

GPU-based Efficient Join Algorithms on Hadoop

Hongzhi Wang, Ning Li , Zheng Wang, Jianing Li

P.O.Box 750, Harbin Institute of Technology, Harbin, China

Abstract

The growing data has brought tremendous pressure for query processing and storage, so there are many studies that focus on using GPU to accelerate join operation, which is one of the most important operations in modern database systems. However, existing GPU acceleration join operation researches are not very suitable for the join operation on big data.

Based on this, this paper speeds up nested loop join, hash join and theta join, combining Hadoop with GPU, which is also the first to use GPU to accelerate theta join. At the same time, after the data pre-filtering and pre-processing, using Map-Reduce and HDFS in Hadoop proposed in this paper, the larger data table can be handled, compared to existing GPU acceleration methods. Also with Map-Reduce in Hadoop, the algorithm proposed in this paper can estimate the number of results more accurately and allocate the appropriate storage space without unnecessary costs, making it more efficient.

The rigorous experiments show that the proposed method can obtain 1.5 to 2 times the speedup, compared to the traditional GPU acceleration equi join algorithm. And in the synthetic data set, the GPU version of the proposed method can get 1.3 to 2 times the speedup, compared to CPU version.

1. Introduction

One of the most serious problems in the computer industry today is the growing data. According to statistics, the rate at which data is generated annually on the network will increase by 10 percent every five years[1]. Therefore, we have to face how to effectively deal with this serious problem in large relational databases. However, the speed of the processor has now grown to the limits of the current level of technology, more and more attention focused on parallel technology. One solution is to increase the number of processors[2] and threads[3]. Another solution is using a single instruction multiple data stream (SIMD) structure to improve the parallelism by processing multiple data under one instruction[4].

Now, due to CPU clock frequency limitations, software optimization has come to an end. Therefore, researchers have to consider other possibilities to speed up the query processing[5], using multicore CPUs[6]. The use of new hardware to speed up the process is also possible[7]. In recent years, many

studies have chosen FPGA as an option for hardware accelerators[8, 9], including the query[10]. Similarly, image processors (GPU) have been widely used in the field of query processing[11].

Join operation is one of the most important operations in relational database operations and one of the longest-running operations in a query. Under this situation, there are many efforts put to speed up the join operation. IBM has added new hardware to its commercial system Netezza[12]. Do[13] integrated CPU processors and DRAM memory into a smart flash device (Smart SSD) to implement query processing. Devarajan[14] and others believe that the GPU is the most advanced distributed tool to handle computationally intensive tasks. Kaldewey[15] and others believed the data needs to be copied to the GPU device memory for processing. He[16] thought using GPU to query co-processing was an effective way to improve memory database performance. Yuan[17] used the GPU device to implement the hash join operation. Pietron[18] and others used the CUDA programming model on the GPU to achieve a part of SQL operations. Rui[19] has conducted detailed experimental studies on how join operation benefits from the rapid growth of the GPU. Angstadt[20] even developed a dedicated to speed up the SQL statement using the CUDA programming model on the GPU. In addition to the two-table join, multi-table join equally occupies a very important position in the relational database, Zhou[21], who proposed GBFSJ (GPUs BloomFilter Star Join) algorithm, achieved a star join on the GPU with the use of a Bloom filter. Cruz[22] and others have used the GPU to achieve such connectivity.

Through the analysis of the existing research results of the new hardware acceleration join, it is concluded that the research results have the following problems:

- Existing join operations based on new hardware are still at the initial stage, most are limited to simple equi join, or lack of research on complex join operation such as theta join. Future research work should focus more on complex join operation for practical applications.
- The existing research carried out by the experiment are based on small data sets, the size of the data set mostly is MB. To use GPU in commercial database systems, future research efforts should be put on scaling operations on large-scale data.
- A serious problem with the new hardware acceleration join operation is how to allocate the appropriate storage space for the join results. Different from the CPU programming language, dynamically allocating storage space, GPU needs to allocate enough storage space in advance. However, the existing research results are not good solutions to this problem.
- A single new hardware cannot meet the needs of modern business databases, but existing research is carried out on monolithic new hardware. There should be more work on deploying new hardware as a distributed architecture to accelerate join operation.

This paper is mainly based on the distributed architecture of the GPU to accelerate the join operation. By combining the Hadoop architecture with the GPU, advantages of both Hadoop's node-level parallelism and GPU's thread-level parallelism can be taken. Hadoop data processing tasks originally performed by the CPU were sent to the GPU, using the GPU's parallelism while opening multiple threads to take advantage of the high computational power and high parallelism. This paper intends to implement nested loop join, hash join and theta join algorithm, the remaining types of join will be studied later.

The second section mainly introduces the basic hardware structure, the thread organization form and the CUDA programming language background, to better understand the following GPU processing join algorithm. The third section and the fourth section mainly introduce the main research contents of this paper, including pre-filtering of data and hardware processing equi join and non-equi join operation. The fifth section introduces the experiment and the results obtained in this paper. The last part concludes the paper, summarizes the contents and points out the innovation.

2. Preliminary

2.1. GPU

The GPU device has a multiparty processor core with multiple instruction streams and multiple streams (SIMDs), and each multiprocessor core contains a number of processors. The GPU hardware structure shown in Figure 1 contains N multiprocessor cores, and each multiprocessor core contains M processors. Each multiprocessor core contains an instruction processing unit and a storage resource. All processors on the multiprocessor share the instruction unit and the storage space, and each processor has a set of registers. GPU devices also have global memory, and global memory can be accessed by all multiprocessor cores. Multi-processor core internal storage resources read and write faster, compared to the global memory.

As shown in Figure 2, in a thread block, the organization of the thread can be one-dimensional, two-dimensional and three-dimensional, with ID identifying each thread. Similarly, the organization of thread blocks in a thread grid can also be divided into one-dimensional, two-dimensional, and three-dimensional. GPU threads have a variety of organizational forms, making it applicable to different issues. For example, when dealing with array problems, the thread grid and the thread block can be used in the one-dimensional organization, and each thread corresponds to a part of the array elements.

2.2. CUDA

GPU devices slowly evolved into a copier processor for intensive computation, bringing a variety of programming languages into birth, such as CUDA. CUDA can be implemented in both CPU and GPU. Additionally, CUDA adds some content related to GPU devices, such as the use of rich thread resources on the GPU.

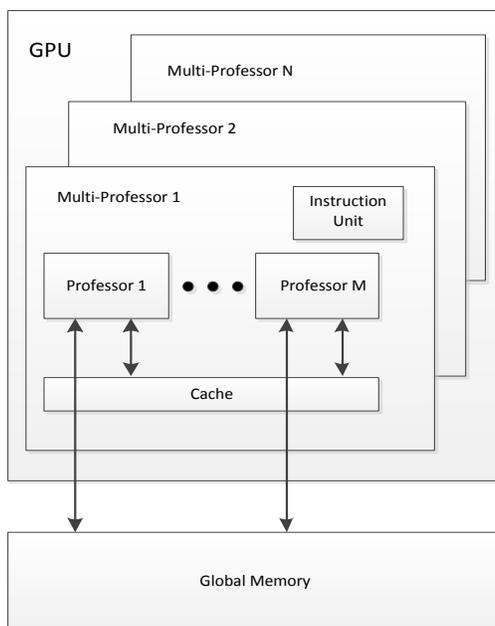


Figure 1: Structure of GPU

CUDA contains two kinds of code, host code, and kernel code. Host code runs on the CPU, which is responsible for the applying for storage space, calling the kernel code, controlling data transformation between CPU and GPU. The kernel code is the code that runs in parallel on the GPU.

2.3. Join Operation

In the relational database, the join operation is the process of the combination of two tables into a relationship table under specific conditions. The attributes that participate in the relationship table are called join keys. If the join key satisfies the query condition, the corresponding tuples in two tables are merged into one tuple and stored in the buffer.

