计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(9):82−90 [doi: 10.15888/j.cnki.csa.008664]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

支持分页显存的高性能哈希表索引系统①

熊轶翔[1,2], 蒋筱斌[1,2], 张　珩[1], 武延军[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100049)
通信作者: 张　珩, E-mail: zhangheng17@iscas.ac.cn

摘　要: 哈希表以访问效率时间复杂度 O(1) 著称, 作为一类可提供大规模数据高效访问的算法和数据结构为各类大数据应用所采用, 例如, 适用于各类新兴高性能 (HPC) 领域、数据库领域的工作负载和场景. 随着高性能协处理器 GPU 硬件性能的日益提升, 面向高性能 GPU 环境的哈希表并行优化已逐渐吸引了大量研究工作. 当前的各类 GPU 哈希表优化方法和解决方案集中于利用 GPU 的大规模线程环境和高内存带宽来提升哈希表的事务高并发性处理和键值对数据快速访问. 然而, 由于现有 GPU 哈希表结构的研究工作普遍忽略了 GPU 资源有效管理, 并没有以如何充分利用 GPU 线程资源和显存资源. 同时, 由于 GPU 显存空间的大小限制, 用于存储哈希表结构数据的空间有限, 无法应对更大规模的哈希表结构. 因此, 面向 GPU 环境下的哈希表方法的可扩展性和性能仍存在着技术挑战. 本文提出并设计了一种面向 GPU 环境的可处理大规模并发事务的哈希表技术, 命名为 Starfish. Starfish 提出了新的基于异步 GPU 流的“交换层” (swap layer) 技术, 用以支持 GPU 显存外的动态哈希表, 同时也保障了 GPU 哈希表的索引方法性能. 为了解决 GPU 大规模线程的访问带来的哈希冲突开销, Starfish 设计了一类紧凑型数据结构, 并研究了一种可分页显存的分配方法, 不仅为 GPU 哈希表技术提供了静态哈希方法的高性能, 而且也支持动态哈希的高可扩展性. 性能评估实验表明, Starfish 显著优于其他 GPU 哈希表技术, 包括 cudpp-Hash, SlabHash.

关键词: 哈希表; GPU; 分页存储; 交换; 索引技术

引用格式: 熊轶翔,蒋筱斌,张珩,武延军.支持分页显存的高性能哈希表索引系统.计算机系统应用,2022,31(9):82–90. http://www.c-s-a.org.cn/1003-3254/8664.html

High-performance Hash Table Indexing System Supporting Paging Memory

XIONG Yi-Xiang[1,2], JIANG Xiao-Bin[1,2], ZHANG Heng[1], WU Yan-Jun[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Hash tables are well known for their access efficiency and time complexity O(1). As an algorithm and data structure available for efficient access to large-scale data, hash tables have been widely adopted by big data applications. For example, they are applicable to various kinds of workloads and scenarios in the emerging high-performance computing (HPC) domain and the database domain. As the hardware performance of the high-performance coprocessor graphics processing unit (GPU) improves continuously, parallel hash table optimization for high-performance GPUs has attracted a lot of researchers. According to our previous research survey, most of the current methods and solutions for GPU-based hash table optimization focus on the large-scale thread scheduling and high memory bandwidth of GPUs to improve the high-concurrency processing of hash table transactions and fast key-value access to data. However, since existing research on GPU hash table structures generally ignores the effective management of GPU resources, no methods are available for making full use of GPU thread resources and memory resources. Moreover, due to the limitation of the

① 收稿时间: 2021-12-04; 修改时间: 2022-01-04; 采用时间: 2022-01-07; csa 在线出版时间: 2022-05-30

© 中国科学院软件研究所　http://www.c-s-a.org.cn

GPU memory size, the space for storing data on hash table structures is limited and therefore unable to handle hash table structures of larger scales. Thus, technical challenges remain for the scalability and performance optimization of the hash table designs for GPUs. This study proposes a hash table technology that can process massive concurrent transactions for GPUs and names it Starfish. Starfish includes a novel "swap layer" technology based on asynchronous GPU streams to support the dynamic hash table outside the GPU memory and also guarantee the high indexing performance of GPU hash tables. To solve the massive hash transaction conflict overhead resulting from access of large-scale GPU threads, we design a class of compact data structures and a pageable memory distribution method, which not only provides the high performance of a static hash method for the GPU-based hash table technology but also supports the high scalability of dynamic hash. Our experimental performance evaluation shows that Starfish is significantly superior to other GPU-based hash table technologies including cudpp-Hash and SlabHash.

Key words: hash table; GPU; paging storage; swap; indexing technology

1 引言

哈希表 (hash table), 也称散列表, 是一种根据关键码值技术 (key value) 来为大规模数据提供直接访问操作的数据结构, 其通过将关键码值映射到表中的一个位置来访问记录数据, 以 O(1) 时间复杂度来加快数据查找速度. 哈希表作为一种广泛采用的基础数据结构和核心算法, 为程序提供对各类数据 (文本、流媒体、结构型数据等) 高吞吐、低时延的增删查改操作. 随着数据规模的不断扩大和各类应用对高性能的需求, 哈希表技术在当前数据库软件、系统软件和应用软件中都得到了大规模采用.

