CPU and GPU Hash Joins on Skewed Data

Yuzhou Cai and Shimin Chen*
SKLP and Center for Advanced Computer Systems, ICT, CAS
University of Chinese Academy of Sciences
{caiyuzhou23s, chensm}@ict.ac.cn

***Abstract*—Hash join is one of the most important and widely used query processing operations. In many real-world applications, such as graph data analysis, join keys can be highly skewed. However, data skew is often only a secondary design consideration in existing CPU and GPU hash join algorithms. We find that the data structures and algorithm steps in existing hash joins are less efficient for handling highly skewed join keys, leading to significant performance degradation. In this paper, we optimize hash joins for skewed data, and propose a CPU Skew-conscious Hash join (CSH) and a GPU Skew-conscious Hash join (GSH). Our main idea is to detect skewed join keys and handle skewed vs. normal keys in separate routines optimized for their target cases. Our preliminary experiments show that compared to state-of-the-art CPU and GPU hash joins with existing skew-handling techniques, our proposed CSH and GSH achieve up to 8.0x and 13.5x improvement, respectively.**

***Index Terms*—Hash join, data skew, skew-conscious hash joins**

I. Introduction

Hash join is widely used and extensively studied in both disk-based and main memory database systems [1]–[34]. Recent research work optimized hash joins by taking advantage of different hardware features, including multi-core CPUs [13], [15]–[17], and GPUs [12], [14], [18], [23]–[26], [32]. In many real-world applications, such as graph data analysis, join keys can be highly skewed. The vertex degrees of real-world graphs often exhibit power-law distributions. A small number of vertices can have millions of neighbors, while most vertices are connected to only a handful of edges. Therefore, join operations on graphs often see highly skewed join keys.

Skew handling is considered in parallel hash joins in shared-nothing databases [4]–[7], [35], multi-threaded hash joins on multi-core CPUs [13], [15]–[17], and hash joins on GPUs [12], [18], [23], [24], [26], [32]. Data skew results in large load variance of join tasks in partition-based hash joins. Therefore, existing techniques are designed to balance the task load, e.g., by dividing large partitions into smaller ones, and performing dynamic task assignment. However, when data is heavily skewed, there can be a large number of tuples with the same join key. This renders load balancing techniques less effective as the data size of tuples with a single skewed key can be much larger than the average partition size.

To better understand the performance impact of data skew, we examine a state-of-the-art CPU-based hash join algorithm [16] (denoted as *Cbase*) and a state-of-the-art GPU hash join [24] (denoted as *Gbase*). We look into the source code to study their skew handling techniques, and investigate the join performance under various levels of data skew. We find that Cbase attempts to divide large partitions into smaller ones, and employs a task queue to dynamically balance the load across CPU join threads. To deal with a skewed partition, Gbase divides the R partition into sub partitions, and joins multiple R sub-partitions with the corresponding S partition with multiple thread blocks. Unfortunately, when both table R and table S are heavily skewed, these techniques are less effective. Moreover, we find that data skew is only a secondary design consideration in the algorithms. Skewed tuples and normal tuples are mixed together in join processing with the same data structures and code routines. The structures and routines are mainly optimized for low skew cases, and incur poor behaviors that impair join performance.

To optimize hash joins for skewed data, we propose a CPU Skew-conscious Hash join (*CSH*) and a GPU Skew-conscious Hash join (*GSH*). Our main idea is to detect skewed join keys and handle skewed vs. normal keys in separate routines optimized for their target cases. CSH detects skewed keys through sampling before the partition phase. It then designs a hybrid partition phase similar to the hybrid hash join [2], [3], to compute join results for skewed tuples during the partition phase. On the other hand, GSH detects skewed keys after the partition phase to avoid the complex code paths in CSH that would cause code divergence and degrade GPU performance. It designs a dedicated phase that better exploits the GPU parallelism to compute join results for skewed tuples. Our preliminary experiments show that compared to Cbase and Gbase, our proposed CSH and GSH achieve up to 8.0x and 13.5x improvement, respectively.

II. Background and Related Work

We briefly review the background on GPU programming with an emphasis on the difference between GPU and CPU performance considerations in Section II-A, then discuss related work on CPU and GPU hash joins in Section II-B.

A. GPU Programming

An NVIDIA GPU (e.g., A100) contains an array of (e.g., 108) Streaming Multiprocessors (SMs). Each SM comprises 2x32 CUDA cores. In accordance with this architecture, CUDA threads are organized hierarchically. Multiple threads form a thread block and multiple thread blocks constitute a grid. All threads within the same thread block execute on the same SM, and are scheduled in groups of 32 parallel

*Shimin Chen is the corresponding author.

threads (a.k.a. warps). Compared to the CPU, the GPU can run orders of magnitude more threads. However, the execution model is quite different. All threads in a warp perform Single Instruction Multiple Threads (SIMT) execution. Hence, code divergence, which can be caused by if-branches or variable numbers of loop iterations, incurs significant cost.

