Heterogeneous Intra-Pipeline Device-Parallel Aggregations

Artem Kroviakov[1]
TUM
kroviakov@in.tum.de

Petr Kurapov
Intel Deutschland GmbH
petr.a.kurapov@intel.com

Christoph Anneser
TUM
anneser@in.tum.de

Jana Giceva
TUM
jana.giceva@in.tum.de

ABSTRACT

The rising hardware heterogeneity in modern systems emphasizes new dimensions of optimizing task execution for data processing frameworks. Specialized hardware is often expected to be the exclusive executor of some particular workload because it was designed for it or is simply the fastest option. In heterogeneous database systems, almost always, the entire operation offloading is considered. However, little attention was given to database systems with horizontal cross-device pipeline parallelization. We argue that such an approach can be applied to systems with morsel-driven parallelism and improve performance. We apply our parallelization strategy to an existing system and accelerate aggregations using two devices by up to 1.5x compared to the fastest exclusive device executor.

CCS CONCEPTS

• **Information systems** → **Database query processing**; *Main memory engines*; **Online analytical processing engines**; **DBMS engine architectures**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

KEYWORDS

Query engine, heterogeneous query processing, dedicated GPUs

ACM Reference Format:
Artem Kroviakov, Petr Kurapov, Christoph Anneser, and Jana Giceva. 2024. Heterogeneous Intra-Pipeline Device-Parallel Aggregations. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3662010.3663441

1 INTRODUCTION

Device heterogeneity is rising and accelerators like GPUs are becoming omnipresent in modern systems [11, 25, 31]. Thus, it is unsurprising that database systems explore various ways to best exploit them [22]. A majority of the research attention has been either on (1) how to facilitate offloading part of or a full query onto an accelerator like the GPU [14, 19, 22, 24, 27, 35] or (2) on how to work around various HW-related constraints (e.g., limited on-device memory capacity or interconnect bandwidth between the devices) [38, 39] with techniques such as caching, overlapping computation and data transfer costs, or pre-processing tasks on the CPU (filters, predicate evaluation) before invoking the GPU.

Most existing systems that support cross-device execution leverage the so-called *vertical device parallelism* [15, 33, 34] and constrain the execution of a particular task to a single device. However, given the rise in complexity of data processing tasks and the increase in data volumes, it is safe to assume that many jobs can benefit from spreading both the workload and the data for co-execution on both processing devices (i.e., the many CPU cores and the GPU). Unfortunately, designing a unified system supporting both types of devices is non-trivial. The first major issue is in their different programming models, which can exacerbate if the engine wants to support accelerators from various vendors. A poor design decision can easily increase code complexity and decrease the system's maintainability. The second major issue is the difference in their execution model and how they support parallelism. For example, CPUs are task-parallel executors that focus on minimizing the given task's latency, whereas GPUs are data-parallel executors that focus on maximizing the throughput. The former has many cores that can operate on different data shares and work on various smaller tasks concurrently. In contrast, the latter requires a larger and more complex kernel function that can leverage the warp-based execution but needs to work on a single task (with notable exceptions) and a larger data chunk to saturate the device capacity and amortize the data-transfer costs. Supporting both types of operation within a single unified system remains a relatively unexplored problem for our community. Yet, if solved, it can unleash a great potential for the growing demand for cross-device co-execution.

In the remainder of the paper, we will refer to that problem as supporting a *horizontal device parallelism*, i.e., architecting the foundation of a system that can leverage all processing units concurrently despite their heterogeneity. One notable prior work proposed an ingenious approach that supports horizontal device parallelism based on the exchange operator (HetExchange [21]), which is compatible with many systems supporting Volcano-based parallelism.

This paper explores an alternative design to enable horizontal device parallelism on an intra-pipeline level for systems that internally use task-based scheduling following the morsel-driven parallelism model [36]. More specifically, we propose a novel approach that can facilitate the efficient execution of analytical queries in heterogeneous CPU-GPU database systems based on *fragment-based parallelism*. Furthermore, we describe our prototype implementation in Heterogeneous Data Kernels (HDK) [3], which is an execution backend for analytical queries based on OmniSciDB [54] along with a discussion on potential optimizations. OmnisciDB (now HeavyDB [4]) represents a meaningful baseline as it is GPU-centric and shows reasonable GPU resource utilization [20]. Finally, we explore the importance of determining the suitable fragment size to experimentally evaluate when co-execution pays off and identify potential contention spots.

[1]Work done while at Intel.

DaMoN '24, June 10, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663441

2 RELATED WORK

Heterogeneous systems are often used for CPU-GPU setups [40]. Such systems showed success in areas like scientific computing [32, 42, 48] and ML frameworks (e.g., TensorFlow [10], PyTorch [43]).

Heterogeneous databases have received a lot of attention in the last decade [16, 18, 21, 30, 44, 45, 47] and were extensively discussed in the context of query acceleration [14, 19, 22, 24, 27, 35]. The research converges to the bottleneck of interconnects and query compilation, focusing on what (e.g., operator, pipeline) to execute where (e.g., CPU, GPU). A recent survey [46] provides a good overview of the existing CPU-GPU systems and classifies their approaches. For instance, SABER [34] schedules *queries* either on CPU or GPU based on the execution time estimates. HetExchange [21] and CoGaDB [15] / GAT [52] take a step further and allow the decision to be made on a *query pipeline* and *operator* granularity. They differ in the policy on when to schedule an operator to the GPU: either on data transfer costs [15], or the compute intensity [52]. HERO [33] is a *task-parallel* system that places *sub-operators* onto execution islands (logical PU groups with known intermediate cardinalities) based on data locality and cardinalities. Various hardware architectures can also be found within one device [41].

Device-parallel databases. Previous work has shown the benefits of device-parallel execution of *operations* [17], *sub-operators* [33] or sub-pipelines [23]. Integrated CPU-GPU systems are a good platform for heterogeneous execution [53] and device-parallel in particular [29] as both devices can directly access the main memory. GPUs are treated as another CPU-like accelerator without the need to explicitly address various interconnects [37], dedicated caching mechanisms, etc. This complexity relaxation allows simple fine-grained parallelism [29] where devices divide input data per pipeline. Our approach (cf. Section 3) also applies to such systems.

