An Empirical Performance Comparison between Matrix Multiplication Join and Hash Join on GPUs

Wenbo Sun Asterios Katsifodimos Rihan Hai
Delft University of Technology
Delft, The Netherlands
{w.sun-2, a.katsifodimos, r.hai}@tudelft.nl

***Abstract*—Recent advances in Graphic Processing Units (GPUs) have facilitated a significant performance boost for database operators, in particular, joins. It has been intensively studied how conventional join implementations, such as hash joins, benefit from the massive parallelism of GPUs. With the proliferation of machine learning, more databases have started to provide native support for the basic building blocks of ML algorithms, i.e., linear algebra operators such as matrix multiplication (MM). Despite the recent increasing interest in processing relational joins using matrix multiplication (MM-join), two crucial questions still remain open: i) how efficient are current MM-join implementations compared to the GPU-based join algorithms; ii) how should practitioners choose among MM-join and conventional GPU-based joins given different data characteristics.**

In this paper, we compare the execution time, and memory I/O of MM-join against multiple GPU hash joins. An empirical analysis of our experimental results reveals that the state-of-the-art hash join implementation shows substantial scalability for various data characteristics. In contrast, MM-join outperforms the SOTA hash join in low join selectivity and low table cardinality but shows unsatisfactory scalability due to synchronous data movement and computation.

***Index Terms*—Matrix Multiplication Join, GPU, Hash Join**

I. Introduction

The rapid advancement of GPUs' massive parallelism and high memory bandwidth has been the strong driving force for GPU-accelerated relational query processing. Joins are among the most useful, yet expensive operations in relational databases. Together with the evolving GPU architectures, intensive research effort [9, 18, 14, 10, 20, 22] has driven significant speedups for join operations over GPUs.

At the moment, many database vendors seek solutions to natively support ML operators (i.e., linear algebra) in databases [6, 5]. However, the mixing relational and LA operators in ML applications introduces diverse data structures and software stacks, causing considerable maintenance costs and data transformation overhead. As such, a unified data representation and operators are highly desirable for ML practitioners. As the most common LA operator in ML workloads, matrix multiplication (MM) has gained significant speedups on GPUs [16], making it promising for bridging the gap between ML and relational query processing. Lately, MM has been shown

This work is co-funded by the European Union Horizon Programme call HORIZON-CL4-2022-DATA-01, under Grant Agreement No. 101093164 (ExtremeXP).

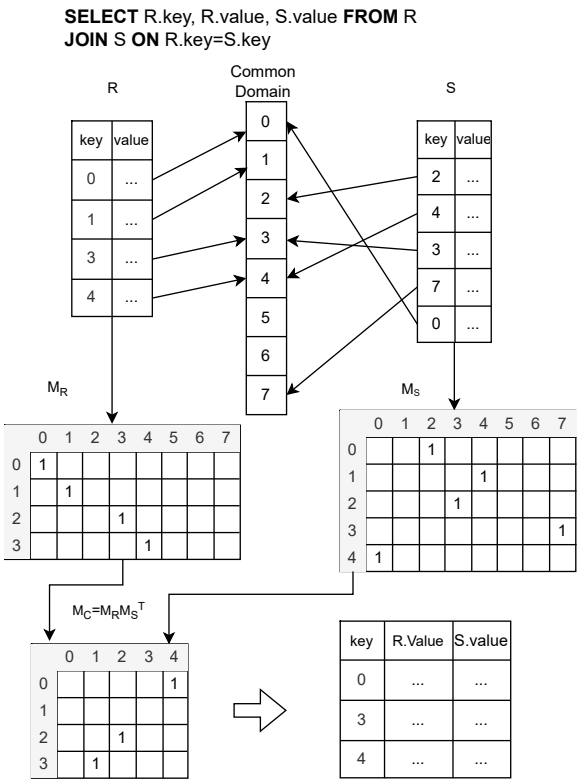


Fig. 1: Example join query and MM-join implementation

capable of relational query processing, such as equi-join, semi-join, and group-by aggregations [1, 7, 12, 11]. Fig. 1 illustrates an example of evaluating join through matrix multiplication (MM-join). MM-join encodes relations as adjacency matrices where keys are mapped to a common domain. The relationship between tuples can be found through matrix multiplication over binary sparse matrices.

