# 《并行计算》实验指导书

# 实验4：基于GPU加速器的异构并行编程

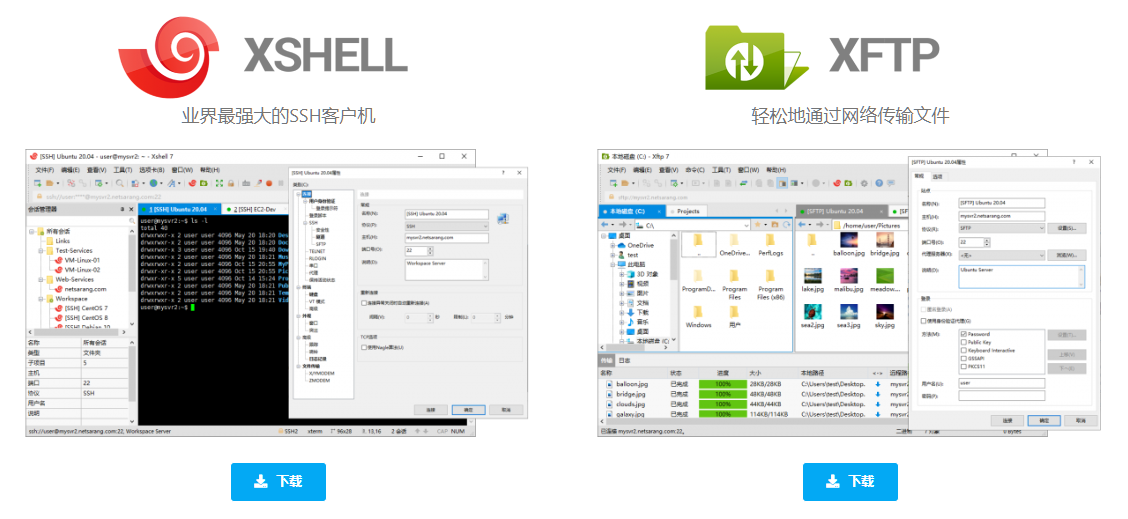
## 实验4.1 GPU实验环境说明

### 实验目的

了解远程访问服务器GPU的过程。

### 实验内容

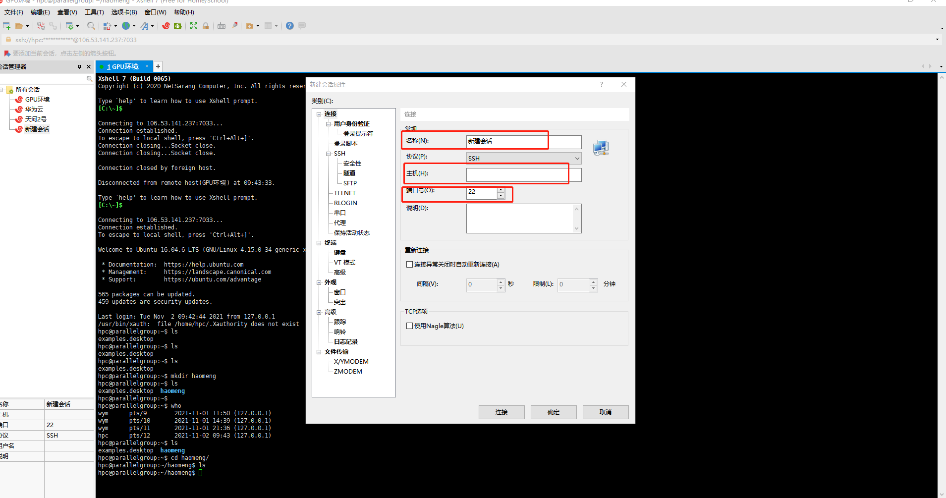
* 步骤一：下载并安装 XSHELL 和 XFTP
  + 打开浏览器，输入 <https://www.xshell.com/zh/free-for-home-school/>
  + 下载并安装 XSHELL 和 XFTP



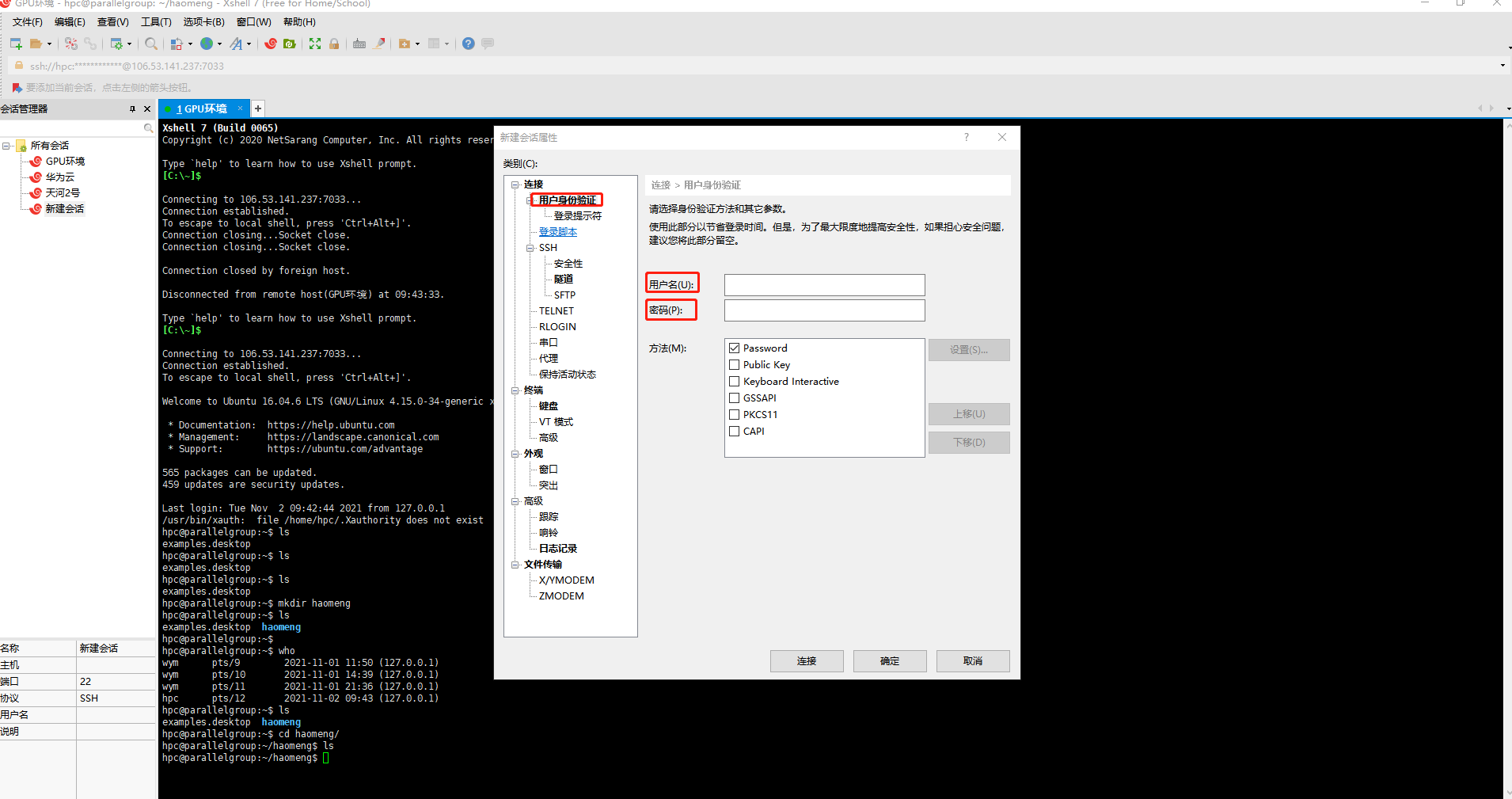
* + 打开软件后，在提示输入用户名和邮箱的窗口中，点击“以后”。



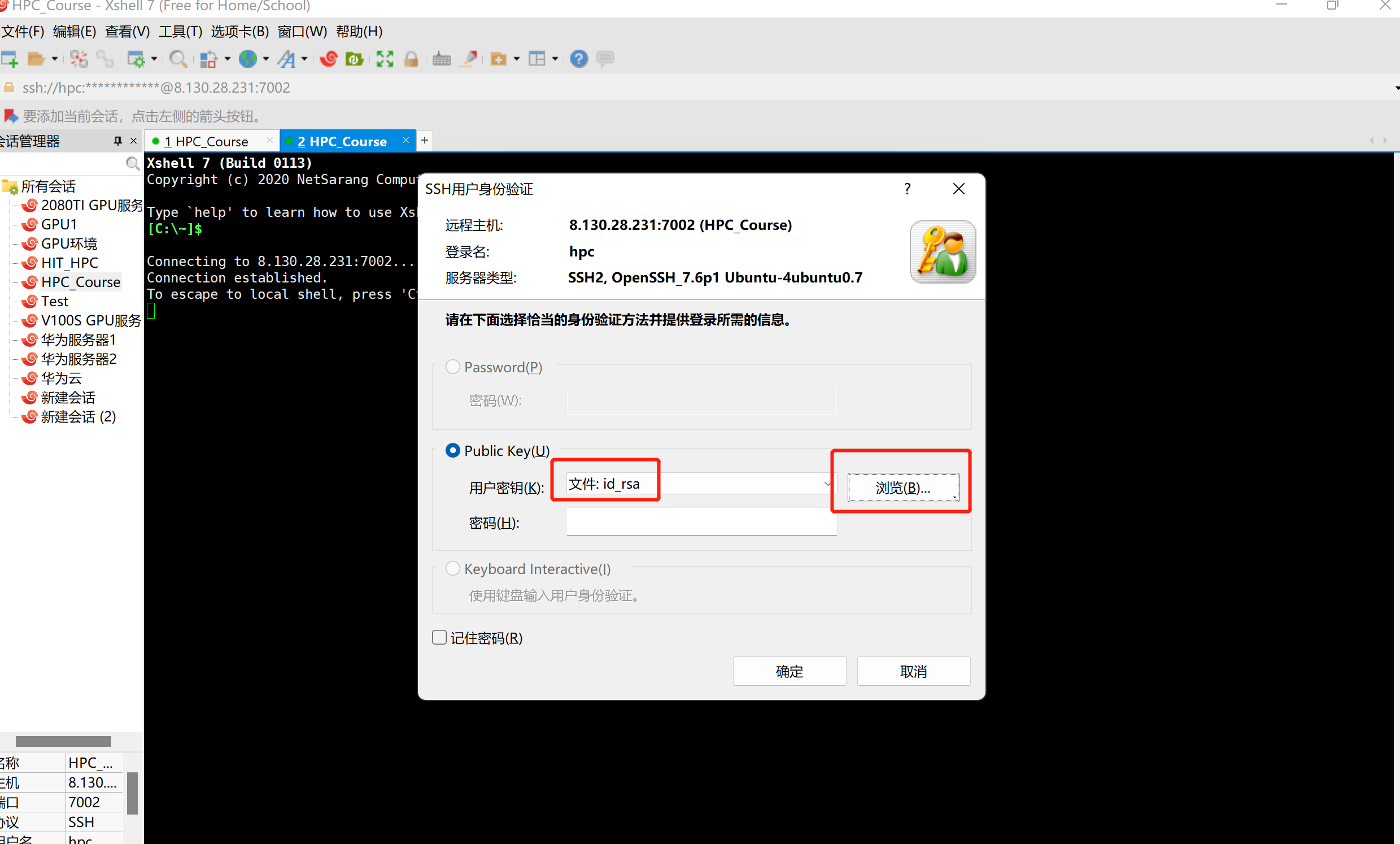
* 步骤二：使用 XSHELL 和 XFTP 远程连接服务器
  + 在 XSHELL 左上角，点击“**文件**”，在下拉菜单中，点击“**新建**”
  + 在弹出窗口中，填写“**名称**”、“**主机**”和“**端口号**”
    - 本次实验提供了两台服务器，它们分别搭载了Nvidia RTX2080Ti GPU 和 Nvidia Tesla V100S GPU。请任选其一完成实验。
    - 若使用RTX 2080 Ti 服务器，则输入以下信息：
      * 名称：GPUServer\_2080Ti
      * 主机：**8.130.28.231**
      * 端口号：**7001**
    - 若使用Tesla V100S 服务器，则输入以下信息：
      * 名称：GPUServer\_V100S
      * 主机：**8.130.28.231**
      * 端口号：**7002**



