Efficient Sparse Convolutional Neural Network Inference on GPUs

INTRODUCTION

With rising computing complexity and complicated task management, deep neural networks have exploded in popularity in the previous decade. It can be used in a variety of fields like computer vision, robotics, speech recognition, and many others.

In sparse linear algebra, sparse matrix-vector multiplication is critical. Sparse operations, in contrast to dense linear algebra's uniform regularity, deal with a wide range of matrices, from regular to severely irregular. To take use of throughput-oriented processors' enormous potential for sparse operations, we must expose significant fine-grained parallelism and impose adequate regularity on execution routes and memory access patterns. We look into SpMV approaches that are well-suited to throughput-oriented systems such as the GPU and take advantage of a variety of popular sparsity classes.

DNNs use a significant amount of storage, memory bandwidth, and compute resources as their model sizes grow.
Weight pruning has been proposed to address this constraint by compressing DNN models by deleting superfluous connections in the networks. However, while pruning can dramatically reduce model size by deleting an average of 80% of weights, it actually degrades inference performance (i.e. speed) when using GPUs to execute CNN models.

First, as the calculation becomes sparse after pruning, the expense of lowering convolution onto matrix multiplication becomes a serious issue. The lowering method has shown overhead for dense convolution since it repeats input features several times, wasting memory bandwidth and limiting data reuse chances.

Second, sparse matrix computation on GPUs is substantially less efficient than dense matrix calculation.

Although sparse matrix multiplication avoids superfluous MAC operations, its memory access pattern is extremely erratic, limiting its ability to fully utilize the GPU architecture's computational power. Furthermore, while sparse matrix computing employing compressed data structures may reduce memory space, decoding the sparse format at runtime incurs overhead.

Escort, an effective sparse CNN approach adapted for GPU's data-parallel architecture, was created to circumvent the restrictions. Rather than lowering the convolution to matrix multiplication, we compute the sparse convolution directly. We modify the dataflow and use a number of optimization approaches based on an understanding of the memory access pattern to take advantage of the GPU's immense computational horsepower.

Escort increases arithmetic intensity by directly computing sparse convolution rather than lowering it to matrix multiplication, and it is specifically tuned for the GPU architecture by taking advantage of the parallelism available.

SPARSIFY

A sparse matrix structure is used to store the weights. We dynamically compute the offset of the input array as the computation progresses, and then use the index to load the proper items into on-chip storage. We do the required index calculation after the computation is completed in order to ave the result in the correct output location.

A sparse matrix structure is used to store the weights. The compressed sparse row (CSR) format, as demonstrated in Fig. 1 of this study, is frequently used to store sparse weight matrices. Three arrays make up the CSR data structure. Row each row, the data array value saves just the non-zero elements. Two auxiliary data structures are added to determine the initial location of each non-zero element. The colidx column-indices array includes nnz integers (nnz is the total number of non-zero elements), and the colidx[i] entry shows the column id of the ith element in value. The row-pointers array rowptr includes M + 1 (M is the number of rows in the matrix) integers, with entry rowptr[i] being the ith row's initial index in colidx. This means that rowptr[i + 1] is the number of non-zero elements in the ith row, and rowptr[i] is the number of non-zero elements in the ith row.

