Fast Equi-Join Algorithms on GPUs: Design and Implementation

Ran Rui and Yi-Cheng Tu*
Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Ave., ENB 118, Tampa, Florida 33620, USA
{ranrui,tuy}@mail.usf.edu

ABSTRACT

Processing relational joins on modern GPUs has attracted much attention in the past few years. With the rapid development on the hardware and software environment in the GPU world, the existing GPU join algorithms designed for earlier architecture cannot make the most out of latest GPU products. In this paper, we report new design and implementation of join algorithms with high performance under today's GPGPU environment. This is a key component of our scientific database engine named G-SDMS. In particular, we overhaul the popular radix hash join and redesign sort-merge join algorithms on GPUs by applying a series of novel techniques to utilize the hardware capacity of latest Nvidia GPU architecture and new features of the CUDA programming framework. Our algorithms take advantage of revised hardware arrangement, larger register file and shared memory, native atomic operation, dynamic parallelism, and CUDA Streams. Experiments show that our new hash join algorithm is 2.0 to 14.6 times as efficient as existing GPU implementation, while the new sort-merge join achieves a speedup of 4.0X to 4.9X. Compared to the best CPU sort-merge join and hash join known to date, our optimized code achieves up to 10.5X and 5.5X speedup. Moreover, we extend our design to scenarios where large data tables cannot fit in the GPU memory.

ACM Reference format:
Ran Rui and Yi-Cheng Tu[1]. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
DOI: 10.1145/3085504.3085521

1 INTRODUCTION

The multitude of modern parallel computing platforms has provided opportunities for data management systems and applications. While CPUs are still the most popular platform for implementing database management systems (DBMSs), GPUs have gained a lot of momentum in doing the same due to its computing power, high level of parallelization, and affordability. In this paper, we present our recent work in the context of a GPU-based data management named G-SDMS [22]. In particular, we focus on the design and implementation of relational join algorithms. Our goal is to develop GPU-based join code that significantly outperform those found in literature [6, 9, 11–13, 20, 21, 24].

In the past few years, in addition to the computing capacity that has grown exponentially, the GPUs have undergone a dramatic evolution in hardware architecture and software environment. On the other hand, existing join algorithms are designed for earlier GPU architectures therefore it is not clear whether they can make the most out of latest devices in the market. Although the GPU code may scale well with the increasing amount of computing resources in newer GPU devices, maximum performance cannot be achieved without optimization towards new GPU components and features in the runtime system software. Our analysis and empirical evaluation of existing GPU join algorithms confirmed such reasoning [19]. Therefore, the objective of our work reported in this paper is a novel design of join algorithms with high performance under today's GPGPU environment. In particular, we overhaul the popular radix hash join and redesign sort-merge join algorithms on GPUs by applying a series of novel techniques to utilize the hardware capacity of latest Nvidia GPU architecture and new features of the CUDA programming framework. As a result, while our implementation borrows code for common data primitives (e.g., sorting, searching and prefix scan) from popular CUDA libraries, our algorithms are fundamentally different from existing work.

Our hash join is based on the well-known radix hash join. We used a two-pass radix partitioning strategy to reorganize the input relations. In order to increase hardware utilization, we keep a shared histogram in the shared memory for each thread block and all threads in the same block update the shared histogram via atomic operations. This reduces the usage of shared memory per thread therefore allows for more concurrent threads working together. We also assign multiple works per thread by loading more data into the large register file in the new GPU architecture. By doing this each individual thread improves instruction-level parallelism and higher overall efficiency is achieved. Previous work [12, 15] requires two scans of the inputs before writing the output to memory. To remove this large overhead, we propose an output buffer manager that enables probe in only one pass. With the help of efficient atomic operations, threads acquire the next available slot from the global buffer pointer and output independently. Finally, we take advantage of the convenient Dynamic Parallelism supported by the latest CUDA SDK to dynamically invoke additional threads to tackle skewed partitions without additional synchronization and scheduling efforts.

Our sort-merge join algorithm shares the same idea of using registers to allow more work per thread. Apart from that, our implementation heavily relies on an efficient parallel merge algorithm

[1]Tu is also affiliated with the Interdisciplinary Data Sciences Consortium (IDSC) of the same university.

SSDBM '17, Chicago, IL, USA

DOI: 10.1145/3085504.3085521

named Merge Path [10, 18] in both sort and merge stages. Merge Path partitions the data in such a way that threads can work independently with balanced load. With a linear total work efficiency, Merge Path is faster than traditional parallel merge algorithm that requires a binary search for each tuple. The sort algorithm is designed in a hierarchical manner. First, each thread sequentially sorts their own chunk of data in register. Then all the threads in the same block work together to merge their data into a list staying in shared memory. After that, all the thread blocks combine their data in the same manner in global memory. It is obvious that this method makes full use of the memory hierarchy of the GPU, especially the register file and shared memory.

We also extend our designs to the scenario of large tables that cannot fit into the GPU global memory. This is an aspect that is largely unexplored in existing work. Our strategy is to maximize the overlap between the transmission of partitions of input tables and the processing of resident data. By using CUDA streams, we divide the single workflow into two pipelines so that input data transfer and kernel execution can overlap.

Experiments show that our new hash join obtains a 2.0X to 14.6X speedup over the best implementation known to date, while the new sort-merge join achieves a speedup of 4.0X to 4.9X. Statistics provided by CUDA Visual Profiler also show that our new algorithms achieve much higher multiprocessor occupancy, higher shared memory bandwidth utilization and better cache locality. Compared with the latest CPU code, our hash join and sort-merge join are respectively up to 5.5X and 10.5X as fast. When handling data larger than the GPU device memory size, our new algorithms achieves 3.6-4.3X and 11-12.8X speedup in hash join and sort-merge join, respectively.

This paper makes the following contributions. First, we design and implement GPU-based join algorithms by optimizing various stages of sort merge and hash joins on the latest GPU architecture. Comparing with previous GPU join algorithms, our code achieves a large speedup, and the utilization of GPU resources increases considerably. It is safe to say that our join code represents the current state-of-the-art in this field. Second, we present a design of GPU joins that reduces I/O overhead in dealing with input tables that cannot be stored in GPU memory. To the best of our knowledge, this is the first reported work in joining tables beyond the memory size of GPU devices. Finally, we carry out a thorough comparison of the performance of GPU-based join algorithms and their CPU counterparts. In addition to the conclusion that GPU-based algorithms are superior over best known CPU counterparts, we provide an anatomy of such algorithms to interpret the observed results.

In the remainder of this article, Section 2 summarizes related work on parallel join algorithms; Section 3 presents the design and implementation of GPU hash and sort-merge join; Section 4 evaluates the new GPU algorithms by comparing them with existing GPU-based join and best CPU parallel join programs; and Section 5 concludes the paper.

2 RELATED WORK

Designing and optimizing algorithms for join and other database operators on many/multi-core systems has been an active topic in the database field. On the CPU side, Kim *et al.* implemented optimized sort-merge join and hash join on a Core i7 system [15]. They took advantage of the SIMD instructions available on the CPU to achieve more data parallelism. They also concluded that the hash join is faster than the sort-merge join but future SIMD instructions may bring more benefits to the latter. Blanas *et al.* [8] studied a wide variety of multi-core hash join algorithms, finding that a simple hash join with a shared hash table and no partition performs sufficiently well over other complex, hardware-conscious ones. However, their conclusion was based on a particular dataset as pointed out by [5]. Albutiu *et al.* designed a massively parallel sort-merge join where each thread only works on its local sorted partitions in a non-uniform memory access (NUMA) system [2]. In [5], Balkesen *et al.* makes a counterclaim to [8], stating that hardware-conscious optimization is still necessary for optimal performance in hash join, and provided with the fastest radix hash join implementation featuring bucket chain method proposed by Manegold *et al.* [17], which is faster than the SIMD implementation in [15]. Balkesen *et al.* later revisited sort-merge join vs. hash join with extensive experiments and analysis [4]. They provided the fastest implementation of both algorithms and claimed that the radix hash join outperforms sort-merge join with the sort-merge catching up only when the data is very large. To deal with the high memory consumption of hash join, Barber *et al.* proposed a memory-efficient hash join by using a concise hash table while maintaining competitive overall performance [7].

