arXiv:2611.30849v1 [cs.DB] 23 Nov 2026

SYCLDB：跨厂商异构分析型数据库管理系统

Ivan Donchev Kabadzhov、Eugenio Marinelli 和 Raja Appuswamy

EURECOM，Biot，法国

摘要。 人工智能和分析型工作负载的快速增长，提高了对快速、交互式分析型数据处理的需求。这进而推动了人们对设计加速型分析数据库引擎的浓厚兴趣，以利用现代 GPU 扩展计算能力。尽管 GPU 提供了现代分析所需的大规模数据并行性，但软件生态仍由闭源、厂商锁定的编程模型（如 CUDA）主导，限制了跨多厂商 GPU 的可移植性。在本工作中，我们提出了 SYCLDB——一个基于 SYCL 开放标准构建的端到端分析型数据库引擎，旨在通过单一代码库，在各种加速器上实现高效的硬件无关执行。
SYCLDB 将核心关系算子实现为模块化 SYCL 内核，将其与基于 Apache Calcite 和 Apache Thrift 的 SQL 前端集成，并支持 Apache Arrow 以实现高效的列式数据摄取，从而减少对厂商特定 GPU 编程栈的依赖，使单一 SYCL 代码库能够在英特尔、 AMD 和 NVIDIA GPU 上执行分析型查询管道。

关键词： 分析型数据库 · SYCL · 异构计算 · GPU加速 · 跨厂商可移植性。

1 引言

人工智能和分析在几乎所有应用领域日益普及，使企业和科研机构收集的数据量迅速增长；预计到2025年，全球数据圈将达到250ZB。现代分析和人工智能工作负载极为多样，涵盖各种标量、向量和矩阵计算。因此，计算行业传统的主力——通用CPU——无法针对这类异构工作负载的多样化需求进行优化。
CPU的这一局限，加上登纳德缩放定律不再适用，推动了人们对采用GPU加速数据库引擎的兴趣，这类引擎能够满足大规模数据分析工作负载的需求。遗憾的是，如今大多数流行方案主要面向NVIDIA GPU开发，因此依赖专有、闭源的CUDA软件栈，这阻碍了它们在其他厂商的GPU和加速器上的可移植性

因此，我们需要开发一种开放、基于标准且跨供应商的硬件加速方案，让数据分析惠及大众。

过去几年间，SYCL 已发展成为一种高级编程模型，为主流 C++ 带来硬件加速，旨在让开发者能够轻松编写类似于 CUDA 的新软件。与其他竞争性编程模型相比，使用 SYCL 表达数据并行计算有两大优势。首先，SYCL 基于可移植的高级编程语言，通过实现开放标准构建而成；相比之下，其他数据并行语言要么是专有的（CUDA），要么本质上属于低级语言（OpenCL）。因此，SYCL 支持开发可移植、跨架构的应用程序。其次，SYCL 天生采用单源编程方式，允许主机和设备代码共存于同一个文件中，不同于其他编程模型。这使 SYCL 编译器能够分析和优化整个程序，而不受代码运行设备的影响。

近期研究考察了以下方面的性能与可移植性权衡：（i）使用 SYCL 加速关键的高性能计算应用程序 [4]，；（ii）利用 SYCL 编程模型中彼此独立的特性 [3, 5, 7]。此前只有一项研究（XJoin [8]）聚焦于在数据库引擎设计中使用 SYCL，但其研究范围仅限于对独立的投影和哈希连接算子进行内核级性能基准测试。因此，尽管已有这项研究，仍有若干尚未得到充分探讨的问题。首先，内核级微基准测试只能反映整体性能的一个方面。为了了解 SYCL 在数据库引擎设计中的实用性，有必要实现一个功能完备、基于 SYCL 的查询执行器，使其能够解读应用基准测试中的复杂查询计划，并将基于 SYCL 的设计与最先进的 CUDA 实现进行比较 [6,11,14]。其次，为了了解使用 SYCL 开发下一代分析引擎的能力，还需要研究 SYCL 工具链与可插拔 SQL 前端的集成能力，以展示完全解释式的端到端 SQL 查询执行。

在这项工作中，我们介绍了**SYCLDB**——一款跨架构、性能可移植的 e, 查询执行引擎。它不仅能利用 NVIDIA GPU（如所有基于 CUDA 的相关工作 [2, 6, 13–15]）加速 SQL 查询执行，还能通过单一代码库利用其他厂商的 GPU（如 Intel Flex、AMD Instinct 等）。**SYCLDB** 的核心是将关键关系算子实现为一系列经过高度调优的 SYCL 独立内核。为从 SQL 前端驱动 SYCL 内核，**SYCLDB** 集成了 Apache Calcite [1] 和 Apache Thrift [12]，构建了可组合的查询解析与优化前端，可将 SQL 查询转换为由 SYCL 内核构成的有向无环图。

使用**SYCLDB**，我们在 NVIDIA , AMD 和英特尔硬件，结果表明：(i) 在端到端查询执行中，**SYCLDB** 的性能优于最先进的 GPU 数据库系统（与 HeavyDB 相比，平均加速比最高达 1.22×，与 SiriusDB 相比最高达 4.72×），也优于领先的内存数据库 DuckDB，快 4.7× 至 10.7×（平均6.37×）；(ii) **SYCLDB**可在多厂商 GPU 上实现完全可移植，从而能在所有评测平台上执行端到端 SQL 查询，并展示了 SYCL 出色的性能可移植性及其已做好

