Scalable GPU Acceleration of Scalar Functions in Analytical Databases: Compilation, Benchmarking, and Optimization

Kaushik Rajan Sampath Rajendra Momin Al-Ghosien Nicolas Bruno Carlo Curino
Matteo Interlandi Yinan Li Lukas M. Maas Craig Peeper Surajit Chaudhuri Johannes Gehrke

Microsoft

ABSTRACT

Accelerating SQL query execution with GPUs is a central focus in database research. While prior systems have achieved notable speedups by offloading relational operators, the acceleration of the wide range of *scalar functions* that are supported by analytical engines remains unaddressed. Our analysis reveals that many *scalar functions* incur substantial computational overhead and often constitute the primary bottleneck in analytical queries on CPUs. This observation motivates a systematic exploration of the opportunities and challenges in accelerating *scalar functions* on GPUs.

Unlike relational operators, which are few in number and standardized, production databases support hundreds of *scalar functions*. The absence of a standardized specification, combined with this diversity, renders manual GPU porting infeasible. To address this, we present an LLVM-MLIR-based compiler toolchain that automatically translates the CPU-based implementations of scalar functions from production databases into efficient GPU kernels, while preserving their original semantics. Our approach lifts *scalar functions* to a high-level intermediate representation, applies resource-optimizing transformations, and generates GPU assembly code, supporting all relevant data types, parameters, and database context variables.

As existing benchmarks do not sufficiently stress test *scalar functions* in analytical queries, we introduce a variant of TPC-H that utilizes *scalar functions* while preserving the original query intent. Integrating our GPU kernels into a state-of-the-art GPU database system, we demonstrate substantial performance gains over a leading CPU database that uses slightly more expensive hardware: 7.6× on enhanced TPC-H and 6.4× on production queries, further widening the gap between GPU and CPU databases. The generated kernels deliver performance comparable to hand-optimized GPU implementations, establishing our approach as a scalable and practical solution for accelerating *scalar functions* on GPUs.

PVLDB Reference Format:
Kaushik Rajan, Sampath Rajendra, Momin Al-Ghosien, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Yinan Li, Lukas M. Maas, Craig Peeper, Surajit Chaudhuri, Johannes Gehrke. Scalable GPU Acceleration of Scalar Functions in Analytical Databases: Compilation, Benchmarking, and Optimization. PVLDB, 19(7): 1441 - 1454, 2026.
doi:10.14778/3801059.3801061

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801061

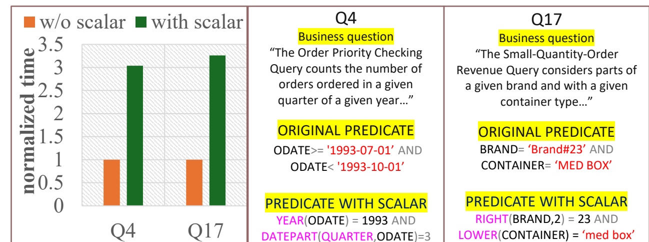

Figure 1: Adding realistic scalar function usage causes a 3× slowdown for TPC-H; figure shows Q4 and Q17 as examples.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.6084/m9.figshare.29452214.

1 INTRODUCTION

The last decade has seen a rapid evolution of the hardware ecosystem, including the emergence of GPUs and other accelerators. The database community has recognized the potential of GPUs for accelerating query processing, and a number of GPU databases have been developed [4, 7, 13, 24, 34, 47, 54, 57]. The primary focus of the research on GPU databases has been to accelerate the execution of relational operators. [29, 33–35, 42, 52, 56, 59, 60]. However, the surface syntax of the SQL query language goes beyond just relational operators and includes a large number of *scalar functions* [8, 11, 12, 14] (sometimes also referred to as *intrinsic functions*, *built-in functions* or plainly as *SQL functions*). These functions, as defined by the SQL specification, perform element-wise transformations, taking a single value as input and producing an output value. *Scalar functions* are extensively used in SQL queries for tasks such as string manipulation, date and time processing, type conversions, and mathematical computations, making them essential for a wide range of analytical workloads.

Accelerating Scalar Functions: Opportunities and Challenges. Most commercial database systems support several hundred *scalar functions* [3, 8, 11, 12, 14] and our evaluation of a large set of SQL queries from a production workload on Microsoft SQL Server shows that these functions are invoked frequently, and often in performance-critical parts of the query execution. Telemetry gathered over a million production queries shows that, in this workload, 58% of queries use at-least one *scalar function* with about 18% of queries using 3 or more of them. These functions often account for 50% or more of total query execution time on CPUs. Consider representative variants of TPC-H queries *Q4* and *Q17* which have been modified to use *scalar functions* while preserving the query specification. As can be seen in Figure 1, the introduction of intrinsics leads to a 3× increase in execution time.

The poor performance of *scalar functions* can be largely attributed to the inherent complexity in the nature of the performed computation. Most functions (with the exception of mathematical operations on basic types) have a high arithmetic intensity (number of operation per byte read), often requiring computations that sequentially traverse the inputs. For example: (a) traversing UTF-8 strings in a collation specific manner, requires sequential processing to identify character boundaries; (b) operations on *date* types, like computing the number of years between two dates, requires sequential loops to detect and account for leap years. These substantial computational overheads on CPUs underscore the need to explore GPU acceleration for *scalar functions*.

From a GPU database perspective, the absence of native support for *scalar functions* fundamentally disrupts efficient query execution. When a query encounters a scalar function that cannot be evaluated on the GPU, execution must either fail or fall back to the CPU, forcing data transfers between GPU and CPU over PCIe. These transfers are substantially slower than the memory bandwidth of CPUs or GPUs, introducing significant overhead and latency. As a result, queries either fail outright or suffer severe performance degradation, with the cost of moving data back and forth often outweighing any computational speedup gained from GPU acceleration [26, 46, 57, 62].

Supporting *scalar functions* on GPUs poses unique challenges, distinct from those of relational operators. First, relational operators are few and well-studied, but *scalar functions* are in the hundreds, each with different semantics and computational patterns [3, 8, 11, 12, 14]. This diversity makes it impractical to design a small set of hardware-specific data structures or rely on generic GPU abstractions like tensor operations [44]. Second, *scalar functions* lack a fully standardized specification. Their precise semantics—including edge case handling, supported types, optional parameters, and context (e.g., string collations or calendar systems [1, 28])—vary across database systems. As a result, the implementation in each system often serves as the de facto specification. The implementation of *scalar functions* in SQL Server is spread over 100,000 lines of code. Re-implementing every *scalar function* for GPUs is infeasible due to code duplication, maintenance overhead, and risk of inconsistencies. A new approach is needed to efficiently and consistently support the full range of *scalar functions* on GPUs. An ideal solution would be to reuse implementations from mature databases, ensuring compatibility and reducing development effort.

Proposed Solution. To address this challenge, we develop an extensible compiler toolchain that automatically translates existing CPU based implementations of *scalar functions* from a production database (SQL Server) into efficient GPU kernels. We leverage the LLVM-MLIR compiler framework to enable GPU code generation.

The Multi-Level Intermediate Representation (MLIR) framework was recently introduced as part of LLVM to facilitate the development of domain-specific compilers targeting CPUs, GPUs and other accelerators. MLIR has gained widespread adoption in the AI and machine learning communities [9, 15–17, 22, 43, 53] to generate code for GPUs from NVIDIA [10], AMD [2] and INTEL [18], as well as accelerators like TPUs [15]. It has also been previously used to build a compilation-based CPU database engine [40]. Most MLIR-based compilers focus on domain-specific languages, supporting only a narrow set of abstractions and code-generation passes tailored to their domains. However, they do not address the broader class of element-wise computations, where *scalar functions* used in analytical databases represent a critical and widely-used set of functions. *Scalar functions* in production databases are implemented in high-level languages and rely on a combination of complex language features such as templates, polymorphism and function pointers, and use of general purpose constructs such as loops, conditionals, structs, and classes. Existing MLIR compilers for GPUs are not designed to handle or optimize such general-purpose implementations, and thus require substantial architectural extensions and novel optimization strategies to efficiently support the full diversity and complexity of scalar functions in analytical databases.

Our proposed compiler toolchain for *scalar functions* operates in two main stages. First, we leverage LLVM's Polygeist [48, 49] front-end compiler that lifts the C++ implementations of *scalar functions* into high-level MLIR dialects instead of generating low-level LLVM IR (or assembly). This approach preserves structured control flow (*while* and *for* loops) and composite data types (*structs* and *classes*) enabling optimizations such as loop transformations, data layout restructuring, and high-level analyses that are difficult to perform on low-level LLVM IR. The core contribution of our work lies in the second stage: a specialized code generator that applies GPU-specific optimizations to generate efficient kernels for *scalar functions*.

The initial IR preserves the loop structure of the columnar interface, allowing our code generator to identify parallelism, apply loop tiling, and map computation to the GPU's *grid-block-thread* hierarchy. To maximize memory bandwidth, the generator targets *wide-word load and store* instructions via an *unroll and jam* pass, enabling each thread to process vectors of inputs efficiently. To further enhance kernel efficiency, the generator restructures data layouts of composite types by splitting struct pointers into field pointers and eliminates unused fields and fields that store constants, reducing register pressure and kernel size. These domain-specific optimizations, combined with standard compiler passes, produce compact and high-performance GPU kernels.

We integrate kernels generated by our compiler into a GPU database that is based on Tensor Query Processing (TQP) [34]. Our compiler toolchain for *scalar functions* preserves the retargetability and performance portability of TQP, ensuring that the database can efficiently support new and future GPU architectures.

