GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP

NILS BOESCHEN, Technical University of Darmstadt & hessian.AI, Germany
TOBIAS ZIEGLER, Technical University of Darmstadt, Germany
CARSTEN BINNIG, Technical University of Darmstadt & DFKI Darmstadt, Germany

In this paper, we suggest a novel GPU-in-data-path architecture that leverages a GPU to accelerate the I/O path and thus can achieve almost in-memory bandwidth using SSDs. In this architecture, the main idea is to stream data in heavy-weight compressed blocks from SSDs directly into the GPU and decompress it on-the-fly as part of the table scan to inflate data before processing it by downstream query operators. Furthermore, we employ novel GPU-optimized pruning techniques that help us further inflate the perceived read bandwidth. In our evaluation, we show that the GPU-in-data-path architecture can achieve an effective bandwidth of up to 100 GiB/s, surpassing existing in-memory systems' capabilities.

CCS Concepts: • **Information systems → DBMS engine architectures**; *Online analytical processing engines.*

Additional Key Words and Phrases: DBMS GPU-Acceleration; OLAP; SSD Storage

ACM Reference Format:
Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 237 (December 2024), 26 pages. https://doi.org/10.1145/3698812

1 Introduction

Recent trend towards SSD-based Databases. Over the past decade, in-memory database systems have become prevalent in research and academia, typically outperforming traditional disk-based systems. This trend was primarily possible by falling DRAM prices from 2000 to 2010 [9]. However, recently, memory prices have plateaued [9], making in-memory databases economically unattractive for large data sets. Therefore, modern analytical systems increasingly favor SSD-based storage, as the cost per terabyte has decreased 30-fold over the past decade. The price reduction in flash storage has driven a transition towards high-performance SSD-backed database systems [12, 15]. However, while SSDs are very cost-effective, they unfortunately cannot match the performance of in-memory systems.

Limited bandwidth of SSD-based Databases. As shown in Figure 1 (green and violet bar), there is still a "performance gap" between in-memory systems and SSD-based systems. In fact, when running SSB query 1.1 on recent hardware, in-memory systems still achieve a bandwidth 3.5 times higher than a DBMS on modern SSDs. One way to bridge this performance gap could be to compress data on SSD and decompress it once a portion of data is loaded into memory. Loading compressed data will let a system load more disk-resident data in the same amount of time, thereby increasing

Authors' Contact Information: Nils Boeschen, nils.boeschen@cs.tu-darmstadt.de, Technical University of Darmstadt & hessian.AI, Germany; Tobias Ziegler, tobias.ziegler@cs.tu-darmstadt.de, Technical University of Darmstadt, Germany; Carsten Binnig, carsten.binnig@cs.tu-darmstadt.de, Technical University of Darmstadt & DFKI Darmstadt, Germany.

ACM 2836-6573/2024/12-ART237
https://doi.org/10.1145/3698812

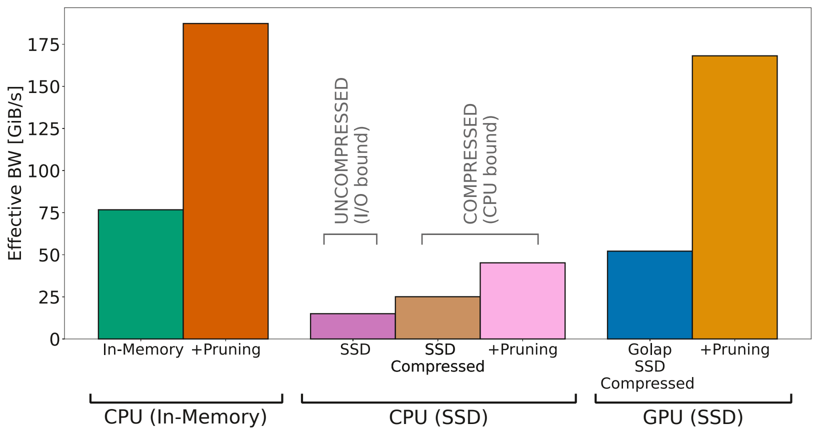


Fig. 1. The effective bandwidth (calculated as processed uncompressed bytes divided by runtime) of SSB query 1.1 is shown here. We compare a hand-coded CPU-optimized implementation for in-memory and SSD-based execution with and without compression and our GPU-accelerated, SSD-based system, GOLAP. Both the CPU and GPU systems using compression achieve a compression ratio of approximately 2.8. Additionally, we illustrate the impact of pruning on these configurations. Overall, the GPU system improves on SSD-based CPU performance and is competitive with the CPU In-Memory architectures.

the perceived read bandwidth. This, of course, necessitates that the time it takes to decompress data in memory does not negate the advantages of loading compressed data. In particular attractive to such an approach are heavy-weight compression schemes, which often achieve higher compression ratios than light-weight compression schemes. However, when integrating such an approach into a CPU-based DBMS, unfortunately, it fails to significantly improve query performance, as evidenced by Figure 1 (brown bar).

GPUs as enablers for a new Architecture? In a CPU-based DBMS, the limitation for heavy-weight compression is the CPU's limited computational power to decompress data at line rate. Compared to CPUs, GPUs have orders of magnitude more computational power, and we thus argue that they are a much more interesting alternative to enabling analytical query processing on heavy-weight-compressed data. Moreover, with recent advances such as GPU Direct Storage[1], data can be directly read into the GPU memory from the SSD without first loading it into CPU memory. Overall, we argue that GPUs thus are a highly attractive platform for building an SSD-based analytical database system which can provide much higher bandwidth than a CPU-based DBMS by storing data in highly compressed form on SSD and uncompressing it only on-the-fly for query processing after loading data into GPU memory.

A novel GPU-in-data-path Architecture. Based on these observations, we propose a novel GPU-in-data-path architecture for analytical query processing in this paper. In this architecture, the main idea is to stream data in heavy-weight compressed blocks from SSDs directly into the GPU and decompress data on-the-fly as part of the table scan, which inflates data for query processing by downstream operators. In our scheme, it is essential that the GPU directly consumes the decompressed blocks and executes query processing operators that overlap with decompression. As a main contribution, this paper presents a first prototype for the GPU-in-data-path architecture which implements these ideas for analytical query processing. As shown in Figure 1 (blue bar), GOLAP can process data efficiently and not only achieve much higher bandwidths compared to SSD-based DBMS using CPUs but is, in fact, much closer to the performance of in-memory databases.

[1]We refer to NVIDIA hardware and terminology in this work.

Further opportunities improve the I/O efficiency. This paper demonstrates that the GPU-in-data-path architecture offers more than just on-the-fly decompression. It capitalizes on the computational resources available on GPUs to improve analytical query processing and I/O efficiency when accessing data from flash. In particular, as a second direction in this paper, we present how more computationally intensive pruning schemes can be integrated into a GPU-in-data-path architecture. The core idea is that, in contrast to existing schemes for block-based pruning, such as storing min-max values per block, we can store more expensive summaries per block (e.g., full histograms) and use them for pruning. In fact, when including such pruning schemes, GOLAP (yellow bar) can achieve an effective bandwidth of over 100 GiB/s. Interesting is that compared to pruning in both CPU variants (orange and pink bar), pruning in GOLAP provides a higher benefit even though data is sorted in the same way due to our novel GPU-based pruning schemes.

Contributions. This paper introduces GOLAP, our prototype of a GPU-in-data-path architecture for analytical query processing. The core of our contributions are: (1)The first contribution is a novel GPU-based table scan operator that utilizes the GPU's high computational power to decompress data on-the-fly from SSDs. This approach loads data directly from SSDs into the GPU using GPU Direct Storage and decompresses data as part of the scan. (2) As the second contribution, we show how the GPU-based table scan can be efficiently integrated with further downstream query processing operators such as joins and aggregations. Important for achieving a high read-bandwidth is that we overlap direct I/O using GPU Direct Storage, decompression, as well as the execution of query processing operators. For this, we present a simple yet efficient scheme that uses multiple decompression buffers. (3) As a third contribution, to further increase the effective bandwidth, we explore GPU-optimized data pruning techniques, enabling selective data block processing. This step happens before even loading the data into GPU memory and is thus an additional opportunistic optimization that yields substantial benefits for many workloads. Importantly, the pruning scheme does not adversely affect the query execution speed if pruning potentials are low. (4) Finally, while our GPU-in-data-path architecture allows us to execute many queries completely on the GPU, GPUs still have limitations regarding the amount of memory they provide and thus restrict which state (e.g., hash tables of joins) can be kept on the GPU. As such, GOLAP in addition implements GPU-CPU co-execution strategies. In particular, we present a novel hybrid approach that allows the system to handle very large data sets in a robust manner.

Outline. The remainder of this paper is structured as follows: In the next section, we will provide an overview of the architecture of a GPU-accelerated SSD-based OLAP system. Section 3 then discusses the details of how to implement an efficient table scan over compressed data in a GPU that can read data directly from SSD. Section 4 afterwards then describes GPU-optimized pruning techniques and how to incorporate them into our system. GPU-CPU-based co-execution for queries that exceed the memory capacities of GPUs is detailed in Section 5. The experimental evaluation of our system is given in Section 6. Section 7 discusses challenges and opportunities for optimizing query plans for GPU-in-data-path systems. We conclude with a review of related work and final remarks in Sections 8 and 9, respectively.

2 System Design Overview

This section outlines the core concepts of GOLAP, which is based on the GPU-in-data-path architecture. The primary components, illustrated in Figure 2, include all contributions mentioned before, such as the opportunistic pruning scheme and the novel GPU-based table scan on heavy-weight compressed data as well as our GPU-CPU-based co-execution for queries which exceed the memory capacities of GPUs. The basic idea is that by applying all these concepts, the effective bandwidth of processing data on SSDs is inflated to up to 100GiB/s (from 20GiB/s on our SSD devices), thus significantly improving OLAP query processing speeds on SSDs.

