Report CA FP1

0410110 林容安 0410137 劉家麟

A. Describe our implementation algorithm and explain our results

1. Architecture of cuda GPU

a. Transfer data between host and device

host = our PC(CPU).
device = our GPU.
kernel = the code we'll run on GPU

Following is the dataflow of executing on GPU.
(1) Move the data(input) we'd like to use from CPU to GPU.
(2) Move the kernel(code) we'd like to execute from CPU to GPU.
(3) CPU is doing its processes, while GPU is executing the kernel.
(4) GPU copy the output back to CPU.

b. CUDA Architecture

CUDA can do parallel programming by dividing grid into blocks which contains many threads. The total threads that we can run depend on the grid dimensions and block dimension.

*# threads = gridDim * blockDim*

For example, below is a kernel which contains 12 threads.
gridDim = 3 (3 blocks in a grid)
blockDim = 4 (4 threads in a block)

grid (kernel)	block 0	thread 0 (id 0)
		thread 1 (id 1)
		thread 2 (id 2)
		thread 3 (id 3)
	block 1	thread 0 (id 4)
		thread 1 (id 5)
		thread 2 (id 6)
		thread 3 (id 7)
	block 3	thread 0 (id 9)
		thread 1 (id 10)

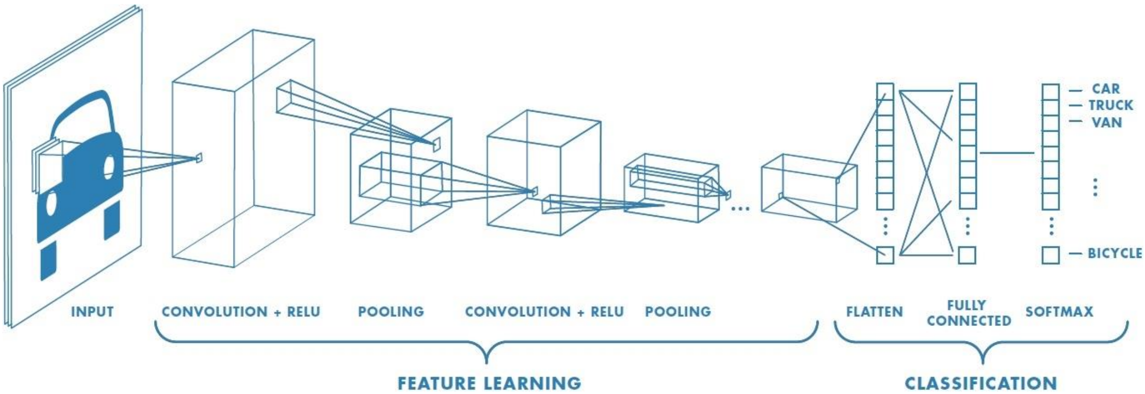
		thread 2 (id 11)
		thread 3 (id 12)

Thus, we use block index and thread index to access the specific thread.

`int id = blockIdx.x * blockDim.x + threadIdx.x`

And we use the `kernel<<<gridDim,blockDim>>>(arguments)` to call kernel. The type of gridDim and blockDim is actually not int but dim3, which means we can create at most 3-D blocks and 3-D threads. However, we should be careful of the maximum of GPU resources.

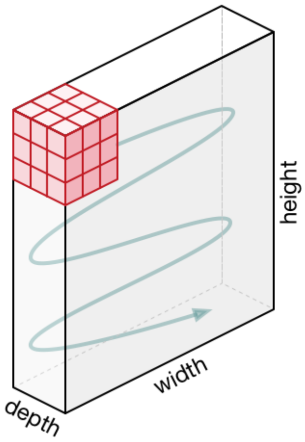
2. Convolution Neural Network



Convolution Neural Network (CNN) is a neural net for the computer to classify a group of images. The input for CNN is 3D array whose size is determined by the resolution of the image, and the output is a group of numbers telling the probability of the image in a certain class.

The way to function the above is to perform image classification by searching low-level features through several layers of convolution.

