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 fundamental barrier. Our measurements show: (1) GPU dispatch overhead of 150 μs median (95 μs minimum) per command buffer, (2) GPU compute throughput of 44 GB/s when dispatch is amortized, and (3) true batching achieving 1.8× speedup over sequential dispatches. We identify the crossover point where GPU batch evaluation becomes competitive: approximately 1,800 positions per batch for throughput parity with CPU NNUE. Our implementation, MetalFish, provides a complete Metal backend with verified true batching, achieving 1.38M nodes/second with CPU evaluation. We conclude that GPU acceleration requires batch-oriented search algorithms; single-position evaluation in alpha-beta remains CPU-bound due to irreducible dispatch latency.

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. The sequential, data-dependent nature of alpha-beta pruning creates fundamental challenges for GPU parallelization.

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 the overhead that traditionally makes GPU evaluation impractical for alpha-beta search. This paper investigates a specific research question:

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
Bucket selection	piece count / 4	piece count / 4
Quantization	6-bit shift	6-bit shift

1.1 Contributions

1. **Dispatch overhead characterization**: We measure Metal command buffer lifecycle costs with percentile distributions, showing 150 μs median overhead per dispatch (Table 2).
2. **True batching verification**: We confirm our GPU implementation uses genuine single-dispatch batching (not sequential per-position dispatches), achieving 1.8× speedup over sequential submission.
3. **Crossover analysis**: We identify the batch size (≈1,800 positions) where GPU throughput matches CPU NNUE, explaining why alpha-beta search cannot benefit.
4. **Complete implementation**: MetalFish provides a tested Metal backend with NNUE weight extraction, feature transformation, and fused forward pass kernels, achieving 1.38M nodes/second.

2 Background

2.1 NNUE Architecture

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

The feature transformer converts sparse HalfKAv2_hm features to dense accumulators. Each active feature (typically 20–30 per position) indexes into a weight matrix, and the corresponding row is added to the accumulator. The forward pass applies FC0 (sparse input with SqrClippedReLU), FC1 (ClippedReLU), and FC2 with a skip connection.

2.2 Metal Compute Model

Apple Metal [6] provides GPU compute through command buffers:

1. **Encoder creation**: `commandBuffer()` allocates a command buffer

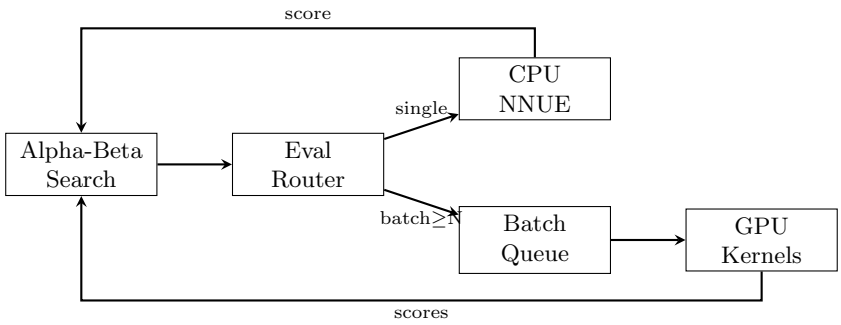

Fig. 1. Evaluation routing: single positions use CPU NNUE; batches $\geq N$ positions use GPU.

2. **Kernel dispatch**: `dispatchThreads()` or `dispatchThreadgroups()` records work
3. **Submission**: `commit()` submits to GPU queue
4. **Synchronization**: `waitUntilCompleted()` blocks until completion

On Apple Silicon, unified memory (`MTLResourceStorageModeShared`) allows CPU and GPU to access the same physical memory without explicit transfers.

3 System Architecture

3.1 Architecture Overview

Figure 1 shows the evaluation pipeline. The key insight is that GPU evaluation requires accumulating positions into batches before dispatch.

