Lab 6: CUDA 矩阵乘法和卷积

学号：19335109	课程：高性能计算
姓名：李雪堃	学期：Fall 2021
专业：计算机科学与技术（超算）	教师：黄聃、卢宇彤
邮箱：i@xkun.me	TAs：江嘉治、刘亚辉

Table of Contents

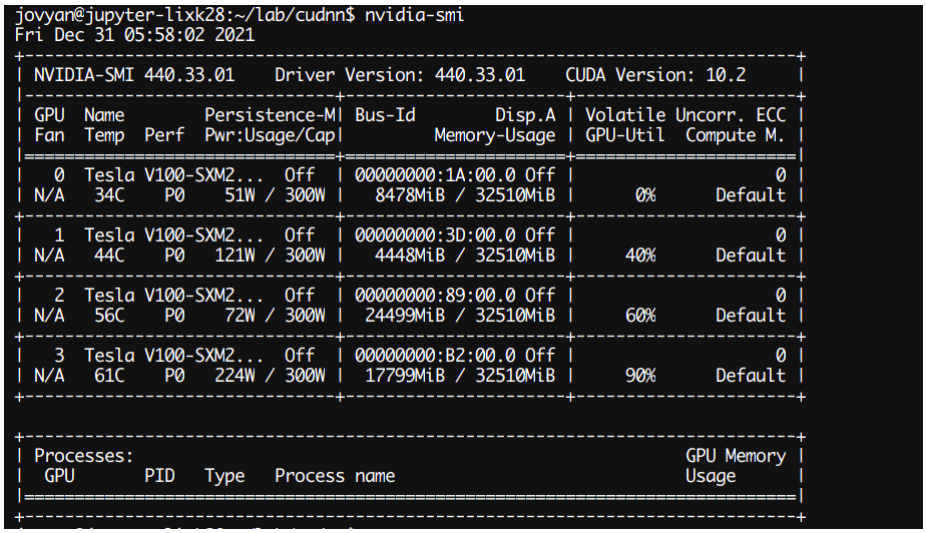
(一) 实验任务

- 任务一：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。
- 任务二：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。
- 任务三：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，bias 设置为 0。
- 任务四：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。

- 任务五：使用 NVIDIA 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) 实验环境

- 学院集群，GPU 配置如下：



(三) 实验过程和核心代码

(1) CUDA 通用矩阵乘法

代码在 `cuda-mm` 下。

CUDA 矩阵乘法：

- 函数将 `cudaMalloc` 和 `cudaMemcpy` 、设置 grid 和 block、调用核函数的过程封装起来，接收矩阵 A (m x k)、B (k x n)、C (m x n)，其中 A 和 B 参与计算，C 是计算结果。
- 根据老师的意思，grid 和 block 都采用一维划分，固定线程总数不变、改变 block size，如果用二维划分就很难写。
- 线程总数为矩阵 C 的行数 m，每个线程负责计算 C 的一行。

```cpp
void matrix_mul_cuda(double *A, double *B, double *C, int m, int k, int n, int block_size)
{
  double *A_d;
  double *B_d;
  double *C_d;
  cudaMalloc((void **)&A_d, sizeof(double) * m * k);  // use double pointer
  cudaMalloc((void **)&B_d, sizeof(double) * k * n);
  cudaMalloc((void **)&C_d, sizeof(double) * m * n);

  cudaMemcpy(A_d, A, sizeof(double) * m * k, cudaMemcpyHostToDevice);
  cudaMemcpy(B_d, B, sizeof(double) * k * n, cudaMemcpyHostToDevice);
  cudaMemcpy(C_d, C, sizeof(double) * m * n, cudaMemcpyHostToDevice);

  dim3 dim_grid(m / block_size);
  dim3 dim_block(block_size);

  #ifdef DEBUG
    printf("dim_grid(%d), dim_block(%d)\n", dim_grid.x, dim_block.x);
  #endif

  matrix_mul_cuda_kernel<<<dim_grid, dim_block>>>(A_d, B_d, C_d, m, k, n);

  cudaMemcpy(C, C_d, sizeof(double) * m * n, cudaMemcpyDeviceToHost);

  cudaFree(A_d);
  cudaFree(B_d);
  cudaFree(C_d);
}
```

CUDA 矩阵乘法核函数：

- 每个线程通过 `threadIdx` 和 `blockIdx` 计算自己负责的行索引，计算 C 的一行，这里还交换了一下循环的次序，减少一部分 cache miss。

```cpp
__global__ void matrix_mul_cuda_kernel(double *A, double *B, double *C, int m, int k, int n)
{
  int row = threadIdx.x + blockDim.x * blockIdx.x;
  for (int l = 0; l < k; l++) // switch the order of loop
  {
    for (int j = 0; j < n; j++)
      C[row * n + j] += A[row * k + l] * B[l * n + j];
  }
}
```

main 函数：

- main 函数的主要代码如下，m、k、n 是从命令行解析的参数，代表矩阵的维数。
- 设置 `srand` 的种子为固定值 20211225，这样每次生成的矩阵是固定的。
- 使用 `get_wall_time()` 函数获得程序运行的墙上时间。

```
double *A = (double *)malloc(sizeof(double) * m * k);
double *B = (double *)malloc(sizeof(double) * k * n);
double *C = (double *)malloc(sizeof(double) * m * n);

srand(20211225);
for (int i = 0; i < m; i++)
  for (int j = 0; j < k; j++)
    A[i * k + j] = 0 + rand() / (double) RAND_MAX * (10 - 0);
for (int i = 0; i < k; i++)
  for (int j = 0; j < n; j++)
    B[i * n + j] = 0 + rand() / (double) RAND_MAX * (10 - 0);
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    C[i * n + j] = 0;

double begin, end;

begin = get_wall_time();
matrix_mul_cuda(A, B, C, m, k, n, block_size);
end = get_wall_time();
printf("wall time of gemm_cuda, matrix size %d, block size %d: %.5lf\n", m, block_size, end - begin);