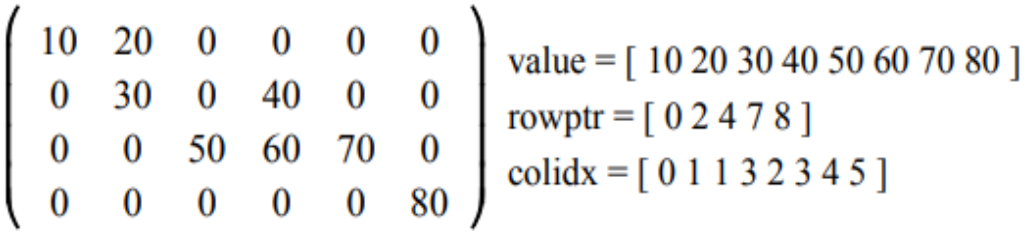

Figure 1: An example of the compressed sparse row (CSR) format. Source: Xuhao Chen (2019)

```
void sparesify(int* matrix)
{
    value=(int*)malloc(sizeof(int)*(W*W));
    rowptr=(int*)malloc(sizeof(int)*(W+1));
    colidx=(int*)malloc(sizeof(int)*(W*W));
    int NNZ = 0;
    rowptr[0]=0;
    for (int i = 0; i < W; i++) {
        for (int j = 0; j < W; j++) {
            if (matrix[i*W+j] != 0) {
                value[position_value++] = matrix[i*W+j];
                colidx[position_colidx++] = j;
                NNZ++;
            }
        }
        rowptr[position_rowptr++] = NNZ;
    }
}
```

Figure 2: Sparsify Function implemented
Source: Xuhao Chen (2019)

SERIAL IMPLEMENTATION:

The above-mentioned lowering method has its own drawbacks, since the decreased matrix implementation can be expensive if the GPU memory amount used for the implementation is constrained. To overcome this, we attempted to implement the sequential technique 2 described in the study, which involves loading the input feature matrix into the gpu's cache memory at runtime.

In this scenario, we'll employ a technique known as dynamic indexing. As the computation advances, we dynamically compute the offset of the input array, and then use the index to load the appropriate elements into on-chip storage. After the computation is performed, we perform the required index calculation in order to save the result in the correct output place.

Algorithm 2 Sequential Sparse Convolution

```
1:  procedure SCONV(in, W, out)
2:      for n in [0, N) do
3:          for m in [0, M) do
4:              for j in [W.rowptr[m], W.rowptr[m+1]) do
5:                  off ← W.colidx[j]
6:                  val ← W.value[j]
7:                  for h in [0, E) do
8:                      for w in [0, F) do
9:                          out[n][m][h][w] += val *
10:                                 in[n][off + f(0, h, w)]
```

Figure 3: Sequential Sparse Convolution
Source: Xuhao Chen (2019)

As discussed in the paper and declared as a global variable accessible by all GPU and CPU functions.The weight matrix sparsing has been implemented in the sparsify function which is stretching the weight matrix beforehand to match the dimension of the input array.
This is only run once and is preprocessed when creating the sparse weight matrix (i.e. the CSR data structures). This preprocessing step is known as weight stretching. This action solely affects the weight matrix's column indices, which are stored in the colidx array. There is no extra memory usage.
For each ofmap and each output channel in the ofmap the algorithm has been designed to get the value of offset and weight from the CSR data structure in a row major order and maps the output to the correct index in the output.

```
int *SPConv_serial(int input_features[H*H], int weight[], int window_size)
{
   int j1, j2;
   //sparesify(weight);

   int *output_matrix = (int *)calloc(window_size*window_size, sizeof(int *));

   for (int i=0;i<position_rowptr-1;i++)
   {
          int row = i;
          for (int j=rowptr[i];j<rowptr[i+1];j++)
          {
              int mat_value = value[j];
              int col = colidx[j];
              for (j1=0; j1<window_size;j1++)
              {
                  for (j2=0; j2<window_size;j2++)
                  {
                      output_matrix[j1*window_size + j2] = output_matrix[j1*window_size + j2]  + input_features[(j1+row)*H + (j2+col)] * mat_value;
                  }

              }
          }
   }
```

Figure 4:Code implemented for convolution.
Source: Xuhao Chen (2019)

UNOPTIMIZED PARALLEL IMPLEMENTATION

If the dataflow is not properly planned for the underlying architecture, a straightforward implementation of Algorithm 2 is not necessarily efficient on GPUs. In normal sparse-matrix computations on GPUs, for example, noncontiguous indirect memory access represents a significant expense. To preserve memory bandwidth, memory requests from consecutive threads in a warp are merged into one or more memory transactions. Memory divergence occurs otherwise, and the GPU memory subsystem's efficiency plummets.

```
int tpb = 8;
int bpg = (tpb + position_rowptr - 1) / tpb;
conv << <bpg, tpb, position_rowptr *sizeof(int) >> > (d_ifmaps, d_value, d_rowptr, d_colidx, d_of, op_size);

cudaMemcpy(d_ofmaps, d_of, op_size * op_size * sizeof(int), cudaMemcpyDeviceToHost);
```

Fig 5: Calling of unoptimised conv function

We used the warp size of 8 as specified in the assignment.
In our scenario, the number of threads started is equal to the number of non zero items in the map of the CSR data structure.
The partial sum is saved in the p_sum variable, which holds the value of each thread computation and then completes the operation by including all of the values and returning the total value.

