云南大学学报(自然科学版), 2023, 45(5):1022~1032 DOI: 10.7540/j.ynu.20220170
Journal of Yunnan University: Natural Sciences Edition

基于 GPU 加速的全源对最短路径并行算法

肖 汉[1,3], 肖诗洋[2], 李焕勤[1**], 周清雷[3]
(1. 郑州师范学院 信息科学与技术学院, 河南 郑州 450044; 2. 东南大学 土木工程学院, 江苏 南京 211189;
3. 郑州大学 计算机与人工智能学院, 河南 郑州 450001)

摘要: 针对最短路径算法处理大规模数据集低效的问题, 提出了基于图形处理器(Graphics Processing Unit, GPU)加速的全源对最短路径并行算法. 首先通过优化矩阵乘法算法实现了在工作组内和组间进行并行运算数据, 然后减少了非规则行造成的工作项分支, 最后降低了工作项对邻接矩阵计算条带存储资源的访问延时. 实验结果表明, 与基于 AMD Ryzen5 1600X CPU 的串行算法、基于开放多处理(Open Multi-Processing, OpenMP)并行算法和基于统一计算设备架构(Compute Unified Device Architecture, CUDA)并行算法相比, 最短路径并行算法在开放式计算语言(Open Computing Language, OpenCL)架构下 NVIDIA GeForce GTX 1 070 计算平台上分别获得了 196.35、36.76 和 2.25 倍的加速比, 验证了提出的并行优化方法的有效性和性能可移植性.

关键词: 最短路径; 重复平方法; 图形处理器; 开放式计算语言; 并行算法
中图分类号: TP311 **文献标志码:** A **文章编号:** 0258−7971(2023)05−1022−11

图分析算法在许多领域具有重要应用价值, 全源对最短路径算法是一种重要的图最短路径算法, 在生物网络比对和路径推断、交通运输、运筹学、机器人动作规划、网络通信等领域有广泛的应用. 但是, 随着最短路径问题应用领域的数据规模迅速增长, 数据结构日益复杂, 数据处理时间过长, 限制了其在实际中的应用[1]. 同时, 全源对最短路径处理算法的计算量大, 如何提高最短路径处理速度是当前图算法处理领域的研究热点之一[2].

当前, 图形处理器(Graphics Processing Unit, GPU)的浮点运算处理能力是 CPU 的 10 倍以上, 带宽是 CPU 的 5 倍, 但成本和能耗远低于 CPU, 在许多方面具有很大的优势[3]. 架构方面, GPU 已经采用了统一计算架构单元, 并实现了线程间通信和细粒度的任务分配, 在通用计算领域 GPU 已经显示出了强大的竞争力. 统一计算设备架构(Compute Unified Device Architecture, CUDA)是一种非开放标准, 仅适用于在 NVIDIA 的 GPU 硬件架构上运行, 从而阻碍了 CUDA 技术的推广. 然而, 随着基于 GPU 的通用计算(General-Purpose computing on GPU, GPGPU)被人们广泛接受, 业界已经针对 GPGPU 制定了如开放式计算语言(Open Computing Language, OpenCL)的开放标准. 设计适用在各种 GPU 上且能够高效执行的基于 OpenCL 的最短路径并行算法是一项有挑战性且非常有意义的工作[4-5].

本文利用 OpenCL 架构实现了一种高效的基于 CPU+GPU 的全源对最短路径并行算法(OCL_SP), 解决了全源对最短路径算法在不同异构计算设备上快速处理大规模顶点集的问题. OCL_SP 并行算法在各种设备上表现出良好的可移植性.

1 相关研究

最短路径算法是许多复杂图算法的一个重要组成部分, 普遍应用于科学研究和工程分析中. Jamour 等[6]将图表分解为双连通组件, 图形中的所有点对最短路径的计算性能提高 3.7 倍. Zhang 等[7]采用方向共享、并行请求和路由点优化减少外部路由请求的数量, 提出了一种能在外部路由请求中找到最佳路点集的贪心算法. Chandio 等[8]通

收稿日期: 2022-04-19; 接受日期: 2022-07-24; 网络出版日期: 2022-09-24
基金项目: 国家自然科学基金(61572444); 河南省高等学校重点科研项目(22A520049).
作者简介: 肖 汉(1970−), 男, 湖北人, 教授, 主要研究大规模并行算法研究与设计、遥感大数据并行处理. E-mail: xiaohan70@163.com.
** 通信作者: 李焕勤(1976−), 女, 河南人, 副教授, 主要研究软件工程、数字图像处理. E-mail: zzsz2005@126.com.

过遵循批量同步并行参数预先计算路段的最短路径距离和速度约束, 提出了一种基于云环境的实时 GPS 轨迹的完全自适应地图匹配策略. Panomruttanarug[9] 利用经典的路径规划方法找到最短的停车路径, 并利用经验学习的能力跟踪设计路径. Hung 等[10] 开发了一种分布式、并行且可扩展的最短路径算法消除网络中的冗余间接边缘. Gyongyosi [11] 在量子中继器网络中以群体智能为基础, 使用并行线程执行路由以通过纠缠梯度系数确定最短路径. 闫春望等[12] 提出了基于自动波理论的并行模糊神经网络最短路径算法, 其迭代次数和收敛速度得到优化. 李平等[13] 采用双线程进行子网分割和拼接汇总得到最短路径, 获得了 2.07 倍加速比.

Stpiczynski 等[14] 通过基于 Cilk 数组符号和内联函数的手动矢量化技术, 实现了 Belman-Ford 单源最短路径并行算法. Cabrera 等[15] 提出了以稀疏矩阵−向量乘法的迭代为基础的单源最短路径算法, 并行 DBMS 系统获得了 3.8 倍加速比. Maleki 等[16] 采用具备 16 个核心节点的共享和分布式存储器系统, 设计了一种带状放松式 Dijkstra 最短路径并行算法, 取得了 1.66 倍加速比. Chakaravarthy 等[17] 采用边缘分类和方向优化, 并提出负载平衡策略处理更高度的顶点, 实现了基于 CPU 集群平台的单源最短路径并行算法. 孙文彬等[18] 采用多粒度通讯的策略, 实现了基于 MPI 的 Dijkstra 最短路径并行算法.

Liu 等[19] 通过解决在稀疏矩阵−矩阵乘法中 3 个不规则性问题, 最短路径问题在 GPU 平台上获得了大约两倍加速比. 李寅等[20] 提出一种基于 GPU 架构的采样混合式全源对最短路径并行算法, 算法采用 BFS 遍历方法, 能达到 7.2 倍的加速比. 叶颖诗等[21] 设计实现了一种并行多标号 Dijkstra 算法, 获得了 3.49 倍的加速比. Djidjev 等[22] 提出了一种基于 GPU 集群的 Floyd-Warshall 最短路径并行算法, 取得了一个数量级的性能提升.

Aridhi 等[23] 采用并行方式求解原始图中每个子图上的最短路径, 提出了一种基于 MapReduce 的解决道路网络中的最短路径问题. Gayathri 等[24] 采用结合传递闭包属性和 Dijkstra 单源最短路径算法中贪婪技术特征寻找到所有点对最短路径, 在 MapReduce 上设计了 ex-FTCD 算法. 何亚茹等[25] 在基于神威平台的单个 SW26010 处理器上, 利用 MPI 技术求解全源最短路径的 Floyd 并行算法, 取得了 106 倍的最高加速比.