```

`get_wall_time()` 函数:

- `get_wall_time()` 函数使用 `gettimeofday` 函数获得墙上时间。

```
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}
```

(2) CUBLAS 矩阵乘法

代码在 `cublas-mm` 下。

CUBLAS 矩阵乘法:

- 同样传递三个矩阵和它们的维数作为函数参数。
- CUBLAS 矩阵乘 API（数据类型为 double）

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

- CUBLAS 矩阵乘法的计算公式

$$C = \alpha op(A) op(B) + \beta op(C)$$

其中，op 是指根据输入的参数来决定是否对矩阵进行转置或共轭转置。

$$op(M) = \begin{cases} M & \text{if transm} == \text{CUBLAS\_OP\_N} \\ M^T & \text{if transm} == \text{CUBLAS\_OP\_T} \\ M^H & \text{if transm} == \text{CUBLAS\_OP\_C} \end{cases}$$

我们需要计算的是 $C = AB$，因此 `alpha` 设置为 1.0、`beta` 设置为 0.0。注意到，`cublasDgemm` 默认矩阵是列优先存储的，而我们的矩阵是行优先存储，所以不对 A 和 B 进行转置的话，会将我们行优先存储的矩阵按照列优先的方式读取。

计算行优先的 C 就是计算 C^T，而 A^T 和 B^T 就是行优先存储的，我们已经准备好了，不需要进行转置：

$$C^T = (AB)^T = B^T A^T$$

所以调用 `cublasDgemm` 如下：

```
cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, &alpha, B_d, n, A_d, k, &beta, C_d, n);
```

不对行优先存储的 `A_d` 和 `B_d` 进行转置，设置为 `CUBLAS_OP_N`。

```
void matrix_mul_cublas(double *A, double *B, double *C, int m, int k, int n)
{
  double *A_d;
  double *B_d;
  double *C_d;

  cudaMalloc((void **)&A_d, sizeof(double) * m * k);  // use double pointer
  cudaMalloc((void **)&B_d, sizeof(double) * k * n);
  cudaMalloc((void **)&C_d, sizeof(double) * m * n);

  cudaMemcpy(A_d, A, sizeof(double) * m * k, cudaMemcpyHostToDevice);
  cudaMemcpy(B_d, B, sizeof(double) * k * n, cudaMemcpyHostToDevice);
  cudaMemcpy(C_d, C, sizeof(double) * m * n, cudaMemcpyHostToDevice);

  cublasHandle_t handle;
  cublasCreate(&handle);
  const double alpha = 1.0;
  const double beta = 0.0;
  cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, &alpha, B_d, n, A_d, k, &beta, C_d, n);  // compute C^T, which is row major of C

  cudaMemcpy(C, C_d, sizeof(double) * m * n, cudaMemcpyDeviceToHost);

  cudaFree(A_d);
  cudaFree(B_d);
  cudaFree(C_d);
  cublasDestroy(handle);
}
```

前一个任务中，实现的 CUDA 矩阵乘法很低效，下面采用二维划分的核函数，性能大大提升。每个线程负责计算 C 中的一个值。

```
__global__ void matrix_mul_cuda_kernel(double *A, double *B, double *C, int m, int k, int n)
{
  int row = blockDim.y * blockIdx.y + threadIdx.y;
  int col = blockDim.x * blockIdx.x + threadIdx.x;

  double value = 0;
  for (int l = 0; l < k; l++)
  {
    value += A[row * k + l] * B[l * n + col];
  }
  C[row * n + col] = value;
}
```

grid 和 block 的 dimension 设置如下：

- 如果 `n` 和 `m` 能被 `block_size` 整除，那么就可以完美地分配线程块，不用浪费。
- 否则向上取整，让 m x n 矩阵可以放进 grid 内，这时会有部分线程浪费掉，不做任何事情。

```
int grid_x = n % block_size == 0 ? n / block_size : n / block_size + 1;
int grid_y = m % block_size == 0 ? m / block_size : m / block_size + 1;
dim3 dim_grid(grid_x, grid_y);
dim3 dim_block(block_size, block_size);
```

