arXiv:2602.08190v1 [cs.DB] 9 Feb 2026

ZipFlow：释放现代 GPU 压缩数据移动潜能的基于编译器的框架

Gwangoo Yeo
KAIST 韩国大田

Zhiyang Shen
清华大学
中国北京

Wei Cui
微软亚洲研究院
温哥华，加拿大

Matteo Interlandi 微软灰色系统实验室 洛杉矶，美国

Rathijit Sen
微软灰色系统实验室
雷德蒙德，美国

Bailu Ding
微软研究院
雷德蒙德，美国

Qi Chen 微软亚洲研究院 温哥华，加拿大

Minsoo Rhu
KAIST
韩国大田

摘要

在 GPU 加速数据分析中，由于 PCIe 带宽有限，当数据规模超过 GPU 内存容量时，从 CPU 向 GPU 传输数据的开销便会成为性能瓶颈。数据压缩有助于减少数据传输量，同时利用强大的 GPU 计算能力进行解压缩。然而，要优化端到端查询性能，就必须根据压缩策略和硬件特性，对数据压缩、传输和解压缩的工作流进行整体设计，以平衡 I/O 延迟与计算开销。本文提出 ZipFlow，这是一种基于编译器的框架，用于优化 GPU 加速数据分析中的压缩数据传输。ZipFlow 根据压缩算法固有的并行性，将其划分为三种不同模式。针对每种模式，ZipFlow 采用通用调度策略，以有效发挥不同架构 GPU 的计算能力。基于这些模式，ZipFlow 实现了灵活、高性能的整体优化，显著提升了端到端数据传输能力。我们在行业标准基准测试 TPC-H 上评估了 ZipFlow 的有效性。总体而言，与最先进的 GPU 压缩库（nvCOMP）相比，ZipFlow 平均提升 2.08×；与基于 CPU 的查询处理引擎（如 DuckDB）相比，速度提升 3.14× 倍。

1 引言

随着人工智能的快速发展，例如深度神经网络和大语言模型，GPU 已得到广泛应用。有趣的是，现代 GPU 具备强大的计算能力和高内存带宽，因此在分析型数据库工作负载方面也展现出巨大潜力。近期研究已证明，利用 GPU 加速分析型数据库系统前景广阔，学术界和业界均开展了相关探索

以及行业 [2, 30, 33–35, 37, 38, 43, 56]。

尽管取得了如此令人鼓舞的结果，但受限的 GPU 内存容量一直是 GPU加速数据库系统面临的关键限制。例如，高端 GPU（如 NVIDIA H100[6]）的内存容量最高也只有 80 GB，而分析型数据库通常会扩展到数百 GB，甚至数 TB。尽管使用

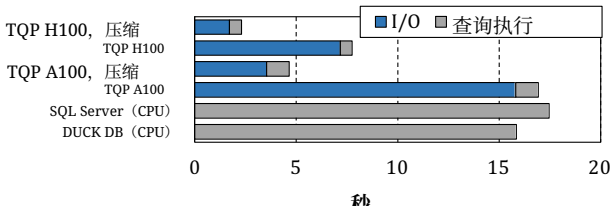


图 1：CPU（AMD EPYC 7V12）[8, 12], A100（PCIe 4）和 H100（PCIe 5）上 22 个 TPC-H 查询（SF=100）的延迟分解；TQP [33] 用于在 GPU 上进行查询处理。

通过高性能互连（即 NVLink [14]）连接的多 GPU 可以在高内存带宽下提升内存容量，但这类硬件的成本可能高得令人难以承受。由于数据库无法完全缓存在 GPU 内存中，查询执行期间通常需要按需通过 PCIe总线将数据从 CPU 传输到 GPU [23, 24, 29, 57, 65]。遗憾的是，由于 PCIe总线速度较慢，数据传输往往会成为性能瓶颈。例如，图 1 展示了基于 CPU 和基于 GPU 的数据库系统执行 22 个 TPC-H 查询时的查询延迟分解。尽管在 A100 [1] 上的查询执行速度比在 CPU 上快 15× 倍以上，但由于数据从 CPU 到 GPU 的传输速度较慢，端到端查询延迟的加速比要温和得多。

幸运的是，分析型数据库系统通常以压缩格式存储数据（例如 Parquet [4]）。这使得数据可以先在 CPU 上完成压缩，随后在查询处理之前由 GPU 解压缩。这一设计选择减少了数据传输量，同时充分利用了 GPU 的计算能力。尽管数据压缩已成为分析型数据库系统的常态，但我们观察到，现有的数据压缩技术是为基于 CPU 的数据库系统设计的，并未充分挖掘 GPU加速数据库系统独特的性能特征。

首先，GPU拥有强大的计算能力，可充分发挥更激进压缩策略的优势，以提高压缩率，进而改善端到端查询性能。现有的数据压缩技术往往会限制可选压缩算法及其嵌套方式，以在

压缩比和解压开销之间取得平衡。例如，广泛使用的Parquet格式只考虑计算效率高的压缩算法，如游程编码（RLE）、增量编码、字典编码和位打包；而开销更大的压缩算法，如基于 LZ77 和基于熵的算法，在某些工作负载上可实现高得多的压缩比。即使是最先进的基于GPU的压缩库，对嵌套压缩方案的支持也很有限。例如，nvCOMP [13] 仅支持特定组合的游程编码、增量编码和位打包嵌套。

其次，要实现最佳性能，压缩方案必须针对目标 GPU硬件进行定制。根据数据特征进行调整的专用算法虽然可能非常有效 [19, 22], 但将它们分别映射到各种 GPU 架构上，需要付出大量开销，以便针对每个目标平台调整资源利用率。现有的 GPU 压缩框架 [13, 18, 57] 仅为传统算法提供预构建内核库，而手动扩展这些框架以纳入新的或自定义的压缩技术，既繁琐又低效。此外，这些框架往往不支持优化嵌套压缩方案，即无法利用预构建内核进行跨阶段优化的内核融合。更重要的是，要在异构 GPU 架构之间实现无缝的内核兼容性，例如通过 CUDA [5] 支持 NVIDIA GPU，以及通过 ROCm [10], 支持 AMD GPU，对现有框架而言仍是一项严峻挑战。

最后，要优化端到端查询性能，需要对数据压缩、传输和解压缩工作流进行整体设计。这包括探索压缩比、解压速度和数据传输调度之间的权衡，以最大限度地降低总体延迟。尽管以往工作研究了各个组件的性能 [18, 41, 46, 57]，或考察了仅涵盖少量算法的设计 [44, 48],，但这种局部决策可能导致端到端性能不佳。

为弥补这些不足，我们提出了 ZipFlow，这是一种面向现代 GPU 的灵活编译器压缩框架。ZipFlow 将通用压缩算法抽象为三种并行计算模式。这些模式充当构建模块，使现有算法和新的自定义算法都能灵活地进行通用设备调度。借助 ZipFlow 配套的编译器技术，每种模式都能轻松优化，从而在各种 GPU 架构上充分发挥其并行性。通过整体探索压缩数据传输工作流中 I/O 与计算开销之间的权衡，ZipFlow 实现了优化的端到端查询性能。我们在业界标准 TPC-H 基准测试上的评估展示了 ZipFlow 的优势。与最先进的 GPU 压缩库 [13] 相比，我们的自定义压缩方法具有更高的压缩率，并节省了 1.85× I/O 延迟。同时，借助自动编译器优化，ZipFlow 还能将解压速度提升 3.26×。与基线相比，这使端到端查询延迟平均加速 2.08×。

综上所述，本文做出以下贡献：

- 我们抽象出三种模式，用于封装（解）压缩算法中常见的核心并行性结构（第 3 节）。这些模式支持对现有和自定义的 GPU（解）压缩算法进行原生优化，同时也有助于未来开发更广泛的 GPU 友好型（解）压缩算法。
- 针对每种模式，我们构建了一个适配 GPU 执行几何结构与单指令多线程范式的通用优化空间（第 4 节）。结合融合与流水线，该方法从整体上最小化端到端压缩数据传输开销。
- 我们在多种数据分布上，针对相同算法对 ZipFlow 与最先进的 nvCOMP 进行了直接对比（第 5.2 节），结果表明我们的硬件感知方法有效优于最先进的 GPU 压缩库（即 nvCOMP）。
- 我们在业界标准 TPC-H 基准测试上对 ZipFlow 进行了全面评估（第 5.4 节），展示了压缩数据传输工作流联合优化对端到端查询性能的影响。最后，我们通过在异构GPU上评估其性能，展示 ZipFlow 如何实现定制化的硬件感知执行。

2 背景

2.1 现有压缩算法

本节将概述一些广泛使用的无损压缩算法，并探讨其背后的基本设计原理。我们还在图 2 中为每种算法绘制了示例示意图。

字典编码通过字典和索引来替代数据。字典存储数据中的唯一值，而索引则将原始数据值映射到字典中对应的条目。**增量编码**将每个值替换为当前值与前一个值之间的差值（增量），同时将初始值存储为基数。增量编码本身无法实现压缩，但可配合游程编码或位打包进一步压缩。

游程编码（RLE）通过将连续重复的整数值替换为一个值和一个计数，减少冗余。值存储重复的值，而计数记录重复次数。

位打包和参考帧编码（FOR）将整数打包为给定数据所需的最小位宽。位宽根据最小值和最大值计算，而最小值则作为 FOR 单独存储。

浮点数转整数 从浮点数中分离有效数字和指数，并将其转换为整数，以便进一步压缩。这种方法适用于压缩小数精度有限的列。若干先前研究采用了相同的核心思想 [19, 29, 36, 41], , 但名称和具体细节有所不同。

字符串字典 通过将重复的字节序列替换为字典索引来消除重复，并构建字典以存储目标数据集中频繁出现的字节序列（参见先前研究 [22, 60]）。压缩比取决于生成的字典消除模式重复的效果。

基于熵的算法，例如 ANS [9, 28] 和霍夫曼编码 [39]，通过利用数据出现频率的偏斜来压缩数据。对于频繁出现的数据，分配较少的位，从而减小整体数据大小。

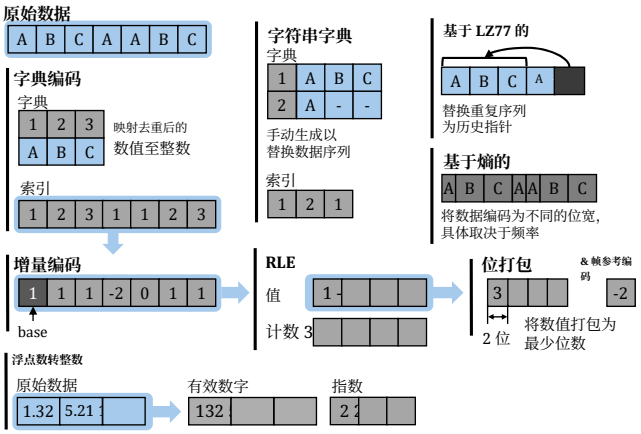


图 2：不同类别的压缩算法。

基于 LZ77 的算法，例如 LZ4 [11] 和 Snappy [15]，通过将重复序列编码为指向先前数据的指针来消除重复。这些算法因兼容性广泛而被称为“通用”算法。Deflate [27] 和 zstd [16] 等最先进的算法将 LZ77 与基于熵的方法相结合，以实现更高的压缩比。

