Report Assignment1 - Softmax Function Implementation

Angelo Nardone

1 Implementation Process

Let's begin with **three premises** that served as the foundation for the implementation of my work:

1. The file `softmax_plain.cpp` was **not modified** in any way and serves as the baseline for comparing the other two implementations.
2. In constructing the `softmax_avx.cpp` and `softmax_auto.cpp` codes, I used the file `softmax_plain.cpp` as a reference. This means that, although the implementations may differ in structure, the **operations** for computing the softmax in both versions **remain the same** as those in `softmax_plain.cpp`.
3. It is assumed that the input value K is always **greater than or equal to 8** (in reality, usually larger). Handling cases where K is smaller than 8 would have required only a simple `if` statement. Therefore, the decision not to modify the code to handle these cases is not due to negligence, but because (for me) it does not make sense to vectorize the calculation of a maximum or exponential for a vector with fewer than 8 elements. For such small numbers, the sequential code works perfectly well. Furthermore, as our study is purely educational, I believe that not considering these cases is a reasonable assumption that should not be condemned.

1.1 Softmax_Avx Implementation

I start by describing the version that uses **intrinsics**. In this case, the function `softmax_avx` underwent several modifications compared to the `softmax_plain` version. The main change was in how I managed the variables. To vectorize the operations, I used `__mm256` to handle 8 32-bit floats at a time. This change also affected the operations within the loops. Specifically, the maximum value was computed using `_mm256_max_ps`, allowing for 8 maximum values to be tracked from 8 comparisons of two floats. The exponentials were computed in batches of 8 using `exp256_ps`, and their sum was handled with `_mm256_add_ps`. Each operation was manually managed as 8 operations at once, similar to the sequential case.

Another important aspect was how I handled cases where K was not a multiple of 8. To address this, I introduced an **auxiliary variable** `Kminus7` to prevent the main loop from accessing invalid memory regions. I then used a **mask** to retrieve information from the remaining input elements. This allowed me to continue processing with `__mm256`, while using the mask in the final step to indicate which data should be computed and which was irrelevant to the process.

Observation 1 (FMA)**.** In this phase, as seen in the commented code, I found an alternative way to add the exponentials calculated with the mask to the vector that had accumulated the sums throughout the main loop. This alternative method used **FMA**. The execution time and results were identical to those obtained with the main version, which was preferred simply because, as discussed in class, the backend nodes of the spmcluster do **not support the FMA operation**.

Another important aspect is that, at the end of the maximum and sum computation, the result to be returned was a **single float**. However, as described earlier, the final output consisted of 8 floats arranged through `__mm256`. In one case, I needed to take the maximum of these 8 floats, and in another, I needed to sum them. To achieve this, I used the `hsum_sse3` and `hmax_sse3` functions, as discussed in class. While the summation case was covered in class, I simply modified the maximum function to compute the max instead. These functions allowed for parallel computations, optimizing both the number of operations and memory usage.

I also added the **compilation flag** `-march=native`, enabling the use of AVX2 (or AVX in the backend node) during computation. For loading and storing data from memory cells to `__mm256` variables, I used `_mm256_loadu_ps` and `_mm256_storeu_ps`, where the `u` ensures compatibility with **unaligned data**.

I tested data alignment using `_mm_malloc`, for both the AVX and auto versions. After several tests, the execution time for aligned vs unaligned data was nearly identical, with the unaligned version performing slightly better. For this reason, and because I preferred not to change the main function but only the `softmax` function, I decided to keep the unaligned data version of the code.

1.2 Softmax_Auto Implementation

I will now proceed to explain the version of the code that uses **auto-vectorization**. This code was fairly simple to modify. In fact, the function `softmax_auto` remained almost identical to the version in `softmax_plain` code. The only difference lies in the function's definition, where I had to use the `__restrict` keyword to pass the input and output parameters. This keyword was necessary to inform the compiler that the `input` and `output` pointers not only refer to different memory locations but, in fact, their corresponding memory regions **do not overlap** in any way. This allowed the compiler to apply optimizations, particularly enabling auto-vectorization.

