GPU Database Systems Characterization and Optimization

Jiashen Cao*
Georgia Tech
jiashenc@gatech.edu

Rathijit Sen
Microsoft GSL
rathijit.sen@microsoft.com

Matteo Interlandi
Microsoft GSL
matteo.interlandi@microsoft.com

Joy Arulraj
Georgia Tech
arulraj@gatech.edu

Hyesoon Kim
Georgia Tech
hyesoon@cc.gatech.edu

ABSTRACT

GPUs offer massive parallelism and high-bandwidth memory access, making them an attractive option for accelerating data analytics in database systems. However, while modern GPUs possess more resources than ever before (e.g., higher DRAM bandwidth), efficient system implementations and judicious resource allocations for query processing are still necessary for optimal performance. Database systems can save GPU runtime costs through just-enough resource allocation or improve query throughput with concurrent query processing by leveraging new GPU resource-allocation capabilities, such as Multi-Instance GPU (MIG).

In this paper, we do a cross-stack performance and resource-utilization analysis of four GPU database systems, including Crystal (the state-of-the-art GPU database, performance-wise) and TQP (the latest entry in the GPU database space). We evaluate the bottlenecks of each system through an in-depth microarchitectural study and identify resource underutilization by leveraging the classic roofline model. Based on the insights gained from our investigation, we propose optimizations for both system implementation and resource allocation, using which we are able to achieve 1.9× lower latency for single-query execution and up to 6.5× throughput improvement for concurrent query execution.

PVLDB Reference Format:
Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. GPU Database Systems Characterization and Optimization. PVLDB, 17(3): 441-454, 2023.
doi:10.14778/3632093.3632107

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jiashenC/gpudb-char-and-opt.

1 INTRODUCTION

Graphics Processing Units (GPUs) have attracted significant interest in the realm of accelerating data analytics due to their potential for massively parallel computing, high-bandwidth memory access capability, and ease of programming as an accelerator. In recent years, a number of GPU database management systems (DBMSs) have been developed in both academic and industrial settings [10, 11, 16, 17, 21–23, 28, 29, 50, 57, 59, 67]. Recent advances in interconnect [1–4] and GPU architecture have made GPUs more attractive as accelerators for data analytics. As a result, we anticipate a rise in the popularity of GPU DBMSs and a proliferation of research into improving their efficiency in the future.

Several factors contribute to query performance in GPU DBMSs, including (1) the resource capacity (including both compute and memory resources) of the GPU; (2) the implementation of the DBMS; (3) the characteristics of the query; and (4) the size of the database. A deeper understanding of GPU resource utilization and encountered bottlenecks is crucial in designing better GPU DBMSs.

Prior Work. While previous studies (*e.g.,* [19, 21, 62]) have compared query performance across GPU DBMSs, a cross-stack analysis that connects query performance with the GPU resource utilization along with microarchitectural metrics is lacking. In contrast, similar studies [5, 54, 55, 61] exist for CPU DBMSs. We aim to address this gap by conducting both resource-utilization and performance-bottleneck analysis for GPU DBMSs. Based on the findings from our studies, we propose optimizations to improve both the system implementations and GPU resource allocations.

With several GPU instance types now available, each with different performance and cost trade-offs, choosing the most suitable GPU for a particular workload can be challenging. For example, a T4 GPU can run the TPC-H benchmark 20% slower compared to a P100, but at $\frac{1}{5}$th of the cost [21]. To complicate things further, starting from the Ampere generation, NVIDIA now allows different resource-allocation mechanisms through the Multi-Instance GPU (MIG) capability [41]. This new capability allows partitioning GPU hardware resources to increase concurrency by sharing the GPU. Developers are now faced with several choices on how to schedule their workloads over GPUs, which requires a deep understanding of both workload characteristics and hardware properties.

Characterization. In this paper, we conduct an in-depth study on both GPU resource utilization and performance bottlenecks. We build on the roofline model [65] to identify the underutilized resources in GPUs. Our findings indicate that many queries underutilize either DRAM or L2 cache bandwidth, even in highly-optimized systems. Since GPU resource utilization is dependent on the implementation of each system, we conduct a GPU execution performance bottleneck study by profiling microarchitectural performance counters to fully comprehend the reasons for the underutilization of resources in each system. The study reveals that several systems still have opportunities for further optimization due to a lack of kernel fusion, inefficient threads termination, and

[*] Work done during an internship at Microsoft Gray Systems Lab.

Proceedings of the VLDB Endowment, Vol. 17, No. 3 ISSN 2150-8097.
doi:10.14778/3632093.3632107

cache underutilization. Our approach can be readily adopted by other systems to identify and address potential performance issues.

Optimization. Based on the insights we gained, we propose remedies to improve the queries' performance. Our strategy focuses on two aspects: efficient system implementation and optimal resource allocation. To optimize system implementations, we improve terminations of idle threads and cache hit rates for the best-performing (academic) GPU database (Crystal). To improve resource allocation, we employ an analytical model to determine the ideal resources to allocate so that more query executions can be concurrently accommodated by the same GPU. This approach improves the overall throughput without modifying any system implementation. As far as we know, we are the first to show how MIG and MPS technology can be used to improve GPU databases throughput.

Contributions. The contributions of this work are as follows:

- **Characterization of GPU DBMSs** (§4 and §5): To close the gap in GPU DBMS characterizations, we conduct two original studies: (1) GPU resource-utilization analysis, and (2) GPU DBMS bottleneck analysis based on GPU microarchitectural metrics. Our studies show that many GPU DBMSs underutilize GPU resources. Additionally, we discover previously unknown performance bottlenecks (*e.g.,* inefficient memory bandwidth utilization) existing in the State-of-the-Art (SotA) GPU DBMSs.
- **Performance optimization of GPU DBMSs** (§6 and §7): To demonstrate that the analyses are useful for guiding optimizations, we showcase implementation optimizations on Crystal, resulting in an improved single-query execution performance with an average speedup of 1.9×. This, we believe, makes it the fastest open-source[1] implementation of SSB queries on GPUs. Furthermore, we demonstrate GPU resource allocation optimizations guided by our proposed analytical model. The analytical model can reason about GPU resource utilization, while concurrently running multiple queries with up to 6.5× throughput improvement. This tool provides system administrators with effective guidelines on how to allocate GPU resources to queries.

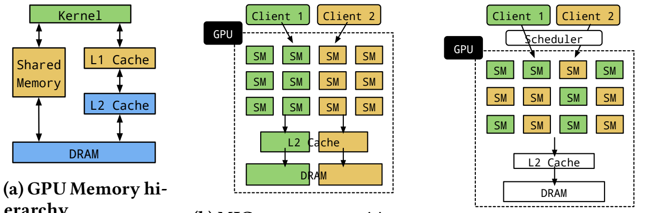
Outline. We introduce the background information and experimental setup in §2 and §3. In §4, we compare the performance among various GPU DBMSs. Motivated by the performance gap between different GPU DBMSs, we use the roofline model to understand the GPU resource utilization of those systems. Then, to understand the causes of GPU resource underutilization, we profile GPU microarchitectural metrics to understand existing bottlenecks in §5. Finally, we showcase our proposed optimizations in §6 and §7 to address our identified performance bottlenecks.

2 BACKGROUND

We first discuss GPU architectures and existing GPU resource-allocation mechanisms in §2.1. We provide information about the roofline performance analysis model in §2.2. We then present an overview of existing GPU database systems in §2.3.

2.1 GPU Architecture

In this section, we provide an overview of the memory hierarchy and available resource-allocation mechanisms in NVIDIA GPUs.



(a) GPU Memory hierarchy.

(b) MIG – resource partitioning for multi-instance GPU.

(c) MPS – resource partitioning in multi-process service.

Figure 1: GPU background and concurrency mechanisms – NVIDIA GPUs memory hierarchy, and supported concurrency mechanisms.

As Figure 1a shows, kernel execution can access data stored either in the shared memory or in the L1 cache. Shared memory is fully managed by the user, whereas the L1 cache is managed by the hardware. Each GPU Streaming Multiprocessor (SM) has a private L1 cache and shared memory region, but the L2 cache and DRAM are shared across all GPU SMs. The memory hierarchy is consistent for all NVIDIA GPUs, but the specific values for capacity and bandwidth vary across GPUs. NVIDIA GPUs support two ways for allocating resources on the GPU, and providing concurrent GPU execution capability to processes: Multi-Instance GPU (MIG [42]) and Multi-Process Service (MPS [40]). MIG is a new feature supported only on the A100, A30, and H100 GPUs [41].

MIG. MIG enables physical partitioning of GPU resources—SMs, L2 cache, DRAM capacity, and bandwidth—which creates full isolation between concurrent processes. Figure 1b shows an example of resource allocation through MIG to support two concurrent clients with equal allocation ($\frac{1}{2}$ GPU resources) in this example. MIG also supports heterogeneous resource partitions to meet the varying needs of different clients. At its finest granularity, MIG currently supports up to seven concurrent clients: each partition gets about $\frac{1}{8}$ of compute and memory resources, while nearly $\frac{1}{8}$ of the resources is reserved for the MIG controller. MIG currently offers a total of 18 choices for resource partitions on the A100.

MPS. MPS employs a logical resource partitioning to support concurrent execution. In previous GPU generations, MPS allows only time-sharing of the GPU. Since Volta, MPS allows concurrent execution through lightweight partitioning of SMs. In MPS, the L2 cache and DRAM are still unified resources without any isolation. When MPS starts, the GPU creates a scheduler to manage resource allocation among different CUDA applications, ensuring that each application receives the requested resources. MPS also allows specifying memory capability limits since CUDA 11.5 [7].

2.2 Roofline Performance Modeling

The roofline model [65] is a well-known performance-modeling method to study resource bottlenecks for existing algorithms. The roofline model assumes that any execution on specific hardware is bounded either by its memory resources or by its compute resources. Visually, as shown in Figure 2, the model contains two lines to indicate the peak memory bandwidth (α) and the peak compute bandwidth (β). Any execution on this hardware will correspond to a point within the space bounded by these lines—those two lines are considered the performance ceilings for that hardware. The X-axis represents the arithmetic intensity (AI), calculated by dividing the

[1] Profiling commands to obtain characterization results and codes for proposed optimizations are released at https://github.com/jiashenC/gpudb-char-and-opt.

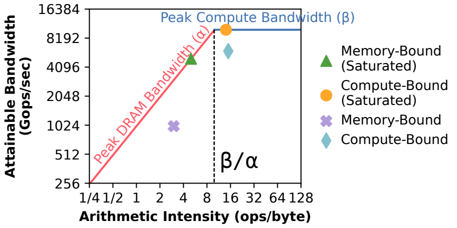


Figure 2: Roofline model with different resource bounds.

Table 1: Qualitative comparison of GPU database systems – We consider the following characteristics. **Coverage**: whether the system is general purpose or only specific queries are supported. **Data Format**: formats for data input to the system. **Backend**: how data operators are compiled for execution. **Open Source**: if the system is publicly available.

System	Crystal	HeavyDB	BlazingSQL	TQP
Coverage	SSB [47] only	General purpose	General purpose	General purpose
Data Format	Binary array	CSV Parquet	CSV, DF Parquet	CSV, DF Parquet
Backend	Hardcoded CUDA	LLVM to PTX	Thrust cuDF	PyTorch
Open Source	Yes	Yes	Yes	No

number of operations (*e.g.,* integer or floating-point operations) by the number of bytes read during execution. The Y-axis indicates the achieved throughput, calculated as the operations per second.

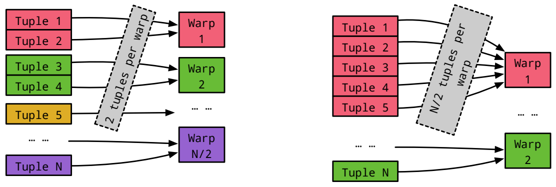
Conventionally, a query could be either memory-bound (AI < $\frac{\beta}{\alpha}$), or compute-bound (AI > $\frac{\beta}{\alpha}$) [46]. The performance of algorithms that already saturate the bandwidth of either of the resources will be impacted by changes in the allocation of the corresponding resource (*e.g.,* memory-bound saturated or compute-bound saturated in Figure 2). Algorithmic or compiler inefficiencies will increase the AI and make an otherwise memory-bound execution compute-bound, causing the query to take more time to complete.

2.3 GPU Database Systems

Table 1 summarizes the key characteristics of four database systems.

Crystal. Crystal [57] is a recently proposed SotA GPU database system that delivers superior query-execution performance compared to other systems. It currently supports only queries from the Star Schema Benchmark (SSB) [47]. All the queries are written in CUDA and have hard-coded parameters for the size of the hash table, the number of groups, and the size of the output table [56]. While this is feasible for these queries in SSB, it is not feasible to predetermine these parameters for arbitrary SQL queries. Crystal also assumes that each column is a binary array generated after preprocessing. String columns in Crystal are converted to binary arrays using dictionary encoding on the CPU.

HeavyDB. HeavyDB [22] is a widely used GPU database system that supports many types of SQL queries. Besides the query executor, it contains other components like query parser and query optimizer. HeavyDB takes various data formats as input, such as CSV and Parquet. Data in HeavyDB is grouped into fragments and transferred to the GPU using different CUDA streams. Internally, it uses LLVM [30] to generate PTX [44] code for query execution. Additionally, HeavyDB leverages a custom LLVM pass for more flexible and optimized execution strategies.



(a) Bulk execution model.

(b) Warp execution model.

Figure 3: Query execution models – Data mapping for query execution.

BlazingSQL. BlazingSQL [11] is another open-source GPU database system. Similar to HeavyDB, BlazingSQL is also very flexible and handles different types of queries. Unlike HeavyDB, it uses Thrust [45] and cuDF [64] as its backend for query execution. Reusing the APIs of Thrust and cuDF has both pros and cons. While less engineering is needed to support various operations, certain operators cannot be implemented in an efficient way (*e.g.,* cannot avoid the unnecessary generation of intermediate results) due to the limited functionalities of existing APIs. BlazingSQL also supports different input data formats, while data moves between CPU and GPU through different CUDA streams.

TQP. TQP [21] is a recently-presented GPU database system from Microsoft. It is designed to be general-purpose (*e.g.,* it supports the full TPC-H benchmark). The interesting aspect of TQP is that internally it uses the PyTorch [48] framework as its backend for executing relational operations. This design choice allows it to quickly support many different operations with existing GPU kernels that are already optimized. As the PyTorch framework already supports various hardware platforms (*e.g.,* NVIDIA GPUs, AMD GPUs), TQP inherits the portability and extensibility of the PyTorch framework.

2.3.1 *Query Optimizations.*

All of these systems have query optimization and query compilation phases except for Crystal, where the query plans are hard-coded and predetermined based on the selectivity of each operator (*i.e.,* selective operators are executed earlier). HeavyDB and BlazingSQL rely on the Apache Calcite framework for query optimization. HeavyDB also uses LLVM to further optimize the physical execution of GPU code. TQP utilizes SparkSQL [6] to optimize the SQL queries and then translates the Spark SQL physical plans to an intermediate representation (IR). Based on the IR, TQP assembles a PyTorch program as a composition of predefined tensor programs, one for each operator in the IR. The implementation is later optimized by the PyTorch compiler.

2.3.2 *Query Executions.*

There are two common execution models, as shown in Figure 3. The bulk execution model (Figure 3a) has a fixed number of tuples per warp. The GPU will launch a variable number of warps to accommodate different data sizes. Crystal, BlazingSQL, and TQP all adopt this execution model. On the other hand, HeavyDB uses a warp execution model, in which the GPU always launches a fixed number of warps (2 as shown in Figure 3b). Consequently, each warp gets a variable number of tuples to process based on the total amount of work. Those two different execution models can also have an impact on GPU memory efficiency.

Table 2: Qualitative summary of the queries from SSB – Key characteristics of queries in SSB: **Number of Joins**, the **Aggregation** type, whether they require **Sorting**, and query **Selectivity**.

Query Group	Group 1			Group 2			Group 3				Group 4		
	Q11	Q12	Q13	Q21	Q22	Q23	Q31	Q32	Q33	Q34	Q41	Q42	Q43
# Joins	1			3			3				4		
Aggregation	Sum			Group By			Group By				Group By		
Sorting	No			Yes			Yes				Yes		
Selectivity	1.9e-2	6.5e-4	7.5e-5	8e-3	1.6e-3	2e-4	3.4e-2	1.4e-3	5.5e-5	7.6e-7	1.6e-2	4.6e-3	9.1e-5

3 EXPERIMENTAL SETUP

We discuss the hardware, query workloads, and profiling toolchains used in this paper in §3.1, §3.2, and §3.3, respectively. We then describe the two execution scenarios that we consider in §3.4.

3.1 Hardware

We use an NVIDIA A100 GPU [39] with the Ampere architecture, 40GB of GPU memory, and 108 SMs on a dual-socket AMD EPYC 7313 machine with 16 physical (32 logical) cores per socket and 3 GHz base clock frequency. Each SM can schedule up to 64 *warps*. A *warp* is the basic execution unit of the NVIDIA GPU. A warp consists of 32 threads scheduled and executing in a *single instruction, multiple threads* (SIMT) fashion (*i.e.,* each thread executes the same instruction on different data but allows divergence between threads). The A100 GPU supports newer features like MIG and MPS, as introduced in §3.1. The GPU is connected to the CPU via PCI-e 4 protocol, which provides up to 32 GB/s of bandwidth. For consistent performance, we set the GPU clock frequency to be 1410 MHz.

3.2 Workloads

Since Crystal currently only supports SSB [47], in our experiments we decided to evaluate SSB queries over the four different GPU database systems. Even if SSB is simpler than other benchmarks like TPC-H and TPC-DS, its queries are complex enough that system characterization on this benchmark enables us to find several interesting insights. We are confident that these insights generalize to other workloads as well.

Table 2 presents a summary of the queries in SSB. Queries in group 1 have only one join, and the results are aggregated into a single scalar value. All the other queries have multiple joins (up to 4), and the final results are aggregated into a table with multiple groups. Based on our profiling, we found that 16 is the largest scale factor on which all DBMSs can run queries without moving data back and forth between GPUs and CPUs or encountering out-of-memory issues on the GPU. So, we present results for most of the experiments with a scale factor of 16 unless specifically mentioned.

3.3 Profiling Toolchains

We extensively use NVIDIA Nsight System [43], Nsight Compute [43], and nvidia-smi [38] tools built with CUDA11.6. The NSight System provides system-wide time breakdown, including time spent on data transfer, memory allocation, kernel execution, etc. We also use Nsight Compute to obtain detailed kernel execution metrics, which include achieved instruction per cycle, cache utilization, etc.

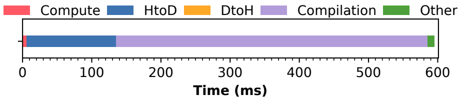


Figure 4: Cold execution characterization over Q21 at SF=16 in HeavyDB – We split the end-to-end performance of cold query execution into six components: *Compute* time, *Host-to-Device* data transfer (HtoD), *Device-to-Host* data transfer (DtoH), *Compilation* time, and *Other* CUDA context setup and memory management time. Similar results apply to other queries and database systems.

3.4 Warm *vs.* Cold Execution Scenarios

In the *warm scenario*, we assume that the data has already been loaded (or cached) in GPU memory, the device has warmed up, and the query has been parsed, optimized, and compiled (*e.g.,* the physical plan is available in the plan cache). In the *cold scenario*, we assume that the data resides in CPU memory and needs to be transferred to the device, and all the other overheads associated with query parsing, optimization, and compilation are considered.

Cold Execution Overhead. In this paper, our focus is on warm query execution, where the majority of the time is spent on GPU computations. However, here we briefly report the timing breakdown for cold query execution to provide an overview of where time is typically spent. With cold query execution, there are several non-negligible overheads from both query optimization and compilation, as well as from data transfer. Figure 4 shows that these overheads are always greater than the actual query execution time (Compute). Data needs to be moved to the GPU before query execution, and this incurs significant overhead. Data transfer and kernel computation can be overlapped, but the time saving is small due to data transfer overhead being the dominant bottleneck. During data transfer, most systems copy only the columns needed for the query to the GPU to reduce the host-to-device (HtoD) data transfer overhead. HtoD overhead in Figure 4 reflects the time spent on copying the required columns.

Systems also need to move results from the device back to the host. However, as all the queries compute aggregated results that are very small, the device-to-host (DtoH) overhead is generally minimal. Query plan optimization and compilation also introduce non-trivial overhead. This is unavoidable when the system receives a query for the first time. Most systems implement plan caching, so recurrent queries do not have the same overhead.

4 PERFORMANCE AND RESOURCE ANALYSIS

In this section, we first provide a high-level overview of execution time for both end-to-end and GPU computation, in all systems, in §4.1. We then conduct a resource-utilization analysis to reason about performance differences between systems in §4.2.

4.1 Query Execution Performance

We present the end-to-end warm query execution performance in Figure 5 [2]. The most notable observation is that Crystal represents the upper bound in execution performance among all considered systems. Both HeavyDB and TQP are on average 8× slower than Crystal. BlazingSQL is even slower, being 30× slower than

[2] Most systems can successfully execute SSB queries except BlazingSQL, which cannot run Q41, Q42, and Q43 due to errors during dataframe encodings.

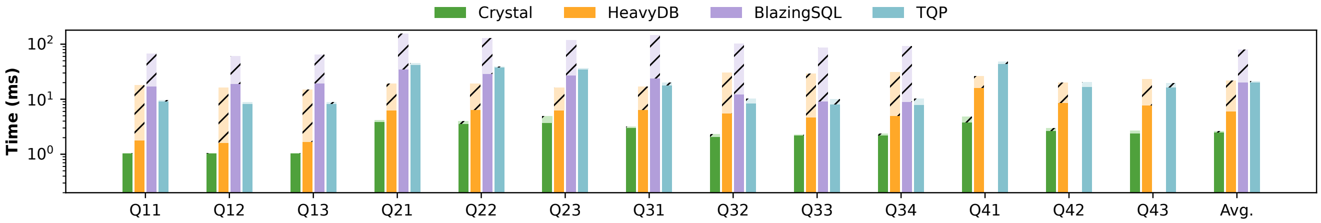


Figure 5: Warm execution characterization – End-to-end query execution time and GPU execution time for all queries (BlazingSQL does not support queries in group 4; those queries are skipped when calculating the average.). The hatched area represents time spent besides the actual GPU execution.

Crystal. Next, we provide our analysis of how time is spent on each system by using the time breakdown information.

Crystal and TQP. For both of these systems, the end-to-end time mostly represents the actual GPU compute time, with some small overheads besides GPU execution. In Crystal, each CUDA source file implements one query from SSB. Once the source file is compiled, there is no additional compilation overhead associated with the query. Unlike Crystal, TQP is a general-purpose system that supports a wider range of SQL queries. In TQP, the workflow for executing queries is done in two phases: the input query is first parsed, optimized, and compiled into a PyTorch model object; then, the already-compiled model is executed over the input data. Because of this workflow, the query execution time does not contain optimization and compilation time. In addition, TQP lazily caches the query results in the GPU until they are requested. This approach is beneficial for reducing end-to-end query execution times if the results are not immediately required, or if there is a subsequent query executing on the previously generated output results.

HeavyDB and BlazingSQL. For HeavyDB and BlazingSQL, their end-to-end time consists not only of the GPU compute time but also of other overheads. For example, HeavyDB uses Calcite for query plan optimization, and LLVM to compile the query into executable code. Although HeavyDB implements a plan cache, it still has non-negligible overhead for query parsing and GPU setup. Similar to HeavyDB, BlazingSQL also has non-trivial overheads beyond the GPU compute time. In practice, if the system is able to look up the cached physical query plan without even parsing the query, like TQP, then it can avoid many unnecessary overheads.

Next, we study each system's GPU execution time and its efficiency. While some works benchmark different GPU database systems [62], studies on only GPU execution efficiency have not been done before. The GPU execution time of the studied systems is shown in Figure 5 with non-hatched bars.

For GPU execution time, Crystal is still the fastest system. However, unlike the end-to-end query time, HeavyDB performance is closer to Crystal (2× slowdown, as opposed to 8× slowdown, reported in the end-to-end comparison). TQP is still 8× slower than Crystal. BlazingSQL has the slowest performance among these systems and is unable to run queries in group 4.

4.2 GPU Resource Utilization

The significant performance gap among different systems motivates us to investigate the reasons behind it. In this section, we attempt to visualize the resource utilization of each system and rely on it to reason about the root causes of the performance gap.

Table 3: Profiled metrics to construct the roofline model.

	Metric Name	Description
DRAM	gpu__time_duration.sum	Execution duration
	dram__bytes.sum	Total bytes from DRAM
	smsp__sass_thread_inst_executed_op_integer_pred_on.sum.per_cycle_elapsed	Achieved compute bandwidth
	sm__sass_thread_inst_executed_op_integer_pred_on.sum.peak_sustained	Peak compute bandwidth
L2 Cache	lts__t_requests_srcunit_tex_op_read.sum	Total requests to L2 cache

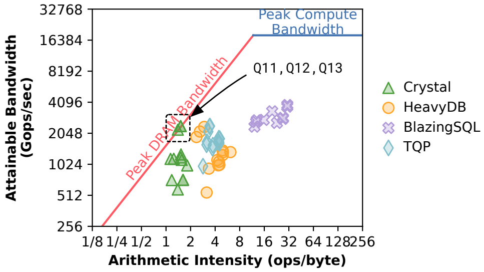


Figure 6: DRAM roofline model – for SSB queries (SF=16).

During our investigation, we explain the reasons for GPU resource underutilization by connecting them to query characteristics and system implementations.

4.2.1 *DRAM Utilization.*

We first present the DRAM-level resource utilization for all systems. We obtain the metrics described in Table 3, from which we derive the AI and the attainable bandwidth. We use the theoretical GPU DRAM bandwidth [39] for the DRAM bandwidth ceiling. We note that GPUs also have other functional units like floating-point operation units. However, because most OLAP queries require only integer operations, those functional units are not needed for constructing the roofline model.

The challenge for this type of modeling is that each query consists of multiple kernels. Our approach is to aggregate scalar metrics such as the execution duration, total bytes, and total integer operation instructions. We then use the aggregated metrics to obtain the required metrics for constructing the roofline model. Figure 6 shows where all 13 SSB queries are located with respect to the roofline model, for each of the four GPU DBMSs.

Query Specific. We observe that most queries do not fully use the provided DRAM bandwidth because they all have hash-join operator causing many random accesses. Queries like Q11, which have only one hash-join operator, are likely to have higher achieved DRAM bandwidth.

Implementations Specific. Although query characteristics influence both AI and achieved bandwidth, they are also largely dependent on the system implementations. As shown, the AIs of Crystal, HeavyDB, and TQP are relatively low. Compared to the other three systems, BlazingSQL is instead more compute intensive. This shows that even simple OLAP queries can be compute intensive depending on the query implementation in the DBMS.

Even the same queries have different achieved bandwidths in different implementations. For example, three queries have already saturated the peak GPU DRAM bandwidth in Crystal. This is because Crystal implements the hash join as a filter for Q11, Q12, and Q13 (§3.2). For those queries, the hash join is just a one-to-one mapping between rows from the fact table and the dimension table. Crystal pushes down the predicate selection from the dimension table to the fact table, which skips the hash join and simply runs a predicate selection (*i.e.,* filter) on the fact table. As the filter operator involves only running a table scan, it is feasible to saturate the GPU DRAM bandwidth (unlike hash join). Even though this may not be applicable to all queries, it is still an interesting optimization to do from the resource-utilization perspective.

4.2.2 *L2 Cache Utilization.*

We also discovered that the GPU DRAM bandwidth is *not* the only resource constraint during query execution. Especially for optimized systems like Crystal and HeavyDB, they are more likely to saturate the peak L2 cache bandwidth. This motivates us to extend the roofline modeling methodology to the L2 cache as well. In prior work, Ilic [26] also proposed to make the roofline model cache-aware for CPUs.

Model Innovation. On top of the previous study, we find that *the AIs of different resources are very different.* For example, when a query has a good L2 cache hit rate, most of the memory requests are handled by the L2 cache. So, the number of bytes loaded from the L2 cache are high. As a result, the AI relative to the L2 cache is low. On the other hand, because there are fewer bytes loaded from the GPU DRAM, the AI is high with a fixed number of integer operation instructions. This requires us to have a separate roofline model to characterize the same query for a different memory resource.

To construct the roofline model for the L2 cache, we reuse most metrics profiled in Table 3 except for the total bytes read from DRAM. To estimate the bytes read from the L2 cache, we profile the number of L2 requests (shown in Table 3) that the kernel loads, which can be used to calculate the total bytes loaded from the L2 cache by knowing that the cache line size is always 128 bytes. We present our profiling results in Figure 7.

Query Specific. We first observe that the AI of queries on the DRAM roofline model is higher than their corresponding AI on the L2 cache roofline model. This is reasonable, especially in the case in which queries have good utilization of the L2 cache bandwidth. Because most memory requests are completed at the L2 cache level, the GPU has less data to handle at the DRAM level. We also observe that many of those queries have reached the peak L2 cache bandwidth. We discover that queries with hash joins are more likely to saturate the L2 cache bandwidth. Because SSB has relatively small dimension tables that can fit into the L2 cache of a high-end GPU (*e.g.,* 40 MB), queries may still exhibit good L2 cache utilization due to the spatial locality of data accesses from the small working set size even though hash joins involve random accesses. In Figure 6, some queries saturate the peak DRAM bandwidth—queries with simple filters are more likely to be bound by DRAM bandwidth. Since there is minimal data reuse, most data is streamed to the kernel, and cache utilization is lower. As a result, visually, those queries move away from the peak L2 bandwidth ceiling.

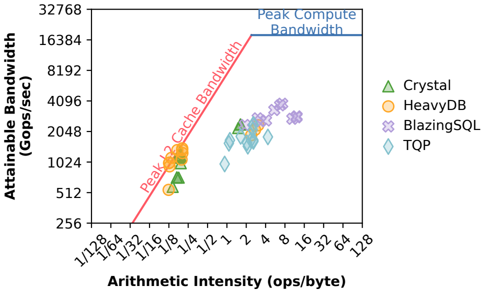


Figure 7: L2 cache roofline model – for SSB queries (SF=16).

Implementations Specific. Because Crystal and HeavyDB are very optimized, their queries saturated the L2 cache bandwidth. In BlazingSQL and TQP, the query implementations are far from the peak L2 cache bandwidth ceiling. Those two systems now have clearly separated clusters from the one from Crystal and HeavyDB, indicating lower cache efficiency in BlazingSQL and TQP.

5 BOTTLENECK INVESTIGATION

In §4, we provide an overview of the performance and resource utilization of various GPU database systems. GPU resource underutilization can be caused by many different factors. Therefore, in this section, we aim to examine the existing implementation bottlenecks, by using hardware performance counters, and offer insights into how to reduce those bottlenecks.

5.1 Operation and Data Efficiency

As mentioned in §2.2, the roofline model is derived from compute operations and data load. Thus, we first study the operation and data efficiency by profiling the following hardware counters: (1) the number of integer operations, and (2) bytes loaded from DRAM.

5.1.1 *Integer Operations and Bytes.*

First, we profile the total integer operations for the four systems, as shown in Figure 8 (top chart). We find that HeavyDB performs slightly more integer operations than Crystal. Conversely, BlazingSQL and TQP execute significantly more integer operations compared to Crystal and HeavyDB. We also examine the number of bytes each system reads from the GPU DRAM (Figure 8, bottom chart). Similar to the number of integer operations each system executes, Crystal and HeavyDB read less data from the GPU DRAM than the other two systems. For most queries besides Q43, Crystal reads slightly more data from the GPU DRAM than HeavyDB due to its bulk execution model. However, BlazingSQL and TQP read significantly more data from the GPU DRAM in most cases. Queries Q32, Q33, and Q34 are exceptions where the amount of data loaded from the GPU DRAM by

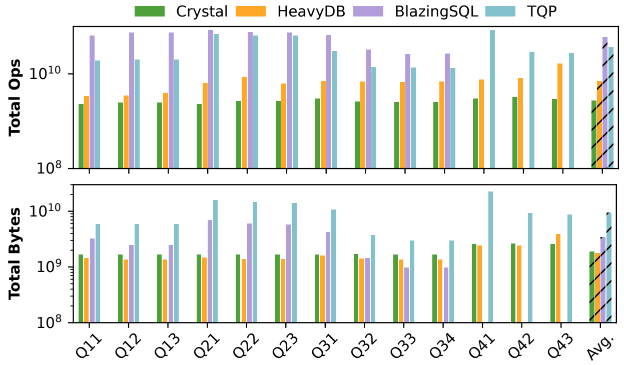


Figure 8: GPU execution efficiency – total integer operations (top) and number of bytes from DRAM (bottom).

BlazingSQL is equal to or even lower than that by Crystal and HeavyDB. Our investigation reveals that Crystal does not fully avoid unnecessary data loading after predicates and joins.

General Purpose. Most of the systems are designed to be general-purpose, so they have additional operations other than the actual algorithm. For example, even though HeavyDB is already very close to the optimal, it still has GPU algorithms to calculate the hash-table size for hash join on-the-fly, while Crystal has a hard-coded hash-table size. Other systems have similar operations to set up needed data structures and meta-data.

Algorithmic Complexities. Optimal implementations like Crystal mostly operate with a bitmap vector. It marks the bit to indicate whether the tuple satisfies predicates or joins without materializing the data. In contrast, systems like BlazingSQL largely depend on tuple indices. For example, it outputs the matched indices of the two tables when it does a join. As a result, it must materialize the tuple data again based on the indices. Additionally, general-purpose systems implement more complicated algorithms, such as MurmurHash for hashing operations. This is crucial to distribute data evenly, especially for an open-addressing linear hash table. Instead, Crystal implements a perfect hash table, which does not make it practical to handle general-purpose use cases.

Kernel Fusion. There are two general approaches for kernel fusion. Systems like Crystal and HeavyDB write their own customized execution engine, allowing them to do rigorous kernel fusion. As a result, they can avoid many intermediate results, which in turn avoid a large number of bytes read from DRAM. On the other hand, BlazingSQL and TQP rely on existing third-party APIs for query execution. Those APIs are designed to be modular but are not customized for query execution. So, generating many intermediate results is often not avoidable. It is also very challenging to do rigorous kernel fusions if a system chooses to use this approach.

Extensibility. As previously mentioned, BlazingSQL and TQP have performance overheads since they reuse functions from other libraries to construct queries. Nevertheless, this approach allows the system to be easily extensible and portable. For instance, TQP uses PyTorch as its backend and can run on any hardware platforms that PyTorch supports (e.g., NVIDIA and AMD GPUs), integrates with ML tools [8], and supports multimodal and trainable queries [20].

Table 4: Summary of most time-consuming kernels – three most time-consuming GPU kernels for executing query Q21 on all systems.

		Top 1	Top 2	Top 3
Crystal	Kernel	*probe_ht*	*build_htp*	*build_hts*
	Time (ms)	3.96 (99.35%)	0.01 (0.37%)	0.01 (0.37%)
HeavyDB	Kernel	*multifrag*	*fill_hj*	*init_hj*
	Time (ms)	5.86 (91.78%)	0.51 (7.96%)	0.01 (0.19%)
BlazingSQL	Kernel	*probe_ht*	*comp_hj_output*	*parallel_fn*
	Time (ms)	13.42 (38.26%)	12.27 (34.97%)	5.55 (15.83%)
TQP	Kernel	*collect_fn*	*idx_select*	*gather_fn*
	Time (ms)	16.88 (40.88%)	9.55 (23.13%)	5.72 (13.86%)

5.1.2 Kernel Execution Time Breakdown (Q21). Following our observation, we study GPU compute-time breakdown for query Q21 over the four systems as a micro-benchmark. Although we only report one query result, we find that the time breakdown between different queries is very consistent. Due to space limitations, we report only the three most time-consuming kernels. We note that BlazingSQL and TQP run 172 and 190 kernels for Q21, respectively. In contrast, both Crystal and HeavyDB run only four kernels.

Crystal and HeavyDB. These systems spend most of the query execution time on only the top kernel. As shown in Table 4, more than 90% of the total compute time is spent on the top kernel. As discussed, this is because both systems implement kernel fusion, through which they execute as many operators as possible in each kernel to avoid the generation of intermediate results. Since only the hash-join operation is a *pipeline breaker* [37] for this particular query, they implement the hash-join build phase in a separate kernel (*e.g., build_htp* and *fill_hj*).

BlazingSQL. In this case, one query is split into many kernels. Among the different kernels, generated data (*e.g.,* bitmap to identify the selected rows) needs to be materialized in order to be visible to other kernels. This causes higher overhead compared to the other systems that do not need to materialize intermediate results. Results show that even the hash-table probing phase already exceeds the execution time of both Crystal and HeavyDB due to the algorithm complexity issue mentioned above.

TQP. Similar to BlazingSQL, TQP executes many kernels to complete one query, and in fact, it spends more time on intermediate results materialization than on actual query execution due to the limited functionalities of PyTorch. For example, PyTorch documentation indicates that *idx_select* function (second kernel) indexes a tensor to create a new tensor. It is used to create a new column after a predicate evaluation, but its overhead is high compared to the other kernels. The creation of intermediate results not only causes storage overhead due to the limited GPU device memory space but also incurs significant costs for the entire query execution.

5.2 Warp Execution Efficiency

We have quantified the number of operations and the amount of data each system processes for each query. We now turn to analyzing the impact of different implementations on execution efficiency by presenting microarchitecture-level performance statistics.

Table 5: GPU scheduler statistics for different warp states – ACTIVE: total warps allocated. ELIGIBLE: warps ready to issue instruction (includes issued warps). STALLED: warps cannot issue instruction. ISSUED: warps selected by the scheduler. Table units are *average number of warps per scheduler per cycle.*

System	Active	Eligible	Stalled	Issued
CRYSTAL	11.08	0.16	10.92	0.1
HEAVYDB	7.94	0.16	7.78	0.14
BLAZINGSQL	8.33	1.97	6.36	0.72
TQP	12.52	0.45	12.07	0.18

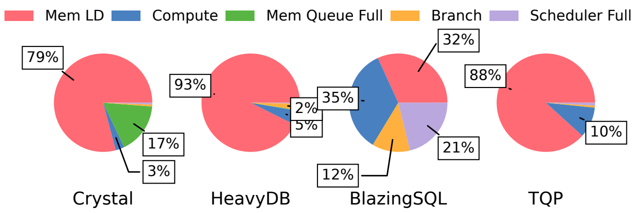


Figure 9: Warp stall causes breakdown – Q21 for all systems.

5.2.1 Warp Bottleneck. One advantage of GPUs is their ability to hide different types of execution stalls with their massive parallelism. If warp x encounters any stopping event (*e.g.,* waiting for instruction fetch or waiting for compute resources to become free), the GPU will quickly schedule other warps that are ready to be executed and have available resources. This approach improves the number of executed instructions per cycle. Table 5 presents the average number of warps per scheduler per cycle in the GPU. For NVIDIA GPUs, each scheduler issues one warp per cycle at its most optimal state. Based on the table, we see that most systems have enough occupancy (*i.e.,* the GPU is occupied with enough warps). However, most systems do not have enough warps that are ready in time. For example, the eligible warp per cycle for CRYSTAL is 0.16. In other words, the scheduler does not have anything to schedule for five out of six cycles. We conclude that since most systems are issue-bound, computation resources after the issue stage are underutilized for OLAP queries. One exception is BLAZINGSQL, which has a high number of both eligible and issued warps.

5.2.2 Warp Stall Breakdown. We then attempt to reason about warp stalls by presenting a breakdown of their causes. The most notable observation, based on Figure 9, is that most systems have the majority of stalls on memory load requests. For HEAVYDB and TQP, they have a small fraction of stalls on computation, meaning that the compute unit is saturated for very few operations. However, for BLAZINGSQL, due to its compute-intensive nature, it becomes more bottlenecked by the computing unit instead. For example, its warps cannot be scheduled due to the busy compute backend and also due to unavailable slots in the scheduler.

5.3 Memory Efficiency

The investigation in the last section shows that most systems are bottlenecked by memory requests. Thus, we analyze the memory efficiency of different systems. Our study focuses on CRYSTAL and HEAVYDB since they represent the most optimal implementations. Additionally, they have different execution approaches, so it is insightful to understand their trade-offs on memory efficiency.

Table 6: GPU memory statistics - Register usage, cache hit rate and number of cache requests for CRYSTAL and HEAVYDB.

System	Register per Thread	L1 Cache Hit (%)	L1 Cache Req. (M)	L2 Cache Hit (%)	L2 Cache Req. (M)
CRYSTAL	40	0.47	19	51.38	127
HEAVYDB	64	17.47	66	89.54	397

Table 6 shows the register and cache usage of both systems. The most notable observation is that the CRYSTAL (bulk execution model) has both worse L1 and L2 cache hit rates compared to HEAVYDB (warp execution model). Nevertheless, in theory, the bulk execution model should have better memory efficiency because of better memory coalescing and better temporal cache reuse. Our study identifies that CRYSTAL has two limitations that cause worse memory efficiency. First, CRYSTAL loads unnecessary data from DRAM, which disrupts its cache locality. Secondly, for CRYSTAL (bulk execution model), it is also more favorable to not bypass the L1 cache, so that data loaded together (*i.e.,* a bulk or a tile) can be reused temporally (CRYSTAL chooses to bypass L1 cache).

6 IMPLEMENTATION OPTIMIZATION

In this section, we discuss optimizations that we have implemented based on the insights gained from the previous analysis.

6.1 Inefficiencies and Optimizations

As mentioned in §2.3, CRYSTAL uses a bulk execution model to improve cache locality. A column is partitioned into multiple tiles, with each tile containing four tuples. This yields the most optimal query execution efficiency.

EXTRA BYTES AND OPERATIONS. The original CRYSTAL work implements a tile-based data loading primitive that loads all tuples for a tile from the GPU DRAM. After evaluating a predicate over a column, it is common for many tuples to become unnecessary, as they do not satisfy the predicate. In this case, the design of the CRYSTAL load primitive is suboptimal as it loads all tuples from the GPU DRAM, even the ones that are not needed. Considering that analytics query execution is mostly stalled due to memory load (§5.2), the extra data loading can cause a performance slowdown. Furthermore, as discussed in §3.1, GPU execution operates in warp granularity, with all threads (*i.e.,* 32) executing instructions in lockstep. During query execution, many threads in a warp find that not all of their assigned tuples satisfy predicates after evaluating a few where clauses, especially for very selective predicates. CRYSTAL does not have any primitive to terminate those threads in time, which causes unnecessary instruction execution on the GPU.

To address this issue, we implement two primitives in CRYSTAL: a predicated loading primitive (*PredLoad*) and a voluntary runtime thread termination primitive (*Term*). Our primitives keep track of valid tuples and idle threads for each tile using a bitmap. During execution, our new primitive performs on-demand tuple loading based on the validity of the tuple stored in the bitmap. In doing so, we avoid loading tuples that have already been invalidated by predicates on other columns. We also monitor the liveness of the thread through the bitmap. If all tuples of a thread become invalid, the

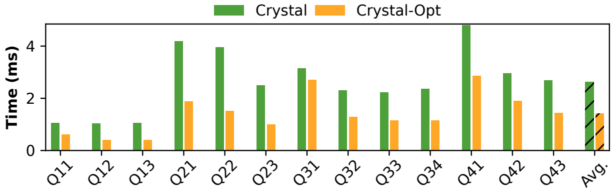


Figure 10: Crystal vs Crystal-Opt. query execution.

Table 7: Performance statistics of Crystal and Crystal-Opt for Q21—Total bytes, total operations, cache hit rate, number of cache requests.

System	Total Bytes (M)	Total Ops (M)	L1 Cache Hit (%)	L1 Cache Req. (M)	L2 Cache Hit (%)	L2 Cache Req. (M)
Crystal	1653	2704	0.47	18	51.49	127
Crystal-Opt	902	2291	41.31	14	67.48	66

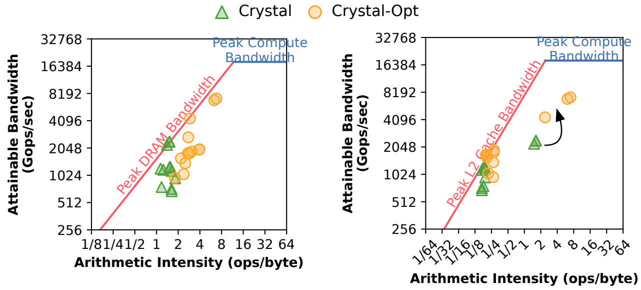
execution chooses to end the thread immediately to avoid executing further instructions. This optimization is particularly beneficial when the entire warp can be terminated early.

Cache Locality. Historically, GPU L1 caches have not been used (L1 cache bypass) during execution because of their limited capacity, particularly for workloads involving many random accesses. However, in our experiments, we discover that the increased L1 cache capacity can bring additional performance benefits by improving the temporal data locality. This is especially true since data in a tile tends to be reused often in the bulk execution model. In contrast to Crystal's approach, we enable the L1 cache by adjusting the compilation flag supported by CUDA GPUs.

6.2 Performance Speedup

We combine all proposed optimizations as Crystal-Opt. In Figure 10, we compare the query execution time of both Crystal and Crystal-Opt. Crystal-Opt achieves an average speedup of 1.9× over Crystal. The speedup is more significant for selective queries, but not as prominent for less selective queries (*e.g.,* Q31). We also conduct a case study for Q21 to understand the reasons for the speedup (shown in Table 7). First, we can see that by using the newly introduced primitives, Crystal-Opt significantly reduces the total bytes and operations compared to Crystal. As a result of the reduced total data read from DRAM, the number of requests to the cache system is also reduced. Second, by enabling the L1 cache, Crystal-Opt increases the cache hit rates for both L1 and L2 caches. Due to the improved L1 cache hit rate, the total number of requests to the L2 cache is also reduced.

We also investigate the impact of Crystal optimization on resource utilization of both DRAM and L2 cache. Figure 11 demonstrates that the AI for GPU DRAM is increased because our optimization reduces unnecessary data brought from GPU DRAM. As a result, the maximum possible attainable bandwidth is also increased, especially for queries Q11, Q12, and Q13. For L2 cache utilization, our optimization on cache locality also improves the attainable bandwidth of queries that have hash join. Compared to Crystal, Crystal-Opt now has more queries that fully saturate the maximum L2 cache bandwidth. Even for queries that do not have hash join (the right-most three dots in Figure 11), their attainable bandwidths are also improved by having better cache locality.



(a) DRAM roofline Model.

(b) L2 roofline Model.

Figure 11: Resource utilization between Crystal and Crystal-Opt – DRAM and L2 cache utilization.

7 RESOURCE UTILIZATION OPTIMIZATION

Our characterization in §5 demonstrates that many queries in GPU DBMSs do not fully saturate the existing GPU resources. As shown in §6, one way to improve resource utilization is to optimize the system implementation. However, with the resource-allocation features in newer generation GPUs (e.g., MIG [41]), DBMSs can also improve resource utilization by running multiple queries concurrently inside a single GPU, in an implementation-agnostic way.

One unsolved challenge of doing concurrent query execution is that it does not always offer performance speedup, especially when a GPU resource is already saturated. Therefore, we need an analytical model to reason about performance improvements given the available GPU resources. In this section, we demonstrate that we can use the roofline model as a guideline to reason about performance speedup when the allocated GPU resources are adjusted.

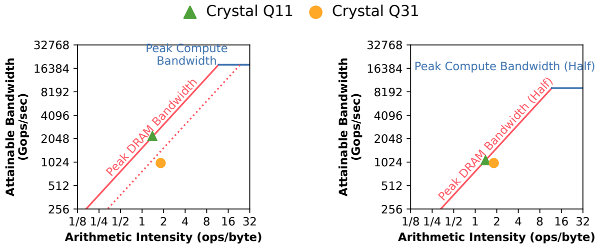
7.1 Model-Driven Resource Allocation

We show how to use the roofline model to estimate the GPU execution performance changes with respect to resource allocation, which can then be used to determine the optimal configuration. The model uses runtime information, which can be obtained from prior executions of recurring queries or from a representative workload.

We use GPU DRAM as the target resource for illustration. Let t denote the query time under the current resource allocation and let $\text{Bandwidth}_{DRAM'}$ denote the new DRAM bandwidth. We predict the new query GPU execution time t′ by using the following equation.

$$t' = \max\left(t, \frac{\#\text{ of Int. Ops}}{\text{AI}_{DRAM} \times \text{Bandwidth}_{DRAM'}}\right)$$

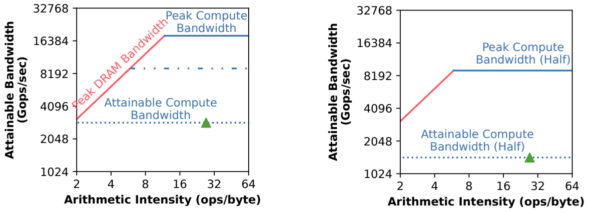
This equation is proposed based on the observation that *AI is determined by the implementation of the query and is unlikely to change when the resource allocation changes.* Our model picks the maximum among the two terms in the equation—the first term for scenarios where bandwidth is not a bottleneck (and so, the time remains unchanged), and the second for scenarios where the change in memory bandwidth hurts performance. For the latter case, the denominator in the fraction is the maximum throughput at the given AI and allocated bandwidth. In this case, the query time is the time to execute the total integer operations at that throughput. Figure 12 shows these two scenarios using representative queries



(a) Query performance, memory bandwidth, and compute bandwidth of *full* GPU resource (*dashed line is projected memory bandwidth of half GPU resource*).

(b) Query performance, memory bandwidth, and compute bandwidth of *half* GPU resource allocation.

Figure 12: Memory-bound queries – Performance impact on memory-bound queries (Q11 and Q31) for Crystal.



(a) Query performance, memory bandwidth, and compute bandwidth of *full* GPU resource (*dashed-dotted line projects compute bandwidth of half GPU resource*).

(b) Query performance, memory bandwidth, and compute bandwidth of *half* GPU resource allocation.

Figure 13: Compute-bound queries – Performance impact on compute-bound queries (Q34 from BlazingSQL).

(Q11, Q31) from Crystal, and full ⟶ half allocation change. Q31 underutilizes the DRAM bandwidth and has no performance impact, whereas Q11 loses throughput (geometrically, the point shifts downwards) since it saturates the bandwidth. Finally, we compute $\text{Slowdown}_{DRAM} = t'/t$. Visually, in Figure 12, the green dot moves down but not the yellow dot (no slowdown).

L2 Bandwidth. As previously discussed, the latest GPUs [39] support both L2 and DRAM bandwidth partitioning. Similar to the above method, we can estimate the execution time due to L2 resource allocation using the L2 roofline model. One challenge is how to combine both the DRAM and L2 roofline models into a unified model to estimate the query slowdown. We solve this by using a max function to estimate the total slowdown as follows.

$$\text{Slowdown} = \max\left(\text{Slowdown}_{DRAM}, \text{Slowdown}_{L2Cache}\right) \qquad (1)$$

The rationale is that we empirically find that *queries can rarely be bottlenecked by both memory resources (L2 and DRAM)*. For example, if a query has very high utilization of the L2 cache bandwidth (*i.e.,* it is bottlenecked by the L2), it generates only minimal traffic to DRAM, so it will not be affected by changes in DRAM bandwidth. Hence, one of the estimated slowdown terms is likely to be 1 (no slowdown). Thus, we only need to use the max function to get the dominating slowdown value.

Compute-Bound. Next, we explain slowdown estimation for compute-bound queries using BlazingSQL, because it has the most compute-bound implementations compared to the other systems.

Figure 13 left shows Q34 attainable bandwidth (*i.e.,* throughput) and AI (2886.74 Gops/sec and 27.15 ops/byte respectively). The peak compute bandwidth of the full GPU is 18247.00 Gops/sec and that for half of the GPU resources is near 9123.50 Gops/sec (dashed and dotted line in Figure 13 left). It is clear that even the peak compute bandwidth for half of the GPU is beyond the attainable bandwidth of Q34, so the traditional roofline model would predict no performance slowdown. Nevertheless, this is not the case as a result of the attainable compute bandwidth per SM being less than its peak due to execution inefficiencies (*e.g.,* memory stalls). The overall compute bandwidth (attainable or peak) is the per-SM value × the number of SMs. When the GPU allocates fewer compute resources, it reduces the number of SMs allocated, but it cannot improve the execution efficiency (*i.e.,* attainable compute bandwidth) of each SM. As a result, the overall attainable compute bandwidth will decrease (Figure 13(b)). To estimate the resulting slowdown, we can simply use the ratio of resource allocations reduction as follows.

$$\text{Slowdown}_{Compute} = \frac{1}{\text{ComputeAllocationRatio}} \qquad (2)$$

For example, if the GPU compute resources are halved, the attainable bandwidth can be calculated as half of the original attainable bandwidth with full GPU resources.

Unified Model. Now that we have proposed two models for estimating slowdowns with changing allocations—one for memory resources and one for compute resources, the last step is to determine which model to use. We use a simple, commonly-used heuristic [36, 46] to determine whether an application is compute- or memory-bound. As shown below, we can determine if an application tends to be compute-bound based on the AI and peak compute and DRAM bandwidths of the GPU. The final Slowdown

$$= \begin{cases} (2), \text{ if } \text{AI}_{DRAM} > \dfrac{\text{Bandwidth}_{Compute}}{\text{Bandwidth}_{DRAM}} \text{ or } \text{AI}_{L2Cache} > \dfrac{\text{Bandwidth}_{Compute}}{\text{Bandwidth}_{L2Cache}} \\ (1), \text{ otherwise} \end{cases}$$

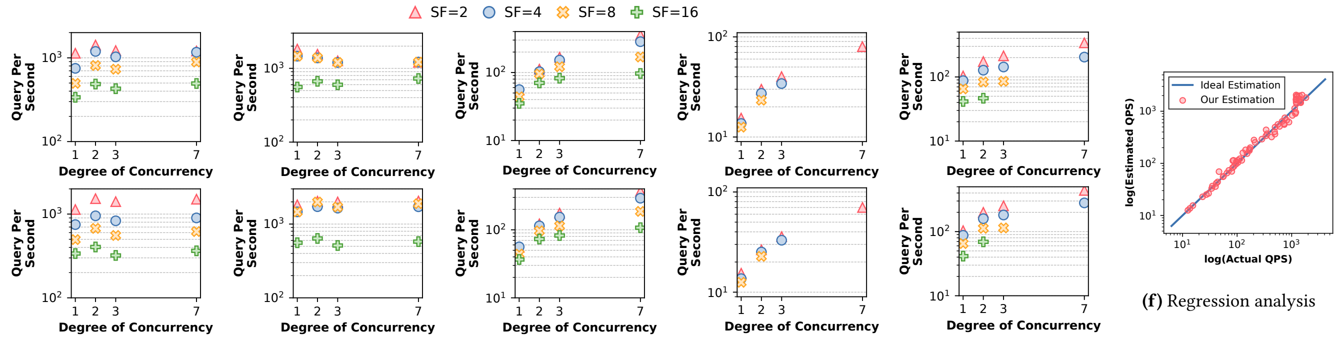
If the application is more compute-bound, then we use the model to account for compute bandwidth reduction. Otherwise, we apply the model for DRAM or L2 cache bandwidth reduction.

This approach can be more easily used to (1) estimate the GPU execution performance impact of downsizing both memory and compute resources, or (2) reason about the performance impact of upsizing compute resources. However, it could have inaccurate estimations for the performance impact of upsizing memory resources when they are no longer a bottleneck.

7.2 Model-Driven Concurrent Scheduling

Next, we extend the model to estimate the end-to-end performance impact for different degrees of concurrency. Using our analytical model, users can accurately predict the resulting end-to-end performance when running queries concurrently on multiple GPU partitions. Our model assumes that the scheduler uses a round-robin scheduling algorithm for load balancing.

CPU and Constant Overhead. To construct the model, we first need to consider additional overheads for query executions. For CPU overhead, our model includes the overhead of query optimization and compilation. For some systems (*e.g.,* HeavyDB, BlazingSQL), even though the same query has already been optimized and compiled to binary form, each query invocation still introduces some constant overhead on the CPU side. For all systems, we also consider those overheads. Later, we show insights into how



(a) Crystal (b) Crystal-Opt (c) HeavyDB (d) BlazingSQL (e) TQP (f) Regression analysis

Figure 14: Throughput *vs.* degree of concurrency – Comparison between actual and estimated throughput of running queries concurrently (we omit data points for BlazingSQL and TQP where they run into out-of-memory issues at larger scale factors). For sub-figures (a), (b), (c), (d), and (e), the top plot shows the actual throughput whereas the bottom plot shows the estimated throughput.

concurrency is also beneficial to alleviate those overheads. We consider two additional major overheads—GPU setup overhead, which includes GPU context initialization and memory allocations, and data transfer overhead. All systems cache tables on the GPU for future query executions, so the data transfer overhead is also only a one-time cost.

End-to-End Performance. We can now estimate the end-to-end query execution time individually for each process. When the system varies the resources, the model will adjust the query GPU execution time. Other overheads will remain unchanged. Now, to consider the time from multiple concurrent processes P, we use a max function because the longest-running process will determine the end-to-end query execution time for concurrent executions.

$$\text{ExecTime} = \max\left(\text{ExecTime}_{P_1}, \text{ExecTime}_{P_2} \ldots \text{ExecTime}_{P_n}\right)$$

Our model can also be used to estimate query performance *vs.* degree of concurrency for MPS (§2.1), excluding accounting for interference in accessing the shared L2 cache and DRAM.

7.3 Resource Allocation Study

In this study, we focus on answering the following questions:

- **RQ1 – How beneficial is concurrent query execution?**
- **RQ2 – How good is our model for estimating the query execution performance *vs.* the degree of concurrency?**
- **RQ3 – What are the trade-offs between MIG and MPS?**

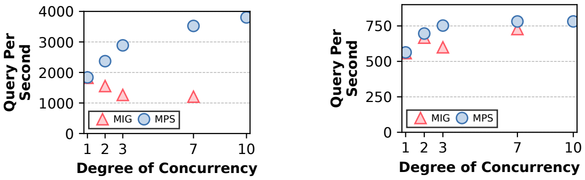
RQ1 – Benefit of Concurrent Query Execution. We implemented concurrent query execution on the four systems and designed the following experiment to evaluate the benefits with Degree of Concurrency (DoC) = 2, 3, and 7, over DoC=1 (no concurrency), for SSB queries [3]. We configured the GPU, through MIG, to support the required DoC and started the corresponding number of system instances, one on each GPU partition. We construct a simple scheduler to dispatch SSB queries in a randomized sequence to each system instance, repeated 1000 times. We then measure the overall throughput (queries per second *i.e.,* QPS) for the GPU.

Figure 14 (top) shows the actual (measured) throughput for different DoC. For large-scale factors (*e.g.,* 8 and 16), we omit query throughput for systems that run into out-of-memory issues. For Crystal, the throughput initially increases by an average of 1.5× as the DoC changes from 1 to 2. Because Crystal spends the majority of the time on GPU execution, the performance speedup mostly comes from the improved resource utilization of the GPU hardware. However, as the DoC increases beyond 2, it does not gain further speedup, because resources are already very close to being fully saturated when running two processes concurrently. In fact, for DoC=3, the throughput is lower than for DoC=2. A further reason is that MIG cannot evenly divide the GPU resource into three partitions. For instance, some of the GPU DRAM banks and L2 cache slices are not being used at all when DoC=3.

Crystal-Opt achieves a 1.3× throughput improvement compared to Crystal-Opt without MIG at scale factor 16, while it does not gain better throughput at smaller scale factors. This is because GPU execution time dominates other overheads for larger scale factors. Increasing the DoC initially reduces GPU execution time, but since GPU resource utilization is quickly saturated, a higher DoC cannot further improve the overall throughput. In contrast, other overheads like data and results movement dominate end-to-end execution for smaller scale factors. Partitioning the GPU also partitions the PCI-e bandwidth, so the overhead grows proportionally as the DoC increases. We also observe that Crystal-Opt with MIG has a 2.2× speedup over Crystal with no MIG, while Crystal only has a 1.5× speedup after enabling MIG. This demonstrates that both system implementations and resource allocations are critical.

Concurrent execution provides sub-linear performance improvement for both HeavyDB and BlazingSQL. For those two systems, queries at small scale factors do not fully saturate resources. More queries can finish within a fixed time with concurrent execution, so the overall throughput increases. In contrast, queries are more easily affected by resource reduction at larger scale factors, leading to a significant increase in query latency that overshadows the gains of concurrent execution. However, concurrency is still able to hide CPU latency overhead, leading to substantial throughput improvements for SF=8 and SF=16, respectively, at DoC=7 (over DoC=1). Concurrent query executions also provide sub-linear performance speedup for TQP. TQP is shown to have low GPU utilization, so most of the performance gain comes from better utilization.

[3] For A100 MIG partitions [42], compute resource has a total of 7 slices but memory resource has a total of 8 slices. When the GPU is partitioned, some resources will end up being idle because they cannot be evenly distributed. We choose the DoC that minimizes the amount of idle GPU resources when partitioned.



(a) SF=2

(b) SF=16

Figure 15: Trade-offs between MIG and MPS – Throughput *vs.* degree of concurrency with different mechanisms tested on Crystal-Opt.

RQ2 – Model Accuracy on Degree of Concurrency. Figure 14 (bottom) shows the estimated overall throughput vs. DoC for the above experimental setting. Figure 14 (f) indicates the prediction accuracy of the model compared to the ideal prediction (estimated = actual). We use our models to estimate end-to-end query execution time for all DoC using profiling data from a single run of the query with full GPU allocation (DoC=1). Our estimated throughputs are close to the actual, for both absolute values and scaling trends. The correlation coefficient between actual and estimated throughput is 0.95, thereby indicating strong correlation. The 50^{th} and 95^{th} percentile of relative errors (= $\frac{|\text{Actual}-\text{Estimated}|}{\text{Actual}}$) are $\leq$ 0.11 and $\leq$ 0.46, while the average is 0.15. The evaluation demonstrates that our model is able to help system administrators determine how to allocate GPU resources and configure the level of query execution concurrency to achieve the best performance.

RQ3 – MIG and MPS Trade-offs. We present an analysis (Figure 15) to compare the trade-offs of executing a batch of queries between MIG and MPS on Crystal-Opt. For SF=2, MPS offers better scalability than MIG. Since all clients share the same PCI-e interconnect, MPS allows different processes to overlap compute with data transfer with the full interconnect bandwidth, considering the fact that the bandwidth is not fully saturated temporally. In contrast, the PCI-e bandwidth is partitioned among different processes in MIG, and thus, each process gets reduced PCI-e bandwidth. For SF=16, MIG and MPS show similar performance. This is because compute overhead dominates at larger scale factors, and thus the benefit of interleaving reduces. In general, MIG is favored for scaling performance without interference between partitions, especially when GPU execution time dominates. In contrast, MPS provides better data and computation overlap when data movement overhead dominates the end-to-end execution time.

8 RELATED WORK

Suh *et al.* [62] is the most recent study to compare different GPU database systems. They only focus on end-to-end execution time comparison of different systems on different GPUs. However, our paper shares more insights into the internals of GPU execution efficiency. Yin and Yang [70] study GPU execution of SSB [52], and present the time breakdown between kernels and data transfers, but never examine the efficiency of each kernel. Pump up the Volume [33] and Triton join [34] study the performance impact from newer CPU-GPU interconnects on out-of-GPU hash-join. GPL [50] also observes that the GPU can be underutilized during query execution. Furst *et al.* [19] study GPU kernel efficiency but only focus on comparing GPU occupancy *vs.* instruction types. Funke *et al.* [17, 18] and Paul *et al.* [49] optimize JIT compilers for better GPU execution efficiency through kernel fusions and thread divergence elimination. Sioulas *et al.* [60] implement partitioned hash join in GPU. Shanbhag *et al.* [58] also extend the Crystal library for in-GPU compression. Instead, HippogriffDB [31] accelerates query performance by supporting GPU execution on compressed data. Mordred [68] and HetExchange [13] explore CPU-GPU query executions. Rosenfeld *et al.* [53] provide an in-depth overview of CPU-GPU database systems. GaccO [12] studies transactional query processing on GPUs. MGJoin [51] and Maltenberger *et al.* [35] evaluate join and sorting algorithms for multi-GPU systems. Doraiswamy and Freire [15] propose using GPUs to process spatial data (*e.g.,* geometric objects) in database systems. Our work provides an in-depth microarchitectural analysis of existing systems.

GPU performance modeling. Hong and Kim [25] model the performance of early generation GPUs, where GPU caches were still not mature, so the focus is on GPU DRAM and compute bandwidths. Zhang *et al.* [71] provide performance optimization suggestions to developers through GPU performance modeling with microbenchmarking profiling. Wu *et al.* [66] instead propose using a machine learning approach to predict kernel performance, which also requires profiling performance counters on real hardware. Baghsorkhi *et al.* [9] use code analysis to consider the performance impact from control flow divergence and memory bank conflicts. Our approach of using the roofline models provides a simple way to analyze the performance impact for different GPU resources without requiring code analysis or machine learning models.

Gables [24] uses roofline models to study SoC platforms with multiple accelerators. Ding *et al.* [14] and Lopes *et al.* [32] also apply roofline models to GPUs. Ding *et al.* capture all instructions for AI, which can lead to inaccurate estimations. Lopes *et al.* also explore the cache-aware aspect but do not differentiate between AI at the cache level and at the DRAM level, in contrast to our work. Additionally, we extend and apply the roofline model to estimate query performance for different resource allocations on the GPU. This has not been explored in prior work.

Concurrent execution in GPUs. Yu *et al.* [69] have also surveyed the trade-off between MIG and MPS. Their paper shares the potential opportunities and use cases of using concurrent execution in GPUs. Tan *et al.* [63] explore accelerating deep neural network (DNN) inference by using MIG. Kass *et al.* [27] instead investigate DNN training in MIG. In our work, we support relational query operations in the form of concurrent execution, which is often limited by the GPU memory resource. Instead, DNN inference is more compute-bound. Our work uses simple models to estimate performance with different GPU resource limits, whereas others [63] require profiling the execution for different configurations.

9 CONCLUSIONS

In this work, we provide microarchitectural insights for existing GPU database systems. We demonstrate how using our insights, we can improve the performance of a state-of-the-art database system by 1.9×. Additionally, we show how these insights can be further leveraged into an analytical model—predicting resource utilization under concurrent query execution—delivering up to a 6.5× better throughput. We expect these contributions to drive further research and development in the field of GPU-accelerated data analytics.

REFERENCES

[1] 2017. PCIe 4.0 specification finally out with 16 GT/s on tap. [Online] Available from: https://techreport.com/news/32064/pcie-4-0-specification-finally-out-with-16-gts-on-tap/.

[2] 2019. PCI-SIG Achieves 32GT/s with New PCI Express 5.0 Specification. [Online] Available from: https://www.businesswire.com/news/home/20190529005766/en/PCI-SIG%C2%AE-Achieves-32GTs-with-New-PCI-Express%C2%AE-5.0-Specification.

[3] 2022. PCI-SIG Announces PCI Express 7.0 Specification to Reach 128 GT/s. [Online] Available from: https://www.businesswire.com/news/home/20220621005137/en.

[4] 2022. PCI-SIG Releases PCIe 6.0 Specification Delivering Record Performance to Power Big Data Applications. [Online] Available from: https://www.businesswire.com/news/home/20220111005011/en/PCI-SIG%C2%AE-Releases-PCIe%C2%AE-6.0-Specification-Delivering-Record-Performance-to-Power-Big-Data-Applications.

[5] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. 1999. DBMSs On A Modern Processor: Where Does Time Go? *PVLDB* (1999).

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[7] Rob Armstrong, Arthy Sundaram, and Fred Oh. 2021. Revealing New Features in the CUDA 11.5 Toolkit. [Online] Available from: https://developer.nvidia.com/blog/revealing-new-features-in-the-cuda-11-5-toolkit/.

[8] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *PVLDB* (2022), 3598–3601.

[9] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. 2010. An Adaptive Performance Modeling Tool for GPU Architectures. In *PPoPP*. 10.

[10] Peter Bakkum and Srimat Chakradhar. 2010. Efficient Data Management for GPU Databases. https://github.com/bakks/virginian/.

[11] BlazingSQL. 2021. BlazingSQL. https://github.com/BlazingDB/blazingsql.

[12] Nils Boeschen and Carsten Binnig. 2022. GaccO - A GPU-accelerated OLTP DBMS. In *SIGMOD*. 1003–1016.

[13] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *PVLDB* (2019), 544–556.

[14] Nan Ding and Samuel Williams. 2019. An Instruction Roofline Model for GPUs. In *PBMS*. 7–18.

[15] Harish Doraiswamy and Juliana Freire. 2020. A GPU-friendly Geometric Data Model and Algebra for Spatial Queries. In *SIGMOD*. 1875–1885.

[16] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. NestGPU: Nested Query Processing on GPU. In *ICDE*. 1008–1019.

[17] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*. 1603–1618.

[18] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *PVLDB* (2020), 884–897.

[19] Emily Furst, Mark Oskin, and Bill Howe. 2017. Profiling a GPU database implementation: a holistic view of GPU resource utilization on TPC-H queries. In *DaMON*. 1–6.

[20] Apurva Gandhi, Yuki Asada, Victor Fu, Advitya Gemawat, Lihao Zhang, Rathijit Sen, Carlo Curino, Jesús Camacho-Rodríguez, and Matteo Interlandi. 2022. The Tensor Data Platform: Towards an AI-centric Database System. In *CIDR*.

[21] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *PVLDB* (2022), 2811–2825.

[22] HeavyDB. 2022. HeavyDB. https://github.com/heavyai/heavydb.

[23] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *PVLDB* (2013), 709–720.

[24] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *HPCA*. 317–330.

[25] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In *ISCA*.

[26] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-Aware Roofline Model: Upgrading the Loft. *IEEE CAL* (2014), 21–24.

[27] Anders Friis Kaas, Stilyan Petrov Paleykov, Ties Robroek, and Pınar Tözün. 2022. Deep Learning Training on Multi-Instance GPUs. arXiv:2209.06018 [cs.LG]

[28] KaiGai Kohei. 2022. PG-Strom. https://github.com/heterodb/pg-strom.

[29] Alexander Krolik, Clark Verbrugge, and Laurie Hendren. 2021. R3d3: Optimized Query Compilation on GPUs. In *CGO*. 277–288.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. 75–88.

[31] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *PVLDB* (2016), 1647–1658.

[32] André Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. 2017. Exploring GPU Performance, Power and Energy-Efficiency Bounds with Cache-aware Roofline Modeling. In *ISPASS*. 259–268.

[33] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *SIGMOD*. 1633–1649.

[34] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *SIGMOD*. 1017–1032.

[35] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *SIGMOD*. 1795–1809.

[36] Lei Mao. 2021. Math-Bound VS Memory-Bound Operations. https://leimao.github.io/blog/Math-Bound-VS-Memory-Bound-Operations/.

[37] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* (2011), 539–550.

[38] NVIDIA. 2016. nvidia-smi Documentation. [Online] Available from: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[39] NVIDIA. 2020. NVIDIA A100 TENSOR CORE GPU Unprecedented Acceleration at Every Scale. [Online] Available from: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf.

[40] NVIDIA. 2021. NVIDIA Multi-Process Service Introduction. [Online] Available from: https://docs.nvidia.com/deploy/mps/index.html.

[41] NVIDIA. 2022. NVIDIA Multi-Instance GPU. [Online] Available from: https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[42] NVIDIA. 2022. NVIDIA Multi-Instance GPU User Guide. [Online] Available from: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[43] NVIDIA. 2022. NVIDIA NSight Systems User Guide. [Online] Available from: https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[44] NVIDIA. 2022. Parallel Thread Execution ISA Version 7.8. [Online] Available from: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[45] NVIDIA. 2022. Thrust. [Online] Availble from: https://docs.nvidia.com/cuda/thrust/index.html.

[46] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel. 2014. Applying the Roofline Model. In *ISPASS*. 76–85.

[47] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*. 237–252.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[49] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving Execution Efficiency of Just-in-Time Compilation Based Query Processing on GPUs. *PVLDB* (2020), 202–214.

[50] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based Pipelined Query Processing Engine. In *SIGMOD*. 1935–1950.

[51] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *SIGMOD*. 1413–1425.

[52] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick O'Neil, and Elizabeth O'Neil. 2013. Variations of the Star Schema Benchmark to Test the Effects of Data Skew on Query Performance. In *ICPE*. 361.

[53] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *Comput. Surveys* (2023), 1–38.

[54] Rathijit Sen and Karthik Ramachandra. 2018. Characterizing resource sensitivity of database workloads. In *HPCA*. 657–669.

[55] Rathijit Sen and Yuanyuan Tian. 2023. Microarchitectural Analysis of Graph BI Queries on RDBMS. In *DaMoN*. 102–106.

[56] Anil Shanbhag. 2020. Crystal GPU Library. https://github.com/anilshanbhag/crystal.

[57] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *SIGMOD*. 1617–1632.

[58] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-Based Lightweight Integer Compression in GPU. In *SIGMOD*. 1390–1403.

[59] Jian Shen, Ze Wang, David Wang, Jeremy Shi, and Steven Chen. 2019. AresDB. https://github.com/uber/aresdb.

[60] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE*. 698–709.

[61] Utku Sirin and Anastasia Ailamaki. 2020. Micro-Architectural Analysis of OLAP: Limitations and Opportunities. *PVLDB* (2020), 840–853.

[62] Young-Kyoon Suh, Junyoung An, Byungchul Tak, and Gap-Joo Na. 2022. A Comprehensive Empirical Study of Query Performance Across GPU DBMSes. *SIGMETRICS* (2022), 1–29.

[63] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem. arXiv:2109.11067 [cs.DC]

[64] RAPIDS Development Team. 2018. RAPIDS: Collection of Libraries for End to End GPU Data Science. https://rapids.ai

[65] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* (2009), 65–76.

[66] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU Performance and Power Estimation Using Machine Learning. In *HPCA*. 564–576.

[67] 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 *CGO*. 44–54.

[68] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *PVLDB* (2022), 2491–2503.

[69] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A Survey of Multi-Tenant Deep Learning Inference on GPU. arXiv:2203.09040 [cs.DC]

[70] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *PVLDB* (2013), 817–828.

[71] Yao Zhang and John D. Owens. 2011. A Quantitative Performance Analysis Model for GPU Architectures. In *HCPA*. 382–393.