中山大学计算机学院本科生实验报告

（2021学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	**LAB6**	专业	计算机科学与技术（超算）
学号	19335162	姓名	潘思晗
Email	pansh25@mail2.sysu.edu.cn	完成日期	2021.12.24

一、实验目的

> *1.*通过 *CUDA*实现通用矩阵乘法（*Lab1*）的并行版本，*CUDA Thread Block size*从 *32* 增加至 *512*，矩阵规模从 *512*增加至 *8192*。
>
> *2.*通过 *NVDIA*的矩阵计算函数库 *CUBLAS*计算矩阵相乘，矩阵规模从*512*增加至 *8192*，并与任务*1*和任务*2*的矩阵乘法进行性能比较和分析，如果性能不如 *CUBLAS*，思考并文字描述可能的改进方法。
>
> *3.*通过 *CUDA*实现直接卷积（滑窗法），输入从 *256*增加至 *4096*或者输入从*32*增加至 *512*
>
> *4.*使用 *im2col*方法结合 *GEMM*实现卷积操作。输入从 *256*增加至 *4096*或者输入从 *32* 增加至 *512*
>
> *5.*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。

二、实验过程和核心代码

1.CUDA实现通用矩阵乘法

（1）算法描述

通用矩阵乘法通常定义为: $C = AB$， $C_{m,n} = \sum_{n=1}^{N} A_{m,n}B_{n,k}$

串行版本`GEMM()`函数实现：传入矩阵A、B并为矩阵C申请空间，通过循环嵌套实现矩阵的相乘，最后返回计算结果的指针，如下所示

```
int** GEMM(int** matA,int** matB)
{
    int** ans = (int**)malloc(sizeof(int*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(int*)malloc(sizeof(int)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j++){
            ans[i][j] = 0;
            for(int t=0;t<n;t++){
                ans[i][j] += matA[i][t]*matB[t][j];
            }
        }
    }
    return ans;
}
```

（2）核心代码

矩阵维度的三个参数m，n，k和CUDA的block_size通过输入参数argv列表获得

```
int m=atoi(argv[1]);
int n=atoi(argv[2]);
int k=atoi(argv[3]);
int block_size=atoi(argv[4]);
```

定义并行矩阵乘法函数

```
__global__ void GEMM_cuda(double *matA,double *matB, double
*matC, int m, int n, int k)
```

先根据目前的参数`blockIdx`和`threadIdx`计算出当前需要计算的row 和 column，其中`blockIdx`指明线程所在grid中的位置，而`threadIdx`指明线程所在block中的位置：

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

再循环进行通用矩阵乘法：

```
if(row < m && col < k) {
    for(int i = 0; i < n; i++) {
        temp += matA[row * n + i] * matB[i * k + col];
    }
    matC[row * k + col] = temp;
}
```

主函数中，先使用cudaMallocHost函数在CPU上分配host内存

```
cudaMallocHost((void **) &host_a, sizeof(double)*m*n);
cudaMallocHost((void **) &host_b, sizeof(double)*n*k);
cudaMallocHost((void **) &host_c, sizeof(double)*m*k);
```

接下来申请device内存，并将host数据拷贝到device

```
cudaMalloc((void **) &device_a, sizeof(double)*m*n);
cudaMalloc((void **) &device_b, sizeof(double)*n*k);
cudaMalloc((void **) &device_c, sizeof(double)*m*k);

cudaMemcpy(device_a, host_a, sizeof(double)*m*n,
cudaMemcpyHostToDevice);
cudaMemcpy(device_b, host_b, sizeof(double)*n*k,
cudaMemcpyHostToDevice);
```

定义grid的参数以及kernel的执行配置，并在cuda中执行函数`GEMM_cuda`

```
unsigned int grid_rows = (m + block_size - 1) / block_size;
unsigned int grid_cols = (k + block_size - 1) / block_size;
dim3 gridSize(grid_cols, grid_rows);
dim3 blockSize(block_size, block_size);

//run the function in cuda
GEMM_cuda<<<gridSize, blockSize>>>(device_a, device_b, device_c,
m, n, k);
```

完成之后将device得到的结果拷贝到host

```
cudaMemcpy(host_c_gpu, device_c, sizeof(double)*m*k,
cudaMemcpyDeviceToHost);
```

使用cuda API的事件管理功能计时，并打印出最终的执行结果

```
cudaDeviceSynchronize();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf("matrixA: %dx%d  matrixB: %dx%d  Block_size = %d\n", m, n,
n, k, block_size);
printf("The time of CUDA_GEMM: %f ms.\n", time);
```

最后释放内存

```
cudaFree(device_a);
cudaFree(device_b);
cudaFree(device_c);
cudaFreeHost(host_a);
cudaFreeHost(host_b);
cudaFreeHost(host_c);
```

