基于 CUDA 的 Gauss-Seidel 迭代法求解 Poisson 方程

姜腾

Teng Jiang

信息科学技术学院

School of EECS, Peking University

2023 年 1 月 10 日

目录

1 问题陈述

考虑在 $\Omega = [0,1] \times [0,1] \times [0,1]$ 上的 Poisson 方程:

$$\begin{cases} -\Delta u(x,y,z) = f(x,y,z), & (x,y,z) \in \Omega \\ u(x,y,z) = 0, & (x,y,z) \in \partial\Omega \end{cases}$$

为了可以数值计算, 我们将其离散化: 将三边分别等分为 $N+1$ 份, 网格步长 $h = 1/(N+1)$. 则 $x_i = ih, y_j = jh, z_k = kh$, 其中 $i,j,k = 0,\ldots,N+1$. 并记 $U_{i,j,k} = U(x_i, y_j, z_k)$. 采用二阶中心差分格式离散得到如下方程组:

$$\begin{cases} 6U_{i,j,k} - U_{i-1,j,k} - U_{i+1,j,k} - U_{i,j-1,k} - U_{i,j+1,k} - U_{i,j,k-1} - U_{i,j,k+1} = h^2 f_{i,j,k}, 1 \le i,j,k \le N \\ U_{0,j,k} = U_{N+1,j,k} = U_{i,0,k} = U_{i,N+1,k} = U_{i,j,0} = U_{i,j,N+1} = 0, 0 \le i,j,k \le N+1 \end{cases}$$

其中变量有 N^3 个, 为 $U_{i,j,k}, 1 \le i,i,j \le N$.

Poisson 方程离散得到的矩阵 A 是一个稀疏矩阵, 其 Gauss-Seidel 迭代相当于每步求解如下关于 $U^*_{i,j,k}$, 其中 $1 \le i,j,k \le N$ 的线性方程组:

$$\begin{cases} 6U^*_{i,j,k} - U^*_{i-1,j,k} - U_{i+1,j,k} - U^*_{i,j-1,k} - U_{i,j+1,k} - U^*_{i,j,k-1} - U_{i,j,k+1} = h^2 f_{i,j,k}, & 1 \le i,j,k \le N \\ U^*_{0,j,k} = U^*_{N+1,j,k} = U^*_{i,0,k} = U^*_{i,N+1,k} = U^*_{i,j,0} = U^*_{i,j,N+1} = 0, & 0 \le i,j,k \le N+1 \\ U_{0,j,k} = U_{N+1,j,k} = U_{i,0,k} = U_{i,N+1,k} = U_{i,j,0} = U_{i,j,N+1} = 0, & 0 \le i,j,k \le N+1 \end{cases}$$

2 更新策略

2.1 Jacobi 法

如果每次更新只用上一次迭代的值进行更新，实则求解问题为如下问题:

$$\begin{cases} 6U^*_{i,j,k} - U_{i-1,j,k} - U_{i+1,j,k} - U_{i,j-1,k} - U_{i,j+1,k} - U_{i,j,k-1} - U_{i,j,k+1} = h^2 f_{i,j,k}, & 1 \le i,j,k \le N \\ U^*_{0,j,k} = U^*_{N+1,j,k} = U^*_{i,0,k} = U^*_{i,N+1,k} = U^*_{i,j,0} = U^*_{i,j,N+1} = 0, & 0 \le i,j,k \le N+1 \\ U_{0,j,k} = U_{N+1,j,k} = U_{i,0,k} = U_{i,N+1,k} = U_{i,j,0} = U_{i,j,N+1} = 0, & 0 \le i,j,k \le N+1 \end{cases}$$

由于 Jacobi 法是用上一次迭代的 U 矩阵的值更新这一次迭代的 U 矩阵，各个格点的更新之间不存在数据依赖, 所以这种方法天然地适合并行处理。这种方法的缺点是由于每次迭代后面更新的但没有用到当次迭代已经更新到的格点的值，所以收敛速度相较于 Gauss-Seidel 法会需要更多次迭代。

