William Okuno

Final Project Report

Winning Othello Quicker Through Parallel Alpha-beta Search

I. Introduction

The minimax algorithm is a popular one for finding solutions in a search tree. It functions by generating a tree of possible outcomes, scoring the outcomes, and finding the best branch to follow. Alpha-beta pruning is a modification to this algorithm, which seeks to remove branches where better scores than the current node are not possible. Both are commonly implemented to determine the best next move in two-player games. For this implementation, a version of Othello (Reversi) was used. This game was chosen because it has a high branching factor which is not constant over the whole tree. This makes it a challenge for both single processors and parallel processing to solve. In fact, Othello has yet to be strongly solved on an 8x8 or higher grid.

While winning a game of Othello is a fun use of search algorithms, they have far more important applications. Their improvement impacts everything from self-driving cars to medical imaging. Starting with a naive single-processor minimax algorithm, many considerations and optimizations have been developed for divergence, data structure, parameterization, amongst many other things. The hope is that by parallelizing the algorithm and adding in optimizations, results in many areas outside games can be improved. This paper will look at the implementations and performance of a parallel alpha-beta search applied to the game Othello.

For this paper, first, the naive minimax implementation was created, which runs on a single thread of the central processing unit (CPU). While this ran quickly on low depth searches, processing time grew exponentially once the depth was increased. For the graphics processing unit (GPU) parallel alpha-beta implementation, expected speedups are in the order of 1000x faster. This specific version saw average improvements of ~810x on an Nvidia GTX 1080 GPU using CUDA. Additionally, processing time grew in a more linear fashion as depth increased. This would indicate that parallel alpha-beta has significant improvements over the naive serial approach.

II. Design Overview

To play Othello, minimax simulates one player taking all possible moves. Then for each of those moves, simulate all the opponent's possible moves. This is repeated until a specified depth is achieved, or until there are no possible moves left. Once a leaf node is hit, a score is evaluated. Then the tree is traversed upwards, alternating between choosing the minimum and the maximum score, depending on which players turn it was. Once the root node has a final "best" score, that path is selected and the move is made.

The largest improvement came from adding alpha-beta pruning on top of minimax. Since all threads are working on a sub-tree, calculating and tracking alpha and beta is simple and can be done at the same time. The other important change is how the process is broken up. There is no known way to predict the shape of the tree. This makes it hard to run in parallel because you can't predict how large the problem will be. In this version, principal variation splitting (PVS) was used to overcome this challenge. This works by serially traversing the leftmost branch of moves, then launching a parallel search of sub-trees for each possible move off of the serial process's current node. Finally, tracking the current node's alpha and beta and comparing it to each child node's value. At the end of this, another serial process looks over the resulting subtree final values and finds the path that has the highest score to follow. Visualizations of this process exist, which aim to show this process. One is provided below.

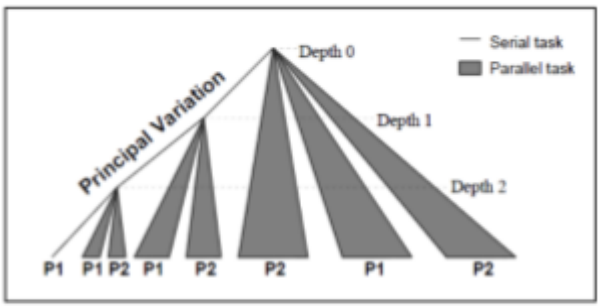

Figure 1 - Diagram of Principal Variation Splitting

III. Implementation

First, to develop this implementation, a version of Othello had to be put together.

Relying on existing projects, the game was put together. Multiple helper functions were implemented and modified to work on both the GPU and GPU implementations. These functions handle game play tasks, such as: finding possible moves, making sure a move is allowed, and scoring the current game. To score the game a simple comparison between the number of white and black pieces is used. From there, naive recursive minimax was added to solve the game using the CPU. Finally, several functions and kernels were developed to modify the serial minimax into parallel alpha-beta. These include the serial portion and parallel portion.