2.CUBLAS计算矩阵相乘

（1）算法描述

CUBLAS的`cublasDgemm`函数完成矩阵乘法$C = \alpha op(A)op(B) + \beta C$，当需要计算$C = AB$时，显然可以直接设置$\alpha = 1, \beta = 0$。其中op操作对决定矩阵是否转置，即决定该矩阵是按照行优先还是列优先。当我们选择CUBLAS_OP_N时表示不转置，按列优先存储；当我们选择CUBLAS_OP_T时表示需要转置，按行优先存储。

每个参数的含义如下（参考官方的API说明）：

① handle：cublas的句柄，通过cublasCreate创建；
② transA：矩阵B是否转置，转置为CUBLAS_OP_T，否则为CUBLAS_OP_N；
③ transB：矩阵A是否转置；
④ M：表示矩阵B的列数，若转置则为B的行数；
⑤ N：表示矩阵A的行数，若转置则为A的列数；
⑥ K：表示 B的行数或A的列数（A的列数=B的行数），即为被吃掉的维度；
⑦ alpha和beta：计算公式$C = \alpha op(A)op(B) + \beta * C$中的两个参数的引用；
⑧ lda：表示矩阵B的leading dimension，在列主序下，B转置表示B的列否则为B的行，值均为K，和参数M相反；
⑨ ldb：表示矩阵A的leading dimension，在列主序下，A转置表示A的行否则为A的列，值均为K，和参数N相反；
⑩ ldc：表示矩阵C的leading dimension，在列主序下始终为N，表示C的列。

（2）核心代码

首先是host内存空间的分配：

```
double *host_A = (double*)malloc (N*M*sizeof(double));
double *host_B = (double*)malloc (N*M*sizeof(double));
double *host_C = (double*)malloc (M*M*sizeof(double));
```

申请device内存，并将host数据拷贝到device

```
int lines = m / numprocs;         //lines表示分块后的行数
matB = new double[n*k];           //每个进程都发送的矩阵B
block_A = new double[lines*n];    //矩阵A的分块
block_C = new double[lines*k];    //计算得到的结果矩阵C
```

初始化CUBLAS库的对象

```
cublasHandle_t handle;
cublasCreate(&handle);
```

核心计算部分，将矩阵数据复制到显存中，定义alpha=1，beta=0，调用`cublasDgemm`函数完成矩阵乘法，并将得到的结果复制回内存

```
//将矩阵复制到显存中
cublasSetVector(N * M, sizeof(double), host_A, 1, device_A, 1);
cublasSetVector (N * M, sizeof(double), host_B, 1, device_B, 1);
cudaDeviceSynchronize();
//赋值alpha和beta，计算矩阵乘法
double alpha=1;
double beta=0;
cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, M, M, N, &alpha,
device_A, N, device_B, M, &beta, device_C, M);
cudaDeviceSynchronize();
// 将结果复制回内存
cublasGetVector(M * M, sizeof(double), device_C, 1, host_C, 1);
```

停止计时并打印结果

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);

printf("matrixA: %dx%d  matrixB: %dx%d\n", M, N, N, K);
printf("The time of CUBLAS: %f ms.\n", time);
```

最后将申请的内存空间释放，并销毁handle

```
// 释放内存
free (host_A);
free (host_B);
free (host_C);
cudaFree (device_A);
cudaFree (device_B);
cudaFree (device_C);
// 释放 CUBLAS 库对象
cublasDestroy (handle);
```

3.CUDA实现直接卷积

（1）问题描述

通过CUDA实现直接卷积（滑窗法），输入从256增加至4096或者输入从32增加至512。

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。

输入：Input和Kernel(3×3)

输出：卷积结果以及计算时间

（2）核心代码

首先定义必要的参数，为了方便测试不同参数，选择使用#define直接指定（或修改）：

```
#define mat_height 6
#define mat_width 6
#define filter_size 3
#define stride 1

#define block_size_x 3
#define block_size_y 3
```

为了保证有数值的部分都使用滑窗法划过，应该增加 padding 部分，为了使卷积操作不缩减数据的维度，padding的size P设置为$P = (F - 1)/2$，F是filter的大小。最终设置的 padding 和增加 padding 后的 input 大小如下：

```
// padding
#define padding_size ((filter_size - 1) / 2)
#define input_height (mat_height + padding_height)
#define input_width (mat_width + padding_width)
```

此题中filter_size=3，因此padding_size=1。

接下来进行矩阵和 filter 的分配内存、初始化等操作。基于CUDA的编程特性，应当给很多输入、中间值和输出进行动态内存分配和初始化：

```
// input1-3，经过 padding ，作为输入的矩阵，共三层
float *input1 = (float *)malloc(input_height * input_width * sizeof(float));
float *input2 = (float *)malloc(input_height * input_width * sizeof(float));
float *input3 = (float *)malloc(input_height * input_width * sizeof(float));

