Bothello: A Multi-Strategy, CUDA-Accelerated Othello Engine

N. Antonelli-Dziri
GPU - programming
Politecnico di Torino
Turin, Italy
s350296@studenti.polito.it

A. Morille
GPU - programming
Politecnico di Torino
Turin, Italy
s350501@studenti.polito.it

F. Papini
GPU - programming
Politecnico di Torino
Turin, Italy
s336426@studenti.polito.it

***Abstract*—Monte Carlo Tree Search (MCTS) provides strong artificial intelligence for board games such as Othello but suffers from high computational costs during sequential execution. This paper presents a comparative analysis of three GPU parallelization strategies implemented in CUDA to overcome these bottlenecks. We evaluate Leaf Parallelization, which offloads playouts to maximize throughput; Tree Parallelization, which utilizes a shared tree structure with atomic operations to manage contention; and Block Parallelization, which leverages shared memory for independent tree searches. The proposed architectures are benchmarked against both a single-threaded Central Processing Unit (CPU) baseline and a multi-threaded OpenMP CPU implementation. Performance is evaluated using computational metrics, including playout throughput (playouts per second) and memory bandwidth utilization, alongside game playing strength measured through win rates in a tournament framework. The study analyzes the trade-offs between thread divergence, memory contention, synchronization overhead, and summarizes practical considerations for optimizing parallel game tree search on SIMD architectures.**

***Index Terms*—CUDA, OpenMP, Monte Carlo Tree Search, Othello, Parallelization, Artificial Intelligence**

I. Introduction

Othello, also known as Reversi, is a classic two-player strategy board game played on a 8×8 grid where players place disks to capture opponent's pieces by flanking them. Despite its simple rules, the game exhibits a substantial strategic depth. Recent work by Takizawa [1] has computationally proven that perfect play results in a draw, establishing Othello among solved games while simultaneously highlighting its complexity: the game has roughly 10^{58} possible game records and 10^{28} distinct positions. This complexity makes exhaustive search infeasible during gameplay, positioning Othello as an ideal testbed for artificial intelligence (AI) research. Monte Carlo Tree Search (MCTS) has emerged as a leading algorithm for decision-making in games with large state spaces [2]. Unlike traditional minimax approaches with alpha-beta pruning, MCTS does not require domain-specific evaluation functions. Instead, it estimates the quality of a move through random playouts (rollouts) and progressively builds a search tree biased towards promising branches using the Upper Confidence Bounds for Trees (UCT) selection policy [3].

This sample-based approach enabled landmark achievements including AlphaGo's mastery of Go [4] and, more recently, OLIVAW [5] an Othello AI that demonstrated very strong and competitive play using the AlphaGo Zero paradigm on modest hardware. However, the inherent sequential nature of standard MCTS presents a significant computational bottleneck. Each iteration requires four tightly coupled phases selection, expansion, playout, and backpropagation that typically execute serially. For real-time gameplay with strict time constraints, this limits achievable playout throughput, directly impacting playing strength. Empirical studies demonstrate that MCTS performance scales with the number of playouts; thus, maximizing playout throughput is crucial for competitive AI agents [6]. Graphics Processing Units (GPUs), with their massively parallel architecture, offer a compelling solution. Mirsoleimani et al. [7] proposed lock-free algorithms for parallel MCTS on multi-core CPUs and other works have shown that GPU implementations can produce large speedups but introduce challenges such as thread divergence during tree traversal, memory contention near shared nodes, and irregular memory access from pointer-based trees. This paper presents a comparative study of three GPU parallelization strategies for MCTS applied to Othello, implemented in CUDA:

1) **Leaf Parallelization:** The CPU manages tree operations while the GPU accelerates the simulation phase. Multiple random playouts from a single leaf node execute in parallel, with results aggregated for backpropagation.
2) **Tree Parallelization:** A shared tree structure resides in GPU global memory. All threads concurrently traverse and expand the tree using atomic operations and virtual loss to prevent race conditions.
3) **Block Parallelization:** Each CUDA block maintains an independent search tree (allocated in a per-block region of global memory) to avoid inter-block synchronization. Shared memory is used for intra-block coordination.

These GPU implementations are benchmarked against both a single-threaded CPU baseline and a multi-threaded OpenMP implementation. We evaluate performance through playouts per second and game playing strength via tournament win rates, providing practical guidelines for parallel game tree search on SIMD architectures.

II. Background

A. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first search algorithm that uses random sampling to evaluate game states [2]. The algorithm builds an asymmetric search tree incrementally, focusing computational resources on promising branches. Each MCTS iteration consists of four phases:

1) **Selection:** Starting from the root, recursively select child nodes according to a tree policy until reaching a node with unexpanded moves or a terminal state.
2) **Expansion:** If the selected node is non-terminal and has unexplored moves, create one child node for a randomly chosen unexpanded move.
3) **Playout (Rollout):** From the expanded node, play a random game to completion using a default policy (random move selection).
4) **Backpropagation:** Propagate the simulation result back through the tree, updating visit counts and win statistics along the selection path.

B. Upper Confidence Bounds for Trees (UCT)

The selection phase uses the UCT formula [3] to balance exploration and exploitation:

$$UCT_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}} \tag{1}$$

where w_i is the accumulated wins for child node i, n_i is the visit count for the child, N is the parent's visit count, and c is the exploration constant (typically $\sqrt{2}$). The first term represents exploitation (average win rate), while the second encourages exploration of less-visited nodes, effectively balancing exploration and exploitation. After the computational budget is exhausted, MCTS selects the move with the highest visit count at the root, providing a more robust estimate than win rate alone.

