MetalFish: A GPU-Accelerated Chess Engine for Apple Silicon Unified Memory Architecture

Nripesh Niketan[1]

Independent Researcher
nripesh14@gmail.com

Abstract. We present MetalFish, a GPU-accelerated UCI chess engine that integrates Stockfish-style alpha-beta search with Apple Metal GPU acceleration on unified memory architecture. MetalFish implements a hybrid CPU-GPU design where the CPU executes traditional search algorithms including Principal Variation Search, null-move pruning, late move reductions, and singular extensions, while the GPU accelerates NNUE neural network evaluation through Metal compute shaders. The implementation leverages Apple Silicon's unified memory to achieve zero-copy data sharing between CPU search routines and GPU evaluation kernels. We describe the complete system architecture, present GPU kernel implementations for feature transformation and incremental accumulator updates, and report empirical results from our test suite. The engine correctly computes perft values for standard test positions, validates move generation through established benchmarks, and demonstrates functional GPU acceleration on Apple M-series processors. This work provides practical insights into the challenges and trade-offs of hybrid CPU-GPU chess engine design.

Keywords: Chess Engine, GPU Computing, Metal, NNUE, Unified Memory, Apple Silicon

1 Introduction

Modern chess engines achieve remarkable playing strength through sophisticated alpha-beta search enhanced with neural network evaluation. Stockfish, the current state-of-the-art engine, combines Principal Variation Search (PVS) with Efficiently Updatable Neural Networks (NNUE) to achieve superhuman performance [1]. However, the emergence of GPU computing presents opportunities to accelerate evaluation-intensive components of chess engines.

The primary challenge in GPU-accelerated chess engines lies in the fundamental mismatch between alpha-beta search's sequential dependencies and GPU architectures' parallel execution model. When an alpha-beta cutoff occurs, remaining sibling nodes can be pruned, creating dependency chains that serialize computation. On GPUs, where threads execute in lockstep within warps, such branch divergence degrades performance significantly [2].

MetalFish addresses this challenge through a hybrid CPU-GPU architecture that leverages Apple Silicon's unified memory. Rather than attempting full GPU parallelization of alpha-beta search, we maintain traditional CPU-based search while offloading neural network evaluation to the GPU. The unified memory architecture eliminates the traditional GPU computing bottleneck of host-device data transfer, enabling efficient hybrid execution without explicit memory copies.

This paper presents the complete MetalFish implementation, describing the search algorithm integration, GPU kernel design, and unified memory utilization. We report empirical results demonstrating correct move generation through perft verification and functional GPU acceleration on Apple M-series hardware.

2 Background and Related Work

2.1 Alpha-Beta Search and Principal Variation Search

The minimax algorithm with alpha-beta pruning forms the foundation of modern chess engines [3]. Alpha-beta maintains bounds $[\alpha, \beta]$ representing the range of possible values for the current node; when $\alpha \geq \beta$, remaining siblings can be pruned without affecting the minimax value.

Principal Variation Search (PVS) refines alpha-beta by assuming the first move searched at each node is optimal [4]. Subsequent moves are searched with zero-width windows; if a move exceeds alpha, a full-window re-search is performed. This approach reduces the number of full evaluations while maintaining correctness.

2.2 NNUE Evaluation

Efficiently Updatable Neural Networks (NNUE) represent a paradigm shift in chess evaluation [5]. The architecture consists of a large sparse input layer representing piece-square combinations, followed by smaller dense layers. The key insight is that network activations can be updated incrementally as moves are made and unmade, rather than recomputing from scratch.

2.3 GPU-Based Chess Engines

Leela Chess Zero (Lc0) pioneered GPU-accelerated neural network evaluation combined with Monte Carlo Tree Search (MCTS) [6]. Unlike alpha-beta, MCTS is naturally suited to GPU parallelization as individual simulations can execute independently. However, alpha-beta's tactical precision through deep selective search motivates hybrid approaches combining traditional search with GPU acceleration.

2.4 Unified Memory Architecture

Apple Silicon's unified memory architecture allows CPU and GPU to access the same physical memory coherently [7]. This eliminates explicit data transfers between separate memory spaces, enabling algorithms that leverage both sequential CPU processing and parallel GPU capabilities without transfer overhead.

