(Final Phase)CUDA Implementation of Convolution for CNN Inference

Shashwat Johri 12041380
Aditya Mishra 12040050
Shivam Shrivastava 12041400
Suresh Bairwa 12041530

Abstract – In this report for the final phase of project evaluation we will refresh the concept of convolution and the approach used by us inspired by [1], followed by the implementation details, improvements to address the limitations from phase-1 and finally the results.

1 Problem Description

Inputs to a convolution layers come as N batches of 3D inputs with dimensions W,H and D in the directions X, Y and Z respectively. Z direction can be considered as channels (example - RGB channels in an image) with X and Y as the 2D coordinates. A tensor is a set of N 3D inputs.

There are M kernels that act on each of the N batches. Each kernel has the dimension of Wf, Hf, D. The kernel acts upon a batch of input by moving over the input with strides of 1 and for each subset of input that the kernel is over - perform point-wise multiplication followed by summation and a non-linear activation function like ReLU. Each kernel yields a 2D matrix of size W-Wf+1, H-Hf+1 (kernel dimensions are usually smaller than input). M kernels together yield M 2D matrices that are stacked together in the Z direction which are used again as input in further convolution layers. See Figure 1 taken from paper for better understanding.

In this project we have limited the scope of the problem to 1 kernel without any activation function.

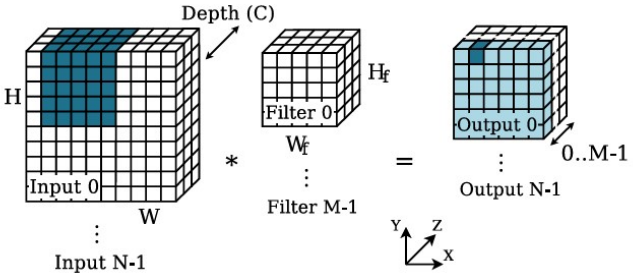

Figure 1: Convolution operations in a convolutional layer. The matrix produced by the convolution of Input 0 with Filter 0 is highlighted in light blue. The darker output element is the result of the dot product of Filter 0 with the highlighted sub-volume of Input 0.

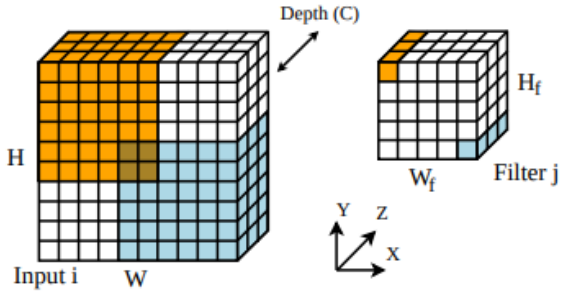

Figure 2: Reuse of input data for two example rows of a filter (highlighted in blue and orange), for a convolution with a stride of 1. The highlighted input rows (two sets of 6x6 rows) are point-wise multiplied with the filter row of the same color during the convolution computation.

2 Approach

The first observation is that each row in the Z direction (hereafter referred to as just a "row" for brevity) in the kernel is dot-producted with various rows in input data (see Figure 2). We will reuse data from filter side by employing a two stage process for convolution. In the first stage we compute all the row-wise dot products between input and kernel rows that are needed. The result of this first step is a set of Hf x Wf temporary matrices, each of size H+Hf-1 x W+Wf-1, as depicted by the scheme from Figure 3.

These Hf x Wf partial results (for one 3D input and one filter) from step one are added using a separate kernel to form the final convolution output.

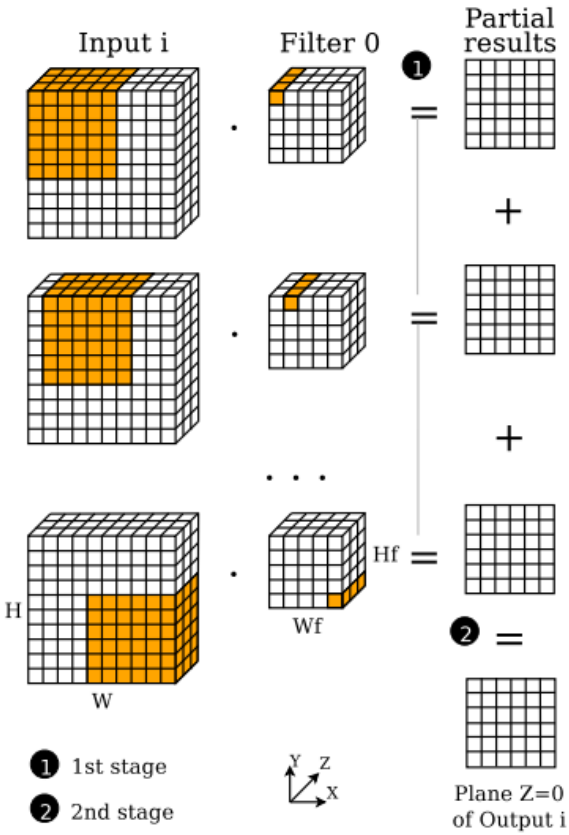

Figure 3: Stages of implementation of convolution depicted for an arbitrary input and the first filter of the convolutional layer. Scalar products in the first stage generate partial results which are aggregated in the second stage to obtain the final output elements.

3 Implementation

3.1 Simple parallel convolution implementation

In stage one Wf x Hf (2D grid) thread blocks are launched with each thread block responsible for the dot products of one filter row with all the rows from the inputs that interact with it during the convolution (the "receptive field" of that row). 32 x 32(2D block) threads are deployed in a thread block. The threads in a thread block firstly work together to bring the block's respective filter row into shared memory.

