since both searches can be performed on the same constructed graph, the results can be evaluated across various problems. Other than the Gripper problem defined in Figure 15, a Logistics planning problem is also used.

1. Evaluation methods

To evaluate both search algorithms, a number of varying tests were conducted in which the runtime and even the number of nodes visited were recorded. For each problem instance after the graph has been constructed, the CPU and the GPU backward searches are performed. Since the CPU algorithm has no randomization, the number of nodes visited is the same for a problem instance. In the case of the GPU search, for a fixed maximum thread threshold, the number of nodes visited during each iteration will also be the same. Thus for a constructed graph, both search algorithms will record approximately the same runtime and the number of nodes

visited. For each problem instance's backward search, the total number of possible nodes is also calculated by running the CPU version and not terminating when $level_0$ node is visited. Table 1 illustrates this for various Gripper problem instances.

To get a better analysis on each tree search, all nodes have their children ordering randomly shuffled during each experiment. Thus, slightly changing the order in which the tree is searched by both algorithms. Once a solution is found in a graph, this randomization experiment is performed followed by running both the CPU and the GPU algorithms and recording their results. For each problem instance, these experiments are conducted 100 times to get a large sample size. From these results, the mean and the standard deviation are calculated for both algorithms in regards to the search runtime and the number of nodes visited. The mean and the standard deviation give a better understanding to how the search works on a particular graph size.

2. Evaluation Setup

All the experiments for the CPU and the GPU implementation were conducted on a PC with the specification detailed in Figure 19.

To avoid external factors and to have fairly comparable results, all experiments were tested on the same computer. For future work, a broader comparison across various hardware configurations would provide better insight.

- Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
- 12GB RAM
- NVIDIA GeForce GTX 560 Ti 4GB
- CUDA Compute capability version 2.1
- CUDA 6.5 runtime
- Windows 7
- Visual Studio 2012

Figure 19. PC Specifications

3. Logistics Planning Problem

As stated earlier, other than the Gripper planning problem, Logistics is also used for testing. Figure 20 illustrates the planning domain definition. All the problem instances are basically where packages are transported from one location to another using trucks and airplanes. Based on the number of packages to transport, different locations, cities, etc. the plan length and the graph varies. Thus, even this planning problem is scalable for various instances.

Similarly, the Logistics problem instances also have high branching factor in their tree searches. Table 2 depicts the search tree sizes for the different problem instances. All these instances have the same tree depth but different branching factors and goal nodes. Based on this table one can see that each successive problem instance

```
(define (domain logistics-strips)
  (:predicates (OBJ ?o) (TRUCK ?t) (at ?o ?l) (in ?o ?t) (AIRPLANE ?p)
    (AIRPORT ?s) (IN-CITY ?s ?city) (CITY ?c) (LOCATION ?l))
  (:action load-truck    :parameters (?o ?truck ?loc)
    :precondition (and (OBJ ?o) (TRUCK ?truck) (at ?o ?loc) (at ?truck ?loc))
    :effect (and (not (at ?o ?loc)) (in ?o ?truck)))
  (:action load-plane    :parameters (?o ?p ?loc)
    :precondition (and (OBJ ?o) (AIRPLANE ?p) (at ?o ?loc) (at ?p ?loc))
    :effect (and (not (at ?o ?loc)) (in ?o ?p)))
  (:action unload    :parameters (?o ?v ?loc)
    :precondition (and (in ?o ?v) (at ?v ?loc))
    :effect (and (at ?o ?loc) (not (in ?o ?v))))
  (:action fly    :parameters (?p ?s ?d)
    :precondition (and (AIRPLANE ?p) (AIRPORT ?s) (AIRPORT ?d) (at ?p ?s))
    :effect (and (at ?p ?d) (not (at ?p ?s))))
  (:action drive    :parameters (?truck ?s ?d ?city)
    :precondition (and (TRUCK ?truck) (at ?truck ?s)
      (IN-CITY ?s ?city) (IN-CITY ?d ?city))
    :effect (and (at ?truck ?d) (not (at ?truck ?s)))))
```

Figure 20. Logistics planning domain expressed in PDDL.

has an even higher branching factor and more goal nodes.

