



CS61064 : CUDA End Term Project

Efficient Sparse
Convolutional Neural Network
Inference

(Group G and Q)

INDEX

INTRODUCTION

SPARSIFY FUNCTION

SERIAL IMPLEMENTATION

PARALLEL UNOPTIMISED
IMPLEMENTATION

PARALLEL OPTIMISED IMPLEMENTATION

PROFILING VIA CNN PIPELINE

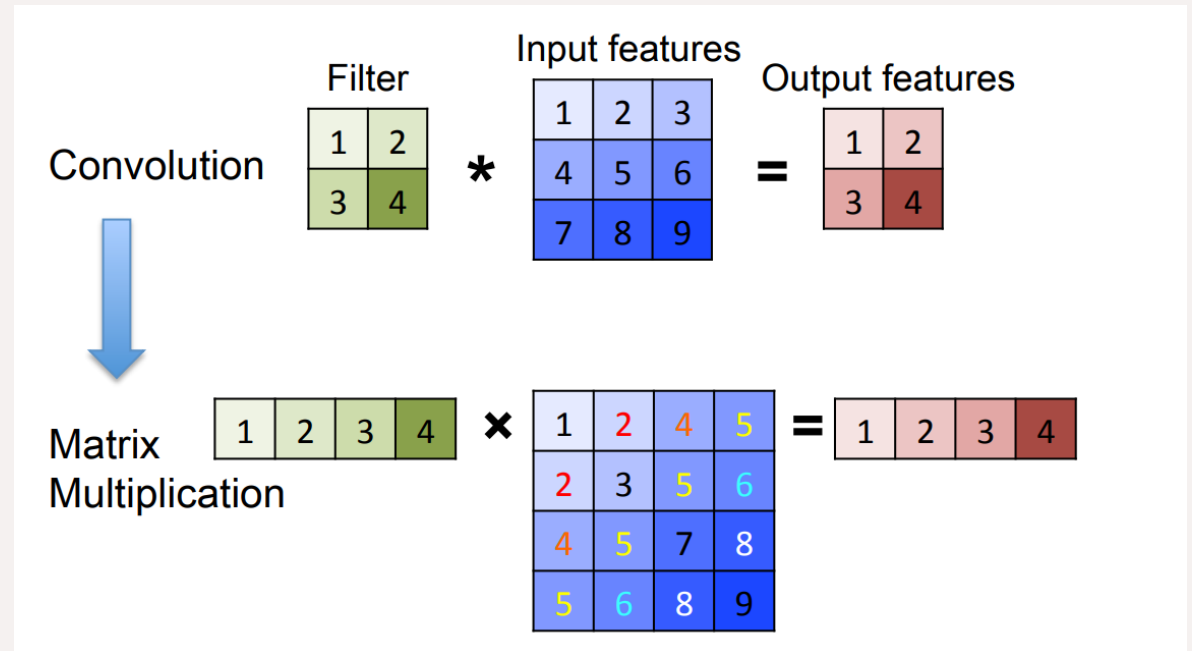
Introduction

Weight pruning can compress DNN models by removing redundant parameters in the networks, but it brings sparsity in the weight matrix and therefore makes the computation inefficient on massively parallel GPUs.

Limitations of pruning

Convolution onto matrix multiplication is diminished, which lowers data reuse and consumes memory bandwidth.

When executing on massively parallel GPUs, the sparsity introduced by pruning causes the computation to be irregular, resulting in inefficiency.



Solution

Escort: An efficient GPU-based sparse convolutional neural network. We choose to compute the sparse convolutions directly rather than utilizing the lowering approach. The parallelism and locality for the direct sparse convolution kernel are then coordinated, and we use specific optimization approaches to increase performance even more.

IMPLEMENTATION



Using the Sparsify function, the weight matrix is represented in CSR format.



Serial implementation of direct sparse convolution.



Unoptimized parallel implementation of serial implementation.



