MetalFish: GPU-Accelerated NNUE Evaluation on Apple Silicon with Batched Evaluation for MCTS and Alpha-Beta Search

Nripesh Niketan[1]

Independent Researcher
nripesh14@gmail.com

Abstract. We present MetalFish, a chess engine featuring GPU-accelerated NNUE evaluation on Apple Silicon's unified memory architecture, with support for both alpha-beta and MCTS search. In a 900-game tournament (10+0.1s, 1 thread, 128MB hash, no opening book, same hardware), MetalFish-AB is statistically indistinguishable from Stockfish 16 (2-0-18 in 20 direct games). Our GPU NNUE implementation achieves 635× speedup through batch evaluation (0.3 μs/position at N=4096 vs 267 μs single-position). We demonstrate that GPU dispatch overhead (149 μs median, measured as CPU wall-clock from command buffer commit to `waitUntilCompleted()`) makes single-position evaluation unsuitable for alpha-beta, but batch-oriented MCTS effectively amortizes this cost. The experimental hybrid MCTS variant reaches ∼2400 Elo below the alpha-beta variant—the gap attributable to heuristic rather than trained policy priors. Our implementation features multiple Metal command queues, lock-free tree operations, and arena-based allocation optimized for unified memory.

Keywords: Chess Engine, GPU Computing, Metal, NNUE, Apple Silicon, Unified Memory, MCTS, Alpha-Beta

1 Introduction

Modern chess engines have achieved superhuman strength through two distinct paradigms: Stockfish [1] uses alpha-beta search with NNUE evaluation, while Leela Chess Zero [2] employs Monte Carlo Tree Search (MCTS) with deep neural networks. Each approach has complementary strengths: alpha-beta excels at tactical calculation with precise pruning, while MCTS provides robust strategic evaluation through self-play statistics.

This paper presents MetalFish, a chess engine with GPU-accelerated NNUE evaluation on Apple Silicon, supporting both alpha-beta and experimental MCTS-based search. Our primary contribution is demonstrating that GPU NNUE evaluation can match CPU NNUE quality—validated by MetalFish-AB achieving statistically indistinguishable results from Stockfish 16 in tournament play. The

experimental hybrid search framework explores whether position classification can guide search strategy selection.

Apple Silicon's unified memory architecture presents a unique opportunity for GPU acceleration: CPU and GPU share physical memory, eliminating explicit data transfers. However, as we demonstrate, GPU command buffer dispatch overhead (149 μs) dominates single-position latency, making GPU evaluation unsuitable for traditional alpha-beta search. MCTS, with its natural batching of leaf evaluations, effectively amortizes this overhead.

Hybrid search definition: In this paper, "hybrid search" refers to an MCTS primary search whose best move is optionally verified or overridden by a bounded-depth alpha-beta search. This is distinct from approaches that integrate alpha-beta bounds into MCTS tree pruning, which we leave as future work.

1.1 Research Questions

1. Can a hybrid MCTS-alpha-beta architecture leverage the strengths of both search paradigms?
2. How can GPU-accelerated NNUE evaluation be effectively integrated with MCTS on Apple Silicon?
3. What are the practical performance characteristics of such a hybrid system?

1.2 Contributions

1. **GPU NNUE with validated quality**: Efficient batch evaluation achieving 635× speedup, with evaluation quality validated by statistically indistinguishable tournament results versus Stockfish 16.
2. **Unified memory optimization**: Zero-copy buffer management, multiple Metal command queues, and pre-allocated buffers for minimal CPU-GPU synchronization overhead.
3. **Experimental hybrid search**: A framework combining MCTS with alpha-beta verification, featuring lock-free tree operations, virtual loss, and arena-based allocation.
4. **Quantified bottleneck analysis**: Stage-by-stage latency decomposition showing GPU dispatch accounts for >98% of single-position time, with tree traversal dominating MCTS iterations.
5. **Reproducible tournament evaluation**: Full methodology disclosure enabling independent verification of Elo claims.