To further elaborate on the game design. Since the goal was to simulate an 8x8 game board, a 10x10 array is used. The values on the outeredge are all set to an edge case value. This leaves an 8x8 board in the middle. This board is stored in an int array of 100 values — 10x10. Positions on the board are encoded by single numbers, where the 10's digit represents the row and the 1's digit represents the column. To make a move, a single integer is given. For instance, the move of row 3 column 5 would be represented as the number 35. The space to simulate moves is set aside in a 64 int matrix, since the board size limits the total number of moves possible. Looking around the board for possible moves, and generating a matrix whose first value - moves[0] - represents the total number of moves possible. If there are 4 possible moves, their values are stored in spots 1-4 of the matrix.

Going deeper on the alpha-beta implementation. The tree is represented by a linked node data structure. Each node stores the current player, the current board state, the parent node pointer, and, of course, alpha and beta. Processing starts by feeding the root node and desired search depth to the primary search function. This function finds all possible moves and takes the first one, then recursively launches itself, taking the next first possible move. From there, an array the size of the number of possible moves is created to store the final score of each move. Next, the GPU memory is allocated and the current node information is provided to the CUDA kernel. The GPU launches enough blocks to cover the number of moves after this first stage. The first thread of each block is assigned to start with the move corresponding to the respective block numbers. Then the kernel launches a search function which generates possible moves. For each move, a child search instance is spawned until either the maximum depth or a leaf has been reached. At this point, a score is generated for the furthest nodes. The tree starts to collapse as alpha and beta are updated. This allows certain branches to be pruned because they could not generate a better score compared with either alpha or beta. The use of alpha or beta depends respectively on the minimum or maximum layers. This results in a final sub-tree score, which is stored. This then exits the kernel and returns to the serial functions. The scores are transferred back to the CPU. The final step for the serial part is to go over the score of each move and determine the best one, returning that best next move.

The serial steps are to help prevent divergence and improve performance. Since the goal is to maximize what each thread is doing, non repeated tasks are conducted

outside the kernel. Additionally, time is lost in each warp to the first thread while conducting its setup for the parallel search operation. This is at a communications cost, since the serial task is spawning multiple kernels sending the same datasets to the GPU multiple times. However, the performance gains are well worth the cost for this implementation, especially for large search depths.

IV. Verification

For verification of the algorithm, two methods were used. Both methods consisted of multiple rounds of adversarial game play between two algorithms - CPU-GPU, or GPU-GPU. It was noted that in CPU-CPU play, the same outcome would occur every round because it was the same algorithms against each other. This is not to mean two CPU algorithms, rather, no variable is changing from the start of each game. The assumption is that the GPU algorithm would have this same characteristic. The reason this was used for verification is because two different versions of the same algorithm were used. While alpha-beta is directly related to minimax, a method that relied on performance of each was not preferable. Additionally, the version of alpha-beta is different from full minimax search, which resulted in some mismatch in game outcomes. If the GPU implementation is able to provide precise results like the CPU implementation it can be assumed to be working correctly. This held true as seen in the figures below. The testing was conducted with a max depth of 5 and across 10 games.

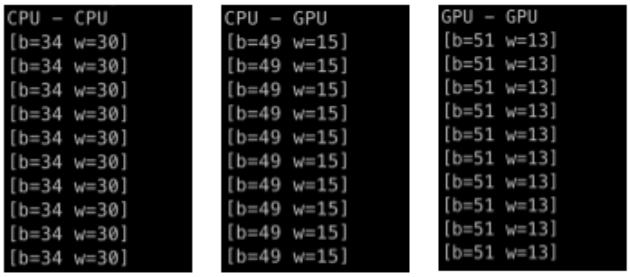

Figure 2 - Precision test result

V. Performance

To look at overall speed up, the execution time of a single game of Othello in the 3 situations is recorded. Games were simulated with a range of depths - from 1 to 6. The plot of depth vs execution time (runtime) shows there is a clear advantage to using the GPU once a depth greater than 4 is required for a task. The CPU-CPU line will continue to increase exponentially, but the GPU-GPU line is increasing at nearly a linear rate. With these values, speedup was calculated for both CPU-GPU and GPU-GPU compared to

CPU-CPU.