Optimization of parallel implementation using Memory Coalescing and Data Localization.

SPARSIFY FUNCTION

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 save the result in the correct output location.

```
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;
    }
}
```

Compressed Sparse Row format

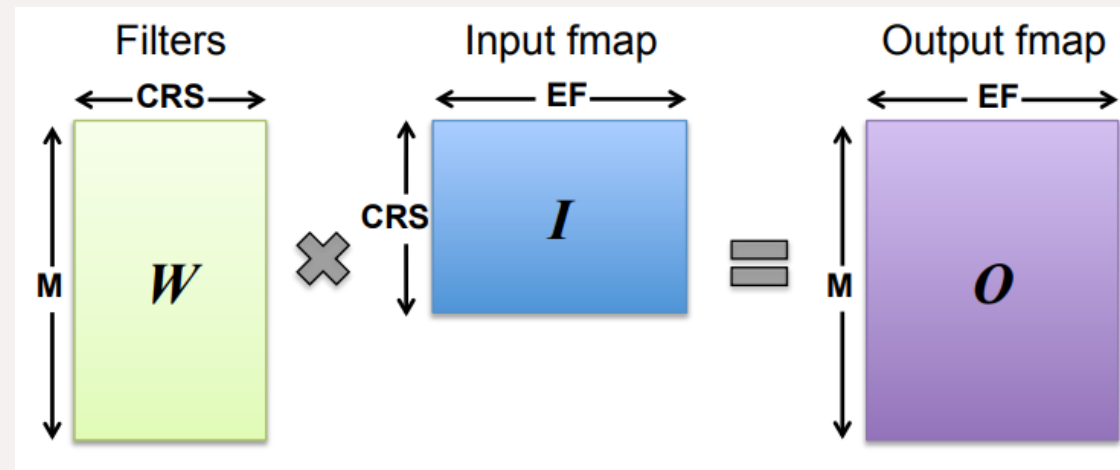
$$\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

value = [10 20 30 40 50 60 70 80]
rowptr = [0 2 4 7 8]
colidx = [0 1 1 3 2 3 4 5]

Computation of output feature maps from input feature maps using matrix multiplication

Shape Parameter	Description
N	batch size
M	# of filters / # of ofmap channels
C	# of ifmap/filter channels
H/W	ifmap height/width
R/S	filter height/width
E/F	ofmap height/width

Table 1: Shape Parameters of a CONV Layer [42]



Pseudo Code for Sequential Convolution

Algorithm 2 Sequential Sparse Convolution [37]

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

Serial Implementation Code

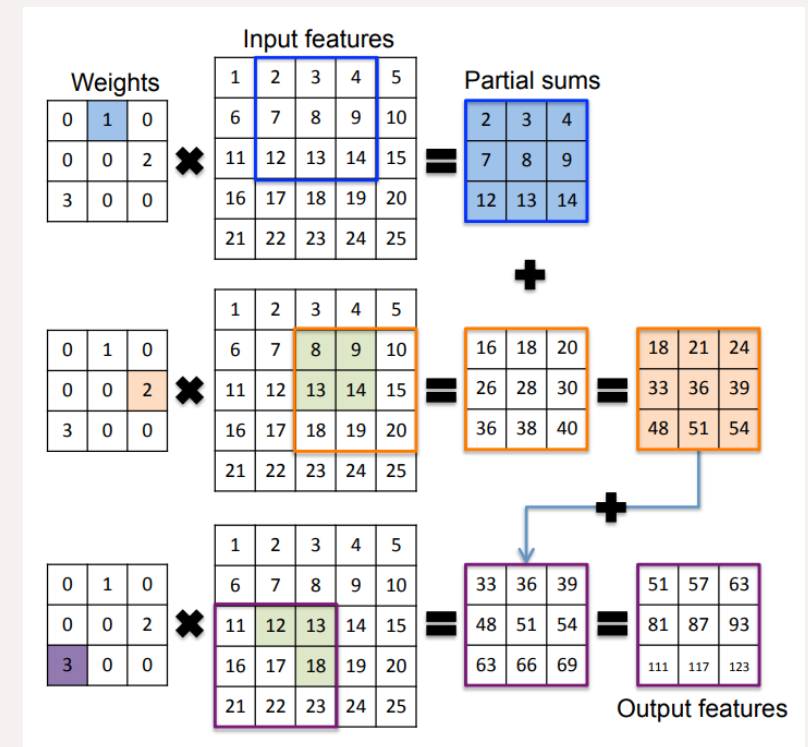
```
✓ int *SPConv_serial(int input_features[H*H], int weight[], int window_size)
{
    int j1, j2;
    sparesify(weight);
    int *output_matrix = (int *)calloc(window_size*window_size, sizeof(int *));

    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;
                }
            }
        }
    }

    return output_matrix;
}
```