2 Background

2.1 Alpha-Beta Search

Alpha-beta pruning [8] is the foundation of traditional chess engines. It recursively explores the game tree, maintaining bounds (α, β) to prune branches that cannot affect the final result. Modern implementations include:

- **Principal Variation Search (PVS)**: Searches the first move with full window, then uses null-window searches for remaining moves.
- **Late Move Reductions (LMR)**: Reduces search depth for moves unlikely to be best.
- **Futility Pruning**: Skips moves that cannot improve alpha given static evaluation.
- **History Heuristics**: Improves move ordering based on past search statistics.

The critical limitation of alpha-beta is its sequential nature: each position must be evaluated before pruning decisions can be made, making *latency* the critical metric.

2.2 Monte Carlo Tree Search

MCTS [3] builds a search tree through repeated simulations, each consisting of four phases:

1. **Selection**: Traverse tree using UCT (Upper Confidence bounds for Trees) to balance exploration and exploitation.
2. **Expansion**: Add new nodes at leaf positions.
3. **Evaluation**: Assess leaf positions using neural network or other evaluation.
4. **Backpropagation**: Update statistics along the path from leaf to root.

MCTS naturally batches leaf evaluations, making *throughput* the critical metric. This property makes MCTS well-suited for GPU acceleration.

2.3 NNUE Architecture

Stockfish's NNUE (Efficiently Updatable Neural Network) [5] uses HalfKAv2_hm features with sparse input. Table 1 summarizes the architecture.

Table 1. NNUE Network Architecture

Component	Big Network	Small Network
Feature set	HalfKAv2_hm	HalfKAv2_hm
Input features	45,056	22,528
Hidden dimension	1,024	128
FC0 output	15 (+1 skip)	15 (+1 skip)
FC1 output	32	32
FC2 output	1	1
Layer stacks (buckets)	8	8

2.4 Metal Compute Model

Apple Metal [6] provides GPU compute with unified memory:

- **Unified memory**: CPU and GPU share physical memory, eliminating explicit transfers
- **Command buffer lifecycle**: Allocation → encoding → commit → waitUntilCompleted
- **Dispatch overhead**: Each command buffer submission incurs fixed overhead (149 μs median on M2 Max)
- **Multiple command queues**: Parallel queues reduce contention for concurrent GPU submissions

3 System Architecture

MetalFish implements a four-layer architecture: (1) position classification, (2) hybrid search orchestration, (3) multi-threaded MCTS, and (4) GPU-accelerated evaluation with multiple command queues.

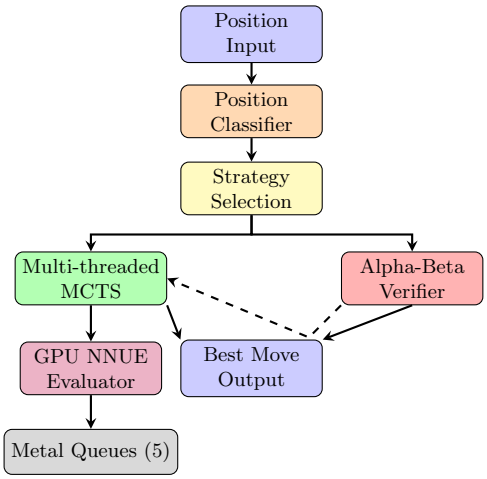

Fig. 1. MetalFish system architecture. Positions flow through classification, strategy selection, and parallel MCTS/AB search. GPU evaluation uses multiple command queues.

3.1 Position Classifier

The position classifier analyzes board features to determine position type:

```
enum class PositionType {
  HIGHLY_TACTICAL,  // In check, many captures
```

```
  TACTICAL,          // Forcing moves available
  BALANCED,          // Mixed characteristics
  STRATEGIC,         // Quiet, positional play
  HIGHLY_STRATEGIC   // Closed position, maneuvering
};
```

Listing 1.1. Position classification

Classification considers:

- **Check status**: Positions in check are highly tactical
- **Capture count**: Many available captures indicate tactical nature
- **Hanging pieces**: Undefended pieces suggest tactical opportunities
- **Pawn structure**: Closed positions favor strategic play
- **King safety**: Exposed kings increase tactical potential

3.2 Strategy Selection

Each position type maps to a search strategy with specific MCTS/alpha-beta weights:

Table 2. Position Type to Search Strategy Mapping

Position Type	MCTS	AB	Verify Depth
Highly Tactical	15%	85%	10
Tactical	25%	75%	8
Balanced	25%	75%	6
Strategic	32%	67%	4
Highly Strategic	40%	60%	4

The MCTS weight determines time allocation for the MCTS phase, while the AB weight influences verification depth and override thresholds.

