软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(4):1225−1239 [doi: 10.13328/j.cnki.jos.005627]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 GPU 平台的复杂网络 core 分解方法研究*

张 珩, 崔 强, 侯朋朋, 武延军, 赵 琛

(中国科学院 软件研究所,北京 100190)

通讯作者: 张珩, E-mail: zhangheng17@iscas.ac.cn

摘 要: 在复杂网络理论中,core 分解是一种最基本的度量网络节点“重要性”并分析核心子图的方法.Core 分解广泛应用于社交网络的用户行为分析、复杂网络的可视化、大型软件的代码静态分析等应用.随着复杂网络的图数据规模和复杂性的增大,现有研究工作基于多核 CPU 环境设计 core 分解并行算法,由于 CPU 核数和内存带宽的局限性,已经无法满足大数据量的高性能计算需求,严重影响了复杂网络的分析应用.通用 GPU 提供了 1 万以上线程数的高并行计算能力和高于 100GB/s 访存带宽,已被广泛应用于大规模图数据的高效并行分析,如广度优先遍历和最短路径算法等.为了实现更为高效的 core 分解,提出面向 GPU 平台下的复杂网络 core 分解的两种并行策略.第 1 种 RLCore 策略基于图遍历思想,利用 GPU 高并发计算能力对网络图结构自底向上遍历,逐步迭代设置各节点所属的 core 层;第 2 种 ESCore 策略基于局部收敛思想,对各节点从邻居节点当前值进行汇聚计算更新直至收敛.ESCore 相比 RLCore 能够大大降低遍历过程中 GPU 线程更新同一节点的同步操作开销,而其算法的迭代次数受收敛率的影响.在真实网络图数据上的实验结果表明,所提出的两个策略在效率和扩展性方面能够大幅优于现有其他方法,相比单线程上的算法高达 33.6 倍性能提升,且遍历边的吞吐性能(TEPS)达到 406 万条/s,单轮迭代的 ESCore 的执行效率高于 RLCore.

关键词: 复杂网络;GPU;Core 分解;大规模图数据;大数据处理

中图法分类号: TP311

中文引用格式: 张珩,崔强,侯朋朋,武延军,赵琛.面向 GPU 平台的复杂网络 Core 分解方法研究.软件学报,2020,31(4):1225−1239. http://www.jos.org.cn/1000-9825/5627.htm

英文引用格式: Zhang H, Cui Q, Hou PP, Wu YJ, Zhao C. Accelerating core decomposition in complex network on GPUs. Ruan Jian Xue Bao/Journal of Software, 2020,31(4):1225−1239 (in Chinese). http://www.jos.org.cn/1000-9825/5627.htm

Accelerating Core Decomposition in Complex Network on GPUs

ZHANG Heng, CUI Qiang, HOU Peng-Peng, WU Yan-Jun, ZHAO Chen

(Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: To analysis the complex networks, core decomposition is a basic and efficient strategy to distinguish the relative importance of nodes and to discover a special family of core subgraphs in networks. After core decomposition, every node in each *k*-core subgraph connects to other *k* neighbor nodes internally. The core decomposition has been widely applied in several application scenarios, e.g., user behavior analysis in social networks, visualization of complex networks, static analysis in large system software project, etc. With the increasing scale and complexity of networks, existing works, which mostly focus on the multi-core CPU-based implementation of core decomposition, cannot satisfy the high performance of core decomposition in large-scale complex networks. Meanwhile, GPU provides not only massive parallelism degree (up to 10 000 threads) but also efficient memory I/O bandwidth (approximately 100 GB/s), which makes it an excellent hardware platform for large graph structure analytic, such as BFS (breadth first search), SSSP (single source shortest path) algorithms. This study proposes two strategies to enhance the parallel performance of core decomposition on GPU-based platform.

* 基金项目: 国家重点研发计划(2018YFB0803600); 国家自然科学基金(61702488, 61732020)

Foundation item: National Key R&D Program of China (2018YFB0803600); National Natural Science Foundation of China (61702488, 61732020)

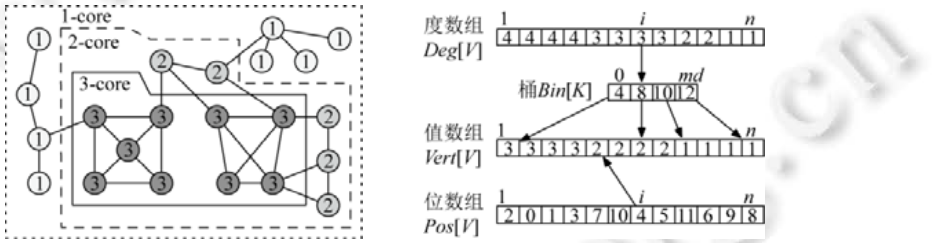
收稿时间: 2017-07-14; 修改时间: 2017-09-01; 采用时间: 2018-02-09

An algorithm, RLCore, is first presented which exploits GPU-based bottom-up traverse approach and recursively distinguishes the core levels of nodes by considering their degree and edges. Then, a second optimal algorithm is proposed to improve performance and scalability, namely ESCore, based on the locality property of core decomposition. In ESCore, nodes gather and update their core level values from their neighbors, until there is no update among nodes. Compared to RLCore, ESCore strategy reduces the synchronization overhead from multi-thread contention when scaling to massive parallelism, whereas the iteration number of ESCore is depended on the convergence of nodes. From the evaluation results, two proposed acceleration algorithms achieve maximum 4.06 billion TEPS (traversed edges per second), which corresponds to up to 33.6X speedup compared to a single threaded CPU execution.

Key words: complex network; GPU; core decomposition; large graph; big data processing

近年来,随着互联网与物联网的大规模发展,各类事物之间的联系更为紧密,这些错综复杂的关联关系组成了规模庞大的网络,例如,以人为节点、以关系为边的社交网络,以地点为节点、以路径为边的地理空间交通网络等.随着复杂网络的规模不断扩大,图的节点数目不断增多,节点间和子网络之间的关联关系变得越发复杂,整体表现出以不规则关联结构化的大数据复杂性.这类复杂网络通常采用图结构(点,边)形式的数据集进行表示,对这类数据集的分析方法能够被广泛应用于定向广告、欺诈检测、缺失链分析、定位相互作用蛋白质的高通量算法等各类领域.

在复杂网络分析方法中,大量研究提出了一系列检测复杂网络中相对“重要”的节点度量方法[1,2],包括利用网络节点的特征向量(eigenvector)、中间性(betweenness)和集中性(centrality)等特性进行评估.Core 分解作为重要的复杂网络分析操作,能够给出复杂网络中核心子图集所包含的节点,通常将所分解出来的各层子图集合命名为 *k*-核(*k*-cores 或 *k*-shells).如图 1(a)所示,复杂网络的 core 分解在给定所有可能的 *k* 值情况下,第 *k* 层最大子图包含所有节点度不小于 *k* 的节点集合.Core 分解从最小度的节点集开始不断迭代设置在当前子图层中所有节点的 core 数,并经过不断的图约减,最终获得每个节点的核数.Core 分解能够支持各类不同类型和模式的复杂网络的结构分析与预测[1],例如社交网络的用户行为量化、复杂图的可视化、蛋白质网络的角色定位、大规模软件系统的静态分析等应用.因此,对复杂网络的 core 分解的策略研究具有重要的研究价值和科学意义.



(a) 图示例与其 core 分解结构 (b) Core 分解的 BZ 算法示例图

Fig.1 A sample graph and its core demoposition, and core decomposition execution flow of BZ alogorithms

图 1 范例复杂网络图与 core 分解结构的示意图以及 BZ 算法进行复杂网络图 core 分解流程示例

随着复杂网络的规模不断扩大,core 分解的计算效率问题越来越为研究者所关注.现有 core 分解研究主要从两个方面进行解决:一方面是算法的改进,例如,目前应用广泛的线性复杂度的 BZ 算法[3]基于桶排序思想优化(如图 1(b)所示),EMcore[4]基于外存顺序 I/O 优化数据读写提升 core 分解性能;另一方面是采用并行计算代替串行计算.在对并行架构的选择考量中,需要综合考虑并行硬件平台的开发和维护成本以及效率的提升率.现有的 ParK[5]算法适用于多核 CPU 下 core 分解的性能扩展,然而该算法受限于 CPU 的多核处理性能和内存带宽;分布式的 core 分解[6]实现了可扩展性算法,然而没有考虑集群计算的可容错性.近年来,GPU 的硬件性能大幅度提升,多处理器内核能够支持超大规模并行的环境(线程数量达到 1 万以上,访存带宽高达 100GB/s),并提供高于 CPU 计算数十倍乃至上百倍性能的硬件处理能力.GPU 已广泛应用于大规模图数据上以加速各类图算法[7−10],如广度优先遍历和最短路径算法等,具有高性能、低成本和低能耗的优势.因此,本文提出了面向 GPU 平台复杂

网络的 core 分解策略的设计与实现,用以大幅度提升复杂网络的信息挖掘和检测的处理效率,并支持更大规模的网络数据分析.利用 GPU 进行复杂网络的 core 分解需要综合考虑大规模 GPU 线程的高效调度和 GPU 各层级内存的有效利用.据对现有工作调研所知,本文的研究为首次在面向 GPU 平台上对复杂网络的 core 分解策略进行设计和实现.

