实验题目：

矩阵乘矩阵算法的并行程序优化

实验内容：

将矩阵相乘算法，应用 CUDA 编程进行并行优化，实现了两种 GPU 上的并行矩阵乘矩阵算法，并分析了这两种算法相对于原始 CPU 上运行的矩阵相乘算法的加速比。

实验环境：

Visual Studio 2017 (v141)

NVIDIA CUDA 9.1

Intel® Core™ i7-8650U CPU @ 1.90GHz

NVIDIA GeForce GTX 1050

原始程序：

```
void multMatrixNoCUDA(const float* a, const float* b, float* c, int m,
int n, int l) {
    for (int i = 0; i<m; i++) {
        for (int j = 0; j<l; j++) {
            double sum = 0;
            for (int k = 0; k<n; k++) {
                sum += a[i*n + k] * b[k*l + j];
            }
            c[i*l + j] = sum;
        }
    }
}
```

该函数为一个矩阵乘矩阵的算法。函数的参数分别为大小为 mxn 的矩阵 A，大小为 nxl 的矩阵 B，C 为 AxB 的结果，大小为 mxl，m、n、l 都作为参数传到该函数中。

根据线性代数，对于 C 中每个元素C_{ij}，$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$，用三层循环完成该函数，时间复杂度为 O(mnl)

并行程序 1：

使用 GPU 进行并行化处理。GPU 中每块可以设置最多 1024 个线程，这些线程并行地进行计算；同时可以设置多个块。

对于 cuda 的每个块和线程，有全局参数 blockIdx, blockDim, threadIdx，表示块编号、块线程编号。

```
__global__  void multMatrixCUDA(const float* a, const float* b, float*
c, int m, int n, int l) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row = idx / l;
    int column = idx % l;

    if (row < m && column < l) {
        float sum = 0;
        for (int i = 0; i < n; i++) {
            sum += a[row * n + i] * b[i * l + column];
        }
        c[idx] = sum;

    }
}
```

该函数的参数列表与原始程序相同。这里的并行思想是用每个块中的每个线程来计算输出矩阵每一位的值。这样一来，理想加速比就是线程数。

如下所示，在调用该函数时，将 nblocks 设置为(m * l + 255) / 256，线程数设为 256，则可以保证，上述函数中 idx 参数的取值范围为 0 ~ nblocks * 256，即 idx

最大的取值大于 m * l。

```
int nblocks = (m * l + 255) / 256;
multMatrixCUDA <<< nblocks, 256 >>> (cuda_a, cuda_b, cuda_c, m, n, l);
```

gpu 中每一个块和线程对应唯一的一个 idx，如果 idx 小于 m*l，计算 idx 对应 C 中的元素的值。由于 block 之间不能并行，最终时间复杂度为 O(nblocks * n)，相对于 CPU 程序最大加速比为 256，实际上由于多线程之间调度，GPU 启动时间等各种原因，达不到这个最佳加速比。

并行程序 2：

分块矩阵乘法：

分块矩阵乘法的原理是，若 C = A x B，可以将 A 分块成子矩阵 A1,1, A1,2, … ，B 分为子矩阵 B1,1, B1,2, …，其中，A 的每个子矩阵的宽与 B 的每个子矩阵的高相同。这样一来，$C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$，C 也由这些子矩阵组成。

实际计算中，我们用每一个 block 来计算 C 的一个子矩阵，每一个线程计算 C 的一个子矩阵的元素，这样以来，我们需要设置一个块宽度 TILE_WIDTH，这里经过测试，块宽度为 16 时，性能较好。子矩阵的每块大小为 TILE_WIDTH x TILE_WIDTH，调用 cuda 函数时，需要设置 gridSize 和 blockSize 如下：

```
    dim3 gridSize, blockSize;
    blockSize.x = TILE_WIDTH;
    blockSize.y = TILE_WIDTH;
    blockSize.z = 1;
    gridSize.x = (m + blockSize.x - 1) / blockSize.x;
    gridSize.y = (l + blockSize.y - 1) / blockSize.y;
    gridSize.z = 1;
    multMatrixCUDA_tiled <<<gridSize, blockSize>>>(cuda_a, cuda_b,
cuda_d, m, n, l);
```

函数如下所示：

```
__global__  void multMatrixCUDA_tiled(const float* a, const float* b,
float* c, int m, int n, int l) {
    __shared__ float shared_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float shared_b[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = bx * TILE_WIDTH + tx;
    int col = by * TILE_WIDTH + ty;
    float sum = 0.0;
    for (int i = 0; i*TILE_WIDTH <= n; i++) {
        if (i * TILE_WIDTH + ty < n && row < m)
            shared_a[tx][ty] = a[row * n + i * TILE_WIDTH + ty];
        else shared_a[tx][ty] = 0;
        if (i * TILE_WIDTH + tx < n && col < l)
            shared_b[tx][ty] = b[(i * TILE_WIDTH + tx)*l + col];
        else shared_b[tx][ty] = 0;
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; k++)
            sum += shared_a[tx][k] * shared_b[k][ty];
        __syncthreads();
    }
    if (row < m && col < l)
        c[row*l + col] = sum;
}
```