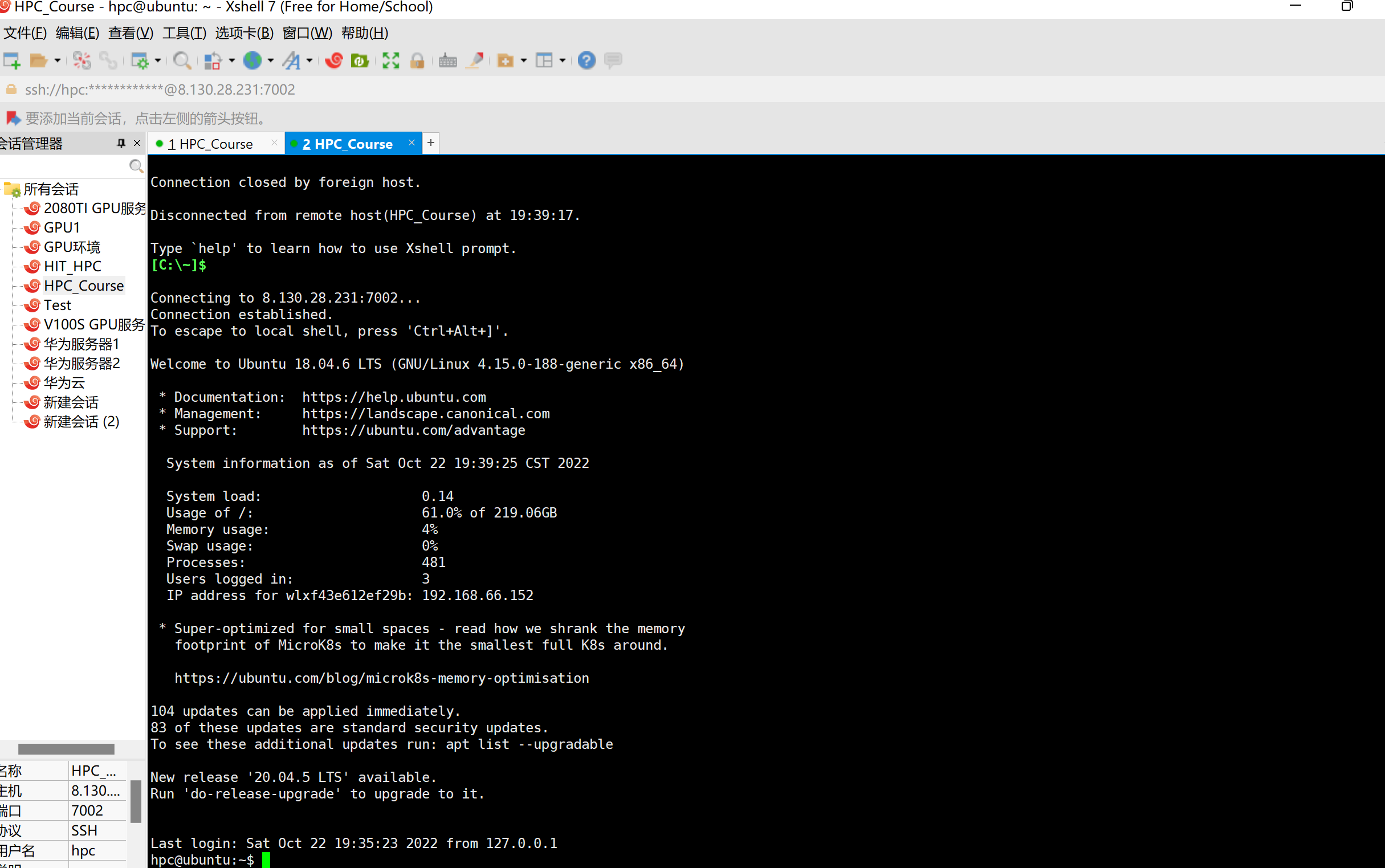
* + 之后，在列表左侧，点击“**用户身份验证**”。然后，输入用户身份验证信息，包括“**用户名**”、“**密码**”。之后点击窗口下方“**连接**”
    - 用户名：**hpc**
    - 密码：**parallel2022**



* + 在新弹出窗口中，载入账户密钥文件。点击“**浏览**”按钮
    - 若使用RTX 2080 Ti服务器，请使用**id\_rsa\_2080Ti**文件
    - 若使用Tesla V100S服务器，请使用**id\_rsa\_V100S**文件



* + 点击“**确定**”，连接服务器



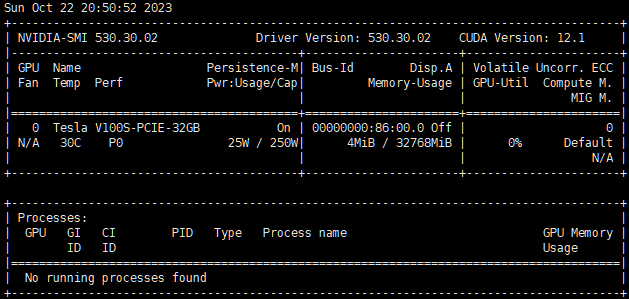
* 步骤三：在 XFTP 中，也添加相应的服务器信息和用户信息，载入对应的用户密钥，并连接服务器
* 步骤四：由于hpc账号为公用账号，所以请创建以自己名字命名的文件夹，程序的修改和编译在自己的文件夹中进行。**请勿使用拼音缩写**，以避免缩写重复问题。在 XSHELL 中，使用以下命令创建并切换到自己的文件夹：

mkdir haomeng(change this to your name) // create your directory

cd haomeng(change this to your name) // change to your directory

* 步骤五：通过以下命令查看 GPU 状态，并验证 GPU 是否可用。

nvidia-smi // check GPU status



* + 上半部分的表格中的重要参数说明：

1. **GPU**：表示该设备的识别序号（device ID）。若主机端连接了多个设备，则会显示0、1、2、3…
2. **Name**：表示设备名称，比如 Tesla V100S-PCIE-32GB。
3. **Fan**：表示设备风扇转速。
4. **Temp**：表示设备温度，单位为摄氏度。
5. **Perf**：表示设备性能状态。包含P0 ~ P12，其中P0表示最大性能，P12表示最小性能。
6. **Persistence-M**：表示“持续模式”的状态，虽然持续模式耗能大，但是在此模式下启动新的GPU应用时，花费的时间更少。
7. **Pwr**：表示设备当前能耗。
8. **Bus-Id**：与设备总线相关。
9. **Disp. A**：表示设备的显示是否初始化，全称为Display Active。
10. **Memory Usage**：表示显存 (GPU Memory) 使用率。显存（全局内存）为显卡内存结构层级中的重要组成部分。
11. **GPU-Util**：表示设备利用率，全称为GPU Utilization。Memory Usage不同于GPU Utilization。显卡由GPU和显存等组成，GPU和显存的关系类似于CPU与内存的关系。
12. **Uncorr. ECC**: 与内存错误检查相关。
13. **Compute M**：计算模式。
    * 下半部分的表格包含每个进程的信息及显存使用率。

## 实验4.2 CUDA程序的编译和运行

### 实验目的

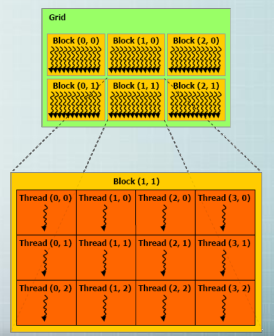
通过在GPU上编译运行CUDA程序，掌握CUDA并行计算的原理和执行过程。

### 实验内容

* CUDA简介

CUDA (Compute Unified Device Architecture) 是由 NVIDIA 公司创立的基于该公司生产的图形处理器 GPU (Graphics Processing Units) 的一个并行计算平台和编程模型。通过 CUDA，可以很方便地利用GPU进行通用并行计算（类似于在CPU中进行数值计算）。在 CUDA 发布前，一般只用 GPU 进行图形渲染（如通过 OpenGL 和 DirectX）。CUDA 发布后，开发人员可以通过调用 CUDA 的 API ，使用 GPU 完成并行编程，达到高性能计算的目的。

* CUDA中的常用术语
  + **主机** (host)：由 CPU、系统内存、IO 设备等构成。
  + **设备** (device)： 由 GPU 及显存等构成，通常合称为显卡。不同架构的设备具有不同容量的显存，比如，V100S 具有 32GB 的显存，而RTX 2080Ti 的显存容量为 11GB。
  + **线程** (thread)：一个 CUDA 程序会被许多线程执行，线程是软件层面的最小执行单元。一个线程一定对应一个 CUDA Core。程序执行时，线程被映射到数据上。
  + **线程块** (block)：由多个线程组成。线程块能包含的线程数量限制取决于 GPU 架构设计。所有线程块并行执行，不能互相通信。多个线程块可以被调度到相同的 Streaming Multiprocessor (SM) 上，线程块执行顺序由 SM 调度决定。
  + **线程格** (grid)：由多个线程块组成。线程格能包含的线程块数量限制取决于 GPU 架构设计。
  + 线程、线程块、线程格的逻辑结构如下所示：



