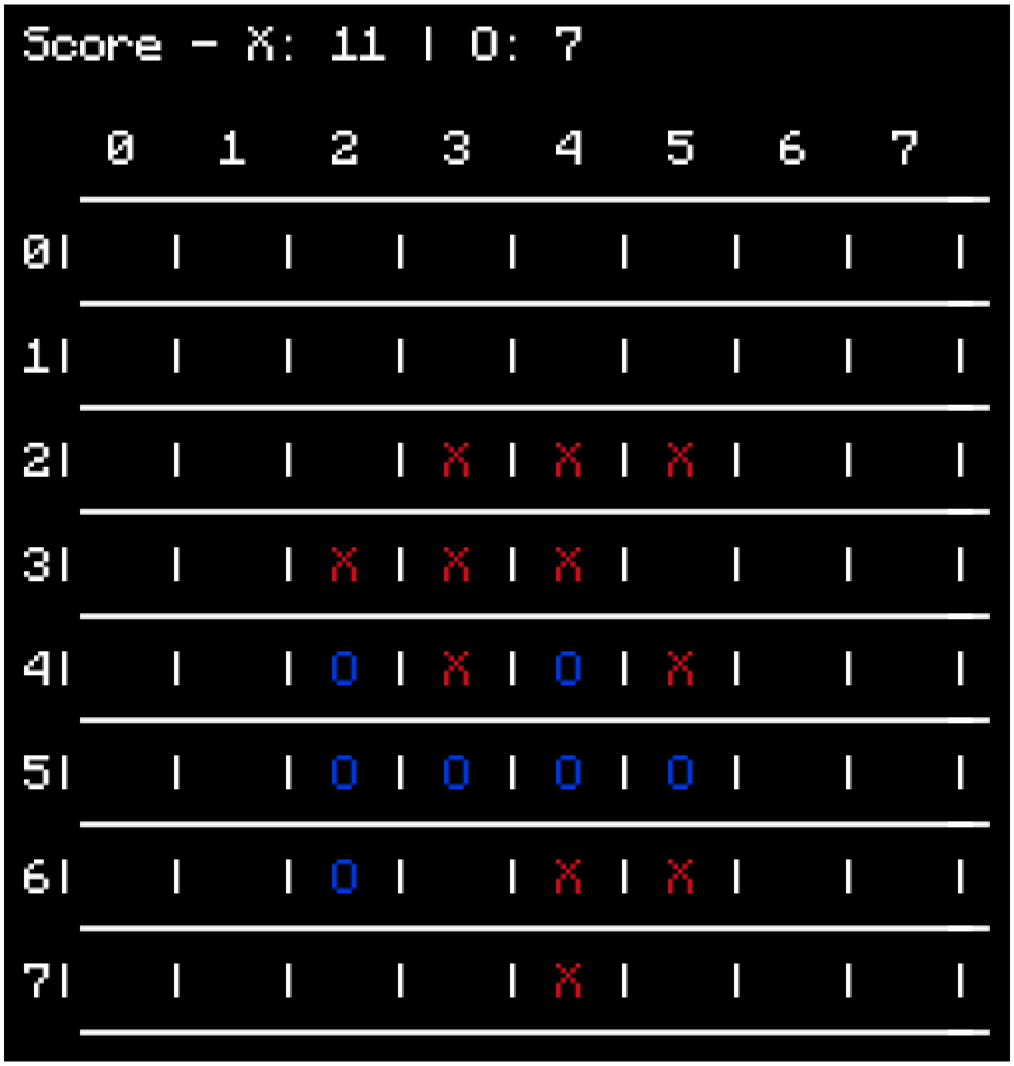
Optimized CUDA-Accelerated Alpha-Beta Pruning for Parallel Minimax Search in Othello

Joshua Brodsky (joshbrod@umich.edu) & Alan Zhang (alanzhng@umich.edu)



```
Score - X: 11 | O: 7

    0   1   2   3   4   5   6   7
  --------------------------------
0|   |   |   |   |   |   |   |   |
  --------------------------------
1|   |   |   |   |   |   |   |   |
  --------------------------------
2|   |   |   | X | X | X |   |   |
  --------------------------------
3|   |   | X | X | X |   |   |   |
  --------------------------------
4|   |   | O | X | O | X |   |   |
  --------------------------------
5|   |   | O | O | O | O |   |   |
  --------------------------------
6|   |   | O |   | X | X |   |   |
  --------------------------------
7|   |   |   |   | X |   |   |   |
  --------------------------------
```

Course: CSE 587 | Semester: Fall 2025

Introduction:

What is Othello

Othello (also known as Reversi) is a two-player strategy board game played on an 8 by 8 grid where each player has discs that are black on one side and white on the other. Players take turns placing a disc with their color facing up, capturing the opponent's discs by trapping one or more of them in a straight line (horizontally, vertically, or diagonally) between the disc just placed and another disc of their own color. All trapped discs are flipped to the capturing player's color. The game continues until neither player can make a legal move, usually when the board is full, and the winner is the player with the most discs of their color showing.