Jacobi 法的 kernel 函数如下：

```
1
2  __global__
3  void jacobi_kernel(double *__restrict__ d_u, double *__restrict__ d_u_new,
       double *__restrict__ d_b){
4      int k = blockDim.x * blockIdx.x + threadIdx.x;
5      int j = blockDim.y * blockIdx.y + threadIdx.y;
6      int i = blockDim.z * blockIdx.z + threadIdx.z;
7
8      d_u_new[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1] =
9          (d_b[i * N * N + j * N + k]
10         + d_u[(i + 0) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1]
11         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 0) * (N + 2) + k + 1]
12         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 0]
13         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 2]
14         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 2) * (N + 2) + k + 1]
15         + d_u[(i + 2) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1]
16         ) / 6.0;
17 }
```

其中 d_u_new 为本次迭代更新的值存放的数组，d_u 为上一次迭代的结果。所以在每次迭代结束后，需要交换两者的指针，完成一次更新。

```
1          //switch pointer
2          temp = d_u_new;
3          d_u_new = d_u;
4          d_u = temp;
```

2.2 红黑染色的 Gauss-Seidel 法

为了解决 Jacobi 因为更新值利用不充分而导致收敛较慢的问题，Gauss-Seidel 法充分利用了当次迭代已经更新的值，所以会有更快的收敛。缺点是，严格的 Gauss-Seidel 法每次迭代需要严格串行，这对我们并行计算带来困难。于是我们希望一定程度上放松部分依赖关系获得一定可以进行并行的空间。其中比较好实现的是红黑法，如下图所示：

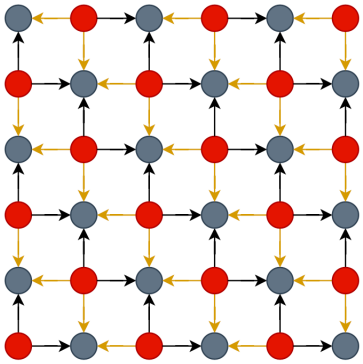

图 1: 红黑着色重排序的 Gauss-Seidel 法

区分红格点和黑格点的最简单的方法就是看其三个维度的坐标之和为奇数还是偶数。我们可以定义三个维度坐标之和为偶数为红色节点，之和为奇数的为黑节点。这样每个节点的周围 6 个节点都是这个节点相反的颜色，我们只需要交替更新两个颜色的节点即可。Kernel 函数如下：

```
1  void gauss_seidel_kernel(double *__restrict__ d_u, double *__restrict__
       d_b,int num){
2      int k = blockDim.x * blockIdx.x + threadIdx.x;
3      int j = blockDim.y * blockIdx.y + threadIdx.y;
4      int i = blockDim.z * blockIdx.z + threadIdx.z;
5
6      if((i+j+k+num)%2==0){ // 判断颜色
7      d_u[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1] =
8          (d_b[i * N * N + j * N + k]
9          + d_u[(i + 0) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1]
10         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 0) * (N + 2)+ k + 1]
11         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 0]
12         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 2]
13         + d_u[(i + 1) * (N + 2) * (N + 2) + (j + 2) * (N + 2) + k + 1]
14         + d_u[(i + 2) * (N + 2) * (N + 2) + (j + 1) * (N + 2) + k + 1]
15         ) / 6.0;}
16 }
```

2.3 分块 Gauss-Seidel 法

另一种放松 Gauss-Seidel 法的数据依赖的方法是将矩阵分块，块内进行 Gauss-Seidel 更新，块之间进行 Jacobi 更新。但是这样其实也损失了至少一半并行性，而且由于边界问题内存加载也比较有挑战，再加上由于实验要求至少迭代 34 次，所以 Gauss-Seidel 法收敛快的优势不易很好体现，这种思想在本次实验中未进行实现，但仍然是一种重要的潜在方法。

3 残差范数计算策略

3.1 Reduction on GPU

每次迭代需要计算残差范数以确定此次迭代是否达到收敛标准。计算残差的方案有两种：

- 每次计算完残差，保存残差矩阵 ($512 \times 512 \times 512$) 传回 CPU 计算。这样传输开销巨大，故不考虑这种方法。
- 每次计算完残差，保存残差矩阵，然后在 GPU 上进行 Sum Reduction。算出结果后只需传输一个数回 CPU 判断敛散。这是本次实验考虑的方法。

如果用 Kernel 全局变量保存残差之和，这样每个格点把自己的残差加入总残差时访存的开销比较大。一种比较好的方法是用 shared memory 保存每个 block 的总残差，这样访存效率大大提升。每个 block 之间的 shared memory 是不共享的，所以对每个 block 的残差和，我们需要进一步 reduction。在本次实验中进行 3 次 Sum reduction，reduction 的方式用的是二叉树层级求和，具体的 reduction 的流程如下图所示，每个 block 有 512 个线程，每次 reduction 使得总元素数/512.

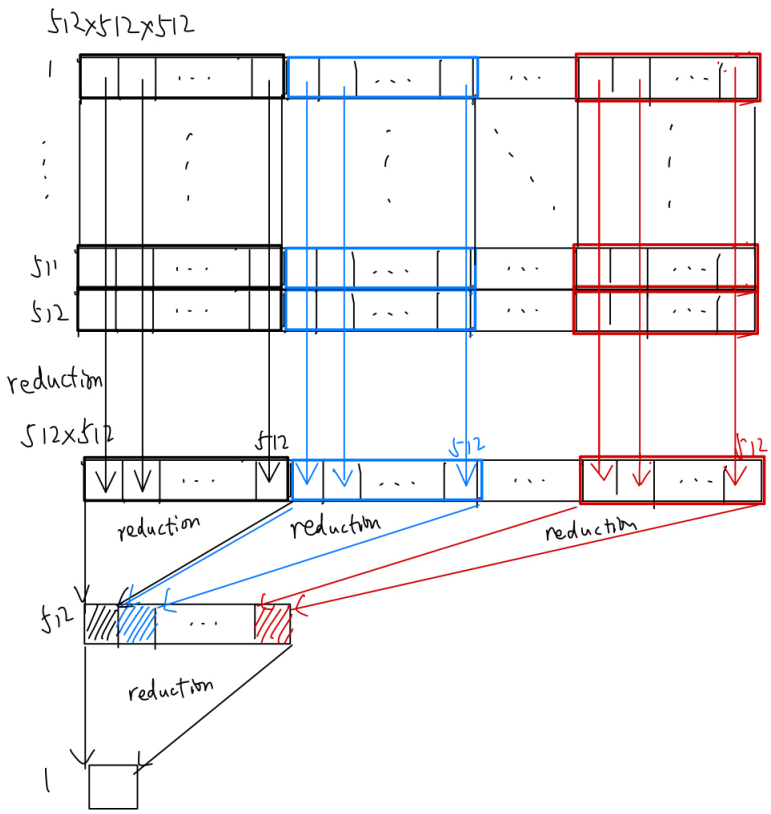

图 2: Reduction 示意图

Kernel 端的 reduction 函数如下所示，使用二叉树策略，代码注释已经比较清晰：

```
__global__ void reduction(float* g_odata, float* g_idata, int len) {
    // first, each thread loads data into shared memory
    __shared__ float sdata[N];

    int tid = threadIdx.x;
    int i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;
    sdata[tid] = 0;

    // each thread loads data into shared memory, and do the first round of
        binary tree reduction
    if(i<len) sdata[tid] = g_idata[i] + g_idata[i + blockDim.x];

    __syncthreads();

    // do reduction in shared mem
```

```
    // this loop starts with s = 512 / 2 = 256
    for (unsigned int s = blockDim.x >> 1; s > 0; s >>= 1) {
        if (tid < s) sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    // finally, first thread puts result into global memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

函数调用方法如下，一共调用三次得到最终残差再开方得到残差范数。需要开两个数组来充当输入和输出。

```
        reduction<<<N*N,N>>>(d_rn_out,d_rn);
        reduction<<<N,N>>>(d_rn,d_rn_out);
        reduction<<<1,N>>>(d_rn_out,d_rn);
```

4 参数设置与性能比较

4.1 Grid Dim 和 Block Dim 的设置对性能的影响

考虑到 memory coalescing 的问题，如果一个 wrap 里的线程取到的地址相对连续是对性能友好的。这体现在我们如果把 block Dim 的第一个维度取成 32 的倍数，理论上会对实验性能有一定提升。（这样被规划到同一个 wrap 的格点地址比较连续）事实上也是这样的，我们在后面进行对比。

4.2 将双精度改写为单精度

由于我们的实验采用的是 Titan XP(GP102) 型号，其中单精度核心比双精度核心多很多，所以考虑如果能在保证精度的前提下把程序中的 double 换为 float 的话，可能可以达到提升效率的目的。实际分别将 Jacobi 和红黑 Gauss-Seidel 法用 float 改写，均实现了性能的提升，将在实验结果部分对比。

4.3 部分实验结果展示与对比

在具体实验中，红黑 Gauss-Seidel 法和 Jacobi 法分别用 21 次与 41 次能达到收敛，但是实际上由于我们要求至少 34 次迭代，所以红黑 Gauss-Seidel 法需要额外迭代 13 次。实验结果如下：

表 1：运行时间 (s) 对比

算法		Jacobi (41 iters)		Gauss-Seidel (34 iters)	
Grid Dim	Block Dim	double	float	double	float
$(16, 64, 128)$	$(32, 8, 4)$	1.00171	0.630003	1.17823	0.723864
$(128, 64, 16)$	$(4, 8, 32)$	0.908164	未实验	未实验	未实验

表 2：带宽（GB/s）对比

算法		Jacobi (41 iters)		Gauss-Seidel (34 iters)	
Grid Dim	Block Dim	double	float	double	float
$(16, 64, 128)$	$(32, 8, 4)$	205.613	163.463	144.963	117.978
$(128, 64, 16)$	$(4, 8, 32)$	113.396	未实验	未实验	未实验

表 3：输出文件号

算法		Jacobi (41 iters)		Gauss-Seidel (34 iters)	
Grid Dim	Block Dim	double	float	double	float
$(16, 64, 128)$	$(32, 8, 4)$	142244	142324	142245	142612
$(128, 64, 16)$	$(4, 8, 32)$	142615	未实验	未实验	未实验

对 block dim 的对比可以看出 memory coalescing 的重要性（一个 wrap 内 32 个线程），一个 wrap 内线程调用地址相对连续对访存友好。对 float 和 double 比较可以看出使用 float 可以对性能进行改进，同时不影响收敛和精确度。最终性能最好的是使用 float，block dim 为 (32,8,4) 的 Jacobi 法。

5 未来展望

5.1 一些失败的尝试：Jacobi 法和分块 Gauss—Seidel 法中用共享内存

我们发现每一个格点在更新过程中会被重复访问 6 次，所以将其加载到 shared memory 中可能会对性能有提升。但实则操作时，对于一个 $M \times N \times K$ 的 block，要加载 $(M+1)(N+1)(K+1)-8$ 个格点，而且每个格点最多也只能用到 4 次，期间还可能出现 branch divergence 的问题，所以最后这个方法并没有能对性能提升，源代码保存在了 *poisson_cuda_jacobi_share.cu* 中。

5.2 对红黑法的改进

发现其实红黑法每次只更新所有格点的一半，所以考虑其实每次开的 Block 数可以少一半。同时每次计算残差的时候（假设我们每一次迭代先更新红再更新黑），其实后被更新的黑色格点残差应该是 0 的，因为他刚被根据附近的值更新过。但是这就需要比较高的格点 indexing 的技巧，但也是一个优化方向。

6 结论

最终性能最好的是使用 float，block dim 为 (32,8,4) 的 Jacobi 法，时间为 0.630003 秒，保存在 *job_142324_gpu01.out* 中，并拷贝了一份 *final.out*。如果允许红黑法 21 个迭代收敛即停止，可以达到 0.47638 秒，详见 *job_142617_gpu01.out* 输出。