The GPU memory hierarchy consists of an (e.g., 192KB) L1 cache/shared memory per SM, a (e.g., 40MB) L2 cache shared across SMs, and a (e.g., 40GB) global memory. All threads in a thread block share the L1 cache/shared memory. Compared to the CPU main memory, the GPU global memory provides much higher bandwidth (e.g., 1555 GB/s), but higher access latency. Therefore, it is important to exploit the shared memory to place frequently accessed thread-block data structures, and optimize data access to the global memory (e.g., with memory coalescing) to take advantage of the high bandwidth.

B. Hash Join

One of the most efficient join algorithms, hash join is widely used and extensively studied in both disk-based and main memory database systems [1]–[34]. We focus on hash joins in main memory database systems in this work. Shatdal et al. [8] proposed a partition-based hash join in main memory that generates CPU cache sized partitions to reduce CPU cache misses. Ailamaki et al. [36] examined the query execution time of commercial DBMSs and found that CPU cache stalls are a significant problem for queries, including joins. Boncz, Manegold, and Kersten [9], [10] refined the partition-based hash join by taking TLB misses into account. To deal with high partition fanouts, which may incur frequent TLB misses and thus impair join performance, they proposed the radix join algorithm to perform two or more passes in the partition phase. Chen et al. [11] exploited CPU cache prefetching to reduce the performance impact of cache misses for hash joins. More recent research work optimized hash joins by taking advantage of different hardware features, including multi-core CPUs [13], [15]–[17], GPUs [12], [14], [18], [23]–[26], [32], Xeon Phi [21], FPGAs [20], [27], [28], and NVM [34], [37].

Skew handling is considered in parallel hash joins in shared-nothing databases [4]–[7], [35], multi-threaded hash joins on multi-core CPUs [13], [15]–[17], and hash joins on GPUs [12], [18], [23], [24], [26], [32]. The skew distribution of the join key column can result in skewed partition size and imbalanced task load across parallel workers/threads in the join phase. Existing techniques aim to balance the task load by dividing large partitions, performing dynamic task assignment, and/or assigning more GPU threads to larger partitions. However, when data is heavily skewed, there can be a large number of tuples with the same join key. The size of all tuples with a single join key can be much larger than the desired average partition size, making existing techniques less effective.

More specifically, we describe two state-of-the-art hash join algorithms (for which we are able to obtain the source code), and discuss their skew handling features in the following. We choose the two algorithms as the baseline CPU hash join and the baseline GPU hash join in this work, respectively.

Cbase: A Baseline CPU Hash Join. We choose a state-of-the-art CPU-based hash join algorithm [16] as the baseline CPU join. We call it *Cbase*. Cbase implements a parallel radix join. It consists of the partition phase and the join phase. In the partition phase, Cbase divides the input relation into equal-sized segments and assigns the segments to threads. Each thread scans its segment twice to avoid thread contention. The first scan counts the number of tuples in the target partitions. Based on the counts, Cbase computes the per-thread partition output offsets in an allocated contiguous memory space. Then, the second scan copies tuples to their target partitions without contention. To reduce TLB misses, Cbase performs the partition phase in two passes. Unlike the first pass, Cbase views each partition as a partition task and adds it into a task queue in the second pass. Then each thread repeatedly obtains and runs a partitioning task from the queue. In the join phase, every pair of R and S partitions is viewed as a join task and added into a task queue. Then each thread repeatedly obtains a join task from the task queue and carries out the join task until all tasks complete.

To deal with data skew, Cbase employs two techniques. First, if a partition is much larger than the average, Cbase breaks up the partition into smaller partitions. Second, the task queue mechanism is designed to tolerate certain load variance of join tasks. However, it is not feasible to break up tuples with the same join key since they always belong to the same partition. Therefore, in heavily skewed cases, the join task load can still be very skewed. A large task can dominate the join phase even with the task queue mechanism.

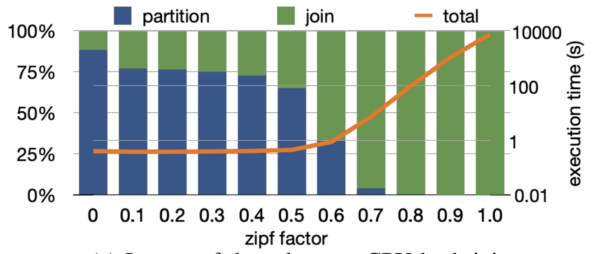
Gbase: a Baseline GPU Hash Join. On the GPU side, we choose a state-of-the-art GPU hash join [24] as our baseline. We call it *Gbase*. In this paper, we focus on joining GPU-resident data. Since the data transfer cost between the CPU and the GPU can be substantial, it is a promising solution to place a portion of the data in the GPU global memory and use GPU to process only the GPU-resident data in heterogeneous query execution [38]. Like Cbase, Gbase also consists of the partition phase and the join phase. In the partition phase, Gbase divides the input tables into shared-memory-sized partitions. All threads scan and copy tuples to the buckets of target partitions. If a bucket is full, Gbase allocates a new bucket and links the buckets of a partition in a linked list. To improve memory performance, Gbase reads a batch of (e.g., 4) tuples into registers and reorder them in the shared memory. Gbase writes reordered tuples to global memory with write coalescing. Since the shared memory size is small, Gbase uses two-pass partitioning. In the join phase, each thread block is used to join a pair of R and S partitions.

