软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007558] [CSTR: 32375.14.jos.007558]

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

面向位图的 *K*-团枚举问题 GPU 优化算法*

曹炜辰[1,2], 孟 轲[1,2], 林志恒[1,2], 谭光明[1,2]

[1](处理器芯片全国重点实验室 (中国科学院 计算技术研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 谭光明, E-mail: tgm@ict.ac.cn

摘 要: *K*-团枚举是子图匹配中的一个重要问题, 位图算法被证明是求解 *K*-团枚举问题的有效方法. 目前最先进的 *K*-团枚举算法都采用 GPU 来加速. 先前工作没有关注真实世界图数据的稀疏性对基于位图的 *K*-团枚举算法的影响, 而是在 GPU 上采用静态的并行方法和位图构造策略, 这导致 GPU 计算效率低下. 提出了基于 thread 并行的位图任务负载均衡调度算法, 在解决线程分歧问题的同时实现位图算法的高并行性. 随后, 提出了一种动态位图构造算法, 使得位图可以在合适的时机被构造并高效启用位图算法. 实现了一个 GPU 友好的 *K*-团枚举问题求解系统 KCMiner, 它可以自适应地选择 *K*-团枚举任务的优化策略. 在 GPU 架构上的实验结果表明, 方法能够比 *K*-团枚举的基线算法最大实现 7.36 倍的加速, 与子图匹配系统的基线算法相比最大实现 30.2 倍的加速.

关键词: *K*-团枚举; 子图匹配; 位图算法; GPU; 性能优化

中图法分类号: TP301

中文引用格式: 曹炜辰, 孟轲, 林志恒, 谭光明. 面向位图的*K*-团枚举问题GPU优化算法. 软件学报. http://www.jos.org.cn/1000-9825/7558.htm

英文引用格式: Cao WC, Meng K, Lin ZH, Tan GM. GPU Optimization Algorithm for Bitmap-oriented K-clique Enumeration Question. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7558.htm

GPU Optimization Algorithm for Bitmap-oriented K-clique Enumeration Question

CAO Wei-Chen[1,2], MENG Ke[1,2], LIN Zhi-Heng[1,2], TAN Guang-Ming[1,2]

[1](State Key Lab of Processors (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: *K*-clique enumeration is an important problem in subgraph matching, and the bitmap algorithm has been proven to be an effective method for solving the *K*-clique enumeration problem. Currently, state-of-the-art *K*-clique enumeration algorithms are accelerated by GPU. Previous studies have not investigated the impact of sparsity in real-world graph data on bitmap-based *K*-clique enumeration algorithms. Instead, static parallelization methods and bitmap construction strategies are commonly used on GPU, which result in low computational efficiency. This study proposes a thread-parallel load-balancing scheduling algorithm for bitmap tasks, which resolves the thread divergence problem while achieving high parallelism in the bitmap algorithm. Furthermore, it introduces a dynamic bitmap construction algorithm, enabling bitmaps to be constructed and activated at appropriate times for efficient execution of the bitmap algorithm. A GPU-friendly *K*-clique enumeration system, KCMiner, is implemented, which adaptively selects optimization strategies for *K*-clique enumeration tasks. Experimental results on GPU platforms show that the proposed method achieves up to 7.36 times speedup over the baseline *K*-clique enumeration algorithm and up to 30.2 times speedup over the baseline subgraph matching system.

Key words: *K*-clique enumeration; subgraph matching; bitmap algorithm; GPU; performance optimization

图是表达对象与对象之间关系的基本数据结构, 也是对现实世界中对象及其关系的一种抽象. 子图匹配算法作为一种知识提取的工具, 能够从图数据中寻找高价值的数据模式, 从而揭示现实世界对象中的隐藏关系, 在人工

* 基金项目: 国家自然科学基金 (T2125013, 62032023); 国家重点研发计划 (2023YFB3001902)
收稿时间: 2025-01-22; 修改时间: 2025-04-15; 采用时间: 2025-09-23; jos 在线出版时间: 2026-01-14

智能时代有着重要的作用, 因此被广泛应用于推荐系统[1,2]、异常检测[3,4]、生物信息学[5−7]以及社交网络分析[8−10]等. 但是匹配图模式不仅需要搜索巨大的空间, 还需要在搜索时物化大量的中间结果, 这给高效的大规模子图匹配带来了巨大的挑战.

K-团也被称为完全子图, 具有 K 个顶点和 $K\times(K-1)$ 条边, *K*-团中的每个顶点都和其他顶点有边相连. *K*-团枚举问题是子图匹配中的一个重要问题. 在图压缩[11,12]、图划分[13]、推荐系统[14,15]和社区检测[16−19]等各个领域都有广泛应用. *K*-团枚举问题可以使用子图匹配方法来解决, 最基本的算法是从数据图中的任意顶点开始, 使用与邻居集的交集运算依次匹配团中的每个顶点[20]. 由于团的搜索空间局限于 1 跳邻居范围内, 先前的工作将此范围划分为子图进行搜索, 通过位图来存储子图内顶点之间的邻接关系, 并使用按位与运算替代二分搜索来加速集合求交[21].

随着 GPU 在子图匹配问题上展示出优异性能[22], 将枚举任务和集合求交运算并行化从而带来性能提升, 现有的研究工作将这种并行框架中的二分搜索替换为按位与运算, 提出了 *K*-团枚举的 GPU 并行算法. 将每一个顶点作为起始顶点, 开启一个枚举任务并分配给一个 warp, 并使用起始顶点的 1 跳邻居构建子图, 子图中的枚举任务用 warp 并行[23]或 thread 并行完成[21]. 然而, 位图算法中做集合求交的计算成本很低, warp 并行把两行位图的按位与计算分配给一个 warp, 会导致并行度低的问题. thread 并行则把两行位图的按位与计算分配给一个线程单独执行, 从而提升并行度. 但 warp 内的不同线程只能同时执行相同指令, 为线程分配任务的条件分支将导致线程分歧的问题, 造成部分线程空闲从而降低性能. 此外, 现实世界中的图数据局部稀疏性差异很大, 固定使用起始顶点的一跳邻居作为子图构建位图空间并非普遍有效. 当位图空间中仅包含极少量 1 位的情况下, 大量的时间将浪费在无效的遍历上. 因此需要进一步关注位图算法带来的问题特征优化 GPU 并行算法.

本文在无向图, 无标签图的图数据场景下, 提出了一种使用位图算法的 *K*-团枚举问题 GPU 优化算法. 通过分析现实世界图数据的特点, 在 GPU 上实现了高并行性.

本文的主要贡献包含以下部分.

(1) 提出了基于 thread 并行的位图任务负载平衡调度. 在解决线程分歧问题的同时, 实现位图算法的高并行性.

(2) 提出了一种稀疏性感知的位图动态构造算法. 在集合交集操作期间, 该算法动态选择构建位图空间的最佳时机, 防止位图算法的低效应用.

(3) 实现了 KCMiner, 一个用于 *K*-团枚举算法求解的系统. KCMiner 识别枚举任务类型并启发式地选择适合任务工作负载的优化策略.

本文使用现实世界数据图和基线算法对 KCMiner 进行了广泛的评估. 实验表明, KCMiner 比最先进的 *K*-团枚举的 GPU 并行工作 KCGPU[21]实现了 7.36 倍的加速. 我们还将 KCMiner 与过去两年发布的包括 *K*-团枚举实现的 5 种最先进的基于 GPU 的子图匹配系统进行了比较, 最大加速为 30.2 倍.

本文首先简要介绍 *K*-团枚举算法的相关方法和研究现状 (第 1 节), 并给出必要的基本概念与问题定义 (第 2 节). 随后介绍如何用集合求交求解 *K*-团枚举算法 (第 3 节), 以及如何用位图算法完成 *K*-团枚举过程 (第 4 节). 以此为基础, 本文介绍一种 *K*-团枚举的优化算法 (第 5 节), 并分析算法的复杂度 (第 6 节). 最后通过实验验证所提算法的有效性 (第 7 节).

1 相关工作

1.1 *K*-团枚举算法

K-团枚举指的是在大规模图数据中找出所有指定大小 *K*-团的子图. 三角形是 *K*=3 时的特殊 *K*-团, 也是 *K*-团枚举中最简单的特例, 在图数据中枚举三角形的问题由来已久. 文献 [24,25] 都是基于排序的精确三角形计数算法. Tom 等人[26]和 Hu 等人[27]对三角形计数问题开发了 GPU 并行算法, 可以用于共享内存的众核环境. Chiba 等人[20]开发了第 1 个 *K*-团枚举的精确计数算法, 它提出了基于回溯的算法并在 DFS 过程中枚举团, 该算法在现实生活中对于 K 值较小的稀疏图非常有效. Finocchi 等人[28]用度数排序改进了这种算法. 随后, Danisch 等人[29]通过使用 *K*-核分解进一步改进了该算法. Li 等人[30]进一步开发了一种基于图染色技术的改进算法. KCGPU[21]是第 1

个在 GPU 上实现 K-团枚举算法的工作, 在使用集合求交完成 K-团枚举任务的基础上, 根据团这一类模式图的特征提出位图算法, 优化集合求交的计算效率. KCGPU 按照 DFS 枚举树的分支划分任务, 固定在第 1 层或第 2 层构建位图, 并使用 thread 并行分配位图的计算任务. 然而在使用位图算法时, KCGPU 的并行方法与使用二分搜索完成集合求交的并行算法基本相同, 目前也尚无工作进一步研究位图算法的高效并行方法.