Mordred [51] integrates operators from the Crystal [47] library, enabling device-parallel query execution of *sub-column segments* and semantic-aware caching. An execution plan is determined for each sub-column segment, and segments with the same plan are assigned to the same segment group. Segment groups are then executed in parallel, enhancing vertical parallelism.

HetExchange [21] is a state-of-the-art parallel query execution framework designed for multi-CPU-multi-GPU servers. It introduces *device-crossing operators* to manage data and control flow transfers between GPU to CPU. They consist of two parts that communicate via an asynchronous queue – one residing on the GPU and the other on the CPU. *Router operators* encapsulate parallelism by efficiently routing tasks between multiple producer and consumer instances based on hash-based and round-robin routing policies that allow for dynamic load-balancing. Router operators do not make any assumptions about the data. Instead, they pass transparent block handles from one to the other operator. Furthermore, HetExchange introduces two data flow operators. The *mem-move operators* facilitate asynchronous data transfers, while the *pack-/unpack operators* optimize these transfers by bundling multiple tuples into blocks to minimize the overhead of moving individual tuples. HetExchange has been integrated with JIT-compiling query engines that fuse these additional operators into efficient pipelines, which are concurrently executed on CPUs, GPUs, or a mix of both.

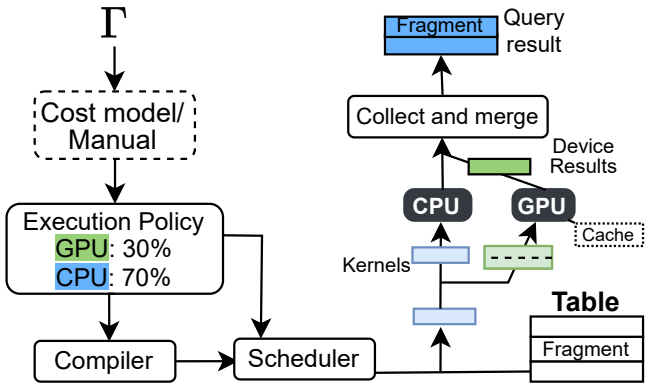


Figure 1: An overview of our approach's execution flow.

SiliconDB [23] presents an execution model for a system with CPU cores and devices that only support a limited set of functions (e.g., FPGAs, DAX engines). Query pipelines are decomposed into finer-grained sub-pipelines, which are then placed into the work queue for the corresponding accelerator. After executing a task, the sub-pipeline's result is materialized, and the next sub-pipeline is pushed to an appropriate queue or gets directly consumed to leverage the proximity of the cached data.

Cost modeling. Heterogeneous databases use cost models to assign work units to devices. Most cost models rely on data locality or execution time estimates of *operators* or *queries* [28, 29, 46, 51]. For example, Mordred [51] models the costs per operator using the Crystal library [47]. Based on the Roofline model [50], Cao et al. [20] analyzed resource utilization across several GPU databases and proposed a new analytical model that estimates the processing time for GPU-accelerated query execution, which also effectively optimizes concurrent query execution. SiliconDB [23] uses a cost model based on queuing theory to adjust the queue size and derive run time estimates. In contrast, we model the cost of data fragment distribution via calibrated graph patterns, as described in more detail in Section 3.

3 APPROACH

HetExchange showed that it is possible to efficiently parallelize a pipeline across devices by following the Volcano-based parallelism (i.e., by leveraging the exchange operator) [26]. Another popular approach in modern databases is morsel-driven parallelism [36]. While both approaches assume batching of data that facilitates work distribution, morsel-driven parallelism allows for a simpler query graph and strives for data locality. To the best of our knowledge and as confirmed by a recent comprehensive survey of heterogeneous query processing systems [46], the morsel-driven approach has not been explored in the context of device-parallel pipeline execution. It is, thus, the focus of this paper.

In this section, we present a novel approach that enables efficient execution of analytical queries in heterogeneous morsel-driven CPU-GPU database systems. For this, we establish the main components that facilitate the key property of scaling pipeline execution horizontally across devices.

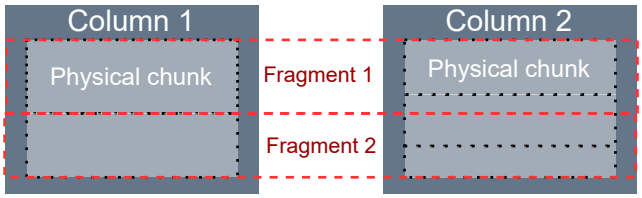


Figure 2: Fragments and a possible data storage layout.

Fragment-based parallelism. Modern processors have hundreds of cores and threads, which our approach aims to utilize by implementing *fragment-based parallelism*. Fragments are horizontal partitions of a data table with a fixed number of tuples. They are a *logical* abstraction guaranteeing that its column's data is stored contiguously in memory during query execution. Figure 2 shows an example of two fragments spanning two columns.

In contrast to morsels [36], we impose no limitations on the GPU's capacity to process only one fragment at a time. Furthermore, instead of employing a direct 1:1 mapping between a task and morsel, our system supports the simultaneous processing of multiple fragments by the GPU, drawing on concepts from adaptive scheduling [49]. Compared to variable-sized morsels, our approach uses fixed-sized fragments to avoid invalidating cached fragments that will trigger expensive data transfers. Finally, a fragment does not impose any restrictions on the storage layout.

During query execution, the database scheduler assigns fragments to processing units (PUs). A PU can either be a single CPU core, a tile of a GPU (e.g., Ponte Vecchio [25]), or the entire GPU. However, the programming models of CPUs and GPUs fundamentally differ: CPUs are task-parallel, while GPUs are data-parallel. On the one hand, working with large fragments can reduce the degree of parallelism and leave some CPU cores idle. On the other hand, the GPU's data parallelism can be best leveraged with coarse-grained fragments. We evaluate the impact of the fragment size on the overall system performance in Figure 4 in Section 4.