本文针对高并行的 GPU 环境设计并实现了两种复杂网络 core 分解策略,RemoveList(RLCore)策略和 Estimate(ESCore)策略.第 1 种 RLCore 策略,基于图遍历计算方法,利用 GPU 硬件平台的高并行计算性能优势,每一轮对网络图自底向上遍历,从外层到内层的节点 core 分解递归得到各个 core 结构和每个节点所属的核数;该策略实现分为 Update 和 Scatter 两个 GPU Kernel 函数,分别进行节点扫描并更新标记和邻居节点的度更新,给定迭代次数为 τ,RLCore 策略的总时间复杂度为 $O(\tau\cdot(m+2n))$.第 2 种 ESCore 策略基于复杂网络 core 分解局部性的定理,各节点的 core 核数的确定能够根据其邻居节点集的 core 值进行迭代更新直至收敛.我们设计 ESCore 策略对各节点根据邻居节点的当前值进行汇聚计算后更新,整个处理流程各线程分离进行节点值写入操作,ESCore 策略的总时间复杂度为 $O(\tau\cdot(m+n))$.通过算法行为分析,ESCore 相比 RLCore,能够大大降低遍历过程中 GPU 线程更新同一节点的同步操作开销,而其算法的迭代次数受收敛率的影响(见第 4.3.4 节验证).最后,我们对所提 RLCore 和 ESCore 策略与经典的 BZ 和现有多核环境下最新的 ParK 算法进行了对比实验,从多角度验证了所提策略的性能和优势.

本文主要贡献总结如下.

(1) 针对复杂网络 core 分解的并行,本文首次提出面向 GPU 并行环境对 core 分解进行性能优化.设计并实现第 1 种面向通用 GPU 的 core 分解并行策略 RLCore,基于图自底向上遍历,GPU 的各线程逐层迭代分解获取各 core 结构和各节点核数;

(2) 进一步地,基于 core 分解的局部性定理,设计并实现了第 2 种 core 分解并行方法,ESCore 策略,优化高并发 GPU 环境下多线程执行能力.结果表明,ESCore 可大幅度降低 GPU 多线程的同步操作开销.

(3) 实验结果同时也表明,RLCore 和 ESCore 策略与经典的 BZ 算法相比达到了 8.8-33.6X 倍的性能提升,同时对比 8 线程下的 ParK 算法,达到了 2X-8X 的加速比,对比 16 线程下的 ParK 算法,达到了 2X-5.6X 的加速比.另外,实验也从多角度验证了所提方法的性能优势.在数据规模不断扩大的情况下,RLCore 与 ESCore 并行处理的效率提升得越发明显.

本文第 1 节介绍面向复杂网路的 core 分解的问题及定义,以及面向复杂网络的 core 分解的相关工作描述,并给出复杂网络 core 分解问题的形式化定义.第 2 节描述对复杂网络 core 分解的方法推导,并基于求解结论分别设计并实现基于 GPU 平台下的复杂网络 core 分解的两个并行策略:RLCore 和 ESCore 策略.进一步地,我们对这两个并行策略进行算法的时间和空间复杂度分析,并对在 GPU 上的并行处理的各方面问题进行分析和对比.第 3 节描述具体的实验平台、实验设计以及实验结果和分析.最后是本文的总结和未来展望.

1 研究背景与相关工作

1.1 GPU计算主要硬件平台

在 GPU 的并行算法设计中,本工作主要的算法设计基于 NVIDIA GPU 来进行(如下数据根据实验 NVIDIA GTX 980 所得).由于 GPU 并行编程不同于 CPU 上的多核和多线程程序并行,因此需要明确如下 GPU 平台下的部分相关概念.

• GPU 处理单位.GPU 中 SMX(streaming multiprocessor),例如 Maxwell 架构的 GTX 980 能够拥有 16 个 SMX,每个 SMX 拥有 2 048 个单精度的 CUDA cores,每一个 GPU 线程在一个 CUDA core 上执行.SMX 支持达 64 个 Warp 单元,每个 Warp 单元同时调度 32 个线程执行.

• Kernel.在 GPU 执行应用过程中,通常编程按照线程块 block 进行调度.线程块(thread block),又称为协作线程组(cooperative thread array,简称 CTA),能够包含 1~64 个 Warp 单元.所有线程块集合称作一个网格(grid),执行相同内核程序的一系列线程块.一个 kernel 定义为在 GPU 上执行的任意函数.通常一个 kernel 可采用不同的并

行粒度(一个线程,Warp,Block 或者 Grid)的一定数量的线程进行调度执行.

• GPU 内存层级.1 个 SMX 拥有 65 335 个寄存器.同时,每个 SMX 提供了 48KB 的共享内存(shared memory,L1 cache)用于 Warp 单元间和线程块间的数据通信.同时,共享内存大小可为应用自定义配置为 16KB、32KB 和 48KB.另外,2MB 的 L2 cache 和 4GB 的全局内存能够提供所有 SMX 之间的数据同步.

表 1 总结了 CPU 和 GPU 的内存层级带宽和数据存储.通过对比可见,GPU 拥有超大规模的寄存器单元,能够提供大规模并行化计算,并且提供一定大小的访存.值得注意的是,通过测算 GPU 下的内存访问,寄存器和共享内存的访问带宽能够达到全局内存 12X.

在对复杂网络的 core 分解之中,本文构建的算法模型提供可选择的两种文件输入格式:图结构以边列表(edge list)和行压缩(compressed sparse row,简称 CSR)进行格式化输入.点与边以结构体形式表示(含权重和编号).对图的预处理采用按文件行读入,在加载图数据后,能够得到图中的点数 n、边数 m 以及以向量(vector)形式表示的点列表(vertex list)和边列表(edge list).在基于大规模并行的 GPU 架构下的计算考虑 CPU 与 GPU 之间的协同计算,其中包括 CPU 与 GPU 擅长处理的数据结构的不同,在 PCI-E 的数据通信、主存与访存间的数据置放以及 GPU 内部计算单元负载均衡与同步等方面.

Table 1 CPU (Xeon E5-2650) vs. GPU (NVIDIA GTX 980) memory and access latency (in CPU and GPU cycles)

表 1 CPU(英特尔 Xeon E5-2650)与 GPU(英伟达 GTX 980)内存大小和访问延迟(cycles)对比,以及两个 core 分解策略(RLCore 和 ESCore 策略)的数据结构分布情况

内存	Intel Xeon E5-2650		NVIDIA GTX 980			
	大小	延迟	大小	延迟	RLCore 数据存储	ESCore 数据存储
寄存器	10	1	65 336	–		
L1 cache	64KB	4	48KB (shared)	–	vplist, rmlist, flaglist	statelist, emlist
L2 cache	256KB	10	2MB	–		
L3 cache	24MB	40	–	–		
DRAM	64GB	55	4GB (global)	200~400	图结构向量(点、边),Buffer 缓存	图结构向量(点、边),Buffer

1.2 相关工作

近些年来,超大规模核处理器 GPU 的高性能处理节点,为大量并行算法的设计提供了高性能、低成本和低能耗的硬件平台.面向 GPU 高并行计算平台的图算法研究工作主要集中于大规模图数据的基础算法并行化,可以提供数十倍至上百倍于多核 CPU 性能的处理能力[7,8,11−13].通过对相关工作的调研可以看出,复杂网络 core 分解策略的优化工作集中在算法的机理优化、算法的并行化以及算法的应用场景研究.其中,机理优化包括单节点下算法复杂度与计算本地性优化以及外存图数据读写与计算优化等;并行化研究包括多核 CPU 下并行机制研究和分布式并行化.此外,若干研究工作提出复杂网络的多样性、异常节点检测等特征 core 分解研究.