3 System Architecture

3.1 Design Overview

MetalFish implements a hybrid CPU-GPU architecture with three primary components:

CPU Search Engine: Executes the complete alpha-beta search algorithm including PVS, move ordering, and all pruning techniques. Search control flow remains entirely on CPU to avoid branch divergence penalties.

GPU Evaluation Engine: Implements NNUE inference through Metal compute shaders, processing feature transformation and network forward passes on GPU.

Unified Memory Interface: Manages shared buffers accessible by both CPU and GPU without explicit copying, using Metal's shared storage mode.

3.2 Search Algorithm Implementation

The search implementation follows Stockfish's architecture, incorporating the following techniques:

Move Ordering Effective move ordering is critical for alpha-beta efficiency. MetalFish implements multiple history heuristics:

Butterfly History: Tracks quiet move success by source and destination squares, stored in a [2][4096] array indexed by [color][from$\times$64+to]. Values are initialized to 68 following Stockfish conventions.

Capture History: Tracks capture move success by piece, destination square, and captured piece type, stored in a [16][64][8] array. Values are initialized to -689.

Continuation History: Tracks move sequence success using the previous moves in the search stack. The implementation uses a [16][64][16][64] array with values initialized to -529.

Killer Moves: Stores refutation moves per ply that caused beta cutoffs in sibling nodes.

Counter Moves: Stores moves that refute specific previous moves.

Search Extensions Singular Extension: When a transposition table move appears significantly better than alternatives, search depth is extended. The implementation uses double and triple extensions for strongly singular moves.

Check Extension: Positions where the side to move is in check receive depth extensions to ensure tactical threats are fully resolved.

Pruning Techniques Null Move Pruning: If passing a turn (null move) still produces a beta cutoff at reduced depth, the position is pruned. Verification search is applied at high depths.

Late Move Reductions (LMR): Moves searched later in the move list are searched at reduced depth. The reduction formula follows Stockfish:

$$R = \left\lfloor \frac{2747}{128} \times \ln(\text{moveNumber}) \right\rfloor \tag{1}$$

with adjustments based on history scores, node type, and whether the position is in check.

Futility Pruning: Near-leaf nodes with static evaluations far below alpha are pruned when no single move is likely to raise the score sufficiently.

SEE Pruning: Moves with negative Static Exchange Evaluation are pruned or reduced based on depth and move type.

ProbCut: Positions are pruned when a shallow search of captures suggests the score will exceed beta by a significant margin.

Transposition Table The transposition table stores previously searched positions using Zobrist hashing [8]. Each entry contains the position hash, evaluation bound type (exact, lower, or upper), best move, search depth, and generation counter for aging. The replacement scheme considers entry depth and age to balance between preserving deep searches and allowing new entries.

3.3 GPU NNUE Implementation

The GPU evaluation engine implements NNUE inference through Metal compute shaders. The network architecture processes 1024-dimensional accumulators through fully-connected layers.

Feature Extraction Position features are extracted using the HalfKAv2 scheme, where features are indexed by king position and piece-square combinations. For each piece on the board, the feature index is computed as:

$$\text{feature} = \text{king\_sq} \times 641 + \text{piece\_sq} \times 10 + \text{piece\_type} \tag{2}$$

where piece positions are mirrored based on the perspective (white or black).

Feature Transformation Kernel The feature transformer converts sparse input features to dense accumulator values. Each GPU thread computes one element of the output accumulator, iterating over all active features and accumulating the corresponding weight values.

```
kernel void feature_transform(
    device const int16_t* weights,
    device const int16_t* biases,
    device const int* features,
    device int32_t* accumulator,
    constant int& num_features,
    constant int& ft_dims,
    uint h [[thread_position_in_grid]])
{
    if (h >= ft_dims) return;

    int32_t sum = biases[h];
    for (int i = 0; i < num_features; i++) {
        int f = features[i];
        sum += weights[f * ft_dims + h];
    }
    accumulator[h] = sum;
}
```

Listing 1.1. Feature transformation kernel computing accumulator values from sparse input features

