Report CA FP1

0410110 林容安 0410137 劉家麟

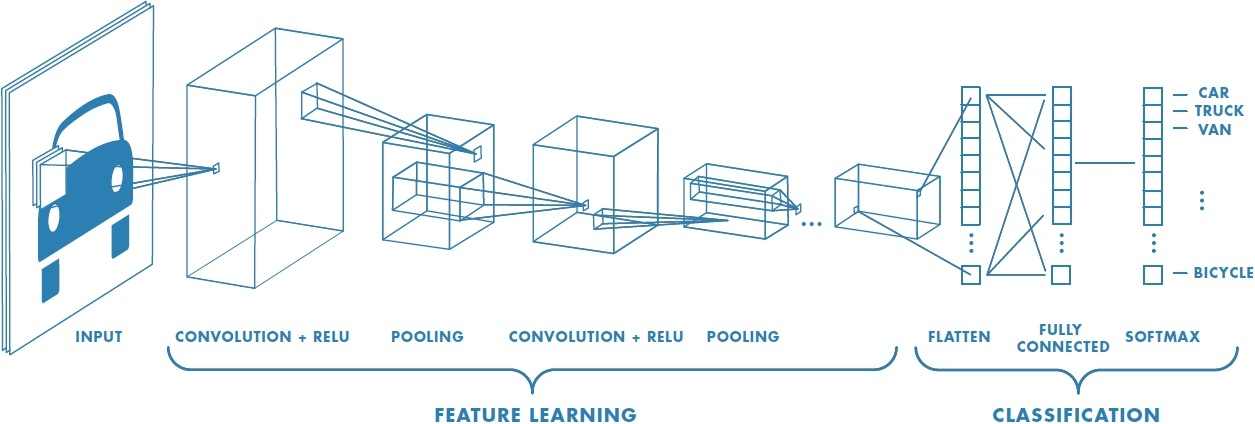
#### **A.** **Describe our implementation algorithm and explain our results**

1. **Architecture of cuda GPU**
   1. **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.
   2. **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.

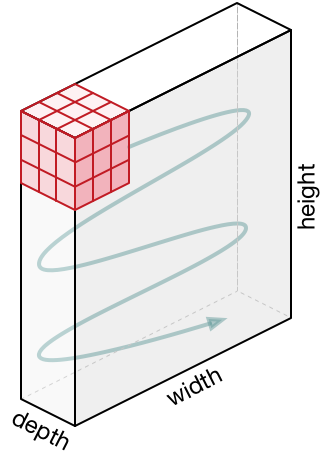
1. **Convolution Neural Network**

****

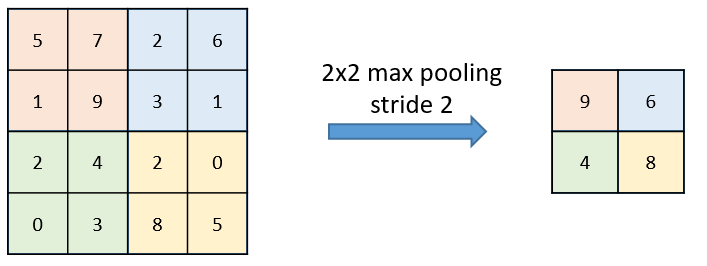
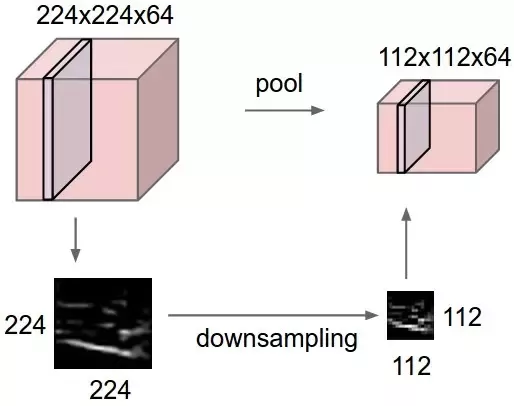
Convolution Neural Network (CNN) is a neural net for the computer to classify a group of images. The input for CNN is 3D array whose size is determined by the resolution of the image, and the output is a group of numbers telling the probability of the image in a certain class.

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

1. **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.

1. **Max pooling**

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.

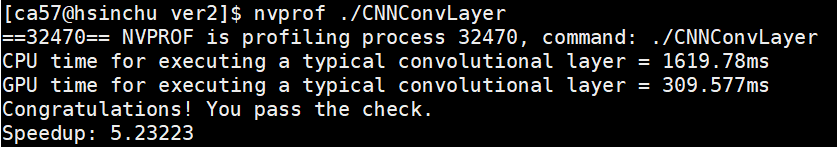
1. **1-D blocks and 2-D threads**

|  |
| --- |
| 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) |

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  }  }  } |

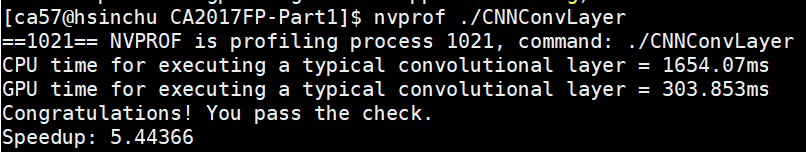
We directly use thread id (blockIdx.x \* blockDim.x + threadIdx.x) to substitute Frame Size, and there will be 128 blocks, which means filter number, run at the same time. And each of one just needs to run 3 for loops, which would be quicker than 6 for loops.  
The result of case1:  


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

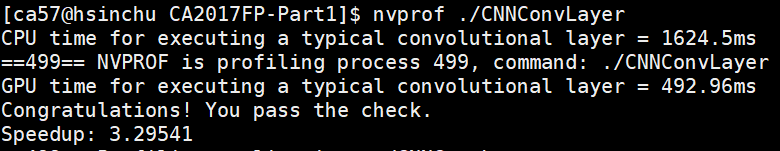
|  |
| --- |
| dim3 numBlocks(FILTNUM); //128  dim3 threadsPerBlock(FMSIZE,FMSIZE); //27\*27 |
| int bx = blockIdx.x; //FILTNUM 128  int tx = threadIdx.x; //FMSIZE 27 x(col)  int ty = threadIdx.y; //FMSIZE 27 y(row) |

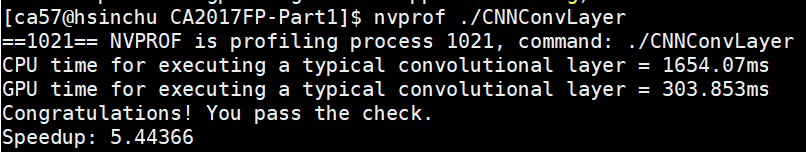
MaxPooling is below.

|  |
| --- |
| 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:  


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);  


w/ cudaSetDevice(2);  


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

#### *original* **cudaMalloc.PNG messageImage_1508993002501.jpg** *nvprof version* **original.PNG**

#### 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* improved.PNG 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.