

Received May 31, 2020, accepted June 10, 2020, date of publication June 15, 2020, date of current version July 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3002610

# Optimization for Multi-Join Queries on the GPU

XUE-XUAN HU<sup>1</sup>, JIAN-QING XI<sup>2</sup>, AND DE-YOU TANG<sup>2</sup>

<sup>1</sup>School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China

<sup>2</sup>School of Software Engineering, South China University of Technology, Guangzhou 510006, China

Corresponding author: Xue-Xuan Hu (wxhxx@163.com)

This work was supported by the Project of the Research on the Intelligent Technology Service Platform and Key Technology under Grant 2016B010124010.

**ABSTRACT** Multi-join queries are important operations in data management systems and data integration systems, and their efficiency has attracted the attention of researchers. In recent years, graphics processing units (GPUs) have developed rapidly and become a powerful tool for parallel computing, providing a new idea for multi-join query optimization. This paper studies the use of GPU technology to optimize multi-join queries and focuses on two points: 1) a multi-phase optimization strategy and 2) optimization methods of each stage. For the first point, we discuss a two-phase optimization strategy on the GPU and prove the effectiveness of this strategy. For the second point, we provide an establishment method of a minimum cost join tree on the GPU, the parallel execution methods of intra-join and inter-join on the GPU, and a strategy of scheduling multiple joins to execute in parallel on the GPU. Experimental results show that the multi-join query optimization proposed in this paper improves the efficiency of multi-join queries, especially in the case of high load and complex join queries, achieving higher throughput than that of previous optimization algorithms.

**INDEX TERMS** GPU, multi-join query, parallel optimization, two-phase optimization strategy.


## I. INTRODUCTION

Multi-join queries aggregate data from multiple tables, or even multiple data sources, to provide material for applications such as data integration, data sharing, and decision support. Especially in this era of big data, aggregating and analyzing data from different data sources for precise prediction and decision-making in various fields, such as business, industry, public utilities and scientific research, can yield highly valued products and services [1]. Therefore, the efficiency of widely used multi-join queries is the guarantee of real-time and effective use of data.

The efficiency of multi-join queries is affected by many factors, such as the number of joins and the amount of data, the join selectivity, the execution order of joins, the storage location of data, the resources, strategies and methods that parallel optimization used. They have become the focus of multi-join query optimization (MJQO), and researchers have done a lot of work for this. For example, built a query execution plan (QEP) with the minimum cost [2]–[7], studied parallel algorithm for basic join methods [8]–[13], researched the parallel optimization in a distributed environment [14],

and explored various parallel scheduling strategies [15], [16], etc., these are optimization of multi-join queries aimed at some factors. Among these optimization methods, parallel execution can increase throughput and improve the efficiency of multi-join queries. The parallel computing power of hardware, the parallel level and scale of the platform are the keys to improving the efficiency. In recent years, graphics processing units (GPUs) have developed into powerful tools for parallel computing because of their advantages of numerous cores and high bandwidth [17], which can be used in various levels of parallelism. Therefore, the application of the GPU to parallel optimization in various fields including databases [18], [19] has become a research hotspot. For example, using the GPU to accelerate the basic SPJ operation of memory database [12], [13], optimize relational queries [20], [21] and concurrent queries [22] on GPU. However, considering all kinds of factors, making full use of the GPU computing power and perceiving its characteristics, the optimization of multi-join for a large number of data has not been systematically studied, which is the focus of this paper.

The architecture of the GPU is different from that of the CPU. First, a large number of transistors integrated on the GPU are used for computation rather than logic control and

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro .

caching, which makes it well suited for performing computations with simple logic and high density. As shown in Fig. 1, the GPU contains a large number of computing cores named stream multiprocessors (SMs), and each SM consists of several execution units named cores; all cores on the same SM execute the same instructions at the same time. Therefore, GPU-based algorithms should adopt a data structure in which each element is independent and equal, as well as simple and with no branch instructions. Second, the capacity of the device memory is limited, and the GPU transmits data to the CPU over a low-bandwidth PCI-E bus. This overhead may offset the efficiency gains yielded by using the GPU. This paper improves transmission efficiency by using Unified Virtual Addressing (UVA) [23] and stream technology. UVA technology can lock the physical memory address so that the GPU can directly read and write data to avoid copying data between the host memory and the device memory. The dotted arrow in Fig. 1 shows the path to read and write data by UVA, and the solid arrow is the path for copying data. The stream is used to manage the execution of operations, the operations within a stream are executed sequentially, and the operations of different streams are executed asynchronously and in parallel. Transmission and computation are executed in different streams, enabling them to be executed asynchronously and in parallel to hide the transmission delays. Finally, compared with the fast computation capability, memory reading and writing are the bottlenecks of parallel computing on the GPU. The storage system of the GPU is a multilevel storage mode that combines shared storage and private storage. Each core has its own register, and all cores in each SM share the shared memory and L1 cache, while the global memory is shared by all threads and cached through a L2 cache. Effective use of these fast on-chip memories and multilevel caches and adoption of a coalesced access mode [17] can greatly reduce the memory access times.

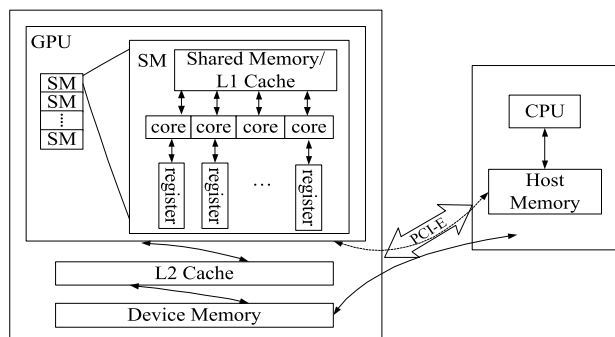


FIGURE 1. The data transmission between the GPU and the CPU.

The unique architecture of the GPU provides huge space for parallel optimization of multi-join queries, but challenges. We cannot transplant the previous optimization methods to the GPU, instead must design new optimization methods suitable for the GPU. The optimization strategies and methods of multi-join queries in this paper, even if they have similar ideas and names to that of the previous, their contents

and implementation are completely different, which are the novelty of this paper. They are shown in the following aspects:

- (1) A two-stage optimization strategy on the GPU and a theoretical proof of its effectiveness.
- (2) The estimation method of the join cost on the GPU.
- (3) The strategies to schedule each join for parallel executing on the GPU, and the main is that of grouping.
- (4) The independent parallel algorithm and the pipelined parallel algorithm based on the stream and the UVA technology, which are on the GPU.

Therefore, we propose a GPU-based Two-Phase Parallel optimization approach (gTPP) for multi-join queries. The contribution of this approach has two aspects: first, providing an overall solution for multi-join query optimization on the GPU; second, designing and implementing a multi-join parallel algorithm on the GPU.

The rest of this paper is organized as follows. In Section II, a brief overview of previous multi-join optimization strategies, methods, and parallel scheduling strategies are given. In Section III, we propose and prove a two-stage optimization strategy on the GPU, design and implement optimization methods of each phase, and a parallel scheduling strategy for multi-join. In Section IV, the efficiency of each strategy and the method proposed in this paper is verified by experiments, and the efficiency of our algorithm is compared with that of other optimization algorithms for multi-join. In Section V, the threats to validity of our optimization are discussed. The future works are summarized and prospected in Section VI.

## II. PRELIMINARY KNOWLEDGE AND RELATED WORK

Multi-join query optimization should not only need to study various optimization methods, but also need to study the strategies that combine these methods for optimization. For example, how to carry out the optimization work? What are the steps? What is the content and purpose of each step?

### A. TWO-PHASE OPTIMIZATION STRATEGY FOR MULTI-JOIN

In a multi-join query, if the optimal solution is searched in all join orders and all parallel optimization methods, it will lead to a huge search space and make the optimization complex. To reduce the search space and optimize quickly and effectively, researchers have proposed multi-phase strategies [16], [24]. For example, Hong and Stonebraker [24] divided the optimization of multi-join queries into two phases. The first phase is to determine the execution sequence of each join and obtain the minimum cost join tree for sequential execution, which is called sequential optimization. The second phase is based on the join tree generated by phase one, considering available resources and according to a certain strategy to schedule multiple joins to execute in parallel, which is called parallel optimization.

### B. SEQUENTIAL OPTIMIZATION

If a multi-join is executed in different orders, then the data amount or join selectivity may be different due to different

joins executed at each step, which will result in different sizes of join results, and different execution costs of their next join which use these results as their join data. Finally, the query efficiency is significantly different [25].

In the sequential optimization phase, the join cost estimation model is first established, and then according to this, the minimum cost execution plan tree is constructed.

According to the cost estimation model, the algorithms for searching the minimum cost join execution plan in the execution plan space include: the enumerated algorithm [2] of exhaustive search; the random search algorithms that search in a subset of the planning space, such as the swarm intelligence algorithm [3] and the simulated annealing algorithm [4]; the heuristic algorithm [5], which uses heuristic rules to search for optimal or near optimal plans; and the genetic algorithm [6], which simulates biological genetic and evolutionary rules to search. The enumeration algorithm can produce the optimal plan, but the complexity increases exponentially with the increase of the join number; the search speed of the heuristic method is the fastest, but it usually only approximates the optimal plan; the time required and the result quality of the random search algorithm and the genetic algorithm are between the two.

Chen et al. [26] proposed a join cost estimation formula of table  $R_i$  join table  $R_j$ , that is  $COST_{MR}$ :

$$COST_{MR}(R_i, R_j) = |R_i \bowtie R_j|$$

Among them,  $(R_i, R_j)$  represents the join between  $R_i$  and  $R_j$ ,  $R_i \bowtie R_j$  represents the result relation of  $(R_i, R_j)$ , and  $|R|$  represents the cardinality of  $R$ . The estimation of the join result cardinality  $|R_i \bowtie R_j|$  is as shown in the appendix in [26]. This formula assumes that the attributes values are uniformly distributed. For the attribute values of the skew distribution, Grady and Puech [27] and Haas and Swami [28] gave corresponding estimation formulas.