Benchmarks and Evaluation. We introduce *TPC-H-scalar*, a benchmark suite that augments TPC-H queries with realistic usage of *scalar functions* (see Figure 1 for examples), based on patterns observed in production workloads. The original TPC-H queries often omit *scalar functions* by assuming canonicalized data (e.g., string columns such as `p_type` are capitalized) and convenient parameters (e.g., `@startQuarter = '1993-07-01'` rather than `@year = 1993` and `@quarter = 3`). Our variant incorporates a diverse set of *scalar functions*, encompassing string manipulation, date operations, and numeric conversions. Most queries (19/22) remain equivalent to the TPC-H specification, ensuring relevance and comparability (Section 5 discusses this in more detail).

We compare the performance of our GPU database with *scalar function* support against a state-of-the-art CPU database system (SQL Server) on 3 sets of benchmark queries. In addition to *TPC-H-scalar*, we also compare against a sample of the production workload, and a set of micro-benchmark queries; each exercising a single

scalar function. We report that the GPU engine we integrate with has significantly better performance than a scale-up CPU engine that uses slightly (28%) more expensive hardware. It achieves an average per-query speedup of 7.6× on *TPC-H-scalar* and 6.4× on production queries. Further, our evaluation shows that adding realistic scalar functions to TPC-H queries has minimal impact on GPU execution time, while causing a 2.8× slowdown on CPUs. This more than doubles the overall speedup of the GPU database. We further establish that the generated kernels are competitive with hand-written CUDA or tensor-operation-based implementations—and in some cases outperform them due to specific optimizations—while also supporting generation across GPUs from multiple vendors.

In summary, in this paper we make the following contributions:

- We quantitatively and qualitatively show that *scalar functions* are expensive to evaluate on CPUs and discuss the challenges in accelerating them with GPUs.
- We develop a novel MLIR-based compiler toolchain that automatically translates existing implementations of *scalar functions* into optimized GPU kernels, thus ensuring that their behavior is consistent with CPU implementations.
- We introduce *TPC-H-scalar*, a benchmark suite that augments TPC-H with diverse and realistic scalar function usage, and make it publicly available, enabling systematic evaluation of *scalar functions* in analytical queries.
- We integrate the generated kernels into a GPU database and demonstrate that the end-to-end system delivers substantial speedups, further widening the performance gap between GPU and CPU databases.

The rest of the paper is organized as follows. Section 2 covers relevant background. Section 3 describes our compiler toolchain. In Section 4 we discuss integration into a GPU database system. In Section 5, we study the usage of *scalar functions* and design a new benchmark suite. Section 6 reports results from performance evaluation. We discuss challenges when adapting to other applications, like accelerating UDFs, in Section 7. Section 8 surveys related work. We conclude with a summary and future work in Section 9.

2 BACKGROUND

We begin this section with a summary of *scalar functions* supported in SQL Server and the span of their implementation. We motivate the possibility of using GPUs to accelerate *scalar functions* and introduce the LLVM-MLIR framework used to build our compiler.

2.1 SQL Scalar Functions

Scalar functions supported by SQL can be classified into four broad categories: *type conversion* functions, *date* functions, *string* functions, and *math/logic* functions[1]. Math and conversion functions each support multiple types and the implementation varies widely across types. Figure 2 shows examples of SQL Server functions of each kind. While the names may differ, equivalent functions are supported in other SQL systems [3, 5, 11, 12].

Figure 3 shows the overhead of executing *scalar functions* from each category on two CPU databases, SQL Server and DuckDB, using simple micro-benchmark queries. Each query reads two columns

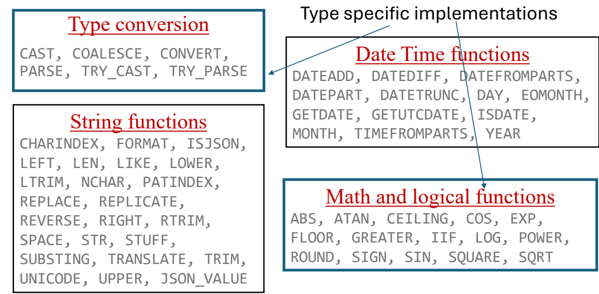

Figure 2: Examples of SQL scalar functions

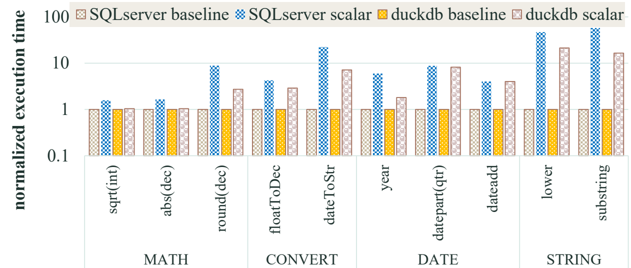

Figure 3: Overhead of evaluating scalar functions on CPU databases. The *baseline* query uses a simple predicate while *scalar* applies a scalar function as part of the predicate.

from the *lineitem* table of *TPC-H* and counts the number of rows that satisfy a predicate of a fixed selectivity. The predicate is either a simple comparison (for bar labeled *baseline*) or a *scalar function* on the same columns (for bar labeled *scalar*). The times are normalized with respect to the baseline times and plotted on a log scale[2]. The results show that executing *scalar functions* incurs substantial overhead, for all but basic math functions (*abs* and *sqrt* in figure). Functions on decimal types including rounding and conversion from floating point expressions incur an overhead of $2-10\times$. Conversion from date to string and other date functions also incur a high overhead of $1.5-12\times$. String functions have an even higher overhead slowing down performance by more than 10×. Below we discuss why *scalar functions* are computationally expensive and why they are a good candidate for GPU acceleration.

String functions Traversing through the individual characters in a string is central to all string functions. A production database like SQL Server supports a large number of character sets and collations. Strings are usually represented in the universal Unicode format using a collation specific encoding like UTF-8 or UTF-16. With such encodings, a single character can be represented by multiple bytes and traversing a string requires processing one character at a time. Traversal is only one part of the implementation of a string function. Additional functions are needed to perform computations like, comparisons between characters or applying transformation tables to change case, all in a collation specific manner.

Extracting parts of a date SQL Server supports several date types, that physically represent a date value using a 32 or 64 bit integer. For

[1]Different database engines may use varying categorizations or terminology; we regroup them into these four categories for ease of exposition.

[2]Note that the baseline times for the engines vary significantly; therefore, direct performance comparisons between the engines are not appropriate.

example, the default *datetime2* type tracks the number of ticks (typically in milliseconds) since the beginning of date 1/1/1 as a 64-bit integer. A common operation for all date functions is to breakdown a date from its integer representation to its constituent parts *year, month, day, hour, minute, second and millisecond fraction*. This is implemented by creating a composite *DateParts* type that has these constituents as separate fields. The conversion involves the invocation of about 15 functions with some, like counting the number of leap years, that require iterative computation, thereby increasing arithmetic intensity. Additional functions maybe invoked on the *DatePart* object to extract parts like *week*, *quarter* and *dayofyear* that are not directly represented as a field. SQL server supports multiple calendars and the implementation of date functions needs to account for calendar specific semantics.

Multi-word operations on decimal types Analytical databases perform high precision computations on decimal types so that they are not affected by rounding errors. Such types need multiple integers to represent each value (128 bits), and are represented by composite struct types. Implementation of *scalar functions* on decimal types, like adjusting the scale, rounding or converting from floating point requires multi-word operations. The implementation relies on a library of multi-word arithmetic operations that handles all the corner cases. The code includes detailed comments and mathematical proofs to reason about the correctness of operations. Despite using highly optimized libraries applying *scalar function* operations has a high arithmetic intensity due to the need to perform multi-word operations on composite types[3].

A note on implementation. The implementation of the above functions, spanning over 100,000 lines of code, makes rich use of advanced programming language features such as templates, polymorphism, function pointers, dynamic memory allocation, and uses standard C++ library functions— many of which are unsupported or only partially supported in GPU-specific languages like CUDA, particularly within device code. These are critical to conform to good software engineering practices, to maintain readability and enable extensibility. The code also gets frequent updates including several critical commits in the recent releases.

Compilation and execution in SQL Server. Scalar functions are compiled into CPU specific binaries during the build process. O3 level optimizations—such as inlining, unrolling, and vectorization—are applied to improve execution efficiency. The query optimizer provides first-class support for *scalar functions*, employing rewrite rules to transform plans with scalar operators in a cost based manner.

2.2 GPU Acceleration

The above discussion highlights that *scalar functions* are computationally expensive, and their implementation is complex. Modern GPUs like NVIDIA A100 are available at a price point [21] comparable to scale-up CPUs and offer a high memory bandwidth and an even higher arithmetic throughput. To understand the potential for performance improvement of GPUs compared with CPUs, we use a roofline plot—a theoretical model that relates attainable performance to arithmetic intensity (AI), defined as the ratio of

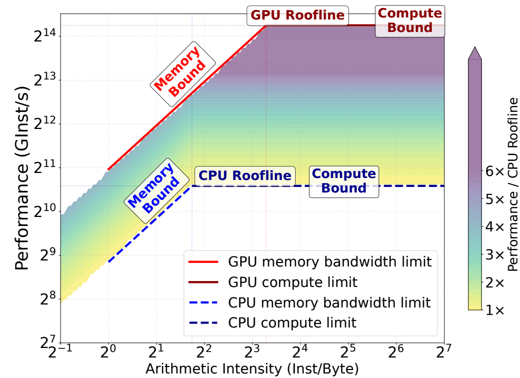

Figure 4: Roofline plot comparing CPU and GPU performance for functions with varying arithmetic intensity. The colored band indicates the speedup over CPU roofline.

operations to memory accesses. The roofline plot illustrates the maximum achievable throughput for a given AI, bounded either by memory bandwidth (for low AI) or by compute capability (for high AI). This visualization helps identify whether a workload is memory-bound or compute-bound and estimates the upper bound on speedup from GPU acceleration over CPU execution.

