并行计算 Lab3：GPU

1. 实验简介

CUDA™ 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决通用的计算问题。

Linux 下编译：`nvcc cuda.cu`

2. 预备知识

2.1 CUDA 中的线程层次

线程、线程块 Block、线程组 wrap、线程网络。

- Thread id:
  a) local id: thread id in a block
  b) global id: thread id in a grid
- Compute thread global id : blockDim*blockId+threadId
- Each thread uses IDs to decide what data to work on
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks cannot cooperate

2.2 同步

CPU 和 GPU 之间的同步

CPU 启动内核 kernel 是异步的，即当 CPU 启动 GPU 执行 kernel 时，CPU 并不等待 GPU 完成就立即返回，继续执行后面的代码。

如果 CPU 在接下来的操作中需要用到 GPU 的计算结果，则 CPU 必须阻塞等待 GPU 执行完毕。可在 kernel 后添加一条同步语句实现。

同一个 block 内的同步

同一个 block 内的线程可以通过 shared memory 共享数据。

`__syncthreads()` 用于实现同一个块内线程的同步.

不同 block 之间的同步

同一个 grid 中的不同线程块之间不能同步，但可以通过终止一个kernel来实现同步。

2.3 限定符

函数限定符

函数限定符	在何处执行	从何处调用	特性
device	设备	设备	函数的地址无法获取
global	设备	主机	返回类型必须为空
host	主机	主机	等同于不使用任何限定符

变量限定符

限定符	位于何处	可以访问的线程	主机访问
device	全局存储器	线程网格内的 所有线程	通过运行时库访问
constant	固定存储器	线程网格内的 所有线程	通过运行时库访问
shared	共享存储器	线程块内的 所有线程	不可从主机访问

2.4 内核函数

内核函数是特殊的一种函数，是从主机调用设备代码唯一的接口，相当于显卡环境中的主函数。

内核函数使用 `__global__` 函数限定符声明，返回值为空

调用内核函数需要使用 `KernelName<<<>>>()` 的方式

`<<<>>>` 内的参数用于指定执行内核函数的配置，包括线程网格，线程块的维度，以及需求的共享内存大小，例如 `<<<DimGrid, DimBlock, MemSize>>>`

- DimGrid（dim3类型），用于指定网格的两个维度，第三维被忽略
- DimBlock （dim3类型），指定线程块的三个维度
- MemSize （size_t类型），指定为此内核调用需要动态分配的共享存储器大小

2.5 运行时 API

- `cudaMalloc()`： 分配线性存储空间
- `cudaFree()`： 释放分配的空间
- `cudaMemcpy()`：内存拷贝

计时：

```
unsigned int timer = 0;
CUT_SAFE_CALL(cutCreateTimer(&timer)); //定义计时器
cudaThreadSynchronize();
CUT_SAFE_CALL(cutStartTimer(timer)); //计时器启动
CudaKernel<<<dimGrid, dimBlock, memsize>>>(); //GPU计算
cudaThreadSynchronize(); //等待计算完成
CUT_SAFE_CALL(cutStopTimer(timer) ); //计时器停止
float timecost=cutGetAverageTimerValue(timer); //获得计时结果
printf("CUDA time %.3fms\n",timecost);
```

3. 向量加法

3.1 核心代码

定义 A，B 两个一维数组，编写 GPU 程序将 A 和 B 对应项相加，将结果保存在数组 C 中。分别测试数组规模为 10W、20W、100W、200W、1000W、2000W 时其与 CPU 加法的运行时间之比。

核心代码如下：

设备端函数：

```
__global__ void ArrayAdd(int *A, int *B, int *C, int N){
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < N) {
        C[index] = A[index] + B[index];
    }
}
```