Chen et al proposed a heuristic algorithm  $G_{MR}$  based on the above formula for constructing the minimum cost execution plan tree, and confirmed that its efficiency is closer to the efficiency of the optimal plan tree than the minimum cost execution plan tree constructed by other algorithms they proposed.

For a multi-join represented by graph  $G = (V, E)$  (where the node set  $V$  represents the relation table set and the edge set  $E$  represents the join set), the  $G_{MR}$  is shown in Algorithm 1 [26]:

**Algorithm 1**  $G_{MR}$

- (h1) while ( $|V| > 1$ )
- (h2) {choose the join  $R_i \bowtie R_j$  from  $G(V, E)$  such that  $COST_{MR}(R_i, R_j) = \min_{R_p, R_q \in V} \{COST_{MR}(R_p, R_q)\}$ ;
- (h3) perform  $R_i \bowtie R_j$ ;
- (h4) merge  $R_i$  and  $R_j$  to  $R_{\min(i,j)}$ ;
- (h5) update the profile of  $R_{\min(i,j)}$ ;

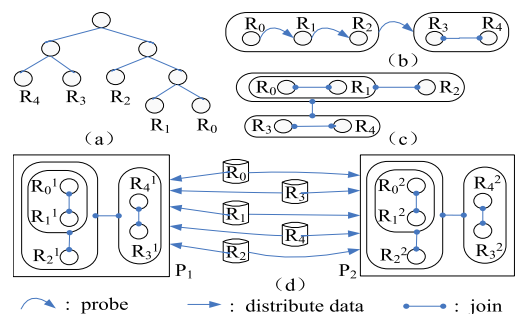
In the above algorithm, first selects the minimum cost join and execute it (h2-h3), and then uses the join result to update

the join graph (h4-h5). Repeat this process until all joins are completed.

**C. PARALLEL OPTIMIZATION**

The parallelism of multi-join can be performed within a single join, namely, intra-join parallelism, or between multiple joins, namely, inter-join parallelism. The four implementation methods of a single join are Non-Indexed Nested-Loops Joins NINLJ, Indexed Nested-Loops Joins INLJ, Sort-Merge Joins SMJ and Hash Joins HJ, and their parallel optimization has been studied extensively on various platforms. For example, Bitton et al. [8] detailed the parallel algorithm of nested-loop joins and sort-merge joins on multi-processors. Using the Gamma cluster, Gerber [9] verified that a hash join algorithm based on join attribute partitioning is both highly parallel and exhibits high-performance. Schneider and DeWitt [10] discussed parallel algorithms for several hash joins and sort-merge joins and compared the performances of these algorithms. With the wide application of the GPU, use of it to optimize the join query has also been studied. He et al. [11], [12] implemented parallel algorithms of four types of single join based on primitives on the GPU. Damos et al. [13] designed a hierarchical, multi-level bulk synchronous parallel algorithm on the GPU that implements relational algebra, such as join, and claimed that the join algorithm is 1.69 times faster than He’s algorithm.

There are three methods for performing inter-join parallelism: (1) pipeline parallelism, which is the parallel execution in pipeline fashion for the dependent joins, dependent join refers to one join result is the input of another join; (2) independent parallelism, which is the parallel execution of different joins whose data are not relevant; and (3) partitioned parallelism, which distributes data to different machines to execute in parallel [29], [30]. Fig. 2 presents the implementation of inter-join parallelism.



**FIGURE 2.** Parallel execution methods of inter-join.

Fig. 2a is a multi-join execution plan tree with the minimum cost. Fig. 2b shows the pipelined execution of 2a.  $R_0$  joins  $R_1, R_2$ , and the join result of  $R_3$  and  $R_4$  in sequence in a pipeline manner without intermediate result. Fig. 2c shows the independent parallel execution of 2a.  $R_0$  joins  $R_1$  first, and then the result is joined with  $R_2$ . Finally, this result is joined with the join result of  $R_3$  and  $R_4$ . Fig. 2d is the partitioned parallel execution of 2a. The data of  $R_0$ - $R_4$  are distributed

to the machines  $P_1$  and  $P_2$  to execute join. The  $R_i^j$  on each machine denotes the part of the data that relation  $R_i$  are distributed to the machine  $P_j$ .

Chen *et al.* [15] performed segmented pipelined hash joins on right-deep trees, significantly improving the efficiency of execution on the right-deep tree. Wilschut *et al.* [16] performed pipeline parallelism and independent parallelism on the minimum cost join tree. Liu and Rundensteiner [31] studied the methods that combine pipelined parallelism, partition parallelism and independent parallelism to achieve better overall efficiency. Koutris *et al.* [32] discuss massively parallel computation of join in large distributed clusters.

The parallel optimization of multi-join on the CPU [8]–[10] and single-join on the GPU [11]–[13], or relational query based on single-join parallel optimization [20]–[22] have been fully studied. However, there is no research on the implementation of a larger degree of parallel optimization of multi-join by using the GPU.

#### D. PARALLEL SCHEDULING STRATEGIES

The strategy of assigning joins to processors to execute in parallel also affects the efficiency of multi-join execution. Wilschut summarizes and compares four parallel scheduling strategies [16]:

(1) Sequential parallel execution (SP), which in turn assigns each join to all processors to execute in parallel but does not implement the parallel execution of inter-join.

(2) Synchronization Execution (SE) [16], which allocates independent subtrees to processors for independent parallel execution, and the number of processors allocated to each join is proportional to the amount of work.

(3) Segmented right-deep execution (RD) [15], [33], [34], which in turn assigns each segment to all processors to execute in pipelined parallel, and the number of processors allocated to each join in a segment is proportional to the amount of work.

(4) Full parallel execution (FP) [35], which simultaneously allocates processors for all joins in the join tree according to the proportion of their amount of work.

Wilschut analyzed and tested the execution efficiency of the use of these four strategies to schedule joins in trees of different shapes. In general, SP is suitable for a multi-join with a large amount of data and simple joins to implement single join parallelism. SE is suitable for a bushy tree to implement independent parallelism. RD is suitable for a right-deep tree to implement pipelined parallelism. FP has good overall performance for all shapes of trees.

### III. GPU-BASED OPTIMIZATION

The optimization of multi-join query on the GPU is different from that on the CPU. We will systematically study the multi-join query optimization on the GPU from the aspects of optimization strategy, sequential optimization, parallel optimization and parallel scheduling.

#### A. TWO-PHASE OPTIMIZATION STRATEGY

Compared with the two-phase optimization strategy on the CPU, the optimization strategy in this paper emphasizes the saturation of tasks performed on the GPU. This is to make full use of its parallel computing power to improve the efficiency of multi-join execution.

##### 1) THE CONTENT OF THE STRATEGY

The multi-join optimization strategy based on the GPU is divided into two phases:

Phase 1, Build a join tree which has the minimum workload;

Phase 2, Group the joins of this join tree to schedule for execution, and make each group has as many joins as possible that can be loaded into the device memory.

The main reason for grouping multiple joins is the limitation of the device memory capacity. The parallel computing power of the GPU and the dependency between joins will not restrict the scheduling of the joins. Because the GPU overload can only make the execution time increase linearly with the increase of workload, will not reduce the efficiency, but make full use of its parallel computing power. The joins with dependency can be optimized by pipeline parallel execution, or if it cannot be optimized, it will not be less efficient than the separate scheduling execution.

The number of joins be scheduled to be executed in parallel in each group is limited by the device memory capacity, which is related to the data transmission method and join execution method adopted in this paper, it will be discussed in Section III-C.

##### 2) THE EFFECTIVENESS OF THE STRATEGY

The execution of the multi-join according to a two-phase optimization strategy, is it sure to improve efficiency? In the past, people just used it, and verified it by experiments at most [24], without giving theoretical proof. The following will prove its effectiveness theoretically.

*Lemma 1:* the effectiveness of the GPU-based multi-join optimization strategy includes two aspects:

(1) The grouping methods that make the GPU full load have higher execution efficiency;

(2) The result of consistent optimization can be obtained by parallel optimization of the minimum cost join tree.

*Prove:* Suppose that multiple joins on a multi-join execution plan tree with total task amount of  $w$  are grouped and scheduled to execute on the GPU. If the processing capacity of the GPU when it is just full load is to complete the join of the workload  $w_c$  using time  $t_c$ , then the execution time  $t_{G_k}$  of the group  $G_k$  with the task amount  $w_{G_k}$  is:

$$t_{G_k} = \begin{cases} \left(\frac{w_{G_k}}{w_c}\right) \times t_c, & w_{G_k} \geq w_c \\ t_c, & w_{G_k} < w_c \end{cases}$$

If it is executed in  $m$  groups, the group with full GPU load is  $G_i$  ( $1 \leq i \leq m_1$ ), and the group with insufficient GPU load



is  $G_j$  ( $m_1 + 1 \leq j \leq m_1 + m_2$ ), and  $m_1 + m_2 = m$ . Then, the total execution time  $t$  of a multi-join on the GPU is:

$$t = \sum_{i=1}^{m_1} \left( \frac{w_{G_i}}{w_c} \right) \times t_c + \sum_{j=m_1+1}^{m_1+m_2} t_c$$

(1) When  $m_2 = 0$ ,  $m_1 = m$ , means each group makes the GPU full load, and its total execution time  $t_f$  is:

$$\begin{aligned} t_f &= \sum_{i=1}^m \left( \frac{w_{G_i}}{w_c} \right) \times t_c \\ &= \left( \frac{w}{w_c} \right) \times t_c \end{aligned}$$

When  $m_2 \neq 0$ , means that there is a group at least makes the GPU load insufficient, and its total execution time  $t_i$  is:

$$\begin{aligned} t_i &= \sum_{i=1}^{m_1} \left( \frac{w_{G_i}}{w_c} \right) \times t_c + \sum_{j=m_1+1}^{m_1+m_2} t_c \\ &> \left[ \sum_{i=1}^{m_1} \left( \frac{w_{G_i}}{w_c} \right) + \sum_{j=m_1+1}^{m_1+m_2} \left( \frac{w_{G_j}}{w_c} \right) \right] \times t_c \quad (w_{G_j} < w_c) \\ &= \left( \frac{w}{w_c} \right) \times t_c \end{aligned}$$

Therefore,  $t_i > t_f$ , that is, the scheduling execution method that each group makes the GPU full load is better than that of the load insufficient.