As illustrated in Figure 4, the speedup ceiling for compute-bound kernels is much higher than for memory-bound kernels. To understand the impact of this on *scalar functions* we position the main categories of *scalar functions* on the roofline plot based on empirical measurements; further details on our methodology are provided in Section 6.5. As can be seen simple math and conversion operations, with arithmetic intensity (AI) in the range of 1–4, are memory-bound, and their theoretical speedup is limited by relative memory bandwidth. In contrast, most other *scalar functions* exhibit higher AI and are compute-bound, making them well-suited to benefit from the GPU's superior computational throughput.

2.3 Compiler Design with LLVM-MLIR

MLIR [45] (Multi-Level Intermediate Representation) was introduced in 2019 as a flexible compiler infrastructure to address the challenges of targeting diverse and rapidly evolving hardware accelerators. Built on top of LLVM, MLIR organizes program representations using a modular system of *dialects*, each defining its own *operations* and *types*. This extensible intermediate representation (IR) allows developers to introduce custom domain-specific abstractions tailored to new hardware or application domains. The MLIR ecosystem includes dialects for standard high-level language constructs such as *functions*, *loops*, *if-then-else*, *memory* and *math* operations, providing a common foundation for expressing a wide range of computations. All MLIR programs are required to be in Static Single Assignment (SSA) form, where each value is assigned exactly once and has a well-defined, immutable type. This design enforces strong typing and enables robust type checking throughout the compilation process. This architecture allows compilers to efficiently target diverse hardware platforms by introducing customizable *optimization* passes to rewrite programs and *lowering passes* that progressively transform high-level abstractions into hardware-specific code.

[3]Note that scalars are more expensive than aggregations as for the latter the implementation can use distributive properties to accumulate and fix precision later.

Most existing MLIR-based compilers are designed for domain-specific languages with a limited set of operations. For example widely used compilers for OpenAI Triton [17] and Tensorflow [9], target specific primitives used in deep learning. Additional compilers for domains like sparse tensor computation [22], decision forest inference [53] or relational algebra [40], define their own custom dialects. Custom compilation passes are then used to match specific patterns and lower from these dialects to existing MLIR dialects. This lowering process continues until the intermediate representation (IR) satisfies the preconditions required by the pre-built code-generation passes to generate hardware-specific code. Unlike these compilers we begin with existing implementations of *scalar functions*, which make extensive use of general-purpose language features such as loops, conditionals, structs, and classes. Current MLIR-based compilers are not equipped to handle this level of language complexity or to optimize such code effectively, making them unsuitable for our use case without significant extension. Building an effective compiler for scalar database functions presents several key challenges even when leveraging MLIR's existing infrastructure. We create a custom compiler with about 50 different optimization and lowering passes, five of which were written from scratch to address specific challenges encountered during compilation of *scalar functions*. Below we describe the steps in code-generation and the challenges we address in each step.

Initial Code Quality Issues. A direct translation from C++ scalar functions to MLIR produces programs which are large and highly inefficient, their code size can sometimes be reduced by 10× through extensive optimization. Certain *scalar functions* create objects with many fields which are represented in the IR as pointers to complex types, and individual elements are access through several loads and stores. As these classes are templatized and shared across multiple functions in the C++ implementation, often times only a subset of fields and code within functions are relevant to a specific *scalar function*. Further, the implementation often involves early exits from functions and loops, which when translated to MLIR results in several redundant branches that read and write state variables allocated using *memref* operations. Addressing this required applying a known optimization technique called *Scalar-Replacement of Aggregate* at high level dialects in MLIR.

Loop optimization. The next step is to prepare the loops for lowering to the *GPU* dialect and do so while maximizing resource efficiency. We enhance existing MLIR loop transformations and introduce custom passes to optimize memory access patterns and thread utilization. Specifically, we implement a novel sequence of optimizations that increases per-thread workload by coarsening loops so that each thread processes multiple elements per kernel invocation. We then generate vector loads to access all elements with a single hardware instruction.

GPU kernel generation. Successfully combining custom optimization passes with existing MLIR passes requires careful sequencing and coordination. Additionally, care needs to be taken to ensure that constants are not aggressively propagated out of kernels and passed as parameters. Sinking such constants into the kernel is crucial for enabling subsequent optimization passes to generate specialized, high-performance code.

We describe the compiler in more detail next.

3 COMPILING SCALARS FUNCTIONS INTO GPU KERNELS

Our compiler targets the batch interface for implementing *scalar functions*, which processes column vectors as input and produces output vectors, aligning with the columnar execution model prevalent in modern analytical databases[4]. Table 1 summarizes the MLIR dialects used during compilation.

Table 1: MLIR dialects used in the implementation.

Dialect	Description
FUNC	The dialect represents functions with operators to encapsulate the function body and calls to other functions.
AFFINE	Represents loops that have statically analyzable bounds and step sizes. Such loops, the outer loop of *scalar functions* being an example, are candidates for parallelization.
SCF	Represent structured control flow operations like *for* and *while* loops and *if-then-else* regions within a function body.
MEMREF	Represents memory operations, like allocations, loads, stores and type casts. The operations support complex types arrays, structs, tensors and arbitrary nests of them. Subsequent passes lower these operations to simpler types and finally to machine memory operations.
ARITH	Represent standard operations on scalar values including arithmetic and comparisons on basic types.
MATH	Represents math functions like those in C++ std libraries.
POLYGEIST	Polygeist has operations in its own dialect that are used in cases where the other high-level dialects prove to be insufficient, e.g., when a *memref* needs to be interpreted as a pointer. The *Polygeist* dialect has a *memref2pointer* op that converts the *memref* to a *LLVM* pointer type.
LLVM	A dialect with identical semantics to LLVM used to represent lower-evel operations. Can appear in high-level IR too, typically when pointer arithmetic are involved.
GPU	Low-level device-independent abstractions for launching GPU kernels. Can be lowered to from parallel loops in *scf*.
VECTOR	Represents operations on multi-dimensional vectors. Can lower to wide-word instructions on the GPU.

3.1 Translation to MLIR

The first step towards compilation is to translate the existing implementation of a function (including all other functions that it refers to) to MLIR [45]. We rely on LLVM Polygeist [48, 49], a frontend compiler that translates the C++ AST generated by a standard compiler—already resolved for templates and polymorphism—into MLIR. It maps C++ constructs (classes, member functions, globals, etc.) to corresponding MLIR types and operations. Classes become structs, and methods are represented in the *func* dialect.Polygeist then inlines functions and applies canonicalization passes [48], producing an MLIR representation using high-level dialects listed in Table 1. Figure 5 illustrates an example of the intrinsic YEAR, whose implementation invokes multiple functions as discussed before in Section 2. It constructs a class (fn2) representing date parts, and calls functions that initialize (fn3) and update (fn5) fields.

3.2 Simplification and Code Size Reduction

The initial MLIR generated by the frontend compiler is sometimes very inefficient. The key to simplifying this code is an effective

[4]SQL Server also supports a row mode interface that is typically used for transactional workloads.

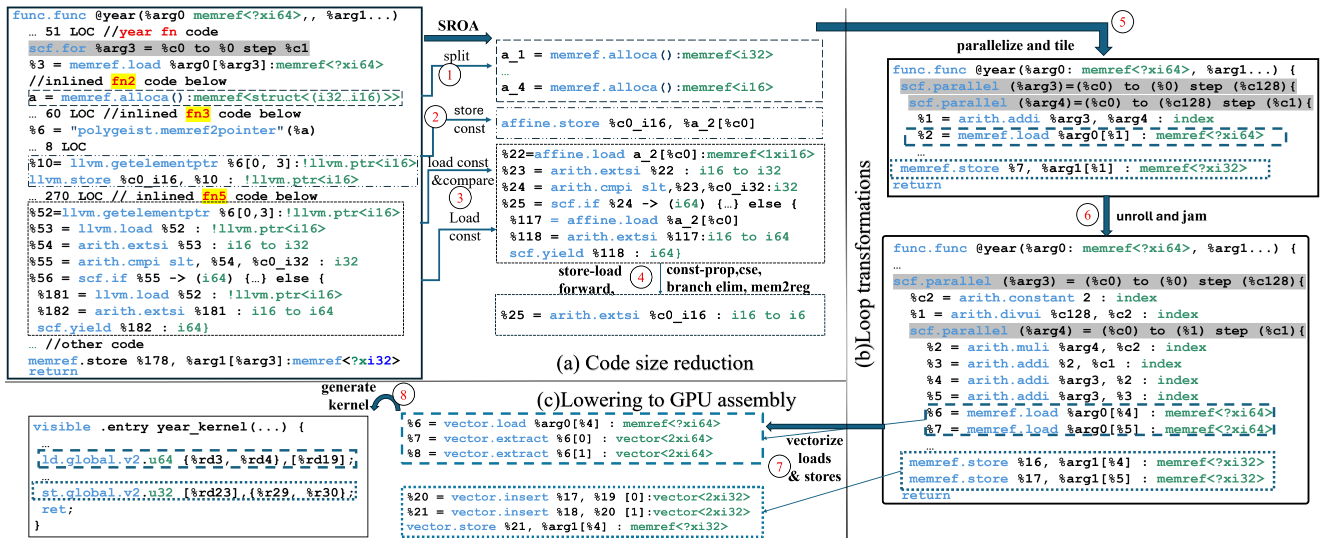

Figure 5: Code generation from MLIR representation of scalar function to GPU kernels. The figure shows how the implementation of YEAR with code from multiple functions is (a) simplified through passes like SROA (b), transformed through loop tiling, unroll and jam (c) and finally lowered to GPU assembly targeting vector instructions. Each box only shows the modifications made to a single block of code, the remaining code is omitted for brevity.

optimization pipeline that can eliminate unused fields, memory accesses and dependent control flow. We find that MLIR lacks effective passes that can reason over the combination of dialects, like *memref* and *polygeist*, to simplify this code. We therefore introduce an SROA (scalar replacement of aggregates) pass that can break down accesses to fields of large *structs* and *classes* that would help eliminate unnecessary fields and code paths within functions.

