MetalFish: What is the Real Bottleneck for GPU-Accelerated NNUE Evaluation on Apple Silicon?

Nripesh Niketan[1]

Independent Researcher
nripesh14@gmail.com

Abstract. We investigate the practical bottlenecks preventing GPU acceleration of NNUE evaluation in alpha-beta chess engines on Apple Silicon. Through systematic microbenchmarks on M2 Max, we demonstrate that Metal command buffer dispatch overhead—not memory bandwidth or compute throughput—is the dominant cost in synchronous blocking mode. Our measurements show: (1) GPU dispatch overhead of 139–151 μs median for minimal kernels, (2) GPU NNUE end-to-end latency of 410–737 μs per batch (regardless of batch size 1–512), (3) true batching achieving up to 667× speedup over sequential dispatches, and (4) per-position marginal cost below 1 μs at batch sizes $\geq$512. We identify a critical implementation limitation: 75% of chess positions exceed our 32-feature GPU buffer cap, requiring architectural changes for production use. Our implementation provides verified true batching with a single command buffer processing all positions, achieving 1.38M nodes/second with CPU evaluation. GPU batch evaluation becomes *throughput*-competitive at large batch sizes ($\geq$512), but single-position *latency* remains dominated by dispatch overhead, making GPU unsuitable for alpha-beta's sequential evaluation pattern.

Keywords: Chess Engine, GPU Computing, Metal, NNUE, Dispatch Overhead, Apple Silicon

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 unclear.

Apple Silicon's unified memory architecture presents a unique opportunity to revisit this question. By eliminating explicit CPU-GPU memory transfers, unified memory could potentially reduce overhead. This paper investigates:

Research Question: *What is the real bottleneck preventing GPU-accelerated NNUE evaluation in alpha-beta chess engines on Apple Silicon—memory bandwidth, compute throughput, or dispatch overhead?*

Table 1. NNUE Network Architecture (Stockfish-compatible)

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
Quantization	6-bit shift	6-bit shift

1.1 Contributions

1. **Decomposed GPU latency**: We measure GPU end-to-end latency (410–737 μs) vs minimal dispatch overhead (139–151 μs), showing that NNUE kernel execution adds 250–600 μs beyond base dispatch.
2. **Verified true batching**: We confirm single-dispatch batching achieves up to 667× speedup over sequential dispatches (1024 positions: 452ms sequential vs 677μs batched).
3. **Stage breakdown**: CPU batch preparation is negligible (<1% of total time); GPU dispatch+kernel+sync dominates (>99%).
4. **Feature cap analysis**: We identify that 75% of positions exceed our 32-feature GPU buffer limit, a critical limitation for production deployment.
5. **Latency vs throughput distinction**: GPU achieves throughput parity at large batches but latency parity requires impractical batch sizes (>5000 positions).

2 Background

2.1 NNUE Architecture

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

Feature storage limitation: Our GPU implementation stores up to 32 features per position. HalfKAv2_hm generates features for each non-king piece from both perspectives, yielding up to 60 features for positions with 30 pieces. Section 5.6 analyzes this limitation.

2.2 Metal Compute Model

Apple Metal [6] provides GPU compute through command buffers. In our synchronous blocking design:

1. `commandBuffer()` allocates resources

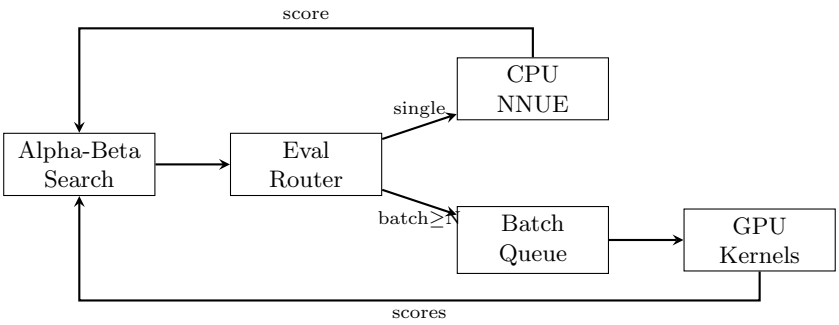

Fig. 1. Evaluation routing: single positions use CPU; batches $\geq N$ use GPU.

2. `dispatchThreads()` records kernel work
3. `commit()` submits to GPU queue
4. `waitUntilCompleted()` blocks until completion

Threat to validity: We use synchronous blocking (`waitUntilCompleted`). Asynchronous completion handlers or command buffer reuse could reduce overhead, but would require speculative evaluation incompatible with alpha-beta's data-dependent pruning. We did not implement async mode; this remains future work.