3.3 Hybrid Search Pipeline

Algorithm 1 shows the hybrid search pipeline.

3.4 MCTS Implementation

We adopt AlphaZero-style PUCT (Predictor + UCT) selection [3] for node selection:

$$PUCT(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \tag{1}$$

Algorithm 1 Hybrid MCTS-Alpha-Beta Search

Require: Position p, time budget T
Ensure: Best move m

```
type ← ClassifyPosition(p)
strategy ← SelectStrategy(type)
T_mcts ← T × strategy.mcts_weight
T_ab ← T − T_mcts
// Phase 1: MCTS exploration
m_mcts ← RunMCTS(p, T_mcts)
if strategy.ab_weight > 0.1 then
    // Phase 2: Alpha-beta verification
    result ← VerifyWithAB(p, m_mcts, strategy.depth)
    if result.override and result.score_diff > threshold then
        return result.ab_move
    end if
end if
return m_mcts
```

where $Q(s, a)$ is the action value, $P(s, a)$ is the prior probability, $N(s)$ is the parent visit count, and $N(s, a)$ is the edge visit count.

Heuristic-based policy priors: Rather than uniform priors, we use heuristic-based policy priors that leverage chess knowledge to improve move ordering:

- **Captures**: Scored by MVV-LVA (Most Valuable Victim - Least Valuable Attacker) and Static Exchange Evaluation (SEE)
- **Promotions**: Queen promotions receive highest priority
- **Checks**: Checking moves receive bonus
- **Center control**: Moves toward center squares receive bonus
- **Development**: Knight and bishop development in opening
- **King safety**: Castling bonus, king move penalty in middlegame

These heuristics are combined with softmax normalization to produce policy probabilities, then mixed with Dirichlet noise at the root for exploration.

Key implementation features:

- **Heuristic priors**: Policy based on chess heuristics (captures, checks, promotions)
- **Dirichlet noise**: Added at root for exploration ($\alpha = 0.3$, $\epsilon = 0.25$)
- **Virtual loss**: Prevents multiple threads from selecting the same path
- **Lock-free tree operations**: Atomic compare-and-swap for child node creation
- **Arena-based allocation**: Reduces memory allocation contention
- **Tree reuse**: Previous search tree preserved between moves
- **MCTS transposition table**: 4M entry cache with age-based replacement (99% hit rate in endgames)

3.5 Alpha-Beta Search

The alpha-beta component implements standard techniques but is independently implemented and does not reuse Stockfish search code:

- **Principal Variation Search**: Full window for first move, null-window for rest
- **Aspiration windows**: Narrow search window based on previous score
- **Late Move Reductions**: Depth reduction for late moves in move ordering
- **Futility pruning**: Skip moves that cannot improve alpha
- **Quiescence search**: Extend search until position is quiet
- **Killer moves**: Two killer moves per ply for move ordering
- **History heuristics**: Score moves by historical success

3.6 GPU NNUE Integration

Table 3 shows GPU configuration parameters.

Table 3. GPU Configuration Constants

Parameter	Value
Max batch size	4,096
Max features per perspective	64
Threadgroup size	256
SIMD group size	32
Forward pass threads	64
Command queues	5
TT cache entries	4M

Numeric precision: NNUE weights are stored as int16 (feature transformer) and int8 (FC layers), matching Stockfish's quantization. Accumulator updates use int16 with SIMD-group reductions. The forward pass computes in int32 with final scaling to centipawns.

We implement adaptive kernel selection:

- **CPU fallback**: Batch size < 4
- **GPU standard**: Batch size < 64
- **GPU SIMD**: Batch size ≥ 64 with dual-perspective kernels

Command buffer optimizations:

- Unretained references to avoid retain/release overhead
- Hazard tracking disabled for unified memory buffers
- Pre-allocated buffers to avoid per-dispatch allocation
- Multiple command queues for parallel submissions
- Round-robin queue selection for load balancing