On the GPU side, He *et al.* designed a series of GPU-based data operators as well as four join algorithms [12]. Their algorithms were designed to take advantage of an early generation of CUDA-enabled GPUs. Bakkum *et al.* implemented an SQL command processor that was integrated into an open-source database software [3]. Yuan *et al.* studied the performance of GPUs for data warehouse queries and provided insights of narrowing the gap between the computing speed and data transfer speed [24]. Wu *et al.* proposed an implementation of compiler and operators for GPU-based query processing [23]. Kaldewey *et al.* revisited the join processing on GPU to utilize the Unified Virtual Addressing (UVA) to alleviate the cost of data transfer [14]. There are also reports of CPUs' working cooperatively with GPUs to process data [11, 13]. Close in spirit to [3, 23, 24], we are in the process of developing a scientific data management system named G-SDMS that features a push-based I/O mechanism and GPU kernels for data processing. A sketch of the G-SDMS design can be found in [22].

There are controversial views on whether GPU is superior to CPU in join processing. In [12], the authors claimed a 2-7X GPU-to-CPU speedup for various join algorithms. However, in [15], more optimized CPU code achieved up to 8X speedup over GPU joins. By studying various operators on CPUs and GPUs, Lee *et al.* claimed that GPU is about 2.5X as efficient as CPU on average [16]. Our previous work [19] showed that hardware development over the past few years affects both CPU and GPU joins. By testing the same CPU and GPU code used in [12], it is shown that the GPUs were up to 19X faster in sort-merge join and 14X faster in hash join. However, such experiments did not consider the most recent development of CPU and GPU joins. In this paper, we propose join algorithms that are optimized for the latest GPUs, and compare their performance with the best CPU code presented in [5] and [4].

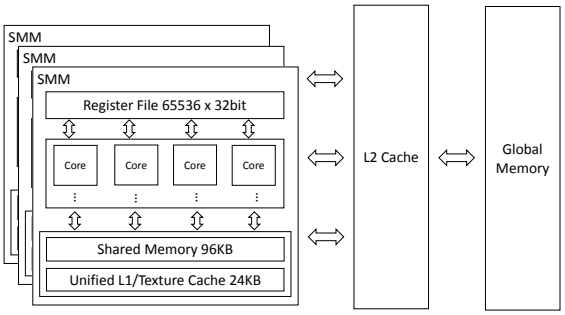

Figure 1: Layout of latest NVidia GPU architecture

3 JOIN ALGORITHM DESIGN ON GPUS

In this section, we introduce the recent development of GPU architecture, and then highlight hardware and software features that are most relevant to join processing. Based on that, we present new GPU hash and sort-merge join algorithms that take advantage of such features to effectively utilize GPU resources.

3.1 GPU Architecture

Before we discuss GPU joins, it is necessary to sketch the main components of the GPGPU environment we work on. In this paper, we focus on NVidia GPU devices and the CUDA programming model. The layout of the latest NVidia GPU (e.g., Maxwell and Pascal) architecture is shown in Figure 1. Such a GPU consists of a few multiprocessors, each of which contains 128 computing cores, a large register file, shared memory and cache system. In CUDA, the threads are grouped into thread *blocks*. Each block runs on one multiprocessor, and 32 threads form a basic scheduling unit called a *warp*. A block may contain several warps. The threads are scheduled in SIMD manner where a warp of threads always execute the same instruction but on different data at the same time.

The memory hierarchy in the GPU also has different scopes. The variables of a thread are stored in the register file and private to that thread. However, CUDA provides *shuffle* instructions that allow threads in the same warp to shared data in the registers. At block level, *shared memory* is a programmable L1-level cache that can be used for fast data sharing among threads in the same block. The *global memory*, or device memory, serves as the main memory for GPU. Although it provides up to a few hundreds GB/s of bandwidth, coalesced memory access is needed to fully utilize the bandwidth. There is also an L2 cache that buffers the global memory access for the multiprocessors.

3.1.1 New features of GPUs. The hardware design of GPUs has experienced drastic changes in recent years. This has deep impacts on our join algorithm design and implementation.

First, the number of computing cores increases steadily, giving rise to much higher GFLOPs of the GPU. The Titan X has nearly 30X more cores than that in 8800GTX, but CPU core counts only increase by 4-5X during the same period of time. Apart from the quantity, the organization of the multiprocessor has also changed over time. For example, one multiprocessor in Maxwell consists of 128 computing cores divided into four blocks. Each block of cores has dedicated scheduler with dual issue capability. This improves the efficiency of scheduling, power consumption and chip area, but requires more parallelism to achieve high utilization.

An important change is the large number of registers starting from Kepler architecture. Each multiprocessor has 64K 32-bit registers, resulting in 256KB capacity, which is larger than that of L1-level cache! This implies that the register file can hold larger amount of data, hence more work per thread is made possible at register speed. Data in registers had been set to be private to each thread, but now they can be shared among threads within the same warp via *shuffle* instructions.

Atomic operations are widely used in parallel algorithms to operate on shared data or to gather results. In early GPUs, atomic operations are supported via a locking mechanism. It is improved in Kepler via native atomic operations in global memory, and the affected memory addresses are aggressively cached (in L2 cache). Maxwell and Pascal go one step further by supporting them in shared memory. This improvement simplifies applications that need to update shared counters or pointers, and more importantly, relieves a major performance bottleneck associated with atomic operations due to the high bandwidth of shared memory.

Dynamic parallelism is another new feature available starting from Kepler. It allows an active kernel to launch other kernel calls, thus dynamically creating additional workload when the parent kernel is running. This feature enables recursive kernel calls which is not possible in earlier generations of GPUs. We will discuss in detail on how we use this feature to tackle the data skewness problem in hash join.

Creating overlaps between the processing of *in situ* data and shipping of new data inputs/outputs is a key technique in joining large tables. Such concurrency of different activities are made possible by a CUDA mechanism called *CUDA stream*. In presenting our algorithm design, we first assume the input tables can be completely placed in global memory, then we remove that assumption in Section 3.4.

3.2 Hash Join

Our hash join is based on the popular idea of radix hash. The process consists of three parts: partitioning input data, building hash table and probing. However, we adopt the idea used in [12] that by reordering the tuples in a relation according to its hash value, the partitioning and building stages are combined into one. Therefore, the tuples with the same hash value are clustered into a continuous memory space, which ensures coalesced memory access when threads load data from a certain partition. Despite this, our hash join algorithm implementation is fundamentally different from the design reported in [12] in most parts.

3.2.1 The Partitioning stage. The partitioning stage starts with building histograms for hash values to reorder the tuples of both input tables. In previous work, a thread reads and processes one tuple at a time because the multiprocessor has very few registers. This method is straightforward but is less capable of hiding latency via instruction-level parallelism. To utilize the large register file in new GPU architecture, our implementation loads VT (short for *values per thread*) tuples into registers of the thread all at once so that each threads are assigned more workload at the beginning.