3.2 GPU Batch Evaluation

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

3.3 Metal Kernel Implementation

The feature transform kernel processes all positions in parallel:

```
kernel void feature_transform(
    device const int16_t* weights,
    device const int16_t* biases,
    device const int32_t* features,
    device const uint32_t* counts,
    device int32_t* accumulators,
    constant uint& hidden_dim,
```

Algorithm 1 GPU Batch NNUE Evaluation

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

```
// CPU: Prepare batch data
for i = 1 to N do
    Extract features from position i
    Store in unified memory buffers
end for
// GPU: Single command buffer
encoder ← CreateEncoder
// Kernel 1: Feature transform (all positions)
DispatchThreads(hidden_dim × N)
Barrier
// Kernel 2: Forward pass (all positions)
DispatchThreadgroups(N, threads=64)
SubmitAndWait(encoder)
return scores from output buffer
```

```
    constant uint& batch_size,
    uint2 gid [[thread_position_in_grid]])
{
  uint pos = gid.y;  // Position index
  uint h = gid.x;    // Hidden dimension
  if (pos >= batch_size || h >= hidden_dim)
    return;

  int32_t acc = biases[h];
  uint count = counts[pos];
  for (uint i = 0; i < count; i++) {
    int32_t f = features[pos * 32 + i];
    acc += weights[f * hidden_dim + h];
  }
  accumulators[pos * hidden_dim + h] = acc;
}
```

Listing 1.1. Feature Transform Kernel (simplified)

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)

Table 2. GPU Dispatch Overhead (Minimal Kernel, N=1,000)

Statistic	Latency (μs)
Mean	168.44
Std Dev	56.97
Median	150.42
P95	282.58
P99	382.58
Min	95.04
Max	945.96

Table 3. GPU Shader Throughput (Vector Addition)

Work Size (elements)	Throughput (GB/s)
1,024	0.05
16,384	0.82
262,144	11.98
1,048,576	44.14

4.2 Timing Methodology

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

- **Warmup**: 100 iterations discarded before measurement
- **Samples**: 1,000–10,000 iterations depending on benchmark
- **Statistics**: Median, P95, P99, min, max reported (not just mean)
- **GPU timing**: Blocking `waitUntilCompleted()` (synchronous)

Definition of "evaluation": For CPU, this includes accumulator refresh and forward pass. For GPU, this includes buffer writes, command buffer creation, kernel dispatch, and synchronization.

5 Results

5.1 GPU Dispatch Overhead

Table 2 shows the irreducible cost of GPU command buffer submission, measured with a minimal kernel that writes a single integer.

The median dispatch overhead of 150 μs represents the minimum cost for *any* GPU work, regardless of kernel complexity. This is the fundamental bottleneck.

5.2 GPU Compute Throughput

Table 3 shows GPU throughput when dispatch overhead is amortized over large workloads.

At 1M elements, the GPU achieves 44 GB/s—demonstrating that compute throughput is *not* the bottleneck. The low throughput at small sizes reflects dispatch overhead domination.

Table 4. True Batching Verification (16 positions)

Method	Median Latency (μs)	Speedup
Sequential (16×1)	2,407	1.0×
Batched (1×16)	1,337	1.8×

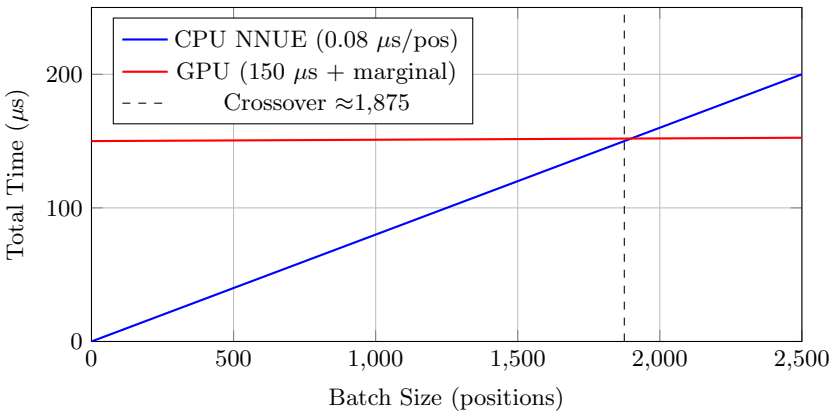

Fig. 2. CPU vs GPU evaluation time. Crossover occurs at ≈1,875 positions where dispatch overhead is amortized.

5.3 True Batching Verification

To verify our implementation uses genuine batching (single dispatch for N positions) rather than sequential dispatches, we compare:

- **Sequential**: 16 separate `evaluate_batch()` calls with batch size 1
- **Batched**: 1 `evaluate_batch()` call with batch size 16

The 1.8× speedup confirms true batching: a single command buffer processes all 16 positions. If batching were "fake" (sequential internal dispatches), we would see no speedup.

