CUDA 下单源最短路径算法并行优化

张 晗，钱育蓉+，王跃飞，陈人和，田宸玮

（新疆大学 软件学院，新疆 乌鲁木齐 830008）

摘 要：为设计基于固定序的 Bellman-Ford 算法在 CUDA 平台下并行优化方案，结合算法计算密集和数据密集的特点。从核函数计算层面，提出访存优化方法和基于固定序优化线程发散；从 CPU-GPU 传输层面，提出基于 CUDA 流优化数据传输开销方法。对不同显卡进行测试，参照共享内存容量划分线程块、缩减迭代后向量维度并使用 CUDA 流缩短首次计算时延，相比传统算法，改进后并行算法加速比在 200 倍左右。该并行优化方案验证了固定序在 CUDA 平台具有可行性和可移植性，可作为多平台研究参照。

关键词：固定序改进算法；Bellman-Ford 算法；并行计算；性能可移植性；图形处理器；统一计算设备架构
中图法分类号：TP391 **文献标识号**：A **文章编号**：1000-7024（2019）08-2181-09
doi：10.16208/j.issn1000-7024.2019.08.014

Parallel optimization of single source shortest path algorithm under CUDA

ZHANG Han, QIAN Yu-rong+, WANG Yue-fei, CHEN Ren-he, TIAN Chen-wei

(School of Software, Xinjiang University, Urumqi 830008, China)

Abstract: To design a parallel optimization scheme based on the fixed-order Bellman-Ford algorithm on the CUDA platform, the algorithm was computationally intensive and data-intensive. From the computational level of kernel function, the memory access optimization method and the fixed-order optimization thread divergence were proposed. From the CPU-GPU transmission level, the data transmission overhead method based on CUDA stream was proposed. After testing different graphics cards, the thread block was divided with reference to the shared memory capacity, the vector dimension was reduced after iteration, and the first calculation delay was shortened using the CUDA stream. The improved parallel algorithm has an acceleration ratio of about 200 times compared with the conventional algorithm. The parallel optimization scheme verifies that the fixed order is feasible and portable on the CUDA platform and can be used as a reference for multi-platform research.

Key words: improved fixed order algorithm; Bellman-Ford algorithm; parallel computing; performance portability; GPU; CUDA

0 引 言

基于固定序的 Bellman-Ford 算法（以下简称改进后算法）通过预先订制参与迭代点的计算顺序，使计算效率获得较大提升，在大规模稠密图上体现了显著优势[1]。常见单源最短路径算法包括 Dijkstra 算法[2,3]和 Bellman-Ford 算法[4,5]。韩伟一等提出了基于固定序的 Bellman-Ford 算法[6]，对上次迭代后最短路径值未改变的点进行标记，以降低本次迭代中参与计算点的数量，使得在稠密图上相比主流的先进先出算法有更卓越的计算效率。其算法整体流程：每次迭代中计算各点路径值、选最小值更新为最短路径值、进行下一次迭代、最终获得最短路径，虽然在每次迭代中算法只计算最短路径值被更新的点并且对各点计算步骤一致，属于 SIMT 计算模式[7]，但串行算法处理耗时依然较长，需设计和验证改进后方案是否适合并行计算。刘磊等对 Bellman-Ford 算法性能可移植的 GPU 并行优化[8]，在 GPU 上利用 OpenCL 架构提出访存优化和极值优化的方法提升了并行粒度，但该方案依然存在一定的访存

收稿日期：2018-07-04；修订日期：2019-03-21
基金项目：国家自然科学基金项目（61562086、61462079）；新疆维吾尔自治区创新团队基金项目（XJEDU2017T002）
作者简介：张晗（1987-），男，辽宁本溪人，硕士研究生，CCF 学生会员，研究方向为软件理论与服务计算；+通讯作者：钱育蓉（1980-），女，山东武城人，博士，教授，硕士生导师，CCF 高级会员，研究方向为网络计算和遥感图像处理；王跃飞（1991-），男，新疆乌鲁木齐人，博士研究生，研究方向为机器学习和数据挖掘；陈人和（1993-），男，重庆人，硕士研究生，研究方向为软件理论与服务计算；田宸玮（1995-），男，陕西西安人，硕士研究生，研究方向为软件理论与服务计算。E-mail：qyr@xju.edu.cn

延迟且未在 CUDA 体系架构下验证其可行性。本文从计算效率和访存优化角度分析并行算法性能，以及在不同显卡上进行测试，验证并行算法移植特性。

1 改进后 Bellman-Ford 算法

Bellman-Ford 算法是为了解决单源最短路径问题公认的最好算法之一。固定序算法是使用邻接矩阵存储稠密图的一种基本改进算法且具有明显优势。其特点是因每次迭代参与计算的点数量不确定，为提高算法计算效率，故每次迭代过程中对参与计算的点遵照固定顺序，在下次迭代计算过程中仅计算距离变更的点，相比传统算法来说，提升了算法计算效率。

算法描述如下：对给定带权有向图，图 $G=(V,E)$，其源点为 0 号节点，加权函数 w 是边集 E 的映射，用 $w(i,j)$ 代表有向边 (i,j) 的权重。$d(i)$ 代表点的距离标号，k 为算法的迭代次数。

步骤 1 初始化阶段。设 $d(0)=0, d(i)=+\infty (2\leqslant i\leqslant n)$，其中 $n=|V|$ 为节点个数。把 0 号节点加入集合 A 中，令迭代次数 $k=1$。

