中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	CUDA 并行矩阵乘法	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.5.31

1. 实验目的（200 字以内）

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。

2. 实验过程和核心代码（600 字左右，图文并茂）

定义随机初始化矩阵和打印矩阵的函数：

```
__host__ void init(double* mat, int row, int col){
    // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> distr(-5,5);

    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            mat[i*col+j] = distr(eng);
        }
    }
}
__host__ void printMatrix(double* mat, int row, int col){
    for(int i = 0; i < row; ++i){
        for(int j = 0; j < col; ++j){
            printf("%lf ",mat[i*col+j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

定义矩阵乘法核函数，由于需要比较不同的访存方式，因此需要编写 2 个不同的矩阵乘法核函数(一个使用全局内存，一个使用共享内存)，和上一次实验一样，我在 2 个不同的源文件里实现了矩阵乘法核函数：
此处应注意行号和列号的获取，以 4 个 2*2 的线程块为例，CUDA 中线程块以及线程块内的线程的排列方式如下：

blockIdx.x=0	blockIdx.x=0	blockIdx.x=1	blockIdx.x=1
blockIdx.y=0	blockIdx.y=0	blockIdx.y=0	blockIdx.y=0
threadIdx.x=0	threadIdx.x=1	threadIdx.x=0	threadIdx.x=1
threadIdx.y=0	threadIdx.y=0	threadIdx.y=0	threadIdx.y=0
blockIdx.x=0	blockIdx.x=0	blockIdx.x=1	blockIdx.x=1
blockIdx.y=0	blockIdx.y=0	blockIdx.y=0	blockIdx.y=0
threadIdx.x=0	threadIdx.x=1	threadIdx.x=0	threadIdx.x=1
threadIdx.y=1	threadIdx.y=1	threadIdx.y=1	threadIdx.y=1
blockIdx.x=0	blockIdx.x=0	blockIdx.x=1	blockIdx.x=1
blockIdx.y=1	blockIdx.y=1	blockIdx.y=1	blockIdx.y=1
threadIdx.x=0	threadIdx.x=1	threadIdx.x=0	threadIdx.x=1
threadIdx.y=0	threadIdx.y=0	threadIdx.y=0	threadIdx.y=0
blockIdx.x=0	blockIdx.x=0	blockIdx.x=1	blockIdx.x=1
blockIdx.y=1	blockIdx.y=1	blockIdx.y=1	blockIdx.y=1
threadIdx.x=0	threadIdx.x=1	threadIdx.x=0	threadIdx.x=1
threadIdx.y=1	threadIdx.y=1	threadIdx.y=1	threadIdx.y=1

可以看到，在同一个线程块内，同一行的线程的 threadIdx.y 是相同的，同一列的 threadIdx.x 是相同的；对不同的线程块，同一行的线程块的 blockIdx.y 是相同的，同一列的 blockIdx.x 是相同的。

因此行号的计算应为 $\mathrm{blockIdx.y} * \mathrm{blockDim.y} + \mathrm{threadIdx.y}$，列号的计算应为 $\mathrm{blockIdx.x} * \mathrm{blockDim.x} + \mathrm{threadIdx.x}$。
全局内存基本上和串行矩阵乘法一致，只是提取了 A 的第 row 行和 B 的第 col 列来计算 C 的第 row 行第 col 列的元素：

```
1.  __global__ void matrix_multiply_global(double* A, double* B, double* C, int M, int N, int K){
2.      int row = blockIdx.y * blockDim.y + threadIdx.y;//计算行号
```

```
    int col = blockIdx.x * blockDim.x + threadIdx.x;//计算列号

    if(row < M && col < K){
        double value = 0;

        //计算A 的第row 行和B 的第col 列的乘积，将其存储在C 的第row 行第col 列
        for(int i = 0; i < N; ++i){
            value += A[row * N + i] * B[i * K + col];
        }
        C[row * K + col] = value;
    }
}
```

对于共享内存，此次使用动态的共享内存，需要使用 2 个大小为 tile_size*(tile_size+1)[避免 bank 冲突]的数组作为线程块内的共享内存，可以将 2 个共享内存合并为一个 2*tile_size*(tile_size+1)的数组，前半部分用于存储矩阵 A 在 row 行的 tile_size*(tile_size+1)大小的块的信息，后半部分用于存储矩阵 B 在 col 列的 tile_size*(tile_size+1)大小的块的信息。

在将矩阵 A、B 中的元素存储到共享内存中时，由于避免 bank 冲突时设置了(tile_size+1)列，要进行索引判断，避免越界。此外对于不完整的块，未填满的部分必须用 0 填充，以此保证计算的正确性和完整性。

```
__global__ void matrix_multiply_shared(double* A, double* B, double* C, int M, int N, int K, int tile_size){
    extern __shared__ double smem[];//一个大小为2*(tile_size)*(tile_size+1)的共享内存，前半部分存储矩阵A 的块，后半部分存储矩阵B 的块
    double *smemA = smem, *smemB = smem + (tile_size+1) * tile_size;
    int row = blockIdx.y * blockDim.y + threadIdx.y, col = blockIdx.x * blockDim.x + threadIdx.x;//获取行号和列号
    double value = 0;