复杂网络的 core 分解首次由 Seidman[14]提出.在复杂网络 core 分解策略中,Batagelj 和 Zaveršnik[3]提出了基于桶排序复杂度 $O(n+m)$的分解算法(简称为 BZ 算法).具体地,该分解算法利用两个数组、值和位移数组分别保存排序后节点及其在值数组的偏移位;每一分解层的节点被移除后,其邻居节点的度被累减后移至对应值数组位置(如图 1(b)所示).考虑到 BZ 算法访问方式为点边随机检索,需要将所有节点和边数据以及中间结果集完全加载到内存中来保证性能,因此,BZ 算法空间复杂度高且数据的随机读写延迟高.随着复杂网络数据集规模的扩大,整个原始数据集无法全部存储于内存中,导致传统的 BZ 算法无法满足更大规模的 core 分解.Cheng 等人[4]提出了外存计算策略 EMcore 以应对更大规模的网络数据,先进行内存存储的子图切分,然后对各个切分的子图递归处理各 K-core 结构,逐步约减所得 K-cores.EMcore 策略能够处理更大规模的图数据,然而,其每一轮迭代均需迭代整个图数据,导致其处理稀疏图结构时引入大量不必要的计算.Khaouid 等人[15]以节点为中心(vertex-centric)思想基础上进行了复杂网络 core 分解的算法实现,在两个外存图处理系统 GraphChi 和 WebGraph 上借鉴 EMcore 和 BZ 算法的思路,调用两个系统编程接口(API)分别进行 core 分解实现.O'Brien 等人[16]提出利用复杂网络 core 分解的局部性进行网络中节点的中心度评估的近似算法,该工作对图结构原数据

进行约减,根据图的节点的半径范围进行近似估算,评价图中节点的中心度.

随着多核服务器和分布式集群计算的普及,复杂网络 core 分解策略研究趋向于利用大规模并行与分布式环境开展并行化算法优化.其中,Naga 等人[5]提出复杂网络 core 分解在多核处理器上的实现.该工作降低了 BZ 算法的大量数据随机访问读写开销,在此基础上提出了 ParK 算法用于复杂网络的并行 core 分解,将图中的每个节点进行层级(level)划分,逐层独立处理后同步.Alberto 等人[6]实现了复杂网络 core 分解的分布式策略,该工作借鉴 Google Pregel[17]的分布式消息传递策略,利用 core 分解局部性的计算特性进行分布式算法实现,复杂网络中每个节点计算得到核数之后,将更新后的核数(如⟨*u*,*core*⟩)进行消息传递给邻居节点,进一步对消息处理得到新的核数.另一方面,也有部分研究利用基于 core 分解提出了对复杂网络的分析策略,支持对网络结构进行结构分析、动态和静态检测等.Shin 等人[1]利用复杂网络进行 core 分解策略,对复杂网络的结构进行了深入的分析,其中包括对网络的模式结构、异常节点检测以及算法等进行了研究,从分析的结果来看,该工作相比相关的其他影响力传播模型算法,在保证可比的精确度的前提下性能提升 17X.

对比于上述相关工作,本文在 GPU 平台上对复杂网络 core 分解加以实现,利用 GPU 的多处理内核实现 core 分解算法的高性能并行计算,相比于上述 CPU 策略,可以提供高达数十倍性能的处理能力,具备高性能、低成本和低能耗的优势.据对现有工作调研所知,本研究工作为首次在 GPU 平台上对复杂网络的 core 分解策略进行设计和实现.

本文工作着重于复杂网络的 core 分解在 GPU 高并行处理环境下研究,提出了针对 core 分解的 Locality 特性的 GPU 的优化方法,并针对 GPU 下 core 分解处理图过程中的负载不均衡和数据读写 I/O 问题进行优化.该策略能够提供上述相关工作 GPU 下图遍历的优化策略和基于 GPU 的图并行处理系统中的底层设计以一定的借鉴意义.同时,上述相关工作所用到的优化方法和策略也能用于复杂网路的 core 分解策略的优化.

2 复杂网络 core 分解的基本概念

给定一个复杂网络图 G(数据图),则 k-core 子图 $G(k)$需要满足 $G(k)$包含的每一个节点的度数至少为 k;图 G 的节点所属核编号定义为包含该节点子图的最大阈值 k,计算 G 中的每个节点的所属核数(coreness).给定节点 $u \in G$ 的核数最大值 k 满足于节点 u 至少有 k 个邻居点属于一个 k-core 或者一个更大的核结构.本小节对复杂网络下 core 分解问题的基本概念进行描述.

定义 1(复杂网络). 复杂网络中,主要表现为以节点与节点间的紧密关联,各个子网络内部节点连接紧密,而子网络外节点连接相对稀疏.随着复杂网络规模不断的扩大,在复杂网络的表现形式中,这种紧密关联关系通常表示为节点(vertex)和边(edge)的图结构形式.

定义图 $G=(V,E)$,点集为 $V(G)=\{\Sigma v_i | i=1,2,...,n\}$且节点数为 n,边集为 $E(G)=\{\Sigma e_i | i=1,2,...,m\}$且边数为 m.给定一个节点子集 $V_c \subseteq V$, 根据 V_c 点集导出的子图 $G(V_c)$为图 G 的一个子图满足 $G(V_c)=(V_c,\{(u,v)\in E(G)\,|\,u,v\in V_c\})$.

定义 2(节点度与邻居). 给定节点 u,节点 u 与图 G 中其他点有边进行关联,其邻居点集 $nbr(u,G)$定义为 $nbr(u,G)=\{v\,|\,(u,v)\in E(G)\}$; 节点 u 的度 $deg(u,G)$为关联边的总数,即 $deg(u,G)=|nbr(u,G)|$.简述起见,后续内容采用 $nbr(u)$和 $deg(u)$分别表示节点 u 在图 G 的邻居点集和节点度.

定义 3(*K*-core). 给定一个图 G 和一个整数 k,那么图 G 的 k-core 子图记为 G_k,表示为图 G 中满足每一个节点的度至少为 k 的一个最大子图, $\forall v\in V(G_k), \deg(v,G_k)\geqslant k$.

给定 $k_{\max}$ 为图 G 中满足一个 k-core 结构存在的最大可能值 k,那么图 G 中对所有在$[1,k_{\max}]$范围内的 k 值的 k-core 结构满足如下特征:

$$\forall 1\leqslant k<k_{\max}: G_{k+1}\subseteq G.$$

定义 4(core 核数). 对图 G 中的每一个点 $v\in V(G)$,其 core 核数记为 $core(v,G)$,表示为点 v 所在的 k-core 结构中的最大的 k 值, $core(v,G)=\max\{k\,|\,v\in V(G_k)\}$. 简述起见,后续内容采用 $core(v)$来表示节点 v 在图 G 中的核数 $core(v,G)$.

给定一个图 G 和一个整数 $k, V_k=\{v\in V(G)\,|\,core(v)\geqslant k\}$, 则 $G_k=G(V_k)$.

3 面向 GPU 的复杂网络 core 分解并行策略

本节详细介绍面向 GPU 的 core 分解策略的构建与实现过程.GPU 采用基于单指令流多数据流(single instruction multiple data,简称 SIMD)架构大规模的并行架构,擅长于高并发的简单处理任务,本文的 core 分解策略能够扩展支持其他高并行平台的实现.GPU 平台的特点是向量与矩阵的计算能力强,同时复杂网络的图结构数据异构性强且现实图的节点度分布通常呈现 Power-law 分布形式,因此,复杂网络的 core 分解需要综合考虑算法的可并行性、网络图数据结构的内存分布和访问策略(随机与顺序,访问页大小等).

首先,我们基于图遍历方法和 core 分解中 *K*-core 结构的定义,提出了 RLCore 策略,自底向上地对网络图进行逐层分解.每个GPU线程访问一个节点,逐步将外层core结构内的各节点根据度数来确定所属的核数,然后移除并更新其所有邻居点的度.RLCore 策略实现 Update 和 Scatter 两个 GPU Kernel 函数分别进行节点扫描并更新标记和邻居节点的度更新(详细讨论见第 3.1 节).其次,我们基于 core 分解的局部性特征,给出方法推导,每个 *K*-core 结构内的每个节点的核数等于其满足 core 条件的邻居数.进一步地,我们提出了第 2 个 core 分解策略:ESCore,各节点的 core 核数根据其邻居节点集的 core 值进行独立的迭代计算,并更新直至所有节点的 core 核数收敛(详细讨论见第 3.2 节).最后,我们对 RLCore 和 ESCore 策略分别进行了算法复杂度和 GPU 下执行的分析对比.RLCore 的迭代次数为 $k_{\max}$,然而其在对各节点的邻居点同时访问更新度时,会导致大量的内存冲突开销,即多个线程同时写入一个内存地址会产生大量的同步等待开销,将会导致算法性能和可扩展性大幅下降.ESCore流程分离迭代计算,有效降低了在高并行GPU计算环境下的同步操作开销,而其迭代次数取决于图中各节点的 core 值的收敛程度.