C. Board Representation

Our implementation uses a bitboard representation, encoding the 8×8 Othello board as two 64-bit integers one for each player's pieces. This compact representation enables efficient move generation and state updates through bitwise operations. The board state is trivially copyable, allowing efficient host-device transfers and enabling each CUDA thread to maintain independent simulation state.

D. Parallelization Challenges

Following the analysis of Mirsoleimani et al. [7], we identify three main challenges for GPU-based MCTS:

- **Thread Divergence:** Different threads following different tree paths cause warp divergence and SIMD underutilization.
- **Memory Contention:** Multiple threads updating shared nodes (especially near the root) compete for atomic access, causing serialization.
- **Irregular Memory Access:** Pointer-based tree structures exhibit poor spatial locality, reducing cache efficiency.

Our three parallelization strategies represent different trade-offs in mitigating these challenges.

Virtual loss: In parallel MCTS, multiple workers may select the same promising path before their simulations are backpropagated, causing redundant work. A common mitigation is virtual loss: when a worker selects a node, it temporarily applies a small penalty (e.g., incrementing the visit count and/or decrementing the win estimate) so that other workers are biased toward different branches until the real result is available.

III. Methodology

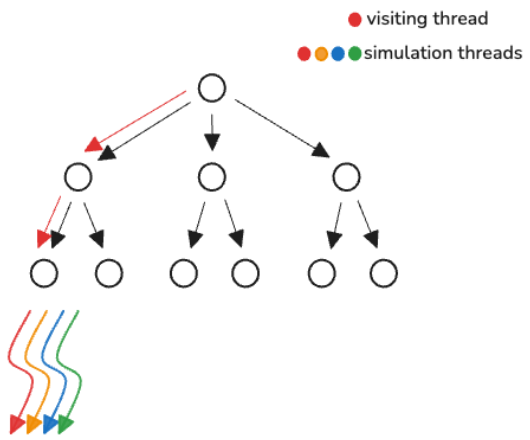
A. CPU Baselines

Before evaluating GPU parallelization strategies, we establish two CPU baselines: a sequential single-threaded implementation and a multi-threaded OpenMP version. These baselines serve as reference points for measuring performance and provide insights into the algorithmic trade-offs when scaling tree search.

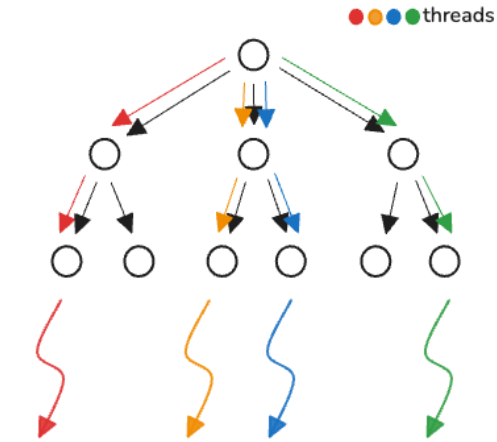
Sequential MCTS: The sequential implementation follows canonical MCTS with UCT. Each iteration is executed serially. The tree is represented using `std::unique_ptr` nodes; each node stores the board state, an unexpanded move list, and atomic statistics for wins and visits to keep the interface compatible with the parallel variants. Selection descends the tree by choosing the child with the maximum UCB value, expansion adds a single child when available, and simulations are simple random playouts.

OpenMP Tree Parallelization: The OpenMP version extends the sequential baseline with multi-threaded tree traversal, following the tree parallelization paradigm where all threads share a single search tree. To allow safe concurrent expansion, each node contains an `omp_lock_t` which is acquired only during the expansion step. Backpropagation updates use atomic variables (`std::atomic<double>` for wins and `std::atomic<long long>` for visits), enabling lock-free updates and reducing synchronization overhead. Each worker thread keeps a thread-local pseudo-random number generator and a local iteration counter, aggregated at thread exit. The implementation performs a check-lock-recheck pattern during selection so that a thread that finds a non-fully-expanded node will acquire the node lock, re-validate the expansion condition, and either expand or resume selection accordingly.

Trade-offs and limitations: We use a time-based termination condition for all baselines and record iteration counts for reporting. The OpenMP design favors low-overhead backpropagation via atomics, but it inherits well-known limitations: contention for expansion near the root can serialize work, per-expansion heap allocations may create allocation hotspots in highly concurrent scenarios, and the pointer-based tree layout reduces spatial locality in caches. Notably, the OpenMP implementation does not use virtual loss. As a result, threads can select the same path before updates are visible, which can increase redundant work.



(a) Leaf parallelization.



(b) Tree parallelization.

Fig. 1. CUDA MCTS parallelization strategies: Leaf and Tree.

B. Leaf Parallelization

Leaf parallelization follows a hybrid CPU–GPU design in which the host maintains the tree operations (selection, expansion and backpropagation) while the GPU performs very large batches of independent simulations. Rather than executing a single rollout per MCTS iteration as in the sequential baseline, this strategy launches N_{sims} parallel playouts from each selected leaf node, aggregating results before backpropagation. This design amortizes kernel launch overhead across many simulations while preserving the standard MCTS tree structure.

Implementation Overview: On each iteration the CPU traverses the tree with the UCT policy to find a leaf, copies the leaf board to device memory, and launches a CUDA kernel with $N_{\text{sims}} = B \times T$ threads (where B is the number of thread blocks and T is the block size). Each GPU thread executes an independent random playout from the copied board state and contributes a per-thread outcome to a block-level accumulator. After the kernel completes, the per-block partial results are reduced to a single aggregated statistic which is copied back to the host for the CPU to then backpropagate it along the selection path.

