lab4-report

- 实验目的
  - 理解 GPU 架构下数据级并行的思想。
  - 熟悉 GPU 下 CUDA 编程框架。
- 实验要求
  - 用 C 语言编写一个程序完成两个矩阵相乘。
  - 使用 CUDA 编程框架来实现两个矩阵相乘，要求以两种方法来实现：
    - 第一种方法是：每个thread计算C中的一个元素。A的大小为[10*blocksize][10*blocksize]，B的大小为[10*blocksize][20*blocksize]
    - 第二种方法：在该程序的基础上编写一个分块（**tiled**）的矩阵乘法
- 实验环境
  - 操作系统：windows 10
  - 编译器：nvcc Cuda compilation tools, release 8.0, V8.0.60
  - CPU：Intel i5-7200U
  - GPU：NVIDIA MX150（2G）
  - memory：8GB
- 编译执行说明

```
// 编译指令
nvcc matrix_mul.cu -o mul
// 执行，需要加命令行参数指定矩阵的blocksize大小
mul.exe 64
```

- 算法核心思想

本次实验中实现的是矩阵乘法，对于给定的矩阵A和矩阵B，实现CPU上的串行算法和GPU的并行算法。在GPU上运算时基于英伟达的CUDA实现线程级的并行算法。实验中并行和串行算法的思想分别如下：

  - 串行算法

    其核心代码如下所示。

```
void serial_compute(float *A, float *B, float *C)
{
    int i, j, k, m, n;
    m = 10 * block_size;
    n = 20 * block_size;

    for(i = 0; i < m; i++)
        for(j = 0;j < n; j++)
        {
            float sum = 0;
            for(k = 0; k < m; k++)
                sum += A[i * m + k] * B[k * n + j];
            C[i * n + j] = sum;
        }
}
```

`serial_compute()` 函数用于计算两个矩阵A和B的串行矩阵乘法，结果保存在矩阵C中。简单的串行算法即按照标准的矩阵乘法进行计算，矩阵A和每一行去依次乘以矩阵B的每一列然后求和得到每个位置的计算结果，这样进行串行计算的时间复杂度为$O(m^2 \cdot n)$，计算效率较低。

- **GPU的并行算法**

  和使用CPU相比，使用GPU进行运算时，显示芯片一般具有更大的内存带宽和更大量的执行单元，所以通过GPU的线程级的并行，可以大大提高矩阵乘法的执行效率。在实验中用GPU实现的并行算法核心思想是通过将计算机内存的数据拷贝到显存，然后在GPU上的大量线程进行矩阵计算任务的分配，可以按照每一个线程计算一个元素的思想或者是利用矩阵分块的思想实现并行化，具体的GPU上的并行实现算法在下面的cuda说明部分详述。

- **CUDA平台的实现和优化**

CUDA 是 NVIDIA 的 GPGPU 模型，在 CUDA 的架构下，一个程序分为两个部份：host 端和 device 端。Host 端是指在 CPU 上执行的部份，而 device 端则是在显示芯片上执行的部份。Device 端的程序又称为 kernel。通常 host 端程序会将数据准备好后，复制到显卡的内存中，再由显示芯片执行 device 端程序，完成后再由 host 端程序将结果从显卡的内存中取回。在实验中该过程的伪代码如下所示

```
    // Allocate GPU Memory spcae
    cudaMalloc(&CUDA_A, sizeof(float) * m * m);
    cudaMalloc(&CUDA_B, sizeof(float) * m * n);
    cudaMalloc(&CUDA_C, sizeof(float) * m * n);
    // Copy to GPU memory
    cudaMemcpy(CUDA_A, A, m * m * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(CUDA_B, B, m * n * sizeof(float), cudaMemcpyHostToDevice);
    // GPU Computing...
    // Copy result from GPU memory
    cudaMemcpy(C2, CUDA_C, m * n * sizeof(float), cudaMemcpyDeviceToHost);
```

