GPU-Accelerated SQL Joins

Aryan Kumar
MIT EECS

Kenneth Choi
MIT EECS

Siddhant Mukherjee
MIT EECS

Tasmeem Reza
MIT EECS

ABSTRACT

Relational joins are essential to SQL-based data processing, playing a key role in applications such as analytics and database management. Recently, there has been an increasing presence of data-intensive workloads; however, GPU computation has also increased in availability. As a result, optimizing join algorithms for GPU architectures has become a critical research focus. GPUs provide significant parallelism and memory bandwidth, making them suitable for these operations, yet challenges such as materialization costs and workload distribution persist.

We build on the work of Wu et al. [7], who proposed an optimized framework for processing large relational joins on GPUs. Their Gather-From-Transformed-Relations (GFTR) technique reduces materialization costs by leveraging sequential memory accesses, outperforming the traditional Gather-From-Untransformed-Relations (GFUR) approach. We implement and benchmark four join algorithms—Sort Merge Join and Partitioned Hash Join, each with and without GFTR optimization—under diverse workloads characterized by varying join width, input relation sizes, and data skewness. Our results validate the state-of-the-art improvements of the GFTR framework, demonstrating significant throughput gains. We also discuss various trade-offs in Sort Merge Join and Partitioned Hash Join implementations. Overall, we enhance the understanding of GPU-accelerated database operations, contributing insights that facilitate the design of efficient query engines for GPU-enabled environments.

PVLDB Reference Format:
Aryan Kumar, Kenneth Choi, Siddhant Mukherjee, and Tasmeem Reza. GPU-Accelerated SQL Joins.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/kenctrl/relational-joins-cuda.

1 INTRODUCTION

Relational join operations are fundamental to SQL-based data processing, playing fundamental roles in applications such as complex analytics and database management. As the world increasingly moves towards data-intensive applications and produces more GPUs, optimizing join algorithms for GPU architectures has become a critical research area. GPUs offer significant parallelism and memory bandwidth, making them ideal for data-intensive relational operations such as SQL joins. However, efficient implementation of these operations on GPUs remains a challenge due to tradeoffs like data materialization costs and workload distribution under varying conditions.

In this paper, we draw inspiration from the work of Wu et al. [7] on efficiently processing large relational joins on GPUs. Their proposed framework integrates three GPU-optimized primitives—radix-partition, sort-pairs, and gather—with various join strategies. More specifically, their novel Gather-From-Transformed-Relations (GFTR) technique significantly reduces materialization costs compared to the conventional Gather-From-Untransformed-Relations (GFUR) approach by leveraging sequential memory accesses.

Our goal is to validate Wu et al.'s [7] conclusions by implementing and benchmarking their four distinct join algorithms: Sort-Merge Join with Unoptimized Materialization (SMJ-UM), Sort-Merge Join with Optimized Materialization (SMJ-OM), Partitioned Hash Join with Unoptimized Materialization (PHJ-UM), and Partitioned Hash Join with Optimized Materialization (PHJ-OM). We control for key factors that influence performance, such as join width, match ratios, and data skewness. Like Wu et al.'s [7] paper, we focus on inner joins.

Through our work, we aim to provide a comprehensive analysis of GPU-based join algorithms, highlighting trade-offs and optimizations that maximize throughput under differing conditions. We not only replicates state-of-the-art results but also extend the understanding of GPU-accelerated database operations, paving the way for more efficient database query engines in environments where GPUs are available.

2 BACKGROUND

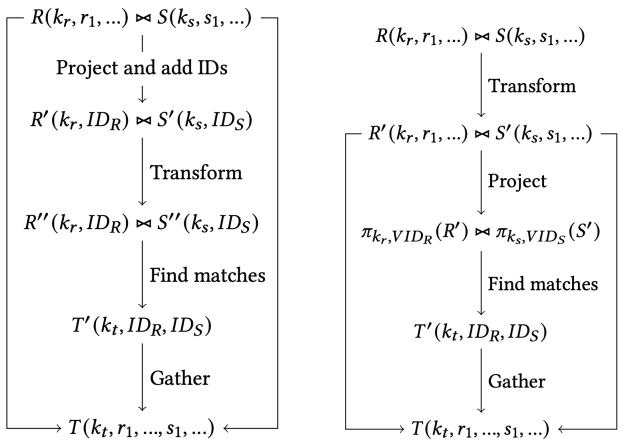
Sort-Merge Join (SMJ) and Partitioned Hash Join (PHJ) each involve three phases: transformation, match-finding, and materialization. During transformation, keys are reordered (e.g., sorted or partitioned) to prepare for efficient matching. In the match-finding phase, these keys are compared to identify matches. Finally, the materialization phase completes the join by writing the matched keys and their associated payloads to the output.

