# SYCLDB: A Cross-Vendor Heterogeneous Analytical DBMS

Ivan Donchev Kabadzhov, Eugenio Marinelli, and Raja Appuswamy

EURECOM, Biot, France

**Abstract.** The rapid growth of AI and analytical workloads has increased the need for fast, interactive analytical data processing. This, in turn, has led to a surge of interest in designing accelerated analytical database engines that can exploit modern GPUs to scale computation. While GPUs provide the massive data-parallelism required by modern analytics, the software landscape remains dominated by closed-source, vendor-locked programming models, like CUDA, limiting portability across multi-vendor GPUs. In this work, we present SYCLDB, an end-to-end analytical database engine built upon the SYCL open standard, designed to enable efficient, hardware-agnostic execution across a wide range of accelerators using a single codebase. By implementing core relational operators as modular SYCL kernels, integrating them with a SQL frontend based on Apache Calcite and Apache Thrift, and supporting Apache Arrow for efficient columnar ingestion, SYCLDB reduces dependence on vendor-specific GPU programming stacks by enabling a single SYCL codebase to execute analytical query pipelines across Intel, AMD, and NVIDIA GPUs.

**Keywords:** Analytical Databases · SYCL · Heterogeneous Computing · GPU Acceleration · Cross-Vendor Portability.

## 1 Introduction

The growing popularity of AI and analytics in virtually all application domains has led to a rapid increase in the amount of data gathered by enterprises and scientific institutions alike, with the global datasphere being projected to reach 250ZB by the year 2025 [13]. Modern analytical workloads demand fast, interactive query processing over massive scalar, vector, matrix datasets, which heavily relies on core relational operations like filtering, star-schema joins, and aggregations. Thus, the traditional workhorse of the computing industry, the general-purpose CPU, cannot be optimized to meet the diverse requirements of such heterogeneous workloads. This shortcoming of CPUs, in combination with the cessation of Dennard scaling [6], has led to a growing interest in the adoption of GPU-accelerated database engines. However, most state-of-the-art systems today are developed to target primarily NVIDIA GPUs, and hence build on the proprietary CUDA software stack that hinders their portability to GPUs and accelerators from other vendors [7, 15, 19]. Thus, there is a need for developing open, standards-based, cross-vendor approach to hardware acceleration.

Over the past few years, SYCL [14] has emerged as a high-level, open-standard programming model integrating data-parallel acceleration into C++ . Expressing data-parallel computations in SYCL provides two key advantages over other competing programming models. First, SYCL builds on a portable high-level programming language by implementing an open standard in contrast to other data-parallel languages that are proprietary (CUDA) or low level in nature (OpenCL [17]). Second, SYCL is a single-source by nature as it allows host and device code to co-exist in a single file. While recent research has investigated the performance–portability trade-offs of using SYCL for accelerating key HPC applications [4] and isolated primitives [3,5,8], its application to database engine design remains limited. Prior work (XJoin [10]) was restricted to kernel-level performance benchmarking of isolated projection and join operators. To understand the utility of an open standard for database design, two open questions must be studied: (i) it is necessary to implement a full-fledged query executor that can interpret complex query plans from application benchmarks and compare the design with state-of-the-art CUDA counterparts [7, 15, 19]; (ii) it is necessary to investigate the integration of SYCL toolchains with pluggable SQL front-ends to demonstrate fully-interpreted, end-to-end SQL query execution.

In this work, we present *SYCLDB*–a cross-architecture, performance-portable, query execution engine that can accelerate SQL query execution using not just NVIDIA GPUs, but also GPUs from other vendors (like Intel Flex, AMD Instinct, etc.) using a single code base. *SYCLDB* implements key relational operators as a sequence of highly tuned, independent kernels in SYCL. To drive SYCL kernels from SQL frontend, *SYCLDB* integrates Apache Calcite [2] and Apache Thrift [16] to build a composable query parsing/optimization front end that can transform SQL queries into a directed acyclic graph of SYCL kernels.

Using *SYCLDB*, we perform a rigorous multi-platform evaluation on NVIDIA, AMD, and Intel hardware, demonstrating that: (i) in end-to-end query execution, *SYCLDB* outperforms state-of-the-art GPU database systems (achieving up to 1.22× speedup on average over HeavyDB and up to 4.72× over SiriusDB) as well as the leading in-memory CPU database DuckDB by 4.7× to 10.7× (6.37× on average); and (ii) *SYCLDB* is fully portable across multi-vendor GPUs, enabling end-to-end SQL query execution on all evaluated platforms and demonstrating both the high performance portability of SYCL and its readiness to integrate with standard SQL front-ends. Crucially, *SYCLDB* does not represent a final design point; rather, it serves as a highly performant, open-source prototype that establishes an actionable engineering trajectory for future compilation research in cross-architecture database design. We make *SYCLDB* publicly available[1] for further research on standards-based analytics acceleration.

## 2 Design

This section details the physical implementation of *SYCLDB*, highlighting the specific features of the SYCL programming model that guide its implementation.