The custom SROA pass is applied together with a built-in pass for store load forwarding and several custom peephole optimizations. The figure shows an example of how these passes interact and reduce code size. SROA splits allocations for individual fields ① and isolates fields which are updated with constants ②. Through additional analyses it determines loads of constants ③, and subsequently eliminates parts of code through peephole optimizations ④ eliminating unnecessary branching. The effect of these can be seen Figure 5(a) where an initial MLIR with over 500 lines is reduced to 50. As we report later Section 6 applying these optimizations at the high level IRs is critical as passes in lower level IR (like LLVM) are not always able to identify and optimize these specific patterns in Polygeist translated program representations.

3.3 Loop Transformations

The simplifications mentioned above are independent of the target. Next (see Figure 5(b)) we transform the loop to tiled parallel loops (⑤) aligning the structure to the GPU thread hierarchy. After tiling, an *unroll and jam* transform is applied to the inner loop ⑥. This has the effect of a thread coarsening [37, 39] transform on the GPU kernel. Such a technique can improve bandwidth utilization, reduce redundant operations across threads and lower warp and thread block scheduling overhead [37, 39]. Once the loop is unrolled and jammed, there tend to be loads that are loading contiguous values. We transform such loads to wider vector loads ⑦ using the *vector* dialect. Usage of wide word memory instructions has been shown to reduce instruction count and improve memory bandwidth utilization[37] of GPU kernels. Even advanced GPU compilers like those for CUDA fail to utilize such instructions.

3.4 Lowering to GPU Assembly

MLIR has a robust pipeline for lowering from parallel loop-nests to GPU assembly, and we leverage this without too many modifications. The function with the tiled *scf.parallel* loops are transformed into operations in the *gpu* dialect using a built-in kernel outlining pass. The body of the loop-nest is transformed into a *gpu.func* with loop indices being computed appropriately for the GPU thread. One detail we take care of is to ensure that all constants are sunk inside the kernel to prevent them from being passed as parameters to the GPU kernel, this helps the subsequent lowering passes to specialize the generated code to specific constant parameters. All used dialects within the kernel are lowered to LLVM and we use LLVMs code-generator to generate GPU assembly ⑧. Note, the final kernel makes use of the wide-word load *ld.global.v2.u64* that loads two values with a single instruction.

3.5 Auto-tuning and Pre-compilation

As the set of *scalar functions* to be supported is known apriori we pre-compile the kernels for each *scalar function*. The pre-compilation is done as part of the system build process, and only needs to be performed once per GPU hardware. To specialize the performance to the hardware, we augment the compiler with an auto-tuner that sweeps through different values for two loop optimization parameters namely, `tileSize` ∈ {64, 128... 1024} and `unroll_factor` ∈ {1, 2, 4} to find the best performing kernel for each *scalar function*. The auto-tuner runs offline, it benchmarks candidate kernels on test inputs from SQL Server, and selects the fastest configuration.

4 SYSTEM INTEGRATION WITH TQP

We extend a GPU engine based on the tensor query processing (TQP) framework [34]. TQP utilizes tensor algebra to design a portable query processing framework, and already has support for common relational operators like *join, filter, group-by, aggregate* and *sort*. Several components of the end-to-end system were extended to enable execution of queries with *scalar functions*. We provide a brief summary of these below.

Query optimizer and tensor abstraction layer The GPU engine's query optimizer builds upon the CPU-based optimizer to identify sub-plans composed of GPU supported operators to offload [38]. By extending the optimizer to recognize and offload scalar operators—often appearing early in the plan—we enable execution of more queries on the accelerator.

The plan pushed to the GPU is then translated to an operators in the tensor abstraction layer (TAL) [38]. We extend TAL with additional operators for *scalar functions* and extended the runtime to support the invocation of pre-compiled kernels. The required kernels are then dynamically loaded during query execution.

Data representation and transfers. Input tables use a compressed columnar format, and data chunks are sent to the GPU in native form to save PCIe and memory bandwidth [38]. The data is decompressed into one dimensional tensors before processing. Each operator consumes one or more tensors and outputs new tensors.

One place where there is a mismatch between the CPU and GPU representation is for string columns. Unlike SQL Server where strings are stored either inline (if small enough) or as pointers to a different part of memory, TQP represents a vector of strings as a set of three tensors, one for the string data, one for the offsets and one for the lengths of each string. We therefore extend the implementation of each *scalar function* on strings with a stub function that takes a triple of vectors as input and produces an output triple of vectors. The stub function invokes the row mode implementation of the *scalar function* in a loop.

Future considerations The other popular mode of query evaluation is JIT compilation [7, 32, 40, 50], where multiple operators are fused together. As our compiler relies on the MLIR framework we believe that it can in future be integrated with MLIR based JIT engines like LingoDB [40] Another important consideration is multi-tenancy, a scenario where multiple queries run concurrently on the same hardware. We note that adding support for *scalar functions* does not preclude multi-tenancy. The generated code is thread-safe, operates on pre-allocated vectors, and avoids global state, making it compatible with multi-query execution models.

5 BENCHMARKING

Existing database benchmarks for analytics, such as *TPC-H* and *TPC-DS*, are effective at capturing prevalent query patterns involving relational operators . However, these benchmarks make only minimal use of *scalar functions*, and thus do not adequately evaluate the performance impact of using them. We therefore propose *TPC-H-scalar* an enhanced variant of the *TPC-H* benchmark that incorporates *scalar functions* in a manner more representative of their usage in production queries. We hope that this benchmark will help establish a starting point for evaluating the performance of *scalar functions* in analytical databases, and provide a common ground for future research in this area.

```
DECLARE @p1,@p2,...;
SELECT    agg_func1(scalar_func(expression)) AS agg1,
          agg_func2(col2) AS agg2,...
FROM      table1 JOIN table2 ON ...
WHERE     scalar_func(col3) binOp @p1 AND col4 binOp @p2...
GROUP BY  scalar_func(col5),col6,...
```

Figure 6: Generic query pattern observed in the workload.

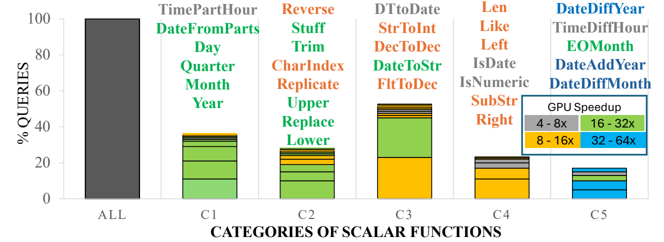

Figure 7: Stack plot showing breakdown of scalar function usage by category across queries that utilize at least one function. Each category shows the constituent scalar functions and their GPU speedup over CPU (color-coded per legend).

5.1 Usage of Scalar Functions

To identify common usage patterns, we analyzed a production critical workload trace from a large analytical database in SQL Server, which contains over a hundred tables with 9 fact tables. The trace contains 935K queries gathered over a period of 6 months. Several queries from this database have been flagged for poor compute performance. The queries follow a simple *select-project-join-groupby-aggregate* pattern shown in Figure 6 applying *scalar functions* during filtering, on grouping columns and on aggregation expressions. The queries vary in the number of *scalar functions* used $(0-10)$, number of tables joined $(2-25)$, the number of filter predicates $(1-8)$, the number of columns grouped on $(1-8)$ and the aggregation expressions used (usually $1-5$ of 15 different expressions). We post-processed the query log to identify common expressions that use a *scalar function*. We also gather metrics on how often they were used used per query. We find that 58% of the queries use at-least one *scalar function*, with some queries using as many as eight.

Next we report usage statisics of individual *scalar functions*. Note that most *scalar functions* support multiple parameters, and their implementation can vary significantly depending on the parameters used (e.g., CONVERT from date to string vs. float to fixed point). Further, different function and parameter combinations sometimes use the same implementation as they perform the same operation (e.g., CONVERT, CAST, PARSE, FORMAT, STR). As our focus is on identifying computationally interesting patterns, we report interesting function/parameter combinations, grouping entries that share implementations under the most used function/parameter combination. Figure 7 shows a stack plot of *scalar functions* used in the workload and the fraction of queries (among queries that use at-least one intrinsic) that utilize them. The usage is broken down into 5 categories (detailed later in this section) each being used by at least 20% of queries[5]. The bar for each category is composed of stacked blocks representing individual functions. The names and

[5]There are a few math (ABS, SQRT, ROUND, CEILING, FLOOR) functions that are used in less than 1% of queries that are not represented by the categories.

