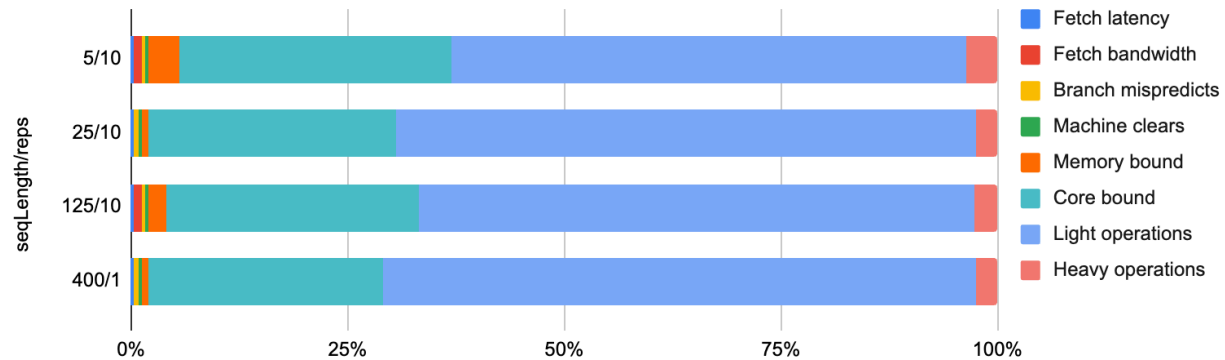


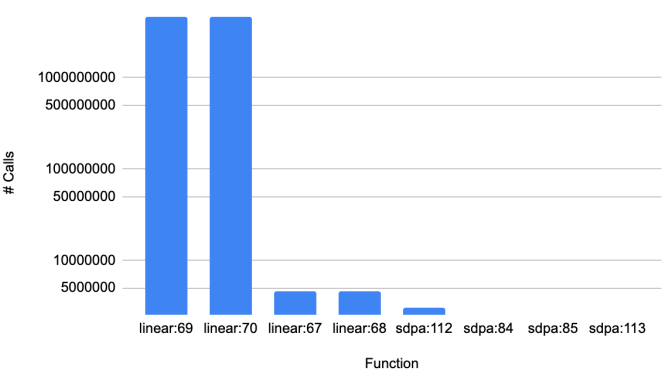
Homework 5, Pierce Hoenigman
Choice: Optimization
Profiling on Base Version

gpt2 unoptimized l2 profiling

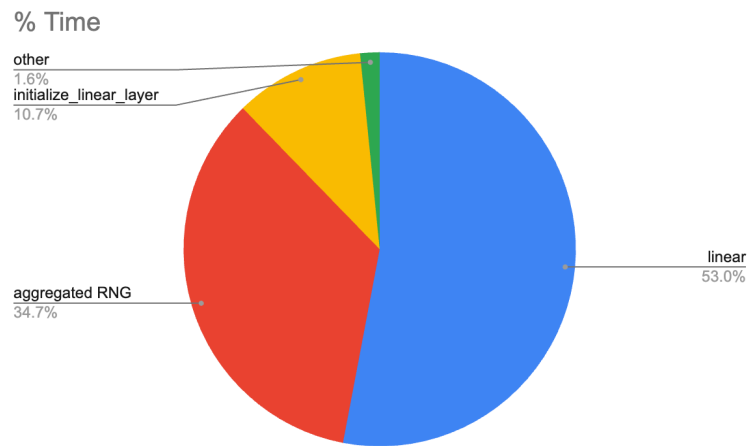


Additional profiling used for pinpointing time sinks:

