中国科学: 信息科学 2023 年 第 53 卷 第 8 期: 1575–1592
SCIENTIA SINICA Informationis
论文

面向动态有向图的单调图算法硬件加速机制

杨赟[1,2,3,4], 余辉[1,2,3,4], 赵进[1,2,3,4], 张宇[1,2,3,4*], 廖小飞[1,2,3,4], 姜新宇[1,2,3,4], 金海[1,2,3,4], 刘海坤[1,2,3,4], 毛伏兵[1,2,3,4], 张吉[5], 王彪[6]

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, 中国
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, 中国
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, 中国
4. 华中科技大学计算机科学与技术学院, 武汉 430074, 中国
5. School of Mathematics, Physics and Computing, University of Southern Queensland, Toowoomba 4350, Australia
6. 之江实验室, 杭州 311121, 中国
* 通信作者. E-mail: zhyu@hust.edu.cn

收稿日期: 2022–05–12; 修回日期: 2022–08–21; 接受日期: 2022–10–27; 网络出版日期: 2023–08–15

国家自然科学基金 (批准号: 61832006, 61825202, 62072193, 61929103)、之江实验室开放课题 (批准号: 2021KD0AB01) 和之江实验室重大科研项目 (批准号: 2022PI0AC03) 资助

摘要 随着现实世界中动态图计算需求的快速增长, 现有的研究工作已经提出了多种方法来有效支持单调图算法在动态有向图中的处理. 然而, 由于动态有向图的图结构频繁发生变化, 其相邻图顶点之间的状态更新存在复杂的依赖关系, 这使得现有的软硬件方法在处理单调图算法时依然面临着数据访问成本高和收敛速度慢的问题. 为此, 本文提出了一种面向动态有向图的单调图算法加速器 DSGraph, 它能够充分利用图顶点之间的依赖关系来加快单调图算法在动态有向图处理中的收敛速度, 并有效降低数据访问成本. 具体来说, DSGraph 通过实时提取动态有向图中图顶点的局部拓扑依赖顺序来执行异步迭代处理, 从而显著减少冗余的图顶点状态更新. 同时, DSGraph 设计了一种异步迭代流水线架构, 其按照依赖顺序对图顶点状态进行异步迭代处理, 从而加速图顶点状态传播速度并减少数据访问开销. 最后, DSGraph 提出了一种无阻塞数据同步机制, 通过并行执行本地图顶点的状态更新和外部图顶点的数据同步来减少系统同步开销. 实验显示, 与目前最先进的面向单调图算法的动态图处理系统 KickStarter 相比, DSGraph 将动态有向图处理速度平均提升了 11.2 倍.

关键词 动态有向图, 单调图算法, 增量计算, 依赖感知, 图加速器

1 引言

现实世界中的有向图结构往往是动态变化的, 称之为动态有向图, 比如社交网络、实时路况地图等. 在动态有向图中, 图更新 (有向图顶点/边的增加/删除) 通常以分批的方式作用于有向图结构

引用格式: 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, doi: 10.1360/SSI-2022-0191
Yang Y, Yu H, Zhao J, et al. An efficient hardware accelerator for monotonic graph algorithms on dynamic directed graphs (in Chinese). Sci Sin Inform, 2023, 53: 1575–1592, doi: 10.1360/SSI-2022-0191

中, 然后基于此更新后的有向图结构 (最新有向图快照) 进行处理分析. 大量图算法的图顶点状态更新算子通常为选择更新函数, 即图顶点状态值是从其所有邻居顶点状态和相应边的计算值中选择得出(如求最大或最小值). 该类图算法被称为单调图算法, 例如单源最短路径算法 (single source shortest paths, SSSP) [1]、强连通分量算法 (strongly connected component, SCC) [2]、单源最宽路径算法 (single source widest paths, SSWP) [3] 和标签传播算法 (label propagation, LP) [4], 它们被广泛用于有向图数据分析 [5].

由于图结构频繁的发生变化, 基于图更新之前 (最近一次有向图快照) 的计算结果与最新图快照需要的结果并不一致, 需要进行处理. 为了能够高效处理基于动态有向图的单调图算法, 目前已经有很多软件系统被提出来. 这些系统通常基于图更新之前 (最近一次有向图快照) 计算出的结果进行增量计算 [6], 来快速获得最新有向图快照的结果. 因为每次图更新批中的顶点和边数量相对于整个有向图结构来说只占一小部分, 而且其只会影响一小部分图顶点, 增量计算在动态有向图中处理单调图算法时通常能够大大减少冗余计算量.

然而, 现有的软件系统在进行增量计算时仍然面临着严重的冗余计算开销和数据访问开销. 受到图更新影响的顶点的状态值通常会不规则地沿着动态有向图拓扑进行传递, 不同的图顶点状态值会分别沿着不同的路径进行传递, 这就导致不同受影响的图顶点状态值会在不同时刻不规则地传递到相同的顶点, 从而产生很多冗余计算开销. 此外, 图计算本身就存在不规则的数据访问特性 [7], 冗余地对同一顶点的计算, 也会引起更多的内存访问, 从而导致更严重的数据访问开销. 动态有向图处理系统 Kineograph [8], GraPU [9] 和 GraphBolt [10] 基于同步迭代模型, 由于存在同步屏障, 这些系统在进行单调图算法处理时会存在大量的冗余图顶点更新. KickStarter [5], GraphIn [11] 和 Tornado [12] 虽然不存在同步屏障, 采用异步迭代模型 [13], 但是也都不可避免地存在着高额地运行时开销和通信开销. 通常地, 在动态有向图中, 单调图算法进行增量计算时的效率由图结构更新 (图顶点/边的增加/删除) 后受影响的图顶点将其影响传播至其他图顶点的传播速度决定, 如果受影响的图顶点能够将其影响沿着图结构中的依赖关系直接传递时, 那么就能够快速收敛. 然而, 现有的动态图处理系统在处理时并没有利用图顶点之间的依赖关系去进行增量处理, 使得图更新产生的影响只能在图结构中缓慢传递, 算法收敛速度较慢.

为了解决软件系统在图处理时的高额开销问题, 最近几年来, 有许多图处理硬件加速器 [14~24] 被提出来. 然而它们大多数都是针对静态有向图而设计实现, 如果要处理动态有向图, 这些加速器需要将动态有向图的最新快照当作静态有向图, 然后在此静态有向图上重新运行整个图计算过程 (即全量计算过程) 来获得最新的计算结果. 例如, 基于混合内存立方体 (hybird memory cube) 的图处理加速器 [14~17] 以及图计算专用加速器 [18~22]. 虽然这些硬件加速器可以通过使用定制化的硬件设计来加快静态有向图的处理速度, 减少运行时开销, 但是在面向动态有向图时, 由于它们采用全量计算, 会产生大量的冗余计算开销和访存开销. 与此同时, 一些动态图处理硬件加速器 (如 JetStream [24]) 被提出来高效支持动态图增量处理. 但是, 现有的这些图处理加速器与动态图有向图中图顶点的状态传播特性并不匹配, 不能很好地满足实时性需求.

为了解决上述问题, 本文提出了一种运行时依赖驱动的增量执行方法. 此方法可以在面向动态有向图的单调图算法处理中实现更加规则化的状态传递, 大大减少冗余计算开销和内存访问开销. 具体地, 此方法的核心思想是, 在图更新之后, 将受图更新影响的图顶点作为根顶点, 然后从根顶点开始, 沿着图拓扑结构的依赖关系按顺序提取图顶点, 生成满足拓扑依赖顺序的图顶点序列, 并按照此图顶点序列来进行单调图算法增量处理. 通过这种方式, 图顶点之间的状态可以更加规则地一次性传递, 大大减少了频繁地加载和处理相关受影响图顶点的状态值. 上面的方法能够有效减少冗余图顶点更新

次数, 但是软件实现方法存在着较高的运行时开销. 因此, 基于上述方法, 本文设计实现了一种硬件加速器 DSGraph, 此加速器通过运行时依赖驱动的增量执行方法来加快单调图算法在动态有向图中进行增量计算时的收敛速度. DSGraph 通过硬件机制来获取满足拓扑依赖顺序的图顶点序列, 然后按照此序列通过定制化的异步迭代流水线来进行增量计算, 从而加速单调图算法在动态有向图上的状态传递. DSGraph 中存在多个并行处理单元, 为了降低不同并行处理单元之间的数据同步开销, DSGraph 通过在每个处理单元内部定义双缓冲区来分别更新和同步本地和外部图顶点状态值, 从而进一步减少随机访问开销, 提高执行效率. 本文通过性能模拟器实现了 DSGraph 的架构, 并对其进行了性能测试. 实验结果显示, DSGraph 平均减少了 67% 的冗余图顶点更新次数, 并且与现有最好的面向动态有向图的单调图算法处理软件系统 KickStarter 相比, 性能平均提升了 11.2 倍.