// 经过cuda运算后，三层结果相加的最终结果
float *conv_result = (float *)malloc(mat_height * mat_width * sizeof(float));

// 三个不同的卷积核
float *filter1 = (float *)malloc(filter_height * filter_width * sizeof(float));
float *filter2 = (float *)malloc(filter_height * filter_width * sizeof(float));
float *filter3 = (float *)malloc(filter_height * filter_width * sizeof(float));
// 分配CUDA空间
float *cuda_input1;
float *cuda_output1;
float *cuda_filter1;
cudaMalloc((void **)&cuda_input1, input_height * input_width * sizeof(float));
cudaMalloc((void **)&cuda_output1, mat_height * mat_width * sizeof(float));
cudaMalloc((void **)&cuda_filter1, filter_height * filter_width * sizeof(float));
float *cuda_input2;
float *cuda_output2;
float *cuda_filter2;
cudaMalloc((void **)&cuda_input2, input_height * input_width * sizeof(float));
cudaMalloc((void **)&cuda_output2, mat_height * mat_width * sizeof(float));
cudaMalloc((void **)&cuda_filter2, filter_height * filter_width * sizeof(float));
```

```
float *cuda_input3;
float *cuda_output3;
float *cuda_filter3;
cudaMalloc((void **)&cuda_input3, input_height * input_width *
sizeof(float));
cudaMalloc((void **)&cuda_output3, mat_height * mat_width *
sizeof(float));
cudaMalloc((void **)&cuda_filter3, filter_height * filter_width
* sizeof(float));

// CUDA 中，经过卷积运算后将三层结果相加后得到的最终结果
float *cuda_result;
cudaMalloc((void **)&cuda_result, mat_height * mat_width *
sizeof(float));
```

将host数据的拷贝到device中：

```
cudaMemcpy(cuda_input1, input1, input_height * input_width *
sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_filter1, filter1, filter_height * filter_width *
sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_input2, input2, input_height * input_width *
sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_filter2, filter2, filter_height * filter_width *
sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_input3, input3, input_height * input_width *
sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_filter3, filter3, filter_height * filter_width *
sizeof(float), cudaMemcpyHostToDevice);
```

设置grid和block的参数：

```
dim3 blockSize(block_size_x, block_size_y);
dim3 gridSize((mat_height+block_size_x-1)/block_size_x,
(mat_width + block_size_y - 1)/block_size_y);
```

定义CUDA计算卷积的 __global__ 函数`convolution_kernel`

```
__global__ void convolution_kernel(float *output, float *input,
float *filter)
```

先计算矩阵的坐标(x, y)

```
int y = blockIdx.y * blockDim.y + threadIdx.y;
int x = blockIdx.x * blockDim.x + threadIdx.x;
```

当坐标能够整除 stride 时，说明此时符合计算要求，进行 filter 的循环

```
if (y % stride == 0 && x % stride == 0)
{
    for (int i = 0; i < filter_height; i++)
    {
        for (int j = 0; j < filter_width; j++)
        {
            sum += input[(y + i) * input_width + x + j] *
filter[i * filter_width + j];
        }
    }
    output[y / stride * mat_width + x / stride] = sum;
}
```

最后设计函数把三层计算的结果相加：

```
__global__ void cuda_add(float *arr1, float *arr2, float *arr3,
float *result){
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    if (y % stride == 0 && x % stride == 0)
        result[y / stride * mat_width + x / stride] = arr1[y /
stride * mat_width + x / stride] + arr2[y / stride * mat_width +
x / stride] + arr3[y / stride * mat_width + x / stride];
}
```

下面开始卷积的计算，为了计时选择使用CUDA的事件管理功能：

```
cudaDeviceSynchronize();
float time_gpu;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
```

接下来分三次调用卷积计算函数`convolution_kernel`，分三层进行计算，并将得到的三个结果相加：

```
convolution_kernel<<<gridSize, blockSize>>>(cuda_output1, cuda_input1, cuda_filter1);
convolution_kernel<<<gridSize, blockSize>>>(cuda_output2, cuda_input2, cuda_filter2);
convolution_kernel<<<gridSize, blockSize>>>(cuda_output3, cuda_input3, cuda_filter3);
cuda_add<<<gridSize, blockSize>>>(cuda_output1, cuda_output2, cuda_output3, cuda_result);
```

结束计时并打印计算时间：

```
cudaDeviceSynchronize();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_gpu, start, stop);
printf("CUDA CONVOLUTION TIME:        %f ms\n", time_gpu);
```

最后将计算结果拷贝到host的内存空间，并输出：

```
// 将结果拷贝到内存空间
cudaMemcpy(conv_result, cuda_result, mat_height * mat_width * sizeof(float), cudaMemcpyDeviceToHost);
// 输出卷积结果
printf("RESULT:\n");
print_mat(conv_result, mat_height, mat_width);
```

释放内存空间：