与标准 SQL 前端集成的准备。我们将**SYCLDB** 公开提供[1]，以支持进一步研究基于标准的分析加速。

2 设计

本节介绍**SYCLDB**库的设计，同时重点说明与**SYCLDB**设计相关的 SYCL 关键特性。

2.1 内存管理

SYCLDB 针对高吞吐量分析型工作负载进行了优化。因此，与先进的 GPU 数据库引擎类似 [6,11,14],，它采用列式存储架构，其中每个表属性都作为一个独立且连续的数组进行维护。**SYCLDB** 支持存储在兼容 Apache Arrow 的列式缓冲区中的数据，从而能够直接从外部数据源摄取数据。与其他 GPU 数据库引擎类似，**SYCLDB** 目前也仅设计用于所有数据都驻留在 GPU 内存中的场景。由于我们的目标是实现跨多厂商 GPU 的可移植性，因此我们不假设存在 NVLink 这类高性能互连，而只考虑连接主机CPU与 GPU 的低带宽 PCIe 链路。在此类场景中，众所周知 [11]，PCIe 数据传输会成为查询执行的瓶颈。为避免这一问题，与其他 GPU 引擎类似，**SYCLDB** 会在启动阶段读取所有输入列，并将其移动到 GPU 内存中。基于 CUDA 的数据库引擎依赖显式函数在主机（CPU）内存与设备（GPU）内存之间分配和传输内存，例如 cudaMalloc 和 cudaMemcpy。此外，CUDA 还提供固定内存，可确保主机与设备之间的数据传输更加高效。在 SYCL 中，统一共享内存（USM）模型提供了类似的功能 [10]，我们利用该模型将数据从主机内存迁移到 GPU 设备内存。

2.2 SYCLDB 关系运算原语

SYCLDB 的核心是一个关系运算原语库，这些原语可用作构建模块，用于实现更高层次的关系算子。共有四种基本原语，即选择、投影、构建和探测。

选择与投影 为了支持选择和投影关系运算，**SYCLDB** 支持一种通用的逐元素选择原语，用于计算布尔谓词。SYCLDB 目前面向定宽数值列，其中选择谓词表示为输入列与常量参考值之间的二元比较。为了支持多种比较运算，我们将选择算子实现为函数对象的基本 switch 语句，以决定所执行的二元运算。图 1a 展示了作用于单个元素（col）的基本选择原语。它使用比较类型 comp 将 col 与值 v 进行比较，并设置所得的布尔值。

[1] https://gitlab.eurecom.fr/dislab/sycldb

```
template <typename T , typename Compare >bool selection_
primitive ( const T& col , const T& v) { Compare comp {}; return
comp ( col , v); } （a）作用于单个元素的通用选择原语 // 传递 1：第一
个选择谓词 bool * flags =sycl :: malloc_device <bool >( F. len , q);
q. parallel_for ( sycl :: range <1>( F. len ) , [=]( sycl :: id <1>idx ) {
size_t i = idx[0]; flags[i] =selection_primitive <int , sycl :: less
<>>(F. data_A[i] , 100) ; }) ; （b）从 SYCL 内核调用选择原语 // 传递
1：第一个选择谓词 bool * flags =sycl :: malloc_device <bool >( F.
len , q); q. parallel_for ( sycl :: range <1>( F. len ) , [=]sycl :: id <1>
idx ) { size_t i = idx[0]; flags[i] =selection_primitive <int , sycl ::
less <>>(F. data_A[i] , 100) ; }) ; // 传递 2：第二个选择谓词 q.
parallel_for ( sycl :: range <1>( F. len ) , [=]sycl :: id <1>idx ) { size_
t i = idx[0]; flags[i]&=selection_primitive <int , sycl :: less <>>
(F. data_B[i] , 50) ; }) ; （c）逐内核执行策略 列表 1：SYCLDB 原语及
算子执行策略
```

构建与探测 为了支持关系哈希连接算子，**SYCLDB** 提供了两个核心原语：构建 和探测。当前实现针对定宽键列上的基于哈希的等值连接，匹配星型模式分析型工作负载中常见的连接模式，例如星型模式基准测试（SSB） [9]。我们采用一种无指针的基于数组的哈希表，使用线性探测和简单的取余哈希。构建 原语从构建关系填充哈希表。随后，探测 原语会从探测关系中查找匹配项。

2.3 关系算子

SYCLDB 利用前文列出的独立原语实现更高层的关系算子，以大规模并行方式在不同GPU后端上处理一个或多个输入关系。与其他加速器编程方法类似，SYCL模型区分主机CPU和加速器设备。SYCL中的关键数据并行编程构造是内核，其执行由主机协调。SYCL提供队列 抽象，使主机代码能够向设备发送命令。队列以提交的形式接收工作请求，每个工作请求都描述一个数据并行内核。提交是一种异步操作；请求进入队列，等待在关联设备上执行后，控制权便返回主机代码。与要求明确指定内核维度的CUDA不同，SYCL允许开发者指定一个简单的一维范围，该范围覆盖需要处理的给定输入数组，并由编译器确定在GPU线程间分配数据的最有效策略。**SYCLDB** 利用这一功能简化关系运算符内核的实现。

选择运算符。 每个选择条件都必须在一列与一个固定参考值之间应用二元运算（即 SELECT x FROM RWHERE $x < v$）。同一查询中可以出现多个此类条件。为实现高效的无物化条件求值，**SYCLDB** 会像其他分析引擎一样，跟踪每个关系的选择标志向量。这些向量用于编码每个元组在经过截至目前已求值的谓词后是否仍处于活动状态。对于合取谓词，每个新谓词的结果都会与之前的标志值组合，因此只有满足截至当前已处理的所有条件的元组才会保持活动状态。清单 1 中的代码展示了包含两个条件的选择运算符所对应的代码。代码包含几个关键变量：表示设备（GPU）上下文的队列（q）、待处理的输入关系（F）、用于存储比较运算结果的选择标志数组（sf），以及关系的基数（F.len）。此外，代码还包含比较运算符（sycl::less），以及用于与列（第一个/第二个条件分别对应 data_A/data_B 中的条目）进行比较的常量值（第一个/第二个条件分别为 100/50）。在清单 1 中，选择原语对单个谓词求值并返回其布尔结果。谓词组合由运算符内核处理。第一次遍历会初始化标志向量，后续遍历则将每个标志与新求值的谓词组合，从而进行更新。内核主体针对大小为 F.len 的一维范围执行简单的并行for循环。在循环内部，选择原语会以适当的参数对索引 i 处的每个元素进行调用。这种设计使每次谓词遍历都保持简洁，同时允许前端组合受支持谓词的任意合取，并保留可移植的逐内核执行模型。目前，**SYCLDB**支持定宽数值属性和简单的比较谓词，例如 $=$、$<$、$>$、$\leq$ 和 $\geq$；这些谓词会直接在 SYCL 内核中执行。

