GPU 编程实验指导书

李世阳

2022 年 5 月 22 日

目录

1 GPU 编程

1.1 NVIDIA 云端课程

英伟达官方课程地址：https://courses.nvidia.com/courses/course-v1:DLI+C-AC-01+V1/。课程优惠码：DLITEACH0522_11_QBKE_37。具体操作流程见实验指导书《实验环境搭建》的“GPU 实验环境”一节。

该课程不仅仅介绍了 CUDA 编程的基本流程与概念，其实还涉及到了一些 CUDA 的高级用法，比如数据预取和 CUDA 流的应用，学有余力的同学可以自行探究并将其应用到自己的大作业中（如果你考虑在大作业中使用 CUDA 的话）。

1.2 CUDA 编程的基本模式

关于 GPU 上的硬件架构与并行计算的基本模式，理论课上应该已经讲过，指导书不再赘述，实验课上会带着大家回顾一下。这里以一段简单的代码展示 CUDA 编程的基本模式。

首先，和在 CPU 端编程一样，数据结构需要分配内存空间，这里又有两种方式。第一种，在 CPU 端给数据分配好内存空间并初始化以后，还需要对应的在 GPU 端分配显存空间，然后显示的调用数据传输的接口将 CPU 端的数据传输至 GPU 端。第二种，通过统一虚拟内存（UVM）接口或者零拷贝内存（zero-copy）接口，将数据分配在 CPU 端，此时 GPU 端的线程将可以通过 PCIE 总线直接访问这块内存区域，但是 UVM 与 zero-copy 其实也存在着差别，有兴趣的同学可以自己了解一下。

数据传输至 GPU 端后，GPU 的线程将可以直接访问这些数据，此时程序进入核函数。但是在执行核函数前，我们需要规定执行这个核函数所使用的计算资源，也就是使用的线程数量。注意，这里的“线程”与之前 CPU 多线程编程中的“线程”不尽相同。主要区别是 GPU 上运行的线程具有多级结构，与 GPU 计算单元的多级结构对应。在调用核函数时要指明多级线程结构中每一级的规模，这些线程就被调度到 GPU 的计算核心上运行。新的 GPU 架构也支持动态创建线程，有兴趣的同学可自行探索。分配的所有线程会一起执行核函数内的代码，我们可以通过线程的索引进行任务划分，给每个线程分配不同的计算任务，达到并行计算的效果。

最后，核函数执行完毕后，我们还需要把计算好的数据再从 GPU 端传输回来，以便我们检验正确性或存入磁盘。下面的代码是一个示例，代码中还包括了如何进行错误检查和计时函数的使用。

```
1  #include<iostream>
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include <iomanip>
5  #include"cuda_runtime.h"
6  #include"device_launch_parameters.h"
7
8  using namespace std;
9  const int N = 1024;
10 const int BLOCK_SIZE=1024;
11 float elm[N][N] = { 0 };
12
13 __global__ void test_kernel(float* m){
```

```
    int tx = threadIdx.x;//线程索引
    ...//执行计算任务
}

int main(){
float* temp=new float[N*N];
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        temp[i*N+j]=elm[i][j];
    }
}
cudaError_t ret;//用于错误检查，当 CUDA 接口调用成功会返回 cudaSucess
float* gpudata;
float* result=new float[N*N];
int size=N*N*sizeof(float);

ret=cudaMalloc(&gpudata,size);//分配显存空间
if(ret!=cudaSuccess){
   printf("cudaMalloc gpudata failed!\n");
}

ret=cudaMemcpy(gpudata,temp,size,cudaMemcpyHostToDevice);//将数据传输至 GPU 端

if(ret!=cudaSuccess){
   printf("cudaMemcpyHostToDevice failed!\n");
}

dim3 dimBlock(BLOCK_SIZE,1);//线程块
dim3 dimGrid(1, 1);//线程网格

cudaEvent_t start, stop;//计时器
float elapsedTime = 0.0;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);//开始计时

test_kernel << <dimGrid, dimBlock >> >(gpudata);//核函数

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);//停止计时
cudaEventElapsedTime(&elapsedTime, start, stop);

```

```
56 printf("GPU_LU:%f ms\n", elapsedTime);
57
58 cudaError_t cudaStatus2 = cudaGetLastError();
59 if (cudaStatus2 != cudaSuccess) {
60     fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStatus2));
61 }
62
63 ret=cudaMemcpy(result, gpudata, size, cudaMemcpyDeviceToHost);//将数据传回 CPU 端
64 if(ret!=cudaSuccess){
65     printf("cudaMemcpyDeviceToHost failed!\n");
66 }
67
68 cudaFree(gpudata);//释放显存空间，用 CUDA 接口分配的空间必须用 cudaFree 释放
69 //销毁计时器
70 cudaEventDestroy(start);
71 cudaEventDestroy(stop);
72 }
```

