文章编号: 1000-5641(2019)05-0178-12

基于 GPU 的关系型流处理系统实现与优化

黄 皓, 李志方, 王嘉伦, 翁楚良

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 现有的基于 CPU 的流处理系统在功能上已支持在大规模数据集上的复杂分析查询, 但由于 CPU 计算能力与特性的限制, 无法在性能上同时满足高吞吐量和低响应时间的要求. 本文提出一种基于 GPU 的流处理系统框架 Serval, 通过充分利用 CPU-GPU 异构资源, 实现了关系型流查询的高效处理. Serval 框架采用流水线模型和流执行缓存技术以优化吞吐量和响应时间, 并实现多种调优策略以适应不同场景. 实验表明, 单节点 Serval 的吞吐量与响应时间性能均优于现有 GPU 数据库 MapD 和三节点分布式服务器上的 Spark Streaming.

关键词: 流处理系统; 关系型查询; GPU 数据库

中图分类号: TP315 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2019.05.015

Implementation and optimization of GPU-based relational streaming processing systems

HUANG Hao, LI Zhi-fang, WANG Jia-lun, WENG Chu-liang

(*School of Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: State-of-the-art CPU-based streaming processing systems support complex queries on large-scale datasets. However, limited by CPU computational capability, these systems suffer from the performance tradeoff between throughput and response time, and cannot achieve the best of both. In this paper, we propose a GPU-based streaming processing system, named Serval, that co-utilizes CPU and GPU resources and efficiently processes streaming queries by micro-batching. Serval adopts the pipeline model and uses streaming execution cache to optimize throughput and response time on large scale datasets. To meet the demands of various scenarios, Serval implements multiple tuning policies by scaling the micro-batch size dynamically. Experiments show that a single-server Serval outperforms a 3-server distributed Spark Streaming by 3.87x throughput with a 91% response time on average, reflecting the efficiency of the optimization.

Keywords: streaming processing system; relational query; GPU database

收稿日期: 2019-07-29

基金项目: 国家重点研发计划(2018YFB1003400)

第一作者: 黄 皓, 男, 硕士研究生, 研究方向为内存型数据库. E-mail: haohuang@stu.ecnu.edu.cn.

通信作者: 翁楚良, 男, 教授, 博士生导师, 研究方向为并行与分布式系统.
E-mail: clweng@dase.ecnu.edu.cn.

0　引　　言

在人工智能与大数据时代, 高效的数据收集与处理已成为大数据系统、平台的核心竞争力, 并服务于基于大数据的科学研究与应用, 因而已成为一个重要的研究方向. 在不断有新数据生成的场景下, 最常用的处理方式有两种: 一是定期或定量的分批次处理, 二是在线、实时的流处理. 后者由于其时效性强, 在物联网、电商平台和交通监控等诸多领域中有着广泛的应用. 但随着数据规模的不断增长, 相关应用也对流处理系统的性能提出了更高的要求.

流处理模型自提出后已经历过多次改进, 而伴随着应用需求和硬件的发展, 相关的流处理系统经历多次革新后, 也已推出较为完善、通用的系统, 例如 Apache Spark Streaming[1], Apache Storm[2]等. 这些流处理系统的实现基于 CPU, 受限于硬件成本与计算能力, 通常要在吞吐量与响应时间之间做出权衡, 例如 Spark Streaming 以较长响应时间为代价, 换取较高的吞吐量, 而 Storm 则与之相反. CPU 计算资源的限制在复杂、数据规模较大的查询上表现得尤为明显. 为了进一步提升吞吐量, 并缓解吞吐量与响应时间的矛盾, 除增加分布式节点数量以提升整体计算能力外, 一种通用、高效的解决方案是使用计算能力更强、并行度更高的新硬件, 如 GPU、FPGA 等.

当下 GPU(Graphics Processing Unit, 图形处理单元)具有高带宽、大量计算单元等特性, 相比 CPU 可以提供更强的通用并行计算能力, 并已被广泛应用于异构系统中, 用于在大规模数据场景下加速并行计算、数据处理等. 在数据处理领域, 已有多支研究团队[3-5]将 GPU 应用于关系型数据库管理系统 (Relational Database Management System, RDBMS) 和流处理系统中, 使系统吞吐量获得了成倍的增长, 但这些系统仍有不足. 基于 GPU 的 RDBMS 系统基本都已实现对较复杂的关系型查询的支持, 并在许多场景下性能超过基于 CPU 的数据库系统. 但受使用场景的限制, 这些系统缺乏实时流处理能力. 而基于 GPU 的流处理系统, 如 GFlink[6]等, 只提供了通用场景下的流处理框架, 对于关系型流查询则没有做出针对性的优化.

本文提出并实现了一种基于 GPU 的关系型流处理框架 Serval, 它充分利用 CPU-GPU 异构体系的计算资源, 实现关系型流查询的高效处理, 可以满足实时交通监控、推荐系统等流分析场景下应用的高吞吐、较低延时的需求. Serval 采用微批次处理模式, 通过对流水线模型和流处理场景的优化, 实现了异构资源的高效利用, 并通过性能模型分析, 提供多种性能策略以动态适应不同场景的需求.

1　背　　景

1.1　流处理模式

当前流处理系统主要采用两种处理模式: “微批次” (Micro-batch)[1]和“纯流式” (Continous Operator Model)[2]. 这两种模式分别针对不同的现实应用场景, 并各自有不同的优缺点. 在本框架拟解决的场景下, 流具有无限行数据, 且实时生成、输入, 由流处理模块生成无限个微批次表: 参考表数据已知, 且更新频率较低.