(2) According to the time  $t_f = \left( \frac{w}{w_c} \right) \times t_c$  of the multi-join execution which makes the GPU full load of each group, we can see that  $t_f$  is the smallest when  $w$  is the smallest. That is, the execution efficiency of the parallel optimization of the minimum cost join tree is certainly higher than that of the parallel optimization of a higher cost join tree. In other words, the optimization in two-phase can achieve consistent optimization results.

In [31], multi-joins are also scheduled and executed by grouping. However, it starts from the join graph. First, select the relation with the largest cardinality as the build table, and then select some connected table with the minimum join cost to form the minimum join cost group. Such an execution plan cannot ensure that the total cost is minimal. Therefore, the method of [31] may not be able to better approach the optimal result.

## B. SEQUENTIAL OPTIMIZATION

As a coprocessor, the GPU needs to read the source data from the CPU and send the join results back to the CPU. Therefore, the execution cost on the GPU contains more data transmission cost than that of on the CPU. It is different from the join cost on the CPU described in [26] and needs to be re-estimated to provide a precise basis for building the execution plan.

### 1) ESTIMATE THE JOIN COST

This paper adopts the hash join as the basic join method because the intrinsic nature of the hash join makes it more suitable for the parallel execution of intra-join, as well as the pipelined and partitioned parallel execution of inter-join.

It is a proven high-performance join method [10]–[12]. The hash join is divided into five tasks: one is to input a relation, namely, a build table; the second is to construct a hash table using the data in this table; the third is to input another relation, namely, a probe table; the fourth is to probe the data of the hash table using the data in the probe table; and the fifth is to output the join result. Therefore, for the multi-join performed by groups, its join cost of each group includes the cost of reading the join data, executing the joins and writing the result data of this group. The total cost is the sum of the costs of each group.

Fig. 3 shows the grouping execution of a multi-join. Where  $R$  denotes source relations,  $I$  denotes join results, the joins within the dotted and solid line belong to the first group and the second group, respectively. In this join tree,  $R_1, R_2, R_4, R_6, I_3$  and  $I_5$  are used to build hash tables,  $R_0, R_3, R_5, I_1, I_2$  and  $I_4$  execute probe, and  $I_2, I_3$ , and  $I_5$  are written by the first group and read by the second group.

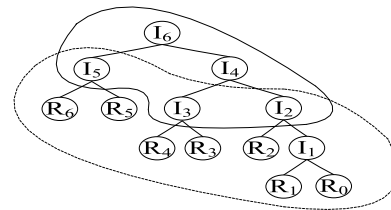


FIGURE 3. The grouping execution of multi-join.

Suppose that a multi-join contains relations  $R_0, \dots, R_n$ , the  $RB = \{R_i, 0 \leq i \leq n\}$  is the set of build tables, and the  $RP = \{R_j, 0 \leq j \leq n\}$  is the set of probe tables. The join results are  $I_1, \dots, I_n$ , each result in the set  $IB = \{I_k, 1 \leq k < n\}$  is used to build hash table, each result in the set  $IP = \{I_q, 1 \leq q < n\}$  is used to probe hash tables, each result in the set  $IR = \{I_r, 1 \leq r < n\}$  needs to be output then input, and the final output is  $I_n$ . Then, the cost  $w_b$  of the build phase, the cost  $w_p$  of the probe phase and the total cost  $w$  are as follows:

$$\begin{aligned} w_b &= (t_{read} + t_{build}) \times \sum_{R_i \in RB} |R_i| + t_{build} \times \sum_{I_k \in IB} |I_k| \\ w_p &= (t_{read} + t_{probe}) \times \sum_{R_j \in RP} |R_j| + t_{probe} \times \sum_{I_q \in IP} |I_q| \\ w &= w_b + w_p + (t_{write} + t_{read}) \times \sum_{I_r \in IR} |I_r| + t_{write} \times |I_n| \end{aligned}$$

Among them,  $|R|$  and  $|I|$  denote the cardinality of source relations and join results, while  $t_{read}$ ,  $t_{write}$ ,  $t_{build}$  and  $t_{probe}$  denote the unit execution time to read data, write data, build a hash table and probe a hash table respectively, that is, the average execution time of each tuple. If the tuple number of all source relations of a multi-join is  $s$ , the total cost  $w$  can be expressed as:

$$\begin{aligned} w &= (t_{read} + t_{build}) \times \sum_{R_i \in RB} |R_i| + t_{build} \times \sum_{I_k \in IB} |I_k| \\ &\quad + (t_{read} + t_{probe}) \times \left( s - \sum_{R_i \in RB} |R_i| \right) + t_{probe} \\ &\quad \times \sum_{I_q \in IP} |I_q| + (t_{write} + t_{read}) \times \sum_{I_r \in IR} |I_r| + t_{write} \\ &\quad \times |I_n| = (t_{read} + t_{probe}) \times s + (t_{build} - t_{probe}) \end{aligned}$$

$$\begin{aligned}
& \times \sum_{R_i \in RB} |R_i| + t_{\text{build}} \times \sum_{I_k \in IB} |I_k| + t_{\text{probe}} \\
& \times \sum_{I_q \in IP} |I_q| + (t_{\text{write}} + t_{\text{read}}) \\
& \times \sum_{I_r \in IR} |I_r| + t_{\text{write}} \times |I_n|
\end{aligned}$$

In the above formulas,  $t_{\text{read}}$  and  $t_{\text{write}}$  are related to the bandwidth and read/write mode (whether or not there is coalesced access),  $t_{\text{build}}$  and  $t_{\text{probe}}$  are related to the parallel algorithm, they are unrelated to the join execution sequences. Similarly, the total number of tuples of all source relations of a multi-join is also unrelated to the join sequence. The join sequence only decides the relations to execute join and the size of intermediate join results, that is,  $|R|$  and  $|I|$ . Thus, if  $(t_{\text{build}} - t_{\text{probe}}) \times \sum_{R_i \in RB} |R_i|$ ,  $\sum_{I_k \in IB} |I_k|$ ,  $\sum_{I_q \in IP} |I_q|$ ,  $\sum_{I_r \in IR} |I_r|$  and  $|I_n|$  take the minimum values, then the total cost  $w$  of the multi-join is minimum. For building hash table is more time-consuming than executing probe ( $t_{\text{build}} > t_{\text{probe}}$ ), the total cost is minimal when the tuple number of the build tables and join results are minimal. Therefore, the estimate formula of the join cost on the GPU is:

$$\text{COST}_{\text{ME}}(R_i, R_j) = |R_i| + |R_i \bowtie R_j| \quad (\text{Formula 1})$$

The above formula gives the minimum estimated value  $\text{COST}_{\text{ME}}$  of the cost of  $R_i$  join  $R_j$ . Among them,  $R_i$  is the build table, which is the table with fewer tuples than the probe table  $R_j$ .

## 2) BUILD THE MINIMUM COST JOIN TREE

A suitable heuristic rule can quickly get the optimal or approximate optimal join plan for complex join queries. Therefore, the heuristic algorithm  $G_{\text{ME}}$  is adopted in this paper, which uses formula 1 as the join cost estimation formula, always prefers to choose those minimum cost joins and finally build the minimum cost join tree. The join tree uses the conventional representation method, namely the left child node represents the build table and the right child node represents the probe table.

For the multi-join represented by graph  $G = (V, E)$ , the heuristic algorithm  $G_{\text{ME}}$  construct its minimum cost join tree  $T$ .  $T = \{n\} = \{(T_i, L_c, R_c)\}$ , it is a set of node  $n$  composed of table identifier  $T_i$ , left pointer  $L_c$  and right pointer  $R_c$ .  $T_i$  gives the identifier of the source relation or the join result relation represented by this node,  $L_c$  points to the build table, and  $R_c$  points to the probe table. The  $G_{\text{ME}}$  for constructing the minimum cost join tree is shown in algorithm 2:

### Algorithm 2 $G_{\text{ME}}$

- (e1) while ( $|V| > 1$ )
- (e2) {choose the join  $R_i \bowtie R_j$  from  $G(V, E)$   
such that  $\text{COST}_{\text{ME}}(R_i, R_j) = \min_{R_p, R_q \in V} \{\text{COST}_{\text{ME}}(R_p, R_q)\}$ ;
- (e3) merge  $R_i$  and  $R_j$  to  $I_{ij}$  in  $G$ ;
- (e4) give the profile of  $I_{ij}$  as  $|R_i \bowtie R_j|$  in  $G$ ;
- (e5) put the node  $(I_{ij}, R_i, R_j)$  into  $T$ ;

For the multi-join of  $|V|$  relations, the  $G_{\text{ME}}$  algorithm performs  $|V| - 1$  times selections according to formula 1 to build a minimum cost join tree (e1). For each selection, at most the cost of  $|E|$  joins need to be calculated (e2). Thus its time complexity is  $O(|V||E|)$ . When the number of joins is large, the minimum cost join tree can still be obtained with reasonable time complexity by the  $G_{\text{ME}}$ .

We use the heuristic rules  $G_{\text{MR}}$  [26] and  $G_{\text{ME}}$  to construct the minimum cost join tree of the join graph of Fig. 4a, and use the  $G_{\text{OPT}}$  [26] scheme to search for its optimal solution. The cardinality of source relations and attribute domains of the join graph is shown in Table 1.

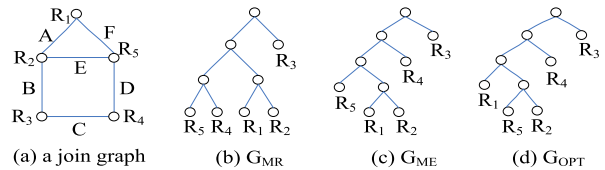


FIGURE 4. Construct the minimum cost join tree by different rules.