步骤 2 在第 k 次迭代中，按照编号顺序对集合 A 中各点进行如下运算

$$d(j)=\min\{d(j),d(i)+w(i,j)\}\quad (i,j)\in E$$

若 $d(j)$ 变小，则当 $j\notin A$ 时，将点 j 加入集合 A，并从集合中移除 i。

步骤 3 若集合 $A=\varnothing$，则算法结束，$d(i)$ 就是从原点到 i 点的最短距离；若集合 $A\neq\varnothing$ 且 $k=n-1$，可判断存在负循环，算法结束；若集合 $A\neq\varnothing$ 且 $k<n-1$ 时，算法执行步骤 4。

步骤 4 $k=k+1$，转到步骤 2。

计算过程如下：

算法 1：基于固定序 Bellman-Ford 算法

```
Bellman-Ford (G,w,s) //图 G，边集函数 w，源点为 0，集合 d_k 和 p_k 代表第 k 次迭代存放短距离和前驱节点集合
//步骤 1：初始化节点最短路径值和前驱节点
(1)for each vertex v∈V(G) do
(2)  d_0[v]=+∞;  p_0[v]=-1;
(3)end for
(4)d_0[0] ←0; //把 0 号节点加入集合 A，准备进入第一次迭代
//步骤 2、步骤 4：对所有边进行松弛迭代运算
(5)for k=1 to |v|-1 do
//计算第 k 次迭代各节点 v 最短路径长度 d_k[v]和前驱节点 p_k[v]，存入集合
(6)    for i=0 to |v|-1 do
(7)        dlin[v]=+∞; //临时存放最短路径长度
(8)        plin[v]=-1; //临时存放前驱节点
(9)    for  j=0 to |v|-1 do
(10)       if dlin[v]> d_k-1[j]+ w(j,v) then //松弛判断
(11)           dlin[v]=d_k-1[j]+w(j,v); //松弛操作
(12)           plin[v]=j; //更新前驱节点
(13)   end for
(14)   d_k[v]=min(dlin[v],d_k-1[v]); //取最短的距离，存入集合//确定最终前驱节点
(15)   if d_k[v]<d_k-1[v] then
(16)       p_k[v]=plin[v];
(17)   else
(18)       p_k[v]=p_k-1[v];
(19) end for//步骤 3;
(20) if d_k[v]=未更新 then
(21)   return d_k-1[v];
(22) if d_k[v]=未更新 and k=|v|-1 then
(23)   return -1;//算法可能存在负循环
(24)end for
(25)end procedure
```

可见，改进后算法对下一次参与迭代运算节点数目做了筛选，仅计算在前一次迭代过程中距离标量发生改变的节点并更新对应的前驱节点，极大地减少了下次迭代参与计算点的数量，提高了计算效率。但其串行算法需对距离标量是否变更逐个标记，此操作对应的时间复杂度为 $O(|V|)$。另外，每次迭代计算是串行访问 $w(i,j)$ 二维表中对应有向边 (i,j) 的权重，串行算法对内存访问存在空间局部性，若考虑并行计算，可提升对存储数据的使用率。因此，对改进后的串行算法来说，设计性能较优的并行算法不仅需要对固定序设计优化方案，同时还需考虑内存访问对性能造成的影响，利用访存空间局部性的特点，从而达到优化访存的目的。下节分别从访存优化、固定序优化和分流优化 3 个方面详细阐述设计方案。

2 基于 CUDA 对改进后算法的并行优化

2.1 算法流程

CUDA 架构在 NVIDIA 显卡上表现出更强的并行计算性能[9,10]，下面给出基于固定序的 Bellman-Ford 算法并行设计方案流程如图 1 所示（标号与各节对应）。