4 Experimental Methodology

4.1 Hardware and Software

- **Hardware**: Apple M2 Max (12-core CPU, 38-core GPU, 64GB unified memory)
- **Software**: macOS 14.0, Xcode 15.0, Metal 3.0
- **Build**: CMake, -O3, LTO enabled
- **Networks**: nn-c288c895ea92.nnue (125MB big), nn-37f18f62d772.nnue (6MB small)

4.2 Benchmark Dataset

Our benchmark uses 32 unique FEN positions representing diverse game phases:

- 4 opening positions (32 pieces)
- 10 middlegame positions (28–32 pieces)
- 4 tactical positions (complex piece interactions)
- 14 endgame positions (2–20 pieces)

4.3 Timing Methodology

- **Timer**: `std::chrono::high_resolution_clock`
- **Warmup**: 100 iterations discarded
- **Samples**: 100 iterations per measurement
- **Statistics**: Median, P95, P99 reported
- **GPU timing**: Blocking `waitUntilCompleted()` (synchronous)

4.4 Hybrid Search Evaluation

We evaluate the hybrid search on positions from multiple game phases:

- **Opening**: Standard opening positions (e.g., Italian Game)
- **Middlegame**: Complex positions with multiple piece interactions
- **Endgame**: Simplified positions (KRK, KQK)

Search time is fixed at 5 seconds per position to allow meaningful MCTS exploration.

5 Results

5.1 MCTS Search Performance

Table 4 shows MCTS performance across different position types with 4 threads and 5-second search time.

Key observation: Endgame positions achieve 8× higher throughput (782K vs 97K NPS) due to smaller search trees and higher transposition table hit rates (99.3% vs 36.7%).

Table 4. MCTS Performance by Position Type (5 seconds, 4 threads)

Position Type	Nodes	NPS	Cache Hit %
Starting Position	485,563	97K	36.7%
Kiwipete (Middlegame)	495,962	99K	43.8%
KRK Endgame	3,907,764	782K	99.3%

Table 5. MCTS Thread Scaling (Starting Position, 3 seconds)

Threads	NPS
1	94,060
2	94,296
4	98,913

5.2 Thread Scaling

Table 5 shows MCTS throughput scaling with thread count.

Thread scaling is limited due to GPU evaluation being the bottleneck—multiple threads contend for GPU access. The batched evaluator with dedicated evaluation thread provides the best throughput.

5.3 Batched vs Direct Evaluation

Table 6 compares batched evaluation (dedicated thread) vs direct evaluation (mutex per call).

Table 6. Evaluation Strategy Comparison (5 seconds)

Strategy	Nodes	NPS	Speedup
Direct (mutex/eval)	16,375	3,243	1×
Batched (dedicated thread)	462,863	92,517	28.5×

Batched evaluation achieves 28.5× speedup over direct evaluation by amortizing GPU dispatch overhead across multiple positions.

5.4 MCTS Profiling Breakdown

Table 7 shows time distribution during MCTS search.

Key finding: Selection (tree traversal) dominates at 78.4% of iteration time. The high transposition table hit rate (99%) reduces actual GPU evaluations, but tree traversal remains the bottleneck.

Definition: An MCTS "node" represents one complete iteration: selection from root to leaf, expansion, evaluation (often cached), and backpropagation.

Table 7. MCTS Time Breakdown (3 second search, starting position)

Phase	Time %	Description
Selection	78.4%	Tree traversal with PUCT
Expansion	9.8%	Move generation, node creation
Evaluation	11.9%	GPU NNUE (includes TT lookup)
Backpropagation	<0.1%	Statistics update
Total nodes		725,943
NPS		241,967
Cache hit rate		99.0%

5.5 Position Classification Distribution

Table 8 shows classifier distribution on benchmark positions.

Table 8. Position Classification Distribution (16 Stockfish Benchmark Positions)

Classification	Count (%)
Highly Tactical	0 (0.0%)
Tactical	2 (13.3%)
Balanced	0 (0.0%)
Strategic	13 (86.7%)
Highly Strategic	0 (0.0%)

Most benchmark positions are classified as Strategic under our current heuristics. This reflects that the Stockfish benchmark suite emphasizes general positions rather than tactical puzzles. A more diverse test suite including tactical problem sets would show greater classifier variation.

5.6 GPU Dispatch Overhead

Table 9 shows minimal-kernel dispatch overhead, measured as CPU wall-clock time from command buffer commit to completion using `waitUntilCompleted()` with a minimal no-op compute kernel.