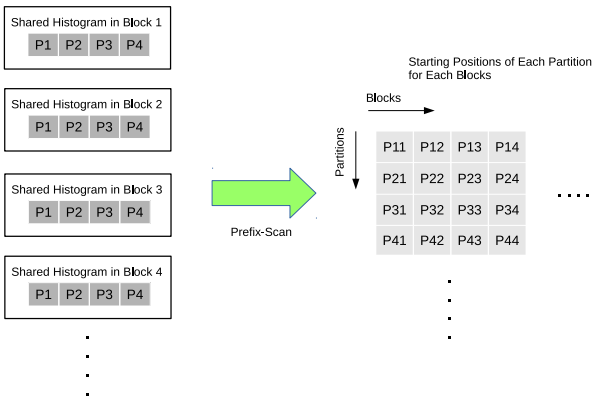

Figure 2: Shared histogram used in Partitioning and Reordering in GPU hash join. *(i,j)* is the shared histogram for partition *i* in block *j*. The prefix-scan of the histograms *P(i,j)* gives the starting position for the data in block *j* that belongs to partition *i*

This increases the instruction-level parallelism within each thread, and the memory access can be overlapped with computation to hide latency. Each thread processes its own data independently and updates the shared histogram in shared memory (Fig. 2).

Being different from the method in [12], where each thread keeps private histograms for each partition in shared memory, our algorithm keeps only one shared copy of histogram in each thread block, as Algorithm 1 shows[3]. In early generation of GPUs, atomic operations are either not supported or involve considerable overhead. It was not feasible to update shared histogram among a number of threads. The problem with keeping private histograms in each thread is that it would consume too much shared memory when either the number of threads in each block or the number of partitions is high, reducing the number of active threads running on each multiprocessor (i.e., called *occupancy*). This might not be a serious issue in old devices such as 8800GTX. Since they only have 8 cores per multiprocessor, a small number of threads are enough to keep it busy. However in newer architectures, more concurrent threads are required to keep the hardware at optimal performance. By using one shared copy of the histogram, the amount of shared memory consumed by a block is reduced by a factor that equals the block size, and is no longer depending on the number of threads in a block, resulting in more active threads for multiprocessors. Also thanks to native atomic operation support on shared memory in Maxwell and Pascal, all the threads in a block can update the shared histograms with a very small overhead.

In previous work, a multi-pass radix, or a variable number of pass partition is used. However, in this method we found there is a non-linear growth of number of partitions with the table size increasing. This results in a non-linear execution time increase. We adopt a two-pass radix partition mechanism in our implementation. We keep the partition size to be small enough (e.g., less than 512 tuples for each thread block) to fit into shared memory, therefore the probe stage only needs to read the data once from the global memory. To achieve such small partition for large input, we have

[3]All pseudocode is written from the perspective of a single thread, following the Single-Program-Multi-Data (SPMD) programming style in CUDA.

Algorithm 1: Histogram in GPU Hash Join

Require: Relation *R*
Ensure: array of histograms *SharedHisto[]*
1: Initialize SharedHisto[nPartitions] to 0;
2: data[VT] ← load VT tuples from relation R;
3: **for** i = 0 to VT-1 **do**
4: h ← Hash(data[i].key);
5: atomicAdd(SharedHisto[h],1);
6: **end for**
7: Write SharedHisto[nPartitions] to global memory;

to create a large number of partitions. If a single-pass method is used, the shared memory is not able to hold that many histograms. Thus, we use a two-pass method where the first pass reorganizes the input into no more than 1024 partitions, and the second pass further divides the partitions from the first pass into smaller ones. By using this method, we can process a single table containing 500 million pairs of integers (key+ value). This is a reasonable size since in our experiment the Titan X with 12GB memory can hold two 128 million-tuple arrays plus intermediate data.

To reorder the tuples (Algorithm 2), each thread block has to know its starting positions of the partitions. The shared histograms are copied to global memory. Then a prefix scan is performed to determine the starting position of all the partitions for each block (Fig. 2). Once the positions are obtained, all the threads can reorder the tuples in parallel by atomically incrementing the pointers for each partition. Since our method uses shared histogram and its prefix sum, the writing positions of the threads in the same block are also localized. This increases locality of memory access, thus the cache would be in use to buffer the writes.

Algorithm 2: Reorder in GPU Hash Join

Input: relation *R*
Output: reordered relation *R'*
1: SharedHisto[nPartitions] ← load the exclusive prefix sum of the histogram from global memory;
2: Synchronize;
3: data[VT] ← load VT tuples from relation *R*;
4: **for** i = 0 to VT-1 **do**
5: h ← Hash(data[i].key);
6: //get current writing position and then increment
7: pos ← atomicAdd(SharedHisto[h],1);
8: R'[pos] ← data[i];
9: **end for**

3.2.2 The Probe stage. In the *probe* stage (Figure 3), each partition of input table *R* is loaded into shared memory by one block of threads. A partition of the other table *S* with the same hash value is loaded into registers by the same threads. This is the same mechanism mentioned in previous section, thus every access to partitions of *S* is at register speed. To write the outputs back to memory, the traditional wisdom (as in [12] and even CPU work such as [15]) is to perform the probe twice. The first probe returns the number of outputs for each partition to determine the location of the output buffer for writing outputs. The total number of outputs and starting position of each partition is obtained by a prefix scan of

these numbers. Given the number of outputs, the output array can be allocated and then the second probe is performed to actually write the output tuples. This scheme eliminates the overhead of synchronization and dynamic allocation of buffers, and efficiently outputs in parallel by doing more work. The pseudocode of such a design of probe is shown in Algorithm 3.

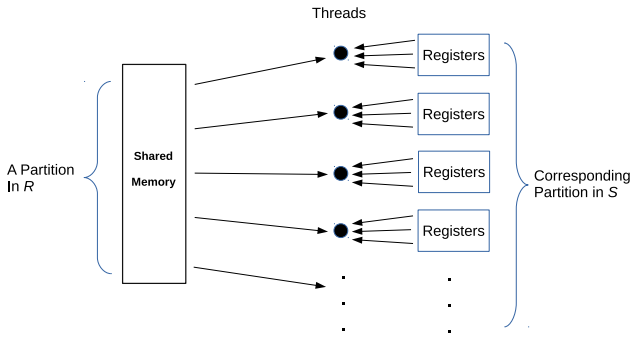

Figure 3: Workflow of threads of probe stage in hash join

Direct data output: However, we realize that the overhead of probing twice is too high. To reduce such overhead, we design a *buffer management* mechanism in which threads directly output to different locations of a buffer pool in global memory (Fig. 4). We first allocate an output buffer pool of size B and divide it into small pages of size b. A global pointer P holds the position of the first available page in the buffer pool. Each thread starts with one page and fills the page with output tuples by keeping its own pointer to empty space in the page. Once the page is filled, the thread acquires a new page pointed to by P via an atomic operation and increment P. With the direct output buffer, threads can output directly in the probe stage in parallel and no complex synchronization is needed. We basically trade the cost of acquiring new pages for elimination of the second probe. Since the atomic operation only happens when a page is filled, we expect little conflicts in accessing the global pointer P. Plus, we can adjust the page size b to reach the desirable tradeoff between such overhead and buffer space utilization (i.e., larger page reduces overhead but may render more empty space within pages).

To tune the output buffer even more aggressively, an alternative is to divide the whole output buffer into chunks. Each thread block is assigned one chunk for output their results. Each block keeps a pointer in the shared memory that redirects to the next available slot in the output chunk. When a thread in a block needs to output, it acquires the current value of the pointer in the shared memory and increases it via an atomic operation, then it outputs the result to the available slot. This technique will take advantage of low cost of atomic operations against shared memory locations.

