中山大学计算机院本科生实验报告

（2023 学年秋季学期）

课程名称：高性能计算程序设计　　**批改人：**

实验	**通用矩阵乘法**	专业（方向）	**信息与计算科学**
学号	21311359	姓名	**何凯迪**
Email	hekd@mail2.sysu.edu.cn	完成日期	2023/12/20

1. 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）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，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.

（4）使用 im2col 方法结合任务 1 实现的 GEMM（通用矩阵乘法）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。

我将任务一二归位了一类，任务三四五归位一类，即在任务一二中的大框架（输入输出数据处理）是类似（或一致）的，在任务三四五中的大框架（输入输出数据处理）是类似（或一致）的。所以只在任务一、三的代码部分详细解释了代码，而在任务二、四五中无特殊情况只展示解释了核心代码，不对其他部分冗余粘贴代码。

2. 实验过程 & 核心代码 & 实验结果

首先编写代码输出一下设备的配置：

```
#include <iostream>
#include <cuda_runtime.h>

int main() {
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    if (deviceCount == 0) {
        std::cerr << "No CUDA devices found." << std::endl;
        return 1;
    }

    std::cout << "Number of CUDA devices: " << deviceCount << std::endl;

    for (int dev = 0; dev < deviceCount; ++dev) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

        std::cout << "\nDevice " << dev << ": " << deviceProp.name << std::endl;
        std::cout << "  Compute capability: " << deviceProp.major << "." << deviceProp.minor << std::endl;
        std::cout << "  Total global memory: " << deviceProp.totalGlobalMem / (1024 * 1024) << " MB" << std::endl;
        std::cout << "  Maximum threads per block: " << deviceProp.maxThreadsPerBlock << std::endl;
        std::cout << "  Maximum dimensions of block: " << deviceProp.maxThreadsDim[0] << " x "
                  << deviceProp.maxThreadsDim[1] << " x " << deviceProp.maxThreadsDim[2] << std::endl;
        std::cout << "  Maximum dimensions of grid: " << deviceProp.maxGridSize[0] << " x "
                  << deviceProp.maxGridSize[1] << " x " << deviceProp.maxGridSize[2] << std::endl;
    }

    return 0;
}
```

得到结果如下：

```
jovyan@jupyter-21311359:~$ ./para
Number of CUDA devices: 4

Device 0: Tesla V100-SXM2-32GB
  Compute capability: 7.0
  Total global memory: 32510 MB
  Maximum threads per block: 1024
  Maximum dimensions of block: 1024 x 1024 x 64
  Maximum dimensions of grid: 2147483647 x 65535 x 65535

Device 1: Tesla V100-SXM2-32GB
  Compute capability: 7.0
  Total global memory: 32510 MB
  Maximum threads per block: 1024
  Maximum dimensions of block: 1024 x 1024 x 64
  Maximum dimensions of grid: 2147483647 x 65535 x 65535

Device 2: Tesla V100-SXM2-32GB
  Compute capability: 7.0
  Total global memory: 32510 MB
  Maximum threads per block: 1024
  Maximum dimensions of block: 1024 x 1024 x 64
  Maximum dimensions of grid: 2147483647 x 65535 x 65535

Device 3: Tesla V100-SXM2-32GB
  Compute capability: 7.0
  Total global memory: 32510 MB
  Maximum threads per block: 1024
  Maximum dimensions of block: 1024 x 1024 x 64
  Maximum dimensions of grid: 2147483647 x 65535 x 65535
```

显然能满足任务要求。

（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。

首先在自己惯用的 IDE 写好代码，保存后将.cpp 改为.cu，上传至 JupyterLab 进行调试。

使用如下代码编译：

```
jovyan@jupyter-21311359:~$ nvcc -o test_cuda_executable test.cu
```

核心代码及解释如下：

```
__global__ void matrixMultiplication(int *a, int *b, int *c, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < K) {
        int sum = 0;
        for (int i = 0; i < N; ++i) {
            sum += a[row * N + i] * b[i * K + col];
        }
        c[row * K + col] = sum;
    }
}
```

这是用于矩阵乘法的 CUDA 核函数，每个线程负责计算结果矩阵中的一个元素。blockIdx 和 threadIdx 是 CUDA 提供的内置变量，用于获取当前线程的块索引和线程索引。

```
int block_size_x = atoi(argv[1]);
int block_size_y = atoi(argv[2]);

// Set matrix dimensions
int M = atoi(argv[3]);
int N = atoi(argv[4]);
int K = atoi(argv[5]);
```

通过命令行参数传递线程块大小和矩阵维度。

```
// Initialize matrices with random values
srand(static_cast<unsigned>(2333));
for (int i = 0; i < M * N; ++i) {
    h_A[i] = rand() % 10;
}

for (int i = 0; i < N * K; ++i) {
    h_B[i] = rand() % 10;
}
```

初始化输入矩阵，元素为 0 到 9 之间的随机整数。

```
// Allocate memory for matrices on device
int *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, M * N * sizeof(int));
cudaMalloc((void**)&d_B, N * K * sizeof(int));
cudaMalloc((void**)&d_C, M * K * sizeof(int));

// Copy matrices from host to device
cudaMemcpy(d_A, h_A, M * N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * K * sizeof(int), cudaMemcpyHostToDevice);
```

在 GPU 上分配内存，并把数据复制到 GPU 上。

```
// Set block size and grid size
dim3 blockSize(block_size_x, block_size_y);
dim3 gridSize((K + blockSize.x - 1) / blockSize.x, (M + blockSize.y - 1) / blockSize.y);
```

设置线程块大小和网格大小。

```
// Launch kernel and measure time
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventRecord(start, 0);

matrixMultiplication<<<gridSize, blockSize>>>(d_A, d_B, d_C, M, N, K);

cudaDeviceSynchronize();

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

调用 GPU 上的核函数执行矩阵乘法，并计算执行时间。

```
// Print matrices and execution time
std::cout << "Matrix A:" << std::endl;
printMatrix(h_A, M, N);

std::cout << "Matrix B:" << std::endl;
printMatrix(h_B, N, K);

std::cout << "Matrix C:" << std::endl;
printMatrix(h_C, M, K);

printf("Elapsed Time: %f ms\n", elapsedTime);
```

把数据复制回主机并打印矩阵和执行时间。

```
// Free allocated memory
delete[] h_A;
delete[] h_B;
delete[] h_C;
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

