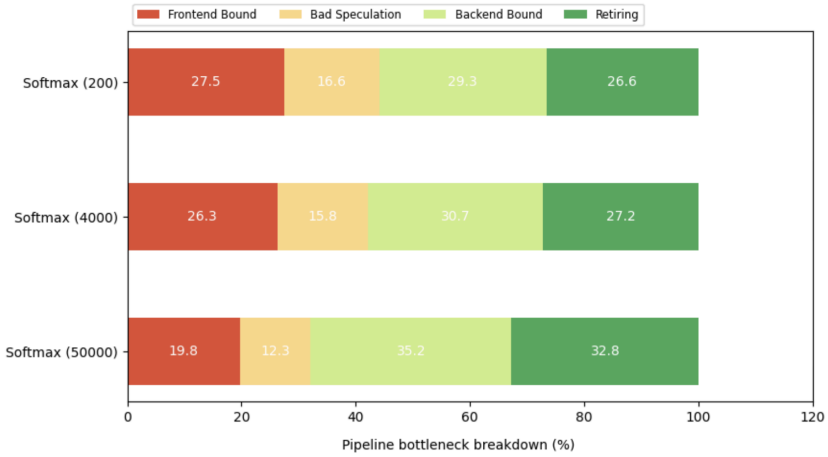
Profiling CNN Functions

Softmax

Profiling



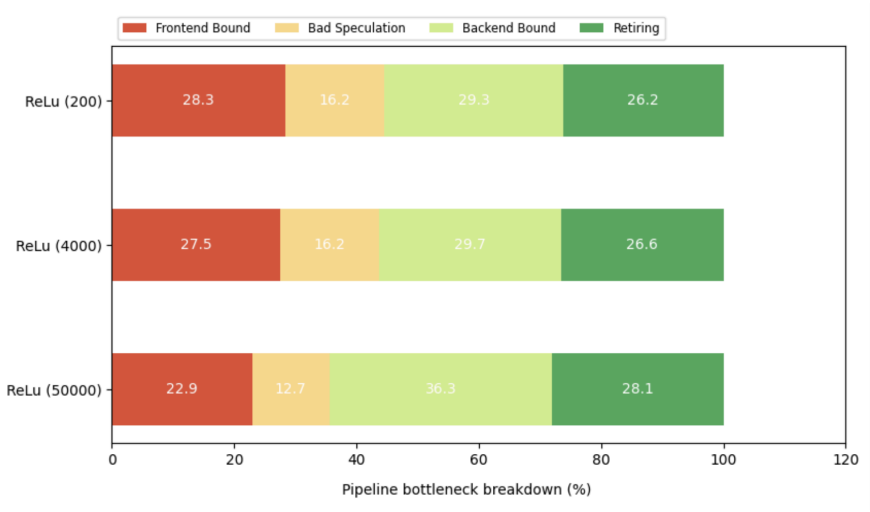
It seems that Backend Bound is the performance bottleneck. And it increases as input size grows, particularly for the large input. This can be caused by cache misses due to larger input data not fitting into lower-level caches, or intensive floating-point operations, for example, exponentiation and division, or data dependency, for example, sum of exponentials.

Implementation

Softmax converts an array of numbers into a probability distribution. The function emphasizes the largest values. I first computed the exponentials and the sum of exponentials. Then, for each element, divide it by the sum of exponentials, and apply log to normalize the result.

Relu

Profiling



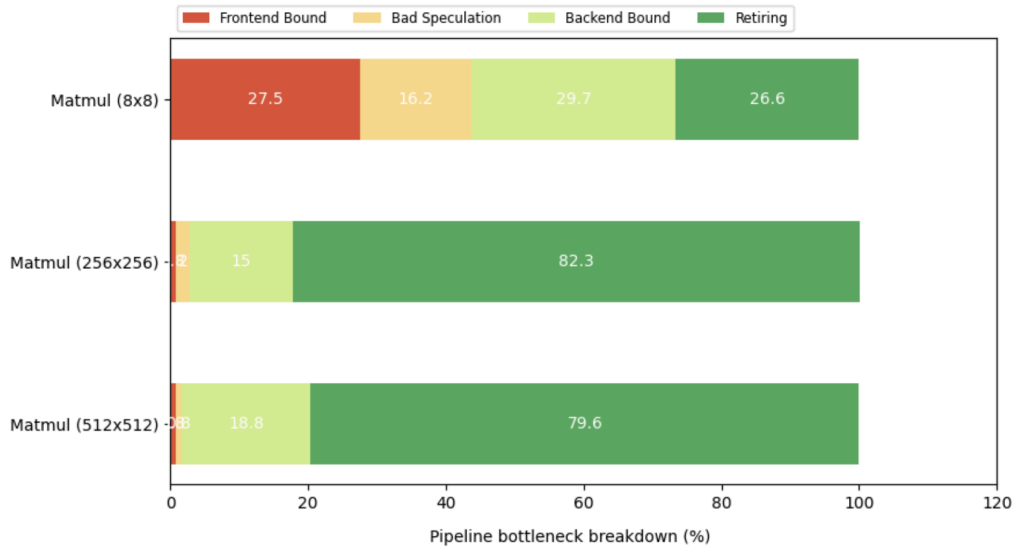
The backend-bound bottle of ReLU indicates that performance is primarily limited by how efficiently the backend (execution units, memory hierarchy) handles the computation. As input size grows, memory and data movement overheads increase, leading to more backend stalls. This might be caused by memory access. For large inputs, memory accesses may lead to cache misses, increasing the backend stalls. This might also be caused by branching (the if condition) which can cause backend stalls.

