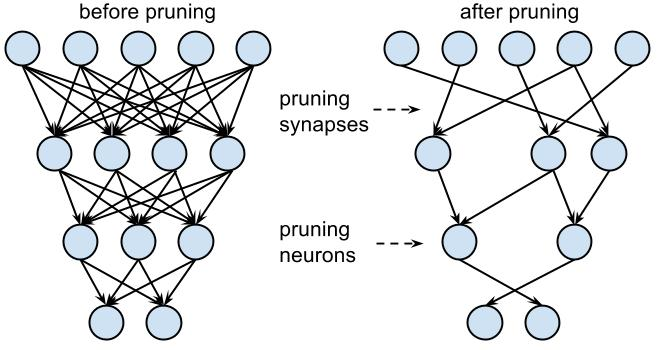
Report CA FP2

0410110 林容安 0410137 劉家麟

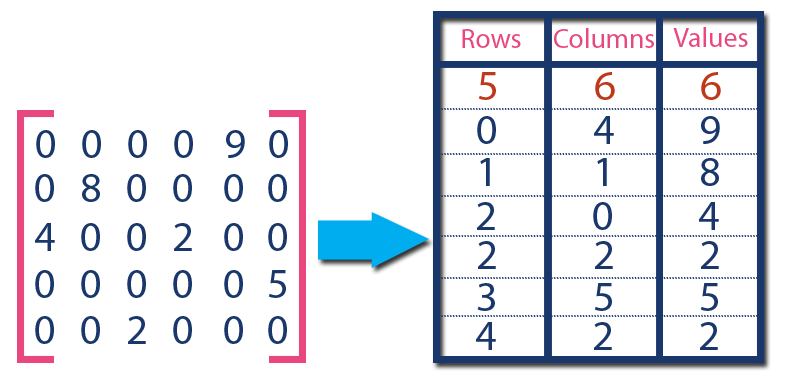
#### **A.** **Describe our implementation algorithm and explain our results**

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.

1. **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  }  }  } |

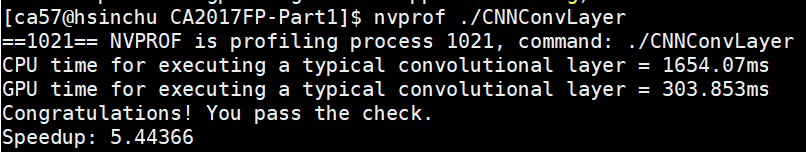
There are two cases.

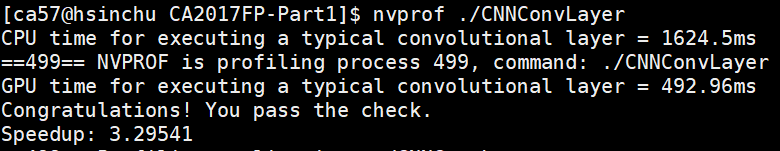
1. **The one we did in FP1 (not considering sparsity)**In this case, we break the whole convCPU into 2 parts: Convolution & ReLU and MaxPooling. Both have 1-D blocks, 2-D threads.

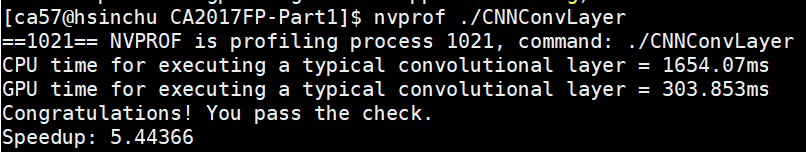
|  |
| --- |
| dim3 numBlocks(FILTNUM); //128  dim3 threadsPerBlock(FMSIZE,FMSIZE); //27\*27 |
| int bx = blockIdx.x; //FILTNUM 128  int tx = threadIdx.x; //FMSIZE 27 x(col)  int ty = threadIdx.y; //FMSIZE 27 y(row) |
| for (sli = 0; sli < FMDEPTH; sli++){  for(y = 0; y < FILTSIZE; y++){ // FILTSIZE 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:***  


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);  


w/ cudaSetDevice(2);  


1. **Convolution & ReLU with sparse CNN (maxpooling unmodified)**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**

