NestGPU: Nested Query Processing on GPU

Sofoklis Floratos*, Mengbai Xiao*, Hao Wang*, Chengxin Guo†,
Yuan Yuan*, Rubao Lee‡, Xiaodong Zhang*
*The Ohio State University, Columbus, OH, USA
†Renmin University of China, Haidian District, China
‡RateUp Inc., Columbus, OH, USA

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00092

***Abstract*—Nested queries are commonly used to express complex use-cases by connecting the output of a subquery as an input to the outer query block. However, their execution is highly time-consuming. Researchers have proposed various algorithms and techniques that unnest subqueries to improve performance. Since this is a customized approach that needs high algorithmic and engineering efforts, it is largely not an open feature in most existing database systems.**

Our approach is general-purpose and GPU-acceleration based, aiming for high performance at a minimum development cost. We look into the major differences between nested and unnested query structures to identify their merits and limits for GPU processing. Furthermore, we focus on the nested approach that is algorithmically simple and rich in parallels, in relatively low space complexity, and generic in program structure. We create a new code generation framework that best fits GPU for the nested method. We also make several critical system optimizations including massive parallel scanning with indexing, effective vectorization to optimize join operations, exploiting cache locality for loops and efficient GPU memory management. We have implemented the proposed solutions in NestGPU, a GPU-based column-store database system that is GPU device independent. We have extensively evaluated and tested the system to show the effectiveness of our proposed methods.

I. Introduction

Nested queries are an important part of SQL for users to express complex use-cases by connecting the output of a subquery (the inner query block) as an input to the outer query block. Among them, a nested query is *correlated* if its subquery refers attributes of the outer query block [1]. An efficient way to execute a correlated subquery is to unnest it into an equivalent flat query. This unnested method has been recognized as an active research field of databases for about 40 years [2], [3], [4], [5], [6], [7], [4], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. However, due to its customized approach, existing database systems cannot unnest arbitrary correlated subqueries by following some general rules and require high engineering efforts to implement unnesting strategies into a database engine.

For an example of such a query presented in Query 1, a conventional method [2] is to replace the subquery with derived tables. By replacing the *table scans* that contain the correlated columns, i.e., `R.col1 = S.col1`, in the nested loop with a `JOIN` in conjunction with a `GROUP BY`, the correlated nested query can be unnested into an unnested one as shown in Query 2. However, if the correlated operator in the subquery is $>$, $<$, $!=$, or others, e.g., the condition is changed from `R.col1 = S.col1` to `R.col1 > S.col1` in the `WHERE` clause of the nested loop, this query cannot be unnested without extending the current SQL standard and introducing new operators [16], [17].

```
SELECT R.col1, R.col2
FROM R
WHERE R.col2 = (
  SELECT min(S.col2)
  FROM S
  WHERE R.col1 = S.col1);
```

Query 1: An example of correlated nested query

```
SELECT R.col1, R.col2
FROM R, (
  SELECT min(S.col2) as t1_min_col2,
         S.col1 as t1_col1
  FROM S
  GROUP BY S.col1) T1
WHERE
  R.col1 = T1.t1_col1 AND
  R.col2 = T1.t1_min_col2;
```

Query 2: The unnested query of Query 1

A more general way to execute a subquery is the nested loop approach that is the final option of most databases when a given nested query cannot be unnested. This nested method directly executes a correlated subquery by iterating all tuples of the correlated columns from the outer query block and executing the subquery as many times as the number of the outer-loop tuples [1]. The method is general-purpose since it is not affected by the type of correlated operators or subquery structures. However, it has a high computational complexity as the correlated columns in the subquery have to be accessed multiple times. In Query 1, the nested method has O(N^2) computational complexity, where N is the size of correlated columns (assuming both `R.col1` and `S.col1` have N tuples). In contrast, the unnested approach in Query 2 has lower computational complexity as it only accesses the correlated columns in the subquery once: when using a `GROUP BY` and a `JOIN` to execute the subquery, the unnested method has O(N) computational complexity in both `JOIN` and `GROUP BY` (assuming the hash join is used). Such a significant complexity gap between the two methods and the optimization opportunities for flat queries have motivated many research and development activities for the unnested

Floratos and Xiao contributed equally to this work as the first author. Yuan is currently employed by Google.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00092

method, even though it is not as general as the nested one and consumes more memory spaces due to the derived tables.

In response to the ending of the Moore's Law, GPU has been widely used to accelerate query processing [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. GPU can accelerate performance over CPU due to its massively parallel architecture and high bandwidth memory. However, nested queries are yet in the scope of existing GPU-based query processing systems due to software complexity, including GPUDB [21], OmniSci [29], and Kinetica [30]. An input query is by default considered as a flat query and unnesting should be done before the execution.

In this paper, we aim to provide a generic method for nested query processing on GPU without unnesting queries. The rationale of our approach is as follows. The high computational complexity of the nested method can be offset by optimized parallel processing on GPU; and its low memory usage can sustain the nested method in the limited GPU memory capacity. This makes the nested method for executing correlated subqueries to become feasible on GPU. However, implementing such a method on GPU is challenging. On CPU, the nested method requires little engineering efforts since recursively executing a subquery in a predicate of an operator (e.g., selection and join) is feasible and efficient, naturally following the same way of processing flat queries. In contrast, implementing a recursive method on GPU is complicated and inefficient. On GPU, the relational operators are usually implemented as GPU kernels [29], [30], [21], [20], [31], [23]. Recursively calling GPU kernels with dynamic parallelism [32] that is the only way for recursive kernel execution on GPU not only incurs significant overhead at runtime [33], [34], [35], [36], but also complicates the system design, where different GPU kernels must be generated in advance for different subqueries or the execution control flow, i.e., analyzing the whole query plan tree, has to be offloaded to GPU. To address this structural issue, we propose a code generation framework for the nested query processing on GPU. Our method generates code to iteratively evaluate a subquery for all correlated tuples before evaluating the predicate containing the subquery, so that we can realize the subquery execution by manipulating pre-implemented and highly optimized GPU kernels in an iterative manner.

In order to execute correlated subqueries efficiently, we design and implement a set of optimizations on GPU: (1) **GPU memory management.** In order to avoid frequent GPU memory allocation and deallocation in relational operators running on GPU, we design and implement a memory pool mechanism and differentiate multiple types of memory requirements inside GPU kernels. (2) **Indexing**. Despite the high throughput of table scan on GPU, building an index and narrowing the scan range can significantly speed up the table scan that are repeatedly invoked in the iteration of the nested method. It is particularly effective when the inner table is large. (3) **Invariant component extraction**. During the code generation, we extract the invariant components inside the nested query. At runtime we execute them only once to avoid redundant computation. (4) **Vectorization**. Inside the nested query block, intermediate data might be too small to fully take advantage of the parallelism of GPU. We implement the query vectorization [37] on GPU by fusing GPU kernels to vectorize operators across multiple nested iterations. This improves GPU occupancy effectively. (5) **Caching for reused outer loop tuples.** With this effort, a large portion of computation can be saved if the correlated column contains duplicate values.

With these optimizations on GPU, the nested method can effectively approach the unnested method in the execution time. However, the gap between the computational complexity still deteriorates the performance of the nested method if the correlated tables become large enough. Thus, we build a cost model to predict the execution time of the nested method. This model can provide a precise estimation of the execution time of the nested method for the query optimizer, so that the optimizer can switch to the unnested method when the nested method is performance unsuitable.

To integrate all these efforts together, we have developed a subquery processing system, called **NestGPU**. In short, GPU accelerated nested query processing in NestGPU has the following unique merits. First, certain nested queries could not be algorithmically unnested [16], [17], thus the only solution on GPU is to use the nested method of NestGPU. Second, due to the high memory capacity requirement, certain algorithmically unnested workloads could not complete the execution on GPU (see Section V). In contrast, they run well with NestGPU by effectively using the GPU device memory. Third, our experiments show that execution performance for representative workloads by NestGPU and the unnested method are very comparable, and in certain cases, NestGPU outperforms the unnested one. Finally and most importantly, NestGPU is general-purpose without complex unnesting efforts, which strongly motivates us to design and implement the nested method on GPU for this critical reason in practice. Our contributions are summarized as follows.