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

### 2.1 Memory Management

*SYCLDB* is optimized for high-throughput analytical workloads. Thus, similar to state-of-the-art GPU database engines [7,15,19], it adopts a columnar storage architecture where each table attribute is maintained as a separate, contiguous array. *SYCLDB* supports data stored in Apache Arrow-compatible columnar buffers to enable direct data ingestion from external sources. *SYCLDB* is also presently designed to work only in the scenario where all data is resident in GPU memory similar to other GPU database engines. As our goal is to be portable across multi-vendor GPUs, we do not assume the existence of high-performance interconnect like NVLink, but rather only low-bandwidth PCIe links connecting the host CPU and GPU. In such scenarios, it is well known [15] that PCIe data transfers become a bottleneck in query execution. To avoid this, similar to other GPU engines, *SYCLDB* reads all input columns during the startup phase and moves them to GPU memory. CUDA-based database engines rely on explicit functions for allocating and transferring memory between host (CPU) and device (GPU) memory, such as `cudaMalloc` and `cudaMemcpy`. Additionally, CUDA offers pinned memory, which ensures that data transfers between host and device are more efficient. In SYCL, similar functionalities are available via the Unified Shared Memory (USM) model [14] which we use to migrate data from host memory to GPU device memory.

### 2.2 *SYCLDB* Primitives

At its core, *SYCLDB* is organized as a library of relational primitives that are used as building blocks for implementing higher-level relational operators. There are four fundamental primitives, namely, selection, projection, build, and probe.

**Selection and Projection** In order to support selection and projection relational operations, *SYCLDB* supports a generic per-element selection primitive that evaluates a boolean predicate. *SYCLDB* currently targets fixed-width numeric columns, where selection predicates are expressed as binary comparisons between an input column and a constant reference value. To support multiple comparison operations, we implement selections as basic switch statements of functionals, deciding the binary operation. Listing 1a shows a basic selection primitive that operates on a single element (`col`). It compares `col` with the value `v` using the comparison type `comp` and sets the resulting boolean value.

**Build and Probe** To support the relational hash join operator, *SYCLDB* implements two core primitives: *build* and *probe*. The design targets hash-based equi-joins over fixed-width key columns, matching the join patterns commonly found in star-schema workloads like the Star Schema Benchmark (SSB) [11]. We utilize a pointer-free, array-based hash table with linear probing and simple remainder hashing. The *build* primitive populates the hash table from the build relation, while the *probe* primitive evaluates matches from the probe relation.

```cpp
template <typename T, typename Compare>
bool selection_primitive(const T& col, const T& v) {
    Compare comp{};
    return comp(col, v);
}
```

(a) Generic selection primitive acting on one element

```cpp
// Pass 1: First selection predicate
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.A[i], 100);
});
```

(b) Invoking a selection primitive from a SYCL kernel

```cpp
// Pass 1: First selection predicate
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.A[i], 100);
});

// Pass 2: Second selection predicate
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.B[i], 50);
});
```

(c) Kernel-at-a-time execution strategy

Listing 1: *SYCLDB* primitive and operator execution strategy

### 2.3 Relational Operators

*SYCLDB* uses these primitives to implement higher-level relational operators that process input relations in a massively parallel fashion across different GPU backends. In SYCL, data-parallel *kernels* are coordinated from the host and submitted asynchronously to a device *queue* via work requests. Control returns immediately to the host code once the kernel is queued for execution on the associated accelerator device. Unlike CUDA which requires kernel dimensions to be explicitly specified, SYCL allows developers to specify a 1D-range that spans a given input array. This design delegates physical thread-block partitioning and warp scheduling entirely to the underlying driver runtime, which automatically maps the work across the hardware's streaming multiprocessors. *SYCLDB* uses this functionality to simplify the implementation of relational operator kernels.

**Selection operator.** Each selection condition has to apply a binary operation between a column and a fixed reference value (that is `SELECT` $x$ `FROM` $R$ `WHERE` $x < v$). Multiple such conditions can appear in the same query. To en-

able efficient, materialization-free condition evaluation, *SYCLDB* keeps track of selection flag vectors for each relation similar to other analytical engines. These vectors encode whether each tuple remains active after the predicates evaluated so far. For conjunctive predicates, each new predicate result is combined with the previous flag value, so a tuple remains active only if it satisfies all conditions processed up to that point.

The code in Listing 1 shows the code corresponding to a selection operator with two conditions. The code contains several key variables: a queue (`q`) representing the device context, the input relation (`F`) to be processed, a selection flags array (`sf`) to store the results of the comparison operation, and the cardinality of the relation (`F.len`). Additionally, it includes comparison operator (`sycl::less`), as well as a constant value (`100`/`50` for first/second conditions) against which the column (`A`/`B`'s entries for first/second conditions) are compared. In Listing 1, the selection primitive evaluates a single predicate and returns its boolean result. Predicate composition is handled by the operator kernel. The first pass initializes the flag vector, while subsequent passes update each flag by combining the previous value with the newly evaluated predicate. The kernel's body employs a simple parallel-for loop over a 1D-range of size `F.len`. Inside the loop, the selection primitive is called with the appropriate arguments for each element at index `i`. This design keeps each predicate pass compact and allows the frontend to compose arbitrary conjunctions of supported predicates while preserving a portable kernel-at-a-time execution model. At present, *SYCLDB* supports fixed-width numeric attributes and simple comparison predicates, such as $=$, $<$, $>$, $\leq$, and $\geq$, which are executed directly inside SYCL kernels.

