计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2018.20170697
55(2):273-288,2018

基于 Multi-GPU 平台的大规模图数据处理

张 珩[1,2] 张立波[1] 武延军[1]
[1](中国科学院软件研究所 北京 100190)
[2](中国科学院大学 北京 100049)
(zhangheng@nfs.iscas.ac.cn)

Large-Scale Graph Processing on Multi-GPU Platforms

Zhang Heng[1,2], Zhang Libo[1], and WuYanjun[1]
[1](*Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract GPU-based node has emerged as a promising direction toward efficient large-scale graph processing, which is relied on the high computational power and scalable caching mechanisms of GPUs. Out-of-core graphs are the graphs that exceed main and GPU-resident memory capacity. To handle them, most existing systems using GPUs employ compact partitions of fix-sized ordered edge sets (i.e., shards) for the data movement and computation. However, when scaling to platforms with multiple GPUs, these systems have a high demand of interconnect (PCI-E) bandwidth. They suffer from GPU underutilization and represent scalability and performance bottlenecks. This paper presents GFlow, an efficient and scalable graph processing system to handle out-of-core graphs on multi-GPU nodes. In GFlow, we propose a novel 2-level streaming windows method, which stores graph's attribute data consecutively in shared memory of multi-GPUs, and then streams graph's topology data (shards) to GPUs. With the novel 2-level streaming windows, GFlow streams shards dynamically from SSDs to GPU devices' memories via PCI-E fabric and applies on-the-fiy updates while processing graphs, thus reducing the amount of data movement required for computation. The detailed evaluations demonstrate that GFlow significantly outperforms most other competing out-of-core systems for a wide variety of graphs and algorithms under multi-GPUs environment, i.e., yields average speedups of 25.6X and 20.3X over CPU-based GraphChi and X-Stream respectively, and 1.3~2.5X speedup against GPU-based GraphReduce (single-GPU). Meanwhile, GFlow represents excellent scalability as we increase the number of GPUs in the node.

Key words large scalegraph; multi-GPU; graph shard; dual streaming windows; data movement

摘 要 在 GPU 高性能节点上构建高效的大规模图数据的算法和系统已经日益成为研究热点,以 GPU 协处理器为计算核心不仅能够提供大规模线程的并行环境,也能提供高吞吐的内存和缓存访问机制.随着图的规模增大,相对大小局限的 GPU 的设备访存空间逐渐不能满足缓存整个图数据的应用需求,也催生了大量以单节点上外存 I/O 优化(out-of-core graph)为主要研究方向的大规模图数据处理系统.为了应对这一瓶颈,现有的算法和系统研究采用对图切分的压缩数据形式(即 shards)用以数据传输和

收稿日期:2017-09-18;**修回日期**:2017-11-03
基金项目:中国科学院战略性先导科技专项(XDA06010600)
This work was supported by the Strategy Priority Research Program of Chinese Academy of Sciences (XDA06010600).
通信作者:张立波(zsmj@hotmail.com)

迭代计算.然而,这类研究扩展到 Multi-GPU 平台上往往性能的局限性表现在对 PCI-E 带宽的高依赖性,同时也由于 Multi-GPU 上任务负载不均衡而缺乏一定的可扩展性.为了应对上述挑战,提出并设计了基于 Multi-GPU 平台的支持高效、可扩展的大规模图数据处理系统 GFlow.GFlow 提出了全新的适用于 Multi-GPU 下的图数据 Grid 切分策略和双层滑动窗口算法,在将图的属性数据(点的状态集合、点/边权重值)缓存于各 GPU 设备之后,顺序加载图的拓扑结构数据(点/边集合)值各 GPU 中.通过双层滑动窗口,GFlow 动态地加载数据分块从 SSD 存储至 GPU 设备内存,并顺序化聚合并应用处理过程中各 GPU 所生成的 Updates.通过在 9 个现实图数据集上的实验结果可以看出,GFlow 在 Multi-GPU 平台下相比其他支持外存图(out-of-core graph)处理的相关系统性能表现更为优异,对比 CPU 下的 GraphChi 和 X-Stream 分别提升 25.6X 和 20.3X,对比 GPU 下支持外存图数据处理的 GraphReduce 系统单 GPU 提升 1.3～2.5X.同时 GFlow 可扩展性在 Multi-GPU 上也表现良好.

关键词 大规模图数据;multi-GPU;图分块;双层滑动窗口;数据传输

中图法分类号 TP316.4

随着单指令集多数据流(single instruction multiple data, SIMD)并行架构的流行,采用 Multi-GPU 架构的服务器节点来支持高性能计算已成为趋势热点.这类服务器节点通常由若干 Host 处理器(CPUs)和多个 GPU 设备组成,各设备间通过通信总线互联(PCI-E 总线或 NVLink 传输线),如图 1 所示.这样的架构相比单颗 GPU 服务器提供了更大规模的协处理器并行硬件环境和内存传输带宽,从而带来更高效的大规模数据聚合处理能力.

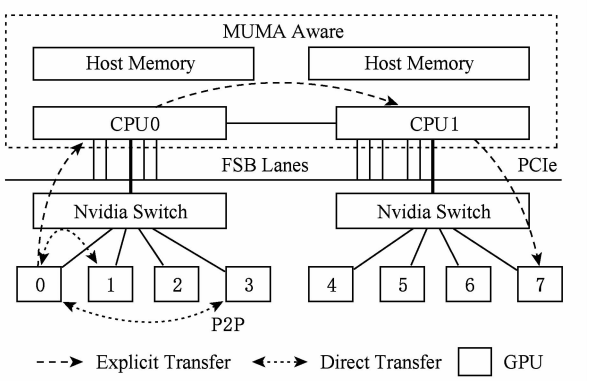

Fig. 1 The architecture of Multi-GPU platforms

图 1 Multi-GPU 平台的硬件架构

近年来采用 GPU 服务器平台对大规模图数据进行处理已经日益成为研究热点[1-2],提出了大量的算法优化和在 GPU 服务器上的图数据处理系统设计,例如 CuSha[3],MapGraph[4] 和 GunRock[5] 等.根据所能支持图数据的处理规模,我们将这类 GPU 图数据处理系统分为了 2 个通用的类别:GPU 设备访存图处理和外存(out-of-core)图数据处理方案.其中,访存内图处理系统类研究工作集中于 GPU 线程的并行优化策略来提升存储在设备访存内的不规则的图数据结构的访问和计算性能,最大限度地挖掘 GPU 线程组的并行处理能力.另外,其他研究提出了在单节点上支持基于外存 I/O 的大规模图数据的处理,这类系统突破了 GPU 访存大小的限制,通过利用图数据切分策略,迭代式加载至 GPU 设备访存,完成处理后利用主存和持久化存储设备进行全局同步.GraphReduce[6] 是第 1 个提出基于外存 I/O 进行 GPU 下大规模图数据计算的系统,其对图数据采用混合型的稀疏矩阵压缩 CSR(compressed sparse row)和 CSC(compressed sparse column)表示方式以降低迭代过程中随机访问开销,进而对原图数据进行均衡切分后,逐个加载分块至设备内存处理.

然而,GraphReduce 只支持单个 GPU 下的并行处理,这类基于外存 I/O 的 GPU 下图数据处理的系统在扩展到 Multi-GPU 平台下仍然存在着挑战:1)如何进行均衡的图数据切分,以适用于 Multi-GPU 下各层级的内存存储;2)在多个 GPU 和 CPU 之间进行协同处理过程中,如何有效地利用局限的传输带宽(PCI-E)进行数据传输.

为了解决上述挑战,本文对基于 Multi-GPU 平台下的外存图数据并行处理系统提出了优化策略,在以存储顺序 I/O 优化前提下,将图数据的属性数据集合(节点状态数据集,点/边权重数据集)先缓存于各 GPU 的设备访存,后对图分块逐个顺序传输至各 GPU 处理后同步.本文进一步设计并实现了 GFlow,在 Multi-GPU 平台上支持高效、可扩展的大规模图数据处理系统.本文在 GFlow 系统中主要提出了 2 点优化策略:适用于 Multi-GPU 下的图数据 Grid 切分策略和双层滑动窗口算法.具体地,

GFlow 在对大规模图数据进行 Grid 切分后,逐层从磁盘存储加载至 GPU 设备内存,并针对同步过程中的随机内存访问开销设计了 Ring Buffer 数据结构用以提升对各 GPU 并行处理过程所生成的 Update 消息数据集聚合传输和节点更新.

本文主要的创新和贡献点有如下 3 点:

1) 提出了一种适用于 Multi-GPU 平台图数据 Grid 切分策略,在采用最大化存储顺序 I/O 的 Streaming 图切分的基础上,根据主存、各 GPU 设备存储的资源大小构建列分块(strip-shard)和格分块(grid-shard),优化 PCI-E 的 I/O 传输的顺序数据块传输和各 GPU 之间的负载均衡;

2) 设计了双层滑动窗口并行化数据传输和处理策略(2-level streaming window),动态地加载数据分块从 SSD 存储至 GPU 设备内存,并顺序化聚合并应用处理过程中各 GPU 所生成的 Updates;

3) 通过在 4 个图基准算法和 9 个真实图数据集上的实验验证,GFlow 性能表现对比 CPU 下的 GraphChi 和 X-Stream 分别提升 25.6X 和 20.3X,对比 GPU 下的 GraphReduce 单个 GPU 下提升 1.3～2.5X,且可扩展性达到 3.8～5.8X(6 个 GPU 配置).

