高性能计算程序设计（5） 秋季2024

提交格式说明

按照实验报告模板填写报告，需要提供源代码及代码描述至https://easyhpc.net/course/212。实验报告模板使用PDF格式，命名方式为高性能计算程序设计学号姓名。如果有问题，请发邮件至zhudp3@mail2.sysu.edu.cn、liux276@mail2.sysu.edu.cn询问细节。

任务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。

通用矩阵乘法（GEMM）通常定义为：

$C = AB$

$C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$

输入：M , N, K三个整数（512 ~8192）

问题描述：随机生成M*N和N*K的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C。

输出：A,B,C三个矩阵以及矩阵计算的时间

代码解释:

矩阵A维度是（m,n），矩阵B维度是（n,k）；

《grid, block》将矩阵分割到每一个cuda thread， 然后每一个thread只有其坐标在(0,0)到(m,k)（也就是目标矩阵的大小内）才有效，这样的thread遍历a的行和b的列，对应相乘加入到本地temp,然后赋值到目标矩阵，**由于一一对应，所以不需要对目标矩阵加锁。**

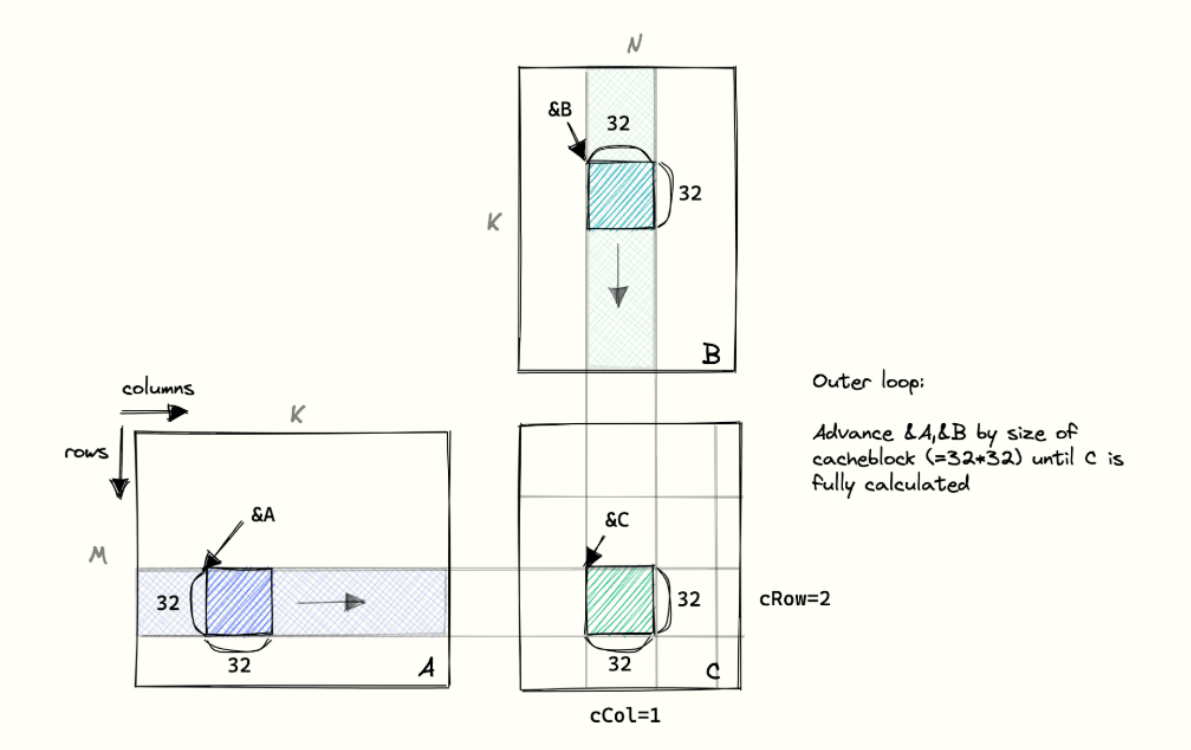
```
/**
 * @brief  implemenntation of the GEMM cuda in global memory
 *
 * @param a  the first matrix in m*n in device memory
 * @param b  the second matrix in n*k in device memory
 * @param c  the result matrix in m*k in device memory
 * @param m  matrix size
 * @param n  matrix size
 * @param k  matrix size
 * @return the result in ptr c
 */
__global__ void gpu_matrix_mult_gm(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;
    float temp = 0;
    if (row < m && col < k) // Ensure bounds are within the matrix dimensions
    {
        for (int i = 0; i < n; i++)
        {
            temp += a[row * n + i] * b[i * k + col];
        }
```

```
            c[row * k + col] = temp;
        }
    }
```

进一步优化，可以考虑分块矩阵，以及共享内存，**分块矩阵乘法在缓存命中率上有提高**，cuda的cache模型是一行128字节,也就是32个float，将矩阵划分为32*32的小块，可以增加缓存读写的使用率，同时**共享内存的读写也比全局内存快很多。**通过访问gou信息，这块4090gpu的每个块最多共享**48KB共享内存**，这些内存是L1级别的读写速度。

分块如图所示，每个BLOCK作为单位来分块，对A横向遍历，对B纵向遍历，多次计算线程对于结果的数值temp(计算是+=运算)，每个线程维护自己的temp，不需要加解锁操作。

图片来自[如何优化 CUDA Matmul 内核以获得类似 cuBLAS 的性能：工作日志]




```
/**
 * @brief  implemenntation of the GEMM cuda in shared memory
 *
 * @param a  the first matrix in m*n in device memory
 * @param b  the second matrix in n*k in device memory
 * @param c  the result matrix in m*k in device memory
 * @param m  matrix size
 * @param n  matrix size
 * @param k  matrix size
 * @return the result in ptr c
 */
#define BLOCK_SIZE = 32
__global__ void gpu_matrix_mult_sm(float *a, float *b, float *c, int m, int n,
int k)
{
    __shared__ float tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float tile_b[BLOCK_SIZE][BLOCK_SIZE];
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    float temp = 0;
```

```
    int block_num = (n + BLOCK_SIZE - 1) / BLOCK_SIZE; // Divide the n dimension
by block size
    for (int i = 0; i < block_num; ++i)
    {
        // Copy the ith block into shared memory
        if (row < m && i * BLOCK_SIZE + threadIdx.x < n)
        {
            int a_index = row * n + i * BLOCK_SIZE + threadIdx.x;
            tile_a[threadIdx.y][threadIdx.x] = a[a_index];
        }
        else
            tile_a[threadIdx.y][threadIdx.x] = 0; // Handle edge case for smaller
matrix

        if (col < k && i * BLOCK_SIZE + threadIdx.y < n)
        {
            int b_index = (i * BLOCK_SIZE + threadIdx.y) * k + col;
            tile_b[threadIdx.y][threadIdx.x] = b[b_index];
        }
        else
            tile_b[threadIdx.y][threadIdx.x] = 0;

        __syncthreads(); // Synchronize threads in the block before computation

        // Compute the contribution for c[row][col]
        for (int j = 0; j < BLOCK_SIZE; j++)
        {
            temp += tile_a[threadIdx.y][j] * tile_b[j][threadIdx.x];
        }
        __syncthreads(); // Ensure all threads finish their computation before
moving on
    }
    if (row < m && col < k)
    {
        c[row * k + col] = temp;
    }
}
```