Overview of Project:

The project that will be discussed is a parallelized implementation of an Othello bot AI that uses the Negamax (or Minimax) algorithm with alpha-beta pruning to find optimal moves in our Othello engine. The bot's AI will explore a game tree of possible positions, however the number of positions grows exponentially with depth, making deeper searches computationally expensive.

The Othello bot will specifically be parallelizing the next-best-move-predictor AI that evaluates board positions using a multi-factor heuristic (positional weights, mobility, corner control, frontier pressure) and searches the game tree to select the best move within a time limit. This task is fundamentally non-trivial because the game states of Othello do not form a simple tree, but rather a transposition graph. In such a structure, multiple distinct sequences of moves, possibly originating from entirely different branches at the same search depth, can converge to the same resulting board state. These transpositions mean that the search space is highly interconnected rather than strictly hierarchical, and as a result, naïvely treating each branch as an independent subtree leads to redundant evaluations, incorrect pruning behavior, and inefficient parallelization. Managing these shared states requires careful coordination, hashing, and caching strategies, making the parallelization effort significantly more sophisticated than working with a traditional tree.

Background information:

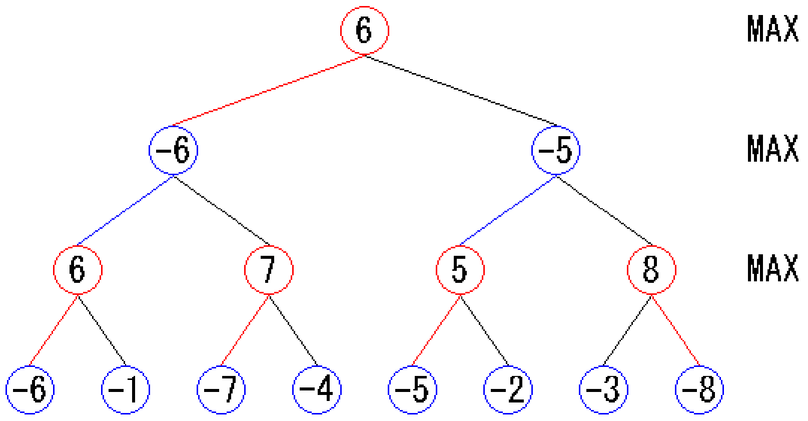
Our implementation relies on two critical concepts in game theory. This section provides necessary background on Negamax and Alpha-Beta pruning to contextualize the architectural choices made in our Othello A

What is Negamax

The algorithm we use for each Othello AI is called negamax. This algorithm is a streamlined version of minimax used in two-player zero-sum games. Instead of treating maximizing and minimizing players separately, negamax relies on the identity:

score(position) = -score(opponent's position).

This symmetry allows the algorithm to use a single recursive function: it assumes the current player is always the maximizer, and the value returned by a child node is negated to reflect the opponent's perspective. This simplification reduces code complexity while using minimax search.



Negamax Example [1]

Example of Execution

The accompanying figure illustrates a Negamax search on a game tree of depth 3. Initially, only the leaf nodes at the bottom of the tree possess heuristic values. The algorithm propagates the best score for the current player to the node above.

The unique characteristic of this algorithm is visible on the right-hand side: every level is labeled MAX. This visualizes the recursive assumption that the node currently being evaluated always belongs to the maximizing player.

To calculate the node values, the algorithm propagates scores upward from the leaf nodes, negating the value at each step to account for the change in perspective.

For instance, consider the bottom-left leaf nodes with values -6 and -1.