采用微批次模式的系统, 如 Apache Spark Streaming 等, 将收集到的输入数据按照数量或者到达时间分为许多微批次. 这类系统在每个批次收集完成后将其置入待处理队列, 并依次按照设计好的查询进行处理. 在每个子查询执行开始前和结束时, 系统进行一次全局同步以保证数据被完全处理, 这被称为“屏障 (Barrier)”. 通过这一模式, 系统结构设计和实现较

为简单, 且在批次较大时, 能够在一定程度上分摊额外开销带来的影响(如容错、通讯延迟等), 以此实现吞吐量的最大化. 但这一批量处理的模式也显著增加查询的响应时间, 因此通常用于对响应时间不敏感的应用场景.

对响应时间敏感的应用则通常采用纯流式执行模式, 代表系统为 Apache Storm. 这类系统没有引入屏障的机制, 而是为每条输入的新记录分配一个新的任务 (Task), 并立即开始并行处理. 在这种场景下, 每个任务都将得到最快的处理和响应速度, 但任务启动、执行和返回过程中都引入了无法避免的开销, 例如通讯、容错机制等. 因此, 尽管这类模式可以实现较低的响应时间, 但由于各类开销限制, 其无法达到较高的吞吐量.

1.2 GPU 执行模式

不同品牌的通用计算 GPU 具体采用的术语不同, 但其执行模式类似, 因此本节以 NVIDIA 系列 GPU 为例说明 GPU 的执行模式.

NVIDIA 采用 CUDA[7]运行时作为通用计算应用的开发与运行时环境. 在 GPU 上执行的程序被称为“核函数 (Kernel Function), 由 CPU 线程调用并启动核函数, 同步或异步执行. 异步执行时 CPU 侧线程可以完全不受影响地执行后续任务. 同步执行时, 仅有调用设备同步接口的 CPU 线程被阻塞, 直至核函数运行结束. 因而, 在 GPU 上核函数执行期间, CPU 可以不被阻塞, 并行执行其他任务. 因此, CPU-GPU 异构硬件体系下的系统能够充分利用这一特性, 合理分发任务, 实现高效的并行计算.

在核函数执行前, 程序需要指定为核函数分配的线程数量; 核函数执行期间, GPU 硬件将全部线程以线程束为单位划分(通常每个线程束内有 32 个线程), 所有线程束调度在流多处理器 (Stream Multiprocessor, SM) 上. 每个线程束内所有线程在各自的数据元组上执行相同的指令, 这被称为单指令多线程 (Single-Instruction Multiple-Thread, SIMT) 执行模式. 当需要发起全局内存访问指令时, 由于全局内存访问耗时较长, GPU 硬件将当前线程束换出, 并换入、启动其他线程束, 以此来掩盖全局内存访问延时带来的硬件闲置. 这一机制保证了在数据量大的场景下 GPU 硬件资源总能处于执行状态. 但数据规模较小时, 需要处理的线程束数量不足, 无法通过调度重叠以减少等待带来的资源闲置, 导致计算资源利用的不充分, 进而降低计算吞吐量. 本框架采用了根据负载特性和使用场景动态调整合适的微批次表大小的方法, 以降低这一问题的影响.

2 框架架构与设计

2.1 架构

Serval 框架的整体结构如图 1 所示, 在实现了高效执行关系型流查询处理的同时, 进一步分别为流处理场景和关系型查询做出优化, 以实现 CPU-GPU 异构体系资源的高效利用. Serval 框架主要分为两个部分: CPU 侧和 GPU 侧. CPU 侧主要承担较轻量数据、或逻辑复杂的处理过程, 例如流记录的预处理和后处理、查询请求分析与优化、流数据序列化与反序列化和解析、导入与导出等; GPU 则作为查询执行器, 承担最主要的查询请求执行部分. GPU 核函数启动时, 框架读取 GPU 状态信息并调用最大线程数量, 以充分利用 GPU 本身的硬件调度功能. 在多 GPU 环境下, 设备管理模块需要将对应的核函数和参考表的数据均匀分发到各 GPU 上以实现负载均衡.

为了适应 GPU 的硬件结构, 系统采用微批次处理模式, 并以列存储格式作为内部存储和处理格式. 其中, 采用列存储主要出于以下两点原因[8]: (1) 大部分查询中不涉及到表中所有的列, 列存储下可以仅传输部分列的数据, 减少 CPU 到 GPU 间的 PCI-E 数据量, 以此

降低传输开销; (2) 列存储格式下可以实现合并内存访问, 相比行存储格式更适合 GPU 的 SIMT 执行模式. 但列存储格式会引入额外开销: 流数据导入为列存储的转换和部分中间结果引起的行列转换. Serval 框架使用流水线模型(第 3.1 节)和流执行缓存(第 4 节)以分别减轻前者和后者所带来的影响.

2.2　模块设计

根据实现功能不同, 框架中各组件可以分为 4 个模块, 在图 1 中通过不同的背景样式区分: 查询处理, 流处理, 数据管理, 以及流执行缓存模块. 每个模块的具体功能如下.

