Themis: A GPU-accelerated Relational Query Execution Engine

Kijae Hong
POSTECH
kjhong@dblab.postech.ac.kr

Kyoungmin Kim
EPFL
kyoung-min.kim@epfl.ch

Young-Koo Lee
Kyunghee University
yklee@khu.ac.kr

Yang-Sae Moon
Kangwon National University
ysmoon@kangwon.ac.kr

Sourav S Bhowmick
Nanyang Technological University
assourav@ntu.edu.sg

Wook-Shin Han*
GSAI, POSTECH
wshan@dblab.postech.ac.kr

ABSTRACT

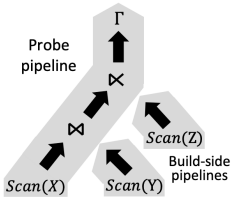
GPU-accelerated relational query execution engines have parallelized the execution of a pipeline, a sequence of operators. For the parallelization, the engines evenly partition the tuples in a table that will be scanned by the pipeline's first operator (a scan), and each thread executes the pipeline for the tuples in a partition. However, this approach leads to load imbalances since an operator returns a varying number of output tuples per input tuple, particularly under non-uniform data distributions such as skewed join key values. The load imbalances are classified into *intra- and inter-warp load imbalances* (intra-WLIs and inter-WLIs) since 1) threads are grouped into warps and 2) every thread in a warp evaluates the same operator for an input tuple concurrently following a *single-instruction-multiple-thread* manner. In contrast, threads in different warps can evaluate different operators concurrently. Although load balancing techniques have been proposed, however, they fail to solve the load imbalances on various workloads. In this paper, we propose a query execution engine, Themis, named after the deity of fairness, which symbolizes balanced workloads within our context. Themis minimizes intra-WLIs and inter-WLIs across various workloads. First, Themis minimizes intra-WLIs by redistributing tuples between the threads in a warp and making the threads evaluate an operator only when all of them hold inputs. Second, Themis mitigates the inter-WLIs by redistributing the tuples of warps with heavy workloads to idle warps. To check whether a warp's workload is heavy, we propose a method to approximate the sizes of warps' workloads. Based on these approximations, Themis adaptively adjusts the threshold for determining a warp's workload as heavy. In a recent benchmark JCC-H, which introduces skewed join key distributions to TPC-H, Themis significantly alleviates the inter-WLIs and intra-WLIs, outperforming the runner-up by up to 379x.

PVLDB Reference Format:
Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S Bhowmick, and Wook-Shin Han. Themis: A GPU-accelerated Relational Query Execution Engine. PVLDB, 18(2): 426-438, 2024.
doi:10.14778/3705829.3705856

PVLDB Artifact Availability:

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705856



(a) Segmented query plan.

Parallel For $t_1 \in X$
For $t_2 \in t_1 \bowtie Y$
For $t_3 \in t_2 \ltimes Z$
$\Gamma(t_3)$

(b) Pseudocode for the probe pipeline.

Figure 1: A plan for a query $(\Gamma((X(A,B) \bowtie Y(B,C)) \ltimes Z(C,A)))$ **segmented into pipelines and the pseudocode for the probe pipeline.** Γ **is a group-by aggregation operator [6], where the aggregation key is the attribute** A**.**

The source code, data, and/or other artifacts have been made available at https://github.com/postechdblab/themis.

1 INTRODUCTION

Today, research endeavors to accelerate relational query processing for analytics on a GPU [1, 9, 12, 23, 34, 46]. One of the primary objectives of such research, which is also our objective, is to fully utilize a massive number of GPU cores for the parallel execution of each *pipeline* on a single GPU [9, 34]. A pipeline is a sequence of non-blocking operators and an optional blocking operator at the end [8, 9, 34, 35]. For a given query plan, the previous studies [8, 9, 34] segment the plan into pipelines and generate a GPU kernel function for each pipeline. To avoid the overhead of materializing intermediate tuples at operator boundaries within a pipeline, the previous studies [8, 9, 34] adopt a tuple-at-a-time approach instead of an operator-at-a-time approach. As an implementation for the tuple-at-a-time approach, the generated kernel function for a pipeline consists of nested-loops to iterate over input tuples for relational operators. Figure 1 shows the query plan and the pseudocode of the generated kernel function for the probe pipeline.

For pipeline execution on a GPU, the previous studies [8, 35] parallelize the outermost loop in a generated kernel function, and this approach leads to load imbalances between threads when distributions of attribute values are non-uniform [9, 34]. For example, threads execute the pseudocode in Figure 1b in parallel. Following the pseudocode, a thread evaluates the join operator for an input tuple t_1 and a table Y, and a non-uniform distribution for join keys results in a varying number of outputs for each input tuple. The variability in output sizes can make threads process different numbers of inputs for the semi-join.

Intra-warp load imbalances (intra-WLIs) cause the suboptimal usage of GPU cores allocated to the threads [9, 18–20, 26, 30, 34]. During pipeline execution, a warp (a group of 32 threads) iteratively evaluates a relational operator for input tuples in a *single-instruction-multiple-threads* manner [8, 9, 34, 35]; every thread in a warp executes the same operator concurrently. Here, we refer to each iteration as a warp iteration [9], and each thread in a warp processes one of the tuples held by the thread in a warp iteration. Due to intra-WLIs, in a warp iteration, some threads hold no input tuple while others hold input tuples. The problem is that the threads that hold no input tuple (i.e., idle threads) must wait for other threads that hold input tuples (i.e., active threads) to evaluate an operator, thereby wasting GPU cores allocated to the idle threads [9].

Inter-warp load imbalances (inter-WLIs) also lead to underutilization of GPU cores [13, 14, 20, 23, 29, 47]. Such imbalances result in warps processing varying numbers of warp iterations during pipeline execution. It indicates that warps finish at different times, and the GPU cores for the warps that finish early will be wasted until the longest-running warp finishes.

Figure 2 illustrates warp iterations of four warps during the execution of the probe pipeline in Figure 1. For simplicity, we assume each warp comprises four threads. Here, the tuples in the table X are evenly distributed to threads, and eight tuples are distributed to the threads in the warp w_1 (ξ_1, ξ_2, ξ_3, and ξ_4). In the warp iteration ①, w_1 evaluates a join for four tuples in the table X. After that, in the next iteration ②, ξ_1 and ξ_3 each hold three and two input tuples, respectively, while ξ_2 and ξ_4 hold no tuple and become idle. The example also shows the inter-WLI where w_3 processes more than 10^8 warp iterations for a large number of join outputs Ⓐ while w_4 finishes early because there is no join output Ⓑ. Here, the GPU cores for w_4 will be wasted until w_3 finishes.

To solve intra-WLIs and inter-WLIs, previous studies [9, 23, 34] redistributed tuples between threads through buffers. However, retrieving and storing tuples in buffers require expensive reads, writes, and synchronization costs, especially when the buffers are located in global memory (GMEM) [34]. The buffers in GMEM are used for the tuple redistribution between threads in different warps (i.e., inter-warp redistribution), while the intra-warp redistribution can be performed through buffers in registers [9]. We observe that the previous studies [23, 34] make a warp access to a buffer in GMEM for each warp iteration in the worst case. Moreover, since all warps share a single buffer in GMEM, severe contention issues arise. We demonstrate in Section 8 that these costs slow down query execution by up to 300 times.

While state-of-the-arts, DogQC [9] and Pyper [34], attempted to tackle intra-WLIs, they did not propose inter-warp load balancing (inter-WLB) methods and just evenly distributed the tuples in the table to be scanned among the warps. As a baseline for the inter-WLB, we employ a *fixed-granularity work-sharing* (FWS) approach [23]. In FWS, a warp checks the heaviness of its workload using a fixed threshold for the output size of an operator for an input, and it redistributes its workload to other warps if its workload is heavy. However, it fails to solve inter-WLIs across various workloads due to its fixed granularity as we shall explain in Section 2.3.

DogQC and Pyper also fail to solve the intra-WLI effectively. Specifically, DogQC proposed two intra-warp load balancing (intra-WLB) methods, push-down parallelism (PP) and lane-refill (LR).

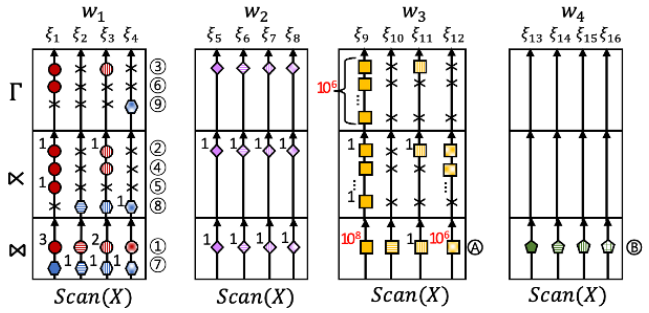
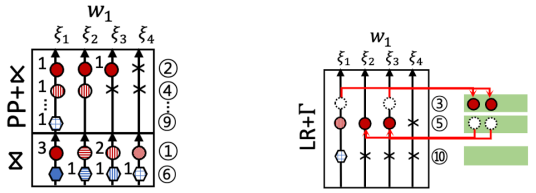


Figure 2: Parallel execution of the probe pipeline in Figure 1 by four warps (w_1-w_4). Horizontally aligned small polygons represent input tuples for an operator in a warp iteration. Each '×' means the absence of an input tuple for a thread. The left-side number on an input tuple is the output size of an operator for the input. If there is no output, we omit the number. Here, the output tuples have the same shape, color, and pattern as the input for clarity. Following the arrows, the outputs of an operator become input tuples for the operator located above. The circled numbers represent the execution order of w_1's warp iterations.

