
Department of Computer Science

**Computer Science Master Degree**

# First Assignment - Softmax

## SPM Course 2024/2025

**Student: Luca Rizzo**

**ID Number: 598992**

# Introduction

In this study, I analyze three different implementations of the softmax function, focusing on their performance and optimization strategies. The goal is to compare a compiler-assisted auto-vectorized version and a manually optimized AVX version to understand their efficiency in leveraging SIMD instructions for improved computation speed.
Since the plain scalar version serves as a baseline for comparison, it is not analyzed in detail for brevity.

# Implementation

## SOFTMAX_AVX

The softmax_plain implementation is structured into three main computational kernels: finding the maximum value, computing the output array using exponentiation while summing the results and normalizing the output using the computed sum.

Each of these kernels represents an opportunity for parallelization using SIMD AVX instructions, allowing operations on contiguous groups of 8 floats in the input array at a time. For clarity and modularity, the implementation has been logically divided into three dedicated functions:

- **avx_max**: the maximum value is found using the progressive selection algorithm seen in the lesson, where the largest 8 float values at fixed strides of 8 positions are stored in a 256-bit AVX register (e.g., at index 3 in the AVX register, we store the maximum between input[3], input[3+8], input[3+16], etc.).
  However, if the total number of elements is not a multiple of 8, we cannot process them as a full AVX block. Instead, we use a static mask with *_mm256_maskload_ps* to load only the valid values from the input array, while leaving the remaining positions in the register set to zero.
  To ensure that these zeroed-out positions do not interfere with the max selection process, we apply a blend operation with -INFINITY, preserving only the meaningful values. Finally, the selected 8 values from the AVX register are stored in a temporary array, where a final scalar comparison determines the global maximum.
- **calculate_output_and_sum:** the exponentiation and summation step broadcasts max_val into an AVX register to apply the same transformation to each 8-float block. The max value is subtracted, and the result is passed to *exp256_ps()* for vectorized exponentiation. The output is stored in 8-element chunks, while a 256-bit AVX register accumulates partial sums across blocks. If K is not a multiple of 8, a masked load loads only valid values, followed by masked store (_mm256_maskstore_ps) to avoid buffer overflow in output array. Since non-significant positions may still contain values, a blend operation resets them to zero before adding them to the sum register. A final horizontal reduction (*hsum_avx* seen in lesson) computes the total sum.
- **divide_output_by_sum**: the computed sum is broadcasted into an AVX register, allowing efficient division of 8-float blocks in output. The updated values are then stored back into the output array. As in the previous steps, if the total number of elements is not a multiple of 8, we use a masked load to retrieve only the valid values, apply the division operation, and then

store the results back using a masked store, ensuring that only the correct positions are updated based on the mask.

The code was compiled using flag -O3 -mavx since the implementation exclusively uses AVX instructions.

### SOFTMAX_AUTO

To optimize the auto-vectorized version of the softmax function, an initial attempt was made using the standard optimization flags *-O3 -march=native -funroll-loops .*
However, compiling with the diagnostic flag *-fopt-info-vec-missed* revealed that the compiler failed to vectorize two key loops: the maximum value search due to a Read-after-Write (RAW) dependency on max_val, and the exponentiation computation, where the use of std::exp prevents efficient vectorization. In fact, moving the summation to a separate loop does not solve the issue, as std::exp itself remains an obstacle to vectorization

```
└─$ g++ -std=c++17 -I. -I./include -fopt-info-vec-missed -march=native -funroll-loops -O3 -o ./bin/soft
max_auto softmax_auto.cpp 2>&1 | grep "softmax_auto.cpp" | grep "missed: couldn't vectorize loop"
softmax_auto.cpp:21:19: missed: couldn't vectorize loop
softmax_auto.cpp:13:19: missed: couldn't vectorize loop
```

Using the directive #pragma GCC ivdep does not help and is not correct for the first loop, as the dependency is real and cannot be ignored without compromising the correctness of the program.
By adding the flag *-ffast-math*, the compiler successfully vectorized both loops by applying more aggressive floating-point optimizations.
To further enhance performance, all three loops were preceded by *#pragma GCC unroll 4*, which expands the loop body and reduces the total number of iterations, improving execution efficiency.

