Lab10 - CUDA并行矩阵乘法

实验要求

CUDA实现并行通用矩阵乘法，并通过实验分析不同线程块大小，访存方式、数据/任务划分方式对并行性能的影响。

输入：m, n, k三个整数，每个整数的取值范围均为[128, 2048]

问题描述：随机生成$m \times n$的矩阵A及$n \times k$的矩阵B，并对这两个矩阵进行矩阵乘法运算，得到矩阵C.

输出：A, B, C三个矩阵，及矩阵计算所消耗的时间t。

要求：使用CUDA实现并行矩阵乘法，分析不同线程块大小，矩阵规模，访存方式，任务/数据划分方式，对程序性能的影响。

实验过程

1.实验思路

每个线程负责输出矩阵一个元素的计算，固定线程块大小，根据矩阵规模调整块的个数。后续使用共享内存方式，一个线程块同样计算与块内线程同等数量的矩阵元素，因此使用共享内存存储矩阵A中计算块内元素所需要的行和矩阵B中计算块内元素所需要的列，但考虑共享内存的存储空间较小，根据矩阵规模的行列而变换使用大小是不合理的，因此仅使用与线程块同等大小的共享内存，对矩阵A、B的行列元素相乘分块计算后相加

2.代码实现

初始化矩阵变量

```
int main(int argc, char** argv) {
    srand(time(NULL));
    constexpr int BLOCK_DIM = 16;
    int m,n,k;
    double* A,* B,* C,* A_dev,* B_dev,* C_dev;
    double iStart,iElaps;    // 开始结束时间
    printf("Enter values for m,n,k[128, 2048]:");
    scanf("%d%d%d",&m,&n,&k);
    // 分配CPU内存,初始化A,B
    A = (double*)malloc(m * n * sizeof(double));
    B = (double*)malloc(n * k * sizeof(double));
    C = (double*)malloc(m * k * sizeof(double));
    initialize(A,m,n);    // 随机初始化 A
    initialize(B,n,k);    // 随机初始化 B
    printf(" Top left corner of matrix A: \n");
    show(A,6);  // 展示矩阵右上角的6*6元素
    printf(" Top left corner of matrix B: \n");
    show(B,6);
    // 分配GPU内存
    cudaMalloc((void**)&A_dev,m * n * sizeof(double));
    cudaMalloc((void**)&B_dev,k * n * sizeof(double));
    cudaMalloc((void**)&C_dev,m * k * sizeof(double));
    cudaMemcpy(A_dev,A,m * n * sizeof(double),cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(B_dev,B,k * n * sizeof(double),cudaMemcpyHostToDevice);
    cudaMemcpy(C_dev,C,m * k * sizeof(double),cudaMemcpyHostToDevice);
    dim3 block(BLOCK_DIM,BLOCK_DIM);
    dim3 grid(m/BLOCK_DIM,k/ BLOCK_DIM);
    // 开始
    iStart=cpuSecond();
    mat_multi<<<grid, block>>>(A_dev,B_dev,C_dev,m,n,k);
    // 结束
    cudaDeviceSynchronize();    //同步CPU和GPU
    iElaps=cpuSecond()-iStart;
    printf("Time taken: %f us\n",iElaps/test_times);

    cudaMemcpy(C,C_dev,n * m * sizeof(double),cudaMemcpyDeviceToHost);
    printf(" Top left corner of matrix C: \n");
    show(C,6);
    // 释放内存
    cudaFree(A_dev);
    cudaFree(B_dev);
    cudaFree(C_dev);
    free(A);
    free(B);
    free(C);
    cudaDeviceReset();
    return 0;
}
```

1. 直接通过全局内存

```
__global__ void mat_multi(double *A, double *B,double *C,int m,int n,int k)
{
    int threadId = (blockIdx.y * blockDim.y + threadIdx.y) * gridDim.x *
blockDim.x + blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId < m * k)
    {
        int row = threadId / k;
        int column = threadId % k;
        C[threadId] = 0;
        for (int i = 0; i < n; i++)
        {
            C[threadId] += A[row * n + i] * B[i * k + column];
        }
    }
}
```

2. 使用共享内存完成矩阵转置

```
template<int BLOCK_DIM>
__global__ void mat_multi_shared(double *A, double *B, double *C, int m, int n,
int k)
{
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;
```

```
    if ((thread_y + block_y * BLOCK_DIM) * n + block_x * BLOCK_DIM + thread_x >=
m * n)
        return;

    int begin_a = block_y * BLOCK_DIM * n;
    int end_a = begin_a + n;
    int step_a = BLOCK_DIM;

    int begin_b = block_x * BLOCK_DIM;
    int step_b = BLOCK_DIM * k;

    int sum = 0;

    for (int index_a = begin_a,index_b = begin_b;
        index_a < end_a; index_a += step_a, index_b += step_b)
    {
        __shared__ double Block_A[BLOCK_DIM][BLOCK_DIM];
        __shared__ double Block_B[BLOCK_DIM][BLOCK_DIM];
        Block_A[thread_y][thread_x] = A[index_a + thread_y * n + thread_x];
        Block_B[thread_y][thread_x] = B[index_b + thread_y * k + thread_x];

        __syncthreads();

        for (int j = 0; j < BLOCK_DIM; j++)
        {
            sum += Block_A[thread_y][j] * Block_B[j][thread_x];
        }

        __syncthreads();
    }
    C[begin_b + block_y * BLOCK_DIM * k + thread_y * k + thread_x] = sum;
}
```

