Optimizing Softmax through Vectorization

Vincenzo Gargano, MAT: 667591

March 13, 2025

1 Introduction

The softmax function is a fundamental component in most modern deep learning architectures like: LLM since we need to convert the output of the model into a probability distribution. Which is the main reason why we want to use these deep learning models, the softmax appears in a lot of fields, from language with LLM to image classification with CNNs and many more. This project investigates two optimization approaches for softmax computation: manual vectorization using AVX intrinsics and auto-vectorization by compiler with the flag `-O3`. We are going to evaluate their performance characteristics by watching at the speedup achieved over a scalar baseline implementation. **Vectorization** is a SIMD (Single Instruction, Multiple Data) technique that allows us to perform multiple operations in parallel on a single instruction. This is done by using vector registers (bigger than usual registers) that can store multiple data elements and execute the same operation on all of them simultaneously.

2 Implementations

2.1 Baseline (Scalar)

The baseline implementation is a straightforward scalar version of the softmax function:

Formally we calculate the **softmax** function as follows:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^{N} e^{x_j - \max(x)}} \tag{1}$$

where x is the input vector of size N and x_i is the i-th element of the input vector.

```
void softmax_plain(const float* input, float* output, size_t N) {
    float max_val = -std::numeric_limits<float>::infinity();
    for (size_t i = 0; i < N; ++i)
        max_val = std::max(max_val, input[i]);

    float sum = 0.0f;
    for (size_t i = 0; i < N; ++i)
        sum += std::exp(input[i] - max_val);

    for (size_t i = 0; i < N; ++i)
        output[i] = std::exp(input[i] - max_val) / sum;
}
```

Listing 1: Scalar Softmax Implementation

2.2 Auto-Vectorized Version

The baseline implementation was modified to enable compiler optimizations, in addition to that we're using the frontend nodes to compile and execute: `srun make` and `srun ./softmax_program`. For autovectorization to work we just enable these flags:

- `-O3` for maximum optimization : this include `-ftree-vectorize` which enables the vectorization of loops
- `-march=native` to enable the use of the full instruction set of the host machine
- `-ffast-math` to enable the use of fast math operations but this can lead to a loss of precision which is just for the 10th decimal, which is not a big deal when dealing with probabilities normalized to 1

2.3 Manual AVX Vectorization

This is the implementation that took most of the time and require the user to parallelize by hand operation, we used a 256-bit vectorization with AVX intrinsics. So we used AVX2, along an implementation of the exponential function for 8 elements at a time, we used the `exp256_ps` function. The main idea here is to take each of our cycles and make them work on a multiple of 8 elements, and then we handle the remainder with usual scalar implementation at the end.

For example let's see how we handle a crucial part of the computation, the others are similar:

```
__m256 max_val_v2 = _mm256_set1_ps(max_val);
__m256 exp_v;

for (size_t i = 0; i < VEC_SIZE; i+=8) {
  input_v = _mm256_loadu_ps(input+i);
  exp_v = exp256_ps(_mm256_sub_ps(input_v, max_val_v2));
  _mm256_storeu_ps(output+i, exp_v);
}

  // Process remaining elements sequentially
  for (size_t i = VEC_SIZE; i < K; i++) {
      output[i] = std::exp(input[i] - max_val);
  }
```

Listing 2: Key AVX Snippet

We compute 8 elements at a time until the biggest multiuple, in this case we parallelize the exponentiation $e^{x_i - max\_val}$, and then we handle the remaining elements with the scalar implementation.

3 Performance Evaluation

Implementation	K	Time (s)	Speedup
Plain	100	6.47e-6	1.0x
Auto-vectorization	100	5.6e-6	1.16x
AVX2	100	5.3e-6	1.22x
Plain	10000	3.42e-4	1.0x
Auto-vectorization	10000	4.52e-5	7.6x
AVX2	10000	3.54e-5	9.7x
Plain	1000000	0.011	1.0x
Auto-vectorization	1000000	0.0038	2.9x
AVX2	1000000	0.0029	3.8x

3.1 Key observations

Since we're working in parallel with **multiples of 8** in our array and then, if needed, we complete with the inefficient scalar implementation, which is really slow, for example for the $K = 100$ which is not a multiple of 8, we have only a speedup of 1.22x. But for $K = 10000$ and $K = 1000000$ we have a speedup of 9.7x and 3.8x respectively, which is a huge improvement since they are multiple of 8. The loss in performances for the bigger array sizes can be due to some of the instruction not purely AVX in the implementation.

4 Conclusion and Improvements

4.1 Why Manual Vectorization

Why using manual vectorization? Because we can have more control of things like memory alignment, that we didn't handled in the manual auto-vec, that can lead to a bigger performance improvement. Moreover this was a simple example, but the code is far less readable and difficult to debug than auto-vectorization, and it's not portable. What is suggested in more far complex application that uses a lot of vectorization, is to check where auto-vectorization could be far slower: when we get branches and when memory access are not clear so we can use manual vectorization in those cases.

4.2 Future Improvements

To further improve the performance we could use the AVX-512 instruction set, which allows for 16-wide vectorization, we can also align the memory before each computation, and maybe try an approach where we could interleave the autovectorization with the manual vectorization when needed.

5 Conclusion

The manual AVX implementation demonstrated a significant speedup of almost 10x in some cases over the scalar and auto-vectorized versions, especially when the input size is a multiple of 8. This showed the power of SIMD vectorization in optimizing performance-critical code.