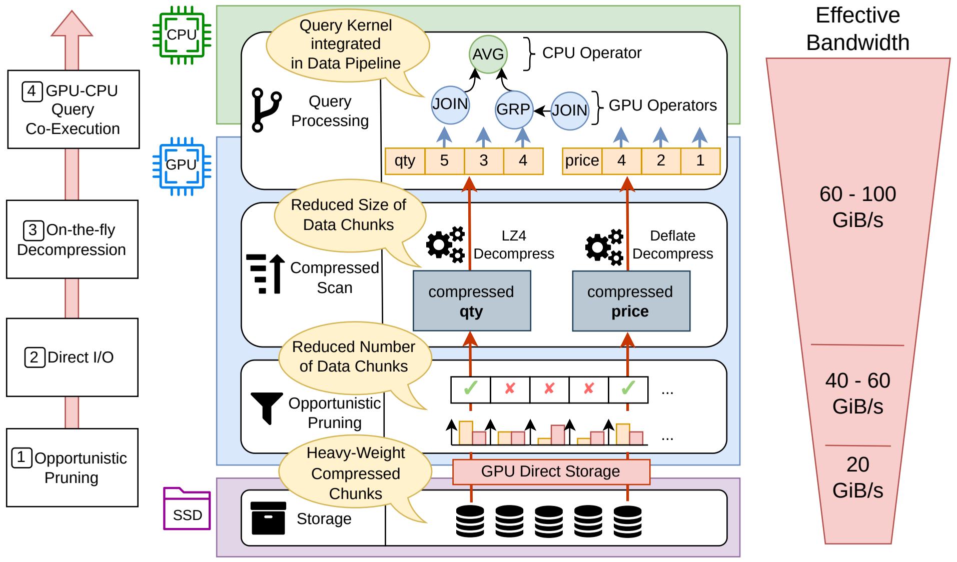


Fig. 2. The GPU-in-data-path architecture, comprised of an opportunistic pruning step, a compressed scan that uses direct storage I/O to read data chunks directly into GPU memory, and GPU-CPU co-execution of query plans. Each step increases the effective processing bandwidth past the flash read bandwidth.

2.1 The Core Building Blocks

In the following, we explain the key concepts mentioned above in the order of how they are applied during query processing to provide an intuition of how GOLAP - our prototype of a GPU-in-data-path architecture - works. The subsequent sections will then dive deeper into each of the concepts.

Compressed Storage and Opportunistic Pruning. In GOLAP, data is maintained on SSD storage using a heavily compressed columnar chunk format. This approach utilizes established heavy-weight compression algorithms specifically optimized for GPU processing. However, choosing the best compression setting remains a challenge. Our system implements an adaptive data compression sampling strategy. This method assesses various compression parameters, including the compression scheme and chunk size, to improve the efficiency of compressed data scanning and query execution. Moreover, before reading compressed blocks of data for processing into GPU memory, GOLAP evaluates metadata for each chunk to see if it needs to be loaded based on the query's predicates. This ensures that only relevant data chunks are loaded from SSD into GPU memory, further increasing the effective bandwidth. Even when faced with less selective predicates and thus limited pruning potential, GOLAP's robust design allows us to opportunistically apply this step without incurring a performance hit since the GPU-accelerated pruning has low processing overhead. However, when pruning can be applied, as we will demonstrate, GOLAP has the potential even to outperform in-memory database systems.

Compressed Scan: A key building block of GOLAP is the table scan on compressed data. A naïve implementation of I/O would be to copy a compressed chunk of data from SSD via CPU to GPU. This would not be ideal since it adds latency for individual chunk accesses and increases memory overhead for pinned buffers on CPU. In this paper, we leverage GPU Direct Storage, a technology that allows us to avoid the additional overhead since data can be copied from SSD directly to GPU memory and be decompressed there. However, leveraging GPU Direct Storage in an efficient manner is not trivial since the I/O is still CPU-initiated, meaning that while the data itself is directly copied into the GPU, the control flow is handled by CPU threads. A major problem is that many concurrent I/Os are needed to saturate the I/O bandwidth. However, when done in a blocking way (i.e., load data and decompress before the next block is read), then a high number of concurrent I/O threads on the CPU are needed simply to trigger the I/O. In the paper, we present a scheme that overlaps I/O and decompression in an efficient way such that the number of CPU I/O threads needed for reaching high data processing bandwidths is minimized.

GPU-in-data-path optimized Query Execution: Once data is decompressed by our scan, GOLAP executes downstream query operators such as joins and aggregates within the GPU. This approach utilizes the GPU's capacity for massive parallelism across various query operations. To make this efficient, it is important to closely integrate query execution kernels with the data pipeline, ensuring sustained concurrent I/O and compute operations. GOLAP achieves this by co-designing the compressed scan and the query execution. To be more precise, query execution in GOLAP is streamlined with I/O, processing table segments on-the-fly once decompressed to minimize the memory footprint on the GPU. The rationale is to get the most use out of the GPU's superior compute power by doing as many operations as possible directly on device-resident data reducing the memory footprint on the GPU until results are finally moved to CPU memory.

However, for some operations, such as hash-based GPU joins, intermediate state (e.g., hash-tables) might exceed GPU memory. For such cases, we employ a novel GPU-CPU co-execution strategy which is tailored to the GPU-in-data-path architecture. The main idea is that we then still use the GPUs capacity for decompression and scanning through data and preparing downstream operators while parts of query processing operations that exceed GPU memory are executed on the CPU. This hybrid approach thus plays to the strengths of both types of processors and enables GOLAP to analyze large datasets that do not fit into the GPU memory.

2.2 Discussion

To conclude the system design overview, we summarize why we think that the the GPU-in-data-path architecture is attractive.

Robustness. Our first point centers on the robust nature of the GPU-in-data-path accelerator. It ensures processing capabilities akin to those of an SSD-based system utilizing a CPU, at the very least but typically surpasses CPU-based systems. A worst-case scenario for GOLAP arises when no query processing can be pushed to the GPU since the first operator exceeds the memory capacities of the GPU. In this case, only the scan can be executed, which necessitates transferring all (decompressed) data to the CPU for further query processing via the GPU-CPU interconnect, typically PCIe, which effectively limits the bandwidth to the PCIe bandwidth. In such worst-case scenarios, GOLAP is however only as slow as (and never slower than) a CPU-based SSD system without compression since it also needs to read the full uncompressed data via PCIe, which is the limiting factor then. However, for many real-world OLAP workloads, GOLAP will effectively accelerate query execution since analytical queries are typically selective (i.e., always some data can be pruned or filtered) and downstream operations such as joins can thus be executed on the GPU, which results in a significant acceleration for OLAP as we show in our evaluation.

Cost-efficiency. As an additional reason for why this architecture is highly attractive, we argue that it can be more cost-effective than an equivalent high-end in-memory CPU system. Costs for in-memory systems increase rapidly for very large databases in contrast to SSD-based systems, since DRAM has much higher costs per GiB than SSD storage. Moreover, the total size of data is still limited for in-memory systems. In contrast, in a GPU-in-data-path architecture that builds on SSDs, we have only the fixed cost of a GPU (independent of the data set size) along with the SSD cost which are substantially lower than memory cost. Furthermore, as we show in our evaluation, similarly high performance as in-memory systems can be achieved using economic flash storage and a GPU instead. As such, we argue that the additional fixed cost that is not affected by database

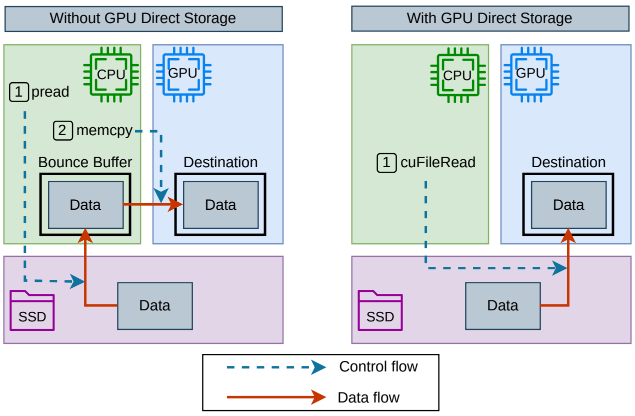


Fig. 3. Options for reading SSD-resident data into GPU memory. Without GPU Direct Storage (left), data is read into a CPU memory bounce buffer first and is then copied to GPU memory. With GPU Direct Storage (right), we alleviate the need for CPU memory bounce buffers, but we still require CPU-initiated data transfers.

size and the high performance makes the GPU-in-data-path architecture a highly attractive platform for data-intensive systems.

3 Compressed Scan on a GPU

The GPU-based table scan on compressed storage is the most important operator in GOLAP, since the SSD read bandwidth is usually the bottleneck for SSD-based database systems. In the following, we outline the cornerstones of an efficient GPU-accelerated table scan that uses direct transfers of heavy-weight compressed data chunks between SSD and GPU memory with decompression on the GPU to increase the effective data processing bandwidth.

3.1 Storage Format of Compressed Tables

Before we explain the scan approach, we first introduce the storage format used in GOLAP. GOLAP uses a columnar storage layout since this is superior for OLAP workloads and is more effectively compressible. In GOLAP, for storing a column on SSD, the column data is turned into chunks containing a fixed number of tuples. The number of tuples per chunk is a configurable parameter, but it is fixed across all chunks for the same column to allow the system to reuse fixed-size buffers for inflating data chunks during a column scan. The number of tuples per chunk and consequently the size of chunks, however, has implications on the effectiveness of data loading (too small requires many I/O requests), the compression ratio (prefers large chunks), as well as the ability to prune data (which prefers small chunks), which we discuss and evaluate in Section 6.4. At the end of this section, we present a procedure for selecting chunk sizes to optimize compression ratios given in a table. Next, we first explain how data can be read from SSDs into GPUs and decompressed efficiently.