Since a join may generate a varying number of outputs per input, PP tried to solve such an intra-WLI by pushing the parallelism down from inputs to outputs of the join [9]. That is, instead of letting each thread take the join outputs of an input it processes, PP lets all threads in a warp evenly take the join outputs of a single input tuple at a time. However, PP fails to solve intra-WLIs if the join output size of an input is less than the number of threads in a warp (*wSize*). In such cases, some threads in a warp become idle when evaluating the succeeding operator of the join. This case occurs frequently if a join key distribution is skewed. For example, in Figure 3a, w_1 distributes the three join outputs marked as red to ξ_1, ξ_2, and ξ_3. However, ξ_4 becomes idle in the warp iteration ②.

LR attempts to solve intra-WLIs by buffering input tuples in a warp iteration where the number of active threads is less than a fixed threshold (e.g., *wSize*). In the warp iteration, a warp flushes the inputs into a buffer and proceeds to the next warp iteration. If flushed tuples become sufficient to fill all the idle threads in a warp iteration, the idle threads are refilled to remove the intra-WLI for this iteration. For example, in Figure 3b, the inputs in the warp iteration ③ are flushed to a buffer, and the idle threads in the warp iteration ⑤ are covered by the flushed tuples.

However, LR cannot solve the intra-WLI in a warp iteration to evaluate an operator *op* if there is no remaining input for the preceding operator (i.e., the producer) that produces inputs for *op*. When a warp buffer tuples in a warp iteration, the warp evaluates the producer of *op* in the next warp iteration if there is no remaining tuple generated by the producer. For example, in Figure 3b, after the warp iteration ③, the warp w_1 evaluates the semi-join in the warp iteration ④ to generate inputs for the group-by aggregation. However, if there is no input for the producer, instead of evaluating preceding operators of the producer, the warp evaluates *op* even if there are idle threads not covered by the flushed inputs. For example, the warp w_1 executes the group-by aggregation in the



(a) Push-down parallelism (PP). (b) Lane refill (LR).

Figure 3: Intra-WLB of DogQC for the parallel execution in Figure 2. The semi-join in figure 3a generates the input tuples in Figure 3b. The circled numbers indicate the execution order of warp iterations.

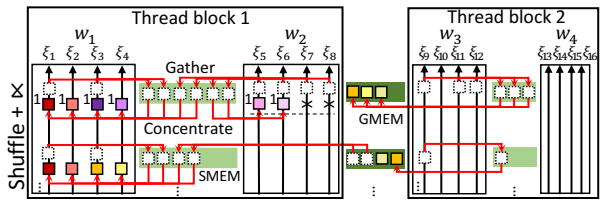


Figure 4: Intra-WLB of Pyper for the parallel execution in Figure 2.

warp iteration ④ even if the thread ξ_4 becomes idle. It is because w_1 already processes all inputs for the semi-join generated in the warp iteration ①. The problem is that inputs for the producer will frequently be insufficient if the inputs are generated by a join with a skewed join key distribution.

Pyper proposed an intra-WLB technique, Shuffle, as an alternative to LR. The difference with LR is that *multiple* warps flush (i.e., gather) tuples into a buffer shared by them, and the gathered tuples are processed by (i.e., concentrated over) fewer warps among the warps that gather the tuples. Note that Shuffle redistributes tuples between warps, but it does not solve inter-WLIs. To avoid the tuple redistribution through a buffer in GMEM, Shuffle attempts the concentration across warps in a *thread block* first, not across all warps. A thread block is a group of warps, and the tuple redistribution between warps in a thread block can be executed through a buffer in *shared memory* (SMEM). SMEM is a specific type of cache we will detail in Section 2.1. If the number of tuples gathered by warps in a thread block is too small (e.g., less than half of *wSize*), the intra-WLI still occurs even if the gathered tuples are concentrated over one warp. To avoid such cases, if the number of the gathered tuples is less than a fixed threshold, Shuffle executes the concentration across all warps through a buffer in GMEM shared by all warps.

Unfortunately, Shuffle also fails to solve intra-WLIs, and it incurs a substantial synchronization cost for a buffer in GMEM [34]. If the number of tuples gathered by warps in a thread block is not a multiplier of *wSize*, the intra-WLI still occurs. For example, in Figure 4, since w_1 and w_2 in the same thread block gather six tuples, w_2 should execute the semi-join for two tuples only. Furthermore, skewed join key distributions frequently trigger the inter-block redistribution, and synchronization costs for the buffer slow down query execution. In the case of w_3 and w_4 in the same thread block, the gathered tuples are too few, so the tuples are flushed into the buffer in GMEM. Then, the flushed tuples are concentrated over w_1 and w_2 in a different thread block. The problem is that such inter-block redistribution occurs frequently while w_3 processes more than 10^8 join outputs in Figure 2.

We propose Themis, a GPU-accelerated relational query processing engine, which consistently solves inter-WLIs and intra-WLIs while minimizing the inter-warp redistribution costs. First, we simplify the execution of a pipeline on a GPU to parallel tree traversal by warps on a fine-grained computation tree, and we refer to the tree as an *evaluation tree*. In the tree, each node corresponds to an input tuple for an operator, and visiting a node (i.e., tuple) corresponds to evaluating an operator for the tuple. Here, the outputs of an operator have child-parent relationships with the input.

For intra-WLB, we can simply view that each warp visits up to *wSize* nodes for each warp iteration, and we propose a traversal order for a warp that minimizes intra-WLIs. This traversal order enables Themis to solve the intra-WLIs without any inter-warp redistribution.

In terms of inter-WLB, we can see that we need to consider not only the number of nodes but the sizes of the subtrees rooted at the nodes when we check the heaviness of a warp's workload. In this work, we propose a heuristic to approximate the workload size of a subtree and a novel work-sharing method that adaptively determines the load balancing granularity based on the approximated workload sizes of subtrees to overcome the limits of the FWS.

We also present the efficient implementation of the proposed inter-WLB and intra-WLB technique. To reduce the costs for retrieving and storing tuples in buffers located in GMEM, we employ a *lazy materialization* of attribute values to represent tuples in buffers concisely. We also propose an inter-warp redistribution method that alleviates contention problems by redistributing tuples through multiple buffers in GMEM.

To demonstrate that Themis effectively solves load imbalances even for skewed join key distributions, we use a benchmark JCC-H [2] as in [17, 22, 39, 42, 43], which introduces skewed join key distributions to TPC-H [41]. Themis significantly alleviates inter-WLIs and intra-WLIs, outperforming the runner-up by up to 379x.

We summarize our contributions as follows. We reinterpret pipeline execution as traversal on an evaluation tree to provide a simple and fine-grained view for the inter-WLB and intra-WLB problems (Section 3). We present a parallel tree traversal algorithm for a pipeline execution on a GPU (Section 4). We devise a traversal order that minimizes intra-WLIs (Section 5). We propose an inter-WLB technique that adaptively solves the inter-WLIs across various workloads (Section 6). We present an efficient implementation of our inter-WLB techniques (Section 7). We empirically validate the benefit of our intra-WLB and inter-WLB techniques on JCC-H [2], and Themis outperforms the baselines up to 379x (Section 8).

2 BACKGROUND

2.1 Characteristics of GPUs and CUDA

Figure 5 illustrates a simplified modern GPU architecture [31] that consists of streaming multiprocessors (SMs), L2 cache, and GMEM. Each SM has processing blocks (PBs) that contain cores, registers

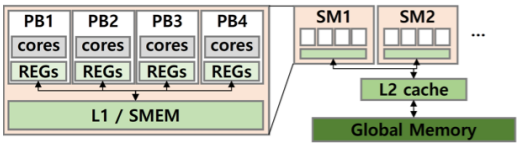


Figure 5: GPU architecture [31]

(REGs), a L1 cache, and SMEM. During execution, a PB executes instructions of threads using its cores [31].The combined L1 cache and SMEM are shared by PBs in an SM. Programmers can reside desired data in SMEM, manually, while load and eviction of data in L1 and L2 caches are managed automatically in hardware [31].

GPU processing is modeled as processing a kernel function in parallel with a massive number of threads managed in a grid-block-warp hierarchy [40]. The number of threads in a thread block, the number of thread blocks in a grid, and the number of grids should be specified before a GPU executes a kernel function. The number of threads in a thread block is usually configured to a multiple of *wSize*, and the threads are automatically divided into multiple warps. When a GPU starts to execute a kernel function, the GPU allocates an SM for each thread block considering available resources (e.g., REGs and SMEM). Here, an SM can be allocated to multiple thread blocks. After that, each warp in a thread block is assigned to a PB based on its identifier [15].

Threads can share data through REG, SMEM, and GMEM [32]. While GMEM allows data sharing between all threads, it is slow and incurs high synchronization costs. The threads within a thread block can share data via SMEM instead of GMEM. Moreover, the threads within a warp can share data via REGs. Accesses to REG and SMEM are at least 375 and 13 times faster than accesses to GMEM, respectively [15].