Function:line with most # calls

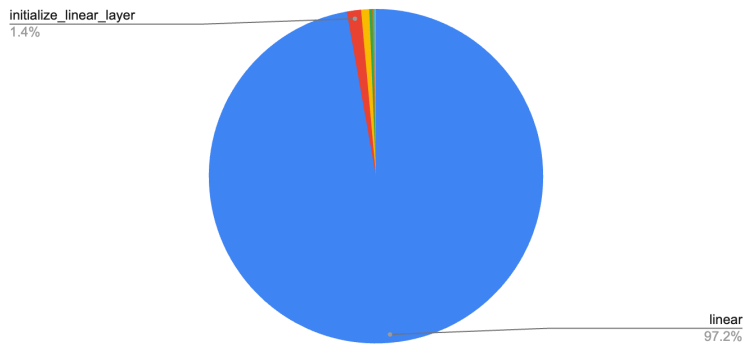


(above via Gcov on running unoptimized with sequence length 5 repeated 10 times)

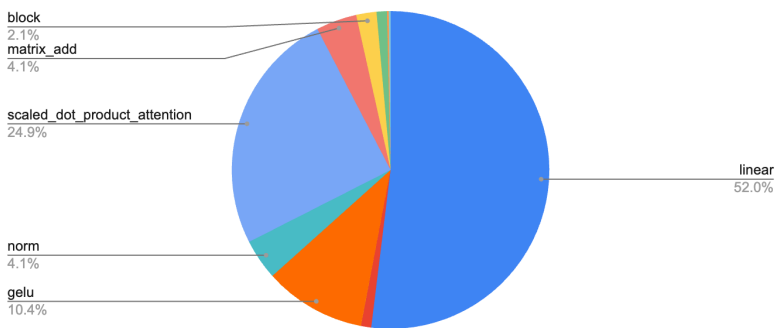


(above via Cachegrind on running unoptimized with sequence length 5 repeated once)

% Time per Function



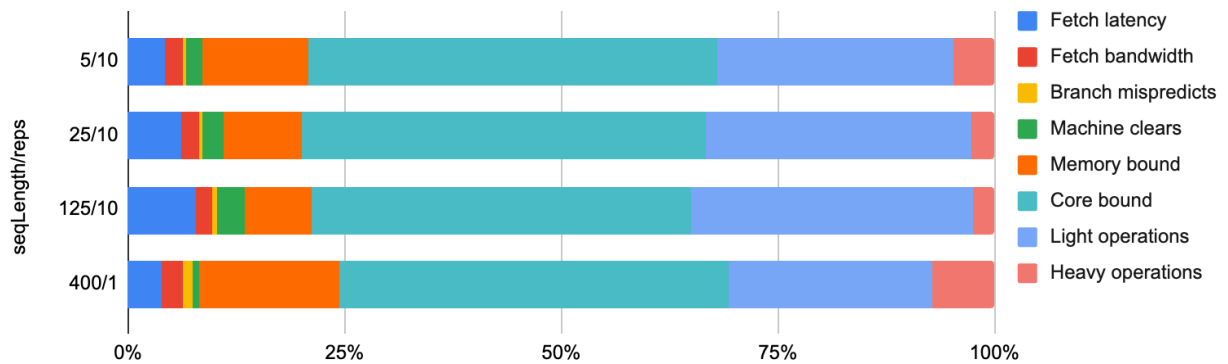
% Calls per Function



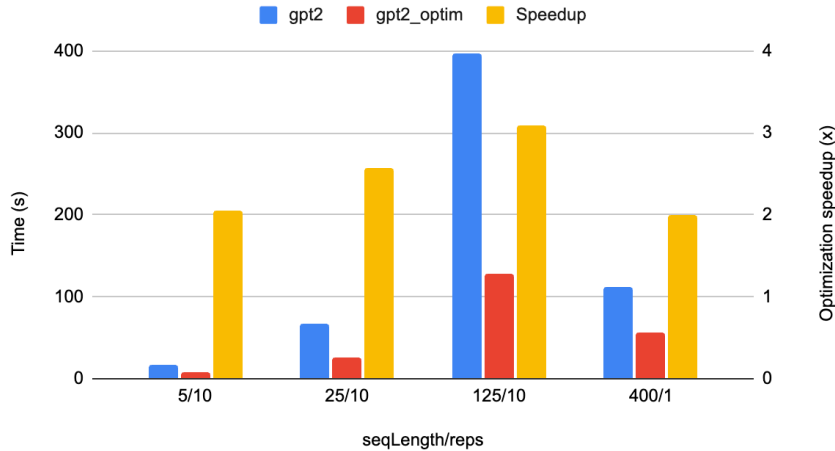
(above via Gprof on running unoptimized with sequence length 5 repeated 10 times)