在初始化矩阵然后在CUDA模型上实现并行化算法，可以分别按照每个线程分配一个元素计算任务和矩阵分块然后利用共享内存实现快速并行化两种思想进行算法设计。

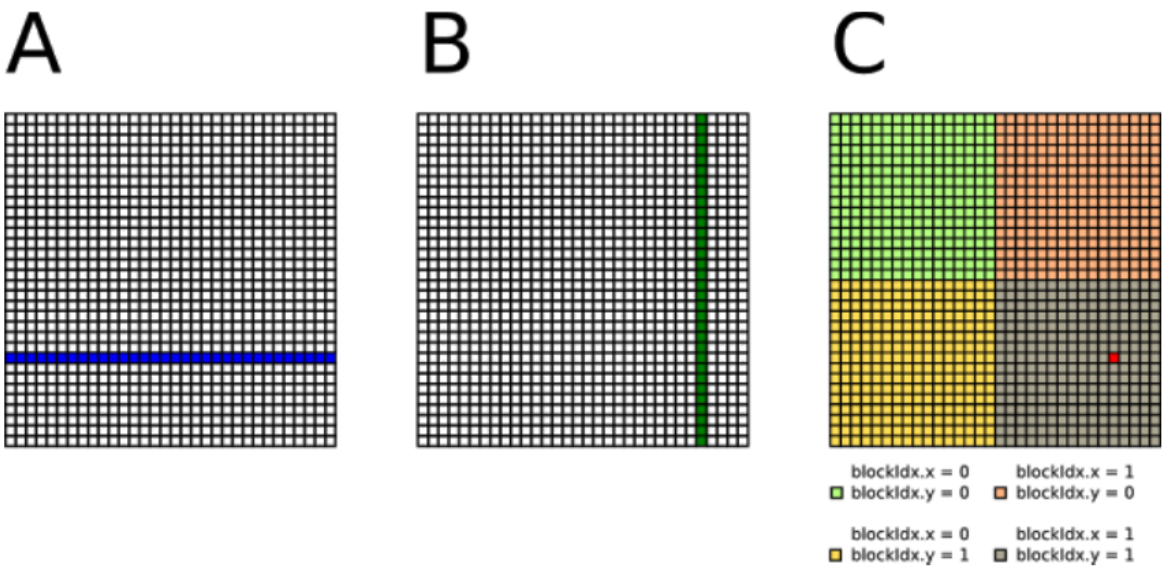
在 CUDA 架构下，显示芯片执行时的最小单位是 thread。数个 thread 可以组成一个 block。所以在配置CUDA计算函数时，首先要根据块大小和矩阵规模确定threads和blocks进行配置。

```
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE, 1);
    dim3 blocks(n / BLOCK_SIZE, m / BLOCK_SIZE, 1);
    // 调用并行算法一计算函数
    parallel_compute<<<blocks, threads>>>(CUDA_A, CUDA_B, CUDA_C, block_size);
    // ...
    // 调用并行算法二计算函数
    tiled_parallel_compute<<<blocks, threads>>>(CUDA_A, CUDA_B, CUDA_C, block_size);
```

- 算法一：每个线程计算**C**的一个元素

  每一个 block 所能包含的 thread 数目是有限的。执行相同程序的 block，可以组成 grid。不同 block 中的 thread 无法存取同一个共享的内存，因此无法直接互通或进行同步。因此，不同 block 中的 thread 能合作的程度是比较低的。但这样可以让程序不用担心显示芯片实际上能同时执行的 thread 数目限制，一个具有很少量执行单元的显示芯片，可能会把各个 block 中的 thread 顺序执行，而非同时执行。不同的 grid 则可以执行不同的程序。

  由于 global memory 并没有 cache，所以要避开巨大的 latency 的方法，就是要利用大量的 threads，基于以上的CUDA模式，直接利用GPU进行并行算法加速，每个thread计算C中的一个元素。CUDA实现的思想如下图所示，通过blockId确定行和列，然后每一个线程计算该行和列确定的结果矩阵C位置，在一个for循环中花费$O(m)$的时间即可完成并行计算过程。



