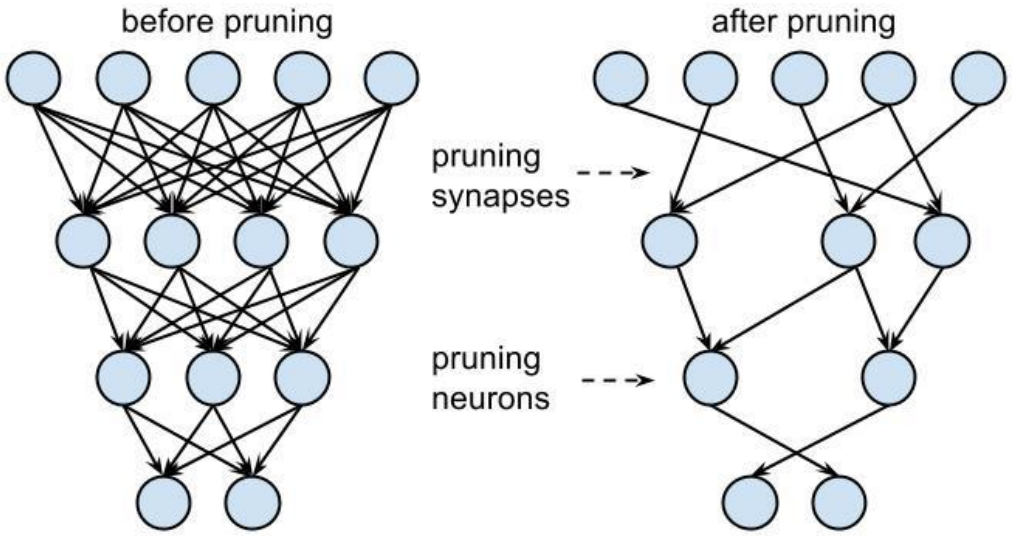
Report CA FP2

0410110 林容安 0410137 劉家麟

A. Describe our implementation algorithm and explain our results

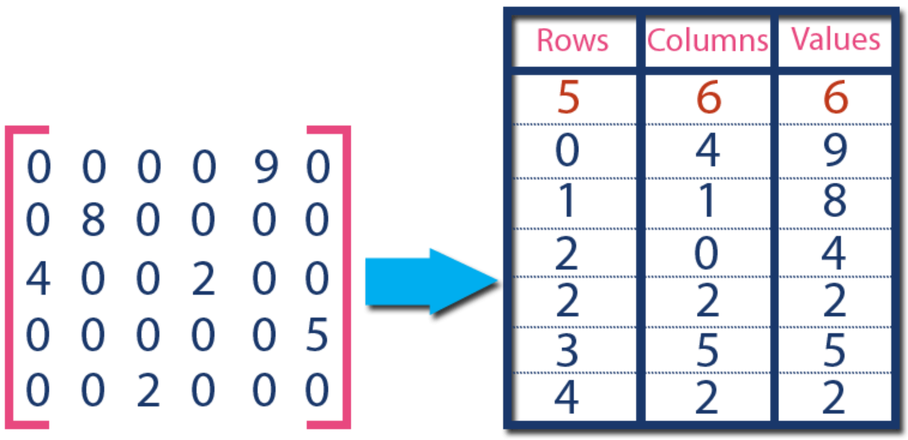
1. Sparse CNN



Sparse CNN is the transformation of CNN, which contains few connection after pruning just like above.

As we have done CNN in Final Project Part1, this time we'd like to use sparse CNN, whose filter and inNeu are composed of many zero elements so that we don't need to take care of those connection.

2. Sparse Matrix Implementation: COO Format



We knew that a sparse matrix stored in a common way is really a waste of space, and sparse format solves this problem.

From the picture, we can see that the matrix originally costs 30 sizeof(int) to store. After implementing the COO format, the size decreased to 18. In Final Project Part 2, our matrix has a sparsity of approximately 75 percent, so COO format come in handy.