Problem Index	Plan Length	Total Node count	Goal Nodes
1	9	20,173	256
2	9	278,223	1280
3	9	3,404,717	2496
4	9	11,840,635	3328
5	9	83,687,441	8896

Table 2. Graph Plan search tree size for few Logistics problems

4. Results

Both the Gripper and the Logistics planning problem instances provide varying graphs for comparison. Table 1 describes 6 instances for the Gripper problem and table 2 depicts 5 for the Logistics problem.

For the Gripper instances, Figure 21 describes the time complexity for both the CPU and the GPU algorithms, with the runtime being the mean of the 100 experiments and error as the standard deviation. The runtime recorded on the $y-axis$ is in seconds. Similarly, Figure 22 illustrates the number of nodes visited by both algorithms on these instances. Based on the graphs, instances 1 and 2 are relatively small and the CPU outperforms the GPU, but from instance 3 onwards, the trees are large enough that the GPU's runtime beats the CPU's. Instances 3 and 4, are equally as dense as instances 5 and 6, but with lesser tree depth. Thus, the number

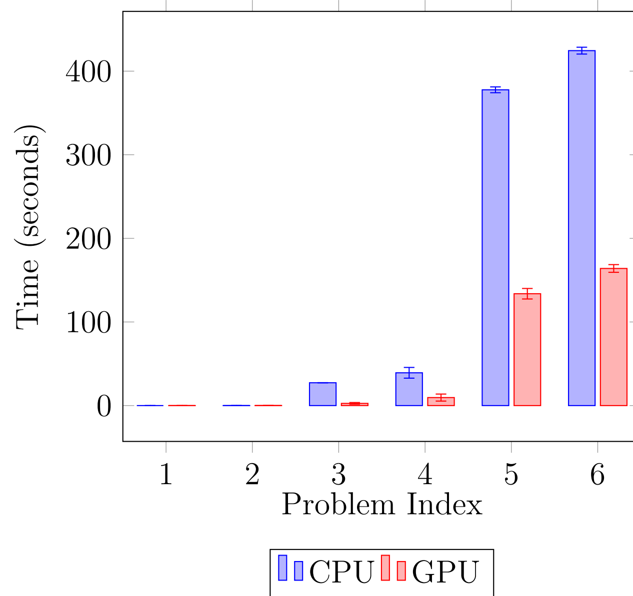


Figure 21. Gripper Problem: Time Complexity

of nodes visited in instances 3 and 4, in comparison to 5 and 6, is lesser for the GPU version than CPU. For both bar graphs, while the standard deviation is visible, it is negligible and does not change the analysis.

For the Logistics instances, Figure 23 describes the time complexity for both the CPU and the GPU algorithms, with the runtime being the mean of the 100 experiments and error as the standard deviation. Similarly, Figure 24 illustrates the number of nodes visited by both algorithms on these instances. Based on the graphs, instances 1 and 2 are relatively small and the CPU outperforms the GPU. While not visible in the graph for instance 3, the GPU does perform almost twice as fast as the CPU version. Instances 4 and 5 have a much faster runtime on the GPU. Since the tree depth for all the instances is the same with increasing branching factor, the number of nodes visited by the GPU is also gradually smaller than that on the CPU.