Afterwards each thread computes dot-product of its block's filter row with one input row in a loop going depth wise, these summations are saved in a temporary matrix with each thread responsible for one element of the matrix, resulting in a W-Wf+1 x H-Hf+1 matrix for one filter row. The result of stage 1 are Wf x Hf (the number of rows) temporary matrices.

In stage two, a 2D grid of thread blocks is deployed of size W-Wf+1 x H-Hf+1 for the summation of the Wf x Hf matrices. Each thread block is responsible for summation at one particular x,y coordinate. The threads in a thread block all work together to perform reduction to achieve the aforementioned final sums. This parallel kernel is run N times to achieve the convolution of NCHW 4D input.

The tensor is taken in the format NCHW in which X(W) direction are stored in contiguous memory locations, then the rows along Y(H) dimension, planes along Z(C) and then the rest of the inputs in 4th dimension(N). In this scheme, the memory accesses to input rows will be almost perfectly coalesced in stage one.

3.2 Tiled convolution implementation

In the above parallel implementation the input size(H x W) is limited by the maximum number of threads allowed in a thread block (32 x 32). This is because in stage 1, each thread is responsible for one input row's dot product with the filter row corresponding to the thread's thread block. In stage 2, the dimension of one temporary matrix is limited by the number of threads. In this section we will talk about tiling implementation to solve this limitation.

In the tiling implementation stage 1, initially 32x32 threads perform dot products in the Z direction with the top left corner of receptive field of the kernel row corresponding to the thread's block. These threads then shift by 32 to cover a new 32x32 input tile and repeat the dot product. This shift is done using two loops in X and Y direction to cover the whole receptive field.

In stage 2 , similar tiling approach is used but for blocks. Initially 32x32 thread blocks perform reduction in the top left corner of the temporary matrices and then move in shifts of 32 to cover the entire size of temporary matrices.

In both the cases edge cases where H/W or H-Hf+1/W-Wf+1 are not divisible by 32 are handled separately.

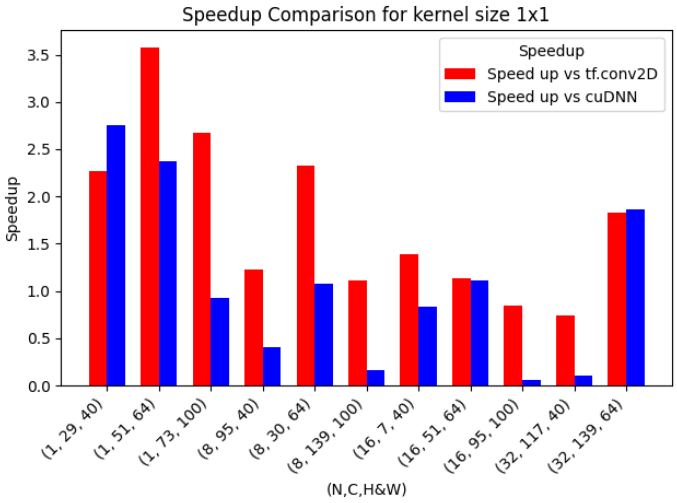

Figure 4: Speedup of our implementation of convolution w.r.t. tf.conv2D(red) and cuDNN algorithm(Blue) for each configuration. Configurations with 1 × 1 filters and batch size up to 32. Labels are formatted as [Batch size N]-[depth C]-[Input size H&W].

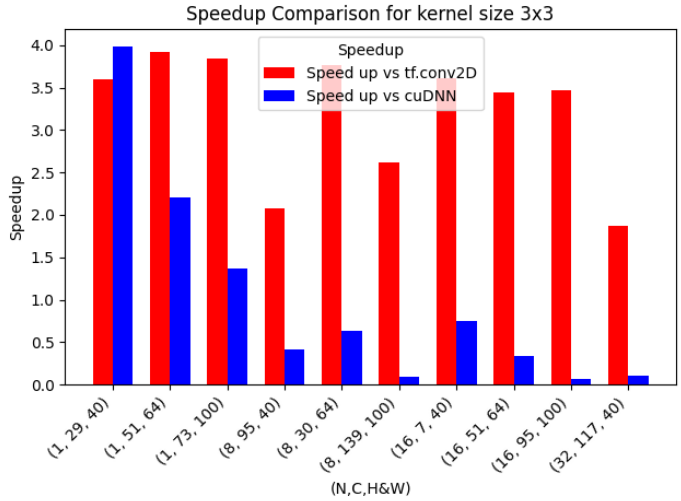

Figure 5: Speedup of our implementation of convolution w.r.t. tf.conv2D(red) and cuDNN algorithm(Blue) for each configuration. Configurations with 2×2 filters and batch size up to 32. Labels are formatted as [Batch size N]-[depth C]-[Input size H&W].

4 Results

The combinations of n,c,h,w are randomly generated with n between [1,32], c between [7,139] and h(and w) between [40,100] for kernels of 1x1 and 3x3. These combinations are run on our implementation, Tensorflow's vanilla sequential convolution using tf.conv2d and CuDNN's convolution implementation which primarily uses implicit-GEMM-based and transform-based convolution algorithms.

Our results agree with the paper's, speedup declines with high values of n,c,h,w for both kernel sizes. See Figures 4 and 5 for visualisation of speedup with different combinations of n,c,h,w.

References

[1] Jordà, M., Valero-Lara, P. & Peña, A.J. cuConv: CUDA implementation of convolution for CNN inference. Cluster Comput 25, 1459–1473 (2022). https://doi.org/10.1007/s10586-021-03494-y