Jonathan Lehman

Connect Four AI

Project Description

The problem I chose to parallelize on the GPU was a Connect Four Artificial Intelligence (AI). Before I outline my algorithm and the results, I will begin with some general background information.

With the introduction of computers came great progress in the field of game theory, the study of strategic decision-making. Machines provided greater computing power and algorithms were developed to harness this new computing ability. Specifically, the field of combinatorial game theory, sequential games with perfect information, such that both players in a two-player game take turns making moves towards a clearly defined winning position, progressed from this development. Connect Four falls under the specific game type of zero-sum games, where the utility of either player can be mathematically represented and the gains or losses of one player is balanced out by the losses or gains of the other player. Other games that fall under the category of zero-sum games are Tic-Tac-Toe, Checkers, and Chess. Applying game theory to computer algorithms, allows for the creation of artificial intelligence to enable computers to play these games against humans (or other computers).

Since computers are capable of processing a lot of data very quickly, the typical approach taken is a brute force approach where each possible combination of moves that can be made from the current board configuration is evaluated, and the one with the highest utility is chosen as the best move and subsequently the one that the computer makes. Though this approach sounds simple, it is expensive. Tic-tac-toe, the simplest of the zero-sum games has 9! possible game states, Connect Four has 10^{13}, Checkers has 10^{18}, and Chess has 10^{50}. All of these games, with the exception of Chess, have been solved, meaning that given any board state, the outcome of the game (win, loss, or draw) can be predicted if each side plays optimally. Connect Four was chosen as the basis of this project since it has a relatively large number of states, but has simpler game logic than Checkers or Chess, making the GPU implementation easier, as the GPU implementation is the focus of this project, not the specific game's logic and rules. Though Connect Four was the basis of this study, the parallelization discussed later can be applied to any of these games.

The typical approach to determining the next best move requires a few fundamental pieces of information and operations. The game state is utilized in most algorithms in some manifestation, typically the location of the pieces and other necessary data to determine potential moves and measure utility is stored in an

object (or struct in this case). Negamax and Minimax are the most common algorithms used to evaluate the next best move. The key feature of these algorithms is the evaluation function, the function that measures the utility of the current board configuration. In Chess, and even in some Checker's implementations a hash table is used to quickly determine the outcomes of many known end games, a much smaller subset of the total game states.

Let's examine the Minimax algorithm as it is the most commonly used for zero-sum games. In order to make sense of the many potential future game states that need to be examined, the game tree structure must be explained. Though game states are not typically stored in a tree structure, as that is very expensive, each potential game state is observed recursively in a tree like structure. Essentially, the current game state (prior to the computer making a move) is passed into the algorithm, which then branches out into a new game state for each possible move, and these new game states each branch out into new game states for each possible move and so on. In the tree structure, each game state is a node, and each move is an edge. Figure 1 (below) is an example of what this tree structure looks like for a Tic-tac-toe, though it is simplified (the first node should branch out into 9 possible moves).