We explore two different ways to techniques for materialization: Gather-From-Untransformed-Relations (GFUR) and Gather-From-Transformed-Relations (GTUR), shown in Figure 1.

2.1 Gather-From-Untransformed-Relations (GFUR)

In traditional implementations, the materialization phase operates on the original (untransformed) relations. This is known as Gather-From-Untransformed-Relations (GFUR). GFUR maintains the original layout of payload data during the transformation and match-finding phases. Materialization relies on "gather" operations, which

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.



(a) *GFUR* pattern (old). **(b) *GFTR* pattern (new).**

Figure 1: GFUR and GFTR patterns (Wu et al. [7]).

copy over payload data from the untransformed input relations based on the indices of matching keys.

GFUR, while relatively simple, suffers from significant inefficiencies due to random memory access patterns. The untransformed keys often correspond to scattered indices in the original relations, leading to uncoalesced memory accesses and high cache miss rates during materialization. GPUs are optimized for sequential memory accesses, so the random nature of these gathers introduces substantial overhead, making GFUR materialization a bottleneck in GPU join performance. The natural optimization would be to make these memory accesses during materialization as sequential as possible, which motivates a new technique.

2.2 Gather-From-Transformed-Relations (GFTR)

To address the inefficiencies of GFUR, Wu et al. [7] introduced the Gather-From-Transformed-Relations (GFTR) technique. In GFTR, the transformation phase is extended to include payload columns alongside keys, producing fully transformed relations after transformation. The materialization phase then operates on these fully transformed relations, ensuring that gather operations access data sequentially. This approach significantly reduces cache misses and enables highly coalesced memory accesses.

The inefficiencies of GFUR emerge mostly in real-world scenarios where match ratios are high, payload columns are wide, or data is skewed. These conditions exacerbate the performance penalties associated with uncoalesced memory accesses. GFTR addresses these challenges by restructuring the materialization process to take advantage of sequential memory access patterns.

We focus on implementing and benchmarking join algorithms that utilize both GFUR and GFTR patterns, demonstrating the advantages of GFTR under varying workload conditions and validating its effectiveness in addressing the materialization bottleneck.

3 OPTIMIZED JOIN IMPLEMENTATIONS

3.1 Sort Merge Join

We implement the Sort-Merge-Join (SMJ) algorithm mentioned in Wu et al. [7]. We begin by sorting the keys of both SQL tables, making use of the cub::DeviceRadixSort primitive like the paper does. This corresponds to the transformation phase. We then efficiently merge the two sorted lists of keys using an approach described in Green et al. [1]. In Figure 2, the two arrays we are merging are shown in the top and on the left. The merge path (depicted in orange) is a path from the top left to the bottom right of the grid, by moving only right or down at each cell, and corresponds to the ordering of a sequential merge. The merge path can also be viewed as the boundary between the 1's and 0's in the grid, where a cell with row i and column j is filled in with a 1 if and only if $A[i] \leq B[j]$.

The merge path algorithm described in the paper works in two phases: a partitioning step and a merge step. In the partitioning step, we partition the merge path by finding points on it at regularly spaced intervals. We do this by shooting a cross-diagonal at regular intervals (shown in black) and finding its intersection with the merge path through binary search. All the cells diagonally to the bottom-left of this intersection points are 1s and all the cells diagonally to the top-right of it are 0s. This gives us points at regular intervals on the merge path. We then enter the merge phase where we perform sequential merges on each section of the merge path in parallel. For instance, given the points $(2, 4)$ and $(7, 6)$ on the merge path, where the first index corresponds to array A and the second corresponds to array B, we would need to merge the 2nd to 6th elements (inclusive) of array A with the 4th to 5th elements (inclusive) of array B. This step is done in parallel on each of the merge path partitions.