连接算子。 连接算子通过两次独立的内核调用实现，分别映射构建原语和探测原语。当前算子支持基于定宽键的等值连接，足以表达评估中使用的星型模式基准测试（SSB）连接模式。构建原语生成的哈希表会被物化，并通过 GPU 全局内存提供给探测内核。探测内核还会将连接值的结果向量物化到 GPU 全局内存中，供下游算子（如聚合）使用。

聚合算子。 全局聚合对于执行求和、计数和最小值/最大值等 SQL 运算符至关重要。SYCLDB 目前支持对定宽数值列进行此类聚合，聚合操作满足结合律和交换律。最先进的 GPU 全局聚合实现（例如 Crystal [11], ）通常会显式编排多级层次结构：使用子组洗牌进行线程束内通信，使用局部内存进行线程束间同步，并使用全局原子操作进行组间最终化。例如，图 1 展示了求和归约。第 0 步中，每个工作项都会在内部循环，将数据从全局内存流式传输到局部寄存器累加器（Acc）中。该策略可确保完全合并内存访问，并通过深度流水线化隐藏指令延迟。流式处理阶段完成后，运行时会自动进入步骤 1 和步骤 2，在局部内存中高效地执行基于树的归约。因此，最终的组间归约只涉及少量部分结果，从而大幅降低全局原子单元上的竞争。虽然这种方法性能出色，但它有一个局限：特定于硬件，并且需要针对底层设备进行复杂的精细调优。借助 SYCL，我们可以用一行代码表达这一整套高性能流水线，该代码利用了 SYCL 2020 修订版引入的一项关键增强：归约器类内置了对常见聚合函数的支持，例如求和、乘积、最小值和最大值，因此用途广泛。清单 3 展示了一个归约示例：在单次内核调用中，对数据[0..N)执行完整的结合律-交换律归约，并将结果写入标量位置 *sum；内核内采用何种归约策略（工作组树形归约、子组洗牌、最终原子合并）则由 SYCL 运行时决定。

从左到右阅读这段调用，可以看到代码向队列 q 提交了一个一维范围内核。第二个参数 sycl::reduction(sum, sycl::plus<>()) 是一个 归约描述符 ，它声明在地址 sum 处有一个标量累加器，并使用 sycl::plus<>() 进行合并。接下来是一个 Lambda 表达式，其参数 sum 刻意遮蔽了外部同名指针。在内核内部，sum 不再是指向内存的指针，而是一个由运行时提供的归约器对象。归约器会累加线程局部（或在 SYCL 中为工作项局部）的部分结果；运行时则使用目标后端提供的共享内存原语，在整个线程块（或在 SYCL 中为工作组）内合并这些部分结果，最后由少量线程块针对底层位置执行原子更新。这一

行代码在语义上等同于 Crystal 等优化系统实现的多阶段聚合流水线，并且 SYCL 运行时会针对每个目标后端自动进行调优。

GROUP BY 运算符。 最先进的 GROUP BY 操作 GPU 实现通常会先按分组键对输入排序，再执行分段归约；或者在共享内存中构建每个线程块的部分直方图，然后在全局范围内合并。这两种方案都是以增加内存流量和内核启动次数为代价，来减少原子操作的竞争。最近，基于现代 GPU 处理线程束级和块级原子操作的能力已大幅提升这一事实，Crystal [11] 提出了一种分块实现。每个线程块都会在共享内存中构建一个私有哈希表，使用原子操作（对 __共享__位置执行 atomicCAS 和 atomicAdd）将自身分块的数据聚合到该本地哈希表中，与 __syncthreads() 同步一次，然后对每个已占用的槽位执行一次 atomicAdd，将本地哈希表合并到全局结果中。该实现假设 GROUP BY 中的分组数量足够少，可以放入共享内存。如果不满足这一条件，Crystal 就会退回到直接使用 atomicAdd 执行全局原子操作。该方法的缺点是特定于硬件，因为它依赖三种不同的 CUDA 同步机制——内部使用线程束洗牌和共享内存原子操作、阶段之间使用块级屏障，以及 NVCC 编译器将 atomicAdd 降低为 PTX 指令 atomic.relaxed.gpu.add.u64，而该指令提供宽松的内存排序语义。宽松内存排序在此处既安全又有价值，原因恰恰在于带求和聚合的 GROUP BY 具有特定的代数结构。每一行都恰好贡献给一个桶。由于各个桶彼此独立，因此不存在跨桶依赖，也就不需要其他线程按顺序观察这些依赖。在一个桶内，唯一的操作是 +=，它同时满足结合律和交换律，因此来自不同行的 fetch_add 操作无论以何种顺序交错执行，结果都相同。

SYCLDB 以代码统一性和后端可移植性换取了 Crystal 的局部性优化。SYCLDB 不同于依赖组大小的两种不同实现，而是使用 sycl::atomic_ref SYCL 特性，以可移植的方式实现相同效果。atomic_ref<T, Order, Scope, Space>是一个非拥有引用，它会将通过该引用进行的读取、写入和读-改-写操作重新解释为针对底层存储的原子操作。atomic_ref 接受的一个关键模板参数是内存模型的选择。使用 memory_order::relaxed 会使 SYCL 编译器针对每种 GPU 后端生成最佳代码，即在 NVIDIA GPU 上生成与 CUDA atomicAdd 相同的 PTX 指令；在 AMD CDNA 设备上生成不带显式 s_waitcnt vmcnt(0) 的全局原子操作；在英特尔 Xe 图形处理器上生成不执行缓存行失效的 Level Zero 原子操作。

