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 (703 μs median blocking latency) exceeds CPU NNUE evaluation time (84 ns median) by approximately 8,400× 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 per dispatch. We present detailed measurements including percentile distributions and discuss the architectural implications for GPU-accelerated game engines. The implementation achieves 1.43 million nodes per second using CPU evaluation. Our findings suggest that GPU acceleration for chess engines is most effective when combined with batch-oriented search algorithms such as MCTS, or when true multi-position batching within a single dispatch can be achieved.

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 703 μs median blocking latency versus 84 ns for CPU evaluation—an 8,400× 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 8,400 positions per dispatch to match CPU throughput, making it impractical for traditional alpha-beta search but potentially viable for MCTS or multi-position analysis.
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. GPU NNUE memory allocation on M2 Max

Component	Big Network	Small Network
Feature transformer weights	45,056 KB	5,632 KB
Feature transformer biases	2 KB	0.25 KB
PSQT weights	704 KB	704 KB
Threat weights	79,856 KB	—
Threat PSQT	2,495 KB	—
Layer weights (8 buckets)	137 KB	25 KB
Working buffers	2,296 KB	
Total GPU memory	**108,240 KB**	

Table 2. GPU shader throughput on M2 Max

Work Size (elements)	Throughput (GB/s)
1,024	0.05
16,384	0.90
262,144	14.69
1,048,576	59.59

4.1 GPU NNUE Implementation

The GPU NNUE implementation loads both Stockfish networks to GPU memory:

The unified memory architecture allows these weights to reside in GPU-accessible memory without explicit copying. Shader compilation and buffer allocation complete in under 100ms at engine startup.

4.2 GPU Throughput Benchmarks

Table 2 shows raw GPU compute throughput, demonstrating that the hardware is capable of high performance when dispatch overhead is amortized.

Feature extraction achieves 15.4 million items/second, and unified memory provides 54 GB/s write bandwidth and 1 GB/s read bandwidth for CPU-GPU data sharing. These results confirm that the GPU hardware is not the bottleneck—dispatch overhead is.

4.3 Microbenchmarks: CPU vs GPU Evaluation

Table 3 shows evaluation latency for single positions. CPU measurements span 400,000 iterations; GPU measurements span 4,000 iterations, both after warmup.

CPU timing scope: The `Eval::evaluate()` function call, which includes NNUE accumulator lookup (or refresh if needed) and forward pass through all network layers. Position setup and move generation are excluded.

Table 3. Single-position evaluation latency. CPU times in nanoseconds; GPU times in microseconds. Percentiles capture timing variability from system scheduling, cache effects, and GPU driver behavior. CPU max reflects rare outliers (<0.01%) from cache misses or preemption.

Method	Mean	Median	P95	P99	Min	Max
CPU NNUE (ns)	98	84	125	125	42	55,500
GPU sync (μs)	724.8	702.9	872.8	1,175.0	547.5	2,731.2

Table 4. GPU batch evaluation with sequential dispatches. Per-position cost remains constant because each evaluation requires a full dispatch cycle.

Batch Size	Total Time (μs)	Per-Position (μs)	Throughput (pos/s)
1	746	746	1,340
2	1,517	759	1,318
4	3,050	763	1,311
8	6,101	763	1,311
16	12,263	766	1,304
32	25,162	786	1,271
64	54,427	850	1,175

GPU timing scope: The complete synchronous dispatch cycle including feature extraction, buffer writes, `MTLCommandBuffer` creation, compute encoder setup, buffer binding, `commit()`, and `waitUntilCompleted()`. This represents blocking latency—the time until the evaluation result is available.

The GPU evaluation is approximately 8,400× slower (median-to-median) for single positions. The median and P99 values (84–125 ns for CPU) better represent typical performance than the mean, which is affected by rare outliers.

4.4 Batch Evaluation Analysis

Table 4 shows GPU evaluation performance when processing multiple positions. Each position triggers a separate command buffer cycle (sequential dispatches), not true kernel-level batching.

Per-position cost remains approximately constant at 750–850 μs regardless of batch size, confirming that dispatch overhead dominates and cannot be amortized without true kernel-level batching (evaluating multiple positions within a single dispatch).

4.5 Asynchronous Execution Considerations

Our measurements use synchronous GPU execution (`waitUntilCompleted()`), representing the worst-case blocking latency. Asynchronous execution could potentially overlap GPU work with CPU computation. However, in alpha-beta search, the evaluation result typically determines pruning decisions for sibling