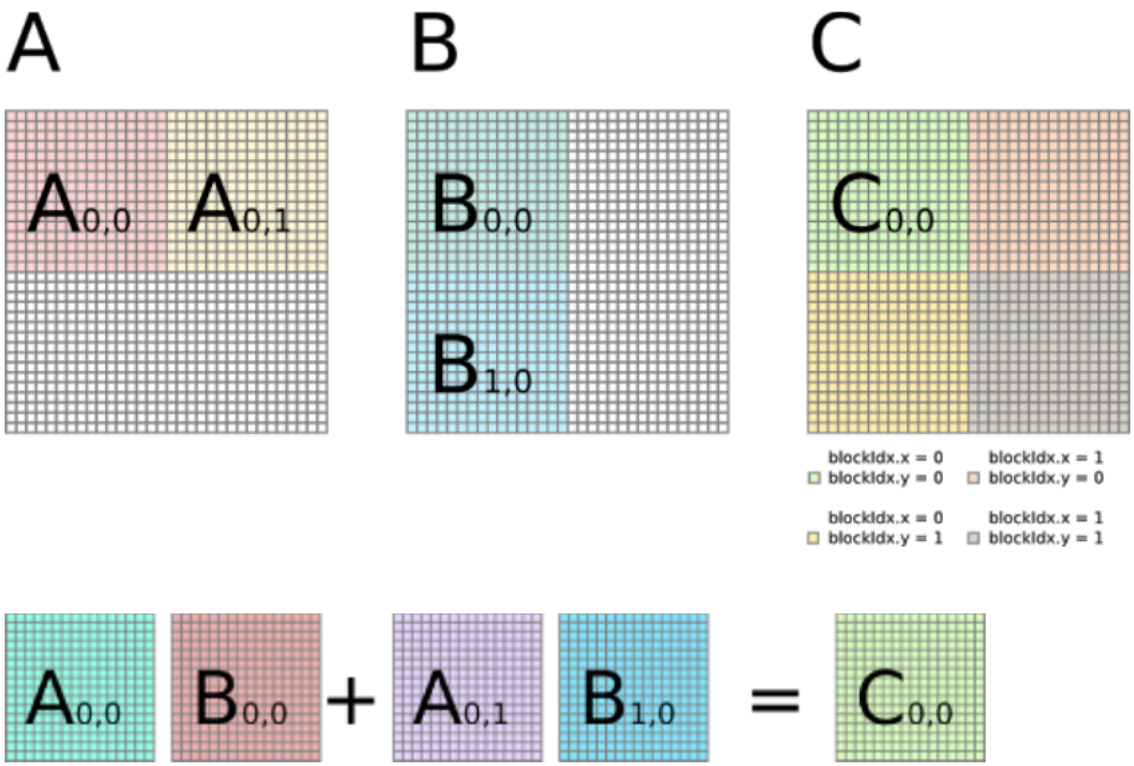
  并行算法一的核心代码如下所示：

```
__global__ void parallel_compute(float *A, float *B, float *C, int block_size)
{
    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int column = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int m = 10 * block_size, n = 20 * block_size;
    float sum = 0;

    if(row < m && column < n)
    {
        for(int i = 0; i < m; i++)
            sum += A[row * m + i] * B[i * n + column];
        C[row * n + column] = sum;
    }
}
```

- 算法二：基于算法一进行分块优化

一个 block 中的 thread 能存取同一块共享的内存，而且可以快速进行同步的动作。在 CUDA 中，thread 是可以分组的，也就是 block。一个 block 中的 thread，具有一个共享的 shared memory，也可以进行同步工作。利用 __shared__ 声明的变量表示这是 shared memory，是一个 block 中每个 thread 都共享的内存。它会使用在 GPU 上的内存，所以存取的速度相当快，可以在并行算法一的基础上进行进一步的优化。矩阵的分块思想如下图所示，在确定分块大小后，利用 shared memory 来储存每个 row 的数据。因为只有同一个 block 的 threads 可以共享 shared memory，因此现在一个 row 只能由同一个 block 的 threads 来进行计算。另外也需要能存放一整个 row 的 shared memory。在实现该算法时，需要进行分块矩阵的同步，利用 CUDA 的内部函数 `__syncthreads()`，block 中所有的 thread 都要同步到该函数这个点，才能继续执行，从而实现计算过程的同步。



分块的并行算法核心代码如下：

```
__global__ void tiled_parallel_compute(float *A, float *B, float *C, int block_size)
{
    __shared__ float block_A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float block_B[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int column = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int m = 10 * block_size, n = 20 * block_size;
    int t_x = threadIdx.x, t_y = threadIdx.y;
    float sum = 0;

    for(int i = 0; i < m / BLOCK_SIZE; i++)
    {
        block_A[t_y][t_x] = A[row * m + i * BLOCK_SIZE + t_x];
        block_B[t_y][t_x] = B[(i * BLOCK_SIZE + t_y) * n + column];
        __syncthreads();

        for(int j = 0; j < BLOCK_SIZE; j++)
            sum += block_A[t_y][j] * block_B[j][t_x];
        __syncthreads();
    }
    C[row * n + column] = sum;
}
```

- 问题回答
  - 第一种方法
    - blocksize代表了什么含义?

      blocksize表示每个二维block的宽度
    - 限制该程序运行速度的瓶颈是什么？

      运行时需要从显存中读取矩阵数据，而**GPU**的带宽是有限的
  - 第二种方法
    - 块的大小应该如何分配才能充分利用每个block的共享显存？

      块要尽可能**大**，块越**大**则对通信的压**力**越小。但是块不能太**大**，如果每个块分配的 **shared memory** 超过了某个阈值，将会直接导致每个**shared memory**上可以驻留的线程数减少，降低并**行**度。因此，要根据具体的显卡的参数决定块的**大小**。
    - 块过大会导致什么问题?