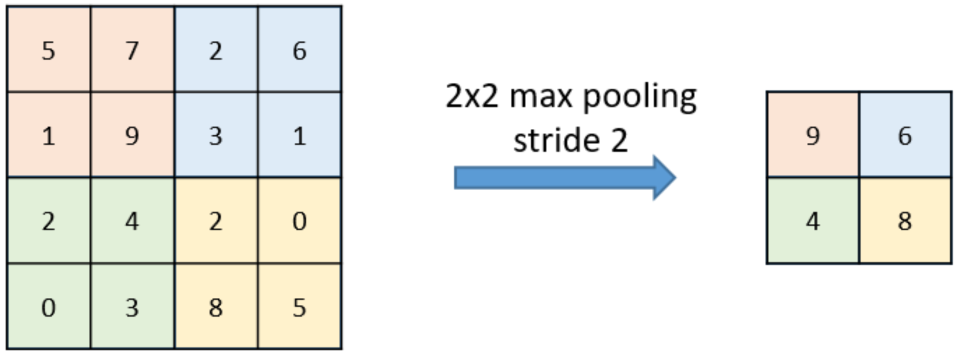
(a) Convolution & Relu



The convolution layer is always the first layer in CNN. In this section, the filter slides through the input image. The value of each pixel of the filter and the input will be multiplied, and the multiplication will be summed up and stored. Thereby, the depth of the filter must be equal to which of the input.

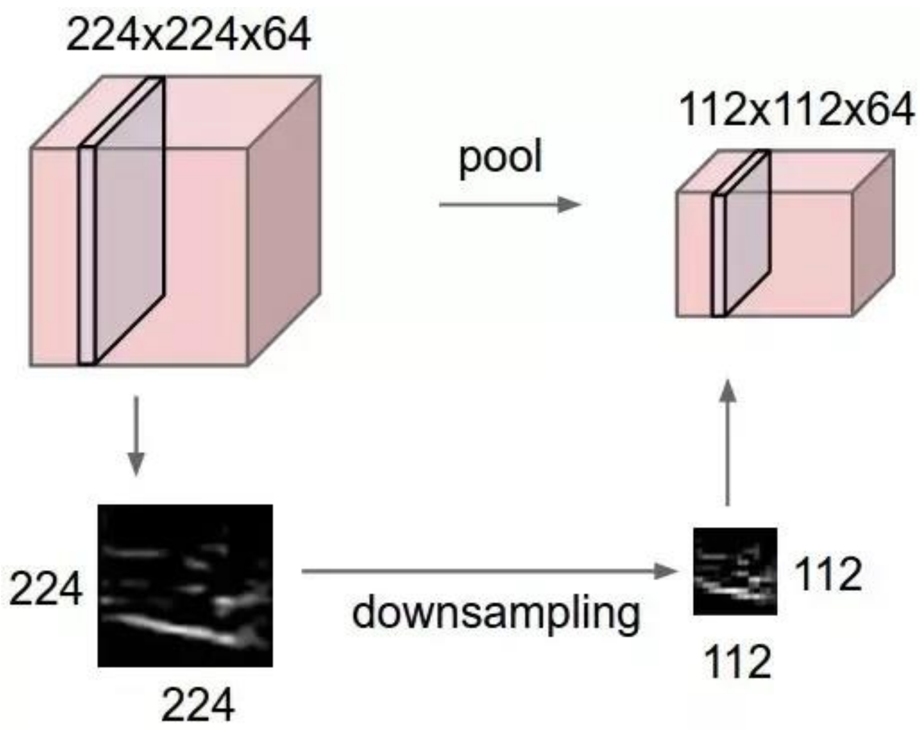
ReLU (rectifier linear unit) can be defined as the function f(x)=max(0,x). The reason we need ReLU in the neural network is that we'd like to have some neuron to be inactive, which makes the process sparser and more efficient.

(b) Max pooling



5	7	2	6
1	9	3	1
2	4	2	0
0	3	8	5

9	6
4	8



Max pooling is used to down-sample a matrix. Similar to the ReLU function, max pooling can also simplify the computation and save cost for the program.

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>1-D blocks and 2-D threads</u>**

```
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)
```

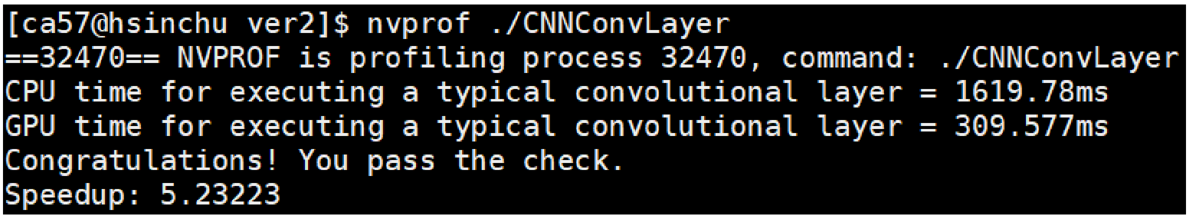
In first case, we use 2-D threads to represent Frame Size(27*27) of inNeu, and use 1-D block to substitute Filter Number(128).