（1）主机分配页锁定内存，读入权代价矩阵到页锁定内存，以避免操作系统分页机制对数据造成频繁换页，影响 CPU-GPU 传输；

（2）主机将数据传送到 GPU 全局内存；

（3）每次迭代中主机都将对 GPU 发起执行命令，此时由多个 GPU 线程块，分别对不同节点计算短路径和前驱节点；

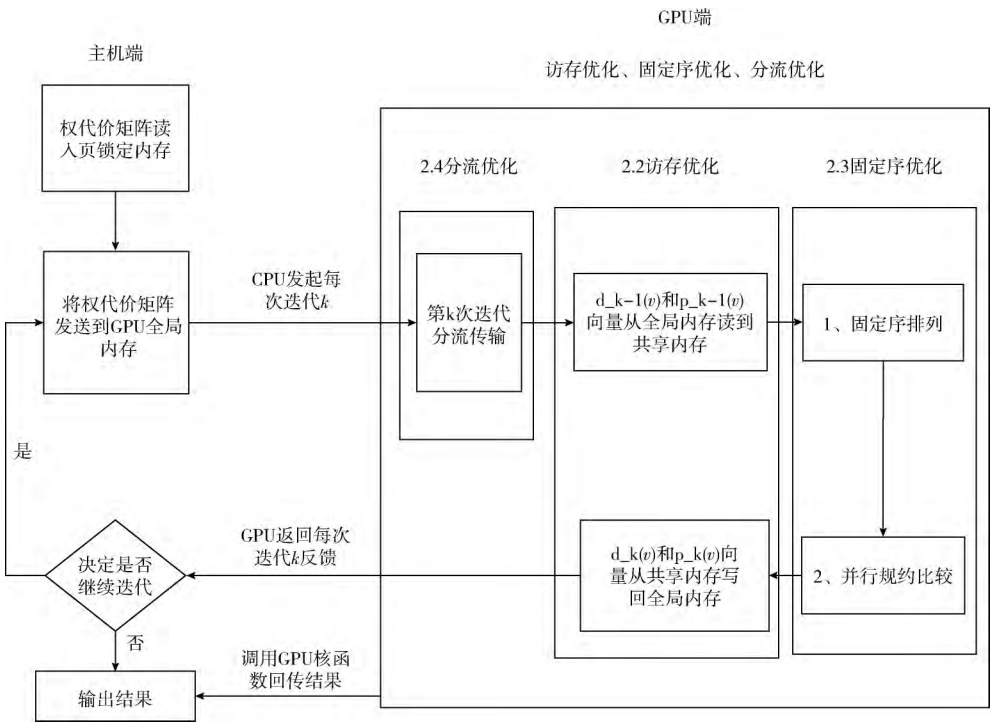


图 1　算法流程

（4）GPU 若完成一次迭代后，由主机再次对 GPU 发起执行命令，直到迭代次数达到上限；

（5）若迭代次数达到上限，将最终结果传输给主机端。

其中步骤（3）、步骤（4）、步骤（5）主机对 GPU 详细迭代过程设计如图 2 所示。

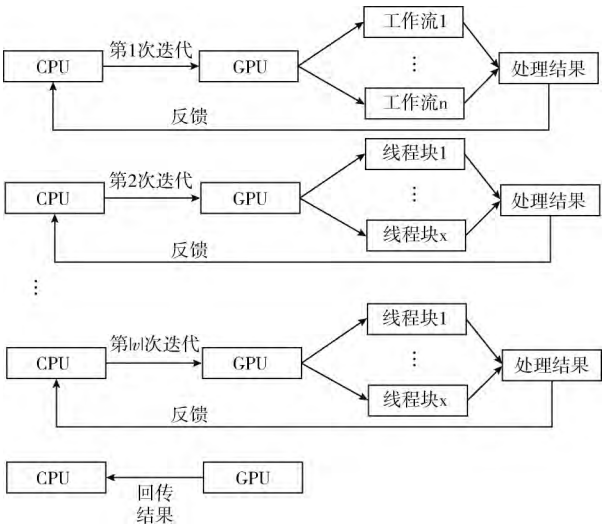


图 2　迭代模式

2.2　访存优化设计方案

设计访存优化方案目的是为了减少读取全局内存次数的问题。具体来说，减少在每次迭代过程中频繁读取全局内次数，避免生成结果频繁写入全局内存，对计算效率造成影响。结合线程块各自计算特点，将全局内存中最短距离向量读入到各线程块对应的共享内存中，每次迭代后生成的结果（当前最短距离向量和前驱节点向量）去替换全局内存旧的数据。结合数据特点，针对算法 1 中第（6）～第（19）行，可采取并行计算方式计算出第 k 次迭代中每个节点 v 的最短距离 d_k(v) 和前驱节点 p_k(v)，设计方案和数据存储结构如图 3 所示。

为了便于说明，列举了 9 个节点构成的有向带权图。图 3（b）代表存储于全局内存中的邻接矩阵创建了 3 个线程块，每个线程块包含 3 个线程 T(3)。图 3（c）所示，在第 2 次迭代中计算每个节点的最短距离 d_2(v) 需参考第一次迭代 d_1(v) 结果，求取各节点 d_2(v) 的值互不影响，属于 SIMT 计算模式，为方便表述创建线程规模为 Block(3) 分组计算，以提高计算效率。图中 d_1(v) 可预先读入到共享内存中，由于存在部分节点最短距离值未更新，需在共享内存中复制一份 d_1(T(3)) 部分副本 d_2(T(3)) 作为结果向量，当线程块计算完成后再将 d_2(v) 回传至全局内存中。同理，如图 3（d）描述的是前驱节点向量，因向量每个元素对应一个节点编号，无需保存副本，可将全局内存中 p_1(v) 读入到共享内存中，若最短距离向量发生更新，需同时更新前驱节点向量对应的节点编号，在一轮迭代完成后，用 p_1(T(3)) 替换全局内存中对应位置的前驱节点向量，命名为 p_2(v)。

