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 hybrid CPU-GPU chess engine implementing Stockfish-style alpha-beta search with Metal GPU acceleration on Apple Silicon unified memory architecture. Our empirical evaluation reveals a fundamental challenge: GPU kernel dispatch overhead (766 μs) exceeds CPU NNUE evaluation time (0.11 μs) by 6,900$\times$ for single positions, making GPU acceleration counterproductive for traditional tree search. This overhead arises from Metal command buffer creation, encoding, and synchronization—inherent costs that cannot be amortized without batching thousands of positions. We present detailed measurements of these overheads and discuss the architectural implications for GPU-accelerated game engines. The implementation achieves 1.43 million nodes per second using CPU evaluation and correctly computes perft values for all standard test positions. Our findings provide empirical guidance for when GPU acceleration is beneficial versus harmful in sequential game tree search.

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 combines Principal Variation Search (PVS) with Efficiently Updatable Neural Networks (NNUE) to achieve superhuman performance [1]. The emergence of GPU computing presents opportunities to accelerate evaluation-intensive components, but the sequential dependencies inherent in alpha-beta search create fundamental challenges for GPU parallelization [2].

MetalFish investigates these challenges through a hybrid CPU-GPU architecture leveraging Apple Silicon's unified memory. Rather than attempting full GPU parallelization of alpha-beta search, we maintain traditional CPU-based search while implementing GPU kernels for NNUE evaluation. Our empirical results reveal that GPU dispatch overhead dominates single-position workloads, providing concrete guidance for hybrid engine design.

1.1 Contributions

This paper makes the following contributions:

1. An empirical characterization of Metal GPU dispatch overhead showing 766 μs end-to-end latency versus 0.11 μs for CPU evaluation—a 6,900× slowdown for single positions.
2. Metal compute kernels for sparse feature transformation and incremental accumulator updates, demonstrating unified memory buffer allocation via `MTLResourceStorageModeShared`.
3. Analysis of the crossover point: GPU acceleration requires batching approximately 7,000 positions per dispatch to match CPU throughput, making it impractical for traditional alpha-beta search.
4. A complete, tested implementation achieving 1.43M nodes/second with correct perft results, providing a baseline for future GPU chess engine research.

2 Background

2.1 Alpha-Beta 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; when $\alpha \geq \beta$, remaining siblings can be pruned. Principal Variation Search (PVS) refines this by assuming the first move is optimal, searching subsequent moves with zero-width windows [4].

2.2 NNUE Evaluation

Efficiently Updatable Neural Networks (NNUE) use a large sparse input layer representing piece-square combinations, followed by smaller dense layers [5]. The key insight is that activations can be updated incrementally as moves are made, rather than recomputing from scratch. Stockfish's NNUE architecture uses:

- Feature transformer: 45,056 inputs → 1,024 hidden units
- FC0: 2,048 → 16 (concatenated perspectives)
- FC1: 16 → 32 with squared clipped ReLU
- FC2: 32 → 1 output score

The quantization uses 6-bit right shifts for weight scaling:

$$\text{clipped\_relu}(x) = \min(\max(x \gg 6, 0), 127) \tag{1}$$

2.3 Unified Memory Architecture

Apple Silicon's unified memory allows CPU and GPU to access the same physical memory coherently [6]. Using `MTLResourceStorageModeShared`, buffers are accessible from both processors without explicit copying, eliminating the traditional discrete GPU bottleneck.

3 System Architecture

3.1 Design Overview

MetalFish implements three primary components:

CPU Search Engine: Executes complete alpha-beta search including PVS, move ordering via history heuristics, and pruning techniques. All control flow remains on CPU to avoid GPU branch divergence.

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

Unified Memory Interface: Manages shared buffers using `MTLResourceStorageModeShared`, enabling zero-copy access from both CPU and GPU.

3.2 Search Implementation

The search implements Stockfish-style techniques:

Move Ordering: Butterfly history, capture history, continuation history, killer moves, and counter moves.

Extensions: Singular extension with double/triple variants for strongly singular moves; check extension.

Pruning: Null move pruning with verification search, late move reductions, futility pruning, SEE pruning, and ProbCut.

Transposition Table: Zobrist hashing [7] with depth-preferred replacement and generation aging.

