Lab9 - CUDA矩阵转置

实验要求

1.CUDA Hello World

本实验为CUDA入门练习，由多个线程并行输出“Hello World！”。

输入：三个整数n, m, k，其取值范围为[1,32]

问题描述：创建n个线程块，每个线程块的维度为，$m \times k$每个线程均输出线程块编号、二维块内线程编号及Hello World！（如，“Hello World from Thread (1, 2) in Block 10!”）。主线程输出“Hello World from the host!”。

要求：完成上述内容，观察输出，并回答线程输出顺序是否有规律？

2.CUDA矩阵转置

使用CUDA对矩阵进行并行转置。

输入：整数n，其取值范围均为[512, 2048]

问题描述：随机生成$n \times n$的矩阵A，对其进行转置得到A^T。转置矩阵中第i行j列上的元素为原矩阵中j行i列元素，即$A_{ij}^T = A_{ji}$。

输出：矩阵A及其转置矩阵A^T，及计算所消耗的时间t。

要求：使用CUDA实现并行矩阵转置，分析不同线程块大小，矩阵规模，访存方式（全局内存访问，共享内存访问），任务/数据划分和映射方式，对程序性能的影响。

实验过程

一、CUDA Hello World

直接展示代码

```
#include <stdio.h>

__global__ void helloFromGPU(){
    printf("Hello World from Thread (%d, %d) in Block
%d!\n",threadIdx.x,threadIdx.y,blockIdx.x);
}
int main(int argc, char** argv) {
    int m,n,k;
    printf("Enter values for n, m, k(1,32):\n");
    scanf("%d%d%d",&n,&m,&k);
    dim3 block(m,k);
    dim3 grid(n,1);
    printf("Hello World from the host!\n");
    helloFromGPU<<<grid, block>>>();
    cudaDeviceReset();
    return 0;
}
```

编译运行：

```
nvcc helloworld.cu -o hello
./hello
```

运行结果：观察线程输出顺序，可看出同一个块的线程通常聚集输出，且输出顺序优先遍历`threadIdx.x`，也就是行顺序优先输出。

```
jovyan@jupyter-21307347:~/parallel$ ./hello
Enter values for n, m, k(1,32):
3 2 2
Hello World from the host!
Hello World from Thread (0, 0) in Block 2!
Hello World from Thread (1, 0) in Block 2!
Hello World from Thread (0, 1) in Block 2!
Hello World from Thread (1, 1) in Block 2!
Hello World from Thread (0, 0) in Block 0!
Hello World from Thread (1, 0) in Block 0!
Hello World from Thread (0, 1) in Block 0!
Hello World from Thread (1, 1) in Block 0!
Hello World from Thread (0, 0) in Block 1!
Hello World from Thread (1, 0) in Block 1!
Hello World from Thread (0, 1) in Block 1!
Hello World from Thread (1, 1) in Block 1!
```

二、CUDA矩阵转置

1.实验思路

考虑访存方式、任务/数据划分的不同，设计三个版本的矩阵转置，第一种只使用全局内存计算，第二种开始引入共享内存运算，之后尝试对数据间隔划分到不同线程，第三种则是每个线程划分得到一行中连续的矩阵元素。

2.代码实现

初始化矩阵变量

```
int main(int argc, char** argv) {
    int n;
    constexpr int block_dim = 16;
    double* A,* A_dev,* A_T_dev;
    printf("Enter values for n:\n");
    scanf("%d",&n);
    // CPU 分配矩阵内存
    A = (double*)malloc(n * n * sizeof(double));
    initialize(A,n);    // 随机初始化 A
    // 打印初始矩阵
    ...
    // GPU 分配矩阵内存
    cudaMalloc((void**)&A_dev,n * n * sizeof(double));
    cudaMalloc((void**)&A_T_dev,n * n * sizeof(double));
    cudaMemcpy(A_dev,A,n * n * sizeof(double),cudaMemcpyHostToDevice);
    cudaMemcpy(A_T_dev,A,n * n * sizeof(double),cudaMemcpyHostToDevice);
```

```
    //
    dim3 block(block_dim,block_dim);
    dim3 grid((n + block_dim - 1) / block_dim,(n + block_dim - 1) / block_dim);
    mat_transpose<<<grid, block>>>(A_dev,A_T_dev,n);
    // 读取GPU内存回到矩阵A
    cudaMemcpy(A,A_T_dev,n * n * sizeof(double),cudaMemcpyDeviceToHost);
    // 打印转置后矩阵
    ...
    // 释放内存
    cudaFree(A_dev);
    cudaFree(A_T_dev);
    free(A);
    cudaDeviceReset();
    return 0;
}
```

1. 直接通过全局内存完成矩阵转置

```
__global__ void mat_transpose(double *A, double *A_T, int n){
    // 在整个矩阵中的坐标
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (y < n && x < n) {
        A_T[x * n + y] = A[y * n + x];
    }
}
```

2. 使用共享内存完成矩阵转置

```
template <int BLOCK_DIM>    // 使用模板参数传入block大小
__global__ void mat_transpose_share(double *A, double *A_T, int n){
    const int bx = blockIdx.x, by = blockIdx.y;
    const int tx = threadIdx.x, ty = threadIdx.y;
    // 数据原位置
    int x = bx * BLOCK_DIM + tx;
    int y = by * BLOCK_DIM + ty;
    // 定义共享内存
    __shared__ double sdata[BLOCK_DIM][BLOCK_DIM+1];// +1避免 bank conflicts
    if (y < n && x < n) {
        sdata[ty][tx] = A[y * n + x];
    }
    __syncthreads();
    // 数据新位置, block起始点变为[by*BLOCK_DIM,bx*BLOCK_DIM]
    x = by * BLOCK_DIM + tx;
    y = bx * BLOCK_DIM + ty;
    if (y < n && x < n) {
        A_T[y * n + x] = sdata[tx][ty];
    }
}
```