    // 接下来将当前块内的矩阵A 和B 的相应的元素存储到共享内存中并进行计算
    for (int i = 0; i < N / tile_size; ++i) {
        if (row < M && (i * tile_size + threadIdx.x) < N) {
```

```
            smemA[threadIdx.y * (tile_size + 1) + threadIdx.x] = A[row * N + i * tile_size + threadIdx.x];
        } else {
            smemA[threadIdx.y * (tile_size + 1) + threadIdx.x] = 0.0;
        }

        if (col < K && (i * tile_size + threadIdx.y) < N) {
            smemB[threadIdx.y * (tile_size + 1) + threadIdx.x] = B[(i * tile_size + threadIdx.y) * K + col];
        } else {
            smemB[threadIdx.y * (tile_size + 1) + threadIdx.x] = 0.0;
        }

        __syncthreads();
        for (int j = 0; j < tile_size; ++j) {
            value += smemA[threadIdx.y * (tile_size + 1) + j] * smemB[j * (tile_size + 1) + threadIdx.x];
        }

        __syncthreads();
    }

    if (row < M && col < K) {
        C[row * K + col] = value;
    }

}
```

在 main 函数中，输入 M、N、K 和线程块内的线程数，初始化矩阵 A 和 B 并输出，然后调用核函数并计时，再输出运算结果矩阵 C 和所用时间，最后释放内存。由于使用了动态共享内存，下面仅展示 2 个源文件调用核函数和最终输出部分，其他的部分基本和上次实验相同：

- 全局内存：

  调用核函数：

```
cudaEventRecord(start, 0);                                          // 开始计时
matrix_multiply_global<<<dimGrid, dimBlock>>>(A, B, C, M, N, K); // 调用核函数
cudaEventRecord(end, 0);                                            // 结束计时
```

```
4.    cudaEventSynchronize(end);                                    // 同步
```

输出：

```
1.    printf("线程块大小:(%d,%d) 矩阵 A 规模:%d*%d 矩阵 B 规模:%d*%d 访存方式:全局内存 所用时
      间 time:%f ms\n", blockDim_x, blockDim_y, M, N, N, K, ms);
```

- 共享内存：

  调用核函数：

```
1.    int tile_size = blockDim_x;
                              // 这里使用共享内存时仅考虑所取线程块大小均为D*D 即线程数开二次方后仍为整数
2.    cudaEventRecord(start, 0);
                              // 开始计时
3.    matrix_multiply_shared<<<dimGrid, dimBlock, 2 * tile_size *(tile_size + 1) * sizeof(double)>>>(A, B,
      C, M, N, K, tile_size); // 调用核函数
4.    cudaEventRecord(end, 0);
                              // 结束计时
5.    cudaEventSynchronize(end);
                              // 同步
```

输出：

```
1.    printf("线程块大小:(%d,%d) 矩阵 A 规模:%d*%d 矩阵 B 规模:%d*%d 访存方式:共享内存 所用时
      间 time:%f ms\n", blockDim_x, blockDim_y, M, N, N, K, ms);
```

注：blockDim_x 和 blockDim_y 表示线程块内的线程分布的 2 个维度，ms 为矩阵乘法所用时间。

3. 实验结果（500 字左右，图文并茂）

➢ 编译：

```
1.    nvcc -g -o filename filename.cu
```

(filename 为 CUDA_Matrix_Multiply_global 或 CUDA_Matrix_Multiply_shared，分别为使用全局内存和使用共享内存进行矩阵乘法的程序源代码)

➢ 运行：

```
1.    ./filename
```

(filename 为 CUDA_Matrix_Multiply_global 或 CUDA_Matrix_Multiply_shared，分别为使用全局内存和使用共享内存进行矩阵乘法的程序源代码)

➢ 运行截图，用于展示正确性：

■ 全局内存：

```
jovyan@jupyter-21307082:~/lab10$ nvcc -g -o CUDA_Matrix_Multiply_global CUDA_Matrix_Multiply_global.cu
jovyan@jupyter-21307082:~/lab10$ ./CUDA_Matrix_Multiply_global
请输入M,N,K以及每个线程块的线程数:
4 4 4 4
Matrix A:
-1.108749 2.640605 1.910906 -1.944276
-3.990538 2.519014 -3.740885 -4.520814
4.253651 -3.754338 4.326214 -2.796182
-1.937358 0.012177 1.440568 4.559577

Matrix B:
-4.309202 -2.498017 1.722062 -3.202948
0.011971 -0.829985 4.880063 -2.358799
-0.354499 -3.499240 -2.133570 -0.990606
0.717504 4.771019 -3.323838 2.915520

请输入线程块的2个维度:
2 2
Matrix C:
2.736991 -15.384889 13.362392 -10.238925
15.308624 -0.600948 28.428884 -2.635140
-21.914692 -35.988750 -10.932580 -17.206393
11.109447 21.542379 -21.505671 18.043033

线程块大小:(2,2) 矩阵A规模:4*4 矩阵B规模:4*4 访存方式:全局内存 所用时间:0.045568 ms
jovyan@jupyter-21307082:~/lab10$
```

■ 共享内存：

