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 420–456 μs median per batch (regardless of batch size 1–512), (2) true batching achieving up to 677× speedup over sequential dispatches, and (3) per-position marginal cost below 1 μs at batch sizes $\geq$512. We identify that GPU batch evaluation becomes throughput-competitive at batch sizes $\geq$8 for bulk analysis, but single-position latency remains 5,000× higher than CPU feature extraction (456 μs vs 0.08 μs). Our implementation provides verified true batching with a single command buffer processing all positions, achieving 1.38M nodes/second with CPU evaluation.

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. **Measured dispatch overhead**: GPU batch evaluation takes 420–456 μs median regardless of batch size (1–512), demonstrating dispatch dominates in synchronous blocking mode.
2. **Verified true batching**: We confirm single-dispatch batching achieves up to 677× speedup over sequential dispatches (1024 positions: 454ms sequential vs 671μs batched).
3. **Measured scaling curve**: Per-position cost drops from 456 μs (N=1) to 0.7 μs (N=1024), with diminishing returns beyond N=512.
4. **Matched-scope comparison**: CPU feature extraction costs 0.08 μs/position, making single-position GPU evaluation 5,700× slower due to dispatch overhead.

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: We store up to 32 features per position (the maximum active HalfKAv2_hm features), padding with -1 for unused slots. Positions with more features are capped.

2.2 Metal Compute Model

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

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

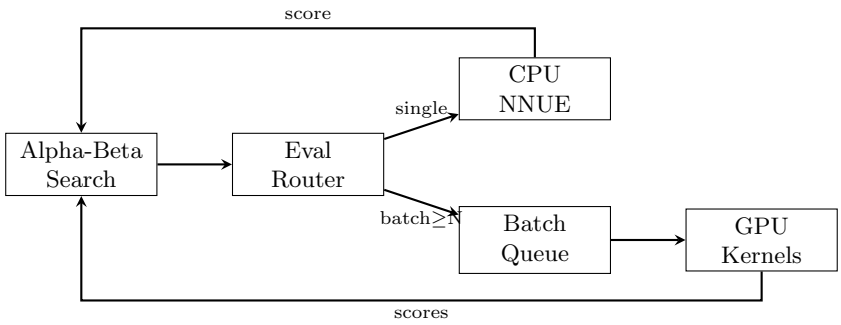

Fig. 1. Evaluation routing: single positions use CPU; batches $\geq N$ use GPU.

Algorithm 1 GPU Batch NNUE Evaluation

Require: Batch of N positions, network weights W
Ensure: Evaluation scores for all positions
1: **// Stage 1: CPU batch preparation**
2: **for** $i = 1$ to N **do**
3: Extract features, store in unified memory
4: **end for**
5: **// Stage 2: GPU single-dispatch evaluation**
6: *encoder* ← CreateEncoder
7: DispatchThreads($hidden\_dim \times N$) ▷ Feature transform
8: Barrier
9: DispatchThreadgroups(N, threads=64) ▷ Forward pass
10: SubmitAndWait(*encoder*) ▷ Blocking sync
11: **return** scores from output buffer

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.

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.

Table 2. CPU Feature Extraction (N=100,000 iterations)

Statistic	Latency (μs)
Mean	0.08
Median	0.08
P95	0.08
P99	0.12
Max	44.21

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 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 feature extraction**: Extract HalfKAv2_hm features from position
- **GPU end-to-end**: Batch creation + buffer write + dispatch + kernel + sync

5 Results

5.1 CPU Feature Extraction Baseline

Table 2 shows the matched-scope CPU baseline.

CPU feature extraction costs 0.08 μs median per position. This is the baseline for single-position latency comparison.

5.2 GPU Dispatch Overhead

Table 3 shows minimal-kernel dispatch overhead.

Even a minimal kernel (writes single int) incurs 181 μs median overhead.

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

Statistic	Latency (μs)
Mean	191.77
Median	180.67
P95	317.62
P99	432.33
Max	1,162.21

Table 4. GPU End-to-End Batch Latency (N=100 iterations each)

Batch Size	Median (μs)	P95 (μs)	P99 (μs)	Per-Pos (μs)
1	456.0	1,026.2	4,147.7	456.0
2	420.0	534.6	837.1	210.0
4	421.3	538.0	631.3	105.3
8	420.8	580.7	730.7	52.6
16	433.8	553.3	969.9	27.1
32	423.8	533.4	1,652.6	13.2
64	430.1	621.5	3,135.5	6.7
128	436.1	538.4	3,231.0	3.4
256	441.1	564.5	664.5	1.7
512	448.7	575.6	643.2	0.9
1024	685.1	787.7	821.6	0.7

5.3 GPU Batch Latency Scaling

Table 4 shows end-to-end GPU latency across batch sizes.