奇怪的问题：

```
        // Copy the ith block into shared memory
        if (row < m && i * BLOCK_SIZE + threadIdx.x < n)
        {
            int a_index = row * n + i * BLOCK_SIZE + threadIdx.x;
            tile_a[threadIdx.y][threadIdx.x] = a[a_index];
        }
        else
            tile_a[threadIdx.y][threadIdx.x] = 0; // Handle edge case for smaller
matrix

        if (col < k && i * BLOCK_SIZE + threadIdx.y < n)
        {
            int b_index = (i * BLOCK_SIZE + threadIdx.y) * k + col;
            tile_b[threadIdx.x][threadIdx.y] = b[b_index];
        }
```

```
        else
            tile_b[threadIdx.x][threadIdx.y] = 0;

        __syncthreads(); // Synchronize threads in the block before computation

        // Compute the contribution for c[row][col]
        for (int j = 0; j < BLOCK_SIZE; j++)
        {
            temp += tile_a[threadIdx.y][j] * tile_b[threadIdx.x][j];
        }
```

使用加速方法，将b矩阵转置后，本来觉得可以加快缓存读取，但是发现程序速度反而变慢了

```
Testting the diy_cuda_global_mem implementation of matrix multiply
510.3349 520.4258 ... 514.6938 517.7357
508.9028 ...... ... ...... 520.2448
...... ...... ... ...... ......
507.9561 ...... ... ...... 513.7372
521.5385 526.4379 ... 520.4982 516.9260
Custom Kernel Time: 3.424504 ms Performance: 5016.746651 GFLOPS

Testting the diy_cuda_shared_mem implementation of matrix multiply
510.3349 520.4258 ... 514.6938 517.7357
508.9028 ...... ... ...... 520.2448
...... ...... ... ...... ......
507.9561 ...... ... ...... 513.7372
521.5385 526.4379 ... 520.4982 516.9260
Custom Kernel Time: 2.797537 ms Performance: 6141.069320 GFLOPS

Testting the cublas implementation of matrix multiply
506.5235 519.2048 ... 503.0756 527.3256
513.7955 ...... ... ...... 526.7343
...... ...... ... ...... ......
502.1938 ...... ... ...... 514.4681
490.7425 508.1088 ... 486.1488 503.2191
cuBLAS Time: 2.810711 ms        Performance: 6112.286092 GFLOPS
(base) hongjie@kemove-ESC8000-G4:~/ml/cuda/MM$ make
nvcc compare.cu -lcublas -o main
(base) hongjie@kemove-ESC8000-G4:~/ml/cuda/MM$ ./main 2048 2048 2048 32

Testting the diy_cuda_global_mem implementation of matrix multiply
510.3349 520.4258 ... 514.6938 517.7357
508.9028 ...... ... ...... 520.2448
...... ...... ... ...... ......
507.9561 ...... ... ...... 513.7372
521.5385 526.4379 ... 520.4982 516.9260
Custom Kernel Time: 3.419901 ms Performance: 5023.499135 GFLOPS

Testting the diy_cuda_shared_mem implementation of matrix multiply
510.3349 520.4258 ... 514.6938 517.7357
508.9028 ...... ... ...... 520.2448
...... ...... ... ...... ......
507.9561 ...... ... ...... 513.7372
521.5385 526.4379 ... 520.4982 516.9260
Custom Kernel Time: 8.448617 ms Performance: 2033.453430 GFLOPS
```

```
Testting the cublas implementation of matrix multiply
506.5235 519.2048 ... 503.0756 527.3256
513.7955 ...... ... ...... 526.7343
...... ...... ... ...... ......
502.1938 ...... ... ...... 514.4681
490.7425 508.1088 ... 486.1488 503.2191
cuBLAS Time: 2.828008 ms        Performance: 6074.901333 GFLOPS
```

后来想想,确实更改后会更慢,因为本来的运行模型是32个线程作为一个warp来进行SIMT,然后如果是之前的代码

```
temp += tile_a[threadIdx.y][j] * tile_b[j][threadIdx.x];
```

虽然在循环的时候涉及了列优先的访问,但是对于**整个warp来说是行优先**的访问(根据Idx.x)。

这本质上是对矩阵的一维分块，从代码可以看到本质是将矩阵分为了条带，此外，每个线程对应一个输出，**计算强度还不够。**

时间测试：最后一起给出

任务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（参考《计算机体系结构-量化研究方法》第四章）。

CUBLAS参考资料《CUBLAS_Library.pdf》，CUBLAS矩阵乘法参考第70页内容。

CUBLAS矩阵乘法例子，参考附件《matrixMulCUBLAS》

以下是比较矩阵乘法的表格，元素左边是计算时间ms，右边是浮点性能GFLOPS

代码解释：

```
//   m*k k*n = m*n
    cublasH     cublasHandle_t handle;
    cublasCreate(&handle);
    // Configure cuBLAS operations
    cublasOperation_t opA = CUBLAS_OP_N; // No transpose for A
    cublasOperation_t opB = CUBLAS_OP_N; // No transpose for B
    cublasSgemm(handle, opA, opB, n, m, k, &alpha, d_B, n, d_A, k, &beta, d_C,
n); // 我日，cublas 居然是列优先的
```

因为cublas使用列优先，所以输入为**行优先分配的矩阵**相当于先转置，BT*AT=CT， CT会列优先存到内存，再行优先就是C（这也太搞了）

测试运行（连同任务1）

包含三个算法，**全局内存，共享内存，cublas**，测试了 blocksize从**8~32**， 矩阵维度从**512到8192**

```
run:
    @for n in 8 16 24 32; do \
        for size in 512 1024 2048 4096 8192; do \
            echo "Running program with size $$size and $$n BLOCK" | tee -a
output.txt; \
            ./$(PROGRAM) $$size $$size $$size $$n | tee -a output.txt; \
        done; \
        echo "End of program with $$n BLOCK SIZE" | tee -a output.txt; \
        echo "" | tee -a output.txt; \
    done; \
    echo "End of program" | tee -a output.txt; \
    echo "" | tee -a output.txt;
```