2 背景与动机

单调图算法的图顶点更新函数为选择函数, 即通过比较所有入边邻居的图顶点状态值 (如使用最大值、最小值或其他比较函数), 然后选择比较结果作为此图顶点状态值. 在动态有向图中, 通常会基于最近一次快照的计算结果通过增量计算的方式来获取最新快照的计算结果. 针对单调图算法, 增量计算的具体过程如下, 对于每一条增加的边 (如 $v_i \to v_j$), 这条边的目的顶点 v_j 状态值首先通过源顶点 v_i 的状态值以及边 $v_i \to v_j$ 的权重计算出来. 然后, v_j 被标记为活跃图顶点, 也即因为增加边受影响的图顶点, 并不断传播其状态值至其他图顶点直到收敛, 从而获得最新快照的计算结果. 对于每一条删除的边 (如 $v_p \to v_q$), 首先通过标签传播 [5] 的方法识别出所有因为删除边而受影响的图顶点 (即与边 $v_p \to v_q$ 的目的顶点 v_q 直接相连或间接相连的顶点). 然后, 所有这些因为删除边而受影响的图顶点的状态值被复位为对应单调图算法的初始值 (如对于单源最短路径算法, 设置为 $+\infty$), 之后, 这些受影响的图顶点根据其所有入边邻居顶点的状态值通过选择更新函数来获取其中间状态值. 这些图顶点接着被标记为活跃图顶点, 也即因为删除边受影响的图顶点, 并不断传播其状态值至其他图顶点直到收敛, 从而获得最新快照的计算结果.

高效地处理基于动态有向图的单调算法变得越来越重要, 目前也已经有大量的动态有向图处理系统被提出来支持单调图算法在动态有向图上高效执行. 动态图处理系统 Kineograph [8] 使用 epoch 提交协议生成动态有向图的最新快照, 并且包含一个增量计算引擎在最新图快照中进行处理. 然而, Kineograph 是通过同步迭代模型 (bulk synchronous parallel, BSP) 来执行增量计算, 图顶点状态值传播缓慢. Tornado [12] 注意到在进行增量计算时, 初始图顶点状态值的不同选取方法能够影响收敛速度, 当要进行图处理时, 通过使用近似图顶点状态值来进行增量计算, 从而加快收敛. 然而, 对于动态有向图中图顶点/边删除的情况, Tornado 这种近似方法无法进行有效处理. KickStarter [5] 采用了一种剪枝的方法来有效处理单调图算法中的图顶点/边删除情况. GraPU [9] 利用了动态有向图的影响只在弱连通分量中传播这一特性, 提出了一种面向单调图算法的动态有向图处理方法. 以上这些软件系统要么采用同步迭代模型 (BSP), 由于同步屏障的存在导致图顶点状态传递缓慢, 要么采用异步执行模型, 存在着高额的运行开销和通信开销, 均不能高效满足动态有向图中单调图算法的处理性能需求.

与此同时, 一些硬件解决方案被提出来以加快动态有向图的处理速度. TESSERACT [15] 和 GraphP [16] 是基于 PIM (processing-in-memory) 的并行图处理架构, 能够利用高内存带宽来高效进行图处理. DREDGE [17] 提出一种硬件动态重划分机制来解决在动态有向图处理中不同处理单元之间的通信开销. AccuGraph [19] 和 ForeGraph [18] 通过利用 FPGA (field programmable gate array) 的可编程性实现了定制化的图计算优化机制, 从而高效支持图处理. 基于 FPGA 的图处理加速器通常

存在主频较低和资源总量较少的特性, 从而限制了其性能. 于是一些基于 ASIC (application specific integrated circuit) 的图处理加速器相继被提出. 为了高效地支持图处理, Graphicionado [20] 基于 GAS (gather-apply-scatter) 执行模型, 设计实现了基于同步迭代模型的多级流水线架构, 来并行地进行图处理, 并且将图顶点状态值存储在片上存储单元中, 从而消除状态值随机访问开销. GraphDynS [21] 提出了一种新的编程模型以及微架构设计, 对图处理中的数据通路进行解耦, 通过数据驱动的动态调度策略来加速图处理过程. GraphPluse [22] 提出了基于事件驱动的计算模型的异步迭代加速器, 该加速器实现了并行的数据流执行以显著加快收敛速度. 然而, 以上这些图处理加速器都是针对静态有向图而提出的, 如果要处理动态有向图, 只能将动态有向图的最新图快照看作一个全新的静态有向图, 然后在此静态有向图重新执行整个计算过程来获得最新图快照的计算结果, 从而导致大量的冗余计算开销和数据访问开销, 特别是对于大图来说. JetStream [24] 是第一个动态图处理硬件加速器, 它通过将图更新转化为增量来加快动态图处理过程, 然而图顶点之间的复杂依赖仍然导致其图顶点状态传递缓慢.

综上, 现有的软硬件方法都不能够从本质上高效地处理基于动态有向图的单调图算法, 对于图更新产生的影响, 都不能快速传递图顶点状态值. 因此, 本文提出了一种高效的硬件加速器, 通过采用依赖驱动的增量执行方法来加快单调图算法在进行增量计算时的收敛速度, 从而解决上述难题.

3 面向动态有向图的单调图算法增量计算加速器

3.1 运行时依赖驱动的增量执行方法

运行时依赖驱动的增量执行方法可以规则化图更新之后受影响图顶点的状态传递, 从而减少冗余计算. 此方法根据图拓扑结构来获得受影响图顶点和其后继顶点之间的依赖关系, 基于依赖关系, 从受影响的图顶点开始, 沿着图拓扑结构来驱动增量计算.

具体地, 对于每一个受图更新影响的图顶点, 此方法首先根据图拓扑结构记录经过此图顶点的状态传播信息, 然后, 对于每一个受影响的图顶点, 当所有需要经过它的状态传播已经被聚集后, 把此顶点作为根顶点, 然后沿着图拓扑通过深度优先搜索[1)]的方式按照依赖关系获取其他顶点, 并按此获取的图顶点依赖序列来进行增量处理. 通过这种方式, 每一个受影响的图顶点都能够在其前继受影响图顶点计算完成后, 使用最新的图顶点状态进行计算, 然后传播最新的图顶点状态值给其所有后继邻居. 这种方法可以避免频繁地对同一图顶点进行加载和处理, 减少了冗余计算开销和内存访问开销. 与此同时, 受影响图顶点地状态值能够快速沿着图拓扑进行传递. 并且此方法对于底层图存储格式透明, 不需要修改底层图存储格式.

如算法 1 所示, 算法输入中, $G(V, E)$ 表示图拓扑结构, 其中 V 为图顶点集合, E 为边集合, S 为图顶点状态值集合, ActiveSet 表示活跃顶点集合, 在动态有向图处理中, 将受图更新直接影响的图顶点作为活跃顶点, 每个图顶点存在一个 TraversalTimes 值, 初始时为 0, 表示当前图顶点会被遍历的次数, WorkQueue 为生成的需要进行增量处理的边序列, 算法输出增量计算后的图顶点状态值 S. 针对活跃图顶点集合 ActiveSet 中的每一个活跃图顶点, 首先执行 DependencyTracking 函数, 将其作为根图顶点进行有限深度优先遍历, DependencyTracking(v, depth) 函数以图顶点 v 为根进行深度优先遍历, 并在遍历的过程中更新 TraversalTimes, 即遍历到一个图顶点, 就将其遍历次数加一, 当所有的活跃图顶点都遍历完之后, 便可以得到受图更新影响的图顶点之间的依赖信息 TraversalTimes. 在动态有向图处理中, 可以根据 TraversalTimes 值的大小来确定图顶点的处理顺序, 将 TraversalTimes 值

1) 深度优先搜索可以使用一个固定深度的栈来实现几乎与更深深度的栈相同的性能, 只需要较小的硬件成本. 然而, 宽度优先搜索需要一个较大的 FIFO 队列才能保证其有效性.

算法 1 Dependency-driven incremenal execution approach

Input: $G(V,E), S$, ActiveSet, TraversalTimes, WorkQueue.
Output: S.

```
1: procedure DependencyTracking(v, depth)
2: for u ∈ v.outNeighbors do
3:     TraversalTimes[u] + +;
4:     if depth < maxDepth then
5:         if edge v → u is not visited then
6:             DependencyTracking(u, depth + 1);
7:         end if
8:     end if
9: end for
10: end procedure
11: procedure GraphDataFetching(v, depth)
12: for u ∈ v.outNeighbors do
13:     TraversalTimes[u] − −;
14:     WorkQueue.push(v → u);
15:     if depth < maxDepth then
16:         if edge v → u is not visited then
17:             GraphDataFetching(u, depth + 1);
18:         end if
19:     end if
20: end for
21: end procedure
22: for each root ∈ ActiveSet do
23:     DependencyTracking(root, 0);
24: end for
25: for each root ∈ ActiveSet do
26:     if TraversalTimes[root] is minimum then
27:         GraphDataFetching(root, 0);
28:     end if
29: end for
30: IncrementalCompute(WorkQueue, S).
```

