Parallelization of Connect 4 Alpha Beta Pruning Algorithm with Global Transposition Table on CUDA

RICHARD WANG, University of Michigan, USA

This project report delves into the parallelization of the Connect 4 Alpha-Beta Pruning Algorithm with a Global Transposition Table using CUDA. The objective is to enhance the performance of the algorithm by leveraging parallel computing capabilities provided by GPUs. The project investigates the implementation of parallel alpha-beta pruning and introduces a global transposition table to mitigate redundant computations. Through a set of 1000 test cases, the report evaluates the effectiveness of the parallelized algorithm, comparing it to a sequential implementation. While the results show no significant improvement in terms of execution time, an increase in nodes per second exploration is observed. Additionally, the incorporation of the global transposition table yields slight speedup compared to a parallel approach that doesn't leverage the global transposition table. The report concludes with reflections on lessons learned, challenges faced, and outlines potential avenues for future research and optimization.

CCS Concepts: • **Computing methodologies → Parallel algorithms**.

Additional Key Words and Phrases: game solving, alpha beta pruning, artificial intelligence, GPU programming, CUDA, transposition tables, negamax

1 INTRODUCTION

Connect 4, a classic two-player game, serves as the backdrop for this project, where the objective is to explore the parallelization of the Alpha-Beta Pruning Algorithm using CUDA. In the realm of game-solving algorithms, Alpha-Beta Pruning stands out as an efficient approach to minimize the search space, making it a prime candidate for parallelization. The report first contextualizes the significance of this project by providing an overview of Connect 4, game-solving algorithms, and the computational power offered by CUDA.

Following an explanation of the sequential implementation of the Alpha-Beta Pruning Algorithm, the report transitions to the core of the project—the parallel implementation on a GPU. This involves an in-depth discussion of CUDA and the intricacies of adapting the algorithm to exploit parallel processing capabilities. To further optimize the parallelized algorithm, a global transposition table is introduced, aiming to reduce redundancy in node exploration.

The following sections present the implementation details, results, and analysis of the parallelized algorithm based on 1000 test cases. Although the observed results do not manifest as a reduction in execution time, the report uncovers an increase in nodes explored per second. Moreover, the integration of the global transposition table yields noteworthy improvements compared to a straightforward parallel approach.

The project report concludes with insights into lessons learned and challenges faced during the project. It also outlines potential directions for future work, recognizing the limited time and constraints to complete this project, as well as the ongoing pursuit of optimizing parallel algorithms for game-solving scenarios. This project's goal to parallelize the Connect 4 Alpha-Beta Pruning Algorithm with a global transposition table not only contributes to the broader field of parallel computing but also serves as a foundation for continued exploration of the use of global transposition tables in CUDA-based game solving algorithms.

2 RELATED WORK

There is little related work on the parallelization of alpha beta pruning for Connect 4. There has been related work on an alpha beta pruning variant known as PV-split, which better supports parallelism but is out of the scope of my project. PV-split is an algorithm involving significant recursion, and is embarrassingly parallel, making it nearly impossible to implement with the resources available to me, but a discussion of its usage is important to contrast with the results of my paper, which primarily focuses on parallelizing the Negamax variant of the alpha-beta pruning algorithm.

The article titled "Parallel Alpha-Beta Algorithm on the GPU" by Damjan Strnad and Nikola Guid, published in the Journal of Computing and Information Technology (CIT) in 2011 [6], explores the parallel implementation of the alpha-beta algorithm on GPUs. The authors compare the speed of the parallel player with the standard serial player using the game of reversi with boards of different sizes. The research aims to assess the efficiency of GPU utilization in parallelizing the alpha-beta algorithm.

Their research methodology involves implementing the parallel alpha-beta algorithm using CUDA using a PV-Split algorithm and testing its performance on the game of reversi with varying board sizes. The results indicate that GPU resources are underused on smaller boards due to search overhead and memory transfers. However, on larger boards with high pruning ratios, multiple parallel acceleration becomes feasible. The study also observes that with increasing search depth, the limits of GPU resources are met, leading to relative degradation of parallel efficiency.

The authors suggest that their findings on GPU-based PV-split alpha-beta implementation in reversi indicate the potential for efficient parallelization in more complex games. They outline future directions, including exploring hybrid CPU/GPU solutions and incorporating GPU-side transposition tables, an exploration that this project seeks to do not with the PV-split algorithm, but instead, my own parallelized version of alpha-beta pruning.

More similar studies have been done involving the PV-split algorithm. Researchers at the University of Nevada present a CUDA implementation of the PV-Split algorithm for parallelizing the game tree search in the context of the game Go [5]. The authors address the challenge of the game's large board size and exponential time complexity. Similar to the reasoning of the researchers in [6], the PV-Split algorithm is designed for two-player zero-sum games and is employed to parallelize the game tree search, aiming to reduce computation time and reach deeper levels of the tree more efficiently.

The paper concludes with results and analysis, presenting execution times and speedups for different board sizes. The authors evaluate the performance of their GPU implementation, discuss memory optimization techniques, and propose future work, including further optimizations and the exploration of multi-GPU implementations. Overall, the paper [5] contributes to the understanding and improvement of parallel algorithms for game tree searches in the challenging domain of Computer Go.