1　相关工作

随着单指令流多数据流并行架构(SIMD)的流行,利用超大规模核处理器 GPU 处理单元(graphical processing unit)的高性能节点进行大规模图数据的处理越来越为研究者们所关注.现实的图数据具有规模巨大、增量迅速且数据表征差异多样化的特点,在利用 GPU 对大规模图数据进行处理过程中,需要考虑到各方面的并行计算因素,如图结构数据的切分与遍历策略、图数据异构多线程并发的负载均衡以及图数据的存储和 I/O 方式等.现阶段,这类研究工作主要集中于图数据处理的基本算法与系统类的研究.

在 GPU 加速的图算法优化中,主流高性能领域的标准测试集 Graph500[7] 采用广度优先遍历算法(breadth first search, BFS)对高性能计算节点和集群以及并行算法的性能与能耗进行评估,因此利用 GPU 对 BFS 算法的高并行加速的研究日益成为热点.其中,You 等人[8] 提出将自顶向下和自底向上的图遍历访问方法综合考量,根据遍历算法的收敛程度动态选择访问方法,优化 GPU 线程对异构图数据遍历.Merrill 等人[9] 提出针对 GPU 下 BFS 遍历的核心操作前缀求和(prefix sum)算法进行任务的细粒度化优化,降低多线程的访问冲突,其遍历性能达到单 NVidia Tesla K40 GPU 上 33 亿条边/秒的 TEPS 访问速度.Hong 等人[10] 提出了虚拟 Warp(GPU 的逻辑线程组并行执行单元)为中心编程的思想,对 GPU 的 BFS 等图算法并发执行过程中的负载不均衡和逻辑运算单元(arithmetic logic unit, ALU)负载过载问题进行优化.Liu 等人[11] 根据遍历过程中的活跃节点集 Frontier 对 GPU 下的 BFS 算法提出 3 点优化:对 GPU 执行线程流水线化降低多线程竞争、根据点的出度情况调整动态负载来达到并行化负载均衡、GPU 下的 BFS 遍历方向优化.本文所设计的 GFlow 大规模图数据处理方法,主要针对 Multi-GPU 平台的大规模图数据进行数据传输、并行计算方面优化.GFlow 在设计的过程中借鉴了上述的 GPU 下的算法优化策略,如虚拟 Warp 线程组调度计算以及图数据遍历动态策略等.

另一方面,现有的大量研究工作开始转向 GPU 平台下图处理的系统级设计[3-4,12-13],为图算法方便实现提供简化的编程接口 API,支持各类图算法的 GPU 并行执行.在这类系统性的工作中,Medusa 系统[12] 第 1 次提出并设计实现了用于 GPU 平台的图计算简化编程系统,其提出了在 CUDA/C++编程框架基础上简化图的 GPU 编程 API 接口,底层采用 BSP(bulk-synchronous parallel)并行模型对 GPU 的大规模线程并发执行细节隐藏,从而简化 GPU 下的图算法编程实现.CuSha 系统[3] 着重对高稀疏性自然图的切分及处理访问方式进行性能提升,在块切分基础上对图数据按照起点-终点索引、节点权重、边权重进行数据分块组织,构建连续窗口图处理达到动态的调整各处理间隔的大小.Gunrock[5] 进一步优化了 GPU 下的高性能图处理接口,重构图处理原语,提供 Advance-Filter-Computation 三个操作来分别进行节点访问、过滤和计算的处理.为了在有限的单 GPU 访存空间上支持更大规模的图数据的处理,GraphReduce[6] 第 1 次提出单 GPU 节点的外存下的图数据处理,在规模无法存储于 GPU 访存的大图数据采用固定大小切块切分,并在 GPU 设备访存与 Host 内存之间设计优化了异步传输与迭代计算.

本文工作在 GraphReduce 基础上进行设计和实现 GFlow 图数据处理系统,提升可扩展性利用 Multi-GPU 平台以支持基于外存 I/O 的大规模图

数据的并行计算模型,主要在 2 个方面设计了优化策略:图数据的切分分块以及各 GPU 在并行计算过程中的数据传输和结果集计算同步机制. 相对比上述相关的研究系统,本文所设计实现的 GFlow 系统充分利用 Multi-GPU 节点的大规模并行环境的计算资源进行算法并行处理,在有限带宽 PCI-E 总线资源下降低图数据的同步和传输开销. 同时,GFlow 沿用了传统的 GAS(Gather-Apply-Scatter)[14] 的图数据并行编程模型,并提供大规模图数据的切分、存储、传输和计算,支持各类图计算的算法的 GPU 实现.

2 基于 GPU 下外存 I/O 的图数据并行处理

在本节中,我们主要介绍针对面向单 GPU 下的图结构上的方法应用. 首先,对图数据在 GPU 下迭代处理过程中的表现形式进行了描述,主要分为 3 个数组部分进行存储,包括 CSR/CSC 稀疏矩阵压缩结构格式、应用定义数据以及节点状态数据. 其次,本文工作对单 GPU 下基于外存 I/O 优化的大规模图数据计算系统的执行流程进行了简要描述. 通过沿用传统的以点为中心的并行计算 GAS(gather-apply-scatter)模型[14],在对原始图数据切块后顺序逐一载入 GPU 设备访存处理后,进行节点更新结果集同步.

2.1 图数据在 GPU 环境下的数据结构形式

考虑到 GPU 设备访存的资源有限性,基于 GPU 的大规模图数据处理研究工作沿用以 CSR 或 CSC 矩阵压缩的表现形式对大规模图数据进行格式化,降低了稀疏性图数据的存储开销. 在图数据进行迭代化并行处理的过程中,主要分为如图 2 所示的 3 种类型数据:

1) 图数据的结构数据(topology data, TD). 以 CSR 压缩的格式数据由各点的出边的索引数组(OutEdgeIdxs)和出边的值数据(OutEdgeIndex)构成.

2) 应用数据(application data, AD). 存储图的各节点的当前值的数组,根据不同的算法和应用对数据定义.

3) 状态数据(state data, SD). 标记了图中各节点在当前迭代轮的状态(0 或 1);标记为 1 表示节点状态活跃参与下一个迭代轮计算,标记为 0 则不参与计算. 状态位的标记能够有效降低图数据并行化处理的无用计算.

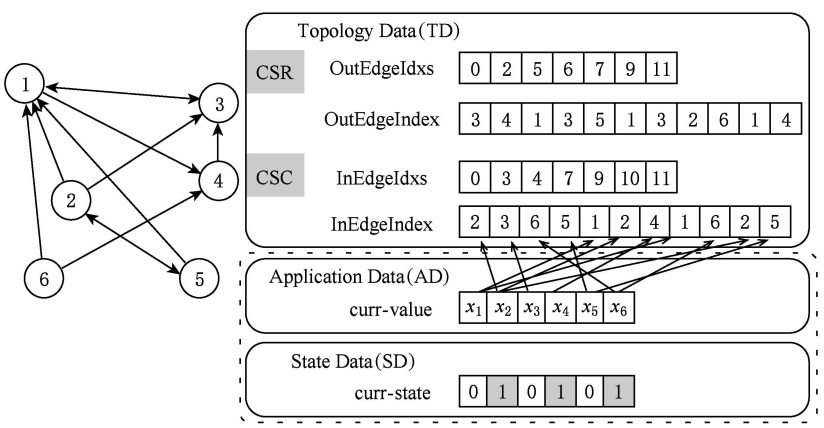

Fig. 2 Graph representation of CSR/CSC based compression

图 2 以 CSR/CSC 格式为例的图数据表现形式

2.2 单 GPU 外存计算的图处理系统执行流程

在第 1 节中,现有的 GPU 下的大规模图数据处理系统主要分为 2 类:访存内(in-memory)计算系统和外存(out-of-core)计算系统. 随着图数据规模的增大,基于 GPU 访存内计算系统的执行受 GPU 设备访存大小的限制,而导致无法应对更大规模的图数据的并行处理. GraphReduce[6] 系统第 1 次设计并支持了基于外存的单 GPU 下大规模图数据的并行处理. 本文所设计的 GFlow 沿用 GraphReduce 的设计原则,进一步优化提出 Multi-GPU 下的图数据 I/O 优化策略方案. 本节对单 GPU 下支持外存图数据处理的执行流程进行了简要的归纳和总结,如图 3 所示. 通过采用分布式图处理引擎 PowerGraph[14] 所设计的点为中心的 GAS 并行处理模型,GPU 下外存图数据处理主要分为 4 个主要处理阶段,包括初始化数据加载、Gather 阶段、Apply 阶段和 Scatter 阶段.