```
free(filter1);
free(input1);
free(filter2);
free(input2);
free(filter3);
free(input3);
free(conv_result);

cudaFree(cuda_output1);
cudaFree(cuda_input1);
```

```
cudaFree(cuda_filter1);
cudaFree(cuda_output2);
cudaFree(cuda_input2);
cudaFree(cuda_filter2);
cudaFree(cuda_output3);
cudaFree(cuda_input3);
cudaFree(cuda_filter3);
cudaFree(cuda_result);
```

4. im2col方法实现卷积操作

（1）问题描述

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或者输入从32增加至512。

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，步幅(stride)分别设置为1，2，3。
注：实验的卷积操作不需要考虑bias(b)，bias设置为0。

输入：Input和Kernel(Filter)

输出：卷积结果和时间

（2）核心代码

首先各个参数定义如下：

```
#define mat_height 256
#define mat_width 256
#define filter_size 3
#define padding 1
#define stride 1
#define channels 3

#define block_size_x 3
#define block_size_y 3
```

im2col核函数首先计算卷积后的尺寸：

```
int height_col=(height + 2*pad - filter_size) / stride+1;
int width_col=(width + 2*pad - filter_size) / stride+1;
int channels_col = channels * filter_size * filter_size;
```

获取每个位置对应的值，先计算卷积核上的坐标，然后进行循环获取imput对应位置的值：

```
for (c = 0; c < channels_col; ++c) {
    int w_offset = c % filter_size;
    int h_offset = (c / filter_size) % filter_size;
    int c_im = c / filter_size / filter_size;
    for (h = 0; h < height_col; ++h) {
        for (w = 0; w < width_col; ++w) {
            int im_row = h_offset + h * stride;
            int im_col = w_offset + w * stride;
            int col_index = (c * height_col + h) * width_col + w;
            data_col[col_index] = im2col_get_data(data_im, im_row, im_col, c_im);
        }
    }
}
```

接下来，应用之前完成的 GEMM 用来对进行重新排列过的矩阵和 filter 进行通用矩阵乘法，就可以得到卷积结果。

先为各个矩阵分配内存空间，im 指向原矩阵，col 指向经过 im2col 重排的矩阵，按照计算好的内存大小进行分配：

```
float *im = (float *)malloc(mat_height * mat_width * channels * sizeof(float));
float *col = (float *)malloc(channels_col * height_col * width_col * sizeof(float));
float *filter = (float *)malloc(channels * filter_size * filter_size * sizeof(float));
```

再进行重排：

```
im2col(im, col);
```

在 cuda 的 device 中给矩阵分配空间，并将 filter 和 col 的数据复制到device中：

```
cudaMalloc((void **)&cuda_a, sizeof(float) * channels * (filter_size * filter_size) );
cudaMalloc((void **)&cuda_b, sizeof(float) * channels_col * (width_col * height_col));
cudaMalloc((void **)&cuda_c, sizeof(float) * channels * (width_col * height_col));
cudaMemcpy(cuda_a, filter, sizeof(float) * channels * (filter_size * filter_size), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_b, col, sizeof(float) * channels_col * (width_col * height_col), cudaMemcpyHostToDevice);
```

调用 GEMM 矩阵乘法并将结果拷贝回host的内存中：

```
// 执行 GEMM_cuda 函数
GEMM_cuda<<<blocks_num, BLOCK_SIZE, 0>>>(cuda_a, cuda_b, cuda_c, channels, (filter_size * filter_size), (width_col * height_col), BLOCK_SIZE);

// 将结果从host中复制回内存
cudaMemcpy(c, cuda_c, sizeof(float) * channels * (width_col * height_col), cudaMemcpyDeviceToHost);
```

最后计时，输出卷积结果以及运算时间：

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_gpu, start, stop);

// 输出卷积结果以及计算时间
printf("im2col CONVOLUTION TIME:        %f ms\n", time_gpu);
printf("RESULT:\n");
print_mat(conv_result, mat_height, mat_width);
```

释放申请的内存（此处略）。

5. 使用cuDNN进行卷积

（1）问题描述

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

CuDNN中计算卷积操作的由`cudnnConvolutionForward()`来实现，其原型为：

```
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[0]*computedValue+beta[0]*priorDstVal
```

- `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。

（2）核心代码

首先要有以下定义：

```
#define checkCUDNN(expression)\
{\
    cudnnStatus_t status = (expression);\
    if (status != CUDNN_STATUS_SUCCESS)\
    {\
        std::cerr << "Error on line " << __LINE__ << ": "   << cudnnGetErrorString(status) << std::endl;
std::exit(EXIT_FAILURE);\
    }\
}
```

main函数中先准备输入数据：

```
// input
cudnnTensorDescriptor_t input_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor,
                                      CUDNN_TENSOR_NHWC,
                                      CUDNN_DATA_FLOAT,1,3,
                                      input_height,
                                      input_width));
```