It is clear to note that much of current academia approach this problem with the PV-split algorithm, and do not approach it on a game like Connect 4. Furthermore, none of them promote the use of a global transposition table, with one [5] referencing global transposition tables as interesting future work.

Thus, I hope to implement a parallelized version of the alpha-beta Negamax algorithm rather than the PV-split algorithm on Connect 4 as well as the use of a global transposition table to help further the research on this topic with new observations.

3 CONTEXTUALIZING THE PROBLEM

The following section contextualizes the problem. First, I discusses the game of Connect 4, its rules, as well as its complextiy

3.1 Connect 4

Connect 4, also known as Four in a Row, is a classic two-player connection game in which the players take turns selecting a column to play one of their colored discs from the into a grid. The game is won by the first player to connect four of their own discs of the same color in a row, either horizontally, vertically, or diagonally.

There are several rules that players must adhere to when playing Connect 4:

(1) Connect 4 is played on a 6x7 grid, resulting in 42 positions.
(2) Two players take turns, each using different colored game pieces.
(3) When selecting a column to place a move, pieces must fall to the lowest available position within the chosen column.
(4) The goal of the game is to connect four of one's own pieces.

While Connect 4 may seem straightforward, its simplicity is deceiving. The game unfolds dynamically, requiring players to anticipate and block their opponent's moves while planning their own moves. The vertical and horizontal nature of the game board introduces a spatial element, making each move a critical decision with long-term consequences.

As a result of the game's grid structure, Connect 4 boasts a vast number of potential positions, contributing to the game's strategic depth. With each player having 21 pieces and considering the 42 available positions, the total number of possible board configurations is in the trillions.

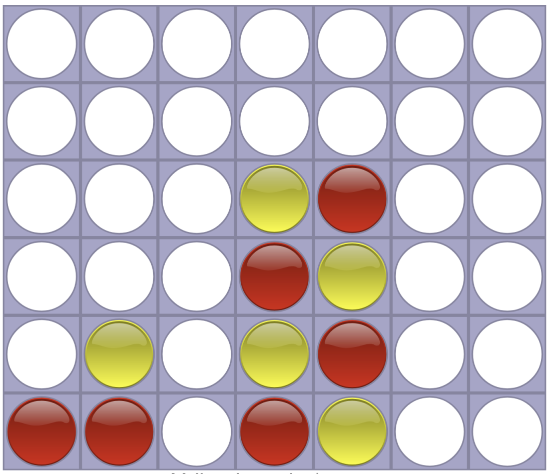


Figure 1: An example state of the Connect 4 board: one of 4,531,985,219,092 possible positions.

3.2 Game Playing Algorithms

In the realm of game-playing algorithms, the trio of Minimax, Negamax, and Alpha-Beta Pruning represents foundational concepts that have significantly shaped the landscape of artificial intelligence in gaming.

3.2.1 Minimax. Minimax, at its core, is a decision-making algorithm designed for two-player zero-sum games, aiming to find the optimal move that maximizes the potential outcome while considering the opponent's best response. This algorithm, however, faces challenges in terms of computational efficiency due to its exhaustive nature, exploring all possible moves within the game tree. The algorithm is based around the concept of recursion, emulating a depth-first search of the game tree to find the corresponding heuristic scores.

Algorithm 1: Simple Minimax algorithm pseudocode [1]

```
int maxi( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}

int mini( int depth ) {
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    for ( all moves) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

3.2.2 Negamax. Negamax is a clever reinterpretation of Minimax, simplifying the implementation by expressing the problem in terms of maximizing the negation of the opponent's score. This simplification not only streamlines the algorithm but also sets the stage for a more straightforward implementation of the algorithm.

Algorithm 2: Simple Negamax algorithm pseudocode [2]

```
int negaMax( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

3.2.3 Alpha-Beta Pruning. There is still one big issue with the previous two algorithms: they continue to search the entire game tree in order to find a score. For a game like Connect 4 with positions in the trillions, this is much too inefficient. This is where Alpha-Beta Pruning comes in.

The integration of Alpha-Beta Pruning into Minimax and Negamax introduces a crucial optimization layer. Alpha-Beta Pruning significantly reduces the number of nodes that need to be evaluated by intelligently discarding branches that cannot influence the final decision.

Algorithm 3: Alpha-Beta Pruning with Negamax algorithm pseudocode [3]

```
int alphaBeta( int alpha, int beta, int depthleft ) {
   if( depthleft == 0 ) return quiesce( alpha, beta );
   for ( all moves)  {
      score = -alphaBeta( -beta, -alpha, depthleft - 1 );
      if( score >= beta )
         return beta;   //  fail hard beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}
```

To do this, the algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.

The algorithm operates during the recursive traversal of the game tree, updating alpha and beta values as it explores different branches. When evaluating a node, if the algorithm discovers a move that is guaranteed to be worse for the current player than a move already considered, it prunes the rest of the subtree rooted at that node. This is achieved by comparing the discovered value with the current alpha (for the maximizing player) or beta (for the minimizing player) and updating the respective bound accordingly.