Algorithm 3: Probe in GPU Hash Join

Input: relations R and S
Output: array of matching pairs *globalPtr*;number of matches for each block *matches*;

```
1:  pid ← blockIdx.x; //Partition id
2:  while pid < nPartitions do
3:      matches ← 0;
4:      SharedBuf[VB] ← load partition pid of R;
5:      Synchronize;
6:      data[VT] ← load VT tuples from partition pid of relation S;
7:      bufPtr ← atomicAdd(globalPtr,bufSize);
8:      count ← 0;
9:      for i = 0 to VT-1 do
10:         for j = 0 to VB-1 do
11:             if Hash(data[i].key) == Hash(SharedBuf[j].key) then
12:                 bufPtr[count++] ← (data[i],ShareBuf[j]);
13:                 if count == bufSize then
14:                     bufPtr ← atomicAdd(globalPtr,bufSize);
15:                     count ← 0;
16:                 end if
17:             end if
18:         end for
19:     end for
20:     pid ← pid + NumBlocks;
21: end while
```

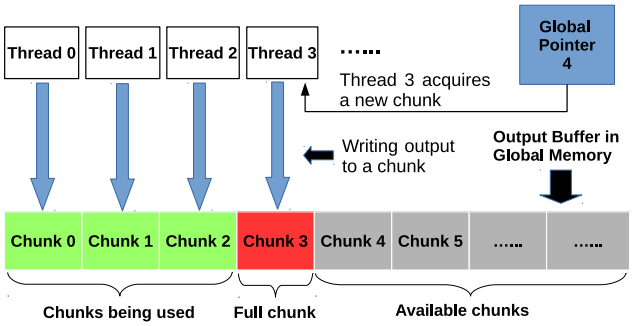

Figure 4: A case of direct output buffer for GPU hash join, showing Thread 3 acquiring chunk 4 as output buffer

3.2.3 Skew Handling. Our hash join design takes data skew into consideration. Here by "data skew" we mean some of the partitions based on the hash value can be larger than others. In extreme cases, most of the data are distributed in just a few partitions. As a result, the corresponding thread blocks in the probe stage become the bottleneck of the whole procedure. To deal with data skew, previous work processes these skewed partitions in a separate kernel function that provides more working threads for the extra data. This method is simple and efficient, but needs to keep more intermediate states for scheduling.

In our implementation, we take advantage of dynamic parallelism that was introduced since Kepler architecture. This technique allows dynamic creation of additional kernels within current workflow. If the size of a certain partition exceeds the predefined threshold, the block that is processing this partition creates a child kernel that exclusively works on this partition. The child kernel runs concurrently with the parent kernel and other child kernels until it finishes. Then it returns to its parent thread. We can dynamically change the launching parameters of the child kernels (i.e.

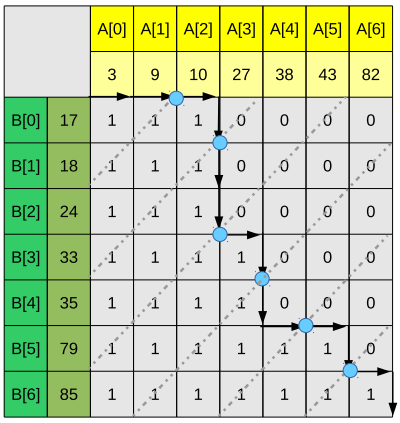

Figure 5: Parallel merge with 7 threads using Merge Path

block size and grid size) according to the sizes of their corresponding partitions. This technique brings more flexibility for dealing with skewed data.

3.3 Sort-Merge Join

As usual, sort-merge join is divided into two stages: (1) sorting the input relations by the attribute(s) involved in the join condition; and (2) merging the two sorted relations to find matching tuples.

3.3.1 The Sort stage. Our program features a highly efficient parallel merge-sort algorithm. Previous work often implements radix sort [25] or bitonic sort [12] that are also suitable for parallel computing. However, they both have limitations in that the radix sort only applies to numeric data and it becomes costly as the key size grows, while the bitonic sort has a unique pattern of comparison which requires power-of-two number of data points. Merge-sort can sort any type of data and are more flexible on data size than bitonic sort. Although bitonic sort in serial code has low time complexity ($O(\log^2 n)$), its best parallel version has a subpar $O(n \log^2 n)$ total computation [1]. It is also hard to exploit locality and coalesced memory access when data is large as it accesses different locations each time. Merge-sort, on the contrary, merges two consecutive chunk of data at a time, which can utilize the register blocking, coalesced global memory access and shared memory of the GPU. According to our experiments, this highly efficient use of memory bandwidth results in a 7X speedup compared with the bitonic sort in existing work.

Our sort is based on a parallel merge algorithm named Merge Path [10, 18], the main idea of which is shown in Fig. 5. Consider the merge of two sorted arrays A and B, a merge path is the history of the merge decisions. It is more clearly illustrated by a $|A| \times |B|$ matrix, in which an element (i,j) is 1 when $A[i] < B[j]$, and 0 otherwise. We can obviously see that the merge path lies exactly on the boundary between the two regions containing only 0s and 1s, respectively. If we break the merge path into equal-sized sections, the projections of each section on A and B arrays correspond to the elements to be merged by this section, thus each section can merge their own data independently. The most essential part in this method is how to find the merge path without actually carry out the merging process. To find the merge path, we need the help of cross-diagonals, which are the dash lines in Fig. 5. By performing binary searches on the pairs of A[i] and B[j] along the cross-diagonals of the matrix, where $i + j$ equals to the length of the corresponding cross-diagonal, we obtain the intersections of the merge path and the cross-diagonals. These intersections provide the starting and ending points of each sections of the merge path. As the sections are equal-sized, load balancing would be naturally achieved without additional effort. Based on this highly parallel and load-balanced merge procedure, efficient merge-sort algorithm can be realized on GPUs.

Algorithm 4: BlockSort

Input: Input relation *R*;
Output: Sorted sublists;
1: data[VT] ← load VT tuples from relation R;
2: sort data[ ] sequentially;
3: copy data[ ] to shared memory;
4: **for** n ← 2, 4, 8, ..., BlockSize **do**
5: L ← VT×n/2
6: find the merge path of two sorted data[ ] of length L;
7: merge the two sorted data[ ] into one list of length 2L in shared memory with n threads cooperatively;
8: **end for**
9: Store the sorted tuples to global memory;

In our sort stage, input relations are first partitioned into small chunks of size VT. Then each thread loads a chunk of input data into its registers as an array using static indexing and loop unrolling to achieve more efficiency, as shown in Algorithm 4. That is to access the array using for loops in a sequential way. This method ensures the whole chunk resides in registers as long as the number of registers needed does not exceed 256 per thread. Each thread performs sequential odd-even sort on its own chunk and stores the sorted chunks into shared memory. Since VT is set to 8 after some tests for optimal performance for the GTX Titan X, the overhead of using odd-even sort on data sitting in registers is acceptable. After each thread has their own chunk sorted, all the threads in a thread block work cooperatively to merge the chunks in shared memory using Merge Path until they become a single sorted array. Then all the blocks store their outputs to global memory and cooperatively merge the arrays using Merge Path again, until the whole relation is sorted (Algorithm 5). The arrays are loaded into the shared memory, and each thread executes serial merge independently on their own partitions, and stores the merged list to registers which is to be output later to global memory in batch. In summary, our sort stage relies heavily on registers (in BlockSort) and shared memory, which were of much smaller volume in early GPUs.

3.3.2 The Merge Join stage. In the merge join stage, the two sorted relations are treated as if they were to be merged into one list. Previous work first partitions relation *R* into small chunks that fit into the shared memory, then searches the other relation *S* for matching chunks. Each tuple in a chunk of *S* finds matches using binary search on the corresponding chunk of *R*.