主机端函数核心代码：（省略了本地的数据开辟、数组初始化和计时器，完整代码请见附件）

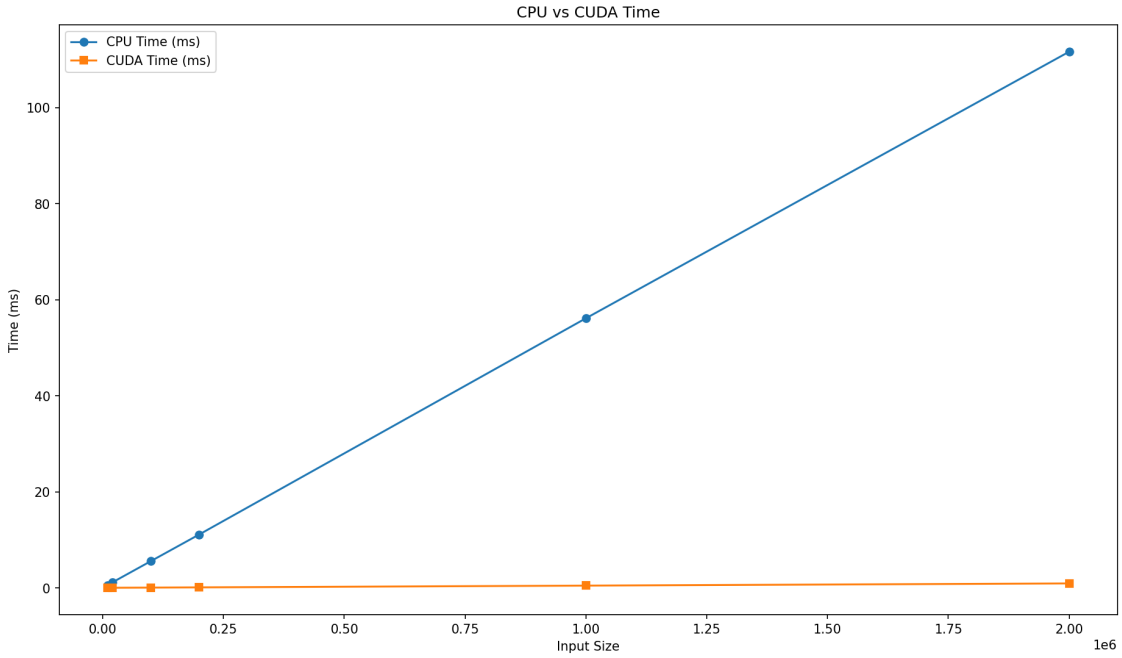
```
int *CUDA_A, *CUDA_B, *CUDA_C;
cudaMalloc(&CUDA_A, N * sizeof(int));
cudaMalloc(&CUDA_B, N * sizeof(int));
cudaMalloc(&CUDA_C, N * sizeof(int));
cudaMemcpy(CUDA_A, A, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(CUDA_B, B, N * sizeof(int), cudaMemcpyHosttoDevice);
ArrayAdd<<<blocks_num, THREAD_NUM, 0>>>(CUDA_A, CUDA_B, CUDA_C, N);
```