According to the different join conditions, the join operation can be divided into equi join and theta join:

- Equi join: The query statement specifies the join condition for the connection of the equation. Consider the relationships $R(A, B)$, $S(C, D)$. When $R.A = S.C$, it is an equi join. Figure 3 shows the result of the join. In SQL, the syntax of this join is:

select A, B, C, D from R join S on R.A = S.C

- Theta join: The query statement specifies the join condition for the connection of the non-equation. Non-equation includes $>$, $<$, \geq , \leq and so on. Figure 4 shows the result of the join. In SQL, the syntax of this join is:

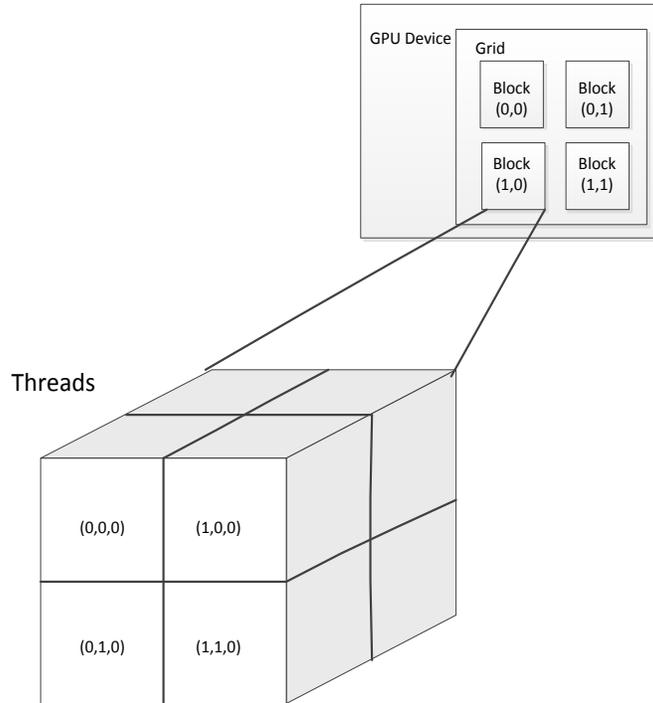


Figure 2: Organization of GPU threads

select A,B,C,D from R join S on R.A < S.C

The join operation can be roughly divided into the following three categories:

- Nested loop join: It is a violent algorithm that converts all tuples in one table to all tuples in the other table, and generates a new result tuple if the join condition is met.
- Hash join: It is to put the smaller table (inner table) into a hash table and store it in memory. And then traverse the larger table (outer table) to find the tuple of the outer table in the hash table.
- Sort merge join: It is to sort two tables at first, and then traverse them in sequence to decide whether to join.

2.4. Hadoop

Hadoop is an open source software framework, which is used for distributed storage and big data processing. The core of this framework is the Hadoop Distributed File System (HDFS) and Map-Reduce.

HDFS, written in Java for the Hadoop framework, is a distributed file system that can store data on commodity machines, providing high bandwidth across the cluster.

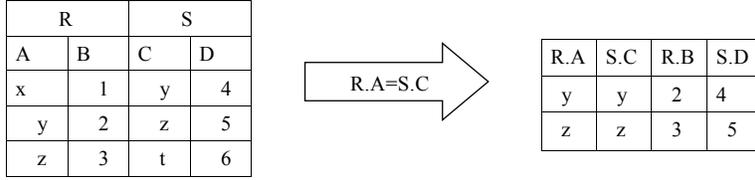


Figure 3: Example of equi join

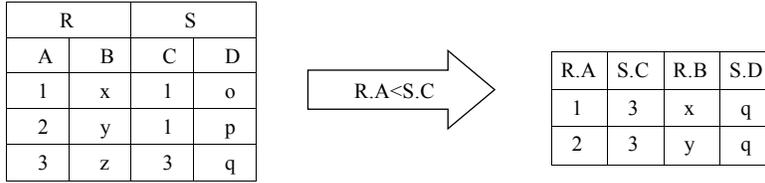


Figure 4: Example of theta join

Map-Reduce is a distributed computing framework to handle big data (greater than 1TB), including Map and Reduce. In the Map phase, the data is read in units of lines and converted to *key – value* pairs. In the Reduce phase, pairs with the same *key* value are passed to the same Reduce function, and the result is finally returned[23, 24, 25, 26, 27].

Using Map-Reduce to handle join operations is the current mainstream approach, roughly divided into three ways, join in Reduce, join in Map, and semi join.

3. GPU-Based Equi Join with Hadoop

In the join algorithm above, only a small part of the data in two tables participates in the equi join operation. Therefore, initial pre-filtering in CPU, passing fewer data between hosts, will be a good solution to this problem. Based on this, this section mainly introduces GPU to accelerate nested loop join and hash join, along with the pre-filtering through Hadoop to reduce the amount of data transferred between devices.

3.1. Data Pre-filtering through Hadoop

In semi join, only tuples of the large group are filtered, by extracting the join key of the small table. When two large tables are connected, filtering only the larger ones is not sufficient to achieve the greatest performance boost. The pre-filter mentioned in this paper extracts common join keys of two tables to pre-filter both tables. So, even if two tables are very large, it can still achieve a good performance. This pre-filtering needs two rounds of Map-Reduce through Hadoop, the following is a specific implementation process.

In the Map phase of the first round of Map-Reduce, firstly read the data of two tables to extract those to be connected attributes, and add a label to

indicate which table they are from. Specifically, Map’s output is (key, tag) , and key is the value of to be connected attribute. When tag is T, it means the tuple is from the table T, the same to those with S. Each tuple corresponds to such a $key - value$ pair. Pairs with the same key are transported to the same Reducer after shuffling, sorting, merging, and so on. In the Reduce phase, the received $(key, value_{list})$ are analyzed. Only those $value_{list}$ contains both T and S key value will be outputted as the result to HDFS in Hadoop. All in all, the first round Map-Reduce’s output is the value of join key attribute in the final result. That is, the tuples whose value of join key attribute are not in it will not appear in the final result.

In the second round of the Map-Reduce task, the result from the previous round is read from the HDFS firstly in the initialization function Setup() and stored in a hash table. Then in the Map phase, read tuples of two to be joined original table, and extract the join key. If the join key exists in the hash table, the tuple is shown in the final result, it is outputted with a tag that identifies its origin. In this step, only those tuples that are determined to be in the final result will be outputted to Reduce to participate in the final operation. In this way, the amount of data to be processed in Reduce declines greatly. At the same time, it reduces the number of tuples that need to be stored on the GPU, freeing up device memory.

In semi join, the raw data is inputted directly to HDFS after filtering, and then a new Map-Reduce is restarted to read and operate the filtered data. However, it will lead to additional Map-Reduce startup time and additional expense of both output and extraction in the new round. Therefore, the pre-filtering method used in this paper incorporates the filtering of data and the execution of the actual join operation into one round of tasks. That is, after filtering, mapping and tagging in Map, data is transferred to Reduce for joining, by shuffling, sorting, merging and other steps. In the Reduce phase, the data from Map is received and processed (row and column transformation), and the processed data is then sent to the GPU for specific join operations. After the GPU is executed, the result is returned to the CPU, and the Reduce outputs it to HDFS. The pseudo code is shown in Algorithm 1.

3.2. Data Preprocessing through Hadoop

Before reading the filtered data and performing the actual join operation, Reduce of Map-Reduce in Hadoop needs to perform some preprocessing of the data. The main processing steps include mapping and row and column transformation. This section describes these two operations and the effects they bring.

3.2.1. Mapping

For a SQL query, only a few attributes in the relational table will be used, and most of them will lead to additional overhead. Therefore, analyzing SQL statement to determine which attributes are query-related will enhance the efficiency. Existing big data query tools, such as HIVE, IMPALA, all follow a

Algorithm 1: Pre-filtering Algorithm

```
1: Map1(null,tuple);
2: join_key ← extract the join key from tuple;
3: emit(join_key,tag);
4: Reduce1(key,tag_list):
5: unique_key ← the key which belongs to both of the two tables;
6: for key in unique_key_list do
7:   emit(key,null);
8: Setup()
9: Build a hash table with the unique_key_list;
10: Map2(null,tuple):
11: join_key ← extract the join key from tuple;
12: join_tuple ← tuple whose join key is contained in the hash table;
13: join_tuple' ← Projection(Join_tuple);
14: emit(join_key/a,tagged join_tuple);
15: Reduce2(key,tag_list):
16: T ← tuples from table T for key;
17: S ← tuples from table S for key;
18: T' ← preprocessing for table T';
19: S' ← preprocessing for table S';
20: NB_T ← number of tuples in T';
21: NB_S ← number of tuples in S';
22: Result ← Join_GPU(T', S', NB_T, NB_S);
23: for tuple in Result do
24:   emit(null,tuple);
```

principle: by analyzing the SQL statement, generate the implementation plan, and optimize the implementation plan, including filtering, mapping operations before the join operation. The algorithm presented in this paper will map the tuples in the Map phase of the second round of Map-Reduce. For each tuple, only query-related attributes are kept, others are removed. Those kept attributes are combined into new tuples. The new tuple has all the query-related attribute values, but the size is smaller than the original tuple. The pseudo code is shown in Algorithm 2.

