中国科学 : 信息科学　2024 年　第 54 卷　第 6 期: 1369–1385
SCIENTIA SINICA Informationis
论文

一种冗余感知的高能效图计算加速器

姚鹏程[1,2,3,4,5], 廖小飞[1,2,3,4], 金海[1,2,3,4*], 周宇航[1,2,3,4], 徐鹏[5], 张伟[5], 曾圳[5], 潘晨高[1,2,3,4], 朱冰[1,2,3,4]

1. 华中科技大学大数据技术与系统国家地方联合工程研究中心, 武汉 430074
2. 华中科技大学服务计算技术与系统教育部重点实验室, 武汉 430074
3. 华中科技大学集群与网格计算湖北省重点实验室, 武汉 430074
4. 华中科技大学计算机科学与技术学院, 武汉 430074
5. 之江实验室, 杭州 311121
* 通信作者. E-mail: hjin@hust.edu.cn

收稿日期: 2023–12–08; 修回日期: 2024–03–04; 接受日期: 2024–03–12; 网络出版日期: 2024–06–13

国家重点研发计划 (批准号: 2023YFB4502300)、中国博士后科学基金 (批准号: BX20230333, 2023M743257, 2023TQ0328, 2023TQ0327) 和浙江省自然科学基金 (批准号: LY24F020014) 资助项目

摘要　图作为一种灵活表达对象之间关系的数据结构, 广泛地应用于各类重要的现实场景. 近年来, 随着性能提升速度放缓, 通用处理器逐渐无法满足图计算应用的需求, 并成为限制图计算发展的主要瓶颈. 因此, 面向图计算的领域专用加速器成为近年来的研究热点. 通过定制化的硬件设计, 图计算加速器可以在图计算应用中取得通用处理器数十倍的性能. 然而, 现有的图计算加速器在运行宽度优先算法时会频繁地重复访问幂律顶点的相关数据, 进而导致了严重的冗余访存问题. 在特定场景下, 现有的图计算加速器的性能甚至低于通用 CPU. 为了解决该问题, 本文提出一种冗余感知的高能效图计算加速器 JiFeng. 当幂律顶点完成迭代计算时, JiFeng 通过跳过剩余的相邻边大幅减少其被重复访问的次数. JiFeng 实现了一系列软硬件协同设计, 在保证负载均衡的同时提升硬件的执行效率. 为了验证 JiFeng 的有效性, 本文采用 FPGA 原型系统对相关设计进行性能评估. JiFeng 在典型的生成图和现实图上实现最高每秒遍历 4612 亿条边的性能和每秒每瓦特遍历 125 亿条边的能效比, 并在 2023 年 11 月的图计算超算排行榜 GreenGraph500 的小数据集榜单上取得第 2 名的成绩.

关键词　图计算, 加速器, 宽度优先搜索, 冗余访存, FPGA

1　引言

随着大数据应用的发展, 关联型数据呈现爆炸式增长. 图作为一种灵活表达关联关系的数据结构, 被广泛地应用于生物制药、社交网络和商品推荐等重要现实场景. 为了挖掘图数据中蕴含的信息, 图

引用格式: 姚鹏程, 廖小飞, 金海, 等. 一种冗余感知的高能效图计算加速器. 中国科学: 信息科学, 2024, 54: 1369–1385, doi: 10.1360/SSI-2023-0387
Yao P C, Liao X F, Jin H, et al. A redundancy-aware energy-efficient graph accelerator (in Chinese). Sci Sin Inform, 2024, 54: 1369–1385, doi: 10.1360/SSI-2023-0387

计算技术逐渐成为大数据领域的研究热点 [1,2]. 其中, 宽度优先遍历 (breadth-first search, BFS) 算法是分析图数据的核心技术之一, 在联通分量、子图扩展和图神经网络等众多图计算应用中扮演重要角色. 由于其普遍性与重要性, BFS 算法近年来受到研究人员的广泛关注 [3,4], 并被图计算超算排行榜 GreenGraph500 [5] 作为唯一的基准测试程序.

由于图数据的不规则性特性, 通用处理器 (如 CPU 和 GPU) 通常难以高效地执行 BFS 算法. 在典型的图数据中, 不同对象间的关系极度稀疏且不规则. 这一特性导致通用处理器在执行 BFS 算法时面临严重的数据冲突与乱序访存问题 [6]. 现有研究表明, 通用处理器执行 BFS 算法时在超过 75% 时间中处于停滞状态 [7], 处理效率极低. 伴随着登纳德缩放定律 (Dennard scaling) 的终结, 通用处理器的计算效率逐渐难以满足图计算应用的需求 [8,9]. 为了解决该问题, 研究人员围绕图计算应用的计算与访存特征提出了一系列图计算领域专用的硬件加速器 [10,11]. 通过定制化的硬件设计, 图计算加速器可以在使用相同工艺的情况下取得相比于通用处理器数十倍的性能提升.

然而, 现有的图计算加速器在幂律图上运行 BFS 算法时面临严重的冗余访存问题. 幂律图结构在现实生活中广泛存在, 其中少部分顶点与绝大部分边相连而大部分顶点仅与少部分边相连 [12]. 例如, 在微博社交网络图中, 网络红人的被关注数往往远高于其他人 [13]. 由于相邻边数量较多, 幂律顶点与根顶点之间的距离通常较小, 因而在 BFS 算法中收敛速度较快. 然而, 现有的图计算加速器会强制执行每个活跃顶点的所有相邻边. 如果当前活跃顶点与已收敛的幂律顶点相邻, 该设计依旧会尝试访问和更新该幂律顶点, 最终导致不必要的相邻边访问操作. 本文第 2 节的实验表明, 现有的图计算加速器在处理典型的幂律图时的平均访存总量是理想情况的 15 倍, 造成了严重的内存带宽浪费问题. 在极端情况下, 现有图计算加速器运行 BFS 算法的性能甚至低于通用 CPU.

为了解决上述问题, 本文提出了一种冗余感知的高能效图计算加速器 JiFeng. 总体而言, JiFeng 通过引入提前跳出的设计思想减少 BFS 算法的冗余访存操作. 当检测到顶点属性值已收敛时, JiFeng 通过跳过剩余的相邻边任务减少当前顶点被重复访问的次数. 显然, 该设计思路的优化效果受到相邻边数据分布的影响, 即被跳过的相邻边在顶点邻居列表中的位置越靠后则性能提升越小. 为此, JiFeng 采用一种冗余感知的数据重排机制. 该机制首先在预处理阶段以较低的开销精确地预测每条相邻边的被跳过概率, 然后以该概率的大小为优先级依据对相邻边进行重排, 保证每个顶点可以跳过尽量多的相邻边任务. 此外, 为了避免片外内存的突发传输模型 (burst model) 预取被跳过的相邻边数据, JiFeng 提出一种边优先级感知的调度结构. 该调度结构按照交错的方式将多个顶点的高优先级相邻边映射到同一缓存行 (cacheline) 内, 并通过数据流调度模型优先调度高优先相邻边任务, 从而避免顶点跳过时低优先级数据被提前预取的情况. 最后, JiFeng 设计了一种负载均衡的缓存架构, 缓解多个处理单元同时访问幂律顶点导致的流水线阻塞问题.

本文的主要贡献如下:

- 深入研究了现有图计算加速器在处理幂律图的局限性, 并揭示了冗余访存问题的原理和严重性.
- 提出了一种冗余感知的高能效图计算加速器 JiFeng, 通过提前跳出的设计思想和一系列软硬件协同设计保证大部分顶点仅需遍历 1 条相邻边即可收敛, 显著减少无效的相邻边访问操作.
- 在 FPGA (field-programmable gate array) 原型系统上实现了 JiFeng 的硬件设计, 并在多个图数据上对其进行了广泛的性能评估. 实验结果表明, JiFeng 可以显著减少冗余访存量, 取得每秒遍历 4612 亿条边的性能和每秒每瓦特遍历 125 亿条边的能效比, 并在 2023 年 11 月的图计算超算排行榜 GreenGraph500 的小数据集榜单上排名第 2 [14].

本文的其余部分组织如下: 第 2 节讨论现有图计算加速器的局限性, 第 3 节介绍 JiFeng 的具体设计, 第 4 节阐述和分析实验结果, 第 5 节介绍相关工作, 第 6 节对本文工作进行总结.

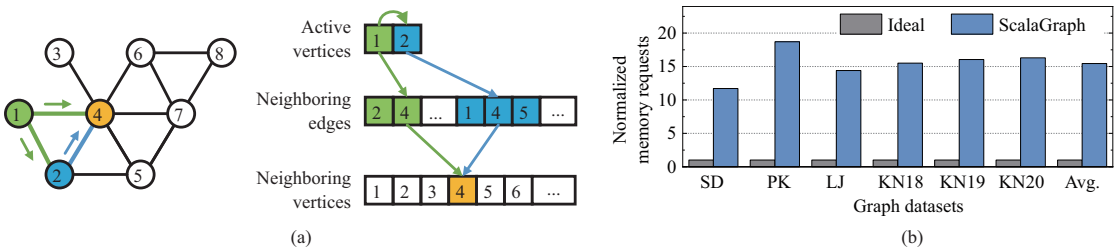

图 1 (网络版彩图) 现有图计算加速器中的冗余访存问题. (a) 不同活跃顶点重复更新相同邻居顶点的实例; (b) 典型图计算加速器的实际访存请求数量

Figure 1 (Color online) Redundant memory access problem in existing graph accelerators. (a) An example of redundant memory access between active and neighboring vertices; (b) the number of off-chip memory requests required by typical graph accelerators

2 背景和动机

本节首先介绍现有图计算加速器的冗余访存问题, 然后阐述 JiFeng 的设计思想与面临的挑战.

