华南理工大学

《高性能计算与云计算》课程实验报告

实验题目：CUDA 编程实验

姓名：何宇航　学号：201830170110

班级：18 计科(2)班　组别：

合作者：

指导教师：

实验概述

【实验目的及要求】

实验目的：

本实验的目的是通过练习掌握 GPU 并行编程的知识和技巧。

λ 了解并行算法的设计方法

λ 掌握 CUDA 并行程序编写的基本步骤

λ 掌握 CUDA 编程环境和工具的使用

λ 了解 CUDA 程序调试和调优的技巧

实验要求：

独立完成实验内容；

λ 实验报告；（简单要求如下）

4) 并行算法的设计思想

5) 程序设计及实现

6) 回答实验中提出的问题；

7) 结果分析；

λ 随实验报告，附代码、程序说明以及运行结果。

【实验环境】

操作系统：Windows XP

实验内容

【实验过程】

实验报告地址：

https://www.wolai.com/dykoYYEoNjgVjqobXrwhK7

任务一

实现并行矩阵乘法，说明原串行算法的原理以及并行化的方法。

问题描述：

输入：随机初始化矩阵 a，矩阵 b，其中矩阵 a 是 a_r*a_c，其中矩阵 b 是 b_r*b_c

输出：并行计算矩阵 c=a*b，串行计算矩阵 d=a*b

注：

- 为了简化，以下实验中矩阵都是方块矩阵(n*n)
- 矩阵 a 和矩阵 b 都按照行扁平化为数列
- CUDA 需要划分 Grid 和 Block，如果使用 share memory 影响不大，故不赘述

串行算法设计

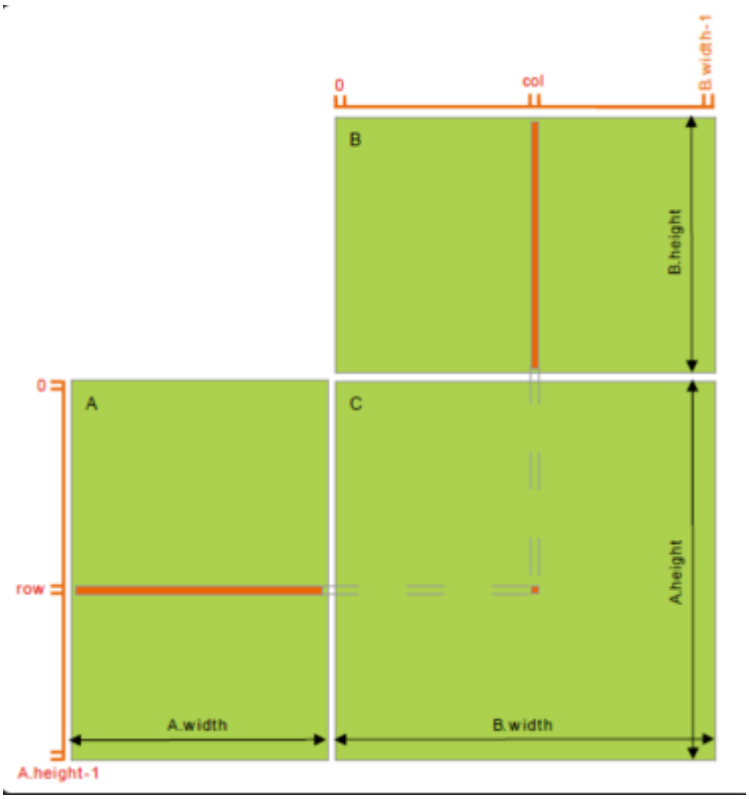
串行的矩阵乘法比较简单，是逐个遍历对应 d 矩阵元素的乘法累加计算，复杂度是 O(N^3)

串行算法实现

```C++
//CPU矩阵乘法，存入矩阵d
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        double t = 0;
        for (int k = 0; k < n; k++)
        {
            t += a[i * n + k] * b[k * n + j];
        }
        d[i * n + j] = t;
    }
}
```

并行算法设计

下面的并行的矩阵思路也比较简单，首先对于每个线程计算矩阵 c 中的一个元素，即要求 GPU 线程的数量要大于等于 c 矩阵元素的数量，如果大于则会判断越界，线程实际上不参与运算。当一个 GPU 线程调用**global** MatMatMul()函数，首先计算线程在 Grid 中的编号，确定计算哪一个结果矩阵中的元素行和列，然后调用公式，乘法累加计算出该位置的结果，由于是 GPU 并行计算复杂度将为 O(N^2)



