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 fully functional GPU-accelerated UCI chess engine implementing Stockfish-style alpha-beta search with Apple Metal GPU acceleration on unified memory architecture. Unlike theoretical approaches, MetalFish represents a complete, tested implementation that successfully integrates over 60 search features from modern chess engines with GPU-accelerated NNUE evaluation. Our implementation demonstrates that unified memory architectures enable practical hybrid CPU-GPU chess engines by eliminating data transfer overhead. MetalFish achieves zero-copy data sharing between CPU search and GPU evaluation through Apple Silicon's unified memory, with GPU kernels for NNUE feature transformation, incremental accumulator updates, and neural network forward passes. The engine passes 102 comprehensive tests including perft verification, UCI protocol compliance, and GPU functionality validation. We present the architectural decisions, implementation challenges, and empirical observations from building a production-ready GPU chess engine, providing insights for future GPU-accelerated game tree search implementations.

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

1 Introduction

Modern chess engines like Stockfish achieve remarkable playing strength through sophisticated alpha-beta search enhanced with neural network evaluation functions. However, the computational landscape has transformed with Graphics Processing Units (GPUs) emerging as powerful parallel computing platforms. This paper presents MetalFish, a complete implementation of a GPU-accelerated chess engine for Apple Silicon, demonstrating practical integration of traditional search algorithms with GPU acceleration.

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.

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 evaluation-intensive operations to the GPU. The unified memory architecture eliminates the traditional GPU computing bottleneck of host-device data transfer, enabling efficient hybrid execution.

Our contributions include: (1) a complete, tested implementation of a GPU-accelerated chess engine with 60+ search features, (2) Metal GPU kernels for NNUE evaluation including feature transformation and incremental updates, (3) empirical analysis of hybrid CPU-GPU execution trade-offs, and (4) a comprehensive test suite validating correctness across 102 tests.

2 Background and Related Work

2.1 Modern Chess Engine Architecture

Contemporary chess engines implement Principal Variation Search (PVS), a refinement of alpha-beta that reduces full-window searches by initially searching sibling nodes with null windows [1]. Stockfish exemplifies this approach through extensive pruning techniques including null-move pruning, late move reductions, and futility pruning [2].

Since 2020, Stockfish has incorporated Efficiently Updatable Neural Networks (NNUE) for position evaluation [3]. NNUE networks are specifically designed for incremental evaluation during tree search, with the key insight that only a small fraction of input neurons are active for any position, enabling efficient updates as moves are made and unmade.

2.2 GPU-Based Chess Engines

Leela Chess Zero (Lc0) pioneered GPU-accelerated neural network evaluation combined with Monte Carlo Tree Search (MCTS) [4]. This approach achieves world-class performance while being naturally suited to GPU parallelization, as individual MCTS simulations can execute independently.

However, MCTS-based engines differ fundamentally from alpha-beta engines in their search characteristics. Alpha-beta's tactical precision through deep selective search remains valuable, motivating hybrid approaches that combine traditional search with GPU acceleration.

2.3 Unified Memory Architecture

Apple Silicon's unified memory architecture allows CPU and GPU to access the same physical memory pool coherently [5]. This eliminates explicit data transfers between separate memory spaces, potentially enabling algorithms that leverage both sequential CPU processing and parallel GPU capabilities without transfer overhead.

3 MetalFish Architecture

3.1 Design Philosophy

MetalFish embraces a fundamental principle: rather than forcing traditional algorithms onto GPU hardware, we design a hybrid system leveraging the unique strengths of both architectures. The CPU excels at sequential decision-making and complex control flow, while the GPU provides massive parallel throughput for regular computational tasks.

The architecture consists of three primary components:

CPU Search Engine: Implements full Stockfish-style alpha-beta search with PVS, maintaining the principal variation and coordinating search across depths. All search control flow, move ordering decisions, and pruning logic execute on CPU.

GPU Evaluation Engine: Handles neural network evaluation using Metal compute shaders. Processes position features through the NNUE architecture with GPU-accelerated matrix operations.

Unified Memory Manager: Leverages unified memory for zero-copy data sharing between CPU and GPU, eliminating traditional host-device transfer bottlenecks.