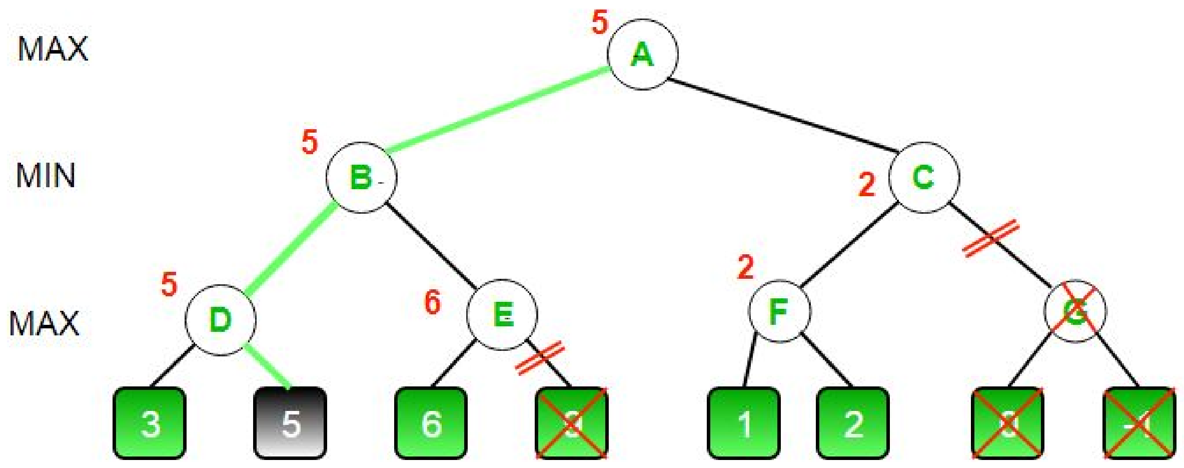
1. The parent node red negates these values to 6 and 1, then selects the maximum (6).

2. Moving up one level, that value of 6 is passed to its parent (blue). It is negated to -6, compared against its sibling (-7), and the maximum is chosen (-6).
3. Finally, at the root, the values -6 and -5 are negated to 6 and 5. The root node selects the maximum, resulting in a final score of 6.

What is Alpha-Beta Pruning

For this project all of the best-move-finder AI implementations utilize alpha-beta pruning. This optimization technique massively narrows the search space allowing for deeper traversal.

Alpha-beta pruning enhances negamax/minimax by eliminating branches of the game tree that cannot influence the final decision. During search, alpha tracks the best guaranteed score for the maximizing player, while beta tracks the best guaranteed score for the minimizing player. If the algorithm discovers that a move leads to a score worse than a previously explored alternative (i.e., alpha ≥ beta), it can immediately prune the remaining siblings because the opponent will never allow that line of play. This pruning reduces the number of evaluated nodes dramatically, often by orders of magnitude, while still producing the exact same result as a full minimax search.



Alpha-Beta Example [2]

Example of Execution:

The accompanying figure demonstrates the efficiency of Alpha-Beta pruning on a game tree (for a minimax example not negamax). Unlike the previous full search, this tree contains unvisited nodes, visually represented by the crossed-out section on the right.

1. The algorithm first explores the left subtree completely. The Maximizer (at the root) assesses the options and secures a guaranteed score of 3 (the minimum of 3 and 5

allowed by the opponent). This establishes the "alpha" value, or the baseline score the Maximizer can ensure.
2. The algorithm then evaluates the right subtree. The Minimizer (at the second level) examines its first child and finds a value of 2.
3. The algorithm immediately identifies a "cutoff." Since this value (2) is already lower than the guaranteed score from the left (3), the Maximizer has no reason to choose this path.
4. Because the opponent can force the score to be at most 2 (or lower) on this branch, the remaining sibling, such as the node valued 9, are pruned (ignored). This saves computational resources without affecting the final decision.

Note: In the above example, α always represents the best score the Maximizer can guarantee, and β always represents the best score the Minimizer can enforce. In the Negamax version, the only difference is that instead of explicitly swapping roles for Max and Min at each depth, we negate α and β at each recursive step. This achieves the same pruning behavior while using a single, symmetric function for both players..

Methodology:

When working on this project we created one game engine with several AIs with different implementations. Each implementation was given the same task of estimating the next best move within a time constraint. The three implementations we created can be classified as serial, naïve parallel, and an optimized pv-split parallel. Each implementation contains its own method of doing negamax search using alpha-beta pruning while still using the high level data structures and functionality provided by the engine.