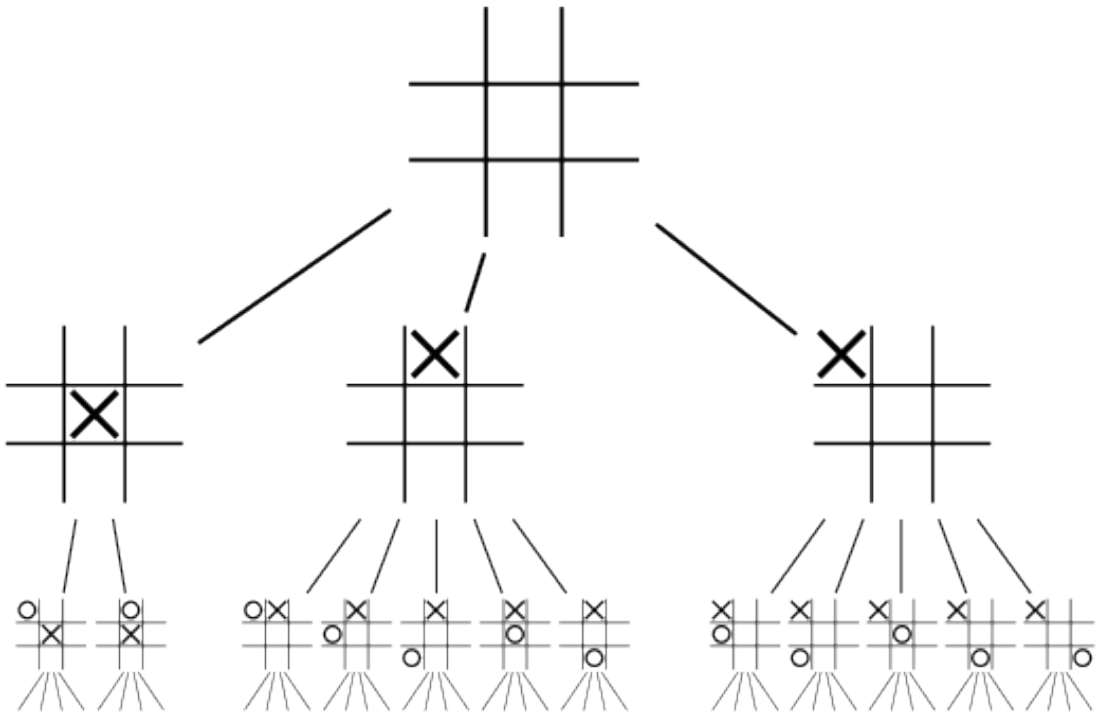

Figure 1

The overall process of the Minimax algorithm works as follows. The current game state, set up of the board and pieces, is passed into the algorithm, along with the maximum depth for the recursive algorithm to go (in larger cases like Checkers

or Connect Four, it would take far too long for each game to be "played out" in a real-time game situation against a human player), and the player to be maximized, that is at the first level of recursion, the player who is currently looking for the optimal move. In general, the essence of the algorithm is that it rotates maximizing, and minimizing the current players move. For example, if it is X's next move that we are concerned with in a game of Tic-tac-toe, the first level of recursion would maximize X's next move (based on utility), then the next level would minimize O's next move, then the next would maximize X's next move and so on. The following is pseudocode for the Minmax algorithm.

```
int minimax(state, depth, maxPlayer)
     if(depth == 0 || gameOver(state))
          return evaluate(state, maxPlayer)

     best = -INFINITY

     for move in getMoves(state)
          score = minimax(state, depth, maxPlayer)
          if score > best
               best = score
               bestMove = move
     return best
```

To explain the Minimax psuedocode, the base cases are first checked, if the algorithm has reached the max level of recursion, or the game is over by a win or draw, the evaluate function is called to return the current utility of the board with respect to the maximizing player. The specifics of the evaluation function will be covered shortly, but for now just keep in mind that it returns the utility of the board. If the base case is not reached, the algorithm iterates through all possible moves, and calls the Minimax algorithm recursively to get the score, or utility of making that move. If the returned score is higher than the old best score, which starts at negative infinity, the score becomes the best score, and the best move is recorded as the move that led to that score. Essentially the tree is explored to the leaves, either max depth or end games, and the utility from these board states is passed back up the tree recursively until the best move is found. In Tic-tac-toe the evaluation function is very simple, return 1 for a win, -1 for a loss, and 0 for a draw. See Figure 2 (below) for an illustration of how this algorithm would work for Tic-tac-toe.