SYCLDB 使用 atomic_ref，在输入列上通过单个并行_for执行分组聚合，不进行块内归约，也不进行显式同步。为此，它假设工作负载采用星型模式，与其他 GPU 引擎类似，其中每个维度列的整数编码范围都较小且连续。因此，以下各项的笛卡尔积

```
q. parallel_for (N , [=]( sycl :: id <1> idx ) { const auto i = idx [0]; if (! fl
ags [i]) return ; int hash = 0, mult = 1; for ( int j = 0; j <G; ++j) {
hash +=( group_cols [j ][i]- min [j]) * mult ; mult *= max [j]- min [j] + 1;
} sycl :: atomic_ref <unsigned , sycl :: memory_order :: relaxed , sycl ::
memory_scope :: device , sycl :: access :: address_space :: global_space >first (
slot_flags [hash ]) ; if ( first . exchange (1) == 0) for ( int j = 0; j <G; ++
j) keys [j][hash ] =group_cols [j ][i ]; sycl :: atomic_ref <uint64_t , sycl ::
memory_order :: relaxed , sycl :: memory_scope :: device , sycl :: access ::
address_space :: global_space >sum ( agg [hash ]); sum . fetch_add ( value [i])
; }) ;
```

清单 2：SYCLDB 分组实现。

```
q. 并行_for ( range , sycl :: reduction (求和 , sycl :: plus <>() ) ,[=](sycl::id<1>
id,auto&sum){sum +=data[id]; });
```

清单 3：使用高级 SYCL 2020 归约 API 进行模块化聚合

任意少量分组列都能轻松放入扁平数组中，哈希函数也因此简化为无冲突的混合基数算术。基于前文提出的稠密键假设，每个活动行都使用混合基数算术，在分组键域上计算其在完美哈希结果数组中的槽位。清单 2 展示了分组的实现。计算出槽位后，每一行都会发出两个宽松设备作用域原子操作：一个交换操作，用于选出负责写入代表性分组键的首个写入者；另一个是 fetch_add，用于累加数值聚合。

2.4 解释型查询执行

为了在现代 GPU 加速器上动态执行 SQL 查询，**SYCLDB**采用了客户端-服务器查询规划与执行架构。基于 Java 虚拟机、使用 Apache Calcite 的优化器 [1] 会解析并优化传入的 SQL 查询，应用基于成本的关系规则来执行

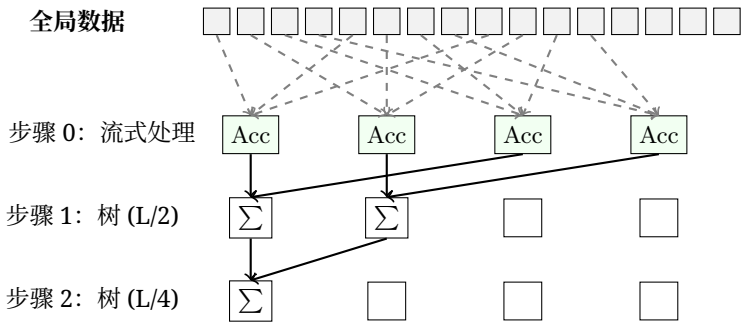


图 1：SYCL2020 归约：寄存器流式处理 + 共享内存树

连接和算子重排序。生成的优化物理执行计划会被序列化，并通过 Apache Thrift 远程过程调用传输至 C++ 执行引擎；该引擎会反序列化此计划，并构建一个有向无环图（DAG），用于表示物理算子的执行顺序。SYCLDB 支持以下物理算子：对列式输入进行扫描、选择算子、投影算子、等值连接、全局聚合和分组聚合。更通用的 SQL 功能，包括嵌套查询、用户定义函数、完整的 NULL 语义、任意字符串谓词，以及对稀疏键进行通用分组，目前不在原型的支持范围内。**SYCLDB** 通过动态按拓扑序遍历该有向无环图，并将预编译的、基于模板的通用算子直接分派到设备队列中，以执行该图。因此，**SYCLDB** 采用逐内核执行来评估任意给定的 SQL 查询。需要注意的是，虽然我们采用逐内核执行，但由于使用了选择标志，因此不会在每个内核执行后都进行完全物化。相反，物化发生在自然的流水线边界处，这与其他向量化数据库引擎类似。现代 SYCL 编译器已开始支持对 SYCL 内核进行高效的即时编译融合，因此我们将支持融合内核执行的工作留待未来在**SYCLDB** 中开展。

3 评估

硬件平台。 我们在三种异构 GPU 架构上评估 **SYCLDB** ，每种架构均通过 PCIe Gen4 x16 互连（理论双向带宽为 31.5 GB/s）连接。主机系统配备双路 **Intel Xeon Platinum 8454H** 处理器（Sapphire Rapids，每路 32 核，共 128 个线程，睿频 3.4GHz）、**1TB DDR5 ECC** 系统内存，以及一块 Gen4 NVMe 固态硬盘，其顺序读取速度可达 7,400MB/s，用于将初始数据暂存到系统内存。

– **NVIDIA L40S：**基于 Ada Lovelace 架构，配备**48 GB GDDR6** 显存，峰值带宽为 **864 GB**/s。其热设计功耗为 350W，并提供高吞吐量的 FP32/INT32 计算单元。

– **AMD Instinct MI210：**基于 CDNA 2 架构，配备**64 GB HBM2e** 显存，峰值带宽为**1.6 TB/s**。HBM2e 堆栈对于充分满足大规模分析型工作负载的高吞吐量需求至关重要。