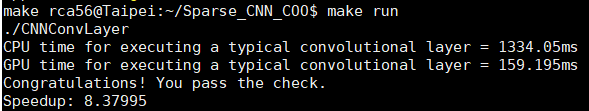
|  |
| --- |
| dim3 numBlocks(1);  dim3 threadPerBlock(FMSIZE,FMSIZE); |
| int tx = threadIdx.x; // FMSIZE 27 x(col)  int ty = threadIdx.y; // FMSIZE 27 y(row) |
| for(int fn = 0 ; fn < FILTNUM ; fn ++){  sum = 0;  for(int sli = 0; sli < FMDEPTH; sli++){  for(int idx=0; idx<FiltCooNNZ[fn\*FMDEPTH+sli];idx++){  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;  }  }  } |

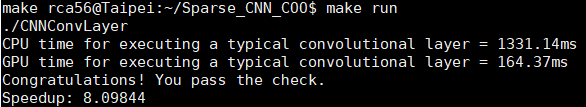
Where FMGSIZE is the frame size after modification, and the value is 29.

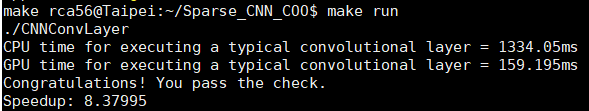
***The result of case 2:***

****

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);

****

w/ cudaSetDevice(2);  
****

#### **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* **original.PNG**

#### ***Program 2: using COO format*** *nvvp version**nvprof version*

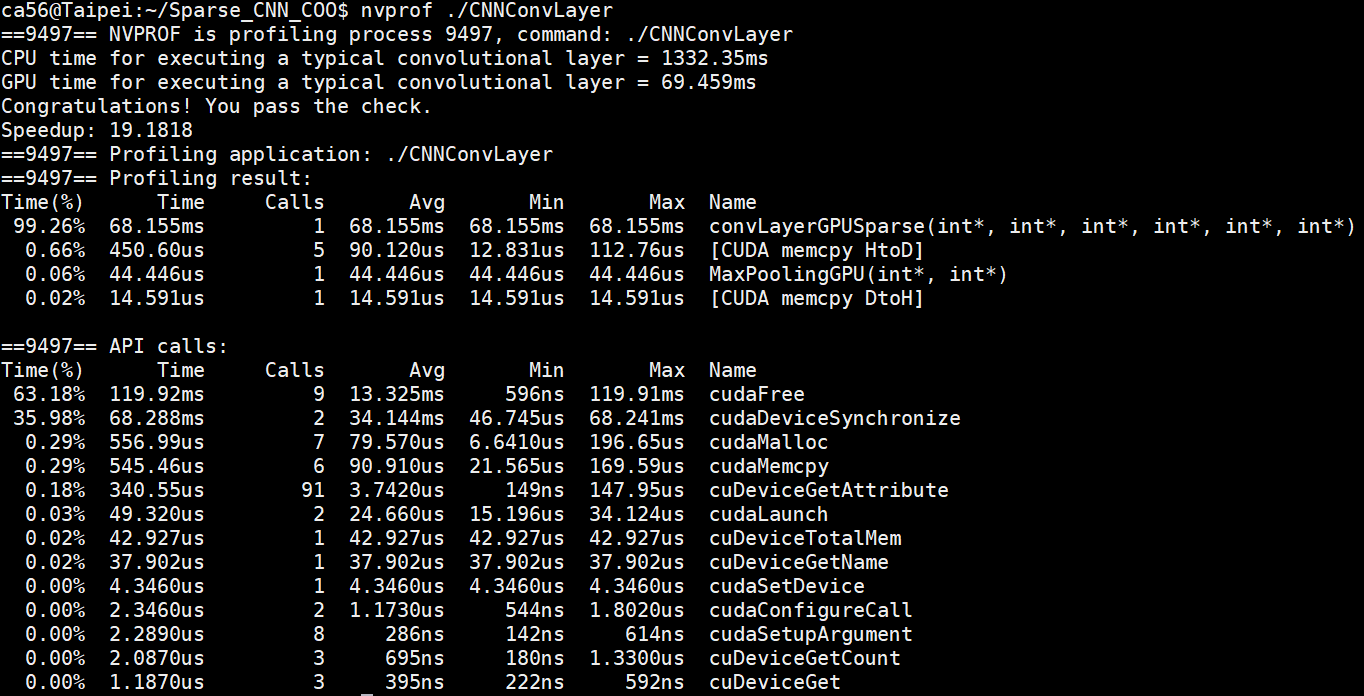
Following are the results after accessing to cuda memory first with **cudaFree(0);**.

#### ***Program 1: same as FP1***

#### *improved version* The speedup has been boosted from about 5 to 47.

***Program 2: using COO format***

*improved version*



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.