* + **线程束** (warp)：在 CUDA 中，32个线程组成一个线程束。线程束是硬件调度的基本单位，其中的线程被“组合在一起”并且“步调一致”地执行。
* GPU内存分类
  + **寄存器** (registers)：在核函数中声明，且没有使用其他修饰符修饰的变量通常存放在寄存器中。寄存器通常存放被频繁访问的线程私有变量。这些变量与核函数的生命周期相同，核函数执行完毕后，就不能再访问它们了。寄存器是GPU中访问速度最快的内存空间，但是一个 SM 中的寄存器数量有限，一旦核函数使用的寄存器数量超过了内存硬件限制，则会使用本地内存来替代多占用的寄存器，这种寄存器溢出 (memory spilling)的情况会降低性能，实际编程时应避免这种情况。
  + **本地内存** (local memory)：在核函数中，符合存储在寄存器中，但不能分配到存储器中的变量，将被溢出到本地内存中。溢出到本地内存中的变量在本质上与全局内存位于用一块区域，因此访问速度很慢。
  + **共享内存** (shared memory)：在核函数中被\_\_shared\_\_修饰符修饰的变量位于共享内存中。每个SM都有一定数量由线程块分配的共享内存，它们在核函数内声明，生命周期伴随整个线程块，一个线程块执行结束后，为其分配的共享内存也被释放以便重新分配给其他线程块使用。线程块中的线程通过使用共享内存中的数据可以实现互相协作通信。不过，使用共享内存后，必须通过以下函数进行同步：

void \_\_syncthreads();

该函数为线程块中的所有线程设置了一个执行路障点，必须等待线程块中的所有线程都执行到该路障点后，才能继续执行，这样就可以避免一些潜在的数据冲突。

* **常量内存** (constant memory): 常量内存用\_\_constant\_\_修饰符修饰，它们必须在全局空间内和所有核函数之外进行声明，对同一编译单元中的核函数都是可见的。常量变量存储在常量内存中，核函数只能从常量内存中读取数据，常量内存必须在主机端代码中使用以下函数进行初始化：

cudaError\_t cudaMemcpyToSymbol(

const void\* symbol,

const void\* src,

size\_t count);

以下的示例展示了如何声名常量内存并与之进行数据交换：

\_\_constant\_\_ float const\_data[256];

float data[256];

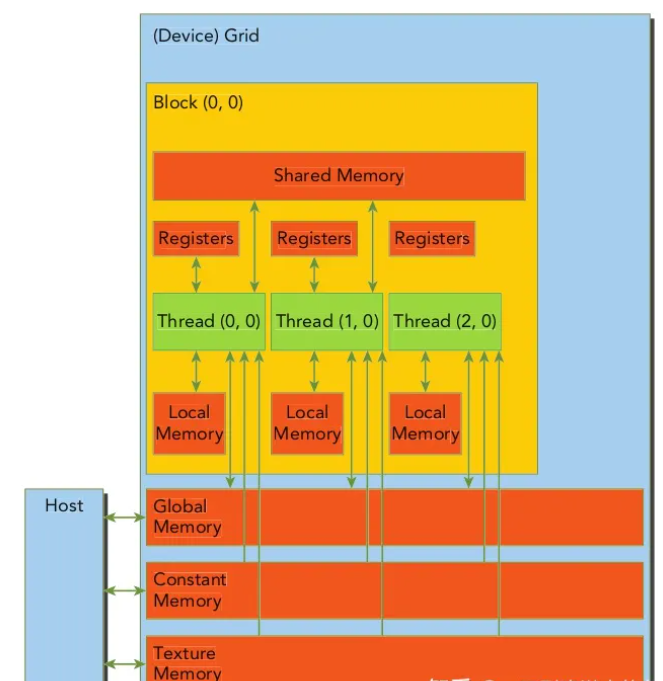
cudaMemcpyToSymbol(const\_data, data, sizeof(data));

cudaMemcpyFromSymbol(data, const\_data, sizeof(data));

常量内存适合用于线程块中的所有线程都需要从相同的内存地址中读取

数据的情况，比如所有线程都需要的常量参数。

* **全局内存** (global memory)：也称为设备内存 (device memory)。全局内存是GPU中容量最大、延迟最高的内存空间，其作用域和生命周期都是全局的。一个全局内存变量可以在主机端代码中使用cudaMalloc函数进行动态声明，或者使用\_\_device\_\_修饰符在设备端代码中静态声明。全局内存变量可以在任何 SM 设备中被访问，其生命周期贯穿整个应用程序的整个生命周期。**在 CUDA 编程中，一般情况下，设备端的核函数不能访问主机端声明的变量，主机端的函数不能直接访问设备端的变量，即使它们是在同一个文件内声明的。**
  + 纹理内存 (texture memory)：存储在全局内存中。面向内存访问具有空间聚簇性特征的程序(例如图像处理方面的计算程序)设计，互相临近的线程读取的数据在物理存储上也是临近的，因此可以减少访存次数，节约带宽，提升效率。纹理内存分为一维与二维：一维纹理内存的声明方式是texture<类型>，使用cudaBindTexture()函数绑定纹理内存，cudaUnbindTexture()函数解除绑定，使用tex1D()函数读取内存数据；二维纹理内存的声明方式是texture<类型,数字>，使用cudaBindTexture2D()函数绑定纹理内存，cudaUnbindTexture()函数解除绑定，使用tex2D()函数读取内存数据。
  + 固定内存：存储在主机内存中，又称为不可分页或页锁定内存。对于固定内存，操作系统不会对其分页，也不会交换到磁盘上，可以确保它始终驻留在物理内存上。在编写程序时可以直接访问这块物理地址，因为它不会被破坏或迁移。固定内存是为了提高访问速度而被设计出来的。GPU 如果知道主机中的物理地址，就可通过 DMA (Direct Memory Access, 直接内存访问) 方式来复制主机与 GPU 之间的数据。当然，用户编写程序时要注意不可一味使用固定内存，这样将导致物理内存迅速消耗完。在使用固定内存时，一般将调用cudaMemcpy()函数时使用的源内存或目的内存设置为固定内存，在调用完后不再需要时立即释放掉。分配固定内存需要使用cudaHostAlloc()函数；释放固定内存需要使用cudaFreeHost()函数。注意复制固定内存是异步的。



* 核函数 (kernel)

1. 在 GPU 中运行的函数都可称为核函数。
2. 核函数可以使用修饰符来修饰，通常使用\_\_global\_\_标识符。调用时必须声明核函数的执行参数，调用方法通常是<<<gridSize, blockSize, sharedMemSize, streamID>>>，其中，gridSize和blockSize分别代表线程格尺寸与线程块尺寸，它们是必填项。sharedMemSize为动态分配的共享内存尺寸，streamID代表执行流的编号，这两个不是必填项。
3. 核函数需要通过线程格 (grid) 来组织，线程格下包含若干线程块(block)，而线程块包含若干个线程 (thread)。
4. 编写程序时要注意的一点是，一定要在调用前提前分配好核函数需要使用的数组或变量的空间，否则在GPU计算时可能会发生错误，例如越界错误。

* 常用的CUDA 对 C 语言的扩展

1. 指定函数执行器件的扩展：定义了函数类型修饰符，可用来确定函数是在 CPU 上执行还是在 GPU 上执行，以及是从 CPU 上调用还是从 GPU 上调用。有如下几种函数类型修饰符：
   * \_\_device\_\_：由它修饰的函数从 GPU 上调用，且在 GPU 上执行，这样的函数能够由\_\_device\_\_或\_\_global\_\_修饰的函数调用。由它修饰的函数有使用限制，比如不允许使用函数指针。
   * \_\_global\_\_：由它修饰的函数从 CPU 上调用，且在 GPU 上执行。这就是前面提到过的核函数 (kernel)。它只能从主机调用，而且它只是表示数据并行的步骤，其指令流由多个线程执行。
   * \_\_host\_\_：由它修饰的函数从 CPU 上调用，且在 CPU 上执行，这也就是传统的C函数。
2. 变量存储位置的扩展：定义了变量类型修饰符。在传统的 CPU 程序中，变量存储位置一般由编译器负责指定。但在 CUDA 中，开发者通常需要用变量类型修饰符指定变量存储位置，比如，GPU 缓存、共享内存、寄存器、全局内存、主机内存等等。在之前的部分介绍了 CUDA 内存分类。下面列出的是常用的变量类型修饰符及其含义：
   * \_\_device\_\_：由该修饰符修饰的变量存放于设备的全局内存中，主机通过运行时库可以对其进行访问，所有线程都可访问此类变量。
   * \_\_shared\_\_：由该修饰符修饰的变量存放于共享内存中，只有它所在的线程块块里的线程可以对其进行访问。
   * \_\_constant\_\_：由该修饰符修饰的变量存放于常量内存中，主机通过运行时库可以对其进行访问，所有线程都可访问此类变量。
3. 执行配置扩展：定义了执行配置运算符<<< >>>，调用核函数时要使用<<< >>>指定执行配置。执行配置包含4个参数：1. **线程格尺寸**（必填）；2. **线程块尺寸**（必填）；3. 共享内存尺寸（选填，默认为0）；4. 执行流序号（选填，默认为0）。
4. 内建变量扩展：内建变量用来在运行时获取线程索引、线程块索引以及线程块尺寸、线程格尺寸等信息。CUDA共有5个内建变量：
   * **gridDim**：描述线程格的尺寸。这是一个结构体，它包含 x，y，z 三个元素，表示线程格在各个维度上的尺寸。
   * **blockDim**：描述线程块的尺寸。和 gridDim 一样也是由 x，y，z 三个元素组成的结构体，表示线程块在各个维度上的尺寸。
   * **blockIdx**：描述线程块在线程格中的坐标位置。和 gridDim 一样也是一个由 x，y，z 三个元素组成的结构体，分别表示当前线程块在线程格中 x，y，z 三个方向上的索引。
   * **threadIdx**：描述线程在线程块中的坐标位置。和 gridDim 一样也是一个由 x，y，z 三个元素组成的结构体，分别表示当前线程在线程块中 x，y，z 三个方向上的索引。
   * **warpSize**：描述线程束的尺寸。 在 CUDA 中，warpSize 为32。

* 常用的CUDA内存管理API

1. cudaMalloc()

* 原型：cudaError\_t cudaMalloc (void \*\* devPtr, size\_t size);
* 参数
  + devPtr：指向分配的设备全局内存
  + size：需要分配的内存尺寸，单位为字节byte
* 作用：用于在 GPU 的全局内存中分配内存。在设备上，分配 size 个字节的连续内存空间，并返回一个指向此内存空间的指针 \*devPtr。分配的内存可用于存储任何类型的变量。
* 返回值
  + 若分配成功，则返回 cudaSuccess
  + 若分配失败，则返回 cudaErrorMemoryAllocation

1. cudaMemcpy()

* 原型：cudaError\_t cudaMemcpy (void \*dst,

const void \*src,

size\_t size,

enum cudaMemcpyKind kind);

* 参数：
* dst：指向目标内存的指针
* src：指向源内存的指针
* size：需要复制的数据尺寸，单位为字节byte
* kind：数据复制的方向
* 作用：从 src 指针指向的内存空间，复制 size 个字节的数据，到 dst 指针指向的空间。其中，kind 用于指定数据移动的方向：
  + cudaMemcpyHostToHost：从主机端至主机端
  + cudaMemcpyHostToDevice：从主机端至设备端
  + cudaMemcpyDeviceToHost：从设备端至主机端
  + cudaMemcpyDeviceToDevice：从设备端至设备端