Algorithm 2: Mapping Algorithm(Projection)

```
1: new_tuple ← null;
2: Attrs[] ← extract attributes from tuple T;
3: new_tuple ← combine relevant attributes in Attrs[] to a new tuple;
```

3.2.2. Row and Column Transformation

Traditional relational databases use row storage patterns, using tuple as a basic storage unit in the disk. All bytes of each tuple are stored adjacently. In online transaction processing (OLTP), the query needs to return all or most of the columns, making this storage mode perform well.

However, in online analytical processing (OLAP), queries often involve millions or more row tuples, but only a few of them are required for queries. For example, a supermarket needs to check the top ten highest sales of commodities this year. Users usually only concern about the name, categories, and sales. At this point, the using of row storage will read a lot of useless data, because

only one line of data can be read at one time, of which only a few field value is required for the query.

The column storage model is a good solution to this flaw. As shown in Figure 5, the column storage model stores the data in columns, and all the data in the same column are stored in adjacent storage space. When storing one row of data, different data fields are stored in the corresponding column's storage space. For the column storage model, it only needs to read the columns associated with the query. In addition, the column storage model performs better in data compression than the line storage model. Although the column storage model has many advantages, after being read from the disk, data still needs to be stored in rows in memory for processing. When the number of to be read columns is too large, the process of row and column transformation will bring too much overhead. Therefore, the column storage model is more suitable for the query with fewer columns involved. At present, many database systems use column storage mode, such as SQL Server[28].

In this paper, because the data has been mapped in the Map of the second round, row and column conversion cannot bring too many benefits. However, for the join operation, a necessary step is to extract the join key value of each tuple. As is shown in Chapter 2.2, CUDA is not quite good at the string processing, for the need of traversing the entire tuple to extract the join key, which will undoubtedly increase the computing burden of the GPU. At the same time, because the tuple contains variables of unsettled length, extracting the join key is not a very wise choice in the kernel. So before sending data to the GPU, row and column still need to be transformed.

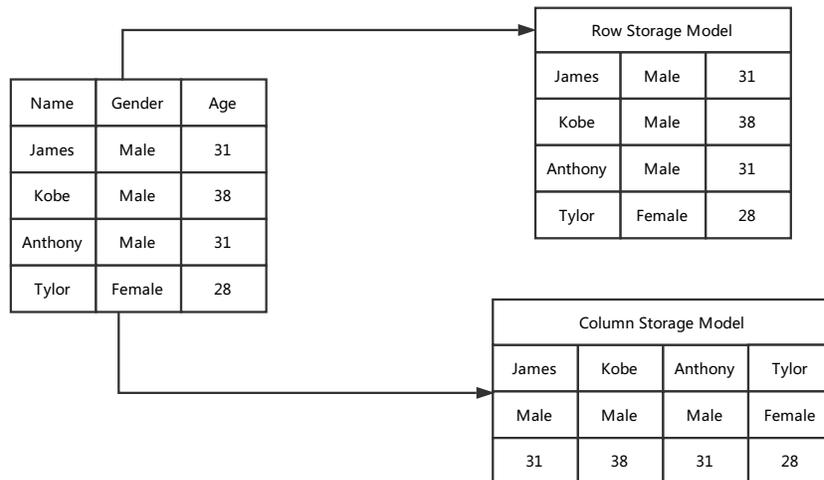


Figure 5: Column storage model

To do the transformation, extract all the attribute values of each tuple and store these attribute values into different buffers. Each buffer stores all the values of an attribute. Note that attributes in each buffer are stored in the same order. For any two different tuples t and s in the original data table, let tuple t be preceded by tuple s . The position of each attribute of the tuple t in the respective buffer is preceded by the corresponding attribute of the tuple. As a result, only comparing the records in the join key buffer and extracting the records in other locations to generate join results are needed. For example, if the m^{th} record in the T table join key buffer matches the n^{th} record in the S table join key buffer, extract the joining result of all m^{th} records of T table buffers and all n^{th} records of S table buffers.

3.3. GPU-Accelerated Equi Join Operation

This section describes the details of equi join operations on the GPU, including the threading model used in CUDA, the specific execution flow of join operations, the estimation and analysis of results number, and the thread hierarchy analysis. In addition, this section implements nested loop joins and hash join operations.

3.3.1. GPU-Based Nested Loop Join

According to Section 2.3, nested loop join needs to loop through two tables, so the use of two-dimensional thread model is essential, that is, the organization of the thread block in the thread grid and the organization of the thread in the thread block are two-dimensional. Figure 6 is a thread grid organization, which contains a total of four thread blocks, each thread block also contains four threads, and are in the form of $2 * 2$ organization. Thus, the line grid in both horizontal axis and vertical axis contain two thread blocks, the same as the organization of threads in the thread block.

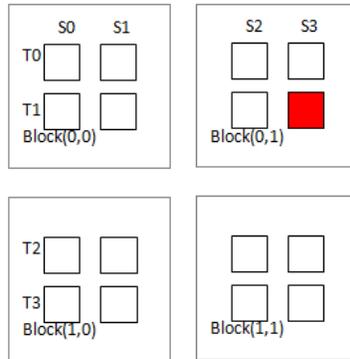


Figure 6: Thread grid organization of nested loop join

When multi-threads process nested loop join, each thread needs to deal with part of the data. Only when all threads have completed their tasks, the GPU will return results to the CPU. Obviously, the total processing time for tasks on

the GPU depends on the last thread. Under this situation, something needs to be done to balance the task for each thread.

In this paper, for table T and table S, assuming that the organization of thread blocks in thread grids is $m * n$, the organization of each thread in blocks is $x * y$. So the organization of whole threads in thread grids is $mx * ny$. The number of tuples to be processed in each table by each thread is given by equation 1 and equation 2:

$$NB_S = \frac{|S|}{m * x} + 1 \quad (1)$$

$$NB_T = \frac{|T|}{n * y} + 1 \quad (2)$$

Suppose that a thread is located in a thread block with the index (a, b) , the index of the thread in the thread block is (c, d) . The position, where to read NB_S and NB_T tuples from table S and T, is shown in equation 3 and 4:

$$ID_S = (a * x + c) * NB_S \quad (3)$$

$$ID_T = (b * y + d) * NB_T \quad (4)$$

ID_S and ID_T are the starting positions of the data in two tables. In Figure 6, the red box represents the thread, which handles the join operation between $T1$ and $S3$. In CUDA, the thread block, the index of the thread in the block, sizes of grid and block can be obtained directly, making it easy to connect the data.

During thread execution, whenever a match is successful and a new tuple is generated, new tuple needs to be cached, and the result is returned when all threads have finished executing. The biggest problem of multithreading writing data into memory is writing memory conflicts. In order to prevent the conflict, each thread can be allocated a fixed storage space. Whenever a thread generates a result, the results will be written to its corresponding storage space. And when all threads have completed the task, the result will be summed and then returned. At the same time, a local variable is set in each thread to record the number of results generated in the thread.

Since the exact number of join results is not known until the join operation is performed, when allocating storage space for each thread, space is set to a maximum possible number of results, that is, $NB_S * NB_T$.

3.3.2. GPU-Based Hash Join

According to Section 2.3, hash join only needs to loop through one data table. So one-dimensional organization of both blocks in grid and threads in the block are more suitable for CUDA. As shown in Figure 7, the grid contains four thread blocks, and each thread block contains two threads.

Assuming that each thread grid contains n_{blocks} thread blocks, each thread block contains $n_{threads}$ threads, and if the table S is the larger table, the number of tuples in the table S allocated to each thread is given by the equation 7:

$$NB_S = \frac{|S|}{m * x} + 1 \quad (5)$$

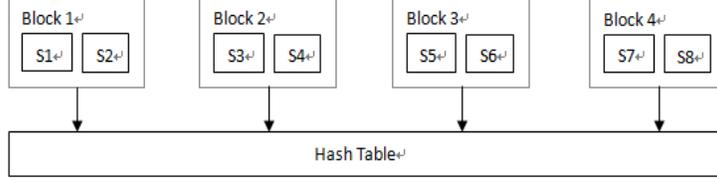


Figure 7: Thread grid organization of hash join

In fact, in addition to the last thread in the last thread block, each thread handles NB_S tuples of table S, and the remaining tuples are handled by the last thread. If a block index in the grid is p , and the index of thread in the block is q , the location from where to start reading NB_S tuples in table S can be calculated by equation 6:

$$ID_S = (p * n_{threads} + q) * NB_S \quad (6)$$

This paper introduces the hash bucket to handle hash conflicts. Whenever a tuple is detected, it is first positioned to the corresponding hash bucket through the hash function, followed by a small range of detection in the bucket. Hash bucket size is fixed, so the space utilization is slightly inadequate, but the detection efficiency is as good as others.

