Monte Carlo Tree Search on CUDA

Authors:

Shlok Agarwal: ashlok@umich.edu
Karan Anand: karanand@umich.edu

Introduction

The realm of game-playing algorithms has witnessed remarkable advancements in recent years, contributing to the development of sophisticated strategies and decision making processes. One intriguing area within this domain is the application of Monte Carlo Tree Search (MCTS) to board games where it has been proven to be a powerful tool for strategic planning and optimal move selection.

Our focus lies in extending the capabilities of MCTS through parallelization with CUDA to the game of Connect 5, a strategic board game derived from well-known Connect 4. Connect 5 introduces an additional layer of complexity and strategic depth, challenging traditional algorithms to adapt and excel in this more intricate environment. The game inherits the essence of Connect 4, where two players take turns placing tokens on a grid with the goal of connecting five of their tokens in a row horizontally, vertically, or diagonally. However, the expanded board size and increased potential for connections makes the decision making possibilities more diverse.

The fundamental objective of this project is to harness the parallel processing capabilities of modern architectures to enhance the efficiency and effectiveness of the MCTS algorithm. By allowing multiple branches of the game tree to be explored simultaneously, we aim to expedite the decision-making process and consequently, elevate the strategic prowess of our decisions.

Throughout this report, we delve into the theoretical foundations of MCTS, discuss the intricacies of adapting it into a parallelized algorithm and present the implementation details. Furthermore, we provide insights into the experimental results and performance metrics.

Monte Carlo Tree Search

At its core, MCTS is a simulation-based search algorithm that combines tree expansion, node selection, simulation, and backpropagation to iteratively build a search tree and converge toward an optimal solution. The algorithm is particularly adept at handling games with enormous state spaces and complex decision trees.

Key Components -

1. Selection: MCTS begins with the selection of nodes in the search tree, emphasizing exploration and exploitation. The selection process involves choosing nodes that strike a balance between exploring new possibilities and exploiting known paths. This is typically achieved using heuristics or statistical methods.

2. Expansion: Once a selection has been made, the algorithm expands the tree by adding new nodes representing possible future states. This step simulates the possible outcomes of different moves, gradually expanding the search space.

3. Simulation: MCTS employs Monte Carlo simulations to estimate the value of each node by playing out random or heuristic sequences of moves from the selected nodes. These simulations provide a stochastic evaluation of the potential outcomes contributing to the decision-making process.

4. Backpropagation: The results of the simulations are then backpropagated through the tree, updating the statistics associated with each node. This information is crucial for guiding future selections and ensuring a more informed exploration of the game tree.

MCTS exhibits several advantages that contribute to its widespread adoption in game-playing strategies, namely:

Adaptability: MCTS is adaptable to a variety of games and domains, making it a versatile choice for strategic decision-making in different contexts.

Informed Exploration: The algorithm dynamically adjusts its exploration strategy based on the evolving knowledge within the tree, enabling it to focus on promising branches.

Sample-Efficiency: MCTS can achieve strong performance with a relatively small number of simulations, making it computationally efficient compared to exhaustive search methods.

Difficulty of the Problem

As described in the section before MCTS is a popular algorithm for decision making in a strategic game and in other domains with sequential and uncertain outcomes. MCTS inherently is parallelizable in theory, but our approach is to develop a fast and efficient parallel MCTS algorithm that can be executed on the GPU. We want our program to use the vast number of resources that are available on a GPU. Parallelizing MCTS on a GPU is a difficult problem because in MCTS we are required to traverse a tree where each node represents a possible game state which can yield in a win, loss, or a draw. The tree traversal itself for MCTS becomes difficult on the GPU because the structure of the tree changes dynamically as more and more game states are explored and simulated by the program. As the traversal happens new game states are created at every level of simulation, for which memory needs to be allocated and deallocated dynamically. We also want to dynamically deallocate memory for nodes that are not as important to us as the exploration and simulation continues because the GPU has a very limited amount of memory and the tree can get very large as the nodes get further expanded upon simulation. For example, for our board of size 8x8 we can have a maximum depth of 64 and each level of the tree would have up to 64 child nodes where each node would hold all the information about a game state as will be described in the approach section. Additionally, performance of a GPU depends on efficient memory access patterns, and irregularity in memory access patterns in a tree structure can lead to a less predictable memory access which can highly affect the performance of our program. Therefore, we have to regularize memory access, allocation, and deallocation as much as possible in our program.

Parallelising MCTS in CUDA also becomes difficult because of irregular workloads. The workload in MCTS is highly irregular as different branches of a tree can have significantly different sizes based on the nodes that are selected to be simulated at every given level of a branch of the tree that is being expanded. This is because the selected sequence of nodes may not lead to a terminal condition (i.e. win, loss, draw) until the very last level of the branch has been evaluated. This makes it challenging to evenly distribute work and synchronize the program for different stages of MCTS algorithm among GPU threads.

Additionally, the data parallelism in MCTS is limited. Many operations in the algorithm depend on the state of the game and structure of the tree which introduce data dependencies for which we have to synchronize and wait for the threads to finish their executions and hence limiting truly independent parallel communication.

