Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment

Ran Rui
University of South Florida
Tampa, Florida, USA
ranrui@usf.edu

Hao Li
University of South Florida
Tampa, Florida, USA
haoli1@usf.edu

Yi-Cheng Tu
University of South Florida
Tampa, Florida, USA
tuy@usf.edu

ABSTRACT

Relational join processing is one of the core functionalities in database management systems. It has been demonstrated that GPUs as a general-purpose parallel computing platform is very promising in processing relational joins. However, join algorithms often need to handle very large input data, which is an issue that was not sufficiently addressed in existing work. Besides, as more and more desktop and workstation platforms support multi-GPU environment, the combined computing capability of multiple GPUs can easily achieve that of a computing cluster. It is worth exploring how join processing would benefit from the adaptation of multiple GPUs. We identify the low rate and complex patterns of data transfer among the CPU and GPUs as the main challenges in designing efficient algorithms for large table joins. To overcome such challenges, we propose three distinctive designs of multi-GPU join algorithms, namely, the nested loop, global sort-merge and hybrid joins for large table joins with different join conditions. Extensive experiments running on multiple databases and two different hardware configurations demonstrate high scalability of our algorithms over data size and significant performance boost brought by the use of multiple GPUs. Furthermore, our algorithms achieve much better performance as compared to existing join algorithms, with a speedup up to 25X and 2.8X over best known code developed for multi-core CPUs and GPUs respectively.

PVLDB Reference Format:
Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. PVLDB, 14(4): 708-720, 2021.
doi:10.14778/3436905.3436927

1 INTRODUCTION

Relational join is one of the core components in database management systems (DBMS). It is an essential operator for many database applications that involve multiple tables. Hence improving join processing performance has been an active topic in database research. The increasing database size in today's business applications has imposed significant challenges to efficient processing of relational joins.

Modern hardware technologies, especially multi-core chips, provide abundant computing capabilities that could benefit database management. As a result, there is a large body of work on join algorithms designed and optimized for such hardware. On multicore CPUs, various strategies [1–3, 5, 12] such as workload partition and assignment, cache optimization, and use of SIMD instructions are proposed.

Many-core computing platforms, represented by Graphics Processing Units (GPUs), also attracted much attention from the research community. It has been demonstrated [8, 9, 11, 19, 21, 22] that GPUs, as a general-purpose parallel computing platform, are very promising in processing relational joins – a speedup of 5-10X has been reported when comparing GPU join code with multi-thread CPU code. All such work, however, is based on the assumption that *input tables and intermediate join results can be fully loaded to the GPU memory*. In a typical GPU, the size of the on-board memory is at the 8-16GB level, this sets a tight limit on the scale of database that can be processed. In this paper, we propose efficient algorithms for processing joins when both input tables are too big to be GPU resident.

Nowadays, more and more workstation and servers carry multiple GPU devices. The combined computing capability of (four to eight) GPUs in such a machine can easily dwarf those of a cluster that consists of dozens of nodes seen a few years ago. In general, the bottleneck in processing relational join is at data transmission rather than in-core computation. While the high bandwidth of GPU memory builds the foundation for efficient GPU join algorithms, multiple GPUs, by providing higher aggregated memory size, can further improve join processing. In this paper, we present design and implementation of GPU algorithms that take advantage of the combined capabilities of multiple GPUs in a single node (although our algorithms also work under single-GPU scenarios). To the best of our knowledge, this is the first work that relaxes the "small table" assumption by using multiple GPUs in join processing.

There are unique challenges in using multiple GPUs for large table join processing. First, the bandwidth of PCI-E is a major bottleneck even though it has steadily increased over the years. Since the memory size of GPUs is limited, it is inevitable that data exchange occurs between the CPU (host) and the GPUs (devices) via PCI-E bus. The fact that multiple GPUs share the host-to-device channels further increases data exchange overhead. Therefore, it is vital to minimize the data transfer between them. Second, in a multi-GPU environment, data sharing among devices is allowed and this provides extra opportunities for better performance. On other hand, inter-GPU communication has to consider the special structure of the PCI-E links, making algorithm design and optimization a non-trivial task. As a result, the goal of our algorithm designs is to overcome the aforementioned challenges and take full advantage of multi-GPU platform for join processing.

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436927

This paper makes the following contributions. First, by exploring the hardware hierarchy and design space of multi-GPU join algorithms, we identify the main obstacles against efficient large table joins on multiple GPUs. Specifically, we conclude that limited bandwidth and complex structure of the data communication links are the main issues. Therefore, we focus on minimizing the data transfer among GPUs and CPUs. Second, we propose three distinctive designs of multi-GPU join algorithms: nested loop, global sort-merge, and hybrid joins. These algorithms can handle extremely large input tables, and each has its applicability to joins with different types of conditions. The nested loop join addresses both the inter-GPU communication and low data transfer rate by taking advantage of PCIE P2P data transfer, while the other two algorithms tackle the low data transfer rate by asymptotically reducing the overall data traffic over PCIE. Last, we evaluate the performance of our algorithms against various data tables with sizes up to 12 billion tuples using two servers featuring PCI-E and NVLink interconnections, respectively. The best of our algorithms show linear scalability on table size, and also achieved much better performance with the use of multiple GPUs (e.g., up to 2.8X speedup for four GPUs compared with a single GPU). Our algorithms significantly outperform multi-thread CPU join code, with a speedup up to 25X.

The paper is organized as follows. In Section 2 we discuss challenges of multi-GPU environments in join processing. In Section 3 we present three join algorithms designed for such hardware environments in detail. In Section 4 we present experimental results and discuss the impact of different factors on join performance. Section 5 briefly summarizes previous work; and Section 6 concludes this paper.

2 CHALLENGES IN MULTI-GPU JOINS

Over the years, the GPUs has experienced the imbalanced growth of GPU resources – increase of interconnection bandwidth has lagged behind that of in-core computing capabilities of GPUs. The slower interconnection has put more restrictions on improving join performance on GPUs. In this section, we discuss the two major challenges of using multiple GPUs for join processing: limited PCI-E bandwidth and complex inter-GPU communication. We will propose our algorithm designs to tackle these challenges in later sections.

2.1 Limited PCI-E Bandwidth

The use of multiple GPUs in processing large table joins, while further complicating the issue, still sees inter-device communication as the major cost. The limited capacity of a main-stream GPU's global memory (up to 16GB) only allows it to process a portion of the database tables at a time. Therefore, data exchange among all the computing devices (including CPUs and GPUs) is inevitable and more frequent than the single-GPU with small data scenario. The actual bandwidth of the PCI-E link (at $32GB/s$ bidirectional) is at most in par with the host memory bandwidth, further restricting fast data transfer. Hence, the overhead of data shipment is more significant when the amount of data becomes larger. As a result, a key aspect of designing GPU join algorithms for larger dataset is to minimize the overhead brought by such data transmission.

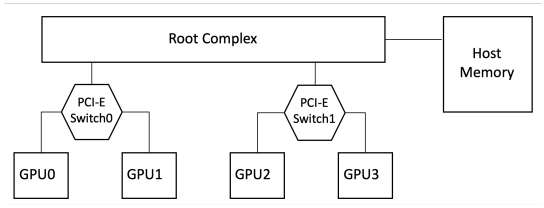

Figure 1: A typical system layout of four GPUs connecting to one CPU socket

Each PCIE connection consists of multiple physical lanes that provide up to 32GB/s bidirectional bandwidth in total. When multiple GPUs need to copy data from/to the host simultaneously, they must share the bandwidth of the only host-to-GPU connection. When the number of GPUs increases from 1 to N, the combined computing power increases by a factor of N, but the total host-to-GPU bandwidth remains the same. This also means the host-to-GPU data transfer rate for each GPU is reduced by N. Hence, it is important to eliminate back-and-forth data transmission between the host and the GPUs in designing join algorithms.

2.2 Complex Inter-GPU Communication Pattern

In addition to its limited bandwidth, the structure of the PCI-E interconnection introduces extra complexity. If there is only one GPU in the system, the connection is end-to-end and bi-directional between the host memory and the GPU. When multiple GPUs are connected to the system, it forms a tree structure that consists of GPUs and PCI-E switches, as shown in Figure 1. Each direct link between two points is bi-directional and has full PCI-E 16X bandwidth. Again, any concurrent traffic passing through the same PCI-E link have to share the bandwidth. Fortunately, the PCI-E links are two-way duplex, meaning it provides bi-directional data exchange capabilities. The number of GPUs supported by one socket can be up to 8, in which case there are two levels of switches and a root complex. Data transmission between GPUs follows the shortest path between the devices, i.e., in Figure 1, GPU0 and GPU1 communicate via Switch0, but GPU1 and GPU2 have to follow a path of *Switch0-root-Switch1*. GPU-to-host communications have to go through the root complex.