2.1 现有图计算加速器的局限性

图计算通常采用 $G = (V, E)$ 对现实实体间的关系进行形式化描述. 其中, V 代表顶点的集合, 主要用于表示图数据中的实体; E 代表边的集合, 主要用于表示实体之间的关联关系. 对于给定的图数据, BFS 算法从一个特定的根 (root) 顶点开始依次遍历每层顶点的所有相邻边, 然后计算相邻顶点与根顶点之间的距离值. 尽管并非所有应用都需要计算顶点之间的距离, 但 BFS 算法的邻居遍历方式在各类图计算算法中普遍存在, 具有重要的研究意义. 例如, 图挖掘算法 [15] 通过宽度优先的方式遍历邻居顶点并生成待匹配的子图结构.

为了提升计算并行度, 现有的图计算加速器通常采用以顶点为中心的模型 [1] 执行 BFS 算法. 在算法执行过程中, 图计算加速器从活跃顶点集合中获取活跃顶点 v, 然后依次访问其所有相邻边, 最后尝试用活跃顶点 v 的属性值更新邻居顶点. 如果邻居顶点的属性值发生变化, 则会被加入活跃顶点集合, 并在下一轮迭代中进行计算. 可以发现, 不同活跃顶点在上述模型中执行相同的计算任务. 因此, 通过迭代执行上述流程, 图计算加速器可以高并发地计算每个顶点与根顶点间的距离.

然而, 由于图数据的不规则性, 现有图计算加速器面临严重的冗余访存问题. 在图计算场景中, BFS 算法以宽度优先的方式从根顶点完成一次全图遍历. 因此, 在图计算场景的 BFS 算法中, 每个顶点仅需被更新一次即可得到最终的距离信息. 但在真实的图数据中, 多个顶点可能与同一顶点 v 相连. 由于现有图计算加速器强制遍历每个活跃顶点的所有相邻边, 因此顶点 v 会被多个顶点重复访问, 从而产生不必要的内存访问操作. 图 1(a) 展示了上述问题的一个例子. 顶点 1 和 2 均与顶点 4 相连. 顶点 1 首先处于活跃状态, 并遍历和更新顶点 4 的属性. 此时, 顶点 4 已经得到最终的计算结果, 理论上无需被再次遍历. 然而, 当顶点 2 处于活跃状态时依旧会遍历顶点 4. 尽管这次遍历不会产生顶点属性的更新, 但对相邻边 2→4 的访问仍然会产生一次额外的内存访问. 由于图计算算法是典型的内存敏感型应用 [16], 冗余的内存访问会降低内存带宽的利用效率, 进而降低整体性能.

当处理幂律图时, 上述问题的严重性愈加凸显. 现实图数据中顶点的相邻边数量 (即度数) 遵循幂律分布, 即少部分顶点度数较高而大部分顶点度数较低 [12]. 例如, 在 Twitter 社交网络图中, 度数最高的顶点拥有超过一百万条相邻边, 而度数最低的顶点仅有一条相邻边 [17]. 不均匀的度数分布会导致低度数顶点频繁尝试更新属性已经收敛的高度数顶点, 导致大量的无效内存访问. 本文统计了典型图

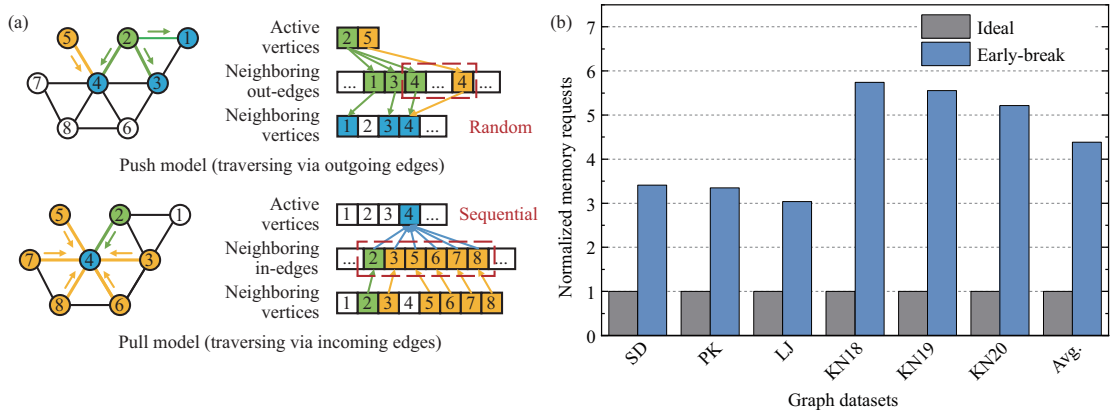

图 2 (网络版彩图) 基于 Pull 模型的提前跳出思想. (a) Push 和 Pull 执行模型中冗余相邻边的保存方式; (b) 基于提前跳出思想的 Pull 执行模型的实际访存请求数量

Figure 2 (Color online) Insight and advantage of early termination design philosophy. (a) The redundant neighboring edges stored in Push and Pull models; (b) the number of off-chip memory requests required by an early-termination-based Pull model

计算加速器 ScalaGraph [11] 在 6 个典型现实图 Slashdot (SD), Pokec (PK), LiveJournal (LJ) [18] 与生成图 Kronecker18 (KN18), Kronecker19 (KN19) 和 Kronecker20 (KN20) [5] 上运行 BFS 算法时的访存请求数量, 具体数据集参数详见第 4 节. 如图 1(b) 所示, ScalaGraph 的实际访存请求数平均为理想情况 (即每个顶点仅被访问一次) 的 15.4 倍, 导致了严重的内存带宽浪费.

2.2 设计思想与挑战

冗余访存问题存在的根本原因是大部分相邻边 (即冗余相邻边) 不会对最终结果产生影响. 现有的图计算加速器强制执行活跃顶点的所有相邻边, 导致已收敛顶点被重复访问和更新. 对于该问题, 本文发现当采用 Pull 模型在幂律图上执行 BFS 算法时, 活跃顶点的冗余相邻边存在较强的连续性. 典型的图计算算法具有自顶向下的 Push 和自底向上的 Pull 两种执行模型. 如图 2(a) 所示, 在 Push 模型中, 活跃顶点通过出边更新邻居顶点. 由于邻居顶点的收敛速度不同, Push 模型中的每个顶点的冗余边分布随机且动态变化, 难以在实际访问相邻边前实现精确的预测. 而在 Pull 模型中, 每个顶点通过入边访问邻居顶点, 然后更新自身的属性值, 即被更新对象从邻居顶点变为活跃顶点. 当活跃顶点收敛时, 所有剩余待遍历的相邻边均为冗余相邻边, 具备较强的连续性. 例如, 相邻边 2→4 和 5→4 在 Push 模型中乱序存储, 但在 Pull 模型中顺序存储.

基于上述发现, 本文基于 Pull 模型进一步提出提前跳出的设计思想. 具体而言, 当检测当前顶点完成迭代计算时, 该设计思想通过跳过剩余所有的相邻边减少冗余的内存访问. 如图 2(a) 所示, 假定仅有顶点 2 的属性值已经收敛. 传统基于 Pull 执行模型的图计算加速器激活顶点 4, 然后强制访问其所有相邻入边, 导致顶点 4 被顶点 2 和 3 重复访问和更新. 而采用提前跳出思想时, 顶点 4 首先访问邻居顶点 2, 并更新自身属性值. 此时, 顶点 4 的属性值已经收敛, 对后续所有相邻边 (如相邻边 3→4) 的访问均不会影响最终结果. 因此, 提前跳出思想会提前结束顶点 4 的计算, 从而避免对后续冗余相邻边的访问. 为了量化分析提前跳出思想的潜在优势, 本文统计了其在图 1(b) 中使用的 6 个现实图与生成图数据集上的访存请求数量. 如图 2(b) 所示, 当使用提前跳出思想时 BFS 算法的平均访存请求数仅为理想情况的 4.4 倍, 相比于 ScalaGraph 降低了 71.4%.

然而, 高效实现提前跳出思想存在以下 3 个挑战.

• **较高的顶点访问开销.** 在幂律图上执行 BFS 算法时, 活跃顶点集合在多数迭代轮次时较小. 例如, 尽管 LiveJournal 图包含超过 4 百万个顶点, 但在 BFS 算法的大多数迭代轮次里活跃顶点数量不超过 1 千. 然而, 提前跳出思想依赖的 Pull 执行模型在每轮迭代计算中均将所有顶点视为活跃顶点, 因而会产生大量无效的顶点访问操作.

• **随机的相邻边访问开销.** 只有当活跃顶点达到收敛状态后, 剩余的相邻边才可以被安全地跳过. 假设将顶点达到收敛状态前最后访问的相邻边定义为有效边, 则有效边在顶点的邻居列表中存放的位置将极大地影响提前跳出思想的效果. 在极端情况下, 如果有效边始终是活跃顶点的最后一条相邻边 (例如将相邻边 2→4 保存在 8→4 之后), 那么提前跳出思想将无法带来任何性能提升.