Connect 5

Connect 5 is a strategic board game that represents an evolution of the well-known Connect 4. Rooted in the tradition of abstract strategy games, Connect 5 introduces additional complexity and depth, challenging players to engage in more intricate and forward-thinking gameplay.

Setup and Objective:
The game is played on a rectangular grid and consists of two players, represented by Black and White tokens, taking turns placing their tokens on the grid. The objective is to achieve a consecutive connection of five similar tokens either horizontally, vertically or diagonally. If no more tokens can be placed on the board, it leads to a draw.

Connect 5 inherits the fundamental mechanics of Connect 4 but distinguishes itself with a larger game board and an increased number of connections to elevate strategic depth.

Strategic Elements:
Connect 5 demands a balance between offense and defense, requiring players to anticipate and block their opponent's potential winning moves while strategically advancing their own position.

Long term Planning: Successful players must exhibit a keen ability to plan several moves ahead, considering the implications of each placement in the context of the enlarged game board.

Adaptability: The game's dynamic nature calls for adaptability as players must adjust their strategies based on their opponent's moves and the evolving state of the board.

Approach

The following section is an overview and analysis of the implemented classes involved in the MCTS as well as the host and device functions defined within them. We will highlight the relation between the functions meant to be executed parallelly on the Graphic Processing Unit (GPU) as well as the bottlenecks to the development process that we experienced.

Classes and Structs:

1. Board:

The `Board` class represents the game board and provides essential functionalities for game logic and state management. It includes features such as updating the board, validating moves, checking for winners, and retrieving valid moves.

Key Attributes:
- Board representations: The board is represented as a 8x8 2D array of tokens (`EMPTY`, `BLACK`, `WHITE`). This allows for efficient manipulation and retrieval of the game state.
- Move Handling: The class provides methods for making moves, checking their validity, and updating the board accordingly. The `make_move` method is responsible for updating the board state based on the player's move.
- Winner Detection: The class checks the entire board for a winner by examining all possible horizontal, vertical, and diagonal connections.
- Host-Device Compatibility: The class is designed to work seamlessly with both the host and the device(GPU) through the use of `__host__` and `__device__` annotations. This

enables potential performance gains by offloading computations to the GPU. Some of the trivial functions such as making moves or printing the board are designed to work the same on the CPU as well as the GPU. However, some of the more complex functions such as checking for a winner or getting all valid moves are separated into different implementations for the host and the device.

- Dynamic Memory Management: The class handles memory allocation and deallocation for the board, and it provides methods (`move_to_cpu` and `move_to_gpu`) to transfer the board between the CPU and GPU. The board is represented by a single dimensional array of 64 Tokens in the GPU.

2. Node:

The `Node` structure is a fundamental building block within the context of the MCTS algorithm. It encapsulates information about a specific game state derived from a move made by any player, tracks statistical data relevant to the exploration-exploitation trade-off and manages the tree structure by linking to parent and child nodes.

Key Attributes:

- Game statistics: Each node keeps track of game statistics during simulations. These statistics include the number of times the node has been visited, score which is proportional to the number of wins observed from the descendants of the node (a terminal condition adds or subtracts from the score depending on the player that wins), total number of simulations conducted from the node, and the score assigned to the node which is calculated from the wins/losses and visits stemming from the node.
- Tree structure: The node is also responsible for maintaining the structure of the tree utilized in the algorithm. It maintains a pointer to its parent node as well as a pointer to its children nodes as they are created. It contains variables to keep track of the number of children, a board variable to represent its state, and a variable to represent the move made by the player in that node (represented by a Position structure containing the row and column as integers).
- Node expansion: Nodes contain functionality to expand itself by generating children. This can be done on the host or the device through different functions. The node generates all valid moves from a particular board state and then creates, initializes, and adds children Nodes corresponding to each valid move.

3. MonteCarloTree:

The `MonteCarloTree` class represents the Monte Carlo Tree Search algorithm for decision-making in the game. It manages the tree structure and facilitates Node creation, deletion, and traversal.

Key Attributes:

- Tree structure: The class maintains the tree structure by keeping track of the root node with the initial game state. The class provides methods to create nodes, get children as well as delete entire trees.
- Simulation and Backpropagation: The `simulate` method simulates games from a given node, the results are back propagated to update the statistics of ancestor nodes. The method invokes a GPU kernel that parallelizes the simulation process.

Kernels

1.GetValidMoves:

The purpose of this GPU kernel is to identify and collect valid moves, specifically empty positions, on a one-dimensional game board array. The kernel is designed to be invoked on a 2x2 grid containing a 8x8 block, with each thread assigned to a unique position on the game board. It takes as input a one-dimensional array representing the game board on the GPU.

During kernel execution, each thread examines its corresponding position on the game board. If the position is empty (denoted by `Token::EMPTY`), the thread increments a counter for valid moves using the atomic addition operation to prevent race conditions. The atomicAdd() function guarantees that a particular thread adds a value to a global variable without being interrupted by another thread. Without this locking effect, multiple threads could attempt to update the counter at the same time, leading to the wrong value being stored in it. Another purpose of atomicAdd() is to return the value of the counter prior to the update by a thread. This value is used by the thread as the index on the array of valid moves it is meant to update with the position it represents. Without the atomic operation multiple threads could share the same index leading to array values being overwritten.