To deal with data skew, Gbase decomposes a long linked list of buckets in an R partition into multiple disjoint sub lists. Then, it assigns multiple thread blocks to the partition, each joining a sub list in the R partition with the full list in the S partition. In this way, Gbase reduces the amount of work in individual join tasks. However, this technique does not handle skewed S partitions. Moreover, a skew partition can produce much larger number of join result tuples. The current code is less optimized for such cases and incurs significant synchronization cost (cf. Section III).

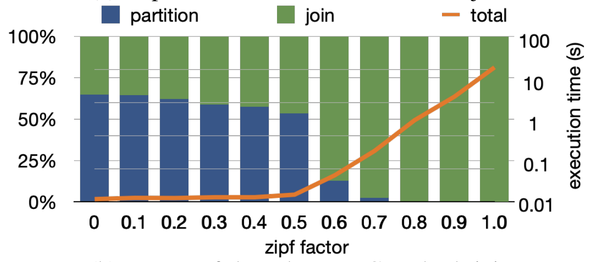
III. Performance Impact of Skewed Data

We would like to understand the join performance when the join key column is skewed. We join two equal-sized tables on both CPU and GPU hardware. Please see the detailed machine configuration in Section V. Each input table contains 32 million tuples. Every tuple is a pair of 4B join key and 4B payload. We randomly generate the join keys to follow the zipf distribution in both tables and vary the zipf parameter from 0 to 1. The join data fits into the CPU main memory and the GPU global memory. In the volcano-style query processing, the join output is often consumed by an upper level query operator. To model this behavior, we allocate a join output buffer per CPU thread or GPU thread block and overwrite the buffer repeatedly when it is full.

Figure 1 reports the execution times of Cbase and Gbase broken down into the partition phase and the join phase. From the figure, we see that given the same input size, as the join keys become more and more skewed, the execution times of both Cbase and Gbase increase drastically. Looking into the results, we find that the partition time stays relatively stable. Data skew has little effect on the partition phase. This is because the partition phase reorders and copies the input tuples to target partitions. The workload depends on the input table size, which keeps the same in the experiments. On the other hand, the execution time of the join phase rockets as the zipf factor increases. As a result, the join phase becomes an increasingly significant component in the execution time. It dominates the execution time at high skew cases where the zipf parameter is 0.8–1. We examine the join phase of Cbase and Gbase in more details in the following.



(a) Impact of data skew on CPU hash join.



(b) Impact of data skew on GPU hash join.

Fig. 1: Performance impact of skewed join keys.

Impact of Skew on the Join Phase of Cbase. First, the partition that contains skewed join keys becomes much larger than the average partition size. There are a large number of tuples with the same join keys. For example, when the zipf parameter is 1.0, the most popular join key is shared by about 1.79 million tuples in each input table. The skew-handling techniques in Cbase are less effective for such cases, leading to significant workload imbalance among join tasks.

Second, Cbase employs a chained hash table. It is not optimized to handle a large number of tuples with the same key. The chains become very long for popular keys, causing many dependent memory accesses for hash table probes. (Note that while the skewed R partition cannot fully fit into the CPU cache, the cache misses due to hash table probing is less damaging because the memory accesses for tuples of the skewed keys are roughly sequential.)

Finally, skewed partitions produce a large number of join result tuples for the skewed keys. For each hash table hit, the algorithm has to compare the R key and the S key to be sure that the two tuples indeed match. The number of key comparisons increases as the number of output tuples.

Impact of Skew on the Join Phase of Gbase. First, Gbase divides a long linked list of buckets in a skewed R partition into sub lists, then joins each sub list with the full list in the corresponding S partition. However, an S tuple is now probed multiple times for the multiple sub lists. Moreover, table S is also skewed in our experiments. The sub list technique does not handle the data skew in table S.

Second, chained hashing is also used in Gbase. Like Cbase, this incurs many dependent memory accesses for probing skewed keys. In addition, threads in a thread block probe the hash table for different S tuples. Since skewed keys and normal keys see drastically different chain lengths, there can be significant code divergence in the probe procedure.