2.2 数据库系统中的现有方法

在现代数据库系统中，高效的数据压缩通常通过组合多种轻量级压缩技术来实现，例如游程编码（RLE）、增量编码、字典编码和位打包。这种范式通常被称为级联压缩，通过按顺序编排各个算法，发挥它们各自的优势，并根据数据的特征量身定制。针对单个数据文件设计良好的嵌套算法可以实现有效的压缩比，同时保持计算模式简单 [25]。BtrBlocks [41]引入了一种自动搜索机制，用于确定优化的嵌套算法，从而显著提高压缩率。

ORC [50] 和 Parquet [4] 等知名列式存储格式采用了这种级联方法，实现了多种精心设计的嵌套压缩方案，例如将游程编码与字典编码结合，或将增量编码与位打包集成。它们还支持在预定义的嵌套算法之上应用通用算法，以进一步压缩。Fastlanes 项目 [17] 旨在增强压缩列式数据格式，通过增加并行解码的机会，提供了多种手工编写的 GPU 内核，例如位打包、增量编码 [18],和浮点数转整数（Float2Int） [36],，以充分利用 CPU 和 GPU 的 SIMD 与 SIMT 处理能力。

3 设计

我们提出了 ZipFlow，这是一种用于加速基于压缩的数据向 GPU 移动的框架。图 3 展示了经 ZipFlow 优化的端到端数据移动流水线的高层概览。不同格式的原始数据（例如 NumPy、Parquet、文本、二进制）首先利用 ZipFlow 灵活的嵌套机制和可自定义算法进行压缩，所得的压缩表示

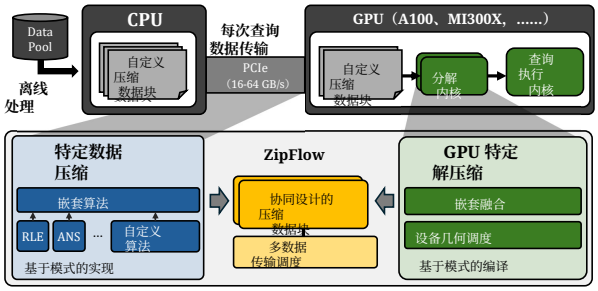


图 3：从存储的数据到 ZipFlow 的端到端执行流水线。

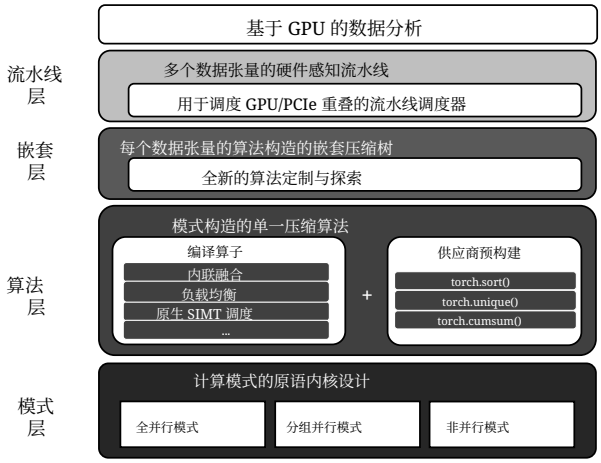


图 4：ZipFlow 架构概览。

存储在 CPU 内存中。对于在线数据传输，ZipFlow 促进压缩数据通过 PCIe 互连进行高效移动，随后执行超高速、针对设备的解压缩阶段。该端到端流程由 ZipFlow 进行协同设计，以充分利用现代 GPU 的异构性，不仅整合了硬件原生的资源管理，还优化了 PCIe 数据传输与设备端解压缩之间的重叠，从而最大化吞吐量并最小化延迟。

图 4 介绍了 ZipFlow 的四层设计。从底层开始，**模式层**基于三种预定义的计算模式，实现了基础 GPU 算子的设计。每个设计的算子都对应一个 GPU 内核，并作为（解）压缩算法的构建模块。这些 GPU 算子既能简化目标算法的实现，又能利用（解）压缩算法中普遍存在的并行模式。在**算法层**中，GPU 模式被组织起来，形成 RLE、字典编码、位打包或 ANS 等基础压缩算法。基于算法层中的基础算法，**嵌套层**探索并自定义应用于数据列的算法嵌套方式（例如，字典编码 + ANS 或 RLE + 位打包），以实现端到端数据移动延迟的最优表现。最上层是**流水线**

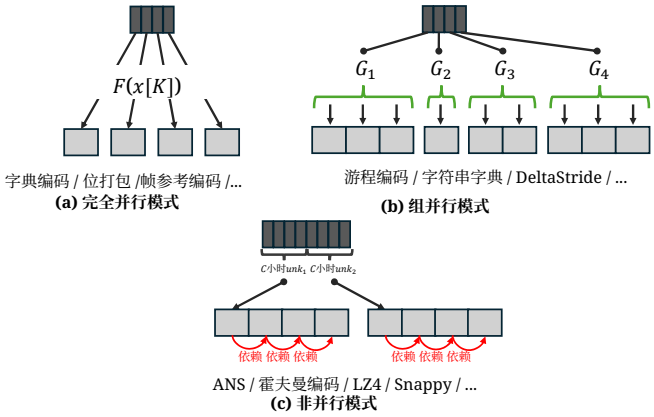


图 5：ZipFlow 并行模式的逻辑依赖关系。

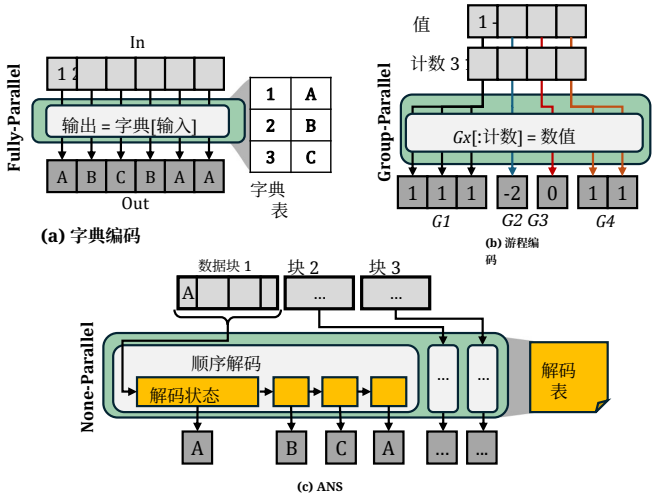


图 6：各模式的数据映射示例。

层，我们的框架在该层调度 I/O 操作的流水线，并在多个数据块之间调度 GPU 解压缩延迟，以最小化总体延迟。

3.1 模式层

根据我们对常见（解）压缩算法的观察，我们确定了三种主要的并行模式。为全面涵盖各种可能的并行性选项，我们将可并行化模式分为三种不同模式：完全并行、组并行和非并行。对于给定的输入数组和输出数组，这些模式分别涵盖（解）压缩算法中的**N 对 1**、**1 对 N**和**N 对 M** 个独立计算块。

完全并行模式 适用于这样的（解）压缩操作：每个输入都能独立映射到输出值，且不存在跨元素依赖，如图 5(a) 所示。尽管图中仅展示了简单的 1:1 场景，但也可以有固定数量的标量输入数组，以支持 N 对 1 的计算块。输入值与输出值之间的计算可以由任意映射函数定义，同时也支持输入索引与输出索引之间的任意映射。字典编码、位打包、浮点数转整数（G-ALP [36]）和 Fastlanes [18] 等算法的解压缩技术，都可以通过完全并行内核实现。图 6(a) 展示了一个使用完全并行内核解压字典编码数据的示例。该内核将字典查找表用作并行映射函数，并将字典作为元数据提供。这个完全并行内核会针对 In 选中的每个元素并行查找字典，并将对应的值写入 Out。

分组并行模式 专注于针对空间非均匀分布自定义压缩算子，其中整个任务被划分为多个大小可变的组 G_1，G_2，..，G_n 每个组内包含相互独立的子任务。G_x 大小不同会导致执行不均衡。例如，图 5(b) 展示了四个组，每组包含数量不同的元素，而每组内部的箭头表示 1 对 N 的并行关系。通常，此模式适用于所有基于游程编码的编码系列，包括增量编码、字符串字典和

GSST [60]。我们展示了如何将 RLE 解码映射到

图 6(b) 中的组并行模式。压缩的 RLE 数据由计数数组和值数组组成，其中计数数组指定每个组中各元素的出现次数，而值数组中的每个元素对应一个输入符号。在此用例中，写入输出时使用直接复制函数作为映射函数，因此组并行内核会并行复制值数组中的每个元素，在每个组内重复该元素计数次，并将结果写入相应的组索引。

非并行模式 适用于所有其他无法采用完全并行和组并行方式进行并行化的压缩方法，例如 Snappy 和霍夫曼编码等本质上串行的算法。我们利用压缩算法的文件分块功能，将目标文件划分为更小的数据块，并分别对其进行压缩和解压，以便并行处理。正如图 5(c) 所示，这种方法使不同处理器能够并发处理彼此独立的文件数据块，从而让单个 GPU 处理器能够像处理为 CPU 设计的代码一样处理串行代码。与许多其他串行算法类似，我们以 ANS解压为例。ANS 中的每个中间解码状态都依赖其前一个状态来构造相应元素，并递归地将自身状态传递给后一个状态。如图 6(c) 所示，在每个数据块内，这一工作流程要求解码状态严格按顺序逐步推进。不过，将来自不同数据块的中间解码状态分组，并以单指令多线程（SIMT）方式分发，可实现并行处理，详情将在第 4 节进一步说明。

3.2 算法层与嵌套层

算法层整合了由模式层编译而成的内核，用于构建基础的（解）压缩算法。ZipFlow 中的基础（解）压缩算法池包括增量编码、字典编码、字符串字典、ANS、位打包和浮点数转整数；如第 5 节所示，必要时还可以添加其他算法。为进一步简化实现，该框架还支持直接调用

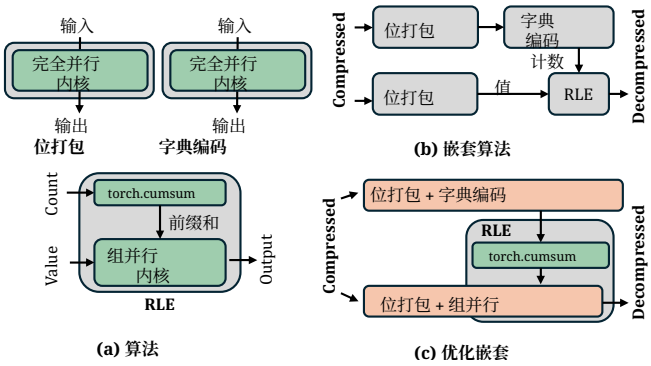


图7：PyTorch 中算法与嵌套层的执行。

PyTorch 开箱即用操作中的预构建算子，例如排序、去重和累积和，作为辅助操作。

图7(a)展示了位打包、字典编码和游程编码的解压函数设计。位打包和字典编码的解压只需一个完全并行内核，而游程编码则需要通过两步内核进行解压。首先，使用 PyTorch 的累积和对计数变量进行预处理，以计算基础输出索引，并将其存储在名为 presum 的变量中。接下来，应用组并行内核，对压缩数据进行并行扩展。在嵌套层中，用户可以从算法层的基础算法池中组合算法，实现嵌套压缩算法。这些嵌套算法被视为目标数据列最终采用的压缩或解压缩方式。图7(b)展示了一个嵌套解压缩示例，其中使用了根据图7(a)设计的子算法；图7(c)则进一步展示了通过将完全并行模式与相邻模式融合来实现的优化。位打包和字典编码中连续的完全并行内核被融合，用于计算计数张量。同样，生成值张量的位打包也与游程编码函数内部的组并行内核融合。重新审视模式层以进行内核融合，可以通过消除冗余内存传输实现进一步优化。