• **严重的负载不均衡问题.** 相比于现有的图计算加速器, 提前跳出思想根据活跃顶点的收敛性动态地跳过相邻边. 换而言之, 即使处理相同的图数据, 根顶点的变化也会导致顶点的访存负载发生显著变化. 因此, 提前跳出思想会进一步加剧不同顶点间任务的不均衡性, 导致严重的尾延迟效应.

为此, 本文提出了一种冗余感知的高能效图计算加速器 JiFeng. 针对 Pull 模型的额外访存开销, JiFeng 采用了基于 Push-Pull 的混合执行模型. 为了避免混合模型大幅增加硬件面积, JiFeng 通过构建统一的流水线架构降低硬件资源使用量. 针对相邻边数据存放顺序的挑战, JiFeng 提出一种高效的相邻边数据重排机制, 提升 BFS 算法的跳出速度. 为了避免内存的冲突访问模式 (即只使用 4 字节但读取 64 字节) 降低上述机制的效果, JiFeng 进一步提出了一种基于数据流的任务调度机制, 减少对无效数据的访问. 针对负载均衡挑战, JiFeng 使用了一种高效的缓存架构, 缓解对相同顶点的频繁访问导致的尾延迟问题. 本文的后续章节将依次介绍上述软硬件协同设计.

3 JiFeng 软硬件协同设计

本节首先介绍 JiFeng 的基于混合执行模型的流水线架构, 然后研究基于提前跳出思想的软硬件协同设计, 最后阐述 JiFeng 的负载均衡设计.

3.1 基于混合执行模型的流水线架构

针对提前跳出思想较高的顶点访问开销, JiFeng 通过引入 Push-Pull 混合执行模型 [19] 实现活跃顶点的高效遍历. 在 BFS 前几轮迭代轮次中, 活跃顶点数量较少 (少于总量的 1%), Pull 模型调度的绝大部分顶点任务均为无效任务. 此时, 相比 Pull 模型, Push 模型通过选择调度活跃顶点的方式大幅减少顶点访存开销. 尽管 Push 模型无法通过提前跳出思想减少相邻边访问, 但访存请求数量在活跃顶点较少时依旧远小于 Pull 模型. 而在中间的轮迭代轮次中, 活跃顶点数量剧增 (约为总量的 80%). 此时, 由于 Push 模型冗余相邻边的访问开销大幅上升, Pull 模型的执行效率远高于 Push 模型. 现有研究表明, Push-Pull 混合执行模型可以降低超过 50% 的顶点访问次数 [3]. 因此, JiFeng 通过根据活跃顶点稀疏度交替使用 Push 和 Pull 两类执行模型, 最小化顶点访问开销.

然而, 混合的执行模型会带来冗余的硬件设计. 例如, 现有工作为 Push 和 Pull 设计了独立的流水线架构, 导致硬件资源使用量增加了一倍 [20]. 针对这一难点, 本文发现 BFS 算法的 Push 和 Pull 模型的计算模式存在较大的相似性. 如图 3(a) 所示, 两个执行模型均遵循 “读活跃顶点数据 – 读边数据 – 读邻居顶点数据 – 处理数据 – 更新顶点” 的计算流程, 不同点在于相邻边的含义 (入边/出边) 与活跃顶点维护方式 (稀疏/稠密). 基于该观察, 本文提出了如图 3(b) 所示的统一数据流抽象. 对于相邻边, 该数据流抽象通过运行时配置动态判断读取边的方向. 对于活跃顶点维护方式, 该数据流抽象参考寄存器重命名的思想, 引入局部活跃顶点集合的设计. 当执行 Push 模型时局部活跃顶点集合被写回全

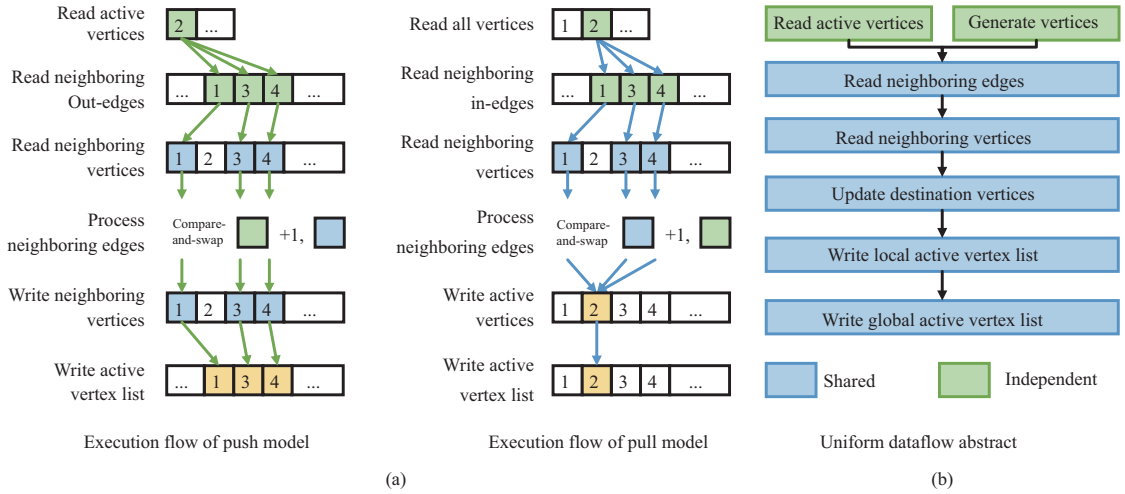

图 3　(网络版彩图) 基于混合执行模型的统一数据流抽象. (a) Push 和 Pull 执行模型的执行流程; (b) 面向 Push 和 Pull 模型的统一数据流抽象

Figure 3　(Color online) Uniform dataflow graph for hybrid execution model. (a) The similarity between the execution flows of Push and Pull model; (b) the uniform dataflow graph for both Push and Pull model

局活跃顶点集合, 而当执行 Pull 模型时局部活跃顶点集合会在运行时被忽略. 可以发现, 统一数据流抽象中大部分流程可以被 Push 和 Pull 模型共享, 减少了潜在的硬件资源浪费.

基于上述数据流抽象, 本文进一步提出如图 4 所示的流水线架构. 该流水线架构由 M 个 Tile 组成, 每个 Tile 与高带宽内存 (high-bandwidth memory, HBM) 的一个通道 (channel) 直接相连, 并通过片上互联网络向其他 Tile 传递消息. 每个 Tile 是独立处理顶点任务的最小硬件单元, 主要由调度器和处理器构成. 每个调度器包含 N 个调度单元和活跃顶点队列. 其中, 每个调度单元负责配置与其相连的处理单元的 Push/Pull 状态并调度活跃顶点队列中的活跃顶点 (见 3.2 小节). 每个处理器包含 N 个处理单元, 每个处理单元由活跃顶点, 邻居顶点, 顶点更新和片上缓存 4 个模块组成. 其中, 活跃顶点模块负责读取活跃顶点数据和相邻边数据, 邻居顶点模块负责读取和处理邻居顶点数据, 顶点更新模块负责将计算结果写回全局状态并维护活跃顶点集合, 片上缓存模块则负责缓存邻居顶点数据 (见 3.3 小节). 为了避免潜在的数据冲突风险, 处理单元 i 仅处理编号 ID$\%N = i$ 的顶点, 这导致一条相邻边的源顶点和目标顶点可能在不同的处理单元内处理. 因此, 处理单元内所有模块均通过片上互联网络而非直接连接的方式进行互联.

当执行不同的执行模型时, JiFeng 中绝大部分的硬件单元均可以被复用. 具体而言, 当执行 Push 模型时, 调度单元首先从活跃顶点队列读取活跃顶点编号. 然后, 活跃顶点模块分别从片上缓存和片外内存中读取活跃顶点数据和其相邻边数据. 当片外内存返回被读取的数据时, 活跃顶点模块根据邻居顶点编号的哈希值将其通过片上互联网络传送至对应的邻居顶点模块. 邻居顶点模块从片上缓存中读取邻居顶点数据并进行处理, 然后将计算结果根据相邻边的目标顶点编号的哈希值再次通过片上互联网络传送至对应的顶点更新模块. 顶点更新模块负责更新相关顶点数据, 生成局部活跃顶点集合, 并在本轮迭代结束时将其写回全局活跃顶点集合 (即调度器中的活跃顶点队列). 当执行 Pull 模型时, JiFeng 仅有少部分硬件逻辑与 Push 模型不同: (1) 由于 Pull 模型将所有顶点视作活跃顶点, 所以调度单元不再读取活跃顶点队列而是从 0 开始逐个生成活跃顶点编号; (2) 顶点更新模块仅会在判断下一轮迭代采用 Push 模型时才会将局部活跃顶点集合写回活跃顶点队列. 可以发现, JiFeng 仅需简单

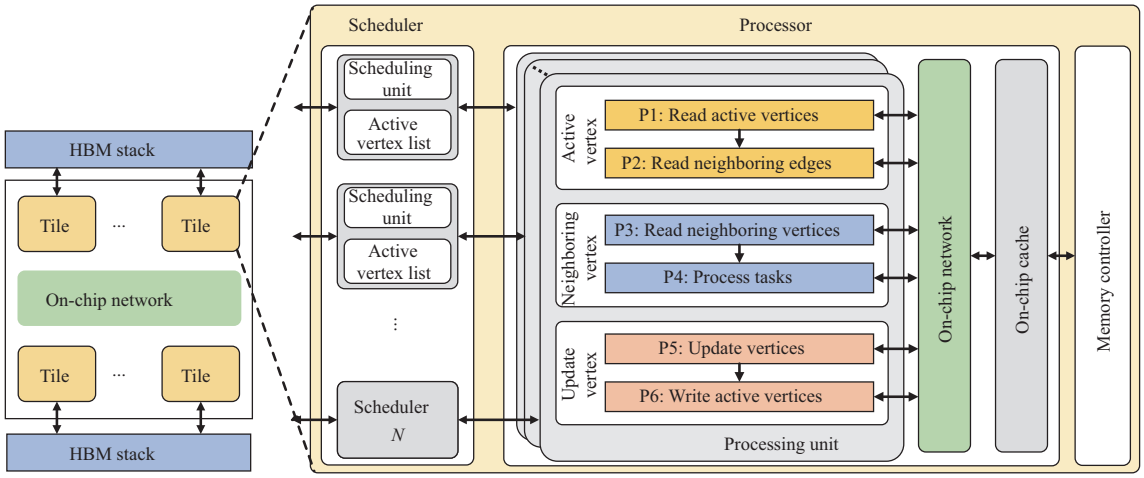

图 4 (网络版彩图) JiFeng 整体架构

Figure 4 (Color online) Overall architecture of JiFeng

的配置即可实现对混合执行模型的支持, 避免了冗余的硬件设计.