Profiling on Optimized Version and Runtimes

gpt2 optimized I2 profiling



Time Comparison



How Profiling Data was Used for Optimization

Profiling was done with multiple sequence lengths and forward passes repeated 10x. Note that a sequence length of 400 was chosen for the final test as the unoptimized version would time out the 5 minute limit at ~600 sequence length. Thus, only one repetition was feasible for that test, whereas the other tests on smaller sequence lengths were repeated 10x to amortize the overhead of weight initialization. This was because the cachegrind output of the sequence length 5 run only once on unoptimized code showed that over $\frac{1}{3}$ of the compute time was spent on random number generation for weight initialization, with more than 10% being spent on other initialize_linear_layer overhead. It would be impossible to optimize this down to half of the total unless the actual forward pass was more than half of the compute time.

Other than Cachegrind, Gprof and Gcov were also used to get a better understanding of the unoptimized code and where the time/compute sinks were located. For these which required running some code to gather profiling data, the first test (5/10) was performed. Some graphs showing output data from this profiling are above. Along with the cachegrind output, these pointed at the linear function as the optimization culprit. The Gprof output showed that it made up more than half of function calls and ~97% of compute time (see above pie charts). The Gcov output also showed that two lines in linear were responsible for orders of magnitude more calls than any other lines. These were distantly followed by more lines in the linear function, and even further behind (invisible despite a log axis), some lines in the scaled dot product attention function. Thus, improving just the linear function's performance by 2x would improve overall performance by 2x.

The L2 profiling showed that the unoptimized version was core bound. Though this is a vector-matrix multiplication operation, it is still the case as with previously discussed matmul operations that the function is core bound. This is due to the large amount of computations that overwhelm capacity, especially since many operations in the unoptimized implementation are waiting on the accumulator. In the optimized version, these operations are streamlined and therefore we get more of a memory bound program, since now the fetching of data from

memory can't quite keep up with demand from the 8x SIMD operations. Despite this increase in memory bound, the optimized implementation reaches 2-3x improvement in the tests performed and shown in the bar plot.

Optimization Techniques Used

The optimization technique that proved to be sufficient to hit the 2x mark for all my tests was SIMD vectorization of the `linear()` function with AVX2 primitives. This was quite confusing to me as to how to go about this, as the instructions are often very specific.

We keep the outer loop through each output element, however inside, we begin with a 256-bit sum vector for holding 8 floats (all set to 8 at start). For each 8 elements in the input size, we then load 8 floats from the input and 8 weights, multiplying them in parallel and adding them to the sum vector. After this, we have a sum split into 8 parts that we need to combine into one. To parallelize this, we take a binary approach: extracting the first and last 4 elements into a 128-bit vector, then adding those together in parallel. To go from 4 to 2, we now use another primitive to extract and concatenate the latter half of the 4-element-vector so that we can add it in parallel with itself to get 2 elements. We can then use a shuffling primitive with a mask to move the second item to the first position, and then add it with the 2-element-vector to get a 128-bit vector with the full sum in the first float position. Finally, we extract this 32-bit float.

If the sequence length/input size is not divisible by 8, then we just use a regular, unoptimized adder to increment the sum. The optimized version was fast enough without further optimizing this part. The biases are then added, and we are done with the linear function.

AI Statement

Claude was used for guidance with which SIMD AVX2 primitives to use for my optimization "Which SIMD primitives (SSE, AVX, AVX2, AVX512) would you recommend to use when optimizing the `linear()` function to reduce the function's runtime by 2x?". Intel and Rust documentation were also consulted for this.