在深度学习框架 TensorFlow 和 PyTorch 对稀疏矩阵的处理中, 哈希表的应用大幅度提高存储效率, 也保证了数据块的吞吐低延迟; 在大规模数据处理应用 (图处理[1,2]) 中, 哈希表的高效查询作用至关重要. 此外, 哈希表作为数据库系统的核心模块[3–6], 频繁地被大规模并发事务的索引、更新、删除等操作调用. 在游戏应用领域, 对动辄上百万体素的游戏模型修改和访问[7], 哈希表提供系统的可靠性保障. 由于哈希表作为各类系统的核心数据结构来使用, 因此, 对于哈希表的高性能优化能使整体系统的性能、可扩展性各方面得到大幅度提升[6,8,9]. 如何进一步优化哈希表性能以及在各类新型的并行计算机架构上设计高性能哈希表一直是学术界和工业界重点关注的研究领域. 当前的研究对哈希表的数据结构设计、并行度优化、索引冲突等多方面开展各类工作. 在并行哈希表的优化上, 这类研究工作多基于多核 CPU 架构开展, 设计了面向非统一内存访问 (NUMA)、对称多处理 (SMP) 等高性能服务器架构的哈希表结构优化[10], 并通过融合其他数据结构 (如索引树形、环形缓存队列等) 以提高哈希表的并行处理性能, 降低多线程下的哈希索引冲突. 随着高性能的体系结构发展, 通用图形处理器 (GPU) 作为协处理器的性能得到了大幅度提升. 单块 GPU 处理单元提供了万级以上线程数的高并行计算能力和高于 100 GB/s 显存带宽, 大幅提高高通量工作负载的性能, 例如, 单块 NVIDIA V100 GPU 卡提供了多达 5 120 流处理单元. 高性能哈希表的研究工作开始利用协处理 GPU 的大规模并行和高性价比的优势来优化哈希表的大规模并发散列事务操作[7,11], 例如, 英伟达 cudpp[12,13] 设计静态哈希操作, 显著地提高了数据索引性能, 相对比 sorted array [14]、cuckoo[15] 等各类 CPU 索引方法性能提升 1–3 倍[12,16]. 在对于 GPU 哈希操作的优化中, 相同哈希值的键值对无法存放在哈希表中的相同位置导致了 GPU 大规模线程间哈希冲突 (溢出处理) 开销, 因此 GPU 哈希表工作着重于优化 GPU 内并行处理来离散线程间访问, 降低索引冲突. 进而, GPU 哈希表研究衍生出动态哈希和静态哈希两类工作.

通过前期的调研, 目前主流的 GPU 哈希操作主要使用 cuckoo 哈希算法的静态哈希, 在插入和查找时采用固定大小的 GPU 显存预分配, 因而其并行度更为线性, 整体性能高, 然而由于预分配需要占用大量 GPU 显存空间, 导致 GPU 显存利用率地下, 缺乏可扩展性; 而使用“桶”策略的动态哈希表在具备了可扩展性的同时弹性的“桶”缓存分配带来了一定的显存操作开销, 其整体性能略逊于静态哈希策略. 具体地, 表 1 给出了当前 GPU 哈希策略方法的优缺点对比.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

表 1 静态哈希与动态哈希对比

哈希策略	具体实现	优点	缺点
静态哈希[12,13]	提前分配固定大小GPU显存缓存空间, 使用缓存替换操作来处理哈希冲突.	性能稍优于动态哈希结构.	(1) 处理哈希冲突时成功率不高; (2) 无法支持动态哈希事务插入的功能; (3) 哈希冲突失败的重新建表开销过高.
动态哈希[17]	构建GPU显存Slab分配机制, 以线程组warp动态分配, “桶”机制规避哈希冲突.	(1) 大幅降低哈希冲突开销; (2) 支持动态插入.	动态的分配GPU显存整体的事务吞吐性能差于静态哈希.

然而, 高性能哈希表的研究多采用单一设计模式, 即选择性地构建静态哈希或动态哈希, 没有很好融合这两类哈希表的优点, 同时, 由于 GPU 显存占用率非常高, 在显存不足以存放整张哈希表时整体系统无法正常运行. 综上, 现有 GPU 高性能哈希表的研究工作仍然存在如下两类缺陷: (1) 由于 GPU 全局显存的局限性 (如 V100 GPU 提供 32 GB DDR5 全局显存), 现有工作受限于存放大规模的哈希表结构, 无法支持超过 GPU 显存的哈希事务, 整体系统的可扩展性和弹性受到限制; (2) 随着数据量不断增大, 导致以多核优化的哈希表方法无法满足高性能的数据索引操作. GPU 显存资源在同一时间会有多个任务同时需要占用, 导致留给高性能哈希表的空间不足以处理更多的并发访问事务. 当空间不足但是任务所需要处理的数据量较大时, 目前的哈希表设计就略显不足了. 因此, 对哈希表结构的空间结构的压缩和数据结构的分布优化显得至关重要.

本文聚焦于结合静态哈希和动态哈希各自的优点, 借鉴 Linux 的页面交换机制, 设计一种兼具高性能和高扩展性且对于显存空间受限 GPU 架构的全新高性能哈希表结构和方法. 本文将传统的哈希表分割为多个子表 (文中可能同时称作子表或 ChildTable), 并使用子表来支持 GPU 优化, 通过 stream 支持流水线 insert 操作, 利用 GPU 多线程机制支持 search 以及 delete, 相对现有工作提升了 insert 操作性能 1.5–2 倍. 本文主要贡献如下:

(1) 针对 GPU 显存占用过高以及静态哈希表插入性能缺陷问题, 本文首次提出多子表技术, 即构建层级页表机制来将结构过大的哈希表分解为多个小结构表来统一管理; 构建局部 GPU 显存驻留策略, 以支持同一时刻只需驻留少部分子表的 GPU 工作流, 大大降低 GPU 显存占用率.

(2) 进一步, 本文设计了 Starfish 原型系统, 以支持各类 GPU 哈希表操作, 不限于 select/insert 以及 update 操作, 并优化多子表技术将 insert 操作流水线化, 在提升并行度的同时大幅度降低插入失败时重新哈希表结构构建开销.

