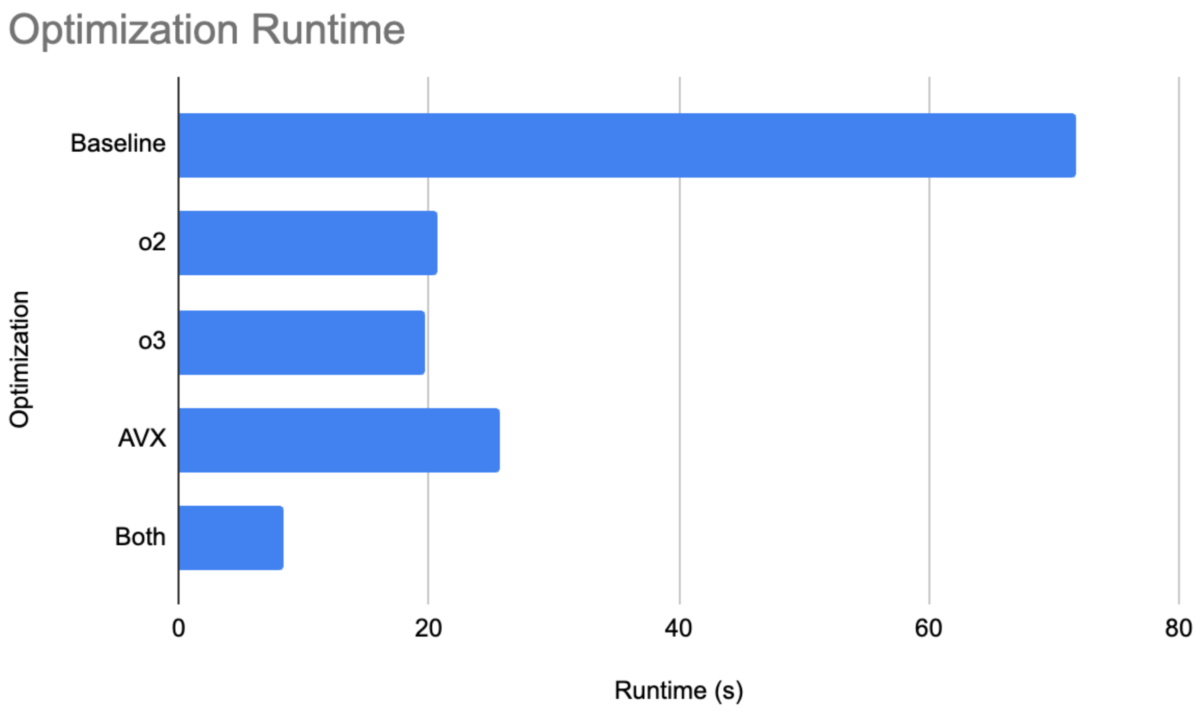
Lab 5

Aneesh Pandoh 12/6/2024

Setup:

Ran Top Down Analysis on each optimization 5 times. For each operation, I ran the model 50 times within the function. I also profiled the code using cachgrind, gprof, and gcov. The optimization include using AVX SIMD, o2, o3 compiler optimization as well as o3 and AVX combined. Please use the makefile to run these optimizations through make gpt.



We can see with the compiler optimizations, there was ~3.5x speed up from the baseline implementation. The compiler optimizations O2 and O3 red ordered and combined some lines of codes which we can see when inspecting the gcov results which do not have exact line to line results of how many times a line was called as it did with the baseline version. The SIMD intrinsic version was also faster than the baseline ~3x which is slower than the compiler optimizations, but these 2 version combined results in about a ~9x speed up as compared to the baseline version.

Baseline:

The baseline implementation includes all the starter code with the filled out TODOs where the multihead attention and MLP was implemented. Running the code through **cachegrind**, **Gprof**, and **Gcov** we can clearly identify the main bottleneck to be the linear function. Gprof had 83% of the time spent on that single function, which was further proven by cachegrind where 53% of the cache events occurred. From Gcov, we could see that line 70, output[i] += fcInput[j] * weights[i][j]; was executed the most in the entire program. This is where the main bottleneck of the model occurs.