3.3 流水线层

基于 GPU 的查询处理需要将多个数据块从 CPU→GPU 传输过来。为解决这些数据传输以及后续 GPU 端解压缩所带来的延迟，ZipFlow 引入了专用的流水线层。该层旨在通过协调 CPU→GPU 数据传输与 GPU 上的解压缩任务进行流水线执行，在多个数据块之间无缝重叠这些操作，从而最大限度地降低端到端查询延迟。我们的实证研究表明，数据块流水线处理的顺序会对性能提升产生显著影响。如图 8 所示，考虑两个具有代表性的数据块 A 和 B：数据 A 的数据传输开销较高，但解压缩速度很快；

压缩，而数据 B 则呈现相反的模式：传输成本低，但解压缩时间较长。通过系统地比较不同的数据块传输顺序，我们发现，先以数据 B 启动流水线，再处理数据 A（即 B→A），能够降低总体延迟。这一性能提升

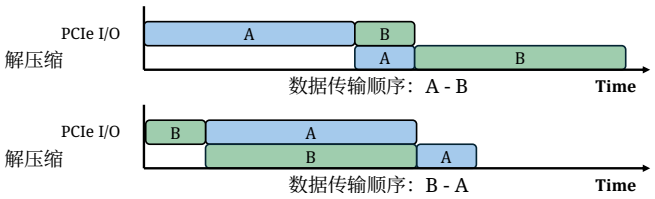


图 8：不同流水线顺序下的传输延迟。

归因于数据移动与解压阶段之间实现了最大程度的并发。

要将这一优化推广到任意查询，关键在于确定数据传输的最佳调度顺序，以使总延迟最小。为此，我们设计了一种利用 Johnson 算法 [40] 的专用流水线调度器。该调度器能够高效地确定最佳数据调度顺序，计算复杂度仅为 $O(n)$。

4 设备几何调度

提高解压内核的效率是 ZipFlow 的另一项关键设计。鉴于不同 GPU 架构在计算单元、内存事务大小、波前大小和缓存容量等片上资源方面存在差异，编译对于将计算任务精细映射到这些异构原生资源至关重要。我们引入一个三维配置向量，记作 $<L、S、C>$。不同的元组指定不同的内核实例，它们执行相同的计算，但利用原生硬件资源的粒度各异。这种可配置格式通过支持根据特定 GPU 的架构特征精确调整向量值，实现内核调度的标准化，从而最大限度地提高片上和片外资源利用率。

通过融合调度完全并行模式。与其他计算模式相比，完全并行模式不受跨元素依赖的限制，因此能够灵活地进行内核内联与融合。这一特性有助于将完全并行模式与各种其他操作集成，例如将类型转换或字典查找无缝融合到 RLE 解码中，从而提高整体计算效率并减少内核启动开销。在无法进行此类融合的场景中，解压缩任务必须使用单独的 GPU 内核。为展示该模式的内核调度策略，图 9 给出了一个具有代表性的配置。在此示例中，变量 C 表示单个 GPU 线程在一次内存事务中共同处理的连续数据元素数量，这通常由所处理数据类型的大小决定。参数 S 指定每个线程块分配的 GPU 线程数，从而决定块内并行度。L 表示主处理循环的迭代次数，因此每个线程会依次处理连续的数据区域；每个区域之间相隔 $S \times C$ 个元素，共处理 L 次。总体而言，乘积 $L \times S \times C$ 有效定义了“瓦片大小”，也就是分配给每个 GPU 块的总工作负载。这种灵活的配置不仅契合硬件执行模型，还能在不同 GPU 架构上实现高效扩展与适配。

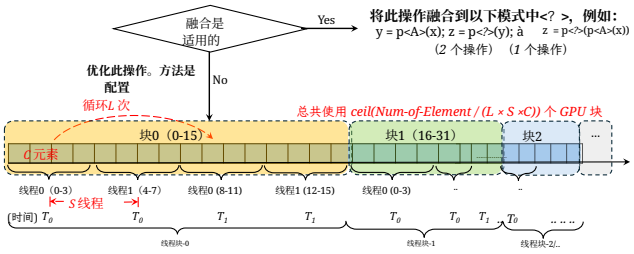


图 9：采用 <*L*、*S*、*C*>=<2、2、4> 的完全并行调度。

通过组并行实现负载均衡。在 GPU 上高效处理非均匀数据颇具挑战。以该模式中的典型示例 RLE 解压缩为例，一种直接的策略是将不同的 RLE 条目均匀分配给所有计算单元并行执行。然而，这种方法往往会导致严重的负载不均衡，因为每个子任务（即每个 RLE 条目中的元素数量）的工作量可能相差悬殊。因此，部分处理器会被分配到负担过重的子任务，而其他处理器则会提前完成并处于空闲状态，导致可用资源利用不足。

ZipFlow 通过结合多个 *GPU* 块在单个组内协同处理 与单个 *GPU* 块跨多个组进行处理来应对这种分歧。如图 10 所示，计算针对从左到右排列的四组数据组织，每组由一根竖条表示，其高度表示该组内可并行处理的独立项。在此配置中，整个数据平面通过调度 8 个 GPU 块（记为 L）进行处理，每个 GPU 块包含 4 个线程（记为 S）。对于每个组，共 16 个线程（记为 C），由 4 个 GPU 块均分贡献（即 C/S），协同处理沿垂直轴的所有子任务。沿水平方向，每个 GPU 块迭代前进以处理后续组，每次跳过步长为 2（即 $L \times S/C$），直至所有组处理完毕。值得注意的是，S可以超过 C，这意味着单个 GPU 块中的多余线程可以以“锁步”方式并发处理跨越相邻 S/C 组的任务，而不是严格地从一个组到下一个组进行顺序执行。该调度使不均衡数据能够沿任意垂直或水平方向扩展，而这仅需一次数据扫描即可实现。

面向单指令多线程的非并行调度。 鉴于该模式下可实现的最大并行性本质上受数据块总数的限制，ZipFlow 会启动 L 个 GPU 块，每个块包含$S \times C$个线程，以处理最多 $L \times S \times C$个数据块；每个数据块由一个线程顺序执行。 L根据数据块数量计算，而

S 固定为硬件线程束大小（例如，所有 NVIDIA GPU 均为 32，而大多数 AMD GPU 为 64 ）。参数 C 取决于架构，使每个 GPU 块能够处理多于 S 个数据块，具体数量取决于每个物理计算单元可用的寄存器文件大小。如图 11 所示，按照物理执行顺序，一组顺序语句会在不同时间步反复执行，且不具备并行性。然而，在每次迭代中，来自独立数据块的项目处理会被归入一个单指令多线程单元，

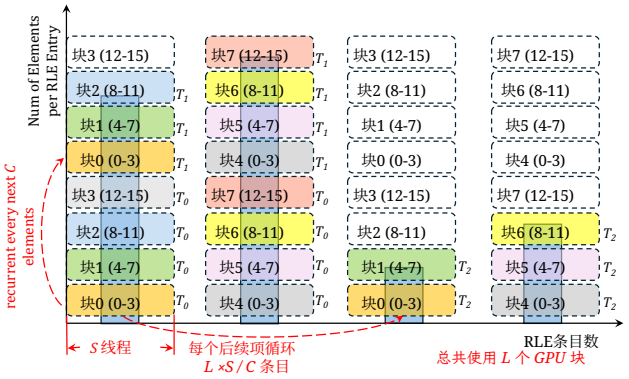


图 10：使用 <*L*、*S*、*C*>=<8,4,16> 进行组并行调度。

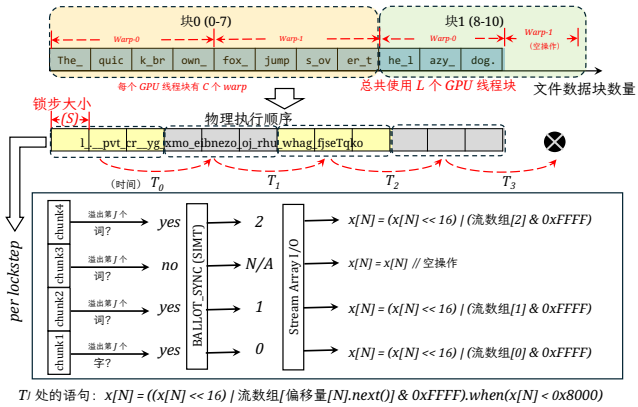


图 11：<*L*、*S*、*C*>=<2、4、2> 的非并行调度。

从而实现锁步执行。这种方法提高了不同数据块之间 I/O 和缓存访问的一致性，进而减少分歧并提升整体内存访问效率。

5 实验

5.1 实验设置

我们使用 TPC-H 基准测试评估 ZipFlow 的压缩比、解压吞吐量和端到端查询执行延迟。默认情况下，所有实验均采用 TPC-H 比例因子 100，并在 NVIDIA A100 GPU [1]上进行，该 GPU 配备 80GB HBM2，通过 PCIe Gen4（理论带宽为 32 GB/s）连接到 AMD EPYC 7V12 64 核处理器。为评估我们的方法在异构 GPU 架构上的设备利用率，我们还在配备 80GB HBM2e 的 NVIDIA H100 GPU [6] 、配备 192GB HBM3 的 AMD MI300X GPU [7] 以及配备 32GB HBM2 的 AMD MI50 GPU [3] 上进行了实验。所有测量均使用固定在 CPU 内存中的数据缓冲区，并将执行限制在单个 CPU 节点上，以尽量减少 NUMA 效应。我们将压缩比定义为压缩后大小与原始大小之比，将解压吞吐量定义为未压缩数据大小除以解压缩时间。