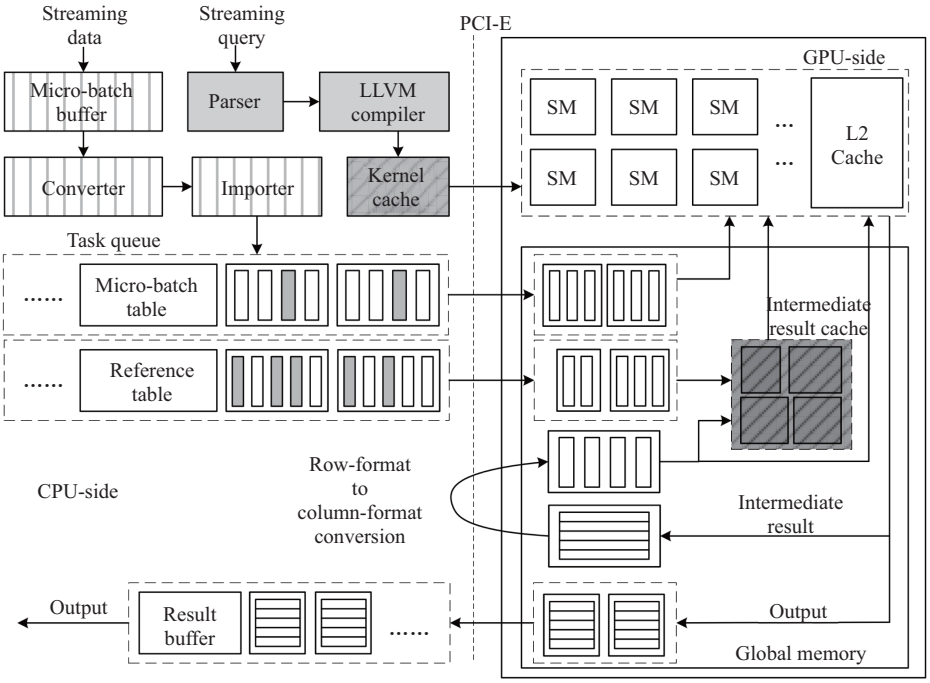

图 1　Serval 架构总览

Fig. 1　Architecture overview of Serval

查询处理模块 (Query Processor) 包含两个组件: 解析器和编译器. 解析器将流查询解析为一个关系代数树, 并将其优化为一个有向无环图 (Directed Acyclic Graphic, DAG), DAG 图中的节点代表关系型算子, 如连接、排序等. 解析器将有向无环图展开为一个算子队列, 以待后续依次执行.

解析器完成解析后, 编译器将算子队列中的每个算子节点编译成独立的核函数. 框架应用 LLVM IR 技术, 编译器在运行时动态地将新输入的请求编译为可以在 GPU 上运行的核函数指令, 并在编译时执行进一步优化. 由于编译中的链接步骤无法实现多线程并行, 因此当查询较为复杂时, 这一步骤可能耗费较长时间, 带来额外开销. 框架引入了核函数缓存(第 4 节)以消除此负面影响.

流处理模块 (Streaming Processor) 不间断收集新到达的流记录到微批次中, 并执行解析、格式转换等预处理和结果的后处理. 实际应用场景中，流记录通常为 CSV 或 JSON 格式, 需要转换为关系型数据库所支持的数据类型, 这一步骤在 CPU 上的执行效率显著高于 GPU. 在每个微批次收集结束后, 由 CPU 执行轻量化的行列转换, 将收集到的微批次转换为

微批次表 (Micro-batch Table), 并由导入模块 (Importer)加载到数据库中.

数据管理模块 (Data Manager) 处理并保存所有微批次表和参考表 (Reference Table) 的数据与元数据信息, 分配与释放 CPU 侧或 GPU 侧的缓冲区, 等等. 微批次表收集完成后加入到任务队列 (Task Queue) 队尾, 并在处理完成后移出队列.

如图 1 所示, CPU 侧和 GPU 侧都拥有任务队列和参考表缓冲区, 但这两者之间存在不同: (1) GPU 侧的缓冲区只持有查询相关的列数据, 即 GPU 侧只缓存图 1 中 CPU 侧不带有阴影的部分列, 而 CPU 侧的缓冲区持有全部数据; (2) GPU 侧的缓冲区只持有有限数目的微批次表, 以此来降低 GPU 全局内存的占用, 而 CPU 侧缓存全部微批次表, 直到处理完成后将其移除, 原因在于 CPU 侧内存空间可以灵活扩充.

流执行缓存模块 (Streaming Execution Cache, SEC) 用于降低特定场景下的重复计算与传输开销, 分为中间结果缓存 (Intermediate Result Cache, IRC) 和核函数缓存 (Kernel Cache, KC). IRC 所缓存的中间结果位于 GPU 的全局内存中, 而被缓存的结果的指针信息存储在 CPU 侧内存中, 使用对应的算子获得. KC 位于 CPU 侧内存, 存储每个流查询的编译结果. 每部分的实现细节将在第 4 节中描述.

3 流水线执行与性能策略

3.1 CPU-GPU 上 4 阶段流水线模型

为了充分利用 CPU 和 GPU 异构体系的资源, 系统中应用了流水线模型来实现资源合理调度, 并在此基础上进一步优化.

根据所需的硬件和任务属性的不同, 执行工作流可以分为 4 个阶段, 如图 2 所示: 读取 (Read), 转换 (Conv), 加载 (Load) 和执行 (Exec). 在流水线模型处于未满载情况下, Serval 吞吐量可以满足输入要求, 此时最优先的性能指标为每个批次的响应时间. 在满载和过载情况下, 输入速率大于吞吐量, 导致输入产生积压, 严重影响后续批次的响应时间, 故此时框架首要考虑的指标为吞吐量.

