中山大学计算机学院本科生实验报告

(2020 学年秋季学期)

课程名称: 高性能计算程序设计　　任课教师: 黄聃　　批改人:

年级 + 班级	18 计科 8 班	专业（方向）	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 12 月 27 号

目录

1 实验目的

- 通过解决实际问题，更加理解 CUDA 的结构。
- 进一步熟悉 CUDA 的基本编程接口。
- 了解 CUDA 编程在人工智能领域的应用

2 实验过程及核心代码

2.1 实验环境

2.1.1 硬件

在这里我采用了超算中心的 th2k 集群上的 GPU 节点, 先用 salloc -N 1 -p gpu_v100 -J zhb 申请一个节点，用 ssh 连接上去。上面的显卡情况如下：

```
[sysu_hpcedu_302@gpu31 ~]$ nvidia-smi
Wed Dec 16 10:50:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:8A:00.0 Off |                    0 |
| N/A   33C    P0    36W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:8B:00.0 Off |                    0 |
| N/A   29C    P0    37W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:B3:00.0 Off |                    0 |
| N/A   30C    P0    35W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:B4:00.0 Off |                    0 |
| N/A   32C    P0    38W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
[sysu_hpcedu_302@gpu31 ~]$
```

对应的 CPU 采用了 Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz。

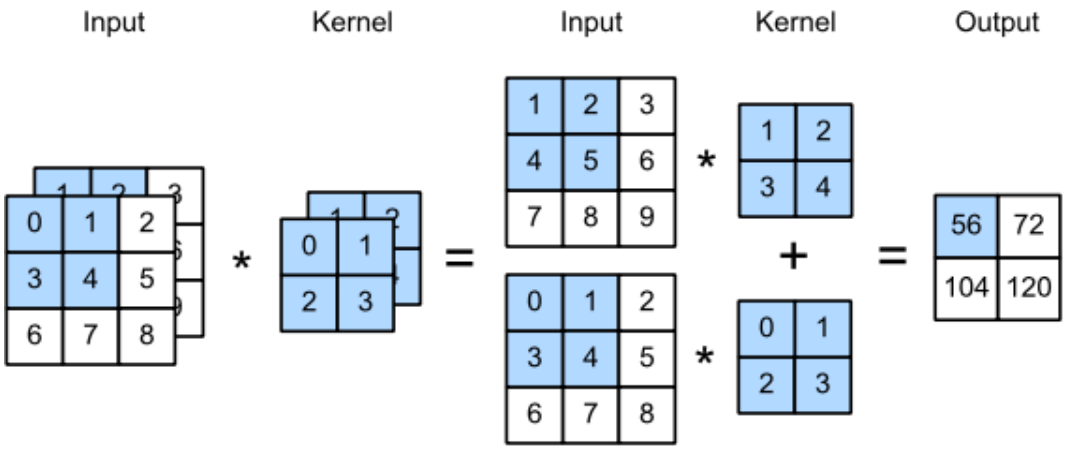
2.1.2 软件

- 操作系统：CentOS Linux release 7.6.1810 (Core)
- Toolkit:Cuda compilation tools, release 10.1, V10.1.243

- Library:cuDNN 7.6.4

2.2 通过 CUDA 实现直接卷积

在这里我们要实现的卷积示意图如下：



我们要根据输入多个通道的矩阵，对应数量的 Filter，根据 padding 的量扩大矩阵，然后再根据 stride 来确定每次滑动的幅度，进而确定计算每一个结果。图中例子的输入矩阵是 3*3*2，filter 的大小是 2*2*2，padding 为 0，步幅为 1.