– **Intel Data Center GPU Flex 170：**基于 Xe-HPG 架构，配备**16 GB GDDR6**显存，峰值带宽为**576 GB/s**。该平台代表了一种高密度、能效出色的加速方案，适用于中等规模的数据集。

数据集与工作负载。与其他近期 GPU 数据库引擎研究类似，我们采用 SSB 作为主要基准测试。根据每种实验设置所用的后端 GPU 调整规模因子（SF），以满足英特尔 GPU 的内存容量限制。每项实验均注明具体的 SF。为单独评估引擎的计算性能和内存层次结构性能，所有查询均在数据完整加载为列式格式并存入显存后执行，从而确保工作集驻留在设备上。

软件栈。我们使用**AdaptiveCpp 25.10.0**（使用 LLVM 18 编译）作为面向 NVIDIA、AMD 和 Intel GPU 的主要 SYCL 工具链，分别针对 CUDA 后端（CUDA 12.6 SDK）、HIP 后端（ROCm 6.3.3）和 OpenCL 后端（Intel Compute Runtime）。

测量协议与基线归一化。为消除暂态硬件噪声和冷启动编译开销，每个查询都会先以冷启动方式执行一次，但不记录结果。报告的计时数据为随后连续运行 5 次所得的平均稳态性能。由于各执行循环之间的稳态方差低于 10%，因此结果中省略误差线。除非另有说明，查询执行时间不包括从持久化存储中初始加载表的时间，并假设输入列已驻留在各引擎使用的内存空间中。对于 GPU 引擎，仅内核时间只包括设备端内核执行；总时间则包括适用情况下的 SQL 规划、主机端算子编排、内核启动开销、同步点以及设备端中间结果物化。对于采用即时编译的系统，仅内核时间不包括编译开销；只有当编译发生在测量的查询执行路径中时，总时间才会将其计入。

3.1 端到端查询执行性能 端查询执行性能 ance

为评估 SYCLDB 在大规模分析型工作负载下的表现，我们在 SSB 基准测试中对 SYCLDB 、 HeavyDB （采用编译式查询执行的工业级 GPU 数据库引擎）、SiriusDB（通过组合数据框内核构建的可组合分析引擎）、Crystal（硬编码的关系运算基本操作库）和 DuckDB（最先进的基于 CPU 的分析引擎）的执行时间进行了基准测试。本次分析使用规模因子 100（SF100），对应约 6 亿行数据。结果如图 2 所示。实心条表示仅内核时间，斜线条形则包含额外的前端开销和主机端内核编排开销。为确保所有系统之间的比较公平，

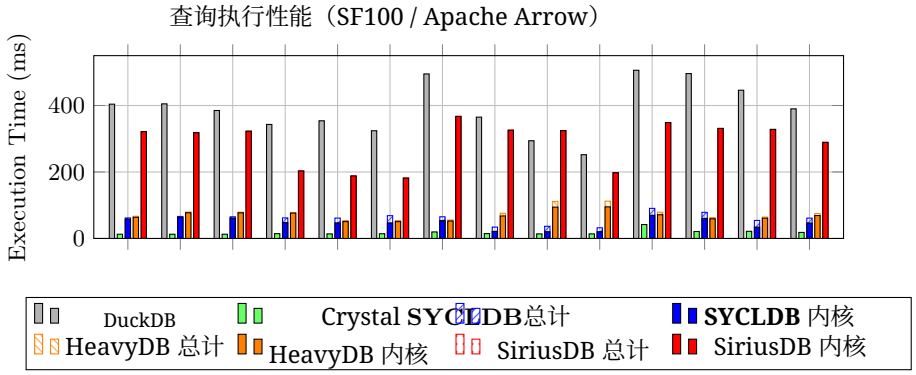


图 2：SF100 规模下的查询执行性能。实心条形表示内核执行时间，假设输入数据已驻留在设备内存中。斜线条形还包含额外的查询前端开销（SQL 解析、优化、内核启动等）。

在所有系统中，数据均从基于 Apache Arrow 的列式存储中读取，并在开始计时之前物化到各引擎所需的执行内存空间中。

DuckDB 为 CPU 查询执行提供了一个有力的参照，因为它是一款高度优化、先进的开源内存分析引擎。与 DuckDB 相比，SYCLDB 在基准测试中的执行时间显著更短，加速比介于 $4.7\times$ 和 $10.7\times$之间。这些结果展现了 GPU 在分析型查询执行方面的潜力，也凸显了进一步优化的空间；尤其值得注意的是，SYCLDB 采用直接的逐内核执行方式，未使用流水线化、内核融合或复杂索引等优化。平均而言，DuckDB 每个查询耗时 389.92 ms，而 SYCLDB 总计耗时 61.23ms（仅计内核时间则为 46.47 ms），这意味着相较于 CPU 基线，端到端执行加速比为 $6.37\times$ （仅内核执行加速比为 $8.39\times$ ）。

另一方面，Crystal 代表一种专用 GPU 实现：它将 SQL 查询直接实现为硬编码 CUDA 内核，从而省去了 SQL 解析、查询规划、前端处理、动态特化以及大部分通用算子边界开销。Crystal 并非旨在成为通用查询执行引擎，但它是专用 GPU 执行时间的绝对下界基线。正如预期，Crystal 在整个基准测试中都取得了最低执行时间；在全部基准测试中，**SYCLDB**的内核时间都比 Crystal 高出 $1.46\times$ 到 $4.93\times$。对于 Q3.2、Q3.3、Q3.4 等若干复杂连接查询，两者差距较小，**SYCLDB** 仅比手工编写的 Crystal 基线慢约 $1.46\times$–$1.47\times$ 。这些结果表明，**SYCLDB**在保持通用性的同时，对于若干查询也能提供与高度专用实现相竞争的性能。比较两者的平均值，手写且高度专用的 Crystal