SIMD Intrinsics

The first way to improve this was to use SIMD intrinsics, which is x86 is implemented as AVX. In this case I was able to process 8 elements at a time (this can be increased depending on the AVX register size). It was able to be further optimized using the **fused multiply add operation**. Here we are able to compute the unoptimized line 70 (output[i] += fcInput[j] * weights[i][j]) in a single atomic operation sum = _mm256_fmadd_ps(inputVec, weightVec, sum), this is something that is only possible with SIMD registers. This effectively parallelize the code on modern processors, running 8 operations at a time. It is important to note that this may also have been implemented automatically with the compiler optimizations as well. This improves cache spatial locality as now we are loading in 8 floats at once resulting in fewer caches misses and reduces the frequency of memory accesses. We can see from the Gcov output that the FMA generated a whole additional file to be included in the execution count output.

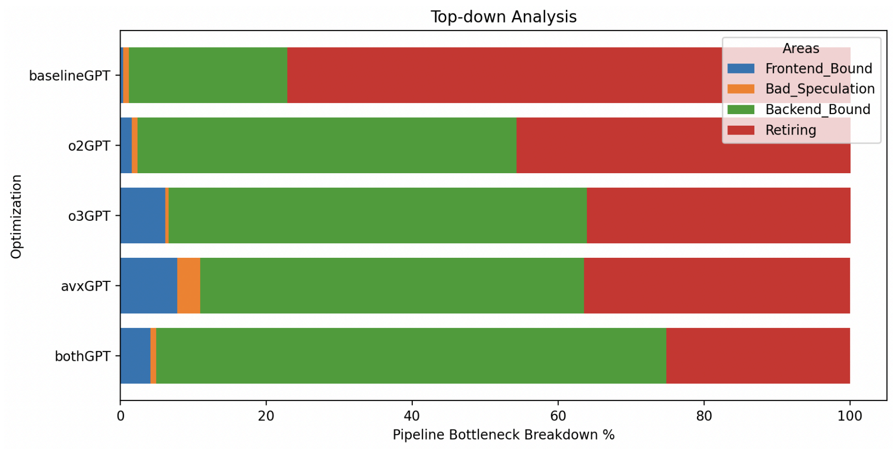
O2 vs O3

The -O2 flag enables a few different optimizations to improve the performance of the code. It includes basic optimizations such as dead code elimination, constant folding, and loop unrolling, which help to reduce unnecessary operations. The compiler may also inline small functions to reduce function call overhead, improve instruction scheduling to optimize CPU pipeline utilization, and allocate values to faster CPU registers rather than memory. Additionally, it applies loop optimizations, such as moving invariant code outside loops and replacing complex operations with simpler ones. Branch prediction and some SIMD may also be improved under -O2. Overall, the compiler can move around the code to reduce bottlenecks, which is reflected by the large speed up compared to our baseline.

The -O3 flag includes all the optimizations from O2 along with some more aggressive/advanced techniques including more vectorizations/SIMD and more branch prediction. This is why we can see a slight increase in speed as well. Due to all the moving around of code, we can see in gprof no longer points to linear function but instead increasingly generalizes runtime to just “model”.

SIMD + Compiler Optimizations:

I was a little surprised to see that the SIMD + Compiler Optimizations were significantly faster than just the -O3 flag, as I would have assumed that the compiler would have optimized the linear function by itself using the same assembly instructions. However, I think due to me not providing the exact architecture of the processors to the compiler during compile time, it wasn’t able to guess what type of AVX would be suitable for the program and would rather be safe by not implementing it.



The top down results are also interesting to see as the unoptimized code actually seems to be the least frontend or backend bound. This is probably because the optimizations now are waiting on memory access as the SIMD operations and optimizations cause the arithmetic to occur very quickly in comparison.