As it shown in Algorithm 1, when the data is sent to the GPU, NB_T and NB_S are sent as well. So $NB_T * NB_S$ is set as the maximum storage space.

3.3.3. Estimation of Join Results Size

As mentioned earlier, before the kernel program performs the actual join operation, it is necessary to preallocate enough memory in the CPU for the result. But before the join is completed, we can not know the specific number of results. Usually, memory is set with regard to the size of the Cartesian product. But obviously, when tables are too large, the size of the Cartesian product will be far larger than that of device memory. The existing research is based on the experiments on small data set, so even if the distribution of memory is in accordance with the Cartesian product, it does not exceed the device memory. Although the method proposed in this paper is based on the Cartesian product as well, due to the data pre-filtering and Map-Reduce structure, the performance of the estimation is far better than that of the existing results.

Assuming that the ratio of tuples participating in the join of table T and table S is α and β . So, after pre-filtering, $\beta * |S|$ and $\gamma * |T|$ tuples respectively perform the actual join operation. As mentioned before, tuples with the same *key* value are passed to the same Reducer. If there are k Reducers, each Reducer deals with $\omega_1 * \beta * |S|, \omega_2 * \beta * |S|, \dots, \omega_k * \beta * |S|$ S-tuples and $\lambda_1 * \gamma * |T|, \lambda_2 * \gamma * |T|, \dots, \lambda_k * \gamma * |T|$ T-tuples, and the parameters satisfy the following conditions:

$$\begin{aligned} \omega_1 + \omega_2 + \dots + \omega_k &= 1 \\ \lambda_1 + \lambda_2 + \dots + \lambda_k &= 1 \end{aligned} \quad (7)$$

R_{size} results need to be allocated storage space:

$$R_{size} = \gamma * \beta * |S| * |T| \sum_{i=1}^k \omega_i * \lambda_i \quad (8)$$

Because parameters in equation 8 are all below 1, it will save a lot of space for pre-allocating memory.

3.3.4. Thread Hierarchy

In the standard CUDA program, the thread is organized in the form of thread block-thread hierarchy, where the number of thread blocks and the number of threads in each thread grid can be set, but there is only one thread grid. The algorithm in this paper combines Hadoop with the GPU. The Map-Reduce architecture is also a multi-threaded task. Multiple Reducers perform tasks in parallel. Each Reducer corresponds to a thread. Therefore, the hierarchy of the proposed algorithm is thread grid (Reducer)-thread block-thread.

In CUDA programming, the number of threads in each block is usually set to a multiple of 32, the number of blocks is based on the amount of data, usually the more the better. Assuming that the GPU has a total of N threads, the reducer number is $n_{reducers}$, and each grid contains n_{blocks} of blocks. Each thread block contains $n_{threads}$ threads. The parameter relationship must satisfy the relationship 9

$$n_{reducers} * n_{threads} * n_{blocks} < N \quad (9)$$

That is, in the case where the relationship is established, the bigger product of parameters the better. When the total number of set threads is greater than the number of threads on the GPU, some tasks need to wait for a free thread. In this way, the task waiting and the creation of multiple threads will bring a lot of additional overhead. As shown in Algorithm 1, the number of Reducers can be changed by modifying the parameter α .

3.4. Mixed Programming of GPU and Hadoop

The Map-Reduce program runs on CPU and is based on Java language. While, the GPU kernel program runs on GPU, based on the CUDA programming model and the C language. Therefore, the combination of Hadoop and GPU involves a compatibility problem between Java and C. This experiment is programmed through the JNI interface to complete this work. Since the file format of the CUDA program is .cu, and the traditional C language is the .c file, there is a need of an additional .c file as a middleware. Through this middleware, GPU kernel function can be called. And then Map-Reduce will call the middleware, so as to achieve the join of CPU and GPU. Because the memory cannot be shared between CPU and GPU, the same data from Java to C needs a copy, and then another copy from C memory space to GPU device memory. That is, to transfer data from Map-Reduce to GPU, and a total of two copies are required. Obviously, this introduces additional overhead and memory footprint, so future research should continue to solve this problem.

4. GPU-Based Theta Join with Hadoop

Given the importance of Hadoop, there are many research focused on using Map-Reduce to deal with theta join operations. Koumarelas[29], Okcan[30] and Penar[31] achieved theta join operations of two tables on Map-Reduce, meanwhile Yan[32], Zhang[33] and Changchun[34, 35] achieved a multi-table theta join. Augustyn[36] used the GPU to estimate the join selectivity for theta join of two tables.

This section focuses on how to combine Hadoop with a GPU to handle theta join operations based on two data tables, multi-table join will be studied later.

4.1. Classic Theta Join Algorithm on Hadoop

For theta joins, simply passing tuples with the same join key to the same Reducer is not sufficient to get the full result. Because in theta join a tuple does not only need to match its tuples with the same join keys, but also need to match the tuples that are greater or less than the join key. As shown in Figure 8, the areas represented by the shaded parts all need to be matched.

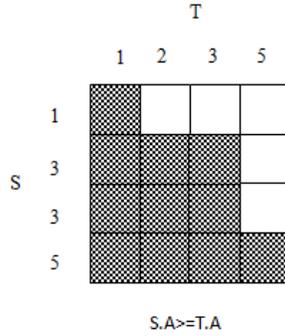


Figure 8: Matrix in classic algorithm

Okcan[30] proposed a classic Map-Reduce processing theta join operation algorithm, named M-Bucket-I. As shown in Figure 9(a), the Cartesian product of two tables is represented by a two-dimensional matrix. The matrix is divided into many areas, each region corresponds to a Reducer, and each region has an ID. In Map function, for each tuple, a $(ID, \langle Record, tag \rangle)$ key-value pair is generated for each region intersecting it, where tag is the identity that identifies the tuple source. As a result, tuples with the same ID, that is, tuples located in the same area, are passed to the same Reducer for processing. In this way, it is possible to ensure that each element in the matrix can be processed and led to the correct result.

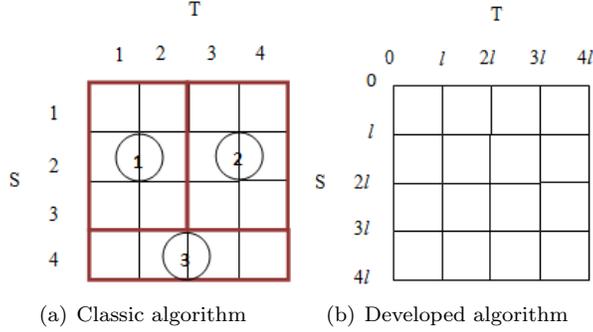


Figure 9: Theta join algorithm on Map-Reduce

In this algorithm, only the area where the result is determined is processed, and no actual processing is performed on the data that does not produce the result area. However, this algorithm requires some information about the original data in advance, which means additional overhead is needed.

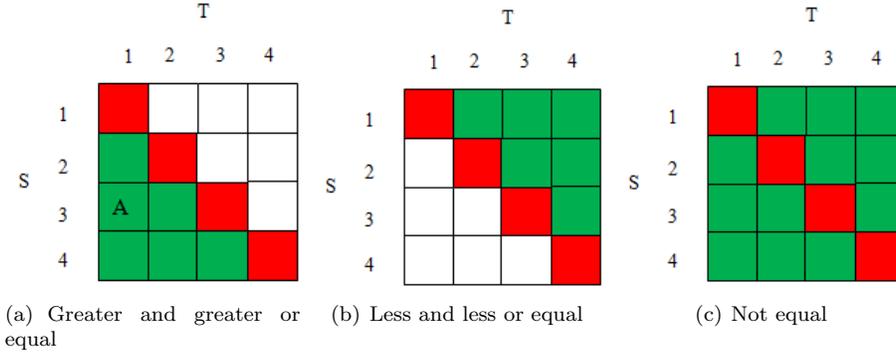


Figure 10: Organization of matrix under different join conditions

4.2. Improved Theta Join Algorithm on Hadoop and GPU

To solve this problem, this paper presents an improved algorithm. On the basis of the classic algorithm, this algorithm can further identify certain areas, where tuples from two tables satisfy the join condition.

