Assignment 1

Softmax function parallelization

Giuseppe Gabriele Russo 583744

March 2025

1 Implementation choices

I tested various implementations, both for the auto-vectorized and manual versions. For the manual version, I experimented with two approaches: one was a **custom implementation**, while the other used maximum **function sourced online** and horizontal summation given in the slide of the course. The goal was to evaluate the performance and efficiency of each approach and explore different strategies for optimizing the computation. Both methods were designed to handle the task efficiently, with the online-sourced functions potentially offering faster execution times due to their optimization.

1.1 Softmax_auto

For the auto-vectorized version of the softmax, I applied loop unrolling to optimize the maximum search loop. To enable auto-vectorization for the summation, I introduced temporary variables. Additionally, I added `#pragma GCC ivdep` for all loops to help with vectorization. For the summation and the inverse multiplication calculation loops, I included `#pragma GCC unroll 8` to further optimize performance.

However, despite trying various combinations—adding and removing the pragmas, using or not using temporary variables, and applying or skipping unrolling—I wasn't able to achieve parallelization. The key factor that made parallelization successful was enabling the `-ffast-math` flag, which allowed the code to parallelize and perform significantly better. The final loop performs the multiplications by inverse values, completing the softmax operation efficiently.

To ensure a comprehensive comparison, the plain version of the code was also compiled with the `--ffast-math` flag.

1.2 Softmax_avx

In the AVX implementation, the maximum value is calculated using the intrinsic `_mm256_max_ps`. This operation returns a vector of 8 elements, representing the maximum values found in parallel. From these 8 values, the final maximum is obtained using a standard loop. Additionally, if the vector size is not a multiple of 8, I perform another simple loop to find the maximum among the remaining elements.

For the denominator summation, I created a vector sigma of size 8, where I store the differences between the 8 values that are processed in a single cycle and the previously found maximum. The results are stored in an 8-element vector of sums. After processing the main vector, I sum up the values in the sum vector. In case the input vector size is not a multiple of 8, the same operations are applied to the remaining values, and their results are added to the sum.

The cycle for the multiplications with the inverse values is executed, and the 8 operations are saved in the `vec` vector, which is then stored. If there are any remaining values, I perform another loop to process the leftovers.

Additionally, an alternative version of this part has been implemented, where instead of performing the inverse multiplication, a division is used. While the division is slower than the inverse multiplication, it provides greater precision. This trade-off between speed and precision was considered for ensuring the accuracy of the softmax operation in the final implementation.

1.3 Softmax2_avx

In the file **softmax2_avx.cpp** implementation, different versions of the maximum and summation functions were tested. One of the maximum functions was sourced online and aims to emulate the maximum search functionality available in **AVX512**, `vmaxps`, which finds the maximum value within a vector in parallel, used to improve the performance of the maximum calculation in this AVX version.

The summation function used in this version was the one provided in the course, which processes the elements of the vector in parallel and accumulates their sums efficiently. The goal was to combine the benefits of both the custom maximum function and the course-provided summation function to optimize the softmax operation's performance on the AVX platform.

2 Performance evaluation and comparisons

All versions of the implementation were **tested three times** on vectors of **sizes 10, 100, 1000,** and **10000**. For each vector size, the error introduced by parallelization was calculated, along with the execution times. By running multiple trials, we were able to compute the mean and variance for each test case, providing a more accurate representation of the performance and reliability of the parallelized versions. This approach allowed us to assess both the efficiency gains and the precision of the parallelization process across different input sizes. The table 1 reports all the time results obtained for the different cases. The results obtained by the auto vectorization of the plain code are omitted because they are equal to the other auto version.

Method	**K**	**Mean (s)**	**Variance (s^2)**
softmax_plain	10	6.824×10^{-6}	5.153×10^{-13}
	100	7.652×10^{-6}	4.766×10^{-13}
	1000	1.720×10^{-5}	1.103×10^{-13}
	10000	1.200×10^{-4}	3.948×10^{-12}
softmax_auto	10	1.1132×10^{-5}	2.5706×10^{-13}
	100	6.156×10^{-6}	1.0333×10^{-13}
	1000	1.1413×10^{-5}	4.6264×10^{-14}
	10000	4.5963×10^{-5}	5.2531×10^{-14}
softmax_avx	10	8.201×10^{-6}	6.5137×10^{-13}
	100	8.440×10^{-6}	4.8257×10^{-14}
	1000	5.356×10^{-6}	1.2877×10^{-13}
	10000	3.2216×10^{-5}	9.3567×10^{-14}
softmax2_avx	10	6.538×10^{-6}	9.1895×10^{-13}
	100	7.4917×10^{-6}	7.1119×10^{-13}
	1000	5.582×10^{-6}	5.0074×10^{-13}
	10000	3.2340×10^{-5}	1.4137×10^{-13}

Table 1: Mean and variance of execution times for different Softmax implementations

2.1 Execution Time Analysis

From the results, we can observe that for small input sizes (K = 10, 100), there are no significant advantages gained from parallelization in any of the cases. The overhead introduced by parallel execution likely offsets the benefits when dealing with such small workloads.