(3) CUDA 二维卷积

CUDA 二维卷积函数：

- 采取 valid padding 策略，不做 padding，丢弃最右边的列和最下边的行。
- `block_size` 会根据 `ouput_size` 做调整，如果 `output_size` 过小，就设置 `block_size` 为 `output_size`。
- grid 的维度取决于 `output_size` 能否被 `block_size` 整除，不能被整除则向上取整，保证计算不会遗漏。

```
void cuda_conv2d(int *input, int *output, int *kernel, int input_height, int input_width, int kernel_height, int kernel_width, int block_size, int stride)
{
  int output_height = (input_height - kernel_height) / stride + 1;
  int output_width = (input_width - kernel_width) / stride + 1;

  int *input_d = NULL;
  int *output_d = NULL;
  int *kernel_d = NULL;

  // allocate memory on device
  cudaMalloc((void **)&input_d, sizeof(int) * input_height * input_width);
  cudaMalloc((void **)&output_d, sizeof(int) * output_height * output_width);
  cudaMalloc((void **)&kernel_d, sizeof(int) * kernel_height * kernel_width);

  // copy memory from host to device
  cudaMemcpy(input_d, input, sizeof(int) * input_height * input_width, cudaMemcpyHostToDevice);
  cudaMemcpy(kernel_d, kernel, sizeof(int) * kernel_height * kernel_width, cudaMemcpyHostToDevice);

  // if block_size is too large, then we set it as the size of output
  // or, we will partition output into blocks
  block_size = output_width < block_size ? output_width : block_size;
  dim3 dim_block(block_size, block_size);

  // if grid dimensions is divisible by block_size, then we can perfectly partition it
  // or, we add one row and one col to contain the remain part of output
  int grid_x = output_width % block_size == 0 ? output_width / block_size : output_width / block_size + 1;
  int grid_y = output_height % block_size == 0 ? output_height / block_size : output_height / block_size + 1;
  dim3 dim_grid(grid_x, grid_y);

  #ifdef DEBUG
    printf("dim_grid(%d, %d)\n", dim_grid.x, dim_grid.y);
    printf("dim_block(%d, %d)\n", dim_block.x, dim_block.y);
  #endif

  // call cuda conv2d kernel
  cuda_conv2d_kernel<<<dim_grid, dim_block>>>(input_d, output_d, kernel_d, input_height, input_width, kernel_height, kernel_width, stride);

  // copy conv2d output from device to host
  cudaMemcpy(output, output_d, sizeof(int) * output_height * output_width, cudaMemcpyDeviceToHost);

  // free allocated memory on device
  cudaFree(input_d);
  cudaFree(output_d);
  cudaFree(kernel_d);
}
```

CUDA 二维卷积核函数：

- 每个线程负责计算输出结果的一个值，即进行一次卷积运算。
- `input` 的索引可以通过步长 `stride` 和该线程负责计算的 `output` 的行列索引得到。

```
__global__ void cuda_conv2d_kernel(int *input, int *output, int *kernel, int input_height, int input_width, int kernel_height, int kernel_width, int stride)
{
  // output[row][col] is to be calculated in this thread
  int row = threadIdx.y + blockDim.y * blockIdx.y;
  int col = threadIdx.x + blockDim.x * blockIdx.x;

  int sum = 0;
  for (int i = 0; i < kernel_height; i++)
  {
    for (int j = 0; j < kernel_width; j++)
    {
      sum += kernel[i * kernel_width + j] * input[(row * stride + i) * input_width + (col * stride + j)];
    }
  }
  output[row * ((input_width - kernel_width) / stride + 1) + col] = sum;
}
```

CUDA 二维卷积 main 函数主要代码：

- 从命令行获取输入的高度和宽度、步长和线程块大小，卷积核设置为 3x3。
- 固定种子的值，每次生成相同数据的矩阵。

```c
int input_height = strtol(argv[1], NULL, 10);
int input_width = strtol(argv[2], NULL, 10);
int stride = strtol(argv[3], NULL, 10);
int block_size = strtol(argv[4], NULL, 10);

int kernel_height = 3;
int kernel_width = 3;

// just valid padding, for simplicity
int output_height = (input_height - kernel_height) / stride + 1;
int output_width = (input_width - kernel_width) / stride + 1;

int *input = (int *)malloc(sizeof(int) * input_height * input_width);
int *output = (int *)malloc(sizeof(int) * output_height * output_width);
int *kernel = (int *)malloc(sizeof(int) * kernel_height * kernel_width);

srand(20211231);
for (int i = 0; i < input_height * input_width; i++)
  input[i] = rand() % 5;

srand(20220101);
for (int i = 0; i < kernel_height * kernel_width; i++)
  kernel[i] = rand() % 5;

#ifdef DEBUG
  printf("input:\n"); print_matrix(input, input_height, input_width);
  printf("kernel:\n"); print_matrix(kernel, kernel_height, kernel_width);
#endif

double begin, end;

begin = get_wall_time();
cuda_conv2d(input, output, kernel, input_height, input_width, kernel_height, kernel_width, block_size, stride);
end = get_wall_time();

printf("wall time of cuda_conv2d, input size = %d, stride = %d: %e\n", input_height, stride, end - begin);

```

(4) im2col 二维卷积

im2col 的原理是通过将输入图像按列或按行重排，从而将卷积操作转化为矩阵相乘，矩阵乘无论是在 CPU 还是 GPU 上都有专门优化加速到极致的库可以直接调用，从而加速卷积运算；另外，im2col 操作对于 `batch_size` 较多时，可以充分利用 GPU 内存，将数据组织为连续的内存，减少 cache miss，CUDA 可以直接一次计算整个 batch。