数据复制以异步方式执行，当函数返回时，并不代表数据复制已完成，而只是将复制操作放入了执行流。若后续操作包含隐式同步功能，则会等待复制完成后开始执行后续操作。

* 返回值：
  + 若操作成功，则返回 cudaSuccess
  + 若数值错误，则返回 cudaErrorInvalidValue
  + 若指针错误，则返回 cudaErrorInvalidDevicePointer
  + 若方向错误，则返回 cudaErrorInvalidMemcpyDirection

1. cudaFree()

* 原型：cudaError\_t cudaFree(void\* devPtr);
* 参数：
  + devPtr：指向要释放的内存空间。
* 作用：释放由 cudaMalloc() 分配的内存空间。
* 返回值
  + 若操作成功，则返回 cudaSuccess
  + 若指针错误，则返回 cudaErrorInvalidDevicePointer
  + 若初始化错误，则返回 cudaErrorInitializationError

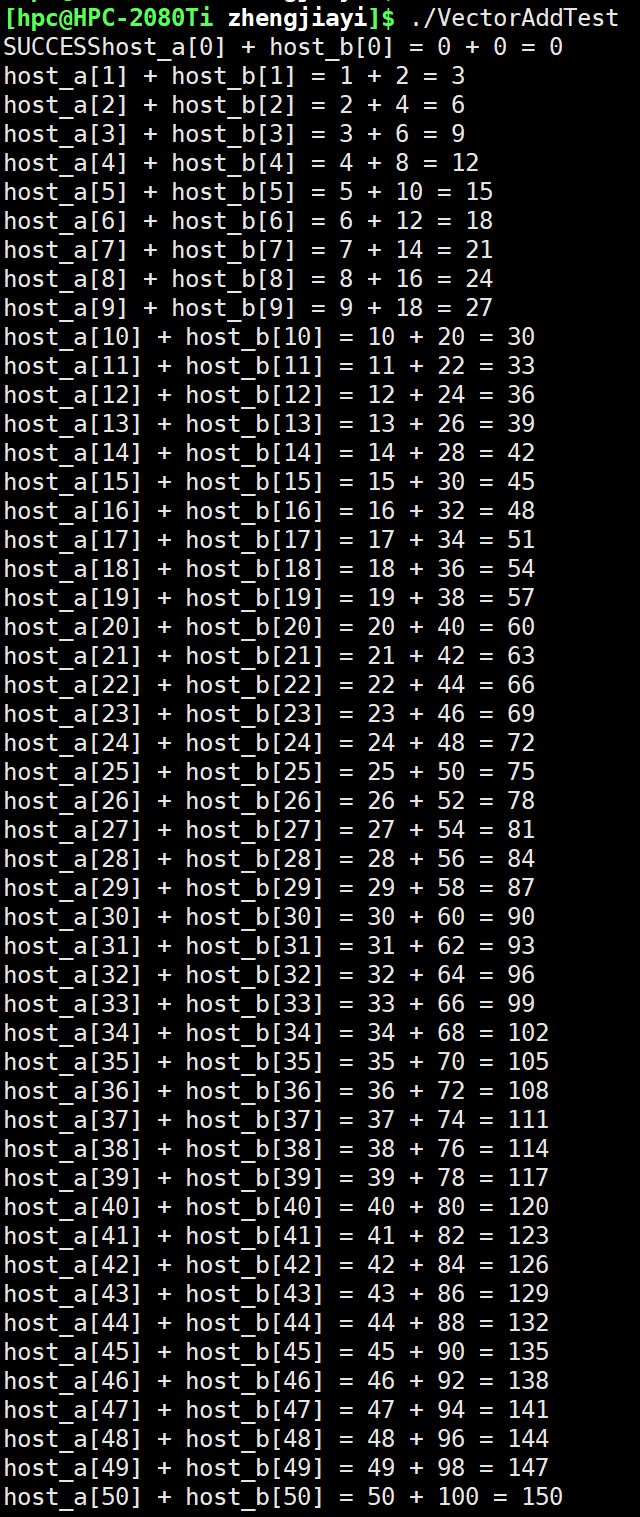
**实验要求**

1. 创建名为 VectorAdd.cu 的 CUDA 源文件。

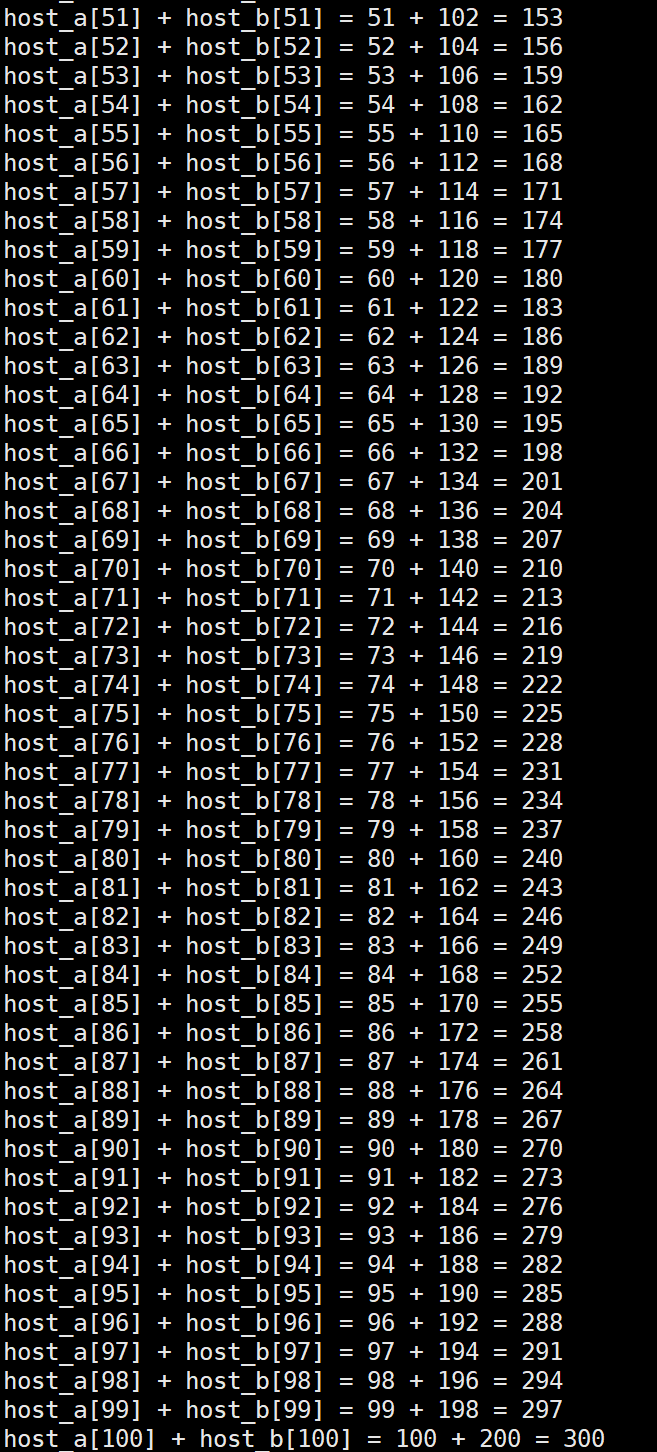
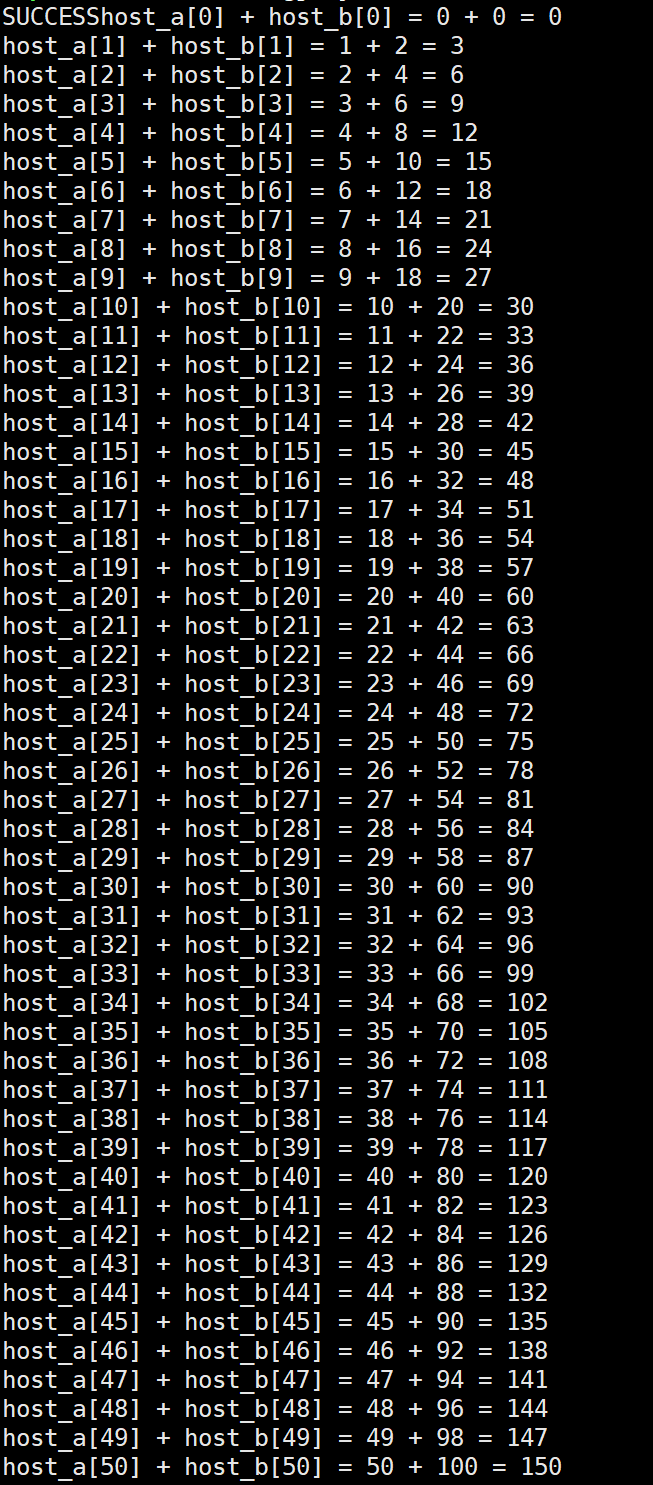
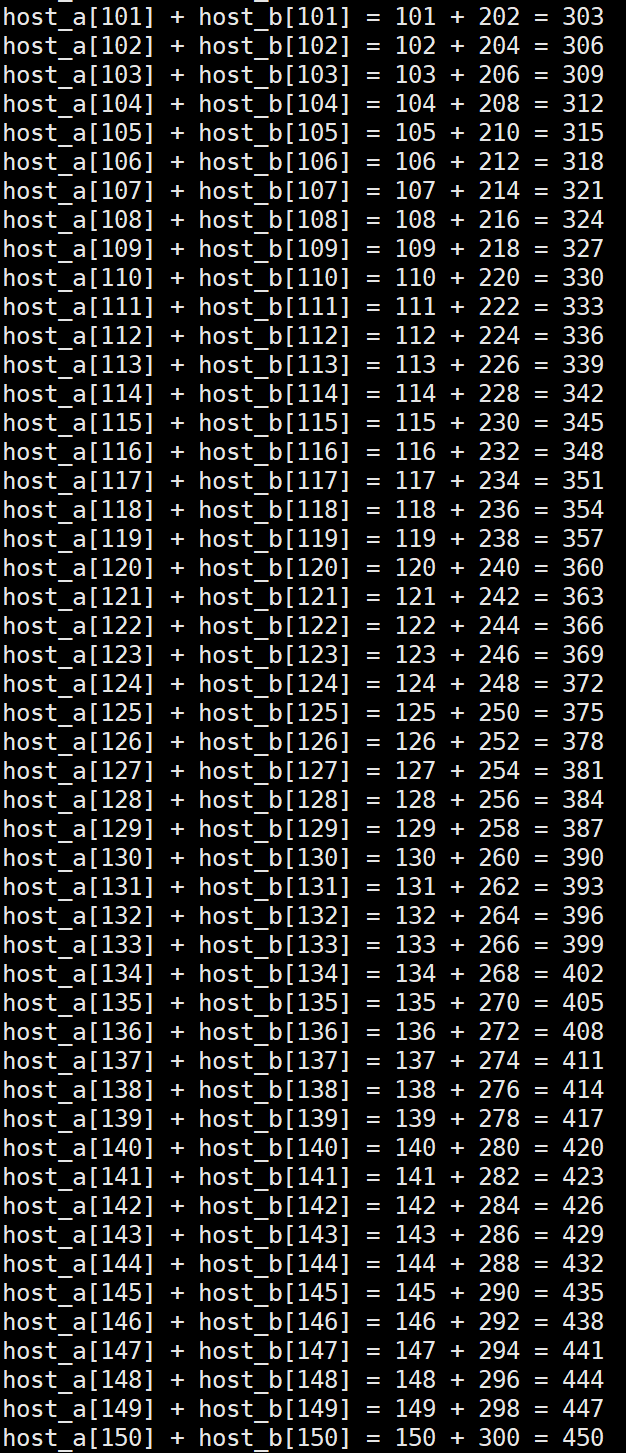
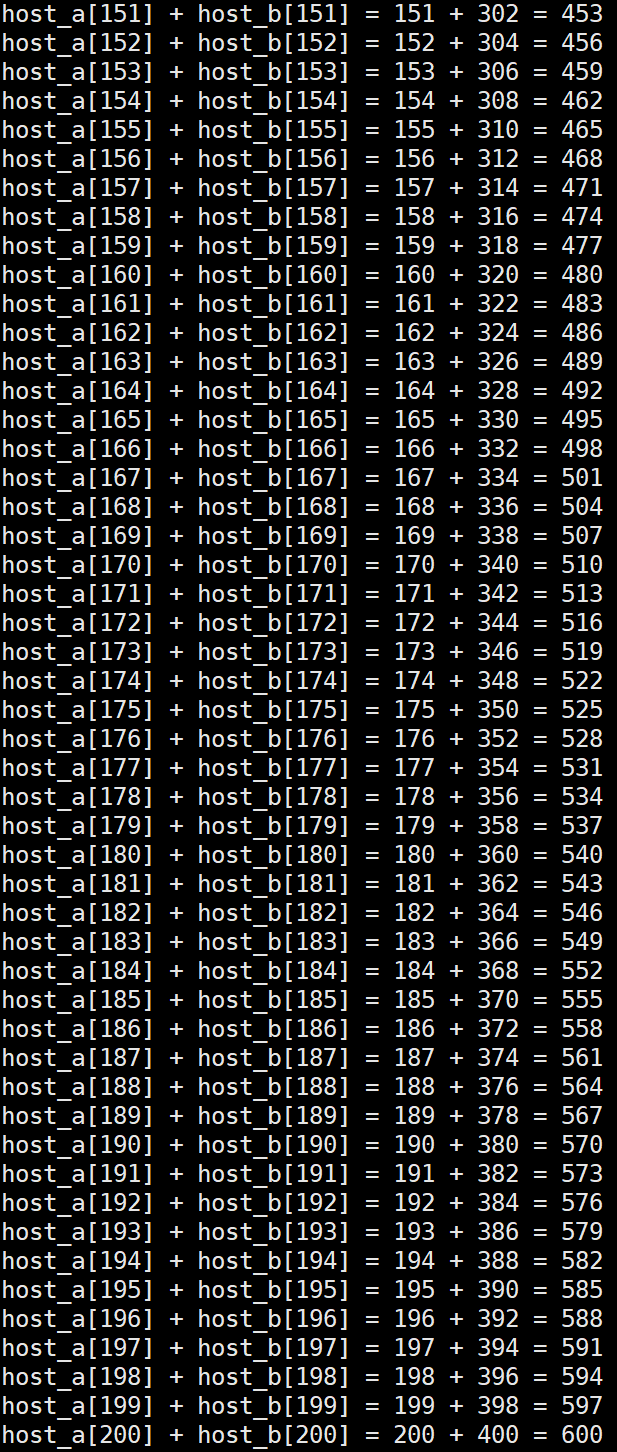
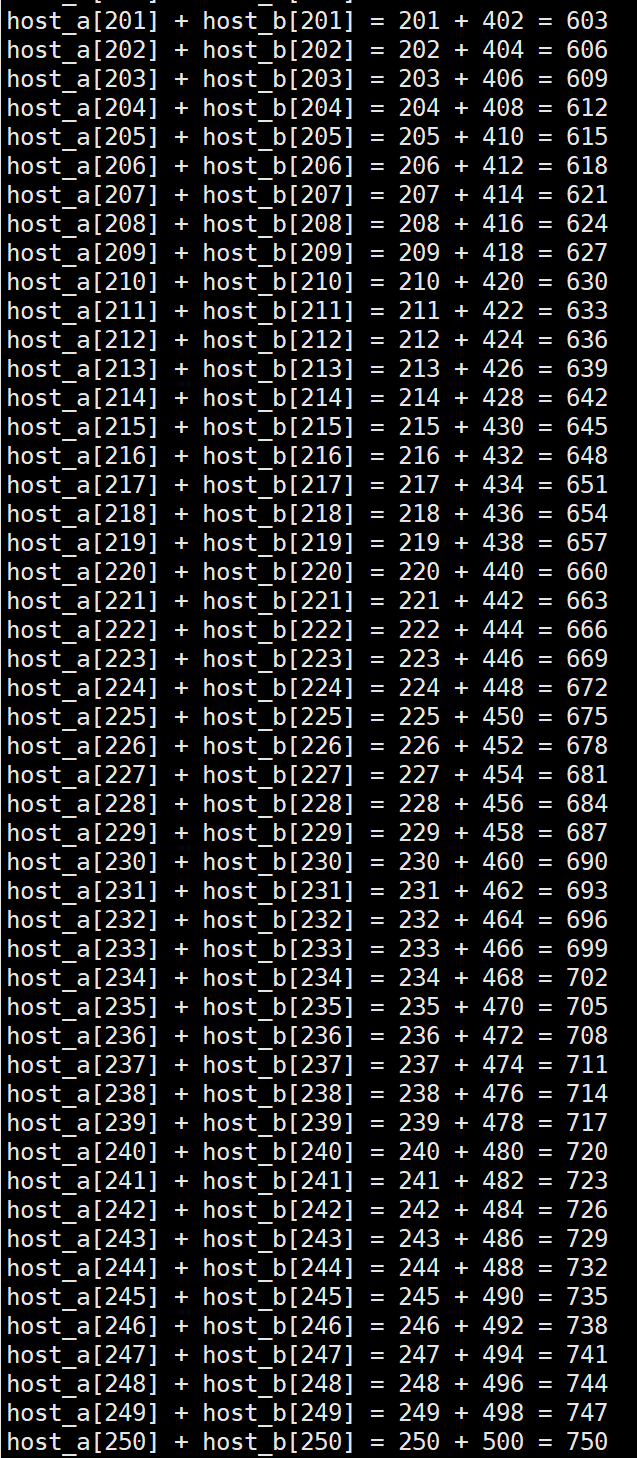
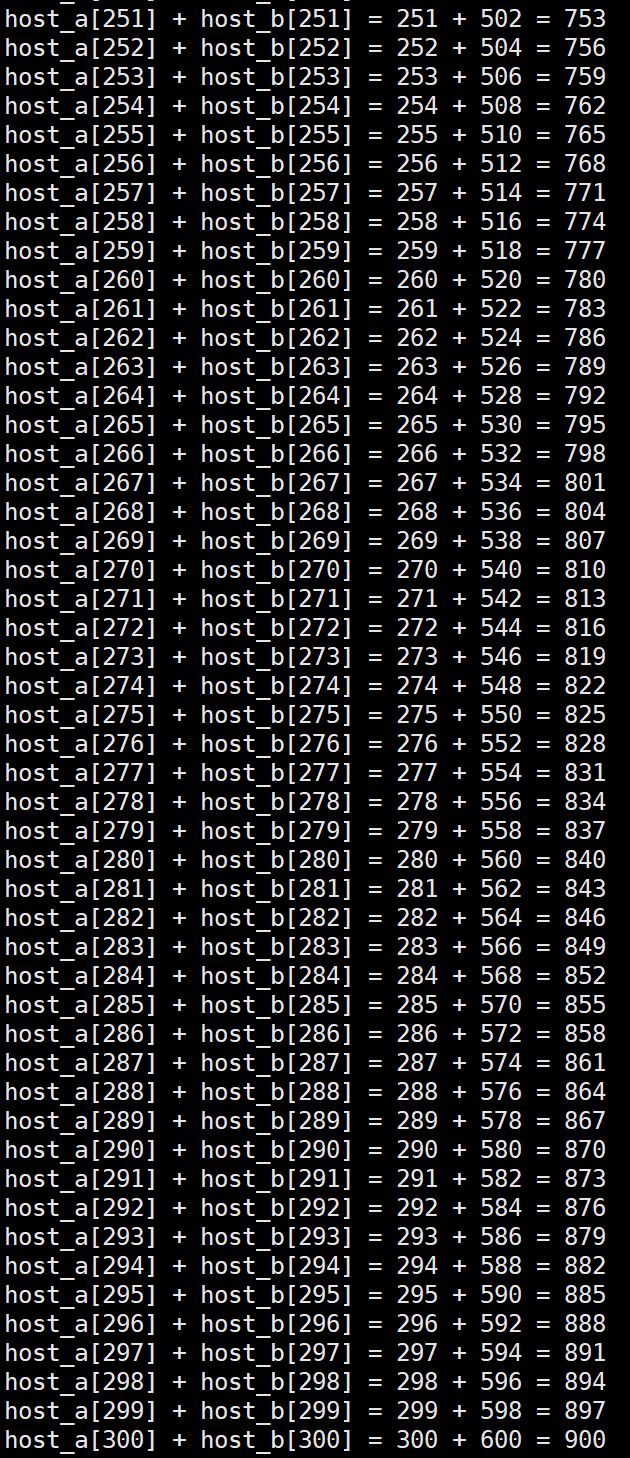
|  |
| --- |
| #include <stdio.h>  #include <cuda\_runtime.h>    //\_\_global\_\_声明的函数，告诉编译器这段代码交由CPU调用，由GPU执行  \_\_global\_\_ void add(const int \*dev\_a, const int \*dev\_b, int \*dev\_c) {  int i = threadIdx.x; // build-in variable  dev\_c[i] = dev\_a[i] + dev\_b[i];  }    int main(void) {  //申请主机内存，并进行初始化  int host\_a[512], host\_b[512], host\_c[512];  for(int i = 0; i < 512; i++) {  host\_a[i] = i;  host\_b[i] = i << 1;  }    //定义cudaError，默认为cudaSuccess(0)  cudaError\_t err = cudaSuccess;    //申请GPU存储空间  int \*dev\_a, \*dev\_b, \*dev\_c;  err = cudaMalloc((void \*\*)&dev\_a, sizeof(int) \* 512);  err = cudaMalloc((void \*\*)&dev\_b, sizeof(int) \* 512);  err = cudaMalloc((void \*\*)&dev\_c, sizeof(int) \* 512);  if(err != cudaSuccess) {  printf("the cudaMalloc on GPU is failed");  return 1;  }  printf("SUCCESS");  //将要计算的数据使用cudaMemcpy传送到GPU  cudaMemcpy(dev\_a, host\_a, sizeof(host\_a), cudaMemcpyHostToDevice);  cudaMemcpy(dev\_b, host\_b, sizeof(host\_b), cudaMemcpyHostToDevice);    //调用核函数在GPU上执行。数据较少，只使用一个Block，含有512个线程  add<<<1,512>>>(dev\_a, dev\_b, dev\_c);  cudaMemcpy(&host\_c, dev\_c, sizeof(host\_c), cudaMemcpyDeviceToHost);  for(int i = 0; i < 512; i++) {  printf("host\_a[%d] + host\_b[%d] = %d + %d = %d\n", i, i, host\_a[i], host\_b[i], host\_c[i]);  }  cudaFree(dev\_a);//释放GPU内存  cudaFree(dev\_b);//释放GPU内存  cudaFree(dev\_c);//释放GPU内存  return 0;  } |