Table 2: Commonly found categories of scalar expressions in the workload. The table lists the category of the expression, the scalar functions used in the category along with their input and output types, gives an example of how the scalar is used in a query, and how this use case can be applied to TPC-H queries.

Category	Scalar	Input type	Output type	Example	Changes for *TPC-H-scalar*
C1. Extract date part for filter or projection	YEAR, MONTH, DAY, DATEPART, EOMONTH	*date, datetime, datetime2*	int	`YEAR(col) = @p` `DATEPART(quarter, col)`	Many queries involve filtering on month, quarter or year [Q4-Q6,Q8,Q10,Q12,Q14-15,Q20]
C2. Normalize string before compare	LOWER, UPPER, REPLACE, REPLICATE	*char, varchar*	*char, varchar*	`LOWER(col) IN` {air, microsoft} `REPLACE(col,'-','/')` (1-1-1981 → 1/1/1981)	Queries assume strings are upper case, while obvious from column name. [Q2-3,Q5,Q7-Q8,Q11-Q12,Q14-21]
C3. Convert types before output or aggregation	CONVERT, PARSE, FORMAT	*date,datetime, datetime2*	char	`CONVERT(char(10), dateCol, 'mm/dd/yy')`	Queries that output dates [Q3,Q18], and those with expressions on identical types, when changed to use mixed types [Q1-3,Q5-7,Q9-10,Q13-15,Q19].
	CONVERT, CAST, ROUND	*decimal, float, money*	*decimal, float, money*	`CONVERT(money,` $price \times 0.999999$`)` `ROUND(`$price \times$`exch_rate,1)`	
C4. Extract or validate part of string	RIGHT, LEFT, SUBSTRING	*char varchar*	*char, varchar*	`RIGHT(FiscalYearStr,2)` e.g. `RIGHT('FY22')` → 22	Columns like *brand, orderpriority* have a numeric field as part of string; existing queries [Q12,Q16,Q17,Q19] filter on whole string
	ISNUMERIC, LIKE, ISDATE	*char, varchar*	*bool*	`ISNUMERIC(col), ISDATE(col),` `p_brand LIKE '%#[∧13]_'`	
C5. Perform date arithmetic	DATEDIFF, DATEADD	*date, datetime, datetime2*	*date, datetime, datetime2, int*	`DATEDIFF(day, c1, c2) IN (2,3)` `DATEADD(day, col1, 5) > col2`	Using date arithmetic instead of less than based comparison [Q1,Q3,Q21].

relevant parameters for the function are shown above the bar in the same order. As shown, a small set of *Scalar functions* (YEAR, MONTH, LOWER, RIGHT, CONVERT(FLTTOMONEY), CONVERT(DATETOSTR)) account for the majority of usage, but at the same time a long tail of other *Scalar functions* are employed throughout the workload. The *scalar functions* are also color coded by the speedup obtained when executed on GPU using our compiler compared to CPU execution. As can be seen most *scalar functions* achieve a speedup of $10 - 32\times$. We discuss the performance results of individual *scalar functions* in more detail in Section 6.

Table 2 groups *scalar function* usage into 5 generic categories, that can be applied to a wide range of analytical workloads. The table also reports how these could be applied to TPC-H queries.

5.2 Incorporating Scalar Functions in TPC-H

The TPC-H benchmark specification [19] defines each query with a *business question* that describes the intent of the query, a *functional query definition* that provides a parameterized SQL query, and a set of *substitution parameters* that define the various values that the parameters can take. We go through the categories in Table 2 and discuss the proposed changes to the benchmark. Note that the modifications for C1—C4 introduce *scalar functions* into TPC-H without changing the the semantics of the query. We report that 19 of the queries in *TPC-H-scalar* are equivalent to the original queries. For all queries we preserve the plan shape and cardinalities throughout the plan, ensuring that the queries preserve the performance characteristics of the relational operators. We modify them by replacing projection and filter expression with *scalar functions* while preserving the high-level analytical intent.

C1: Extract date part. The *business question* for several TPC-H queries include statements of the form *'in a given quarter of a given year', 'in a given month and year', 'in a given year', or 'in a given two year period'* that directly lend to usage of *scalar functions* in C1 from Table 2. Nine queries each include such a statement but their *functional query definition* compares a date column against string parameters that represents a date in a database specific format. In similar scenarios, production queries are parameterized by a numeric parameter that represents the value of the date part to compare against. We modify these TPC-H queries to to apply *scalar functions* to a date column (one of `l_shipdate, o_orderdate, l_receiptdate`) and compare against a value representing the *year, month, quarter*. For example, Q4 is modified to take a parameter `@pyear` and `@pquarter` and filter on `YEAR(o_orderdate) = @pyear AND DATEPART(quarter,o_orderdate) = @pquarter`.

C2 : Normalize string before comparison. TPC-H queries assume that string columns are in upper case, this is somewhat arbitrary and does not hold in production settings where string comparisons are commonly performed after normalizing with operators like UPPER, LOWER, REPLACE. This is done even for columns that contain names (e.g. country) as there could be ambiguity between upper case and initial capital words (*ASIA vs Asia*) or usage of punctuations (*U.S.A vs USA*). We modify queries that perform string comparison to use *scalar functions* before comparing against constants. For example, we modify Q8 to use `LOWER(P_TYPE)= 'economy anodized steel'` instead of `P_TYPE = 'ECONOMY ANODIZED STEEL'`.

C3: Type conversion. Conversions are commonly used in two scenarios. First production queries often format *date* columns (*mm-dd-yyyy,mm/dd/yy*,etc.) before output. We modify two TPC-H queries Q3,Q18 that output `O_ORDERDATE`, to format the date string using CONVERT before output. For example, we modify Q3 to use `CONVERT(char(10), O_ORDERDATE, 111)` which formats the date as 'yyyy/mm/dd'. Note that the TPC-H specification [19] (section `2.2.3.3f`) explicitly states that usage of *scalar functions* for formatting of output is a permissible modification of the benchmark.

The second use case is to use CONVERT to ensure precise arithmetic. Production queries often aggregate over expressions that perform computations to combine *floating point* and *decimal* columns (sometimes of varying scale). Interestingly, just like *TPC-H* the decimal columns in the production database represent *money* types, however they are used in expressions that perform currency conversion (by multiplying with `exchange_rate`) and hence require CONVERT functions to ensure precise arithmetic. In TPC-H all *decimal* columns have the same *precision* and *scale* and the database does not have any float columns. To simulate this scenario we modify expressions using `l_discount` or `l_tax` by scaling the use of

these columns by float constants[6], and converting the whole expression to *money* type. For example, Q5 uses SUM(CONVERT(money, l_extendedprice × (1 - l_discount ×0.999999))) instead of SUM (l_extendedprice × (1 - l_discount)).

C4 : Extract or validate part of string. A common production use case for this category is to extract a numeric part of a string and use it in a predicate. TPC-H Q22 already uses *substring* to extract country code from a phone number. Few other queries Q12,Q16,Q17 filter on strings that have a numeric part which is the basis for the predicate. For example Q12 looks for o_orderprioriy = '1-Urgent' and Q17 looks for p_brand = 'Brand#23', we modify these queries to use the *scalar functions* RIGHT, LEFT to extract the numeric part of the string and compare it with a parameter. For example, we modify Q12 to use RIGHT(o_orderpriority,1) = @p, and Q16,Q17,Q19 to use RIGHT(p_brand,2) = @p to filter on a brand number.

C5 : Perform date arithmetic. Production queries often compare two date columns, for example to determine the period of a subscription. Some TPC-H queries compare two date columns directly or compare each of them against the same constant. For example Q21 identifies *'suppliers who were not able to ship products in a timely manner'* this involves a direct comparison *l_receiptdate* > *l_commitdate*, we modify this query to perform a filter that includes DATEADD in a manner that it retains a similar predicate selectivity. We use the predicate DATEADD (dd,-2,L1.l_receiptdate) > L1.l_commitdate. In addition, we modify Q3 to use DATEDIFF(dd, o_orderdate, l_shipdate) IN (@p,@p+1,@p+2) instead of o_orderdate < @d' AND l_shipdate > @d where @d is a date string (example '1995-03-01'). Finally we modify Q1 that uses the expression l_shipdate < @d with a function l_shipdate <= DATEADD(month, 3, l_commitdate), the original query looks for *'items shipped within 60-120 days of greatest shipping date'* the modified query looks for *'items shipped at least 3 months before commit date'*. These are the only three queries that are not equivalent to the original TPC-H queries.

6 PERFORMANCE EVALUATION

The objective of our evaluation is to showcase that using our compiler to generate GPU kernels for *scalar functions*, from existing CPU implementations, is a practical way to improve their performance. We demonstrate that executing queries with *scalar functions* on GPUs yields significant performance benefits on a variety of different queries. Additionally, we show that the generated kernels are competitive with hand-written CUDA or tensor algebra kernels. Since manually rewriting kernels for every *scalar function* is tedious and error-prone, we only implement hand-written kernels for a limited set of operators, often under simplified assumptions. Finally, we show portability across GPU vendors, evaluating performance primarily on NVIDIA GPUs while demonstrating support for AMD GPUs.

Benchmarks. We evaluate the performance on three sets of the queries. In addition to *TPC-H-scalar* described in Section 5 (at scale factor 100), we also evaluate our compiler on a slice of the production workload and on micro-benchmark queries. The production

[6]Alternately, one could modify the types and values of these columns in the database. But this would involve changes to data generation or loading.