For systems with multiple CPU sockets, each socket could connect to a network shown above. As a result, the system has more PCI-E lanes for device-device data transfer. However, data movement between sockets have to go through the inter-socket connection (i.e., QPI for Intel and X Bus for IBM), which has much lower bandwidth than in intra-socket transfer.

Moreover, the data have to hop to different regions of the host memory before reaching the target GPU, giving rise to large data transfer latency. Such regions are generally called *non-uniform memory access* (NUMA) regions. Hence it is critical to minimize the data exchange between NUMA regions and GPUs that are attached to different CPU sockets.

Note that recent Nvidia cards are equipped with high-speed NVLink interconnection with 80GB/s of bi-directional bandwidth. From our experiments, we observe that the NVLink shares similar characteristics with the PCI-E except the former provides higher

Table 1: Common Symbol Definitions.

Symbol	Definition
D	Number of GPUs
R, S	Relations to be joined
M	Number of chunks in relation R
N	Number of chunks in relation S
nRP	Number of Radix Partitions

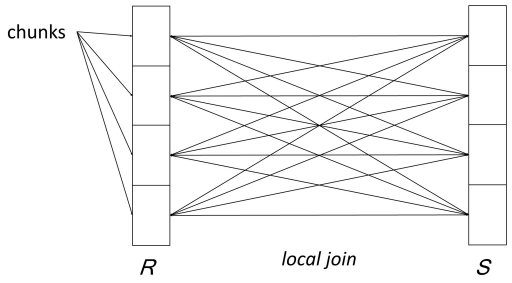

Figure 2: Overview of block-based nested-loop join

bandwidth. With NVLink, the data transfer is still a bottleneck in processing joins between large tables. Details of such experiments can be found in our technical report [20].

3 LARGE TABLE JOIN ALGORITHMS

In this section, we describe three variants of multi-GPU join algorithms in detail. Since the focus of this work is to explore the design of large joins with multiple GPUs by reducing the inter-device data transfer overhead, we utilize single-GPU join algorithms proposed in [21] as the basic building block of our work. We chose such primitives because they deliver the best performance (as we have verified via extensive experiments) in single-GPU-small-table joins.

Even though the three algorithms have different running time complexity, all of them will find their uses in an actual GPU-based DBMS. The nested-loop join can handle joins with arbitrarily complex conditions. The GSM join is suitable for joins with equality or range match join conditions. The hybrid join can process joins with equality and integer type range conditions. One other advantage of the GSM join is that the resulting table is ordered in a particular way. This could lower the costs of downstream operators such as "order by" or "group by" in a database query, and is a frequently utilized mechanism in query optimization.

In the following discussions, we assume that both input tables are too big to reside in GPU memory. However, we assume the host main memory is large enough to host both tables. Our algorithm design and implementation are based on NVidia's CUDA programming framework. Symbols and notations used throughout this paper are listed in Table 1.

3.1 Block-Based Nested-Loop Join

We start by the intuitive idea of nested-loop join with blocking strategy as the first multi-GPU join algorithm. By blocking, we mean that the join is carried out between blocks (or chunks) of the input tables. This algorithm is convenient for implementation of theta join or for use with data that are already partitioned. As shown in Figure 2, the two input tables R and S are split into equally-sized chunks so that a pair of chunks (one from each of R and S) along with intermediate data can fit into the global memory of a GPU. Upon loading a chunk from the outer table R to each GPU, all chunks of the inner table S will be loaded one by one into the same GPU. Depending on the specific join conditions, we can use different in-core join algorithms such as nested loop, sort-merge, or hash joins shown in [21] and [9]. Our algorithm terminates when all chunks of R are consumed. Algorithm 1 shows the pseudocode of the nested-loop join. In all algorithms presented in this paper, by "omp parallel" we mean to unroll the loop and distribute the workload (via the loop index) to different GPUs via OpenMP.

Algorithm 1: Block-Based Nested-Loop Join with in-core Sort-Merge Join

```
Input: R, S, D, M, N
Output: Join result res
 1: r[] = partition(R, M);
 2: s[] = partition(S, N);
 3: for i = 0 to M-1 omp parallel do
 4:    setCudaDevice(i%D);
 5:    r' = r[i];
 6:    for j = i; j < N; j+=D do
 7:       s' = s[j]
 8:       res.append(gpuInCoreJoin(r', s')); //local join
 9:       for k = 1 to D-1 do
10:          s' = memCopy(s',(i+1)%D); //copy local s[i] to next
             device
11:          res.append(gpuInCoreJoin(r',s'));
12:       end for
13:    end for
14: end for
```

While the above algorithm seems straightforward, the design of the inner loop (line 11 in Algorithm 1) calls for optimizations that are unique in a multi-GPU environment. The easiest way would be to directly fetch chunks of S from the host memory. However, this requires bandwidth sharing in the PCI-E root complex among all GPU cards (Figure 1), leading to a major bottleneck. Our strategy is to have the GPUs exchange their chunks of S (Figure 3) before asking for new chunks from the host memory. Recall (Figure 1) that peer-to-peer data transmission could be done in lower levels of the PCI-E network tree, thus reducing the load on the high-level switches including the root complex. Given that, the inner loop of Algorithm 1 is divided into stages, in each stage the D GPUs will each load a chunk of S, the D chunks of S will be consumed by all GPUs before the next stage starts.

An interesting problem here is: *given the D chunks, how do we arrange the P2P transmission among D GPUs such that time for every GPU to see all chunks is minimized?* With the PCI-E network structure represented in Figure 1, our solution is as follows: in a PCI-E network of D GPUs, the i-th GPU ships its data to its immediate neighbor, e.g., the $(i+1)\%D$-th GPU (Figure 3). By this, it takes $D-1$ rounds to share all the data, and we will take full advantage of the data channels in the entire PCI-E network. We can prove that with the PCI-E network structure represented in Fig. 1, the Ring

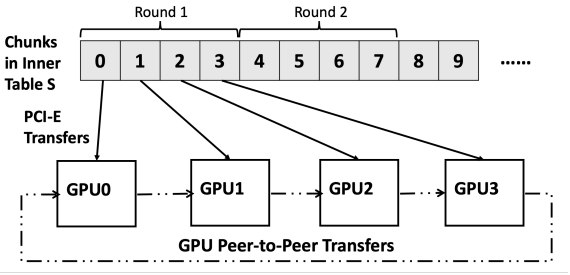

Figure 3: Loading of inner table chunks and GPU peer-to-peer data transmission in the Nested-Loop Join

exchange plan requires the smallest amount of time to exchange all data for all GPUs, though the proof is omitted here due to the page limits.

Performance Analysis: Suppose there are a total of M chunks in table R and N chunks in table S, the number of devices is D and the PCI-E uni-directional bandwidth is B. Intuitively, without P2P data transmission, the total amount of data transferred between host and devices is

$$\mathcal{T}_1 = |R| + M|S| \tag{1}$$

where $|\cdot|$ is the size of a table. By taking advantage of the PCI-E P2P transfer, we reduce the amount of data transferred between the host and the devices to $|R| + \frac{M}{D}|S|$. In addition to that, we have to add the data traffic via P2P transfer. For each step of the inner loop, there are D chunks of S that need to be exchanged, and each of the chunk has to be transferred $D-1$ time. As a result, there are $D(D-1)$ P2P exchanges to be done in total. Thus, the total amount of P2P data transfer is $\frac{M}{D} \cdot \frac{|S|}{D} D(D-1) = \frac{M|S|}{D}(D-1)$. Given those, the amount of data transfer of the whole algorithm (with P2P) is

$$\mathcal{T}_2 = |R| + \frac{M}{D}|S| + \frac{M|S|}{D}(D-1) \tag{2}$$

However, to compare the time needed to accomplish the transfer of $\mathcal{T}_1$ and $\mathcal{T}_2$, we need to apply a D-fold discount to the 3rd item of Eq. (2), and the bandwidth-adjusted $\mathcal{T}_2$ is

$$\mathcal{T}_2' = |R| + \frac{M}{D}|S| + \frac{M|S|}{D^2}(D-1) = |R| + M|S|\left(\frac{2D-1}{D^2}\right) \tag{3}$$

By comparing Eq. (3) with Eq. (1), we can clearly see a performance advantage brought by the P2P data transfer. By ignoring the low-order item $|R|$ in both equations, the improvement is by a factor of $D^2/(2D-1)$.

