Assignment 1 report
SPM course
2024/25

Francesco SCARFATO

March 13, 2025

1 Objective

The objective is to improve the efficiency of the softmax function using two optimization approaches: compiler auto-vectorization and manual vectorization with AVX intrinsics.

2 Implementation

2.1 Compiler auto-vectorization

The auto-vectorized version of the softmax function is very similar to the plain implementation with a few key modifications to help compiler optimizations. The `__restict__` is applied to the `input` and `output` pointers, inform that these pointers point to different memory locations (no aliasing), allowing for vectorization. A minor improvement is reducing the memory accesses by writing to the `output` array only once per element. The flag used to auto-vetcorize the loops in the softmax function are:

- `-O3`: Activate all the optimizations, including loop unrolling and splitting, ...
- `-ftree-vectorize`: Enable the vectorization.
- `-march=native`: Generate optimized instructions for the local CPU architecture.
- `-ffast-math`: Enable floating point optimizations.

With these flags, the compiler is able to auto-vectorize the three loops in the softmax function.

2.2 Vectorization with AVX intrinsics

The first step is to compute the maximum value of an input array. This implementation initializes a 256-bit vector with `-INFINITY` and processes eight elements per iteration, loading them into an AVX register and updating the maximum value across the vector. After completing the vectorized loop, the final maximum is extracted using `hmax_avx`. If the array size is not a multiple of eight, any remaining elements are processed sequentially.
The second step is to normalize each element by subtracting the computed maximum and applying the exponential function. Using AVX intrinsics, it processes eight elements at a time by loading them into a 256-bit vector, subtracting the maximum, and computing the exponential using `exp256_ps`. The results are stored in the output array while simultaneously accumulating their sum in a separate AVX register. After completing the loop, the final sum is obtained using `hsum_avx`, and any remaining elements are processed sequentially.

The final step is to divide each exponential elements by the total sum. A 256-bit vector initialized with the sum is used to process eight elements at a time. The exponential values are loaded into a vector, divided by the sum, and the results are stored back in the output array. Any remaining elements are processed sequentially.

3 Performance evaluation

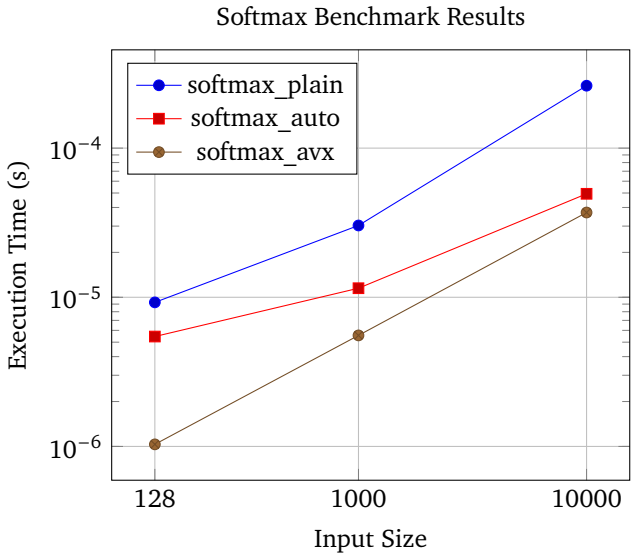
The performance tests were conducted on the internal nodes of the cluster. Each test was repeated five times, and the average execution time was calculated to ensure reliable results. Performance is measured as speedup factor relative to the plain implementation with three different input sizes: 128, 1000, and 10000. The plain version is executed without the `-O3` flag to measure the effective speedup compared to a non-optimized version. However, with the flag enabled, the `softmax_auto` and `softmax_avx` also show better performance.

Input Size	Softmax Plain (s)	Softmax Auto (s)	Softmax AVX (s)
128	9.242800e-06	5.454400e-06	1.032400e-06
1000	3.029340e-05	1.149940e-05	5.550600e-06
10000	2.620686e-04	4.940200e-05	3.700540e-05

Table 1: Average execution times for different implementations

Input Size	Speedup (Auto vs Plain)	Speedup (AVX vs Plain)
128	1.69	8.95
1000	2.63	5.46
10000	5.30	7.08

Table 2: Speedup of auto-vectorized and AVX implementations over the plain version.



The `softmax_avx` is the best implementation in terms of performance across all input sizes. The best result was obtained with the small input size (128 elements) with speedup of ~9x compared to the plain version. Increasing the size, the gap between the plain implementation and AVX become smaller. This decrease can be attributed to:

- Memory bandwidth bottleneck: data transfer overhead exceeds the computation;

- Cache misses: for the smaller input size, the data probably fits entirely in the cache, while for larger input there are more cache misses.

Compiler auto-vectorization improves performance but is less effective than using manual AVX. However, as the input size increases, the speedup improves. For the largest input, this version obtains the best result with speedup of ~5x. For larger input, both `softmax_auto` and `softmax_avx` stabilize, or `softmax_auto` surpasses `softmax_avx` in performance.

4 Conclusions

These results highlight that the optimized versions of the softmax improve massively the efficiency and the performance. Use AVX bring several trade-offs. One is the portability, because AVX is supported on specific hardware and the binary generated couldn't work properly on another machine. Additionally, the implementation is more complex and verbose, and is less readable.
Using the compiler auto-vectorization is surely easier, but there is less control on how the compiler optimize the code. The compiler might not recognize optimization opportunity and the analysis of the optimized code could become difficult.

5 Further improvements

Several improvements can address the accuracy of the performance evaluation and the efficiency of the implementation.
Testing with larger input sizes would provide a better analysis of how the performance of different implementations change and if they stabilize. Another important improvement is increasing the number of test runs when computing the average execution time. Using only five iterations, the test could have a sort of noise caused by the system, the scheduler or something else.
Another improvement could be to implement an aligned memory version. This can reduce the memory access overhead and improve the performance.