Consequently, we can take off the first three for loops of convolution, and it became like this.

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

We directly use thread id (`blockIdx.x * blockDim.x + threadIdx.x`) to substitute Frame Size, and there will be 128 blocks, which means filter number, run at the same time. And each of one just needs to run 3 for loops, which would be quicker than 6 for loops.

The result of case1:



```
[ca57@hsinchu ver2]$ nvprof ./CNNConvLayer
==32470== NVPROF is profiling process 32470, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1619.78ms
GPU time for executing a typical convolutional layer = 309.577ms
Congratulations! You pass the check.
Speedup: 5.23223
```

2. **<u>Convolution & ReLU / MaxPooling: 1-D blocks, 2-D threads</u>**

In this case, we break the whole convCPU into 2 parts: Convolution & ReLU and

MaxPooling. Both are 1-D blocks, 2-D threads.
Convolution & ReLU part is same as case 1.

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

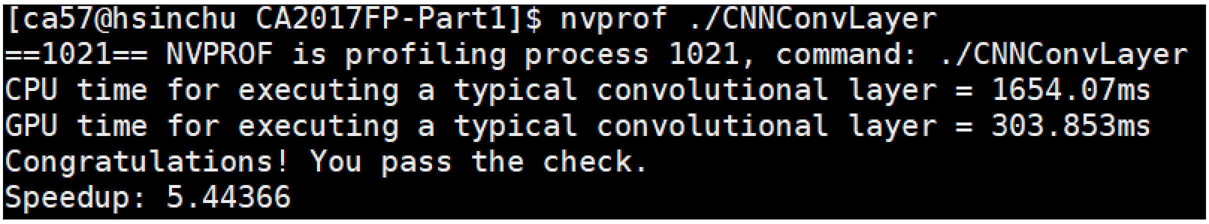
```
int bx = blockIdx.x; //FILTNUM 128
int tx = threadIdx.x; //FMSIZE 27 x(col)
int ty = threadIdx.y; //FMSIZE 27 y(row)
```

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 case2:

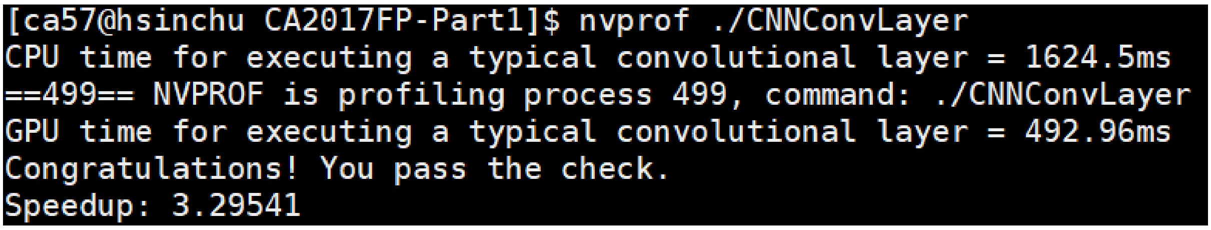


```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366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

We found that the second case is a little bit quicker than case 1, so we will turn in the case 2 version.

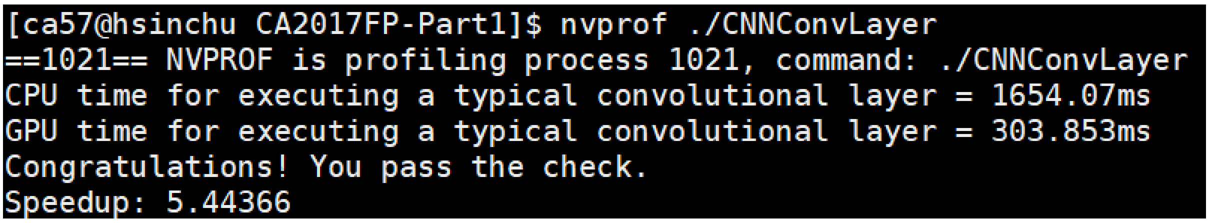
Besides, we found out that setting device first by adding `cudaSetDevice(2);` at the beginning of the main code can also speed up from 3 to 5.

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
```