基线的平均查询时间为 18.21 ms，而 SYCLDB 仅计算内核的执行时间平均为 46.47 ms。这表明，在保持完整关系型通用性的同时，SYCLDB 的性能仍可与专用的非通用 CUDA GPU 基线相媲美，差距在 $2.55\times$ 倍以内。

将 **SYCLDB** 与 SiriusDB 进行比较可以看出，SiriusDB 比 **SYCLDB** 慢 $4\times$- $16.0\times$ 。对于 Q3.2 和 Q3.3，这一差距尤为明显：SiriusDB 分别需要 325.72 毫秒和 324.53 毫秒，而 **SYCLDB** 执行相同的内核工作负载仅需 21.31 毫秒和 20.25 毫秒，加速比分别为 $15.3\times$ 和$16.0\times$。这是由于在算子边界之间物化中间状态所产生的开销。在 **SYCLDB** 中，通用执行通过设备驻留标志数组，在各算子之间传递中间条件结果，并在连续算子之间额外向全局设备内存执行写入和读取。这一开销在 SiriusDB 中更加明显，因为 SiriusDB 依赖于将高度优化的 libcudf 库函数进行链接。尽管每个独立的库原语都经过了优化，但整个流水线被分割在外部库调用之间。因此，筛选算子和连接探测产生的中间输出必须先物化到全局设备内存中，之后下一个库算子才能开始执行，从而导致整个基准测试中的执行时间显著增加。考虑内核执行时间后可以发现，**SYCLDB** 基于 SYCL 编译器和运行时工具链进行构建的方法优于 SiriusDB 采用的数据框库组合模型，因为 **SYCLDB** 的专用关系型内核使执行过程更贴近物理查询计划，并避免了链接通用库原语所产生的开销。从宏观平均值来看，SiriusDB 的平均执行时间为 289.08 毫秒，比 SYCLDB 端到端平均时间 61.23 毫秒慢 $4.72\times$ ，也比 SYCLDB 仅计算内核的平均时间 46.47 毫秒慢 $6.22\times$，凸显了数据框库链接所带来的显著显存物化开销。

HeavyDB 采用完全不同的设计策略，通过自定义 LLVM 后端生成即时编译内核。将 **SYCLDB** 与 HeavyDB 进行比较可以发现，**SYCLDB** 仍然极具竞争力。在原始内核执行时间方面，**SYCLDB** 在评估的 13 个 SSB 查询中有 11 个优于 HeavyDB，在 Q3 查询族中更是取得了显著的加速比：对于 Q3.2，**SYCLDB**执行内核工作负载耗时 21.31 ms，而 HeavyDB 耗时 67.60 ms（快 $3.2\times$）；对于 Q3.3 和 Q3.4，**SYCLDB**分别耗时 20.25 ms 和 20.21 ms，而 HeavyDB 分别耗时 94.30 ms 和 95.10 ms，两种情况下的加速比分别为 $4.7\times$。在 Q3.1 和 Q4.2 的内核时间方面，HeavyDB 略快，但差异不到 5%：Q3.1 为 52.50 ms 对 53.79 ms，Q4.2 为 59.20 ms 对 60.45 ms。这些结果展现了 SYCL 在开发基于 GPU 的分析引擎方面的潜力，也为未来通过即时编译融合 SYCL 内核、进一步优化性能打开了大门。从整体基准测试平均值来看，HeavyDB 的端到端执行平均耗时为 74.75 ms（内核时间为 68.98 ms），而 SYCLDB 平均耗时为 61.23 ms（内核时间为 46.47 ms），这表明，与 HeavyDB 自定义生成的 LLVM 查询流水线相比，SYCLDB 的端到端执行加速比为 $1.22\times$ ，内核执行加速比为 $1.48\times$ 。

最后，斜线条形与实心条形之间的差值反映了前端和内核编排的开销。在**SYCLDB**中，每个关系运算符都会作为独立的 SYCL 内核提交，随后执行显式同步点，以确保执行顺序并收集性能分析信息。这些开销包括内核调度、同步和中间状态管理。在整个基准测试中，这类开销约为 3.4 ms 至 21.6 ms，平均每个查询约为 13.2 ms。Q1.1–Q1.3 的开销最小，内核时间与总时间之间仅相差约 3–4 ms；而 Q2.3 和 Q4.1–Q4.3 等更复杂的查询则会产生更高的编排开销。尽管存在这些开销，**SYCLDB**在所有查询上的速度仍显著快于 DuckDB 和 SiriusDB，这表明通用执行路径和可组合前端对于大规模分析型工作负载而言，仍然切实可行且高效。

3.2 性能可移植性与跨厂商分 nce 可移植性与跨厂商 A 分析

本节分析**SYCLDB**在不同硬件架构上的性能可移植性，以验证：(i) 同一代码库能够在多个厂商软件栈上无缝运行；(ii) 可移植的 SYCL 实现能够与原生的、高度专用化的 GPU 基准实现竞争，甚至超越它们。我们将完整的**SYCLDB**引擎与仅用于评估的**SYCL-Hardcoded** 参考实现进行比较。
SYCL-Hardcoded 并非**SYCLDB**所提供的独立执行路径；相反，它在手写查询内核中使用相同的 SYCLDB 关系运算原语，类似于 Crystal，用以估算没有 SQL 前端时可达到的性能。

为展示跨厂商可移植性，我们在三种不同的 GPU 架构上运行完整的 SSB 基准测试：NVIDIA L40S、AMD Instinct MI210 和 Intel Data Center GPU Flex 170。实验采用规模因子 40（SF40），因为它对应英特尔 Flex 170 设备的内存上限；在 NVIDIA 和 AMD 平台上则采用规模因子 100（SF100）。我们使用 ROCm HIPIFY 工具 [2] 将 CUDA Crystal 基准实现转换为 HIP 版本，以便在 AMD 平台上进行评估。值得注意的是，手写的 CUDA 基准实现（Crystal）及其 HIP 转译版本都被厂商锁定，无法在 Intel Flex 170 上运行。这凸显了 SYCL 等开放标准在构建通用跨厂商数据库引擎方面的作用。

