中国科学: 信息科学 2022 年 第 52 卷 第 2 期: 314–333
SCIENTIA SINICA Informationis
论文

FJoin: 一种基于 FPGA 的流连接并行加速器

林力韬, 陈汉华*, 金海

华中科技大学计算机科学与技术学院, 大数据技术与系统国家地方联合工程研究中心, 服务计算技术与系统教育部重点实验室, 集群与网格计算湖北省重点实验室, 武汉 430074
* 通信作者. E-mail: chen@hust.edu.cn

收稿日期: 2021–06–25; 修回日期: 2021–08–26; 接受日期: 2021–09–03; 网络出版日期: 2022–01–18

国家重点研发计划 (批准号: 2016QY02D0302) 和国家自然科学基金 (批准号: 61972446) 资助项目

摘要 流连接广泛应用于提取多源流数据之间的关键信息, 是大数据处理的重要支撑技术. 但连接两条大数据流时大规模的连接谓词计算, 使其易成为性能瓶颈. 为提高处理性能, 流连接系统常采用并行和分布式两种方式扩展. 然而, 采用多核并行的流连接系统的扩展性受到 CPU 核数限制, 无法应对大规模数据流. 采用分布式扩展的流连接系统由于引入分布式框架运行的开销, 导致硬件处理效率严重下降. 为实现高效大规模扩展, 本文提出一种利用 FPGA 加速器外设向上扩展的流连接系统 FJoin. 加速器可进行高并行的流动连接, 载入多个流元组后, 连接窗口中的数据流经一次即可完成所有连接计算. 对于逻辑容易在 FPGA 实现的连接谓词, 通过大量基本连接单元串联构成深度连接流水线, 实现大规模并行. 通过主机 CPU 和 FPGA 设备协同进行连接控制, 将连续的流连接计算划分为独立的小批量任务, 高效地保证并行化流连接的完整性. 在装备 FPGA 加速卡的平台实现了 FJoin, 基于大规模真实数据集的测试结果表明, 对比部署在 40 个节点集群上的目前最好的分布式流连接系统, 本文提出的流连接加速器 FJoin 可在单一 FPGA 加速卡上将连接计算速度提升 16 倍, 达到 5 倍的系统吞吐, 且时延满足实时流处理要求.

关键词 流连接, FPGA, 流处理, 硬件加速, 并行计算

1 引言

万物互联时代, 实时流数据泛在多源, 无处不在. 据国际数据公司 (IDC) 预计, 全球数据将在 2025 年增至 175 ZB, 其中 30% 是规模日益增长的实时数据 [1]. 由于连接不同来源的流能够提取多源流数据之间的关键信息, 流连接成为流处理中的重要操作. 然而, 在流处理的各种算子中, 流连接 (stream join) 算子因计算开销大易成为系统性能瓶颈. 连接算子需要将数据流的元组存储在内存中, 形成连接窗口, 再与另一条流到达的元组连接. 为使得流连接算法通用于不同的连接谓词 (join

引用格式: 林力韬, 陈汉华, 金海. FJoin: 一种基于 FPGA 的流连接并行加速器. 中国科学: 信息科学, 2022, 52: 314–333, doi: 10.1360/SSI-2021-0214
Lin L T, Chen H H, Jin H. FJoin: an FPGA-based parallel accelerator for stream join (in Chinese). Sci Sin Inform, 2022, 52: 314–333, doi: 10.1360/SSI-2021-0214

predicate), 需要逐一判断来自两条流的元组对是否满足给定的连接谓词条件, 决定是否产生连接结果. 这使得连接算子所需的计算量正比于两条流流速的乘积 [2], 计算开销远高于一般的映射 (map)、过滤 (filter)、聚合 (aggregation) 等流处理算子. 当流处理系统的计算能力无法满足实际流数据规模时, 会立即引发严重的拥塞, 导致处理时延快速上升, 无法实现流处理低时延的要求. 因此, 研究高性能的流连接系统至关重要.

现有研究 [3,4] 一般通过设计多核并行系统, 以获得低时延高吞吐的处理性能. 并行流连接系统能够将连接窗口划分为多个小的分片, 分配给不同的 CPU 核心, 将处理每个新元组的开销分摊到多个单元. 基于多核架构的优势是容易将并行度扩展至 CPU 核数, 同时保持流元组的全局顺序. 在全局顺序下, 容易保证连接结果不重复和遗漏, 即满足流连接的完整性 (completeness) [5]. 并行方式还容易收集所有连接核产生的结果, 以连接元组的原有顺序排序, 即满足保序输出 (order preserving of results) [4]. 但是, 并行方式的并行度扩展受限于机器的资源, 无法应对大规模数据.

为了进一步提高流连接处理的并行度, 支持大规模向外扩展 (scaling-out) 的分布式流连接系统近几年受到广泛关注 [2,6~8]. 分布式方式一般建立在分布式流处理系统之上, 如广泛使用的 Apache Storm 系统 [9] 等. 这引入了框架运行的固有开销, 同时增加了节点间通信开销. Zeuch 等 [10] 最新的研究显示由于采用 Java 虚拟机实现平台独立性等原因, 当前的分布式流处理系统无法充分利用多核处理器和高速网络的硬件性能. 我们针对流连接的实验也证明了这一点, 使用滴滴公司1)提供的真实数据集与相应连接谓词, 我们对比了 16 个单核计算节点的分布式系统和 16 核单一节点的并行系统每秒连接谓词计算量. 图 1 的结果显示在相同并行度下, 并行方式实现的每秒平均连接计算数量是分布式方式的 4 倍左右, 这证明分布式方式浪费了大部分硬件处理性能. 由于硬件性能的损失, 流连接系统向外扩展需要大量 CPU 核, 使得分布式流连接系统部署成本和运维成本高企. 图 2 显示在给定流速时, 分布式方式所需 CPU 核心数量的预测曲线和实际测试结果. 实验使用固定为 180 或 300 s 的滑动时间窗口进行流连接, 假设每个 CPU 核心的连接计算速度为固定数值, 以不同流速所需的连接计算量估计所需 CPU 核心数, 再通过实际数据测试结果进行验证. 结果显示连接所需集群规模随流速的二次方快速增长, 进行大规模流连接所需的资源将远超我们 40 个节点 640 核的实验平台配置. 综上所述, 多核并行的流连接系统容易高效的扩展, 但并行度受限, 而向外扩展的分布式方式在提高并行度的同时会带来硬件处理效率严重下降.

针对流连接系统高效大规模扩展问题, 本文重点关注利用现场可编程门阵列 (field programmable gate array, FPGA) 并行加速流连接的方法. 一方面, 对于寄存器转换级电路 (register transfer level, RTL) 容易实现的连接谓词而言, 相比分布式扩展, FPGA 具有实现大量专用连接核电路的优势, 且几乎没有额外的框架开销. 与投入大量 CPU 核进行扩展的分布式系统比较, 基于 FPGA 实现大规模并行的性价比优势明显. FPGA 可编程的特点提供灵活部署的能力, 允许调整电路设计以实现不同的连接谓词. 另一方面, 并行流连接系统具有多种优点, 但难以增加 CPU 和内存实现扩展, 而为节点增加 FPGA 加速器外设则容易实现向上扩展 (scaling-up). FPGA 加速器利用流水并行方式可以同时处理大量元组, 并有效减少访存, 更具有低功耗的优势. 综合这些优势, FPGA 是非常适合流连接计算的加速器平台.

然而, 基于 FPGA 大规模并行化流连接计算面临一些困难. 不同于通用处理器 CPU, 专用的 FPGA 连接计算核不具备独立的访存通道, 存在大量并行化连接核的访存瓶颈问题; 大量连接核并行运行还会降低系统频率, 如何有效组织这些连接核成为关键; 现有流连接计算控制逻辑由流连接系统程序管理, 可精确地保证流连接完整性, 而 FPGA 并行计算没有这一条件, 难以实现连接完整性.

1) DiDi Chuxing GAIA Initiative. https://gaia.didichuxing.com.

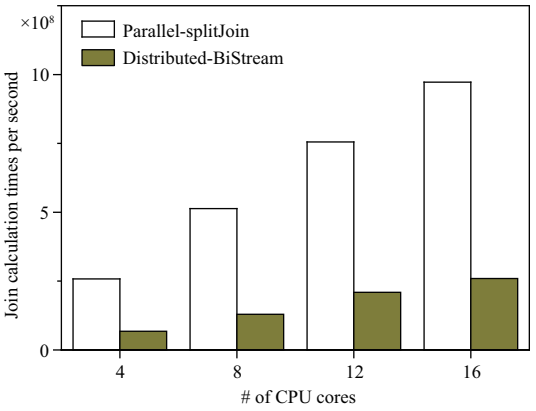

图 1 (网络版彩图) 分布式与并行扩展方式连接计算速度对比

Figure 1 (Color online) Comparison of join speed between distributed and parallel extensions

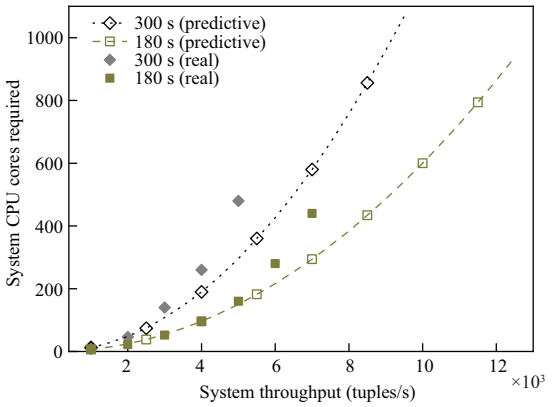

图 2 (网络版彩图) 扩展吞吐所需的 CPU 核心数

Figure 2 (Color online) CPU cores required corresponding to different throughputs

针对这些问题, 本文提出并设计实现了一种 FPGA 流连接并行加速器 FJoin, 它由多核的主机与多块具有独立访存通道的 FPGA 共同组成. 每个 FPGA 中使用大量基本连接单元串联形成深度连接流水线, 单元中包括自定义的连接谓词. 连接流水线通过一次载入一条流的多个元组, 在另一条流的连接窗口元组流经流水线时所有单元进行高并行流动连接. 一个基本连接单元直接作为流水线的一段, 它无需全局控制信号, 因而流水线可以大规模地线性扩展. 剩余的 FPGA 资源放置必要的流水线控制与访存模块, 由于一条流水线只需在头部设置一个访存接口, 节约了访存资源的使用. 考虑到 FPGA 并行连接难以保证完整性, 在系统主机部分为每条流水线分配一个托管线程, 使 CPU, FPGA 协同工作. 主机的连接完整性控制模块将流划分为连接计算任务 Task, 连接任务调度器将 Task 分配给各托管线程, 托管线程将 Task 执行所需数据同步至 FPGA 设备内存后, 运行连接流水线. 协同工作使得复杂的连接过程控制大部分由主机 CPU 完成, 简化了系统设计. 这种并行加速框架不仅能突破多核并行流连接系统的扩展限制, 同时还能避免分布式向外扩展的硬件性能浪费.