CPU-based MM-join is not a good match for data-intensive applications, due to the overhead of data transformation and high computational complexity. The recent advances in GPU have alleviated these problems and facilitated MM-joins as a promising implementation of join algorithms. The dedicated numeric instructions and hardware accelerators on modern GPUs contribute substantial speedup to linear algebra (LA) operators like spMM. The spMM kernel tested on Nvidia A100

[16] can achieve a 25x speedup to its CPU counterpart. From the view of the programming model, branching instructions for tuple matching in hash joins introduce expensive thread divergence [4]. In contrast, simple algebraic instructions in MM-join can benefit from data prefetching [21] and efficient caching [8].

In this work, we focus on the discussion over hash joins, which have been shown to be highly effective on both CPUs and GPUs in previous studies [19, 9, 18]. Recent work *TCUDB* [11] has reported that in certain cases, the GPU-based implementation of MM-join outperforms a GPU hash join algorithm[22]. Hence, MM-join seems to be a promising option to implement join algorithms over GPU hardware. However, none of the existing works provide a systematic investigation on the cases where MM-join is faster. Therefore, in this paper, we attempt to answer the following research question:

When do MM-joins outperform GPU hash joins?

More specifically, we are interested in exploring how data characteristics such as cardinality, skewness, and join selectivity affect the relative performance of MM-join as compared to GPU hash joins.

Contribution. This paper reports on an empirical performance comparison between MM-joins and GPU hash joins in investigating the usability of MM-join given diverse data characteristics. Our contributions can be summarized as follows:

- We choose an implementation of MM-join and four representative GPU hash join algorithms (Table I), and compare their execution time given varying values of dataset cardinality, skewness, and join selectivity.
- We evaluate the memory usage and I/O of MM-join regarding a wide range of data characteristics, and show how memory I/O affects execution time.
- We implement the MM-join using Pytorch tensor primitives to evaluate the performance of MM-joins in Python environment - a very common choice of practitioners for data science workloads. [1]
- We validate the performance of MM-join on real-world datasets and observe that MM-join is more suitable for small-scale data with low selectivity.

II. Join Approaches

In this section, we elaborate on the MM-join, and briefly explain four representative hash join implementations. Table I gives an overview of their characteristics.

A. Matrix Multiplication Joins

MM-joins essentially represent relations as adjacency matrices under a common key domain and join tuples using matrix multiplication, which have been widely used in graph query processing like graph traversal. Recent works [3, 7, 12] also propose MM-based methods to solve set intersection and relational join processing.

[1]We plan to release our code soon as open-source.

[2]https://github.com/rapidsai/cudf

[3]https://thrust.github.io/

TABLE I: Overview of join algorithms to be evaluated

Algorithm	Approach	Description	Library
Hash Join	CLASSIC [9]	Radix hash join	CUDA
	PW[20]	Warp execution; bucket-chain pool	CUDA
	AMHJ [14]	Cooperative groups; load balancing	CUDA
	TQP [10]	Radix hash join	Pytorch + CUDA
	cuDF[2]	Partitioned hash join	Thrust[3] + CUDA
MM-join	TCUDB [11]	MM-join	CUDA
	TCU_MM	MM-join	Pytorch + CUDA

Algorithm 1: Matrix Multiplication Join

Input : R,S: input relations
Output: $pairs$: matched index pairs

1 C_R, C_S = len(R), len(S)
2 //CUDA reduce
3 R_{max}, S_{max} = max(key_R), max(key_S)
4 K = max(R_{max}, S_{max})
5 key_dict = dict(zip(union(key_R, key_S), range(0, K))
6 $rows_R$ = range(0, C_R), $rows_S$ = range(0, C_S)
7 $columns_R$ = 0, $values_R$ = 1
8 $columns_S$ = 0, $values_S$ = 1
9 //CUDA parallel
10 **for** $i \in [0, C_R)$ **do**
11 $\quad columns_R$[i] = key_dict[key_R[i]]
12 **end**
13 //CUDA parallel
14 **for** $i \in [0, C_S)$ **do**
15 $\quad columns_S$[i] = key_dict[key_S[i]]
16 **end**
17 M_R = cuda_construct_CSR($rows_R$, $columns_R$, $values_R$)
18 M_S = cuda_construct_CSR($columns_S$, $rows_S$, $values_S$)
19 M_C = cuda_sparse_multiplication(M_R, M_S).to_COO()
20 $pairs$ = list(zip(M_C.rows, M_C.columns))
21 **return** $pairs$

MM-join process. We illustrate the process of matrix multiplication join with the pseudo-code in Algorithm 1, which has four steps. We explain Algorithm 1 with the running example in Fig. 1.