The primary outputs of the kernel include an array (`valid_moves`) containing all the positions eligible for player moves and a pointer to an integer (`valid_moves_count`) representing the total number of valid moves found during the execution. This approach ensures that each thread contributes its discovered valid move to a shared array while avoiding conflicts through atomic operations.

Prior to the kernel call, memory is allocated for all the global resources shared amongst the threads. When the kernel is called from a host function, this memory allocation is conducted using the cudaMalloc() function and the arrays are populated using cudaMemCpy(). When the kernel is called from a device function, it is not necessary to use the CUDA specific functions to allocate memory as the execution is already on the device. In these cases, memory is allocated dynamically on the heap to be shared by the threads of the kernel, and it would not be necessary to allocate memory to create another version of the game board as the current version in the device can be reused for the purposes of the kernel.

Following the kernel call, memory is similarly deallocated from the device. When the kernel is called from a host function, the deallocation is conducted using the cudaFree() function and necessary data is extracted using cudaMemCpy() from the device, back into the host. On the device, it would not be necessary to use the respective CUDA functions as mentioned previously, and so we have deallocated the memory on the heap using delete[] on the pointer to the memory.

When getting the valid moves from the device, we deallocate the shared counter of the number of valid moves immediately after the kernel has finished its execution, however we only deallocate the memory related to the total number of valid moves after it has been utilized to expand a node and create the appropriate child for it.

In summary, the kernel efficiently identifies and records valid moves on the game board, leveraging parallelism with the given grid and block configuration. It uses atomic operations to update a shared counter for valid moves, facilitating a race condition-free accumulation of valid moves in the output array.

2.CheckWinner:

The purpose of the `check_winner_kernel` is to identify a winning player on a game board represented by a 1D array (`board`) on the GPU. The kernel is invoked with a grid with a block and threads (configuration is same as for GetValidMoves), with each thread corresponding to an individual position on the game board. The kernel takes as input the game board, a pointer to a variable indicating the winner (`winner`), the size of the board (`size`), and the winning length (`win_len`).

During the kernel execution, each thread evaluates the state of the game at its assigned position on the board. If the position is empty, the thread returns, indicating that no winner can be determined at this position.

Next, the kernel examines possible winning combinations, including vertical, horizontal, and diagonal alignments. It creates arrays to store the tokens in these directions and checks for winning patterns by comparing the tokens with the player's token.

In a similar fashion to the `GetValidMoves` kernel, the `CheckWinner` kernel also allocates and deallocates memory required for the execution of the kernel. The only memory that is not immediately deallocated following the kernel execution in the device is the value stored within `winner`. This value is utilized by the `simulatekernel` kernel to determine if the simulation should continue or if it has reached a terminal state.

3.Simulate:
The `simulatekernel` GPU kernel is designed for parallelized Monte Carlo Tree Search (MCTS) simulations on an array of child nodes (`children`). Invoked with a grid of blocks and threads, each thread processes a specific child node within the array. The kernel initializes random number generators, simulates games for each child node, and updates game states based on random moves. The simulation continues until a predefined time limit is reached (9000 clock cycles). Key functionalities include utilizing the `Node` class methods, such as `expand_device`, to generate valid moves and updating game states during simulations.

During each iteration, the kernel checks for a winner or a draw, adjusting scores and statistics accordingly. If a winner is found, scores are updated recursively up the tree, considering the winning player. In case of a draw, scores are adjusted based on the player at each level. The simulation loop terminates once the time limit is reached.

The kernel is only called from the host device and so, it would only require memory to be allocated and deallocated using the designated CUDA functions. The information copied back from the device after the execution of the kernel is the array of children nodes that were used as the starting points of the simulations. These nodes contain the necessary game statistics to determine the best possible move to be made by the player. The best move is currently determined by the highest/lowest (depending on the player making the move) product of node score and number of completed simulations. The scoring algorithm can be improved by introducing an Upper Confidence Bound (UCB) [1]. The UCB is typically calculated as:

$$v_i = C\,x\sqrt{\frac{lnN}{n_i}}$$

Where v_i is the estimated score of the node, n_i is the number of times the node has been visited and N is the total number of times that its parent has been visited. C is a tunable bias parameter which allows us to account for greater exploitation or exploration.

Dynamic Parallelism in CUDA

Dynamic parallelism in CUDA is a powerful feature that allows GPU kernels to launch new kernels dynamically during their execution. This feature was introduced in CUDA 5.0 and is supported on GPUs with compute capability 3.5 and higher. Dynamic parallelism enables more flexible and adaptive GPU programming, providing the ability to create hierarchical and recursive parallel algorithms.

With dynamic parallelism, a GPU kernel can launch new kernels without involving the CPU. This allows for more efficient and complex parallel algorithms where workloads can be dynamically divided and assigned to GPU threads on the fly. This contrasts with the traditional static approach where all GPU work is defined upfront by the CPU.

Dynamic parallelism is particularly beneficial in scenarios where the depth or structure of the computation is not known beforehand. For example, in tree-based algorithms like certain types of searches or traversals, dynamic parallelism allows kernels to spawn child kernels as needed, adapting to the complexity of the data.

