

GPU 编程实验指导书

李世阳、孙辉

2024 年 6 月

华志远 孔德嵘 张逸非

2025 年 6 月

目录

1 GPU 编程	3
1.1 CUDA 编程的基本模式	3
1.2 基于 CUDA 的高斯消元	5
1.3 关于 CUDA 的 debug 与 profile 工具	7
1.4 一些 GPU 编程学习资源	8
1.5 没有显卡又想用 CUDA 该怎么办	8
2 口令猜测选题	8
3 ANN 选题	9
4 NTT 选题	9

1 GPU 编程

1.1 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，然后显式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。

Notes: 零拷贝不是无需拷贝，而是一种隐式异步即时拷贝策略，每次隐式拷贝还是要走 PCIe 总线，所以频繁的对零拷贝内存进行读写，性能也会显著降低。

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){
14     int tx = threadIdx.x;//线程索引
15     ...//执行计算任务
16 }
17
18 int main(){
19     float* temp=new float[N*N];
20     for(int i=0;i<N;i++){
```

```
21     for(int j=0;j<N;j++){
22         temp[i*N+j]=elm[i][j];
23     }
24 }
25 cudaError_t ret;//用于错误检查, 当 CUDA 接口调用成功会返回 cudaSuccess
26 float* gpudata;
27 float* result=new float[N*N];
28 int size=N*N*sizeof(float);
29
30 ret=cudaMalloc(&gpudata,size);//分配显存空间
31 if(ret!=cudaSuccess){
32     printf("cudaMalloc gpudata failed!\n");
33 }
34
35 ret=cudaMemcpy(gpudata,temp,size,cudaMemcpyHostToDevice);//将数据传输至 GPU 端
36
37 if(ret!=cudaSuccess){
38     printf("cudaMemcpyHostToDevice failed!\n");
39 }
40
41 dim3 dimBlock(BLOCK_SIZE,1);//线程块
42 dim3 dimGrid(1, 1);//线程网格
43
44 cudaEvent_t start, stop;//计时器
45 float elapsedTime = 0.0;
46 cudaEventCreate(&start);
47 cudaEventCreate(&stop);
48 cudaEventRecord(start, 0);//开始计时
49
50 test_kernel << <dimGrid, dimBlock >> >(gpudata);//核函数
51
52 cudaEventRecord(stop, 0);
53 cudaEventSynchronize(stop);//停止计时
54 cudaEventElapsedTime(&elapsedTime, start, stop);
55
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 }

```

1.2 基于 CUDA 的高斯消元

关于普通高斯消元算法在 GPU 端的并行化有不只一种任务划分方式, 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。

```

1  __global__ void division_kernel(float* data, int k, int N){

```

```

2     int tid = blockDim.x * blockIdx.x + threadIdx.x; //计算线程索引
3     int element = data[k*N+k];
4     int temp = data[k*N+tid];
5     //请同学们思考，如果分配的总线程数小于 N 应该怎么办？
6     data[k*N+tid] = (float)temp/element;
7
8     return;
9 }

```

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。

```

1 __global__ void eliminate_kernel(float* data, int k, int N){
2     int tx = blockDim.x * blockIdx.x + threadIdx.x;
3     if(tx==0)
4         data[k*N+k]=1.0; //对角线元素设为 1
5
6     int row = k+1+blockIdx.x; //每个块负责一行
7
8     while(row<N){
9         int tid = threadIdx.x;
10        while(k+1+tid < N){
11            int col = k+1+tid;
12            T temp_1 = data[(row*N) + col];
13            T temp_2 = data[(row*N)+k];
14            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。

```

init data:
83 86 77 15 93 35 86 92 49 21
62 27 90 59 63 26 40 26 72 36
11 68 67 29 82 30 62 23 67 35
29 2 22 58 69 67 93 56 11 42
29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81
5 25 84 27 36 5 46 29 13 57
24 95 82 45 14 67 34 64 43 50
87 8 76 78 88 84 3 51 54 99
32 60 76 68 39 12 26 86 94 39

SimpleU: 0.008000 ms
GPU U: 0.188544 ms

1 1.03614 0.927711 0.180723 1.12048 0.421687 1.03614 1.10843 0.590361 0.253012
0 1 -0.87221 -1.2834 0.17373 0.0038822 0.650922 1.1472 -0.950501 -0.545454
0 0 1 0.938693 0.563665 0.236819 0.129598 -0.509843 1.0767 0.594276
0 0 0 1 1.30687 1.39491 1.91782 0.926246 -0.0261781 0.830518
0 0 0 0 1 1.5022 1.57229 3.10131 -0.229655 -1.38638
0 0 0 0 0 1 0.698366 2.21009 -0.0249354 -0.137396
0 0 0 0 0 0 1 0.27822 -1.31087 1.37729
0 0 0 0 0 0 0 1 3.86782 4.27948
0 0 0 0 0 0 0 0 1 9.16021
0 0 0 0 0 0 0 0 0 1

1 1.03614 0.927711 0.180723 1.12048 0.421687 1.03614 1.10843 0.590361 0.253012
0 1 -0.864865 -1.27027 0.162162 -0 0.648649 1.13514 -0.945946 -0.540541
0 0 1 0.933333 0.571429 0.238095 0.12381 -0.504762 1.08571 0.590476
0 0 0 1 1.29545 1.38636 1.90909 0.909091 0 0.818182
0 0 0 0 1 1.5 1.58333 3.16667 -0.166667 -1.45833
0 0 0 0 0 1 0.710145 2.24638 -0.0144928 -0.173913
0 0 0 0 0 0 1 0.240741 -1.33333 1.35185
0 0 0 0 0 0 0 1 3.82609 4.52174
0 0 0 0 0 0 0 0 1 7.98113
0 0 0 0 0 0 0 0 0 1