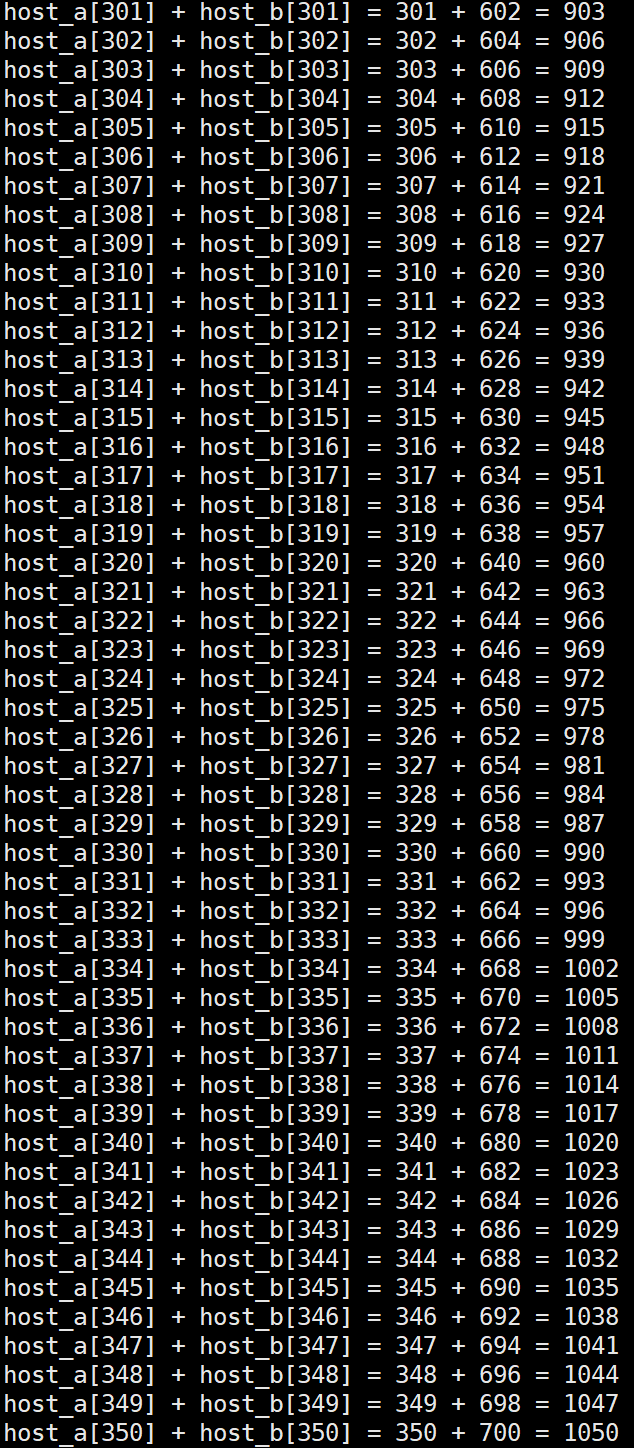
1. 编译并运行 VectorAdd.cu，记录实验结果。

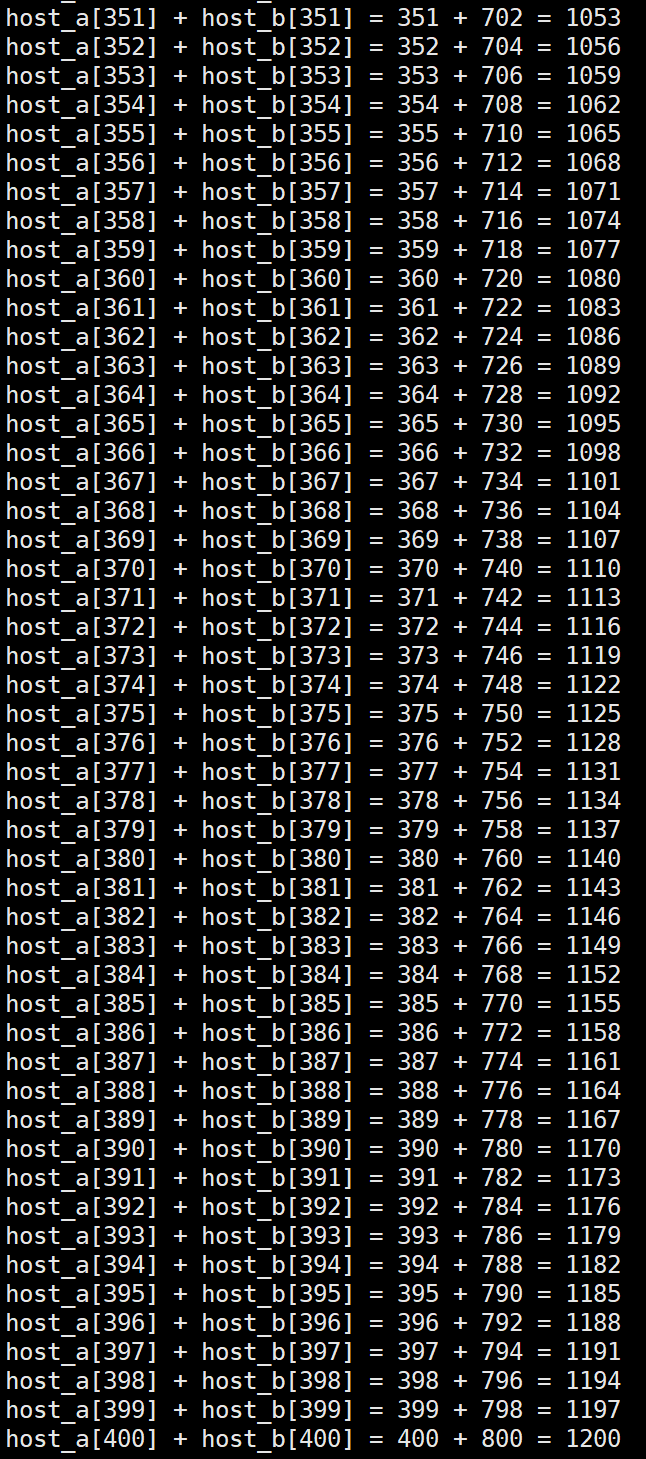
|  |
| --- |
| nvcc VectorAdd.cu -o VectorAddTest  ./VectorAddTest |