综上, 最短路径并行算法研究有些以新型理论为基础提出新的最短路径算法, 有些侧重对 Dijkstra 等传统算法本身进行优化, 有些基于 CPU 集群计算、CUDA 和 GPU 集群等并行计算方式实现最短路径并行算法, 有些则以大数据 Hadoop 平台为基础设计最短路径高性能算法. 可是, 全源对最短路径问题采用多种算法多类平台系统进行性能评估报道目前较少. 多数研究的加速效果不显著且均受限于一种算法或平台. 本文将根据 OpenCL 框架特征和最短路径问题的特点, 研究在 GPU 加速下的最短路径算法和在不同并行计算平台下的算法移植性能.

2　最短路径算法分析

2.1　GPU 加速处理通用计算　OpenCL 是一种开放的、免版税的标准, 用于对个人计算机、服务器、移动设备和嵌入式平台中的各种处理器进行跨平台、并行编程. OpenCL 定义了具有模型层次结构的计算体系结构: 平台模型、内存模型、执行模型和编程模型. OpenCL 平台由主机和连接到主机的计算设备组成[26]. 主机通常是 CPU, OpenCL 设备可以是支持 OpenCL 的设备, 如 CPU、GPU 和 FPGA. OpenCL 设备执行内核, 每个内核都是程序代码中的一个函数. 内核以工作项为单位执行, 工作项是并行执行的单位. 整个工作项集合称为一个 N-Dimensional Range (NDRange). 共享其结果的工作项被组织成工作组的一个单元[27].

支持 OpenCL 的计算设备由 OpenCL 计算架构中的主机端系统统一管理和调度. 当 kernel 被主机端提交到设备端时, OpenCL 为任务并行处理定义了索引空间 NDRange. 执行内核的各个工作项的标识由其在 NDRange 中的坐标表示. 多个工作项可组织为工作组, 提供了对 NDRange 更粗粒度的划分, 同一工作组中的多个工作项将在一个计算单元的不同处理单元上并发执行[28].

2.2　算法原理　矩阵乘法求最短路径算法的核心思想是通过多次迭代计算求得点对间的最短距离. 在该算法中是将原矩阵乘法中的“×”运算以“+”运算代替, “∑”运算以“min”运算代替[29]. 文中矩阵乘法运算内涵均按照此处说明进行. 而重复平方法是对矩阵乘法的变形应用, 重复平方法求最短路径是对每次计算得到的结果矩阵本身进行“×”运算[30].

对于一个n个顶点的加权有向图$G(V,E)$,其有权邻接矩阵由$\boldsymbol{W}_{n\times n}$表示. 假设$L_{ij}^{(m)}$是从顶点$i$到顶点$j$的最多包含$m$条边的任何一条路径的权值最小值. 递归定义$L_{ij}^{(m)}=\min(l_{ij}{}^{(m-1)},\min\limits_{1\leqslant k\leqslant n}\{l_{ik}{}^{(m-1)}+w_{kj}\})$,输入矩阵$\boldsymbol{W}=w_{ij}$,对于所有的顶点$i,j\in V$,$\boldsymbol{L}_{ij}^{(l)}=w_{ij}$,有$\boldsymbol{L}^{(1)}=\boldsymbol{W}$,最后矩阵$\boldsymbol{L}^{(n-1)}$得到的是实际的最短路径权值. 研究发现,当$m\geqslant n-1$时,有$\boldsymbol{L}^{(m)}=\boldsymbol{L}^{(n-1)}$. 经过推算,当$2^{\lceil\log(n-2)\rceil}\geqslant n-1$时,有$\boldsymbol{L}^{\left(2^{\lceil\log(n-2)\rceil}\right)}=\boldsymbol{L}^{(n-1)}$. 因此,只需要计算$\log(n-1)$次矩阵乘积就能得到$\boldsymbol{L}^{(n-1)}$[31].

在重复平方法最短路径算法处理的各个步骤中,建立有向图的带权邻接矩阵步骤、对邻接矩阵进行初始化步骤和建立邻接矩阵的镜像矩阵步骤的时间复杂度均为$O(n^2)$. 利用邻接矩阵计算有向图各顶点间的最短路径步骤的时间复杂度为$O\left(\sqrt{n^7}\right)$. 因此,本文主要关注顶点间的最短路径阶段的并行性及优化.

2.3 可并行性分析 存在于算法中的数据关联性与算法的可并行性好坏有关. 若数据运算先后关联性越强,算法可并行性越低. 反之,若数据运算先后关联性越弱,算法可并行性越高,并行化后算法的性能会越好. 对加权有向图的重复平方法最短路径算法的可并行性分析如下,如图 1 所示.

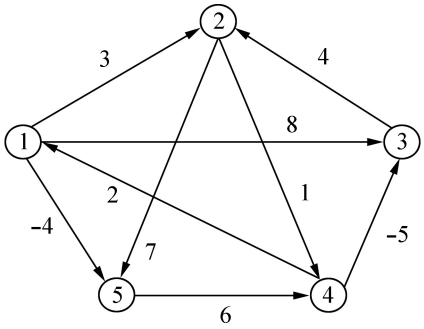


图 1 加权有向图
Fig. 1 Weighted digraph

图 1 中有权邻接矩阵$\boldsymbol{L}^{(1)}$如式 (1) 所示. 按照重复平方法计算,则有权邻接矩阵$\boldsymbol{L}^{(2)}$如式 (2) 所示("⊕"符号代表两个矩阵进行重复平方法运算). 在计算$\boldsymbol{L}^{(2)}$的过程中,发现$\boldsymbol{L}^{(2)}$中一个任意元素的计算和其余元素的计算过程相互无关,不存在依赖性. 即在$\boldsymbol{L}^{(2)}$中计算某个元素值时,可对其余元素值同时进行运算. 结合 OpenCL 模型的特性,在工作项中进行每个元素的计算,这样,$\boldsymbol{L}^{(2)}$中每个元素的结果将由工作项计算得出. 如果计算完矩阵中的每个元素,则本轮结束计算,如果需要迭代,则重复上面的过程.

$$\boldsymbol{L}^{(1)}=w=\begin{bmatrix}0&3&8&\infty&-4\\\infty&0&\infty&1&7\\\infty&4&0&\infty&\infty\\2&\infty&-5&0&\infty\\\infty&\infty&\infty&6&0\end{bmatrix},\tag{1}$$

$$\boldsymbol{L}^{(2)}=\boldsymbol{L}^{(1)}\oplus\boldsymbol{L}^{(1)}=\begin{bmatrix}0&1&-3&2&-4\\3&0&-4&1&-1\\7&4&0&5&3\\2&-1&-5&0&-2\\8&5&1&6&0\end{bmatrix}.\tag{2}$$

基于 CPU 的全源对最短路径串行算法(CPU_SP)是通过遍历整个邻接矩阵数据进行计算. 这样,CPU_SP 的时间复杂度为$O\left(\sqrt{n^7}\right)$. 现在利用具有众多工作项可进行并行计算的 GPU 完成矩阵输出数据的矩阵乘计算. 若u个工作项被系统启动,那么 OCL_SP 并行算法的时间复杂度降为$O\left(\sqrt{n^7}/u\right)$.

