高技术通讯 2023 年 第33 卷 第11 期:1123-1135

doi:10.3772/j.issn.1002-0470.2023.11.001

基于 FPGA 的软硬件协同的多表哈希连接加速器①

吴婧雅② 卢文岩 鄢贵海③ 李晓维

(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)

摘 要 多表连接操作难以实现硬件加速。一方面,多表连接请求中表的数目不确定且连接方式多变,这种灵活的计算请求与固定的硬件行为之间存在矛盾;另一方面,多表连接的中间结果随表的增加而扩充,数据结构的管理和维护也要求更高的硬件开销。为支持灵活高效的多表连接计算,本文提出一种软硬件协同的优化方法。软件部分,将多表连接抽象为正向和反向 2 种计算模式并支持不同方式的多表连接。硬件设计采用访存和计算协同优化的方法:设计一种规则的硬件哈希表结构以提高内存访存带宽;设计支持正反向计算的同构专用计算引擎,配置多数据通道和指令控制系统实现高效的并行运算,提升多表哈希连接的计算效率。实验结果表明,相比中央处理器(CPU)执行表连接操作,单计算引擎能够提升性能 9.2 ~ 11.0 倍。通过多路并行的技术,实现 8 路并行的多表哈希引擎,能够充分利用板卡片外(DDR)内存带宽,实现相比 CPU 超过 71.1 倍的性能提升。

关键词 现场可编程门阵列(FPGA); 多表连接; 哈希连接; 软硬件协同

0 引 言

在大数据时代,数据量呈现爆发式增长,数据库管理系统(database management system, DBMS)发挥着越来越重要的作用。海量数据和复杂应用请求对 DBMS 提出更高的性能需求。数据库的表连接操作是 DBMS 中最常用的一类数据操作类型,由于其连接条件多变且复杂,越来越成为限制数据库系统性能的一类操作[1]。

表连接的实现方式主要有 3 种:嵌套循环连接、排序合并连接和哈希连接(Hash join)。其中,排序合并连接和哈希连接具有低复杂度和高计算效率的优势,在优化执行时被广泛采用。特别是哈希连接,由于在大数据量下优势更加明显而被设定为大部分 DBMS 可支持的缺省优化方案[2]。多表连接是表连接操作中最复杂的,是通过筛选多个数据表中的某一个或几个具有相同值的属性,直到找到满足筛选条件的多表内记录的过程。多表连接在实际应用中请求频繁。在数据库基准测试程序 TPC-H 中,多表查询请求数目超过全部基准程序的一半[3]。多表连接另一个典型案例是分析社交网络中的复杂的相互“好友”关系[4]。

为提高哈希连接的计算效率,相关学者提出利用异构平台,即现场可编程门阵列(field programmable gate array, FPGA)设计支持多路并行的哈希连接专用计算引擎[5]。但是这些方法都很难支持多表哈希连接的实现,主要原因包括以下几方面。

(1)查询请求多样与固定硬件设计之间的矛盾。多表连接的表项数目不确定,且多表连接的连接方式多样。例如,多表单一连接条件、多表多连接条件、链式多表连接、星形多表连接等[5]。灵活的多表连接请求要求硬件计算设计需要具有一定的灵活性,这与硬件结构的固定计算功能之间存在较大

① 国家自然科学基金(61572470, 62002340, 61872336)和中国科学院大学青促会基金项目(Y404441000)资助。
② 女,1994 年生,博士生;研究方向:计算机系统结构,异构计算系统优化;E-mail: wujingya@ ict. ac. cn。
③ 通信作者,E-mail: yan@ ict. ac. cn。
(收稿日期:2022-03-04)

的矛盾。

(2)硬件维护中间结果困难。多表连接时需要维护每次连接的中间结果,由于连接请求多样,导致中间结果的数据规模无法预判,给硬件维护和存储中间结果带来难度。同时,多表连接中当出现一对多或多对一查询结果时,需要更灵活的数据存储和管理机制,使硬件设计增加了难度。

(3)计算并发困难。多表连接需要按照表连接的顺序依次进行,否则连接判定条件的过滤会变得混乱,从而导致多表连接难以实现计算并发。虽然利用树形结构优化执行顺序能够提高计算并行性[6],但没有解决频繁的数据访存冲突对并发性能的限制。

这些原因导致支持两表连接的专用引擎无法有效地支持多表连接。

为实现多表哈希连接的计算加速,本文提出一种软硬件协同的设计方法,该方法可充分利用系统的计算资源和存储资源,实现提升多表哈希连接计算性能的目标。软件实现静态的两表关系数据的维护以及多表连接灵活的调度;硬件基于 FPGA 设计支持多表连接操作的专用计算引擎。本文的主要贡献有以下 4 个方面。

(1)设计了一种软硬件协同的多表哈希连接加速系统,提升了多表哈希连接的计算性能。软件实现高效的数据管理和指令调度,硬件设计专用的计算加速引擎。

(2)研究多表连接的计算过程,定义一种静态维护两表关系的数据结构,并将多表连接的计算模式抽象为正向和反向过程的组合,定义规则的多表连接计算模式。

(3)基于 FPGA 设计多表哈希连接专用计算引擎和数据结构,提高访存效率和计算性能。

(4)微结构设计支持指令控制和多数据通路,支持高并发的多表哈希连接操作。

实验平台基于 Intel Xeon 服务器中央处理器(central processing unit, CPU)平台和 Altera Arria10 FPGA 板卡搭建的计算平台,实现了本文所提出的哈希连接的加速系统。实验结果表明,单个多表哈希连接计算引擎相比 CPU 能够获得超出 9.2 ~ 11.0 倍的性能提升,8 路并行的多引擎结构设计相比 CPU 能够获得超出 71.1 倍的性能提升。

1　数据库的多表哈希连接

多表连接是一种复杂的表连接请求。多表连接可分为链式表连接和星形表连接 2 大类[4],图 1 给出一个多表连接方式的示意。这 2 种连接方可以通过不同组合形成灵活复杂的多表连接请求。

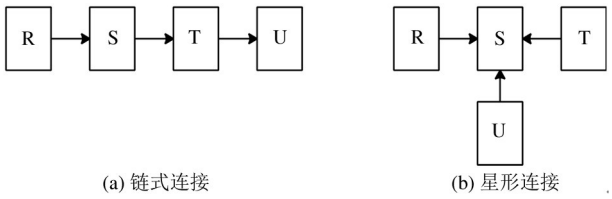

图 1　多表连接分类

多表连接的执行过程:按照软件给定的执行顺序依次完成两表连接。每执行一次两表连接,在存储的中间结果(第 1 次执行时,为两表中一个表的过滤结果)中加入连接条件过滤后的新表结果,直到对所有表的条件过滤结束。以数据库基准测试程序 TPC-H 中标准查询语句 Q9 中的多表连接查询请求为例[3],分析设计硬件多表连接加速器存在的困难,图 2 给出执行过程示意图。

执行多表连接时,由于每次加入过滤的表的主键和外键条目不确定,导致用于暂存中间结果的表格结构的数据存储容量和数据格式不固定。中间结果的容量和结构的灵活性还将随表数量的提升而变得更加复杂。例如在图 2 给出的示意图中,按照指令顺序执行 4 次执行多表连接操作时,前 2 次的中间结果的列数依次从 6 列增长到 8 列,第 3 次则保持不变。如果改变数据表的连接顺序,那么中间结果的存储结构和大小也相应要改变。

在多表连接操作中,内存的随机访问是不可避免的,且几乎时时发生。由于多表连接中每次执行的连接过滤条件是指令确定的,导致无法确定中间结果的列的访存,从而引起随机内存访问,进一步降低了计算效率和访存带宽性能。对于常用的动态随机存取存储器(dynamic random access memory,DRAM)

内存,其连续读写带宽基本可以达到标称值,而随机读写性能损失严重。例如 DDR4-3200 内存,其标称内存带宽为 25.6 GB/s。然而在实际使用中,随机读写带宽实际只能达到标称性能的不足 60%,甚至更低。图中给出的多表连接的示意过程中,4 次表连接的过滤条件不同,访问的中间结果和新加入数据表的索引键值的位置也不相同,从而引起随机内存读写,造成访存性能的下降。

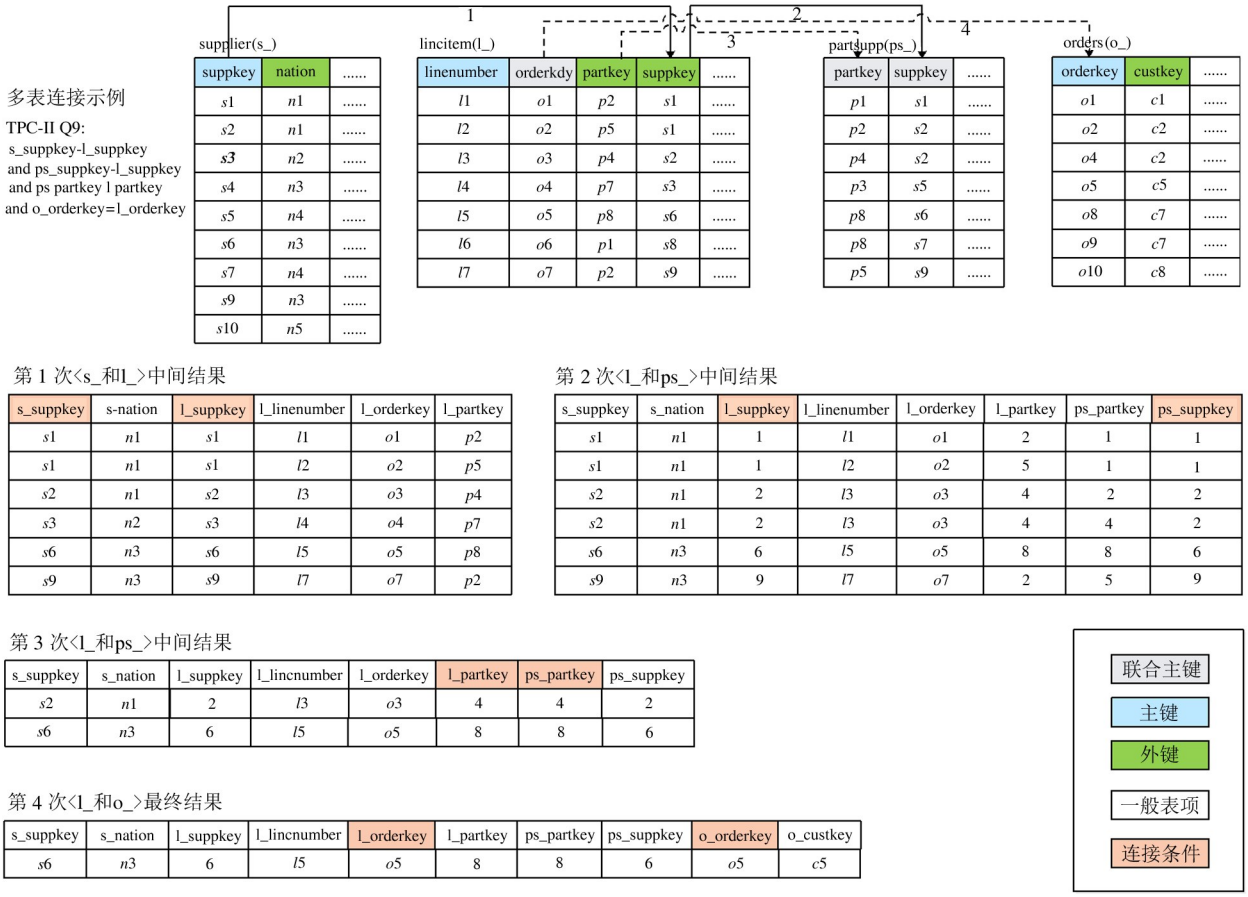

图 2　三表连接实例(TPC-H Q9)

设计专用的硬件加速器,一种最高效的实现方法是,将复杂的计算任务抽象为一种或几种固定的计算模式,并设计专用计算引擎,从而使得硬件加速器在灵活性和专用性之间取得一个合理的权衡。利用这样的方式设计两表连接操作的加速器是行之有效的[7]。然而,由于不同连接方式和过滤条件的组合,多表连接的操作具有复杂的计算模式。对于具有固定计算结构和功能的硬件计算引擎来说,实现灵活的多表计算操作是十分困难的。例如在图 2 中体现的三表对相同列属性过滤(s _ suppkey = l _ suppkey = ps _ suppkey)、三表对不同列属性过滤(s _ suppkey = l _ suppkey and l _ partkey = ps _ partkey)以及相应形成的链式多表连接和星形多表连接方式,其计算灵活性很高,使得计算引擎的控制单元和结构设计难度增加了。

2　相关工作

大数据量时代,数据库系统在数据的分析和管理等应用中面临更大的挑战,如何提升大数据的计算性能,成为数据库管理系统中急需解决的问题[1,8]。随着 CPU 性能提升速率不断放缓,越来越多研究开始转向利用具有高并行性的图形处理器(graphics processing unit, GPU)或 FPGA 等异构计算加速数据库系统[8]。利用 FPGA 加速数据库常用操作,能够根据不同计算的特征,定制设计专用的计

算单元,具有灵活和高效的特点。

为加速哈希连接过程,相关研究工作采用 FPGA 设计高效的哈希连接加速器。一方面,利用 FPGA 的片上硬件计算资源,能够提升哈希计算的并行性和鲁棒性[9]。另一方面,优化访存带宽和计算并行,从而提升哈希运算效率,能够显著提升表连接的计算性能[5,10]。基于加速两表连接的方法,将多表连接的过程转化为多个两表连接过程,并将中间结果作为算法另一阶段的输入,能够实现多表连接的一般操作[11-12]。但这种直观的实现方式的实际计算效率很低。

为加速多表连接过程,利用软件优化器实现多表连接的并行是一种常用的方法。主要技术手段是设计一种优化表连接顺序的机制以减小中间结果的数据量。利用树形结构抽象多表连接顺序以加速多表连接的执行,是一种常用的方法[4,13]。在此基础上,文献[9,14]设计了稠密树和数据预筛选的方法以减少连接产生的中间结果数据量。为优化多表连接的执行顺序,文献[6]利用启发式算法找到最小代价的两表连接顺序。但是,设计需要综合考虑数据中间结果体量、多表之间的对应关系等因素,实际优化效率并不高[4]。文献[15,16]针对 Apache Storm 平台中的多表连接计算,分别提出最低访存和通信开销规则、最低访存开销和最大计算并行规则的多表连接优化执行策略,能够从不同角度优化多表连接的执行顺序和并行度。进一步地,也有相关研究工作对借助新型存储器件执行计算的过程提出优化策略。文献[17]提出一种面向非易失性存储器的多表连接优化策略。文献[18]提出一种内存存储模型上的多表连接优化策略,上述方法主要目的是降低中间结果的访存开销及存储开销,从而提高多表连接的效率。这些优化方法和调度策略,都可以在本文所提出的软件优化器中实现。

为进一步提升多表连接的计算性能,文献[19]提出一种基于粗力度可重构架构的多路哈希连接加速器设计方法,将表连接中最大的中间结果进行划分,并使其以流的方式输入计算引擎;同时将小的中间结果存在片上缓冲中,支持计算引擎直接访问,以解决访存带宽限制和缓存空间限制。但这种方式当 FPGA 的片上缓存资源不足以支持小表要求的缓存空间时将会失效。文献[20]提出一种基于 FPGA 实现的支持多路并行的哈希链接加速系统。该方案设计了支持多对一的多数据通道的哈希建表和探查并行,且支持对单表多列或多表多列的哈希建表,从而使多表连接只经历一次连接操作即可完成,这种方法也成为大多数加速器设计所采用的通用实现方式。但这种一般的方法仍然没有解决计算中间结果的存储和管理。一旦连接表数目增加,或连接条件变成多关键字匹配连接,这种方式仍然无法避免存储中间计算结果时复杂的内存访问开销。基于 CPU 灵活的内存管理机制,利用 HashTeam[21] 或 SHARP[22] 方法,能够使结构化查询语言(structure query language,SQL)执行时有效支持多路并行的多表哈希连接。但由于灵活的访存和计算方式与固定的加速器架构设计之间存在冲突,这种方式无法适用于基于 FPGA 实现的专用加速器的设备。

3 软硬件协同的多表连接优化方法

针对第 1 节中分析的利用硬件加速多表哈希连接所面临的问题,本节提出针对性的解决方案。该方案主要针对联机分析处理(online analytic processing, OLAP)类型事务优化,并假设数据库支持索引。通常,在 OLAP 类型的应用中,数据库存储的数据记录稳定,几乎没有增删和修改[4,8]。

3.1 软硬协同的多表连接优化原则

在 OLAP 应用场景中,可以通过预处理得到数据表中数据记录的等值关系,从而提高表连接的计算效率。这一过程利用 FPGA 设计专用的哈希计算引擎[3],对数据表进行预处理,得到各数据表之间常用字段(通常以索引的形式存储)的等值关系。本文提出一种专用硬件哈希表来维护两表关键字(或索引)的对应等值关系。并规定,硬件哈希表存储在板卡的内存中,硬件哈希表的预处理和存储的管理控制利用软件实现。具体实现将在 4.1 节中说明。

基于预处理得到的等值关系信息(即硬件哈希表),实现硬件加速多表哈希连接的计算任务可以

被抽象为:如何快速筛选满足条件的等值记录并支持多表连接。利用硬件实现特定计算时最具有挑战性的问题是:如何解决计算灵活性与硬件专用性之间的冲突。为解决这一矛盾,本文提出将多表连接操作过程抽象为规则化的正向连接过程和反向连接过程,简称为正向过程和反向过程。从而,执行多表连接可按照图 3 所示的步骤。其中,正向过程是按照给定的查询条件对现有的硬件哈希表或中间结果执行条件过滤;反向过程是加入筛选后的新表结果反向筛选原中间结果,以得到表连接的新中间结果。最终多表连接的结果由软件组织合并并输出。具体的硬件结构的设计将在 4.2 节中说明。

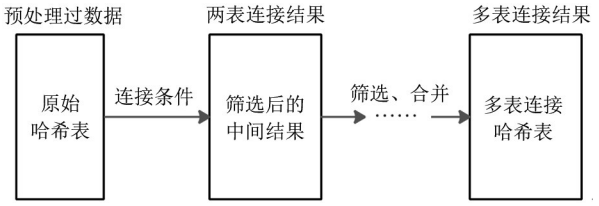

图 3　哈希表连接加速系统的执行过程

针对多表连接中间结果存储的不确定性,本文提出一种规则的数据存储结构,其目的是使硬件存储控制单元和计算引擎的访存变得简单且高效。在正向过程中,中间结果只存储与两表连接相关的索引信息;在反向过程中,中间结果只存储与两表连接相关的索引信息及相对位置信息。具体的设计规则将在 4.3 节中说明。

根据上述设计原则,多表哈希连接加速系统的整体结构(如图 4 所示)包括软件和硬件两部分。其中,软件部分主要负责:(1)管理数据表间两表等值关系的维护过程;(2)多表连接的最终结果的整合;(3)对给定的多表连接操作按照正向和反向的划分方式生成相应的计算指令;(4)优化指令执行顺序及提高并行性。硬件部分通过设计专用的计算引擎和数据存储格式,实现从计算和访存两方面提升多表哈希专用加速器的性能。

3.2　软件系统优化

软件系统实现的主要功能包括:维护静态的两表等值关系,多表查询语句的调度和管理。

两表等值关系的维护,需要参考硬件结构的实现进行统一管理。根据 FPGA 板卡上缓存资源的存储空间结构,本文规定对每个数据表以 32 k 行为一段进行切分,则每个两表等值关系表记录 2 个 32 k 行的部分表的对应关系。图 5 给出大数据表划分规则的示意图,其中不足 32 k 行的部分也按照 32 k 行为单位进行统一管理。则对于图 5 所示的 S 表和 T 表,共需要维护 12 个静态等值关系表。由于这 12 个静态等值关系表之间是相互独立的,从而可以依靠软件的合理调度和管理,实现数据表连接的计算并行。

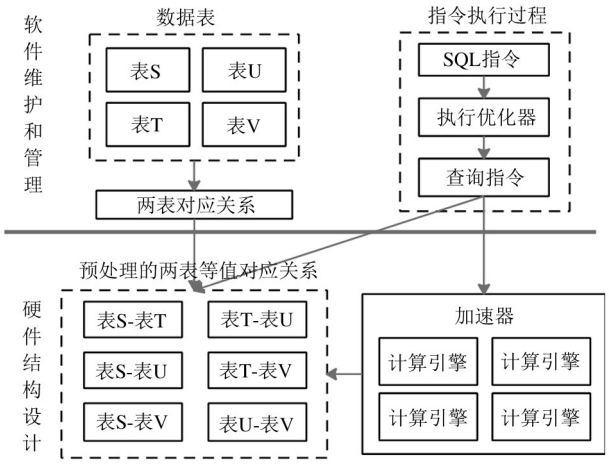

图 4　哈希表连接加速系统的整体结构

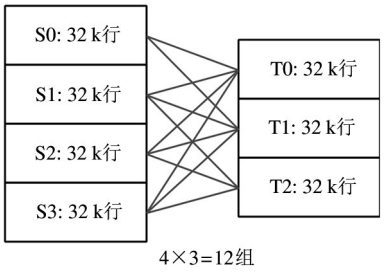

图 5　哈希表连接加速系统的整体结构

多表查询语句的管理和维护,按照正向和反向过程的划分被统一为 2 种操作的组合。例如,对于 n 表连接请求,每次只需要执行 $(n-1)$ 次正向过程和 $(n-2)$ 次反向过程。图 6 给出一个三表链式连接的过程示意图。三表连接的顺序由上层软件优化器决定,假设排序好的 3 个表分别为 S、T、U。第 1 步,计算引擎执行正向过程,即根据查询表 S-表 T 的哈希映射表中符合查询条件(相同索引)的记录,并生成中间结果表。第 2 步与第 1 步类似,找到表

T-表 U 中满足条件的记录,并生成中间结果表。第 3 步,执行正向连接得到表 T 和表 U 的中间结果。第 4 步,执行反相连接过程重新筛选表 S 和表 T 的中间结果。第 5 步,由软件合成最终结果并输出。中间结果表的定义和实现将在 4.2 节中介绍。

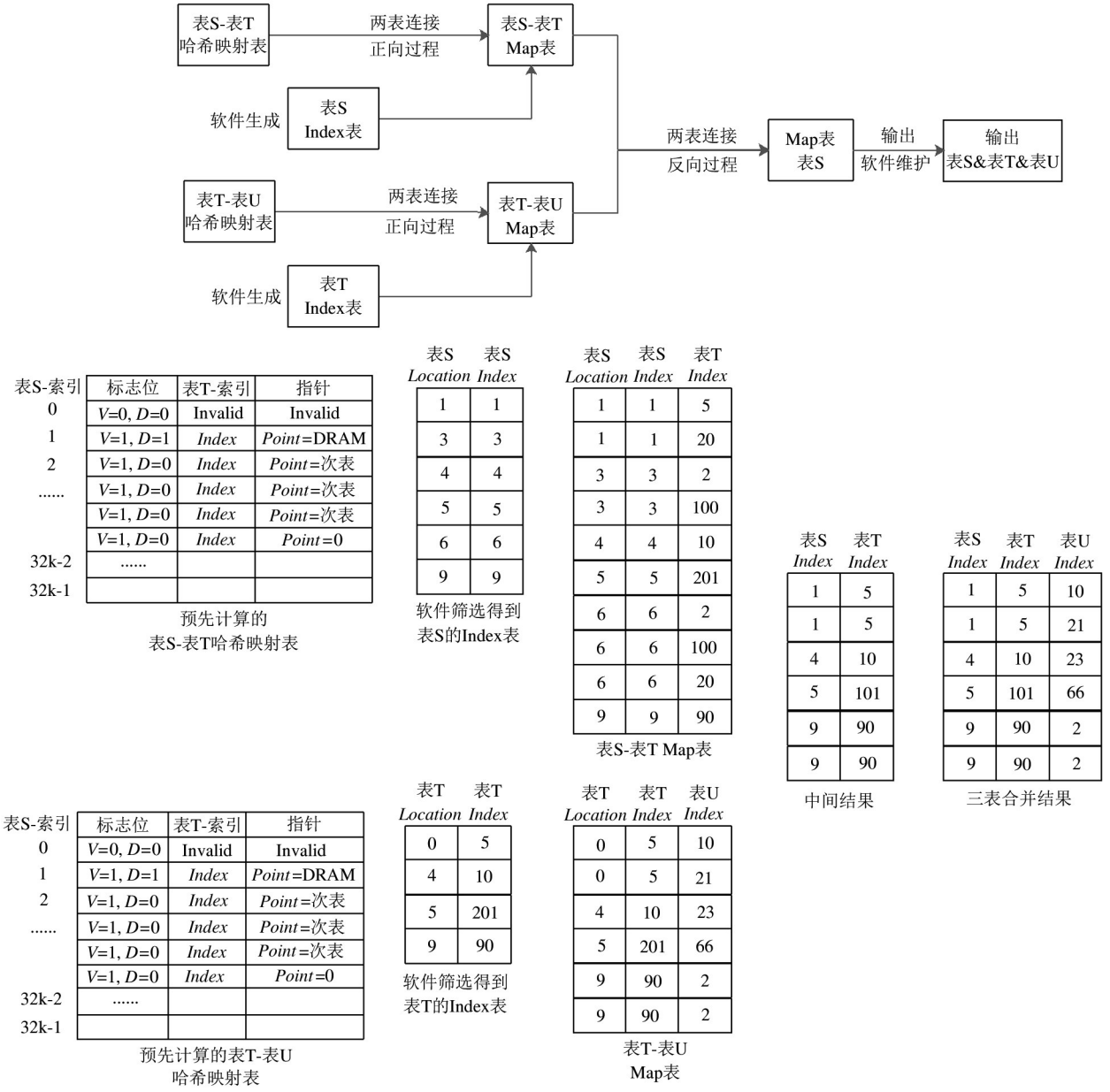

图 6　三表连接过程示意图

4　硬件结构优化设计

4.1　硬件哈希表设计

为存储静态的等值关系,本文设计了一种灵活高效的硬件哈希表数据结构。该数据结构能够灵活存储一对一、一对多和多对一的哈希映射关系,且具有规则的数据组织方式,能够降低数据解析的复杂度。数据结构的设计以最大化利用内存读写带宽为原则,从而降低专用计算引擎的数据读写延迟。例如,FPGA 片外存储 DDR4-3200 的最小突发长度为 64 bit,突发长度最大为 8,则一次突发读写大小最大为 512 bit。从而硬件哈希表的每行设置为 64 bit(即 8 Bytes)。硬件哈希表的组织按照以下的方式进行:

硬件哈希表的每一行对应当前表的关键字哈希值,并在该行中维护另一个数据表中具有相同哈希值的记录。在保持数据存储格式规则的基础上,设计一种可扩充的主次表结构,以便有效解决哈希冲突。

硬件哈希表具有以下主要特征。

(1)计算效率高:规则的数据结构可以支持对齐的内存读写,控制逻辑简单,数据解析方式固定,计算效率高。

(2)读写效率高:硬件哈希表的结构设计能够充分利用突发读写的内存访存带宽高的性能。

(3)有效解决哈希冲突:两级表能够有效处理哈希冲突,设计次表大小为主表的一半,在结构设计的硬件资源需求和数据哈希冲突的概率之间取得的权衡,该空间足够解决数据的哈希冲突[7]。

图 7 描述了两级硬件哈希表的结构。硬件哈希表的每行大小为 8 Bytes。硬件哈希表的存储管理以 FPGA 的页面大小为原则划分。假设页面大小为 512 kB,则每个原始数据表大小为 32 k 行,次表为 16 k 行,两级哈希表共有 48 k 行。硬件哈希表每行大小为 8 Bytes,包括 2 Bytes 数据标志位,4 Bytes 索引和 2 Bytes 地址空间。其中标志位存储 V 和 D 两类标记,V 标识当前行数据的有效性($V=1$ 时有效);D 表示指针数据的类型。$D=1$ 表示指针内容为 DRAM 地址,$D=0$ 且 *Point* 不为 0 表示指针内容指向次表中的地址;$D=0$ 且 *Point* 为 0 表示表 S 的当前索引在表 T 中没有其他的对应索引。

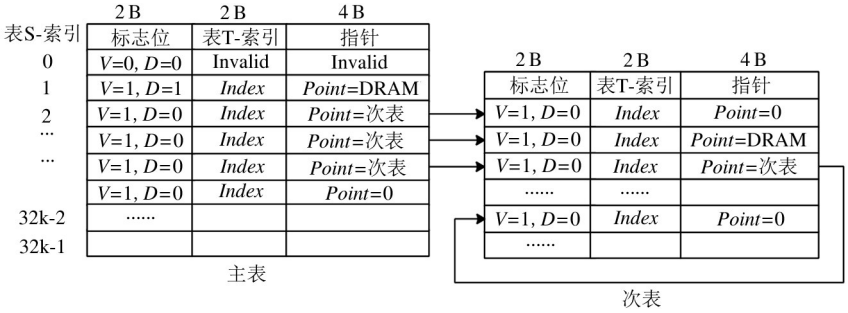

图 7　硬件哈希表结构

4.2　支持多表连接操作的计算引擎设计

根据第 3 节中对软硬件协同多表连接加速系统的设计,硬件加速引擎主要的计算任务是:(1)正向完成哈希表查找和条件过滤;(2)反向完成数据过滤。为降低计算的控制逻辑,提升并行效率,在加速器中设计 2 种哈希连接计算引擎,即正向计算引擎和反向计算引擎,并规定一个正向计算引擎和一个反向计算引擎构成一个标准计算单元。如图 8 所示,正向或者反向模式的选择仅依靠指令解析后的信息,通过指令单元的控制信号实现模式选择。虽然这种全功能的设计使得计算引擎内部的硬件资源开销比较大,但指令的控制逻辑简单,计算引擎并行的控制逻辑简单且高效。同时,在实际结构设计中,本方案还考虑了两类计算引擎的资源复用,以进一步降低同构引擎设计中的硬件资源开销。这一部分将在第 6 节加速器结构实现部分具体说明。

同时,为提升计算的并行效率,计算引擎还充分利用片上的高带宽、低延迟的 BRAM 作为缓存单元存储计算所需数据,利用其高读写带宽的特性进一步缓解访存瓶颈,从而提升计算效率。

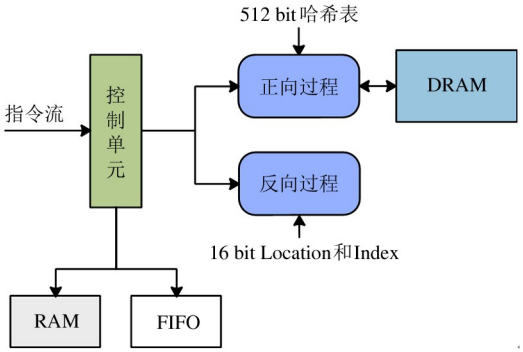

图 8　计算引擎结构

4.3　中间结果的数据结构设计

根据第 3 节提出的中间结果存储原则,中间结果可以只存储两表对应的索引值和相对位置,大大减少了中间数据的存储空间需求和数据解析的复杂

度。为实现配合计算单元实现高效的哈希连接，本文为计算引擎设计了专用的数据结构，如图 9 所示。其中：标志位（ = 1）用于标识输出的 *MAP* 值含有 *Point* 信息；控制信号和有效位共同作用，用于控制计算引擎执行计算；有效位（ = 1）标识该行数据有效。Index 表用于存储经过筛选后的原数据表索引及其在硬件哈希表中的地址信息。Map 表包含 Index 表的内容并增加一列用于存储另一个数据表的对应索引值。两表中的 *Location* 信息主要用于反向查询时确定前一个表的索引信息。

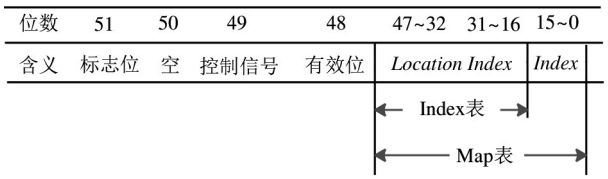

图 9　计算引擎支持的数据格式

5　基于 FPGA 的硬件多表连接计算引擎设计

5.1　多表连接计算引擎

哈希连接的正向过程和反向过程可以通过同构的硬件计算单元实现。其中，反向过程可以重用正向过程的部分结构实现，避免硬件资源的冗余设计。计算引擎的结构设计如图 10 所示。计算引擎中有 2 个共享的缓存单元：1 个 RAM 单元和 1 个 FIFO（first in first out）存储器。其中，RAM 的深度为 6 k，宽度为512 bit。在正向过程中，RAM划分为2个独立的逻辑空间，RAM1 的深度为 4 k 用来存硬件哈希表的主表，RAM2 的深度为 2 k 用来存硬件哈希表的次表。在反向过程中，RAM 直接存储 MAP 表中的 *Location* 和 *Index* 信息。RAM 的位宽设计为 512 bit，能够充分利用内存读写带宽实现高效的数据读写。FIFO 是固定的存储空间，其位宽为 16 bit，用于缓存 Index 表（正向过程）或 *Location* 信息（反向过程）。对于计算单元，反向过程和正向过程共享 MAP _ generate1 模块，根据控制信号的控制，分别在正向和反向过程完成不同计算任务。

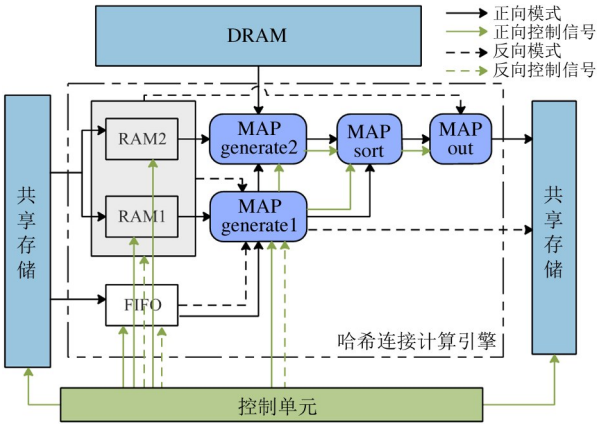

图 10　哈希连接计算引擎

哈希连接计算引擎的正向过程主要功能是通过硬件哈希表和 Index 表的筛选，输出两表连接的结果，即 Map 表。在实际实现时，将这一部分的功能拆分为几个部分：（1）实现硬件哈希表的两级查询的两级计算引擎 MAP _ generate1 和 MAP _ generate2；（2）结果合并的 MAP _ sort 模块；（3）结果输出的 MAP _ out 模块。

正向过程中，首先将硬件哈希表从共享存储中加载到 RAM（其中，RAM1 的深度为 4 k 用来存主表；RAM2 的深度为 2 k 用来存副表）；然后将 Index 表从共享存储中加载到 FIFO。直到硬件哈希表在 RAM 加载完成，MAP _ generate1 模块开始从 FIFO 中读取 *Index* 值，并根据值从 RAM1 中读取对应的 LUT 值。若 *Point* 有值，则将 *Point* 值输出到 MAP _ generate2 模块，MAP _ generate1 模块始终保持连续输出。MAP _ generate2 模块根据 *Point* 值从 RAM2 或 DRAM 中取出对应的 *Point* 值。MAP _ generate1 模块和 MAP _ generate2 模块将生成的 Map 表输出到 MAP _ sort 模块，MAP _ sort 模块判断 MAP _ generate1 和 MAP _ generate2 中的输出。其中，控制位信息确定 MAP _ generate2 模块是否输出有效数据。根据 *index* 值将这些 Map 表按照硬件哈希表中的顺序依次输出到 MAP _ out 模块。MAP _ out 模块会筛选掉无效的数据，并根据情况控制 MAP 表的控制信号，再根据哈希表中的地址信息，输出哈希连接正向过程的结果。

MAP _ generate1 模块为流水线设计，共分为 4 级，由 FIFO 信号控制。第 1 级，向 FIFO 发出读指令。第 2 级，根据 FIFO 的输出的 *Index* 值向 RAM1

发出读地址。第 3 级,根据 RAM1 的输出截取对应的位。第 4 级,输出 MAP 值,并根据第 3 级中截取的哈希表主表的 *Point* 值判断是否需要使能 MAP _ generate2 模块,若需要同时输出 *Point* 值。

MAP _ generate2 当且仅当存在次表地址时使能。其功能是根据 RAM2 输出的 *Point* 值依次读取哈希表的次表或 DRAM 地址,依次输出 MAP 表,并对有效的控制位信号赋值,使其标识 MAP _ generate2 模块的输出数据有效。

处于反向模式时,首先将 *Location* 和 *Index* 值从共享存储中加载到 RAM;然后将 *Location* 列从共享存储中加载到 FIFO。当 RAM 加载完成,MAP _ generate1 模块开始从 FIFO 中读出 *Location* 值,将 *Location* 的 3 ~ 15 位作为 RAM 的读地址输出,再根据 *Location* 的 0 ~ 2 位从读出数据中取出对应的数据整合输出。

反向过程的 MAP _ generate1 模块流水线结构与正向模式相同,只是在第 4 级直接输出处理结果,而不再向其他处理单元发出信号。

5.2 多计算引擎并行

为进一步提升系统的计算性能,本文还提出了灵活可配置的多同构的计算单元设计,以支持多指令并发的需求。同时,为提升计算的并行效率,设计一种多数据通道和数据预取机制,进一步缓解访存瓶颈,提升计算效率。图 11 给出了这种支持多路并行的计算单元和多数据通道设计的结构示意图。

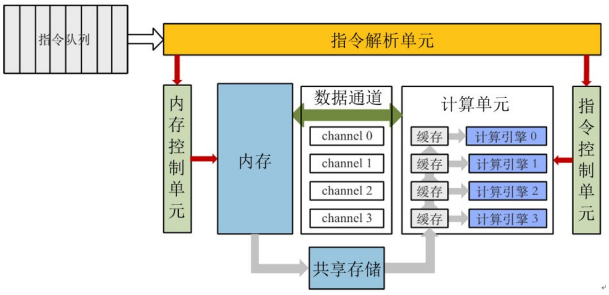

图 11 多路并行和多数据通道

计算和数据访存的控制是异步的,无论是否有空闲的计算引擎,计算所需的数据都可以从内存写入共享存储中,也就是可以实现数据的预取,利用计算掩盖数据读写延迟,进一步提高计算效率。当指令到达指令队列中时,指令解析单元解析每一条计算指令。根据指令中的数据访存地址,激活内存管理单元,将计算所需的数据从内存写入共享存储。内存管理单元对数据写入共享存储的顺序严格按照指令到来的顺序控制其执行。且异步机制能够支持即使没有空闲的计算引擎该过程也不会被阻塞。

计算的并行依靠指令控制单元的统一管理。指令控制单元根据指令中的计算指令激活空闲的计算引擎。若此时没有空闲计算引擎,则等待出现执行完上一个计算任务的空闲计算引擎,再启动下一次计算任务。由于所有的计算引擎均为同构设计,因此只需要根据计算引擎的状态确定空闲引擎,即可启动计算。具体执行正向还是反向过程,由计算引擎在启动后根据指令相应的控制信息及输入数据类型决定。

6 实验与结果

6.1 实验平台和测试基准

为测试专用的多表哈希连接计算引擎的实际性能,实验的硬件平台的搭建以服务器和 FPGA 板卡实现。其中,服务器的型号为 Intel Xeon CPU E5-2630(@2.20 GHz)配备 25 MB 的 L3 缓存和 64 GB 的内存(DDR4 2667 MHz)。FPGA 板卡选用搭载 Altera Arria 10 GX1150 的 DE5a-Net 板卡。两者互联采用 PCIe 3.0 x8 总线接口。FPGA 的板卡资源见表 1。

表 1 FPGA 硬件资源配置

资源类型		性能参数
存储	DDR3 1066 空间	8 GB
	DDR3 1066 带宽	17 GB/s
SRAM	QDRII + SRAM 空间	32 MB
	QDRII + SRAM 带宽	288 Gbps
嵌入式存储器	M20K(数目)	54 kb(2, 713)
	MLAB(数目)	13 kb(21k)
逻辑资源	寄存器	1, 709 k
	ALM*	427 k

* 每个 ALM 可根据应用需求拆分为 2 个 ALUT,但有时考虑综合布局布线的开销,会利用 2 个 ALM 实现 2 个 ALUT。

为评估数据库索引的查询和维序性能，实验的测试数据和程序选择数据库基准测试程序 TPC-H 提供的标准数据库表和决策支持基准测试程序[3]。TPC-H 包括 9 个数据表，测试数据记录了某一类仓库在不同地区的存货量、存储的不同商品库存以及地区消费者信息和货物订单等信息，测试程序为 22 条查询指令。本文选取其中比较有代表性的三表连接指令及两表连接指令作为实验的测试程序指令。实验数据为随机生成的 1SF（对应 1 GB）的数据表。

其中，多表连接分为链式连接和星形连接 2 类，包含 2 次正向操作和 1 次反向操作数次，两表连接可以看作一次正向操作，具体指令见表 2。

表 2　表连接指令

查询	连接类型	连接条件
Q7，Q8	多表连接（星形）	s _ suppkey = l _ suppkey, and o _ orderkey = l _ orderkey
Q3，Q5，Q10	多表连接（星形）	c _ custkey = o _ custkey and l _ orderkey = o _ orderkey
Q9	多表连接（链式）	s _ suppkey = l _ suppkey and l _ suppkey = ps _ suppkey
Q9	多表连接（链式）	ps _ partkey = l _ partkey and p _ partkey = l _ partkey
Q3，Q4，Q5	单表连接	l _ orderkey = o _ orderkey

6.2　多表哈希连接计算引擎的性能

本小节主要评价多表连接的性能。以 6.1 节中介绍的实验环境为基础实现多表哈希计算引擎，实验测试数据为表 2 中的多表查询语句。本节中，以单位时间完成两表正向或反向连接的表的记录数目作为衡量系统性能的指标（行/s），实验结果与 CPU 执行两表查询性能为比较基准。

实验的参照对比标准选取 CPU 的计算性能。在 CPU 执行一次计算时规定，对小表的哈希构建过程已经完成，一次计算只是执行大表的探查。得到一组匹配则记为完成一行匹配的执行时间。总的性能为平均单位时间内完成的匹配行数。CPU 的执行没有正向或反向的区分。

图 12 显示了系统从单计算引擎实现到 8 路并行计算引擎实现的性能对比。实验中，多表哈希连接引擎的实际工作频率为 235.67 MHz。实验结果表明，单计算引擎的正向计算性能大约为 CPU 基准性能的 9.3 倍，反向计算性能大约为 CPU 基准性能的 11.0 倍。随着计算引擎并行度的提升，8 路并行的多表哈希连接加速系统能实现 71.1 倍以上的性能提升。同时，整个系统最大的带宽性能也达到 15.2 GB/s，基本达到内存访存带宽的理论值（大约为 89%）。

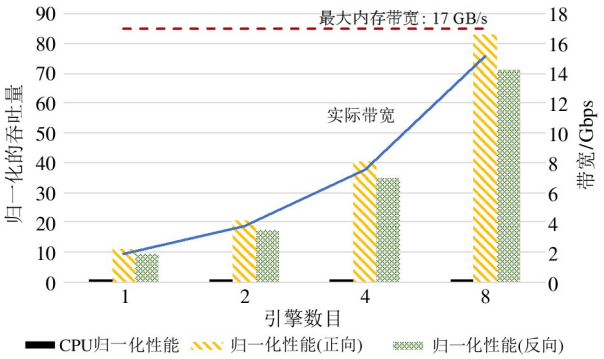

图 12　多表哈希连接计算引擎性能

8 路并行计算能够获得近似 8 倍的性能提升。这主要是由于在设计硬件加速引擎时，设置的内存突发读写长度为 8。8 路并行刚好可以直接平均到每一路计算中，相当于满带宽的并发读写。之所以会有一定程度的性能损耗，主要是因为当并行的引擎增加后，控制电路部分会引入一定的开销，但这种开销相比于性能的提升几乎可以忽略不计，理论上 8 倍的性能提升大约会带来 4.4% 的额外开销。

实际实验结果中，反向计算引擎的效率要略高于正向计算引擎，但多路并行计算引擎的性能提升倍率则低于正向计算引擎。反向计算引擎的效率略高，主要是由于反向过程引擎的功能单元更少，完成单条数据的指令周期略短于正向过程，单位时间内完成哈希连接的表的记录数目更多。多路并行的效率提升有限，则是由于正向过程的平均计算数据量要远高于反向过程（按照数据生成的统计规律不同而有所区别，大约相差 1 ~ 2 个数量级）。相对地，执行同等表记录的数目情况下，指令并行控制占用了更多周期，且数据管理单元延迟影响更大，所以导致多路并行时计算性能的平均提升略低于正向过程。

大多数基于 FPGA 实现的哈希连接加速器只分

析了两表连接的计算性能,所以本文不与其对比哈希计算过程的性能[4,9]。事实上,对哈希计算的加速工作研究已经比较成熟,本文的实验也采用与这些研究工作类似的实现方式来实现哈希计算。

为进一步分析本文所提出的方法的性能,将本实验结果与基于 GPU 实现的多表连接操作[11]的性能进行对比。在实现多表连接时,相比基于 CPU 实现的多表连接,利用 GPU 能够获得平均 4.33 ~ 4.81 倍的性能提升。这一结果远低于 FPGA 实现的专用加速器的性能提升。与 CPU 实现的多表连接计算相比,基于 FPGA 设计的专用加速器能够提升11.25 倍的计算性能[10]。但文献[10]所提出的方法在实际执行时,由于受到哈希冲突的影响,且实现方案的内存访存带宽利用效率低,会导致超过 87% 的计算阻塞,影响计算性能,同时造成系统额外的运行时开销。而本文所提出的方法充分利用了内存带宽的性能,从而使得多表连接性能提升达到大约 2 个数量级。

6.3 计算引擎的灵活性

本小节主要评价多表连接加速系统对多表连接计算灵活性的支持性能。以 6.1 节中介绍的实验环境为基础实现多表哈希计算引擎,实验测试数据为表 2 中的多表查询语句。同时,为测试多表连接引擎对复杂的多表连接请求的执行性能,实验中将 Q9 的 2 条链式三表连接合并为一个兼有多对一及星形和链式组合方式的四表连接操作(组合后的多表连接共包含 1 次两表多条件连接和 3 次多表连接请求,相当于进行 5 次表连接)。

实验过程中,重新生成测试数据。按照 TPC-H 基准测试程序中对数据表的格式规定,实验设计每次生成的数据表量均以 32 k 行为一个增长单位,则数据表量从 32 k 行依次增长至 128 k 行。生成的硬件哈希表对应“1H”,是指利用软件直接将 32 k 行的原始数据表预生成 1 个硬件哈希表。“2H”表示原始的数据表可分为 1 组和 2 组,从而生成 2 个硬件哈希表。“4H”表示原始的数据表可分为 2 组和 2 组,从而生成 4 个硬件哈希表。以此类推,直到共生成 16 个硬件哈希表。每次实验中,实际使用的硬件哈希表的个数与多表连接请求中的操作数目相同,即三表连接输入 2 个硬件哈希表,五表连接输入 4 个硬件哈希表。

本节统计在不同表的容量下,不同组合方式的三表连接及四表连接的平均完成时间,以此作为评价指标。实验的对比基准选择单引擎完成一次链式计算的平均时间。图 13 给出了实验结果。

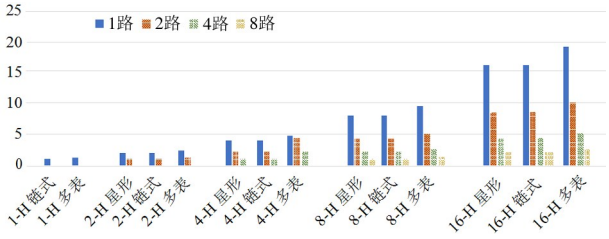

图 13　多表哈希连接归一化平均完成时间

实验结果表明,星形计算或链式计算的平均时间几乎没有显著的差别,这也体现了本文提出的多表哈希连接加速系统能够有效支持不同类型的多表连接请求。随着表连接数目的增加,计算响应时间增加,但增长几乎与表的数目增减成线性关系,也证明了本方案具有一定的灵活性和可扩展性。同时,随着表容量的增加,多路并行计算的性能也有显著的提升,表明软件能够有效控制多表连接的并发,硬件也能够按照并发控制执行计算任务。

6.4 计算引擎的资源消耗

本节评估多表哈希连接加速去系统的可扩展性。计算架构的可扩展性可以衡量一种计算架构设计在增加计算节点时,硬件资源需求的增加情况以及实际性能的提升倍率。在实验中,设计系统的计算引擎数目为单计算引擎及 2、4 和 8 路并行引擎。当实现 8 路计算引擎时,系统的带宽基本达到实际使用可实现的最大值。资源消耗情况见表 3。

表 3　资源消耗情况

引擎数目	ALM	ALUT	寄存器	M20K	Block memory
1	1679	1998	3250	131	2 467 488
2	3115	3380	5112	289	4 937 536
4	6589	6912	9172	547	9 878 144
8	12 250	13 569	5134	1156	19 748 096

随着计算并行度的提升,计算资源的消耗并没有成倍的增长,也就是说,多路并行是平均资源利用率最高效的实现方案。造成这一现象的主要原因是在电路综合中,会出现一些固定的自适应逻辑模块(adaptive logic module,ALM)资源使用。这一部分随并行度的提升几乎没有太大的变化,在单引擎的 ALM 资源消耗中占比超过 13.9%。虽然提高计算并行度增加了计算引擎的数目,但实际所需的硬件资源的开销并没有按照相应的倍率增加。同时,算术逻辑单元(arithmetic and logic unit, ALU)资源的消耗也没有成倍的增长,除了与 ALM 资源的增长保持基本的相关关系以外,在布局布线中,软件所采用的电路优化策略也会使得 ALU 资源的增长不完全呈现线性的关系。

7 结 论

本文提出一种软件硬件协同优化的多表哈希连接加速系统,以解决多表哈希连接中数据访存效率低、数据格式不规则、计算模式复杂、计算效率低等问题。该加速系统的核心思想是对多表哈希连接中计算模式和数据格式进行规则设计。硬件设计采用访存和计算协同优化的方法,将多表哈希连接过程抽象为正向和反向 2 种规则计算模式,并根据这种规则计算模式设计能够支持 2 种计算的同构硬件计算引擎。同时设计专用的硬件哈希表,提升计算引擎的访存带宽。基于这样的硬件设计,软件只需要维护固定的硬件哈希表信息,并无差别地实现控制正向和反向计算过程。基于同构的计算引擎的设计,降低了多指令并行的控制成本,有利于提升指令级多表哈希连接的并行计算效率。

实验基于 FPGA 板卡实现从单引擎到 8 路并行引擎的多表哈希连接加速系统。实验结果表明,单计算引擎能够相比 CPU 执行表连接操作将性能提升约 9.2 ~ 11.0 倍;而 8 路并行多表哈希引擎,能够充分利用板卡片外 DDR 内存带宽,并实现超过71.1 倍的性能提升。

参考文献

[1] HOOZEMANS J, PELTENBURG J, NONNEMACHER F, et al. FPGA acceleration for big data analytics: challenges and opportunities[J]. IEEE Circuits and Systems Magazine, 2020,21:30-47.

[2] BALKESEN C, ALONSO G, TEUBNER J, et al. Multi-core, main-memory joins: sort vs. Hash revisited[J]. Proceedings of the VLDB Endowment, 2013,7(1):85-96.

[3] Transaction Processing Performance Council(TPC). TPC-H benchmark[R/OL].[2022-02-28]. http://www.tpc.org/tpch.

[4] HENDERSON M, LAWRENCE R. Are multi-way joins actually useful? [C]//Proceedings of the 15th International Conference on Enterprise Information Systems. Angers: ICEIS, 2013:13-22.

[5] CHEN X, CHEN Y, BAJAJ R, et al. Is FPGA useful for Hash joins? [C]//The 10th Annual Conference on Innovative Data Systems Research. Amsterdam: CIDR, 2020: 1-9.

[6] DOSSINGER M, MICHEL S. Scaling out multi-way stream joins using optimized, iterative probing[C]//2019 IEEE International Conference on Big Data. Los Angeles: IEEE, 2019:449-456.

[7] WU W, XUE M, ZHU T, et al. High-throughput parallel SRAM-based Hash join architecture on FPGA[J]. IEEE Transactions on Circuits and Systems II: Express Briefs, 2020,67(11):2502-2506.

[8] WANG H, XIONG F, LI J, et al. Data management on new processors: a survey[J]. IEEE Transactions Parallel Computing, 2018,72:1-13.

[9] ZHANG H, ZHAO B, LI W, et al. Resource-efficient parallel tree-based join architecture on FPGA[J]. IEEE Transactions on Circuits and Systems II: Express Briefs, 2019,66(1):111-115.

[10] HU X, XI J, TANG D. Optimization formulti-join queries on the GPU[J]. IEEE Access, 2020,8:118380-118395.

[11] SIRIN U, AILAMAKI A. Micro-architectural analysis of OLAP: limitations and opportunities[J]. Proceeding of the VLDB Endowment, 2020,13(6):840-853.

[12] HALSTEAD R, SUKHWANI B, MIN H, et al. Acceleratingjoin operation for relational databases with FPGAs [C]//2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines. Seattle: IEEE, 2013:17-20

[13] XUE M, XING Q, FENG C, et al. FPGA-accelerated hash join operation for relational databases[J]. IEEE Transactions on Circuits and Systems II: Express Brief, 2020,67(10):1919-1923.

[14] LIU F, BLANAS S. Forecasting the cost of processing multi-join queries via hashing for main-memory databases [C]//Proceedings of the 6th ACM Symposium on Cloud Computing. Kohala Coast: Association for Computing Machinery, 2015,153-166.

[15] 宗枫博,赵宇海,王国仁,等. 面向多表数据连接投影和连接顺序的优化方法[J]. 计算机科学与探索, 2022, 16(1): 106-119.

[16] DOSSINGER M, MICHEL S. Optimizing multiple multi-way stream joins[C]//2021 IEEE 37th International Conference on Data Engineering. Chania: IEEE, 2021:1985-1990.

[17] WI S, HAN W, CHANG C, et al. Towards multi-way join aware optimizer in SAP HANA[J]. Proceeding of the VLDB Endowment, 2020,13(12):3019-3031.

[18] 马竹琳,李心池,诸葛晴凤,等. 面向非易失性存储器的多表连接写操作的优化研究[J]. 计算机学报, 2019,42(11):2417-2428.

[19] 张延松,于利胜,王珊,等. 内存存储模型上的多表连接优化技术研究[C]//第 26 届中国数据库学术会议论文集. 南昌:中国计算机学会, 2009:303-313.

[20] SINGHAL R, ZHANG Y, ULLMAN J, et al. Efficient multiway Hash join on reconfigurable hardware[C]// Technology Conference on Performance Evaluation and Benchmarking. Los Angeles, USA: Springer, 2019:19-38.

[21] KEMPER A, KOSSMANN D, WIESNER C. Generalised Hash teams for join and group-by[C]//Proceedings of the 25th International Conference on Very Large Data Bases. San Francisco: VLDB, 1999:30-41.

[22] BIZARRO P, DEWITT D. Adaptive and robust query processing with SHARP, Technical Report #1562[R]. Madison: University of Wisconsin, 2006.

FPGA-based accelerator by software-hardware co-design for multi Hash join

WU Jingya, LU Wenyan, YAN Guihai, LI Xiaowei

(State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Science, Beijing 100190)

Abstract

It is hard to implement multi-table Hash join on hardware accelerators. On one hand, multi-table join has an indefinite number of tables and various connection modes. The flexibility in multi-table Hash join is in contradiction with fixed hardware architectures. On the other hand, the capacity of intermediate results expands with the number of tables increasing. The capability of data management and monitoring asks for higher hardware overhead. To enable flexible and efficient multi-table Hash join, a software-hardware co-optimization methodology is proposed. Software subsystem abstracts multi-table Hash join into forward and reverse computation modes, and agilely organizes Hash join processes. Additionally, the memory access and computing are collaboratively optimized in hardware design. A regular hardware Hash table is designed to improve memory bandwidth. Meanwhile, a homogeneous computing engine is designed to perform both forward and reverse computation. To further improve the efficiency of Hash join, multi data channels and an instruction control system are configured. The experiment results showed that a single computing engine could improve the performance of multi-table Hash join 9.2-11.0 times higher than contral processing unit (CPU). Furthermore, the 8-way parallel multi-table Hash join engines could make full use of DDR bandwidth resources and get 71.1 times performance of CPU.

Key words: field programmable gate array(FPGA), multi-table join, Hash join, hardware-software co-design