(3) 实验结果表明, 在多类现实标准数据集上的测试, Starfish 在 insert 以及重新构建的性能方面, 相对于 NVIDIA 最新的 GPU 哈希表工作 cudpp-Hash 显著提高了 1.5–2 倍; 显存占用量降低为 cudpp-Hash 的 5%–10% 以及 SlabHash 的 1%–2%.

2 相关工作

2.1 面向 GPU 的高性能哈希表

随着各类高性能协处理器单元 (如 GPU) 的架构性能不断提升, 当前的学术研究方向开始转向利用 GPU 服务器来设计高性能的哈希表操作. 相对比 CPU 处理器上有限线程的并行计算环境, 通过 GPU 协处理器上基于单指令多数据流 (SIMD) 芯片架构特性来构建全新的哈希表并行事务策略的处理性能得到了显著提升, 提升了若干数量级[12,13,18,19]. 各类并行哈希操作的优化工作利用 GPU 等大规模并行硬件特性来进行加速, 主要通过如下两个方面进行: 1) SIMT 架构为哈希表的并行操作提供了只读性事务的大规模并行; 2) 高内存带宽为哈希表的数据访问带来了大数据量的访问吞吐提升以及高效的原子性操作. GPU 哈希表 cudpp-Hash[12,13] 利用的 cuckoo 哈希方法以及公共溢出区策略, 第一次提出了标准的 GPU 下哈希操作, 被广泛应用于 NVIDIA 下的各类算法库[18,19]. 通常地, 现有 GPU 哈希工作主要分为两类: 一类是面向静态 GPU 显存空间分配的静态哈希表, 另一类是以动态页表方式提供弹性 GPU 显存优化的动态哈希表. 这两类工作各有所长, 具体如本节接下来所述.

2.2 静态哈希相关研究工作

面向 GPU 的静态哈希表都是基于 cuckoo 策略[15] 来处理哈希冲突, 但也有不使用 cuckoo 策略的哈希表设计, 例如 Khorasani 等人提出的 stadium hash[20] 在解决冲突的时候不会驱逐原有的键值对, 而是直接寻找下一个合适的位置. 最经典的 GPU 静态哈希表为 Alcantara 等人提出的 cudpp-Hash[12,13,16], 已被 Nvidia作为标准哈希库推出, 其算法策略采用 cuckoo 实现, 此外还加入了

© 中国科学院软件研究所 http://www.c-s-a.org.cn

stash 的机制来尽量减少插入失败的频率. cuckoo 的多个备用哈希函数机制会带来重复探测次数过多而导致的性能下降问题, 因此 Breslow 等人提出的 Horton table[21] 使用了一种紧凑的数据结构来存放键值对使用的备用哈希函数索引, 以减小重复探测的次数. 除此之外, 与 cudpp 类似且比较能适应大规模并行的哈希表还有使用罗宾汉哈希方法[22] 的 coherent parallel hashing[23], 但是同样面临着静态哈希的常见困境.

2.3 动态哈希相关研究工作

Ashkiani 等人提出了一种面向 GPU 的动态哈希表 SlabHash[17], 利用动态哈希“桶”的思想避开了处理哈希冲突的挑战, 而是在发生哈希冲突时将键值对直接插入目标“桶”中. 在进行事务操作过程中, 以 warp 为单位并大量使用 ballot、shuffle 等 warp 内部通信的 API 来加速处理. 此外还设计了 SlabAlloc 机制来以 warp 为单位动态分配新的 slab 空间. 这样的实现在用户看来是动态的, 且不会像静态哈希那样发生插入失败的问题, 但实际上是以较大的显存空间浪费为代价的.

作为最新的 GPU 哈希表研究工作, slab hash[17] 和 stadium hash[20] 基于 GPU 架构提出了当前性能和可扩展性较好的设计, 其中 slab hash 是一类典型的动态哈希表, stadium hash 是一种静态的哈希表. 此外, 已广泛应用于 NVIDIA RAPIDS 框架[19] 的 cuDF 哈希策略和 HashGraph[24] 两种哈希表的实现更加具有可扩展性和灵活性, 然而由于其采用过高显存频率操作而造成内存性能瓶颈, 导致了严重的性能抖动.

2.4 当前工作在局限 GPU 显存条件下存在的缺陷

从目前研究工作来看, GPU 哈希表策略大多基于将整个哈希表存放于显存中来设计的, 但是如果哈希表太大而 GPU 所剩的显存空间无法满足要求时, 这些设计就无法正确运行起来. 因此 Khorasani等人提出的 stadium hash[20] 提出了一种 out-of-core 的策略, 用户可以自行选择将哈希表存放于显存或者主存, 当存放于主存时, stadium hash 使用一种 ticket board 的机制来减少显存与主存之间不必要的 PCI-E 交换, 以此达到降低使用锁页内存带来的性能损耗. 另外同样使用类似 out-of-core 方法的哈希表还有 Hopscotch hashing[25] 和 Cache-Oblivious hashing[26].

3 挑战及应对

总结 GPU 哈希表设计面临的技术挑战如下.

3.1 挑战 1: 哈希表键值对在冲突处理的开销过大

传统的静态哈希表在发现某个键值对完全无法插入的情况下 (failure) 则会重建整张表, 当这种事件发生在整个建表的尾期时, 则将会将前期所有完成的工作都丢弃然后进行重新构建 (restart), 这样会造成极大的浪费. 因此本文拟提出一种 failure 处理机制, 将 failure 的影响范围缩小到一个较小的区间, 在这个区间内进行 restart, 将整个 failure 处理的过程都控制在这个区间内, 这样发生 failure 之后和进行 restart 都不至于影响整个哈希表. 发生 failure 的后果就是影响建表性能, 因此本文的设计旨在降低 failure 对于性能的影响.

