Relational Joins on GPUs: A Closer Look

Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, Hideyuki Kawashima

Abstract—The problem of scaling out relational join performance for large data sets in the database management system (DBMS) has been studied for years. Although in-memory DBMS engines can reduce load times by storing data in the main memory, join queries still remain computationally expensive. Modern graphics processing units (GPUs) provide massively parallel computing and may enhance the performance of such join queries; however, it is not clear yet in what condition relational joins perform well on GPUs. In this paper, we identify the performance characteristics of GPU computing for relational joins by implementing several well-known GPU-based join algorithms under various configurations. Experimental results indicate that the speedup ratio of GPU-based relational joins to CPU-based counterparts depends on the number of compute cores, the size of data sets, join conditions, and join algorithms. In the best case, the speedup ratios are up to 6.67 times for non-index joins, 9.41 times for sort index joins, and 2.55 times for hash joins. The execution time of GPU-based implementation for index joins, on the other hand, is only about 0.696 times less than the execution time of the CPU's counterparts.

Index Terms—Graphics processors, Query processing, Parallelism and concurrency

1 Introduction

A database management system (DBMS) is increasingly designed to process a large amount of data. Typical examples include Facebook's Presto, Google's Dremel, and Cloudera's Impala, in which several hundreds terabytes of data are generated on a daily basis. As the amount of data associated with web services and real-world applications is continuously growing while response times of data processing need to be maintained, DBMS engines are required to be more high-performance and scalable for large data sets.

"Relational Join" is one of the most frequently used and computationally-expensive queries in the DBMS. It requires a lot of computing and memory resources to execute its algorithm. Many DBMS engines use relational joins to integrate multiple pieces of data into a resultant data table. In particular, emerging data-oriented services and applications demand a large-scale data fusion mechanism based on relational joins to produce new valuable information. Scaling out relational join performance, therefore, is a key challenge for the database community.

Given that tuples in a relational DBMS are often independent, massively parallel computing is a reasonable and promising approach to high-performance DBMS engines. Using a large number of computing cores, algorithms of relational joins can be massively parallelized to accelerate their computation. In previous work, graphics processing units (GPUs) have been especially considered as such computing environments, demonstrating a significant improvement in performance for relational joins [1], [2], [3], [4], [5], [6]. GPUs integrate more than hundreds to thousands cores on a chip. Many high-performance computing applications now leverage GPUs to achieve an order-of-magnitude improvement in performance. It is natural that relational joins containing data parallelism in algorithms could also obtain a significant performance benefit from GPUs. The aforementioned previous work, however, largely focused on a particular standalone GPU, being evaluated with limited data sets. In general, performance of GPU computing can easily change depending on hardware architectures and data sets as well as implemented algorithms. It is not clear yet in what condition relational joins perform well on GPUs. To generalize the results from previous work, a closer look into relational join performance, considering multiple types of GPUs and multiple data sets, is needed.

Contribution: This paper presents the performance characteristics of GPU computing for relational joins under various configurations. Specifically, we quantify the performance of relational joins on GPUs with respect to the numbers of compute cores, the size of data sets, join conditions, and join algorithms. We also investigate why GPUs can speed up relational joins in terms of execution time and memory usage. In summary, this paper provides answers to the following questions, by consolidating the results from previous work.

- Can the performance of relational joins scale with advancement in GPU technology? To answer this question, we evaluate three GPUs: NVIDIA's GTX 560 Ti, GTX TITAN Black, and Tesla K20Xm.
- How does the performance of relational joins scale with the size of data sets? We evaluate multiple data sets ranging up to 128M×128M tuples to answer this questions.
- How is the performance of relational joins affected by data communication between the GPU and the host CPU? For this question, we evaluate the run-to-completion of workloads including CPU times and data transfer times rather than considering GPU times.
- How is the performance of relational joins affected by changing join algorithms? To answer this question, we evaluate four different join algorithms: non-index join(NIJ), index join(IJ), sort index join(SIJ), and hash join(HJ).

- *M. Yabuta, Anh Nguyen, S. Kato, M. Edahiro are with Graduate School of Information Science, Nagoya University.*
- *H. Kawashima is with Graduate School of System and Information Engineering, University of Tsukuba.*

- How is the performance of relational joins affected by loading data? To answer this question, we evaluate both in-memory and disk-based scenarios.

Organization: The remainder of this paper is organized as follows. In Section 2, we discuss related work on GPU-based query processing techniques. The system model we use in this paper is described in Section 3. Section 4 presents the design and implementation of relational joins on GPUs, and Section 5 provides experiments we perform to evaluate GPU-accelerated relational joins. In Section 6, we discuss several lessons we have learned from our experiments. This paper is concluded in Section 7.

2 Related Work

In-memory databases [7], [8], [9] and column-oriented databases [10], [11] are often used for fast query processing. However, time-consuming queries, such as "Relational Join" and "Group By", still prevent these databases from providing fast responses. General-purpose computing on GPUs (GPGPU) is becoming an alternative solution to accelerating such time-consuming queries [1], [2], [3], [4], [5].

This paper explores the performance of relational joins executed on GPUs. While we reference previous work [2] for GPU-based implementation of join algorithms, this paper provides a different viewpoint of performance analysis. The previous work focused on increasing the computation speed of those algorithms using a particular GPU, and investigated how the overall execution time changes upon the match rate of join algorithms and the domination of data transmission times between the CPU and the GPU. On the other hand, this paper focuses on the computational scalability of those algorithms using multiple GPUs and multiple data sets. As a result, we clarify in what condition relational joins perform well on GPUs. We also demonstrate that the join algorithms presented in the previous work [2] can be used with multiple generations of GPUs, although their performance depends highly on the number of compute cores, the size of shared memory, and the size of data tables. The impact of GPU architectures and data sets on relational join performance is discussed in Section 5.

Join algorithms can be further accelerated on GPUs by leveraging unified virtual addressing (UVA) [6]. UVA allows the GPU to directly access the main memory on the host computer. Kaldewey et al. [6] reduced data transfer times in join algorithms using the characteristics that the data access time becomes aligned with the UVA read time if data access to the device memory is random or non-coalesced. The data transfer times between the CPU and the GPU can be also reduced by coupling the CPU and the GPU via coprocessors [12], and He et al. [13] has presented a similar approach. This approach integrates queries on the GPU and the CPU. However, previous studies aimed to reduce transmission times of data copies between the GPU and the CPU mainly, whereas, in many cases, data sets are stored in secondary storage devices, such as hard drives and solid state drives. Hence, we set up an experiment to find out how reading data from secondary storage devices affects the overall performance of GPU-based join algorithms. We also investigate how the ratio of the data transmission times between the GPU memory and the host memory to the execution time of the whole join process changes under various conditions.