作为 GPU（解）压缩基线，我们采用 nvCOMP v4.0.1.0[13], ，它被广泛认为是最先进的压缩

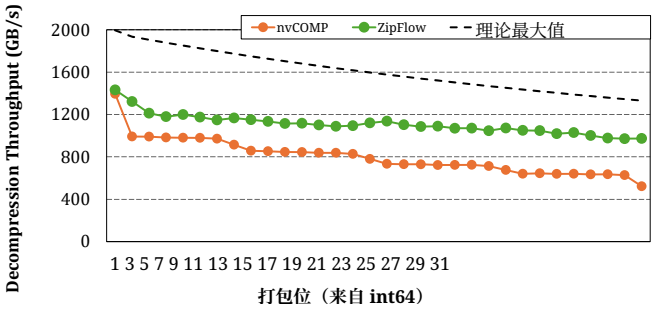


图 12：在不同待解包位宽分布下，位打包（完全并行）的吞吐量。

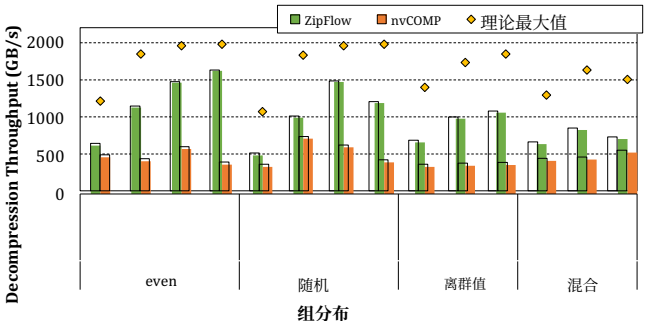


图13：不同组大小分布下 RLE（组并行）的吞吐量：均匀、随机、离群、混合。

库，适用于 NVIDIA GPU。它提供了一套全面的 GPU 优化算法，包括 LZ4、Snappy、Deflate、Zstandard (ZSTD)、非对称数制 (ANS) 和霍夫曼编码等标准方法，以及针对结构化数据（Cascaded）、科学数据（Bitcomp）和 GPU 优化 Deflate（Gdeflate）的专用技术。其中，nvCOMP 中的 Cascaded 框架支持对预定义的游程编码 (RLE)、增量编码和位打包进行可配置的嵌套。

在端到端 TPC-H 查询执行实验中，我们将“ZipFlow + TQP [33]”结合起来，在 PyTorch 环境（Python 3.12 + Pytorch 2.6.0）中构建一个完整运行的 CPU-GPU 混合查询引擎。我们还将其与最先进的基于 CPU 的 DBMS 进行比较，具体来说，是 DuckDB v1.4.1 [8] 和 Microsoft SQL Server 2022 RTM [12], ; 二者均在与 GPU 基线相同的 AMD 处理器上使用 64 个物理线程运行。

5.2 微基准测试

首先，我们在相同的压缩模式下，将 ZipFlow 与 nvCOMP 进行同类比较。对于每种基本模式，我们选取位打包、游程编码和 ANS 作为代表，并将其与 nvCOMP 库中对应的算法在多种数据分布下进行比较。鉴于 nvCOMP 的实现为闭源，我们对 ZipFlow 和 nvCOMP 的算法设置进行校准，使二者达到相同的压缩比，从而能够公平地评估性能指标。

5.2.1 完全并行实例：位打包。图 12 比较了 ZipFlow 和 nvCOMP 中位打包实现的解压性能，虚线表示根据公式 1 计算的理论最大吞吐量：

$$GpuMemBandwidth \times \frac{\text{(plain size)}}{\text{(compressed size)} + \text{(plain size)}} \quad (1)$$

为确保评估公平，我们通过均匀采样 int64 数值，生成压缩比经过校准、位宽粒度各异的 4 GB 合成输入。得益于 ZipFlow 针对 A100 的原生内核调度，随着压缩位宽增加，其性能始终优于 nvCOMP。这些结果表明，即使使用相同的压缩算法，通用库也未必总能提供最佳的每

性能取决于输入数据模式。例如，对于 TPC-H 中的 L_PARTKEY 列（压缩至 25 位宽），我们的方法

在解压缩性能方面实现了 1.63× 的提升。

5.2.2 组并行实例：***RLE***。我们评估了不同组大小分布（即 RLE 中计数 数组的分布）下 RLE 解压缩的吞吐量，考察了四种具有代表性的情形：均匀、随机、离群和混合。在均匀-*X* 分布中，所有组包含相同数量的元素，具体而言，每组大小为 X。例如，均匀 *-2* 会生成如下未压缩序列：A-A-B-B-C-C。随机[L, R] 分布会为每个组分配一个从 L 到 R 范围内均匀随机选取的大小。离群 分布主要由大小为 1 的组构成，其中穿插少量明显更大的组（例如大小为 1024 的组），以模拟高度偏斜的组大小。混合 分布通过拼接上述任意两种情形生成，从而能够评估混合组大小模式下的性能。

在图 13 中，我们全面比较了 ZipFlow 和 nvCOMP 在 14 种具有代表性的数据分布情况下进行 RLE 解压缩的性能。柱形表示测得的吞吐量，点则表示受内存带宽限制时可达到的理论峰值。结果表明，nvCOMP 对数据分布变化的适应能力有限，性能始终不理想，与理论上限相去甚远。这一局限主要源于 nvCOMP 固定的并行化方法：无论组大小如何，都为每个输出元素分配一个线程，导致内存读取争用，并降低可用带宽的利用效率 [49]。相比之下，在所有测试的数据分布下，ZipFlow 的性能都更接近理论最大值。这种高度吻合表明，ZipFlow 能够有效利用数据局部性和 GPU 并行性，从而提高占用率并实现更高的带宽利用率。

5.2.3 非并行实例：***ANS***. 针对非并行模式，我们从两个关键维度进行性能评估：1）改变数值频率比以调节熵，因为熵会显著影响 ANS 的压缩效率（图14）；2）分析不同总数据量下的最佳分块大小（图15），因为该参数会同时影响压缩比和解压吞吐量。我们的评估数据集旨在捕捉

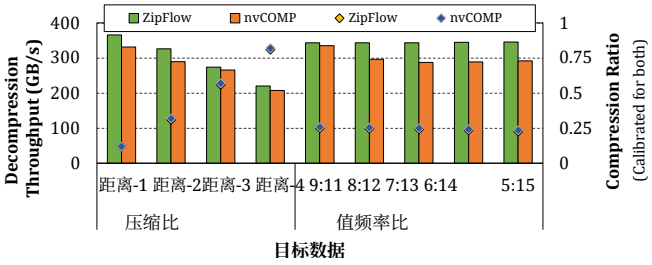


图14：不同压缩比（左）和频率分布（右）下 ANS（非并行）的吞吐量。

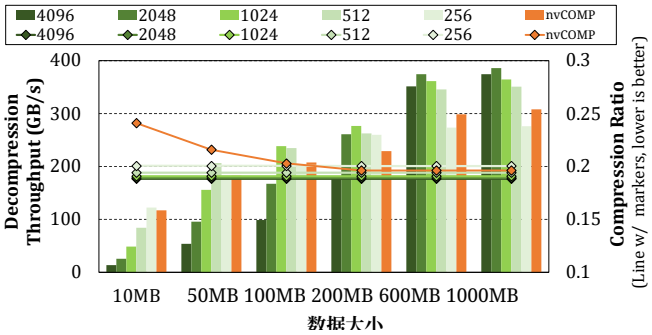


图15：使用不同块大小的 ANS 吞吐量（例如，每个分块 4096 = 4KB；采用不透明设置下的 nvCOMP）。

TPC-H 基准测试中 L_RETURNFLAG 列的分布特性；ANS 压缩非常适合对具有不同频率分布的数值进行编码。

图14从两个维度展示了解压吞吐量：左侧四个柱状图展示压缩比逐渐增大时的解压吞吐量，右侧其余四个柱状图则展示数据频率分布从均衡转为不均衡时的解压吞吐量。当通过提高某些值的频率比使数据分布更加倾斜时，nvCOMP 的吞吐量会明显下降。相比之下，ZipFlow 始终保持稳定的吞吐量，这一点在完全并行和组并行模式下均有所体现；其性能主要受压缩比影响，对频率变化或数据倾斜的敏感度极低。

图15展示了这样一种权衡：采用较大的块大小可获得更高的压缩比，而采用较小的块大小则有助于提高并行性和解压吞吐量。由于 nvCOMP 使用的块大小并未明确记录，我们在 ZipFlow 实现中系统地遍历一系列块大小，以便进行校准后的比较。研究结果表明，对于较小的输入，通过减小块大小来利用更细粒度的并行性，可以提高吞吐量。相反，对于较大的输入，采用较大的数据块可实现更好的可扩展性和更高的吞吐量。对于所有评估过的输入大小，我们的方法都能持续找到最优的块大小配置，在提高解压吞吐量的同时，达到相同或更高的压缩比。

表 1：ZipFlow 与公开方法（nvCOMP [13], Parquet [4], BTRBLOCKS [41]）的比较，其中 F.P.、G.P. 和 N.P 分别表示不同的模式族：完全并行、组并行和非并行。

支持		nvCOMP	Parquet	BTRBLOCKS	ZipFlow
嵌套		预定义	预定义	可扩展	可扩展
GPU支持		✓	✗	✗	✓
F.P.	位打包与FOR	✓	✓	✓	✓
	增量编码	✓	✓	✓	✓
	字典编码	✗	✓	✓	✓
	浮点数转整数	✗	✗	✓	✓
G.P.	RLE	✓	✓	✓	✓
	RLE + 增量编码	✓	✗	✓	✓
	RLE + 字典	✗	✓	✓	✓
	字符串字典	✗	✗	✓	✓
	DeltaStride	✗	✗	✗	✓
N.P.	LZ4	✓	✓	✗	✓
	ANS	✓	✗	✗	✓
	霍夫曼	✓	✗	✗	✓
算法可扩展性		✗	✗	✗	✓

总而言之，对三种模式（完全并行、组并行和非并行）的性能比较表明，ZipFlow 的性能更优。这一提升主要源于其细粒度的 GPU 调度适应性。这种适应性更好地契合了不同数据分布的特征以及现代 GPU 的架构优势。

5.3 TPC-H 数据集上的压缩评估

在本次评估中，我们力求为每个数据列实现尽可能最佳的性能，并利用框架的嵌套机制和灵活的内核池自定义功能，设计经过优化的压缩算法。我们重点关注最大的三张表（L、O 和 PS），它们是 TPC-H 端到端查询延迟的主要瓶颈。我们的自定义压缩算法整合了来自不同模式族的多种嵌套技术，例如位打包、游程编码、增量编码、字典编码、ANS 和浮点数转整数。我们还采用了两种专用压缩变体来处理特定的数据模式。首先，DeltaStride（一种游程编码变体）通过将单调递增整数序列表示为*(起始值、步长、计数)* 三元组来进行压缩。其次，我们为字符串字典压缩器生成了一个自定义字典，详情见第5.3.1节。我们使用的嵌套压缩算法列于表2。

5.3.1 压缩比. 本节将 ZipFlow 达到的压缩率与一个基于 GPU 的基线 nvCOMP [13], 以及三个基于 CPU 的基线进行比较：Parquet [4], Parquet +zstd 和 BTRBLOCKS [41], ，不论它们是否兼容 GPU。表 1 总结了各框架支持的压缩算法，图 16 则展示了它们的压缩率。对于 nvCOMP 和 Parquet，我们考察了它们在表 1 所列全部支持算法中达到的最佳压缩率。下文还将针对具有代表性的列簇对结果进行详细分析。

如图 16 所示，包含日期值的列（即 L_COMMITDATE 和 O_ORDERDATE 之间绘制的四列）可从字典编码中显著受益。相比之下，nvCOMP 不支持这种数据表示，因此其压缩率始终低于所有其他替代方案。对于数值列（即从