使用 im2col 的 CUDA 二维卷积函数：

- 仍然采用 valid padding，首先将输入的图像经过 `im2col` 函数处理得到 `im_col`，再调用之前的 CUDA 矩阵乘函数计算卷积结果。

```c
void cuda_conv2d(int *input, int *output, int *kernel, int input_height, int input_width, int kernel_height, int kernel_width, int block_size, int stride)
{
  int output_height = (input_height - kernel_height) / stride + 1;
  int output_width = (input_width - kernel_width) / stride + 1;

  int im_col_height = kernel_height * kernel_width;
  int im_col_width = output_height * output_width;

  int *im_col = (int *)malloc(sizeof(int) * im_col_height * im_col_width);
  im2col(input, im_col, input_height, input_width, output_height, output_width, kernel_height, kernel_width, stride);

  #ifdef DEBUG
    printf("im2col:\n");
    print_matrix(im_col, kernel_height * kernel_width, output_height * output_width);
  #endif

  matrix_mul_cuda(kernel, im_col, output, 1, kernel_height * kernel_width, output_height * output_width, block_size);

  free(im_col);
}
```

im2col 函数：

- 以步长 `stride` 遍历输入的图像 `input`，将卷积核覆盖的部分按列重排。
- `im_col` 的行数为 `kernel_height * kernel_width`，即卷积核的大小；`im_col` 的列数是 `output_height * output_width`，即输出结果的大小。

```cpp
__host__ void im2col(int *im, int *im_col, int input_height, int input_width, int output_height, int output_width, int kernel_height, int kernel_width, int stride)
{
  int im_col_height = kernel_height * kernel_width;
  int im_col_width = output_height * output_width;

  // output_count is the col index of im_col
  int output_count = 0;

  // valid padding, drop the rest cols and rows
  for (int i = 0; i + kernel_height - 1 < input_height; i += stride)
  {
    for (int j = 0; j + kernel_width - 1 < input_width; j += stride)
    {
      for (int i_k = 0; i_k < kernel_height; i_k++)
      {
        for (int j_k = 0; j_k < kernel_width; j_k++)
        {
          int row = i + i_k;
          int col = j + j_k;
          im_col[(i_k * kernel_width + j_k) * im_col_width + output_count] = im[row * input_width + col];
        }
      }
      output_count += 1;
    }
  }
}
```

(5) cuDNN 二维卷积

cuDNN 二维卷积：

- 记录两个时间，一个包括 cuDNN 句柄、描述符、工作空间、算法所有的准备时间和销毁时间；另一个只记录 cuDNN 执行卷积操作的时间。

这里的代码过于冗长、大部分是重复性的工作，不再赘述。

```
double begin_1, begin_2, end_1, end_2;

begin_1 = get_wall_time();

cudnnHandle_t cudnn_handle;
cudnnCreate(&cudnn_handle);

cudnnTensorDescriptor_t input_descriptor;
cudnnCreateTensorDescriptor(&input_descriptor);
cudnnSetTensor4dDescriptor(input_descriptor, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, input_height, input_width);
float *input_d = NULL;
cudaMalloc((void **)&input_d, sizeof(float) * input_height * input_width);
cudaMemcpy(input_d, input, sizeof(float) * input_height * input_width, cudaMemcpyHostToDevice);

cudnnFilterDescriptor_t kernel_descriptor;
cudnnCreateFilterDescriptor(&kernel_descriptor);
cudnnSetFilter4dDescriptor(kernel_descriptor, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 1, 1, kernel_height, kernel_width);
float *kernel_d = NULL;
cudaMalloc((void **)&kernel_d, sizeof(float) * kernel_height * kernel_width);
cudaMemcpy(kernel_d, kernel, sizeof(float) * kernel_height * kernel_width, cudaMemcpyHostToDevice);

cudnnConvolutionDescriptor_t conv_descriptor;
cudnnCreateConvolutionDescriptor(&conv_descriptor);
cudnnSetConvolution2dDescriptor(conv_descriptor, 0, 0, stride, stride, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT);  // valid padding, dilation = 1 ?

cudnnGetConvolution2dForwardOutputDim(conv_descriptor, input_descriptor, kernel_descriptor, &output_n, &output_channel, &output_height, &output_width);

#ifdef DEBUG
  printf("output_n = %d\n", output_n);
  printf("output_channel = %d\n", output_channel);
  printf("output_height = %d\n", output_height);
  printf("output_width = %d\n\n", output_width);
#endif

cudnnTensorDescriptor_t output_descriptor;
cudnnCreateTensorDescriptor(&output_descriptor);
cudnnSetTensor4dDescriptor(output_descriptor, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, output_n, output_channel, output_height, output_width);
float *output_d = NULL;
cudaMalloc((void **)&output_d, sizeof(float) * output_n * output_channel * output_height * output_width);

cudnnConvolutionFwdAlgo_t alg;
cudnnGetConvolutionForwardAlgorithm(cudnn_handle, input_descriptor, kernel_descriptor, conv_descriptor, output_descriptor, \
                                    CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &alg);

// workspace size && allocate memory
size_t workspace_size = 0;
cudnnGetConvolutionForwardWorkspaceSize(cudnn_handle, input_descriptor, kernel_descriptor, conv_descriptor, output_descriptor, \
                                        alg, &workspace_size);
void * workspace = nullptr;
cudaMalloc(&workspace, workspace_size);

// convolution
float alpha = 1.0f;
float beta = 0.0f;
begin_2 = get_wall_time();
cudnnConvolutionForward(cudnn_handle,
                        &alpha, input_descriptor, input_d,
                        kernel_descriptor, kernel_d,
                        conv_descriptor, alg,
                        workspace, workspace_size,
                        &beta, output_descriptor, output_d);
end_2 = get_wall_time();

cudaMemcpy(output, output_d, sizeof(float) * output_n * output_channel * output_height * output_width, cudaMemcpyDeviceToHost);

// free and destroy
cudaFree(input_d);
cudaFree(output_d);
cudaFree(kernel_d);
cudnnDestroyTensorDescriptor(input_descriptor);
cudnnDestroyTensorDescriptor(output_descriptor);
cudnnDestroyFilterDescriptor(kernel_descriptor);
cudnnDestroyConvolutionDescriptor(conv_descriptor);
cudaFree(workspace);
cudnnDestroy(cudnn_handle);

end_1 = get_wall_time();

printf("wall time of cudnn_conv2d, input size = %d, stride = %d: %e\n", input_height, stride, end_2 - begin_2);
printf("wall time of cudnn_conv2d, input size = %d, stride = %d (including handle and descriptor): %e\n", input_height, stride, end_1 - begin_1);
```

