MetalFish: A GPU-Accelerated Chess Engine for Apple Silicon Unified Memory Architecture

Nripesh Niketan[1]

Independent Researcher
nripesh14@gmail.com

Abstract. We present MetalFish, a GPU-accelerated UCI chess engine implementing Stockfish-style alpha-beta search with Apple Metal GPU acceleration on unified memory architecture. MetalFish employs a hybrid CPU-GPU design where the CPU executes traditional search algorithms while the GPU accelerates NNUE neural network evaluation through Metal compute shaders. The implementation leverages Apple Silicon's unified memory via `MTLResourceStorageModeShared` to achieve zero-copy data sharing between CPU search routines and GPU evaluation kernels. We present microbenchmark results showing CPU single-position evaluation at 0.09 μs compared to 783 μs for GPU dispatch, demonstrating the overhead that motivates our hybrid approach. The engine achieves 1.4 million nodes per second on standard benchmark positions and correctly computes perft values for all standard test positions. This work provides empirical insights into the challenges and trade-offs of hybrid CPU-GPU chess engine design on unified memory architectures.

Keywords: Chess Engine, GPU Computing, Metal, NNUE, Unified Memory, Apple Silicon

1 Introduction

Modern chess engines achieve remarkable playing strength through sophisticated alpha-beta search enhanced with neural network evaluation. Stockfish combines Principal Variation Search (PVS) with Efficiently Updatable Neural Networks (NNUE) to achieve superhuman performance [1]. The emergence of GPU computing presents opportunities to accelerate evaluation-intensive components, but the sequential dependencies inherent in alpha-beta search create fundamental challenges for GPU parallelization [2].

MetalFish addresses these challenges through a hybrid CPU-GPU architecture leveraging Apple Silicon's unified memory. Rather than attempting full GPU parallelization of alpha-beta search, we maintain traditional CPU-based search while offloading neural network evaluation to the GPU. The unified memory architecture eliminates host-device data transfer overhead, enabling efficient hybrid execution.

1.1 Contributions

This paper makes the following contributions:

1. A unified-memory-friendly design for NNUE evaluation offload that avoids explicit host-device copies on Apple Silicon, using Metal's `MTLResourceStorageModeShared` buffer allocation.
2. Metal compute kernels for (a) sparse feature transformation and (b) incremental accumulator updates suitable for NNUE-style inference during tree search.
3. An empirical characterization of kernel dispatch overhead versus batch size on M-series GPUs, with measurements showing the crossover point where GPU execution becomes beneficial.
4. A complete, tested implementation achieving 1.4M nodes/second with correct perft results for all standard test positions.

2 Background

2.1 Alpha-Beta Search

The minimax algorithm with alpha-beta pruning forms the foundation of modern chess engines [3]. Alpha-beta maintains bounds $[\alpha, \beta]$ representing the range of possible values; when $\alpha \geq \beta$, remaining siblings can be pruned. Principal Variation Search (PVS) refines this by assuming the first move is optimal, searching subsequent moves with zero-width windows [4].

2.2 NNUE Evaluation

Efficiently Updatable Neural Networks (NNUE) use a large sparse input layer representing piece-square combinations, followed by smaller dense layers [5]. The key insight is that activations can be updated incrementally as moves are made, rather than recomputing from scratch. Stockfish's NNUE architecture uses:

- Feature transformer: 45,056 inputs $\rightarrow$ 1,024 hidden units
- FC0: 2,048 $\rightarrow$ 16 (concatenated perspectives)
- FC1: 16 $\rightarrow$ 32 with squared clipped ReLU
- FC2: 32 $\rightarrow$ 1 output score

The quantization uses 6-bit right shifts for weight scaling:

$$\text{clipped\_relu}(x) = \min(\max(x \gg 6, 0), 127) \tag{1}$$

2.3 Unified Memory Architecture

Apple Silicon's unified memory allows CPU and GPU to access the same physical memory coherently [6]. Using `MTLResourceStorageModeShared`, buffers are accessible from both processors without explicit copying, eliminating the traditional discrete GPU bottleneck.