- We look into the major structural differences between the nested and unnested methods, by comparing their merits and limits on GPU. Our study gives insights into the reasons why we are able to deliver high performance in GPU accelerated processing for the nested method.
- We create a new code generation framework and develop a set of GPU optimizations for effective implementations of the nested method. We also develop a cost model to guide subquery processing by selecting the most effective execution path, which ensures to achieve optimal performance for each subquery processing.
- We design and implement a subquery processing system, called NestGPU. Having extensively evaluated and tested the system, we show the effectiveness of the system and accuracy of the cost model. The unique value of NestGPU is its simple and general-purpose structure without complex unnesting efforts, which we believe is a fundamental contribution for subquery processing.

II. Background

A. Nested Queries

In SQL, a query can be nested in the `SELECT`, `FROM`, `WHERE` or `HAVING` clause along with an expression operator, e.g., $<$, $>$, $=$, and others. It can also be part of an `UPDATE` or `DELETE` statement, or a SQL view. Four nesting types are generalized to *type-A*, *type-N*, *type-J* and *type-JA* nesting [2]. For type-A and type-N queries, the nested query block can be executed once and then results are served to the outer query block. The execution of these two types of subqueries is similar to that of flat queries. For type-J and type-JA nesting, the inner query block contains parameters resolved from a relation of the outer block. Type-JA nesting further contains an aggregate function. These two types of nested queries are named as correlated subqueries. Because correlated subqueries are most popular and cannot be executed straightforwardly as type-A and type-N nested queries, they are extensively studied in existing research of the unnested method. In this work, we also focus on correlated subqueries.

B. Nested and Unnested Approaches

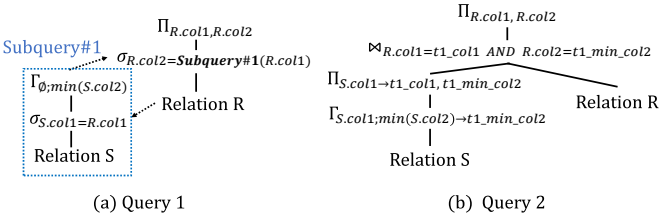


Fig. 1: The query-plan trees of Query 1 and Query 2

Although the nested method is in a simple structure, it executes a correlated subquery in an algorithmically inefficient way. In principle, the subquery must be re-evaluated for each tuple of the referenced column from the outer query block. This leads to high computation complexity. When evaluating such a subquery, the execution engine usually analyzes the query plan trees for the outer query block and the inner query block separately, and uses a syntax like the `Subplan` of PostgreSQL to connect them. For the nest subquery in Query 1, the query plan trees are shown in Figure 1 (a), where the left side represents the nested query block and the right side is the outer query block. The engine evaluates the subquery for each tuple of the column `R.col1` and accesses the relation `S` multiple times.

A common way to reduce the complexity of the nested method is to unnest the nested loops. By performing `JOIN` and `GROUP BY` operations, the correlated subquery can be merged into its upper level query block. One can use the algorithm described in [2] to unnest Query 1 and get the equivalent Query 2. Query 2 is a flat query and its query plan tree is shown in Figure 1 (b). In the execution, a derived table `T1` that is the result of grouping `S.col1` and then aggregating on `S.col2` is first generated. The final result is projected by a join on the relation `R` and the derived table `T1` with the conditions `R.col1 = t1_col1` and `R.col2 = t1_min_col2`. Notice that an unnested query usually requires larger memory space than that of the unnested one because of the generated derived table, the size of which depends on the content of the inner query block.

C. Query Processing on GPU

A general framework of a GPU query-processing engine includes three major components. The first is a set of pre-implemented and optimized relational operators, such as Selection, Join, Aggregation, and many others. Each of these operators is usually dismantled into several GPU kernels (called as primitives), e.g., *scan*, *scatter*, *prefix-sum*, etc. The second is a parser to transform the SQL queries into their query plan trees and apply necessary optimizations. The third is a code generator to translate the query plan trees into programs (called as drive programs) with optimizations on both CPU and GPU. The commonly used operators are highly optimized on GPU [18], [20], [23], [38]. The memory management is also a major concern. To avoid unnecessary data movement, the intermediate tables between operators could be preserved on GPU device memory. A pipelined design is also used to overlap GPU computation and data movement [21], [24], [28].

D. Problem Statement

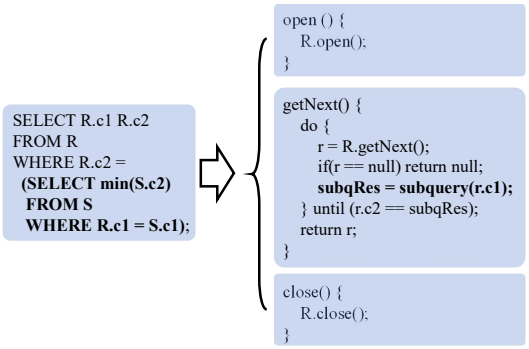


Fig. 2: The nested method on CPU: the pseudo-code to execute Query 1 with the iterator interface method. The subquery is re-evaluated every time when a new tuple is retrieved.

Figure 2 shows a standard pseudo-code to execute Query 1 in the nested method on CPU. The subqueries are recursively evaluated for different correlated values by following the iterator interface method [39]. As shown in the figure, a subquery function, i.e., the aggregate operator `min` in this case, can be directly called at any level of nested queries, because the recursive processing is naturally supported on CPU. Thus, the execution of a correlated subquery on CPU in the nested method is same to that of a flat query. However, to implement such a framework on GPU is cumbersome and inefficient for two reasons. First, as mentioned above, a common practice of GPU query processing engines is to pre-implement the relational operators as GPU kernels while to leave the control flow on CPU. Thus, the execution is in a push mode, i.e., calling GPU kernels of operators from leaves

to the root of the query plan tree. If implementing the recursive execution of subqueries on GPU, one has to implement the control flow completely on GPU, i.e., the traversal of the query plan tree, and also re-implement GPU operators as GPU device functions to allow an operator to call another one, since CPU program invocation from GPU is not allowed. This will lead to an increasingly large, complex, and inefficient GPU kernel. Second, dynamic parallelism [32] is the typical way to support recursive kernel invocation on GPU. However, this approach causes inferior performance due to the kernel launch overhead, the under utilization of GPU resources in sub-kernels, and the divergence branch overhead in upper-level kernels [33], [34], [35], [36]. More importantly, the maximum depth of kernel invocation and the total amount of memory reserved in dynamic parallelism are limited and varied on different generations of GPU [32], which makes the recursive implementation on GPU that uses dynamic parallelism not as general as that on CPU. A new framework rather than the recursive one for the nested method on GPU is needed.

Even if we have such a framework that can support the nested method on GPU, we still need to resolve the following performance issues, due to the nature of repeated evaluations of the subquery. The first one is the kernel launch issue. The nested method may incur high kernel launch overhead, because the kernel launch times grow along with the iterations of the outer block. The second one is the memory management issue. The nested method requires more advanced device memory management. Allocating and deallocating device memory per operator is usual in unnested methods, but it will lead to unacceptable memory management overhead in the nested method, since each operator in a subquery is called as many times as the number of correlated tuples from the outer block. The third one is the inadequate parallelism issue. If the data set to be processed in an operator of a subquery is too small, GPU occupancy is lowered and the overall performance is degraded. The issue may not be a big concern in unnested methods on GPU since the operator in the subquery manipulates the results of the join on the intermediate relation and the relation of the outer block. In contrast, in each iteration of the nested method, the operator in the subquery operates on results of the selection that uses a single tuple of the outer-loop relation as input and produces much less output than that of a join.

