

MetalFish: GPU-Accelerated NNUE Evaluation on Apple Silicon

Nripesh Niketan¹

Independent Researcher

`nripesh14@gmail.com`

Abstract. We present MetalFish, a GPU-accelerated chess engine leveraging Apple Silicon’s unified memory architecture for NNUE evaluation. Through systematic benchmarking on M2 Max, we demonstrate: (1) single-position ($N=1$) blocking latency of $285\ \mu s$ median, (2) per-position cost of $0.4\ \mu s$ at batch size 4096, (3) true single-dispatch batching with up to $565\times$ speedup over sequential dispatches at $N=1024$, and (4) adaptive kernel selection with dual-perspective feature transformation, zero-copy buffer management, and asynchronous evaluation support. We support 64 features per perspective, and observed maximum 30 in our dataset. While synchronous blocking dispatch overhead ($148\ \mu s$) makes single-position GPU evaluation unsuitable for alpha-beta search without speculative evaluation, batch evaluation becomes throughput-competitive at $N \geq 512$, enabling efficient bulk analysis and MCTS-style evaluation.

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

1 Introduction

Modern chess engines combine alpha-beta search with neural network evaluation (NNUE) to achieve superhuman playing strength. While GPU acceleration has proven effective for batch-oriented algorithms like Monte Carlo Tree Search in Leela Chess Zero [2], its applicability to traditional alpha-beta search remains challenging due to sequential evaluation patterns.

Apple Silicon’s unified memory architecture eliminates explicit CPU-GPU memory transfers, potentially reducing the overhead that has historically limited GPU adoption in alpha-beta engines. This paper presents MetalFish, a GPU-accelerated chess engine that explores the practical limits of GPU evaluation on Apple Silicon.

1.1 Contributions

1. **Optimized GPU NNUE implementation:** We achieve $285\ \mu s$ median single-position blocking latency and $0.4\ \mu s$ per position at batch size 4096.
2. **Adaptive kernel selection:** We implement strategy-based kernel dispatch that selects dual-perspective or single-perspective kernels based on batch size for optimal performance.

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

3. **Zero-copy buffer management:** Pre-allocated staging buffers eliminate per-call allocations, writing directly to unified memory.
4. **Asynchronous evaluation:** We provide completion handler support for CPU/GPU work overlap.
5. **Verified true batching:** We demonstrate single-dispatch batching achieving up to $565\times$ speedup over sequential dispatches at batch size 1024.
6. **Comprehensive benchmarking:** We provide detailed stage breakdowns, latency percentiles, correctness verification, and scaling analysis across batch sizes 1–4096.

2 Background

2.1 NNUE Architecture

Stockfish’s NNUE [1, 5] uses HalfKAv2_hm features with sparse input and efficient incremental updates. Table 1 summarizes the architecture.

Feature requirements: HalfKAv2_hm generates one feature per non-king piece from each perspective. A position with 30 pieces (excluding 2 kings) generates up to 30 features per perspective. We support 64 features per perspective, providing headroom for all legal positions.

2.2 Metal Compute Model

Apple Metal [6] provides GPU compute through command buffers with unified memory access. Key characteristics:

- **Unified memory:** CPU and GPU share the same physical memory, eliminating explicit transfers
- **Command buffer lifecycle:** Allocation, encoding, submission, and synchronization
- **Threadgroup memory:** Fast on-chip memory for inter-thread communication
- **SIMD groups:** 32-wide SIMD execution on Apple GPUs
- **Completion handlers:** Asynchronous notification of GPU work completion

Table 2. GPU Configuration Constants

Parameter	Value
Max batch size	4,096
Max features per perspective	64
Max total features	128
Threadgroup size	256
SIMD group size	32
Forward pass threads	64

3 System Architecture

3.1 GPU Constants and Limits

Table 2 shows the key GPU configuration parameters.

3.2 Adaptive Kernel Selection

We implement strategy-based kernel selection through the `GPUTuningParams` structure:

```

1 enum class EvalStrategy {
2     CPU_FALLBACK,      // batch < 4
3     GPU_STANDARD,      // batch < 64
4     GPU_SIMD,          // batch >= 64
5     GPU_FEATURE_EXTRACT // batch >= 2048
6 };
7
8 EvalStrategy select_strategy(int batch_size) {
9     if (batch_size < min_batch_for_gpu)
10         return CPU_FALLBACK;
11     if (batch_size >= simd_threshold)
12         return GPU_SIMD;
13     return GPU_STANDARD;
14 }
```

Listing 1.1. Evaluation strategy selection

For batches ≥ 64 , we use the dual-perspective kernel that processes both white and black perspectives in a single 3D dispatch, reducing kernel launch overhead.