设置一个外层循环和一个参数 sum，sum 初始为 0，循环迭代参数为 i。

每块计算的子矩阵编号为(bx, by)，其中 bx = blockIdx.x，by = blockIdx.y。

每次循环时，计算 Abx， x B i,by 的值。具体实现为，设置两个 shared 矩阵 shared_a 和 shared_b，每次由一个线程将该线程对应的元素赋值给 shared_a 和 shared_b，线程同步一次后，由每一个线程计算 shared_a x shared_b 在对应线程位置的值，加在 sum 参数上，再次同步，进入下一次循环迭代。

特别的，如果矩阵长宽不能被 TILE_WIDTH 整除的话，可以将 shared_a 和 shared_b 中，矩阵中没有对应的数据的地方置为 0，再进行相乘，这样就可以很好地进行矩阵大小任意时的计算。

这种算法的时间复杂度为 O(nblocks * n), 相对于 CPU 算法的最好加速比为 256，即 TILE_WIDTH x TILE_WIDTH，实际运行时虽然不能达到这么好的加速比，但当规模较大时，也可以轻易达到超过 200 的加速比。

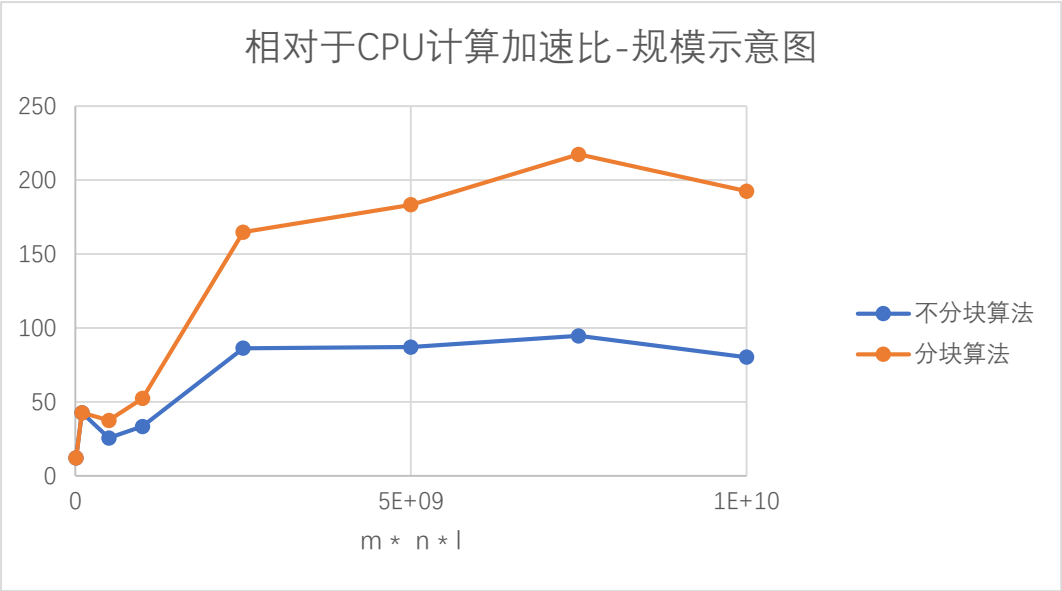
总体来说，在同样使用 GPU 计算的情况下，分块的矩阵算法的用时要比不分块的算法短。