我们在配备 FPGA 加速器卡的设备上实现了 FJoin, 使用两个真实数据集 (网约车平台流数据、骨干网流量轨迹流数据) 和相关的连接谓词, 与部署在 40 个节点集群上的目前最好的分布式流连接系统进行性能对比. 实验结果表明本文提出的加速器 FJoin 可在单一 FPGA 加速器卡上达到 5 倍的系统吞吐, 将连接计算速度提升 16 倍, 同时保证连接完整性, 满足实时流处理的时延要求.

总结来说, 本文的主要贡献包括:

(1) 提出一种 FPGA 流连接并行加速器 FJoin, 设计并实现了 FPGA 高并行的流动连接方法, 利用基本连接单元串联形成的深度连接流水线, 有效解决了 FPGA 中连接计算大规模并行化的问题;

(2) 充分利用 CPU-FPGA 协同工作, 将连续流连接计算划分为独立任务, 使复杂过程控制大部分由 CPU 完成, 在高效的并行连接计算的同时, 保证了流连接的完整性;

(3) 使用真实数据集对 FJoin 进行性能评估, 实验结果表明, 相对于目前最好的分布式流连接系统, FJoin 显著提高了系统并行度, 优化了系统吞吐, 且无需编写繁琐的并行化代码.

本文其余部分组织如下. 第 2 节介绍流连接系统的相关工作. 第 3 节详细介绍 FJoin 的架构与设计. 第 4 节介绍实现 FJoin 的一些细节. 第 5 节评估系统性能. 第 6 节为讨论, 第 7 节为结论.

2 相关工作

现有的流连接系统主要分为基于多核并行的流连接系统和分布式流连接系统.

并行流连接系统利用多核架构加速连接处理, 容易将并行度扩展至 CPU 核数, 并保持各单元的同步. Kang 等 [11] 首先提出了 3 步流连接算法, 成为连接两条持续数据流的基础方法. 通过扫描、加入、淘汰 3 个步骤, 可以保证来自两条流的每个元组对仅连接一次. Gedik [12] 等将 3 步流连接算法应用于多核系统以获得加速效果. 但是连接窗口整体存储只适合于统一访存的架构, 在广泛使用的 NUMA 架构中, 跨越不同核心访问内存的开销是不可接受的. Teubner 等 [3] 提出了握手连接 (handshake join) 模型, 使两条流以相反的顺序流经多个 CPU 连接核, 当每对元组处于同一核心时进行连接操作. 这种设计避免了多核访存的冲突, 充分利用 NUMA 架构的优势, 然而元组排队传递使得处理延迟高企. Roy 等 [13] 之后的改进工作使得元组无需在队列中阻塞等待, 可以立即传递, 有效降低了连接延迟. 但是这一方法仍存在元组跨核心移动时需要复杂的控制协议的问题, 浪费了一部分 CPU 性能, 限制了系统吞吐. Najafi 等 [4] 创新地提出了 Split Join, 它使用单向数据流替代握手连接的双向数据流. Split Join 将所有元组以同一全局顺序广播到所有 CPU 核, 每个核只保存流的一部分, 彼此独立进行连接计算. 这种方法充分发挥了现有多核并行结构的性能, 但由于资源限制, 并行流连接系统无法应对大规模的流数据.

由于并行方式可扩展性有限, 近年的研究主要关注分布式流连接系统. Elseidy 等 [6] 提出了一种基于矩阵连接模型 (join-matrix) 的分布式流连接系统. 它将多个处理节点组织成一个矩阵, 一条流划分到矩阵的各行存储, 另一条划分到矩阵的各列, 两条流的元组对总会在矩阵中一个节点完成连接. Fang 等 [14] 研究了如何组织矩阵以最大化系统吞吐和优化资源利用效率. 但是, 这一方法由于内存中存储的窗口元组拷贝数等于矩阵行数或列数, 内存利用效率低, 且难以扩展. 针对此问题, Lin 等 [7] 提出了一种基于二部图的连接模型 (join-biclique), 它将一条流分组存储在一组节点中, 另一条流分组存于另一组节点, 同时流元组发送到对立流的那组节点中进行连接. 这种连接模型不会产生窗口元组的拷贝, 易于伸缩扩展, 是当前最先进的分布式流连接模型. Zhang 等 [8] 提出 BiStream 系统在处理等值连接谓词时, 会在计算节点之间产生严重的负载不均. 其设计了一种轻量化的指数计数方法区分出流连接中的热键, 对热键使用广播策略, 对其他键使用哈希分发策略, 能够有效均衡负载. Zhou 等 [2] 提出了流连接哈希分发策略的数据倾斜问题, 分析了其对连接负载的影响, 并设计了一种轻量化的负载迁移策略. Gulisano 等 [15] 提出流连接中数据流可能来自多个平行的流源, 影响连接结果的完整性, 并设计了一种无锁的多线程同步算法对数据流进行排序. 尽管分布式流连接系统可以大规模扩展, 但最新的一些研究也指出独立于硬件设计分布式流处理系统导致性能的降低, 这也影响在其之上运行的流连接系统. Zeuch 等 [10] 的研究显示由于采用 Java 虚拟机实现平台独立性等原因, 当前的分布式流处理系统无法充分利用多核处理器和高速网络的硬件性能. Zhang 等 [16] 的调查报告了最近基于硬件架构考虑的流处理系统设计.

最近的一些工作在 FPGA 加速流处理方面进行了有益的探索. Najafi 等 [17] 提出一种多路流连接的循环流水线, 其将每条数据流和连接谓词关联到一个 FPGA 上的处理核心, 所有处理核之间通过两条数据环路相连. 由于处理核数量始终等于数据流和连接谓词的数量, 这种环路设计只能针对大量的数据流进行扩展, 但不能在两条流连接时针对大规模流数据进行扩展. Thomas 等 [18] 设计了一个基于 FPGA 高度并行的流处理框架, 它只处理一条数据流, 可以实现映射、过滤、聚合等流处理算子, 但不能进行连接操作. Wu 等 [19] 在边缘服务器中设计了 FPGA 加速的流处理系统, 但不包括易成为瓶颈的连接算子.

表 1　标识符

Table 1　Notations

Notation	Description	Notation	Description
R, S	Stream relation	r, s	Stream tuple
t_R, t_S	Stream speed	Win$R$, Win$S$	Join window
T	System throughput	P	Join predicate
ϖ	Sliding time window size	γ	R/S stream speed ratio
C	Calculate cost	n_b	Number of basic join units
η	Clock cycle utilization rate		

3　基于 FPGA 的流连接加速框架

3.1　系统整体架构

本文描述的流连接遵循相关文献 [7] 中的定义, 并使用表 1 列出的符号以便说明.

为了连接两条持续不断的数据流 R 和 S, 需要在内存中保存 R 和 S 一段时间内到达的元组, 这部分流元组称为连接窗口 Win$R$, Win$S$. 新到达的流元组与对立流的连接窗口进行连接, 使用给定的连接谓词 P, 在每个元组对之间进行判断, 符合条件时产生连接结果元组. 由于流元组继续加入连接窗口, 连接谓词的计算量与两条流流速的乘积 $t_R \cdot t_S$ 相关, 成为计算负载最重的部分. 除了连接谓词的计算, 流连接系统还需要保证连接完整性, 避免连接结果的重复或遗漏. 最理想情况下, 连接结果仍依循输入流原有顺序输出, 满足保序输出.

针对流连接的以上要求, 我们设计了一种高并行的流动连接方法, 以解决大量专用的 FPGA 连接谓词计算核不具备独立访存通道的问题. 由于到达时间紧邻的流元组所对应的连接窗口高度重合, 可以共同与一个扩大的连接窗口连接, 这一特点为并行提供了机会. 如若每个 FPGA 计算单元保存一个流元组, 当加速器具有大量并行计算单元时, 连接窗口元组在所有单元中流动一遍, 即可完成这批流元组的连接计算. 为此我们设计了一种深度连接流水线, 它能串联大量基本连接单元, 各单元在连接窗口元组流动时独立计算元组对的连接谓词, 产生并提交连接结果. 为实现线性扩展, 基本连接单元没有全局的控制信号, 只与上下游单元相连, 自然地成为流水线的一段. 然而, 高并行流动连接需要确定一个扩大的连接窗口, 这需要将连续的流连接计算划分为独立的小批量任务, 增加了额外的过程控制, 较难保证连接完整性. 考虑到 FPGA 控制连接过程会占用并行计算的逻辑资源, 我们将计算任务划分的复杂逻辑交由主机 CPU 完成, 由 FPGA 配合对结果流加以过滤. 这简化了系统设计, 在高效地并行化连接计算同时, 保证了流连接的完整性.

FJoin 整体架构设计如图 3 所示, 系统分为上半部分的主机侧软件和下半部分设备侧硬件. 硬件部分包括每个 FPGA 中的流水线状态机 (finite state machine, FSM)、访存接口 (Mem I/F), 以及由大量基本连接单元串联组成的深度连接流水线. 软件部分包括流连接任务调度器 (task scheduler)、生成连接计算批量任务的连接完整性控制模块 (completeness controller)、结果后处理模块 (post-process module), 以及每条 FPGA 连接流水线对应的一个托管线程.

