CUDA Lab

CUDA（Compute Unified Device Architecture）是显卡厂商 NVIDIA 推出的运算平台，是一种通用的并行计算架构，该架构使 GPU 能够解决复杂的计算问题。我们可以使用 GPU 来并行例如神经网络、图像处理等在 CPU 上运行起来比较耗时的程序，通过 GPU 并行计算可以大大提高算法的运行速度。

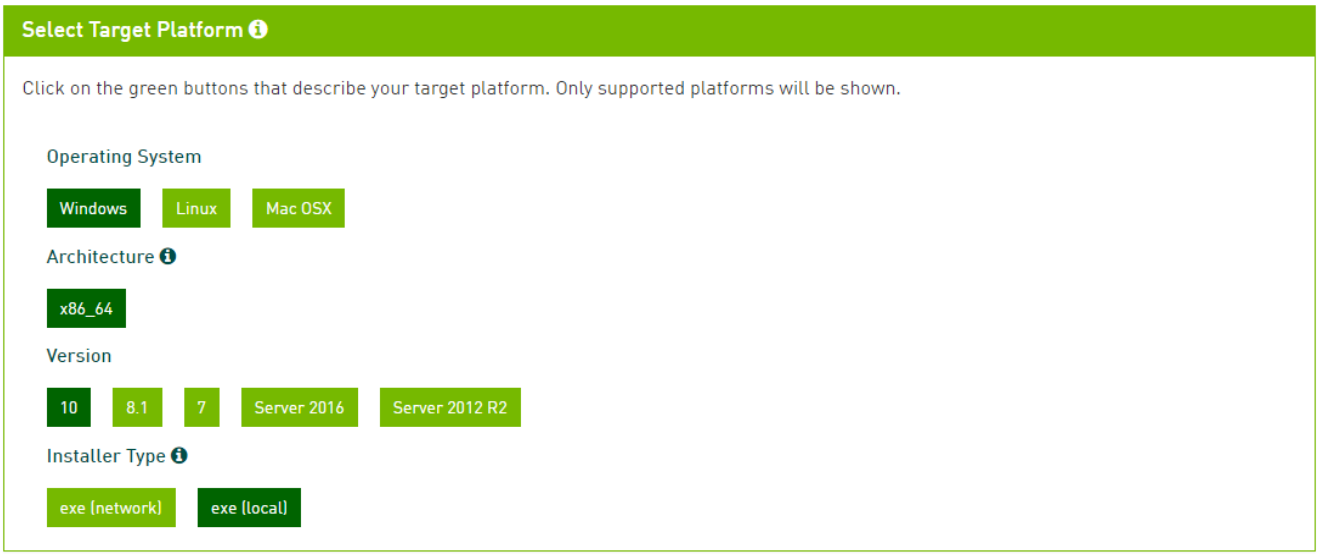
本实验包含两部分。

- 第一部分（Sum of Squares）：计算数组元素的平方和，并通过不断的优化来提高程序的性能，以此来学习和理解 CUDA 编程需要注意之处。
- 第二部分（Matrix Multiplication）：利用 CUDA 实现矩阵乘法并行化。