Implementation

ReLu is an activation function that filters and adjusts data. I implemented it using a if-else condition - if x >= 0, keep it still; else make x=0

Matmul

Profiling



The Backend-Bound percentage is more pronounced in smaller inputs. This could be due to the overhead of accessing smaller matrices. As the input size increases, Retiring becomes the most significant bottleneck, indicating that the implementation effectively completes most of the computation. This might be because matrix multiplication is straightforward, in terms of element indexing and non-intensive computation.

Implementation

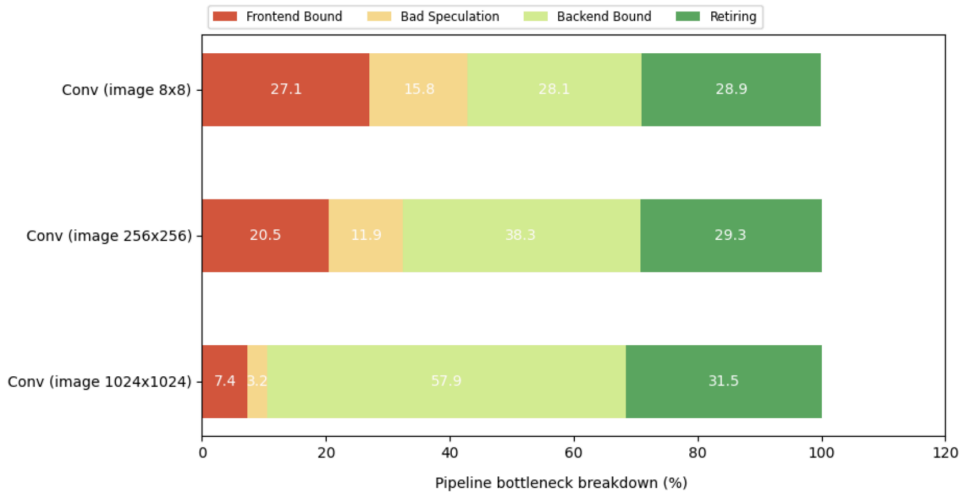
I first did a validation check to ensure that the two matrices have valid dimensions for multiplication. Then I accessed elements in the two matrices by indexes and performed multiplication.

Test

I tested matrices with small length (1), matrices with 0s, negatives, to ensure that the function is robust across different input sizes and values.

Conv

Profiling



It seems that Backend-Bound becomes the primary bottleneck as input size increases, particularly for medium and large image sizes. This can be caused by memory access and computational load, such as sliding windows and kernel operations.