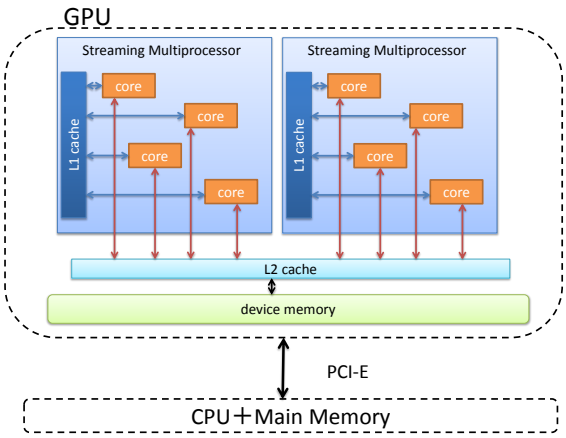


Fig. 1. General structure of GPU computing.

3 System Model

GPUs provide massively parallel computing capabilities with thousands of compute cores integrated on a single chip [14], [15]. The concept of general-purpose computing on GPUs (GPGPU) [16] has received significant attention due to the emergence of programming languages, such as CUDA [17], [18] and OpenCL. The general structure of GPU computing is shown in Figure 1. GPUs typically provide four instances of memory including device memory, two types of caches (L1 and L2), and registers. The device memory contains large memory space that is accessible to and from all compute cores. The L1 cache represents on-chip memory for each streaming multiprocessor (SM), and the L2 cache is shared cache for the device memory. Finally, the registers are extracted on the fast but small memory space of each SM so that compute cores can allocate local variables to them. In CUDA, these memory spaces are abstracted such that the device memory is used for global memory, constant memory, and texture memory. The system uses the L1 cache for a secondary storage of local variables and shared memory, and the L2 cache is used for shared global memory cache.

CUDA is an abstraction of compute unified device architecture, including a programming language, compiler, and runtime stack designed for massively parallel computing [18]. The primary advantages of CUDA are that a large number of compute threads can be executed simultaneously, and heterogeneous memory management is supported. Since the GPU often operates as a coprocessor in conjunction with the CPU, conventionally a set of the CPU and the main memory is called a host system, and the GPU (including the device memory) is called a device system. A program running on the CPU is called a host program, and a component of that program running on the GPU is called a kernel. The basic approach to GPU programming is that (i) the host program transfers the kernel code and its data to the global memory, (ii) function calls on the host program launch the kernel on the GPU, and (iii) the result

TABLE 1
Definition of symbols related to the relational join query.

R and S:	Tables on which the relational join query is executed. In the case of index join, R serves as the index table. In hash join, R is used as the criteria for table partitioning.
$\|R\|$ and $\|S\|$:	Counts of table rows of R and S, respectively.
ResultTable	Table that stores the results of the relational join query.
K and M:	K stands for kilo (1024) and M stands for mega (1024×1024). We use K and M as units of the table size, since the power of two is more convenient for GPU computing.

of computation obtained on the GPU is copied back to the main memory from the global device memory.

In order for a large number of compute threads to run simultaneously, CUDA employs the concepts of blocks and grids. A block is a set of threads, and a grid is a set of blocks. In current CUDA versions, each block can contain at most 1024 threads. The number of threads per block is referred to as the block size. Each block is represented by a three dimensional structure of {x, y, z} threads, where $x \leq 1024$, $y \leq 1024$, and $z \leq 64$. Each grid is also represented by {x, y, z} blocks, where $x \leq 2\hat{}32 - 1$, $y \leq 65535$, and $z \leq 65535$. Threads in the same block can use the same shared memory as fast memory. The shared memory size is 48 KB at maximum, which is allocated from L1 cache as shown in Figure 1. The system allocates computing cores by units of blocks and SMs. In other words, threads within the same block are assigned to computing cores of the same SM. Each SM can contain multiple blocks. The number of blocks is determined by the resource requirements, such as the number of threads and the shared memory size.

Coalesced memory access is a principal method of GPU programming that supports fast memory access. In this method, 32 threads are packed by hardware as a unit of dispatching on the GPU, which is often referred to as a warp. When threads within the same warp access sequential memory addresses, the entire memory accessing can be aggregated by one or a more instruction. Coalesced memory access helps the GPU maximize performance, as well as the allocation of computing cores and shared memory

4 Relational Joins on GPUs

Relational joins refer to database processing that links tuples from tables R and S based on a given condition. Table 1 defines the symbols related to the relational join query.

In this section, we assume that data tables are already loaded in the main memory, and four well-known join algorithms are considered: NIJ, IJ, SIJ and HJ.

NIJ compares all rows of the tables in an exhaustive fashion. This is easy-to-implement and conditionally flexible, but data processing is slow.

To reduce search time, IJ and SIJ use a search method where one table contains the index and the other contains the searched values. Creating an index is performed by two methods, i.e., (i) partitioning the tables and (ii) sorting the tables. IJ sorts the tables into partitions and stores the starting position of each tuple group.
In this paper, the former is referred to as IJ and the latter as SIJ. To search the index, the algorithm identifies the starting position based on the matching value, followed by a sequential search. The tables are divided into partitions, each having the same remainder of division performed by some appropriate constant, *i.e.*, the hash value. SIJ sorts the table and performs a binary search. Although these two algorithms require indices to be created before the execution of join, our experiments were performed under the assumption that the indices were created in advance.

HJ is implemented by referencing a partition hash join [19], [8], [20]. HJ has two phases: (i) partitioning and (ii) join. In the partitioning phase, tuples in both tables are divided by some constant and sorted by the remainder of division (hash value). In the join phase, the algorithm executes NIJ across partitions with the same hash value.
Note that HJ is faster than NIJ; however, it is slower than IJ because HJ must perform the hash function. HJ is also restricted relative to join conditions. For example, an inequality sign cannot be used in HJ conditional expressions.

We accelerate these four joins using the GPU. The following subsections describe the details of implementation using CUDA, and most of which are based on previous work [2].

4.1 Counting Sort

A major issue of GPU-accelerated join is confliction among threads when they attempt to write their results to the same location in the memory. To avoid this problem, we implement Algorithm 1 using the existing primitives [2].

Algorithm 1 Counting Sort

1: Input tuples are split and assigned to threads. Each thread counts the number of matched tuples and writes results to an array named $count$.
2: The prefix sum is performed on the $count$ array to calculate the total number of matched tuples and positions for writing matched tuples in each thread.
3: Each thread writes matched tuples into the positions determined in step 2.