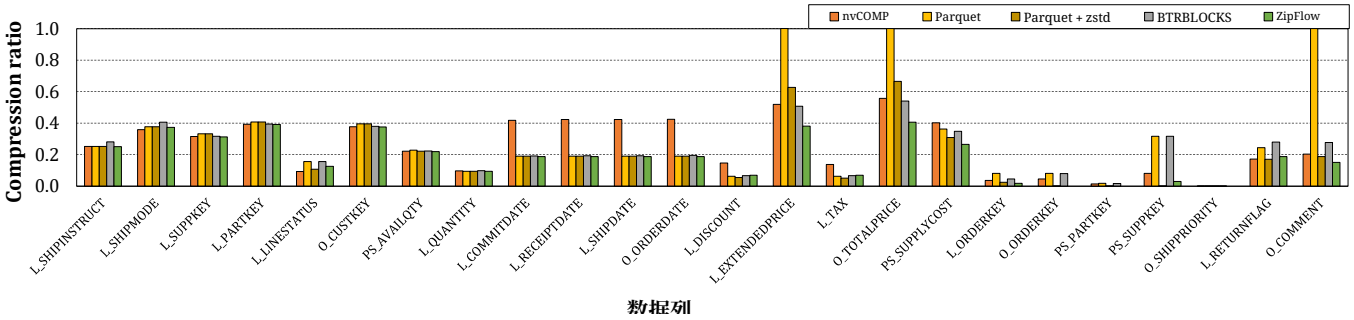


图 16：ZipFlow 与四种不同基线之间的压缩比比较。对于 Parquet 和 nvCOMP，我们遍历所有可能的算法选项，并仅报告最佳压缩比。对于 nvCOMP 的 Cascaded，我们遍历 RLE、增量编码和位打包的 72 ($6 \times 6 \times 2$) 种不同可能的嵌套层级。

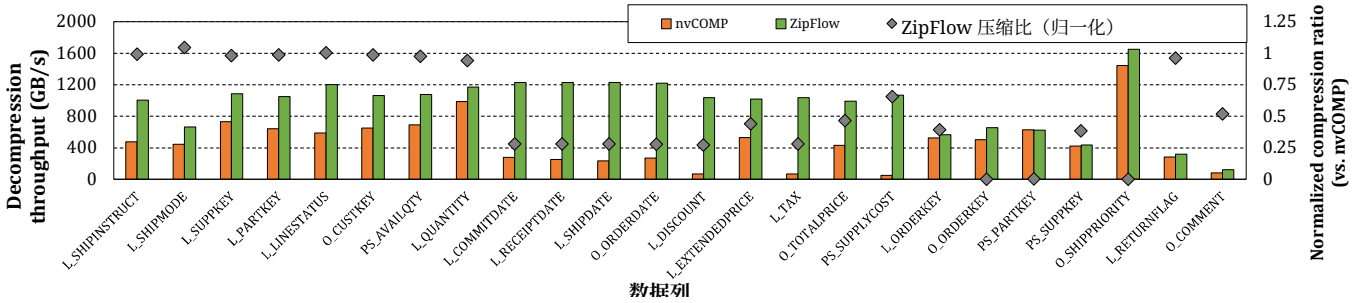


图 17：nvCOMP 与 ZipFlow 的解压吞吐量对比，其中菱形标记表示 ZipFlow 的压缩比相对于 nvCOMP 归一化后的结果。

L_DISCOUNT 到 PS_SUPPLYCOST 绘制的五列），我们的框架采用完全并行类别中的自定义浮点数转整数压缩技术，其压缩效率优于其他方法。对于存储主键的列（例如 O_ORDERKEY），我们采用一种结合轻量级方法的混合方案：分别嵌套组并行和完全并行类别中的 DeltaStride 与位打包算法。该策略非常适合编码“近似”单调递增的整数序列，从而显著提高压缩率。对于字符串值列（如 O_COMMENT），我们引入一种组并行类别中的定制字典构建方法，按空格和句点进行分词。由此得到一个包含 1，878 个唯一单词的紧凑字符串字典，索引还可进一步位打包为 12 位。解压缩时，每个唯一单词都作为组并根据查找字典进行扩展。字符串字典方法避免了 LZ77（非并行）算法高昂的解压开销。我们还将自定义字符串字典编码与位打包和 ANS 级联。实验结果表明，该方法的压缩率始终高于 nvCOMP（zstd）和 BTRBLOCKS（FSST [22]）。

总体而言，ZipFlow 扩大了可选自定义算法的范围，从而拓宽了可能的嵌套组合；这是针对不同列类型发现更优压缩方案的关键因素。

5.3.2 解压吞吐量。 本节评估 GPU 解压缩吞吐量及其对文件级数据移动的影响。与压缩实验不同，nvCOMP 充当

表2：每个文件采用的自定义压缩算法。“|”用于分隔各个嵌套算法。包含两个或更多输出的算法用“[,]”括起来。

数据列	嵌套算法
L_SHIPINSTRUCT	位打包
L_SHIPMODE	位打包
L_SUPPKEY	位打包
L_PARTKEY	位打包
行状态	位打包
O_CUSTKEY	位打包
供应商可用数量	位打包
行数量	位打包
L_COMMITDATE	字典编码 | 位打包
行收货日期	字典编码 | 位打包
L_SHIPDATE	字典编码 | 位打包
O_ORDERDATE	字典编码 | 位打包
L_DISCOUNT	浮点数转整数 | 位打包
L_EXTENDEDPRICE	浮点数转整数 | 位打包
L_TAX	浮点数转整数 | 位打包
订单总价	浮点数转整数 | 位打包
PS_SUPPLYCOST	浮点数转整数 | 位打包
L_ORDERKEY	游程编码 | [DeltaStride | [增量编码 | 游程编码 | [位打包，位打包], 位打包], 位打包]]
O_ORDERKEY	DeltaStride | [增量编码 |游程编码 | [位打包，位打包], 位打包]
零件键	游程编码 | [DeltaStride，游程编码]]
供应商键	增量编码 | 字典编码 | 位打包 | 字典编码 | 位打包
订单配送优先级	RLE
L_RETURNFLAG	ANS
O_COMMENT	字符串字典 | 位打包 | ANS

作为唯一的最先进基线进行比较，因为 Parquet 和 BTRBLOCKS 不具备在压缩数据传输至 GPU 后进行解压缩的能力。

在 nvCOMP 库中，我们选择能够通过最小化综合

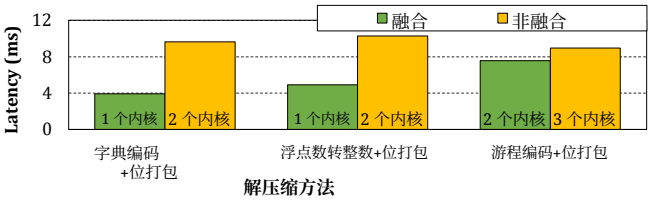


图 18：嵌套解压缩内核上的内核融合。

I/O 和解压缩延迟，在所提供的函数中实现最高端到端性能的算法。图 17 展示了 nvCOMP 和 ZipFlow 在所有数据列上的解压吞吐量，并通过灰色菱形标记进一步说明 ZipFlow 相较于 nvCOMP 的压缩比优势。最左侧的八个数据点对应采用位打包的列。对于这些列，nvCOMP 的最优算法是 Cascaded 函数中的位打包实现，其压缩比与我们的方法相同。对于从 L_COMMITDATE 到 PS_SUPPLYCOST 的列，nvCOMP 的性能明显较差。由于 nvCOMP 不支持字典编码和浮点数转整数算法，它只能采用 Bitcomp、Gdeflate、ANS 和 Cascaded 等多种替代技术来优化数据移动。然而，如图 17 所示，这些复杂算法在压缩比或解压吞吐量方面都未能带来优势。相比之下，ZipFlow 采用专用且轻量的完全并行方案，平均将压缩比提升 21%，同时显著提高了解压吞吐量。总而言之，与 nvCOMP 相比，ZipFlow 将文件级数据搬运开销平均降低了 2.07× 倍。

5.3.3 融合对解压缩的影响。 我们开展了一项消融研究，以评估内核融合对嵌套解压缩的影响。图 18 展示了三种嵌套函数上融合与非融合内核的延迟：浮点数转整数 +位打包、字典编码 + 位打包以及游程编码 + 位打包。每种算法分别应用于 L_SHIPDATE、L_EXTENDEDPRICE 和 L_ORDERKEY，其中三个文件的大小相同。字典编码 +位打包和浮点数转整数 + 位打包的融合内核仅执行一个内核，而游程编码 + 位打包则执行两个内核（即 PyTorch 的累积和与 ZipFlow 运算符）。非融合版本不对位打包运算符进行融合，因此分别执行两个、两个和三个独立的内核。

从实测来看，与非融合版本相比，内核融合使字典编码 2.45× 位打包的性能提升了 +，使浮点数转整数 + 位打包的性能提升了 2.08×。如此显著的性能差距源于片外内存访问的差异。在公式 2 中，我们说明了上述两种情况下融合与非融合内核所需内存流量的差异。单个融合内核的内存流量仅包括压缩大小和原始大小（即未压缩大小），而非融合内核则需要额外进行中间值的内存往返。由于单独使用字典编码和浮点数转整数并不会压缩数据，因此中间数据的开销按原始大小计算。

$$\frac{1\times(\text{compressed size})+3\times(\text{plain size})}{1\times(\text{compressed size})+1\times(\text{plain size})}>2 \qquad (2)$$

这一结果凸显了非融合流水线中哪怕只增加一次内存往返也会带来显著的性能损失。相比之下，对于 RLE + 位打包，我们观察到的加速幅度较为有限：内核融合相较于非融合基线使性能提升了 1.18×。这是因为位打包仅应用于体积计数，而该计数已由 RLE 压缩，且仅占原始数据的 12.5%。因此，融合内核带来的性能提升有限。

5.4 端到端 TPC-H 查询性能

我们在完整的 TPC-H 工作负载（全部 22 个查询）下，采用全流水线、端到端查询执行设置，将 ZipFlow 与多个基线进行基准测试。对于每个查询，我们仅将该查询所需的列传输到 GPU。压缩仅应用于表2中列出的列，而国家ID等其他列规模较小，进行压缩不会带来可测量的收益。在 GPU 内存中准备好所需数据后，我们将 TQP [33] 在同一 A100 GPU 上测得的查询执行延迟以合成方式纳入其中。为公平比较端到端成本，并均衡查询处理阶段的成本，我们在所有基于 GPU 的基线中采用 TQP 作为查询处理引擎，同时按以下方式改变压缩配置：（1）不应用压缩的方法（标记为“noCOMP”）；（2）使用 nvCOMP 库，并采用第 5.3.2 节确定的性能最佳算法。对于基于 CPU 的基线，我们将其与最先进的数据库系统 DuckDB 和 Microsoft SQL Server 进行比较。

5.4.1 查询整体延迟。 图19展示了所有配置下各查询的执行时间。基线 noComp 的延迟最高，因为 PCIe 数据传输占据了大部分运行时，平均占总执行时间的 91.3%。因此，nvCOMP 和 ZipFlow 都大幅降低了 PCIe 开销，延迟分别减少了 1.91× 和 4.08×。在整个基准测试中，ZipFlow 始终优于 nvCOMP，端到端平均加速比达到 2.08×。