1) We first calculate the maximum key value in R and S to construct the common domain (Lines 1-5), resulting in $[0, 7]$;
2) Then we fill non-zero values and positions in CSR format to get M_R and M_S (Lines 6-18). The column indexes of the matrices are identical to the keys' positions in the common domain, and the row indexes are the row numbers of keys in original relations;
3) We execute spMM over M_R and transposed M_S (Line 19);
4) Finally, we extract row and column numbers of non-zero values in M_C as the join results (Line 20). The row and column indexes of M_C are row numbers of matching tuples in R and S, respectively.

Complexity analysis. The high computational complexity and memory consumption have hindered the application of MM-joins in CPU-based databases. In Algorithm I, transforming

relations to matrices requires extra time and memory space based on the number of tuples and distinct keys, which is infeasible for relations with billions of rows. Even though the CSR format can reduce memory usage, the computational complexity of sparse matrix multiplication (spMM) can not be further reduced: the best-known complexity of spMM is $O(n^2)$[4] [23], which is higher than $O((|R| + |S|) * log(|R|))$ of a radix hash join algorithm [2], where $|R|$ and $|S|$ represent the cardinalities of the two tables participating the join.

TCUDB [11]. As discussed in Section I, the recent advances in GPU have alleviated these problems and facilitated MM-joins as a promising implementation of join algorithms. TCU is the first GPU query engine using pure LA operators to evaluate relational queries including selection, projection, equi-join, and aggregations. This work was inspired by the highly efficient tensor core (TCU) [15], hardware accelerators for matrix computations.

B. Parallel Hash Joins

A parallel hash join algorithm contains three phases: *partition*, *build*, and *probe*. First, in the *partition* stage, the original relations are first sliced into segments according to the available parallelism. Second, each thread *builds* a local partition which then will be merged with other partitions, resulting in the global partitions. Third, threads subsequently build a hash table for each partition and locate matching pairs by *probing* tuples in the hash tables. Existing approaches [9, 22, 18, 20] for GPU hash join implementations, facilitate performance optimization through utilizing evolving GPU architecture features. In the following, we explain the GPU-based optimization techniques in each approach and their reported performance boost.

CLASSIC [9] proposes six parallel primitives to implement four classic joins (hash join, sort-merge join, indexed nested loop join, non-indexed nested loop join) on GPUs. The GPU implementations utilize coalesced memory access and shared memory besides the high parallelism of GPU, gaining up to a 7x speedup compared with their CPU counterparts.

PW [20] The conventional parallel hash join needs global synchronizations between *partition*, *build*, *probe* phases. Such a coarse-grained data dependency prohibits overlapping data movement and computation, introducing considerable memory stall. To overcome the large memory stall caused by global synchronization, PW pre-allocates a *reusable memory pool* containing a chain of buckets for thread-local storage. When a bucket is fully filled, the subsequent data are stored in the succeeding empty bucket, and the filled bucket can be concurrently used in the subsequent phases. Although the pre-allocated pool requires more memory space on average, the fine-grained buckets in the pool break the global data dependency and pipeline the three phases in hash joins, significantly reducing the memory stall. The optimization brings up to 10x speedup compared to CLASSIC.

AMHJ [14] Apart from the techniques in PW, AMHJ utilizes *cooperative groups* introduced by CUDA 9.0 for more flexible thread partitioning and execution. In this way, threads in a block can be further partitioned according to the number of data segments, reducing the negative impact of uneven data distribution in the hash partitioning stage. Moreover, with the assistance of *dynamic parallelism*, the work-sharing strategy is applied to mitigate the imbalanced workloads of each warp group. The evaluation in [14] shows AMHJ outperforms PW by up to 9x with real-world multi-way join tasks. However, AMHJ is designed for multi-way joins. We will re-evaluate the speedup in two-way join tasks.