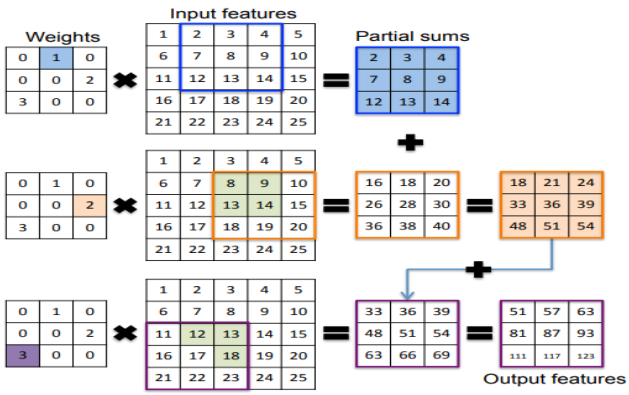

Figure 6:Example for data use.
Source: Xuhao Chen (2019)

PARALLEL IMPLEMENTATION WITH WARP-SIZE 8

This section discusses a parallel version of Algorithm 2 with a warp-size of 8. The parallel processing advantage of GPUs is not completely leveraged if the dataflow is not carefully tailored for the underlying technology. To avoid memory divergence, the dataflow for taking advantage of parallelization is chosen. If multiple threads in a warp access the same memory address at the same time, the memory requests are merged into one or more memory transactions to save memory bandwidth. Otherwise, memory divergence occurs, and the GPU memory subsystem's efficiency plummets.

To reduce memory divergence, we use a dataflow. Figure 2 depicts the basic concept. The sparse convolution of a 3x3 filter with a 6x6 input feature can be split into two parts: the nonzero weight "2" in the filter times

a 4x4 sub-matrix (blue), and the other nonzero weight "3" in the filter times another 4x4 sub-matrix (red) . The final results are then produced by simply adding the two products together. When the multiplications are done independently, unstructured computation is avoided.

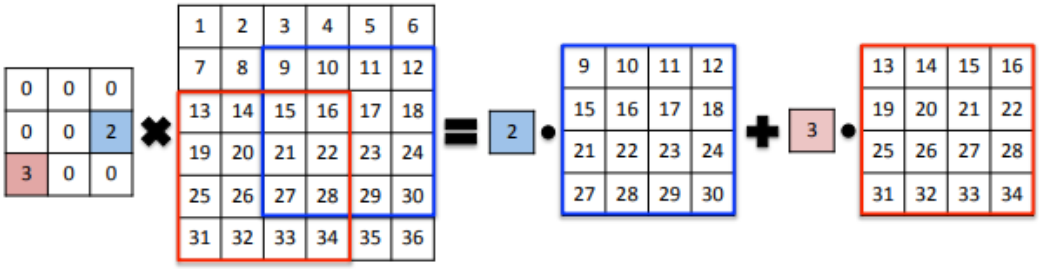

Fig 7: An example sparse convolution of one 3×3 filter against an 6×6 input feature with 1 channel.
Source: Xuhao Chen (2019)

Figure 8 depicts the data-to-thread mapping on the GPU. Assuming a four-thread warp, accesses to the input array by a warp are consolidated as long as the array elements have contiguous row or column indices. In the output matrix, each thread is responsible for calculating one output element. Because consecutive threads are assigned to calculate consecutive output positions, the accesses to the product sum into the output array are also contiguous. Each non-zero weight is multiplied by the partial sum of the corresponding output element assigned to the thread for each non-zero weight.