1) 第 1 阶段(初始化数据加载,图 3 中 1a 和 1b 阶段). 原始图转换为以 CSR 和 CSC 混合表示的图

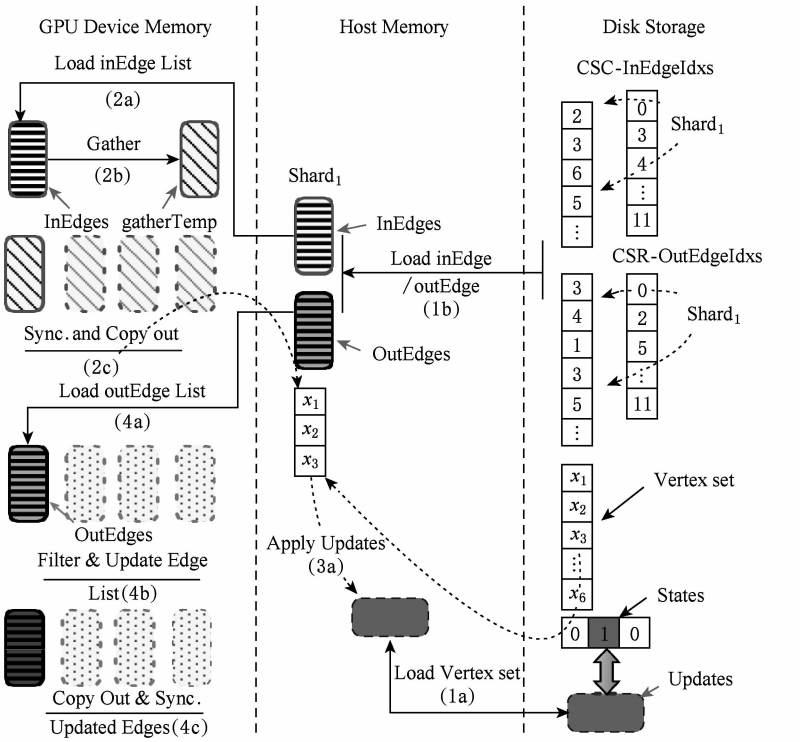

Fig. 3　Execution flow for large graph in single GPU

图 3　单 GPU 下大规模图数据处理流程

模型,并将边集切分为包含相等边数的分块 Shard(入边+出边数量),进一步将各边集分块 Shard 以及对应的 AD、SD 数据块加载至主存空间;

2) 第 2 阶段(Gather,图 3 中 2a、2b 和 2c 阶段).逐个将包含 in-edge(入边)分块 Shard 加载至 GPU 访存后,对当前的对应活跃点集遍历获取对应的节点值,对所加载边集执行用户自定义 gather.在对各个 in-edge 分块处理完毕同步后,获取本地化的 gather 操作结果更新(节点权值/边值)集合;并进一步拷贝更新集合值内存;

3) 第 3 阶段(Apply,图 3 中 3a 阶段).对所聚合的局部 updates 集合执行 apply 操作,对磁盘存储的全局 updates 数组的对应项进行更新并标记相应节点状态位.

4) 第 4 阶段(Scatter,图 3 中 4a,4b 和 4c 阶段).在逐一拷贝 out-edge(出边)分块 Shard 以及所对应更新的节点值集合至 GPU 访存后,执行 scatter 用户自定义函数对边集及其终点状态进行更新.

其中,值得注意的是,在做边集分块 Shard 的过程中,分块大小的确定以各分块能够完整存储于 GPU 的设备访存为准.另外,为了避免不必要的 memcpy 操作和 kernel 初始化开销,我们采用了对活动点集进行动态记录的策略,即首先调用 CUDA 指令 *any*()探测在第 1 阶段计算过程中是否存在活动节点需要传输,然后在 warp 线程组内和组外逐步规约(binary reduction)来获取活跃节点集合.

3　支持 Multi-GPU 平台的大规模图数据处理

大规模图数据的类型多样化,包括社交网络数据、路网数据、协作者网络数据以及大量的 Internet 网页链接数据等.这类现实中的复杂网络所构建的图数据类型特征各不相同,我们采用实验中的数据对这类图数据进行了特征分析.其中 coAuthor① 为论文合作者网络,webbase② 为网页链接数据,road-CA③ 为路网数据.从图 4(a)～(c)中,各图数据在进行 BFS 广度优先遍历过程中各轮迭代过程中参与计算的活跃点集数量不尽相同,同时迭代轮次数差距也较大,如 coAuthors 和 webbase 在 5～6 轮迭代之后完成,而 road-CA 路网半径较大导致需要至少

① https://sparse.tamu.edu/DIMACS10/coAuthorDBLP

② https://sparse.tamu.edu/Williams/webbase-1M

③ https://sparse.tamu.edu/SNAP/roadNet-CA

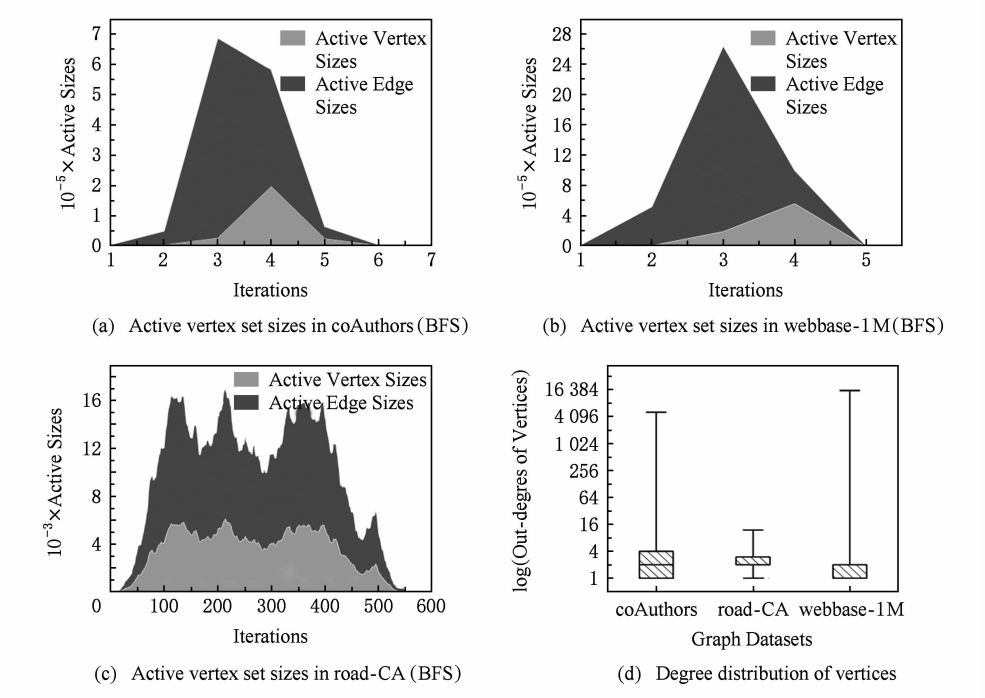

Fig. 4 Different representations for three graphs

图 4 3 个图数据(coAuthor,webbase 和 road-CA)的不同特征

520 轮迭代计算.进一步,我们对 3 个图的各节点出度情况进行统计,从图 4(d)中可见,road-CA 路网数据相比来看出度更为集中,大量节点的出度在 2～4 之间,这也反映了道路交通网络的节点度均匀特征.

在构建图数据处理系统过程中,为了支持多样化的图数据类型和各类图数据算法,在设计上需要着重考虑如下 2 个方面的因素:大规模图数据的均匀切分和图数据的并行处理.首先,为了 Multi-GPU 环境下的各设备的计算负载均衡,对多样化大规模图数据的均衡切分能够有效降低图数据并行处理过程中 straggler 开销问题;其次,在并行处理方面,GPU 的外存数据传输开销往往占用大量运行时时间,因此在并行化的数据传输和计算过程中,采取重叠式轮转调度和异步式的数据传输能够大幅降低外存数据的传输与同步开销.

本节对 GFlow 支持 Multi-GPU 平台的大规模图数据的处理机制进行了详细描述,主要在图数据集均衡切分和基于窗口的异步并行处理 2 个方面提出了优化与改进.

3.1 GFlow 系统设计思想和执行流程

在 GPU 服务器节点上,GPU 的各层级内存由用于 Warp 单元间和线程块间的数据通信的共享内存(shared memory,L1 cache)和用于所有 SMX (streaming multiprocessor)的全局内存(global memory)以及少量的 L2 Cache 缓存构成,而 CPU 端以主存为主.在进行 Multi-GPU 平台下的图数据处理系统构建时规划了各层内存的数据结构存储如下所示(以 Nvidia GTX 980 型号 GPU 为例):

1) 共享内存(shard memory).48 KB,用于存储各节点的状态数组和各 SMX 本地节点分块集合的应用执行结果集;

2) 全局内存(global memory).4 GB,用于存储分块图结构数据集(Shard),包括点、边数据,以及部分用作缓存(Buffer)存储临时结果集合.

3) Host 主存(main memory).64 GB,用于存储部分图结构分块数据集合,以及对应的 AD 应用数据和 SD 节点状态数据集.

GFlow 系统沿用 GAS 并行模型设计的思路,提供用户 Gather/Apply/Scatter 接口函数以实现并行图算法.算法 1～3 分别给出了 PageRank 算法采用 Gather/Apply/Scatter 函数的各自实现,其中,预定义边数据结构体 *ED*(PageRank 的边无权值)和点数据结构体 *VD* 包含节点权值 *rank* 和出边数值 *numOutEdges*.GFlow 系统总体的流程分为4 个阶段:输入(Input)、核心执行(Core)、结果集合并(Merge)以及输出(Output).图 5 给出了 GFlow 的整体执行流程示意图.