3.1 RLCore并行策略

在对复杂网络 core 分解的定义 4 中进行推导各节点的 core 分解特性[3],对所有节点的 core 值满足如下的定理 1.

定理 1. 给定复杂网络的图 *G*,节点 *v* 在图 *G* 的 *k*-core 结构中的条件是,当且仅当节点 *v* 至少有 *k* 个邻居点在 *k*-core 的子图 G_k 中.

因此,给定复杂网络的图 *G* 和整数值 *k*,递归删除所有度数小于 *k* 的节点(其边也会删除,因此邻居节点的度也会累减),并标记这些节点,剩余的图就是图 *G* 的 *k*-core 结构 G_k.

依据上述定理 1,我们构建了基于 GPU 下的 core 分解的 RemoveList 算法.算法的基本思想为(如图 2 所示):每一轮迭代自底向上遍历点,从外层的节点度为 1 的点开始计算核数并删除,直到所有点处理完成,最终递归得到最大的 *K* 层 core 结构和每个点所属的核数.该算法中,利用 3 个数组分别记录对应的中间结果集,以利于递归的 core 分解操作.

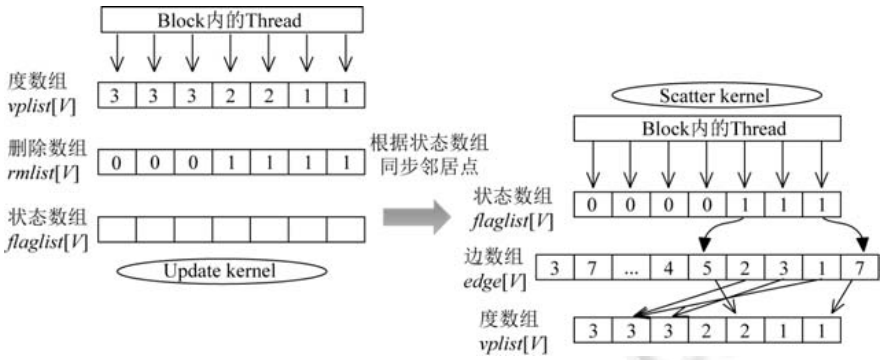
• 节点属性(度)数组 vplist.在复杂网络的 core 分解中主要用到的节点的属性为各节点的度(含出度和入度),需要根据度来确定各节点的所属 core 结构.其中,需要注意的是,每一轮迭代删除度数小于 *k* 的节点后,其邻居节点的度也会累减,因此,vplist 每一轮迭代需要大量的读写访问.

• 删除点集数组 rmlist.用于减少不必要的计算量,标记已处理完成后删除的点.记录复杂网络的 core 分解的每一轮迭代过程中不在 *K*-core 结构(不满足整数 *K*)的点的编号集合;以 bool 向量(长度为 *n*)$\langle r_0,r_1,...,r_n\rangle$ 对每个节点是否删除的状态进行标记.

• 点状态位数组 flaglist.考虑到在 core 分解中删除不满足 *k*-core 结构的点之后,所删除的点的邻居节点的度也会累减,我们创建 flaglist 以标记各点在被标记为已删除状态后需要进一步对其所关联的邻居点集处理的状态位,若点在迭代过程中不满足当前的 *k*-core 值,则标记为删除状态且对应 flaglist 位标记为活跃,反之则标记为不活跃;以 bool 向量(长度为 *n*)$\langle s_0,s_1,...,s_n\rangle$ 来进行标记图中每个节点的活动状态.

根据上述定理 1,基于 GPU 的 RLCore 策略,整体输入为复杂网络的图 *G* 和外部迭代循环控制的每层 *k*-core 整数值.另外,创建设备的状态标记 device_finish 和全局的删除点集个数标记 rmlist length;分别创建上述 3 个数组的 Host 变量和 Device 变量,用于记录节点属性、删除点集以及点状态位标记,对 Device 变量进行缓存分配

并于 Host 对应变量间建立映射.进一步地,核心的复杂网络 core 分解处理流程拆分为两个 GPU 的 kernel.



(a) RemoveList 第 1 步对节点状态更新 (b) RemoveList 第 2 步邻居节点同步

Fig.2 Parallel execution flow of RLCore core decomposition

图 2 RLCore 并行策略流程示例

算法 1(a)中的 UpdateKernel 阐述了第 1 个步骤,每个线程根据所在 Block 的 threadId 判断处理一个对应节点(第 1 行~第 2 行),给定点 v_i 对 rmlist 中的标志 r_i 进行判断(第 5 行),若其已被剔除,说明点 v_i 已隶属于外层 k-core 结构,否则,继续判断 vplist 中所记录的点 v_i 的当前度数 $deg(v_i)_{cur}$(第 7 行).若值 $deg(v_i)_{cur}$ 不小于当前所处理的 core 核数 $kcore$,说明 v_i 属于当前 core 结构,否则,v_i 不属于当前 core 结构,对 rmlist 标记剔除点 v_i(第 9 行),并标记 flaglist 对应位为 true(第 10 行),表示 v_i 已被删除.同时,相应的 v_i 关联的边也会被删除,因此需要进一步对其邻居点的度进行修正.

算法 1(a). RLCore Flag Update Kernel (RemoveList 更新及点集状态更新).

输入:数据图 Graph,核号 k-core,点集的度数组 vplist,删除点集标记 rmlist,点集状态位 flaglist;

输出:remove 标记数组 rmlist.

方法://RLCore 的第 1 步 Step 1: Update the RemoveList and States of Vertices

(1) **Tid=**blockIdx.x*blockDim.x+threadIdx.x;//GPU 线程号
(2) **IF Tid>=**total count of vertices in Graph
(3) **THEN** return;
(4) **Vid=Tid**; //GPU 线程<=>Vid 节点
(5) **IF** rmlist[**Vid**]**==false** //Vid 节点的当前处理状态
(6) **THEN**
(7) **IF vplist**[**Vid**]**<kcore** //Vid 节点在当前子图的度
(8) **THEN**
(9) Set **rmlist**[**Vid**] value to **true**;
(10) Set **flaglist**[**Vid**] value to **tru***e*;
(11) **AtomicAdd** removed count (**rmlist length**) by 1;

算法 1(b)的 ScatterKernel 阐述了 RLCore 分解的第 2 个步骤,对算法 1(a)中 flaglist 所标记的活跃点进一步进行处理.由于删除了不属于该层 core 结构的点,同时点所关联的边也一并被删除了,因此需要对删除的点集的邻居点的度进一步加以修正.遍历 flaglist 标记位数组,对已标记的点进行如下处理流程:遍历点所关联的边集,取得每条边的终点,对所取的邻居点判断是否已删除,若未被删除,则其度累减.其中所涉及到的图的 3 个函数接口说明如下.

• firstedge_index.根据点的 id,得到其第 1 条边的索引位置.
• lastedge_index.根据点的 id,得到其最后一条边的索引位置.

• edge_dest.根据边的索引,得到该边的终点的 id.

算法 1(b). RLCore Scatter Kernel(根据 RemoveList 对邻居点集的度数组更新).

输入:数据图 Graph,点集的度数组 vplist,删除点集标记 rmlist,点集状态位 flaglist;

输出:剩余点集更新后的 vplist.

方法://RLCore 的第 2 步 Step 2: Update the degree of neighborhoods for new added vertices in rmlist array

(1) **Tid=**blockIdx.x*blockDim.x+threadIdx.x;//GPU 线程号
(2) **IF Tid>=**total count of vertices in Graph
(3) **THEN** return;
(4) **Vid=Tid**; //GPU 线程<=>Vid 节点
(5) **IF** flaglist[**Vid**] value is true
(6) **THEN**
(7) Set **StartID** value to the first edge index of **Vid** vertex invoking Graph.firstedge_index(**Vid**);
(8) Set **EndID** value to the last edge index of **Vid** vertex invoking Graph.lastedge_index(**Vid**);
(9) **FOR EACH** vertex ID in **StartID** to **EndID** range //对 Vid 的邻居点集遍历
(10) Set **DestID** value to the destination vertex invoking Graph.edge_dest(ID);
(11) **IF rmlist**[**DestID**]==false
(12) **THEN**
(13) **AtomicSub** &(**vplist**[DestID]) by 1;
(14) Set **flaglist**[**Vid**] value to false;

3.2 ESCore并行策略

我们对复杂网络的 core 分解进行进一步的方法推导,根据 core 分解的局部性定理能够对各个节点的 core 的确定流程做到分离迭代计算.定理如下所述.