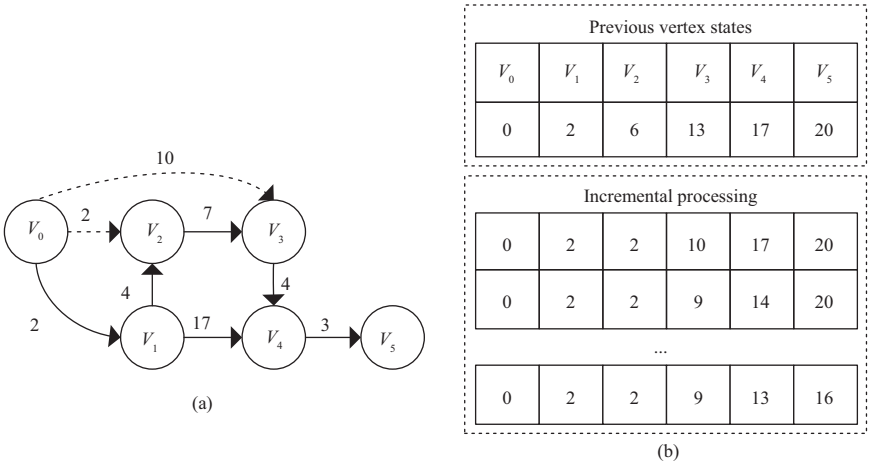

图 1 依赖驱动的增量执行方法示例图

Figure 1 An example of dependency-driven incremenal execution approach. (a) An example graph; (b) the incremental processing of SSSP

较小的图顶点称作高拓扑顺序的图顶点, 优先处理高拓扑顺序的图顶点能够快速传播受图更新影响的图顶点状态值至整个图拓扑. 所以接下来, 从活跃图顶点集合 ActiveSet 中每次选取 TravesalTimes 值最小的图顶点调用 GraphDataFetching 函数进行有限深度优先遍历, 从而获得满足图拓扑依赖顺序的图顶点序列, GraphDataFetching(v, depth) 函数以图顶点 v 为根进行深度优先遍历, 每当遍历到一个图顶点的时候, 将 TraversalTimes 值减一, 并将其对应的边添加到 WorkQueue 中, 当所有的活跃图顶点都遍历完成之后, 便可以得到满足受图更新影响满足拓扑依赖顺序的边序列 WorkQueue, 然后 IncrementalCompute$(\text{WorkQueue}, S)$ 按照 WorkQueue 中的边序列来进行增量处理, 更新图顶点状态 S.

我们通过一个例子来显示此方法及其有效性. 如图 1 所示, 假设在一次图更新中增加了边 $v_0 \to v_2$ 和 $v_0 \to v_3$, 生成了最新的有向图快照, 那么受影响的图顶点为 v_2 和 v_3, 首先将 v_2 和 v_3 设置为活跃

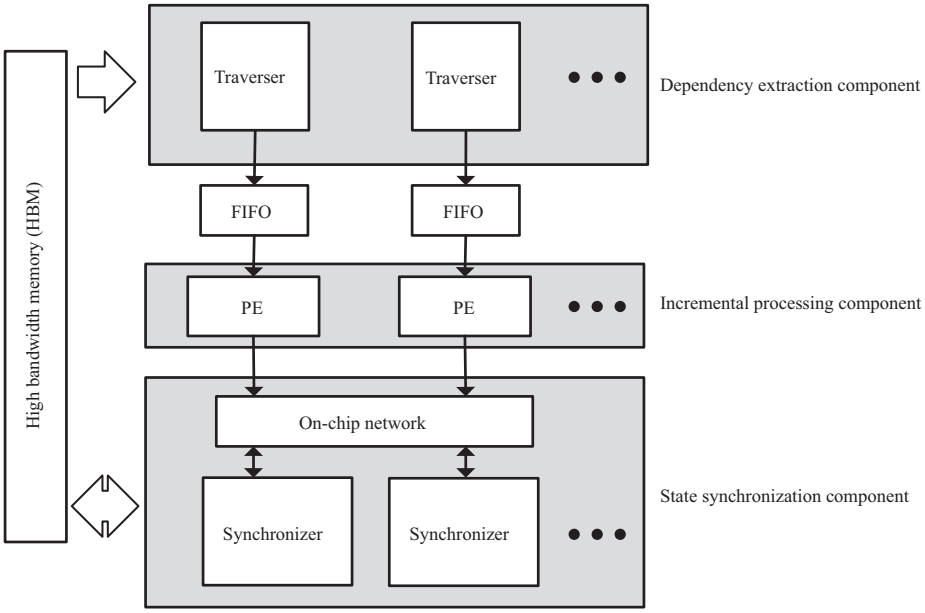

图 2 DSGraph 系统架构

Figure 2 System architecture of DSGraph

顶点, 然后对活跃图顶点 v_2 和 v_3 分别执行 DependencyTracking 函数, 将 v_2 和 v_3 的 TraversalTimes 值更新为 0 和 1. 接下来选择 TraversalTimes 值较小的图顶点 v_2, 执行 GraphDataFetching 函数获取图顶点依赖序列, 即 $v_2 \to v_3$, $v_3 \to v_4$, $v_4 \to v_5$, 最后, IncrementalCompute 函数依照此依赖序列进行异步增量处理. 由于图顶点 v_2 和 v_3 之间存在依赖, 当源自 v_2 的图顶点状态传播到 v_3 之后, 受影响的图顶点 v_3 再进行后续处理, 因此, 图顶点 v_2 和 v_3 的状态能够一起传递到他们的后继图顶点 v_4 和 v_5, 图顶点 v_4 和 v_5 的状态也只需要加载并且计算一次, 否则, 如果先处理图顶点 v_3, 再处理图顶点 v_2, 会导致图顶点 v_4 和 v_5 被多次加载和计算.

3.2 DSGraph 设计和实现

运行时依赖驱动的增量执行方法如果通过软件方法实现, 会引入较大的运行时开销, 从而抵消其所能带来的性能提升. 因此, 本文设计实现了一种面向动态有向图的单调图算法加速器 DSGraph. DSGraph 基于依赖驱动的增量执行方法, 设计实现了专门的硬件架构, 它能够以低运行时开销, 根据图更新之后受影响的图顶点, 实时感知和提取满足拓扑依赖关系的图顶点序列, 并按照此拓扑序列进行单调图算法的增量处理. 此加速器需要配合主控制器, 主控制器完成动态有向图的更新部分以及初始化部分. 图 2 显示了 DSGraph 的系统架构, 整体架构可以分为 3 大核心组件: 依赖提取组件、增量处理组件和状态同步组件. 各个组件的结构和功能如下.

依赖提取组件 (dependency extraction component). 依赖提取组件是运行时依赖驱动增量执行方法的核心组件, 如算法 1 所示, 其执行过程主要分为两个阶段, 即依赖信息获取阶段和图数据提取阶段. 由于两个阶段具有相同的执行步骤, 都是通过深度优先搜索的方式遍历有向图结构, 故可以复用同一套硬件逻辑. 依赖提取组件由多个遍历单元 (traverser) 组成, 每个遍历单元负责一块图分区, 从高带宽存储器 (high bandwidth memory, HBM) 中访问对应的动态有向图结构数据, 通过两阶段深度优先遍历获取满足拓扑依赖顺序的序列. 每个遍历单元将获取的边序列通过队列 (first in first out,

FIFO) 的方式发送给增量处理组件进行增量处理.

增量处理组件 (incremental processing component). 增量处理组件从与依赖提取组件相连的 FIFO 中获取待处理的边序列进行增量处理, 更新图顶点状态值. 增量处理组件包含多个处理单元 (processing element, PE), 每个处理单元对应依赖提取组件中一个遍历器的 FIFO. 处理单元对于每一条边数据, 根据源图顶点状态值以及对应的单调图算法定义的更新函数, 更新目的图顶点状态值. 此外, 处理单元从状态同步组件中的片上存储区中读取所需的图顶点状态, 避免了直接访问内存而带来的随机访问开销. 处理单元中包含单调图算法常见的运算模块, 如求最大值、求最小值、求和等. 用户可以根据对应单调图算法的需求通过 API 调用对应的运算模块.

状态同步组件 (state synchronization component). 状态同步组件用来同步不同片上存储单元之间的图顶点状态数据. 状态同步组件包含多个片上存储单元, 用来减少图顶点状态数据的随机访问开销. 每个片上存储单元分为两部分, 本地存储区和外部存储区, 本地存储区与增量处理组件中的一个处理单元对应, 存储当前图分区的所有图顶点状态, 外部存储区用来存储来自其他处理单元通过片上网络 (on-chip network) 发送过来的图状态数据, 并通过同步单元 (synchronizer) 将其同步到本地存储区中, 片上存储单元通过片上网络与增量处理组件中的处理单元相连. 一般情况下, 图顶点状态数据可完全存放到片上存储单元中, 对于大规模动态有向图, 类似 JetStream [24], DSGraph 将动态有向图快照的图顶点划分为多个连续区段, 每个图划分块是一个区段的图顶点及其相关边, 从而保证了每个图划分块对应的图顶点状态数据可以完全存放到片上存储单元中.

3.3 DSGraph 执行流程

DSGraph 的执行流程可以分为 4 个阶段: 初始化阶段、依赖提取阶段、增量处理阶段和数据同步阶段. 其中依赖提取阶段、增量处理阶段和数据同步阶段通过解耦可以以流水线的方式进行工作, 减少了整个加速器的关键路径. 各个阶段的执行流程如下.