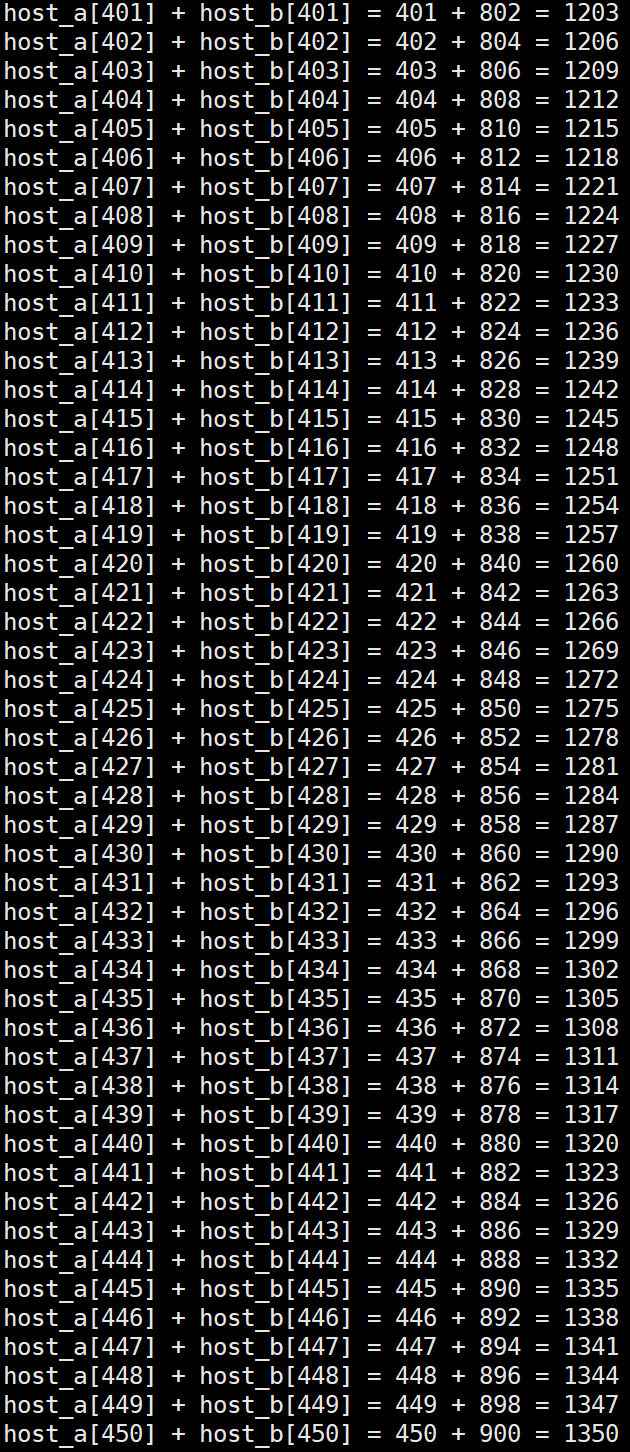


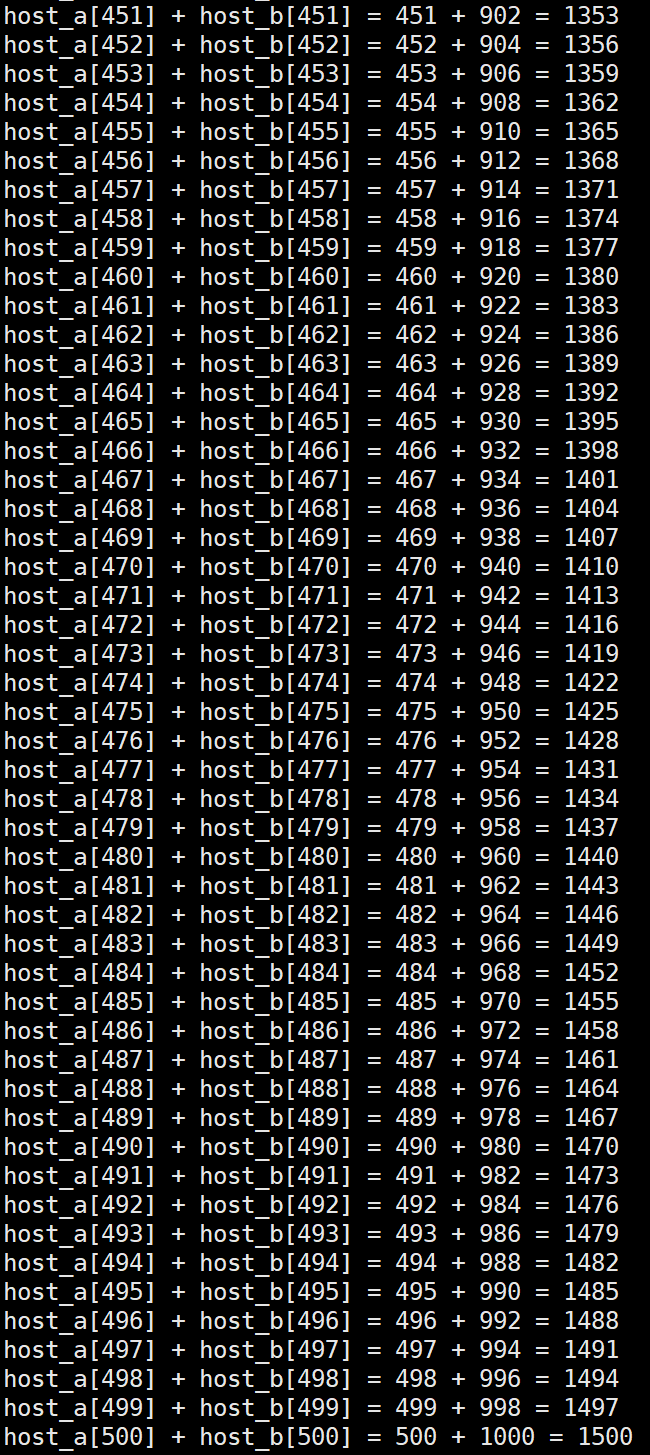
**第一张图我截了两次，下一张是没有zhengjiayi的！**

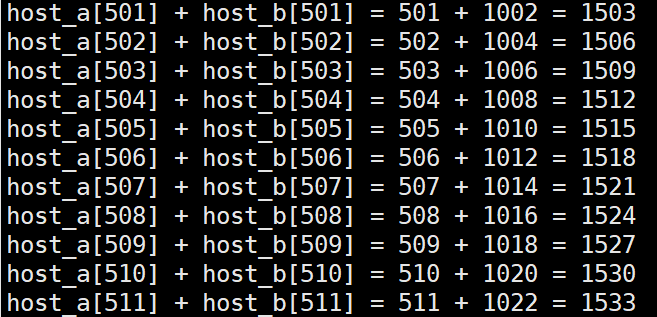
    











## 实验4.3 基于CUDA优化矩阵乘法

### 实验目的

1. 掌握基于 CUDA 的优化科学计算的方法。
2. 比较各种 CUDA 优化方式的性能提升效果。
3. 掌握利用加速比、运行时间、效率等测度分析并行程序性能。

### 实验内容

* 实验原理

设 为 的矩阵， 为 的矩阵，则 与 的乘积为 的矩阵 ，记作 ，其中矩阵 中的第 行第 列元素可表示为：

本次实验中，假设 与 均为尺寸为 的正方形矩阵。另外，不必更深入地优化代码。

* 程序逻辑



* + matgen：输入为矩阵首地址、矩阵行数、矩阵列数，功能是使用介于0与1之间的浮点随机数填充矩阵。
  + matmult：输入为两个矩阵，功能是将它们相乘。该函数是运行于主机端的C语言程序。该函数既是确保计算正确性的基准程序，将其计算结果与并行矩阵乘法程序的计算结果对比，判断并行程序计算结果正确与否，也是衡量并行程序性能的基准程序。为提高计算精度，使用双浮点数据类型存储计算中间结果。
  + matmultCUDA：输入为两个矩阵，功能是使用 GPU 计算乘积。首先，在设备全局内存中，分配用于存放输入矩阵和输出矩阵的内存空间。之后，将输入矩阵从主机端内存复制到设备全局内存。在这里，使用**cudaMemcpy2D()**函数复制二维数组，这样只需一次函数调用即可完成复制。然后，调用核函数matMultCUDA，它通过blockIdx.x和threadIdx.x计算每个线程需要使用的数据坐标，并将计算结果写入输出矩阵。最后，将输出矩阵从设备全局内存复制到主机端内存。
  + compare\_mat：输入为两个矩阵，功能是计算它们的平均相对误差以及最大相对误差，并打印比较结果。
* 程序说明

1. 共享内存的使用

matMultCUDA<<<gridSize, blockSize, sharedMemSize>>>(parameters…);

在核函数的执行配置中，指定了线程块的共享内存尺寸。共享内存由线程块中的线程共享，每个线程块负责计算一行输出数据，所以在线程块的共享内存中，存储了输出数据需要使用的那一行输入数据。以下代码将一行数据从全局内存复制到共享内存中：

extern \_\_shared\_\_ float data[];

const int tid = threadIdx.x;

const int row = blockIdx.x;

int i, j;

for(i = tid; i < n; i += blockDim.x) {

data[i] = a[row \* lda + i];

}

1. 分配全局内存的方式

为提升全局内存分配效率，在分配内存时，可以使用CUDA 提供的 cudaMallocPitch() 函数，查找最佳地址对齐数。

优化后的内存分配代码如下：

size\_t pitch\_a, pitch\_b, pitch\_c;

cudaMallocPitch((void\*\*) &ac, &pitch\_a, sizeof(float) \* n, n);

cudaMallocPitch((void\*\*) &bc, &pitch\_b, sizeof(float) \* n, n);

cudaMallocPitch((void\*\*) &cc, &pitch\_c, sizeof(float) \* n, n);

cudaMallocPitch()函数会选择合适的对齐数。

将输入矩阵复制到全局内存时要使用对齐数，代码如下：

cudaMemcpy2D(ac, sizeof(float) \* n, a, sizeof(float) \* lda, sizeof(float) \* n, n, cudaMemcpyHostToDevice);

cudaMemcpy2D(bc, sizeof(float) \* n, b, sizeof(float) \* ldb, sizeof(float) \* n, n, cudaMemcpyHostToDevice);

调用核函数的代码如下：

matMultCUDA<<<n, NUM\_THREADS, sizeof(float) \* n>>>(ac, pitch\_a / sizeof(float), bc, pitch\_b / sizeof(float), cc, pitch\_c / sizeof(float), n);

将输出矩阵从全局内存复制回主机端内存时，要使用对齐数，代码如下：

cudaMemcpy2D(c, sizeof(float) \* ldc, cc, pitch\_c, sizeof(float) \* n, n, cudaMemcpyDeviceToHost);

**实验要求**

1. 编译并运行 MatrixMul.cu

nvcc MatrixMul.cu -o MatrixMulTest

./MatrixMulTest

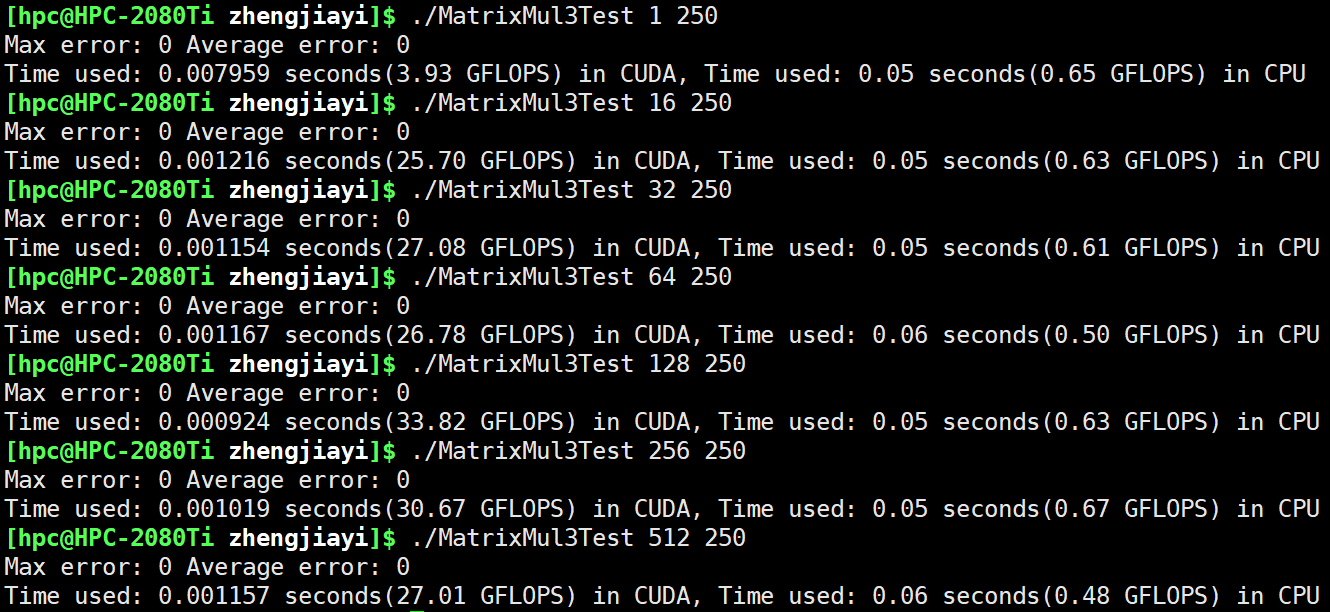
1. 运行程序，进行如下实验并记录数据（实验报告中给出数据并绘图）

* 实验一：设置矩阵尺寸为，调整线程块尺寸，记录程序运算时间及运算速度。

基准运行时间(s)：0.007959

基准运算速度(GFLOPS)：3.93

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算时间(s) | 0.001216 | 0.001154 | 0.001167 | 0.000924 | 0.001019 | 0.001157 |
| 运算速度(GFLOPS) | 25.70 | 27.08 | 26.78 | 33.82 | 30.67 | 27.01 |