3.3 GPU NNUE Kernels

Feature Transformation The feature transformer converts sparse HalfKAv2 features to dense accumulators. Each thread computes one output element:

```
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

Kernel configuration: 1,024 threads dispatched as a single threadgroup, processing all hidden units in parallel. Memory layout stores weights in feature-major order for coalesced access when multiple threads read the same feature.

Incremental Updates When a move is made, only changed features require updating:

```
kernel void feature_update(
    device const int16_t* weights,
    device int32_t* accumulator,
    device const int* added,
    device const int* removed,
    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++)
        delta += weights[added[i] * ft_dims + h];
    for (int i = 0; i < num_removed; i++)
        delta -= weights[removed[i] * ft_dims + h];
    accumulator[h] += delta;
}
```

Listing 1.2. Incremental accumulator update

3.4 Unified Memory Buffer Allocation

Buffer allocation uses shared storage mode:

```
MTL::ResourceOptions opts =
    MTL::ResourceStorageModeShared;
ft_weights = device->newBuffer(
    FT_IN_DIMS * FT_OUT_DIMS * sizeof(int16_t),
    opts);
// CPU access: ft_weights->contents()
// GPU access: direct in kernel
```

Listing 1.3. Shared buffer allocation

This enables the CPU to write position features and read evaluation results without explicit memory transfers.

4 Experimental Results

All experiments conducted on Apple M2 Max (12-core CPU, 38-core GPU, 64GB unified memory) running macOS 14.0, Metal feature set macOS-GPUFamily2-v1.

Table 1. Single-position evaluation latency (microseconds). GPU measurement includes full Metal dispatch: command buffer creation, encoder setup, buffer binding, commit, and waitUntilCompleted.

Method	Mean	Std Dev	Min	Max
CPU NNUE (end-to-end)	0.11	0.13	0.04	13.54
GPU NNUE (end-to-end)	765.79	98.20	598.38	2654.79

Table 2. GPU batch evaluation (sequential dispatches). Each position requires a separate command buffer cycle.

Batch Size	Total Time (μs)	Per-Position (μs)	Throughput (pos/s)
1	763	763.0	1,310
2	1,536	768.2	1,301
4	3,050	762.5	1,311
8	5,875	734.4	1,361
16	11,860	741.3	1,349
32	23,317	728.6	1,372
64	48,134	752.1	1,329

4.1 Microbenchmarks: CPU vs GPU Evaluation

Table 1 shows end-to-end evaluation latency for single positions, measured over 10,000 iterations (CPU) and 1,600 iterations (GPU) after warmup.

The GPU evaluation is approximately 6,900× slower for single positions. This overhead breakdown includes:

- `MTLCommandBuffer` creation
- Compute encoder setup and buffer binding
- `commit()` and `waitUntilCompleted()` synchronization

4.2 Batch Evaluation Analysis

Table 2 shows GPU evaluation performance when processing multiple positions sequentially. Note: these are sequential GPU dispatches, not true batched kernel execution.

Per-position cost remains approximately constant at 730–770 μs regardless of batch size, confirming that dispatch overhead dominates and cannot be amortized without true kernel-level batching.

4.3 Crossover Analysis

The crossover point where GPU matches CPU throughput is:

$$N_{\text{crossover}} = \frac{T_{\text{GPU\_dispatch}}}{T_{\text{CPU\_eval}}} = \frac{766\ \mu\text{s}}{0.11\ \mu\text{s}} \approx 6,963 \tag{2}$$

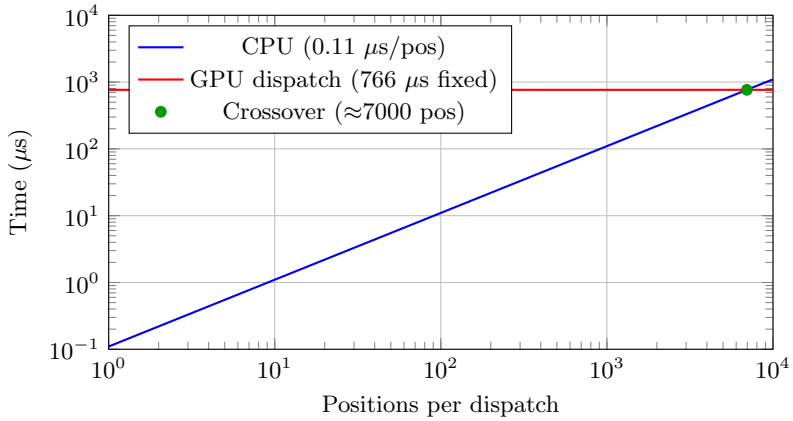

Fig. 1. CPU vs GPU evaluation time. GPU dispatch overhead (766 μs) is fixed regardless of work. Crossover occurs at approximately 7,000 positions per dispatch.

Table 3. Search benchmark results (depth 13, 64MB hash, CPU evaluation)

Metric	Value
Total Nodes	2,477,446
Total Time	1,736 ms
Nodes/Second	1,427,100

GPU acceleration only becomes beneficial when batching approximately 7,000 positions per kernel dispatch. In traditional alpha-beta search, this is impractical because:

1. Search is sequential: each node depends on parent's bounds
2. Pruning eliminates most nodes before evaluation
3. Speculative evaluation wastes work on pruned branches

4.4 Search Performance

The engine achieves the following search performance on the standard 50-position benchmark suite at depth 13:

This NPS is achieved using CPU NNUE evaluation exclusively, as GPU dispatch overhead would reduce throughput by orders of magnitude.

4.5 Move Generation Verification

Table 4 shows perft results matching established correct values.

Additional tests verify Kiwipete (depth 5: 193,690,690), en passant, castling, and promotion positions.