`input_descriptor`保存了输入的信息，其中`CUDNN_TENSOR_NHWC`是数据数据的结构，`CUDNN_DATA_FLOAT`为计算的数据类型，1和3分别表示batch_size和channels数目。

下面准备输出数据：

```
// output
cudnnTensorDescriptor_t output_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&output_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor,
                                      CUDNN_TENSOR_NHWC,
                                      CUDNN_DATA_FLOAT,1,1,
                                      output_height,
                                      output_width));
```

准备卷积核：

```
// kernel
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                      CUDNN_DATA_FLOAT,
                                      CUDNN_TENSOR_NCHW,
                                      1,3,3,3));
```

下面四个参数分别表示：out_channels=1，in_channels=3，kernel_height=3，kernel_width=3

卷积描述子设置如下：

```
// convolucion descriptor
cudnnConvolutionDescriptor_t convolution_descriptor;
checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution_descriptor));
checkCUDNN(cudnnSetConvolution2dDescriptor(
                                    convolution_descriptor,
                                    padding,padding,
                                    stride,stride,
                                    1,1,
                                    CUDNN_CROSS_CORRELATION,
                                    CUDNN_DATA_FLOAT));
```

主要设置步长、填充等参数。

下面选择卷积计算的算法：

```
// algorithm
cudnnConvolutionFwdAlgo_t convolution_algorithm;
checkCUDNN(cudnnGetConvolutionForwardAlgorithm(
                cudnn,
                input_descriptor,
                kernel_descriptor,
                convolution_descriptor,
                output_descriptor,
                CUDNN_CONVOLUTION_FWD_PREFER_FASTEST,
                0,
                &convolution_algorithm));
```

告诉cuDNN优先使用计算速度快的算法，并且没有内存限制。

准备计算所用的空间，即计算cuDNN的操作需要多少内存：

```
// workspace size && allocate memory
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(
                                    cudnn,
                                    input_descriptor,
                                    kernel_descriptor,
                                    convolution_descriptor,
                                    output_descriptor,
                                    convolution_algorithm,
                                    &workspace_bytes));
```

最重要的convolution计算，使用前向卷积：

```
// convolution
const float alpha = 1, beta = 1;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
checkCUDNN(cudnnConvolutionForward(cudnn,
                                   &alpha,
                                   input_descriptor,
                                   d_input,
                                   kernel_descriptor,
                                   d_kernel,
                                   convolution_descriptor,
                                   convolution_algorithm,
                                   d_workspace,
                                   workspace_bytes,
                                   &beta,
                                   output_descriptor,
                                   d_output));
```

最后输出运算时间和相关结果，方法与前面类似。

销毁：

```
cudnnDestroyTensorDescriptor(input_descriptor);
cudnnDestroyTensorDescriptor(output_descriptor);
cudnnDestroyConvolutionDescriptor(convolution_descriptor);
cudnnDestroyFilterDescriptor(kernel_descriptor);

cudnnDestroy(cudnn);
```

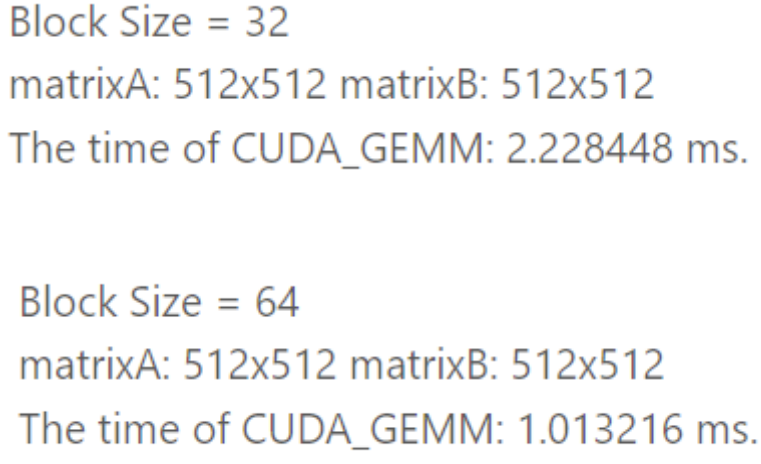
三、实验结果

> 实验环境：
>
> 超算习堂 *CUDA*编程
>
> 超算集群

1.CUDA实现通用矩阵乘法

矩阵规模与**Block Size**递增

令矩阵规模为512，Block Size从32开始递增，结果如下：



```
Block Size = 128
matrixA: 512x512 matrixB: 512x512
The time of CUDA_GEMM: 1.047872 ms.
```

```
Block Size = 256
matrixA: 512x512 matrixB: 512x512
The time of CUDA_GEMM: 1.049632 ms.
```