We use the CUDA thrust prefix sum program [21] to perform the prefix sum in Step 2 of Algorithm 1.

By computing the write location for each thread, the Counting Sort algorithm prevents threads from accessing the write locations of other threads, thereby solving the write conflicts among threads. In addition, the total number of matched tuples calculated by the prefix sum enables the GPU to allocate an appropriate memory buffer for the result table.

4.2 NIJ

As shown in Fig. 2, NIJ divides tables R and S into subtables R_n and S_m, and allocate them to blocks so that it can

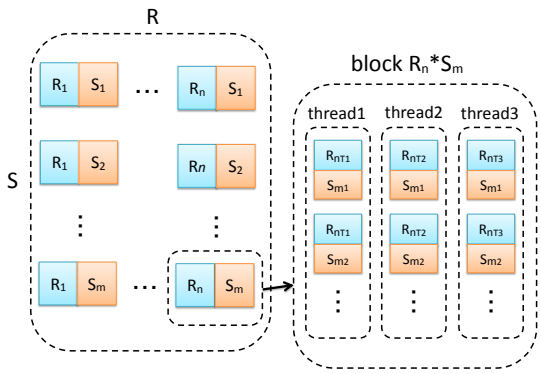


Fig. 2. Structure of NIJ.

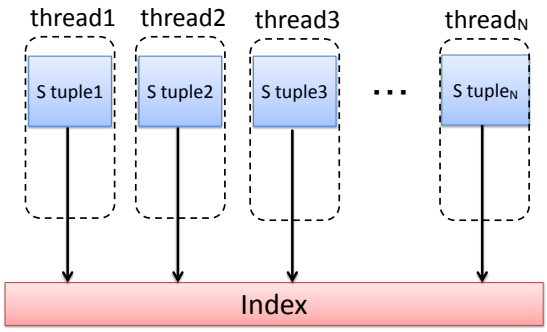


Fig. 3. Structures of IJ and SIJ

compare all the combinations of these sub-tables. In each block, each thread compares every tuple of sub-table R_n with all tuples of sub-table S_m. To improve performance, we store S_m in the shared memory, since it is accessible to all threads within the block. Therefore, we must partition tables such that the size of sub-table S_m does not exceed the shared memory capacity. In our implementation of NIJ, we first execute the Counting Sort algorithm to determine the size of the result table and the writing location for each thread. In the next phase, we divide tables again using the same method, and this time, each thread writes the result of join to the result table according to the write location that is computed in advance by the Counting Sort algorithm.

4.3 IJ

As shown in Fig. 3, our implementation of IJ creates $|S|$ threads to perform the Counting Sort algorithm. Similar to NIJ, after calculating the write location for each thread, we write the result of join to the result table according to the calculated locations. Each thread in turn reads a tuple from S and compares that tuple to the index to determine whether the tuples' indices match. If the size of $|S|$ exceeds the upper bound on the number of threads, the algorithm reads the second and third tuples into the same thread.

4.4 SIJ

Similar to IJ, SIJ creates $|S|$ threads to perform the Counting Sort algorithm. The procedure of writing the results of join to the result table is identical to the one in NIJ: namely, each thread writes the result to the location determined by the Counting Sort algorithm. However, each thread searches for tuples of a non-indexed table in the index table using binary search. Since the device memory of GPU is accessed randomly during binary search, to improve the memory access speed, we copy such data that are present at the beginning search positions of the index array to the shared memory. We found experimentally that 4096 bytes per blosk is an appropriate size for the shared memory to achieve the best performance. In our experiments, this size of shared memory is used for evaluation.

4.5 HJ

HJ contains two phases in the algorithm, partitioning and join. The partitioning phase has two steps, i.e., (i) recursive execution of Split [2], [22] (Algorithm 2) to sort tuples by partitions and (ii) calculation of the starting position of each partition based on the sorted table.

Algorithm 2 Split

1: Read each tuple and calculate the number of tuples that belong to each partition.
2: Scan the number of tuples in each partition to calculate the starting position of each partition.
3: Re-read each tuple to identify the partition and starting position based on the tuple value, then write to that partition.

We use Algorithm 2 to partition tables based on hash values. On the GPU, the algorithm first divides input tables into multiple partitions. Each thread then calculates the number of tuples that correspond to each partition, as shown in Fig. 4. The algorithm creates a histogram representing the numbers counted by each thread per partition. It next scans the histogram to identify where each partition starts writing tuples. Finally, the algorithm re-partitions the table into threads, and identifies the partition and position set based on the tuple value to sort the data. Our $Split$ implementation uses shared memory to improve access speed when creating a histogram. Since the maximum size of shared memory is 48 KB per block, the number of partitions is limited to $48KB/(sizeof(int) \times BlockSize)$.

Here, an issue of concern is that the number of partitions created by the above method is insufficient. We extend the Split algorithm to execute recursively by shifting the partitioning bits to increase the number of partitions per execution to the power of N. For example, if the number of partitions per execution is 64, we use bits 0 to 5 for the first execution and bits 6 to 11 for the second execution as the base of expression, representing the partitioning condition. Writing to the same partition is performed in the order of thread numbers, as shown in Fig. 4. The $Split$ algorithm divides the table into 64 partitions in the first execution. Tuples assigned to the same partition in the second execution are written in the order of partition numbers given by the first execution.

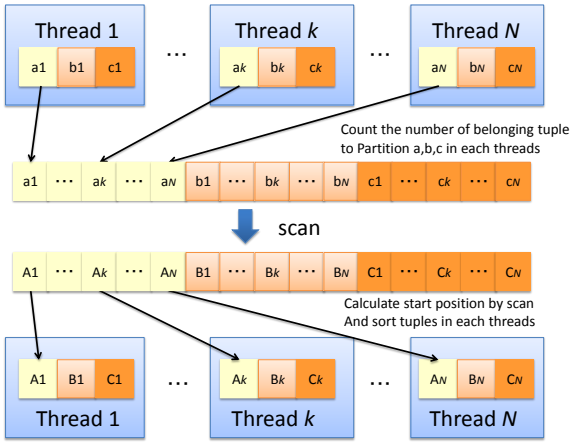


Fig. 4. Structure of Split.

