## Results

This section presents a comparative analysis of three softmax function implementations under varying conditions. We evaluate the `softmax_auto` implementation with and without the `parallel` directive and compare performance between AVX2 and AVX512 instruction sets.

**Experimental Setup** All experiments were conducted on a system equipped with dual Intel(R) Xeon(R) Gold 5318Y CPUs, providing a total of 96 threads (48 per CPU). The system has 1TB of RAM and runs Ubuntu 22.04 with Linux kernel 5.15.0-119-generic. All code was compiled using GCC 11.4.0 with appropriate optimization flags.

## Performance

We compare the execution time of three softmax implementations across various input sizes, analyzing the effects of parallelization and vectorization instruction sets on the auto-vectorized implementation. Figures 1 and 2 demonstrate performance without parallelization, while Figures 3 and 4 show results with parallelization enabled.

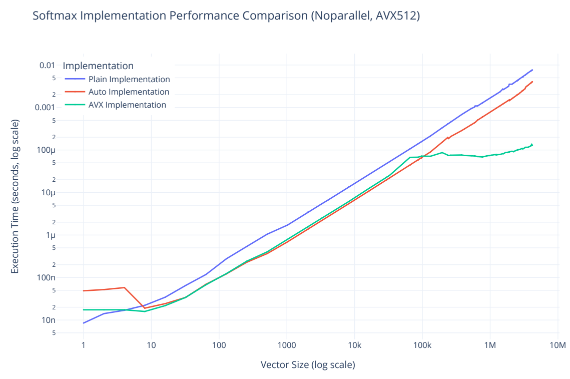

Figure 1: Performance of softmax implementations without parallelization and with AVX512 instructions.

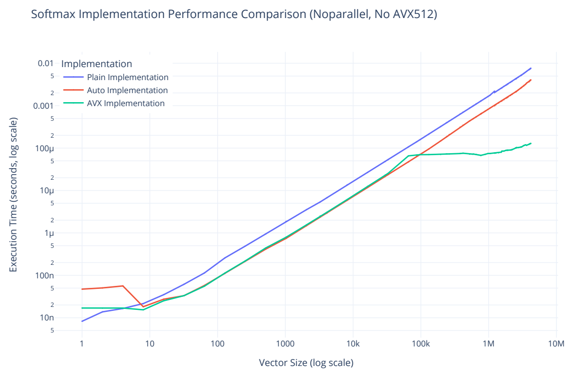

Figure 2: Performance of softmax implementations without parallelization and without AVX512 instructions.

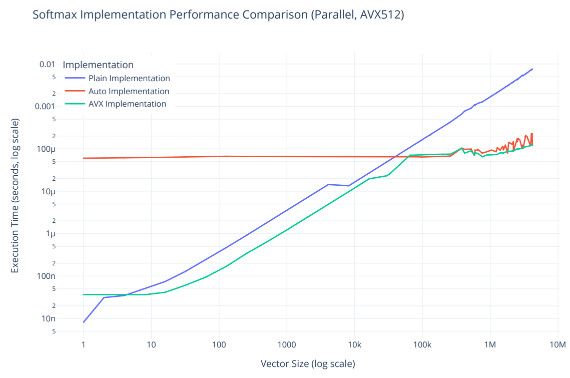

Figure 3: Performance of softmax implementations with parallelization and AVX512 instructions.

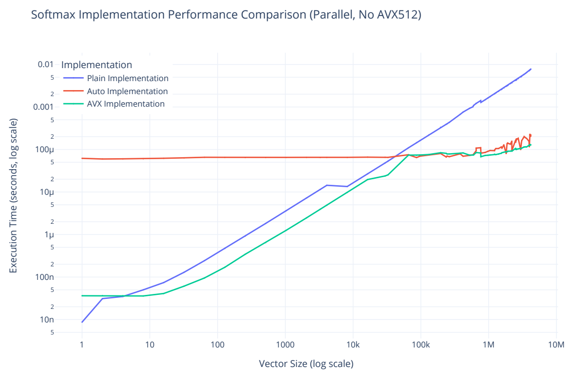

Figure 4: Performance of softmax implementations with parallelization but without AVX512 instructions.

The manually vectorized implementation `softmax_avx` employs an optimization strategy where parallelization is deliberately not enabled for small input sizes (fitting within L1 cache) to avoid thread management overhead. As evidenced in the performance profiles, when the input size becomes larger, the manually vectorized implementation (that is parallelized) outperforms the auto-vectorized implementation (both with and without parallelization).