3.2 冗余感知的软硬件协同设计

虽然上述设计可以有效减少提前跳出思想的顶点访问数量, 但相邻边访问依然面临严峻的挑战. 一方面, 相邻边访问总量受有效相邻边 (即每个顶点收敛前最后访问的相邻边) 保存位置影响. 如图 2(b) 所示, 每个顶点平均需要遍历 4.4 条边才能达到收敛状态, 远低于理想情况. 另一方面, 相邻边访问数量受内存的突发传输模型影响. 即使每个顶点仅需要访问一条相邻边 (4 字节), 现有的内存设计依旧会一次性访问该相邻边所在的 64 字节数据行, 造成严重的内存带宽浪费. 如图 5(a) 所示, 当在 Xilinx Alveo U55C 加速卡上运行基于提前跳出思想的 BFS 算法时, 每个顶点实际平均需要遍历 8.6 条边才能达到收敛状态, 是理论值的 1.9 倍. 因此, 高效地提前跳出设计需要精确地定位与访问有效相邻边, 减少无效相邻边带来的冗余内存访问.

针对上述挑战, 本文做出如下两点发现:

• **在幂律图上运行 BFS 算法时, 少部分顶点会成为大部分顶点的父亲顶点.** 该发现的内在原因在于, 幂律图的直径通常较小. 导致活跃顶点数量通常遵循 “稀疏 – 稠密 – 稀疏” 的规律. 当活跃顶点数量从稀疏变为稠密时, 少部分顶点会更新大部分顶点并成为其父亲顶点. 图 5(b) 展示了上述 6 个自然图和生成图在所有根顶点情况下的累积分布结果, 其中横坐标代表当前已统计的顶点占顶点总数的比例, 纵坐标代表已统计顶点成为父亲顶点的次数占父亲顶点总数的比例. 当按照成为父亲顶点次数的降序对顶点排序后, 5% 的顶点最多可以成为 93% 的顶点的父亲顶点.

• **在幂律图上运行 BFS 算法时, 高度数顶点成为父亲顶点的频率极高.** 对于任意根顶点, 高度数顶点因为相邻边较多, 所以在 “稀疏” 迭代轮次中达到收敛状态的概率极高. 因此, 在后续 “稠密” 迭代轮次中, 剩余顶点大概率会被高度数顶点更新并成为其子顶点. 如图 5(c) 所示, 按照度数降序排序的曲线与按照成为父亲顶点次数降序排序的误差不超过 4.5%, 且随着顶点占比的增加而显著下降. 这意味着顶点的一条相邻边是有效边的概率与相邻顶点的度数成近似正相关关系.

基于上述发现, 本文提出如算法 1 所示的冗余感知的数据重排机制. 总体而言, 冗余感知的数据重排机制以顶点度数作为定位有效相邻边的主要指标. 该机制首先统计所有顶点的度数, 然后按照相

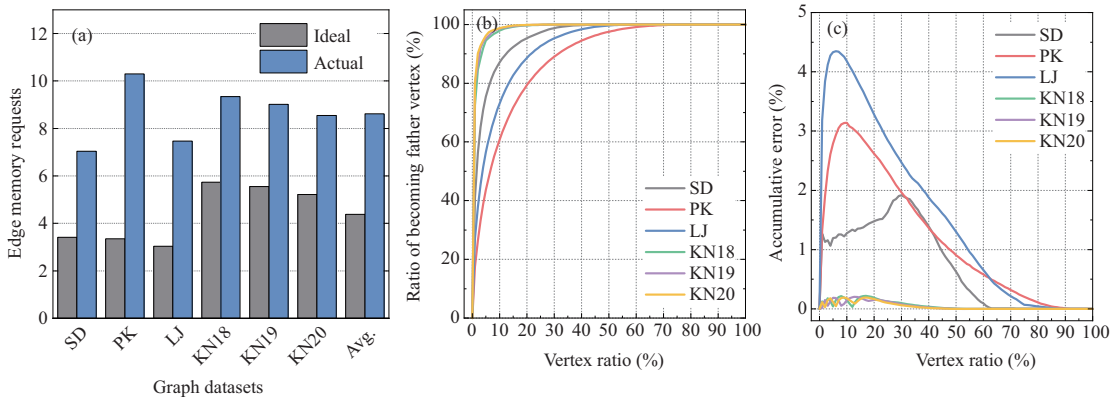

图 5 (网络版彩图) 冗余感知的数据重排机制. (a) 内存突发传输模式显著增加相邻边访问数量; (b) 少部分顶点会成为大部分顶点的父亲顶点; (c) 用顶点度数预测有效相邻边的误差极小

Figure 5 (Color online) Redundancy-aware sorting mechanism. (a) The burst model of memory introduces significant read amplification; (b) a small portion of vertices can be the father of most vertices in BFS; (c) degree-based prediction is effective

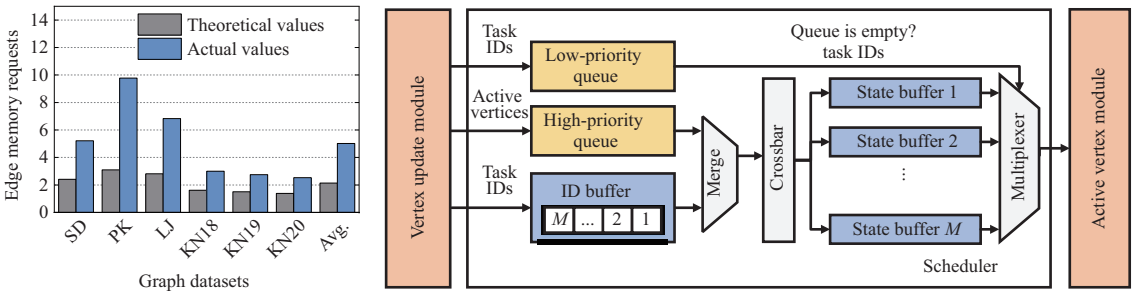

图 6 (网络版彩图) 优先级感知的调度结构. (a) 内存突发传输模型依旧带来一定的额外开销; (b) 基于提前跳出的调度器

Figure 6 (Color online) Priority-aware scheduler. (a) Burst model introduces non-negligible overheads; (b) early-termination-based scheduler

邻顶点度数的降序对每个顶点的相邻边进行重新排序 (算法 1 第 1~6 行), 保证顶点可以尽早达到收敛状态. 由于相邻边顺序不影响 BFS 算法最终的计算结果, 冗余感知的数据重排机制在保证了正确性的同时显著减少了相邻边的访问数量. 如图 6(a) 所示, 在重排后的图数据上运行 BFS 算法时, 每个顶点实际平均仅需遍历 2.1 条边即可达到收敛状态, 降低了约一半的相邻边访问次数.

然而, 内存突发传输的读放大问题依然存在, 导致相邻边读取效率降低至 5.0 条边. 为此, 冗余感知的数据重排机制对重排后的数据采用了差分管理策略 (算法 1 第 7~12 行). 具体而言, 本文假定绝大部分顶点仅需遍历 K 条边即可达到收敛状态. 因此, 该数据管理策略将重排后数据中每个顶点的前 K 条相邻边定义为高优先级相邻边, 并将其保存在单独的内存地址空间 HLE. 在 BFS 算法执行过程中, 该数据管理策略首先遍历所有顶点的高优先级相邻边, 然后再遍历未收敛顶点的其余相邻边 (低优先级相邻边). 当遍历高优先级相邻边时, 尽管突发传输模型依旧会一次性访问 16 条相邻边, 但这些边具有极高的概率是下一个被调度顶点的有效边, 从而避免内存带宽的浪费. 由于大部分顶点会在访问高优先级相邻边时收敛, 所以大部分低优先级相邻边不会被访问, 此时突发传输模型的影响可以忽略不计.