5.4 CPU vs GPU Crossover Analysis

Figure 2 shows the crossover analysis. CPU NNUE evaluation costs approximately 0.08 μs per position (from engine NPS of 1.38M). GPU evaluation has fixed dispatch overhead plus marginal per-position cost.

The crossover point is approximately:

$$\text{Crossover} = \frac{\text{GPU dispatch overhead}}{\text{CPU per-position cost}} = \frac{150\,\mu\text{s}}{0.08\,\mu\text{s}} \approx 1,875 \text{ positions}$$

Table 5. Search Benchmark Results

Metric	Value
Total Nodes	2,477,446
Total Time	1,792 ms
Nodes/Second	1,382,503
Depth Reached	13

Table 6. GPU Memory Allocation

Component	Size (KB)
Big network (feature transformer)	45,760
Big network (threat weights)	82,351
Big network (layers)	137
Small network (total)	6,361
Working buffers	2,156
Total	**108,240**

5.5 Search Performance

The engine achieves 1.38M nodes/second on the standard 50-position benchmark (depth 13), using CPU NNUE evaluation:

5.6 GPU Memory Usage

Table 6 shows GPU memory allocation for NNUE networks.

6 Discussion

6.1 Why Dispatch Overhead is Irreducible

The 150 μs dispatch overhead reflects Metal's command buffer lifecycle:

1. **Buffer allocation**: `commandBuffer()` allocates GPU-side resources
2. **Encoder setup**: Compute encoder state machine initialization
3. **Queue submission**: Inter-process communication with GPU driver
4. **Synchronization**: Kernel completion signaling

This overhead is architectural, not implementation-dependent. Apple's Metal Best Practices Guide [7] recommends minimizing command buffer submissions, acknowledging this cost.

6.2 Implications for Alpha-Beta Search

Alpha-beta search evaluates positions sequentially, with each evaluation determining whether to prune subsequent branches. This creates fundamental incompatibility with GPU batching:

- **No natural batches**: Positions are evaluated one at a time
- **Data-dependent pruning**: Cannot predict which positions will be evaluated
- **Latency-bound**: Search speed depends on single-position evaluation latency

To benefit from GPU, alpha-beta would need to accumulate ≈1,875 positions before dispatching—but this would require speculative evaluation of positions that may be pruned, wasting compute.

6.3 When GPU Acceleration Helps

GPU acceleration benefits batch-oriented workloads:

- **MCTS**: Naturally generates batches of leaf evaluations [2]
- **Database analysis**: Evaluating thousands of positions
- **Training**: Batch gradient computation
- **Multi-position analysis**: Simultaneous analysis of multiple games

6.4 Limitations

- Single hardware configuration (M2 Max)
- Synchronous GPU timing (asynchronous overlap not explored)
- No integration of GPU batching into search (would require speculative evaluation)
- 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 that neural network evaluation can replace handcrafted evaluation when combined with batch-oriented search.

For alpha-beta search, Rocki and Suda [4] explored GPU parallelization through parallel subtree evaluation, finding communication overhead limited speedup. Our work extends this analysis to unified memory hardware, identifying dispatch overhead as the specific bottleneck.

Apple's Metal documentation [6, 7] provides guidance on minimizing command buffer overhead through techniques like indirect command buffers (ICBs), which could reduce dispatch costs for future work.

8 Conclusion

We investigated GPU-accelerated NNUE evaluation on Apple Silicon, identifying dispatch overhead—not memory bandwidth or compute throughput—as the fundamental bottleneck. Key findings:

1. **Dispatch overhead**: 150 μs median per command buffer, irreducible
2. **Compute throughput**: 44 GB/s when amortized, demonstrating GPU capability
3. **True batching**: Confirmed 1.8× speedup from single-dispatch batching
4. **Crossover**: ≈1,875 positions needed for GPU throughput parity
5. **Search performance**: 1.38M nodes/second with CPU NNUE

GPU acceleration for alpha-beta chess engines requires either: (1) batch-oriented search algorithms like MCTS, or (2) speculative evaluation strategies that accept wasted compute. Single-position evaluation in traditional alpha-beta remains CPU-bound due to irreducible dispatch latency.

Reproducibility

Hardware: Apple M2 Max, 64GB unified memory. **Software**: macOS 14.0, Xcode 15.0. **Build**: CMake, -O3, LTO. **Source**: https://github.com/NripeshN/MetalFish. **Networks**: nn-c288c895ea92.nnue, nn-37f18f62d772.nnue from Stockfish.

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)