As the input size increases, we notice a clear difference in execution times. In particular, the manually parallelized version exhibits an **exponential increase in speedup**, whereas the automatically parallelized version scales less aggressively. This suggests that manual optimization allows for better exploitation of computational resources when handling large workloads.

Additionally, for larger input sizes (N = 1000, 10000), there is **no significant difference** between the two manually parallelized implementations (softmax_AVX and softmax2_AVX). This indicates that both versions reach a similar level of efficiency.

All the evaluations made for the auto version also apply to the code obtained by compiling the plain version with the flag.

Overall, while automatic parallelization offers a reasonable improvement in performance, manually optimized implementations demonstrate superior scalability for large-scale computations.

2.2 Numerical Stability Analysis

An important aspect to consider between all the different implementations is their numerical stability. It was observed that the automatic parallelization (‘softmax_auto‘) exhibited higher instability, possibly due to the ‘`-ffast-math`‘ flag. In some cases, this resulted in deviations from the ‘softmax_plain‘ implementation, even for small input sizes such as 10. The same problem was observed in the code plain auto optimized, highlighting how the issue does not depend on changes made to the code but on the problem itself.

For both AVX versions, discrepancies were observed only for large input sizes (over 10000). This suggests that while vectorized implementations generally maintain stability, numerical errors can still emerge as input values grow. Additionally, further analysis showed that even the ‘softmax_plain‘ implementation tends to lose stability with very large values, indicating that numerical issues are not exclusive to parallelized or vectorized versions but are an inherent challenge in softmax computations.

3 Manual vs auto vectorization

When comparing manual and automatic parallelization, several trade-offs emerge in this specific case.

The manually optimized implementation provides better performance, particularly for large input sizes. This is because it allows fine-tuning operations, optimizing memory access patterns, and leveraging hardware-specific instructions such as AVX. However, this comes at the cost of increased development effort. Writing and debugging manually parallelized code is significantly more complex than relying on automatic compiler optimizations, which can generate efficient vectorized code with minimal effort.

Another important aspect is scalability versus portability. Manual parallelization is often more scalable since it provides full control over execution and can be adapted to different workloads. However, this may reduce portability, as manually tuned implementations may depend on specific CPU architectures or instruction sets. In contrast, automatic parallelization benefits from better portability, as compilers can optimize code for different architectures without requiring significant changes.

In this specific experiment, we observed that for small input sizes, automatic parallelization performs similarly to manual implementations, as the overhead of parallel execution outweighs its benefits. For larger input sizes, manually optimized versions outperform the automatic approach, particularly when using AVX instructions. However, the difference between the two versions of AVX implementations is minimal, suggesting that optimizations may already be reaching hardware limits.

Overall, the choice between manual and automatic parallelization depends on the context. If maximum performance is required and the hardware is well understood, manual optimization is preferable. However, if ease of implementation, maintainability, and portability are priorities, automatic parallelization remains a viable solution.

4 Challenges and Future Improvements

During the implementation and testing of different parallelization approaches, several challenges were encountered. One of the main difficulties was balancing performance and numerical stability; for some implementations, like softmax_auto, the use of the flag `-ffast-math` introduced some error with respect to the plain version.

Another challenge was measuring performance accurately. Since execution times were relatively short, small variations in system load or scheduling could introduce noise into the results. The trade-off between portability and performance was also evident. While manually optimized implementations performed better for larger inputs, they were more architecture-dependent. Ensuring compatibility across different processors while maintaining peak performance remains an open challenge.

For future improvements, several directions could be explored. First, implementing hybrid approaches that combine manual optimizations with compiler-assisted vectorization could provide better portability while retaining some benefits of fine-tuned performance. Another potential improvement is exploring different parallelization strategies, such as OpenMP for multi-core parallelism.

Overall, while the current implementations demonstrate the effectiveness of parallelization, there is still room for refinement, especially in optimizing performance while maintaining numerical stability and portability.