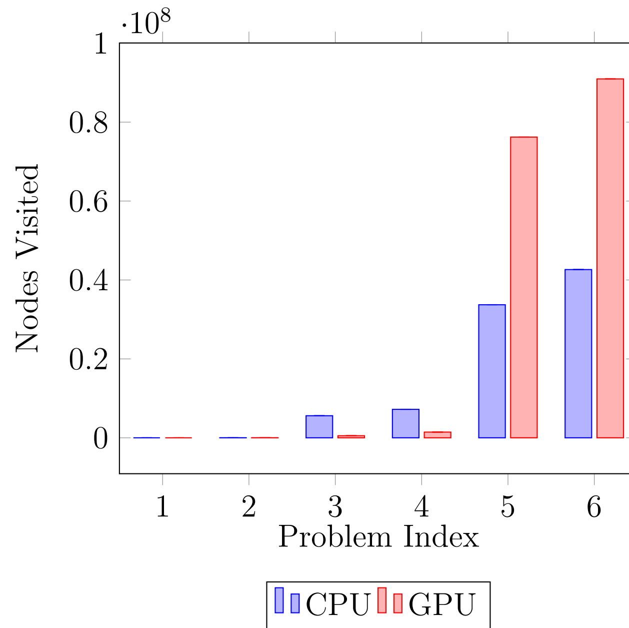


Figure 22. Gripper Problem: Nodes visited

For both bar graphs, while the standard deviation is visible, it is negligible and does not change the analysis.

Tables 3 and 4 illustrate the average speedup and nodes visited for the GPU algorithm over its CPU counterpart. The Problem Index state the instances are the same as those illustrated in their respective Figures. These tables also clearly depicts that for a high branching factor, the massive parallelism on the GPU performs better than a single threaded CPU search algorithm. For large trees with decent branching factors, the nodes visited by the GPU before finding a goal are generally higher than by the CPU algorithm. But for high branching factors, the GPU even performs lesser work than the CPU.

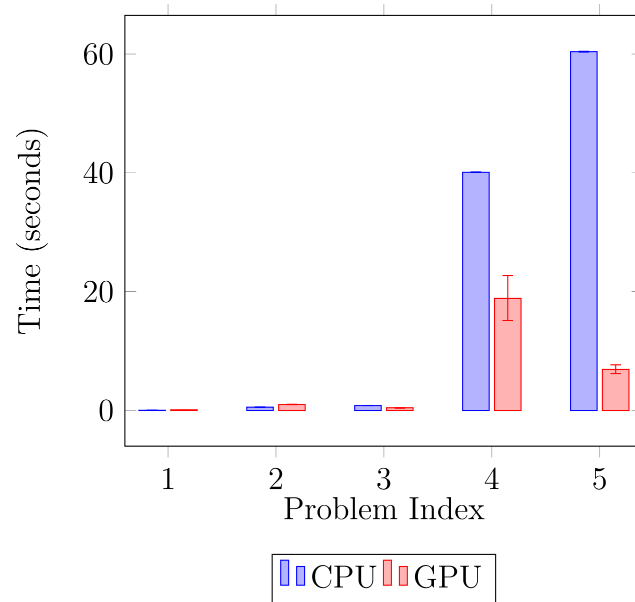


Figure 23. Logistics Problem: Time Complexity

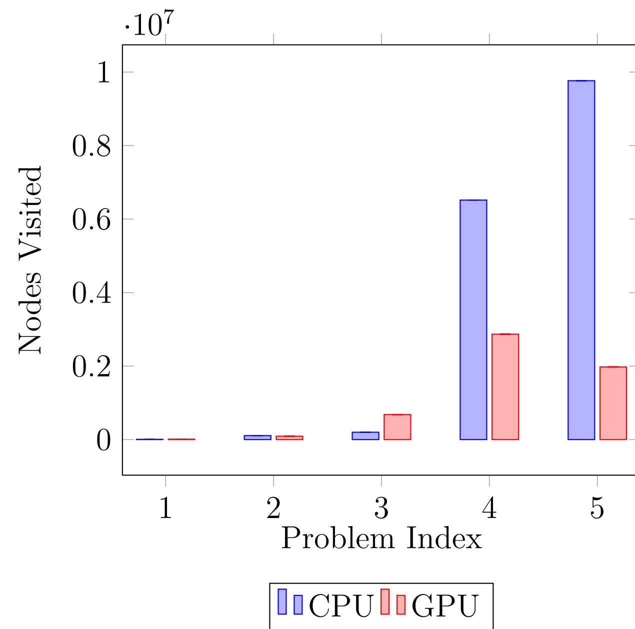


Figure 24. Logistics Problem: Nodes visited

Problem Index	Time	Nodes Visited
1	0.00x	1.00x
2	0.22x	0.56x
3	10.39x	10.59x
4	4.09x	4.97x
5	2.82x	0.44x
6	2.59x	0.47x