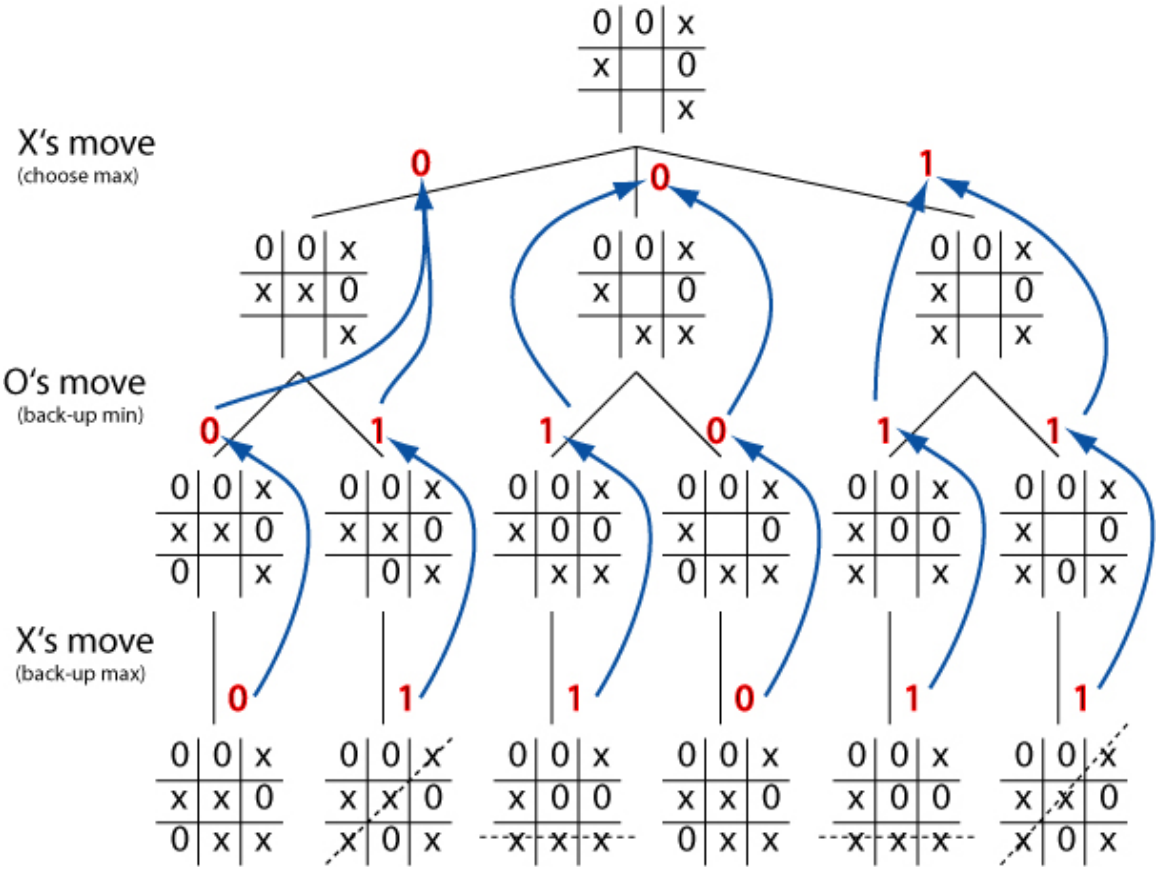

Figure 2

Despite Tic-tac-toe's simple evaluation function, Connect Four is not as simple. The evaluation function I used was to give higher value to squares in the center of the board, and to make longer connects worth more, such that three in a row is seen as better than two in a row. Making longer connections worth more is intuitive as the purpose of the game is to get four in a row, and squares in the center have a higher probability of either blocking an opponent or making connections.

With this general knowledge, it is simple to create an AI to play Connect Four on the CPU, but the trick is to parallelize the work being done to make it more efficient on the GPU using CUDA.