我们要解决的问题是，根据输入，然后确定 padding 的值，使得 padding 后的矩阵恰好可以被 Filter 以对应的 stride 滑完。然后将矩阵扩充为 padding 后的矩阵，计算完每个 channel 的矩阵加和后，再将他们 reduce 起来即可。

卷积的核函数如下：

```
__global__ void convolution(float * mat,float * filter,float * res,
                            int height_stride,int width_stride,
                            int mat_height,int mat_width,
                            int filter_height,int filter_width,
                            int res_height,int res_width)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0;
    if(i < res_height && j < res_width){
        for(int x = 0;x < filter_height;x++){
            for(int y = 0;y < filter_width;y++){
                sum += mat[IDX2C(i * height_stride + x,j * width_stride + y,mat_width)]
                    * filter[IDX2C(x,y,filter_width)];
```

```
            }
        }
        *(res + IDX2C(i,j,res_width)) += sum;
    }
}
```

在 main 函数中，我定义了两个指针数组，指针数组的每个元素都申请了一段空间，用来存储不同通道的矩阵，还有不同通道对应的 filter。然后再根据 stride 和 padding 计算出结果矩阵的规模，然后申请一段空间来存储结果矩阵。

在 main 函数中用 for 循环将每一个通道的矩阵与 filter 做卷积的结果加到结果矩阵中，然后最终可以输出结果矩阵。

在这里的核心代码如下：

```
    for(int i = 0;i < channel;i++){
        convolution<<<numBlocks, threadsPerBlock>>>(d_Mat[i],d_filter[i],final_res,
            stride,stride,height + 2 * padding,width + 2 * padding,filter_height,
            filter_width,res_height,res_width);
    }
    cudaMemcpy(res,final_res,res_size,cudaMemcpyDeviceToHost);
```

在实现的过程中值得注意的细节是 padding 和结果的规模的计算方法。如下：

```
    int padding = ((((height - filter_height) / stride + 1) * stride - (height -
        filter_height)) % stride ) / 2;
    int res_height = (height - filter_height + 2 * padding) / stride + 1;
    int res_width = (width - filter_width + 2 * padding) / stride + 1;
```

为了说明程序的正确性，我构造了一个样例，如下：

```
[sysu_hpcedu_302@gpu44 ~/asc21/zhb/lab7/Q1]$ nvcc convolution.cu -o  convolution -w
[sysu_hpcedu_302@gpu44 ~/asc21/zhb/lab7/Q1]$ ./convolution
Input threadsPerBlock.x:4
Input threadsPerBlock.y:4
Input problem size:4
Input stride:3
Input channel 0 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0
Input channel 1 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0
Input channel 2 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0

Filter channel 0:
1 2 3
4 5 6
7 8 9
Filter channel 1:
1 2 3
4 5 6
7 8 9
Filter channel 2:
1 2 3
4 5 6
7 8 9

Res:
939 957
1137 939
```

我们很容易推出，当矩阵为 4*4，stride 为 3 的时候，如果 padding 为 1，可以恰好满足老师的要求让 filter 滑完。说明样例中的 padding 计算正确。

而我们很容易计算出，Res 的第一个元素可以由

$$7\times5+8\times6+13\times8+14\times9+7\times5+8\times6+13\times8+14\times9+7\times5+8\times6+13\times8+14\times9=923$$