释放内存。

首先验证其正确性：

CUDA 实现：

$M = N = K = 10$

```
jovyan@jupyter-21311359:~$ ./test_cuda_executable 32 10 10 10
Matrix A:
5 1 4 4 6 0 9 8 8 0
0 8 0 0 3 0 4 2 5 5
6 3 4 3 7 7 2 8 6 5
8 1 6 2 8 2 2 7 2 2
8 4 0 0 4 3 3 8 7 8
4 4 3 8 9 1 7 1 1 3
8 9 5 4 1 5 8 5 2 2
7 2 6 9 3 3 4 6 1 4
6 7 8 2 7 7 5 5 1 6
0 9 5 5 6 8 2 6 3 7
Matrix B:
1 2 9 7 3 4 0 7 2 4
1 9 3 1 3 1 1 8 8 4
4 8 3 1 6 1 9 8 8 4
5 9 8 7 8 1 1 1 0 4
7 4 5 0 5 8 3 8 8 3
2 2 2 8 3 0 9 4 8 9
8 6 0 6 5 9 7 6 2 7
2 9 3 9 1 9 9 7 9 7
0 4 1 2 2 4 2 3 8 3
3 8 1 3 4 6 4 3 4 8
Matrix C:
172 269 154 210 173 262 210 261 252 217
80 186 55 75 91 136 93 156 174 138
150 288 183 237 181 234 244 286 324 276
139 226 178 178 154 206 188 259 243 195
110 256 149 212 134 243 182 251 270 249
192 257 175 161 204 197 147 229 187 195
154 304 180 241 190 195 219 295 261 265
161 293 203 236 203 187 201 239 212 237
186 326 186 222 215 227 271 334 331 297
161 340 156 206 201 193 251 281 334 287
```

串行实现：

```
jovyan@jupyter-21311359:~$ ./verify 10 10 10
Matrix A:
5 1 4 4 6 0 9 8 8 0
0 8 0 0 3 0 4 2 5 5
6 3 4 3 7 7 2 8 6 5
8 1 6 2 8 2 2 7 2 2
8 4 0 0 4 3 3 8 7 8
4 4 3 8 9 1 7 1 1 3
8 9 5 4 1 5 8 5 2 2
7 2 6 9 3 3 4 6 1 4
6 7 8 2 7 7 5 5 1 6
0 9 5 5 6 8 2 6 3 7
Matrix B:
1 2 9 7 3 4 0 7 2 4
1 9 3 1 3 1 1 8 8 4
4 8 3 1 6 1 9 8 8 4
5 9 8 7 8 1 1 1 0 4
7 4 5 0 5 8 3 8 8 3
2 2 2 8 3 0 9 4 8 9
8 6 0 6 5 9 7 6 2 7
2 9 3 9 1 9 9 7 9 7
0 4 1 2 2 4 2 3 8 3
3 8 1 3 4 6 4 3 4 8
Matrix C:
172 269 154 210 173 262 210 261 252 217
80 186 55 75 91 136 93 156 174 138
150 288 183 237 181 234 244 286 324 276
139 226 178 178 154 206 188 259 243 195
110 256 149 212 134 243 182 251 270 249
192 257 175 161 204 197 147 229 187 195
154 304 180 241 190 195 219 295 261 265
161 293 203 236 203 187 201 239 212 237
186 326 186 222 215 227 271 334 331 297
161 340 156 206 201 193 251 281 334 287
```

CUDA 实现：

$M = N = K = 5$

```
jovyan@jupyter-21311359:~$ ./test_cuda_executable 32 5 5 5
Matrix A:
5 1 4 4 6
0 9 8 8 0
0 8 0 0 3
0 4 2 5 5
6 3 4 3 7
Matrix B:
7 2 8 6 5
8 1 6 2 8
2 2 7 2 2
8 4 0 0 4
3 3 8 7 8
Matrix C:
101 53 122 82 105
152 57 110 34 120
73 17 72 37 88
91 43 78 47 96
119 56 150 99 130
```

串行实现：

```
jovyan@jupyter-21311359:~$ ./verify 5 5 5
Matrix A:
5 1 4 4 6
0 9 8 8 0
0 8 0 0 3
0 4 2 5 5
6 3 4 3 7
Matrix B:
7 2 8 6 5
8 1 6 2 8
2 2 7 2 2
8 4 0 0 4
3 3 8 7 8
Matrix C:
101 53 122 82 105
152 57 110 34 120
73 17 72 37 88
91 43 78 47 96
119 56 150 99 130
```

可见结果一致，故正确性验证成功。

实验结果如下：

T(ms) 规模 Size	512	1024	2048	4096	8192
32	0.370176	2.697536	21.389343	187.922501	1279.208618
64	0.242144	1.671264	14.600768	149.819321	1743.250854
128	0.172864	1.039360	10.919808	90.166176	895.044250
256	0.164160	1.192736	9.612000	78.972893	607.525208
512	0.165664	1.016640	8.202688	72.038429	512.743103

由于我使用的是二维，经实验表明例如将 32 拆为 4*8 和 8*4、1*32 和 32*1 等造成的性能是有差异的，但本次实验中不深入探究。

记录过程类似如下：

```
jovyan@jupyter-21311359:~$ ./test_cuda_executable 8 8 1024 1024 1024
Matrix A:
Matrix B:
Matrix C:
Block size: (8, 8, 1)
Grid size: (128, 128, 1)
Elapsed Time: 1.671264 ms
jovyan@jupyter-21311359:~$ ./test_cuda_executable 8 8 512 512 512
Matrix A:
Matrix B:
Matrix C:
Block size: (8, 8, 1)
Grid size: (64, 64, 1)
Elapsed Time: 0.242144 ms
jovyan@jupyter-21311359:~$ ./test_cuda_executable 8 8 8192 8192 8192
Matrix A:
Matrix B:
Matrix C:
Block size: (8, 8, 1)
Grid size: (1024, 1024, 1)
Elapsed Time: 1744.124878 ms
```

(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。

使用如下代码编译：

```
jovyan@jupyter-21311359:~$ nvcc -o b B.cu -lcublas
```

核心代码及解释如下：（与上一题的相似部分不重复说明）

```
// 初始化CUBLAS
cublasHandle_t handle;
cublasCreate(&handle);
```

先初始化 CUBLS。

使用 CUBLAS 做矩阵相乘前先定义参数：

```
float alpha = 1.0f;
float beta = 0.0f;
```

alpha 和 beta 用来控制矩阵的缩放，cublasSgemm 实际上完成的是矩阵乘加操作 C = alpha * op (A) * op (B) + beta * C 。我在此将它设置为默认值，即不做缩放。

```
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, k, n, &alpha, d_B, m, d_A, k, &beta, d_C, m);
```