TQP [10]. The tensor query processor (TQP) is a query processing engine built on tensor computing runtime (TCR) designed for intensive algebraic computing. Popular TCR implementations like Pytorch [17] provide abundant parallel primitives. TQP supports fully functional relational operators, including radix hash join and sort-merge join. The radix hash join in TQP is a naive implementation. Due to the nature of algebraic operators, TQP uses multiple tensor primitives to implement equivalent item-wise operators in hash join, such as scan and filter. The extra time consumption of composite operators deteriorates the performance compared to dedicated GPU hash joins.

C. Comparison goals

MM-join outperforms GPU hash join algorithms in certain cases, as reported in recent studies [11, 3]. However, the comparison was not conducted between MM-join and the state-of-the-art hash join algorithms, i.e., PW, AMHJ. Thus, in Section III, we extensively compare the performance of MM-join with representative GPU-based parallel hash joins approaches in Section II-B. TCUDB can process both two-way and multi-way natural joins over sparse matrix representations. As an early work, we only evaluate the two-way natural join because it is the most fundamental join operation.

For a fair comparison, we make the following remarks: i) conventional hash join algorithms were not designed for execution over tensor cores. Thus, in this work we focus on the evaluation of MM-join in TCUDB with CUDA cores. i) TQP is implemented in Python. To compare the MM-join of TCUDB with TQP, besides the CUDA-native TCUDB, we have also developed a Python-based MM-join following Algorithm 1, i.e., TCU_MM in Table I.

III. Evaluation

We evaluate the performance of join algorithms in Table I w.r.t. execution time and memory Below we first introduce the experimental settings, including a large variety of real and synthetic datasets. The experiment results are then presented together with an empirical analysis of performance with respect to different parameter combinations.

[4]The complexity of spMM depends on matrix shapes and sparsity. Here we use an approximate value to show the complexity gap between spMM and hash join.

TABLE II: Characteristics of dataset in the evaluation

Dataset	$\|R\|$	ratio	selectivity	z
synthetic	$2^{12...21}$	[0.1, 0.9]	[0.1, 0.9]	[0, 1.0]
company_year	1,901,021	0.29	0.09	0.82
cast_movie	1,832,199	0.32	0.50	0.02

A. Experimental settings

We focus on the performance in two-way join tasks without materialization. All experiments are carried out on an NVidia GeForce RTX3060 with 6GB global memory. Each measurement is conducted with ten times repetition.

Parameters. To test the impact of different data characteristics on performance, we first evaluate the algorithms considering four parameters: cardinalities of the two relations to be joined ($|R|$ and $|S|$), cardinality ratio of right and left relations ($ratio = \frac{|S|}{|R|}$), selectivity (sel), skewness of the left relation (z, shape parameter of Zipf Distribution).

Synthetic datasets. We have implemented a data generator based on the aforementioned four parameters. It first produces integers as left keys according to cardinality and z, then uses $ratio$ and $selectivity$ to determine the range of right keys. The ranges of parameters produce 1375 pairs of relations.

Real datasets. In addition to the synthetic data, we selected two pairs of real-world relations from the IMDB dataset [13] for evaluating the usability of TCUDB. Namely, we choose cast_movie that contains cast_info and movie_info tables; company_year covers movie_companies and movie_title tables. The data characteristics are listed in Table II.

B. Evaluation on synthetic data

The performance of a join algorithm is not only related to its theoretical complexity but also affected by the characteristics of datasets, such as skewness, and selectivity. In this section, we will use synthetic datasets covering different data characteristics to test the performance of join algorithms in Table I.

1) Execution Time: We evaluate the execution time of four CUDA-native implementations in Table I on synthetic datasets, i.e., CLASSIC, PW, AMHJ, and TCUDB.

Q_1 : *For what cardinalities does TCUDB perform the best?*

Fig. 2 shows the mean and standard error of execution time regarding different $|R|$. PW and TCUDB remarkably outperform the AMHJ and CLASSIC in terms of both stability and performance. In the following analysis, we will focus on the comparison between PW and TCUDB.