3. 不同的数据划分方式

使用A的行元素时，使用共享内存读取时每个线程各读取一个矩阵元素，考虑空间局部性，将A中连续行元素划分到同一个线程，在方法2基础上通过扩大共享内存块实现。

```
template<int BLOCK_DIM,int NUMS_THREAD>
__global__ void mat_multi_shared_v1(double *A, double *B, double *C, int m, int
n, int k)
{
    int block_x = blockIdx.x;
    int block_y = blockIdx.y;
    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;

    if (((thread_y + block_y * BLOCK_DIM) * n + block_x * BLOCK_DIM + thread_x)*
NUMS_THREAD >= m * n)
        return;

    int begin_a = block_y * BLOCK_DIM * n;
    int end_a = begin_a + n;
```

```
    int step_a = BLOCK_DIM * NUMS_THREAD;   // 修改

    int begin_b = block_x * BLOCK_DIM;
    int step_b = BLOCK_DIM * k * NUMS_THREAD;   // 修改

    int sum = 0;

    for (int index_a = begin_a, index_b = begin_b;
        index_a < end_a; index_a += step_a, index_b += step_b)
    {
        // 修改共享内存块大小
        __shared__ double Block_A[BLOCK_DIM][BLOCK_DIM * NUMS_THREAD];
        __shared__ double Block_B[BLOCK_DIM * NUMS_THREAD][BLOCK_DIM];
        for(int i = 0; i < NUMS_THREAD; i++){
            Block_A[thread_y][thread_x* NUMS_THREAD+i] = A[index_a + thread_y *
n + thread_x* NUMS_THREAD+i];
            Block_B[thread_y* NUMS_THREAD+i][thread_x] = B[index_b + (thread_y*
NUMS_THREAD+i) * k + thread_x];
        }

        __syncthreads();

        for (int j = 0; j < BLOCK_DIM * NUMS_THREAD; j++){  // 修改
            sum += Block_A[thread_y][j] * Block_B[j][thread_x];
        }
        __syncthreads();
    }
    C[begin_b + block_y * BLOCK_DIM * k + thread_y * k + thread_x] = sum;
}
```

3. 运行结果

编译运行指令：

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

运行结果如下：

```
jovyan@jupyter-21307347:~/parallel$ ./mat
Enter values for m,n,k[128, 2048]:128 128 128
 Top left corner of matrix A:
         34          35          13          68          90          85
         36          55          40          60          91          67
         63          69          56          86          75          52
         37          93          12          31          45          38
         40          86           1          48          17          27
         60          52          14          26          72           4
 Top left corner of matrix B:
         89          21          23          36          31          88
         80          80          97          24          24          55
         64          45          31          70          65          60
         63          92          10          83          63          79
         32          82          80           7          73          37
         10          63          10          85          51          41
Time taken: 97.000000 us
 Top left corner of matrix C:
      287399      307554      321875      310493      332116      321413
      309872      305162      331223      322362      318131      328021
      308482      326946      301039      311318      318839      282391
      302005      313680      315496      305020      289417      298116
      295414      308509      332094      300978      305146      319196
      278222      332601      364722      312358      313225      335901
```

性能分析

1.全局内存

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	1870	16033	138064
16	1523	11757	99357
32	1499	11411	91627

2. 使用共享内存

线程块大小\矩阵规模	512	1024	2048
8	3750	28426	226269
16	3640	27720	221291
32	4108	29572	231297

3. 不同的数据划分方式

每个线程连续读取2个元素到共享内存：

线程块大小\线程划分数量	512	1024	2048
8	1947	14342	113745
16	1940	14111	110368
32	2377	14621	115258

每个线程连续读取4个元素到共享内存：

线程块大小\线程划分数量	512	1024	2048
8	1150	10781	82154
16	1940	7245	110368

每个线程连续读取8个元素到共享内存：

线程块大小\线程划分数量	512	1024	2048
8	720	4077	29120

可以看出，在使用全局内存时，线程块越大，平均运行时间越短，而使用共享内存后，依旧每个线程执行相同内容的计算，但时间反而变长，之后对其改进，使每个线程读取矩阵A、B的行列时连续读取一段数据，比对不同数据划分方式对性能的影响，可以看出当线程块不大，线程连续元素越大时，矩阵计算的平均运行时间越短。