PARALLEL UNOPTIMISED IMPLEMENTATION

- We have considered the warp size to be 8 as given in the assignment.
- The number of threads that has been launched in our case is equivalent to the number of non-zero elements in the map of the CSR data structure.
- The partial sum is stored in the p_sum variable which holds the value of each thread computation and finally finishes the operation after it involves all the values and returns the final value.



Parallel Unoptimized Implementation Code

```
__global__ void conv(int* d_if, int *val, int *row, int * col, 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)
    {
        int r = tid;
        for (int j = row[tid]; j < row[tid + 1]; j++)
        {
            int mat_value = val[j];
            int c = col[j];
            printf("tid = %d row = %d col = %d val = %d\n", tid, r, c, mat_value);
            for (int j1 = 0; j1 < window_size; j1++)
            {
                for (int j2 = 0; j2 < window_size; j2++)
                {
                    active_tid = tid;

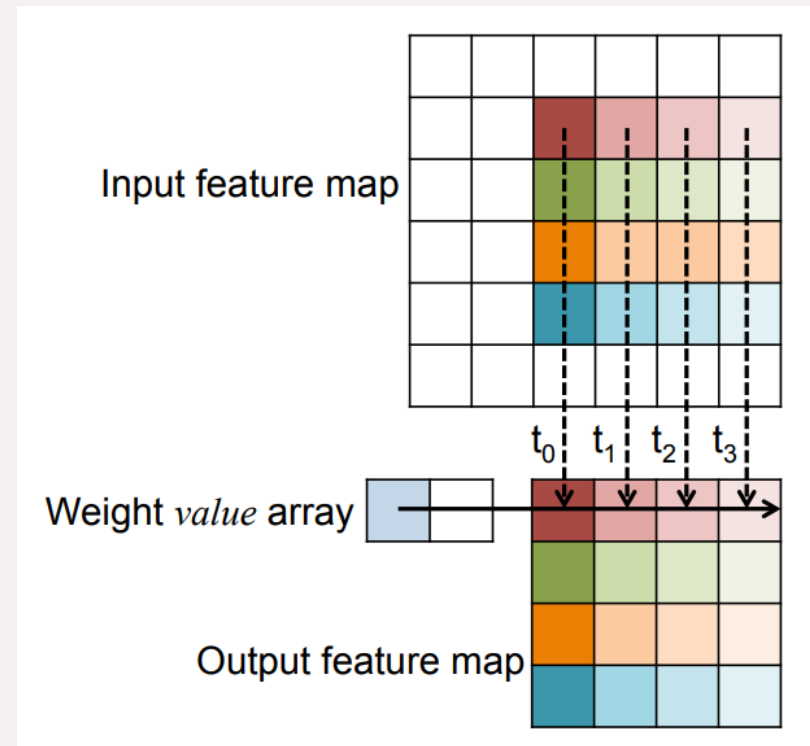
                    psum[tid] = d_if[(j1 + r) * H + (j2 + c)] * mat_value;

                    __syncthreads();

                    if (tid == active_tid)
                    {
                        for (int idx = 1; idx < position_rowptr; idx++)
                        {
                            psum[0] += psum[idx];
                        }
                        d_of[j1 * window_size + j2] = psum[0];
                        if (tid == active_tid)
                        {
                            for (int idx = 0; idx < position_rowptr; idx++)
                            {
                                psum[idx] = 0;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Parallel Implementation: Warp Size 8

- The accesses to the input array by a warp are coalesced as long as the array elements with contiguous row or column indices are stored contiguously.
- Each thread is responsible to calculate one corresponding output element of the output matrix.
- It is to be noted that the memory accesses required to write the product sum into the output array are also contiguous.



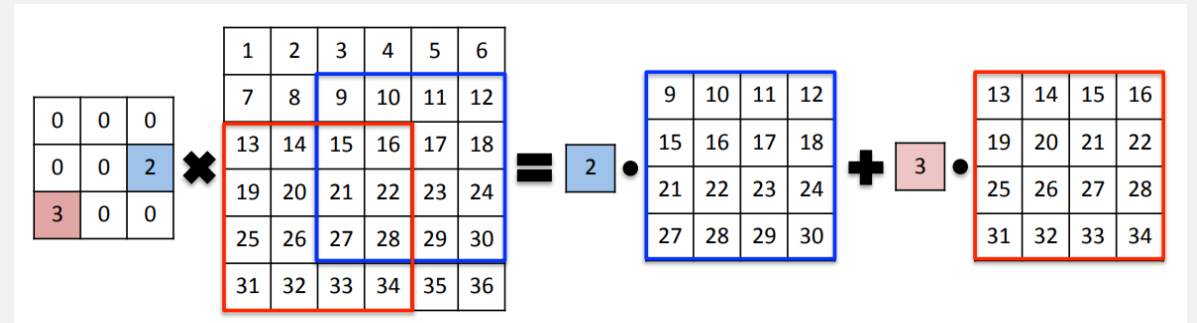
Algorithm for implementation

- Each of the non-zero weights present in the matrix are multiplied with consecutive input data.
- The corresponding product obtained thereafter is added to the partial sum for the output element pertaining to the thread.
- This helps us to avoid uncoalesced memory access to the global memory in the GPU, hence improving memory access efficiency.