需要说明：①从数据分散度上分析：邻接矩阵在全局内存中是按行连续存储，为避免跳跃访问每行元素，可以

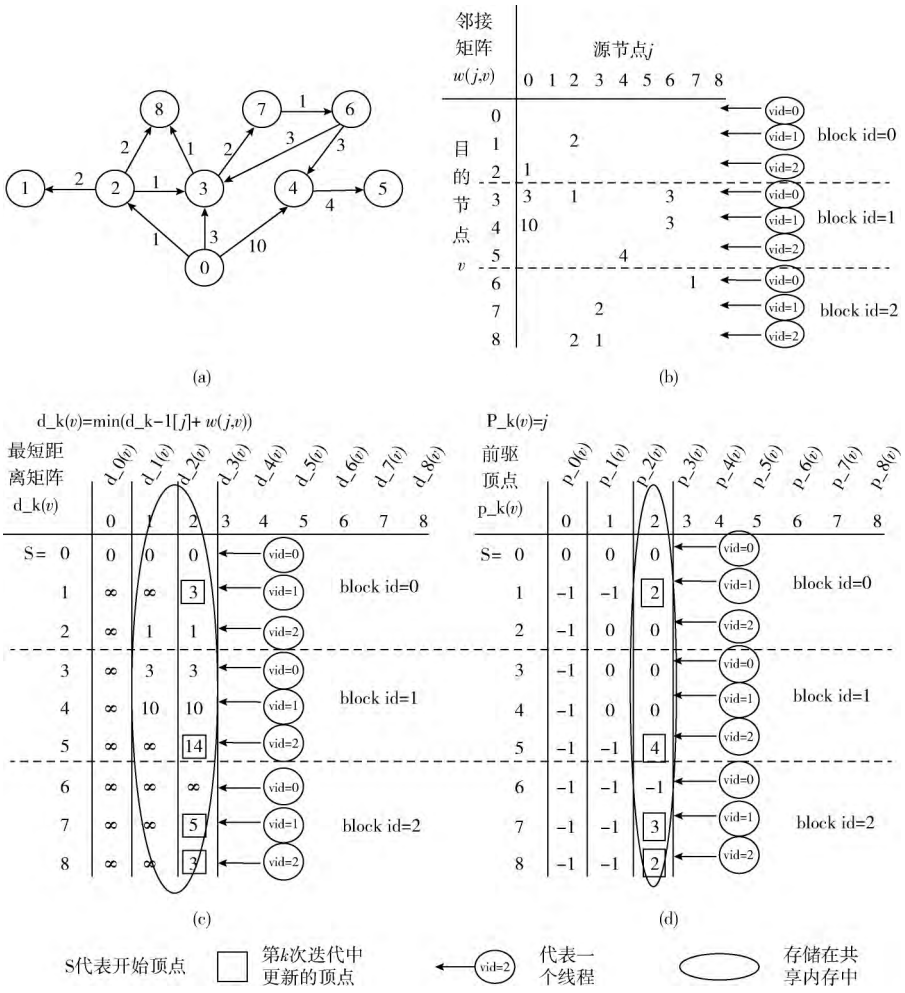


图 3 访存优化分析

考虑在 CPU 写入 GPU 之前将矩阵转置；②从访问存储次数上分析：利用一维线程块将最短距离矩阵和前驱节点向量读入到共享内存中计算，减少了对全局内存访问次数，为之后的分流设计方案做了铺垫；③从存储空间上分析：因共享内存容量受到硬件的限制，当节点较多时应考虑每个线程块线程数量上限条件，以充分利用存储器和计算能力

$$Sizeof(\text{int}) * (|V|+2|T|) \leqslant SMC \tag{1}$$

其中，SMC 为共享内存容量缩写。

当节点数较少且满足上述公式时，可以考虑将图 3（b）邻接矩阵横向分组读入到共享内存中，进一步减少访问全局内存次数。下面给出此种情况下应满足的线程上限公式

$$Sizeof(\text{int}) * (|V||T|+|V|+2|T|) \leqslant SMC \tag{2}$$

2.3 基于固定序优化线程发散

引入固定序是为了避免各线程遍历全部节点的问题。具体来说，对各节点松弛过程中，仅针对在上一次迭代中最短距离标量 $d\_k-1(v)$ 发生改变的点参与步骤二的松弛计算，因为在前一次迭代后并不是所有节点的 $d\_k-1(v)$ 都发生更新，若依然按照节点个数创建 $|V|$ 个线程的设计思路，会存在较多的闲置线程，使 warp 中的线程存在执行分散的情况，极大降低各线程块中线程的执行效率。因此，在第 k 次迭代前对距离标量改变的点集中化，降低复制到共享内存中 $d\_k-1(v)$ 向量元素数量。计算过程：首先，针对最短距离矩阵 $d\_k(v)$ 设计数据表；其次，对距离标量发生改变的点排序，使这些点更集中；最后，创建线程数目等于上一次迭代后更新的距离标量数目，以降低创建线程数量，避免线程执行分支的出现，从而提高并行度。

算法详细说明：算法 1 第（6）行，每个节点对应一个线程 $t(v)$，并行计算出各节点的最短距离矩阵 $d\_k(v)$ 和前驱节点向量 $p\_k(v)$，第（9）～第（13）行每个线程 $t(v)$ 遍历所有节点，为了边获得各边权重和各节点在上一次迭代后的 $d\_k-1(v)$ 作为计算参数，需经 $|V|-1$ 次比较选出最小的路径距离作为第 k 次迭代的最终结果 $d\_k(v)$。可见，若对上一次迭代后距离标量发生更新的点进行集中管理，那么在下一次迭代中只需创建数量较少的线程，再由线程块内并行规约比较设计方案（本小节介绍），即可并行

化地计算出各节点权重最小的 d_k(v)，达到收敛快的目的。如图 4 所示。

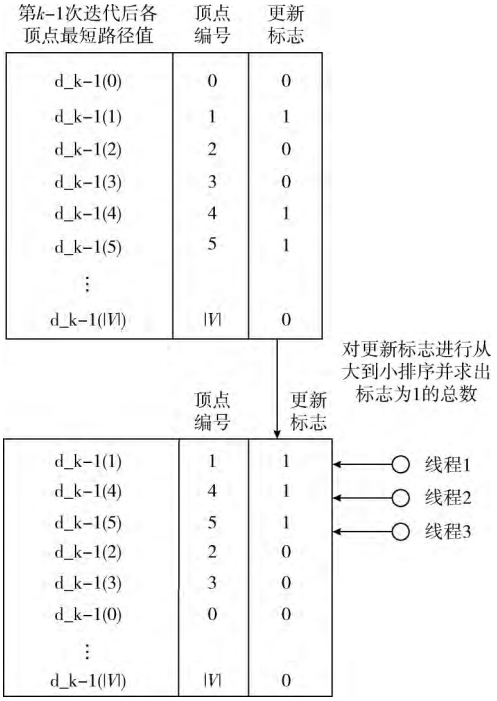


图 4　优化线程发散设计

列举了节点 1、4、5、在上一次迭代中最短距离 d_k−1(1)、d_k−1(4)、d_k−1(5) 发生过更新，在接下来的第 k 次迭代中计算每个点 v 的最短距离时，创建线程数量由改进前算法的一维线程 $T(|v|)=\{t_0,t_1,\cdots,t_{|v|-1}\}$ 减少到 $T(3)=\{t_0,t_1,t_2\}$ 的 3 个线程数量。其设计仅在第 k 次（$k=0,1,\cdots,|v|-1$）迭代前加入排序操作，通过增加一次排序操作减少了迭代过程中创建线程的数量，同时降低各线程块中线程分散的程度，充分利用 GPU 的计算能力。

并行规约比较设计是采用折半比较方法在每次迭代中针对各节点求最短距离 d_k(v) 问题提出来的。具体来说：首先，经固定序排序后距离标量发生改变的节点个数是 m。考虑到相同线程块内线程调度的基本单位是 $warp$[11,12]，为充分利用 GPU 处理核心，创建线程数目 M 可满足下列公式

$$M=\left|\left(m+(warp-1)\right)/warp\right|*warp \tag{3}$$

其中，$M\in$（1，线程块线程最大上限）。

其次，采用折半比较方式，用左半边线程去比较右半边，选较小的值更新至左半边，该过程是迭代过程，下次迭代中选左侧半边/2 的线程数目继续折半比较，迭代 $\lceil\log_2 M\rceil+1$ 次结束，直到剩一个线程即最小值。如图 5 所示，给出了 8 个节点的折半比较过程。

可见对 d_k 结果按照更新标志位固定序之后，可减少

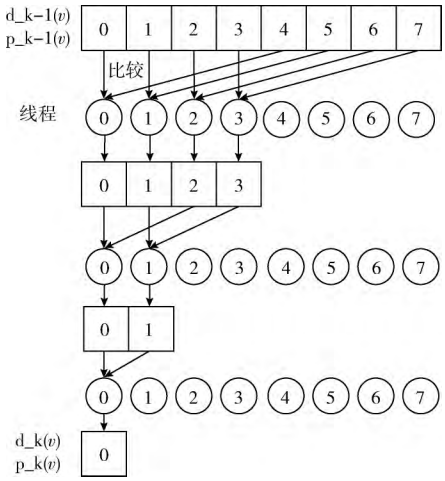


图 5　并行规约比较迭代

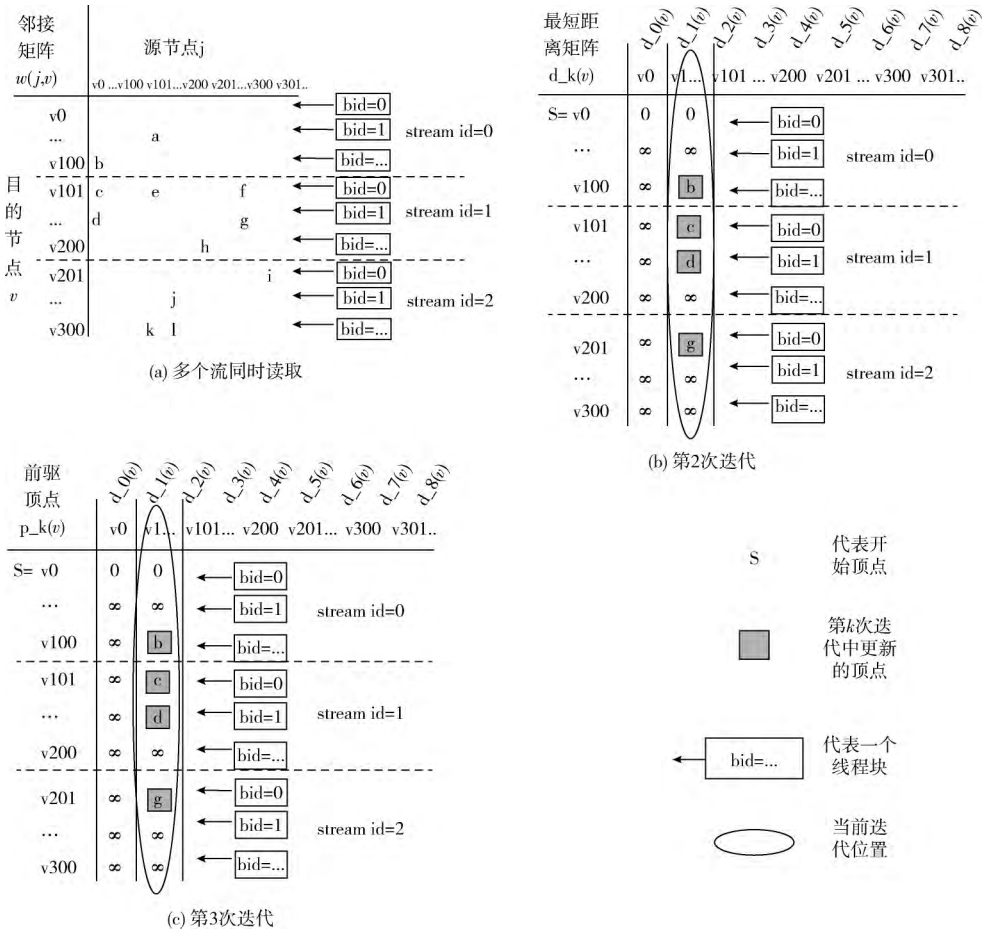
并行规约迭代的输入数据规模，有利于折半比较更快地收敛。

2.4　基于 CUDA 流优化数据传输开销

分流方案主要解决在第一次迭代计算中主机与 GPU 数据传输占用时间久和首次计算延时长的问题。具体来说：①此算法数据传输耗时最久的阶段是主机第一次将邻接矩阵传送到 GPU 全局内存的阶段。因为邻接矩阵是二维表，随着节点数量逐渐增加，矩阵尺度也会逐渐增加，而整个程序的计算均离不开读取邻接矩阵，所以，将邻接矩阵传输到 GPU 全局内存的速度决定了第一次迭代计算的时延，若采用分流方式传送，各流之间是异步向 GPU 传输邻接矩阵的部分分组，各分组传送期间 GPU 也参与计算 d_1(v)，解决了第一次迭代计算延时久的问题；②在以后的迭代中，因邻接矩阵已经被传入到 GPU 全局内存中，无需再次传入，故不再采用分流方式，而使用多个线程块的方式直接从全局内存中获取邻接矩阵部分分组。具体来说：在每次迭代完成后需将更新标志和当前迭代完成标识反馈给主机，由主机判断是否达到迭代上限，以决定是否需要停止计算。例如，将第 k 次迭代（$k=2,3,\cdots,|v|$）划分为不同线程块，在逻辑上等同于将邻接矩阵横向划分成多组，每个线程块仅仅负责计算一部分 d_k(v) 和 p_k(v)，每次迭代完毕后生成的 d_k(v) 和 p_k(v) 向量再写回全局内存，以覆盖 d_k−1(v) 和 p_k−1(v)，达到节省存储空间的目的。

如图 6（a）示例中横向虚线将邻接矩阵 $w(i,j)$ 划分为不同分组，在第一次迭代开始时，各个工作流是异步方式传送和计算数据的，这样设计的好处是将庞大的数据划分为各个分组，使等待数据时延限制在各个工作流中。如图 6 所示，列举了第一次迭代使用 CUDA 工作流的方式。

结合 2.2、2.3、2.4 提出的优化方案，最后给出并行算法如下：



(a) 多个流同时读取

(b) 第2次迭代

(c) 第3次迭代

图 6 第一次迭代中各工作流异步工作

算法 2：基于固定序的 Bellman-Ford 并行算法

```
//初始化节点最短路径值和前驱节点模块
(1) __global__ void kernel_Initial(G,d,p){
(2)  d_0[vid]=+∞;  p_0[vid]=-1;
(3)  if(vid==0) d_0(0)=0;
(4)}//固定序模块
(5) __global__ void kernel_fixed_order(G,d,p){
(6)  merge_sort_gpu<<<1,1024>>>(G,d);//按照更新标志位排序
(7)}//第 k 次迭代计算最短距离和前驱节点模块
(8) __global__ void kernel_Iter(G,w,d,p){
(9)  __shared__int sw_iv[G.v];//共享内存存储权代价
(10)  __shared__int sd_k-1[G.m];//存储第 k-1 次迭代结果
(11)  __shared__int sp_k-1[blockDim.x];//存储第 k-1 次迭代前驱节点和第 k 次迭代完成后的输出结果
(12)  __shared__int sd_k[blockDim.x];//存储第 k 次迭代后最短距离
(13)  int idx=threadIdx.x;//线程 id
(14)  int x = threadIdx.x + blockIdx.x * blockDim.x;//全局坐标
(15)  copy_data();//把数据从全局内存复制到共享内存中
(16)  kernel_Comparison();//并行规约比较求 d_k(v) 和 p_k(v),同时更新标志位
(17)}
(18)void Stream_cal(){
(19) cudaStream_t stream[分流数];
(20) for(int i=0;i<分流数;i++){
(21) kernel_Iter<<<(G.v-1+warp)/warp,warp,stream[i]>>>(G,w,d,p);
(22) }
(23)cudaStreamDynchronize(stream); //流同步
(24)}
(25)void cpu(G,w,d,p){
(26) kernel_Initial<<<(G.v-1+warp)/warp,warp>>>(G,d,p);//初始化//进行迭代进行 G.v-1 次
(27) for(k=1 k<=G.v-1;k++){
```

```
(28)  if(k==1) Stream_cal();//分流迭代计算
(29)  else kernel_Iter<<<(G.v-1+warp)/warp, warp>>>(G,w,d,p);
(30)  test();//并检测是否满足迭代条件
(31) }
(32)obtain();//获得结果
(33)}
```

3　实验分析

计算平台分为主机与支持 CUDA 的 GPU 显卡构成：主机系统配置为 Intel® Xeon® CPU E5-2620 v4 时钟频率 2.10 GHz，配置 32 GB RAM。GPU 规格见表 1。

表 1　NVIDIA GeForce GTX 1080 规格

规格	容量
CUDA 处理核	3840 个
全局内存容量	8 GB
Warp 大小	32 个
每个块共享内存总量	48 KB
每个块线程最多数目	1024 个

实验时采用随机方式分别生成具有代表的节点数量为 512、1024、2048、8192 的图。测试算法性能，并对本文提出的优化方案验证。

3.1　访存优化对计算性能影响测试

此部分测试随着节点规模的不同，分别选择式（1）、式（2）分配线程，对计算性能的影响。见表 2。

表 2　结合硬件容量线程数量 $T(x)$ 对比

理论线程数量	节点规模			
	512	1024	2048	8192
式(1)	5888	5632	5120	2048
式(2)	23	11	5	不符合
选用计算公式	式(2)	式(2)	式(2)	式(1)