3 最短路径并行算法的设计和优化

3.1 并行算法描述 设有相同邻接矩阵$\boldsymbol{A}$、$\boldsymbol{B}$,邻接矩阵$\boldsymbol{A}$、$\boldsymbol{B}$中的子矩阵分别存储于数组A_s和数组B_s中,每次计算之后的最小值由v保存,矩阵$\boldsymbol{C}$存放的是完成计算后点对间最短路径长度数据. 重复平方法最短路径并行算法描述见算法 1.

算法 1 SIMT 模型上的重复平方法最短路径并行算法

输入 $\boldsymbol{A}(0,j,k)=w_{jk},0\leqslant j,k\leqslant n-1$.

输出 $\boldsymbol{C}(0,j,k)$中是v_j到v_k的最短路径长度,$0\leqslant j,k\leqslant n-1$.

```
begin
  (1) for j = 0 to n − 1 par − do
        for k = 0 to n − 1 par − do
          (1.1) if j ≠ k and A(0, j, k) = 0 then
                  A(0, j, k) ← ∞
                end if
          (1.2) B(0, j, k) ← A(0, j, k)
        end for
      end for
  (2) for i = 1 to ⌈log(n − 1)⌉ do
    (2.1) for a = a1 to a2, b = b1 to b2 do
            for k = 0 to B_size par − do
              if (value > A_s(ty, k) + B_s(k, tx))
                value = A_s(ty, k) + B_s(k, tx)
              end if
            end for
```

end for

$\boldsymbol{C}[c+wB\times ty+tx]=v$

(2.2) **for** $j=0$ **to** $n-1$ **par – do**

for $k=0$ **to** $n-1$ **par – do**

$\boldsymbol{A}(0,j,k)\leftarrow \boldsymbol{C}(0,j,k)$,

$\boldsymbol{B}(0,j,k)\leftarrow \boldsymbol{C}(0,j,k)$

end for

end for

end for

end

矩阵乘法是重复平方法最短路径并行算法的核心运算. 每一个工作组负责矩阵$\boldsymbol{C}$中v的计算, 每一个工作项计算比较得出v的最小元素值. 每个工作组中的工作项被划分成为$B_{\text{size}}\times B_{\text{size}}$的正方形计算工作项, 与正方形计算工作项对应的矩阵$\boldsymbol{A}$、$\boldsymbol{B}$中的数据就是需要进行计算的源数据. 在每次进行运算前, 将源数据存入本地存储器A_{s}和B_{s}中. 在每个工作组内进行计算时, 工作组 ID 不变, 工作项 ID 也不变, 每循环一次需要更新一次本地数组A_{s}和B_{s}中存放的数据. 当A_{s}和B_{s}中数据更新完毕, 对于A_{s}中的每一行元素、B_{s}中的每一列元素, 由每个工作项负责数组A_{s}和数组B_{s}中对应元素的求和、取最小值运算, 并将取得的最小值存入v中. 当该正方形中的元素全部读取计算完毕, 则读取该计算条带中的下一个正方形区域源数据, 直到每个工作组中的元素全部读取计算结束. 工作组中的正方形计算区域每次移动的步长为B_{size}. 当工作组中所有元素读取计算完毕时, v中保存的就是这些矩阵中对应元素和的最小值. 直到所有工作组中的所有线程全部计算结束, 把最终得到的结果再从本地存储器写回全局存储器.

3.2　算法并行化思路　OCL_SP 算法具体执行步骤如下.

步骤 1　根据顶点数, 在主机端初始化邻接矩阵$\boldsymbol{A}$并保存.

步骤 2　初始化 OpenCL 平台.

步骤 3　命令序列对象通过上下文及指定设备创建. 在设备和上下文之间通过 clCreateCommandQueue 建立逻辑连接, 以协调内核计算.

步骤 4　输入源文件并创建、编译程序对象. 依据在上下文中的设备特性, 通过运行时编译构建程序对象, OCL_SP 并行算法具备了较强的可移植性.

步骤 5　设置 OpenCL 内存对象和传输数据. 首先创建 buffer 存储器对象, 接着将存储器的访问任务加载到命令队列中, 最后利用 clCreateBuffer 将邻接矩阵$\boldsymbol{A}$、$\boldsymbol{B}$隐式的从主机写入到设备的全局内存中.

步骤 6　构建 kernel 对象. 利用 clCreateKernel 将内核函数与参数封装到指定的 kernel 对象中.

步骤 7　为 kernel 对象设置参数传递.

步骤 8　构建顶点数目满足的条件, 创建内核函数, 调度内核执行.

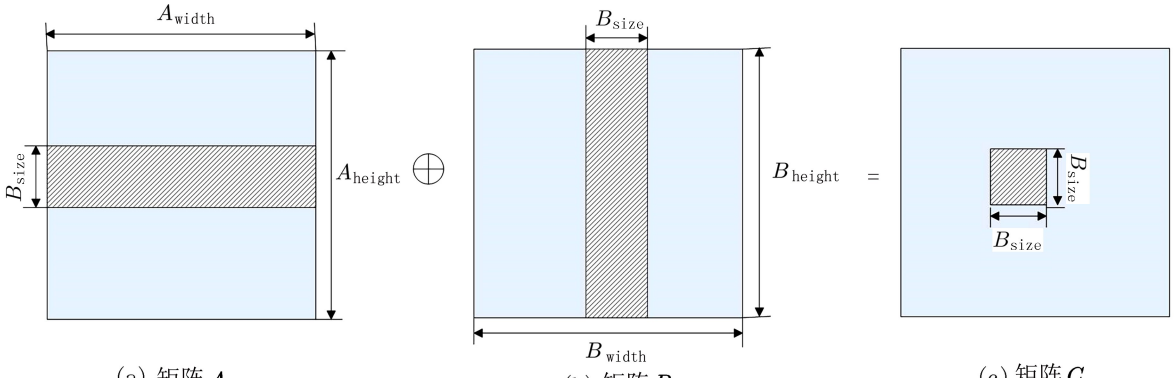
步骤 9　循环调用内核函数对数据的处理, 计算点对间的最短路径距离值.

步骤 10　在设备端中完成运算任务后, 将结果传输到主机端存储器, 释放设备端内存空间; 并将最终计算结果保存至相应文件.

3.3　并行算法的加速策略

3.3.1　理论设计　设计重复平方法最短路径并行算法时, 矩阵乘法采用了混合划分法实现. 混合划分法由带状划分法和棋盘划分法组成, 在工作空间 NDRange 中采取的是带状划分法, 在每个工作组内采取的是棋盘划分法.