Incremental Update Kernel When a move is made, only the changed features need updating rather than recomputing the entire accumulator. The kernel processes added and removed features in parallel.

```
kernel void feature_update(
    device const int16_t* weights,
    device int32_t* accumulator,
    device const int* added_features,
    device const int* removed_features,
    constant int& num_added,
    constant int& num_removed,
    constant int& ft_dims,
    uint h [[thread_position_in_grid]])
{
    if (h >= ft_dims) return;

    int32_t delta = 0;
    for (int i = 0; i < num_added; i++) {
        int f = added_features[i];
        delta += weights[f * ft_dims + h];
    }
    for (int i = 0; i < num_removed; i++) {
        int f = removed_features[i];
        delta -= weights[f * ft_dims + h];
    }
    accumulator[h] += delta;
}
```

Listing 1.2. Incremental accumulator update kernel processing added and removed features

Network Forward Pass The NNUE forward pass processes the accumulated features through fully-connected layers with clipped ReLU activations:

$$\text{clipped\_relu}(x) = \min(\max(x \gg 6, 0), 127) \tag{3}$$

The network architecture consists of:

- FC0: 2048 $\rightarrow$ 16 (concatenated white/black accumulators)
- FC1: 15 $\rightarrow$ 32 with squared clipped ReLU
- FC2: 32 $\rightarrow$ 1 producing the final evaluation score

3.4 Metal Backend Implementation

The Metal backend manages GPU resources and kernel execution. The device initialization establishes the GPU context and command queue.

```
Device::Device() {
    device_ = MTL::CreateSystemDefaultDevice();
    if (!device_) {
        throw std::runtime_error(
            "Failed to create Metal device");
    }
    queue_ = device_->newCommandQueue();
    architecture_ = device_->name()->utf8String();
}
```

Listing 1.3. Metal device initialization establishing GPU context and command queue

Buffer allocation uses shared storage mode to enable unified memory access, allowing both CPU code (via `buffer->contents()`) and GPU kernels to access the same memory without explicit transfers.

```
bool GPUNNUEWeights::allocate(MTL::Device* device) {
    MTL::ResourceOptions options =
        MTL::ResourceStorageModeShared;

    ft_weights = device->newBuffer(
        FT_IN_DIMS * FT_OUT_DIMS * sizeof(int16_t),
        options);
    ft_biases = device->newBuffer(
        FT_OUT_DIMS * sizeof(int16_t),
        options);
    return ft_weights && ft_biases;
}
```

Listing 1.4. Buffer allocation with shared storage mode for unified memory access

4 Experimental Results

4.1 Move Generation Verification

Move generation correctness is verified through perft (performance test) calculations, which count the number of leaf nodes at a given depth. Table 1 shows results for the standard starting position.

Table 1. Perft results for the standard chess starting position. All values match established correct results, validating move generation and position update logic.

Depth	Nodes
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324

Additional perft tests verify correctness for complex positions including the Kiwipete position (depth 5: 193,690,690 nodes), positions with en passant, castling rights, and promotion scenarios.

4.2 GPU Backend Validation

Table 2 presents hardware characteristics reported by the Metal backend during initialization on an Apple M2 Max system.

Table 2. GPU backend characteristics reported during MetalFish initialization on Apple M2 Max hardware.

Property	Value
Device Name	Apple M2 Max
Unified Memory	Enabled
Maximum Buffer Size	19,169 MB
Maximum Threadgroup Memory	32,768 bytes
Maximum Threads per Threadgroup	1,024

4.3 Test Suite Results

MetalFish includes unit tests covering core functionality. Table 3 summarizes the test categories and results.

Table 3. Test suite results validating MetalFish components. All tests pass, confirming correct implementation of core functionality.

Test Category	Result
Bitboard Operations	Pass
Position Representation	Pass
Move Generation	Pass
Search Components (History, SEE, TT)	Pass
Metal GPU Backend	Pass
GPU NNUE Kernels	Pass
UCI Protocol Handling	Pass
Perft Verification	Pass

4.4 GPU Memory Utilization

Table 4 shows GPU memory allocation for NNUE evaluation, as reported by the engine during initialization.

Table 4. GPU memory allocation for NNUE evaluation components.

Component	Allocation
Feature Transformer Weights	45,056 KB
Feature Transformer Biases	2 KB
PSQT Weights	704 KB
FC Layer Weights	137 KB
Working Buffers	2,296 KB
Total GPU Memory	48,195 KB

5 Discussion

5.1 Hybrid Execution Trade-offs

Our implementation reveals important trade-offs in hybrid CPU-GPU chess engine design:

Kernel Dispatch Overhead: GPU kernel dispatch and synchronization introduce latency that can exceed computation time for single-position evaluation. The Metal command buffer creation, encoding, commit, and wait cycle imposes overhead that favors CPU execution for individual positions.