3.2 挑战 2: 无法支持 GPU 显存外哈希表索引

由于 GPU 协处理器的显存资源有限, 因此每个应用程序在占用 GPU 显存时所能获得的资源多采用共享 GPU 显存模式, 即各自占用百分比显存空间, 这将导致在 GPU 显存空间受限情况下大规模哈希表结构无法完整缓存于 GPU 显存空间, 将导致整体的性能和可扩展性受限. 通过前期调研, 当前 GPU 哈希表的实现均在显存可容纳整张表或者整张表完全放在 pinned memory 的前提下开展研究, 然而有限显存空间无法容纳整张哈希表; 若干工作提出利用统一地址空间对 GPU 显存扩展[20], 然而这类哈希事务的访问仍需要利用 PCI-E 来进行大量的 host 端和 GPU device 端的数据交换操作, 导致数据访问效率低下. 因此, 本文提出了一套全新的 GPU 核外计算 (out-of-GPU-memory) 的策略, 将哈希表中暂时不用的子表缓存空间从显存中交换 (swap) 到 host 端主存中, 而当前需要访问的热度哈希表部分则驻留在 GPU 显存中. 此外, 通过自定义显存驻留表大小和数目, 本文设计了显存的占用的弹性伸缩机制, 根据用户配置项设置显存空间占用.

上述的全新哈希表结构设计对于局部性的 GPU 显存访问模式来说, 不会因为数据在主存中而降低读写速度, 反而会因为不常用的数据不占用显存空间而节省空间, 让同一个进程的其他部分甚至其他 CUDA 进程获得更多可用的显存空间.

3.3 挑战 3: 可扩展性与性能缺陷

针对当前的动态哈希和静态哈希优缺点和高性能的技术挑战, 本文提出了一种新的基于页表切分区域管理的哈希表数据结构表示, 以融合两种哈希表的高性能和高可扩展性优势. 进一步, 本文设计了哈希表原型系统 StarFish 以支持上层的高性能哈希应用, 支持各类增删改查的哈希表事务操作.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

4 方案设计

本文聚焦于结合动态哈希表和静态哈希表各自的优点, 基于分层页表机制提出了一种全新的 GPU 动态化显存页分配与静态操作的哈希表——Starfish, 并且具备低显存的能力.

4.1 Starfish 架构设计

面向 GPU 的哈希表系统 Starfish 的总体技术架构图如图 1 所示. 顶层架构分为两个部分: 主控逻辑端 (host) 的 CPU 处理器主内存的 Daemon 模块 (图 1 中的①)、事务性哈希表数据 I/O 模块②以及协处理器控制逻辑 GPU 执行部分 (kernel 计算服务模块③).

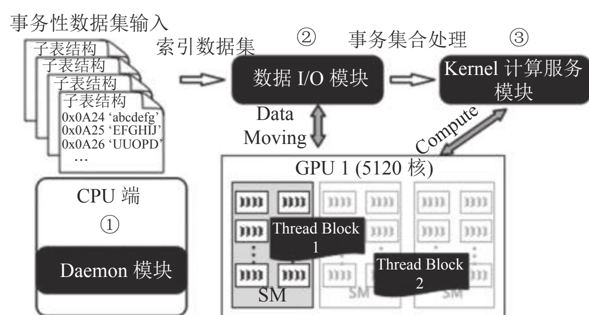

图 1　Starfish 整体架构

其中, Daemon 主控逻辑①用于管理整个系统, 记录了当前的子表数、驻留在 GPU 中的子表索引, 以及指向主存和显存中多个子表的指针. 一个页面 (子表) 就是一个可以向下兼容的完整的底层哈希表, 可以使用 cudpp-Hash、horton table 等来作为底层哈希表. 具体地, Starfish 的数据 I/O 模块②通过构建页表的数据结构 (ChildTable, 子表) 方式来对动态哈希页进行统一化管理; 所有的底层哈希表 (称为子表) 以固定内存页 (默认情况下 64 MB) 形式进行构建, 其组成形式以若干<Key, Value>结构数组队列驻留于 GPU 显存中, 同时同步一份缓冲备份于主存中. 在 GPU 显存中分配的空间小于所有的子表容量的总和, 因此同一时间内 GPU 显存中只缓存一定比例的子表数组, 如图 1 中显示只能容纳 4 个. 进一步, 为了克服 GPU 显存空间的大小局限性而无法处理大规模的哈希表, Starfish 在统一化 GPU 内存页管理基础上构建了高效的 I/O 缓冲区 (swap) 机制, 即在增删查改过程 (kernel 计算服务模块③) 中, 其他的未能缓存于 GPU 显存中的子表缓存页以轮转方式 (默认先进先出策略) 通过 PCI-E 总线加载至 GPU 显存, 执行并行的哈希事务操作. 每一个子表由若干 (默认为 100 万) 对键值对以 8 字节对齐的方式组成. 此外, 位于 host 端的 Daemon 里面主要包含两种指针, 第一种包含多个指向子表的指针, 另一种指针为指向 GPU 全局内存中为子表开辟的空间.

通过如上的架构设计, Starfish 构建了子表内存页的独立缓存空间, 通过对大规模的哈希表结构进行均衡切分和轮转实现局部独立处理. 此外, Starfish 系统支持动态增减子表的数目, 不仅能够为上层的哈希表的索引操作提供静态哈希表策略的高性能操作, 而且通过弹性化的 GPU 显存控制, 实现了动态化大规模哈希表应用.