并行算法实现

```cpp
template <size_t BLOCK_SIZE>
void __global__ MatMatMul(
    const int* A,
    const int* B,
    int* C,
    const size_t m,//a_r
    const size_t n,//b_r a_c
    const size_t k)//b_c
{
    int bx = blockIdx.x;    int by = blockIdx.y;
    int tx = threadIdx.x;    int ty = threadIdx.y;

    //确定结果矩阵中的行和列
    int row = by * BLOCK_SIZE + ty;
    int column = bx * BLOCK_SIZE + tx;

    if (row < m && column < k)
    {
        int t = 0;

        for (int i = 0; i < n; i++)
        {
            t += A[row * n + i] * B[i * n + column];
        }
        C[row * n + column] = t;
    }

}
```

C++

算法正确性比较:结果与串行一致

n1*n2:4096*4096
TILE_WIDTH:32
dimGrid:128,128,1
dimBlock:32,32,1
start of CUDA:8141
end of CUDA and start of CPU:78381
end of CPU:1415862
rate 19
80745 80745
83321 83321
82781 82781
83087 83087
82103 82103
84893 84893
82762 82762
83588 83588

82976 82976
81974 81974
84791 84791
82936 82936
82148 82148
84610 84610
84097 84097
.....

任务二

给定如下函数 $f(x)$ ，使用规约求和求出 $\sum_{i=0}^{32767} f(i)$ 的值（注意：请不要修改本函数，请从求和入手）。

```
__device__ int f(int x)
{
    int sum = 0;
    int b = x, e = b + x;
    for (int i = b; i <= e; ++i)
        sum += int(sin(double(i)) * 2);
    return sum;
```

问题描述

输入：存放 0~32767 的数列 data

输出：函数 f(x)的累加结果 result

并行算法设计

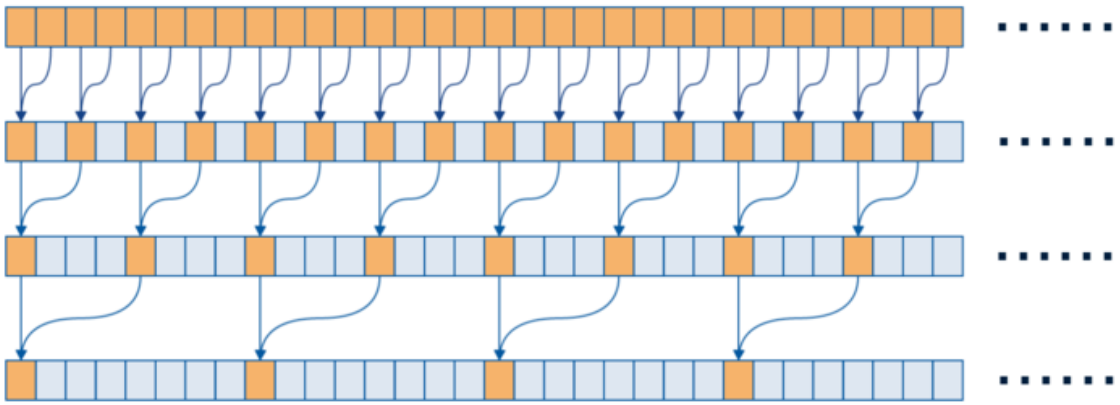
首先生成数列 data，将数据分块，分块的方法对运行速度有很大影响，这里的策略是：

//将数据分成 64 个 Block 每个 Block 里面有 128 个线程，故每个线程处理 32767/(128*64)个元素的求和
#define THREAD_NUM 128
#define BLOCK_NUM 64

其次，每个块内有一个 128 长度的共享内存，用于在 Block 内进行规约求和的计算，每个线程对共享数组里面的结果规约求和，并把求和的结果放到 result 里面，最好 result 得到 64 个结果，result 再拷贝到内存计算最后的结果，由于每个线程的任务负载会不

一样，所以需要在特定的地方同步块内的各个线程。

会出现每个线程调用 f(x)计算多个结果的情况，所以要根据线程数和 data 的长度设计好每个线程的负载，也就是计算 data 的偏移量。同时在规约求和的时候，也要注意偏移量的问题。规约求和的思路如下：