It should be noted that the logarithmic scaling on both axes helps with visualization across multiple orders of magnitude but obscures less evident performance differentials between implementations. The following section will give a clearer visualization of the performance differences.

## Speedup

We analyze the relative speedup of various configurations compared to the baseline plain implementation. Figures 5 through 8 illustrate the performance gains achieved through different combinations of parallelization and vectorization techniques.

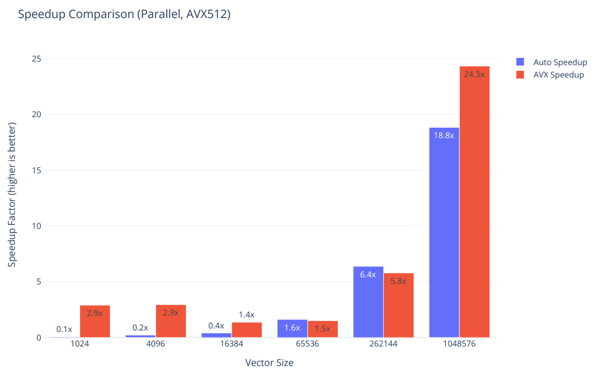

Figure 5: Speedup with parallelization and AVX512 instructions.

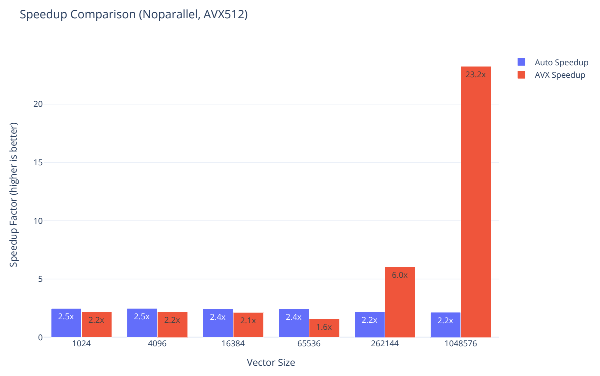

Figure 6: Speedup without parallelization but with AVX512 instructions.

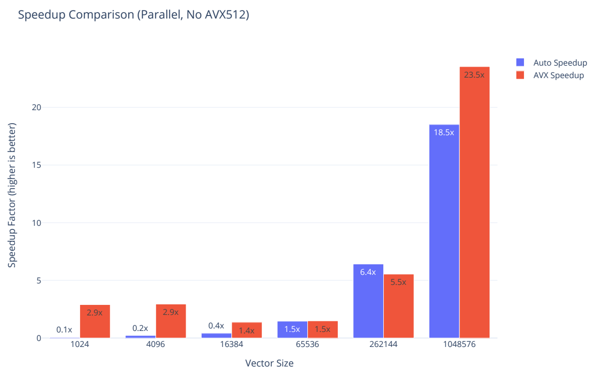

Figure 7: Speedup with parallelization but without AVX512 instructions.

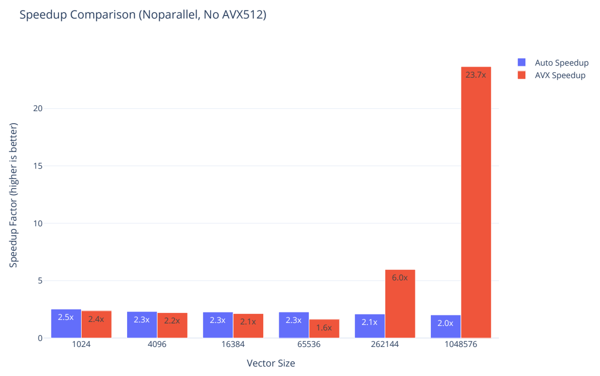

Figure 8: Speedup without parallelization and without AVX512 instructions.

Figures 5 and 7 demonstrate that parallelization yields substantial performance benefits for the auto-vectorized implementation with large input dimensions, while introducing counterproductive overhead for smaller inputs, reducing efficiency to levels comparable with scalar implementation. Regarding instruction set influence, comparative analysis between Figures 5 and 7, as well as between Figures 6 and 8, indicates that AVX512 provides marginal performance improvements over AVX2, with improvements coefficients remaining closely above statistical significance[1]

[1] I am very surprised by this result, as I expected a more significant performance boost from AVX512. I even looked at the disassembly code generated by the compiler to verify that the AVX512 instructions were indeed being used.

