

# MetalFish: Practical Limits of GPU-Accelerated NNUE Evaluation on Apple Silicon

Nripesh Niketan<sup>1</sup>

Independent Researcher  
nripesh14@gmail.com

**Abstract.** We present MetalFish, a GPU-accelerated chess engine exploring the practical limits of NNUE evaluation on Apple Silicon’s unified memory architecture. Through systematic benchmarking on M2 Max, we demonstrate that synchronous GPU dispatch overhead (148  $\mu$ s median) dominates single-position latency (253  $\mu$ s), making GPU evaluation approximately  $337\times$  slower than CPU NNUE (0.75  $\mu$ s at 1.34M NPS) for alpha-beta search. However, batch evaluation amortizes this overhead effectively: per-position cost drops to 0.3  $\mu$ s at batch size 4096, with true single-dispatch batching achieving  $567\times$  speedup over sequential dispatches. We provide stage-by-stage latency decomposition showing GPU dispatch accounts for  $>99\%$  of single-position time, implement adaptive kernel selection with dual-perspective feature transformation using 8-way loop unrolling, and verify 100% evaluation consistency across 1,000 positions. GPU acceleration remains promising for batch-oriented workloads (MCTS, database analysis) but is unsuitable for traditional alpha-beta search without speculative evaluation or asynchronous queuing.

**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. The critical path in alpha-beta search is *latency*, not throughput: each position must be evaluated before pruning decisions can be made. In contrast, Monte Carlo Tree Search (MCTS) is *throughput*-oriented, naturally batching leaf evaluations.

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 systematically measures where time is spent in GPU NNUE evaluation, revealing that command buffer dispatch—not compute—dominates single-position latency.

### 1.1 Research Question

What are the practical limits of GPU-accelerated NNUE evaluation on Apple Silicon, and where exactly does the time go?

**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             |

## 1.2 Contributions

1. **Quantified CPU vs GPU NNUE comparison:** CPU NNUE achieves 1.34M positions/second ( $0.75 \mu\text{s}/\text{pos}$ ); GPU single-position blocking latency is  $253 \mu\text{s}$  ( $337\times$  slower).
2. **Stage-by-stage latency decomposition:** We show GPU dispatch and synchronization account for  $>99\%$  of single-position time, with CPU feature extraction negligible ( $0.04 \mu\text{s}$ ).
3. **Batch scaling analysis:** Per-position cost drops from  $253 \mu\text{s}$  ( $N=1$ ) to  $0.3 \mu\text{s}$  ( $N=4096$ ), with dispatch overhead amortized across positions.
4. **True batching verification:** Single command buffer with two dispatches achieves  $567\times$  speedup over  $N$  sequential command buffers at  $N=1024$ .
5. **GPU evaluation consistency:** 100% reproducibility across 1,000 positions with non-zero scores.

## 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. We support 64 features per perspective; in our 2,048-position dataset, maximum observed was 30 per perspective (60 total).

### 2.2 Alpha-Beta vs MCTS

Alpha-beta search [8] evaluates positions sequentially with data-dependent pruning. Each evaluation must complete before the next pruning decision. This makes *latency* the critical metric.

MCTS [3] accumulates positions at leaf nodes before evaluation, naturally forming batches. This makes *throughput* the critical metric, where GPU acceleration excels.

**Table 2.** GPU Configuration Constants

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

### 2.3 Metal Compute Model

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

- **Unified memory:** CPU and GPU share physical memory, eliminating explicit transfers
- **Command buffer lifecycle:** Allocation  $\rightarrow$  encoding  $\rightarrow$  commit  $\rightarrow$  wait-UntilCompleted
- **Dispatch overhead:** Each command buffer submission incurs fixed overhead regardless of kernel complexity

## 3 System Architecture

### 3.1 GPU Configuration

Table 2 shows the key GPU configuration parameters.

### 3.2 Adaptive Kernel Selection

We implement strategy-based kernel selection:

```

1 enum class EvalStrategy {
2     CPU_FALLBACK,    // batch < 4
3     GPU_STANDARD,    // batch < 64
4     GPU_SIMD         // batch >= 64
5 };

```

**Listing 1.1.** Evaluation strategy selection

For batches  $\geq 64$ , we use dual-perspective kernels that process both white and black perspectives in a single 3D dispatch. The feature transform kernel uses 8-way loop unrolling for maximum instruction-level parallelism, with feature index bounds checking removed since CPU extraction guarantees valid indices.