4.2 面向局限 GPU 显存空间的哈希表事务优化

由于 GPU 显存空间的局限性, 大规模哈希表事务操作无法完整地利用 GPU 进行并行操作, 这也带来了目前各类哈希表的高性能操作的可扩展性局限. 为了解决大规模哈希表结构在 CPU-GPU 之间的高效吞吐问题, Starfish 基于 CUDA Streaming 对象策略, 在 ChildTable 数据结构基础上构建了高效的哈希表内存页置换策略. 通过设置哈希表数据缓存区大小阈值, Starfish 在哈希表数据量只够部分存放于 GPU 显存空间情况下, 自动地启用 CPU 与 GPU 间数据 I/O 轮转策略, 将整体的哈希操作进行均衡切分, 存放哈希表的部分于显存. 下面, 本小节将对 Starfish 所支持的哈希表操作的插入 (insert)、检索 (search) 等关键操作的实现流程进行详细阐述.

4.2.1 基于 CUDA Streaming 异步策略的插入操作

在哈希表插入操作 (insert) 过程中, 无需把所有的输入数据和整个哈希表都存放于显存中. 如图 2 所示, 在输入 (input) 哈希事务为 3 个事务数据列表时, 每段输入事务数据列表长度相等且都为默认的 100 万个键值对, 允许最后一段长度小于其他段. 在显存上只开辟两段对应的 input 事务列表空间以及两段 ChildTable 空间. 以图 2 为例, Starfish 的插入过程: (1) 考虑到 GPU 缓存空间局限, 将只对 input1 和 input2 的插入事务列表加载至 GPU 显存空间. 进一步据这两个事务集合构建 GPU 哈希表内存页 ChildTable1 和 ChildTable2; 进而, input3 插入事务集合的执行无法完整地缓存至 GPU 显存空间, 因此 Starfish 将 ChildTable1 从显存中传输到主机上存放, 并且将 input3 加载至显存, 在原有 ChildTable1 的缓存空间构建 ChildTable3 显存页; 以此操作循环, 直至将所有的插入事务集合处理完成, 并且所有 ChildTable数组都传输到 CPU 主存端备份. 由于对多个互相独立的 ChildTable 操作独立化, 因此

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Starfish 在构建插入事务哈希表空间时实现了高性能的并行加载和备份机制. 进一步, 基于 CUDA Streaming 对象策略, Starfish CUDA 通过设置若干个 Stream 来异步化 ChildTable内存页的 CPU-GPU 的 I/O 操作和 GPU 核函数的并行执行, 通过对不同 ChildTable 构建任务异步化, 从而能让不同 ChildTable 的 PCI-E 数据传输和 kernel 运行同时进行, 隐藏数据传输时间或者 kernel 运行时间.

算法 1. Insert 伪代码

```
(1)  FUNCTION(Keys, Vals)
(2)    myKey = Kyes[myId]; myVal = Vals[myId];
(3)    myKV = makeKV(myKey, myVal);
(4)    myLocation = hashFunction(constants[0], myKey);
(5)    FOR I in 1 to maxAttempts do
(6)      KV = atomicExch(table[location], myKV);
(7)      IF KV is EMPTY or DELETED THEN break;
(8)      END IF
(9)      determine_next_location(KV);
(10)   END FOR
(11)   IF KV is not EMPTY THEN
(12)     IF TryToInsertIntoStash(KV) is successed THEN
(13)       RETURN true;
(14)   END IF
(15)   RETURN false;
(16)  END IF
(17)  RETURN true;
(18) END FUNCTION
```

4.2.2 基于顺序粒度组合的高效索引 (search) 操作

如图 3 中所示, 在索引 (search) 过程中, 考虑到各个索引查询 (query) 的粒度大小对于整个表的缓存空闲相对较小, 因此将所有查询集合 (queries) 操作放入显存空间后再进行并行事务操作更为适合. 以图 3 为例, 具体的查找过程如下: (1) Starfish 构建统一化 query 查询数组对各个查询事务进行组合封装形成查询集, 归并后加载至 GPU 显存后, 执行索引 GPU 核函数在 GPU 哈希缓存页中进行并行检索. ChildTable1 和 ChildTable2 内存页为阶段性缓存于 GPU 内存页中, GPU 将执行检索 (search) 核操作对所有输入的查询事务集合并行检索. 然而, 在检索过程中, 由于存在哈希索引冲突, 为了进一步降低被重新索引开销, 我们将成功哈希检索到的键值 key 在查询集合 (queries) 中置空. (4) 在对 ChildTable1 和 ChildTable2 的操结束作之后, Starfish 将 ChildTable1 内存页置换到主机内存上, 然后将 ChildTable3 放入显存, 再在 ChildTable3 上对查询集合中还未被成功查找到的 queries 进行查找.