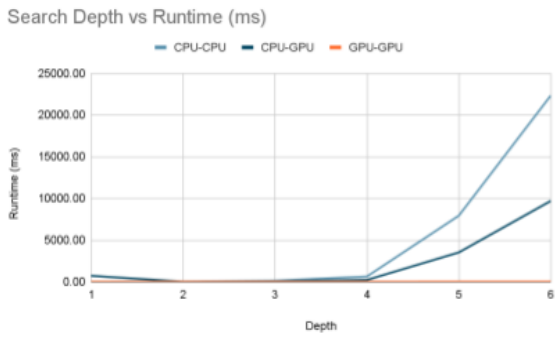

Figure 3 - Search Depth vs Runtime (ms)

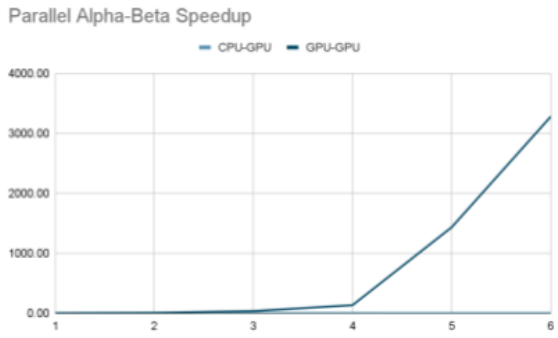

Figure 4 - Parallel Alpha-Beta Speedup

The initial expectation was to see speedups in the 1000x range. There were still multiple steps of serial processing occurring, but most tasks became parallelized. A maximum speed up of ~3200x was achieved. The average speed up within this test was ~810x. This is near the expected 1000x speed up. However, this value should increase as the highest depth tested is also increased, as indicated by the ever increasing speedup. Further investigation into the overhead of the GPU was conducted using nvprof. Tests were done at a depth of 1 and 100. Which showed that the kernel's performance was only bound to its ability to transfer data. Mostly because the task of each thread is relatively simple compared to the amount of data that needs to be transferred. Only comparing values and the only calculation is generating the score. There seems to be room to further improve the performance of the alpha-beta algorithm as well.

1 Depth Kernel Profile		
Percentage	Time	Use
38.65%	101.76us	CUDA memcpy HtoD
34.42%	90.625us	CUDA memset
26.93%	70.913us	CUDA memcpy DtoH

100 Depth Kernel Profile		
Percentage	Time	Use
38.74%	2.5100ms	CUDA memcpy HtoD
34.35%	2.2254ms	CUDA memset
26.91%	1.7436ms	CUDA memcpy DtoH

Figure 5 - nvprof results for search depth of 1 and search depth of 100

Given more time, more optimization and performance improvements could have been made. For instance, sorting out a few errors in the approach taken by the GPU implementation should yield more accurate results. Additionally, the PVS technique utilized for this search is simple compared to modern approaches, such as dynamic tree splitting. Moreover, optimizing data flow could be handled better. Right now, there is no utilization of several features of the GPU that would lead to improved performance. For

instance, the memory structure of the GPU. Since the serial operation currently sends nearly the same results to the kernel, this increases the amount of information being transferred between the host and the GPU. Finally, the scoring algorithm used here is very simple. Better approaches exist that would take into account the branch, not just the leaf's board. These, with a few other improvements, could lead to several more degrees of speedup.

VI. **Conclusion**

This implementation has been achieved after many iterations, and thanks to the wealth of knowledge amongst research and projects. They have provided significant help and guidance when it comes to understanding minimax and alpha-beta, when to take advantage of parallel computation, and how to solve performance bottlenecks. More work needs to be done to increase the accuracy of the results and further improve performance. This could be done by taking advantage of the GPU memory and implementing newer approaches to parallel search, both of which have been explored in other research.

At the end of this project, a successful implementation of parallel alpha-beta pruning was created. When playing itself in Othello, the implementation saw an average speedup of ~810x compared to the naive serial approach. As the depth of the search increased, the new approach was able to maintain linear increases in runtime, while the naive approach increased at an exponential rate. This implementation has been tested up to a depth of 100, while producing precise results. Because of this testing, the implementation is stable, not producing unforeseen errors and working over a wide range of situations.

VII. **Reference**

Rocki, Kamil & Suda, Reiji. (2009). Parallel Minimax Tree Searching on GPU. 449-456. 10.1007/978-3-642-14390-8_47.

Strnad, Damjan & Guid, Nikola. (2011). Parallel Alpha-Beta Algorithm on the GPU. CIT. 19. 269-274. 10.2498/cit.1002029.

Sieg, Peter. C, (2017), GitHub repository, https://github.com/petersieg/c