2.2 Pipeline Processing on a GPU

2.2.1 Pipeline. A pipeline P is a sequence of *non-blocking* operators, optionally followed by a *blocking* operator at the end, denoted as $[op_0, op_1, ..., op_{k-1}]$, where k is the number of operators in P. The blocking operator (e.g., Γ) is such that the operator consuming its outputs (i.e., consumers) cannot be executed until the blocking operator has processed all input tuples. In contrast, it is feasible to execute the consumer of a non-blocking operator (e.g., σ).

2.2.2 Base Table. The GPU-based pipeline processing methods [3, 8, 9] store tuples in a table as arrays in GMEM. For a fixed-sized attribute of the table, one array is used to store attribute values. To store a variable-sized attribute, an additional array is used to store pointers to attribute values.

Therefore, the i-th tuple in a table T can be simply represented as an offset i-1 with an indicator to T. If necessary, a thread processing this tuple loads its attribute values from the i-th slots of the arrays.

Further extending this idea, to maintain the outputs of all operators in a concise and unified way each operator is implemented to produce an *offset range* to represent its set of outputs. For example, a base table T itself can be represented as $[0, |T|)$, where $|T|$ is the number of tuples in T. We now explain other operators' implementations and output representations.

2.2.3 Selection. For each input tuple, if the input satisfies the condition of a selection operator (σ), the operator is considered to return an output tuple identical to the input. Otherwise, there is no output tuple. If the offset of the input tuple is i, the selection operator produces either $[i,i+1)$ or $[i,i)$.

2.2.4 Join. For a join operator, previous studies [3, 8, 9, 23, 46] concisely represent the tuples in a build-side table with the same join-key value as an offset range by making a build-side pipeline store such tuples consecutively in the arrays for the build-side table. The offset ranges of all join-key values are stored in a hash table. Once a tuple from the probe side arrives, the join partners of the build-side table can be retrieved from the hash table. Instead of concatenating the join partners with the probe-side tuple, the corresponding build-side offset range and the probe-side tuple (also represented as an offset range) are returned as the join outputs.

2.2.5 Group-by Aggregation. Previous studies [8, 9] employ the hash-based aggregation [16]. For an input tuple, the group-by aggregation (Γ) is performed by updating the hash table where each bucket in the hash table stores the values of the group key and aggregation result. Since the group-by aggregation is a blocking operator (Section 2.2.1), it produces outputs once all inputs are consumed, where the hash table is returned, not an offset range.

2.2.6 Materialization. For materialization, in previous studies [8, 9, 34], the attribute values of an input tuple are appended to the arrays for the output table, to make the representation consistent with the base tables. As a blocking operator, it directly returns these arrays instead of an offset range.

2.3 Fixed-granularity Work-Sharing

As an inter-WLB technique, FWS was proposed in a previous study only for join queries [23]. In FWS, after a warp executes a join for an input tuple, the workload of a warp is considered heavy if the number of outputs exceeds a fixed threshold. To redistribute the heavy workload to other warps, the warp divides the join outputs into partitions and pushes these partitions to a queue in GMEM. The FWS keeps each partition size as the fixed threshold. When a warp becomes idle, it attempts to pop one of the pushed partitions from the queue. For example, in Figure 2, FWS makes the warp w_3 divide the large number of join outputs into partitions and push the partitions to the queue in GMEM.

However, FWS cannot solve inter-WLIs consistently for various workloads because its granularity can be too coarse-grained or too fine-grained depending on a workload. If the partition size is too large, a small number of partitions is pushed to the queue, and idle warps cannot become busy as the queue frequently runs out of partitions. Conversely, if the partition size is too small, the states of warps frequently transit from a busy state to an idle state. The tuple redistribution through a single queue in GMEM also requires expensive costs for pop or push operations of the queue. Even worse, the contention on this single queue makes idle warps wait long until they succeed to pop a partition.

3 ABSTRACTION FOR PIPELINE EXECUTION ON A GPU: EVALUATION TREE

We simplify the pipeline execution on a GPU as parallel traversal on a fine-grained computation tree (i.e., evaluation tree) by warps. On the evaluation tree, we reinterpret the inter-WLB problem to redistributing the nodes from busy warps to idle warps, and the intra-WLB problem to selecting a traversal order for a warp that minimizes idle threads.

For a given pipeline, an evaluation tree T consists of nodes where each node corresponds to an input tuple for an operator in the pipeline. Here, a pipeline is a sequence of k operators, [op_0, op_1, ..., op_{k-1}], and visiting a node (i.e., a tuple) t at a level l in T indicates evaluating op_l for t. If a tuple t' belongs to the output tuples of op_l for t, t' and t have child-parent relationships. Since op_0 is a scan operator, for notational purposes, we consider that the root node of T is an empty tuple, and tuples in a scanned table are children of the root node. Figure 6 shows an evaluation tree for the pipeline in Figure 2. For example, visiting a node at level one corresponds to executing the join of a probe-side tuple in table X and a build-side table Y in Figure 2.

Now, we reinterpret the parallel pipeline execution by warps as follows. Initially, the nodes located at level one are divided into partitions, and one partition is distributed to each warp. Here, each partition is defined as a set of nodes. Then, for each warp iteration, at most *wSize* nodes at the same level (inside the warp's partition) are visited by a warp, and their child nodes are expanded to the tree as new leaves. Formally, a warp's partition *part* is updated as follows after the warp visits nodes t_1, t_2, ..., t_n in a warp iteration.

$$part \leftarrow part - \{t_1, t_2, ..., t_n\} \cup c(t_1) \cup c(t_2) \cup ... \cup c(t_n)$$

Here, $c(t)$ is a set of a node t's children. Once the nodes for all the non-blocking and final blocking operators are visited, all nodes have been expanded, and the outputs of the blocking operator are returned as the pipeline results.

We can then view the inter-WLB problem as selecting nodes from partitions of busy warps with heavy workloads and redistributing them to idle warps. In the context of tree traversal, a warp's workload W is represented as {a subtree rooted at a node $t | t \in part$}, where *part* is the warp's partition. We can also define the size of W as the number of nodes in the subtrees of W. However, this size cannot be precomputed since all the descendant nodes are not known in advance. In Section 6, we instead propose a heuristic that approximates the workload size of a subtree and redistributes subtrees based on the approximations, while adaptively determining the granularity of subtrees to resolve the limitation of FWS.

The intra-WLB problem can be viewed as selecting a traversal order, or nodes for a warp to visit at each iteration. In Section 5, we show that the breadth-first-search (BFS) minimizes the intra-WLIs but requires an indefinite memory consumption up to the tree size. To overcome such a problem, we propose a novel traversal order that minimizes the intra-WLIs while using a fixed size of memory.

4 OVERVIEW OF THEMIS

In Themis, each warp executes Algorithm 1 to visit nodes in an evaluation tree. After the equi-partitioning the nodes at level one (Line 1), each warp starts to visit nodes. Here, *part* denotes the workload assigned to this warp, consisting of one or more subtrees

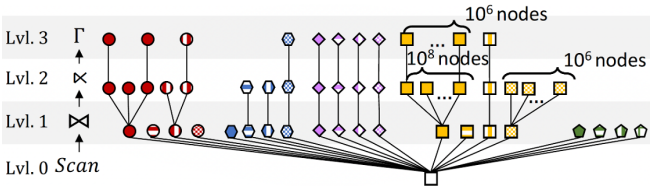


Figure 6: An evaluation tree of the pipeline in Figure 2.

in the evaluation tree. In each warp iteration, a warp chooses nodes in its partition based on our traversal order (Line 6) After that, it visits the chosen nodes (Line 7). The partition is updated to add their child nodes (Lines 8-9). For inter-WLB, each warp periodically checks if its workload is heavy and tries to redistribute its workload to idle warps. If the warp is not idle (Line 13), it redistributes its workload to an idle warp as long as it is determined as heavy (Line 14). If the warp is idle (Line 15), it instead waits for a busy warp to redistribute some workload and resume iterations (Lines 16-19) or ends iterations if all warps become idle (Line 20). After the redistribution, the warp chooses the next warp iteration number to check its heaviness (Line 21).

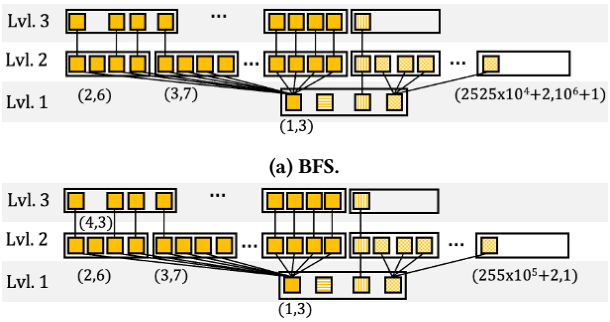
Algorithm 1: Traversal function for an evaluation tree T executed by a warp.

```
1  part ← a set of level-one nodes assigned to this warp
2  iteration ← 0
3  iterationToRedistributeWorkload ← 1
4  while true do
5      while part ≠ ∅ do
6          nodesToVisit, part ← chooseNodesToVisit(part)
7          visit the nodes in nodesToVisit concurrently
8          children ← a set of the child nodes of the visited nodes
9          part ← part ∪ children
10         iteration ← iteration + 1
11         if iteration > iterationToRedistributeWorkload :
12             break
13     if part ≠ ∅ : part ← redistributeWorkloadIfItIsHeavy(part)
14     else
15         while true do
16             if a busy warp shares nodes with this warp :
17                 part ← the set of the shared nodes
18                 break
19             if numIdleWarps = numAllWarps : return
20     iterationToRedistributeWorkload ← chooseIteration(iteration)
```

5 NO-IMBALANCE-FIRST-SEARCH FOR INTRA-WARP LOAD BALANCING

We propose a traversal order for a warp that minimizes the intra-WLIs while using a constant size of memory.