1.2 *K*-团枚举的 GPU 优化

K-团枚举问题是子图匹配的子问题, 相比于 K-团枚举问题, 先前的研究中已有更多关于子图匹配的 GPU 优化工作, 使用二分搜索或哈希表等方法完成集合求交. 自 AutoMine[31]问世以来, 集合求交已成为求解子图匹配问题的主流方法. GraphPi[32]进一步引入了基于调度+限制的集合交集模式匹配范式. AutoMine 和 GraphPi 均为 CPU 环境中实现的子图匹配系统, 主要贡献在于定义了以集合操作为中心的子图匹配问题求解范式. Pangolin[22]是第 1 个使用 GPU 加速子图匹配系统的应用, 这一工作提出了“扩展-消除-过滤”的抽象流程. G2Miner[33]进一步实现了用 GPU 加速子图匹配问题的优化, G2Miner 根据顶点或边划分模式匹配任务, 并以 warp 为并行单元分配这些任务. G2Miner 还可以自动选择最佳内核并自动生成高性能应用程序的代码. 然而, G2Miner 仅对 K-团这类模式图启用有向化的特定优化, 以此缩小搜索空间, 但其使用二分搜索完成集合求交, 相比位图算法性能提升尚且不足. 此外, 子图匹配系统的工作侧重于对不同的模式图提出通用的优化方法. DecoMine[34]提出了模式图分解算法, 但这一方法不能用于 K-团枚举问题的求解. 上述子图匹配系统的 GPU 优化工作更关注对各种模式图都适用的通用优化, 虽然可以支持 K-团问题的求解, 但通常难以带来明显的加速效果.

GraphFold[23]和 KCGPU 是在 K-团枚举问题上使用位图算法进行优化的工作. GraphFold 使用了 warp 并行的位图算法来优化 K-团枚举问题. GraphFold 还优化了子图匹配过程中会出现的细粒度冗余问题, 但 K-团枚举中不存在该问题. KCGPU 使用 thread 并行的位图算法优化 K-团枚举问题. KCGPU 和 GraphFold 只是使用先前子图匹配系统提出的 GPU 并行方法对位图算法的 K-团枚举进行移植, 然而 GraphFold 的 warp 并行对于位图算法并行度低, KCGPU 的 thread 并行会出现线程分歧问题. 虽然使用位图算法相比于二分搜索的集合求交能带来一定加速, 但使用位图算法的 K-团枚举问题在 GPU 架构上仍有较大性能潜力可以发掘.

2 基础知识

给定一个有限的点集 V 和连接它们的边集 $E \subseteq V \times V$, 一个图可以表示为 $G=(V,E)$. $N(v)$ 表示点 v 的邻居集合, 邻居集合的大小即为点 v 的度数, 记作 $deg(v)$. $v \sim u$ 表示点 v 和点 u 相连, $v \nsim u$ 则表示不相连.

定义 1. 子图匹配. 给定一个具有 m 个顶点 $(m>2)$ 的模式图 $P=(V_P,E_P)$, 对于任意顶点 $V \in V_P$, 子图匹配算法需要发现所有映射 $v \to u, v \in V_p, u \in V$, 使得顶点诱导子图 $\{u_0,u_1,\ldots,u_{m-1}\}$ 和给定模式图 P 同构. 先前的研究提出子图匹配算法可以用以集合为中心的方法来求解, 只需要使用集合求交 $(\cap)$ 等集合运算即可实现[22,23,31–33].

定义 2. K-团. 团也被称为完全子图, 是子图匹配算法中的一种特殊模式图. K-团是具有 K 个顶点的模式图 $P_K=(V_K,E_K)$, 其中每一个点的度数都是 $K-1$, 即 $deg(v)=K-1, v \in V_K$. 图 1 展示的是 $K=3$至$K=7$ 的团.

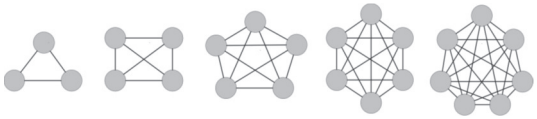


图 1 K-团的模式图

3 基于集合求交的 *K*-团枚举算法

算法 1 是用集合求交枚举 K-团的算法流程. 该算法首先遍历数据图中的所有点, 每一个点 v_0 都独立作为起始点开启一轮枚举任务. v_0 的邻居集合表示为 $N(v_0)$, $N(v_0)$ 成为下一个待枚举点 v_1 的候选集. 遍历 $N(v_0)$ 选择一个点 v_1, 并将 $N(v_0)$ 和 $N(v_1)$ 求交集, 即可得到点 v_0 和 v_1 的公共邻居集合. 此时得到的枚举结果为 3-团[35,36]. 重复迭代这

一过程, 迭代至第 K 轮即可完成 K-团的枚举. 该过程以递归方式生成一棵搜索树, 算法的执行过程是以 DFS 顺序遍历搜索树[31].

算法 1. 基于集合求交的 K-团枚举算法.

输入: 数据图 $G(V,E),K$;
输出: 所有 K-团的实例 e.

```
1. for v_0 ∈ V do // Level 1
2.    Intersect(N(v_0),1)
3. end for
4. Function Intersect(vlist_i, k)
5.    if k == K then
6.       Get an embedding e
7.       return
8.    else
9.       for v_i ∈ vlist_i do
10.         vlist_{i+1} = vlist_i ∩ N(v_i) //使用二分搜索完成集合求交
11.         Intersect(vlist_{i+1}, k+1)
12.      end for
13.   end if
14. end Function
```

由于现实世界数据图具有稀疏性, 先前的工作通常用 CSR 格式存储图. 将图的邻接矩阵中所有非零元压缩成一个一维数组, 并使用偏移量来存储每个顶点邻居列表的起始索引. 在 CSR 图存储格式下, 集合求交需要用二分搜索的方式完成.

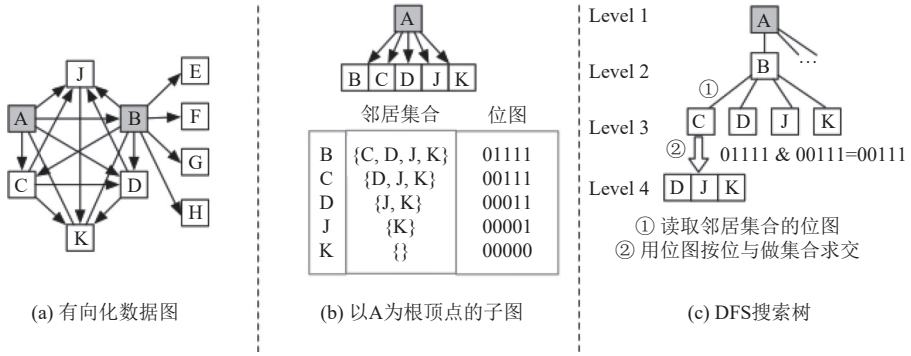
以边为中心的任务划分和图的有向化是两种常用的优化策略. 先前的研究表明, 相比于从数据图的点集 V 选取一个顶点 v_0 开启一个枚举任务, 从数据图的边集 E 中选取一条边 e, 并从 e 的两个节点 v_0 和 v_1 开启一个枚举任务可以使得每个枚举任务的负载更均衡[33]. 图的有向化是一种预处理技术, 可以将无向图转换为有向图, 通过设置的有向化规则 (如度数优先规则) 使得数据图中任意两个顶点之间只会有单向连接. 图的有向化可以有效剪枝搜索空间, 避免重复的枚举[28,30,37–39].

4 使用位图的 K-团枚举基础算法

4.1 位图的构建与使用

位图算法是一种可以专用于团这一类模式图的优化算法[21,40]. 团的主要特征是所有点全相连, 即所有点之间都互为一跳邻居. 遍历数据图中的点集 V 得到 v_0 作为枚举任务的起始点后, v_0 同时成为构造子图所使用的根顶点, $N(v_0)$ 即为子图. 如图 2(a) 所示, 数据图按照顶点顺序的规则将所有边有向化, 从而使得点和点之间单向连接, 避免自同构带来的重复枚举. 在数据图中选取一个根顶点 A 后, 可以用 A 的一跳邻居集合{B, C, D, J, K}构建子图. 如果目标是完成 K-团的枚举任务, 由于已经确定根顶点 A, 因此只需在子图范围内完成枚举 (K–1)-团的任务. 为了加速集合求交的效率, 可以在子图范围内求出每个点和子图中其他点的邻接关系, 并记录为位图. 如图 2(b) 所示, 按照子图内的遍历顺序{B, C, D, J, K}, 子图内每个点 v_1 的邻接列表将映射到一个 $deg(v_0)\times deg(v_0)$ 的位图空间内. B 点的子图内邻接列表为{C, D, J, K}, 映射后 B 点的位图记录为 01111. 此后所有子图内的集合求交都可以取每个点的位图用按位与操作完成.