Finally, Gbase uses a write bitmap to coordinate the writing of join output tuples in a thread block. Each thread probes the next tuple in the chain of its target hash bucket. Then it atomically sets a bit in the write bitmap to indicate whether a join output tuple is to be generated. The threads synchronize at this point. After the synchronization, each thread counts the number of bits to compute its output offsets and writes to the join output buffer. This write intention checking is done for every tuple in the chain of the hash table. Because skewed keys see long chains, the costly synchronization and atomic operations also dramatically increase.

Challenges for Handling Skewed Join Keys. A key observation is that there are a large number of tuples with the same join keys. This leads to two main challenges. First, key-based partitioning cannot evenly divide data. The generated partitions can be very skewed. The join tasks are imbalanced, leading to idle join threads and waste of resources. Second, existing data structures and algorithms are optimized mainly for normal keys and exhibit poor behaviors with skewed keys. Long chains in the chained hash table restrict the instruction-level parallelism in CPU. The probe and write procedure in Gbase sees code divergence and a lot of synchronization overhead. Hence, we are motivated to treat skewed keys as a first-class citizen to better utilize the hardware resources.

IV. Skew Conscious Hash Joins

The key problem of existing solutions is that skewed join keys are mixed with normal join keys. They are processed using the same data structures and procedures. Hence, our approach is to identify skewed keys and then process skewed keys and normal keys separately. This approach benefits both normal keys and skewed keys. First, after skewed keys are filtered out, partitions with normal tuples can fit into the CPU cache or the GPU shared memory as expected. There is lower variance across the load for join tasks. Hence, the join task load can be easily balanced among CPU threads. Moreover, normal tuples often see similar small number of matches. Therefore, there is lower code divergence among GPU threads. Second, skewed keys can also benefit from this approach because we can design more efficient join and output procedures for skewed keys to fully exploit the hardware.

In Section IV-A and Section IV-B, we propose a CPU skew conscious hash join, *CSH*, and a GPU skew conscious hash join, *GSH*, respectively.

A. CSH: CPU Skew Conscious Hash Join

We propose *CSH*, a CPU Skew conscious Hash join. As shown in Figure 2, CSH is based on the parallel partitioned hash join. We introduce a skew-detection phase before the partition phase, and process skewed tuples explicitly:

(1) Detect skewed keys through sampling: CSH samples (e.g., 1‰) keys from table R and uses a hash table to compute the frequencies of the sampled keys. If the frequency of a key exceeds the pre-defined threshold (e.g., 2), the key is marked as a skewed key. Each skewed key is allocated a skewed partition for the R tuples with the given key.

(2) Partition table R: When partitioning table R, CSH places normal tuples and skewed tuples into separate partitions. For each R tuple, it checks the tuple in the skew checkup table. If the join key is a skewed key, then the tuple is appended to the associated skewed partition as indicated by the part_id in the skew checkup table. Otherwise, the R tuple is appended to a normal partition.

(3) Partition table S: When partitioning table S, CSH also checks each S tuple in the skew checkup table. For a normal tuple, it appends the tuple to a normal partition. On the other hand, for a skewed tuple, CSH directly visits the associated skewed partition and produces join result tuples for all R tuples in the skewed partition.

(4) Join normal partitions: Since skewed tuples have been already processed in the partition phase. The remaining partitions contain only normal tuples. Therefore, CSH can efficiently join each pair of normal partitions. We call this phase NM-join in the figure.

Our implementation parallelizes all the phases with multiple CPU threads in the similar fashion as Cbase.

CSH can efficiently handle skewed tuples for the following reasons. First, detecting skewed keys before the partition phase allows CSH to process skewed tuples explicitly. We design a hybrid partition phase similar to that in the hybrid hash join [2], [3]. Skewed S tuples are not written to partitions. CSH produces join result tuples for skewed S tuples on the fly. In this way, skewed S tuples are only read once. Hence, CSH avoids the cost of copying skewed S tuples to partitions. Second, skew detection is based on a small sample of the R tuples. Thus, the introduced detection overhead is quite low. Third, CSH efficiently generates join results for skewed tuples. For a skewed S tuple, CSH performs efficient sequential memory accesses to retrieve all R tuples in the associated skewed partition. This procedure also avoids the cost of verifying if the R and S keys match before generating every join result tuple, which would be necessary after hash table probes in the join phase. Finally, the join phase processes only normal partitions, making it easy to achieve load balance for multiple CPU threads.

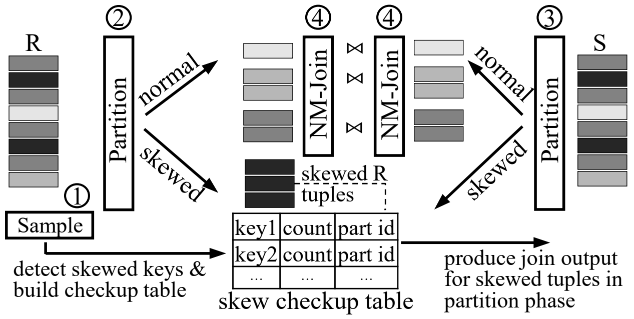


Fig. 2: CPU skew conscious hash join (CSH).

