TCUDB: Accelerating Database with Tensor Processors

Yu-Ching Hu
University of California, Riverside
Riverside, California, USA
yhu130@ucr.edu

Yuliang Li
Megagon Labs
Mountain View, California, USA
yuliang@megagon.ai

Hung-Wei Tseng
University of California, Riverside
Riverside, California, USA
htseng@ucr.edu

ABSTRACT

The emergence of novel hardware accelerators has powered the tremendous growth of machine learning in recent years. These accelerators deliver incomparable performance gains in processing high-volume matrix operators, particularly matrix multiplication, a core component of neural network training and inference. In this work, we explored opportunities of accelerating database systems using NVIDIA's Tensor Core Units (TCUs). We present TCUDB, a TCU-accelerated query engine processing a set of query operators including natural joins and group-by aggregates as matrix operators within TCUs. Matrix multiplication was considered inefficient in the past; however, this strategy has remained largely unexplored in conventional GPU-based databases, which primarily rely on vector or scalar processing. We demonstrate the significant performance gain of TCUDB in a range of real-world applications including entity matching, graph query processing, and matrix-based data analytics. TCUDB achieves up to 288× speedup compared to a baseline GPU-based query engine.

CCS CONCEPTS

• **Information systems** → **Relational database model**; **DBMS engine architectures**; **Query optimization**; **Query operators**; **Query planning**; **Join algorithms**; • **Hardware** → **Hardware accelerators**.

KEYWORDS

Tensor Cores, database engine

ACM Reference Format:
Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517869

1 INTRODUCTION

The enormous demand for artificial intelligence (AI) and machine learning (ML) workloads has driven the development and integration of accelerators containing instructions operating on two-dimensional tensors (i.e., matrices). Examples include NVIDIA's Tensor Core Units (TCUs) [60], Google's Tensor Processing Units (TPUs) [75], and Apple's Neural Processing Units (NPUs) [6]. Improving matrix algebra through matrix units (MXUs), which popular AI/ML models heavily rely on, drastically increases the orders of magnitude speedup and energy efficiency. This is particularly true when compared with conventional scalar processors (e.g., CPUs) and vector processors (e.g., graphical processing units [GPUs]).

In this work, we explore opportunities of integrating Tensor Core Units (TCUs) into a database engine's architecture. Despite being originally designed for AI/ML workloads, tensor processors also hold potential performance improvements for database engines. This is due to both the increasing demand for native support of linear algebra queries (e.g., matrix multiplication itself) in SQL DB engines [3, 24, 26, 39, 57] and the observation that a large number of regular query operators can be cast into matrix multiplication. For example, one can show that the most commonly used natural joins [5, 20] and group-by aggregates can be encoded as matrix multiplication, which enables TCUs to deliver exceptional performance.

However, the presence of these AI/ML accelerators, or more generally matrix processors, does not provide a drop-in upgrade to the query engine's performance. Three major challenges must be addressed.

Challenges. First, the conventional GPU databases primarily implement the physical operators (e.g., the partitioned hash join algorithm [46]) in a non-matrix-friendly manner. These algorithms and operators typically do not operate on tensors directly. As a result, it is hard to modify them with the intent of taking advantage of TCUs' computation power.

Second, although DB operators such as joins can theoretically be encoded as matrix multiplications, executing all of them as dense multiplication might not always be beneficial. For example, the underlying data distributions can cause the two operands to be sparse matrices, which require a different data organization and APIs to achieve the best performance.

Next, a DB engine with TCUs must prevent itself from generating erroneous query results because of the low-precision nature of the tensor processors. The current tensor processors are limited in precision as AI/ML applications are error-tolerant because NVIDIA's TCUs only support 16-bit floating-point numbers while Google's TPUs only work on at most 8-bit integers. Moreover, these tensor processors share the same data movement overhead with other hardware accelerators while additionally suffering from the data transformation overhead (i.e., table → tensor). A higher precision requirement means introducing more data movement and transformation overhead. As a result, the proposed system must maintain a balance between two factors.

TCUDB. This paper presents TCUDB, an analytic database query engine that explores the potential of tensor processors to accelerate analytic query workloads using TCUs by tackling the aforementioned challenges. Figure 1 provides an overview of the system architecture of TCUDB. TCUDB extends the common architecture of GPU-accelerated databases [13, 34, 54, 72, 76, 83, 87, 89, 90, 92] as a way to further accommodate executing query operators with TCU acceleration in the query analyzer, the query optimizer, the code generator, and the program driver.

To address the challenge of executing queries using matrix operations, we re-engineered a set of query operators that are theoretically feasible to be mapped to tensor/matrix algebra operations for TCUDB. The query operators cover a large set of commonly used ones including natural joins and group-by aggregates. As shown in Figure 1, TCUDB features a code generator for generating executable code mapping input tables to tensor format and processes the query as matrix multiplication via WMMA or cuBLAS API calls. Depending on the data sparsity, TCUDB provides the

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517869

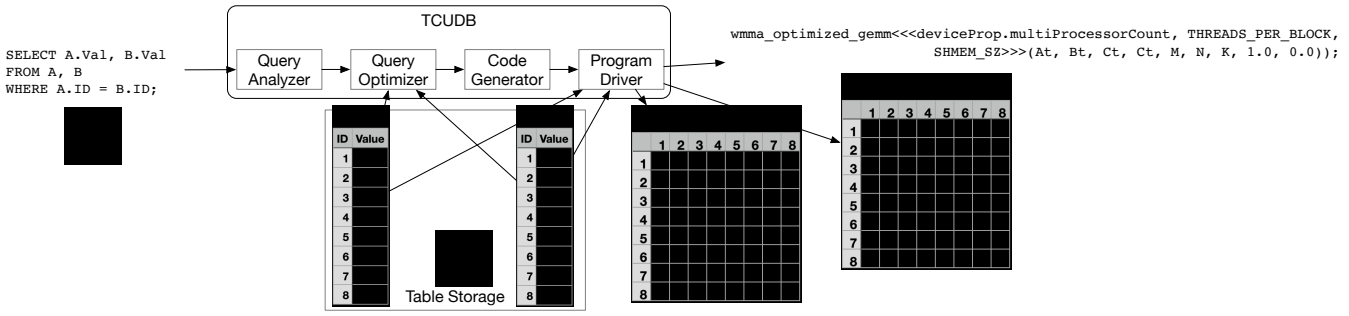


Figure 1: An overview of TCUDB's workflow.

option of sparse tensor encoding with sparse matrix multiplication. We developed the TCU-SpMM operator to support sparse matrix multiplication with TCU acceleration. Then, the TCUDB query analyzer is capable of generating query plans, which use these TCU-accelerated physical operators.

To resolve the challenge of limited precision and overhead in modern tensor processors, TCUDB's query optimizer carefully gauges the parameters in precision, data movement overhead, data transformation overhead, and computation throughput — as using lower data precision yields lower data movement overhead and higher computation throughput, but also takes higher risks of leading into unacceptable answers as well as higher data transformation overhead. TCUDB presents an adaptive mixed-precision query optimization that dynamically selects the most appropriate precision in delivering the desired level of accuracy using the shortest end-to-end latency to handle queries.

Contributions. By presenting, implementing and evaluating TCUDB, this paper makes the following contributions:

- We explored the space of opportunities of optimizing a GPU-accelerated analytic query engine by leveraging TCUs. In our initial investigation, we found that TCU delivers >5× performance gains for matrix multiplication compared to the conventional CUDA cores in GPUs. This finding contradicts the conventional wisdom that considers matrix multiplication a slow operator because of its high computational complexity. As such, TCUs provide new opportunities to optimize processing analytic queries as matrix multiplication.
- Next, we identified a collection of query patterns that can potentially be accelerated by TCUs. The query patterns include the most commonly used SQL operators in analytic queries such as joins and group-by aggregates (e.g., `SUM` and `COUNT`). We demonstrate simple algorithms for transforming relational tables into matrix format and translating SQL operator into one or more matrix multiplication operators. Our algorithmic design is generic as it can be generalized to multi-way joins and aggregation over joins.
- We designed and implemented TCUDB, a TCU-accelerated analytic database engine. On top of a traditional GPU database [1], TCUDB features a query optimizer that identifies (1) the most efficient TCU query plan and (2) the best GPU/CPU-based plan and decides which plan to execute via cost estimation. If a TCU-accelerated plan is selected, TCUDB leverages a code generator to rewrite (parts of) the query into C programs that invoke NVIDIA's CUDA API. To the best of our knowledge, TCUDB is the first analytic database engine with TCU-accelerated built-in.
- We evaluated TCUDB on 4 real-world use cases: (1) linear algebra (LA) queries, (2) entity matching (EM), (3) graph analytics, and (4) analytic queries such as the star-schema benchmark. TCUDB demonstrates an outstanding performance advantage over a GPU-based engine (YDB), by achieving up to 288× speedup. Our results also highlight the necessity of the query optimizer and TCUDB's scalability advantage in future GPU architecture.

[1]We archive the source code and workloads at our GitHub page: https://github.com/escalab/TCUDB

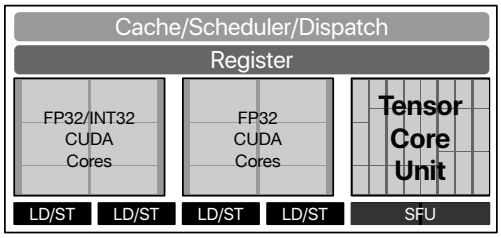


Figure 2: The GA102 Streaming Multiprocessor (SM) architecture in GeForce RTX 30-series GPUs.

2 BACKGROUND AND MOTIVATION

This section describes the background of the conventional query processing on a GPU and the motivation inspired by the characteristic of Tensor Core Units (TCUs). By comparing to the traditional vector processing model, we demonstrate the tensor processing model in a database system that can deliver better performance on linear algebra queries in terms of computing capability and scalability.

2.1 Tensor Core Units (TCUs)

As deep neural networks heavily rely on operations using matrix multiplications (e.g., convolution), recent hardware accelerators feature matrix units (MXUs) in their microarchitectures to significantly boost the performance in machine learning (ML) workloads. Famous examples include NVIDIA's Tensor Core Units (TCUs), Google's Tensor Processing Units (TPUs), and Apple's Neural Engine.

This paper selects TCUs as the underlying accelerators for the following reasons: (1) Programmability: TCUs expose their low-level C++ API to programmers such as highly optimized cuBLAS APIs or customizable WMMA (Warp Matrix Multiply-Accumulate) APIs, giving programmers complete freedom in implementing algorithms and integrating with existing systems. By contrast, their counterparts are only programmable through domain-specific languages tailored for ML. (2) Accessibility: TCUs are now standardized components in NVIDIA's GPU architectures, ranging from high-end server solutions, gaming solutions, to embedded solutions. Conversely, high-performance TPUs are only accessible through Google's cloud services and Apple's NPUs are only available on their machines. (3) Flexibility: Tensor cores together with other

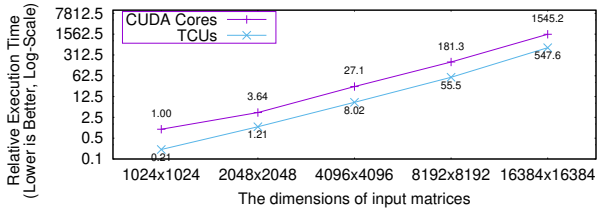


Figure 3: The performance of performing matrix multiplications using conventional CUDA cores and TCUs.