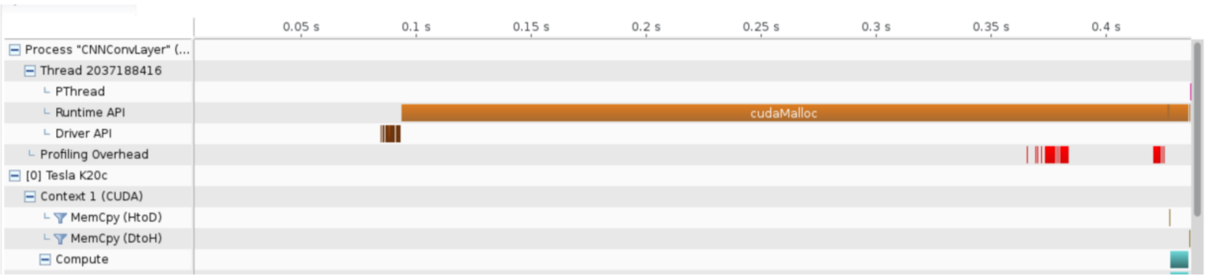
w/ `cudaSetDevice(2);`

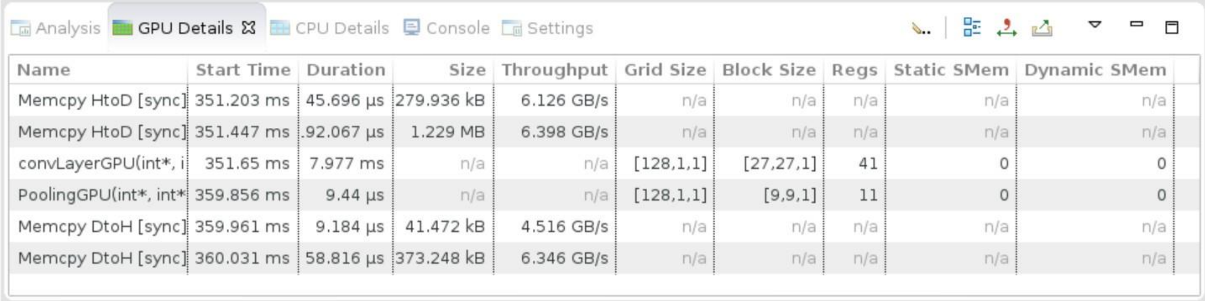


```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366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

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

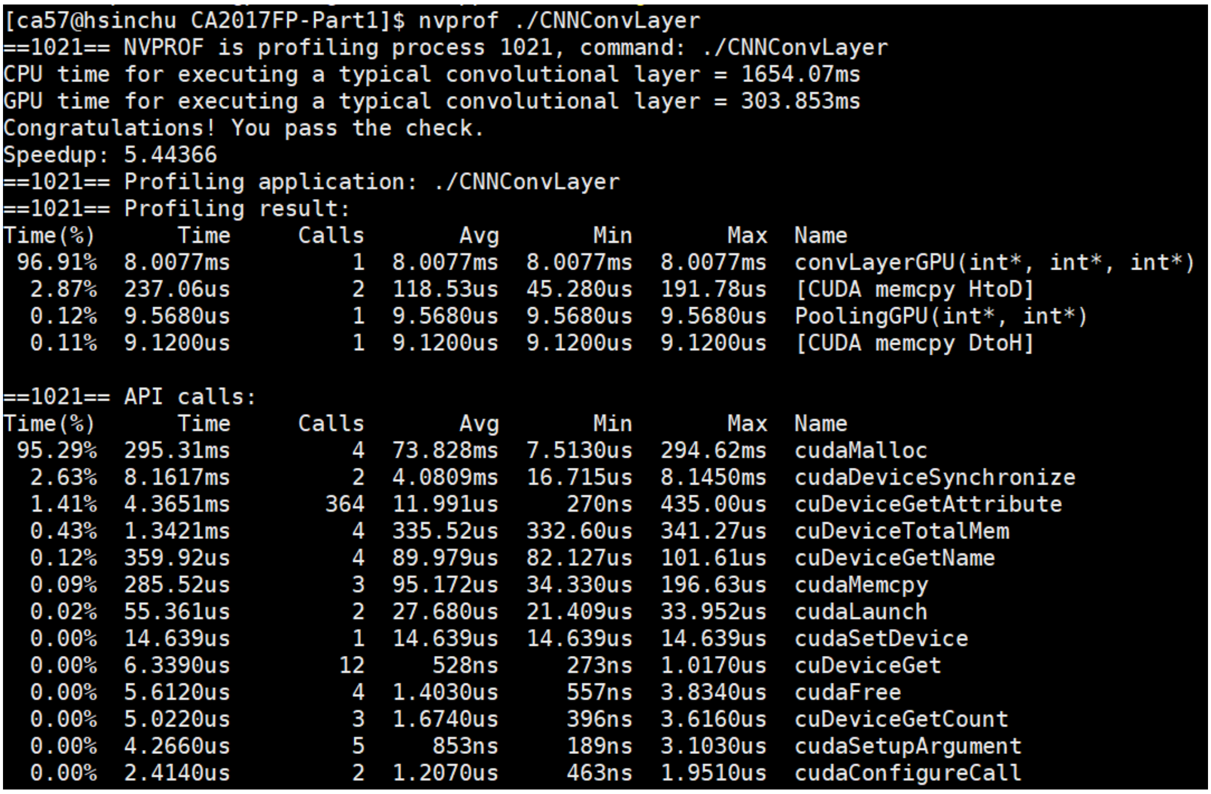
original





Name	Start Time	Duration	Size	Throughput	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem
Memcpy HtoD [sync]	351.203 ms	45.696 µs	279.936 kB	6.126 GB/s	n/a	n/a	n/a	n/a	n/a
Memcpy HtoD [sync]	351.447 ms	.92.067 µs	1.229 MB	6.398 GB/s	n/a	n/a	n/a	n/a	n/a
convLayerGPU(int*, i	351.65 ms	7.977 ms	n/a	n/a	[128,1,1]	[27,27,1]	41	0	0