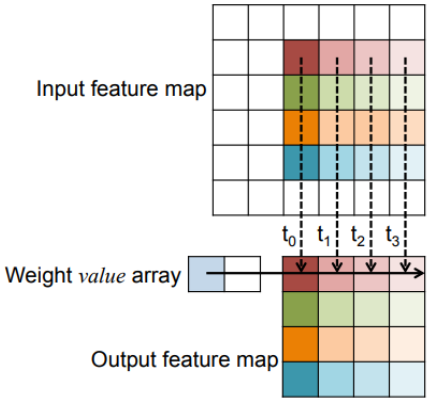

Fig 8: Data-to-thread mapping for a pseudo 4-thread warp.(Source: Xuhao Chen (2019))

The following code snippet depicts the parallelization part of Algorithm 2, with warp size is warp_size = 8.

```
__global__ void conv_mul_parallel(int* d_if, int *val, int *row, int * col_id, int * row_id, int* d_of, int window_size)
{
    extern __shared__ int psum[] ;
    __shared__ int active_tid;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid == 0)
    {
        for (int idx = 1; idx < position_rowptr; idx++)
            psum[idx] = 0;
    }

    if (tid <= position_rowptr - 1)
    {
        for (int k=0; k < position_value; k++)
        {
            int mat_value = val[k];
            int row = row_id[k];
            int col = col_id[k];

            for (int j1 = 0; j1 < window_size; j1++)
            {
                for (int j2=0; j2 < window_size; j2++)
                {
                    if (tid == j2 % warp_size)
                    {
                        active_tid = tid;
                        psum[tid] = d_if[(row + j1) * H + (col + j2)] * mat_value;
                    }

                    __syncthreads();
                    d_of[j1 * window_size + active_tid] += psum[active_tid];

                }

            }
        }
```

Fig 9: Implementation of optimised convolution function

PROFILING

To capture reuse, we have three forms of dataflow:
1) The goal of Weight Stationary is to reduce the overhead of loading weights by increasing weight accesses in the on-chip cache.
2) Output Stationary is designed to reduce the time spent reading and writing partial amounts. It stores partial sum accumulation in the on-chip cache, streams input features across the processor, and broadcasts weights.
3) Input Stationary reduces the overhead of reading inputs by caching the input characteristics and streaming the weights.

```
__global__ void compute(unsigned int n)
{
    extern __shared__ float arr[];
    // Ref: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared
    float *params = (float *)arr;

    int tid = blockIdx.x * blockDim.x + threadIdx.x,
    toffset = gridDim.x * blockDim.x * blockIdx.y;
    for (int i=threadIdx.x; i<NUM_PARAMS; i+=blockDim.x) {
        params[i] = d_computeParams[i];
    }
    __syncthreads();

    float *w = (float *)&arr[sizeof(int)*NUM_PARAMS];
    int *cols = (int *)&arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(params[9])];
    int *rows = (int *)&arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(params[9]) + sizeof(int)*(params[10])];
    float *ip = (float *)&arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(params[9]) + sizeof(int)*(params[10]) + sizeof(int)*(params[11])];
    float *op = (float *)&arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(params[9]) + sizeof(int)*(params[10]) + sizeof(int)*(params[11]) + sizeof(float)*(params[1] *
params[2] * params[3] )];
    /*
       grid(X, Y, Z): dim3(E*F/BLOCK_DIM, M, 1)
       Y -> diff output layer
       X -> ceil(E*F/BLOCK_DIM)
    */

    for (int i=threadIdx.x; i < params[9]; i += blockDim.x) {
        w[i] = d_weights[i];
    } // Load weights

    for (int i=threadIdx.x; i < params[11]; i += blockDim.x) {
        rows[i] = d_rowIdx[i];
    } // Load rows

    for (int i=threadIdx.x; i < params[10]; i += blockDim.x) {
        cols[i] = d_colIdx[i];
    } // Load cols

    for (int i=threadIdx.x; i < params[1] * params[2] * params[3] ; i += blockDim.x) {
        ip[i] = d_ifmap[i];
    } // Load inputs
    __syncthreads();
```