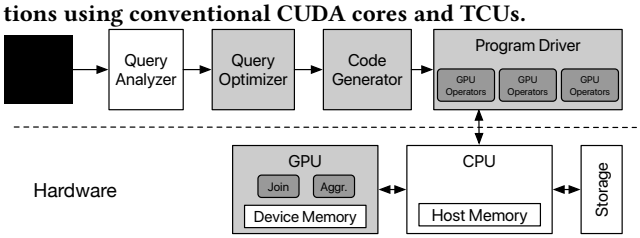


Figure 4: Typical GPU-accelerated database architecture.

ALUs on the GPU supports multiple data precision with various operations. Other ML accelerators only support limited precision.

TCUs are currently available as separated functional units from conventional vector floating-point and integer ALUs within the current generation of streaming multiprocessors (SM) as Figure 2 depicts. Figure 3 compares the latency of multiplying matrices with different sizes, ranging from 1024×1024 inputs matrices to 16384×16384 ones, using conventional vector processing units (CUDA cores) and TCUs, on NVIDIA's RTX 3090 GPU. The results show that TCUs consistently outperform CUDA cores by up to a 5× speedup. By translating the latency to TFLOPs, we measured a peak of 63 TFLOPs on TCUs and 19 TFLOPs using mixed precision on CUDA cores only.

Despite the significant speedup in matrix operations, TCUs still have limited precision drawbacks seen in other AI/ML accelerators in a way that TCUs only support at most 16-bit numbers as inputs and incur additional overhead in casting data into the desired 16-bit formats. Being separated functional units within an SM and the nature that an SM can only perform a single type of operations simultaneously, a compute kernel can activate either conventional vector units or TCUs, but not both of them due to the power constraints and the hardware architecture. Therefore, if programmers do not specifically enable TCUs and rewrite algorithms to perform matrix multiplications, a GPU program cannot automatically take advantage of TCUs. Instead, it wastes the rich speedup that the TCUs can provide.

2.2 GPU-accelerated Database System Architecture (GPUDB)

Prior to the introduction of TCUs in GPU architectures, database systems have exploited the potential of using the massive amount of vector processing units within GPUs to accelerate query processing [8, 13, 82, 92]. The rich thread-level parallelism from these vector processing units delivers better performance on easily parallelizable operations (e.g., arithmetic computation). Figure 4 shows the architecture of a typical GPUDB system that Yinyang DB (YDB) [7, 92] and GPUQP [36] adopt. Upon receiving a query, the GPU-accelerated DB will go through the following stages: (1) Query plan generation: the query parser translates SQL query into query plan tree and the query optimizer analyzes the costs and benefits of query plans to determine the most efficient implementation (i.e., the cheapest plan) as the physical query plan. (2) Code generation: the query engine is in charge of the query execution flow by generating

```
-- Matrix multiplication query:
SELECT A.col_num, B.row_num, SUM(A.val * B.val) as res
FROM A, B
WHERE A.row_num = B.col_num
GROUP BY A.col_num, B.row_num;
```

Figure 5: Example matrix multiplication query.

the back-end system-level code (e.g., program driver) that maps the selected query plan to utilize CPU and GPU cores. According to the type of target queries, different GPU kernels are implemented to execute relational database operators. (3) Data movements: data movements involve loading table data to the host main memory from back-end storage, moving essential data from the host main memory to GPU device memory and copying results back to the host main memory.

In the aforementioned database system architecture, data movement between GPU and CPU usually dominates the execution time [36] and cancels out the performance gain in the computation part. Therefore, GPU database architecture should make full use of an in-memory technique such as keeping all tables in GPU RAM [31] to mitigate the I/O bottleneck. There is no common-use GPU algorithm suitable for all database systems; the challenge is to identify which operators can leverage the GPU and combine it with traditional database query processing. Additionally, the data storage format also affects the performance of data movement. Due to the GPU memory access pattern, column-store [1, 2, 31] helps to exploit coalesced memory as well as reduce data volume going through the PCIe bus by only sending the needed data.

2.3 The Missing Opportunities of GPU Databases in TCUs

Before the emergence of TCUs, conventional wisdom assumed that matrix multiplication is an inefficient operation. Therefore, state-of-the-art GPUDB systems are designed in favor of vector processing, yet completely avoid the usage of matrix multiplications. Without redesigning application algorithms and data layout, existing GPUDB systems cannot reap the benefits of TCUs.

The query in Figure 5 provides an example of how an existing GPUDB misses the potential of using TCUs. The result of this query is essentially a list of triples of (row_num, col_num, val) with unique combinations of row_num, col_num and the val in each triple is the sum of the pairwise multiplications on val fields from a record in table A with its row_num matching another record's col_num from table B. This is essentially an SQL query that performs matrix multiplication on elements from two tables A and B. This query can be implemented through one matrix multiplication if we can layout the matching elements in matrices appropriately.

However, conventional GPUDB query processing algorithms are designed at the operator level with each operator as a kernel function running on GPUs. To execute the above query, conventional GPUDB uses operators to build hash tables for A and B, scanning both tables, performing `HashJoin`, and aggregating the final result. Among these GPU operators, `HashJoin` where performs join operation in a pairwise, vectorized fashion to find matching tuples between two hash tables usually takes the most time during the query execution. The aggregation operator is second to `HashJoin`, which is also time-consuming in accumulating the computation result using vector operations. As the above computation only requires vector inner-products, the generated GPU kernel code will never enable TCUs.

3 TCU-ACCELERATED QUERY PATTERNS

As mentioned above, TCUs can potentially improve the performance of an analytic query by executing (parts of) the query as matrix multiplication. Next, to achieve this goal, we start by identifying a number of query patterns that TCUDB can execute as matrix multiplications.

3.1 Two-way natural join

The first supported query pattern is the simple 2-way join. For example, given two tables A and B with two attributes (ID, Val), consider the following query:

```
-- Q1:
SELECT A.Val, B.Val
FROM A, B
WHERE A.ID = B.ID;
```

To process this query as a matrix operation, we first need to convert the two tables into a matrix format. Suppose table A contains n tuples $\{a_1, \ldots, a_n\}$ and table B contains m tuples $\{b_1, \ldots, b_m\}$ where each a_i and b_i are unique row IDs. Let dom(A.ID) and dom(B.ID) be the domains of the ID column of A and B respectively. Let dom(ID) to be the union of the two domains dom(A.ID) ∪ dom(B.ID) having k distinct values $\{v_1, \ldots, v_k\}$. To compute the join, we construct a $n \times k$ matrix mat(A) and a $m \times k$ matrix mat(B) where

$$\mathtt{mat(A)}_{ij} = 1 \text{ if } a_i.\mathtt{ID} = v_j, \text{ otherwise } 0 \ ;$$

$$\mathtt{mat(B)}_{ij} = 1 \text{ if } b_i.\mathtt{ID} = v_j, \text{ otherwise } 0 \ .$$

The result of the join A ⋈ B is then the n by m matrix

$$\mathtt{C} = \mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}.$$

It is easy to show that a tuple (a_i, b_j) is in the join result if and only if $\mathtt{C}_{ij} > 0$.

Alternatively, when the domains dom(A.Val) and dom(B.Val) are small, one can also construct mat(A) and mat(B) as the adjacency matrices where $\mathtt{mat(A)}_{ij} = 1$ if $(u_i, v_j) \in$ A (and respectively for mat(B)) otherwise 0. The number of rows of mat(A) and mat(B) will be |dom(A.Val)| and |dom(B.Val)| respectively.

Note that in this query pattern, the single attributes A.ID, A.Val, B.ID and B.Val can be generalized to sets of multiple attributes. The attribute sets ∗.ID and ∗.Val can potentially overlap thus it is general enough to cover all cases of 2-way natural join.

3.2 Multi-way joins

Next, we extend the querying capability with matrix multiplication to multi-way joins. Consider the following snippet of a 3-way join query where the 3 input tables are $\mathtt{A}(\mathtt{ID}_1, \mathtt{Val})$, $\mathtt{B}(\mathtt{ID}_1, \mathtt{ID}_2, \mathtt{Val})$, and $\mathtt{C}(\mathtt{ID}_2, \mathtt{Val})$ respectively.

```
-- Q2:
SELECT A.Val, B.Val, C.Val
FROM A, B, C
WHERE A.ID_1 = B.ID_1 AND B.ID_2 = C.ID_2;
```

As in conventional join processing, we assume a join order of A → B → C. To evaluate this join, one needs to (1) first compute A ⋈ B as $\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}$, (2) convert the resulting n by m matrix back to table format and (3) compute the join with table C as a second matrix operator. By repeating step (2) and (3) to convert intermediate results to tables, we can generalize this algorithm from 3-way joins to multi-way joins.

To avoid unnecessary data transfer from GPU memory to the host, in step (2), one can perform the matrix-table conversion with a CUDA-enabled nonzero(·) operator [71]. Formally, given a matrix M, nonzero(M) computes $\{(i, j) | M_{ij} > 0\}$. Next, to perform the second join, let

- n' be the size of $\mathtt{nz} = \mathtt{nonzero}(\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}})$,
- m' be the size of table $\mathtt{C} = \{c_1, \ldots, c_{m'}\}$ and
- k' be the size of $\mathtt{dom}(\mathtt{B.ID}_2) \cup \mathtt{dom}(\mathtt{C.ID}_2) = \{u_1, \ldots, u_{k'}\}$.

We denote by $\mathtt{nz}_i$ the i-th pair of the nz array. Next, we construct a n' by k' matrix mat(AB) and a m' by k' matrix mat(C) where

$$\mathtt{mat(AB)}_{ij} = 1 \text{ if } b_{i'}.\mathtt{ID}_2 = u_j \text{ for } \mathtt{nz}_i = (\_, i'), \text{ otherwise } 0;$$

$$\mathtt{mat(C)}_{ij} = 1 \text{ if } c_i.\mathtt{ID}_2 = u_j, \text{ otherwise } 0.$$

The result of the 3-way join is then $\mathtt{mat(AB)} \times \mathtt{mat(C)}^{\mathsf{T}}$.

There is an exception case where the intermediate matrix-table conversion can be omitted. When B.Val = ∅ (i.e., relation B is projected out entirely), the result of the join can be simplified as

$$\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}} \times \mathtt{mat(C)}^{\mathsf{T}}$$

where mat(B) is a k by k' matrix constructed as $\mathtt{B}_{ij} = 1$ if $(v_i, u_j) \in$ B otherwise 0.

Similar to the 2-way join case, the method can be generalized to multi-way joins consisting of multiple join and/or return attributes.

3.3 Group-by aggregates over joins

A simple yet useful extension of the above two query patterns with joins is to add group-by aggregates. For example, over the same schema (ID, Val) of the previous 2-way join case:

```
-- Q3:
SELECT SUM(A.Val), B.Val
FROM A, B
WHERE A.ID = B.ID
GROUP BY B.Val;
```

A naive method to evaluate this query is to first evaluate the natural join in the TCU-optimized manner, convert the matrix result to the table format, and then compute the group-by and SUM aggregate with CPU or GPU-based methods. We propose the following method that avoids any unnecessary intermediate computation via 2 matrix operations. First, we construct the two input matrices. For the matrix dimensions, we let

- n be the size of A,
- m be the size of $\mathtt{dom}(\mathtt{B.Val}) = \{u_1, \ldots, u_m\}$, and
- k be the size of $\mathtt{dom}(\mathtt{A.ID}) \cup \mathtt{dom}(\mathtt{B.ID}) = \{v_1, \ldots, v_k\}$.