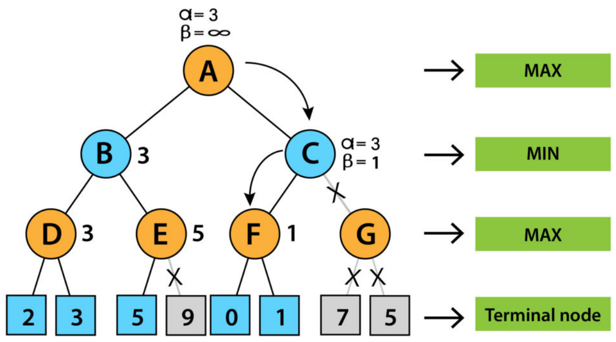


Figure 2: A diagram depicting alpha-beta pruning being completed on a tree. Branches are pruned based on the alpha and beta values. [7]

For example, when the maximizing player finds a move that results in a score higher than the current alpha, it updates alpha to the new, higher score. Similarly, when the minimizing player finds a move that leads to a score lower than the current beta, it updates beta to the new, lower score. These bounds allow the algorithm to disregard entire subtrees that cannot possibly influence the final decision, hence "pruning" the search space.

Alpha-Beta Pruning is incredibly strong in games with large branching factors, allowing the algorithm to delve deeper into the game tree without an exponential increase in computation. This makes it a perfect choice for application to Connect 4.

3.3 Transposition Tables

In the realm of game tree search algorithms, transposition tables play a pivotal role in optimizing computational efficiency. A transposition occurs when the same game position is reached through different move sequences. Game tree search algorithms such as the ones discussed earlier, can encounter a significant number of repeated positions during exploration. Transposition tables act as a memory cache, storing previously computed results associated with specific board configurations.

The primary purpose of transposition tables is to avoid redundant computations. When a position is encountered that has been previously evaluated, the algorithm can retrieve the stored information from the transposition table rather than re-evaluating the position. This not only saves computation time but also allows the algorithm to avoid revisiting the same branches of the game tree.

3.4 CUDA Programming

CUDA programming represents a groundbreaking paradigm in the realm of parallel computing, offering a robust framework for harnessing the immense processing power of GPUs beyond their traditional graphics rendering capabilities. Developed by NVIDIA, CUDA provides a platform that allows developers to parallelize computationally intensive tasks, enabling acceleration of applications across various domains.

In CUDA, the fundamental building blocks are threads, which are grouped into blocks, and these blocks are further organized into a grid. Each block is assigned to a streaming multiprocessor (SM), which is a set of CUDA cores capable of executing parallel threads. The organization of threads into blocks and grids allows for efficient parallel processing on GPUs.

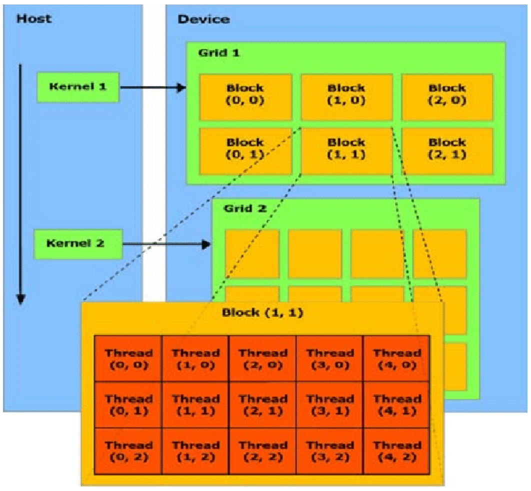


Figure 3: A diagram depicting the basic architecture of CUDA, with the abstractions of threads, blocks, and grids [4]

The CUDA architecture facilitates parallelism by breaking down tasks into threads that can be executed simultaneously on multiple CUDA cores within an SM. This structure allows for the execution of a large number of threads concurrently, taking advantage of the parallel processing capabilities of GPUs.

3.5 Significance of Parallelizing this Algorithm

Parallelizing the algorithm holds significant importance for various reasons. In the dynamic and expanding field of artificial intelligence, particularly in game tree search, continuous advancements are made to enhance performance. Game engines are continually evolving with improved hardware and innovative algorithms to stay competitive.

The specific algorithm under consideration for parallelization is alpha-beta pruning within the context of Connect 4. During the project, it became evident that implementing Alpha-Beta Pruning on the GPU poses considerable challenges. The GPU is not an ideal platform for applications requiring dynamic parallelization, and alpha-beta pruning, being a recursive algorithm, faces limitations in CUDA's programming model, which is better suited for math-intensive iterative algorithms. Each thread in alpha-beta pruning introduces new work through node exploration, and CUDA's limited support for dynamic parallelization and recursion poses obstacles in optimizing this process.

Due to the competitive nature of game engine development, there is a lack of publicly available CUDA implementations for game tree search, particularly due to the challenges posed by alpha-beta pruning and its compatibility with CUDA. Therefore, contributing to this field by introducing an algorithm and exploring ways to make it compatible with CUDA is crucial for advancing research.

Additionally, to facilitate communication between threads, a global transposition table is implemented. This approach has not been thoroughly analyzed in existing literature on CUDA game search, likely because of the lack of support for hash maps on CUDA and the extended access times to global memory. Thus, the parallelization results using this method aim to provide insights into the considerations for leveraging the speedup offered by a global transposition table in GPU game tree search programming.