3.2 Search Implementation

MetalFish implements over 60 search features from modern chess engines:

Move Ordering

- **ButterflyHistory:** Quiet move success tracking by from/to squares
- **KillerMoves:** Refutation moves per ply
- **CounterMoveHistory:** Moves that refute the previous move
- **CapturePieceToHistory:** Capture move success tracking
- **PawnHistory:** Pawn structure-aware history indexed by pawn key
- **ContinuationHistory:** Move sequence success tracking
- **Staged Move Generation:** Efficient MovePicker with capture/quiet phases

Search Extensions and Pruning

- **Singular Extension:** Extend clearly best moves with double/triple extension
- **Null Move Pruning:** With verification search at high depths
- **Late Move Reductions:** Multiple adjustment factors including cutoffCnt
- **Futility Pruning:** For quiet moves and captures
- **SEE-based Pruning:** Static Exchange Evaluation
- **ProbCut:** Prune with shallow capture search
- **Razoring:** Drop to quiescence search for low evaluation positions

Infrastructure

- **Transposition Table:** With aging, generation tracking, and rule50 handling
- **Aspiration Windows:** With dynamic delta sizing
- **Lazy SMP:** Multi-threaded parallel search
- **Correction History:** Pawn, minor piece, and continuation corrections

The Late Move Reduction formula follows Stockfish:

$$R_{LMR} = \frac{2747}{128} \times \ln(\text{moveNumber}) \tag{1}$$

3.3 GPU NNUE Implementation

The GPU evaluation engine implements NNUE inference through Metal compute shaders. The architecture handles:

Feature Transformation The feature transformer converts sparse piece-square inputs to dense hidden layer activations:

Algorithm 1 GPU Feature Transform Kernel

```
procedure FeatureTransform(weights, biases, features, acc)
    h ← thread_position_in_grid
    if h ≥ ft_dims then return
    end if
    sum ← biases[h]
    for i ← 0 to num_features do
        f ← features[i]
        sum ← sum + weights[f × ft_dims + h]
    end for
    acc[h] ← sum
end procedure
```

Incremental Updates A key optimization is incremental accumulator updates. When a move is made, only the changed features need updating:

$$acc_{new} = acc_{old} - w_{removed} + w_{added} \tag{2}$$

The GPU kernel processes added and removed features in parallel:

Algorithm 2 GPU Incremental Update Kernel

procedure FEATUREUPDATE(weights, acc, added, removed)
  $h \leftarrow$ thread_position_in_grid
  $delta \leftarrow 0$
  for each feature f in added **do**
    $delta \leftarrow delta + weights[f \times \text{ft\_dims} + h]$
  end for
  for each feature f in removed **do**
    $delta \leftarrow delta - weights[f \times \text{ft\_dims} + h]$
  end for
  $acc[h] \leftarrow acc[h] + delta$
end procedure

Forward Pass The NNUE forward pass implements the network layers:

- Input: 1024-dimensional accumulator (concatenated perspectives)
- FC0: $2048 \rightarrow 16$ with clipped ReLU
- FC1: $15 \rightarrow 32$ with squared clipped ReLU
- FC2: $32 \rightarrow 1$ output score

The activation functions are implemented as:

$$\text{clipped\_relu}(x) = \text{clamp}\left(\frac{x}{2^6}, 0, 127\right) \tag{3}$$

$$\text{sqr\_clipped\_relu}(x) = \frac{v^2}{128}, \quad v = \text{clamp}\left(\frac{x}{2^6}, 0, 127\right) \tag{4}$$

3.4 Metal Backend Implementation

The Metal backend provides GPU device management, buffer allocation, and kernel execution:

```
Device::Device() {
  device_ = MTL::CreateSystemDefaultDevice();
  queue_ = device_->newCommandQueue();
  architecture_ = device_->name()->utf8String();
  // Unified memory enabled by default
}
```

Listing 1.1. Metal Device Initialization

Buffer allocation uses shared storage mode for unified memory access:

```
MTL::ResourceOptions options =
    MTL::ResourceStorageModeShared;
buffer = device->newBuffer(size, options);
// Both CPU and GPU can access buffer->contents()
```