Similar to the M-Bucket-I, this algorithm calculates the range of each table join key and k quantiles of each range. Figure 9(b) shows the organization of the two-dimensional matrix when k is 4. In this matrix, each element does not represent a tuple pair, but a range pair. Unlike the M-Bucket-I where the divided area is the rectangle of any size, this algorithm divides the area into strict squares. The advantage of this division is, for some regions, if any one of the tuple pairs meets the join condition, there is no need to judge the data in this region for the join.

As shown in Figure 10(a), when the join condition is that join key attribute value of S-table is greater than or greater than or equal to that of T-table, any pair of tuple pairs in the green region satisfies the join condition, and the tuple pair in the red region needs to be judged. While the tuple pair in the white region must not satisfy the join condition. Figure 10(b) shows that of S-table is less than or less than or equal to that of T-table. And Figure 10(c) show the condition that two values are not equal. So only the red area of the data needs to be sent to the GPU for theta join processing. And the other areas can be judged directly by the area ID. The pseudo code is shown in algorithm 3.

Algorithm 3: theta join on Map-Reduce

```

1: Map(key:null, value: a tuple from a split of either table)
2: join_key ← extract the join key from value;
3: if value ∈ S then
4:   RegionID.x ← bucket ID of value according to its join key;
5:   for all possible RegionID.y according to the join condition do
6:     emit(< RegionID.x, RegionID.y >, < value, "S" >);
7: if value ∈ T then
8:   RegionID.y ← bucket ID of value according to its join key;
9:   for all possible RegionID.x according to the join condition do
10:    emit(< RegionID.x, RegionID.y >, < value, "T" >);
11: Reduce(key:RegionID, value_list: tagged tuples corresponding in RegionID)
12: T ← null;
13: S ← null;
14: for each tuple t in value_list do
15:   add t to S or T according to its tag;
16: if RegionID.x == RegionID.y then
17:   result ← GPU_theta_join(S, T);
18:   for each record in result do
19:     emit(null, record);
20: else if RegionID.x matches RegionID.y according to the join condition then
21:   cartesian_result ← do cross join for S and T;
22:   for each record in cartesian_result do
23:     emit(null, record);

```

As shown in Figure 10(a), the A region coordinates are (3, 1) and the abscissa is greater than the ordinate. Therefore, any tuple in the A region satisfies the condition, that join key attribute value of table S is greater than that of table T. When the abscissa is equal to the ordinate, the region may have a join result, so it is necessary to send it to the GPU for theta join judgment.

In the theta join operation on the GPU, nested loop join algorithm can be used, the thread organization form and the processing mode are the same as those of the GPU in Chapter 3.

The method used in this article only applies the data in the diagonal area to the GPU, so it is only necessary to allocate storage space for the data in the diagonal area, which can increase the storage space utilization.

5. Experimental Results

This section compares the methods proposed in this paper, through the existing GPU accelerated join operation algorithm to verify whether it has improved performance, being more efficient. At the same time, compared with the CPU

implementation of the proposed algorithm in this paper, to verify whether the GPU implementation has the speedup for the join operation.

The experiments done in this section are based on larger datasets. Unless special instructions, all experimental raw data are TPC-H data sets. GPU devices running on the Linux operating system, the version of Ubuntu 14.04, and Hadoop is version 2.6.0.

5.1. Nested Loop Join

This section focuses on the experiments of the nested loop join. It is compared separately with the GPU accelerated nested loop join with a single device and the proposed algorithm in CPU. At the same time by changing the value of α in algorithm 1 to change the number of Reducers started by Map-Reduce, changes in the execution time can be observed. At the same time, through the synthetic data, the performance of the method can be observed. If there is no special description, the value of α is 100. This experiment is based on the small dataset, because the cost of the nested loop join is very large, the traditional GPU acceleration method is not able to effectively support large data sets.

5.1.1. Comparison of Nested Loop Join with Single GPU

In this experiment, in order to ensure the reliability of the experimental results, the proposed method is also implemented on a single GPU device, and Hadoop is a pseudo-distributed structure. As shown in Figure 16, when the data set is small, the efficiency of the proposed method is lower than that of the traditional one. However, the execution time of the traditional method is significantly increased with the increase of the original dataset, which is different to the proposed algorithm. As shown in the figure, it will have at least one times the speedup over the traditional one, which means under the same accelerating condition of GPU, the proposed method is more efficient.

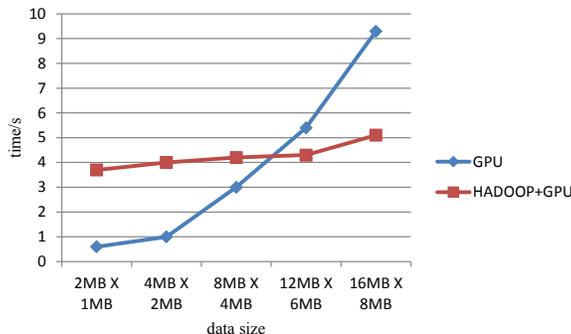


Figure 11: Comparison of nested loop join with single GPU

Data that does not appear in the final result is filtered out, and the data that actually performs the join operation is much smaller than the original data set. After pre-filtration, the actual processing time is greatly reduced, but the pre-filtration process also introduces additional time overhead.

Therefore, when the data set is small, the gain effect of the pre-filtering is lower than that of the additional cost, so the method proposed in this paper has a long execution time at first. With the increase of the data set, the gain effect exceeds the overhead, so the method proposed in this paper is significantly lower than the traditional method.

5.1.2. Comparison of Nested Loop Join with CPU on TPC-H Data Set

The value of α in the CPU is 1, which ensures the best execution efficiency. Except for the value of α and the equipment, the implementation process of GPA and CPU is exactly the same. As shown in Figure 12, compared with the CPU implementation, using the GPU to achieve the proposed nested loop join algorithm has no obvious speedup. This is because, in Reduce phase of the second round of Map-Reduce of CPU version, only tuples with the same join key will be passed to the same Reducer. In this experiment, the distribution of the TPC-H data join keys is even. Each join key value corresponds to only a small number of tuples. Therefore, the actual execution time of the join operation in the CPU version is very small, most of the time is taken by data pre-filtering. And GPU can only accelerate small proportion of data, so the effect is not obvious.

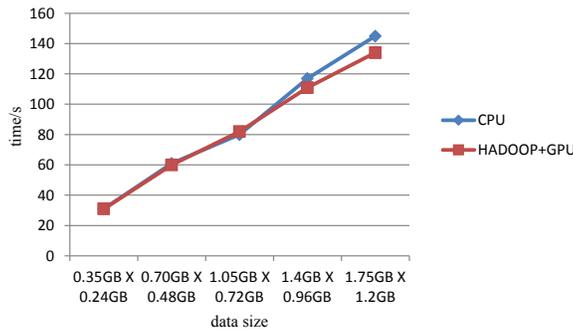


Figure 12: Comparison of nested loop join with CPU on TPC-H data set

5.1.3. Comparison of Nested Loop Join with CPU on Synthetic Data Set

Based on the unobvious effect of the proposed algorithm on the TPC-H data set, the synthetic data set is made for another experiment. In the synthetic data set, the number of tuples corresponding to each join key is more than that of the TPC-H data set. To ensure the reliability of the experimental results, the distribution of join keys has been modified on the basis of the experimental data set shown in Figure 12. Specifically, dividing the join key by a positive integer, the number of tuples corresponding to each join key increases. As can be seen from Figure 13, in the synthetic dataset, compared to the CPU, using the GPU can get nearly 2 times the speedup. Through Figure 16 and Figure 12, the proposed method can be applied to larger data sets than traditional GPU algorithms. Because the join key value range in the synthetic data set is much

smaller than that in the TPC-H dataset, when the value of α is small, a good speedup can be obtained. Therefore, α is 5 in this experiment.