TABLE 1. The profile of the join in Fig. 4a.

relation	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>	R <sub>5</sub>	
cardinality( $\times 10^6$ )	255	261	240	295	258	
(a) cardinalities of relation						
attribute	A	B	C	D	E	F
cardinality( $\times 10^6$ )	127	114	129	146	129	117
(b) cardinalities of attributes						

The minimum cost join trees obtained by  $G_{\text{MR}}$ ,  $G_{\text{ME}}$ , and  $G_{\text{OPT}}$  are shown in Fig. 4b, 4c, and 4d, and their total costs are  $2079.7 \times 10^6$ ,  $1037.1 \times 10^6$  and  $1035 \times 10^6$ , respectively. It can be seen that in the multi-join of T-level data scale, the cost of the multi-join execution plan obtained by the  $G_{\text{ME}}$  is more about 2 megabytes than the optimal solution, which is much smaller than the cost gap between the  $G_{\text{MR}}$  and the optimal solution. The result is not an accident. In section IV-C, we will test the query efficiency of join trees derived from these three heuristic rules.

## C. DATA TRANSMISSION METHOD

This paper uses the UVA technology to transfer data. This technology unifies the virtual address of the device memory and the host memory, so that the GPU can read data from the host memory directly without copying to the device memory. On the one hand, the device memory can be effectively use; on the other hand, the GPU can read data at PCI-E rate, which improves the transmission efficiency. Therefore, the UVA technology can be used to transfer data that is read only once and does not need to be modified, and data that is directly written back after GPU processing.

In the hash join, the build tables used to build hash tables and the probe tables used to probe hash tables only need to be read once, they do not need to be written to the device memory

and can be transmitted by the UVA technology, and the final result also can be written back to the host memory directly by the UVA technology. Only the hash tables, which be probed by each data item in the probe tables, needs to be established in the device memory [23]. We will build a compact hash table of the same size as the build table. Therefore, joins that can be scheduled for parallel execution on the GPU are limited by the size of the build tables that the device memory can hold, that is, the total size of all left leaf nodes on these join trees of parallel execution.

**D. THE PARALLEL EXECUTION METHOD FOR SINGLE JOIN**

On the GPU, the method of parallel execution of hash joins is that multiple threads build hash tables, probe data and write results in parallel. Building hash tables and writing results back to memory require concurrent writing multiple data to the target area, which may result in the conflict that multiple threads write to the same location in a bucket. Therefore, it is necessary to calculate the address that each data should be written to before do it. Similar to the split primitives of He et al. [11], we also uses the histogram, which is used for the cardinality sort as the location for writing data.

Fig. 5 shows an example of writing data in parallel using a histogram. In the figure, two threads  $t_0$  and  $t_1$  are each responsible for writing 4 data items of InsertData into buckets in parallel. First, compute the bucket index BucketNo into which each data item should be inserted; next, calculate the histogram, which stores the amount of data (processed by each thread) that should be inserted into buckets  $b_0$  to  $b_3$ ; then, output them to the array Location. The prefix sum of Location is the offset that the thread writes the data into the result array.

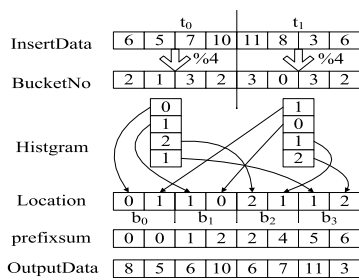


FIGURE 5. An example of the parallel writing of data.

This paper constructs a histogram in cache to improve efficiency. Suppose that the size of the cache in each SM is  $m$ , the thread number of each block is  $tpb$ , and each item in the histogram is 4 bytes; then, each thread can be responsible for writing  $m/(4*tpb)$  data.

**E. THE PARALLEL EXECUTION METHOD FOR MULTIPLE JOINS**

There is no research on methods for implementing multiple joins parallelism on the GPU, and we will discuss to take

advantage of the architecture features and related technologies of the GPU to achieve it in this section.

1) INDEPENDENT PARALLELISM

Those independent and unrelated joins can be simultaneously assigned to the GPU to execute in parallel. There are two ways to implement parallel execution of multiple independent joins on the GPU. One is to execute multiple joins in parallel using one stream. According to the work amount of joins, proportionally allocate the threads to execute each join. The more threads are allocated, the more cores to execute the join, which can achieve load balancing. The other is to execute multiple joins in parallel using multiple streams. Within a stream, each task is scheduled to be executed sequentially. Between streams, the data transmission tasks of each stream are sequentially scheduled to execute on the memory replication engine, and the computation tasks of each stream are sequentially scheduled to execute on kernel execution engines, so that data transmission and kernel functions are executed in parallel to hide transmission delay.

For example, in Fig. 6a,  $n$  tables perform  $n/2$  independent joins. Each join contains five tasks: input the build table (Input  $R_i$ ), construct the hash table ( $B_{(i+1)/2}$ ), input the probe table (Input  $R_{i+1}$ ), execute the probe ( $P_{(i+1)/2}$ ) and output the result (output  $RS_{(i+1)/2}$ ). Fig. 6b, 6c, and 6d show the queuing execution of the  $n/2$  independent joins in the two engines of the GPU. In the figures, the longitudinal axis is the execution time line, and the arrows between two engines indicate the dependence of the execution. Assume that the time to perform each task on a table is 1 unit time.

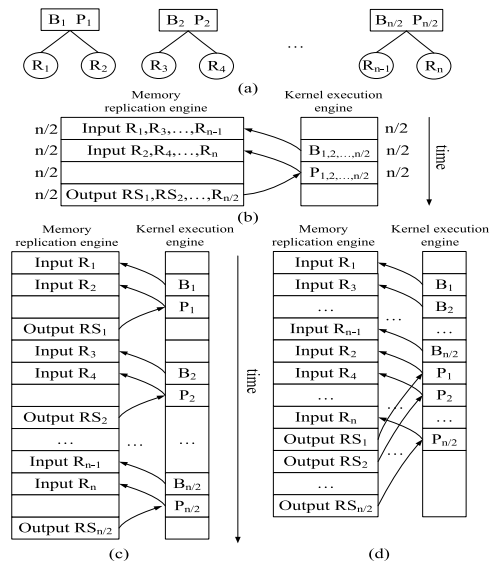


FIGURE 6. Independent parallel execution.

If each task of multiple joins are executed in one kernel function, and multiple streams are not used (the default is one stream). For example, build hash tables of tables  $R_1, R_3, \dots, R_{n-1}$  in the function  $B_{1,2,\dots,n/2}$ ; probe the data of the hash table with the data of tables  $R_2, R_4, \dots, R_n$  in the

function  $P_{1,2,\dots,n/2}$ . Then, the execution in both engines is as shown in Fig. 6b. Before executing  $B_{1,2,\dots,n/2}$ , it is necessary to wait for the finish of the input of all build tables (Input  $R_1, R_3, \dots, R_{n-1}$ ); before outputting results, it is necessary to wait for the finish of all probes ( $P_{1,2,\dots,n/2}$ ). Then, the memory replication engine takes  $4 * n/2 = 2n$  units of time, and the kernel execution engine takes  $3 * n/2$  units of time, for a total of  $2n$  units of time.

If each task of each join is executed in one kernel function and multiple streams are not used, the scheduling execution is as shown in Fig. 6c. The five tasks of each join are respectively scheduled to be executed on two engines in sequence. It can be seen that the output of results needs to wait for the finish of the probe, and the mode of sequential scheduling execution hinders the subsequent input. Therefore, before the output of each join results, the memory replication engine has a waiting time. The construction task of the next join needs to wait for the input of its build table, and then before it executes the construction task of each join, it must wait for the finish of the output of the previous join results and the input of the build table of this join, i.e., the kernel execution engine has two waiting times. In this execution mode, the memory replication engine takes  $4 * n/2 = 2n$  units of time, and the kernel execution engine takes  $4 * n/2 - 1 = 2n - 1$  units of time, totaling  $2n$  units of time.

If  $n/2$  joins are performed using  $n/2$  streams  $S_1, S_2, \dots, S_{n/2}$  and the operations are put into the queues of the streams in a width-first manner, the scheduling execution is as shown in Fig. 6d. The width-first manner refers to preferentially adding the same operation to all streams, not preferentially adding all operations of a same stream. That is, we first add all tasks of the input build table to each stream, next add all construction tasks to each stream, then add all tasks of the input probe table and probe, and finally add all output tasks, as described in algorithm 3.

**Algorithm 3** indepParaJoin

- (i1) Input  $R_1, R_3, \dots, R_{n-1}$  in  $S_1, S_2, \dots, S_{n/2}$ , respectively;
- (i2) Execute  $B_1, B_2, \dots, B_{n/2}$  in  $S_1, S_2, \dots, S_{n/2}$ , respectively;
- (i3) Input  $R_2, R_4, \dots, R_n$  in  $S_1, S_2, \dots, S_{n/2}$ , respectively;
- (i4) Execute  $P_1, P_2, \dots, P_{n/2}$  in  $S_1, S_2, \dots, S_{n/2}$ , respectively;
- (i5) Output  $RS_1, RS_2, \dots, RS_{n/2}$  in  $S_1, S_2, \dots, S_{n/2}$ , respectively;

Under this method, the scheduling execution of the tasks of  $n/2$  joins on both engines is as shown in Fig. 6d. It can be seen that the execution on two engines never has to wait. The memory replication engine takes  $3 * n/2$  units of time, the kernel execution engine takes  $1 + 2*n/2 = n + 1$  units of time, and the total is  $3n/2$  units of time.

If the width-first method is not adopted, but the depth-first method is used, that is, after the execution of all operations of one stream, all operations of the next stream are performed. In this way, even if each join uses a different stream, the scheduling execution is also similar to Fig. 6c, and the operation in the front stream will block the operation in the behind stream. Therefore, only by properly using multiple streams to execute multiple independent joins in parallel, the transmission delay can be hidden, and the execution efficiency of multi-join can be improved.