3.2 Global Sort-Merge (GSM) Join

While the nested-loop join can handle arbitrary join conditions, it carries a quadratic data transmission cost. For joins with more special conditions such as equality or range matches, we introduce our second join algorithm named global sort-merge (GSM) join. It is based on the well-known idea of sort-merge join which brings the data transmission cost down to subquadratic. The key challenge of it would be to design a sorting algorithm that works with multiple GPUs and is able to consume input larger than the total GPU memory size. First, we have to let several GPUs work independently and

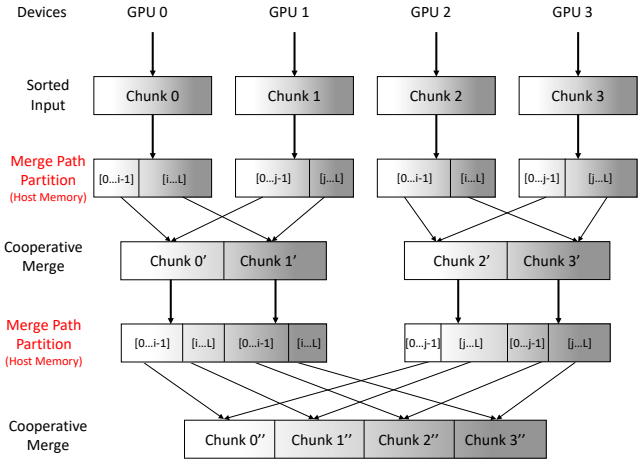

Figure 4: Global sorting with two GPUs

cooperatively to sort and merge the data out-of-core while reducing the data transfer impact. Second, we should keep all their workload balanced throughout the process. Both tasks are not trivial.

In traditional disk-based DBMS, sorting depends on serial merge of two sorted lists, each of which has to be accomplished by one worker (CPU) thread. For GPUs, we can do the same, but it is less efficient. In the in-memory setup, the random-access host memory allows for CPUs to search through the input and partition the workload for multiple GPUs. Therefore we developed a multi-GPU sorting algorithm that adopts CPU-based merge path partition[6, 17] to enables multi-GPU merge.

The algorithm consists of two steps: sort and join. In sort step, the GPUs cooperatively sort the input tables. In join step, they work independently to join small chunks of the sorted tables. In order to realize the out-of-core processing and multi-GPU parallelism, the sort step is further split in to two parts: sorted-run generation and multi-GPU merge. The join step applies the same idea of multi-GPU merge to conduct multiple in-core joins. With the CPU-assisted merge path partitioning, each GPU works on a pair of disjoint partitions of the two tables and multiple GPU devices are capable of working independently and cooperatively. We elaborate each step in the following sections.

3.2.1 Sorted Run Generation. Since we assume the input tables can be arbitrarily large and the GPUs' global memory capacity is limited, divide-and-conquer is necessary. Therefore the sorting must be split into two parts: generating small sorted runs and merging the sorted runs into one sorted table. The sorted run generation ensures that the sorted runs are small enough to fit in a GPU's global memory along with its intermediate data structures, and each sorted run can be processed locally by a GPU. As shown in Algorithm 2 and Figure 4, we first split the input relations into equal-sized chunks whose sizes fit in the global memory of a single GPU. Each of the GPUs takes one chunk at a time from the host memory with a fixed stride. Then it uses efficient in-core GPU radix sort algorithm [16] to sort the chunk locally, and transfer it back to the host memory. All GPUs work together in parallel until all the chunks are sorted.

3.2.2 Multi-GPU Merge of Sorted Runs. When the sorted runs are ready, they need to be merged into larger sorted lists. However, it is impossible for a GPU to hold two small sorted runs in its global memory. Multiple GPUs must work cooperatively when merging the sorted runs and perform several rounds of merging if the sublists to be merged are larger than the total size of all GPUs' global memory. Therefore a data partition is necessary to ensure parallelism and load balancing among the GPUs. To tackle this challenge, we propose a multi-threaded CPU-assisted multi-GPU merge, the core mechanism that enables out-of-core processing with multiple GPUs, which is shown in Algorithm 3. It features merge-path partition which is a parallel partition algorithm that utilizes binary search on two sorted lists to partition them into load-balanced workload for multi-threaded merging. It breaks the merge workload of two sorted lists A and B into multiple equal-sized portions by using binary search along the pairs $A[i]$ and $B[j]$, where $i + j = L$ and L is the partition size.

In each round of merge, multiple threads are launched using OpenMP in accordance with the number of GPUs D so that there is one CPU thread working with a GPU. Before the merge begins, the threads apply merge path partition on the two sublists to be merged. Each GPU works on a pair of partitions independently. Since the merge path partition is a binary search into both sorted lists to ensure the merges are load-balanced, the GPUs all have the same amount of work even though the partitions may not be of the same size. The GPUs fetch their partitions from the host memory and write the merged list back to the host memory afterwards. If the size of the pair of sublists to be merged are too large, they will be split into more partitions so that multiple GPUs can work cooperatively on them. If there is only one GPU available, it can process the partitions as well.

Figure 4 illustrates the workflow with an example of two GPUs and four input chunks. After GPU 0 and 1 take turns to sort the chunks, the data are all transferred back to the host. Each of chunks 0-3 is partitioned by CPU into two partitions using merge path in host memory(labeled by red font). GPU0 first works on merging elements 0 to i-1 of chunk 0 and elements 0 to j-1 of chunk 1, and then merging the corresponding partitions of chunks 2 and 3. Meanwhile, GPU1 first works on merging elements i to L of chunk 0 and elements j to L of chunk 1, and then merging the other partitions of chunks 2 and 3. As a result, chunk 0' and 1' are generated and form a single sorted list, while chunk 2' and 3' form another one. Then the two newly generated sorted lists go through another round of partitioning and merging by CPUs and GPUs respectively, until the four chunks are merged into a single sorted list.

3.2.3 Multi-GPU Merge Join. As both input tables are sorted and ready in the host memory, we can proceed to the merge join stage (Algorithm 4). Similar to the idea of multi-GPU merge, we launch D CPU threads by OpenMP and use merge path partition to split both tables into D partitioins. Each GPU acquires a pair of partitions from R and S and joins its own pair independently. It is possible that the size of each pair of partitions still exceeds a GPU's global memory capacity. Therefore for each GPU we apply the merge path partition again that splits each pair of partitions in to even smaller ones so that one GPU can join partitions within the limit of global memory size. Each GPU joins the smaller partitions in a sequential manner until it finishes all pairs of partitions.

3.2.4 Overlapping Computation with Data Transfer. A key challenge in GPU computing is to hide the overhead of data transfer via the shared PCI-E channels. The overhead is especially significant in join processing because relational joins often involve very little in-core computation (e.g., simple data comparison instead of heavy arithmetic operations). Normally, the running time of a GPU task consists of the following three parts: time T_1 for shipping input data from main memory to GPU global memory, time T_2 for GPU computation, and time T_3 for shipping output data back to CPU. The total time for running a task is therefore $T_1 + T_2 + T_3$. GPU programming frameworks often provide asynchronous APIs to create a pipeline that is capable of reducing the overhead by overlapping data transfer with in-core computation. Specifically, we implement such a pipeline using CUDA Stream for the global sort-merge join, and the third join algorithm in Section 3.3. We use three CUDA streams, each performing the complete host-to-device transfer, sorting and device-to-host transfer process as a pipeline. Therefore the three pipelines can overlap each other so that the compute resources and PCI-E bandwidth are kept busy most of the time. To facilitate the execution of the pipelines, the available global memory space on the GPU is divided into three parts. So each stream has its own work space, there is no contention among the streams when accessing or transferring data. Theoretically, use of CUDA streams can lead to a total running time of $\max(T_1, T_2, T_3)$. As shown in previous work [21], T_2 is generally a non-dominant part of total running time.