```
jovyan@jupyter-21307082:~/lab10$ nvcc -g -o CUDA_Matrix_Multiply_shared CUDA_Matrix_Multiply_shared.cu
jovyan@jupyter-21307082:~/lab10$ ./CUDA_Matrix_Multiply_shared
请输入M,N,K以及每个线程块的线程数:
4 4 4 4
Matrix A:
-0.508182 -3.111959 -0.813860 3.284512
-3.784496 4.691712 2.616951 4.731943
-2.315434 -1.027422 -3.866085 2.855484
-4.673103 -3.758280 1.165048 -1.200930

Matrix B:
1.239772 0.340081 -4.743119 4.115901
4.700658 4.795638 1.213844 -4.543007
0.878043 1.561768 -1.308263 -4.671943
-3.889089 2.175078 -4.949625 -4.131663

请输入线程块的2个维度:
2 2
Matrix C:
-28.746646 -9.223643 -16.559427 2.277836
1.257067 35.592132 -3.199681 -68.668133
-22.199989 -5.541609 0.659527 1.401725
-17.766484 -20.405166 22.023084 -2.641335

线程块大小:(2,2) 矩阵A规模:4*4 矩阵B规模:4*4 访存方式:共享内存 所用时间:0.045792 ms
jovyan@jupyter-21307082:~/lab10$
```

实验结果：

➢ 全局内存：

线程块大小	矩阵规模				
	128	256	512	1024	2048
16	0.594432 ms 0.826848 ms 0.347776 ms	5.420224 ms 5.241408 ms 6.539424 ms	44.776127 ms 64.883072 ms 82.401405 ms	393.388641 ms 1078.572998 ms 713.552124 ms	4103.061035 ms 11231.841797 ms 5763.865234 ms
64	0.262656 ms 0.725440 ms 0.260448 ms	2.696384 ms 20.875040 ms 2.999808 ms	23.097631 ms 268.542572 ms 75.380127 ms	197.245407 ms 3027.614990 ms 691.442627 ms	1760.379883 ms 28211.908203 ms 5664.819336 ms
256	0.154944 ms 0.700704 ms 0.240384 ms	1.004192 ms 51.459263 ms 2.630400 ms	9.664032 ms 504.949432 ms 76.031105 ms	78.566017 ms 4052.691162 ms 696.499878 ms	707.527100 ms 32727.707031 ms 5583.478516 ms
1024	0.169952 ms 2.027424 ms 0.830272 ms	0.733312 ms 18.411776 ms 9.761696 ms	6.300224 ms 253.000519 ms 84.884163 ms	50.487297 ms 4606.426758 ms 691.279846 ms	407.602173 ms 35597.371094 ms 5550.045898 ms

红色：以块划分，如线程块大小为 16 的以(4,4)的块划分数据。
蓝色：以列划分。
绿色：以行划分。

➢ 共享内存：

线程块大小	矩阵规模				
	128	256	512	1024	2048
16	0.572192 ms	6.075904 ms	45.840992 ms	383.143890 ms	3088.755371 ms
64	0.222208 ms	1.943744 ms	16.713440 ms	150.373627 ms	1471.548218 ms
256	0.156320 ms	0.811168 ms	8.601824 ms	77.821281 ms	672.780762 ms
1024	0.203040 ms	0.739904 ms	5.237376 ms	40.900257 ms	335.631866 ms

共享内存只考虑以块划分的情况。

分析：

①可以看到随着矩阵规模的增大，需要进行计算的数据增加，所用时间自然会增加。

②可以看到按块划分要优于按行和按列划分，这是因为按块划分可以更自然地实现负载平衡能够更好地实现内存合并，能够很好地匹配 GPU 的 SM（Streaming Multiprocessor）资源分配，确保较高的占用率和调度效率。而按行划分的效率要高于按列划分的效率，这是因为使用的是行主序，按行划分具有更好的空间局部性。

③可以看到使用共享内存的效率比使用全局内存的效率更高，这是因为共享内存只被线程块内的线程所共享，读写周期要小于全局内存的读写周期，当没有发生 bank 冲突时，访问共享内存和访问寄存器一样快。

④可以看到随着线程块的增大，按块划分的效率是越来越高的，这是与其能更好地负载平衡有关。对于按行划分和按列划分，总体上来说当矩阵规模较大时，线程块越大、效率越高，而当矩阵规模较小时，随着线程块的增大，效率呈现先提高后下降的趋势。先提高是因为随着线程块增大，并行度提高，效率自然增加；而后下降是因为矩阵规模相对较小，过度并行化反而会带来更大的开销，这部分开销超过了并行带来的性能提升。

4. 实验感想（300 字左右）

通过此次实验，我进一步熟悉了 CUDA 编程，学习了如何使用动态共享内存。在此次实验中，在进行共享内存矩阵乘法实验时，在将矩阵的值存储到共享内存中时，一开始并未对避免 bank 冲突后的边界处进行特殊处理，导致计算结果有误，后续通过和其他同学进行讨论并查阅资料后，解决了问题，这也让我意识到在实验过程中要更加小心谨慎。