SMP Assignment 1

DAVIDE BORGHINI

March 2025

Contents

1 Introduction **1**

2 Implementations **1**
2.1 Plain 1
2.2 Auto 1
2.3 AVX 1

3 Experiments **2**

4 Results **2**

5 Conclusions and possible further optimization **3**

1 Introduction

The objective of this report is to show 3 different implementations to compute the Softmax function:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

The goal is not to achieve the most computationally accurate approximation but rather to provide a reasonably efficient implementation, taking into account the computational time required. We can achieve this by starting with a straightforward sequential version, then utilizing compiler flags to vectorize the computation whenever feasible, and finally directly implementing a version using Intel Intrinsic instructions.

2 Implementations

2.1 Plain

This implementation was given by the professor. The computation is divided into 3 main kernels (loop).

The first loop computes the maximum of the array; although this is not strictly needed, it is fundamental to ensure numerical stability in the exponential computation. It can be proven that if we subtract from each number the maximum of the array, the final output will remain unchanged.

The second loop computes two things:

- $e^{z_i - \max}$ this is, for each element, the numerator part of the Softmax equation.
- A value that is the sum of all the numerators needed as the denominator of the Softmax.

The third and final loop calculate the division of the elements computed in the previous loop (each numerator divided by the total sum).

This fully sequential implementation is useful because it provides us with 3 different parts that can be optimized independently.

2.2 Auto

First, the flag `-fopt-info-vec-all` was used to see which loop needed to be optimized and to keep track of what was optimized; by default, none was optimized. After this some trials with various flags were done, the one that was found to be the most useful was `-ffast-math` which by itself was enough to vectorize all loops; the addition of `__restrict` in the function signature allowed the compiler to remove the alias check and making the code more efficient.

2.3 AVX

The last implementation explicitly utilizes intrinsics. To simplify the code, the program does not handle arrays shorter than 8 elements. In some parts of the code, branching is avoided by either using additional memory or performing operations that could be skipped in specific cases.

Loops are optimized in three distinct stages. First, intrinsics are used to process eight operations at a time. Next, masking or other techniques handle any remaining elements of the array [2]. Finally, if necessary, an horizontal reduction is performed on an 8-element array [3] (e.g., to find the maximum).

In the first loop, the maximum is computed in chunks of eight elements at a time, initializing the max array with the first eight elements of the input array. The remaining elements, which cannot be processed directly in the loop when the array size is not a multiple of eight, are handled by comparing them with the max array. To determine the maximum of these eight values, I modified and reused code from the course slides [3]. This code leverages intrinsics to maximize parallelism and avoid unnecessary looping over eight elements.

In the other loops, the same structure is followed. The only difference is in handling the remaining elements that is done with masks used along with masked load and store operations [2]. I tested two different ways of creating and using masks; the one directly from Stack Overflow and a different one that avoid the use of if else without noticing significant differences.

3 Experiments

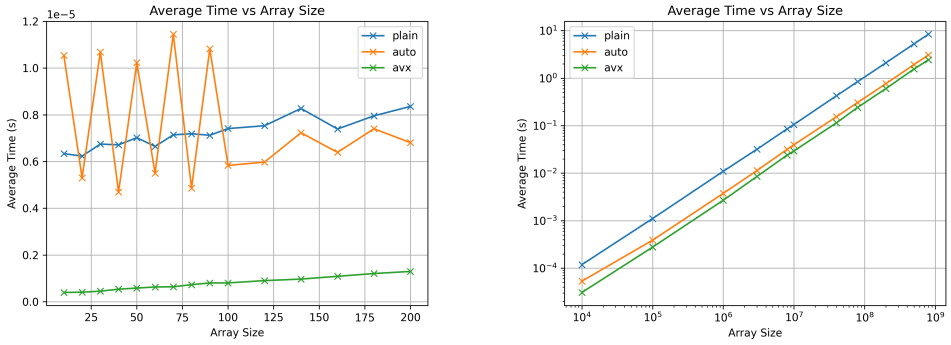
Some preliminary experiments were conducted using both aligned and unaligned memory. In theory, aligned memory should introduce some initialization constraints but also improve performance. Surprisingly, in both the Intrinsic and Autovectorization versions, performance worsened. For this reason, the final code and experiments refer to the unaligned version.

To evaluate the performance of the various implementations, two different tests were conducted: one on small arrays (ranging from 10 to 200 elements) and another on larger arrays (ranging from 10,000 to 800,000,000 elements). This distinction was made because I expected the AVX version to perform significantly better on large arrays, as it can execute most operations within a loop that computes eight values at a time, thereby minimizing the impact of handling the remaining elements outside the loop.

To run test the command `sbatch slurm_run_tests.sh` was runned, and the results were visually plotted using the script in `compute_avg.py`.

4 Results

The various execution times can be founded in Figure 1. Form these plots we can see different behaviours and speedups moving from short arrays to longer ones.



(a) (b)

Figure 1: (a) Graph showing the computing time with different (small) array sizes. (b) Graph showing the computing time with different (big) array sizes in log log scale.

Let's begin by analyzing the compute time for the largest arrays. We observe that the speedup is almost constant for all implementations. Therefore, we can confidently conclude that for sufficiently large arrays, the AVX implementation is the fastest. The time taken by the AVX implementation is quite close to the version autovectorized by the compiler. The approximated speedups can be found in Table 1.

Comparison	Speedup Factor
Auto vs Plain	2.75
AVX vs Plain	3.5
AVX vs Auto	1.25

Table 1: Performance comparison at steady state

When considering shorter arrays, we observe that the AVX version consistently outperforms the other two, achieving a speedup of approximately 6×—even greater than the improvement seen with larger arrays. This could be due to the computation becoming memory-bound as array size increases. As for the autovectorized version, it runs faster than the standard implementation when processing array sizes that are multiples of 4. However, for non-multiples of 4, not only does it fail to provide a speedup, but it actually performs worse than the plain version.

5 Conclusions and possible further optimization

The AVX implementation proves to be significantly faster, especially for small arrays. However, this performance boost comes at the cost of a considerable investment in optimization, requiring nearly a full day of fine-tuning using intrinsics. At a certain point—though determining that exact threshold is beyond the scope of this report—it may become more efficient to offload the computation to a GPU.

The tests conducted with aligned memory yielded unsatisfactory results, showing performance roughly twice as slow as the unaligned version. While I do not have a definitive explanation for this, a reasonable assumption is that maintaining memory alignment introduces overhead, and modern CPUs are already well-optimized to handle unaligned memory efficiently.

Understanding the behavior of autovectorization with multiples of four would require a deeper analysis to determine the exact conditions under which it consistently outperforms the plain version. However, whenever feasible, explicitly using Intel Intrinsics appears to be the preferred approach.

Some improvements might be achievable by designing custom arrays that are always allocated in multiples of eight, which would simplify handling edge cases at the end of the array. However, I do not expect this to result in a substantial performance gain. A more detailed study, examining each loop individually, could provide further insights and suggest directions for additional optimizations. At the very least, it would offer a more in-depth understanding of the current performance bottlenecks.

References

[1] GitHub repo containing the code: GitHub. (2025)

[2] Handle indivisible vector lengths with SIMD intrinsics: Stack Overflow. (2022)

[3] 7-SIMD-on-cpu.pdf: MS Teams. (2025)