Implementation

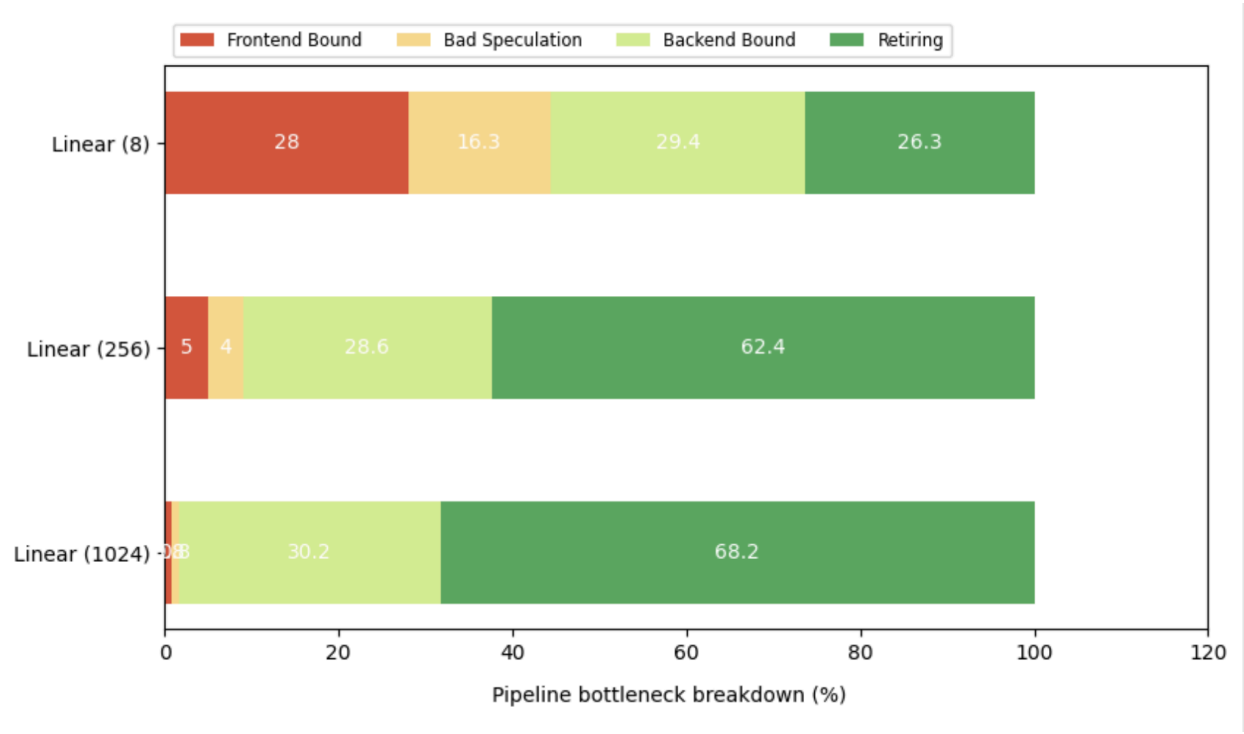
Given that stride = 1 and no padding, I first allocated memory for the output array. Then, for each filter, for each channel, I took the patch from the image and multiplied it by the kernel, sum up values, then sum results from each channel. After that, I applied biases and ReLu to values in each filter.

Test

I tested input images and kernels with varying sizes, channels and filters, also with different values (positives, negatives, zeros).

Linear

Profiling



The backend-bounded percentage becomes larger for larger inputs, which can be caused by memory access and cache misses. The retiring percentages suggest that the function performs well on larger inputs, because the there's no data dependencies and small computational overhead.

Implementation

In the linear function, I applied corresponding weights and biases to the input - the implementation is straightforward.

Test

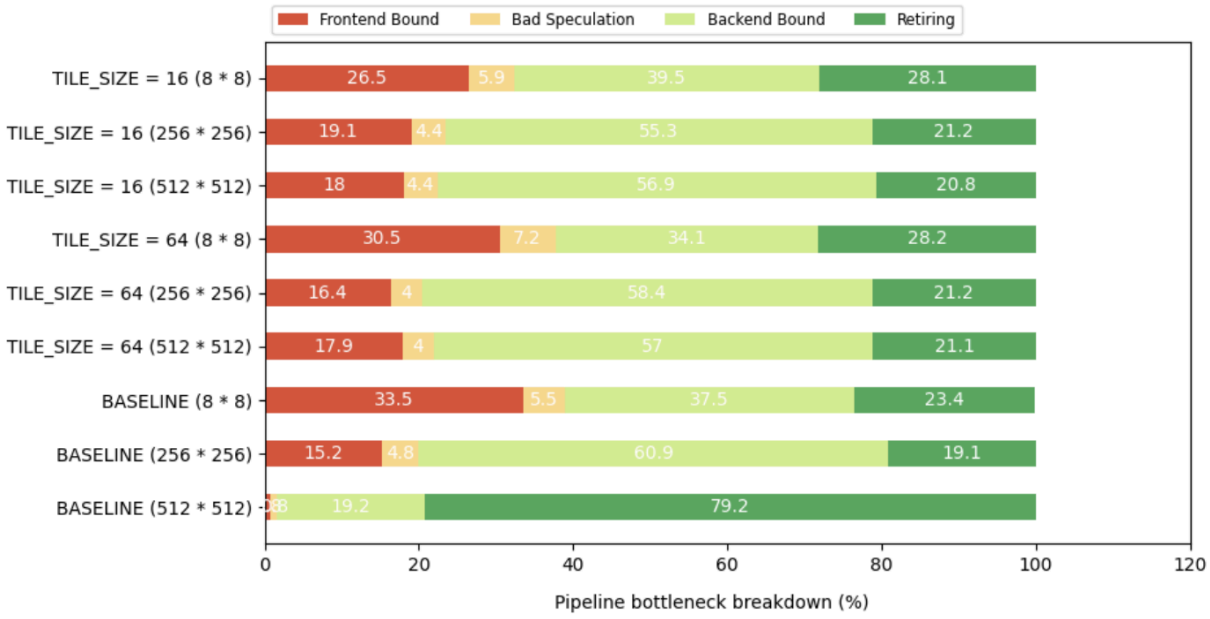
I tested the function across different input sizes, different values. Additionally, when profiling, to minimize the impact of the setup, verification, and cleanup code, I isolate the function I was profiling by repeating the critical section multiple times while ensuring that setup and cleanup are done only once. This way, the profiling primarily captures the runtime of the core function, reducing the noise from the setup and cleanup. I did this also in profiling other functions like conv and matmul.