使用的4090显卡数据如下：

```
Device 0: NVIDIA GeForce RTX 4090
  Clock Rate (GHz): 2.52 GHz
  CUDA Cores: 16384
  Theoretical Peak Performance (FP32): 82.5754 TFLOPS
Shared memory per block:  49152  bytes
```

DIY CUDA Global Memory Matrix Multiply (Time/Performance)

Matrix Size	BLOCK 8	BLOCK 16	BLOCK 32
512	0.1193 ms / 2250.54	0.1390 ms / 1930.83	0.0946 ms / 2836.53
1024	0.6139 ms / 3497.88	0.4744 ms / 4527.01	0.4727 ms / 4542.81
2048	4.2528 ms / 4039.63	3.4347 ms / 5001.92	3.4229 ms / 5019.14
4096	40.2058 ms / 3418.39	26.9783 ms / 5094.43	26.9785 ms / 5094.38
8192	1086.5767 ms / 1011.90	303.5841 ms / 3621.77	252.0375 ms / 4362.49

DIY CUDA Shared Memory Matrix Multiply (Time/Performance)

Matrix Size	BLOCK 8	BLOCK 16	BLOCK 32
512	0.0584 ms / 4597.13	0.0552 ms / 4861.82	0.0557 ms / 4821.65
1024	0.3469 ms / 6189.68	0.3154 ms / 6808.68	0.3524 ms / 6093.97
2048	2.6140 ms / 6572.33	2.3665 ms / 7259.53	2.7014 ms / 6359.50
4096	20.6395 ms / 6659.03	18.6708 ms / 7361.15	23.2045 ms / 5922.94
8192	155.7412 ms / 7059.86	144.1234 ms / 7628.96	188.4394 ms / 5834.83

cuBLAS Matrix Multiply (Time/Performance)

Matrix Size	BLOCK 8	BLOCK 16	BLOCK 32
512	0.6181 ms / 434.32	0.5777 ms / 464.68	0.6056 ms / 443.29
1024	0.6850 ms / 3135.05	0.6126 ms / 3505.37	0.6703 ms / 3203.91
2048	0.9074 ms / 18933.86	0.8666 ms / 19824.02	0.8702 ms / 19741.35
4096	3.3481 ms / 41050.03	3.3021 ms / 41621.72	3.3437 ms / 41104.28
8192	19.0395 ms / 57749.12	18.9696 ms / 57961.68	19.0230 ms / 57798.94

结果分析：

首先查看GPU信息

```
Device 0: NVIDIA GeForce RTX 4090
  Compute Capability: 8.9
  Max Threads per Block: 1024
  Max Threads per Multiprocessor: 1536
  Threads per Warp: 32
  Max Warps per Multiprocessor: 48
  Max Registers per Block: 65536
  Max Registers per Multiprocessor: 65536
  Registers Allocation Granularity: Warp
  Total Global Memory: 24217 MB
  Max Shared Memory per Block: 48 KB
  Shared Memory per Multiprocessor: 100 KB
  Multiprocessor Count: 128
```

1. cublas库的运行不受blocksize的影响，cublas有一套控制逻辑选择blocksize和矩阵乘算法；

2. 对于普通的全局内存的函数，增大blocksize有助于提高运行速度，这是因为block是GPU调度的基本单位，提高blocksize，能减少gpu控制调度的时间，提高并行能力；

3. 对于共享内存的版本，每个SM上的共享内存块是100KB,一个block分配了2x32x32x4B共8KB共享内存，可以支持10加个块共同运行，这不会成为限制。然而当blocksize增多到一定限度的时候，会影响SM同时运行块的数量，从而使得占用率下降，比如16x16=256个线程，8个warp，9472个寄存器，一个SM有1536个线程，48个warps，65536个寄存器，可以同时支持6个BLOCK同时进行，此时的占用率是100%。（数量更少的8x8也会是100,但是需要更多的控制调度时间，导致性能没有那么好），但是对于24x24=576，32x32=1024,一个SM只能同时支持2个和1个块运行，占用率分别是75%和66%。这就可以**解释为什么块线程数增加，性能反而下降**；

4. 对于规整的矩阵乘法，在部分任务上，**在矩阵维度小于等于1024的时候,** 共享内内存版本能超越cublas，但是也能看到cublas对于不同问题规模的运算的效率都保持在**高水平。**

5. 在cublas和共享内存的对比中，当矩阵维度增大的时候,大于1024的时候，共享内存函数是**不如cublas的。**下面分析一下原因，首先查看GPU的占用率是否有限制。对于一个流多处理器，同时只能有一个块在运行，这是因为此时一个块需要1024线程，一个流多处理器允许1536个线程运行。此时的占用率是66%（根据有效的warps的数量计算，32/48=66%）。这不应该是性能这么低的原因（现在只有7TFLOPs,峰值有82TFLOPs，不足10%)。

所以考虑访存和计算的平衡问题，推测在访问共享内存的时候出现了stall for 共享内存，我们的程序一个线程只针对一个对应元素，每次的共享内存访问，执行一次加乘操作，可以**考虑增加计算强度**，在访存不变的情况下，同一个thread可以做更多的计算任务，之前是一个thread负责一个对应元素，现在可以**负责一个小区域的元素（将结果矩阵分块）**。

任务3：

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或者输入从32增加至512.

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，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.