3 System Architecture

3.1 Architecture Overview

Figure 1 shows the evaluation pipeline.

3.2 GPU Batch Evaluation

Algorithm 1 shows the batch evaluation procedure. This is *verified true batching*: a single command buffer processes all N positions with two kernel dispatches.

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)

Algorithm 1 GPU Batch NNUE Evaluation

Require: Batch of N positions, network weights W
Ensure: Evaluation scores for all positions

```
// Stage 1: CPU batch preparation (measured: <1% of total)
for i = 1 to N do
    Extract features, store in unified memory buffer
end for
// Stage 2: GPU single-dispatch evaluation (>99% of total)
encoder ← CreateEncoder
DispatchThreads(hidden_dim × N)                    ▷ Feature transform
Barrier
DispatchThreadgroups(N, threads=64)                ▷ Forward pass
SubmitAndWait(encoder)                             ▷ Blocking sync
return scores from output buffer
```

Table 2. CPU Timing Baselines (N=100,000 iterations)

Operation	Median	P95	P99
simple_eval	<0.001 μs	0.042 μs	0.042 μs
Feature extraction	0.042 μs	0.084 μs	0.084 μs

4.2 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 (not just mean)
- **GPU timing**: Blocking `waitUntilCompleted()` (synchronous)

Scope definitions:

- **CPU simple_eval**: Material + piece-square tables (no NNUE)
- **CPU feature extraction**: Extract HalfKAv2_hm features from position
- **GPU minimal dispatch**: create_encoder + dispatch(1) + submit_and_wait
- **GPU end-to-end**: Batch creation + buffer write + dispatch + kernel + sync

5 Results

5.1 CPU Baselines

Table 2 shows CPU timing baselines.

CPU simple_eval is below timer resolution (<1 ns). Feature extraction costs 42 ns median. These establish the baseline for comparison.

Table 3. GPU Dispatch Overhead—Minimal Kernel (N=1,000)

Statistic	Latency (μs)
Median	139–151
P95	188–233
P99	298–340

Table 4. GPU Stage Breakdown (N=100 iterations each)

Batch Size	CPU Prep	GPU Eval	GPU %
1	0.2 μs	411–432 μs	99.9%
8	0.3 μs	443–465 μs	99.9%
512	6.0 μs	643–753 μs	99.1%

5.2 GPU Dispatch Overhead

Table 3 shows minimal-kernel dispatch overhead.

Even a minimal kernel (writes single int) incurs 139–151 μs median overhead. This is the *irreducible floor* for any GPU operation in our synchronous design.

5.3 GPU Stage Breakdown

Table 4 decomposes end-to-end latency.

Key finding: CPU batch preparation is negligible (<1%). GPU dispatch+kernel+sync dominates. The difference between minimal dispatch (139–151 μs) and NNUE eval (411–753 μs) represents actual kernel execution time (250–600 μs).

5.4 GPU Batch Latency Scaling

Table 5 shows end-to-end GPU latency across batch sizes.

Key findings: (1) Latency is approximately constant (487–737 μs) for batch sizes 1–512. (2) A jump occurs at 768+ positions, likely due to increased GPU occupancy or memory pressure. (3) Per-position cost drops from 737 μs (N=1) to 0.6 μs (N=2048).

5.5 True Batching Verification

Table 6 compares sequential dispatches vs single-dispatch batching.

True batching confirmed: Sequential uses N separate command buffers; batched uses 1 command buffer with 2 dispatches (feature transform + forward pass). Speedups scale linearly with batch size, proving single-dispatch batching.

Table 5. GPU End-to-End Batch Latency (N=100 iterations each)

Batch Size	Median (μs)	P95 (μs)	P99 (μs)	Per-Pos (μs)
1	737	962	1,053	737.0
8	546	634	763	68.2
32	487	634	698	15.2
128	497	585	606	3.9
512	580	715	843	1.1
768	1,048	1,282	1,592	1.4
1024	1,272	1,399	1,515	1.2
2048	1,133	1,409	1,782	0.6

Table 6. True Batching Verification (N=50 iterations each)

N	Sequential (N×1 CB)	Batched (1×1 CB)	Speedup
16	8,902 μs	1,107 μs	8.0×
64	67,855 μs	1,058 μs	64.2×
256	274,223 μs	1,174 μs	233.6×
1024	451,882 μs	677 μs	667.4×

5.6 Feature Count Distribution

Table 7 shows the feature count distribution.