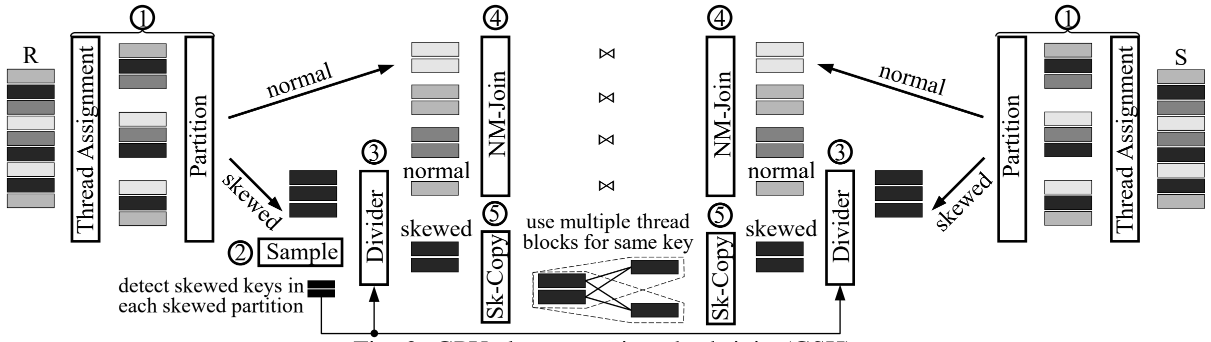
B. GSH: GPU Skew Conscious Hash Join

We propose *GSH*, a GPU Skew conscious Hash join. Figure 3 illustrates the GSH algorithm. Like CSH, the high level idea of GSH is to detect skewed keys and process skewed tuples separately. However, unlike CSH, GSH detects skewed keys after the partition phase rather than before the partition phase. This is because the partition procedure of CSH handles normal tuples and skew tuples in different code paths. This can incur severe code divergence in GPU, leading to poor performance. On the other hand, the high bandwidth of the GPU global memory can alleviate the cost of copying S tuples in the partition phase. Therefore, we choose to perform skew detection after the partition phase in GSH.

GSH consists of the following phases, as shown in Figure 3:

(1) Partition table R and S: GSH partitions input tables into shared memory sized partitions. Given the high global memory bandwidth, we implement a simple count then partition procedure, which avoids the complexity of dynamic buffer allocation of Gbase. It requires two scans for each partition pass. Since the shared memory size is limited, like Gbase, GSH uses two passes to partition the input tables.

(2) Detect skewed keys in large partitions: After the partition phase, the size of each partition is known. GSH can easily identify normal vs. large partitions by comparing the partition size with a pre-defined threshold. A large partition may contain both skewed tuples and normal tuples. For each large partition, GSH samples (e.g., 1%) tuples to detect skewed keys. GSH uses a linear probing based hash



R
S
Thread Assignment
Partition
normal
skewed
Sample
detect skewed keys in each skewed partition
Divider
normal
skewed
NM-Join
Sk-Copy
use multiple thread blocks for same key

Fig. 3: GPU skew conscious hash join (GSH).

table to compute the frequencies of sampled keys. After sampling, GSH marks the top-k most frequent keys in each large partition as the skewed key. k should be chosen to remove most skewed keys so that the normal partition containing the remaining tuples can fit into the shared memory. In our experiments, we find that k=3 is sufficient to achieve this purpose.

(3) Divide large partitions into normal and skewed partitions: For each large partition, GSH checks each R tuple against the skewed keys. If the tuple is skewed, it is appended to an array allocated for the associated skewed key. If the tuple is normal, GSH appends the tuple to a normal partition. For the corresponding S partition, GSH checks the S tuples and appends them to either skewed tuple arrays or the normal partition in a similar fashion.

(4) Join normal partitions: A thread block is used to join a pair of normal partitions. For the normal partition, we implement a normal join procedure (NM-Join) similar to Gbase. We allocate a chained hash table in the shared memory. Each thread scans tuples in the R partition to insert R tuples into the hash table. Then, each thread scans tuples in the S partition to probe the hash table for matches.

(5) Produce join results for skewed partitions: After all NM-joins, GSH processes the skewed tuples. To fully exploit the GPU capability, GSH computes join result tuples for a skewed key using multiple thread blocks. Each thread block focuses on one R tuple from the skewed R tuple array. The threads in the thread blocks read the skewed S tuples and writes the join result tuples in parallel. In this way, the thread block performs coalesced memory accesses to read skewed S tuples and write join results.