Block Size超过64后，提速不明显。

矩阵规模为1024，Block Size从32增加到512，结果如下：

```
Block Size = 32
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUDA_GEMM: 10.227488 ms.
```

```
Block Size = 64
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUDA_GEMM: 4.779456 ms.
```

```
Block Size = 128
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUDA_GEMM: 3.339136 ms.
```

```
Block Size = 256
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUDA_GEMM: 2.617440 ms.
```

```
Block Size = 512
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUDA_GEMM: 2.595552 ms.
```

随着Block Size的增加，计算时间逐步缩短，但超过一定阈值后，差异不大。

矩阵规模为2048，Block Size从32增加到512，结果如下：

```
Block Size = 32
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUDA_GEMM: 72.109825 ms.
```

```
Block Size = 64
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUDA_GEMM: 8.916864 ms.
```

```
Block Size = 128
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUDA_GEMM: 9.137888 ms.
```

```
Block Size = 256
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUDA_GEMM: 8.725952 ms.
```

```
Block Size = 512
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUDA_GEMM: 8.718016 ms.
```

同样，可以看到Block Size从32增加到64时有一个明显的提速，但是再增加Block Size对计算速率的影响不大，看不出明显差异。

接下来直接将矩阵规模提升至8192，查看Block Size从128增加到512对于计算的影响，结果如下：

```
Block Size = 128
matrixA: 8192x8192 matrixB: 8192x8192
The time of CUDA_GEMM: 129.434341 ms.
```

```
Block Size = 256
matrixA: 8192x8192 matrixB: 8192x8192
The time of CUDA_GEMM: 129.408157 ms.
```

```
Block Size = 512
matrixA: 8192x8192 matrixB: 8192x8192
The time of CUDA_GEMM: 130.412506 ms.
```

在矩阵规模扩大至8192后，不论Block Size是128，256还是512，运行时间相差无几。

需要注意的是，由于cuda系统内置的计时方式计时单位是毫秒ms，因此实际差异对比没有那么大，只是基于数据比较结果的阐述。

2.CUBLAS计算矩阵相乘

CUBLAS计算结果

矩阵规模从512增加至8192，结果如下：

输出

```
matrixA: 512x512 matrixB: 512x512
The time of CUBLAS: 2.121024 ms.
```

输出

```
matrixA: 1024x1024 matrixB: 1024x1024
The time of CUBLAS: 5.784352 ms.
```

输出

```
matrixA: 2048x2048 matrixB: 2048x2048
The time of CUBLAS: 22.278175 ms.
```

输出

```
matrixA: 4096x4096 matrixB: 4096x4096
The time of CUBLAS: 106.407005 ms.
```

输出

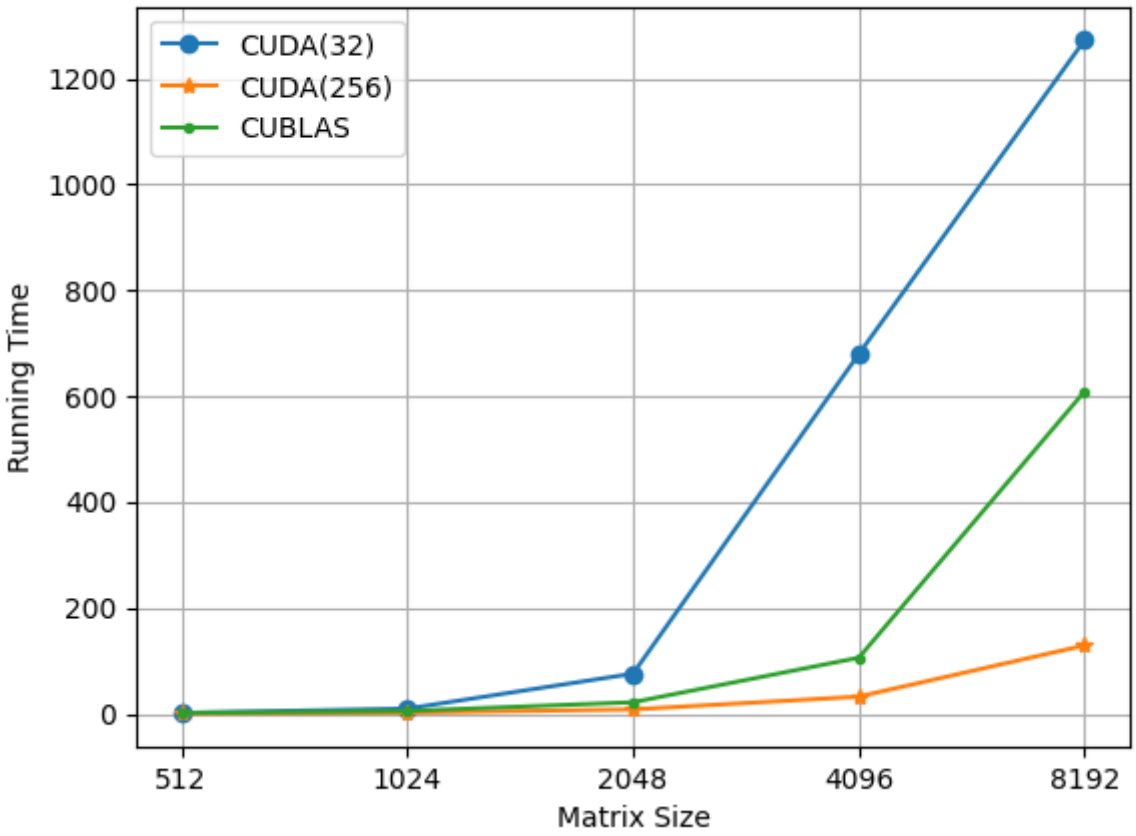
```
matrixA: 8192x8192 matrixB: 8192x8192
The time of CUBLAS: 609.340515 ms.
```

