ML Pipeline Optimizations

Profiling Analysis

Initially, I implemented the different matmul implementations from previous labs to check which of them performed faster. From these, only the CSR implementation resulted in a faster runtime but only negligibly suggesting that the program executable is not spending enough time in the matmul function like previous labs. To confirm my hypothesis, I decided to use the ***gprof*** profiling tool to better understand the proportions of the various functions within the executable.

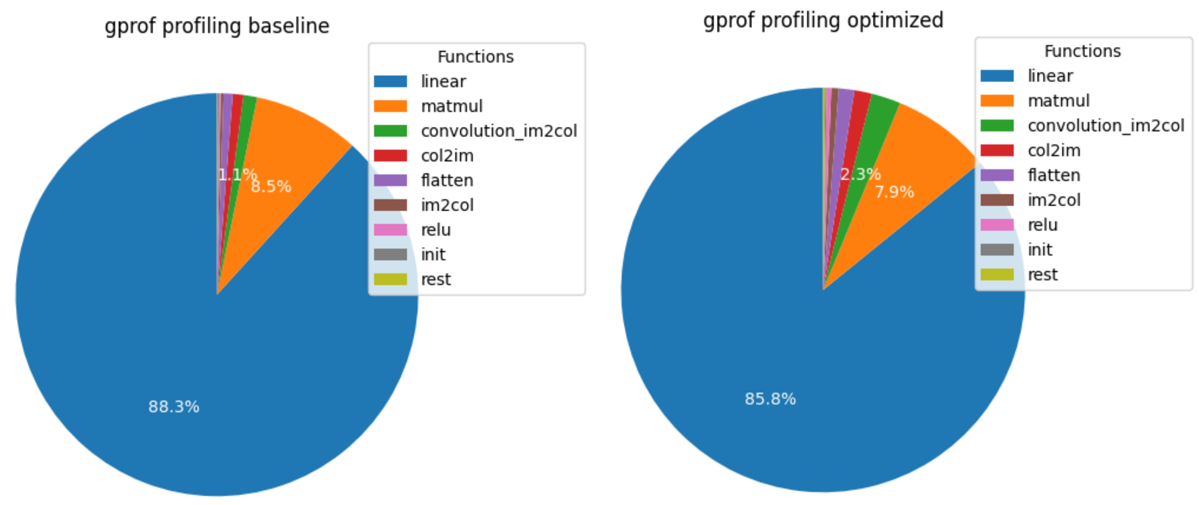


Figure 1 Baseline pipeline function division [left]. Optimized pipeline function division [right].

I compiled my code using the *-pg* flag to include detailed profiling data in the binary and ran the *gprof lab5 gmon.out > analysis.txt* to retrieve the data. From the analysis from Figure 1 (left), it is evident majority of the time is spent in the linear function while the matmul function is the second most used. This explains why the initial matmul optimizations did not result in any significant performance improvement.

Optimizations:

To handle the bottleneck in the linear function, I decided to vectorize the implementation using SSE commands on 128-bit data types. The original code comprises an outer loop iterating over each element of the output vector, with each element initialized by its corresponding bias value. This initialization leverages the _mm_set_ps1 function for setting all elements of a 128-bit vector to the bias value. The inner loop processes the input vector in four-element chunks, aligning with the 128-bit SIMD register capacity. It loads four adjacent weights from the weight matrix and four elements from the input vector, performing element-wise multiplication of these vectors. The results are accumulated in a sum vector using _mm_add_ps. After the completion of the inner loop in each iteration, a horizontal addition is executed using _mm_hadd_ps, summing all elements in the sum vector. The horizontal sum essentially takes 8 32-bit elements or in this case two 128-bit sum variables and adds adjacent elements in pairs of two in the 128-bit sum variable overwriting its values. This parallel sum is done twice so that each element of the sum variable then contains the sum of all four elements before the sum, or in other words, the sum represents the total accumulated value for the current iteration of the output element. This value is finally stored in the output vector using the __mm_store_ps function. This function automatically only fetched the first 32-bit value of the 128-bit value passed to it which for us is the desired final value of the output vector.

Overall, using SIMD operations, we allow parallel processing of multiple data points with a single instruction in the linear function, thereby enhancing computational throughput significantly. In our approach, we utilize the SSE3 instruction set which allows for parallel processing of four data elements in parallel using 128-bit data type, using AVX we could have parallel processed eight data elements in parallel using 256-bit data types, potentially increasing the runtime even further.