初始化阶段. 初始化阶段用于初始化进行单调图算法处理所需的各种数据, 需要加速器与主控制器端相互配合完成. 当一批图更新操作来临时, 主控制器端首先进行图更新, 生成最新有向图快照, 除此之外, 主控制器根据图更新将受影响的顶点设为活跃顶点, 向 HBM 对应的位置上写入活跃顶点集合, 最近一次图快照的计算结果作为初始图顶点状态由主控制器和加速器共同维护在加速器片上存储单元中. 加速器可以直接从对应存储位置上获取有向图结构、活跃图顶点集合和图顶点状态. 加速器自身在 HBM 中维护每个顶点被遍历的次数 (即 TraversalTimes 值). 活跃图顶点集合被均匀地分配给依赖提取模块的不同遍历单元, 以实现不同处理单元之间的负载均衡. 初始化完成之后, DSGraph 便可以根据初始化数据开始工作.

依赖提取阶段. 依赖提取阶段用来获取满足拓扑依赖关系的处理序列, 包括两个步骤: 依赖信息获取和图数据提取. 首先, 通过有限深度优先搜索的方式去遍历最新有向图快照来获得图拓扑依赖信息. 然后根据此依赖信息来提取满足拓扑依赖关系的处理序列. 具体地, 每一个遍历单元每次从当前活跃顶点集合中选取高拓扑顺序的活跃顶点 (即具有较小 TraversalTimes 值的活跃顶点) 按照有限深度优先遍历的方式预取边存放到队列中. 增量处理组件从队列中按顺序取出边进行增量处理.

增量处理阶段. 增量处理阶段用来根据获得地满足拓扑依赖顺序的边序列进行单调图算法的增量计算. 增量处理组件中存在多个处理单元, 每个处理单元和依赖提取组件的遍历单元通过队列相连. 处理单元并发地从队列中获取边数据, 对于每一条边, 根据用户自定义的相应单调图算法指定的更新函数, 从对应的状态同步组件的片上存储单元中读取并更新源顶点的图顶点状态值, 并将状态增量传播至目的顶点. 增量处理阶段不断从队列中获取边进行增量处理, 直至队列为空.

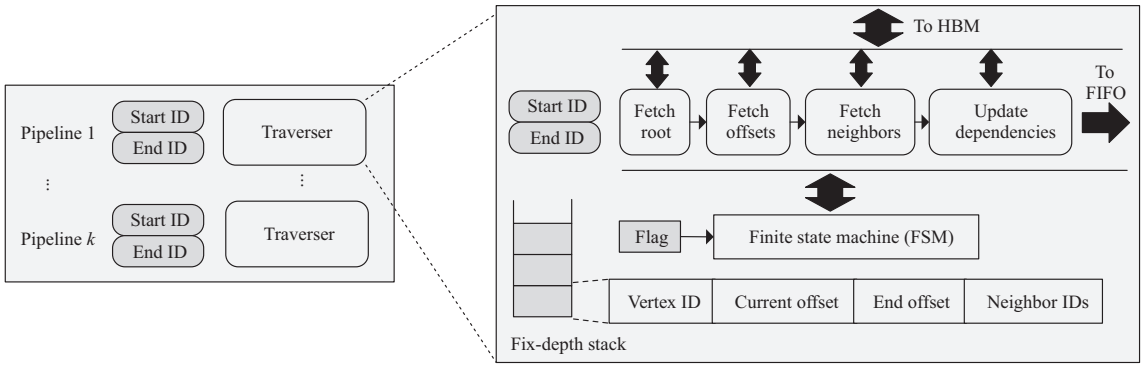

图 3 DSGraph 依赖提取组件架构

Figure 3 Architecture of the dependency extraction component of DSGraph

数据同步阶段. 数据同步阶段用于同步不同处理单元更新后的图顶点状态. 图顶点状态值被划分到多个片上存储单元中, 每个处理单元对应一个片上存储单元. 增量处理中每条边的目的顶点状态值可能随机分布在不同片上存储单元中, 当某个处理单元所处理边的目的顶点状态值存放在其片上存储单元中, 则直接在其本地存储区进行图顶点更新. 否则, 通过片上网络发送到其他处理单元的外部存储区中, 由同步单元将其同步到对应的本地存储区中.

3.4 依赖提取组件设计细节

依赖提取组件由多个遍历单元组成, 每个遍历单元负责动态有向图的一个图数据块. 当图数据块分配给对应的遍历单元之后, 遍历单元通过依赖信息获取和图数据提取两个阶段来获得对应的满足依赖关系的图数据. 遍历单元首先获取图拓扑依赖信息, 以此识别出高拓扑顺序的图顶点, 加速增量处理收敛速度, 然后根据拓扑依赖信息沿着图拓扑预取待处理的图结构数据. 由于依赖信息获取阶段和图数据提取阶段都通过有限深度优先搜索 (默认深度为 10[2)]) 的方式进行图结构遍历, 所以这两个阶段共享同一套硬件资源. 为了区别当前所处的阶段, DSGraph 在遍历单元内部使用了一个标志位 (初始值为 0), 当标志位为 0 时, 则表示遍历单元当前正处于依赖信息获取阶段, 当标志位为 1 时, 则表示当前处于图数据提取阶段. 有限深度优先搜索需要一个栈结构用来维护遍历过程中的信息. DSGraph 的每个遍历单元都维护了一个固定深度的硬件栈结构来存储深度优先搜索的中间信息. 如图 3 所示, 栈的每一层存储一个图顶点的以下信息: (1) 顶点 ID; (2) 顶点未访问的边的偏移数组区间; (3) 一个缓存行大小的当前顶点未访问的邻居顶点 ID. 依赖信息获取阶段和图数据提取阶段的具体硬件实现如下.

依赖信息获取. 为了高效地获取当前有向图数据块的依赖信息, 如图 3 所示, DSGraph 通过以下 4 个阶段来进行深度优先遍历: 获取根图顶点 (fetch root)、获取偏移量 (fetch offsets)、获取邻居顶点 (fetch neighbors) 和更新依赖信息 (update dependencies), 这 4 个阶段被设计成多级流水线并通过有限状态机 (finite state machine, FSM) 的方式进行管理, 以充分利用内存级并行, 缩短数据访问的延迟. 一开始, 当硬件栈结构为空时, 获取根图顶点阶段从活跃顶点列表中取出一个图顶点作为遍历的根顶点, 将此顶点存放到栈中. 获取偏移量阶段读取当前栈顶的图顶点 ID, 然后从图结构的偏移数组中读取邻居顶点偏移量信息, 存放到当前栈顶. 在获取邻居顶点阶段, 状态机从邻居顶点数组中获取

2) 通过测试 DSGraph 在采用不同栈深度时的性能变化情况发现, 当栈的深度增加到 10 之后, DSGraph 的性能提升基本不变或者放缓. 因此我们将深度优先搜索的深度 (栈的深度) 默认设置为 10.

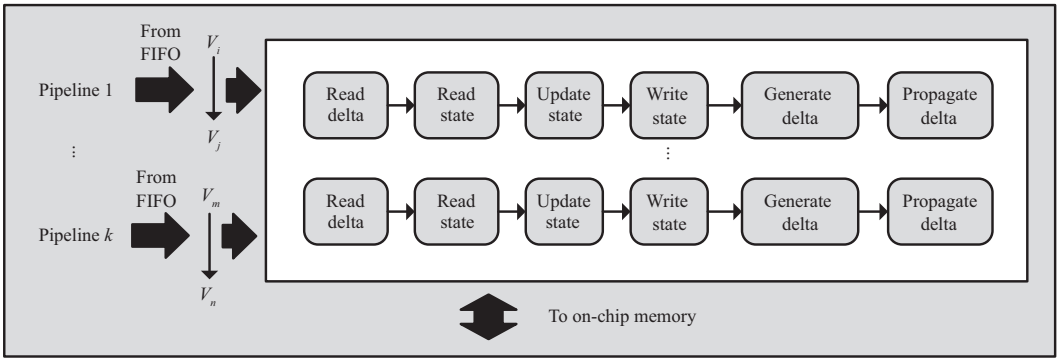

图 4 DSGraph 增量处理组件架构

Figure 4 Architecture of the incremenal processing component of DSGraph

一个缓存行大小的未访问过的邻居顶点, 将其放到栈顶. 之后, 在更新依赖信息阶段, 从当前栈顶图顶点的邻居顶点列表中取出一个邻居图顶点, 将其压入栈中作为新的栈顶, 同时将其遍历次数信息 (即 TraversalTimes) 加 1, 表示此次遍历经过当前图顶点一次. 在最后一个阶段, 如果栈顶图顶点是活跃顶点或者没有未访问的邻居顶点, 则弹出栈顶元素, 从新的栈顶图顶点开始, 将其作为根图顶点重复上述步骤遍历其未访问过的邻居顶点. 以上过程不断重复, 直到所有活跃图顶点即其后继顶点的边被访问过. 最后, 遍历次数信息即为获取的图顶点之间的依赖信息.