```
__global__ void conv_mul_parallel(int* d_if, int *val, int *row, int * col_id, int * row_id,  
{  
    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];  
  
            // printf("\n mat-value: %d", mat_value);  
            // printf("\n row num: %d", row);  
            // printf("\n col num: %d\n", col);  
  
            for (int j1 = 0; j1 < window_size; j1++)  
            {  
                for (int j2=0; j2 < window_size; j2++)  
                {  
                    if (tid == j2 % warp_size)  
                    {  
                        // printf("\nActive threadID: %d\n", j2);  
                        active_tid = tid;  
                        psum[tid] = d_if[(row + j1) * H + (col + j2)] * mat_value;  
                        // printf("\npsum value: %d\n", psum[tid]);  
                    }  
  
                    __syncthreads();  
                    d_of[j1 * window_size + active_tid] += psum[active_tid];  
                }  
            }  
        }  
    }  
}
```

Dataflow in the underlying architecture

- The purpose of an optimum dataflow here is to minimize memory divergence.
- The sparse convolution of a 3x3 filter against a 6x6 input feature map can be divided as shown in the figure.
- The results can be obtained by simply accumulating the two products, avoiding any unstructured computations.



PARALLEL OPTIMISED IMPLEMENTATION

We load the weight matrix, which is saved in CSR format, into shared memory, together with the input feature map, output feature map, and the parameters associated with the matrices, to improve the parallel CUDA code. Because we only load successive array members, they're all merged memory accesses.

Output channels are mapped to the y axis of the grids. while the x-axis contains the $E \times F$ number of threads that process the m^{th} output layer. This was done because of the limit on the maximum number of threads in a block at 1024 threads.

Global Memory Coalescing and Data locality

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

    int tid = blockIdx.x * blockDim.x + threadIdx.x,
        teffset = blockDim.x * blockDim.x * blockIdx.y;
    for ([tid]=threadIdx.x; tid<NUM_PARAMS; tid+=blockDim.x) {
        param[tid] = d_computeParam[tid];
    }
    __syncthreads();

    float *w = (float *)arr[sizeof(int)*NUM_PARAMS];
    int *cols = (int *)arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(param[9])];
    int *rows = (int *)arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(param[9]) + sizeof(int)*(param[10])];
    float *ip = (float *)arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(param[9]) + sizeof(int)*(param[10]) + sizeof(int)*(param[11])];
    float *op = (float *)arr[sizeof(int)*NUM_PARAMS + sizeof(float)*(param[9]) + sizeof(int)*(param[10]) + sizeof(int)*(param[11]) + sizeof(float)*(param[1]) +
        param[2] * param[3] );

    /*
    grid(X, Y, Z): dim3(E*F/BLOCK_DIM, M, 1)
    X -> diff output layer