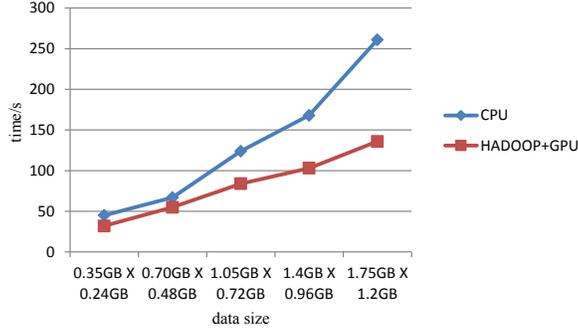


Figure 13: Comparison of nested loop join with CPU on synthetic data set

5.1.4. Effect of α on the Execution Efficiency of Nested Loop Join

In the algorithm `refprefilteringalgo`, the value of α will affect the parallel number of the algorithm. To clarify the influence of α on the performance of the algorithm, we can observe the execution time of the algorithm by changing the value of α . This experiment uses a TPC-H data set, the data set size is 1.75GB and 1.2GB. The experimental results are shown in Figure 14. When α is small, with the increase of α , the execution time of the algorithm decreases gradually. When α increases to a certain extent, with the increase of α value, time also increases.

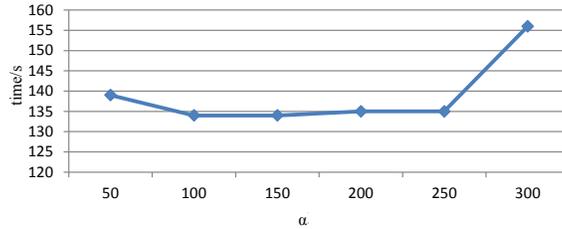


Figure 14: Effect of α on the execution efficiency of nested loop join

When the value of α is small, the number of Reducers is larger, resulting in the total number of threads of the algorithm beyond the total number of GPU devices. Some tasks need to wait for the other tasks to be completed before they can be assigned to the free thread to perform the task. The waiting time plus the overhead of creating multiple threads make the execution time of the algorithm higher when the value of α is less. When the value of α is large enough that the number of total threads of the algorithm is smaller than that of the total number of GPU devices, with the increase of α , the number of Reducer decreases. It leads to the decrease of the total number of threads, the increase

of data amount processed by each thread, so the algorithm execution time will increase with the increase of α . It can be seen from Figure 14 that the algorithm will obtain the ideal execution effect when the value of α is between 100 and 250.

5.2. Hash Join

This section focuses on the experiment of accelerating the hash join algorithm. The experimental results are observed by comparing the hash algorithm with the single GPU and the CPU implementation. And in the synthetic data set, the efficiency and the speedup of the proposed method can be observed. Additionally, by changing the α value, this section observes the influence of α on the algorithm. If there is no special instruction, α value is 10000.

5.2.1. Comparison of Hash Join with Single GPU

In this experiment, the value of α is 10000, which is much larger than the α when dealing with nested loop join. This is because the hash join calculation task is much smaller than the nested loop join, so larger amounts of data processed in each Reducer can give full play to the advantages of GPU.

As can be seen from the Figure 15, when the data set is small, the proposed method has lower efficiency, compared to the traditional GPU algorithm. When the data set size is good enough, the proposed method is better than the traditional GPU algorithm. The experimental results show that the proposed method can achieve 2 times the speedup over traditional one, which means under the same accelerating condition of GPU, the proposed method is more efficient.

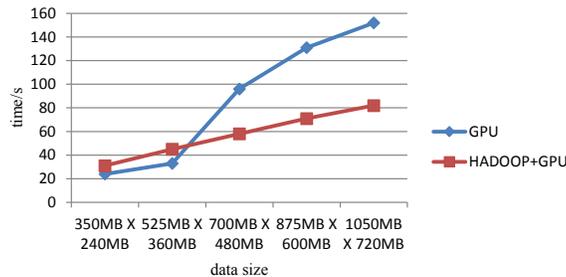


Figure 15: Comparison of hash join with single GPU

5.2.2. Comparison of Hash Join with CPU on TPC-H Data Set

When the data set is small, most of the time is used to process the data pre-filter. When the data pre-filter takes longer than the time it reduces on the actual join operation, the total time will increase. While as the data set continues to increase, the method proposed in this paper will achieve better results, because the data pre-filter is less time-consuming. By comparing with Figure 16, it is found that traditional GPU processing algorithms can handle

larger amounts of data when handling hash joins. This is because when the data set is large, the hash join algorithm performance is much higher than the nested loop join, so it can handle more data sets.

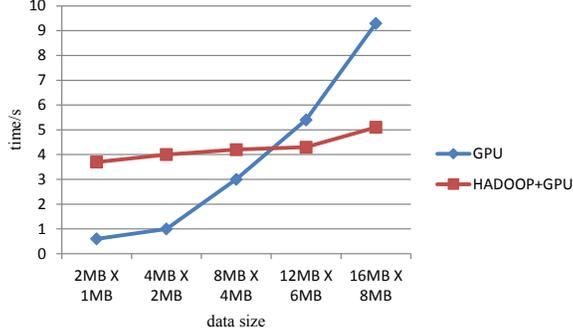


Figure 16: Comparison of hash join with CPU on TPC-H data set

5.2.3. Comparison of Hash Join with CPU on Synthetic Data Set

Data is synthesized in the same way as nested loops. As shown in Figure 17, in the synthetic data, the proposed GPU algorithm is better than its CPU version, with 1.3 times speedup. Hash join speedup is not good as nested loop join, because the nested loop join spends more time in the actual join processing, so it can be optimized in the proportion of large.

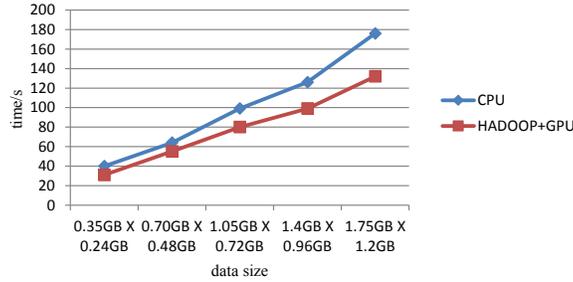


Figure 17: Comparison of hash join with CPU on synthetic data set

5.2.4. Effect of α on Execution Efficiency of Hash Join

This experiment uses TPC-H data set, the data set size is 1.75GB and 1.2GB. Similarly, α is changed in the experiment to observe its effect on algorithm performance. Figure 18 shows the change in the execution time of the hash join algorithm proposed in this paper when the value of α changes. When α is small, the execution time of the algorithm decreases obviously as α becomes larger. When the value of α is too small, the total number of threads of the algorithm exceeds the total number of threads of the GPU device, so the execution time

is longer. Since the computational task of the hash join is much smaller than the nested loop join, the overall execution time does not increase significantly when the α value continues to increase, although the amount of data processed in each Reducer increases. If the value of α continues to increase to a certain extent, the algorithm's overall execution time will continue to increase.

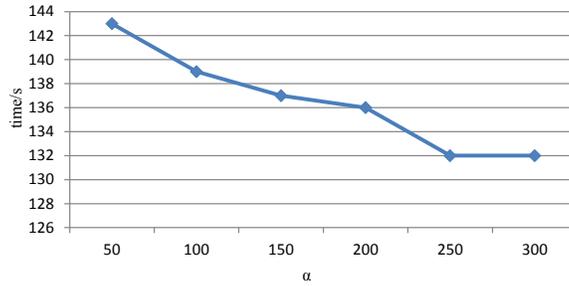


Figure 18: Effect of α on the execution efficiency of hash join

5.3. Theta Join

This section compares the theta joins. Because there is no available mature GPU accelerating theta join, this experiment compares with the CPU. This experiment is based on a small dataset. In order to ensure the validity of the experimental results, implementation details are exactly as described in Algorithm 3

In contrast to the equi join, the theta join has a high selectivity, so the size of the theta join results is larger than the magnitude of input data. The results of the experiment are shown in Figure 19. It can be seen from the figure that the Hadoop-based theta join algorithm on the GPU is better than its CPU version, and the speedup is about twice as much as the latter, which means under the same accelerating condition of GPU, the proposed method is more efficient.

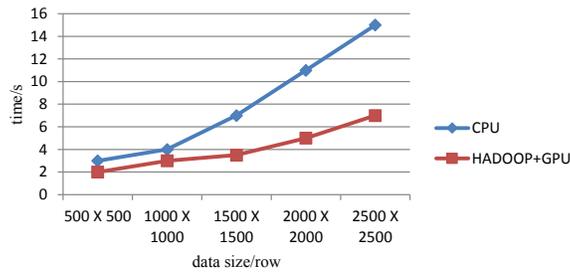


Figure 19: Comparison of theta join on small data set

We also experimented on larger datasets. Since the join results are out of memory when the data set is large, we do not store the results. As shown in Figure 20, when the data set is large, GPU implementation is still better than the CPU implementation.