并行算法实现

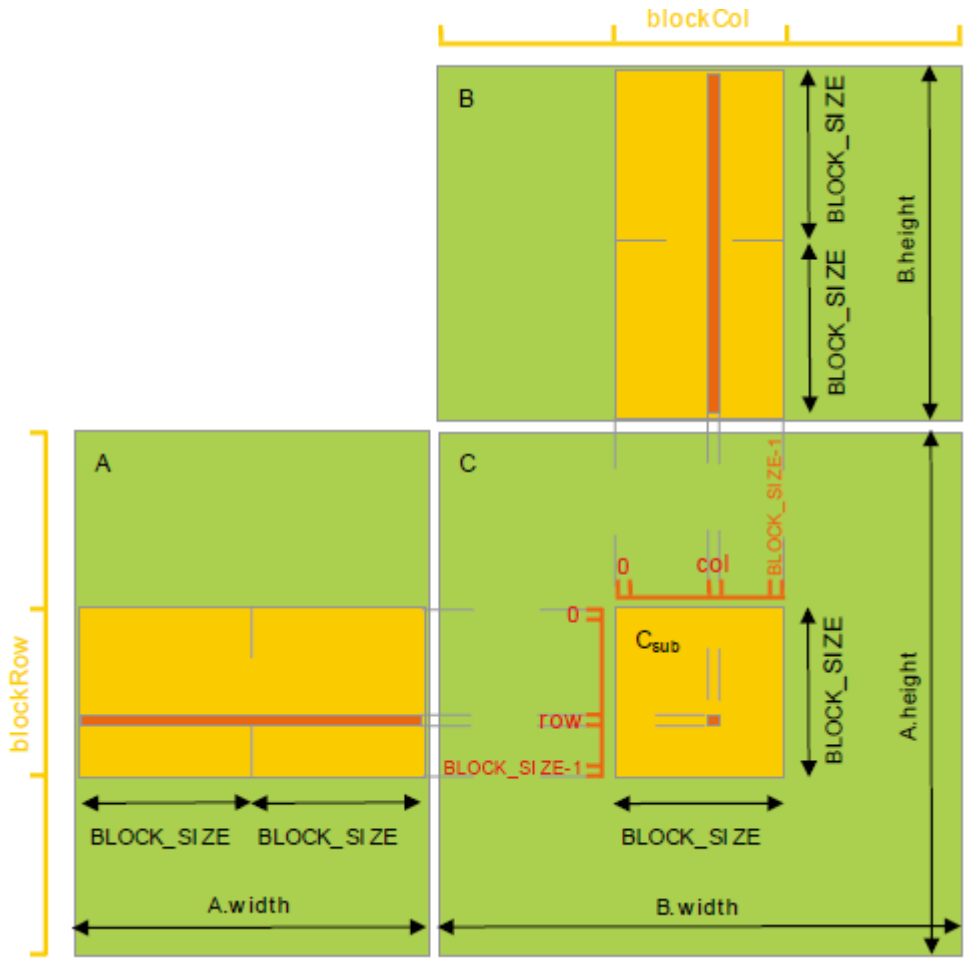
```C++
__global__ static void sumOfF(long int* num, long int* result)
{

    //声明一块共享内存
    extern __shared__ long int shared[THREAD_NUM];

    //表示目前的 thread 是第几个 thread（由 0 开始计算）
    const int tid = threadIdx.x;

    //表示目前的 thread 属于第几个 block（由 0 开始计算）
    const int bid = blockIdx.x;
    shared[tid] = 0;
    int i;
    //thread需要同时通过tid和bid来确定
    for (i = bid * THREAD_NUM + tid; i < DATA_SIZE; i += BLOCK_NUM * THREAD_NUM) {

        shared[tid] += f(num[i]);

    }
    __syncthreads();
    //树状加法
    int offset = 1, mask = 1;

    while (offset < THREAD_NUM)
    {
        if ((tid & mask) == 0)
        {
            shared[tid] += shared[tid + offset];
        }

        offset += offset;
        mask = offset + mask;
        __syncthreads();

    }
    if (tid == 0)
    {
        result[bid] = shared[0];
    }

}
```

程序调优

找出耗时最大的部分，并进行优化；针对改进进行性能比较。

分析：对于矩阵乘法，实际上在每个块内进行运算对于结果矩阵 c 的一块区域，也就是说同一个块的不同线程会使用到矩阵 a 和矩阵 b 的相同区域，如果把每个线程要读取的