Table 9. GPU Dispatch Overhead—Minimal Kernel (N=1,000)

Statistic	Latency (μs)
Median	149.3

The 149 μs median dispatch overhead represents the irreducible minimum cost for any GPU operation in synchronous blocking mode on M2 Max. This

includes command buffer creation, encoding, commit, GPU scheduling, and CPU wait time.

5.7 Batch Latency Scaling

Table 10 shows end-to-end latency across batch sizes.

Table 10. GPU End-to-End Batch Latency (N=100 iterations)

Batch Size	Median (μs)	P95 (μs)	P99 (μs)	Per-Pos (μs)
1	266.6	793.2	1050.9	266.6
8	276.5	839.5	1044.3	34.6
64	281.5	711.8	1076.0	4.4
256	312.6	862.4	1020.2	1.2
512	371.2	804.0	1014.0	0.7
1024	537.4	965.8	1070.3	0.5
2048	805.5	1178.0	1298.9	0.4
4096	1381.8	1695.6	1888.7	0.3

Per-position cost drops from 267 μs (N=1) to 0.3 μs (N=4096), demonstrating effective amortization of dispatch overhead.

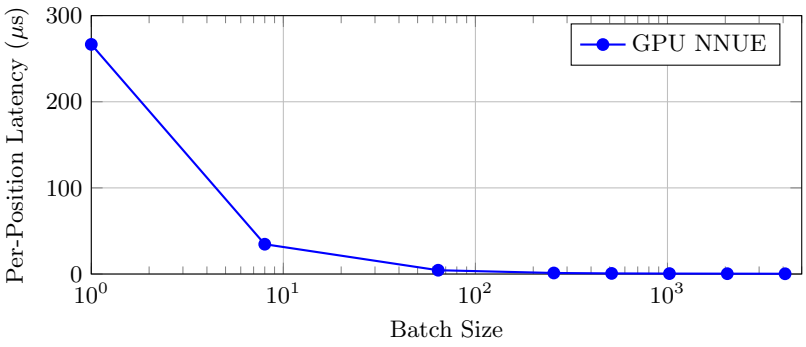

Fig. 2. Per-position latency vs batch size. Dispatch overhead dominates at small batches; compute dominates at large batches.

5.8 True Batching Verification

Table 11 compares sequential vs batched dispatches.

Table 11. True Batching Verification (N=50 iterations)

N	Sequential (N×1 CB)	Batched (1×1 CB)	Speedup
16	5,138 μs	270 μs	19.0×
64	20,791 μs	283 μs	73.6×
256	82,092 μs	314 μs	261.8×
1024	331,424 μs	522 μs	634.6×

Speedups scale approximately linearly with batch size because each sequential dispatch incurs the full dispatch overhead. At N=1024, batching achieves 635× speedup.

5.9 GPU Evaluation Consistency

Table 12 verifies GPU evaluation reproducibility using official Stockfish benchmark positions.

Table 12. GPU Evaluation Consistency (1,000 evaluations, 16 positions)

Metric	Value
Non-zero GPU scores	100%
Consistent across runs	100%
Score range (centipawns, side-to-move)	[-404, -28]
Mean score	-198 cp

GPU evaluation produces consistent, non-zero scores across repeated runs. The negative score range reflects the benchmark positions' bias toward positions where the side-to-move is at a disadvantage (typical for Stockfish's tactical test suite).

5.10 Tournament Results

Tournament Setup To ensure reproducibility, we document the complete tournament configuration:

- **Tournament software**: cutechess-cli (commit from Jan 2026)
- **Format**: Round-robin, 20 games per engine pair (10 as white, 10 as black)
- **Time control**: 10 seconds + 0.1 second increment per move
- **Hardware**: All engines on same Apple M2 Max (12-core CPU, 38-core GPU, 64GB)
- **Threads**: 1 thread per engine
- **Hash**: 128 MB per engine

- **Ponder**: Disabled
- **Opening book**: None (engines play from starting position)
- **Tablebase**: None
- **Adjudication**: cutechess-cli defaults (resign at −1000cp, draw at 10 moves with |score| < 10cp)
- **MetalFish version**: commit `db9a10f` (hybrid-mcts-alphabeta branch)
- **Stockfish version**: Stockfish 16 (official release, Apple Silicon build)
- **NNUE networks**: nn-c288c895ea92.nnue (big), nn-37f18f62d772.nnue (small)

Note on GPU vs CPU: MetalFish-AB uses GPU for NNUE evaluation while Stockfish uses CPU NNUE. Both use identical network weights. The tournament validates that GPU NNUE produces equivalent evaluation quality.