3.2 Reading Data with GPU Direct Storage

As a first step, we now explain how to access compressed chunks on SSDs and load them into GPU memory. Until recently, accessing data on SSDs from the GPUs was only possible by a "detour" over CPU memory: For reading, data had to be first moved from SSD to CPU memory using a "bounce buffer" before it was then copied from there to the memory of the graphical processor (see Figure 3, left). These kinds of indirect data transfers via CPU memory, however, suffer from high latency, use

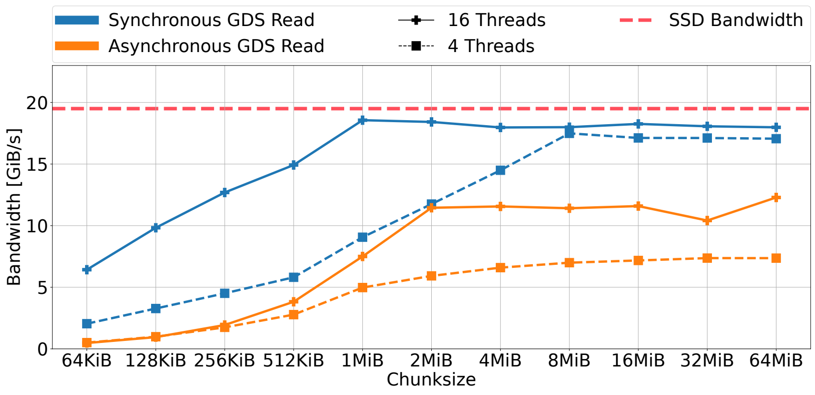


Fig. 4. SSD to GPU I/O bandwidth using synchronous (blocking) and asynchronous (in CUDA Streams) GPU Direct Storage. Asynchronous GDS in its current state and this setup can not fully exploit the full I/O bandwidth.

up pinned memory on the host, and put unnecessary strain on the bandwidth-limited GPU-CPU interconnect.

Unlike this data path, NVIDIA's GPU Direct Storage (GDS) technology [19] allows to transfer data directly between NVMe devices such as SSDs and GPU memory. To be more precise, the DMA controllers on SSDs and GPUs can be instructed to move data from and to SSD devices without involving CPU memory for data loads using memory bounce buffers (Figure 3, right). However, while GDS enables direct transfer of data from SSD to GPU memory, the I/O is still CPU controlled; i.e., the CPU is still necessary to control and initiate these direct data transfers from SSD to GPU using specialized I/O API calls such as `cuFileRead`. Consequently, this means that a GPU-in-data-path architecture still requires a CPU-side control flow capable of orchestrating data loads from SSDs to GPUs.

Moreover, at practical I/O sizes, SSDs require multiple concurrent I/Os to saturate the available I/O bandwidth. An important aspect of using GDS, concurrent I/Os can only be achieved using multiple CPU threads issuing synchronous (blocking) GDS reads or writes, or, more recently, by placing I/O calls to asynchronous GDS primitives into different CUDA Streams. Intuitively, the asynchronous variant seems favorable for reducing the number of CPU I/O threads to reach peak read bandwidth. We compare the bandwidth characteristics of both variants in Figure 4.

In this setup (and all others we had access to), the asynchronous variant, however, could not exploit the full available SSD bandwidth, even with a significant amount of threads and large I/O sizes. We attribute this to the fact that the asynchronous variant serializes parts of the necessary CPU work in the GDS driver, which is a known limitation of the underlying `cuLaunchHostFunc` API. In our system, we thus employ the synchronous GDS variant instead, which reliably reaches peak SSD read bandwidths at a reasonable number of threads and I/O sizes as we see in Figure 4.

3.3 On-the-fly Decompression

Compressing data allows the same amount of information to be transferred in a shorter time. Loading compressed data chunks from SSD and decompressing it on-the-fly therefore has the potential to increase the perceived I/O bandwidth at which data chunks can be loaded to GPU memory. However, an additional decompression step is necessary to materialize the uncompressed data. If the added overhead of the needed decompression does not outweigh the reduction in I/O time, this will not lead to an overall increase in the effective bandwidth at which uncompressed

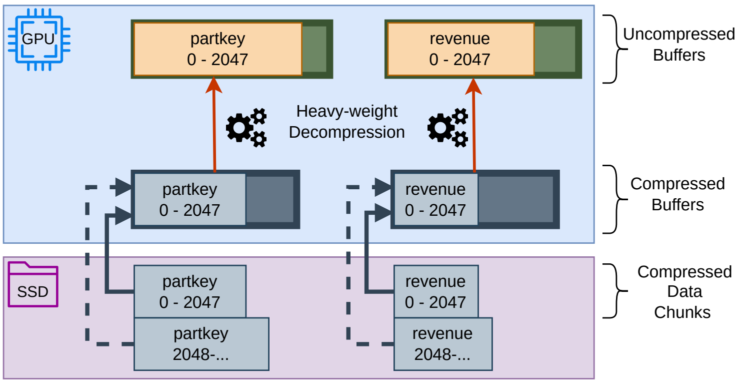


Fig. 5. The compressed table scan of two columns. Column chunks of multiple columns can be decompressed concurrently using multiple fixed-sized decompression buffers.

chunk data is materialized in memory. However, as we show next, when carefully integrating decompression with I/O, we can efficiently use the massive compute resources of GPUs and thus significantly inflate the perceived (i.e., effective) bandwidth.

A compressed table scan that significantly inflates the bandwidth is implemented in GOLAP using multiple so-called *data pipelines* over SSD-resident data chunks, combining direct I/O and heavy-weight decompression. For each column of a table scan, the compressed bytes of the current data chunk are first read by a data pipeline from the appropriate storage location into an allocated GPU buffer by direct I/O (bottom part in Figure 5). When compressed data chunks from SSD are fully read into GPU buffers, they are subsequently decompressed inside another GPU buffer for uncompressed data. Decompression amounts to a kernel call parametrized by a given compression configuration. While this is a highly parallelized and compute-intensive operation, even multiple concurrent decompression kernels do not exhaust the compute resources of modern GPUs and allow for further processing kernels (e.g., operators for downstream query execution) to be executed as part of the same data pipeline.

Figure 6 (lower part) depicts the overall scheme in GOLAP: kernels for read and decompression, as well as query kernels or any subsequent operations like data transfers to CPU memory, are being enqueued in separate streams to maximize the use of the parallel GPU compute resources. To fully utilize the I/O resources of a GPU and thus maximize data loading bandwidth, multiple pipelines, as shown in Figure 6, need to be instantiated to orchestrate I/O and compute kernel calls to work simultaneously on different chunks.

Moreover, since query kernels can only start once the required chunks of all relevant columns are read and decompressed, synchronization between the data loading and query processing streams is needed; i.e., query processing can only start once data is decompressed to ensure that data of all involved columns is fully materialized uncompressed in memory when e.g., a query processing kernel is executing (marked as dashed lines in Figure 6). What is important is, that the query kernels (or transfers of uncompressed tuple data, e.g., GPU-CPU query co-execution) and next direct I/O calls can be overlapped after this synchronization shown as intra-pipeline overlap in Figure 6 . Overall, by overlapping I/O, decompression and query execution the number of CPU threads needed to drive data pipelines can be reduced (see experiments in Section 6.3).

3.4 Data Loading and Compression

Finally, we discuss how new tables are loaded into GOLAP. Here the most important question is how we find an optimal compression scheme and chunk size per column of a new table since

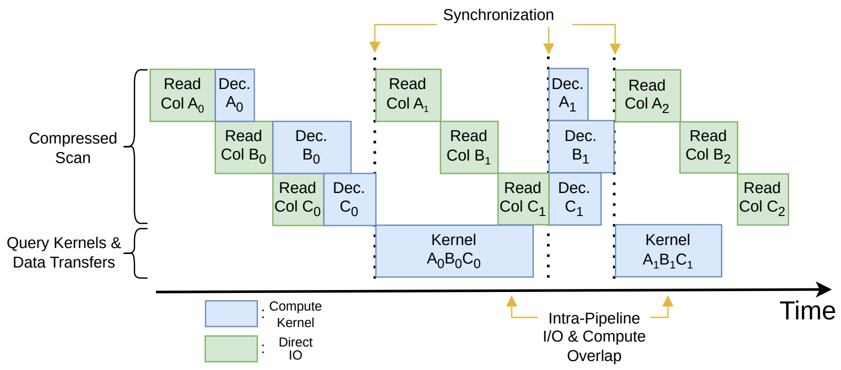


Fig. 6. Timeline of on-the-fly decompression of one data pipeline of a compressed table scan with direct I/O and heavy-weight decompression, as well as possible query kernels working on decompressed data.

different heavy-weight compression schemes result in different compression ratios depending on the data characteristics of a table. For compressing column chunks as described before, we use NVIDIA's nvcomp library [18] which provides GPU-optimized implementations of a variety of lossless heavy-weight compression schemes.

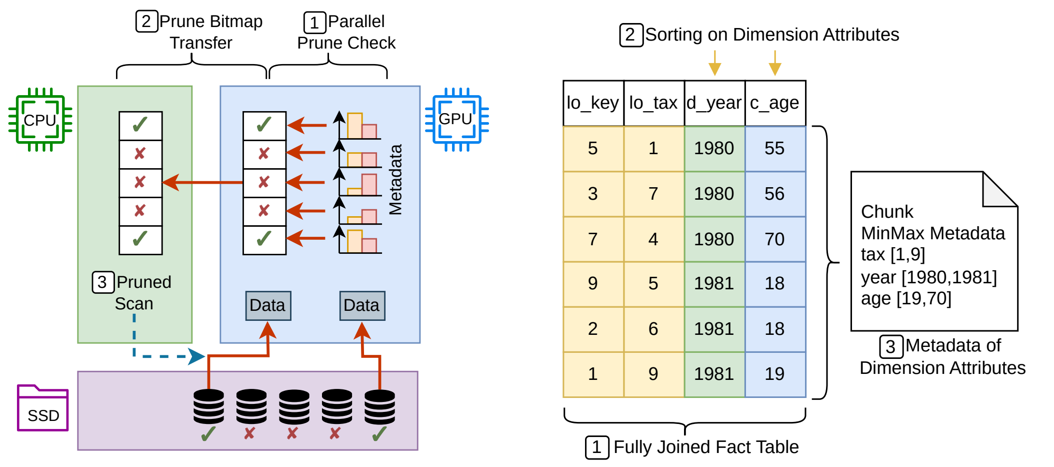
We use a random sampling-based approach to choose the right compression scheme and block sizes (for column chunks) for a given table in GOLAP. By selecting a ratio of random chunks of a larger column, we can determine the best configuration (compression scheme and chunk size) for a given objective for the sampled data using a grid search over available configuration parameters. E.g., the heavy-weight "Deflate" compression algorithm with a chunk size of 4MiB and 8 pipelines might have the highest decompression speed for a sample 1% of a column, while the "Snappy" scheme with 8MiB chunks has the highest compression ratio.