```

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

1.3 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 system 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。

1.4 一些 GPU 编程学习资源

GPU 编程没有太多得技巧可言, “唯手熟尔”。只能在了解大致 GPU 架构和各类编程模型得特点上一点点看代码, 多练习。目前 GPU 编程模型包括 OpenCL, OpenACC, CUDA 等。CUDA 社区具有较为完备得学习资料, 且 Nvidia 在 GPU 的积累优势, 建议初学者以 CUDA 并行编程模型位主要学习主线。以下是笔者总结的一些学习资源:

1. 《CUDA-C 编程权威指南》。这本书是 CUDA 社区较早的一本书, 简要介绍了 CUDA 的编程模式, 内存管理等。该书可使读者使用 CUDA 平台、CUDA 工具包和 CUDA C 语言快速上手 GPU (图形处理单元) 计算。本书中设置的范例与练习也将带你快速了解 CUDA 的专业知识。
2. B 站上的各种 CUDA 编程视频, 由于缺乏大部分视频缺乏系统性安排, 建议刷视屏时候挑选感兴趣的部分。
3. CUDA 安装包自带的 Samples。这些示例是官方精心挑选的示例, 代码写的很简洁, 在读懂代码的基础上, 替换成自己需要解决的问题, 可以快速上手 CUDA。

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。

2 口令猜测选题

对于基础要求, 将多线程实验中需要加速的两个循环 (即 PT 内部的口令生成) 通过 gpu 进行并行化即可。

进阶要求如下:

1. 在 MPI 实验中, 如果你尝试了 PT 层面的并行 (即一次性取出多个 PT 并且进行并行生成), 在这次实验中同样可以尝试往 gpu 上一次装载多个 PT 进行生成。
2. 将任务打包好并传给 gpu, 然后等 gpu 计算完毕后传回 cpu 的这段时间, cpu 的计算资源被“浪费”了。与其让 cpu “忙等待”, 是否可以更好地“压榨”这段时间里 cpu 的计算能力, 将其充分利用起来?

3. 在先前的实验中，你可能会观察到一个现象：不同 PT 所能生成的口令总数天差地别。换言之，对于不同的 PT 而言，其进行口令生成时的计算量是有很大的差别的。如果你一次只向 GPU 传送一个 PT，是否可以根据不同 PT 之间的差别，在 cpu 计算和 gpu 计算上进行调整（有时候用 gpu 造成的开销会导致并行计算得不偿失）？如果你一次性进行多个 PT 的并行生成，相较于单次向 GPU 上传固定数目的 PT，是否有更好的调整方法？

3 ANN 选题

由于 GPU 很难对单个查询进行加速，所以 GPU 加速通常考虑将多个查询作为一个 batch，然后对一个 batch 的查询共同处理（即查询间并行）。大家可以回忆一下内积距离的计算实际上与矩阵的一行和一列相乘是一致的，所以 ANN 的距离计算实际可以转换为矩阵乘法进行计算。假设数据集 base data 大小为 n ，维度为 d ，一个 batch 的查询数量为 m 。那么可以将 base data 写成一个大小为 $n * d$ 的矩阵，将 query 写成一个大小为 $d * m$ 的矩阵，相乘后可以得到一个大小为 $n * m$ 的矩阵，最后在该矩阵的每一列中取距离最小的前 k 个点，即为该列对应查询的最近邻。大家可以探索如何使用 GPU 更快速的进行矩阵乘法，以及如何更快速的找到距离最小的前 k 个点。

大家同样可以思考在 IVF 上如何进行查询间并行。我们可以将 IVF 的一个簇里的向量也写成一个矩阵，在计算一个簇中向量到查询距离时使用矩阵乘法。但是，由于我们将查询向量作为一个 batch，batch 内不同查询向量的最近 nprobe 个簇的编号也不同，这样会导致矩阵乘法中一些计算是浪费的。所以我们可以考虑一些分组策略，尽可能让 batch 内的向量的最近簇的编号重合度较高，避免计算资源的浪费。

4 NTT 选题

可以先预处理所有需要用到的旋转因子，存储在 GPU 内存中，内部计算时快速访问旋转因子。

并行加速方面，与多线程多进程划分一致，对主体循环的第二三层循环进行划分，需要具体查阅对应 GPU 的 SM 数量和最大线程数确定分块大小、网格大小。

能够证明在 GPU 上，规约算法对模乘优化的加速比较高，需要同时对比基础模乘和两种优化模乘的加速比，注意 baseline 还应有 cpu 下的数据。