输出：输出卷积结果以及计算时间

代码解释：

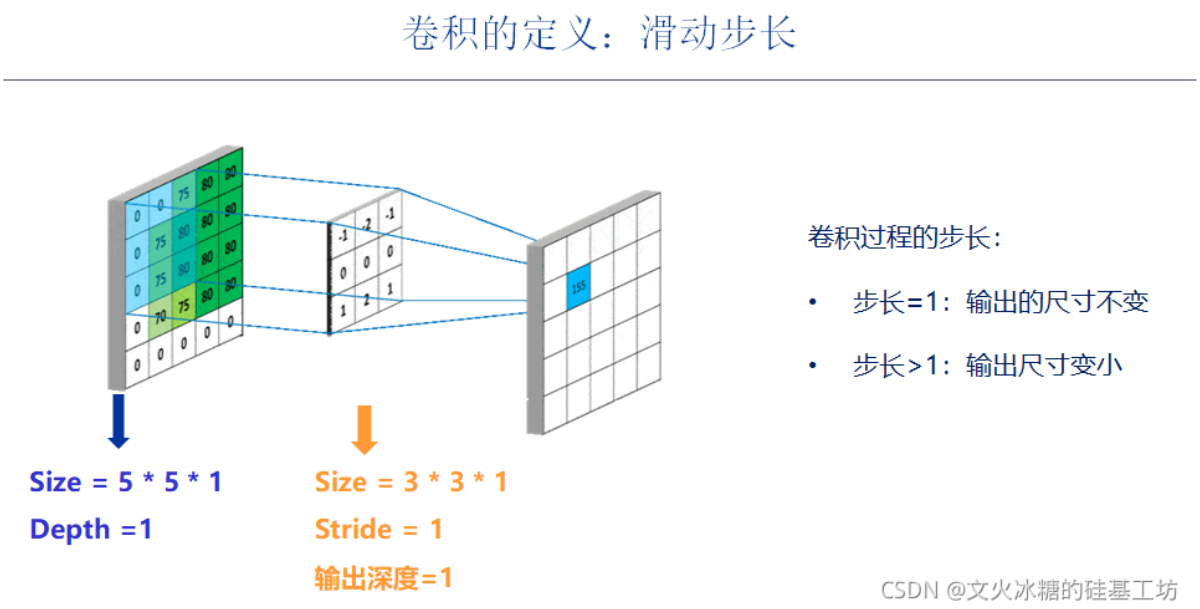
以常用的张量描述（N, C, H,W）,其中**N是batchsize，也就是张量数目， C是张量通道数，这里是RGB也就是3通道， H是高度也就是矩阵的行数, W是宽度也就是张量的列数，**

首先需要对输入矩阵进行padding. paddedBlock是矩阵pdding之后的大小， （i+padding）遍历是为了0到padding-1行都是0， 然后（j+padding）是为了让左边的列padding为0.

```
// m n, depth分别是矩阵的行，列，通道数
float *padMatrix(const float *matrix, float *padded_matrix, int m, int n, int
depth, int padding)
{
    int rows = m;
    int cols = n;
    int paddedCols = n + 2 * padding;
    int paddedRows = m + 2 * padding;
    int paddedBlcok = paddedRows * paddedCols;
    int originblock = rows * cols;
    for (int k = 0; k < depth; k++)
    {
        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < cols; ++j)
            {
                padded_matrix[k * paddedBlcok + (i + padding) * paddedCols + j +
padding] = matrix[k * originblock + i * n + j];
            }
        }
    }
    return padded_matrix;
}
```

```
    memset(temp, 0, sizeof(float) * paddedRows * paddedCols * input.matrix_nums);
 //需要把原来的内存设置为0，否则padding的时候可能会出错
    padMatrix(h_input, temp, m, n, input.matrix_nums, padding);
```

然后是滑窗卷积函数：



- (row % stride == 0 && col % stride == 0是为了步长滑动，相当于滑动卷积窗口
- row < paddedm - kernelSet.rows + 1 && col < paddedn - kernelSet.cols + 1是为了卷积窗口在边界的时候不会越界（避免滑出矩阵边界）
- 然后就是三次遍历，分别是通道，行，列，将卷积核和对应的局部矩阵作向量点积，最后化为一个数字，temp
- 因为是每一个thread负责一个对应输出的元素，所以不存在共享变量读取竞争的问题

```
//输入m,n是原本的行列数
__global__ void conv2d_cal(float *input, float *output, int output_block, int m,
int n, const Kernel kernelSet, int stride, int padding)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int paddedm = m + 2 * padding;
    int paddedn = n + 2 * padding;

    int conved_n = (n - kernelSet.cols + 2 * padding) / stride + 1;  //计算卷积后的
列数

    // extern __shared__ float temp[]; // 使用动态共享内存，适配 kernelSet.numKernels
注意会出现竞争！！！
    float temp;

    if (row % stride == 0 && col % stride == 0 && row < paddedm - kernelSet.rows
+ 1 && col < paddedn - kernelSet.cols + 1)
    {
        for (int k = 0; k < kernelSet.numKernels; k++)
        {
            for (int i = 0; i < kernelSet.rows; i++)
            {
                for (int j = 0; j < kernelSet.cols; j++)
```

```
                {
                    int input_row = row + i;
                    int input_col = col + j;
                    int kernel_idx = k * kernelSet.rows * kernelSet.cols + i *
kernelSet.cols + j;
                    temp += input[input_row * paddedn + input_col] *
kernelSet.deviceKernels[kernel_idx];
                }
            }
            // 写入输出
        }
        output[row * conved_n + col] = temp;
    }
}
```

测试运行：

正确性

对于卷积任务（1，3，5，5） （1，3，3，3）stride =x padding =1验证

```
Kernel 0 values:
1 2 3
4 5 6
7 8 9
Kernel 1 values:
1 2 3
4 5 6
7 8 9
Kernel 2 values:
1 2 3
4 5 6
7 8 9
atfer padding:
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 0.000000
0.000000 6.000000 7.000000 8.000000 9.000000 0.000000 0.000000
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 0.000000
0.000000 6.000000 7.000000 8.000000 9.000000 0.000000 0.000000
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

验证stride =1的正确性，打印输出矩阵(0,2)元素的一个通道的计算过程，可以看到shape正确，计算过程也正确(3个通道要乘3)

```
row: 0, col: 2, i: 0, j:0, input:0.000000, kernel:1.000000, temp: 0.000000
row: 0, col: 2, i: 0, j:1, input:0.000000, kernel:2.000000, temp: 0.000000
row: 0, col: 2, i: 0, j:2, input:0.000000, kernel:3.000000, temp: 0.000000
row: 0, col: 2, i: 1, j:0, input:2.000000, kernel:4.000000, temp: 8.000000
row: 0, col: 2, i: 1, j:1, input:3.000000, kernel:5.000000, temp: 23.000000
row: 0, col: 2, i: 1, j:2, input:4.000000, kernel:6.000000, temp: 47.000000
row: 0, col: 2, i: 2, j:0, input:7.000000, kernel:7.000000, temp: 96.000000
row: 0, col: 2, i: 2, j:1, input:8.000000, kernel:8.000000, temp: 160.000000
row: 0, col: 2, i: 2, j:2, input:9.000000, kernel:9.000000, temp: 241.000000
Custom Kernel Time for sliding conv: 4.198948 ms
```