4 Opportunistic Pruning

A compressed scan can inflate the effective processing bandwidth of an SSD-backed database significantly. However, an efficient compressed scan might only improve performance up to SSD bandwidth times the compression ratio. If a system is I/O bound this way, further computational resources can be traded off to reduce the amount of data chunks that need to be read from storage, which can be accomplished by data pruning. The following discusses how we leverage this idea in GOLAP.

4.1 Inflating Bandwidth by Pruning

The key idea of pruning techniques is to save additional metadata of the contents of each data block separately to the table data (in storage or memory). Before a potentially scanned data block is read from storage, the metadata (e.g., minimum and maximum values) can be consulted first. Together with the given query predicates, a pruning decision can be made if the data block is irrelevant for answering the query at hand (i.e., the selection will not hold for any tuple in the block). The data block will only be read from storage if it cannot be ruled out that a contained tuple might qualify for the predicate. Since storage access is usually the performance bottleneck for any disk-based system, skipping otherwise unnecessary I/O on this path can greatly improve overall system performance. However, as opposed to acceleration by a compressed scan, pruning is workload-dependent and the predicate selectivity and the data storage layout influence the expected gains. In the following, we detail how GPU-optimal pruning decisions can be integrated in CPU-orchestrated I/O control flow and how the compute power of GPUs allows for more fine-grained pruning.



(a) Execution flow for GPU-accelerated pruning, exemplary for a single predicate. While the pruning decisions can be made efficiently on many chunks in parallel by the GPU, the bitmap of which chunks need to be loaded has to be copied to the CPU's main memory, since I/Os are initiated on the CPU.

(b) A fully joined fact table can be used to sort on dimension table values (year, age) and inflate fact table metadata with information on dimension attributes [34]. Depicted is a chunk that is prunable with histogram metadata for predicates that filter for age's between 20 and 54, which is not detected with MinMax pruning.

Fig. 7. GPU-based pruning and optimizations.

4.2 GPU-based Parallel Pruning

All pruning schemes can be massively parallelized over large amounts of chunk metadata, which can be exploited by GPUs parallel compute. Instead of checking (possibly intricate) predicates on the CPU during query execution, we add a parallel prune check of involved chunks before starting any data loading (Figure 7a): For each simple predicate, its satisfiability is checked against the metadata of multiple chunks in parallel on the GPU before loading the blocks.

The resulting decision bitmaps per simple predicate are then bitwise combined (using bitwise AND/OR) to create a final pruning bitmap, which is a task that profits from the GPU's high memory bandwidth. Afterward the prune bitmap has to be transferred to CPU main memory, since the GPU table scan needs to be steered by I/O threads on the CPU as described in Section 3. The CPU I/O threads initiate the CPU-controlled data pipelines for the actual table scan, but in contrast to the normal scan, we can now skip loading unnecessary data chunks based on the pruning bitmaps available on the CPU.

4.3 GPU-based Fine-grained Pruning

In addition to parallelized pruning decisions, GPUs offer ample compute power to also afford more expensive (fine-grained) pruning techniques. While MinMax metadata is a classical pruning technique which efficiently allows to prune uniform and sorted data distributions, more detailed metadata that is more expensive to check against can offer an additional trade-off of storage and compute for reduced I/O pressure.

In our system, we implement GPU-accelerated variants of bloom filter and histogram pruning approaches that allow robust pruning decisions with tunable impacts on metadata storage and compute resources: Histogram pruning allows for trading off additional storage as well as additional compute resources at pruning time for possibly more prunability by increasing the number of buckets for histogram metadata. Bloom filter-based pruning instead allows GOLAP to trade off additional storage (by using more bytes per Bloom filter) and additional compute (by using more hash functions).

Finally, since even fine-grained metadata (like a histogram or bloom filter) is much smaller than the full chunk data, metadata is usually gathered for all columns to be able to do a prune check for all possible predicates. Those fine-grained schemes provide many benefits, especially for unsorted columns, while MinMax pruning would not help much for unsorted columns.

4.4 Sideways Pruning on GPUs

Typically, OLAP workloads use a relational star schema, comprised of small dimension tables and a large fact table with foreign keys to all dimensions. For these databases, the fact table scan is the most data-intensive and the best candidate for acceleration by pruning. The predicates of OLAP queries will however usually filter the dimension table attributes. Even the fact table columns rarely allow significant pruning for unsorted data.

For these situations, we employ the optimizations for sideways pruning as described in [34] for data warehouse schemas (see Figure 7b) and adapt it to GPUs: To prepare the metadata creation for pruning, in an offline phase we therefore join the fact table with all dimension tables. This makes it possible to create fact table metadata for the values of the dimension columns referenced by each fact table tuple. This clearly increases the metadata since fact tables can be large, but given the parallel pruning, as discussed before, the overhead for scanning metadata is negligible, as we will see in our evaluation.

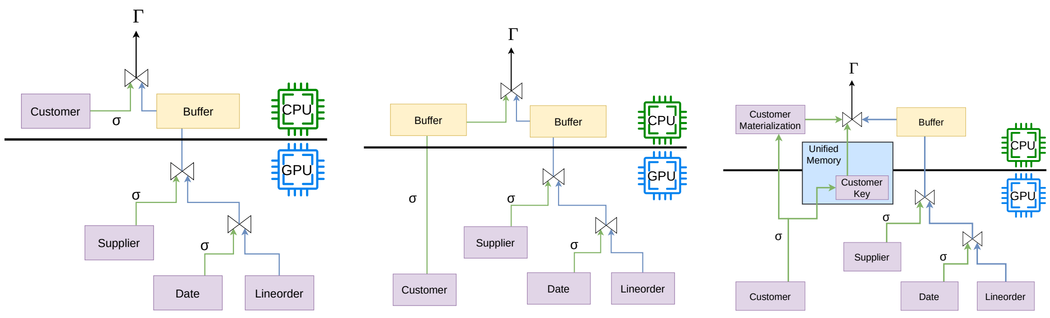
To determine the order on which to sort the set of columns on, which is important for pruning efficiency, we use the number of unique values of a column as a simple heuristic. This means fact table data is sorted by the column with the least number of contained values first, and inside groups of the same column values sorting is repeated recursively based on the next column (with the second least number of unique values). For example, as shown in Figure 7b, the fact table can be sorted on `c_region` first (5 different values), then by `d_year` (7 values), etc.

After gathering the metadata for sideways pruning, the denormalized output of the join is discarded since it might cause high storage overheads. For query processing, we only use the much smaller (derived) metadata to eliminate chunks of the fact table based on filters on dimension attributes.

Non-Star Schemas: The techniques discussed before can be applied for non-star schema joins without modifications. However, finding sort orders for efficient metadata-based pruning in general schemas is more difficult, but this is an orthogonal problem for any system that uses pruning. Similarly, for N-M joins, the metadata for pruning can be captured similarly. Since metadata of N-M join can also be represented as histograms or bloom filters, the size of metadata can be reduced by sizing these data structures appropriately. However, when reducing metadata's size w.r.t data size, the impact of the metadata on the ability to prune is expected to be lower.

5 GPU-CPU Query Co-Execution

In this section, we explain how the GPU-CPU co-execution scheme of GOLAP works to enable the processing of queries with intermediate result sizes that exceed GPU memory. For example, using a (unpartitioned) hash join to join two large tables will produce large hash tables as part of the build phase. Since current datacenter-grade GPUs still have around 10x-50x less memory than what is available as CPU main memory, out-of-memory scenarios should be considered for GPU-accelerated systems. In this section, we focus on scenarios for which large intermediate join results are expected (e.g., as detected by cardinality estimation); similar techniques also apply to other operators (e.g., group-by aggregates).



(a) Naïve Strategy (b) CPU Fallback Strategy (c) Hybrid Join Strategy

Fig. 8. Co-Execution Strategies in GOLAP. In case hash join tables exceed the GPU memory GOLAP provides the two strategies in (b) and (c). The strategy in (a) is a naïve baseline.

We present a basic strategy (CPU Naïve) for a partitioned query plan that moves whole sections of query plans that create large intermediate results to CPU. Specifically for GPU-in-data-path architecture, we propose a strategy how these parts of a query can still benefit from compressed scan performance on the GPU (and subsequent operators), even if results are moved to CPU memory afterward (called the *CPU Fallback*). Then, we show how even more parts of query processing can be GPU-accelerated by employing unified memory and splitting join hash tables into key and tuple data parts in a strategy we call *Hybrid Join*. While the techniques presented in this section generalize, we use the plan for query set 3 of the star schema benchmark [21] as a running example.

5.1 CPU Naïve

As a first alternative to a fully GPU-accelerated query, the parts of the query plan that directly lead to the large intermediate result can be scheduled for full CPU execution. In the running example of SSB query 3, the large hash table for the join of `Lineorder` and `Customer` table would be identified as exceeding GPU memory capabilities, planned as the last join in the deep query plan, and scheduled for CPU execution. The table scan, filtering and hash join build phase of `Customer` tuples can then be done by the CPU (Figure 8a). All operations until the last (large) join can be done with high bandwidth on the GPU. For SSB query 3, this means pruning and compressed scans of `Lineorder`, `Supplier` and `Date` tables, as well as filtering and joining of the three tables until intermediate results need to be materialized and transferred to a CPU main memory buffer right before the join with `Customer`. CPU threads will then finish the last join and aggregation and produce the results.