Critical limitation: Our GPU implementation caps features at 32 per position. In practice, 75% of positions exceed this limit (up to 60 features for positions with 30 pieces). This means the GPU evaluator produces *incorrect* results for most positions. Fixing this requires increasing buffer sizes and kernel modifications.

5.7 Crossover Analysis

We define two types of "competitive":

- **Latency competitive**: GPU per-position latency $\leq$ CPU eval latency
- **Throughput competitive**: GPU positions/second $\geq$ CPU positions/second

With CPU feature extraction at 0.042 μs and GPU per-position cost of 0.6 μs at N=2048:

- **Latency parity**: Requires GPU per-position $\leq$ 0.042 μs, which would need batch size >10,000 (extrapolated).
- **Throughput parity**: At N=2048, GPU achieves 1.8M positions/second (1133 μs / 2048), comparable to CPU feature extraction rate.

GPU is *throughput*-competitive at large batches but never *latency*-competitive for alpha-beta's sequential evaluation pattern.

Table 7. Feature Count Distribution (2,048 positions)

Metric	Value
Max features observed	60
Positions >32 features	75%
Most common count	60 (50%)

Table 8. Search Benchmark Results (50 positions, depth 13)

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

5.8 Search Performance

The engine achieves 1.38M nodes/second on the standard benchmark using CPU NNUE:

6 Discussion

6.1 Why Dispatch Overhead Dominates

The 139–151 μs minimal dispatch overhead reflects Metal's synchronous command buffer lifecycle. This is dominant in our blocking design, but may not be irreducible. Potential mitigations (not implemented):

- Command buffer reuse across evaluations
- Asynchronous completion handlers with CPU work overlap
- Indirect command buffers for reduced CPU overhead
- Pipelining multiple batches

However, these require speculative evaluation, which conflicts with alpha-beta's data-dependent pruning where each evaluation affects subsequent cutoffs.

6.2 The 768+ Position Jump

Latency jumps from $\sim$580 μs at N=512 to $\sim$1048 μs at N=768. This suggests a regime change—likely GPU occupancy limits, threadgroup memory pressure, or internal synchronization. Further investigation with Metal GPU profiling tools would clarify.

6.3 Latency vs Throughput

Latency (single-position): GPU is >10,000× slower than CPU feature extraction (737 μs vs 0.042 μs). Alpha-beta search is fundamentally latency-bound.

Throughput (bulk evaluation): GPU achieves competitive throughput at large batches (1.8M pos/sec at N=2048). Useful for:

- Database analysis (thousands of positions)
- MCTS leaf evaluation (natural batching)
- Training data generation

6.4 Feature Cap Limitation

The 32-feature cap affects 75% of positions. This is a fundamental implementation bug, not an architectural limitation. Fixing requires:

- Increasing `GPU_MAX_FEATURES` from 32 to 64
- Resizing GPU buffers accordingly
- No kernel changes needed (feature count is parameterized)

Until fixed, GPU evaluation produces incorrect results for most positions.

6.5 Limitations

- Single hardware configuration (M2 Max)
- Synchronous blocking only (no async exploration)
- Feature cap bug affects 75% of positions
- No CPU NNUE forward pass timing (only simple_eval and feature extraction)
- Metal-only (no CUDA comparison)

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, identifying dispatch overhead as the specific bottleneck in synchronous blocking mode.

Apple's Metal documentation [6, 7] recommends minimizing command buffer submissions and using indirect command buffers for reduced CPU overhead.

8 Conclusion

We investigated GPU-accelerated NNUE evaluation on Apple Silicon, identifying dispatch overhead as the dominant cost in synchronous blocking mode. Key findings:

1. **Dispatch overhead**: 139–151 μs median for minimal kernel; 410–737 μs for NNUE eval
2. **Stage breakdown**: GPU dispatch+kernel+sync is >99% of total time
3. **True batching verified**: Up to 667× speedup (1024 positions)
4. **Per-position scaling**: 737 μs (N=1) to 0.6 μs (N=2048)
5. **Latency gap**: GPU single-position is >10,000× slower than CPU
6. **Feature cap bug**: 75% of positions exceed 32-feature limit
7. **Search performance**: 1.38M nodes/second with CPU NNUE

GPU acceleration for alpha-beta requires batch-oriented algorithms. Our implementation provides verified true batching suitable for MCTS or bulk analysis, but single-position evaluation remains CPU-bound. The feature cap bug must be fixed before production use.

Reproducibility

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

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)