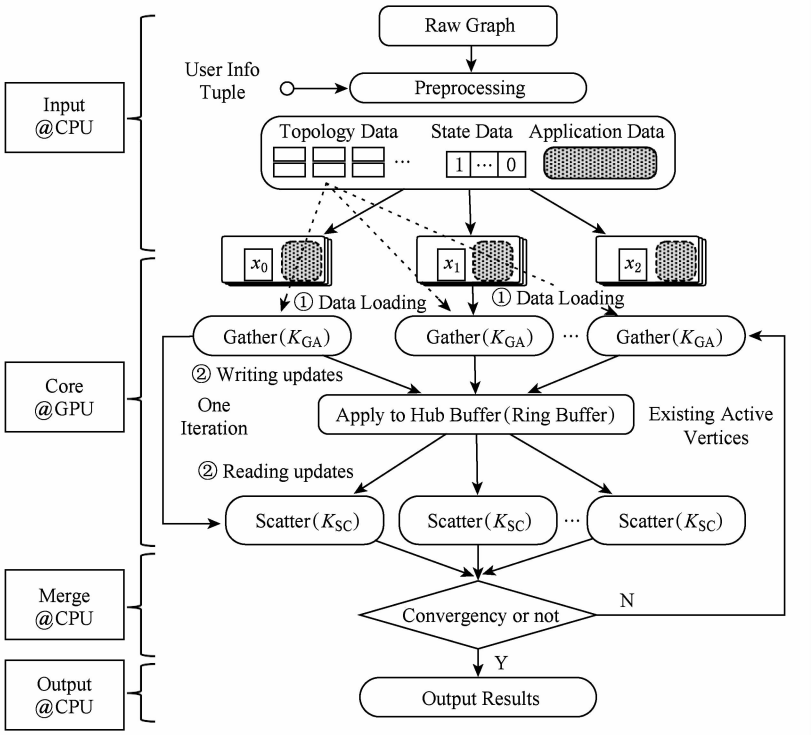

Fig. 5　Execution flow in GFlow of multi-GPU

图 5　Multi-GPU 下 GFlow 的执行流程示意图

为了解决大规模图数据集无法完整存储于 GPU 设备内存的问题,本文采用了一种基于 Grid 切分策略对 2.2 节所提及的图分块(Shard)进行优化改进,主要设计目的在于以最大化 PCI-E 的 I/O 传输的顺序数据块传输,优化各 GPU 之间的负载均衡同时保证了图数据的预处理的高效.进一步,将各分块对应的节点状态位数组信息 SD 和应用数据 AD 共同进行封装,以整体数据块形式来进行数据传输拷贝(memcpy).所设计的图数据 Grid 切分策略详细流程如 3.2 节所述.

算法 1. GFlow 中 PageRank 的 Gather 函数 K_{GA}.

输入:边的起点数据 D_u、终点数据 D_v、边数据 $D_{(u,v)}$;

输出:边的终点数据的累加更新.

① 根据边的起点数据权值计算其对终点的贡献均值 $\overline{rank}(D_u)=\dfrac{rank(D_u)}{numOutEdges(D_u)}$;

② return $rank(D_v)+\overline{rank}(D_u)$.

算法 2. GFlow 中 PageRank 的 Apply 函数 K_{AP}.

输入:当前节点数据 D_v、累加聚合结果 $gatherResult$;

输出:更新后 D_v 的权值.

① $\alpha=0.85$f; /* 定义阻尼系数 */

② $new\_pr=1-\alpha+\alpha\times gatherResult$;

③ 更新 D_v 的权值 $rank=new\_pr$.

算法 3. GFlow 中 PageRank 的 Scatter 函数 K_{SC}.

输入:节点更新数据 D_v、边数据 $D_{(u,v)}$;

输出:节点 D_v 的活跃状态.

① $THRESHOLD=0.01$f;

　　/* 定义节点活跃状态阈值 */

② 计算 D_v 当前权值与原权值的绝对差值 $\Delta rank$;

③ IF $\Delta rank<THRESHOLD$

④　　return false;

⑤ ELSE

⑥　　return true.

在 CPU 进行预处理过程之前,各个 GPU 设备各自分配初始化内存 Buffer 用于保存 TD Buffer、SD Buffer、当前和生成的 AD Buffer.接收 Shard 块传输之后,各 GPU 设备并行化执行 GAS 并行计算模型.其中,K_{GA}函数和 K_{SC}函数分别为用户自定义实现的 GPU 的 Kernel 函数 Gather 和 Scatter.在 Gather 执行完成后,GFlow 分配全局的 Hub Buffer 保存各 GPU 节点生成的更新集合(updates),在计算完成对应的节点或边的更新权值之后组装成数据块分配至对应的 GPU 设备.此过程中,为了降低

数据 I/O 传输和更新的同步开销,GFlow 着重对 2 个阶段的数据传输进行了优化,如图 5 所示的数据加载阶段(data loading)和各 GPU 之间并行执行所生成的 Updates 的读写操作(writing/reading updates). GFlow 提出并设计了双层数据读写滑动窗口对数据传输和并行计算阶段进行阶段,具体的实现细节介绍见 3.3 节.

在各 GPU Kernel 执行完毕之后,根据所生成的 Updates 集合对图数据的节点 SD 状态集标记,在 Host Memory 中更新合并,整体流程的迭代计算的收敛判断是否还存在活跃点集或者达到指定的迭代次数.

3.2 适用于 Multi-GPU 下的图数据 Grid 切分策略

为了满足 Multi-GPU 下的各种类型不同的图数据的处理需求,GFlow 提出并设计了一种适配多层级内存资源的 Grid 切分策略. 现有 CPU 下外存图切分策略的研究[15-18]利用磁盘的顺序读取的高效性原理,在引入一定图切分的预处理开销下采取更为均衡和高效的图数据顺序化分块存储. GFlow 借鉴其中 Grid 切分的思想[19],在对设备的内存层级资源探测后,构建适用于各层级内存存储的数据结构. 具体地,GFlow 构建了 2 层数据分块:列分块(strip-shard)和格分块(grid-shard).

1) 列分块. 存储于 host memory 大小,采用 Streaming 切分策略顺序读取所有节点的出边集合,按照各节点的 ID 范围 S_n 来分配,切分原则为各 strip-shard 的范围 S 内的边集大小相当.

2) 格分块. 用于 GPU 的设备内存计算,切分准则采用各个 grid-shard 所包含的边个数相当. 在对 strip-shard 进行二次切分为多个 grid-shard,在综合考虑各个节点的活跃状态,各 grid-shard 在各迭代轮之间进行合理地调整,以对各 GPU 的负载进行均衡分布.

图 6 给出了简要的 GFlow 的均衡 grid 子图分

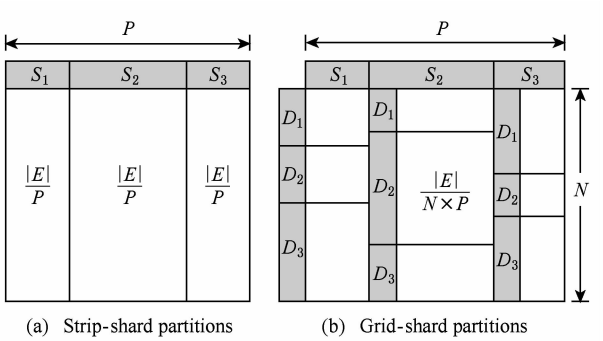

Fig. 6 Balanced grid-based graph partition in GFlow

图 6 GFlow 的均衡 Grid 图切分

块的切分流程. 在确定图的起点集合划分为 P 个分块,在各个分块中边的分配数量为均衡的 $\frac{|E|}{P}$,即保证各 strip-shard 的出边子集相等(图 6(a)所示);进而在对 strip-shard 的切分过程中,选择以 GPU 的数量 N 为基准进一步划分子边集,即 grid-shard 的终点子区间个数为 N.

具体地,对边集中的各条边分别按照其起点和终点的范围进行组织,这样能够在引入一定预处理开销($O(|E|)$)的前提下获取合理负载均衡的分块. GFlow 设计的切分算法在顺序化磁盘读取边之后即可将各边分配切分完成,因此其相对比其他外存系统(GraphReduce)能够大幅度降低引入的预处理开销. 我们对基于 Grid 的图分块视图采用了 CSR/CSC 格式的节点的 2 个边索引数组 in-EdgeIdxs 和 out-EdgeIdxs 进行图数据的切分策略实现. 根据各图的 CSR 存储,GFlow 通过 *out-EdgeIdxs* 数组获取各节点的度数组($d_1, d_2, \cdots, d_n$). 进而通过获取节点上 host memory, share memory 和 global memory 的资源大小(分别标记为 M_h, M_s, M_g),GFlow 根据各节点的 $ID(i)$ 和节点出度 d_i 来进行 strip-shard 和 grid-shard 的切分. GFlow 在满足各切分块的对应 AD 和 SD 数据和各 grid-shard 分块的边集能够存储入全局内存中,没有限制切分块的数量. 进一步,我们也采用了更为直接的策略对切分块的数量进行了定义:给定 GPU 设备数量为 N,主存、单个设备全局内存的大小分别为 M_h, M_g,以及单个节点、边的应用和状态数据的大小均为 U、节点 ID 大小 B 和点集和边集数量 $|V|, |E|$,则所切分的节点区间 S 的个数 P 为满足 strip-shard 节点区间和对应 grid-shard 边集合的存储条件的最小整数中的较大者:

$$P=\max\left\{\left\lfloor\frac{2(B+U)|E|}{NM_g}\right\rfloor, \left\lfloor\frac{2(B+U)|E|}{M_h}\right\rfloor\right\}.$$

进一步,为了降低各 shard 内节点的索引查询的开销,我们对各点集范围 S 内的节点采用连续存储,只保存了第 1 个节点的偏移量(offset),将节点的状态和属性数据进行连续存储. 这样有效地利用了 CSR/CSC 存储格式提升空间利用率.

不同于传统的 2-D 的图切分策略[19-20],GFlow 适用于 GPU 节点多层级内存存储,能够最大限度提高各层级内存的存储利用率和执行效率. 另外,图 7 给出了 grid 切分后顺序读写过程. 其中,在对同一 strip-shard 内的 grid-shard 执行过程中能够

对内部各分块的起点范围内的节点集 S_i 的各节点的状态和权值复用,而不需要频繁对节点子集在内存缓存和磁盘间的换入换出,降低了不必要的数据读写开销.