```
Printing matrix of printing 0th matrix
384.000000 606.000000 723.000000 570.000000 312.000000
318.000000 513.000000 648.000000 603.000000 354.000000
483.000000 738.000000 873.000000 648.000000 339.000000
318.000000 513.000000 648.000000 603.000000 354.000000
150.000000 228.000000 291.000000 264.000000 150.000000
```

验证stride =2的正确性,打印出输出矩阵的(0,1)元素一个通道的计算过程，可以看到，shape是正确的

```
row: 0, col: 2, i: 0, j:0, input:0.000000, kernel:1.000000, temp: 0.000000
row: 0, col: 2, i: 0, j:1, input:0.000000, kernel:2.000000, temp: 0.000000
row: 0, col: 2, i: 0, j:2, input:0.000000, kernel:3.000000, temp: 0.000000
row: 0, col: 2, i: 1, j:0, input:2.000000, kernel:4.000000, temp: 8.000000
row: 0, col: 2, i: 1, j:1, input:3.000000, kernel:5.000000, temp: 23.000000
row: 0, col: 2, i: 1, j:2, input:4.000000, kernel:6.000000, temp: 47.000000
row: 0, col: 2, i: 2, j:0, input:7.000000, kernel:7.000000, temp: 96.000000
row: 0, col: 2, i: 2, j:1, input:8.000000, kernel:8.000000, temp: 160.000000
row: 0, col: 2, i: 2, j:2, input:9.000000, kernel:9.000000, temp: 241.000000
Custom Kernel Time for sliding conv: 4.179901 ms
Printing matrix of printing 0th matrix
384.000000 723.000000 312.000000
483.000000 873.000000 339.000000
150.000000 291.000000 150.000000
```

验证stride =3的正确性,打印出输出矩阵的(0,1)元素一个通道的计算过程，可以看到，shape是正确的 结果也正确

```
row: 0, col: 3, i: 0, j:0, input:0.000000, kernel:1.000000, temp: 0.000000
row: 0, col: 3, i: 0, j:1, input:0.000000, kernel:2.000000, temp: 0.000000
row: 0, col: 3, i: 0, j:2, input:0.000000, kernel:3.000000, temp: 0.000000
row: 0, col: 3, i: 1, j:0, input:3.000000, kernel:4.000000, temp: 12.000000
row: 0, col: 3, i: 1, j:1, input:4.000000, kernel:5.000000, temp: 32.000000
row: 0, col: 3, i: 1, j:2, input:5.000000, kernel:6.000000, temp: 62.000000
row: 0, col: 3, i: 2, j:0, input:8.000000, kernel:7.000000, temp: 118.000000
row: 0, col: 3, i: 2, j:1, input:9.000000, kernel:8.000000, temp: 190.000000
row: 0, col: 3, i: 2, j:2, input:0.000000, kernel:9.000000, temp: 190.000000
Custom Kernel Time for sliding conv: 4.153343 ms
Printing matrix of printing 0th matrix
384.000000 570.000000
318.000000 603.000000
```

时间测试

在最后和其他方法一起给出

任务4:

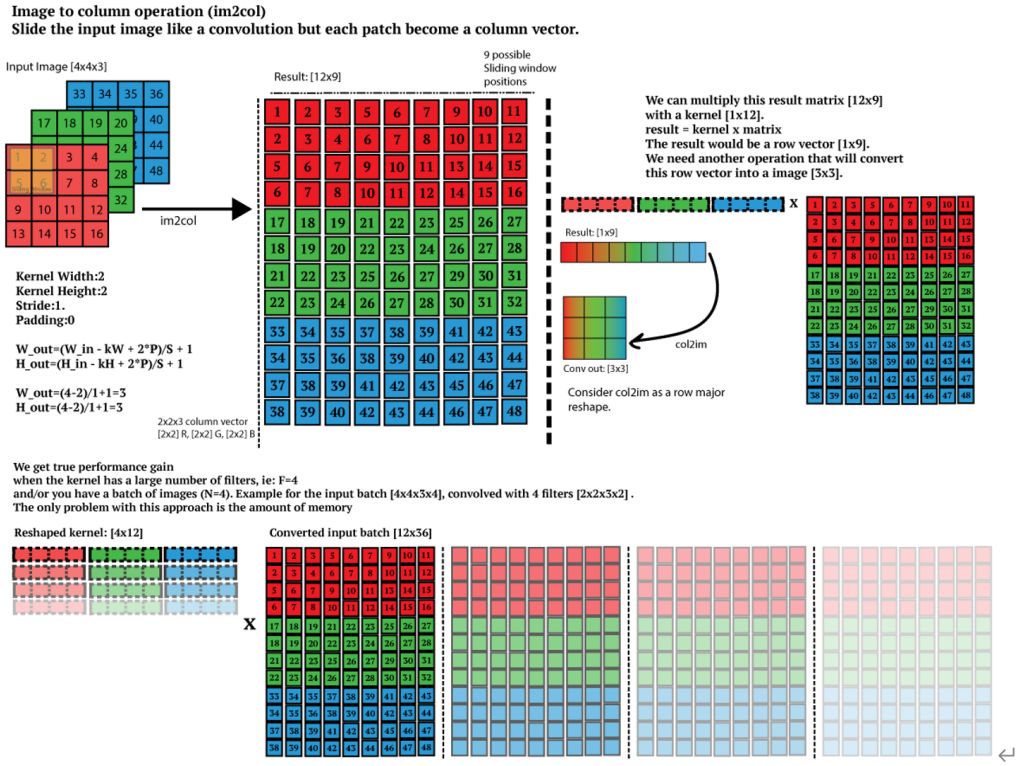
使用im2col方法结合任务1实现的GEMM（通用矩阵乘法）实现卷积操作。输入从256增加至4096或者输入从32增加至512，具体实现的过程可以参考下面的图片和参考资料。

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。 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。



代码解释：

首先还是先补充padding,同上省略

然后是最麻烦的，将张量展开为矩阵

先计算出展开后的形状，高度（行数）是卷积核的大小 filter_size， 宽度是列数，是卷积后矩阵的大小 **converted_n**

然后五层展开，第一二层移动卷积窗口，里面三层是复制原矩阵的内容到展开后的矩阵