The GSH design has the following benefits. First, NM-join handles only normal shared-memory sized partitions. It can efficiently exploit the GPU threads to process normal tuples. Second, like CSH, skew detection is based on a small sample of tuples. Thus, the introduced detection overhead is quite low. Third, compared to Gbase, GSH performs an additional copy operation for large partitions. It divides a large partition into skewed tuple arrays and a normal partition. Due to the high global memory bandwidth, this copying overhead is modest. Finally, GSH generates the join result tuples for skewed tuples efficiently. It parallelizes the computation of join results for a single skewed key with multiple thread blocks. Compared to Gbase, this procedure better exploits the GPU's parallelism.

V. Performance Evaluation

We evaluate the performance of our proposed skew conscious hash joins in this section. Section V-A describes the experimental setup, then Section V-B reports the preliminary experimental results.

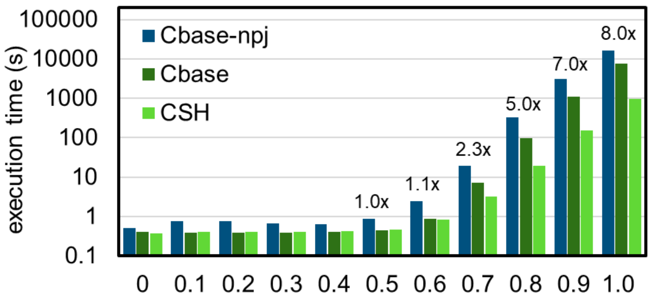
A. Experimental Setup

Machine Configuration. Both CPU and GPU programs run on the same machine. It is equipped with two Intel Xeon E5-2640 v4 CPUs (2.40GHz, 10 cores/20 threads per CPU, 25 MB L3 cache) and 128GB DDR4-2133 memory. There is a NVIDIA A100-PCIE-40GB GPU (108 SMs, 6912 CUDA cores, 192KB L1 cache/shared memory per SM, 40MB shared L2 cache, and 40GB global memory) with CUDA 12.1. The machine runs 64-bit Ubuntu 18.04 LTS Linux. The programs are compiled with GCC 7.5.0 and NVCC 12.1.

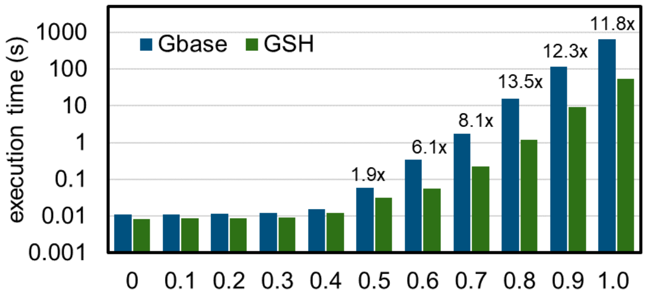
Solutions to Compare. For CPU hash joins, we compare our proposed CSH with Cbase [16]. As described in Section II-B, Cbase is a radix join. Besides Cbase, we also compare with a no-partition join in the same code repository. We denote it *cbase-npj*. CSH is a partitioned hash join based on Cbase radix join. We add the skew detection phase before the partition phase and then enhance the partition phase to process skewed tuples separately, as described in Section IV-A. We run all CPU hash joins with 20 threads.

For GPU hash joins, we compare our proposed GSH with Gbase [24]. GSH performs skew detection after the partition phase, then divides large partitions into skewed and normal parts. Skewed tuples are processed in a separate phase that exploits multiple GPU thread blocks to generate join results for each skewed key, as described in Section IV-B. Our GSH implementation considers the top-3 most frequent keys in the sample of a large partition as skewed keys.

Workload. Both input tables contain 32 million tuples. Every tuple is a pair of 4B join key and 4B payload. We randomly generate the join keys to follow the zipf distribution in both tables and vary the zipf parameter from 0 to 1. Specifically, we generate an array of intervals for a given zipf factor. Each array element stores an interval whose length corresponds to the probability of the element in the zipf distribution. Then we randomly assign a unique key to each interval. After that, for



(a) CPU hash joins. (The speedup of CSH over Cbase is reported.)



(b) GPU hash joins. (The speedup of GSH over Gbase is reported.)

Fig. 4: Hash join performance varying the zipf factor.

each input tuple, we generate a random number, and search it in the interval array. If the number falls into an interval, the join key of the tuple is set to the corresponding unique key. We repeatedly generate the input tuples until the table reaches the expected size. In our experiments, we model highly skewed cases by using the same interval array and unique key array for both table R and table S for a given zipf factor.

B. Preliminary Experimental Result

Figure 4 shows the hash join performance varying the zipf factor from 0 to 1. Table I reports the time breakdown of the hash joins for the zipf factor from 0.5 to 1.