尽管基于 GPU 的解决方案在大多数情况下都展现出显著的性能优势，但 CPU 基线在若干查询中仍具有竞争力：在 22 个查询中，DuckDB 和 SQL Server 分别有 6 个和 9 个查询的表现优于 nvCOMP 基线。这是因为 nvCOMP 的数据移动开销超过了 GPU 加速带来的收益。得益于 ZipFlow 对数据传输进行了更深入的优化，基于 GPU 的查询执行在除 TPC-H 6 之外的所有查询中都优于 DuckDB；在除 TPC-H 6、17 和 19 之外的所有查询中都优于 SQL Server。TPC-H 6 这类查询主要受一次性扫描操作支配，并且主要受内存带宽和 PCIe 带宽限制。由于 CPU 内存带宽远高于 PCIe 带宽，这些查询更适合在 CPU 上执行，将数据传输至 GPU 几乎无法带来收益。总体而言，ZipFlow 相比 DuckDB 的平均加速比为 3.14×，相比 SQL Server 的平均加速比为 3.32×。

5.4.2 ***GPU*** 解压缩期间的延迟。 图 20 将 ZipFlow 和 nvCOMP 的查询执行延迟分解为

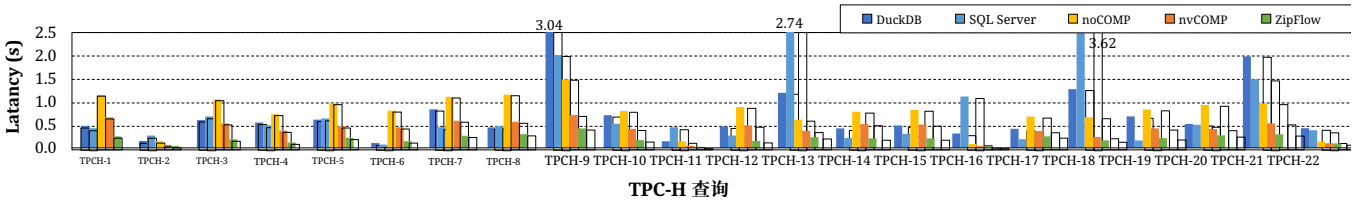


图 19：不同 CPU 和 GPU 基线下的 TPC-H 查询延迟对比。

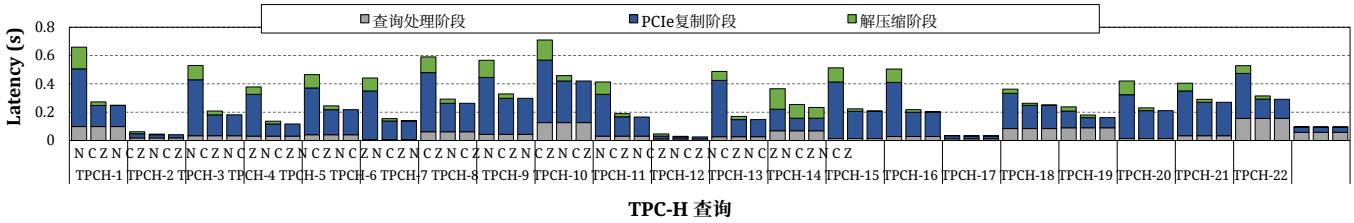


图 20：按执行阶段划分的 TPC-H 查询延迟——“N”表示 nvCOMP，“C”表示仅应用（解）压缩的 ZipFlow，“Z”表示包含 PCIe 流水线的完整 ZipFlow 实现。

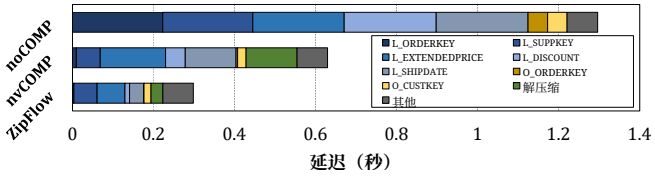


图 21：按输入列划分的 TPC-H 查询 7 执行延迟（“其他”指查询处理内核和未压缩数据传输所花费的开销）。

各个组成部分。为了更清楚地展示各部分对延迟的影响，我们绘制了标签为“C”的柱形，表示未对复制和解压阶段进行流水线处理的 ZipFlow；这清楚地显示了通过重叠这些操作所节省的延迟。平均而言，我们的方法使 GPU 内核延迟降低了 3.26×。在 22 个查询中，我们观察到，对于包含 L_TAX 和 L_DISCOUNT 列的查询（TPC-H 1、3、5–10、14、15 和 19），我们的方法与 nvCOMP 在 GPU 解压缩延迟方面存在显著差异。这一差距源于 nvCOMP 对 L_TAX 和 L_DISCOUNT 采用的最优算法是 Gdeflate，其解压速度相对较低，平均为 66.6GB/s。相比之下，我们提出的方法采用专用的浮点数转整数解压缩，平均速度达到 1030GB/s，从而有效降低了解压开销。

5.4.3 ***PCIe I/O*** 期间的延迟. 得益于所传输压缩数据规模的减小，ZipFlow 大幅降低了 PCIe I/O 延迟，平均而言，其改进幅度达到 nvCOMP 所实现改进的 1.85×。由于 PCIe 传输时间对压缩比高度敏感，所观察到的改进主要来自使用字典编码和浮点数转整数算法压缩的列，我们的框架在这些列上展现出显著的压缩比优势（如图 17 所示）。为进行更细致的考察，我们选取 TPCH-7 作为代表性示例，并在图 21 中进一步给出三种不同执行场景的详细延迟分解。我们的观察表明，在面向 TPCH-7 的基于 nvCOMP 的压缩策略中，L_EXTENDEDPRICE 与 L_SHIPDATE 贡献了对 I/O 延迟最为显著的部分。相比之下，我们提出的方案针对这些列采用了经过优化的压缩方法，从而使所有列之间的延迟分布更加均衡。有趣的是，尽管 L_ORDERKEY 和 L_SUPPKEY 与上述列具有相同的原始大小，它们对端到端延迟的影响却小得多。这是因为它们具有较高的压缩比和解压速度，因而仅占整体查询延迟的一小部分。在 TPCH-13 中也观察到类似模式，其延迟主要由 O_COMMENT 列主导，该列占数据传输总量的 89.9%。基于上述观察，我们得出结论：尽管压缩数据移动是一种有效的解决方案，但对少数数据模式缺乏压缩支持会成为明显的异常点。具体而言，当其他大小相近的列能够被高效压缩和解压缩时，不受支持的数据模式可能导致不成比例的延迟，从而使某些列在端到端查询性能中凸显为瓶颈。

5.4.4 ***PCIe/GPU*** 流水线的影响。我们通过比较启用与未启用流水线的 ZipFlow（在图 20 中分别标记为“Z”和“C”），分析流水线层的影响。除 TPCH-13 外，对于所有查询，流水线都显著降低了解压内核延迟的暴露部分，与非流水线执行相比，端到端查询延迟平均降低了 10%。我们观察到，在当前的压缩方法设计中，除 RLE/Delta 相关压缩外，所有列的主要瓶颈都是 PCIe I/O 延迟。因此，大部分 GPU 解压缩延迟都能得到充分隐藏，但这也表明 PCIe I/O 仍是整个查询流水线中的主要瓶颈。此观察揭示了设计者面临的一项关键优化机会：通过进一步嵌套复杂压缩方法来提高压缩比，可以缓解 PCIe I/O 瓶颈，但代价是解压吞吐量下降。还需注意的是，I/O 与解压缩延迟之间的权衡主要由 PCIe 与 GPU 内存之间的带宽差异决定，这也是未来值得探索的方向。

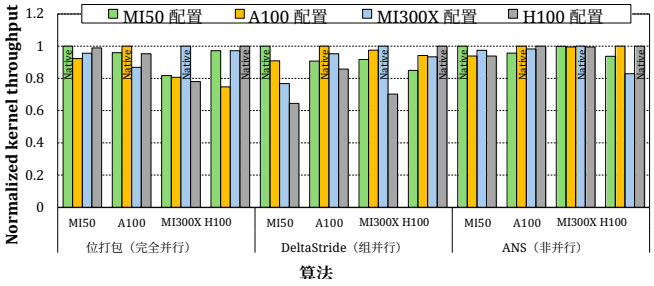


图22：“原生配置”和“共享配置”在 4 种 GPU类型上的内核效率。所有数值均以“原生配置”下相应的内核效率进行归一化。

5.5 异构 GPU 上的评估

本节分析 ZipFlow 配置在四种不同类型 GPU 上的影响，以及从配置空间中筛选最优候选项所产生的开销。尽管 GPU 在核心数量、缓存容量和其他架构资源方面各不相同，ZipFlow 仍能让针对一种 GPU 设计的内核在另一种 GPU 上运行。然而，采用非最优配置通常会导致解压性能不佳。鉴于最优内核配置可能因硬件平台而异，我们将针对特定 GPU 调优的配置定义为其“原生配置”，并将最初为其他 GPU 设计的配置称为“共享配置”。如图22所示，我们首先在四种具有代表性的 GPU（MI50 [3],A100 [1],H100 [6], 和 MI300x [7]）上微调所需算子，得到四种对应的“原生配置”。随后，我们在其余 GPU 上测试这些配置（作为“共享配置”），以评估其性能影响。在所有场景中，我们观察到，采用“共享配置”的 GPU 无法充分发挥其固有性能，性能最多下降 35%。

在真实世界的数据分析场景中，需要一种高效的离线调优方法，以便预先确定最优的“原生配置”，从而避免运行时使用“共享配置”所导致的性能下降。我们在经过精心缩小的搜索空间内采用强化学习，降低离线配置探索的成本；该搜索空间中的所有数值均限制为 2 的幂，具体见表3。作为基准，“B.F Search”（又称暴力搜索）是指探索搜索空间内所有有效的候选项组合。相比之下，“R.L. Search”（又称强化学习搜索）利用性能分布中隐藏的单调性来剪枝搜索空间，从而进一步将实际候选项探索成本降低至十以内。

6 相关工作

GPU加速查询执行。目前已开发出多种开源的基于 GPU 的数据库系统，例如 Ocelot [35], 、BlazingSQL [2],、HeavyDB [34], 和 RAPIDS [53]。大量研究广泛探索了在GPU硬件上优化查询执行的技术 [26, 30–32, 42, 51, 52, 58, 59, 61, 62]。一些研究还进一步探讨了利用 PyTorch 实现来加速基于 GPU 的查询处理 [20, 33]。构建