(四) 实验结果

(1) CUDA 矩阵乘法

首先保证计算结果的正确性。取消 `#define DEBUG` 的注释，编译运行。设置矩阵规模为 777，block size 为 16，可以看到计算结果与 CPU 矩阵乘法结果的误差为 0。

```
jovyan@jupyter-lixk28:~/lab/cuda-mm$ ./cuda_mm 777 777 777 16
m = 777
n = 777
k = 777
block size = 16
dim_grid(49), dim_block(16)
wall time of gemm_cuda, matrix size 777, block size 16: 0.35962
wall time of gemm_cpu: 3.01785
Error = 0.00000
```

下面是测试的运行时间，矩阵规模从 512 到 8192，对于每种规模的矩阵，block size 32 到 512。

```
jovyan@jupyter-lixk28:~/lab/mat-mul-cuda$ make
nvcc matmul.cu -o matmul_cuda
jovyan@jupyter-lixk28:~/lab/mat-mul-cuda$ make test
for size in 512 1024 2048 4096 8192 ; do \
        for block_size in 32 64 128 256 512 ; do \
                ./matmul_cuda $size $size $size $block_size ; \
        done ; \
done
wall time of gemm_cuda, matrix size 512, block size 32: 0.47464
wall time of gemm_cuda, matrix size 512, block size 64: 0.43110
wall time of gemm_cuda, matrix size 512, block size 128: 0.34714
wall time of gemm_cuda, matrix size 512, block size 256: 0.62513
wall time of gemm_cuda, matrix size 512, block size 512: 1.16624
^Cmake: *** [Makefile:13: test] Interrupt

jovyan@jupyter-lixk28:~/lab/mat-mul-cuda$ make test
for size in 512 1024 2048 4096 8192 ; do \
        for block_size in 32 64 128 256 512 ; do \
                ./matmul_cuda $size $size $size $block_size ; \
        done ; \
done
wall time of gemm_cuda, matrix size 512, block size 32: 0.46039
wall time of gemm_cuda, matrix size 512, block size 64: 0.37763
wall time of gemm_cuda, matrix size 512, block size 128: 0.54041
wall time of gemm_cuda, matrix size 512, block size 256: 0.81292
wall time of gemm_cuda, matrix size 512, block size 512: 1.16355
wall time of gemm_cuda, matrix size 1024, block size 32: 0.73229
wall time of gemm_cuda, matrix size 1024, block size 64: 1.19107
wall time of gemm_cuda, matrix size 1024, block size 128: 1.31830
wall time of gemm_cuda, matrix size 1024, block size 256: 1.73578
wall time of gemm_cuda, matrix size 1024, block size 512: 2.75457
wall time of gemm_cuda, matrix size 2048, block size 32: 4.92529
wall time of gemm_cuda, matrix size 2048, block size 64: 5.14425
wall time of gemm_cuda, matrix size 2048, block size 128: 5.77179
wall time of gemm_cuda, matrix size 2048, block size 256: 6.59279
wall time of gemm_cuda, matrix size 2048, block size 512: 10.27546
wall time of gemm_cuda, matrix size 4096, block size 32: 41.85747
wall time of gemm_cuda, matrix size 4096, block size 64: 41.84869
wall time of gemm_cuda, matrix size 4096, block size 128: 41.77652
wall time of gemm_cuda, matrix size 4096, block size 256: 34.70924
wall time of gemm_cuda, matrix size 4096, block size 512: 30.46254
wall time of gemm_cuda, matrix size 8192, block size 32: 229.56521
wall time of gemm_cuda, matrix size 8192, block size 64: 230.14497
wall time of gemm_cuda, matrix size 8192, block size 128: 240.16955
wall time of gemm_cuda, matrix size 8192, block size 256: 244.97605
wall time of gemm_cuda, matrix size 8192, block size 512: 195.06577
jovyan@jupyter-lixk28:~/lab/mat-mul-cuda$
```

可以看到，我们对 grid 和 block 进行一维划分，在线程数一定的情况下，当矩阵规模较小时 (<= 2048)，随着线程块大小增大，运行时间增加；而在矩阵规模较大时 (>= 4096)，block size 增大会减少运行所需时间。