Observation & Analysis. TCUDB outperforms PW when $|R| < 2^{18}$. The results of TCUDB when $|R| > 2^{20}$ are missing because its memory usage exceeds memory capacity. Moreover, the total volume of data in the join operation is determined by both $|R|$ and $|S|$. Since we use $ratio$ to control $|S|$, we present a heatmap in Fig. 3 to demonstrate

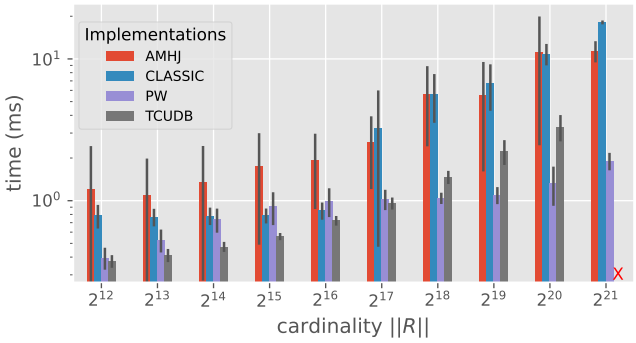


Fig. 2: Average execution time under various $|R|$

to speedup of TCUDB to PW under different cardinality. The peak speedup at different R varies to $ratios$. As $ratio$ in combination with $|R|$ reflects $|R| + |S|$, we conclude that the peak speedup of TCUDB is related to *total cardinality*.

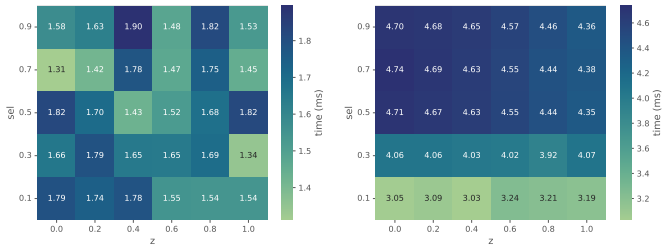
Q_2 : *How do selectivity and skewness affect the performance of TCUDB?*

We now turn to examine the impact of skewness and selectivity on the performance of different approaches. Fig. 4a and 4b show the execution time of PW and TCUDB with different skewness-selectivity combinations respectively.

Observation. Fig. 4a does not show a clear correlation between the performance of PW and skewness, whereas TCUDB performs significantly faster when $sel \in [0.1, 0.3]$, and shows slight speedup when z increases.

Analysis. This specific behavior of TCUDB to selectivity is the result of an important difference: hash join needs to compare every pair of tuples to determine whether there is a match or not; MM-based TCUDB, on the other hand, only computes non-zero values in sparse matrices. Therefore, the selectivity determines the actual amount of computation. The data skewness principally determines the balance of partitions in hash join, but many recent implementations, including PW,

Fig. 3: Speedup heatmap regarding $|R|$ and $|S|$



(a) PW (b) TCUDB

Fig. 4: Execution time regarding selectivity and skewness

perform optimizations to mitigate the negative implication of unbalanced partitions. As a result, we can not identify a significant relationship between skewness and execution time. TCUDB, on the other hand, is more sensitive to data skewness because high skewness means values in matrices are spread in multiple clusters. The spMM kernel can skip regions with zero values, further reducing the workload. However, this effect is not as significant as selectivity.

2) Memory footprint and I/O: Through the analysis of execution time, we discover that skewness and selectivity have a remarkable impact on the performance of TCUDB. Because skewness and selectivity serve as the primary determinant of data layout and memory I/O in the spMM kernel, we analyze the differences between PW and TCUDB regarding memory I/O to pinpoint the root cause of performance differences.

Q_3 : *How does the cardinality affect memory usage and I/O of TCUDB and PW?*

Memory footprint. Due to the GPU memory limitation, the peak memory usage of an in-memory algorithm determines whether the algorithm can be executed successfully. Fig. 5 shows the peak memory footprint of PW and TCUDB. The chained-buckets data structure of PW requires pre-allocated memory space, so PW has a high memory footprint, even with low cardinality. Sizes of sparse matrices in TCUDB are related to cardinality and sparsity, so we can see the increasing trend as cardinality increases. The 2-dimensional matrix representation of relations has higher storage complexity. Due to the non-partitionable data structures in TCUDB, the memory usage exceeds the device capacity when $|R| > 2^{20}$.