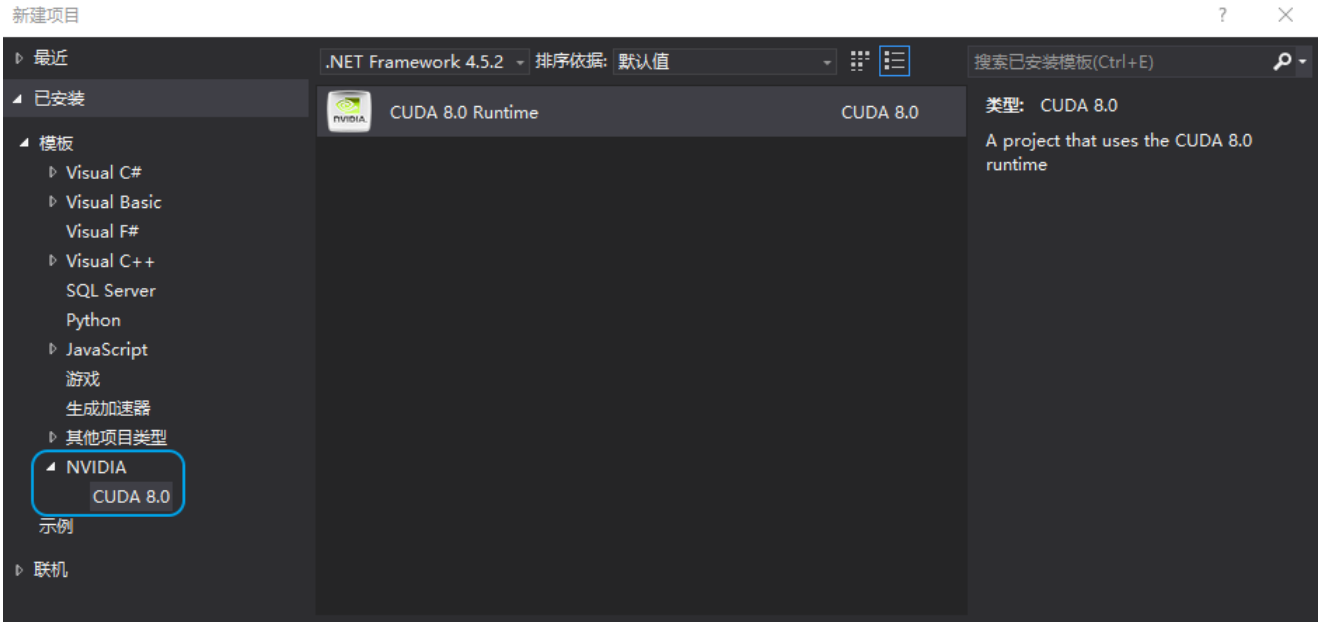
CUDA安装

- 推荐 Visual Studio 2015 Community Download，CUDA Tookit 与 Visual Studio 2017 Community 暂不兼容。
- CUDA Tookit Download。