One of the traversal orders that minimize intra-WLIs is BFS, but BFS requires a large memory space for a queue of the nodes to visit. Note that a warp iteration visits *wSize* nodes at the same level regardless of the traversal order. Therefore, if the number of nodes at a level is not a multiple of *wSize*, then any order occurs an intra-WLI at least once. In BFS, such an intra-WLI occurs in the last warp iteration for each level. However, BFS requires a large memory space for a queue to store whole nodes at a level. Figure 7a



(a) BFS.

(b) NIFS.

Figure 7: An example of BFS and NIFS traversal on the evaluation tree in Figure 6 by the warp w_3 in Figure 2. Each large rectangle covers the nodes visited in each warp iteration. Here, the two numbers below each rectangle indicates the warp iteration number and the number of offset ranges the warp maintain after the warp iteration.

shows an example of BFS on the tree in Figure 6, and the intra-WLI occurs only in the last warp iteration at each level.

In order to minimize intra-WLIs as BFS while using a fixed size of memory, we propose *no-imbalance-first-search* (NIFS) as in Algorithm 2 that can be plugged into Algorithm 1. First, a warp tries to choose the highest level with at least *wSize* nodes and takes *wSize* nodes from the level (Lines 2-4). If it fails to choose any, it chooses the lowest level with at least one node and take *all* nodes from the level (Lines 5-7), where the number of nodes will be less than *wSize*. Figure 7b presents an example of NIFS that minimizes the intra-WLIs as BFS. After the iteration ①, ② becomes the highest level with at least *wSize* nodes. Then ③ becomes the highest since there are only three nodes at level 3 at the moment. ④ is chosen next. Following NIFS, w_3 minimizes the intra-WLIs as BFS, and it also maintains the small number of offset ranges during the traversal, while w_3 following BFS in Figure 7a needs to maintain 10^6+1 offset ranges after the last warp iteration at the level two.

We can prove that NIFS minimizes the intra-WLIs as BFS as follows. Note that the intra-WLI occurs if a warp chooses a level l with less than *wSize* nodes, i.e., Line 6 in Algorithm 2. This indicates that 1) there is no level with more than *wSize* nodes and 2) no remaining nodes at levels lower than l. 1) indicates that an imbalance must occur. 2) indicates that once making an imbalance for this level l, there will be no additional nodes at l; nodes will only appear at higher levels than l. Therefore, as in BFS, only the last warp iteration at level l generates intra-WLIs in NIFS.

The usage of a constant-size memory space to store nodes can be shown as follows. For each level, a warp maintains a queue in its REGs, whose elements are the offset ranges representing the assigned nodes at the level (Section 2.2). Hence, the set of all queues corresponds to the *part* in Algorithm 1. After equi-partitioning the nodes at level one, a warp stores one offset range in its *level-1* queue (Line 1 of Algorithm 1). Then, at each iteration, following Algorithm 2, a warp selects a level l, and it chooses nodes from offset ranges in the front of the level-l queue. In detail, a warp starts with the first offset range in the queue. If the offset range contains more nodes than it needs, it chooses the front *wSize* nodes in the range and adjusts this range to discard the chosen nodes. Otherwise, it chooses all nodes from the range and removes the range from the queue. After choosing nodes, the warp visits the nodes, and it appends the offset ranges corresponding to the child nodes of the visited ones to the level-(l+1) queue (Lines 8-9 of Algorithm 1).

The key aspect is that the number of offset ranges for the new child nodes is at most *wSize*, since each visited node can generate one offset range as explained in Section 2.2. Here, according to the NIFS, the number of nodes (and thus the number of offset ranges) in the level-(l+1) queue must had been less than *wSize*. Otherwise, level l+1 should had been selected instead of level l. Therefore, at any time (even after adding new child nodes), the number of offset ranges in the level-(l+1) queue is less than *wSize* + *wSize* = 2 × *wSize*. This proves that NIFS requires a constant-size memory of at most 2 × (k-1) × *wSize* × 8B (the size of an offset range, of just two integers), where k is the height of an evaluation tree.

Algorithm 2: Function executed by a warp to choose tuples to visit.

```
   Function chooseNodesToVisit(part)
1  |  nodesToVisit ← ∅
2  |  if there is a level with at least wSize nodes in part :
3  |  |  l ← the highest level with at least wSize nodes in part
4  |  |  nodesToVisit ← a set of wSize nodes at l in part
5  |  else
6  |  |  l ← the lowest level with at least one node in part
7  |  |  nodesToVisit ← a set of all nodes at l in part
8  |  return nodesToVisit, part - nodesToVisit
```

6 ADAPTIVE WORK-SHARING FOR INTER-WARP LOAD BALANCING

We propose a novel *adaptive-granularity* work-sharing (AWS) that solves the limitations of FWS and reduces inter-WLIs. Since it is hard to allocate equi-sized workloads to warps in the first place, we first set indirect goals to reduce the number of busy warps that become idle and the average waiting time of an idle warp until it becomes busy again.

To achieve the first goal, we let the *busiest* warp redistribute *half* of its workload to an idle warp. The rationale for selecting the busiest warp, i.e., the warp with the largest workload, is that redistributing workload from a warp with a small workload to an idle warp would result in both warps becoming idle soon thereafter. The rationale for selecting half of its workload is to balance the loads between the giver and the receiver.

To determine the busiest warp, we approximate the size of each warp's workload W under the heuristic that a subtree's size increases exponentially with its height, so the highest subtrees will dominate the workload size. Formally, the workload size is approximated as a pair $(h, |\{ST|ST \in W \wedge height(ST) = h\}|)$, where $h = \max_{ST \in W} height(ST)$, and $|\cdot|$ is a cardinality function for a set. Given a subtree, we use (the number of operators in a pipeline, i.e., k) - (the level of the subtree's root) as its height. If a workload size (h_1, n_1) is larger than another size (h_2, n_2), either 1) $h_1 > h_2$ or 2) $h_1 = h_2$ and $n_1 > n_2$.

AWS provides a theoretical bound for the number of times a warp becomes idle as follows: $I \leq \lceil \log_2(N) \rceil \times nWarps \times (k\text{-}1)$. Here, I

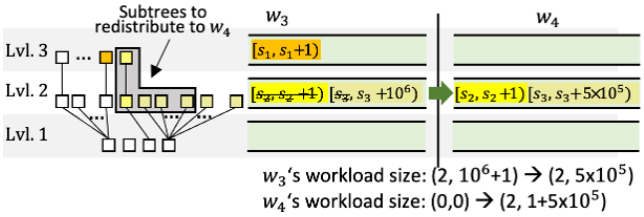


Figure 8: An example of the inter-warp tuple redistribution that redistributes half of a warp w_3's workload to an idle warp w_4. Each white square indicates a visited node. Each light green rectangle indicates a queue of offset ranges in REGs to maintain the input nodes at a level (Section 5). Here, each offset range in a queue is marked as the same color with corresponding nodes. For example, the offset range $[s_2, s_2+1)$ represents the node at the level 2 marked as yellow. The s_1, s_2, and s_3 are the start offsets of the offset ranges.

is the number of times a warp becomes idle, N is the number of the nodes in an evaluation tree, k is the height of the evaluation tree, and *nWarps* denotes the number of warps. This can be proved as follows. First, the busiest warp can have (k-1, n) as the largest workload size for some n. Formally, $\max_i(s_i) \leq (k\text{-}1, n)$, where s_i is the workload size of the i-th warp. After warps become idle *nWarps* times, $\max_i(s_i) \leq (k\text{-}1, \lfloor n/2 \rfloor)$ since 1) whenever a warp becomes idle, half the workload of the busiest warp is shared with the idle warp, and 2) thus the busiest warp at the moment must have shared its workload with some idle warp. Then, as $n < N$, we can guarantee that $\max_i(s_i) < (k\text{-}1, 1)$ after warps become idle $\lceil \log_2(N) \rceil \times$ *nWarps* times. Using the same approach, warps can become idle at most $\lceil \log_2(N) \rceil \times$ *nWarp* $\times(k\text{-}1)$ in total. AWS corresponds to Line 14 of Algorithm 1 and is implemented as the first function in Algorithm 3. First, the (approximated) workload size (h, n) is calculated in Lines 1-2 of Algorithm 3.

However, we instead use the log scale of n as in Line 3 to avoid frequent changes in its value and reduce the tracking cost of the busiest warp's workload size in Lines 4-5. This tracking is implemented using a hash table in GMEM of workload sizes (keys) and the numbers of warps having that workload sizes (values). During the traversal, each warp updates the hash table if its workload size changes. Using the log scale for n reduces the frequency of hash table updates. Each warp scans the hash table from the largest key to check if it is the busiest warp (Line 6).

If this warp is not the busiest warp, no work sharing occurs (Lines 4-5). Otherwise, it tries to redistribute half of its workload to an idle warp and returns the remaining workload (Lines 6-7). Figure 8 shows an example of the redistribution. In the example, the warp w_3 redistributes half of the highest subtrees to the idle warp w_4, and w_3 updates its level-2 queue to discard the redistributed nodes.