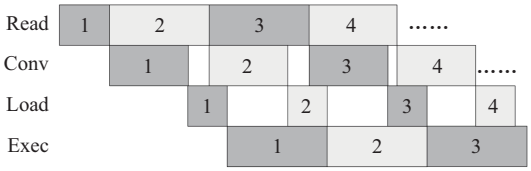

图 2 流水线模型

Fig. 2 Pipeline model

读取阶段, 流处理模块不断接收并缓存原始的流记录, 保存在微批次中. 在转换阶段, 这些微批次被解析为关系型数据库内置的数据类型, 如整数、浮点数、文本等. 之后, 在加载阶段, 每个微批次都被单独保存为一个新的微批次表, 以列存储格式导入数据库中, 并放入任务队列末尾. 执行阶段包含数个子阶段, 但这些阶段耦合程度较高, 无法进一步划分, 且内部子阶段无法并行执行: 包括微批次表到 GPU 全局内存的拷贝, 运行时代码编译, 设备上核函数启动和结果收集等.

为了进一步分析框架性能与微批次表的关系, 对该流水线模型做出如下建模. 首先定义每个阶段的执行时间分别为 $t_{\text{read}}, t_{\text{conv}}, t_{\text{load}}$ 和 t_{exec}, 定义微批次的大小为 $size$, 可以得出请求响应时间 $L(size)$ 为所有子阶段的响应时间的总和, 而整体吞吐量 $T(size)$ 受耗时最久的

阶段限制, 如式 (1) 与 (2) 所示.

$$L(size) = t_{\text{read}} + t_{\text{conv}} + t_{\text{load}} + t_{\text{exec}}, \tag{1}$$

$$T(size) = \frac{size}{\max\{t_{\text{read}}, t_{\text{conv}}, t_{\text{load}}, t_{\text{exec}}\}}. \tag{2}$$

实验结果测试表明, $t_{\text{read}}, t_{\text{conv}}, t_{\text{load}}$ 三个阶段与微批次表的大小近似成线性关系, 而 t_{exec} 除去与查询相关的固定开销 C 后, 剩余执行时间与微批次表的大小成线性关系. 因而为每个阶段定义一个比率 r_i, 将上式进一步简化后得到:

$$L(size) \approx (r_{\text{read}} + r_{\text{conv}} + r_{\text{load}} + r_{\text{exec}}) \times size + C, \tag{3}$$

$$T(size) \approx \min\left\{\frac{1}{r_{\text{read}}}, \frac{1}{r_{\text{conv}}}, \frac{1}{r_{\text{load}}}, \frac{1}{r_{\text{exec}} + C/size}\right\}. \tag{4}$$

式 (3) 与 (4) 表明, 微批次表越大, 即微批次内记录数量越多, 对应的响应时间就越久, 吞吐量也将在一定范围内得到提升. 因此, 通过调整微批次表大小的方式, 可以达成不同的吞吐量与响应时间目标. 相反, 微批次表大小固定时, 提高每一步骤处理效率, 可以降低总查询响应时间; 降低耗时最长阶段的执行时间, 则可提高 Serval 整体吞吐量.

3.2　性能策略

实际应用中, 不同场景下对系统吞吐量和响应时间有着不同的要求. 如上一小节所述, 通过合理地调整微批次表的大小, 可以使框架达到不同的指标要求. 因而, 为了适应各种场景, Serval 实现了 3 种性能策略.

在 GPU 上执行的微批次表较小或较大时, 微批次表中的流记录在 GPU 计算单元上的分布特点不同. 微批次表较小时, 将有未被分配处理线程的 GPU 计算单元处于闲置状态, 或在等待全局内存访问时没有可以换入的线程束, 降低计算资源利用率. 而微批次表足够大时, 每个 GPU 计算单元上分配了多个线程和流记录, 运行中极少处于闲置状态. 且在全局内存访问等待阻塞时, 流多处理器可以换入其他线程束, 降低硬件空闲, 提高处理效率, 进而提升吞吐量.

因此, 微批次中记录数量较多, 或参考表较大的场景下, Serval 需要同时启动大量工作线程, 以实现 GPU 计算资源的充分利用: 但较大的微批次同样会引起流水线上各阶段用时增加, 导致响应时间增长. 为了同时兼顾到实际应用场景需求, Serval 中实现了如下 3 种性能策略.

最大吞吐量(Max Throughput, MT). 由式 (4) 可以得出, Serval 框架吞吐量由流水线执行中耗时最久的阶段所限制. 因此, 根据瓶颈的不同, 得到式 (5), 当瓶颈处于执行阶段时, 微批次表越大则可以获得的吞吐量越大; 当瓶颈处于其他阶段时, 则最大吞吐量与微批次表大小无关, 而是受限于该阶段的处理效率.

$$T_{\max} = \max\{T(size)\} = \begin{cases} size_{\max}, & \text{若 } t_{\text{exec}} \geqslant t_i, \\ \min\left\{\dfrac{1}{r_{\text{read}}}, \dfrac{1}{r_{\text{conv}}}, \dfrac{1}{r_{\text{load}}}\right\}, & \text{若 } t_{\text{exec}} < t_i. \end{cases} \tag{5}$$

最小响应时间 (Min Latency, ML). 由式 (3) 容易得到, 在各阶段处理效率不变的情况下, 微批次表越小, 响应时间就越短. 因而, 这一策略总是选择尽可能小的微批次表大小, 但这将严重降低吞吐量, 在实际应用中没有太大的应用价值, 如第 5.2 节实验中所述. 该策略由

于吞吐量性能过差, 在实际应用中没有太大的应用价值, 证明 Serval 并不适用于这类响应时间敏感的场景.