In `simulatekernel`, we leverage dynamic parallelism in CUDA to enhance the flexibility and adaptability of our Monte Carlo Tree Search (MCTS) simulations. Specifically, the nodes within the kernel autonomously call their functions to obtain all valid moves, initiating parallel kernels to handle this operation efficiently. This dynamic approach allows the GPU to dynamically allocate resources for generating valid moves without relying on CPU intervention.

The `Node` structure within the kernel, during simulation, further employs dynamic parallelism to check for a winner. The node calls a relevant function to perform this check, which in turn invokes a CUDA kernel responsible for determining the game's winning conditions. This dynamic invocation of kernels enables the node being simulated to autonomously assess its game state without relying on external CPU control.

This dynamic parallelization strategy enhances code modularity within `simulatekernel`. Each node can independently initiate operations such as obtaining valid moves and checking for a winner, promoting a more modular and readable code structure.

Synchronization

In our program, synchronization is crucial, occurring both outside the device on the CPU and within the device on the GPU. Notably, dynamic kernel calls introduce asynchronous behavior, necessitating a robust synchronization method within the GPU kernel. The parallelization method we attempted to use was a hybrid block and leaf parallelization strategy [2]. Each thread is responsible for the simultaneous simulations generated from a particular leaf node from the shared board. If a simulation is completed before the assigned time has elapsed, the thread backpropagates the results of the simulation and then begins a new randomized simulation that utilizes previous statistics. This helps in ensuring appropriate load balancing for all threads. It is also necessary to synchronize our threads so that all threads complete their simulations over the allotted time and the results from each of them are collected back to the children nodes to be copied into the CPU.

Initially, attempts were made using `__syncthreads()`, but it proved insufficient as it only synchronizes threads within the same block and often failed to provide the desired synchronization across blocks. A subsequent strategy involved utilizing subsequent kernel calls with `cudaStreamTailLaunch`. This approach demonstrated a synchronizing effect; however, it introduced challenges related to memory management, as multiple kernel calls produced a greater strain on the available GPU memory..

The final successful approach involved reverting to an older CUDA module version (10.2.89). This allowed us to use `cudaDeviceSynchronize` within the GPU kernel effectively. It's noteworthy that, in versions beyond 11.6, CUDA deprecated and ultimately removed the ability to use `cudaDeviceSynchronize` within a kernel.

Bottlenecks

We utilized the GPUs available to us through the Great Lakes High-Performance Computing clusters by the University of Michigan. The GPUs that can be accessed on the cluster are the NVIDIA TESLA V100 and NVIDIA A100. The bottleneck we encountered was the finite GPU memory available on both the TESLA V100 and A100 models. While these GPUs offer substantial memory capacities compared to consumer-grade GPUs, complex simulations, especially those involving dynamic parallelism and multiple kernel launches, can quickly consume the available memory. For large datasets or deep hierarchies of dynamic parallelism, the GPU memory may become saturated, leading to out-of-memory errors.

The architecture of the TESLA V100 and A100 GPUs is designed to handle massive parallelism, making them well-suited for high-performance computing tasks. However, the challenge arises when attempting to scale simulations to match the capabilities of these GPUs. Complex algorithms and large datasets may not fit entirely into the available memory, necessitating careful optimization or partitioning of the workload.

We were initially working on creating our algorithm to simulate moves on a 16x16 board. However, the GPU available to us consistently ran out of memory in a single kernel call before each of the children of the root node could make substantial simulations to reach a meaningful game state. This led to us adapting our program to simulate moves on an 8x8 game board. Since data is retained over multiple simulations (this includes game statistics and the generated trees) to produce a single move, there is a limit on the total time we can allocate to each thread for it to generate simulations. The limit we have placed now is 9000 clock cycles for each thread, but we feel that increasing the allocated time for each thread could possibly produce a better representation of the potential move selections from a given game state.

Analysis

The performance of an MCTS algorithm is determined by the number of simulations per node it can conduct in a given amount of time. A complete simulation is when a parent node is completely expanded and one of its children hits a terminal state (i.e. win, loss, draw). When a node goes through a large number of simulations it can be said that it has gone through a lot of scenarios, which implies that the algorithm is better informed about all the possible outcomes that can arise from a game state. Having this information makes the program more confident to make a move that could maximize the score for the player that it is making a move for (i.e. higher probability of a win) or minimize the score for the opponent player (i.e. a lower probability of loss, may lead to a draw).

The maximum amount of simulation time that we evaluated was for 9000 clock cycles which translates to 9 seconds (given the clock rate of the GPU). The number of simulations per node increases as the time allocated for simulation and expansion increases. Since MCTS is a dynamic tree expansion and traversal algorithm the number of simulations per node also depends on the maximum achievable depth upon complete expansion of the node. The number of simulations increase as the maximum achievable depth decreases. The number of simulations also increases if the board size decreases as the total number of possible moves decreases and the maximum achievable depth for an empty board decreases as well. These claims will further be solidified by the experiments shown below.