```
    int padded_m = h_input.rows + 2 * padding;  //padding
    int padded_n = h_input.cols + 2 * padding;
    int convable_m = (h_input.rows + 2 * padding - kernelSet.rows) / stride + 1;
//after convolution
    int convable_n = (h_input.cols + 2 * padding - kernelSet.cols) / stride + 1;

    int converted_n = convable_m * convable_n;  // converted to a flat matrix
    int converted_m = kernelSet.rows * kernelSet.cols;

    int block = converted_m * converted_n;
    int padded_block = padded_m * padded_n;

    float *temp;
    // printf("hello\n");
    cudaMallocHost((void **)&temp, sizeof(float) * padded_block *
h_input.matrix_nums);  //create a temp to get the padded matrix in host mem
```

```
    padMatrix(h_input.h_input, temp, h_input.rows, h_input.cols,
h_input.matrix_nums, padding);


    float *h_temp;
    cudaMallocHost((void **)&h_temp, sizeof(float) * h_input.matrix_nums *
block);  //for converted matrix
    for (int i = 0; i +kernelSet.rows-1 < padded_m; i+=stride) // the next 2
iterations are for sliding the conv window
    {
        for (int j = 0;  j+kernelSet.cols-1<padded_n; j+=stride)
        {
            for (int a = 0; a < kernelSet.rows; a++) // the naxt 2 iterations are
for inside the conv window
            {
                for (int b = 0; b < kernelSet.cols; b++)
                {
                    for (int k = 0; k < h_input.matrix_nums; k++) // the k
iteration for the input matrixs(could be more than 1)
                    {
                        int li = k * block + (a * kernelSet.cols + b) *
converted_n + (i/stride) * convable_n + (j/stride);
                        int ri = k * padded_block + (i + a) * padded_m + j + b;
                        h_temp[li] = temp[ri];
                    }
                }
            }
        }
    }
```

然后是卷积核的展开，然后直接调用之前实现的矩阵乘算法，就可以实现结果的计算.

Img2col展开后的矩阵乘法是**不规整的**,对于普通实现的矩阵乘法压力很大,所以我在矩阵乘法阶段直接使用了**cublas的矩阵乘法库函数**

```
void gpu_gemm_cublas(const float *d_a, const float *d_b, float *d_result, int m,
int n, int k)
{
    //   m*n x n*k =m *k
    cublasHandle_t handle;
    cublasCreate(&handle);
    // Configure cuBLAS operations
    const float alpha = 1.0, beta = 0.0;
    cublasOperation_t opA = CUBLAS_OP_N;
  // No transpose for A
    cublasOperation_t opB = CUBLAS_OP_N;
  // No transpose for B
    cublasSgemm(handle, opA, opB, k, m, n, &alpha, d_b, k, d_a, n, &beta,
d_result, k); // 我日，cublas 居然是列优先的
    // cudaMemcpy(h_output.h_input, d_result, sizeof(float)*converted_n,
cudaMemcpyDeviceToHost);
}
```

另外在实验中发现,程序的大量时间都用在了**展开矩阵上,后续优化可以考虑将展开写为核函数.**

展开优化:

将Im2col的padding和展开部分分配到gpu上运行,参考caffe的实现方法

```
template <typename Dtype>
__global__ void im2col_gpu_kernel(const int n, const Dtype *data_im,
                                  const int height, const int width, const int
ksize, const int pad,
                                  const int stride, const int height_col, const
int width_col,
                                  Dtype *data_col)
{
    CUDA_KERNEL_LOOP(index, n) //宏函数,为每一个index启动线程
    {
        int w_out = index % width_col; //计算线程对应的输出元素的列索引
        int h_index = index / width_col;
        int h_out = h_index % height_col; //计算线程对应的输出元素的行索引
        int channel_in = h_index / height_col; //计算线程对应的输出元素的通道位置
        int channel_out = channel_in * ksize * ksize;
        int h_in = h_out * stride - pad;
        int w_in = w_out * stride - pad; //反向计算im2col映射前的元素位置
        Dtype *data_col_ptr = data_col;
        data_col_ptr += (channel_out * height_col + h_out) * width_col + w_out;
//获取映射后的指针位置
        const Dtype *data_im_ptr = data_im;
        data_im_ptr += (channel_in * height + h_in) * width + w_in; //映射前的位置
        for (int i = 0; i < ksize; ++i)
        {
            for (int j = 0; j < ksize; ++j)
            {
                int h = h_in + i;
                int w = w_in + j;
                *data_col_ptr = (h >= 0 && w >= 0 && h < height && w < width) ?
data_im_ptr[i * width + j] : 0;
```

```
                data_col_ptr += height_col * width_col;
            }
        }
    }
}
```

问题调试

在实现这个Img2col算法的时候，出现了奇怪的问题，核函数只能算出32个数字，根据对核函数，也就是之前实现的矩阵乘法函数的分析：

```
__global__ void gpu_matrix_mult_gm(const 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;
    float temp = 0;
    if (row < m && col < k) // Ensure bounds are within the matrix dimensions
    {
        for (int i = 0; i < n; i++)
        {
            temp += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = temp;
        printf("%d %d index:%d, reuslt: %d\n", row, col, row * k + col ,temp);
    }
}
```

以及切分矩阵的语句：

```
    dim3 blockDim(32,32);
    dim3 gridDim(1, 1); //这里我以为1024的线程足够计算简单任务，没有设置
```

可以发现对于**瘦高的**矩阵乘法，之前对方块矩阵的切分方式可能导致有的元素没有参加运算，比如（1， 3， 1024，1024）的图像矩阵和(1, 3, 3, 3)卷积核的卷积， 等价(1, 27)的卷积核向量和（27，1024*1024）的矩阵相乘，之前的矩阵分割算法，是对右边矩阵的列和左边矩阵的行分别进行分割，然后对应相乘计算出结果，这里行是1，列是1024x1024，如果使用32，32的block shape，就会有行方向31组线程浪费， 列方向又远远不足以计算任务。

解决的办法：重新设置切割语句，适应不同矩阵乘法的需求

```
    int blockx = 256, blocky = 1;  //和warp对应，还是需要blocksize是32的倍数， 但这个时候，设置为矮宽的形状
    dim3 blockDim(blockx, blocky);
    int gridx = (converted_n + blockx - 1) / blockx;  //计算网格
    int gridy = 1;  //由于左边向量的高度还是1，所以考虑块的排布也都是扁平的
    dim3 gridDim(gridx, gridy);
```

这个时候出现了另外一个问题，就是使用滑动窗口算法，和使用img2col算法（共享内存版本），他们都只能算矩阵维度最大是1022的，由于1022+2=1024是32的倍数，这个时候我考虑是blocksize大小的问题，我还需要重新检查这两个算法，确定他们的切分方式和blocksize的关系，同时共享内存之前的写法也是和blocksize关联的，但是内存是有限的，不能让blocksize太大，所以需要进一步检查。

好吧，原来是我错误的将block和grid写反了，而4090gpu的threadperblock的限制是1024，当维度一大，将grid和block搞反就以为着会超过1024的限制，导致报错。