CUDA编写与**CUBLAS**库函数性能对比

选取Block Size = 32的情况下，CUDA计算的时间性能与CUBLAS对比，得到的结果如下图（单位ms）：

矩阵规模\并行方式	**CUDA(32)**	**CUDA(256)**	**CUBLAS**
512	2.23	1.05	2.12
1024	10.23	2.62	5.78
2048	76.25	8.73	22.28
4096	679.853	32.89	106.4
8192	1272.55	129.65	609.34

将上述数据利用python分析并对比制作图像，结果如下所示：



由上图可以看出，Block Size=32时，CUBLAS函数的计算速度明显优于CUDA编写的GEMM函数，但是当Block Size增加至256时，CUDA编写的函数性能更优，因此认为之前编写的GEMM函数效率是很优的。

改进方向

还可能存在以下几种改进方法：

- 最大化内存吞吐量：

1. 考虑局部性原理和对齐问题，降低访存所消耗的时间。
2. 尽量减少 host 和 device 之间的内存传输；

- 优化指令流：

1. 尽量减少条件分支；
2. 使用 CUDA 算数指令中的快速指令

3.CUDA实现直接卷积

卷积结果

输入维度6×6，kernel维度3×3，输出维度6×6，结果如下：

输出

```
CUDA CONVOLUTION TIME: 0.040256 ms
RESULT:
1.895500 1.753900 1.611900 1.735100 1.623600 1.834300
1.569600 1.568300 1.487100 1.487700 1.860600 1.801700
1.945500 1.772600 1.516700 1.474500 1.712400 1.616300
2.076800 1.537500 1.519500 1.584300 1.555400 1.505300
2.045300 1.941900 1.959600 1.452400 1.474200 1.656800
1.851900 1.756100 1.944400 1.388600 1.322300 1.501100
```

可以看到输出了卷积结果以及耗时0.04ms。

时间对比

在不同规模的输入和不同步长的情况下分别做测试，得到的结果汇总如下表：

INPUT	STRIDE=1	STRIDE=2	STRIDE=3
512	0.216	0.223	0.228
1024	0.673	0.812	0.814
2048	3.185	3.284	3.147
4096	12.622	12.743	12.635

当 stride 固定不变时，随着矩阵规模的增大， CUDA 直接卷积运算的时间变长。

而当矩阵规模固定不变，stride 增大时，卷积运算时间没有明显变化。

注意：由于时间单位是毫秒ms，因此实际差异对比没有那么大，只是基于数据比较结果的阐述。

4. im2col方法实现卷积操作

时间对比

在不同规模的输入和不同步长的情况下分别做测试，得到的结果汇总如下表：

INPUT	STRIDE=1	STRIDE=2	STRIDE=3
64	0.85	0.27	0.14
128	3.77	0.85	0.45
256	25.87	3.85	1.77
512	180.56	23.46	7.79

当 stride 固定不变时，随着矩阵规模的增大， im2col 卷积运算的时间变长。

而当矩阵规模固定不变，stride 增大时，im2col 卷积运算时间显著缩短。

此外，很明显可以看出，im2col方法的运行时间比CUDA直接计算卷积慢得多，原因可能是 im2col 在调整排列的过程中会跨区域的读取数据，导致频繁的刷新缓存，如果卷积核数目少，那么 im2col 操作之后的新排列就只会用一次，这样就达不到加快卷积运算的效果。