(2) CUBLAS 矩阵乘法

设置 block size 为 32，即采用最大规模 (1024) 的线程块，矩阵规模从 512 增加到 8192。

```
jovyan@jupyter-lixk28:~/lab/cublas$ make clean && make
rm -rf ./cublas_mm
nvcc cublas_mm.cu -o cublas_mm -lcublas
jovyan@jupyter-lixk28:~/lab/cublas$ make test
for size in 512 1024 2048 4096 8192 ; do \
        ./cublas_mm $size $size $size 32 ; \
done
wall time of gemm_cuda, matrix size 512, block size 32: 0.36292
wall time of cublas_gemm, matrix size 512, block size 32: 0.23264
error: 0.00000
wall time of gemm_cuda, matrix size 1024, block size 32: 0.21014
wall time of cublas_gemm, matrix size 1024, block size 32: 0.22867
error: 0.00000
wall time of gemm_cuda, matrix size 2048, block size 32: 0.24455
wall time of cublas_gemm, matrix size 2048, block size 32: 0.24749
error: 0.00000
wall time of gemm_cuda, matrix size 4096, block size 32: 0.40021
wall time of cublas_gemm, matrix size 4096, block size 32: 0.35210
error: 0.00000
wall time of gemm_cuda, matrix size 8192, block size 32: 1.32517
wall time of cublas_gemm, matrix size 8192, block size 32: 0.82638
error: 0.00000
```

可以看出，在矩阵规模较小时 (<= 2048)，我们的 CUDA 矩阵乘法运行时间比 `cublasDgemm` 要快，但在矩阵规模较大时 (>= 4096)，我们的 CUDA 矩阵乘法要明显慢于 `cublasDgemm`。

一个改进方法是，通过 `__shared__` 关键字来定义一块共享内存，同一个线程块内的所有线程共享该内存区域。并使用 `__syncthreads()` 进行线程同步。每个线程块定义两块共享内存，一块用于缓存该线程块需要用到的 A 的部分，一块用于缓存需要用到的 B 的部分，这样将访存操作从多次的访问显存变为一次访问显存加上其余的访问线程块共享内存，减少了大量访存时间（because shared memory is much faster than global memory)。

(3) CUDA 二维卷积

首先测试卷积结果的正确性。设置输入大小为 5x5，stride 分别是 1 和 2，block size 为 32。

可以看到，卷积结果正确（valid padding policy）。

```
jovyan@jupyter-lixk28:~/lab/cuda-conv$ ./cuda_conv2d 5 5 1 32
input:
1       3       2       3       0
2       0       4       4       2
4       2       1       3       1
3       4       1       2       4
0       4       0       2       1

kernel:
0       4       0
1       0       1
0       0       1

dim_grid(1, 1)
dim_block(3, 3)
wall time of cuda_conv2d, input size = 5, stride = 1: 2.758231e-01
output:
19      15      19
6       23      22
12      12      18
```

```
jovyan@jupyter-lixk28:~/lab/cuda-conv$ ./cuda_conv2d 5 5 2 32
input:
1       3       2       3       0
2       0       4       4       2
4       2       1       3       1
3       4       1       2       4
0       4       0       2       1

kernel:
0       4       0
1       0       1
0       0       1

dim_grid(1, 1)
dim_block(2, 2)
wall time of cuda_conv2d, input size = 5, stride = 2: 2.157590e-01
output:
19      19
12      18
```

设置输入规模从 256 到 4096，步长分别为 1、2、3，运行时间如下。

```
jovyan@jupyter-lixk28:~/lab/cuda-conv$ make
nvcc cuda_conv2d.cu -o cuda_conv2d
jovyan@jupyter-lixk28:~/lab/cuda-conv$ make test
for size in 256 512 1024 2048 4096 ; do \
        for stride in 1 2 3 ; do \
                ./cuda_conv2d $size $size $stride 32 ; \
        done ; \
done
wall time of cuda_conv2d, input size = 256, stride = 1: 3.377650e-01
wall time of cuda_conv2d, input size = 256, stride = 2: 2.254920e-01
wall time of cuda_conv2d, input size = 256, stride = 3: 2.120621e-01
wall time of cuda_conv2d, input size = 512, stride = 1: 2.218409e-01
wall time of cuda_conv2d, input size = 512, stride = 2: 2.172799e-01
wall time of cuda_conv2d, input size = 512, stride = 3: 2.057059e-01
wall time of cuda_conv2d, input size = 1024, stride = 1: 3.420191e-01
wall time of cuda_conv2d, input size = 1024, stride = 2: 2.051260e-01
wall time of cuda_conv2d, input size = 1024, stride = 3: 2.147241e-01
wall time of cuda_conv2d, input size = 2048, stride = 1: 2.205830e-01
wall time of cuda_conv2d, input size = 2048, stride = 2: 2.162418e-01
wall time of cuda_conv2d, input size = 2048, stride = 3: 2.106080e-01
wall time of cuda_conv2d, input size = 4096, stride = 1: 2.657571e-01
wall time of cuda_conv2d, input size = 4096, stride = 2: 2.402761e-01
wall time of cuda_conv2d, input size = 4096, stride = 3: 2.274489e-01
```