下载 CUDA Tookit 时选择相应的操作系统及版本等平台参数。例如，在 Win10 下进行实验，参数选择如下。

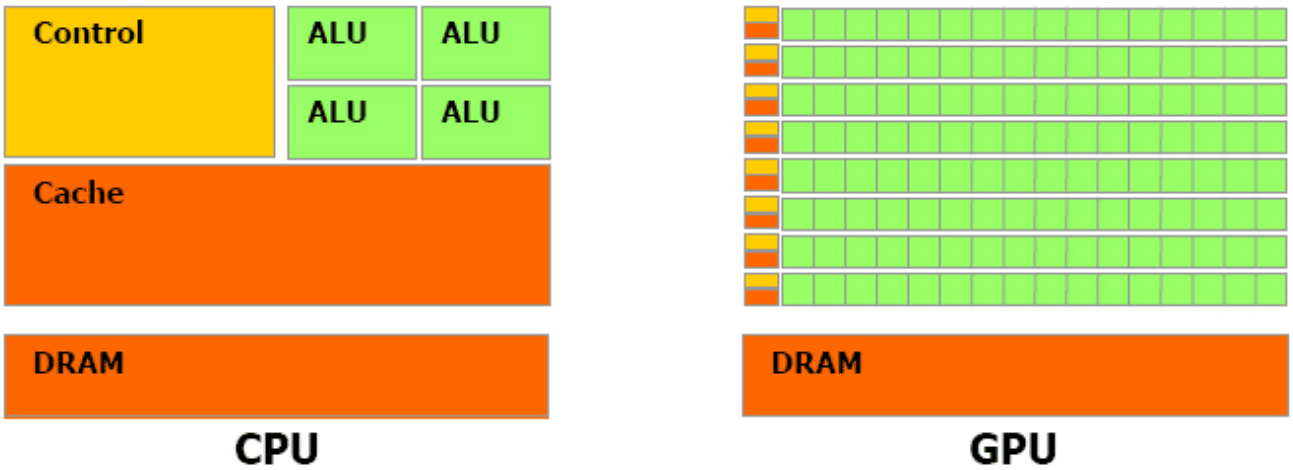


注意：应先安装 **Visual Studio** 再安装 **CUDA Tookit**，因为在安装 **CUDA Tookit** 时会自动配置 **CUDA for Visual Studio**，**CUDA Tookit** 安装成功后可以在 **Visual Studio** 中创建 **CUDA Project**。



CPU & GPU

下图为 CPU 和 GPU 的对比图，相比于 CPU，GPU 更加适用于计算强度高，多并行的计算中。GPU 拥有更多的晶体管，而不是数据 Cache 和控制器，这样设计的意图是在并行计算过程中，每个数据单元经常执行相同的程序，不需要繁琐的复杂流程控制和 Cache，而更需要强大的计算能力。