In summary, the success of GPU-accelerated processing for correlated subqueries depends on two critical components: (1) a new code generation framework that best fits GPU for the nested method; and (2) a set of effective optimization methods to deliver high performance. With these two components, we have accomplished our goal of high performance and minimum development cost in the query processing engine, which will be presented in the rest of sections.

III. System Design

A. Query Plan Trees

NestGPU introduces a new operator `SUBQ` into the query plan tree to represent a subquery. When the parser of NestGPU identifies a subquery, it will insert a `SUBQ` into the predicate containing the subquery, which will be eventually substituted with an iteration loop of the execution code. Since the subquery itself is a complete query block, it will be parsed into another query plan tree. Meanwhile, because a subquery may include another subquery, the NestGPU parser will finally generate a tree-of-trees structure. In our implementation, all query plan trees except the outermost one are stored into a list, so that any of them can be located by their indices when traversing the tree during the code generation. The operator `SUBQ` has varying operands, including the subquery index and the correlated columns. Figure 3 shows an example of a three-level query plan tree that includes two subqueries. The first subquery reside in the predicate on the right table of the join operator at Level 0, and the second one is in the selection operator at Level 1.

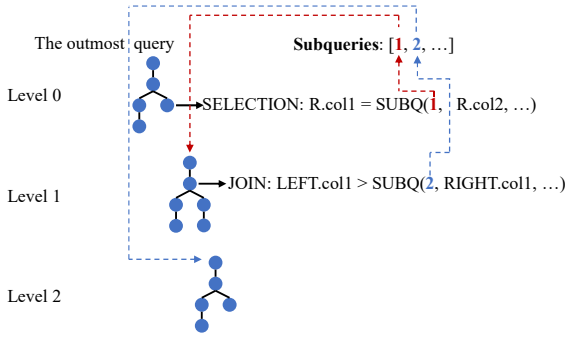


Fig. 3: A query plan tree with two correlated subqueries

B. Code Generation

By traversing the query plan tree, NestGPU generates a drive program that will be further compiled and linked to pre-implemented relational operators. The code generator starts from the outermost query plan tree, from the leaves to the root. For any node, i.e., a query operator, in the query-plan tree having a subquery as an operand of its predicate, the code generator generates code that evaluates the subquery for all correlated values passed from the outer loop and setups the results as a vector. Then the operator containing the subquery is evaluated with the result vector as the input.

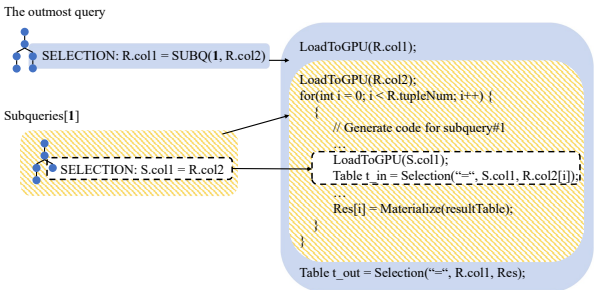


Fig. 4: Generating code for a subquery in Selection

Selection with a subquery. Figure 4 shows the pseudo code generated for a selection operator containing a correlated subquery in the drive program. The regular execution code of a selection is to load all operands, e.g., a constant "10"

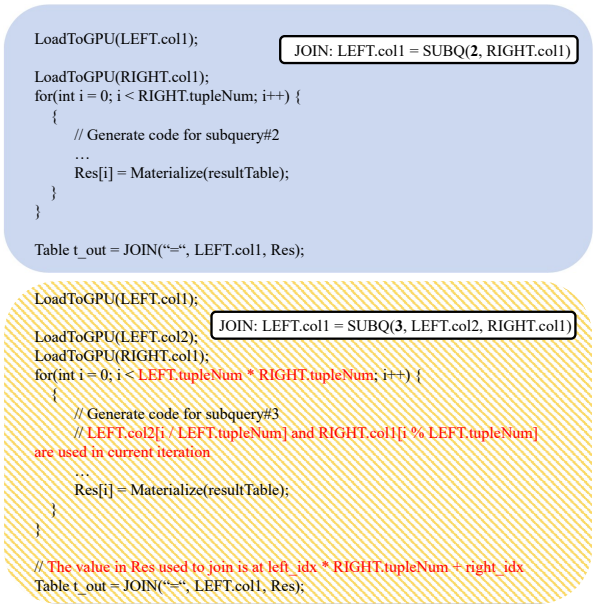


Fig. 5: Generating code for a subquery in Join

or a column `R.col1`, into GPU memory first, then to run GPU kernels over the loaded data to complete the selection operation, and to create a table that can be the input of the next operator. However, if one operand of the selection is a subquery, the code generator needs to generate code that processes the subquery. As shown in Figure 4, the correlated columns, i.e., `R.col2`, are loaded into GPU device memory. The code processing the subquery will be generated in an iterative loop with the size of the input table of the selection operator, i.e., the tuple number of `R`. In the loop, the generated code uses each correlated value, i.e., `R.col2[i]`, to do the selection. The results may be used as the input for the following operator in the subquery, and the final results are stored into the vector according to current iteration number, i.e., `Res[i]`. Following the loop, an upper level selection is to evaluate the predicate containing the subquery, i.e., the selection on `R.col1` and `Res`. In this method, our code generator recursively traverses the query plan tree but generates iterative code for the subquery in the selection operator.

Join with a subquery. The code generation for a join operator that contains a correlated subquery is different from that for a selection operator, because we need to determine the iteration number for the subquery. There are two cases. First, if correlated tuples come from one of the two tables of the join, the iteration number is equal to the corresponding table size. Second, correlated tuples come from both of the join tables. The iteration number is the Cartesian product of two tables. Figure 5 shows the generated code for these two cases, where the difference is marked as red. Besides the code generated for the subquery, the code for the join operator containing the subquery is also different. For the second case, the join operator following the loop needs to use both the left and right tuple IDs to locate the subquery result from `Res`. If the join containing correlated subqueries is a nature join and all its predicates are connected by `AND`, we can optimize the query by first joining two tables with the predicates without correlated subqueries, and then performing a selection over the result table for predicates containing the subqueries. This can effectively reduce the iteration numbers to call subqueris and hence improve the performance.

Subquery in a subquery. Our code generation framework can handle a query with arbitrary levels of correlated subqueries. Figure 6 shows a complete work flow to generate the drive program for the three-level query of Figure 3, in which both subqueries are correlated with its upper-level query block. In Figure 6, the code pieces marked by different colors correspond to the subqueries at different levels. At each level, the generated code evaluates the subquery and stores its results into a vector that will be fed back to the operator at the upper level. In this way, we handle nested subqueries by recursively traversing the query plan tree on CPU and generating the iterative code blocks in a nested loop with the push mode.