3.2 运行结果

如果**仅统计计算所用时间**，即使用 GPU 时，不统计 cudaMalloc 和 MemCpy 的时间：

列表如下：（单位：毫秒 ms）

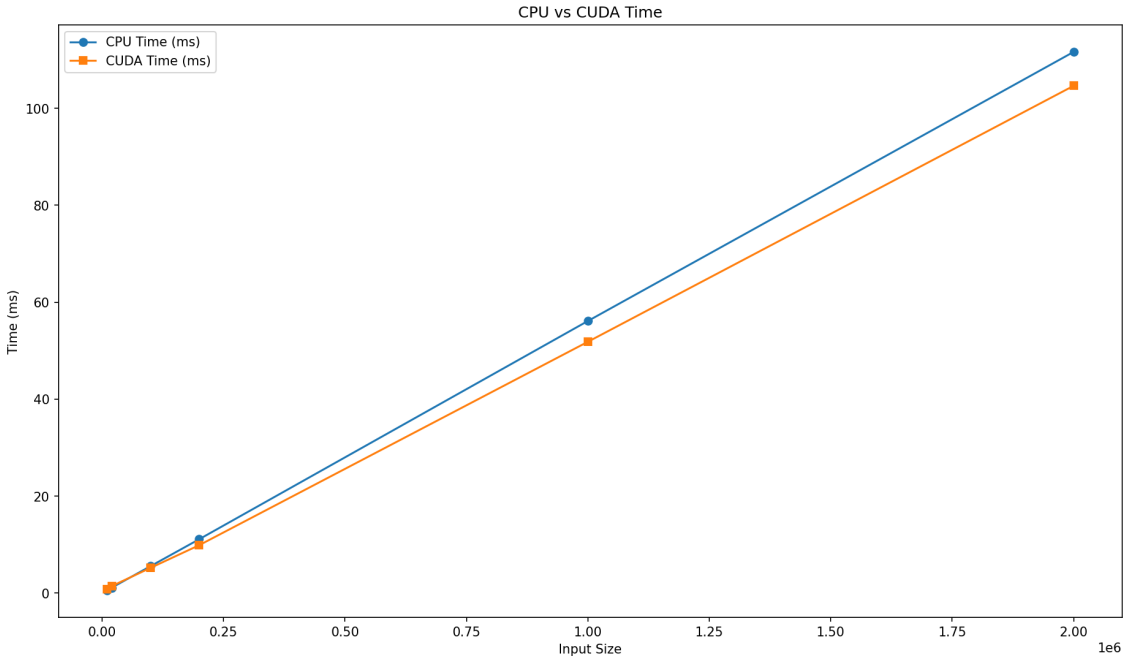
数据规模	10W	20W	100W	200W	1000W	2000W
串行	0.5660	1.1260	5.5710	11.1200	56.1330	111.6590
GPU	0.0153	0.0184	0.0529	0.0992	0.4638	0.9203
运行时间之比	36.9	61.2	105.3	112.1	121.0	121.3



可以看出，使用 GPU 加速的效果非常明显。

但是在实际工程使用中，cudaMalloc 和 MemCpy 过程必要的，它们所占用的无法省去，那么**考虑分配设备端现存和内存拷贝**， 运行时间如下：

数据规模	10W	20W	100W	200W	1000W	2000W
串行	0.5660	1.1260	5.5710	11.1200	56.1330	111.6590
GPU	0.7796	1.4518	5.2209	9.8913	51.8465	104.6543
运行时间之比	0.72	0.77	1.07	1.12	1.08	1.07



可以发现，GPU 加速版本与 CPU 串行版本的时间差别较小，甚至在数据规模较小时，GPU加速后的运行时间反而更长。

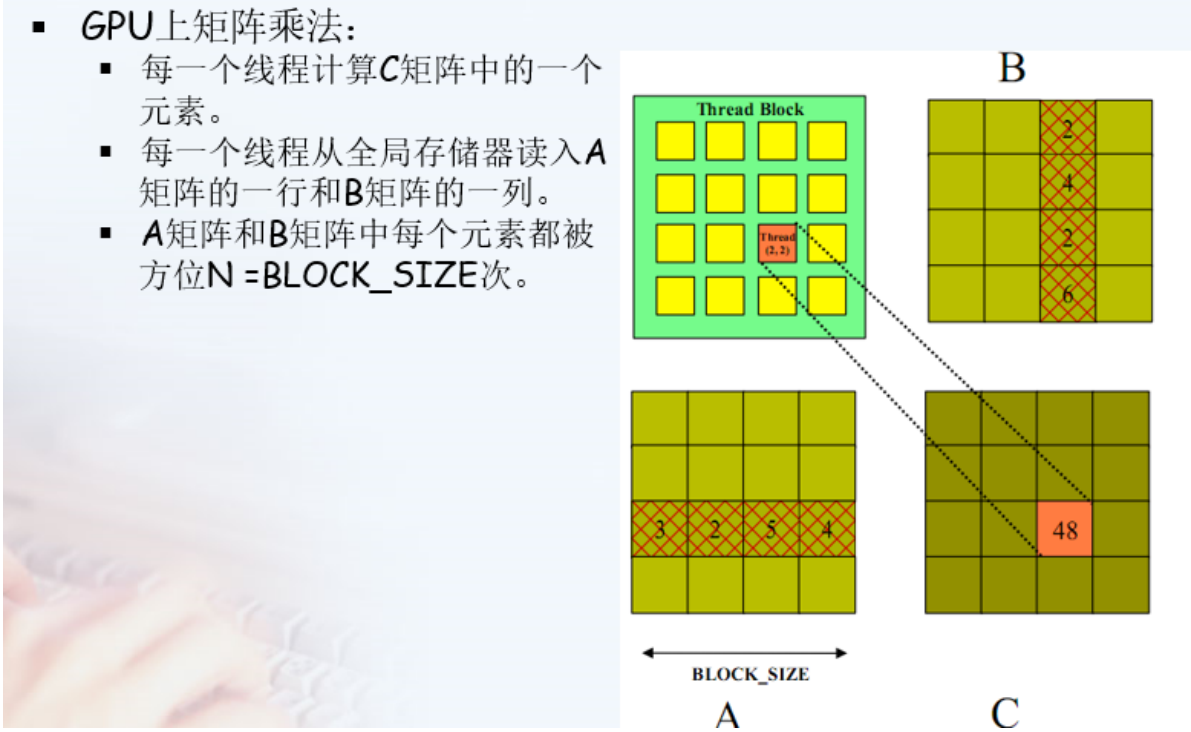
这是因为 GPU 并行计算的优势主要在于处理大规模数据时可以**大幅减少计算时间**。但是 GPU 加速版本会额外多出启动开销和数据传输开销，上面我们就把数据传输开销考虑在内，所以统计出的时间会变长。

