实验五 CUDA 编程实验
PB13011079 杨智

实验目的

1. 理解 GPU 架构下数据级并行的思想。
2. 熟悉 GPU 下 CUDA 编程框架。

实验环境

操作系统:	ubuntu 16.04 LTS 64bit
编译器:	nvcc 7.5.17 (for cuda)
CPU:	Intel i7-4720HQ CPU @ 2.60GHz * 8
GPU:	NVIDIA GTX965M 2G GDDR5
memory:	8GB DDR3L

实验要求

1. 用 C 语言编写一个程序完成两个矩阵相乘。
2. 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]。思考:
   - blocksize代表了什么含义?
   - 限制该程序运行速度的瓶颈是什么?

   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**。思考:
   - 块的大小应该如何分配才能充分利用每个block的共享显存?
   - 块过大会导致什么问题?
   - 如果矩阵大小任意，应该如何修改?
3. 分别比较三个程序在不同规模下的计算性能，将结果绘制成图表并进行分析。
4. 你可以再加入其它的算法来实现矩阵相乘，并进行结果分析，属于加分项。

实验报告

- 算法的核心思想
- 算法在CUDA平台上的实现
  - 算法的并行性，在CUDA上并行性又如何实现
  - Grid、block的设置，shared memory如何分配
  - .....
- 算法在CUDA平台上的优化
- 实验结果分析：
  - 在某种条件(Grid、block设置等等)下，Kernel函数的执行时间。
  - 在一定条件下(Grid、block设置等等)，与其他方法比较（不同算法间、CPU和GPU、优化前优化后）。

实验结果：

yangzhi@yangzhi-CP65S:~/Desktop/pc/cuda$ for i in {1..5}; do ./a.out 500; done

sequential matrix multiplication: 0.410527sec

parallel matrix multiplication without using Tiles: 0.067675sec

parallel matrix multiplication using Tiles: 0.00512sec

speedup without using Tiles: 6.06615

speedup using Tiles: 80.1811

check parallel result without using Tiles:

the sequential result and parallel result are equal

check parallel result using Tiles:

the sequential result and parallel result are equal

sequential matrix multiplication: 0.38543sec

。。。。。。。。。。。（略）

算法的核心思想

串行矩阵乘法的**naive method**：

```
void main(){

   define A, B, C

   for i = 0 to M do
      for j = 0 to N do
         /* compute element C(i,j) */
         for k = 0 to K do
                  C(i,j) <= C(i,j) + A(i,k) * B(k,j)
         end
      end
   end
}
```