直接调用函数执行矩阵乘法操作。

其结构如下：

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

官方文档中对该函数的 API 说明如下：

Param.	Memory	In/out	Meaning
handle		input	handle to the cuBLAS library context.
transa		input	operation op(A) that is non- or (conj.) transpose.
transb		input	operation op(B) that is non- or (conj.) transpose.
m		input	number of rows of matrix op(A) and C .
n		input	number of columns of matrix op(B) and C .
k		input	number of columns of op(A) and rows of op(B).
alpha	host or device	input	<type> scalar used for multiplication.
A	device	input	<type> array of dimensions lda x k with lda>=max(1,m) if transa == CUBLAS_OP_N and lda x m with lda>=max(1,k) otherwise.
lda		input	leading dimension of two-dimensional array used to store the matrix A .
B	device	input	<type> array of dimension ldb x n with ldb>=max(1,k) if transb == CUBLAS_OP_N and ldb x k with ldb>=max(1,n) otherwise.
ldb		input	leading dimension of two-dimensional array used to store matrix B .
beta	host or device	input	<type> scalar used for multiplication. If beta==0 , C does not have to be a valid input.
C	device	in/out	<type> array of dimensions ldc x n with ldc>=max(1,m) .
ldc		input	leading dimension of a two-dimensional array used to store the matrix C .

https://docs.nvidia.com/cuda/cublas/index.html

由于 CUBLAS 中矩阵按行优先存储，所以此处进行相关阐释。

1、C++按行优先存储矩阵 A 和 B

2、CUBLAS 按列优先读取矩阵 A 和 B，此时实际存储的是A^T和B^T

3、在代码中我将 transa 和 transb 都设置为 CUBLAS_OP_N，并用 d_B 左乘 d_A，实际上是用 cublasSgemm 计算B^T左乘A^T得到C^T

4、C^T在 CUBLAS 中按列优先存储

5、C++按行优先读取，即对C^T做转置，得到 C

```
// 销毁CUBLAS句柄
cublasDestroy(handle);
```

最后销毁句柄。

同样地，进行正确性验证：

CUBLAS 实现：

$M = N = K = 10$

```
jovyan@jupyter-21311359:~$ ./b 10 10 10
Matrix A:
5 1 4 4 6 0 9 8 8 0
0 8 0 0 3 0 4 2 5 5
6 3 4 3 7 7 2 8 6 5
8 1 6 2 8 2 2 7 2 2
8 4 0 0 4 3 3 8 7 8
4 4 3 8 9 1 7 1 1 3
8 9 5 4 1 5 8 5 2 2
7 2 6 9 3 3 4 6 1 4
6 7 8 2 7 7 5 5 1 6
0 9 5 5 6 8 2 6 3 7
Matrix B:
1 2 9 7 3 4 0 7 2 4
1 9 3 1 3 1 1 8 8 4
4 8 3 1 6 1 9 8 8 4
5 9 8 7 8 1 1 1 0 4
7 4 5 0 5 8 3 8 8 3
2 2 2 8 3 0 9 4 8 9
8 6 0 6 5 9 7 6 2 7
2 9 3 9 1 9 9 7 9 7
0 4 1 2 2 4 2 3 8 3
3 8 1 3 4 6 4 3 4 8
Matrix C:
172 269 154 210 173 262 210 261 252 217
80 186 55 75 91 136 93 156 174 138
150 288 183 237 181 234 244 286 324 276
139 226 178 178 154 206 188 259 243 195
110 256 149 212 134 243 182 251 270 249
192 257 175 161 204 197 147 229 187 195
154 304 180 241 190 195 219 295 261 265
161 293 203 236 203 187 201 239 212 237
186 326 186 222 215 227 271 334 331 297
161 340 156 206 201 193 251 281 334 287
```

串行实现：

```
jovyan@jupyter-21311359:~$ ./verify 10 10 10
Matrix A:
5 1 4 4 6 0 9 8 8 0
0 8 0 0 3 0 4 2 5 5
6 3 4 3 7 7 2 8 6 5
8 1 6 2 8 2 2 7 2 2
8 4 0 0 4 3 3 8 7 8
4 4 3 8 9 1 7 1 1 3
8 9 5 4 1 5 8 5 2 2
7 2 6 9 3 3 4 6 1 4
6 7 8 2 7 7 5 5 1 6
0 9 5 5 6 8 2 6 3 7
Matrix B:
1 2 9 7 3 4 0 7 2 4
1 9 3 1 3 1 1 8 8 4
4 8 3 1 6 1 9 8 8 4
5 9 8 7 8 1 1 1 0 4
7 4 5 0 5 8 3 8 8 3
2 2 2 8 3 0 9 4 8 9
8 6 0 6 5 9 7 6 2 7
2 9 3 9 1 9 9 7 9 7
0 4 1 2 2 4 2 3 8 3
3 8 1 3 4 6 4 3 4 8
Matrix C:
172 269 154 210 173 262 210 261 252 217
80 186 55 75 91 136 93 156 174 138
150 288 183 237 181 234 244 286 324 276
139 226 178 178 154 206 188 259 243 195
110 256 149 212 134 243 182 251 270 249
192 257 175 161 204 197 147 229 187 195
154 304 180 241 190 195 219 295 261 265
161 293 203 236 203 187 201 239 212 237
186 326 186 222 215 227 271 334 331 297
161 340 156 206 201 193 251 281 334 287
```

CUBLAS 实现：

$M = N = K = 5$

```
jovyan@jupyter-21311359:~$ ./b 5 5 5
Matrix A:
5 1 4 4 6
0 9 8 8 0
0 8 0 0 3
0 4 2 5 5
6 3 4 3 7
Matrix B:
7 2 8 6 5
8 1 6 2 8
2 2 7 2 2
8 4 0 0 4
3 3 8 7 8
Matrix C:
101 53 122 82 105
152 57 110 34 120
73 17 72 37 88
91 43 78 47 96
119 56 150 99 130
```

串行实现：

```
jovyan@jupyter-21311359:~$ ./verify 5 5 5
Matrix A:
5 1 4 4 6
0 9 8 8 0
0 8 0 0 3
0 4 2 5 5
6 3 4 3 7
Matrix B:
7 2 8 6 5
8 1 6 2 8
2 2 7 2 2
8 4 0 0 4
3 3 8 7 8
Matrix C:
101 53 122 82 105
152 57 110 34 120
73 17 72 37 88
91 43 78 47 96
119 56 150 99 130
```