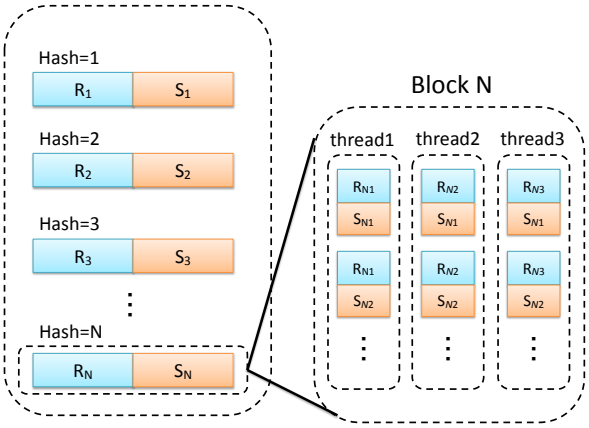


Fig. 5. Structure of the join phase of HJ.

Therefore, after the second execution of the $Split$ algorithm, tables are sorted automatically with up to the 64^2 partitions. After the last step, the algorithm counts and scans the final number of tuples in each partition to calculate the starting position of each partition. Hereafter, we refer to the number of partitions resulting from a single execution of the $Split$ algorithm as the number of splits, and the overall number of partitions as the number of partitions.

In the join phase, the $Split$ algorithm partitions the table into blocks with the same hash value, as shown in Fig. 5. For example, R_N and S_N are present in the same block since both belong to partition N. Similar to NIJ, HJ places S_N in the shared memory and reads the tuple from R_N on a thread-by-thread basis within a block. It reads tuples R_{N1}, R_{N2}, and R_{N3} from R_N into each thread and compares them with tuples S_{N1}, S_{N2}, and S_{N3} from S_N. For both CPU-based and GPU-based implementations of join algorithms, we find and use the number of partitions such that the executions of join algorithms demonstrated the fastest performance in preliminary experiments conducted beforehand. After partitioning S, if size of each partition of S exceeds the shared memory size, the system re-partitions S to fit the shared memory size. During the join phase, we calculate the size of the result table and the write location using the Counting Sort algorithm, before each thread writes the result of join to the output buffer.

4.6 Expansion for Large Tables

The size limit of the input table used in this implementation is 1M for NIJ and 128M for other algorithms. When input tables are larger than this limitation, we divide them into multiple smaller sub-tables. Thus, the GPU can perform the join algorithms on the sub-tables. Each pair of sub-tables is joined before the final result is summarized and written to the output buffer. The above method is only applied for NI and HJ, due to the complexity of partitioning indices in IJ and SIJ.

5 Evaluation

We evaluated the performance of GPU-accelerated join in terms of the number of computing cores, the size of data sets, and different types of algorithms (i.e.,NIJ, IJ, SIJ, and HJ). The objective of this evaluation was to measure end-to-end performance of GPU-accelerated join. We revisit previous work [2], [6] with state-of-the-art GPUs, and enhance their results by utilizing multiple generations of GPUs and multiple types of data sets.

TABLE 2
CPU and GPU Specification

	CPU	GPU
clock frequency	3.30GHz	732MHz
core	4	2688
memory	64GB	6GB

5.1 Experimental Setup

Our experiments were performed on a system with an Intel (R) Xeon (R) E5–2643 CPU and an NVIDIA Tesla K20Xm GPU(specifications are shown in Table 2). In addition, we implemented CPU-oriented join algorithms in two different ways,i.e., single-threaded and multi-threaded. The multi-threaded implementation for CPU HJ was implemented in the same manner as previous work [19], [20].

Each tuple of the experimental tables was constructed from two 4-byte integer variables, i.e., key and val. In the basic configuration, the size of tables R and S was the same. The join condition was defined by $R : val = S : val$. In addition, to determine the impact of the rate of the matched tuples to overall performance, we defined a "match rate" parameter, which is calculated as follows: match rate $=$ number of matched tuples in S/$|S|$.

The value of val in each tuple is unique in the scope of each table. By assigning a unique value to each val, the size of $ResultTable$, which is described in Table 1, can be predicted to some extent. First, unique values were randomly generated in R. In S, to meet the match rate, val's value in a specific number of S's tuples was set to be the same as that of R's tuples, while the rest were assigned with randomly selected values that do not overlap the values of R's tuples. After being generated, S's tuples are randomly rearranged to ensure that they were not partially sorted.

We investigated the performance of GPU-accelerated join algorithms and compared to them with each other, as

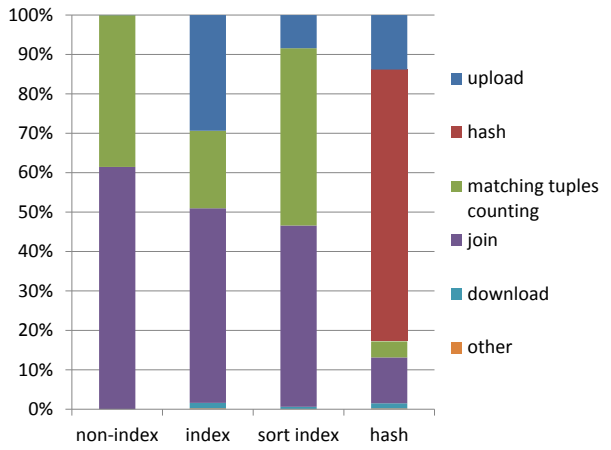


Fig. 6. Breakdown of the execution time of GPU implementation.

well as to CPU-oriented counterparts. We also considered the impact of table size, match rate, join condition, and GPU type on the overall performance of the join algorithms. In addition, we performed our experiments under different conditions, such as when the data set is initially stored in an external storage device rather than main memory and when the join condition is more complicated. The results from those experiments help us figure out how the performance of the join algorithms varies under such conditions. In particular, the NIJ algorithm was tested in the case when the size of the input tables was extremely large, because this algorithm is the most time-consuming. In addition, we address the best split number and partition number for partitioning in HJ. Finally, we evaluated the performance of improved memory access.

5.2 Comparison of Algorithms

Each GPU-accelerated join algorithm was compared to the corresponding CPU multi-threaded join algorithm in terms of execution time. The number of table tuples was 1M×1M for NIJ, and 128M×128M for the other algorithms. This is because GPU memory cannot accommodate memory space allocated by NIJ with tables larger than 1M×1M. The size of R_n was set to 1024, and the size of S_m was set to 4096. The block size was 1024 in IJ, 256 in SIJ, and 16 in HJ. Note that, in HJ, the number of tuples processed by the Split algorithm per thread, the number of splits, and the number of partitions were set to 256, 64, and 2M, respectively.

TABLE 3
Comparison of execution times.