We construct a n by k matrix mat(A) and a m by k matrix where

$$\mathtt{mat(A)}_{ij} = a_i.\mathtt{Val} \text{ if } a_i.\mathtt{ID} = v_j, \text{ otherwise } 0;$$

$$\mathtt{mat(B)}_{ij} = 1 \text{ if } (u_i, v_j) \in \mathtt{B}, \text{ otherwise } 0.$$

Next, the query result can be computed as

$$\mathbf{1}^{1 \times n} \times \mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}$$

where $\mathbf{1}^{1 \times n}$ is an $1 \times n$ matrix consisting of only ones. We can show the following:

Lemma 3.1. *(Q3, informal) For every tuple $(a_i^{\mathsf{sum}}, b_i)$ and for $M = \mathbf{1}^{1 \times n} \times \mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}$, $(a_i^{\mathsf{sum}}, b_i)$ is in the query result of Q3 if and only if $M_{i,1} = a_i^{\mathsf{sum}}$.*

Intuitively, we leverage the first multiplication with $\mathtt{mat(B)}^{\mathsf{T}}$ to compute the join. By filling the input matrices mat(A) with actual values instead of 0's or 1's, we keep track of those values in the intermediate matrix product $\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}$. The multiplication with $\mathbf{1}^{1 \times n}$ then serves as a reduction operator that sums up all columns of $\mathtt{mat(A)} \times \mathtt{mat(B)}^{\mathsf{T}}$.

In addition to SUM, we are able to apply the same method to support the COUNT and AVG aggregate functions. For COUNT, when we construct mat(A), we simply need to set $\mathtt{mat(A)}_{ij}$ to 1 for $a_i.\mathtt{ID} = v_j$ (instead of $a_i.\mathtt{Val}$). We can obtain AVG by dividing SUM by COUNT.

For aggregate queries without GROUP BY, such as

```
-- Q4:
SELECT SUM(A.Val * B.Val)
FROM A, B
WHERE A.ID = B.ID;
```

we set $\texttt{mat(A)}_{ij} = a_i.\texttt{Val}$ for $a_i.\texttt{ID} = v_j$ and $\texttt{mat(B)}_{ij} = b_i.\texttt{Val}$ for $b_i.\texttt{ID} = v_j$ and compute the sum as $\texttt{mat(A)} \times \texttt{mat(B)}^{\top} \times \mathbf{1}^{m\times 1}$ with an additional reduction by multiplying $\mathbf{1}^{1\times n}$.

3.4 Other supported operators

The above query patterns can also be extended with the ORDER BY clause to sort the results in ASC/DESC order by a certain column. Instead of sorting after the multiplication operators, we preserved the specified order in the input matrices (e.g., mat(A) and mat(B)) so that the result matrix is naturally sorted.

Another class of supported query pattern is the non-equi join such as:

```
-- Q5:
SELECT A.Val, B.Val
FROM A, B
WHERE A.ID < B.ID;
```

We can compute this query by slightly adjusting the translation for Q1 by setting $\texttt{mat(A)}_{ij} = 1$ for $a_i.\texttt{Val} < v_j$. The same method applies to the other comparison operators $\{<, >, \leq, \geq, \neq\}$.

Last but not least, for the query pattern that is of the semantics of matrix multiplication as Figure 5 shows, we can directly map the query to the corresponding matrix operation.

Beyond the supported patterns. For queries that do not match exactly with any of the supported query patterns, as part of the query optimization workflow (Figure 6), TCUDB relies on pattern matching to identify subqueries that can be TCU-accelerated from the input query's AST. We note that there are common subqueries that are beyond the expressiveness of the TCU platform, such as aggregation with MIN/MAX or arithmetic operators such as addition and division. The limited expressiveness is mainly due to NVIDIA's current TCU programming interface which only supports matrix multiply-accumulate. However, since the underlying hardware is powerful enough to perform the aforementioned operators, we anticipate a more flexible programming interface in the future so that TCUDB can support a wide range of query patterns.

4 TCUDB: A TCU-ACCELERATED DB ENGINE

To leverage TCUs for queries in relational database systems, this paper presents TCUDB, a DB engine that identifies, optimizes, evaluates and implements aforementioned query patterns in Section 3. This section provides an overview of the design of TCUDB's extensions and discusses the optimizations on a TCU-accelerated query plan.

4.1 Overview

TCUDB implements the system architecture in Figure 1 to execute queries on TCUs using the following major components.

Query Optimizer In a system with TCUs presented, the query plan in exercising a query is from either (1) the most efficient TCU-accelerated query plan or (2) the most efficient conventional CPU/GPU-based plan, depending on which one can deliver the lowest cost (i.e., the shortest end-to-end latency). TCUDB leverages existing infrastructure in GPUDB to evaluate the second option but extends the query optimizer in creating, optimizing and evaluating the latency of TCU-accelerated query plans.

Program Driver TCUDB extends the program driver to additionally contain a set of library functions that implement operators mentioned in Section 3 using TCUs. These functions invoke NVIDIA's CUDA C++ Warp Matrix Multiply and Accumulate (WMMA) or cuBLAS API functions to achieve the series of computation that each operator requires. These operators also present interfaces in various data types to support the demand for the most efficient query plan.

Code Generator If TCUDB selects a TCU-accelerated query plan to exercise an incoming query, the code generator will rewrite the query as C code and dynamically compile the code to execute the selected query plan. The TCUDB code extension is responsible for creating the input matrices, calling operator functions in corresponding data types and remapping the output from the operator outcome.

Among these three intensively extended modules, the query optimizer is the most critical component as it serves as the core controlling the use of TCUs as well as code generation for queries. In the rest of this section, we will focus on the query optimizer.

4.2 TCUDB query optimizer

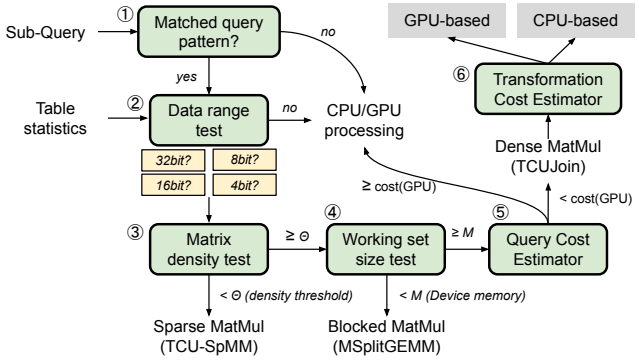


Figure 6: The workflow of the TCUDB query optimizer.

Figure 6 shows the workflow of the TCUDB query optimizer. The optimizer takes a subquery from the query AST as input and performs a series of tests to determine whether the subquery should be executed with TCU and how. The optimizer first checks if the subquery falls in one of the supported query patterns. Next, it performs the data range feasibility test (Section 4.2.1) to decide if particular data types can provide sufficient precision to the query. After that, the input tables may also result in matrices too large to fit in the GPU's device memory or sparse matrices for which dense multiplication algorithm is sub-optimal. For these cases, the optimizer estimates the working set sizes and matrix density from statistics pre-computed from input tables. TCUDB applies blocked matrix multiplication (MSplitGEMM, Section 4.2.3) and sparse matrix multiplication (TCU-SpMM, Section 4.2.4) respectively. Finally, the optimizer estimates the query execution cost with TCU and tests whether the cost is lower than the estimated cost with CPU/GPU (Section 4.2.2). If any of the tests fail, TCUDB falls back to the standard CPU or GPU-based query execution.

Note that the query cost estimator needs to take into account the data transformation cost which consists of both computation and data movement overhead. If the original table size plus the working set size fits in the device memory, TCUDB can transform tables into matrix format within GPU to save the overhead of transforming data within CPU and moving large matrices into the GPU device.

4.2.1 Feasibility Test. Even though a query contains patterns matching identified patterns in Section 3, a query may still be unfeasible for TCUs due to the limitations of TCUs in input precision and data types. If applying TCUs would result in loss of precision or lead to unwanted outcomes, TCUDB should not use TCUs to evaluate the incoming query.

Therefore, TCUDB must perform a feasibility test for each query that contains qualified patterns by evaluating the input data ranges, identifying the most compact inputs/outputs data types and estimating the working set sizes for operators within a query. To facilitate this process, TCUDB adds metadata to each database table to contain three values for each column, including (1) the minimum value, (2) the maximum value, and (3) the number of distinct values.

If the operator works with the numerical computation on the input data values directly, TCUDB first uses the minimum and maximum values along with the raw data types of the operator's input data. If the input data can be represented by TCU-compatible data types, including 16-bit half floating-point (`half`), 8-bit integers (`int8`), and 4-bit integers (`int4`), this stage will also determine the most compact data type. However, if the dataset cannot leverage any TCU-compatible data type, the feasibility test will suggest that the system not use TCUs in the incoming query. The database system can use other available options (e.g., a CPU-based or a pure GPU-based query engine) instead.

The number of records, the number of distinct values and the maximum/minimum values of each column also help the feasibility test to identify the case where the result value can surpass the range of 16-bit numbers and potentially lead to errors. Let m_1 represents the maximum of the maximum value and the absolute value of the minimum value within a column of n elements in one of the input matrix and that of a row with n elements is m_2 for another input matrix, the feasibility test can conservatively estimate the maximum value in the resulting matrix as $m_1 \times m_2 \times n$. If the maximum result value falls beyond the range of TCUs 16-bit number ranges, TCUDB will use query executors based on other hardware components instead.

4.2.2 Cost estimation of query plans. The cost of a TCU-accelerated operator contains:
(1) the data transformation cost `DT_op` which equals the latency for creating input matrices to perform the TCU-accelerated operators from the input tables,
(2) the data movement overhead `DM_op` for copying data between the host main memory or data storage to the GPU's device memory, and
(3) the computation time `CT_op`, the actual running time that the TCUs spend on executing the generated TCU code.

Depending on the estimated working set size of the query, the data transformation process of TCUDB can take place using the CPU or the GPU. The costs of `DT_op` and `DM_op` vary according to the approach.

CPU-based data transformation. The most general data transformation approach in TCUDB uses the host main memory and CPU to prepare inputs for the designated TCU-accelerated operator. This approach fills input matrices for a TCU-accelerated operator using methods described in Section 3 and works regardless of the estimated working set size of the query.

Consider the example of the 2-way natural join. To create the input matrices for an operator, TCUDB typically needs to scan through qualified/valid records for the operator and convert the values into the desired matrix representations. The data transformation cost is linear to the number of qualified/valid records. Let A and B be two input tables (which can also be intermediate results from subqueries) of size m and n respectively. Assume the throughput of the host system in scanning the raw data is a constant α. If their matrix representations `mat(A)` and `mat(B)` are not yet created, the scan operator will take $\texttt{DT\_op} \approx \alpha \cdot (m+n)$ in transforming input data to the desired matrices. The cost can also be $\alpha \cdot m$ or $\alpha \cdot n$ if either matrix is already created.

