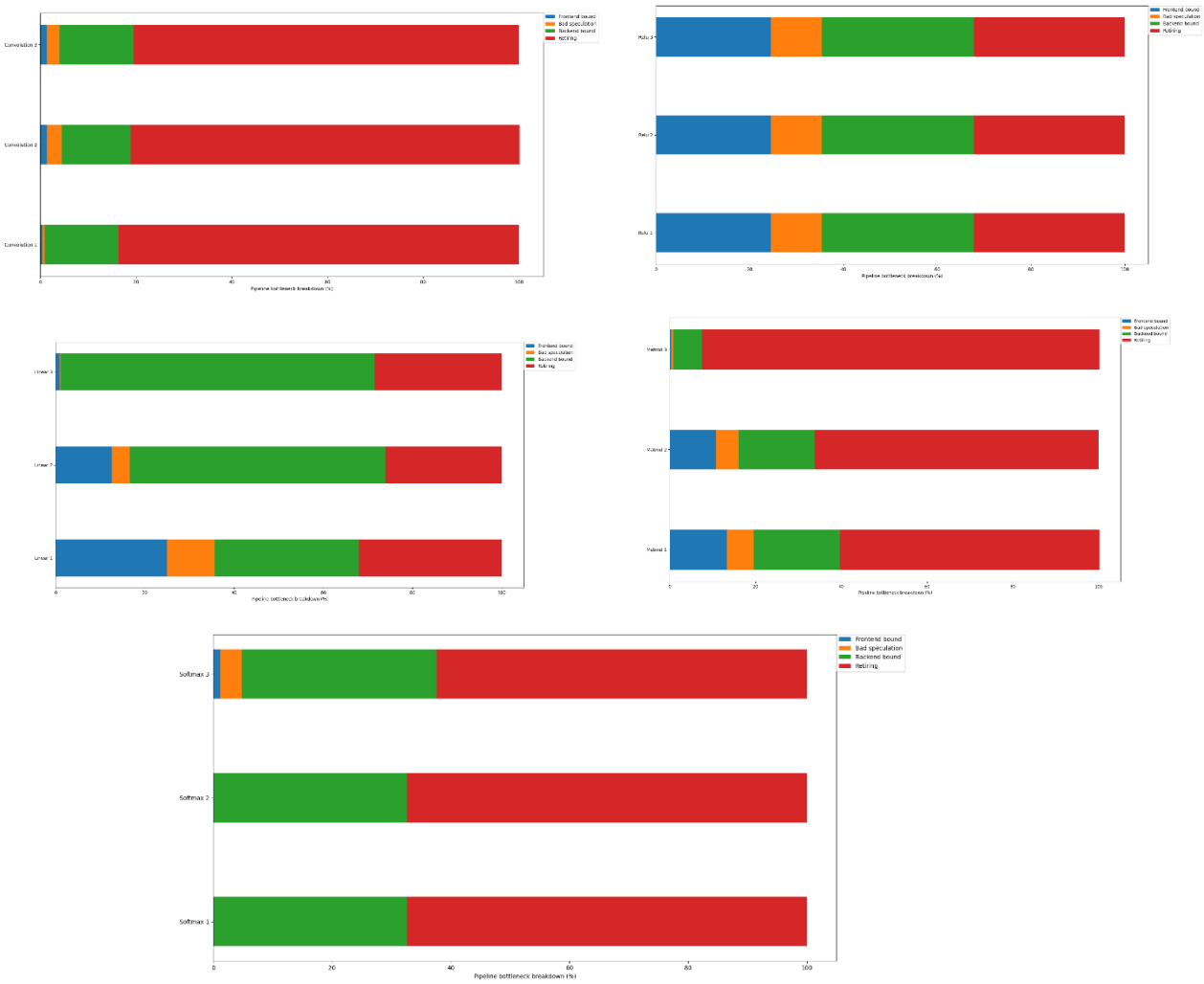
1. Profiling:



The above graphs visualize the profiling data for the five functions: convolution, relu, linear, matmul, and softmax (left to right, top to bottom) for three test cases (input data size increases as test number increases). All the actual function computations in the functions were iterated in a for loop to run thousands to millions of times to nullify the time taken by other operations. This wrapper for loop is not considered below in the function implementation section.

From the data, the major bottleneck for the convolution function is in the backend bound which suggests we can parallelize the function for a more efficient execution. Also, in the second and third test cases, more channels and filters are added in the image and kernel data which results in an increase in bad speculation and frontend bound as one might expect.

For the relu function, there seems to be a significant backlog in all three areas frontend bound, bad speculation, and backend bound when we even run the actual function operations millions of times.

The linear function makes for an interesting case. The first case inputs a very small input and weights matrix data, the second case increases the input data by 1e3 times, while the third case increases the weights matrix by a further 1e3 times. Essentially, the larger the input data, the lesser the frontend inefficiencies in the frontend bound and bad speculation which highlights the fact that either we are not running the operations enough times for the first two cases, or the larger the matrix, the better the branch speculation and frontend bound in the CPU.