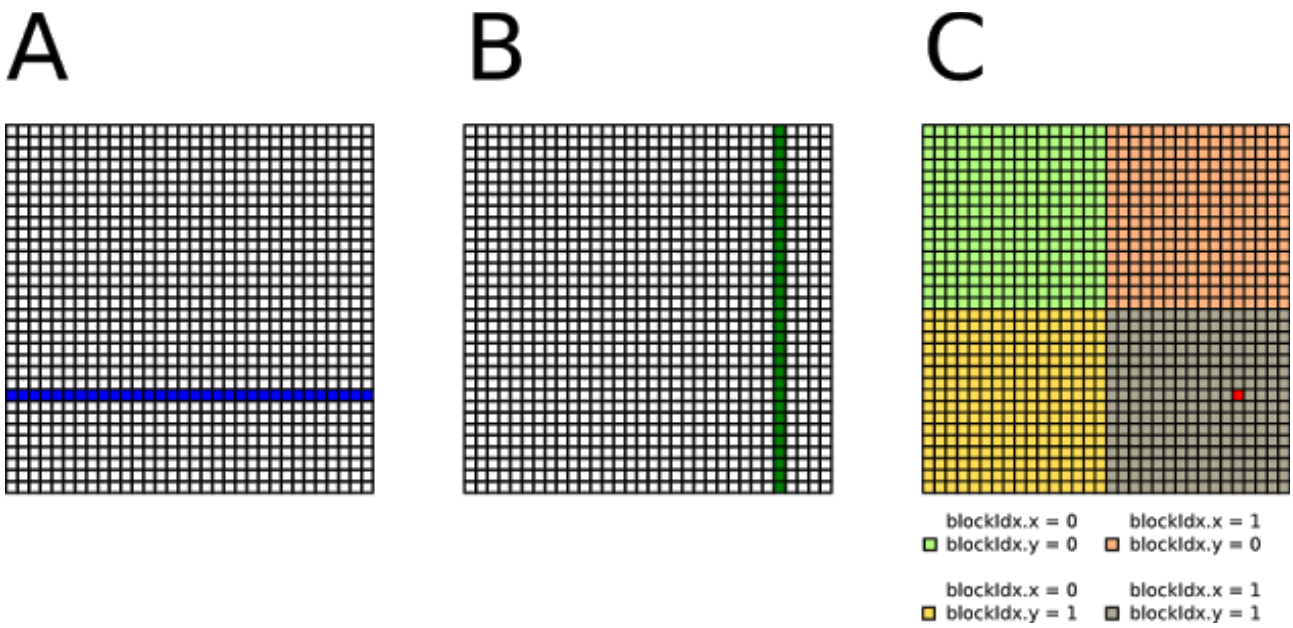
不需要解释

GPU上实现的**naive**矩阵乘法算法
每个**thread**计算**C**中的一个元素

```
/* Codes running on CPU */

void main(){

   define A_cpu, B_cpu, C_cpu in the CPU memory
   define A_gpu, B_gpu, C_gpu in the GPU memory

   memcopy A_cpu to A_gpu
   memcopy B_cpu to B_gpu

   dim3 dimBlock(16, 16)
   dim3 dimGrid(N/dimBlock.x, M/dimBlock.y)

   matrixMul<<<dimGrid, dimBlock>>>(A_gpu,B_gpu,C_gpu,K)

   memcopy C_gpu to C_cpu

}
```

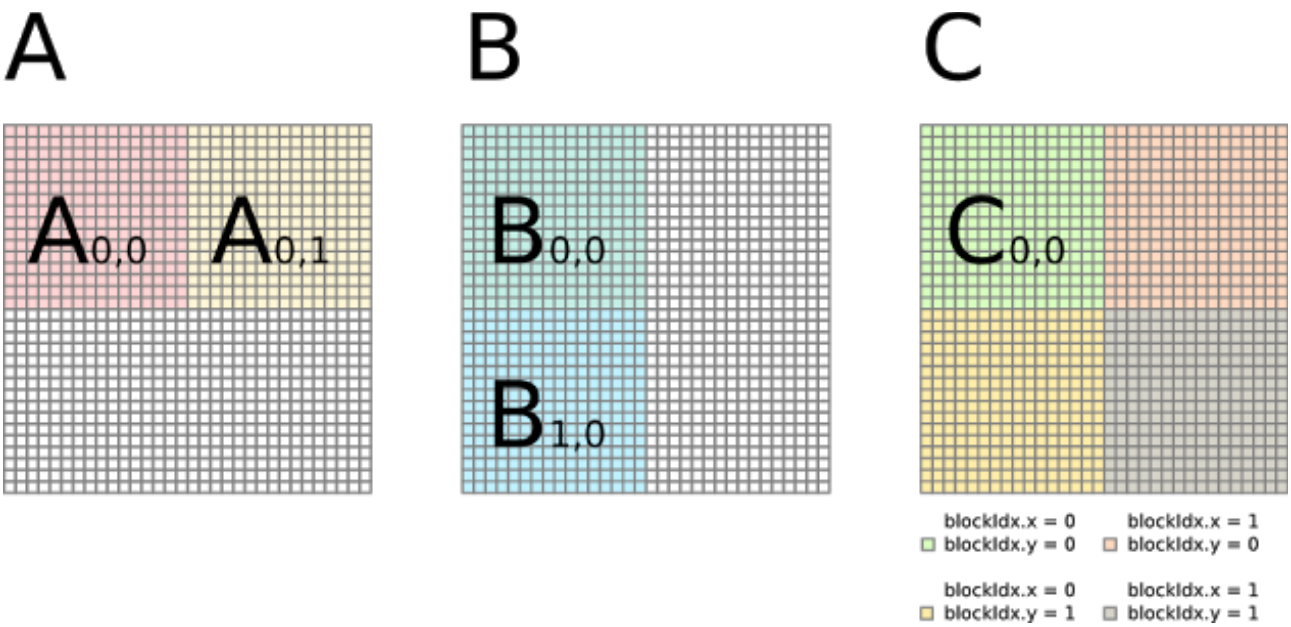
```
/* Codes running on GPU */

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){

    temp <= 0

    i <= blockIdx.y * blockDim.y + threadIdx.y    // Row i of matrix C
    j <= blockIdx.x * blockDim.x + threadIdx.x    // Column j of matrix C

    for k = 0 to K-1 do
        accu <= accu + A_gpu(i,k) * B_gpu(k,j)
    end

    C_gpu(i,j) <= accu

}
```