可见结果一致，故正确性验证成功。

实验结果如下：

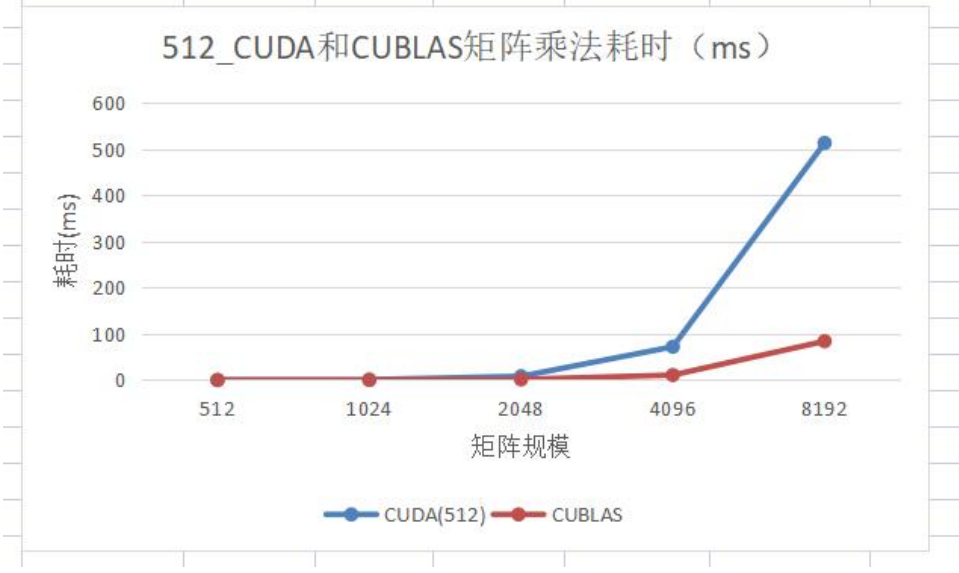
矩阵规模	512	1024	2048	2096	8192
时间（ms）	0.097504	0.200352	1.415552	10.859904	84.060799

实验过程如下：

```
jovyan@jupyter-21311359:~$ nvcc -o b B.cu -lcublas
jovyan@jupyter-21311359:~$ ./b 512 512 512
Elapsed Time: 0.097504 ms
jovyan@jupyter-21311359:~$ ./b 1024 1024 1024
Elapsed Time: 0.200352 ms
jovyan@jupyter-21311359:~$ ./b 2048 2048 2048
Elapsed Time: 1.415552 ms
jovyan@jupyter-21311359:~$ ./b 4096 4096 4096
Elapsed Time: 10.859904 ms
jovyan@jupyter-21311359:~$ ./b 8192 8192 8192
Elapsed Time: 84.060799 ms
```

与任务一性能对比如下：

矩阵规模	512	1024	2048	4096	8192
CUDA(512)	0.165664	1.01664	8.202688	72.038429	512.743103
CUBLAS	0.097504	0.200352	1.415552	10.859904	84.060799



由于 CUBLAS 是高度优化的、专门为 GPU 设计的线性代数库，它充分利用了 NVIDIA GPU 的硬件和架构特性。通用的 CUDA 实现没有经过类似的优化，导致性能差异。

可能通过循环展开、向量化、异步执行等步骤优化，我用 CUDA 实现的代码中同步等待时间可能占用了不少时间开销。

（3）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，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.

核心代码及解释如下：

```
__global__ void convolution(float *input, float *kernel, float *output, int input_size, int kernel_size, int stride, int padding, int channels, int
output_size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Calculate row and column indices
    int row = idx / output_size;
    int col = idx % output_size;

    // Iterate over channels
    for (int channel = 0; channel < channels; ++channel) {
        float sum = 0.0f;

        // Iterate over the kernel
        for (int i = 0; i < kernel_size; ++i) {
            for (int j = 0; j < kernel_size; ++j) {
                int input_row = row * stride + i - padding;
                int input_col = col * stride + j - padding;

                // Check if the input indices are within bounds
                if (input_row >= 0 && input_row < input_size && input_col >= 0 && input_col < input_size) {
                    int input_index = (input_row * input_size + input_col) * channels + channel;
                    int kernel_index = (i * kernel_size + j) * channels + channel;
                    sum += input[input_index] * kernel[kernel_index];
                }
            }
        }

        // Store the result in the output array
        output[(row * output_size + col) * channels + channel] = sum;
    }
}
```

这是核函数，根据线程和块的索引计算全局唯一的线程索引 idx。再通过 idx 计算出在输出矩阵中的行 row 和列 col 的索引。对每个通道进行迭代。初始化一个用于存储卷积结果的变量 sum。使用嵌套的循环遍历卷积核和输入数据，计算卷积的结果。最后将卷积结果存储在输出数组中。

```
int input_size = atoi(argv[1]);
int kernel_size = atoi(argv[2]);
int stride = atoi(argv[3]);
int channels = atoi(argv[4]);
int padding = atoi(argv[5]);

int output_size = (input_size + 2 * padding - kernel_size) / stride + 1;
```

通过命令行参数传递各卷积参数，并计算输出层大小。

```
// 分配和初始化输入数据和卷积核
float *h_input = new float[input_size * input_size * channels];
float *h_kernel = new float[kernel_size * kernel_size * channels];
float *h_output = new float[output_size * output_size];
```

在主机上分配内存。

```
// 设置种子
std::srand(2333);
// 初始化 h_input 和 h_kernel 为随机值
for (int i = 0; i < input_size * input_size * channels; ++i) {
    h_input[i] = static_cast<float>(rand()) / RAND_MAX;
}

for (int i = 0; i < kernel_size * kernel_size * channels; ++i) {
    h_kernel[i] = static_cast<float>(rand()) / RAND_MAX;
}
```

将 Input 和 Kernel 初始化为随机值。

```
// 输出初始化的矩阵
std::cout << "Input Matrix:" << std::endl;
for (int i = 0; i < input_size; ++i) {
    for (int j = 0; j < input_size; ++j) {
        std::cout << h_input[(i * input_size + j) * channels] << " ";
    }
    std::cout << std::endl;
}

// 输出卷积核矩阵
std::cout << "Kernel Matrix:" << std::endl;
for (int i = 0; i < kernel_size; ++i) {
    for (int j = 0; j < kernel_size; ++j) {
        std::cout << h_kernel[(i * kernel_size + j) * channels] << " ";
    }
    std::cout << std::endl;
}
```

输出 Input 和 Kernel，由于三个通道的 Input 都被设置为统一的，因此只输出了一次。

```
// 在GPU上分配内存
float *d_input, *d_kernel, *d_output;
cudaMalloc((void**)&d_input, input_size * input_size * channels * sizeof(float));
cudaMalloc((void**)&d_kernel, kernel_size * kernel_size * channels * sizeof(float));
cudaMalloc((void**)&d_output, output_size * output_size * sizeof(float));

// 将输入数据和卷积核从主机内存复制到GPU内存
cudaMemcpy(d_input, h_input, input_size * input_size * channels * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_kernel, h_kernel, kernel_size * kernel_size * channels * sizeof(float), cudaMemcpyHostToDevice);
```