Table 3. Gripper Problem: Performance Ratio of CPU to GPU

5. Possible CPU+GPU Approach

One of the possibilities for the GPU still not being fast enough could have been that since it is a parallel DFS, the number of nodes parallelized may not be high enough. To test this, the amount of time spent on the GPU was recorded for when the size of *openNodes* array was less than 4,000. Table 5 illustrates the percentage of the GPU search runtime spent when few nodes were parallelized. Similar to the conclusion drawn in the previous section, for smaller problem instances, the GPU algorithm is slower than the CPU. But for other instances, very little time is needed for the parallel DFS to grow and have enough nodes to parallelize.

For future work, based on these tables, a heterogeneous approach can be performed. The search can be delegated to the CPU if there are not enough number of nodes to parallelize, else the execution can be performed on the GPU.

Problem Index	Time	Nodes Visited
1	0.29x	0.59x
2	0.54x	1.17x
3	1.93x	0.29x
4	2.12x	2.27x
5	8.73x	4.94x

Table 4. Logistics Problem: Performance Ratio of CPU to GPU

Problem Index	Percent time
1	100.00
2	90.00
3	7.08
4	1.70
5	0.01
6	0.01

Table 5. Gripper Problem: Percentage of the search time taken when number of $openNodes$ is less than 4,000

Problem Index	**Percent time**
1	95.49
2	91.83
3	3.69
4	1.50
5	2.03

Table 6. Logistics Problem: Percentage of the search time taken when number of $openNodes$ is less than 4,000

CHAPTER 10

Conclusions

In this thesis, the research was intended to look at ways to harness the GPU resources for AI purposes, mainly search problems. The BFS and the DFS are common techniques used in many different subject areas. At one point, lot of research was dedicated to fast sequential algorithms, but now these require a very fast computer to be executed on. Parallelizing using the GPGPU provides a cheaper alternative at a fraction of the cost. CUDA is a powerful computing technology that abstracts GPGPU programming, making it easier and faster for designing and implementing parallel algorithms. Using CUDA enables us to utilize the GPU for a wide range of applications and not restricting us only to graphics programming.

We have seen a BFS and SSSP algorithm implemented using CUDA in Chapter 4. While the GPU-Bellman Ford is considered as the basic GPU implementation, various other improvements were discussed that can significantly improve the runtime since these improvements relate more to the underlying GPU architecture.

Chapter 5 introduced us to the MCTS which is a DFS style algorithm. We

discussed the various approaches to parallelize it and focused on the leaf parallelization that has been performed on the GPU. While the original MCTS is not a BFS, for massively parallelizing it on the GPU this rule is relaxed. This massive leaf parallelization provides a significant speedup for the simulation stage. This algorithm also explains how even though the GPU can provide a lot of computing power, the CPU is used as the means to manage the data outputted by the GPU kernels.

A Graph Planner is outlined in Chapter 7, that performs a search on the constructed graph during the backward pass. We have seen this search implemented on the CPU and the GPU, and their results explained in Chapter 9. Both algorithms were tested on the same machine and also the same search trees were used on each algorithm. This has ensured that the data does not affect the results and the conclusion. Thus allowing their performances to be fairly compared and analyzed.

We also saw, in Chapter 9, different problems to get an understanding of which algorithm is better suited for an environment. Figures 21 and 23 shows that the Algorithm 6, the DFS on the CPU, is best suited for search trees with smaller depth and branching factor. However, Algorithm 8, parallel DFS on the GPU, provides significant performance increase for search trees with a high branching factor. Thus, for search trees with varying depth and branching factor a modified hybrid algorithm would be beneficial.

Additionally, the number of visited nodes by both Algorithms, discussed in Section 4, provides a different insight into the algorithm. For large graphs with a decent branching factor, the GPU algorithm tends to visit more nodes, but as the

branching factor increases, the algorithm ends up visiting fewer nodes over its CPU counterpart.