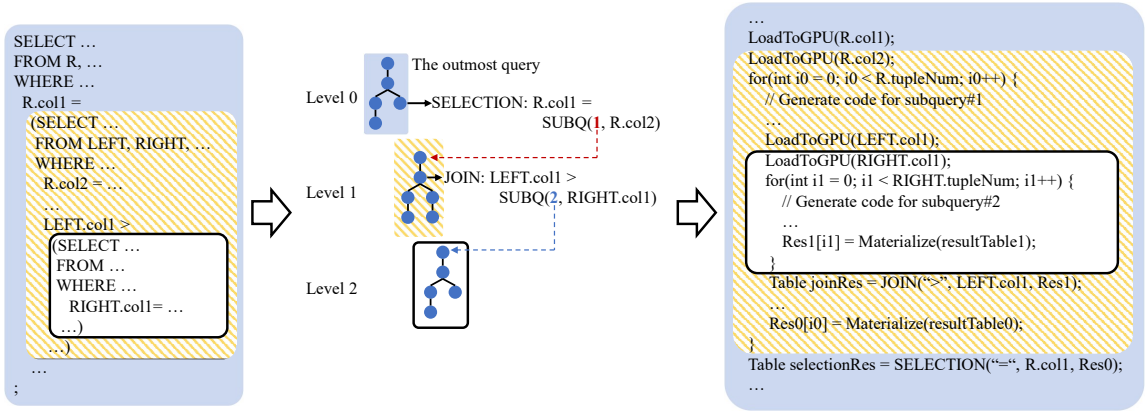
For type-JA correlated subqueries, every subquery evaluation returns a scalar and the result size is fixed. The results can be organized as a vector residing in a space of continuous memory, like any table columns. But type-J correlated subqueries are different because their results have variable lengths. An example is a type-J subquery with an `IN` operator, e.g., `R.col1 IN (...)`. For such subqueries, we use a two-level arrays to store the results of subquery evaluations. The first level is to store the lengths of variable results for each subquery evaluation and the second is to store the results.

C. Memory Management

Due to the iterative execution of subqueries, memory management becomes inefficient if the raw system interfaces like `malloc` are frequently called. Thus, NestGPU build memory pools for efficiently managing memory towards table columns, meta data, intermediate tables, and inter-kernel results.

Memory pools: Memory pools are designed to contain data that needs to be frequently allocated and freed on both the host and device memory. There are three types of data that should be put into memory pools, including meta data, intermediate tables, and inter-kernel results. The meta data is the meta information used at the host side, including the column types, the tuple number of a table, and others. The intermediate tables are composed of column data produced by an operator, which is the input of the next operator. As an operator may consist of several GPU kernels, the inter-kernel results are temporary data on GPU from one kernel to the following one, e.g., a 0-1 vector generated by the prefix-sum kernel will be used as the input of a materialization kernel to determine which tuples should be written to the device memory.

In NestGPU, individual memory pools are used for these three types of data. The memory is linearly allocated in all memory pools, by moving forward the tail pointers. Deallocation means moving backward the tail pointers while setting the tail pointer to the head of a memory pool means releasing all allocated memory in it. For inter-kernel results, NestGPU



SQL query Query plan trees Drive program

Fig. 6: The work flow eventually generating a drive program for a 3-level correlated nested query

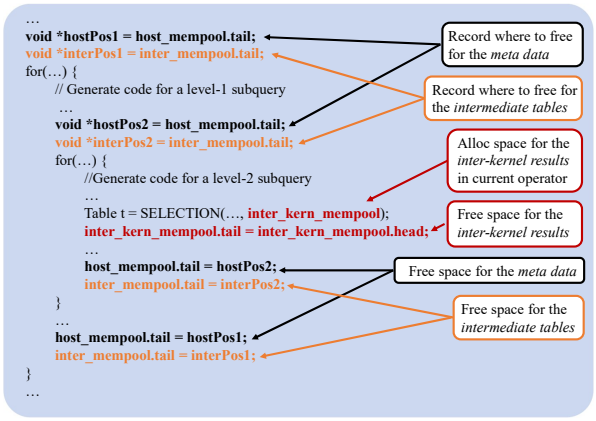


Fig. 7: The use cases of memory pools for three types of data: *meta data*, *intermediate tables*, and *inter-kernel results*

clears the memory pool after the execution of an operator. For meta data and intermediate tables, their tail positions are recorded before the subquery execution. After the execution of an iteration of the subquery, the tail pointer position is recovered so that the space allocated in the previous iteration can be reused. Figure 7 shows how the memory pools are used in a drive program generated for a 3-level subquery.

Preloaded columns: To avoid frequent data loading, we preload the required columns and move them to GPU device memory before the subquery loop is executed and release them after all iterations. If the device can not hold all columns, we separate the device memory into two parts. One holds the preloaded columns and the other is used for on-demand loading. Two principles guide column preloading: (1) the columns used by a more inner-level subquery have a higher priority; and (2) for the columns at the same level, a smaller table has a higher priority as sequential reads are more efficient.

Most of those techniques to improve GPU memory management, e.g., allocating large memory pools, assigning memory regions to different data, and preloading partial data dependent on available memory size, are used in the existing systems [40] [26]. We apply these techniques into the nested structures of subqueries for high performance of memory accesses.

D. Other Optimizations

Indexing: Since the passed-in parameter varies and the nested query block has to be re-evaluated in the loop, building an index over the correlated columns could narrow the scan range in every iteration and improve the performance. In Query 1, the index could be built on `S.col1`. Every time when we scan table `S` with a different passed-in `R.col1` tuple in the predicate `R.col1 = S.col1`, we can locate a valid range of `S.col1` by using the binary search on the index and only scan a portion of table `S`. However, building such an index for correlated columns needs additional time as well as space, e.g., O($N \cdot logN$) time is used to sort a correlated column (N is the size of the column) and O($2N$) more space is required to store the sorted values and their original positions. Thus, we carefully evaluate if the time saved by indexing is offset by the extra time for the sort and if we have enough space to hold the indexing results.

Vectorization: Inside a nested loop, NestGPU evaluates the subquery iteratively, where operators are sequentially evaluated on GPU. If the data size of intermediate results between two operators is too small to fully exploit all cores of GPU, the performance is degraded. Query 3 shows an example that may have such an issue. In the nested part of the query, a selection using the passed-in `T.col1` tuple is executed before joining `T` and `S`. If the selection only produces few tuples as the result, the join may have poor performance as most GPU cores are not used. Thus, we implement the vectorization that fuses GPU kernels of multiple subqueries with different passed-in parameters and evaluates them in a single kernel. This can increase the GPU occupancy and thus improve the overall performance.

Invariant Component Extraction: In the nested query block, we usually find invariant components that won't change with different passed-in parameters. In Query 3, despite that we join `T` and `S` every time when we evaluate the subquery, we can build a hash table for table `S` once and only perform the

```
SELECT R.col1, R.col2
FROM R
WHERE R.col2 = (
  SELECT min(T.col2)
  FROM T, S
  WHERE T.col1 = R.col1 AND
  S.col1 > 0 AND
  T.col3 = S.col3);
```

Query 3: Two selections and a join in a nested query block

probe in the iterative. To generally extract such invariant components from a nested query block, we implement the method in [8] to distinguish which operations could be executed once and which ones are not.

Before generating the iterative code for a subquery, we mark all nodes in its query plan tree as *invariant* except the ones containing correlated columns that are marked as *transient*. Then the transient flag starts to spread upward. If one child of a node is transient, the flag of this node is modified to transient until we reach the root node. After traversing the query plan tree of a subquery and all nodes are marked correctly, NestGPU generates code for the invariant nodes, which are small trees embedded in the original one. Their results are intermediate tables inside the nested query block and will be used later. A specific optimization is for the join operator. If the children nodes of the join are both transient (or invariant), the code of each child should be put inside (or outside) the iteration. If one child of the join is transient and the other is invariant, NestGPU will build the hash table on the invariant child and use the transient child to probe.

Caching: If the passed-in parameter is not a primary key, subqueries would be re-evaluated with duplicate values. Thus, we build a cache for the results evaluated from the subquery, whose key is the combination of the passed-in parameters. This caching technique improves performance if a highly skewed distribution is observed on the passed-in columns.

Although these techniques are used in the existing systems and are not the major contributions of this work, some of them are specifically efficient for the nested query processing. For example, under the nested structure, a single kernel is likely to underutilize GPU resources. We use vectorization to fuse multiple kernels. Also, under the nested structure, if the passed-in tuples from the outer loop have locality, we use caching to avoid redundant computation in the subquery.