2) PIPELINED PARALLELISM

Multiple joins with dependencies cannot be executed in parallel independently, can only be executed in a sequential or pipelined fashion. As the multi-join in Fig. 7a, all left child nodes of this right-deep join tree are source relations, hash tables can be constructed in parallel, and pipeline probe can be executed. But the multi-join in Fig. 7b, the left child nodes of this left-deep join tree, except for  $R_0$ , are all intermediate join results, hash tables cannot be constructed in parallel, and the pipeline probe cannot be performed. It is not suitable for pipeline parallel execution, can only be executed in sequence.

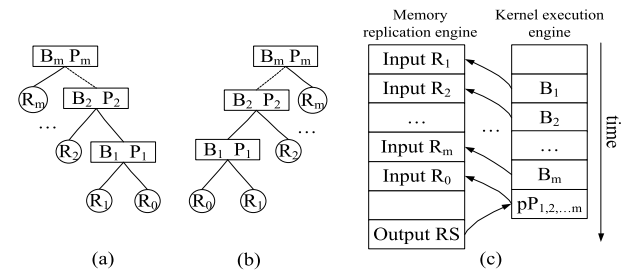


FIGURE 7. Pipelined parallel execution.

Under the pipelined execution mode, the data of the probe tables successively probes the data of the hash tables on the pipeline without generating intermediate results, and its advantages of speed and less space occupation make it competitive in multi-join optimization. The method to perform a hash-based pipelined parallel join on the GPU is:

- (1) Read all construction table data from the host memory to the GPU, and build each hash table to the device memory in parallel;
- (2) Read the probe table data from the host memory, and probe the hash tables on the pipeline in turn;
- (3) Finally, the results are compactly written back to the host memory in parallel.

Although multi-stream can also be used to perform pipelined parallelism, in the single-stream execution mode, all tables be continuous input without blocking and be executed in parallel with the calculation in the kernel execution engine. Therefore, multi-stream execution does not help to hide the transmission delay. For example, for the right-deep tree of  $m$  joins as shown in Fig. 7a, 7c show the execution in the memory replication engine and kernel execution engine without using multiple streams. Among them,  $pP_{1,2,\dots,m}$  show



the pipelined probe to the hash tables built by  $R_1, R_2, \dots, R_m$ , other symbolic meanings are the same as in Fig. 6. The two engines are basically executed in parallel with no other wait except for the necessary wait (the memory replication engine waits for probe results to execute output). Pipeline parallel execution is shown in algorithm 4.

---

**Algorithm 4** pipelineParaJoin
 

---

```
(p1) Input  $R_1, R_2, \dots, R_m$  in S;
(p2) Execute  $B_1, B_2, \dots, B_m$  in S;
(p3) Input  $R_0$  in S;
(p4) Execute  $pP_{1,2, \dots, m}$  in S;
(p5) Output RS in S;
```

---

The pipeline probe algorithm  $pP_{1,2, \dots, m}$  in the above line p4 is shown in algorithm 5.

---

**Algorithm 5** pipelineProbe
 

---

```
(pp1) for each tuple  $R_0[i]$  of  $R_0$  parallel do{
(pp2) do{
(pp3)   for each tuple  $R_j[k]$  in the bucket[hash( $R_0[i].key$ )]
         of  $R_j$ 
(pp4)   { if(  $R_0[i].key == R_j[k].key$  )
(pp5)     matchNum[i]++;}
(pp6)   j = j + 1; // j is the count of the hash table to be
         probed
(pp7)   }while(matchNum[i] > 0 && j ≤ m)}
(pp8) resultNum = sum(matchNum[i]);
```

---

In the lines pp1-pp7, each thread in charge of the probe of each data successively probes the tables on the pipeline. If the table  $R_j$  is probed, and there is no matching data in its corresponding bucket (bucket[hash( $R_0[i].key$ )]), namely  $matchNum[i] = 0$ , then end the probe of the remaining tables on the pipeline; otherwise, continue to probe the next table until all tables have been probed ( $j > m$ ). Because the device memory is allocated from a fixed-size heap [17], and heap size remains unchanged once a kernel function is started, namely all space of the device memory cannot be dynamically allocated when the kernel function is running. Therefore, before writing the result, the size of the result (resultNum) should be accurately calculated, so as to allocate the space for storing the result data in advance.

In this algorithm, the parallel probes to multiple data are synchronized at the end of the pipeline, and the probes in the middle do not need to be paused, which realizes the idea of the pipeline. According to [31], the maximally pipelined processing is not always the most effective. However, this conclusion is based on the join graph. After the minimum cost join tree is determined, pipeline parallel processing for right-deep tree joins can improve its processing efficiency. The longer the pipeline, the more effective, it will be verified in the experimental section.

When the work amount of right-deep tree joins or single joins performed on the GPU cannot meet its computing

power, they can be scheduled for parallel execution at the same time, so that independent parallelism and pipeline parallelism complement each other to further improve the efficiency of multi-join queries. At this time, each right-deep tree join or each single join is executed with a stream respectively, and its algorithm multiStreamParaJoin is the combination of algorithm 3 and algorithm 4.

## F. THE PARALLEL SCHEDULING STRATEGY

Due to the use of the UVA technology, the number of joins be executed in parallel in each group is limited to the size of all build tables that the device memory can accommodate. If a single build table is also larger than the device memory capacity, this table and its probe table can be partitioned into smaller disjoint subtables, so that hash tables of the sub build tables can be built in the device memory, to schedule them to the GPU to execute [36].

The parallel scheduling strategy in this paper first divides the joins in the tree into groups from bottom to top, and then schedules in turn each group to the GPU to execute. The execution of the joins within a group is according to their work amount to allocate the threads proportionately, and nothing is special. The grouping is the key to the scheduling strategy, and it will produce various costs, such as:

(1) Startup cost, which is the time to start the GPU. The greater the number of start up the GPU to execute a multi-join, the higher the start cost. As described by Wilschut *et al.* [16], SP divide the operation of multi-join to the largest number and thus have the highest startup costs. Alternately, FP has the lowest startup cost, while SE and RD live in between.

(2) Collaboration cost, the cost of data transfer between the two schedules. After each scheduling is executed, the result needs to be saved and passed to the group scheduled for the next time. Saving and passing this intermediate footprint reduces the execution efficiency. Similarly, SP has the most intermediate results and the highest cost, FP does not have this cost, and SE and RD are between the two.

(3) Dependency cost, that is the time that the processor which execute upper layer join need to wait for the execution of lower layer join to complete when multiple joins with dependency relations are allocated to multiple processors at the same time. In this paper, when the multi-join shaped as right-deep tree are executed in pipeline parallel fashion, the processor does not need to wait and there is no dependency cost. However, the multi-join shaped as left-deep tree can't perform pipeline parallel, and there is a dependence cost. Therefore, there is no dependency cost for SP and RD scheduling; the dependency cost in SE and FP is related to the number and data volume of subjoins with dependency relations.

In order to reduce scheduling cost and perform multi-join more efficiently, grouping should meet three conditions as far as possible:

(1) The work amount of the joins in each group is greater than the execution power of the GPU to makes maximum use of its computational power, and should first be satisfied.

(2) Minimize the number of groups to reduce the start-up costs and collaboration costs.

(3) Avoid appearing the joins shaped as left subtrees in a group, to remove dependency cost. However, when the memory capacity allows and there is no other unrelated single-join or right-deep tree join, they can also be added to the same group for parallel execution.

Under such a scheduling strategy, when the subjoin in each group is a single join, the strategy is SP; if it is a right-deep tree, the strategy is RD; if it is a bushy tree that consists of right-deep trees and single node(s), the strategy is SE; if the entire scheduling has only one group, then the strategy is FP. However, the groups are usually not a case, which may be a mixture of the four strategies. Our scheduling strategy considers not only the shape of the join tree but also the workload of joins, so it is more suitable for the application requirements.

This paper uses a greedy algorithm to group. In simple terms, it adds as many unrelated subjoins to the same group as possible. The grouping algorithm joinsGrouping is shown as algorithm 6:

---

**Algorithm 6** joinsGrouping
 

---

```
(j1) while(T.nodeNum>1) {
(j2)   searchRTS(T, RTSs);
(j3)   addJoinsToGroup(RTSs, Gs[k]);
(j4)   mergeST(T, RTSs);
(j5)   k=k+1;}
```

---

The algorithm is divided into following steps:

(1) Search all the unrelated right-deep trees and single-join trees at the bottom of the join tree T and put them into the set RTSs (j2).

(2) Select subtrees in the RTSs, and put them into the group Gs[k], so that the total size of all left leaf nodes in the group does not exceed the current remaining capacity rcdm of the device memory, nor smaller than rcdm - th. Here, the final remaining capacity of the device memory is limited to a threshold th, in order to load as many joins as possible to execute (j3).

(3) The subtrees on the join tree T that exists in the group Gs[k] are merged into one node separately (j4).

(4) Perform the next grouping (j5), namely, repeat the above process until there is only one node in T.

The searchRTS for searching independent right-deep trees and single-join trees is shown in Algorithm 7.

The searchRTS function handles three cases when searching in the tree T:

(1) When T is a right-deep tree or a single-join tree, T is placed in the candidate set RTSs (s1-s2) and returned (s10).

(2) When T is a bushy tree, continues to search in its left subtree (s4-s6) or right subtree (s7-s9), respectively.

(3) If T is a leaf node, return (s10).

The above isRT function is shown in algorithm 8.