这也能得到一个结论，那就是对于大规模数据和计算密集型任务，GPU 加速通常能显著减少计算时间。而对于数据量较小，计算任务轻，或者需要大量数据传输的任务，GPU 加速效果就不明显。

4. 矩阵乘法

4.1 实验题目

定义 A，B 两个二维数组。使用 GPU 实现矩阵乘法。并对比串行程序，给出加速比。



4.2 核心代码

设备端函数：

```
__global__ static void CUDAkernal(const float* a, const float* b, float* c, int n)
{
    //block内的threadID
    const int tid = threadIdx.x;
    //blockID
    const int bid = blockIdx.x;
    //全局threadID
    const int idx = bid * THREAD_NUM + tid;
    const int row = idx / n;
    const int column = idx % n;
    //计算矩阵乘法
    if (row < n && column < n)
    {
        float t = 0;
        for (int i = 0; i < n; i++)
        {
            t += a[row * n + i] * b[i * n + column];
```

```
        }
        c[row * n + column] = t;
    }
}
```

主机端函数核心代码，即开辟数组，初始化矩阵，分配设备端显存、将矩阵从内存复制到显存、调用设备端函数，将结果从显存中复制回内存，最后释放空间。其间统计运行时间。

4.3 运行结果

如果**仅统计计算所用时间**，即使用 GPU 时，不统计 cudaMalloc 和 MemCpy 的时间：

```
root@kkeGPU:~/Parallel-Computing-Labs/Lab3# ./MatrixS
CPU time 4423.2910ms
root@kkeGPU:~/Parallel-Computing-Labs/Lab3# ./MatrixG
CUDA time 0.0832ms
```

加速比大概为：53746，可以看到加速效果是很明显的。

但是在实际工程使用中，cudaMalloc 和 MemCpy 过程必要的，它们所占用的无法省去，那么**考虑分配设备端现存和内存拷贝**， 运行时间如下：

```
root@kkeGPU:~/Parallel-Computing-Labs/Lab3# ./MatrixS
CPU time 4425.2370ms
root@kkeGPU:~/Parallel-Computing-Labs/Lab3# ./MatrixG
CUDA time 5.2517ms
```

加速比约为 842.6，可以看出，即使考虑数据传输，加速效果仍然是很明显的。

5. 附录

- `PB20061343_徐奥_实验三.pdf`：实验报告
- `ArrayAddS.c`：向量加法 CPU 串行版本
- `ArrayAddG.cu`：向量加法 GPU 并行版本
- `MatrixS.c`：矩阵乘法 CPU 串行版本
- `MatrixG.c`：矩阵乘法 GPU 并行版本