CPU Hash Joins. As shown in Figure 4a, CSH is comparable to Cbase at low to medium skew where the zipf factor is 0–0.4. Cbase-npj is the worst performing solution. As the data is more and more skewed, CSH sees higher improvement over Cbase. Compared to Cbase, CSH achieves up to 8.0x improvement for medium to high skew cases where the zipf factor is 0.5–1.0.

In Table I, we consider the time components that include the computation of join results for skewed tuples, i.e. Cbase join and CSH sample+partition. Cbase mixes skewed tuples and normal tuples in its join phase. Its skew handling techniques work poorly for highly skewed cases because a skewed key can be shared by a large number of tuples. In comparison, CSH detects the skewed keys before the partition phase and explicitly handles the skewed tuples in the partition phase using a technique similar to the hybrid hash join. This significantly reduces the overhead for processing skewed tuples. When the zipf factor is 1.0, CSH detects 870 skewed keys and generates 5.26x10^{12} join result tuples from skewed tuples, which contribute to about 99.6% of the total output. Interestingly, the NM-join time in CSH increases modestly as the data is more skewed. This is because the skew detection collects mainly the highly skewed keys. Keys that are modestly skewed are treated as normal keys and processed in the NM-join phase. This leads to the increase of the workload of the NM-join phase as the zipf factor increases.

TABLE I: Execution time breakdown.

zipf factor	0.5	0.6	0.7	0.8	0.9	1.0
Cbase partition	0.29s	0.29s	0.29s	0.29s	0.28s	0.26s
Cbase join	0.16s	0.59s	7.05s	96.9s	1084s	7593s
CSH sample+part	0.22s	0.36s	2.24s	17.6s	152s	941s
CSH NM-join	0.25s	0.47s	0.9s	1.65s	2.36s	2.55s
Gbase partition	6.78ms	6.6ms	6.8ms	6.9s	7.0ms	7.4ms
Gbase join	52ms	0.33s	1.7s	16s	115s	643s
GSH partition	5.9ms	5.9ms	6.1ms	7.7ms	12.8ms	24.5ms
GSH all other	25.8ms	49.3ms	0.214s	1.17s	9.37s	54.5s

GPU Hash Joins. As shown in Figure 4b, as the data becomes more skewed, like the CPU hash joins, GSH also sees significant improvement over Gbase. Compared to Gbase, GSH achieves up to 13.5x improvement for medium to high skew cases where the zipf factor is 0.5–1.0. The improvement of GPU hash joins is more substantial than that of CPU hash joins. This may result from the higher level of parallelism available in the GPU.

When the zipf factor is 0–0.4, none of the partitions is larger than the shared memory, and therefore our skew handling steps are not used. In these cases, GSH is comparable to Gbase.

In Table I, we compare Gbase join and GSH all other as both process skewed tuples to generate join results. We see that as the zipf factor increases, the time of the Gbase join phase rockets. Gbase's sub list technique is less effective for handling highly skewed data. In contrast, GSH spends significantly lower amount of time to process skewed tuples. This shows the effectiveness of our skew handling approach that detects skewed keys and explicitly handle skewed tuples.

Experiments with Larger Input Tables. We scale up the two input tables to have 560 million tuples with the zipf factor = 0.7. In this case, Gbase uses about 38.5GB of the 40GB global memory. Our experimental results show that CSH achieves 3.5x speedup over Cbase, and GSH achieves 10.4x improvement over Gbase.

VI. Conclusion

In this paper, we focus on skew handling for CPU and GPU hash joins in main memory databases. We find that existing skew handling techniques are less effective for high skew cases, and propose a CSH and a GSH algorithms for CPU and GPU, respectively. Our preliminary experimental study shows that CSH and GSH can achieve significant performance improvement for skewed data.

Acknowledgment

This work is partially supported by National Key R&D Program of China (2023YFB4503600) and Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

References

[1] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, “Application of hash to data base machine and its architecture,” *New Generation Computing*, vol. 1, pp. 63–74, 1983.

[2] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood, “Implementation techniques for main memory database systems,” in *SIGMOD’84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pp. 1–8, 1984.

[3] D. J. DeWitt and R. H. Gerber, “Multiprocessor hash-based join algorithms,” in *VLDB’85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pp. 151–164, 1985.

[4] M. Kitsuregawa and Y. Ogawa, “Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC),” in *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pp. 210–221, 1990.

[5] K. A. Hua and C. Lee, “Handling data skew in multiprocessor database computers using partition tuning,” in *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pp. 525–535, 1991.

[6] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, “An effective algorithm for parallelizing hash joins in the presence of data skew,” in *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pp. 200–209, 1991.

[7] A. Shatdal and J. F. Naughton, “Using shared virtual memory for parallel join processing,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pp. 119–128, 1993.