In this approach, the data movement overhead is controlled by (1) the volume of transformed matrices or input data and (2) the available bandwidth between the GPU and the host processor denoted by $\texttt{Bandwidth}_{\texttt{GPU/host}}$. If A is of dimension $M \times K$ with `type_A` and B is of dimension $K \times N$ with `type_B`, the data movement cost can be estimated by

$$\texttt{DM\_op} \approx \frac{MK \cdot \texttt{sizeof(type\_A)} + NK \cdot \texttt{sizeof(type\_B)}}{\texttt{Bandwidth}_{\texttt{GPU/host}}}. \quad (1)$$

GPU-assisted data transformation. To optimize the data transformation overhead `DT_op`, the query plan may perform the data transformation on the GPU to leverage its massive parallelism to convert thousands of pairs of values simultaneously into matrix format. In other words, we can take advantage of the GPU's parallelism to speed up the data transformation operation as well as avoid the additional data movement that copies the transformed matrix from the host memory to the GPU device memory. In contrast to the CPU-based approach, the data movement occurs before the data transformation in the GPU-assisted approach as the raw data must be present in the GPU's device memory in advance for the transformation to begin. Therefore, TCUDB can only use GPU-assisted data transformation when both the estimated working set size and the volume of necessary raw data (e.g., columns from the selected table) for transformation can fit in GPU's device memory. Leveraging the same 2-way natural join example, TCUDB can estimate the corresponding `DM_op` using Equation 2 as:

$$\texttt{DM\_op} \approx \frac{M \cdot \texttt{sizeof(type\_A)} + N \cdot \texttt{sizeof(type\_B)}}{\texttt{Bandwidth}_{\texttt{GPU/host}}}. \quad (2)$$

where M and N are the numbers of elements in the raw data columns of the joined columns and (`type_A`) and (`type_B`) are the raw data types of both columns before transformation.

In terms of `DT_op`, the GPU-based scan operator still takes $\approx \alpha \cdot (m+n)$ operations in transforming input data to the desired matrices – but a GPU can perform p of these in parallel if the GPU has p vector processors available. In modern GPU architectures, p is typically more than 2,000. The `DT_op` in GPU-assisted approach is estimated as $\texttt{DT\_op} \approx \frac{\alpha \cdot (m+n)}{p}$. Notice that the GPU-based approach needs to move raw data in Equation 2, TCUDB still needs to evaluate the summation of `DM_op` and `DT_op` to determine the most appropriate data transformation method.

Computation cost. Finally, the dimensions of the transformed input matrices also determine the TCU computation time. Using the number of records, the number of distinct values and the most compact data type derived from the feasibility test, TCUDB can estimate the required device memory and the density of input matrices for the operator. Based on the estimation, TCUDB can potentially take three different approaches in performing an operator.
(1) If all inputs and outputs fit within the device memory, TCUDB simply needs to copy all inputs into the device memory and invokes the matrix multiplication function once.
(2) In case the working set size exceeds the available device memory, TCUDB's query plan will need to apply the blocked and pipeline matrix multiplication algorithm [52, 97] to move parts of input and output data as well as perform matrix multiplications block-by-block. (Section 4.2.3)

(3) If the densities of input matrices are lower than a certain threshold (an architecture-dependent value), TCUDB will use sparse matrix multiplications instead. (Section 4.2.4)

Since each pair of values in input matrices requires 2 operations for multiplication and accumulation, the computation time in the simplest case where all input matrices fit in the device memory can be estimated by

$$\texttt{CT\_op} \approx MNK \times \frac{2}{\texttt{peak\_TCU\_TFLOPS}} \quad (3)$$

where peak_TCU_TFLOPS is the TCUs' peak number of floating-point operations per second (FLOPS). If the query results in inputs larger than device memory, TCUDB still leverages Equation 3 to estimate the cost but replaces peak_TCU_TFLOPS with the measured FLOPS from the blocked/pipelined matrix multiplications. For the cases where input matrices are sparse, TCUDB estimates the computation costs not only using the FLOPS from our sparse matrix multiplication implementation but also multiplying the cost by the density of inputs.

The final cost estimation is then the summation of the above three terms DT_op+DM_op+CT_op. TCUDB then compares this estimated cost with the estimated cost of the other CPU/GPU-based operators to decide whether to use TCUs. TCUDB obtain the most up-to-date estimations for $\texttt{Bandwidth}_{\texttt{GPU/host}}$ and peak_TCU_TFLOPS by checking the execution time of previous queries.

Note that there can be more than one TCU-accelerated plan because the system can choose a higher or lower-precision data type, which can change the decision of whether to perform transformation operator within the GPU or not.

4.2.3 Handling large datasets. Due to the limited device memory capacity (e.g., 24 GBs in our case), the input matrices of TCUDB's operators cannot fit in the GPU's device memory if the datasets are extremely large and dense. Once TCUDB catches such a case during the feasibility test, TCUDB will consider applying a blocked matrix multiplication algorithm for the corresponding query operators. The blocked matrix multiplication algorithm works by fetching a submatrix from the system main memory as a multiplicand, gradually fetching other same-sized submatrices as the multiplier, and aggregating the result to the corresponding submatrix in the result matrices.

TCUDB's implementation of blocked matrix multiplication extends MSplitGEMM [97] to support blocked matrix multiplications using TCUs. Both TCUDB's implementation and MSplitGEMM exploit pipeline parallelism by creating multiple streams in fetching input submatrices, performing matrix multiplication and accumulation, and writing back results simultaneously. TCUDB's implementation uses TCUs for matrix multiplication and accumulation instead of conventional GPU cores. During the periodical microbenchmark tests, TCUDB also performs a series of tests to figure out the optimal size of submatrices that balances the latency of each stage in the pipeline to maximize the computation throughput. The measured throughput using these optimal parameters will also be used as the metrics for evaluating the costs of large and dense inputs in Section 4.2.2.

4.2.4 Handling sparse matrices. Due to the current capability of TCU hardware in handling sparse matrices, conventional TCU operators that assume dense matrices as their inputs may not always outperform a GPU plan when the input matrices to a TCU-accelerated operator are very sparse. Therefore, TCUDB implements a TCU-accelerated sparse matrix multiplication (TCU-SpMM) operator that

- transforms an input into a compressed sparse row matrix format (CSR)
- partitions an input matrix into 16×16 submatrices,
- skips submatrices containing all 0s,
- multiplies the rest using TCUs and accumulates results [94].

By doing so, the TCU-SpMM operator can still leverage TCU's computation power but on a much smaller number of submatrices pairs when the input matrices are large and sparse.

To determine whether a TCU-SpMM-based plan should replace the dense multiplication plan, TCUDB needs to estimate the cost similar to the regular cases with dense matrices. We estimate the total cost by multiplying the estimated dense operator cost by the inputs' densities. In addition, the TCU-SpMM-based operator requires scanning inputs to construct/partition a matrix and filter those all-0-submatrices. TCUDB estimates this part of the cost with a simple linear function with respect to the input size.

Finally, the query optimizer of TCUDB still needs to evaluate plans using the GPU-based HashJoin cost model [92], in particular sparse matrix multiplication on conventional CUDA cores to determine whether a TCU-SpMM-based plan is more efficient.

5 EXPERIMENTAL RESULTS

Leveraging TCUs' capabilities in optimizing matrix algebra, TCUDB delivers up to 14× speedup over a conventional GPU-based DB engine for the sample queries that Section 3 describes. Inspired by the result, we experimented with TCUDB in real-world application query workloads with inputs as large as 24 GBs. In summary, TCUDB achieves up to 7.52× speedup in matrix multiplications, up to 3.96× speedup for analytic queries in the star schema benchmark, up to 288× speedup in entity matching queries, and up to 4.22× speedup for the core of the PageRank algorithm. The comparison of TCUDB performance on different GPU architectures also reveals the strong potential of TCU-accelerated DB engines in the future.

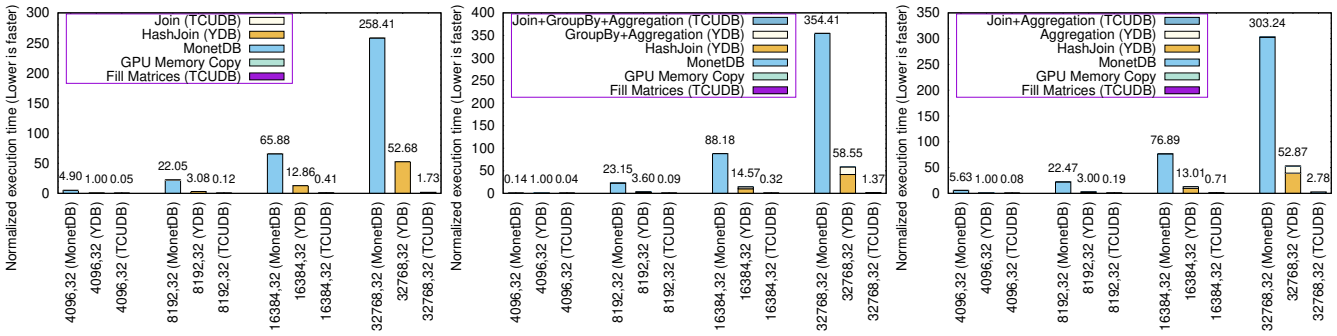
5.1 Experimental Methodology

We conducted experiments on a machine with an Intel Core i7-7700K processor, 32 GB DDR4 DRAM. The processor contains 4 cores and each processor core runs at 4.2 GHz by default. The GPU in our experiments is an NVIDIA GeForce RTX 3090 GPU based on Ampere architecture. This GPU contains 24 GB GDDR6X device memory and 328 Tensor Cores and attaches to a PCIe 3.0 x16 slot. The TCU-accelerated operator library in TCUDB is implemented using a NVIDIA CUDA Toolkit 11.2. The system runs a Linux 4.15.0 kernel with the NVIDIA driver version in 460.32.03. We compared TCUDB with a state-of-the-art GPU execution engine for warehouse-style queries, YDB [92] and a pure CPU-based execution engine, MonetDB [10], as reference designs.

5.2 Microbenchmark

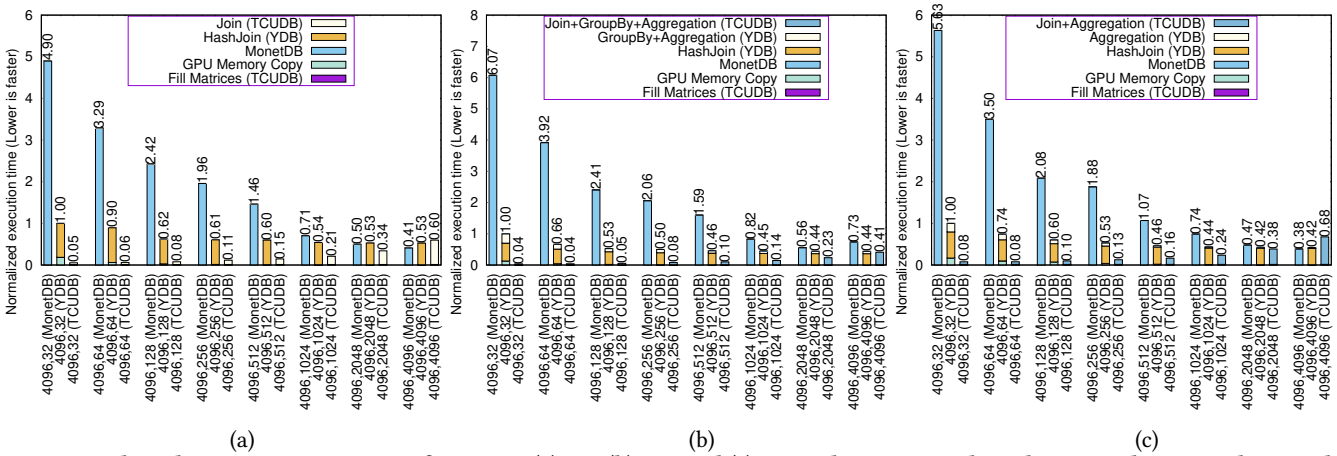
To allow query optimizers to select the right query plans, the database engine must obtain samples of executing workloads using TCU-accelerated operations. Upon installing TCUDB in the system or when the system detected any change in hardware configurations, TCUDB will perform a one-time sampling process that runs a set of microbenchmark workloads to collect critical timing information for query optimizations.