	GPU (ms)	Multi-threaded CPU (ms)	Speedup
NIJ	24863	165843.6	6.67
IJ	1986	1382	0.696
SIJ	6083	57242.2	9.41
HJ	2446	6233.2	2.55

Table 3 shows the execution time of each join algorithm. Compared to the multi-threaded CPU implementation, the GPU implementation is 6.67, 9.41, and 2.55 times faster for NIJ, SIJ, and HJ, respectively.

Note that IJ works better on the CPU than on the GPU because IJ must transfer significant amount of data between host memory and device memory in the GPU implementation. The base of the index also affects performance. Increasing the base causes the data transmission time to increase. As a result, the GPU implementation is forced to use a small base, which allows the CPU implementation to perform better.

The breakdown of execution time of the GPU implementation for the join operation is shown in Fig.6. According to these results, counting the matching tuples and joining tuples are the most time-consuming operations for NIJ and SIJ. In particular, the index search requires significant time with SIJ because random accesses to global memory are performed frequently. In HJ, most of the operation time is spent on hashing because this operation includes many random memory accesses. Consequently, the overhead of hashing in HJ is significant. The rates of execution time for uploading, counting matched tuples, and joining in IJ are approximately the same. Compared to SIJ, the number of global memory accesses in IJ is relatively small; thus, the executing kernel in IJ is faster than in SIJ. However, IJ requires a considerable time to upload a large index. The result is the domination of the uploading rate of in the overall execution time with IJ.

5.3 Impact of Table Size

We investigated how the size of the input tables affects the execution time of join operations. The match rate is fixed at 10%. The experimental result of are shown in Figs. 7, 8, 9, and 10.

The overhead imposed on algorithms by data communication has been studied previously [6], [13]. To highlight the impact of data communication to overall performance of join algorithms, we added a configuration called "GPU w/o com", which is execution time excluding data communication for GPU join algorithms.

Because the most optimal values of block size, grid size, and shared memory size for each algorithm fluctuate according to table size, we used the most suitable values for those parameters which were determined experimentally. The parameters in the following sections were also set by those values.

Figs. 7, 8, 9, and 10 show change in execution time as a function of table size. It is evident that each algorithm exhibits virtually linear execution times.

As can be seen in to Fig. 7, NIJ resulted in the worst performance when performing joins on 1M×1M tuples. This may be due to the increase in the amount of shared memory consumed per block. The experimental results show that when NIJ was performed on tables with size less than or equal to 512K tuples, 1024 tuples were stored in shared memory, and the join operation achieved the best performance. When the size of the input tables reached 1M tuples, due to the limited number of elements that can be scanned, approximately 4096 tuples were stored in shared memory. Since the size of each tuple was 8 bytes, the total amount of shared memory consumed in each block increased from 8 KB with 512K-tuple tables to 32 KB with 1M-tuple tables. Because shared memory is integrated in each SM, when

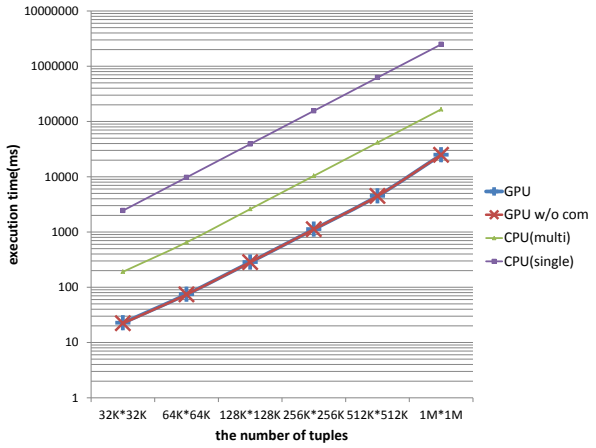


Fig. 7. Execution time for NIJ as a function of table size.

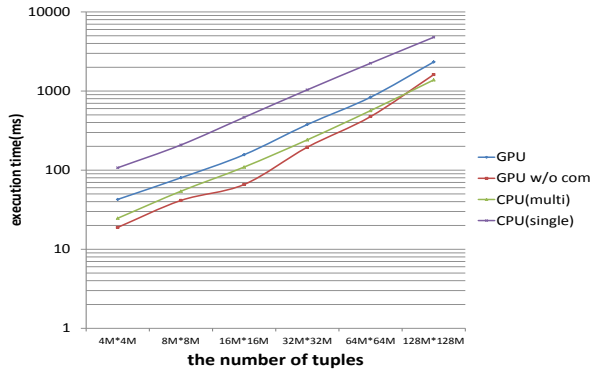


Fig. 8. Execution time for IJ as a function of table size.

the memory space required for each shared memory area increases and the number of blocks decreases, the number of parallel threads decreases. Consequently, the acceleration rate of NIJ in the case of joining 1M-tuple tables became slightly less than when joining smaller size tables.

The acceleration rates for IJ and SIJ dropped after peaking with 16M×16M, as shown in Figs. 8 and 9. This was largely due to the fact that the level of parallelism was too high for the algorithms to perform parallel execution. In contrast, for HJ, the acceleration rate increased as the table

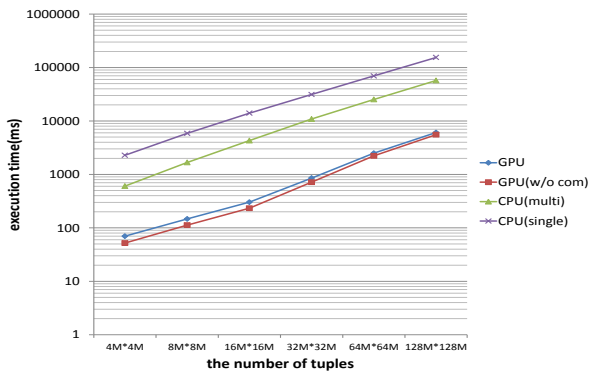


Fig. 9. Execution time for SIJ as a function of table size.

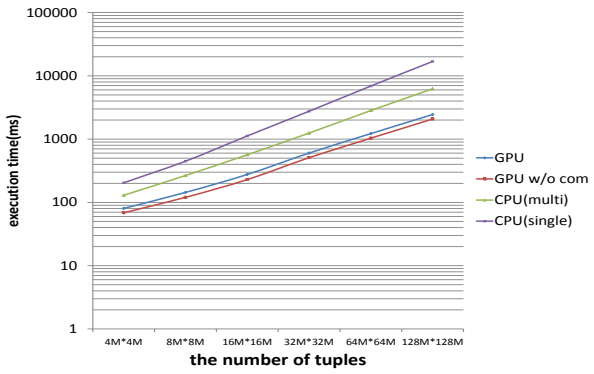


Fig. 10. Execution time for HJ as a function of table size.