B. Discuss what kind of optimization you did (it is better or worse?)

As we talked in part A, we use the CUDA architecture to divide convolution for loops into several blocks and threads. We had cut them in different ways.

The original convolution is about this. There are 6 for loops representing Filter Number, Frame Size, Frame Depth and Filter Size, separately.

```
for(fn = 0; fn < FILTNUM; fn++){
     for(fmy = 0; fmy < FMSIZE; fmy += STRIDE){
          for(fmx = 0; fmx < FMSIZE; fmx += STRIDE){
               sum = 0;
               for(sli = 0; sli < FMDEPTH; sli++){
                    for(y = 0; y < FILTSIZE; y++){
                         for(x = 0; x < FILTSIZE; x++){
                              //do convolution
                         }
                    }
               }
               //do ReLU
          }
     }
}
```

There are two cases.

1. <u>The one we did in FP1 (not considering sparsity)</u>

In this case, we break the whole convCPU into 2 parts: Convolution & ReLU and MaxPooling. Both have 1-D blocks, 2-D threads.

```
dim3 numBlocks(FILTNUM); //128
dim3 threadsPerBlock(FMSIZE,FMSIZE); //27*27
```

```
int bx = blockIdx.x; //FILTNUM 128
int tx = threadIdx.x; //FMSIZE 27 x(col)
int ty = threadIdx.y; //FMSIZE 27 y(row)
```

```
for (sli = 0; sli < FMDEPTH; sli++){
     for(y = 0; y < FILTSIZE; y++){ // FILTSIZE 5 y(row)
          for(x = 0; x < FILTSIZE; x++){
           // do convolution
}}}
```

MaxPooling is below.

```
dim3 P_numBlocks(FILTNUM); //128
dim3 P_threadsPerBlock(FMSIZE/3,FMSIZE/3); //9*9
```

```
int bx = blockIdx.x; //FILTNUM 128
int fmx = threadIdx.x; //FMSIZE/3 9 x(col)
int fmy = threadIdx.y; //FMSIZE/3 9 y(row)
```

The result of case1:

```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366
```

Besides, we found out that setting device first by adding cudaSetDevice(2); at the beginning of the main code can also obtain increased speedup from 3 to 5.

w/o `cudaSetDevice(2);`

```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1624.5ms
==499== NVPROF is profiling process 499, command: ./CNNConvLayer
GPU time for executing a typical convolutional layer = 492.96ms
Congratulations! You pass the check.
Speedup: 3.29541
```

w/ `cudaSetDevice(2);`


```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366
```


2. **<u>Convolution & ReLU with sparse CNN (maxpooling unmodified)</u>**

Since the NNZ of each fiter of each channel varies, we could only let the GPU compute a little portion (FMSIZE x FMSIZE) simultaneously. Utilizing the characteristics of sparse matrix, in the sparse CNN, we only have to consider the data which is not zero, so the convolution becomes pretty simple.
Though we parallelize smaller portion than we do in case 1, by implementing sparse CNN, we still obtained a speed up 8.49x, which is a better result compared to case 1(5.44x).

convolutioning with 1-D blocks, 2-D threads

```
dim3 numBlocks(1);
dim3 threadPerBlock(FMSIZE,FMSIZE);
```

```
int tx = threadIdx.x; // FMSIZE 27 x(col)
int ty = threadIdx.y; // FMSIZE 27 y(row)
```

```
for(int fn = 0 ; fn < FILTNUM ; fn ++){
   sum = 0;
   for(int sli = 0; sli < FMDEPTH; sli++){
      for(int idx=0; idx<FiltCooNNZ[fn*FMDEPTH+sli];idx++){
          CooIdx = tmp + idx;
          ifmx = tx  + FiltCooCol[CooIdx]; //col
          ifmy = ty  + FiltCooRow[CooIdx]; //row
          inNeuIdx = sli*FMGSIZE*FMGSIZE+ifmy*FMGSIZE + ifmx;

          sum += FiltCooData[CooIdx] * InNeu[inNeuIdx];
          __syncthreads();
      }
      tmp = FiltCooNNZ[fn*FMDEPTH + sli] + tmp;
   }

     outNeuIdx = fn * FMSIZE * FMSIZE + ty*FMSIZE + tx;
     if(sum <= 0)
          outNeural[outNeuIdx] = 0;
     else
          outNeural[outNeuIdx] = sum;
}
```