To reduce the average waiting time of an idle warp until it becomes busy again, we introduce an adaptive mechanism allowing busy warps to adjust the interval of the warp iterations to redistribute subtrees to an idle warp (Line 4 of Algorithm 1) based on the total number of idle warps. This approach is implemented in the second function of Algorithm 2. In Line 8, a warp chooses a small interval if the majority of warps are idle, to make such idle warps quickly busy again. Otherwise, it chooses a large interval to avoid the overhead of unnecessary redistribution attempts. The maximum interval is set to 32.

Algorithm 3: Functions executed by a warp for work-sharing.

Function *redistributeWorkloadIfItIsHeavy(part)*
1 $h \leftarrow$ the height of the highest subtrees in *part*
2 $n \leftarrow$ the number of subtrees with height h
3 *workloadSize* $\leftarrow (h, \lceil \log_2(1 + n) \rceil)$
4 **if** *workloadSize* < current busiest warp's *workloadSize* **: return** *part*
5 try to redistribute $\lceil n/2 \rceil$ subtrees with the height h to an idle warp
6 **return** the set of remaining nodes after the redistribution
Function *chooseNextIteration(currentIteration)*
7 *interval* $\leftarrow 1 + \min(32, \lfloor \log_2(numAllWarps \,/\, numIdleWarps) \rfloor)$
8 **return** *currentIteration* + *interval*

7 EFFICIENT IMPLEMENTATION OF THEMIS

We propose an efficient node redistribution approaches that allow warps to concurrently redistribute their subtrees to idle warps and lazily materialize attribute values.

To reduce the cost of inter- and intra-warp tuple redistribution, Themis employs a lazy materialization of attribute values. When a warp evaluates the succeeding operator of a scan or a join, in DogQC and Pyper, the warp eagerly loads attribute values that will be used for succeeding operators and stores the values into buffers. In Themis, a warp stores the offset for table arrays that store the values (Section 2.2.2) and lazily loads the values when they become necessary, for example, filtered attributes for selections and group-by attributes for aggregations. This approach avoids unnecessary memory accesses to GMEM. Furthermore, this technique reduces the redistribution cost by enabling threads to share offsets only not the attribute values for the redistribution. As we will experimentally demonstrate in Section 8.3.1, this optimization reduces query execution times by up to 30%, especially when the filtering ratios of selection operators are low (< 25%).

For efficient AWS, we allow busy warps to concurrently share their workloads with idle warps using the hierarchical bitmap [28] in GMEM and a dedicated buffer in GMEM for each warp. This parallel redistribution enables us to alleviate the contention problem compared to the single buffer approach of Pyper and FWS.

The hierarchical bitmap of Themis comprises two bitmaps, bit_1 and bit_2. The i-th bit in bit_2 is set to one to indicate that the i-th warp is idle. Similarly, the j-th bit in bit_1 is set to one if there is any idle warp between the $(64 \times (j-1) + 1)$-th and $(64 \times j)$-th warps.

When a warp becomes idle, it uses atomic operations to 1) mark the status of its buffer as empty, 2) update the number of idle warps, and 3) update the bitmap to indicate that it is idle. Then, it periodically checks its buffer and the number of idle warps. If the buffer becomes non-empty, it retrieves the stored nodes from its buffer and stores the nodes into its queue in REGs. If the number of idle warps becomes equal to the total number of warps, the warp terminates.

To identify an idle warp, a busy warp first randomly selects a 64-bit segment in bit_1 and picks a bit set to one. It then examines the corresponding 64-bit segment in bit_2, chooses a bit set to one, and set the bit to zero using atomic operations. To redistribute its workload to the identified idle warp, it stores its nodes to the idle

warp's buffer in GMEM. Here, the busy warp divides each offset range in its queue into two offset ranges, and it stores one of the two offset ranges to the idle warp's buffer. This ensures that the cost remains constant regardless of the number of nodes being shared, and this inter-warp redistribution occurs $O(\log(N))$ times as we explained in Section 6. In Figure 8, the buffer for w_4 lies between w_3 and w_4's queues. The buffer for a warp requires a fixed size of memory as the queues for a warp in Section 5.

8 PERFORMANCE STUDY

We conduct a comprehensive evaluation of Themis against DogQC and Pyper, focusing on query processing times and addressing intra- and inter-WLIs. The workload for our experiments is the JCC-H benchmark, the variant of the TPC-H [41], where join key distributions are skewed. We also conduct experiments using the TPC-H benchmark. We compare the methods on various environments by varying 1) a scale factor, 2) the number of warps per block, 3) the number of warps. Due to the space limitation, the key results are presented here, and the additional results are available at [21].

8.1 Experimental Setup

To conduct evaluations in a realistic environment, we employ JCC-H benchmark [2], which introduces skewed join key distributions into the TPC-H benchmark [41]. The experiments were performed using the dataset on a scale factor of 30. For the dataset, as we explained in Section 2, we build tables and indices for the primary-key and foreign-key relationships.

We evaluate DogQC, Pyper, and Themis for all 22 JCC-H queries. Our focus is on assessing their load balancing techniques, so we use the same plan for direct comparison to see if they can effectively and efficiently eliminate both intra-WLIs and inter-WLIs. Because of the absence of a query optimizer available for all 22 queries [34], we generate the variants of the DogQC's plans [7] considering *index hash joins*, and choose the best plan that demonstrates the fastest execution time for each query when using a single thread. The index hash join is a hash join that uses a pre-built hash table, and it allows us to choose plans that are 1.15 to 4 times faster than plans using hash joins only. However, the plans using index hash joins have significant load imbalances as we will explain in Section 8.3.1.

We extend the released implementation of DogQC [7] to support index hash joins. We also make LR available in warp iterations to evaluate any operators. In the original implementation, LR is not supported in warp iterations to evaluate succeeding operators of a join in a pipeline where the join produces multiple outputs for an input. We refer to our extended version as DogQC++. In the experiments, we set the number of warps of DogQC++ following the TPC-H benchmark experiment of its paper (80 warps per SM which can execute 32 warps concurrently).

For Pyper, we develop an implementation following its paper, owing to the absence of publicly available source code or binaries. To investigate the trade-off of the expensive inter-block redistribution (IBR) of Pyper, we evaluate two variants of Pyper: *Pyper-w/o-ibr*, which omits IBR, and Pyper, which incorporates it. Pyper uses the same number of warps with DogQC++.

For Themis, for the ablation study of our load balancing techniques, we evaluate two versions: *Themis-w/o-aws*, which excludes AWS, and Themis, which exploits both NIFS and AWS. Themis also uses the same number of warps with DogQC++.

To evaluate the load balancing techniques, we mainly present the execution times for a given query. Additionally, we assess the intra-WLIs and inter-WLIs by reporting two metrics: the *intra-warp idle ratio* (IIR) and the *Inter-Warp Load Imbalance Factor* (ILIF). The IIR is (the average number of idle threads in a warp iteration)/*wSize*. ILIF for a given pipeline is (the maximum execution cycles)/(average execution cycles) of warps when they process tuples. For each query, we report the weighted sum of the ILIFs for all pipelines. The weight for each pipeline is (the total execution cycles of warps for each pipeline)/(the total execution cycles of warps).

We conduct all the experiments on a machine running Ubuntu 20.04, equipped with an NVIDIA RTX3090 GPU, which boasts 82 SMs and 24GB of GMEM. Each SM has four PBs, with the L1 cache and a register file size being 128KB and 256KB, respectively, for each SM. We utilize NVIDIA driver version 470.103.01 and CUDA version 11.4. Because each SM in an NVIDIA RTX3090 can execute 64 warps concurrently, we set the number of warps to 82×160 ($\approx$ 13K) following the DogQC's experimental setup for the TPC-H benchmark (80 warps per SM which can execute 32 warps concurrently). We also set four warps to compose a single thread block.

8.2 Categorization of JCC-H Queries

To simplify exposition, we categorize the queries into two groups: 1) queries where the query plans include operators that produce multiple outputs per input (e.g., a join), and 2) queries where plans only contain operators that produce at most one output per input (e.g., a semi-join). Hereafter, we refer to the former as QG-M and the latter as QG-1 (Table 1). For QG-M queries, index hash joins are typically the operators generating multiple outputs per input. It is because indices enable us to scan a small dimension table first and execute a join operation on a large fact table at a low cost. We observe that DogQC++ and Pyper still have load imbalance problems for QG-M queries while they effectively solve the imbalances for QG-1 queries as we will show in Figures 10 and 11 later.

We investigate the characteristics of the queries to facilitate the analysis of the inter-WLI and intra-WLI problems of the baselines on JCC-H. Each query is represented by the pipeline that takes the longest execution time. Here, DogQC++, Pyper, and Pyper-w/o-ibr have the same longest pipeline for all queries except Q8. For Q8, the longest pipeline varies for each method, and we select the longest pipeline of DogQC++ since DogQC++ outperforms other baselines for Q8. The selected pipelines account for more than 50% of the total execution time for their respective queries. They also contain the operators that return multiple outputs for an input.