Another modification compared to the plain version was in the **compilation flags**. In addition to the flags already present in the **Makefile**, which apply to all code (such as `-O3`), I added the following:

- `-ftree-vectorize`: Enables automatic vectorization to leverage the processor's SIMD capabilities.
- `-ffast-math`: Activates arithmetic optimizations that can improve performance. As seen in the lecture, these optimizations may compromise numerical precision. It is best to avoid using this flag in contexts where precision is critical. However, in our case, as we will see, it does not affect the results.
- `-march=native`: Optimizes the code for the specific microarchitecture of the machine on which I compiled the code.

To **verify** that the code was correctly auto-vectorized, I also used the following flags during the development and testing phases: `-fopt-info-vec-all` and `-fopt-info-vec-missed`.

In particular, I found that using the `-ffast-math` flag was necessary to enable auto-vectorization. While it is generally advisable to avoid this flag when not required, as it may affect stability, it was crucial in our case. I also experimented with **temporary variables** to manually eliminate potential **loop dependencies** and tried using `#pragma` directives as an alternative. However, I was unable to enable auto-vectorization without the `-ffast-math` flag. Despite this, the output of `-fopt-info-vec-missed` clearly indicated that auto-vectorization still could not be achieved without it.

2 Performance Evaluation

Let's begin by stating that all the codes were compiled and executed on a **backend node** of the machine provided by `spmcluster`. Additionally, it should be considered that the **results obtained with the three codes are equivalent**, except for a few values that differ beyond the tenth decimal place. This discrepancy is due to the **numerical oscillations** present in the various calculations. Therefore, we are discussing functions that, as expected, return the same output.

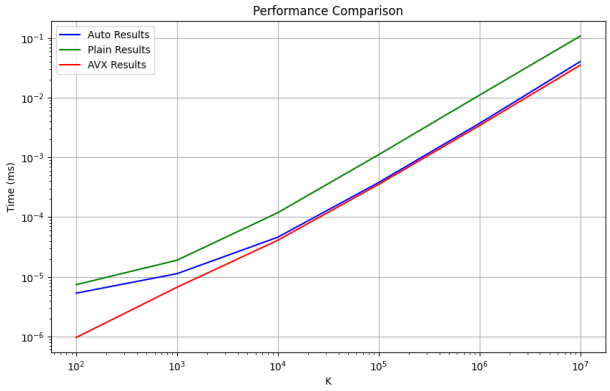

Figure 1: Execution times for varying input sizes.

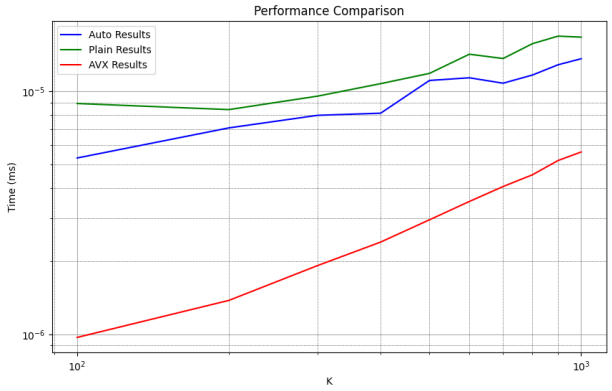

Figure 2: Execution times for smaller inputs ($K \in [100, 1000]$).

Let's focus initially on the **execution times** (computed as the average over different iterations with the same input K). As can be seen in Figure 1, the `avx` code is always the fastest for any input. I believe this is due to the fact that the **optimizations were manually handled**, meaning the process is fully directed by the written code. We can also observe that the auto-vectorized version, although initially slower, quickly aligns with the performance of `avx` and maintains this performance. In particular, the figure shows that from a certain point onward, all three codes exhibit a **linear increase in execution time with respect to the growth of the input**. However, in the earlier iterations, the `auto` and `plain` versions struggle to maintain this stability.