平衡策略(Balanced Policy, BP). 如前所讨论, 微批次表越大, 则吞吐量就越大, 同时响应时间越长. 然而, 吞吐量与响应时间的增长速率并不相同: 随着微批次表大小的增加, 响应时间增长速率高于吞吐量增长速率. 因此, 本策略尝试寻找一个平衡点, 在该点处, 微批次表大小增长 δ 时, 牺牲响应时间换取吞吐量的效率达到最高.

$$\frac{T(size+\delta)}{T(size)} \leqslant \frac{L(size+\delta)}{L(size)} \Rightarrow \frac{T(size)}{L(size)} \geqslant \frac{T(size+\delta)}{L(size+\delta)}. \tag{6}$$

式 (6) 证明, 微批次表大小使 T/L 比值达到最大时, 效率将达到最高点. 因而定义性能效率 β 与平衡点 $size_b$, 如式 (7). Serval 采用该策略运行时, 通过不断监测当前微批次表大小和处理时间, 即可得到该平衡点 $size_b$, 此时 β', 即性能效率对 size 的导数应为 0.

$$\beta(size) = \frac{T(size)}{L(size)}, \quad \beta'(size_b) = 0. \tag{7}$$

4 流执行缓存

流查询处理中, 一方面在参考表输入未发生变化时, 常有一部分算子查询产生的中间结果保持不变, 因而可以将这部分未变化的中间结果加以缓存, 以减少重复计算; 另一方面, 流查询通常固定, 因而查询请求解析、编译的过程也可以加以缓存. 因此, Serval 采用流执行缓存, 包括两个组件: 中间结果缓存和核函数缓存, 以针对上述场景做出优化.

中间结果缓存(Intermediate Result Cache, IRC) 用于减少不同微批次表执行时的相同中间结果的冗余计算. 一个较为复杂的查询可能含有多个输入, 通常包括一个流输入和若干个参考表输入. 当某个算子的参考表输入未发生变化时, 该算子生成的中间结果可以被缓存在位于 GPU 全局内存中的 IRC 中, 以此避免算子在 GPU 上的重新计算, 还可以减少行列格式转换所带来的 CPU 与 GPU 间的冗余拷贝.

每个算子的计算开始前, 首先检查参考表输入的更新时间, 若所有输入的更新时间与 IRC 内所记录的时间相同, 则可以从 IRC 中获取已有的缓存, 跳过执行阶段; 若参考表输入的更新时间晚于 IRC 内所记录的时间, 则说明参考表已被更新, 因而需要重新计算, 新的中间结果将覆盖原有结果. 进一步, 所有子算子的参考表输入均未发生变化时, 其父算子同样可以从 IRC 中获取已缓存的输出结果.

中间结果缓存对于较简单的查询能带来的提升有限, 但可以为复杂查询带来较大的性能提升. 原因在于, 简单查询通常很少出现符合中间结果缓存条件的算子, 因而无法得到性能提升, 而复杂查询常常包含较多这一类算子, 中间结果缓存可以降低这类查询计划的执行时间.

核函数缓存(Kernel Cache, KC) 用于避免相同查询请求的重复编译, 在每个微批次表的查询请求执行前, 可以从 KC 中取得已有的编译结果. 这要求在编译时动态生成的代码与具体数据及数据量无关, 以生成可复用的指令. 具体实现方式为, 运行时生成的核函数与具体数据无关, 仅在每次执行时将元数据信息作为参数传入, 如每一列的缓冲区指针、行数, 输出预留缓冲区的指针信息等等. LLVM 动态编译生成的代码即可在不同批次间共享, 避免重复编译, 此时核函数启动前的元数据信息由数据管理模块提供. 核函数缓存能够进一步降低执行阶段中的固定开销, 实现整体性能提升.

5 实　　验

环境与配置. Serval 框架测试使用单服务器, 配有两个 Intel Xeon 4110 8 核 CPU, 160 GB 内存和一个具有 24 GB 全局内存的 NVIDIA Tesla P40 GPU. 测试中作为对比的 Apache Spark Streaming 2.3.2 和 Apache Flink 1.7.1 使用三个相同 CPU 和内存配置的服务器, 节点间采用 10 GbE 万兆网络互联. 服务器均安装操作系统 CentOS 7.4 与 NVIDIA CUDA 9.1 版本.

系统集成. Serval 框架实现于 GPU 数据库系统 MapD[9-10] 的开源版本 4.4.1 之上. MapD 本身具有将流记录导入到数据库内、经过同步后作为参考表以及事实表处理的功能, 但缺少实时流查询和本文所述的优化功能. Serval 框架集成实现主要包括: 部分利用其关系型查询处理模块, 重新设计和实现原有的部分模块和组件(数据管理模块, 请求解析、编译模块), 并加入新的流处理模块、流执行缓存 (IRC 与 KC) 模块等.

实验数据集. 由于关系型查询的流处理应用场景繁多, 没有通用的基准测试, 因此本文采用修改版的 TPC-H 关系型测试标准数据集 Scale Factor 10 和 100, 总数据集大小分别为 10 GB(SF10) 和 100 GB(SF100) 作为实验测试基础. 修改内容为, 将 TPC-H 数据集中 LINEITEM 表作为测试中的流记录表, 以文件流的形式输入. 其余表作为参考表, 预先加载到系统文件存储中, 并在启动时加载入内存. 所测试的查询涵盖与 LINEITEM 表相关联的 Q1, Q3, Q6, Q12, Q14, Q17 和 Q18, 其余与 LINEITEM 表无关的查询不在测试范围之内.

5.1 与流处理系统对比

Serval 的最大吞吐量策略 (MT) 和平衡策略 (BP) 与 Spark Streaming, Apache Flink 和 MapD 的对比结果如图 3、图 4 所示, 图中 (a) 与 (b) 分别采用 TPC-H SF10 和 SF100, 以查询请求分组. 图 3 为各系统吞吐量与 Spark Streaming 的比值, 比值高表示系统吞吐量高, 性能更优; 图 4 为各系统响应时间与 Spark Streaming 的比值, 比值低表示系统响应时间更短, 性能更优.