5.2 CPU Fallback

Notice that while a fully GPU-resident join of two large tables might not be feasible, previous operators on both sides might still benefit from the inflated effective bandwidth of GPU-acceleration. In the running example, a compressed scan of `Customer` table chunks, as well as the subsequent filtering can both be done at high bandwidths by the graphical processing unit, shown on the left side of Figure 8b. The key idea for this strategy is to delay a transition to CPU processing as long as possible in every branch of the query to then fall back on subsequent CPU processing at the uppermost operator that will not be fully executable on GPU. Depending on the maximum GPU-CPU interconnect bandwidth and the filter selectivity this already promises faster processing than the CPU Naïve strategy, but in the worst case should still be able to match its query run time.

5.3 Hybrid Join

Since in the CPU Naïve strategy, the CPU executed data pipeline (for the `Customer` table) will not profit from the increased effective bandwidth of GPU execution, the overall query run time is bottle-necked by I/O read bandwidth and CPU compute capabilities. For increasingly large build side tables, this degrades query performance. In the CPU Fallback strategy, a GPU-accelerated compressed scan, as well as the filtering, can be pushed to the GPU. However, the insertion into the CPU resident hash table is done by CPU threads.

We propose a strategy that allows scaling to much larger build side tables while still retaining many of the benefits of a fully GPU-accelerated system. The goal is to make use of the abundant CPU memory, while most of the processing is done on the graphical processor. Here, we built upon the possibility of oversubscribing GPU memory using NVIDIA Unified Memory [20]. This feature allows the GPU to directly access pages of specially allocated CPU memory in device kernels. Accessing CPU-resident unified memory from GPU will generate a page fault and initiate a page migration from host to device memory.

The main idea of a hybrid join is to split the hash join data structures into two parts: A materialization table containing the scanned tuple data of the build side table, as well as a build side join key hash table. While the materialization table contains the largest part of necessary join information (all necessary columns of the build side), writing into it can be done sequentially in dense blocks of data. The actual join key hash table has far less deterministic access patterns (by definition), since adding a join key is a random write access in the range of allocated hash buckets. However, in comparison to the materialized tuple data, the join key hash table is limited in size. A hybrid join will place the hash key table in unified memory (since it could still be too large for full GPU memory residency) to benefit from the possible memory over-subscription. However, the build side materialization table does not profit from unified memory the same way, since data is only written once (a traditional pinned CPU memory buffer suffices).

After the build side has finished processing, all remaining join key hash table data in unified memory can be prefetched to the CPU using a respective memory hint. For the probe side query plan, similarly, as much as possible (compressed scan, filtering, previous joins) will be scheduled for GPU execution. After tuples materialize in a CPU buffer, the last join can be executed by CPU threads (same as for the other strategies).

This approach has two main advantages: First, it can profit from the very high effective data loading bandwidths for both the build and probe side (not only the probe side, as CPU Naïve). Second, the processing required for a lock-free build side execution is much faster on the graphical processing units, even with possible page migrations. While paging traffic over the GPU-CPU interconnect will naturally increase for large build side tables, performance will degrade much more gracefully for this Hybrid Join.

6 Experimental Evaluation

In this section, we investigate the performance of GOLAP. First, we report query end-to-end processing performance of GOLAP. Then, we discuss in detail the effects of compression and pruning on scan performance. Afterward, we report the results of experiments on GPU-CPU co-execution and finally compare the overall query performance of our system against two other baseline systems.

6.1 Setup

System: All experiments are run on an NVIDIA DGX A100 machine running Ubuntu 22.04 with CUDA library 12.2 and Linux kernel 5.15. The CPUs used are AMD EPYC 7742 64-Core processors, and the GPUs are NVIDIA A100s with 40 GB device memory. CPUs and GPUs are connected via

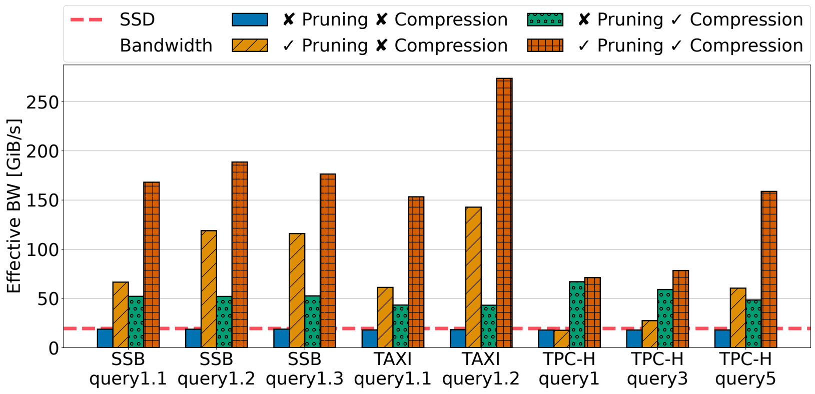


Fig. 9. The effective bandwidth of the system in different query processing settings: Processing uncompressed data, adding pruning, processing compressed data without pruning and a setting with both compression and pruning enabled.

16 PCIe 4.0 lanes. The system uses 3 Samsung SSDs in a RAID 0 configuration for direct storage access over 16 PCIe 4.0 lanes with a maximum read bandwidth of 19.5 GiB/s measured by `fio`[2]. For every experiment, data is loaded from storage; i.e., no parts are cached in CPU or GPU memory. If GPU-accelerated data pruning is enabled, it is explicitly stated in the respective experiments. The main metric in many experiments is the *Effective Bandwidth*, which we define as the number of uncompressed bytes processed (including pruned data) divided by the end-to-end system runtime (from before any data is read from storage until query results are materialized in CPU main memory). For a compressed scan, this is the same as the decompression speed; for an uncompressed scan, the effective bandwidth equals the I/O read bandwidth.

Datasets: We use data from the star schema benchmark [21] (**SSB**, scale factor 200, 120GiB), TPC-H benchmark [29] (**TPC-H**, scale factor 200, 130GiB without comments columns) as well as real-world data gathered from the NYC yellow taxi trip records [27] (**Taxi**, 5 years worth, 60GiB). Real-world data is in particular interesting for the effect of compression on those data distributions which are typically different from (synthetic) benchmark data. For the SSB and TPC-H datasets we use the queries of the benchmark specification, without `orderby` and `limit`. As workload for the taxi data set, we use two query sets which are part of the Google BigQuery public workload: The first query is an aggregation over the trips each taxi made each month. The second query computes the average speed of taxis during each day. For both queries we vary predicates on trip distance and fare amount, respectively.

6.2 Exp. 1: End-to-end Query Execution

We evaluate full queries from start to finish, measuring the time until results are available to the CPU. To better understand the impact of GOLAP's compression and pruning techniques on the effective bandwidth, we execute the experiment workloads with and without enabling them. These mechanisms are orthogonal and can be used either in combination or individually. For brevity, we will focus on SSB query 1, the first Taxi query, and the TPC-H queries 1, 3, 5, and in this experiment. At the same time, a comprehensive evaluation of the full SSB and Taxi benchmarks is presented in 6.6.

Figure 9 shows the effective bandwidth across the three datasets and various queries. Initially, without pruning and compression, GOLAP's effective bandwidth is constrained by the I/O bandwidth of the storage device. When parallel pruning is enabled (as shown in the second bar), the amount of

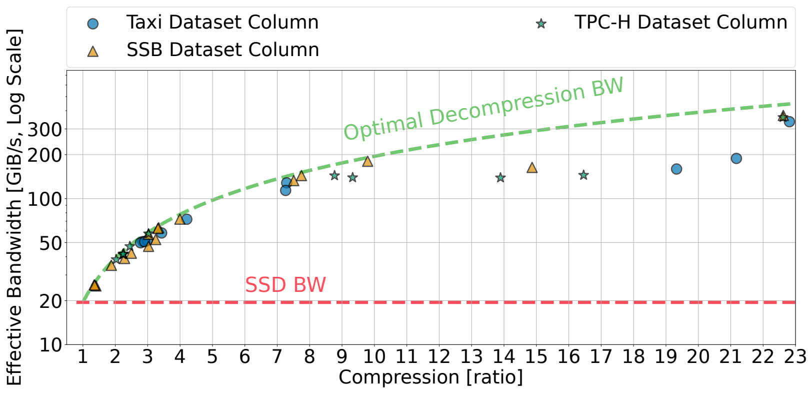


Fig. 10. Effectiveness of Compressed Scan. Compression ratio and effective bandwidth of column scans in SSB, Taxi and TPC-H datasets. The compression ratio and the flash device's read bandwidth give the optimal decompression bandwidth (green dashed line). Note the logarithmic scale of the bandwidth axis.

data accessed is reduced by approximately 4-9x, consequently increasing the effective bandwidth by 3-8x. The impact of pruning largely depends on the selectivity of the queries; thus, SSB query 1.2, 1.3, and Taxi query 1.2 benefit most from pruning, as they are more selective. On the other hand, TPC-H query 1 has very low selectivity and thus needs to scan all tuples in most chunks, leading to small speedups from pruning. For TPC-H query 5, we see large speedups when employing the sideways pruning techniques, by pruning `lineorder` tuples by joined `order` tuples metadata, resulting in a roughly 3 time increase in effective bandwidth.

When using a compressed scan without a pruning phase (represented in the third bar), performance improves by more than 2.5x, closely aligning with the compression ratios of 2.6 for Taxi and 2.8 for SSB. In this scenario, the variance in selectivity among queries does not affect performance, as all data is read when pruning is disabled. Furthermore, the influence of selectivity on query processing is minimal within GOLAP's chunk-based query kernels since we stream the data through the GPU and are not CPU-GPU interconnect bound in these queries. Consequently, TPC-H query 1, with its low selectivity, still sees significant performance speedups by the compressed scan.

When pruning and compression are enabled simultaneously (shown in the fourth bar), GOLAP achieves an effective bandwidth approximately 8 times greater than the flash read bandwidth. This bandwidth improvement stems from the compression factors (approximately 2.8 for SSB, 2.6 for Taxi queries and between 2.7 and 5 for the TPC-H queries) with the reduction in data loaded post-pruning – typically three times less data for all queries, except for the highly selective Taxi query 1.2, which loads seven times less data. For all tested queries, the combination of pruning and heavy compression results in the highest effective query processing bandwidth. It is important to note that the effective bandwidth is still constrained by the SSD bandwidth, not by our system's limitations.