在 GPU 分配内存，并把数据复制到 GPU。

```
// 调用CUDA内核函数执行卷积
dim3 blockDim(1024);  // 一维线程块
dim3 gridDim((output_size * output_size + blockDim.x - 1) / blockDim.x);

// 记录开始时间
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

convolution<<<gridDim, blockDim>>>(d_input, d_kernel, d_output, input_size, kernel_size, stride, padding, channels, output_size);
cudaDeviceSynchronize();

// 记录结束时间
cudaEventRecord(stop);
cudaEventSynchronize(stop);
```

调用内核函数并记录时间。

```
// 将结果从GPU内存复制回主机内存
cudaMemcpy(h_output, d_output, output_size * output_size * sizeof(float), cudaMemcpyDeviceToHost);

// 输出卷积后的矩阵
std::cout << "Convolution Output Matrix:" << std::endl;
for (int i = 0; i < output_size; ++i) {
    for (int j = 0; j < output_size; ++j) {
        std::cout << h_output[i * output_size + j] << " ";
    }
    std::cout << std::endl;
}

// 计算时间差并输出
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
std::cout << "Convolution time: " << milliseconds << " ms" << std::endl;
```

从 GPU 把数据复制回主机，并输出卷积结果和执行时间。

```
// 释放GPU内存
cudaFree(d_input);
cudaFree(d_kernel);
cudaFree(d_output);

// 释放主机内存
delete[] h_input;
delete[] h_kernel;
delete[] h_output;
```

释放内存。

结果验证：（为方便起见，将通道数设置为 1，元素为整型进行验证）

Input：4*4，Kernel：3*3，padding=1，stride=3

CUDA：

```
Input Matrix:
7 8 9 10
13 14 15 16
19 20 21 22
25 26 27 28
Kernel Matrix:
1 2 3
4 5 6
7 8 9
Convolution Output Matrix:
313 319
379 313
```

pytorch：

```
PS D:\python文件> & D:/PY/python.exe d:/python文件/test.py
Input:
[[ 7.  8.  9. 10.]
 [13. 14. 15. 16.]
 [19. 20. 21. 22.]
 [25. 26. 27. 28.]]
Kernel:
[[[[1. 2. 3.]
   [4. 5. 6.]
   [7. 8. 9.]]]]
卷积结果:
[[313. 319.]
 [379. 313.]]
```

Input：4*4，Kernel：3*3，padding=1，stride=1

CUDA：

```
Input Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
4 6 6 4
6 9 9 6
6 9 9 6
4 6 6 4
```

pytorch：

```
Input:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Kernel:
[[[[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]]]
卷积结果:
[[4. 6. 6. 4.]
 [6. 9. 9. 6.]
 [6. 9. 9. 6.]
 [4. 6. 6. 4.]]
```

Input：5*5，Kernel：3*3，padding=0，stride=1

CUDA：

```
Input Matrix:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
9 9 9
9 9 9
9 9 9
```

pytorch：

```
Input:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
Kernel:
[[[[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]]]
卷积结果:
[[9. 9. 9.]
 [9. 9. 9.]
 [9. 9. 9.]]
```

可见结果均一致，故正确性验证成功。

实验结果：（时间单位为 ms）

步幅 \ 规模	256	512	1024	2048	4096
1	0.030880	2.40278	3.78432	3.86436	3.87491
2	0.023296	2.36525	3.42523	3.59374	3.89847
3	0.193088	2.27030	3.59487	3.73893	3.72923

命令行参数分别为：Input 规模，Kernel 大小，步幅 stride，通道 channel，填充 padding，线程块大小 block_size

部分输出结果如下：由于篇幅问题不一一展示

```
jovyan@jupyter-21311359:~$ ./c 256 3 1 3 0 16
Convolution time: 0.03088 ms
jovyan@jupyter-21311359:~$ ./c 256 3 2 3 1 64
Convolution time: 0.023296 ms
jovyan@jupyter-21311359:~$ ./c 256 3 3 3 1 64
Convolution time: 0.193088 ms
jovyan@jupyter-21311359:~$ ./c 512 3 1 3 0 128
Convolution time: 2.40278 ms
jovyan@jupyter-21311359:~$ ./c 512 3 2 3 1 128
Convolution time: 2.36525 ms
jovyan@jupyter-21311359:~$ ./c 512 3 3 3 1 128
Convolution time: 2.2703 ms
```

对于 stride 为 1 的情况，设置 padding=0

对于 stride=2/3 的情况，通常是设置为（kernel_size-1）/2,此处为 1

对于不同规模的问题，不同线程块大小也会对性能有影响，所以最终得到的实验数据其实是对线程块大小不断调参后得到的较小时间值。

实验过程中我还在网上看到另一种计算 padding 的方法，并不是往外扩 padding 圈，而是将左上方和右下方分别扩充 padding 层。由于这样的操作相对麻烦，因此本次实验没有采用。链接如下：

https://blog.csdn.net/rain6789/article/details/78754516

（4）使用 im2col 方法结合任务 1 实现的 GEMM（通用矩阵乘法）实现卷积操作。输入从 256 增加至 4096 或者输入从 32 增加至 512。

大部分和上一任务一致，故不赘述。

核心代码：

```
__device__ void matrixMultiplication(float *a, float *b, float *c, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    //printf("Thread (%d, %d) - Row: %d, Col: %d\n", threadIdx.x, threadIdx.y, row, col);
    if (row < M && col < K) {
        int sum = 0;
        for (int i = 0; i < N; ++i) {
            sum += a[row * N + i] * b[i * K + col];
        }
        c[row * K + col] = sum;
    }
}
```

矩阵乘法，使用了 CUDA 的线程和块的索引来确定当前线程要计算的元素的位置。对于每个线程，它计算输出矩阵中的一个元素，通过遍历输入矩阵的对应行和列，执行累加操作。

```
__device__ void im2col( float* input, int channels, int height, int width,
                        int kernel_size, int stride, int padding,
                        int output_height, int output_width, float* col_matrix) {
    // Iterate over output matrix
    for (int ky = 0; ky < kernel_size; ++ky) {
        for (int kx = 0; kx < kernel_size; ++kx) {
            for (int c = 0; c < channels; ++c) {
                for (int oy = 0; oy < output_height; ++oy) {
                    for (int ox = 0; ox < output_width; ++ox) {
                        // Calculate input indices
                        int iy = oy * stride - padding + ky;
                        int ix = ox * stride - padding + kx;

                        // Check if indices are within bounds
                        bool valid = (iy >= 0 && iy < height && ix >= 0 && ix < width);

                        // Calculate linear indices
                        int input_index = c * height * width + iy * width + ix;
                        int col_index = (ky * kernel_size + kx) * channels + c;

                        // Assign value to col_matrix, use 0 if indices are out of bounds
                        col_matrix[oy * output_width + ox + col_index * output_height * output_width] =
                            valid ? input[input_index] : 0.0f;
                    }
                }
            }
        }
    }
}
```