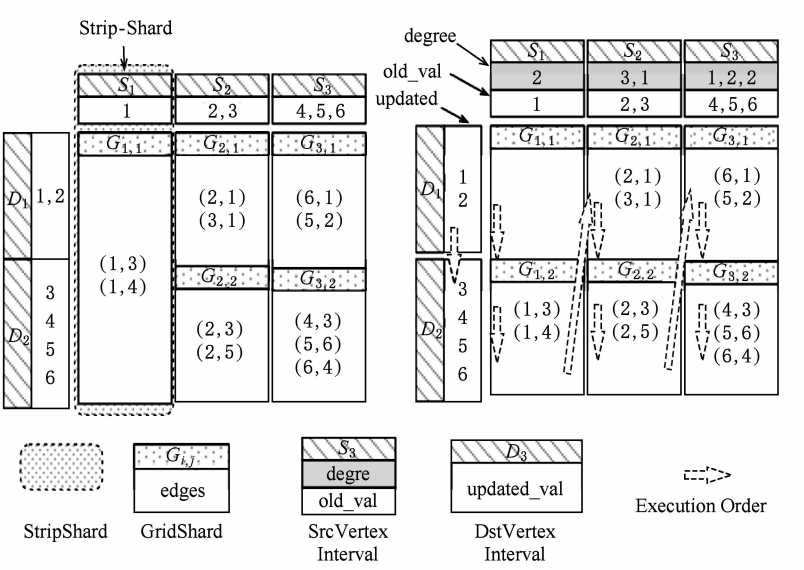

Fig. 7　Grid-based graph partition and streaming I/O operations in GFlow

图 7　GFlow 的 Grid 图切分及顺序读写图

3.3　基于双层滑动窗口的异步数据传输和计算

在 Multi-GPU 平台下,数据的传输在 Host 和各 GPU 之间可以通过 H2D,D2H 的 CUDA 接口定义进行数据拷贝(memcpy);而 GPU 与 GPU 设备之间的数据的交换和传输的方式能够通过显性传输定义(explicit transfer)和直接传输(GPUDirect)[21-23].为了进一步分析外存图计算系统在 Multi-GPU 平台下能够高效地进行数据传输,我们对 CUDA 编程接口的 2 个技术进行分析:1) 采用 cudaMemcpy 接口和 H2D/D2H 方式的显性数据传输定义;2) 由 GPU 硬件支持的统一虚拟地址技术(unified virtual addressing, UVA)所提供的 GPUDirect 直接数据传输.图 8 中显示采用显性数据传输定义接口能够先比 UVA 支持的 GPUDirect 策略能够提升 5～6 倍的数据传输带宽,同时 Pinned 分配的块内存传输相比页式内存更为高效.尽管 UVA 特性能够对 GPU 之间的数据传输的细节隐藏,但是其由于支持直接内存地址传输带来大量数据一致性访问所需加锁的开销.因此,我们在设计 GFlow 的数据传输策略时,尽量考虑采用 Pinned 后内存数据块的显性数据传输策略,这样也能在数据预取和 GPU 合并内存数据块并行处理方面取得性能优势.

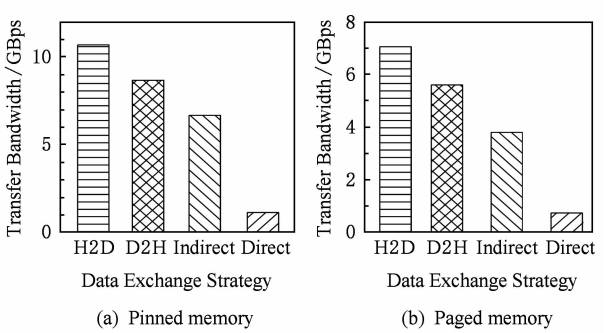

Fig. 8　Experimental analysis in data movement strategies supported in Multi-GPU platform

图 8　Multi-GPU 下数据传输策略分析

在数据传输和计算执行过程中,GFlow 将 grid-shard 数据块与相对应的属性数据(AD 和 SD)进行合并为结构化文件块,通过 PCI-E 总线分配至各 GPU 设备内存.考虑到 PCI-E 数据传输效率远低于 GPU 全局内存读写效率,因此本文提出并设计了基于顺序块 I/O 的异步双层滑动传输和计算窗口(2-level streaming windows),主要对 GFlow 执行过程中的 2 个方面的执行效率如图 5 所示的数据加载阶段(data loading)和各 GPU 之间并行执行所生成的中间结果集的读写操作(writing/reading updates)阶段优化设计.

首先,在①数据加载(data loading)阶段,为了对磁盘存储中的数据块组织,我们构建了 tile-window 对 strip-shard 进行管理.每一个 tile-window 的大小由 Host 主存大小决定,通常我们设定内存阈值 δ 来对总体主存的使用率进行设置(默认 $\delta=0.8$),因此在 tile-window 滑动数据加载和执行过程中,GFlow 按照 Z 折线顺序化(zigzag order)逐列读取每个 strip-shard.

其次,GFlow 在主存中构建 stream-window 的第 2 层文件读写窗口对 strip-shard 内的 Grid 结构进行分块索引获取. stream-window 窗口主要对 grid-shard 和对应的属性数据分块加载到 GPU 设备和管理每轮迭代计算过程各 GPU 中的中间结果集(on-the-fly). 具体地,stream-window 的窗口大小由多个 grid-shard 组成,各 GPU 内设备总内存所决定,并根据各 grid-shard 的节点状态数组(SD)调整所含大小边集. 进一步,GFlow 以 stream-window 为单位对各 GPU 所生成的本地更新值(AD)存储并进行同步至 Host 内存.

最后,在②中间结果集(updates)的读写操作阶段,各 GPU 并行地执行生成 updates 的分配和读写,考虑到各 GPU 所生成的 updates 消息的稀疏性,GFlow 采用 Hub Buffer 机制来对各 update 消息组进行缓存和同步. Hub Buffer 中,update 的消息数量相对巨大($O(|E|)$)且各 GPU 生成的每个部分在拷贝至 Hub Buffer 的数量大小不尽相同(如图 4 所示活跃点集所决定). 我们采用了一种固定大小、静态分配的数据结构进行 Hub Buffer 的构建,即 Ring Buffer. Ring Buffer 数据结构能够有效避免动态内存分配的开销并为在处理单个 stream-window 中数据块所生成的 update 消息提供高效的索引结构. 具体地,Ring Buffer 由一个大小为 η 的索引环数组(index ring array)和索引所指向的数据块数组(block array)组成,η 取决于 stream-windows 中 grid-shard 数量. 索引环数组的每个项由 3 个指针组成,分别指向 stream-window 中的 grid-shard 以及其所对应的保存在数据块数组(block array)中的 AD 和生成的 update 消息集合. 每个数据块数组(block array)采用固定大小页对齐的文件块组成(默认 64 MB),一旦一个 block 填满即可分配新的 block. 采用 Ring Buffer,数据通过 PCI-E 链路传输能够连续传输,大幅度提升了所生成的 update 的读写传递效率.

我们对本文策略在 GPU 环境下进行实现,使用 Nvidia GPU 的 CUDA C/C++环境进行算法实现,在实现过程中利用 CUDA Stream Object 特性对数据从 Host 到 Device 之间的传输和 Kernel Function 的执行流程上进行优化,尽量降低了数据传输所带来的开销,如图 9 所示流程. 进一步,GPU 内线程组(warp)的各线程根据各节点的状态进行执行,一旦 SD 获取的 vid 对应的状态为 inactive,该线程则不需要进行处理,继续获取 SD 中的 active 的节点进行下一步处理,从而有效降低了 GPU 的空闲率. 通过对 GPU 的优化实现,本文所实现的方法极大提高了大规模图数据的处理性能和高并发的可扩展性,能够支持大规模图结构数据集的处理.

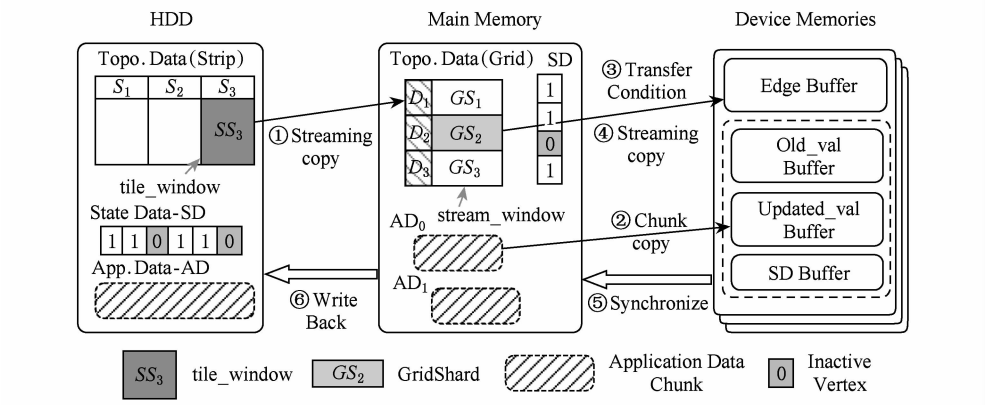

Fig. 9 Asynchronized data movement and processing phases in GFlow

图 9 GFlow 的异步读写与并行处理步骤

3.4 GFlow 实现细节

算法 4 给出了 GFlow 的并行化执行流程. GFlow 在将子图分块 strip-shard 的属性数据(点的状态数据 SD,点/边权重数据 AD)缓存于各 GPU 设备后,在启动 Stream 滑动窗口进行图的拓扑结构化数据(点/边集合)至各 GPU,进而执行用户 Kernel 函数,得到迭代计算结果. 具体地,初始化的 GPU 设备缓冲区 $OVBuf$, $UVBuf$, $SDBuf$ 分别保存原节点权值、更新节点权值以及节点状态. $tile\_window$ 载入 strip-shard 至主存中,并对标记为未执行的子图块初始化权值和状态集合,构建点与分块索引 $v2sMap$ 和 $s2vMap$(行④~⑩). 进而 $stream\_window$ 读取 N 个 grid-shard 载入各 GPU 访存,执行 K_{GA}, K_{AP}, K_{SC} 对边集处理得到节点的权值聚合后更新写入

$UVBuf$,状态写入 $SDBuf$(行⑭～㉖).最后对各 GPU 设备的缓存导出,并同步更新各节点的最终应用权值和状态值.

