MetalFish: GPU-Accelerated NNUE Evaluation on Apple Silicon

Nripesh Niketan[1]

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 278 μs median, (2) per-position cost of 0.3 μs at batch size 4096, (3) true single-dispatch batching with up to 530× speedup over sequential dispatches, and (4) SIMD-optimized kernels with 8-way unrolled accumulation. We support 64 features per perspective, and observed maximum 30 in our dataset. While synchronous blocking dispatch overhead (147 μs) 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 278 μs median single-position blocking latency and 0.3 μs per position at batch size 4096.
2. **Complete feature coverage**: We support 64 features per perspective (128 total), and observed maximum 30 per perspective in our benchmark dataset—well within the limit for all standard chess positions.

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. **Verified true batching**: We demonstrate single-dispatch batching achieving up to 530× speedup over sequential dispatches at batch size 1024.
4. **SIMD-optimized kernels**: We present Metal compute kernels with 8-way unrolled accumulation, dual-perspective feature transformation, and threadgroup memory optimization.
5. **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

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 Kernel Optimizations

We implement three key optimizations in our Metal compute kernels:

1. SIMD-aware feature transformation: The feature transform kernel processes features with memory coalescing, accessing weights with stride patterns that maximize cache utilization.

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

```
for (; i + 7 < hidden_dim; i += 8) {
    int8_t c0 = clipped_relu(acc[i] >> SCALE);
    int8_t c1 = clipped_relu(acc[i+1] >> SCALE);
    // ... c2-c7
    sum += c0 * weights[(i)*stride + out];
    sum += c1 * weights[(i+1)*stride + out];
    // ... 6 more
}
```

Listing 1.1. 8-way unrolled FC0 accumulation

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

3.3 Batch Evaluation Pipeline

Algorithm 1 shows the batch evaluation pipeline.

Algorithm 1 GPU Batch NNUE Evaluation

```
Require: Batch of $N$ positions
Ensure: Evaluation scores for all positions
// Stage 1: Feature extraction (CPU)
for each position do
    Extract white/black features from bitboards
    Store in contiguous buffers with offsets
end for
// Stage 2: GPU evaluation (single command buffer)
Upload features to unified memory buffers
DispatchThreads($hidden\_dim \times N$)                ▷ Feature transform
Barrier
DispatchThreadgroups($N$, threads=64)                  ▷ Forward pass
SubmitAndWait
return scores from output buffer
```

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`:

- **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.

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	147
P95	258
P99	320

5.2 GPU Dispatch Overhead

Table 4 shows minimal-kernel dispatch overhead.

The 147 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.

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 (271–289 μs) for batch sizes 1–128, showing dispatch dominance. (2) Per-position cost drops from 278 μs (N=1) to 0.3 μs (N=4096). (3) Latency increases linearly beyond N=512, indicating kernel compute becoming significant.

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 linearly with batch size, confirming true single-dispatch batching.

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

Batch Size	CPU Prep	GPU Eval	GPU %
1	0.2 μs	275 μs	99.9%
8	0.3 μs	266 μs	99.9%
512	5.9 μs	373 μs	98.5%

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

Batch	Median	P95	P99	Per-Pos
Size	(μs)	(μs)	(μs)	(μs)
1	278	370	487	278.0
8	271	365	374	33.8
32	271	365	387	8.5
128	289	378	418	2.3
512	377	483	603	0.7
1024	549	648	687	0.5
2048	833	953	980	0.4
3072	1,122	1,240	1,370	0.4
4096	1,433	1,579	1,602	0.3

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.

5.7 Search Performance

The engine achieves 1.38M nodes/second using CPU NNUE evaluation:

6 Discussion

6.1 When GPU Evaluation Helps

GPU batch evaluation becomes throughput-competitive at $N \geq 512$ (0.7 μs/position). This enables:

- **Database analysis**: Evaluating thousands of positions from game databases
- **MCTS evaluation**: Monte Carlo Tree Search naturally batches leaf evaluations
- **Training data generation**: Bulk position evaluation for neural network training

Table 7. True Batching Verification (N=50 iterations)

N	Sequential (N×1 CB)	Batched (1×1 CB)	Speedup
16	4,679 μs	274 μs	17.1×
64	18,647 μs	291 μs	64.1×
256	74,702 μs	316 μs	236.2×
1024	297,663 μs	562 μs	529.5×

Table 8. GPU Evaluation Correctness (100 positions)

Metric	Value
Non-zero GPU scores	100%
Consistent across runs	100%
Mean |GPU score|	534

6.2 Alpha-Beta Limitations

Single-position GPU blocking latency (278 μs) 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.

6.3 Optimization Impact

Our optimizations target three areas:

- **Memory access**: SIMD-aware coalesced access patterns
- **Compute**: 8-way unrolled accumulation loops
- **Communication**: Threadgroup memory for intermediate results

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.

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.

Table 9. Search Benchmark (50 positions, depth 13)

Metric	Value
Total Nodes	2,477,446
Total Time	1,792 ms
Nodes/Second	1,382,503

8 Conclusion

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

1. **278 μs** median single-position blocking latency
2. **0.3 μs** per-position cost at batch size 4096
3. **530**× true batching speedup at N=1024
4. **100%** GPU evaluation consistency
5. **1.38M** nodes/second search performance

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 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)