工作空间中的带状划分法如图 2 所示, 其中一

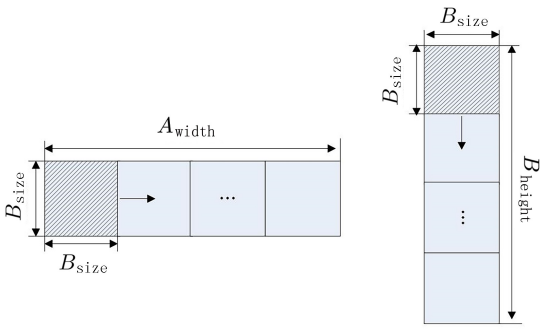


(a) 矩阵 $\boldsymbol{A}$　(b) 矩阵 $\boldsymbol{B}$　(c) 矩阵 $\boldsymbol{C}$

图 2　NDRange 带状划分图

Fig. 2　NDRange banded partition diagram

个工作组对应矩阵$\boldsymbol{A}$和矩阵$\boldsymbol{B}$中一对计算条带的运算，运算完成后的结果放在矩阵$\boldsymbol{C}$的对应位置上. 在矩阵$\boldsymbol{A}$中阴影区域计算条带的高度为B_{size}，矩阵$\boldsymbol{B}$中阴影区域计算条带的宽度为B_{size}. 在每个计算条带中，再对数据元素进行棋盘式的划分. 每次计算时，块坐标不变，但是每次读取数据的步长按照B_{size}增加，以更新将要计算的数据. 计算条带内划分方法如图 3 所示.



(a) 矩阵 $\boldsymbol{A}$ 中条带　　(b) 矩阵 $\boldsymbol{B}$ 中条带

图 3　工作组内棋盘划分图

Fig. 3　Chessboard division diagram in the working group

在进行矩阵乘法运算时，工作组每次加载一个计算条带. 由于计算条带宽度为B_{size}，所以每次进行迭代的步长是B_{size}. 根据带状划分的方法，矩阵$\boldsymbol{A}$从全局存储器完全被加载到本地存储器，需要被工作组内的本地存储器读取$B_{\text{width}}/B_{\text{size}}$次. 而和矩阵$\boldsymbol{A}$进行矩阵乘法运算的矩阵$\boldsymbol{B}$需要被工作组内的本地存储器读取的次数是$A_{\text{height}}/B_{\text{size}}$次，这样的数据读取方式大量减少了直接从全局存储器读取数据的延迟时间. 工作组内按照棋盘划分法设计矩阵乘法. 每个工作项负责读取矩阵$\boldsymbol{A}$中的一行元素和矩阵$\boldsymbol{B}$中的一列元素，计算得出矩阵$\boldsymbol{C}$中对应的元素值，并储存在全局存储器中.

对于每一个工作组，都需要由$\boldsymbol{A}$的一行元素和$\boldsymbol{B}$的一列元素进行比较计算之后才能得到最终结果，如图 4 所示. 矩阵$\boldsymbol{A}$中高度为B_{size}长度为A_{width}的矩形代表一个工作组，U和U'分别代表读取到本地存储器A_{s}和B_{s}的正方形区域源数据. 每个工作组内的操作过程如下：

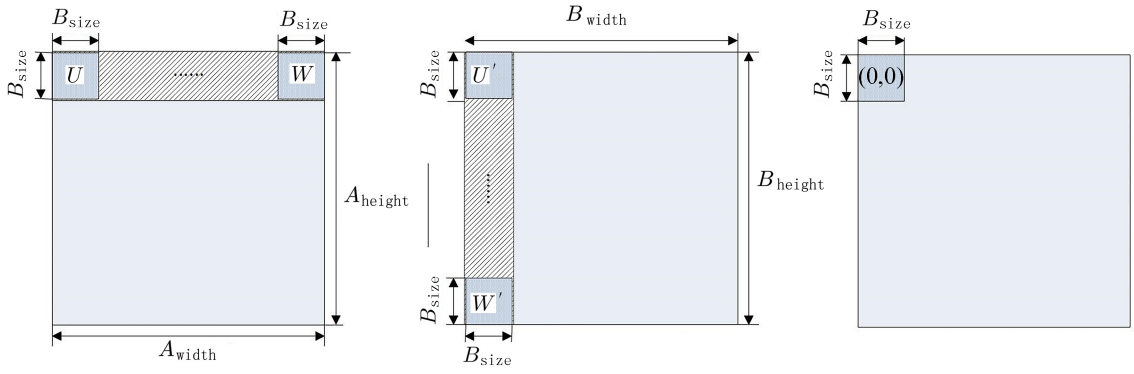
步骤 1　把U和U'从全局存储器中读出，放入本地存储器A_{s}和B_{s}中. 在边长为B_{size}的正方形计算区域中，一个工作项对应加载一对元素，然后利用矩阵乘法计算比较得出最小的元素值，并存入v.

步骤 2　进行 for 循环，以边长B_{size}为步长进行迭代，更新A_{s}和B_{s}中保存的数据. 重复步骤 1 直到读取完该计算条带A的每行元素和相应计算条带B的每列元素.

步骤 3　在最后一次内层 for 循环中，W和W'被放入A_{s}和B_{s}中，在最后一轮比较运算中所有工作项将所有计算任务完成后，每个工作项计算比较得出的最终结果保存在v. 相应在图 4(c) 中，计算得出了图中的块标号为(0,0)的块中的数据.

步骤 4　对于矩阵$\boldsymbol{C}$中的每个小块，都进行上述相同的操作，直到全部计算结束，矩阵$\boldsymbol{C}$中保存的就是该轮计算得出的结果.

3.3.2　kernel 函数的配置设计　整个邻接矩阵$\boldsymbol{A}$依据互不重叠的划分原则，可以被分割为若干个计算条带. 计算条带作为基本处理单位，存在于计算条带中的大量矩阵乘法计算可交给 OpenCL 工作组和工作项处理. 文中采用二维索引空间来设计，从数据角度来讲，二维索引空间在x、y方向上的维度均为 D，工作组在x、y方向上的维度均为B_{size}，工作空间在方向x、y方向上工作项总量均为 base$\times B_{\text{size}}$. 因此，内核函数的网格配置如下：



(a) 矩阵 $\boldsymbol{A}$　　(b) 矩阵 $\boldsymbol{B}$　　(c) 矩阵 $\boldsymbol{C}$

图 4　矩阵乘法的分块计算

Fig. 4　Block calculation of matrix multiplication

$$D = \left\lfloor (\text{POINT\_NUMBER} + B_{\text{size}} - 1) / B_{\text{size}} \right\rfloor$$

size_t szLocalWorkSize[] = $\{B_{\text{size}}, B_{\text{size}}\}$

size_t szGlobalWorkSize[] = $\{D \times B_{\text{size}}, D \times B_{\text{size}}\}$

$$\text{clEnqueueNDRangeKernel}\left(\begin{array}{c}\text{cqCommandQueue},\\ \text{ckKernel}, 2, \text{NULL},\\ \text{szGlobalWorkSize},\\ \text{szLocalWorkSize}, 0,\\ \text{NULL},\\ \&\text{GPUExecution}\end{array}\right)$$

