中山大学数据科学与计算机院本科生实验报告

(2020 秋季学期)

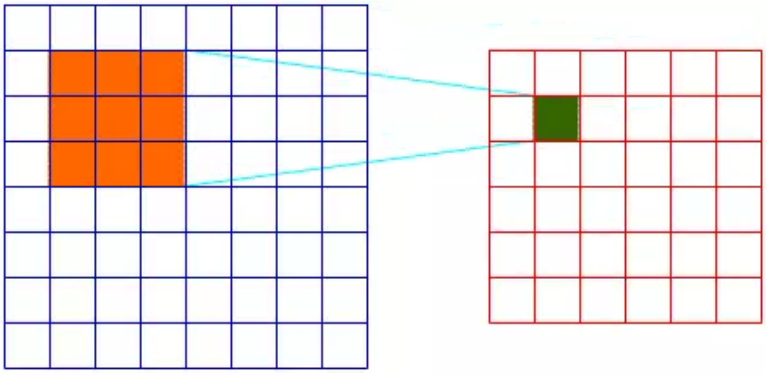
课程名称：高性能程序设计　**任课老师**：黄聃　**批改人**：

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.01.08

1 实验目的

1.1 任务 1

在信号处理、图像处理和其他工程/科学领域，卷积是一种使用广泛的技术。在深度学习领域，卷积神经网络 (CNN) 这种模型架构就得名于这种技术。在本实验中，我们将在 GPU 上实现卷积操作，注意这里的卷积是指神经网络中的卷积操作，与信号处理领域中的卷积操作不同，它不需要对 Filter 进行翻转，不考虑 bias。



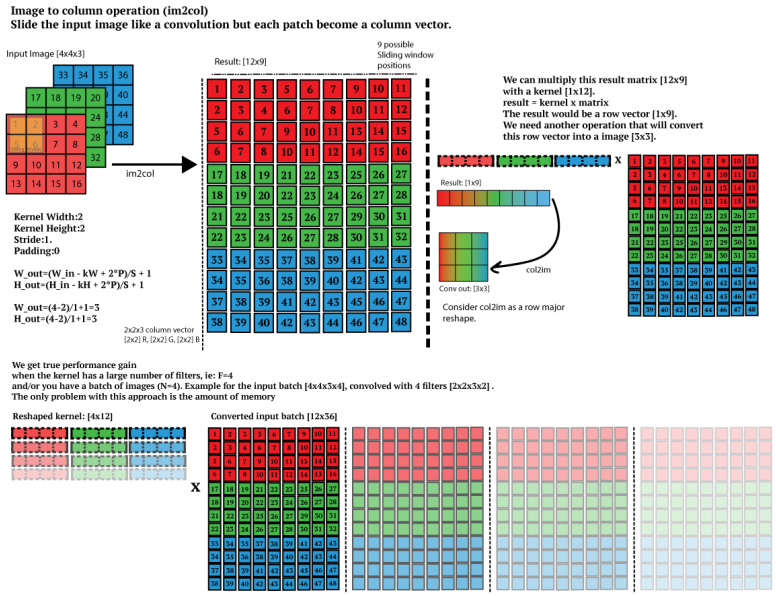
任务一通过 CUDA 实现直接卷积（滑窗法），输入从 256 增加至 4096.

输入：Input 和 Kernel(3x3)

问题描述：用直接卷积的方式对 Input 进行卷积，这里只需要实现 2D, height*width，通道 channel(depth) 设置为 3，Kernel (Filter) 大小设置为 3*3*3，个数为 3，步幅 (stride) 分别设置为 1，2，3，可能需要通过填充 (padding) 配合步幅 (stride) 完成 CNN 操作。注：实验的卷积操作不需要考虑 bias(b)，bias 设置为 0.

1.2 任务 2

任务二使用 im2col 方法结合上次实验实现的 GEMM 实现卷积操作。输入从 256 增加至 4096，具体实现的过程可以参考下面的图片和参考资料。



输入：Input 和 Kernel (Filter)

问题描述：用 im2col 的方式对 Input 进行卷积，这里只需要实现 2D, height*width，通道 channel(depth) 设置为 3，Kernel (Filter) 大小设置为 3*3*3，个数为 3。注：实验的卷积操作不需要考虑 bias(b)，bias 设置为 0，步幅 (stride) 分别设置为 1，2，3。