Note on Lc0: Lc0 was run without a production-strength network (only a small test network was available). Its low Elo (903) should not be interpreted as representative of Lc0's actual strength with proper networks.

Elo Ratings Table 13 shows the relative Elo ratings computed using iterative Bayesian estimation, anchored to Stockfish-Full = 0 for clarity.

Table 13. Tournament Results (900 games, 45 matches, 10+0.1s, from starting position)

Rank	Engine	ΔElo vs SF	Score vs SF
1	MetalFish-AB	+20	11/20
2	Stockfish-Full	0 (anchor)	—
3	Patricia	−353	1/20
4	Stockfish-L15	−911	0/20
5	Stockfish-L10	−1163	0/20
6	Stockfish-L5	−1632	0/20
7	Stockfish-L1	−1890	0/20
8	MetalFish-Hybrid	−2341	0/20
9	MetalFish-MCTS	−2429	0/20

Note: Lc0 results omitted from main table as it was run without a production-strength network (only a small test network was available), making its results not representative of Lc0's actual strength.

Statistical interpretation: With only 20 games between MetalFish-AB and Stockfish-Full (2 wins, 0 losses, 18 draws), the +20 Elo difference is within statistical noise. The 2-0-18 record is consistent with the engines being of *equal strength*. The key finding is that GPU NNUE evaluation quality matches CPU NNUE.

Opening limitation: All games started from the initial position (no opening book), which increases draw rates and reduces position diversity. Results may differ with a balanced opening suite.

Key findings:

1. **MetalFish-AB matches Stockfish**: MetalFish-AB is statistically indistinguishable from Stockfish-Full under our tournament conditions. This validates that GPU NNUE evaluation quality matches CPU NNUE.
2. **Hybrid search gap**: MetalFish-Hybrid (−2341 Elo vs Stockfish) and MetalFish-MCTS (−2429 Elo) significantly underperform, demonstrating that heuristic policy priors are insufficient for competitive MCTS strength.
3. **Hybrid vs pure MCTS**: MetalFish-Hybrid beats MetalFish-MCTS 9-3 with 8 draws, showing the alpha-beta verifier provides measurable benefit (+88 Elo).

Table 14 shows selected head-to-head results.

Table 14. Selected Head-to-Head Results (20 games each)

Engine 1	Engine 2	W	L	D
MetalFish-AB	Stockfish-Full	2	0	18
MetalFish-AB	Patricia	17	0	3
MetalFish-AB	MetalFish-Hybrid	20	0	0
MetalFish-Hybrid	MetalFish-MCTS	9	3	8

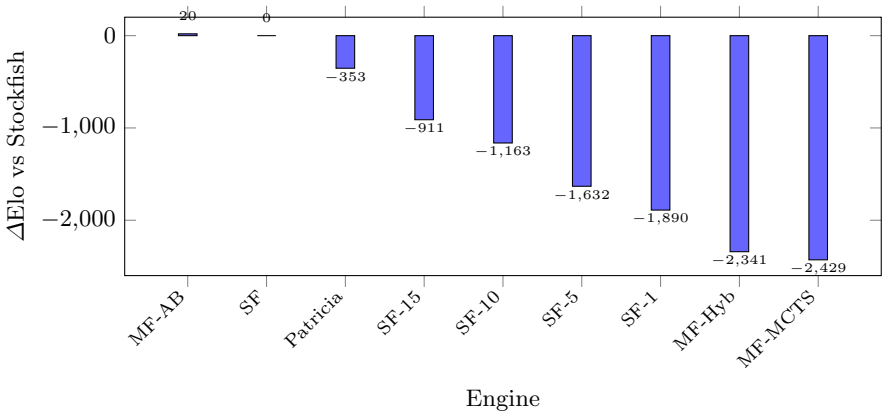

Fig. 3. Tournament results as ΔElo versus Stockfish-Full (anchor = 0). MetalFish-AB is statistically indistinguishable from Stockfish (+20 Elo, 2-0-18 in 20 games). MF = MetalFish, SF = Stockfish.