In our implementation of merge path for joins, we mostly follow the paradigm above. For the partition step, we launch a kernel where each thread is responsible for finding the intersection of a cross-diagonal with the merge path (via binary search). This result then gets stored in memory. We then perform additional kernel launches for the merge step. The first kernel launch has each thread perform a sequential merge on a section of the merge path. We do not load into shared memory, since each thread works on a different section of the merge path, so there is not much data re-use. The result of this kernel launch is the merged array, which is stored in memory. Then, we perform a kernel launch that has each thread count the number of key-matches on a section of the merged array, checking adjacent elements to see if their key matches. We use thrust::inclusive_scan to compute a prefix sum of this. Finally, we do a final kernel launch that each thread writes the key-matches (in a section of the merged array) to the appropriate index in the output SQL table (along with information about the key's index in the SQL table/array).

This initial implementation of merge did perform significantly better than a sequential merge implementation. However, it was still roughly 50 times slower than Wu et al. [7]'s implementation of the merge step. We noticed our implementation was performing a significant amount of memory accesses (including writing the merged array to memory). We tried improving this bottleneck by adding `const` and `restrict` keywords into function arguments,

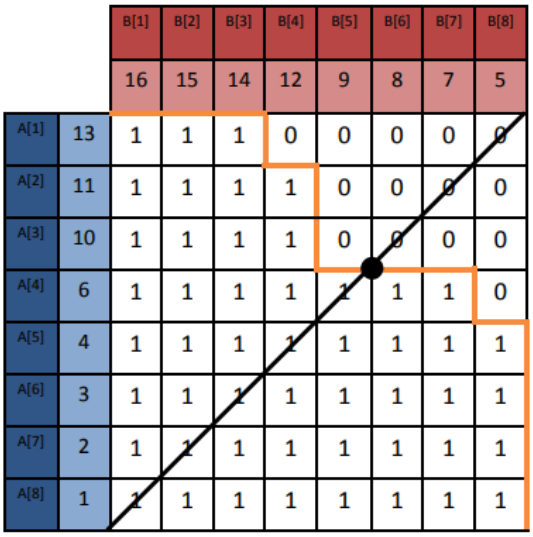


Figure 2: Diagram depicting the merge path (Green et al. [1]).

resulting in `__ldg` and loads from L1 cache being inserted by the compiler. This gave a significant speed-up. We also did not need to actually write the merged-array to memory, since our goal is to just find matched-keys, so we removed excessive memory writes like this. We also refactored our code to reduce the number of control-flow statements, and did some parameter tuning (e.g. on the number of threads/blocks to launch). The result was a merge path implementation that gets around 5-8x of the paper's implementation. The paper's implementation of the merging phase of the merge path algorithm seemed to implement the same underlying idea as our implementation, but it used more advanced techniques like async memory copies and inter-leaving memory reads with computation. While we ideally wanted to further optimize our merge, we also needed to implement the rest of SMJ with GFTR and GFUR, PHJ with GFTR and GFUR, test our code for correctness, and benchmark the results which was time-consuming.

Finally, in the last step of SMJ (materialization), we need to copy the payload columns from the two original SQL table into the new SQL table. We use the `thrust::gather` primitive for this, just like the paper does. Depending on whether we are implementing GFTR or GFUR, our implementation slightly differs. In GFTR's materialization phase, we perform gather on the sorted table. More specifically, we sort each column by the table key (using cub's `SortPairs`) and then gather on this column, using a map specified on the sorted keys. In GFUR's materialization phase, we perform a gather on the unsorted table. Concretely, we insert a column with the original index of each key, merge/match the keys, and then perform a gather using the original indexes from the matches keys as the map and the original unsorted columns as the data.

