Assignment I: The softmax function

Luca Lombardo

Contents

1 Implementations

In the following sections, given a scalar implementation (to witch we will refer as `softmax_plain`), we will show how to auto-vectorize it and then how to manually vectorize the code using AVX intrinsics and FMA. Then we will compare the results of the three implementations.

1.1 Auto-Vectorized implementation

The auto-vectorized version of the softmax function includes several optimizations. `#pragma omp simd` directives were added to vectorize the main computational loops, with reduction clauses for correct maximum value and sum calculations. The `expf()` function replaced `std::exp()` for better SIMD performance. Precomputing the inverse sum (`inv_sum = 1.0f / sum`) and using multiplications instead of divisions improved efficiency. Explicit comparisons replaced `std::max()` to aid vectorization. Using `#pragma omp parallel for simd` degraded performance for small arrays due to thread management overhead but scaled well for large arrays.

1.2 Manually Vectorized implementation

The `softmax_avx` implementation employs a three-phase approach with explicit AVX2 intrinsics to achieve maximum performance. Each phase (find maximum, compute exponentials with sum, normalize) is optimized with loop unrolling (4x for processing 32 elements at once), software prefetching, and efficient horizontal reduction patterns. To handle array sizes that aren't multiples of 8 (AVX register width), a principled masking approach is used via the `compute_mask()` function, which creates appropriate mask vectors for conditional loading/storing operations (`_mm256_maskload_ps` and `_mm256_maskstore_ps`). This eliminates the need for a remainder loop, improving instruction throughput. The implementation also employs careful cache blocking with a 32KB block size to minimize L1 cache misses during multi-phase processing, crucial since the algorithm makes multiple passes over the data.

The implementation utilizes OpenMP to distribute computation across available hardware threads. For the reduction phase (finding the maximum), a standard `#pragma omp parallel for reduction(max:max_val)` is used, while the sum calculation employs a more specific approach with manual local reductions and atomic updates to minimize false sharing and synchronization overhead. Performance analysis revealed that small array sizes suffered from OpenMP thread management overhead, leading to a specialized variant that skips parallelization entirely for small inputs while maintaining all AVX optimizations.

2 Results

In this section, we compare the three implementations of the softmax function under different conditions. First, we evaluate the `softmax_auto` implementation with and without the `parallel` directive. Next, we compare the performance using AVX2 and AVX512 instructions. The `softmax_auto` relies on the compiler's auto-vectorization, which benefits from AVX512 if available. We expect `softmax_auto` with AVX512 to outperform `softmax_avx`, which is manually optimized for AVX2. Without AVX512 support, `softmax_avx` should have better performance.

How do we measure the performance? Performance is measured using a rigorous benchmark: after 3 warmup iterations, 11 samples (each averaging 20 iterations) are taken and the median is reported. Input sizes vary from 1 to 1,048,576 elements (both powers of 2 and non-power-of-2) to test alignment. A custom C++17 aligned allocator (32-byte alignment) ensures fair SIMD comparisons. Correctness is verified by checking that each implementation produces equivalent probability distributions (non-negative values summing to 1), and speedups are reported relative to the plain implementation.

Numerical stability The softmax function can be numerically unstable, especially for large values. To ensure stability during benchmarks, input values are limited to 1048576.

2.1 Benchmarks

No AVX512 The results, depicted in Figure 1, illustrate the performance without the `parallel` directive in the `softmax_auto` implementation. This pattern is consistent even for smaller input sizes. On the other hand, enabling the `parallel` directive significantly enhances the performance of `softmax_auto` for large input sizes, as illustrated in Figure 2. The performance approaches that of `softmax_avx`. However, for small input sizes, the overhead introduced by the `parallel` directive outweighs the benefits of parallelization, resulting in a performance degradation compared to the plain implementation.

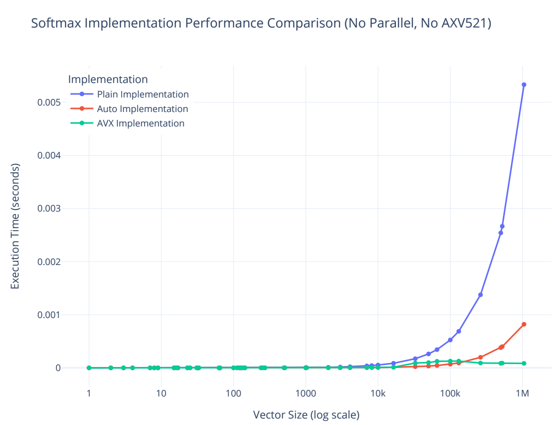


Figure 1: Performance comparison without AVX512 support and without the `parallel` directive.