IV. Cost Model

The cost model is to predict the execution time of a nested query, involving a number of unique factors on GPU, e.g., the number of GPU cores, the bandwidth of data transfers, the latency of memory accesses, and the number of GPU kernels. These factors are also considered on top of the regular problems in CPU query processing, including join cardinality, filter selectivity estimation, time spent on fetching pages from the disk, etc. To keep the complexity reasonable, we assume nested queries are executed on a single GPU exclusively.

We first estimate the time of a single database operator, e.g., a selection or an aggregation. In NestGPU, these operations are performed by multiple GPU kernels followed by a materialization step. Thus, we breakdown the execution time of a single operator (Top) into several parts as:

$$Top = \sum_{i=1}^{N_k}(K_i * \lceil \frac{D_i}{Th_i} \rceil) + (M * D_r * \sum_{i=1}^{Rc}(Rs_i)) + C. \quad (1)$$

The first part of Eq. (1) estimates the time needed by GPU to execute all primitive kernels. We denote the number of GPU kernels involved as N_k, the data input size as D_i and the number of GPU threads as Th_i. We estimate how many iterations will be performed by each thread by dividing D_i with Th_i. In the equation, K_i is the time needed to perform a single iteration that depends on the instructions executed in the kernel. One method to estimate K_i is actually running the query once and reusing this value for all predictions. An alternative way for K_i estimation is based on the total number of memory transactions [21]. In our evaluation, we use the former method since the optimizations, e.g., using GPU registers for a fast sorting implementation [38], make the global memory transaction-based method less accurate. K_i is actually the parameter that includes the runtime stats. It is different from the previous studies that abstract runtime parameters into the cost model, but we find it is efficient to incorporate runtime information in this way for two reasons. First, we assume that GPU does not process any other workload and thus, K_i is stable and does not rely on the operating system. More importantly, in the nested execution structure that includes many iterations to execute a subquery, it is practical to execute the inner part only once and get all K_is for different operators. The second part of Eq. (1) is on the materialization cost. For most operations, the materialization cost is linear with the size of output data. Thus, we calculate it by multiplying the number of rows in the result D_r, the time needed to materialize a single byte M, and the accumulation of Rs_i that is the size of column i with Rc the number of columns. The third part is a constant time C. It is the time to execute a GPU kernel with an empty input.

Notice that the estimation on a join operation is more complicated than that by Eq. (1). As mentioned in Section III-D, NestGPU may build the hash table once and reuse them multiple times in the nested query block. Thus, we need to estimate the costs of building a hash table and probing a hash table separately. Furthermore, the materialization of a join operation is more complex as it uses two different kernels to materialize the columns from the left table and the right table, respectively. We decompose the cost of a join as the time to build the hash table (Tj_h), to probe the hash table (Tj_p), and to materialize the result (Tj_m):

$$Tj_h = Ht * \lceil \frac{D_i}{Th_i} \rceil, \quad (2)$$

$$Tj_p = P * \lceil \frac{D_i}{Th_i} \rceil, \quad (3)$$

$$Tj_m = (M_l * D_r * \sum_{i=1}^{Rc_l}(Rs_i)) + (M_r * D_r * \sum_{i=1}^{Rc_r}(Rs_i)). \quad (4)$$

In Eq. (2), the execution time of an iteration to build the hash table is denoted as Ht and in Eq. (3), the time of an iteration to probe an element is denoted as P. For Tj_m in Eq. (4), the materialization cost is estimated as the sum of materializing results from the left and right tables. For data from the left table, we estimate the cost by multiplying the time needed to materialize a single byte M_l, the number of rows in the result D_r, and the size of each row from the left table that is the accumulated sum of the column lengths included in the result ($\sum_{i=1}^{Rc_l}(Rs_i)$). Similarly, we can get the estimation time of materializing results from the right table. Notice that we differ M_l and M_r since the materialization time is different on the left and right tables observed in our experiments. This is because the structures of the tables, including the lengths of materialized columns and their strides, will significantly affect the time of memory copy on GPU. The final cost of a join denoted as Tj can be estimated as:

$$Tj = Tj_h + Tj_p + Tj_m + C. \tag{5}$$

We also need to consider those optimizations proposed in the previous section, specifically the cache optimization. If the size of the outer table is S and the cache hit times is C_h, the nested part will be executed $S-C_h$ times. To estimate the time of the nested part, we add the cost of scans and aggregations from Eq. (1) and the cost of joins from Eq. (5). Notice that for the join we only add the Tj_p and Tj_m into the nested computation if the hash table build (with the time Tj_h) can be moved out. Assuming there are N_{Ops} scans and aggregations and N_{Jops} joins in the nested query block, the total cost of the nested part, denoted as N, can be estimated as:

$$N = (S-C_h)*[\sum_{i=1}^{N_{Ops}}(Top_i)+\sum_{i=1}^{N_{Jops}}(Tj_{pi}+Tj_{mi})]+\sum_{i=1}^{N_{Jops}}(Tj_h). \tag{6}$$

NestGPU optimizes memory accesses by using memory pools and preloading table columns. We consider both optimizations and estimate the memory-related cost only once for each operation in the nested block. We denote the memory-related cost for each operation as $Tmem$ and calculate it based on how much data the operation needs to transfer and how much memory to be allocated. We estimate the whole memory-related costs of the nested part, denoted as $Nmem$, as:

$$Nmem = \sum_{i=1}^{N_{Ops}+N_{Jops}}(Tmem_i). \tag{7}$$

Finally, to complete the estimation, we also need to add the cost of the outer block that is denoted as U. We estimate it as the sum of all scans and aggregations, denoted as U_{Ops} and all joins, denoted as U_{Jops} and have:

$$U = \sum_{i=1}^{U_{Ops}}(Top_i) + \sum_{i=1}^{U_{Jops}}(Tj_i) + \sum_{i=1}^{U_{Ops}+U_{Jops}}(Tmem_i). \tag{8}$$

The total cost of a nested query, denoted as Q, is:

$$Q = N + Nmem + U. \tag{9}$$

In our implementation we also estimate the I/O time as a division between the needed data and the available disk bandwidth. The I/O time is not a consideration in the cost model evaluation of this work.

A good estimation of the cost model depends on D_rs in Eq. (1) and Eq. (4) that represent filter selectivity and join cardinality, respectively. Estimating these two parameters have been exploited in both the offline methods and the online solutions, e.g., sampling and indexing [41] and parallel algorithms [42]. We notice that there is a promising solution [43] for heterogeneous computing resources. To accurately estimate these parameters, we break the subquery into pieces as "execution islands" [43], by partitioning the iterations of the subquery. For each partition, we execute several iterations and use the results to estimate D_rs. Under the nested structure, executing several iterations for each partition of the subquery may be expensive. We set a threshold to determine the trade-off between the estimation cost and the prediction accuracy. We expect that this method will highly improve the accuracy of the cost model since D_rs are adaptive in the subquery execution. In the cost model evaluation (Sec. V-C), we make the assumption that D_r is known in advance for the simplicity.

The reasons why we design our cost model for correlated subqueries rather than use an existing one, like GPL [23], are summarized as follows: (1) The execution code generated by NestGPU is structurally different from other GPU-based database systems. For example, GPL is designed for flat queries and focuses on how to improve GPU utilization when concurrently executing multiple kernels. In GPL, TPC-H Q5, Q7, Q8, Q9, and Q14 are evaluated. None of them has correlated subqueries. (2) Furthermore, GPL's cost model might yield different results for correlated subqueries. Under the nested structure, a single kernel is likely to underutilize GPU resources. Thus, we use the vectorization technique in Section III-D to fuse multiple kernels (from many to one); while in order to overlap CPU-GPU data movement and GPU kernel execution in a pipeline mode, GPL's cost model considers how to partition data into chunks and launch concurrent kernels (from one to many). (3) Some optimizations that are not included in GPL are crucial for efficient nested query processing. For example, caching can avoid redundant computations in the iterative structure of subquery processing in NestGPU. It is simple but efficient. Therefore, in our cost model, C_h is added to characterize this optimization.