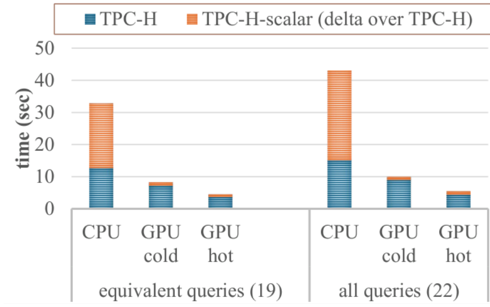

Figure 8: The stack plot shows the overhead of adding scalar functions to *TPC-H* to produce *TPC-H-scalar*, *equivalent* consists of 19 queries in *TPC-H-scalar* that have identical semantics to the *TPC-H* query and *all* contains all 22 queries.

benchmark consists of 50 queries derived from the workload used in Section 5. We run the queries against a 100GB database consisting of a subset of the tables and columns that were made available for evaluation. We picked queries from the workload that cover all the expressions (over columns that were in the database) that involve *scalar functions*. The micro-benchmarks, as introduced in Section 2, run against TPC-H data and are used to assess the performance of a single *scalar function*. They scan a single table, apply a filter based on a *scalar function* and perform a count() aggregation.

System Setup. The GPU evaluation was primarily conducted on an Azure NC24ads A100 V4 VM equipped with 24 CPU cores, and an 80GB NVIDIA A100 GPU connected through PCIe 4.0. All CPU evaluation was done on a CPU VM that is 28% more expensive [21], an Azure E64ads v4 VM with 64 CPU cores and 512GB RAM. We compare against SQL Server, a commercial CPU database system (labeled CPU in figures). Our GPU database system (labeled GPU in figures) shares the same query optimizer but uses a different runtime as discussed before. Each query is run five times, and we skip the first run and report the average of the remaining runs. This is done to ensure that the data is warm in the CPU or GPU memory[7] for later runs. We further evaluate performance under an additional cold start scenario where we disable GPU caching. In this setting the first run brings the data into CPU memory, but all runs incur the overhead of transferring data from CPU to GPU memory. We additionally demonstrate portability on AMD GPUs.

6.1 Performance Evaluation on *TPC-H-scalar*

Figure 8 reports the overhead of adding *scalar functions* to TPC-H to create *TPC-H-scalar*. As can be seen, the cumulative execution time (the time to run all queries in the benchmark) on CPU increases by 2.8×. In comparison, incorporating the compiler kernels on the GPU database has much lower overhead — 24% when data is hot on the GPU and 9.8% when data needs to be transferred over PCIe.

Figure 9 reports the execution time of all 22 queries in *TPC-H-scalar* on both CPU and GPU (both hot and cold runs). As can be seen, when data is already on GPU memory a majority of queries achieve a more than 5× speedup, the cumulative time reduced by 7.6×. When data needs to be brought over PCIe, the speedups are lower but still significant, reducing the cumulative latency by 4.3×. The overall effect is that many queries that take several seconds to

[7]At scale factor 100, the working set of the query can fit entirely in GPU memory.

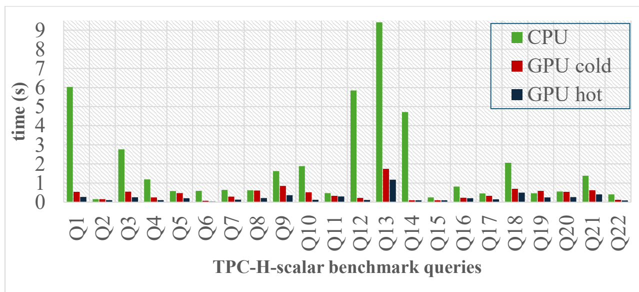

Figure 9: Performance Evaluation on *TPC-H-scalar*

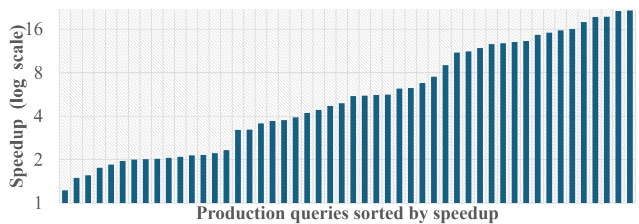

Figure 10: Speedups observed on production queries

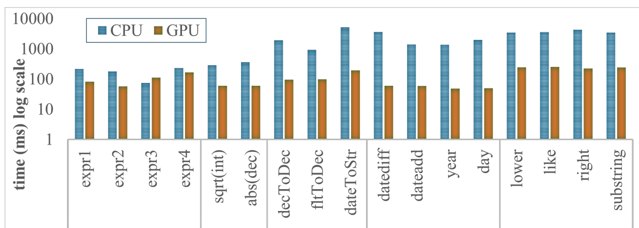

Figure 11: Micro-benchmarks comparing the performance of *select-project* queries. Note that the y-axis is in log scale.

run on the CPU, run at interactive speeds on the GPU (assuming their input is already in GPU memory).

6.2 Performance on Production Queries

Figure 10 shows the speedup (GPU hot) observed on a set of 50 production queries. As can be seen, just like with *TPC-H-scalar* a majority of the queries achieve a speedup of more than 4×, with about a third of them achieving more than 8×. We also report that the average speedup across all queries is 6.4×.

6.3 Performance on Micro-benchmark Queries

To provide a more detailed view of the performance summary in Figure 7, we report CPU and GPU execution time for micro-benchmark queries. We report results for the most frequently used *scalar functions* from each category (as reported in Figure 7) and for reference also include simple *math* scalars and expressions without any *scalar functions*. As can be seen in Figure 11, simple queries that do not have any *scalar functions*, are the fastest to run on CPU, and the GPU only achieves limited (up to 3×) performance gains over CPU. The CPU has a highly optimized scan that can often perform single table operations on compressed data — sometimes using SIMD instructions. Further we find that queries with simple math *scalar functions* like ABS, SQRT also yield limited (up to 4×) benefits. The computation in these functions have a limited arithmetic intensity, and the speedup is limited to the memory region of the roofline (see Section 6.5 for additional discussion). The other micro-benchmark queries employ functions with higher arithmetic intensity, they typically run longer on CPU, and the GPU attains at-least 10× speedup. It is important to note that speedup does not always scale directly with CPU execution time. The achievable speedup depends on the computational characteristics of each function. In particular, string functions with data-dependent memory access patterns are more challenging to optimize on GPUs and typically yield lower speedups than others (10 – 15× compared to 12 – 60×). We discuss this further in Section 6.5.

6.4 Comparing Alternatives

We perform additional experiments to understand the benefits of the compiler optimizations and to establish that the generated kernels are competitive with alternatives like CUDA and tensor algebra. For these analyses we pick a subset of *scalar functions*, covering the different categories, and evaluate different kernels alternatives (see Figure 12) using a stand-alone benchmarking framework (no GPU engine) that copies data on GPU, performs multiple timed runs of the kernels and copies the results back. We generate unoptimized kernels from our compiler by only using the minimal set of passes (parallelization-and-tiling from Section 3.3 and lowering GPU assembly from Section 3.4) to produce a GPU kernel. As can be seen in Figure 12, the newly added optimizations in our compiler are crucial to achieve good performance. Benchmarks like YEAR, DATEPART, DATEDIFF get significant benefits from both code simplification and the use of wide word instructions, and CONVERT to decimal gets most of its benefits from wide word instructions. The other benchmarks achieve smaller but non negligible benefits.

The CUDA kernels were manually written with two different strategies. The first was to write expert kernels for specific functions (SUBSTRING and LIKE), by carefully designing GPU friendly algorithms. Note that the CUDA kernel for LIKE only supports one special character (% for matching zero or more characters) while the compiled kernel supports all supported special characters (requires a more complex algorithm) and has identical behavior to the CPU database. These kernels were written by CUDA experts with significant experience in performance optimization. Each kernel took about a month to implement. As can be seen in Figure 12 the CUDA kernel for LIKE has identical performance while the manually written CUDA kernel for SUBSTRING is faster. We report that the algorithm used in both these CUDA implementations is very different from the one used in the CPU database. Second, for specific intrinsics CONVERT(FLTTODEC), DATEPART, DATEDIFF and YEAR we extract the core files and functions from the CPU implementation and write CUDA kernels based on them by replacing constructs not supported by CUDA with suitable alternatives, and annotating functions and relevant global arrays with `__device__` directives. No algorithmic effort was involved in this re-implementation. They took between a few days to a week to implement. As seen the automatically generated kernels were faster than CUDA kernels written using this approach. Finally we also implement YEAR in tensor algebra with a series of six division and modulus operations on tensors, followed by a scaled addition. This implementation took an hour, does not cover all date ranges, and is about 7× slower.

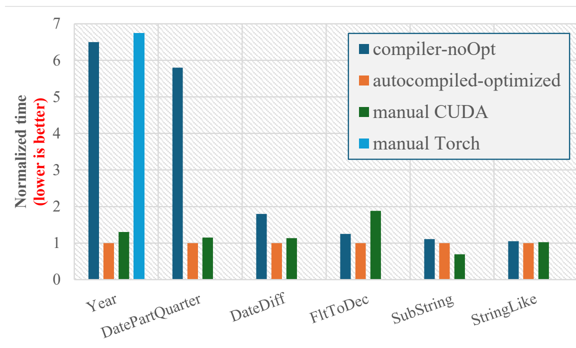

Figure 12: Performance comparison of automatically compiled kernels versus hand-written CUDA implementations and analysis of the impact of compiler optimizations.All results are shown relative to the optimized kernel generated by our compiler. The CUDA kernels were compiled with `nvcc` version 12.6 with `-O3` optimizations.