Listing 1.2. Unified Memory Buffer Allocation

3.5 Hybrid Execution Strategy

MetalFish dynamically selects between CPU and GPU execution based on workload characteristics. For single-position evaluation during search, CPU execution often outperforms GPU due to kernel dispatch overhead. GPU acceleration provides benefit for:

- Batch evaluation of multiple positions
- Full accumulator recomputation
- Move scoring across large move lists
- SEE calculations for move ordering

The decision logic considers batch size and current GPU utilization:

$$\text{use\_gpu} = (\text{batch\_size} > \text{threshold}) \wedge (\text{gpu\_available}) \tag{5}$$

where threshold is empirically determined (typically 8-16 positions).

4 GPU Operations

4.1 Batch Move Generation

The GPU assists with move generation through parallel piece processing:

- Pawn move generation: 8 threads per position (one per potential pawn)
- Knight move generation: 2 threads per position
- King move generation: 1 thread per position

Attack tables (pawn attacks, knight attacks, king attacks) are uploaded to GPU memory once during initialization.

4.2 Static Exchange Evaluation

SEE calculations for move ordering are parallelized across positions:

Algorithm 3 Batch SEE Kernel

```
procedure BatchSEE(positions, moves, results)
    idx ← thread_position_in_grid
    pos ← positions[idx]
    move ← moves[idx]
    results[idx] ← compute_see(pos, move)
end procedure
```

4.3 Zobrist Hashing

Position hashing is accelerated for batch operations:

```
kernel void compute_zobrist_hash(
    device const GPUPosition* positions,
    device const uint64_t* piece_keys,
    device uint64_t* hashes,
    uint gid [[thread_position_in_grid]])
{
    uint64_t hash = 0;
    for (int sq = 0; sq < 64; sq++) {
        int piece = positions[gid].board[sq];
        if (piece != NO_PIECE)
            hash ^= piece_keys[piece * 64 + sq];
    }
    // Add castling, en passant, side to move
    hashes[gid] = hash;
}
```

Listing 1.3. GPU Zobrist Hash Computation

5 Evaluation and Results

5.1 Test Suite

MetalFish includes a comprehensive test suite validating correctness:

Table 1. Test Suite Coverage

Category	Tests	Status
Bitboard Operations	8	Pass
Position Handling	12	Pass
Move Generation	15	Pass
Search Components	10	Pass
Metal GPU Backend	8	Pass
GPU NNUE	10	Pass
UCI Protocol	9	Pass
Perft Verification	30	Pass
Total	**102**	**All Pass**

5.2 Perft Verification

Move generation correctness is verified through perft (performance test) calculations:

Table 2. Perft Results (Starting Position)

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

All perft results match the established correct values, validating move generation and position update logic.

5.3 GPU Performance Characteristics

Testing on Apple M2 Max demonstrates unified memory benefits:

Table 3. GPU Backend Characteristics (M2 Max)

Metric	Value
Device	Apple M2 Max
Unified Memory	Yes
Max Buffer Size	19,169 MB
Max Threadgroup Memory	32,768 bytes
Max Threads/Threadgroup	1,024

5.4 Hybrid Execution Trade-offs

Empirical observation reveals important trade-offs:

GPU Overhead: For single-position evaluation, kernel dispatch and synchronization overhead (approximately 10-50μs) can exceed the computation time, making CPU evaluation faster for individual positions.

Batch Threshold: GPU acceleration provides net benefit when batch size exceeds 8-16 positions, where parallel execution amortizes dispatch overhead.

Unified Memory Benefit: Zero-copy data sharing eliminates the 1-10μs PCIe transfer latency per operation that would exist on discrete GPU systems, enabling finer-grained CPU-GPU cooperation.

5.5 Memory Utilization

GPU memory allocation for NNUE evaluation:

Table 4. GPU Memory Allocation

Component	Size
Feature Transformer Weights	45 MB
FC Layer Weights	2 MB
Accumulator Buffers	8 KB
Working Buffers	2.3 MB
Total	**∼50 MB**

6 Implementation Challenges

6.1 Branch Divergence

Alpha-beta search's conditional pruning creates severe branch divergence on GPU. Our solution maintains all search logic on CPU, using GPU only for evaluation where computation is more regular.

