Optimizing Softmax through Vectorization

Vincenzo Gargano, MAT: 667591

March 11, 2025

1 Introduction

The softmax function is a fundamental component in machine learning systems, used to convert logits into probability distributions. This project investigates two optimization approaches for softmax computation: manual vectorization using AVX intrinsics and auto-vectorization by compiler with OpenMP. We are going to evaluate their performance characteristics by watching at the speedup achieved over a scalar baseline implementation.

2 Implementations

2.1 Auto-Vectorized Version

The baseline implementation was modified to enable compiler optimizations:

- Loop restructuring for vectorization-friendly patterns
- OpenMP SIMD pragmas for explicit vectorization hints
- Compiler flags: `-O3 -ffast-math -march=native`

2.2 Manual AVX Vectorization

The AVX implementation features:

- 256-bit vector operations using AV2/FMA instructions
- Memory alignment handling with `__mm256_loadu_ps`
- Horizontal reductions for max/sum calculations
- Remainder handling for non-multiples of 8 elements
- Numerical stability through max subtraction

Listing 1: Key AVX Reduction Snippet

```
// Horizontal max reduction
alignas(32) float max_buffer[8];
_mm256_store_ps(max_buffer, max_v);
for (size_t j = 0; j < 8; ++j)
    max_val = std::max(max_val, max_buffer[j]);
```

Version	Time (ms)	Speedup
Baseline (Scalar)	12.4	1.0x
Auto-Vectorized	4.2	3.0x
Manual AVX	1.8	6.9x

Key observations:

- AVX version shows 6.9x speedup over scalar baseline
- Alignment issues caused initial 40% performance penalty
- Remainder handling adds ¡5% overhead for large K

3 Trade-offs Analysis

3.1 Manual Vectorization

- **Pros**: Full control over vector operations, optimal memory access patterns
- **Cons**: Platform-specific, complex debugging, alignment sensitivity

3.2 Auto-Vectorization

- **Pros**: Portable, maintainable, compiler-optimized
- **Cons**: Limited by compiler heuristics, less predictable

4 Challenges & Improvements

4.1 Challenges Faced

- Segmentation faults from misaligned memory accesses
- Precision differences between `exp256_ps` and `expf`
- Register pressure in horizontal reductions

4.2 Future Improvements

- AVX-512 for 16-wide vectorization
- Blocked processing for better cache utilization
- Hybrid approach combining auto/manual vectorization
- Asynchronous prefetching for large inputs

5 Conclusion

The manual AVX implementation demonstrated 6.9x speedup over scalar code, while the auto-vectorized version achieved 3x improvement. The choice between approaches depends on target architecture and maintainability requirements. Proper memory alignment proved critical for AVX performance, accounting for up to 40% of runtime differences.