系统运行时, 流数据输入连接完整性控制模块 (completeness controller), 以 R, S 流新到达系统的一小批元组生成任务 Task. 连接完整性控制模块在主机内存中存储全部流元组, 将不同流源发送的元组加入存储, 并确定 Task 连接窗口范围. Task 信息中包括待处理流元组的地址、数量, 以及需要连接的窗口范围. 流连接任务调度器 (task scheduler) 依据轮询策略, 将 Task 中 R 和 S 的流元组分开调度

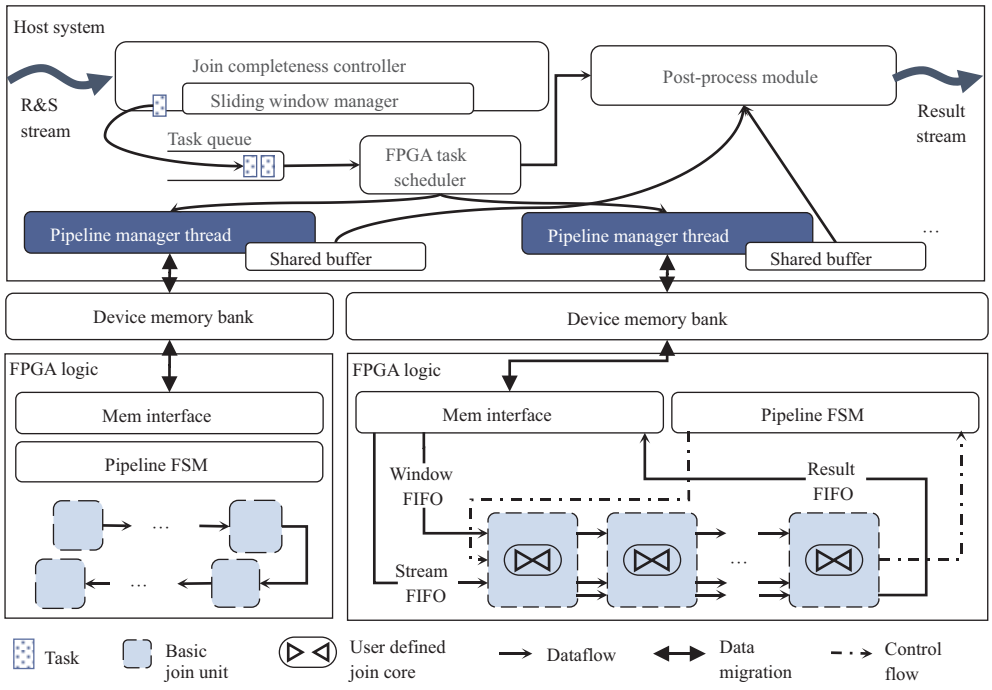

图 3 (网络版彩图) FJoin 整体结构

Figure 3 (Color online) FJoin architecture

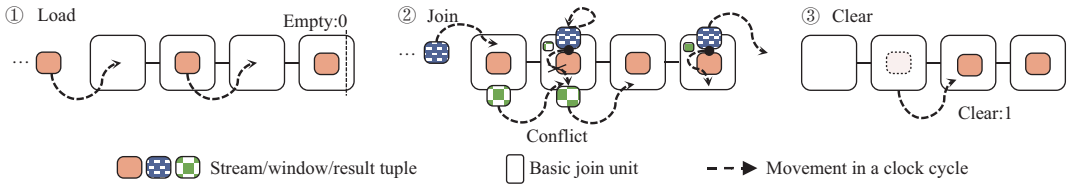

图 4 (网络版彩图) 连接流水线运行步骤

Figure 4 (Color online) Join pipeline operations

给两条流水线的托管线程, 对于非对称的连接谓词, 需要区分硬件中的流水线所处理的流. 托管线程控制 Task 的计算状态, 并控制计算所需的数据在主机和设备内存间的传输, FPGA 则接受指令和数据进行本地计算. 根据 Task 的信息, 流元组和连接窗口从主机内存增量同步至设备内存, 并在数据迁移完成后启动连接流水线. 连接结果同样由托管线程迁移回主机内存, 并在共享的缓冲区交付给结果后处理模块 (post-process module). 结果后处理模块将各个托管线程交付的连接结果进行属性恢复和排序, 生成保序输出的最终结果流.

连接流水线的运行主要分为载入、连接、清空 3 个步骤, 如图 4 所示. 运行时, 第 ① 步载入将 Task 中待处理的多个流元组从流水线头部逐级流入, 保存在每个单元中, 填充整个流水线. 第 ② 步连接读取 Task 确定的设备内存中另一条流的连接窗口, 在窗口元组流经每个基本连接单元时, 与第 ① 步保存的流元组进行连接计算, 如果产生结果元组则放入本单元的结果寄存器当中, 再逐级传递到流水线尾部出口. 共用的结果寄存器可能导致冲突, 如同图中 Join 步骤的左二连接单元, 连接流水线应对冲突保证结果完整性的设计在 3.2 小节介绍. 第 ③ 步清空是在所有窗口元组连接完毕后, 传递 clear

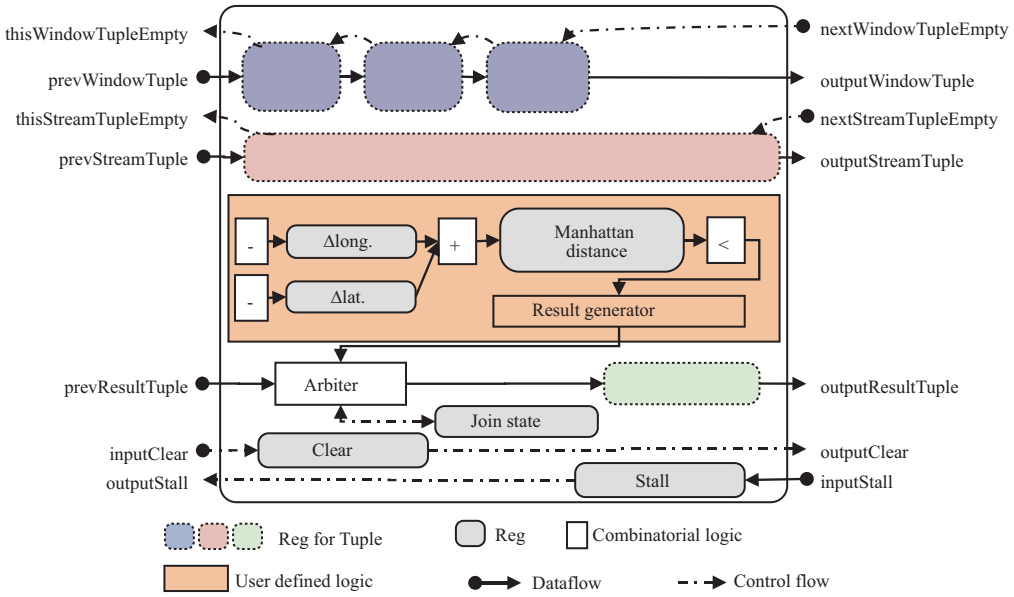

图 5　(网络版彩图) 基本连接单元

Figure 5　(Color online) Basic join unit

信号清空所有单元. 基本连接单元数量可用于评估连接流水线的处理能力, 记录在对应的托管线程中, 以便将 Task 调度到不同的连接流水线上执行. 通过访存接口 (Mem I/F), 连接流水线访问一片独立的设备内存, 读写流元组、连接窗口元组和结果元组. 流水线状态机 (FSM) 用于接受托管线程发送的指令, 控制流水线的启停和运行.

以下 3.2~3.4 小节详细介绍连接流水线、完整性控制模块、后处理模块 3 个部分的设计.

3.2　流动连接的基本单元

实现连接计算的最重要部分是 FPGA 芯片中的连接流水线, 其设计面临的主要难题是在大规模地扩展以充分利用资源的同时, 正确地完成所有连接计算. 我们将大量的基本连接单元串联组成一条深度流水线, 使每个单元只与上下游的单元相连, 消除所有全局的控制信号. 这使得深度流水线容易增加单元数量, 实现线性扩展, 并通过一个单元的资源使用情况, 预估在所使用的 FPGA 中能够复制的单元数量. 流水线中的基本连接单元无需独立访存, 所需数据从流水线的头部逐级传递, 这样一条流水线中的大量单元只需共用一个访存通道.

基本连接单元的结构如图 5 所示, 其中包含可使用 RTL 自定义的连接逻辑, 从前往后传递的流元组 (StreamTuple)、窗口元组 (WindowTuple)、结果元组 (ResultTuple) 三条流, 逐级传递的 clear 和 stall 控制信号, 以及一个连接状态寄存器. 流水线运行时首先需要载入 Task 中待处理的流元组. 流元组从内存读取后, 从流水线头部开始, 通过每个基本连接单元中的流元组寄存器逐级传递. 当下一个单元的流元组寄存器为空时, 本单元保存的流元组流动至下一单元. 从载入状态开始, 输入流水线最后一段的相关 empty 信号保持为 0, 使得读入的流元组最终保存在各单元的流元组寄存器中.