im2col，将输入矩阵的小块转换成一个列矩阵。函数使用嵌套循环迭代卷积核的大小、通道数和输出矩阵的尺寸，将输入矩阵的每个小块按照 im2col 规则排列到一个列矩阵中。

```
__global__ void convolutionWithIm2Col( float* input,  float* kernel, float* output,
                                       int input_channels, int input_height, int input_width,
                                       int kernel_size, int stride, int padding,
                                       int output_height, int output_width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < output_height && col < output_width) {
        // Allocate space for im2col result
        float* col_matrix = new float[kernel_size * kernel_size * input_channels];

        // Use im2col function to convert input matrix to column matrix
        im2col(input, input_channels, input_height, input_width,
               kernel_size, stride, padding, output_height, output_width, col_matrix);

        // Call matrix multiplication kernel for convolution
        matrixMultiplication(col_matrix, kernel, &output[row * output_width + col],
                             kernel_size * kernel_size * input_channels, 1, output_height * output_width);

        // Free memory
        delete[] col_matrix;
    }
}
```

这个函数调用了前两个函数，实现卷积。

它首先分配了存储 im2col 结果的内存，然后调用 im2col 函数将输入矩阵转换为列矩阵，最后调用 matrixMultiplication 函数进行矩阵乘法计算。所以在主函数中调用该核函数即可。

同样做结果验证：

Input：4*4，Kernel：3*3，padding=1，stride=3

CUDA：

```
Input Matrix:
7 8 9 10
13 14 15 16
19 20 21 22
25 26 27 28
Kernel Matrix:
1 2 3
4 5 6
7 8 9
Convolution Output Matrix:
313 319
379 313
```

pytorch：

```
PS D:\python文件> & D:/PY/python.exe d:/python文件/test.py
Input:
[[ 7.  8.  9. 10.]
 [13. 14. 15. 16.]
 [19. 20. 21. 22.]
 [25. 26. 27. 28.]]
Kernel:
[[[[1. 2. 3.]
   [4. 5. 6.]
   [7. 8. 9.]]]]
卷积结果:
[[313. 319.]
 [379. 313.]]
```

Input：4*4，Kernel：3*3，padding=1，stride=1

CUDA：

```
Input Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
4 6 6 4
6 9 9 6
6 9 9 6
4 6 6 4
```

pytorch：

```
Input:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
Kernel:
[[[[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]]]
卷积结果:
[[4. 6. 6. 4.]
 [6. 9. 9. 6.]
 [6. 9. 9. 6.]
 [4. 6. 6. 4.]]
```

Input：5*5，Kernel：3*3，padding=0，stride=1

CUDA：

```
Input Matrix:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
9 9 9
9 9 9
9 9 9
```

pytorch：

```
Input:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
Kernel:
[[[[1. 1. 1.]
   [1. 1. 1.]
   [1. 1. 1.]]]]
卷积结果:
[[9. 9. 9.]
 [9. 9. 9.]
 [9. 9. 9.]]
```

可见与上一任务得到的结果一致，故正确性验证成功。

实验结果：（时间单位为 ms）

步幅 \ 规模	256	512	1024	2048	4096
1	0.013600	0.018560	1.830453	2.673698	3.236704
2	0.011392	0.015776	1.516862	2.827235	3.029104
3	0.012736	0.017440	1.491236	2.648562	2.816160

命令行参数分别为：Input 规模，Kernel 大小，步幅 stride，通道 channel，填充 padding，线程块大小 block_size

部分输出结果如下：由于篇幅问题不一一展示

```
jovyan@jupyter-21311359:~$ ./d 256 3 1 1 1 1024
Convolution time: 0.0136 ms
jovyan@jupyter-21311359:~$ ./d 256 3 2 1 1 1024
Convolution time: 0.011392 ms
jovyan@jupyter-21311359:~$ ./d 256 3 3 1 1 1024
Convolution time: 0.012736 ms
jovyan@jupyter-21311359:~$ ./d 512 3 1 1 1 1024
Convolution time: 0.01856 ms
jovyan@jupyter-21311359:~$ ./d 512 3 2 1 1 1024
Convolution time: 0.015776 ms
jovyan@jupyter-21311359:~$ ./d 512 3 3 1 1 1024
Convolution time: 0.01744 ms
```

为方便起见，padding 统一设置为（kernel_size-1）/2,此处为 1

同样的，对于不同规模的问题，不同线程块大小也会对性能有影响，但本次任务中我将 block_size 固定为 1024。

不难看出，本次任务得到的实验结果与上一次相比得到了很大的改进。

参考资料：

https://blog.csdn.net/m0_45388819/article/details/120757424?ops_request_misc=%257B%2522request%255Fid%2522%253A%252217030530371680018653316 0%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=170305303716800186533160&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-120757424-null-null.142^v96^pc_search_result_base5&utm_term=im2col&spm=1018.2226.3001.4187

（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。

由于本次任务做得更加清晰，故将整体代码详细阐释。

cuDNN 中提供的计算卷积的函数为 cudnnConvolutionForward()

```
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 );
```

cudnn: cuDNN 库的句柄，它是对 cuDNN 库的初始化和配置。

&alpha: 输入数据与卷积核的缩放因子。

input_descriptor: 输入数据的描述符，包含有关输入张量的信息

d_input: 指向存储输入数据的设备（GPU）内存的指针。

kernel_descriptor: 卷积核的描述符，包含有关卷积核的信息。

d_kernel: 指向存储卷积核数据的设备（GPU）内存的指针。

convolution_descriptor: 卷积操作的描述符，包含有关卷积的配置信息。

convolution_algorithm: 卷积算法的选择，指定了卷积操作的实现方式。这是一个输出参数，函数执行后会将选择的算法存储在这个变量中。

d_workspace: 用于存储卷积操作中的中间计算结果的工作空间的设备（GPU）内存指针。

workspace_bytes: 工作空间的大小（以字节为单位）。这是一个输入参数，指定了提供的工作空间的大小。

&beta: 输出数据与原输出数据的缩放因子。

output_descriptor: 输出数据的描述符，包含有关输出张量的信息。

d_output: 指向存储输出数据的设备（GPU）内存的指针。

代码部分：

```
#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);                                  \
    }                                                           \
  }
```

cudnnCreate，像所有其他 cuDNN 例程一样，返回一个 cudnnStatus_t 类型的错误代码。因此可以定义一个宏来检查这个状态对象是否包含任何错误条件，并在程序发生问题时中止执行。在使用 cuDNN 函数时，可以用这个宏包裹函数调用，以提高代码的可读性和错误处理机制。

```
int input_height;
std::cout << "Input Height: ";
std::cin >> input_height;

int input_width;
std::cout << "Input Width: ";
std::cin >> input_width;

int kernel_height = 3;

int kernel_width = 3;

int vertical_stride;
std::cout << "Vertical Stride: ";
std::cin >> vertical_stride;

int horizontal_stride;
std::cout << "Horizontal Stride: ";
std::cin >> horizontal_stride;

int channels;
std::cout << "Channels: ";
std::cin >> channels;

int padding_height = (kernel_height - 1) / 2;
int padding_width = (kernel_width - 1) / 2;

int output_height = (input_height - kernel_height + 2 * padding_height) / vertical_stride + 1;
int output_width = (input_width - kernel_width + 2 * padding_width) / horizontal_stride + 1;
```