In the following experiments we will zoom in on the individual contributions and analyze the impact of various parameters.

6.3 Exp. 2 Compressed Scan

This section delves deeper into how storage access and compressed table scans enable the system's efficiency.

Effectiveness of Compressed Scan: As discussed in Section 3, the optimal effective bandwidth of a compressed column scan (without any pruning) is given by the maximum SSD read bandwidth

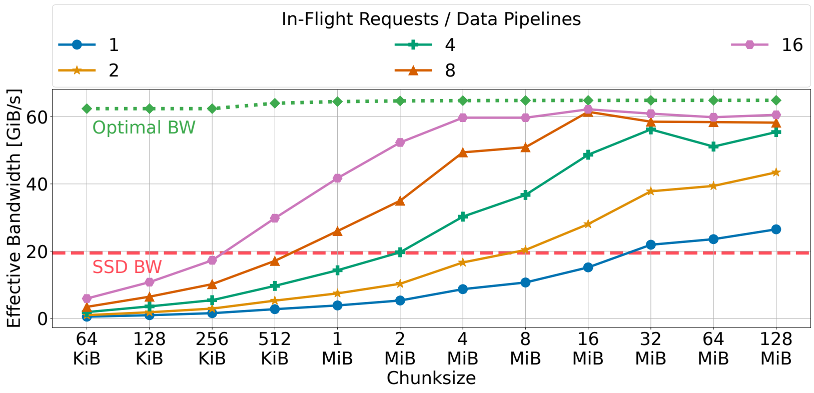


Fig. 11. Importance of Concurrent I/O and Decompression. Bandwidth for different chunk sizes and number of concurrent I/O requests triggered by the same number of CPU I/O threads. Red dashed line marks the maximum read bandwidth of the SSD. The Green dotted line marks the maximum possible bandwidth as the compression ratio dictates.

times the compression ratio. For example, if the compression ratio is 2, the maximum attainable bandwidth is 2 times the storage read bandwidth since twice the amount of original information can be transferred simultaneously (but a decompression step is needed). In Figure 10, we show the effective bandwidth of column scans of the SSB, Taxi, and TPC-H dataset columns. For each column, we run a streamed decompressed scan using the combination of chunk size and compression scheme reaching the lowest overall runtime. Plotted is the effective bandwidth (= decompression speed for a scan) at the achieved compression ratio. We derive three key insights from this experiment: First, the column compression ratios for both synthetic and real-world datasets suggest a promising theoretical effective bandwidth exceeding 50 GiB/s, given realistic SSD read bandwidths. Second, our system achieves bandwidths close to this theoretical limit despite the computational overhead required for decompression. Additionally, it is noteworthy that even at lower compression ratios, the added decompression task does not impede the scanning process, underscoring the robustness of our approach. Finally, with compression ratios greater than 10 (as seen in low entropy columns like those in the Taxi dataset, or some TPC-H columns), the heavy-weight decompression kernels that we use as part of the `nvcomp` library struggle to maintain high effective bandwidths, although the needed decompression bandwidth is still below the maximum GPU memory read and write bandwidth. This scenario also presents an interesting avenue for further research into GPU-tailored (de-) compression techniques, especially for highly compressible data.

Concurrent I/O and Decompression: In our second experiment, we show the importance of using multiple I/O threads on the CPU to maximize the bandwidth of the GPU scan. For this, we study the effect of varying chunk sizes and the number of CPU-orchestrated data pipelines doing I/O and decompression. We plot the achieved effective bandwidth over increasing chunk sizes in Figure 11. We use a column with moderate compression, SSB data set column `commitdate` with a compression ratio of around 3.2 (see Figure 10 for performance over different compression ratios). Each plotted line corresponds to a setting where the system keeps up to 1, 2, 4, 8 or 16 I/O requests of that specific chunk size in-flight (= number of data pipelines). We plot the theoretical optimal data access bandwidth (green dotted line), computed as the column's compression ratio at this chunk size multiplied by the SSD read bandwidth. With fewer data pipelines (e.g., 4), larger chunk sizes are needed (32 MiB) to reach effective bandwidths near the upper bound. With only 1 thread, we can not reach the full bandwidth at all. It becomes apparent that to reach close to optimal

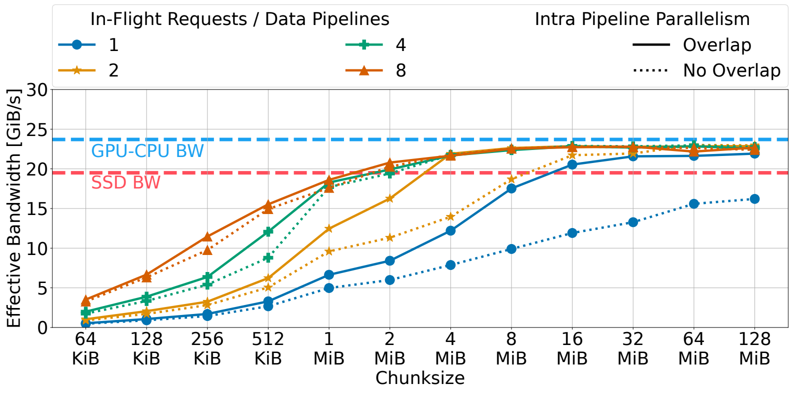


Fig. 12. Importance of Overlap of Data Access and Query Execution for a Compressed Scan with CPU materialization. The results show that overlap significantly improves the effective bandwidth and allows GOLAP to reduce the number of I/O threads on the CPU, which trigger the scan.

effective bandwidths, concurrent requests are crucial to exhaust I/O capabilities and efficiently schedule decompression kernels on the graphical processor.

Overlap of Data Access and Query Execution: The next experiment shows the importance of overlapping I/O and decompression work inside a data pipeline with downstream operations like GPU query kernel execution or data transfers. This is, in particular, important for GPU-CPU co-execution since the GPU kernel in such cases, needs to transfer the result of a partial plan (e.g., the output of a filter) to the CPU memory before the CPU side of the plan is executed, as discussed in Section 5. In such scenarios, overlapping is particularly important since the copy step can cause significant overhead if it blocks subsequent I/O operations since we would start the I/O only once the previous query kernel is finished. For the experiment, we use a simple setup to measure this effect: We execute a GPU table scan, and the query kernel only copies the results of the GPU scan to the CPU memory. We execute the same query as before (e.g., a scan of column `commitdate`) but additionally copy the result in the query kernel to the CPU memory one time with and one time without overlap.

Figure 12 shows the results. The dotted lines show results without overlap of data transfer operations and I/O reads. While close to optimal performance is attainable in this setting, either large chunk sizes or more than 4 concurrent data pipelines are necessary. Enabling intra-pipeline overlap of I/O and data transfer improves this, especially for a smaller number of data pipelines, reaching expected bandwidths much earlier. A noteworthy characteristic of a compressed scan with data materialization in CPU main memory but no filtering is that while the effective bandwidth can not possibly exceed the bandwidth of the GPU-CPU interconnect, it might still promise performance gains over the SSD bandwidth. In our system, while the scan with CPU materialization is hard-bound by around 23 GiB/s, this is still an improvement over the 19 GiB/s of raw SSD read bandwidth. With increasing bandwidths of in-system interconnects, like AMD Infinity Fabric or NVIDIA NVLINK, we suspect this might be an even more favorable source of performance improvement.

Quality of Sample-based Compression Selection: As we discussed before in Section 3, the configuration parameters of a compressed data chunk, like the choice of compression algorithm and size of the chunk, can vastly influence the overall data loading performance. In this experiment, we evaluate the sampling-based configuration choice for GPU-accelerated decompression in our system. We sample contiguous chunks of data of each column of a dataset up to a maximum ratio of

Table 1. Average relative deviation between optimal objective value and value reached by configuration chosen by sampling. 1.0 is the optimal relative deviation, meaning the chosen configuration chosen by sampling reached the optimal value for the objective on the full column.

Sampling Objective		Best Time			Best Ratio		
Sampling Ratio		1%	5%	10%	1%	5%	10%
SSB	AVG	1.26	1.13	1.06	1.01	1.0	1.0
	MAX	2.4	1.43	1.26	1.01	1.0	1.0
Taxi	AVG	2.03	1.36	1.44	1.033	1.03	1.05
	MAX	4.15	3.32	2.06	1.26	1.11	1.3

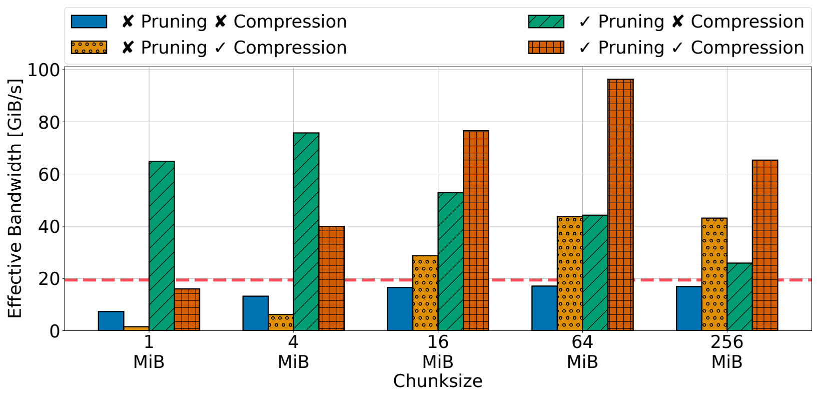


Fig. 13. Average effective bandwidth of SSB set 1 queries over different chunksizes.

the total column size. We enumerate available compression configurations on the sampled column and measure relevant metrics during compression and subsequent decompression of the sampled data. Based on these results, we select the two best configurations relevant to decompression speed and compression ratio objectives. Especially for this experiment, we run an enumeration of configuration on the complete columns of the datasets. We then define the error as the relative deviation between the actual best objective value reachable (time or compression ratio) and the objective value is reached if the configuration chosen based on the sampled column data is applied to the full column.