Memory I/O. Similar to the memory footprint, the total memory I/O of TCUDB in Fig. 6 also increases with cardinality, resulting to greater execution times, due to the increasing number of memory operations. Notably, PW still performs better with more memory I/O when $|R| > 2^{17}$. Apart from the gap of algorithmic complexity, concurrent data transfer between the host and GPU also helps PW gain superior performance. Concurrent data transfer can move the data required for *build* and *probe* in small segments, enabling overlapping computation and data transfer and reducing the memory stall. In contrast, the spMM kernel in TCUDB needs to wait for

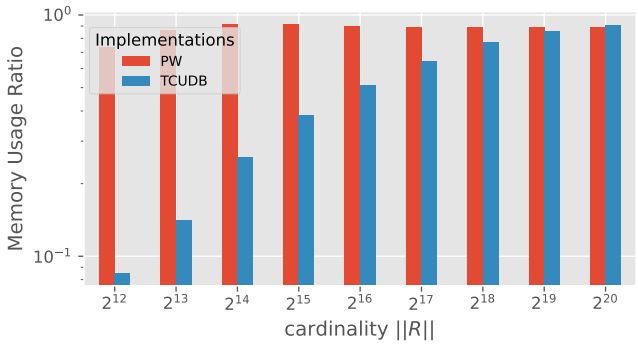


Fig. 5: Peak memory usage of PW and TCUDB

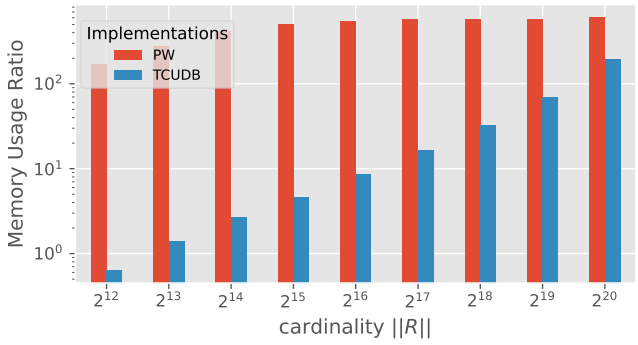


Fig. 6: Memory I/O measured in MB

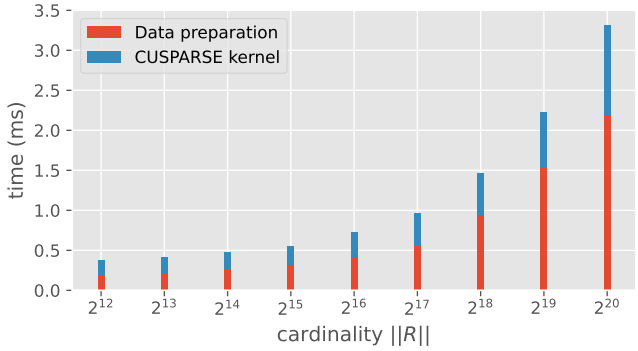


Fig. 7: Performance breakdown of TCUDB. Data preparation means data transformation from relations to sparse matrices.

sparse matrices to be fully built and filled (line 1-17 in Alg. 1). In Fig. 7 we can observe that the memory preparation time dominates the total execution time. Lack of data-computation concurrency results in the inferior performance of TCUDB when joining large relations.

Q_4 : *How do* selectivity *and* skewness *affect memory I/O?*

In the execution time experiments (Fig. 4b), TCUDB performs better with low selectivity. We now investigate the relationship between memory I/O and selectivity-skewness combinations in Fig. 8. The memory I/O is less for small

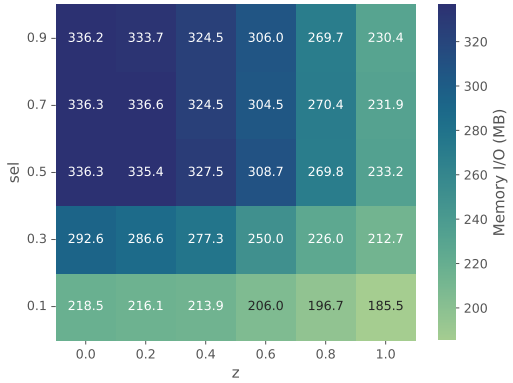


Fig. 8: Mem. I/O of TCUDB varying selectivity & skewness.

selectivities $sel \in [0.1, 0.3]$. Together with Fig. 4b, we conclude that, for TCUDB, low selectivity and high skewness indicates less computation and memory I/O, implying high speedup compared with the SOTA GPU hash join.