```
<<<BLock, Grid>> kernel_function();//我在这里搞反了顺序,Grid应该在前面
```

测试运行：

正确性:

和上面的任务使用同样的输入确定正确性

```
stride :1
Custom Kernel Time for img2col conv: 450.193542 ms
Printing matrix of printing 0th matrix
384.000000 606.000000 723.000000 570.000000 312.000000
318.000000 513.000000 648.000000 603.000000 354.000000
483.000000 738.000000 873.000000 648.000000 339.000000
318.000000 513.000000 648.000000 603.000000 354.000000
150.000000 228.000000 291.000000 264.000000 150.000000
stride:2
Custom Kernel Time for img2col conv: 446.803680 ms
Printing matrix of printing 0th matrix
384.000000 723.000000 312.000000
483.000000 873.000000 339.000000
150.000000 291.000000 150.000000
stride:3
Custom Kernel Time for img2col conv: 424.988342 ms
Printing matrix of printing 0th matrix
384.000000 570.000000
318.000000 603.000000
```

时间测试

Img2Col Convolution 时间统计表

Size	Stride 1 (Total)	Stride 2 (Total)	Stride 3 (Total)
512	521.000000	453.000000	655.000000
1024	792.000000	735.000000	456.000000
2048	2101.000000	791.000000	559.000000
4096	6533.000000	2118.000000	914.000000
8192	55305.000000	8891.000000	2390.000000

其中展开时间表：

Size	Stride 1 (ms)	Stride 2 (ms)	Stride 3 (ms)
512	88.000000	21.000000	7.000000
1024	344.000000	89.000000	30.000000

Size	Stride 1 (ms)	Stride 2 (ms)	Stride 3 (ms)
2048	1490.000000	357.000000	121.000000
4096	6115.000000	1678.000000	486.000000
8192	54858.000000	8463.000000	1957.000000

计算时间表：

Size	Stride 1 (ms)	Stride 2 (ms)	Stride 3 (ms)
512	433.000000	432.000000	648.000000
1024	448.000000	646.000000	426.000000
2048	611.000000	434.000000	438.000000
4096	418.000000	440.000000	428.000000
8192	447.000000	428.000000	433.000000

Img2Col Convolution with parallel convertion时间表（ms）

Size	Stride 1 (ms)	Stride 2 (ms)	Stride 3 (ms)
512	648.353394	680.233948	688.773865
1024	438.955048	436.664185	428.066742
2048	443.277802	447.922974	443.355103
4096	731.962097	490.945740	497.696686
8192	1125.347046	849.608459	931.019470

对比滑窗法：

Matrix Size	Stride 1 Time (ms)	Stride 2 Time (ms)	Stride 3 Time (ms)
512	**7.709874**	**7.900139**	**9.811203**
1024	**26.421349**	**34.015903**	**30.129812**
2048	**104.506943**	**108.413010**	**105.430489**
4096	**444.949066**	**417.461884**	**434.954102**
8192	1763.79	1575.86	1716.98

结果分析：

对于im2col，当矩阵维度变得比较大的时候，大部分的计算任务是在矩阵转换为列矩阵的部分，这部分可以用cuda并行加速完成，

对比滑窗法，im2col仅仅在size>8192的时候能比滑窗法要快，这是因为im2col之后的矩阵乘法是一个1x27x8192^2的矩阵乘法。对计算时间的进一步分析发现，cublas句柄分配是一个很消耗时间的任务：

```
GPU memory allocation and copy time: 362.197876 ms
GPU im2col_gpu time: 9.224544 ms
GPU cublas handle malloc time: 611.127319 ms
GPU cublasSgemm time: 8.189952 ms
GPU result copy back to host time: 62.132927 ms
Total CPU time: 1053 ms
Custom Kernel Time for img2colv2 conv: 1061.025513 ms
```

可以说**90%以上**的时间用在了**句柄分配和显卡内存开辟和复制**，而且句柄分配是一个和问题规模无关的时间，**这导致了在小任务的时候，一次im2col卷积比滑窗法要更费时间**，查询资料可以知道，目前的深度学习训练框架也有在使用im2col，这可能时因为一次句柄分配后，后面的矩阵乘时间可以比分配时间要少很多，另外Im2col对于反向传播也有好处。

总结：在对im2col的展开和矩阵乘部分使用了GPU并行加速之后，剩下的时间大多是在**分配内存和句柄的阶段。**

任务5：

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。

CNN参考资料，见实验发布网站

斯坦福人工智能课件Convolutional Neural Networks，by Fei-Fei Li & Andrej Karpathy & Justin Johnson

其他参考资料 （搜索以下关键词）

[1]如何理解卷积神经网络（CNN）中的卷积和池化

[2] Convolutional Neural Networks (CNNs / ConvNets) https://cs231n.github.io/convolutional-networks/

[3]im2col的原理和实现

[4]cuDNN安装教程

[5] convolutional-neural-networks

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

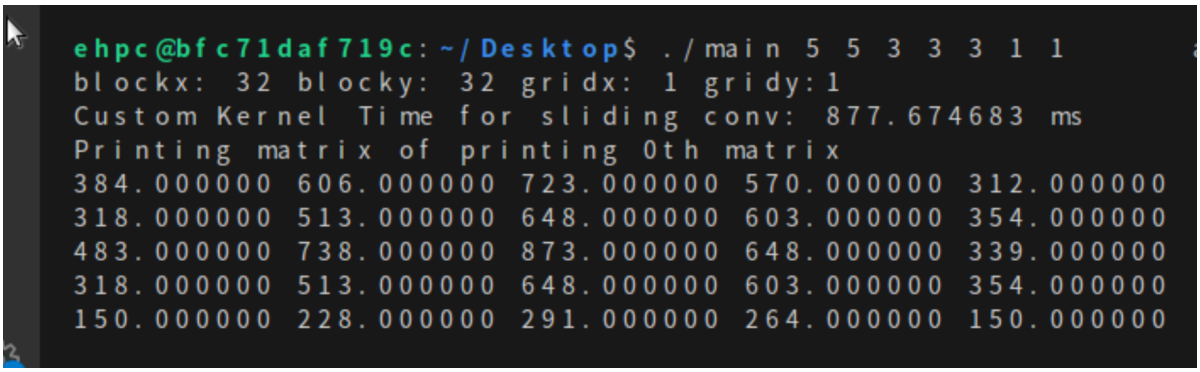
代码解释:

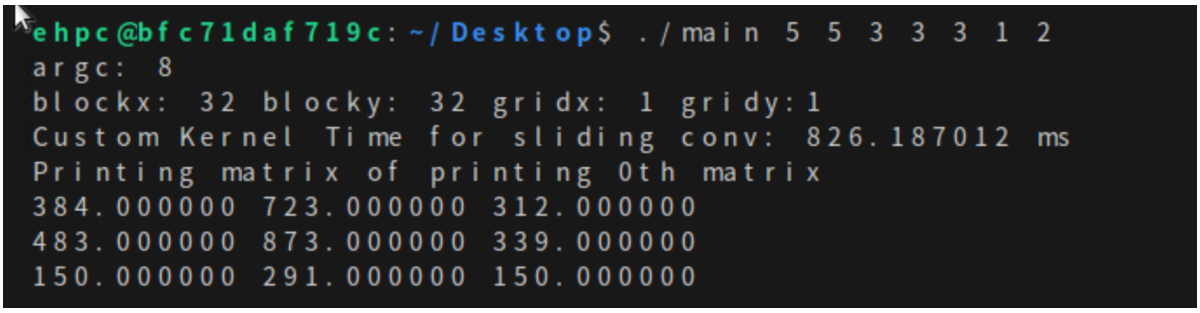
主要包含句柄分配,设置矩阵,卷积核,设置卷积参数,这期间可以比较卷积输出是否符合预期,然后实现卷积,最后将结果返回主机内存.

