南開大學
Nankai University

计算机学院

并行程序设计第 6 次作业

高斯消去法的 CUDA 并行化

姓名：丁屹

学号：2013280

专业：计算机科学与技术

2022 年 6 月 15 日

目录

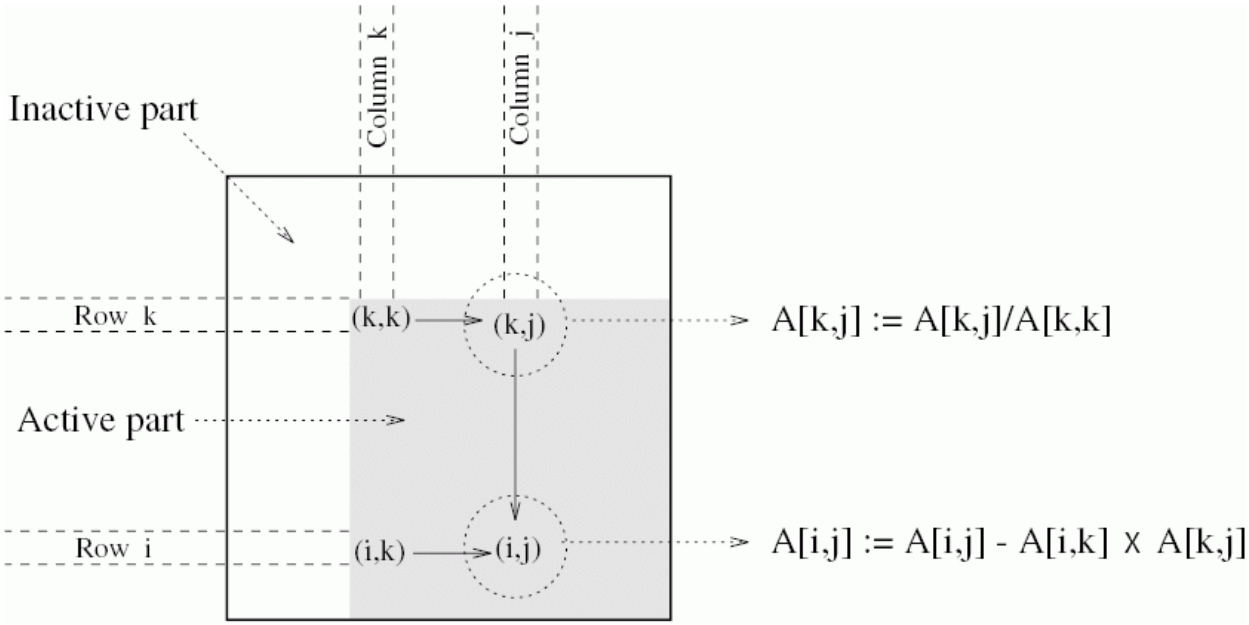

图 1.1: 高斯消去法示意图

1 问题描述

高斯消去的计算模式如图 1.1 所示，在第 k 步时，对第 k 行从 (k,k) 开始进行除法操作，并且将后续的 $k+1$ 至 N 行进行减去第 k 行的操作，串行算法如下面伪代码所示。

Algorithm 1 普通高斯消元算法伪代码

```
1:  function LU
2:      for k := 0 to n do
3:          for j := k + 1 to n do
4:              A[k, j] := A[k, j]/A[k, k]
5:          end for
6:          A[k, k] := 1.0
7:          for i := k + 1 to n do
8:              for j := k + 1 to n do
9:                  A[i, j] := A[i, j] − A[i, k] ∗ A[k, j]
10:             end for
11:             A[i, k] := 0
12:         end for
13:     end for
14: end function
```

观察高斯消去算法，注意到伪代码第 4，5 行第一个内嵌循环中的 $A[k,j] := A[k,j]/A[k,k]$ 以及伪代码第 8 9 10 行双层 for 循环中的 $A[i,j] := A[i,j]-A[i,k]\times A[k,j]$ 都是可以进行向量化的循环。可以通过 CUDA 对这两步进行并行优化。

2 算法设计

源码链接：https://github.com/ArcanusNEO/Parallel-Programming/tree/master/6

2.1 测试用例的确定

由于测试数据集较大，不便于各个平台同步，所以采用固定随机数种子为 12345687 的 mt19937 随机数生成器。经过实验发现不同规模下，所有元素独立生成，限制大小在 $[0, 100]$，能够生成可以被正确消元的矩阵。

代码如下：

```
uniform_real_distribution<float> dist(0, 100);
mt19937 mt(12345687);
int n;
istringstream iss(argv[1]);
iss >> n;
cout << n << endl;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= n; ++j) cout << dist(mt) << " \n"[j == n];
```

2.2 实验环境和相关配置

实验在本地 Arch Linux x86_64 平台完成，使用 cmake 构建项目，均开启 O3 加速；

- nVIDIA GPU 型号：NVIDIA GeForce RTX 2060 with Max-Q Design
- 驱动版本：Driver Version: 515.48.07
- CUDA 版本：CUDA Version: 11.7

2.3 默认平凡算法

使用一维数组模拟矩阵，避免改变矩阵大小时第二维不方便调整、必须设成最大值的问题，可以减少 cache 失效；

使用 $\#define\ matrix(i,j)\ arr[(i)*n+(j)]$ 宏，增强可读性；

```
void func(int& ans, float arr[], int n) {
  for (int k = 0; k < n; ++k) {
    for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
    matrix(k, k) = 1.0;
    for (int i = k + 1; i < n; ++i) {
      for (int j = k + 1; j < n; ++j)
        matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
      matrix(i, k) = 0;
    }
  }
}
```

2.4 CUDA 加速算法

对于访存，这里使用了基本的内存空间分配方式，即在 CPU 端给数据分配好内存空间并初始化以后，在 GPU 端对应地分配显存空间，然后显式地调用数据传输的接口将 CPU 端的数据传输至 GPU 端；最后，核函数执行完毕后，还需要把计算好的数据再从 GPU 端传输回来。

对于任务划分，使用了最简单直观的任务划分方法：除法和消去部分串行执行，分别编写两个核函数。

对于除法部分,每个线程负责第 K 次循环中单独一列的计算,因为通过 cudaDeviceProp::maxThreadsPerBlock 可以发现在我的显卡上每个线程块的最大线程数为 1024, 所以对于大于 1024×1024 的矩阵可以分成多个块并行执行。

```
__global__ void division_kernel(float arr[], int n, int k) {
  auto tid = blockDim.x * blockIdx.x + threadIdx.x;
  if (k < tid && tid < n) matrix(k, tid) /= matrix(k, k);
}
```

对于消去部分,在第 k 次循环中,对第 $k+1$ 行至最后一行进行消去,每一行计算完成后需要将这一行的第 k 列设为 0, 矩阵最后变为一个上三角矩阵。对于同步问题, 在核函数内部调用 ___syncthreads() 接口来同步块内线程，但是 GPU 的硬件架构导致的没有接口用于块间同步，所以为了保证正确同步的同时最大化利用 GPU 的并行性，我们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。

```
__global__ void eliminate_kernel(float arr[], int n, int k) {
  auto tx = blockDim.x * blockIdx.x + threadIdx.x;
  if (tx == 0) matrix(k, k) = 1.0;
  for (auto row = k + 1 + blockIdx.x; row < n; row += gridDim.x) {
    auto tid = threadIdx.x;
    for (auto col = k + 1 + tid; col < n; col += blockDim.x)
      matrix(row, col) -= matrix(row, k) * matrix(k, col);
    __syncthreads();
    if (tid == 0) matrix(row, k) = 0;
  }
}
```

在外层通用驱动程序 func 中，我们需要做的工作如下：

1. 分配合适大小的显存
2. 把内存中的矩阵复制进显存
3. 对外层循环的每一轮确定合适的线程块大小和线程块数量
4. 在外层循环的每一轮中依次调用 division_kernel 和 eliminate_kernel 函数并同步所有线程
5. 把显存中的结果复制回内存，释放显存

```
void func(int& ans, float arr[], int n) {
  float* gpu_arr;
  auto   siz = sizeof(float) * n * n;
```

```
  if (cudaMalloc(&gpu_arr, siz) != cudaSuccess)
    cerr << "cudaMalloc failed" << endl;
  if (cudaMemcpy(gpu_arr, arr, siz, cudaMemcpyHostToDevice) != cudaSuccess)
    cerr << "cudaMemcpyHostToDevice failed" << endl;

  for (int k = 0; k < n; ++k) {
    auto rest = n - k - 1;
    dim3 grid(std::max(std::ceil(rest / 1024.0), 1.0));
    dim3 block(1024);
    division_kernel<<<grid, block>>>(gpu_arr, n, k);
    cudaDeviceSynchronize();
    if (auto ret = cudaGetLastError(); ret != cudaSuccess)
      cerr << "division kernel failed: " << cudaGetErrorString(ret) << endl;
    dim3 eliminate_grid(32);
    eliminate_kernel<<<eliminate_grid, block>>>(gpu_arr, n, k);
    cudaDeviceSynchronize();
    if (auto ret = cudaGetLastError(); ret != cudaSuccess)
      cerr << "eliminate kernel failed: " << cudaGetErrorString(ret) << endl;
  }
  if (cudaMemcpy(arr, gpu_arr, siz, cudaMemcpyDeviceToHost) != cudaSuccess)
    cerr << "cudaMemcpyDeviceToHost failed" << endl;
  cudaFree(gpu_arr);
}
```

3 结果分析

使用 GPU 加速需要考虑到浮点数精度的问题，在 solve.cc 文件内有一段被注释的代码控制打印结果矩阵，四舍五入保留 4 位小数。比较 8×8 矩阵的计算结果图 3.2 和 3.3 发现二者结果完全一致；比较 16×16 矩阵的计算结果图 3.4 和 3.5 发现某些元素最后一位小数略有差异。

```
1.0000 0.3769 0.6592 0.5188 1.1310 0.2509 0.9155 1.1059
0.0000 1.0000 -0.6619 0.8717 -0.2745 0.3532 -1.0903 -0.5025
0.0000 0.0000 1.0000 -0.1824 0.2861 0.8056 1.1926 0.5901
0.0000 0.0000 0.0000 1.0000 -1.2733 -0.6247 0.4767 0.3878
0.0000 0.0000 0.0000 0.0000 1.0000 0.1061 -0.7968 0.2038
0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 8.2033 1.5449
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.2861
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
file: 8.in repeat: 100 avg-time: 0.000001372410 s
```

图 3.2: CPU 平凡算法 8×8 矩阵结果

经过测试得到表 1 发现大数据规模下 CUDA 加速效果显著，有十余倍性能提升。

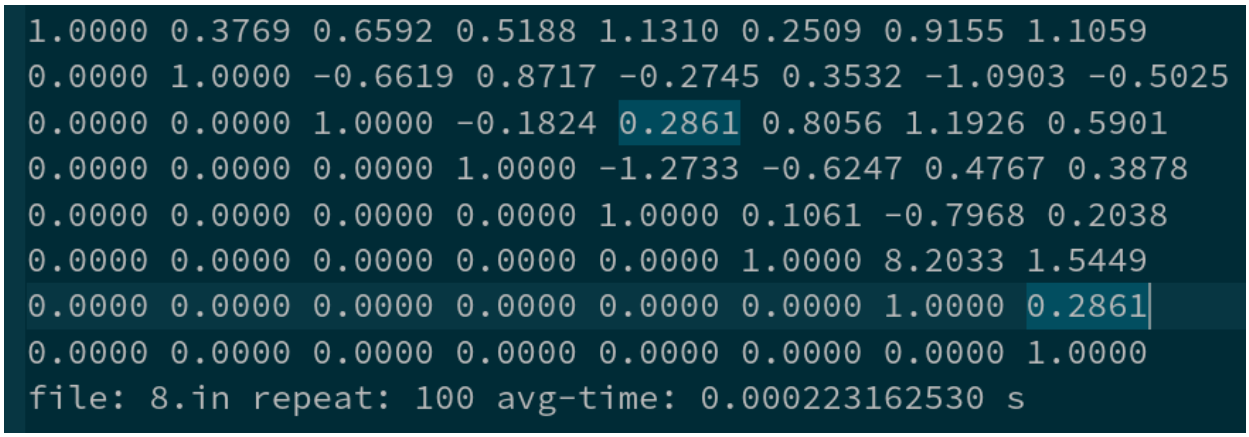

图 3.3: GPU CUDA 加速算法 8 × 8 矩阵结果

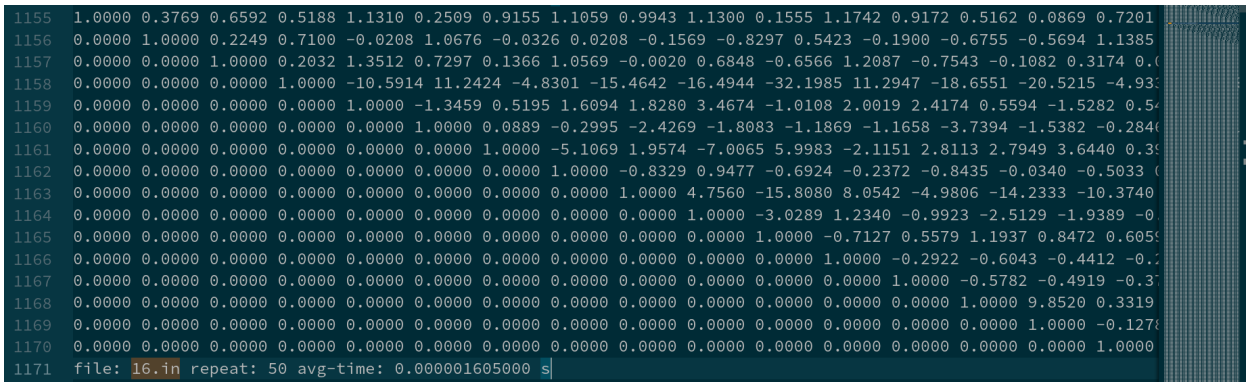

图 3.4: CPU 平凡算法 16 × 16 矩阵结果

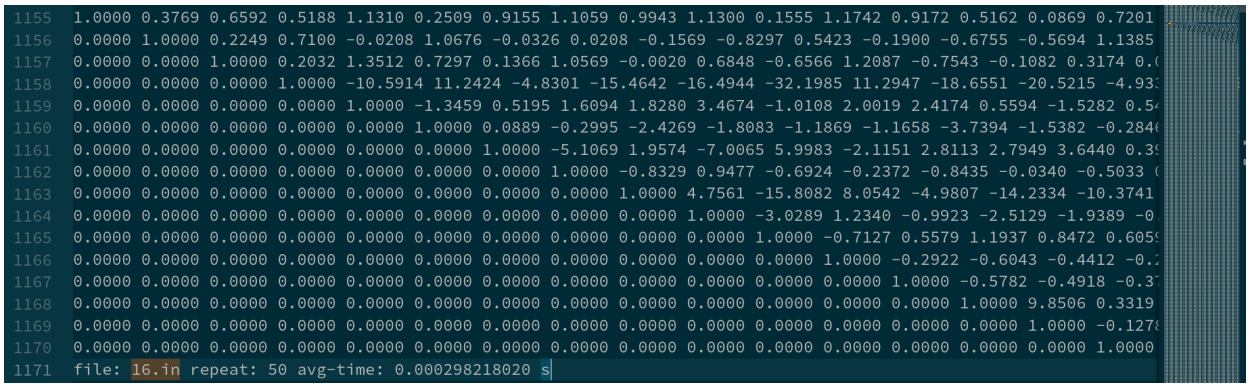

图 3.5: GPU CUDA 加速算法 16 × 16 矩阵结果

Scale	Reperat times	x86 ordinary (s)	x86 CUDA (s)
8×8	100	0.000001330460	0.000202092680
16 × 16	50	0.000001706920	0.000348178360
32 × 32	50	0.000003640080	0.000486327180
64 × 64	20	0.000015253300	0.000934846400
128 × 128	15	0.000098880800	0.002263211200
256 × 256	10	0.000716408500	0.003708655900
512 × 512	10	0.006722607300	0.008892275300
1024 × 1024	5	0.064893815400	0.054395600600
2048 × 2048	3	1.400074583333	0.240155499000
4096 × 4096	1	10.705585484000	1.255606927000

表 1: 平凡算法和 CUDA 加速算法运行时间