相同区域设置为 share memroy 会增加性能。同时由于 share memory 的限制，如果把矩阵 a b 的整个行和列同时设计为共享内存效果不佳，所以可以进一步将 a b 应该共享的整行和整列再划分成对于的片区，每个 CUDA 块内循环读取相邻的片区，读取完后同步，然后线程进行改片区内的矩阵运算，求出 c 矩阵结果元素的部分和，然后再各个进行同步，确保读取的当前矩阵 a b 片区已经运算完成，然后进入下一片区进入下一循环。示意图：



算法实现

```
template <size_t BLOCK_SIZE>
void __global__ MatMatMul(
  const int* A,
  const int* B,
  int* C,
  const size_t m,//a_r
  const size_t n,//b_r a_c
  const size_t k)//b_c
{
```

```
  //申请共享内存，存在于每个 block 中
  __shared__ int ds_A[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ int ds_B[BLOCK_SIZE][BLOCK_SIZE];


  int bx = blockIdx.x;    int by = blockIdx.y;
  int tx = threadIdx.x;    int ty = threadIdx.y;

  //确定结果矩阵中的行和列
  int Row = by * BLOCK_SIZE + ty;
  int Col = bx * BLOCK_SIZE + tx;

  //临时变量
  int Cvalue = 0;

  //循环读入 A,B 瓦片，计算结果矩阵，分阶段进行计算
  for (int t = 0; t < (n - 1) / BLOCK_SIZE + 1; ++t)//n = a_c,b_r
  {
    //将 A,B 矩阵瓦片化的结果放入 shared memory 中，每个线程加载相应于 C 元素的
A/B 矩阵元素
    //瓦片就是一个窗口，将要运算的 A 和 B 矩阵再划分
    if (Row < m && t * BLOCK_SIZE + tx < n)    //越界处理，满足任意大小的矩
阵相乘
      //ds_A[tx][ty] = A[t*TILE_WIDTH + tx][Row];
      ds_A[tx][ty] = A[Row * n + t * BLOCK_SIZE + tx];//以合并的方式加载瓦片
    else
      ds_A[tx][ty] = 0.0;

    if (t * BLOCK_SIZE + ty < n && Col < k)
      //ds_B[tx][ty] = B[Col][t*TILE_WIDTH + ty];
      ds_B[tx][ty] = B[(t * BLOCK_SIZE + ty) * k + Col];
    else
      ds_B[tx][ty] = 0.0;

    //保证 tile 中所有的元素被加载
    __syncthreads();

    for (int i = 0; i < BLOCK_SIZE; ++i)
      Cvalue += ds_A[i][ty] * ds_B[tx][i];//从 shared memory 中取值

    //确保所有线程完成计算后，进行下一个阶段的计算
    __syncthreads();

    if (Row < m && Col < k)
```

```
        C[Row * k + Col] = Cvalue;
    }

}
```

性能比较

未优化：
n1*n2:4096*4096
TILE_WIDTH:16
dimGrid:256,256,1
dimBlock:16,16,1
start of CUDA:2780
end of CUDA:178684
cost:175.904
优化后：
n1*n2:4096*4096
TILE_WIDTH:16
dimGrid:256,256,1
dimBlock:16,16,1
start of CUDA:2780
end of CUDA:86259
cost:83.479