Algorithm Description

The simplest approach to parallelizing the Minimax algorithm on the GPU would be to generate the next possible moves, and run Minimax on each of them in parallel in a different thread. This could even be taken further and Minimax could be run on the CPU until a specified number of game states need to be explored (probably a relatively large number, though getting to that would not take long, even on the CPU), and run all of these in parallel in different threads. The problem with this approach is that recursion is not supported, which is required for the

Minimax algorithm. I was able to get an implementation working by fooling CUDA by using templates, such that a new method is created for each recursive call, however, the resources on the GPU are limited, and I was unable to run this parallelized Minimax function with a depth over 3, meaning the computer only looks 3 moves ahead, which simply would not create a smart AI, and would have no real benefits of running on the GPU over running on the CPU. Instead of using the traditional Minimax approach, I sought a way to produce a similarly powerful AI that does not rely on recursion, and can be parallelized.

To make move generation highly parallelizable on the GPU, I utilize the block and thread IDs to generate moves. By making the grid and block sizes multiples of the number of columns (7 in this case), which is the number of possible moves that can be made on the board, I can generate one move per power that 7 is raised to for each the block and thread IDs in the X and Y directions. For example, if I have blocks with 7x7 threads, I can generate 2 moves based on the thread ID, and if the grid is 7^3x7^3, 6 moves can be generate based on the grid ID, meaning a total of 8 moves can be generated instantly in parallel for each of the 7^8 threads. This allows for maximum parallelization, though some of these "move sets" (the set of moves unique to a thread based on block and thread IDs) are invalid because they may contain a move that is no longer valid due to a full column, but this problem can easily be taken care of by checking if a move is valid, and ignoring any move sets with invalid moves.

Though parallel move generation is simple using block and thread IDs, the final thing needed is a non-recursive method of utilizing the evaluation function. The first general strategy that I implemented was applying all of the moves in a valid move set, comparing the utility of each resulting game state, and selecting the move with the best utility. This worked fairly well, but was naïve in its logic. The AI was optimistic, and selected the move with the highest utility, which generally meant that this move set contained the current player's best moves and the opposing player's worst moves, essentially assuming that the opponent would make bad moves. Though the timings were good, the AI did not typically win or force a draw.

Since this was clearly not a good solution, I sought to create a new strategy that would provide the AI with the best move, not the best move under the best circumstances. In this strategy, I decided to focus on the first move being made, as this is the move that we are currently trying to optimize. As with the previous strategy, each valid move set is applied on board state, but then the utilities with the same first move are summed together to get a total utility for the next move. The resulting utility totals for first moves are than compared (7 in this case because that is the number of columns and possible first moves), and the one with the highest utility is selected. One addition to this algorithm, is that if the current player has a move that will result in a win on their next turn, that will be the move assigned, and if the opponent can win in one move, the current player will make a move to block the win, even if there are moves that technically have higher utilities because of the evaluation function. Shared memory is utilized to hold the utilities per move set, as

well as each move set, so that memory access is fast to these commonly used data, and because there are occasions in the algorithm when other threads in the block need to access memory modified by another thread.

The algorithm will now be described using pseudocode. The full kernel and device functions' code can be viewed in the attached cuda file beginning with line 710 to the end. In this optimal example, the grid size is 7^3x7^3 and the block size is 7x7, though the algorithm is flexible to support other block and grid sizes. This will be discussed in later detail in the timing section, as it is not essential during the algorithm description.

This first part of the algorithm essentially gets the utility score per move set, which is done in parallel in each thread.

```
__shared__ moves[tx][ty][numMoves] = generateMovesForEach()
__shared__ evals[tx][ty][numMoves] = 0

for move in moves
     if canMove(board, move)
          doMove(board, move, player)
          if noWinner(move)
               evals[tx][ty][0] += evaluate(board, player)
          else
               move = blockOrWin(move, board, player)
          changePlayer(player)
     else
          evals[tx][ty][0] = -invalidScore

syncThreads()
```