3.4　优化设计

3.4.1　本地内存的设计　GPU 计算性能常受制于存储器速度. 由于运算速度与显存带宽间的不平衡, OpenCL 定义了多层存储器体系结构. 文中依据算法特征将存储器的合理分配和优化运用于计算流程中的访存方式和数据传输. 在重复平方法最短路径算法的矩阵乘法运算中, 邻接矩阵$A_{n\times n}$、$B_{n\times n}$中的每一个数据都需要被重复读取n次. 如果将计算条带放在具有 400 ~ 800 个时钟周期读写延迟的全局存储器中, 将大大降低并行数据访问速率. 通过引入仅具有 1(无 bank 冲突) ~ 16 个时钟周期(发生 16 路 bank 冲突)读写延迟的本地存储器, 将需要大量重用的计算条带数据在计算前预置在本地内存A_s和B_s中.这样既能方便地在工作项间进行通信, 也使工作项的访存速度更快从而大量的全局存储器延迟被透明地隐藏起来.因此, 并行算法中采用静态分配方式在内核函数中对A_s和B_s分配本地存储器空间:

__local float$A_s[B_{\text{size}}][B_{\text{size}}]$

__local float$B_s[B_{\text{size}}][B_{\text{size}}]$,

同时, 在不同的本地存储器 bank 中存储了每条计算条带中的数据, 保证了在同一个 warp 中相邻的一组 32 个访问工作项与 bank 是一一对应, 避免了 bank 冲突, 有效提高了运算速度.

3.4.2　NDRange 优化　计算单元应该保持较高的占用率. 因为工作项指令在 OpenCL 中是顺序执行, 隐藏延迟的有效方法就是在一个 warp 繁忙时执行其他 warp. 若存在足够多的工作组数量, 就可避免因 warp 块填充不足导致 OpenCL 系统的计算资源浪费. 一个工作组中保持的工作项最好是 32 的倍数. 由于有向图的顶点数是变化的, 工作组和工作组中的工作项数应该根据所占用的寄存器空间及本地存储器空间动态分配, 以达到最佳效率. 针对 OpenCL 加速的全源对最短路径图算法, 当有向图顶点数为 5 000 时, 不同工作组维度下, 算法运算时间比较如表 1 所示. 从表 1 中可见, 当工作组维度为 16×16 时, 全源对最短路径并行算法的运算时间最短.

表 1　算法运算时间比较

Tab. 1　Compare the operation times of algorithm

工作项数	运行时间/s
4×4	0.96
8×8	0.94
16×16	0.85
32×32	0.91

3.5　其他优化方案　为了便于各种并行化方案的性能比较, 本文分别在 OpenMP 和 CUDA 并行计算体系结构上实现了一种符合 2.2 节最短路径算法原理的并行算法.

3.5.1　基于 OpenMP 的最短路径并行算法(OMP_SP)　在 Windows 系统中, 使用 OpenMP 内置的 Microsoft Visual Studio.Net 2017 编译器, 在项目的 Microsoft.Cpp.x64.user 属性页中设置'C/C++→Language→OpenMP Support'为"是 (/OpenMP)", 在 Visual Studio.net 2017 中打开 OpenMP 编译选项. 然后将选项"C/C++→Gode Generation→Runtime Library"更改为"多线程调试 (/mtd)".

OpenMP 是一种粗粒度并行. 根据 CPU 核心的数量, 并行块通过 omp_set_num_thread (thread) 启用子线程进行并行执行. 主线程执行 OMP_SP 并行计算系统, 当遇到位于构造初始邻接矩阵$\boldsymbol{L}^{(1)}$、保存$\boldsymbol{L}^{(1)}$到$\boldsymbol{A}$和$\boldsymbol{B}$、矩阵乘法等函数之前的编译引导指令#pragma omp for 时, 将对后面的循环体进行多线程分解. 每个线程并行处理结果矩阵的一个元素.

3.5.2　基于 CUDA 的最短路径并行算法(CUDA_SP)　CUDA 是一个细粒度的平行. 将最短路径算法可并行的矩阵乘法功能部分划分为一个任务. 该任务由相应的 global 函数处理. 在算法运行过程中, global 函数与网格一一对应, 网格将任务分配给线程块, 线程块将任务重新分配给线程进行处理. 主机端调用 global 函数, 在 GPU 端的多个计算处理单元中进行并行处理.

4　实验结果及分析

4.1　测试环境和数据记录　选择单精度数据类型实现是因为它的准确性, 而且 GPU 针对现代计算

机在单精度浮点运算上进行了高度优化.

用 OpenCL C 语言开发了全源对最短路径并行算法的核心源代码. 该并行算法由两部分构成,在主机上执行主程序部分,另外内核程序部分是在 OpenCL 设备上执行. 核心代码实现了矩阵乘法运算.

实验工作是在 CPU/GPU 异构混合并行计算平台上进行. 硬件环境参数如下: 平台 1: CPU 为 AMD Ryzen5 1600X 3.6 GHz 六核; 系统内存容量为 24 GB; 显卡采用 NVIDIA Geforce GTX 1070 GPU. GTX 1070 拥有 1 920 个 CUDA 核,核心频率是 1 506 MHz. 搭配 8 GB GDDR5 显存容量,显存位宽 256 位. 平台 2: CPU 为 AMD Ryzen5 1600X 3.6 GHz 六核; 系统内存容量为 24 GB; 显卡采用 AMD Radeon RX 570 GPU. Radeon RX 570 拥有 2 048 个核心,核心频率是 1 168 MHz. 搭配 8 GB GDDR5 显存容量,显存位宽 256 位.

软件平台: 采用 Microsoft Windows 10 64 位为操作系统; Microsoft Visual Studio.Net 2017 为集成开发环境; CUDA Toolkit 10.0 为系统编译环境并支持 OpenCL 1.2 标准.

有向图的顶点集大小n分别设为 10、50、200、300、1 400、2 000、5 000、8 920 和 10 000,从小规模顶点集到大规模顶点集都均匀覆盖,有利于准确反映全源对最短路径并行算法的性能可扩展性. rand()随机数函数的随机数种子采用n可生成一组随机数,生成具有代表的节点数量为 10、50、200、300、1 400、2 000、5 000、8 920 和 10 000 的图,并构成相应的邻接矩阵 $\boldsymbol{A}$. 邻接矩阵是使用二维数组模拟线性代数中矩阵的结构. 此种数据结构所使用的储存空间较大,但其使用较为便捷. 根据重复平方法最短路径算法的原理,分别基于 OpenMP 和 CUDA 并行计算架构实现了最短路径并行算法.

测试并记录 CPU_SP 系统(平台 1 测试), OMP_SP 系统(平台 1 测试)、CUDA_SP 系统、基于 AMD GPU 的 OCL_SP 系统和基于 NVIDIA GPU 的 OCL_SP 系统的处理时间,如表 2 所示. 从表 2 可以看出,随着顶点数的增大,最短路径算法的执行时间越来越长. 基于 GPU 的最短路径算法运行时间要比 CPU_SP 和 OMP_SP 算法的运行时间少的多,而 OMP_SP 算法运行时间要比 CPU_SP 算法的运行时间更短.

用加速比作为加速效果的衡量标准,可以验证各种架构下并行算法的效率. 加速比定义如下:

$$S = \frac{T_{\mathrm{s}}}{T_{\mathrm{p}}}, \tag{3}$$

式中: T_{s}是指算法运行在单线程 CPU 平台上的时间, T_{p}是指算法运行在多线程 CPU 或 CPU+GPU 协同计算平台上的时间.