经过计算可以看出，节点规模越少时，参照式（1）计算出一个线程块中可以创建的线程数量远超过了 CUDA 规定的线程上限（1024 个），如果选用式（1）方案分配 1024 个线程的话，不可避免增加了访问全局内存中邻接矩阵的次数。故对于节点数量较少的情况来说，式（2）作为创建线程数目的参照较为理想。随着节点数量增多，参照式（1）创建线程较为理想。如表 3 所示，其根据表 2 确定的线程规模测试存储优化对计算性能的影响。

从优化前后可以看出，当节点数量较少时，参照式（2）将全局内存中的邻接矩阵读入到共享内存中，使每个线程访问全局内存的频度有所减少，从而获得了性能上的提升；随着节点数量不断增加，采用式（2）创建的线程数量会减少，从 512 规模的 23 个线程上限减少为 2048 规模的 6 个线程上限，同时线程块数量会增多，此时影响程序执行时间的主要因素是线程块的数量过多；当节点数量为 2048 时，优化前后的结果相比照 512 和 1024 不是很明显，因为线程数量太少的原因；当节点数目增加为 8192 时，此刻式（2）不满足，使用式（1）（即每个线程读取全局内存中邻接矩阵）的方式获得了性能的提升。可见影响算法性能的因素，一方面与线程访问全局内存的频率有关；另一方面与线程块的数量有关。所以，需要结合节点规模选择合适的优化方案。

表 3　优化前后时间对比/ms

工作模式	节点规模			
	512	1024	2048	8192(不满足式(2)未优化)
访存优化前	4.64	13.89	29.07	156.61
访存优化后	2.72	8.26	36.43	159.08

3.2　固定序优化测试

固定序优化是针对缩减全局内存中第 $k-1$ 次迭代后向量维度的问题（针对 d_k−1(v)），以缩减第 k 次迭代读入到共享内存的数据量。测试情况见表 4。

表 4　优化前后时间对比/ms

工作模式	节点规模			
	512	1024	2048	8192
访存优化	2.72	8.26	36.43	159.08
固定序优化	2.85	4.58	16.82	112.62

当节点较少时，固定序优化不是很明显，随着节点数量的增加固定序优化带来了约 17%的性能提升。可以预见，当节点数量越小时，固定序优化不明显，甚至会出现比访存优化差的情况，但随着节点数量的不断增多，固定序优化相比访存优化较优。其原因如下：首先，固定序思想需要对距离标量发生改变的点进行排序，当节点数量较少时，距离标量改变的点相对不多，则排序会带来额外的时间付出；其次，当节点数量增多时，相对来说更新的距离标量会相对增多，通过集中距离标量改变的点，可以极大地减少下次迭代中参与计算节点的数量，节省了每次迭代的计算时间。可见，采用固定序优化设计方案，对大规模计算节点来说可稳步提升并行算法的计算效率。

3.3　基于 CUDA 流优化数据传输开销测试

此部分测试在初次迭代中使用 CUDA 工作流对 CPU-GPU 数据传输时间的影响，以解决首次计算时延长的问题，度量公式：$T = T_{(传送)} + T(计算)$。测试情况见表5。

表 5 初次迭代不同工作模式计算时延对比

工作模式	节点规模	平均传输时间比例/%	平均计算时间比例/%
使用 CUDA 工作流	512 1024	11.5	88.5
未使用 CUD 工作流	2048 8192	38.4	61.6

可见，初次使用 CUDA 工作流分组传送邻接矩阵，其平均计算时间相比未优化前平均计算时间明显增加，隐藏了 CPU-GPU 数据传输时间片，极大提升了算法执行效率，有效地解决了初始迭代中计算时延久的问题。从 GPU 计算时间上看，缩短了数据传输时间，等同于提升了平行算法性能。由此可见，此方法优化效果明显。

为了验证基于固定序思想的并行方案时效性，对比了固定序串行算法和基于固定序的并行算法，同时列出了改进前传统 Bellman-ford 并行算法以示参照（参考文献［6，10］），说明本文提出的优化方法适用于固定序设计方案。见表 6～表 8。

表 6 文献［6］固定序串行算法执行时间/ms

节点规模	运行时间
	GPU 运行时间
6000	184.9
8000	248.3
10 000	305.7

表 7 文献［10］并行算法与传统串行算法执行加速比

运行	节点规模					
	256	512	1024	2048	4096	8192
加速比(倍)	25	75	125	190	210	230

表 8 本文并行与改进前后串行算法执行时间/ms 与加速比

运行时间	节点规模			
	512	1024	2048	8192
固定序的 GPU 运行时间	2.85	4.58	16.82	112.62
固定序的 CPU 运行时间	3.7	7.8	38.2	480.8
固定序改进后加速比(倍)	0.27	2.17	2.27	4.26
改进前 CPU 运行时间	28.6	228.8	1304.5	32 356.9
对比改进前加速比(倍)	10.0	49.9	77.6	287.3

表 7 是改进前算法加速比，由于文献实验部分未给出串行算法和并行算法各自的执行时间，本文通过实验，计算出传统算法的执行时间，可参见表 8 中改进前 CPU 执行时间，可以看出加速比虽然比较高，但执行总时间也相对较长；虽然固定序并行算法比固定序串行算法加速比稳定在 5 倍以内，但是相比改进前算法来说，基于固定序思想在 CUDA 平台下得到了可行性验证，也获得了 280 倍的加速，验证了基于固定序的改进方案适合并行算法设计且优化方法适合此算法。