During the sampling process, TCUDB will execute three main queries, Q1, Q3 and Q4 from Section 3, with various-sized, random-generated input datasets. TCUDB does not evaluate Q2 and Q5 as they are essentially combinations of other queries. The sampling process also helps us to classify the cases where TCUDB is superior to the conventional GPU-accelerated engine and identify the source of performance gain/loss in TCUDB. With large system main memory and aggressive file system caching by operating systems as well



(a) (b) (c)

Figure 7: The relative execution time of running (a) Q1, (b) Q3, and (c) Q4 with various number of records and 32 distinct values in the target attribute on TCUDB, YDB, and MonetDB.



(a) (b) (c)

Figure 8: The relative execution time of running (a) Q1, (b) Q3, and (c) Q4 with 4096 records and various distinct values in the target attribute on TCUDB, YDB, and MonetDB.

as the underlying high-performance NVMe SSD, we have not observed significant disk load time in each DB engine's initialization phase.

As MonetDB is a full-fledged system, we excluded the additional steps/overheads by measuring only the time to execute the physical plan for a fair comparison. (We use the "–timer=performance" option and disable the resulting output to report the runtime part only.)

Figure 7 and Figure 8 present a subset of microbenchmark results from the sampling process on the default testbed described in Section 5.1. We label the x-axis of each sample in this figure with two parts in the configuration. The first part is the parameters for the query, M, K and N, that represent the sizes of the input matrices for each evaluated operator where one matrix has the dimension of $M \times K$ and the other is $K \times N$. To save space, we only present the cases when $M = N$ and label each configuration with their values of M and K as M, K in these figures. The second part is the DB engine (i.e, TCUDB, YDB, or MonetDB). The vertical axis in each figure shows the aggregated execution time in each step of running these queries, normalized to the total time when running the same query using YDB, the conventional GPU-accelerated engine, with $M = N = 4096$ and $K = 32$.

Figure 7(a) shows the performance of Q1 for TCUDB, YDB and MonetDB from input sizes 4096 to 32768. Both TCUDB and YDB significantly outperform MonetDB for this query. TCUDB outperforms YDB in most configurations. The advantage of TCUDB is especially significant when datasets grow. TCUDB outperforms YDB by 14× for the case of (32768, 32) and 9.3× for (16384, 32), but only 1.18× for (4096,32). Observing the breakdown of execution time in Figure 7(a), we found the major speedup comes from the significant reduction of computation time from the TCU-accelerated join operator, despite the additional overhead in filling and transforming datasets into the desired matrices for TCUDB.

Figure 8(a) varies the number of distinct values that affect the sparsity of input matrices in Q1 for TCUDB's join operator. As the number of distinct values becomes larger, the performance advantage of TCUDB's join operator over YDB and MonetDB begins to shrink. Because the sizes of one dimension of both input matrices for the TCUDB join operator in Q1 depends on the number of distinct values from the chosen attribute to perform matching, matching on an attribute with more distinct values will lead to computation on larger but sparse matrices. In contrast, YDB's and MonetDB's `HashJoin` algorithm produces smaller vectors as the chance (i.e., total number) of records sharing a single value reduces if the number of distinct values increases. Therefore, even though YDB's and MonetDB's `HashJoin` operator needs to work on more pairs of vectors, each pair of vectors have smaller dimensions. However, TCUDB's join operator still outperforms YDB and MonetDB in all cases until the number of distinct values reaches 4096. This profiling result suggests that TCUDB select a GPU-hash-join-based or sparse-matrix-based implementation if the density of input matrices is below 0.04% on our testbed.

Figure 7(b) presents the performance of running Q3 using TCUDB, YDB and MonetDB. Q3 evaluates the group-by and aggregations

over join query. Unlike the conventional GPU-accelerated DB engine where group-by and aggregations are separate operations after the hash join, TCUDB can implement the whole Q3 using just one matrix multiplication. As a result, the execution time of using TCUDB of executing Q3 remains similar to executing Q1 when the input parameters are the same. However, YDB or MonetDB always have to perform the additional group-by operations and leads to a longer execution time than performing Q1 for the same inputs. Therefore, the performance advantage of TCUDB becomes more significant for Q3. For (32768, 32), TCUDB can outperform YDB by 45×.

When we increase the number of distinct values as in Figure 8(b), TCUDB becomes less advantageous, similar to the phenomenon in Q1. However, as TCUDB still uses single-matrix-multiplication-based Join/Aggregation/GroupBy operation to perform operations where YDB or MonetDB needs multiple-step HashJoin and GroupBy/Aggregation operators, TCUDB still outperforms YDB and MonetDB in all cases.

Figure 7(c) presents the relative execution time of Q4 on TCUDB, YDB and MonetDB. YDB and MonetDB will perform Q4 using *HashJoin* and then an aggregate query but without a group-by operator. Therefore, the overall execution time in each configuration of YDB and MonetDB is less than Q3 because of the elimination of group-by operator. However, again, TCUDB still implements this operator using single matrix multiplication on the transformed input matrices. Therefore, TCUDB achieves 19× speedup for (32768, 32).

As in Q1 and Q3, TCUDB becomes less advantageous when we increase the number of distinct values as in Figure 8(c). Because the amount of operations in YDB and MonetDB for Q4 is fewer than Q3, we still see TCUDB falls short when the number of distinct reaches 4096 and suggest an alternative plan for cases where input matrix densities are below 0.04%.

5.3 Analytic queries: Star Schema Benchmark

We evaluate the performance of TCUDB on the popular Star Schema Benchmark (SSB) [68], a benchmark suite modeling the data warehouse workloads. SSB is widely used in benchmarking analytic engines due to its realistic modeling of data warehousing workloads. The database form a star schema consisting of one fact table (`lineorder`) and four dimension tables (`supplier, customer, date` and `part`) connected to the fact table by foreign keys.

The benchmark provides 13 queries in 4 flights. TCUDB supports all the 13 SSB queries. Figure 9 compares the performance of TCUDB, YDB and MonetDB in running SSB queries with scaling factors varying from 1 to 8 resulting in data sizes from 0.7GB to 5.6GB.

Figure 9 summarizes the results. TCUDB outperforms both YDB and MonetDB in all evaluated SSB workloads with up to 3.96× speedup when running Q4.1 with scaling factor as 8. Even with the worst performing SSB Q3.1, TCUDB still maintains the same level of performance as YDB. These promising results show that TCUDB has the potentials of being integrated into real-world analytic engines.

5.4 Case studies: matrix multiplication, entity matching, and PageRank

In addition to individual operators, we also evaluated three representative use cases, matrix multiplication, entity matching and PageRank to demonstrate TCUDB's capabilities in handling intensive operations and large datasets.

	2048 ×2048	4096 ×4096	8192 ×8192	16384 ×16384	32768 ×32768
$x = 0, 1$	0	0	0	0	0
$-2^7 \leq x < 2^7$	0	0	0.00076%	0.00076%	0.00076%
$-2^{15} \leq x < 2^{15}$	0.00114%	0.00450%	0.00908%	0.00908%	0.00908%
$-2^{31} \leq x < 2^{31}$	0.00122%	0.00451%	0.00909%	0.00909%	0.00909%

Table 1: The mean absolute percentage error rates (MAPE) of matrix multiplication queries with various value ranges.

5.4.1 Matrix Multiplication. Matrix multiplication was once considered inefficient for relational databases. With the help of hardware-accelerated matrix multiplications, TCUDB can make queries containing complex linear algebra operations more efficient. We use a query in Figure 5 to demonstrate this use case. We create two tables A and B where each record in both tables has three attributes (row_num, col_num, val) as the input. We generate the synthetic dataset according to this schema with input matrices of dimensions up to 32768×32768 and data volume up to 24 GB, approximately 2.14 billion records.

Figure 10 presents the relative execution time and breakdown of performing matrix multiplication on TCUDB and YDB, using YDB with each table containing 4096×4096 records as the baseline. We did not include MonetDB's result in these Figures as MonetDB cannot finish these queries within a reasonable amount of time and present MonetDB's results in Figure 10 would render the results of TCUDB and YDB invisible. When the dataset contains fewer than 16384×16384 records, the input matrices that TCUDB creates for the TCU's `Join + Aggregation + GroupBy` operator completely fit in the GPU's device memory. TCUDB consistently outperforms YDB and delivers up to 7.51× speedup. When the dataset contains 32768×32768 records for each table, TCUDB must partition the input matrices into submatrices, use the block algorithm, and pipeline the swapping in/out of submatrices to perform the Join/Aggregation/GroupBy operator. TCUDB still performs multiplication and aggregation of submatrices using TCUs. Even with the overhead of data exchanges in the blocked Join/Aggregation/GroupBy operator, TCUDB is still able to outperform YDB by 7.92× for the case of 32768×32768 records for each table. As datasets fit in the system's main memory as well as the operating system's aggressive caching and the help of high-speed NVMe SSD, the data load time from storage is relatively insignificant in these experiments. The data movement (`cudaMemcpy`) time is the most timing critical stage for TCUDB. However, the amount of time is comparable to TCUDB and YDB because both engines only transfer the required data to the device memory. The most time-consuming parts for YDB are `HashJoin` and `GroupBy` operations because code using conventional CUDA cores needs to iterate tables row by row. YDB spends up to 14× (in the case of 16384×16384 records in each table) more execution time in `HashJoin` and `GroupBy` than TCUDB's single Join/Aggregation/GroupBy operator.

Due to the limited 16-bit precision of TCUs, they cannot generate 100% accurate results in some cases. Table 1 shows the mean absolute percentage error (MAPE) rates in performing matrix multiplication queries. In the cases where the values are only 0s and 1s – similar to the cases of Q1 and Q2, the generated TCUDB operations can always produce accurate outputs. Therefore, the result implies that TCUDB never leads to incorrect outcomes for subqueries like Q1 and Q2. When we enlarge the value ranges, we start to see errors in results, but with very limited imprecision – even in the worst case, the MAPE is lower than 0.01%. We believe this error rate is acceptable in most cases. This level of data error does not cause any inexact query results for the entity matching or the microbenchmark workloads. For numerical analysis such

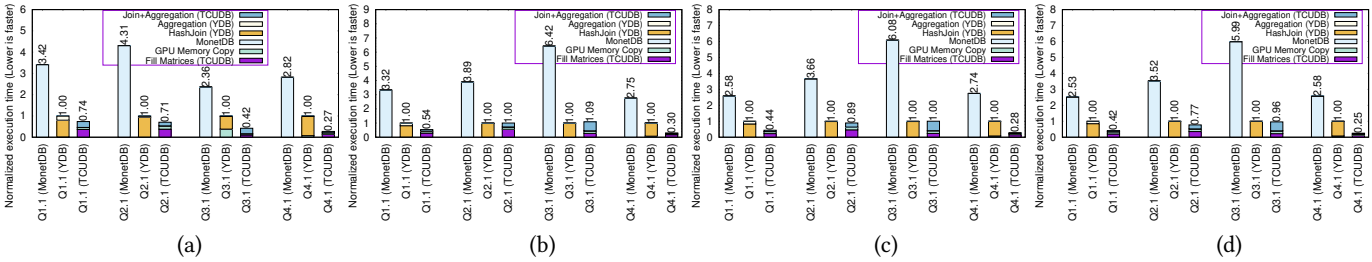


Figure 9: The relative runtime of star schema benchmark on TCUDB compared to MonetDB and YDB running the same query as the baseline with scaling factor (a) 1, (b) 2, (c) 4 and (d) 8.