输出：卷积结果和时间。

1.3 任务 3

NVIDIA cuDNN 是用于深度神经网络的 GPU 加速库。它强调性能、易用性和低内存开销。

使用 cuDNN 提供的卷积方法进行卷积操作，记录其相应 Input 的卷积时间，与自己实现的卷积操作进行比较。如果性能不如 cuDNN，用文字描述可能的改进方法。

2 实验过程

2.1 任务 1

cuda 通过将一组线程组成一个 thread block，并一次执行若干个 thread block 的形式来进行并行计算，形成一种网格式的并行结构。在卷积计算中，每个通道的计算结果矩阵中，各个元素对应的 Filter 相同而卷积的小矩阵不同，因此考虑一个线程计算一个结果矩阵元素。而实验中通道数设置为 3，即我们需要将各个通道的结果矩阵累加，得到最终的卷积结果，首先设计出单通道的卷积和函数代码如下：

```
__global__ void convolution(float *matrix, float *filter, float *result, int height_stride, int width_stride, int matrix_height, int matrix_width, int filter_height, int filter_width, int result_height, int result_width)
{
    //     计算元素的行号
    int i = blockIdx.x * blockDim.x + threadIdx.x;
     //     计算元素的列号
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    //     卷积结果
    float sum = 0;
    if (i < result_height && j < result_width)
    {
        for (int x = 0; x < filter_height; x++)
            for (int y = 0; y < filter_width; y++)
                sum += matrix[index(i * height_stride + x, j * width_stride + y, matrix_width)] * filter[index(x, y, filter_width)];
        //     结果累加
        *(result + index(i, j, result_width)) += sum;
    }
}
```

由于步长设置，我们需要根据输入矩阵的大小对矩阵进行补全，为了方便计算在实验中输入的矩阵都为方阵，因此根据矩阵的大小与卷积核大小，我们可以计算得到补全的长度padding，计算代码如下：

```
    //     矩阵大小 定义为方阵
    int matrix_height = size;
    int matrix_width = size;
    //     卷积和大小
    int filter_height = 3;
    int filter_width = 3;
    //     根据步长计算出需要补全的长度
    int padding = ((((matrix_height - filter_height) / stride + 1) * stride - (matrix_height - filter_height)) % stride) / 2;
```

在实验中为了方便，直接按补全后的矩阵大小初始化我们的矩阵，补全方式则是在输入的矩阵外面进行补全，则最终的正方形矩阵的大小为输入矩阵的大小 +2*padding。在实验中使用了指针数组的方式表示不同通道的矩阵与卷积核，按要求定义我们的输入矩阵与卷积核，并进行随机初始化，实现代码如下，这里要注意输入矩阵 padding 位置的值需要初始化为 0。

```
    for (int i = 0; i < channel; i++)
    {
        matrix[i] = (float *)malloc(matrix_size);
        memset(matrix[i], 0, sizeof(matrix[i]));
        FillMatrix(matrix[i], matrix_height, matrix_width, padding);
    }
    for (int i = 0; i < channel; i++)
    {
        filter[i] = (float *)malloc(filter_size);
        for (int j = 0; j < filter_height * filter_width; j++)
            filter[i][j] = j + 1;
    }
    result = (float *)malloc(result_size);
```

将上述定义的数转移到 GPU 中，调用核函数进行计算，再将最终结果返回，数据转移与线程块定义较简单，不做赘述，调用核函数的代码如下：

```
    for (int i = 0; i < channel; i++)
    {
        convolution<<<numBlocks, threadsPerBlock>>>(cuda_matrix[i],
                                                    cuda_filter[i],
                                                    cuda_result,
                                                    stride, stride,
                                                    matrix_height + 2 *
                                                    ↪  padding,
                                                    matrix_width + 2 *
                                                    ↪  padding,
                                                    filter_height,
                                                    filter_width,
```

```
11                                                  result_height,
12                                                  result_width);
13      }
```

矩阵的初始化与打印代码与之前相同，此处不做赘述，详细见源码文件”CNN_cuda.cu”。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图1，可知以上实现的卷积运算正确。

```
Matrix Size:4   Stride:1
Calculation time:177ms
Matrix after padding of channel 0:
1.000000 7.000000 0.000000 7.000000
5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000
5.000000 7.000000 5.000000 4.000000
Matrix after padding of channel 1:
6.000000 0.000000 7.000000 1.000000
8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000
1.000000 1.000000 5.000000 0.000000
Matrix after padding of channel 2:
0.000000 3.000000 5.000000 3.000000
1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000
5.000000 4.000000 5.000000 2.000000
Filter of channel 0:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 1:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 2:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Result:
624.000000 563.000000
602.000000 511.000000
```