**Join operator.** The join operator is implemented by two separate kernel invocations, each mapping the build and probe primitives respectively. The current operator supports equi-joins over fixed-width keys, which are sufficient to express the Star Schema Benchmark (SSB) join patterns used in our evaluation. The hash table produced by the build primitive is materialized and made available to the probe kernel via GPU global memory. The probe kernel also materializes its result vector of joined values in GPU global memory for consumption by downstream operators (like aggregation).

**Aggregation operator.** Global aggregation is essential for executing SQL operators such as `SUM`, `COUNT`, and `MIN/MAX`. *SYCLDB* currently supports these aggregations over fixed-width numeric columns, where the aggregation operation is associative and commutative. State-of-the-art GPU implementations of global aggregation, like Crystal [15], often explicitly orchestrate a multi-level hierarchy: using sub-group shuffles for intra-warp communication, local memory for inter-warp synchronization, and global atomics for inter-group finalization. For instance, Figure 1 illustrates a `SUM` reduction. Each work-item in step zero loops internally to stream data from global memory into a local register accumulator (`Acc`). This strategy ensures perfectly coalesced memory accesses and hides instruction latency through deep pipelining. Once the streaming phase is com-

plete, the runtime automatically transitions to Step 1 and Step 2, performing an efficient tree-based reduction within *local memory*. As a result, the final inter-group reduction involves a minimal number of partial results, which drastically reduces contention on global atomic units.

While this approach provides high performance, it has the limitation that it is hardware specific and requires complex fine tuning corresponding to the underlying device. With SYCL, we can express this entire high-performance pipeline in a single line of code that leverages a key enhancement introduced in the SYCL 2020 revision [14]: the *reducer* class–equipped with built-in support for common aggregation functions, such as sum, product, minimum, and maximum. Listing 2 shows an example reduction that performs an entire associative-commutative reduction over `data[0..N)` into the scalar location `*sum` in a single kernel call, leaving the choice of in-kernel reduction strategy (work-group tree reduction, subgroup shuffle, final atomic merge) to the SYCL runtime.

Reading the call left to right, we can see that the code submits a single, 1-D range kernel to the queue `q`. The second argument, `sycl::reduction(sum, sycl::plus<>())`, is a *reduction descriptor* that declares that there is a scalar accumulator at the address `sum`, to be combined using `sycl::plus<>()`. This is followed by a lambda whose parameter `sum` deliberately shadows the outer pointer of the same name. Inside the kernel, `sum` is no longer a pointer to memory but a *reducer* object supplied by the runtime. The reducer accumulates thread-local (or `work-item`-local in SYCL) partials, the runtime combines partials across the threadblock (or `work-group` in SYCL) using whatever shared-memory primitive the target backend exposes, and a small number of threadblocks perform the final atomic update against the underlying location. This unified abstraction delegates the structural arrangement of the multi-stage aggregation pipeline to the underlying SYCL compiler toolchain, which automatically optimizes execution for each respective hardware backend.

**GROUP BY operator.** State-of-the-art GPU implementations of the `GROUP BY` operation would either sort the input by group key and run a segmented reduction, or build per-block partial histograms in shared memory and then merge them globally. Both alternatives trade off more memory traffic and more kernel launches for reduced contention with atomics. More recently, based on the fact that modern GPUs are much better at handling warp and block-level atomics, Crystal [15] proposed a tiled implementation. Each thread block builds a *private* hash table in shared memory, aggregates its own tile against that local table using atomics (`atomicCAS` and `atomicAdd` on `__shared__` locations), synchronizes once with `__syncthreads()`, and then merges the local table into the global result with one `atomicAdd` per occupied slot. This implementation assumes that the number of groups in a `GROUP BY` is small enough to fit in shared memory. When this is not the case, Crystal falls back to direct global atomics with `atomicAdd`. The downside of this approach is that it is hardware specific, as it relies on three different CUDA synchronization mechanisms–warp shuffles and shared-memory atomics internally, a block-wide barrier between

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

Listing 2: Modular aggregation using the high-level SYCL 2020 reduction API

phases, and the fact that NVCC compiler lowers the `atomicAdd` it to a PTX `atom.relaxed.gpu.add.u64` instruction, which provides relaxed memory ordering semantics. Relaxed memory ordering is safe and valuable here precisely because of the algebraic structure of `GROUP BY` with a summing aggregate. Each row contributes to exactly one bucket. As buckets are disjoint, there is no cross-bucket dependence that another thread might need to observe in order. Within a bucket, the only operation is `+=`, which is both associative and commutative, so any interleaving of `fetch_add`s from different rows produces the same result.

*SYCLDB* trades Crystal's locality optimization for code uniformity and backend portability. Instead of two different implementations that depend on the group size, *SYCLDB* in contrast, uses the `sycl::atomic_ref` SYCL feature to achieve the same effect in a portable fashion. An `atomic_ref<T, Order, Scope, Space>` is a *non-owning* reference that re-interprets reads, writes, and read-modify-writes through it as atomic operations on the underlying storage. A key template parameter taken by `atomic_ref` is the choice of memory model. The use of `memory_order::relaxed` results in the SYCL compiler producing the optimal code for each GPU backend, namely the same PTX instruction as CUDA `atomicAdd` on NVIDIA GPUs, a global atomic without an explicit `s_waitcnt vmcnt(0)` on AMD CDNA devices, and a Level-Zero atomic without cache-line invalidation on Intel Xe GPUs. *SYCLDB* uses `atomic_ref` to perform grouped aggregation in a single `parallel_for` over the input column with no intra-block reduction and no explicit synchronization. In order to do this, *SYCLDB* assumes a star-schema workload configuration identical to Crystal [15], where each dimension column exhibits a small, contiguous range of integer codes. Hence, the Cartesian product of the group-by columns fits in a flat array, allowing each active row to map directly to its slot via collision-free mixed-radix arithmetic. Listing 3 shows this GROUP BY implementation. After computing the slot, each row issues two relaxed device-scope atomics: an `exchange` that elects the first writer responsible for installing the representative group keys, and a `fetch_add` that accumulates the numeric aggregate.