To implement Sparse CNN, we also modified the input neuron matrix as below.

```
for(int i = 0 ; i < FMDEPTH ; i++){
     for(int j = 0 ; j < (FMGSIZE) ; j++){
          for(int k = 0 ; k < (FMGSIZE) ; k++){
               inNeuIdx = i*(FMGSIZE)*(FMGSIZE) + j*(FMGSIZE) +
k;
               inNeuIdy=i*(FMSIZE)*(FMSIZE)+(j-5/2)*FMSIZE+(k-
5/2);
               if(j>=2 && j<=28 &&k>=2 &&
k<=28)tmp=inNeu[inNeuIdy];
               else tmp = 0 ;
               inGNeu[inNeuIdx] = tmp;
          }
     }
}
```

Where FMGSIZE is the frame size after modification, and the value is 29.

The result of case 2:

```
make rca56@Taipei:~/Sparse_CNN_COO$ make run
./CNNConvLayer
CPU time for executing a typical convolutional layer = 1334.05ms
GPU time for executing a typical convolutional layer = 159.195ms
Congratulations! You pass the check.
Speedup: 8.37995
```

Similarly, we tried adding cudaSetDevice(2); at the beginning of the main function, but found a relatively insignificant improvement in the speedup (8.10x to 8.38x). Therefore, we concluded that this case obtains the speedup mostly from sparse CNN rather than parallelism.

w/o cudaSetDevice(2);

```
make rca56@Taipei:~/Sparse_CNN_COO$ make run
./CNNConvLayer
CPU time for executing a typical convolutional layer = 1331.14ms
GPU time for executing a typical convolutional layer = 164.37ms
Congratulations! You pass the check.
Speedup: 8.09844
```

w/ cudaSetDevice(2);

```
make rca56@Taipei:~/Sparse_CNN_COO$ make run
./CNNConvLayer
CPU time for executing a typical convolutional layer = 1334.05ms
GPU time for executing a typical convolutional layer = 159.195ms
Congratulations! You pass the check.
Speedup: 8.37995
```

C. Show how you use NVVP to help you find and solve perf. Issues

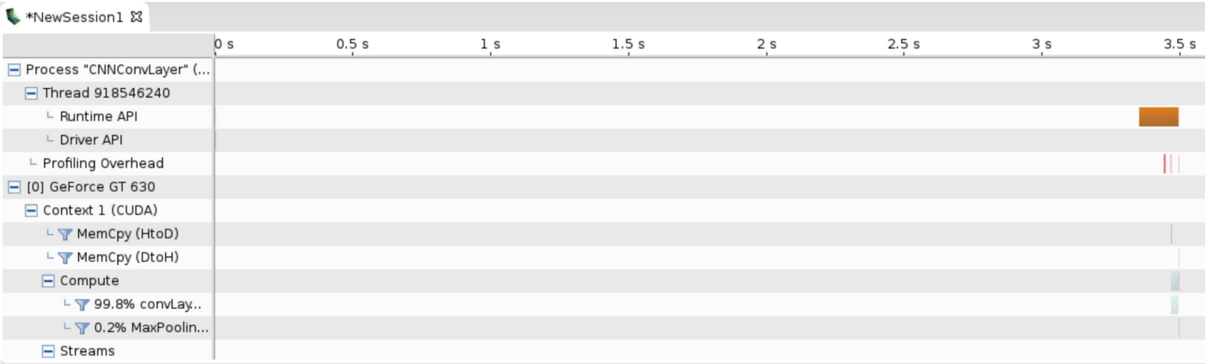
Checking NVVP of these two programs, both these two programs spend most time on **cudaMalloc();** and **cudaDeviceSynchronize();**. Thus, according to the Amdahl's Law, we improved **cudaMalloc();** by first access to GPU memory with **cudaFree(0);** before **cudaMalloc();**.

