

计算机体系结构第 2 次实验报告

220110927

王江阔

计科 9 班

一、实验四：

1.实现流程：

流程按照实验指导书进行填空，代码实现仿照实验原理的矩阵加法，代码如下：

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int width)
{
    // Calculate the row index of the P element and M
    // *** TO DO: Compute the row index for the current thread ***
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate the column index of the P element and N
    // *** TO DO: Compute the column index for the current thread ***
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure the thread is within bounds
    if ( (row < width) && (col < width) ) {
        float pValue = 0.0;

        // Each thread computes one element of the matrix
        // *** TO DO: Implement the matrix multiplication for a single element ***
        for(int k = 0; k < width; k++){
            pValue += d_M[row * width + k] * d_N[k * width + col];
        }
        // Store the computed value into the output matrix
        // *** TO DO: Write the computed value to the correct position in d_P ***
        d_P[row * width + col] = pValue;
    }
}
```

2.测试结果和原理分析：

(1) 正确性验

```
Kernel Elapsed Time: 134.665 ms
Performance= 14.85 GFlop/s, Time= 134.665 msec, Size= 2000000000 Ops
Computing result using host CPU...done.
Listing first 100 Differences > 0.000010...

Total Errors = 0
```

(2) 不同大小的矩阵对比:

```
Kernel Elapsed Time: 134.722 ms
Performance= 14.85 GFlop/s, Time= 134.722 msec, Size= 2000000000 Ops
Kernel Elapsed Time: 3648.934 ms
Performance= 14.80 GFlop/s, Time= 3648.934 msec, Size= 54000000000 Ops
```

对比上述（1）（2）（3）的结果可以看出，矩阵大小会极大程度地影响矩阵乘法的时间，而适当增大块大小可以提高矩阵乘法的计算效率；

原理：

CUDA 进行矩阵乘法优化的原理（同指导书）：

线程并行计算：CUDA 通过 GPU 上成千上万的并行线程来加速计算。在矩阵乘法中，每个元素的计算是相互独立的，CUDA 可以将每个计算任务分配给一个线程，多个线程同时执行，从而极大地提高计算效率。

块和网格结构：在 CUDA 中，线程被组织成线程块和网格。每个线程块负责计算矩阵的一部分，并在共享内存中存储中间计算结果。这种分层结构允许更好的内存管理和并行处理。例如，可以将一个线程块的每个线程分配给结果矩阵中的一个元素计算任务。

隐藏内存延迟：GPU 拥有极高的线程数量，这些线程通过切换掩盖内存延迟。

当某些线程因内存读取而被阻塞时，其他线程可以继续执行，从而能够有效地隐藏延迟，使计算单元保持高利用率。

适当增大线程块大小可以提高矩阵乘法计算效率的原因：

每个线程块需要进行调度，而调度会产生开销。增大线程块的大小意味着相同运算规模下需要调度的线程块数量减少，从而减少调度开销。线程块中的线程会共享存储器（如共享内存），增大线程块的大小有助于更充分地利用共享存储器，减少对全局内存的访问频率，提高数据访问的局部性，从而加速计算。增大线程块大小可以提高每个线程块对计算资源（如寄存器、共享内存等）的利用率，从而使 GPU 资源得以更高效地利用。

二、实验五：

1.实验流程：

（1）总体流程同指导书，主要在于 `MatrixMulSharedMemKernel` 函数的实现

该函数实现的关键代码如下：

①导入 A，B 的值：

```
if(a + ty*wA + tx < wA * wA) {  
  
    As[ty][tx] = A[a + ty*wA + tx];  
  
}  
  
else{  
  
    As[ty][tx] = 0.0f;  
  
}  
  
if(b + ty*wB + tx < wB * wB) {  
  
    Bs[ty][tx] = B[b + ty*wB + tx];  
  
}  
  
else{  
  
    Bs[ty][tx] = 0.0f;  
  
}
```

此处采用两个 if 语句对数据写入进行判断，如果数组访问越界，则会直接导入 0，否则将导入 A,B 矩阵对应位置的元素值。

②计算线程负责的子矩阵元素的累加值：

```
for(int k = 0; k < BLOCK_SIZE; k++){  
    Csub += As[ty][k]*Bs[k][tx];  
}
```

此实现即常规矩阵乘法的实现；

③写回 C 矩阵对应元素的位置：

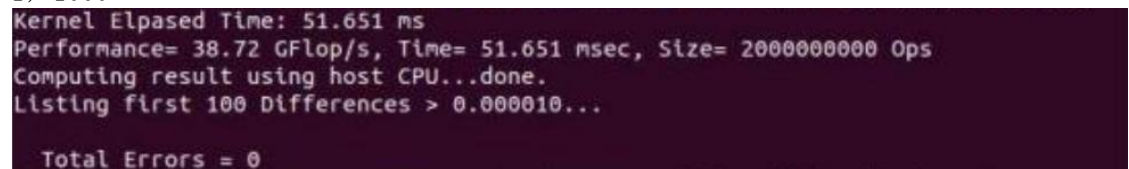
```
if(c + ty*wA + tx < wA*wA){  
    C[c + ty*wA + tx] = rintf(Csub);  
}
```

此实现采用 rintf（）函数将计算结果舍入成最接近的单精度浮点数，防止因精度问题导致的结果报错，同时，再次使用 if 语句对写入进行判断，越界则不写入；

2.测试结果和原理分析：

（1）正确性验证：

1, 1000



```
Kernel Elapsed Time: 51.651 ms  
Performance= 38.72 GFlop/s, Time= 51.651 msec, Size= 2000000000 Ops  
Computing result using host CPU...done.  
Listing first 100 Differences > 0.000010...  
  
Total Errors = 0
```

1, 1024:

对比测试：

1000:

```
Kernel Elapsed Time: 51.070 ms  
Performance= 39.16 GFlop/s, Time= 51.070 msec, Size= 2000000000 Ops
```

2000:

```
Kernel Elapsed Time: 90.517 ms  
Performance= 176.76 GFlop/s, Time= 90.517 msec, Size= 16000000000 Ops
```

从（2）的结果可以看出，该方法的优化效果相较于实验四的优化有了极大的改进；

共享内存法原理（同指导书）：

共享内存（Shared Memory）是一种位于 CUDA 架构中每个流多处理器（Streaming Multiprocessor, SM）内的快速缓存，用于同一线程块内的线程共享数据。相比全局内存，访问共享内存的速度更快，因此在需要频繁访问相同数据时，共享内存能大幅提升性能。通过将多次使用的数据提前加载到共享内存，可有效减少全局内存的访问次数，从而降低延迟、提高带宽利用率。共享内存的加速原理可以概况为以下关键点：

快速访问速度：共享内存位于 CUDA 的每个流多处理器上，具有非常低的访问延迟，速度远快于全局显存。这使得它非常适合存储需要频繁访问的数据。

线程块内共享：共享内存可以被同一线程块内的所有线程访问和共享，这意味着多个线程可以协同工作，减少重复读取相同的数据，从而节省内存访问的时间。