Serial Implementation

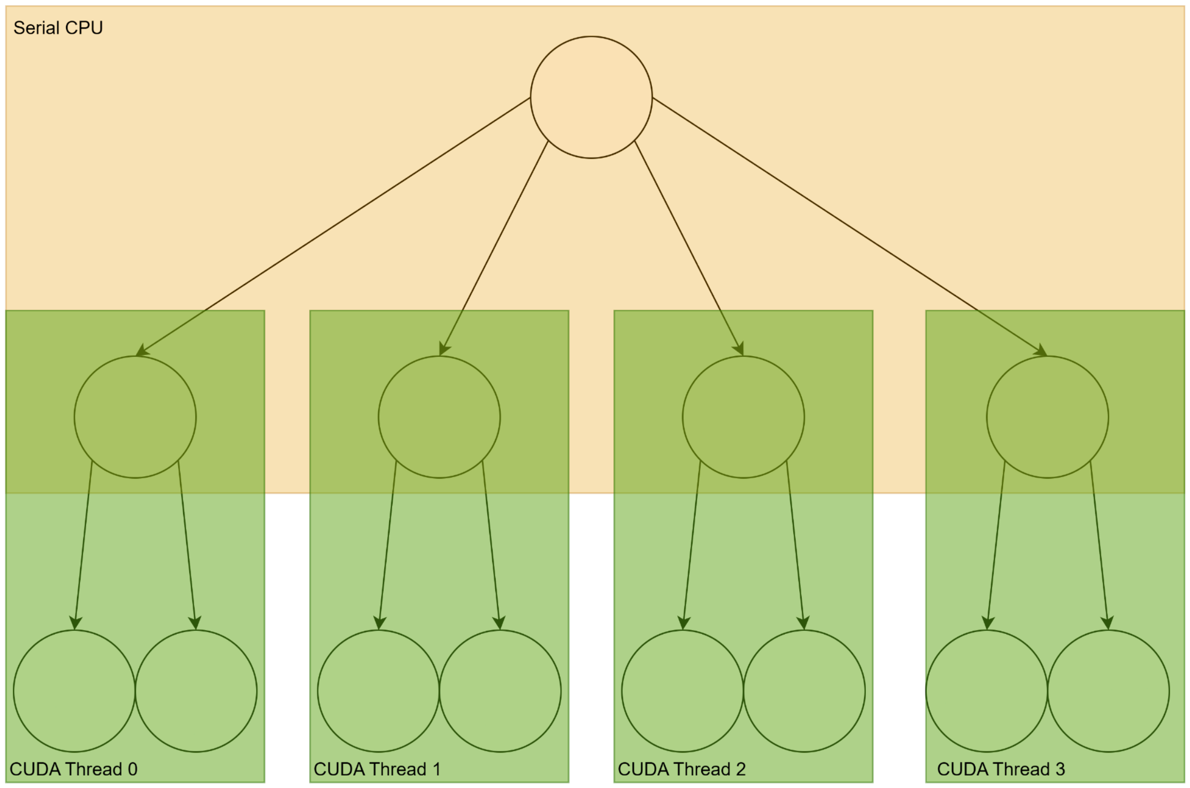
For our first attempt, we created a baseline implementation of the algorithm that runs purely in serial. This implementation uses iterative deepening to estimate the next-best move, starting with an initial search depth of three and, upon completion, repeating the search with the depth limit increased by one. This process repeats until maximum allocated time per move is taken each turn.

We cannot directly reuse the results of a previous search because the heuristic function is imperfect. While it provides an estimate of the game state's value, its accuracy improves at deeper search depths; consequently, a heuristic score at depth three is less informative than a score at depth ten. Although we cannot reuse the exact scores, we can prioritize the best move found in the previous iteration. By evaluating this branch first, we establish a strong alpha value early, which significantly improves pruning efficiency at the new depth.

This iterative deepening search continues until a chosen time limit (e.g., five seconds) has elapsed. Once the time limit is reached, we terminate the search and select the best move found at the last fully completed depth. This approach ensures that a valid move is always available, even if the deeper searches are interrupted.

Naïve Parallel Implementation

This implementation represents our first attempt to translate the algorithm to a CUDA environment. While similar to the serial version, the key difference lies in the scope of the search tree. In this approach, the CPU first enumerates the immediate legal moves (depth-1 children) of the current board state. It then assigns each move to a separate GPU thread.



These threads operate in an embarrassingly parallel manner, each building a separate game tree and calculating the Negamax function using independent alpha and beta values. Because the threads do not share data in this implementation, they cannot prune branches based on findings from sibling threads. We deliberately maintained this simplistic design to establish a clear performance baseline against which our future optimizations could be measured.