---

**Algorithm 7** searchRTS
 

---

```
(s1) if( isRT(T) || isST(T) )
(s2) { T→RTSs;}
(s3) else if(!isLeaf(T))
(s4) { if( !isLeaf(T.lc))
(s5)   { T=T.lc;
(s6)     searchRTS (T,RTSs);}
(s7)   if( !isLeaf(T.rc))
(s8)     { T=T.rc;
(s9)       searchRTS (T,RTSs);}
(s10) return;
```

---



---

**Algorithm 8** isRT
 

---

```
((ir1) while ( !isLeaf (T) && isLeaf(T.lc))
(ir2) { T = T.rc; }
(ir3) if (isLeaf (T)) return 1;
(ir4) else return 0;
```

---

The above algorithm determines whether T is a right-deep tree by searching the left subtree (ir1-ir2) of T. If T is not a leaf node, and it has no left subtree, T is a right-deep tree (ir3); otherwise, T is not a right-deep tree (ir4). The functions isST and isLeaf determine whether T is a single-join tree and leaf nodes respectively, and these two algorithms are similar to isRT.

The addJoinsToGroup function that selects joins and adds them to a group is shown in Algorithm 9.

---

**Algorithm 9** addJoinsToGroup
 

---

```
(a1) for each RTSs[i] of RTSs
(a2) { if(RTSs[i].alls<rcdm
(a3)   { RTSs[i]→Gs[k];
(a4)     rcdm -= RTSs[i].alls; } }
(a5) if(rcdm > th)
(a6) { if(!isEmpty(RTSs)
(a7)   { cutTreeOrData(RTSs,rcdm, Gs[k]); }
(a8)   else
(a9)     { mergeST(T,Gs[k]);
(a10)      searchRTS (T,RTSs);
(a11)      addJoinsToGroup(RTSs, Gs[k]); } }
(a12) return;
```

---

The steps of the addJoinsToGroup function to perform grouping are as follows:

(1) Select the subtrees in the RTSs whose total size of all left leaf nodes (alls) is less than the remaining capacity of the device memory (rcdm), and put them into the group Gs[k] (a1-a4).

(2) If rcdm is larger than a threshold th and RTSs is not empty (the total size of all left leaf nodes of each subtree of RTSs is larger than rcdm), then intercept a subtree of RTSs or partition the tables of a single-join, so that the total size of all build tables of this result is smaller than rcdm and larger than

rcdm - th. Add it to the group  $G_s[k]$  (a6-a7), then complete this grouping and return (a12).

(3) If rcdm is larger than th and RTSs is empty, then merge each subtrees that existing in the group  $G_s[k]$  on the join tree T into one node separately. Repeat the operations of searching right-deep trees and single-join trees in this tree, selecting joins to add into the group  $G_s[k]$  (a9-a11).

(4) If rcdm is less than th, then return (a12).

The above functions such as mergeST, isLeaf, isST and cut-TreeOrData have clear functions and simple implementation, and will not be described in detail for limited space.

According to the above functions, the GPU parallel optimization algorithm gTPP can be given, such as algorithm 10.

---

#### Algorithm 10 gTPP

---

- (g1)  $G_{ME}(G, T)$ ;
  - (g2) joinsGrouping(T,  $G_s$ );
  - (g3) for each  $G_s[k]$  in  $G_s$
  - (g4) multiStreamParaJoin( $G_s[k]$ );
- 

The algorithm gTPP first uses the heuristic algorithm  $G_{ME}$  to generate the minimum cost join tree T (g1) from the join graph G, and then uses the grouping function joinsGrouping to group the joins on the join tree T and put them into  $G_s$  (g2), and finally, the subjoins of each group in  $G_s$  are scheduled for parallel execution on the GPU (g3-g4).

## IV. EXPERIMENTS

The experiments will test the efficiency improvement brought by the GPU-based multi-join query optimization method and strategy proposed in this paper. It includes some tests for specific optimization strategies and methods, such as sequential optimization of multi-join, multiple streams parallel optimization of multiple independent joins, pipeline parallel optimization of the right-deep tree join, and also includes the test of overall optimization strategies and methods.

### A. EXPERIMENTAL SETUP

The experiment used Intel Core i7-4770k, quad-core 3.50GHz CPU, 32GB quad-channel DDR3-1600 RAM. The GPU is a NVIDIA GeForce GTX 980 Ti, which uses the Maxwell architecture with a clock frequency of 1076 MHz and 2816 Cores. The GPU is equipped with 6 GB of GDDR5 memory, its bandwidth is 336.5 GB/s, and it provides a 24-KB unified L1 cache and 2-MB L2 cache. It supports the parallel execution of data transmission and kernel function and supports UVA technology. The GPU communicates with the CPU over the PCI-E 3 bus, and the bidirectional bandwidth is up to 32 GB/s. The test environment is the CentOS 6 operating system, and the development tool is the CUDA (Compute Unified Device Architecture) toolkit launched by NVIDIA, version 7.5.

### B. EXPERIMENTAL SCHEME DESIGN

Experiments will examine the effect of multi-join optimization on the execution efficiency of multi-join with different

data amount and join number. For multiple single joins with same amount of data and different join selectivity, their execution processes and efficiencies are the same, but their join result sizes are different, and the efficiencies of their next join using these results as join tables are different. Therefore, the effect of multi-join optimization on the execution efficiency of different join selectivity is equivalent to the effect on the execution efficiency of different data amounts, and a fixed value can be used.

The benchmark TPC-H has 8 tables and 22 queries, most of these queries have no more than 5 joins, which do not meet the test requirements of this paper. The TPC-DS has 25 tables and 99 queries. These queries are missing some specific join operations. For example, lots of complex joins, multiple independent single joins and long right-deep tree joins, etc. Therefore, it is not suitable for testing specific joins optimization methods, but it can be used for testing the overall performance of the algorithm.

The Experiments to test specific optimization strategies and methods (Sections IV-C to IV-E) will use automatically generated joins and tables. Among them, each tuple in tables for single join has two attributes and 8 bytes, and each tuple in tables for multi-joins has 3 attributes and 12 bytes. The values of join attributes are 4-byte integers and uniformly distributed over all tuples in a table, and the join selectivity is 20%.

The experiments to test the overall performance of the algorithm (Section IV-F) will use the TPC-DS benchmark. TPC-DS provides query testing of GB or TB data of a distributed environment, but this paper studies the join optimization on a single GPU. Therefore, the experiment will select those queries that have more join operations (than other operations) and a large number of joins, to perform tests on the data distributed on a single machine.

The experiment uses the oracle database to store data. Before the data is processed on the GPU, it is read into the host memory.

### C. THE SEQUENTIAL OPTIMIZATION PERFORMANCE ON THE GPU

This section test the query efficiency of join trees  $T_{MR}$ ,  $T_{ME}$  and  $T_{OPT}$  generated by  $G_{MR}$ ,  $G_{ME}$  and  $G_{opt}$  under different data amount and join number.

For multi-join queries with data amount of  $10^5$ ,  $10^6$ ,  $10^7$  and  $10^8$ , and join number of 8, Fig. 8 shows the required time for sequential parallel execution according to the join plans generated by the above three methods.

In Fig. 8a, when the amounts of data are  $10^5$ , the query times of the trees  $T_{MR}$ ,  $T_{ME}$  and  $T_{OPT}$  are all a few milliseconds, and the difference is not obvious. With the increase of data amount, the query time gap between tree  $T_{MR}$ ,  $T_{ME}$  and  $T_{OPT}$  is increasing, but the gap between  $T_{ME}$  and  $T_{OPT}$  is not significant. When the data amount is about  $10^8$ , the query time of  $T_{MR}$  is more than 100ms than that of  $T_{OPT}$ , while the query time of  $T_{ME}$  is only about 20ms more than that of  $T_{OPT}$ . It can be seen that the sequential optimization of the multi-join query based on the GPU, the efficiency of the join

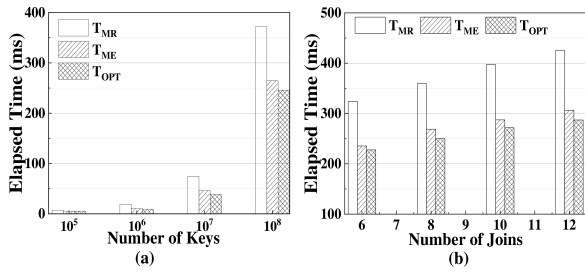


FIGURE 8. The execution time of each join tree under different amounts of data and numbers of joins.

tree generated by our heuristic method is closer to that of the optimal join tree.

For multi-join queries with data amount of  $10^8$ , and join number of 6, 8, 10 and 12, Fig. 8b shows the required time for sequential parallel execution according to the join plans generated by the above three methods. In this figure, the query time of  $T_{ME}$  is much closer to the optimal solution  $T_{OPT}$  than  $T_{MR}$ . As the number of joins increases, the difference between the query times of the trees  $T_{MR}$  and  $T_{OPT}$  is also getting larger, from less than 100ms of 6 joins to more than 100ms of 12 joins, but the difference between  $T_{MR}$  and  $T_{OPT}$  is always about 10ms. It can be seen that the sequential optimization using the heuristic rule of this paper can always obtain the query results efficiently.

#### D. THE EFFECT OF UVA AND STREAM ON THE JOIN EFFICIENCY

In this section, we will test the improvement of the efficiency of multi-join queries caused by using UVA and stream technology.

Fig. 9 compares the multi-join query efficiency with and without UVA technology, measured by the number of execution cycles per tuple on the GTX 980 Ti. The experiments test the efficiency of multi-join queries with data amount of  $10^5$ ,  $10^6$ ,  $10^7$ ,  $10^8$  and  $10^9$ , and join number of 8. We use the  $G_{ME}$  rule to generate the join tree and execute it in sequential parallel. In the figure, using UVA to read a single tuple (HToD/UVA) from the host memory takes 1-1.4 GPU cycles, i.e., its maximum rate is nearly 12 GB/s (12 bytes/tuple, 1076 MHz), copy data (HToD/no\_UVA) instead of using UVA, each tuple takes 3-4 cycles, and the rate

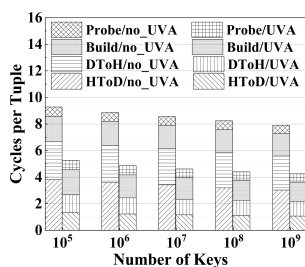


FIGURE 9. The impact of UVA on join efficiency.

is approximately 3-4 GB/s. The rate of using UVA to transmit data is limited only to the bandwidth of PCI-E; compared with the measured value of 12.2 GB/s, this algorithm has greatly improved the transmission efficiency. When results are returned, the write cycles using UVA become longer (DToH/UVA is longer than HToD/UVA). This is due to the irregular writing of the result data, and coalesced access is not possible for the destination address. In addition, the two data transmission modes have no effect on the execution of the build and probe. Moreover, in the process of the build, to prevent multithreaded write to the same location in a bucket, before writing data, it is necessary to calculate the location of each data to be written by using scan (time complexity of  $O(\log(n))$ , where  $n$  is the size of the scan data). This affects the rate of the build, making the average build times of a tuple approximately 1.8 cycles, which is longer than the cycles of the probe process.

