MetalFish: A Hybrid MCTS-Alpha-Beta Chess Engine with GPU-Accelerated NNUE on Apple Silicon

Nripesh Niketan[1]

Independent Researcher
nripesh14@gmail.com

Abstract. We present MetalFish, a novel chess engine that combines Monte Carlo Tree Search (MCTS) with traditional alpha-beta pruning, using GPU-accelerated NNUE evaluation on Apple Silicon's unified memory architecture. Our hybrid approach dynamically classifies positions as tactical or strategic, allocating search resources accordingly: MCTS for exploration in strategic positions, alpha-beta for tactical verification. On Apple M2 Max, our GPU NNUE implementation achieves 572× speedup through batch evaluation (0.3 μs/position at N=4096 vs 258 μs for single positions), enabling MCTS throughput of 190K–280K nodes/second. The position classifier identifies five position types with distinct search strategies, while the alpha-beta verifier provides tactical oversight with configurable override thresholds. We demonstrate that GPU dispatch overhead (140 μs median) makes single-position GPU evaluation unsuitable for pure alpha-beta search, but batch-oriented MCTS effectively amortizes this cost. Our hybrid architecture achieves competitive play while providing a framework for combining the strategic depth of MCTS with the tactical precision of alpha-beta.

Keywords: Chess Engine, Hybrid Search, MCTS, Alpha-Beta, GPU Computing, Metal, NNUE, Apple Silicon

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 hybrid chess engine that combines both search paradigms with GPU-accelerated NNUE evaluation on Apple Silicon. Our key insight is that *position type* should determine search strategy: tactical positions benefit from alpha-beta's precise calculation, while strategic positions benefit from MCTS's exploratory nature.

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 (140 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.

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. **Hybrid search architecture**: A novel combination of MCTS and alpha-beta with dynamic position classification and strategy selection.
2. **GPU NNUE integration**: Efficient batch evaluation achieving 572× speedup over sequential dispatches, with 100% consistency across 1,000 positions.
3. **Position classifier**: Five-category classification (highly tactical to highly strategic) with distinct MCTS/alpha-beta weight allocation.
4. **Alpha-beta verifier**: Full PVS implementation with LMR, futility pruning, and history heuristics for tactical move verification.
5. **Quantified bottleneck analysis**: Stage-by-stage latency decomposition showing GPU dispatch accounts for >98% of single-position time.

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.

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

2.2 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.

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 (140 μs median on M2 Max)

3 System Architecture

MetalFish implements a three-layer architecture: (1) position classification, (2) hybrid search orchestration, and (3) GPU-accelerated evaluation.

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

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:

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

Our MCTS implementation uses PUCT (Predictor + UCT) 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 $m$
$type \leftarrow$ ClassifyPosition($p$)
$strategy \leftarrow$ SelectStrategy($type$)
$T_{mcts} \leftarrow T \times strategy.mcts\_weight$
$T_{ab} \leftarrow T - T_{mcts}$
// Phase 1: MCTS exploration
$m_{mcts} \leftarrow$ RunMCTS($p$, $T_{mcts}$)
if $strategy.ab\_weight > 0.1$ then
    // Phase 2: Alpha-beta verification
    $result \leftarrow$ 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.

Key implementation features:

- **Virtual loss**: Prevents multiple threads from selecting the same path
- **Dirichlet noise**: Added at root for exploration ($\alpha = 0.3$, $\epsilon = 0.25$)
- **Tree reuse**: Previous search tree preserved between moves
- **MCTS transposition table**: Caches evaluations across searches

3.5 Alpha-Beta Verifier

The alpha-beta component provides tactical verification with full search features:

- **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.

We implement adaptive kernel selection:

- **CPU fallback**: Batch size < 4
- **GPU standard**: Batch size < 64

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

- **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

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)

Table 4. Hybrid Search Performance by Position Type

Position Type	MCTS Nodes	NPS	Time (ms)
Highly Strategic	443,511	277K	1,600
Strategic	178,776	244K	731
Balanced	164,756	219K	750
Endgame (KRK)	127,440	283K	450

Table 5. Position Classification Examples

Position	Classification	Strategy
Starting position	Highly Strategic	40% MCTS, 60% AB
Italian Game (move 4)	Strategic	32% MCTS, 67% AB
Central tension	Balanced	25% MCTS, 75% AB
KRK endgame	Balanced	20% MCTS, 80% AB

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 Hybrid Search Performance

Table 4 shows hybrid search performance across position types.

The MCTS component achieves 190K–283K nodes per second, with throughput varying based on position complexity and tree structure.

5.2 Position Classification Results

Table 5 shows position classification for benchmark positions.

5.3 GPU Dispatch Overhead

Table 6 shows minimal-kernel dispatch overhead.

The 140 μs median dispatch overhead represents the irreducible minimum cost for any GPU operation in synchronous blocking mode.

Table 6. GPU Dispatch Overhead—Minimal Kernel (N=1,000)

Statistic	Latency (μs)
Median	140.6
P95	265.0
P99	334.7

Table 7. GPU Stage Breakdown (median, N=100 iterations)

Batch Size	CPU Prep (μs)	GPU Eval (μs)	Total (μs)	GPU %
1	0.2	262.0	262.2	99.9%
8	0.3	271.2	271.5	99.9%
512	6.0	348.5	354.5	98.3%

5.4 GPU Stage Breakdown

Table 7 decomposes end-to-end GPU latency into stages.