### 2.4 Interpreted Query Execution

To execute SQL queries dynamically on modern GPU accelerators, *SYCLDB* employs a decoupled server-client query planning and execution architecture. A Java Virtual Machine based optimizer using Apache Calcite [2] parses and optimizes the incoming SQL query, applying cost-based relational rules to perform join and operator reordering. The resulting optimized physical execution plan is serialized and transmitted via Apache Thrift RPC to the C++ execution engine, which deserializes this plan and constructs a directed acyclic graph

```
q.parallel_for(N, [=](sycl::id<1> idx) {
    const auto i = idx[0];
    if (!flags[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(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]);
});
```

Listing 3: *SYCLDB* GROUP BY implementation.

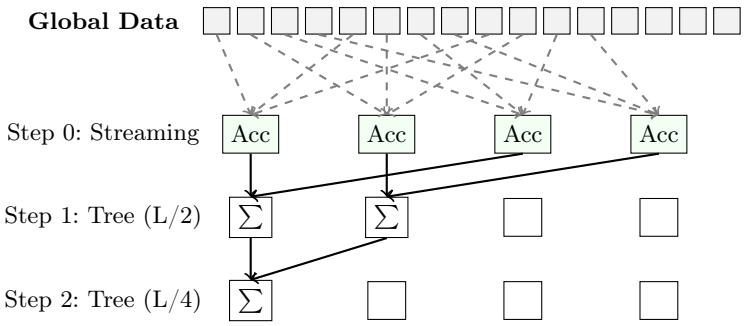


Fig. 1: SYCL2020 Reduction: Register Streaming + Shared-Memory Tree

(DAG) representing the physical operator execution order. *SYCLDB* executes the DAG by dynamically traversing it in topological order and dispatching pre-compiled, template-based general-purpose operators directly to the device queue. Thus, *SYCLDB* performs kernel-at-a-time execution to evaluate any given SQL query. It is important to note that while we do kernel-at-a-time execution, we do not perform full materialization after each kernel due to use of selection flags. Instead, materialization happens at natural pipeline boundaries (such as join probes). Currently, the prototype targets a high-performance subset of core relational operators optimized for fixed-width numeric attributes; we detail our exact feature boundaries and schema limitations in Section 4.

## 3 Evaluation

**Hardware Platforms.** We evaluate *SYCLDB* across three heterogeneous GPU architectures, each connected via a PCIe Gen4 x16 interconnect (31.5 GB/s theoretical bi-directional bandwidth). The host system is equipped with a dual-socket **Intel Xeon Platinum 8454H** processors (Sapphire Rapids, 32 cores/socket, 128 threads total, 3.4GHz boost), **1 TB DDR5 ECC** system memory, and a Gen4 NVMe SSD capable of 7,400 MB/s sequential reads, used for initial data staging into system memory.

- **NVIDIA L40S:** Based on the Ada Lovelace architecture, featuring **48 GB of GDDR6** memory with a peak bandwidth of **864 GB/s**. It operates at a 350W TDP and provides high-throughput FP32/INT32 compute units.
- **AMD Instinct MI210:** Built on the CDNA 2 architecture, providing **64 GB of HBM2e** memory with a peak bandwidth of **1.6 TB/s**. The HBM2e stack is critical for saturating the high-throughput requirements of analytical workloads at scale.
- **Intel Data Center GPU Flex 170:** Based on the Xe-HPG architecture, featuring **16 GB of GDDR6** memory with a peak bandwidth of **576 GB/s**. This platform represents a high-density, power-efficient acceleration point for moderate-scale datasets.

*Dataset and Workload.* We utilize the industry-standard Star Schema Benchmark (SSB) as our primary evaluation workload, because its data warehousing query patterns isolate the core microarchitectural bottlenecks of GPU-accelerated analytical execution: high-selectivity predicate pushdowns, multi-way pointer-chasing hash joins, and dense global aggregations.

**Software Stack.** We use **AdaptiveCpp 25.10.0** (compiled with LLVM 18) as our primary SYCL toolchain for NVIDIA, AMD, and Intel GPUs, targeting the CUDA backend (CUDA 12.6 SDK), HIP backend (ROCm 6.3.3), and OpenCL backend (Intel Compute Runtime), respectively [1, 9].

**Measurement Protocol and Baseline Normalization.** To eliminate transient hardware noise and cold-start compilation overhead, each query is first executed as an unrecorded cold pass. Reported timings correspond to the steady-state performance averaged over 10 subsequent consecutive runs. Because the steady-state variance across execution loops was under 10%, error bars are omitted from results. Unless otherwise stated, query execution time excludes initial table loading from persistent storage and assumes that input columns are already resident in the memory space used by each engine. Once the data is entirely device-resident, there is no structural layout or storage-layer difference between the evaluated engines, successfully isolating pure compute and memory hierarchy performance. For GPU engines, kernel-only time includes only device kernel execution, while total time includes SQL planning where applicable, host-side operator orchestration, kernel launch overheads, synchronization points, and device-side intermediate materialization. For systems that perform JIT compilation, compilation overhead is excluded from kernel-only time and included in total time only when it occurs as part of the measured query execution path.