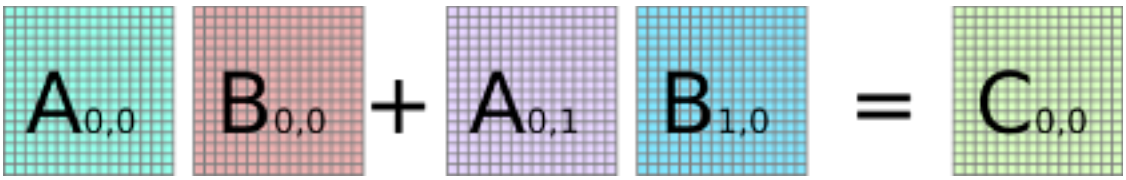
A naive implementation on GPUs assigns one thread to compute one element of matrix C. Each thread loads one row of matrix A and one column of matrix B from global memory, do the inner product, and store the result back to matrix C in the global memory. The figure shows the memory footprint of one thread on global memory where matrix A, B, and C are stored.



In the naive implementation, the amount of computation is 2 x M x N x K flop, while the amount of global memory access is 2 x M x N x K word. The "computation-to-memory ratio" is approximately 1/4 (flop/byte). Therefore, the naive implementation is bandwidth bounded.

在该程序的基础上编写一个分块（**tiled**）的矩阵乘法






```
/* Codes running on GPU */

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){

    __shared__ float A_tile(blockDim.y, blockDim.x)
    __shared__ float B_tile(blockDim.x, blockDim.y)

    accu <= 0

    /* Accumulate C tile by tile. */

    for tileIdx = 0 to (K/blockDim.x - 1) do

        /* Load one tile of A and one tile of B into shared mem */

        // Row i of matrix A
        i <= blockIdx.y * blockDim.y + threadIdx.y
        // Column j of matrix A
        j <= tileIdx * blockDim.x + threadIdx.x
        // Load A(i,j) to shared mem
        A_tile(threadIdx.y, threadIdx.x) <= A_gpu(i,j)
        // Load B(j,i) to shared mem
        B_tile(threadIdx.x, threadIdx.y) <= B_gpu(j,i) // Global Mem Not coalesced
        // Synchronize before computation
        __sync()

        /* Accumulate one tile of C from tiles of A and B in shared mem */

        for k = 0 to threadDim.x do
```

```
            // Accumulate for matrix C
            accu <= accu + A_tile(threadIdx.y,k) * B_tile(k,threadIdx.x)
        end
        // Synchronize
        __sync()

    end

    // Row i of matrix C
    i <= blockIdx.y * blockDim.y + threadIdx.y
    // Column j of matrix C
    j <= blockIdx.x * blockDim.x + threadIdx.x
    // Store accumulated value to C(i,j)
    C_gpu(i,j) <= accu

}
```

In the tiled implementation, the amount of computation is still 2 x M x N x K flop. However, using tile size of B, the amount of global memory access is 2 x M x N x K / B word. The "computation-to-memory ratio" is approximately B/4 (flop/byte). We now can tune the "computation-to-memory" ratio by changing the tile size B.