GPU dispatch and synchronization dominate (>98% of total time). CPU feature extraction is negligible due to zero-copy buffer management.

5.5 Batch Latency Scaling

Table 8 shows end-to-end latency across batch sizes.

Per-position cost drops from 258 μs (N=1) to 0.3 μs (N=4096), demonstrating effective amortization of dispatch overhead.

5.6 True Batching Verification

Table 9 compares sequential vs batched dispatches.

Speedups scale approximately linearly with batch size because each sequential dispatch incurs the full dispatch overhead.

5.7 GPU Evaluation Consistency

Table 10 verifies GPU evaluation reproducibility.

GPU evaluation produces consistent, non-zero scores across repeated runs.

6 Discussion

6.1 Why Hybrid Search?

The fundamental insight driving our hybrid architecture is that different position types benefit from different search paradigms:

Table 8. GPU End-to-End Batch Latency (N=100 iterations)

Batch	Median	P95	P99	Per-Pos
Size	(μs)	(μs)	(μs)	(μs)
1	258.8	375.3	496.0	258.8
8	261.6	360.0	407.1	32.7
64	280.5	410.5	1669.0	4.4
256	303.0	440.2	1408.0	1.2
512	362.2	495.0	1721.5	0.7
1024	503.6	618.2	689.2	0.5
2048	788.7	911.3	2279.8	0.4
4096	1,360.3	1,464.5	1,536.0	0.3

Table 9. True Batching Verification (N=50 iterations)

N	Sequential	Batched	Speedup
	(N×1 CB)	(1×1 CB)	
16	4,415 μs	259 μs	17.0×
64	18,975 μs	282 μs	67.2×
256	74,989 μs	309 μs	242.7×
1024	306,066 μs	534 μs	572.3×

- **Tactical positions**: Alpha-beta's precise pruning excels at calculating forcing sequences. A hanging piece or checkmate threat requires exact calculation, not statistical estimation.
- **Strategic positions**: MCTS's exploratory nature handles quiet positions well, where many moves are roughly equal and long-term planning matters more than immediate tactics.

Our position classifier bridges these paradigms, dynamically allocating search resources based on position characteristics.

6.2 GPU Acceleration Trade-offs

GPU dispatch overhead (140 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: 258 μs/position (dominated by dispatch)
- At batch size 4096: 0.3 μs/position (compute-dominated)

Our MCTS implementation achieves 190K–280K nodes/second by leveraging this batching, with GPU NNUE providing high-quality evaluations.

Table 10. GPU Evaluation Consistency (1,000 positions)

Metric	Value
Non-zero GPU scores	100%
Consistent across runs	100%
Mean |score|	221.6
Score range	[-407, 258]

6.3 Alpha-Beta Verification Benefits

The alpha-beta verifier provides several benefits:

1. **Tactical oversight**: Catches tactical errors that MCTS might miss due to insufficient exploration.
2. **Move validation**: Verifies MCTS's best move against alpha-beta's recommendation.
3. **Policy improvement**: Alpha-beta search results inform move ordering and policy priors.

In our experiments, the verifier rarely overrides MCTS (0 overrides in benchmark positions), suggesting MCTS produces reasonable moves. However, the verifier provides confidence that tactical blunders are caught.

6.4 Limitations

- **Single-threaded MCTS**: Our current implementation uses single-threaded MCTS due to Position object thread-safety constraints. Multi-threaded MCTS would significantly increase throughput.
- **Policy network**: We use uniform policy priors with Dirichlet noise. A trained policy network would improve MCTS efficiency.
- **Synchronous GPU**: We use blocking GPU dispatch. Asynchronous dispatch with completion handlers could enable CPU/GPU overlap.

6.5 Future Work

1. **Multi-threaded MCTS**: Thread-safe position representation for parallel tree search.
2. **Policy network training**: Train a policy network on MCTS self-play data.
3. **Asynchronous evaluation**: Overlap CPU search with GPU evaluation.
4. **Deeper AB integration**: Use alpha-beta bounds to prune MCTS subtrees.

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 hybrid chess engine combining MCTS with alpha-beta search and GPU-accelerated NNUE evaluation on Apple Silicon. Our key findings:

1. **Hybrid architecture**: Position classification enables dynamic allocation between MCTS exploration (40% for strategic positions) and alpha-beta verification (up to 85% for tactical positions).
2. **MCTS throughput**: 190K–280K nodes/second with GPU NNUE evaluation, leveraging batch processing to amortize dispatch overhead.
3. **GPU batch efficiency**: 572× speedup through batching (0.3 μs/position at N=4096 vs 258 μs for single positions).
4. **Dispatch overhead**: 140 μs irreducible minimum makes GPU unsuitable for pure alpha-beta but effective for batch-oriented MCTS.
5. **Evaluation consistency**: 100% reproducibility across 1,000 positions with meaningful score differentiation.

Key insight: The hybrid MCTS-alpha-beta architecture combines the strategic exploration of MCTS with the tactical precision of alpha-beta, while GPU acceleration provides efficient batch evaluation for MCTS leaf nodes. This architecture demonstrates a practical approach to leveraging GPU compute for chess engines on unified memory systems.

Reproducibility

Hardware: Apple M2 Max, 64GB. **Software**: macOS 14.0, Xcode 15.0. **Build**: CMake, -O3, LTO. **Source**: https://github.com/NripeshN/MetalFish. **Benchmarks**: `gpubench` and `mcts` UCI commands.

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)