6.2 Incremental Update Complexity

NNUE's incremental update mechanism, while efficient on CPU, requires careful GPU implementation. We maintain both full recomputation and incremental update kernels, selecting based on the number of changed features.

6.3 Synchronization Overhead

GPU kernel dispatch and synchronization introduce latency that dominates small workloads. The hybrid approach mitigates this by batching GPU operations and using CPU for latency-sensitive single-position evaluation.

6.4 Memory Coherency

While unified memory eliminates explicit copies, cache coherency still requires attention. We structure data access patterns to minimize coherency traffic between CPU and GPU caches.

7 Limitations and Future Work

7.1 Current Limitations

- GPU acceleration limited to evaluation; search remains CPU-bound
- Single-position evaluation faster on CPU due to dispatch overhead
- NNUE network training not included (uses pre-trained Stockfish networks)
- Syzygy tablebase interface implemented but file loading pending

7.2 Future Directions

Parallel Best-First Search: Investigating GPU-friendly search algorithms that maintain evaluation quality while enabling parallelism.

Batch Search: Exploring simultaneous search of multiple positions to amortize GPU overhead.

Network Architecture: Designing NNUE architectures optimized for GPU execution patterns.

Cross-Platform: Extending to CUDA for NVIDIA GPUs while maintaining the unified memory approach on supported platforms.

8 Conclusion

MetalFish demonstrates that practical GPU-accelerated chess engines are achievable through careful hybrid architecture design. By maintaining traditional alpha-beta search on CPU while offloading evaluation to GPU, we preserve tactical precision while leveraging GPU parallelism where beneficial.

The unified memory architecture of Apple Silicon proves particularly valuable, enabling zero-copy data sharing that makes fine-grained CPU-GPU cooperation practical. Our implementation achieves full correctness across 102 tests while providing GPU-accelerated NNUE evaluation.

Key insights from this implementation:

1. Hybrid CPU-GPU approaches outperform pure GPU implementations for alpha-beta search due to sequential dependencies
2. Unified memory eliminates transfer overhead, enabling practical hybrid execution
3. GPU acceleration provides benefit for batch operations but not single-position evaluation
4. Comprehensive testing is essential for validating complex hybrid systems

The MetalFish codebase provides a foundation for future research in GPU-accelerated game tree search, demonstrating both the possibilities and practical limitations of current approaches.

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. Special recognition to the Metal and MLX teams at Apple for GPU programming resources.

References

1. Reinefeld, A.: An improvement to the scout tree-search algorithm. ICCA Journal, 6(4), 4–14 (1983)
2. Romstad, T., Costalba, M., Kiiski, J.: Stockfish: A strong open source chess engine. https://stockfishchess.org/ (2008)
3. Nasu, Y.: NNUE: Efficiently updatable neural networks for board game position evaluation. Master's Thesis, University of Electro-Communications (2018)
4. Leela Chess Zero Team: Leela chess zero: Neural network chess engine. https://lczero.org/ (2024)
5. Apple Inc.: Metal performance shaders optimization guide. Technical report, Apple Inc. (2023). https://developer.apple.com/documentation/metalperformanceshaders
6. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence, 6(4), 293–326 (1975)
7. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
8. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep blue. Artificial Intelligence, 134(1-2), 57–83 (2002)
9. NVIDIA Corporation: CUDA C++ programming guide. Technical report, NVIDIA Corporation (2023)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue, 6(2), 40–53 (2008)
11. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97–109 (1985)
12. Zobrist, A.L.: A new hashing method with application for game playing. ICCA Journal, 13(2), 69–73 (1970)
13. Beal, D.F.: Experiments with the null move. Advances in Computer Chess, 5, 65–79 (1989)
14. Heinz, E.A.: Adaptive null-move pruning. ICGA Journal, 23(3), 123–132 (2000)
15. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: Computers and Games, LNCS vol. 4630, pp. 72–83. Springer (2006)