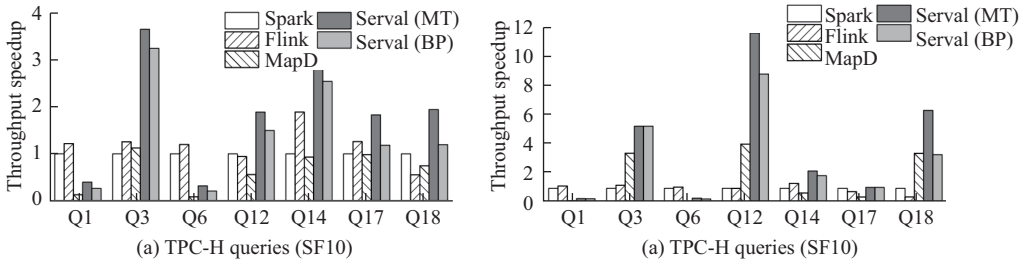

图 3　与现有流处理系统对比: 吞吐量

Fig. 3　Comparison of different streaming processing systems: throughput

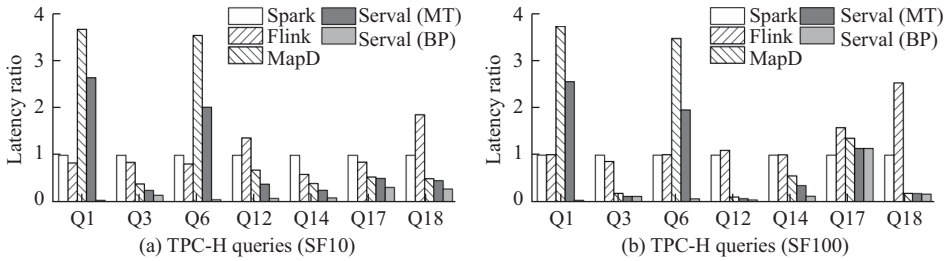

图 4　与现有流处理系统对比: 响应时间

Fig. 4　Comparison of different streaming processing systems: latency

由图 3、图 4 可知, 在 SF10 测试中 Q1 与 Q6 之外的其他查询下, 单服务器的 Serval 采用最大吞吐量策略时, 相比于三节点分布式的 Spark Streaming 和 Flink 均可以获得较多的吞吐量提升, 且响应时间同样快于后两者, 这部分性能提升主要出于 GPU 的高计算能力与内存吞吐量. SF100 数据集下性能提升进一步扩大, 原因在于, 在连接等计算密集型的算子上, 较大的参考表带来大量线程和任务, 可以更好发挥 GPU 的并行计算性能.

Serval(MT) 在 Q1 和 Q6 上表现不如 Spark Streaming 与 Flink, 原因在于这两个请求仅处理微批次表中数据, 不涉及与参考表的连接操作, 因而计算量较小, 在 GPU 上的执行阶段可以迅速完成, 瓶颈转移到 CPU 侧的行列转换阶段, 此时三节点的 Spark Streaming 和 Flink 拥有更强的 CPU 计算能力, 且无行列转换效率的限制, 因而达到了更高的吞吐量. 尽管有上述限制, Serval 的绝对性能达到了超过 800 k 行/s 的吞吐量, 和每个微批次表少于 1 300 ms 的响应时间, 均优于 Serval 上其他查询.

如图 3、图 4 中 Serval 与 MapD 对比所示, 最大吞吐量策略下，Serval 相较于 MapD 同样在吞吐量与响应时间性能上带来了较大幅度提升. MapD 的批量导入、更新的执行模型无法实现 CPU-GPU 并行执行, 而 Serval 的流水线模型实现了 CPU-GPU 异构资源的利用, 有效提升了吞吐量. Serval 引入了流执行缓存, 减少了部分微批次表的执行时间, 因而在未修改 MapD 算子实现的基础上, 仍可进一步缩短平均响应时间.

图 3、图 4 中展示了采用平衡策略时 Serval (BP) 与最大吞吐量策略时 Serval (MT) 的对比. Serval 可以在仅损失平均 19% 吞吐量的情况下，降低 63% 的平均响应时间, 证明该平衡策略可以有效地寻找并利用平衡点. 而最小响应时间策略下, 由于其吞吐量过低, 且此类应用和场景不适合使用 GPU 进行处理, 因此将其在结果中排除, 未展示在图中.

上述结果表明, 采用 GPU 加速可以为瓶颈位于执行阶段的查询带来显著的吞吐量提升, 同时能降低大部分测试语句下的响应时间. 整体而言, Serval(MT) 对比 Spark Streaming, 响应时间性能保持不变, 在 SF10 和 SF100 下分别得到 1.8 倍和 3.87 倍的平均吞吐量.

5.2 流查询深入分析

本节选取 Q14(SF100), 深入分析其中各部分时间开销随着微批次表大小的变化. 如图 5 所示, 横坐标为不同的微批次表大小, 即微批次表行数, 由 8 k 行逐渐增长到 512 k 行; 左侧纵坐标表示流水线上不同阶段的时间开销, 并采用颜色区分 4 个阶段; 右侧纵坐标表示系统吞吐量的绝对值, 以折线图表示; 为方便展示相对关系, 性能效率 β 以 $\beta(size)/\beta(8\text{ k})$ 以虚线表示在图中, 并标注了其最大值与最小值.