Table 4. Perft verification (starting position)

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

5 Discussion

5.1 Why GPU Acceleration Fails for Alpha-Beta

Our measurements quantify a fundamental mismatch between GPU computing and traditional game tree search:

Dispatch overhead dominates: The 766 μs GPU dispatch cost includes unavoidable Metal framework operations. Even with unified memory eliminating data transfer, the command buffer lifecycle imposes significant latency.

Sequential dependencies prevent batching: Alpha-beta search is inherently sequential—each node's evaluation affects pruning decisions for siblings. True batching would require speculative evaluation of entire subtrees, wasting computation on pruned branches.

CPU evaluation is highly optimized: Stockfish's NNUE implementation uses SIMD intrinsics and incremental updates, achieving sub-microsecond evaluation. GPU acceleration must overcome not just dispatch overhead but also compete with highly-tuned CPU code.

5.2 When GPU Acceleration Helps

GPU acceleration becomes beneficial for:

- **Multi-position analysis:** Analyzing thousands of positions simultaneously (e.g., database analysis, puzzle solving)
- **Monte Carlo Tree Search:** MCTS naturally generates large batches of leaf evaluations
- **Training:** Neural network training requires batch processing

Leela Chess Zero [8] demonstrates successful GPU acceleration via MCTS, where batch sizes of 64–256 positions amortize dispatch overhead effectively.

5.3 Limitations

- GPU NNUE weights require separate loading; our benchmark used fallback evaluation

- True batched GPU evaluation (single dispatch for multiple positions) not implemented
- Single-threaded search; multi-threaded Lazy SMP would increase CPU throughput further
- No comparison with discrete GPU systems where unified memory is unavailable

6 Related Work

Leela Chess Zero [8] pioneered GPU-accelerated MCTS for chess, demonstrating that batch-oriented search algorithms map efficiently to GPUs. AlphaZero [9] showed that neural network evaluation can replace traditional handcrafted evaluation when combined with MCTS.

For alpha-beta search, GPU acceleration attempts have generally focused on move generation and position evaluation batching [10]. Our work provides concrete measurements showing why single-position GPU evaluation is counterproductive.

7 Conclusion

MetalFish demonstrates that GPU acceleration for chess engines requires careful consideration of dispatch overhead. Key findings:

1. GPU dispatch overhead (766 μs) exceeds CPU evaluation time (0.11 μs) by 6,900$\times$ for single positions
2. Crossover requires batching approximately 7,000 positions per dispatch
3. Traditional alpha-beta search cannot exploit GPU parallelism due to sequential dependencies
4. Unified memory eliminates data transfer overhead but not dispatch overhead
5. CPU-based evaluation achieves 1.43M nodes/second, competitive with modern engines

These findings suggest that GPU acceleration for chess engines is most effective when combined with batch-oriented search algorithms (MCTS) rather than traditional alpha-beta search.

Reproducibility

Hardware: Apple M2 Max, 64GB unified memory. Software: macOS 14.0, Xcode 15.0, Metal feature set macOS-GPUFamily2-v1. Source code: https://github.com/NripeshN/MetalFish. NNUE networks: nn-c288c895ea92.nnue (125MB), nn-37f18f62d772.nnue (6MB).

Acknowledgments

Thanks to the Stockfish and Leela Chess Zero teams for open-source contributions informing this work.

References

1. Stockfish Developers: Stockfish 16 NNUE documentation. https://github.com/official-stockfish/Stockfish (2024)
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.: Efficiently updatable neural-network-based evaluation functions for computer shogi. The 28th World Computer Shogi Championship Appeal Document (2018)
6. Apple Inc.: Metal Best Practices Guide: Resource Storage Modes. https://developer.apple.com/documentation/metal/resource_fundamentals/setting_resource_storage_modes (2024)
7. Zobrist, A.L.: A new hashing method with application for game playing. Tech. Rep. 88, Computer Sciences Department, University of Wisconsin (1970)
8. Leela Chess Zero: Neural network based chess engine. https://lczero.org/ (2024)
9. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815 (2017)
10. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep Blue. Artificial Intelligence 134(1-2), 57–83 (2002)