In our implementation, the Merge Path method is used at this stage as well. To find matching tuples, we start from partitioning the input relations using merge path so that each thread can work

Algorithm 5: Merge Data from different blocks

Input: sorted sub-arrays of size VT×BlockSize;
Output: a single sorted list;
1: VB ← VT×BlockSize;
2: **for** n = 2, 4, 8, ..., NumBlocks **do**
3: L ← VB×n/2;
4: find the merge path of two sorted sub-arrays of length L;
5: dataShared[VB] ← corresponding partitions of sub-arrays for current block;
6: merge the tuples in dataShared[ ] into one list of length 2L to registers;
7: store the sorted list to global memory;
8: **end for**

on individual chunks of the input. After loading the corresponding chunks from the two inputs into register, each thread loops over each elements of R and runs merge path to find the starting point(e.g. the lower bound) of matching in S. This procedure resembles a serial merge of two sorted lists, thus the total work of all threads is linear to the number of inputs. The second step is similar to the first one, except that this step is to find the starting point of matching of R for each elements in S, which is exactly the ending point (e.g., the upper bound) of matching in S for tuples in R. By subtracting the starting position from the ending position, the number of matches for each tuple in R is obtained. Before output results, a prefix scan on the array of number of matches gives the total size for allocating output buffer. Since we know where to find the matches, a second scan is no longer needed in the output stage.

3.4 Handling Large Input Tables

So far we have made the assumption that both tables as well as the intermediate results of the join can be put into the GPU global memory. This sets a limit on the size of tables that can be processed. In this section, we report our efforts in removing that assumption. Following the ideas of disk-based joins, we can obviously break the input tables into chunks and process pairs of chunks (one from each table) in a GPU using the aforementioned join algorithms. Join results of each pair of chunks are written back to host memory.

The first aspect is how to schedule the shipping / processing of different data chunks to/in the GPU. Note that a thorough study has to consider the relative table sizes and the number of GPU devices. In this paper, we focus on the following scenario: there is only one GPU, table R can be completely stored in the global memory while table S is of an arbitrarily (large) size. Such a scenario represents a typical business database design such as the one found in TPC-H. Furthermore, solutions developed for such will build the foundation for more complex scenarios. Given that, we first load R entirely into GPU, and join R with each and every chunk of S, and ship results back to host memory. Apparently, as R resides in GPU, we conduct the first stage (e.g., partition, sorting) of the join only once for R.

Another aspect is to hide the data shipping latency with join computation on the device. In particular, we take advantage of the CUDA Stream mechanism to allow concurrent data transfer and kernel execution between neighboring rounds of chunked joins (Fig. 6). Specifically, each chunked join involves a kernel launch, and the series of kernel launches are encapsulated into CUDA streams. After table R is transmitted to GPU memory, the kernel for processing (i.e., sorting or building hash) R and the transfer of S_1 are issued simultaneously. When the join results C_1 are being written back to the host, the shipping of S_2 happens at the same time. In this way, the work flow is pipelined and the overlapping of kernel execution and data transfer helps reduce the total running time.

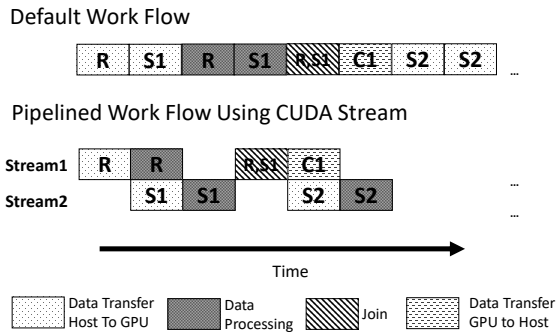

Figure 6: Overlapping data transmission and join processing using two CUDA streams

We also worked on the scenario of processing joins in multiple GPU devices. It involves innovative data transmission scheduling among the different GPU cards as well as between the card and host. Note that the two types of transmission are done in different physical PCI-E channels therefore we can handle cases in which one table can only be placed in multiple GPUs without much performance penalty. Due to page limits and the complex techniques involved, we unfortunately have to skip such details. We leave the study of joins between very large tables (such that neither table is smaller than the aggregated memory size of multiple GPUs) as future work.

4 EVALUATIONS

We evaluate the performance of our GPU-based join algorithms by comparing them with existing GPU and latest CPU join code. In addition, we also show the effects of different factors on the performance. The hardware and software configurations are described in Section 4.1.

4.1 Experimental Setup

We choose two Intel CPUs and two NVidia GPUs for our experiments, and the specifications of the hardware are listed in Table 1. The E5-2650v3 and Titan X represent a recent generation of their kind while the E5-2670 and Titan represent high-end hardware that are 3-4 years old. Plus, the corresponding CPU and GPU hardware have very similar price tags. The E5-2630v3 and E5-2670 are installed on two separate servers running Red Hat Linux under kernel version 2.6.32 and GCC version 4.4.7. The GPUs are connected via PCI-E 3.0 16X interface to the same server that hosts the E5-2630v3. Our GPU code is compiled under NVCC 7.5. We also use an NVidia tool named *NVProfiler* to study the runtime characteristics of our GPU code. To maximize the performance of the CPUs, we run 16 threads for the CPU code, which is the optimal number obtained from a series of tests.

Table 1: Specifications of hardware mentioned in this paper. Information is mainly extracted from the Intel and Nvidia corporate websites, with other information obtained from www.techpowerup.com and www.cpu-world.com

Device	CPU		GPU	
	Xeon E5-2630v3	Xeon E5-2670	Maxwell Titan X	Kepler Titan
Clock Rate	2.40GHz	2.60GHz	1.00GHz	0.84GHz
Core counts	8	8	24×128	14×192
L1 Cache	256KB	256KB	96KB×24	64KB×14
L2 Cache	2MB	2MB	3MB	1.5MB
L3 Cache	20MB	20MB	–	–
Memory*	128GB DDR4	64GB DDR3	12GB GDDR5	6GB GDDR5
Memory Bandwidth *	59GB/s	51.2GB/s	337GB/s	288GB/s
Max GFLOPS	153.6	166.4	6144	4494
* For CPUs, here we refer to the host memory of the computer. For GPUs, we mean the global memory.				

Unless specified otherwise, we set the two input relations to be of the same size. Each tuple in the tables consists of two parts: a 32-bit integer unique key and a 32-bit integer payload that serves as the ID of the tuple. The keys are first generated in order and then shuffled randomly. The keys are uniformly distributed between 1 and table size N. We perform equi-join on the key, the selectivity of the join condition is set to render one output item per tuple.

We first report results on in-memory join where the data size fits the capacity of GPU memory. We compare our code with existing GPU join algorithms and the latest CPU join code, and go through different factors that potentially affect join performance. Finally, we use the GPU to handle large data that exceeds its memory capacity, and compare its performance with CPU.

4.2 Experimental Results

4.2.1 Comparing with Existing GPU Code. Defining the appropriate baseline for such experiments has been surprisingly difficult. After a thorough investigation of the known related work, our comparisons are focused on the GPU join programs presented in He *et al.* [12]. Among the multitude of studies on GPU database systems, few discussed join algorithm design and implementation. Others [23, 24] focus on query engine without clearly modularized code for joins. Another work [14] aims at improving data transmission efficiency by UVA while uses the code of [12] as building blocks. Therefore, we are confident that [12] is by far the most up-to-date and systematic work on GPU-based joins. Plus, their code is also used by CPU-based parallel join work [15] as a comparative baseline. Our attempts to extract and test standalone join code from the work of [23] and [24] failed due to compilation errors and lack of documentation to help fix such errors.