Optimizations

It seems that backend bound is a dominant bottleneck. So optimizing could involve improving memory access patterns by ensuring data locality, using more efficient data structures, using less intensive floating-point operations, and better parallelization.

Profiling Tiling

Top-down analysis



Performance data (runtime)

Setup	TILE_SIZE = 16	TILE_SIZE = 16	TILE_SIZE = 16	TILE_SIZE = 64	TILE_SIZE = 64	TILE_SIZE = 64	Baseline	Baseline	Baseline
Input size	8 * 8 matrices	256 matrices	512 matrices	8 * 8 matrices	256 matrices	512 matrices	8 * 8 matrices	256 matrices	512 matrices
Runtime	0m0.0	0m0.6	0m5.4	0m0.0	0m0.7	0m5.7	0m0.0	0m0.7	0m5.7

	05s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s	63s 0m0.662s 0m0.663s 0m0.662s 0m0.662s 0m0.661s 0m0.663s 0m0.662s 0m0.663s 0m0.661s	22s 0m5.433s 0m5.431s 0m5.436s 0m5.427s 0m5.428s 0m5.426s 0m5.427s 0m5.429s 0m5.442s	05s 0m0.005s 0m0.004s 0m0.004s 0m0.004s 0m0.005s 0m0.004s 0m0.004s 0m0.004s 0m0.004s	21s 0m0.721s 0m0.720s 0m0.720s 0m0.720s 0m0.721s 0m0.721s 0m0.720s 0m0.720s 0m0.720s	23s 0m5.723s 0m5.723s 0m5.724s 0m5.723s 0m5.725s 0m5.723s 0m5.723s 0m5.724s 0m5.724s	05s 0m0.005s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.004s 0m0.005s 0m0.004s	29s 0m0.729s 0m0.728s 0m0.728s 0m0.728s 0m0.729s 0m0.729s 0m0.729s 0m0.728s 0m0.728s	95s 0m5.796s 0m5.795s 0m5.794s 0m5.793s 0m5.794s 0m5.794s 0m5.794s 0m5.796s 0m5.794s
Average	0m0.004s	0m0.662s	0m5.427s	0m0.004s	0m0.720s	0m5.723s	0m0.004s	0m0.728s	0m5.795s

Discussion

For small matrices (8x8), The average runtime is very small (0.004s) across all implementations and tile sizes, indicating that for this size, memory and CPU resources are underutilized, so caching techniques or other architectural optimizations don't have a major impact.
For medium and large matrices, the baseline (no blocking) runtime is higher than that of matmul_blocking with tile size 16 and tile size 64. And the Backend Bound is fairly high for the baseline, suggesting that memory stalls are limiting performance, as this matrix size starts to exceed the L1/L2 cache capacities and requires more frequent memory accesses. The blocking strategy improves cache locality, resulting in fewer memory stalls, so more instructions can be retired effectively. Tile size 16 seems to have an edge here due to the finer granularity fitting better in cache.
Therefore, the main advantage of matmul_blocking over matmul is the reduction in memory stalls, as the blocking technique leverages cache locality, especially for medium-sized matrices. Tile size 16 appears to be optimal for the 256x256 matrix due to fitting well within cache, while tile size 64 does not fit as efficiently. Additionally, with blocking, the Frontend Bound is generally reduced as the CPU does not have to frequently fetch new instructions. Instead, it can reuse instructions for operations within each tile. This increases the Retiring percentage, meaning more CPU cycles are spent on useful computations, leading to better utilization.