Q_5 : How do TCU_MM, TQP and cuDF perform, given varying values of $|R|$?

Python-based implementations. Next, we compare the performance of python-compatible GPU-based join implementations, i.e., TCU_MM, TQP, and cuDF. TQP uses the same PyTorch primitives as TCU_MM, while cuDF uses thrust, a general-purpose parallel primitives library. Pytroch primitives are designed for parallel algebraic computation, lacking effective support for iterative computation. As a result, TQP combines multiple primitives to implement iterative operations, such as scan, filter, etc. Such an indirect method involves significant overhead compared to native implementations.

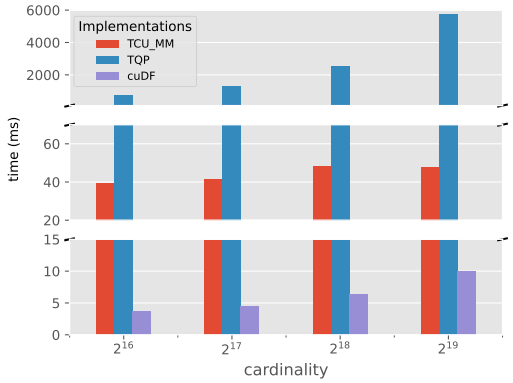


Fig. 9: Average execution time of Python implementations.

Observation & Analysis. In Fig. 9, TQP is substantially slower than the other two methods. Pytorch primitives can efficiently compute sparse matrices in TCU_MM. The high algorithmic complexity and memory stall of TCU_MM are even higher in their Python counterpart, due to an additional translation layer between Python and CUDA, making TCU_MM inferior to the well-optimized hash join in cuDF.

C. Evaluation on real-world data

In order to validate the performance of TCUDB in practical scenarios, we use a number of relations (Table II) in IMDB dataset to evaluate the execution time of PW and TCUDB. Since TCUDB currently does not support some data types, such as text or date, we only examine categorical attributes.

Q_6 : How does TCUDB perform on real-world datasets?

Observation & Analysis. In Fig. 10, TCUDB performs better in company_year dataset, whereas it exhibits performance degradation with cast_movie data. The company_year data has low total cardinality and selectivity within $[0.1, 0.3]$. Although the cast_movie contains approximately equivalent $|R|$, the large $ratio$ makes the total cardinality fall out of the suitable range for TCUDB. Combined with the execution time analysis of the previous section, TCUDB presents satisfactory speedup in synthetic data with similar selectivity and skewness. These observations validate our analysis that TCUDB performs faster with low selectivity and high skewness. Thus, we reach a preliminary discovery. That is, to some extent, MM-join has been proven to be a good choice of join implementation for practical data science tasks when *low selectivity is low and skewness is high*.

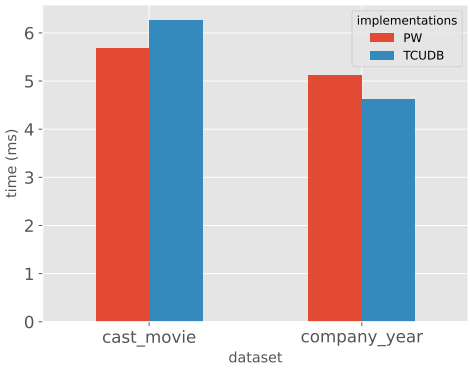


Fig. 10: Execution time tested on real-world datasets.

IV. Conclusion

In this work, we explored the applicability of MM-join through an extensive performance comparison between MM-join and multiple representative GPU hash join implementations. We have used synthetic data to test the impact of selectivity and skewness in both compute performance and memory I/O. We drew the following conclusions from our experiments on the given hardware:

- PW exhibits substantial scalability and stability over various data characteristics.
- MM-join presents superior performance in real-world data processing tasks with low cardinality ($|R| < 2^{18}$)

and selectivity ($sel < 0.3$) but is not the best choice for large data.
- The 2-dimensional matrix representation of relations has higher storage complexity, and the synchronized execution of data movement and computation introduces considerable memory stalls, causing performance degradation at high cardinalities.