      块太**大**会导致**shared memory**不够**用**
    - 如果矩阵大小任意，应该如何修改？

      通过在矩阵中填充**0**，来使矩阵的长宽一致。
- 实验结果和分析

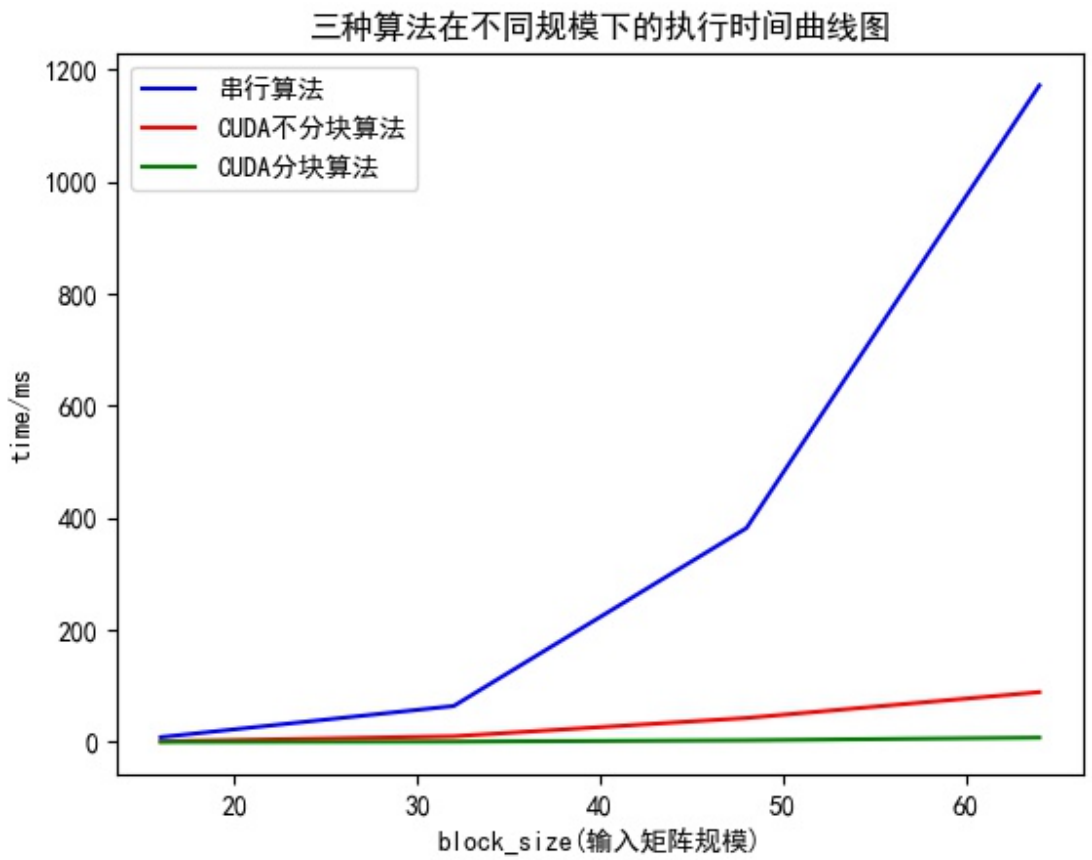
在实验中根据输入的blocksize大小随机初始化矩阵A和B，然后通过串行算法计算结果矩阵C，并且记录串行算法的时间，然后分别执行两种CUDA并行算法，得到结果并且与串行算法的结果进行比较，结果显示最终乘法得到的矩阵相同，证明算法正确的执行，同时分别记录并行的执行时间。多次改变矩阵规模后，记录的运行时间如下表所示。

程序的输入为blocksize，矩阵规模为A[10*blocksize][10*blocksize]，B[10*blocksize][20*blocksize]