This discrepancy in the early iterations can be explained by the fact that, in the initial stages, the `auto` and `plain` codes do not benefit from the same level of optimization as the `avx` code. The auto-vectorization, though effective, still

depends on the compiler's ability to vectorize the code efficiently, which may not always occur optimally for small inputs. As the input size grows, both `auto` and `plain` versions begin to show more consistent performance, eventually converging with the `avx` code's behavior. Figure 2 provides a detailed view of the execution times for smaller inputs (K between 100 and 1000), showing the stability of `avx` and the slight oscillations in the `auto` and `plain` versions.

Finally, let's study the **speedup** between the various codes. We can consider the **absolute speedup** as the one obtained by comparing `avx` and `auto` to the `plain` version, while the **relative speedup** refers to the comparison between `avx` and `auto`. The table below shows the different speedups. We observe that the absolute speedup stabilizes for both `avx` and `auto` from a certain point onward. While for `auto` the speedup is approximately 2.75, for `avx` it is around 3, indicating that we have managed to find a code that runs about **three times faster** than `plain`. Despite performing 8 operations simultaneously, the improvement is not 8 times faster: this is due to factors such as overhead, memory access latency, and parallelization efficiency, which prevent a perfect scaling of performance. Finally, we also observe the relative speedup, which shows that from a certain point onward, `avx` is on average **1.1 times faster** than `auto`, indicating that while both codes run at very similar speeds, `avx`'s implementation with less management proves to be slightly more efficient.

Speedup / K	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
Auto vs Plain	1.388044	1.679662	2.574279	2.926106	2.971977	2.670091
AVX vs Plain	7.647344	2.834978	2.893365	3.148978	3.230186	3.071345
AVX vs Auto	5.509438	1.687826	1.123952	1.076167	1.086881	1.150277

3 Trade-offs Between `auto` and `avx`

When considering the trade-offs between `auto` and `avx`, it is important to understand their respective advantages and limitations. `avx` is the optimal choice when the primary objective is to achieve the **best execution time** and **numerical stability** (compared to the auto-vectorised version), as it does not rely on the `-ffast-math` flag, which could compromise precision. However, the syntax required to implement `avx` is much more complex, making it **more difficult to manage and write** efficient code.

On the other hand, `auto-vectored` code is a suitable alternative when stringent time and numerical precision constraints are not required, but performance improvement is still desired. With `auto-vectored` code, only minimal modifications are needed, and the process can be guided using the `info` flag, which provides **insight** into missed vectorization opportunities. This approach strikes a balance between ease of implementation and performance enhancement, making it ideal for situations where time and precision constraints are less critical (but always to be taken into account).

4 Challenges and Future Improvements

One of the greatest **challenges** I encountered was working with `avx`, as this was my first experience with vectorization. Understanding the intricacies of low-level computer operations and getting into the mindset of vectorizing the code was the most difficult part of the job.

Regarding optimizations, there are certainly many areas for improvement. One such area I am aware of is the **handling of the mask**. I found a very smart way to create the mask, particularly described on this page: Smart Mask Creation. However, since I had already implemented my own method, which worked well, I decided to keep it. This decision was largely because it was my original work, not something taken from an online source, and because I felt it was a minimal optimization that did not significantly affect the main objective of the assignment, which was to demonstrate proficiency with intrinsics.

Another optimization lies in **memory management**. In simplifying my work with the code, I likely defined variables that could have been better managed or even avoided. Efficient memory handling, particularly in the context of large-scale data processing, could certainly improve the performance and scalability of the implementation. Furthermore, there is room for improvement in the organization of the code to ensure better modularity and readability, which would make it easier to extend the implementation in the future.

Other possible improvements include exploring different approaches to fine-tune the vectorization process, optimizing data structure alignment for AVX, potentially reducing unnecessary computations by eliminating redundant algorithm steps (though I do not currently see any) or refining the auto-vectorized code to enable vectorization even without the `-ffast-math` flag.