According to Fig. 7, our GPU code significantly outperforms that introduced in [12]. Specifically, the new sort-merge join achieves 4.0-4.9X speedup, with speedup goes slightly higher as the data size increases. On the other hand, a 2.0-14.6X speedup is observed for the new hash join. The same results can be seen in both the Maxwell Titan X and Kepler Titan cards. Only issue is that due to the small

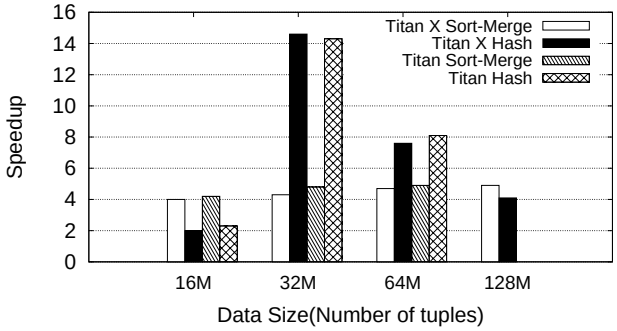

Figure 7: Speedup of new GPU join algorithms over existing GPU code under different table sizes

global memory of Titan (6GB), the join code cannot run under a 128M table size. The large variation of the speedup in hash join is caused by the partitioning strategy of the old code. In particular, when table size reaches 32 million tuples, the partitioning process changes from two-pass to three-pass in order to keep each partition small. This results in a sudden increase of running time. In contrast to that, the new hash join generates more partitions per pass thus we ensure two passes is enough for a large range of data sizes. As a result, its running time grows proportionally to the input size.

Resource Utilization of join code: To get insights on the big performance gap between old and new joins, we study the GPU resource utilization achieved by major kernels in both pieces of code. Such data are collected via NVProfiler and presented in Tables 2 and 3. Note the block sizes shown represent those that deliver the best kernel performance. For sort-merge join (Table 2), the old code used a bitonic sorting network that directly operates on global memory. Only when sorting a partition of the data (kernel *PartBitonic*), the shared memory is used but only 50% bandwidth (1586GB/s) is utilized. When combining all the partitions (kernel *Bitonic*), the accesses to the global memory are entirely random and non-coalesced. Although these kernels have relatively high multiprocessor occupancy (e.g., the number of threads that can run at the same time on a multiprocessor), they are bounded by the utilization of shared memory and bandwidth of global memory, respectively. On contrary, our new sort-merge join makes every step local to the threads. In the *blocksort* kernel, each thread sorts their own items in registers in a sequential manner with zero latency. Then the the whole block of threads combine their tiles together in the shared memory. Even though the occupancy of this kernel is only 62%, the nearly 100% (3.3TB/s) bandwidth utilization on the shared memory ensures the overall performance. Furthermore, all the merging operations are also completed in shared memory. Finally, all the data are in order and can be output to global memory efficiently with coalesced access.

For hash join (Table 3), the main problem with the old code is the unbalanced use of GPU resources. In particular, due to the lack of atomic operations in older GPUs, each thread keeps its own copy of an intermediate output (i.e., histogram of radix partition) in the shared memory. As a result, in the *Histogram* and *Reorder* kernels, only eight threads can be put into each block. That is even smaller than the basic scheduling unit of the GPU, which is 32 threads (a warp) at a time. Because of that, only 16% occupancy is

Table 2: Resource utilization of major kernels in the new and old GPU sort-merge join code

Kernel	New Algorithms		Existing Algorithms	
	BlockSort	Merge	partBitonic	Bitonic
Block Size	256	256	512	512
Registers/Thread	41	31	16	10
Shared Memory/Block	9KB	9KB	4KB	0KB
Occupancy Achieved	62.1%	98.8%	93.2%	84.8%
Shared Memory Bandwidth Use	3308.2GB/s	1098.6GB/s	1585.9GB/s	0GB/s
L2 Cache Bandwidth Use	84.6GB/s	295.3GB/s	110.1.0GB/s	262.6GB/s
Global Memory Bandwidth Use	84.5GB/s	253.3GB/s	109.5GB/s	262.9GB/s

Table 3: Resource utilization of major kernels in the new and old GPU hash join code

Kernel	New Algorithms			Existing Algorithms		
	Histogram	Reorder	Probe	Histogram	Reorder	Probe
Block Size	256	256	256	8	8	128
Registers/Thread	13	20	22	14	16	18
Shared Memory/Block	4KB	4KB	4KB	8KB	8KB	4KB
Occupancy Achieved	87.6%	89.1%	91.0%	16.6%	16.4%	83.1%
Shared Memory Bandwidth Use	201.5GB/s	19.5GB/s	775.3GB/s	275.9GB/s	85.6GB/s	637.3GB/s
L2 Cache Bandwidth Use	357.3GB/s	171.3GB/s	28.3GB/s	36.4GB/s	59.8GB/s	28.6GB/s
Global Memory Bandwidth Use	103.2GB/s	98.1GB/s	8.5GB/s	36.4GB/s	58.9GB/s	23.3GB/s

achieved by these kernels, meaning that the multiprocessors are extremely underutilized. In our redesigned hash join kernels, both the histogram kernel and reorder kernel achieve more than 87% occupancy. With the help of atomic operation, one copy of shared histogram is kept for a block, thus only 4KB of shared memory is used even for a block size of 256. Writing to global memory is also improved because of the shared histogram. All threads in a block write to a limited space of the output. This increases locality thus the utilization of L2 cache increases. In both sort-merge and hash joins, use of registers per block has increased significantly to take advantage of the large register file in the latest GPU.

Hash join vs. sort-merge join: Previous work [4, 15] concluded that hash join is more efficient than sort-merge join in current CPU hardware, while the latter would benefit from wider SIMD instructions. For GPUs, the key to this problem is the utilization of the memory system. The sorting stage in the sort-merge join relies heavily on the fast shared memory and register file to reorganize the inputs. However, the radix partition of the hash join has more random access, thus is hard to be localized into shared memory. At best, the memory access can be cached by L2, but its bandwidth is one magnitude lower than that of shared memory. Therefore, in our code the sort-merge join is up to 26% faster than the hash join.

4.2.2 Comparing with latest CPU code. The CPU code we use for our comparisons are developed by Balkesen and co-workers [4, 5], which is obviously the most efficient parallel developments for both sort-merge and hash joins. Fig. 8 shows the relative performance of our GPU code to the latest CPU-based joins. We first want to point out that the older E5-2670 outperforms the newer E5-2630v3 in all cases but the newer Titan X GPU is always the winner. Therefore, the relative performance between Titan X and E5-2630v3 shows the maximal GPU-to-CPU speedup while that of Titan to E5-2670

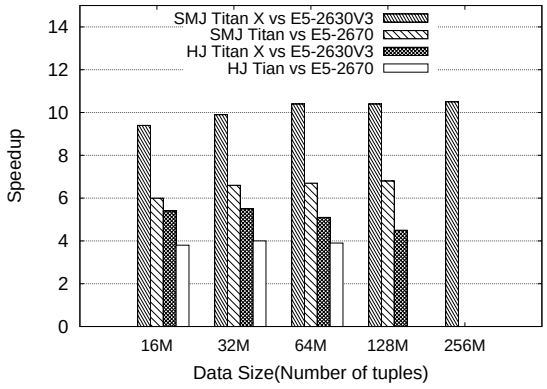

Figure 8: Speedup of our GPU code over the latest CPU code

shows the minimal in our tests. Clearly, the GPUs outperform CPUs in both sort-merge join and hash join by a large margin. In sort-merge join, the Maxwell Titan X achieves more than 10X speedup over the Haswell E5-2630V3, while the Kepler Titan has up to 6.8X speedup over the Sandy-Bridge E5-2670. In hash join, the advantage of GPUs shrinks but is still considerable, our code running on Titan X achieves a 5.5X speedup over the E5-2630V3, while the Titan obtains a 4.0X speedup over the E5-2670.