定义 3. Level 层. 如图 2(c) 所示, 枚举过程会形成一棵 DFS 搜索树, 每一层对应于算法 2 中的一层循环. 每一层需要遍历该层的候选集, 并用集合求交得到下一层的候选集.



(a) 有向化数据图 (b) 以A为根顶点的子图 (c) DFS搜索树

图 2 使用位图算法的 *K*-团枚举算法

算法 2. 使用位图优化的 *K*-团枚举算法.

输入: 数据图 $G(V,E),K$;
输出: 所有实例 e.

```
1.  for v0 ∈ V do // Level 1
2.    for v1 ∈ N(v0) do // Level 2 构建位图
3.      vlist(v1) ← N(v0) ∩ N(v1)
4.      bitmap(v1) ← vlist
5.    end for
6.    for v1 ∈ N(v0) do
7.      BitmapIntersect(bitmap(v1), deg(v0), 3) // Level 3
8.    end for
9.  end for
10. Function BitmapIntersect(cur_bitmap, deg, k)
11.   if k == K then
12.     Get an embedding e
13.     return
14.   else
15.     for i ∈ (0, deg) do
16.       if cur_bitmap[i] == 1 then
17.         next_bitmap = cur_bitmap & bitmap(i) // 用按位与做集合求交, 得到 Level 4
18.         BitmapIntersect(next_bitmap, deg, k + 1) // Level K+1
19.       end if
20.     end for
21.   end if
22. end Function
```

位图算法的执行流程如算法 2 所示. 第 1 行为第 1 层循环, 遍历数据图 G 的点集 V, 确定一个点 v_0 作为根顶点; 第 2–5 行为位图构建过程, 遍历子图 $N(v_0)$ 中的每一个点 v_1; 第 3 行用 v_1 的邻居集合 $N(v_1)$ 和子图点集 $N(v_0)$

求交得到 v_1 在子图内的邻接列表 $vlist(v_1)$, 使用 $vlist(v_1)$ 即可在第 4 行映射出 v_1 点的位图结果并存储.

完成位图构建后, 从第 6 行开始使用位图做集合求交. 该递归算法展开后即为多层循环嵌套的形式, 每一层循环枚举 K-团中的一个点, 由于 v_0 点已经确定, v_1 点在第 6 行遍历得到, 因此第 7 行初次调用 *BitmapIntersect* 函数将从求解第 3 个点开始. 首先将 v_1 点的位图输入作为初始 *cur_bitmap*, 将 v_0 点的度数输入 (即子图中点的数量) 作为位图的位宽, 后续算法中所有位图的位宽都为 $deg(v_0)$. 在每一层循环中, 遍历 *cur_bitmap* 的每一位, 如果第 i 位为 1, 则读取位图空间中的第 i 个位图, 与 *cur_bitmap* 做按位与得到一个 *next_bitmap*. 继续将 *next_bitmap* 作为下一层循环的 *cur_bitmap*, 以此嵌套下一层的循环. 重复这一嵌套循环的过程直到指定第 K 层, 即可得到图 2(c) 中展示的 K-团枚举结果.

4.2 位图算法的 GPU 基础实现

GPU 有两种不同的并行计算架构: SIMT (单指令、多线程) 和 MIMD (多指令、多数据流). 在 GPU 架构中, 线程被聚合成 warp, warp 中的每个线程同步执行相同的操作. 当线程处理条件分支时, 不同的路径需要单个 warp 中的线程串行执行不同的分支, 从而导致线程分歧. 这种现象会降低计算效率, 因为在同一时刻, warp 中只有一部分线程可能处于活跃状态. 在 GPU 上, warp 是资源调度的基本单位, 允许在不同的 warp 上同时执行不同的指令, 每条指令作用于不同的数据. 这些 warp 被组织成更大的单元, 称为 block, 它们被集体调度到 GPU 的流式多处理器进行处理.

下面介绍基于 warp 并行和 thread 并行的两种位图算法并行方式. 如图 3(a) 和 (b) 所示, 根据 GPU 的 MIMD 特性, 图数据中的全部枚举任务将以 warp 为执行单元进行任务划分. 所有 warp 并行分配第 1 层顶点, 随后一个 warp 完成一个位图空间的构建任务. 完成位图构建后, 使用位图的 K-团枚举后续计算过程可分为两种并行方式: warp 并行和 thread 并行.

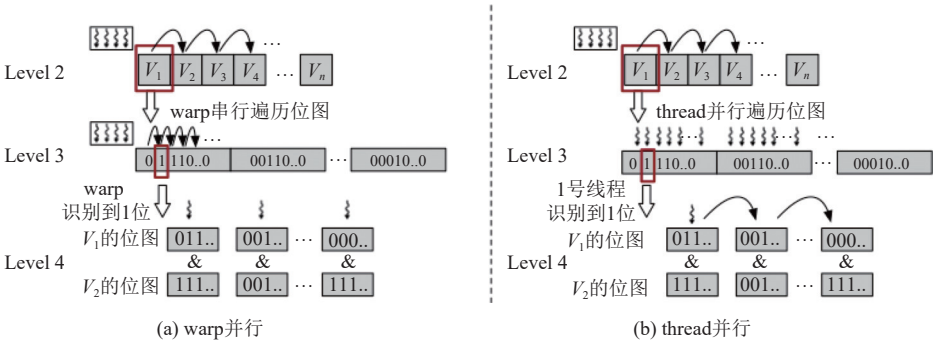


图 3 warp 并行和 thread 并行

warp 并行的并行效率较低. 如图 3(a) 所示, 一个 warp 内的所有线程读取同一个点的位图, 以串行方式一起遍历一个位图以判别其中 1 位. 当识别到 1 位要执行集合交集计算时, 每个线程处理位图中的一个 32 位片段, 完成部分按位与计算. 然而, 由于一个 warp 中包含 32 个线程, 因此位图长度必须达到 32×32 = 1024 才能充分利用 warp 的并行能力. 由于现实世界数据图具有幂律分布的特征, 一个点导出的子图很少包含 1 024 个点. 另一方面, 在使用位图求解 K-团枚举问题时, 主要开销在于位图遍历而非按位与计算, 而 warp 并行时位图的遍历需要串行执行. 因此, warp 并行无法在使用位图的 K-团枚举问题上充分发挥 GPU 的并行性.

thread 并行虽然能同时处理多个位图按位与计算, 但会出现线程分歧问题. 如图 3(b) 所示, thread 并行是每个 warp 读取一个位图, 每个线程独立检查其中一位是否为 1 位以启动位图按位与任务, 进入下一层循环. 这种方法允许多个线程可以并行执行多个位图按位与运算, 但这是理想情况. 位图中的 1 位具有稀疏性且分布不均匀, 导致每个位图的工作负载可能相差很大. 因此, 如果线程在算法 2 第 16 行判定为假, 它将进入空闲状态, 且由于 GPU 的 SIMT 特性, 在其他线程未完成计算时它不会做任何操作, 造成资源浪费. 因此, thread 并行的做法会带来线程分歧的问题.

5 使用位图的 *K*-团枚举改进算法 KCMiner

5.1 基于 thread 并行的任务调度算法

为了在提高并行度的同时优化线程分歧的问题, 针对使用位图的 *K*-团枚举算法, 本文提出一种基于 thread 并行的负载均衡任务调度算法.

5.1.1 有效位图搜索任务预分配

当一个位图空间构建完成后, 一个 warp 中的 32 个 thread 并行遍历一个位图, 将算法 2 第 15 行的循环划分给每个线程以并行求解. 如图 4(a) 所示, 在 Level 2 中, 线程如果从头开始并行遍历位图, 则线程 T_0、T_1、T_2 会读到 0 位, 不会进入下一层的枚举任务, 过早进入空闲状态. 这一行位图中虽然只有 2 个 1 位, 但却需要 4 个线程迭代 2 轮才能执行完 Level 2 的任务. 我们希望每个线程都可以在第 1 层读取到 1 位, 领取到位图计算任务, 因此需要先过滤位图中的 1 位, 并将这些有效枚举任务预分配给线程.

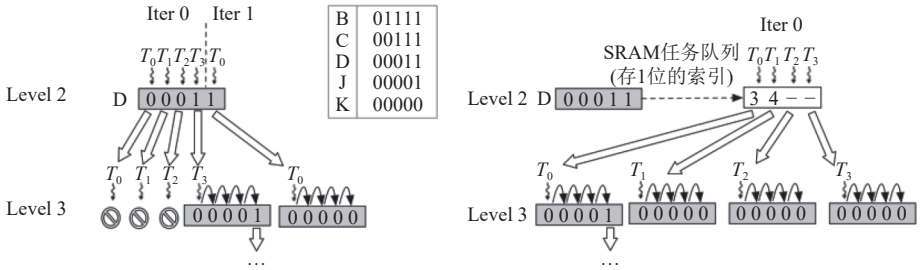


图 4 任务预分配后线程之间负载仍然不均的情况

我们在 GPU SRAM 中设置一个任务队列为每个线程分配有效任务. 首先, 以 warp 为单位并行遍历位图空间中的一个位图, 并将其中 1 位的索引入队. 将一个位图中的全部有效枚举任务入队后, 每个线程单独出队一个有效任务并启动对应的位图按位与计算, 进入下一层枚举任务. 我们希望所有线程都可以参与第 5.1.2 节中将要介绍的线程间任务调度, 但由于 GPU 的 SIMT 特性, 如果 Level 2 的位图中, 1 位的数量不为 warp 中线程数量 32 的倍数, 未领取到枚举任务的线程则将直接退出, 不会执行下一层枚举任务的指令. 因此我们在完成入队后, 需要用空任务填充任务队列到 32 的倍数, 使得所有线程都可以执行相同的指令, 进入下一层循环.