进行一些输入和变量计算与赋值。本次实验中卷积核大小固定为 3×3；padding 的值由卷积核大小计算所得，可以防止信息损失以及保持图像边缘信息；计算得出输出矩阵的大小。

```
    float *h_input = new float[input_height * input_width * channels];
    float *h_kernel = new float[kernel_height * kernel_width * channels];


    std::srand(2333);
    for (int i = 0; i < input_height * input_width * channels; ++i) {
        h_input[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    for (int i = 0; i < kernel_height * kernel_width * channels; ++i) {
        h_kernel[i] = static_cast<float>(rand()) / RAND_MAX;
    }
```

给输入矩阵和卷积核分配内存并随机初始化.

```
if (isprint == 1) {
    std::cout << "Input Matrix:" << std::endl;
    for (int i = 0; i < input_height; ++i) {
        for (int j = 0; j < input_width; ++j) {
            std::cout << h_input[(i * input_width + j) * channels] << " ";
        }
    std::cout << std::endl;
    }

    std::cout << "Kernel Matrix:" << std::endl;
    for (int i = 0; i < kernel_height; ++i) {
        for (int j = 0; j < kernel_width; ++j) {
            std::cout << h_kernel[(i * kernel_height + j)] << " ";
        }
    std::cout << std::endl;
    }
}
if (isprint == 1) {
    std::cout << "Convolution Output Matrix:" << std::endl;
    for (int i = 0; i < output_height; ++i) {
        for (int j = 0; j < output_width; ++j) {
            std::cout << h_output[i * output_width + j] << " ";
        }
    std::cout << std::endl;
    }
}
```

选择是否打印输入矩阵、卷积核和卷积结果矩阵。

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

创建 cuDNN 句柄。

```
cudnnTensorDescriptor_t input_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor,
                                      /*format=*/CUDNN_TENSOR_NHWC,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*batch_size=*/1,
                                      /*channels=*/channels,
                                      /*image_height=*/input_height,
                                      /*image_width=*/input_width));
```

创建和设置一个输入张量描述符 cudnnTensorDescriptor_t，包含了有关输入张量的信息，如数据格式、数据类型、批处理大小以及图像的空间维度。

```
cudnnTensorDescriptor_t output_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&output_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor,
                                      /*format=*/CUDNN_TENSOR_NHWC,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*batch_size=*/1,
                                      /*channels=*/1,
                                      /*image_height=*/output_height,
                                      /*image_width=*/output_width));
```

创建和设置输出张量描述符 cudnnTensorDescriptor_t，包含了有关输出张量的信息，如数据格式、数据类型、批处理大小以及图像的空间维度。特别地，我将输出张量设置为单通道（channels=1），因此最后得到的结果应该是输入矩阵和卷积核各通道卷积后相加的结果。

```
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*format=*/CUDNN_TENSOR_NCHW,
                                      /*out_channels=*/1,
                                      /*in_channels=*/channels,
                                      /*kernel_height=*/kernel_height,
                                      /*kernel_width=*/kernel_width));
```

创建和设置卷积核的描述符 cudnnFilterDescriptor_t，包含了有关卷积核的信息，如数据类型、数据格式、输出通道数、输入通道数以及卷积核的大小。

```
cudnnConvolutionDescriptor_t convolution_descriptor;
checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution_descriptor));
checkCUDNN(cudnnSetConvolution2dDescriptor(convolution_descriptor,
                                           /*pad_height=*/padding_height,
                                           /*pad_width=*/padding_width,
                                           /*vertical_stride=*/vertical_stride,
                                           /*horizontal_stride=*/horizontal_stride,
                                           /*dilation_height=*/1,
                                           /*dilation_width=*/1,
                                           /*mode=*/CUDNN_CROSS_CORRELATION,
                                           /*computeType=*/CUDNN_DATA_FLOAT));
```

创建和设置卷积操作的描述符 cudnnConvolutionDescriptor_t，包含了有关卷积操作的信息，如填充大小、步幅、缩放大小（此处设置为 1，不进行缩放）、卷积模式以及计算元素的类型。

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

此处传递了之前定义的描述符，以及另外两个非常重要的参数：首先是我们希望 cuDNN 用于卷积的算法类型的偏好，其次是卷积可用内存的上限。对于后者，如果没有限制，我们可以将其设置为零。对于前者，我们可以选择几个选项之一。

在上面的代码中，我指定了 CUDNN_CONVOLUTION_FWD_PREFER_FASTEST，这告诉 cuDNN 使用最快的可用算法。

在调用 cudnnGetConvolutionForwardAlgorithm 返回后，convolution_algorithm 将包含 cuDNN 决定使用的实际算法，例如：

CUDNN_CONVOLUTION_FWD_ALGO_GEMM，将卷积建模为显式矩阵乘法，

CUDNN_CONVOLUTION_FWD_ALGO_FFT，使用快速傅立叶变换进行卷积，

CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD，使用 Winograd 算法执行卷积。

```
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn,
                                                   input_descriptor,
                                                   kernel_descriptor,
                                                   convolution_descriptor,
                                                   output_descriptor,
                                                   convolution_algorithm,
                                                   &workspace_bytes));
```

用于获取卷积前向操作所需的工作空间大小。

```
void* d_workspace{nullptr};
cudaMalloc(&d_workspace, workspace_bytes);
```

在 GPU 上为卷积操作分配工作空间。

```
float* d_input{nullptr};
cudaMalloc(&d_input, image_bytes);
cudaMemcpy(d_input, h_input, image_bytes, cudaMemcpyHostToDevice);

float* d_output{nullptr};
cudaMalloc(&d_output, output_height * output_width * sizeof(float));
cudaMemset(d_output, 0, output_height * output_width * sizeof(float) );


float* d_kernel{nullptr};
cudaMalloc(&d_kernel, channels * kernel_height * kernel_width * sizeof(float));
cudaMemcpy(d_kernel, h_kernel, channels * kernel_height * kernel_width * sizeof(float), cudaMemcpyHostToDevice);
```