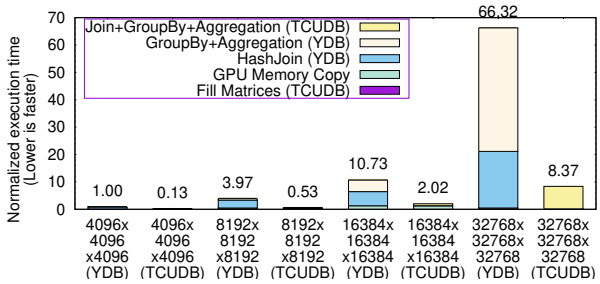


Figure 10: The relative execution time and breakdown of matrix multiplication query on TCUDB and YDB.

as SSB, the result values can have minor error rates typically less than 0.001% for cases with input values larger than 2^{15} or matrices with a dimension larger than 8192 due to the 16-bit representation. However, the error rate is very insignificant and never results in misplacement of rankings and orderings of the query results.

5.4.2 Entity Matching. Entity matching (EM), also known as entity resolution, fuzzy join, and record linkage, searches records correspond to the same real-world entities from different data sources [16, 25, 27, 50]. A key component of EM is *blocking* [30, 50, 70]. Given two tables of entity records, the goal of blocking is to apply matching heuristics to quickly generate candidate pairs of records that are likely to be real matches, which are later processed by a more accurate pairwise classifier (aka the *matcher*). Scalability is the main challenge of blocking as the heuristics are typically natural join conditions (e.g., selecting products with the same brand) that often produce large join results. Therefore, we expect that TCUDB can provide significant performance gain for this EM workload.

To validate this hypothesis, we evaluate TCUDB's performance on two real EM datasets BeerAdvo-RateBeer and iTunes-Amazon from the Deepmatcher benchmark [62]. The BeerAdvo-RateBeer dataset contains two tables, where one of them contains 3,777 rows and the other contains 2,671 rows, from different sources. Each table has the same table schema with five attributes {`ID, BEER_NAME, FACTORY, STYLE, ABV`}. Table 2 reveals the number of distinct values of each attribute, which acts as one matrix dimension for TCUDB when performing join operation. We evaluate the following query on BeerAdvo-RateBeer dataset to perform blocking:

```
-- EM-blocking query for BeerAdvo-RateBeer dataset:
SELECT TABLE_A.ID, TABLE_A.BEER_NAME,
       TABLE_B.ID, TABLE_B.BEER_NAME
FROM TABLE_A, TABLE_B
WHERE TABLE_A.ABV = TABLE_B.ABV; -- attributes may vary
```

The iTunes-Amazon dataset contains two tables, where one of them has 6,907 rows and the other has 55,923 rows, from iTunes and Amazon music. Both tables share the same table schema with seven attributes `ID`, `PRICE`, `GENRE`, `TIME`, `ARTIST`, `COPYRIGHT`, and `ALBUM`.

Attribute	ABV	Style	Factory	BeerName
#distinct values	20	71	3678	6228

Table 2: Distinct values in BeerAdvo-RateBeer dataset.

Attribute	Price	Genre	Time	Artist	Copyright	Album
#distinct values	12	813	908	2418	3197	6004
#distinct values (scaled)	25	1614	1208	6420	8199	11005

Table 3: Distinct values in iTunes-Amazon dataset.

Table 3 shows the number of distinct values for each attribute in the iTunes-Amazon dataset. We perform the following query on the iTunes-Amazon dataset for blocking:

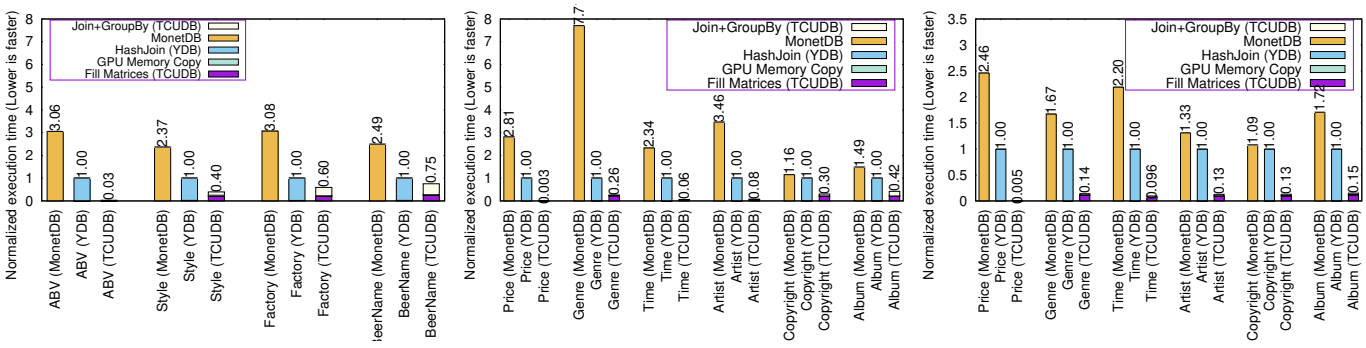
```
-- EM-blocking query for iTunes-Amazon dataset:
SELECT TABLE_A.ID, TABLE_A.SONG,
       TABLE_B.ID, TABLE_B.SONG
FROM TABLE_A, TABLE_B
WHERE TABLE_A.ARTIST = TABLE_B.ARTIST; -- attributes may
     vary
```

Figure 11 presents the result of running the above EM-blocking queries on the two datasets and different attributes. As the execution time varies significantly among different queries, we use YDB running the same query as the baseline and show the relative execution time. TCUDB outperforms YDB in most cases, achieving a maximum speedup of 288× among our experiments.

TCUDB is especially effective when the number of distinct values is small. For the BeerAdvo-RateBeer dataset in Figure 11(a), TCUDB is at most 33× faster than YDB when searching for matches on the ABV attribute where there are only 20 distinct values. For the iTunes dataset in Figure 11(b), TCUDB further shows 288× speedup over YDB when performing entity matchings on the Price attribute that only has 12 distinct values. When the number of distinct values becomes larger, the performance advantage of TCUDB's operators relying on dense matrix operations over YDB starts to shrink, for the reason we have described in Section 5.2. However, as TCUDB uses TCU-spMM in these cases, TCUDB still outperforms YDB and MonetDB in all cases.

Scaling up. To demonstrate the ability of TCUDB and the query optimizer in dealing with larger EM datasets, we synthesized an iTunes-Amazon dataset by randomly duplicating each input table's entry values. The resulting dataset contains 111,846 records in the larger input source and 13,814 in the smaller one. The #distinct values (scaled) show the resulting distinct values in each attribute field of this synthetic dataset.

Figure 11(c) shows the relative execution time of TCUDB, compared with YDB running the same query. TCUDB still outperforms YDB in most cases, by up to 216× when performing matching on



(a) (b) (c)

Figure 11: The relative runtime of the EM-blocking queries on TCUDB using the default deepmatcher datasets (a) BeerAdvo-RateBeer (b) iTunes-Amazon and (c) scaled iTunes-Amazon, compared to MonetDB and YDB running the same query as the baseline.

#Nodes	1024	2048	3072	4096	8192	16384	32768
#Edges	2058	4152	6280	8450	17444	37106	82070

Table 4: Reduced graph information.

the price field. When TCUDB performs the query on artist, album and copyright fields, the query optimizer detects that these cases contain way too many distinct values and the pure TCU operator cannot efficiently process the query since the input matrices are sparse. Therefore, TCUDB uses a TCU-SpMM operator for query processing and achieves more than 6.67× and 7.8× speedup on Copyright and Album, respectively, over YDB that essentially performs sparse matrix multiplications using CUDA cores.

5.4.3 PageRank. To demonstrate TCUDB's ability in processing graph-related queries as well as data analytics, we also evaluate TCUDB in performing the PageRank algorithm. PageRank algorithm consists of three steps: (1) computing the out-degree of each node, (2) initializing the value of each node, and finally, (3) calculating the PageRank iteratively. The whole PageRank algorithm can be implemented as the following three queries:

```
-- PR Q1: compute out-degree
SELECT NODE.ID,
       COUNT(EDGE.SRC)
FROM NODE, EDGE
WHERE NODE.ID = EDGE.SRC
GROUP BY NODE.ID;
```

```
-- PR Q2: initialize values
SELECT NODE.ID,
       (1-@alpha)/@num_node as rank
FROM NODE, OUTDEGREE
WHERE NODE.ID = OUTDEGREE.ID;
-- @alpha is 0.85 by default
```

```
-- PR Q3: calculate the PageRank score
SELECT
      SUM(@alpha * PAGERANK.rank / OUTDEGREE.DEGREE)
      + (1-@alpha)/@num_node
FROM PAGERANK, OUTDEGREE
WHERE PAGERANK.ID = OUTDEGREE.ID;
-- @alpha is 0.85 by default
```

Among these three queries, PR Q1 represents step 1, PR Q2 represents step 2 and PR Q3 represents step 3. A complete run of the PageRank algorithm will invoke PR Q1 and PR Q2 once and execute PR Q3 several times until the PageRank scores converge or reach the maximal number of iterations.

We used the Pennsylvania road network dataset from SNAP [53] that contains 1.08M nodes and 1.54M edges as the input dataset. Evaluated TCUDB under different sizes of graphs, we created a subset of the original graph for our experiments using the most popular N nodes and preserving the connectivity of selected nodes in the original graph. Table 4 describes the characteristics of the resulting graphs. Figure 12 illustrates the relative execution time and the breakdown of latency in each system component for all three queries. We normalized the execution time to run the same query using the graph with 1K nodes on YDB.

Though the computation of out-degree using PR Q1 is a one-pass task (Figure 12(a)), TCUDB's pure TCU Join/Aggregation/-Groupby operator still has advantages when the graph is small, by up to 3.6× speedup with 1K graph. For graphs with more than 3K nodes, TCUDB selects TCU-SpMM to exercise the Join/Aggregation/Groupby operator due to the low density in their adjacency matrices. Compared with a pure TCU Join/Aggregation/Groupby operator, a TCU-SpMM-based operator spends more time in creating operator inputs. However, as the TCU-SpMM-based operator skips submatrices with all 0s, TCU-SpMM significantly reduces the computation time on matrix multiplications and allows TCUDB to outperform YDB that essentially performs sparse matrix operations on CUDA cores by up to 7.69×.