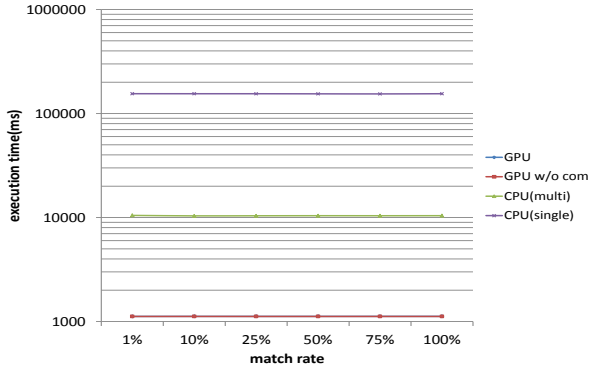


Fig. 11. Execution time for NIJ as a function of the match rate.

size increased, as shown in Fig. 10. Although hashing occupies the most execution time for HJ, the overall acceleration rate increased due to the high acceleration rate of the split operation.

5.4 Impact of Match Rate

In this experiment, we considered the impact of match rate on the overall acceleration rate of join algorithms implemented with the GPU. The table size was fixed to 256K tuples for NIJ and 16M tuples for the other algorithms. Figs. 11, 12, 13 and 14 show the changes in execution time as a function of the match rate. When the match rate changed from 1% to 100%, the execution time increased approximately 76% for IJ, 41% for SIJ, and 16% for HJ. However, the change in NIJ was negligible.

Fig. 15 shows the breakdown of execution time when the match rate varied from 1% to 100%. Because NIJ verifies the join condition in all pairs of tuples, but only combines pairs that meet the condition, combining pairs only occupies a small portion of the overall execution time. Thus, although the match rate of NIJ changed, it only affects the execution time of combining matched tuples, which is relatively small compared to other operations. Therefore, the overall acceleration rate of NIJ, did not significantly change when the match rate changed.

The impact of the match rate on the execution time of IJ was the most significant among all join algorithms.

In IJ and SIJ, instead of evaluating join predicate on every pair of rows from tables, the system first performs searching

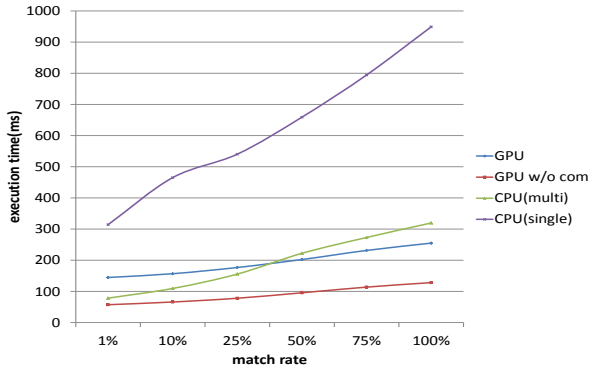


Fig. 12. Execution time for IJ as a function of the match rate.

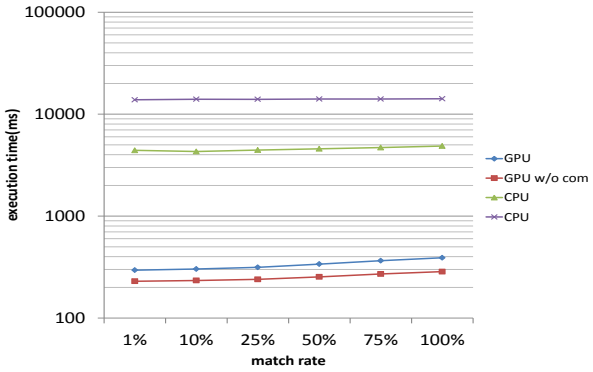


Fig. 13. Execution time for SIJ as a function of the match rate.

on the index values of the index table. Since index values are sorted, we can use some advanced searching methods like binary search to look for the appropriate tuples. Those methods require less number of comparison than traditional sequential search. As the result, searching in IJ and SIJ is much faster than in NIJ. In contrast, the acceleration rate of SIJ decreased when the match rate increased. In SIJ, the operation of searching for an index ends as soon as matched tuples are found. Note that as the match rate increases, the fewer searches are performed. However, since all threads in a warp are synchronized in GPU computation, eventually the deepest search dominates performance. This leads to a low acceleration rate when the match rate is high. Finally,

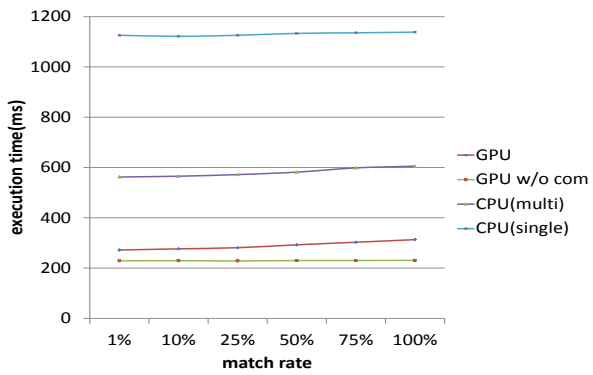


Fig. 14. Execution time for HJ as a function of the match rate.

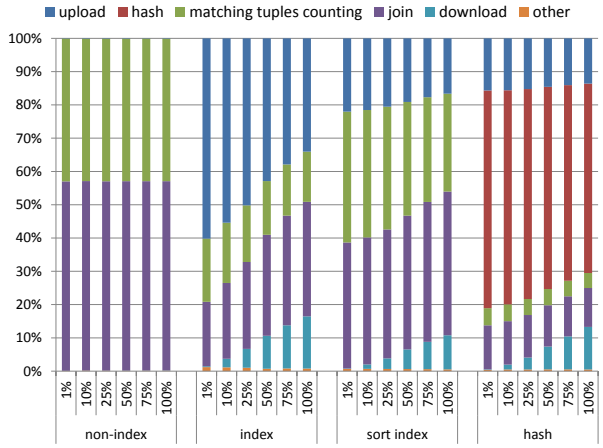


Fig. 15. Breakdown of the execution time when the match rate is 1% and 100%.

in HJ, the acceleration rate dropped slightly due to the overhead incurred by writing data on the GPU.

5.5 NIJ and HJ with Large Tables

We also evaluated NIJ and HJ when the size of the input tables exceeded the amount of global GPU memory. We changed the size of the input tables to 128K, 256K, 512K, 1M, 2M, 4M, 8M, and 16M for NIJ, and to 6M, 32M, 64M, 128M, 256M, 512M and 1G for HJ.