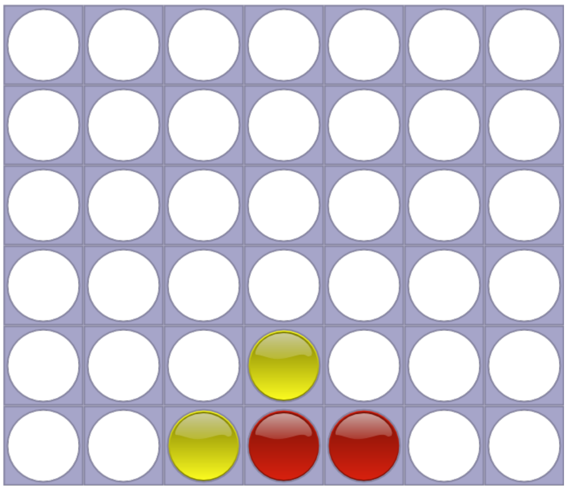
4 CONNECT 4 IMPLEMENTATION

The Connect 4 implementation was heavily inspired by the implementation provided by Pascal Pons. The description of his project can be found herE: http://blog.gamesolver.org/.

The board and its state were all represented as a class. This class had several private variables, including a board private variable, and a height private variable. First, the board was represented as a 2D array with "i" rows and "j" columns. The height array was a 1D array of length "j". The height array was used to keep track of the index of the highest available slot in the grid. Additionally, the number of moves taken to get to a specific board was stored as an integer, "moves".

This board came with its own representation in literature. The state of a board could be further represented as string of numbers. Each number would correspond to the move taken at that specific. For example, the move string "4453" means that player 1 placed their piece in column 4, player 2 placed their piece in column 4, then player 1 placed their piece in colmumn 5, and lastly player 2 placed their piece in column 3. The figure below depicts this state.

Figure 4: The visual representation of the board state "4453"



There were several functions that implemented for the board. Besides the constructors and setters, there were three important functions that would be used in the sequential and parallel alpha beta pruning algorithms:

(1) The first function was "canPlay". The "canPlay" function accepts an integer as an argument. This integer would represent the 0 based column index to play a move. This function checks whether a move cold be played based on the inputted column and would reference the height array.
(2) The next function was "play". This function, similar to the "canPlay" function, plays a move on the current board based on the inputted 0-based column index to play the move at.
(3) Finally, the "isWinningMove" function also accepts the integer column as an argument. This function is important in finding leaf nodes for game tree search and is the reason why the board state must be represented as a 2D array and not a string. The 2D array representation is needed to check for wins, as the code must manually iterate through the surrounding 8 positions to determine whether the played move wins.

Lastly, an important thing to discuss is how each position is scored:

- A positive score is assigned if the current player has the potential to win. The score is determined by counting down from 1 if they win with their last stone, 2 if they win with the second-to-last stone, and so forth.
- A zero score is assigned if the game is projected to end in a draw.
- A negative score is given if the current player is set to lose regardless of their move. The score is calculated by counting down from -1 if the opponent wins with their last stone, -2 if they win with the second-to-last stone, and so on.

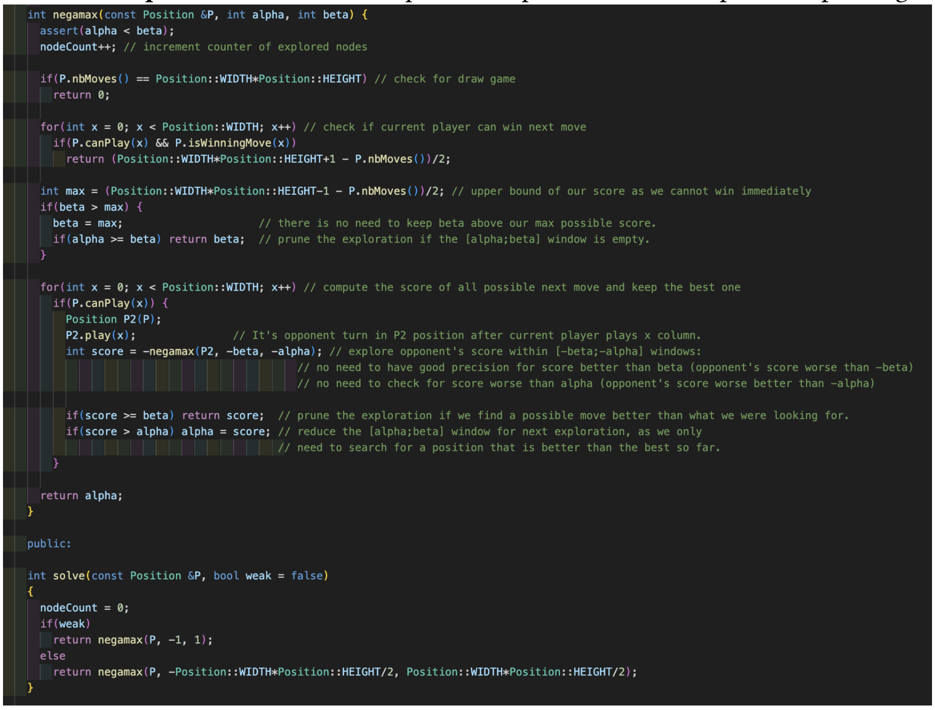
The computation for a winning score involves subtracting the number of stones played by the winner at the end of the game from 22. Conversely, the computation for a losing score involves subtracting 22 from the number of stones played by the winner at the end of the game. After playing the best move, the opponent's score is the opposite of the player's score.