The first experiment that we ran was to see the dependence of the total number of simulations to the time allotted for simulations. We ran this experiment on a new 8x8 game board where only one initial move was made. Having a root node where only one move is made implies that in the first layer of the tree it will have a total of 63 children as that is the total number of moves that can be made. Each child in the first layer has a maximum achievable depth of 62 and the total number of children decreases as we expand more layers of the tree. In the first layer, each child is simulated simultaneously and the results are cumulated into the child array as described in the approach. Given this maximum number of simulations that we got was 464 at 9000 clock cycles.

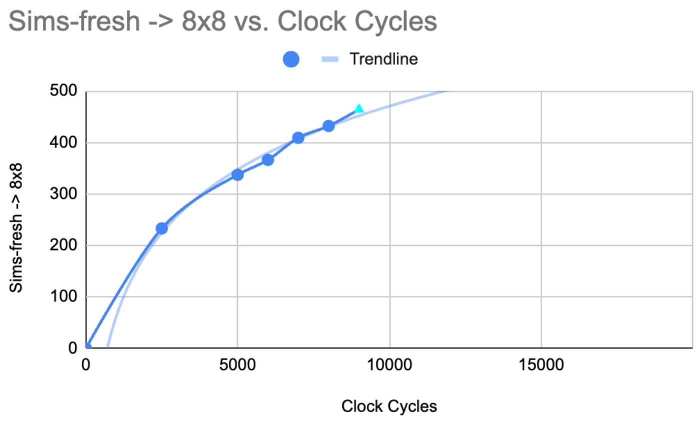


Figure 1. Number of Simulations vs Clock Cycles

As it can be seen in figure 1 that the total number of simulations per node increases logarithmically with time (from the trendline). Therefore, at some point increasing time would increase the total number of simulations per node only marginally.

Similarly we ran an experiment with 4 moves made initially and setting that game state as the root node of the tree. The number of simulations still increases logarithmically with respect to time, but we have more total number of simulations as the maximum possible depth for the tree has decreased by 4. Which implies that as the game continues and the total number of possible moves and the maximum possible depth of the tree decreases the total number of simulations for the root node increases. Less number of possible moves and the decreasing maximum possible depth implies that the data that needs to be evaluated decreases dynamically as the program simulates through all the possibilities. And since there is less data to evaluate for the given number of threads, there are more simulations per node which implies weak scaling for the program. This can be seen in Figure 2 and Figure 3 shown below.

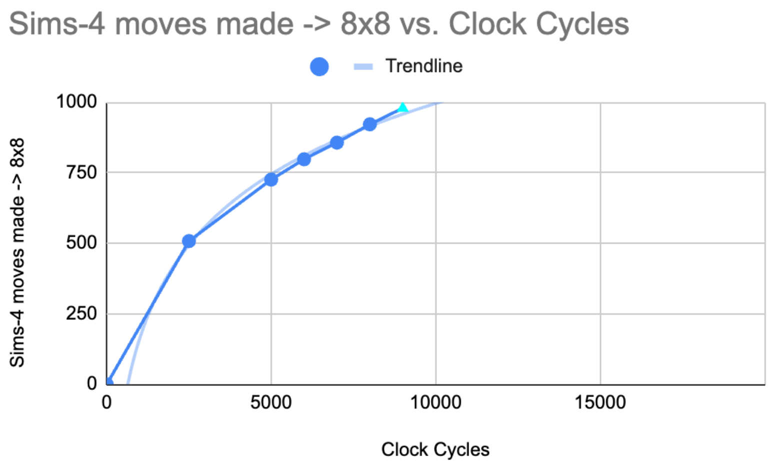


Figure 2. Number of simulations vs Clock Cycles (4 moves made)