Execution model. We do not make any assumptions on the underlying engine's execution model. For a compiled query engine, we can use existing compiler infrastructure to reduce the complexity of supporting various devices and/or vendors (e.g., we can use a multi-level intermediate representation).

Pipeline execution. The work distribution is based on so-called fragments *proportions* that are specified per device by individual policies (cf. Figure 1). The execution policy determines the data flow of the query and can be used to support data locality.

Every pipeline can be described as a *function* that operates on *input buffers* (i.e., fragments) and utilizes *additional memory buffers* (e.g., output buffers). These components jointly constitute a *kernel* (cf. Figure 1), which is an abstraction that our approach uses to pass execution tasks transparently to the database's scheduler to be executed on either device.

Intermediate results. To enable fragment-based parallelism of the next pipeline, a pipeline's result needs to be in the same format as the input buffers (i.e., organized in fragments). For cases where we need a single output, we gather their intermediate results on the CPU and merge them in a post-processing step (e.g., in the case of aggregations).

Scheduling. While we can manually specify the fragment proportions between CPU and GPU, a system should balance the task placement automatically. There are two approaches: implementing a comprehensive cost model and/or relying on adaptive scheduling. We have implemented a cost model that decides on the distribution among devices, optimizing for efficiency but also accounting for the type of computation tasks within the pipeline. Unfortunately, compilation-based techniques make cost estimation of individual operators quite difficult. We would like to note that some of the other cost models mentioned in Section 2 are also applicable to model performance in our system. For example, Cao et al. use the Roofline analysis and previous run times to derive the query execution time [20]. Alternatively, SiliconDB's cost model considers the accelerator's utilization and hardware queue size to determine the maximum number of work elements that can be processed/stored by the accelerator [23].

In contrast, our cost model (cf. Figure 1) identifies computational patterns (e.g., hash group by) from the pipeline graph. These computational patterns, called 'dwarfs' [12], capture the system's typical computations. After establishing the patterns, we measure their execution times across various input sizes and parameters for each device in our hardware setup. This calibration helps place reference points in the problem space for each pattern on a device. We then fit a hyperplane through these points, which allows us to estimate the execution time for arbitrary input parameters.

Calibrating the patterns is costly and time-consuming, as we must evaluate them across a range of realistic and large input sizes. This calibration must be conducted for each hardware configuration the database system runs on. Furthermore, supporting more patterns (e.g., other than aggregations) would also add to the calibration time. However, these costs can be amortized over long-running queries or recurring workloads. Once we detect a calibrated pattern, we search for a minimal execution time on the fitted hyperplane by selecting a suitable input data distribution (i.e., fragments). The proportions are then included in the scheduler's execution policy.

Adaptive scheduling. An alternative approach to balance the fragment distribution is to leverage *adaptive scheduling*, which aims to reduce a pipeline's tail latency by dynamically adjusting the morsel sizes. For example, at the pipeline's end, morsel sizes are progressively decreased to avoid 'stragglers', thereby reducing idle PUs and achieving an effect called 'photo finish' [49]. Existing heterogeneous systems like HetExchange [21] that employ such runtime adaptive scheduling would probably also achieve this effect.

While our approach assumes fixed-size fragments, it could still benefit from adaptive scheduling. For instance, Figure 3 shows a typical CPU-GPU pipeline execution timeline, including the data transfer times for the GPU and the post-processing step. Before the results can be merged, the CPU Processing Units (PUs) must wait for the completion of data transfer from GPU PU#1, which increases the tail latency. Adaptive scheduling could mitigate this issue by dynamically allocating fragments based on each query's estimated remaining processing time. For example, had Fragment 6 been assigned to the CPU rather than the GPU, the pipeline's tail latency could have been reduced.

When does device-parallel acceleration help? Considering Figure 3 again, we observe that device-parallel acceleration is useful when (1) the processed data set has enough fragments, (2) it spends

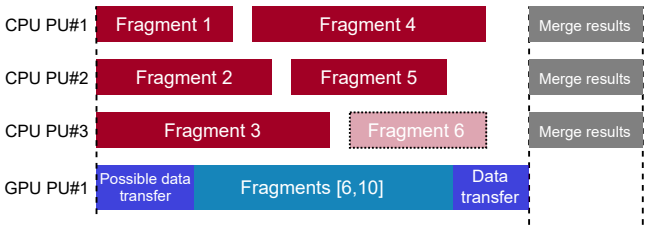


Figure 3: Timeline example of a heterogeneous query pipeline with 10 fragments. Please notice the adaptive scheduling opportunities for fragment 6.

a non-negligible time period performing computation, and (3) it transfers a reasonably sized output. Otherwise, data transfer will outweigh execution time.

Inter-query parallelism. Since executing single heterogeneous queries might not fully utilize the GPUs, fortunately, our approach also allows for inter-query parallelism. We abstract from physical hardware threads of CPUs and GPUs and consider them as Processing Units (PU) instead. While PUs handle tasks sequentially, multiple PUs can simultaneously process fragments and kernels from different queries. To efficiently schedule fragments of different queries in parallel, we must update our cost model and include new parameters like the current device's utilization, the number of cached fragments on a device, or the device's memory utilization. We plan to integrate inter-query parallelism into our prototypical implementation in HDK.

Heterogeneity beyond aggregations. While this work focuses on fragment-based heterogeneous aggregations that require a special merge phase of the output results, other operators like joins could be implemented similarly. For example, one strategy to implement efficient heterogeneous joins is based on hash partitioning the input data first [13]. The resulting hash-partitioned tables can be split into fragments such that a fragment does not cross a partition. A hash partition that consists of possibly multiple fragments, can then be either scheduled as part of a multi-fragment kernel (cf. Section 4) on GPU or processed regularly by the CPU. The primary advantage of using hash partitioning is to eliminate the costly task of merging join results from different devices. Although our cost model can estimate the build and probe phases effectively, it does not yet account for the benefits of hash partitioning and would need to be extended.

