Assignment 1: The softmax function

Luca Lombardo

March 9, 2025

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

In implementing[1] the autovectorized version of the softmax function, I made several key modifications compared to the plain implementation. First, I added `#pragma omp simd` directives to explicitly instruct the compiler to vectorize the three main computational loops, allowing parallel processing of multiple array elements with SIMD instructions. For the first two loops, I included appropriate reduction clauses (i.e., `reduction(max : max_val)` and `reduction(+ : sum)`) to ensure correct calculation of the maximum value and sum while maintaining vectorization. I replaced `std::exp()` with the single-precision `expf()` function, which is specifically optimized for floating-point operations, offers better performance with SIMD instructions, and avoids unnecessary double-precision calculations that would be performed by `std::exp()` before converting back to float. Rather than using repeated divisions in the normalization step, I precomputed the inverse of the sum (`inv_sum = 1.0f / sum`) and used multiplication operations, which are generally more efficient in vectorized code. Instead of using `std::max()`, I implemented an explicit comparison with an `if`-statement that might be more amenable to autovectorization for the compiler.

It is worth noting that replacing `#pragma omp simd` with `#pragma omp parallel for simd` results in significantly degraded performance for small values of K (becoming even slower than the plain implementation). This counterintuitive behavior occurs because thread creation and management introduce considerable overhead that outweighs any computational benefits when processing small arrays. The cost of spawning threads, synchronizing their execution, and handling potential load imbalances becomes expensive relative to the limited work being performed. On the other hand, fot large values of K, the parallelized version scales well: the computation workload here is sufficiently large to justify the overhead of thread management, and the performance gains from parallel execution become evident.

1.2 Manually Vectorized implementation

In designing the softmax implementation, I developed two distinct versions to address different input size scenarios. For large input arrays, the primary variant employs cache blocking with a block size derived from the L1 cache capacity (4096 bytes) to ensure that each data block remains in cache during processing. This decision minimizes memory latency through effective prefetching and loop unrolling, while OpenMP parallelization distributes the computational load across threads. In contrast, the "small" version avoids the overhead associated with thread management and cache blocking, which is unnecessary for data sizes that already reside in cache. This dual-approach allows the implementation to achieve optimal performance:

[1] This version is implemented in the file `softmax_auto.cpp`.

the large variant excels with high data volumes where memory access becomes the bottleneck, and the small variant delivers better throughput for cache-resident datasets.

For the primary function, several specific optimization choices were made to maximize efficiency. Data is processed in chunks of 32 elements per iteration to match the width of AVX registers, and for indices not multiple of 8, scalar processing is used rather than vector masking; the infrequent residual iterations render the simpler scalar approach both effective and low-overhead. The use of the `_mm256_fnmadd_ps` intrinsic in the exponential computation phase was carefully integrated to combine the subtraction of the maximum value with the preparation of the exponent input in one step, thereby reducing the overall instruction count and streamlining the computation pipeline.