We report the average and maximum relative deviation for different datasets and sample ratios in table 1. It becomes apparent that even for small sample sizes of 1%, close to optimal configurations can be assessed for optimizing the compression ratio for both data sets. However, the configuration for the best decompression time is harder to determine based on a reduced dataset. A reason for this is that the absolute sizes of a sampled column are relatively small and, as such, do not allow an efficient execution over a longer time period. For larger sample ratios of about 10%, this is not the case and the average relative deviation ratios are close to optimal. For the taxi dataset, another reason for larger average and maximum relative errors is that many columns have low entropy and reach very high compression rates. Here, the very low decompression times skew the otherwise close results of the moderately compressible columns.

In all experiments of this article that involve compressed scans, we use the selected compression parameters that reach the best runtime on 10% of sampled data, if not stated otherwise.

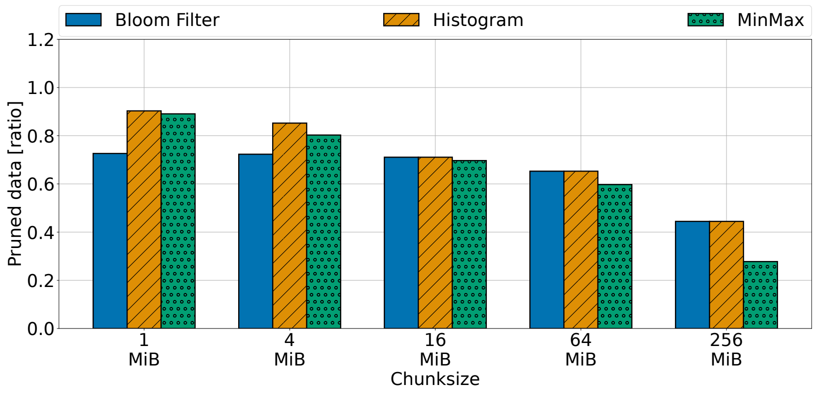


Fig. 14. Average ratio of pruned chunks for SSB set 1 queries over different chunksizes and pruning techniques.

6.4 Exp. 3: Opportunistic Pruning

Having thoroughly examined the efficiency of our compressed scan, we now explore opportunistic pruning, another crucial technique within GOLAP to improve the effective bandwidth. We will particularly focus on how compression, various pruning strategies, and chunk sizes interplay in GOLAP. For the experiment, we use SSB query set 1, since it uses predicates on both dimension and fact table attributes and only includes one join (minimizing unrelated effects of query execution). First, we use MinMax, Bloom (for point predicates, 1KiB, 8 hashes) as well as histogram (256 bins) pruning, with both dimension-based data clustering and sorting optimizations enabled (see [34] and Section 4.4): The fact table is sorted based on the dimension attributes that are used as filter predicates and metadata of these columns is gathered in the fact table.

We show the effective bandwidths achieved in Figure 13, and the prune ratios as an accompanying figure in 14. Since pruning time has low overhead for all variants (less than 1% of total query time, including metadata transfer) the ratio of pruned data chunks is the performance-defining metric. Here, Figure 14 suggests that histogram-based pruning is a robust choice, reaching the highest ratio of pruned data across all chunk sizes.

As a baseline, we plot results for not using compression or pruning (blue bar, Figure 13). The system benefits from larger chunk sizes but can not pass the SSD bandwidth limitations. The second light orange bar shows results for using compressed scans but no data pruning. Clearly, larger chunk sizes benefit the compressed scan (better compression ratios, better I/O and compute utilization). The apparent overhead of compression for small chunk sizes does not generalize to other settings, since these configurations are sub-optimal, even for uncompressed accesses.

Results of using only pruning of data chunks and no compression are reported as the green bars in Figure 13. Large chunk sizes understandably offer less pruning capabilities since the potential amount of contained values per chunk increases, reducing the amount of prunable data. Small chunks with limited data variety achieve this scenario's best query runtimes. Even though the amount of processing needed to make the pruning decisions increases, it has negligible latency since it is done in parallel.

Finally, the orange bars in Figure 13 show the effects of pruning and compression together. Note that the disadvantage of compression for small chunks and the ineffectiveness of pruning for very large chunks dominate the respective edges of the chunk size range. To balance I/O sizes for both efficient GPU-accelerated decompression and pruning, we suggest to optimize for decompression first, since the expected worst-case gains are generally higher, and then study pruning techniques

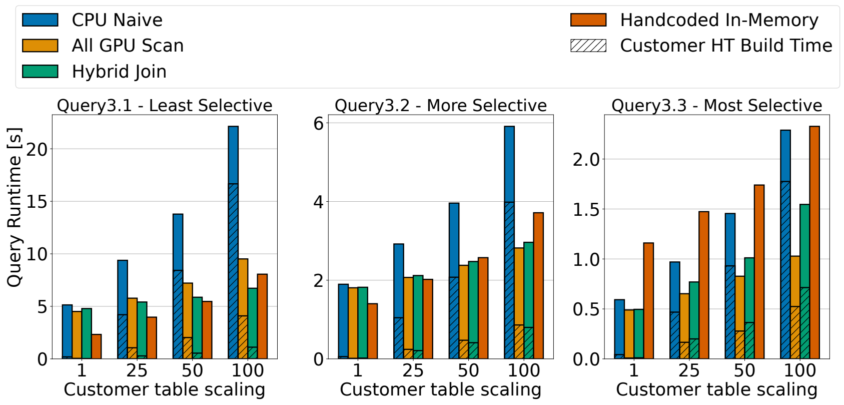


Fig. 15. Runtimes of SSB query set 3 with increasing join table sizes using different GPU-CPU co-execution strategies. Note the different Y-Axes.

and data distributions. In our evaluation, we constantly saw that the shortest runtimes (i.e., the highest bandwidths) can be achieved for setting the maximum chunk size to a value between 16 and 64 MiB. In this range, I/O and compute can be used efficiently for decompression, yet it is still possible and beneficial for pruning.

6.5 Exp. 4: GPU-CPU Co-Execution

To study GPU-CPU co-execution of analytical workloads in scenarios with large intermediate results, we use SSB query set 3 and augment the dataset with a scaled version of the `Customer` table. We linearly scale this dimension table with a given factor (from 6 Million tuples at `Customer` factor 1 to 600 Million at `Customer` factor 100) and update the fact table to reference tuples in the updated table with the original distribution. When using a hash join to join the `Customer` and `Lineorder` tables, this leads to a linear increase in the size of the build side hash table. We implement and compare the CPU Naive, All GPU Scan and Hybrid Join strategy in `GOLAP`. runtimes of the 3 strategies with increasing join hash table size is shown in Figure 15. Also depicted for comparison are runtimes of a hand-coded in-memory version of the same data and queries. We omit the results of SSB query 3.4 since selectivity and runtime are similar to query 3.3.

For all co-execution strategies, the last step of executing the large join on the CPU is similar. The remaining runtime of the query after the `Customer` hash table for the join is prepared is consequently similar (seen in the figure as the unmarked part of the bars). The time the `Customer` build side takes varies highly between the strategies. As expected, the CPU naive query execution strategy without the compressed scan spends more runtime on this part, slowing down overall query execution. All GPU Scan and Hybrid Join can significantly reduce this large part of the query runtime (by at least 3x across table scalings and selectivities).

The Hybrid Join solution employing paged memory and split hash tables can additionally gain advantages in scenarios with increased computational need. Especially for SSB query 3.1, with low selectivity and a high number of `Customer` tuples that need to be inserted into the hash table, GPU-acceleration for the hash table insertion pays off, leading to the lowest query runtimes. For more selective queries, it is not as beneficial to incur the additional memory page migration overhead, leading to similar and even worse runtimes for queries 3.2 and 3.3, respectively. However, as for highly selective queries, query runtimes are much lower already. The hybrid join is the preferred strategy in general, offering robust query time improvements across selectivities.

In comparison to our hand-coded CPU in-memory solution, CPU Naive is always slower except for the highly selective scenarios. However, all GPU Scan and Hybrid Join are generally competitive (for less selective queries) and even faster for highly selective queries, where the least amount of data needs to be materialized to CPU memory. This leads us to conclude that these GPU-CPU co-execution strategies will in the worst-case not slow down the system, but can exploit the GPU's acceleration potential in common scenarios to achieve considerable speedup.

6.6 Exp. 5: System Comparison

In this experiment, we benchmark against various GPU and CPU systems by comparing raw query runtimes of different benchmarks (SSB, Taxi, and TPC-H). While available GPU-accelerated systems such as CuDF [17] that support flash access and compression were also tested, they were not competitive compared to CPU systems across all experiments and were thus excluded from the reported results. As an existing GPU-accelerated system, we compare GOLAP against the Mordred prototype [32] (which only supports SSB queries). Mordred is a GPU-CPU co-processing systems for in-memory workloads and various caching strategies.

In addition, we include results from a commercial CPU system, referred to as "System X," famous for its robust performance with larger-than-memory data sets. This comparison is particularly relevant as System X employs light-weight compression of columns, providing an interesting baseline for our analysis.

Furthermore, we include DuckDB, an embedded CPU OLAP system, utilizing views of Parquet files with optimized row group sizes. The setup ensures that no data is cached in memory at the start of a query and allows to test different compression and metadata pruning scenarios. We allocate 64 CPU threads and allow main memory allocations several times the size of the datasets. Since DuckDB supports heavy-weight compression in Parquet files, we present runtimes for uncompressed data and compressed data, noting that the best performance was achieved using Snappy compression. Additionally, to show the effect of pruning, we generate Parquet files with data ordered the same as GOLAP. We only show DuckDB with pruning on SSB since the gains are similar on the other queries where GOLAP outperforms DuckDB.

Figure 16 displays the runtimes for query sets on the SSB, Taxi, and TPC-H datasets, the latter two without pruned DuckDB and Mordred, as explained before. A consistent observation with CPU systems, as seen in the DuckDB results, is that despite heavy-weight compression reducing data sizes by approximately 2.5x, the difference in runtime between uncompressed (green bar) and compressed access (yellow bar) is less than that, indicating that the execution is predominantly compute-bound in both scenarios.