5 SEQUENTIAL IMPLEMENTATION OF ALPHA-BETA PRUNING

The following section describes the sequential implementation of alpha-beta pruning.

The sequential implementation of alpha-beta pruning is the one implemented by Pascal Pons. It leverages the Connect 4 implementation mentioned from earlier to perform Alpha Beta Pruning. A screenshot of the sequential implementation can be found below.

Code Sample 1: Screenshot of sequential implementation of alpha-beta pruning



```
int negamax(const Position &P, int alpha, int beta) {
  assert(alpha < beta);
  nodeCount++; // increment counter of explored nodes

  if(P.nbMoves() == Position::WIDTH*Position::HEIGHT) // check for draw game
    return 0;

  for(int x = 0; x < Position::WIDTH; x++) // check if current player can win next move
    if(P.canPlay(x) && P.isWinningMove(x))
      return (Position::WIDTH*Position::HEIGHT+1 - P.nbMoves())/2;

  int max = (Position::WIDTH*Position::HEIGHT-1 - P.nbMoves())/2; // upper bound of our score as we cannot win immediately
  if(beta > max) {
    beta = max;                     // there is no need to keep beta above our max possible score.
    if(alpha >= beta) return beta;  // prune the exploration if the [alpha;beta] window is empty.
  }

  for(int x = 0; x < Position::WIDTH; x++) // compute the score of all possible next move and keep the best one
    if(P.canPlay(x)) {
      Position P2(P);
      P2.play(x);               // It's opponent turn in P2 position after current player plays x column.
      int score = -negamax(P2, -beta, -alpha); // explore opponent's score within [-beta;-alpha] windows:
                                  // no need to have good precision for score better than beta (opponent's score worse than -beta)
                                  // no need to check for score worse than alpha (opponent's score worse better than -alpha)

      if(score >= beta) return score;  // prune the exploration if we find a possible move better than what we were looking for.
      if(score > alpha) alpha = score; // reduce the [alpha;beta] window for next exploration, as we only
                                // need to search for a position that is better than the best so far.
    }

  return alpha;
}

public:

int solve(const Position &P, bool weak = false)
{
  nodeCount = 0;
  if(weak)
    return negamax(P, -1, 1);
  else
    return negamax(P, -Position::WIDTH*Position::HEIGHT/2, Position::WIDTH*Position::HEIGHT/2);
}
```

The sequential implementation of alpha-beta pruning is the C++ implementation of the alpha-beta pruning pseudocode algorithm, with minor changes to support the Connect 4 interface described earlier.

6 PARALLEL IMPLEMENTATION

The following section discusses the parallel implementation that I designed for the alpha-beta pruning algorithm, as well as how threads communicate with each other through a global transposition table.

6.1 Parallel Alpha Beta Pruning Algorithm

The solution I devised to parallelize alpha beta pruning on CUDA involves utilizing a stack to simulate function calls of negamax. This implementation had to be done as CUDA does not provide support for deep levels of recursion. Furthermore, since the CUDA device only supports C, and there is no native support for stacks in C, my own stack structure had to be created. This includes designing a stack entry with a flag for whether the function calls were going deeper, or going up. The resulting code, although much longer and complex, results in the same functionality as the recursive version.

My parallel alpha beta algorithm on CUDA works as follows:

(1) Initialize a stack. Take the initial position entered and put it into a stack entry. Flag this stack entry as going down and push it onto the stack.
(2) While the stack is not empty:

(3) If the stack entry is going down, first check if the current position is a leaf node. This includes checking if it is a draw, checking if a player can win at this position, or if the alpha value is greater than the max score possibly obtained at this position. If the position is a leaf node, the flag the stack entry as a return entry an continue to the next iteration of the loop. This part of the code emulates the checking of leaf nodes at the start of the sequential algorithm.
   Otherwise, if the node is not a leaf node, that means we need to continue to go down. In this case, we generate a list of all possible moves at this position. We then take the first move, copy the position, play this position to create a new position, and then put this new position into a new stack entry. We then flag this stack entry as going down and push this stack entry onto the stack. This part of the code emulates the first call of the Negamax function in the for loop in the sequential implementation.
(4) If the stack entry is going up, we first pop it off of the top (since it is emulating the return stack frames) and then obtain the next stack entry on the top. We check to see if the returned score is greater than the stack entry's beta, and if it is, we flag the stack entry for return in a similar process as described above. Otherwise, if this returned score is greater than the stack entry's alpha, then we update the alpha. We then find the next move that is possible at this position. If no moves are possible, we flag the stack entry for return, and return the alpha value. This part of the code emulates the remainder of the for loop iterations of the sequential algorithm.