Parallel Optimized Implementation

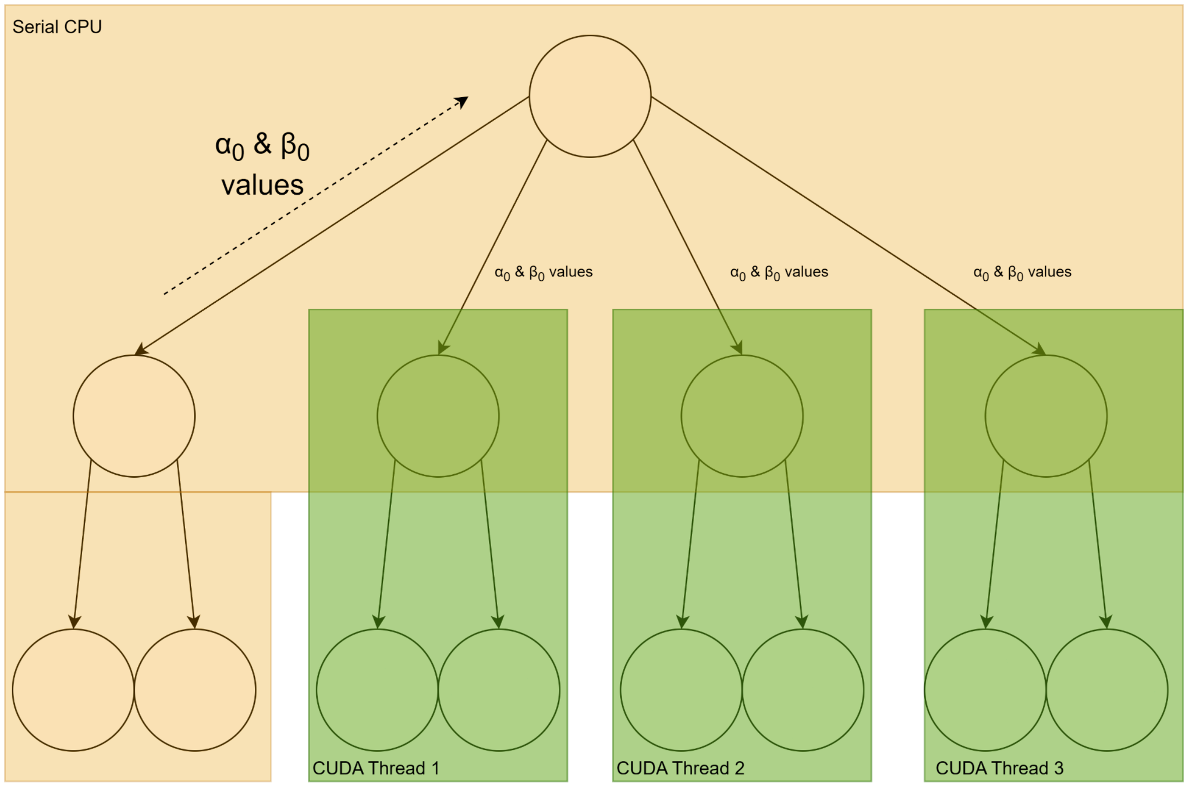
For our optimized parallel implementation we added two main optimizations to increase performance: using an initial CPU workload with Principal Variation Split (PV Split) and distributed shared memory using Transposition Tables across the GPU

Principal Variation Split Parallel Implementation:

For our first optimization in this parallel implementation, we incorporated a Principal Variation Split (PV-Split). This algorithm is designed to optimize the initialization of our alpha-beta values. In a standard parallel search, independent threads often waste computational resources exploring branches that would have been pruned had they known the results of a sibling node. The PV-Split approach mitigates this slightly by initially running the most promising branch, the Principal Variation (PV), in serial on the CPU. This establishes tight alpha-beta bounds which are then passed to the parallelized CUDA algorithm, effectively allowing the GPU threads to prune more aggressively and search more efficiently.

The core philosophy of this architecture is to balance the high-quality sequential search capabilities of the CPU with the massive parallel throughput of the GPU. As illustrated in our design, the CPU handles the "Principal Variation", the move currently assessed as the best, while the GPU evaluates all remaining root moves in parallel.

The diagram below illustrates this workflow, showing the initial serial execution passing optimized α_0 and β_0 values to the parallel CUDA threads.


Serial CPU
α_0 & β_0 values
α_0 & β_0 values
α_0 & β_0 values
α_0 & β_0 values
CUDA Thread 1
CUDA Thread 2
CUDA Thread 3

The coordination process follows a strict iterative deepening protocol:

1. Move Generation & Sorting: The CPU generates all valid moves for the current state and sorts them based on a heuristic estimation. This sorting increases the probability that the first move (the PV) is truly the strongest.

2. Serial Execution (CPU): The CPU performs a full alpha-beta search on the first move. This not only yields a score for that move but, generates a refined alpha value.
3. Parallel Execution (GPU): The remaining moves, along with the updated alpha value, are batched into a GPUBatchProcessor. This ensures that the GPU threads start their work with a "window" of scores that allows them to immediately prune branches that fall below the standard set by the CPU's PV move.
4. Integration & Iteration: Once the GPU returns the batch results, the move list is re-sorted. If a GPU thread finds a move better than the CPU's result, that move becomes the new PV for the next depth of iterative deepening.

While this implementation is still embarrassingly parallel, the PV-Split architecture offers distinct advantages over a purely serial or a naive parallel approach. By utilizing the CPU to establish the search window, we avoid the "search overhead" common in parallel processing where threads do unnecessary work. The GPU handles the width of the search tree, evaluating non-PV moves simultaneously, while the CPU handles the depth of the primary line. This synergy results in a system that maintains high search quality through iterative deepening while significantly accelerating the evaluation of the full game tree.