Kernel Design and optimizations: The simulation kernel uses persistent per-thread cuRAND states initialized at startup to avoid the reseeding overhead. To minimize the global-atomic contention, each block performs an intra-block accumulation in shared memory and only one thread per block writes the block's subtotal to global memory.

Trade-offs and limitations: Leaf parallelization significantly improves playout throughput but has limitations. Execution is synchronous: each iteration calls the GPU and waits for completion, so CPU tree work cannot overlap with GPU simulations. Only a single leaf is batched per iteration and the same fixed number of playouts is executed for every leaf, which prevents adaptive allocation of simulation effort. Finally, batching many independent playouts produces excellent arithmetic throughput but can expose occupancy and memory-access inefficiencies that require profiling and tuning for each GPU.

C. Tree Parallelization (CUDA)

Tree parallelization represents the most ambitious GPU strategy, placing the entire MCTS tree in GPU global memory where thousands of threads concurrently traverse, expand, and update a shared tree structure. This approach maximizes GPU utilization but requires careful synchronization to maintain tree consistency.

Implementation Overview: Unlike leaf parallelization where the CPU manages the tree, a persistent CUDA kernel performs repeated MCTS iterations until a host-set stop flag is raised via mapped, pinned memory. The device holds a preallocated array of `GpuNode` entries which threads reference by integer indices. Work is organized at warp granularity, each warp cooperatively performs selection, expansion, simulation and an aggregated backpropagation step.

Node Pool Architecture: Nodes are compact, index-addressed records that store parent and child indices, a small child count, the move encoding and atomic statistics (visits and wins). Children for a given node are allocated contiguously in the global pool to improve locality and the allocations are handled by a global counter updated with `atomicAdd` so that each expansion reserves a contiguous block of node slots for cache efficiency.

Synchronization Strategy: When a warp finds an unexpanded node, lane 0 attempts to acquire a lightweight expansion lock. A successful expander computes how many child slots are needed, initializes the child nodes, issues a memory fence, and publishes the first child index to release the lock. If allocation would overflow the pool (`start index + needed > max_nodes`) the expansion attempt is aborted. To reduce repeated selection of in-progress paths, the kernel uses a lightweight virtual-loss-like mechanism by atomically incrementing visit counts before descent. This biases other warps away until the update is applied.

Lane 0 computes UCB values and selects the best child; the result is broadcast via `__shfl_sync` to all lanes. All 32 threads maintain synchronized board state. Upon reaching

a leaf, all 32 warp threads independently simulate random playouts. Results are reduced using warp shuffle operations. This yields 32 simulations per warp iteration with minimal synchronization overhead and the aggregated result (32× win values) is backpropagated atomically, updating each ancestor once per warp rather than 32 times.

Root Contention Mitigation: The root is a severe contention hotspot. To reduce global atomics, each warp accumulates visit/win deltas in per-warp shared variables (for example, `shared_root_visits` and `shared_root_wins`) and periodically flushes them to global memory and at kernel exit. This amortization lowers the frequency of expensive global atomics at the root.

Trade-offs and limitations: The CUDA tree approach exchanges host-level locks and dynamic pointer allocation (as used in the OpenMP tree variant) for a static device pool and fine-grained atomics. The OpenMP implementation acquires `omp_lock_t` only during expansion and relies on atomic backpropagation, but does not implement virtual loss; it therefore faces different bottlenecks (heap allocation contention and coarse lock contention). The GPU design avoids OS-level locking overhead and benefits from warp-level reductions, but it must address atomic contention near the root and the fixed-size pool limitation. Threads within a warp must synchronize during selection, causing idle lanes when branches differ in depth. All of those factors require profiling and careful tuning to achieve best performance.

D. Block Parallelization

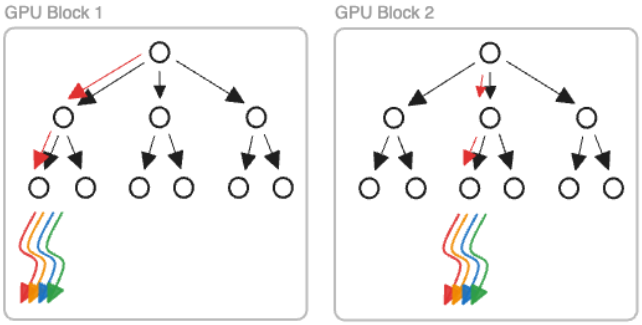


Fig. 2. Block Parallelization

Block parallelization combines elements of both root and leaf parallelization into a hybrid strategy. Each CUDA block maintains an independent search tree (root parallelism), while threads within a block cooperate to run parallel simulations from shared leaf nodes (leaf parallelism). This design eliminates inter-block synchronization while maximizing intra-block SIMD efficiency.

Implementation Overview: At initialization each block is assigned a local node pool (configurable, defaults to 10 000 nodes). Thread 0 in each block performs the serial parts of MCTS for that block selection, expansion and backpropagation while the remaining threads participate in the simulation phase. On a typical iteration thread 0 walks its local tree to a leaf, copies the leaf board into shared memory, calls `__syncthreads()`, and then all threads in the block launch independent playouts from that shared board. The block accumulates results in shared memory and a single thread writes the block subtotal to global memory; when the host stop condition is reached the CPU copies block roots back and aggregates visit counts across blocks to produce a final vote.