在该执行过程中,我们采用了 CTA(cooperative thread array)的线程组管理(Modern GPU 函数库提供)[10,21].通过借鉴 Virtual Warp[10]所设计的单个 warp,在加载完成 grid-shard 后,我们将利用 Share Memory 缓存对应的 SD 数据,单个 warp 中的各线程对单个节点所有出边进行同时处理.具体地,在对 warp 和内部线程的 offset 进行计算后,在遍历到需要处理的节点时即分配 warp 对该节点的出边进行线程处理,此外对各节点的聚合更新加入原子性操作(AtomicAdd),各线程处理完毕后采用 *__threadfence_block*()对同一 warp 内线程同步.

在数据传输过程中,我们利用 GPU CUDA Stream Object 的特性对数据传输和各 GPU 之间的计算的重叠以提高并行效率的细节实现.该特性提供了一个 GPU 的操作队列,使用 Stream Object 实现了任务级的并行,CUDA 的运行时库提供 GPU 在执行 Kernel 函数的同时,在 Host Memory 和 Device Memory 之间交换数据.在创建若干个 Stream 的对象管理 $GPU_{1,N}$ 的计算和 Shard 的传输后,传入各 Stream 对象至调用 Kernel 函数和 *cudaMemcpyAsync*.*Stream* 根据程序的任务执行情况提供运行时优化,来维护该 GPU 的操作队列的先后顺序.

算法 4. GFlow 异步 Multi-GPU 并行处理主流程.

输入:图数据 $G=(V,E)$;K_{GA},K_{SC},K_{AP}分别表示用户自定义的 Gather,Scatter 和 Apply 函数.

① 为设备 $GPU_{1,N}$ 全局内存中 device memory 中分别分配 $EDBuf$, $OVBuf$, $UVBuf$, $SDBuf$;
② 调用函数 *cudaMemcpyFromSymbol* 转换 K_{GA},K_{SC},K_{AP}三个 Kernel 函数标记;
③ while active vertices exist do
④　for *tile_window*=next strip-shard ID i from G
⑤　　if *tile_window* 标记为已处理 then
⑥　　根据起点状态集合 SD_i 过滤 *edges*;
⑦　　else
⑧　　获取起点范围 S_i 权值 AD_i、状态集合 SD_i;
⑨　　起点状态集合 SD_i 过滤 *edges*;
⑩　　end if
⑪　　Async-copy 权值 AD_i 和状态 SD_i 集合至各设备 $GPU_{1,N}$;
⑫　　*stream_window*=从 *tile_window* 读取 N 个 grid-shard 子图分块集合;
⑬　　对 *stream_window* 中的各 grid-shard 在索引环数组中构建 v2sMap,s2vMap 索引值;
⑭　　for each device id $j\in GPU_{1,N}$ do
⑮　　$G_{i,j}$=*stream_window* 中 *nextGrid* 子图;
⑯　　当前 $G_{i,j}$的 AD_i 初始化 $OVBuf$, $UVBuf$;
⑰　　Async-copy 分配 $G_{i,j}$ to $EDBuf$ in GPU_j;
⑱　　调用 K_{GA}对 $G_{i,j}$边集处理,更新 $UVBuf$;
⑲　　$nextGrid=\bigcup_{i=1}^{N} nextGrid_i$ in *stream_window*;
⑳　　GPU 同步 *ThreadSynchronization*;
㉑　　end for
㉒　　Async-copy 各 *GPU* 的 $UVBuf$ 至主存内的索引环数组对应的块数组中;
㉓　　v2sMap 调整 Shard 节点 AD_i 和 SD_i 索引;
㉔　　更新节点集合的权值集 Update 调用 K_{AP};
㉕　　调用 K_{SC}更新节点状态写回的 $SDBuf$;
㉖　　$GPU_{1,N}$同步 *cudaStreamSynchronization*;
㉗　　for each shard $s\in RingBuf$ do
㉘　　调用 s2vMap 写回全局 Update 的权值;
㉙　　end for
㉚　end for
㉛ end while

4　实验与结果

在本节中,我们使用本文构建的基于 Multi-GPU 平台的大规模图数据处理系统 GFlow,采用 9 个公开的现实数据集(如表 1 所示)进行了对比实验,并且通过性能和 GPU 上执行情况衡量了算法的高效性和可扩展性.

Table 1 The Evaluation Datasets

表 1 用于实验验证的网络数据集

Dataset		Vertices $\|V\|$	Edges $\|E\|$	Dataset Size *size*
GPU In-Memory	coAuthorsDBLP	299 067	977 676	45.5 MB
	belgium osm	1 441 295	1,549,970	98.5 MB
	kron-g500-logn20	1.05 million	44.6 million	1.8 GB
	amazon	403 394	3 387 388	182 MB
	road-CA	1 965 206	5 533 214	283 MB
	webbase-1M	1.0 million	3.1 million	167 MB
GPU Out-of-Core	kron-g500-logn21	2.1 million	91.0 million	5.6 GB
	uk-2002	18.5 million	298.1 million	16.1 GB
	twitter	41.7 million	1.4 billion	52.4 GB

实验环境采用 8 块 NVIDIA GTX 980 GPU 工作站上完成测试,服务器配置 2 颗 10 核的 Intel Xeon E5-2650 v3 的 CPU 和 64 GB 大小的 GDDR5 内存,存储为 512 GB PCI-E 的 SSD 硬盘 RAID-0 配置. 考虑到 X-Stream 和 GraphChi 采用 Direct IO 读写数据,在实验过程中,我们对读写过程中采用配置 DirectIO 避免内存页缓存. 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 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.

为了与现有的大规模图数据处理系统作对比,我们选取了 2 类系统:1)支持 CPU 下外存图计算的系统,其中最为广泛应用的 GraphChi[15](https://github.com/GraphChi/graphchi-cpp)和 X-Stream[17](https://github.com/epfl-labos/x-stream);2)支持 GPU 的大规模图数据计算的系统,此类系统中选择了最新的支持外存图数据计算的单 GPU 下的系统 GraphReduce[6]和支持 Multi-GPU 下内存内计算的图数据处理系统 WS-VR[22].

另外,在基准算法测试方面,我们采用了广泛认可的 PageRank[24]、广度优先遍历算法(BFS)、单源最短路径算法(single source shortest path, SSSP)以及联通子图算法(CC).

下面我们分 3 个部分进行实验的对比:

1) 对比常用的 CPU 下的外存计算系统 GraphChi 和 X-Stream 和支持单 GPU 下的外存图数据处理系统 GraphReduce;

2) 对比支持 Multi-GPU 平台的图数据处理系统 WS-VR;

3) 对 GFlow 所设计的策略在 Multi-GPU 下的可扩展性进行验证.

4.1 数据集

为了对 GFlow 在 Multi-GPU 平台上的性能与可扩展性的验证,本文在 9 个真实的网络数据集(数据集来源 http://snap.stanford.edu/data/)进行测试,其中,coAuthorsDBLP, belgium osm, kron-g500-logn20, amazon, road-CA 和 webbase-1M 用于测试小规模的内存内图数据的计算;其他 3 个图 kron-g500-logn21, uk-2002 和 twitter 用于测试外存存储图数据的计算. 这 9 个数据集合具备各自不同的特征,其中包括有社交网络数据集、路网数据集、商品关联数据集以及网页链接数据集. 例如,coAuthorsDBLP, twitter 表述了社交网络内的用户的交互行为(协作、相互关注等); webbase-1M, uk-2002 为网页链接数据集; amazon 代表了亚马逊内的商品之间的关联规则关系; road-CA 为路网数据集. 在第 3 节中我们对这些图数据集的多样性特征进行了分析,可以发现各类数据集的点边分布以及稀疏程度都存在特征性的差异. 下面我们利用这 9 个现实网络图数据集对 GFlow 进行性能的实验对比分析.

我们以 twitter 图数据在 8 GB 的 Host Memory、4 GB 设备访存的服务器上为例对 Shard 的切分配置参数进行说明如下. twitter 图数据(1.4 billion 边规模,节点和应用数据大小取 16 B)在 8 GB 可用主存下的分块取区间个数 $P=10$. 通过 8 个 GPU 设备的分块,总的 grid-shard 个数为 80. 设置的 *tile-window* 大小为阈值 $\delta \times 8\ \text{GB} = 6.4\ \text{GB}$, *stream-window* 为 global memory 总大小 32 GB. 在加载过程中,twitter 图的大小为 52.4 GB(CSR/CSC 格式),对子图分块的索引占用一定的空间,通过 *tile-window* 加载一个 5.2 GB 的 strip-shard 子图分块

到 Host Memory 后构建各索引(v2sMap 和 s2vMap)占用,切分为 0.65 GB 左右的 grid-shard.