3.2 Partitioned Hash Join

For the Partitioned-Hash-Join for GFUR, we referred the algorithm sketched by Wu et al. [7]. We begin by partitioning the keys of both tables into buckets, implementing two-pass radix partitioning along with bucket chaining to deal with partition overflow for GFUR. Concretely, each bucket is a physically fixed size, pre-allocated region which maps to a single partition. These buckets are connected through a chain for each partition. The buckets are allocated by offsetting pre-allocated memory. The last bucket in a partition may be incompletely filled leading to fragmentation. Further, there may be non-determinism in the partitions because we use atomics to construct a histogram in shared memory and the order of entries depends. We need to divide the relation into small enough block-sized clusters that fit in the shared memory of a SM.

The materialization process for GFUR with Partitioned-Hash-Join aligns with that in the Sort-Merge-Join. Similar to the GFUR in Sort-Merge-Join, we incur overhead from random accesses due to unclustered keys due to permutations in the non-deterministic partitioning.

Along with implementing the algorithm described in the paper for Partitioned-Hash-Join, we experimented with our own version of the partition kernel to avoid multiple kernel launches and bucket-chaining for GFUR. In order to do this, we wrote a kernel based on decoupled-lookback for scans[3] along with double buffering in shared memory to hide the load latencies. However, this entailed a compromise in doubling shared memory footprint and requiring persistent thread groups with grid synchronization. Eventually, we decided to follow the paper's implementation of bucket-chaining in view of performance.

4 EVALUATION

4.1 Experimental Setup

We use the microbenchmarks introduced by Wu et al. [7], testing our optimized joins by varying four different parameters:

(1) Input relation size (`nr`): The total number of rows in the input relations, $||R|| = ||S||$. We experiment with $nr = ns = (2^{22}, 2^{23}, 2^{24})$.
(2) Payload columns (`pr`): The number of payload columns per input relation. We set $pr = (1, 2, 3)$. More payload columns means a wider join.
(3) Input relation size ratio (`ratio`): The size of the second input relation relative to the first. We experiment with $ratio = (0, 1, 2)$, i.e. $(||R||, ||S||) = ((2^{23}, 2^{23}), (2^{23}, 2^{24}), (2^{23}, 2^{25}))$.
(4) Data skew (`skew`): The skewness of the foreign key distribution, controlled by a Zipf distribution parameter. We set $skew = (0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2)$.

We ensure that our implementations are correct before evaluating their performances by using the correctness check from the original implementation.

We make several adjustments to provide a fair comparison between our implementations and the original implementations. Specifically, we remove the early materialization optimization from the