Intra-Block Coordination: The block uses shared memory to store the selected board bytes, the current node index and the depth used for win attribution so every thread can simulate from the exact same leaf state. To reduce contention the kernel first accumulates per-block black win totals in shared memory, then performs a single global add per block. This shared-memory-first reduction keeps global atomic traffic low and keeps the simulation phase efficient.

Tree Structure: Unlike tree parallelization's contiguous children layout, block trees use sibling-linked lists, allowing incremental expansion one child at a time by thread 0. After the run finishes the host reads each block's root and its first-level children, sums visit counts for identical moves across blocks and selects the move with maximum aggregated visits. The host aggregation uses a small safety bound when traversing children to avoid pathological loops and to keep the procedure robust. This voting mechanism combines the exploration of B independent trees into a single decision.

Trade-offs and limitations: This approach eliminates cross-block locking at the cost of some trade-offs. Giving thread 0 exclusive responsibility for structural updates simplifies per-block synchronization but can leave other threads idle during selection/expansion phases. Fixed per-block memory partitions simplify allocation but may waste memory on small trees or limit growth on large ones. Because each block searches independently, blocks can redundantly explore similar branches, so overall coverage is not guaranteed to be better than a coordinated shared-tree approach. Finally, the CPU-side aggregation step adds a measurable overhead proportional to the number of blocks and their children, which should be accounted for in end-to-end timing.

IV. Experimental Setup

This section describes the hardware platform, profiling tools, kernel configurations, and tournament methodology used to evaluate the parallelization strategies.

All experiments were conducted on the hardware and software configuration summarized in Tables I and II.

TABLE I
Hardware Specifications

Component	Specification
GPU	NVIDIA GeForce RTX 4060 Laptop GPU
VRAM	8 GB
Driver	NVIDIA driver 590.48.01
SM Frequency	2.10 GHz
Memory (DRAM) Clock	7.00 GHz
Compute Capability	8.9
CPU	13th Gen Intel Core i7-13650HX (14 cores, 20 threads)

TABLE II
SOFTWARE ENVIRONMENT

Component	Specification
CUDA Toolkit	NVIDIA CUDA Toolkit **13.1** (nvcc V13.1.115)
Compiler	NVCC with `-arch=compute_89`; host C++ compiler: `g++ 15.2.1`
Operating System	Linux (EndeavourOS); kernel 6.18.6-arch1-1
OpenMP	GNU OpenMP runtime (`libgomp`); compiler reports `_OPENMP = 201511` (OpenMP 4.5)

A. Profiling Tools

Nsight Systems: NVIDIA Nsight Systems provides timeline-based profiling for system-wide performance analysis. We used it to capture end-to-end execution timelines, including CUDA kernel launches, NVTX ranges for MCTS phases, and host–device transfer activity.

Nsight Compute: NVIDIA Nsight Compute provides detailed kernel-level metrics. We collected per-kernel samples (with a fixed number of launches per kernel) to extract occupancy, IPC, warp-level efficiency, cache behavior, memory throughput, and divergent branch statistics.

Key metrics extracted:

- Kernel execution times and launch counts
- CUDA API timings
- Memory transfer statistics: transfer time, total transfer size and counts
- NVTX range timing for MCTS phases
- Compute throughput: SM utilization, IPC
- Memory throughput: L1/L2 hit rates, DRAM bandwidth
- Occupancy: Theoretical vs. achieved warp occupancy
- Warp efficiency: Active threads per warp, divergent branches

B. Kernel Launch Configuration

All GPU implementations use the following default parameters (from `src/gpu_config.h` and device code):

TABLE III
GPU KERNEL CONFIGURATION

Parameter	Value
Block Size (threads/block)	256 (sweep: 128, 256, 512)
Number of Blocks	64 (sweep: 32, 64, 128, 256)
Total Threads	$256 \times 64 = 16{,}384$
Nodes per Block Tree (Block impl.)	10,000
Max Node Pool (Tree impl.)	5,000,000

V. EXPERIMENTAL RESULTS AND ANALYSIS

This section presents comprehensive performance measurements and tournament results for the five MCTS implementations. We analyze computational throughput, GPU profiling metrics, and game playing strength.

A. Tournament Evaluation

To assess playing strength beyond raw throughput, we conducted round-robin tournaments between implementations. In the experiments reported here each player received a default time budget of **200 ms** per move and pairs played **20 rounds**. This short time control was chosen to enable a statistically significant number of games; however, rankings can be sensitive to the per-move budget, and implementations with higher fixed overheads may be disadvantaged at short limits. More games reduce statistical variance and are recommended for definitive comparisons. Ratings were computed using the Elo system with $K = 32$.

Tournament results provide a measure of effective playing strength, capturing whether throughput improvements translate to better gameplay decisions.

For each implementation, we also measure:

- **Playouts per second (PPS):** Total simulations divided by wall-clock time
- **Tournament Elo:** Relative playing strength
- **Win/Loss record:** Direct head-to-head results

TABLE IV
TOURNAMENT RESULTS (200MS PER MOVE)

Bot	Elo	Wins	Losses	Draws
CUDA Tree	1489.4	136	20	4
OMP MCTS	1449.2	117	39	4
CUDA Block	1250.0	87	64	9
CUDA Leaf	961.4	34	121	5
Serial MCTS	850.0	14	144	2

The GPU tree parallel implementation achieved the highest Elo, combining high simulation throughput with effective tree expansion and UCT selection to produce a dominant win/loss record. The OpenMP tree variant was a close runner-up: although its raw throughput is lower, CPU-based tree updates avoid many GPU atomic-contention effects and therefore maintain greater move diversity per search iteration. Leaf parallelization performed poorly under the strict 200 ms per-move budget because frequent kernel launches limit the number of distinct iterations, and batched playouts bias the search toward early discovered branches, the tree explores only 370 unique leaves. This type of parallelization performs better with more time. The single-threaded baseline confirms that parallelization yields substantial improvements when it preserves broad, balanced exploration.