图 1: CNN_cuda 运行结果

调节步长与矩阵规模进行输入，测试时输入的矩阵为方阵，则矩阵规模为 32 时表示对应长宽为都为 32。其运算时间结果（单位：ms）如表1。可见在相同矩阵规模下，stride 的大小并不影响运算时间。这是由于采用的划分方式为每一个线程计算一个矩阵位置的值，则无论 stride 的大小为多少，每个线程分配到的计算量都是相同的。而矩阵规模的扩大，会扩大每个线程在计算对应位置元素值的循环的大小，但由于实验中测试矩阵过小，因此效果并

不明显。

表 1: CNN_cuda 运算时间 (ms) 与 stride、矩阵规模的关系

stride \ 矩阵规模	**32**	**64**	**128**	**256**	**512**
1	176	115	115	175	163
2	184	118	117	187	165
3	150	119	146	172	164

2.2 任务 2

根据 im2col 方法，我们需要将输入矩阵中卷积核对应的小矩阵按行展平，同时将卷积核按列展平，这样该位置的卷积运算变成了一次向量乘法。而如果我们将输出矩阵的每一个元素的计算都变为一次向量乘法，则最终的卷积运算变成了输入矩阵展开组成的大矩阵与卷积核展开得到的大矩阵的矩阵运算。因此我们可以定义一个核函数进行展平操作，实现代码如下。这里展平的索引主要是从结果矩阵找到对应的小矩阵第一个元素，再按卷积核的大小将其展平到大矩阵中。

```
__global__ void im2col(float *matrix, int channel_id,
                       int channel, float *matrix_flatten,
                       int height_stride, int width_stride,
                       int matrix_height, int matrix_width,
                       int filter_height, int filter_width,
                       int result_height, int result_width)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    //    展平到大矩阵
    if (i < result_height && j < result_width)
        for (int x = 0; x < filter_height; x++)
            for (int y = 0; y < filter_width; y++)
                matrix_flatten[index(index(i, j, result_width),
                                     index(x, y, filter_width) + channel_id *
                                     ↪  filter_height * filter_width,
                                     channel * filter_height * filter_width)]
```

```
17                  = matrix[index(i * height_stride + x, j * width_stride + y,
                     ↪  matrix_width)];
18      __syncthreads();
19  }
```

而卷积核的展平较为简单，我们可以将各通道的卷积核按顺序用一维的数组放置，而结果的列保持不变，则可以简单地实现展平。

最终对以上得到的两个矩阵进行乘法运算，可以得到卷积后的结果，矩阵运算的核函数与上一实验实现类似，都是每个元素由一个 cuda 的线程进行运算。

矩阵的初始化与打印代码与之前相同,此处不做赘述,详细见源码文件 im2col_cuda.cu”。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图2，可知以上实现的卷积运算正确。

```
Matrix Size:4   Stride:1
Calculation time:279ms
Matrix after padding of channel 0:
1.000000 7.000000 0.000000 7.000000
5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000
5.000000 7.000000 5.000000 4.000000
Matrix after padding of channel 1:
6.000000 0.000000 7.000000 1.000000
8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000
1.000000 1.000000 5.000000 0.000000
Matrix after padding of channel 2:
0.000000 3.000000 5.000000 3.000000
1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000
5.000000 4.000000 5.000000 2.000000
Filter of channel 0:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 1:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 2:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Result:
624.000000 563.000000
602.000000 511.000000
```

图 2: im2col_cuda 运行结果

调节步长与矩阵规模进行输入，测试时输入的矩阵为方阵，则矩阵规模为 32 时表示对应长宽为都为 32。其运算时间结果（单位：ms）如表2。可见在相同矩阵规模下，stride 的大小并不影响运算时间。这是由于采用的划分方式为每一个线程计算一个矩阵位置的值，则无论 stride 的大小为多少，每个线程分配到的计算量都是相同的。而矩阵规模的扩大，会扩大每个线程在计算对应位置元素值的循环的大小，但由于实验中测试矩阵过小，因此效果并不明显。

根据 im2col 算法的实现原理可以知道，im2col 将卷积运算变成了简单的矩阵乘法，实现了空间换时间。而在实验中由于设置的矩阵规模较小，用于卷积运算的时间比较小。而 im2col 多了一步展平的操作，因此从结果上看，并没有取得较好的加速效果。