如图 3 所示，可移植的 SYCL-Hardcoded 参考实现始终能够达到或超越原生 CUDA 和 HIP 基线。在 NVIDIA L40S 上，SYCL-Hardcoded 的平均执行时间为 8.33 ms，相比基于原生 CUDA 的 Crystal（平均 15.88 ms）实现了$1.91\times$ 加速比。同样，在 AMD Instinct MI210 上，SYCL-Hardcoded 的平均执行时间为 14.31ms，相比原生 HIP 化的 Crystal（平均 16.17ms）实现了 $1.13\times$加速比。这一性能优势源于 Crystal 手动基于 tile 的实现存在结构性限制，其依赖显式的共享内存同步和循环结构。相比之下，SYCL 编译器（AdaptiveCpp）会动态内联

[2] https://github.com/ROCm/HIPIFY

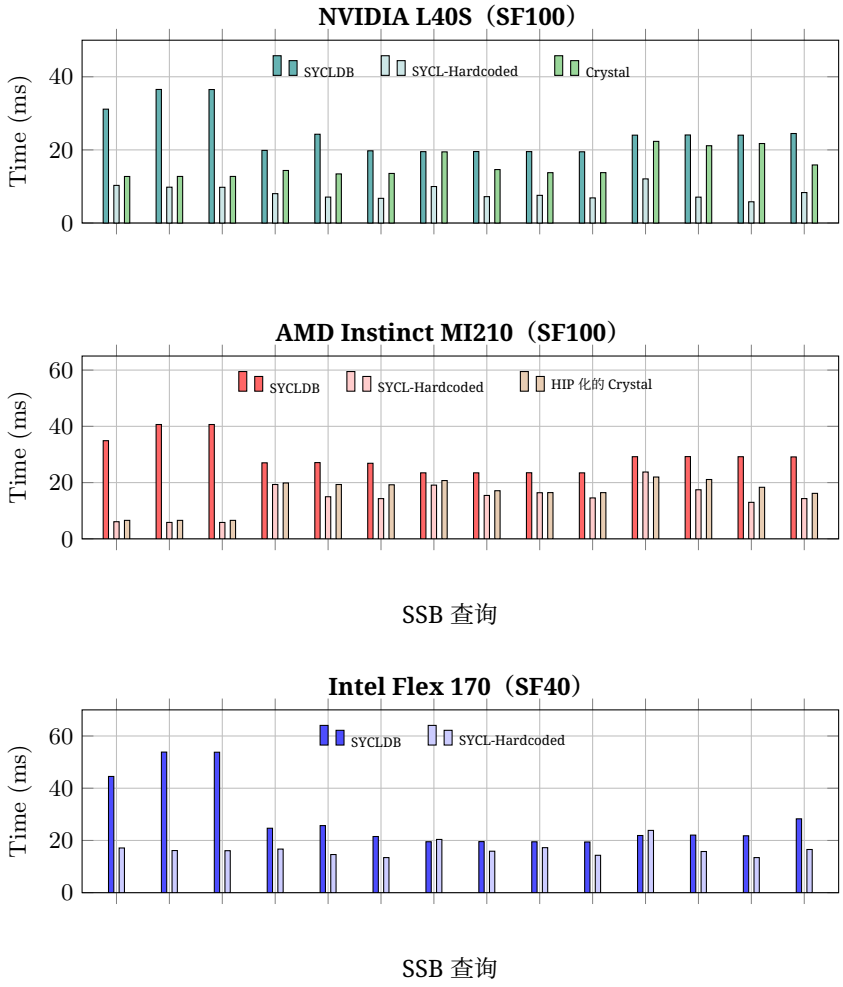


图 3：对比完整 SYCLDB 引擎与仅用于评估的 SYCL-Hardcoded 参考实现，在全部 13 个 SSB 查询上的跨平台性能可扩展性分析，以及宏观基准测试算术平均值（平均值）。

硬编码内核所使用的 SYCLDB 关系运算原语，并执行积极的全局寄存器分配和指令调度。这消除了活跃的线程屏障，并生成极其紧凑、无屏障的机器指令，使硬件执行单元达到饱和。

将 SYCLDB 与 SYCL-Hardcoded 进行比较，可以隔离通用查询执行的开销。这种开销在微体系结构层面主要由两个因素驱动：中间结果物化和指令膨胀。在完整的 SYCLDB 引擎中，关系算子被调度到多个独立内核中执行，并且引擎必须在算子边界写入并读回行选择标志向量。

在 SYCL-Hardcoded 参考实现中，扫描或探测同一张表的算子会合并到一个针对该表的内核中，而连接和表之间的转换仍各自构成独立的内核边界。每张表的选择标志采用与 Crystal 类似的局部变量实现，因此会驻留在寄存器中。对于 SF100 下的 Q1.1，完整引擎产生 28.60GB 的全局 DRAM 流量。相比之下，SYCL-Hardcoded 将选择标志保留在寄存器中，使全局 DRAM 流量降至 9.60GB，在所有架构上均统一减少 66%。此外，SYCLDB 每个查询最多需要启动 18 个不同的内核，从而导致显著的指令膨胀（AMD 上最高达$11.6\times$，NVIDIA 上最高达 $9.3\times$），并引入会使执行串行化的边界同步栅栏。对于同一张表内的算子，SYCL-Hardcoded 参考实现消除了这些算子级物化点，因此性能更佳。这些结果指明了一条自然的未来研究方向：探索利用 SYCL 内核融合，在 **SYCLDB** 中实现融合查询执行。此类方案既能达到 SYCL-Hardcoded 实现的性能，又能保留通用性。