**Algorithm 1** GPU Batch NNUe Evaluation**Require:** Batch of  $N$  positions**Ensure:** Evaluation scores for all positions

---

```

1:  $strategy \leftarrow \text{SELECTSTRATEGY}(N)$ 
2: if  $strategy = \text{CPU\_FALLBACK}$  then
3:   return CPU evaluation
4: end if
5: // CPU: Extract features to unified memory
6: for each position do
7:   Write features directly to GPU buffers
8: end for
9: // GPU: Single command buffer
10:  $\text{DISPATCHTHREADS}(hidden\_dim, 2, N)$  ▷ Feature transform
11:  $\text{BARRIER}$ 
12:  $\text{DISPATCHTHREADGROUPS}(N, \text{threads}=64)$  ▷ Forward pass
13:  $\text{SUBMITANDWAIT}$  ▷ Blocking sync
14: return scores from output buffer

```

---

**3.3 Command Buffer Optimization**

We minimize dispatch overhead through several techniques:

- **Unretained references:** Use `commandBufferWithUnretainedReferences` to avoid Objective-C retain/release overhead
- **Hazard tracking disabled:** Use `MTLResourceHazardTrackingModeUntracked` for unified memory buffers with explicit barriers
- **Pre-allocated buffers:** All working buffers allocated at initialization to avoid per-dispatch allocation

**3.4 Zero-Copy Buffer Management**

We pre-allocate all working buffers at initialization, writing features directly to unified memory:

```

1 // Write directly to unified memory (zero-copy)
2 int32_t* features_ptr =
3     static_cast<int32_t*>(features_buffer_ ->data());
4 for (int i = 0; i < batch_size; i++) {
5     features_ptr[offset++] = feature_index;
6 }

```

**Listing 1.2.** Direct buffer writes**3.5 Batch Evaluation Pipeline**

Algorithm 1 shows the 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 big), nn-37f18f62d772.nnue (6MB small)

### 4.2 Benchmark Dataset

Our benchmark uses 32 unique FEN positions representing diverse game phases:

- 4 opening positions (32 pieces)
- 10 middlegame positions (28–32 pieces)
- 4 tactical positions (complex piece interactions)
- 14 endgame positions (2–20 pieces)

These are cycled to create 2,048 test positions. Of these, 1,984 positions have no king in check (valid for NNUE evaluation).

### 4.3 Timing Methodology

- **Timer:** `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)

### 4.4 CPU NNUE Baseline

CPU NNUE performance was measured using the engine’s standard `bench` command, which runs depth-limited searches on 50 diverse positions:

- **Nodes searched:** 2,477,446
- **Total time:** 1,846 ms
- **NPS:** 1,342,061 nodes/second
- **Per-position latency:**  $\approx 0.75 \mu s$

This represents full NNUE evaluation including accumulator updates and forward pass, providing a matched-scope baseline for GPU comparison.

**Table 3.** CPU Evaluation Baselines

| Metric         | Simple Eval NNUE (bench) |              |
|----------------|--------------------------|--------------|
| Median latency | 0.00 $\mu$ s             | 0.75 $\mu$ s |
| Throughput     | >10M/s                   | 1.34M/s      |

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

| Statistic Latency ( $\mu$ s) |     |
|------------------------------|-----|
| Median                       | 148 |
| P95                          | 260 |
| P99                          | 316 |

## 5 Results

### 5.1 CPU Baseline Measurements

Table 3 shows CPU evaluation performance.

CPU NNUE evaluation achieves 1.34 million positions per second, or approximately 0.75  $\mu$ s per position. This is the baseline against which GPU evaluation must be compared.

### 5.2 GPU Dispatch Overhead

Table 4 shows minimal-kernel dispatch overhead.

The 148  $\mu$ s median dispatch overhead represents the *irreducible minimum* cost for any GPU operation in synchronous blocking mode. This alone is 197 $\times$  slower than CPU NNUE evaluation.

### 5.3 GPU Stage Breakdown

Table 5 decomposes end-to-end GPU latency into stages.

**Key finding:** GPU dispatch and synchronization 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.

**Key findings:**

1. Latency is approximately constant (247–341  $\mu$ s) for batch sizes 1–256, confirming dispatch dominance.
2. Per-position cost drops from 253  $\mu$ s (N=1) to 0.3  $\mu$ s (N=4096).
3. GPU becomes throughput-competitive with CPU NNUE (0.75  $\mu$ s) only at batch sizes  $\geq 512$ .