PoolingGPU(int*, int*	359.856 ms	9.44 µs	n/a	n/a	[128,1,1]	[9,9,1]	11	0	0
Memcpy DtoH [sync]	359.961 ms	9.184 µs	41.472 kB	4.516 GB/s	n/a	n/a	n/a	n/a	n/a
Memcpy DtoH [sync]	360.031 ms	58.816 µs	373.248 kB	6.346 GB/s	n/a	n/a	n/a	n/a	n/a

nvprof version



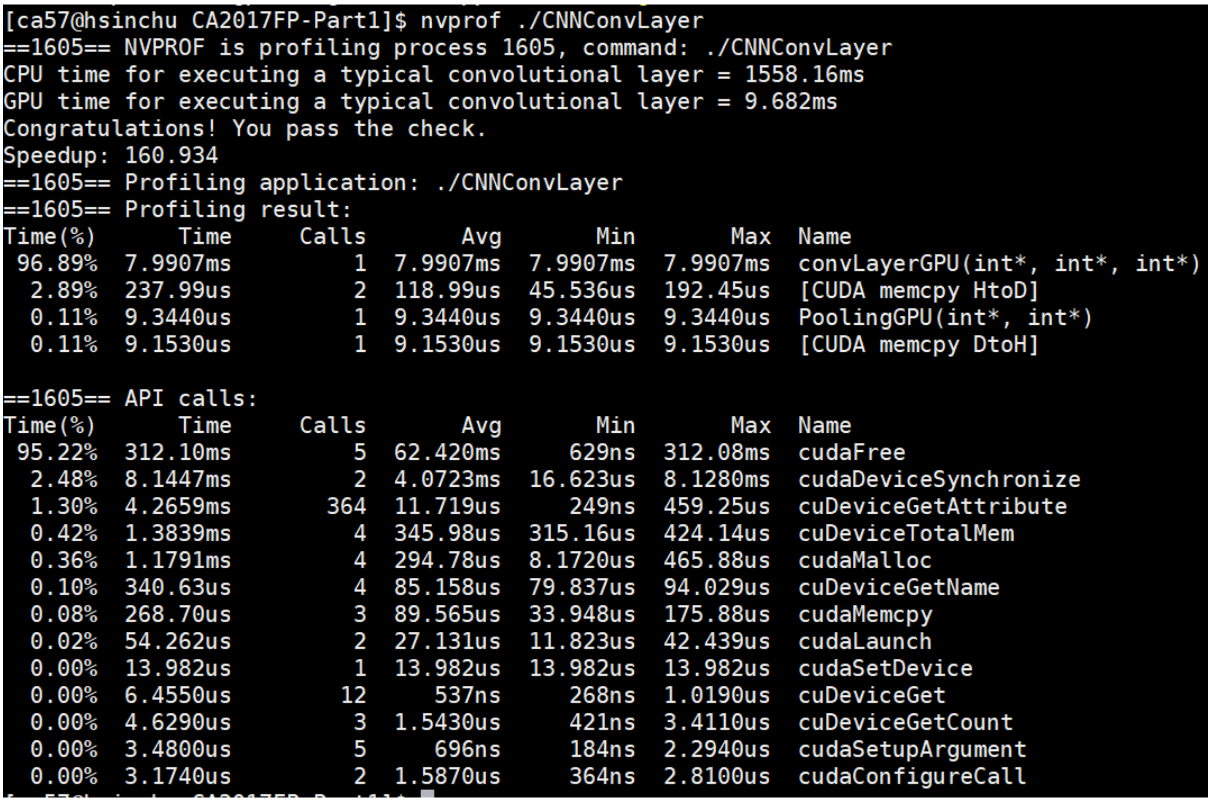
```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1021== NVPROF is profiling process 1021, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1654.07ms
GPU time for executing a typical convolutional layer = 303.853ms
Congratulations! You pass the check.
Speedup: 5.44366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