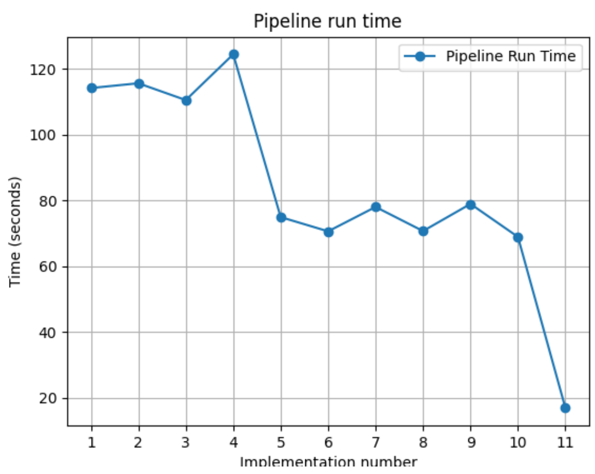
Having optimized the linear function, the matmul function was also optimized by using CSR matrix multiplication instead of the naïve implementation. The insight behind leveraging CSR matrix multiplication is that the MNIST image data is very sparse (over 90%) because only the pixels or matrix cells which have the digits are non-zero.

To further improve program runtime, the -O2 flag was added during compile time. There are numerous other optimizations performed by the compiler that result in an improved runtime, but some significant ones include:

1. **Loop unrolling** in which the compiler makes several copies of the loop to leverage the spatial locality of the cache and reduce the memory access bottleneck which can be significant for matrix operations.
2. **Subexpression elimination** in which redundant calculations and intermediate variables are removed to directly reduce the number of calculations performed at runtime while also removing the necessity to cache temporary variables.
3. **Constant propagation** is where the compiler replaces constant variables in the final instructions of the executable with their original value to reduce unnecessary computations of fetching the values at runtime. This is useful for both the linear and matmul functions.

Analysis:

Figure 1 (right) gives a pie chart showing the division of the time used by different functions after the optimized implementation of the pipeline. From the chart, it is evident that the overall time used by the linear and the matmul functions has reduced and that of other functions increased but only slightly. This is expected when we analyse the runtime of the two pipelines. The baseline implementation took a total of 114.5 seconds to run and resulted in an accuracy of 100%. The optimized implementation on the other hand took only 15.5 seconds to run while still giving an accuracy of 100%. Essentially, even though there was an improvement of almost an order of magnitude in the total runtime of the pipeline, the time dominated by the matmul and linear functions of the is around two orders of magnitude greater than that of the rest of the functions combined.



ID	Description
1	baseline
2	tiling 16
3	CSR matmul
4	parallel matmul
5	SIMD Linear
6	SIMD Linear + tiling 16
7	SIMD Linear + Parallel matmul
8	SIMD Linear + SIMD matmul
9	SIMD Linear + SIMD Parallel matmul
10	SIMD Linear + CSR matmul
11	SIMD Linear + CSR matmul + O2

Figure 2 Runtime for different pipelines drawn on the left and marked on the right.

Figure 2 gives a comprehensive overview of the runtime of the various implemented optimizations of the pipeline. The runtime was measured using the Linux time function and hence measures the time of the entire pipeline from the creation of image objects to their destruction. However, because the optimizations are implemented in only two functions—linear and matmul—the runtime improvement directly maps to the improvement of the inference section within the pipeline which is the focus.

Conclusion:

From the runtime graph, it is evident that the CSR matrix matmul results in an improvement of 5 seconds while the tiling and parallel implementations of matmul result in an even worse runtime. This shows that parallelizing the matmul function introduces a significant overhead of creating and destroying threads which is not offset by any gains in parallel computations. This suggests that the matrices are not large enough which is supported by the fact that tiling does not help improve the runtime either which was the conclusion from lab 2.

On the other hand, by vectorizing the linear function, the runtime drops straight by around 50 seconds. Even with a vectorized linear function, the best optimization in the matmul function is the CSR implementation which performs even better than a vectorized matmul. Vectorization of the matmul function is quite complicated because the data is not contiguous as we are operating on matrices and must fetch the data to create an additional contiguous data parameter as is required for SIMD processing. Overall, SIMD leverages the power of contiguous data for fast and efficient operations which is not as significant for matrix multiplications as is for linear computations.

Finally, adding the -O2 flag during compilation results in another 50-second improvement during runtime for the entire pipeline. We can hence conclude that a vectorized implementation of the linear function with a CSR implementation of the matrix multiplication and optimizer flag during compile time results in the best runtime performance (9x speedup) for the inference section of the pipeline.

At the same time, it is important to notice that in our implementation, the values for the hidden layers for the linear functions are hard-coded to be multiples of 8. This is important because our SIMD implementation does not treat edge cases where the data might fill the entire 128-bit vector and hence calculate illegal values corrupting our calculations. By adding an edge case treatment, our implementation will become minutely slower but given that it is not needed, it was not implemented.