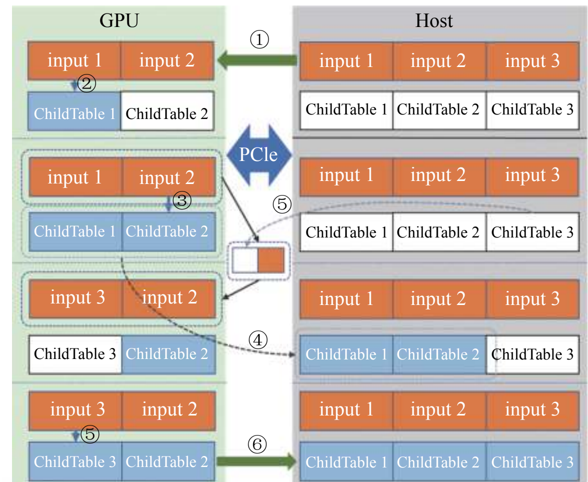

图 2 Starfish 哈希表事务 insert 操作

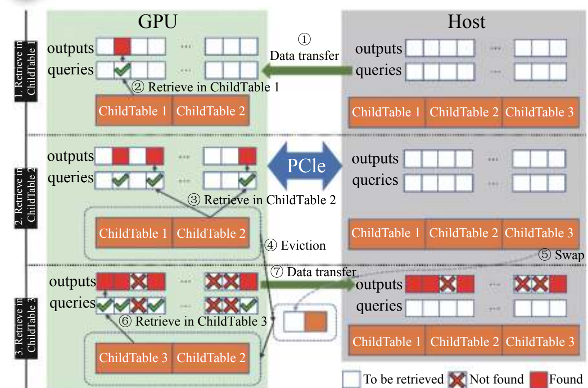

图 3 Starfish 哈希表事务 search 操作

4.3 基于高并发的 Starfish 哈希事务核函数优化

在 GPU 的 SIMD 编程模式下, GPU 中的若干个线程可以并行运行相同的代码, 考虑到 Starfish 已经具备了动态插入的功能, 因此在核函数的设计中沿用了传统的 cuckoo 策略来提高插入/查找的性能. 为了增强子表的容错性, Starfish 在每个子表末尾增加了 108 个<Key, Value>的数组 (称为 stash 哈希表) 来容纳无法插入当前子表的键值对. 考虑到 stash 长度较短, 出于对性能的考虑, 在 stash 数组中只使用唯一的哈希映射, 且忽略了溢出处理.

插入过程的核函数的伪代码如算法 1 所示, 在第 2–4 行中每个线程从输入的<Key, Value>数组中根据线程自身的 ID 号取出对应的键值对, 第 5–10 行进行多次 cuckoo 尝试, 在第 9 行为被挤出的<Key, Value>寻找合适的下一个位置, 在第 11–16 行如果最后被挤出的<Key, Value>不为空则说明超过了最大探测次数 (maxAttempts), 需要将其再放入 stash 数组中.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

算法 2. Search 伪代码

```
(1)  FUNCTION(Queries, Outputs)
(2)    myKey = Queries[myId];
(3)    IF myKey is EMPTY THEN RETURN;
(4)    END IF
(5)    FOR j in 0 to numFunc do
(6)      location = hashFunction(constants[i], myKey);
(7)      KV = table[location];
(8)      IF getKey[KV] equals myKey THEN
(9)        Outputs[myId] = getVal[KV];
(10)       Queries[myId] = EMPTY; RETURN;
(11)     END IF
(12)   END FOR
(13)   Location = SearchInStash(myKey);
(14)   IF location is valid THEN
(15)     Outputs[myId] = stash[location];
(16)     Queries[myId] = EMPTY;
(17)     RETURN;
(18)   END IF
(19)   RETURN;
(20) END FUNCTION
```

查找过程核函数的伪代码如算法 2 所示, 第 2 行从输入请求查找数组中取出键, 第 5–12 行按照候选哈希函数的顺序对键进行单词或多次探测, 若成功命中则将请求查找数组中当前键的位置置空, 以避免接下来在其他子表中对其重复查找.

5 实验和结果分析

5.1 实验环境及数据

实验运行环境为 Ubuntu 16.09, GPU 为 NVIDIA P100 12 GB, 代码为 C++, 用于测试的数据为随机生成的不重复 key 的键值对序列. 本节会将 Starfish 与经典的静态哈希—cudpp-Hash 以及经典的动态哈希—SlabHash 进行对比, 将 Starfish 中每个 ChildTable 的容量设为最高 100 万个键值对.

5.2 哈希表构建性能评估

正常情况下, 哈希表构建或者插入的时延应该将输入键值对传入显存的时间计算在内, 因此对于 Starfish 的构建性能评估将会包含数据传输的时间.

由图 4 可以看出, 在数据量超过 500 万个键值对时 Starfish-x (这里的 x 代表显存中开辟的空间可容纳 x 个 ChildTable) 的性能得到了显著提升, 达到了 cudpp-Hash 的 2 倍. 同时可以发现 ChildTable 数据量 (x) 逐步增大时, Starfish 的性能表现更为优越, 这是因为 Child Table 数据量越大时, 通过 PCI-E 进行缓存空间 swap 的频率越低, 而 PCI-E 传输相对事务性并行操作开销较大.

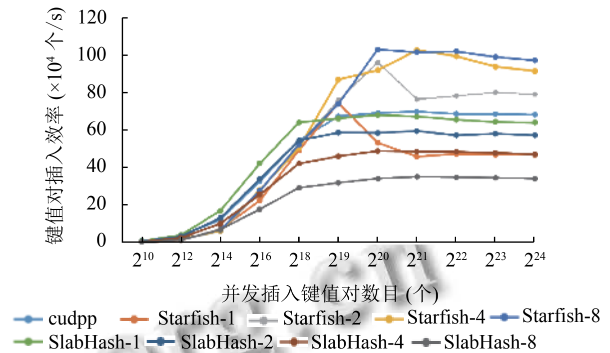

图 4　构建性能对比

缓存空间 (swap) 的频率越低, 而 PCI-E 传输相对事务性并行操作开销较大. 另外, SlabHash-n (n 代表平均链表长度) 在平均链表长度越大时要进行的平均显存次数也会更大, 所以性能也越低.

静态哈希表在构建过程中还可能发生插入失败后重新构建的情况, 因此只需要与 cudpp-Hash 进行比较. 由图 5 可以看出, 在数据量超过 100 万时, Starfish 的重新构建性能均超过了 cudpp-Hash, 甚至能达到 cudpp-Hash 的两倍, 这是因为 cudpp-Hash 在插入失败时需要重新构建整个表, 而 Starfish 只需要构建当前的 ChildTable.