Furthermore, this work provides application for many different research areas and is not just limited to the Graph Plan. The parallelized DFS applied on the backward search is a generic DFS that can be applied to any DFS problem that satisfies a certain criteria. For trees with decent tree size i.e. the node count and an adequate branching factor, this GPU algorithm can be applied to improve the search time. The vast usefulness of such applications can result in beneficial future researches.

1. Future Work

The research presented here was successful in designing and implementing a parallel DFS algorithm and comparing its performance on a common machine as well as stating improvements where possible. This work can provide a strong platform for future work, be it in CUDA improvements, general GPGPU programming or new approaches designed following this thesis.

As stated in the Conclusion and Chapter 9, the GPU Algorithm 8 performs better than the CPU Search when the tree is large with a decent branching factor. Even the Table 5, suggests that for fewer nodes a sequential search on the CPU yields faster. Currently, there is no check and either the search is run on the the CPU or the GPU. Thus, a hybrid approach where if the plan's depth is low then a CPU search is performed, but after a certain depth, instead, the GPU Algorithm is executed.

Another approach to this same problem is to check for the number of nodes to be parallelized in the current iteration as illustrated in Algorithm 8. If the node count is below a threshold, then the partial-DFS can be run either sequentially or by parallelizing it on the CPU. Since there in an overhead of executing a GPU kernel, it must be justified by performing massive parallel node searches. This approach may even work for problems with less plan depth, thus not needing the above stated approach. More tests will need to be conducted for these 2 approaches to determine an effective algorithm.

When concerned with the backward search for a Graph Planner, another approach could, also, be to perform a sequential search on the CPU till a certain depth. Beyond this threshold, a BFS can be executed on the remaining tree, as illustrated in Algorithm 7. Since a BFS is easier to parallelize and has had substantial research, all current improvements can also be easily applied to this algorithm. Running the same tests described in Chapter 9 on this approach can provide more insight into parallelizing the backward search.

The Graph Plan's backward search illustrated in Algorithm 9 is also the basic implementation for the CPU. There have been heuristics utilized to help improve the search, domain independent and dependent ones. Further research can be conducted to implement these and also apply similar heuristics to the GPU algorithm. In the case of the CPU version, being sequential, there will be no collision and the work can be reduced. For the GPU version, there will definitely be some work overlap performed by the parallel threads. Thus either on the CPU or the GPU, a collision

detection and avoidance technique can be applied to reduce this overlap.

The data structure utilized for these searches for storing the graph and the search nodes could also be improved. Creating a better data structure solution could also improve the locality of reference, thus reducing the lookup time as that is performed quite a lot during the search. This in a sense is also the drawback of the current approach. While there is a massive parallelization, there is not much computation performed by the individual threads. Researching a new storage method can be beneficial at reducing this overhead.

The BFS improvements that perform a warp-centric search can also be researched for this DFS. Currently, while each thread performs a partial-DFS, all threads in a block are not performing this from the same depth and, also, some threads may terminate earlier than other threads. This leads to unbalanced workload distribution in a thread block. The warp-centric approach [17] for a BFS, can be applied to enable fewer threads to perform more work at less time. There are different methods to approach this problem, and it is another field where further research will be beneficial.

2. Final Conclusion

The research questions stated in Chapter 1 have been answered by this research. Here, the answers to those questions are summarized.

1. *How to port Graph search algorithms to the GPU?*

The primitive search algorithms like the BFS and the DFS can be solved on the GPU as expressed in Chapters 4 and 6. The underlying data structure is very important when implementing for the GPU as your normal methods for the CPU while abstract are not well suited for parallelization. The CSR data structure is an important format for storing nodes and edges, as it provides a locality of reference. Massive parallelism is also important for these algorithms, else the GPU may end being underutilized and run slower than a sequential CPU implementation. Algorithms 2 and 8 perform these very tasks.