1.3 基于 CUDA 的高斯消元

关于普通高斯消元算法在 GPU 端的并行化有不止一种任务划分方式，这里介绍一种基本的思路，感兴趣的同学可以进行自由探索其他任务划分方式，网上也能搜到很多资料或论文。

在普通高斯消元算法中，第一层循环内嵌套了两个循环，这两个循环分别负责除法和消去的任务，由于这两个循环存在前后依赖，因此不能同时并行。我们可以使用 GPU 端的线程对这两个循环分别进行展开，因此需要两个核函数。

```
1  cudaError_t ret;
2  for(int k=0;k<width;k++){
3      division_kernel<<<grid,block>>>(data_D, ...);//负责除法任务的核函数
4      cudaDeviceSynchronize();//CPU 与 GPU 之间的同步函数
5      ret = cudaGetLastError();
6      if(ret!=cudaSuccess){
7          printf("division_kernel failed, %s\n",cudaGetErrorString(ret));
8      }
9
10     eliminate_kernel<<<grid,block>>>(data_D, ...);//负责消去任务的核函数
11     cudaDeviceSynchronize();
12     ret = cudaGetLastError();
13     if(ret!=cudaSuccess){
14         printf("eliminate_kernel failed, %s\n",cudaGetErrorString(ret));
15     }
16 }
```

在核函数内部，如何进行具体的任务划分也有一定的讲究，需要同学们对 GPU 的硬件架构有基本的了解。对于第一个核函数的设计是比较简单的，我们只要让每个线程负责第 K 次循环中单独一列的计算即可。但是 GPU 上可用的线程数量也是有限的，一个块内一般最多只有 1024 个线程，而具体能使用多少线程块取决于 GPU 型号，这一数值可以通过 CUDA 提供的接口在代码中手动查询，也可以去 NVIDIA 官网查看产品手册获知。此核函数的一个示例如下所示。

```
__global__ void division_kernel(float* data, int k, int N){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;//计算线程索引
    int element = data[k*N+k];
    int temp = data[k*N+tid];
    //请同学们思考，如果分配的总线程数小于 N 应该怎么办？
    data[k*N+tid] = (float)temp/element;

    return;
}
```

对于消去的过程，在第 K 次循环中，我们需要对第 k+1 行至最后一行进行消去，但是要注意到在每一行计算完成后需要将这一行的第 k 列设为 0，以保证我们的矩阵最后变为一个上三角矩阵，因此这里存在一个同步问题。

在 CUDA 中，cudaDeviceSynchronize() 接口用于同步 GPU 上的所有线程，除此以外，我们也可以在核函数内部调用 ___syncthreads() 接口来同步块内线程，但是没有接口用于块间同步，这是 GPU 的硬件架构导致的。不过最新的安培架构理论上已经具备了实现块间同步的可能，但是 CUDA 目前还没有提供一个这样的统一接口，理论上可以通过信号量等方式在安培架构上手动实现块间同步，不过同学们手里大概率也没有安培架构的 GPU。

因此为了保证正确同步的同时最大化利用 GPU 的并行性，可以让一个线程块负责固定一行的计算任务，块内的线程分别负责这一行的不同位置上的元素的运算任务，最后进行块内同步。一个示例如下所示。

```
__global__ void eliminate_kernel(float* data, int k, int N){
    int tx = blockDim.x * blockIdx.x + threadIdx.x;
    if(tx==0)
    data[k*N+k]=1.0;//对角线元素设为 1

    int row = k+1+blockIdx.x;//每个块负责一行

    while(row<N){
        int tid = threadIdx.x;
        while(k+1+tid < N){
            int col = k+1+tid;
            T temp_1 = data[(row*N) + col];
            T temp_2 = data[(row*N)+k];
            T temp_3 = data[k*N+col];
```

```
15              data[(row*N) + col] = temp_1 - temp_2*temp_3;
16              tid = tid + blockDim.x;
17          }
18          __syncthreads();//块内同步
19          if (threadIdx.x == 0){
20              data[row * N + k] = 0;
21          }
22          row += gridDim.x;
23      }
24      return;
25  }
```

至此高斯消元的计算已经完成，只需要将数据传回 CPU 端即可。助教本人使用 NVIDIA P100 显卡进行测试时，在矩阵规模分别为 1024 和 2048 时得到的加速效果如表1所示。

表 1: CUDA 与平凡算法性能对比

N\Algo	simple	CUDA
1024	2157.57ms	22.97ms
2048	17324.63ms	103.68ms

可以看到加速的效果能达到超过百倍，如果数据规模进一步上升加速比还能扩大，GPU 的强大算力体现的淋漓尽致。

不过同学们在对比 GPU 与 CPU 的计算结果时可能发现最后算的值并不完全相同，图1.1是我用 10x10 矩阵测试时的结果。这并不是我们的算法出现了逻辑错误导致的，而是由于 GPU 线程的浮点计算精度与 CPU 不同导致的，一般来说 GPU 端的浮点计算精度比 CPU 小一些。