5.3 哈希表查找性能评估

进一步, 本小节对 Starfish 的并发查找 (search) 操作的性能进行了评估. 对比系统包括经典的 GPU 下 hash table 静态哈希表 cudpp-Hash 和动态哈希表 SlabHash (bucket 长度设置为 2), 用以评估 Starfish 的查找性能和并发事务处理性能.

如图 6 所示, 本文对这两类哈希表和 Starfish 分别进行了并发查找性能测试, 每种表均由 800 万个不重复的键值对进行构建. 在显存驻留空间设置为 2 个 ChildTable 配置的情况下, Starfish-2 在并发查找请求平均分布于第一个子表和前两个子表时, 查找速度显著优于 cudpp-Hash 和 SlabHash, 达到 cudpp-Hash 的 4.7 倍和 2.6 倍. 这是由于 Starfish 在并发量满足 GPU 计算局部性的情况下, 只需要对驻留于显存中的子表进行查找, 且在驻留子表中查找所用的开销会远低于在整张静态哈希表 (cudpp-Hash) 中查找的开销. 同时, 在并发查找请求局部性较差时 (图 6 中并发事务

© 中国科学院软件研究所　http://www.c-s-a.org.cn

达到 200 万以上), Starfish 需要在非驻留子表中进行查找, 因此需要通过 PCI-E 总线交换子表数据, 在访存和 CPU 主存间的数据通信过程带来了一定的开销, 因此 Starfish 的并发事务查找速度略低于 cudpp-Hash 查找性能一半. 进一步实验验证, Starfish-4 与 Starfish-2 在查找性能上表现相似特征. 考虑到真实业务场景中在对哈希表查找事务的并发性要求着重于提供分布式配置优化, 本文拟将本部分大规模并发性查找事务的性能优化留于未来工作.

5.4 低显存性能评估

对比现有哈希表的 GPU 并行化工作, Starfish 的性能优势之一来源于 GPU 显存的节约效果显著以及显存资源的充分利用. 因此, 本文将 Starfish 与 cudpp-Hash 以及 SlabHash 进行了显存占用量的对比, 评估中除去了 CUDA context 的空间, 只关注于哈希表本身.

如图 7 所示, 在同等输入规模下, SlabHash 的显存使用量明显高于 Starfish 与 cudpp-Hash, 因为 SlabHash 以预分配 GPU 显存空间机制, 在进行哈希事务操作之前提前占用了所有的 GPU 显存实现哈希动态化插入策略, 带来了大量的显存资源开销. 而 Starfish 所占用的显存空间比 cudpp-Hash 要明显降低, 并且 Starfish 在设置固定的驻留表数目之后的显存占用量不会随着输入规模的增加而增加, 这也显著地降低了 cudpp-Hash 以及 SlabHash 所带来的 GPU 显存开销. Starfish 不仅能够支持高性能的 GPU 哈希表操作, 而且提供了相对更为节约的 GPU 显存空间利用, 为大规模哈希表操作在 GPU 上的并行高性能应用提供了支撑.

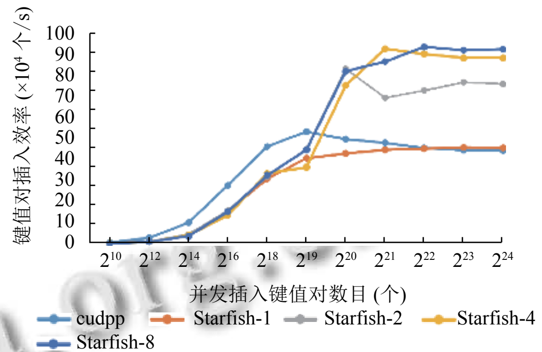

图 5 重新构建性能对比

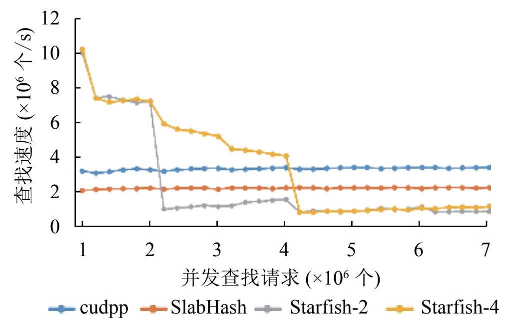

图 6 并发查找性能对比

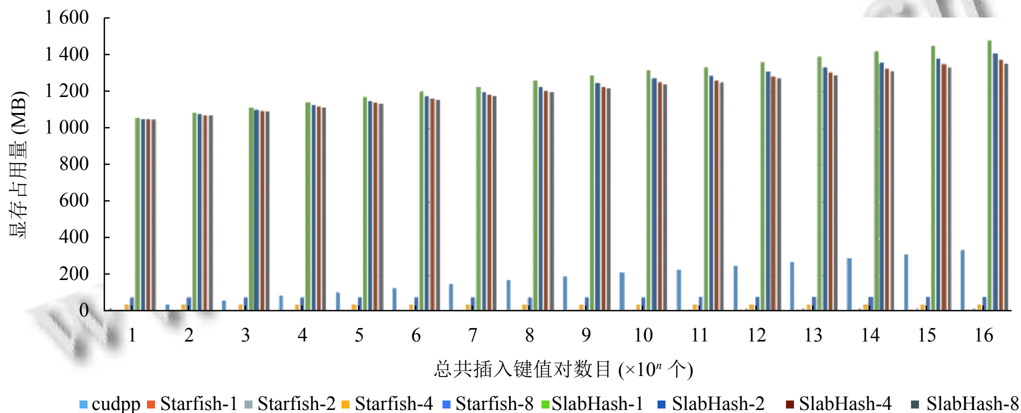

图 7 显存使用量对比

6 结语

针对目前 GPU 静态哈希表面临的可扩展性差以及 failure 处理代价高昂的问题, 提出了子表技术来应对. 对于动态哈希表面临的性能不佳问题, 提出了在子表技术的基础上, 将子表布局为静态哈希表来提高局部性能的思路. 进而, 为了应对当前 GPU 显存资源不足的情况, 将子表技术结合页面交换机制, 在保证性能的前提下极大降低了哈希表所占用的显存空间, 并且在保证并发查找局部性的前提下显著提高了查找性能. 此外, 具有高可扩展性、高性能以及低显存占用的 Starfish 哈希表可用于大型数据库、高性能计算中超大模型的局部计算等大数据加速任务当中.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