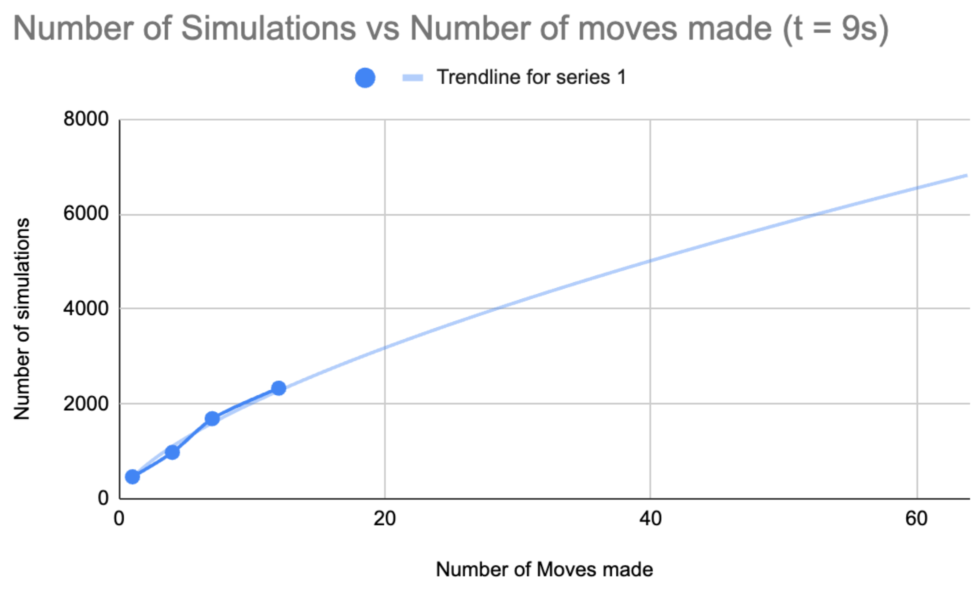


Figure 3. Number of simulations vs Number of moves made

We already know that the program dynamically scales weakly for a given board size and number of threads. To further analyze the scaling of the program we tested our program against different board sizes and the results can be seen in Figure 4. In figure 4, simulations for three different board sizes, i.e. 6x6, 7x7, 8x8, are conducted with different simulation times to analyze the performance of the program. From the graph we can clearly see that a smaller board produces more simulations consistently for the same amount of time and number of threads. This implies weak scaling of the program with respect to the board size as well. Therefore we can say that as the size of data increases, the time for simulations needs to be increased as well to get a similar performance (Performance metric is number of simulations) given the number of threads.

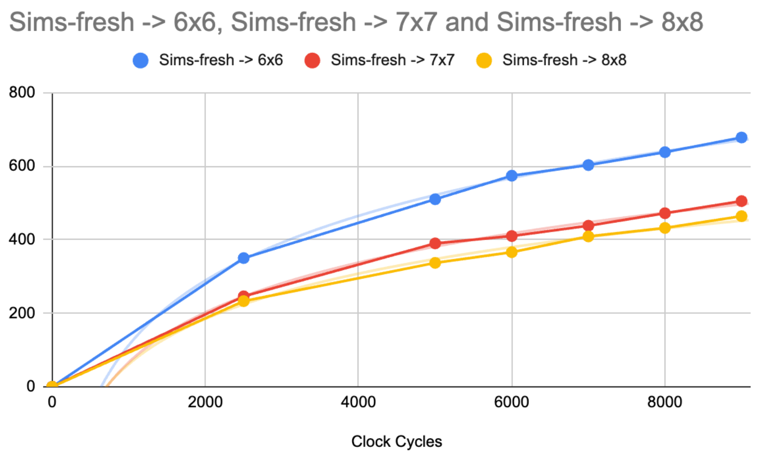


Figure 4. Number of simulations vs Clock Cycles (for different board sizes)

Future Plans

We feel that the project has the scope to expand beyond generating simulations and producing the best move for the game Connect 5. The algorithm can be adapted with some modifications to produce the best move in the case of many other abstract strategy based board games as well. A commonly tested game for an MCTS is the game Go. Therefore, it would be interesting to test the performance of our algorithm to produce the best moves for Go and compare the results with existing algorithms.

At the moment, we provide an empty or intermediate board state to the program and observe one or more moves that the algorithm produces. Another idea we had for the project was to create a User Interface (preferably Graphical) to allow a user to play the game of Connect 5 and have our MCTS be the designated opponent of the player so that the decisions made by the algorithm could be monitored and configured more easily to create a more robust analysis of the simulations. This could be followed by the customization of game settings such as board size and depending on the user's preference.

As mentioned previously, our selection strategy to select the best move is a linear algorithm. We could update this selection strategy as well as the randomized move selection within the simulations to include a UCB to make a better informed decision that provides a payoff between exploration and exploitation within the simulations.

A major improvement to our project would be the incorporation of advanced memory optimization techniques to efficiently utilize the memory available to us on the GPUs and reduce the memory footprint of simulations. This could be done by utilizing data compression and data streaming algorithms within our kernels.

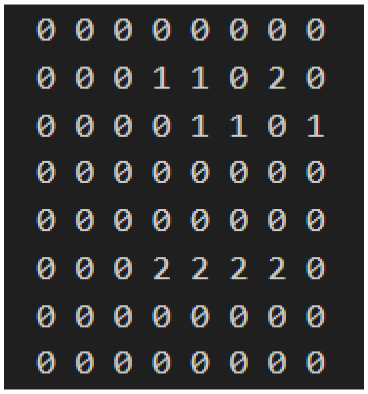
References

[1] Auer, P., Cesa-Bianchi, N. & Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem.*MachineLearning***47**,235–256(2002). https://doi.org/10.1023/A:1013689704352

[2]Chaslot, G.M.J.B., Winands, M.H.M., van den Herik, H.J. (2008). Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds) Computers and Games. CG 2008. Lecture Notes in Computer Science, vol 5131. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-87608-3_6

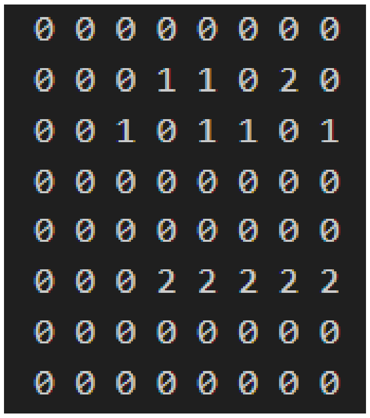
Appendix A:

The following images showcase a move selected and performed by our MCTS algorithm on an intermediate state that is nearly at a terminal state. The number 0 represents an EMPTY token, the number 1 represents a BLACK token, and the number 2 represents a WHITE token. A fresh board is populated by moves from both players to represent the board state depicted in Figure 5. This board state is then supplied to the MCTS algorithm which predicts the best move for the WHITE player. The algorithm places a token at the location (5,7) as seen in Figure 6 (columns and rows are indexed starting from 0) as this is the location corresponding to the highest generated selection score as depicted in Figure 7 which is representing the statistics of the children nodes from the given board state. The simulation time allotted for this test was 9s.