流元组载入完成后, 开始连接计算, 这一步连接窗口元组流经所有基本连接单元, 需要保证流水线的连接结果不重复和遗漏. 在连接步骤中, 基本连接单元每个时钟周期的行为逻辑如算法 1 所示, 该算法以串行过程描述了基本连接单元处于连接步骤时单周期并行的处理逻辑. 由于基本连接单元

Algorithm 1 Serialized description of the behavior of the basic join unit during a single clock cycle in the join step

Input: nextWindowTupleEmpty, prevWindowTuple, prevResultTuple, inputStall.
Output: thisWindowTupleEmpty, outputWindowTuple, outputResultTuple, outputStall;
Local Regs: joined, thisWindowTuple, thisResultTuple, thisStageStall.

```
1: windowTupleStall ← false, outputWindowTuple ← null, thisResultTuple ← null;
   /*Transfer the result stream tuple continuously through the result tuple registers in basic join units.*/
2: if prevResultTuple is valid then
3:    thisResultTuple ← prevResultTuple;
4: end if
5: if thisWindowTupleEmpty = true AND prevWindowTuple is valid then
6:    thisWindowTuple ← prevWindowTuple, thisWindowTupleEmpty ← false;
7: else if thisWindowTupleEmpty = false then
8:    windowTupleStall ← ! nextWindowTupleEmpty;
   /*Submitting newly generated join result tuple to the result tuple register of this unit after arbitration.*/
9:    if joined = false AND JC produced jcJoinResult then
10:      if prevResultTuple is valid OR thisStageStall = true then
11:         windowTupleStall ← true;
12:      else
13:         thisResultTuple ← jcJoinResult;
14:         joined ← ! nextWindowTupleEmpty;
15:      end if
16:   end if
17:   if windowTupleStall = false then
18:      outputWindowTuple ← thisWindowTuple, thisWindowTuple ← null;
19:      thisWindowTupleEmpty ← true, joined ← false;
20:   end if
21: end if
22: outputStall ← thisStageStall, thisStageStall ← inputStall.
```

并行计算连接谓词, 产生的结果元组通过结果元组寄存器传递, 有限数量的寄存器会导致寄存器占用冲突. 为减少冲突, 每个时钟周期, 结果元组寄存器的值都送至下一个基本连接单元, 并接收一个新的结果 (第 2∼4 行). 与结果元组传递不同, 窗口元组在本单元窗口元组寄存器空时, 从上游相邻单元流入 (第 5, 6 行). 每个时钟周期, 单元中流经的窗口元组与保存的流元组满足连接条件时, 产生一个连接结果元组, 经过仲裁 (arbiter) 提交给本单元的结果元组寄存器 (第 9∼16 行). 结果元组寄存器的写入仲裁保证了上一个单元传递的结果比本单元新产生的结果优先写入. 即两个来源的元组冲突时, 丢弃本单元新产生的结果, 这时需阻塞窗口元组流, 否则将遗漏掉这个连接结果 (第 10, 11 行). 任何一处发生窗口元组流的阻塞后, 这一阻塞都会使得前向的窗口元组寄存器陆续被阻塞, 直至流水线头部入口, 使得窗口元组的读取量减少. 如若窗口元组已在当前所处的单元提交过连接结果, 却因下一单元被阻塞无法流动, 则连接状态寄存器记录完成结果提交的状态, 避免重复的连接计算 (第 14 行). 由于结果元组流的持续传递, 结果元组寄存器的冲突会在若干个周期后消失, 这种因冲突导致的阻塞终会恢复. 连接步骤在所有窗口元组和结果元组传递到流水线尾部出口后结束, 随后的清空操作中, 逐级传递 clear 信号清空所有单元.

对于较复杂的连接谓词, 可能需要划分多个时钟周期计算, 以获得更好的时序. 在基本连接单元中, 可以通过配置多个窗口元组寄存器将复杂的逻辑分段, 但暂存的中间结果需要与对应的窗口元组在时钟周期上对齐. 图 5 中以滴滴系统订单匹配中计算两个 GPS 位置经纬度的曼哈顿距离 (Manhattan

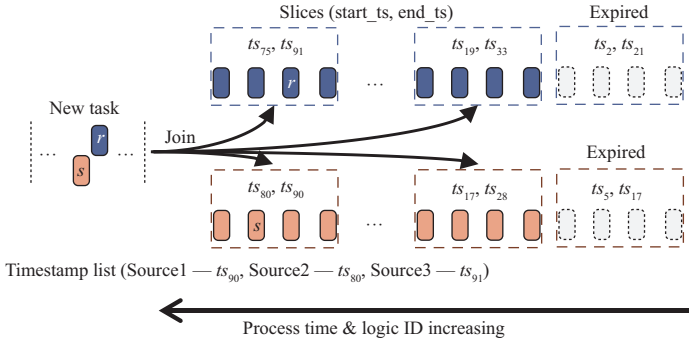

图 6　(网络版彩图) 连接完整性控制模块的主要数据结构

Figure 6　(Color online) Main data structure of join completeness controller

distance) 为例, 每个基本连接单元配置有 3 个窗口元组寄存器. 连接逻辑第 1 段分别计算经度和纬度的差值, 在保存中间结果的同一时刻, 对应的窗口元组从首个寄存器流至第 2 个寄存器. 连接逻辑第 2 段对两个差值的绝对值求和, 保存得到的曼哈顿距离, 同时对应的窗口元组进入第 3 个寄存器. 最后一段只进行小于阈值的判断, 满足连接条件时, 从第 3 个窗口元组寄存器生成连接结果元组.

FPGA 的大部分资源用于实现更多单元的连接流水线, 但仍需一部分资源用于访存和流水线状态机的实现. 流水线状态机 (FSM) 用于接受主机的指令, 控制流水线的运行. 当连接结果不能及时送至设备内存时, 为避免后续的连接结果丢失, 流水线尾部的 stall 信号输入需置为 1 暂停流水线, stall 传递至所有单元阻塞新的连接结果提交 (第 10, 22 行). 如若 Task 中待处理流元组数量多于流水线基本连接单元数, 一次连接无法完成, 则 3 个步骤需要重复进行, 直至 Task 连接计算完成.

3.3　连接计算分批的完整性

高并行流动连接可以发挥 FPGA 并行计算优势, 但由于 Task 需要确定扩大的连接窗口, 流元组的连接并非逐个进行的. 主机的连接完整性控制模块将连续的流连接计算进行粗略划分, 每个 Task 包括一小批新到达的流元组和它们共同的连接窗口范围, FPGA 配合对结果流精确地加以过滤. 这一设计高效地保证了连接完整性, 同时避免了 FPGA 控制复杂连接过程占用并行计算的资源.

生成 Task 时, 需要仔细考虑连接窗口的边界. 如图 6 所示的例子, 假设来自两条流的 $r \in R$ 和 $s \in S$ 两个元组相继到达, 它们被划入同一个 Task, 但并非立即连接. 随着更多新元组到达, 该 Task 会被调度到一对托管线程执行. 该 Task 连接窗口的最新边界将由最后到达的元组确定, 在其之前的 R 流元组都属于 S 流对应的连接窗口, 在其之前的 S 流元组则属于 R 流对应的连接窗口. 此时, 虽然 s 元组在 r 元组之后到达, 但为 Task 中包括 r 元组在内的所有 R 流元组确定的连接窗口中却包含元组 s, 同时 s 元组对应的连接窗口中也包含元组 r. 在 FPGA 的计算中, 两条连接流水线上都会产生 r 和 s 的连接结果, 这是扩大连接窗口导致的重复连接问题.

为了解决这一问题, 需要基于逻辑时间戳来确保完整连接. 在所有流元组进入系统时, 完整性控制模块为其分配一个元组 id 作为逻辑时间戳, 这一 id 确保元组具有一个全局的到达顺序. 元组 id 是持续自增的 32 位整数, 但随着系统长时间高吞吐运行, 自增的整数会发生溢出. 所以元组 id 的最高位作为标记位, 当元组 id 发生自增溢出后, 完整性控制模块向连接流水线发送新的标记位表示一轮循环. 理想情况下, 流元组只与在其之前到达的元组连接, 跳过元组 id 顺序靠后的窗口元组, 就可以避免重复连接. 但这需要为每个基本连接单元增加元组 id 比较模块, 资源开销较大. 考虑到这种重复连接只在 Task 生成的时间范围内出现, 占比并不高, 所以基本连接单元不进行元组 id 的比较, 而是在结

果流出口处逐一检查结果是否满足逻辑时间戳的条件, 过滤掉重复的结果.

如若两条流所有元组都保存在内存中, 上述做法可实现流元组实际时间戳任意排列的完整连接. 但是, 保存全历史的流连接无法持续运行, 随着元组的数量增长, 必须以适当的方式丢弃元组来保证实时处理. 最广泛使用的流连接窗口是滑动时间窗口 [7,11], 即给定时间窗口大小 ϖ, 对于任何流元组 $r \in R$ (对应 $s \in S$), 不满足时间戳条件 $0 \leqslant r.\text{time} - s.\text{time} \leqslant \varpi$ (对应 $0 \leqslant s.\text{time} - r.\text{time} \leqslant \varpi$) 的元组 $s \in S$ (对应 $r \in R$) 不会被连接, 这一约束条件可作为元组过期丢弃依据.

流连接语义中, 需要连接的两条流 R 和 S 可能来自多个流源, 每个流源输入物理时间戳递增的元组. 窗口元组虽然按照元组 id 顺序在内存中连续排布, 但实际时间戳可能因为多个流源的物理时钟并非完全同步, 或是网络传输乱序的原因打乱顺序. 为了确定滑动时间窗口的边界, 完整性控制模块需要同步多流源之间的时间戳并维护滑动时间窗口, 图 6 展示了其主要数据结构. 多流源的时间戳同步机制要求所有流源在完整性模块注册, 并定期在数据流中嵌入时间戳信号元组, 完整性模块维护来自每个流源的最新时间戳信号表 (timestamp list). 如果所有流源发来的最新时间戳都到达某个时间点, 如信号表中最小时间戳 89, 则流连接系统肯定已经接收时间戳 89 之前的所有元组. 基于这一条件, 完整性模块在将窗口划分为多个分片 (slices), 在每个分片中使用第 1 个元组到达时, 时间戳信号表里的最小时间戳标注起始点, 即 start_ts, 使用最后一个元组到达时信号表里的最大时间戳标注终止点, 即 end_ts. 两者表示出该分片包含元组的时间范围. 由此可以确定 Task 的连接窗口范围, Task 内最早的元组到达系统时, 信号表的最小时间戳减去 ϖ 所对应的窗口分片, 就是 Task 连接窗口的最旧边界. 更早的窗口分片已经不会再与未来到达的流元组连接, 但与正在执行的 Task 相关的分片暂时无法从内存丢弃, 不涉及任何 Task 的过期元组分片 (expired slices) 才可以安全的丢弃.

通过元组分片确定的连接窗口最旧边界的粒度为一个分片, 并不完全准确, 可能产生少量不满足滑动时间窗口约束条件的连接结果, 为了提供精确的时间窗口约束检查, 在连接流水线的结果流出口处设置有滑动时间窗口约束检查模块. 当需要精确时间窗口时, 主机可以将窗口大小 ϖ 设置到窗口检查模块, 从而过滤掉不满足窗口约束条件的结果. 由于 Task 生成时使用到的总是冗余的窗口范围, 所以不会有连接的遗漏.