Table 4 shows the results of the NIJ evaluation when the tables were divided into 1M sub tables. The system achieved the best performance when the size of each sub-table was 512K for NIJ and 128M for HJ. The reason for this improvement in NIJ is that when the size of each sub-table was 1M, the number of tuples each thread had to process and the number of threads per block both increased, compared to the case of 512K sub-tables. Thus, the size of the writing positions array computed by the Counting Sort algorithm exceeded the size of global GPU memory; consequently, the number of tuples each thread had to process increased. As the amount computation increases the degree of parallelism decreases, which results in decreased of overall performance.

TABLE 4
Execution time when Partitioning a Table of 1M Tuples.

Number of table partition	1M	512K*2	256K*4	128K*8
Execution time (ms)	24863	22171	22177	22239

Figure 16 and 17 show the results of experiments with large tables with NIJ and HJ, respectively. Since large tables are divided into sub-tables and the sub-tables are joined, the join process may become slower than the case of one-time processing. In HJ, when the input table size exceeded 128M, the growth of processing time increased more quickly than predicted. However, in NIJ, the growth of processing time did not change, even when the size of the input tables was larger than 1M. The reason for this is that when the number of threads becomes very high, all blocks must wait in a specific order rather than executing simultaneously. Note

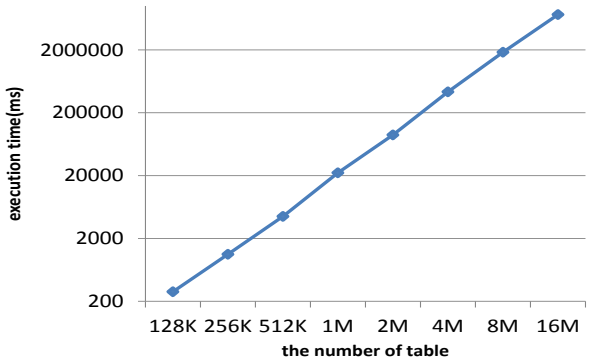


Fig. 16. Result of joining large size table in NIJ

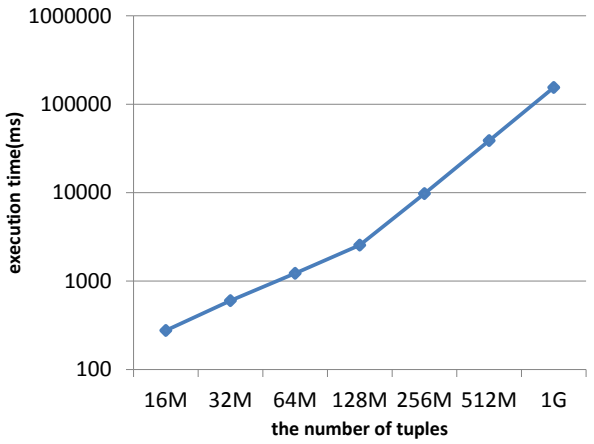


Fig. 17. Result of joining large size table in HJ

that this mechanism does not affect the growth of processing time when sub-tables are joined in NIJ.

5.6 Disk Overhead

To this point, we have assumed that data tables are present in memory. Here, we consider the condition where data sets are stored on disk. In this experiment, data were initially loaded from disk to memory. Then, the join operation was executed in the same manner as the previous case where data tables were present in memory. For IJ and SIJ, measurements were made with the index present in memory. The table size was set to 1M for NIJ and 128M for the other algorithms. The match rate was set to 10%. The other parameters were set to the values described in Section 5.2.

Table 5 shows the execution time and speedup achieved by GPU and CPU multi-threading, with the percentage of disk read time for GPU-accelerated join. Although the change for NIJ was negligible due to the small table size and the significant join execution time, the other algorithms required approximately 505 ms to read the tables, which is a 6-35% increase in execution time.

5.7 Comparison of GPUs

All previous experiments were conducted using a K20Xm GPU. Here, we use a GTX 560Ti and GTX TITAN Black to compare join performance on diffrent GPUs. The results from a high-end CPU are also provided as a reference. Table 6 lists the specifications of these two GPUs. Since the GTX 560Ti does not have sufficient memory space to perform a one-time join for 1M tables, the input tables were divided into 256K sub-tables.

TABLE 5
Percentage of Disk Read Overhead for each Algorithm.

	GPU join (ms)	CPU join (ms)	Speedup	percentages
non-index	24877	165977	6.67	0%
index	2870	1892	0.659	35%
sort index	6644	65501	9.86	8%
hash	2988	6920	2.32	18%

TABLE 6
GPU Speciications.

type	Clock frequency	Cores	Memory
560Ti	1645 MHz	384	1 GB
TITAN B	980 MHz	2880	6 GB

The size of the input tables was set to 1M in NIJ and 16M in other cases, and the match rate was maintained at 10%. The experimental results are shown in Table7.

The GTX TITAN Black showed the best performance. However, while the performance of NIJ and HJ on the TITAN Black were 13% and 30% better than the K20m, respectively, those for IJ and SIJ were nearly the same between the two GPUs. Since IJ and SIJ cannot leverage coalesced GPU memory access, the speed of reading data from memory is significantly slow. As a result, the join processing time cannot be reduced even with the large number of cores on the GTX TITAN Black. On the other hand, for HJ, as the amount of computation in the hash process was significant, the GTX TITAN Black showed high performance. In addition, the proportion of data transmission in IJ was 30% of the total processing. The data transmission time was approximately the same among all GPUs. Taking the percentage of data transmission time into account, the difference in the overall join processing time was not very significant among the tested GPUs.

6 Discussion

In this section, we discuss major factors that appear to be bottlenecks for performance speedup in GPU-accelerated joins, i.e., data transmission and memory capacity.

First, impact of data transmission on overall processing time is non-trivial. The proportions of time for data transmission to the whole join process were 30%, 8%, and 13% for IJ, SIJ, and HJ, respectively, according to our experimental results. This is relatively high. The development of GPU technology would continue with more SM cores and more memory space. However, as mentioned in Section 5.7, an improvement in GPU computing capabilities does not contribute to faster data transmission. It is a core challenge for GPU-accelerated joins to develop a novel way of reducing data transmission time.

Secondly, the size of global memory and shared memory is important. As explained in Section 5.5, for HJ, when the size of the input tables exceeds the size of global memory,

TABLE 7
Execution time and acceleration rate of GPU-accelerated Join relative to CPU-oriented Join.