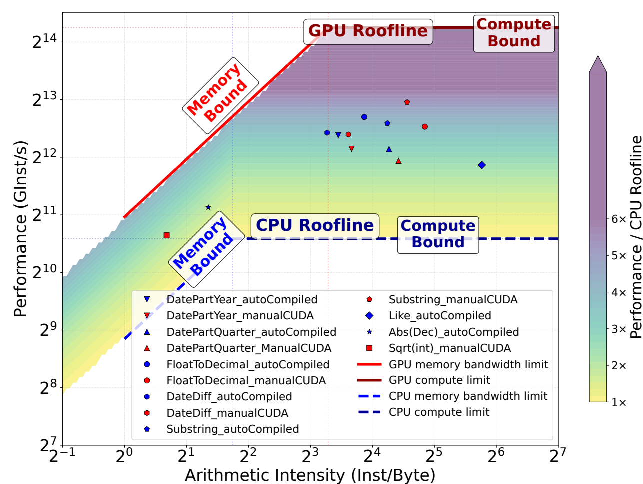

Figure 13: Roofline plot for kernel alternatives. The x-axis is the arithmetic intensity (instructions per byte) and the y-axis is the achieved performance (GInst/sec). The roofline bounds for the CPU and GPU are shown for reference.

6.5 Performance Characteristics of Kernels

We collect performance counters for GPU kernels and visualize them using a roofline plot, following the methodology described in [26]. The roofline boundaries are determined using the theoretical peak memory bandwidth of 2 TB/s and compute throughput[8] (19,500 GInst/sec) of the A100 GPU. For reference, we also plot the roofline for the CPU used in our experiments [6], with a memory bandwidth of 460 GB/s. The CPU's theoretical peak compute throughput is estimated by multiplying the number of cores (64) by the peak instructions per cycle (IPC) of 8 at a clock speed of 3 GHz, resulting in 1,536 GInst/sec. This value represents an upper bound; it assumes all integer operations are single cycle and that all the out-of-order cores can find 8 independent instructions every cycle.

[8] *Scalar functions* primarily use integer operations and do not utilize tensor cores. The compute throughput is calculated assuming 32 bit integer operations.

As can be seen from the roofline plots in Figure 13, the *scalar functions* that we target in this paper all have a very high arithmetic intensity [9] and our compiler is able to achieve good performance in terms of billions of instructions per second (GInsts/sec). Importantly note that the achieved performance is always well above the CPU roofline, indicating that, for such computations, the CPU would be at-least $3-5\times$ slower; even at peak theoretical performance. This provides more empirical evidence for why *scalar functions* are a good candidate for GPU acceleration.

The plot also explains the performance differences observed in Figure 12 (the roofline plot includes manual CUDA kernels) and Figure 11. In particular, note that ABS(DEC) and SQRT(INT) have a low arithmetic intensity and due to their memory-bound nature [10] only achieve a moderate performance improvement over CPU.

In addition to the roofline model, we look at other performance counters and report that the string functions only achieve a coalescing efficiency of 40−80% due to the data dependent memory accesses required during traversal. We also find that date functions sometimes suffer from branch divergence, due to data dependent control flow (leap year check). These observations explain why *scalar functions* still do not achieve peak compute throughput. These effects are inherent to the algorithms used in these functions and we plan to investigate this further in future.

6.6 Performance portability across GPU vendors

We further evaluate our compiler on AMD MI300 GPUs, a latest-generation accelerator with hardware characteristics that differ substantially from NVIDIA A100. For memory-bound functions like ABS(DEC) and SQRT, we observe the kernels are faster by 1.2–1.7× relative to A100, while compute-intensive kernels over DATE and STRING types are 1.7–2.5× faster.

These results demonstrate that the compiler achieves portable performance across GPU vendors by decoupling code generation from vendor-specific toolchains such as CUDA.

7 DISCUSSION ON PRACTICAL CONSIDERATIONS FOR GPU COMPILATION

This section discusses practical considerations that arise when compiling pre-existing code for GPUs and how they are addressed in the context of *scalar functions*. For additional perspective we discuss the applicability of the proposed approach to the related problem of accelerating UDFs.

7.1 Challenges for GPU compilation

Support for library functions. To generate GPU kernels, the compiler must translate all code used in the existing implementation to MLIR. Library functions without available source code can therefore present a challenge, but this has not been an obstacle for *scalar functions*. The implementation of *scalar function* relies

[9] Note that arithmetic intensity is an algorithmic, hardware-independent measure of the ratio of computations to memory accesses. While the realized arithmetic intensity—measured via profiling—can vary depending on the hardware, compiler, and toolchain, a well-optimized implementation should not deviate significantly from the theoretical limit imposed by the algorithm.

[10] We report that the benchmarks achieve a memory bandwidth of over 1.5TB/s. The reason it does not appear close to the memory bandwidth limit in the plot, is that the roofline model does not take into account the output bytes, which often accounts for half the memory operations for *scalar functions*. The points are indeed close to the memory limit if we assume only half the bandwidth (1 TB/s) is available for reads.

on standard C library functions for math operations and memory operations, many of which are implemented directly in assembly. However, MLIR has pre-exisiting dialects to support important classes of standard functions. In particular it has built-in *math*, *llvm* and *memref* dialects that support functions—such as *sin*, *log*, *exp*, *memcpy* and *alloca*—which are used in the implementation of *scalar functions*. The frontend Polygeist compiler has default support for translating such functions, it recognizes calls to these standard library functions and directly maps them to corresponding dialects.
Dynamic memory allocation. Another practical concern is dynamic memory allocation, while many GPUs support such allocation, it can introduce prohibitive overhead in GPU kernels [51]. Dynamic memory allocation within *scalar functions* is rare and managed by our toolchain based on specific scenarios. The main use case involves allocating output buffers for string functions. As noted in the paper, the GPU engine uses a specialized string representation—comprising bytes, offsets, and lengths—with these buffers pre-allocated on the GPU and write operations adapted accordingly. Other dynamic allocations are typically short-lived and are allocated on the stack. These are translated to *memref.alloca* operations, which are efficiently supported on GPUs. Note that as is common for many database engines, memory allocations in SQL Server are not strictly dynamic, the database system has its own memory manager that pre-allocates large buffer pools and sub-allocates from them as needed. The string handling in our GPU engine could be viewed as an extension of this idea to GPUs.

7.2 Beyond scalar functions

Many database engines offer extensibility through User-Defined Functions (UDFs), which can be written in languages such as Python, C/C++, or Java. C/C++ UDFs are recommended [30] for performance-critical workloads due to tighter runtime integration. We investigate UDFs from UDFBench [30], a benchmark suite with UDFs implemented in multiple languages. It includes 24 scalar UDFs to perform custom data cleaning, extraction and format/type conversion tasks. An investigation of the source code reveals that the benchmark is composed of two classes of functions. One class is easily supported by tool-chains like ours, this class includes twelve functions that do not make use of any library functions or dynamic memory allocation and one LOG10 that makes use of *Math* for which MLIR has existing support. Although UDFBench provides Python implementations for many functions, we translated them to C++ in a straight-forward manner, ran the generated kernels and validated our compiler. The remaining functions use library functions by importing Python libraries *re, json*, whose source code is not available. We note that several full source C++ libraries provide similar functionality, and if the UDFs were implemented using such libraries, our toolchain would generate GPU kernels for them. Database engines often provide guidelines for writing high-performance UDFs, and we believe that with appropriate guidelines focused on GPU compatibility (e.g., avoiding dynamic memory allocation and minimizing library dependencies), users can write UDFs that can be automatically accelerated using our toolchain.

8 RELATED WORK

GPU databases are an active area of research today, with primary focus being on running analytical queries on GPU hardware [4, 7, 24, 25, 27, 31, 34, 57]. Several different abstractions have been tried out for running relational operations on GPUs. HeavyDB [7] is a commercial GPU database that internally uses LLVM to generate code for query execution. BlazingDB [4] uses cuDF as its backend for query execution. Ocelot [36] adds OpenCL extensions to MonetDB [23] to support GPU execution. The tensor query processing system [34] builds upon the success of tensor based abstractions in deep learning to map relational operators to tensor operations. However, there is little discussion on support and evaluation of *scalar functions* in any of these systems. Among these we find that TQP is a natural fit for our work as it has similar portability goals as our compiler. A few papers in literature have looked into performance characteristics of GPU databases [26, 41, 57]. A recent study [41] finds that while interconnects still pose limits, GPU-bound execution is becoming more common as hardware improves. Recent papers propose mechanisms to overcome the memory capacity and interconnect limitations, including mechanisms to aggressively filter the compressed data on the CPU [46] and leveraging multiple GPUs for I/O routing, while executing queries on a single GPU [61]. Our work finds that *scalar functions* are inherently hard to evaluate and that queries that use them run significantly faster on GPUs even with PCIe transfers.

Next we discuss related compiler frameworks. Polygeist [49] is a frontend compiler designed to lift C++ code to MLIR dialects, with the goal of enabling polyheadral optimizations. We use it to lift to MLIR and introduce a new backend GPU code generator. MLIR based compilers [39] have also been used to re-optimize CUDA kernels, with optimizations like thread coarsening (at a different abstraction than ours). However, they do not generate wide-word instructions. In the database context, there is extensive literature on JIT query compilation [7, 32, 40, 50], including systems that introduce custom IRs prior to code generation. Our work highlights a new application of compilation techniques to data systems.