To conclude this subsection, although this algorithm is complex and involves a self-managed stack system, it emulates the recursive calls of the original recursive negamax accurately, and in a way that can be utilized on CUDA device threads. How thread communication is implemented will be descried in the next few sections.

6.2 Thread Task Queue Population and Host Function

Firstly in order to make this parallel, several steps were taken. Let us call the algorithm described in the previous section as "parallelGTS".

First, a breadth first search of the game tree search was performed on the host device. This breadth first search would run until a certain depth S. It would obtain the nodes that still had successors at this depth and populate them into a position array.

This position array was then copied to the GPU alongside a corresponding score array. The GPU architecture was set up in a very specific way: blocks were of size 256 threads. There would be one thread for each position that was copied to the GPU. The grid size would be adjusted to ensure that there was enough space for all blocks. The kernel function would then be called. Each thread would access the position array based on their global index to discover which position they would calculate the negamax score for. Once the thread completed its calculation, it put its score in its corresponding index in the score array.

After all threads were completed, the score array was copied back to the CPU. A sequential negamax would then be run until depth S, however, rather than continuing the search at depth S, it would pull the found scores from the score array.

6.3 Thread Communication: Global Transposition Table Implementation

An optimization added to this algorithm to allow thread communication is another common optimization found in game solving algorithms was implemented: the transposition table. The transposition table and its purpose was described in section 3.3.

In order to implement a transposition table that was accessible by all threads, a global hash table structure was implemented. NVIDIA provides no official support of a hash map structure for devices, so I decided to design one. An array was used as the underlying structure to emulate the hash-map. This array would store a custom object called the "PositionEntry". This position entry would hold a position, the score of that position, as well as an integrity hash value. A hash function was designed to hash a position based on its 2D board state. Another hash function was designed to hash a position and value to ensure the integrity of the PositionEntry in order to deal with race conditions.

Slight modifications were done to the stack emulation algorithm. The global hash map would be updated whenever a return in the stack emulation occurred. The global hash map would be read from as the stack entry was going down, and would be done after leaf node checks. In order to read from the hash table, a thread had to first calculate the hash of the position it was currently expanding. It would then access the array based on this hash. After the entry was found, similar to cybersecurity integrity preservation methods, the thread would then calculate the integrity hash of this position. If it did not match the integrity hash stored in the PositionEntry, that means a race condition caused a value modification, resulting in an unusable hash entry and a transposition table "miss". In the event this occurred, reading the hash entry was skipped.

A workaround situation like this was utilized as there is no CUDA support for hash maps and no CUDA support for an atomic modification of an array. These safeguards, with principles leveraged from cybersecurity, were taken to ensure that reads where something was overwritten at the same time would not go through. This implementation provided correct results.

7 RESULTS

7.1 Setup

The sequential code was run locally on a 2020 M1 Macbook Air. The parallel code was run on the Great Lakes cluster, which boasts an NVIDIA Tesla V100 PCle for GPU programming and execution. Both code was run on 1000 test cases, containing positions where there were less than 14 total moves remained. The test cases were limited in terms of total moves remaining as positions with more than 14 moves remaining would take longer than the 5 minute limit for both sequential and parallel implementations. The implementations would be measured on three different variables: time to run, number of nodes explored, and number of hash hits. These three variables are all important to look at in terms of measuring the performance of these algorithms and analyzing speedup.

The timing was measured for the entire execution process for the sequential algorithm. For the parallel algorithms, the time measured was also the entire algorithm, except the time to allocate and copy data into the GPU device was subtracted.

The number of blocks and threads used in the parallel implementations was explained in section 6.2.

The serial algorithm was run once. The parallel algorithms were ran with two different *S* values:

- S = 1, representing immediate division of work between threads of the child nodes. The branching factor of the Connect 4 tree is 7, one successor for each move possible. Thus, the maximum amount of threads launched with S = 1 is 7.
- S = 4. The BFS would run until a depth of 4 and find all remaining nodes with successors. The maximum amount of threads launched with S = 4 is 2,401

The results of the 1000 test cases were aggregated into the following statistics, with a corresponding acronym:

- *AT:* Average time per test case in microseconds

- *TT:* Total time to run all test cases in microseconds
- *ANE:* Average nodes explored per test case
- *TNE:* Total number of nodes explored across all test cases
- *NEMS:* Nodes explored per microsecond
- *ATH:* Average number of transposition table hits (for Transposition Table Implementation)
- *TTH:* Total number of transposition table hits (for Transposition Table Implementation)

The tables for the results can be found below:

Table 1: Aggregate Results of 5 Tested Algorithms on 1000 Testcases

	AT	***TT***	***ANE***	***TNE***	***NEMS***	***ATH***	***TTH***
Sequential	283.591	283591	77.351	77351	0.273	-	-
S = 1, Parallel, No Table	402.254	402254	2544.273	2544273	6.325	-	-
S = 1, Global Transposition Table	399.448	399448	3376.461	3376461	8.453	1.049	1049
S = 4, Parallel, No Table	3804.356	3804356	3599.069	3599069	0.946	-	-
S = 4, Global Transposition Table	3774.369	3774369	4293.899	4293899	1.138	7.611	7611

8 DISCUSSION

The following section presents a discussion of the results found, explanations of any observed trends, discussion on the difficulty of this project, and lessons I learned while completing this project.