    X -> cell(E*F/BLOCK_DIM)
    */

    for ([tid]=threadIdx.x; tid < param[9]; tid += blockDim.x) {
        w[tid] = d_weights[tid];
    } // load weights

    for ([tid]=threadIdx.x; tid < param[11]; tid += blockDim.x) {
        rows[tid] = d_rowIdx[tid];
    } // load rows

    for ([tid]=threadIdx.x; tid < param[10]; tid += blockDim.x) {
        cols[tid] = d_colIdx[tid];
    } // load cols

    for ([tid]=threadIdx.x; tid < param[1] * param[2] * param[3]; tid += blockDim.x) {
        ip[tid] = d_ip[tid];
    } // load inputs
    __syncthreads();
}
```

```

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();

```

Parallelized Convolution

```

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

```

Global Write Coalescing

The CNN pipeline

For the CNN pipeline, we employ a Python script that imports the weights and biases of a pre-trained AlexNet's convolutional layers, trimming 80% of the weights and saving them in text files, which are then read by a host C program and fed into the optimized CUDA code that conducts the sparse convolution.

```
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 convolutional 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)

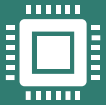
print(idx)

#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))))
```

Current issues and future improvements



We used this [reference](#) to implement data locality. However we faced several issues while running the code without hardcoding in the length of the array stored in the shared memory on Google Colab. This might have been a CUDA version issue. ([refer](#))



Future improvements would involve reduction in the shared memory storage by only loading a part of the arrays involved at once as a function of `gridDim.y`. For example, we can get away with loading weight and `colidx` from the indices `[rowidx[gridDim.y], rowidx[gridDim.y+1])` and load the `gridDim.y` th layer of the output array.



We can also implement the other layers used such as maxpooling and the ReLU activation layers using CUDA and streamline the entire inference pipeline.

Contributions



1


Implement the Sequential Sparse Convolution as described in Algorithm 2 in the paper: **(21CD72P01, 21CD92P01, 21CD92R01)**

2

Apply sparse convolution of 3×3 filter and the parallelism strategy considering the warp size as 8: **(21CS91P01, 20CE30034, 20CE30035, 20CH10089)**

3

Minimize coalesced memory accesses and maximize data locality for optimized performance. Give a detailed analysis for the optimizations with an example of AlexNet CNN inference pipeline. **(20EC10034, 20EC10072, 20CH10087)**



The top of the image features a decorative border with a dark red background. It contains several overlapping semi-circular shapes. Some of these shapes are filled with a pattern of concentric dotted lines, while others are solid dark red. The patterns are arranged in a way that creates a sense of depth and movement.

THANK YOU !