5.1.2 双缓冲区线程间任务调度策略

即使通过任务预分配确保每个线程在 Level 2 都领取到一个有效任务, 每个线程从 Level 3 开始需要处理的枚举任务负载差距仍然很大. 如图 5 所示, 当每个线程领取到一个有效任务进入 Level 3 时, 每个线程在 Level 3 需要处理的枚举任务数量仍会有差异, 其中 1 位的数量等同于子枚举树的数量. 为了在深入循环层级后线程之间依旧能保持负载均衡, 我们提出了双缓冲区线程间任务调度算法.

双缓冲区是一个大小为 64 的循环队列, 分为两个大小为 32 (一个 warp 的线程数) 的缓冲区. 该调度算法包括两个主要部分: 有效任务的装填和下一级任务的发射. 在上一级完成一次位图按位与计算后, 每个线程都会同时得到一个位图结果, 作为当前层级输入双缓冲区任务调度算法的 *cur_bitmap*. 在装填阶段中, 筛选框从所有线程的 *cur_bitmap* 的第 1 位开始筛查 1 位, 所有线程同步筛选. 识别到 1 位的 thread 并行将有效任务插入双缓冲队列. 该操作可通过 CUDA 提供的 *__ballot_sync*() 和 *__popc*() 原语并行实现. 任务插入缓冲区的格式为 (*cur_bitmap_idx*,*i*), 这足以指示任务需要加载哪个 *cur_bitmap* 和位图空间中的第几个位图, 用以完成算法 2 的第 17 行按位与计算. 每完成一轮任务装载后, 一个 warp 的所有线程同步检查双缓冲区是否已填充 32 个有效任务, 确保每个线程都能从缓冲区领取到一个有效任务. 如果未满, 则筛选框移位以检查 *cur_bitmap* 的下一个位. 双缓冲区中一旦积累了 32 个有效任务, 就会进入一次发射阶段, 每个线程都会出队一个有效任务以执行其指示的枚举任务. 完成后续所有嵌套循环后, 线程都返回到执行任务调度的层级, 继续进行任务装填. 重复这一操作, 直到筛选框筛查完所有 *cur_bitmap*. 这种调度策略可以应用于嵌套循环的每一层.

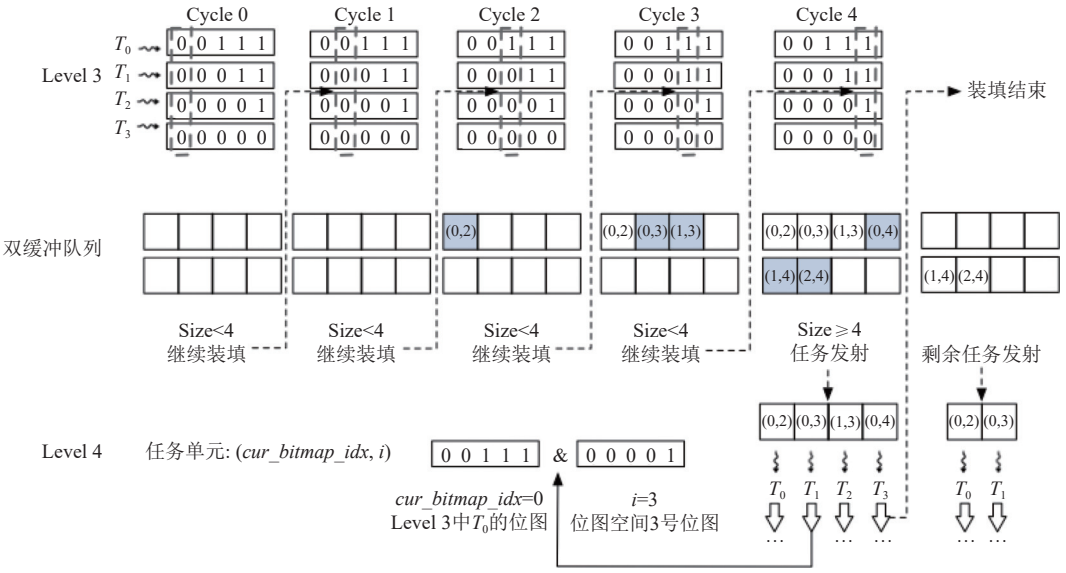


图 5 双缓冲线程任务调度队列

算法 3. 双缓冲队列线程任务调度算法.

输入: *warp_size*, 线程号 *thread_id*, *cur_bitmap*;
输出: 所有实例 *e*.

1. **Function** *BitmapIntersect*(*cur_bitmap*, *deg k*)
2. **if** $k == K$ **then**
3. Get an embedding *e*
4. **return**
5. **else**
6. **for** $i \in (0, deg)$ **do**
7. **if** $cur\_bitmap[i] == 1$ **then**
8. *load in buffer*(*thread_id*, *i*)
9. **end if**
10. **if** $buffer\_size \geqslant warp\_size$ **then**
11. $launch(cur\_bitmap, bitmap(i))$
12. $next\_bitmap = cur\_bitmap \ \& \ bitmap(i)$
13. $BitmapIntersect(next\_bitmap, deg, k+1)$
14. **end if**
15. **end for**
16. **end if**
17. **end Function**

图 5 给出了双缓冲区任务调度算法的示例. 假设 4 个线程共同管理 4 个 *cur_bitmap*, 双缓冲区大小设置为 8. 筛选框从 Cycle 0 开始筛查 *cur_bitmap*, 执行到 Cycle 2 时, 筛选框在线程 T_0 的 *cur_bitmap* 中找到一个有效任务, 任务 (0,2) 被插入双缓冲区队列中. 但此时双缓冲区队列尚未填满一个缓冲区, 即 4 个任务, 筛选框继续筛选更多任务. 当 Cycle 4 完成时, 双缓冲区中总共装填了 6 个任务, 已经装满一个缓冲区的大小, 此时双缓冲区启动一次任

务发射. 第 1 个缓冲区中的 4 个任务全部发射, 4 个线程各独立出队一个任务, 并按照任务单元指示完成位图按位与计算进入 Level 4, 并继续执行后续的团枚举操作. 每个线程执行一个子枚举树下的枚举任务, 全部完成后, 4 个线程同时返回到双缓冲区调度层 Level 3 继续装填任务. 由于该示例在 Cycle 4 时已经到达 *cur_bitmap* 的末尾, 因此将直接发射第 2 个缓冲区中剩余的任务, 否则将继续筛选有效任务, 直到第 2 个缓冲区中填满 4 个任务, 再发射 4 个任务. 两个缓冲区交替发射任务, 因此称为双缓冲区任务调度算法.

设置为两个缓冲区交替发射任务的原因是一个缓冲区的大小等同于一个 warp 中的线程数, 任何时刻填充任务都不会使待发射的任务数超过两倍缓冲区大小. 亦可理解为两个缓冲区形成一个大小固定的循环队列.

5.2 稀疏性感知的位图动态构造策略

现实世界图数据中局部稀疏性差异很大, 位图算法并非在任何情况下性能都优于用二分搜索完成集合求交. 当位图空间中只有极少量的 1 位时, 位图算法的大部分执行时间要花费在遍历并访存无效的 0 位上, 使得位图算法性能低效. 相应地, 二分搜索此时存在性能优势区间. 二分搜索的复杂度仅和位图空间中的 1 位数量有关, 因此在较为稀疏的场景下使用二分搜索会出现性能优于位图算法的情况.

先前的工作静态使用一个顶点的邻居集合或一条边上两个顶点的公共邻居集合作为子图, 并构建子图的位图空间. 然而在不同层构建位图空间时, 位图算法的性能受稀疏度影响会存在较大差异, 第 7.2.4 节也证实了这一见解.

为了仅在位图算法能够高效执行时才构造位图空间, 我们提出了一种稀疏性感知的位图动态构建算法. 首先, 我们用一个位图空间中 1 位的占比来定义稀疏度 γ.

$$\gamma = \left(\sum_{i=0}^{deg} \#bit1\right) \Big/ deg^2.$$

我们不会固定地使用根顶点的邻居集合或一条边上两点的公共邻居集合来构建位图. 相反, 我们在嵌套循环的枚举过程中在每一层评估位图的稀疏度 γ, 动态地使用位图算法.