3.2.5 Performance Analysis. In the sorted run generation, we still assume there are M sorted runs in R and N in S for simplicity. The total data transferred between the host and GPUs including copying sorted runs back to main memory is $2(|R| + |S|)$. The multi-GPU merge stage requires $log_2 M$ and $log_2 N$ rounds to merge all the sorted runs in R and S respectively. Therefore the total data transferred between host and GPUs is $2(|R|log_2 M + |S|log_2 N)$. During the above two steps, we take advantage of CUDA streams and bi-directional data transfer, so the transfer bandwidth is $2B$. In the final merge join, the data traffic is $|R| + |S|$. In summary, the total amount of data transmitted between host and GPUs is

$$3(|R| + |S|) + 2(|R|log_2 M + |S|log_2 N) \tag{4}$$

As a result, the time spent on data transfer over PCI-E is

$$\frac{2(|R| + |S|)}{B} + \frac{|R|log_2 M + |S|log_2 N}{B} \tag{5}$$

As compared to the Nested-Loop Join, the advantage of the GSM Join lies in its low I/O cost. It was achieved by the global sorting such that only those chunks of R and S that could generate join results will be processed by the GPUs for the final in-core merge.

3.3 Hybrid Join

Although the GSM join is handy when dealing with equi-join and range join, it comes with a price of transferring data back and forth to accomplish the sorting task. To further reduce the data transfer, we need a partitioning or partial sorting mechanism that is more efficient in I/O. The radix paritioning widely used in hash

Algorithm 2: Global Sort-Merge Join

Input: *R, S, D, M, N*
Output: Join result *res*

```
1:  r[] = partition(R, M);
2:  s[] = partition(S, N);
3:  for i = 0 to M-1 omp parallel do
4:      setCudaDevice(i%D);
5:      gpuSort(r[i]);
6:  end for
7:  cooperativeMerge(r, M, D);
8:  for i = 0 to N-1 omp parallel do
9:      setCudaDevice(i%D);
10:     gpuSort(s[i]);
11: end for
12: cooperativeMerge(s, N, D);
13: res = globalJoin(r, s, D);
```

Algorithm 3: cooperativeMerge

Input: Partitioned Relation *r, M, D*
Output: Sorted Relation *r*

```
1:  stride = 2;
2:  numPairs = M/stride;
3:  for i = 0 to numPairs-1 do
4:      for k = 0 to stride-1 omp parallel do
5:          setCudaDevice(i%D);
6:          cpuMergePath(r[2*k], r[2*k+1]);
7:          gpuMerge(r[2*k], r[2*k+1]);
8:      end for
9:  end for
```

Algorithm 4: globalJoin

Input: Sorted Relation *R* and *S*, *D*, Max Join Size *T*
Output: Join Results *res*

```
1:  [numPartitions, r[], s[]] = mergePathPartition(R, S, T);
2:  for i = 0 to numPartitions-1 omp parallel do
3:      setCudaDevice(i%D);
4:      res += inCoreSMJ(r[i], s[i]);
5:  end for
```

join algorithms [3, 9, 12] serve this purpose. On the other hand, our previous work on GPU-based hash and sort merge joins [21] revealed the fact that *the in-core sort-merge join is more efficient than the hash join based on radix partitioning on GPUs*. That leads to the idea of a *hybrid* design that enjoys the merits of both worlds - a combination of radix partitioning and in-core sort-merge join(see details in Algorithm 5). In this algorithm, we first group the tuples with the same radix value in each of the input tables into small partitions, then each GPU takes a pair of partitions from *R* and *S* and proceed to in-core sort-merge joins independently. Both steps are scalable with more GPUs. We want to point out that, due to the global radix partitioning and in-core sort merge, this algorithm works for any type of equi-joins as well as joins with range conditions when the join key of integer type.

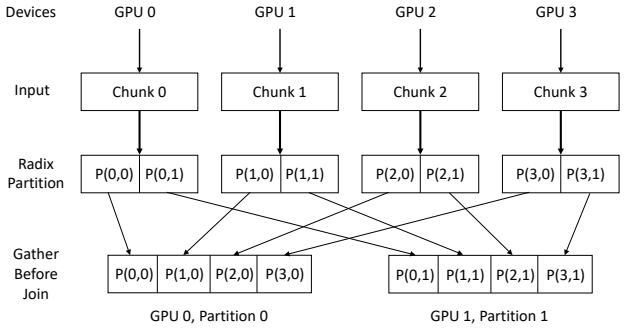

Figure 5: Radix partition with two GPUs

Algorithm 5: Hybrid Join

Input: *R, S, M, N, nRP, D*
Output: Join result *res*

```
1:  r[] = partition(R, M);
2:  s[] = partition(S, N);
3:  presumR[M][nRP] = gpuRadixPartition(r[], M, D);
4:  presumS[N][nRP] = gpuRadixPartition(s[], N, D);
5:  for i = 0 to nRP omp parallel do
6:      setCudaDevice(i%D);
7:      r' = gather(r[], presumR[][], M, i);
8:      s' = gather(s[], presumS[][], N, i);
9:      gpuSort(r');
10:     gpuSort(s');
11:     res.append(inCoreSMJ(r', s'));
12: end for
```

3.3.1 Multi-GPU Radix Partition. The first step of hybrid join is to partition the input tables. Unlike other partitioning strategies, radix partitioning creates disjoint partitions that are suitable for parallel sort-merge join tasks since one partition in one table has only one corresponding partition in the other table, hence reducing computation and memory access cost. In addition, extracting the radix value only requires bit operations which are more efficient than the comparison-based mechanism. The radix partition consists of three steps: building histogram, computing prefix-sum and scattering tuples to new positions. Building histogram is to count the number of tuples falling in each bucket that represents a particular radix value. A prefix scan on the histogram results in prefix-sum, an array of numbers that can be used to determine the starting position to scatter the tuples in each bucket. Based on the prefix-sum, we can reorder the tuples in parallel. Previous work addressed the radix partition on multi-core CPUs and on single GPUs [9, 12, 21]. In this section, we focus on our design with multiple GPUs.

To make the GPUs work simultaneously, we need to distribute the input among them and then combine their local histograms. Algorithm 6 describes the workflow in detail. To build the histogram, each input table is scanned once. Therefore each GPU simply fetch a chunk of the input that fits in the global memory at a time. Then the GPUs proceed to in-core histogram kernel for each chunk, and keep an array of the partial histograms of every thread block in the GPU's global memory. Every time a GPU processes a new chunk, it counts the partial histograms for that particular chunk accordingly.

The GPUs keep fetching chunks and computing histograms until the input is exhausted. Upon the completion of counting histogram, each chunk have its own partial histogram on the data.

There are two options for the following prefix-sum and scatter. The intuitive way is to combine the partial histograms from all the chunks, consequently generating a global prefix sum and the tuples are scattered to different contiguous spaces (or buckets). On the contrary, the other option allows for lower overhead by letting the GPUs build local prefix-sum within each chunk. In other words, the radix partition is done at chunk level rather than the whole table.

Specifically, right after the histogram is built for a particular chunk, the corresponding GPU begins to calculate its local prefix-sum and then scatters the tuples to the buckets in GPU's memory space. Therefore, it is not necessary to keep the histogram of that particular chunk afterwards, reducing the memory consumption and extra data transfer overhead. It also helps reduce scatter overhead since it can be done within the global memory, eliminating the need for accessing host memory. The GPUs keep working chunk by chunk until all of them are processed. The only minor issue with this method is that the tuples belonging to the same bucket spread in several different chunks although they are clustered locally within each chunk. In the in-core join stage, the separated sub-partitions of each partition have to be fetched separately. Since we have the prefix-sums for all the chunks that help us decide the starting and ending positions of the sub-partitions, the problem should be easily solved by copying them one by one.

Algorithm 6: gpuRadixPartition

Input: Partitions of Relation *r[], M, nRP, D*
Output: prefix of the radix partitions of all the chunks *globalHisto*

```
1: for i = 0 to M-1 omp parallel do
2:    setCudaDevice(i%D);
3:    histogram[nRP] = gpuComputeHistogram(r[i], nRP);
4:    presum[nRP] = gpuComputePrefix(histogram);
5:    gpuReorderRelation(r[i],presum);
6:    globalHisto.append(presum[]);
7: end for
```

The workflow of the multi-GPU radix partition is shown in Figure 5, where an example of four GPUs and four chunks is used. In the example, a partition P(i,j) represents a sub-partition of partition j processed by GPU i. For GPU 0, it takes chunk 0 of table R at first from the main memory and perform radix partition on it(we use two partitions P(0,0) and P(0,1) for illustration purpose), then puts the reordered chunk back to main memory. The two GPUs partition the chunks in stride manner untill all the chunks are partitioned. The GPUs do the same to the chunks of table S, we omit this step for simplicity. The GPUs gathers the data of the same partition from different chunks before the in-core join starts.