图数据提取. 为了高效地根据获取到的依赖信息进行图数据提取, DSGraph 同样使用有限状态机的方式进行深度优先遍历, 获取满足依赖顺序的边序列. 图数据提取阶段复用依赖信息获取阶段的状态机, 其遍历过程也分为 4 个阶段: 获取根图顶点 (fetch root)、获取偏移量 (fetch offsets)、获取邻居顶点 (fetch neighbors) 和更新依赖信息 (update dependencies). 一开始, 当硬件栈结构为空时, 获取根图顶点阶段从活跃顶点列表中根据依赖信息选取具有高拓扑顺序 (即 TraversalTimes 值小) 的图顶点作为遍历的根顶点, 将此顶点存放到栈中, 然后将其设置为非活跃状态. 获取偏移量和获取邻居顶点阶段的执行过程和依赖信息获取阶段相同. 在更新依赖信息阶段, 从当前栈顶图顶点的邻居顶点列表中取出一个邻居图顶点, 将其压入栈中作为新的栈顶, 同时将其遍历次数信息 (即 TraversalTimes) 减 1, 除此之外, 将其对应的边存放到与增量处理组件的处理单元相连的队列中, 供处理单元进行处理. 在最后一个阶段, 如果栈顶图顶点的遍历次数信息大于零或者没有未访问的邻居顶点, 则弹出栈顶元素, 从新的栈顶图顶点开始, 将其作为根图顶点重复上述步骤遍历其未访问过的邻居顶点. 以上过程不断重复, 直到当前有向图数据块中的所有活跃图顶点都被访问过.

3.5 增量处理组件设计细节

增量处理组件中的每个处理单元并发地从其指定队列中获取待处理的边进行单调图算法的增量处理. 每个处理单元采用多级流水线架构来加速图顶点状态值的更新和传递, 充分利用内存级并行, 减少数据访问时延. 如图 4 所示, 其主要包括读取源顶点状态增量、读取源顶点状态、利用增量更新当前源顶点状态、写回源顶点状态、生成目的顶点增量, 传递增量 6 个阶段. 针对队列中的每一条待处理边 $v_i \to v_j$, 首先是更新源顶点 v_i 的状态, 处理单元从片上存储单元读取源顶点的状态增量 ΔS_i 和状态值 S_i, 然后利用 ΔS_i 和 S_i, 使用用户定义的相应单调图算法的规约操作 $\oplus$ (如单源最短路径算法的规约操作为求最小值) 更新源顶点的状态 S_i, 即 $S_i^{\text{new}} = S_i^{\text{old}} \oplus \Delta S_i$. 更新完成之后, 将其写回对应的片上存储单元中. 在更新源顶点状态的过程中源顶点状态的变化量为 Delta_i. 接着, 流水线使

用源顶点的状态变化量 Delta_i 根据用户定义的相应单调图算法的状态传播操作 $g_{\{i,j\}}(\text{Delta}_i)$ (如单元最短路径算法的状态传播操作为状态变化量本身) 生成目的顶点的增量 ΔS_j, 即 $\Delta S_j = g_{\{i,j\}}(\text{Delta}_i)$. 生成的目的顶点的增量需要传递到对应的片上存储单元中, 如果目的顶点状态值存储在当前处理单元对应的片上存储单元中, 则直接通过 "读 – 修改 – 写" 原子操作将增量直接合并到本地存储区中, 如果存储在其他处理单元对应的片上存储单元中, 则通过片上网络发送到对应片上存储单元的外部存储区中, 由状态同步组件将其进行合并.

3.6 状态同步组件设计细节

状态同步组件中包含多个片上存储单元, 图顶点状态值被均匀地分配到各个片上存储单元中. 每个片上存储单元被划分为两部分, 即本地存储区和外部存储区. 本地存储区直接接收当前处理单元对本地图顶点状态增量进行的 "读 – 修改 – 写" 原子操作, 其他处理单元通过片上网络发送过来的图顶点状态增量将以二元组 $(v_i, \Delta S_i)$ (即顶点 ID 和对应的状态增量) 的形式存储在片上存储单元的外部存储区中. 每个片上存储单元在状态同步组件中对应一个同步单元, 同步单元不断地扫描外部存储区, 如果外部存储区非空, 则读取相应的的状态增量并通过规约操作 $\oplus$ 合并更新到本地存储区中. 对于外部存储区中状态增量的同步操作, 同样采用 "读 – 修改 – 写" 原子操作来进行. 通过这种方式, 减少了随机访问开销以及不同片上存储单元中的数据竞争.

4 实验与结果

4.1 实验环境

系统建模. 本文设计实现了一个精确到周期的模拟器对 DSGraph 的依赖提取组件、增量处理组件和状态同步组件进行建模. 通过模拟器运行单调图算法获得的周期数来反映加速器的执行时间. DSGraph 包括 8 个频率为 1 GHz 的计算单元 (一个计算单元包括依赖提取组件中的一个遍历单元、增量处理组件中的一个处理单元以及状态同步组件中的一个同步单元), 每个遍历单元存在 32 kB 大小的数据缓存, 采用 GRASP [25] 策略对依赖信息以及图数据进行管理. 总的片上存储为 64 MB 的 eDRAM (1 GHz, 0.8 ns latency), HBM 的带宽为 128 GB/s.

比较对象. 本文将 DSGraph 与现有的面向单调图算法的软件动态有向图处理系统 KickStarter [5] 进行比较, 之所以选择 KickStarter 是因为它是目前最先进的面向单调图算法的软件动态有向图处理系统. KickStarter 的运行测试环境为: 两个八核英特尔至强 E5-2670 处理器, 每个处理器的 Last-level-cache 的大小为 20 MB, 内存大小为 64 GB. 除此之外, 本文还与目前最先进的动态有向图硬件加速器 JetStream [24] 进行了比较. 为了验证依赖驱动的增量执行方法带来的性能提升, 本文基于 DSGraph, 移除了其依赖提取组件, 仅使用增量计算对活跃顶点进行处理, 实现了加速器 DSGraph-w, 并将其与 DSGraph 进行比较. 同时, 也测试了 DSGraph 基于普通内存 (4 × DDR3, 带宽为 16 GB/s) 实现 (即 DSGraph-m) 时的性能.

算法和数据集. 本文使用了 3 个常见的单调图算法, 包括 weakly connected components (WCC), SSSP 和 SSWP. 数据集如表 1 所示. 基于此有向图数据集, 通过现有动态有向图系统常用的方法 [5, 10], 构造出动态有向图来进行实验测试. 具体地, 对于每一个有向图数据集, 先使用前 50% 的边构造出基础有向图快照, 然后将剩下的 50% 的边看作是图更新中的增加边, 图更新中的删除边从已加载的有向图中进行随机选取, 将图更新中增加边和删除边组合成动态有向图中的更新批. 对于图更新操作, 本文使用 GPMA [26] 中提出的方法来进行实现, 更新完成后, 将图格式转化为 CSR 格式用于增量计算.

表 1 图数据集 ($|V|$ 表示图顶点个数、$|E|$ 表示边条数、d 表示图的直径、$\overline{D}$ 表示平均图顶点度数)

Table 1 Graph datasets ($|V|$ is the number of vertices, $|E|$ is the number of edges, d is the graph diameter and $\overline{D}$ is the average vertex degree)

Datasets	$\|V\|$	$\|E\|$	d	$\overline{D}$
NotreDame	325729	1497134	9.4	17
Google	916428	5105039	15.3	29
BerkStan	685231	7600595	4.9	6
LiveJournal	4847571	68993773	20.5	76

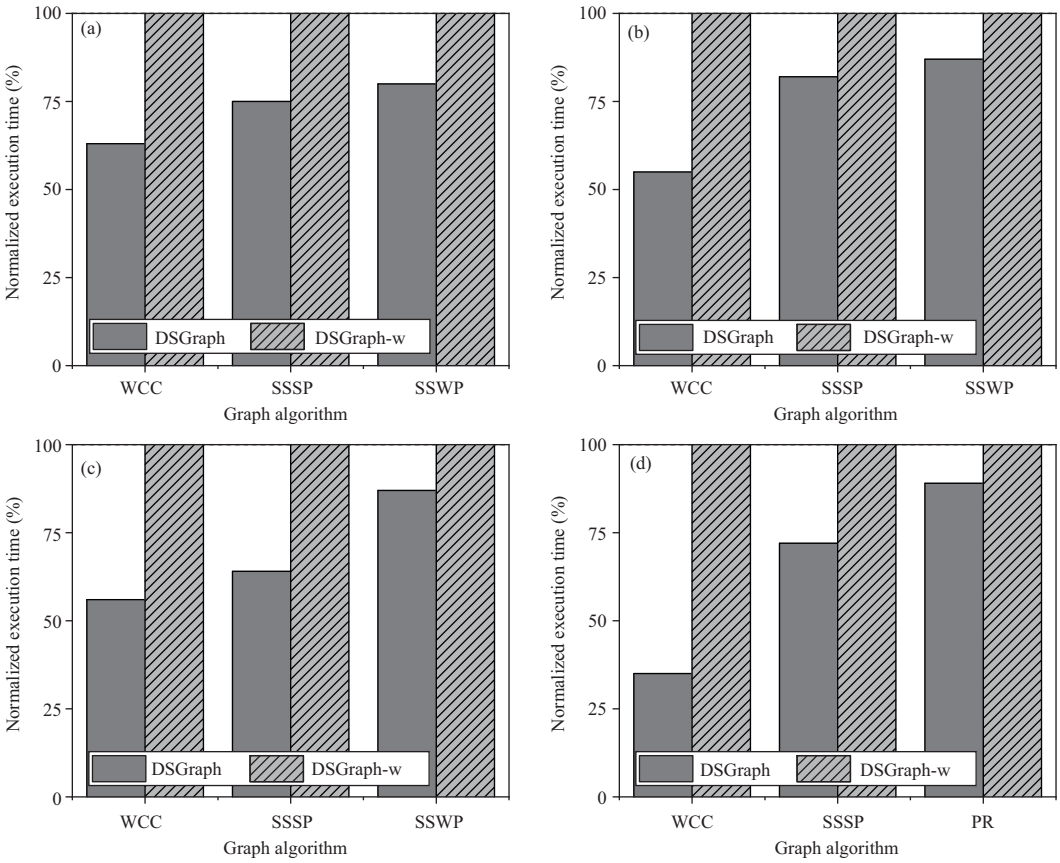

图 5 标准化的动态有向图单调图算法增量计算执行时间

Figure 5 Normalized execution time for incremental processing of dynamic directed graphs. (a) NotreDame; (b) Google; (c) BerkStan; (d) LiveJournal