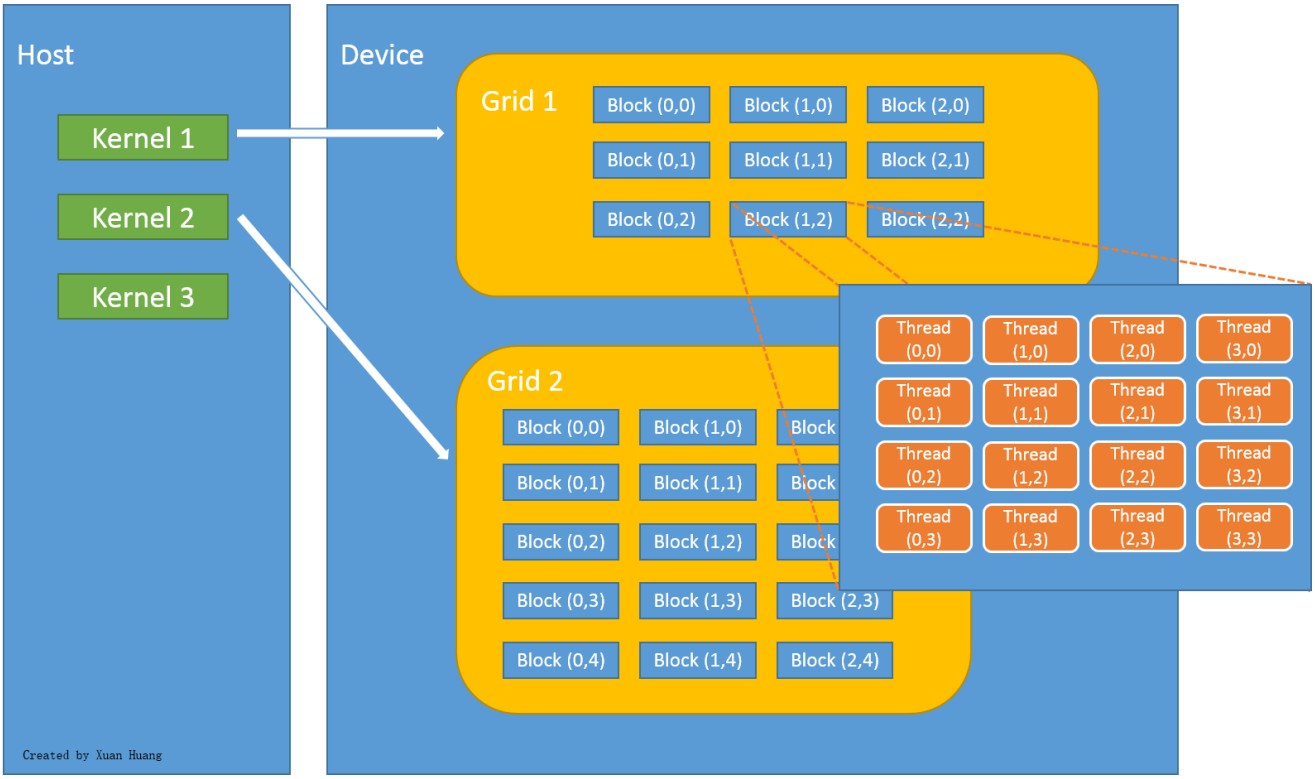


使用 GPU 来进行运算工作，和使用 CPU 相比，主要有以下优势。

- GPU 通常具有更大的内存带宽。例如，NVIDIA 的 GeForce 8800GTX 具有超过50GB/s 的内存带宽，而目前高阶 CPU 的内存带宽则在 10GB/s 左右。
- GPU 具有更大量的执行单元。例如 GeForce 8800GTX 具有 128 个 "stream processors"，频率为 1.35GHz。CPU 频率通常较高，但是执行单元的数目则要少得多。
- 和高阶 CPU 相比，GPU 的价格较为低廉。例如目前一张 GeForce 8800GT 包括512MB 内存的价格，和一颗 2.4GHz 四核心 CPU 的价格相当。

CUDA架构

在 CUDA 架构下，一个程序被划分为两个部分：Host 端和 Device 端。Host 端是指运行在 CPU 的部分，而 Device 端则是在 GPU 上运行的部分，如下图所示。



在程序执行过程中，通常 Host 端程序会将数据在主内存中准备好并复制到显存，再由 GPU 执行 Device 端程序，完成后由 Host 端程序将结果从显存拷贝至主内存。由于 CPU 存取显存时只能通过 PCI Express 接口，因此速度较慢（PCI Express x16 的理论带宽是双向各 4 GB/s），因此不能太频繁的执行这类操作，以免降低效率。

在 CUDA 架构下，GPU 执行时的最小单元是 Thread，数个 Thread 可以组成一个 Block，一个 **Block** 中的 **Thread** 能够存取同一块共享的内存，而且可以快速进行同步操作。每一个 Block 所能包含的 Thread 数目是有限的，但是执行相同程序的 Block 可以组成 Grid。不同 Block 中的 Thread 无法存取同一个共享内存，因此无法进行通信和同步。

由于 GPU 具备大量适用于并行计算的特性，因此 GPU 处理问题的方式和 CPU 是不同的，主要体现在以下两点。

- 内存存取 latency 的问题：CPU 通常使用 Cache 开减少存取主内存的次数，以避免内存 latency 影响到执行效率。GPU 通常没有 Cache（或很小），其利用并行化的方式来隐藏内存的 latency（当一个 Thread 需要等待内存读取时，开始执行另一个 Thread）。
- 分支指令的问题：CPU 通常利用分支预测等方式来减少分支指令造成的 pipeline bubble。GPU 则多半使用类似处理内存 latency 的方式。不过，通常 GPU 处理分支的效率会比较差。

Section 1: Sum of Squares

目标：对于给定数组，利用 CUDA 编程实现对该数组所有元素的平方进行求和，并计算每次求和所消耗的时间及显存带宽。

我们首先实现在 GPU 上的串行求和程序，然后将串行程序并行化，并从两个维度对并行程序进行优化。

Step 1: CUDA初始化

首先实验需要使用 CUDA 的 RunTime API，所以需要引入头文件 `cuda_runtime.h`。

```
// CUDA Runtime API
#include <cuda_runtime.h>
```

编写程序对当前环境进行检测，如果存在支持 CUDA 的设备，需要设置相应的设备。如下所示。

```
/* CUDA初始化 */
bool initCUDA() {
    int count, i;
    // 取得支持CUDA的装置的数目
    cudaGetDeviceCount(&count);

    if (0 == count) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }

    for (i = 0; i < count; i++) {
        cudaDeviceProp prop;
        if (cudaGetDeviceProperties(&prop, i) == cudaSuccess) {
            if (prop.major >= 1) {
                break;
            }
        }
    }

    if (i == count) {
        fprintf(stderr, "There is no device.\n");
        return false;
    }

    cudaSetDevice(i);

    return true;
}
```