定理 2[4,6]. 给定图 G,对所有点 $v \in V(G)$ 的核数 $core(v)$必须同时满足如下两点.

1) 存在两个条件的 $V_k \subseteq nbr(v)$, 满足: $|V_k| = core(v)$, 对 $\forall u \in V_k$, $core(u) \geqslant core(v)$;并且

2) 不存在满足如下条件的: $V_{k+1} \subseteq nbr(v)$: $|V_{k+1}| = core(v)+1$ 并且对 $\forall u \in V_{k+1}$, $core(u) \geqslant core(v)$; 进一步,依据上述定义,给定 $u_1,u_2,u_3,...$表示为节点 v 的 k-排序邻居节点,我们推导了如下的求解公式:

$$core(v)=k_{\max}, \text{满足} \left|\{u \in nbr(v) \mid core(u) \geqslant k\}\right| \geqslant k \text{ 的最大 } k \text{ 值 } k_{\max} \tag{1}$$

证明:

∵定义 3 中,给定一个图 G 和在 k-core 子图 G_k 中的任意节点 v,满足每一个节点 $\forall v \in V(G_k), k \leqslant core(v) \leqslant \deg(v,G_k)$.

∴节点 v 的核数 $core(v)$的范围为$[k,\deg(v,G_k)]$.

∵定理 1 中,节点 v 至少有 k 个邻居点在 k-core 的子图 G_k 中.

∴节点 v 的这 k 个邻居节点的核数范围为

$$\{u \in nbr(v) \mid core(u) \geqslant k\}.$$

∵节点 v 的核数 $core(v)$取决于在 G_k 中邻居点的个数,即

$$core(v)=|V_k|.$$

∴其邻居点个数取 $\left|\{u \in nbr(v) \mid core(u) \geqslant k\} \geqslant k\right|$, 即 $coe(v)=k$ 的条件满足

$$\left|\{u \in nbr(v) \mid core(u) \geqslant k\}\right| \geqslant k.$$

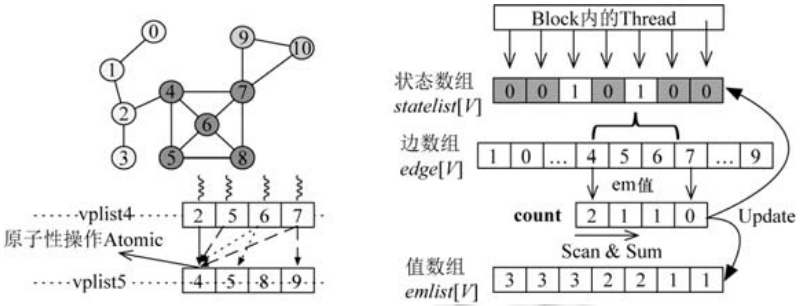
考虑节点 v 的核数 $core(v)$的上限满足上述条件取最大的 k 值. □

根据定理 2 的特性和公式(1),我们能够更高效地对复杂网络的 core 分解在 GPU 上的实现进行优化.如图 3(b)所示,ESCore 策略的基本思想为:按节点为中心进行状态更新处理,思路如下所述,复杂网络图中的每个节点

根据自己的度数值进行初始化各自的核数.同时,每个节点也会根据其邻居点所同步的核数,然后利用所接收的更新通知来重新计算自己的核数;进一步地,一旦节点的核数更新,便会立即重新通知其邻居点集,直到图中不存在有节点更新为止.同时,根据定理 2,值得注意的是,对节点 u 的核数的确定.

1) 处理过程中中间结果总是大于或者等于节点 u 的最终核数;

2) 每个节点每轮计算后所更新的核数一定是递减的.



(a) 图样例与 RemoveList 策略中 vplist 的原子操作 (b) Estimating 策略执行流程

Fig.3 Parallel execution flow of ESCore core decomposition and atomic operation overheads in RemoveList

图 3 ESCore 策略的执行流程与 RLCore 策略的原子操作分析示例

算法 2 中的 ESKernel 主要阐述了基于定理 2 的 ESCore 并行计算策略.算法的输入为复杂网络的图 G,输出为每个节点更新的核数.中间变量需要创建记录点集的当前 core 核数的结果集数组 emlist.另外,考虑到在递归迭代过程中,若所有的节点都参与计算,则会带来大量的无用计算量,通过 statelist 数组标记上一轮活动的节点集,而对不活动的节点集不需要进行任何计算,从而降低了大量无用计算.给定节点 v_i 通过判断其活动状态决定是否参与计算(第 8 行);算法初始化 *COUNT* 数组记录从 1 到 v_i 的当前核数($[1,core(v_i)_{cur}]$)范围内,所包含的对应核数的 v_i 邻居节点的个数(第 7 行),通过对比当前 $core(v_i)_{cur}$ 与 v_i 各邻居点的当前核数,取较小值赋值 count 数组(第 11 行~第 14 行).根据 $1\leqslant i\leqslant core(v_i)_{cur}$ 范围的 $COUNT[i]$数组,进一步得到 v_i 邻居点中核数大于等于 K 的节点个数,记为 SUM(第 15 行~第 19 行),即得到满足条件 $\{u\in nbr(v)\mid core(u)\geqslant k\}$ 的节点总个数;最后,根据定理 2 中的公式(1),判断条件 $SUM\geqslant K$,可得到最大的满足条件的 K 值.根据 K 值与 $core(v_i)_{cur}$ 取较小值得到更新的节点 v_i 的当前核数,并标记节点 v_i 已更新,下一轮作为活动节点(第 20 行~第 23 行).算法 2 的迭代计算直到复杂网络中的所有节点均没有更新状态为未活动为止,即 statelist 数组全部标记为 false 状态.

算法 2. ESCore Kernel(Locality 属性的点集状态更新).

输入:数据图 graph,点集的 core 评估值数组 emlist,点集状态位 statelist;

输出:点集的 core 评估值数据 emlist 更新值.

方法://ESCore

(1) Tid=blockIdx.x*blockDim.x+threadIdx.x;//GPU 线程号

(2) **IF** Tid>=total count of vertices in Graph

(3) THEN **return**;

(4) Vid=Ti*d*; //GPU 线程<=>Vid 节点

(5) **IF statelist**[**Vid**]==**false** //Vid 节点的状态

(6) **THEN** return;

(7) Set $\mathbf{Core_{cur}}$ value to emlist[vid];

(8) Setup **COUNT**[$\mathbf{Core_{cur}}$] array to record the number of neighbors related to core value;

(9) Set StartID value to the first edge index of Vid vertex invoking Graph.firstedge_index(Vid);

(10) Set EndID value to the last edge index of Vid vertex invoking Graph.lastedge_index(Vid);

(11) **FOR EACH** vertex ID in StartID to EndID range //对 Vid 的邻居点集遍历
(12) Set DestID value to the destination vertex invoking Graph.edge_dest(ID);
(13) Set **POS** to compared $\mathbf{Core_{cur}}$ with **emlist[DestID] invoking min**($\mathbf{Core_{cur}}$,**emlist[DestID]**);
(14) **COUNT**[**POS**]=**COUNT**[**POS**]+1;
(15) Setup **SUM** value to 0 for recoding summary value of vertices count;
(16) **For K** in $\mathbf{Core_{cur}}$ to 2 and K--
(17) **SUM=SUM+COUNT**[**K**];
(18) **IF SUM>=K**
(19) **THEN** break;
(20) **IF K<**$\mathbf{Core_{cur}}$
(21) **THEN**
(22) **emlist**[**Vid**]=**K**;
(23) **statelist**[**Vid**]=**true**;

3.3 策略对比与讨论

1. RLCore 与 ESCore 策略的复杂度对比

从时间复杂度上来看,RLCore 策略与 ESCore 策略的算法的基本步骤均根据节点 v 对其邻居节点进行遍历,复杂度均为 $O(\deg(v))$.其中,RLCore 需要进一步删除节点集标记 rmlist 和点集的读数组 vplist 进行扫描,即为 $O(2n)$;ESCore 需要进一步根据各节点的邻居点的核层数与满足条件的邻居节点数进行对比,算法复杂度记为 $O(k_{max})$,进一步地有定义 3 中所述 $k_{max}\leqslant \deg_{max}$.因此,给定迭代次数为 τ,RLCore 策略的总时间复杂度为 $O(\tau\cdot(m+2n))$;ESCore 策略的总时间复杂度均为 $O(\tau\cdot(m+n))$.