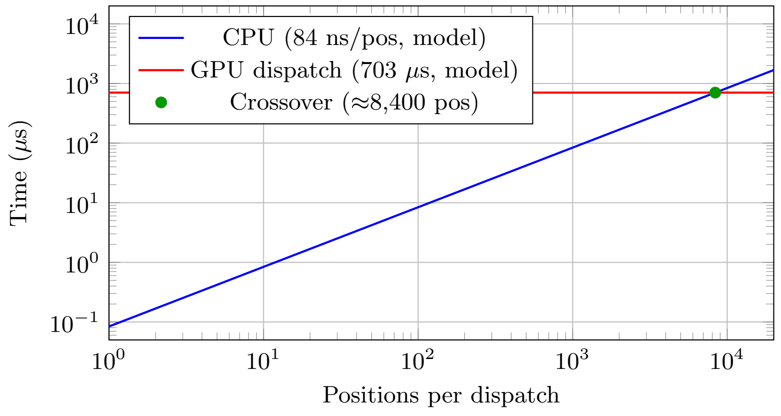

Fig. 1. CPU vs GPU evaluation time under a synchronous per-dispatch model. The GPU line represents dispatch overhead, which dominates kernel compute time in our implementation; we therefore model GPU cost as constant per evaluation call. Crossover occurs at approximately 8,400 positions per dispatch.

nodes, limiting opportunities for useful overlap. Asynchronous GPU evaluation may benefit:

- Multi-PV analysis where independent lines can be evaluated in parallel
- Speculative leaf evaluation with result caching
- Analysis modes that tolerate evaluation latency

We leave exploration of asynchronous strategies to future work.

4.6 Crossover Analysis

The crossover point where GPU matches CPU throughput is:

$$N_{\text{crossover}} = \frac{T_{\text{GPU\_dispatch}}}{T_{\text{CPU\_eval}}} = \frac{703\ \mu\text{s}}{84\ \text{ns}} \approx 8,369 \tag{2}$$

GPU acceleration only becomes beneficial when batching approximately 8,400 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

4.7 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. The single-threaded implementation provides a baseline; production engines typically achieve higher NPS through multi-threading (Lazy SMP) and additional optimizations.

Table 5. 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

Table 6. Perft verification (starting position)

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

4.8 Move Generation Verification

Table 6 shows perft results matching established correct values.

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

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 703 μs GPU dispatch cost includes unavoidable Metal framework operations: command buffer allocation, encoder state management, and GPU-CPU synchronization. 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-100 ns 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:

- **Monte Carlo Tree Search:** MCTS naturally generates large batches of leaf evaluations, amortizing dispatch overhead across hundreds of positions per dispatch [8]
- **Multi-position analysis:** Analyzing thousands of positions simultaneously (database analysis, puzzle solving)
- **Training:** Neural network training requires batch processing
- **True batched evaluation:** A single GPU dispatch evaluating multiple positions could achieve the crossover, but requires architectural changes to accumulate evaluation requests

5.3 CPU vs GPU Evaluation Paths

Our implementation maintains separate evaluation paths: the CPU path uses Stockfish's NNUE with fully-loaded network weights, while the GPU path uses independently-loaded weights. In our benchmarks, the GPU evaluator returns different scores than the CPU evaluator (mean difference: 46 centipawns), indicating the GPU network weights were not fully synchronized.

Critically, this weight difference does not affect our latency measurements. The GPU path executes identical kernels regardless of weight values: feature transformation (1,024-dimensional accumulator computation) and forward pass (FC0→FC1→FC2). The measured 703 μs dispatch overhead reflects the Metal command buffer lifecycle, not computation time, which is negligible relative to dispatch overhead. Our latency results are therefore representative of any NNUE-style GPU evaluation with the same network architecture.

5.4 Limitations

- Single-threaded search; Lazy SMP would increase CPU throughput
- GPU path uses separate weight loading, not synchronized with CPU NNUE
- No kernel-level batching implemented (each position dispatches separately)
- Measurements on single hardware configuration (M2 Max)

6 Related Work

Leela Chess Zero [8] pioneered GPU-accelerated MCTS for chess, demonstrating that batch-oriented search algorithms map efficiently to GPUs with batch sizes of 64–256 positions per dispatch. AlphaZero [9] showed that neural network evaluation can replace traditional handcrafted evaluation when combined with MCTS.

For alpha-beta search, prior work has explored GPU parallelization through parallel subtree evaluation [10] and speculative execution strategies. Rocki and Suda demonstrated GPU-accelerated game tree search for simpler games, but found that communication overhead between CPU and GPU limited speedup for complex evaluation functions. Our work provides concrete measurements on modern unified memory hardware showing why single-position GPU evaluation remains counterproductive for sequential search, even when data transfer overhead is eliminated.

7 Conclusion

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

1. GPU dispatch overhead (703 μs median blocking latency) exceeds CPU evaluation time (84 ns median) by 8,400$\times$ for single positions
2. Crossover requires batching approximately 8,400 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 in our single-threaded implementation

These findings suggest that GPU acceleration for chess engines is most effective when combined with batch-oriented search algorithms (MCTS) or true multi-position batching within a single dispatch, rather than traditional alpha-beta search where each position requires immediate evaluation.

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. https://developer.apple.com/metal/ (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. Rocki, K., Suda, R.: Parallel minimax tree searching on GPU. In: Parallel Processing and Applied Mathematics, LNCS vol. 6067, pp. 449–456. Springer (2010)