3.4 结果后处理

FPGA 连接流水线产生的连接结果保存在设备内存中, 由托管线程迁移回主机内存. 由于已经过滤了重复的结果和不满足滑动时间窗口约束条件的结果, 迁移回主机内存的是完整的连接结果集合. 结果后处理模块需要收集所有托管线程交付的连接结果流, 按照 Task 任务队列的原有顺序产生最终结果流. 此时后处理模块将可选地对结果加以额外处理, 其主要实现保序输出和结果恢复.

保序输出. FPGA 连接流水线的多个基本连接单元并行计算, 流水线设计只能保证每个连接单元产生的结果顺序地添加进结果流, 而来自不同单元的结果之间并非顺序的. 如果要求输出的结果流依循输入元组顺序, 则需对一个 Task 中各个元组产生的连接结果重新排序. 开启保序输出时, 流元组将增加流元组顺序编号 seq 字段, 该字段标明流元组在 Task 多个流元组中的序号. 基本连接单元产生连接结果时, 将流元组的 seq 字段复制到结果元组中, 以便后处理模块区分不同流元组产生的结果. 算法 2 显示了后处理模块对结果流元组重新排序的具体过程. 具有相同 seq 的元组顺序是正确的, 而不同 seq 对应于不同的流元组产生的结果, 应当从小到大重新排序 (第 5~20 行). 且来自 Task 中 R 流的连接结果和 S 流的连接结果之间需要依据 seq 最后进行归并排序 (第 3 行).

结果恢复. 基本连接单元中包含保存流元组、结果元组的寄存器, 以及多个保存窗口元组的寄存器, 因此读入 FPGA 中的元组位宽过大会对基本连接单元的资源占用量产生不利影响. 在连接操作中,

Algorithm 2 Process of sorting result stream of Task to get order preserving results

```
Input: rStreamResults, sStreamResults.
Output: sortedResultStream.
 1: rSortedResults ← GetSortedResultStream(rStreamResults);
 2: sSortedResults ← GetSortedResultStream(sStreamResults);
    /*Merge sort two ordered result streams according to the tupleSeq field.*/
 3: sortedResultStream ← MergeSort(rSortedResults, sSortedResults);
 4: return sortedResultStream;
    /*Sort the result set according to the tupleSeq field of each result tuple.*/
 5: function GetSortedResultStream(joinResults)
 6:     for i = 0 → joinResults.getLength() − 1 do
 7:         tupleSeq ← joinResults[i].getTupleSeq();
 8:         if tupleSeq ⩾ Buckets.getBucketNum() then
 9:             Buckets.addBuckets(tupleSeq − Buckets.getBucketNum());
10:         end if
11:         Buckets[tupleSeq].add(joinResults[i]);
12:     end for
13:     for i = 0 → Buckets.getBucketNum() − 1 do
14:         for j = 0 → Buckets[i].getTupleNum() − 1 do
15:             sortedResults.add(Buckets[i][j]);
16:         end for
17:         Buckets[i].clear();
18:     end for
19:     return sortedResults;
20: end function
```

与连接谓词相关的元组字段属于关键字段, 而与连接谓词无关的字段无需全部读入到 FPGA 中. 通过事先根据需要对元组进行裁剪, 保留关键字段, 降低基本连接单元的资源开销. 待连接完成后, 后处理模块再对连接结果元组进行恢复. 开启元组裁剪与结果恢复时, 完整性控制模块会将到达系统的元组存储两次. 一组简单按照元组 id 顺序在内存中连续排布, 不区分来自 R 流或 S 流, 以便依据元组 id 进行寻址. 该组保留元组的原始内容, 用于后处理模块恢复结果元组. 另一组也按照元组 id 顺序存储, 但将 R 流和 S 流的连接窗口分开存储, 并根据分片维护滑动时间窗口. 该组只保留元组的关键字段, 用于 FPGA 的连接计算. FPGA 产生的连接结果对中包含被连接的两个元组 id, 后处理模块使用两个 id 在第 1 组存储中查找元组的原始内容, 从而完成结果元组的恢复.

3.5 性能预估方法

对于固定滑动时间窗口长度的通用连接算法, 需要计算流元组与连接窗口的所有元组对, 流连接的计算开销与时间窗口大小和流速相关. R 流流速 t_R 带来的计算开销应表示为 t_R 与时间窗口 ϖ 中 S 流到达的所有元组 $\sum_{i=0}^{\varpi} t_{Si}$ 相乘, 流连接的总体开销是两条流计算开销的和, 表示为

$$C = t_R \sum_{i=0}^{\varpi} t_{Si} + t_S \sum_{i=0}^{\varpi} t_{Ri}. \tag{1}$$

式 (1) 以离散时间片求和表示连接窗口, 难以计算, 为了简化讨论, 假设两条流的流速固定为平均值, 流速之比固定为 $t_R/t_S = \gamma$, 系统整体吞吐记为 $T = t_R + t_S$. 由此可以得到式 (2), 连接窗口以平均流速与时间窗口大小的乘积表示, 这时两条流的计算负载是对称相等的, 流连接整体开销与窗口大

小 ϖ 和吞吐的二次方 T^2 相关.

$$C = t_R \cdot \varpi t_S + t_S \cdot \varpi t_R = 2\varpi t_R t_S = \frac{2\varpi\gamma}{(\gamma+1)^2} T^2. \tag{2}$$

为了预估分布式流连接系统性能, 假设各个 CPU 核的计算性能 C_{cpu} 是相当的, 则可以通过估计 C_{cpu}, 建立所需 CPU 核数量与系统吞吐之间的预估模型. 如若实测得知, 具有 n_1 核数的系统无法持续处理流速大小为 T 的流, 表示为计算性能 $n_1 \cdot C_{\text{cpu}}$ 小于流连接所需, 而具有 n_2 核数的系统可以满足 T 的吞吐. 则使用 n_1, n_2 的平均数估计 C_{cpu}, 表示为式 (3). 变形等式, 以此预估系统吞吐 T 对应所需的 CPU 核数量 n', 如式 (4) 所示.

$$n_1 \cdot C_{\text{cpu}} \leqslant \frac{2\varpi\gamma}{(\gamma+1)^2} T^2 \leqslant n_2 \cdot C_{\text{cpu}}, C_{\text{cpu}} \approx \frac{4\varpi\gamma T^2}{(n_1+n_2)(\gamma+1)^2}, \tag{3}$$

$$n' = \frac{2\varpi\gamma T^2}{(\gamma+1)^2 C_{\text{cpu}}}. \tag{4}$$

在 FPGA 流连接系统中, 连接流水线性能随基本连接单元数量 n_b 线性扩展, 最大计算性能以主频和单元数乘积表示为 fn_b. 但系统设计导致并非所有时钟周期都用于连接计算, 例如基本连接单元为了等待下一单元的窗口元组寄存器清空会损失至少 50% 的周期, 流水线运行的载入和清空状态操作也会损失部分周期. 假设以 η 表示时钟周期整体利用率, 则系统整体性能 $\eta f n_b$ 应当大于流连接计算所需, 由此得到的式 (5) 可以用于估计系统吞吐 T 的上限. 该式中假设各条连接流水线性能的和即为系统整体性能, $\eta f n_{bi}$ 计算第 i 条连接流水线的计算性能, n_p 条流水线性能的和为 $\eta f n_b$.

$$C = \frac{2\varpi\gamma}{(\gamma+1)^2} T^2 < \eta f n_b = \sum_{i=1}^{n_p} \eta f n_{bi}, \quad T < \sqrt{\frac{\eta f n_b (\gamma+1)^2}{2\varpi\gamma}}. \tag{5}$$

4 系统实现

我们在配备 Xilinx Alveo 系列数据中心加速器卡的设备中实现了 FJoin, 它基于 Vitis 环境开发, 包括大约 2100 行 C++ 代码和 1900 行 Verilog 代码, 可在 GitHub 获取公开的源代码[2)].

Vitis 为 FPGA 加速应用开发者提供一种标准开发框架与运行时平台, 该框架包括 C++ 开发的主机程序和 RTL 开发的加速内核. FJoin 的主机侧软件作为框架中的主机程序实现, 设备侧硬件中每条连接流水线作为框架中的一个 RTL 内核实现. 主机系统运行多个线程, 分别用于托管连接流水线与实现功能模块, 各线程间以共享内存的形式交互数据. 在主机系统与部署在基于 PCIe 的加速器卡中的加速内核之间, 使用 Xilinx 运行时库 (XRT) 进行通信, 并提供设备内存管理和 FPGA 资源管理功能. 主机程序通过 OpenCL API 调用与加速器的运行时交互, 通过 XDMA 进行数据传输.

FJoin 主机程序基于 OpenCL 框架, 在初始化阶段为加速器卡中的多个 RTL 内核创建独立的异步命令队列. 流连接任务调度器和流水线的托管线程使用这些命令队列对 Task 进行调度. 托管线程在每次启动 FPGA 上的内核前, 需要提前完成内核参数设置和数据同步. Task 所包含的任务信息作为内核参数, 其中标量参数直接写入数值, 指针参数则传递指向设备内存缓冲区的指针, 流元组与连接窗口增量同步到这些设备内存缓冲区中. 为避免 XDMA 进行无效的内存拷贝, 连接窗口数据在内存中以 4 kB 为单位对齐, 这需要限制元组的位宽为 4 字节的整数倍.

2) https://github.com/CGCL-codes/FJoin.

