



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

<b>1 问题描述</b>	<b>2</b>
<b>2 算法设计</b>	<b>3</b>
2.1 测试用例的确定 . . . . .	3
2.2 实验环境和相关配置 . . . . .	3
2.3 默认平凡算法 . . . . .	3
2.4 CUDA 加速算法 . . . . .	4
<b>3 结果分析</b>	<b>5</b>

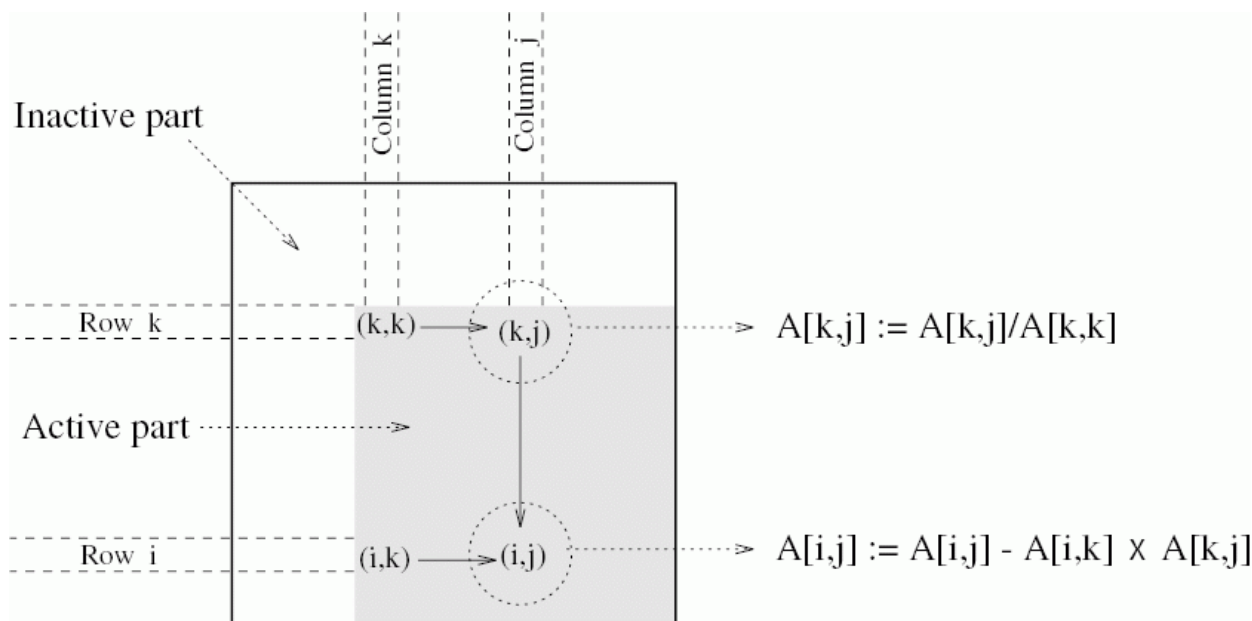


图 1.1: 高斯消去法示意图

## 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:

```
1 uniform_real_distribution<float> dist(0, 100);
2 mt19937 mt(12345687);
3 int n;
4 istream iss(argv[1]);
5 iss >> n;
6 cout << n << endl;
7 for (int i = 1; i <= n; ++i)
8     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;

```
1 void func(int& ans, float arr[], int n) {
2     for (int k = 0; k < n; ++k) {
3         for (int j = k + 1; j < n; ++j) matrix(k, j) = matrix(k, j) / matrix(k, k);
4         matrix(k, k) = 1.0;
5         for (int i = k + 1; i < n; ++i) {
6             for (int j = k + 1; j < n; ++j)
7                 matrix(i, j) = matrix(i, j) - matrix(i, k) * matrix(k, j);
8             matrix(i, k) = 0;
9         }
10    }
11 }
```

## 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  $1024 \times 1024$  的矩阵可以分成多个块并行执行。

```
1  __global__ void division_kernel(float arr[], int n, int k) {
2      auto tid = blockDim.x * blockIdx.x + threadIdx.x;
3      if (k < tid && tid < n) matrix(k, tid) /= matrix(k, k);
4  }
```

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。

```
1  __global__ void eliminate_kernel(float arr[], int n, int k) {
2      auto tx = blockDim.x * blockIdx.x + threadIdx.x;
3      if (tx == 0) matrix(k, k) = 1.0;
4      for (auto row = k + 1 + blockIdx.x; row < n; row += blockDim.x) {
5          auto tid = threadIdx.x;
6          for (auto col = k + 1 + tid; col < n; col += blockDim.x)
7              matrix(row, col) -= matrix(row, k) * matrix(k, col);
8          __syncthreads();
9          if (tid == 0) matrix(row, k) = 0;
10     }
11 }
```

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

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

```

4  if (cudaMalloc(&gpu_arr, siz) != cudaSuccess)
5      cerr << "cudaMalloc failed" << endl;
6  if (cudaMemcpy(gpu_arr, arr, siz, cudaMemcpyHostToDevice) != cudaSuccess)
7      cerr << "cudaMemcpyHostToDevice failed" << endl;
8
9  for (int k = 0; k < n; ++k) {
10     auto rest = n - k - 1;
11     dim3 grid(std::max(std::ceil(rest / 1024.0), 1.0));
12     dim3 block(1024);
13     division_kernel<<<grid, block>>>(gpu_arr, n, k);
14     cudaDeviceSynchronize();
15     if (auto ret = cudaGetLastError(); ret != cudaSuccess)
16         cerr << "division kernel failed: " << cudaGetErrorString(ret) << endl;
17     dim3 eliminate_grid(32);
18     eliminate_kernel<<<eliminate_grid, block>>>(gpu_arr, n, k);
19     cudaDeviceSynchronize();
20     if (auto ret = cudaGetLastError(); ret != cudaSuccess)
21         cerr << "eliminate kernel failed: " << cudaGetErrorString(ret) << endl;
22 }
23 if (cudaMemcpy(arr, gpu_arr, siz, cudaMemcpyDeviceToHost) != cudaSuccess)
24     cerr << "cudaMemcpyDeviceToHost failed" << endl;
25 cudaFree(gpu_arr);
26 }

```

### 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  $8 \times 8$  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  $16 \times 16$  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  $8 \times 8$  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。

```

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: 0.000223162530 s

```

图 3.3: GPU CUDA 加速算法  $8 \times 8$  矩阵结果

```

1155 1.0000 0.3769 0.6592 0.5188 1.1310 0.2509 0.9155 1.1059 0.9943 1.1300 0.1555 1.1742 0.9172 0.5162 0.0869 0.7201
1156 0.0000 1.0000 0.2249 0.7100 -0.0208 1.0676 -0.0326 0.0208 -0.1569 -0.8297 0.5423 -0.1900 -0.6755 -0.5694 1.1385
1157 0.0000 0.0000 1.0000 0.2032 1.3512 0.7297 0.1366 1.0569 -0.0020 0.6848 -0.6566 1.2087 -0.7543 -0.1082 0.3174 0.0
1158 0.0000 0.0000 0.0000 1.0000 -10.5914 11.2424 -4.8301 -15.4642 -16.4944 -32.1985 11.2947 -18.6551 -20.5215 -4.93
1159 0.0000 0.0000 0.0000 0.0000 1.0000 -1.3459 0.5195 1.6094 1.8280 3.4674 -1.0108 2.0019 2.4174 0.5594 -1.5282 0.5
1160 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0889 -0.2995 -2.4269 -1.8083 -1.1869 -1.1658 -3.7394 -1.5382 -0.284
1161 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -5.1069 1.9574 -7.0065 5.9983 -2.1151 2.8113 2.7949 3.6440 0.3
1162 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.8329 0.9477 -0.6924 -0.2372 -0.8435 -0.0340 -0.5033
1163 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 4.7560 -15.8080 8.0542 -4.9806 -14.2333 -10.3740
1164 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -3.0289 1.2340 -0.9923 -2.5129 -1.9389 -0.
1165 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.7127 0.5579 1.1937 0.8472 0.605
1166 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.2922 -0.6043 -0.4412 -0.
1167 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.5782 -0.4919 -0.3
1168 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 9.8520 0.3319
1169 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.127
1170 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1171 file: 16.in repeat: 50 avg-time: 0.000001605000 s

```

图 3.4: CPU 平凡算法  $16 \times 16$  矩阵结果

```

1155 1.0000 0.3769 0.6592 0.5188 1.1310 0.2509 0.9155 1.1059 0.9943 1.1300 0.1555 1.1742 0.9172 0.5162 0.0869 0.7201
1156 0.0000 1.0000 0.2249 0.7100 -0.0208 1.0676 -0.0326 0.0208 -0.1569 -0.8297 0.5423 -0.1900 -0.6755 -0.5694 1.1385
1157 0.0000 0.0000 1.0000 0.2032 1.3512 0.7297 0.1366 1.0569 -0.0020 0.6848 -0.6566 1.2087 -0.7543 -0.1082 0.3174 0.0
1158 0.0000 0.0000 0.0000 1.0000 -10.5914 11.2424 -4.8301 -15.4642 -16.4944 -32.1985 11.2947 -18.6551 -20.5215 -4.93
1159 0.0000 0.0000 0.0000 0.0000 1.0000 -1.3459 0.5195 1.6094 1.8280 3.4674 -1.0108 2.0019 2.4174 0.5594 -1.5282 0.5
1160 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0889 -0.2995 -2.4269 -1.8083 -1.1869 -1.1658 -3.7394 -1.5382 -0.284
1161 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -5.1069 1.9574 -7.0065 5.9983 -2.1151 2.8113 2.7949 3.6440 0.3
1162 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.8329 0.9477 -0.6924 -0.2372 -0.8435 -0.0340 -0.5033
1163 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 4.7561 -15.8082 8.0542 -4.9807 -14.2334 -10.3741
1164 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -3.0289 1.2340 -0.9923 -2.5129 -1.9389 -0.
1165 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.7127 0.5579 1.1937 0.8472 0.605
1166 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.2922 -0.6043 -0.4412 -0.
1167 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.5782 -0.4918 -0.3
1168 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 9.8506 0.3319
1169 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 -0.127
1170 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1171 file: 16.in repeat: 50 avg-time: 0.000298218020 s

```

图 3.5: GPU CUDA 加速算法  $16 \times 16$  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