8.1 Results Analysis

8.1.1 Parallel vs. Serial. Several discernible trends emerge when comparing the sequential algorithm with its parallel counterpart.

Firstly, there is a notable increase in the time required to execute the algorithm and obtain the correct result across all parallel algorithms. This phenomenon may be attributed to the initial breadth-first search (BFS) undertaken to populate the thread task pool. Consequently, there is an overhead when the sequential algorithm finds the solution within the first S levels, while the parallel version continues its work regardless of whether the solution is within those S levels. Additionally, the increase in time can be linked to the storage and copying of objects due to the stack implementation and the optimizations for recursion introduced by modern compilers. Unfortunately, from this observation, it becomes apparent that parallel algorithms do not provide any speedup compared to their sequential counterparts.

This rise in total explored nodes can be explained by the aforementioned factors. It is noteworthy, however, that the number of nodes explored per microsecond is greater for parallel algorithms than for sequential ones. This is logical, considering that in parallel functions, the increased number of threads working simultaneously leads to a more extensive exploration of nodes, resulting in a higher number of nodes explored per microsecond. Therefore, positive scaling is observed in terms of the number of nodes explored per microsecond, even though there is no improvement in overall performance when considering execution time.

8.1.2 No Table vs Global Transposition Table. A clear correlation emerges between the no use of the global transposition table and use of the global transposition table, leading to a reduction in the overall execution time of the algorithm. This reduction is logical since the threads no longer need to delve deeper into the tree and explore additional nodes when a transposition hit occurs. Instead, they can directly utilize the previously determined value, streamlining the computational process.

8.1.3 S = 1 vs S = 4. Finally, there is an observable trend when comparing the use of S = 1 with S = 4. The most significant observation is that, with S = 4, the time required for the search increases.

This can be explained by considering that if the solution is found in levels preceding S = 4, the S = 1 algorithm would discover the solution more rapidly due to the efficiency of alpha-beta pruning. In contrast, threads in the S = 4 algorithm navigate paths of node exploration that are less likely to yield crucial results and often involve traversing pruned paths. Despite the lack of importance in these explorations, the threads must complete the pruning work, contributing to the substantial increase in time from S = 1 to S = 4. Consequently, a crucial parameter to fine-tune in implementing this algorithm is the S level, as adjustments to this value result in dramatic changes in execution time.

This also explains the increase in total number of nodes explored between the S = 1 and S = 4 solution. As there are threads in the S = 4 solution that are further going down branches that would've potentially been pruned in lower S values, the number of nodes explored continues to increase.

8.2 Challenges faced during parallelization

The parallelization of this algorithm presented several challenges, beginning with my limited familiarity with the CUDA language. Despite having completed a smaller CUDA assignment involving stenciling, this project posed a significantly higher level of difficulty. Consequently, I had to delve into additional CUDA techniques to effectively tackle this project.

As discussed in section 3.5, the algorithm in question involves tree search, a process that contrasts with the typical use of the CUDA Programming interface designed for math-intensive graphical computation. The alpha-beta pruning algorithm, being a recursive tree search, presents a dynamic work generation problem as each exploration uncovers varying amounts of work. While CUDA has introduced support for dynamic parallelism, this approach proved too simplistic for the complexities of this project. In my experimentation, recursion and dynamic parallelism failed to progress beyond depths of 2.

Converting the alpha-beta pruning algorithm into a form compatible with CUDA proved exceptionally challenging. While stack emulation of recursion is a known technique, translating the intricate alpha-beta pruning algorithm into this stack-emulated version posed difficulties. The algorithm's complexity, multiple return points, and the need for return values in later stages made debugging and completion challenging. Moreover, the conversion had to be done in C to facilitate GPU execution, lacking the convenient data structures provided by C++.

Despite these formidable challenges, they were eventually overcome, leading to the successful implementation of the parallel algorithm.

Creating the thread work array also posed challenges. Various implementations, including running Negamax to depth S before thread work, were attempted to avoid unnecessary work compared to the sequential algorithm. However, due to the nature of alpha-beta pruning and the importance of each node at depth S with a successor, a BFS approach was adopted, prioritizing correctness at the expense of performance.

Another difficulty in the creation of this algorithm was the creation of my own transposition table system. CUDA does not provide official support for hash tables. Another layer of difficutly for this specific problem is that C also does not provide support for hash tables. Therefore, I embarked to create my own hash table system. The underlying structure of the hash table was an array, and I had to design a hash function that provided minimal conversion. An issue that was quickly ran up on was the handling of race conditions. Due to the nature of alpha beta pruning, I allowed writes to happen immediately, however, reading became an issue. If two threads were writing the value at the same time, the written value may be modified or different in the middle. Thus, I adopted the integrity hash approach from cybersecurity to ensure that this would not affect the correctness of the algorithm when performing reads.

These are just several of the challenges I faced in the parallelization of this algorithm, and after putting in days and weeks of hard effort, a solution was finally implemented correctly with subtle improvements in nodes explored, despite the performance decrease.