PR Q2 is also a one-time process in the PageRank algorithm but requires additional arithmetic to initialize the values for PR Q3. Figure 12(b) shows that TCUDB consistently performs better than YDB. with speedup ranging from 1.40× to 4.18×. Similar to Q1, TCUDB uses a dense TCU operator for graphs smaller than 2K and uses TCU-SpMM's Join/Aggregation/Groupby to exercise queries for larger graphs.

Figure 12(c) shows the performance of TCUDB and YDB in performing PR Q3, the core of the PageRank algorithm that the algorithm executes multiple times until values converge. In our experiments, we performed PR Q3 for 50 iterations for each configuration. For PR Q3, TCUDB's Join/Aggregation/Group operator improves the execution time of arithmetic calculations over the multi-step process in YDB. TCUDB is 4.22× faster than YDB with 1K nodes in the graph. Even with graphs containing 8K nodes, TCUDB still outperforms YDB by 3.24×, as TCU-SpMM's Join/Aggregation/-Groupby skips submatrices containing all 0s.

5.5 Comparison with Graph Database Systems

TCUDB demonstrates the potential of using relational database engines to analyze datasets that are originally graphs through case studies on PageRank. On the other hand, graph database systems

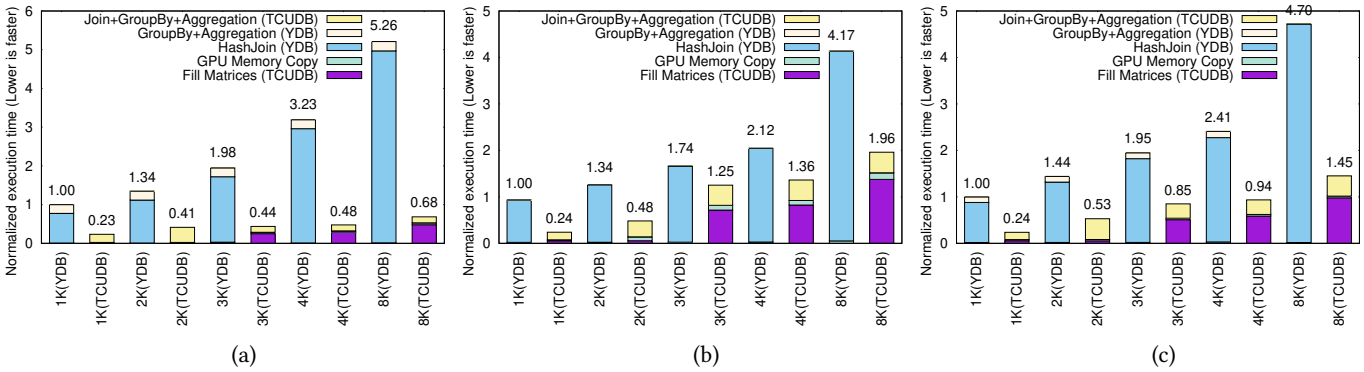


Figure 12: The relative execution time of executing PageRank queries (a) Q1, (b) Q2, and (c) Q3 on TCUDB, using YDB running the same query as the baseline. Each value equals the actual query time divided by YDB's runtime on the 1k table.

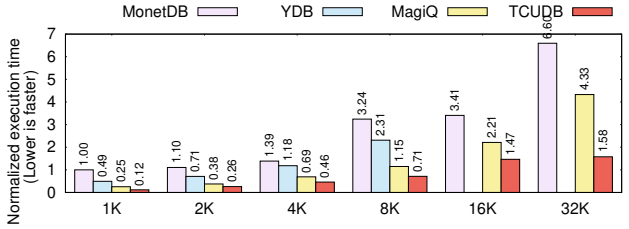


Figure 13: The relative latency of the core join and aggregation operation when running PageRank Q3 in MonetDB, YDB, MAGiQ, and TCUDB.

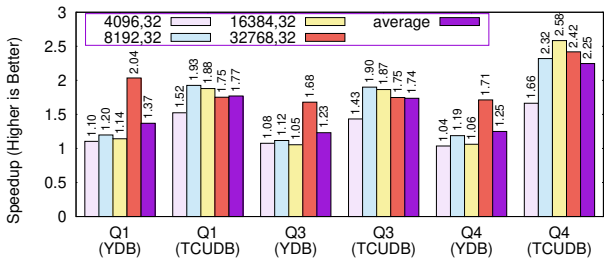


Figure 14: The microbenchmark speedup of using RTX 3090 over RTX 2080 for Q1, Q3, Q4 on TCUDB and YDB. Each value equals RTX 2080 time divided by RTX 3090 time.

provide more natural representations and storage layouts to serve the same purpose. To investigate the strength and the implications of TCUDB in the future advancement of graph database systems, this section compares the performance of TCUDB on the PageRank algorithm with the state-of-the-art graph query engine MAGiQ [43]. In contrast to the table-style storage that relational database systems and TCUDB use, MAGiQ's backend storage is organized as 2-dimensional key-value pairs, typically already in some sparse matrix formats. MAGiQ translates the queries described by SPARQL into a set of GraphBLAS [19] calls on these sparse matrices.

We use the same SNAP dataset as in Section 5.4.3 to evaluate the PageRank performance of MAGiQ with GPU and TCUDB. Figure 13 compares the performance of MAGiQ and TCUDB with MonetDB and YDB as references. However, the released version of YDB can only support these queries with datasets containing at most 8,192 nodes. Due to the large overhead of retrieving sparse matrices in MAGiQ compared to other counterparts, we only present the latency of the core join and aggregation operations in each experiment. The presented numbers are PageRank Q3's performance on the sub-sampled graphs listed in Table 4. MAGiQ outperforms YDB, the pure GPU query engine on relational databases, in all cases, demonstrating that a customized graph database engine does provide a more efficient platform for graph analytics on the same architecture. Meanwhile, TCUDB outperforms MAGiQ in all evaluated cases. The main reason is that TCUs allow TCUDB to more efficiently exercise these queries than GraphBLAS that uses only conventional GPU cores at this moment. We observed that the difference is more significant as the graph becomes larger and more sparse. These results help us generate two insights. First, with TCUs, graph analytics can be efficient with existing relational databases. Second, graph databases can also be more efficient if their backends can leverage TCUs as TCUDB does.

5.6 TCUDB on different GPU architectures

To investigate the performance scaling on different GPU architectures and their implications to the design of the TCU-accelerated DB engine, we perform experiments on NVIDIA's 2080, which uses an earlier Turing GPU architecture with the last generation TCU available.

Figure 14 compares the performance of microbenchmarks on the same queries Q1, Q3, Q4 mentioned in Section 5.2 using both YDB and TCUDB on RTX 3090 GPU and RTX 2080 GPU. The baseline ran the same query using the same DB engine on RTX 2080. We observed that TCUDB performs better generation-over-generation – when using RTX 3090 TCUDB achieved an average speedup of 1.77× on Q1, 1.74× on Q3 and 2.25× on Q4, but YDB only achieved 1.37× on Q1, 1.23× on Q3 and 1.25× on Q4. It is worth noting that RTX 3090 contains only 328 Tensor Cores compared to 368 Tensor Cores in RTX 2080. On the other hand, the RTX 3090 has 10496 conventional CUDA GPU cores for vector processing while RTX 2080 only has 2944 of them. The results reveals that the performance scaling of Tensor Cores in newer generations of GPU architectures is stronger than conventional vector processing cores, given that RTX 3090 has fewer Tensor Cores, 3.4× more CUDA cores, but TCUDB's speedup is more significant on RTX 3090. This result also indicates applications, including DB engines, with a larger portion relying on TCUs will expect to receive more performance gains when new GPU architectures are used.

6 RELATED WORK

Hardware-accelerated DB's. Integrating advanced hardware accelerators into database systems has been an active line of research for the past few decades. Commonly considered accelerators include GPUs [12, 13, 17, 29, 34, 54, 72, 76, 83, 87, 89, 90, 92, 93] and

FPGAs [28, 59, 63, 69, 88]. Optimization techniques have been proposed for database operators including Select [79], Join [35, 37, 38, 78], Sort [33] and Group-by Aggregate [47]. In particular, to support star schema queries, YDB [92] implements these operators into a data warehousing engine, which we used as a baseline for TCUDB. GPUs have also been incorporated into industrial DB engines such as OmnisciDB [67], Kinetica [49], and BlazingSQL [9].

With GPUs reducing the computation time but the increasing volume of datasets, the data movement overhead becomes more significant to the degree that DB engines must be aware [11, 73]. Several GPUDB systems incorporate GPU RDMA techniques [4, 48, 55, 66, 81, 95] to directly access data on the storage devices [15, 54, 96] or efficiently exchange data among multiple GPUs [58], bypassing the host system's main memory. This paper is orthogonal but will receive significant benefit from this line of research projects. To fundamentally address the data movement overhead, DB systems can push down the computation of query processing into existing or additional hardware logic to offload part of the computation instead of using computing resources on the host system [22, 23, 44, 45, 51, 85, 86]. However, due to the power and hardware budget of memory/storage devices, the computing resources near data locations are typically limited. For the cases studied in this paper, DB systems still have to rely on host computing resources (i.e., GPUs, TCUs, FPGAs and TPUs) to efficiently perform the received queries. With modern matrix processors need to partition matrix data and accept reduced precision values, DB system like this paper can still leveage near data processing models to reduce precisions [41] or reshape data [56] if the processing power in storage devices is permitted.

Matrix processors in relational databases. To the best of our knowledge, TCUDB is the first database system that fully leverages Tensor Core Units (TCUs) as matrix processors to accelerate compute-intensive database queries. Prior work [18] leverages TCUs for scan/reduction operators by mapping scan/reduction into matrix-vector products. However, [18] only treats TCUs as wider vector processors leveraging TCU's fused operations that can perform multiplications and accumulations in a single operation. In contrast, TCUDB transforms queries into matrix-matrix operations so that it can fully utilize TCUs' nature as matrix processors. Prior work [40] investigated the feasibility of accelerating relational queries using Google Cloud's closed-architecture TPU platform and proprietary version of TensorFlow. However, due to limitations of the platform, [40] only accelerates vector-based operators such as reduced sum. Its implementation can only support single-table queries (called Dimension Join in [40]). On the other hand, TCUDB can support a wide range of queries include two-way natural joins by leveraging TCUs for matrix operations.

Join processing as matrix multiplication. A key technical contribution of TCUDB is to cast the join operator as dense matrix multiplication. While being unconventional due to the high theoretical computational complexity, this idea was explored in [5] and more recently in [20]. In particular, [20] proposed a fast join algorithm that combines worst-case optimal join algorithms [65] and fast matrix multiplication. The authors also provide a CPU-based implementation highlighting performance gain from the highly-optimized linear algebra framework such as Intel MKL [84]. The implementation achieves up to 50× performance improvement compared to baselines. In TCUDB, we further push this trend by leveraging NVIDIA's TCUs that are specialized for tensor processing, which commonly appears in deep learning workloads to achieve up to 288× performance gain.

Graph queries as matrix operators. Processing queries as matrix operators have also been considered in the context of graph databases. In particular, MAGiQ [43] accelerates SPARQL queries on RDF graphs by translating queries into sparse matrix linear algebra programs. We have discussed the key differences between TCUDB and MAGiQ in Section 5.5. Our experiment results also show that integrating TCUDB's strategy of executing those matrix operators in TCUs can be an interesting optimization opportunity for graph query engines like MAGiQ.