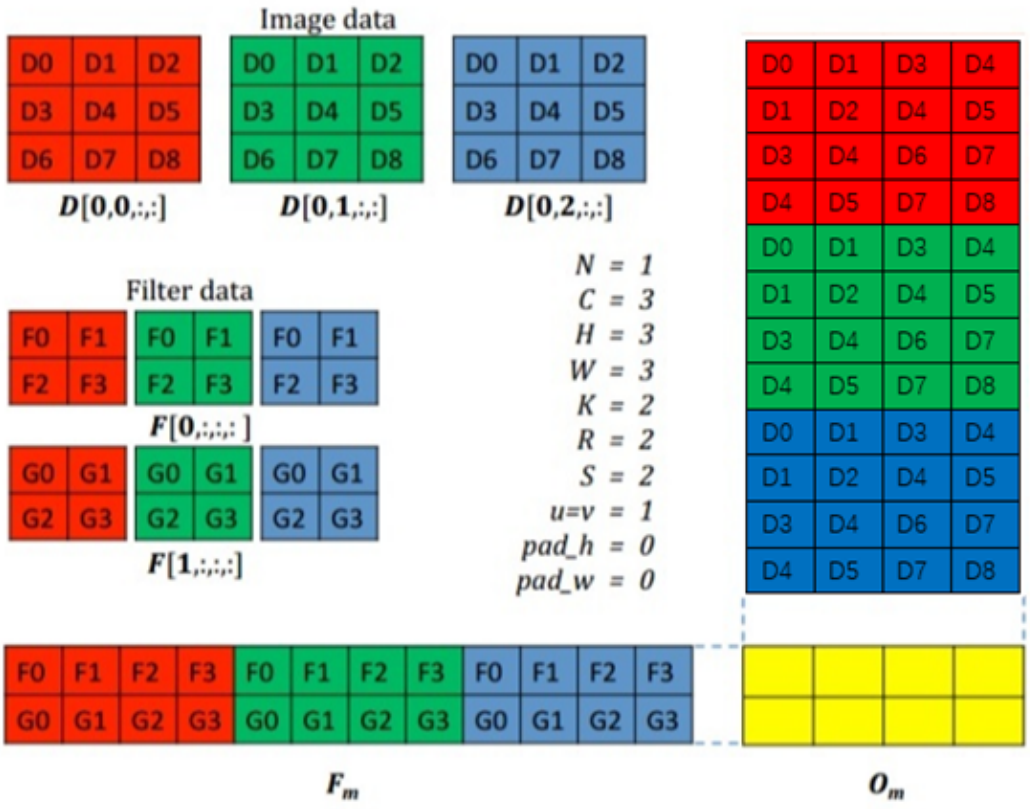
得到。

其他的元素也可以用类似的方法计算，经验证，所有的元素都计算正确，说明算法正确执行。

2.3 im2col 方法实现卷积

在这里我主要参考了im2col 方法实现卷积算法这篇文章。

im2col 的原理如下：



在实现的时候，我将矩阵展开得到的大矩阵作为矩阵乘法的第一个矩阵，将 filter 展开得到的矩阵作为矩阵乘法的第二个矩阵，虽然公式上的表示与上图略有不同，但是本质上是一致的。

这部分的核心代码如下：

```
__global__ void load(float * mat,int channel_id, int channel_count, float * unroll,
                     int height_stride,int width_stride,
                     int mat_height,int mat_width,
                     int filter_height,int filter_width,
                     int res_height,int res_width)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if(i < res_height && j < res_width){
        for(int x = 0;x < filter_height;x++){
            for(int y = 0;y < filter_width;y++){
                unroll[IDX2C(IDX2C(i,j,res_width),IDX2C(x,y,filter_width) + channel_id *
```

```
                    filter_height * filter_width ,channel_count * filter_height *
                    filter_width)] = mat[IDX2C(i * height_stride + x,j * width_stride +
                    y,mat_width)];
            }
        }
    }
    __syncthreads();
}
```

在将输入对应的数据分发好后，用 cudaMemcpy 将 filter 拷贝到列矩阵中，然后执行矩阵乘法操作，如下：

```
for(int i = 0;i < channel;i++){
    load<<<numBlocks, threadsPerBlock>>>(d_Mat[i],i,channel, unroll,stride,stride,height
        + 2 * padding,width + 2 * padding,filter_height,filter_width,res_height,
        res_width);
    cudaMemcpy(temp,unroll,res_size * filter_height * filter_width * channel,
        cudaMemcpyDeviceToHost);
}
float * W;
cudaMalloc(&W,filter_size * channel);
for(int i = 0;i < channel;i++){
    cudaMemcpy(W + i * (filter_height * filter_width),filter[i],filter_size,
        cudaMemcpyHostToDevice);
}
dim3 numBlocks1((res_height * res_width % threadsPerBlock.x) ? res_height * res_width /
    threadsPerBlock.x + 1 :res_height * res_width / threadsPerBlock.x ,(filter_height *
    filter_width % threadsPerBlock.y) ? filter_height * filter_width / threadsPerBlock.y
    + 1 : filter_height * filter_width / threadsPerBlock.y);
MatMul<<<numBlocks1,threadsPerBlock>>>(unroll,W,d_res,res_height*res_width,filter_height
    * filter_width * channel, 1);
```