5. 使用cuDNN进行卷积

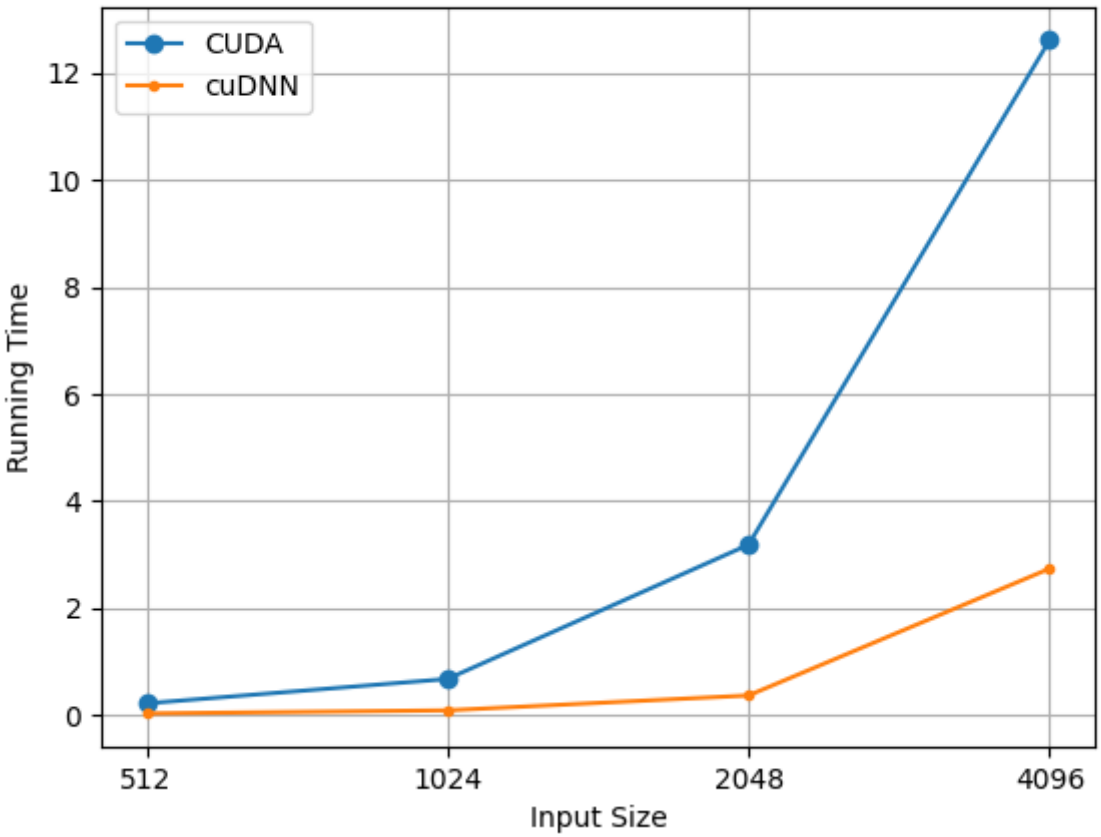
时间对比

在不同规模的输入和不同步长的情况下分别做测试，得到的结果汇总如下表：

INPUT	STRIDE=1	STRIDE=2	STRIDE=3
512	0.027	0.013	0.008
1024	0.086	0.032	0.021
2048	0.364	0.128	0.095
4096	2.738	0.792	0.384

输入矩阵规模相同时，随着 stride 增大，运算时间缩短。同样 stride 时，运算时间会随输入规模的增大而增大。当然，由于cuDNN计算卷积的时间极短，甚至不到1ms，因此此结论不明显。

显然， cuDNN的运算速度比前面两种卷积计算方法快很多，下面展示它们之间的对比，由于im2col远比另外两种方法慢，图中不展示im2col的数据。



可以看到cuDNN效率最优。

改进方向

还可能存在以下几种改进方法：

- 最大化内存吞吐量：

1. 考虑局部性原理和对齐问题，降低访存所消耗的时间。
2. 尽量减少 host 和 device 之间的内存传输；

- 优化指令流：

1. 尽量减少条件分支；
2. 使用 CUDA 算数指令中的快速指令

- 合理化数据分配：

1. 调整grid和block大小
2. 减少一些中间变量

四、实验感想

这次实验首次使用了CUDA进行编程，我了解CUDA编程与CUBLAS函数库的使用，发现计算的效率非常高。

在使用各类方法实现卷积的过程中，对于深度学习的卷积神经网络有了进一步的认识和了解，对于卷积算法的底层实现也更加熟悉。实验过程中遇到了很多问题，大多通过网上查阅资料的方法得到解决，比如滑窗法实现卷积时如何正确地进行填充。还有cuDNN环境的配置安装也遇到了不少问题，后来通过跟同学交流了解了找到链接库正确位置的方法。

总体而言，通过这次实验，学习到了大量新知识，对于CUDA编程也还需要不断学习和巩固。

五、参考资料

1.https://zhuanlan.zhihu.com/p/34587739

2.https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm

3.https://zhuanlan.zhihu.com/p/386052987

4.https://blog.csdn.net/panda1234lee/article/details/83154504

5.https://blog.csdn.net/ice__snow/article/details/79699388

6.https://blog.csdn.net/ChuiGeDaQiQiu/article/details/81265471