4.2　实验和结果

在本节中,我们首先将 GFlow 与常用的 CPU 下的外存计算系统 GraphChi 和 X-Stream 的性能进行对比评估,其中 GraphChi 和 X-Stream 部署于多核 CPU 上进行实验,2 个系统采用 16 线程进行对比(X-Stream 支持的多线程配比为 2^n);GraphReduce 和本文所设计的 GFlow 系统部署于 GPU 平台. 为了验证的公平性,GraphReduce 只支持单个 GPU 的执行,我们在对比配置中将 GFlow 设置为单 GPU 下的运行状态.

1) 对比 X-Stream 和 GraphChi. 如图 10 所示,GFlow 相比 GraphChi 和 X-Stream 分别平均能够提升 10.3X 和 3.7X 的执行效率. 首先,对 BFS 和 SSSP 的图遍历算法(图 10(a)(d)),GFlow 相比其他系统的性能更优,最大的提升来自于 uk-2002 数据集,相比 GraphChi 和 X-Stream 在 SSSP 算法取得了 21X 和 20.3X 的加速比,在 BFS 算法取得 20.8X 和 7.5X 的加速比. 同时 GFlow 在 billion 边规模的 twitter 图数据上总体执行时间 85.2 s 和 35.2 s,相对比也能取得 2.2～8.0X 的执行效率提升. 这些性能提升的原因主要来自于 2 个方面:①GPU 的高并行运行时环境提供了 GFlow 在图数据处理上的性能优势;②通过利用重叠数据传输和计算开销,GFlow 充分得到了异步并行计算的性能优势. 在 PageRank 和 CC 算法上(图 10(b)(c)),GFlow 仍然能够得到一定的效率优势,例如,在 kron-g500-logn21 和 uk-2002 数据集上,GFlow 在 CC 算法上取得了 3.8 s 和 212.2 s 执行时间,对比 GraphChi 和 X-Stream 分别提升性能 2.1～25.6X 和 4.3～6.5X. 值得注意的是在 kron-g500-logn21 上的 PageRank 执行效率,GFlow 略逊于 X-Stream,其中原因在于 kron-g500-logn21 能够完整地存储于节点的主存中,X-Stream 能够充分利用本地性的数据的访问,而 GFlow 加载到 GPU 设备内存带来了大量的 PCI-E 的传输开销,导致性能略有所降低.

2) 对比 GraphReduce 系统. 从图 10 中结果可以看出,GFlow 在单个 GPU 的执行性能上能够大部分领先于 GraphReduce. 在大部分的基准测试中,

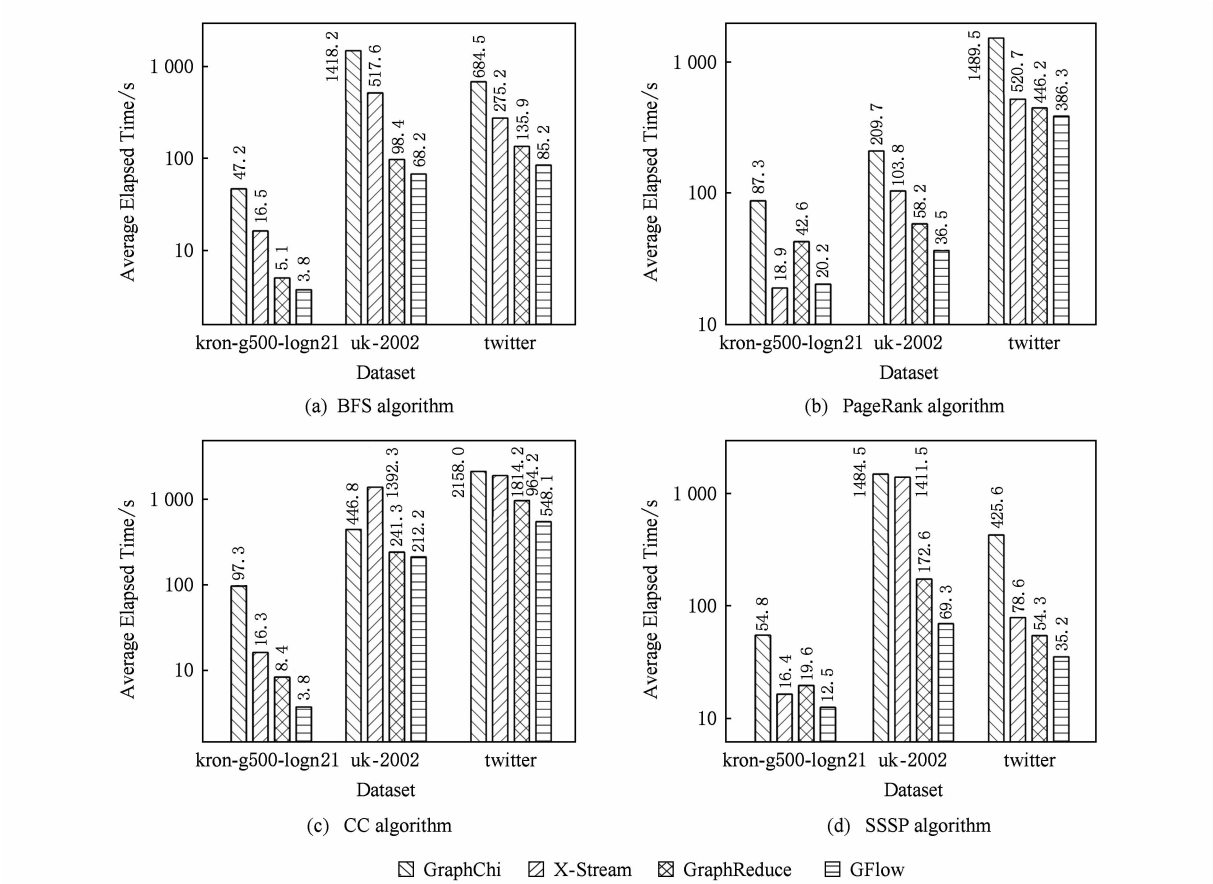

Fig. 10　The comparison results for elapsed time of out-of-core graph parallel-processing systems and GFlow

图 10　GFlow 与其他外存图数据计算系统的性能对比,限制节点内存为 8 GB

GFlow 能够对比 GraphReduce 得到 1.3～2.2X 的性能提升,所得到的最大提升性能来自于 kron-g500-logn21 数据集.这其中的性能优势原因主要来源于 GFlow 在 GPU 执行上的性能优化策略,例如,采用 Ring Buffer 来降低动态 Update 数据块的读写有效地提升 PCI-E 的吞吐;利用双层顺序读写窗口以加速数据加载和并行处理的策略等.

进一步,我们也将 GFlow 与 GPU 内存计算的图数据系统 WS-VR[22] 进行了性能对比.WS-VR 通过利用 GPU 内部 Warp 线程组的调度对 GPU 在 CSR 图数据上的执行效率得到了大幅提升,同时提出了节点集约减来提高 Multi-GPU 上的可扩展性和执行效率.从图 11 数据所示,我们分析了 GFlow 与 WS-VR 在 Multi-GPU 上的总体执行效率对比(WS-VR 配置为 Vertex Reduce 模式优化).我们可以看出,GFlow 的执行性能相对比 WS-VR 有一定的可对比性,尤其在扩展到采用多个 GPU 的平台下.例如,在 BFS 算法 belgium 图数据上,GFlow 执行 1.89 ms 和 1.67 ms 在 6 和 7 个 GPU 配置下,相比 WS-VR 达到 1.30～1.51X 的加速比.需要注意的是,在多个基准测试算法上(belgium BFS 和 road-CA SSSP),WS-VR 在 GPU 数量增大的情况下执行的性能反而降低,这也说明多个 GPU 所带来的数据传输和同步开销增大反而导致整体性能的降低.而 GFlow 相比来看能够从动态活跃节点集和异步数据传输的策略上得到可扩展性的提升.另外,相比处理这类内存内的数据集,GFlow 也能够从所设计的数据结构上得到数据传输的性能提升,从而降低了 PCI-E 的传输开销和达到多个 GPU 之间的负载均衡.