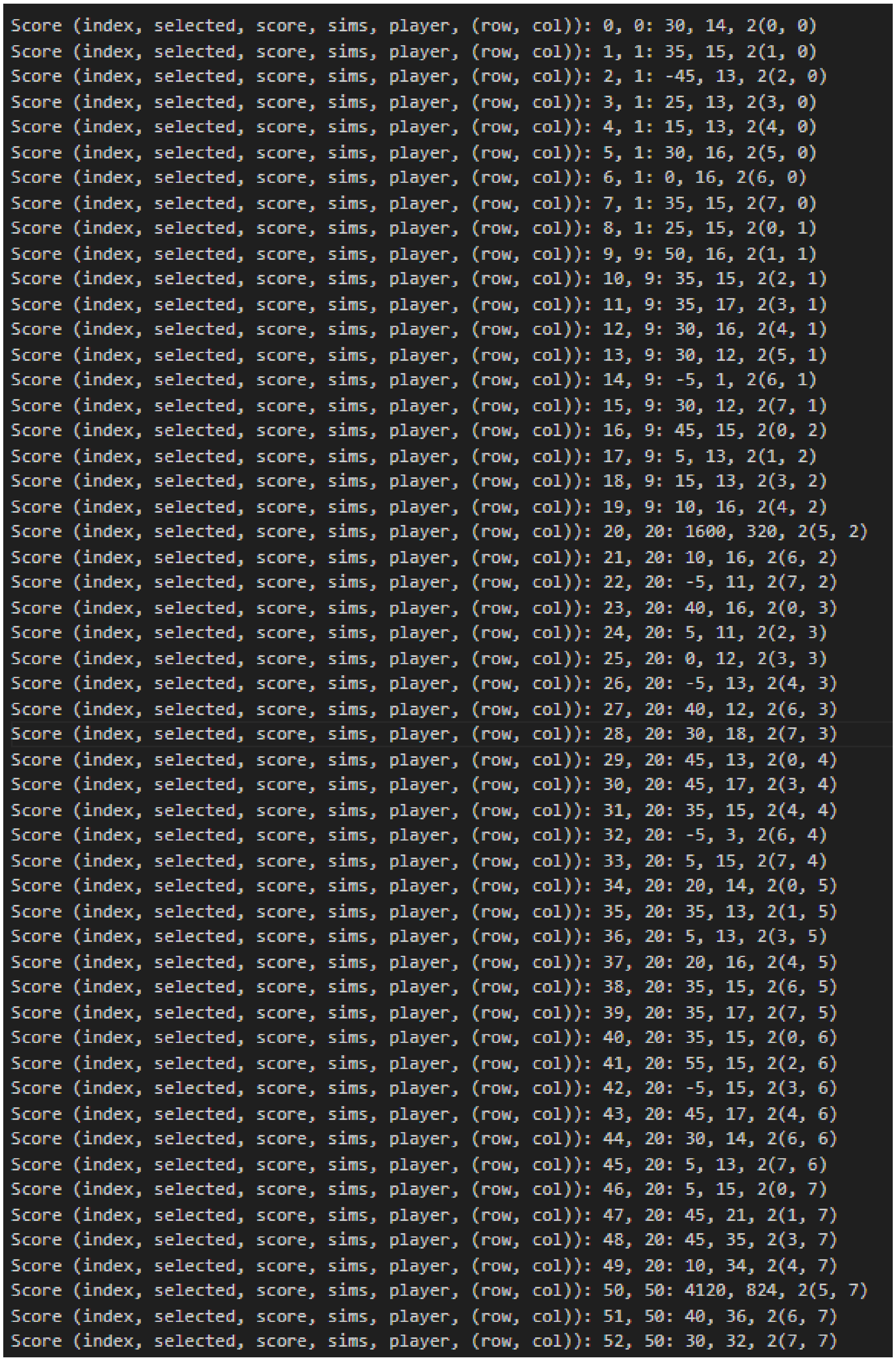
```
0 0 0 0 0 0 0 0
0 0 0 1 1 0 2 0
0 0 0 0 1 1 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 2 2 2 2 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Figure 5. Board state provided to the algorithm