3 System Architecture

3.1 Design Overview

MetalFish implements three primary components:

CPU Search Engine: Executes complete alpha-beta search including PVS, move ordering via history heuristics, and pruning techniques. All control flow remains on CPU to avoid GPU branch divergence.

GPU Evaluation Engine: Implements NNUE inference through Metal compute shaders for feature transformation and network forward passes.

Unified Memory Interface: Manages shared buffers using `MTLResourceStorageModeShared`, enabling zero-copy access from both CPU and GPU.

3.2 Search Implementation

The search implements Stockfish-style techniques:

Move Ordering: Butterfly history ([2][4096], init: 68), capture history ([16][64][8], init: −689), continuation history ([16][64][16][64], init: −529), killer moves, and counter moves.

Extensions: Singular extension with double/triple variants for strongly singular moves; check extension for positions in check.

Pruning: Null move pruning with verification search, late move reductions using $R = \lfloor 2747/128 \times \ln(\text{moveNumber}) \rfloor$, futility pruning, SEE pruning, and ProbCut.

Transposition Table: Zobrist hashing [7] with depth-preferred replacement and generation aging.

3.3 GPU NNUE Kernels

Feature Transformation The feature transformer converts sparse HalfKAv2 features to dense accumulators. Each thread computes one output element:

```
kernel void feature_transform(
    device const int16_t* weights,
    device const int16_t* biases,
    device const int* features,
    device int32_t* accumulator,
    constant int& num_features,
    constant int& ft_dims,
    uint h [[thread_position_in_grid]])
{
    if (h >= ft_dims) return;
    int32_t sum = biases[h];
    for (int i = 0; i < num_features; i++) {
        int f = features[i];
        sum += weights[f * ft_dims + h];
    }
    accumulator[h] = sum;
```

```
}
```

Listing 1.1. Feature transformation kernel

Kernel configuration: 1,024 threads dispatched as a single threadgroup, processing all hidden units in parallel. Memory layout stores weights in feature-major order for coalesced access when multiple threads read the same feature.

Incremental Updates When a move is made, only changed features require updating:

```
kernel void feature_update(
    device const int16_t* weights,
    device int32_t* accumulator,
    device const int* added,
    device const int* removed,
    constant int& num_added,
    constant int& num_removed,
    constant int& ft_dims,
    uint h [[thread_position_in_grid]])
{
    if (h >= ft_dims) return;
    int32_t delta = 0;
    for (int i = 0; i < num_added; i++)
        delta += weights[added[i] * ft_dims + h];
    for (int i = 0; i < num_removed; i++)
        delta -= weights[removed[i] * ft_dims + h];
    accumulator[h] += delta;
}
```

Listing 1.2. Incremental accumulator update

3.4 Unified Memory Buffer Allocation

Buffer allocation uses shared storage mode:

```
MTL::ResourceOptions opts =
    MTL::ResourceStorageModeShared;
ft_weights = device->newBuffer(
    FT_IN_DIMS * FT_OUT_DIMS * sizeof(int16_t),
    opts);
// CPU access: ft_weights->contents()
// GPU access: direct in kernel
```

Listing 1.3. Shared buffer allocation

This enables the CPU to write position features and read evaluation results without explicit memory transfers.

Table 1. Single-position evaluation latency (microseconds)

Method	Mean	Std Dev	Min	Max
CPU Eval	0.09	0.08	0.00	5.08
GPU Eval	782.74	143.93	582.58	5172.04

Table 2. GPU batch evaluation performance

Batch Size	Time (μs)	Per-Position (μs)	Throughput (pos/s)
1	0.05	0.05	19,669,551
2	0.04	0.02	47,483,380
4	0.07	0.02	57,151,021
8	0.11	0.01	75,294,117
16	0.16	0.01	102,960,102
32	0.24	0.01	135,083,794
64	0.40	0.01	158,517,858

4 Experimental Results

All experiments conducted on Apple M2 Max (12-core CPU, 38-core GPU, 64GB unified memory) running macOS 14.0, Metal feature set macOS-GPUFamily2-v1.

4.1 Microbenchmarks: CPU vs GPU Evaluation

Table 1 shows evaluation latency for single positions, measured over 1,000 iterations after 100 warmup iterations.