表 2: im2col_cuda 运算时间 (ms) 与 stride、矩阵规模的关系

矩阵规模 / stride	**32**	**64**	**128**	**256**	**512**
1	181	171	199	179	173
2	206	209	188	176	179
3	157	185	189	163	173

2.3 任务 3

2.3.1 cuDNN 的实现

实验中简单地采用了 cuDNN 库中的 cudnnConvolutionForward 函数进行矩阵乘法运算，函数定义如下

```
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);
```

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：

$$dstValue = alpha[0] * computedValue + beta[0] * priorDstValue$$

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 cudaCreate() 来初始化。

首先我们需要创建句柄，由于 cudnnConvolutionForward 函数会自动进行 padding 操作，在实验中我们只需简单地初始化输入矩阵与卷积核再创建对应的描述子即可，实现代码如下：

```
    //创建句柄
    cudnnHandle_t cudnn;
    cudnnCreate(&cudnn);

    //输入矩阵的描述子
    cudnnTensorDescriptor_t matrix_desc;
    cudnnCreateTensorDescriptor(&matrix_desc);
    cudnnSetTensor4dDescriptor(matrix_desc, CUDNN_TENSOR_NCHW,
    ↪  CUDNN_DATA_FLOAT, 1, channel, matrix_height, matrix_width);

    //卷积核的描述子
    cudnnFilterDescriptor_t filt_desc;
    cudnnCreateFilterDescriptor(&filt_desc);
    cudnnSetFilter4dDescriptor(filt_desc, CUDNN_DATA_FLOAT,
    ↪  CUDNN_TENSOR_NCHW, 1, channel, filter_height, filter_width);
```

与上述描述子创建方式相同，我们需要创建输出的描述子，卷积描述子，再根据以上描述子选择计算的算法，准备计算所用的空间，实现代码如下：

```
   //输出结果的描述子
    cudnnTensorDescriptor_t result_desc;
```

```
    cudnnCreateTensorDescriptor(&result_desc);
    cudnnSetTensor4dDescriptor(result_desc, CUDNN_TENSOR_NCHW,
    ↪  CUDNN_DATA_FLOAT, result_n, result_c, result_h, result_w);

    //卷积的描述子
    cudnnConvolutionDescriptor_t conv_desc;
    cudnnCreateConvolutionDescriptor(&conv_desc);
    cudnnSetConvolution2dDescriptor(conv_desc, padding, padding, stride,
    ↪  stride, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT);

    //选择计算的算法
    cudnnConvolutionFwdAlgo_t algo;
    cudnnGetConvolutionForwardAlgorithm(cudnn, matrix_desc, filt_desc,
    ↪  conv_desc, result_desc, CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0,
    ↪  &algo);

    //准备计算所用的空间
    size_t ws_size;
    cudnnGetConvolutionForwardWorkspaceSize(cudnn, matrix_desc, filt_desc,
    ↪  conv_desc, result_desc, algo, &ws_size);
    float *ws_data;
    cudaMalloc(&ws_data, ws_size);
```

完成所有的描述子创建后，我们只需要将计算的矩阵与卷积核拷贝到 GPU，调用 **cudnnConvolutionForward** 函数即可实现卷积操作。

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

    cudaMemcpy(cuda_matrix, matrix, 1 * channel * matrix_height *
    ↪  matrix_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_filter, filter, 1 * channel * filter_height *
    ↪  filter_width * sizeof(float), cudaMemcpyHostToDevice);

```

```
7       cudnnConvolutionForward(cudnn, &alpha, matrix_desc, cuda_matrix,
        ↪  filt_desc, cuda_filter, conv_desc, algo, ws_data, ws_size, &beta,
        ↪  result_desc, cuda_result);
```

矩阵的初始化与打印代码与之前相同，此处不做赘述，详细见源码文件 cuDNN.cu”。为验证其正确性，选择小规模的矩阵进行计算并打印结果，如图3，可知以上实现的卷积运算正确。