4.2 综合性能测试

4.2.1 DSGraph 性能测试

图 5 比较了 DSGraph 相对于 DSGraph-w 分别在 4 个动态有向图数据集下进行单调图算法增量计算时的执行时间. 由图可以看出, DSGraph 相较于 DSGraph-w 在不同的数据集和算法下实现了平均 1.5 倍 (1.12~2.86 倍) 的性能提升, 能够更快达到收敛状态. DSGraph 更快的收敛速度来源于其使用硬件加速的方式实现了运行时依赖驱动的增量执行方法. 运行时依赖驱动的增量执行方法通过在

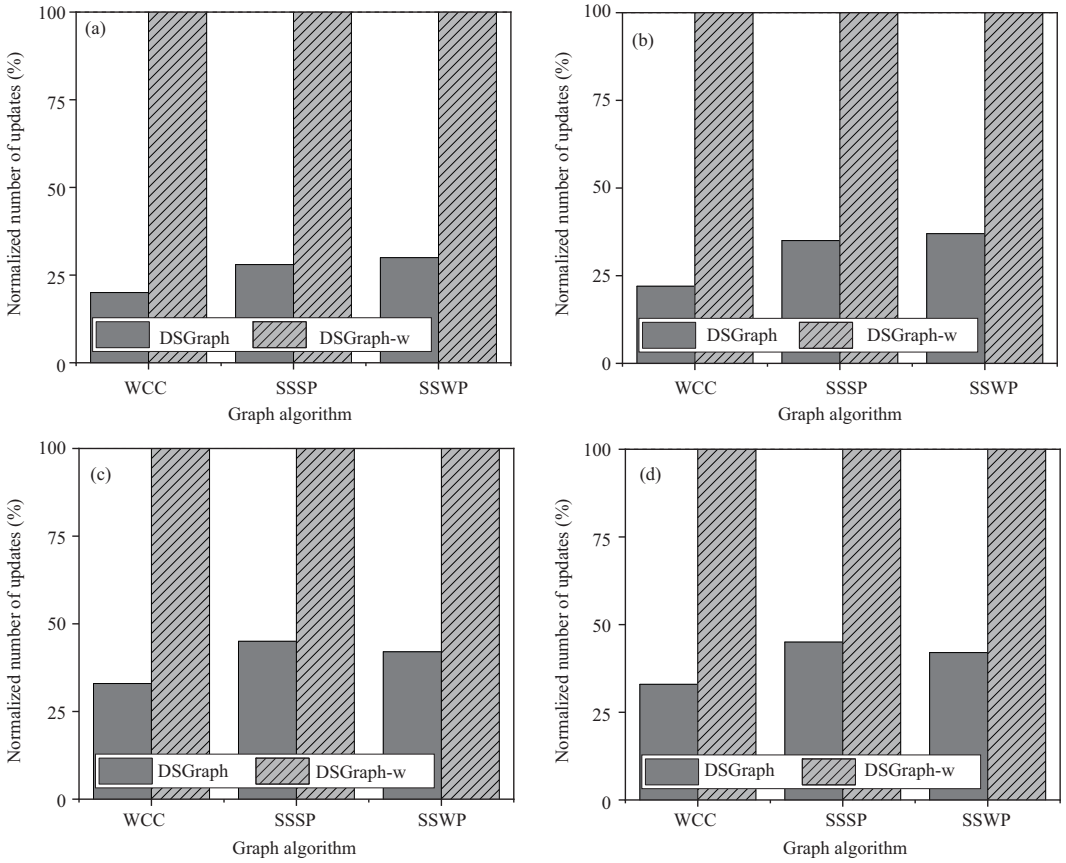

图 6　标准化的图顶点更新次数

Figure 6　Normalized number of vertex state updates. (a) NotreDame; (b) Google; (c) BerkStan; (d) LiveJournal

从受图更新影响的图顶点开始, 在运行时根据图拓扑结构来获得图顶点之间的依赖关系, 然后基于此依赖关系来驱动增量计算, 加快了图顶点状态传递, 减少了冗余计算开销和数据访问开销. 除此之外, DSGraph 的依赖提取组件和增量计算组件之间通过队列实现了解耦, 从而依赖提取组件可以和增量计算组件并行地进行依赖信息提取和增量处理, 减少了运行时开销. 实验显示, 依赖提取组件的运行时开销占总执行时间的 23.7%.

4.2.2　图顶点更新次数测试

运行时依赖驱动的增量执行方法可以提高计算效率的一个最直接的原因就是图顶点更新次数的减少. 为了验证, 本文通过实验比较了 DSGraph 相较于 DSGraph-w 在不同数据集和算法的情况下的图顶点更新次数. 为了便于统计, 配置 DSGraph 为一个计算单元. 图 6 显示了标准化后的结果. 结果显示, 相比于 DSGraph-w, DSGraph 需要更少的图顶点更新次数, DSGraph 在不同数据集和算法的情况下, 图顶点更新次数平均减少了 67.3% (55.4%~81.7%). 因为运行时依赖驱动的增量执行方法, 图顶点的增量计算会沿着拓扑依赖信息进行顺序处理, 每个图顶点都能传递其最新的图顶点状态进行增量计算, 从而减少了一个图顶点被反复处理的情况发生, 有效减少了冗余图顶点更新次数.

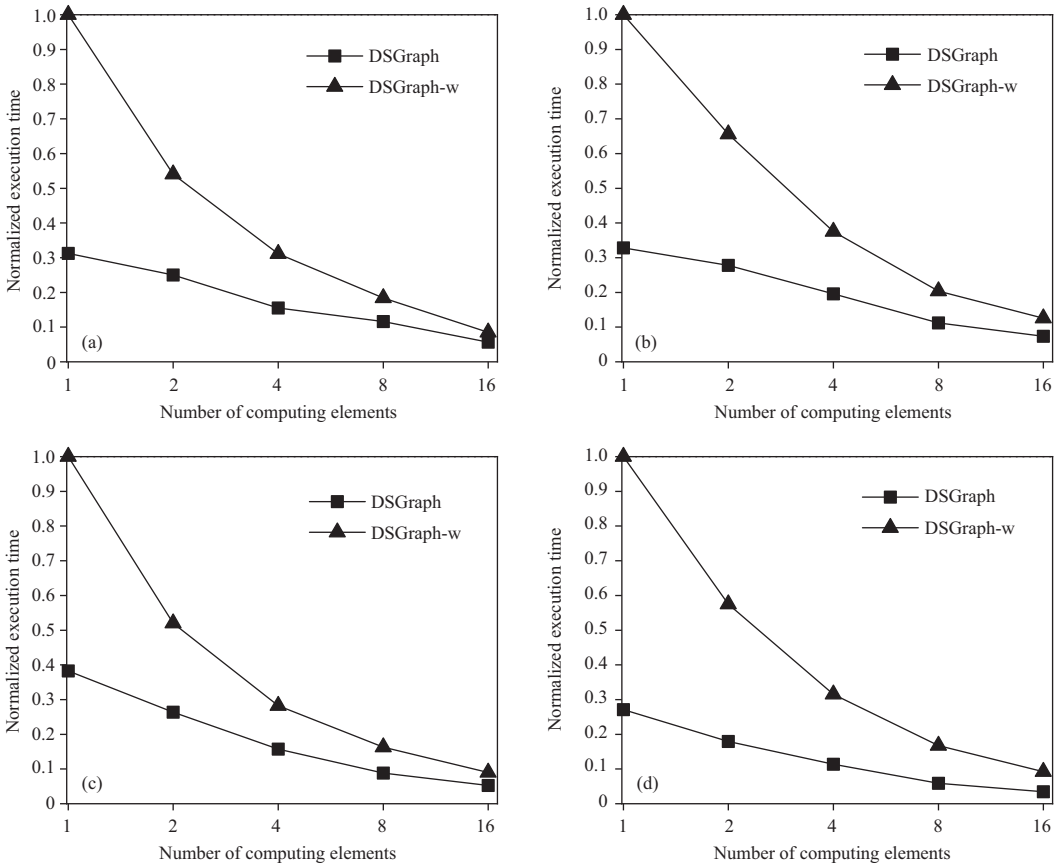

图 7 不同有向图数据集上 WCC 的扩展性

Figure 7 Scalability of WCC on different directed graphs. (a) NotreDame; (b) Google; (c) BerkStan; (d) LiveJournal

4.2.3 DSGraph 扩展性测试

不同的计算单元数量也会对 DSGraph 的性能产生影响, 为此, 图 7 测试了 DSGraph 和 DSGraph-w 在不同计算单元数量的情况下在 4 个动态有向图数据集上运行 WCC 算法的执行时间. 选择 WCC 算法的原因是 WCC 算法是最常见的单调图算法且在每一轮迭代时拥有较多的活跃图顶点, 避免了活跃顶点数过少而影响实验的准确性. 实验分别配置了 DSGraph 的计算单元数量为 1, 2, 4, 8 和 16 个. 实验结果显示, 随着计算单元数量的增加, DSGraph 的执行时间不断缩短. 除此之外, 可以看出, 随着计算单元的数量增加, 性能提升逐渐放缓, 这是由于状态同步组件执行时间占总执行时间的比例在逐渐增加. 但是, DSGraph 能够通过依赖驱动的增量执行方法有效减少冗余图顶点状态更新和内存访问开销以及不同图分区之间的通信次数, 保证了较好的扩展性.