我们使用异步的命令队列在托管线程等待内核计算结果时隐藏数据传输的开销. 因此托管线程能够支持多个 Task 排队执行, 以便在一个 Task 执行内核计算同时, 完成前后 Task 的结果传输和增量同步. 由于托管线程对 Task 任务信息中指向的数据是只读的, 所以完整性控制模块和所有托管线程之间共享主存中的连接窗口, 托管线程只额外保存 Task 的任务信息副本. 托管线程从 FPGA 设备读回的连接结果需要进行后处理操作, 该操作需要读写内存数据, 因此托管线程与结果后处理模块之间有多个主存的缓冲区用于数据交换. 连接流水线在返回连接结果时, 首先返回一个结果信息块, 该信息块包含连接结果位置和数量, 在第 2 次读取设备内存时返回连接结果. 由于连接窗口缓冲区和流水线输出的结果流在设备内存中都具有连续的地址, 连接流水线访问设备内存时均可采用连续突发传输. 使用片上的缓冲 FIFO, 每次突发读写的数据大小可达 AXI 协议支持的最大值.

FJoin 的流连接任务调度器使用轮询方式调度 Task 到托管线程处执行. 具体来说, 完整性控制模块将新接收的流元组放入一个缓存队列中, 使用一个游标指向最新处理的元组位置. 每次访问该缓存队列时, 先提取 60% 的已有元组到最新的 Task 中, 已有元组是指当前队列尾部到上一次游标位置的范围. 访问缓存队列还会等待一个固定时长以将新到达的元组也加入到最新的 Task 中, 这一时长约为流连接系统预期时延的一半. 在补充连接窗口范围信息后, 最新的 Task 进入任务队列中, 由任务调度器转移给托管线程执行. 任务调度器将托管线程排成循环队列, 使用一个游标指向下一个将被调度的线程. 从任务队列出队的 Task 调度给游标指向的线程后, 该线程保留一个 Task 信息副本, 游标指向下一线程, 以此完成轮询调度. 生成 Task 的固定等待时长与 Task 轮询调度方式保持每个 Task 所需的计算量大致相同, 这在各条流水线的基本连接单元数量相同时十分有效.

5 性能评估

我们使用两个真实数据集与构造的连接谓词对 FJoin 系统的性能进行了评估, 重点测试流连接的吞吐与时延指标. 评估结果与目前性能最好的分布式流连接系统 BiStream [7] 进行对比, 因为该系统也能够通过大规模向外扩展来提升系统处理能力.

5.1 实验配置

实验环境. 基于 FPGA 的实验环境为一台装备 4 个 14 核 Xeon Gold 5117 CPU, 16×32 GB 内存的服务器, 通过 PCIe Gen3×16 连接一张 Xilinx Alveo U280 数据中心加速器卡. 服务器安装有 Xilinx 运行时环境 (Xilinx Runtime), 使用 XDMA 平台与加速器卡交互. 基于分布式流连接的基准系统 BiStream 运行在具有 40 个节点的集群中, 每个节点装备 2 个 8 核 E5-2670 CPU, 8×8 GB 内存, 1000 Mbps 网络. BiStream 系统运行在 Storm 平台 [9] 上, 使用的 Storm 版本为 1.2.3, JDK 版本为 1.7.0. 我们将每个节点在逻辑上划分为 16 个 CPU 核, 因此最多可使用 640 个 CPU 核.

数据集. 实验使用了两个大规模的真实数据集, 如表 2 所示. 第 1 个数据集是滴滴出行盖亚计划的乘车数据集, 该数据集包含两条数据流, 第 1 条是出租车的实时 GPS 位置形成的轨迹, 包含约 30 亿条记录, 第 2 条是乘客的查询订单, 包含约 700 万条记录. 这些数据是 2016 年 11 月在中国四川省成都市采集的. 第 2 个数据集来自 WIDE 骨干网络数据包轨迹[3)], 使用 2020 年 12 月两个不同的采样点采集的数据作为两条数据流, 它们分别包含约 9000 万和约 1 亿 3000 万条记录.

连接谓词. 对于滴滴出行的数据集, 实验使用乘客订单流 ORDER 和出租车位置流 GPS 的经度、纬度 (long, lat) 作为连接的属性, 因为通常订单会派送给地理位置较近的出租车. 真实的订单调度是

3) WIDE Project. https://mawi.wide.ad.jp/mawi.

表 2 流数据集列表

Table 2 Statistics of the stream datasets

Dataset	#of tuples	Type
DiDi orders	3 B and 7 M	GPS position
WIDE backbone traffic	90 M and 130 M	Packet IP trace

十分复杂的, 我们考虑一个简化的查询: 当一个元组到达系统时, 输出滑动时间窗口内与该元组 GPS 位置的曼哈顿距离小于阈值 diff 的所有记录, 表示如下:

SELECT * **FROM** ORDER, GPS

WHERE ABS(ORDER.long − GPS.long) + ABS(ORDER.lat − GPS.lat) <diff.

对于 WIDE 骨干网的数据集, 实验使用两个记录点采样的流 R 和 S, 以 sourceIP 和 destinationIP 作为连接的属性. 使用一个简化的查询: 在一个较短的滑动时间窗口内, 输出两条流源 IP 或目的 IP 前缀相同的记录对, 表示如下:

SELECT * **FROM** R, S

WHERE R.sourceIP ⊕ S.sourceIP <diff **OR** R.destinationIP ⊕ S.destinationIP <diff.

实验设置. 实验测试了不同规模的系统. 我们在 FJoin 共实现了两条连接流水线, 调整的变量是基本连接单元总数, 以 FJoin-x 表示共有 x 个基本连接单元, 它们等分到两条流水线上. 在 BiStream 中, 调整的变量是用于连接计算的 CPU 核总数, 以 BiStream-y 表示有 y 个 CPU 核用于连接计算, y 的值不含连接计算以外用途的 CPU 核数量. 所有实验都使用固定长度的滑动时间窗口, 其长度在具体实验中有不同取值. 我们主要从系统吞吐和处理时延两个方面比较了 FJoin 和 BiStream 的性能. 其中, 系统吞吐被定义为单位时间两条流所有流源输入系统的元组总数, 两条流的流速比值 γ 在具体实验中有不同取值. 处理时延被定义为结果最终输出的时间戳与其连接的两个元组中最大时间戳的差值, 时延分布或平均时延由一百万个连接结果取得数值. 系统启动前期性能可能存在显著变化, 因此我们在系统稳定运行两个滑动时间窗口长度后统计数据. 时间戳是本地取得的, 服务器的时钟误差控制在 10 ms 以内. 除实时结果外, 实验会重复多次取得误差在 10% 以内的均值作为结果.

5.2 系统吞吐

我们首先评估了 FJoin 的连接处理能力, 并与 BiStream 系统相比较. 依照文献 [7] 中的方法实现了 BiStream 的完整性保证, 以便与之进行公平比较.

流连接系统处理能力的直接体现是单位时间完成连接计算的数量, 如式 (1) 表示. 图 7(a) 是 FJoin 与 BiStream 运行每秒完成连接计算数量的实时对比. 我们使用滴滴出行的数据集与相应连接谓词, 将滑动时间窗口大小 ϖ 设置为 180 s, 多次测试提高输入流速. 运行一段时间后, 连接窗口将包含时间窗口内到达系统的所有流元组, 此时新元组与连接窗口进行连接所需的谓词计算速度接近系统实时计算速度的最大值. 如若所需的实时计算量超出系统连接谓词计算速度, 就会导致处理时延持续性上升. 图 7(a) 结果表明, 具有 1024 个基本连接单元的 FJoin 每秒可以完成超 1000 亿次连接谓词的计算. 而运行在 40 个节点的集群中, 具有 512 个 CPU 核的 BiStream 每秒完成连接谓词计算数量在 60 亿次左右. 从连接谓词计算量角度, FJoin 取得的加速比约为 17. FJoin 取得加速效果的原因主要有两个. 其一是基本连接单元中的自定义连接核可以专为连接谓词的计算设计, 计算时钟周期少, 且在连接流水线设计中访存开销可以忽略. 其二是基于 FPGA 设计的加速器避免了运行时平台、JVM、操作系统开销导致的硬件计算资源浪费, 这些浪费致使分布式流连接系统即使使用主频更高的 CPU, 单核处理

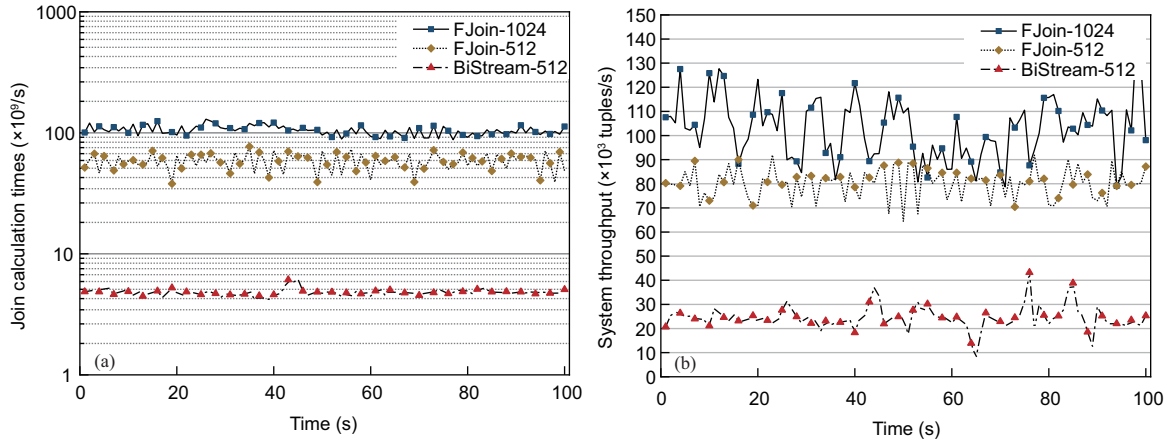

图 7 (网络版彩图) (a) 滴滴出行连接谓词的系统实时计算速度; (b) 网络流量轨迹连接 15 s 窗口下的系统实时吞吐

Figure 7 (Color online) (a) Realtime calculation speed of DiDi join predicate; (b) realtime system throughput of network trace join predicate in 15 s window

性能仍低于基于 FPGA 的基本连接单元. 为说明这一原因, 图 7(a) 表示了 CPU 核与基本连接单元数量相同时, FJoin 的连接谓词计算速度同样有 10 倍左右的提升.