```
Matrix Size:4   Stride:1
Calculation time:7ms
Matrix of channel 0:
1.000000 7.000000 0.000000 7.000000
5.000000 7.000000 1.000000 3.000000
6.000000 1.000000 5.000000 4.000000
5.000000 7.000000 5.000000 4.000000
Matrix of channel 1:
6.000000 0.000000 7.000000 1.000000
8.000000 8.000000 6.000000 6.000000
8.000000 8.000000 8.000000 4.000000
1.000000 1.000000 5.000000 0.000000
Matrix of channel 2:
0.000000 3.000000 5.000000 3.000000
1.000000 7.000000 4.000000 7.000000
6.000000 0.000000 0.000000 2.000000
5.000000 4.000000 5.000000 2.000000
Filter of channel 0:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 1:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Filter of channel 2:
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
Result:
624.000000 563.000000
602.000000 511.000000
```

图 3: cuDNN 运行结果

调节步长与矩阵规模进行输入，测试时输入的矩阵为方阵，则矩阵规模为 32 时表示对应长宽为都为 32。其运算时间结果（单位：ms）如表3。

调节矩阵规模进行输入，对实现的任务 1、任务 2 和任务 3 的程序测试运算时间，结果（单位：ms）如图4。可以看到，我们实现的任务 1、任务 2 的性能远低于 cuDNN，说明我们实现的卷积操作有着很大的提升空间。而任务 2 的性能在小规模矩阵的情况下与任务

表 3: cuDNN 运算时间 (ms) 与 stride、矩阵规模的关系

矩阵规模 / stride	**32**	**64**	**128**	**256**	**512**
1	6	6	7	7	12
2	6	6	6	7	8
3	6	6	6	6	7

1 基本相似，由于矩阵规模过小，无法体现任务 2 的算法提升。理想状态下，任务 2 在中规模矩阵的卷积运算性能应高于任务 1；在大规模的卷积运算中，由于任务 2 需要进行矩阵的展平操作，可能会发生缓存的缺少导致性能的进一步下降。

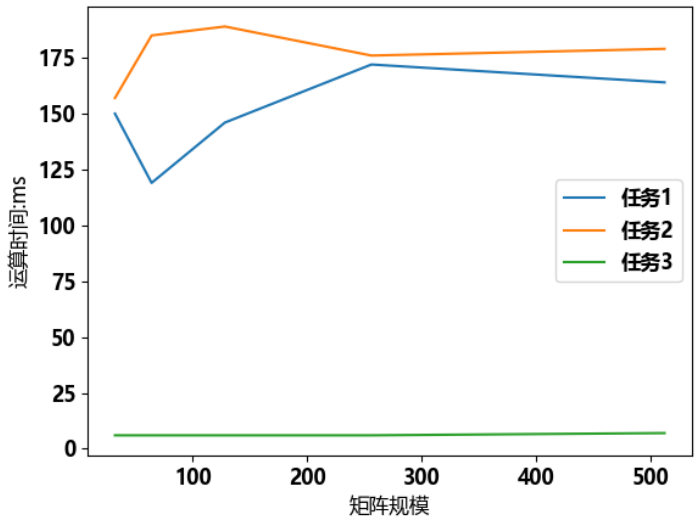


图 4: 实现的比较

2.3.2 改进方法

从比较中，程序还需进一步优化。与之前的改进方向类似，可以从访存与算法两方面进行改进。在实验中卷积核存在对输入矩阵的同一元素进行访问的情况，因此可以采用 shared memory 来提升访存的速度。在任务 1 中，对于每个通道的结果矩阵采用了循环的方式进行获取，在这里可以考虑使用 OpenMP 来同时对不同通道的矩阵进行卷积，再将结果累加，这样可以减少大量的运算时间，但随之而来的则是对临界区的问题，可以考虑 reduce 解决。

而任务 2 使用的 im2col 算法将卷积变成了矩阵乘法运算，这就意味着我们之前进行的一系列对矩阵乘法的改进方式都可以考虑用于最终的乘法过程。而其最终在小规模矩阵的

运算效果不如任务 1，也说明在不同规模下的矩阵卷积可以考虑采用不同的算法进行改进。

与之前相同地，由于寄存器在线程间是不能共享的，因此可以通过优化寄存器进一步提高效率。

3 实验感想

- 本次实验主要使用了 GPU 并行加速卷积运算，GPU 的特点是集成大量的流处理器和计算单元, 具有并行处理的优势, 非常适合桌面和移动平台进行大规模超级计算。CUDA 是在 GPU 上实现的一种技术, 该技术可以将 GPU 由专用的处理器变为通用的处理单元, 发挥 GPU 的强大的运算能力。
- 实验采用的 NVIDIA cuDNN 是用于深度神经网络的 GPU 加速库，从运算性能上可以看到，好的加速库能够大大地提高我们计算性能，在人工智能这种需要进行大量运算的领域更为重要。因此为了更快地促进人工智能的发展，我们也需要关注 CUDA 与 GPU 的发展，如果没有 cuDNN 的卷积库，卷积时采用任务 1、任务 2 实现的操作，那么人工智能终将完蛋。