Transposition Table Optimization

To break from the previously embarrassingly parallel iterations and optimizations, we introduce a shared memory between CUDA threads using a transposition table. This enhances the parallel search by addressing the critical inefficiency of redundant evaluations. In the standard approach, threads operate in isolation, often re-evaluating the same board states that have been reached via different move orders (transpositions). To mitigate this, we integrated a GPU-side Transposition Table (TT) and a host-side duplicate filter.

The centerpiece of this optimization is a global hash table residing in GPU global memory. We allocated approximately 384MB to store 16 million TTEntry structures. Each entry caches the evaluation results of a specific board position, keyed by a 64-bit hash derived from the player bitboards.

- Hashing & Collision: We utilized a hashing strategy to generate a unique input for the hash function involving the boolean value of the current player's turn with each player's bitboards. To handle collisions, inevitable in a table of this size, we implemented a linear probing mechanism. When accessing the table, a thread checks the target index and its immediate three neighbors (4 slots total) before replacement of a slot. If all slots are full the slot that is replaced will be one of minimal depth, specifically less than or equal depth of current board state.This reduces the likelihood of overwriting valuable data without the performance cost of a full scan.
- Entry Data: Each entry stores the computed score, the search depth at which it was found, and a "flag" indicating the nature of the score:
    - Exact (1): The score is a definitive value for that sub-tree.
    - Lower Bound (2): The score caused a beta cutoff (value >= beta).
    - Upper Bound (3): The score failed to exceed alpha (value <= alpha).

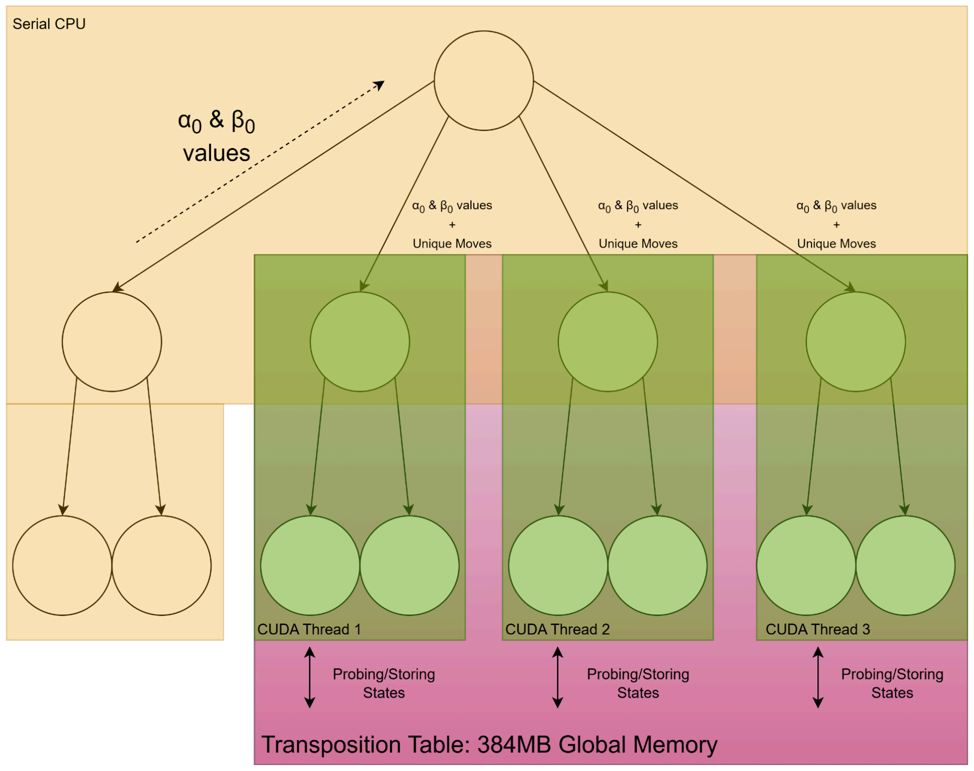
Modified Alpha-Beta Kernel

With the introduction of the transposition table, the alphabeta kernel was significantly modified to interact with this table at two critical points:

1. Probing (Pre-Search): Before expanding children, the kernel queries the TT. If a valid entry is found with sufficient depth (entry.depth >= current_depth), the function attempts to resolve the node immediately. If the stored flag allows, either an exact match or a bound that triggers a cutoff against the current alpha/beta, the search returns instantly, effectively "short-circuiting" the recursion.
2. Storing (Post-Search): Upon completing a search at a node, it is classified based on how it compares to the original alpha and beta bounds. It is then written back to the table with the appropriate flag, preserving the work for other threads or future iterations.

Host-Side Optimization: Batch Processing

While the GPU handles internal tree transpositions, we optimized the "roots" of the parallel jobs on the CPU host. In the standard parallel version, duplicate states generated by different move orders were treated as distinct jobs. The optimized version introduces a pre-processing step using a std::unordered_set to filter out duplicate game states before the batch is sent to the GPU. This ensures that the device never wastes threads solving the exact same problem twice in a single kernel launch.