V. Evaluation

We evaluate NestGPU on different queries and datasets. **Hardware configurations.** We use a server with two Intel Xeon E5-2680 v4 CPUs (28 cores in total) running at 2.40GHz. The server also has 128GB main memory with Linux CentOS 7. The GPU on it is an NVIDIA Tesla V100 with 32GB high bandwidth memory (HBM). We evaluate GPU memory utilization on a desktop machine that has an Intel Core i7-3770K CPU running at 3.50GHz, 32GB main memory, and an NVIDIA GTX 1080 GPU with 8GB GDDR5 memory. **Queries.** We use TPC-H Q2, Q4, and Q17 [44]. These three

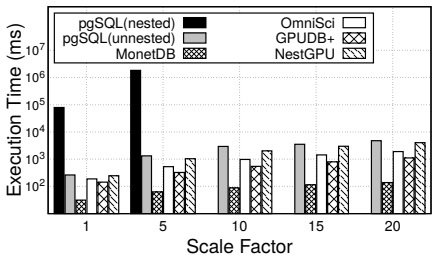


Fig. 8: TPC-H Q2

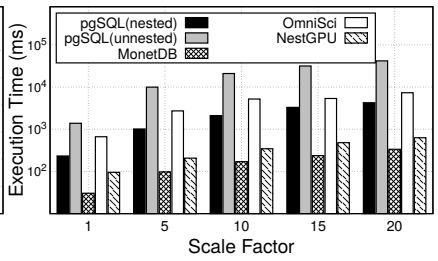


Fig. 9: TPC-H Q4

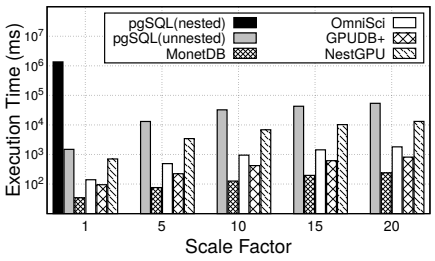


Fig. 10: TPC-H Q17

queries have a type-JA subquery. We also use the modified Q2 queries to evaluate NestGPU under different conditions. **Datasets.** We generate all datasets with the standard TPC-H data generator and change the scale factor from 1 and 20 (up to 100 only for the memory utilization test). **Peer systems.** We use PostgreSQL v12.1 [45] and MonetDB v11.37 [46] on CPU and the open-source software OmniSci [29] and GPUDB [21] on GPU for comparisons. We first enhance GPUDB from only allowing Star Schema queries to accept TPC-H queries. We also enhance it by applying our memory management mechanisms. Thus, we call the enhanced GPUDB as GPUDB+ in the following experiments. All nested queries used in our experiments cannot be automatically unnested by PostgreSQL, OmniSci, and GPUDB+. Thus, if the queries can be unnested by Kim's method [2], we manually rewrite the nested queries to their unnested equivalents. For MonetDB, we measure the nested queries since they can be automatically unnested by MonetDB during execution. We report the execution time in milliseconds where the disk I/O time is excluded.

A. Evaluations on TPC-H Q2, Q4, and Q17

Figures 8, 9, and 10 show the execution time of PostgreSQL, MonetDB, OmniSci, GPUDB+, and NestGPU with TPC-H Q2, Q4, and Q17 on our server node. For TPC-H Q2 shown in Figure 8, PostgreSQL takes ∼13 minutes and ∼31 minutes to run the nested version even with the table scales at 1 and 5. So we will not present its results of the larger scales. For the unnested version of TPC-H Q2, PostgreSQL takes 263 ms to 4763 ms to execute the query. By looking into the runtime information, we find that PostgreSQL only uses a single thread even on a multicore system. On GPU, OmniSci takes 188 ms to 1904 ms and GPUDB+ takes 143 ms to 1109 ms to run the unnested query, while NestGPU uses 246 ms to 4041 ms to run the nested one. OmniSci and GPUDB+ are at most 2.12x and 3.73x faster than NestGPU, respectively. Even though the higher computational complexity, in this case NestGPU has the comparable performance to OmniSci and GPUDB+ on GPU.

Figure 9 shows the performance results on TPC-H Q4. In Q4, the subquery is in an `EXISTS` operator and the execution can be optimized by applying a *semi-join* operator, instead of executing the whole nested loop. OmniSci and GPUDB+ do not use this optimization. Furthermore, GPUDB+ has a memory issue in `GROUP BY` that failed the execution. Thus, we compare NestGPU with other systems than GPUDB+ in this test. The results show that NestGPU can outperform PostgreSQL in all cases. NestGPU is 2.44x, 4.91x, 6.10x, 6.86x, and 6.76x faster than PostgreSQL on the nested Q4 when the scale factors varying from 1 to 20. The performance gains come from the massive parallelism of the semi-join implemented on GPU. The unnested Q4 is executed slower by PostgreSQL compared to the nested one. That is because an additional `GROUP BY` is used to remove redundant tuples for the inner table. NestGPU is 14.55x, 48.17x, 60.65x, 65.65x, and 66.35x faster than PostgreSQL and is 6.97x, 13.17x, 15.11x, 11.18x, and 11.66x faster than OmniSci when they run the unnested Q4 with different scale factors, respectively.

Figure 10 shows the execution time on TPC-H Q17. The structure of Q17 is similar to that of Q2, but Q17 has a much larger inner table, i.e., `LINEITEM`. PostgreSQL is significantly inferior than others when it executes the nested Q17. It takes ∼23 minutes even on the smallest table with the scale factor 1. Thus, we don't present its results with the larger scale factors. In this case, NestGPU is 2.11x, 5.51x, 4.74x, 4.19x, and 4.09x faster than PostgreSQL executing the unnested Q17. However, OmniSci and GPUDB+ that execute the unnested Q17 both have better performance, running up to 7.27x and 16.58x faster than NestGPU even if NestGPU enables all optimizations.

MonetDB is one of the fastest database systems on CPU with optimized query plans for TPC-H queries and hardware-related optimizations. In our evaluation, MonetDB takes 31 ms to 138 ms, 30 ms to 334 ms, and 34 ms to 239 ms to run TPC-H Q2, Q4, and Q17, respectively. It achieves the best performance and its excellent performance comes from the following three parts. First, the data movement between CPU and GPU counts a non-negligible percentage of the execution time in GPU-accelerated systems. For TPC-H Q2, the CPU-GPU data movement consumes up to 19.55% of the execution time of NestGPU. This overhead doesn't exist in MonetDB. Second, the evaluation of MonetDB is carried out on two Intel Xeon E5-2680 v4 CPUs, each of which has 14 cores and 35 MB L3 cache. That is a powerful hardware configuration. Third, most importantly, MonetDB applies unique techniques to optimize query plans, e.g., it can reduce cardinalities in the inner query by pushing down predicates from the outer query. This method can mostly optimize performance of Q2 and Q17.