这段程序首先调用`cudaGetDeviceCount`函数获取支持 CUDA 的设备的数量，如果不存在支持 CUDA 的设备，则会传入 1（Device 0），Device 0 只是一个仿真设备，CUDA 的很多功能都不支持，因此如果要真正的确定是否存在支持 CUDA 的设备，需要对每个设备调用`cudaGetDeviceProperties`函数来获取具体的参数。

Step 2: CUDA核函数

在介绍核函数前，通过一段程序创建待求和的数组，如下所示。

```
/* 产生0-9之间的随机数 */
void generateNumbers(int *numbers, int size) {
    int i;
    for (i = 0; i < size; i++) {
        numbers[i] = rand() % 10;
    }
}
```

如果要进行求和计算，需要将生成的数组从主内存拷贝至显存。因此，我们需要在显存开辟一块合适的空间，然后将数组拷贝至显存空间，相关代码如下。

```
int *gpudata, *result;

generateNumbers(data, DATA_SIZE);

// 在显存上分配空间
// 思考：为什么cudaMalloc函数原型的第一个参数类型为 (void **)？
// 原因：gpudata指向某块内存区域的首地址，cudaMalloc在显存中分配一块内存，然后将该内存区域的首地址
//       赋值给gpudata，因此cudaMalloc修改的是gpudata本身的值，而不是gpudata指向的内存区域的值。
cudaMalloc((void**)&gpudata, sizeof(int) * DATA_SIZE);
cudaMalloc((void**)&result, sizeof(int));

// 将数据从内存复制到显存
cudaMemcpy(gpudata, data, sizeof(int) * DATA_SIZE, cudaMemcpyHostToDevice);
```

在完成主内存和显存的数据拷贝后，开始着手编写核函数实现在 GPU 上的求和计算，核函数在编写时需要在函数返回值（`void`）加添加`__global__`，同时，核函数不允许有返回值。目前我们仅实现串行求和，其核函数如下所示。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *result) {
    int sum, i;
    sum = 0;
    for (i = 0; i < DATA_SIZE; i++) {
        sum += numbers[i] * numbers[i];
    }
    *result = sum;
}
```

Step 3: 执行核函数

在 CUDA 中，使用如下规则执行核函数。

```
函数名称<<<Block Num, Thread Num, Shared Memory Size>>>(参数...);
```

因为目前我们仅需要实现串行计算，因此设置Block Num = 1，Thread Num = 1，Shared Memory Size = 0，相关程序如下所示。同时，在执行核函数后不要忘记释放程序分配的显存。

```
sumOfSquares << < 1, 1, 0 >> > (gpudata, result);
// 把计算结果从显存复制到内存
cudaMemcpy(&sum, result, sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(gpudata);
cudaFree(result);
```

Step 1，Step 2 和 Step 3 的完整程序`sum_squares_1.cu`见 GitHub 。

Step 4: 评估程序表现

在介绍 CUDA 架构时提到，GPU 通常没有 Cache（或很小），因此我们在编写程序时需要避免内存 latency 影响到执行效率。本实验通过核函数的运行时间来计算程序所使用的显存带宽，以此作为评估程序性能表现的标准之一。

对`sum_squares_1.cu`进行部分改动：利用`clock`函数获取当前时间，时间差值即为核函数的运行时间，相关代码如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *result, clock_t *time) {
    int sum, i;
    clock_t start, end;

    // 获取起始时间
    start = clock();

    sum = 0;
    for (i = 0; i < DATA_SIZE; i++) {
        sum += numbers[i] * numbers[i];
    }
    *result = sum;

    // 获取结束时间
    end = clock();

    *time = end - start;
}
```