	CPU (single)	CPU (multi)	K20Xm	560Ti	TITAN Black
NIJ (ms)	2486105	165843	24863	64980	21648
IJ (ms)	465	109	157	261	153
SIJ (ms)	14013	4300	303	730	295
HJ (ms)	1122	565	276	847	198

the processing time increases because the algorithm must divide tables into multiple sub-tables. Regarding the size of shared memory, on the other hand, since the current size of shared memory is 48 KB, each SM can only execute a small number of blocks. This causes the overall degree of parallelism to be extremely limited. To improve join performance, it is desired to increase parallelism by integrating more shared memory space to each SM.

Another problem not fully investigated in this paper is the memory capacity of registers. Our evaluation used synthetic data sets rather than real-world data sets. Since real data would require much more complex processing, more local memory may be needed. Each SM has 65,536 registers at the moment, each of which is represented by 4 bytes. Considering that each SM can occupy 1024 threads, each thread can eventually use 256 bytes of local memory at most. Complex database processing may require more space than this amount of local memory. We conjecture that 256 bytes of local memory may be insufficient, and therefore reducing local memory consumption in join algorithms would be a key to apply GPU technology for future DBMS engines.

7 Conclusion

We have presented the performance characteristics of relation joins on GPUs. Whereas previous work demonstrated non-trivial performance improvements using a particular GPU, this paper assessed improvements across multiple GPUs with various setup changes, such as the number of compute threads, size of the data sets, join conditions, and join algorithms. Our results indicate that NIJ and SIJ are scalable relative to the number of compute cores, whereas the performance of IJ and HJ might not improve with GPUs because IJ and HJ incur data transfer and partitioning overhead.

The achievable speedup of relational joins obtained by using GPUs varied relative to the number of compute cores, the size of data sets, join conditions, and join algorithms. According to our results, the best scenario exhibited speedup of 6.67 times for NIJ, 0.696 times for IJ, 9.41 times for SIJ, and 2.55 times for HJ under the condition of equal joins.

Acknowledgement

This work was partially supported by CREST, JST and JSPS KAKENHI Grant Number 25280043.

REFERENCES

[1] R. Fang, B. He, M. Lu, K. Yang, N. Govindaraju, Q. Luo, and P. Sander, "GPUQP: Query Co-processing Using Graphics Processors," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, 2007, pp. 1061–1063.

[2] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 511–524.

[3] M. Lieberman, J. Sankaranarayanan, and H. Samet, "A Fast Similarity Join Algorithm Using Graphics Processing Units," in *Proceedings of the 24th IEEE International Conference on Data Engineering*, 2008, pp. 1111–1120.

[4] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010, pp. 351–362.

[5] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010, pp. 339–350.

[6] T. Kaldewey, G. Lohman, R. Muller, and P. Volk, "GPU Join Processing Revisited," in *Proceedings of the 8th International Workshop on Data Management on New Hardware*, 2012, pp. 55–62.

[7] H. Molina and K. Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, pp. 509–516, 1992.

[8] P. Boncz, S. Manegold, and M. Kersten, "Database Architecture Optimized for the new Bottleneck: Memory Access," in *Proceedings of the 25th ACM International Conference on Very Large Data Bases*, 1999, pp. 54–65.

[9] M. Stonebraker and A. Weisberg, "The VOLTDB Main Memory DBMS," *IEEE Data Engineering Bulletin*, vol. 36, 2013.

[10] D. J.Abadi, P. A.Boncz, and S. Harizopoulos, "Column-oriented Database Systems," *Proceedings of VLDB Endowment*, pp. 1664–1665, 2009.

[11] P. Hasso, "A Common Database Approach for OLTP and OLAP Using an In-memory Column Database," *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, 2009.

[12] J. He, M. Lu, and B. He, "Revisited Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture," *Proceedings of VLDB Endowment*, vol. 6, pp. 889–900, 2013.

[13] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander, "Relational Query Co-Processing on Graphics Processors," *ACM Transactions on Database Systems*, vol. 34, pp. 21:1–21:39, 2009.

[14] V. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the 37th ACM Annual International Symposium on Computer Architecture*, 2010, pp. 451–460.

[15] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proceedings of the 36th ACM Annual International Symposium on Computer Architecture*, 2009, pp. 152–163.

[16] D. Luebke, M. Harris, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: General-purpose Computation on Graphics Hardware," in *ACM SIGGRAPH 2004 Course Notes*, 2004.

[17] Parallel Programming and Computing Platform | CUDA | NVIDIA. NVIDIA. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[18] N. Wilt, *THE CUDA HANDBOOK A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.

[19] C. Kim, T. Kaldewey, V. Lee, E. Sedlar, A. Nguyen, N. Satish, J. Chhugani, A. Blas, and P. Dubey, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs," *Proceedings of VLDB Endowment*, vol. 2, pp. 1378–1389, 2009.

[20] S. Blanas, Y. Li, and J. Patel, "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 37–48.

[21] NVIDIA ACCELERATED COMPUTING. NVIDIA. [Online]. Available: https://developer.nvidia.com/accelerated-computing

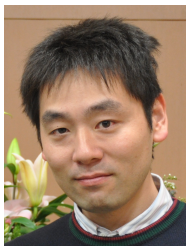
[22] S. Patidar and P. Narayanan, "Scalable Split and Gather Primitive for the GPU," Tech. Rep., 2009.

Makoto Yabuta is a Ph.D. student at the School of Information Science, Nagoya University. He received his B.E. degree from Nagoya University in 2013. His research interests include parallel computing and autonomous vehicles.

Anh Nguyen is a Ph.D. student in the School of Information Science at Nagoya University. He received his B.E. degree from Ritsumeikan University in 2014. His research interests include Low-Latency GPU Computing and Database Management Systems.

Shinpei Kato is an Associate Professor at the Graduate School of Information Science and Technology, the University of Tokyo. He received his B.S., M.S., and Ph.D. degrees from Keio University in 2004, 2006, and 2008, respectively. He also worked at The University of Tokyo, Carnegie Mellon University, and the University of California, Santa Cruz from 2009 to 2012. His research interests include operating systems, real-time systems, and parallel and distributed systems.

Masato Edahiro is a Professor at the School of Information Science, Nagoya University. He received his Ph.D. degree from Princeton University in 1999. He has also worked at NEC Corporation from 1985 to 2010. His research interests include graph and network algorithms and software for multi- and many-core processors.



Hideyuki Kawashima is an Assistant Professor in the Faculty of Information, Systems and Engineering at the University of Tsukuba. He received his Ph.D. degree from Keio University Japan in 2005. He was a research associate at Keio University from 2005 to 2007. From 2007 to 2011, he was an Assistant Professor at the Graduate School of Systems and Information Engineering and the Center for Computational Sciences at the University of Tsukuba.