4 结论

在这项工作中，我们介绍了 SYCLDB，这是一款完全基于 openSYCL 标准构建的端到端分析型数据库引擎。SYCLDB 使用模块化的 SYCL 原语实现核心关系运算符，并将其与基于 Apache Calcite 和 Apache Thrift、可组合的 SQL 前端集成，展示了开放标准编程模型能够在不牺牲性能的前提下，支持在 GPU 上进行完全解释执行的端到端 SQL 查询执行。

我们在星型模式基准测试上的评估表明，单一 SYCLDB 代码库可在 NVIDIA、AMD 和 Intel GPU 上无缝运行；后两者是现有基于 CUDA 的引擎无法运行的平台。尽管具备可移植性，SYCLDB 的性能仍分别比 DuckDB 快 $6.37\times$ 倍、比 SiriusDB 快 $4.72\times$ 倍，并在端到端性能上比 HeavyDB 快 $1.22\times$ 倍，同时与经过手工调优且受厂商限制的 Crystal 库相比，性能差距保持在 $2.55\times$ 以内。SYCLDB 与 SYCL-Hardcoded 之间剩余的差距，主要源于选择标志的跨内核物化，以及算子边界处的指令膨胀。这表明未来研究方向十分明确：利用新兴的 SYCL 内核融合能力，将相邻算子合并为融合流水线，在保留 SYCLDB 的 SQL 通用性的同时，逐步接近手写内核的效率。除融合外，我们还计划扩展 SYCLDB，以支持多 GPU 执行、更广泛的 SQL 覆盖范围（字符串谓词、稀疏分组、NULL 语义），以及针对异构加速器的基于成本的优化。总而言之，我们的结果表明，开放的、基于标准的 GPU 编程如今已成为构建下一代跨厂商分析型数据库系统的可靠基础。

参考文献

1. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: Apache calcite：一种基础框架，用于在异构数据源上执行优化的查询处理。载于：2018 年 ACM SIGMOD 数据管理国际会议（SIGMOD）论文集。第 221–230 页（2018）。https://doi.org/10.1145/3183713. 3190662 2. Cao, J., Sen, R., Interlandi, M., Arulraj, J., Kim, H.: GPU 数据库系统的特征分析与优化。《VLDB 基金会会刊》**17**(3), 441–454（2023 3632107 3 10 14778 3632093 ）。https://doi.org/ . / . . Crisci, L., Carpentieri, L., Thoman, P., Alpay, A., Heuveline, V., Cosenza, B.: SYCL-Bench 2020：在 AMD、Intel 和 NVIDIA GPU 上对 SYCL 2020 进行基准测试。载于：第 12 届 OpenCL 和 SYCL 国际研讨会论文集。IWOCL '24，计算机协会，纽约，美国（2024）。https://doi. org/10 3648120 4 1145 3648115 3648120, 10 1145 3648115 . / . https://doi.org/ . / . . Deakin, T., McIntosh-Smith, S.: 评估 HPC 风格 SYCL 应用程序的性能。载于：IWOCL（2020）5. Deakin, T., McIntosh-Smith, S., Alpay, A., Heuveline, V.: 基准测试并扩展 SYCL 层次并行。载于：2021 年 IEEE/ACM 百亿亿次计算层次并行国际研讨会（HiPar）。第 10–19 页（2021）。https://doi.org/10 00007 6 1109 54615 2021 2025. /HiPar . . . HEAVY.AI: HeavyDB（），https://www.heavy.ai/，存档于 https://web.archive.org/web/20250108153216/https://www.heavy.ai/ 7. Lal, S., Alpay, A., Salzmann, P., Cosenza, B., Hirsch, A., Stawinoga, N., Thoman, P., Fahringer, T., Heuveline, V.: SYCL-Bench：用于异构计算的通用跨平台基准测试套件。载于：欧洲并行处理会议（2020 220444462 8 ），https://api.semanticscholar.org/CorpusID: . Marinelli, E., Appuswamy, R.: XJoin：使用 oneAPI 在不同 XPU 架构上实现可移植的并行哈希连接。载于：DAMON（2021）9. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: 星型模式基准测试与增强型事实表索引。载于：性能评估与基准测试技术会议（TPCTC）。第 237–252 页。施普林格（2009）10. Reyes, R., Brown, G., Burns, R., Wong, M.: SYCL 2020：内涵远不止表面所见。载于：OpenCL 国际研讨会论文集（IWOCL '20）（2020）。https://doi.org/10 3388870 11 1145 3388333 . / . . Shanbhag, A., Madden, S., Yu, X.: 研究 GPU 和 CPU 在数据库分析方面的基本性能特征。载于：2020 年 ACM SIGMOD 数据管理国际会议论文集。第 1617–1632 页（2020 3380595 12 10 1145 3318464 ）。https://doi.org/ . / . . Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift：可扩展的跨语言服务实现。白皮书，Facebook（2007）13. Yogatama, B., Gong, W., Yu, X.: 将混合 CPU-GPU 数据库管理系统扩展至多个 GPU。《VLDB 基金会会刊》**17**(13), 4709–4722（2024）。https: //doi.org/10 3704977 14 14778 3704965 . / . . Yogatama, B., Yang, Y., Kristensen, K., Sarda, D., Kim, A., Cockcroft, A., Teng, Y., Patterson, J., Kimball, G., McKinney, W., Gong, W., Yu, X.: 重新思考 GPU 时代的分析处理。载于：第 14 届创新数据系统研究会议（CIDR '26）。Chaminade，美国（2026）15. Yogatama, B.W., Gong, W., Yu, X.: 在异构 CPU-GPU 数据库管理系统中协调数据放置与查询执行。《VLDB 基金会会刊》**15**(11), 2491–2503（2022）。https://doi.org/10.14778/3551793.3551809