由于核函数不能有返回值，因此需要在显存开辟一块空间来保存运行时间，然后将该运行时间值拷贝至内存。注意，该时间值的单位是时钟周期，我们需要将消耗的时钟周期数除以 GPU 自身的频率得到以秒为单位的时间值。

对于显存带宽的计算，整个过程程序传输的数据量大小为 4MB，将其除以时间得到程序运行过程中占用的显存带宽。

Step 4 的完整程序`sum_squares_2.cu`见 GitHub。

Step 5: 串行程序并行化

在 CUDA 中，数据是从主内存复制到显存的 Global Memory，而Global Memory 是没有 Cache 的 ，因此存取 Global Memory 所需要的时间是非常长的（通常有数百个时钟周期）。由于之前的程序只有一个线程，当线程读取 Global Memory 时，线程会等待至实际数据读取成功，才能进行下一步计算。

如果程序有多个线程，当其中一个线程在等待读取 Global Memory 时，GPU 可以立即切换至另一个线程。因此，通过增加线程的数量来提高显存的带宽是有效的策略。

首先在程序中定义线程的数量，如下所示。

```
// 线程数
#define THREAD_NUM 256
```

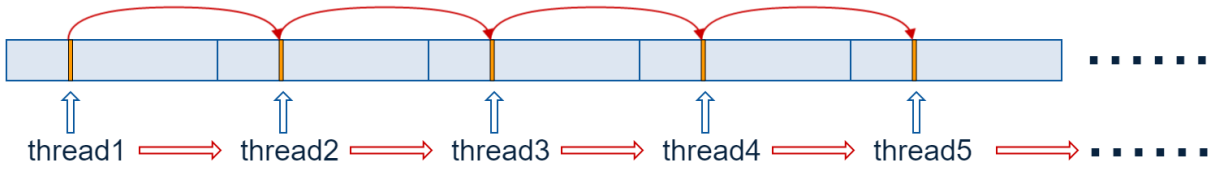
程序将数组划分为`THREAD_NUM`段，每个线程负责计算其中一段。最后，CPU 负责将各个线程计算得到的子结果进行累加。相关代码如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *sub_sum, clock_t *time) {
    int i;
    // 获取当前线程Id（从0开始）
    const int thread_id = threadIdx.x;
    // 每个线程累加元素的个数
    const int size = DATA_SIZE / THREAD_NUM;

    sub_sum[thread_id] = 0;
    for (i = thread_id * size; i < (thread_id + 1) * size; i++) {
        sub_sum[thread_id] += numbers[i] * numbers[i];
    }
}
```

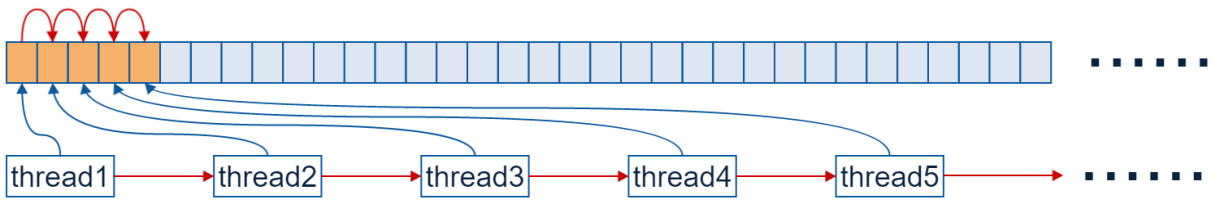
Step 5 的完整程序`sum_squares_3.cu`见 GitHub。

Step 6: 改进显存存取模式

显卡内存一般都是 DRAM，因此最有效的显存存取方式为连续存取。对于`cuda_sample_3`，虽然每个线程操作的都是一块连续的内存，但是考虑当一个线程等待 Global Memory 的数据时，GPU 切换至另一个线程，而另一个线程会从显存的其它位置存取数据，下图描述了上述过程。



综上分析，虽然在同一个线程中是在一块连续的显存空间读取数据，但是在实际执行时并不是连续读取的，而是跳跃式的存取模式。因此，我们需要将显存的存取模式改进为连续存取模式，如下图所示。



修改核函数，改进显存的存取模式，相关程序如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *sub_sum, clock_t *time) {
    int i;
    const int thread_id = threadIdx.x;

    sub_sum[thread_id] = 0;
    // 线程0获取第0个元素，线程1获取第1个元素，以此类推...
    for (i = thread_id; i < DATA_SIZE; i += THREAD_NUM) {
        sub_sum[thread_id] += numbers[i] * numbers[i];
    }
}
```