3.3 Zero-Copy Buffer Management

We pre-allocate all working buffers at initialization, eliminating per-call `std::vector` allocations:

```

1 // Get pointers to pre-allocated GPU buffers
2 int32_t* white_features_ptr =
3     static_cast<int32_t*>(white_features_buffer_ ->data());
4 uint32_t* white_counts_ptr =
5     static_cast<uint32_t*>(white_counts_buffer_ ->data());
6
7 // Write directly to unified memory (zero-copy)
8 for (int i = 0; i < batch_size; i++) {
9     white_offsets_ptr[i] = white_feature_idx;
10    // Extract features directly into GPU buffer
11    white_features_ptr[white_feature_idx++] = feat;
12 }

```

Listing 1.2. Direct buffer writes via unified memory

3.4 Kernel Optimizations

We implement three key optimizations in our Metal compute kernels:

1. Dual-perspective feature transformation: For batches ≥ 64 , we use a 3D dispatch that processes both perspectives simultaneously:

```

1 // 3D dispatch: (hidden_dim, 2 perspectives, batch_size)
2 encoder->dispatch_threads(hidden_dim, 2, batch_size);

```

Listing 1.3. Dual-perspective kernel dispatch

2. Unrolled accumulation: The forward pass uses 8-way unrolled loops for the FC0 layer, reducing loop overhead and enabling instruction-level parallelism.

3. Threadgroup memory: Intermediate results (FC0 outputs, skip connections) are stored in threadgroup memory, enabling efficient inter-thread communication without global memory round-trips.

3.5 Asynchronous Evaluation

We provide completion handler support for overlapping CPU and GPU work:

```

1 bool evaluate_batch_async(
2     GPUEvalBatch& batch,
3     std::function<void(bool)> completion_handler,
4     bool use_big_network = true);
5
6 // Usage: CPU can prepare next batch while GPU works
7 gpu_manager.evaluate_batch_async(batch, [](bool ok) {
8     // Called when GPU completes
9     process_results(batch.positional_scores);
10 });

```

Listing 1.4. Asynchronous batch evaluation

Algorithm 1 GPU Batch NNUE Evaluation

Require: Batch of N positions**Ensure:** Evaluation scores for all positions

```

1: // Stage 1: Strategy selection
2:  $strategy \leftarrow \text{SELECTSTRATEGY}(N)$ 
3: if  $strategy = \text{CPU\_FALLBACK}$  then
4:   return CPU evaluation
5: end if
6: // Stage 2: Feature extraction (direct to GPU buffers)
7: for each position do
8:   Write features directly to unified memory buffers
9: end for
10: // Stage 3: GPU evaluation (single command buffer)
11: if  $N \geq 64$  then
12:    $\text{DISPATCHTHREADS}(hidden\_dim, 2, N)$  ▷ Dual-perspective
13: else
14:    $\text{DISPATCHTHREADS}(hidden\_dim, N)$  ▷ Single-perspective
15: end if
16:  $\text{BARRIER}$ 
17:  $\text{DISPATCHTHREADGROUPS}(N, \text{threads}=64)$  ▷ Forward pass
18:  $\text{SUBMITANDWAIT}$ 
19: return scores from output buffer

```

3.6 Batch Evaluation Pipeline

Algorithm 1 shows the optimized batch evaluation pipeline.

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), nn-37f18f62d772.nnue (6MB)

4.2 Benchmark Dataset

Our benchmark uses 8 unique FEN positions representing diverse game phases (opening, middlegame, endgame), cycled to create 2048 test positions. Piece counts range from 2 to 32, with most positions containing 28–32 pieces.

4.3 Timing Methodology

All measurements use `std::chrono::high_resolution_clock`:

Table 3. Feature Count Distribution (2,048 positions)

Metric	Value
Max features observed (total)	60
Max features per perspective	30
GPU limit per perspective	64
Positions exceeding limit	0%

Table 4. GPU Dispatch Overhead—Minimal Kernel (N=1,000)

Statistic Latency (μs)	
Median	148
P95	300
P99	944

- **Warmup:** 100 iterations discarded
- **Samples:** 100–100,000 iterations depending on variance
- **Statistics:** Median, P95, P99 reported
- **GPU timing:** Blocking `waitUntilCompleted()` (synchronous)

5 Results

5.1 Feature Coverage

Table 3 shows the feature count distribution.

5.2 GPU Dispatch Overhead

Table 4 shows minimal-kernel dispatch overhead.

The 148 μs median dispatch overhead represents the minimum cost for any GPU operation in synchronous blocking mode.

5.3 GPU Stage Breakdown

Table 5 decomposes end-to-end latency.

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

5.4 Batch Latency Scaling

Table 6 shows end-to-end latency across batch sizes up to the maximum supported (4096).

Key findings: (1) Latency is approximately constant (279–311 μs) for batch sizes 1–256, showing dispatch dominance. (2) Per-position cost drops from 285 μs (N=1) to 0.4 μs (N=4096). (3) Latency increases linearly beyond N=512, indicating kernel compute becoming significant.