Future research. Compared to the well-studied GPU hash join, MM-Join calls for further research. For example, its unsatisfactory performance due to sequential data movement can be improved through MM-Join over multiple sliced inputs, which resembles partitions in hash joins. By implementing a block-wise spMM kernel [24], it is possible to overlap the data movement and computations over disjoint blocks of matrices. Additionally, the intersection of machine learning and databases can benefit from global optimizations across both data processing and ML models through a unified mathematical representation.

REFERENCES

[1] Rasmus Resen Amossen and Rasmus Pagh. "Faster Join-Projects and Sparse Matrix Multiplications". In: *ICDT*. 2009, pp. 121–126.

[2] Cagri Balkesen et al. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *ICDE*. 2013, pp. 362–373.

[3] Christos Bellas. "Advanced Joins on GPUs". In: *PhD Thesis* (2022).

[4] Piotr Bialas and Adam Strzelecki. "Benchmarking the cost of thread divergence in CUDA". In: *PPAM*. 2016, pp. 570–579.

[5] Matthias Boehm et al. "SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle". In: (2020).

[6] Paul G Brown. "Overview of SciDB: large scale array storage, processing and analysis". In: *SIGMOD*. 2010, pp. 963–968.

[7] Shaleen Deep, Xiao Hu, and Paraschos Koutris. "Fast Join Project Query Evaluation Using Matrix Multiplication". In: *SIGMOD*. 2020, pp. 1213–1223.

[8] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication". In: *SIGGRAPH/EUROGRAPHICS*. 2004, pp. 133–137.

[9] Bingsheng He et al. "Relational joins on graphics processors". In: *SIGMOD*. 2008, pp. 511–524.

[10] Dong He et al. "Query processing on tensor computation runtimes". In: *VLDB Endowment*. Vol. 15. 11. 2022, pp. 2811–2825.

[11] Yu-Ching Hu, Yuliang Li Li, and Hung-Wei Tseng. "TCUDB: Accelerating Database with Tensor Processors". In: *SIGMOD*. 2022.

[12] Zichun Huang and Shimin Chen. "Density-Optimized Intersection-Free Mapping and Matrix Multiplication for Join-Project Operations". In: vol. 15. 10. VLDB Endowment, 2022, pp. 2244–2256.

[13] *IMDB datasets*. URL: https://www.imdb.com/interfaces/.

[14] Zhuohang Lai et al. "Accelerating multi-way joins on the GPU". In: *VLDB Endowment*. Vol. 31. 3. 2022, pp. 529–553.

[15] Stefano Markidis et al. "NVIDIA Tensor Core Programmability, Performance &amp Precision". In: *IPDPS*. May 2018.

[16] Nvidia. *Nvidia A100 Architecture Whitepaper*. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[17] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *NeurIPS*. Vol. 32. 2019.

[18] Ran Rui and Yi-Cheng Tu. "Fast equi-join algorithms on GPUs: Design and implementation". In: *SSDBM*. 2017, pp. 1–12.

[19] Stefan Schuh, Xiao Chen, and Jens Dittrich. "An experimental comparison of thirteen relational equi-joins in main memory". In: *SIGMOD*. 2016, pp. 1961–1976.

[20] Panagiotis Sioulas et al. "Hardware-conscious hash-joins on gpus". In: *ICDE*. 2019, pp. 698–709.

[21] Da Yan, Wei Wang, and Xiaowen Chu. "Demystifying tensor cores to optimize half-precision matrix multiply". In: *IPDPS*. 2020, pp. 634–643.

[22] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. "The Yin and Yang of processing data warehousing queries on GPU devices". In: *VLDB Endowment* 6.10 (2013), pp. 817–828.

[23] Raphael Yuster and Uri Zwick. "Fast sparse matrix multiplication". In: *TALG*. 2005, pp. 2–13.

[24] Orestis Zachariadis et al. "Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores". In: *Computers & Electrical Engineering* 88 (2020), p. 106848.