B. Computational Performance

Table V summarizes playouts per second (PPS) measured for the three GPU implementations with BLOCK_SIZE=256 and NUM_BLOCKS=64. Across the evaluated time limits the tree parallel implementation attains the highest throughput, reaching 17.52M PPS at the 1 s limit approximately 32% higher than the Leaf implementation (13.22M) and about 20% higher than the Block implementation (14.66M). This result runs counter to our initial hypothesis that atomic contention would significantly limit the Tree implementation's performance.

We investigated why the Tree implementation outperformed our expectation despite the assumed atomic contention bottleneck. Our analysis points to three plausibly synergistic causes. First, warp-level aggregation (per-warp accumulators and periodic flushes) reduces the number of global atomic

updates compared to a naive per-thread backpropagation scheme, lowering pressure on high-contention nodes near the root. Second, the kernel's virtual-loss-like behavior (temporarily incrementing visit counts during selection, then applying a single aggregated backpropagation per warp) further reduces the frequency and concurrency of atomic updates on internal nodes by biasing other warps away from in-progress paths. Third, the tree layout and typical search behavior provide some spatial locality: a relatively shallow effective search depth under short budgets and frequent reuse of recently touched nodes can increase L2 reuse and reduce expensive global memory round-trips. While we do not isolate each factor quantitatively in this work, these mechanisms together offer a plausible explanation for the unexpectedly good scaling.

TABLE V
PLAYOUTS PER SECOND VS. TIME LIMIT (BLOCK_SIZE=256, NUM_BLOCKS=64)

Time Limit	Tree CUDA	Leaf	Block
100 ms	14.35M	10.18M	11.33M
200 ms	15.24M	10.50M	12.17M
500 ms	17.45M	12.08M	14.14M
1000 ms	17.52M	13.22M	14.66M

Throughput for all implementations increases as the time limit grows, which indicates that fixed startup costs (RNG initialization, kernel launch overheads and similar startup work) become less significant as runs lengthen. The largest relative gains occur when increasing the time limit up to several hundred milliseconds. Beyond 500 ms the Tree implementation shows near-saturation (17.45M $\rightarrow$ 17.52M from 500 ms to 1 s), whereas Leaf and Block continue to gain modestly.

The Leaf parallelization exhibits the lowest absolute throughput among the GPU approaches. We attribute this to per-iteration CPU–GPU synchronization that constrains overall throughput compared with the device-resident Tree and Block approaches.

Scaling with Thread Count: We conducted a parameter sweep over the block sizes and block counts reported in Table III at a 500 ms time limit.

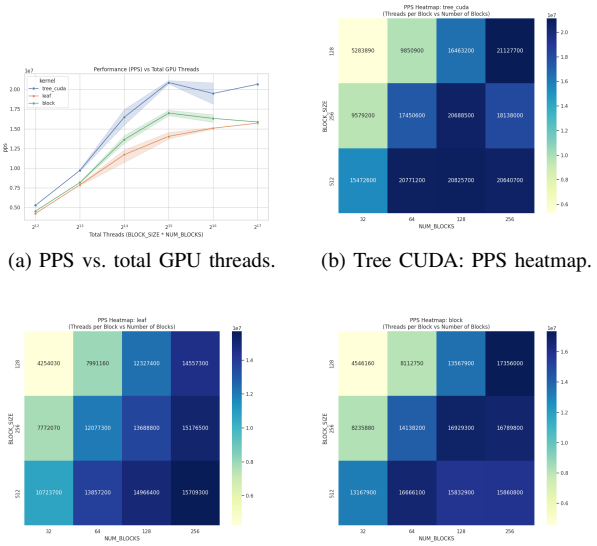
Figure 3a reveals:

- All implementations scale positively with thread count up to ~32K threads.
- Tree parallelization peaks at ~21M PPS with 128 blocks $\times$ 256 threads.
- Block parallelization benefits most from increased block count (more independent trees).
- Leaf parallelization shows diminishing returns beyond 16K threads.

Configuration Heatmaps: Figure 3 combines the PPS scaling curve and the PPS heatmaps for each implementation across the parameter space.

The heatmaps show that:

- Tree CUDA prefers larger block sizes (512 threads) with moderate block counts.



(a) PPS vs. total GPU threads. (b) Tree CUDA: PPS heatmap.

(c) Leaf parallelization: PPS heatmap. (d) Block parallelization: PPS heatmap.

Fig. 3. Scaling behavior and configuration sensitivity: PPS vs. total thread count and PPS heatmaps across BLOCK_SIZE and NUM_BLOCKS.

- Leaf parallelization is relatively insensitive to block size, favoring high block counts.
- Block parallelization shows optimal performance at 128 blocks with 256 threads.

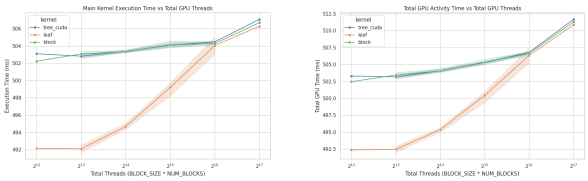
C. GPU Profiling Analysis

Kernel Time Distribution: Table VI shows the kernel execution time breakdown at 500 ms with BLOCK_SIZE=256 and NUM_BLOCKS=64.