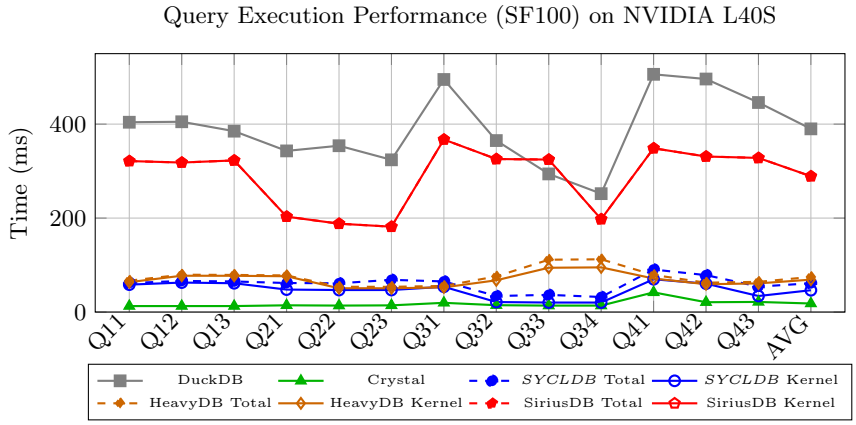


Fig. 2: Query execution performance at SF100 scale on NVIDIA L40S. The solid lines report kernel execution time, assuming that the input data is already resident in device memory. The dashed lines include the additional query frontend overhead (SQL parsing, optimization, kernel launches, etc.).

### 3.1 End-to-End Query Execution Performance

To evaluate *SYCLDB* under a large-scale analytical workload, we benchmarked the execution time of *SYCLDB*, HeavyDB (industry-grade GPU database engine with compiled query execution) [7], SiriusDB (composable analytics engine based on composing dataframe kernels) [19], Crystal [15], and DuckDB (state-of-the-art CPU-based analytical engine) [12] under the SSB benchmark [11]. For this analysis, we use Scale Factor 100 (SF100), corresponding to approximately 600 million rows. The results are shown in Figure 2.

DuckDB provides a strong reference point for CPU query execution as it is a highly-optimized, state-of-the-art, open-source in-memory analytical engine. Compared with DuckDB, *SYCLDB* achieves substantially lower execution times across the benchmark with a speedup between 4.7× and 10.7×. These results show the potential of GPUs for analytical query execution and underscore potential for further improvements, especially given the fact that *SYCLDB* performs straight-forward kernel-at-a-time execution without optimizations like pipelining, kernel fusion, or indexing. On average, DuckDB requires 389.92 ms per query, whereas *SYCLDB* Total requires 61.23 ms (and 46.47 ms for Kernel time only), which represents a 6.37× speedup in overall end-to-end execution (8.39× speedup in kernel-only execution) over the CPU baseline.

Crystal, on the other hand, represents a specialized GPU implementation in which SQL queries are directly implemented as hard-coded CUDA kernels, thus, removing SQL parsing, query planning, frontend processing, dynamic specialization, and most general-purpose operator-boundary costs. While Crystal is not built to be a general-purpose query execution engine, it acts as an absolute lower-bound baseline for specialized GPU execution time. As expected, Crystal

achieves the lowest execution times across the benchmark, with *SYCLDB*'s kernel time between 1.46× and 4.93× higher than Crystal across the entire benchmark. For several complex join queries such as Q3.2, Q3.3, Q3.4, the gap is small with *SYCLDB* only about 1.46×–1.47× slower than the manual Crystal baseline. These results show that the *SYCLDB* preserves generality while providing performance competitive with a highly specialized implementation for several queries. Comparing their averages, the hand-written Crystal baseline achieves a mean query time of 15.88 ms, while *SYCLDB*'s interpreted kernel-only execution averages 46.47 ms. This demonstrates that *SYCLDB* remains competitive within a 2.93× factor of a specialized CUDA reference. Crucially, as isolated in our performance portability analysis (Sect. 2), this execution gap is not a limitation of the SYCL abstraction layer, but rather represents the necessary query orchestration and relational modularity tax of a fully interpreted database engine running over independent operator boundaries.

Comparing *SYCLDB* to SiriusDB, we see that SiriusDB is 4× - 16.0× slower than *SYCLDB*. The gap is especially visible for Q3.2 and Q3.3, where SiriusDB requires 325.72 ms and 324.53 ms, while *SYCLDB* executes the same kernel workloads in 21.31 ms and 20.25 ms, corresponding to speedups of 15.3× and 16.0×, respectively. This is due to the cost of materializing intermediate states across operator boundaries. In *SYCLDB*, general-purpose execution propagates intermediate condition outcomes across operators through device-resident flag arrays, with additional write and read from global device memory between successive operators. This cost is even more pronounced in SiriusDB, which relies on chaining highly optimized `libcudf` library functions. Although each individual primitive is optimized, the pipeline is fragmented across external library calls. Consequently, intermediate outputs produced by filters and join probes must be materialized in global device memory before the next library operator can begin, leading to substantially higher execution times across the benchmark. Considering kernel execution time, the comparison shows that *SYCLDB*'s approach of building on SYCL compiler and runtime toolchain is more efficient than a dataframe-library composition model adopted by SiriusDB, as *SYCLDB*'s specialized relational kernels keep execution closer to the physical query plan and avoid the overhead incurred by chaining general-purpose library primitives. In terms of macro-level averages, SiriusDB exhibits a mean execution time of 289.08 ms, which is 4.72× slower than *SYCLDB*'s end-to-end average of 61.23 ms and 6.22× slower than *SYCLDB*'s kernel-only average of 46.47 ms, highlighting the significant VRAM materialization tax of dataframe library chaining.