从空间复杂度来看,RLCore 策略需要创建若干中间变量,包括节点集的度数组 vplist、删除节点集标记 rmlist(长度 rm_cnt)、节点集状态位 flaglist,内存占用定界为 $O(n)$.同样,ESCore 策略需要对节点集创建状态数组 statelist、评估值数组 emlist 和节点数量记录数组 count,内存占用定界为 $O(n)$,因此 RLCore 与 ESCore 策略的总空间复杂度为 $O(n)$.

2. GPU 下并行处理对比

在本文工作中,我们进一步利用 GPU 的内存层级对所设计的策略进行分析对比,主要考虑如下两个方面.

(1) 线程的高并发性;

(2) 数据在 GPU 中内存的存储分布.

线程的高并发性决定了 GPU 下的算法并行度,能够直接影响算法执行性能:从 RLCore 的执行角度来看,算法本身的时间复杂度和空间复杂度能够达到最优化,然而我们发现,RLCore 中实现的并行策略利用大量的原子操作进行数组的更新(如图 3(a)所示),会导致在大规模并行的情况下同步开销增大.尤其是 RLCore 在处理以异构性强的复杂网络结构时,多个线程同时对同一个节点进行写操作时,为了保证一致性必须要对节点的更新写入进行原子性操作,这样势必会带来大量的同步开销.相比来看,ESCore 策略利用了复杂网络的局部性定理,对各邻居节点执行只读操作,能够大幅降低同步开销,从而达到 GPU 上高并发执行算法的优化.GPU 具有大量的全局内存(global memory)供各 block 之间的全局数据交换,而各 block 拥有独立的较小的、传输数据较快的共享内存(shared memory)和 Cache.我们对两个 core 分解策略的数据在 GPU 内存中存储的分布进行分析,见表 1.对比来看,RLCore 和 ESCore 算法所需要的共享内存均为 $O(n)$,网络的图结构数据缓存于全局内存中.

4 实 验

为验证本文提出的两类基于 GPU 的复杂网络 core 分解策略,与相关工作的不同并行模式在计算效率和可扩展性上的差异,我们进行了一系列的验证实验,采用了不同规模的开源的真实复杂网络结构图数据和人工仿真数据进行了测试实验,度量了不同并行模式下 RLCore 和 ESCore 并行复杂网络 core 分解策略的有效性、高

性能和可扩展性等特征.从实验数据可见,对比 CPU 下的相关算法工作,基于 GPU 的 RLCore 和 ESCore 复杂网络 core 分解策略能够在性能上达到至少一个数量级的提升,验证了本文所提策略的有效性.

4.1 实验平台与Benchmark

将本文提出的复杂网络 core 分解策略在配置 NVIDIA GTX 980(16 个 Maxwell 流处理器,128 Cores/MP, 4GB GDDR5)的工作站服务器上完成测试,服务器中主存和 GPU 访存对比配置参数的相关数值见表 1.服务器 Host 端配置有 2 个 10 核的 Intel Xeon(R) E5-2650 v3 主频为 2.30GHz 的 CPU 和 64GB 大小的 DDR4 内存,磁盘采用单块 512GB 的 SSD 存储盘.服务器操作系统采用 Ubuntu 16.04(内核版本 v4.4.0-38),配置版本 v7.5 的 CUDA/C++开发环境.所有的程序编译采用-O3 标志,配置目标 GPU 硬件的 streaming 多处理器生成器.

本文实验采用的测试数据集可见表 2,包含有 4 个开源的真实复杂网络结构图数据(http://snap.stanford.edu/data/index.html)和 2 个人工仿真生成数据集.采用的实验数据集均具有不同规模大小及不同结构分布.其中,AuthorsDBLP 和 Livejournal 数据集为复杂社交网络数据集,边表示为用户间的关联关系(合作者和朋友).Webbase 为网页链接关系数据集,Road-CA 为表示加州(Califormia)地区的复杂路网数据集.另外两个 RMAT 数据集,采用 Powerlaw 分布的 RMAT 人工生成数据图结构进行自动生成,沿用通用的 Graph500[18]标准测试集的配置,图数据分布参数 a=0.57,b=c=0.19.根据算法统计,我们对各个测试数据集的最大度数、最大核数与平均核数进行分析.表 2 所示各数据集具有相差较大的异构性与特性分布,能够较好地验证本文工作的有效性.

Table 2 Evaluation datasets

表 2 测试数据集

复杂网络数据集	节点集大小|V|	边集大小|E|	最大度数($\deg_{max}$)	最大核数(k_{max})	平均核数(k_{avg})	预处理耗时(s)
AuthorsDBLP	299 067	977 676	115	112	2.81	4.85
Webbase	1 000 005	3 105 536	4 700	494	2.7	10.52
Road-CA	1 965 206	5 533 214	12	5	2.32	4.21
Livejournal	4 847 571	68 993 773	20 293	9l80	8.54	147.23
RMAT23V67E	8 388 608	67 108 864	133 570	415	8.60	56.18
RMAT24V134E	16 777 216	134 217 728	205 772	531	8.99	141.96

4.2 实验设计

为了充分验证基于 GPU 平台的复杂网络 core 分解的两个策略的性能,实验验证部分重点从如下 4 个方面进行了本文所提的 GPU 下的复杂网络 core 分解的验证.

1) 相关工作性能对比:对部署的 GPU 环境下的 RLCore 和 ESCore 策略的整体性能进行评估.我们选择对比了两个应用最为广泛且最新的复杂网络 core 分解算法:BZ 算法和 ParK 算法,具体的配置如下.

- BZ 算法[3]:线性算法复杂度 BZ 算法,主要用于串行处理.配置 BZ 算法执行为 CPU 单线程.
- ParK 算法[5]:多核 CPU 并行最优性能.实验配置为 2、4、8 和 16 线程多核并行处理环境.

2) Kernel 执行效率对比:重点分析 RLCore 和 ESCore 的 GPU kernel 并行执行效率及占整体开销比例.

3) 图数据的吞吐性能对比:着重分析 RLCore 和 ESCore 在 GPU 加速下的以图结构数据形式进行处理的边吞吐率(traversed edges per second,简称 TEPS)的对比.

4) 迭代收敛率分析:考虑到 EMCore 收敛率对性能的影响,进一步分析 ESCore 迭代轮节点值的收敛率.

4.3 实验结果及分析

4.3.1 与相关工作对比结果

为了验证 RLCore 和 ESCore 并行策略的性能,我们对配置 RLCore 和 ESCore 并行策略采用单 GPU 配置.为了保证实验的有效性和公平性,性能实验排除了数据预处理与加载到内存(host memory)的时间开销,并对每组数据集重复 10 次实验结果取平均值.另外,我们进一步分析了 RLCore 和 ESCore 策略在 GPU 上执行的两个数据传输阶段(HostToDevice、DeviceToHost)和执行阶段(kernel execution)的时间开销,并分别进行度量.

在性能评估实验中,我们统计了 BZ 算法、ParK(2 线程)、ParK(4 线程)、ParK(8 线程)、ParK(16 线程)以及

RLCore 和 ESCore 策略的并行执行的总体运行时间.如表 3 所示,我们列出了各个不同配置的算法的实验结果.表 3 中,执行时间单位为 s.其中,最优化采用红色加粗进行标记,MT 表示多线程的数目配置,BZ 算法采用单线程执行所需的执行时间,包含多轮计算迭代中各个操作所消耗的总时间的平均值.同时,我们对各个并行执行策略的最佳执行进行了红色加粗标记.从结果可以看出,在所采用的 6 个图数据集中,基于 GPU 平台下执行的 RLCore 和 ESCore 策略能够达到最优化性能效果,并且加速明显,其中,ESCore 在 4 个数据集上的性能表现最优,RLCore 在 Road 和 RMAT23V67E 两个数据集上的性能表现最优,对上亿规模关联结构的网络 RMAT24V134E 能够在 16.3s 之内执行完成 core 分解,理论上可以极大地提高复杂网络 core 分解的执行效率.由表 3 所列数据说明,在各类不同特性分布的真实图基准数据集上,基于 GPU 平台下构建的 RLCore 和 ESCore 算法运行时间随图规模的扩大呈近似线性变化,这也与本文第 3.3 节中对算法所做出的时间复杂度讨论的结论相吻合.