TABLE VI
KERNEL TIME ANALYSIS (500MS, 256$\times$64 CONFIGURATION)

Metric	Tree	Leaf	Block
Main Kernel Time (ms)	503.2	494.4	503.2
Total GPU Time (ms)	503.9	495.0	503.9
Memory Transfer (ms)	0.019	0.44	0.65
Kernel Instances	2	370	2

Memory operations remain small in absolute time compared with kernel execution, yet their distribution differs across implementations. The Block profile shows the largest accumulated device-to-host transfer time (0.65 ms across 768 D2H transfers), while Leaf has a combined memcpy/memset overhead of 0.44 ms (370 D2H transfers plus memsets). Tree's measured memory time is negligible (0.02 ms). These differences indicate that the performance gap arises not from lost GPU compute time (the main kernel accounts for 99% of GPU time in all cases) but from differing launch and transfer patterns: Leaf's many short kernel invocations increase launch overhead, and Block's greater number of small transfers increases host–device traffic, both of which reduce effective throughput compared to the persistent-tree execution model.



(a) Main kernel time (b) Total GPU time

Fig. 4. Kernel execution time vs. total thread count.

Memory Transfer Analysis: Figure 5 shows the fraction of total runtime spent in device memory operations (the "memory overhead ratio") for each GPU implementation across the configurations. The Block implementation exhibits the largest memory overhead (0.13% at its worst), driven by the per-block aggregation pattern that produces many small device→host transfers; although this overhead is small in absolute time (sub-millisecond), the repeated transfers increase host–device traffic and reduce effective compute throughput relative to a persistent device-resident kernel.

The Leaf implementation shows a clear downward trend in memory overhead as total thread count rises. This behavior is consistent with per-iteration transfers being amortized as each kernel instance performs more useful work when more GPU threads are active; in practice the per-iteration device-to-host and memset costs become a smaller fraction of the total runtime as parallelism increases.

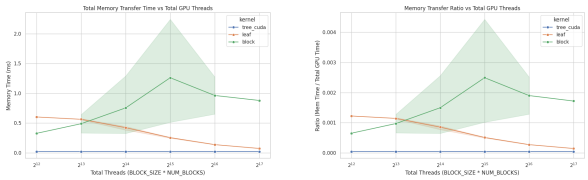
By contrast, the Tree implementation maintains a negligible memory overhead across all tested configurations (below 0.004%). The persistent-tree design issues very few host transfers and therefore keeps the ratio of memory time to total runtime essentially constant and minimal, which helps explain its superior throughput in our measurements.

Occupancy Analysis: The GPU occupancy measures how effectively the hardware resources are utilized.

Key observations:

- All the implementations achieve 80–100% theoretical occupancy.
- Leaf achieves the highest occupancy due to simpler kernel structure.
- Tree and Block show similar occupancy patterns despite different synchronization schemes.

Warp Efficiency and Control Flow: Figures 8 and 9 summarize warp-level activity, instruction throughput (IPC),



(a) Memory transfer time (b) Memory time ratio

Fig. 5. Memory transfer overhead vs. total threads.

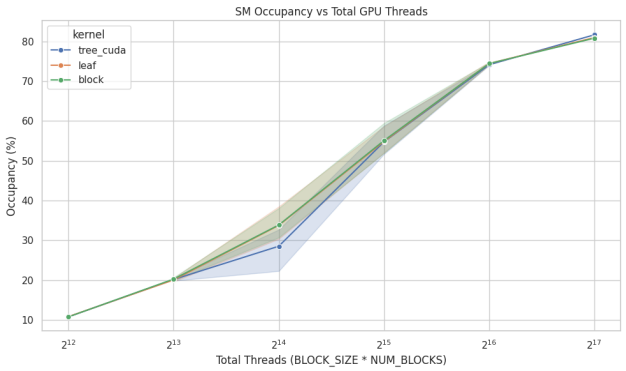
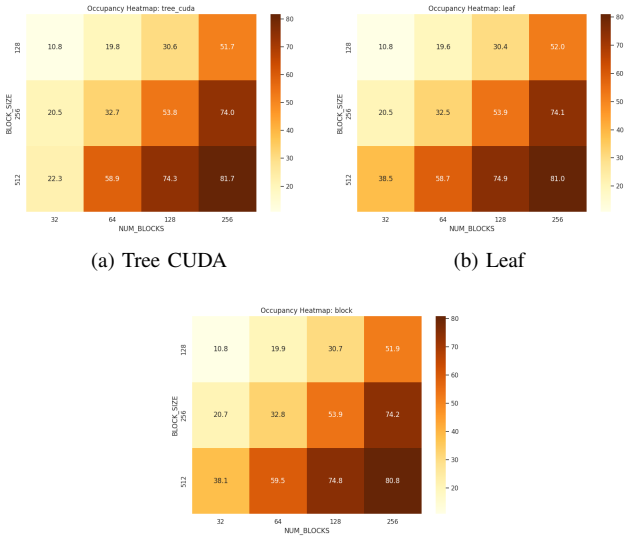


Fig. 6. Achieved occupancy vs. total thread count for each implementation.

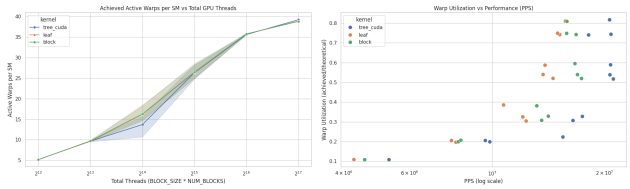


(a) Tree CUDA (b) Leaf

(c) Block

Fig. 7. Occupancy heatmaps (BLOCK_SIZE vs. NUM_BLOCKS) for each implementation.

and branch divergence for the 500 ms resource sweep. For each configuration, we compute *warp utilization* as the ratio of achieved to theoretical active warps per SM, and we use the Nsight Compute metric `executed_ipc_active` as an IPC proxy.