```
    // Create cuDNN descriptors
     CHECK_CUDNN_ERR(cudnnCreateTensorDescriptor(&input_desc));
     CHECK_CUDNN_ERR(cudnnCreateTensorDescriptor(&output_desc));
     CHECK_CUDNN_ERR(cudnnCreateFilterDescriptor(&kernel_desc));
```

```
    CHECK_CUDNN_ERR(cudnnCreateConvolutionDescriptor(&conv_desc));

    // Set input tensor descriptor (NCHW format)
    CHECK_CUDNN_ERR(cudnnSetTensor4dDescriptor(input_desc, CUDNN_TENSOR_NCHW,
CUDNN_DATA_FLOAT, N, C, H, W));

    // Set output tensor descriptor (NCHW format)
    CHECK_CUDNN_ERR(cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NCHW,
CUDNN_DATA_FLOAT, N, h_output.matrix_nums, conved_H, conved_W));

    // Set kernel descriptor (filter)
    CHECK_CUDNN_ERR(cudnnSetFilter4dDescriptor(kernel_desc, CUDNN_DATA_FLOAT,
CUDNN_TENSOR_NCHW, N, C, R, S));

    // Set convolution descriptor
    CHECK_CUDNN_ERR(cudnnSetConvolution2dDescriptor(conv_desc, padding, padding,
stride, stride, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));

    // Get the output dimensions from cuDNN
    int n, c, h, w;
    CHECK_CUDNN_ERR(cudnnGetConvolution2dForwardOutputDim(conv_desc, input_desc,
kernel_desc, &n, &c, &h, &w));

    // Check the computed dimensions
    if (n != 1 || c != h_output.matrix_nums || h != conved_H || w != conved_W)
    {
        printf("expected: %d,%d,%d\n", h_output.matrix_nums, conved_H, conved_W);
        printf("Err: computed output dimensions don't match expected ones. Got
(%d, %d, %d, %d)\n", n, c, h, w);
        return;
    }

    // Set the output tensor descriptor with the computed dimensions
    CHECK_CUDNN_ERR(cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NCHW,
CUDNN_DATA_FLOAT, N, c, h, w));

    // Perform the convolution
    float alpha = 1.0f, beta = 0.0f;
    CHECK_CUDNN_ERR(cudnnConvolutionForward(cudnn, &alpha, input_desc, d_input,
kernel_desc, d_kernels, conv_desc, CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM,
NULL, 0, &beta, output_desc, d_output));

    // Copy the result back to host memory
    cudaMemcpy(h_output.h_input, d_output, h_output.matrix_nums * conved_H *
conved_W * sizeof(float), cudaMemcpyDeviceToHost);
```

测试运行:





时间测试:

Sliding Conv Time (ms)

Matrix Size	Stride 1	Stride 2	Stride 3
512	13.31	13.26	13.24
1024	36.38	35.84	35.87
2048	126.31	126.02	124.41
4096	488.36	481.27	480.27
8192	1933.46	1925.79	1900.13

img2colv2 Conv Time (ms)

Matrix Size	Stride 1	Stride 2	Stride 3
512	1041.13	1011.65	1026.68
1024	1029.99	1008.89	1027.53
2048	1017.85	1033.57	1046.15
4096	1037.97	1058.15	1026.54
8192	1135.36	1115.61	1092.38

cuDNN Conv Time (ms)

Matrix Size	Stride 1	Stride 2	Stride 3
512	30.27	28.79	29.67

Matrix Size	Stride 1	Stride 2	Stride 3
1024	31.40	30.88	28.27
2048	42.17	35.85	34.07
4096	84.95	55.01	48.55
8192	257.61	133.21	110.77

结果分析

1. 滑窗卷积在矩阵维度比较小,小于等于1024的时候,和cudnn的实现在同一个时间数量级,**部分任务能比cudnn还要快**,这是因为cudnn包含判断卷积任务而选择算法的控制模块,这一部分在任务量小的时候成为了bottle neck;
2. im2col算法全部保持在一个稳定的时间量级,通过查看各个步骤的时间消耗,发现最主要的时间消耗来自cublas句柄分配,甚至占据了95%以上的时间,如下图:

   ```
   GPU memory allocation and copy time: 1.483744 ms
   GPU im2col_gpu time: 0.164096 ms
   GPU cublas handle malloc time: 1038.322754 ms
   GPU cublasSgemm time: 0.810816 ms
   GPU result copy back to host time: 0.534560 ms
   Total  time: 1041 ms
   ```

   可以看到大部分的时间分配在cublas句柄分配,考虑优化,可以**提前分配句柄,或者实现句柄复用**
3. cudnn算法在小矩阵的时候不如滑窗算法,这也是因为cudnn分配句柄需要一定的时间,分配句柄需要分配内存资源和初始化,是非常消耗时间的.但是对于大矩阵,cudnn的效果要比滑窗法和Im2col要快,可能是cudnn**对cublas句柄实现了优化.**