8.3 Reflection

The journey of parallelizing this algorithm has been riddled with challenges, each providing me with valuable insights and lessons. The first lesson learned revolves around the necessity of improving my expertise in the specific parallel programming language, in this case, CUDA. While prior exposure in a smaller assignment proved helpful, the complexity of this project underscored the importance of mastering additional CUDA techniques to address the intricacies of parallelizing a complex algorithm effectively. I learned a lot about CUDA programming in undertaking this project.

The experience also shed light on the crucial consideration of aligning the algorithm's nature with the design principles of the chosen parallel programming interface. Tree search algorithms, such as the alpha-beta pruning employed here, inherently pose challenges in a CUDA environment primarily designed for math-intensive graphical computations. Understanding these mismatches in computational paradigms is crucial for making informed decisions about the feasibility and complexity of parallelization efforts. I learned a lot about performing tree search on CUDA.

Moreover, the project highlighted the limitations and challenges associated with recursive algorithms in CUDA, despite the introduction of dynamic parallelism support. Experimentation revealed the shortcomings of relying solely on recursion and dynamic parallelism, letting me realize that alternative strategies must be explored to overcome the limitations encountered.

The process of converting the alpha-beta pruning algorithm into a CUDA-compatible form emphasized the need for an understanding of both the algorithm's intricacies and the capabilities of the programming language. Translating a recursive algorithm into a stack-emulated version in C required attention to detail, particularly considering the multiple return points and the need for return values in subsequent stages of the algorithm.

Creating a thread work array brought forward a lesson on trade-off between performance and correctness. The attempt to minimize unnecessary work through various implementations, including running Negamax to depth S, underscored the importance of prioritizing correctness, even if it comes at the expense of performance.

The development of a custom global transposition table system introduced challenges related to the lack of native support for hash tables in both CUDA and C. Crafting a hash function with minimal conversion and addressing race conditions during reads improved my innovative problem-solving, emphasizing the importance of adaptability in overcoming platform constraints. I learned a lot about the importance of locks, race conditions, and thread synchronization.

9 CONCLUSION AND FUTURE WORK

In the pursuit of enhancing game-playing algorithms, this project has delved into the intricate world of game solving, leveraging the power of parallel programming and optimization techniques. The focal point was the parallelization of the Alpha-Beta Pruning algorithm using a global transposition table in CUDA programming, a project that sought to address the computational challenges posed by the game's complex state space.

The choice of Connect 4 as the game for these advancements provided a tangible and straightforward context for the application of parallel computing principles. The complexity of Connect 4's branching factor and the trillions of potential game states underscored the necessity of innovative approaches to overcome computational bottlenecks. The integration of CUDA programming introduced a paradigm shift, tapping into the parallel processing capabilities of modern GPUs to expedite the exploration of the game tree.

The resulting implementation contributed valuable insights into the broader landscape of parllel game-solving algorithms, especially with an analysis of the use of global tranposition tables in CUDA, something that has not been wideley investigated at all.

From completing this project, I've learned many valuable lessons regarding parallelization techniques on CUDA. The challenges I encountered during the parallelization of this algorithm highlighted the importance of continuous learning, adaptability, and an understanding of both the algorithm and the chosen programming environment. Despite the complexities faced, the successful implementation underscores the resilience and persistence required in tackling this intricate problem in parallel computing.

Due to the limited time constraints, there is still some more future work that can be done to improve and optimize the algorithm. Firstly, a vastly improved stack emulation algorithm may result in an improved alpha-beta pruning execution time on the GPU. Furthermore, if there was a way to quickly find a solution to the alpha-beta pruning game tree before the depth of S on the host, it should be implemented to ensure a performance faster, or as fast as the sequential variant. Moreover, once hash map support has been added to CUDA, the usage of an industry-standard optimized hash map will provide much more improved results warranting further exploration. Regardless, this project and its analysis into the use of a global hash map as a global transposition table in CUDA has been informative and has opened the door for more future research.

ACKNOWLEDGMENTS

Pascal Pons - for his straightforward guide on implementing Connect 4 and its game solving agents.

REFERENCES

[1] [n.d.]. https://www.chessprogramming.org/Minimax
[2] [n.d.]. https://www.chessprogramming.org/Negamax
[3] [n.d.]. https://www.chessprogramming.org/Alpha-Beta
[4] Juan Fernández, Manuel Acacio, Gregorio Bernabé, José L. Abellán, and Joaquín Franco. 2008. Multicore Platforms for Scientific Computing: Cell BE and NVIDIA Tesla. 167–173. https://doi.org/10.13140/2.1.3822.9445
[5] Christine Johnson, Lee Barford, Sergiu M. Dascalu, and Frederick C. Harris. 2016. CUDA Implementation of Computer Go Game Tree Search. In *Information Technology: New Generations*, Shahram Latifi (Ed.). Springer International Publishing, Cham, 339–350.
[6] Damjan Strnad and Nikola Guid. 2011. Parallel alpha-beta algorithm on the GPU. In *Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces*. 571–576. https://doi.org/10.2498/cit.1002029
[7] Great Learning Team. 2023. Alpha beta pruning in ai. https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/