(4) im2col 二维卷积

首先测试 im2col 操作和卷积结果的正确性。设置输入大小为 5x5，步长为 1 和 2。

可以看到，im2col 的重排结果正确，最后的卷积结果也正确（valid padding）。

```
jovyan@jupyter-lixk28:~/lab/im2col$ ./im2col_conv2d 5 5 1 32
input:
1       3       2       3       0
2       0       4       4       2
4       2       1       3       1
3       4       1       2       4
0       4       0       2       1

kernel:
0       4       0
1       0       1
0       0       1

im2col:
1       3       2       2       0       4       4       2       1
3       2       3       0       4       4       2       1       3
2       3       0       4       4       2       1       3       1
2       0       4       4       2       1       3       4       1
0       4       4       2       1       3       4       1       2
4       4       2       1       3       1       1       2       4
4       2       1       3       4       1       0       4       0
2       1       3       4       1       2       4       0       2
1       3       1       1       2       4       0       2       1

dim_grid(1, 1), dim_block(32, 32)
wall time of im2col_conv2d, input size = 5, stride = 1: 2.927151e-01
output:
19      15      19
6       23      22
12      12      18
```

```
jovyan@jupyter-lixk28:~/lab/im2col$ ./im2col_conv2d 5 5 2 32
input:
1       3       2       3       0
2       0       4       4       2
4       2       1       3       1
3       4       1       2       4
0       4       0       2       1

kernel:
0       4       0
1       0       1
0       0       1

im2col:
1       2       4       1
3       3       2       3
2       0       1       1
2       4       3       1
0       4       4       2
4       2       1       4
4       1       0       0
2       3       4       2
1       1       0       1

dim_grid(1, 1), dim_block(32, 32)
wall time of im2col_conv2d, input size = 5, stride = 2: 2.183371e-01
output:
19      19
12      18
```

设置输入规模从 256 到 4096，步长分别为 1、2、3，运行时间如下。

```
jovyan@jupyter-lixk28:~/lab/im2col$ make
nvcc im2col_conv2d.cu -o im2col_conv2d
im2col_conv2d.cu(78): warning: variable "im_col_height" was declared but never referenced

jovyan@jupyter-lixk28:~/lab/im2col$ make test
for size in 256 512 1024 2048 4096 ; do \
        for stride in 1 2 3 ; do \
                ./im2col_conv2d $size $size $stride 32 ; \
        done ; \
done
wall time of im2col_conv2d, input size = 256, stride = 1: 2.198699e-01
wall time of im2col_conv2d, input size = 256, stride = 2: 4.045050e-01
wall time of im2col_conv2d, input size = 256, stride = 3: 2.049370e-01
wall time of im2col_conv2d, input size = 512, stride = 1: 2.749181e-01
wall time of im2col_conv2d, input size = 512, stride = 2: 2.078230e-01
wall time of im2col_conv2d, input size = 512, stride = 3: 2.929571e-01
wall time of im2col_conv2d, input size = 1024, stride = 1: 3.088410e-01
wall time of im2col_conv2d, input size = 1024, stride = 2: 4.023750e-01
wall time of im2col_conv2d, input size = 1024, stride = 3: 2.578530e-01
wall time of im2col_conv2d, input size = 2048, stride = 1: 4.872921e-01
wall time of im2col_conv2d, input size = 2048, stride = 2: 3.038929e-01
wall time of im2col_conv2d, input size = 2048, stride = 3: 4.266109e-01
wall time of im2col_conv2d, input size = 4096, stride = 1: 1.341374e+00
wall time of im2col_conv2d, input size = 4096, stride = 2: 4.836798e-01
wall time of im2col_conv2d, input size = 4096, stride = 3: 1.440797e+00
```

可以看到，运行时间明显比上面普通的二维卷积要慢，这是当矩阵规模增大时，im2col 的重排时间开销变大，而我们测试时的 batch size 为 1（即只有一张图片），主要的时间都花在 im2col 上了。