表3：NVIDIA 和 AMD GPU 上的模式探索成本。我们将模式类型标记为 F.P.（完全并行）、G.P.（组并行）和 N.P.（非并行）。

		$range(L)$	$range(S)$	$range(C)$
F.P.	配置空间	$2^{0..4}$	$Hw.WarpSize..2^{10}$	$\lceil 4/dtype.size \rceil$
	B.F. 搜索	$NVIDIA = 5 \times 6 \times 1$， $AMD = 5 \times 7 \times 1$		
	R.L. 搜索	$NVIDIA \approx 3 + 4 + 0$， $AMD \approx 3 + 4 + 0$		
G.P.	配置空间	$Hw.numCUs$	$Hw.WarpSize..2^{10}$	$2^{0..10}$
	B.F. 搜索	$NVIDIA = 1 \times 6 \times 11$， $AMD = 1 \times 7 \times 11$		
	R.L. 搜索	$NVIDIA \approx 0 + 4 + 5$， $AMD \approx 0 + 4 + 5$		
N.P.	配置空间	$Hw.numCUs$	$Hw.WarpSize$	$2^{0..10}$
	B.F. 搜索	$NVIDIA = 1 \times 1 \times 11$， $AMD = 1 \times 1 \times 11$		
	R.L. 搜索	$NVIDIA \approx 0 + 0 + 5$， $AMD \approx 0 + 0 + 5$		

基于以往研究取得的重大进展，我们的研究聚焦于下一个主要瓶颈：向 GPU 移动数据。

优化向 GPU 的数据移动。压缩已被广泛用于缓解向 GPU 传输数据时的数据移动瓶颈 [21, 29, 44, 48, 57, 65, 66]。先前的研究 [29, 44] 提出了规划器，根据压缩比和 CPU 解压吞吐量，选择轻量级、面向特定列的算法的最佳组合。多项研究强调了设计 GPU 友好型解压缩算法的重要性 [18, 57]。较新的研究 [21] 探索了将重量级压缩算法与即时数据解压缩相结合，以实现数据传输的细粒度流水线处理。我们的框架进一步补充了这些工作，提供灵活且可扩展的基础设施，用于开发经过优化的解压缩内核，从而支持更广泛的算法设计及其设备调度。管理基于 GPU 的数据分析中的内存限制，还有其他解决方案，包括混合使用 CPU 和 GPU [45, 47, 63], 、对数据移动进行流水线处理 [21, 55], 、通过惰性调度和传输共享最大限度地减少冗余移动 [54], , 以及在多 GPU 场景中利用所有 CPU-GPU PCIe 链路 [64]。

7 结论

本文提出了基于编译器的框架 ZipFlow，以系统性地加速从 CPU 到 GPU 的压缩数据移动。该方法利用数据特定特征和 GPU 架构特征，缓解了传输未压缩数据所带来的显著 PCIe开销。

压缩比和解压吞吐量都是关键且相互依赖的指标，必须联合优化，才能最大限度地提高端到端传输效率。要实现较高的压缩比，很大程度上取决于可用压缩算法池的多样性和适应性；而要获得较高的解压吞吐量，则依赖复杂的内核调度策略。ZipFlow 通过引入三种新颖的设计模式，应对这些挑战，从而支持在 GPU 上对各种压缩类别进行灵活优化。这种方法不仅使用户能够高效组合自定义算法，以支持新颖的压缩策略和嵌套，还能确保这些算法在异构设备上具有较高的解压吞吐量。实验结果表明，与针对有限设备集定制的预定义解决方案相比，该方法具有显著优势。

参考文献

[1] 2020. NVIDIA A100 张量核心 GPU。https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf。访问日期：2025 年 1 月 4 日。[2] 2021. blazingsql。https://github.com/BlazingDB/blazingsql。访问日期：2025 年 1 月 4 日。[3] 2023. Instinct MI50 (32GB) 规格。https://www.amd.com/en/support/downloads/drivers.html/accelerators/instinct/instinct-mi-series/instinct-mi50-32gb.html#amd_support_product_spec。访问日期：2025 年 1 月 4 日。[4] 2024. Apache Parquet。https://parquet.apache.org/。访问日期：2025 年 1 月 4 日。[5] 2024. CUDA C++ 编程指南。https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html。[6] 2024. NVIDIA H100 张量核心 GPU。https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet?ncid=04, no-ncid。访问日期：1 月 2025 日。[7] 2025. AMD MI300x。https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/data-sheets/amd-instinct-mi300x-data-sheet.pdf。访问日期：2025 年 1 月 4 日。[8] 2025. DuckDB—— 进程内 SQL OLAP 数据管理系统。https: //duckdb.org/。访问日期：2025 年 1 月 4 日。[9] 2025. FiniteStateEntropy。https://github.com/Cyan4973/FiniteStateEntropy。访问日期：2025 年 1 月 4 日。[10] 2025. HIPIFY。https://github.com/ROCm/HIPIFY。访问日期：2025 年 1 月 4 日。[11] 2025. lz4。https://github.com/lz4/lz4。访问日期：2025 年 1 月 4 日。[12] 2025. Microsoft SQL Server。https://www.microsoft.com/en-us/sql-server。访问日期：2025 年 1 月 4 日。[13] 2025. nvCOMP：使用 NVIDIA GPU 进行高速数据压缩。https: //developer.nvidia.com/nvcomp。访问日期：2025 年 1 月 4 日。[14] 2025. NVLink 和 NVSwitch：速度最快的 HPC 数据中心平台。https://www.nvidia.com/en-us/data-center/nvlink/。访问日期：2025 年 1 月 4 日。[15] 2025. snappy。https://github.com/google/snappy。访问日期：2025 年 1 月 4 日。[16] 2025. zstd。https://github.com/facebook/zstd。访问日期：2025 年 1 月 4 日。[17] Azim Afroozeh 和 Peter Boncz。2023。FastLanes 压缩布局：使用标量代码每秒解码 > 100 十亿个整数。《VLDB基金会会刊》16, 9 2023 2132 2144 10 14778 3598581 3598587 （5 月），–。[18] 2024. https://doi.org/ . / 。Azim Afroozeh、Lotte Felius 和 Peter Boncz。. 使用 FastLanes 压缩加速 GPU 数据处理。载于 DaMoN '24：第 20 届新型硬件数据管理国际研讨会论文集。ACM，1–11。https://doi.org/ 10 1145 3662010 3663450 . / 。Azim Afroozeh、Leonardo Kuffó 和 Peter Boncz。2024。ALP：自适应无损浮点压缩。载于 SIGMOD '24：2024 年 ACM SIGMOD 国际数据管理会议论文集。ACM，1–13。https: //doi.org/10.1145/3626717[19] Yuki Asada、Victor Fu、Apurva Gandhi、Advitya Gemawat、Lihao Zhang、Dong He、Vivek Gupta、Ehi Nosakhare、Dalitso Banda、Rathijit Sen 和 Matteo Interlandi。2022。共享张量的魅力：数据库如何利用机器学习生态系统。《VLDB基金会会刊》15, 12（2022 年 8 月），3598–3601。https://doi.org/10.14778/3554821.3554853[20] Nils Boeschen、Tobias Ziegler 和 Carsten Binnig。2024。GOLAP：用于高速 OLAP 的 GPU 数据路径架构。《ACM数据管理会刊》2, 6，文章237 2024 26 10 1145 3698812 （12 月），页码。[21] 2020. https://doi.org/ . / 。Peter A. Boncz、Thomas Neumann 和 Viktor Leis。. FSST：快速随机访问字符串压缩。《VLDB基金会会刊》13, 11（2020），2649–2661。http: //www.vldb.org/pvldb/vol13/p2649-boncz.pdf[22] Jiashen Cao、Rathijit Sen、Matteo Interlandi、Joy Arulraj 和 Hyesoon Kim。2023。GPU 数据库系统的特征分析与优化。《VLDB基金会会刊》17, 3 2023 441 454 10 14778 3632093 3632107 （），–。[23] 2024. https://doi.org/ . / 。Wei Cui、Qianxi Zhang、Jesús Camacho-Rodríguez、Spyros Blanas、Brandon Haynes、Yinan Li、Peng Cheng、Ravishankar Ramamurthy、Rathijit Sen 和 Matteo Interlandi。2023。游戏主机上的查询处理。载于第 19 届新型硬件数据管理国际研讨会（DaMoN '23），2023 年 6 月 18–23 日，美国华盛顿州西雅图。https://www.microsoft.com/en-us/research/publication/query-processing-on-gaming-consoles/[25] Patrick Damme、Dirk Habich、Juliana Hildebrandt 和 Wolfgang Lehner。2017。轻量级数据压缩算法：实验调查（实验与分析）。载于第 20 届扩展数据库技术国际会议论文集（EDBT 2017），意大利威尼斯，2017 年 3 月 21–24 日，Volker Markl、Salvatore Orlando、Bernhard Mitschang、Periklis Andritsos、Kai-Uwe Sattler 和 Sebastian Breß（编）。OpenProceedings.org，72–83。https: //doi.org/ 10 5441 002 2017 08 . / /EDBT. 。Shaleen Deep、Hangdong Zhao、Austen Z. Fan 和 Paraschos Koutris。2024。输出敏感的合取查询求值。《ACM数据管理会刊》2, 5（2024 220 1 220 24 10 1145 3695838），：–：。[26] 1996. https://doi.org/ . / L. Peter Deutsch。. DEFLATE 压缩数据格式规范版本 1 3 1951 10 17487 1951 . 。RFC 。https://doi.org/ . /RFC Jarek Duda、Khalid Tahboub、Neeraj J. Gadgil 和 Edward J. Delp。2015。将非对称数制用作霍夫曼编码的精确替代方案。[27] 2015. 6th International Conference on Picture Coding Symposium (PCS)。载于2015 中国图像编码研讨会（PCS）。65–69。[28] Wenbin Fang、Bingsheng He 和 Qiong Luo。2010。图形处理器上的数据库压缩。《VLDB基金会会刊》3，1（2010），670–680。https://doi.org/10. 14778/1920841.1920927[29] Henning Funke、Sebastian Breß、Stefan Noll、Volker Markl 和 Jens Teubner。2018。协处理器环境中的流水线查询处理。载于《2018年国际数据管理会议论文集》（美国得克萨斯州休斯敦）（SIGMOD '18）。美国计算机协会，美国纽约州纽约市，1603 1618 10 1145 3183713 3183734 – 。[30] 2020. https://doi.org/ . / . Henning Funke 和 Jens Teubner。. 非均匀数据上的数据并行查询处理。《VLDB基金会会刊》13，6（2020年2月），884–897。https://doi.org/10.14778/3380750.3380758[31] Bingsheng He、Ke Yang、Rui Fang、Mian Lu、Naga Govindaraju、Qiong Luo 和 Pedro Sander。2008。图形处理器上的关系连接。载于《2008年ACM SIGMOD国际数据管理会议论文集》（加拿大温哥华）（SIGMOD '08）。美国计算机协会，美国纽约州纽约市，511–524。https://doi.org/10.1145/1376616.1376670[32] Dong He、Supun C Nakandala、Dalitso Banda、Rathijit Sen、Karla Saur、Kwanghyun Park、Carlo Curino、Jesús Camacho-Rodríguez、Konstantinos Karanasos 和 Matteo Interlandi。2022。张量计算运行时上的查询处理。《VLDB基金会会刊》15，11（2022年7月），2811–2825。https://doi.org/10.14778/ 3551793.3551833[33] HEAVYDB.AI。2025。全球最快的分析型数据库。由 GPU 驱动。https://www.heavy.ai/product/heavydb。[34] Max Heimel、Michael Saecker、Holger Pirk、Stefan Manegold 和 Volker Markl。2013。内存列存储的硬件无关并行性。《VLDB基金会会刊》6, 9 2013 709 720 10 14778 2536360 2536370 （7 月），–。[35] 2022. https://doi.org/ . / . Sven Hepkema、Azim Afroozeh、Charlotte Felius、Peter Boncz 和 Stefan Manegold。2025。G-ALP：重新思考适用于 GPU 的轻量级编码。载于第21届新型硬件数据管理国际研讨会论文集（DaMoN '25）。美国计算机协会，美国纽约州纽约市，文章 11, 10 10 1145 3736227 3736242 页。https://doi.org/ . / . Yu-Ching Hu、Yuliang Li 和 Hung-Wei Tseng。. TCUDB：使用张量处理器加速数据库。载于 SIGMOD '22：国际数据管理会议，美国宾夕法尼亚州费城，2022年6月12日至17日，Zachary G. Ives、Angela Bonifati 和 Amr El Abbadi（编）。ACM，1360–1374。https://doi.org/10. 1145/3514221.3517869[36] Zezhou Huang、Krystian Sakowski、Hans Lehnert、Wei Cui、Carlo Curino、Matteo Interlandi、Marius Dumitru 和 Rathijit Sen。2025。基于压缩数据的 SQL 分析加速。arXiv:2506.10092 [cs.DB] https://arxiv.org/abs/ 2506.10092[37] David A. Huffman。1952。最小冗余码的构造方法。《IRE 论文集》40，9（1952），1098–1101。https: //doi.org/ 10 1109 1952 273898 . /JRPROC. . Selmer Martin Johnson。. 包含准备时间的两阶段和三阶段最优生产调度。《海军研究后勤季刊》1，1（1954），61–68。[38] 1954. [39] Maximilian Kuschewski、David Sauerwein、Adnan Alhomssi 和 Viktor Leis。2023。BtrBlocks：面向数据湖的高效列式压缩。《ACM数据管理会刊》1，2（2023），118:1–118:26。https://doi.org/10.1145/3589263[40] Harald Lang、Andreas Kipf、Linnea Passing、Peter Boncz、Thomas Neumann 和 Alfons Kemper。2018。充分利用 SIMD 投资：在编译型查询流水线中应对控制流分歧。载于第14届新型硬件数据管理国际研讨会论文集（得克萨斯州休斯敦）（DAMON '18）。美国计算机协会，美国纽约州纽约市，文章5，8页。https://doi.org/10.1145/3211922.3211928[41] Rubao Lee、Minghong Zhou、Chi Li、Shenggang Hu、Jianping Teng、Dongyang Li 和 Xiaodong Zhang。2021。平衡的艺术：构建 CPU/GPU 混合数据库产品的 RateupDBTM 实践。《VLDB基金会会刊》14，12（2021年7月 2021 2999 3013 10 14778 3476311 3476378 ），–。https://doi.org/ . / . Jing Li[42]、 Hung-Wei Tseng 、 Chunbin Lin 、 Yannis Papakonstantinou 和 Steven Swanson。2016。HippogriffDB：在大数据分析中平衡 I/O 与 GPU 带宽。《VLDB基金会会刊》9，14（2016年10月），1647–1658。https://doi.org/10. 14778/3007328.3007331[43] Yinan Li、Bailu Ding、Ziyun Wei、Lukas M. Maas、Momin Al-Ghosien、Spyros Blanas、Nicolas Bruno、Carlo Curino、Matteo Interlandi、Craig Peeper、Kaushik Rajan、Surajit Chaudhuri 和 Johannes Gehrke。2025。突破 GPU 内存容量限制，扩展 GPU 加速数据库。《VLDB基金会会刊》18，11（2025年9月），4518 4531 10 14778 3749646 3749710 – 。https://doi.org/ . / . Hanwen Liu、Mihail Stoian、Alexander van Renen 和 Andreas Kipf。2024。Corra：相关性感知列压缩。arXiv 预印本 arXiv:2403.17229（2024）。[44] Haotian Liu、Bo Tang、Jiashu Zhang、Yangshen Deng、Xiao Yan、Xinying Zheng、Qiaomu Shen、Dan Zeng、Zunyao Mao、Chaozu Zhang、Zhengxin You、Zhihao Wang、Runzhe Jiang、Fang Wang、Man Lung Yiu、Huan Li、Mingji Han、Qian Li 和 Zhenghai Luo。2022。GHive：通过 CPU-GPU 异构计算加速 Apache Hive 中的分析型查询处理。载于第13届云计算研讨会论文集，SoCC 2022，美国加利福尼亚州旧金山，2022年11月7日至11日，Ada Gavrilovska、Deniz Altinbüken 和 Carsten Binnig（编）。ACM，[45] 2016. [46]