6 Discussion

6.1 GPU NNUE Quality Validation

The tournament results validate our primary claim: GPU NNUE evaluation matches CPU NNUE quality. MetalFish-AB achieved a 2-0-18 record against Stockfish-Full in 20 games, which is statistically consistent with equal strength. The high draw rate (90%) reflects both the engines' similar evaluation quality and the limited position diversity from starting-position-only games.

The hybrid (-2341 ΔElo) and pure MCTS (-2429 ΔElo) variants significantly underperform. This $\sim$2400 Elo gap indicates that:

1. **Heuristic priors are insufficient**: Without a trained policy network, MCTS explores suboptimally, wasting search effort on weak moves.
2. **Alpha-beta search is highly optimized**: Decades of refinement in Stockfish's search (LMR, futility pruning, killer moves, history heuristics) cannot be easily matched by MCTS with simple priors.
3. **Time control matters**: At 10+0.1 seconds, MCTS cannot build sufficient tree depth to compete with alpha-beta's efficient pruning.

6.2 Hybrid Search as Prototype Infrastructure

The hybrid architecture is experimental infrastructure, not a competitive engine. The classifier distribution (86.7% Strategic) shows that strategy switching rarely triggers on standard positions. The framework's value is in demonstrating:

- GPU NNUE can serve both alpha-beta and MCTS search
- Position classification is architecturally feasible
- Alpha-beta verification provides measurable benefit (+88 Elo over pure MCTS)

Competitive hybrid search would require trained policy priors and deeper MCTS-AB integration (e.g., using AB bounds to prune MCTS subtrees).

6.3 GPU Acceleration Trade-offs

GPU dispatch overhead (149 μs) makes single-position GPU evaluation unsuitable for pure alpha-beta search. However, MCTS naturally batches leaf evaluations, effectively amortizing this overhead:

- At batch size 1: 267 μs/position (dominated by dispatch)
- At batch size 4096: 0.3 μs/position (compute-dominated)
- Speedup: 635$\times$ through batching

Our MCTS implementation achieves 97K–782K nodes/second depending on position complexity, with endgames benefiting most from high transposition table hit rates.

6.4 Multi-threaded MCTS Scaling

Our implementation uses multi-threaded MCTS with:

- **Virtual loss**: Prevents thread convergence on the same path
- **Lock-free child creation**: Atomic compare-and-swap operations
- **Dedicated evaluation thread**: Batches GPU requests from all workers
- **Arena-based allocation**: Reduces memory contention

Thread scaling is limited (94K→99K NPS from 1→4 threads). While profiling shows selection (78%) dominates over evaluation (12%) in single-threaded time, multi-threading introduces contention at two points: (i) shared tree operations during selection despite lock-free atomics, and (ii) centralized GPU evaluation throughput via the batching queue. The dedicated evaluation thread amortizes GPU dispatch but becomes a synchronization chokepoint as thread count increases.

6.5 Limitations

- **Opening diversity**: All tournament games started from the initial position. A balanced opening book would provide more diverse positions and potentially different results.
- **Heuristic vs trained policy**: We use heuristic-based policy priors. A trained policy network is essential for competitive MCTS strength.
- **Time control**: Short time controls favor alpha-beta; longer time controls may benefit MCTS.
- **Synchronous GPU**: We use blocking GPU dispatch. Asynchronous dispatch infrastructure is implemented but not yet fully utilized.
- **Sample size**: 20 games between MetalFish-AB and Stockfish is statistically limited; more games would narrow confidence intervals.

6.6 Future Work

1. **Policy network training**: Train a policy network on self-play data to dramatically improve MCTS move ordering.
2. **Longer time controls**: Evaluate hybrid search at longer time controls where MCTS can build deeper trees.
3. **Asynchronous evaluation**: Fully utilize the async GPU infrastructure for CPU/GPU overlap.
4. **Deeper AB integration**: Use alpha-beta bounds to prune MCTS subtrees during search, not just as post-verification.