性能提升，下面的实验探究皆使用优化后的版本。

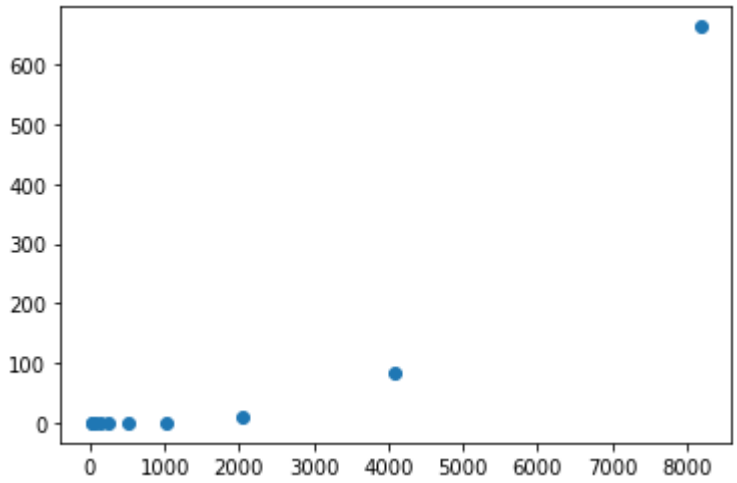
自动生成不同大小的矩阵或数据，多次运行程序，记录在不同矩阵大小或数据

长度下的运行时间。

实验数据

方块矩阵大小 耗时
16 0.001
32 0.001
64 0.001
128 0.004
256 0.021
512 0.167
1024 1.309

2048 10.492
4096 83.479
8192 663.827



分析

并行算法在数据规模很大的时候，耗时变长，且大致呈指数上升。在规模中等（4096）的时候相对于串行有优良的表现，也就是规模小、中等的时候矩阵并行算法效果好，规模大的时候，规模越大，表现相对变差