4.2.4 与软件动态有向图方法的性能比较

图 8 比较了 DSGraph, DSGraph-w, DSGraph-m 和软件动态有向图系统 KickStarter [5] 在不同数据集和算法情况下的性能. 实验数据以 KickStarter 系统的执行时间作为基准, DSGraph, DSGraph-w 和 DSGraph-m 的执行时间基于 KickStarter 进行标准化. 由图可知, 相比于 KickStarter, DSGraph-w 的执行效率平均提升了 6.2 倍. DSGraph-w 相较于 KickStarter 的性能提升主要来源于硬件加速机制, 采用了专用的硬件处理单元 (多级流水线) 和数据同步单元, 并且使用片上存储单元来存储所有图顶

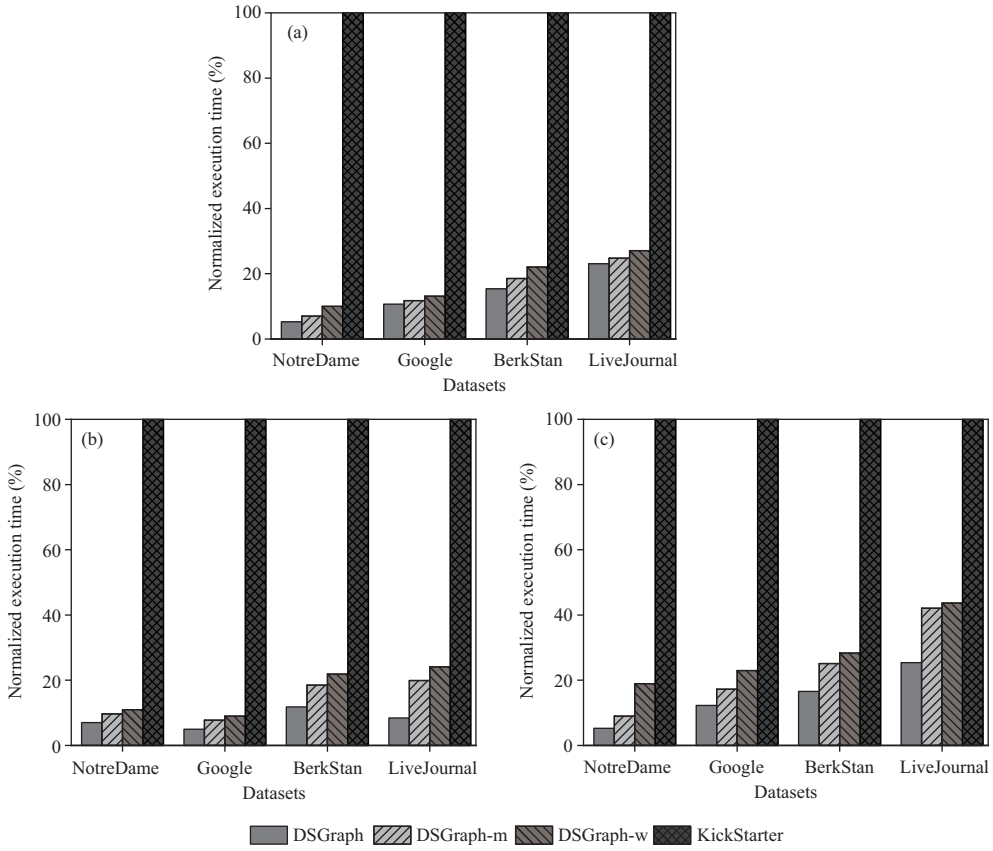

图 8　相对于 KickStarter 的标准化的执行时间

Figure 8　Normalized execution time over KickStarter. (a) WCC; (b) SSSP; (c) SSWP

点状态, 大大减少了图顶点状态的随机访问开销. DSGraph 相比于 KickStarter 的执行效率平均提升了 11.2 倍. DSGraph 的性能提升除了采用专用的硬件处理单元和片上存储单元外, 还能够精确识别图顶点之间的依赖关系, 并按照依赖关系进行图顶点的处理, 而 KickStarter 对于图顶点的处理并没有考虑其之间的依赖. 实验结果显示, DSGraph-m 相较于 KickStarter 的执行效率平均提升了 7.8 倍. 此外, 测试了 LiveJournal 在采用不同比例的边 (10%, 30%, 50% 和 70%) 来构造基础有向图快照后, DSGraph 在支持 WCC 算法时的性能情况. 实验结果显示, 基于 10%, 30%, 50% 和 70% 的边构造基础有向图快照后, DSGraph 相对于 KickStarter 分别获得 5.13, 6.34, 6.71 和 7.32 倍的性能提升.

4.3　与硬件动态有向图加速器性能比较

实验将动态有向图单调图算法加速器 DSGraph, DSGraph-m 与目前最好的加速器 JetStream [24] 进行性能分析比较. 将软件动态有向图系统 KickStarter [5] 作为基准, 分别对这 3 个加速器相对于 KickStarter 的加速比进行分析. JetStream 根据模拟器 [21, 24] 的配置参数进行配置. 实验分别在 3 个图算法和 4 个数据集下对性能进行测试分析. 如图 9 所示, DSGraph 在这 4 个数据集上的加速比都高于 JetStream, 这意味着 DSGraph 能够加快单调图算法在动态有向图上的收敛速度, 更有效地支持动态有向图处理. 具体来说, 相对于 JetStream, DSGraph 在数据集 NotreDame, Google, BerkStan 和 LiveJournal 上分别获得了 1.8∼2.9, 1.3∼3.2, 1.2∼1.6 和 1.7∼3.3 倍的性能提升. DSGraph-m 相较

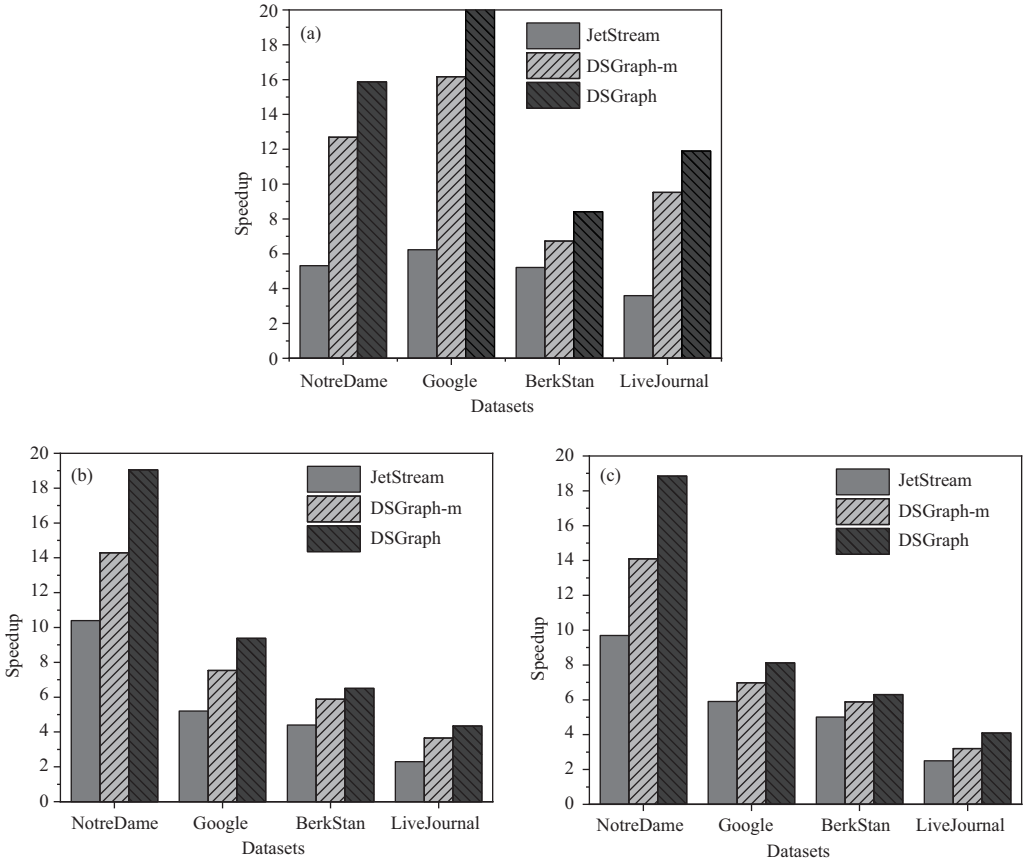

图 9 JetStream, DSGraph-m 和 DSGraph 相对于 KickStarter 的加速比

Figure 9 Speedups over KickStarter for JetStream, DSGraph-m, and DSGraph. (a) WCC; (b) SSSP; (c) SSWP

于 JetStream 在数据集 NotreDame, Google, BerkStan 和 LiveJournal 的执行效率平均提升了 1.5∼2.4, 1.1∼2.6, 1.1∼1.4 和 1.3∼2.6 倍. 主要的原因是 DSGraph 能够充分利用图顶点之间的依赖关系来有效加速单调图算法在动态有向图上的增量计算过程, 相比于 JetStream 的执行方式与图顶点之间的传播特征不匹配, DSGraph 能显著地提升单调图算法在动态有向图上执行时的收敛速度, 满足动态图处理的实时性需求.