8.2.1 Characterization in Terms of the Inter-warp Load Imbalance. For characterizing the queries, we collect metrics for the selected pipeline per each query as evidence of the load balancing techniques' ILIF values as shown in Table 1. One such metric is the number of tuples in a scanned table (i.e., the size of a scanned table). If there are operators (e.g., a semi-join) that filter scanned tuples, we track the number of filtered tuples instead of the number of the scanned tuples. A *small* size of a scanned table, less than the number of warps, suggests potential inter-WLIs if we just evenly distribute the tuples in a scanned table to warps, as is the case with

baselines, DogQC++, Pyper-w/o-ibr, and Pyper. We consider the scanned table for a query to be *large* if the size of the table is larger than or equal to the number of warps.

To analyze inter-WLIs on a query with a large scanned table, we characterize the query by the skewness in the number of tuples derived from a scanned tuple. In the query, processing the derived tuples is the main bottleneck for pipeline execution time. Ideally, if the baselines can evenly distribute the derived tuples to warps, they will not suffer from inter-WLIs. In this context, we refer to the workload per warp in this ideal scenario as the Ideal Workload per Warp (IWW), which is equal to (the # tuples derived from all tuples in a scanned table) / (# warps). However, it is impossible if there is a tuple t in a scanned table with an extremely large number of tuples derived from it that exceeds IWW. This is because DogQC++ and Pyper cannot redistribute the tuples derived from t. So, if there is a tuple in a scanned table where the number of tuples derived from it is larger than IWW, we consider the distribution of the number of tuples derived from each scanned tuple as highly skewed.

Table 1: Characteristics of JCC-H queries related to inter-WLIs of the baselines. The gray color indicates categories where we expect to observe the inter-WLIs of the baselines.

Query group	The size of a scanned table	Skewness in # tuples derived from each tuple in a scanned table	Query ID
QG-M	Small	High	Q2, Q8, Q9, Q11, Q21
		Low	Q5, Q7, Q17, Q18, Q20
	Large	High	Q4, Q10, Q13, Q16, Q22
		Low	Q3
QG-1	Large	Low	Q1, Q6, Q12, Q14, Q15, Q19

8.2.2 Characterization in Terms of the Intra-warp Load Imbalance. To explain the trade-off of Pyper's IBR, we categorize QG-1 queries based on the *filtering ratio* of the selected pipeline as shown in Table 2. Here, the filtering ratio is (# filtered tuples in a scanned table)/(# tuples in a scanned table). The selected pipeline includes an operator that filters scanned tuples, and alleviating intra-WLIs when a warp processes the output tuples of the operator (i.e., filtered tuples in a scanned table) is crucial. As we will explain in Section 8.3.2, we expect the intra-WLIs of Pyper-w/o-ibr due to the absence of IBR, especially for the queries where the filtering ratio is low (< 25%). In contrast, Pyper is anticipated to demonstrate longer execution times than Pyper-w/o-ibr due to the overhead of IBR.

In the case of QG-M queries, we classify the queries by the size of a scanned table to explain the significant intra-WLIs of Pyper. To guarantee that there is no remaining tuple in buffers after a query execution, Pyper has a post-processing step that processes the remaining tuples without IBR. For the queries marked as blue in Table 2, Pyper will suffer from the intra-WLIs since the tuples in a tiny scanned table will be stored in the buffers, and the tuples will be processed in the post-processing step.

We also characterize the selected pipeline of each QG-M query based on the average output size of the last join per input to demonstrate the DogQC++'s limitations discussed in Figure 3. In the selected pipeline, processing the outputs of the last join is the main bottleneck for the pipeline execution, and alleviating the intra-WLIs for the outputs is crucial. However, DogQC++ will show higher IIRs compared to Themis-w/o-aws on queries marked as orange in Table 2 where the output size of the last join is small (< *wSize*).

Table 2: Characteristics of JCC-H queries related to intra-WLIs. Pyper-w/o-ibr, Pyper, and DogQC++ are anticipated to show intra-WLIs on categories marked as sky-blue, blue, and orange, respectively.

Query group	# filtered tuples in a scanned table / # tuples in a scanned table	Query ID
QG-1	High (> 98%)	Q1
	Low (< 25%)	Q6, Q12, Q14, Q15, Q19

Query group	The size of a scanned table	The average # output tuples from the last join operator	Query ID (# join operators)
QG-M	Tiny	Small	Q5 (4), Q8 (3), Q9 (2)
		Large	Q2 (3), Q7 (1), Q11 (3), Q17 (1), Q18 (1), Q20 (1), Q21 (2)
	Large	Small	Q3 (2), Q4 (1), Q10 (1), Q16 (1)
		Large	Q13 (1), Q22 (1)

8.3 Performance Study on JCC-H

8.3.1 Execution Times on JCC-H. Figure 9 illustrates the query processing times of the baselines, Themis-w/o-aws, and Themis.

In terms of inter-WLB, for queries where we expect the inter-WLIs (marked as gray in Table 1), Themis outperforms DogQC++, Pyper-w/o-ibr, Pyper, and Themis-w/o-aws by up to 173x with AWS. In the case of other queries, Themis shows 1.04x longer execution times compared to Themis-w/o-aws due to the overhead for the work-sharing, such as the cost to track the number of idle warps. Despite the overhead, Themis still outperforms DogQC++, Pyper-w/o-ibr, and Pyper by 9% on average because of its inter-WLB technique and the lazy materialization of attribute values.

Among the intra-WLB techniques, Themis-w/o-aws generally exhibits shorter execution times than the DogQC++, Pyper, and Pyper-w/o-ibr up to 2x, 30x, and 440x, respectively. It is because NIFS efficiently reduces the IIRs as we will explain in Section 8.3.2. According to Figure 9, DogQC++ also shows shorter execution times than Pyper and Pyper-w/o-ibr, but Themis outperforms DogQC++ especially on QG-M queries marked as orange in Table 2.

For QG-1 queries especially where the filtering ratio is low (marked as sky-blue in Table 2), Themis-w/o-aws is up to 1.4x faster than DogQC++ and Pyper even if they have no intra-WLI. Pyper-w/o-ibr shows longer execution times than DogQC++ and Themis-w/o-aws due to the remaining load imbalances. In contrast, Pyper suffers from the expensive cost of IBR. The advantage of Themis-w/o-aws compared to the runner-up, DogQC++, comes from the lazy materialization of attribute values. It enables Themis-w/o-aws to avoid the unnecessary memory accesses for the attribute values of scanned tuples filtered out, while DogQC++ eagerly loads attribute values of all scanned tuples.

On the queries in QG-M, Pyper-w/o-ibr and Pyper exhibit longer execution times than Themis-w/o-aws. The slow pipeline execution

of Pyper-w/o-ibr is due to the significant intra-WLIs. While Pyper alleviates the intra-WLIs of Pyper-w/o-ibr, the expensive IBR makes Pyper still slower than DogQC++ and Themis-w/o-aws.

For QG-M queries with small average output sizes from the last join per input tuple (marked as orange in Table 2), Themis-w/o-aws outperforms DogQC++ by up to 2x, averaging 1.3x. This is because Themis-w/o-aws effectively solves intra-WLIs for the queries where DogQC++ struggles with the small join output size per input.

In QG-M queries where the average output size of the last join per input tuple is large, Themis-w/o-aws exhibits shorter execution times than DogQC++ by up to 1.4x and on average 1.07x. Here, both Themis-w/o-aws and Pyper effectively alleviate the intra-WLIs on these queries. This marginal improvement of Themis-w/o-aws over DogQC++ is due to the lazy materialization (Section 7).

We also evaluate the effect of lazy materialization, and it reduces query execution times of Themis and Themis-w/o-aws by 38% and 4% for the JCC-H queries. We also enable the lazy materialization for the baselines, and we observe that Themis-w/o-aws and Themis still exhibited 1.17x and 44.01x shorter execution times compared to the baselines that use the lazy materialization.

8.3.2 Intra-Warp Load Imbalances on JCC-H. Figure 10 depicts the IIRs for the baselines and Themis-w/o-aws, and we explain the cause of the intra-WLIs in this section.

For all queries, Themis-w/o-aws consistently demonstrates low IIRs with an average value of 1%, which is 14 times smaller than that of the runner-up, DogQC++, across all 22 JCC-H queries. The highest IIRs of Themis-w/o-aws is 8%, while DogQC++, Pyper-w/o-ibr, and Pyper exhibit IIRs up to 69%, 97%, and 96%, respectively. This shows the effectiveness of NIFS always prioritizing operators with a sufficient number of input tuples ($\geq wSize$).

For QG-1 queries where the filtering ratio is low (marked as sky-blue in Table 2), Pyper-w/o-ibr shows a high average IIR of 22% while those of other methods are lower than 3%. With the high intra-WLIs, Pyper-w/o-ibr shows 1.4x longer execution times than Themis-w/o-aws. It is due to the absence of IBR which is a solution for when the total number of active threads in a thread block drops below the threshold of *wSize*, as demonstrated in Figure 4. The selected pipeline of each QG-1 query contains an operator that filters scanned tuples. After the warps in a thread block execute a filtering operator, if the filtering ratio is less than 25%, the total number of active threads in the warps becomes lower than *wSize* because each thread block is composed of four warps in our experimental setup.

On the queries in QG-M, Pyper-w/o-ibr shows a higher average IIR (85%) than QG-1 queries. It is because the need for IBR increases on QG-M queries due to the join operators, which yield a varied number of output tuples per input compared to QG-1 queries where the operators produce at most one output.