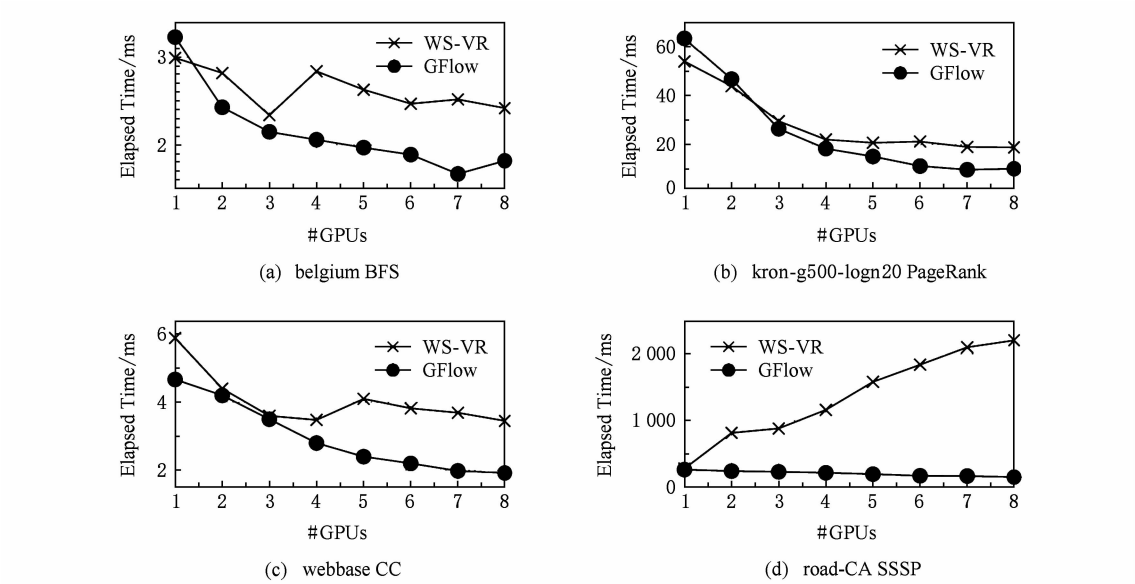

Fig. 11 Execution time of in-memory system WS-VR and GFlow

图 11 GFlow 与 GPU 图数据计算系统 WS-VR 在 Multi-GPU 上的执行时间对比

最后,我们对 GFlow 在 Multi-GPU 上的可扩展性进行了分析.对可扩展性的实验选择了 4 个图数据集,包括内存内的和外存存储数据集,以及 PageRank,BFS 和 SSSP 三个基准测试算法来验证 GFlow.从表 2 中数据可见,GFlow 的可扩展性在 1 个 GPU 到 6 个 GPU 上表现良好.首先,在 3 个外存存储的图数据集 kron-g500-logn21,uk-2002 和 twitter,随着 GPU 个数的加速,能够大幅降低整体的执行时间,提升了 4.95～5.6X 的性能.另外,在 2 个内存内图数据集 webbase,GFlow 仍然取得了一定可比性的性能提升,在 6 个 GPU 配置下相比提升了 3.8X.

综合以上结果可以得到:本文所提出和实现的 GFlow 系统能够很好地应用于 Multi-GPU 下的大规模图数据处理应用上.该系统对 GPU 下图数据的处理得到大幅度的性能提升,尤其在基于外存的大规模图数据的处理上.相对比现有的 CPU 上的外存计算系统 GraphChi 和 X-Stream,GFlow 利用 GPU 的高性能计算达到了数十倍的性能提升.同时,相对比 GPU 环境下的 GraphReduce 和 WS-VR,GFlow 也能表现出相当的加速比,同时在 Multi-GPU 平台下得到 3.8～5.8X(6 个 GPU 配置)的可扩展性性能提升.

Table 2　Speedup of GFlow on Different Graph Applications and Datasets

表 2　算法在 Multi-GPU 上的可扩展性性能对比

Algorithm	#GPU	webbase	kron-g500-logn21	uk-2002	twitter
PageRank	2 vs 1	1.43	1.59	1.64	1.52
	6 vs 1	2.74	3.30	2.98	2.86
BFS	2 vs 1	1.60	1.35	1.41	1.72
	6 vs 1	2.42	4.14	2.83	2.26
SSSP	2 vs 1	1.36	1.23	1.30	1.46
	6 vs 1	2.23	2.98	2.72	2.82

Note: That speedup ratio is measured on *N*-GPU vs 1-GPU.

5　总　　结

本文主要介绍了一个 Multi-GPU 平台下高可扩展的大规模图数据处理框架 GFlow,支持在有限的计算资源(内存、处理器)下对大规模图数据进行处理. GFlow 提出了适用于 Multi-GPU 平台多层级内存结构的 Grid 分块存储方式,将大规模图数据转换 strip-shard 和 grid-shard 分块,利用顺序数据块的并行读写提升数据传输性能. 同时,为了降低数据在 PCI-E 链路上传输和通信的开销,GFlow 设计并实现了双层滑动窗口读写和处理策略以最大化图分块的顺序数据传输,并采用 Ring Buffer 来合并各 GPU 所动态生成的 Update 和节点状态数据从而以聚合的文件块(block)形式提升消息数据的读写能力. 从实验验证中可以看出,GFlow 能够大幅度提升 GPU 平台下外存图数据的吞吐和执行性能,并在多个 GPU 下可扩展性良好.

参　考　文　献

[1] Cheng Xueqi, Jin Xiaolong, Wang Yuanzhuo, et al. Survey on big data system and analytic technology [J]. Journal of Software, 2014, 25(9): 1889-1908 (in Chinese)
(程学旗, 靳小龙, 王元卓, 等. 大数据系统和分析技术综述[J]. 软件学报, 2014, 25(9): 1889-1908)

[2] Yu Ge, Gu Yu, Bao Yubin, et al. Large scale graph data processing on cloud computing environments [J]. Chinese Journal of Computers, 2011, 34(10): 1753-1767 (in Chinese)
(于戈, 谷峪, 鲍玉斌, 等. 云计算环境下的大规模图数据处理技术[J]. 计算机学报, 2011, 34(10): 1753-1767)

[3] Khorasani F, Vora K, Gupta R, et al. Cusha: Vertex-centric graph processing on GPUs [C] //Proc of the 23rd Int Symp on High-Performance Parallel and Distributed Computing. New York: ACM, 2014: 239-252

[4] Fu Z, Personick M, Thompson B. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs [C] //Proc of Workshop on Graph Data Management Experiences and Systems. New York: ACM, 2014: 1-6

[5] Wang Y, Davidson A, Pan Y, et al. Gunrock: A high-performance graph processing library on the GPU [C] //Proc of the 21st ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2016: No. 11

[6] Sengupta D, Song S L, Agarwal K, et al. GraphReduce: Processing large-scale graphs on accelerator-based systems [C] //Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2015: No. 28

[7] Graph500. The Graph500 list [EB/OL]. [2017-11-03]. http://www.graph500.org/

[8] You Y, Bader D, Dehnavi M M. Designing a heuristic cross-architecture combination for breadth-first search [C]//Proc of the 43rd Int Conf on Parallel Processing (ICPP). Piscataway, NJ: IEEE, 2014: 70-79

[9] Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal [C] //Proc of the 17th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2012: 117-128

[10] Hong S, Kim S K, Oguntebi T, et al. Accelerating CUDA graph algorithms at maximum warp [C] //Proc of the 17th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2011: 267-276

[11] Liu H, Huang H H. Enterprise: Breadth-first graph traversal on GPUs [C]//Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2015: No. 68

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

[13] BenNun T, Sutton M, Pai S, et al. Groute: An asynchronous multi-GPU programming model for irregular computations [C] //Proc of the 22nd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New Yor: ACM, 2017: 235-248

[14] Gonzalez J E, Low Y, Gu H, et al. Powergraph: Distributed graph-parallel computation on natural graphs [C] //Proc of the 10th USENIX Symp on Operating Systems Design and Implementation. Berkey, CA: USENIX, 2012: 17-30

[15] Kyrola A, Blelloch G, Guestrin C. Graphchi: Large-scale graph computation on just a PC [C] //Proc of the 10th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX, 2012: 31-46

[16] Cheng J, Liu Q, Li Z, et al. Venus: Vertex-centric streamlined graph computation on a single PC [C] //Proc of the 31st IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2015: 1131-1142

[17] Roy A, Mihailovic I, Zwaenepoel W. X-stream: Edge-centric graph processing using streaming partitions [C] //Proc of the 24th ACM Symp on Operating Systems Principles. New York: ACM, 2013: 472-488

[18] Cheng L, Kotoulas S. Scale-out processing of large RDF datasets [J]. IEEE Trans on Big Data, 2015, 1(4): 138-150

[19] Zhu X, Han W, Chen W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning [C] //Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX, 2015: 375-386

[20] Chi Y, Dai G, Wang Y, et al. Nxgraph: An efficient graph processing system on a single machine [C] //Proc of the 32nd IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2016: 409-420

[21] Zhang Jun, He Yanxiang, Shen Fanfan, et al. Two-stage synchronization based thread block compaction scheduling method of GPGPU [J]. Journal of Computer Research and Development, 2016, 53(6): 1173-1185 (in Chinese)
(张军, 何炎祥, 沈凡凡, 等. 基于2阶段同步的GPGPU线程块压缩调度方法[J]. 计算机研究与发展, 2016, 53(6): 1173-1185)

[22] Khorasani F, Gupta R, Bhuyan L N. Scalable SIMD-efficient graph processing on GPUs [C] //Proc of 2015 Int Conf on Parallel Architecture and Compilation (PACT). Piscataway, NJ: IEEE, 2015: 39-50

[23] Faraji I, Mirsadeghi S H, Afsahi A. Topology-aware GPU selection on multi-GPU nodes [C] //Proc of 2016 IEEE Int Parallel and Distributed Processing Symp Workshops. Piscataway, NJ: IEEE, 2016: 712-720

[24] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: Bringing order to the Web [R]. Stanford, CA: Stanford InfoLab, 1999

Zhang Heng, born in 1990. PhD candidate in the Institute of Software, Chinese Academy of Sciences. His main research interests include distributed and parallel processing systems, operating system and large-scale graph analytics.

Zhang Libo, born in 1989. Received his PhD degree from the University of Chinese Academy of Sciences, Beijing, in 2017. Now assistant professor in the Institute of Software, Chinese Academy of Sciences. His main research interests include image processing, knowledge graph and deep learning (zsmj@hotmail.com).

Wu Yanjun, born in 1979. Received his PhD degree from the Institute of Software, Chinese Academy of Sciences, in 2006. Currently, professor at the Institute of Software, Chinese Academy of Sciences. His main research interests include operating system, big data platforms and artificial intelligence (yanjun@iscas.ac.cn).