(Explanation: Cuda is a lazy initialization, which means it won't give us its context until we first cudaMalloc() it. So, the way to reduce cudaMalloc() time is that we call cudaFree(0) first, and then it would have given us its context by the time we cudaMalloc it.)

Before improvement, we can tell that **cudaMalloc();** and **cudaDeviceSynchronize();** spend most of time from following diagrams.

Program 1: same as FP1

nvvp version



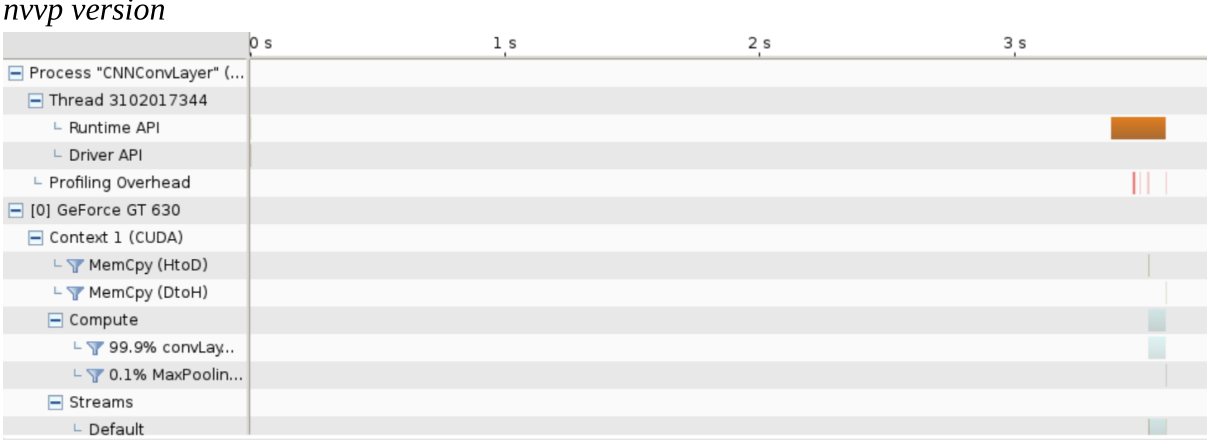
nvprof version

```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366
==1021== Profiling application: ./CNNConvLayer
==1021== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 96.91%  8.0077ms         1  8.0077ms  8.0077ms  8.0077ms  convLayerGPU(int*, int*, int*)
  2.87%  237.06us         2  118.53us  45.280us  191.78us  [CUDA memcpy HtoD]
  0.12%  9.5680us         1  9.5680us  9.5680us  9.5680us  PoolingGPU(int*, int*)
  0.11%  9.1200us         1  9.1200us  9.1200us  9.1200us  [CUDA memcpy DtoH]

==1021== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.29%  295.31ms         4  73.828ms  7.5130us  294.62ms  cudaMalloc
  2.63%  8.1617ms         2  4.0809ms  16.715us  8.1450ms  cudaDeviceSynchronize
  1.41%  4.3651ms       364  11.991us     270ns  435.00us  cuDeviceGetAttribute
  0.43%  1.3421ms         4  335.52us  332.60us  341.27us  cuDeviceTotalMem
  0.12%  359.92us         4  89.979us  82.127us  101.61us  cuDeviceGetName
  0.09%  285.52us         3  95.172us  34.330us  196.63us  cudaMemcpy
  0.02%  55.361us         2  27.680us  21.409us  33.952us  cudaLaunch
  0.00%  14.639us         1  14.639us  14.639us  14.639us  cudaSetDevice
  0.00%  6.3390us        12     528ns     273ns  1.0170us  cuDeviceGet
  0.00%  5.6120us         4  1.4030us     557ns  3.8340us  cudaFree
  0.00%  5.0220us         3  1.6740us     396ns  3.6160us  cuDeviceGetCount
  0.00%  4.2660us         5     853ns     189ns  3.1030us  cudaSetupArgument
  0.00%  2.4140us         2  1.2070us     463ns  1.9510us  cudaConfigureCall
```