参考文献

1 Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal. ACM SIGPLAN Notices, 2012, 47(8): 117–128. [doi: 10.1145/2370036.2145832]

2 Zhang YP, Du B, Zhang LP, *et al*. Parallel DNN inference framework leveraging a compact RISC-V ISA-based multi-core system. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Online: ACM, 2020. 627–635.

3 Boncz PA, Manegold S, Kersten ML. Database architecture optimized for the new bottleneck: Memory access. Proceedings of the 25th International Conference on Very Large Data Bases. Edinburgh: Morgan Kaufmann Publishers Inc., 1999. 54–65.

4 DeWitt DJ, Katz RH, Olken F, *et al*. Implementation techniques for main memory database systems. Proceedings of 1984 ACM SIGMOD International Conference on Management of Data. Boston: ACM, 1984. 1–8.

5 Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. Proceedings of the IEEE 27th International Conference on Data Engineering. Hannover: IEEE, 2011. 195–206.

6 Fan B, Andersen DG, Kaminsky M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. Boston: USENIX Association, 2013. 371–384.

7 Lefebvre S, Hoppe H. Perfect spatial hashing. ACM Transactions on Graphics, 2006, 25(3): 579–588. [doi: 10.1145/1141911.1141926]

8 Polychroniou O, Raghavan A, Ross KA. Rethinking SIMD vectorization for in-memory databases. Proceedings of 2015 ACM SIGMOD International Conference on Management of Data. Melbourne: ACM, 2015. 1493–1508.

9 Ross KA. Efficient hash probes on modern processors. Proceedings of the IEEE 23rd International Conference on Data Engineering. Istanbul: IEEE, 2007. 1297–1301.

10 Lang H, Leis V, Albutiu MC, *et al*. Massively parallel NUMA-aware hash joins. Proceedings of the 1st and 2nd International Workshops in Memory Data Management and Analysis. Hongzhou: Springer, 2015. 3–14.

11 Zhang K, Wang KB, Yuan Y, *et al*. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. Proceedings of the VLDB Endowment, 2015, 8(11): 1226–1237. [doi: 10.14778/2809974.2809984]

12 Alcantara DAF. Efficient hash tables on the GPU [Ph.D. Thesis]. Davis: University of California, Davis, 2011.

13 Alcantara DA, Sharf A, Abbasinejad F, *et al*. Real-time parallel hashing on the GPU. ACM Transactions on Graphics, 2009, 28(5): 1–9.

14 Merrill D, Grimshaw A. Revisiting sorting for GPGPU stream architectures. Charlottesville: University of Virginia, 2010.

15 Pagh R, Rodler FF. Cuckoo hashing. Proceedings of the 9th Annual European Symposium on Algorithms. Denmark: Springer, 2001. 121–133.

16 Alcantara DA, Volkov V, Sengupta S, *et al*. Building an efficient hash table on the GPU. GPU Computing Gems Jade Edition. San Francisco: Elsevier, 2012. 39–53.

17 Ashkiani S, Farach-Colton M, Owens JD. A dynamic hash table for the GPU. Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Vancouver: IEEE, 2018. 419–429. [doi: 10.1109/IPDPS.2018.00052]

18 Harris M, Owens JD, Sengupta S, *et al*. CUDA data parallel primitives library. http://cudpp.github.io. (2006-11-30).

19 RAPIDS Development Team. RAPIDS: Collection of libraries for end to end GPU data science. https://rapids.ai. [2022-01-09].

20 Khorasani F, Belviranli ME, Gupta R, *et al*. Stadium hashing: Scalable and flexible hashing on GPUs. Proceedings of 2015 International Conference on Parallel Architecture and Compilation. San Francisco: IEEE, 2015. 63–74.

21 Breslow AD, Zhang DP, Greathouse JL, *et al*. Horton tables: Fast hash tables for in-memory data-intensive computing. Proceedings of 2016 USENIX Annual Technical Conference. Denver: USENIX Association, 2016. 281–294.

22 Celis P, Larson PA, Munro JI. Robin hood hashing. Proceedings of the 26th Annual Symposium on Foundations of Computer Science (SFCS 1985). Portland: IEEE, 1985. 281–288. [doi: 10.1109/SFCS.1985.48]

23 García I, Lefebvre S, Hornus S, *et al*. Coherent parallel hashing. Proceedings of 2011 SIGGRAPH Asia Conference. Hong Kong: ACM, 2011. 161.

24 Green O. HashGraph—Scalable hash tables using a sparse graph data structure. ACM Transactions on Parallel Computing, 2021, 8(2): 11. [doi: 10.1145/3460872]

25 Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. Proceedings of the 22nd International Symposium on Distributed Computing. Arcachon: Springer, 2008. 350–364.

26 Pagh R, Wei ZW, Yi K, *et al*. Cache-oblivious hashing. Algorithmica, 2014, 69(4): 864–883. [doi: 10.1007/s00453-013-9763-6]

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn