GPU 数据库实现技术发展演进

刘 鹏 陈 红 张延松 李翠平

(数据工程与知识工程教育部重点实验室 北京 100872)
(中国人民大学信息学院 北京 100872)
(数据库与商务智能教育部工程研究中心 北京 100872)

摘 要 爆炸式增长的数据对存储和处理数据提出了更高的需求,GPU 数据库作为新硬件数据库的一个重要分支,在大容量和高性能处理方面有其独特的优势.GPU 数据库作为高性能数据库的代表,在最近几年受到学术界和产业界的关注,一批具有代表性的研究成果和标志性的实际产品已经出现.GPU 数据库的技术发展按照 GPU 加速型和 GPU 内存型两种技术路线展开.两种技术路线都有相应的原型系统或产品出现.虽然两种 GPU 数据库的发展路线在实现上有所不同,但 GPU 数据库最基本的功能部分和核心技术是相似的,都有查询编译、查询优化、查询执行以及存储管理等功能.当前主流的数据传输方案除了 PCIe 之外,NVLink、RDMA 和 CXL 等传输方案也为不同处理器之间的数据传输提供了更多的可能性.大多数 GPU 数据库使用列存储模型来存储数据,少数 GPU 数据库(如 PG-Strom)对两种存储模型都支持.在列存储模型上利用压缩技术能减少数据的存储空间和传输时延.在 GPU 数据库上进行的压缩和解压的时间应该在整个数据处理的过程中占比很少.在 GPU 数据库上建立和维护索引不应该有很大的系统开销.JIT 编译时间短、编译效率高,是 GPU 数据库编译的主流.操作符对数据库查询性能的影响非常明显,连接操作、分组聚集和 OLAP 运算符是目前研究最多的三个类型.目前大多数的研究中,连接和分组聚集算子通常结合在一起研究.在连接算子执行的过程中还和表的连接顺序结合在一起进行考虑.OLAP 算子是 GPU 数据库中的又一个被大量研究的算子,GPU 数据库在 OLAP 算子和模型方面持续受到研究者的关注.GPU 数据库有三种查询处理模型,即行处理、列处理和向量化处理.向量化处理和列处理在实际系统中应用较多.由于 GPU 加速型数据库技术的发展,CPU-GPU 协同处理模型上的查询方案与查询引擎也有一定数量的研究成果出现.当前 GPU 数据库的查询优化研究主要有三部分:多表连接顺序、查询重写和代价模型.然而,GPU 数据库的代价评估模型在目前还没有很好的解决方案,GPU 数据库的查询优化在未来仍有很大的研究空间.事务在 GPU 数据库中没有得到很好的研究,尽管有单独的原型系统,但目前的研究还没有取得重大进展.本文总结了 GPU 数据库各种关键技术已有的研究成果,指出 GPU 数据库当前存在的问题和面临的挑战,对未来的研究方向进行了展望.

关键词 GPU 数据库;数据压缩;算子优化;OLAP 查询;查询处理
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2024.02691

The Evolution of GPU Database Implementation Technologies

LIU Peng CHEN Hong ZHANG Yan-Song LI Cui-Ping

(*Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China, Beijing* 100872)
(*School of Information, Renmin University of China, Beijing* 100872)
(*Engineering Research Center for Database and Business Intelligence (MOE), Renmin University of China, Beijing* 100872)

Abstract The explosive growth of data has increased the demands for data storage and processing. GPU databases, as an important branch of new hardware databases, have unique advantages in high-capacity and high-performance processing. As representatives of high-performance databases, GPU databases have attracted the attention of both academia and industry in recent years, with a

收稿日期:2023-09-20;在线发布日期:2024-07-17.本课题得到国家自然科学基金(62072460,62076245,62172424,62276270)、北京市自然科学基金(4212022)资助.**刘 鹏**,博士研究生,中国计算机学会(CCF)学生会员,主要研究方向为新硬件数据库、内存数据库.E-mail:cs_liupeng@ruc.edu.cn.**陈 红**(通信作者),博士,教授,博士生导师,中国计算机学会(CCF)杰出会员,主要研究领域为数据库技术、大数据管理.E-mail:chong@ruc.edu.cn.**张延松**,博士,副教授,主要研究方向为新硬件数据库、数据仓库.**李翠平**,博士,教授,博士生导师,中国计算机学会(CCF)杰出会员,主要研究领域为大数据技术、大数据挖掘.

number of representative research results and landmark practical products emerging. The technical development of GPU databases unfolds along two routes: GPU-accelerated and GPU-memory-based. Both routes have corresponding prototype systems or products. Although these development routes differ in implementation, the basic functionalities and core technologies of GPU databases are similar, including query compilation, query optimization, query execution, and storage management. The rapid development of new hardware offers more possibilities for data processing, storage, and transmission. Current mainstream data transmission solutions, besides PCIe, include NVLink, RDMA, and CXL, which provide more possibilities for data transfer between different processors. Most GPU databases use a columnar storage model for data storage, while a few GPU databases (such as PG-Strom) support both storage models. The columnar storage model can utilize compression techniques to reduce data storage space and transmission latency. Data compression schemes on GPU databases generally adopt lightweight compression methods, ensuring that the time spent on data compression and decompression constitutes a small portion of the overall data processing time and does not significantly increase the system's time overhead. Building and maintaining indexes on GPU databases should be lightweight and should not incur significant system overhead. Compilation time directly affects query performance, with JIT compilation being the mainstream for GPU database compilation due to its short compilation time and high efficiency. Operators significantly impact database query performance, with join operations, group aggregation, and OLAP operators being the most studied types. In most current studies, join and group aggregation operators are often researched together, considering the join order of tables during the execution of join operators. OLAP operators are another extensively researched type in GPU databases, with the advantages of GPU databases in handling analytical workloads drawing continuous attention from researchers. GPU databases have three query processing models: row processing, column processing, and vectorized processing. Vectorized processing and column processing are more commonly applied in practical systems. Additionally, due to the development of GPU-accelerated database technology, a certain number of research results on query schemes and query engines for the CPU-GPU collaborative processing model have emerged. The query optimization of GPU databases mainly involves three aspects: multi-table join order, query rewriting, and cost models. However, there is currently no good solution for the cost evaluation model of GPU databases, indicating that query optimization in GPU databases still has significant research space in the future. Transactions, a major feature and an important function of database systems, have developed very maturely and comprehensively on disk databases. However, this critical technology has not been well studied in GPU databases. Although there are individual prototype systems, current research has not achieved significant progress. This paper summarizes the existing research results of various key technologies of GPU databases, points out the current problems and challenges faced by GPU databases, elaborates on the overall development trends and evolution processes of GPU databases, summarizes the most promising research points at present, and provides an outlook on future research directions.

Keywords GPU database; data compression; operator optimization; OLAP query; query processing

1 引 言

图形处理器(Graphics Processing Unit,GPU)从诞生以来,以其高速、性价比高、大规模并行计算性能优异而广受学术界和产业界的青睐.数据库作为计算机系统的基础软件,在整个信息产业中的地位不言而喻.

随着数据量的飞速增长，数据库的查询性能越来越多地受到数据访问带宽和数据处理性能的制约. 传统磁盘数据库受制于较低的磁盘访问带宽(如磁盘带宽几百 MB/s，最新 SSD 带宽 10GB/s)，数据处理性能较低；新兴的内存数据库在带宽性能和多核并行处理性能方面得到极大的提升(最新 DDR5 带宽达到 500 GB/s，核数量 96～128 核)，但核心数量增长相对带宽仍然较为缓慢，数据库查询处理能力相对于 10 TB 以上的内存存储容量仍然难以提供理想的分析处理性能. GPU 配置大容量 HBM 高带宽内存和接近 2 万个计算核心，相对于传统以 CPU 为中心的内存计算平台提供了强大的亚秒级分析处理能力，在硬件上为数据库的数据访问和并行计算提供了最为强大的硬件支持，也是解决大数据时代数据库实时查询处理能力的关键手段. 从 2003 年第一个 GPU 数据库诞生开始，GPU 数据库作为数据库领域的一个重要分支，已经走过了 20 年的历史[1]. GPU 数据库一词作为数据库领域经常出现的一个术语，通常是指使用 GPU 实现数据库的一些操作. 因此，GPU 数据库包含下面两种形态：(1) 开发出基于 GPU 架构的数据库系统；(2) 使用 GPU 来加速某些数据库的功能. GPU 数据库(GPU database management system or GPU accelerate database management system)的概念从提出至今，已经取得了较为长足的发展，无论是在学术领域还是在商业领域，都有一批原型系统或商业产品问世[2]. GPU 数据库技术随时间演变趋势如图 1 所示.

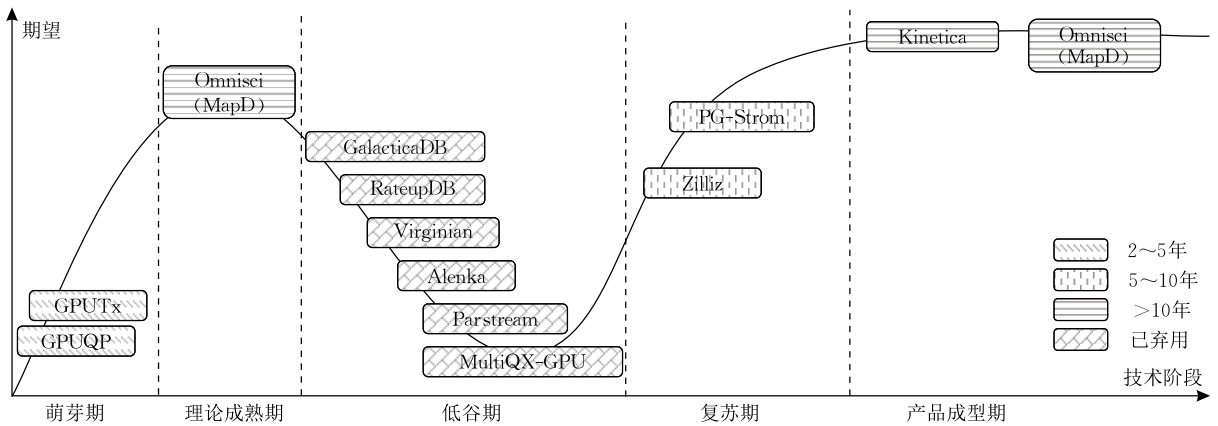

图 1　GPU 数据库发展历史趋势图

在 GPUQP[3] 系统中，每个关系操作都被 CPU 或 GPU 执行. GPUQP 奠定了 GPU 加速数据库系统的先河，它使用内存作为主要的存储设备，有效的减少了磁盘 IO 的传输代价. 在 2011 年，He 等人开发了 GPUTx[4]，他们使用 GPU 来进行事务处理，因此 GPUTx 也是目前已知的能够支持事务的 GPU 数据库. GPUTx 执行预编译的存储过程，这些存储过程被分组到一个 GPU 内核中. 2013 年 CoGaDB[5]、GPUDB[6]、MapD(Omnisci)[7-8]、Ocelot[9] 等 GPU 数据库开始出现，其中 Omnisci 作为 GPU 数据库理论成型的标志性系统，它的存储管理方案、查询编译和查询处理模型成为后续 GPU 数据库系统的样例. 在 Omnisci(现改名为 HeaveDB)出现之后，GalacticaDB[10]、RateupDB[11]、Virginian[12]、Parstream[13]、MultiQX-GPU[14] 等 GPU 数据库和原型系统陆续诞生. 然而，后面出现的这些 GPU 数据库在技术成熟度上并未得到显著的提升，同时期出现的 GPU 和前一代相比，处理速度和带宽得到明显的提高. 前期技术稳定的 GPU 数据库在新一代 GPU 平台上发挥了更大的性能优势. 这些因素导致后续出现较晚的部分 GPU 数据库停止更新和迭代. 停止迭代更新的 GPU 数据库如图 1 中“已弃用”标识所示. 在大批 GPU 数据库不在更新之后，以 Zilliz 为代表的 GPU 加速数据库和 PG-Strom 为代表的 GPU 加速磁盘数据库在系统级功能上较为完备，成为 GPU 数据库复苏期的代表性产品，但在性能上存在不足，已在技术上转型或发展相对缓慢. 在进入产品成型期之后，Kinetica 被称为新一代 GPU 数据库. 它不仅包括了经典的传统数据库的特色，而且提供了更高级的 GPU 加速计算、数据可视化和更加友好的用户交互界面. 在这一时期内，传统数据库厂商也推出了自家公司的 GPU 加速数据库产品，如 Oracle 公司的 Oracle Database In-Memory，微软公司用 tensor 实现的 GPU 数据库原型系统

TQP[15],其他传统数据库厂商也在跟进,陆续推出了他们自家的 GPU 数据库.GPU 数据库的技术迭代和商业化发展趋势持续推进.

GPU 数据库在实际应用中主要体现为四个方面:(1)实时业务计算的需求;(2)一站式数据管理的需求;(3)极端计算能力的需求;(4)硬件快速发展需求[11].对于实时业务计算需求来说,其目标是及时处理和更新数据,用这些数据来获取瞬间的业务价值.最典型的例子莫过于自动驾驶和实时车辆管理需求.例如,Uber 公司使用的 AresDB[16]方案和 USPS 使用的 Kinetica 方案.对于一站式数据管理的需求来说,用户需要一站式的数据解决方案来对数据的写入和分析进行统一,与常用的两阶段解决方案(RocksDB[17]+Spark)相比,单一的数据库可以明显地降低用户的使用和管理成本.对于极端计算能力的需求,爆炸性增长的非结构化数据严重影响了单机 GPU 数据库系统有限的计算能力.这些需求希望通过硬件加速解决方案实现较高的性能,例如 DNA 测序、空间数据分析[18]、三维结构分析[19]等方面.对于硬件快速发展需求来说,GPU 的快速发展使得其能够提供越来越多的并行计算能力和巨大的存储空间,当用户使用 NVLink 协议[20],单个服务器可以提供快速访问 GPU 内存池的功能.因此,硬件的快速发展需要创造一种类似 GPU 风格的内存计算环境,这类工作已经在近几年的数据库研究中有所体现[21].

本文对 GPU 数据库各部分的关键技术做了综述,对 GPU 数据库各部分已解决和未解决的问题进行了分析和归纳总结.找到了当前 GPU 数据库发展所面临的挑战和有研究潜力的技术点.本文除了介绍 GPU 数据库的各部分技术细节之外,对影响 GPU 数据库发展的硬件平台、数据传输方案等方面的研究也进行了综述.和现有的 GPU 数据库综述论文[2]相比,本文增加了 GPU 数据库发展的历史趋势、GPU 数据库技术发展路线、跨 GPU 数据传输方案、典型 GPU 数据库系统的分析等内容和章节.除此之外,和前文[2]相比,本文在数据存储和压缩、操作符算子、查询编译与执行等方面补充了更多的文献,并对这些技术做了更为详细的对比分析.

本文第 2 节介绍 GPU 数据库的研究概况;第 3 节介绍 GPU 体系结构和 GPU 之间的高速互联方案;第 4 节从 GPU 数据库存储管理的角度介绍数据存储模型、数据压缩和索引功能;第 5 节介绍查询编译、数据库核心算子、查询执行模型和查询优化等 GPU 数据库核心技术;第 6 节以经典的 GPU 数据库系统为样例,介绍和分析 GPU 数据库技术在实际产品中应用;第 7 节对 GPU 数据库当前面临的问题进行分析,对进一步研究的技术进行展望;第 8 节总结全文.

2 研究概况

当前单核 CPU 的性能已经接近物理极限,同时期的 CPU 和 GPU 之间的性能差距不断加大.对于 GPU 数据库来说,不管是 GPU 加速型还是 GPU 内存型,GPU 的性能直接影响数据库的性能,存储设备带宽决定数据库性能上限.GPU 和 CPU 之间的性能差异在数据库上直接体现为:(1)磁盘数据库用 GPU 进行加速性能提升有限;(2)GPU 数据库的性能优于内存数据库,内存数据库性能优于磁盘数据库;(3)GPU 数据库的两种技术中,GPU 内存型比 GPU 加速型性能更优.GPU 数据库和其他类型数据库的对比测试如图 2 所示[22-23].

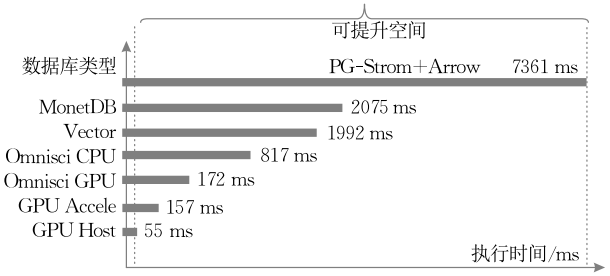

图 2 GPU 数据库和其他类型数据库对比测试

随着时间的推移,CPU 和 GPU 之间的性能差距会进一步加大,而且从图 1 可以看出,GPU 数据库发展到现在已经进入到一个产品相对成熟的阶段,硬件和软件之间的差距也在进一步加大,当前软件的发展速度低于硬件的发展速度,不同时期硬件之间的差距以及硬件和软件之间的差距为 GPU 数据库中的各种实现技术提供了很大的发展空间.

2.1 GPU 数据库体系结构

从体系架构来说,GPU 数据库的体系架构可以分为三种.第一种在磁盘数据库引擎的基础上,通过列式存储,利用 GPU 加速,实现了比 PG-Strom[24]性能更好的数据库.PG-Strom 是在 PostgreSQL 的基础上,通过 GPU 的并行计算,来实现数据库的加速.第一种体系结构的代表性产品有 PG-Strom、Apache arrow[25]等.第二种使用 GPU 加速内存数据库,取得了远高于 GPU 加速磁盘数据库的收益,

代表性产品有 Milvus[26] 等；第三种架构是在 GPU 加速内存数据库的基础上，增加了 JIT 实时编译和 GPU 加速，再加上优化后的向量化执行引擎，执行效率比第二种更高，代表性的数据库产品有 Omnisci[7]. 通过对 GPU 数据库产品的发展历程，我们可以得到的结论是：(1) PG 磁盘引擎内存化和 GPU 加速效果有限；(2) 内存数据库在 GPU 平台上效果更优；(3) 用 GPU 加速内存数据库优于 GPU 加速磁盘数据库.

虽然当前的 GPU 数据库有不同的体系结构，但是从总体上来说，GPU 数据库依然拥有典型数据库的基本功能. 本文分析了 3 个经典的 GPU 数据库系统，提出的 GPU 数据库体系结构如图 3 所示.

本文将 GPU 数据库分为数据传输、存储管理和查询处理三大模块，这三部分涵盖了 GPU 数据库主要的功能和核心技术. 在数据传输模块中，我们介绍了 GPU 上的存储结构和三种不同的数据传输方案. 在存储管理模块中，我们介绍了数据的存储模型、数据压缩和索引. 本文将查询编译、操作符算子、查询执行、事务处理以及查询优化放在查询处理模块中，该模块中的查询优化部分包括 Join order 优化和代价评估两个方面. 本文介绍的主要技术分类如表 1 所示.

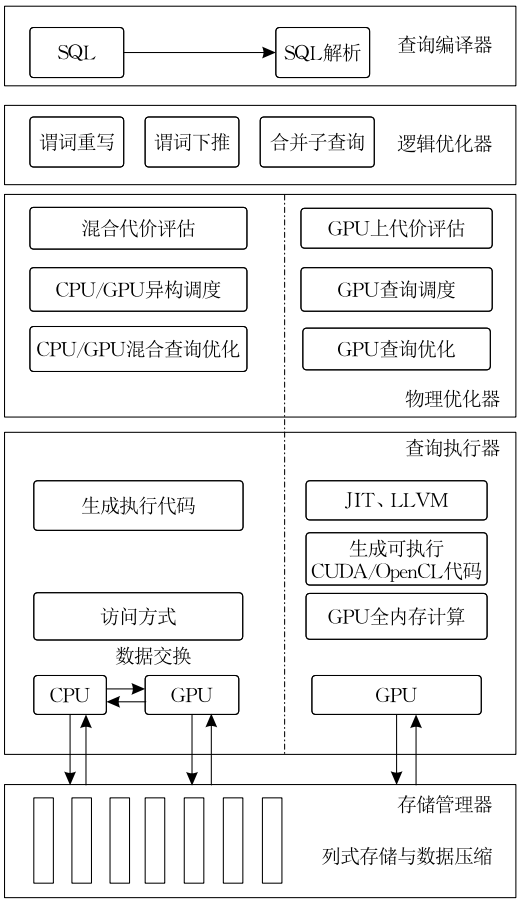

图 3　GPU 数据库体系结构图

表 1　GPU 数据库核心技术分类

功能模块	核心技术	论文数量	文中章节	整体发展情况	待解决的挑战
查询编译	SQL 编译	6	5.1	使用 JIT 查询编译方式提高编译效率	对除 CPU 平台之外的处理器平台支持欠缺
查询优化	代价评估	2	5.4.2	各数据库厂商提供不同的代价模型	GPU 数据库缺乏明确的代价评估模型
	多表顺序优化	3	5.4.1	沿用传统方法，有些方案采用 AI 来训练	GPU 数据库目前没有成熟完整的解决方案
查询执行	操作符算子	32	5.2	用 GPU 优化 join 等耗时算子研究很多	缺乏通用的解决方案，比较依赖 GPU 硬件特性
	执行模型	10	5.3	主要有火山模型和向量化处理模型	使用 GPU 优化向量化处理模型还有待提升
	事务	1	5.5	主流的 GPU 数据库均不支持事务	GPU 在处理事务时需要考虑 ACID 和并行性
	索引	9	4.3	对常见索引类型的 GPU 化	在 GPU 数据库中维护索引开销较大
	查询调度	5	3.2	主要是在 GPU 和 NUMA 架构上的调度	在 GPU 和协同处理架构上查询调度研究较少
存储管理	存储模型	4	4.1	行存和列存两种模型，GPU 更适合列存	GPU 处理列存模型中的非数值属性有待提升
	数据压缩	10	4.2	查询过程中使用轻量级压缩解压	在压缩数据上直接计算还有提升空间

2.2　GPU 数据库技术发展路线

从技术路线来说，GPU 数据库的发展路线主要有两条：一条路线是 GPU 加速引擎，它主要是将部分数据放到 GPU 内存中，将计算密集型工作交给 GPU 来完成，利用 GPU 的并行计算，加速数据库的查询执行操作，主要的挑战在于如何为 GPU 显存选择合适的数据分布和负载分布模型来最小化 PCIe 的数据传输代价，以及如何将 GPU 的计算效率最大化；另一条路线是 GPU 内存处理，它主要是将全部数据都加载到 GPU 内存中，目的是充分利用 GPU 的计算性能和高带宽内存性能，不足之处在于 GPU 的显存容量有限，这使得数据集的大小和稀疏访问降低了 GPU 显存的存储效率.

两种技术路线代表了两种不同的实现方案，GPU 加速型数据库将 GPU 看做是一个协处理器，每个查询通过低带宽的 PCIe 通道将数据从 CPU 传输到 GPU 进行计算，GPU 显存用作 CPU 内存的缓存使用，GPU 加速型数据库的工作原理与磁盘数据库类似，CPU 内存类似磁盘，GPU 显存类似缓冲区，PCIe 通道类似于 I/O 通道，基本假设是缓冲区

(GPU 显存)不够,不足以存储查询处理的全部数据,需要通过 CPU 内存和 GPU 显存之间的数据交换来完成查询处理工作,主要的性能瓶颈是 PCIe. GPU 内存数据库可以看做是 GPU 端的内存数据库,它的基础假设是 GPU 显存能够存储查询的热数据,如 Omnisci. GPU 不断增长的显存容量使显存能够存储的查询数据集也越来越大,为 GPU 内存数据库提供了硬件支持.

2.3 GPU 数据库各部分技术发展概况

在本文列出的参考文献中,共有期刊论文 46 篇,会议论文 63 篇,技术报告 6 篇,其中 CCF A 类论文 58 篇,CCF B 类论文 45 篇,CCF C 类论文及其他技术资料 12 篇. 结合表 1 中的数据,我们可以得到 GPU 数据库各部分技术发展数量如图 4 所示.

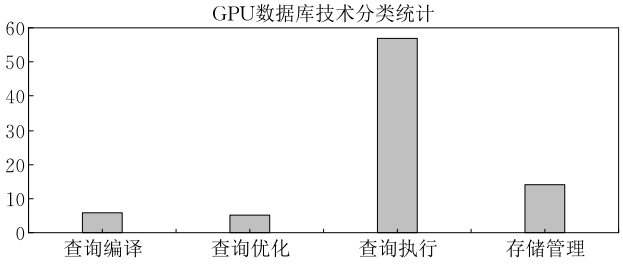

图 4 GPU 数据库技术分类统计

从表 1 和图 4 中可以看出,GPU 数据库的研究主要分布在查询执行和存储管理方面,在查询执行功能部分中,对操作符算子的研究较为充分,主要集中在连接算子和 OLAP 算子两部分. 然而,当前的连接算子依然集中在两表的连接上,对多表连接方面的研究依然有欠缺. 在查询调度中,充分利用新的高速互联设备和各种不同的处理器平台,这些平台和设备上的协同调度依然存在很大的研究空间. 在存储管理方面,现有的存储模型上的研究已经比较充分,数据压缩方面也有相当数量的研究,但是如何在压缩数据上直接进行计算并没有深入地探讨. 在查询优化方面存在很大的研究空间,尤其是在 GPU 数据库上做代价评估方面. 代价评估模型的准确度在一定程度上直接影响了物理计划的生成和执行.

本文除了对 GPU 数据库的核心技术进行分析之外,对完整的 GPU 数据库系统也进行了分析和总结. 通过对关键技术和 GPU 数据库系统的分析,论文总结了 GPU 数据库需要进一步研究的问题,相关内容已经在未来技术展望中较为详细地列出.

3 GPU 体系结构和数据传输

GPU 和 CPU 相比,具有很高的性价比. 从 20 世纪 70 年代诞生到现在,GPU 取得了长足的发展. 利用 GPU 进行程序设计,主要考虑的是数据在 GPU 中的存放位置、存取效率以及和 CPU 之间的连接交互问题. 除此之外,不同平台之间的数据的传输也同样重要. 在数据传输方面,CPU 和 GPU 以及 GPU 和 GPU 之间的数据传输一直是硬件设计开发者和硬件强相关科研工作者重点关注的内容. GPU 内存结构如图 5 所示.

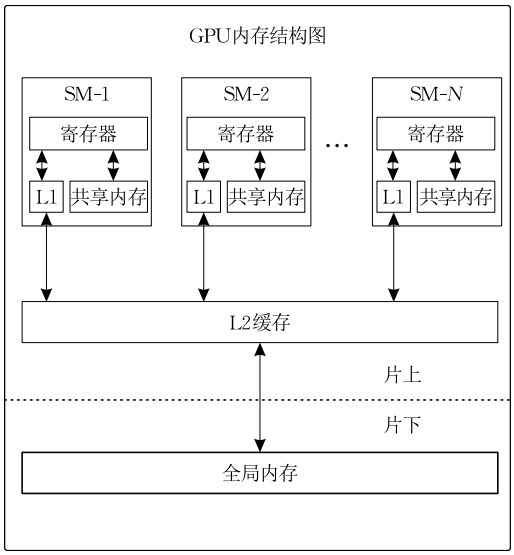

图 5 GPU 内存结构图

本节首先对 GPU 的逻辑结构做一个简要介绍,然后分析和对比 GPU 和其他设备之间不同的数据传输技术. 从数据传输的视角了解不同处理器平台上的数据传输和调度.

3.1 GPU 体系结构

在 GPU 存储结构中,Cache 对程序性能的影响至关重要. NVIDIA GPU 上有几种不同的 Cache,第一种是 L2Cache,它是所有 SM(Streaming Multiprocessor)共享的,一个 SM 由多个 SP(Streaming Processor)加上其他一些资源组成,L2Cache 速度比全局内存快,所以为了提高速度,有些小的数据可以缓存到 L2 上面,容量通常在百 MB 级别;第二种是 L1Cache,它是每个 SM 独享的,SM 内的运算单元能够共享 L1,但跨 SM 之间的 L1 不能相互访问,容量为 KB 级别. GPU 存储单元基本情况如表 2 所示.

表 2　GPU 存储单元介绍

存储单元	位置	是否有 cache	读写	作用域	生命期	容量	速度
Register	On Chip	N/A	R/W	Thread	Kernel	小	快
Local Memory	Off Chip	N	R/W	Thread	Kernel	逐级递减	逐级递减
Shared Memory	On Chip	N/A	R/W	Block	Kernel	逐级递减	逐级递减
Constant Memory	Off Chip	Y	R	Grid	Application	逐级递减	逐级递减
Global Memory	Off Chip	Y	R/W	Grid	Application	逐级递减	逐级递减
Texture Memory	Off Chip	Y	R	Grid	Application	大	慢

3.2　跨 GPU 数据传输

PCIe 作为 CPU 和 GPU 之间的数据传输方案，经过数次迭代，已经达到近百 GB 的数据传输速率. 然而，这依然无法满足高速数据带宽的需求. 除此之外，由于 GPU 连接数量的增加，GPU 之间的通信技术如 GPU Direct、NVLink 和 RDMA 等技术被大量应用. 本小节将对 CPU-GPU 和 GPU-GPU 两种架构下不同的传输技术进行介绍.

3.2.1　PCIe 传输

PCIe(Peripheral Component Interconnect express)是一种高速串行计算机扩展总线标准，它属于高速串行点对点双通道高速带宽传输. 所有连接的分配独享通道带宽. PCIe 从 2003 年的 1.0 版本提出到现在的 6.0 版本，经历了多次更新，传输速度也发生了巨大变化，其基本的参数如表 3 所示. 通过通道数扩展，将原来 PCIe 的传输从单通道(X1)扩展到 16 通道(X16)，得到理论上的最大传输率.

表 3　各个版本的 PCIe 传输速度表

PCIe 版本	时间	原始传输率/(GT/s)	带宽				
			X1/(MB/s)	X2/(GB/s)	X4/(GB/s)	X8/(GB/s)	X16/(GB/s)
1	2003	2.5	250.0	0.50	1.000	2.000	4.000
2	2007	5.5	500.0	1.00	2.000	4.000	8.000
3	2010	8.0	984.6	1.97	3.938	7.877	15.754
4	2017	16.0	1969.0	3.94	7.877	15.754	31.508
5	2019	32.0	3938.0	7.88	15.754	31.508	63.015
6	2021	64.0	7877.0	15.75	30.250	60.500	121.000

Li 等人[27]提出了一种基于优先级的调度策略，旨在为不同的应用程序重叠数据传输和 GPU 执行. 他们还将提出的方案与基于循环调度的 PCIe 调度器相比，使用基于优先级的 PCIe 调度方案，系统吞吐量提高了 7.6%. Raza 等人[28]提出了一种混合物化方法，该方法将迫切需要传输的数据和惰性数据传输相结合. GPU 和 PCIe 吞吐量之间的巨大差距通过高效的数据共享技术来弥补. 他们通过实验证明了标准 PCIe 互联在很大程度上限制了最新的 GPU 的性能.

在协同架构下，CPU 和 GPU 之间的数据传输是目前 GPU 数据库(GPU 加速型数据库)发展的主要瓶颈之一. 由于 GPU 的处理速度远高于 CPU 和 GPU 之间的数据传输速度. 因此，在单位时间内尽可能多地传输数据是目前急需解决的问题. 主要的方法有优先级调度策略、在传输之前的数据压缩策略等. 需要特别注意的是，在传输之前对数据进行压缩和传输之后的数据进行解压缩，以总体开销时间不高于两者之间未压缩的时间为原则. 除此之外，如果能将要操作的数据尽可能多地放在 GPU 显存上，减少 CPU 和 GPU 之间的数据传输，也能在某种程度上克服 PCIe 的传输瓶颈.

3.2.2　NVLink 传输

NVLink 是世界上首项高速 GPU 互联技术，与传统的 PCIe 相比，它能为更多 GPU 系统提供更快速的替代方案. NVLink 技术通过连接多个 NVIDIA 显卡，能够实现显存和性能扩展，从而最大限度的满足工作的负载要求. NVLink 能在多 GPU 之间和 GPU 与 CPU 之间实现高速的连接带宽. NVLink 控制器主要由 3 层组成，分别是：物理层、数据链路层和传输层. NVLink 的传输速率和实现架构如表 4 所示.

表 4　不同版本 NVLink 传输速度表

连接方式	时间	原始传输率/(Gb/s)	实现架构
NVLink1.0	2016	160	Pascal
NVLink2.0	2019	300	Volta
NVLink3.0	2020	600	Ampere
NVLink4.0	2022	900	Hopper

Li 等人[29]通过使用 NVLink 的拓扑研究表明，选择正确的 GPU 组合可以对 GPU 通信效率以及应用程序的整体性能产生相当大的影响.

Lutz 等人[30]分析了 NVLink2.0 的性能和新功能.他们研究快速互联如何有效的执行临时数据传输和大数据量的查询处理.基于 NVLink2.0 设备,他们提出的新的协同处理方法,能有效处理执行过程中的临时中间结果传输和大数据量的查询.他们将提出的协同处理方案在 PCIe3.0 和 NVLink2.0 上进行对比,实验结果表明,和前者相比,NVLink2.0 将处理速度提高了 18 倍.

3.2.3 GPUDirect 传输

传统方案中,当数据需要在 GPU 和另一个设备之间传输时,数据必须通过 CPU,从而导致潜在的瓶颈并增加延迟.GPU Direct 是 NVIDIA 开发的一项新技术,可实现 GPU 与其他设备之间的直接通信和数据传输而不涉及 CPU.GPU Direct 技术主要有如下几个关键特性:(1)加速与网络和存储设备的通信;(2)GPU 之间的 peer-to-peer 传输(GPUDirect P2P);(3)GPU 之间的 peer-to-peer 存储访问(GPUDirect Storage);(4)RDMA 支持(GPUDirect RDMA);(5)针对 Video 的优化.下面对 GPUDirect P2P 和 GPUDirect RDMA 分别进行介绍:

GPUDirect peer-to-peer(P2P)技术主要用于单机 GPU 之间的高速通信,它使得 GPU 可以直接访问目标 GPU 的显存,避免了通过拷贝到 CPU 主机内存作为中转,大大降低了数据交换的延迟.

GPUDirect RDMA 是指计算机 1 的 GPU 可以直接访问计算机 2 的 GPU 内存.它包含了 RDMA 和 GPUDirect 技术的特性,这种数据传输技术为优化查询处理提供了新的机会.

3.2.4 RDMA 传输

RDMA 是 Remote DMA(Direct Memory Access)的简称,它是一种绕过远程主机而访问其内存中数据的技术,解决网络传输中数据处理时延而产生的一种远端内存直接访问技术.RDMA 可以简单理解为利用相关的硬件和网络技术,服务器 1 的网卡可以直接读写服务器 2 的内存,最终达到高带宽、低延迟和低资源利用率的效果.

目前 RDMA 的实现方式主要分为 infiniBand[31]和 Ethernet 两种传输网络.而在以太网上,又可以根据与以太网融合的协议分为 iWARP 和 RoCE.其中,infiniBand 是最早实现 RDMA 网络协议,被广泛用到高性能计算中.但是 infiniBand 和传统的 TCP/IP 网络的差距非常大,需要专用的硬件设备,承担昂贵的价格.相比之下,RoCE 和 iWARP 硬件成本则要低很多.当前,RDMA 技术更多的是和 GPUDirect 技术结合在一起形成 GPUDirect RDMA 来解决数据的传输问题.

Guo 等人[32]使用 GPUDirect RDMA 重新讨论连接算法,针对多 GPU 集群分布式连接算法中的数据通信问题,提出了一种基于 GPUDirect RDMA 的分布式连接算法.实验表明,基于 GPUDirect RDMA 的分布式连接算法与现有的分布式连接算法相比,性能提升了 1.83 倍.

上述的研究表明,GPU 与 CPU 之间连接问题的瓶颈依然存在,但是 NVLink 和 GPUDirect RDMA 这两种技术为多 GPU 之间的连接提供了一个有效的解决方案.与此同时,在 GPU 的发展过程中,NVLink 技术也从高端的 GPU 逐渐向普通的 GPU 下沉,这也使得从硬件层面解决协同架构下的传输瓶颈进一步成为可能.

3.2.5 CXL 传输

CXL(Compute Express Link)技术是一种新型的高速互联技术[33].它允许在计算机系统内部的不同组件之间进行快速、可靠的数据传输.除此之外,它还支持内存共享和虚拟化,使设备之间的协作更加紧密和高效.

CXL 技术具有如下的优势:(1)更快的数据传输速度.CXL 技术可以实现高达 25 GB/s 的数据传输速度,比目前常用的 PCIe 4.0 技术还要快;(2)更低的延迟.CXL 技术可以将 CPU、GPU、FPGA 等计算设备与内存直接连接,避免了传统的 I/O 总线带来的时延,从而实现更低的延迟,提高了计算效率;(3)更高的能效.CXL 技术支持在多台计算设备之间共享内存,降低了内存冗余,提高了能效.此外,CXL 技术还支持内存虚拟化,可以根据应用负载动态分配内存资源,进一步提高了系统能效;(4)更强的可扩展性.CXL 技术可以支持内存扩展,允许在不停机的情况下添加更多的内存容量,从而增加系统的可扩展性,为未来的应用需求做好准备.

CXL 技术的应用场景非常广泛,其中包括数据中心、人工智能和处理器互联等领域.在处理器互联方面,CXL 技术可以实现不同厂商的处理器之间的互联,提高系统的整体性能和灵活性.

3.3 GPU 互联技术总结

在进行数据传输的过程中,按照处理器类型可以分为 CPU-GPU 连接,GPU-GPU 连接,CPU-其他设备和 GPU-其他设备之间的连接.不同的连接技术的适用范围和使用情况如表 5 所示.

表 5　不同传输技术使用情况对比

互联类型	互联方案	技术扩展(实现)	功能描述	优势
单 GPU 卡	PCIe	无	高速串行点对点双通道高带宽传输	能支持多种不同类型硬件设备可扩展性强
	CXL	CXL memory	支持多种平台的新型高速互联技术	能将不同计算设备和内存直连且扩展性高
多 GPU 卡	GPUDirect	GPUDirect Storage	允许 GPU 直接访问存储设备	无需将数据复制到 CPU 的内存中直接访问
		GPUDirect P2P	将数据从源 GPU 复制到同一节点的另一个 GPU 不需要数据暂存	无需 CPU 的参与而直接进行数据访问
		GPUDirect RDMA	GPUDirect 技术和 RDMA 技术结合，允许 GPU 直接访问 RDMA 中的数据	直接在 GPU 和 RDMA 网络设备进行数据传输和通信，显著降低了延迟
	NVSwitch	无	实现了单服务器 8 个 GPU 的全连接	扩展了 NVLink 在 GPU 连接数量上的限制
	NVLink	无	连接多个 GPU 之间或 GPU 与其他设备	解决了 PCIe 相对较低的传输带宽问题
	RDMA	InfiniBand(IB)	通过 RDMA 操作节点之间的高速直接内存访问和数据传输	通过原生 RDMA 支持来快速访问内存和进行高效的数据传输
		RoCE	标准以太网上实现的 RDMA 技术	实现 RDMA 功能来访问远程主机中的数据
		iWAPP	基于 TPC/IP 协议的 RDMA 实现	实现 RDMA 功能来高速访问数据

从表 5 中可以看出，基本的互联方案经过各种定制或优化可以衍生出多种不同的连接方案. 在当前的研究中，PCIe、GPUDirect RDMA、NVLink、InfiniBand 是学术界目前使用较多的互联方案. NVLink 的出现在一定程度上解决了 PCIe 带宽和传输瓶颈的问题，但是在较低速的连接需求中，PCIe 依然是一种适合的解决方案. GPUDirect RDMA 和 InfiniBand 通常在分布式系统和多 GPU 集群中有大量的应用. CXL 具有更好的灵活性和可扩展性，能支持不同设备之间的混合连接.

在数据库系统中，各种互联方案通常是和较耗时的算子、数据置换及任务调度在一起研究. 充分利用不同互联方案的适用场景和特性，根据不同的处理器平台和系统架构，合理地选择所需要的数据传输方案，是提升 GPU 数据库性能的重要方法. 除此之外，在新的传输方案广泛应用之后，新平台下的任务划分和查询调度算法也需要进一步的研究.

4　GPU 数据库存储管理

数据库系统最主要的功能之一就是对数据的存储. 在本节中，我们将讨论 GPU 数据库的存储管理. 本节首先讨论当前数据库的存储结构，通过对存储系统和存储模型的分析，引入行存储和列存储的效率对比. 通过行列存储优劣对比，引入索引和数据压缩的方案.

4.1　数据存储

GPU 数据库按照存储系统可以分为两种，一种是基于磁盘的系统，另一种是基于内存的系统. 在数据库系统根据其需求决定采用哪种存储之后，相应的存储模型也就确定了. 一般来说，基于磁盘的存储系统采用行存储模型，基于内存的存储系统采用列存储模型. 两种格式的数据存储和访问如图 6 所示.

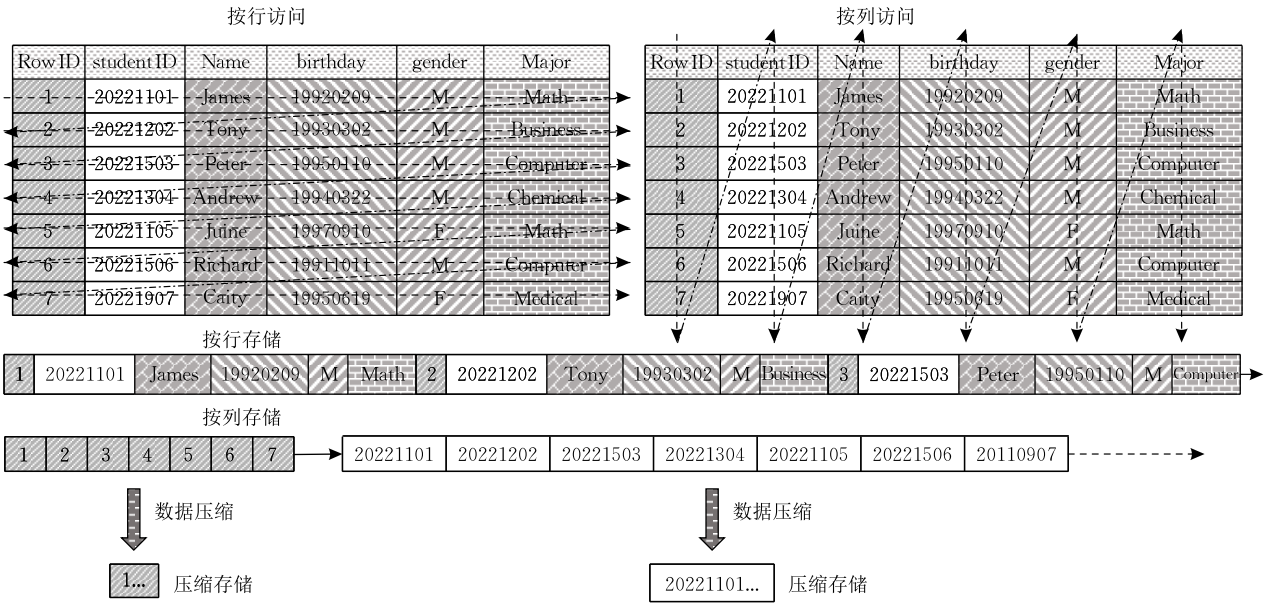

图 6　行列存储和访问示意图

数据库行存储方式在操作符间一次只能传递一条记录,指令的利用率低,也难以使用当前处理器主流的 SIMD(Single Instruction Multiple Data)技术,与行存储相对的是数据库列存储方式采用一次一列(column-at-a-time)技术,能够充分利用当前处理器的向量计算技术实现并行计算,提高 GPU 指令的执行效率.行存储的查询实现技术相对比较复杂,而列存储将查询分解为一系列简单的列操作符,算法实现相对简单,代码的执行效率高.

目前已经出现的 GPU 数据库大多数采用列存储的方式进行存储.采用列存储的一大优点就是可以对数据进行压缩,可以有很高的压缩比.除此之外,采用列存还可以基于 AVX 指令集进行 SIMD 优化,提高数据库的查询执行效率.采用列存储的另外一个优势就是可以采用多数据节点的向量化处理,进一步提高 GPU 的处理效率.行列存储优势对比如图 6 所示.

Ghodsnia[34]比较了行存储和列存储对 GPU 加速查询处理的适用性.他们得出结论,列存储比行存储更合适,因为列存储允许在 GPU 上合并内存访问,能实现更高的压缩率(GPU 内存容量有限)以及减少需要传输的数据量.在列存储的情况下,只有数据处理所需的列必须在处理设备之间传输.而在行存储的情况下,要么必须传输完整的关系,要么必须通过投影将关系简化为处理查询所需的数据.不管是哪种方法,所花费的代价都要比列存储高昂很多.由此作者得出结论,GPU 敏感的数据库管理系统应该使用列式存储[35].

除了存储格式方面的创新,存储引擎相关的研究成果也不断被提出.具有代表性的是混合存储系统和多种数据访问模式.

在混合存储系统方面,Grund 等人[36]提出了一个名为 HYRISE 的主存混合数据库系统.它根据访问表列的方式自动将表划分为不同宽度的垂直分区.对于作为 OLAP 查询的一部分访问列(例如,通过顺序扫描),窄分区执行得更好.因为在扫描单个列时,如果该列的值连续存储,则缓存局部性将得到改善.相反,对于作为 OLTP 风格的查询,更宽的分区执行得更好.因为这样的事务频繁地插入、删除、更新或访问一行的许多字段,并且将这些字段放在同一位置会有更好的缓存局部性.

在多种数据访问模式方面,Alagiannis 等人[37]提出了 H2O 体系,它可以灵活地在单个引擎中支持多种存储布局和数据访问模式.他们证明,虽然现有系统无法在所有工作负载上实现最大性能,但 H2O 总能在不需要任何调优或工作负载知识的情况下实现最佳性能.

CPU/GPU 系统中的缓冲区管理问题与传统的基于磁盘的或内存系统中遇到的问题类似.研究者希望在更小更快的空间(GPU 显存)来处理数据,在更大更慢的空间(CPU 内存)来存储数据.虽然两种存储空间都可以处理数据,但是由于 PCIe 的限制,研究者希望能尽可能地将数据存储在 GPU 显存中.

GPU 缓冲区管理的研究主要集中在分页缓存和列缓存两种方式上.基于页缓存的优点是它的成熟性和广泛性,目前几乎所有的 DBMS 系统都实现了页缓存.但是,如果页面数据量太小,总线未能充分利用.因此,对于页缓存来说,传输少量的大数据集比传输大量小数据集更有效.由于页缓存的一些缺陷,在某种程度上,缓存整个列可能更有益处[35].

从磁盘数据库到 GPU 数据库,已不再是单一的存储模式.对多种存储模式下存储引擎的优化和研究,需要研究者们持续关注.除此之外,数据分区分片技术的引入和 GPU 显存容量的不断提高,将会使 GPU 数据库的容量瓶颈在某种程度上得到解决.

4.2 数据压缩

由于目前大多数的 GPU 数据库采用列存的方式进行存储.而列存储的一大优点就是可以对数据进行压缩.除此之外,由于 PCIe 的传输瓶颈问题,有些数据传输方案提出将数据压缩后再进行传输.因此,数据压缩技术在 GPU 数据库占有重要的位置.数据库中采用的压缩方案主要是无损压缩.无损压缩技术在现代列存数据库中被广泛使用.无损压缩技术按照压缩类型可分为两类:轻量级压缩和重量级压缩.轻量级压缩算法主要用于内存中的列存储,重量级压缩算法主要用于基于磁盘的列存储.GPU 加速的内存数据库中主要采用轻量级压缩方案.目前有五种基本的轻量级压缩方法:参考帧压缩(Frame-Of-Reference,FOR)、字典压缩(Dictionary Compression,DICT)、delta 编码(delta coding,DELTA)、游程编码(Run-Length Encoding,RLE)和可变长度压缩(Null Suppression with variable length,NS).下面对这五种压缩方式进行简要介绍:

(1)参考帧(FOR).将序列中的每一个值表示为给定参考值的差值.FOR 应用于一整个数据块,选择的参考值通常是使所有值为正的最小值.当整

数块具有相似的值时，FOR 压缩的表现效果很好. PFOR 及其变体对整数块进行编码，使得大多数整数可以被编码为 b 位，并将其余整数存储在末尾.

(2) 德尔塔编码(DELTA)[38]. 将每个值表示为与前一个值的差值，当数组已经是排序或者是半排序的情况下，DELTA 效果很好.

(3) 字典编码(DICT). 字典编码方法是利用数据类型的一致性，将相同的值提取出来，生成符号表，每个列值则直接存储该映射成的符号表的 ID 即可. 字典编码其核心思想是利用简短的编码代替列中某些重复出现的字符串. 通过维护一个编码与被代替字符的映射就可以快速确定编码指向哪个字符. 字典编码或域编码是一种重要的压缩形式，字典编码适用于列中存在很多相同字符串的情况，尤其适合低势集列上的压缩处理，它使用双映射将大的域中的属性替换为有限域，这种编码既减少了数据存储，又允许更高效地查询执行，传统的字典编码只支持有效的相等查询，而范围查询要求对编码的值进行解码，以便对谓词求值.

Müller 等人[39]研究了字典压缩的几种方法和变体，即以压缩方式存储域编码字典的数据结构. 对于特定列，哪种方法具有最佳压缩比的问题很大程度上取决于其内容的特定特性. 因此，他们提出了字典格式的非平凡采样方案，使其能够估计到给定列的大小. 通过这种方式，可以识别针对特定列内容的专门压缩方案. Liu 等人[40]通过引入最有序的字典来弥补这一差距. 这些字典使用基于输入数据集采样的字典生成. 在基本有序的字典上进行查询计算会尽可能地避免解码，在有序值比例的降低的情况下，性能会下降.

(4) 游程编码(RLE). RLE 编码的核心思想是将一个有序列中相同的列属性值转化为三元组. 该三元组包括列属性值，在列中第一次出现的位置，出现的次数. 适用于列有序或者列可以转化为有序且列中 distinct 值较少的情况. 通过 RLE，整个列使用两个简单的三元组就可以表示了.

(5) 可变长度编码(NS). NS 的基本思想是通过去除小整数位表示中的前导零来处理位的物理级别. 科研工作者们目前提出了许多不同的 NS 技术，这些技术可以分为位对齐、字节对齐和字对齐. 位对齐的 NS 算法将整数压缩到最小位数，字节对齐的 NS 压缩具有最小字节数的整数，字对齐的 NS 将尽可能多的整数编码为 32/64 位的字. 在数据压缩的过程中，NS 方法通常是和数据布局的方法一起使用. 常见的数据布局方式有两种，水平布局和垂直布局. 在水平布局中，后续值的压缩表示位于后续的内存位置. 在垂直布局中，每个后续值存储在一个单独的内存分段. 在 CPU 和 GPU 平台上已经将多个压缩方案级联在一起，以实现更好的压缩效果.

Wang 等人[41]介绍了数据库和信息检索领域中位图和反向列表的压缩情况，在这些方案中 VB 是 NS 算法的一种变体，具有可变字节对齐封装. Simple-N 及其变体是字对齐压缩方法，具有 4 个状态位来表示位宽的 N 个组合并用于存储数据的 28 个数据位. ByteSlice[42]通过以字节而非位为单位进行单独内存分段来提高 VBP 的性能，但其缺点是需要占用更大的存储空间.

HippogriffDB[43]提出的压缩方案支持五种基本的轻量级压缩技术级联 NSV. NSV 允许以可变长度的字节对每个值进行编码. 为了生成压缩方案，他们使用压缩计划器为每列生成一个计划，根据列操作来进行排序，这些操作将生成具有最佳压缩比的计划.

Mallia 等人[44]介绍了两种新的 NS 算法(GPU-BP 和 GPU-VByte). GPU-BP 以类似于 SIMD 扫描的水平布局对数据进行编码. GPU-VByte 解码具有可变长度字节的输入阵列.

Shanbhag 等人[45]开发了基于位打包的压缩方案及其优化技术在 GPU 上下文中的高效实现，即 GPU-FOR、GPU-DFOR 和 GPU-RFOR. 他们提出的方案可以实现 GPU 中的最优压缩，该方案同时具有很高的解压速度. 除此之外，他们将提出的压缩和解压方案整合成为一个解压模型，在全局内存的单次传递中解压缩编码数据，并与查询执行进行内联.

上述这些方案通常是与 Delta 或 RLE 级联的 NS 算法，因为它们在排序的数据上运行.

除了在常见的编码方式上所做的研究之外，一些压缩框架也被提了出来. 特别是依赖 GPU 硬件相关的压缩框架的提出，丰富了 GPU 数据库压缩方案的选择. 例如，Vijaykumar 等人[46]提出了 CABA 框架，CABA 提供了灵活的机制，可以自动生成 GPU 内核上执行的“辅助 warp”，以执行可以提高 GPU 性能和效率的特定任务. CABA 用于实现数据压缩时，在各种内存带宽敏感的 GPGPU 应用程序中，性能提高超过 2.26 倍.

Lee 等人[47]介绍了 warped 压缩，这是一种用

于降低 GPU 功耗的 warp 级寄存器压缩方案. 它使用了低成本且实现高效的 BDI 压缩方案,并充分利用了 GPU 中使用的存储寄存器文件组织. 通过选择一个线程寄存器作为基值,然后计算所有其他线程寄存器相对于基值的增量值. BDI 可以快速压缩和解压 Warp 寄存器. 实验结果表明,一个 warp 中的线程读写值的差异很小.

上述五种压缩方案和其变体形式如表 6 所示.

表 6 压缩方案和算法总结

压缩方法名称	方法改进	平台	引用	时间	特征描述
FOR	GPU-FOR	GPU	[45]	2022	与 FOR 一起进行位打包,可以处理数据倾斜
DELTA	GPU-DFOR	GPU	[45]	2022	使用带有位打包的增量编码,并针对排序或半排序的数据进行压缩
RLE	GPU-RFOR	GPU	[45]	2022	针对高于平均运行长度的数据,使用 RLE 编码与位进行打包
DICT	非平凡压缩采样	CPU	[39]	2014	以压缩方式存储域编码字典的数据结构
	MOP	CPU	[40]	2019	引入最有序的字典来弥补编码与字典中值的相同顺序
NS	ByteSlice	CPU	[42]	2015	以字节为单位进行分段来提高 VBP 的压缩性能
	NSV	GPU	[43]	2016	允许可变长度的字节对每个值进行编码
	GPU-BP	GPU	[44]	2019	GPU-BP 使用类似于 SIMD 扫描的水平布局对数据进行编码
	GPU-VByte	GPU	[44]	2019	类似于 NSV,可以解码具有可变长度字节的输入阵列
Warp 压缩	CABA 框架	GPU	[46]	2015	利用空闲片上资源来缓解 GPU 执行中的瓶颈
	Warp 压缩	GPU	[47]	2015	利用 GPU 中的存储器文件组织可以快速压缩和解压

从表 6 中可以看出,位打包的压缩方案和优化技术在 GPU 平台上的实现是目前最新的研究思路. 当无法直接在压缩数据上进行计算时,通过将轻量级的压缩方案算法利用 GPU 来实现,能有效地提高数据传输和查询处理的性能. 因此,轻量级压缩算法的 GPU 实现在数据库系统中依然存在价值.

通过数据压缩算法,可以一次性传输较多的数据,从而间接地缩短存储设备和处理设备之间的速度差异. 值得注意的一点是,数据在不同设备之间采用压缩方案进行传输时,压缩和解压缩的代价不应高于未采用压缩的方案. 针对 GPU 数据库中 PCIe 的传输瓶颈问题,采用轻量的压缩和解压方案传输数据,是解决目前 PCIe 瓶颈的一个重要手段.

4.3 索 引

在数据库中,为了加快查询的速度和效率,通常会在数据的某些列上创建索引,使用较多的有 B+树索引、哈希索引等. 在 GPU 数据库中,使用的索引技术已经与传统关系数据库中的索引完全不同. 因此,传统关系数据库中的索引技术无法直接照搬到 GPU 数据库系统中,必须对其进行改进或者是重新设计全新的索引结构. 常见的索引类型如表 7 所示.

表 7 常见索引类型和功能描述,部分总结引用自文献[2]

索引类型	使用场景	功能细节	索引性能表现
GPU LSM[48]	GPU 缓存	可更新日志结构树	N/A
GPU B-Tree[50]	GPU 缓存	实现可更新的 B-Tree 索引	600 M 点查询
Hybrid B+Tree[51]	内存	动态使用 CPU-GPU,使用负载均衡策略来调整计算资源	240 M
Radix Trees[49]	内存	点查询和范围查询	100 M 以上查询效果
Hash Index[52]	内存	对 KV 数据存储进行索引	160+MOPS
Bitmap Index[53]	GPU 缓存	使用字对齐对位图进行混合压缩	实现较高压缩比和压缩性能
CSB+Tree[54]	内存	保留首节点的地址,其余的子节点可以通过偏移量找到	提高了索引对缓存的利用率

基于树的索引查询主要采用分层查找的策略,通过访问尽可能少的点来找到真正的要查找的元组. 在 GPU 数据库的查找过程中,使用多线程和共享内存是一个极为重要的考虑因素. 和 CPU 的实现相比,GPU 对树节点的大小要求不高. 因此可以在 GPU 上实现较为低矮的树形索引结构,同时由于 GPU 的并行计算特性,GPU 索引的查询效率理论上可以达到一个更高的层次.

Ashkiani 等人[48]在 GPU 上实现了日志结构合并树(LSM)的数据结构,该数据结构支持基于 LSM 树的快速插入和删除. GPU LSM 支持查找、计数和范围查询的搜索操作,在排序数组上的搜索速度几乎是原来的两倍. Alam 等人[49]用 GPU 加速自适应基树(ART),该方法与 GPU 上的其他索引搜索方案进行了性能比较,对于点查询的吞吐量达到了 1.06 亿到 1.3 亿次. 对于范围查找,在 6400 万个 32 位的大型数据集上,ART 对稀疏键和密集键分别产生每秒 6 亿次和 10 亿次的查找.

内存中树结构索引搜索是一种基本的数据库操作，现代处理器通过集成多个内核来提供巨大的计算能力. 然而，由于树遍历中的不规则和不可预测的数据访问，树搜索面临巨大的挑战.

B+树作为关系数据库系统中最常使用的一种索引结构，被各种数据库厂商广泛使用. 如何将 GPU 的计算并行性与 B 树/B+树的优良特性结合起来，是 GPU 数据库中一直关注的问题.

将 B-tree 在 GPU 上的首次实现是由 Awad 等人[50]提出，这是一种 GPU 缓存敏感的 B-Tree 节点的数据结构设计. 它使用 warp 协作的共享工作策略，实现了内存合并访问，避免了分支分歧，允许相邻线程运行插入、删除和查询等不同操作. 该实现优于 GPU LSM，特别是在范围查询方面. 此外，如果批量插入的数据小于 100k，该实现也优于 GPU LSM. 该方法主要缓存了树的上层. 当索引大于 GPU DRAM 容量时也拥有较高的吞吐量. Shahvarani 等人[51]在混合 B+树方面也做了一些工作，他们提出了一种基于异构计算平台和 GPU 混合存储体系结构的 B+树新设计. 他们将 CPU-GPU 系统的计算和内存资源联合起来同时使用. 他们的设计受益于混合内存架构，异构平台增加了系统潜在的计算能力，他们使用一个独立的 GPU 来加速索引搜索，它提供了比集成 GPU 更高的计算能力. 新设计的 HB+树每秒可以执行多达 2.4 亿次索引搜索，比使用相同解决方案的 CPU 实现性能高出 2.5 倍.

Zhang 等人[52]用 GPU 加快键值存储操作，他们提出了一个基于 GPU 的键值存储系统 Mega-KV. 实现了高性能和高吞吐量，有效的利用 GPU 的高带宽内存和延迟隐藏能力. Mega-KV 提供快速的数据访问并显著的提高系统的性能，Mega-KV 每秒可以处理多达 1.6 亿次的键值操作，这是传统 CPU 平台上最高效存储系统的 1.4～2.8 倍.

GPU 可以显著减少数据库位图索引查询的处理时间，位图索引通常用于大型只读数据集. Tran 等人[53]提出了三种 GPU 内存算法增强策略：(1)数据结构重用；(2)不同类型对齐的元数据的创建；(3)预分配内存池. 数据结构的重用大大减少了高昂的系统调用次数，元数据的使用利用了位图的不可变特性来预先计算和存储必要的中间结果. 元数据减少了所需的查询处理步骤的数量，预分配内存池可以减少查询处理期间的内存操作开销. 与未增强的实现相比，执行这些策略的组合可以实现 33 到 113 倍的加速.

另一种是基于哈希的索引. Li 等人[55]认为，现有的大多数工作都集中于静态场景，并占用大量 GPU 内存以最大限度地提高插入效率. 在许多情况下，存储在哈希表中的数据会被动态更新，现有方法会使用不必要的大内存资源. 一个简单的解决方案是，当哈希表的大部分空间被填充时，就重建哈希表. 然而这种方法会带来大量的开销. 为此，他们提出了一种新的动态哈希技术. 他们为动态场景设计了一种调整大小的策略. 该策略将搜索性能与调整大小的效率进行权衡，这种权衡可以由用户来配置. 为了进一步提高效率，他们提出了两阶段布谷鸟哈希方案. 该方案可以确保查找和删除操作最多进行两次，同时保留与普通 DyCuckoo 相似的插入性能. 将该方案与 GPU 上的其他哈希实现相比，布谷哈希在细粒度的实现和内存控制方面具有更高的效率.

除了在传统索引结构上的改进，一些索引框架也被陆续提出. 常见的有：张等人[56]提出了一种 GPU 混合访问缓存索引框架 CCHT(Cache Cuckoo Hash Table)，该方案提供了两种适应不同命中率和索引开销要求的缓存策略. 允许写入与查询操作并发执行，最大程度地利用了 GPU 硬件性能与并发特性，减少了内存访问与总线传输. 通过在 GPU 硬件上的实现与实验验证，CCHT 在保证 GPU 缓存命中率的同时，性能优于其他的缓存索引散列表.

通常来说，索引的维护是一种比较耗时的操作. 在 GPU 数据库中，如果需要对树结构的数据进行查询和处理时，使用索引能提高查询的效率. 在 GPU 数据库上进行建立和维护索引时，应当使索引能够在 GPU 缓存中进行创建和维护，这样能有效利用 GPU 的并行计算性能，同时避免频繁向 GPU 内存传输数据而带来的大量数据传输开销.

在 GPU 数据库中，都优先考虑将主要数据或者是热数据直接加载到内存或 GPU 显存. 因此，在高速内存处理方面，创建索引会额外占用存储空间. 另一方面，现有的 GPU 索引技术大多都是基于传统的索引技术改进而来的，并未直接利用 GPU 的特性. 因此，目前的这些索引效果虽然有提升，但是提升有限. 未来在并行索引技术、索引更新等方面可以做进一步的研究.

5　GPU 数据库查询处理

GPU 技术的发展(灰色部分)对查询处理技术的影响(白色部分)如图 7 所示[57].

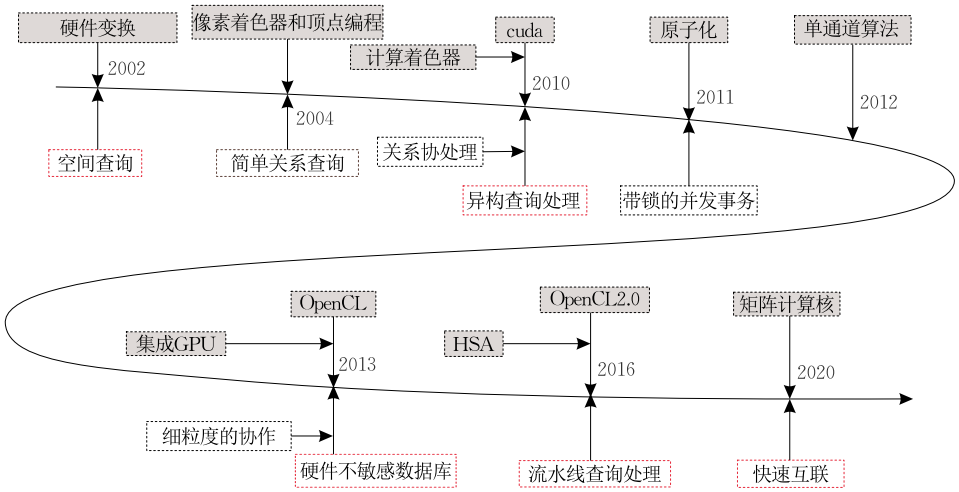

图 7 GPU 与查询处理技术发展时间线

从图 7 可以看到,CUDA 出现之后,异构查询处理模型和关系的协同处理开始发展起来. GPU 上原子化出现之后,带锁的并发事务开始发展,这使得 GPU 理论上也能处理事务相关的操作. OpenCL 的出现使得硬件不敏感的数据库得以出现和发展. 集成 GPU 为细粒度合作提供了基础,异构系统架构(Heterogeneous System Architecture, HSA)和 OpenCL2.0 为流水线查询处理提供了可能.

本节首先分析 GPU 数据库的几种查询编译技术,从查询编译的角度介绍编译优化的技术和研究方向;然后对操作符算子进行介绍和分析,重点分析了连接和分组聚集算子在 GPU 数据库领域的研究和发展;除此之外,对 GPU 数据库领域比较常见的 OLAP 查询也进行了介绍和分析;最后对查询处理模型、查询优化和事务进行了分析和总结.

5.1 查询编译

目前,GPU 数据库系统常见的编译方式有如下四种:(1)利用底层虚拟机(Low Level Virtual Machine, LLVM)的 JIT 编译框架,采用 NVCC 编译工具链进行编译;(2)SQL 虚拟机模式;(3)适配器模式[2];(4)向量化编译等四种方式. JIT 编译是指编译过程时间非常短,编译效率非常高,它是使用 NVCC 编译工具链的底层虚拟机 LLVM 编译器,用它来实现即时编译,用户几乎感觉不到编译的存在,特别适合于较短的程序代码尤其是 SQL 语句;底层虚拟机方式采用 SQL 虚拟机模式,以 SQLite 作为基础,将 SQL 语言转换为操作码用来做中间表示,它将整个 DB 系统认为是一个虚拟机,有效地减少了编译过程对平台和环境的依赖度;适配器编译模式主要是为了解决不同的 GPU 提供商之间的接口兼容问题,通过将 SQL 编译为 OpenCL 或着是在 CUDA 驱动能执行的代码,使得 SQL 语句能在不同的 GPU 平台上执行;向量编译模式主要应用在某些内存数据库上,通过向量化的编译方式能有效提高编译效率,这种编译方式也被一些 GPU 数据库系统所借鉴.

目前已知的这几种编译方式,不同的数据库系统在不同程度上选择性实现一种或者是几种方案,典型数据库系统编译采用的编译方式如表 8 所示.

表 8 典型 GPU 数据库采用的编译方式

DBMS	编译方式	简要描述
CoGaDB	适配器模式	定制查询编译器 Hawk,生成专门代码
GPUDB	适配器模式	将查询编译到驱动程序和程序中
GPUTx	预编译存储	执行预编译存储过程
Ocelot	JIT	使用 MonetDB 的 JIT 编译方式
Omnisci	JIT	基于 LLVM 的 JIT 编译方式
PG-Strom	JIT	基于 LLVM 的 JIT 编译方式
SQream	适配器模式	编译决定代码在 CPU 还是 GPU 执行

为了充分利用代硬件平台,越来越多的数据库系统支持将查询编译为本地代码. Tahboub 等人[58]介绍了 LB2,这是一个高级查询编译器,它从根本上将解释器和编译器结合在一起,它的性能与标准 TPC-H 基准测试上的最佳编译查询引擎相当. Chrysogelos 等人[59]介绍了 HetExchange,这是一个并行查询执行框架,它封装了现代多 CPU 多 GPU 服务的异构并行性,并支持将预先存在的顺序关系操作符并行化. HetExchange 的设计目的是与 JIT 编译引擎一起使用,以便于操作符紧密集成,并为异构硬件生成高效的代码. Paul 等人[60]提出了一个名为 Pyper 的基于 JIT 编译的查询引擎. Pyper 为查询计划转换提供了 Shuffle 和 Segment 两个新

的操作符，这两个操作符可以分别插入到物理查询计划中，以减少执行分歧和解决资源争用，提高了查询执行期间的 GPU 利用率. 实验结果表明，对发散执行和资源争用的分析有助于提高代价模型的准确性. 由于非均匀数据分布引起的控制流发散阻碍指令的执行效率，查询编译技术在结合 GPU 的过程中更容易受到这个问题的困扰. Funke 等人[61]提出了平衡发散技术，他们将平衡发散技术应用到了他们提出的查询编译器 DogQC 中，这种技术使得 DogQC 比其他查询协处理器具有更广泛的功能.

现代硬件中不断增加的单指令多数据(SIMD)功能允许编译数据并行查询，这出现了控制流的分歧导致向量处理单元利用率不足. Lang 等人[62]在指定的架构上(AVX-512)提出了有效算法来解决这个问题，这些算法允许细粒度地将新元组分配到空闲的 SIMD 通道. 除此之外他们还提出了将元组与编译查询管道集成的策略，在处理利用率不足时，执行速度有了大幅度的提高.

将 SQL 查询即时编译为本机代码成为一种可行的查询处理技术，也是主流的基于解释的方法的替代方案. Viglas[63]介绍了他们在这个领域的新的研究成果，解决了从传统的查询编译技术到托管环境中的编译，再到关于中间和本机代码处理的最新方法. 他们参考并类比当代编译器技术中使用的一般代码生成技术，同时对该领域的开放性问题进行研究.

从典型的 GPU 数据库采用的编译方式可以看出，适配器编译方式和 JIT 编译方式是当前 GPU 数据库查询编译的主流. 从最近几年研究成果来看，基于 LLVM 的 JIT 编译方式能获得较高的编译性能，从支持的处理器平台的广度来看，适配器编译方式能支持更多的处理器平台，两种编译方式各有优点，在实际的使用过程中，应该根据具体的需要，进行灵活的应用.

GPU 数据库依赖于传统的数据库编译模型，因此，传统数据库中的大部分编译和优化技术同样适用于 GPU 数据库. 和传统数据库模型所不同的是，引入了 GPU 及其他异构计算平台，使得执行方式和并发控制等方面产生了很大的不同. 基于 LLVM 的 GPU 通用编译工具能够很好地隔离硬件的多样性，在各个编译阶段做到彼此孤立，为 GPU 数据库在编译的各个阶段进行优化提供了可能的方向. 与此同时，JIT 编译和向量化编译也逐渐从理论层面走向具体的应用，对这两种编译方案的优化和完善是目前比较重要的一个研究方向.

5.2　操作符算子

算子是一个函数空间到另一个函数空间上的映射 $O:x\rightarrow y$，即对于任意函数空间 x，通过 Operator 方式将其映射到函数空间 y，Operator 方式(简称 O)即为一个算子. 广义地来讲，任何函数进行某一项操作都可以认为是算子. 广义上的算子可以推广到任何空间. 总而言之，算子就是关系，也是变换. 在数据库中，SQL 算子可以理解为 SQL 语句执行过程中各个步骤的具体操作. 基本的 SQL 算子大概有 9 类，如图 8 所示.

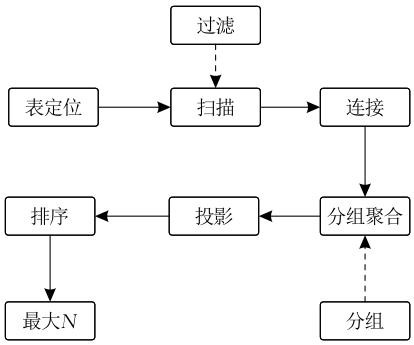

图 8　基本数据库算子

基本算子构成了数据库的基础操作. 在 GPU 数据库中，研究较多的是过滤、排序、TopN、连接和分组聚集等算子.

Filter 算子主要执行条件过滤，用于在 SQL 语句中根据一定的条件过滤表或视图中的数据，filter 算子中至少包含一个过滤条件，且 filter 算子中可以包含非常复杂的过滤条件. 在数据库查询语句中 filter 算子的表现形式为 where 后面的条件，条件的种类和样式非常之多，在执行过滤的过程中，filter 算子更多的是与扫描和索引结合起来，快速地筛选出符合条件的元组. 在提升数据库性能的研究中，filter 算子的功能通常通过 bitmap、布隆过滤器等数据结构实现. Gubner 等人[64]使用布隆过滤器进行查询剪枝，加速了过滤与连接，该方案解决了那些数据密集型 CPU 任务可以用 GPU 加速这一问题.

Sort 算子是 SQL 中常用的操作，order 是 SQL 语句中常用来执行排序操作的算子. 数据库中的 sort 算子更多的是通过将其用类似的排序算法来实现. Guo 等人[65]使用 GPU 对 sort 进行了进一步的研究，他们提出了一种内存访问减少的混合排序，这是一种结合内存有效基数排序和局部排序方法，该方法取得了很好的效果. Schmid 等人[66]将七种不

同模型上最快的分段排序在 GPU 上的实现与各种输入数据场景进行比较，通过对不同段大小和相等段大小的数组进行排序，实验结果表明相同段大小的数组更节约时间.

Top N 算子是完成限定操作的算子的统称. 限定操作指的是基于结果并在结果集上完成某种行为的操作，Top N 算子最典型的应用场景主要是在升序或降序的数据中寻找前 N 条数据.

Shanbhag等人[67]提出了几种用于 GPU 的 top-k 算法，他们在各种基准测试中研究了各种不同 top-k 算法的性能特征，通过改变数据集的大小、k 的值、数据类型和数据的初始分布来进行研究. 他们提出的 bitonic top-k 算法通常优于所有其他算法，比 sort 快 15 倍，比 k 值为 256 的各种其他实现方案快 4 倍.

分组算子最常用的是 Group 和 Having. Group 算子是完成分组操作的核心算子，Having 算子则是提供了分组筛选的条件. Having 子句通常与 Group by 子句一起使用，用来过滤 Group by 子句返回的分组结果集. GPU 上聚合算子的研究如表 9 所示.

表 9 分组聚集算法总结

类型	方法名称	平台	论文	时间	特点描述
并行访问同步	原子级排序聚合	GPU	[68]	2021	使用原子访问同步线程
优化分组聚集	数据压缩类聚合	CPU-GPU	[69]	2018	协同处理提供了显著的加速，压缩能缓解 PCIe 的瓶颈
	对表多层分区	GPU	[71]	2015	GPU 使用共享和全局内存的多级分区哈希算法
实验型分析	动态选择线程参数	GPU	[72]	2019	执行参数严重依赖 GPU 硬件，优化执行参数方法
	GPU 上分组聚合	GPU	[70]	2015	元组数量的变化对 GPU 上分组和聚合性能分析

Gurumurthy 等人[68]对 GPU 执行分组聚集操作进行了深入的研究. 对于 GPU 上的分组计算，要么依赖排序，要么依赖散列. 通过实验证明在 GPU 平台上，基于散列的方法好于基于排列的方法. Tomé 等人[69]研究表明协同处理比任何单独的处理器都提供了显著的额外加速，在分组聚集方面同样也是如此. Karnagel 等人[70]深入研究了向 GPU 加载分组和聚集操作符，带有启发式优化器的查询优化可以在执行时自动选择最佳的性能参数，从而提高查询优化性能.

Meraji 等人[71]利用 GPU 创建分组/聚合操作原型. 他们发现影响 group by/aggregates 算法性能的参数之一是 group 的数量和散列算法. 他们使用 GPU 共享和全局内存的多级哈希算法，与多核 CPU 实现相比，他们的方法获得了 7.6 倍的内核性能改进. Rosenfeld 等人[72]研究了不同的执行参数对 GPU 加速散列聚合的影响，他们发现最佳执行参数高度依赖于 GPU. 针对特定 GPU 优化的执行参数在其他 GPU 上要慢 21 倍. 由于这种硬件依赖性，他们提出了一种在运行时优化执行参数的算法，他们的算法在不到 1%的时间内收敛了结果.

从表 9 中可以看出，并行访问和优化分组聚集依然是 GPU 平台上分组聚集算法的主要研究方向，GPU 平台上的聚集算法一方面依赖于硬件的特性，有时候需要针对特定的 GPU 来优化，另一方面，聚集算法在有序的结果集上效果更佳. 因此，在执行聚集操作之前，利用 GPU 的并行计算特性对输出的结果进行排序，能有效提高聚集算法的性能.

Join 算子是所有连接算子的总称，join 算子本身包含了连接条件. 在连接操作中，等值连接、非等值连接和自然连接是较为常见的连接方法. OLAP 负载主要使用等值连接，GPU 上的连接优化主要也以等值连接为主. 在具体的 join 算法中，主要有嵌套循环连接、排序合并连接、索引连接、哈希连接等. 其中，嵌套循环连接分为外层循环和内层循环，输出满足条件的结果；索引连接算法是指，其中有个表的连接属性已经建立了索引，直接通过索引找到对应的元组，然后连接起来；排序合并连接主要是先排序，后连接，具体操作的时候主要是先检查表有没有排序，如果没有，先对表按照连接属性进行排序，排序后需要对两个表分别扫描一次，输出满足条件的元组；哈希连接是指用相同的哈希函数把两个表的元组散列到 hash 表中，最后把每个桶里面的元组按照条件连接起来. GPU 平台上连接算法的研究如表 10 所示.

Nam 等人[73]发现，现有内存中 OLAP 系统往往不能有效地处理 FK-FK 这种复杂的查询，因为 FK-FK 查询会生成大量的中间结果或是产生大量探测代价. 针对复杂的 OLAP 查询，他们提出了一种有效的 n 元连接算子处理方法. 该方法将 FK-FK 连接转化为图进行处理. 他们将提出的方法集成到一个开源的系统 SPRINTER 中. 通过使用 TPC-DS 进行实验证明，SPRINTER 的性能优于之前最优的 OLAP 系统.

表 10　GPU 上连接算法总结

类型	算法名称	平台	文献	时间	特点描述
Multi-way Join	Sprinter Join	GPU	[73]	2020	在 TPC-DS 上实现 FK-FK 连接
	MHJ,LFTJ	GPU	[74]	2022	GPU 上实现多路连接
	M-way Joins	CPU	[75]	2020	多路连接 aware 优化在 TPC-H 上具有很大优势
GPU 上多表连接	MPDP	GPU	[76]	2022	GPU 加速 1000 个表的连接
	MPDP On GPU	GPU	[79]	2022	GPU 加速 join order 问题
多 GPU 上的连接	Multi-GPU Hash Join	GPU	[77]	2021	用 1000 个 GPU 实现分布式连接
	Hybrid Join	GPU	[78]	2020	多 GPU 上大表连接
	Distributed SMJ	GPU	[32]	2019	使用 RDMA 在 GPU-GPU 上实现分布式哈希连接
GPU 加速连接	Trition Join	GPU	[80]	2022	使用 NVLink 突破 PCIe 限制来实现连接
	GPU 优化 Join	UVA	[81]	2020	Multi-join 上实现两阶段优化技术
	GPU 布隆过滤 Join	GPU	[64]	2019	使用流并行处理技术在GPU和布隆过滤器上加速连接
	SMJ,HJ	GPU	[82]	2019	GPU 上大表连接
	PHYJ	GPU	[83]	2019	GPU 上并行连接算法
	GPU 硬件相关 Join	UVA/UM	[84]	2019	基于分区的硬件敏感的 GPU 连接算法
	NLJ	GPU	[85]	2018	GPU-accelerate index NLJ
	Block Sort/Merge	GPGPU	[86]	2017	GPU 上多种优化技术的连接算法
CPU-GPU 架构连接	Fine-grained SHJ/PHJ	APU	[87]	2013	APU 上连接算法

Lai 等人[74]发现，AP 负载中常见的多路连接在 GPU 上效率很低. 因此，他们用 GPU 加速多路哈希连接(MHJ)和最快情况下最优连接(Leapfrogtrijoin LFTJ). 他们设计了一个基于 warp 的并行化策略来减少线程发散，并促进表上并行搜索中的合并内存访问. 他们通过线程之间的动态工作负载共享和消除结果计数阶段来优化他们的实现方案，该方案适合在 GPU 上运行. 除此之外，他们在实现核外多路连接上使用流水线技术. 他们优化的 MHJ 和 LFTJ 在 NVIDIA V100 上的性能比之前最优的 GPU 算法高 67 倍.

Wi 等人[75]发现，许多不同的二路连接存在不必要的连接枚举. 为了解决这个问题，他们引入了多路(m-way)连接，他们利用 m-way 连接单元特性实现了优化方案，和同类方案相比，该方案在 TPC-H 上取得了良好的性能.

Mageirakos 等人[76]发现，虽然使用 GPU 等加速器在复杂查询方面已经取得了很大的进展. 但是这些方案要么不能有效的并行化，要么在做大量不必要的工作时效率低下. 为了解决这个问题，他们提出了大规模并行动态规划(MPDP)，该方案包含 MPDP 和 UnionDP 两种启发式技术，该优化方案可以优化包含 1000 个连接的查询. 将他们的方案和之前最优的方案进行对比，他们的方案比之前的方案执行效率提高了 7 倍.

Gao 等人[77]提出了一种可扩展的分布式连接算法和一种 GPU 友好的压缩方案，压缩方案用以减少通信量. 他们采用了两极 shuffle 算法针对现代异构 GPU 集群进行了优化. 通过在 1024 个 GPU 上进行实验，可扩展的分布式连接的性能提高 1.77 倍.

Rui 等人[78]针对多 GPU 进行大表连接的瓶颈问题：(1) PCIe 总线数据传输带宽的限制；(2) GPU 之间复杂的通信模式. 提出了三种 join 方式，分别是嵌套循环连接、全局排序合并连接、混合连接. 其中混合连接是基于基数划分和排序合并连接的混合方式. 在这三种 join 方法中，混合连接的数据传输量最低，性能最佳. 同时也证明了多 GPU 间的数据通信和数据交换是最耗时的过程.

GPU 由于受到内存容量和互联带宽的限制，无法将连接扩展到大量数据. Lutz 等人[80]提出了一种新的连接算法 triton join，该算法利用 NVLink 来扩展到大量数据. 在相同的 GPU 上，triton join 比无分区的哈希连接在性能上高出 100 倍以上，在 CPU 上比基数分区连接的性能高出 2.5 倍.

Hu 等人[81]利用 GPU 技术对多连接查询进行优化，他们重点研究了多阶段优化策略和各阶段优化方法. 对于第一点，他们讨论了 GPU 上两阶段优化策略，对于第二点，他们用 GPU 上建立最小代价连接树的方法，在 GPU 上并行执行 intra-join 和 inter-join，以及调度多个连接在 GPU 上并行执行的策略. 通过实验证明，他们提出的多连接方法提高了多连接查询效率，在高负载和复杂连接查询的情况下，该方法比以往的优化方法具有更高的吞吐量.

Guo 等人[82]研究了大表连接和运算符优化，提出了 GPU 上哈希连接和排序合并连接的方案. 该方案基于流水线机制和 GPU 数据分区的方法. 为了更好的利用 GPU，他们使用 shuffle 指令和 CUDA 流，在 NVIDIA GTX1080ti-Pascal GPU 和 TitanV-Volta

GPU 上,他们的哈希连接算法比之前的算法在两种平台上分别提供了 1.51 倍和 1.24 倍的加速.对于排序合并连接,他们的方法在两种 GPU 上分别实现了 3.52 倍和 2.21 倍的改进.

Guo 等人[83]还研究了 GPU 上连接算法的优化,他们提出了 GPU 上的并行混合连接算法 PHYJ.该方法结合了哈希连接和排序合并连接的优点,用流水线机制将数据通信和 GPU 执行进行重叠,他们提出的方法 PHYJ 在 NVIDIA GTX 1080ti-Pascal GPU 上分别比最新的 HJ 和 SMJ 算法加速了 1.72 倍和 1.55 倍,在 TitanV-Volta GPU 上,与基准 HJ 和 SMJ 算法相比,分别可以实现高达 1.54 倍和 1.42 倍的改进.

Sioulas 等人[84]系统的研究了分区的维度、数据的位置和访问模式在 GPU 上连接算法的分析.他们提出了基于分区的 GPU 连接算法的设计和实现.实验表明 Hardware-conscious 在 GPU 上的连接类似于 CPU 上的连接,在 GPU 上使用分区的哈希连接方法优于未分区的哈希连接方法,以及 Hardware-conscious GPU 连接方法可以有效的克服 GPU 的容量限制和传输瓶颈.

Nguyen 等人[85]研究了 GPU 加速索引嵌套循环连接 INLJ 的设计与实现.他们实现了 GPU 加速的 INLJ 算法,并在 voldb 系统上进行了实验,实验结果表明,GPU 加速的 INLJ 算法比 voldb 默认的 INLJ 算法快 2～14 倍.

He 等人[87]在 CPU-GPU 耦合的框架上实现了哈希连接算法,他们研究了有分区和无分区哈希连接的细粒度协同处理机制.他们在 AMD APU 上进行了实验.实验结果表明耦合架构可以实现细粒度的协同处理和缓存重用.细粒度协同处理比纯 CPU、纯 GPU 和传统的 CPU-GPU 协同处理性能分别提高了 53%、35%和 28%.

Paul 等人[88]介绍了近几年如何利用 GPU 的特性实现哈希算法,研究了 GPU 特性对 hash join 的性能影响.他们采用了一种硬件无关的两阶段设计来实现无分区哈希连接算法.在构建阶段,他们采用全局内存原子操作生成哈希表,然后在探测阶段使用两个输入关系探测哈希表.除此之外,他们在实现中对数据倾斜和匹配也做了优化.Paul 等人[89]还研究了多 GPU 体系结构上分区哈希连接的性能,并且证明了跨 GPU 的通信开销会严重阻碍可扩展性.他们开发了 MG Join,这是一种可扩展的分区哈希连接的实现,用于现代多 GPU 体系结构,它利用了所提出的自适应多跳路由策略,有助于最小化共享互联链路之间的拥塞,并确保更高效的使用 GPU 硬件和互联链路.实验结果表明,MG join 将 TPC-H 查询的总体性能提高到开源 GPU 数据库 Omnisci 的多 GPU 版本的 4.5 倍.

Yabuta 等人[90]通过各种配置实现了几种知名的 GPU 连接算法,用来确定 GPU 计算关系连接的性能特性.实验结果表明,基于 GPU 的关系连接与基于 CPU 的关系连接加速比取决于计算核的数量、数据集的大小、连接条件和连接算法.在最佳情况下,非索引连接的加速比达到 6.67 倍,排序索引连接的加速比达到 9.41 倍,哈希连接的加速比达到 2.25 倍,另一方面,基于 GPU 的索引连接实现的执行时间仅比 CPU 的执行时间少 69.6%.

由于很难预测中间结果的性质,理想的哈希连接实现必须既能快速处理典型查询,又能对异常的数据分布进行很好的处理.Birler 等人[91]提出了一个简单但有效的内存中非链哈希表设计,Unchained 表结合了构建分区、邻接数组布局、流水线探测、布隆过滤器和软件写合并等技术.该哈希表在偏斜度很高的数据中实现了显著的性能提升.他们设计的哈希表性能比开放地址寻址高出 2 倍.

从表 10 中可以看出,用 GPU 进行加速连接在近些年研究较多.用 GPU 实现连接算子,或者是将传统的数据库算子 GPU 化确实在一定程度上提高了查询处理的效率.但是,在 GPU 平台上研究连接算子时,更需要考虑在不同的架构下(如 CPU-GPU)以及多 GPU 平台上的连接.不同的处理器平台以及相同平台下处理器的数量对连接算法的影响至关重要.尤其是在异构平台下涉及到数据的传输和调度.除此之外,表之间不同的模式也影响着 GPU 平台上连接算法的性能.除了加速连接之外,在单个平台的多表连接以及在连接过程中多个表之间的顺序,同样也影响着连接的性能.因此,在 GPU 平台上的连接算法的研究,已经不单单是需要做连接算子,而且需要对数据划分、数据调度和传输等方面综合进行考虑.

连接算法是影响数据库性能的主要因素,也是性能提升的关键,连接算法和连接顺序的研究一直是数据库中研究的热点重点和难点,随着新硬件技术的发展,对传统算法的改进和新算法的研究必将在 GPU 数据库和其他新硬件数据库中占有及其重要的位置,也将是未来最有潜力的一个研究方向.

联机分析处理(OLAP)是在海量数据上的基于多维数据模型的复杂分析处理技术，OLAP 为用户提供了基于多维模型的交互式分析和数据访问技术. OLAP 是一种多维数据操作，它为数据提供的多维分析操作包括切片(Slice)、切块(Dice)、上卷(Roll-up)、下钻(Drill-down)、旋转(Pivot). 切片和切块操作是从数据立方体中分离出部分数据，用以进行分析操作. 上卷和下钻对数据进行聚合操作，其中上卷操作增加数据聚合程度，下钻操作降低数据的聚合程度. 旋转操作是一种多维数据视图布局控制操作，对应着数据视图布局的改变.

OLAP 技术分为多维 OLAP(Multidimensional MOLAP)、关系 OLAP(Relational ROLAP)、混合 OLAP(Hybird OLAP，HOLAP). MOLAP 模型采用多维数组存储数据立方体，事实数据直接存储在多维数组中，多维操作可以直接执行，查询性能高. ROLAP 采用关系数据库存储多维数据. 关系模型中没有维度、度量等概念，需要将多维数据分解为维表和事实表，并通过参照完整性约束定义事实表和维表之间的多维关系. ROLAP 在事实表中只存储实际的事实数据，不需要 MOLAP 预设多维空间的存储代价，存储效率高. HOLAP 是一种混合结构，目标是将 ROLAP 对大量数据的存储效率优势和 MOLAP 系统的查询速度优势相结合，将大部分数据存储在关系数据库中，将用户最常访问的数据存储在多维数据系统中，系统地实现在多维数据系统和关系数据库中的访问.

将 OLAP 中经常用到的算子，结合 sum、avg、max、min、count 等各类聚集函数，为用户提供了复杂信息的汇总能力，通过借助 GPU 的并发高性能计算能力，可以有效提升 GPU 数据库的竞争力. GPU 平台上的 OLAP 研究情况如表 11 所示.

表 11　OLAP 研究成果总结

类型	方法名称	平台	文献	时间	特点描述
模型类	3Layer OLAP Model	CPU，GPU	[23]	2020	三层 OLAP 模型，提出了针对数据密集型的向量分组算法
	Fusion OLAP	CPU，GPU	[92]	2019	MOLAP(Multi-dimension)＋ROLAP(Relation)＝Fusion OLAP
	A-Store	CPU	[93]	2016	定制 OLAP 引擎 a-store，将数组索引作为主键来虚拟
算法类	Surrogate Key Index	CPU，GPU	[94]	2019	代理键索引作为外键连接，为 OLAP 定制外键连接
	LMW	CPU	[95]	2013	列存数据库上的连接和物化下推，提出轻量级的物化
	HG-Bitmap join index	CPU，GPU	[96]	2013	OLAP 上使用 CPU/GPU 架构做位图索引

Zhang 等人[23]将 GPU 内存处理模式和 GPU 加速模式两种技术路线集成到 OLAP 查询引擎中，研究了一个定制化的混合CPU-GPU平台上的 OLAP 框架，OLAP Accelerator 设计了 CPU 内存计算、GPU 内存计算和 GPU 加速等三种 OLAP 计算模型，实现了 GPU 平台向量化查询处理技术，优化显存利用率和查询性能，探索了 GPU 数据库的不同技术路线和性能特征，实验结果表明 GPU 内存向量化查询处理模型在性能和内存利用率方面获得最佳性能. Zhang 等人[93]研究表明，内存数据库/GPU 数据库可以从非规范化中受益，因为它能够显著简化查询处理计划并降低计算成本，他们定制了一种 OLAP 引擎 A-Store，它通过将数组索引作为主键来进行虚拟非规范化.

从工作负载特性的角度来看，哈希连接算法可以通过多维映射进一步简化，外键连接算法可以从多个角度而不是单一的性能角度进行评估. Zhang 等人[94]引入面向代理键索引连接作为模式敏感(schema-conscious)的外键连接和 OLAP 工作负载定制的外键连接，以综合评估最新的连接算法在 OLAP 工作负载下的表现. 他们引入代理键索引机制，列索引通过为 PK-FK 约束表创建代理键索引，简化了外键连接作为数据索引引用. 实验结果表明，针对 OLAP 工作负载的自定义外键连接算法比通用哈希连接的性能提高了 1 倍，在大多数的基准测试下，简单的硬件无关的共享哈希表连接优于复杂的硬件相关的基数分区哈希连接，除此之外，具有代理键索引的自定义外键连接算法简化了硬件加速器(如 GPU)算法复杂度，便于不同硬件加速器的实现.

Zhang 等人[97]研究关系操作符在 GPU 平台上的算法实现和性能优化技术，以哈希连接的 GPU 并行算法研究为中心，提出了面向 GPU 向量计算混合 OLAP 多维分析模型 Semi-MOLAP，他们将 MOLAP 模型的直接数组访问和计算特性与 ROLAP 模型的存储效率结合在一起，实现了一个基于完全数组结构的 GPU semi-MOLAP 多维分析模型，简化了 GPU 数据管理，降低了 GPU semi-MOLAP 算法复杂度，提高了 GPU semi-MOLAP 算法的代码执行效率. 同时，基于 GPU 和 CPU 的

计算特点,将 semi-MOLAP 操作符拆分为 CPU 和 GPU 平台的协同计算,提高了 GPU 和 CPU 的利用率以及 OLAP 查询的整体性能.

从表 11 中可以看出,OLAP 近年以来的研究已经从算法方面上升到模型方面,在做 GPU 平台上 OLAP 的研究时,不同 GPU 数据库的数据调度方式、数据划分、数据存储模型等方面都对 OLAP 算子的表现性能影响很大. 因此,在做 GPU 上的 OLAP 研究时,针对不同的 GPU 数据库进行定制化的研究是一种可行的研究方案.

目前 OLAP 优化技术的研究包括算子和模型两个层面,使用 GPU 来加速 OLAP 操作,对混合 OLAP 模型值得进一步研究和分析.

5.3 查询执行

在数据库管理系统中,查询处理和查询执行技术通常都会整合到查询执行器当中,本节首先对查询处理和执行相关的研究做一个分析和总结,然后介绍一些具体的查询执行引擎,最后对查询执行技术进行总结.

在传统的关系数据库中,执行器架构一般采用经典的火山模型(Volcano Model),该模型的执行方式如图 9 所示,它是基于行的执行方式.

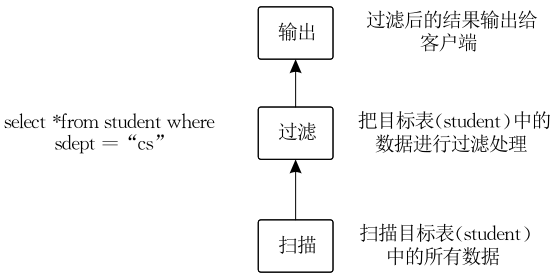

图 9 火山模型图

行处理模型基于编译的推送模型,行存储数据在查询中以行方式访问记录并执行流水线查询处理. 不需要位图和向量索引等中间结果数据,数据库表中的记录被顺序访问,需要大量的分支语句判断记录的处理,在低选择率的情况下处理效果提升不太显著.

列处理模型将流水线处理分解为一系列基于列数据的处理阶段,每个列完成全部的数据处理,代码执行效率和 cache 局部性较好. 但一次一列的处理模式需要将中间结果物化下来,产生额外的物化数据存储和访问代价.

向量化处理模型是一种 cache 内的列处理模型,通过将数据划分为适合 L1 cache 大小的向量,执行基于向量粒度的批量列式处理. 位图、向量索引等中间结果物化在 L1 cache 中,既保持了列处理模型较高的代码执行效率,又通过 L1 cache 的物化机制消除中间结果的内存物化和访问代价,从整体上也体现了流水线处理模型的特征,可以看作是行处理模型和列处理模型面向 cache 结构优化的混合处理模型. 向量化查询处理模型在 GPU 平台上已经有了初步的探索[98],但当前 GPU 数据库主要还是采用列式处理模型[11].

对于查询处理模型,学术界目前没有提出更新的模型,更多的是在上述几种查询处理模型上所做的一些优化工作. 目前研究更多的主要是研究如何利用 GPU 等新硬件加速现有的查询处理模型. 和查询处理模型相比,工业界更关注查询处理引擎的研究和开发,对各种查询处理模型进行落地和实现是他们关注的重点. 除此之外,一些有代表性的工作也被提了出来.

Paul 等人[99]提出了一种新的流水线查询执行引擎 GPL,用来提高 GPU 上查询协同处理的资源利用率. 该引擎与现有的基于内核的执行不同,GPL 利用了新一代 GPU 的硬件特性,包括并发内核执行和内核之间高效的数据通信通道,能高效生成流水线查询计划. 它能基于代价的方式调整流水线查询执行的分片大小. 通过在 GPU 上的 TPC-H 查询来评估 GPL,GPL 能够显著优于之前最优的基于内核的查询处理方法,改进率高达 48%.

在流水线查询执行中,跨处理器元组传输对整个查询执行性能起着至关重要的作用. 之前的解决方案使用类似队列的数据结构实现跨处理器的元组传输. 然而,用粗粒度的锁结构虽然能实现线程安全,但是会导致性能问题. Yang 等人[100]提出了一种细粒度的元组传输机制,他们采用解耦的排队/出队方式,使不同处理器上的两个线程都能同时访问队列. 他们实现了一个原型系统 QC,它采用细粒度的元组传输,QC 比现有的 GPU 数据库(HeavyDB)实现了一个数量级的性能提升.

Chrysogelos 等人[101]提出了一个分析查询处理引擎 HAPE. 这是一个高效并发多 CPU 多 GPU 查询执行的硬件相关的分析引擎,HAPE 将异构硬件上的查询执行分解为高效的单设备查询执行和并发的多设备查询执行. 它使用专门为单个设备执行而设计的硬件感知算法,并通过代码生成将它们组合成高效的设备内硬件感知执行模块. HAPE 将这些模块结合起来,通过处理数据和控制传输来实现多设备执行.

Breß 等人[102]开发了一个混合查询处理引擎.它扩展了传统的物理优化过程,以生成混合查询计划,并结合 CPU 和 GPU 的优点执行基于代价的优化.此外,他们的目标是在不同的 GPU 加速数据库管理系统之间提供一个可移植的解决方案,以最大限度提高适用性,初步结果表明这种实现方法潜力巨大.

Funke 等人[103]对现有执行方式进行研究,并指出了其存在的问题.对于一次一内核操作,需要多次从主存到 GPU 显存传输数据.对于批量处理,GPU 显存带宽限制了内核的数据处理速度.对于一次一向量,实现困难,启动 GPU 内核开销大.他们提出的方案和一次一操作相比,在 SSB[104]上将内存访问量减少了 7.5 倍,从而使内核执行时间缩短了 9.5 倍,最适合做星形连接.与此同时,他们还打破了实际操作功能的操作符和中间结果交错的概念,变为一个复合 kernel,减少了 kernel 的数量,通过片上内存中的流水线来减少 GPU 全局的内存访问量.

在查询引擎的实现层面,Lee 等人[11]开发了 RateupDB,成为国产 GPU 数据库的代表.他们是第一个完整通过 TPC-H 测试的 GPU 数据库系统.Lee 等人将 GPU 引入数据库结构,实现了基于 GPU 数据库的并行算法.在 RateupDB 内部,CPU 执行 OLTP 业务,GPU 执行 OLAP 业务,两种芯片各自承担自己擅长的任务,在同一份数据上协同工作.除此之外,他们充分利用了 GPU 的高带宽和强算力,和传统的数据库相比,RateupDB 性能得到了巨大的提升.

虽然现代 GPU 拥有比以往更多的资源(例如更高的 DRAM 带宽),然而高效的系统实现和查询处理资源分配依然是优化性能所必须的.数据库系统可以通过刚好足够的资源分配来节省 GPU 运行时的代价,或者是利用新的 GPU 资源分配能力来提高并发查询处理的查询吞吐量.Cao 等人[105]对四个 GPU 数据库系统(Crystal、HeavyDB、BlazingSQL、TQP)进行了跨堆栈性能和资源利用分析.他们将资源分配进行了优化.使用这些优化,他们将单查询执行的延迟将近缩短了一半,将并发查询执行的吞吐量提高了 6.5 倍.

除了处理模型之外,软件预取技术对查询处理同样重要.使用软件预取技术来提高带宽利用率是磁盘数据库中常见的做法,因为它重叠了计算成本和内存访问延迟.为了研究软件预取技术在 GPU 查询处理中的有效性,Deng 等人[106]在 GPU 上实现了四种软件预取方法,即组预取(GP)、软件流水线预取(SPP)、异步内存访问链(AMAC)和交错多矢量化(IMV).他们将这四种方法在哈希连接探测和 B 树搜索任务上进行实验.与没有软件预取的实现相比,它们可以在哈希连接探测上实现 1.19 倍的加速,在 B 树搜索上实现 1.31 倍的加速.结果证实了软件预取技术在 GPU 查询处理中的有效性.

从最近几年的研究成果来看,在查询处理模型方面的创新基本上很少,更多的是利用硬件的特性对模型进行的优化.或者是根据特殊的需求对查询处理模型进行定制.为了进一步提高查询处理的性能,一些研究团队从查询执行引擎着手,对现有的查询引擎进行工程层面的优化,或者是提出一些新的查询处理引擎,这种方案在一定程度上也提高了查询处理的性能.

查询处理模型和查询处理引擎的研究是数据库系统从理论走向产品落地的重要一步,当 GPU 数据库的理论体系走向稳定和成熟的时候,查询处理模型也趋于稳定,研究人员更多的是将注意力放在查询处理模型的优化和查询引擎的优化与实现等方面,对于这两方面的工作也需要研究者们关注.

5.4　GPU 加速查询优化

数据库查询优化技术追求的目标,就是数据库查询优化引擎生成一个执行策略的过程中,尽量使得查询的总体开销(包括 IO、CPU、GPU、数据传输、网络传输等)达到最小.

查询优化技术主要包括:(1)查询重用技术;(2)查询重写规则;(3)并行查询优化技术;(4)分布式查询优化技术;(5)框架结构的优化技术等.这几项技术构成了一个广义的数据库查询优化的概念.与之相对的是一个狭义的查询优化概念,其中包括了查询规则重写、查询算法优化以及多表连接顺序的选择等.

对于查询优化技术按照不同的级别,可以划分为以下几种层次:(1)语法级.查询语言层面的优化,基于语法进行优化;(2)代数级.查询使用逻辑形式进行优化,运用关系代数原理进行优化;(3)语义级.根据完整性约束,对查询语句进行语义理解,推知一些可能的优化操作;(4)物理级.物理优化技术,基于代价估算模型,比较得出各种执行方式中代价最小的查询计划.

本小节将从多表连接优化和代价评估两方面对 GPU 数据库的查询优化技术进行分析.

5.4.1 Join Order 优化

目前,多表连接的顺序问题依然是数据库优化领域的一个开放问题.GPU 数据库目前也没有一个成熟完整的方案来解决这个问题.

多表连接算法需要解决的两个问题:(1)多表连接的顺序.表的不同连接顺序,会产生许多不同的连接路径,不同的连接路径有不同的执行效率;(2)多表连接的搜索空间.由于多表连接的顺序不同,产生的连接组合体会有多种,如果这个组合的数目巨大,连接次序会达到一个很高的数量级,最大的可能是 $N!$(N 的阶乘).如何将搜索空间限制在一个可接受的时间范围之内,并生成高效的查询执行计划.目前这个问题依然没有很好的解决.

多表之间的连接顺序表示了查询计划树的基本形态,一棵树就是一种查询形态.SQL 的语义可以由多颗这样的树来表达,从中选择执行代价最小的树,就是寻找最优查询计划的过程.通常情况下,大多数的实现系统倾向于左深树、右深树、紧密树这三种.不同的连接顺序,会生成不同大小的中间结果,对应的处理器消耗和 IO 的消耗不同.

在找最优查询树的过程中,形成了两种策略:(1)自顶向下.从 SQL 表达式树的树根开始,向下进行,估计每个节点可能的执行方法,计算每种组合的代价,从中挑选最优的;(2)自底向上.从 SQL 表达式树的叶节点开始,向上进行,计算每个子表达式树的所有实现方法的代价,从中挑选最优的,再和上层(靠近树根)的节点进行连接,直到树根.在数据库的实现中,多数数据库系统采用了自底向上的策略.

常见的多表连接顺序方法如表 12 所示.

表 12 常见的多表连接顺序方法

方法名称	特点和适用范围	缺 点
启发式方法	适用于任何范围,与其他算法结合,能有效提高整体效率	不知道得到的解是否最优
贪婪算法	非穷举类型算法,适合解决较多关系表的搜索	得到局部最优解
爬山法	适合查询中包含较多关系的搜索,基于贪婪算法	随机性很强,得到局部最优解
遗传算法	非穷举类型的算法,适合解决较多关系的搜索	得到局部最优解
动态规划算法	穷举类型算法,适合包含较少关系的搜索,可得到全局最优解	搜索空间随着关系的个数增长呈指数增长
System R 优化	基于自底向上的动态规划方法,可得到局部最优解	搜索空间可能比动态规划算法更大一些

现代数据分析工作负载通常需要在大量表上进行查询,在可接受的时间范围内得到这些查询的最佳查询计划至关重要.对于涉及许多表的查询,查找最佳连接顺序成为查询优化中的瓶颈,由于连接顺序优化的复杂性,优化器在表的数量超过阈值之后采用启发式解决方案.因此,解决 join order 问题的主要目标之一就是减少最优方案的优化时间.为了解决这个问题,Mancini 等人[79]提出了一种新的大规模并行算法 MPDP.该方案在 GPU 上实现,它可以有效地修剪大规模搜索空间,同时利用了 GPU 的大规模并行性.当使用 PG 对实际的查询进行评估时,MPDP 比之前的方法快一个数量级.此外,为了处理更多表的查询,他们将 MPDP 优化到 IDP2,该优化能搜索更大的空间并且计算速度也更快.

在多表连接方面,Meister 等人[107]讨论了用 GPU 加速多表连接的顺序问题,以及使用动态规划算法对多表连接时所面临的挑战做了讨论.动态规划算法是顺序方法,并不容易并行化.GPU 的存储结构和执行方式与 CPU 完全不同,拥有更大规模的并行处理能力.这在理论上是有潜力彻底解决多表连接的顺序问题,但是需要解决分支消除、传输瓶颈、访存优化和并行计算等问题.

在 GPU 数据库中,选择一个好的连接顺序对于高效执行查询仍然非常重要.连接顺序的选择主要基于中间连接结果的基数估计.然而,存储器的顺序或随机访问对最终的结果产生巨大的影响.Schubert 等人[108]通过评估 n-ary 外键连接的执行时间来验证上述的猜想.他们提出了一种新的连接顺序排序算法,该算法能在运行时根据内存的访问模式来调整相应的连接顺序,实验结果表明,他们的自适应重排算法能快速收敛到一个良好的连接顺序,并达到 5.7 倍的改进.

从目前的研究成果来看,对 GPU 数据库和内存数据库的代价评估和连接优化问题,使用传统的解决方案和模型都已无法满足需要.通过在 PostgreSQL 上使用学习的方法来做多表连接顺序和代价评估的成功经验证明,这种模式在 GPU 数据库上也必然可行.因此,使用学习的方法在 GPU 数据库上做代价评估和多表连接顺序选择将会成为一个十分具有潜力的研究方向.

5.4.2 代价评估

准确预测操作符执行时间是数据库查询优化的前提.数据库研究者们尽管对传统的基于磁盘的 DBMS 进行了广泛的研究,但在 GPU 数据库中的

代价模型建模仍然是一个开放性问题.内存访问越来越成为数据库操作的一个重要代价组成部分.如果代价评估得当,快速但容量较小的缓存可以帮助降低内存访问成本.

代价模型是查询代价估算的重点,同时也是物理查询优化的依据.目前除了磁盘数据库之外,其他类型的数据库还没有一个成熟的代价评估模型.内存数据库主要的查询处理代价发生在处理器中.和内存数据库一样,GPU 数据库也是需要将数据从磁盘中尽可能多地调往内存和 GPU,目的是消除磁盘的传输瓶颈.因此,GPU 数据库中存储层次体系结构包括磁盘-内存-GPU 的三级存储结构.除此之外,还包括主机内存和缓存的管理.因此 GPU 数据库在做代价估计的时候,也应该考虑这些因素.

GPU 数据库的代价不仅包括 CPU 计算,还包括 GPU 计算.因此,传统的估算方法已经明显不再适用.由于 GPU 的计算能力比 CPU 高出了至少 10 倍以上,而且必须考虑要估算的元组在 CPU 上还是 GPU 上,GPU 数据库应该对 CPU 和 GPU 分别进行考虑.GPU 数据库还面临 Host Memory 和 GPU 显存之间的数据拷贝,这点和分布式数据库中的网络通信开销类似.

基于学习的代价评估方法如表 13 所示.基于学习的代价评估方法主要是通过机器学习或者是其他学习类算法,通过对现有的数据进行学习,得出代价评估模型或代价评估函数.

表 13　代价评估模型对比

类型	方法名称	文献	时间	特点描述
基于学习的方法做代价评估	粗粒度预测建模方法	[109]	2012	不同粒度下查询执行预测建模技术
	学习型优化器代价评估	[9]	2014	硬件无关的处理引擎和学习型查询优化器用于决策(HyPE 系统)

Akdere 等人[109]提出并评估了在不同粒度下查询执行预测建模技术.从粗粒度的计划级模型到细粒度的操作级模型,他们提出的方法在静态工作负载查询的高准确性和动态工作负载中查询不可预见性之间进行了权衡.除此之外,他们引入一种混合方法,通过在查询准确性预测过程中有选择地组合它们来发挥各自的优势,他们通过实验证明了基于学习的查询准确性预测建模对于静态和动态工作负载场景都是可行且有效的.

Breß 等人[9]介绍了代价函数和几种启发式方法,以便在所有可用的设备上有效地放置操作.他们在文章中展示了 HyPE 系统,该系统集成了一个硬件无关的处理引擎和一个学习型查询优化器用于进行决策,从而形成了一个专门针对异构硬件环境的高度自适应数据库系统.

GPU 数据库中的查询优化技术,在吸收和借鉴磁盘数据库优化技术的同时,又将 GPU 的并行性结合起来,同时引入了一些人工智能相关的方法.这些方法在提高优化效率的同时,大量使用 GPU 的硬件特性,在某种程度上实现了硬件技术和数据库技术的"互补".这部分的研究是目前数据库优化技术研究的热点,在未来也值得持续研究和关注.

5.5　GPU 数据库事务处理和数据一致性

事务作为数据库系统的一个核心功能,在目前的磁盘数据库系统上均有实现.GPU 数据库系统对事务的支持很少.许多 GPU 数据库系统还没有解决并发事务处理的问题.

在数据一致性方面,为了保持 CPU 和 GPU 之间的数据一致性,有三种可行的方法:(1)将每次要处理的数据保存在一个分区,这种情况可以不考虑任何的数据复制管理机制,但是需要保证数据修改后的重新分区问题;(2)使用已经建立的复制机制,如读取一次写入多个副本,或者是对主要操作的数据进行拷贝;(3)始终在一个设备上(如 CPU)更新处理,定期将这些更改同步到其他设备中(如 GPU).

He 等人[4]的研究表明,基于锁的一致性策略显著破坏 GPU 的性能.为了在 GPU 数据库上实现事务相关的功能,他们提出了一个 GPU 数据库处理事务的执行引擎 GPUTx,从理论上讨论了实现并发处理事务的可能.他们开发了一种无锁协议来确保 GPU 上序列化的并行事务冲突.他们为事务赋予一个全局唯一的时间戳,形成事务依赖图 T-dependency.该事务依赖图是无环图,通过简单的拓扑排序形成无冲突的事务集合.他们在 GPUTx 上实验了 3 种并发控制策略:即两阶段锁、单分区事务和无冲突事务集 K-set.他们在 GPU 上实现了以数据为中心的操作排序.通过在 GPU 上执行该事务处理模式,得到了比 CPU 算法高 4 到 10 倍的吞吐量.但是,他们假设 GPUTx 在处理事务时在同一时刻完成,这种场景要求严苛并且在实践中面临事务的读写冲突,无法支持并发读写冲突和长事务的

执行等问题.

除了 GPUTx 之外,大多数系统都假设 GPU 不能有效支持 OLTP,这是因为工作负载由大量的短事务组成,潜在的降低了 GPU 在系统中的作用. Arefyeva 等人[110]研究了 GPU 支持高效 OLTP 所需的条件:(1)较高的请求到达率对大量的细粒度操作来说,几乎不需要等待时间;(2)请求到达率适中,但可以将每个请求分解为足够数量的并行操作. 只要满足上述两个条件,GPU 也能够支持高效的 OLTP 查询. 他们还开发了 GPU 加速 OLTP 系统的原型. 该原型系统能支持逐行和逐列的操作,具有并发控制和有界的旧一致性读取,他们将进一步开发他们的系统以处理更复杂的 OLTP 工作负载.

GPU 数据库的并发和事务处理目前仍然是一个未被完全解决的问题. 需要在 GPU 的并发控制、故障恢复机制、乐观锁控制以及日志恢复等方面深入探索,存在较大的研究空间.

6 典型 GPU 数据库系统

GPU 数据库技术的发展离不开内存数据库 MonetDB. 在 MonetDB 的基础上,经过不断地技术迭代,形成了经典的 GPU 数据库 Omnisci. 因此,对 Omnisci 技术的追溯要从 MonetDB 开始介绍. MonetDB 的主要技术亮点有内存列存储、SIMD 优化、cache-conscious 缓存优化技术、Radix hash cache 优化技术、BAT Algebra 列代数等. Vectorwise 在 MonetDB 内存列处理的基础上,增加了向量化处理. Vectorwise 是 MonetDB 的商业版本,它的代表性技术主要有向量化处理(Vectorized Processing)模型,面向 L1 cache 的优化技术,它衍生出来的系统有 ActianX, VectorH 等系统,支持 HTAP 和 Hadoop. Hyper 在向量化处理模型的基础上,增加了 JIT 即时编译技术,这是面向寄存器级优化设计. Omnisci 将 hyper 采用的 JIT 编译技术也实现在自己的系统上. 除此之外,Omnisci 将 JIT 即时编译技术和向量化处理技术相结合,支持 CPU 和 CPU-GPU 处理模式,形成了一个经典的 GPU 数据库技术栈,具有很强的代表性.

本节将对一些经典的 GPU 数据库系统如 PG-Strom、CoGaDB、Omnisci 等做介绍和分析,包括功能特性和非功能特性. 其中功能特性包括存储系统、存储模型、处理模型、缓冲区管理、查询处理和优化、一致性和事务处理等方面;非功能性属性包括性能和可移植性等.

6.1 PG-Strom

(1)系统架构. PG-Strom 是 PostgreSQL[111]第 11 版或者更高版本设计的扩展模块,其核心模块是 GPU 代码生成器和异步并行执行引擎,在 GPU 设备上运行 SQL 工作负载. PG-Strom 的一个特征是 GPUDirect SQL,它绕过 CPU/RAM,将数据直接读取到 GPU,GPU 上 SQL 处理可以最大化这些设备带宽.

PG-Strom 结构如图 10 所示. 首先,SQL 解析器将提供的 SQL 查询分解为解析树,然后 GPU 代码生成器根据 SQL 命令和异步并行执行引擎自动生成 GPU 程序.

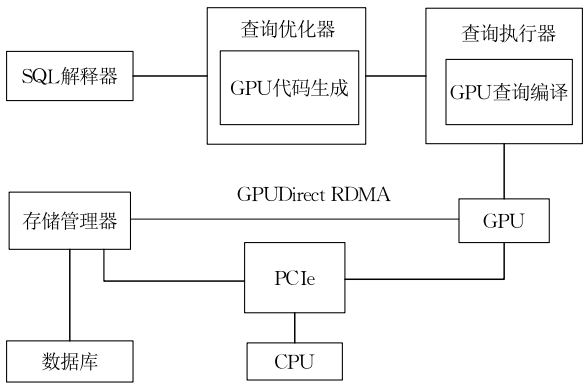

图 10 PG-Strom 系统结构图[3]

(2)存储管理. PG-Strom 使用 PostgreSQL 作为其基本的数据库系统,因此可以说它是一个基于磁盘的系统. 然而,它不同于其他的磁盘数据库系统. 相反,它利用了 SSD 带来的性能优势. 尽管 PCIe 总线传输代价巨大,而且被认为是 GPU 执行中的主要开销. 但是 PG-Strom 有一个独特的功能,即 SSD 到 GPU 直接 SQL 执行,可以绕过 PCIe 传输,在一定的程度上能提高性能. PG-Strom 使用行存储模型,因为它使用了 PostgreSQL. 然而,最新版本的 PG-Strom 支持将数据在 GPU 存储器上从行存储转化为列存储. 使用列存储可以显著减少数据量,从而在整个存储层次结构中高效利用带宽.

(3)缓冲区管理. PG-Strom 使用 PostgreSQL 堆缓冲区,在需要数据时向 GPU 提供所需的数据,这个缓冲区控制共享内存和持久化存储之间的数据传输,使得执行非常高效. PG-Strom 还支持 GPU 设备的按需分页,因此,它只加载执行过程中所需要的页面.

(4)查询处理. PG-Strom 根据操作的要求,使用所有可用的处理器来增强查询处理性能,它实现

了一个面向批处理、基于运算符的查询执行方案. 在这种方案下，表数据被分割为大块，并从一个操作推送到另一个操作来处理.

(5) 查询优化. PG-Strom 查询优化器与 PostgreSQL 的查询执行计划器相协作，如果 CPU 上执行计划的估计成本更高，它将为 GPU 提供代查询执行计划. 然而，它要求使用运算符、函数、和数据类型必须得到 PG-Strom 的支持，才能在 GPU 上执行.

(6) 事务处理和一致性. 主流的磁盘数据库几乎都能支持事务相关的操作，事务作为 OLTP 操作的精华，被完整地实现和应用. 遗憾的是，GPU 数据库并没有很好地解决事务和并发操作相关的问题.

PG-Strom 依赖于 PostgreSQL 事务策略，因此当多个事务操作同时修改系统状态时，它支持并发实现，并充分支持系统的约束和属性. 并发性问题可以被完整的处理，因为 PostgreSQL 是完全 ACID 兼容并实现事务隔离.

(7) 可移植性与性能对比. PG-Strom 利用 SSD 到 GPU 直接 SQL 执行，使得平均执行速度提高 3.5 倍. PG-Strom 允许直接将数据加载到 GPU，绕过 PCIe 总线. PG-Strom 允许在 GPU 上实现并行程序. 因此，PG-Strom 是硬件不敏感的数据库，可移植性好.

6.2　CoGaDB

(1) 系统架构. CoGaDB[5] 是一个面向列的协处理器加速数据库系统，它提供了一个高性能的 OLAP 查询引擎，它可以有效地利用 CPU、GPU 或 MIC(Xeon Phi)进行查询处理. CoGaDB 使用并行查询处理器和混合查询处理引擎(HyPE)[102] 实现了一种专门的成本建模算法. 用于选择最高效的异构处理器环境. 它还使用一个名为 Hawk[112] 的定制查询编译器为不同的处理器生成专门的代码. 这些新生成的专门代码在执行的过程中对工作负载和数据集充分利用.

CoGaDB 由 Breß 等人开发，它是一个模块化的体系结构. 如图 11 所示. 与大多数的 DBMS 一样，CoGaDB 提供了 SQL 前端接口，可用于编写混合 SQL 语言查询并启动它，目的是从数据库中获取结果. 然后 CoGaDB 的逻辑优化器应用预定义的优化策略生成一个新的查询计划. 然后提供给 HyPE 进行处理. 混合查询优化器根据逻辑查询计划创建优化的物理查询计划，然后将该计划传递给 Hawk 以生成机器代码.

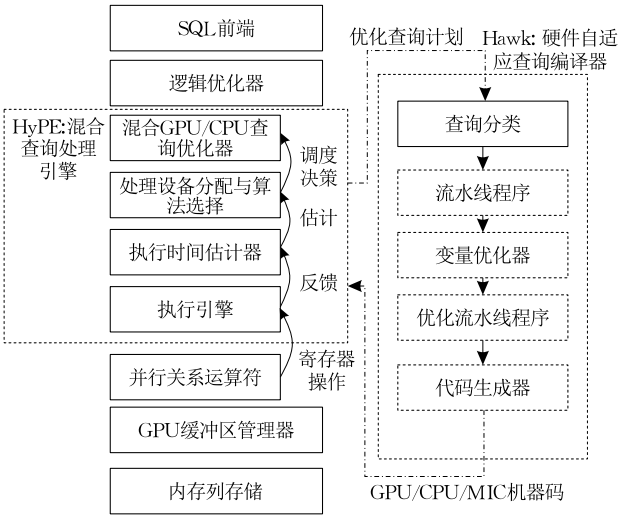

图 11　CoGaDB 系统结构图[5]

(2) 存储管理. CoGaDB 使用主存作为存储系统，它可以高效地将数据从主存传输到 GPU，从而极大地提高 GPU 的处理速度. CoGaDB 需要一个高效的存储模型，以满足启动期间在主存中加载整个数据库的要求. 它使用列存储作为存储模型，因为更高的压缩率可以节省大量内存，并且由于列的存储性质，可以更有效地存储数据.

(3) 缓冲区管理. CoGaDB 使用专用的 GPU 缓冲区管理器模块处理此需求. GPU 缓冲区管理器负责在 GPU 操作请求输入列时候向他们提供处理所需要的数据.

(4) 查询处理. CoGaDB 的设计目标是根据操作使用所有可用的处理器来改进查询处理和性能. 因此，CoGaDB 通过基于操作的调度实现了一次一个操作符的批量处理模型，该调度将查询集分布在所有可用的处理资源上，有效利用内存层次结构.

(5) 查询优化. CoGaDB 通过构建定制的解决方案-混合查询处理引擎(HyPE)，解决了查询处理和优化问题. 该模块通过应用多种优化算法来优化物理查询计划，并利用 Hawk 引擎为目标处理设备生成高度优化的机器代码.

(6) 事务处理和一致性. CoGaDB 没有实现任何明确的一致性标准，并且不支持事务. 它提供了一些容错机制以支持系统的稳定性，最终实现了一致性的数据存储.

(7) 可移植性与性能对比. CoGaDB 采用 SSB[104] 做性能评估. 通过和 MonetDB[113] 做性能对比发现，单纯使用 CPU 的 CoGaDB 耗时最多，使用 GPU 进行加速之后，执行时间明显变短. 除此之外，CoGaDB 试图通过在系统中实现所有特定于硬件的操作来实

现硬件无关的设计.这种策略会导致较高的开发和实施成本,但是提供了良好的加速比.

6.3 Omnisci

(1)系统架构.Omnisci起源于麻省理工学院计算机科学和人工智能实验室Todd Mostak的研究.它最初的名字叫MapD(大规模并行数据库).它是作为一个数据处理和可视化引擎开发的,它将DBMS的传统查询处理能力与高级分析和可视化功能相结合,利用GPU强大的计算能力来进行数据的处理和分析,2018年MapD[8]被重新命名为Omnisci.

Omnisci架构如图12所示.Omnisci采用了内存存储结构,利用SSD进行持久化的存储.通过基于JIT的LLVM编译器将SQL查询编译成机器码.Omnisci在设计时充分利用GPU的特性,能快速执行SQL语句,实现了常用的SQL分析操作,如过滤、分组和连接等操作.Omnisci的架构实现了一个类似金字塔模型,虽然每个连续的内存级计算较慢,但是容量比上一级大.Omnisci通过将DBMS的传统查询处理能力与高级分析功能相结合,成为该领域内最具代表性和性能优势的数据库系统.

(2)存储管理.Omnisci在GPU内存中以列的方式存储数据,利用SSD进行持久化存储.Omnisci将列划分为块,块作为Omnisci内存管理器的基本单元,它的基本处理模型是一次一操作符.由于将数据划分成块,因此,也可以在每个块的基础上进行处理.Omnisci通过将大部分数据保存在内存中以避免磁盘访问.与其他将数据存储在CPU中以便在查询期间传输到GPU的系统相比,Omnisci core缓存了高达512GB的最近访问的数据,这种方式能有效避免低下的传输效率.

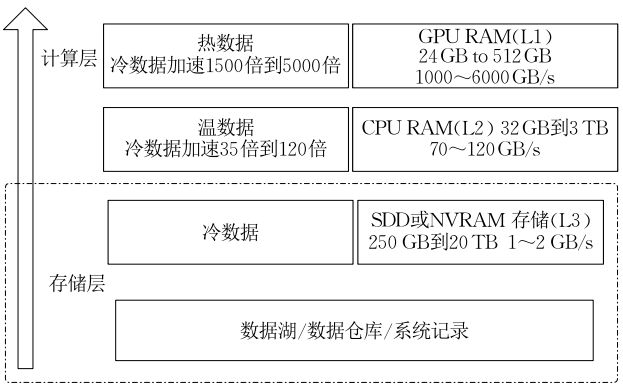

图12 Omnisci系统结构图[8]

(3)缓冲区管理.Omnisci在执行查询时,如果GPU RAM可用,则Omnisci core数据库优化器首先使用它.通常可以将GPU RAM视为是CPU上的L1缓存.Omnisci数据库首先尝试缓存热数据,如果GPU RAM不可用或者是已满,Omnisci数据库优化器将使用CPU RAM(L2),如果L1和L2都被填满,查询记录溢出到磁盘(L3).

(4)查询处理.Omnisci使用基于LLVM(低级虚拟接)的JIT(即时)编译框架,允许Omnisci将查询计划转化为与体系结构无关的中间代码表示(LLVM IR).Omnisci将查询编译为CPU和GPU的可执行代码来处理查询.Omnisci的基本处理模型是一次处理一个操作符.块是Omnisci内存管理器的基本单元.由于将数据划分成为块,因此可以用块为基础处理数据.Omnisci能够应用面向块的处理,它还通过向量化提高了性能.

(5)查询优化.Omnisci使用流机制进行数据处理.优化器尝试将查询计划分成几个部分,并在最合适的处理设备上处理每个部分,在一个节点内,Omnisci在GPU之间采用无共享的架构.在查询处理上,每个GPU进程独立处理来自其他GPU的数据.

(6)事务处理和一致性.Omnisci不支持事务处理功能.Omnisci以即时性和强一致性的方式保证分布式系统的一致性.

(7)可移植性与性能对比.由于Omnisci是硬件相关的,所以可移植性较差.Omnisci是当前列出的几种数据库中支持OLAP和OLTP的,它在开源GPU数据库中综合性能最强.

6.4 技术分类

对GPU数据库系统设计和优化,最需要考虑的功能特性主要有存储特征、查询处理模型和可移植性(硬件依赖性)等方面.在存储功能中,存储系统、存储模型、缓冲区管理等部分作为存储功能的核心,需要数据库开发和设计者们额外关注.

表14列出了一些经典GPU数据库的功能特性.CoGaDB、GPUTx、Ocelot、Virginian、GPUDB等系统使用主存作为底层存储系统.Omnisci支持基于主存和磁盘两种存储系统.从存储模型来划分,文中列出的所有数据库系统均支持列存储,Brytlyt和PG-Strom支持行存储和列存储两种方式.PG-Strom使用PostgreSQL作为它们的数据库引擎.PG-Strom支持SSD,能够将查询所需的数据从SSD传输到GPU,利用GPU来执行查询.virginia属于早期的GPU数据库系统,缺少目前主流数据库的大多数功能.上述数据库系统都实现了数据放置策略,以最优化方式将数据从CPU移动到GPU.所有这些策略都是针对系统及其体系结构设计的定制策略.CoGaDB、Omnisci和PG-Strom实现了一个专用模块,用于支持有效的缓冲区管理.

表 14　典型 GPU 数据库存储特性表

GPU 数据库系统	存储系统		存储模型		存储位置		存储模型特征		缓冲区管理	
	基于主存	基于磁盘	列存储	行存储	GPU 显存	SSD 硬盘	模型压缩	分区	专用缓存	自定义
CoGaDB[5]	√		√						√	
GPUDB[6]	√		√				√			√
GPUQP[3]	√	√	√							
GPUTx[4]	√		√							
Omnisci[8]	√	√	√		√	√		√	√	
Ocelot[9]	√		√							
Virginian[12]	√		√							√
Brytlyt[114]		√	√	√						√
SQream[115]		√	√					√		√
PG-Strom[24]		√	√	√		√			√	

在 GPU 数据库系统中，一次一操作符或一次一个数据块处理模型可以使 GPU 发挥其并行性. 典型 GPU 数据库系统查询处理方式和查询处理设备支持如表 15 所示.

表 15　典型 GPU 数据库查询处理技术表

GPU 数据库系统	查询处理方式		查询处理设备支持	
	一次/操作	一次/块	单设备	跨设备
CoGaDB	√		√	√
GPUDB	√	√	√	
GPUQP	√		√	√
GPUTx		√	√	
Omnisci	√	√	√	√
Ocelot	√		√	
Virginian	√	√	√	
Brytlyt	√		√	
SQream	√	√	√	
PG-Strom	√	√	√	

CoGaDB、GPUQP、Brytlyt、Ocelot 这四种数据库仅支持一次一操作处理. 文中列出的其他几种数据库均支持一次一操作和一次一块两种查询处理方式. 在查询设备支持方面，只有 CoGaDB、GPUQP 和 Omnisci 支持跨 GPU 设备处理模型，其他系统都只能在单个设备上运行.

常见的 GPU 数据库事务支持情况如表 16 所示[35]. 从表 16 可以看出，除了 GPUTx 和 PG-Strom 之外，其他的 GPU 数据库均不支持事务[17]. GPUTx 将数据严格保存在 GPU 的 RAM 中，只需要将传入的事务传输到 GPU，并将结果传输回 CPU. PG-Strom 作为 PostgreSQL[111] 的 GPU 扩展版，在处理事务相关的工作时，能直接调用 PostgreSQL 事务处理模块.

硬件对几种经典的 GPU 数据库的影响如表 16 所示. 从表 16 可以看出，只有 Ocelot 是可移植的、硬件无关的数据库系统[35]. 其他几种系统在可移植性方面还存在很大的改进和提升空间.

表 16　GPU 数据库事务与硬件特性表

GPU 数据库系统	事务支持	可移植性	
		硬件敏感	硬件不敏感
CoGaDB		√	
GPUDB		√	
GPUQP		√	
GPUTx	√	√	
Omnisci		√	
Ocelot			√
Virginian		√	
Brytlyt		√	
SQream		√	
PG-Strom	√	√	

几种 GPU 数据库的性能对比如表 17 所示. CoGaDB 使用硬件无关的查询优化器 HyPE，进行并发查询时资源的优化利用. HyPE 由三个组件组成，分别是估计组件、算法选择器和混合查询优化器. HyPE 的混合查询优化器为查询计划中的每个操作符分配合适的目标处理器和算法. 使用 SSB 作为基准测试集，和 CPU-only 的模式相比，CoGaDB 在 CPU-GPU 模式下运行速度提高了 34 倍.

表 17　GPU 数据库性能评估表

GPU 数据库系统	Benchmark	加速比/倍
CoGaDB	SSB	34
GPUDB	SSB	10
Virginian	OLAP	17～35
Brytlyt	TPC-H	300
SQream	85 TB 测试数据集	100
PG-Strom	PG 在 CPU 上测测试为基准	3.5

GPUDB 使用两种方式来加速查询：(1) 将数据固定在 GPU 内存；(2) 将数据放置在空闲分页内存中. 当数据在固定内存中可用时，可以利用数据压缩技术和传输重叠技术来加速 GPU 上的查询执行，当数据在可分页内存中可用时，只使用数据压缩技术来加速 GPU 上的查询执行. 在所有的 SSB 查询中，该数据库在 GPU 的性能都优于在 CPU 上的性能，运行速度提高了 10 倍.

Virginian 在 GPU 上使用映射执行的方式，假设数据和结果可以驻留在 GPU 上，将主存映射到 GPU 设备上，用来实现更快的数据访问和结果写入. 和没有使用主存映射的方式相比，这种方法能将数据传输的时间减少，达到 17～35 倍的加速比.

Brytlyt 支持直接在 GPU 上直接进行连接操作，数据被分割成块并分发给使用水平分区进行搜索的 GPU 内核. brytlyt 由于其体系结构和 SQL 操作的专门执行，比传统的数据库获得了巨大的性能改进，在独立的基准测试中，它的运行速度可以提高 300 倍左右.

SQream 使用 Apache Phoenix 关系处理引擎来进行数据处理，尤其擅长处理大型数据集，在 85Tb 的数据集上进行测试，其他关系数据库需要 5 个小时才能完成，SQream 将查询时间缩短到 5 分钟，运行速度提高了 100 倍.

PG-Strom 采用 SSD 到 GPU 的直接 SQL 执行，允许数据直接加载到 GPU 而绕过 PCIe，利用这一优势，可以充分利用 GPU 的并行计算能力，将平均处理速度提高了 3.5 倍.

通过对上述 GPU 数据库的分析，可以得出结论，所有这些系统都比 CPU 提供更好的性能. 有些系统(如 brylyt 和 SQream)在 OLAP 和 OLTP 工作负载上提供了 100～300 倍的性能. 其他系统也提供了类似的加速效果，这清楚地说明了 GPU 加速的数据库系统在计算领域的巨大实用价值.

7 挑战与展望

虽然 GPU 数据库在近些年取得了长足的发展，但依然面临如下问题，有待进一步研究.

(1) 缓解数据传输瓶颈

由于 GPU 的数据处理速度远高于 PCIe 总线的数据传输速度，导致数据传输成为瓶颈. 解决这一问题有两种途径，一种对数据进行压缩，使得单位时间内传输的数据尽可能地多；另一种是使尽可能减少需要通过 PCIe 传输的数据. 对于第一种解决方案而言，需要研究和应用各种轻量级压缩算法以及在压缩数据上直接计算的方法，这种方案不仅可以在一定程度上缓解 PCIe 的传输瓶颈问题，而且可以缓解 GPU 显存容量的问题. 对第二种方案而言，为了尽可能地减少数据传输，需要研究如何对数据和任务进行合理划分和分配，以尽可能局部化数据访问.

(2) 数据库复杂算子的优化

在数据库基本算子中，连接、分组聚集操作与选择、过滤、投影等操作相比，占用了大量的执行时间. 虽然 GPU 连接算法和分组聚焦算法已经有大量的研究，但目前海量数据下的 GPU 连接算法性能仍有待提高，特别是多表连接，如何减少多表连接操作中中间结果重分布的开销是一个难点，如何设计更适合 GPU 特点的多表连接算法、优化多表连接顺序的选择、连接和分组聚集作为一个整体进行优化等问题都值得进一步地探索.

(3) CPU-GPU 协同架构下的任务调度与执行

CPU 在程序控制、低延迟时序处理、执行有序性等方面具有独特的优势，缺点是计算单元较少，不具备大规模并行计算能力，GPU 具有大量的并行计算单元，能进行大量高并发计算，但是在逻辑控制方面不如 CPU. CPU-GPU 协同处理可以充分利用两种处理器各自的优势，但由于 PCIe 的数据传输瓶颈没有突破，现有研究通常假设数据全部放在 GPU 显存中，将计算和控制全部交给 GPU 来做，并没有将 CPU 的控制优势和 GPU 的计算优势发挥出来. 因此，协同架构下 CPU 和 GPU 之间如何划分数据、如何划分任务、如何调度任务、如何控制查询执行，这些问题目前仍没有解决.

(4) GPU 数据库查询优化

GPU 数据库的查询优化需要考虑更多的因素，因而更复杂. 现在研究主要聚焦在查询算法上，对查询优化关注不多，尚无被普遍接受的代价模型. HyPE 查询处理引擎利用机器学习技术解决查询计划中的代价评估，但是 HyPE 框架提出的方案容易陷入局部最优，还可能会造成不必要的数据传输. 如何完善 GPU 数据库查询计划的代价模型，如何利用深度学习技术对数据分布、查询负载、性能表现等特征进行建模和学习，自动地进行查询负载预测、数据划分、查询优化和调度，都是值得深入探索的.

(5) 基于 NVLink 的算法研究

NVIDIA 开发了 NVLink 技术，尝试从硬件层面上缓解 PCIe 的传输瓶颈问题. 虽然目前 PCIe 仍然是主流平台的标配，但是 NVLink 凭借自身的优势，未来将会有广泛的应用. 用 NVLink 替代 PCIe 之后，以减少数据传输为优化目标的查询优化策略将面临重构，数据划分、任务划分、任务调度、查询算法等也需要重新设计. 因此，基于 NVLink 的查询处理有很大的研究价值.

8 总　结

GPU 数据库作为新硬件数据库中最重要的一个，随着 GPU 的发展而快速发展. GPU 数据库在发展的过程中，GPU 加速型和 GPU 内存型两种技术路线并存，两种技术路线的代表性产品也都在快速发展. 不管是哪种 GPU 数据库，都具有存储管理、查询编译、查询处理、查询优化、查询执行等功能. 本文从上述几个功能作为切入口，分析了 GPU 数据库中的关键技术，综述了这些技术对 GPU 数据库的影响. 文章最后对 GPU 数据库现有的问题进行了总结，对未来的研究方向进行了展望.

参　考　文　献

[1] Sun C, Agrawal D, El Abbadi A. Hardware acceleration for spatial selections and joins//Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. San Diego, USA, 2003: 455-466

[2] Pei Wei, Li Zhan-Huai, Pan Wei. Survey of key technologies in GPU database system. Journal of Software, 2021, 32(3): 859-885(in Chinese)
(裴威，李战怀，潘巍. GPU 数据库核心技术综述. 软件学报，2021，32(3)：859-885)

[3] He B, Lu M, Yang K, et al. Relational query coprocessing on graphics processors. ACM Transactions on Database Systems, 2009, 34(4): 1-39

[4] He B, Yu J X. High-throughput transaction executions on graphics processors. arXiv preprint arXiv:1103.3105, 2011

[5] Breß S. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. Datenbank-Spektrum, 2014, 14: 199-209

[6] Yuan Y, Lee R, Zhang X. The Yin and Yang of processing data warehousing queries on GPU devices. Proceedings of the VLDB Endowment, 2013, 6(10): 817-828

[7] Mostak T. MapD has rebranded to Omnisci. https://www.Omnisci.com/blog/mapd-has-rebranded-to-Omnisci/, Sept., 2018

[8] Mostak T. An Overview of MapD(Massively Parallel Database). White Paper, Massachusetts Institute of Technology, USA, 2013

[9] Breß S, Köcher B, Heimel M, et al. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. Proceedings of the VLDB Endowment, 2014, 7(13): 1609-1612

[10] Yong K K, Ho W K, Chua M W, et al. A GPU query accelerator for geospatial coordinates computation//Proceedings of the 2015 International Conference on Cloud Computing Research and Innovation(ICCCRI). Singapore, 2015: 166-172

[11] Lee R, Zhou M, Li C, et al. The art of balance: A RateupDB™ experience of building a CPU/GPU hybrid database product. Proceedings of the VLDB Endowment, 2021, 14(12): 2999-3013

[12] Bakkum P, Chakradhar S. Efficient data management for GPU databases. High Performance Computing on Graphics Processing Units, 2012

[13] Mondal S, Maji R K, Ghosh Z, et al. ParStream-seq: An improved method of handling next generation sequence data. Genomics, 2019, 111(6): 1641-1650

[14] Li H, Tu Y C, Zeng B. Concurrent query processing in a GPU-based database system. PLoS One, 2019, 14(4): e0214720

[15] Cao J, Sen R, Interlandi M, et al. Revisiting query performance in GPU database systems. arXiv preprint arXiv:2302.00734, 2023

[16] Ferreira I, Beisken S, Lueftinger L, et al. Species identification and antibiotic resistance prediction by analysis of whole-genome sequence data by use of ARESdb: An analysis of isolates from the Unyvero lower respiratory tract infection trial. Journal of Clinical Microbiology, 2020, 58(7): e00273-20

[17] Cao Z, Dong S, Vemuri S, et al. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook //Proceedings of the 18th USENIX Conference on File and Storage Technologies(FAST 20). Santa Clara, USA, 2020: 209-223

[18] Aji A, Teodoro G, Wang F. Haggis: Turbocharge a MapReduce based spatial data warehousing system with GPU engine// Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. Dallas, USA, 2014: 15-20

[19] Liang Y, Vo H, Aji A, et al. Scalable 3D spatial queries for analytical pathology imaging with MapReduce//Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. Burlingame, USA, 2016: 1-4

[20] Foley D, Danskin J. Ultra-performance pascal GPU and NVLink interconnect. IEEE Micro, 2017, 37(2): 7-17

[21] Yuan Y, Salmi M F, Huai Y, et al. Spark-GPU: An accelerated in-memory data processing engine on clusters//Proceedings of the 2016 IEEE International Conference on Big Data(Big Data). Washington, USA, 2016: 273-283

[22] Zhang Yan-Song, Han Rui-Chen, Liu Zhuan, Zhang Yu. An in-memory database implementation technique based on separation of management, computation and storage. Chinese Journal of Computers, 2023, 46(4): 761-779(in Chinese)
(张延松，韩瑞琛，刘专，张宇. 一种基于管算存分离的内存数据库实现技术. 计算机学报，2023，46(4)：761-779)

[23] Zhang Y, Zhang Y, Lu J, et al. One size does not fit all: Accelerating OLAP workloads with GPUs. Distributed and Parallel Databases, 2020, 38: 995-1037

[24] Kohei KaiGai. PG-Strom v2.0 Technical Brief. https://www.slideshare.net/kaigai/pgstrom-v20-technical-brief-17apr2018, Apr., 2018

[25] Peltenburg J, van Straten J, Brobbel M, et al. Supporting columnar in-memory formats on FPGA: The hardware design of Fletcher for apache arrow//Proceedings of the 15th International Symposium on Applied Reconfigurable Computing (ARC 2019). Darmstadt, Germany, 2019: 32-47

[26] Wang J, Yi X, Guo R, et al. Milvus: A purpose-built vector data management system//Proceedings of the 2021 International Conference on Management of Data. Virtual, China, 2021: 2614-2627

[27] Li C, Sun Y, Jin L, et al. Priority-based PCIe scheduling for multi-tenant multi-GPU systems. IEEE Computer Architecture Letters, 2019, 18(2): 157-160

[28] Raza S M A, Chrysogelos P, Sioulas P, et al. GPU-accelerated data management under the test of time//Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR). Amsterdam, the Netherlands, 2020

[29] Li A, Song S L, Chen J, et al. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPU-Direct. IEEE Transactions on Parallel and Distributed Systems, 2019, 31(1): 94-110

[30] Lutz C, Breß S, Zeuch S, et al. Pump up the volume: Processing large data on GPUs with fast interconnects//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, USA, 2020: 1633-1649

[31] Potluri S, Hamidouche K, Venkatesh A, et al. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs//Proceedings of the 2013 42nd International Conference on Parallel Processing. Lyon, France, 2013: 80-89

[32] Guo C, Chen H, Zhang F, et al. Distributed join algorithms on multi-CPU clusters with GPUDirect RDMA//Proceedings of the 48th International Conference on Parallel Processing. Kyoto, Japan, 2019: 1-10

[33] Das Sharma D, Blankenship R, Berger D S. An introduction to the Compute Express Link (CXL) interconnect. arXiv preprint arXiv:2306.11227, 2023

[34] Ghodsnia P. An in-GPU-memory column-oriented database for processing analytical workloads//The VLDB PhD Workshop. VLDB Endowment. Istanbul, Turkey, 2012, 1

[35] Breß S, Heimel M, Siegmund N, et al. GPU-accelerated database systems: Survey and open challenges. Transactions on Large-Scale Data-and Knowledge-Centered Systems XV, 2014, 15: 1-35

[36] Grund M, Krüger J, Plattner H, et al. Hyrise: A main memory hybrid storage engine. Proceedings of the VLDB Endowment, 2010, 4(2): 105-116

[37] Alagiannis I, Idreos S, Ailamaki A. H2O: A hands-free adaptive store//Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. Snowbird, USA, 2014: 1103-1114

[38] Lemire D, Boytsov L. Decoding billions of integers per second through vectorization. Software: Practice and Experience, 2015, 45(1): 1-29

[39] Müller I, Ratsch C, Faerber F. Adaptive string dictionary compression in in-memory column-store database systems//Proceedings of the International Conference on Extending Database Technology. Athens, Greece, 2014, 14: 283-294

[40] Liu C, Umbenhower M K, Jiang H, et al. Mostly order preserving dictionaries//Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE). Macao, China, 2019: 1214-1225

[41] Wang J, Lin C, Papakonstantinou Y, et al. An experimental study of bitmap compression vs. inverted list compression//Proceedings of the 2017 ACM International Conference on Management of Data. Chicago, USA, 2017: 993-1008

[42] Feng Z, Lo E, Kao B, et al. ByteSlice: Pushing the envelop of main memory data processing with a new storage layout//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne, Australia, 2015: 31-46

[43] Li J, Tseng H W, Lin C, et al. HippogriffDB: Balancing I/O and GPU bandwidth in big data analytics. Proceedings of the VLDB Endowment, 2016, 9(14): 1647-1658

[44] Mallia A, Siedlaczek M, Suel T, et al. GPU-accelerated decoding of integer lists//Proceedings of the 28th ACM International Conference on Information and Knowledge Management. Beijing, China, 2019: 2193-2196

[45] Shanbhag A, Yogatama B W, Yu X, et al. Tile-based lightweight integer compression in GPU//Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 1390-1403

[46] Vijaykumar N, Pekhimenko G, Jog A, et al. A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. ACM SIGARCH Computer Architecture News, 2015, 43(3S): 41-53

[47] Lee S, Kim K, Koo G, et al. Warped-compression: Enabling power efficient GPUs through register compression. ACM SIGARCH Computer Architecture News, 2015, 43(3S): 502-514

[48] Ashkiani S, Li S, Farach-Colton M, et al. GPU LSM: A dynamic dictionary data structure for the GPU//Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Vancouver, Canada, 2018: 430-440

[49] Alam M, Yoginath S B, Perumalla K S. Performance of point and range queries for in-memory databases using Radix trees on GPUs//Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). Sydney, Australia, 2016: 1493-1500

[50] Awad M A, Ashkiani S, Johnson R, et al. Engineering a high-performance GPU B-tree//Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. New York, USA, 2019: 145-157

[51] Shahvarani A, Jacobsen H A. A Hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms//Proceedings of the 2016 International Conference on Management of Data. New York, USA, 2016: 1523-1538

[52] Zhang K, Wang K, 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

[53] Tran B, Schaffner B, Sawin J, et al. Increasing the efficiency of GPU Bitmap index query processing//Proceedings of the International Conference on Database Systems for Advanced Applications. Cham: Springer International Publishing, 2020: 339-355

[54] Sun F D, Wang L. Evaluation of B+-tree and CSB+-tree in main memory database. Applied Mechanics and Materials, 2014, 571: 580-585

[55] Li Y, Zhu Q, Lyu Z, et al. DyCuckoo: Dynamic hash tables on GPUs//Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE). Chania, Greece, 2021: 744-755

[56] Zhang Hong-Jun, Wu Yan-Jun, Zhang Heng, et al. Hybrid access cache indexing framework adapted to GPU. Journal of Software, 2020, 31(10): 3038-3055(in Chinese)
(张鸿骏，武延军，张珩等. 一种适应 GPU 的混合访问缓存索引框架. 软件学报，2020，31(10)：3038-3055)

[57] Rosenfeld V, Breß S, Markl V. Query processing on heterogeneous CPU/GPU systems. ACM Computing Surveys, 2022, 55(1): 1-38

[58] Tahboub R Y, Essertel G M, Rompf T. How to architect a query compiler, revisited//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 307-322

[59] Chrysogelos P, Karpathiotakis M, Appuswamy R, et al. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. Proceedings of the VLDB Endowment, 2019, 12(5): 544-556

[60] Paul J, He B, Lu S, et al. Improving execution efficiency of just-in-time compilation based query processing on GPUs. Proceedings of the VLDB Endowment, 2020, 14(2): 202-214

[61] Funke H, Teubner J. Data-parallel query processing on non-uniform data. Proceedings of the VLDB Endowment, 2020, 13(6): 884-897

[62] Lang H, Kipf A, Passing L, et al. Make the most out of your SIMD investments: Counter control flow divergence in compiled query pipelines//Proceedings of the 14th International Workshop on Data Management on New Hardware. Houston, USA, 2018: 1-8

[63] Viglas S D. Just-in-time compilation for SQL query processing //Proceedings of the 2014 IEEE 30th International Conference on Data Engineering. Chicago, USA, 2014: 1298-1301

[64] Gubner T, Tomé D, Lang H, et al. Fluid co-processing: GPU Bloom-filters for CPU joins//Proceedings of the 15th International Workshop on Data Management on New Hardware. Amsterdam, the Netherlands, 2019: 1-10

[65] Guo C, Chen H, Li C, et al. A memory access reduced sort on multi-core GPU//Proceedings of the 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). Exeter, UK, 2018: 586-593

[66] Schmid R F, Pisani F, Cáceres E N, et al. An evaluation of fast segmented sorting implementations on GPUs. Parallel Computing, 2022, 110: 102889

[67] Shanbhag A, Pirk H, Madden S. Efficient top-*k* query processing on massively parallel hardware//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 1557-1570

[68] Gurumurthy B, Broneske D, Schäler M, et al. An investigation of atomic synchronization for sort-based group-by aggregation on GPUs//Proceedings of the 2021 IEEE 37th International Conference on Data Engineering Workshops(ICDEW). Chania, Greece, 2021: 48-53

[69] Tomé D G, Gubner T, Raasveldt M, et al. Optimizing group-by and aggregation using GPU-CPU co-processing//ADMS@VLDB. Rio de Janeiro, Brazil, 2018: 1-10

[70] Karnagel T, Müller R, Lohman G M. Optimizing GPU-accelerated group-by and aggregation//ADMS@VLDB. Kohala Coast, USA, 2015: 13-24

[71] Meraji S, Keenleyside J, Kamath S, et al. Towards a combined grouping and aggregation algorithm for fast query processing in columnar databases with GPUs//Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. Hyderabad, India, 2015: 594-603

[72] Rosenfeld V, Breß S, Zeuch S, et al. Performance analysis and automatic tuning of hash aggregation on GPUs//Proceedings of the 15th International Workshop on Data Management on New Hardware. Amsterdam, the Netherlands, 2019: 1-11

[73] Nam Y M N, Han D H, Kim M S K. Sprinter: A fast n-ary join query processing method for complex OLAP queries//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, USA, 2020: 2055-2070

[74] Lai Z, Sun X, Luo Q, et al. Accelerating multi-way joins on the GPU. The VLDB Journal, 2022, 31(3): 529-553

[75] Wi S, Han W S, Chang C, et al. Towards multi-way join aware optimizer in SAP HANA. Proceedings of the VLDB Endowment, 2020, 13(12): 3019-3031

[76] Mageirakos V, Mancini R, Karthik S, et al. Efficient GPU-accelerated join optimization for complex queries//Proceedings

of the 2022 IEEE 38th International Conference on Data Engineering (ICDE). Kuala Lumpur, Malaysia, 2022: 3190-3193

[77] Gao H, Sakharnykh N. Scaling joins to a thousand GPUs// ADMS@VLDB. Copenhagen, Denmark, 2021: 55-64

[78] Rui R, Li H, Tu Y C. Efficient join algorithms for large database tables in a multi-GPU environment. Proceedings of the VLDB Endowment, 2020, 14(4): 708-720

[79] Mancini R, Karthik S, Chandra B, et al. Efficient massively parallel join optimization for large queries//Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 122-135

[80] Lutz C, Breß S, Zeuch S, et al. Triton join: Efficiently scaling to a large join state on GPUs with fast interconnects// Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 1017-1032

[81] Hu X X, Xi J Q, Tang D Y. Optimization for multi-join queries on the GPU. IEEE Access, 2020, 8: 118380-118395

[82] Guo C, Chen H. In-memory join algorithms on GPUs for large-data//Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). Zhangjiajie, China, 2019: 1060-1067

[83] Guo C, Chen H, Zhang F, et al. Parallel hybrid join algorithm on GPU//Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). Zhangjiajie, China, 2019: 1572-1579

[84] Sioulas P, Chrysogelos P, Karpathiotakis M, et al. Hardware-conscious hash-joins on GPUs//Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE). Macao, China, 2019: 698-709

[85] Nguyen A, Edahiro M, Kato S. GPU-accelerated VoltDB: A case for indexed nested loop join//Proceedings of the 2018 International Conference on High Performance Computing & Simulation (HPCS). Orleans, France, 2018: 204-212

[86] Rui R, Tu Y C. Fast equi-join algorithms on GPUs: Design and implementation//Proceedings of the 29th International Conference on Scientific and Statistical Database Management. Chicago, USA, 2017: 1-12

[87] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. arXiv preprint arXiv: 1307.1955, 2013

[88] Paul J, He B, Lu S, et al. Revisiting hash join on graphics processors: A decade later. Distributed and Parallel Databases, 2020, 38: 771-793

[89] Paul J, Lu S, He B, et al. MG-join: A scalable join for massively parallel multi-GPU architectures//Proceedings of the 2021 International Conference on Management of Data. Virtual Event, China, 2021: 1413-1425

[90] Yabuta M, Nguyen A, Kato S, et al. Relational joins on GPUs: A closer look. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(9): 2663-2673

[91] Birler A, Schmidt T, Fent P, et al. Simple, efficient, and robust hash tables for join processing//Proceedings of the 20th International Workshop on Data Management on New Hardware. Santiago, Chile, 2024: 1-9

[92] Zhang Y, Wang S, Lu J. Fusion OLAP: Fusing the pros of MOLAP and ROLAP together for in-memory OLAP. IEEE Transactions on Knowledge and Data Engineering, 2018, 31(9): 1722-1735

[93] Zhang Y, Zhou X, Zhang Y, et al. Virtual denormalization via array index reference for main memory OLAP. IEEE Transactions on Knowledge and Data Engineering, 2015, 28(4): 1061-1074

[94] Zhang Y, Zhang Y, Zhou X, et al. Main-memory foreign key joins on advanced processors: Design and re-evaluations for OLAP workloads. Distributed and Parallel Databases, 2019, 37: 469-506

[95] Zhu Y, Zhang Y, Zhou X, et al. A framework for OLAP in column-store database: One-pass join and pushing the materialization to the end//Web Technologies and Applications: 15th Asia-Pacific Web Conference (APWeb 2013). Sydney, Australia, 2013: 646-653

[96] Zhang Y, Zhang Y, Su M, et al. HG-Bitmap join index: A hybrid GPU/CPU bitmap join index mechanism for OLAP// Web Information Systems Engineering—WISE 2013 Workshops. Nanjing, China, 2014: 23-36

[97] Zhang Yu, Zhang Yan-Song, Chen Hong, et al. GPU adaptive hybrid OLAP query processing model. Journal of Software, 2016, 27(5): 1246-1265(in Chinese)
(张宇, 张延松, 陈红等. 一种适应 GPU 的混合 OLAP 查询处理模型. 软件学报, 2016, 27(5): 1246-1265)

[98] Shanbhag A, Madden S, Yu X. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, USA, 2020: 1617-1632

[99] Paul J, He J, He B. GPL: A GPU-based pipelined query processing engine//Proceedings of the 2016 International Conference on Management of Data. San Francisco, USA, 2016: 1935-1950

[100] Yang Z, Pan Q, Xu C. Fine-grained tuple transfer for pipelined query execution on CPU-GPU coprocessor//Proceedings of the International Conference on Database Systems for Advanced Applications, 2023: 19-34

[101] Chrysogelos P, Sioulas P, Ailamaki A. Hardware-conscious query processing in GPU-accelerated analytical engines// Proceedings of the 9th Biennial Conference on Innovative Data Systems Research. Asilomar, USA, 2019

[102] Breß S, Saake G. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. Proceedings of the VLDB Endowment, 2013, 6(12): 1398-1403

[103] Funke H, Breß S, Noll S, et al. Pipelined query processing in coprocessor environments//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 1603-1618

[104] O'Neil P, O'Neil E, Chen X, et al. The star schema benchmark and augmented fact table indexing//Performance Evaluation and Benchmarking: First TPC Technology Conference. Lyon, France, 2009: 237-252

[105] Cao J, Sen R, Interlandi M, et al. GPU database systems characterization and optimization. Proceedings of the VLDB Endowment, 2023, 17(3): 441-454

[106] Deng Y, Chen S, Hong Z, et al. How does software prefetching work on GPU query processing?//Proceedings of the 20th International Workshop on Data Management on New Hardware. Santiago, Chile, 2024: 1-9

[107] Meister A, Saake G. Challenges for a GPU-accelerated dynamic programming approach for join-order optimization//Proceedings of the Workshop Grundlagen von Datenbanken. Nörten Hardenberg, Germany, 2016: 86-91

[108] Schubert N L, Grulich P M, Zeuch S, et al. Exploiting access pattern characteristics for join reordering//Proceedings of the 19th International Workshop on Data Management on New Hardware. Seattle, USA, 2023: 10-18

[109] Akdere M, Çetintemel U, Riondato M, et al. Learning-based query performance modeling and prediction//Proceedings of the 2012 IEEE 28th International Conference on Data Engineering. Arlington, USA, 2012: 390-401

[110] Arefyeva I, Durand G C, Pinnecke M, et al. Low-latency transaction execution on graphics processors: Dream or reality?//ADMS@VLDB. Rio de Janeiro, Brazil, 2018: 16-21

[111] Momjian B. PostgreSQL: Introduction and Concepts. New York, USA: Addison-Wesley, 2001

[112] Breß S, Köcher B, Funke H, et al. Generating custom code for efficient query execution on heterogeneous processors. The VLDB Journal, 2018, 27: 797-822

[113] Idreos S, Groffen F, Nes N, et al. MonetDB: Two decades of research in column-oriented database architectures. IEEE Data(base) Engineering Bulletin, 2012, 35(1): 40-45

[114] A GPU Database that's just right for you: A guide. https://www.brytlyt.com/resources/articles/gpu-database-guide/, Feb., 2019

[115] SQream. GPU-Accelerated Data Warehouse. https://sqream.com/product/architecture/, 2018

附录 A.

论文中专业术语和缩略语如下表所示：

英文全称	英文缩写	中文名称	术语含义
Just in Time Compilation	JIT	即时编译	在程序执行过程中动态编译
No partition Join	NOP	不分区哈希连接	在连接过程中不对数据进行分区
No Partition Join Optimized	PRO	不分区哈希连接(优化)	优化后的不分区哈希连接
Partition Radix Based Join	PRB	有分区哈希连接	使用基数分区方法的哈希连接
Concise Hash Table Join	CHTJ	简明哈希连接	使用线性探测的哈希连接
Multi-way Sort Merge Join	MWAY	多路排序合并连接	使用多路排序方法进行哈希连接
Sort Merge Join	SMJ	排序合并连接	对连接键排序再进行哈希连接
Massively Parallel Dynamic Programming	MPDP	大规模并行动态规划	使用大量 GPU 执行哈希连接
Parallel Hybrid Join	PHYJ	并行混合连接	在 GPU 上执行并行连接算法
Nest Loop Join	NLJ	嵌套循环连接	使用嵌套方式执行哈希连接
Unified Virtual Address	UVA	统一虚拟地址	通过统一的虚拟地址确定虚拟内存
Unified Memory	UM	统一内存管理	使 CPU 和 GPU 共同使用虚拟内存
On-Line Analytical Processing	OLAP	联机分析处理	基于数据仓库多维模型实现分析操作

LIU Peng, Ph. D. candidate. His main research fields are new hardware database and main-memory database.

CHEN Hong, Ph. D., professor, Ph. D. supervisor. Her research interests include database technology and big data management.

ZHANG Yan-Song, Ph. D., associate professor. His research interests include new hardware database and data warehouse.

LI Cui-Ping, Ph. D., professor, Ph. D. supervisor. Her research fields are big data technology and big data mining.

Background

This paper is a review article, which mainly gives a brief introduction to the optimization technology of GPU database, and introduces and analyzes the technologies that have a great impact on query performance. By investigating the latest research results of these key technologies, researchers related to GPU database can understand the latest technological changes and research trends. The key technologies that affect the performance of GPU database are further studied and developed, and the potential research points and development trends in the future are predicted.

This paper mainly from the hardware architecture, data storage and management, query optimization key technology, core technology implementation and so on to expand. Specifically: Explain the causes of data transfer and access bottlenecks by understanding and analyzing CPU, GPU and their collaborative architectures. Subsequent technological improvement is also carried out around these two aspects. In the section of data storage and management, it mainly introduces the data storage and access mode, and introduces and analyzes the compression technology brought by the data storage format. In addition, the index access to the database is also very important. Although the cost of establishing and maintaining the index in the GPU database is very high, the benefits brought by the index are also very considerable, so the research on the index can not be ignored. The current query optimization technology of GPU database is mainly reflected in the operator level. The research on sorting, joining, grouping aggregation and OLAP operators has always been the focus and hotspot. Meanwhile, the problems of join algorithm and multi-table join order determine the upper limit of query optimization to some extent, which is also the hot and important difficulty as operator research. Because the GPU database is very different from the traditional disk database in the architecture, the problems such as cardinality estimation and cost evaluation of GPU database can no longer be solved by the original method. By using the learning method to do cost evaluation, the query plan with the lowest execution cost can be selected quickly and accurately. It is also a good combination of artificial intelligence technology and database technology. In terms of query processor and query execution engine, the main research is real-time compilation technology and optimization of processing model. In the optimization process, full use of hardware characteristics, optimization of execution process and execution engine has achieved good results, and the research on this aspect should also continue to pay attention.

This work is supported by the National Natural Science Foundation of China (Nos. 62072460, 62076245, 62172424, 62276270) and the Beijing Natural Science Foundation (No. 4212022).