Batch Amortization: GPU acceleration provides benefit when evaluating multiple positions simultaneously, amortizing dispatch overhead across the batch. Our observations suggest batch sizes of 8–16 positions or more are needed for GPU execution to outperform CPU.

Unified Memory Advantage: The unified memory architecture eliminates explicit data copying between CPU and GPU address spaces. This enables the hybrid approach where CPU search code and GPU evaluation share data structures directly through pointer access.

5.2 Search Algorithm Considerations

The sequential nature of alpha-beta search with pruning fundamentally limits GPU parallelization. Cutoff decisions depend on results from previously searched moves, creating data dependencies that serialize execution. Our hybrid approach accepts this limitation by keeping search on CPU while accelerating the evaluation function.

The history heuristics (butterfly, capture, continuation) require frequent small updates during search that would incur significant GPU synchronization overhead. Maintaining these tables on CPU and accessing them directly avoids this overhead.

5.3 Limitations

Several limitations affect the current implementation:

- GPU acceleration is limited to evaluation; the search algorithm executes entirely on CPU
- Single-position evaluation is faster on CPU due to kernel dispatch overhead
- The implementation uses pre-trained NNUE networks; training is not included
- Syzygy tablebase probing interface is implemented but file loading is not complete

6 Related Work

Fat Fritz combined Stockfish's alpha-beta search with GPU-accelerated neural network evaluation, demonstrating the viability of hybrid approaches. Our work extends this direction to unified memory architectures where CPU-GPU data sharing is more efficient.

The MLX framework from Apple provides high-level abstractions for GPU computing on Apple Silicon. MetalFish uses lower-level Metal APIs for finer control over kernel execution and memory management.

7 Conclusion

MetalFish demonstrates that hybrid CPU-GPU chess engines are practical on unified memory architectures. By maintaining alpha-beta search on CPU while accelerating NNUE evaluation on GPU, we preserve the tactical precision of traditional search while leveraging GPU parallelism for neural network inference.

The unified memory architecture of Apple Silicon proves valuable for this hybrid approach, enabling zero-copy data sharing between CPU search routines and GPU evaluation kernels. This eliminates the transfer overhead that would otherwise penalize fine-grained CPU-GPU cooperation.

Key findings from this implementation:

1. Alpha-beta search's sequential dependencies make full GPU parallelization impractical; hybrid approaches are more effective
2. Unified memory enables efficient CPU-GPU cooperation without explicit data transfers
3. GPU kernel dispatch overhead favors CPU execution for single-position evaluation
4. Batch processing amortizes GPU overhead, providing benefit for multi-position evaluation

The MetalFish implementation provides a foundation for future research in GPU-accelerated game tree search on unified memory systems.

Acknowledgments

The author thanks the Stockfish development team for their open-source contributions that informed this implementation, and the Leela Chess Zero team for advancing neural network chess evaluation.

References

1. Romstad, T., Costalba, M., Kiiski, J.: Stockfish: A strong open source chess engine. https://stockfishchess.org/ (2008)
2. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue, 6(2), 40–53 (2008)
3. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326 (1975)
4. Reinefeld, A.: An improvement to the scout tree-search algorithm. ICCA Journal, 6(4), 4–14 (1983)
5. Nasu, Y.: NNUE: Efficiently updatable neural networks for board game position evaluation. Master's Thesis, University of Electro-Communications (2018)
6. Leela Chess Zero Team: Leela chess zero: Neural network chess engine. https://lczero.org/ (2024)
7. Apple Inc.: Metal programming guide. Technical report, Apple Inc. (2023). https://developer.apple.com/metal/
8. Zobrist, A.L.: A new hashing method with application for game playing. ICCA Journal, 13(2), 69–73 (1970)
9. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep blue. Artificial Intelligence, 134(1-2), 57–83 (2002)
10. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
11. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97–109 (1985)
12. Beal, D.F.: Experiments with the null move. Advances in Computer Chess, 5, 65–79 (1989)
13. Heinz, E.A.: Adaptive null-move pruning. ICGA Journal, 23(3), 123–132 (2000)