图 7(b) 对比了 FJoin 与 BiStream 的实时吞吐. 该测试使用网络流量轨迹的数据集与相应连接谓词, 滑动时间窗口大小 ϖ 设置为 15 s. 我们同样分为多次测试提高输入流速, 测试系统能够承受的吞吐值. 相比具有 512 个 CPU 核的 BiStream 系统, 具有 1024 个基本连接单元的 FJoin 可达 5 倍的实时吞吐. 需要指出的是, 当滑动时间窗口大小固定时, 流连接所需处理能力与吞吐的二次方相关. FPGA 加速器的运行频率达到最大频率 300 MHz, 据测算周期利用率 η 在 0.31 左右. 损失的时钟周期主要是等待下一单元的窗口元组寄存器清空和等待流水线载入和清空状态操作, 在这些测试中结果元组阻塞导致的周期损失可以忽略. 另外, 对比同样 512 个单元的系统规模, FJoin 的吞吐提升至 4 倍左右. 相比 BiStream, FJoin 的实时吞吐曲线表现出更剧烈的波动, 这是因为 FPGA 系统需要累积一小批流元组才能生成 Task 调度执行, 而 BiStream 可以在每个元组到达时进行处理.

为了进行全面的对比, 我们分别使用两个数据集与相应谓词, 将系统规模从 16 至 1024 单元取 7 种不同设置, 对比两个系统的平均吞吐. 由于集群规模限制, BiStream 无法测试 1024 核. 使用滴滴出行数据集时将滑动时间窗口大小分别设置为 180 或 300 s, 使用网络流量轨迹数据集时分别设置为 15 或 30 s. 图 8 展现了测试结果, 当 FJoin 和 BiStream 单元数量相同时, FJoin 的吞吐提升为至少 3 倍, 且加速器卡的资源使得 FJoin 可以轻松扩展至 1024 单元, 进而获得 5 倍的吞吐提升. 相比较下, 部署大规模的分布式流连接系统在扩展的硬件成本、运维成本方面都没有优势.

5.3 系统时延

本小节实验探究 FJoin 的处理时延. FJoin 的处理时延主要受两个因素影响. 第一是 Task 调度到流水线上执行时, 遍历读取设备内存中所有连接窗口元组的时间. 在本小节的实验中, FPGA 访存接口的频率为 300 MHz, 访存位宽的 512 bit 大于元组最大位宽, 因此遍历访问一百万个连接窗口元组的时间仅为 3 ms 左右. 第二是调度 Task 的时间间隔. 由于每个 Task 调度到托管线程执行时, 都要提前将主机内存中的连接窗口增量同步至 FPGA 设备内存, 数据迁移的时间开销限制了 Task 的调度频率. 我们给出一个可以设置的系统参数, 称为预期时延, 其包括了 Task 生成到执行完毕的总时间, 流

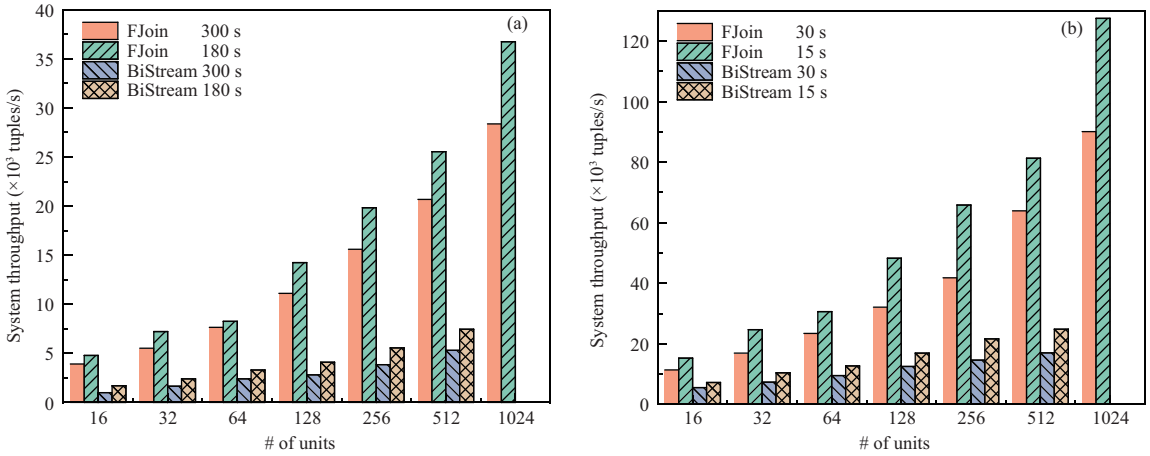

图 8 (网络版彩图) (a) 滴滴出行, (b) 网络流量轨迹连接不同并行度下的平均吞吐

Figure 8 (Color online) Average throughput with different parallelism of (a) DiDi join and (b)network trace join

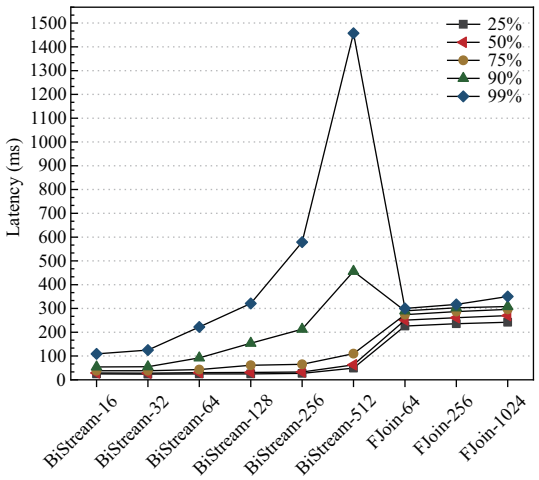

图 9 (网络版彩图) 连接处理时延分位点

Figure 9 (Color online) Join processing latency quantile

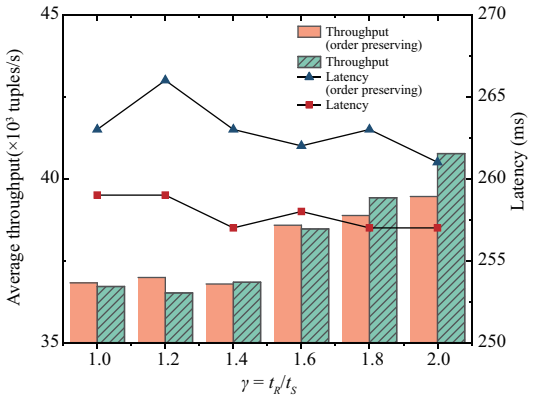

图 10 (网络版彩图) 不同流速比值下, 保序输出的吞吐与时延

Figure 10 (Color online) System throughput and latency of different R/S stream rate ratios with order preserving output

连接任务调度器会根据预期时延决定合适的 Task 调度间隔. 在本小节的实验中, 预期时延最低被设置为 200 ms, 这是根据实际测试取得的. 若预期时延低于该值, 数据在主机和设备间的迁移无法被连接计算隐藏, 系统的吞吐和时延性能都会出现恶化.

图 9 展现了不同规模的 FJoin 与 BiStream 系统的处理时延, 各条曲线分别表示时延不同的分位点. 实验使用滴滴数据集与相应连接谓词, 窗口设定为 180 s, FJoin 预期时延设定为 200 ms, 本小节实验均为类似设置. BiStream 系统具有一定冗余资源, 以获得最好的时延表现. 结果表明 64, 256 和 1024 个单元的 FJoin 超过 99% 的连接处理时延在 200~400 ms, 集中分布在预期时延以上的小范围中. 由于大规模的 FJoin 调度 Task 的时间间隔更长, 处理时延略为增加. 小规模 BiStream 的处理时延优

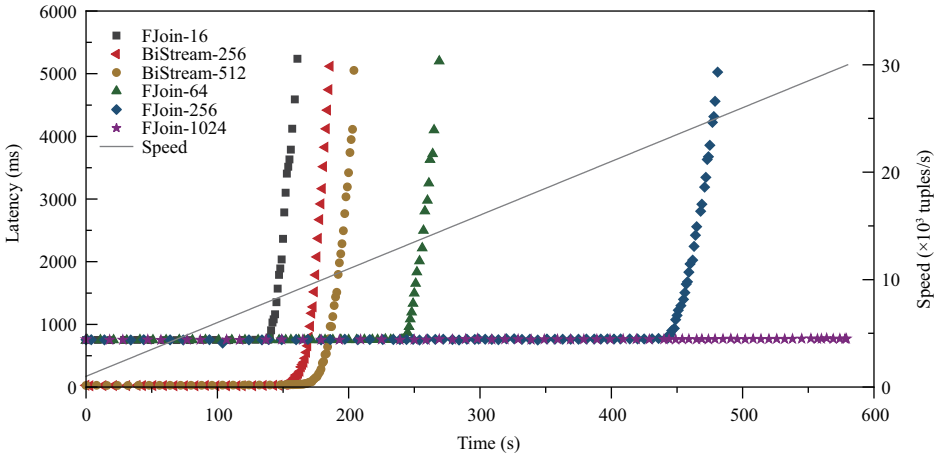

图 11　(网络版彩图) 持续增加流速的时延变化

Figure 11　(Color online) Latency change in continuously increasing of stream speed

表 3　连接谓词的代码行数

Table 3　Lines of code for join predicate

Join predicate	FJoin LoC	BiStream LoC
Manhattan distance	C++ 21 and RTL 121	JAVA 28
IP address similarity	C++ 22 and RTL 130	JAVA 25

于 FJoin, 例如 128 核以下 BiStream 的 99% 时延分位点低于 FJoin, 256 核 BiStream 的 90% 时延分位点低于 FJoin. 这是因为 CPU 可以在每个元组到达时立即处理, 无需等待批量调度. 而在 BiStream 的规模扩展至最大 512 核时, 其 90% 时延分位点高于 FJoin 的 99% 分位点, 这是因为分布式系统同步的开销随规模增大快速上升, 而基于 FPGA 的 FJoin 不存在此类开销.