通过这种方式可以实现空间换时间，将每一次的矩阵的一部分与 filter 对应相乘转换为矩阵乘法中某一行与 filter 矩阵相乘。这样子就可以简化计算的复杂度，在一定程度上简化计算。

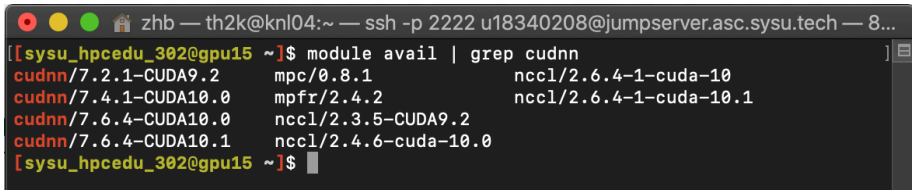
为了验证计算的正确性，我再次使用一样的问题来验证，结果如下：

```
[sysu_hpcedu_302@gpu35 ~/asc21/zhb/lab7/Q2]$ nvcc im2col.cu -o im2col
[sysu_hpcedu_302@gpu35 ~/asc21/zhb/lab7/Q2]$ ./im2col
Input threadsPerBlock.x:4
Input threadsPerBlock.y:4
Input problem size:4
Input stride:3
convolution time is:4μs
Input channel 0 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0
Input channel 1 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0
Input channel 2 after padding:
0 0 0 0 0 0
0 7 8 9 10 0
0 13 14 15 16 0
0 19 20 21 22 0
0 25 26 27 28 0
0 0 0 0 0 0

Filter channel 0:
1 2 3
4 5 6
7 8 9
Filter channel 1:
1 2 3
4 5 6
7 8 9
Filter channel 2:
1 2 3
4 5 6
7 8 9

Res:
939 957
1137 939
```

与任务一的结果相比，很容易发现两次的结果相同，说明算法正确计算出结果。

2.4 用 cuDNN 实现卷积

在超算中心的 th2k 集群上有安装 cuDNN 库，如下：



因为在这里 CUDA 的版本是 10.1，所以使用 module load cudnn/7.6.4-CUDA10.1 来加载 cuDNN 库。

我们在这里做的卷积是不需要对 Filter 进行翻转的，也称为互相关操作，即 NVIDIA 文档中的 cross-correlation。如下：

Normal Convolution (using cross-correlation mode)

$$y_{n,\,k,\,p,\,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,\,c,\,p+r,\,q+s} \times w_{k,c,r,s}$$

cuDNN 中提供的计算卷积操作的函数为 cudnnConvolutionForward()，原型如下：

```
cudnnStatus_t CUDNNWINAPI cudnnConvolutionForward(
                                cudnnHandle_t                       handle,
                                const void                         *alpha,
                                const cudnnTensorDescriptor_t       xDesc,
                                const void                         *x,
                                const cudnnFilterDescriptor_t       wDesc,
                                const void                         *w,
                                const cudnnConvolutionDescriptor_t  convDesc,
                                cudnnConvolutionFwdAlgo_t           algo,
                                void                               *workSpace,
                                size_t                              workSpaceSizeInBytes,
                                const void                         *beta,
                                const cudnnTensorDescriptor_t       yDesc,
                                void                               *y );
```

其中：

- x 为输入数据的地址，w 为卷积核的地址，y 为输出数据的地址，对应的 xDesc、wDesc 和 yDesc 为描述这三个数据的描述子，比如记录了数据的 batch size、channels、height 和 width 等。
- alpha 对卷积结果 x*w 进行缩放，beta 对输出 y 进行缩放，其表达式为：

```
dstValue = alpha*computedValue + beta*priorDstValue
```

- workspace 是指向进行卷积操作时需要的 GPU 空间的指针
- workSpaceSizeInBytes 为该空间的大小
- algo 用来指定使用什么算法来进行卷积运算
- handle 是创建的 library context 的句柄，使用 cuDNN 库必须用 cudnnCreate() 来初始化。

因为每个 CUDA 函数都会返回一个 cudaError_t 类型的值来显示函数是否正确执行，所以我在 debug 的时候编写了一个宏来判断函数执行的正确性，如下：

```
#define CUDA_CALL(f) { \
  cudaError_t err = (f); \
  if (err != cudaSuccess) { \
    cout \
        << "␣␣␣␣Error␣occurred:␣" << err << endl; \
    exit(1); \
  } \
}

#define CUDNN_CALL(f) { \
  cudnnStatus_t err = (f); \
  if (err != CUDNN_STATUS_SUCCESS) { \
    cout \
        << "␣␣␣␣Error␣occurred:␣" << err << endl; \
    exit(1); \
  } \
}
```

具体的代码模块如下：

首先是创建 handle：

```
    cudnnHandle_t cudnn;
    CUDNN_CALL(cudnnCreate(&cudnn));
```

然后要创建输入，Filter，输出对应的描述子，来表示矩阵的格式，如下：

```
    cudnnTensorDescriptor_t in_desc;
    CUDNN_CALL(cudnnCreateTensorDescriptor(&in_desc));
    CUDNN_CALL(cudnnSetTensor4dDescriptor(
          in_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
          in_n, in_c, in_h, in_w));

    cudnnFilterDescriptor_t filt_desc;
```

```
    CUDNN_CALL(cudnnCreateFilterDescriptor(&filt_desc));
    CUDNN_CALL(cudnnSetFilter4dDescriptor(
            filt_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW,
            filt_k, filt_c, filt_h, filt_w));
    cudnnTensorDescriptor_t out_desc;
    CUDNN_CALL(cudnnCreateTensorDescriptor(&out_desc));
    CUDNN_CALL(cudnnSetTensor4dDescriptor(
            out_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT,
            out_n, out_c, out_h, out_w));
```

然后创建卷积的描述子：

```
    cudnnConvolutionDescriptor_t conv_desc;
    CUDNN_CALL(cudnnCreateConvolutionDescriptor(&conv_desc));
    CUDNN_CALL(cudnnSetConvolution2dDescriptor(
            conv_desc,
            pad_h, pad_w, str_h, str_w, dil_h, dil_w,
            CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
```

该描述子会确定 padding，步长和 Filter 的 dilation，在这里 dil_h 和 dil_w 都取 1。该描述子还确定了卷积的方式，采用互相关的方式。

再根据这些描述子选择算法：

```
    CUDNN_CALL(cudnnGetConvolutionForwardAlgorithm(
            cudnn,
            in_desc, filt_desc, conv_desc, out_desc,
            CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &algo));
```

然后确定 workspace：

```
    size_t ws_size;
    CUDNN_CALL(cudnnGetConvolutionForwardWorkspaceSize(
            cudnn, in_desc, filt_desc, conv_desc, out_desc, algo, &ws_size));
```

最后就可以进行卷积操作了：

```
    float alpha = 1.f;
    float beta = 0.f;

    CUDNN_CALL(cudnnConvolutionForward(
        cudnn,
        &alpha, in_desc, in_data, filt_desc, filt_data,
        conv_desc, algo, ws_data, ws_size,
        &beta, out_desc, out_data));
```

也可以通过输入样例，在这里因为我初始化矩阵的时候没有采用之前的初始化，而是采用了直接在 GPU 中赋值，所以初始状态与之前不同。并且因为 padding 是通过调用库函数实现的，所以我就不需要自己手动进行 padding。运行样例的结果如下：

```
[[sysu_hpcedu_302@gpu3 ~/asc21/zhb/lab7/Q3]$ nvcc cuDNN.cu -o cuDNN -w -lcudnn
[[sysu_hpcedu_302@gpu3 ~/asc21/zhb/lab7/Q3]$ ./cuDNN
[Input problem size:4
[Input stride:3
Convolution algorithm: 0

Workspace size: 0

in_data:
n=0, c=0:
   0   1   2   3
   4   5   6   7
   8   9  10  11
  12  13  14  15
n=0, c=1:
  16  17  18  19
  20  21  22  23
  24  25  26  27
  28  29  30  31
n=0, c=2:
  32  33  34  35
  36  37  38  39
  40  41  42  43
  44  45  46  47

filt_data:
n=0, c=0:
   1   2   3
   4   5   6
   7   8   9
n=0, c=1:
   1   2   3
   4   5   6
   7   8   9
n=0, c=2:
   1   2   3
   4   5   6
   7   8   9

res:
1593 1515
1311 1065
```

我们很容易知道，padding 后，每个初始矩阵会多一圈 0。所以可以验证 res 的第一个值，如下：

$$0\times5+1\times6+4\times8+5\times9+16\times5+17\times6+20\times8+21\times9+32\times5+33\times6+36\times8+37\times9=1593$$

res 中的其他几个值经过验证，也是正确的结果，说明程序正确计算出结果。

3 实验结果

3.1 准备工作

首先编写了 Makefile：

```
CXX = nvcc
CXXFLAGS = -O3

all: convolution im2col cuDNN


convolution: Q1/convolution.cu
	$(CXX) $(CXXFLAGS) -o $@ $< -w

im2col: Q2/im2col.cu
	$(CXX) $(CXXFLAGS) -o $@ $< -w

cuDNN: Q3/cuDNN.cu
	$(CXX) $(CXXFLAGS) -o $@ $< -lcudnn -w

clean:
	rm convolution im2col cuDNN
```

然后加入测试时间的模块，就可以开始测试。

3.2 任务一

经过测试，我发现每个块的线程结构为 1*128 的时候运行时间最短。并且根据老师的要求，为了使得 padding 后的矩阵恰好滑完，根据 stride kernel 和 padding 调整 input size。运行的结果如下：

步长 \ 问题规模	256	512	1024	2048	4096
1	309007	310812	314675	322200	328918
2	304054	302719	310692	310660	317690
3	297801	309374	309614	305029	315641

表 1: 直接卷积在不同问题规模不同步长的计算时间 (μs)

很容易看到，问题规模增大的时候，运行的时间几乎没有变化。不过我额外测试问题规模为 8192

的时候，运算的时间飙升到 40000 μs 左右。这说明在这里没有明显变化是因为问题规模较小，可以将数据存储在 Cache 中，而问题规模较大的时候可能会超出 Cache，这种时候就容易发生缺页的现象，这就影响了从 Host 到 device 的内存拷贝过程。

3.3 任务二

在这里经过测试，我发现 1*128 的时候运行时间最短，同样的我调整了 input size 来适应问题。测试的结果如下：

步长 \ 问题规模	256	512	1024	2048	4096
1	285304	290829	291819	305350	392704
2	288545	292095	294137	294738	337114
3	284652	289619	288472	296905	321568

表 2: im2col 卷积在不同问题规模不同步长的计算时间 (μs)

可以看出，使用 im2col 算法实现的时候，最开始是比直接卷积稍微快要点点，但是问题规模较大的时候，由于需要拷贝的内存过多，使得整个求解的时间变得较大。

但是很多时候，计算任务可以用 GPU 初始化，所以在这种情况下，使用 im2col 算法无疑会使得计算变得非常迅速。

3.4 任务三

在这里只需要指定问题规模和步数，运行结果如下：

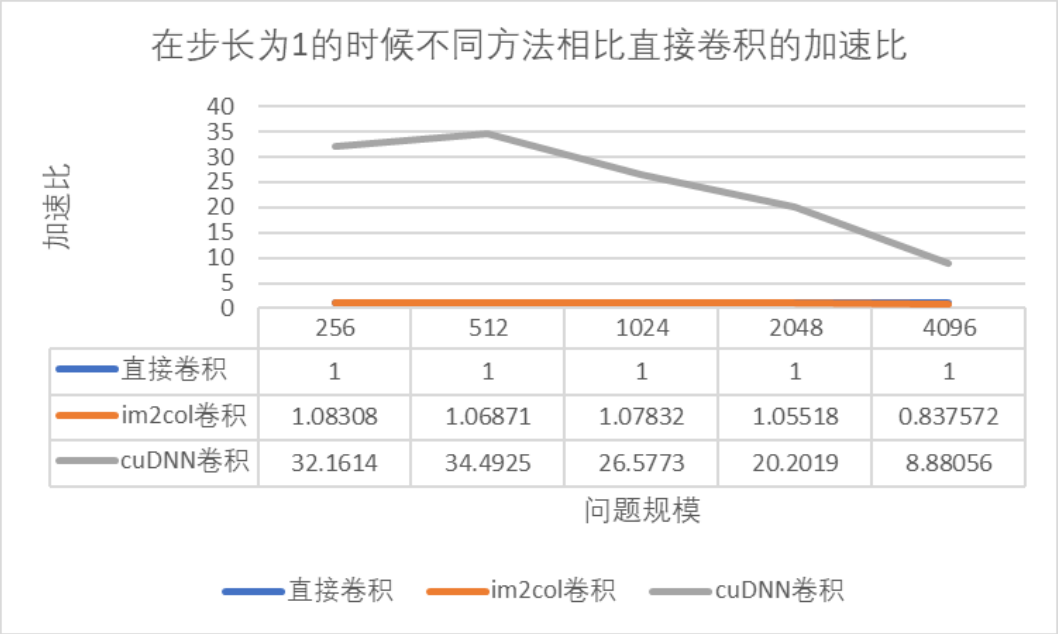
步长 \ 问题规模	256	512	1024	2048	4096
1	9608	9011	11840	15949	37038
2	8011	12093	10166	15906	37845
3	7944	8731	12176	17017	37066

表 3: cuDNN 实现的卷积在不同问题规模不同步长的计算时间 (μs)

在这里可以看到，相比自己实现的 Convolution，cuDNN 的加速非常可观。然后在运行过程中打印算法的选择，发现在不同规模下选择了不同的 Convolution algorithm，说明在这方面 cuDNN 的优化非常强劲。

3.5 结果分析

因为在相同问题规模下，不同步长计算的结果大致相同，所以在这里我选择步长为 1 的结果来分析加速比。如下：



可以看出，相比 cuDNN 的加速，直接卷积和 im2col 卷积的速度都大致相同，说明在这方面可以优化的空间非常大。

3.6 可能的加速方法

我们自己实现的卷积离 cuDNN 的差距非常大，说明还有很多优化空间，我根据课堂所学和官方教程找到了如下可能可以优化的方法：

- 使用 shared memory、constant memory、device memory 或 device memory 进行优化。通过提升访存的速度来提升性能。
- 在直接卷积的时候我是使用多层循环来将不同层之间的结果做加和，但是其实可以考虑先将不同 channel 的结果存储在不同的位置，然后用 openMP 并行 for 循环，再将不同 channel 的结果做一个 reduce 操作，
- 尝试使用 float2,float3,float4 矢量类型来加速计算。
- 针对问题的规模选择合适的算法。

4 实验感想

通过这次实验我进一步理解来 CUDA 在人工智能领域的应用，也更加理解来高性能计算的魅力。我也了解到人工智能常用到 pytouch 等库也与 CUDA 密切相关，这也是 CUDA 在人工智能领域的重要运用吧。