算法 1 冗余感知的数据重排机制

输入: 顶点集合 V, 顶点相邻边列表 Ngh.
输出: 高优先级相邻边 HLE, 低优先级相邻边 LLE.

for $i \in V$ **do**
  $D = |\mathrm{Ngh}[i]|$;
end for
for $i \in V$ **do**
  Sort $\mathrm{Ngh}[i]$ in the descending order of $D[\mathrm{Ngh}[i][j]]$;
end for
for $i \in V$ **do**
  for $j \in [0, K)$ **do**
    $\mathrm{PUSH}(\mathrm{HLE}[i], \mathrm{Ngh}[i][j])$;
  end for
  for $j \in [K, |\mathrm{Ngh}[i]| - 1]$ **do**
    $\mathrm{PUSH}(\mathrm{LLE}[i], \mathrm{Ngh}[i][j])$;
  end for
end for

差分的数据管理策略要求分批处理不同优先级的相邻边, 对硬件架构带来了新的挑战. 为此, 本文进一步提出了如图 6(b) 所示的优先级感知的任务调度结构. 为了差分处理不同优先级的相邻边数据, 调度器将活跃顶点列队细分为高优先级队列和低优先级队列. 为了避免潜在的死锁和队列溢出风险, 调度器参考经典的数据流调度思想为每一个顶点任务分配一个唯一的任务编号. 当顶点在本轮迭代第 1 次被调度时, 调度器会将其写入高优先级队列, 为其分配编号缓冲区中的一个编号, 然后根据编号将其送往状态缓冲区内等待处理. 当低优先队列为空时, 调度器会随机选取状态缓冲区中的一个顶点任务, 并处理其高优先级相邻边. 当高优先级相邻边处理完成后, 如果顶点已收敛, 则调度器会提前跳过所有低优先级相邻边; 否则, 调度器将任务编号写入低优先级队列. 此时, 调度器会根据低优先级队列中的任务编号读取对应的状态缓冲区, 然后再依次处理所有低优先级相邻边. 由此, 优先级感知的任务调度结构以统一的硬件结构实现了高优先级相邻边的优先处理和低优先级相邻边的提前跳出, 保证了差分数据管理策略的高效部署.

3.3 负载均衡设计

提前跳出思想显著降低了相邻边访问数量, 但是会带来严重的负载不均衡问题. 如 3.2 小节所述, 由于幂律的度数分布, 少部分顶点会成为大部分顶点的父亲顶点. 尽管这一特性使得活跃顶点仅需要遍历极少的相邻顶点即可收敛, 但也使得少部分顶点的被访问次数远远高于其余顶点, 最终导致严重的读冲突问题. 例如, 在图 1(a) 的例子中, 顶点 1 和 2 均需要访问幂律顶点 4 的属性值. 尽管顶点 1 和 2 可能被不同的处理单元处理, 但对顶点 4 的访存请求由于内存地址的唯一性会被发往同一块片上缓存进行处理. 由于标准的缓存单元仅能统一处理一条读请求和写请求, 这一不均衡的访存行为会导致流水线因为缓存吞吐量不足而频繁停滞, 造成较长的尾延迟.

针对这一挑战, 本文发现导致尾延迟的顶点数量极少 (通常为数十个). 这是因为, 顶点的被访问次数具有较强的幂律性, 即少部分顶点被大部分顶点访问. 因此, 仅需为这少部分顶点保存多个备份, 即可在不显著增加缓存容量的同时显著减少 BFS 算法中的读冲突问题. 基于上述思路, 本文提出如图 7 所示的异构缓存架构. 相较于典型的图计算加速器, 该缓存架构在每个处理单元内额外引入了一块独立的局部缓存, 用以保存少部分频繁被访问的顶点. 为了避免潜在的数据不同步风险, 局部缓存

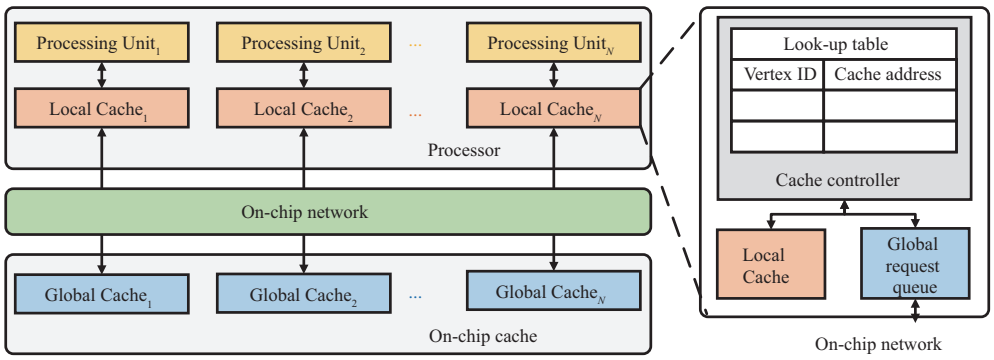

图 7　(网络版彩图) 异构缓存架构

Figure 7　(Color online) Hybrid on-chip memory hierarchy

与全局缓存以包含式 (inclusive) 缓存结构进行组织. 局部缓存由缓存控制器、全局缓存请求队列和高频顶点局部缓存组成. 其中, 缓存控制器负责在运行时通过查找表判断访存请求由局部或全局缓存进行处理, 全局缓存请求队列负责缓存和维护送往全局缓存的访存请求, 而高频顶点局部缓存负责静态保存高频顶点的属性值. 由于每个顶点被访问次数与其度数成正相关关系, 所以高频顶点局部缓存仅保存度数最高的 C 个顶点. 由于处理单元可以在本地获取高频顶点的属性值, 因此不会向全局缓存发送访存请求, 缓解了潜在的负载不均衡问题.

在图 1(a) 所示的图结构中, 假设顶点 4 是一个高频顶点, 并被静态保存于局部缓存中. 当顶点 1 进入处理单元时, 处理单元首先检测相邻边是否命中缓存控制器中的查找表. 若相邻顶点命中查找表 (如顶点 4), 则缓存控制器会从查找表中取出对应的局部缓存地址, 并基于该地址生成访问局部缓存的访存请求. 若没有命中查找表 (如顶点 2), 缓存控制器则直接根据相邻顶点编号生成全局缓存的访问请求, 并通过片上互联网络发送至全局缓存. 为了保证数据一致性, 局部缓存在每轮迭代进行时不处理任何写请求. 只有当每一轮迭代计算结束时, 处理单元会生成特殊的读写请求将全局缓存中的高频顶点数据同步至局部缓存. 可以发现, 通过将高频顶点缓存至局部缓存, 该架构以较低的硬件开销大幅提升了片上缓存的等效吞吐量, 从而显著减少流水线的停滞时间.

4　实验与结果

本节采用具有代表性的自然图和生成图数据集评估 JiFeng 的性能, 首先将 JiFeng 与相关的 CPU 系统, GPU 框架和加速器进行对比, 然后分析 JiFeng 的软硬件协同设计的有效性.

4.1　实验配置

实验环境设置. 本小节采用 Verilog RTL 实现 JiFeng, 并将其部署在 Xilinx Alveo U55C 加速卡上. 该加速卡包含一块 XCU55 FPGA 芯片, 提供包含 1.3M 查找表 (look-up tables, LUTs), 2.6M 寄存器 (registers, REGs) 和 9 MB 块随机访问内存 (block random access memory, BRAM) 在内的可编程硬件资源和 2 块 8 GB 容量 230 GB/s 内存带宽的 HBM2 堆栈. 在本节的实验中, JiFeng 实现了 32 个 Tile 和 6 MB 片上缓存. 每个 Tile 与一个 HBM 堆栈中的伪通道 (pseudo channel) 直接相连, 且不允许直接访问其余 HBM 伪通道. 由于每个 HBM 伪通道一次读取 16 条相邻边 (4×16 字节 $= 64$ 字节), 所以每个 Tile 内配置 16 个吞吐量为每时钟周期处理 1 条相邻边的处理单元. 为了保障 JiFeng 的高效扩展, 本章采用了基于 HyperX 的分级网络实现 Tile 之间和处理单元之间的互联. 由于 JiFeng 采用同

表 1 图数据集

Table 1 Graph datasets

Graph	|Vertex|	|Edge|	Description
Slashdot (SD)	0.07M	0.5M	Social network, directed graph
Pokec (PK)	1.6M	30.6M	Social network, directed graph
LiveJournal (LJ)	4.8M	68.9M	Social network, directed graph
Kronecker18 (KN18)	0.3M	3.8M	Synthetic graph, undirected graph
Kronecker19 (KN19)	0.5M	7.7M	Synthetic graph, undirected graph
Kronecker20 (KN20)	1.0M	15.7M	Synthetic graph, undirected graph

步模型, 活跃顶点的深度信息可以直接根据迭代轮次获取. 因此, 片上缓存主要负责保存相邻顶点的深度信息 (1 字节), 并被均匀的分给所有处理单元. JiFeng 的时钟频率信息通过 Xilinx Vivado 2019.1 获取. 本节所有实验均使用 150 MHz 作为 JiFeng 最终的时钟频率.

对比对象设置. 本节将 JiFeng 与典型的 CPU 图计算系统 Ligra, GPU 图计算框架 Gunrock 和 FPGA 图计算加速器 ScalaGraph 进行对比. 所有实验均在一台包含 2 颗 28 核心 2 GHz 的 Intel Xeon Gold 6330 处理器, 1 TB DDR4 3200 内存, 1 块 NVIDIA A100 GPU 和 1 块 Xilinx Alveo U55C 加速卡的服务器上运行. 其中, DDR4 内存一共提供 409 GB/s 的聚合带宽, 与 Xilinx Alveo U55C 的 HBM 内存带宽接近. NVIDIA A100 GPU 包含 80 GB HBM2e, 聚合带宽为 2 TB/s, 内存容量和带宽均远高于 Xilinx Alveo U55C 加速卡. Ligra 和 Gunrock 均采用系统默认的参数和 BFS 实现, ScalaGraph 采用与 JiFeng 一样的处理单元 (512) 和片上缓存 (6 MB) 配置.