For QG-M queries with tiny scanned tables (marked as blue in Table 2), Pyper demonstrates the high average IIR (60%) as Pyper-w/o-ibr. This is because Pyper cannot use the IBR for the selected pipelines of the queries. As we explained in Section 8.2.2, Pyper stores the tuples in a tiny scanned table to a buffer, and it processes the tuples in the processing step without IBR.

For QG-M queries where the average output size of the last join operator per input is less than *wSize* (marked as orange in Table 2), DogQC++ exhibits ten times higher average IIRs (44%) compared to other QG-M queries. With intra-WLIs, DogQC++ shows up to 2x longer execution times than Themis-w/o-aws because of the limitations of DogQC++'s PP and LR explained with Figure 3. PP makes the threads in a warp evenly take outputs of a join for an input tuple, and the number of outputs less than *wSize* causes intra-WLIs. In LR, a warp cannot use the buffering technique in a warp iteration for the evaluation of an operator if there is no remaining input for the producer of the operator. In other words, when the last join produces inputs for the producer, LR cannot solve the intra-WLIs if the join produces a small number of outputs per input.

8.3.3 Inter-warp Load Imbalances on JCC-H. Figure 11 presents the values of ILIF for the baselines and Themis across the 22 JCC-queries. For the queries where we expect the inter-WLIs (marked as gray in Table 1), we observe that Pyper, Pyper-w/o-ibr, DogQC++, and Themis-w/o-aws demonstrate the high ILIF values of 777, 943, 907, and 793 respectively. In the case of Themis, the average and the maximum ILIF values are 1.8 and 4.4, respectively, due to AWS. By alleviating the inter-WLIs, Themis shows up to 173x shorter execution times compared to DogQC++, Pyper, Pyper-w/o-ibr, and Themis-w/o-aws on queries where the competitors suffer from inter-WLIs, as previously explained in Section 8.3.1. For other queries, all methods exhibit low ILIFs below two, indicating no inter-WLIs.

8.3.4 Experiments on Various Environments. We also evaluate the baselines and Themis by varying 1) a scale factor, 2) the number of warps per block, and 3) the number of warps. We also compare the methods on TPC-H. Themis consistently outperforms the baselines on the various environments, due to NIFS and AWS. First, Themis consistently demonstrates shorter execution times on average 7x, 18x, 19x, 23x, 35x, 35x, and 33x, compared to the runner-up, DogQC++, for the scale factors: 1, 5, 10, 15, 20, 25, and 30, respectively. Themis also shows 32x, 33x, 32x, and 34 shorter execution times than the runner-up, DogQC++, when we vary the number of warps per thread block from 1 to 2, 4, and 8. The number of warps is set to 13K. Themis also demonstrates shorter execution times on average 2x, 6x, 21x, 31x, and 33x, compared to DogQC++, when we increase the number of warps from 4 to 32, 256, 2048, and 16384. The number of warps per thread block is set to 4. In TPC-H, Themis shows 35x shorter execution times than DogQC++.

8.4 Cost Breakdown of Work-Sharing

To delve deeper into the effectiveness of AWS over FWS, we perform a cost breakdown for cost breakdown for the three queries with the highest ILIF values in each gray-colored category in Table 1 where the baselines experience inter-WLI. For this breakdown, we measure the execution times of warps to process tuples (i.e., processing time). We also record 1) the duration that an idle warp waits until the idle warp becomes busy again (i.e., waiting time) and 2) the frequency of the state transitions of warps from busy to idle.

For the evaluation, we implement a variant of Themis that exploits FWS (fws) [23]. To demonstrate the effect of the granularity of fws (i.e., the threshold for the join output size per input tuple), we evaluate fws with four different thresholds, 1K, 2K, 4K, 8K, and 16K. In [23], the threshold was set to 1K for relational data.

Figure 12 presents the cost breakdown of the work-sharing strategies. For the selected queries of the gray-colored categories in Table 1, Themis consistently outperforms fws due to AWS. In Figure

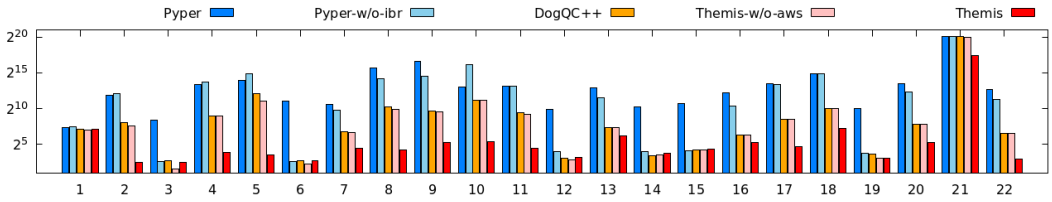


Figure 9: Execution times (ms) on JCC-H queries.

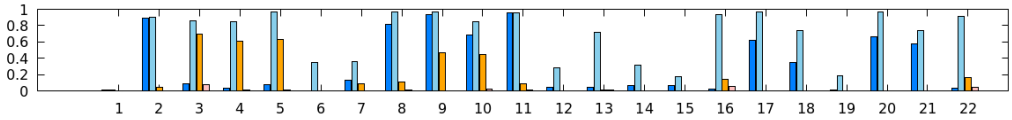


Figure 10: The intra-warp idle ratios (IIRs) on 22 queries of JCC-H.

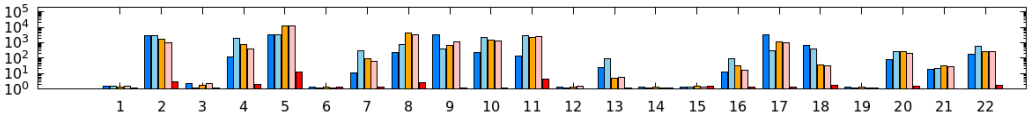


Figure 11: Inter-warp load imbalance factors (ILIFs) on JCC-H queries.

12, fws fails to consistently solve inter-WLIs because of the fixed granularity as explained in Section 2.3. If the granularity is too coarse-grained, few partitions are pushed to the queue in GMEM, leaving idle warps idle, as seen in Figures 12c and 12b. Conversely, if the granularity is too fine-grained, the states of warps frequently transit from a busy state to an idle state, and the total waiting times increase as shown in Figures 12e and 12f. Additionally, contention on the queue further extends waiting times. In contrast, Themis consistently solves inter-WLIs because of AWS. It also quickly transits idle warps to a busy through parallel tuple redistribution, facilitated by the hierarchical bitmap and multiple buffers (Section 7).

We also evaluate the effect of our parallel tuple redistribution method, and Themis shows 2x shorter execution times compared to the Themis that uses a global single buffer instead of the hierarchical bitmap and the multiple buffers. We also compare Themis with the fws that maintains a buffer per warp and uses the hierarchical bitmap to detect the non-empty buffer. Our parallel tuple redistribution method decreases the query execution times of fws 1.6x, however the fws still demonstrates 1.6x longer times than Themis. For this comparison, we compared the execution time of Themis with the shortest result among the execution times of the fws for the thresholds, 1K, 2K, 4K, 8K, and 16K, as used in Figure 12.

8.5 Summary

In summary, Themis outperforms the baselines, DogQC++, Pyper for the queries where load imbalances exist. It is because Themis solves the intra-WLIs with NIFS (Section 5), and also resolves inter-WLIs by AWS (Section 6). It also reduces the tuple redistribution cost with the lazy materialization of attribute values and the parallel tuple redistribution between warps (Section 7).

9 RELATED WORK

We discuss the related work in the database area that was not explained in sections before. First, we explain the previous work that tries to solve intra-WLIs. In the CPU area, vectorized analytical query processing methods [24, 37] can be applied for intra-WLI issues. DogQC's lane-refill strategy is derived from the strategy in [24], and VIP [37] also uses a similar strategy that gathers enough intermediate results in a buffer before evaluating the intermediate results for an operator. However, these studies focus on the utilization of SIMD lanes during execution of one operator (e.g., join or filter), while Themis solves intra-WLIs for pipeline.

Second, we present the related work to solve inter-WLIs. Materializing intermediate results and distributing the results to threads are used in [10, 35, 46] to solve inter-WLIs as well as intra-WLIs. These methods can be applied for inter-WLB. However, the intermediate results materialization method suffers from a severe overhead of materialization. Morsel-driven parallelism [25] in CPU area can also be a baseline as FWS for inter-WLB. However, the morsel-driven approach is too coarse-grained because the smallest unit for load balancing is a tuple in a table scanned by the first operator in a pipeline. The approach divides the tuples in a table to scan into fixed-size groups (i.e., morsels), and the morsels are assigned to threads in a round-robin fashion.