在 GPU 上为输入矩阵、卷积核以及输出矩阵分配空间，并将输入矩阵和卷积核的数据从 CPU 复制过去。

```
const float alpha = 1, beta = 0;
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));

cudaEventRecord(stop);
cudaEventSynchronize(stop);
```

执行卷积操作并计算执行时间。

alpha 和 beta 声明了两个常量，用于权衡输入和输出数据的缩放，这里分别设置为 1 和 0，即不进行缩放。其它参数说明在本任务报告开头已给出，不重复说明。

```
float* h_output = new float[output_height * output_width * sizeof(float)];
cudaMemcpy(h_output, d_output, sizeof(float) * output_height * output_width, cudaMemcpyDeviceToHost);
```

在CPU上给输出矩阵分配内存空间,并将卷积结果从GPU复制到CPU上。

```
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
std::cout << "Convolution time: " << milliseconds << " ms" << std::endl;

delete[] h_output;
cudaFree(d_kernel);
cudaFree(d_input);
cudaFree(d_output);
cudaFree(d_workspace);

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

cudnnDestroy(cudnn);

return 0;
```

输出卷积操作执行时间、释放内存以及销毁描述符和句柄。

结果验证：

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 5
Input Width: 5
Vertical Stride: 2
Horizontal Stride: 2
Channels: 3
print?(1 or 0)1
Input Matrix:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
12 18 12
18 27 18
12 18 12
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 6
Input Width: 3
Vertical Stride: 1
Horizontal Stride: 2
Channels: 3
print?(1 or 0)1
Input Matrix:
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
12 12
18 18
18 18
18 18
18 18
12 12
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 5
Input Width: 5
Vertical Stride: 1
Horizontal Stride: 1
Channels: 3
print?(1 or 0)1
Input Matrix:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 4
Input Width: 5
Vertical Stride: 1
Horizontal Stride: 1
Channels: 3
print?(1 or 0)1
Input Matrix:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Kernel Matrix:
1 1 1
1 1 1
1 1 1
Convolution Output Matrix:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
```

由一些简单的卷积不难看出结果正确。（本来是想将随机数种子设置为一样的，然后与之前编写的代码进行比较从而验证正确性，但是不知道为什么种子一样但是其中部分元素还是不同，所以只好这样用简单数据验算正确性了。本来用 pytorch 编写了一个验证代码，但是手动输入相同的数据太麻烦了，就没用上。）

实验结果：（时间单位为 ms）

步幅 \ 规模	256	512	1024	2048	4096
1	0.07872	0.12736	0.347264	1.20362	4.60496
2	0.06656	0.079808	0.139904	0.376096	1.20435
3	0.06368	0.071776	0.115008	0.20144	0.602272

部分结果如下：篇幅问题不一一展示

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 256
Input Width: 256
Vertical Stride: 1
Horizontal Stride: 1
Channels: 3
print?(1 or 0)0
Convolution time: 0.07872 ms
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 256
Input Width: 256
Vertical Stride: 2
Horizontal Stride: 2
Channels: 3
print?(1 or 0)0
Convolution time: 0.06656 ms
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 256
Input Width: 256
Vertical Stride: 3
Horizontal Stride: 3
Channels: 3
print?(1 or 0)0
Convolution time: 0.06368 ms
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 4096
Input Width: 4096
Vertical Stride: 1
Horizontal Stride: 1
Channels: 3
print?(1 or 0)0
Convolution time: 4.60496 ms
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 4096
Input Width: 4096
Vertical Stride: 2
Horizontal Stride: 2
Channels: 3
print?(1 or 0)0
Convolution time: 1.20435 ms
```

```
jovyan@jupyter-21311359:~$ ./e
Input Height: 4096
Input Width: 4096
Vertical Stride: 3
Horizontal Stride: 3
Channels: 3
print?(1 or 0)0
Convolution time: 0.602272 ms
```

与任务三和任务四对比，显然用 cuDNN 的性能会好很多

如下可能的优化方法:

对于直接卷积，一些轻量级模型可以考虑使用深度可分离卷积，它可以减少参数数量，提高计算效率。对于一些冗余的卷积核，可以考虑使用模型剪枝技术来减少参数数量，从而提高性能。

对于 im2col，可以考虑一次性将多个通道转换为列矩阵，以减少数据移动和提高并行性。或者将输入图像的局部区域存储在缓存中，以减少内存访问次数。这可以通过在循环中适当地调整迭代顺序来实现。尝试将 im2col 操作与后续的卷积操作融合在一起，以减少中间步骤和数据传输。

3. 实验感想

//可以写写过程中遇到的问题，你是怎么解决的。以及可以写你对此次实验的一些理解……

· 第一题，线程块大小设置为 32 时运行正常，而设置为 33、64、128 等的时候矩阵 C 全为 0，经长时间测试发现后者压根没调用核函数。我还直接在程序里获取了允许的块大小进行校验判定，仍没有报错，也就是说块大小大于 32 是被允许的，但又确实报错了“invalid configuration argument”。再后来，通过和同学交流我发现他只用了三维中的一维，而我用了两维，我突然意识到题目所说的“CUDA Thread Block size”大概是指线程数而不是各个维度的大小，否则假设该值设置为 32，对于只使用一维的所用线程为 32，而对于使用二维甚至三维的所用线程则分别为 32*32 和 32*32*32 了。这也就是为什么我把 block_size 设置为 64 时会出错，因为此时实际设置的线程数为 64*64 已经大于设备的范围（1024）了。我也尝试过用一维，但是结果表明性能其实不如二维。然后在实验过程中我还发现把“CUDA Thread Block size”拆成两个不同乘积的顺序也对性能有影响，例如 32=4*8 和 32=8*4 是不一样的，也许和 CUDA 线程块的索引方式有关，但是深入研究的话内容就太大了，故没有展开篇幅。

第二题，在学习过程中我发现 CUBLAS 中矩阵是列优先的，一开始还没理解到位，后来不断更改参数得到正确结果，再动笔作图演算后才有了深刻理解。

第三题，在本次实验中学习到了很多计算 padding 以匹配 stride 的方法实验过程中发现对于不同规模的问题，不同线程块大小也会对性能有影响，所以最终得到的实验数据其实是对线程块大小不断调参后得到的较小时间值。

第四题，在我一开始学习 im2col 的时候发现自己甚至连矩阵乘法和卷积的区别都没彻底弄明白，再去重新复习对比一遍才有了更深的认识。

第五题，阅读完官方文档后很清晰，无较大问题。还是在验证的时候出了些问题，想用之前直接卷积或 im2col 的代码通过设置相同的随机数来验证，但最后不知道为什么其中还是有部分元素不同。而用 pytorch 编写的卷积算法又由于数据输入不便导致被弃用。最后还是使用把元素全设置为 1 的方法来人工验算正确性，算是一个遗憾吧。