Finally, we note parallels between *scalar functions* and user-defined functions (UDFs) (also see Section 7.2). Scalars functions are treated as first-class operators during query optimization, compilation, and execution. On the other-hand, UDFs are treated as black boxes, leading to optimization and execution bottlenecks - such as out-of-process invocation, row-at-a-time processing, and limited parallelism [30]. Prior work addresses these by translating UDFs into relational algebra [20, 55, 58], including via cost-based transformation [20]. UDFBench [30] recommends JIT compilation as an orthogonal technique for improved performance. With appropriate interface design, our techniques can similarly enable portable UDF performance across hardware backends.

9 CONCLUSIONS

This paper identifies that database scalar functions have high computational overhead and are good candidates for GPU acceleration. It presents a novel compiler toolchain to generate efficient GPU code while preserving the semantics of an existing implementation, and contributes a new benchmark suite for future evaluation of scalar functions. In future we plan to expand the usage of the compiler to accelerate other classes of element-wise data transformation operators including UDFs. We also plan to investigate algorithmic improvements that could enable portable performance of *scalar functions* across multiple hardware.

REFERENCES

[1] [n. d.]. https://learn.microsoft.com/en-us/sql/t-sql/statements/sql-server-collation-name-transact-sql?view=fabric. [Accessed June 2025].

[2] [n. d.]. AMDGPU Dialect in MLIR. https://mlir.llvm.org/docs/Dialects/AMDGPU/.

[3] [n. d.]. BigQuery Docs: Functions and Operators. https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators. Accessed: June 2025.

[4] [n. d.]. BlazingSQL. https://github.com/BlazingDB/blazingsql.

[5] [n. d.]. DuckDB Docs: Functions and Operators. https://duckdb.org/docs/sql/functions.html. Accessed: June 2025.

[6] [n. d.]. EPYC 9004 series processors data sheet. https://www.amd.com/content/dam/amd/en/documents/products/epyc/epyc-9004-series-processors-data-sheet.pdf. Accessed: June 2025.

[7] [n. d.]. HeavyDB. https://www.heavy.ai/product/heavydb.

[8] [n. d.]. Microsoft Docs: T-SQL Functions. https://learn.microsoft.com/sql/t-sql/functions. Accessed: June 2025.

[9] [n. d.]. MLIR in TensorFlow. https://www.tensorflow.org/mlir.

[10] [n. d.]. NVGPU Dialect in MLIR. https://mlir.llvm.org/docs/Dialects/NVGPU/.

[11] [n. d.]. Oracle Docs: About SQL Functions. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/About-SQL-Functions.html. Accessed: June 2025.

[12] [n. d.]. PostgreSQL Docs: Functions and Operators. https://www.postgresql.org/docs/current/functions.html. Accessed: June 2025.

[13] [n. d.]. RAPIDS Accelerator For Apache Spark. https://github.com/NVIDIA/spark-rapids.

[14] [n. d.]. Snowflake Docs: Functions. https://docs.snowflake.com/en/sql-reference/functions-all.html. Accessed: June 2025.

[15] [n. d.]. Sophgo TPU. https://www.sophgo.com/en/products/tpu.

[16] [n. d.]. Torch-MLIR. https://github.com/llvm/torch-mlir.

[17] [n. d.]. Triton: A Language and Compiler for Efficient Deep Learning. https://triton-lang.org.

[18] [n. d.]. XeGPU Dialect in MLIR. https://mlir.llvm.org/docs/Dialects/XeGPU/.

[19] 2023. TPC-H Benchmark Specification. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.

[20] Samuel Arch, Yuchen Liu, Todd C. Mowry, Jignesh M. Patel, and Andrew Pavlo. 2024. The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining. *Proceedings of the VLDB Endowment* 18, 1 (2024), 1–13. https://doi.org/10.14778/3696435.3696436

[21] Microsoft Azure. 2025. Azure Virtual Machines Pricing. https://azure.microsoft.com/pricing/details/virtual-machines/windows. [Accessed June 2025].

[22] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (Sept. 2022), 25 pages. https://doi.org/10.1145/3544559

[23] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[24] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (2014), 199–209.

[25] Sebastian Breß and Gunter Saake. 2013. Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1398–1403. https://doi.org/10.14778/2536274.2536325

[26] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[27] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[28] Microsoft Fabric Community. 2024. The Importance of New Case-Insensitive Collation in Fabric. https://community.fabric.microsoft.com/t5/Data-Warehouse-Community-Blog/The-Importance-of-New-Case-Insensitive-Collation-in-Fabric/ba-p/4289617. [Accessed June 2025].

[29] Harish Doraiswamy, Vikas Kalagi, Karthik Ramachandra, and Jayant R. Haritsa. 2023. A Case for Graphics-Driven Query Processing. *Proc. VLDB Endow.* 16, 10 (June 2023), 2499–2511. https://doi.org/10.14778/3603581.3603590

[30] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. The UDFBENCH Benchmark for General-Purpose UDF Queries. *Proc. VLDB Endow.* 18, 9 (Sept. 2025), 2804–2817. https://doi.org/10.14778/3746405.3746409

[31] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[32] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. 2023. Bringing Compiling Databases to RISC Architectures. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1222–1234. https://doi.org/10.14778/3583140.3583142

[33] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[34] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[35] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 889–900. https://doi.org/10.14778/2536206.2536216

[36] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[37] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (March 2023), 81 pages. https://doi.org/10.1145/3570638

[38] Matteo Interlandi, Nicolas Bruno, Brandon Haynes, Carlo Curino, Rathijit Sen, Yinan Li, Kaushik Rajan, Bailu Ding, Lukas Maas, Wei Tsui, Kevin Gaffney, Mingsheng Hong, Brian Kroth, Sampath Rajenda, Peng Cheng, Surajit Chaudhuri, Johannes Gehrke, Raghu Ramakrishnan, Lidong Zhou, Momin Al-Ghosien, Craig Peeper, Marius Dumitru, Conor Cunningham, Kevin Bocksrocker, Prasanna Sundar, Ron Boskovic, YongChul Kwon, Marko Zivanovic, Runbin Shi, Krystian Sakowski, Denis Lamtsov, Josep Aguilar Saborit, Krish Srinivasan, Adarsh Kapil, Andrew Putnam, Anudeep Kambapu, Aparna Konduri, Aaron Landy, Aaron Moore, Aaron Pitman, Artem Oks, Ashit Gosalia, Babu Kandimalla, Blake Pelton, Bogdan Crivat, Cesar Galindo-Legaria, Chidamber Kulkarni, Esteban Calvo, Evgeny Babin, Hans Lehnert Marino, Igor Konev, Israel Arroyo, Jesus Camacho Rodriguez, Luis Penaranda, Mandar Datar, Morten Borup Petersen, Nicholas Simmons, Parminder Poonian, Pravija Danda, Rob Rydberg, Ronak Bajaj, Sai Sumanth Kammal Shetty, Sharanya Bhat, and Zach Iracheta. 2026. CoddSpeed: Hardware Accelerated Query Processing in Microsoft Fabric. In *Proceedings of the 2026 International Conference on Management of Data.* To Appear at SIGMOD 2026 (Industry Track).

[39] Ivan R. Ivanov, Oleksandr Zinenko, Jens Domke, Toshio Endo, and William S. Moses. 2024. Retargeting and Respecializing GPU Workloads for Performance Portability. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization* (Edinburgh, United Kingdom) *(CGO '24)*. IEEE Press, 119–132. https://doi.org/10.1109/CGO57630.2024.10444828

[40] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[41] Marko Kabić, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4350–4363. https://doi.org/10.14778/3749646.3749698

[42] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[43] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-based code generation for GPU tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/3497776.3517770

[44] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *Proc. VLDB Endow.* 14, 10 (2021), 1797–1804. https://doi.org/10.14778/3467861.3467869

[45] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[46] 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, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (2025), 4518–4531. https://doi.org/10.14778/3749646.3749710

[47] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards

a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1951–1960. https://doi.org/10.1145/2882903.2903735

[48] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist GitHub Repository. https://github.com/llvm/Polygeist. [Accessed June 2025].

[49] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[50] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[51] NVIDIA. [n. d.]. GPU Performance Background: User's Guide. https://docs.nvidia.com/deeplearning/performance/pdf/GPU-Performance-Background-User-Guide.pdf. Accessed: June 2025.

[52] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[53] Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R Govindarajan, and Uday Bondhugula. 2024. SilvanForge: A Schedule-Guided Retargetable Compiler for Decision Tree Inference. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3694715.3695958

[54] ProteusDB. 2025. ProteusDB. https://proteusdb.com/. [Accessed June 2025].

[55] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, Cesar Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444. https://doi.org/10.1145/3164135.3164140

[56] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[57] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[58] Tarique Siddiqui, Arnd Christian König, Jiashen Cao, Cong Yan, and Shuvendu Lahiri. 2025. QURE: AI-assisted and Automatically Verified UDF Inlining. *Proceedings of the ACM on Management of Data (SIGMOD)* 3 (February 2025). https://www.microsoft.com/en-us/research/publication/qure/

[59] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[60] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM Manag. Data* 1, 1, Article 29 (May 2023), 26 pages. https://doi.org/10.1145/3588709

[61] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2024. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. In *51st International Conference on Very Large Databases (VLDB 2025)*. ACM.

[62] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210