中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

实验	CUDA并行矩阵乘法	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.5.27

1、实验目的

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：3个整数m,n,k，每个整数的取值范围均为[128, 2048]

问题描述：随机生成m×n的矩阵A及n×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。

2、实验过程和核心代码

执行命令

```
nvcc -o main main.cu
./main
```

kernel函数

```
__global__ void matrixMulKernel(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;

    if (row < m && col < k) {
        float value = 0;
        for (int e = 0; e < n; ++e) {
            value += A[row * n + e] * B[e * k + col];
        }
        C[row * k + col] = value;
    }
}
```

矩阵初始化

```
void initializeMatrix(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = static_cast<float>(rand()) / RAND_MAX;
    }
}
```

矩阵乘法

```
void matrixMulCUDA(int m, int n, int k, int blockSize) {
    size_t sizeA = m * n * sizeof(float);
    size_t sizeB = n * k * sizeof(float);
    size_t sizeC = m * k * sizeof(float);

    // 分配主机内存
    float* h_A = (float*)malloc(sizeA);
    float* h_B = (float*)malloc(sizeB);
    float* h_C = (float*)malloc(sizeC);

    // 初始化矩阵A和B
    initializeMatrix(h_A, m, n);
    initializeMatrix(h_B, n, k);

    // 分配设备内存
    float* d_A; cudaMalloc(&d_A, sizeA);
    float* d_B; cudaMalloc(&d_B, sizeB);
    float* d_C; cudaMalloc(&d_C, sizeC);

    // 将主机内存数据拷贝到设备内存
    cudaMemcpy(d_A, h_A, sizeA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, sizeB, cudaMemcpyHostToDevice);

    // 定义CUDA网格和块结构
    dim3 blockDim(blockSize, blockSize);
    dim3 gridDim((k + blockSize - 1) / blockSize, (m + blockSize - 1) / blockSize);

    // 记录开始时间
```

```
    auto start = std::chrono::high_resolution_clock::now();

    // 调用矩阵乘法核函数
    matrixMulKernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, m, n, k);

    // 同步设备
    cudaDeviceSynchronize();

    // 记录结束时间
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float, std::milli> duration = end - start;

    // 将结果从设备内存拷贝回主机内存
    cudaMemcpy(h_C, d_C, sizeC, cudaMemcpyDeviceToHost);

    // 输出计算时间
    std::cout << "Running time: " << duration.count() << " ms" << std::endl;

    // 释放内存
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

3、实验结果

线程块大小	矩阵规模				
	128	256	512	1024	2048
1	0.000655	0.003588	0.024682	0.217280	1.127790
2	0.000069	0.001138	0.008838	0.074999	0.396947
4	0.000085	0.000492	0.003701	0.028831	0.192990
8	0.000057	0.000267	0.001908	0.014485	0.115038
16	0.000043	0.000186	0.001219	0.009058	0.071836

实验结果分析：

1.随着线程块大小的增加，矩阵计算时间通常减少，但矩阵规模越小时，影响效果更显著。

2.线程块大小对于不同矩阵规模的影响不同，矩阵规模较小时，矩阵计算时间的降低幅度更大。

线程块为4：

访存方式/矩阵规模	128	256	512	1024	2048
共享内存	0.000424	0.000997	0.007727	0.060807	0.360910
非共享内存	0.000085	0.000492	0.003701	0.028831	0.192990

实验结果分析：

1.共享内存：随着矩阵规模的增大，计算时间显著增加，可能由于管理共享内存的开销带来一些性能劣势。

2.非共享内存：非共享内存方式在所有矩阵规模下都表现出更好的性能，尤其在小规模矩阵上，非共享内存表现出极低的计算时间。尽管计算时间随矩阵规模的增大而增加，但增速较共享内存方式更为缓慢。

4、实验感想

1.通过这个实验，进一步加深了对CUDA编程的理解和认识，并掌握了一些基本的性能优化技巧。未来可以尝试更多的实验和优化策略，以进一步提升并行计算的性能。

2.访存模式对性能有显著影响，利用了内存的空间局部性，减少了缓存未命中率和内存延迟，在较大规模的矩阵上表现得尤为明显。

3.较大的线程块大小可以更好地利用GPU的计算资源和内存带宽，减少内存访问延迟，从而提高性能。然而，过大的线程块也可能导致线程间的同步和调度开销增加，需要找到一个平衡点来提升性能。

4.性能优化是一个复杂的过程，需要综合考虑多个因素，包括硬件架构、算法实现、数据划分和内存访问模式等。不同的GPU硬件和应用场景可能会有不同的优化策略，因此需要针对具体情况进行调整和测试。

5.CUDA编程可以显著提升计算密集型任务的性能，但需要对GPU硬件的深入理解、对并行算法的设计和实现等。