问题回答

对于第一种方法：每个thread计算C中的一个元素

1. blocksize代表了什么含义?

blocksize表示每个二维**block**的宽度（在这个题目中）

2. 限制该程序运行速度的瓶颈是什么?

如上面**“**算法的核心思想**”**中所说：**the naive implementation is bandwidth bounded.**

对于第二种方法：在该程序的基础上编写一个分块（tiled）的矩阵乘法

1. 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。

2. **块过大会导致什么问题?**

   块太大会导致shared memory不够用

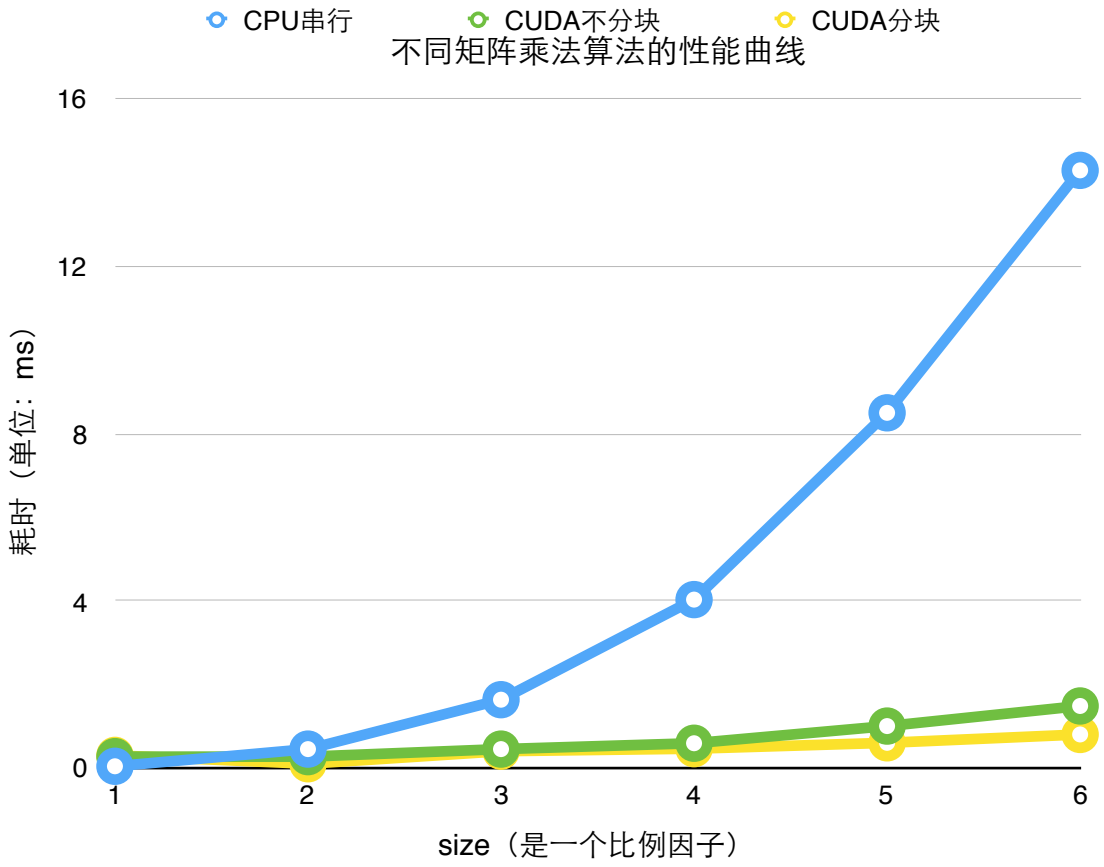
3. **如果矩阵大小任意，应该如何修改?**

   通过在矩阵中填充0，来使矩阵的长宽一致。

计算性能比较1

两个矩阵的大小：A:(size*32,size*32)，B:(size*32,size*2*32)

size 算法	1	2	3	4	5	6
CPU串行	0.05ms	0.46ms	1.64ms	4.03ms	8.49ms	14.28ms
CUDA不分块	0.28ms	0.28ms	0.46ms	0.60ms	1.01ms	1.49ms
CUDA分块	0.31ms	0.11ms	0.41ms	0.47ms	0.61ms	0.81ms