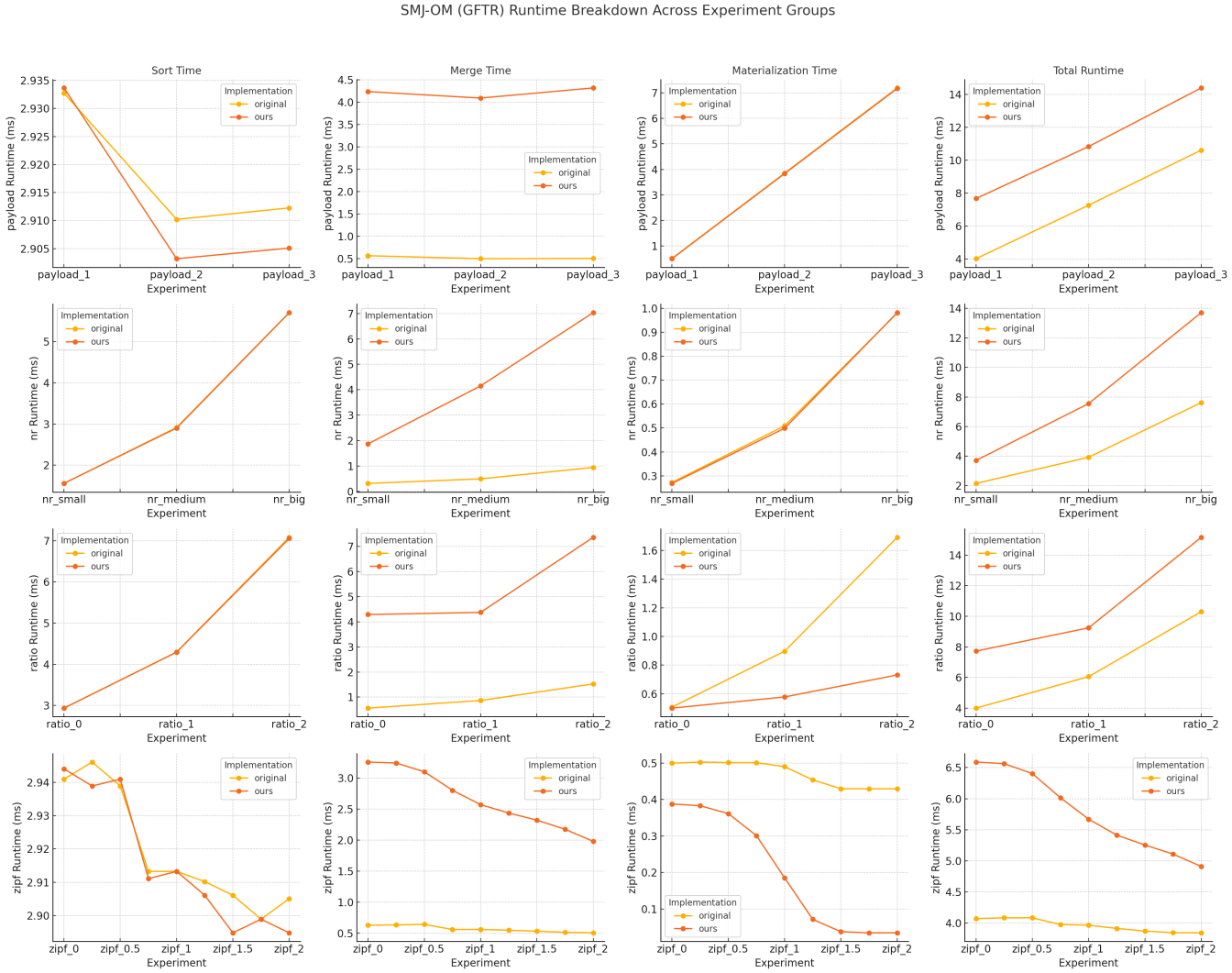


Figure 3: Results for SMJ-OM (GFTR) on microbenchmarks.

original SMJ implementation. This helps us compare our implementation's materialization more fairly with the original implementation's, or else there would be no materialization costs in the original implementation.

4.2 Sort Merge Join Performance

Figures 3 and 4 show the results of the microbenchmarks on our SMJ implementation with GFTR and GFUR. We consistently achieve within 3-4x of the original implementation's performance, though we oftentimes have comparable performance to their implementation. Some results may be inaccurate due to timing issues on Telerun infrastructure when results were collected, but most should be accurate.

When timing the 3 main phases of SMJ, we noticed our implementation and the paper's implementation have very similar sort and materialization runtimes, including in the cases where our implementation runs significantly slower than the paper's implementation. The merge step appears to be the bottleneck, being up 8 times slower compared to the paper's merge implementation on certain benchmarks. The merge path workload is memory bound and has low arithmetic intensity. Our current implementation of merge path does not fully utilize registers as in the paper's implementation which involves warp shuffles, and additional techniques like async mem copies and interleaving memory fetches and computation. This gives us a comparatively lower performance bottlenecked on memory bandwidth.

Figure 5 compares the sort, merge, and materialization times between SMJ-OM (GFTR) and SMJ-UM (GFUR) in our SMJ implementation. As claimed by the paper, the figure shows SMJ with GFTR tends to run faster than SMJ with GFUR, and we find the materialization phase in GFTR to be roughly 20-30% faster in general. The Wu et al. [7] paper claims this improvement in materialization time is because the gather tends to be sequential and batched in