## Scalability

We evaluate thread scalability using a fixed input size ($K = 2^{30}$) while varying thread count from 1 to 96. Figure 9 shows the execution times, while Figure 10 presents the speedup alongside the theoretical Amdahl's Law prediction. Notably, a performance discontinuity occurs at approximately half the maximum thread count, corresponding to the number of cores in one of the system's two physical processors.

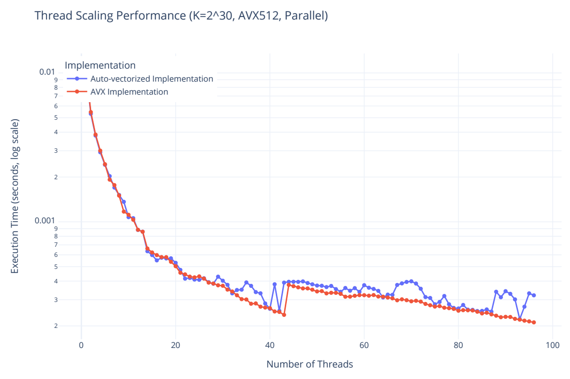

Figure 9: Execution time scaling with thread count for the softmax implementations.

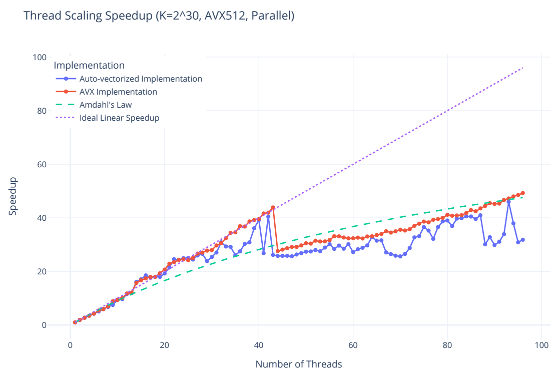

Figure 10: Thread scaling speedup compared to Amdahl's Law prediction.

## Numerical Stability

We evaluate numerical stability by measuring how closely the sum of each implementation's output approximates the theoretical value of 1.0. Figure 11 demonstrates that the plain implementation shows numerical instability at moderate dimensions ($K = 2^{20}$), while both vectorized implementations maintain stability. Figure 12, which excludes the significantly divergent plain implementation for clarity, reveals that the manually vectorized implementation achieves superior numerical precision compared to the auto-vectorized variant at very large dimensions. Nevertheless, the auto-vectorized implementation maintains acceptable numerical stability even at substantial input sizes ($K = 2^{30}$). The superior numerical stability of `softmax_avx` can be attributed to several implementation factors: its controlled horizontal reduction pattern, block-based processing that limits error propagation, precise handling of non-aligned elements through explicit masking, and deterministic operation ordering. In contrast, `softmax_auto` relies on compiler-determined reduction strategies and lacks the explicit error-controlling mechanisms of the manually vectorized implementation.

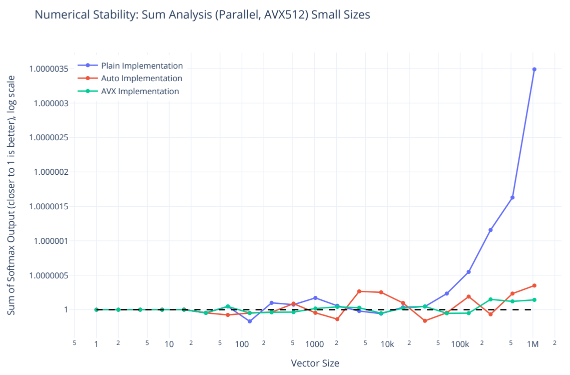

Figure 11: Numerical stability with small input sizes.

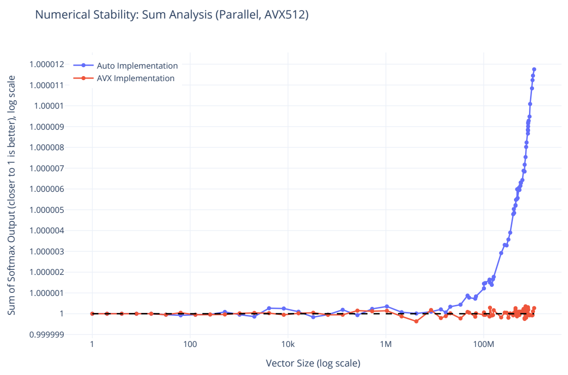

Figure 12: Numerical stability without large input sizes.