在嵌套循环进行团枚举的过程中, 使用更深层次的公共邻居集合也可以构造位图, 从而得到位图长度更短, 更加稠密的位图空间. 如图 6 所示, 在示例中, 如果在第 1 层构造位图, N(B) 的大小等同于位图的长度. 然而在由 N(B) 构成的子图中, 只有顶点{D, J, K}之间相互连接, 这导致很长的位图中只包含少量的 1 位. 因此, 我们可以在第 2 层用{D, J, K}作为子图构造更小的位图. 如图 6 所示, 在第 2 层构造出的局部小位图空间中, 包含了将近一半的 1 位. 这种方法避免了位图算法冗余的无效遍历, 可以显著降低无效遍历和过长按位与计算的开销.

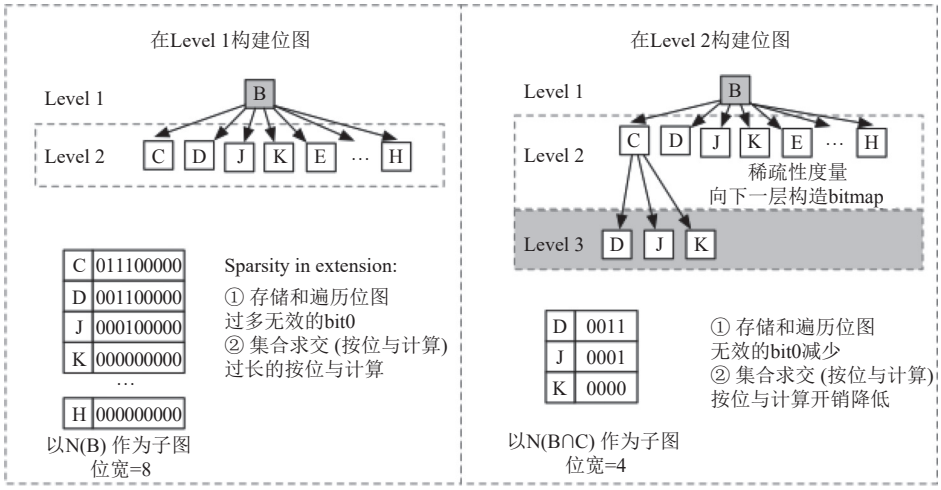


图 6　不同层构造 bitmap 的稀疏性差异

我们提出一种在运行时基于稀疏性 γ 动态构造位图的方法. 首先, 我们以 warp 为单位串行遍历枚举任务的搜索树. 在完成集合求交进入枚举任务的下一层时, 我们首先评估构造位图的 γ. 对于任何潜在子图, 利用二分搜索

来计算子图中的边数. 它对应于子图位图中 1 位的数量. 如果不满足启用位图算法的阈值, warp 将使用二分搜索完成集合交集并进入下一级. 如果满足设定的 γ 阈值, 则启用位图算法, 从当前层开始用按位与计算完成集合求交.

算法 4. 位图动态构造算法.

输入: 数据图 $G(V,E),K$;
输出: 所有实例 e.

1. **for** $v_0 \in V$ **do** // Level 1
2. $build\_or\_intersect(N(v_0),deg(v_0),1)$
3. **end for**
4. **Function** $build\_or\_intersect(vlist_i,deg,\ k)$
5. **for** $v_i \in vlist_i$ **do**
6. $vlist(v_i) \leftarrow vlist_i \cap N(v_i)$
7. $bitmap(v_i) \leftarrow vlist(v_i)$
8. $\#bit1 += popc(bitmap(v_i))$
9. **end for**
10. $\gamma = \#bit1/deg^2$
11. **if** $\gamma > \lambda(k)$ // 本层构建的位图稀疏度大于阈值, 启用位图算法
12. **for** $v_i \in vlist_i$ **do**
13. $BitmapIntersect(bitmap(v_i),deg,k+1)$
14. **end for**
15. **else** // 本层构建的位图稀疏度小于阈值, 使用 *vlist* 结果进入下一层
16. **for** $v_i \in vlist_i$ **do**
17. $vlist_{i+1} \leftarrow bitmap(v_i)$
18. $build\_or\_intersect\ (vlist_{i+1},len(vlist_{i+1}),k+1)$
19. **end for**
20. **end if**
21. **end Function**

在实验评估部分, 我们测试了不同位图稀疏性 γ 对性能的影响, 并探测启用位图算法的 $\lambda(k)$ 阈值. 这指导我们选择更合适的稀疏度 γ 来构建并使用位图算法.

5.3 GPU 友好的算法实现 KCMiner

结合算法 4, 我们实现了一种 GPU 友好的 *K*-团枚举求解器 KCMiner. 基于 thread 并行的负载均衡任务调度策略和位图动态构建算法都被 KCMiner 启用. KCMiner 以图 *G* 中的一条边 *e* 作为任务单元, 遍历边集合 *E* 中的每一条边 *e* 作为一个枚举任务. KCMiner 首先对枚举任务做预处理分类, 计算一条边 *e* 上两点公共邻居集合大小, 以此作为每个枚举任务的尺寸. 对小于 32 的任务和大于 1 024 的任务分别使用额外优化.

如果枚举任务尺寸小于 32, 则构建出的位图空间位宽也小于 32, KCMiner 将启用共享内存优化. 由于其占用的存储空间很小, 位图空间构建后将被存储在 GPU 共享内存中, 每个线程在枚举过程中产生的位图中间结果也将存储在寄存器中, 这种方法可以显著提高位图访问效率.

KCMiner 将根据枚举任务尺寸选择并行处理单元. 枚举任务尺寸大于 1 024 时, 将改用一个 block 处理一个枚举任务, 位图构建以 block 并行方式完成, 从第 2 层开始每个嵌套循环的分支被分配给一个 warp 来执行, warp 内再按照 thread 并行划分任务. 枚举任务尺寸小于 1 024 时以 warp 为单位处理一个枚举任务.

6 算法分析

6.1 双缓冲区任务调度策略

对于基础的 thread 并行算法, 由于 GPU 的 SIMT 并行特性, 一个 warp 中所有线程执行相同的指令. 因此一个 warp 中的所有线程同步遍历 *cur_bitmap* 中的每一位, 只要还有线程在执行枚举任务, 所有线程都需要执行相同的指令空跑. 此时算法时间复杂度将由任何时刻下任务量最多的线程决定, 假设每一时刻只有 1 个线程在 *cur_bitmap* 中读取到 1 位, 最坏时间复杂度为 $O(deg\times warp\_size\times K)$.

对于改进算法, 所有线程将统一分摊枚举任务. 需要用 warp 原语实现向双缓冲队列入队的操作, 这些操作都可以由 thread 并行执行, 与 SIMT 并行特性保持一致, 这一步时间复杂度为 $O(deg)$. 对于原先每一时刻只有 1 个线程在 *cur_bitmap* 中读取到 1 位的情况, 最坏的时间复杂度为 $O(deg\times K)$. 我们将在实验中测量启用该优化后对 warp 中平均活跃线程数量的提升.

将调度队列设置为双缓冲区形式是出于对内存优化和 thread 并行调度优化的考虑. 对于稠密的枚举任务, 位图空间中将包含大量 1 位. 如果预先筛出所有 *cur_bitmap* 中的所有 1 位, 将占用极大的存储空间. 抢占式任务调度将给线程级调度带来较大开销, 并且也需要较大的任务队列[41,42]. 相比基础算法, 在空间复杂度上双缓冲区只需多开辟一个大小固定为 64 个任务单元的存储空间, 最坏空间复杂度为 $O(2\times warp\_size\times warp\_num)$.

此外, 将筛选框设定为逐位统一移动确保加载到缓冲区中的任务需要加载的位图存储位置相近. 这是因为每个 *cur_bitmap* 中第 i 位的 1 位表示加载位图空间中的第 i 个位图, 同一个 Cycle 中识别出有效任务会加载相同的位图. 这有利于在 warp 内不同的线程访存位置相同或邻近, 是一种面向非规则访存问题的优化.

6.2 动态构建位图算法的开销和收益

对于基础算法, 固定在 Level 1 构建位图, 使用位图算法求解 K-团枚举的时间复杂度为 $O\left(deg(v_0)^2\times K\right)$.

对于改进算法, 该方法会产生两种类型的开销: 位图空间稀疏度测量和二分搜索的位图构建. 在尝试构建位图时用 *__popc*() 完成一次累加计算, 即可得到当前层构建出的位图空间的稀疏度, 开销很低. 增加的用二分搜索构建位图的开销等同于直接使用二分搜索进行团枚举的开销, 需要设置合适的阈值从而在合适的 Level 启用位图算法.

在更深的层次上构建位图可以提高位图的计算效率, 在第 k 层构建位图时后续计算的时间复杂度为 $O(deg(\bigcap_{i=1}^{k}N(v_i))^2\times(K-k))$. 虽然 1 位的总数和执行集合求交的次数不会减少, 但每个位图的长度和 0 位的数量可以显著减少. 这可以大幅降低对位图进行按位与操作的计算开销. 另外, 这也可以使位图算法不需要遍历大量无效的 0 位. 以此提高计算效率, 同时减少冗余操作.

另一方面, 在更深的层次上构建位图可以直接适配以边为任务划分的负载均衡优化. 例如在第 2 层构建位图时, 可以直接从边集 E 中取一个边 e 分配给 warp, 对边 e 的两点用二分搜索做集合求交后, 以求交得到的点集作为子图构建位图.