In terms of performance improvement between two generations of hardware, the GPUs see more benefit. The Maxwell Titan X improves by 22% and 35% in overall performance over the Kepler Titan for sort-merge join and hash join, respectively. This can be easily interpreted as the result of the computing capacity of new generations of GPUs that increased significantly over the past few years (Table 1). On the CPU side, the newer Haswell E5-2630v3 is even 26% and 2% slower than the older E5-2670 in sort-merge

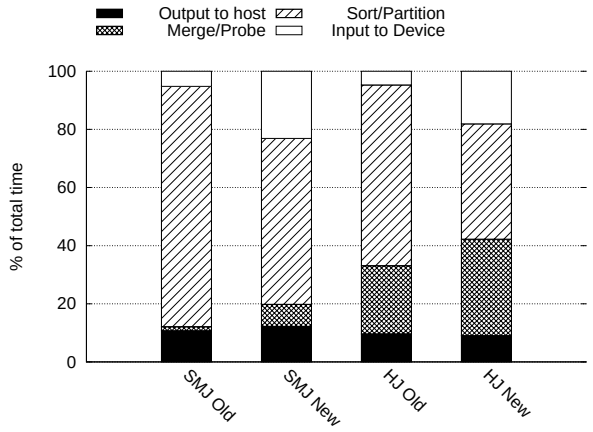

Figure 9: Execution time breakdown (percentage) of new and old GPU algorithms running on Titan X

join and hash join, respectively. This shows that the architectural update on CPUs does not bring any performance advantage in join processing. Although the E5-2630v3 works on a new generation of memory (i.e., DDR4), the higher clock rate of E5-2670 cores actually makes better use of the memory bandwidth.

4.2.3 Time Breakdown. The execution time breakdown of our GPU code and that provided by [12] is shown in Fig. 9. The first thing we notice is that the transmission of input/output data to/from GPU is an extra cost for the GPU code, and it counts for 35% and 27% of the total time in the new sort-merge join and hash join, respectively. Since the join kernels of sort merge is faster than hash join, the data transfer time takes up higher percentage in hash join – almost 1/3 – of the total execution time.

When comparing the new algorithms with the old ones, we find that the join processing time in new code contributes less to the total running time while the data transfer time contributes more. In sort-merge join, the percentage of sorting stage time drops from 82.7% to 57.1%, which corresponds to a 7X of performance speedup. The merge-join is, however, not a time consuming stage, taking up less than 8% of execution time. The reason why the merge-join stage in our new code is a little slower is that the old code uses a different mechanism. It builds tree indexes for one of the input relation after sorting. The merge stage gained some benefit from the indexes. But our sort-merge join is still much faster in terms of GPU processing time. In hash join, both partition and probe stages are much faster than existing code, achieving 6.2X and 3.8X speedup respectively. The results indicate that our newly designed kernels are more efficient than those in the existing code by using optimizations that take advantages of the new GPU architectural features. If we do not consider the time for data transfer between host and GPU, both sort-merge and hash in GPU will get a much higher speedup. For sort-merge the speedup would become 15.5-17.5X while for hash join it is 6.3-8.3X. Obviously, a GPU is way more efficient than a CPU in processing the join itself but gets a big hit in data communication via the PCI-E bus.

4.2.4 Effects of Join Selectivity. Fig. 10 shows the impact of varying selectivity, i.e., the total number of output tuples. The GPU

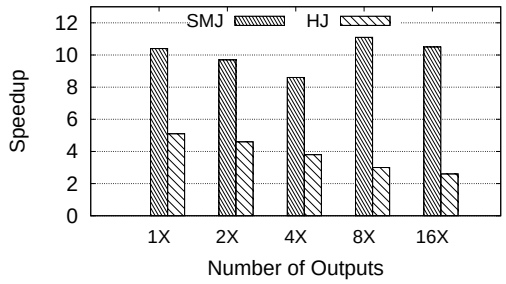

Figure 10: Impact of join selectivity on speedup of Titan X over E5-2630v3 under data size 64M

sort-merge join enjoys a speedup around 10X over the CPU except at 4X of outputs where it drops to 8.5X. On the other hand, the GPU hash join suffers from the increasing outputs with a decreasing speedup over the CPU from 5.1X to 2.6X. It is expected that when more tuples are generated as a result of the join, the GPU program will bear a higher overhead as more data will be written back to host via the PCI-E bus. This explains why the hash join performance degrades. However, the impact of selectivity on sort join performance does not seem obvious. By scrutinizing the behavior of our code, we found that the actual running time of our sort merge code does increase as more output tuples are returned. On the other hand, due to a special design of a data structure for holding output tuples, the CPU-based sort-merge join code sees serious performance cut when the output size increases.[7] This overshadows the performance loss observed in GPU code therefore the GPU-to-CPU speedup stays on the same level. As a general trend, we believe lower selectivity will hurt the performance of GPU programs to a extent that there is no competitive advantage of GPUs, as we discussed earlier in 4.2.3. But our strategy of overlapping data transmission and join processing can also offset such effects.

4.2.5 Effects of Direct Output. By using the direct output buffer, the hash join sees a significant benefit. Fig. 11A shows the results of our hash join code comparing with the same code without using a direct output buffer. Under page size of one, improvement starts with 25% under 16M data size and, as the input data becomes larger, the improvement gradually drops down to 20%. Such drop is due to the increase of atomic operations to acquire the pointer to the buffer in global memory. When the input size increases, the number of output tuples also grows proportionally. Each thread has to request more chunks to store the output, thus increases the number of atomic operations, as a overhead to the code. For the data sizes we tested, the overhead is acceptable. We test this technique with the sort-merge join as well, but it does not improve the performance because the join stage in sort-merge join is different from that in hash join. A linear search is used for the sorted data to determine the range of the output without scanning the whole table, so it saves more time compared with the double-probing approach in the hash join.

We also ran tests to determine the optimal page size for the output buffer. To our surprise, small page sizes of one or two helps

[7] To be fair, this is likely a small problem that can be easily fixed. However, we decided to keep the CPU code as intact as possible for a more accurate comparison.

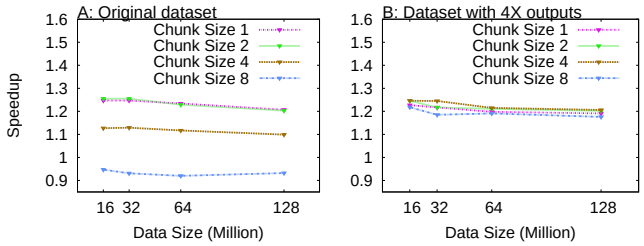

Figure 11: Speedup of direct output vs. double probe in the new hash join

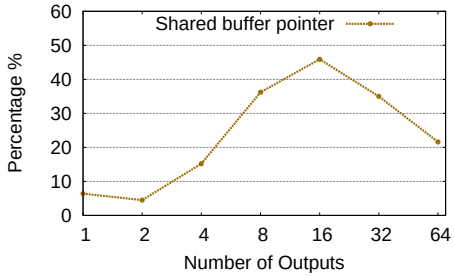

Figure 12: Performance gained by using shared memory buffer pointer vs. global buffer pointer

achieve the best performance with our original dataset. This is mainly because larger page size also requires larger overall buffer size since there may be empty holes in some of the pages. The time spent on transferring the output buffer back to main memory increases as the result of increasing buffer size. This could offset the benefit of reducing atomic operations. However, larger chunk size may help when the number of outputs per thread increases. Therefore we ran the test on a dataset of the same size as our original dataset but generates 4 times of the outputs, and the result is shown in Fig. 11B. As we see that the four different chunk sizes have similar performance at 64M and 128M, while the chunk size four stands out at smaller data sizes. Chunk size of eight is the worst case, indicating that there are still empty holes in it.

We also tested how the buffer chunk size affects the performance when the total number of threads decreases and work per thread increases. Since when outputs per thread increases, a larger page size helps reduce the number of requests to the global pointer. However, the results indicate that larger chunk size only brings marginal improvement. It is possible that the atomic operation in GPU is implemented very efficiently and the pointer is cached in L2, thus the atomic operation is not so sensitive to contentions.