(a) Active warps vs. threads (b) Warp utilization vs. PPS

Fig. 8. Warp-level metrics showing correlation between warp activity and throughput.

Higher warp utilization and higher IPC indicate better latency hiding and fewer stalls, which typically increases

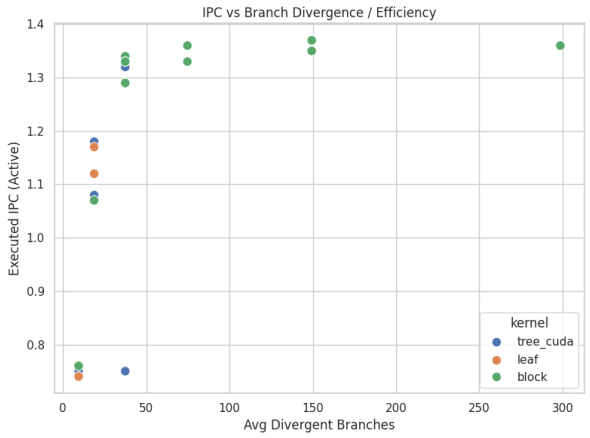


Fig. 9. Instructions per cycle vs. branch divergence. Higher IPC indicates better instruction throughput.

PPS. The positive association with divergence should be interpreted cautiously: across configurations, "raw" divergence can increase simply because higher-throughput kernels execute more dynamic control flow per unit time. Therefore, divergence remains a cost to mitigate when feasible, but it is not, by itself, a reliable predictor of end-to-end throughput.

Memory Bandwidth and Cache Behavior: Figure 10 reports the achieved memory throughput as a function of total thread count. This plot provides a direct view of whether configurations become bandwidth-limited as parallelism increases.

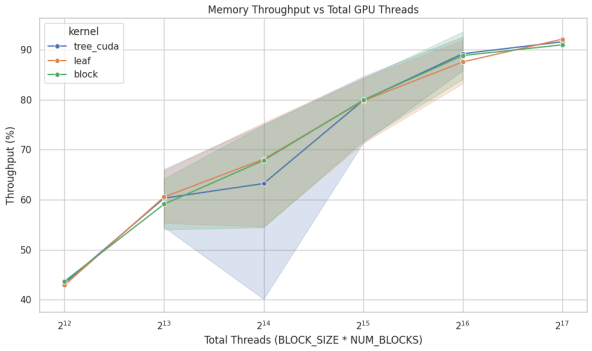


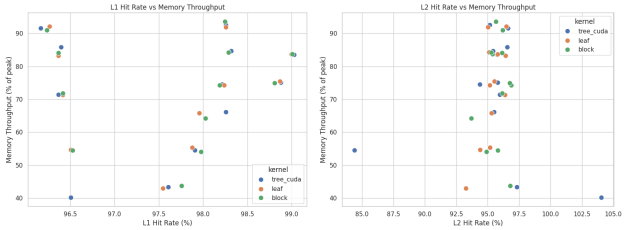
Fig. 10. Memory throughput vs. total thread count.

To better understand how efficiently memory traffic is served, Figure 11 relates L1 and L2 hit rates to memory throughput. Higher hit rates indicate that a larger fraction of accesses is satisfied on-chip, reducing DRAM pressure and improving effective throughput.

VI. Discussion

A. Key Findings Analysis

Tree Parallelization Outperforms Expectations: A surprising result is that Tree CUDA parallelization achieves both the highest throughput (17–21M PPS) and the strongest tournament performance (85% win rate, 1489 Elo). This outcome can



(a) L1 cache hit rate (b) L2 cache hit rate

Fig. 11. Cache hit rates vs. memory throughput showing cache efficiency.

be explained by the way the kernel organizes work at warp granularity: a full warp cooperates on the tree operations while simultaneously generating simulations. In particular, lane 0 performs selection and expansion, then broadcasts decisions to the other lanes so that all 32 threads remain aligned on the same path. In addition, the use of a virtual-loss-like mechanism (temporarily incrementing visit counts before descent) reduces redundant selection of the same in-progress branches, and shared-memory buffering amortizes expensive global atomics at the root by accumulating updates locally before flushing them to global memory.

Leaf Parallelization's Exploration Deficit: Leaf parallelization achieves competitive raw throughput (12–15M PPS) but underperforms in tournament play. This performance gap highlights a fundamental distinction between simulation throughput and search depth. In MCTS, the decision quality depends on the number of iterations (sequential selection/expansion steps), not just the total number of simulations. Because Leaf parallelization requires a CPU-GPU synchronization at every iteration, it is structurally limited to a few hundred iterations under tight 200ms time controls ($\sim$370 unique leaves explored). Consequently, while it generates millions of playouts, they are concentrated on a very narrow subset of the game tree. This strategy is effectively a throughput-oriented approach that lacks the search reactivity of Tree-based strategies. While Leaf parallelization might become competitive with much larger time budgets where the search can naturally broaden, it is inherently ill-suited for the low-latency requirements of competitive real-time play.

a) OpenMP's Competitive Strength: OpenMP MCTS achieves a 73% win rate with only $\sim$500K PPS, roughly 35$\times$ fewer playouts than Tree CUDA. This indicates that its decision quality is driven less by raw simulation volume and more by the structure of its search: performing (approximately) one rollout per selected leaf naturally maintains exploration diversity, and CPU-based tree traversal avoids the high-root atomic contention that can arise in GPU-resident shared trees. Under moderate time budgets, these effects can outweigh the advantage of higher throughput.