The first thing that is done is generating the move set based on block and thread IDs. Though this is simply done with modulo arithmetic and for loops, the pseudocode will be simplified and simply state that these moves are being stored to shared memory to be used later. The rest of the work described in this section of the algorithm is done in parallel by each thread. It iterates through each move and checks if the move is valid. If it is not valid, it is marked with an invalid score and ignored later on. If the move is valid it is applied to the current board state, that is the move is made. Then it checks if there is a winner, if there is no winner from that move the evaluation function determines the current utility and adds this to the utility total. If a winner was found, the move is set such that it either blocks the opponents win, or causes the current player to win, depending on who won on the previously examined move. The player is than changed and the process is switched. It should be noted that the evaluate function, takes account of the maximizing and minimizing player, such that the utility of the minimizing player is negative and brings down the total utility, while the utility of the maximizing player is positive

and increases the total utility. Lastly the threads are synced to ensure each thread is done before moving onto the summation of these values.

The next part of the algorithm sums up the thread utilities for each starting move for each block to get the block utility sums. This makes use of shared memory.

```
if threadID == 0
     for each thread in block
          localArray[index] += evals[firstMove]

     scoreArray[index] = localArray[index]
syncThreads()
```

This section of the algorithm is part of the summation part of the overall algorithm, which essentially parallelizes the summation of the utility totals found by each thread in the first section of the algorithm. Each block sums up these items in the block's shared memory, and stores them to global memory. It should be noted that a local array is used to store the sum and these totals are stored to a global array (global so it can be accessed later), to reduce the amount of costly accesses to global memory.

The next section is rather clever, in that it parallelizes more of the summations to increase the efficiency. Rather than summing all of the block sums based on first move in one thread in a loop, the work is split up to get a sub-grid sum of utility, which is then summed later on to get the total utilities for each first move. This is done by using all of the threads in the first block, block 0, to do part of the summation of the global memory array.

```
if blockID == 0
     work, start //based on threadID

     for start to (work + start)
          evals[move] += scoreArray[index]

syncThreads()
```

The amount of work for each thread to do, and the index for the thread to begin summing is calculated based on the thread ID. Each thread then does its sub-sum in parallel, reusing the evals shared memory as it is no longer needed, and by reusing this shared memory it is more efficient than using other shared memory or global memory.

The last part of the summation process is to get the grid sums for utility, which is easily done by adding the sub-sums from the last step.

```
for threads in block0
     finalScores[move] += evals[move]

syncThreads()
```

This section barely needs any explanation, as it is rather intuitive. The sub-sums, which are in shared memory, are added up and stored to an array (by thread 0 in block 0), finalScores, such that finalScores is an array of length columns (the number of possible moves, in this case 7) with the total utilities for each possible move by the current player.

Lastly the utilities in the array are compared, and the best overall move (the index of the highest utility) is returned. This work is also still in the scope of thread 0 in block 0.

```
bestOverallMove
bestScore = -invalidScore

for each start move
     if finalScores[startMove] > bestScore
          bestScore = finalScores[startMove]
          bestOverallMove = startMove

move = bestOverallMove
```

With the returned move (the move is copied back to CPU memory form the GPU), the rest of the game framework runs on the CPU, and prints out the board states as well as timings in the terminal.

For any specific details of the algorithm, the attached code can be examined, starting at line 710 until the end of the CUDA file.

Results

The implementation was tested with various block and grid sizes, though these numbers were very limited by system resources and the fact that block and grid sizes have to be multiples of 7, the column size, in order for the move generation to work correctly. The GPU implementation with the best timings ran with a depth of 8 (generated 8 moves based on block and thread IDs), and had a grid size of 7^3x7^3 and a block size of 7x7. The problem is that not all the resources were utilized to their full potential, as the number of threads was limited on the GPU. It would be ideal to have a block size of 49x7 and generate 3 moves via thread IDs, but this caused the shared memory to be to high, which is why the grid sizes are so large to make up for this. However, when the grid size was increased to 7^4x7^4 to generate 8 moves with the block IDs and 2 moves with the thread IDs for a total of 10 moves,

the GPU timings significantly increased to about 30 seconds because of the size of the nested for loops and amount of data being compared and processed. The CPU implementation with a depth of 10 was faster than this with a time of about 12 seconds. Despite this, looking 8 moves ahead provides a very challenging AI for Connect Four, though for other games such as Checkers or Chess, a new strategy would probably need to be used.