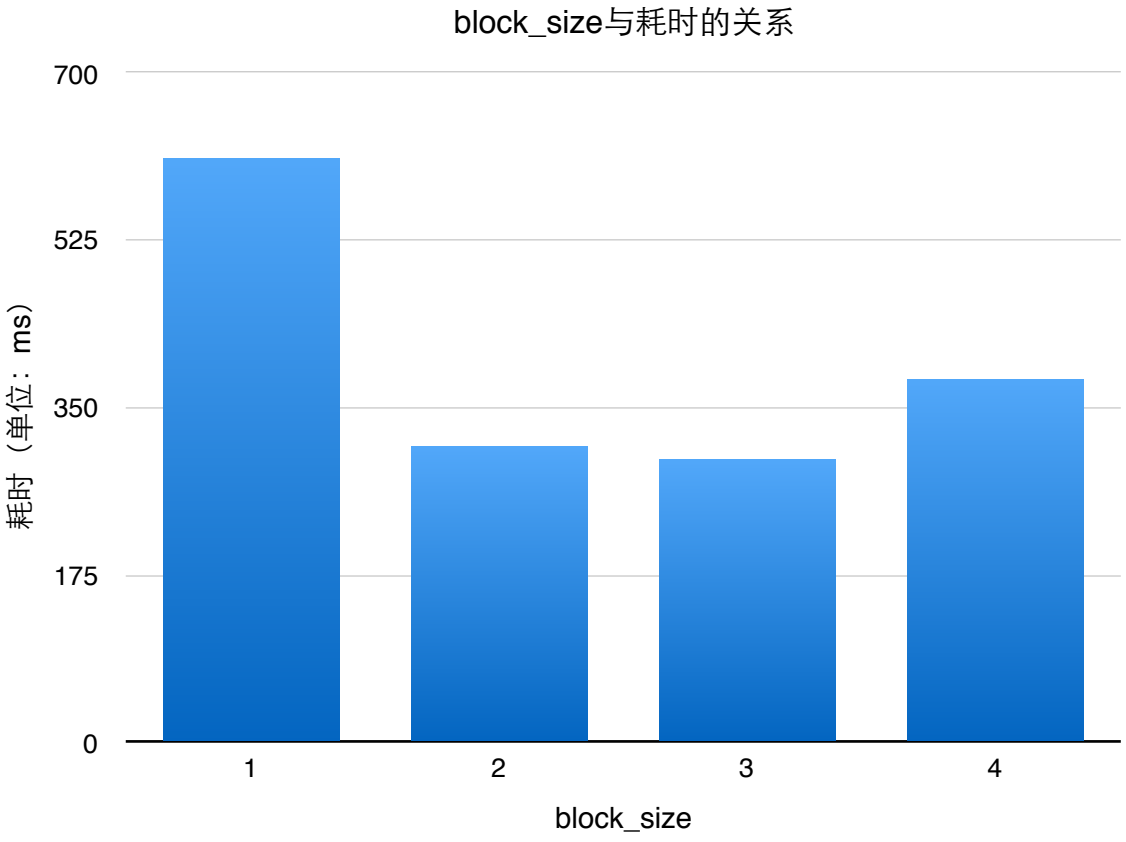
实验结论1:

由上面的图表可以看出，“分块的GPU并行”优于“不分块的GPU并行”优于“CPU串行”。而且CPU串行的性能极低。

计算性能比较2

两个矩阵的大小：A:(size*32,size*32)，B:(size*32,size*2*32)

当矩阵size一定的时候，耗时与block_size有如下关系：



实验结论2:

block_size取太大或者取太小都不合适，要根据设备的硬件配置和具体问题来选择block_size。

实验总结:

CUDA很厉害，
以后问题适合并行时要尽可能用GPU进行加速（例如训练神经网络时）。