B. The Throughput-Strength Paradox

A central finding of this work is that **raw throughput does not reliably predict playing strength**. The most striking example is OpenMP MCTS, which achieves a 73% win rate

with only ∼500K PPS, 35× fewer playouts than Tree CUDA while still reaching a competitive Elo (1449 vs. 1489). This paradox can be understood through the exploration–exploitation balance inherent to MCTS. Exploration corresponds to visiting a diverse set of game states to discover strong moves, whereas exploitation concentrates simulations on already-promising branches to refine value estimates. Leaf parallelization's batch simulation model strongly favors exploitation: each selected leaf receives thousands of playouts, which concentrates statistics on early discovered branches. In contrast, OpenMP's one-simulation-per-leaf behavior preserves exploration diversity, enabling the discovery of stronger moves even with a lower total playout count.

C. Parallelization Strategy Trade-offs

Table VII summarizes the qualitative trade-offs between strategies.

TABLE VII
PARALLELIZATION STRATEGY COMPARISON

Strategy	Throughput	Diversity	Complexity	Strength
Serial CPU	Very Low	High	Low	Baseline
OpenMP	Low	High	Medium	Strong
Leaf CUDA	High	Low	Low	Weak
Block CUDA	High	Medium	Medium	Medium
Tree CUDA	Highest	High	High	Strongest

D. Practical Recommendations

Based on our findings, we offer the following guidance for practitioners:

1) **For maximum playing strength:** Use Tree CUDA parallelization with sufficient time budget. The implementation complexity is justified by superior performance.
2) **For resource-constrained environments:** Tree OMP provides excellent strength and is simpler to implement and debug.
3) **Avoid Leaf parallelization for competitive play:** Despite reasonable throughput, the exploration bias severely limits playing strength under typical time controls.
4) **Block parallelization for experimentation:** Independent trees enable easy result aggregation and debugging, making it suitable for development and analysis.

E. Threats to Validity

Several factors may limit the generalizability of our results:

- **Game-specific bias:** Othello's branching factor and game length may favor certain parallelization strategies differently than other games.
- **Hardware dependence:** Results on RTX 4060 laptop may not transfer to other GPU architectures.
- **Time control sensitivity:** Relative rankings may change with the per-move time budget (see Section V, Tournament Evaluation). The leaf parallelization works better with a higher time limit.
- **Implementation effects:** Performance differences may partly reflect implementation quality rather than fundamental algorithmic trade-offs.

VII. CONCLUSION

This paper presented a comparative study of GPU parallelization strategies for Monte Carlo Tree Search applied to Othello. We implemented and evaluated five approaches: sequential CPU, OpenMP tree parallel, CUDA leaf parallelization, CUDA tree parallelization, and CUDA block parallelization.

A. Key Findings

1) **Tree parallelization achieves both highest throughput and strongest play.** Contrary to expectations about atomic contention, the fully GPU-resident tree with warp-level collaboration achieves 17–21M playouts per second while maintaining 85% tournament win rate (1489 Elo).
2) **Throughput does not predict playing strength.** OpenMP MCTS achieves 73% win rate with 35× fewer playouts than GPU implementations, demonstrating that exploration quality matters as much as simulation quantity.
3) **Leaf parallelization underperforms despite reasonable throughput.** The batch simulation model biases tree exploration, resulting in only 21% win rate (961 Elo) despite 12M PPS. Under tight time controls, only ∼370 unique leaves are explored.
4) **Block parallelization offers a middle ground.** Independent trees per block avoid synchronization overhead but sacrifice unified exploration, achieving 54% win rate (1250 Elo).

B. Main Insight

The central contribution of this work is the empirical demonstration that MCTS parallelization should prioritize tree exploration quality over raw simulation throughput. Strategies that maintain diverse tree exploration whether through traditional one-simulation-per-leaf (OpenMP) or sophisticated GPU tree coordination with virtual loss (Tree CUDA) outperform approaches that sacrifice exploration for batch efficiency.

C. Practical Guidelines

For practitioners implementing parallel MCTS:

- Use **Tree CUDA** when maximum strength is required.
- Use **OpenMP** for resource-constrained or development environments it provides competitive strength with simpler implementation.
- **Avoid Leaf parallelization** for competitive play under typical low time controls.
- Consider **Block parallelization** for experimental setups where independent tree analysis is valuable.

D. Future Work

Several directions remain for investigation:

- **Hybrid approaches:** Combining Tree and Leaf parallelization with adaptive leaf batching based on node importance.
- **Neural network integration:** Replacing random rollouts with learned value/policy networks following the AlphaZero paradigm.
- **Multi-GPU scaling:** Extending Tree parallelization across multiple GPUs with inter-device synchronization.

- **Generalization:** Evaluating whether the throughput-strength trade-offs observed in Othello generalize to games with different characteristics (Go, Chess, Hex).

REFERENCES

[1] H. Takizawa, "Othello is solved," 2024. [Online]. Available: https://arxiv.org/abs/2310.19387

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[3] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan 2016. [Online]. Available: https://doi.org/10.1038/nature16961

[5] A. Norelli and A. Panconesi, "Olivaw: Mastering othello without human knowledge, nor a fortune," *IEEE Transactions on Games*, vol. 15, no. 2, pp. 285–291, 2023.

[6] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 60–71.

[7] K. Rocki and R. Suda, "Large-scale parallel monte carlo tree search on gpu," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 2034–2037.