With NVVP(nvprof), obviously cudaMalloc is a large propotion of runtime. In Amdahl's law, we learned that improve the largest part of runtime will speed up the most. Thus, we decided to improve the `cudaMalloc()` part.

We googled and found out that cuda is 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.

improved version



```
[ca57@hsinchu CA2017FP-Part1]$ nvprof ./CNNConvLayer
==1605== NVPROF is profiling process 1605, command: ./CNNConvLayer
CPU time for executing a typical convolutional layer = 1558.16ms
GPU time for executing a typical convolutional layer = 9.682ms
Congratulations! You pass the check.
Speedup: 160.934
==1605== Profiling application: ./CNNConvLayer
==1605== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 96.89%  7.9907ms         1  7.9907ms  7.9907ms  7.9907ms  convLayerGPU(int*, int*, int*)
  2.89%  237.99us         2  118.99us  45.536us  192.45us  [CUDA memcpy HtoD]
  0.11%  9.3440us         1  9.3440us  9.3440us  9.3440us  PoolingGPU(int*, int*)
  0.11%  9.1530us         1  9.1530us  9.1530us  9.1530us  [CUDA memcpy DtoH]

==1605== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.22%  312.10ms         5  62.420ms     629ns  312.08ms  cudaFree
  2.48%  8.1447ms         2  4.0723ms  16.623us  8.1280ms  cudaDeviceSynchronize
  1.30%  4.2659ms       364  11.719us     249ns  459.25us  cuDeviceGetAttribute
  0.42%  1.3839ms         4  345.98us  315.16us  424.14us  cuDeviceTotalMem
  0.36%  1.1791ms         4  294.78us  8.1720us  465.88us  cudaMalloc
  0.10%  340.63us         4  85.158us  79.837us  94.029us  cuDeviceGetName
  0.08%  268.70us         3  89.565us  33.948us  175.88us  cudaMemcpy
  0.02%  54.262us         2  27.131us  11.823us  42.439us  cudaLaunch
  0.00%  13.982us         1  13.982us  13.982us  13.982us  cudaSetDevice
  0.00%  6.4550us        12     537ns     268ns  1.0190us  cuDeviceGet
  0.00%  4.6290us         3  1.5430us     421ns  3.4110us  cuDeviceGetCount
  0.00%  3.4800us         5     696ns     184ns  2.2940us  cudaSetupArgument
  0.00%  3.1740us         2  1.5870us     364ns  2.8100us  cudaConfigureCall
```

cudaMalloc have decreased from 295.31ms to 1.1791ms.

D. Feedback of this part

In Final Project Part 1, we found out that GPU is indeed quicker than CPU. However, when the GPU is occupied by too many people, it is worse than CPU. We have got the results that speedup is about 0.4~0.5 or even worse several times. Thus, before using GPU to speed up, we should check whether there is enough hardware resource.

When the deadline is on the corner, the utilization of CPU and GPU is really high, so we think that providing more GPUs and CPUs can solve this problem. If the budget is too tight, maybe TAs can allocate CPU using time for different groups, so that every group can have enough resource to run and no need to line up for CPU anymore.