**Table 5.** GPU Stage Breakdown (median, N=100 iterations)

| Batch Size | CPU Prep ( $\mu s$ ) | GPU Eval ( $\mu s$ ) | Total ( $\mu s$ ) | GPU % |
|------------|----------------------|----------------------|-------------------|-------|
| 1          | 0.3                  | 258                  | 258               | 99.9% |
| 8          | 0.3                  | 258                  | 258               | 99.9% |
| 512        | 5.8                  | 362                  | 368               | 98.4% |

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

| Batch Size | Median ( $\mu s$ ) | P95 ( $\mu s$ ) | P99 ( $\mu s$ ) | Per-Pos ( $\mu s$ ) |
|------------|--------------------|-----------------|-----------------|---------------------|
| 1          | 253                | 363             | 395             | 253.0               |
| 8          | 247                | 347             | 470             | 30.8                |
| 64         | 279                | 538             | 662             | 4.4                 |
| 256        | 341                | 446             | 497             | 1.3                 |
| 512        | 347                | 462             | 485             | 0.7                 |
| 1024       | 500                | 596             | 626             | 0.5                 |
| 2048       | 824                | 937             | 960             | 0.4                 |
| 4096       | 1,359              | 1,467           | 1,607           | 0.3                 |

## 5.5 True Batching Verification

Table 7 compares sequential vs batched dispatches.

Speedups scale approximately linearly with batch size because each sequential dispatch incurs the full 146  $\mu s$  overhead, while batching amortizes this cost.

## 5.6 GPU Evaluation Consistency

Table 8 verifies GPU evaluation reproducibility.

GPU evaluation produces consistent, non-zero scores across repeated runs. The score range indicates meaningful differentiation between positions.

# 6 Discussion

## 6.1 Why GPU is Slower for Single Positions

The fundamental bottleneck is command buffer dispatch overhead, not compute:

- Minimal kernel dispatch: 148  $\mu s$
- NNUE kernel dispatch: 253  $\mu s$
- CPU NNUE evaluation: 0.75  $\mu s$

Even with zero kernel execution time, GPU would still be  $197\times$  slower than CPU for single positions due to dispatch overhead alone.

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

|      | N Sequential<br>(N×1 CB) | Batched Speedup<br>(1×1 CB) |        |
|------|--------------------------|-----------------------------|--------|
| 16   | 4,708 $\mu$ s            | 283 $\mu$ s                 | 16.6×  |
| 64   | 18,573 $\mu$ s           | 291 $\mu$ s                 | 63.8×  |
| 256  | 73,986 $\mu$ s           | 300 $\mu$ s                 | 246.3× |
| 1024 | 292,860 $\mu$ s          | 516 $\mu$ s                 | 567.2× |

**Table 8.** GPU Evaluation Consistency (1,000 positions)

| Metric                 | Value       |
|------------------------|-------------|
| Non-zero GPU scores    | 100%        |
| Consistent across runs | 100%        |
| Mean  score            | 222         |
| Score range            | [-407, 258] |

## 6.2 When GPU Evaluation Helps

GPU batch evaluation becomes throughput-competitive at  $N \geq 512$  ( $0.7 \mu$ s/position vs CPU’s  $0.75 \mu$ s). This enables:

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

## 6.3 Alpha-Beta Limitations

Single-position GPU blocking latency ( $253 \mu$ s) makes GPU evaluation unsuitable for alpha-beta search in synchronous blocking mode. Alpha-beta’s sequential, data-dependent pruning prevents effective batching without significant architectural changes such as:

- Speculative evaluation of multiple branches
- Asynchronous queuing with CPU fallback
- Lazy evaluation with deferred GPU dispatch

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

## 6.4 Threats to Validity

- **Synchronous timing:** Our measurements use blocking `waitUntilCompleted()`. Asynchronous dispatch with completion handlers could reduce apparent latency by overlapping CPU work.



- **Command buffer reuse:** We create new command buffers per evaluation. Reusing command buffers could reduce allocation overhead.
- **Dataset size:** Our 32 unique positions may not represent all game phases equally.

## 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 quantified bottleneck analysis.

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 that quantifies the practical limits of GPU NNUE evaluation on Apple Silicon:

1. **CPU NNUE baseline:** 1.34M positions/second ( $0.75\ \mu\text{s}/\text{pos}$ )
2. **GPU single-position:**  $253\ \mu\text{s}$  median ( $337\times$  slower than CPU)
3. **GPU dispatch overhead:**  $148\ \mu\text{s}$  irreducible minimum
4. **GPU batch (N=4096):**  $0.3\ \mu\text{s}/\text{pos}$  (throughput-competitive)
5. **True batching:**  $567\times$  speedup at N=1024
6. **Consistency:** 100% reproducibility across 1,000 positions

**Key insight:** GPU dispatch overhead, not compute, is the bottleneck. Single-position GPU evaluation is unsuitable for alpha-beta search, but batch evaluation is effective for MCTS, database analysis, and training data generation.

### 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)