3.3.2 In-Core Sort-Merge Join. After the radix partition, the paired partitions with the same radix value from the two tables can be joined on the GPUs independently. As mentioned above, the tuples of a particular partition are scattered into many sub-partitions across multiple chunks. Hence at the beginning of the in-core join, the sub-partitions need to be gathered as a whole partition, as shown in Figure 5. To do so, we iterate over all the chunks and fetch the corresponding sub-partitions with the particular radix value one at a time with the help of the partial prefix-sum generated in the previous stage. The sub-partitions are copied into a buffer in the GPUs' global memory. GPU 0 is assigned to gather P(0,0) through P(3,0) since they all belong to partition 0. As soon as the input partitions are ready in the global memory, the GPUs perform in-core sort-merge join in parallel. The join results are buffered in another array in the global memory. When the in-core join procedure finishes, the results are transferred to the main memory and combined into one list.

3.3.3 Performance Analysis. In the hybrid join approach, data transfers occur at radix partitioning stage and only involve a linear scan of the input from the main memory and writing back to the main memory. The in-core join stage only reads the reordered input once. Therefore, the total data traffic over PCI-E is

$$3(|R| + |S|). \tag{6}$$

Due to the fact that the scan and write in the radix partitioning stage can be overlapped using bidirectional bandwidth of the PCI-E channels, the data transfer time of that stage is reduced by half.[1] Thus, the total data transfer time is

$$\frac{2(|R| + |S|)}{B} \tag{7}$$

The above shows that the hybrid join is clearly superior to the other algorithms introduced earlier.

4 EXPERIMENTS

To evaluate the performance and scalability of three join algorithms, we test them on two different hardware platforms. The first one consists of two Intel Xeon E5-2630V3 CPUs with 384GB of RAM, where each Xeon processor socket connects four Nivida GTX Titan X Pascal GPUs via PCI-E3.0 16X. The second one is an IBM Minsky server with two Power8 processors, 512GB of RAM, and each processor socket connects two Nvidia Tesla P100 GPUs via NVLink. Both platforms have CUDA 9.0 installed on Linux operating systems. Each GTX Titan X Pascal has 12GB GDDR5X RAM, while the Tesla P100 has 16GB HBM2 RAM.

We follow the convention found in many existing work [8, 9, 11] on relational joins in generating synthetic data: the tuples are $< key, value >$ pairs where both the key and value are 32-bit integers. The keys are randomly generated following an uniform distribution, and for the tests against skewed data, a Zipf distribution. Unless specified otherwise, we set the table size $|R|$ and $|S|$ to be the same.

The parameters M and N (number of chunks) of all algorithms are chosen in a way such that the largest chunks of R and S are used, as allowed by the specific algorithms, and we keep the same chunk sizes for both R and S. In particular, the chunk size for the nested-loop join is 250M records, 50M records for GSM and hybrid joins.

For benchmarking purposes, we compare our code with the state-of-the-art parallel CPU hash join code [2] with multi-threading,

[1] The actual data transfer rate may be different. For example, our tests show that we can achieve 12GB/s uni-directional and 22GB/s bi-directional. Thus, the data transfer time will reduce by a factor of 1-12/22=0.45, which is slightly lower than 0.5.

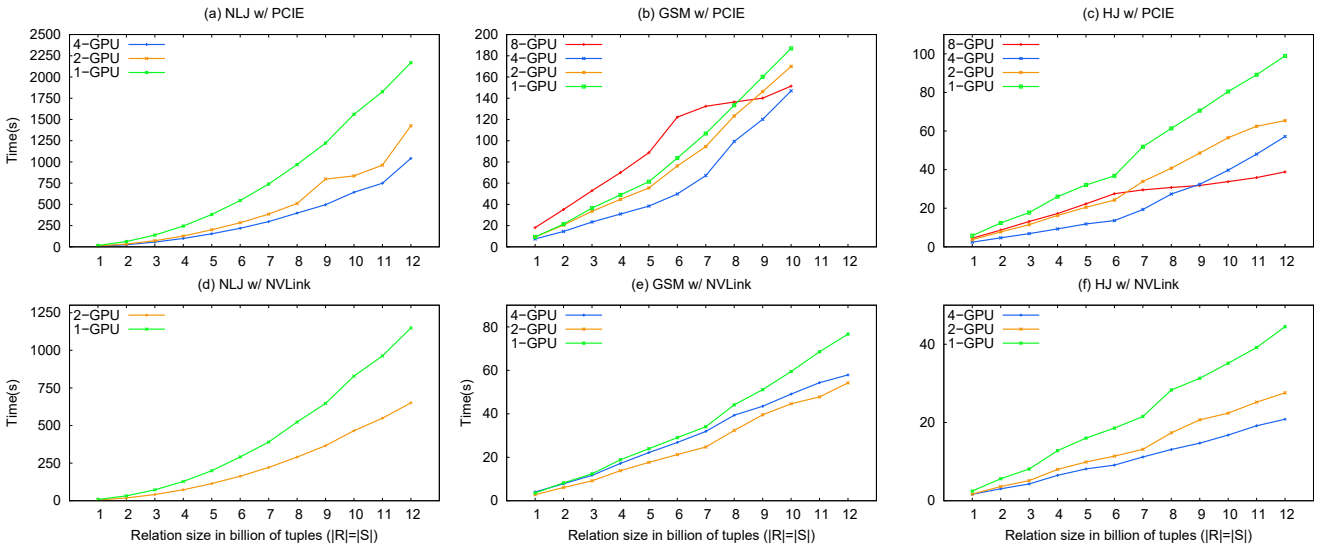

Figure 6: Running time of three join algorithms with PCIE and NVLink

SIMD and cache-conscious optimization techniques, and the GPU code in [22] which addressed out-of-core join with hash join and a single GPU.

4.1 Total Running Time

Figure 6 shows the end-to-end running time of the three GPU join algorithms under various input sizes and hardware configurations. The top three parts in Figure 6 show the result of Titan X pascal with PCI-E interconnections. Clearly, the nested-loop join is the least performant, requiring more than 2,000 seconds to join two 12-billion-record tables with one GPU as shown in Fig. 6(a). Using more GPUs improves the performance significantly, with the marginal improvement shrinks as the number of GPUs increases.

The GSM join is about one order of magnitude faster than the nested-loop join. As shown in Figure 6(b), the curve is less steep than the quadratic growth of nested-loop join. When using two and four GPUs, the improvement over one GPU is noticeable but not as significant as in the nested-loop join. As we will see in section 4.1.2, the reason for this is that the GSM join has a lower amount of work relative to its amount of data transferred, hence the performance is bottlenecked mainly by the saturated bandwidth. Due to the larger size of host side double buffer to facilitate the sort, the GSM join can only handle tables with a size up to 10 billion records.

Obviously, the hybrid join is the winner among the three, outperforming the GSM join by around 2X, as shown in 6(c). The multi-GPU speedup is more scalable than GSM join. Although it suffers from the same issue as the latter, the performance of 8-GPU surpasses that of 4-GPU from table size of 9 billion, taking only less than 40 seconds to process the 12-billion-tuple tables.

For the Global Sort Merge and Hybrid joins, wee also run experiments under eight GPUs for the PCI-E-based system, with each group of four GPUs connected to one CPU. Note that each such group of four GPUs represent a PCI-E network structure shown in Figure 1. Data transfer across the two groups is accomplished via Intel's QPI link with low bandwidth and large latency [20]. Therefore, it is expected that the use of GPUs across the QPI link will lower the performance. In our case, after the memory in the first node is used up, as shown at a table size of 6 billions along the red line in Figure 6(b), GPUs in both nodes need to access the other node's memory space. At this point, the QPI's bidirectional bandwidth can be utilized and the running time curve of 8-GPU becomes less steep. We were not able to run the Nested-Loop Join under eight cards because a key idea of the algorithm, the P2P data sharing, only works for cards connected by one socket.

The bottom three plots in Figure 6 show the results of Tesla P100 with NVLink interconnections. The curves exhibit simpler trends than those of the Titan X pascal. The running time grows proportionally across the three algorithms and various number of GPUs used, with the nested-loop join being the slowest and the hybrid join the fastest again. On average, a single Tesla P100 achieves a speedup of 1.9X, 2.9X and 2.2X over a Titan X pascal in the nested loop, global and hybrid joins, respectively. The cases of four GPUs in the Tesla P100 experiments are similar to those of the eight GPUs in the Titan X - they both are connected to two CPUs. As a result, the running time of the GSM join under four P100 is higher than in two GPUs. However, the situation is better in the hybrid join - the performance under four GPU is better than under two GPU setups.