整体而言, 随着微批次表增大, Serval 框架的吞吐量也逐渐上升, 但增长速度渐渐放缓; 同时响应时间也接近线性增长, 且流水线模型中 4 个阶段的时间开销增长速度各不相同: 执行阶段的时间增长最慢, 而其他阶段的时间增长与微批次表的大小的增长成线性关系.

采用最大吞吐量策略时, Serval 选取微批次表大小上限 512 k 行为最优, 同样获得了最高吞吐量; 而采用平衡策略时, 在 128 k 行时 β 达到最大值, 因而取 128 k 行为平衡点 $size_b$, 此时吞吐量转化为响应时间的效率曲线达到拐点: 微批次表大小小于平衡点时, 吞吐量增长速度快于响应时间增长, 大于该点后则相反. 相比于最大吞吐量策略, 最短响应时间策略选择 8 k 行作为最优微批次表大小, 以 13% 的响应时间获得了 7% 的吞吐量, 平均响应时间虽然较短, 但吞吐量过低, 无法满足常见场景下的需求. 对比之下, 平衡策略仅牺牲了 15% 的吞吐量, 即可缩短 63% 的响应时间, 效率达到了最短响应时间策略的 4.5 倍.

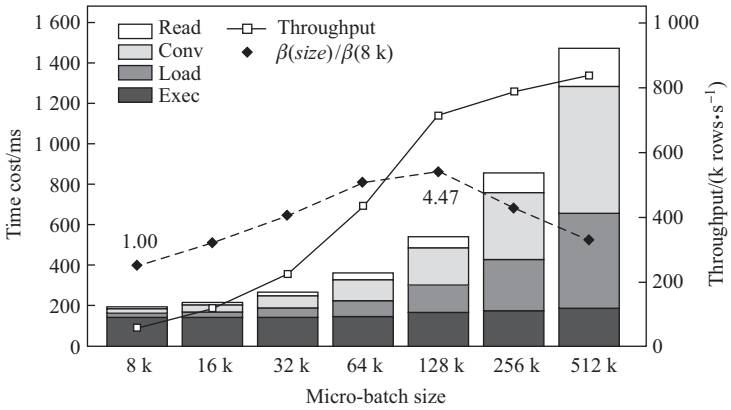

图 5　Q14 (SF100) 执行分析

Fig. 5　Q14 (SF100) execution analysis

5.3　流执行缓存评估

实验查询中, 各流水线阶段的处理时间与流执行缓存所降低的开销如图 6 所示, 图中横坐标表示实验中的所有查询, 纵坐标表示各阶段所花费的时间开销绝对值, 其中斜线部分的负值代表加入流执行缓存后降低的部分. 实验结果显示, 并非所有查询的性能开销都能得到提升, 原因在于实现系统所基于的 MapD 已部分支持核函数缓存, 因此尽管 Serval 做出了进一步优化, 但在一部分场景下 MapD 本身已实现高度优化, 无法继续提升.

由图 6 可知, 中间结果缓存在 TPC-H SF100 数据集下 Q1 与 Q6 之外的查询中降低了 5~15% 的执行时间开销, 主要原因在于 SF100 数据集中参考表较大, 导致计算时间较长, 因此通过减少参数表计算量的方式, 中间结果缓存可以显著缩短复杂查询的执行时间. 而 SF10 下, TPC-H 参考表较小, 总计仅为 3.4 GB, 在 MapD 原生优化下已被加载在 GPU 全局内存中, 重新计算较快, 因而这部分优化带来的提升不明显. 核函数缓存则可以进一步在 Q1 与 Q6 之外的请求上, 平均降低 15% 的执行时间. 其中查询 Q1 与 Q6 只涉及到微批次表, 因而中间结果缓存在该场景下不生效, 且 MapD 原生的核函数缓存功能足以优化这两个查询, 核函数缓存也无法带来进一步的优化.

在本测试场景与数据集 Q1 与 Q6 之外的查询中, 流执行缓存进一步为 Serval 系统降低了约 7% 的平均响应时间, 并同时提升了 20% 的平均吞吐量.

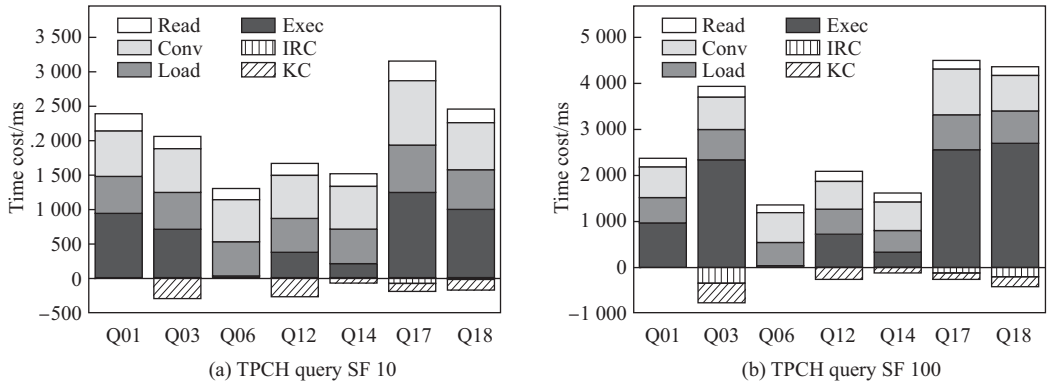

图 6　执行时间花销分析

Fig. 6　Execution time cost analysis

6 相关工作

GPU 关系型数据库系统. 当前基于 GPU 的关系型数据库系统, 如 OmniDB[5], Ocelot[3], MapD[9-10], Kinetica[11], SQream[12] 等等, 主要关注于静态表下的查询处理与优化. 其中只有部分系统如 MapD 等提供流导入功能, 允许用户导入流记录, 并追加在参考表、事实表末尾, 并每隔一段周期执行一次查询. 这些基本功能无法用于处理连续、实时的流查询. 本文工作则实现了这一功能, 并带来了诸多后续优化.