图 10 展现了两条流流速比值 γ 不同取值时, 打开或关闭保序输出对 FJoin 吞吐与时延的影响. 结果显示开启结果后处理的保序输出功能不会明显降低系统吞吐, 这是因为保序输出的排序操作由主机 CPU 完成, 不占用 FPGA 计算时间. 流速比值 γ 从 1 增加至 2, 系统吞吐逐渐增加, 这与系统吞吐定义为两条流流速之和有关, 大于 1 的 γ 取值使得相同吞吐数据下连接计算量相对减少. 额外的保序输出排序会增加 FJoin 的处理时延, 但结果显示时延增加 10 ms 左右, 没有显著影响.

图 11 展现了流速超过系统处理能力时, 处理时延的快速上升. 该实验中 FJoin 预期时延设定为 600 ms. 我们首先将流速固定为 1000 元组每秒, 持续运行超过两个窗口时间后, 每秒将流速固定增加 50 元组每秒, 使用右轴刻度的 Speed 曲线标注了此刻开始的每秒流速. 具有 16 个连接单元的 FJoin 与具有 256 个 CPU 核 BiStream 系统同时在开始加速后约 150 s 出现时延快速上升, 之后其他规模的系统也相继出现时延快速上升. 这表明连接新元组和窗口的实时计算开销已经超出系统处理能力, 导致流数据的不断堆积, 处理时延在短时间内快速、无限地上升. 而在加速 600 s 范围内, 具有 1024 个单元的 FJoin 则一直保持低时延, 满足实时处理的要求.

5.4　资源开销

表 3 显示了评估使用的两个连接谓词在 FJoin 和 BiStream 中实现所需的代码行数. BiStream 基于 Storm [9] 搭建, 提供流连接通用拓扑, 定义连接谓词只需修改元组定义和连接 Bolt 的连接谓词计

表 4 连接流水线的资源开销

Table 4 Resource occupancy information of join pipeline

Resource type	#of basic units	Used	Usage ratio (%)
LUTs	512	209k	62.8
	256	105k	31.5
Registers	512	450k	66.7
	256	225k	33.3
BRAMs	512	37	7.9
	256	23	4.9
Clock frequency (MHz)	512	300	100
	256	300	100

算方法两处 Java 代码. 在 FJoin 中, 连接谓词的定义包括主机代码和 FPGA 加速器代码两部分. 主机代码使用 C++ 语言描述, 定义连接谓词时只需修改元组定义, 因此代码行数少于 BiStream. 为使用 FPGA 加速器, 需要额外使用 RTL 实现基本连接单元中的连接谓词逻辑. 即使使用基本连接单元的通用框架, RTL 实现连接谓词逻辑所需代码行数仍多于 Java 实现的 BiStream 系统.

表 4 显示了一条连接流水线中, 实例化不同数量的基本连接单元的资源开销. 基本连接单元实现相同的滴滴出行数据集对应连接谓词, 整体资源使用量与单元数量线性相关. 除必须的访存和控制逻辑外, 其他 FPGA 资源用于实例化连接单元. 除接入同一时钟外, 连接流水线没有全局控制信号, 只需对连接谓词逻辑进行合适的周期分割, 运行频率可以达到 FPGA 的最大时钟频率.

6 讨论

本文提出的基于 FPGA 并行加速流连接的方法主要解决通用连接谓词的并行计算, 但对于一些连接谓词, 可以通过设计不同的连接窗口元组索引以加速连接过程. 如何设计高效的连接索引是一个复杂的问题, 一些研究进行了深入探讨 [20~22]. 在 FPGA 平台上部署复杂的索引逻辑十分困难, 能否在 FPGA 上采用针对特定谓词的索引结构是一个有待研究的问题. 另外, FPGA 容易实现简单的连接谓词, 一些复杂的谓词可能不适合大规模并行化的加速方法. 本文的评估过程中使用了真实数据与构造的查询, 产生的连接结果数量并未严重阻塞连接流水线. 高连接度的流连接属于计算和数据双重密集的应用, FPGA 中流水线的设计并不能突破大量结果元组导致的访存瓶颈, 如何处理高连接度的大规模流连接是另一个值得研究的问题.

7 结论

本文提出一种基于 FPGA 的流连接并行加速器 FJoin, 通过高并行的流动连接设计, 实现了流连接计算的高效大规模扩展, 有效提升流连接系统吞吐, 且时延满足流处理要求. 利用 FPGA 大规模向上扩展突破了并行流连接系统的扩展性瓶颈, 同时避免了分布式流连接系统向外扩展的硬件性能浪费. FJoin 通过基本连接单元串联深度连接流水线的方法, 解决 FPGA 实例化的大量连接计算核不具备独立访存通道的问题, 并有效减少访存. 利用软硬件各自优势, FJoin 在大规模并行计算的同时, 高效实现了连接完整性的计算控制. 我们基于现有 FPGA 加速应用开发框架实现了 FJoin. 评估实验

结果表明, 部署在单个 FPGA 加速器卡上的 FJoin 吞吐性能成倍优于集群部署的分布式流连接系统 BiStream, 且处理时延同样满足实时连接的要求.

参考文献

1 IDC & Seagate White Paper. The Digitization of the World: From Edge to Core. 2018. https://www.seagate.com/cn/zh/our-story/rethink-data

2 Zhou S J, Zhang F, Chen H H, et al. Fastjoin: a skewness-aware distributed stream join system. In: Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium, Rio de Janeiro, 2019. 1042–1052

3 Teubner J, Mueller R. How soccer players would do stream joins. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Athens, 2011. 625–636

4 Najafi M, Sadoghi M, Jacobsen H A. SplitJoin: a scalable, low-latency stream join architecture with adjustable ordering precision. In: Proceedings of USENIX Annual Technical Conference (USENIX ATC), Denver, 2016. 493–505

5 Yuan J, Wang Y, Chen H, et al. Eunomia: efficiently eliminating abnormal results in distributed stream join systems. In: Proceedings of the 35th IEEE/ACM International Symposium on Quality of Service, Tokyo, 2021

6 Elseidy M, Elguindy A, Vitorovic A, et al. Scalable and adaptive online joins. Proc VLDB Endow, 2014, 7: 441–452

7 Lin Q, Ooi B C, Wang Z, et al. Scalable distributed stream join processing. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Melbourne, 2015. 811–825

8 Zhang F, Chen H H, Jin H. Simois: a scalable distributed stream join system with skewed workloads. In: Proceedings of the 39th International Conference on Distributed Computing Systems, Dallas, 2019. 176–185

9 Toshniwal A, Taneja S, Shukla A, et al. Storm@twitter. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Snowbird, 2014. 147–156

10 Zeuch S, Monte B D, Karimov J, et al. Analyzing efficient stream processing on modern hardware. Proc VLDB Endow, 2019, 12: 516–530

11 Kang J, Naughton J F, Viglas S D. Evaluating window joins over unbounded streams. In: Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, 2003. 341–352

12 Gedik B, Bordawekar R R, Yu P S. CellJoin: a parallel stream join operator for the cell processor. VLDB J, 2009, 18: 501–519

13 Roy P, Teubner J, Gemulla R. Low-latency handshake join. Proc VLDB Endow, 2014, 7: 709–720

14 Fang J H, Zhang R, Wang X T, et al. Distributed stream join under workload variance. World Wide Web, 2017, 20: 1089–1110

15 Gulisano V, Nikolakopoulos Y, Papatriantafilou M, et al. Scalejoin: a deterministic, disjoint-parallel and skew-resilient stream join. In: Proceedings of IEEE International Conference on Big Data, Santa Clara, 2015. 144–153

16 Zhang S H, Zhang F, Wu Y J, et al. Hardware-conscious stream processing: a survey. SIGMOD Rec, 2020, 48: 18–29

17 Najafi M, Sadoghi M, Jacobsen H A. Scalable multiway stream joins in hardware. IEEE Trans Knowl Data Eng, 2020, 32: 2438–2452

18 Thomas J, Hanrahan P, Zaharia M. Fleet: a framework for massively parallel streaming on FPGAs. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, 2020. 639–651

19 Wu S, Hu D, Ibrahim S, et al. When FPGA-accelerator meets stream data processing in the edge. In: Proceedings of the 39th International Conference on Distributed Computing Systems, Dallas, 2019. 1818–1829

20 Shahvarani A, Jacobsen H A. Parallel index-based stream join on a multicore CPU. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Portland, 2020. 2523–2537

21 Li C L, Li T, Han Y H, et al. Research on FPGA acceleration technology of DNS authoritative server. Sci Sin Inform, 2020, 50: 576–587 [李成龙, 李韬, 韩玉浩, 等. DNS 权威服务器 FPGA 加速技术研究. 中国科学: 信息科学, 2020, 50: 576–587]

22 Chen X, Chen Y, Bajaj R, et al. Is FPGA useful for Hash joins. In: Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR), Amsterdam, 2020

FJoin: an FPGA-based parallel accelerator for stream join

Litao LIN, Hanhua CHEN* & Hai JIN

National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074*, China*
* Corresponding author. E-mail: chen@hust.edu.cn

Abstract Stream join is widely used to extract key information between multi-source stream data and is an important supporting technology for big data processing. Join is easy to become a performance bottleneck because of the large-scale join predicate calculation when joining two big data streams. To improve performance, stream join systems often adopt parallel or distributed expansion methods. However, the multi-core parallel stream join system cannot cope with large-scale data streams because scalability is limited by the number of CPU cores. And the distributed extended stream join system introduces the overhead of distributed framework, resulting in a serious drop in hardware processing efficiency. To achieve efficient and large-scale expansion, this paper proposes a stream join system FJoin that uses the FPGA accelerator to scale up. FJoin can do High-Parallel Flow Join, in which data of the join window can flow through once to complete all join calculations after loading multiple stream tuples. For join predicates whose logic is easy to implement in FPGA, a large number of basic join units are connected in series to form a deep join pipeline to achieve large-scale parallelism. The host CPU and FPGA device coordinate control, divide the continuous stream join calculation into independent small-batch tasks and efficiently ensure completeness of parallel stream join. FJoin is implemented on a platform equipped with an FPGA accelerator card. The test results based on large-scale real data sets show that FJoin can increase the join calculation speed by 16 times using a single FPGA accelerator card and reach 5 times system throughput compared with the current best stream join system deployed on a 40-node cluster, and latency meets the real-time stream processing requirements.

Keywords stream join, FPGA, stream process, hardware accelerate, parallel computing