3. 不同的数据划分方式

前面的代码中，一个线程块中每个线程只负责一个矩阵元素的转置，下面将在使用共享内存基础上改进，使一个线程能够划分到多个矩阵元素，利用内存访问的空间局部性

a) 取同一行中相同间隔下的矩阵元素划分给一个线程

```
template <int BLOCK_DIM,int NUMS_THREAD>
__global__ void mat_transpose_share(double *A, double *A_T, int n){
    const int bx = blockIdx.x, by = blockIdx.y;
    const int tx = threadIdx.x, ty = threadIdx.y;

    __shared__ double sdata[BLOCK_DIM][BLOCK_DIM+1];

    int x = bx * BLOCK_DIM + tx;
    int y = by * BLOCK_DIM + ty;

    int stride = BLOCK_DIM/NUMS_THREAD;

    if(x < n){
        #pragma unroll
        for (int y_off = 0; y_off < BLOCK_DIM; y_off += stride) {
            if (y + y_off < n) {
                sdata[ty + y_off][tx] = A[(y + y_off) * n + x];
            }
        }
    }
    __syncthreads();
    x = by * BLOCK_DIM + tx;
    y = bx * BLOCK_DIM + ty;
    if (x < n) {
        for (int y_off = 0; y_off < BLOCK_DIM; y_off += stride) {
            if (y + y_off < n) {
                A_T[(y + y_off) * n + x] = sdata[tx][ty + y_off];
            }
        }
    }
}
```

b) 取同一行中连续分布的矩阵元素划分给一个线程

```
template <int BLOCK_DIM,int NUMS_THREAD>
__global__ void mat_transpose_share_v1(double *A, double *A_T, int n){
    const int bx = blockIdx.x, by = blockIdx.y;
    const int tx = threadIdx.x, ty = threadIdx.y;

    __shared__ double sdata[BLOCK_DIM][BLOCK_DIM+1];

    int x = bx * BLOCK_DIM + tx;
    int y = by * BLOCK_DIM + ty * NUMS_THREAD;

    if(x < n){
        #pragma unroll
        for (int y_off = 0; y_off < NUMS_THREAD; y_off += 1) {
            if (y + y_off < n) {
                sdata[ty + y_off][tx] = A[(y + y_off) * n + x];
            }
```

```
        }
    }

    __syncthreads();

    x = by * BLOCK_DIM + tx * NUMS_THREAD;
    y = bx * BLOCK_DIM + ty;
    if (x < n) {
        for (int y_off = 0; y_off < NUMS_THREAD; y_off += 1) {
            if (y + y_off < n) {
                A_T[(y + y_off) * n + x] = sdata[tx][ty + y_off];
            }
        }
    }
}
```

3. 运行结果

编译运行指令：

```
nvcc matrixT.cu -o mat
./mat
```

正常运行结果如下：图中仅展示矩阵的左上角36个元素，可看出矩阵正常转置

```
jovyan@jupyter-21307347:~/parallel$ nvcc matrixT.cu -o mat
jovyan@jupyter-21307347:~/parallel$ ./mat
Enter values for n[512, 2048]:
512
 Top left corner of matrix A:
         83          86          77          15          93          35
         77          34          90          26          24          57
         51          64          90          63          91          72
         85          97           0          97           8          29
         51          80          20          97           0          88
         46          39          84          82          50          72
Time taken: 20.074100 us
 Top left corner of matrix A_T:
         83          77          51          85          51          46
         86          34          64          97          80          39
         77          90          90           0          20          84
         15          26          63          97          97          82
         93          24          91           8           0          50
         35          57          72          29          88          72
```

性能分析

首先比较直接使用全局内存情况下，线程块大小和矩阵规模对平均运行时间的影响，运行时间以us为单位

线程块大小\矩阵规模	512	1024	2048
8	20.074	81.371	328.316
16	25.026	102.743	406.609

线程块大小\矩阵规模	512	1024	2048
32	34.183	129.247	483.369

使用共享内存情况下，记录运行时间如下：

线程块大小\矩阵规模	512	1024	2048
8	18.089	80.527	326.672
16	14.329	78.441	313.401
32	16.916	73.127	287.478

分析上面两表可得，矩阵规模越大，运行时间越长，当只是用全局内存时，运行时间随着线程块大小增大，这一点未能分析到原因，使用共享内存后运行时间随着线程块大小而减小。这两种访存方式在线程块较大时的耗时差异不大，而在线程块较小时，使用共享内存访存能够在更短时间内完成矩阵转置。

而对于新的数据划分类型，使用矩阵规模为2048大小来观察不同线程块大小和同一线程划分数据量对平均运行时间(us)的对比如下

每个线程间隔取矩阵元素：

线程块大小\线程划分数量	1	2	4	8	16	32
8	326.49	327.33	326.94	330.31	*	*
16	313.49	323.52	333.44	330.16	330.78	*
32	287.67	297.41	304.71	304.42	301.57	296.80

每个线程连续取矩阵元素：

线程块大小\线程划分数量	1	2	4	8	16	32
8	326.56	348.91	352.08	462.59	*	*
16	313.61	302.32	271.85	272.24	1924.91	*
32	287.69	282.69	242.38	245.13	943.17	2592.00

可看出当线程划分到间隔元素时，运行时间没有得到减少反而有所增加，而当连续取矩阵行元素时，当线程划分数量在4及以内时，运行时间有所减少，性能得到提升。

总的来说，当使用共享内存进行矩阵转置时，在正常范围内，当使用线程块越大，每个线程连续划分行的大小为4时，矩阵转置运算的性能越好。