3.4 并行设计方案性能可移植测试

以上测试均在 GTX 1080 显卡上测试，为了测试并行方案的在不同 NVIDIA 显卡上的可移植性，选用 TITAN V 显卡进行实验，详细参数见表 9。

表 9 NVIDIA TITAN V 规格

规格	容量
CUDA 处理核	15 360 个
全局内存容量	12 GB
Warp 大小	32 个
每个块共享内存总量	48 KB
每个块线程最多数目	1024 个

从表 1 和表 9 可以看出，TITAN V 的 CUDA 处理核心数是 GTX 1080 的 4 倍，全局内存容量增大了 4 G。如表 10 所示，当节点规模较小时由于算法采用共享内存设计方案参与计算，影响算法计算效率的因素是 TITAN V 较多的 CUDA 处理核心。随着节点规模不断增大，算法未采用共享内存计算，相比照 GTX 1080 来说此时 TITAN V 依然凭借着较多的处理核心，获得了较好的性能提升。可见此并行设计方案在 NVIDIA 不同显卡上具有性能可移植的特性。

表 10 GTX 1080 与 TITAN V 执行时间对比/ms

运行时间	节点规模			
	512	1024	2048	8192
GTX 1080 GPU 运行时间	2.85	4.58	16.82	112.62
TITAN V CPU 运行时间	3.51	3.62	13.24	97.87

4 结束语

本文为验证固定序思想在 CUDA 下的可行性并设计优化方案优化其性能。从访存优化、基于固定序优化线程分散和基于 CUDA 流优化数据传输开销，深入分析计算瓶颈和数据传输开销：①参照共享内存容量分配线程；②固定序缩减了下次迭代参与计算的数据量；③CUDA 流利用时间片分块传输计算邻接矩阵各个分组，解决了初次迭代计算时延久。经实验，设计方案在 CUDA 平台的不同显卡上得到了验证，具有性能可移植的特点。相比改进后算法来说，加速比稳定在 5 倍以内。相对改进前算法来说，固定序设计方案具有更高的加速优势，同时本研究为多平台相关研究提供了参照。

参考文献：

[1] HAN Weiyi. An improvement of fixed-order Bellman-Ford algorithm [J]. Journal of Harbin Institute of Technology, 2014, 46 (11): 58-62 (in Chinese). [韩伟一. 固定序 Bellman-Ford 算法的一个改进 [J]. 哈尔滨工业大学学报, 2014, 46 (11): 58-62.]

[2] Makariye N. Towards shortest path computation using Dijkstra algorithm [C] //International Conference on Iot and Application. IEEE, 2017: 1-3.

[3] KANG Wenxiong, XU Yaozhao. Hierarchical Dijkstra algorithm for node constrained shortest paths [J]. Journal of South China University of Technology (Natural Science Edition), 2017, 45 (1): 66-73 (in Chinese). [康文雄，许耀钊. 节点约束型最短路径的分层 Dijkstra 算法 [J]. 华南理工大学学报(自然科学版), 2017, 45 (1): 66-73.]

[4] Busato F, Bombieri N. An Efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures [J]. IEEE Transactions on Parallel & Distributed Systems, 2016, 27 (8): 2222-2233.

[5] Swathika OVG, Hemamalini S, Mishra S, et al. Shortest path identification in reconfigurable microgrid using hybrid Bellman Ford-Dijkstra's algorithm [J]. Advanced Science Letters, 2016, 22 (10): 2932-2935.

[6] HAN Weiyi. Improvement of Bellman-Ford algorithm based on fixed order [J]. Operations Research and Management Science, 2015 (4): 111-115 (in Chinese). [韩伟一. 基于固定序的 Bellman-Ford 算法的改进 [J]. 运筹与管理, 2015 (4): 111-115.]

[7] Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing [J]. ACM Sigarch Computer Architecture News, 2015, 43 (3): 105-117.

[8] LIU Lei, WANG Yanyan, SHEN Chun, et al. Performance portable GPU parallel optimization technique on Bellman-Ford algorithm [J]. Jilin Daxue Xuebao, 2015, 45 (5): 1559-1564 (in Chinese). [刘磊，王燕燕，申春，等. Bellman-Ford 算法性能可移植的 GPU 并行优化 [J]. 吉林大学学报(工), 2015, 45 (5): 1559-1564.]

[9] Grossman, Max. Professional CUDA C programming [M]. United States: Wrox Press, 2014: 14-22.

[10] Wang Z, Grewe D, O'Boyle MFP. Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems [J]. ACM Transactions on Architecture & Code Optimization, 2015, 11 (4): 1-26.

[11] Narasiman V, Shebanow M, Chang JL, et al. Improving GPU performance via large warps and two-level warp scheduling [C] //IEEE/ACM International Symposium on Microarchitecture. IEEE, 2017: 308-317.

[12] Rupnow K, Rupnow K, Rupnow K, et al. An accurate GPU performance model for effective control flow divergence optimization [J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2016, 35 (7): 1165-1178.

(上接第 2122 页)

[10] Liu J, Wang C, University SL. The improvement of weighted centroid localization algorithm based on RSSI [J]. Journal of Shenyang Ligong University, 2017, 36 (4): 11-13.

[11] Alomari A, Comeau F, Phillips W, et al. New path planning model for mobile anchor-assisted localization in wireless sensor networks [J]. Wireless Networks, 2017, 30 (8): 1-19.

[12] Mahfouz S, Mourad-Chehade F, Honeine P, et al. Target tracking using machine learning and kalman filter in wireless sensor networks [J]. Sensors Journal IEEE, 2014, 14 (10): 3715-3725.