B. Evaluations on TPC-H Q2 Variants

We further evaluate NestGPU with TPC-H Q2 variants. The variants are all derived from the base query presented in Query 4, which is TPC-H Q2 with an additional predicate

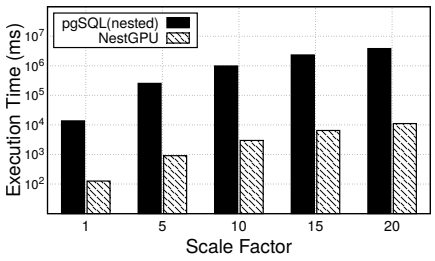


Fig. 11: Query 5 that cannot be unnested

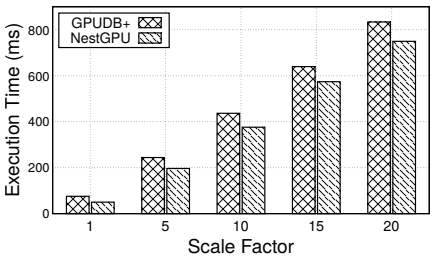


Fig. 12: Query 6 having a smaller outer table

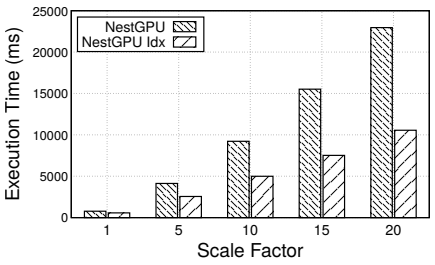


Fig. 13: Query 7 having a larger outer table

```
 1: SELECT
 ...
+13:  AND p_brand = 'Brand#41'
```

Query 4: TPC-H Q2 with an additional predicate in the `WHERE` clause of the outer query block

to reduce the execution time on large tables. First, we change the predicate in Q2 and have Query 5 that cannot be unnested [16], [17] by the commonly used strategies. Second, we make Query 6 that has a smaller outer table. Third, we make Query 7 that has a larger outer table to evaluate the indexing. Finally, we make Query 8 that has a larger inner table to evaluate GPU memory footprint in both the nested and unnested methods.

```
16:  AND ps_supplycost > (
 ...
22:     p_partkey != ps_partkey
```

Query 5: from Query 4 but cannot be unnested

For Query 5 that cannot be unnested, PostgreSQL recursively evaluates the subquery as what NestGPU does. The results are shown in Figure 11. When the scale factor varies from 1 to 20, the speedup of NestGPU over PostgreSQL is from 109x to 359x. In summary, when a nested query cannot be unnested, the nested method is the solution to execute the query, and NestGPU on GPU can achieve two orders of magnitude better performance over PostgreSQL on CPU.

```
+11: AND p_container like '%BAG'
12:  AND p_size = 20
```

Query 6: from Query 4 but having a smaller outer table

We compare NestGPU with GPUDB+ on Query 6 that has a smaller outer table, and the results are shown in Figure 12. For all scale factors, NestGPU outperforms GPUDB+ in execution time. More specifically, the nested query block of Query 6 is only evaluated 116 times because of the added predicate. For such a small outer table, it is more efficient to perform a simple aggregation over a column multiple times than a `GROUP BY` followed by a large `JOIN`. The experiment shows that when the outer table is sufficiently small, NestGPU can provide better performance than the unnested method on GPU. The cost model further provides the quantified information to the query optimizer if the nested method is better.

We evaluate the indexing technique by using Query 7 that has a larger outer table. Figure 13 shows the execution time

```
-13:  AND p_brand = 'Brand#41'
```

Query 7: from Query 4 but having a larger outer table

of NestGPU with and without the indexing technique. Without it, NestGPU completes the query in 772 ms to 22956 ms on different scale factors of datasets; while with it, the execution time is reduced to 570 ms to 10557 ms that even includes the index building time. The experimental results show that for a larger outer table, indexing could lead to better performance.

```
-25: AND r_name != 'ASIA'
```

Query 8: from Query 4 but having a larger inner table

We also check when the unnested method will run out of GPU device memory. We make Query 8 that has a larger inner table and run the test on our desktop machine that has an NVIDIA GTX 1080 GPU with 8GB device memory. In the experiment, we change the scale factors to 20, 40, 60, 80, and 100 to enlarge the memory usage and show the experimental results in Figure 14. At the scale factors of 20, 40, and 60, the performance gap between NestGPU and GPUDB+ ranges from 2x to 3.7x. The unnested method GPUDB+ has higher performance due to its lower computation complexity than that of the nested method NestGPU. However, when the scale factor is beyond 80, GPUDB+ runs out of GPU device memory, while NestGPU can still correctly execute the query.

C. Cost Model Verification

Finally, we evaluate the cost model of NestGPU. We verify the accuracy of three relational operators from Query 4 and then verify the accuracy for the whole nested query. In Figure 15, we denote the real execution time with (R) and the estimated time by the cost model with (E). The error rate between the estimated time and the real execution time is from 0.49% to 17.75%, from 4.03% to 17.48%, and from 0.15% to 7.66% for selection, join, and aggregation, respectively. Figure 16 shows that the results on the whole nested query. The error rate is up to 12.7% with the scale factor 20.

VI. Related work

Optimizing query processing operators on GPU. He et al. [18] implement and optimize the operators, e.g., map, $scatter/gather$, $prefixScan$, etc., on GPU and use them to construct join algorithms. Paul et al.[23] propose a pipelined execution mechanism on GPU, focusing on concurrent kernel

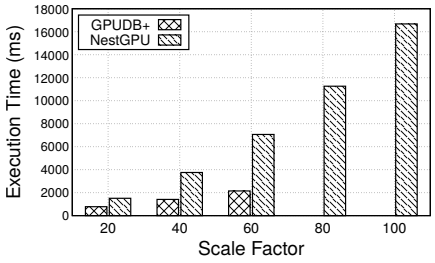


Fig. 14: Query 8 having a larger inner table

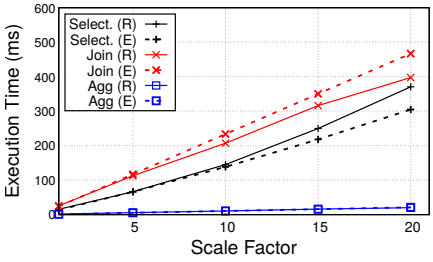


Fig. 15: Cost model verification for ops.

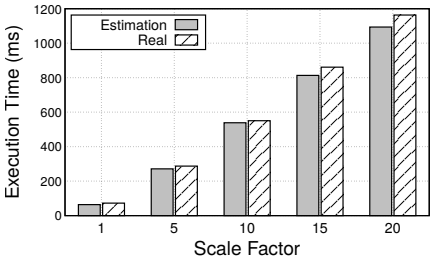


Fig. 16: Cost model verification for Query 4

execution and communication optimization between kernels. Sioulas et al. [26] optimize hash join on GPU by exploiting GPU architecture characteristics, partitioning large dataset, and executing Join in a pipelined manner.

GPU-accelerated database systems. GPUDB [21] is a query processing system on GPU. For a SQL query, GPUDB generates a drive program with pre-implemented GPU kernels and executes the program on GPU. With the performance analysis on multiple queries, GPUDB concludes that the query processing performance on GPU is determined by CPU-GPU data transfer and GPU device memory access times. MapD [22] (rebranded to OmniSci) is another GPU-based query processing system. It implements a memory management system that uses the LRU policy to manage GPU device memory and swap data between CPU and GPU. HorseQC [24] is a pipelined query processing system on GPU. It uses a compound kernel method to fuse multiple kernels into a single kernel so that intermediate data movement between kernels can be significantly reduced. HAPE [25] is a Heterogeneity-conscious Analytical query Processing Engine that uses both multi CPUs and multi GPUs for query execution. It implements the hardware conscious hash join operators [26] and the efficient intra-device execution methods for multiple operators. H^2TAP [27] optimizes HAPE by using the lazy data loading and data transfer sharing so as to increase the CPU-GPU bandwidth usage and to bridge the throughput gap between GPU and PCIe. Hawk [47] is a hardware-adaptive query compiler that can compile queries into OpenCL kernels with low compilation time and without manual tuning. DogQC [28] identifies filter and expansion divergence in GPU-accelerated database systems and proposes the push-down parallelism and lane refill techniques to balance the divergence effects. HetExchange [40] is another query execution engine on heterogeneous hardware and focuses on the multi-CPU-multi-GPU configuration. SEP-Graph [48] is a GPU query processing engine for graph algorithms. It automatically selects the execution path with three pairs of critical parameters, i.e., sync or async mode, push or pull communication mechanism, and topology- or data-driven traversing scheme, with an objective to achieve the shortest execution time in each iteration.