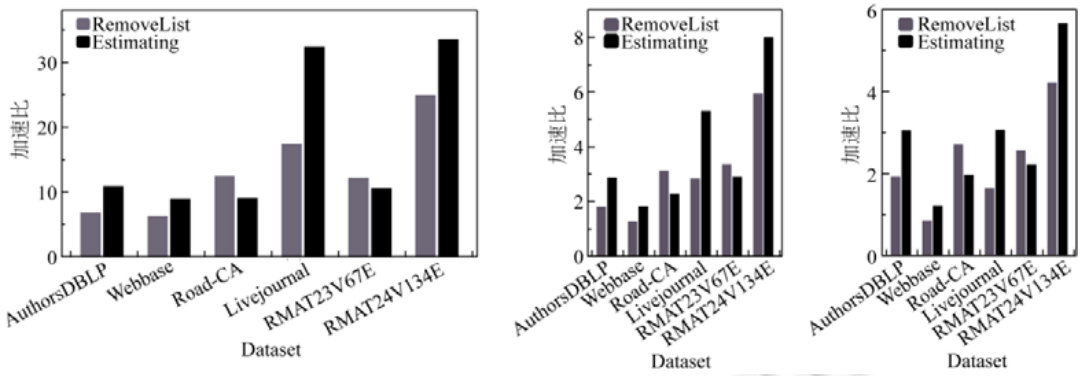
Table 3 Runtime (in seconds without preprocessing) comparison over various datasets

表 3 RLCore 和 ESCore 策略与相关工作的执行性能(不包含数据预处理与加载内存阶段)对比

数据集	BZ	ParK(2 MT)	ParK(4 MT)	ParK(8 MT)	ParK(16 MT)	RLCore	ESCore
AuthorsDBLP	1.85	0.61	0.64	0.48	0.52	0.27	0.17
Webbase	32.17	19.41	12.24	6.55	4.38	5.13	3.6
Road-CA	1.35	0.95	0.58	0.34	0.29	0.11	0.15
Livejournal	491.0	256.2	140.1	80.1	46.3	28.2	15.1
RMAT23V67E	197.0	119.9	73.5	54.2	41.4	16.18	18.6
RMAT24V134E	548.6	275.4	174.5	130.6	92.6	21.9	16.3

进一步分析,RLCore 和 ESCore 算法相比 BZ 算法在性能上有极大的提高,主要原因在于 BZ 算法在可扩展性上存在设计缺陷,其性能明显受约束于串行执行能力,如图 1(b)所示.尽管 BZ 的时间复杂度为线性,然而,若扩展至多线程高并发环境,其每一步均需要所创建的 3 个数组(节点的度数组、core 值数组以及位数组)为各线程所共享,并且 3 个数组的对应位置的节点值在访问过程中被同时更新,因此,大量额外的内存地址空间加锁的同步开销导致 BZ 算法不适用于并行环境.相比来看,RLCore 和 ESCore 算法策略受益于 GPU 多线程加速,分别从图遍历和局部性上对 core 分解进行了并行过程优化,显著降低了多线程内存地址访问一致性开销,增强了算法的可扩展性.此外,与 ParK 算法相比,RLCore 策略的实现思路有相似之处,均采用了多线程对图中各节点先根据其度数进行层级(level)划分,然后逐层同步子集节点的 core 值,两个时间复杂度均为 $O(m+n)$,ParK 和 RLCore 也会在子层级内的节点处理过程中产生大量原子性同步开销(atomic);相比而言,RLCore 利用 flaglist 对各节点状态位进行标记尽量降低了每轮迭代中的已处理节点;ESCore 相比 ParK 而言,利用 core 分解的局部性定理从邻居节点汇聚计算后更新,降低了对同一节点更新操作的同步开销,更适用于 GPU 高并行的计算环境.实验数据见表 3,进一步验证了本文所设计的算法 RLCore 和 ESCore 的性能优势,不仅能够利用 GPU 高并行环境提高复杂网络 core 分解的执行效率,而且算法在高并行环境下的可扩展性良好.

根据表 3 中的数据,我们进一步得到 RLCore 和 ESCore 策略与 BZ 和 ParK 算法所花费总时间的加速比值.首先,对于 BZ 算法,我们从图 4(a)中可以看出本文提出的基于 GPU 的 RLCore 和 ESCore 策略能够达到加速比 8.8X-33.6X.随着图结构的数据节点和关联关系量的增大,RLCore 和 ESCore 策略能够表现出更好的性能加速比,例如在 RMAT24V134E 数据集上,ESCore 策略表现得最优,能够达到 33.6X 的性能加速比.其次,相比于多核 CPU 并行的 ParK 算法,RLCore 和 ESCore 策略同样能够达到可观的加速比,我们对 8 线程和 16 线程下的 ParK 算法执行结果进行了策略的加速比分析,如图 4(b)和图 4(c)所示.两个策略能够对比 8 线程下的 ParK 算法达到 2X-8X 的加速比,对比 16 线程下的 ParK 算法达到 2X-5.6X 的加速比,从性能对比的效果可以看出,利用 GPU 的加速的复杂网络 core 分解能够大幅度优于多核 CPU 下的性能.



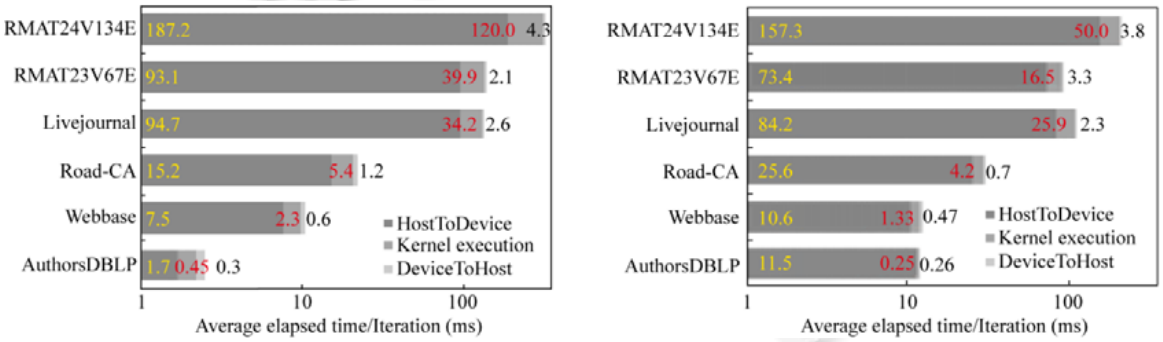
(a) RLCore 与 ESCore 策略对比 BZ 算法的加速比 (b) 对比 ParK 的 8 线程 (c) 对比 ParK 的 16 线程

Fig.4 The speedup ratio of RLCore and ESCore vs. BZ and ParK algorithms

图 4 RLCore 和 ESCore 策略与相关工作的性能加速比

4.3.2 Kernel 执行效率对比

为了分析两个策略在 GPU 上的 kernel 执行性能,我们对 RLCore 和 ESCore 策略在 GPU 上的执行分布进行了度量.图 5 列出了两个策略在 GPU 上的一轮迭代计算所进行的操作的分布情况,主要度量了 HostToDevice 和 DeviceToHost 两个数据传输操作的统计用时和 Kernel Execution 的执行用时.从图中可见,RLCore 和 ESCore 并行策略的大量开销在 HostToDevice 的数据传输开销上,而 Kernel Execution 的执行开销占了很小的比例,其中 RLCore 的 Kernel 执行占比在 18%~38.5%之间,ESCore 策略的 kernel 执行占比在 3%~23.8%之间,且 ESCore 在 kernel 执行上的表现明显优于 RLCore 策略.结果说明,GPU 下两个策略的 kernel 函数执行高效,能够极大地提高计算效率,core 分解的执行性能得到了大幅提升.