Third, we discuss the earlier methods that generate GPU codes for a given pipeline. RedFox [45] generates a code following the operator-at-a-time approach that materializes the whole intermediate results. Kernel weaver [44], HorseQC [8], HetExchange [5], Voodoo [36], and Crystal [38] fuse relational operators for a given pipeline. Kernel weaver and HorseQC focus on reducing PCI-e communication costs between CPU and GPU by fusing operators

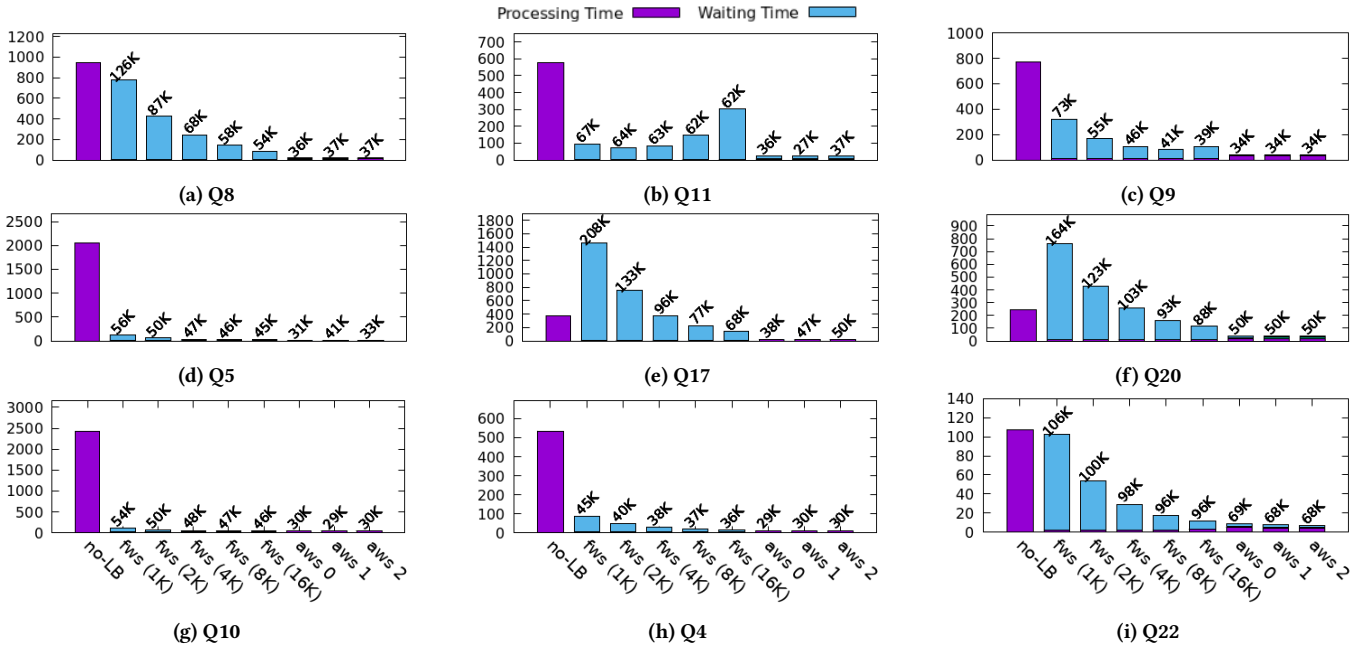


Figure 12: The cost breakdown of inter-WLB techniques for JCC-H queries Q2, Q4-5, Q7-11, Q13, Q16-18, and Q20-22. The Y-axis indicates the execution time in milliseconds. *no-LB* represents Themis-w/o-aws, and *fws* is a variant of Themis that uses FWS. We evaluate fws varying its granularity. *aws* denotes Themis, and we execute a query three times to show that aws alleviates the inter-WLIs consistently. The subsequent number following the name of aws indicates the execution ID. The processing time refers to the duration required to process tuples, while the waiting time denotes the period during which idle warps wait before becoming busy again. The number above each bar is the frequency of state transitions of warps from a busy state into an idle state.

because they send the materialized intermediate tuples to CPU memory from GPU memory. Voodoo generates an OpenCL code to execute on both CPUs and GPUs. HetExchange proposes code generation that utilizes multiple CPUs and GPUs concurrently. Crystal provides a library of data processing primitives that enables a programmer to implement code where operators are fused. Since the operator fusion methods do not have inter- and intra-WLB techniques, we compared Themis only with DogQC and Pyper that propose intra-WLB techniques. Recently, Crystal-Opt [4] shows that Crystal can be improved by applying the lazy materialization of attribute values, but it still has inter- and intra-WLI problems. Themis also shows 2.14x shorter execution times over Crystal on the star schema benchmark [33] used in Crystal's experiment.

Fourth, we compare the load balancing approach of the subgraph enumeration methods [10, 11, 27] with Themis. The subgraph enumeration can also be viewed as a tree traversal problem, and [10, 11, 27] visit nodes in a DFS manner. Specifically, they iterate visiting a fixed number of nodes at the highest level among levels where there are nodes to visit. In each iteration, warps visit nodes in parallel, and each warp generate the children of the nodes it visits in GMEM to enable any warps to visit the child nodes in the next iteration. This parallelization approach materializes all nodes in GMEM. Furthermore, it still has the load imbalance problems since the number of children varies for each node. Themis also shows shorter query execution times than the load balancing approach of the subgraph enumeration methods on the graph queries and datasets of [11] by up to 13x, with an average of 5x.

10 CONCLUSIONS

In this paper, we have presented Themis, a GPU-accelerated relational query processing engine that solves inter- and intra-WLIs. First, we view the pipeline execution as the tree traversal on an evaluation tree. On the evaluation tree, we reinterpret the inter-WLB problem to redistributing the subtrees from busy warps to idle warps, and the intra-WLB problem to selecting a traversal order that minimizes idle threads. Second, we propose NIFS that minimizes intra-WLIs. Third, we present AWS that solves the inter-WLIs across various workloads. Last, the experimental results have shown that Themis effectively solves inter- and intra-WLIs significantly and outperforms the baselines, DogQC and Pyper on JCC-H by up to 379x.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1A2B5B03001551) and Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No. RS-2018-II181398, Development of a Conversational, Self-tuning DBMS, 50%).

REFERENCES

[1] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2012. Building an efficient hash table on the GPU. In *GPU Computing Gems Jade Edition*. Elsevier, 39–53.

[2] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: adding join crossing correlations with skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era: 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers 9*. Springer, 103–119.

[3] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27, 6 (2018), 797–822.

[4] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454.

[5] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines.* Technical Report.

[6] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *The VLDB Journal* 27 (2018), 617–641.

[7] Funke. 2022. *Github repository of DogQC.* https://github.com/Henning1/dogqc

[8] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. 1603–1618.

[9] Henning Funke and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *Proceedings of the VLDB Endowment* 13, 6 (2020), 884–897.

[10] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1067–1082.

[11] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4231–4244.

[12] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 511–524.

[13] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. 2018. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.

[14] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. MultiGraph: Efficient graph processing on GPUs. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 27–40.

[15] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[16] Peng Jiang and Gagan Agrawal. 2017. Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[17] David Justen. 2022. Cost-efficiency and Performance Robustness in Serverless Data Exchange. In *Proceedings of the 2022 International Conference on Management of Data*. 2506–2508.

[18] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the 48th International Symposium on Microarchitecture*. 204–215.

[19] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 39–50.

[20] Farzad Khorasani, Bryan Rowe, Rajiv Gupta, and Laxmi N Bhuyan. 2016. Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 524–533.

[21] Young-Koo Lee Yang-Sae Moon Sourav S Bhowmick Kijae Hong, Kyoungmin Kim and Wook-Shin Han. 2024. *The full paper of Themis.* https://drive.google.com/drive/folders/1FoDxT4uS25iezcEgOY2ZMMHMQD0N0ej5

[22] Jan Kossmann, Daniel Lindner, Felix Naumann, and Thorsten Papenbrock. 2022. Workload-driven, lazy discovery of data dependencies for query optimization. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[23] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* 31, 3 (2022), 529–553.

[24] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2018. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *Proceedings of the 14th international workshop on data management on new hardware*. 1–8.

[25] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[26] Zhihao Li, Haipeng Jia, and Yunquan Zhang. 2017. HartSift: A high-accuracy and real-time SIFT based on GPU. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 135–142.

[27] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2016. Network motif discovery: A GPU approach. *IEEE transactions on knowledge and data engineering* 29, 3 (2016), 513–528.

[28] Mikołaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. 2003. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *Advances in Databases and Information Systems: 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003. Proceedings 7*. Springer, 236–252.

[29] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and Ponnuswamy Sadayappan. 2019. Load-balanced sparse mttkrp on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 123–133.

[30] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[31] NVIDIA. 2020. *NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.* https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[32] NVIDIA. 2022. *CUDA C++ Programming Guide.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-5-x

[33] Patrick E ONeil, Elizabeth J ONeil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

[34] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proceedings of the VLDB Endowment* 14, 2 (2020), 202–214.

[35] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*. 1935–1950.

[36] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1707–1718.

[37] Orestis Polychroniou and Kenneth A Ross. 2020. VIP: A SIMD vectorized analytical query engine. *The VLDB Journal* 29, 6 (2020), 1243–1261.

[38] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[39] Yijie Shen, Jin Xiong, and Dejun Jiang. 2020. SrSpark: Skew-resilient spark based on adaptive parallel processing. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 466–475.

[40] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Real-time top-k personalized pagerank over large graphs on gpus. *Proceedings of the VLDB Endowment* 13, 1 (2019), 15–28.

[41] TPC. 2022. *TPC-H.* https://www.tpc.org/tpch/

[42] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Transactions on Database Systems (TODS)* 46, 3 (2021), 1–45.

[43] Ziyun Wei and Immanuel Trummer. 2022. SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms. *Proceedings of the VLDB Endowment* 16, 4 (2022), 905–917.

[44] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 107–118.

[45] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 44–54.

[46] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar Yalamanchili. 2014. Multipredicate join algorithms for accelerating relational graph processing on GPUs. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, Vol. 10.

[47] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.