流处理系统. 近年来, 研究者们提出并发布了多种流处理系统, 分别针对不同的场景以解决各类问题, 并在不同的维度有着优异的表现, 如 Apache Storm[2], Spark Streaming[1], Flink[4] 等. 这些系统提供分布式集群上的通用流处理计算, 分别应用于不同的场景: 高吞吐量或低延时. 除了上述基于 CPU 的系统之外, 也有许多研究者探索 GPU 在流处理场景下的应用, 如 GStorm[13] 和 GFlink[14]分别将 GPU 集成入 Storm 和 Flink, 实现了通用计算下的运行和接口, 并做出进一步的优化, 为计算密集型负载进一步带来了性能提升. 这些工作与本文工作的不同之处在于, 前者主要面向通用计算, 并未为关系型流查询带来针对性的优化.

GPU流处理框架. 除了上述完善的系统之外, 有些研究者提出并实现了多种基于 GPU 的流处理框架, 如 GStream[15], SnuCL[16], GStreamMiner[17] 和文献 [18] 等. 然而, 这些工作仅为相关开发者提供了编程库和接口, 仍然需要开发者在此基础上编程实现流处理应用, 而非开箱即用的完整系统.

7 总 结

本文介绍了支持关系型流查询的流处理框架 Serval 的实现与优化. Serval 以微批次的形式, 利用 GPU 的硬件特性, 高效地实现了微批次表与参考表之间的关系型操作. 通过流水线模型, Serval 充分利用了 CPU-GPU 异构计算资源, 进一步提高了系统的吞吐量. 在该模型基础上, 本文提出了多种性能策略以适应不同场景, 实现在吞吐量与响应时间之间的平衡. 除此之外, 为减少流处理场景下的重复计算与传输, Serval 进一步采用流执行缓存的方式, 对中间结果和核函数进行缓存. 实验结果表明, 经过优化后的单节点 Serval 框架的吞吐量显著超过三节点分布式 Spark Streaming 和 Flink, 同时基本保持了响应时间性能.

[参 考 文 献]

[1] ZAHARIA M, DAS T, LI H Y, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters [C]//Proceedings of the 4th Workshop on Hot Topics in Cloud Computing. USENIX Association, 2012.

[2] IQBAL M H, SOOMRO T R. Big data analysis: Apache storm perspective [J]. International Journal of Computer Trends and Technology, 2015, 19(1): 9-14.

[3] BRESS S, KÖCHER B, HEIMEL M, et al. Ocelot/HyPE: Optimized data processing on heterogeneous hardware [J]. Proceedings of the VLDB Endowment, 2014, 7(13): 1609-1612.

[4] CARBONE P, KATSIFODIMOS A, EWEN S, et al. Apache Flink: Stream and batch processing in a single engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 28-38.

[5] ZHANG S, HE J, HE B, et al. Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures[J]. Proceedings of the VLDB Endowment, 2013, 6(12): 1374-1377.

[6] CHEN C, LI K, OUYANG A, et al. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(6): 1275-1288.

[7] Nvidia Cooperation. CUDA C Programming Guide[R/OL]. (2018-04-01)[2019-05-02]. https://docs.nvidia.com/cuda/ archive/9.1/pdf/CUDA_C_Programming_Guide.pdf.

[8] BRESS S, HEIMEL M, SIEGMUND N, et al. GPU-accelerated database systems: Survey and open challenges [M]//Transactions on Large-Scale Data and Knowledge-Centered Systems XV. Berlin: Springer, 2014: 1-35.

[9] MOSTAK T. An overview of MapD (massively parallel database) [R]. White paper. Massachusetts Institute of Technology, 2013.

[10] ROOT C, MOSTAK T. MapD: A GPU-powered big data analytics and visualization platform [C]// ACM SIGGRAPH 2016 Talks. ACM, 2016: 73.

[11] Kinetica DB Inc. Kinetica high performance analytics database [EB/OL]. [2019-05-11]. https://www. kinetica.com.

[12] SQream Technologies. SQream: Big Data SQL database [EB/OL]. [2019-05-02]. https://sqream.com/.

[13] CHEN Z, XU J, TANG J, et al. GPU-accelerated high-throughput online stream data processing [J]. IEEE Transactions on Big Data, 2016, 4(2): 191-202.

[14] CHEN C, LI K, OUYANG A, et al. GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(6): 1275-1288.

[15] ZHANG Y, MUELLER F. GStream: A general-purpose data streaming framework on GPU clusters [C]// 2011 International Conference on Parallel Processing. IEEE, 2011: 245-254.

[16] KIM J, SEO S, LEE J, et al. SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters [C]// Proceedings of the 26th ACM International Conference on Supercomputing. ACM, 2012: 341-352.

[17] HEWANADUNGODAGE C, XIA Y, LEE J J. GStreamMiner: A GPU-accelerated data stream mining framework [C]// Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. ACM, 2016: 2489-2492.

[18] HUYNH H P, HAGIESCU A, WONG W F, et al. Scalable framework for mapping streaming applications onto multi-GPU systems [C]// ACM Sigplan Notices. ACM, 2012, 47(8): 1-10.

(责任编辑: 林　磊)