Block Sparse Convolution
CS6023 : GPU Programming

Abhishek Nair (EE16B060)
Anant Shah (EE16B105)

25 November 2018

Problem Description

Convolutional Neural Networks are widely used for image processing applications but have not been majorly tuned based on certain properties of the input. Research has been carried out on speeding up inference durations given block sparse kernels, and desirable results have been achieved [1]. However, block sparsity in the input tensor is a rather unexplored area. If we were to implement a normal convolution algorithm on a block sparse input, it can be seen that we will be wasting resources by performing many unnecessary operations. This problem is highlighted when the sparsity percentage is very large. In this report, we look at different methods to implement block sparse tensor convolution on a GPU.

Existing Work

Recently, the Uber ATG team published a paper [2] in which they showed that the inference time of a CNN is drastically reduced when the inputs are block sparse. Using their open source algorithm(SBNet) which has been integrated into the tensorflow library, they were able to obtain speedups of upto one order of magnitude over the inbuilt dense convolution operation in tensorflow. They have tested the effectiveness of their approach on a 3D Lidar point cloud dataset which is not available in the public domain.

They have defined block sparsity in the tensor using a mask which indicated the locations where the activations are non-zero. Their focus was to convert the sparse tensor into a dense tensor and then apply highly optimized dense convolutions algorithms on that processed tensor. To achieve the same, they defined two operations : a gather operation and a scatter operation. The gather operation takes the non-zero tiles(based on the mask) and then stacks them into a new dense tensor. Highly optimized dense convolution algorithms are then applied to this new tensor. They have used the Winograd algorithm to perform the dense convolution. The scatter operation, an inverse operation, transforms the dense convolution output back to the original form. 1 depicts the sparse convolution operation :

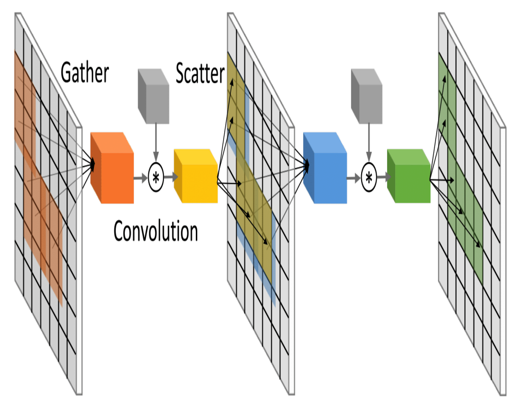

Figure 1: Block Sparse Convolution Operation

Proposed Methodology

We now present the main ideas that were conceptualized for improving the performance of the convolution operation. The first insight is that matrix multiplication based approaches to convolution perform well when the input tensor sizes are small. In the case of sparse input images which we consider, the sparse image can be converted into a matrix with relatively small dimensions. This leads to a speed-up in the matrix multiplication and consequently speeds up the convolution operation as well. Another key insight is that the matrix multiplication based convolution operation can be further divided into three separate operations which must be performed sequentially. These operations are the **im2col**, **matrix multiplication** and **col2im** operations.

Therefore our implementation performs the convolution operation as follows: Given an input sparse tensor, we first perform the im2col operation on it to convert it to a dense matrix. This operation converts the input tensor to a matrix where each row contains all the pixels that affect one pixel of the output tensor. The kernel matrix is also converted to a matrix with each column containing all the elements of a kernel and the number of columns equal to the number of kernels as is described in Figure 2. Note that in Figure 2 the kernel is unravelled along a row and the input tensor is unrolled along a column. We have interchanged these operations in our implementation to ensure that matrix multiplication is coalesced. Next we multiply the kernel matrix and the tensor matrix to obtain the output tensor. This is followed by the col2im operation to convert the output matrix back to the tensor form. Therefore the col2im operation is the inverse of the im2col operation.

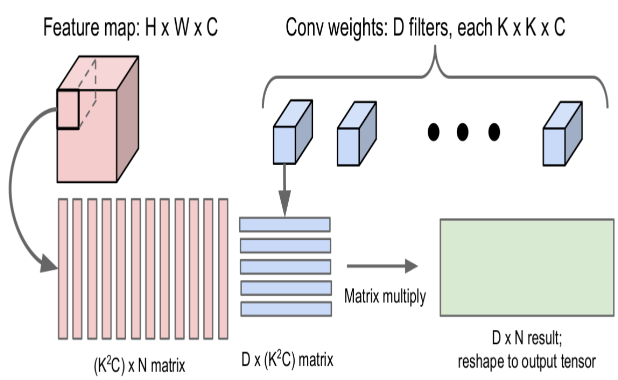

Figure 2: im2col operation

We draw a parallel between the division of the convolution operation into three parts and the division of an instruction pipeline into various stages. Just as pipe-lining exploits instruction level parallelism by executing multiple instructions at the same time, we execute independent kernels concurrently with the help of CUDA streams. We also have the added advantage that we need to perform the same operations over and over again unlike the varied instructions that enter a pipeline. Among the three operations, only matrix multiplication utilizes the compute units of the GPU while the other two only transform the data from one form to another. This means that it is possible to overlap the execution of these kernels as long as they act on different input images. We use CUDA streams to implement this wherein the kernels of one stream run concurrently with kernels of other streams. Another advantage of using streams is that it allows us to overlap data transfer between the host and device with computation on the device using asynchronous memory transfers. A graphical representation of the parallelization present in our algorithm can be found in Figure 3.

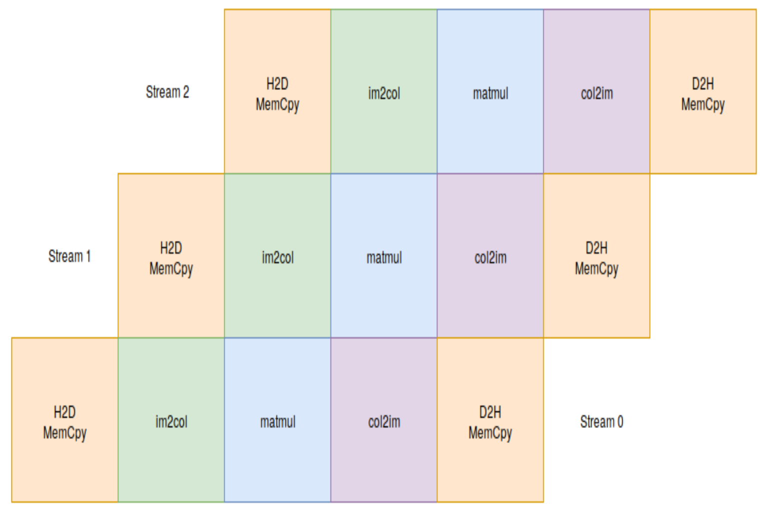

Figure 3: Overlapped Execution of Kernels

Main Results

We now present the salient results obtained from this project. Along with the CUDA kernels we have also developed a serial implementation of the block sparse convolution operation. We use this serial implementation in order to verify the correctness of our CUDA kernels. We planned to use the implementation by Uber (SBNet) as a benchmark for our code, however upon timing their python script we obtain execution times almost two orders of magnitude greater than our implementation. We suspect that the reason for this is the underlying library(tensorflow) that SBNet uses. Tensorflow builds a computational graph with each node representing an operation. The data (input tensor) flows through the graph along the edges and an operation is performed on it whenever it reaches a node. We believe that the process of creating and initializing the computational graph could be the reason why the SBNet module performs so poorly. In order to obtain a fair comparison we plan to directly time the CUDA kernels present in the SBNet implementation. This will be our first step after this project.

Figure 4 below shows how the execution time of our program varies with the size of the input tensor. The results obtained are as expected with an increase in the input tensor size resulting in an increase in the execution time.

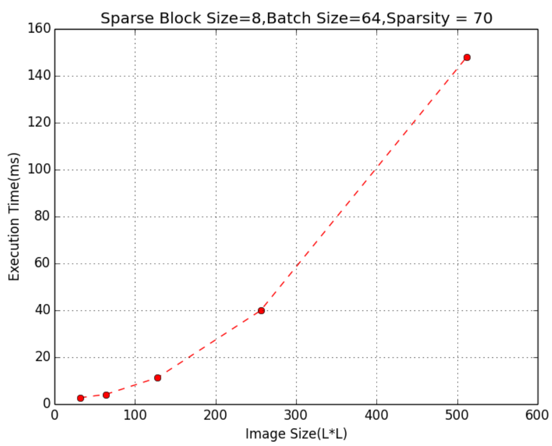

Figure 4: Execution Time vs. input tensor size

Next we plot the execution time of our implementation while keeping the input tensor dimensions the same and varying the percentage sparsity. Here the results we obtain are rather unexpected. As the percentage sparsity is varied from 50% to 90%, the execution time follows no particular order and hardly varies at all. We had expected that due to the decrease in the amount of computation involved, the execution time would decrease for inputs with higher sparsity. Figure 5 below displays the results we have obtained.

Finally we also plot the execution time of the program while varying the size of the sparsity block. As Figure 6 clearly shows, the general trend is that the execution time increases with an increase in the sparsity block size. However when the sparsity block size increases from 2 to 4, there is a decrease in the execution time. This is probably because with a sparsity block size of 2, there would be no output pixel whose input pixels were completely sparse (The size of the kernel is 3). Therefore there would be no reduction in the size of the matrix formed after the im2col operation, leading to an increase in the execution time.

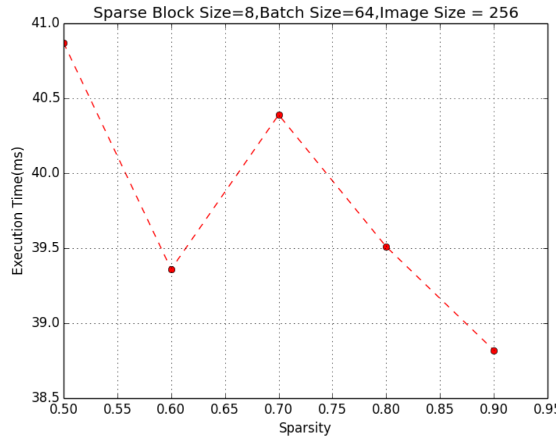

Figure 5: Execution Time vs. sparsity

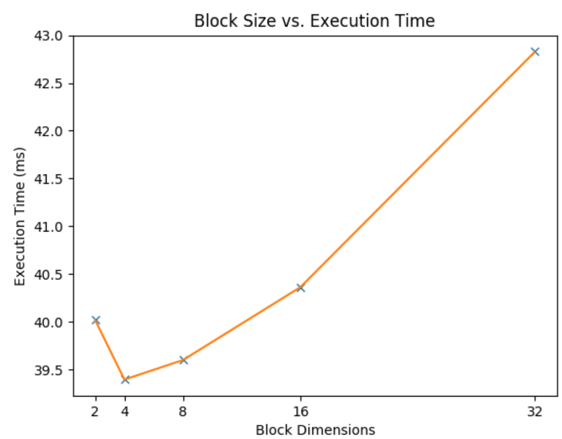

Figure 6: Execution Time vs. sparsity block size

To analyze our application, we used the command line profiling tool nvprof. We observed that majority of the run-time(around 50%) was being spent in the sparsematmul kernel. This is expected as im2col and col2im are transformation kernels while sparsematmul performs the actual computation. The leading dimensions of the matrices formed are also large hence leading to a longer time in the computation. Figure 7 is a graphic, depicting the run-time analysis of the application.

Future Work

There are certain aspects of our implementation which can be improved upon in the future. Some of them are listed down below :

1. The matrix formed from the im2col operation is sparse to some extent. Hence, rather than performing normal matrix multiplication, performing sparse matrix multiplication might give a speed-up. We chose not to implement the sparse-matrix multiplication operation because the sparsity in the matrix is irregular and the sparsity percentage of the matrix is not very high.

2. The im2col kernel has been implemented without the use of shared memory. A speed-up can be obtained if shared memory is used for the im2col operation.

3. We were unable to test our implementation on actual data sets. This was because of the fact that Uber has not released their dataset into the public domain. We used synthetic functions to fill our tensors, kernels and our mask.

4. Replacing our kernel calls with already existing function calls such as im2col from Caffe and matrix multiplication from cuBlas and comparing the performance obtained to our current performance.

5. Integrating our implementation of block sparse tensor convolution in a neural network and check if any improvements are obtained in the inference time.

Main Learning's

We thoroughly enjoyed the experience of attacking an open ended problem statement and developing a solution for it from the ground up. It allowed us to get a glimpse of what it meant to engage in academic research and was very enlightening. We also gained experience in several important tasks such as searching for open source software related to our problem statement, debugging and using open source software as a benchmark and developing open source software. This project also gave us an opportunity to implement some of the advanced concepts we had learnt in the course such as streams and asynchronous memory transfers.

References

[1] Scott Gray, Alec Radford, and Diederik P Kingma. *GPU kernels for block-sparse weights*. Tech. rep. Technical report, OpenAI, 2017.

[2] Mengye Ren et al. “SBNet: Sparse Blocks Network for Fast Inference”. In: *CoRR* abs/1801.02108 (2018).

```
==15== Profiling application: ./blocksparsematmul 256 512 512 3 3 3 1 70 8
==15== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.21%  219.48ms       128  1.7147ms  1.7129ms  1.7159ms  sparsematmul(float*, float*, unsigned int*, int, float*, int, int, int)
                   18.81%  77.604ms       257  301.96us  1.3120us  634.73us  [CUDA memcpy HtoD]
                   12.09%  49.872ms       128  389.63us  389.57us  390.02us  col2im(float*, int*, unsigned int*, float*, int, int, int, int, int)
                    9.42%  38.854ms       128  303.54us  302.47us  305.16us  im2col(float*, int*, float*, unsigned int*, int*, int, int, int, int, int, int, i
nt, int, int)
                    6.46%  26.644ms       128  208.15us  206.24us  251.59us  [CUDA memcpy DtoH]
                    0.00%  3.6480us         2  1.8240us     768ns  2.8800us  [CUDA memset]
      API calls:   50.63%  617.95ms         2  308.98ms  2.2906ms  615.66ms  cudaHostAlloc
                   44.91%  548.11ms       384  1.4274ms  4.1210us  6.1429ms  cudaMemcpyAsync
                    2.86%  34.913ms       384  90.918us  5.2810us  31.657ms  cudaLaunch
                    0.84%  10.267ms         9  1.1408ms  4.5490us  6.5117ms  cudaMalloc
                    0.38%  4.5945ms       128  35.894us  8.3290us  380.77us  cudaStreamCreate
                    0.17%  2.0654ms       128  16.136us  2.4350us  456.87us  cudaStreamDestroy
                    0.08%  964.65us        94  10.262us     202ns  388.12us  cuDeviceGetAttribute
                    0.06%  778.12us      3968     196ns     125ns  9.5760us  cudaSetupArgument
                    0.02%  294.90us         1  294.90us  294.90us  294.90us  cuDeviceTotalMem
                    0.01%  108.65us       384     282ns     128ns  7.2550us  cudaConfigureCall
                    0.01%  92.086us       384     239ns     120ns  5.6160us  cudaGetLastError
                    0.01%  91.478us         1  91.478us  91.478us  91.478us  cuDeviceGetName
                    0.01%  79.152us         1  79.152us  79.152us  79.152us  cudaMemcpy
                    0.01%  61.401us         2  30.700us  6.4460us  54.955us  cudaMemset
                    0.00%  53.033us         2  26.516us  1.3010us  51.732us  cudaEventCreate
                    0.00%  10.560us         2  5.2800us  4.3680us  6.1920us  cudaEventRecord
                    0.00%  7.2830us         1  7.2830us  7.2830us  7.2830us  cudaDeviceSynchronize
                    0.00%  6.2970us         1  6.2970us  6.2970us  6.2970us  cudaEventSynchronize
                    0.00%  2.8520us         1  2.8520us  2.8520us  2.8520us  cudaEventElapsedTime
                    0.00%  2.7610us         2  1.3800us     411ns  2.3500us  cuDeviceGetCount
                    0.00%     646ns         2     323ns     229ns     417ns  cuDeviceGet
```

Figure 7: Profiling