OMP_SP 并行算法执行时间T_{po}与基于 NVIDIA GPU 的 OCL_SP 并行算法执行时间T_{pnc}的比值为相对加速比 R_1,定义为:

$$R_1 = \frac{T_{\mathrm{po}}}{T_{\mathrm{pnc}}}. \tag{4}$$

CUDA_SP 并行算法执行时间T_{pc}与基于 NVIDIA GPU 的 OCL_SP 并行算法T_{pnc}执行时间的比值为相对加速比 R_2,定义为:

$$R_2 = \frac{T_{\mathrm{pc}}}{T_{\mathrm{pnc}}}, \tag{5}$$

$$T_{\mathrm{p}} = T_{\mathrm{k}} + T_{\mathrm{h}} + T_{\mathrm{o}}, \tag{6}$$

表 2 不同计算平台下最短路径算法执行时间

Tab. 2 The shortest path algorithm execution time under different computing platforms

有向图顶点数	CPU_SP/s	并行处理时间/s			
		OMP_SP	CUDA_SP	OCL_SP (AMD)	OCL_SP (NVIDIA)
10	0.09	0.08	0.09	0.05	0.04
50	0.14	0.12	0.07	0.07	0.07
200	0.27	0.22	0.25	0.13	0.12
300	1.44	0.91	0.28	0.25	0.24
1 400	7.57	2.88	0.32	0.31	0.30
2000	43.47	11.94	0.56	0.55	0.54
5 000	123.59	26.41	0.88	0.86	0.85
8 920	225.80	45.07	1.43	1.41	1.40
10 000	407.10	76.09	2.13	2.08	2.07

式中: 在 CPU 和 GPU 上 OpenCL 内核的计算时间之和为T_k, CPU 和 GPU 之间数据传输时间之和为T_h, 系统初始化时间之和为T_0.

加速比可用于评估并行算法相比 CPU 串行算法的性能提升. 相对加速比 R_1 可用于评估基于 NVIDIA GPU 的 OCL_SP 并行算法相比 OMP_SP 并行算法的性能提升. 相对加速比 R_2 可用于评估基于 NVIDIA GPU 的 OCL_SP 并行算法相比 CUDA_SP 并行算法的性能提升. 根据表 2 可得算法加速比情况, 如表 3 所示.

4.2　并行算法性能分析

4.2.1　系统性能瓶颈　OCL_SP 并行算法需要 $n\times n\times\log(n-1)\times 2$次右邻接矩阵的存储器读取操作, 以及$n\times n\times\log(n-1)$次输出矩阵的存储器写入操作. 顶点集大小为$n=10\,000$, 每个数据占用 2 B 的存储空间. 因此, 存储器访问的数据总量约为 6 GB. 除以设备上内核执行所用时间 0.028 s. 系统实际获得的带宽约为 214.29 Gb/s, 接近 GeForce GTX1070 显存的理论带宽. 由此可见, 全局存储器的带宽限制了 OCL_SP 并行算法效率的进一步提高. 因此, OCL_SP 并行算法的性能瓶颈是显示内存带宽.

4.2.2　不同架构下最短路径算法处理时间分析　根据表 2 显示, 对不同顶点数的加权有向图进行全源对最短路径处理将在 4 种计算平台上进行. 在顶点数一样时, 最短路径算法在不同并行计算架构下的处理时间相对串行处理时间均有不同程度缩减, 即获得了性能提升效果. 如: 对于顶点数为 10 000 的最短路径串行运算时间为 407.10 s, OMP_SP 算法运算时间缩短为 76.09 s, CUDA_SP 算法运算时间则大幅缩减为 2.13 s, 而基于 OpenCL 并行计算平台上则保持在 2.07 s 左右的低运算时间.

设n为加权有向图$G(V,E)$的顶点数, c 为多核 CPU 核心数, OMP_SP 并行算法的时间复杂度为 $O\left(\sqrt{n^7}/c\right)$. 在相同顶点数下, 存在 OMP_SP 并行算法的时间复杂度$O\left(\sqrt{n^7}/c\right)<O\left(\sqrt{n^7}\right)$. 因此, 从表 2 可以看出, OMP_SP 并行算法相比 CPU_SP 串行算法的处理时间有明显降低. 同时可以看出, 设置线程数 c 恒定时, 随着顶点数n的增大, OMP_SP 并行算法的时间复杂度也随之增加. 每一个 CPU 线程负责处理的邻接矩阵乘的数据规模也随之增大, OMP_SP 并行算法处理时间也越来越长, 基本符合线性增长的趋势.

当邻接矩阵的计算规模较小时, 在 GPU 算法中启动的工作项计算负荷不足, 系统在调度方面有较大时间开销. 此时 GPU 算法与 CPU 算法的处理时间接近, GPU 并行计算的优势没有展现. 随着邻接矩阵的计算规模增大, 每个工作项计算负荷也逐渐提高, 而系统调度的时间占比降低, GPU 并行计算占比上升, 速度提升明显.

4.2.3　并行计算架构下最短路径算法加速效果对比分析　设n为加权有向图$G(V,E)$的顶点数, u 为 GPU 启动的工作项的数据量, CUDA_SP 和 OCL_SP 并行算法的时间复杂度为$O\left(\sqrt{n^7}/u\right)$. 由于在 GPU 中可以维护大量的活动线程和工作项, 即$u\gg c$. 因此, 在相同顶点数下, 存在 CUDA_SP

表 3　不同计算平台下最短路径并行算法性能对比

Tab. 3　Performance comparison of parallel algorithm of shortest path on different computing platforms

有向图顶点数	加速比				相对加速比R_1	相对加速比R_2
	OpenMP	CUDA	OpenCL（AMD）	OpenCL（NVIDIA）		
10	1.13	1.00	1.80	2.25	2.00	2.25
50	1.17	2.00	2.00	2.00	1.71	1.00
200	1.23	1.08	2.12	2.32	1.83	2.08
300	1.58	5.14	5.68	5.87	3.79	1.17
1 400	2.63	23.66	24.65	25.12	9.60	1.07
2000	3.64	77.63	78.62	78.87	22.11	1.04
5 000	4.68	140.44	143.57	144.56	31.07	1.04
8 920	5.01	157.90	160.23	161.32	32.19	1.02
10 000	5.35	191.13	195.32	196.35	36.76	1.02

和 OCL_SP 并行算法的时间复杂度$O\left(\sqrt{n^7}\,|\,u\right) \ll O\left(\sqrt{n^7}/c\right)$. 从表 3 可以看出, 当有向图的顶点集合较小时, OMP_SP 并行算法与基于 GPU 的最短路径并行算法的加速效果相当. 分析其原因, OpenMP 的并行开销要小于 GPU. 由于 OMP_SP 并行算法是在 CPU 端执行, 不同于 GPU 并行算法需要在 CPU 与 GPU 之间进行数据传递. 而且 OpenMP 启动的线程少, 线程维护成本低. 当顶点集合逐步增大时, OMP_SP 并行算法的加速效果始终维持在一个较低的水平, 而 GPU 并行算法的加速效果更好. 这主要是由于 OMP_SP 并行算法需要操作系统将线程调度到对应数目的 CPU 核心上执行, 多核 CPU 的核心数是有限的. 反观计算资源充裕的 GPU 可以创建足够的工作项以满足大规模数据的并行处理. 同时, 随着顶点数n的增大, CUDA_SP 和 OCL_SP 并行算法的时间复杂度也随之增加, 每一个线程或工作项负责处理的邻接矩阵乘的数据量也随之增大. 从表 2 可以看出, CUDA_SP 和 OCL_SP 并行算法处理时间也适当的增长.