[8] A. Shatdal, C. Kant, and J. F. Naughton, “Cache conscious algorithms for relational query processing,” in *VLDB’94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pp. 510–521, 1994.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten, “Database architecture optimized for the new bottleneck: Memory access,” in *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pp. 54–65, 1999.

[10] S. Manegold, P. Boncz, and M. Kersten, “Optimizing main-memory join on modern hardware,” *IEEE transactions on knowledge and data engineering*, vol. 14, no. 4, pp. 709–730, 2002.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, “Improving hash join performance through prefetching,” in *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pp. 116–127, 2004.

[12] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, “Relational joins on graphics processors,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 511–524, 2008.

[13] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, “Sort vs. hash revisited: Fast join implementation on modern multi-core cpus,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.

[14] P. Bakkum and K. Skadron, “Accelerating sql database operations on a gpu with cuda,” in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pp. 94–103, 2010.

[15] S. Blanas, Y. Li, and J. M. Patel, “Design and evaluation of main memory hash join algorithms for multi-core cpus,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 37–48, 2011.

[16] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, “Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware,” in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 362–373, IEEE, 2013.

[17] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, “Multi-core, main-memory joins: Sort vs. hash revisited,” *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.

[18] J. He, M. Lu, and B. He, “Revisiting co-processing for hash joins on the coupled CPU-GPU architecture,” *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 889–900, 2013.

[19] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe, “Memory-efficient hash joins,” *Proc. VLDB Endow.*, vol. 8, no. 4, pp. 353–364, 2014.

[20] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, “Fpga-based multithreading for in-memory hash joins,” in *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[21] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, “Improving main memory hash joins on intel xeon phi processors: An experimental approach,” *Proc. VLDB Endow.*, vol. 8, no. 6, pp. 642–653, 2015.

[22] O. Polychroniou, A. Raghavan, and K. A. Ross, “Rethinking SIMD vectorization for in-memory databases,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 1493–1508, 2015.

[23] R. Rui and Y.-C. Tu, “Fast equi-join algorithms on gpus: Design and implementation,” in *Proceedings of the 29th international conference on scientific and statistical database management*, pp. 1–12, 2017.

[24] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, “Hardware-conscious hash-joins on gpus,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 698–709, IEEE, 2019.

[25] R. Rui, H. Li, and Y.-C. Tu, “Efficient join algorithms for large database tables in a multi-gpu environment,” *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 708–720, 2020.

[26] J. Paul, B. He, S. Lu, and C. T. Lau, “Revisiting hash join on graphics processors: A decade later,” *Distributed and Parallel Databases*, vol. 38, pp. 771–793, 2020.

[27] M.-T. Xue, Q.-J. Xing, C. Feng, F. Yu, and Z.-G. Ma, “Fpga-accelerated hash join operation for relational databases,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 10, pp. 1919–1923, 2020.

[28] X. Chen, Y. Chen, R. Bajaj, J. He, B. He, W.-F. Wong, and D. Chen, “Is fpga useful for hash joins?,” in *CIDR*, 2020.

[29] E. Marinelli and R. Appuswamy, “Xjoin: Portable, parallel hash join across diverse xpu architectures with oneapi,” DAMON ’21, 2021.

[30] M. Bandle, J. Giceva, and T. Neumann, “To partition, or not to partition, that is the join question in a real system,” SIGMOD ’21, p. 168–180, 2021.

[31] S. Jahangiri, M. J. Carey, and J.-C. Freytag, “Design trade-offs for a robust dynamic hybrid hash join,” *Proc. VLDB Endow.*, vol. 15, no. 10, p. 2257–2269, 2022.

[32] W. Sun, A. Katsifodimos, and R. Hai, “An empirical performance comparison between matrix multiplication join and hash join on gpus,” in *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pp. 184–190, 2023.

[33] J. Yang, H. Li, Y. Si, H. Zhang, K. Zhao, K. Wei, W. Song, Y. Liu, and J. Cui, “One size cannot fit all: a self-adaptive dispatcher for skewed hash join in shared-nothing rdbmss,” *arXiv preprint arXiv:2303.07787*, 2023.

[34] W. Huang, Y. Ji, X. Zhou, B. He, and K.-L. Tan, “A design space exploration and evaluation for main-memory hash joins in storage class memory,” *Proc. VLDB Endow.*, vol. 16, no. 6, p. 1249–1263, 2023.

[35] D. A. Schneider and D. J. DeWitt, “A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment,” in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, pp. 110–121, 1989.

[36] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “Dbmss on a modern processor: Where does time go?,” in *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pp. 266–277, 1999.

[37] S. Chen, P. B. Gibbons, and S. Nath, “Rethinking database algorithms for phase change memory,” in *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pp. 21–31, 2011.

[38] B. W. Yogatama, W. Gong, and X. Yu, “Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS,” *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2491–2503, 2022.