Advanced in-database analytics. To accommodate the exponential growth in data science and machine learning applications, a recent line of work [3, 14, 24, 26, 39, 42, 57, 80] focuses on supporting advanced analytics queries that involve linear algebra (LA) operators. TCUDB shares the goal of LevelHeaded [3] in identifying the worst-case optimal join (WCOJ) [65] or LaraDB's rule-based translation between relational queries and parallel LA queries, but TCUDB additionally provides the capability of translating (parts of) the query to TCU-accelerated matrix multiplication operator(s) and different sets of opportunities from the orders of magnitude speedup by TCUs in such operations. TCUDB also offers a better system architecture by making TCU-accelerated operators as integral parts of the DB engine and thus incurs zero system overhead in processing TCU-accelerated queries. In contrast, query analyzers like AIDA [26] that rely on external parallel libraries from different language frameworks from the query engine always lead to redundant memory copies that are especially significant in our use cases. Compared with proposals relying on SQL extensions that introduce data type labels (e.g., vector and matrix) to support LA queries [57] or new query languages [14], TCUDB does not require any change to the SQL.

Entity Matching and PageRank. A major challenge in EM [16, 25, 27, 50] is in the blocking phase [30, 50, 70] to reduce the number of candidate pairs to be matched by heuristics specified as natural joins. Our case study demonstrates that TCUDB delivers over 300× speedup for blocking queries compared to a GPU-accelerated `HashJoin` implementation. This indicates the potential of building scalable EM systems with TCUDB as the backend.

PageRank is a graph-based ranking algorithm with applications from web searches to basic science (see [32] for a survey). PageRank is also commonly used in benchmarks of graph databases [21, 61, 64]. While there has been an effort to accelerate PageRank (and other graph analytic queries) using GPUs [74, 77, 91], to our knowledge, TCUDB is the first to attempt to accelerate PageRank using TCUs.

7 CONCLUSION

This paper proposes, implements and evaluates TCUDB, an efficient database query engine with TCUs, an emerging type of AI/ML hardware accelerator presented in modern GPU architectures. This paper identifies query patterns that match TCUs' acceleration model. Through solving technical difficulties such as remapping inputs and limited precision, the resulting TCUDB shows ours achieves up to 288× speedup against the baseline GPU-accelerated DB engine. The performance gain of TCUDB over conventional GPU-based DB engines indicates a strong performance scaling in new GPU architectures. For future work, we plan to extend TCUDB by exploring more potential workloads and addressing the complex query optimization problem with multiple accelerators of different types.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by the two National Science Foundation (NSF) awards, CNS-1940048 and CNS-2007124. This work was also supported by new faculty start-up funds from University of California, Riverside.

REFERENCES

[1] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.

[2] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: How different are they really?. In *SIGMOD*. ACM, 967–980.

[3] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 449–460.

[4] AMD Inc. 2014. AMD FirePro DirectGMA. http://developer.amd.com/community/blog/2014/09/08/amd-firepro-gpus-directgma/.

[5] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*. Association for Computing Machinery, 121–126.

[6] Apple Inc. 2020. Apple M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[7] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. 2013. GPUDB source code. http://code.google.com/p/gpudb

[8] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. Association for Computing Machinery, 94–103.

[9] BlazingSQL Inc. 2015. BlazingDB. https://blazingsql.com.

[10] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[11] Sebastian Breβ, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-Accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1891–1906.

[12] Sebastian Breβ, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal* 27, 6 (Dec. 2018), 797–822.

[13] 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 (2013), 1398–1403.

[14] Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. 2019. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4, Article 15 (Oct. 2019).

[15] W. G. Choi, D. Kim, H. Roh, and S. Park. 2020. OurRocks: offloading disk scan directly to GPU in write-optimized database system. *IEEE Trans. Comput.* (2020), 1–1.

[16] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. Entity resolution in the web of data. *Synthesis Lectures on the Semantic Web* 5, 3 (2015), 1–122.

[17] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Proceesings of the 9th Biennial Conference on Innovative Data Systems Research*.

[18] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing*. Association for Computing Machinery, 46–57.

[19] Tim Davis, Michel Pelletier, and Scott Kolodziej. 2017. GraphBLAS Standard. https://github.com/GraphBLAS.

[20] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation Using Matrix Multiplication. In *SIGMOD*. Association for Computing Machinery, 1213–1223.

[21] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E Lee. 2020. Aggregation support for modern graph analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 377–392.

[22] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1221–1230.

[23] Jaeyoung Do and Jignesh M. Patel. 2009. Join Processing for Flash SSDs: Remembering Past Lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. 1–8.

[24] Oksana Dolmatova, Nikolaus Augsten, and Michael H Böhlen. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2573–2587.

[25] Xin Luna Dong and Divesh Srivastava. 2013. Big data integration. In *2013 IEEE 29th international conference on data engineering (ICDE)*. IEEE, 1245–1248.

[26] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA: Abstraction for Advanced in-Database Analytics. *PVLDB* 11, 11 (2018), 1400–1413.

[27] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. 2006. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering* 19, 1 (2006), 1–16.

[28] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* 29, 1 (2020), 33–59.

[29] 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 (SIGMOD '18)*. 1603–1618.

[30] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT 2019: 22nd International Conference on Extending Database Technology*.

[31] Pedram Ghodsnia. 2012. An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads. 54–59.

[32] David F Gleich. 2015. PageRank beyond the Web. *siam REVIEW* 57, 3 (2015), 321–363.

[33] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*. ACM, 325–336.

[34] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast computation of database operations using graphics processors. In *SIGMOD*. ACM, 215–226.

[35] C. Guo and H. Chen. 2019. In-Memory Join Algorithms on GPUs for Large-Data. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 1060–1067.

[36] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34 (2009).

[37] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *SIGMOD*. 511–524.

[38] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *VLDB* 6, 10 (2013), 889–900.

[39] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib Analytics Library. *Proceedings of the VLDB Endowment* 5, 12 (2012).

[40] Pedro Holanda and Hannes Mühleisen. 2019. Relational Queries with a Tensor Processing Unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, Article 19, 3 pages.

[41] Yu-Ching Hu, Murtuza Taher Lokhandwala, Te I, and Hung-Wei Tseng. 2019. Dynamic Multi-Resolution Data Storage. In *52th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2019 (Best Paper Honorable Mention))*.

[42] Dylan Hutchison, Bill Howe, and Dan Suciu. 2017. LaraDB: A Minimalist Kernel for Linear and Relational Algebra Computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR'17)*. Article 2, 10 pages.

[43] Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery.

[44] Yanqin Jin, Hung-Wei Tseng, Steven Swanson, and Yannis Papakonstantinou. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *23th International Symposium on High Performance Computer Architecture (HPCA 2017)*.

[45] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ACM, 1–13.

[46] 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*. 55–62.

[47] Tomas Karnagel, René Müller, and Guy M Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@ VLDB* 8 (2015), 20.

[48] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking abstractions for GPU programs. In *OSDI*. 6–8.

[49] Kinetica DB Inc. 2016. Kinetica. https://www.kinetica.com/.

[50] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1197–1208.

[51] Gunjae Koo, Kiran Kumar Matam, Te I, Hema Venkata Krishna Giri Narra, Jing Li, Steven Swanson, Hung-Wei Tseng, and Murali Annavaram. 2017. Summarizer: Trading Bandwidth with Computing Near Storage. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2017)*. 219–231.

[52] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review* 25, Special Issue (1991), 63–74.

[53] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[54] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *PVLDB* 9, 14 (2016), 1647–1658.

[55] Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2016. Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 376–379.

[56] Yu-Chia Liu and Hung-Wei Tseng. 2021. NDS: N-Dimensional Storage. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2021 (Best Paper Nomination))*.

[57] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. 2019. Scalable Linear Algebra on a Relational Database System. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2019), 1224–1238.

[58] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1633–1649.

[59] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB* 11, 11 (2018).

[60] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[61] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. 2015. FrogWild! Fast PageRank Approximations on Graph Engines. *Proc. VLDB Endow.* 8, 8 (4 2015), 874–885.

[62] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. Association for Computing Machinery, 19–34.

[63] Rene Mueller and Jens Teubner. 2009. FPGA: What's in It for a Database?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, 999–1004.

[64] Mark Needham and Amy E Hodler. 2019. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j.* O'Reilly Media.

[65] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.

[66] NVIDIA. 2017. GPUDirect RDMA. https://developer.nvidia.com/gpudirect.

[67] OmniSci Inc. 2018. Open Source Analytical Database & SQL Engine. https://www.omnisci.com/platform/omniscidb.

[68] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*. 237–252.

[69] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the Latency of Data Processing Pipelines through FPGA Based Hardware Acceleration. *Proc. VLDB Endow.* 13, 1 (2019), 71–85.

[70] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.

[71] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

[72] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*. 1935–1950.

[73] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. 2013. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment* 7, 2 (2013), 121–132.

[74] Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast pagerank computation on a gpu cluster. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 450–456.

[75] Kaz Sato, Cliff Young, and David Patterson. 2017. An in-depth look at Google's first Tensor Processing Unit (TPU). *Google Cloud Big Data and Machine Learning Blog* 12 (2017).

[76] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. 1617–1632.

[77] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 1–35.

[78] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709.

[79] Evangelia A Sitaridi and Kenneth A Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–8.

[80] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-Based Analytics. *Proc. VLDB Endow.* 11, 13 (2018), 2168–2182.

[81] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. *Gullfoss: Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources.* Technical Report. UCSD Technical Report.

[82] P. Volk, D. Habich, and W. Lehner. 2010. GPU-Based Speculative Query Processing for Database Operations. In *ADMS@VLDB*.

[83] Slawomir Walkowiak, Konrad Wawruch, Marita Nowotka, Lukasz Ligowski, and Witold Rudnicki. 2010. Exploring utilisation of GPU for database applications. *Procedia Computer Science* 1, 1 (2010), 505–513.

[84] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

[85] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. 2017. MILC: Inverted List Compression in Memory. *Proc. VLDB Endow.* 10, 8 (4 2017).

[86] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD In-Storage Computing for Search Engines. *IEEE Trans. Comput.* (2016).

[87] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *VLDB* 7, 11 (7 2014), 1011–1022.

[88] Zeke Wang, Huiyan Cheah, Johns Paul, Bingsheng He, and Wei Zhang. 2016. Accelerating Database Query Processing on OpenCL-based FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 274–274.

[89] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO*. IEEE Computer Society, 107–118.

[90] Haicheng Wu, D. Zinn, M. Aref, and S. Yalamanchili. 2014. Multipredicate join algorithms for accelerating relational graph processing on GPUs. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*.

[91] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient PageRank and SpMV Computation on AMD GPUs. In *2010 39th International Conference on Parallel Processing*. IEEE, 81–89.

[92] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *VLDB* 6, 10 (2013), 817–828.

[93] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*. 273–283.

[94] Orestis Zachariadis, Nitin Satpute, Juan Góumez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix-matrix multiplication with GPU Tensor Cores. *Computers and Electrical Engineering* 88 (2020), 106848. https://doi.org/10.1016/j.compeleceng.2020.106848

[95] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *PACT*. IEEE, 13–24.

[96] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. 2015. Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources. *Journal of Computer Science and Technology* 30, 4 (2015), 657–678.

[97] Zach Zimmerman. 2016. MSplitGEMM: Large matrix multiplication in CUDA. https://github.com/zpzim/MSplitGEMM.