158 172 10 1145 3542929 3563503 – 。https://doi.org/ . / . Hamish Nicholson、Konstantinos Chasialis、Antonio Boffa 和 Anastasia Ailamaki。2025。压缩对 GPU 加速查询处理内存溢出数据集的有效性。载于第21届新型硬件数据管理国际研讨会论文集（DaMoN '25）。美国计算机协会，美国纽约州纽约市，文章10，10页。https: // doi.org/ 10 1145 3736227 3736240 . / . Nikolay Sakharnykh 和 Rene Mueller。2020。面向分析型工作负载的基于软件的压缩。在 NVIDIA GTC 2020 上展示。https://developer.nvidia. com/gtc/2020/video/s21597-vid。[50] Apache ORC。2025。适用于 Hadoop 工作负载的最小、最快列式存储。https://orc.apache.org/。[51] Johns Paul、Bingsheng He、Shengliang Lu 和 Chiew Tong Lau。2019。重新审视图形处理器上的哈希连接：十年之后。载于第35届 IEEE 国际数据工程会议研讨会，ICDE Workshops 2019，中国澳门，4月 *8 12, 2019* 294 299 10 1109 2019 00008 - 。IEEE，– 。https://doi.org/ . /ICDEW. . Johns Paul、Bingsheng He、Shengliang Lu 和 Chiew Tong Lau。 . 提高基于即时编译的 GPU 查询处理执行效率。《VLDB基金会会刊》14，2（2020年10月），202–214。https://doi.org/10.14778/3425879. 3425890[53] RAPIDS。2025。GPU 加速数据科学。https://rapids.ai/。[54] Aunn Raza、Periklis Chrysogelos、Panagiotis Sioulas、Vladimir Indjic、Angelos-Christos G. Anadiotis 和 Anastasia Ailamaki。2020。历经时间考验的 GPU 加速数据管理。载于第10届创新数据系统研究会议，CIDR 2020，荷兰阿姆斯特丹，2020年1月12日至15日，在线论文集。www.cidrdb.org。http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf[55] Ran Rui、Hao Li 和 Yi-Cheng Tu。2020。多 GPU 环境下大型数据库表的高效连接算法。《VLDB基金会会刊》14，4（2020），708 720 10 14778 3436905 3436927 – 。https://doi.org/ . / . Anil Shanbhag、Samuel Madden 和 Xiangyao Yu。2020。GPU 和 CPU 在数据库分析中的基本性能特征研究。载于《2020年国际数据管理会议论文集》，SIGMOD Conference 2020，在线会议 *[*美国俄勒冈州波特兰*],* 2020年6月14日至19日，David Maier、Rachel Pottinger、AnHai Doan、Wang-Chiew Tan、Abdussalam Alawini 和 Hung Q. Ngo（编）。ACM，1617–1632。https://doi.org/10.1145/ 3318464.3380595[57] Anil Shanbhag、Bobbi W. Yogatama、Xiangyao Yu 和 Samuel Madden。2022。GPU 上基于瓦片的轻量级整数压缩。载于 SIGMOD '22：国际数据管理会议，美国宾夕法尼亚州费城，2022年6月12日至17日，Zachary G. Ives、Angela Bonifati 和 Amr El Abbadi（编）。ACM，1390–1403。https://doi.org/ 10 1145 3514221 3526132 . / . Panagiotis Sioulas、Periklis Chrysogelos、Manos Karpathiotakis、Raja Appuswamy 和 Anastasia Ailamaki。2019。GPU 上的硬件感知哈希连接。载于第35届 IEEE 国际数据工程会议，ICDE 2019，中国澳门，2019年4月8日至11日。IEEE，698–709。https://doi.org/10.1109/ICDE. 2019.00068[59] Elias Stehle 和 Hans-Arno Jacobsen。2017。GPU 上内存带宽高效的混合基数排序。载于《2017年ACM国际数据管理会议论文集》（美国伊利诺伊州芝加哥）（SIGMOD '17）。美国计算机协会，美国纽约州纽约市，417–432。https: //doi.org/ 10 1145 3035918 3064043 . / . Robin Vonk 、 Joost Hoozemans 和 Zaid Al-Ars。2025。GSST：在 GPU 上以 191 GB/s 并行解压缩字符串。载于第5届高效高性能存储系统的挑战与机遇研讨会论文集（荷兰鹿特丹）（CHEOPS '25）。美国计算机协会，美国纽约州纽约市， 8 14 10 1145 3719330 3721228 – 。https://doi.org/ . / . Bowen Wu、Wei Cui、Carlo Curino、Matteo Interlandi 和 Rathijit Sen。2025。转瞬之间完成太字节级分析。CoRR abs/2506.09226（2025）。https://doi.org/ 10 48550 2506 09226 2506 09226 . /ARXIV. . arXiv: . Bowen Wu、Dimitrios Koutsoukos 和 Gustavo Alonso。2025。高效处理 GPU 上的连接与分组聚合。《ACM数据管理会刊》3，1，文章 39 2025 27 10 1145 3709689 （2月），页数。https://doi.org/ . / Bobbi W. Yogatama、Weiwei Gong 和 Xiangyao Yu。 . 在异构 CPU-GPU 数据库管理系统中统筹数据放置与查询执行。《VLDB基金会会刊》 15, 11 2022 2491 2503 10 14778 3551793 3551809 （），– 。https://doi.org/ . / . Yichao Yuan、Advait Iyer、Lin Ma 和 Nishil Talati。 . Vortex：克服 GPU 加速大规模数据分析中的内存容量限制。《VLDB基金会会刊》18，4（2025年5月），1250–1263。https://doi.org/10.14778/3717755. 3717780[65] Yuan Yuan、Rubao Lee 和 Xiaodong Zhang。2013。在 GPU 设备上处理数据仓库查询的阴与阳。《VLDB基金会会刊》6，10（2013），817–828。https://doi.org/10.14778/2536206.2536210[66] Hongyu Zhu、Ruofan Wu、Yijia Diao、Shanbin Ke、Haoyu Li、Chen Zhang、Jilong Xue、Lingxiao Ma、Yuqing Xia、Wei Cui、Fan Yang、Mao Yang、Lidong Zhou、Asaf Cidon 和 Gennady Pekhimenko。2022。ROLLER：面向深度学习的快速高效张量编译。载于第16届 USENIX 操作系统设计与实现研讨会，OSDI 2022，美国加利福尼亚州卡尔斯巴德，2022年7月11日至13日，Marcos K. Aguilera 和 Hakim Weatherspoon（编）。USENIX 协会，233–248。https://www.usenix.org/conference/osdi22/presentation/zhu