4.1.1 Scalability. The scalability of a parallel join algorithm is one of the key factors we study. First, let us study the scalability over data size. The running time of the nested-loop join clearly grows in a quadratic manner. The increase of running time for the other two algorithms follows a pattern that is closer to linear. There is no significant difference between the growth patterns of these two algorithms.

Second, we look at the scalability of the join algorithms over different numbers of GPU devices. In terms of computing power, the performance theoretically grows linear with the number of GPUs used. However, the data transfer bandwidth and the host memory

	Hybrid Join			Global SMJ			Nested-Loop Join	
	8 GPU	4 GPU	2 GPU	8 GPU	4 GPU	2 GPU	4 GPU	2 GPU
Average	1.81	2.38	1.52	0.81	1.47	1.08	2.40	1.81
Range	1.30–2.55	1.73-2.80	1.42–1.60	0.52–1.23	1.26–1.68	1.01–1.13	2.06–2.49	1.52–1.92

Table 2: Speedup of multi-GPU runs of different algorithms over single-GPU runs on GTX Titan X pascal

	Hybrid Join		Global SMJ		Nested-Loop
	4 GPU	2 GPU	4 GPU	2 GPU	2 GPU
Average	1.97	1.58	1.12	1.36	1.76
Range	1.50 – 2.16	1.47 – 1.64	0.88 – 1.32	1.29 – 1.43	1.73 – 1.80

Table 3: Speedup of multi-GPU runs of different algorithms over single-GPU runs on Tesla P100

bandwidth do not change (for both PCI-E and NVLink systems). The impacts of shared bandwidth on performance are as follows.

Tables 2 and 3 show the speedup of multiple GPUs versus a single GPU achieved in running the three join algorithms on GTX Titan X pascal and Tesla P100, respectively. The nested-loop join algorithm is the least affected by the bandwidth due to two factors. First, its in-core computation carries a heavier weight as compared to the other two variants, minimizing the impact of data transfer (Section 4.1.2). Second, the nested-loop join takes advantage of P2P data transfer, effectively multiplying the available bandwidth even though all the data ultimately comes from host memory. As a result, the performance of two GPUs almost doubles the performance of one GPU with x86 and PCI-E, while it shrinks slightly to 1.7X with Power8 and NVlink. Four GPUs working together on PCI-E further improves the performance to 2.4X.

The hybrid join suffers more from the bandwidth issue but is able to maintain an adequate scalability. With two GPUs, the hybrid join achieves more than 1.5x speedup on both x86 and Power8 platforms. The four Titan X on the x86 achieves 2.4X speedup, the same as in the nested-loop join. Both platforms begin to suffer from additional overhead incurred by the cross-die communication when more GPUs are used, resulting in speedup under 2.0X with eight GPUs on the x86 and four GPUs on the Power8. This is because the in-core computation load contributes less to the total running time compared to the nested-loop join, leading to a higher impact of the data transfer bandwidth.

The GSM join was impacted the most by bandwidth in terms of scalability on both testbeds. Even the GPUs attached to the same CPU node cannot work together towards high efficiency. The maximum speedup only reaches 1.5X with four Titan X on the x86. In the GSM join, we need to transfer the data back and forth between the host memory and the GPUs in order to sort and merge the sublists of the input tables. These data transfers have to share the same bandwidth that does not increase with the number of GPUs. The hybrid join, on the other hand, only transfers the data a constant number of times therefore it is less affected.

4.1.2 Running Time Breakdown. Figure 7 shows the time breakdown of the three join algorithms. The statistics are gathered from a CUDA tool named *nvprof*. Note that we report time measured for each activity (i.e., shipping input data to GPUs, in-core join processing, shipping output data to host memory, and P2P transfer

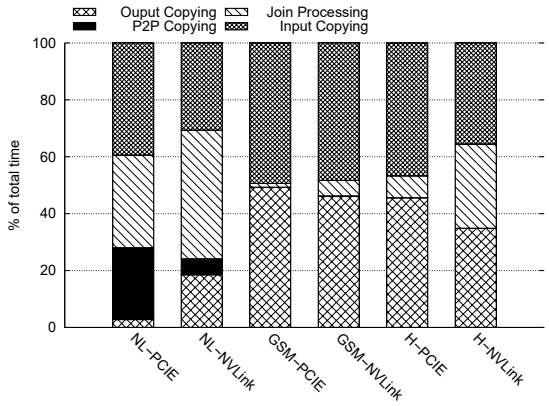

Figure 7: Running time breakdown of join algorithms under 8-billion records and 4 GPUs. NL: Nested Loop Join; H: Hybrid

between GPUs) of our algorithms. With the overlapping among such activities via CUDA streams, the sum of all time values reported here will be larger than the total running time reported in 4.1.1.

We first look at the difference between PCI-E and NVLink. In all three join variants, the proportion of time spent on data transfer is noticeably higher with PCI-E than that with NVLink. This indicates that the high bandwidth of NVLink directly impacts the performance given that the computing capabilities of the Tesla P100 and GTX Titan X Pascal are roughly the same.

Then we look at the difference among the three join variants. The nested-loop join spends 30% and 40% of the total execution time on join processing with PCI-E and NVLink respectively, while the other two variants have lower percentage. This is due to the increased amount of work for nested-loop join. Although it has the lowest percentage on data transfer, the absolute data transfer time is still much longer than the other two algorithms. The GSM join spends more than 90% of the time on data transfer as a result of sorting the whole input arrays, since sorting consumes more bandwidth than computing resources. The hybrid join has a higher percentage on join processing than the GSM join because it has linear data transfer time.

Figure 8 shows the data transfer time measured by disabling the in-core join processing code. The experimental results validate our cost analysis for the three algorithms when comparing them side-by-side. The data transmission time of nested-loop join grows quadratically as data size increases since the number of chunks M also depends on data size, while that of GSM join has a flatter curve due to the data transfer reduction by global sorting algorithm. The running time of hybrid join grows much slower, thanks to the one pass radix-partition.

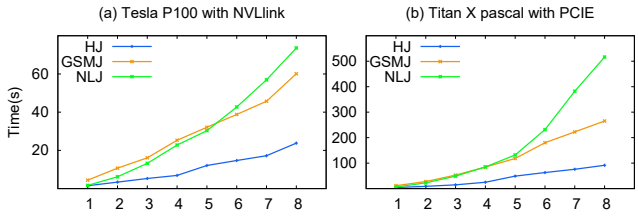

Figure 8: Data transfer time of the three join algorithms

4.1.3 The Effects of CUDA Stream Pipeline. Figures 9 shows how the 3-stream pipeline affect the performance of the join performance. In (b) and (d), the bars show the speedup of using three CUDA streams over using one stream in the GSM join under different number of GPUs on both systems. When only one GPU is used for the GSM join, more streams bring a decent 1.5X speedup in both test platforms. With more GPU used, the two platforms both benefit less from more streams with one exception where two Tesla P100 rise to more than 2X from 2 billion to 6 billion of data size, where the GPUs access local host memory region rather than other NUMA regions. In Figures (a) and (c), the hybrid join shares the same overall trend with the GSM join but with much lower speedup and more fluctuations. Using more GPUs does not always benefit from multiple streams. In (rare) cases, streams even deteriorate the performance on the Power8 platform.

By running the code with CUDA profiler, we found that the maximum data transfer speed achieved by asynchronous data transfer is about 10% lower than synchronous data transfer, indicating that asynchronous operations incurs considerable overhead. It turns out that some of the CUDA API calls (e.g., memory allocation/release) are serialized by the system even though the host code is multi-threaded. Besides, the inevitable memory allocations and deallocations also incur more synchronizations, since we cannot reuse the buffers due to the variation in input and output sizes among iterations.