2. *How these search algorithms can be used in AI techniques to parallelize on the GPU?*

   Chapter 4 not only explains how a BFS can be parallelized, but also illustrates Algorithm 3 to parallelize an SSSP. The general idea of parallelizing on the GPU is to enable more data calculation and use it for multiple purposes. Champandard [8] explains this by using the SSSP data for path-finding and AI decision making. Chapter 5 focuses on the MCTS which is a powerful algorithm with a lot of potential, and has grown in importance since its usage for the Go AI. The chapter also discusses on how the search can be parallelized, followed by an algorithm to utilize the GPU. Finally Chapter 7 illustrates how a parallel DFS is used to speed up the backward search pass of a Graph Planner.

3. *How do GPU search algorithms compare to their CPU counterparts?*

   While this thesis does not focus on the BFS searches on the GPU other than their basic implementation, there have been various papers that do this in

detail [15, 23, 19]. They provide evidence to the CUDA algorithms performing significantly faster than their CPU versions. For the parallel DFS (Algorithm 8), Chapter 9 gives plenty of evidence that the GPU algorithm performs faster with an average speedup of ranging from 2x to 10x over its CPU counterparts.

REFERENCES

[1] http://mcts.ai/index.html.

[2] Pddl - the planning domain definition language, 1997.

[3] Civilization v gets open cl update for mac pro. 2014.

[4] Guy E. Blelloch. *CUDA C PROGRAMMING GUIDE*. Nvidia, Cambridge, MA, USA, 2014.

[5] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *ARTIFICIAL INTELLIGENCE*, 90(1):1636–1642, 1995.

[6] Alex J. Champandard. Planning in games: An overview and lessons learned. 2013.

[7] Alex J. Champandard and Eric Jacopin. Brute classical planner: Memory efficient forward-chaining. 2014.

[8] Alex J. Champandard and Alexander Shafranov. Advanced opencl techniques for artificial intelligence by example. 2013.

[9] Alex J. Champandard and Alexander Shafranov. An introduction to opencl for massively parallel game ai algorithms. 2013.

[10] Guillaume Maurice Jean-Bernard Chaslot, Sander Bakkes, István Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, pages 216–217, Stanford Univ., California, 2008.

[11] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer, 2008.

[12] David H Eberly. *GPGPU Programming for Games and Science*. CRC Press, 2014.

[13] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 291–294. ACM, 2012.

[14] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[15] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the

gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[16] Jared Hoberock and Nathan Bell. Thrust: C++ template library for cuda. *URL: http://code. google. com//thrust/(: 14.05. 2010)*, 2009.

[17] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, volume 46, pages 267–276. ACM, 2011.

[18] Eric Jacopin. Game ai planning analytics: The case of three first-person shooters, 2014.

[19] JEREMY KEMP. *All-Pairs Shortest Path Algorithms Using CUDA*. PhD thesis, Durham University, 2012.

[20] David B Kirk and W Hwu Wen-mei. Parallel patterns: Prefix sum. In *Programming massively parallel processors: a hands-on approach*, pages 197–216. Newnes, 2012.

[21] Todd Kopriva. Cuda, opencl, mercury playback engine, and adobe premiere pro. 2011.

[22] Duane Merrill, Michael Garland, and Andrew Grimshaw. High performance and scalable gpu graph traversal. *Department of Computer Science, University of Virginia, Tech. Rep*, 2011.

[23] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[24] Jeff Orkin. Three states and a plan: the ai of fear. *Game Developers Conference*, 2006:4, 2006.

[25] Alex Quach, Firas Abuzaid, and Justin Chen. Cs224w project final report. 2012.

[26] Kamil Marek Rocki. Large scale monte carlo tree search on gpu. 2012.

[27] Jason Sanders and Edward Kandrot. *CUDA BY EXAMPLE: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[28] Damian Sulewski. *Large scale parallel state space search utilizing graphics processing units and solid state disks*. PhD thesis, 2012.

[29] Damian Sulewski, Stefan Edelkamp, and Peter Kissmann. Exploiting the computational power of the graphics card: Optimal state space planning on the gpu. In *ICAPS*, 2011.

[30] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in Computer Games*, pages 21–32. Springer, 2010.

[31] Jun Zhou. *Parallel Go on CUDA with Monte Carlo Tree Search*. PhD thesis, University of Cincinnati, 2013.