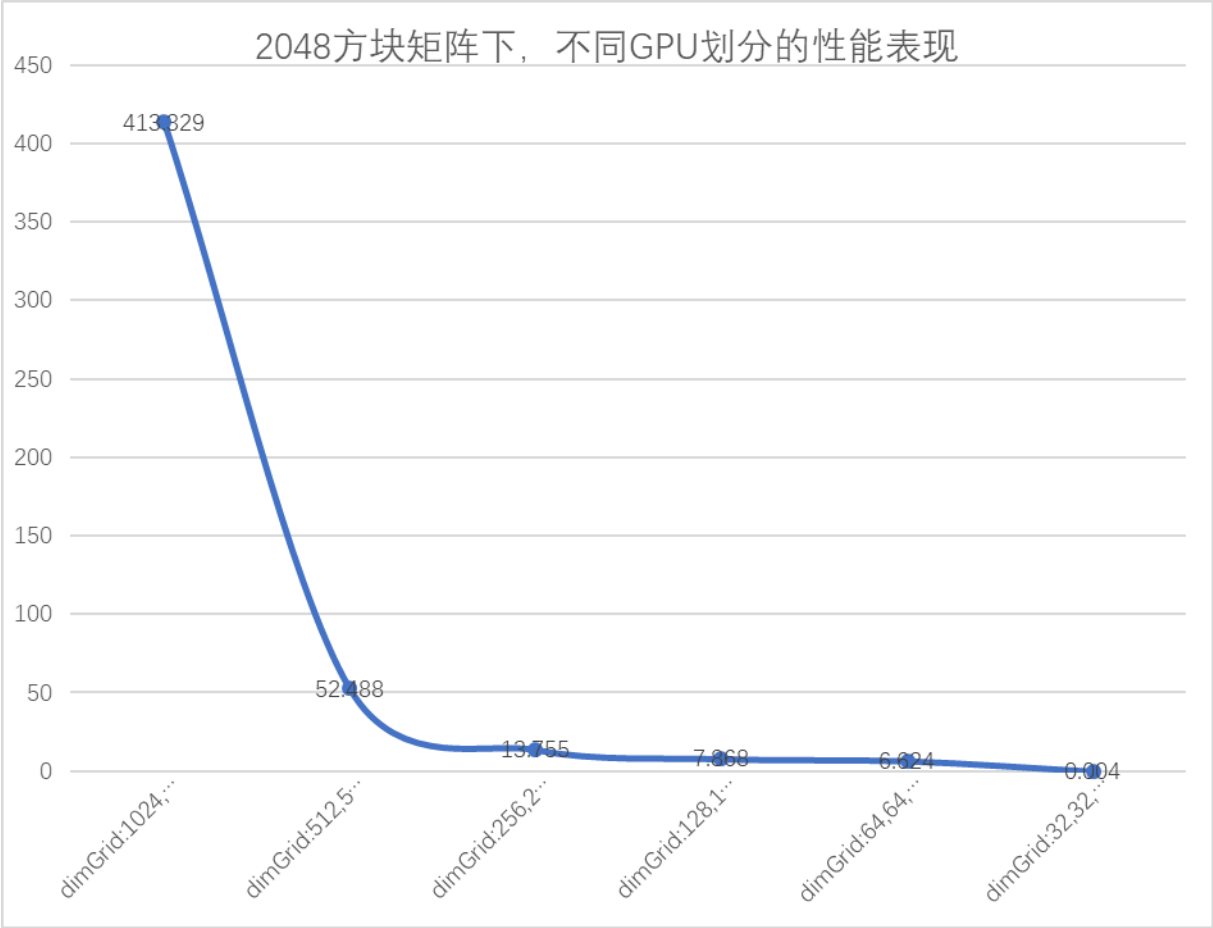
设置环境变量调整处理器数目，并得出相应的性能曲线。

实验数据

n1*n2:2048*2048
TILE_WIDTH:2
dimGrid:1024,1024,1
dimBlock:2,2,1
start of CUDA:978
end of CUDA and start of CPU:414307
cost:413.329

n1*n2:2048*2048
TILE_WIDTH:4
dimGrid:512,512,1
dimBlock:4,4,1
start of CUDA:932
end of CUDA and start of CPU:53420
cost:52.488

```
n1*n2:2048*2048
TILE_WIDTH:8
dimGrid:256,256,1
dimBlock:8,8,1
start of CUDA:992
end of CUDA and start of CPU:14747
cost:13.755

n1*n2:2048*2048
TILE_WIDTH:16
dimGrid:128,128,1
dimBlock:16,16,1
start of CUDA:980
end of CUDA and start of CPU:8848
cost:7.868

n1*n2:2048*2048
TILE_WIDTH:32
dimGrid:64,64,1
dimBlock:32,32,1
start of CUDA:1064
end of CUDA and start of CPU:7688
cost:6.624

n1*n2:2048*2048
TILE_WIDTH:64
dimGrid:32,32,1
dimBlock:64,64,1
start of CUDA:982
end of CUDA and start of CPU:986
cost:0.004
```



分析

由实验数据可见,CUDA 的网格划分方法对算法性能影响很大,在划分为 dimGrid:32,32,1 dimBlock:64,64,1 的时候表现最好，效果十分显著，相对于串行算法也有很大的提升。随着每个 Block 内存在的进程数量的增加，矩阵被划分地更加细致，但是总的线程的个数是不变的，这样一来开辟的共享存储个数也就越多，每个块内进程同步的用时也相对较少，如此划分，能很好地利用 CUDA 共享内存的并行算法设计。

任务二

请实现一个单核串行版本，然后将 CUDA 版本的输出结果跟串行版本对比，看是否一致。

串行实现

```
    final_sum = 0;
```

```
    for (int i = 0; i < DATA_SIZE; i++) {
        final_sum += fcpu(data[i]);
    }
    printf("CPUsum: %d \n", final_sum);
```

结果一致 为-439

规约求和采取了什么同步措施，如果不采取这些措施会有什么后果，请分析原因？

使用了同步__syncthreads();。如果不使用这个措施结果会不一致，在实现规约求和前和规约求和的每一步规约都要使用同步。第一次不同步，有可能导致比较快的进程读取先进入规约求和，导致读取到一些不正确的数据，有可能数据还没计算出来，此时为 0，导致结果出错；第二次不同步影响更大，会导致规约求和结果出现错误，而且这种错误是连锁影响的，比较快的线程实际上使用了未更新的数据进行规约求和。

小结

与其他同学（或本人）实现的相同算法的 OpenMP 并行实验结果进行对比和分析

在实验中遇到的问题以及解决方法

在本次实验中主要遇到的问题是 CUDA 编程的学习，作为初学者需要了解 CUDA 的基本原理同时要弄清楚 CUDA 划分 Grid Block 的思想和基础知识，入门学习是比较耗时的地方。另外，在对问题进行优化的过程中，如何使用__share__和__syncthreads();对算法进行优化也是比较难理解的地方，通常实验上用到的算法思路其实已经很常见，要把他们用 CUDA 实践并且理解 CUDA 代码的运作方式比较困难，最后通过阅读相关工具书和博客学习解决了上述困难。

程序中所采取的并行化方式与算法的详细说明

并行化算法的思路大致就是 PCMA 的设计思路，使用 CUDA 将问题划分，然后再每个子问题里面用串行的算法思路，最后将子问题归总，具体见上述。

比较 CUDA 算法和 CPU 串行算法的运行时间

在如下相同条件下提速 20 倍，实际上调整 dimGrid 和 dimBlock 应该能提速更多

指导教师评语及成绩

评语：

成绩：　　指导教师签名：

批阅日期：