HeavyDB follows a completely different design strategy by using a custom LLVM [9] backend to generate JIT-compiled fused kernels. Comparing *SYCLDB* and HeavyDB, we see that *SYCLDB* remains highly competitive. In raw kernel execution time, *SYCLDB* outperforms HeavyDB on 11 out of the 13 evaluated SSB queries with a significant speedup for the Q3 query family: for Q3.2, *SYCLDB* executes the kernel workload in 21.31 ms compared to 67.60 ms for HeavyDB (3.2× faster), while for Q3.3 and Q3.4 it requires 20.25 ms and 20.21 ms compared to 94.30 ms and 95.10 ms for HeavyDB, corresponding to speedups of

4.7× in both cases. HeavyDB is slightly faster in kernel time for Q3.1 and Q4.2, but the difference is less than 5%: 52.50 ms versus 53.79 ms for Q3.1, and 59.20 ms versus 60.45 ms for Q4.2. These results show the potential of SYCL for developing GPU-based analytical engines and open the door for further performance optimizations that can be achieved by using JIT compilation to fuse SYCL kernels in the future. Looking at the overall benchmark averages, HeavyDB requires an average of 74.75 ms for end-to-end execution (and 68.98 ms for kernel time), whereas *SYCLDB* requires 61.23 ms (and 46.47 ms for kernel time), showing that *SYCLDB* achieves a 1.22× speedup in end-to-end execution and a 1.48× speedup in kernel execution over HeavyDB's custom-generated LLVM query pipeline.

Finally, the gap between the dashed lines and the solid lines quantifies the overhead of the frontend and kernel orchestration. In *SYCLDB*, each relational operator is submitted as a separate SYCL kernel and followed by an explicit synchronization point used to enforce execution order and collect profiling information. This overhead includes kernel dispatch, synchronization, and intermediate state management. Across the benchmark, this overhead ranges from approximately 3.4 ms to 21.6 ms, with an average of about 13.2 ms per query. The smallest overheads appear in Q1.1–Q1.3, where the gap between kernel time and total time is only about 3–4 ms, while more complex queries such as Q2.3 and Q4.1–Q4.3 incur larger orchestration costs. Even with this overhead, *SYCLDB* remains substantially faster than DuckDB and SiriusDB across all queries, indicating that the general-purpose execution path and the composable front end both remain practical and efficient for large-scale analytical workloads.

### 3.2 Performance Portability and Cross-Vendor Analysis

In this section, we analyze the performance portability of *SYCLDB* across diverse hardware architectures, validating that (i) the same codebase executes seamlessly across multiple vendor stacks, and (ii) portable SYCL implementations can compete with or outperform native, highly specialized GPU baselines. We compare the full *SYCLDB* engine against an evaluation-only SYCL-Hardcoded reference implementation. SYCL-Hardcoded is not a separate execution path exposed by *SYCLDB*; rather, it uses the same *SYCLDB* relational primitives inside hand-written query kernels similar to Crystal to estimate the performance available without a SQL front end.

To demonstrate cross-vendor portability, we execute the full SSB benchmark on three distinct GPU architectures: the NVIDIA L40S, the AMD Instinct MI210, and the Intel Data Center GPU Flex 170. We select Scale Factor 40 (SF40) for the Intel Flex 170 to match the device's 16 GB physical memory capacity boundary, while evaluating the high-throughput NVIDIA and AMD platforms at SF100. We used the ROCm HIPIFY tool [2] to convert the CUDA Crystal baseline into its HIP version for AMD evaluation. Notably, the hand-written CUDA baseline (Crystal) and its HIP translation are entirely vendor-

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

locked and cannot run on the Intel Flex 170. This highlights the role of open standards like SYCL in building general-purpose, cross-vendor database engines.

As shown in Figure 3, the portable SYCL-Hardcoded reference consistently matches or outperforms the native CUDA and HIP baselines. On NVIDIA L40S, SYCL-Hardcoded achieves a mean execution time of 8.33 ms, representing a 1.91× speedup over native CUDA-based Crystal (15.88 ms average). Similarly, on AMD Instinct MI210, SYCL-Hardcoded averages 14.31 ms, securing a 1.13× speedup over native Hipified Crystal (16.17 ms average). This performance advantage is driven by the structural limitations of Crystal's manual tile-based implementation, which relies on explicit shared-memory synchronization and loop structures. In contrast, the SYCL compiler (AdaptiveCpp) dynamically inlines the *SYCLDB* relational primitives used by the hardcoded kernels, performing aggressive global register allocation and instruction scheduling. This eliminates active thread barriers and produces extremely tight, barrier-free machine instructions that saturate the hardware's execution units.