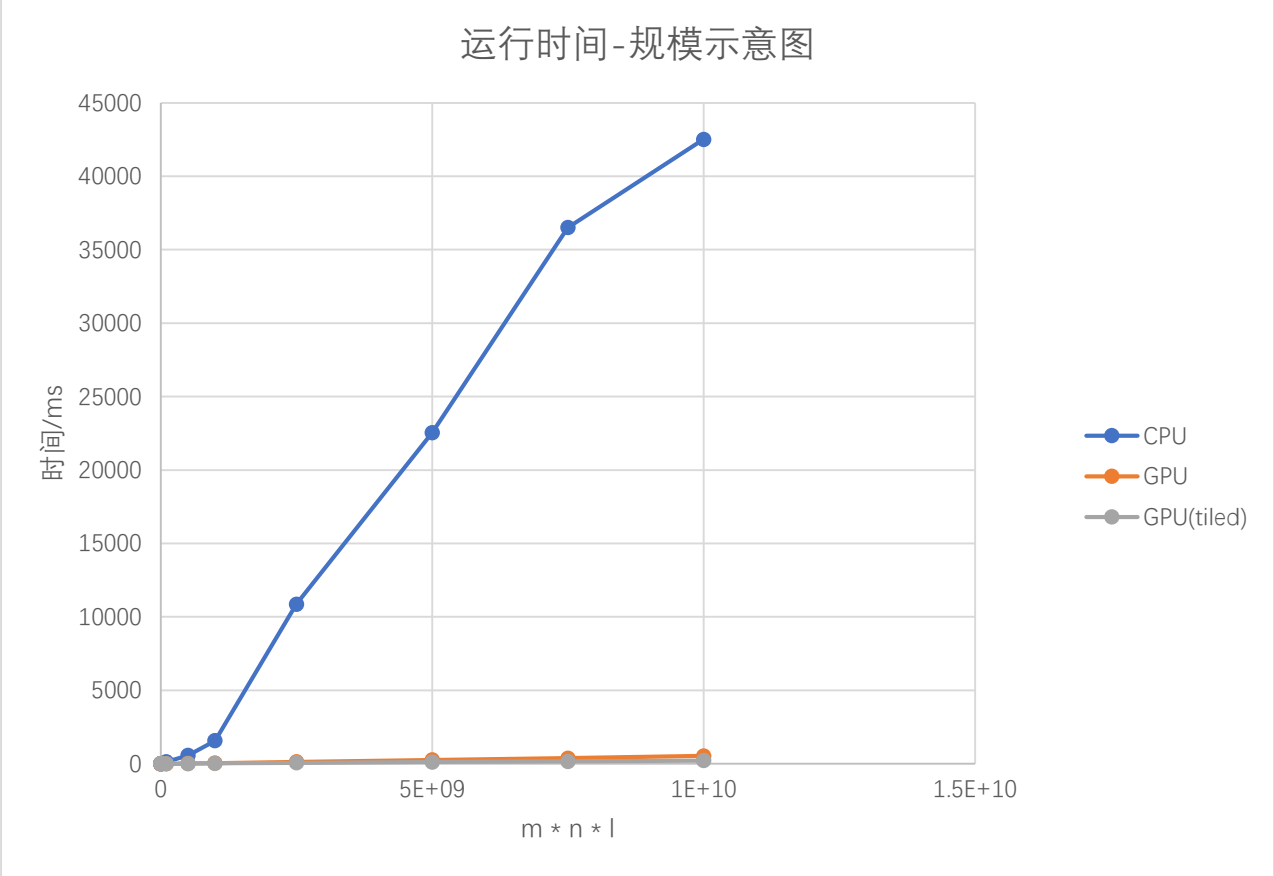
性能分析：

分别在 m*n*l 不同的情况下进行测试，其中，两个使用 GPU 的程序，线程数都为 256.

测试结果如下，其中，第四列为不分块 GPU 算法相对于 CPU 计算的加速比，第六列为分块 GPU 算法相对于 CPU 计算的加速比。(所有的时间单位为毫秒)

m*n*l	CPU	GPU	加速比	GPU(tiled)	加速比
1000	0	0	/	1	/
10000	0	0	/	0	/
100000	0	0	/	1	/
1000000	1	0	/	0	/
10000000	12	1	12	1	12
100000000	128	3	42.66667	3	42.66667
500000000	562	22	25.54545	15	37.46667
1000000000	1567	47	33.34043	30	52.23333
2500000000	10868	126	86.25397	66	164.6667
5000000000	22536	259	87.01158	123	183.2195
7500000000	36511	386	94.58808	168	217.3274
10000000000	42513	530	80.21321	221	192.3665





结果分析，可以看出，随着计算规模的提升，CPU 矩阵乘法的耗时大致也是等比提升的，而使用使用 GPU 算法时，加速比开始时都较低，随着计算规模的增加上涨，最高甚至达到了 200 多。

原因大致是规模较小时，计算所花费时间很短，GPU 计算中却需要花费时间在线程切换、调度等上面，导致了加速比开始时不理想的情况。

比较起来看，使用分块算法的时间的耗时是明显小于不分块 GPU 算法的，但实际上这两个算法，包括 CPU 算法进行矩阵元素相乘的次数是一样的，CPU 算法由于没有任何并行导致效率最慢，而分块算法利用了共享显存，有效降低了访存的开销，使得耗时较短。

附：

源代码见 main.cu，其中包括了原始程序和两种并行优化算法。执行时，需要输入三个参数，m、n、l，程序接受这三个参数后，随机生成大小为(mxn)和(nxl)的两个矩阵，并分别用三种算法对这两个矩阵做乘法运算，最后输出这三种算法所用的事件。

需要注意的是，m*n*l 最好不要大于 10^10，否则 GPU 计算可能因为规模过大无法进行。