Another way to reduce contention is to distribute the acquisitions of the shared pointer to thread block level. We divide the output buffer into small chunks so that each block can take one of them and outputs independently. The threads in the same block share a pointer in the shared memory that points to the next available slot in their own chunk. A thread acquires the pointer and increase it using atomic operation, then outputs to the available position. Larger selectivity benefits from this method, as shown in Fig. 12. Maximum improvement of 45.9% is achieved when the number of output is 16X. However, as the number of outputs continues to increase, the number of atomic operations on shared pointers also comes to a point where it begins to limit performance improvement.

4.2.6 Effects of Skewed Data. This section we present the performance of both the CPU and new GPU hash joins when the data has a skewed distribution (in the hashed domain). Specifically, we generate data that follow the Zipf distribution with different z factors. We run a version of our hash join without the dynamic parallelism (DP) code, and it obviously suffers from imbalance among the partitions under skewed data (Fig. 13). As the z-factor increases, data is more skewed and there is more performance degradation. Particularly, when the z-factor goes beyond 0.5, only a few blocks are kept busy processing the largest partitions while most of other blocks finish early. In the extreme case of $z = 1$, it causes a 4X slowdown as compared to the case of balanced data (i.e., $z = 0$). After applying DP to the code, threads can determine whether current partition is too large for their thread blocks to process, thus launch additional threads in a child kernel to work only on this partition. The total execution time does not change significantly as the z-factor increases. However, we do notice that there is a slight penalty when the z-factor reaches 0.75. This is mainly due to the overhead of launching new kernels. The CPU code is not affected much by data skew. In fact, the CPU code tackles this problem using a similar idea but in a slightly different way. It decomposes unexpectedly large partitions into smaller chunks. The small chunks are processed by using all the threads.

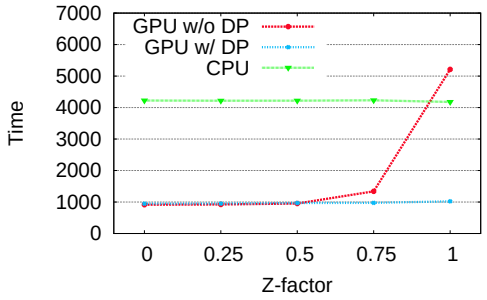

Figure 13: Performance of CPU/GPU code under different levels of data skewness. Here we only show results of Titan X and E5-2630v3

4.2.7 Joins under Large Data. Now we report the results of using new GPU join algorithms to handle large data that exceeds the capacity of GPU global memory. In such experiments, we keep the size of table R fixed (128M tuples for hash join and 256M tuples for sort-merge join) and vary the size of table S from 256M to 2.56 billion tuples. In order to process such a large table, we slice it into chunks and all of the chunks take turns to join with table R . It is worth mentioning that since the memory usage of hash join is higher than the sort-merge join, hash join can only handle a 128M-tuple chunk at a time while the sort-merge join takes a 256M-tuple chunk in each iteration. So for a given data size, the hash join have to go through more loops which impacts the overall performance.

Fig 14 shows the speedup of the Titan X over the E5-2630v3. The sort-merge join on GPUs is more capable of processing large data, resulting in speedup between 11X to 13X. Its speedup fluctuates but does not decrease as the size of table S increases. Since the

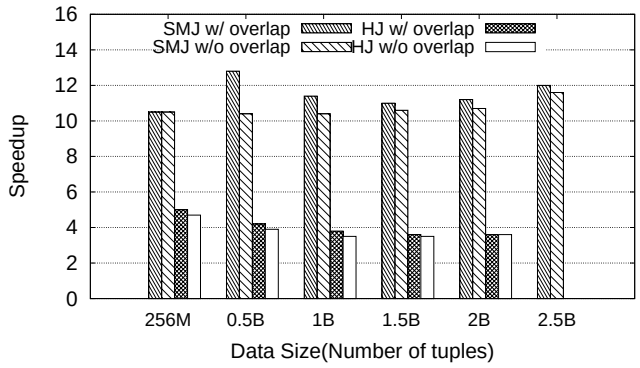

Figure 14: Speedup of Titan X over E5-2630v3 with large tables. For each join, we test the code with and without overlaps between data transmission and join processing

GPU sort-merge join algorithm needs fewer loops than the hash join, the running time grows in a nearly linear manner. This is the reason why it maintains the high speedup. The hash join on GPU achieves a 5.1X speedup under 256M tuples. However, it decreases as the table size increases and converges to around 3.5X. The kernel execution and data transfer overlapping (via multiple CUDA streams) is effective for both algorithms. However, the effects of such are less significant than we thought: on average, there is a performance gain of 8% and 6% for sort-merge join and hash join, respectively. By looking into the profiles of our code, we found that the main reason is that various kernel synchronization activities decrease the level of concurrency at runtime. Note that the CPU hash join code actually sets a limit on table size such that it cannot handle the case of 2.5B records in table *S*.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we propose new GPU-based hash join and sort-merge join algorithms. We take advantage of the various new features in the latest GPU hardware and CUDA software. On one hand, it achieves considerable performance boost over the existing state-of-the-art algorithm. The kernels have improved in many aspects including work efficiency and bandwidth utilization. On the other hand, experiments show that our optimized GPU code far outperforms the latest CPU hash join and sort-merge join code. This indicates that the GPU is a promising platform for join processing. Of course, the performance advantage of GPU is not only brought by raw computing power, but also by carefully designed algorithms towards the GPU hardware's features.

Future work can be conducted along a few directions. An immediate task is to extend our work to more scenarios of joins, such as joins of more than two tables, or two tables each with an arbitrarily large size. With the promise of many times of memory and communication bandwidth in the coming GPU architectures, it is necessary to test how that affects the performance of our GPU algorithms, or the design of such algorithms. We can also explore the application of GPUs in data stream systems where GPU's computing power can be fully utilized and the latency of data transfer can be amortized in concurrent queries.

ACKNOWLEDGMENT

The project described was supported by grants IIS-1253980 and CNS-1513126 from the National Science Foundation (NSF) of USA.

REFERENCES

[1] 2005. GPU Gems 2, Chapter 46. (Mar 2005). http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html.
[2] 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.
[3] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Procs. 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*. 94–103.
[4] 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.
[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. ..zsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. 362–373.
[6] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2004. Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations. In *Procs. of VLDB*. 1021–1032.
[7] 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.
[8] 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*. 37–48.
[9] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast Computation of Database Operations Using Graphics Processors. In *Procs. of SIGMOD*. 215–226.
[10] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Procs of ICS*. 331–340.
[11] 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.
[12] 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*. 511–524.
[13] 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.
[14] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Procs. DaMoN*. 55–62.
[15] 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.
[16] 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.
[17] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE TKDE* 14, 4 (Jul 2002), 709–730.
[18] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. 2012. Merge Path - Parallel Merging Made Simple. In *IPDPSW*. 1611–1618.
[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] Evangelia A. Sitaridi and Kenneth A. Ross. 2012. Ameliorating Memory Contention of OLAP Operators on GPU Processors. In *DaMoN*. 39–47.
[21] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2003. Hardware Acceleration for Spatial Selections and Joins. In *Procs. of ACM Intl. Conf. on Management of Data (SIGMOD)*. 455–466.
[22] Yi-Cheng Tu, Anand Kumar, Di Yu, Ran Rui, and Ryan Wheeler. 2013. Data Management Systems on GPUs: Promises and Challenges. In *SSDBM*. Article 33, 4 pages.
[23] 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*. Article 44, 11 pages.
[24] 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.
[25] Marco Zagha and Guy E. Blelloch. 1991. Radix Sort for Vector Multiprocessors. In *Procs. 1991 ACM/IEEE Conference on Supercomputing (SC '91)*. 712–721.