Fig. 10 compares the execution efficiency of multi-join using three execution modes: sequential parallel execution, parallel execution using one stream, and parallel execution using multiple streams. The experiment carries out 4-12 independent joins for 8-24 tables, each join table has a data amount of approximately  $5 \times 10^6$ , and the UVA technology is used to transmit data. In the figure, when performing 4 independent joins, the throughput of the sequential parallel execution and single stream execution is approximately 5 GB/s, the efficiency of the two is not much different, and the throughput of asynchronous parallel execution of multiple streams is higher than the two. This is because the asynchronous parallel execution of computation and data transmission can hide the transmission delay. As the number of joins increases, the throughput of the single stream execution increases faster than that of the sequential parallel execution and the throughput of the multiple streams execution increases faster. Under the single stream implementation, the number of times of the kernel function start-up and shutdown is small, but its improved efficiency is very limited. However, the asynchronous parallel execution of multiple streams, which hides the transmission delay and brings about the improvement of efficiency, is worthy of attention. When performing 12 independent joins, the throughput of the sequential parallel execution and single stream execution is less than 6 GB/s, while the throughput of multiple streams parallel execution is greater than 7 GB/s.

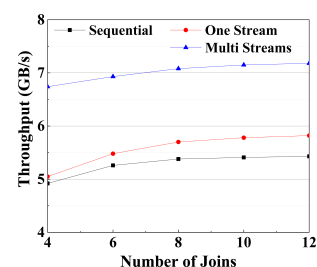


FIGURE 10. The impact of stream on join efficiency.



**E. THE EFFICIENCY OF PIPELINED PARALLELISM**

This section will test the efficiency of pipelined parallel execution on the GPU and its promotion compared to the sequential parallel execution.

Fig. 11a compares the efficiency of sequential parallel and pipelined parallel execution of right-deep tree joins which data amount is  $10^5$ ,  $10^6$ ,  $10^7$ ,  $10^8$  and  $10^9$ , and join number is 8. Similarly, data is transmitted using UVA technology. As shown, the throughput of pipelined parallel execution can be as high as 7 GB/s, and the maximum throughput of sequential parallel execution is about 5.6 GB/s; the throughput of pipelined parallel execution is significantly improved compared to that of the sequential parallel execution. Moreover, when the amount of data is less than  $10^7$ , the throughput of the two methods of execution rises rapidly and then tends toward a stable value after it exceeds. At this point, the GPU runs at full load.

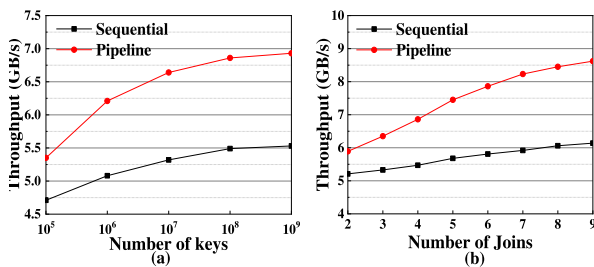


FIGURE 11. The efficiency of pipelined parallel execution.

Figure 11 (b) compares the efficiency of the execution by sequential parallel and pipeline parallel for right-deep tree joins of 3-10 tables. The amount of data for each table is approximately  $5 \times 10^6$ , and UVA technology is also used to transmit data. In the figure, when 2 joins are executed, the throughput of sequential parallel execution is 5.2 GB/s, and the throughput of pipelined parallel execution is close to 6 GB/s, which is slightly higher than that of sequential parallel execution. As the increases of the number of joins, both throughput increase, and the pipelined parallelism increases at a greater rate than the sequential parallelism. When 9 joins are executed, the throughput of pipelined parallel execution is more than 8.5 GB/s, and the throughput of sequential parallel execution is approximately 6 GB/s. As we know regarding the pipelined parallel execution, probing multiple tables continuously without producing intermediate results can save time and increase efficiency. Moreover, the longer the pipeline, the more efficient it is.

**F. THE COMPARISON OF EXECUTION EFFICIENCY WITH OTHER MULTI-JOIN ALGORITHMS**

Because this paper studies the utilization of a single GPU to improve the performance of multi-join, therefore, this experiment will compare the efficiency of the multi-join optimization algorithm between ours and others which on a single machine, rather than the algorithms which on the higher parallel level [14].

In this experiment, the TPC-DS benchmark is used to compare the performance of the gTPP algorithm, the multi-join algorithm proposed in [31] based on a segmented bushy parallel processing strategy (denoted as SBP), and the multi-join algorithm of sequential parallel execution based on a hash single join algorithm in [11] (denoted as gSP). In order to specifically compare the join efficiency, elapsed time only measures the execution time of join operation (including the time of join and the time of data transfer between the host memory and the device memory), but not the execution time of other operations.

Fig. 12a shows the case that the three algorithms execute the sample query Q25 on the database instances with data scales of 1GB-4GB on a single machine.

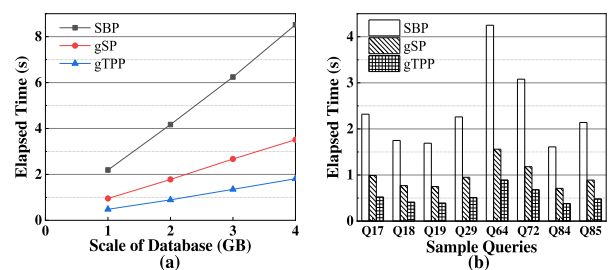


FIGURE 12. The efficiency of the multi-join query optimization algorithm.

When the data scale is 1GB, the time of the SBP algorithm is the longest, about 2 seconds; followed by the gSP, about 1 second; the gTPP is the shortest, less than 0.5 seconds. As the data amount increases, the gap increases significantly. When it increases to 4GB, the time of the SBP algorithm increases to 8.5 seconds, which is about 5 times that of the gTPP. The elapsed time of three algorithms and speedup ratio of the gTPP are shown in Table 2.

TABLE 2. Elapsed time and speedup ratio on various data scale.

data scale	algorithms	SBP	gSP	gTPP	speedup ration
	time(s)/speedup				
1GB		2.19	0.95	0.48	4.56
2GB		4.17	1.78	0.89	4.68
3GB		6.24	2.67	1.35	4.62
4GB		8.52	3.51	1.81	4.71

It can be seen from the above that when the gSP algorithm and the gTPP algorithm perform multi-join with large data amount on the GPU, the efficiency improvement brought by parallel computing is greater than the overhead of data transmission between the GPU and the CPU, so the algorithm have higher efficiency. In particular, compared with the gSP algorithm, the gTPP algorithm achieves not only intra-join parallelism, but also inter-join parallelism, as well as transmission and joins parallelism, which makes its speedup ratio as high as 4.71.

Fig. 12b shows the case that the three algorithms execute the sample queries Q17, Q18, Q19, Q29, Q64, Q72, Q84 and Q85 to the database instances with data scale of 1GB on a single machine. In these queries, for Q18, Q19 and Q84 with 5 or 6 joins, the SBP algorithm takes more than 1.5 seconds, the gSP algorithm takes less than 1 second, and the gTPP algorithm takes less than 0.5 seconds. For Q17, Q29 and Q85 of 10 joins, the SBP algorithm takes 2-2.5 seconds, the gSP algorithm takes about 1 second, and the gTPP algorithm takes about 0.5 seconds. For the Q64 with the largest number of joins, the SBP algorithm takes more than 4 seconds, the gSP algorithm takes about 1.5 seconds, and the gTPP algorithm takes less than 1 second. Table 3 shows the elapsed time of the three algorithms and the speedup ratio of the gTPP algorithm by taking Q17, Q19 and Q64 with join amounts of 10, less than 10 and more than 10 as examples.

**TABLE 3. Elapsed time and speedup ratio of sample queries.**

queries	algorithms time(s)/ speedup	SBP	gSP	gTPP	speedup ratio
Q17		2.32	0.99	0.52	4.46
Q19		1.69	0.75	0.39	4.33
Q64		4.25	1.56	0.89	4.78

In the above table, the speedup ratios of the gTPP algorithm are all above 4, with an average of 4.52. It can be seen that the gTPP algorithm has always maintained a large advantage for such multi-joins with a large amount of data and join.

## V. THREATS TO VALIDITY

This section will discuss the internal and external threats to the validity of multi-join query optimization.

### A. THREATS TO INTERNAL VALIDITY

In this paper, the optimization strategies and methods are used to change the factors that affect multi-join efficiency, thereby improving multi-join efficiency. For example, by determining the join execution order to determine the join executed in each step, the join selectivity of each step, the join result size and the data amount of next join are determined, so as to minimize the total task amount of this multi-join; by using various parallel strategies and join method to achieve highly parallel execution of the multi-join. Therefore, in order to test the effectiveness of the optimization strategies and methods in this paper, for those unrelated factors, the experiment will adopt preset values. Such as bucket capacity and the amount of data executed in parallel, these factors are related to the parallel computing power of the hardware. For a join, the larger the bucket capacity, the less of the number of buckets required, the longer the probe time, and the shorter the build time. Otherwise, the probe time is shorter, and the build time is longer. Through experiments, we select a bucket capacity of 128, which minimizes the sum of the build time

and the probe time, as the preset value of bucket capacity in the experiments. Similarly, the experiment shows that, when the data amount exceeds  $10^6$  data items of 8 bytes, the join time increases linearly with the increase of the data amount. Therefore, in parallel scheduling, we ensure that the work per group is not less than this value.