4.4 硬件开销分析

本文使用加速器 JetStream [24] 中的方法对 DSGraph 的面积和功耗也进行了评估分析. 分析结果显示, 在 65 nm 工艺下, DSGraph 依赖提取组件中的每个遍历单元占用 2.23 mm^2 的面积, 增量处理组件中的每个处理单元占用 0.5 mm^2 的面积, 状态同步组件中每个同步单元加上对应的片上存储单元占用 2.84 mm^2 的面积. 可以看到依赖提取组件和状态同步组件拥有较大的面积开销, 这是由于这两个组件都存在较大的存储单元, 依赖提取组件的每个遍历单元要维护一个硬件栈结构, 状态同步组件的同步单元需要操作一个片上存储单元. DSGraph 的总面积为 44.6 mm^2, 在相同工艺下, DSGraph 的总面积只有 JetStream 总面积的 77.5%. 关于能耗, 本文比较了 DSGraph 和 JetStream 的能耗. 结果显示, 在动态图上运行单调图算法时, DSGraph 相比于 JetStream 能耗减少了 37.5%∼67.7%, 这是由于 DSGraph 通过运行时依赖驱动的增量执行方法, 大大减少了冗余计算开销和数据访问开销.

5 结论

针对现有软硬件方法在动态有向图上执行单调图算法时所面临的状态传递缓慢问题, 本文提出了一种运行时依赖驱动的增量执行方法, 为了减少运行时开销, 基于此方法设计实现了硬件加速器 DSGraph. DSGraph 能够在运行时有效感知动态有向图中图顶点状态之间的拓扑依赖关系, 生成局部依赖拓扑序列, 然后按照此局部拓扑序列对图顶点进行异步增量处理, 加快图顶点状态传递. 实验结果表明, DSGraph 的性能优于现有最好的动态有向图处理加速器 JetStream. 并且相对于现有最好的软件动态有向图处理系统 KickStarter, DSGraph 能够平均获得 11.2 倍的性能提升.

参考文献

1 Meyer U. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, 2001. 797–806

2 Hong S, Rodia N C, Olukotun K. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2013. 1–11

3 Jiang X, Xu C, Yin X, et al. Tripoline: generalized incremental graph processing via graph triangle inequality. In: Proceedings of the 16th European Conference on Computer Systems, 2021. 17–32

4 Zhang Y, Gao Q, Gao L, et al. PrIter: a distributed framework for prioritized iterative computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011. 1–14

5 Vora K, Gupta R, Xu G. KickStarter: fast and accurate computations on streaming graphs via trimmed approximations. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, 2017. 237–251

6 Liao X F, Chen Y C, Zhang Y, et al. An efficient incremental strongly connected components algorithm for evolving directed graphs. Sci Sin Inform, 2019, 49: 988–1004 [廖小飞, 陈意诚, 张宇, 等. 一种高效的面向动态有向图的增量强连通分量算法. 中国科学: 信息科学, 2019, 49: 988–1004]

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

8 Cheng R, Hong J, Kyrola A, et al. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, 2012. 85–98

9 Sheng F, Cao Q, Cai H, et al. GraPU: accelerate streaming graph analysis through preprocessing buffered updates. In: Proceedings of the ACM Symposium on Cloud Computing, 2018. 301–312

10 Mariappan M, Vora K. Graphbolt: dependency-driven synchronous processing of streaming graphs. In: Proceedings of the 14th EuroSys Conference, 2019. 1–16

11 Roy P, Khan A, Alonso G. Augmented sketch: faster and more accurate stream processing. In: Proceedings of the International Conference on Management of Data, 2016. 1449–1463

12 Shi X, Cui B, Shao Y, et al. Tornado: a system for real-time iterative analysis over evolving data. In: Proceedings of the International Conference on Management of Data, 2016. 417–430

13 Low Y, Gonzalez J, Kyrola A, et al. Distributed graphlab: a framework for machine learning in the cloud. In: Proceedings of the VLDB Endowment, 2012. 716–727

14 Ahn J, Hong S, Yoo S, et al. A scalable processing-in-memory accelerator for parallel graph processing. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015. 105–117

15 Ahn J, Yoo S, Mutlu O, et al. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015. 336–348

16 Zhang M, Zhuo Y, Wang C, et al. GraphP: reducing communication for PIM-based graph processing with efficient data partition. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture, 2018. 544–557

17 McCrabb A, Winsor E, Bertacco V. Dredge: dynamic repartitioning during dynamic graph execution. In: Proceedings of the 56th ACM/IEEE Design Automation Conference, 2019. 1–6

18 Dai G, Huang T, Chi Y, et al. Foregraph: exploring large-scale graph processing on multi-fpga architecture. In: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017. 217–226
19 Yao P, Zheng L, Liao X, et al. An efficient graph accelerator with parallel data conflict management. In: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018. 1–12
20 Ham T J, Wu L, Sundaram N, et al. Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016. 1–13
21 Yan M, Hu X, Li S, et al. Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019. 615–628
22 Rahman S, Abu-Ghazaleh N, Gupta R. Graphpulse: an event-driven hardware accelerator for asynchronous graph processing. In: Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture, 2020. 908–921
23 Zhang Y, Liao X, Jin H, et al. An adaptive switching scheme for iterative computing in the cloud. Front Comput Sci, 2014, 8: 872–884
24 Rahman S, Afarin M, Abu-Ghazaleh N, et al. JetStream: graph analytics on streaming data with event-driven hardware accelerator. In: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021. 1091–1105
25 Faldu P, Diamond J, Grot B. Domain-specialized cache management for graph analytics. In: Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture, 2020. 234–248
26 Zhang Y, Liao X, Jin H, et al. DiGraph: an efficient path-based iterative directed graph processing system on multiple GPUs. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 601–614

An efficient hardware accelerator for monotonic graph algorithms on dynamic directed graphs

Yun YANG[1,2,3,4], Hui YU[1,2,3,4], Jin ZHAO[1,2,3,4], Yu ZHANG[1,2,3,4*], Xiaofei LIAO[1,2,3,4], Xinyu JIANG[1,2,3,4], Hai JIN[1,2,3,4], Haikun LIU[1,2,3,4], Fubing MAO[1,2,3,4], Ji ZHANG[5] & Biao WANG[6]

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. *School of Mathematics, Physics and Computing, University of Southern Queensland, Toowoomba* 4350*, Australia*;
6. *Zhejiang Lab, Hangzhou* 311121*, China*
* Corresponding author. E-mail: zhyu@hust.edu.cn

Abstract With the rapidly growing demand for dynamic graph processing in the real world, existing research has proposed various methods to efficiently support the processing of monotonic graph algorithms on dynamic directed graphs. Despite of many research efforts, for iterative analysis of evolving directed graphs, existing solutions suffer from slow convergence speed and high data access cost, because many vertices are ineffectively reprocessed for lots of rounds so as to update their states according to other active vertices regardless of their dependencies. In this paper, we propose a novel and efficient accelerator DSGraph for incremental processing of dynamic directed graphs for monotonic graph algorithms. Compared with existing methods, the unique feature of DSGraph is that it can efficiently take advantage of the dependencies between the vertices to reduce the data access cost and improve the convergence speed of iterative processing of dynamic directed graphs. Specifically, DSGraph performs asynchronous iterative processing according to the topological dependency order of vertices on dynamic directed graphs at runtime, thus significantly reducing redundant vertex state updates. Meanwhile, DSGraph designs a multi-stage pipeline architecture. It performs asynchronous iterative processing of vertices according to the dependency order, which speeds up the vertex state propagation and reduces the data access overhead. Finally, DSGraph proposes a non-blocking data synchronization mechanism to reduce the system synchronization overhead by performing state updates of local vertices and data synchronization of external vertices in parallel. Experimental results show that DSGraph speeds up iterative dynamic graph processing by 11.2 times on average in comparison with the state-of-the-art software dynamic graph processing systems.

Keywords dynamic directed graph, monotonic graph algorithms, incremental processing, dependency-aware, graph accelerator