# Performance analysis

A PHONY TARGET named launch_benchmark is defined in the Makefile, which triggers a bash script to execute the three compiled programs five times for different values of K, specifically K_VALUES=(64 71 517 1031 10204 102040 1048576 1048580 1048583). The results are saved in a CSV file located at ./out/benchmark_results.csv for further analysis, including computing the average runtime across multiple runs and evaluating the speedup relative to the plain version. The selected values include both cases where K is divisible by 8 and cases where it is not, allowing an analysis of potential performance differences when handling non-multiple-of-8 workloads.
Since the auto-vectorized version is compiled with the -march=native flag, it is necessary to fully recompile the sources before running benchmarks. This is ensured by setting cleanall as a dependency of the benchmark target, allowing the generation of executables optimized for the specific machine architecture where they will be executed, ensuring maximum compatibility and performance. The following benchmarks are based on five executions per configuration, with the average performance recorded. The experiments were conducted on internal cluster nodes using the command *srun make run_benchmark*. The results show that the AVX version consistently achieves a higher speedup compared to the auto-vectorized version when measured against the baseline softmax_plain implementation.

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 64 | 5.429800e-06 | 1.720984 |
| softmax_avx | 64 | 7.574000e-07 | 12.337734 |

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 71 | 1.050860e-05 | 0.617894 |
| softmax_avx | 71 | 8.864000e-07 | 7.325361 |

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 10204 | 0.000048 | 2.442987 |
| softmax_avx | 10204 | 0.000034 | 3.413258 |

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 1048583 | 0.003872 | 2.947221 |
| softmax_avx | 1048583 | 0.003222 | 3.542053 |

It is interesting to observe that for low values of K that are not divisible by 8, the auto-vectorized version optimized with compiler flags performs worse than the plain implementation. This could be due to inefficient handling of non-multiple-of-8 workloads, potentially leading to suboptimal vectorization or excessive branching. Further investigation could analyze how the compiler processes such cases.
Despite its limitations for low values of K that are not divisible by 8, it still outperforms the plain implementation for higher values of K, even when K is not a multiple of 8.
However, running make run_benchmark on the frontend node leads to different speedups for high values of K. This discrepancy is likely due to the fact that the frontend node supports AVX2, and the code may be optimized using newer instructions. Some results are shown only for some values of K and it is evident that the auto version performs significantly faster.

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 102040 | 0.000150 | 5.465195 |
| softmax_avx | 102040 | 0.000282 | 2.894993 |

| Target | K | Average | Speedup (Plain) |
|---|---|---|---|
| softmax_auto | 1048576 | 0.001429 | 5.814541 |
| softmax_avx | 1048576 | 0.002784 | 2.984076 |

Further confirmation comes from modifying the compilation of the auto version by replacing the -march=native flag with -mavx, thereby forcing optimization specifically for AVX instructions. In this case, the performance of the auto version on the frontend node matches that observed on the internal nodes (not shown for brevity).

# Correctness analysis

A PHONY TARGET named diff_outputs is defined in the Makefile, which triggers a Bash script to execute the three compiled programs and store their stderr output in a file to compare the results. For small values of K (32 to 8,000), the output files are identical, but as K increases, minor differences appear in the least significant digits for certain values between the auto/AVX and plain versions. The AVX and auto versions show no differences even for larger K. Additionally, as K grows, the sum of the output values deviates slightly from 1, likely due to accumulated floating-point precision errors. For **1048576** elements, the sum of the output values for each file is as follows:

```
Sum of out/softmax_auto.txt: 0.980842
Sum of out/softmax_avx.txt: 0.980842
Sum of out/softmax_plain.txt: 0.9807739999999999
```