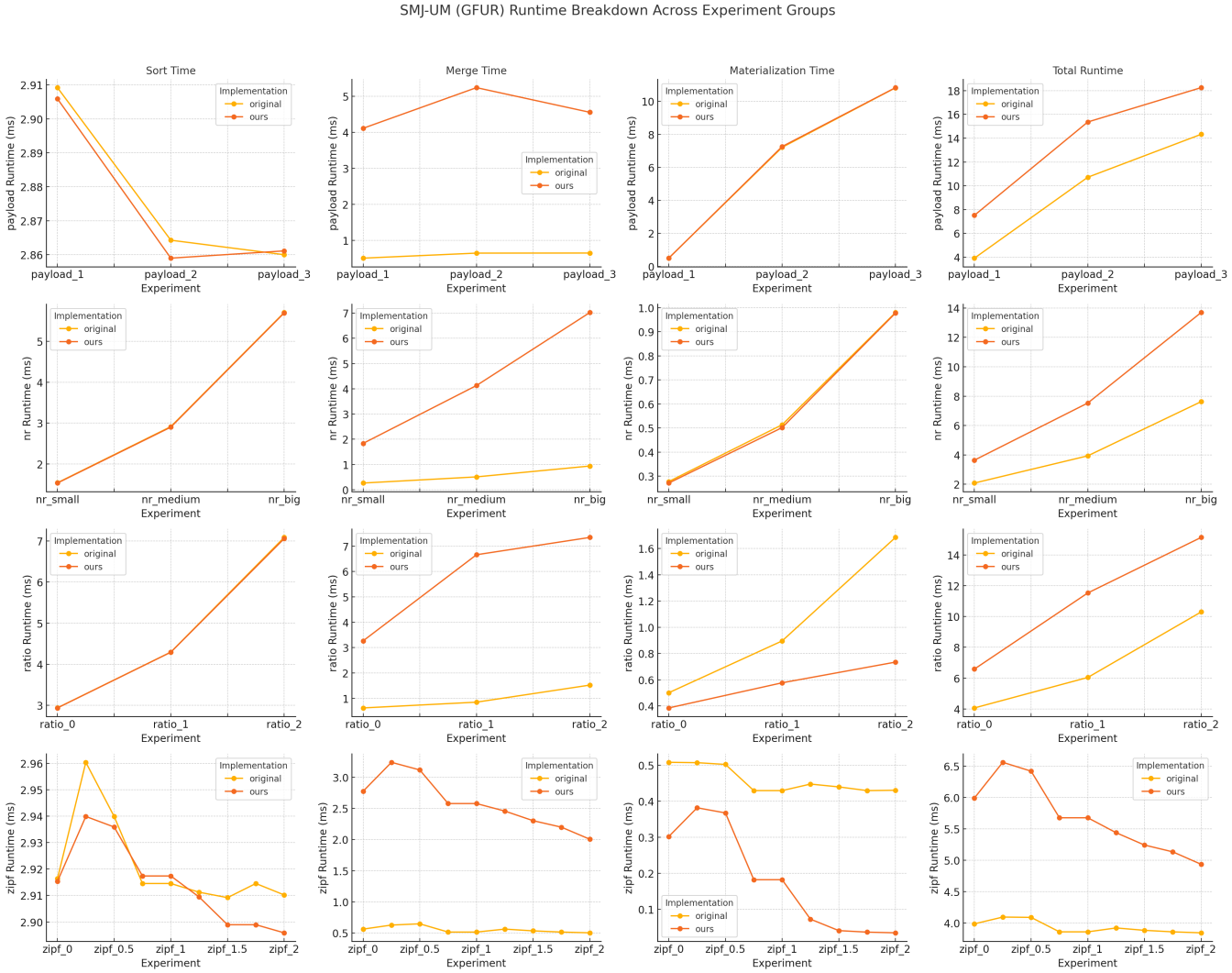


Figure 4: Results for SMJ-UM (GFUR) on microbenchmarks.

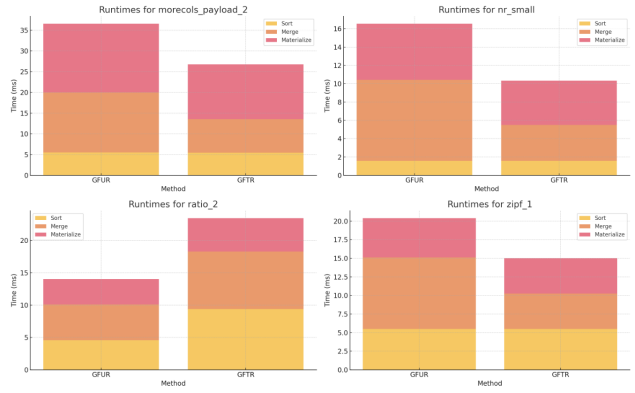


Figure 5: Comparison between GFTR and GFUR sort, merge, and materialization times in our SMJ implementation.