4.2 Effects of Other Factors

Relative Table Size: To investigate how $|R| : |S|$ ratio affects the join performance, we conduct experiments where $|R| + |S|$ is fixed to 16 billion and four GPUs are used wherever possible (except for nested-loop join with NVLink). Figure 10 shows the relative running time fluctuation when $|R|$ varies from one billion to seven billion. The nested-loop join and hybrid join take longer to run as $|R|$ increases. However, the sort-merge join have different behaviors on NVLink and PCIE platforms. With PCIE, the running time are higher on both ends as a result of combined effects of input data transfer cost and output cost. With NVLink, the four GPUs reside in two separate CPU nodes which causes fluctuation in running time. According to the data transfer model in Section 3, it affects the three algorithms in different ways. Given that $|R| + |S|$ is fixed, we can assume there are Z total pages in R and S. For nested-loop join and GSM join, the input data transfer costs are only related to $(Z-M)M$ and $Mlog_2M + (Z-M)log_2(Z-M)$, respectively. Therefore, the nested-loop join reaches maximum cost when $|R| = |S|$. The GSM and the hybrid join are not affected by this variation.

Selectivity: Selectivity determines how many tuples are selected relative to the input size, thus affecting the output size of the join.

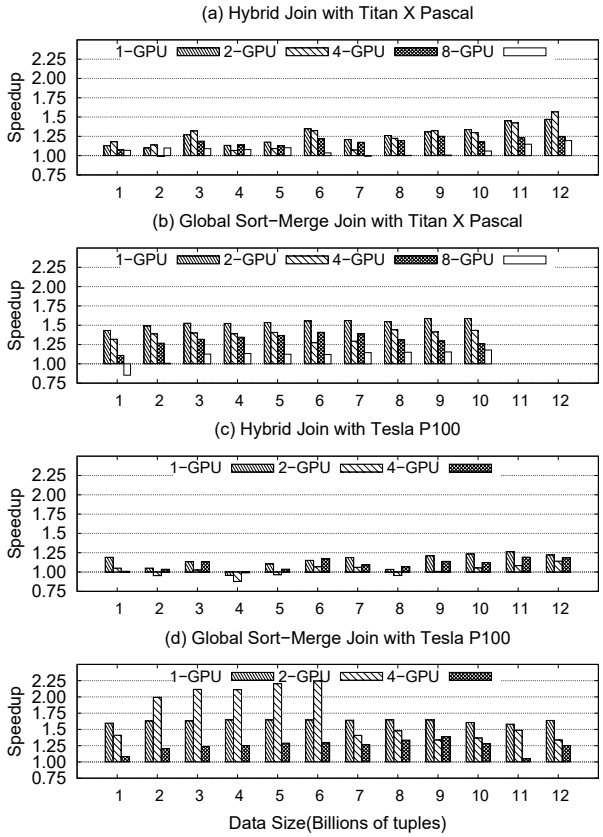

Figure 9: Speedup of 3-stream pipeline

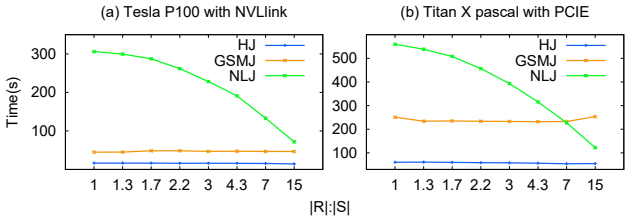

Figure 10: Running time of various $|R| : |S|$ ratio

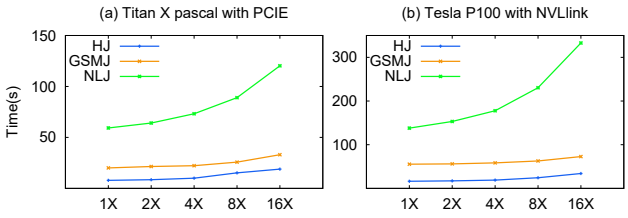

Figure 11: Running time under various selectivity

We test the effect of selectivity by varying the output ratio from 1X to 16X, and the results are shown in Figure 11. It is clear that the number of output significantly affect the running time of all the join algorithms. However, the nested-loop join is the most affected by percentage due to its running time surging quadratically. The GSM join and hybrid join have similar trends.

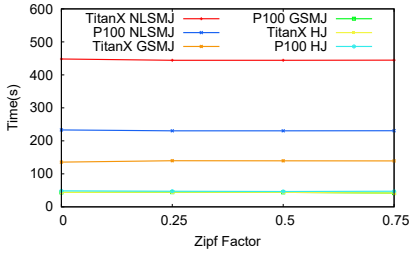

Figure 12: Running time of three GPU join algorithms against skewed data

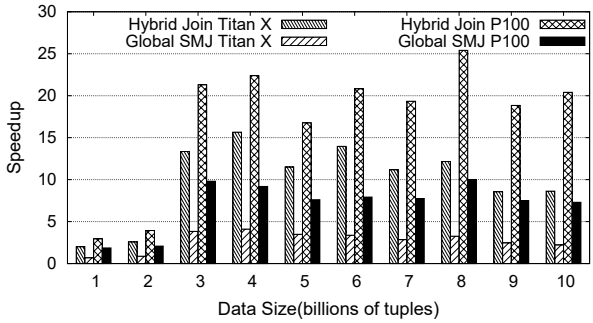

Figure 13: The speedup of our out-of-core GPU join algorithms over a CPU-based hash join algorithm

Data Skewness: We run our out-of-core GPU join algorithms against skewed data generated following the well-known Zipf distribution. Specifically, we generate join key for one of the tables with a Zipf factor ranging from 0 to 0.75. The test uses four GPUs and tables of eight billion rows. The result is shown in Figure 12, where we can see that none of the algorithms are significantly affected by the skewness of the data. The nested-loop join takes same-sized chunks in each iteration, while the global-sort-merge join utilizes merge-path partitions. Therefore these two algorithms are naturally load-balanced. The only possible issue would be in the hybrid join, where some of the radix-based partitions are considerably larger than others, potentially resulting in imbalanced loads. However, it turns out that the way that the GPUs handling the partitions actually helps deal with data skew. First, the radix partition uses the least significant bits, and creates fewer large coarse-grained partitions rather than many small fine-grained partitions for the GPUs to work on. As a result, the partitions are less different in size. Second, the GPUs take input partitions in a round-robin manner, hence each GPU works on both larger and smaller partitions, reducing the workload gap among the GPUs. Therefore the overall performance is less affected. By examining the runtime statistics, we notice that the workload difference among the GPUs is only about 10% at most in multiple runs.

4.3 Speedup Over The State-of-The-Art CPU and GPU Algorithms

We first compare the performance of the GSM join and hybrid join algorithms we proposed with the CPU-based parallel join code presented in [2]. This code is arguably the most efficient CPU-based join code, and is used to benchmark GPU-based solutions in recent studies [21, 22]. For both the CPU and GPU running time, we choose the best results among different setup (number of threads/GPUs) and report the speedup as shown in Figure 13. To ensure fair comparisons, we again use the end-to-end running time for GPU algorithms. Specifically, the CPU join algorithm uses 40 threads running on a separate system that consists of two 10-core 20-threads Xeon 2640V4 and 512GB memory. This is the best result we can obtain from available hardware. For the GPU algorithms, four Titan X Pascal are used on the x86 platform, while two Tesla P100 are used on the Power8 platform except for the hybrid join algorithm where four Tesla P100 perform the best.

From Figure 13, we can see that in most cases the GPU algorithms outperform the CPU hash join algorithm. The GSM join observes a speedup of 0.7-4.1X and 1.9-10X over the CPU on x86 and Power8, respectively. The hybrid join is of the best performance among all the GPU and CPU join algorithms tested, resulting in 2.0-15.7X and 3.0-25.4X speedup with PCI-E and NVLink over the CPU respectively. We also compared the nested-loop join with its CPU counterpart. However, the CPU code takes days to run even with the smallest test case. At table size less than 3 billion, the advantage of GPUs is smaller. As the data size increases, the multi-core CPU's performance doesn't scale as well as the GPUs. We noticed that when we run the CPU code with only one socket (i.e., 20 threads), the running time curve tends to be flatter, meaning it is faster when data size is smaller and slower when data size is larger compared to the results of using two nodes. This observation is in accordance with what we found with GPU algorithms. It indicates that the inter-CPU connection is a bottleneck that is only alleviated when the data is large enough to moderately occupy both nodes' memory. Previous work has shown that one mid-range GPU is more capable in join processing than a workstation level multi-core CPU. For example, the best GPU-to-CPU speedup for small table joins was reported in [21], with a range of 5.5-10.5X. Our work apparently enlarges the performance gap by using multiple GPUs.