Serial CPU
α_0 & β_0 values
α_0 & β_0 values + Unique Moves
α_0 & β_0 values + Unique Moves
α_0 & β_0 values + Unique Moves
CUDA Thread 1
CUDA Thread 2
CUDA Thread 3
Probing/Storing States
Probing/Storing States
Probing/Storing States
Transposition Table: 384MB Global Memory

The diagram above represents the wholistic hybrid CPU-GPU design, extending the standard first optimization, PV-Split strategy, by introducing two key optimizations: (1)

Host-side duplicate filtering to ensure only unique game states are batched to the GPU, and (2) A shared global Transposition Table (384MB) allowing CUDA threads to communicate and prune redundant branches across the search space.

Theoretical Impact of Optimization

The optimized implementation represents a deliberate trade-off between memory consumption and computational efficiency.

- Memory: The optimized version requires significantly more device memory (allocated for the TT) compared to other versions, which relies solely on local register/stack memory.
- Speed: By transforming the search from a strict tree traversal to a graph traversal, the TT drastically reduces the effective branching factor. The addition of duplicate removal ensures that all the GPU's throughput is directed toward unique, necessary evaluations.

Results:

The way real performance was measured to compare these implementations was through average depth each method got to through iterative deepening. While in a competitive Othello bot performance would likely be measured with metrics such as ELO or win rate, since this project was focused on increasing parallelism to see more board states we felt this was a better metric to utilize for this project.

We evaluated three Othello search implementations: serial (CPU-only), naïve GPU parallel (GPU Embarssingly-parallel), and an optimized parallel (Hybrid CPU-GPU with PV-split and Transposition Table) approach, measured by the maximum fully completed search depth achieved within fixed time budgets. Experiments were conducted under three time constraints (5s, 15s, and 30s) and across three game phases: early game (moves 1-16), midgame (moves 17-40), and late game (moves 41-60). This phase-based evaluation was used to capture differences in branching factor and search complexity throughout the course of a typical game.

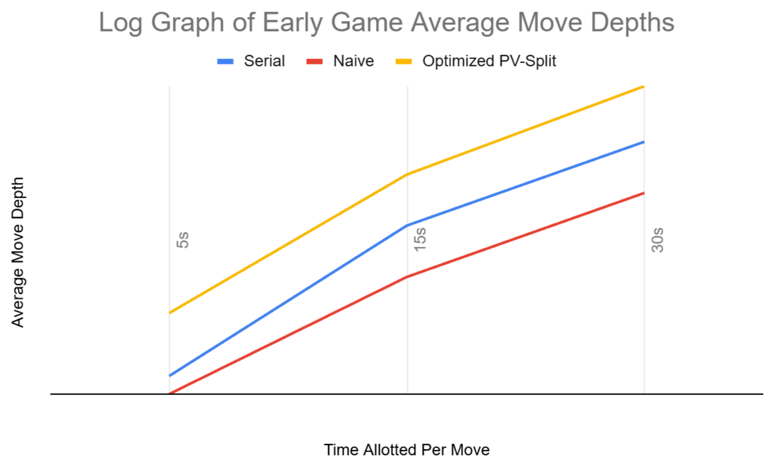
In the late-game benchmarks, all four implementations were able to completely solve the game tree to terminal states at every tested time limit. Because the achieved depths were identical in this phase and reached the theoretical maximum, late-game results are not further differentiated. This indicates that, given the relatively low branching factor and limited remaining move count late in the game, even the serial implementation is sufficient to fully explore the remaining search space within modest time budgets.

Results were measured as an average across 3 games. The benchmark harness was run on an Intel i7-10750H CPU and an NVIDIA 1660 TI GPU. We also used NVCC for all parallel implementations and O3 compiler flag with GCC for the serial implementation and execution framework. The reason we didn’t utilize Great Lakes was due to the length of time each game would be far too great to fit within the five minute time limit threshold instructed by the course.