7 实验分析

7.1 实验设置

● 实验平台. 我们在一台搭载 8 张 NVIDIA V100 GPU 的服务器上评估 KCMiner, 以及 K-团枚举基线工作. 每张 GPU 具有 80 个流式多处理器 (SM) 和 32 GB 内存, 服务器上配备 2.10 GHz E5-2620 v2 Intel Xeon CPU, 用以运行 CPU 的 K-团枚举算法. 服务器的操作系统是 Linux kernel-5.15.0, NVIDIA nvcc 编译器版本为 12.2.

● 数据集. 我们选择与 KCGPU 相同的数据集, 这是目前 GPU 上最先进的 K-团枚举工作. 我们使用 5 个真实世界的图数据集来证明 KCMiner 的性能, 如表 1 所示. 这些图是从 SNAP 存储库中获得的, 通常用于评估子图匹配系统的相关工作.

我们测量了每个算法的运行时间, 不包括从磁盘读取图数据集所花费的时间. 在默认情况下, KCMiner 使用边作为划分枚举任务的单位, 每个 warp 领取一条边启动一个枚举任务.

表 1 图数据集基本参数

数据集	顶点数	边数	最大度	有向化后最大度
as-skitter	1.7M	22M	35 455	231
com-dblp	426k	2M	343	113
Orkut	3.1M	213M	33 313	535
Friendster	65.6M	3.6B	5 214	868
Livejournal	4.9M	85.7M	20 333	686

7.2 实验结果与分析

7.2.1 整体性能

为评估 KCMiner 是否比其他 *K*-团枚举工作的端到端性能更优, 我们将 KCMiner 与 3 种类型最先进的工作进行性能比较: *K*-团枚举的 GPU 实现 KCGPU, 它使用了位图算法并启用 thread 并行; 两个包含 *K*-团枚举的 GPU 子图匹配系统 G2Miner 和 GraphFold. G2Miner 不使用位图算法, 全部使用二分搜索完成集合求交, GraphFold 在 *K*-团枚举问题中开启了 warp 并行的位图算法优化; *K*-团枚举的 CPU 实现 EBBkC[38], 我们使用 64 线程运行 EBBkC.

表 2 对比了 KCMiner 与 GPU 优化的 *K*-团枚举算法 KCGPU 在真实世界图数据集上的端到端执行时间. 实验结果表明, KCMiner 在整体性能上显著优于 KCGPU, 最高可实现 7.36 倍的加速, 平均加速比为 2.94 倍.

表 2 KCMiner 与其他 *K*-团枚举问题求解算法的运行时间 (s)

K	数据集	EBBkC	G2Miner	GraphFold	KCGPU	KCMiner
4	as-skitter	5.75	0.034	0.029	0.034	**0.031**
	com-dblp	0.95	0.006	0.012	0.008	**0.003**
	Orkut	83.5	0.596	0.491	0.426	**0.457**
	Friendster	63.74	7.543	3.566	10.215	**3.257**
	Livejournal	46.17	0.325	0.307	0.214	**0.102**
5	as-skitter	8.83	0.119	0.078	0.069	**0.033**
	com-dblp	1.02	0.016	0.019	0.016	**0.005**
	Orkut	127.6	2.413	1.309	1.014	**0.81**
	Friendster	164.7	15.404	5.624	11.796	**5.567**
	Livejournal	283.5	11.957	2.612	0.957	**0.845**
6	as-skitter	18.5	1.098	0.558	0.245	**0.101**
	com-dblp	2.13	0.343 8	0.055	0.042	**0.013**
	Orkut	243.8	11.416	5.66	3.506	**2.019**
	Friendster	284.6	30.857	10.76	17.22	**10.046**
	Livejournal	6 350.7	579.471	132.68	31.34	**23.474**
7	as-skitter	87.7	12.153	2.654	1.434	**0.314**
	com-dblp	20.12	7.381	1.025	0.545	**0.074**
	Orkut	503.7	58.305	25.96	11.719	**4.27**
	Friendster	896.3	88.899	21.712	45.697	**12.16**
	Livejournal	>144000	>144000	6 146.57	1 386.73	**442.96**
8	as-skitter	523.3	115.2	28.881	9.531	**3.681**
	com-dblp	230.5	134.259	12.532	9.031	**1.389**
	Orkut	1 874.1	295.927	130.64	45.319	**25.508**
	Friendster	>144 000	673.547	213.58	99.866	**33.086**
	Livejournal	>144000	>144000	>144000	96 450	**44 616**

GPM 框架中通常包含 *K*-团枚举的 GPU 实现, 我们选取 G2Miner 和 GraphFold 两个最先进的 GPM 框架来比较性能. 结果显示, 与 G2Miner 相比, 最高可实现 99.1 倍的加速比, 与 GraphFold 相比, KCMiner 最高可实现 30.2 倍的加速比.

为了证明 *K*-团枚举是一项适合在 GPU 上优化的任务, 我们将 KCMiner 与 EBBkC 进行比较. 如表 2 所示, 与

EBBkC 相比, KCMiner 的平均加速为 187.8 倍, 最高可达 304.1 倍. 此外, EBBkC 无法支持 Friendster 和 Livejournal 等大型图数据.

总体结果表明, KCMiner 有效地提高了 K-团枚举在 GPU 上的性能. 与 3 种最先进的 K-团枚举方法相比, 它实现了显著的加速. 这主要得益于以下 3 个方面: (1) KCMiner 的线程间任务调度算法提高了 GPU 的线程利用率; (2) KCMiner 根据稀疏度动态构建位图空间, 提高位图算法的计算效率; (3) KCMiner 根据枚举任务的尺寸适配特定的 GPU 优化技术.

与此同时, 根据实验结果我们发现, 当问题规模较小时 (K=4), KCMiner 的优化效果不明显, 甚至性能有小幅退化. 这是因为搜索层数浅, 开启线程间任务调度的开销不能被性能收益所掩盖, 且位图算法中构建位图的开销也占比较大. 当问题规模增大时, KCMiner 的优化效果逐渐得以凸显.

7.2.2 消融实验

为了分别展示线程间任务调度、位图动态构建算法和 GPU 友好的算法实现带来的性能贡献, 我们以 thread 并行的 K-团枚举 GPU 开始设置了 5 组对比实验, 分别是 thread 并行的原始算法、单独使用任务调度、单独使用位图动态构建算法、任务调度叠加位图动态构建算法和进一步叠加 GPU 友好的算法实现.

为了便于统一比较使用每种优化方法后端到端时间减少的程度, 我们对每组图数据集的性能测试结果进行了归一化. 以 thread 并行的性能为基线, 其他性能以此为基准进行归一化. 如图 7 所示, 线程任务调度带来的性能提升最为显著, 与 thread 并行相比运行时间平均减少 60.3%. 单独使用位图构建时, 与 thread 并行相比运行时间平均减少 28.5%. 除此之外, 叠加位图动态构建和 GPU 友好的算法实现后可以进一步提升性能, 运行时间平均降低 22.6% 和 37.5%. 这表明 KCMiner 中的优化方法可以相互兼容.

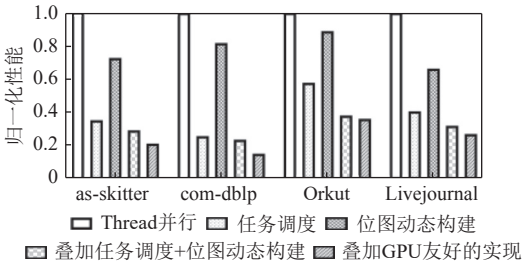


图 7 7-团枚举消融实验结果

7.2.3 线程间任务调度的优化效果

此外, 我们使用 NVIDIA Nsight Compute 性能分析工具对 KCMiner 进行了测试. 活跃线程数是指 warp 内同时执行同一条指令的线程数. 当一个 warp 中所有线程都在执行同一条指令时, GPU 的效率达到最大化, 活跃线程数将达到 32, 充分利用了其 SIMT 架构的并行计算能力. 如后文图 8 所示, 随着 K-团规模增大, 不使用任务调度时 warp 内平均活跃线程数逐渐减少, 而使用线程间任务调度后可使 warp 内活跃线程数的减少幅度降低. 在 7-团枚举中, 使用线程任务调度后, 每个 warp 中的平均活跃线程数平均增加了 2.6 倍. 这表明线程任务调度策略缓解了线程分歧问题.

7.2.4 敏感性分析

为了探索不同稀疏度下在不同层级构造位图的性能差异, 我们将枚举任务按稀疏度划分进行测试. 我们计算了用每条边的两个端点的公共邻居集合作为子图构建位图空间的稀疏度 γ, 并将每个 γ 间隔内的任务划分为一个任务区间. 每个任务区间执行两次, 一次在第 2 层构建位图空间, 即用输入边的两点公共邻居作为子图构建位图. 另一次在第 3 层构建位图空间. 如图 9 所示, 实验结果证实了本文之前的见解, 不同位图构建策略下 K-团枚举的性能并不是恒定的. 在 7-团枚举中, 在第 3 层构建位图的性能始终优于在第 2 层构建位图. 但是, 对于 6-团枚举, 只有当稀疏度小于 25% 时, 在第 3 层构建位图才是更优选择的. 这是更深的位图构建开销与我们之前介绍的性能

优势之间的权衡. 当整体搜索深度不够时, 额外增加 1 层二分查找带来的开销无法被位图算法的性能优势所抵消. 这个实验指导我们应该根据指定的枚举规模 K 设置不同的 γ 阈值. 每一层也应当设置不同的 γ 阈值.