4 IMPLEMENTATION IN HDK

A recent survey has noticed that most modern heterogeneous query processors are CPU-centric [46]. In contrast, this work shows how a GPU-centric database system can be adapted for the device-parallel pipeline execution using morsel-driven parallelism. More specifically, we build upon Heterogeneous Data Kernels (HDK), a fast execution library for data analytics tasks [3]. HDK's architecture is inspired by OmnisciDB [54], an LLVM-compiling database system supporting CPU and GPU. As most other heterogeneous systems described in Section 2, OmnisciDB executes pipelines exclusively on CPU *or* GPU. This section first provides an overview of HDK's architecture and then explains how we integrated our approach to enable intra-pipeline device parallelism for aggregation queries.

Storage. HDK's in-memory representation is based on Apache Arrow [1]. Columns are stored as *ChunkedArrays*, which support fast, multi-threaded construction. Figure 2 conceptually visualizes the storage layout of a table with two columns in HDK. HDK's storage layer does not guarantee that the data within a fragment will be stored linearly, which may lead to additional materialization overhead. For example, column 1's physical chunks are aligned with the logical fragments, allowing direct memory access by simply passing a pointer to PUs. In contrast, column 2's physical chunks have a different alignment, requiring their prior materialization. An alternative would be to select a fragment size that matches the underlying Arrow's physical chunk size. However, this could also negatively impact pipeline parallelization. To optimize device parallel pipeline execution, it is crucial to identify an appropriate fragment size.

Fragment size. As discussed before, CPUs and GPUs work best with different fragment sizes. Task-parallel CPUs perform better with a larger number of smaller fragments, facilitating a more efficient workload distribution across cores. In contrast, data-parallel GPUs work better with fewer but larger fragments.

We have explored how the number of fragments affects performance by running Q2 on Machine 1 (cf. Section 5) with a uniform distribution over 200 million rows and a varying number of result groups. The results are presented in Figure 4. Please note that the GPU launches one kernel with multiple fragments to avoid the overhead of scheduling multiple kernels in this experiment. The number of result groups at the top of every plot corresponds to the number of aggregation groups. We assume that the corresponding fragment proportions are cached on the GPU. In HDK, CPU performance deteriorates when its parallelism is underutilized, as seen with fewer fragments. Additionally, we observe that the combination of many fragments and many result groups also diminishes CPU performance (cf. Section 5). In contrast, although the GPU's performance degrades with more fragments, its overall performance across possible fragment counts is more stable than the CPU. Scheduling a kernel with a single fragment for GPUs causes kernel launch overhead. Therefore, instead of processing one fragment per kernel, we

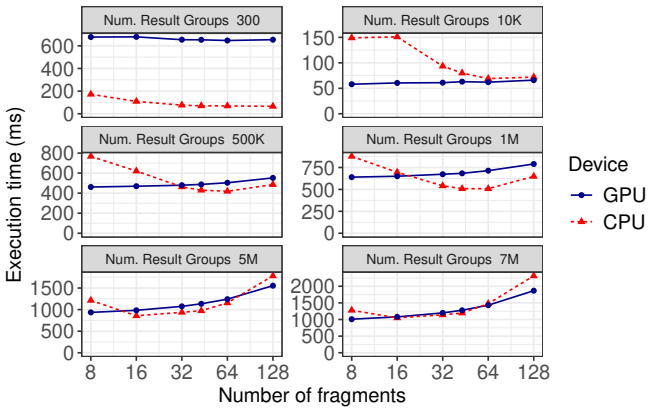


Figure 4: The query execution time on the CPU is more sensitive to changes in fragment count compared to the GPU. Setup: Machine 1, Query 2, 200M rows.

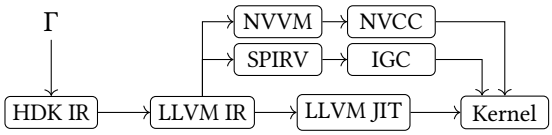


Figure 5: High-level overview of compilation in HDK.

use the *multifragment mode* that runs one kernel on multiple fragments, e.g., we loop over the fragments within a launched kernel. This mode amortizes such overheads and retrieves only one result from the GPU once the last fragment has been processed. Based on the findings of our experiments, we heuristically set the fragment size to $2 \times (\#rows/\#cpu\_physical\_cores)$. This approach generates sufficient fragments to fully utilize the CPU while ensuring that additional fragments are available for processing by the GPU.

Compilation. Figure 5 shows an overview of HDK's query compilation approach. HDK implements a custom intermediate representation (IR) to express relational algebra. In the next step, HDK's IR is lowered to LLVM IR and device-specific representations (e.g., NVVM for Nvidia or SPIRV for Intel GPUs). Last, specialized compilers like NVCC [6] or IGC [5] produce executable binary files. The CPU compilation follows the normal LLVM JIT compilation path.

Scheduling. The cost model described in Section 3 is implemented as an external component using SYCL [9], which is a high-level language that allows lowering patterns into many execution targets. The system's compiler generates code for different device *types*. The pattern implementation in SYCL should approximate the code produced by the system's compiler, hence the pattern implementation is meaningful for the device types supported by the system's compiler. For example, we might express a pattern of scan and selection in SYCL that would be compiled into a device-specific binary, which can then be calibrated on a device under various parameters (e.g., input size). Hence, we approximate HDK's compiler in a portable and transparent way for a set of query graph patterns. The calibrated estimates are used for a linear regression fit to find an optimal time estimate by iteratively changing the input size (i.e., fragment count) of a pattern for a device. Once the fragment distribution is known, we place it into the pipeline's execution policy (cf. Section 3).

Execution. After constructing all kernels, we launch them in asynchronous threads. CPU kernels are executed using LLVM JIT, whereas GPU kernels are launched using drivers such as CUDA [2] or LevelZero[7]. For device-parallel execution of aggregations, we must wait for all devices to complete their tasks before merging their results in a post-processing step.