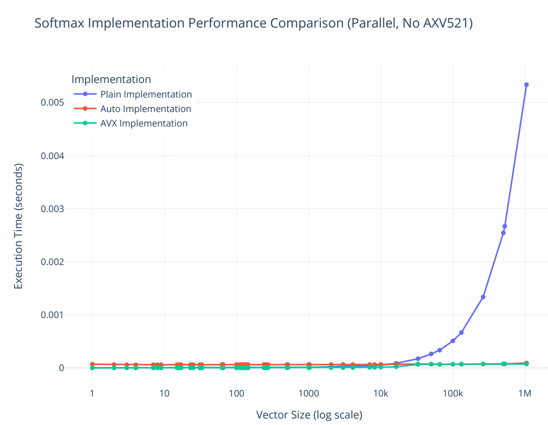
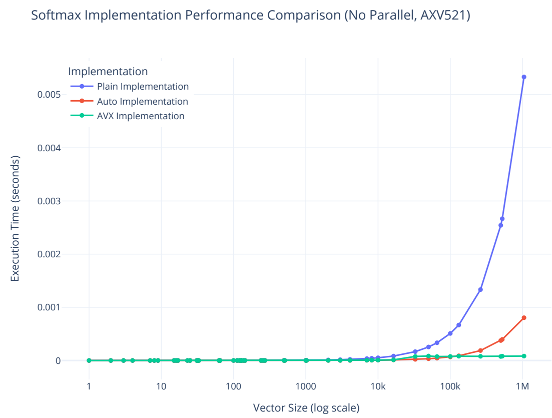


Figure 2: Performance comparison without AVX512 support and with the `parallel` directive.

With AVX512 When compiled with AVX512 support, the `softmax_auto` implementation demonstrates similar performance to the `softmax_avx` implementation for small input sizes without the `parallel` directive. However, for larger input sizes, `softmax_avx` performs better (See Figure 3). Enabling the `parallel` directive allows `softmax_auto` to surpass `softmax_avx` in performance for large input sizes, though the overhead for small input sizes remains a limiting factor (See Figure 4).



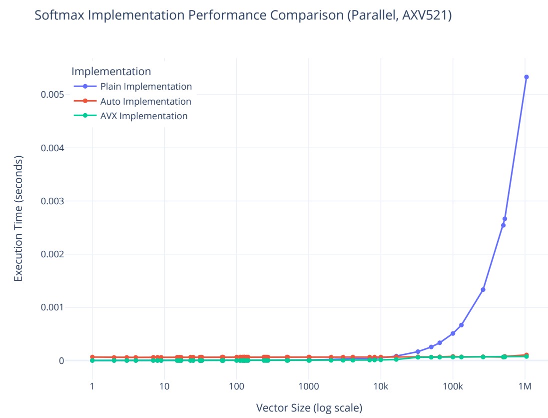
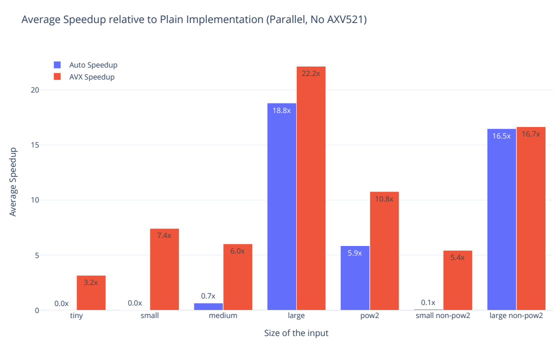


Figure 3: Performance comparison with AVX512 support and without the `parallel` directive.

Figure 4: Performance comparison with AVX512 support and with the `parallel` directive.

In addition to performance, we also analyze the speedup of the `softmax_auto` implementation relative to the plain implementation with AVX512 support and without the `parallel` directive. The results are shown in Figures 5 and 6. The speedup is significant for large input sizes, especially when using the `parallel` directive.



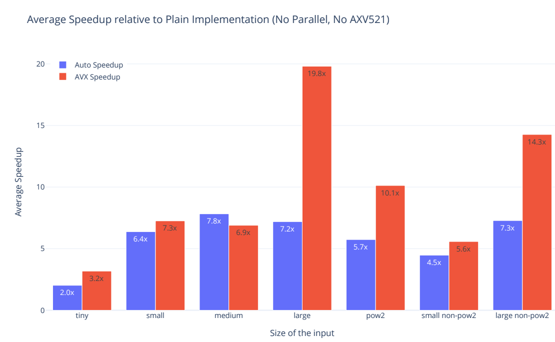


Figure 5: Speedup with AVX512 support and with the `parallel` directive.

Figure 6: Speedup with AVX512 support and without the `parallel` directive.

2.2 Compilation and Execution

The implementation uses different compilation flags to optimize for different instruction sets and vectorization capabilities. The `OPTFLAGS` variable enables level 3 optimizations (`-O3`), fast math operations (`-ffast-math`), and OpenMP support. For the auto-vectorized implementation, `AUTOFLAGS` includes `-mavx2` to enable AVX2 instructions and `-ftree-vectorize` to explicitly request vectorization. During testing, we compiled with both `-mavx2` and `-march=native` settings to evaluate performance with AVX2 and AVX512 instructions respectively. The manually vectorized implementation uses `AVXFLAGS` with `-mavx2`, `-mfma` to enable fused multiply-add operations, and alignment optimizations (`-malign-double` and `-falign-loops=32`). To build all implementations, run `make all` in the project directory. The command `make test` builds and executes the test suite that verifies correctness across all implementations and saves the results to a .csv file. Individual executables can be built using `make softmax_plain`, `make softmax_auto`, or `make softmax_avx`.