(5) cuDNN 二维卷积

输入规模从 256 到 4096，步长从 1 到 3。分别记录两个时间，一个包括句柄、描述符等创建销毁的时间，另一个只记录做卷积运算的时间。

可以看出，卷积运算的速度非常快，但是句柄、描述符等创建销毁的开销非常大，导致每次完整的卷积操作都要 1 秒多。

```
jovyan@jupyter-lixk28:~/lab/cudnn$ make clean && make
rm -rf ./cudnn_conv2d
export LD_LIBRARY_PATH=/opt/conda/lib && nvcc -I/opt/conda/include -L/opt/conda/lib -o cudnn_conv2d cudnn_conv2d.cu -lcudnn
jovyan@jupyter-lixk28:~/lab/cudnn$ make test
for size in 256 512 1024 2048 4096 ; do \
        for stride in 1 2 3 ; do \
                ./cudnn_conv2d $size $size $stride 32 ; \
        done ; \
done
wall time of cudnn_conv2d, input size = 256, stride = 1: 7.915497e-05
wall time of cudnn_conv2d, input size = 256, stride = 1 (including handle and descriptor): 1.427963e+00
wall time of cudnn_conv2d, input size = 256, stride = 2: 3.600121e-05
wall time of cudnn_conv2d, input size = 256, stride = 2 (including handle and descriptor): 1.377608e+00
wall time of cudnn_conv2d, input size = 256, stride = 3: 3.910065e-05
wall time of cudnn_conv2d, input size = 256, stride = 3 (including handle and descriptor): 1.290022e+00
wall time of cudnn_conv2d, input size = 512, stride = 1: 1.168251e-04
wall time of cudnn_conv2d, input size = 512, stride = 1 (including handle and descriptor): 1.379360e+00
wall time of cudnn_conv2d, input size = 512, stride = 2: 5.984306e-05
wall time of cudnn_conv2d, input size = 512, stride = 2 (including handle and descriptor): 1.399129e+00
wall time of cudnn_conv2d, input size = 512, stride = 3: 4.100800e-05
wall time of cudnn_conv2d, input size = 512, stride = 3 (including handle and descriptor): 1.396500e+00
wall time of cudnn_conv2d, input size = 1024, stride = 1: 3.790855e-05
wall time of cudnn_conv2d, input size = 1024, stride = 1 (including handle and descriptor): 1.278091e+00
wall time of cudnn_conv2d, input size = 1024, stride = 2: 4.291534e-05
wall time of cudnn_conv2d, input size = 1024, stride = 2 (including handle and descriptor): 1.375414e+00
wall time of cudnn_conv2d, input size = 1024, stride = 3: 3.695488e-05
wall time of cudnn_conv2d, input size = 1024, stride = 3 (including handle and descriptor): 1.390517e+00
wall time of cudnn_conv2d, input size = 2048, stride = 1: 4.601479e-05
wall time of cudnn_conv2d, input size = 2048, stride = 1 (including handle and descriptor): 1.264238e+00
wall time of cudnn_conv2d, input size = 2048, stride = 2: 4.601479e-05
wall time of cudnn_conv2d, input size = 2048, stride = 2 (including handle and descriptor): 1.435897e+00
wall time of cudnn_conv2d, input size = 2048, stride = 3: 4.291534e-05
wall time of cudnn_conv2d, input size = 2048, stride = 3 (including handle and descriptor): 1.384110e+00
wall time of cudnn_conv2d, input size = 4096, stride = 1: 4.100800e-05
wall time of cudnn_conv2d, input size = 4096, stride = 1 (including handle and descriptor): 1.414320e+00
wall time of cudnn_conv2d, input size = 4096, stride = 2: 4.410744e-05
wall time of cudnn_conv2d, input size = 4096, stride = 2 (including handle and descriptor): 1.309677e+00
wall time of cudnn_conv2d, input size = 4096, stride = 3: 7.009506e-05
wall time of cudnn_conv2d, input size = 4096, stride = 3 (including handle and descriptor): 1.321242e+00
```

改进自己的 im2col + gemm 卷积：

- 使用共享内存：
  - gemm 可以使用共享内存进行优化，加速矩阵乘
- 优化浮点指令：
  - 使用 CUDA 提供的浮点向量指令，一次存取多个浮点数，从而减少访存次数
- im2col：
  - im2col 可以不用 CPU 完成，本身也非常容易并行，可以用封装一个 im2col 核函数，加速 im2col 过程

(五) 实验感想

这次实验用 CUDA 实现了矩阵乘法，熟悉了基本的 CUDA 编程，CUDA gemm 速度非常快，让我真实体会到了 GPU 在向量化计算上的威力。

另外，学习了卷积的操作过程和原理，用 CUDA 自己编写了二维卷积，使用的是 valid padding，same padding 我没有想到优雅的解决办法，一个普通的想法是将原有图像复制到新的加了 padding 的矩阵中，但是这么做我觉得开销非常大，所以采用 valid padding，其实就是不 padding，舍弃掉卷积核无法卷积的部分。

还有，im2col 是卷积操作中常用的技术，我在 github 上找到了 caffe 的源码，caffe 中就是用 im2col 来做卷积运算，而且 caffe 是将 im2col 作为核函数的，这样加速了 im2col 的过程。

调用和使用了 NVIDIA 深度学习和线性代数库函数，让我了解到已经有很成熟的库可以帮助我们做计算。

最后，虽然这是 HPC 最后一次实验，但这次实验内容非常有启发性，我在 github 上看到了一些深度学习框架比如 caffe，主要是卷积神经网络 CNN 的框架，因为 RNN 一般用 CPU 进行训练。

由于这学期太贪心选了太多可，下学期课比较少，我计划在寒假以及下学期的空余时间，自己从零开始实现一个 CNN 框架，进行一定程度的优化（比如针对嵌入式设备、移动设备使用 ARM 指令优化），在 MINIST 数据集上进行测试、与其他现有开源框架进行性能比较，并实际部署到硬件上（如树莓派），以后我就想做 HPC 方面的研究，为上层应用提供接口、优化应用性能。

附录：参考资料

- https://www.cnblogs.com/skyfsm/p/9673960.html
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf
- https://www.cnblogs.com/feng9exe/p/6722990.html
- https://stackoverflow.com/questions/7989039/use-of-cudamalloc-why-the-double-pointer
- https://stackoverflow.com/questions/14595750/transpose-matrix-multiplication-in-cublas-howto
- https://www.cnblogs.com/cuancuancuanhao/p/7763256.html
- https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm
- https://stackoverflow.com/questions/56043539/cublassgemm-row-major-multiplication
- http://www.goldsborough.me/cuda/ml/cudnn/c++/2017/10/01/14-37-23-convolutions_with_cudnn/
- https://blog.csdn.net/ice__snow/article/details/79699388
- https://github.com/BVLC/caffe/blob/master/src/caffe/util/im2col.cu
- https://www.cnblogs.com/combfish/p/9259362.html
- https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html
- https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnConvolutionForward