Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS

Bobbi W. Yogatama
University of Wisconsin-Madison
bwyogatama@cs.wisc.edu

Weiwei Gong
Oracle Corporation
weiwei.gong@oracle.com

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

ABSTRACT

There has been a growing interest in using GPU to accelerate data analytics due to its massive parallelism and high memory bandwidth. The main constraint of using GPU for data analytics is the limited capacity of GPU memory.

Heterogeneous CPU-GPU query execution is a compelling approach to mitigate the limited GPU memory capacity and PCIe bandwidth. However, the design space of heterogeneous CPU-GPU query execution has not been fully explored. We aim to improve state-of-the-art CPU-GPU data analytics engine by optimizing *data placement* and *heterogeneous query execution*. First, we introduce a semantic-aware fine-grained caching policy which takes into account various aspects of the workload such as query semantics, data correlation, and query frequency when determining data placement between CPU and GPU. Second, we introduce a heterogeneous query executor which can fully exploit data in both CPU and GPU and coordinate query execution at a fine granularity. We integrate both solutions in Mordred, our novel hybrid CPU-GPU data analytics engine.

Evaluation on the Star Schema Benchmark shows that the semantic-aware caching policy can outperform the best traditional caching policy by up to 3×. Compared to existing GPU DBMSs, Mordred can outperform by an order of magnitude.

1 INTRODUCTION

Graphics Processing Units (GPU) have shown great potential in accelerating data analytics queries due to their massive computational power and high memory bandwidth compared to CPUs. GPU databases have been studied in both academic research [17, 40, 41, 47, 54] and developed as commercial products [1, 5, 8, 11, 12], demonstrating more than 10× speedup over the CPU counterparts. Today, the wider adoption of GPU databases is limited by the small

GPU memory capacity (up to 80 GB) such that only small workloads can fit in and fully leverage GPU acceleration.

Existing solutions address GPU memory limitation using two different approaches. The first approach is to *transfer data to GPU on demand* through PCIe when a query accesses data that is not in GPU. This solution is straightforward and has been adopted in both commercial systems [8] and research projects [40, 52, 54]. However, one downside is the potentially significant data traffic over PCIe, which can become a new performance bottleneck [47]. In certain cases, the PCIe bottleneck may cause the system to even under-perform a highly-optimized CPU database [47]. Although the interconnect bandwidth will increase through new hardware technologies like NVLink [7] and CXL [3], it will likely remain much lower than GPU memory bandwidth in the foreseeable future.

To mitigate the limited GPU memory capacity and PCIe bandwidth, a second approach is to *leverage both CPU and GPU for heterogeneous query processing* [17, 27]. Such a design can fully exploit the computational power of both devices and avoid excessive data transfer over PCIe by running certain sub-queries in CPU. However, this performance advantage comes at a cost of higher design complexity for data placement and heterogeneous query execution across devices; existing designs have not fully explored the design space and achieved suboptimal performance in many cases. For example, GDB [33] and HetExchange [27] do not have a data placement strategy for query execution. CoGaDB [17] and Ocelot [37] adopt primitive replacement policies by simply placing the most frequently or recently used columns in the GPU memory.

In this paper, we aim to improve existing heterogeneous GPU and CPU databases by optimizing both data placement policies and heterogeneous query execution. We develop a heterogeneous analytical engine called ***Mordred***, which innovates mainly in the following two aspects:

Data Placement. Similar to prior work [17], Mordred models data placement as a caching problem where a subset of data is cached in GPU memory and the CPU maintains a copy of the entire database. Different from prior work, however, Mordred manages caching at a fine granularity (i.e., sub-column) and uses a novel semantic-aware cache replacement policy. The new policy considers various aspects of the workload such as the query semantics, data correlation, query frequency, etc. Mordred uses a lightweight cost-based performance model to estimate the benefit of caching, guiding the decision of replacement. For example, Mordred prioritizes the caching of segments that are part of joins over filters. The optimized cache replacement policy can lead to 3× speedup compared to the best prior baseline we compared against.

Heterogeneous Query Execution. Caching data at a fine granularity in GPU presents new challenges during query execution. Ideally, a heterogeneous query executor should exploit data in

both devices and coordinate query execution at a fine granularity, which prior systems could not achieve. In Mordred, we support fine-grained heterogeneous execution by introducing *segment-level query plan* — Mordred allows different segments of a column to execute different query plans depending on the segments' location. To further improve performance, Mordred also adopts various optimization techniques (including late materialization, operator pipelining, segment skipping, and lightweight memory allocation) to reduce the traffic through PCIe or device memory.

In summary, this paper makes the following contributions:

- We develop a semantic-aware fine-grained caching policy for heterogeneous CPU and GPU databases. The policy takes into account query semantics, data correlation, and query frequency to determine data placement.
- We develop a fine-grained heterogeneous CPU-GPU execution engine which converts a query plan into a segment-level query plan to be executed in parallel on both CPU and GPU.
- We build Mordred, a heterogeneous CPU-GPU analytics engine based on fine-grained semantic-aware caching and heterogeneous query execution; Mordred includes various performance optimizations. The source code will be made publicly available.
- We conduct a detailed evaluation and demonstrate that *semantic-aware caching* can outperform the best traditional caching policy by 3×. Mordred can outperform the best-prior GPU DBMSs by an order of magnitude.

The rest of the paper is organized as follows: We discuss the background in Section 2. Section 3 describes the fine-grained semantic-aware caching policy. Section 4 presents the heterogeneous CPU-GPU query execution engine and its optimizations. Section 5 describes Mordred's system architecture and some implementation details. Section 6 evaluates the performance of Mordred. Section 7 discusses related work and Section 8 concludes the paper.

2 BACKGROUND

In this section, we describe the basics of GPU architecture and past systems that execute queries on GPUs.

2.1 GPU Architecture

GPU data analytics applications are typically bounded by the bandwidth of global and/or the shared memory [47, 54]. The *global memory* is at the bottom of the GPU memory hierarchy and is typically implemented using high bandwidth memory (HBM). A modern GPU can have up to 80 GB of global memory with 2 TB/s bandwidth [6]. The basic compute unit is called *streaming multiprocessors* (SM). Each SM has a fixed set of registers and a shared memory (SMEM) that is accessed by all cores in the SM. Accesses to the global memory can be cached in the L1 and L2 caches. L1 cache is local to an SM and the L2 cache is shared by all SMs.

The GPU programming models (e.g., CUDA [2], HIP [4], and OpenCL [9]) group threads into collections of 32 to 1024 threads called a thread block. Each thread block executes on one SM. A thread block is further divided into groups of 32 threads called *warps*. Threads in the same warp execute the same instruction stream following the Single Instruction Multiple Threads (SIMT) model. A warp coalesces memory accesses to neighboring memory addresses to reduce memory traffic.

2.2 GPU for Data Analytics

Previous works have shown great speedup for data analytics in GPU. There are three design categories in existing GPU data analytics systems: (1) GPU as the primary execution engine, (2) GPU as a coprocessor, and (3) Heterogeneous CPU-GPU query execution.

In the first category, all or a significant portion of the working sets are stored in one or multiple GPUs [1, 5, 8, 11, 12, 47]. These designs are limited by GPU memory capacity, which can be smaller than the data set size. To fit more data into GPU, some prior works [1, 5, 8, 11, 12] use multiple GPUs for larger aggregated memory.

The second design category treats GPU as an accelerator in a coprocessor model [30, 40, 52, 54]. In these systems, the data mainly resides on the CPU and is transferred to GPU on demand during query execution. Several systems in the first category adapt to this model once the data does not fit in GPU. Systems in this category do not suffer from limited GPU memory capacity, but the excessive data movement over PCIe may become a performance bottleneck due to its limited bandwidth.

The third design category addresses the GPU memory capacity constraint by using both CPU and GPU for query execution [17, 27, 33, 36, 37, 39]. By executing the query partially on the CPU, we could avoid excessive data transfer through PCIe as in the coprocessor model. This paper focuses on this category. We will discuss related works belonging to each category in greater detail in Section 7.

2.3 Crystal Library

We build Mordred on top of Crystal [47], which is a library of CUDA device functions that execute analytic queries fully on GPU. Crystal develops the idea of *tile-based execution model*. Instead of viewing each thread as an independent execution unit, it views a thread block as the basic execution unit with each thread block processing a tile of entries at a time. A thread block holds a large group of elements (called a *tile*) collectively in shared memory. Typically, one tile consists of 512 entries. The key benefit of this model is that after a tile is loaded into the shared memory, subsequent passes over the tile will be served directly from shared memory, avoiding multiple round-trips to global memory. Therefore, Crystal can pipeline consecutive operators executed on GPU in a single kernel call. Crystal could execute analytics queries close to memory bandwidth speed.

Crystal also provides a cost model to accurately estimate the query runtime in CPU and GPU. The cost model assumes that the queries can always saturate the memory bandwidth and derives the execution time from the data traffic to the CPU/GPU memory. Mordred borrows the same cost model in Crystal to estimate the query runtime as part of its replacement policy which we will discuss in Section 3.2.2.

3 DATA PLACEMENT

Mordred treats data placement as a caching problem following previous work [17] — the complete data set resides in CPU memory and a mirrored subset of data is cached in GPU. Compared to the alternative design that partitions data into disjoint sets across CPU and GPU, maintaining a copy of all the data in CPU allows for more flexible query scheduling — the CPU can process a query if the GPU is occupied with other tasks. It also allows the CPU to reconstruct results to reduce PCIe traffic, as will be discussed in Section 4.
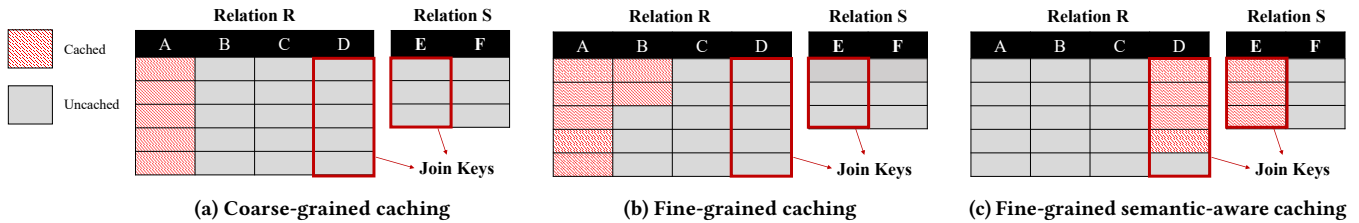
Figure 1: Illustration of Different Caching Policies — The example assumes a cache size of seven data segments.

A key design decision with this model is to decide which data to cache in GPU memory. Previous work [17, 37] explored caching at column granularity using least-recently used (LRU) and least-frequently used (LFU) replacement policies. We observe that such a design cannot optimally capture the benefit of GPU acceleration. In Section 3.1, we argue (1) how a sub-column fine-grained policy can improve caching efficiency and (2) how semantic-aware replacement leads to better performance. Then in Section 3.2 we explain the proposed fine-grained semantic-aware caching policy in detail.

3.1 Motivation

3.1.1 A Case for Fine-Grained Caching.
Caching data in column granularity has several problems. First, it is prone to fragmentation where a portion of the cache is empty due to the lack of space to fit in another column. This prevents us from making use of the full capacity of GPU memory. Second, column granularity caching cannot capture access skewness within a column. In real workloads, data accesses are often nonuniform, e.g., recent data is more frequently accessed than older data. In such cases, hotter data within the same column should be prioritized in caching, which cannot be captured in column-granularity caching.

Figures 1a and 1b illustrate the difference between coarse-grained (i.e., column granularity) and fine-grained (i.e., sub-column granularity) caching. Each column is split into *segments* with the same size and we assume a cache size of seven segments. We use the term *segment* to refer to sub-column for the rest of the paper. Column-granularity caching (Figure 1a) caches a single column of table A and leaves the remaining two cache slots empty since no column can fit in. In contrast, fine-grained caching can utilize all seven cache slots. Fine-grained caching does introduce new complexity in query execution; we will describe Mordred's solution in Section 4.

3.1.2 A Case for Semantic-Aware Caching.
One way to implement fine-grained caching is to use LRU or LFU on segments rather than columns. However, such semantic-agnostic replacement policies may not be able to accurately identify data that benefits the most from GPU acceleration. For example, join operators are computationally complex and comprise significant execution time of queries. Therefore, data that participates in joins should be cached with higher priority. Such semantic-awareness is not exploited in conventional LRU and LFU.

Furthermore, semantic-aware caching should consider the correlation between multiple columns when deciding what data to cache. In particular, some operators can execute on GPU only if all the involved columns are cached simultaneously. One example is a join operator, where both join keys should be cached. Other examples include columns that are involved in the same filtering predicate or the same aggregation function.

Figures 1b and 1c illustrate the difference between naive fine-grained caching and semantic-aware fine-grained caching. In this example, the DBMS identifies that caching segments in the join columns leads to higher speedup over other segments.

3.2 Semantic-Aware Fine-Grained Caching

This section describes the proposed semantic-aware fine-grained caching policy. We extend conventional LFU with *weighted frequency counters*, where the weight reflects the potential benefits of caching the segment and is derived using a cost model. The cost model captures (1) the relative speedup of caching a segment and (2) the correlation among segments from different columns. Section 3.2.1 describes the general caching framework and Section 3.2.2 describes the cost model for the weighted frequency counters.

3.2.1 Cache Replacement Policy.
The cache replacement policy is based on conventional LFU where each data segment is associated with a frequency counter, which is incremented when the segment is accessed by a query. The cache stores segments with the largest frequency counters. Different from the baseline LFU that increments the counter always by 1 for each access, the replacement policy in Mordred increments it with a weight that is calculated based on semantic information.

Algorithm 1 demonstrates the semantic-aware weight update algorithm in Mordred. We call *UpdateWeightedFreqCounter*() after accessing each segment S to update its weight. The algorithm first estimates the query runtime $RT_{uncached}$ if segment S is not cached in GPU (line 2). The runtime estimation is based on the current content in GPU (i.e., *cached_segments*) with S removed. The function *estimateQueryRuntime*() uses a simple model to predict runtime with the assumption that the CPU/GPU memory and PCIe bandwidth are the performance bottleneck; we will describe details of the cost model in Section 3.2.2. Then, the algorithm estimates the query runtime RT_{cached} if S and *all segments correlated with S* are cached in GPU (line 3), besides the currently cached segments.

We use the difference between $RT_{uncached}$ and RT_{cached} to represent the *weight* (line 4). The weight is added to the weighted frequency counter of S (line 5). Furthermore, it is also evenly distributed to all the correlated segments of S (line 6–7) through dividing the weight by |*correlated_segments*|. This means more correlated segments leads to smaller weight assigned to each, which bounds the total weight incremented by accessing a segment.

The precise definition of *correlated segments* for segment S depends on the operator being performed. So far, Mordred considers three operators: selection, join, and group-by aggregation.

For selection, we consider segments correlated if they are (1) involved in a predicate where the attributes cannot be separated by logic "and" or "or" and (2) correspond to the same set of rows.

Algorithm 1: Update the *weighted frequency counter* for segment S

1   **UpdateWeightedFreqCounter**(*segment S*)
     # estimate query runtime when S is not cached.
2     $RT_{uncached}$ = **estimateQueryRuntime**(*cached_segments \ S*)
     # estimate query runtime when S and segments correlated with S are cached.
3   RT_{cached} = **estimateQueryRuntime**(*cached_segments* ∪ S ∪ *correlated_segments*)
4   *weight* = $RT_{uncached} - RT_{cached}$
5   *S.weighted_freq_counter* += *weight*
6   **for** *C* in *correlated_segments* **do**
       # evenly distribute weight to all segments correlated with S
7   *C.weighted_freq_counter* += *weight* / |*correlated_segments*|

In such a case, all the segments in this correlation set will be incremented by the same weight.

For hash join, we make a key observation that GPU acceleration is effective only if at least one column (i.e., the build column) is completely cached. Therefore, we consider the build and probe relations differently. Specifically, we perform Algorithm 1 on every segment S in the probe column (i.e., typically the larger one) and consider all segments in the build column as correlated.

For group-by aggregation, a correlation exists between the aggregation column and the grouping key column. We consider two cases. First, if aggregation and grouping columns are all in the same table, we perform Algorithm 1 on every segment S in the aggregation column and consider segments in the same set of rows as correlated. Second, if aggregation and grouping columns are in different tables (i.e., they are joined together), we perform Algorithm 1 for every segment S in the aggregation column and consider all segments in the grouping column as correlated.

3.2.2 Cost Models.

We now explain how the *estimateQueryRuntime*() function in Algorithm 1 works. In particular, we use the cost model presented in Crystal [47] to estimate the execution time of subqueries in both CPU and GPU. The cost model assumes that the queries can saturate the memory bandwidth and derives the execution time from memory traffic. The accuracy of the model has been verified in Crystal on simple operators. We extend the model to support more complex queries and to support PCIe. While the model may not be as precise in this more complex environment, we find the accuracy to be acceptable for the purposes of cache replacement policy.

In particular, we model filtering cost as follows:

$$filter\ runtime = \frac{size(int) \times N}{B_r} + \frac{size(int) \times N \times \sigma}{B_w}$$

where N is the cardinality of input segments, σ is the predicate selectivity, B_r and B_w are read and write memory bandwidth, respectively. The first term of the equation is the time taken to scan the input column and the second term is the time taken to write the matched entries to the output array. This and the following equations assume relations of integers.

For hash join, the probe runtime is modeled as follows:

$$probe\ runtime = \frac{size(int) \times N}{B_r} + (1-\pi)\frac{N \times C}{B_r} + \frac{size(int) \times N \times \sigma}{B_w}$$

where the extra parameter C is the cache line size, π is the probability the accessed cache line is in the last level cache. The first term is the time taken to scan the probe relation from memory, the second term is the time taken for probing the hash table, and the third term is the time taken to write the matched entries to the output array. Other database operations such as hash aggregation and sorting can be modeled in a similar fashion.

In heterogeneous query execution, there is an extra cost for materialization, merging, and data transfer between CPU and GPU. We model the cost of data transfer as follows:

$$data\ transfer\ time = \frac{size(int) \times N}{B_{pci}}$$

where B_{pci} is the interconnect bandwidth. This equation represents the time taken to transfer N integers over PCIe.

Materialization is the process of reconstructing tuples from intermediate results expressed as row IDs (more details in Section 4.2.1). We model the materialization cost as follows:

$$materialization\ time = \frac{size(int) \times N}{B_r} + \frac{N \times C}{B_r}$$

where the first term is the time taken to scan the row IDs from the intermediate results and the second term is the time taken to reconstruct the tuples from the row IDs.

Finally, the following equation models the cost of merging the final results from CPU and GPU:

$$merging\ time = \frac{2 \times size(int) \times N}{B_r} + \frac{size(int) \times N}{B_w}$$

where the first term is the time taken to scan the two columns to be merged and the second term is the time taken to write the results.

These equations are used as building blocks to estimate the query runtime in the *estimateQueryRuntime*() function. The cost model is lightweight such that its evaluation incurs negligible overhead.

3.2.3 Example of Semantic-Aware Caching.

```
Qx: SELECT R.B, SUM(R.A) FROM R,S
    WHERE R.D = S.E

Qy: SELECT R.A AND R.B FROM R
```

We present an example to illustrate how semantic-aware caching works. Consider the schema in Figure 1 and performing replacement after executing Qx and Qy. Assume a cache size of seven segments.

Coarse-grained LFU will cache either column $R.A$ or $R.B$ since both columns are used by both queries and the cache has space for only one column, as illustrated in Figure 1a. In contrast, fine-grain LFU (Figure 1b) can fill in the two empty cache slots with two segments from $R.B$. Different from both LFU policies, the semantic-aware policy caches segments that maximize performance using the cost model in Section 3.2.2. Therefore, even though $R.D$ and $S.E$ are used only by Qx, our policy assigns higher weight for segments in both columns since the GPU offers higher speedup for join over filtering operations. The caching decision is shown in Figure 1c.

4 HETEROGENEOUS QUERY EXECUTION

A new challenge introduced by the fine-grained caching policy is the extra complexity of query execution. It is possible that only a subset of data required by an operator exists in GPU memory so that the entire operator cannot directly run on GPU. Ideally, a heterogeneous query executor should fully exploit the data in GPU
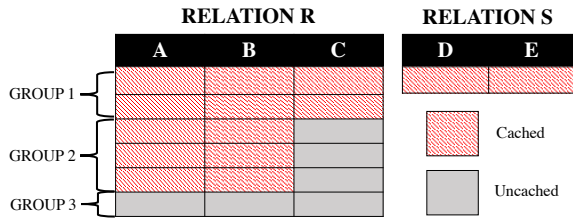
Figure 2: Example of Segment Grouping.

Figure 3: Example of Segment-level Query Plan.

and coordinate query execution across the two devices at segment granularity instead of column granularity. While some existing systems [8] have adopted fine-grain caching, they still execute the entire query in GPU and transfer the uncached data to GPU during query execution, leaving the available CPU cores unutilized.

In Mordred, we exploit both intra-device parallelism within CPU/GPU and inter-device parallelism across CPU/GPU. Our heterogeneous query executor targets the following three goals:

Goal 1: Minimize inter-device data transfer. Currently, the interconnect bandwidth (i.e., PCIe) between CPU and GPU is very limited. Transferring too much data through this interconnect could throttle system performance.

Goal 2: Minimize CPU/GPU memory traffic. Data analytics applications are memory-bound. Reducing the data traffic to both CPU and GPU memory leads to more efficient query execution.

Goal 3: Fully exploit parallelism in both CPU and GPU. Ideally, we want to utilize all the available computational power in both multicore CPU and GPU during query execution.

4.1 Segment-Level Query Execution

A line of previous research [18, 19, 23–25, 39] has studied the problem of heterogeneous query execution in column granularity. In this section, we discuss how we address heterogeneous query execution in segment granularity in hybrid CPU and GPU systems.

4.1.1 Operator Placement.

Previous works have discussed operator placement strategies for CPU-GPU systems. In particular, Breß et al. [19] proposed a *data-driven operator placement* heuristic, where an operator is pushed to where the input columns reside. The operator is executed in GPU only if *all* the input columns are cached in GPU, otherwise, the operator is executed in CPU. This heuristic has been shown to outperform cost-based optimizer while being more lightweight.

In Mordred, we adopt the *data-driven operator placement* heuristic but apply it at segment granularity — instead of executing an operator in the device where the entire input *columns* reside, Mordred executes portions of the operator in the device where all the input *segments* reside. This means a single operator can be split to run in both CPU and GPU depending on the location of input segments. We call the resulting plan a *segment-level query plan*.

In Mordred, every operator can be split between CPU and GPU. For filter, each partition can be filtered independently in either CPU or GPU. For hash join, we require the build column to be entirely cached and split the probe operator across the two devices. For group-by, if the corresponding grouping and aggregation segments are cached, we perform group-by on them in GPU and send the results back to CPU.
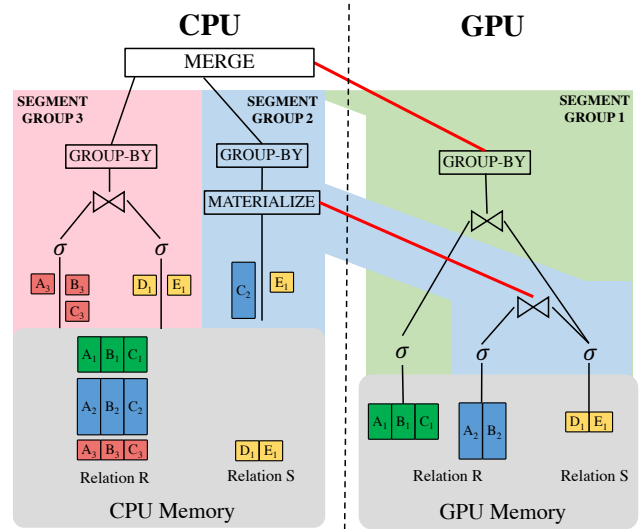
4.1.2 Segment-Level Query Plan.

Given a particular data placement determined by the caching policy, Mordred puts input segments into groups and executes them in parallel. In particular, Mordred applies the *data-driven operator placement* heuristic to determine the execution plan for each segment and puts segments with the same plan into the same *segment group*. The query optimizer and executor work in segment-granularity.

Figure 2 demonstrates one example of segment grouping where relation R is partially cached and relation S is fully cached in GPU. One obvious grouping strategy is to split relation R into three segment groups — Group 1 comprises the first two rows of segments, Group 2 comprises 3rd to 5th rows of segments, and Group 3 comprises the last row of segment. Note that the grouping strategy may differ depending on the query plan. For example, if a query accesses only columns A and B in relation R, then Groups 1 and 2 above can be merged into a single large group. This is because both groups have identical execution plan for such a query.

After the grouping phase, each segment group is executed in parallel according to its execution plan. Finally, after all segment groups finish execution, all results will be sent back to CPU and be merged. The merge operation is lightweight. It simply combines the aggregation results from different segment groups together. In Section 6.3, we show that merging contributes to < 2.5% of total query execution time in our workload. When the query result is very large, however, merging could potentially be a bottleneck. Such a case is partially captured in the cost model (Section 3.2.2), which will choose not to cache the data and run the query in CPU. A more general solution to the problem requires a full-fledged heterogeneous query optimizer, which we defer to future work.

4.1.3 Example of Query Execution.

```
Q0: SELECT S.D, SUM(R.C) FROM R,S
    WHERE R.B = S.D AND R.A > 10 AND S.E > 20
    GROUP BY S.E
```

We present an example to illustrate how segment-level query execution works using the query $Q0$ shown above with the table schema and cache layout in Figure 2. In this example, R is the probe relation

and S is the build relation. Since the query accesses all the three columns in relation R, we split R into three segment groups.

Figure 3 shows a physical query plan generated by Mordred. Activities in CPU and GPU are shown in the left and right halves of the figure, respectively. The red lines crossing the device boundary indicate the transfer of intermediate results across PCIe. The sub-query plans for individual segment group are shaded with different background colors. Below, we explain the sub-query plan for each segment group individually.

Segment Group 1: All segments in this group are fully cached and thus the entire sub-query is executed in GPU. After the group-by operator, the results are sent back to CPU.

Segment Group 2: Segments in this group are partially cached. Specifically, segments used for filter (A_2, E_1) and join (B_2, D_1) are in GPU but segments used for group-by (C_2) reside in CPU. The join can still be performed in GPU; in fact, the hash table on relation S can be reused for segment group 2 and segment group 1.

To minimize the data transfer across PCIe, the result of the join is expressed using row ID pairs between the two tables, i.e., ($RowID_R$, $RowID_S$), to indicate the rows that join. Using the row ID pairs, the CPU will *materialize* the join results by reading the values from C_2 and E_1. The materialized results are sent to the group-by operator and the output is merged with other sub-queries' results.

Segment Group 3: In this group, no segment from relation R is cached in GPU. Therefore, join cannot be performed in GPU. The entire sub-query is executed in CPU with the final output merged with the other sub-queries' results.

4.2 Other Performance Optimizations

Memory and PCIe traffic is typically the performance bottleneck in data analytics applications [47]. In this section, we describe other optimization techniques in Mordred to further reduce data traffic.

4.2.1 Late materialization.
With heterogeneous query execution, there are often cases that require the transfer of intermediate results of sub-queries from one device to another. Transferring the whole intermediate relation across PCIe could be expensive. Previous CPU-based columnar databases used the *late materialization* strategy to reduce data transfer [13], where the intermediate relation can be expressed in the form of row IDs. The receiver side can then reconstruct the tuples. We implement this technique in Mordred to significantly reduce the amount of data transfer.

Another benefit of late materialization is that we can execute an operator in GPU with the minimum number of input columns. For example, joining two relations in GPU requires only the two join key columns to be present in GPU. The rest of the attributes can be reconstructed in CPU using late materialization.

4.2.2 Operator Pipelining.
Previous work [27, 47] has developed an optimization technique for GPU query execution by pipelining operators into a single kernel. Operator pipelining can avoid storing intermediate results in memory after each operator, and only materialize the results at the end of each pipeline. Crystal [47] has shown that pipelining operators in GPU query execution can further improve the performance by storing the intermediate result from each operator in the shared
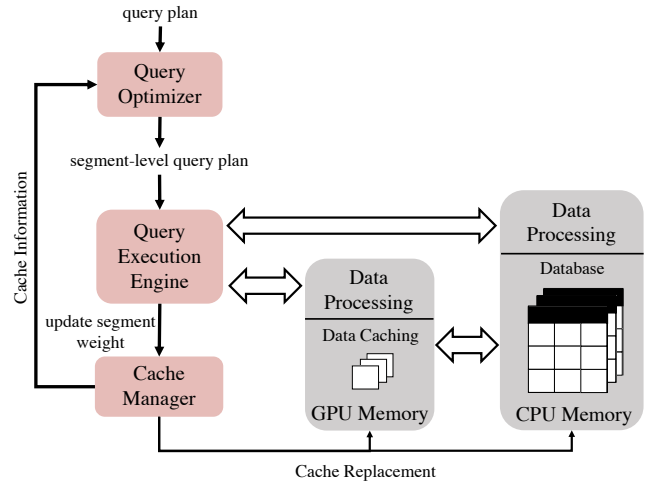

Figure 4: Mordred System Overview

memory, which can reduce the total round trips to the global memory. We apply this optimization to Mordred by always pipelining consecutive operators on the same device whenever possible.

4.2.3 Segment Skipping.
Many queries in OLAP workload only access a portion of the data after predicate evaluation. For example, business intelligence queries often access only data in a specific period of time. Minmax pruning is a well-known technique to reduce the amount of data being scanned by the query [10]. The technique maintains per-segment minimum and the maximum values in the segment. During query execution, an entire segment can be skipped if the predicate cannot possibly be satisfied based on the min and max values.

Typically, minmax pruning is applied only to predicate evaluation. In Mordred, we extend it to filter during join operators as well in order to further reduce the amount of data that needs to be cached in GPU.

```
Q1: SELECT R.y FROM R
    WHERE  R.datekey = S.datekey AND S.year > 1997
```

We demonstrate this using an example query $Q1$ above. R is the probe relation and S is the build relation. In the build phase, Mordred maintains the min and max values of all keys (S.datekey) in the hash table. After the build phase is finished, we use these min-max values to skip segments in the probe relation. In particular, segments from R can be skipped by looking at the min-max of the hash table and the min-max of the segment in R.datekey. This optimization allows us to further reduce memory and PCIe traffic and identify the truly hot segments when performing cache replacement.

5 SYSTEMS INTEGRATION

This section describes the implementation of Mordred, a hybrid CPU-GPU DBMS with *fine-grained semantic-aware caching* and *segment-level query execution*. Figure 4 shows the overview of Mordred, which consists of three main modules that will be described in the following subsections.

5.1 Cache Manager

The Cache Manager performs periodic data replacement in GPU memory based on the caching policy described in Section 3. The segment size is user defined and by default 1,048,576 records (2^{20}).

Mordred divides GPU memory into the following two regions.

Data Caching Region: The data caching region handles the caching of raw data in segment granularity. Data in this region is managed by the cache manager, using the *semantic-aware caching policy* described in Section 3.2.

Data Processing Region: The data processing region stores intermediate data (e.g., hash table, intermediate query result, etc.) during execution. Since frequent memory allocation in GPU is expensive, we develop a lightweight memory allocation strategy.

Specifically, when the cache manager is initialized, the data processing region is preallocated. The cache manager maintains a pointer to the starting address of the data processing region. Whenever a memory allocation request arrives, the cache manager returns the address of the pointer and advances it by the size of the allocated region. After a query is executed, we reset the pointer to the starting address of the data processing region. The same lightweight memory allocation strategy is also applied in CPU.

The cache manager also handles metadata management. Metadata involves statistics information of each segment (e.g., access timestamp and weighted frequency counters) and the metadata required during query execution (e.g., min-max of each segment, offset of cached segments, etc.). We use bitmap to mark if the segment is cached in GPU. We use a hash table to track the offset and the statistics of each segment (min-max, counter, etc). Finally, we use a free list to track available segments in GPU. The whole metadata management in Mordred is handled by the CPU.

5.2 Query Optimizer

The query optimizer module converts a query plan into a *segment-level query plan.* Currently, we take the query plan from Crystal [47], which is already highly optimized for GPU.

Taken the input query plan, Mordred performs segment grouping as described in Section 4.1 based on the data-driven operator placement heuristic. Meanwhile, Mordred also reorders the operators based on where they will be executed. For example, operators that will be executed on GPU will be clustered together. We do this to avoid *ping-pong effect* where the execution plan will go back and forth between CPU and GPU, causing excessive data transfer. Segment skipping as described in Section 4.2.3 is also partially performed in this stage by the query optimizer.

For simplicity, our optimization is currently heuristic-based and focuses on minimizing interconnect traffic. As interconnect bandwidth improves, a cost-based optimizer that takes into account the interconnect bandwidth might outperform our current design. We leave this to future work.

5.3 Query Execution Engine

The query execution engine executes segment-level query plan generated by the query optimizer. Specifically, the engine executes the sub-query plan for each segment group in parallel and merges the final results. Finally, the execution engine notifies the cache manager to update the weight of segments following Algorithm 1.

To launch operators in CPU, Mordred uses IntelTBB parallel programming library. For GPU kernel implementation, we use Crystal [47], a CUDA-based library for query execution. We add new functions to express *late-materialization* in Crystal.

During execution, each segment group is assigned to a dedicated CPU thread which holds information about the execution plan. These CPU threads start the execution of all segment groups in parallel. Each thread will launch either CPU kernels or GPU kernels to execute operators according to its corresponding execution plan. Since each CPU kernel will also be executed using multiple threads, Mordred leaves the thread assignment to the thread scheduler of the parallel programming framework (i.e., Intel TBB).

Currently, we implement a compound kernel for each pipeline to enable operator pipelining (cf. Section 4.2.2). We leave automatic code generation of pipelined kernels as part of our future work.

6 EVALUATION

In this section, we report the performance of Mordred. The section will answer the following key questions:

- How does fine-grain semantic-aware caching perform compared to traditional caching policies?
- How does segment-level query execution improve the performance of heterogeneous query processing?
- How much performance improvement is achieved through various optimizations (i.e., segment skipping, operator pipelining, and late materialization) implemented in Mordred?
- How does Mordred perform compared to existing heterogeneous CPU/GPU DBMSs?

6.1 Experimental Setup

Hardware configuration: We use a virtual machine instance in Oracle Cloud with NVIDIA V100 GPU connected to Intel Xeon Platinum 8167M CPU via PCIe3. The Nvidia V100 GPU has 16 GB of HBM2 memory with read/write bandwidth of 880 GBps. The Intel Xeon Platinum CPU has 24 virtual cores and 180 GB DRAM. The bidirectional PCIe bandwidth is 12.8 GBps. The system is running on Ubuntu 18.04 and the GPU instance runs CUDA 11.2.

Benchmark: For our experiments, we use the *Star Schema Benchmark* (SSB) [43] which has been widely used in various data analytics research studies [30, 40, 51, 54]. We use Scale Factor 40 (i.e., 25 GB data set) in all experiments unless otherwise stated. To enable efficient query execution in GPU, we dictionary encode the string columns into integers prior to data loading and manually rewrite the queries to directly reference the dictionary-encoded value. Therefore, we ensure that all column entries are 4-byte in value. We also sort the fact table by the `orderdate` column to enable the *segment skipping* optimization. In our evaluation, the entire data set is loaded to CPU memory before each experiment starts.

Measurement: Before each experiment, we first warm up the GPU memory by running 100 random queries from the SSB benchmark and then perform a replacement to populate the cache. We run 500 queries in each experiment unless otherwise stated and perform replacement after every 50 queries. We found the replacement cost to be negligible (< 1.5%) and thus do not include it in our measurement. For each measurement, we repeat the experiment 3 times and report the average results.
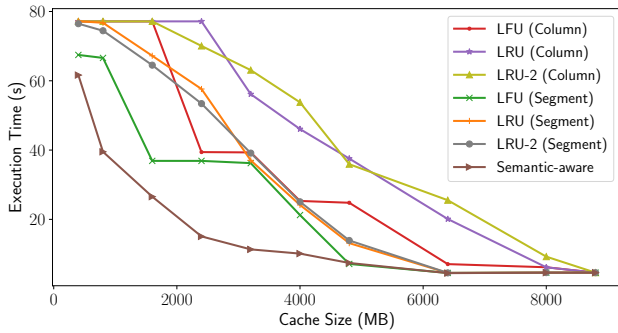
Figure 5: Execution Time of Various Caching Policies with Different Cache Size (Uniform distribution with $\theta = 0$)

Figure 6: Memory Traffic Breakdown for Each Caching Policy

6.2 Comparison with Other Caching Policies

This subsection evaluates the performance of *fine-grained semantic-aware caching policy*. Specifically, we will compare the performance of seven different caching policies:

- **LRU (Column):** This policy maintains the access timestamp for each column and caches columns with the latest timestamps. This policy is used by [17, 37].
- **LFU (Column):** This policy maintains a frequency counter for each column and cache columns with the largest counters. This policy is used by [17].
- **LRU-K (Column):** This policy maintains the backward K-distance for each column — the distance backward to the Kth most recent reference to the column. Columns with the smallest backward K-distance will be cached. We use $K = 2$ in our experiments.
- **LRU (Segment):** Similar to LRU (Column) but timestamps are maintained for segments instead of columns.
- **LFU (Segment):** Similar to LFU (Column) but frequency counters are maintained for segments instead of columns.
- **LRU-K (Segment):** Similar to LRU-K (Column) but the backward K-distance are maintained for segments instead of columns.
- **Semantic-aware:** The segment-granularity semantic-aware caching policy described in Section 3.

6.2.1 Performance on Standard SSB.

In this experiment, we sweep the GPU cache size and measure the performance of different caching policies when running SSB queries. We sweep the cache size from 400 MB to 8.8 GB; all columns that are accessed by queries fit in an 8.8 GB cache. The query access distribution is uniform following the default configuration.

Figure 5 shows the result of our experiment. Overall, LFU-based schemes perform better than LRU-based schemes. LRU-2 performs similarly to LRU but slightly better for small cache sizes. For each policy, the fine-grained version always outperforms its coarse-grained counterpart. This is because the coarse-grained policy often suffers from fragmentation, causing a portion of the cache unused.

The semantic-aware caching policy outperforms all the other policies in all cache sizes, especially when the cache size is small (3–7× lower runtime). This is because the new policy can more accurately identify hot segments to cache leading to higher speedup.

Specifically, Figure 5 illustrates the limitation of conventional schemes. For example, in LFU (Segment) from the cache size of 1.8 GB to 3.6 GB, there is little performance improvement even though twice as much data has been cached. A closer investigation reveals
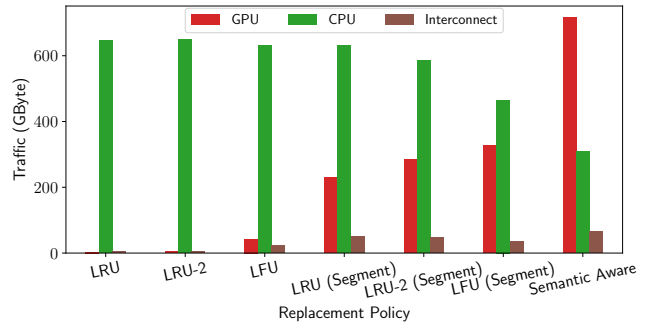
that the newly cached data comes from the lo_revenue column which is used only for group-by. Since the join is performed in CPU, the heuristics decides to execute group-by also on CPU to avoid excessive PCIe traffic, rendering the cache ineffective. This results in performance stall from 1.8 GB to 3.6 GB. The semantic-aware caching policy, in contrast, is aware of the query semantics and thus does not suffer from this performance stall.

6.2.2 Memory Traffic Breakdown.

To gain deeper insight on each caching policy, we compare the data traffic going through the CPU memory, GPU memory, and interconnect (PCIe) for each policy. We use the same setup as Section 6.2.1 with a cache size of 1.6 GB (which can hold 20% of all accessed columns).

Figure 6 shows the traffic breakdown for different caching policies. The interconnect traffic remains low across all policies; this is partly because of the *data-driven operator placement* heuristic (see Section 4.1) where an operator is executed on GPU only if the input data is cached. Column-granularity caching shows a very low GPU memory traffic and high CPU memory traffic. For these policies, only a single column from the fact table fits in the cache resulting in most of the queries completely being executed on the CPU. The fine-grain version of each policy always has higher GPU memory traffic and lower CPU memory traffic.This shows that with segment granularity caching, more work can be offloaded to the GPU.

Across all policies, semantic-aware caching has the highest GPU memory traffic and lowest CPU memory traffic. This is because the policy caches only critical segments which would greatly reduce the total CPU memory traffic. Since GPU has a larger cache line (128B) compared to CPU (64B), the GPU memory traffic can be much higher especially when random reads/writes are involved. However, since GPU memory has the highest bandwidth (10× of CPU memory bandwidth), this still results in better performance. Overall, our traffic breakdown from this experiment is aligned with the query performance results from Section 6.2.1.

6.2.3 Varying Query Access Pattern.

This experiment evaluates the performance of semantic-aware caching policy under varying query access distribution. To simulate nonuniform query access distribution, we incorporate skewness into the predicates of SSB queries. We pick the values following a Zipfian [32] distribution with a tunable skewness that is controlled by a parameter θ. Skewness is applied to the date predicate such that more recent data has a higher probability to be accessed by the query. A larger θ indicates higher skewness.
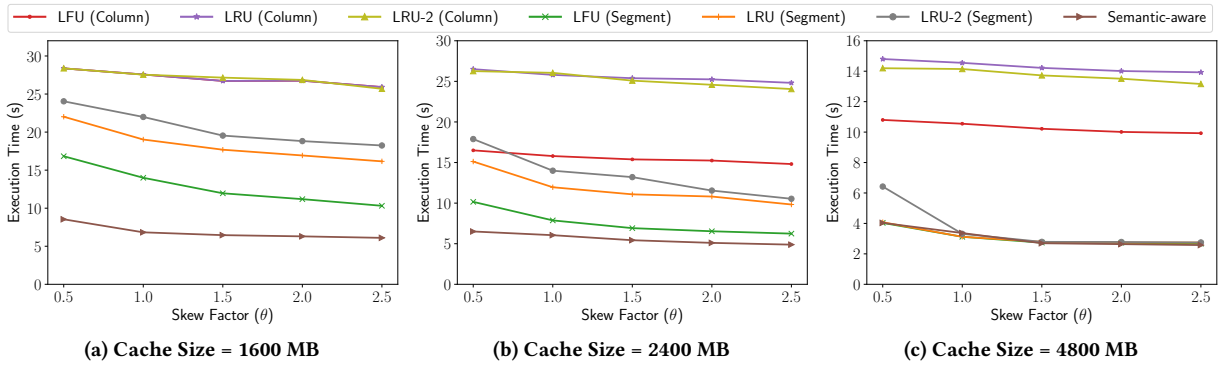
(a) Cache Size = 1600 MB　　　(b) Cache Size = 2400 MB　　　(c) Cache Size = 4800 MB

Figure 7: Execution Time of Various Caching Policies with Varying Query Access Distribution
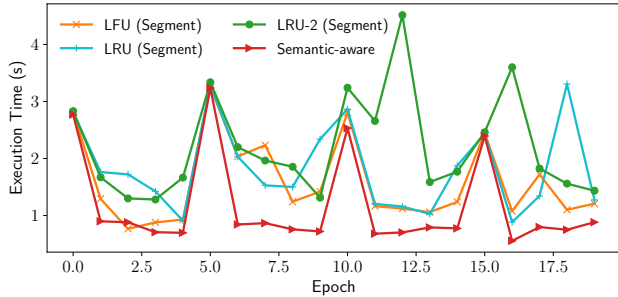
Figure 8: Execution Time of Various Caching Policies under Phase Changing Workload

Figure 9: Cached Columns in Different Replacement Policies

Figure 7 shows the execution time with three different cache sizes when we sweep the skew factor. When the cache size is small (Figure 7a), fine-grained caching is more sensitive to skewness than column-granularity caching. This is because for nonuniform query accesses, fine-grain policies can cache only the hot portion of the data. The higher the skew factor, the more accurate these policies can capture the hot portion. We can also see that LFU (Segment) performs better than LRU (Segment) and LRU-2 (Segment) in most cases. This shows that access frequency is better than access timestamp for capturing skewness in query distribution.

Across experiments in Figure 7, semantic-aware caching outperforms traditional caching policies. For large cache sizes (Figure 7c), the performance of LFU (Segment), LRU (Segment), and LRU-2 (Segment) are very close to semantic-aware caching. This is because the cache size is enough to fit in almost all the hot portion of the data. For smaller cache sizes (see Figure 7a), however, the performance gap is bigger since not all the hot portion can fit in. In this case, semantic-aware caching can more accurately identify critical segments that provide the most benefit from GPU acceleration.

6.2.4　Performance on Phase-Changing Workload.
This experiment evaluates different caching policies under a phase-changing workload. We incorporate skewness into the date predicate of SSB queries. We use a normal distribution with $\delta = 0.5$ years and a tunable mean controlled by parameter μ. After every 5 batches of queries (50 queries per batch), we shift the mean by 3 years. The results are shown in Figure 8.

For frequency-based policies (LFU and semantic-aware), we multiply the current weight by an aging factor (default is 0.5) when a new epoch starts. This would prioritize recent frequency information over history in the past. Our experiment shows that LFU-based
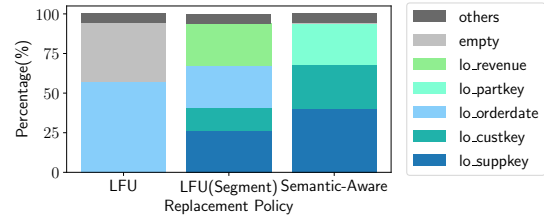
policy performs better than LRU and LRU-2. LRU and LRU-2 suffer from random performance spikes since the cache content depends on the last query in the previous batch, which can be random. If the last query accessed data from the cold data region, replacement can significantly degrade performance. Overall, our experiment shows that our policy adapts quicker and outperforms other schemes under a phase-changing workload.

6.2.5　Caching Statistics.
In this experiment, we show the content of the cache for each caching policy. We use a cache size of 1.6 GB (which can hold 20% of all accessed columns). Figure 9 shows the result of our experiment.

LFU (Column) policy can only cache a single column in the fact table (`lo_orderdate`) and leaves 37% of the cache unused. Column `lo_orderdate` is a foreign key to the `d_datekey` column from the `DATE` relation. Caching this column enables us to perform join against the `DATE` relation in GPU. However, the `DATE` table is not as selective as the other dimension tables. This could result in suboptimal performance since transferring the join result to CPU will be expensive.

LFU (Segment) policy caches a more diverse set of columns compared to the LFU (Column). It does not suffer from fragmentation and tends to cache the hot portion of the data. A large chunk of the cache, however, is still for `lo_revenue` which is only used in `GROUP BY` expression. `GROUP BY` is often lightweight and therefore should not be prioritized over caching columns used for `JOIN`.

Our semantic-aware caching prioritizes caching the foreign keys from the fact table (`lo_suppkey`, `lo_custkey`, `lo_partkey`). This often enables us to perform join with the `SUPPLIER`, `CUSTOMER`, and `PART` relations in GPU. Joins involving these relations are very selective. Therefore, caching segments from these columns is very beneficial since the join output transferred from GPU is usually small. This also leaves CPU with a much more lightweight execution over a smaller relation in the later stage of query execution.
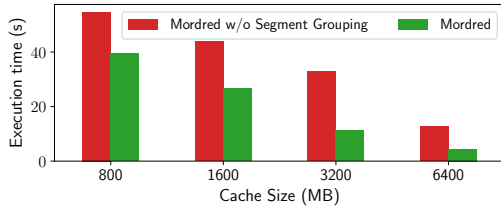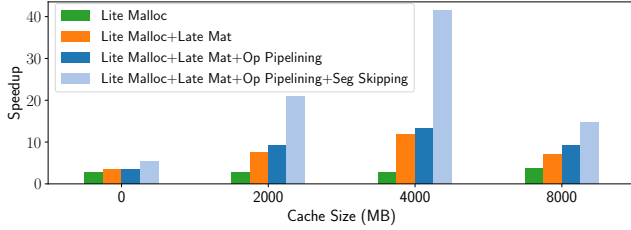
Figure 10: Impact of Segment Grouping in Mordred

Figure 11: Performance Speedup after Each Optimization

6.3 Evaluating Segment-level Query Execution

One important optimization in segment-level query execution is the grouping phase prior to query execution (see Section 4.1). In this section, we will evaluate the performance difference when we enable vs. disable segment grouping. Figure 10 shows the results with different cache sizes.

Without segment grouping, the execution engine will launch kernels in the granularity of segments instead of segment groups. This leads to an excessive number of kernels launched, which leads to performance degradation.

With segment grouping, all the segments with the same execution plan are grouped and executed with only a single kernel launch. We see from Figure 10 that segment grouping can speed up query execution by up to 3×. The gain increases as the cache size gets larger as more work can be offloaded to the GPU. The segment grouping optimization is novel in Mordred and has not been adopted by existing approach (i.e., HetExchange [27]).

We also measure the breakdown of each phase in segment-level query execution (grouping, execution, and merging). Our experiment shows that for a small cache size (0.8 GB), Mordred spends only 0.3% of the time for grouping, 99.2% of the time for execution, and 0.5% of the time for merging. For a large cache size (6.4 GB), the execution is much faster, and therefore merging and grouping contribute to larger portions of the runtime; Mordred spends 4% of the time for grouping, 93.6% of the time for execution, and 2.4% of the time for merging. In either case, merging and grouping are not performance bottlenecks.

6.4 Breakdown of Mordred Optimizations

6.4.1 Query Runtime Speedup.

We now measure the speedup from each optimization applied to Mordred. We evaluate the performance gain from four optimizations: (1) lightweight memory allocation (Section 5.1), (2) late materialization (Section 4.2.1), (3) operator pipelining (Section 4.2.2), and (4) segment skipping (Section 4.2.3). We perform the measurement for 4 different cache sizes as shown in Figure 11. The query will run completely in CPU when the cache size is 0 and will run completely
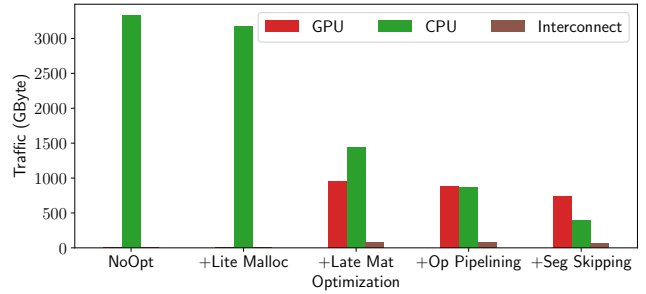

Figure 12: Memory Traffic after Each Optimization

in GPU when the cache size is 8 GB. For 2GB and 4GB cache sizes, Mordred caches data following the semantic-aware policy.

Across all cache sizes, lightweight memory allocation (`Lite Malloc`) consistently improves performance by around 3.3×, which is due to eliminating the expensive memory allocation operations.

Late Materialization (`Late Mat`) provides performance speedup by up to 3×. Without late materialization, operators are forced to be executed on the CPU since we need to cache the whole relation to execute an operator on the GPU. With late materialization, however, we can offload some operators to GPU or divide the operators between CPU and GPU which will provide significant speedup.

Operator pipelining (`Op Pipelining`) provides an extra speedup by up to 1.3×. Operator pipelining reduces the memory traffic during query execution Since our GPU implementation with Crystal is close to saturating the memory bandwidth, the benefit from operator pipelining is more significant in GPU.

Finally, segment skipping (`Seg Skipping`) provides another 1.6–3× speedup. Segment skipping reduces the amount of data being processed by the query. In the original SSB queries, segment skipping manages to skip 48% of the data across all the queries. The speedup is more significant when the data is partially cached in GPU. This is because it can more accurately identify data that benefits the most from GPU caching. Specifically, when segment skipping is not applied, a full column scan is often required during query execution. This causes the weight of all segments in the column to be incremented following Algorithm 1. When segment skipping is applied, however, the skipped segment will not get its weight incremented. This improves the accuracy of our semantic-aware caching and improves the overall the query performance.

6.4.2 Memory Traffic Breakdown.

To reveal deeper insight of the performance optimizations, we show the memory traffic breakdown after every optimization is applied. Figure 12 shows the breakdown with 1.6 GB cache size (which can hold 20% of all accessed columns).

After lightweight memory allocation, the traffic does not really affected since it just eliminates memory allocation operations. After late materialization, we can offload more work to the GPU. This reduces the CPU memory traffic and increases the interconnect and the GPU memory traffic. After operator pipelining, both CPU traffic and GPU traffic are reduced even further since the intermediate results are not materialized. The reduction in CPU traffic is more significant since the cache size is small and therefore most of the operators are executed in CPU. Finally, segment skipping reduces the CPU traffic and GPU traffic by reducing the total amount of data being processed by the queries.
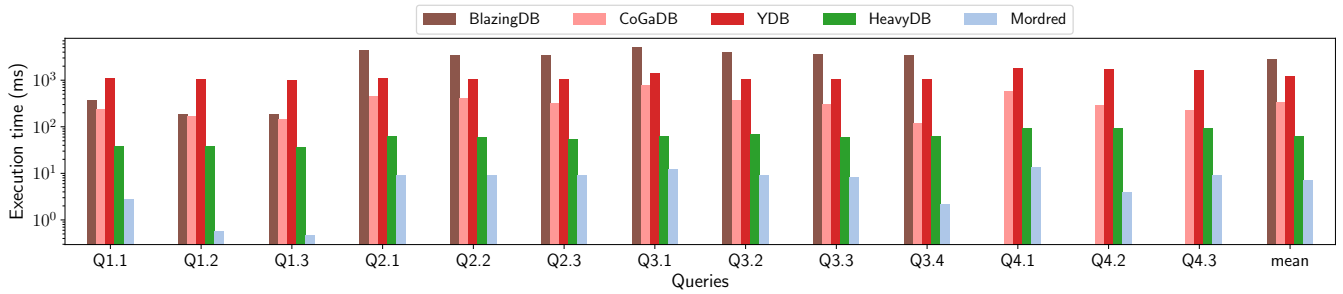
Figure 13: SSB Query Performance of Different CPU/GPU DBMS (Data fits in GPU)
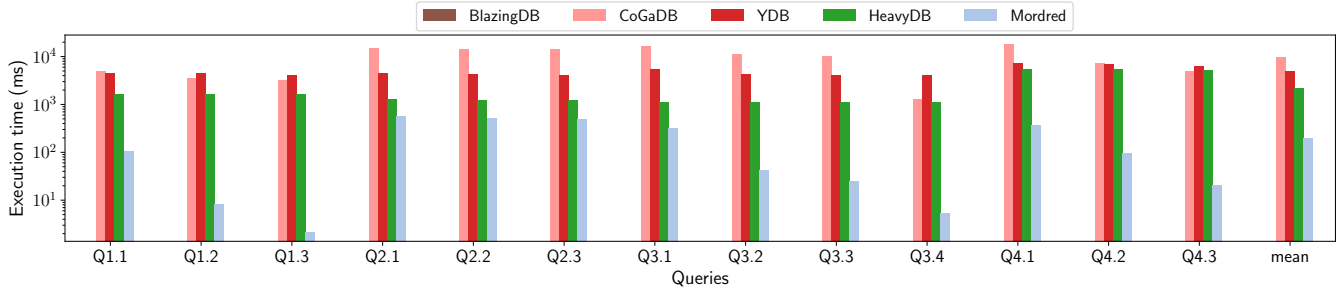

Figure 14: SSB Query Performance of Different CPU/GPU DBMS (Data does not fit in GPU)

To summarize, in this experiment we see how various optimizations in Mordred could: (1) lower the CPU traffic by offloading more work to GPU and (2) lower the total traffic by reducing the data being processed and pipelining operations.

6.5 Comparison with Other CPU/GPU DBMS

This subsection reports the end-to-end performance evaluation of four existing CPU/GPU DBMSs:

- **CoGaDB:** CoGaDB [17] is a prototype of column-store CPU-GPU DBMS. CoGaDB uses column-granularity LRU and LFU based replacement policy and utilizes a learning-based optimizer called Hype [23].
- **HeavyDB:** HeavyDB [8] is a commercial GPU DBMS. HeavyDB treats GPU as the primary execution engine. When the data does not fit in GPU, HeavyDB will divide the query plan into multiple stages and execute each stage on GPU one step at a time. To reduce the amount of data transfer, HeavyDB caches the data in GPU using the LRU policy.
- **BlazingDB:** BlazingDB [1] is a commercial GPU DBMS. BlazingDB uses the RAPIDS library [11] as its execution engine.
- **YDB:** YDB [54] is a prototype of column-store GPU DBMS. When the data does not fit, the input data will be transferred to GPU following the coprocessor model.
- **Mordred:** Mordred is our prototype of Hybrid CPU-GPU DBMS which utilizes fine-grain semantic-aware caching policy and segment-level query execution.

We run two sets of experiments: (1) when data fits in GPU and (2) when data does not fit in GPU. For Mordred and CoGaDB, we use 8 GB cache size. For HeavyDB, BlazingSQL, and YDB, we let the system control the GPU memory. When the data does not fit in GPU, we enable the coprocessor mode in HeavyDB and YDB. We use a scale factor of 40 for when the data fits in GPU and a scale factor 160 (4× the cache size) for when the data does not fit in GPU.

6.5.1 Data Fits in GPU.
Figure 13 shows the query performance when the data fits in GPU. For Q4.1-Q4.3, BlazingDB suffers an out-of-memory exception error and not shown in the figure. Compared to BlazingDB, YDB, CoGaDB, and HeavyDB, Mordred is around 400×, 175×, 48×, and 9× faster, respectively. These systems do not have the tile-based execution model in Crystal and thus do not utilize the GPU memory bandwidth as efficiently as Mordred. They also do not have all the performance optimizations we implement in Mordred, such as operator pipelining, segment-grouping, and segment skipping.

Across the 13 queries, Mordred's performance is significantly better compared to other systems especially in Q1.2, Q1.3, and Q3.4. These queries only access data in the range of a week or a month and benefit significantly from the segment skipping optimization.

6.5.2 Data Does Not Fit in GPU.
Figure 14 shows the query performance when the data does not fit in GPU. Since BlazingDB requires the dataset to fit in the GPU, we do not include it in this experiment.

Compared to CoGaDB, Mordred is around 48× faster. Apart from the lack of optimizations described in Section 6.5.1, our semantic-aware caching policy will be superior compared to the column-granularity LFU/LRU policy used by CoGaDB.

Compared to HeavyDB and YDB, Mordred is around 11× and 25× faster. This is because when the data does not fit in GPU, HeavyDB and YDB will switch to coprocessor mode and stream the data on demand from CPU memory during query execution. This results in suboptimal performance due to excessive data transfer.

Across the 13 queries, Mordred's performance gain in Q3.1-Q4.3 is more significant than Q2.1-Q2.3. This is because our semantic-aware policy chooses to prioritize caching more segments from Q3.1-Q4.3 which results in performance difference between the query sets. This behavior would not be apparent in other systems which do not adopt semantic-aware policy.

7 RELATED WORK

In this Section, we discuss prior works that are related to Mordred.

7.1 GPU as the Primary Execution Engine

There has been a number of research projects and commercial systems [1, 5, 8, 11, 12, 47] that treat GPU as the primary execution engine. These systems, however, will either force query execution on the CPU or trigger an out of memory execution error when the data does not fit in the GPU. To allow more data to fit in GPU memory, existing works have also attempted to use multiple GPUs [1, 5, 8, 11, 12] for query execution. These GPUs are connected via NVLink, a modern interconnect with higher bandwidth to mitigate the inter-GPU bottleneck. In this paper, we focus on systems with a single GPU device due to their wider deployment.

7.2 GPU as a Coprocessor

Some previous works in this category focused on accelerating individual database operations such as selection [49], join [34, 35, 38, 44–46, 48, 53], and sorting [31, 50] in one or more GPUs. To accelerate a single database operation, it is required to transfer the data from CPU to GPU and transfer the result back to the CPU main memory.

Several full-fledged GPU-as-co-processor engines have also been developed in the past [30, 40, 52, 54]. YDB [54] and HippogriffDB [40] stream the data from CPU memory and execute one operator at a time. To reduce the overhead of data transfer, they support compression over input data. Commercial systems such as HeavyDB [8] adopt GPU as a primary execution engine and switch to coprocessor model when the data no longer fits in GPU memory. We compared our performance against HeavyDB in Section 6.5.

7.3 Heterogeneous CPU-GPU Query Execution

There have been a few previous systems that attempted to leverage both CPU and GPU for query execution [17, 27, 33, 36, 37, 39, 42, 55].

GDB [33] was the earliest effort in this direction. GDB can execute an operator on both CPU and GPU by partitioning the input data prior to execution (e.g. partitioned hash join). OmniDB [29] improved GDB with kernel adapter design so that it could target different hardware architectures efficiently. Unlike Mordred, these systems do not handle data placement between CPU and GPU.

He et al. [36] discussed heterogeneous query execution that specifically targets an integrated CPU-GPU architecture in a single chip (e.g AMD APU). In this architecture, PCIe is no longer a bottleneck but the GPU is less powerful and has a much lower memory bandwidth than the ones in a discrete architecture.

DB2 BLU [42] showed how to use GPU and CPU cores for faster processing in IBM DB2 database. This work uses a heuristic to decide where to execute an operator based on its runtime features (e.g. input size, number of groups, etc). This work, however, only supports limited number of operators (group-by, aggregation, and sort) and does not address data placement between CPU and GPU.

Ocelot [21, 37] is a hardware-oblivious database engine which integrates GPU backend to the in-memory column-store MonetDB [14, 15]. Ocelot uses OpenCL [9] runtime to enable hardware-agnostic operator implementation. For its data placement policy, Ocelot caches the most recently used columns in GPU memory.

CoGaDB [17] is a main-memory DBMS with GPU accelerator. CoGaDB utilizes a framework called Hype [16, 18, 20, 22–26], which uses a learning-based model to assign operators on either CPU or GPU. The latest work of CoGaDB [19] introduced *data-driven operator placement* heuristic as an alternative to Hype (see Section 4.1.1). Experiments showed that this heuristic manages to outperform the learning-based optimizer [19]. Mordred also adopts the same heuristic as CoGaDB. Similar to Ocelot, CoGaDB uses column-granularity LRU or LFU policy to cache data in GPU. We compared the performance of Mordred against CoGaDB in Section 6.5.

HERO [39] investigated work placement in heterogeneous computing. Operator placement decision typically depends on the processed and transferred data in terms of data cardinalities. This work proposed a placement optimization strategy that can be completely independent on cardinality estimation of the intermediate result. However, unlike Mordred, this work focuses on placement optimization and does not address data placement and heterogeneous query execution. HERO is developed as an extensible virtual layer on top of YDB [54] which we compared against in Section 6.5.

Lutz et al. [41] discussed query execution across CPU and multiple GPUs through fast interconnect. This work, however, specifically targets CPU-GPU with NVLINK interconnect instead of PCIe, which is available only for IBM Processor in the current market. Most other processors (e.g., Intel and AMD) still use PCIe as the inter-device interconnect. Moreover, this work only focuses on hash join and does not support general queries like Mordred.

Finally, HetExchange [27, 28] introduced a query execution framework to encapsulate heterogeneous parallelism in hybrid CPU/GPU system through redesigning the classical *Exchange* operator. This framework is also integrated with just-in-time compilation engine to enable operator pipelining. HetExchange, however, does not address the data placement between CPU and GPU and lacks an optimizer component to generate the heterogeneous query plan based on the data location. Mordred addresses these issues through *segment-level query plan* which we described in Section 4.

8 CONCLUSION

This paper advances the state-of-the-art for heterogeneous CPU-GPU DBMS by contributing in two aspects: (1) data placement and (2) heterogeneous query execution. We introduce semantic-aware fine-grained caching policy which takes into account query semantics, data correlation, and query frequency when determining data placement between CPU and GPU. We also introduce a heterogeneous query executor which can fully exploit data in both devices and coordinate query execution at a fine granularity. We integrate both solutions in Mordred, our hybrid CPU-GPU analytical engine. Evaluation on the Star Schema Benchmark shows that our semantic-aware caching policy manages to outperform the best traditional caching policy by 3×. Mordred also manages to outperform existing GPU databases by an order of magnitude.

REFERENCES

[1] [n.d.]. BlazingSQL. https://blazingsql.com. Accessed 15-May-2022.
[2] [n.d.]. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 15-May-2022.
[3] [n.d.]. CXL. https://www.computeexpresslink.org/. Accessed 15-May-2022.
[4] [n.d.]. HIP Programming Guide. https://github.com/ROCm-Developer-Tools/HIP. Accessed 15-May-2022.
[5] [n.d.]. Kinetica. https://kinetica.com/. Accessed 15-May-2022.
[6] [n.d.]. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/. Accessed 15-May-2022.
[7] [n.d.]. NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed 15-May-2022.
[8] [n.d.]. OmniSci. https://omnisci.com. Accessed 15-May-2022.
[9] [n.d.]. Opencl. https://www.khronos.org/opencl/. Accessed 15-May-2022.
[10] [n.d.]. Parquet Encoding Format. https://github.com/apache/parquet-format/blob/master/Encodings.md. Accessed 15-May-2022.
[11] [n.d.]. RAPIDS. https://rapids.ai. Accessed 15-May-2022.
[12] [n.d.]. The RAPIDS Accelerator for Apache Spark. https://nvidia.github.io/spark-rapids/. Accessed 15-May-2022.
[13] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. 466–475. https://doi.org/10.1109/ICDE.2007.367892
[14] Peter Alexander Boncz et al. 2002. *Monet: A next-generation DBMS kernel for query-intensive applications*. Universiteit van Amsterdam [Host].
[15] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.
[16] Sebastian Breí, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient Co-Processor Utilization in Database Query Processing. *Inf. Syst.* 38, 8 (nov 2013), 1084–1096. https://doi.org/10.1016/j.is.2013.05.004
[17] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (2014), 199–209.
[18] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. 2013. Efficient co-processor utilization in database query processing. *Inf. Syst.* 38 (2013), 1084–1096.
[19] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-Accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936
[20] Sebastian Breß, Ingolf Geist, Eike Schallehn, Maik Mory, and Gunter Saake. 2012. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. *Control and Cybernetics* 41 (2012).
[21] Sebastian Breß, Max Heimel, Michael Saecker, Bastian Köcher, Volker Markl, and Gunter Saake. 2014. Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware. *Proc. VLDB Endow.* 7 (2014), 1609–1612.
[22] Sebastian Breß, Siba Mohammad, and Eike Schallehn. 2012. Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms.
[23] Sebastian Breß and Gunter Saake. 2013. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1398–1403. https://doi.org/10.14778/2536274.2536325
[24] Sebastian Breß, Norbert Siegmund, Max Heimel, Michael Saecker, Tobias Lauer, Ladjel Bellatreche, and Gunter Saake. 2014. Load-aware inter-co-processor parallelism in database query processing. *Data Knowl. Eng.* 93 (2014), 60–79.
[25] Sebastian Breß, Felix Beier, Hannes Rauhe, Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. 2012. Automatic Selection of Processing Units for Coprocessing in Databases, Vol. 7503. 57–70. https://doi.org/10.1007/978-3-642-33074-2_5
[26] Sebastian Breß, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2013. An Operator-Stream-Based Scheduling Engine for Effective GPU Coprocessing, Vol. 8133. 288–301. https://doi.org/10.1007/978-3-642-40683-6_22
[27] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760
[28] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf
[29] Shuhao Zhang et.al. 2013. OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1374–1377. https://doi.org/10.14778/2536274.2536319
[30] 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*. ACM, 1603–1618.
[31] Naga Govindaraju et al. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*.

[32] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886
[33] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (dec 2009), 39 pages. https://doi.org/10.1145/1620585.1620588
[34] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *SIGMOD*.
[35] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB* (2013).
[36] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-Cache Query Co-Processing on Coupled CPU-GPU Architectures. *Proc. VLDB Endow.* 8, 4 (dec 2014), 329–340. https://doi.org/10.14778/2735496.2735497
[37] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* (2013).
[38] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *DaMoN*.
[39] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (mar 2017), 733–744. https://doi.org/10.14778/3067421.3067423
[40] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.
[41] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705
[42] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. 2016. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1951–1960. https://doi.org/10.1145/2882903.2903735
[43] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.
[44] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. *MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254
[45] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927
[46] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. ACM, 17.
[47] Anil Shanbhag, Xiangyao Yu, and Samuel Madden. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data*. ACM.
[48] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. *Hardware-conscious Hash-Joins on GPUs*. Technical Report.
[49] Evangelia A Sitaridi and Kenneth A Ross. 2013. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. ACM, 4.
[50] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. ACM.
[51] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.
[52] 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.
[53] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on GPUs: A closer look. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2663–2673.
[54] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB* (2013).
[55] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. 2015. Hetero-DB: Next Generation High-Performance Database Systems by Best Utilizing Heterogeneous Computing and Storage Resources. *Journal of Computer Science and Technology* 30 (2015).