图数据集设置. 本节采用了 3 个来源于 Stanford Network Analysis Project [18] 的自然图 Slashdot, Pokec 和 LiveJournal 与 3 个来源于 Graph500 [5] 的生成图 Kronecker18, Kronecker19 和 Kronecker20 评估 Ligra, Gunrock, ScalaGraph 和 JiFeng 的性能. 6 个图数据集的详细参数如表 1 所示, 所有图采用 CSR (compressed sparse row) 和 CSC (compressed sparse column) 结构保存. 本章测试方案与 Graph500 一致, 即对于任意数据集随机生成 64 个根顶点执行 BFS 算法, 最后取平均值作为最终的测试结果.

4.2 整体性能评估

吞吐量. 本节采用图计算超算榜单 Graph500 中定义的 GTEPS (Giga-traversed edge per second) 评估各类系统与加速器的吞吐量. 具体而言, GTEPS = Edges/Time, 其中 Edges 代表根顶点所在连通分量的边的数量 (无向边仅算一条边而非两条有向边), Time 代表 BFS 算法从开始到收敛的总时间 (非每轮迭代的时间). 如图 8(a) 所示, 展示了测试对象的吞吐量结果.

相比于基于通用处理器的图计算系统 Ligra 和 Gunrock, JiFeng 分别可以提供平均 98.6 和 55.9 倍的吞吐量提升. 一方面, Ligra 和 Gunrock 受限于低效的乱序顶点访问, 在超过 50% 的时间中都处于流水线停滞状态. JiFeng 通过定制化的缓存架构静态缓存顶点数据, 大幅减少乱序片外访问. 另一方面, 不规则的图结构导致 Ligra 和 Gunrock 面临严重的数据同步开销. JiFeng 通过高效的片上互联设计避免多个处理单元同时更新相同顶点, 从而减少原子更新操作带来的流水线停滞.

相比于 ScalaGraph, JiFeng 可以提供最高 461.2 GTEPS 的吞吐量和平均 28.3 倍的吞吐量提升. 一方面, 当活跃子图稠密时, 基于 Push 执行模型的 ScalaGraph 会面临严重的活跃顶点维护开销. JiFeng 通过基于 Push/Pull 混合执行模型的流水线架构显著降低活跃顶点维护开销, 提升处理稠密子图时的

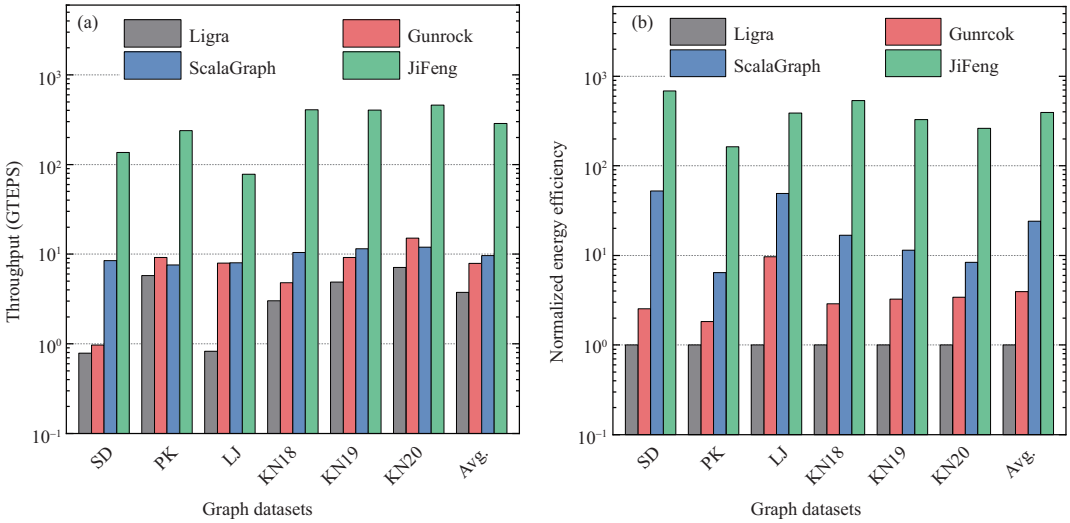

图 8　(网络版彩图) (a) 整体吞吐量; (b) 整体性能功耗比

Figure 8　(Color online) (a) Overall throughput; (b) overall energy efficiency

吞吐量. 另一方面, Push 执行模型会频繁访问无效的相邻边, 导致严重的冗余边访存问题. JiFeng 通过基于提前跳出思想的软硬件协同设计, 大幅减少了需要遍历的相邻边数量, 从而进一步提升吞吐量. 在所有数据集中, JiFeng 在生成图 Kronecker 上的加速比高于自然图. 这是因为, Kronecker 图中约 40% 的顶点均为孤立顶点, 导致其幂律特性相比自然图更加显著. JiFeng 在处理 Kronecker 图时可以遍历比自然图更少的相邻边, 因此取得了更高的加速比.

性能功耗比. 图 8(b) 展示了以 Ligra 为基准的归一化性能功耗比. 本文使用 Intel Performance Counter Monitor 收集 CPU 的能耗信息, Nvidia System Management Interface 收集 GPU 的能耗信息, 用 Xilinx Board Utility 收集 FPGA 的能耗信息. 相比于 Ligra 和 Gunrock, JiFeng 可以分别提供平均 394.6 倍和 127.3 倍的性能功耗比提升. 这是因为 JiFeng 通过定制化的流水线设计以更低的 (150 MHz vs. >1.5 GHz) 运行频率取得了更高的指令执行效率, 从而大幅降低片上单元的能耗. 相比于 ScalaGraph, 尽管 JiFeng 采用了更复杂的执行模型 (Push/Pull vs. Push), 但由于更低的运行频率 (150 MHz vs. 250 MHz), 所以整体功率与 ScalaGraph 相近. 此外, JiFeng 通过提前跳出思想大幅减少了需要遍历的相邻边数量, 因此整体性能功耗比相比 ScalaGraph 提升了 23.1 倍. 值得注意的是, JiFeng 在 Kronecker20 数据集上取得了 12.5 GTEPS/W 的性能功耗比 (461.2 GTEPS 的吞吐量和 36.8 W 的平均功率), 并在 2023 年 11 月的 GreenGraph500 小数据集榜单上取得了第 2 名的成绩. 据作者们所知, JiFeng 是目前已公开的基于 FPGA 的 BFS 加速器中性能功耗比最高的设计.

4.3　优化技术有效性评估

基于混合执行模型的流水线架构. 为了验证基于混合执行模型的流水线架构的有效性, 本文首先将 JiFeng 与 ScalaGraph 的硬件资源使用量进行对比. 如图 9(a) 所示, 尽管 JiFeng 采用了比 ScalaGraph 更复杂的执行模型 (Push-Pull vs. Push), 但查找表, 寄存器和块随机访问内存的资源使用量仅分别为 ScalaGraph 的 1.2 倍, 1.3 倍和 1.03 倍. 这是因为, JiFeng 采用了基于混合执行模型的统一数据流抽象, 使得 Push 和 Pull 模型可以复用大部分流水线单元, 进而降低硬件资源使用量.

为了验证基于混合执行模型的流水线架构的健壮性, 本文对不同稀疏度阈值进行了性能敏感性分

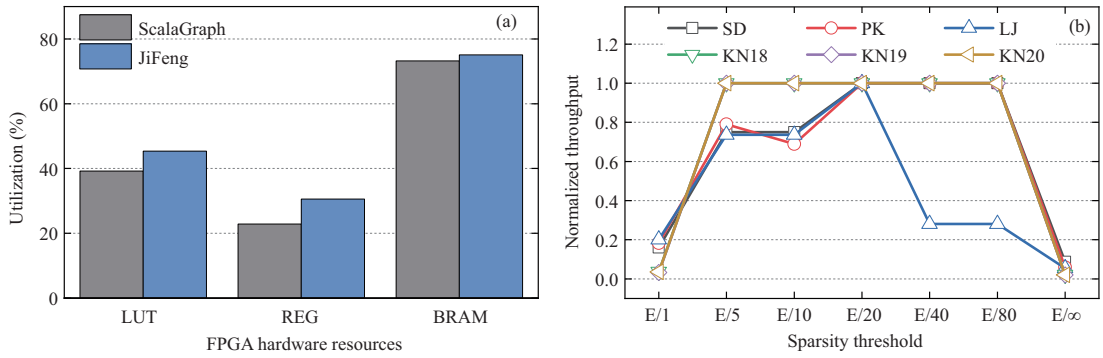

图 9 (网络版彩图) JiFeng 流水线架构的 (a) 硬件资源使用量和 (b) 性能敏感性

Figure 9 (Color online) (a) Hardware resources utilization and (b) sensitivity analysis of JiFeng architecture

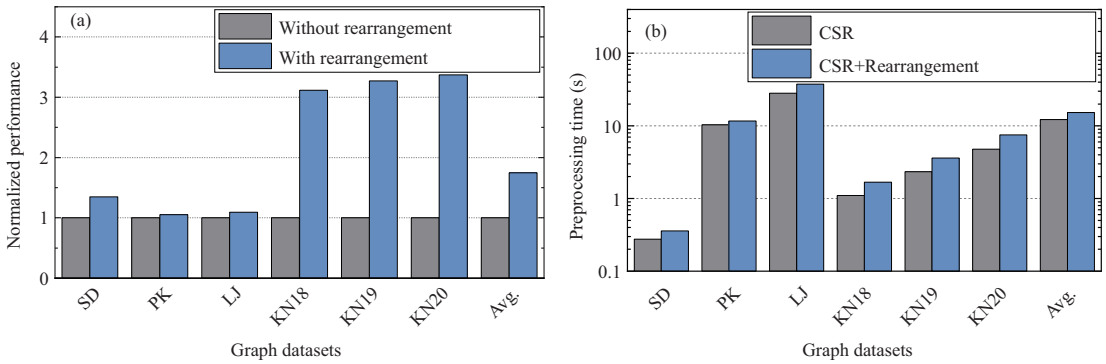

图 10 (网络版彩图) 冗余感知的数据重排机制的 (a) 有效性和 (b) 预处理开销

Figure 10 (Color online) (a) Effectiveness and (b) overheads of redundancy-aware sorting

析. 对于 Push 与 Pull 模型之间的切换阈值与策略, JiFeng 沿用了前人工作 Ligra 中的设计. 具体而言, 假设当前活跃顶点数量为 U, 所有活跃顶点的出边数量为 E_U, 输入图的总边数量为 E, 则 JiFeng 根据 $U+E_U$ 和默认稀疏度阈值 $E/20$ 间的大小采用对应的执行模型 (当 $U+E_U > E/20$ 时采用 Pull 模型; 否则采用 Push 模型). 为了分析不同阈值对加速结果的影响, 本文将阈值从 $E/1$ 逐渐降低至 E/∞, 并测试 JiFeng 在不同数据集上的执行性能. 如图 9(b) 所示, 当阈值等于 $E/1$ 和 E/∞ 时, JiFeng 的性能相对较差. 这是因为, 当阈值为 $E/1$ 或 E/∞ 时, $U+E_U$ 恒定小于或大于阈值, 导致 JiFeng 仅采用 Push 或 Pull 中的一个执行模型, 性能差于混合执行模型. 当采用默认阈值 $E/20$ 时, JiFeng 在所有数据集上均达到最优的性能, 验证了默认阈值的有效性.

冗余感知的数据重排机制. 为了验证冗余感知的数据重排机制的有效性, 本文将 JiFeng 部署与不部署该机制的性能进行对比. 如图 10(a) 所示, 冗余感知的数据重排机制可以带来 1.1~3.4 倍的性能提升. 这是因为, 相邻边被跳过的概率与相邻顶点的度数成正相关关系. 冗余感知的数据重排机制通过将边数据按相邻顶点度数排序, 显著减少了每个顶点访问的相邻边数量. 在所有数据集中, 该机制在生成图上的效果优于自然图. 这是因为, 一方面生成图中大量的孤立点使得其度数的幂律分布特征更加明显, 导致冗余感知的数据重排机制可以更早地结束每个顶点的执行. 另一方面, 自然图中更均匀的度数分布也导致其受内存突发模型的影响更大, 从侧面验证了软硬件协同设计的必要性.

图 10(b) 展示了冗余感知的数据重排机制和传统数据压缩方法 (CSR) 的预处理时间结果. 可以发

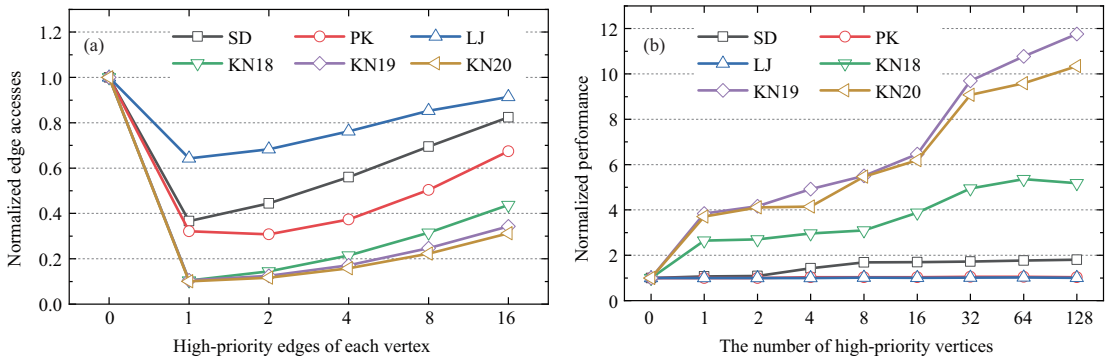

图 11　(网络版彩图) (a) 优先级感知的调度结构的有效性; (b) 负载均衡的异构缓存架构的有效性

Figure 11　(Color online) (a) The effectiveness of priority-aware scheduler; (b) the effectiveness of hybrid cache hierarchy

现, 当输入图以边表的形式存储时, 冗余感知的数据重排机制的预处理时间并未比传统压缩方法显著增加. 这是因为, 冗余感知的数据重排仅对每个顶点的相邻边单独进行排序操作, 一定程度上降低了算法的复杂度. 此外, 由于冗余感知的数据重排机制与具体根顶点无关, 因此不同的根顶点查询请求可以复用预处理后的数据, 进一步降低预处理开销在整体执行时间的占比.

优先级感知的调度结构. 图 11(a) 展示了优先级感知的调度结构的性能随每个顶点的高优先级边数量 (算法 1 中的 K) 的变化情况. 由于当 $K = 0$ 时该调度结构等价于传统的图计算加速器, 因此本实验选择 $K = 0$ 的性能作为归一化标准. 总体而言, 当 $K > 1$ 后, JiFeng 的性能随 K 值的增大而降低. 这是因为, 每个顶点在经过重排后平均仅需遍历 1∼2 条边即可达到收敛状态. 维护更多的高优先级边会导致每个顶点从片外读取超过其需求的相邻边, 最终导致性能降低. 因此, JiFeng 采用 $K = 1$ 作为默认配置. 此外, 优先级感知的调度结构对自然图的有效性差于生成图. 这是因为, 自然图中度数分布的幂律性弱于生成图, 导致大量顶点依旧需要遍历超过 1 条相邻边才能达到收敛状态. 如图 6(a) 所示, 在考虑内存突发传输影响时自然图顶点平均遍历边数是生成图的 2∼3 倍. 这一实验结果从侧面印证了 JiFeng 在自然图上的吞吐量低于生成图的实验结果.

负载均衡的异构缓存架构. 图 11(b) 展示了负载均衡的异构缓存架构的性能随高频顶点局部缓存内保存的顶点数 C 的变化情况. 由于当 $C = 0$ 时, 该缓存架构等价于传统的图计算加速器, 因此本实验选择 $C = 0$ 的性能作为归一化标准. 总体而言, JiFeng 的性能随 C 值的增大而提升. 这是因为, 一方面, 负载不均衡问题造成的流水线停滞随局部缓存的大小增加而降低; 另一方面, 局部缓存内的顶点数据较小, 其数据同步开销相较于活跃顶点处理开销可以忽略不计. 然而, 当 $C > 32$ 时, 局部缓存的数据同步开销在整体执行时间的占比显著增加, 导致 JiFeng 的性能提升速度放缓甚至负增长. 因此, 为了在性能和硬件开销之间取得平衡, JiFeng 采用 $C = 32$ 作为默认配置. 为了降低该设计的资源开销, JiFeng 采用 1 字节表示顶点的属性值信息. 由于 JiFeng 包含 512 个 PE, 因此该额外的缓存设计共计需要 $512 \times 32 \times 1 = 16$ KB 的 BRAM 资源, 相比于 6 MB 的总缓存容量仅额外增加了 $16/(6 \times 1024) = 0.26\%$ 的资源开销, 可以忽略不计. 此外, 负载均衡的异构缓存架构对于自然图 LiveJournal 和 Pokec 几乎没有效果. 这是因为 LiveJournal 和 Pokec 幂律性较弱, BFS 算法对高度数顶点的访问相对均匀, 因此极少命中容量较小的局部缓存.

5 相关工作

基于通用处理器的图计算系统. 围绕图计算中的冗余边访问, 现有工作提出了一系列算法优化技术减少高昂的片外通讯开销 [21]. 例如, Ligra [19] 提出了基于 Push-Pull 混合执行模型的图计算编程框架, 通过在运行时自适应的调整执行模型减少对幂律顶点的重复更新与激活. CLIP [22] 对于已获取的图数据采用多次计算的方式加快收敛速度, 减少整体访存次数. Garaph [23] 通过 CPU 和 GPU 之间高效协同计算减少图数据传输开销. 然而, 通用处理器难以高效地处理 BFS 算法不规则的访问行为, 导致这些工作的效果受限于较低的缓存命中率和内存带宽利用率, 性能功耗比较低. 尽管这些技术在通用处理器上取得了较好的效果, 但其性能依旧受限于 CPU 较差的指令效率和较低的内存带宽, 绝对性能较低. 相较于这些工作, JiFeng 充分利用 FPGA 平台的可重构特性, 通过稀疏性感知的缓存设计显著提升内存带宽利用率, 取得远高于 GPU 的性能功耗比.