We also compare our code with the state-of-art GPU join that works on big tables [19]. Their design features CPU-based multi-threaded radix partitioning and a in-core GPU hash join. A major constraint in their algorithm is that it relies on CPUs to partition the workload. It also depends on a hardcoded number of partitions which results in partitions that are larger than the global memory - this greatly limits the sizes of input tables. Our experiments by running the code show that it is only able to handle tables with 3B tuples before the GPU runs out of memory. In the cases that their algorithm does run, our hybrid join algorithm is 11% faster with one GPU at 3 billion and 190% faster with four GPUs respectively. We believe with the multi-GPU radix partitioning and efficient in-core sort-merge join, our code would scale better in larger data size.

4.4 Discussions

There have been long debates over *which join algorithm is the best.* The sort-merge join and hash join are both commonly used while the hash join is asymptotically faster. In terms of parallel join processing, the answer for the aforementioned question is more complicated since it depends on the specific hardware architecture and the parallel algorithm design and optimization. In general, the algorithm that utilizes the most stringent hardware resources more effectively would perform better. Based on our experiments, the winner is the hybrid join with significantly shorter running time

than the other two algorithms. Such results confirm the theoretical bandwidth consumption analysis in Section 3, which shows that the hybrid join is superior. However, this by no means makes the nested-loop join and GSM join obsolete, as each of them is suitable for certain types of join conditions.

5 RELATED WORK

Parallel Join Algorithms: Parallel in-memory join processing has been studied thoroughly. The focus on the CPU-based algorithms are to exploit data level and task level parallelism. SIMD instructions such as SSE and AVX on Intel processors are often used for data level parallelism. Kim *et al.* proposed sort-merge join and hash join algorithms with SIMD optimizations on a Core i7 system [12]. By comparing the two algorithms, the authors concluded that the hash join is faster and a wider SIMD instruction could benefit sort-merge join. In [5], by studying various hash join implementations, the authors found that a simple non-partition hash join with shared hash table outperformed other more complex and hardware-conscious implementations. However, this result was based on a particular dataset. In [3], Balkesen *et al.* drew an opposite conclusion. They claimed that hardware-conscious optimization is still required to achieve optimal performance in hash join. Their radix hash join implementation with the bucket chain method proposed by Manegold *et al.* [15] is the fastest. In [2], Balkesen *et al.* revisited the classic sort-merge join vs. hash join topic with comprehensive experiments and analysis. They provided the fastest implementation of both algorithms, and claimed that in most cases the hash join outperforms sort-merge join. The sort-merge join was only able to narrow the gap when the data is very large. To resolve the high memory consumption of hash join, Barber *et al.* proposed a memory-efficient hash join by using a concise hash table while not sacrificing performance [4]. Albutiu *et al.* proposed a parallel sort-merge join in which each thread works on its local sorted runs in a NUMA environment to avoid expensive cross-region communication [1].

In-Core GPU Joins: Early work on GPU-based join focuses on in-core processing of GPU-resident tables. He *et al.* designed several GPU-based database operators and join algorithms [9] that take advantage of early generations of CUDA-enabled GPUs. Rui *et al.* further improved join performance on GPUs by designing novel algorithms that take advantage of hardware and software features in newer generations of GPUs [21]. Yuan *et al.* studied the potentiality of GPUs for data warehouse use cases and provided insights on reducing the overhead caused by slow data transfer speed of the GPU [25]. Wu *et al.* proposed a compiler implementation as well as other operators for GPU-based query processing [24]. Similarly, the authors of [23] developed a pushed-based scientific data management system that features GPU data processing. Some other articles also explored the possibility of CPUs' working cooperatively with GPUs to process data [8, 10]. A number of studies also compared the performance of CPUs and GPUs in terms of join processing [9, 12, 14, 19, 21] and showed the superiority of GPUs. GPU joins were also implemented in commercial GPU-based DBMSs such as OmniSci [18] and Kinetica [13].

Out-of-Core GPU Joins: Current studies rarely address the issue of joining large-table on GPUs. The earliest study can be traced back to [11], which utilizes Unified Virtual Addressing (UVA) for controlling data transfer in early generations of GPUs. The UVA has since been enhanced with hardware page-fault and software prefetching support, and renamed as Unified Memory (UM). More recent work studied the performance of using UM in out-of-core join processing on GPUs [22]. Both studies found that the throughput achieved by UM is many times lower than a carefully designed data movement strategy.

The only work explicitly addressing large-table GPU join is found in [22] and [7]. Both work target a single-GPU setup, thus the data transmission becomes simple – only a chunk-by-chunk host-to-device transferring schedule is needed. As a result, they do not enjoy the speedup brought by multiple GPU as shown in our algorithm. Specifically, [22] overhauls the classic idea of radix hash join, and used CPU-based radix partitioning to create smaller datasets for in-core join processing. A fundamental issue is that the algorithm cannot handle data of arbitrary size (i.e., less than 2B records as seen in Section 4.3). The size of each partition exceeds the global memory size of the GPU due to a hardcoded number of partitions. In [7], sort-merge-based and hash-based in-core algorithms were used along with out-of-core radix partitioning. Unlike in [22], radix partition is conducted on the device for workload creation, but it also limits the performance of SMJ since it cannot preserve the order of the tuples. The partitions are scattered in main memory so that the data of the same partition from different chunks reside in contiguous memory space. This step carries significant CPU overhead, and is eliminated in our hybrid join via local prefix-sum and scattered read (Section 3.3). As a result, a speedup less than 2X was reported for this design, as compared to the same CPU code [2] we used as baseline in Section 4.3.

6 CONCLUSIONS

In this paper, we discussed the issues of designing multi-GPU join algorithms to exploit the computing power and minimize the overhead of multi-GPU environments. In particular, we argue that the major bottleneck in a multi-GPU environment in order to fully utilize the capability of all the GPUs is to reduce the time spent on data transfer since the GPUs nowadays have tremendous computing power but the bandwidth is limited. To that end, we designed three join algorithms with different out-of-core data transfer patterns and the same in-core join processing. By a series of experiments, we demonstrated that the inter-GPU communication and data exchange drastically affect the running time of the join algorithms. The peer-to-peer access of PCI-E devices does not help improve the performance although it enables utilization bidirectional bandwidth. Using multiple GPUs also increased join processing performance. The multi-GPU join algorithms achieved up to 2.8X speedup when using up to eight GPUs versus using just one GPU, and up to 25X speedup versus multi-core CPUs. The scalability is mainly limited by the relatively low bandwidth of PCI-E. Therefore the hybrid join with the lowest data traffic among the three algorithms we proposed achieved the best performance. Moreover, we investigate several factors that affect the performance of the join algorithms, including global memory capacity, PCI-E peer-to-peer access, CUDA streams and NUMA regions.

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proc. VLDB Endow.* 5, 10 (June 2012), 1064–1075.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. 362–373.

[4] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-efficient Hash Joins. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 353–364.

[5] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Procs. of SIGMOD* (Athens, Greece). 37–48.

[6] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Procs of ICS* (San Servolo Island, Venice, Italy). 331–340.

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

[8] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (Dec. 2009), 39 pages.

[9] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Procs. of SIGMOD* (Vancouver, Canada). 511–524.

[10] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endowment* 6, 10 (Aug. 2013), 889–900.

[11] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Procs. DaMoN* (Scottsdale, Arizona). 55–62.

[12] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1378–1389.

[13] Kinetica. 2020. *Kinetica.* https://www.kinetica.com

[14] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 451–460.

[15] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE TKDE* 14, 4 (Jul 2002), 709–730.

[16] Nvlabs. 2020. CUB. http://https://nvlabs.github.io/cub/.

[17] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. 2012. Merge Path - Parallel Merging Made Simple. In *IPDPSW*. 1611–1618.

[18] OmniSci. 2020. *OmniSci.* https://www.omnisci.com

[19] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join algorithms on GPUs: A revisit after seven years. In *Big Data*. 2541–2550.

[20] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. *Join Algorithms For Large Database Tables in a Multi-GPU Environment.* Technical Report CSE/20-289. University of South Florida, Tampa, FL.

[21] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (Chicago, IL, USA) *(SSDBM '17)*. ACM, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[22] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[23] Yi-Cheng Tu, Anand Kumar, Di Yu, Ran Rui, and Ryan Wheeler. 2013. Data Management Systems on GPUs: Promises and Challenges. In *SSDBM* (Baltimore, Maryland). Article 33, 4 pages.

[24] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Procs. CGO* (Orlando, FL, USA). Article 44, 11 pages.

[25] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endowment* 6, 10 (Aug. 2013), 817–828.