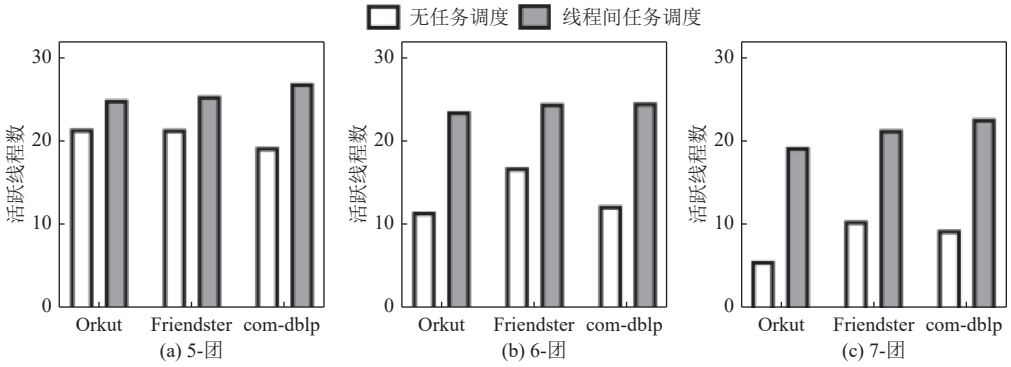


图 8 K-团枚举中任务调度前后每个 warp 平均活跃线程数变化

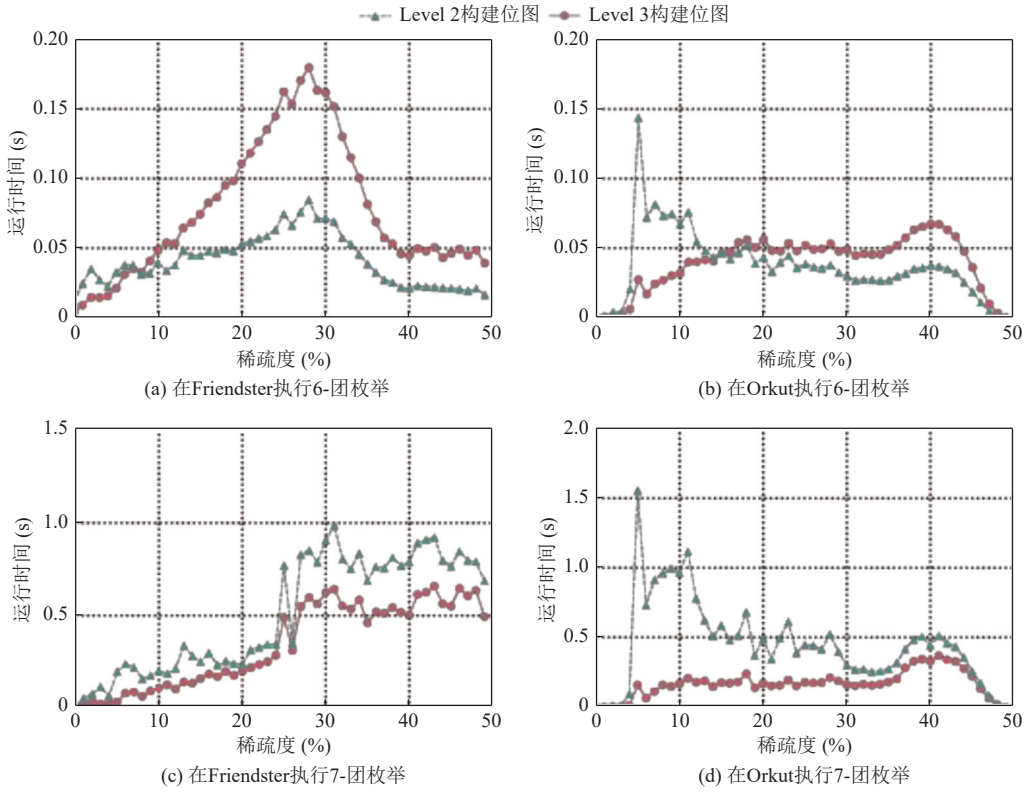


图 9 不同层构建位图的性能差异

7.2.5 可扩展性测评

为了验证 KCMiner 在多 GPU 环境运行的可扩展性, 我们在一台配备 8 个 NVIDIA V100 GPU 的机器上进行实验. 为了使 KCMiner 适应多 GPU 配置, 我们将数据图的边列表用轮询的方式切分为多个独立的边集合, 并打包分配给每个 GPU. 图 10 显示, 随着 GPU 数量增加, 在 5-团和 6-团上可见线性扩展, 8 卡加速比最高达到 7.53×和 7.36×. 但当问题规模增大, 搜索总层数加深时, 7-团中扩展效果略有衰减, 8 卡加速比最高只能达到 5.25×. 这是因为搜索层数增多后, 每条边导出的枚举任务负载不均现象更为显著, 而本工作只是以边列表为任务划分, 等同于在

第 2 层划分任务, 需要在更深的搜索层上划分任务或实现 warp 之间的任务调度才能进一步提升可扩展性. 在施加多 GPU 的扩展后, 与 KCGPU 只支持单卡环境相比, KCMiner 的性能提升可达 29.37 倍.

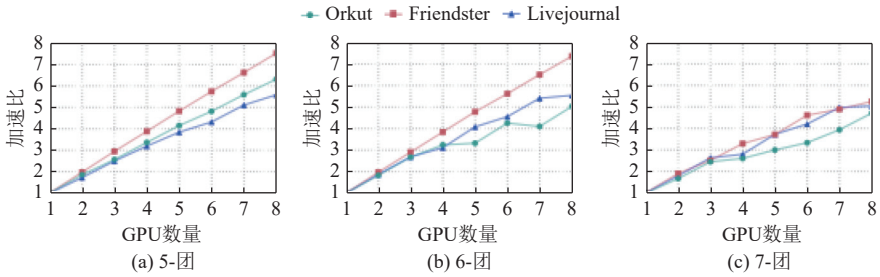


图 10 KCMiner 的多 GPU 可扩展性

8 总 结

在本文中, 我们提出了 KCMiner, 一种基于位图的 *K*-团枚举问题 GPU 优化算法. 现实世界图数据的稀疏性使位图算法成为一个稀疏问题, 它带来了严峻的挑战. 这些挑战包括低效的静态并行方法、低效位图算法的使用, 以及不同枚举任务之间的优化场景变化. 为了应对这些挑战, 本文首先提出了一种基于 thread 并行的负载平衡位图任务调度策略. 为了确保高效使用位图, 设计了一种稀疏感知的动态位图构造算法. 此外, KCMiner 通过输入分析识别任务特征, 从而能够根据特定任务要求自适应地选择优化策略. 在 NVIDIA V100 GPU 上使用真实数据集进行全面的评估, 实验结果表明, KCMiner 的性能比最先进的 GPU 并行 *K*-团枚举算法 KCGPU 最多提高了 7.36 倍.

References

[1] Ebrahimi F, Asemi A, Nezarat A, Ko A. Developing a mathematical model of the co-author recommender system using graph mining techniques and big data applications. Journal of Big Data, 2021, 8(1): 44. [doi: 10.1186/s40537-021-00432-y]

[2] Kutty S, Nayak R, Chen L. A people-to-people matching system using graph mining techniques. World Wide Web, 2014, 17(3): 311–349. [doi: 10.1007/s11280-013-0202-z]

[3] Akoglu L, Tong HH, Koutra D. Graph based anomaly detection and description: A survey. Data Mining and Knowledge Discovery, 2015, 29(3): 626–688. [doi: 10.1007/s10618-014-0365-y]

[4] Noble CC, Cook DJ. Graph-based anomaly detection. In: Proc. of the 9th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Washington: ACM, 2003. 631–636. [doi: 10.1145/956750.956831]

[5] Deshpande M, Kuramochi M, Wale N, Karypis G. Frequent substructure-based approaches for classifying chemical compounds. IEEE Trans. on Knowledge and Data Engineering, 2005, 17(8): 1036–1050. [doi: 10.1109/TKDE.2005.127]

[6] Kuhl FS, Crippen GM, Friesen DK. A combinatorial algorithm for calculating ligand binding. Journal of Computational Chemistry, 1984, 5(1): 24–34. [doi: 10.1002/jcc.540050105]

[7] Rupp M. Graph kernels for chemoinformatics—A critical discussion. Journal of Cheminformatics, 2011, 3(1): O8. [doi: 10.1186/1758-2946-3-S1-O8]

[8] Milo R, Shen-Orr S, Itzkovitz S, Kashtan N, Chklovskii D, Alon U. Network motifs: Simple building blocks of complex networks. Science, 2002, 298(5594): 824–827. [doi: 10.1126/science.298.5594.824]