```
0 0 0 0 0 0 0 0
0 0 0 1 1 0 2 0
0 0 1 0 1 1 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 2 2 2 2 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Figure 6: Board state generated by the algorithm placing a token at the position (5,7)



```
Score (index, selected, score, sims, player, (row, col)): 0, 0: 30, 14, 2(0, 0)
Score (index, selected, score, sims, player, (row, col)): 1, 1: 35, 15, 2(1, 0)
Score (index, selected, score, sims, player, (row, col)): 2, 1: -45, 13, 2(2, 0)
Score (index, selected, score, sims, player, (row, col)): 3, 1: 25, 13, 2(3, 0)
Score (index, selected, score, sims, player, (row, col)): 4, 1: 15, 13, 2(4, 0)
Score (index, selected, score, sims, player, (row, col)): 5, 1: 30, 16, 2(5, 0)
Score (index, selected, score, sims, player, (row, col)): 6, 1: 0, 16, 2(6, 0)
Score (index, selected, score, sims, player, (row, col)): 7, 1: 35, 15, 2(7, 0)
Score (index, selected, score, sims, player, (row, col)): 8, 1: 25, 15, 2(0, 1)
Score (index, selected, score, sims, player, (row, col)): 9, 9: 50, 16, 2(1, 1)
Score (index, selected, score, sims, player, (row, col)): 10, 9: 35, 15, 2(2, 1)
Score (index, selected, score, sims, player, (row, col)): 11, 9: 35, 17, 2(3, 1)
Score (index, selected, score, sims, player, (row, col)): 12, 9: 30, 16, 2(4, 1)
Score (index, selected, score, sims, player, (row, col)): 13, 9: 30, 12, 2(5, 1)
Score (index, selected, score, sims, player, (row, col)): 14, 9: -5, 1, 2(6, 1)
Score (index, selected, score, sims, player, (row, col)): 15, 9: 30, 12, 2(7, 1)
Score (index, selected, score, sims, player, (row, col)): 16, 9: 45, 15, 2(0, 2)
Score (index, selected, score, sims, player, (row, col)): 17, 9: 5, 13, 2(1, 2)
Score (index, selected, score, sims, player, (row, col)): 18, 9: 15, 13, 2(3, 2)
Score (index, selected, score, sims, player, (row, col)): 19, 9: 10, 16, 2(4, 2)
Score (index, selected, score, sims, player, (row, col)): 20, 20: 1600, 320, 2(5, 2)
Score (index, selected, score, sims, player, (row, col)): 21, 20: 10, 16, 2(6, 2)
Score (index, selected, score, sims, player, (row, col)): 22, 20: -5, 11, 2(7, 2)
Score (index, selected, score, sims, player, (row, col)): 23, 20: 40, 16, 2(0, 3)
Score (index, selected, score, sims, player, (row, col)): 24, 20: 5, 11, 2(2, 3)
Score (index, selected, score, sims, player, (row, col)): 25, 20: 0, 12, 2(3, 3)
Score (index, selected, score, sims, player, (row, col)): 26, 20: -5, 13, 2(4, 3)
Score (index, selected, score, sims, player, (row, col)): 27, 20: 40, 12, 2(6, 3)
Score (index, selected, score, sims, player, (row, col)): 28, 20: 30, 18, 2(7, 3)
Score (index, selected, score, sims, player, (row, col)): 29, 20: 45, 13, 2(0, 4)
Score (index, selected, score, sims, player, (row, col)): 30, 20: 45, 17, 2(3, 4)
Score (index, selected, score, sims, player, (row, col)): 31, 20: 35, 15, 2(4, 4)
Score (index, selected, score, sims, player, (row, col)): 32, 20: -5, 3, 2(6, 4)
Score (index, selected, score, sims, player, (row, col)): 33, 20: 5, 15, 2(7, 4)
Score (index, selected, score, sims, player, (row, col)): 34, 20: 20, 14, 2(0, 5)
Score (index, selected, score, sims, player, (row, col)): 35, 20: 35, 13, 2(1, 5)
Score (index, selected, score, sims, player, (row, col)): 36, 20: 5, 13, 2(3, 5)
Score (index, selected, score, sims, player, (row, col)): 37, 20: 20, 16, 2(4, 5)
Score (index, selected, score, sims, player, (row, col)): 38, 20: 35, 15, 2(6, 5)
Score (index, selected, score, sims, player, (row, col)): 39, 20: 35, 17, 2(7, 5)
Score (index, selected, score, sims, player, (row, col)): 40, 20: 35, 15, 2(0, 6)
Score (index, selected, score, sims, player, (row, col)): 41, 20: 55, 15, 2(2, 6)
Score (index, selected, score, sims, player, (row, col)): 42, 20: -5, 15, 2(3, 6)
Score (index, selected, score, sims, player, (row, col)): 43, 20: 45, 17, 2(4, 6)
Score (index, selected, score, sims, player, (row, col)): 44, 20: 30, 14, 2(6, 6)
Score (index, selected, score, sims, player, (row, col)): 45, 20: 5, 13, 2(7, 6)
Score (index, selected, score, sims, player, (row, col)): 46, 20: 5, 15, 2(0, 7)
Score (index, selected, score, sims, player, (row, col)): 47, 20: 45, 21, 2(1, 7)
Score (index, selected, score, sims, player, (row, col)): 48, 20: 45, 35, 2(3, 7)
Score (index, selected, score, sims, player, (row, col)): 49, 20: 10, 34, 2(4, 7)
Score (index, selected, score, sims, player, (row, col)): 50, 50: 4120, 824, 2(5, 7)
Score (index, selected, score, sims, player, (row, col)): 51, 50: 40, 36, 2(6, 7)
Score (index, selected, score, sims, player, (row, col)): 52, 50: 30, 32, 2(7, 7)
```

Figure7: Generated selection scores for the children of the given board state