GFTR, whereas in GFUR the gather is randomly scattered. This indeed makes sense since the merged keys are in increasing order in GFTR, so the gather's map will have increasing indices for the sorted column data. Our experiment results provide further confirming evidence of the paper's hypothesis.

We do notice that SMJ with GFTR performs worse in the `ratio` benchmark shown in Figure 5. This is slightly surprising, as our code passed the correstness tests laid out by the original implemenation. We suspect that this may be due to Telerun's unreliability close to the final presentation time. Regardless, we still claim our experiment for SMJ with GFTR and GFUR shows GFTR performs better most of the time, and shows reduced materialization costs.

4.3 Partitioned Hash Join Performance

We benchmark the Partition and Hash-Join phases of the Partitioned-Hash-Join algorithm as the materialize phases are nearly identical

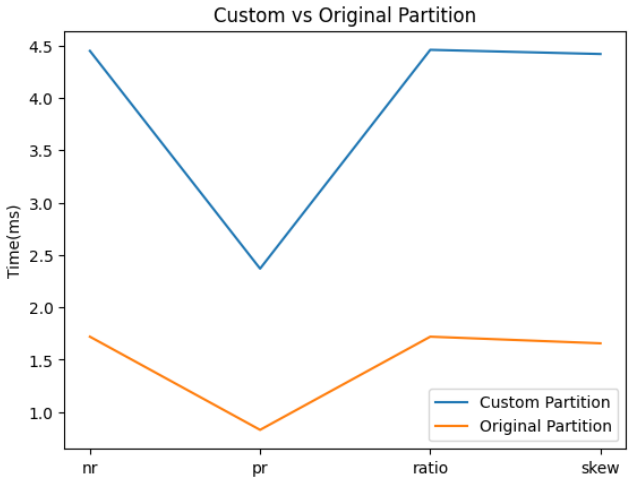


Figure 6: Partition benchmark for the Partitioned-Hash-Join algorithm. "Custom" refers to our implementation.

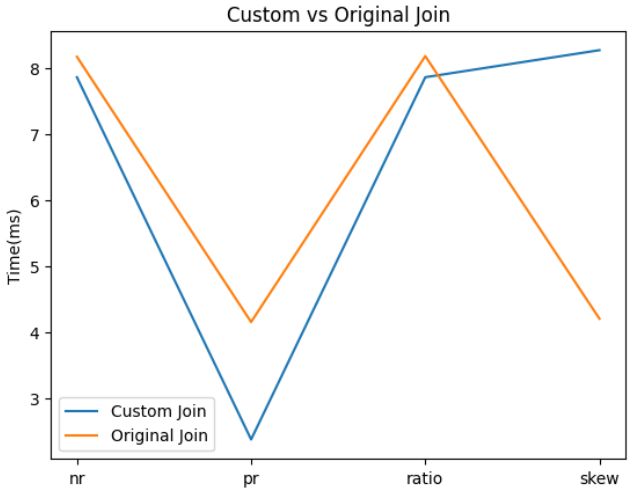


Figure 7: Join benchmark for the Partitioned-Hash-Join algorithm. "Custom" refers to our implementation.

to the Sort-Merge-Join algorithm, relying on the gather primitive provided by NVIDIA's cub library to collect rows from the two relations being joined. The results are shown in Fig. 6 and 7. As mentioned above, these results are subject to timing accuracy on the Telerun infrastructure. We beat the benchmark on join performance.

We matched the workload-balancing as in the original paper with splitting partitions into small block-nested joins over buckets that fit in shared memory. We hypothesize that the performance difference arises from different hashing strategies—open addressing (ours) versus chaining (original)—which leads to differing collision behavior.