The experiments use multiple iterations to obtain reliable results. In experiments testing specific optimization methods, we generated the multi-joins of each data amount and each join amount 20 times, to consider different workloads in various applications. In the TPC-DS benchmark test, we also generated database instances of various data scale 10 times. And, each test result is the average of 10 tests.

### B. THREATS TO EXTERNAL VALIDITY

Because this paper is specifically optimized for various join queries, and the TPC-DS benchmark is not dedicated to join testing. In particular, the TPC-DS benchmark is not suitable for testing some optimization methods for specific join. Therefore, for the data and queries in the experiments, we used two methods of random generation and TPC-DS benchmark. For the former, we considered the join queries for various applications and generated multi-joins with different data amounts, number of joins, and join shapes to examine the optimized performance of this paper under different loads. For the TPC-DS benchmark, we generate database instances with different data amounts, and select query samples with a large number of joins to test. This all helps to accurately evaluate the effectiveness of the optimization in this paper in various applications.

## VI. CONCLUSION

The purpose of this paper is to improve the efficiency of multi-join queries by using the parallel computing power of the GPU. Because of the unique architecture features of the GPU, the previous multi-join query optimization method on the CPU cannot be simply transplanted to the GPU. Therefore, we propose a two-phase optimization strategy on the GPU, the execution methods of independent parallelism and pipelined parallelism on the GPU, and utilization of the UVA technology and multiple-stream asynchronous parallel technology on the GPU to speed up the data transmission between the CPU and the GPU, executing data transmission and computing in parallel to hide the transmission delay.

There are still some interesting research directions for future work. First, the data transmission between the host memory and the device memory reduces the efficiency improvement brought by using the GPU, and its impact needs to be reduced as much as possible. Therefore, in addition to using UVA and stream technology, we can also consider using novel data model to effectively express the same information with less data, or using compression technology and column storage to reduce the data to be transmitted.

Second, we also need to further improve the parallelism of multi-join processing. For high-throughput application requirements of multi-join queries with terabytes of data amount, multiple GPUs are often required to perform parallel processing. At this time, a higher level parallel platform can be formed by single machine multiple GPUs, or a cluster of multiple machines and multiple GPUs, or even multiple nodes and multiple GPUs of supercomputers to achieve a greater degree of parallel execution optimization for multi-join queries. On a single machine with shared memory structure, we can use OpenMP supported by CUDA to bind CPU thread and GPU device to control these GPUs to execute their multi-join tasks in parallel; or on multiple machines, in conjunction with MPI to distribute multi-join tasks to multiple GPUs for parallel execution to achieve higher execution effectiveness. The load balancing and cooperative processing of multiple GPUs will be important work. This paper only focuses on multi-join query optimization that perceives the characteristics of the GPU architecture. A higher-level parallel optimization of multi-GPU based on this is a follow-up work.

## REFERENCES

- [1] V. Mayer-Schönberger and K. Cukier, "Now," in *Big Data: A Revolution That Will Transform How we Live, Work, and Think*, 10th ed. New York, USA: Houghton Mifflin Harcourt, 2014, ch. 1, p. 2.
- [2] A. Meister and G. Saake, "Challenges for a GPU-accelerated dynamic programming approach for join-order optimization," in *Proc. 28th GI-Workshop GvDB*, Berlin, Germany, 2016, pp. 86–91.
- [3] A. K. Z. A. Saedi, R. B. Ghazali, and M. B. M. Deris, "An efficient multi join query optimization for DBMS using swarm intelligent approach," in *Proc. 4th World Congr. Inf. Commun. Technol. (WICT)*, Dec. 2014, pp. 113–117.
- [4] Y. E. Ioannidis and E. Wong, "Query optimization by simulated annealing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, San Francisco, California, USA, 1987, pp. 9–22.
- [5] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *VLDB J. Int. J. Very Large Data Bases*, vol. 6, no. 3, pp. 191–208, Aug. 1997.
- [6] M. Jafarnejad and M. Amini, "Multi-join query optimization in bucket-based encrypted databases using an enhanced ant colony optimization algorithm," *Distrib. Parallel Databases*, vol. 36, no. 2, pp. 399–441, Mar. 2018.
- [7] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Houston, TX, USA, 2018, pp. 677–692.
- [8] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson, "Parallel algorithms for the execution of relational database operations," *ACM Trans. Database Syst.*, vol. 8, no. 3, pp. 324–353, Sep. 1983.
- [9] R. H. Gerber, "Data-flow query processing using multiprocessor hash-partitioned algorithms," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 672, Oct. 1986.
- [10] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, Portland, Oregon, USA, 1989, pp. 110–121.
- [11] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2008, pp. 511–524.
- [12] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, p. 21, Dec. 2009.
- [13] G. F. Diamos, H. Wu, A. Lele, and J. Wang, "Efficient relational algebra algorithms and data structures for GPU," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GIT-CERCS-12-01, Feb. 2012.
- [14] Y. Tao, M. Zhou, L. Shi, L. Wei, and Y. Cao, "Optimizing multi-join in cloud environment," in *Proc. IEEE 10th Int. Conf. High Perform. Comput. Commun. IEEE Int. Conf. Embedded Ubiquitous Comput.*, Nov. 2013, pp. 956–963.
- [15] M. S. Chen, M. L. Lo, P. S. Yu, H. C. Young, "Using segmented right-deep trees for the execution of pipelined hash joins," in *Proc. 18th Int. Conf. Very Large Data Bases*, Ottawa, ON, Canada, Aug. 1992, pp. 23–27.
- [16] A. N. Wilschut, J. Flokstra, and P. M. G. Apers, "Parallel evaluation of multi-join queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, San Jose, CA, USA, 1995, pp. 115–126.
- [17] Nvidia. *CUDA C Programming Guide*. version. Accessed: May 10, 2019. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations>
- [18] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units (GPGPU)*, Pittsburgh, PA, USA, 2010, pp. 94–103.
- [19] N. K. Govindaraju and D. Manocha, "Efficient relational database management using graphics processors," in *Proc. 1st Int. Workshop Data Manage. New Hardw. (DAMON)*, Baltimore, MD, USA, 2005, pp. 29–34.
- [20] H. Pirk, S. Manegold, and M. Kersten, "Waste not. Efficient co-processing of relational data," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Chicago, IL, USA, Mar./Apr. 2014, pp. 508–519.
- [21] S. Meraji, B. Schiefer, L. Pham, "Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration," in *Proc. Int. Conf. Manage. Data*, San Francisco, CA, USA, Jun. 2016, pp. 1951–1960.
- [22] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang, "Concurrent analytical query processing with GPUs," *Proc. VLDB Endowment*, vol. 7, no. 11, pp. 1011–1022, Jul. 2014.
- [23] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU join processing revisited," in *Proc. 8th Int. Workshop Data Manage. New Hardw. (DaMoN)*, New York, NY, USA, 2012, pp. 55–62.
- [24] W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in XPRS," *Distrib. Parallel Databases*, vol. 1, no. 1, pp. 9–32, Jan. 1993.
- [25] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 204–215, Nov. 2015.
- [26] M.-S. Chen, P. S. Yu, and K.-L. Wu, "Optimization of parallel execution for multi-join queries," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 3, pp. 416–428, Jun. 1996.
- [27] D. Grady and C. Puech, "On the effect of join operations on relation sizes," *ACM Trans. Database Syst.*, vol. 14, no. 4, pp. 574–603, Dec. 1989.
- [28] P. J. Haas and A. N. Swami, "Sequential sampling procedures for query size estimation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1992, pp. 341–350.
- [29] F. N. Afrati, M. R. Joglekar, C. M. Re, S. Salihoglu, and J. D. Ullman, "GYM: A multiround distributed join algorithm," in *Proc. 20th Int. Conf. Database Theory*, Mar. 2017, p. 4.
- [30] S. Chu, M. Balazinska, and D. Suciu, "From theory to practice: Efficient join query evaluation in a parallel database system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 63–78.
- [31] B. Liu and E. A. Rundensteiner, "Revisiting pipelined parallelism in multi-join query processing," in *Proc. 31st Int. Conf. Very Large Data Base*, Aug. 2005, pp. 829–840.
- [32] P. Koutris, S. Salihoglu, and D. Suciu, "Algorithmic aspects of parallel query processing," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Houston, TX, USA, 2018, pp. 10–15.
- [33] D. A. Schneider and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *Proc. 16th Int. Conf. Very Large Databases* Sep. 1990, pp. 469–480.
- [34] D. A. Schneider and D. J. DeWitt, "Complex query processing in multiprocessor database machines," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, Wisconsin, USA, Tech. Rep. 965, Sep. 1990.
- [35] A. Wilschut, P. Apers, and J. Flokstra, "Parallel query execution in PRISMA/DB," in *Proc. Workshop Parallel Database Syst.*, Noordwijk, The Netherlands, Jun. 1991, pp. 424–433.
- [36] C. KimM T. Kaldewey, V. W. Lee, and E. Sedlar, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," in *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, vol. 2009.



**XUE-XUAN HU** received the M.S. degree from the Department of Computer Science, Central China Normal University, in 2006. She is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, South China University of Technology. Her research interests include high-performance computing, parallel computation on heterogeneous platform, and design and implementation of a new generation database systems.



**DE-YOU TANG** received the M.Sc. degree from Hunan University, in 2004, and the Ph.D. degree from the South China University of Technology (SCUT), in 2007. He is currently a Full Associate Professor with SCUT. His main research interests include database systems, data provenance, parallel computing, and bioinformatics.

...



**JIAN-QING XI** received the M.Sc. degree from the National University of Defense Technology, in 1988, and the Ph.D. degree, in 1992. He is currently a Full Professor with the South China University of Technology. He is also the Head of the Infrastructure Software and Application Construction Technology Laboratory, Guangdong. His main research interests include cloud computing platform, parallel scheduling, and software architecture.