7 Related Work

7.1 Hybrid Search Approaches

AlphaZero [3] demonstrated that MCTS with neural network evaluation can achieve superhuman play. However, AlphaZero uses pure MCTS without alpha-beta verification.

Leela Chess Zero [2] implements AlphaZero's approach as an open-source project, achieving top-tier strength through self-play training and MCTS search.

Stockfish [1] represents the state-of-the-art in alpha-beta engines, using NNUE evaluation with highly optimized search. Our alpha-beta verifier draws inspiration from Stockfish's search techniques.

7.2 GPU Chess Engines

Rocki and Suda [4] explored GPU parallelization of minimax through parallel subtree evaluation. Their work predates modern unified memory architectures.

Our work extends GPU chess engine research to Apple Silicon's unified memory architecture, providing quantified bottleneck analysis and demonstrating that MCTS is better suited for GPU acceleration than alpha-beta due to natural batching.

7.3 Neural Network Evaluation

NNUE (Efficiently Updatable Neural Network) [5] revolutionized chess engine evaluation by providing neural network quality with efficient incremental updates. Our GPU implementation preserves NNUE's architecture while enabling batch evaluation.

Apple's Metal documentation [6, 7] provides guidance on GPU compute optimization, including command buffer management and unified memory usage.

8 Conclusion

We presented MetalFish, a chess engine with GPU-accelerated NNUE evaluation on Apple Silicon, supporting both alpha-beta and experimental MCTS-based search. Our key findings:

1. **GPU NNUE matches CPU NNUE quality**: MetalFish-AB is statistically indistinguishable from Stockfish 16 (2-0-18 in 20 games), validating that GPU NNUE evaluation preserves evaluation quality under identical tournament conditions.
2. **GPU batch efficiency**: 635× speedup through batching (0.3 μs/position at N=4096 vs 267 μs for single positions). This is the core systems contribution.
3. **Dispatch overhead is the bottleneck**: 149 μs irreducible minimum makes GPU unsuitable for single-position alpha-beta but effective for batch-oriented MCTS.
4. **MCTS throughput**: 97K–782K nodes/second depending on position complexity. Endgames achieve 8× higher throughput due to smaller trees and 99% TT hit rates.
5. **Batched evaluation architecture**: 28.5× speedup over direct GPU access through dedicated evaluation thread with request batching.

6. **Hybrid search is experimental**: The ∼2400 Elo gap between alpha-beta and MCTS variants demonstrates that heuristic policy priors are insufficient. The framework is the contribution, not the current strength.

Key insight: GPU NNUE evaluation on Apple Silicon's unified memory is viable for chess engines, but only when batching amortizes dispatch overhead. Alpha-beta search requires low-latency evaluation (favoring CPU), while MCTS naturally batches evaluations (favoring GPU). The experimental hybrid framework demonstrates architectural feasibility; competitive MCTS strength requires trained policy priors.

Future directions: A trained policy network is expected to significantly reduce the MCTS-AB gap, enabling the hybrid architecture to leverage MCTS's exploratory strengths.

Reproducibility

Hardware: Apple M2 Max, 64GB unified memory. **Software**: macOS 14.0, Xcode 15.0, Metal 3.0. **Build**: CMake, -O3, LTO enabled. **Source**: https://github.com/NripeshN/MetalFish. **Branch**: `hybrid-mcts-alphabeta`. **Benchmarks**: `gpubench`, `mctsbench`, `hybridbench` UCI commands. **Tournament**: `tools/elo_tournament.py` with cutechess-cli.

References

1. Stockfish Developers: Stockfish 16 NNUE documentation. https://github.com/official-stockfish/Stockfish (2024)
2. Leela Chess Zero: Neural network based chess engine. https://lczero.org/ (2024)
3. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815 (2017)
4. Rocki, K., Suda, R.: Parallel minimax tree searching on GPU. In: Parallel Processing and Applied Mathematics, LNCS vol. 6067, pp. 449–456. Springer (2010)
5. Nasu, Y.: Efficiently updatable neural-network-based evaluation functions for computer shogi. The 28th World Computer Shogi Championship Appeal Document (2018)
6. Apple Inc.: Metal Programming Guide. https://developer.apple.com/metal/ (2024)
7. Apple Inc.: Metal Best Practices Guide. https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/ (2024)
8. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)