Comparing *SYCLDB* with SYCL-Hardcoded isolates the cost of general-purpose query execution. This overhead is microarchitecturally driven by two main factors: intermediate materialization and instruction inflation. In the full *SYCLDB* engine, relational operators are scheduled across independent kernels and the engine must write and read back row selection flag vectors across operator boundaries. In the SYCL-Hardcoded reference, operators that scan or probe the same table are combined into a per-table kernel, while joins and table-to-table transitions remain separate kernel boundaries. Per-table selection flags are implemented as local variables similar to Crystal, resulting in them being resident in registers. For Q1.1 at SF100, the full engine generates 28.60 GB of global DRAM traffic. In contrast, SYCL-Hardcoded keeps selection flags resident in registers, reducing global DRAM traffic to 9.60 GB leading to a uniform 66% reduction across all architectures. Furthermore, *SYCLDB* requires launching up to 18 distinct kernels per query, introducing substantial instruction inflation (up to 11.6× on AMD and 9.3× on NVIDIA) and boundary synchronization fences that serialize execution. The SYCL-Hardcoded reference removes these per-operator materialization points where they occur within the same table leading to better performance. Similar to the monolithic compilation strategy by HeavyDB [7], these microarchitectural insights point to a natural direction for future work: runtime compilation to dynamically fuse adjacent multi-operator blocks into a single device kernel. This approach would allow *SYCLDB* to provide the performance of the hardcoded reference while preserving declarative relational generality.

## 4 Discussion and Future Work

In this work, we presented *SYCLDB*, an end-to-end analytical database engine built entirely on the open SYCL standard. By implementing core relational operators using modular SYCL primitives, and by integrating them with a composable SQL frontend based on Apache Calcite and Apache Thrift, SYCLDB demonstrates that an open-standard programming model can support fully in-

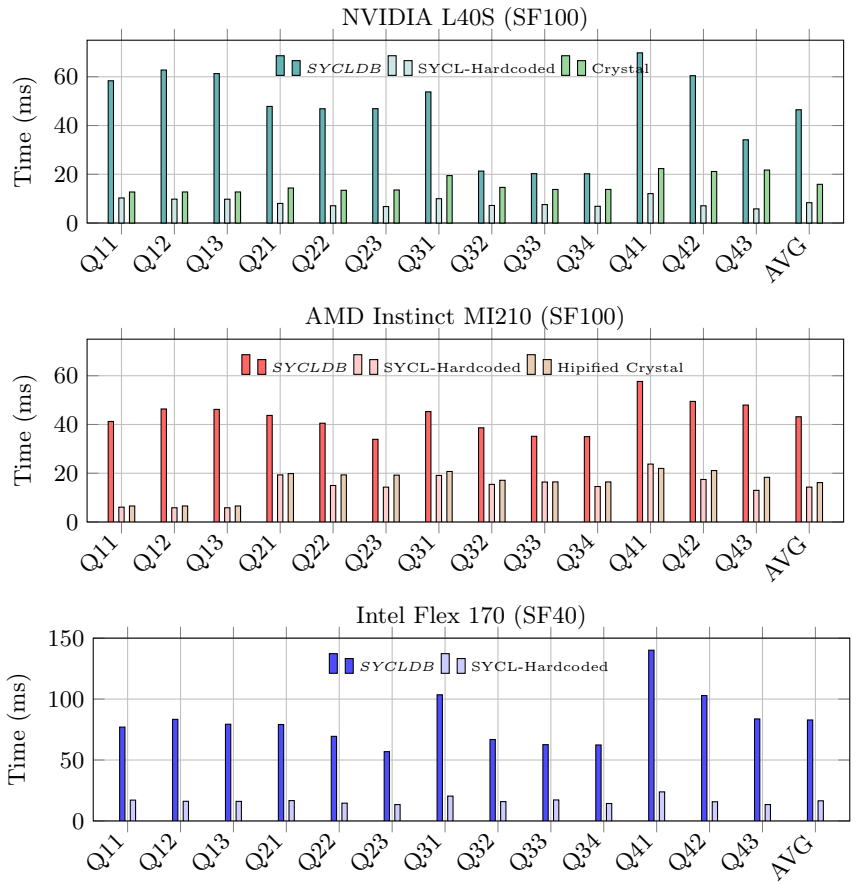


Fig. 3: Cross-architecture execution performance comparison across SSB queries.

terpreted, end-to-end SQL query execution on the GPU. Our evaluation across NVIDIA, AMD, and Intel hardware demonstrates seamless portability and competitive execution speeds compared to proprietary, vendor-locked reference engines [7, 15, 19]. However, the current prototype exhibits operational boundaries that outline key avenues for future development:

*Workload Diversity and SQL Coverage.* Broader schema configurations and query patterns, such as those in TPC-DS [18], are currently outside the scope of our execution library. At present, *SYCLDB* excludes complex relational features like comprehensive `NULL` semantics, non-equi joins, variable-length strings, sparse-key grouping (requiring dynamic hash-collision resolution), and User-Defined Functions (UDFs). Integrating these capabilities and data types remains a primary path for future work. Furthermore, we plan to extend beyond standard relational boundaries to natively support convergent AI workloads.