从表 3 可以看出, 当顶点集合规模较小时, GPU 加速的最短路径并行算法性能改善并不明显. 当顶点集合规模较大时, CUDA_SP 和 OCL_SP 并行算法的性能提升较为显著, 在 3 种不同并行计算架构下有效的性能收益明显. 当顶点集合n = 10 000时, 基于 NIVIDA GPU 的 OCL_SP 并行算法的加速比更达到了 196.35 倍. 然而, 当顶点集合规模$n > 5\,000$以后, GPU 加速的最短路径算法加速比曲线呈现一种缓慢上升的过程. 说明在顶点集规模较大的情况下, 系统已不能获得更快速的性能提高. 主要原因是 GPU 的计算资源接近饱和. 主机和设备间的数据传输时间占比越来越大, 并行计算时间的缩短已对加速比贡献甚微.

采用离线编译内核读写数据文件方式的 OCL_SP 并行算法, 相比采用在线编译内核读写数据文件方式的 CUDA_SP 并行算法减少了应用初始化时间. 如表 3 所示, 在同等数据集规模下, OCL_SP 并行算法的运算耗时更少, 与 CUDA_SP 并行算法性能相比略有提升, 最大获得了 2.25 倍加速比. OCL_SP 并行算法性能与 OMP_SP 并行算法性能相比则有很大的提高, 最大获得了 36.76 倍加速比.

将本文提出的 OCL_SP 算法的整体加速效果与文献 [25] 进行对比. 由于其他文献大多是利用最短路径算法进行各种应用研究, 很少专门针对全源对最短路径算法进行加速效果的研究. 因此, 无法直接进行加速效果的对比. 根据文献 [25] 中提供的测试数据, 当每个核组开启 64 个从核时, 经过数组划分优化后的全源最短路径并行算法获得了 106 倍加速比. 而根据表 3 的测试结果可以看到, OCL_SP 并行算法最高获得了 196.35 倍加速比. 因此, OCL_SP 并行算法相比文献 [25] 中的算法取得了更好的加速性能.

4.2.4 OCL_SP 并行算法跨平台性分析 源码级别的算法可移植性不仅要求在不同的平台上源码可以成功地编译和运行, 同时还需要算法保持相当的性能. CUDA_SP 并行算法受到硬件平台的限制, 表 3 中数据显示, 在多种硬件平台上 OCL_SP 并行算法具备了最大程度的兼容性和可移植性.

5 结束语

针对全源对最短路径图算法中数据量大、运算量大和高计算密集度等问题, 本文设计了采用 GPU 加速的最短路径并行算法. 分别从算法可并行性分析、算法描述和框架的设计、矩阵乘法的加速策略以及存储器优化策略等方面进行了阐述. 实验结果表明, OCL_SP 并行算法分别与 CPU_SP 串行算法、OMP_SP 并行算法和 CUDA_SP 并行算法的性能相比, 取得了 196.35、36.76 和 2.25 倍的加速比, 不但实现了高性能, 而且在不同 GPU 计算平台间实现了性能移植. 拥有大量计算单元和强大计算能力的 GPU, 为图论有关问题的快速求解提供了一种廉价而高效的计算平台, 为高效地大规模数值计算进行了有益探索. 在下面的工作中, 我们将把多 CPU 与多 GPU 构成的异构计算平台作为并行计算模式, 采用并行 I/O 技术, 以达到更大规模数据集对计算能力的要求.

参考文献:

[1] 刘正平, 钟 诚, 张雄宝, 等. 基于启发信息并行求解无环 K 最短路径[J]. 小型微型计算机系统, 2017, 38(7): 1 506-1 511. DOI: 10.3969/j.issn.1000-1220.2017.07.016.
Liu Z P, Zhong C, Zhang X B, et al. Parallel solving acyclic K shortest paths based on heuristic information [J]. Journal of Chinese Computer Systems, 2017, 38(7): 1 506-1 511.

[2] Khanda A, Srinivasan S, Bhowmick S, et al. A parallel algorithm template for updating single-source shortest

paths in large-scale dynamic networks[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(4): 929-940. DOI: 10.1109/TPDS.2021.3084096.

[3] Keren C H, Kaski P, Korhonen J, et al. Algebraic methods in the congested clique[J]. Distributed Computing, 2019, 32(6): 461-478. DOI: 10.1007/s00446-016-0270-2.

[4] Farias R, Kallmann M. Optimal path maps on the GPU[J]. IEEE Transactions on Visualization and Computer Graphics, 2020, 9(26): 2 863-2 874. DOI: 10.1109/TVCG.2019.2904271.

[5] Yin S Y, Hu Y M, Ren Y C. The parallel computing of node centrality based on GPU[J]. Mathematical Biosciences and Engineering, 2022, 19(3): 2 700-2 719. DOI: 10.3934/mbe.2022123.

[6] Jamour F, Skiadopoulos S, Kalnis P. Parallel algorithm for incremental betweenness centrality on large graphs [J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 29(3): 1-14. DOI: 10.1109/TPDS.2017.2763951.

[7] Zhang D T, Chow C Y, Liu A, et al. Efficient evaluation of shortest travel-time path queries through spatial mashups[J]. Radio Engineering, 2018, 22(3): 3-28. DOI: 10.1007/s10707-016-0288-4.

[8] Chandio A A, Tziritas N, Zhang F, et al. Towards adaptable and tunable cloud-based map-matching strategy for GPS trajectories[J]. Frontiers of Information Technology and Electronic Engineering, 2016, 17(12): 1 305-1 319. DOI: 10.1631/FITEE.1600027.

[9] Panomruttanarug B. Application of iterative learning control in tracking a Dubin's path in parallel parking[J]. International Journal of Automotive Technology, 2017, 18(6): 1 099-1 107. DOI: 10.1007/s12239-017-0107-4.

[10] Hung L H, Shi K Y, Wu M G, et al. fastBMA: scalable network inference and transitive reduction[J]. GigaScience, 2017, 6(10): 1-10. DOI: 10.1093/gigascience/gix078.

[11] Gyongyosi L, Imre S. Entanglement-gradient routing for quantum networks[J]. Scientific Reports, 2017, 7(1): 1-14. DOI: 10.1038/s41598-017-14394-w.

[12] 闫春望, 黄 玮, 王劲松. 一种并行模糊神经网络最短路径算法[J]. 计算机应用研究, 2016, 33(11): 3 391-3 395.
Yan C W, Huang W, Wang J S. Parallel fuzzy neural network shortest path algorithm[J]. Application Research of Computers, 2016, 33(11): 3 391-3 395.