The code generation for subqueries presented in this paper is uniquely different from these GPU systems. First, a common practice in the existing systems is to use CPU to call different pre-implemented GPU kernels in the generated program, as using an GPU kernel to call another GPU kernel is inefficient and impractical (on the contrary, on CPU, the code generation for a subquery operator SUBQ is to use a CPU function to call another CPU function just like the flat query). Thus, a database operator, e.g., Join and Selection, is represented as a single operator and corresponds to one or limited-numbers of GPU kernels in the existing work and also in ours. However, since a subquery can implement any query logic and hence can have arbitrary numbers of combinations of database operators, it is impossible to pre-implement limited-numbers of GPU kernels for a subquery. Therefore, in our work, a crucial design is to generate a `for` loop for a SUBQ, instead of compiling it into GPU kernels in other GPU database systems or into a CPU function in CPU systems. Second, when dealing with a subquery, we analyze the query and apply corresponding optimizations into the query plan tree for different cases, e.g., Selection with a subquery, Join with a subquery, and Subquery in a subquery (in Section III-B). The subquery-specific optimizations are not used in existing systems. In summary, both the structural change in compilation and the specific optimizations for subqueries are new contributions.

Unnested methods. The unnested methods have been extensively researched in the past 40 years [2], [3], [9], [14], [16]. There are two basic strategies for type-JA subqueries [2], [3]. Kim's method does an aggregation on the inner table followed by a join with the outer table [2]. Dayal's method first does an outerjoin on correlated tables and then an aggregation followed by a filtering [3]. Galindo-Legaria et al. [9] develop the `APPLY` operator, identifying that both Kim's and Dayal's methods can be included in the normalization rule. Cao et al. [14] focus on unnesting type-J subqueries. They propose a nested relational algebra based approach to unnest non-aggregate subqueries with hash joins and to treat all subqueries in a uniform manner. Neumann and Kemper [16] show that the efficiency of an unnested strategy depends on the correlated operator of the subquery and the structure of the subquery. They propose dependent joins to unnest arbitrary subqueries with a new operator, i.e., `MAGIC`.

VII. Conclusion

We present NestGPU, a GPU-based database system to accelerate subquery processing under the nested structure. In contrast to a customized approach of the unnested method, our nested method is general-purpose with minimized development efforts in its usage, and also retains high performance. To accomplish our goals, we develop a new code generation

framework to make the nested method best fit on GPU along with a set of optimizations for high performance. With NestGPU, we are able to efficiently process nested queries that cannot be algorithmically unnested and nested queries that can be unnested but failed to run on GPU due to limited GPU memory capacity. In other representative cases, NestGPU achieves comparable performance with that of the unnested method, while outperforms the unnested method if the outer table is relatively small. We also develop a cost model to predict the execution time of the relational operators under the nested structure, aiming to guide the database query optimizer to select the shortest execution method for nested queries.

In the post-Moore's Law era, a highly efficient program is no longer machine independent; but comes from a balance among three critical factors: low algorithm complexity, low data movement, and high parallelism. In this paper, we have made a case to significantly raise the execution performance of the nested method in high complexity by exploiting massive parallelism and device memory locality on GPU. In this way, we accomplish the goal for both general-purpose in software design and high performance in subquery processing.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, IIS-1718450, and CCF-2005884.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979.
[2] W. Kim, "On optimizing an sql-like nested query," *TODS*, vol. 7, no. 3, 1982.
[3] U. Dayal, "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," in *VLDB*, 1987.
[4] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS*, 1998.
[5] M. Muralikrishna, "Improved unnesting algorithms for join aggregate sql queries," in *VLDB*, 1992.
[6] L. Baekgaard and L. Mark, "Incremental computation of nested relational query expressions," *ACM Trans. Database Systs.*, vol. 20, no. 2, 1995.
[7] P. Seshadri, H. Pirahesh, and T. C. Leung, "Complex query decorrelation," in *ICDE*, 1996.
[8] J. Rao and K. A. Ross, "Reusing invariants: A new strategy for correlated queries," in *SIGMOD*, 1998.
[9] C. Galindo-Legaria and M. Joshi, "Orthogonal optimization of subqueries and aggregation," in *SIGMOD*, 2001.
[10] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong, "Winmagic: subquery elimination using window aggregation," in *SIGMOD*, 2003.
[11] M. O. Akinde and M. H. Bohlen, "Efficient computation of subqueries in complex olap," in *ICDE*, 2003.
[12] M. Brantner, N. May, and G. Moerkotte, "Unnesting scalar sql queries in the presence of disjunction," in *ICDE*, 2007.
[13] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, "Execution strategies for sql subqueries," in *SIGMOD*, 2007.
[14] B. Cao and A. Badia, "A nested relational approach to processing sql subqueries," in *SIGMOD*, 2005.
[15] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zait, and C.-C. Lin, "Enhanced subquery optimizations in oracle," in *VLDB*, 2009.
[16] T. Neumann and A. Kemper, "Unnesting arbitrary queries," in *BTW*, 2015.
[17] J. Hölsch, M. Grossniklaus, and M. H. Scholl, "Optimization of nested queries using the nf2 algebra," in *SIGMOD*, 2016.
[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, 2008.
[19] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *DaMoN*, 2012.
[20] S. Zhang, J. He, B. He, and M. Lu, "Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures," *PVLDB*, vol. 6, no. 12, 2013.
[21] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," *PVLDB*, vol. 6, no. 10, 2013.
[22] T. Mostak, "An overview of mapd (massively parallel database)," *White paper. MIT*, 2013.
[23] J. Paul, J. He, and B. He, "Gpl: A gpu-based pipelined query processing engine," in *SIGMOD*, 2016.
[24] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner, "Pipelined query processing in coprocessor environments," in *SIGMOD*, 2018.
[25] P. Chrysogelos, P. Sioulas, and A. Ailamaki, "Hardware-conscious query processing in gpu-accelerated analytical engines," in *CIDR*, 2019.
[26] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on gpus," in *ICDE*, 2019.
[27] A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. Christos Anadiotis, and A. Ailamaki, "Gpu-accelerated data management under the test of time," in *CIDR*, 2020.
[28] H. Funke and J. Teubner, "Data-parallel query processing on non-uniform data," *PVLDB*, vol. 13, no. 6, 2020.
[29] https://www.omnisci.com/platform/downloads.
[30] https://www.kinetica.com/.
[31] S. Breundefined and G. Saake, "Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms," *PVLDB*, vol. 6, no. 12, 2013.
[32] A. Adinets, "Cuda dynamic parallelism api and principles," https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/, 2014.
[33] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *IISWC*, 2014.
[34] G. Chen and X. Shen, "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *MICRO*, 2015.
[35] I. E. Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "Klap: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *MICRO*, 2016.
[36] J. Zhang, A. M. Aji, M. L. Chu, H. Wang, and W.-c. Feng, "Taming irregular applications via advanced dynamic parallelism on gpus," in *CF*, 2018.
[37] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *PVLDB*, vol. 11, no. 13, 2018.
[38] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *ICS*, 2017.
[39] G. Graefe, "Volcano/spl minus/an extensible and parallel query evaluation system," *TKDE*, vol. 6, no. 1, 1994.
[40] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines," *PVLDB*, vol. 12, no. 5, 2019.
[41] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling." in *CIDR*, 2017.
[42] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *EDBT*, 2013.
[43] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *PVLDB*, vol. 10, no. 7, 2017.
[44] "TPC-H benchmark," http://www.tpc.org/tpch, 2018.
[45] PostgreSQL Global Development Group, "PostgreSQL," http://www.postgresql.org, 2017.
[46] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution." in *CIDR*, 2005.
[47] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl, "Generating custom code for efficient query execution on heterogeneous processors," *VLDB Journal*, vol. 27, no. 6, 2018.
[48] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, "Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu," in *PPoPP*, 2019.