5 CONCLUSIONS AND DISCUSSION

In this work, we explored GPU-accelerated join algorithms, focusing on Sort Merge Join (SMJ) and Partitioned Hash Join (PHJ) with both Gather-From-Untransformed Relations (GFUR) and Gather-From-Transformed Relations (GFTR) materialization strategies from Wu et al. [7]. We implemented and benchmarked SMJ-OM (SMJ with GFTR pattern), SMJ-UM (SMJ with GFUR pattern), PHJ-OM (PHJ with GFTR pattern), and PHJ-UM (PHJ with GFUR pattern). Our results validate the effectiveness of the GFTR technique in reducing the materialization bottleneck and improving throughput across various workloads.

Our results show we were able to replicate the 4 algorithms with performance comparable to the original implementation. Our SMJ implementation often has similar runtime to the paper's implementation, and in the worst case is 3-4x slower. Our PHJ implementation similarly performs roughly 2-3x slower than the original implementation. While there is still improvement to be done with our implementations, we implemented several complex optimizations and algorithms, tested for correctness, and benchmarked in a short-span. Our work shows the overall feasibility of using GPU's to accelerate inner-join operations, and also adds further evidence for Wu et al. [7] paper's hypothesis: GFTR reduces the materialization costs.

Overall, our work contributes valuable insights into the design and optimization of GPU-based join operations. Future work can extend our exploration to more complex query patterns, heterogeneous compute environments (CPU- and GPU-based), and real-world benchmarks such as TPC-H [6] and TPC-DS [5].

6 RELATED WORK

Early explorations into GPU-accelerated join operations demonstrated substantial performance gains over traditional CPU-based methods. He et al. [2] introduced hash joins on coupled CPU-GPU architectures, motivating the potential for fine-grained co-processing and cache reuse, which achieved up to 53% performance improvement over CPU-only processing. Other explorations have focused on evaluation on GPU-only joins. Rui et al. [4] evaluated the performance of various join algorithms on modern GPU hardware.

Moving towards recent join optimizations, Wu et al. [7] introduced the GFTR technique to address materialization costs in GPU-based joins, achieving throughput gains of up to 2.3 times over previous methods. The TPC-H and TPC-DS benchmarks [5][6] are widely used for assessing database performance; however, we were unable to use these benchmarks due to their large size.

There has also been related work in accelerating joins on GPUs for various other tasks. For example, Wu et al. [8] developed multi-predicate join algorithms for accelerating relational graph processing on GPUs, introducing significant speedup in computation on complex graphs.

7 CODE

Our code, containing implementations of the 4 algorithms, can be found at `github.com/kenctrl/relational-joins-cuda`. The `README` has instructions to run our code. We decided to initially copy over the repository provided by Wu et al. [7] as it includes setup to test for correctness and run various benchmarks on the

algorithms. We simply replaced their implementation of the algorithms with ours in key files and adjusted the test cases as needed. Our sort-merge-join implementation with GFTR and GFUR can be found in `sort_merge_join.cuh` and our merge-path code can be found in `merge_path.cuh`. The Partitioned-Hash-Join implementation is in the partitioner branch, and the file `PH.cuh` contains a full implementation of the Partitioned-Hash-Join algorithm. The repository contains instructions for running the code against various benchmarks.

ACKNOWLEDGMENTS

We would like to thank the 6.S894 staff—Prof. Jonathan Ragan-Kelley, William Brandon, and Nikita Lazarev—for their extended support throughout the final project period and for creating the Telerun infrastructure on which we developed and benchmarked our code.

REFERENCES

[1] Oded Green, Robert McColl, and David Bader. 2012. GPU Merge Path - A GPU Merging Algorithm. In *ICS '12: Proceedings of the 26th ACM international conference on Supercomputing*. AC<, 331–340.
[2] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *arXiv preprint arXiv:1307.1955* (2013).
[3] Duane Merrill and Michael Garland. 2016. Single-pass Parallel Prefix Scan with Decoupled Lookback. https://api.semanticscholar.org/CorpusID:51919482
[4] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join algorithms on GPUs: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2541–2550.
[5] TPC. 2024. TPC-DS Benchmark. http://www.tpc.org/tpcds/
[6] TPC. 2024. TPC-H Benchmark. http://www.tpc.org/tpch/
[7] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2023. Efficiently Processing Large Relational Joins on GPUs. *arXiv preprint arXiv:2312.00720* (2023).
[8] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar 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*, Vol. 10.