Figure 10: Global memory coalescing and data locality

```
if (tid < params[6] * params[7]) {
    float psum = 0;
    for (int c_idx=rows[blockIdx.y]; c_idx<rows[blockIdx.y+1]; ++c_idx) {
        float val = w[colIdx];
        int col = cols[c_idx];
        int j = col % (params[4] * params[5]); // point: (j%S, j/S)
        for (int i_idx = 0; i_idx < C; ++i_idx) {
            // top-left corner of weight matrix
            int i_val = i_idx * params[2] * params[3] + (params[8])*(params[3]*(tid/params[7]) + tid%params[7]); // H*W*c + stride*(W*(tid/F) + tid%F)
            psum += ip[i_val + params[3]*(j/params[5]) + (j%params[5])] * val;
        }
    }
    op[tid + params[6] * params[7] * blockIdx.y] = psum;
}
__syncthreads();
```

Figure 11: Parallelized Convolution

```
for (int i=tid; i<(params[0] * params[4] * params[5] )/gridDim.y; ++i) {
    d_ofmap[i] = op[i];
}
__syncthreads();
```

Figure 12: Global Write Coalescing

The Alexnet is made up of eight layers, each having its own set of learnable parameters. The first convolution layer is of size 11X11, second layer is reduced to 5X5, followed by 3X3. The network also adds maximum pooling layers with a window shape of 3X3 and a stride of 2 after the first, second, and fifth convolutional layers.
We are using a python script to import the weights and biases of a pretrained AlexNet's convolutional layers, pruning 80% of the weights and saving them in text files. These text files are then read by a host C program and fed into the sparse convolution optimised CUDA code.

```python
import torch
import torch.nn as nn
import torch.nn.utils.prune as prune

model = torch.hub.load('pytorch/vision:v0.10.0', 'alexnet', pretrained=True)

#the model parameters
print(model.parameters)

#pruning the convolutational neural networks of the pretrained model
idx = []
for i,module in enumerate(model.features.modules()):
    if isinstance(module,nn.Conv2d):
        prune.l1_unstructured(module,name="weight",amount=0.8)
        idx.append(i)

print(idx)

#we store the pruned weights in a txt file to be read later by the
for i in idx:
    weights = model.features[i-1].weight.flatten().tolist()
    bias = model.features[i-1].bias.tolist()
    with open("Conv2D_layer" + str(i) +".txt","w") as f:
        f.write(" ".join(list(map(str,list(model.features[i-1].weight.shape)))))
        f.write("\n")
        f.write(" ".join(list(map(str,bias))))
        f.write("\n")
        f.write(" ".join(list(map(str,weights))))
```

Fig 13: Code to get the network parameters

REFERENCES

1) Xuhao Chen. 2019. Escoin: Efficient Sparse Convolutional Neural Network Inference on GPUs. In Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17), 9 pages
2) https://d2l.ai/chapter_convolutional-modern/alexnet.html
3) Xuhao Chen, Cheng Chen, Jie Shen, Jianbin Fang, Tao Tang, Canqun Yang, and Zhiying Wang. 2017. Orchestrating parallel detection of strongly connected components on GPUs. Parallel Comput. (2017)
4) Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. International Conference on Learning Representations (ICLR) (2016).
5) Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. CoRR abs/1410.0759 (2014)
6) Fred´ eric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfel- ´ low, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. CoRR abs/1211.5590 (2012).
7) Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse MatrixMatrix Multiplication for the GPU. ACM Trans. Math. Softw. 41, 4, Article 25 (Oct. 2015), 20 pages
8) L. Deng, J. Li, J. T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, Y. Gong, and A. Acero. 2013. Recent advances in deep learning for speech research at Microsoft. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 8604–8608.
9) Jongsoo Park, Sheng R. Li, Wei Wen, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Holistic SparseCNN: Forging the Trident of Accuracy, Speed, and Size. CoRR abs/1608.01409 (2016).