The matmul function has a similar story as the linear function for both the test cases data size (increases with test number) and relative backlogs in the different stages (decrease for frontend bound and bad speculation as data size increases). Still, the computations for the matmul function take less time in the ALU (backend bound) than they take for the linear function.

Finally, the softmax function can be optimized for the backend bound through parallelizing. The third case which computes large exponential values though, has non-negligible frontend bound and bad speculation inefficiencies. This probably points to the inefficient computations inside the exponential function when computing large values.

2. Function Implementation

1. ***convolution():*** For this function, I make sure that the input image, kernel, and bias data are valid as well as the respective sizes where none should be negative and kernelSize <= inputSize. Then, I malloc the appropriate space for the convolution output.

   Finally, I compute the output in a 6-step algorithm where in the first 3 steps, I iterate first through the filters and then each cell of the output matrix along the rows and columns. Then I initialize the value of each cell in the output matrix to the value of the corresponding filter bias as that needs to be added to the final output.

   We know now that each output cell of the convolution is a superimposed multiplication of the kernel matrix with the corresponding sub-matrix in the image where the multiplication in the image matrix begins from the current row and column index of the output image. Because the matrices are 2D, we need a 2-step algorithm to compute this product, but because we also have multiple channels in the image and kernel, we wrap the 2-step algorithm in a for loop for the channel to create a 3-step algorithm where all channel computations are summed in the same output cell through the iterations.

   Hence, we finally obtain a 6-step algorithm to compute the convolution.

2. ***relu():*** This is a simple function where I return 0 if the input value is non-positive and the value otherwise.

3. ***linear():*** In this function, I check if the inputs are valid. The function must also check the following two properties which I could not do given the complexity in C and would be much easier if the input was a structure where more data could be stored:
   a. The inputSize should be the same as the rows in the weight matrix.
   b. The outputSize should be the same as the columns of the weight matrix.

   Irrespective, I simply malloc an outputSize float array which is initialized to the bias added in every cell. The output for each cell is then computed as the product of the input array with the corresponding column in the weights matrix.

4. ***matmul():*** This function is a simple linear algebra implementation of matrix multiplication. I first check if the number of columns in the first matrix is equal to the number of rows in the second matrix because if not then we cannot compute the product of the matrices. Then, I malloc the output array whose size is equal to the number of rows in the first matrix times the number of columns in the second matrix. Finally, I compute the matrix multiplication by

iterating through each output cell (matrix size *A_rows x B_cols*) in a 2-step algorithm and then further running a third *for loop* (*A_col* times) to compute the product of each element of the corresponding row of the first matrix (A) with the column of the second matrix (B).

5. ***softmax():*** This is again a simple function where I simply malloc the output array whose size is the same as that of the input array. Then, I compute the exponential probability distribution of the input array by first computing the total exponential sum of all the inputs and then returning each individual ratio with the total sum.

3. Test Implementation

1. ***test_conv():*** For these tests, I have defined several helper functions to malloc the matrices and assign them specific values. I even added debug macros defined in the command line in the makefile to check my output when needed.

   I implemented some functions to test the output when NULL values are passed (should be NULL). Then I implemented three functions named slides (because the input and output were copied from the data in the lab instructions) which respectively test the convolution as follows:

   1. first function tests with one filter and one channel to check that the multiplication of the sliding convolution is correct.
   2. Second function adds the channel dimension to check if the convolution multiplication is properly handled and summed across the different image and kernel channels.
   3. Third function adds the filter dimension to check if the channel iteration is correct.

   Once these tests pass, I am certain of the correctness of the algorithm except for edge cases with max float values where overflows might occur. Assuming that the overflows do not occur, and reasonable values are input in the function, I created a fourth test function that generates random image and kernel data. These values are then tested with the same algorithm implemented in the convolution function which does not serve any purpose for the moment but will do so once the original convolution function is optimized.

2. ***test_linear():*** I created 7 test functions which follow the same idea as for the convolution function out of which 3 check if NULL inputs are treated correctly. The other 4 are as follows:
   a. checks if output of a small input array is computed correctly
   b. checks if output of a very long input array is computed correctly
   c. checks if the output of a very long input array with a very large matrix is computed correctly
   d. random test uses the same function as linear to check against randomly generated inputs

3. ***test_matrix_ops():*** I created 5 test functions which follow the same idea as previous tests:
   a. checks if product of two square matrices of size 2 is computed correctly
   b. checks if product of two rectangular matrices is computed correctly
   c. checks if product of two very large rectangular matrices is computed correctly
   d. checks if product of 2 randomly generated matrices is computed correctly using the same algorithm as implemented in the matmul function.
   e. Checks if incompatible matrix sizes are not accepted.