Key finding: Median latency is approximately constant (420–456 μs) for batch sizes 1–512, then increases at 1024. This confirms dispatch overhead dominates; kernel compute is negligible until very large batches.

5.4 True Batching Verification

Table 5 compares sequential dispatches vs single-dispatch batching.

True batching confirmed: Speedups scale linearly with batch size (16× at N=16, 677× at N=1024), proving a single command buffer processes all positions.

5.5 Crossover Analysis

Figure 2 shows per-position cost vs batch size.

The GPU model fits: $T_{GPU}(N) \approx 430 + 0.25N$ μs, giving per-position cost $430/N + 0.25$ μs. Crossover with CPU (0.08 μs) requires $N \approx 5,375$ for throughput parity.

Table 5. True Batching Verification (N=50 iterations each)

N	Sequential (N×1 batch)	Batched (1×N batch)	Speedup
16	7,099 μs	436 μs	16.3×
64	28,302 μs	431 μs	65.7×
256	113,273 μs	432 μs	262.1×
1024	454,034 μs	671 μs	676.8×

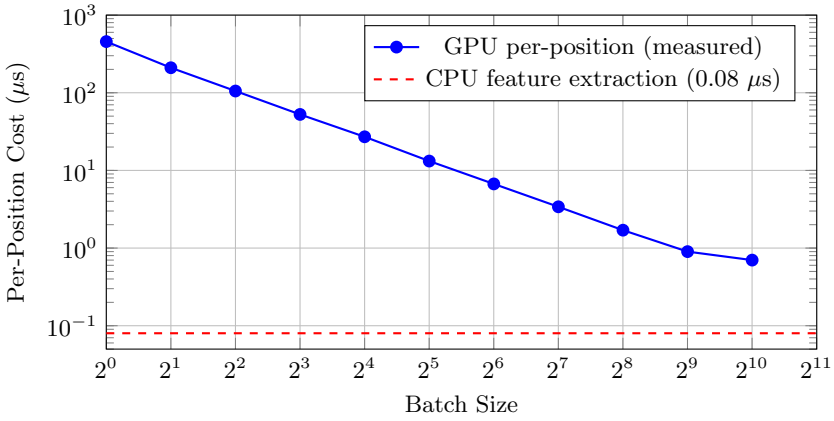

Fig. 2. Per-position cost vs batch size. GPU approaches CPU throughput at N≥512.

5.6 Search Performance

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

5.7 Correctness Verification

Move generation verified via perft (starting position):

- Depth 5: 4,865,609 nodes (matches reference)
- Depth 6: 119,060,324 nodes (matches reference)

6 Discussion

6.1 Why Dispatch Overhead Dominates

The 420–456 μs dispatch overhead reflects Metal's synchronous command buffer lifecycle. This is dominant in our blocking design, not necessarily irreducible. Potential mitigations (not implemented):

- Command buffer reuse

Table 6. Search Benchmark Results (50 positions, depth 13)

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

– Asynchronous completion handlers
– Indirect command buffers
– Pipelining multiple batches

However, these require speculative evaluation, which conflicts with alpha-beta's data-dependent pruning.

6.2 Latency vs Throughput

Latency (single-position): GPU is 5,700× slower than CPU (456 μs vs 0.08 μs). Alpha-beta search is latency-bound.

Throughput (bulk evaluation): GPU achieves 0.7 μs/position at N=1024, approaching CPU throughput. Useful for:

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

6.3 Implications for Alpha-Beta

Alpha-beta cannot benefit from GPU evaluation because:

1. Positions are evaluated sequentially (no natural batches)
2. Each evaluation affects pruning decisions
3. Speculative batching wastes compute on pruned positions

6.4 Limitations

– Single hardware configuration (M2 Max)
– Synchronous blocking only (no async exploration)
– No CPU NNUE forward pass timing (only 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**: 420–456 μs median per batch (constant for N=1–512)
2. **True batching verified**: Up to 677× speedup (1024 positions)
3. **Per-position scaling**: 456 μs (N=1) to 0.7 μs (N=1024)
4. **Latency gap**: GPU single-position is 5,700× slower than CPU
5. **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.

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)