[9] Oliveira M, Gama J. An overview of social network analysis. WIREs Data Mining and Knowledge Discovery, 2012, 2(2): 99–115. [doi: 10.1002/widm.1048]

[10] Feng S, Shen DR, Nie TZ, Kou Y. Maximum common subgraph based social network alignment method. Ruan Jian Xue Bao/Journal of Software, 2019, 30(7): 2175–2187 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5831.htm [doi: 10.13328/j.cnki.jos.005831]

[11] Rossi RA, Zhou R. GraphZIP: A clique-based sparse graph compression method. Journal of Big Data, 2018, 5(1): 10. [doi: 10.1186/s40537-018-0121-z]

[12] Feder T, Motwani R. Clique partitions, graph compression and speeding-up algorithms. In: Proc. of the 23rd Annual ACM Symp. on Theory of Computing. New Orleans: ACM, 1991. 123–133. [doi: 10.1145/103418.103424]

[13] Glaria F, Hernández C, Ladra S, Navarro G, Salinas L. Compact structure for sparse undirected graphs based on a clique graph partition. Information Sciences, 2021, 544: 485–499. [doi: 10.1016/j.ins.2020.09.010]

[14] Manoharan S, Ammaiappan S. Patient diet recommendation system using K clique and deep learning classifiers. Journal of Artificial Intelligence and Capsule Networks, 2020, 2(2): 121–130. [doi: 10.36548/jaicn.2020.2.005]

[15] Vilakone P, Park DS, Xinchang K, Hao F. An efficient movie recommendation algorithm based on improved *k*-clique. Human-centric Computing and Information Sciences, 2018, 8(1): 38. [doi: 10.1186/s13673-018-0161-6]

[16] Fang YX, Yu KQ, Cheng R, Lakshmanan LVS, Lin XM. Efficient algorithms for densest subgraph discovery. Proc. of the VLDB Endowment, 2019, 12(11): 1719–1732. [doi: 10.14778/3342263.3342645]

[17] Yuan L, Qin L, Zhang WJ, Chang LJ, Yang JY. Index-based densest clique percolation community search in networks. IEEE Trans. on Knowledge and Data Engineering, 2018, 30(5): 922–935. [doi: 10.1109/TKDE.2017.2783933]

[18] Gregori E, Lenzini L, Mainardi S. Parallel *k*-clique community detection on large-scale networks. IEEE Trans. on Parallel and Distributed Systems, 2013, 24(8): 1651–1660. [doi: 10.1109/TPDS.2012.229]

[19] Tsourakakis C. The K-clique densest subgraph problem. In: Proc. of the 24th Int'l Conf. on World Wide Web. Florence: Int'l World Wide Web Conferences Steering Committee, 2015. 1122–1132. [doi: 10.1145/2736277.2741098]

[20] Chiba N, Nishizeki T. Arboricity and subgraph listing algorithms. SIAM Journal on Computing, 1985, 14(1): 210–223. [doi: 10.1137/0214017]

[21] Almasri M, El Hajj I, Nagi R, Xiong JJ, Hwu WM. Parallel K-clique counting on GPUs. In: Proc. of the 36th ACM Int'l Conf. on Supercomputing. New York: ACM, 2022. 21.

[22] Chen XH, Dathathri R, Gill G, Pingali K. Pangolin: An efficient and flexible graph mining system on CPU and GPU. Proc. of the VLDB Endowment, 2020, 13(8): 1190–1205. [doi: 10.14778/3389133.3389137]

[23] Lin ZH, Meng K, Shui CY, Zhang KW, Xiao JM, Tan GM. Exploiting fine-grained redundancy in set-centric graph pattern mining. In: Proc. of the 29th ACM SIGPLAN Annual Symp. on Principles and Practice of Parallel Programming. Edinburgh: ACM, 2024. 175–187. [doi: 10.1145/3627535.3638507]

[24] Latapy M. Main-memory triangle computations for very large (sparse (power-law)) graphs. Theoretical Computer Science, 2008, 407(1–3): 458–473. [doi: 10.1016/j.tcs.2008.07.017]

[25] Ortmann M, Brandes U. Triangle listing algorithms: Back from the diversion. In: Proc. of the 16th Workshop on Algorithm Engineering and Experiments. Portland: Society for Industrial and Applied Mathematics, 2014. 1–8.

[26] Tom AS, Sundaram N, Ahmed NK, Smith S, Eyerman S, Kodiyath M, Hur I, Petrini F, Karypis G. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In: Proc. of the 2017 IEEE High Performance Extreme Computing Conf. (HPEC). Waltham: IEEE, 2017. 1–7. [doi: 10.1109/HPEC.2017.8091054]

[27] Hu L, Zou L, Liu Y. Accelerating triangle counting on GPU. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: ACM, 2021. 736–748. [doi: 10.1145/3448016.3452815]

[28] Finocchi I, Finocchi M, Fusco EG. Clique counting in MapReduce: Algorithms and experiments. Journal of Experimental Algorithmics, 2015, 20: 1.7. [doi: 10.1145/2794080]

[29] Danisch M, Balalau O, Sozio M. Listing k-cliques in sparse real-world graphs*. In: Proc. of the 2018 World Wide Web Conf. Lyon: Int'l World Wide Web Conferences Steering Committee, 2018. 589–598. [doi: 10.1145/3178876.3186125]

[30] Li RH, Gao S, Qin L, Wang GR, Yang WH, Yu JX. Ordering heuristics for *k*-clique listing. Proc. of the VLDB Endowment, 2020, 13(12): 2536–2548. [doi: 10.14778/3407790.3407843]

[31] Mawhirter D, Wu B. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 509–523. [doi: 10.1145/3341301.3359633]

[32] Shi TH, Zhai MS, Xu Y, Zhai JD. GraphPi: High performance graph pattern matching through effective redundancy elimination. In: Proc. of the 2020 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2020. 1–14. [doi: 10.1109/SC41405.2020.00104]

[33] Chen XH, Arvind. Efficient and scalable graph pattern mining on GPUs. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation (OSDI 22). Carlsbad: USENIX Association, 2022. 857–877.

[34] Chen JJ, Qian XH. DecoMine: A compilation-based graph pattern mining system with pattern decomposition. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (Vol. 1). Vancouver: ACM, 2023. 47–61. [doi: 10.1145/3567955.3567956]

[35] Al Hasan M, Dave VS. Triangle counting in large networks: A review. WIREs Data Mining and Knowledge Discovery, 2018, 8(2): e1226. [doi: 10.1002/widm.1226]

[36] Bisson M, Fatica M. Update on static graph challenge on GPU. In: Proc. of the 2018 IEEE High Performance Extreme Computing Conf. (HPEC). Waltham: IEEE, 2018. 1–8. [doi: 10.1109/HPEC.2018.8547514]

[37] Eppstein D, Löffler M, Strash D. Listing all maximal cliques in sparse graphs in near-optimal time. In: Proc. of the 21st Int'l Symp. on Algorithms and Computation. Jeju Island: Springer, 2010. 403–414. [doi: 10.1007/978-3-642-17517-6_36]

[38] Wang KX, Yu KQ, Long C. Efficient k-clique listing: An edge-oriented branching strategy. Proc. of the ACM on Management of Data, 2024, 2(1): 7. [doi: 10.1145/3639262]

[39] Shi J, Dhulipala L, Shun J. Parallel clique counting and peeling algorithms. In: Proc. of the 2021 SIAM Conf. on Applied and Computational Discrete Algorithms (ACDA21). Society for Industrial and Applied Mathematics, 2021. 135–146. [doi: 10.1137/1.9781611976830.13]

[40] Yuan ZR, Peng Y, Cheng P, Han L, Lin XM, Chen L, Zhang WJ. Efficient *k*-clique listing with set intersection speedup. In: Proc. of the 38th IEEE Int'l Conf. on Data Engineering (ICDE). Kuala Lumpur: IEEE, 2022. 1955–1968. [doi: 10.1109/ICDE53745.2022.00192]

[41] Wei YH, Jiang P. STMatch: Accelerating graph pattern matching on GPU with stack-based loop optimizations. In: Proc. of the 2022 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2022. 1–13. [doi: 10.1109/SC41404.2022.00058]

[42] Pan Z, He SB, Li X, Zhang XC, Wang R, Chen G. Efficient maximal biclique enumeration on GPUs. In: Proc. of the 2023 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2023. 16. [doi: 10.1145/3581784.3607059]

附中文参考文献

[10] 冯朔, 申德荣, 聂铁铮, 寇月, 于戈. 一种基于最大公共子图的社交网络对齐方法. 软件学报, 2019, 30(7): 2175–2187. http://www.jos.org.cn/1000-9825/5831.htm [doi: 10.13328/j.cnki.jos.005831]

作者简介

曹炜辰, 博士生, CCF 学生会员, 主要研究领域为高性能计算, 图计算.

孟轲, 博士, 副研究员, CCF 专业会员, 主要研究领域为高性能计算, 并行算法设计与优化, 图计算.

林志恒, 博士生, CCF 学生会员, 主要研究领域为高性能计算, 异构计算, 大规模图数据处理.

谭光明, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为并行算法设计与分析, 并行编程和优化, 计算机体系结构, 生物信息学, 大数据.