(a) RLCore 并行策略执行时间分布 (b) ESCore 并行策略执行时间分布

Fig.5 The execution time distribution of GPU operations for RLCore and ESCore

图 5 RLCore 和 ESCore 在 GPU 上并行执行的各操作执行时间分布

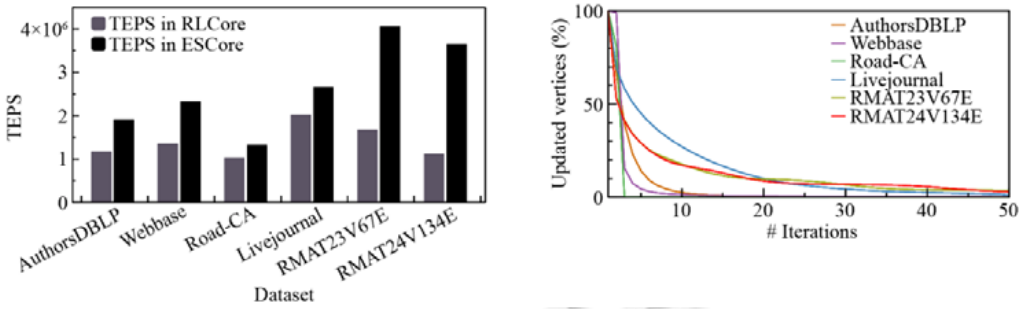
4.3.3 图数据的吞吐性能对比

为了量化 GPU 下 core 分解的图数据访问性能,我们对 RLCore 和 ESCore 并行策略每一轮的迭代的已访问边的效率(traversed edges per second,简称 TEPS)指标作了对比,从图 6(a)中可以看出,RLCore 的 TEPS 指标在 5 个网络图数据上表现为 102 万条/s~217 万条/s;ESCore 的 TEPS 指标表现为 131 万条/s~406 万条/s.因此,ESCore 的 TEPS 指标相比 RLCore 策略要优化 1.3-2.4X.随着网络图数据量的增大,ESCore 的 TEPS 吞吐性能指标表现得更为优化,因此说明,利用 GPU 加速的 RLCore 和 ESCore 策略能够大幅度提升算法的执行效率.

4.3.4 迭代收敛率对比

为了验证 ECore 策略算法的迭代收敛率的影响,我们对节点的更新状态进行了分析(如图所 6 示),并统计了

每轮迭代后更新节点集大小的占比.其中,考虑到 ESCore 策略从第 1 轮迭代开始在节点加入到 emlist 初始化,后续轮迭代不断更新点的值,因此每一轮迭代的更新节点集个数呈现不断降低的趋势.从图 6 可以看出,各个不同分布和特性的网络图数据在 core 分解过程中所更新节点集个数占比分布也不均衡,例如,Road-CA 路网的数据集在进行 4 轮迭代后基本上所有节点的 core 已确定,后续轮迭代不会有更新节点;而 Livejournal 社交网络图的数据集的节点更新频率趋于平缓,一般超过 50 轮迭代后才能达到 1%的节点未更新,趋于收敛.



(a) RLCore 与 ESCore 并行策略边吞吐率(TEPS)对比 (b) ESCore 并行策略执行时间分布

Fig.6 TEPS comparison and updated vertices ratio in iterations of ESCore core decomposition

图 6 TEPS 指标对比与 ESCore 复杂网络分解中每轮迭代更新的节点数占比情况

综合以上结果可以得到:本文所提出的基于 GPU 的 RLCore 和 ESCore 策略能够为复杂网络分解提供高性能的并行处理,能够支持大规模社交网络、万维网、道路交通网等各类型复杂网络的高效分析.相比于现有网络分解策略和算法,RLCore 和 ESCore 策略利用 GPU 的并行环境可达到数十倍的性能提升.同时,本文所提 RLCore 和 ESCore 策略 GPU kernel 函数执行高效,且 ESCore 策略的边吞吐性能最高达到了 406 万条/s.

5 结束语

本文针对应用广泛的复杂网络的 core 分解的执行性能问题,提出了面向更为高效的 GPU 平台下的复杂网络 core 分解的优化理论推导和策略的设计与实现.首先,根据复杂网络的 core 分解推导提出了 RLCore 策略利用 GPU 高并发性能优势,对网络自底向上的节点进行遍历处理,从外至内逐层进行 core 分解.其次,本文进一步基于 core 分解局部性的定理,提出了适用于多线程高并发处理的 ESCore 策略,对各节点根据邻居节点的当前值进行更新,从而使节点的 core 值确定流程能够做到分离迭代计算,有效降低了在高并行 GPU 计算环境下的同步操作开销.实验测试采用 6 组具有不同规模大小及不同结构分布的复杂网络结构进行验证,与最为广泛的 BZ 算法和多核 CPU 上优化的最优性能的 ParK 算法进行性能对比,多角度验证了本文工作的性能和优势.下一步工作,我们将集中精力对复杂网络的 core 分解进行高可扩展化优化,利用 GPU 架构的分布式计算机集群环境实现 core 分解.

References:

[1] Shin K, Eliassi-Rad T, Faloutsos C. CoreScope: Graph mining using *k*-core analysis-patterns, anomalies and algorithms. In: Proc. of the ICDM. 2016.

[2] Han ZM, Chen Y, Liu W, Yuan BH, Li MQ, Duan DG. Research on node influence analysis in social networks. Ruan Jian Xue Bao/Journal of Software, 2017,28(1):84–104 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5115.htm [doi: 10.13328/j.cnki.jos.005115]

[3] Batagelj V, Zaversnik M. Fast algorithms for determining (generalized) core groups in social networks. Advances in Data Analysis and Classification, 2011,5(2):129–145. [doi:10.1007/s11634-010-0079-y]

[4] Cheng J, Ke Y, Chu S, *et al*. Efficient core decomposition in massive networks. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2011. 51–62.

[5] Dasari NS, Desh R, Zubair M. ParK: An efficient algorithm for *k*-core decomposition on multicore processors. In: Proc. of the 2014 IEEE Int'l Conf. on Big Data (Big Data). IEEE, 2014. 9–16.

[6] Montresor A, De Pellegrini F, Miorandi D. Distributed *k*-core decomposition. IEEE Trans. on Parallel and Distributed Systems, 2013,24(2):288–300.

[7] Zhong J, He B. Medusa: Simplified graph processing on GPUs. IEEE Trans. on Parallel and Distributed Systems, 2014,25(6): 1543–1552.

[8] Khorasani F, Vora K, Gupta R, *et al*. CuSha: Vertex-centric graph processing on GPUs. In: Proc. of the 23rd Iint'l Symp. on High-performance Parallel and Distributed Computing. ACM, 2014. 239–252.

[9] Wang Y, Davidson A, Pan Y, *et al*. Gunrock: A high-performance graph processing library on the GPU. In: Proc. of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2016. 11:1–11:12.

[10] Sengupta D, Song SL, Agarwal K, *et al*. GraphReduce: Processing large-scale graphs on accelerator-based systems. In: Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. ACM, 2015. 28.

[11] Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal. ACM SIGPLAN Notices, 2012,47(8):117–128.

[12] You Y, Bader D, Dehnavi MM. Designing a heuristic cross-architecture combination for breadth-first search. In: Proc. of the 43rd Int'l Conf. on Parallel Processing (ICPP). IEEE, 2014. 70–79.

[13] Zhang H, Zhang L, Wu Y. Large-scale graph processing on multi-GPU platforms. Journal of Computer Research and Development, 2018,55(2):273–288 (in Chinese with English abstract).

[14] Seidman SB. Network structure and minimum degree. Social Networks, 1983,5(3):269–287.

[15] Khaouid W, Barsky M, Srinivasan V, et al. K-core decomposition of large networks on a single PC. Proc. of the VLDB Endowment, 2015,9(1):13–23.

[16] OBrien MP, Sullivan BD. Locally estimating core numbers. In: Proc. of the 2014 IEEE Int'l Conf. on Data Mining (ICDM). IEEE, 2014. 460–469.

[17] Malewicz G, Austern MH, Bik AJC, et al. Pregel: A system for large-scale graph processing. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data. ACM, 2010. 135–146.

[18] Graph500. http://www.graph500.org/

附中文参考文献:

[2] 韩忠明,陈炎,刘雯,原碧鸿,李梦琪,段大高.社会网络节点影响力分析研究.软件学报,2017,28(1):84−104. http://www.jos.org.cn/1000-9825/5115.htm [doi: 10.13328/j.cnki.jos.005115]

[13] 张珩,张立波,武延军.基于 Multi-GPU 平台的大规模图数据处理.计算机研究与发展,2018,55(2):273–288.

张珩(1990－),男,湖北天门人,博士,CCF 学生会员,主要研究领域为分布式与并行计算,大数据处理,操作系统.

武延军(1979－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为操作系统.

崔强(1985－),男,博士,主要研究领域为数据挖掘,推荐算法,众包测试.

赵琛(1967－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为编程语言,编译技术.

侯朋朋(1985－),男,在读博士生,主要研究领域为操作系统,推荐系统.