[13] 李平, 李永树. 一种基于 Dijkstra 并行线程算法的研究与实现[J]. 测绘与空间地理信息, 2014, 37(9): 50-53. DOI: 10.3969/j.issn.1672-5867.2014.09.015.
Li P, Li Y S. Research and implementation of one parallel thread algorithm based on Dijkstra[J]. Geomatics and Spatial Information Technology, 2014, 37(9): 50-53.

[14] Stpiczynski P. Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus[J]. Journal of Supercomputing, 2018, 11(1): 1 461-1 472. DOI: 10.1007/s11227-017-2231-3.

[15] Cabrera W, Ordonez C. Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries[J]. Distributed and Parallel Databases, 2017, 35(3): 335-362. DOI: 10.1007/s10619-017-7200-6.

[16] Maleki S, Nguyen D, Lenharth A, et al. DSMR: A shared and distributed memory algorithm for single-source shortest path problem[J]. ACM Sigplan Symposium on Principles and Practice, 2016, 51(8): 31-32. DOI: 10.1145/2851141.2851183.

[17] Chakaravarthy V T, Checconi F, Murali P, et al. Scalable single source shortest path algorithms for massively parallel systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(7): 2 031-2 045. DOI: 10.1109/TPDS.2016.2634535.

[18] 孙文彬, 谭正龙, 王江, 等. 基于多粒度通讯的 Dijkstra 并行算法优化[J]. 中国矿业大学学报, 2014, 43(5): 938-943. DOI: 10.13247/j.cnki.jcumt.000226.
Sun W B, Tan Z L, Wang J, et al. A parallel Dijkstra algorithm based on multi-granularity communication[J]. Journal of China University of Mining and Technology, 2014, 43(5): 938-943.

[19] Liu W, Vinter B. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors[J]. Journal of Parallel and Distributed Computing, 2015, 85(4): 47-61.

[20] 李寅, 邓仰东. 基于 GPU 的混合式全源对最短路径算法研究[J]. 微电子学与计算机, 2016, 33(2): 76-82. DOI: 10.19304/j.cnki.issn1000-7180.2016.02.017.
Li Y, Deng Y D. GPU-based hybrid algorithm of all-pairs shortest paths[J]. Microelectronics and Computer, 2016, 33(2): 76-82.

[21] 叶颖诗, 魏福义, 蔡贤资. 基于并行计算的快速 Dijkstra 算法研究[J]. 计算机工程与应用, 2020, 56(6): 58-65. DOI: 10.3778/j.issn.1002-8331.1903-0119.
Ye Y S, Wei F Y, Cai X Z. Research on fast Dijkstra algorithm based on parallel computing[J]. Computer Engineering and Applications, 2020, 56(6): 58-65.

[22] Djidjev H, Chapuis G, Andonov R, et al. All-pairs shortest path algorithms for planar graph for GPU-accelerated clusters[J]. Journal of Parallel and Distributed Computing, 2015, 85(4): 91-103. DOI: 10.1016/j.jpdc.2015.06.008.

[23] Aridhi S, Lacomme P, Ren L, et al. A MapReduce-based approach for shortest path problem in large-scale

networks[J]. Engineering Applications of Artificial Intelligence, 2015, 41(2): 151-165. DOI: 10.1016/j.engappai.2015.02.008.

[24] Gayathri R G, Nair J. Ex-FTCD: A novel MapReduce model for distributed multi source shortest path problem[J]. Journal of Intelligent and Fuzzy Systems, 2018, 34(6): 1 643-1 652.

[25] 何亚茹, 庞建民, 徐金龙, 等. 基于神威平台的 Floyd 并行算法的实现和优化[J]. 计算机科学, 2021, 48(6): 34-40. DOI: 10.11896/jsjkx.201100051.
He Y R, Pang J M, Xu J L, et al. Implementation and optimization of Floyd parallel algorithm based on Sunway platform[J]. Computer Science, 2021, 48(6): 34-40.

[26] Georg M, Florian P, Matthias W, et al. Real-time multitarget tracking for sensor-based sorting a new implementation of the auction algorithm for graphics processing units[J]. Journal of Real-Time Image Processing, 2019, 16(6): 2 261-2 272. DOI: 10.1007/s11554-017-0735-y.

[27] Rodriguez-canal G, Torres Y, Andujar F J, et al. Efficient heterogeneous programming with FPGAs using the controller model[J]. Journal of Supercomputing, 2021, 77(12): 13 995-14 010. DOI: 10.1007/s11227-021-03792-7.

[28] Firmansyah I, Yamaguchi Y. Real-time FPGA implementation of FIR filter using OpenCL design[J]. Journal of Signal Processing Systems for Signal Image and Video Technology, 2022, 94(1): 117-129. DOI: 10.1007/s11265-021-01723-6.

[29] Chen G, Wen G H, Yu X H. Performance analysis of distributed short-path set based routing in complex networks[J]. IEEE Transactions on Circuits and Systems II-Express Briefs, 2019, 66(8): 1 426-1 430. DOI: 10.1109/TCSII.2018.2882515.

[30] Chen B Y, Chen X W, Chen H P, et al. A fast algorithm for finding K shortest paths using generalized spur path reuse technique[J]. Transactions in GIS, 2021, 25(1): 516-533. DOI: 10.1111/tgis.12699.

[31] Wehmuth K, Ziviani A, Costa L C, et al. You shall not pass: Avoiding spurious paths in shortest-path based centralities in multidimensional complex networks[J]. IEEE Transactions on Network Science and Engineering, 2021, 8(1): 138-148. DOI: 10.1109/TNSE.2020.3030749.

All-source pair shortest path parallel algorithm based on GPU acceleration

XIAO Han[1,3], XIAO Shi-yang[2], LI Huan-qin[1**], ZHOU Qing-lei[3]

(1. School of Information Science and Technology, Zhengzhou Normal University, Zhengzhou 450044, Henan, China;
2. School of Civil Engineering, Southeast University, Nanjing 211189, Jiangsu, China;
3. School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou 450001, Henan, China)

Abstract: Aiming at the problem that the shortest path algorithm is inefficient in processing large-scale datasets, this paper proposes an all-pairs shortest paths parallel algorithm based on Graphics Processing Unit (GPU) acceleration. Firstly, the optimized matrix multiplication algorithm is used to realize the parallel computing data inter-workgroup and intra-workgroup. Then the branch of work-items caused by irregular rows is reduced. Finally, the access delay of work-items to the strip storage resources of adjacency matrix calculation is reduced. The experimental results show that compared with the performance of the serial algorithm based on AMD Ryzen5 1600X CPU, parallel algorithm based on Open Multi-Processing (OpenMP) and parallel algorithm based on Compute Unified Device Architecture (CUDA), the shortest path parallel algorithm obtains 196.35, 36.76 and 2.25 times speedup in the NVIDIA GeForce GTX 1070 computing platform under the Open Computing Language (OpenCL) architecture respectively. The validity and performance portability of the proposed parallel optimization method are verified.

Key words: shortest path; repeating square method; Graphics Processing Unit (GPU); Open Computing Language (OpenCL); parallel algorithm