Step 6 的完整程序`sum_squares_4.cu`见 GitHub。

Step 7: 进一步并行

在介绍 CUDA 架构时提到，CUDA 除了提供了 Thread，还提供了 Block 及 Grid等重要的机制。每个 Block 的数量是有限的，但是可以通过增加 Block 的数量来成倍的增加线程的数量。需要注意的是，不同 Block 内的线程相互不能同步和通信，不过在我们的程序中线程之间并不需要进行同步或通信。因此，Step 7 将会使用多个 Block 来进一步并行化程序。相关代码如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *sub_sum, clock_t *time) {
    int i;
    // 获取当前线程所属的Block号（从0开始）
    const int block_id = blockIdx.x;
    const int thread_id = threadIdx.x;

    sub_sum[block_id * THREAD_NUM + thread_id] = 0;
    // Block0-线程0获取第0个元素，Block0-线程1获取第1个元素...Block1-线程0获取第THREAD_NUM个元素，以此类推...
    for (i = block_id * THREAD_NUM + thread_id; i < DATA_SIZE; i += BLOCK_NUM * THREAD_NUM) {
        sub_sum[block_id * THREAD_NUM + thread_id] += numbers[i] * numbers[i];
    }
}
```

Step 7 的完整程序`sum_squares_5.cu`见 GitHub。

Step 8: 共享内存和线程同步

在 Step 7 中，CPU 需要进行 `BLOCK_NUM * THREAD_NUM` 个元素的累加，如果可以让 GPU 执行一部分的累加，理应能够进一步提高程序的并行度。因为在一个 Block 内，线程是可以共享内存的，因此可以在 GPU 上实现一个 Block 内的线程结果累加，CPU 仅需要完成各个 Block 的计算结果的累加。

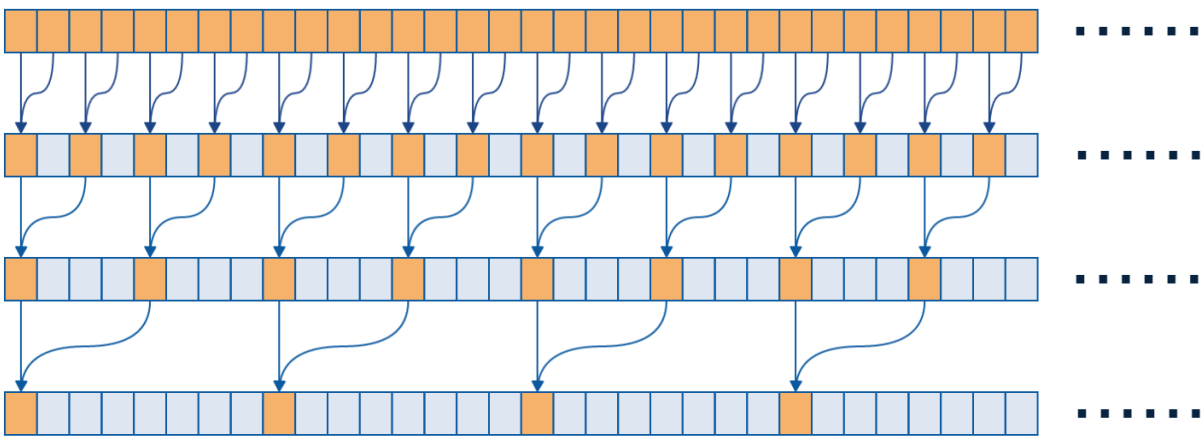
需要注意：需要等待一个 Block 内的所有线程都完成计算才进行各个线程的计算结果的类型。因此，程序需要进行线程的同步，在 CUDA 中，通过`__syncthreads`函数进行线程的同步，相关代码如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *sub_sum, clock_t *time) {
    int i;
    // 声明共享内存区域，用于存储每个Block中线程计算结果的累加和
    extern __shared__ int shared[];

    const int block_id = blockIdx.x;
    const int thread_id = threadIdx.x;

    shared[thread_id] = 0;

    for (i = block_id * THREAD_NUM + thread_id; i < DATA_SIZE; i += BLOCK_NUM * THREAD_NUM) {
        shared[thread_id] += numbers[i] * numbers[i];
    }

    // 线程同步，所有线程需要执行到此处方可继续向下执行
    __syncthreads();

    // 线程0负责计算所有线程的计算结果累加和
    if (0 == thread_id) {
        for (i = 1; i < THREAD_NUM; i++) {
            shared[0] += shared[i];
        }
        sub_sum[block_id] = shared[0];
    }

}
```