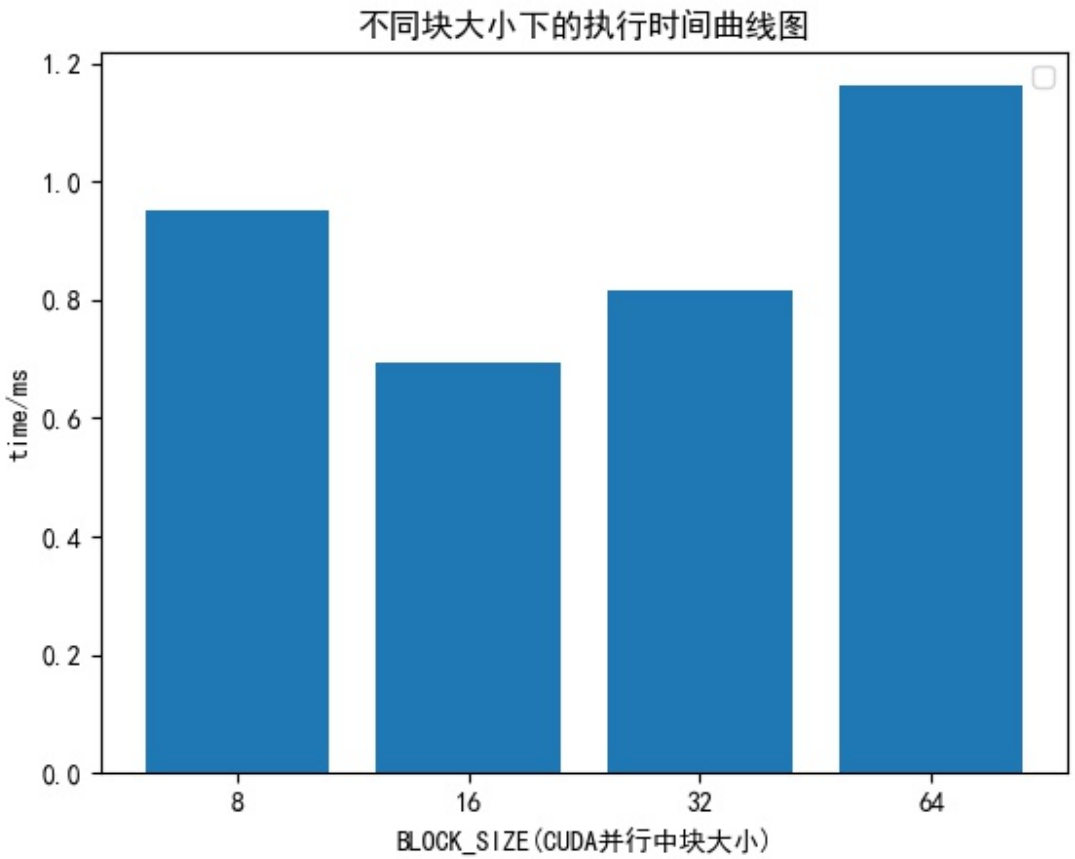
(取分块大小BLOCK_SIZE = 16)

规模(blocksize)\算法	串行算法	**CUDA**并行不分块算法	**CUDA**并行分块算法
16	8.71ms	1.921ms	0.109ms
32	64.3ms	10.3ms	0.694ms
48	382ms	42.9ms	3.19ms
64	1172ms	89.1ms	7.84ms

绘制曲线得到：



改变BLOCK_SIZE大小，矩阵A，B的输入规模固定为A[10*32][10*32]，B[10*21][20*32]。得到的执行时间统计柱状图如下：



分析结果可知，通过CUDA并行化后，利用GPU的大量线程进行加速可以显著提高矩阵乘法的执行效率，并且加速比随着矩阵规模的变大而增大，CUDA用分块进行优化后的时间也会得到很大的优化。在进行CUDA函数的配置时，设定的矩阵的块大小也会影响时间程序的执行效率，需要根据pc的硬件配置选择恰当的块大小从而获得最优的加速效果。

- 实验总结

此次实验实现了矩阵乘的算法以及利用CUDA模型在GPU上实现并行化加速算法，通过实验可以发现对于密集运算的程序，尤其是向量或者矩阵的运算，CUDA可以显著的改善时间性能，充分利用GPU的大量线程进行加速，在训练神经网络或者其他数据计算时可以代替CPU加快执行时间。