Program 2: using COO format

nvvp version



nvprof version

```
ca56@Taipei:~/Sparse_CNN_COO$ nvprof ./CNNConvLayer
==9958== NVPROF is profiling process 9958, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1336.1ms
GPU time for executing a typical convolutional layer = 187.069ms
Congratulations! You pass the check.
Speedup: 7.14231
==9958== Profiling application: ./CNNConvLayer
==9958== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.26%  68.171ms         1  68.171ms  68.171ms  68.171ms  convLayerGPUSparse(int*, int*, int*, int*, int*, int*)
  0.66%  450.73us         5  90.146us  12.831us  112.80us  [CUDA memcpy HtoD]
  0.06%  43.710us         1  43.710us  43.710us  43.710us  MaxPoolingGPU(int*, int*)
  0.02%  14.591us         1  14.591us  14.591us  14.591us  [CUDA memcpy DtoH]

==9958== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 63.04%  118.13ms         7  16.876ms  5.7980us  117.79ms  cudaMalloc
 36.46%  68.313ms         2  34.156ms  46.481us  68.266ms  cudaDeviceSynchronize
  0.29%  549.19us         6  91.531us  20.359us  171.90us  cudaMemcpy
  0.13%  252.92us        91  2.7790us     146ns  109.04us  cuDeviceGetAttribute
  0.02%  43.291us         2  21.645us  15.062us  28.229us  cudaLaunch
  0.02%  35.392us         1  35.392us  35.392us  35.392us  cuDeviceTotalMem
  0.02%  29.879us         1  29.879us  29.879us  29.879us  cuDeviceGetName
  0.01%  16.961us         8  2.1200us     603ns  12.435us  cudaFree
  0.00%  4.4910us         1  4.4910us  4.4910us  4.4910us  cudaSetDevice
  0.00%  2.7470us         3     915ns     187ns  1.5170us  cuDeviceGetCount
  0.00%  2.2560us         8     282ns     140ns     746ns  cudaSetupArgument
  0.00%  1.8100us         2     905ns     617ns  1.1930us  cudaConfigureCall
  0.00%  1.2240us         3     408ns     229ns     715ns  cuDeviceGet
```

Following are the results after accessing to cuda memory first with **cudaFree(0);**.