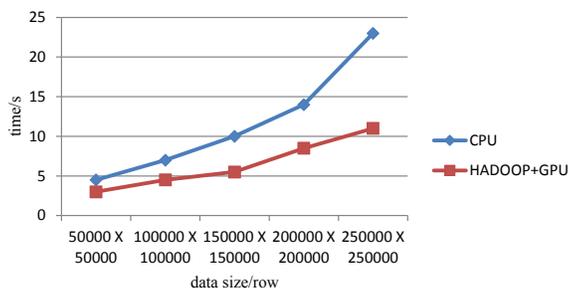


Figure 20: Comparison of theta join on small data set

6. Conclusion

This paper focuses on Hadoop-based join operation acceleration tasks on image processors (GPUs). GPU was originally developed as an image processing, and nowadays, more and more GPU applications appear in general computing tasks, such as machine learning, data mining and other fields. Based on the GPU's powerful computing power and high parallelism, there is a lot of research focused on using it to speed up database operations. In the field of the modern database, the join operation as a computationally intensive task is the main problem.

In research results of existing GPU accelerating join operations, although the use of its strong parallelism significantly increases the efficiency of join execution, it is not good because of the limited storage resources of the GPU device and the limited functionality of the universal programming language CUDA. The existing research results are based on smaller data sets, and therefore can not be applied to the practical application on a large scale.

Based on this idea, the distributed computing platform Hadoop is combined with the GPU, by referring to the idea of a CPU filtering join algorithm. By initially filtering the raw data table through the first round of the Map-Reduce task at the CPU, it will filter the tuple that does not appear in the results, and only send the connectable tuples to the GPU device for actual join operations. By reducing the amount of data actually processed, it is possible to reduce the utilization of the storage space on the device while improving the efficiency of the algorithm execution so that it can handle a larger amount of data. At the same time, it is possible to estimate the number of join results more accurately without introducing additional overhead, to allocate accurate storage in advance and reduce the storage space occupancy rate. In addition to the equi join, this article is the first to use the GPU to accelerate the theta join operation, which still uses the combination of Hadoop and GPU.

The followings are the conclusions:

- The proposed algorithm is more efficient than the existing single GPU device, and it can be applied to the larger dataset. Compared with the CPU implementation, the GPU algorithm proposed in this paper has no obvious

speedup on the dataset with fewer key numbers and fewer tuples corresponding to each join key. However, the algorithm proposed in this paper can achieve 2 times the speedup on a dataset with more corresponding tuples.

- The accelerating hash join algorithm proposed in this paper can achieve 2 times the speedup, compared with the existing GPU acceleration hash join. Similarly, the GPU implementation of the algorithm proposed in this paper has no obvious speedup compared to the CPU implementation, when the number of connected keys is large, but each join key value corresponds to a few tuples. When the number of corresponding tuples is big enough, GPU implementation can get 1.3 times the speedup.
- In this paper, the theta join processing algorithm, compared to the CPU implementation, GPU implementation can get 2 times the speedup.

Compared with the existing research results, the research content of this paper has achieved the following innovative achievements:

- This article is the first to use the filter join algorithm on the GPU to deal with the equi join operation. One round of Map-Reduce filters out non-connectable tuples and sends only connectable tuples to the GPU device. By reducing the processing time of the actual join operation and the occupancy rate of the device memory, it can handle more data.
- The proposed method can accurately estimate the size of the equi join result without introducing additional overhead, and allocate the appropriate storage space for the result, making the GPU storage space more efficient, more suitable for large-scale data sets.
- Among the existing GPU accelerating equi join operation, this article is the first to have an experiment on a larger data set (GB level).
- This article is the first to use GPU to accelerate theta join operations.

Although there are a lot of research results on GPU accelerating join, these results are not enough to be applied to the commercial database system. So the future work should continue to conduct in-depth research, including the following:

- The future work should use multiple GPU devices to deal with large datasets on the join operation, in a distributed architecture. Although in this article Hadoop and GPU were combined, because of limited resources, the experiment is not completed in the real large data set. Future research should deal with TB-level datasets on multiple GPU devices.
- This paper only implements the join operation of two tables. Future research work should include more complicated join operations, such as accelerating multi-table join and similarity join.

Acknowledgement. This paper was partially supported by NSFC grant U1509216,61472099, National Sci-Tech Support Plan 2015BAH10F01, the Scientific Research Foundation for the Returned Overseas Chinese Scholars of Heilongjiang Province LC2016026 and MOE-Microsoft Key Laboratory of Natural Language Processing and Speech, Harbin Institute of Technology.

7. References

References

- [1] J. F. Gantz, The diverse and exploding digital universe, An Idc White Paper Retrieved.
- [2] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, A. Zeier, Fast updates on read-optimized databases using multi-core cpus, PVLDB 5 (1) (2011) 61–72.
URL http://www.vldb.org/pvldb/vol15/p061_jenskrueger_vldb2012.pdf
- [3] B. W. Low, B. Y. Ooi, C. S. Wong, Scalability of database bulk insertion with multi-threading, in: J. M. Zain, W. M. B. W. Mohd, E. El-Qawasmeh (Eds.), Software Engineering and Computer Systems - Second International Conference, ICSECS 2011, Kuantan, Pahang, Malaysia, June 27-29, 2011, Proceedings, Part III, Vol. 181 of Communications in Computer and Information Science, Springer, 2011, pp. 151–162. doi:10.1007/978-3-642-22203-0_14.
URL https://doi.org/10.1007/978-3-642-22203-0_14
- [4] J. Zhou, K. A. Ross, Implementing database operations using SIMD instructions, in: M. J. Franklin, B. Moon, A. Ailamaki (Eds.), Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, ACM, 2002, pp. 145–156. doi:10.1145/564691.564709.
URL <http://doi.acm.org/10.1145/564691.564709>
- [5] B. Zhang, X. Wang, Z. Zheng, The optimization for recurring queries in big data analysis system with mapreduce, Future Generation Computer Systems.
- [6] V. Silva, J. Leite, J. J. Camata, D. de Oliveira, A. L. Coutinho, P. Valduriez, M. Mattoso, Raw data queries during data-intensive parallel workflow execution, Future Generation Computer Systems 75 (Supplement C) (2017) 402 – 422.
- [7] D. J. DeWitt, DIRECT - A multiprocessor organization for supporting relational database management systems, IEEE Trans. Computers 28 (6) (1979) 395–406. doi:10.1109/TC.1979.1675379.
URL <https://doi.org/10.1109/TC.1979.1675379>

- [8] J. Teubner, R. Müller, G. Alonso, Frequent item computation on a chip, *IEEE Trans. Knowl. Data Eng.* 23 (8) (2011) 1169–1181. doi:10.1109/TKDE.2010.216.
URL <https://doi.org/10.1109/TKDE.2010.216>
- [9] L. Woods, J. Teubner, G. Alonso, Real-time pattern matching with fpgas, in: S. Abiteboul, K. Böhm, C. Koch, K. Tan (Eds.), *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, IEEE Computer Society, 2011*, pp. 1292–1295. doi:10.1109/ICDE.2011.5767937.
URL <https://doi.org/10.1109/ICDE.2011.5767937>
- [10] J. Singaraju, A. Thamarakuzhi, J. A. Chandy, Active storage networks: Using embedded computation in the network switch for cluster data processing, *Future Generation Computer Systems* 45 (Supplement C) (2015) 149 – 160. doi:<https://doi.org/10.1016/j.future.2014.10.020>.
URL <http://www.sciencedirect.com/science/article/pii/S0167739X14002143>
- [11] Á. B. Hernández, M. S. Perez, S. Gupta, V. Muntés-Mulero, Using machine learning to optimize parallelism in big data applications, *Future Generation Computer Systems*.
- [12] M. Singh, B. Leonhardi, Introduction to the IBM netezza warehouse appliance, in: J. W. Ng, C. Couturier, M. Litoiu, E. Stroulia (Eds.), *Center for Advanced Studies on Collaborative Research, CASCON '11, Toronto, ON, Canada, November 7-10, 2011, IBM / ACM, 2011*, pp. 385–386.
URL <http://dl.acm.org/citation.cfm?id=2093965>
- [13] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, D. J. DeWitt, Query processing on smart ssds: opportunities and challenges, in: K. A. Ross, D. Srivastava, D. Papadias (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, ACM, 2013*, pp. 1221–1230. doi: 10.1145/2463676.2465295.
URL <http://doi.acm.org/10.1145/2463676.2465295>
- [14] N. Devarajan, S. Navneeth, S. Mohanavalli, GPU accelerated relational hash join operation, in: *International Conference on Advances in Computing, Communications and Informatics, ICACCI 2013, Mysore, India, August 22-25, 2013, IEEE, 2013*, pp. 891–896. doi:10.1109/ICACCI.2013.6637294.
URL <https://doi.org/10.1109/ICACCI.2013.6637294>
- [15] T. Kaldewey, G. M. Lohman, R. Müller, P. B. Volk, GPU join processing revisited, in: S. Chen, S. Harizopoulos (Eds.), *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012, ACM, 2012*, pp. 55–62. doi:

10.1145/2236584.2236592.
URL <http://doi.acm.org/10.1145/2236584.2236592>

- [16] J. He, M. Lu, B. He, Revisiting co-processing for hash joins on the coupled CPU-GPU architecture, *PVLDB* 6 (10) (2013) 889–900.
URL <http://www.vldb.org/pvldb/vol16/p889-he.pdf>
- [17] Y. Yuan, R. Lee, X. Zhang, The yin and yang of processing data warehousing queries on GPU devices, *PVLDB* 6 (10) (2013) 817–828.
URL <http://www.vldb.org/pvldb/vol16/p817-yuan.pdf>
- [18] M. Pietron, P. Russek, K. Wiatr, Accelerating select where and select join queries on a GPU, *Computer Science (AGH)* 14 (2) (2013) 243–252. doi:10.7494/csci.2013.14.2.243.
URL <https://doi.org/10.7494/csci.2013.14.2.243>
- [19] R. Rui, H. Li, Y. Tu, Join algorithms on gpus: A revisit after seven years, in: 2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015, IEEE, 2015, pp. 2541–2550. doi:10.1109/BigData.2015.7364051.
URL <https://doi.org/10.1109/BigData.2015.7364051>
- [20] K. Angstadt, E. Harcourt, A virtual machine model for accelerating relational database joins using a general purpose GPU, in: L. T. Watson, J. Weinbub, M. Sosonkina, W. I. Thacker (Eds.), Proceedings of the Symposium on High Performance Computing, HPC 2015, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015, SCS/ACM, 2015, pp. 127–134.
URL <http://dl.acm.org/citation.cfm?id=2872615>
- [21] G. Zhou, G. Wang, GBFSJ: bloom filter star join algorithms on gpus, in: 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015, Zhangjiajie, China, August 15-17, 2015, IEEE, 2015, pp. 2427–2431. doi:10.1109/FSKD.2015.7382334.
URL <https://doi.org/10.1109/FSKD.2015.7382334>
- [22] M. S. H. Cruz, Y. Kozawa, T. Amagasa, H. Kitagawa, GPU acceleration of set similarity joins, in: Q. Chen, A. Hameurlain, F. Toumani, R. Wagner, H. Decker (Eds.), Database and Expert Systems Applications - 26th International Conference, DEXA 2015, Valencia, Spain, September 1-4, 2015, Proceedings, Part I, Vol. 9261 of Lecture Notes in Computer Science, Springer, 2015, pp. 384–398. doi:10.1007/978-3-319-22849-5_26.
URL https://doi.org/10.1007/978-3-319-22849-5_26
- [23] J. Myung, J. Shim, J. Yeon, S. Lee, Handling data skew in join algorithms using mapreduce, *Expert Syst. Appl.* 51 (2016) 286–299. doi:10.1016/j.eswa.2015.12.024.
URL <https://doi.org/10.1016/j.eswa.2015.12.024>

- [24] T. Yuan, Z. Liu, H. Liu, Optimizing hash join with mapreduce on multi-core cpus, *IEICE Transactions 99-D* (5) (2016) 1316–1325.
URL http://search.ieice.org/bin/summary.php?id=e99-d_5_1316
- [25] M. A. H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using mapreduce, in: D. Abramson, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, P. M. A. Sloot (Eds.), *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, Vol. 29 of *Procedia Computer Science*, Elsevier, 2014, pp. 145–158. doi:10.1016/j.procs.2014.05.014.
URL <https://doi.org/10.1016/j.procs.2014.05.014>
- [26] T. Csar, R. Pichler, E. Sallinger, V. Savenkov, Using statistics for computing joins with mapreduce, in: A. Cali, M. Vidal (Eds.), *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management, Lima, Peru, May 6 - 8, 2015.*, Vol. 1378 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015.
URL http://ceur-ws.org/Vol-1378/AMW_2015_paper_13.pdf
- [27] F. N. Afrati, N. Stasinopoulos, J. D. Ullman, A. Vasilakopoulos, Sharesskew: An algorithm to handle skew for joins in mapreduce, *CoRR* abs/1512.03921. arXiv:1512.03921.
URL <http://arxiv.org/abs/1512.03921>
- [28] P. Larson, E. N. Hanson, S. L. Price, Columnar storage in SQL server 2012, *IEEE Data Eng. Bull.* 35 (1) (2012) 15–20.
URL <http://sites.computer.org/debull/A12mar/apollo.pdf>
- [29] I. K. Koumarelas, A. Naskos, A. Gounaris, Binary theta-joins using mapreduce: Efficiency analysis and improvements, in: K. S. Candan, S. Amer-Yahia, N. Schweikardt, V. Christophides, V. Leroy (Eds.), *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014)*, Athens, Greece, March 28, 2014., Vol. 1133 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014, pp. 6–9.
URL <http://ceur-ws.org/Vol-1133/paper-02.pdf>
- [30] A. Okcan, M. Riedewald, Processing theta-joins using mapreduce, in: T. K. Sellis, R. J. Miller, A. Kementsietsidis, Y. Velegrakis (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, ACM, 2011, pp. 949–960. doi:10.1145/1989323.1989423.
URL <http://doi.acm.org/10.1145/1989323.1989423>
- [31] M. Penar, A. Wilczek, The design of the efficient theta-join in map-reduce environment, in: S. Kozielski, D. Mrozek, P. Kasprowski, B. Malysiak-Mrozek, D. Kostrzewa (Eds.), *Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery - 12th International Conference, BDAS 2016, Ustroń, Poland, May*

31 - June 3, 2016, Proceedings, Vol. 613 of Communications in Computer and Information Science, Springer, 2016, pp. 204–215. doi:10.1007/978-3-319-34099-9_15.

URL https://doi.org/10.1007/978-3-319-34099-9_15

- [32] K. Yan, H. Zhu, Two mrjs for multi-way theta-join in mapreduce, in: M. Pathan, G. Wei, G. Fortino (Eds.), Internet and Distributed Computing Systems - 6th International Conference, IDCS 2013, Hangzhou, China, October 28-30, 2013, Proceedings, Vol. 8223 of Lecture Notes in Computer Science, Springer, 2013, pp. 321–332. doi:10.1007/978-3-642-41428-2_26. URL https://doi.org/10.1007/978-3-642-41428-2_26
- [33] X. Zhang, L. Chen, M. Wang, Efficient multi-way theta-join processing using mapreduce, PVLDB 5 (11) (2012) 1184–1195. URL http://vladb.org/pvladb/vol15/p1184_xiaofeizhang_vldb2012.pdf
- [34] C. Zhang, J. Li, L. Wu, M. Lin, W. Liu, SEJ: an even approach to multi-way theta-joins using mapreduce, in: J. Liu, J. Chen, G. Xu (Eds.), 2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012, IEEE, 2012, pp. 73–80. doi:10.1109/CGC.2012.9. URL <https://doi.org/10.1109/CGC.2012.9>
- [35] C. Zhang, J. Li, L. Wu, M. Lin, W. Liu, SEJ: an even approach to multi-way theta-joins using mapreduce, in: J. Liu, J. Chen, G. Xu (Eds.), 2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012, IEEE, 2012, pp. 73–80. doi:10.1109/CGC.2012.9. URL <https://doi.org/10.1109/CGC.2012.9>
- [36] D. R. Augustyn, L. Warchal, Gpu-accelerated method of query selectivity estimation for non equi-join conditions based on discrete fourier transform, in: N. Bassiliades, M. Ivanovic, M. Kon-Popovska, Y. Manolopoulos, T. Palpanas, G. Trajcevski, A. Vakali (Eds.), New Trends in Database and Information Systems II - Selected papers of the 18th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events, ADBIS 2014 Ohrid, Macedonia, September 7-10, 2014 Proceedings II, Vol. 312 of Advances in Intelligent Systems and Computing, Springer, 2014, pp. 215–227. doi:10.1007/978-3-319-10518-5_17. URL https://doi.org/10.1007/978-3-319-10518-5_17