Impact of heterogeneity on infrastructure. OmnisciDB is a GPU-centric database that assumes large fragments. Our approach, however, considers multi-core CPUs as equal executors to GPUs and expects tables to produce significantly more fragments. For instance, in a system with 512 physical CPU cores and a GPU accelerator, we would need > 512 fragments to assign at least one fragment to every PU. However, experimenting with larger fragment numbers showed significant slowdowns (cf. Figure 6): despite the constant number of allocations for each column, the infrastructure was overwhelmed after processing a few columns as the

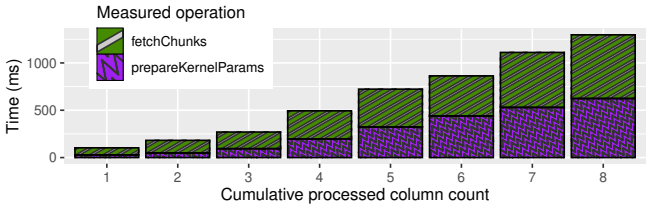


Figure 6: Table with 80M rows divided into 2000 fragments, the buffer manager faced increasing difficulties in managing allocations.

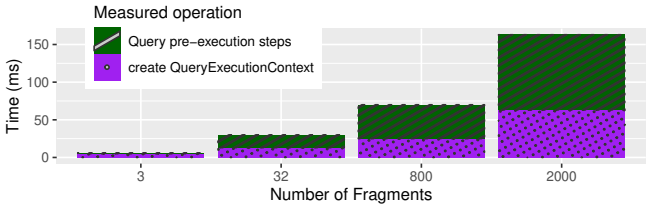


Figure 7: The original system copied each fragment's metadata, which significantly slowed down the system even before query execution began.

buffer managers in OmniSciDB were not designed for this use case. The search for a free spot took too long. We, thus, implemented a simple index structure for free buffers, resulting in constant column processing time. The original system assumed few fragments and thus experienced little overhead from copies of the fragments' metadata. We now need more fragments and the copies may become a considerable overhead (cf. Figure 7) which we solve by sharing fragments' metadata.

Discussion. HDK has room for improvement on its path towards more efficient intra-pipeline parallelism. In the future, we plan to adapt fragments to directly represent physical memory chunks, alleviating the need for redundant materializations. This would, however, put more restrictions on the storage layer. One of the more problematic points is how we approximate the HDK's compiler performance. As we do the calibration outside of HDK, we fail to capture all steps of a pattern processing that are non-negligible (cf. Section 5). Furthermore, the calibration process is costly due to the problem's dimensionality. The combination of large engineering effort for pattern implementation with the costly calibration process restrained us from further development, and in the future, we may also resort to adaptive scheduling.

5 EVALUATION

After discussing the key system infrastructure changes needed to implement heterogeneous execution, we continue with our system's evaluation. We answer the following three research questions:

RQ1 Which pipeline parameters allow heterogeneous acceleration?

RQ2 Can the morsel-driven execution model efficiently support device-parallelism?

RQ3 How does a device-parallel system scale with the input size?

Queries. Listing 1 shows the two aggregation queries that we use to evaluate our approach's performance in HDK considering

```
-- The table layout
CREATE TABLE tab (
    group_<nr>     INT64,
    rand_data_0     DOUBLE,
    rand_data_1     DOUBLE
);
-- Query 1: 1 output column aggregation
SELECT group_<nr>, AVG(rand_data_0)
FROM tab
GROUP BY group_<nr>;
-- Query 2: 2 output columns aggregation
SELECT group_<nr>, AVG(rand_data_0), MAX(rand_data_1)
FROM tab
GROUP BY group_<nr>;
```

Listing 1: The table layout and two aggregation queries. group_<nr> represents a column with <nr> groups.

different parameters, such as the input cardinality (number of rows), the output cardinality (number of result groups), and the number of aggregates (total result size). Grouping columns in the table are 64-bit integers, while columns used for aggregation calculations, such as averages and maximums, are doubles.

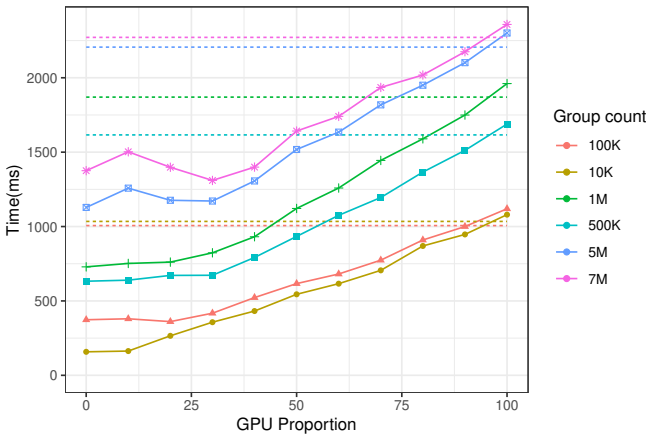
Testbed. Table 1 shows the configuration of the two machines we used for the experimental evaluation. They are equipped with GPUs from different vendors to show our approach's generalizability.

GPU Caching. In HDK, the device-relevant input portion can be accessible without data transfers when the relevant data is already cached in the accelerator's memory. On the other hand, a query result always needs to be sent back to the host to construct the final result. All experiments show the end-to-end query performance with the cached data and include the result transfer times. Prior fragment materializations are not required as we store columns in contiguous memory for testing purposes.

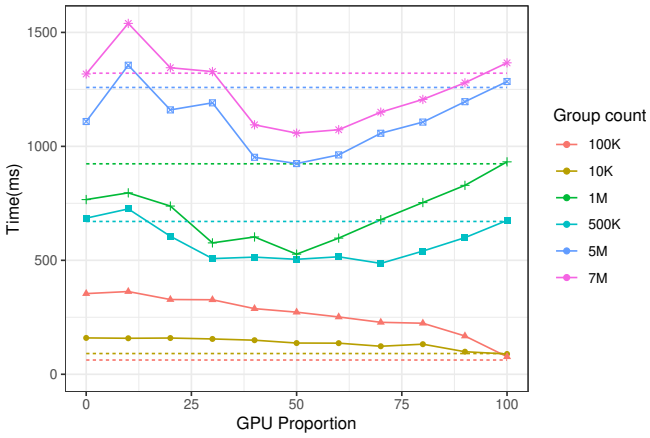
Expecting data reuse and caching is reasonable within the explorative setting of analytical queries, but we also conduct experiments for cases when the entire proportion needs to be sent to the GPU in Figure 8a where the baseline (dashed lines) refers to the default setting of Omnisci (i.e., no device results merge phase and very large fragments). Figure 8a indicates that the input proportion to transfer is too large to be compensated by the CPU execution. In the worst case, we match the GPU baseline. However, we expect this to be less of a problem with newer interconnects. Figure 8b shows the same benchmark with cached data, where large outputs might still be bottlenecks. We are, therefore, interested in aggregation queries over large datasets with small outputs.

As shown in Table 1, Machine 2 has an Intel GPU interconnected via PCI Express 5. We compare the time difference between cold and hot runs on the PCIe 3.0 machine (cf. Figure 9a) and PCIe 5.0 machine (cf.Figure 9b). The results demonstrate that the better interconnect combined with a more sophisticated transfer logic [8] significantly mitigates the drawbacks of 'cold' data on the query execution time.

Low cardinality tables (RQ1). Processing small tables typically results in fast execution, which reduces the benefits of parallel computation. In many cases, we can also expect low cardinality results for aggregation queries, so the benefit from hiding transfer costs is



(a) "Cold" (i.e., uncached) data.



(b) "Hot" (i.e., cached) data.

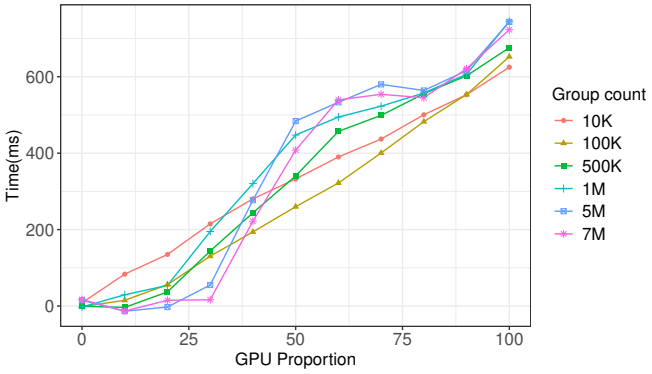
Figure 8: Example of Q2 on 300M rows and 32 Fragments running on Machine 1.

Table 1: Testbed setup.

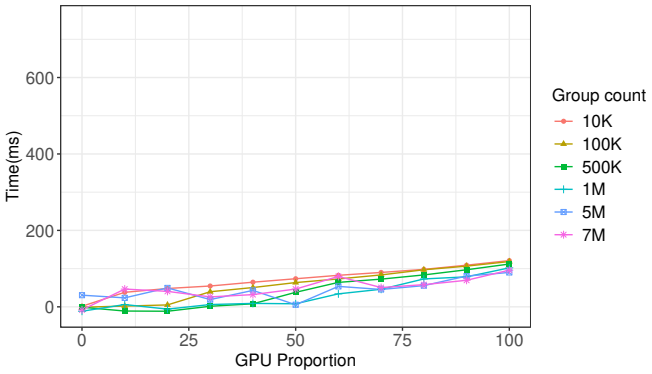
	Machine 1	Machine 2
CPU	Intel Xeon 6226R (64 cores)	2x Intel Xeon 8468 (32 cores)
GPU	NVIDIA Quadro RTX 6000	Intel GPU Max Series 1100
RAM	96 GB DDR4-2933 MHz	1 TB DDR5-4800 MHz
PCI-e	x16 PCI Express 3	x16 PCI Express 5

also minimal. Indeed, for small tables, the pipelines execute so fast on either device that execution-independent system components start to play a significant role. Consider the compilation: additional steps for GPU (cf. Figure 5) mean that the compilation is typically slightly longer than for CPUs. Even with parallel compilation, this disparity introduces an additional execution-invariant overhead. Other examples include initializing CPU buffers and merging kernels' results; each can take longer than the execution or result transfer. Therefore, to pay off, heterogeneous aggregations in HDK require sufficiently large tables, typically consisting of millions of rows.

Data properties (RQ1). We also observe that the optimal fragment size depends on the analyzed data, which can significantly impact



(a) Machine 1 (PCIe 3.0).



(b) Machine 2 (PCIe 5.0).

Figure 9: Time difference Δ(t_cold_run − t_hot_run) for Q1 on 300M rows and 64 Fragments.

the query execution time. Figure 4 shows that the result size (i.e., the number of groups) also impacts the execution time of CPU- and GPU-based execution. For example, we expect that CPU-only execution benefits from an increasing number of fragments due to more concurrent execution. Interestingly, Figure 4 demonstrates that this expectation does not always hold. For instance, the query returning 7M result groups executed faster on the CPU using fewer fragments, which needs a more thorough investigation.

Some logical steps of query processing cannot be avoided. For example, aggregation results can overlap and must be merged after the local aggregations have finished. In Figure 10, we investigate the merging and execution time of an aggregation query when executed on a CPU for different numbers of fragments. We can observe a linear dependence between the number of fragments and the merging time. Notably, the CPU execution time decreases when the fragment count is reduced to 16, which is fewer than the 64 available CPU cores.

Initially, we expected the fragment count to at least equal the number of CPU cores to leverage parallelism fully. However, we discovered that the primary factor influencing the CPU execution time is the initialization of the hash tables per kernel. The size of each hash table is based on the estimated number of distinct elements *in a column*. As a result, hash tables are often larger than necessary, and multiple threads initializing their large buffers concurrently can saturate the memory bandwidth. Therefore, in this case, having

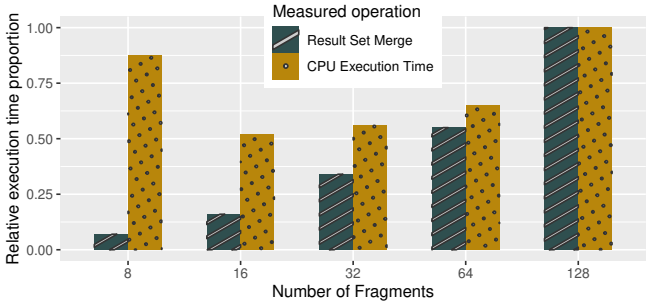
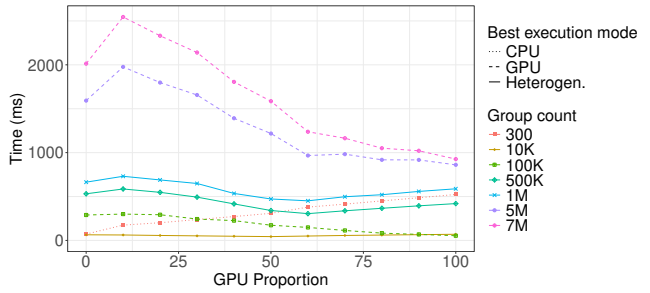
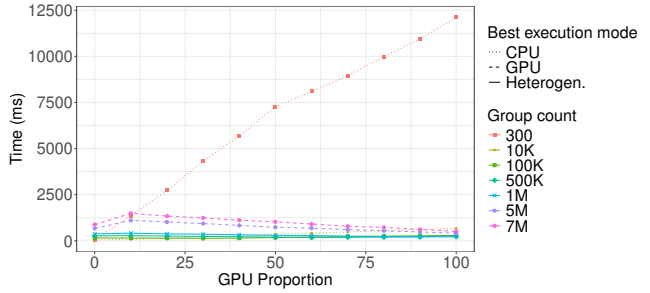


Figure 10: CPU aggregation on 200M rows and 7M result groups on Machine 1.



(a) Machine 1, Query 2, 128 fragments.



(b) Machine 2, Query 1, 64 fragments.

Figure 11: Example queries 1 and 2 executed on machines 1 and 2 over 300M rows for different result group numbers. The x-axis shows the amount of data the GPU processed.

fewer threads can improve CPU execution time despite using less parallelism. This demonstrates that more fragments do not always improve CPU performance but can lead to longer merge times.

The CPU processing in the original OmnisciDB struggles with smaller fragments. To combat merging issues, we may reside in a hash partitioning scheme [13]; early attempts showed little improvement in the end-to-end time of various workloads.

Aggregation Analysis (RQ2). Figure 11 investigates the impact of the data proportion executed by the GPU for different result group counts on the query execution time. Figure 11a shows the results for running Q2 on Machine 1 for 300M rows and 128 fragments, and Figure 11b shows the results for Q1 running on Machine 2 for 300M rows and 64 fragments. In both experiments, each line represents the execution time for a different number of result groups after the aggregation. We exclude compilation times from the reported measurements, as the query remains unchanged. The results

highlight how heterogeneous aggregations with varying properties favor different execution settings. The line styles indicate the optimal execution mode, e.g., dotted lines indicate that CPU-only execution performed best, and dashed lines indicate that GPU-only execution performed best. Solid lines show the cases where heterogeneous execution yielded the lowest execution time. The presence of heterogeneous mode for both machines indicates heterogeneous acceleration, as expected per our analysis earlier.

The observed device preferences at specific group sizes not only depend on the merging behavior but also on the device's properties and the operator implementation. When we have a low number of unique elements (e.g., 300) in a big table, many GPU threads will try to access the same memory, and the synchronization overhead makes the CPU a better choice. On the other hand, many distinct elements may cause issues during the CPU-resident result merging.

Balancing opportunities (RQ3). Intra-pipeline device parallelism offers another edge in balancing capabilities. For example, we may balance out the negative impact of the increased table size by varying the device proportions. Figure 12 shows how heterogeneous pipeline execution helps to amortize the execution time better than device-exclusive runs when the dataset doubles in size. Notice the hotspots in the upper and lower rows in Figure 12. They indicate that doubling the dataset size imposes a larger performance penalty when executing device-exclusively. In contrast, the area between the upper and lower rows shows milder performance reductions compared to device-exclusive execution

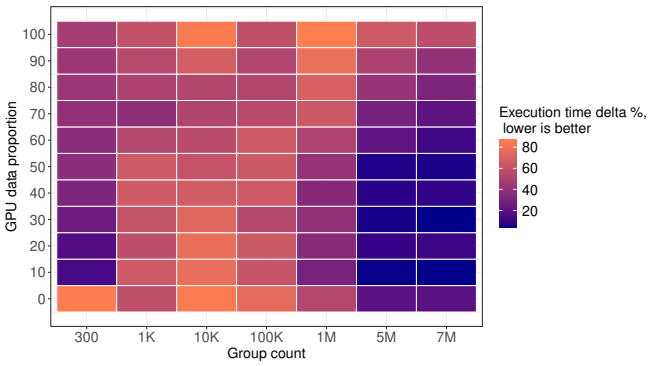


Figure 12: Time difference (in %) of aggregations after doubling the data size from 100M rows to 200M rows. Machine 1, Query 2, 64 fragments.

Discussion. We conclude that the determining factors for heterogeneity benefit in aggregations are: the input table cardinality, the number of groups, the number of results for CPU, and the output size for GPU. In such a system, when we improve the execution performance of one device, the "heterogeneity valley" shifts to a different set of query or data parameters.

6 CONCLUSIONS

This paper presented a new approach to enable device-parallel aggregations in heterogeneous database systems with fragment-based parallelism. Our system-level integration into HDK in Section 4 has shown that our ideas can be transferred to a real system. Furthermore, a comparison between heterogeneous and device-exclusive execution showed query performance improvements by up to 1.5x compared to the fastest executor (CPU or GPU) as well as the amortization capacity for increasing input cardinalities.

Outlook. In the context of the presented system, the advent of multi-GPU deployments alongside advanced interconnects promises significant scalability improvements. This is primarily due to the utilization of DMA for CPU-GPU transfers and the system infrastructure's support for device-to-device fragment passing. Furthermore, it would be interesting to investigate the impact of Heterogeneous Memory Management (HMM) support for CUDA devices. Comparing this to explicit fragment/morsel caching in systems equipped with modern interconnects, such as PCI Express 5, could provide valuable insights into task elasticity, which is a fundamental aspect of heterogeneous execution.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2024. Apache Arrow. https://arrow.apache.org/ Accessed on 16.02.2024.
[2] 2024. CUDA Driver API. https://docs.nvidia.com/cuda/cuda-driver-api/ Accessed on 07.02.2024.
[3] 2024. HDK. https://github.com/intel-ai/hdk Accessed on 16.02.2024.
[4] 2024. Heavy.AI. https://www.heavy.ai/ Accessed on 07.02.2024.
[5] 2024. Intel Graphics Compiler. https://github.com/intel/intel-graphics-compiler Accessed on 08.02.2024.
[6] 2024. NVIDIA CUDA Compiler. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html Accessed on 08.02.2024.
[7] 2024. OneAPI Level Zero Specification. https://spec.oneapi.io/level-zero/latest/index.html Accessed on 07.02.2024.
[8] 2024. Pull request with a transfer scheme improvement proposal. https://github.com/intel-ai/hdk/pull/711 Accessed on 07.05.2024.
[9] 2024. SYCL. https://www.khronos.org/sycl/ Accessed on 07.02.2024.
[10] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 [cs.DC]
[11] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. 2020. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 26–38. https://doi.org/10.1109/PMBS51919.2020.00009
[12] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
[13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839
[14] Sebastian Breß. 2013. Why it is time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.* 6, 12 (2013), 1398–1403. https://doi.org/10.14778/2536274.2536325
[15] 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
[16] 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, 13 (2014), 1609–1612. https://doi.org/10.14778/2733004.2733042

[17] Sebastian Breß, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2013. An Operator-Stream-Based Scheduling Engine for Effective GPU Coprocessing. In *Advances in Databases and Information Systems*, Barbara Catania, Giovanna Guerrini, and Jaroslav Pokorný (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–301.

[18] S. Breß, Igor Geist, E. Schallehn, M. Mory, and Gunter Saake. 2012. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. *Control and Cybernetics* 41 (01 2012), 715–742.

[19] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. *GPU-Accelerated Database Systems: Survey and Open Challenges*. Vol. 8920. 1–35. https://doi.org/10.1007/978-3-662-45761-0_1

[20] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (nov 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[21] Periklis Chrysogelos, Manos Karpathiotakis, R. Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proceedings of the VLDB Endowment* 12 (01 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[22] Hawon Chu, Seounghyun Kim, Joo-Young Lee, and Young-Kyoon Suh. 2020. Empirical evaluation across multiple GPU-accelerated DBMSes. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 3 pages. https://doi.org/10.1145/3399666.3399907

[23] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.* 12, 12 (2019), 2218–2229.

[24] 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 *Proceedings of the 13th International Workshop on Data Management on New Hardware* (Chicago, Illinois) *(DAMON '17)*. Association for Computing Machinery, New York, NY, USA, Article 3, 6 pages. https://doi.org/10.1145/3076113.3076119

[25] Wilfred Gomes, Altug Koker, Pat Stover, Doug Ingerly, Scott Siers, Srikrishnan Venkataraman, Chris Pelto, Tejas Shah, Amreesh Rao, Frank O'Mahony, Eric Karl, Lance Cheney, Iqbal Rajwani, Hemant Jain, Ryan Cortez, Arun Chandrasekhar, Basavaraj Kanthi, and Raja Koduri. 2022. Ponte Vecchio: A Multi-Tile 3D Stacked Processor for Exascale Computing. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 42–44. https://doi.org/10.1109/ISSCC42614.2022.9731673

[26] Goetz Graefe. 1990. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, USA) *(SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 102–111. https://doi.org/10.1145/93597.98720

[27] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. *ISPASS 2011 - IEEE International Symposium on Performance Analysis of Systems and Software*, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[28] 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

[29] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (aug 2013), 889–900. https://doi.org/10.14778/2536206.2536216

[30] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proceedings of the VLDB Endowment* 6 (08 2013). https://doi.org/10.14778/2536360.2536370

[31] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[32] Homin Kang, Jaehong Lee, and Duksu Kim. 2021. HI-FFT: Heterogeneous Parallel In-place Algorithm for Large-scale 2D-FFT. *IEEE Access* PP (08 2021), 1–1. https://doi.org/10.1109/ACCESS.2021.3108404

[33] 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

[34] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. 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, 555–569. https://doi.org/10.1145/2882903.2882906

[35] Petr Kurapov and Areg Melik-Adamyan. 2023. Analytical Queries: A Comprehensive Survey. arXiv:2311.15730 [cs.DB]

[36] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[37] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[38] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9, 14 (oct 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[39] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[40] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (jul 2015), 35 pages. https://doi.org/10.1145/2788396

[41] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. 2014. Heterogeneity-Conscious Parallel Query Execution: Getting a Better Mileage While Driving Faster!. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (Snowbird, Utah) *(DaMoN '14)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/2619228.2619230

[42] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. 2008. An efficient, model-based CPU-GPU heterogeneous FFT library. (2008), 1–10. https://doi.org/10.1109/IPDPS.2008.4536163

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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. arXiv:1912.01703 [cs.LG]

[44] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-Based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1935–1950. https://doi.org/10.1145/2882903.2915224

[45] Chrysogelos Periklis, 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

[46] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (jan 2022), 38 pages. https://doi.org/10.1145/3485126

[47] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[48] Marc Tallada and Enric Morancho. 2023. Heterogeneous programming using OpenMP and CUDA/HIP for hybrid CPU-GPU scientific applications. *The International Journal of High Performance Computing Applications* 37 (08 2023). https://doi.org/10.1177/10943420231188079

[49] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD Conference*. ACM, 1879–1891.

[50] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[51] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[52] Bowen Zhang, Yanyan Shen, Yanmin Zhu, and Jiadi Yu. 2018. A GPU-Accelerated Framework for Processing Trajectory Queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1037–1048. https://doi.org/10.1109/ICDE.2018.00097

[53] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 633–647. https://www.usenix.org/conference/atc20/presentation/zhang-feng

[54] Yansong Zhang, Yu Zhang, Jiaheng Lu, Shan Wang, Zhuan Liu, and Ruichen Han. 2020. One size does not fit all: accelerating OLAP workloads with GPUs. *Distributed and Parallel Databases* 38 (12 2020). https://doi.org/10.1007/s10619-020-07304-z