This implementation uses 3087 bytes of shared memory per block, 31 registers per block, 168 bytes of local memory per block, and 6.28 megabytes of global memory overall. Based on the CUDA Occupancy Calculator provided by NVIDIA on their website, only 25% of the each multiprocessor is being utilized, though 4 out of the 8 possible blocks are on a multiprocessor at a time. This does not sound very good, though the move generation based on thread and block IDs requires a lot of shared memory to store the parallel data, and is limited by the number of threads per block, 49, which is nowhere near the capacity of 512. The failure of this strategy to use all of the threads in the block, means that warps are not full being taken advantage of, which is why the occupancy calculator indicates that this setup is not ideal. To improve this, more threads would need to be used per block, which means a new strategy to find the best move in a non-recursive, and easily parallelizable manner would need to be developed.

Despite these dismal numbers, this implementation is extremely parallel as 7^8 move sets are being examined at a time by each thread, though some of these are invalid because they require moves on columns that are already full, and therefore are not examined, which is a waste, but is made up for by the amount of parallelization being done. This also makes such a brute force method more acceptable to use.

When run on the William & Mary Sciclone cluster, specifically hurricane, the tests revealed that the GPU implementation was much faster than the CPU implementation, despite the fact that the GPU's multiprocessors was not being fully utilized (each timing is the average of 30 runs). To decide the first move on an empty board took **1.956** seconds on the CPU, and only **0.506** seconds on the GPU. However, to find the last move on the board, it only took the CPU **0.00002** seconds and the GPU **0.132** seconds, which is due to the fact that the CPU implementation does not actually recurse to the depth of 8 because it finds that there is only one valid move initially, while the GPU still goes through all of the work and does not exit early because I was unable to get it to exit the kernel early and return the correct move (which is a problem I know several other people had in the class). The average times (the average of times for each move in a game) had the CPU at **1.643** seconds and the GPU at a much faster **0.327** seconds.

This implementation was very parallel, and despite some inefficiencies and improvements that could be made (discussed in the conclusion of this paper), the GPU implementation created a strong AI that was capable of finding the next best

move in an average of 20% of the time it takes the CPU, which can be viewed as a success.

Conclusion

The GPU implementation of a Connect Four AI was more efficient than its sequential counterpart on the CPU, though there are many areas where it could be improved. To decrease the memory footprint required, I wanted to use bitboards to represent the board, which would also make the finding moves and other functions faster as they would only require bitwise operations, and all potential moves could be found on the board simultaneously. Bitboards are typically used on Chess and Checkers because these have even large memory footprints and are slower if represented with arrays. I did not have time to make this change, as it would have also required a lot of change in game logic algorithms for move generation and the evaluation function.

Another improvement would be to change the method of move generation such that more threads per block can be used. This is something that I was unable to figure out how to do with the time constraints, and the only fix might be to take an entirely different approach to find the next best move. Rather than generating the moves based purely on block and thread IDs, the total amount of work could be divided evenly between more threads, such that the full 512 threads per block are utilized and the use of less shared memory so more blocks could fit on a multiprocessor (each could use no more than 2000 bytes so that the maximum of 8 blocks could fit on a multiprocessor, rather than the 4 that are on a multiprocessor with the current implementation). The trick with this would be coming up with a method for dividing up the work, the work being the game tree with a depth of 8, or even more if the GPU has resources for it.

Further research could take some of these same game theory techniques and parallelize something with an even larger total number of possible game states like Checkers or Chess. Despite, some of the problems mentioned and improvements that could be made, the implementation of Connect Four AI using CUDA on the GPU was overall successful, and with the computing and speed benefits of the GPU with its large parallel computing capability, it is clear that is a relevant and useful architecture to make use of for problems that can be broken up into parallel parts.