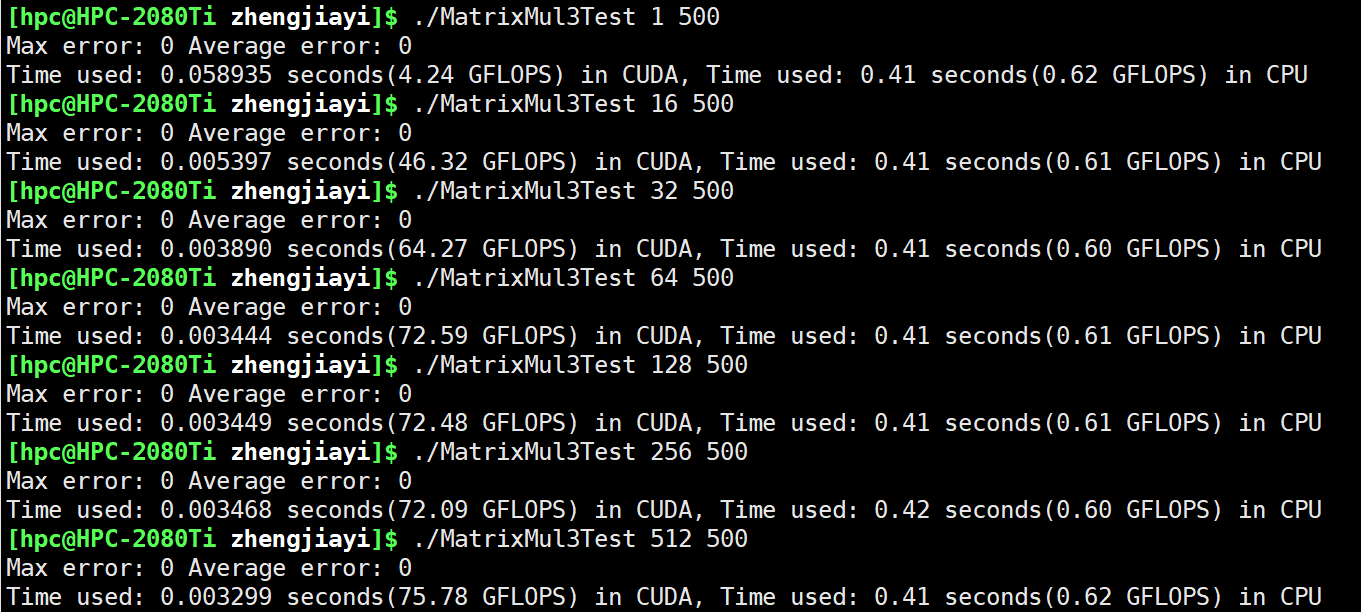


* 实验二：设置矩阵尺寸为，调整线程块尺寸，记录程序运算时间及运算速度。

基准运行时间(s)： 0.058935

基准运算速度(GFLOPS)：4.24

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算时间(s) | 0.005397 | 0.003890 | 0.003444 | 0.003449 | 0.003468 | 0.003299 |
| 运算速度(GFLOPS) | 46.32 | 64.27 | 72.59 | 72.48 | 72.09 | 75.78 |

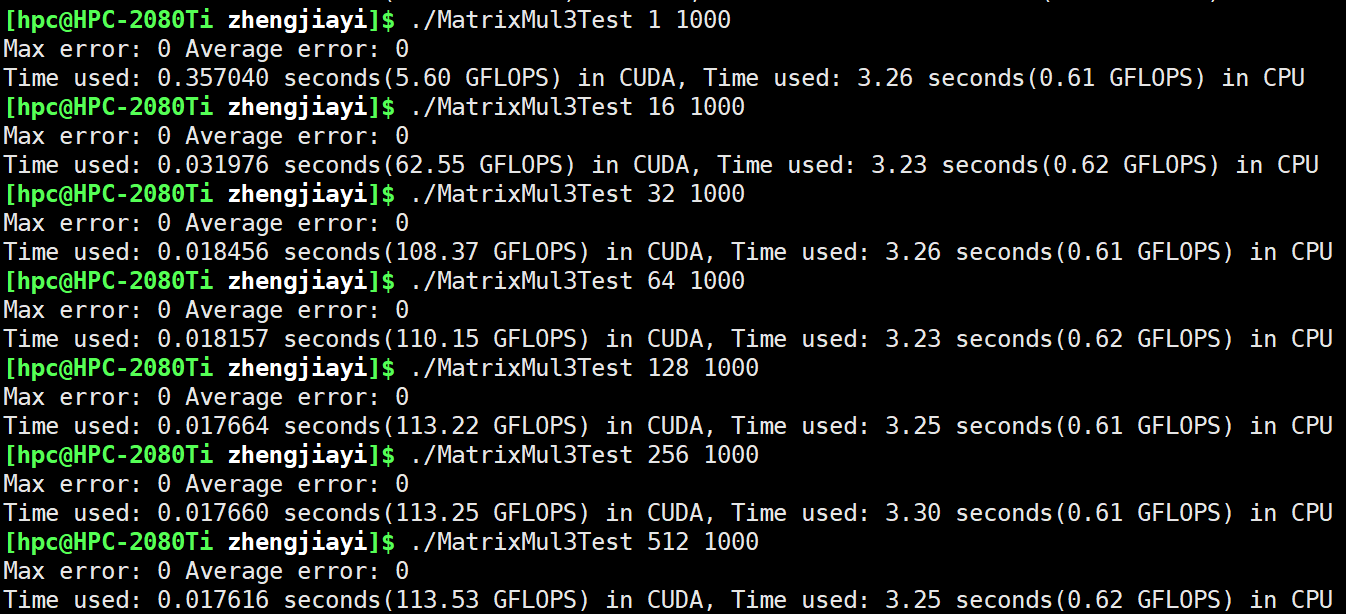


* 实验三：设置矩阵尺寸为，调整线程块尺寸，记录程序运算时间及运算速度

基准运行时间(s)：0.357040

基准运算速度(GFLOPS)：5.60

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算时间(s) | 0.031976 | 0.018456 | 0.018157 | 0.017664 | 0.017660 | 0.017616 |
| 运算速度(GFLOPS) | 62.55 | 108.37 | 110.15 | 113.22 | 113.25 | 113.53 |



（3）根据记录的数据，计算运算速度提升比（GFLOPS的比值）并绘图

* 实验一：设置矩阵尺寸为

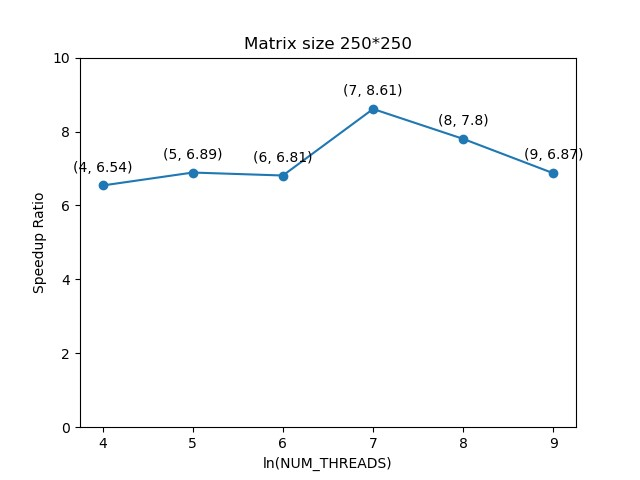
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算速度提升比 | 6.54 | 6.89 | 6.81 | 8.61 | 7.80 | 6.87 |

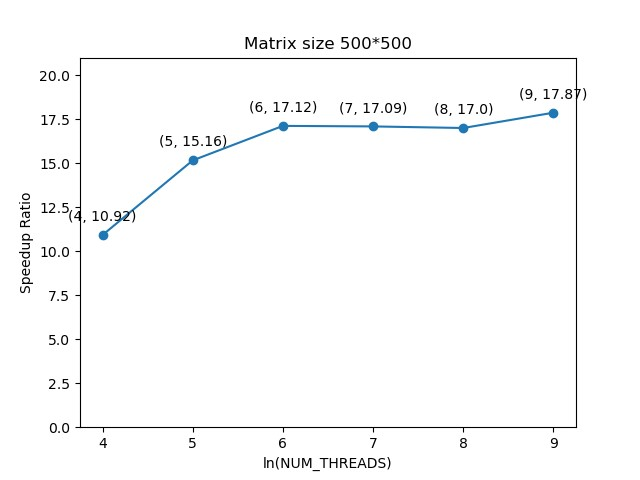
* 实验二：设置矩阵尺寸为

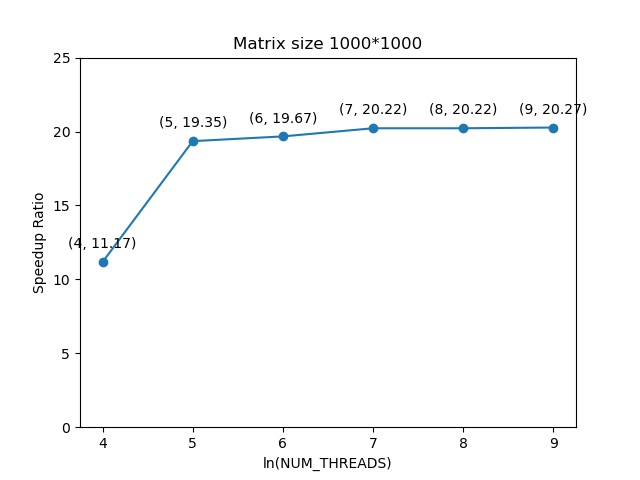
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算速度提升比 | 10.92 | 15.16 | 17.12 | 17.09 | 17.00 | 17.87 |

* 实验三：设置矩阵尺寸为

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NUM\_THREADS | 16 | 32 | 64 | 128 | 256 | 512 |
| 运算速度提升比 | 11.17 | 19.35 | 19.67 | 20.22 | 20.22 | 20.27 |







### 实验评分标准

一、课堂表现（10分）

二、实验结果（50分）

三、实验报告（40分）

### 思考题

思考题1：查阅相关资料，对实验所得的加速比图进行分析。