*Modularity and Optimization Paradigms.* The observed 2.93× performance gap relative to Crystal's specialized CUDA references represents the relational modularity and orchestration tax of an interpreted query engine, rather than an inherent limitation of the SYCL language abstraction. Future work can eliminate this overhead by taking advantage of AdaptiveCpp's recently introduced just-in-time (JIT) compiler support for kernel fusion[3].

*Lakehouse Integration and Storage Heterogeneity.* Modern analytics are rapidly shifting toward decentralized data lakes and cloud lakehouse architectures [20]. Future iterations of *SYCLDB* will investigate native data streaming and zero-copy ingestion from open table formats (such as Apache Iceberg or Delta Lake) directly into device-accessible memory spaces. Leveraging SYCL's Unified Shared Memory (USM) properties alongside networked file systems will allow our portable execution layer to process massive, remote data lakes directly, making *SYCLDB* a highly generic acceleration plug-in for cross-platform data meshes.

*Heterogeneous Multi-GPU Execution.* While this work proves performance portability by evaluating separate hardware platforms independently, modern large-scale clusters increasingly deploy diverse accelerators in tandem pipelining a single query plan across concurrent GPUs from different vendors.

Taken together, our results demonstrate that open, standards-based GPU programming is now a mature foundation for the next generation of cross-vendor analytical database systems.

## References


1. Alpay, A., Heuveline, V.: Sycl beyond opencl: The architecture, current state and future direction of hipsycl. In: Proceedings of the International Workshop on OpenCL (2020). https://doi.org/10.1145/3388333.3388658, https://doi.org/10.1145/3388333.3388658
2. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources. In: Proceedings of the 2018 International Conference on Management of Data (SIGMOD) (2018). https://doi.org/10.1145/3183713.3190662
3. Crisci, L., Carpentieri, L., Thoman, P., Alpay, A., Heuveline, V., Cosenza, B.: Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus. In: Proceedings of the 12th International Workshop on OpenCL and SYCL (2024). https://doi.org/10.1145/3648115.3648120
4. Deakin, T., McIntosh-Smith, S.: Evaluating the performance of hpc-style sycl applications. In: Proceedings of the International Workshop on OpenCL (2020). https://doi.org/10.1145/3388333.3388643
5. Deakin, T., McIntosh-Smith, S., Alpay, A., Heuveline, V.: Benchmarking and extending sycl hierarchical parallelism. In: 2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar). pp. 10–19 (2021). https://doi.org/10.1109/HiPar54615.2021.00007


[3] https://github.com/AdaptiveCpp/AdaptiveCpp/releases/tag/v24.06.0

6. Esmaeilzadeh, H., Cao, T., Yang, X., Blackburn, S.M., Burger, D.: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA). pp. 365–376 (2011). https://doi.org/10.1145/2000064.2000108
7. HEAVY.AI: Heavydb (2025), https://www.heavy.ai/
8. Lal, S., Alpay, A., Salzmann, P., Cosenza, B., Hirsch, A., Stawinoga, N., Thoman, P., Fahringer, T., Heuveline, V.: Sycl-bench: A versatile cross-platform benchmark suite for heterogeneous computing. In: European Conference on Parallel Processing (2020), https://api.semanticscholar.org/CorpusID:220444462
9. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. p. 75. CGO '04, IEEE Computer Society, USA (2004)
10. Marinelli, E., Appuswamy, R.: Xjoin: Portable, parallel hash join across diverse xpu architectures with oneapi. In: Proceedings of the 17th International Workshop on Data Management on New Hardware (2021). https://doi.org/10.1145/3465998.3466012
11. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: Technology Conference on Performance Evaluation and Benchmarking (TPCTC). pp. 237–252 (2009)
12. Raasveldt, M., Mühleisen, H.: Duckdb: an embeddable analytical database. In: Proceedings of the 2019 International Conference on Management of Data (SIGMOD). pp. 1981–1984 (2019). https://doi.org/10.1145/3299869.3320212
13. Reinsel, D., Gantz, J., Rydning, J.: The digitization of the world: From edge to core. In: IDC White Paper (2018), framingham, MA: International Data Corporation. Document number: US44413318. Forecasts the Global Datasphere to grow to 175 ZB by 2025.
14. Reyes, R., Brown, G., Burns, R., Wong, M.: Sycl 2020: More than meets the eye. In: Proceedings of the International Workshop on OpenCL (IWOCL '20) (2020). https://doi.org/10.1145/3388333.3388870
15. Shanbhag, A., Madden, S., Yu, X.: A study of the fundamental performance characteristics of gpus and cpus for database analytics. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 1617–1632 (2020). https://doi.org/10.1145/3318464.3380595
16. Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable Cross-Language Services Implementation. White paper, Facebook (2007)
17. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science and Engg. **12**(3), 66–73 (May 2010)
18. Transaction Processing Performance Council (TPC): TPC Benchmark DS (TPC-DS) Specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v3.2.0.pdf (2021), accessed: 2026-06-23
19. Yogatama, B., Yang, Y., Kristensen, K., Sarda, D., Kim, A., Cockcroft, A., Teng, Y., Patterson, J., Kimball, G., McKinney, W., Gong, W., Yu, X.: Rethinking analytical processing in the gpu era. In: 14th Conference on Innovative Data Systems Research (CIDR '26) (2026)
20. Zaharia, M.A., Ghodsi, A., Xin, R., Armbrust, M.: Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In: Conference on Innovative Data Systems Research (2021), https://api.semanticscholar.org/CorpusID:229576171