Table 5. GPU Stage Breakdown (N=100 iterations each)

Batch Size	CPU Prep	GPU Eval	GPU %
1	0.2 μs	283 μs	99.9%
8	0.3 μs	278 μs	99.9%
512	6.3 μs	419 μs	98.5%

Table 6. GPU End-to-End Batch Latency (N=100 iterations)

Batch Size	Median (μs)	P95 (μs)	P99 (μs)	Per-Pos (μs)
1	285	386	1,044	285.0
8	279	424	1,179	34.8
32	280	496	1,038	8.7
64	282	501	1,041	4.4
128	289	457	970	2.3
256	311	448	1,244	1.2
512	384	642	1,068	0.8
1024	574	1,015	1,329	0.6
2048	926	1,295	1,564	0.5
3072	1,260	1,871	2,009	0.4
4096	1,623	1,796	2,022	0.4

5.5 True Batching Verification

Table 7 compares sequential vs batched dispatches.

The sequential case creates N separate command buffers; the batched case uses one command buffer with two dispatches (feature transform + forward pass). Speedups scale super-linearly due to reduced per-dispatch overhead amortization.

5.6 GPU Evaluation Correctness

Table 8 verifies GPU evaluation consistency.

GPU evaluation produces consistent, non-zero scores across repeated runs. Note: GPU uses NNUE weights while CPU baseline uses simple material+PST, so absolute values differ.

6 Discussion

6.1 When GPU Evaluation Helps

GPU batch evaluation becomes throughput-competitive at $N \geq 512$ (0.8 μs /position). This enables:

- **Database analysis:** Evaluating thousands of positions from game databases

Table 7. True Batching Verification (N=100 iterations)

	N Sequential (N×1 CB)	Batched (1×1 CB)	Speedup
16	5,395 μ s	287 μ s	18.8×
64	21,051 μ s	305 μ s	69.0×
256	83,820 μ s	345 μ s	243.2×
1024	334,604 μ s	592 μ s	564.9×

Table 8. GPU Evaluation Correctness (100 positions)

Metric	Value
Non-zero GPU scores	100%
Consistent across runs	100%
Mean GPU score	188

- **MCTS evaluation:** Monte Carlo Tree Search naturally batches leaf evaluations
- **Training data generation:** Bulk position evaluation for neural network training
- **Parallel analysis:** Evaluating multiple candidate moves simultaneously

6.2 Alpha-Beta Limitations

Single-position GPU blocking latency (285 μ s) makes GPU evaluation unsuitable for alpha-beta search in synchronous blocking mode and without speculative evaluation. Alpha-beta search evaluates positions sequentially with data-dependent pruning, making batch accumulation impractical without significant architectural changes.

Our asynchronous evaluation API (`evaluate_batch_async`) enables CPU/GPU overlap, but the fundamental sequential nature of alpha-beta limits its applicability.

6.3 Optimization Impact

Table 9 summarizes our optimization contributions.

7 Related Work

Leela Chess Zero [2] demonstrates successful GPU acceleration through MCTS, which naturally batches evaluations. AlphaZero [3] showed neural network evaluation can replace handcrafted evaluation with batch-oriented search.

Table 9. Optimization Summary

Optimization	Benefit
Zero-copy buffers	Eliminates allocation overhead
Dual-perspective kernel	Single dispatch for both perspectives
Adaptive selection	Optimal kernel per batch size
8-way unrolling	Improved ILP in FC0
Threadgroup memory	Fast inter-thread communication
Async evaluation	CPU/GPU work overlap

For alpha-beta, Rocki and Suda [4] explored GPU parallelization through parallel subtree evaluation. Our work extends this to unified memory hardware with optimized NNUE kernels.

Apple’s Metal documentation [6, 7] recommends minimizing command buffer submissions and using threadgroup memory for intermediate results.

8 Conclusion

We presented MetalFish, a GPU-accelerated chess engine achieving:

1. **285 μ s** median single-position blocking latency
2. **0.4 μ s** per-position cost at batch size 4096
3. **565 \times** true batching speedup at N=1024
4. **100%** GPU evaluation consistency
5. **Zero-copy** buffer management via unified memory
6. **Adaptive** kernel selection for optimal performance
7. **Asynchronous** evaluation support for CPU/GPU overlap

GPU acceleration is effective for batch-oriented workloads (MCTS, database analysis, training) but synchronous blocking dispatch overhead makes it unsuitable for alpha-beta’s sequential evaluation pattern without speculative evaluation or asynchronous queuing. Our optimized Metal kernels with adaptive selection and zero-copy buffer management provide a solid foundation for GPU-accelerated chess evaluation on Apple Silicon.

Reproducibility

Hardware: Apple M2 Max, 64GB. **Software:** macOS 14.0, Xcode 15.0. **Build:** CMake, -O3, LTO. **Source:** <https://github.com/NripeshN/MetalFish>. **Benchmark:** `gpubench UCI` command.

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)