Step 8 的完整程序`sum_squares_6.cu`见 GitHub。

Step 9: 树状加法

在 Step 8 中，一个 Block 中所有线程的结果的累加是通过一个线程（Thread 0）实现的，如果能够把每个 Block 内的加法并行化，那么程序的并行度应该能够进一步提高。我们利用树状加法将加法并行，树状加法计算过程如下图所示。



上图一个格子代表一个线程的计算结果，当进行第一轮迭代时，步长等于 1，Thread 0 和 Thread 1 相加，Thread 2 和 Thread 3 相加，依次类推...当进行第二轮迭代时，步长等于 1 + 1 = 2，Thread 0 和 Thread 2 相加，Thread 4 和 Thread 6 相加，依次类推...当进行第三轮迭代时，步长等于 2 + 2 = 4， Thread 0 和 Thread 4 相加，Thread 8 和 Thread 12 相加...依次类推。相关代码如下。

```
/* 计算平方和（__global__函数运行于GPU）*/
__global__ static void sumOfSquares(int *numbers, int *sub_sum, clock_t *time) {
    int i;

    extern __shared__ int shared[];

    const int block_id = blockIdx.x;
    const int thread_id = threadIdx.x;
    // 定义步长和计算掩码
    int offset, mask;

    shared[thread_id] = 0;
    for (i = block_id * THREAD_NUM + thread_id; i < DATA_SIZE; i += BLOCK_NUM * THREAD_NUM) {
        shared[thread_id] += numbers[i] * numbers[i];
    }

    __syncthreads();

    /* 并行加法代码段 */
    offset = 1;
    mask = 1;
    while (offset < THREAD_NUM) {
        // 注意 & 的优先级小于 ==
        if ((thread_id & mask) == 0 && thread_id + offset < THREAD_NUM) {
            shared[thread_id] += shared[thread_id + offset];
        }
        offset += offset;
        mask += offset;
        // 每迭代一轮需要所有线程进行一次同步
        __syncthreads();
    }

    sub_sum[block_id] = shared[0];
}
```

Step 9 的完整程序`sum_squares_7.cu`见 GitHub。

Section 2: Matrix Multiplication

目标：利用 CUDA 编写矩阵乘法并行程序，为了简化程序编写，假设待相乘的两个矩阵均为方块矩阵。

在 CUDA 上实现矩阵乘法有多种方法，我自己实现了一个版本，供参考。

实验要求

- 编程实现两部分实验，修改程序参数（如线程的数量），对程序性能变化的原因进行分析，撰写实验报告（附程序运行截图）。

- 实验报告发送至 tinylcy@gmail.com。