The benchmarks and log visualization of performance are as follows:

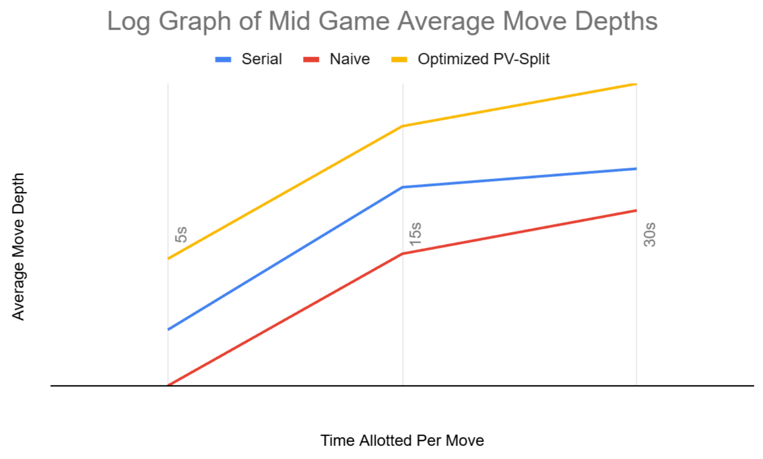
Early Game:

	5s	15s	30s
Serial	12.26	13.25	13.85
Naive	12.19	12.93	13.52
Optimized PV-Split	12.67	13.59	14.2



Mid Game:

	5s	15s	30s
Serial	12.42	13.26	13.38
Naive	12.10	12.86	13.12
Optimized PV-Split	12.83	13.59	13.91



Performance Analysis:

The data demonstrates distinct performance tiers among the implementations, driven by how effectively each method manages pruning and overhead.

1. Optimized PV-Split (Best Performance): This implementation consistently achieved the greatest search depths across all time limits. This success is attributed to the initial construction of alpha-beta values on a smaller subgraph, which allows parallel threads to prune more rapidly. Furthermore, the integration of the Transposition Table (TT) reduced redundant calculations by reusing precalculated heuristic values. The performance gap widened at the 15s and 30s marks, where the optimized design could better amortize fixed GPU costs.

2. Serial Implementation: The serial CPU implementation was the second fastest, outperforming the Naïve parallel approach. Although serial execution lacks concurrency, it benefits from efficient information sharing; it carries alpha-beta values between subtrees, allowing it to prune branches that the Naïve implementation must explore.

3. Naïve Parallel Implementation: The Naïve approach resulted in the shallowest search depths. Because the threads operate in an "embarrassingly parallel" manner without communication, they cannot share alpha-beta values to effectuate pruning as efficiently as the serial version. Additionally, this implementation incurs the overhead of GPU memory instantiation and kernel launches without the algorithmic optimizations required to offset those costs.

Ultimately, while naïve parallelism offers theoretical throughput, these results confirm that algorithm-aware strategies like PV-splitting and Transposition Tables are essential for outperforming optimized serial search in complex game trees.

Phase Comparison:

When comparing the early and mid-game phases of the data above, it further validates the conclusions made above, demonstrating consistent algorithmic behavior despite the evolving board state. Both phases present high branching factors that highlight the efficiency gaps between the implementations, maintaining a stable hierarchy where the Optimized PV-Split strategy consistently outperforms both the Serial and Naïve approaches. This performance advantage becomes particularly decisive during the mid-game, which represents the peak complexity of the Othello game tree. It is in this phase that the Transition Table Entries in shared memory becomes indispensable: as the search space transforms from a tree into a highly interconnected transposition graph, the ability to cache and retrieve Transition Table data (containing other GPU thread’s calculated scores/depths of the same node) allows the algorithm to "short-circuit" redundant search paths. By effectively pruning these repeated states, the Optimized PV-Split implementation successfully mitigates the explosion in branching factor, maintaining deep search capabilities (13.91 ply at 30s) even as the Naïve and Serial approaches struggle with the increased combinatorial load.

Conclusion:

This work demonstrates that effective parallelization of adversarial search is not achieved by brute-force concurrency alone, but by aligning the parallel architecture with the structure of the underlying algorithm. While GPU acceleration provides substantial raw throughput, our results show that naïve parallel minimax fails to translate this throughput into deeper search because independent threads cannot exploit the global pruning information that gives alpha-beta its power. In contrast, the optimized hybrid design, combining CPU-based PV splitting with GPU parallelism and a shared transposition table, consistently achieved the deepest searches across all realistic phases of the game, particularly in the high branching factor midgame where search complexity is greatest.

Overall, this project confirms that high-performance game tree search on GPUs requires algorithm-aware parallelism. The strongest gains emerged not from maximizing parallel thread count, but from preserving and accelerating the pruning dynamics that make alpha-beta efficient. These findings suggest that future work in parallel game AI should continue to treat pruning knowledge as a first-class resource, rather than viewing parallelism as a drop-in replacement for sequential search.

Citations

[1]
D. Sanno, “NegaMax Example,” *Es-cube.net*, 2025.
http://www.es-cube.net/es-cube/reversi/sample/img/minmax.gif (accessed Dec. 08, 2025).
[2]
*“*Alpha-Beta Example,*”Geeksforgeeks.org*, 2018.
https://media.geeksforgeeks.org/wp-content/uploads/MIN_MAX1.jpg (accessed Dec. 08, 2025).