Program 1: same as FP1

improved version

```
ca56@Taipei:~/Sparse_CNN_COO$ nvprof ./CNNConvLayer
==10391== NVPROF is profiling process 10391, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1329.38ms
GPU time for executing a typical convolutional layer = 28.043ms
Congratulations! You pass the check.
Speedup: 47.4049
==10391== Profiling application: ./CNNConvLayer
==10391== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.10%  26.866ms         1  26.866ms  26.866ms  26.866ms  convLayerGPU(int*, int*, int*)
  1.69%  462.83us         2  231.41us  86.908us  375.92us  [CUDA memcpy HtoD]
  0.16%  43.327us         1  43.327us  43.327us  43.327us  MaxPoolingGPU(int*, int*)
  0.05%  14.591us         1  14.591us  14.591us  14.591us  [CUDA memcpy DtoH]

==10391== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 80.58%  118.10ms         9  13.122ms     639ns  118.09ms  cudaFree
 18.59%  27.242ms         2  13.621ms  45.691us  27.196ms  cudaDeviceSynchronize
  0.30%  432.59us         4  108.15us  6.4380us  210.58us  cudaMalloc
  0.23%  341.34us        91  3.7500us     144ns  148.63us  cuDeviceGetAttribute
  0.20%  292.31us         3  97.436us  36.989us  199.41us  cudaMemcpy
  0.04%  51.523us         2  25.761us  14.642us  36.881us  cudaLaunch
  0.03%  43.594us         1  43.594us  43.594us  43.594us  cuDeviceTotalMem
  0.03%  38.932us         1  38.932us  38.932us  38.932us  cuDeviceGetName
  0.00%  5.1520us         1  5.1520us  5.1520us  5.1520us  cudaSetDevice
  0.00%  2.2120us         2  1.1060us     482ns  1.7300us  cudaConfigureCall
  0.00%  2.1720us         3     724ns     161ns  1.4190us  cuDeviceGetCount
  0.00%  1.8830us         5     376ns     140ns     739ns  cudaSetupArgument
  0.00%  1.0190us         3     339ns     204ns     460ns  cuDeviceGet
```

The speedup has been boosted from about 5 to 47.

Program 2: using COO format

improved version

```
ca56@Taipei:~/Sparse_CNN_COO$ nvprof ./CNNConvLayer
==9497== NVPROF is profiling process 9497, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1332.35ms
GPU time for executing a typical convolutional layer = 69.459ms
Congratulations! You pass the check.
Speedup: 19.1818
==9497== Profiling application: ./CNNConvLayer
==9497== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.26%  68.155ms         1  68.155ms  68.155ms  68.155ms  convLayerGPUSparse(int*, int*, int*, int*, int*, int*)
  0.66%  450.60us         5  90.120us  12.831us  112.76us  [CUDA memcpy HtoD]
  0.06%  44.446us         1  44.446us  44.446us  44.446us  MaxPoolingGPU(int*, int*)
  0.02%  14.591us         1  14.591us  14.591us  14.591us  [CUDA memcpy DtoH]

==9497== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 63.18%  119.92ms         9  13.325ms     596ns  119.91ms  cudaFree
 35.98%  68.288ms         2  34.144ms  46.745us  68.241ms  cudaDeviceSynchronize
  0.29%  556.99us         7  79.570us  6.6410us  196.65us  cudaMalloc
  0.29%  545.46us         6  90.910us  21.565us  169.59us  cudaMemcpy
  0.18%  340.55us        91  3.7420us     149ns  147.95us  cuDeviceGetAttribute
  0.03%  49.320us         2  24.660us  15.196us  34.124us  cudaLaunch
  0.02%  42.927us         1  42.927us  42.927us  42.927us  cuDeviceTotalMem
  0.02%  37.902us         1  37.902us  37.902us  37.902us  cuDeviceGetName
  0.00%  4.3460us         1  4.3460us  4.3460us  4.3460us  cudaSetDevice
  0.00%  2.3460us         2  1.1730us     544ns  1.8020us  cudaConfigureCall
  0.00%  2.2890us         8     286ns     142ns     614ns  cudaSetupArgument
  0.00%  2.0870us         3     695ns     180ns  1.3300us  cuDeviceGetCount
  0.00%  1.1870us         3     395ns     222ns     592ns  cuDeviceGet
```

The speedup has been boosted from about 8 to 19.

D. Feedback of this part

In FP2, we had spent lots of time figuring out how to use COO format, and even given the frame paddings to meet the function judgement. Also, we cut less blocks of GPU this time so that the speedup of GPU is limited.

However, after improving the GPU memory access time, which is the largest part of GPU computing, we find out that the more blocks we divide, the quicker the programs can be. In the other words, using cudaFree(); to exclude memory access time improved the first one more than the second. And the best performance is the improved version of FP1, so we will hand in that one.