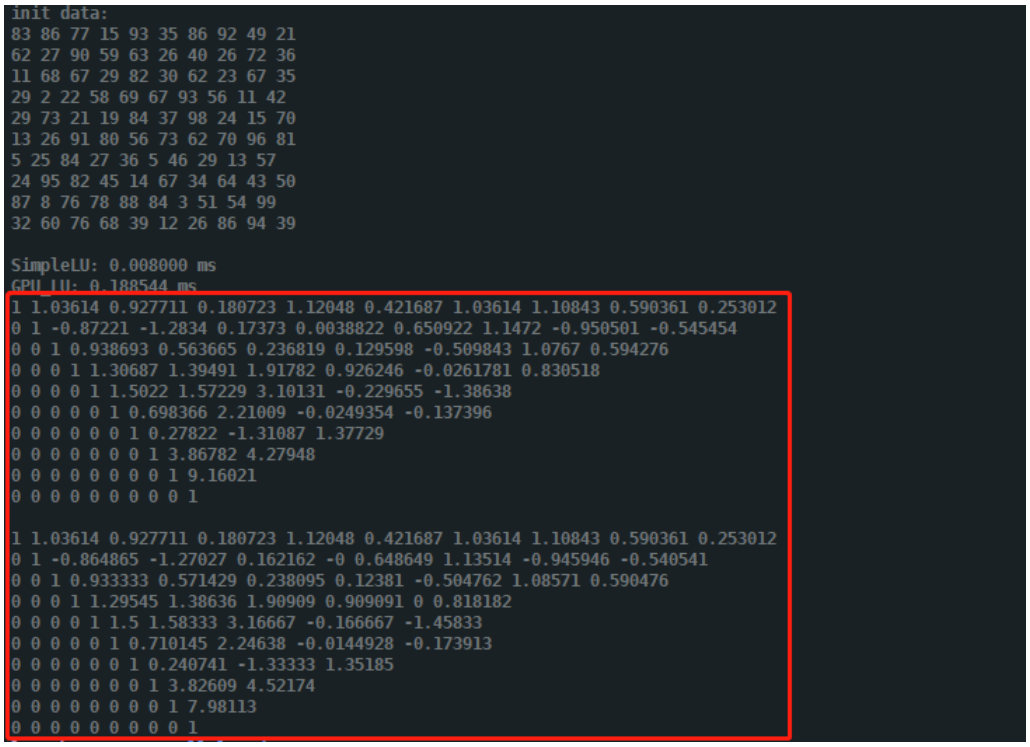

图 1.1: 红色方框中第一个矩阵是 CPU 端平凡算法的结果，第二个矩阵是 GPU 得到的结果

1.4 关于 CUDA 的 debug 与 profile 工具

CUDA 编程的 debug 是一个较为痛苦的过程，虽然也有一些特定的 debug 工具比如 cuda-gdb，但是根据助教本人的经验来看，这些工具也并不总是十分靠谱，很多时候还是需要自己意念调试。Visual Studio 和 VScode 也都对 CUDA 提供了相应支持。另外，据说 Clion 为 CUDA 调试提供了较好的支持，Clion 官网也提供了学生版下载通道，可以免费使用 1 年。

关于性能分析，VTune 应该也支持部分关于 GPU 的性能分析，自己的笔记本有显卡的同学可以试试。在命令行工具中，nvprof 是常用的 CUDA 性能分析工具。在安装 CUDA 时，官方提供的 toolkit 里有一款用于 debug 和性能分析的工具 nsight system，既可以在 linux 命令行中使用，也可以在 windows 上提供有图形界面的性能分析和 debug，有条件的同学也可以尝试。

在 NVIDIA 的云端课程中，给大家提供了一个虚拟的云端环境，可以编译运行 CUDA 代码，并且能用 nsight sysytem 进行性能分析，这样得到的性能虽然是不完全真实的，但大家也可以尝试一下。

1.5 没有显卡又想用 CUDA 该怎么办

对于电脑上没有 NVIDIA 显卡的同学来说，如果想要用 CUDA 做点东西，目前有两种方式：

1. 在 NVIDIA 官方课程提供的虚拟环境中运行 CUDA 代码，该环境中还配套了虚拟桌面，可以在可视化界面中使用 nsys。但是此种方式得到的性能是不真实的。

2. 使用 GPGPU-sim（仅支持 Linux 系统），环境搭建教程可参考我的博客。这是一个类似于 qemu 的针对 GPU 的硬件模拟器，但在此情况下得到的性能仍然是不真实的且精度较差。但是如果你对 GPU 的内部硬件架构感兴趣，GPGPU-sim 是一个值得考虑的学习工具，它甚至支持你在模拟器里自己修改 GPU 的硬件架构以提升自己的程序性能，感兴趣的同学可自行探索。