The GPU evaluation is approximately 8,700× slower for single positions due to kernel dispatch overhead. This motivates our hybrid approach where single-position evaluation remains on CPU.

4.2 Batch Evaluation Throughput

Table 2 shows GPU batch evaluation performance, demonstrating throughput scaling with batch size.

Per-position cost decreases from 0.05 μs at batch size 1 to 0.006 μs at batch size 64, demonstrating effective amortization of dispatch overhead.

4.3 Search Performance

The engine achieves the following search performance on the standard 50-position benchmark suite at depth 12:

Table 3. Search benchmark results (depth 12, 64MB hash)

Metric	Value
Total Nodes	2,470,322
Total Time	1,761 ms
Nodes/Second	1,402,795

Table 4. Perft verification (starting position)

Depth	Nodes
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324

4.4 Move Generation Verification

Table 4 shows perft results matching established correct values.

Additional tests verify Kiwipete (depth 5: 193,690,690), en passant, castling, and promotion positions.

4.5 Hardware Characteristics

5 Discussion

5.1 Kernel Dispatch Overhead

The 783 μs GPU evaluation latency versus 0.09 μs CPU latency demonstrates that Metal command buffer creation, encoding, commit, and synchronization dominate single-position workloads. This overhead is inherent to the GPU programming model and motivates keeping single-position evaluation on CPU during search.

5.2 Batching Strategy

For batch evaluation (e.g., multi-PV analysis or position database evaluation), GPU acceleration provides substantial throughput gains. The crossover point where GPU matches CPU per-position cost occurs around batch size 8,700 based on our measurements, though total throughput benefits appear much earlier due to parallelism.

Table 5. Metal backend properties (M2 Max)

Property	Value
Unified Memory	Enabled
Max Buffer Size	19,169 MB
Max Threadgroup Memory	32,768 bytes
Max Threads/Threadgroup	1,024

5.3 Limitations

- GPU acceleration limited to evaluation; search remains CPU-bound
- Single-position GPU evaluation impractical due to dispatch overhead
- Uses pre-trained networks; training not implemented
- CPU and GPU evaluations may differ slightly due to quantization paths

6 Related Work

Leela Chess Zero [8] pioneered GPU-accelerated MCTS for chess. Fat Fritz combined Stockfish search with GPU evaluation. Our work specifically targets unified memory architectures where the CPU-GPU data sharing model differs fundamentally from discrete GPUs.

7 Conclusion

MetalFish demonstrates practical hybrid CPU-GPU chess engine design on unified memory architectures. Key findings:

1. GPU kernel dispatch overhead (783 μs) exceeds CPU evaluation time (0.09 μs) by 8,700$\times$ for single positions
2. Batch processing amortizes overhead, achieving 158M positions/second at batch size 64
3. Unified memory via `MTLResourceStorageModeShared` enables zero-copy CPU-GPU cooperation
4. Hybrid approach achieves 1.4M nodes/second with correct perft results

Reproducibility

Hardware: Apple M2 Max, 64GB unified memory. Software: macOS 14.0, Xcode 15.0, Metal feature set macOS-GPUFamily2-v1. Source code available at https://github.com/NripeshN/MetalFish. NNUE networks: nn-c288c895ea92.nnue (125MB), nn-37f18f62d772.nnue (6MB).

Acknowledgments

Thanks to the Stockfish and Leela Chess Zero teams for open-source contributions informing this work.

References

1. Stockfish Developers: Stockfish 16 NNUE documentation. https://github.com/official-stockfish/Stockfish (2024)
2. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (2008)
3. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)
4. Reinefeld, A.: An improvement to the scout tree-search algorithm. ICCA Journal 6(4), 4–14 (1983)
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 Best Practices Guide: Resource Storage Modes. https://developer.apple.com/documentation/metal/resource_fundamentals/setting_resource_storage_modes (2024)
7. Zobrist, A.L.: A new hashing method with application for game playing. Tech. Rep. 88, Computer Sciences Department, University of Wisconsin (1970)
8. Leela Chess Zero: Neural network based chess engine. https://lczero.org/ (2024)
9. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815 (2017)
10. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep Blue. Artificial Intelligence 134(1-2), 57–83 (2002)