图计算加速器. 针对通用处理器的局限性, 现有工作提出了大量面向图计算的领域专用加速器尝试提升图计算算法的访存效率 [24]. 例如, Graphicionado [10] 通过将顶点数据静态缓存于片上便笺存储器的方式, 将慢速的乱序片外访问转化为快速的乱序片上访问, 提升内存带宽利用率. Hats [25] 提出了面向图计算的顶点数据预取器, 降低内存访问延迟. Gramer [26] 通过静态缓存频繁被访问的图数据和动态替换剩余图数据的方式, 减少片外访存总量. 然而, 这些工作由于强制执行活跃顶点的所有相邻边导致了严重的冗余边访问问题. 相较于这些工作, JiFeng 通过基于提前跳出思想的软硬件协同设计, 显著降低了图计算的冗余边访问数量, 大幅提升 BFS 算法的执行效率.

也有一些工作尝试通过减少被调度的顶点数量以降低图计算的相邻边访问量. 例如, 威斯康星大学麦迪逊分校 (University of Wisconsin-Madison) 的研究人员 [20] 实现了基于 Push-Pull 混合执行模型的 FPGA 加速器, 减少对幂律顶点的重复激活. PolyGraph [27] 提出了灵活的图计算加速架构, 通过自适应的调整数据同步策略减少顶点被调度次数. ScalaGraph [11] 设计了缓存结构感知的任务映射机制和跨迭代的流水执行机制, 减少每个顶点的激活与通信开销. 尽管这些工作一定程度上缓解了冗余边访问问题, 但依然没有改变需要遍历每个顶点所有相邻边的本质. 此外, 这些工作的性能还受到内存突发传输模型的限制. 相较于这些工作, JiFeng 通过冗余感知的数据重排机制和优先级感知的任务调度结构, 保证 BFS 算法中大部分顶点仅需遍历一条相邻边即可收敛, 实现了相比现有加速器数十倍的性能提升.

6 结论

本文提出了一种冗余感知的 FPGA 图计算加速器 JiFeng, 通过引入提前跳出的设计思想大幅减少 BFS 算法的冗余访存操作. JiFeng 通过基于混合执行模型的流水线架构自适应地调整活跃顶点维护策略, 避免无效的活跃顶点访问操作. 此外, JiFeng 设计了冗余感知的数据重排机制和优先级感知的任务调度结构, 通过优先访问和处理高频顶点的相邻边加快活跃顶点的收敛速度, 从而大幅缩减相邻边访问总量. 最后, JiFeng 实现了负载均衡的异构缓存架构, 通过局部缓存频繁被访问的顶点数据, 减少因冲突的全局数据访问导致的流水线停滞开销. 实验结果表明, 与最先进的 CPU 和 FPGA 方案相比, JiFeng 能够在相同的内存带宽下实现数十倍的性能提升. 在典型的生成图 Kronecker 上, JiFeng 取得了最高 461 GTEPS 的吞吐量和 12.5 GTEPS/W 的能效比, 在 2023 年 11 月的图计算超算排行榜 GreenGraph500 的小数据集榜单上位列第 2 名.

参考文献

1 Malewicz G, Austern M H, Bik A J, et al. Pregel: a system for large-scale graph processing. In: Proceedings of ACM SIGMOD International Conference on Management of Data, 2010. 135–146

2 Xu S X, Liao X F, Shao Z Y, et al. Maximal clique enumeration problem on graphs: status and challenges. Sci Sin Inform, 2022, 52: 784–803 [许绍显, 廖小飞, 邵志远, 等. 图数据中极大团枚举问题的求解: 研究现状与挑战. 中国科学: 信息科学, 2022, 52: 784–803]

3 Beamer S, Asanovic K, Patterson D. Direction-optimizing breadth-first search. In: Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis, 2012. 1–10

4 Zhang C, Cao H, Wang G, et al. Efficient optimization of graph computing on high-throughput computer. J Comput Res Develop, 2020, 57: 1152–1163

5 Hoefler T. Green graph500. 2012. http://green.graph500.org/

6 Wang J, Zhang L, Wang P Y, et al. Memory system optimization for graph processing: a survey. Sci Sin Inform, 2019, 49: 295–313 [王靖, 张路, 王鹏宇, 等. 面向图计算的内存系统优化技术综述. 中国科学: 信息科学, 2019, 49: 295–313]

7 Jin H, Yao P, Liao X, et al. Towards dataflow-based graph accelerator. In: Proceedings of IEEE International Conference on Distributed Computing Systems, 2017. 1981–1992

8 Yan M, Li H, Deng L, et al. A survey on graph processing accelerators. J Comput Res Develop, 2021, 58: 862–887

9 Yang Y, Yu H, Zhao J, et al. An efficient hardware accelerator for monotonic graph algorithms on dynamic directed graphs. Sci Sin Inform, 2023, 53: 1575–1592 [杨赟, 余辉, 赵进, 等. 面向动态有向图的单调图算法硬件加速机制. 中国科学: 信息科学, 2023, 53: 1575–1592]

10 Ham T J, Wu L, Sundaram N, et al. Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture, 2016. 1–13

11 Yao P, Zheng L, Huang Y, et al. Scalagraph: a scalable accelerator for massively parallel graph processing. In: Proceedings of IEEE International Symposium on High-Performance Computer Architecture, 2022. 199–212

12 Gonzalez J E, Low Y, Gu H, et al. Powergraph: distributed graph-parallel computation on natural graphs. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, 2012. 17–30

13 Han W, Zhu X, Zhu Z, et al. Weibo, and a tale of two worlds. In: Proceedings of International Conference on Advances in Social Networks Analysis and Mining, 2015. 121–128

14 Hoefler T. November 23 green: small data. 2023. https://graph500.org/?page_id=1244

15 Teixeira C H, Fonseca A J, Serafini M, et al. Arabesque: a system for distributed graph mining. In: Proceedings of Symposium on Operating Systems Principles, 2015. 425–440

16 Beamer S, Asanovic K, Patterson D. Locality exists in graph processing: workload characterization on an ivy bridge server. In: Proceedings of IEEE International Symposium on Workload Characterization, 2015. 56–65

17 Kwak H, Lee C, Park H, et al. What is twitter, a social network or a news media? In: Proceedings of International Conference on World Wide Web, 2010. 591–600

18 Leskovec J, Krevl A. SNAP datasets: Stanford large network dataset collection. 2014. http://snap.stanford.edu

19 Shun J, Blelloch G E. Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013. 135–146

20 Zhang J, Li J. Degree-aware hybrid graph traversal on FPGA-HMC platform. In: Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018. 229–238

21 Zhao J, Jiang X, Zhang Y, et al. An efficient storage system towards high throughput of concurrent graph processing jobs. Sci Sin Inform, 2022, 52: 111–128 [赵进, 姜新宇, 张宇, 等. 一种高效的面向高并发图分析任务的存储系统. 中国科学: 信息科学, 2022, 52: 111–128]

22 Ai Z, Zhang M, Wu Y, et al. Squeezing out all the value of loaded data: an out-of-core graph processing system with reduced disk I/O. In: Proceedings of USENIX Annual Technical Conference, 2017. 125–137

23 Ma L, Yang Z, Chen H, et al. Garaph: efficient GPU-accelerated graph processing on a single machine with balanced replication. In: Proceedings of USENIX Annual Technical Conference, 2017. 195–207

24 Qian X. Graph processing and machine learning architectures with emerging memory technologies: a survey. Sci China Inf Sci, 2021, 64: 160401

25 Mukkara A, Beckmann N, Abeydeera M, et al. Exploiting locality in graph analytics through hardware-accelerated

traversal scheduling. In: Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture, 2018. 1–14

26 Yao P, Zheng L, Zeng Z, et al. A locality-aware energy-efficient accelerator for graph mining applications. In: Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture, 2020. 895–907

27 Dadu V, Liu S, Nowatzki T. Polygraph: exposing the value of flexibility for graph processing accelerators. In: Proceedings of Annual International Symposium on Computer Architecture, 2021. 595–608

A redundancy-aware energy-efficient graph accelerator

Pengcheng YAO[1,2,3,4,5], Xiaofei LIAO[1,2,3,4], Hai JIN[1,2,3,4*], Yuhang ZHOU[1,2,3,4], Peng XU[5], Wei ZHANG[5], Zhen ZENG[5], Chengao PAN[1,2,3,4] & Bing ZHU[1,2,3,4]

1. *National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
2. *Service Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
3. *Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
4. *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China*;
5. *Zhejiang Lab, Hangzhou* 311121*, China*
* Corresponding author. E-mail: hjin@hust.edu.cn

Abstract Graph plays an essential role in a wide range of real-world applications. Due to graph irregularity, general-purpose processors are not an ideal platform for graph processing. Therefore, there has been a significant interest in developing domain-specific accelerators for graph processing in the past few years. With dedicated hardware specialization, graph accelerators can deliver considerable performance speedups compared to CPUs and GPUs. However, existing graph accelerators perform unnecessary accesses on high-degree vertices when running BFS on power-law graphs, resulting in severe off-chip memory overheads. To solve the problem, we architect JiFeng, a redundancy-aware graph accelerator. When a high-degree vertex finishes execution, JiFeng aggressively skips all its edges to avoid redundant memory accesses. Several software/hardware co-designs are proposed to improve memory efficiency and load-balance. We have implemented JiFeng in RTL and evaluated it on a Xilinx Alveo U55C accelerator card. JiFeng achieves at most 461.2 GTEPS throughput and 12.5 GTEPS/W energy efficiency, and ranks 2nd in the SMALL DATA list of GreenGraph500.

Keywords graph processing, accelerator, breadth-first search, redundant memory access, FPGA