In comparison, System X demonstrates a consistent speedup across all query sets, utilizing a database with light-weight compression (compression ratio around 2x). On the other hand, GOLAP (depicted as the first, blue bar, without pruning) is capable of processing heavy-weight compressed data at near-optimal bandwidth, achieving substantial speedups in query execution runtime—ranging from 4x to 23x—over the CPU-based systems. Only the GPU-CPU system Mordred matches the runtimes of GOLAP on the SSB queries but assumes that data is memory-resident and even (partly) accelerator-resident.

6.7 Exp. 6: System Cost

As a last experiment, we evaluate the costs of a GPU-in-data-path system. Since the costs of servers, CPU, and GPU hardware are hard to compare between different models, we choose a cloud setting (and, in particular, AWS EC2 instances) to quantify the expected price metrics. For this, we ranked the CPU/GPU instances from AWS EC2 with an SSD attached by their cost per I/O operations per second (IOPS/\$). For pure CPU-based execution, the instance type i3 offers the best

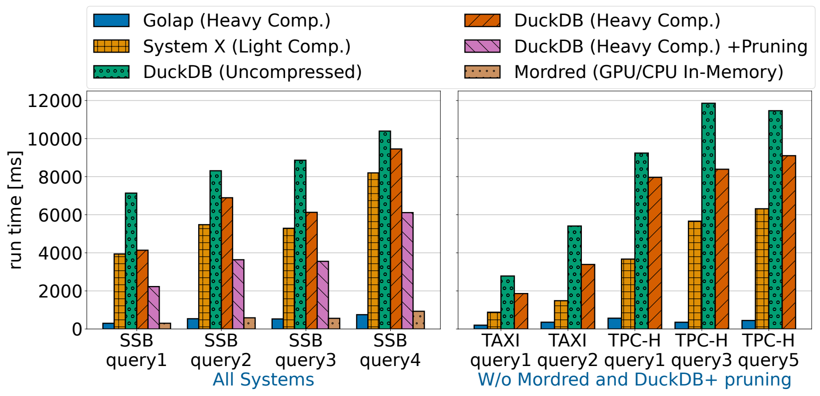


Fig. 16. Query runtimes for different systems over synthetic and real world datasets.

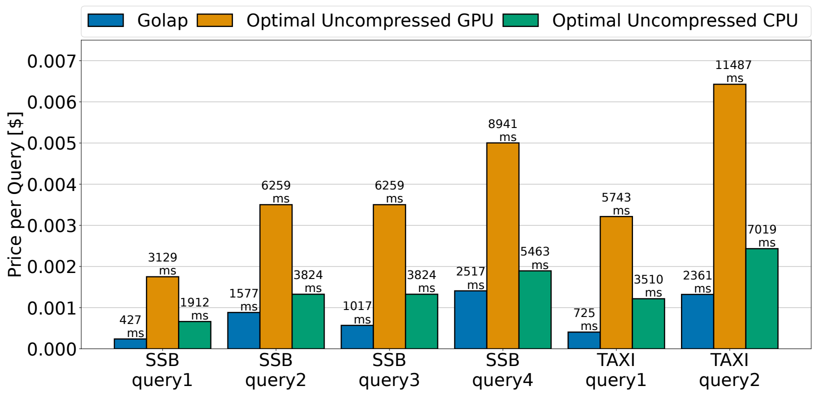


Fig. 17. Price per query on the GPU and CPU instances (bars). The optimal price is computed based on the measured I/O bandwidth. Query run times annotated above bars.

cost performance (in IOPS/$). For GPU-accelerated instances, the type `g6.8xlarge` with a single NVIDIA L4 GPU and 32 vCPU cores is the most efficient one, with a three times worse IOPS/$ ratio. As the CPU instance type, we choose the slice of the `i3` instance that has at least as much SSD read bandwidth as the GPU instance (4 GiB/s vs. 2.5 GiB/s).

For the experiment, we report the price in $ per query that a CPU and a GPU-based system causes. For this, we use the following procedure: we first measure the maximum bandwidth of sequential reads from the instance SSD storage for CPU and GPU instances. Afterward, we analytically compute the runtime of SSB and Taxi queries (on scale factors 100 and 60, respectively, without pruning) by dividing the data size by bandwidth. Subsequently, based on this, we derive the cost per query a user would need to pay.

This analytical way to compute query cost for a GPU and a CPU-based system yields the lower bound of the costs of running each query and removes incidental implementation details. We do this for a CPU and a GPU-based system without compression since compression is harder to estimate. For GOLAP (GPU-system on compressed data), we actually run all the queries on the GPU-enabled I/O instance `g6.8xlarge` and report the actual query prices, i.e., the costs incurred.

The prices of running the SSB and Taxi queries are depicted in Figure 17. While the price of processing the uncompressed data in the GPU instance (yellow bar) is roughly 3 times higher than in the CPU one (green bar), the price of running GOLAP (GPU compressed) (blue bar) is way lower than the optimal price of an uncompressed CPU system. The price decrease ratio between optimal uncompressed GPU and GOLAP surprisingly even surpasses the compression ratio, which we attribute to additional caches and optimizations inside the enterprise-grade SSDs in the cloud. The query run times also favor the bandwidth-effective GPU compressed solution, even though for the evaluated instances, the SSD read bandwidth is lower for the GPU-enabled instance.

7 Discussion: Query Optimization

In our evaluation, we have shown the benefits of a GPU-in-data-path architecture. However, integrating this in full DBMS opens up additional challenges. One important challenge is query optimization. In the following, we discuss potential routes for this.

The first challenge for query optimization is deciding on the operator placement (GPU vs. CPU). Established statistics of query planners already offer valuable information for this. For example, cardinality estimation can approximate the intermediate sizes of hash join build tables, which can be used to decide if a plan can be fully executed in GPU or if CPU-GPU co-execution is needed.

Second, there is potential in rethinking join ordering. Instead of executing the join order, which minimizes the aggregated sum of cardinalities, we also see potential for other join orders. The intuition is that plans that maximize work that can be jointly executed on the GPU might be beneficial instead of those that necessarily minimize cardinalities overall. An example would be a plan starting with a huge join that prevents all upstream operators from executing on the GPU. An alternative plan that starts instead with smaller joins that can be located on the GPU thus might be beneficial since the processing on the GPU speeds up the overall query execution.

Third, as a consequence, we think cost estimation models need to change and cost estimation for data paths thus needs to be made access bandwidth- (for CPU/GPU/interconnect) and compression-aware since this allows us to better estimate the benefits of placing operators on the GPU. For example, a column with a compression ratio of 5 can be accessed at five times the SSD bandwidth. Integrating this information into cost models helps us to better estimate the overall runtime of query plans in a GPU-in-data-path architecture.

Finally, we want to make the optimizer pruning aware since this further amplifies the bandwidth of data access on the SSD from the GPU. For this, the sort orders of involved columns and the existence of metadata for table chunks need to be taken into account to allow for estimates of the data pruning opportunities for a given plan.

8 Related Work

For related work, we focus on three sub-areas of database research.

Analytical Query Processing on GPUs. The highly parallel execution model, as well as the high memory bandwidth have made graphical processors most interesting for the data-intensive nature of OLAP. Here, single operators like joins (e.g., [11, 26]) have traditionally seen the most active research. There has also been substantial work on GPUs as accelerators of partial or complete query plans for analytical processing. [24] analyzes the performance of GPU-based analytical workloads given a tile-based execution model with minimal materialization. [33] discusses warehouse queries on GPU-enabled systems and analyzes the effects of light-weight compression in these. [16, 32] explore data placement techniques and analytical query execution for systems with GPUs and CPUs, with the latter putting a focus on caching strategies for disk-based data access. The most closely related previous work is [14], which also tackles the problem of analytical query processing of SSD-resident data on GPUs using direct data transfers. In contrast to our work, they implement their own primitives for direct I/O, which use heavy CPU processing, they study light-weight compression techniques, do not include GPU-CPU co-execution, and require that intermediate query results fit in GPU memory. [23] implements GPU-initiated SSD data transfers using custom Linux drivers that map NVME data structures to GPU Memory using GPU Direct RDMA. They compare their system against GPU Direct Storage (GDS) and show how it can benefit graph algorithms and other analytical workloads. Finally, [3, 5, 6, 28] show the opportunities of GPU-CPU co-processing in general. Unlike those results, we mainly focus on how to keep scans and decompression on the GPU, which is important for an SSD-based GPU DBMS.

Compressed Databases. Significant work has been done on compression for disk-based, in-memory, and GPU-accelerated DBMSs. with light-weight compression techniques, e.g., dictionary/ run-length encodings, null suppression, etc., and have seen the most active research and application. [8] is a report on light-weight compression in disk-based database systems and the possible performance improvements for various schemes and workloads. [30] discusses how to use the CPU's SIMD capabilities to enable fast filtering and decompression of null-suppressed data, reaching high effective data processing bandwidths. The authors discuss that while their methods could be integrated on GPUs, the limited CPU-GPU interconnect bandwidth can not sustain the bandwidth advantage they achieve on CPUs in the in-memory scenario (see also Figure 12). [4, 10] discuss compression-aware query optimization and processing respectively. [31] explores automatic adaptive compression of cold column sections based on access statistics.

Light-weight compression on graphical processors has been explored in [7], in which the authors detail the advantage of these approaches generally and the potential in light of limited CPU-GPU bandwidth especially. [25] and [1] explore how light-weight GPU compression techniques can be optimized to keep data in faster register or on-chip memory during a decompression and query processing pipeline. All these approaches are naturally not data agnostic.

Processing near Storage. The authors of [22] envision a database running fully on a computational SSD. They report how such a system can significantly reduce I/O time (which takes up the largest part of total benchmark runtime) for typical transactional workloads. The opportunities and design space of co-designing applications and storage devices are detailed in [13].

While these advances in future hardware for data processing emphasize the need for further optimizations of the I/O path, we regard these solutions as mostly orthogonal. For example, our system can still benefit from light-weight near-storage pruning or selection mechanisms, while computationally intensive operations do not seem to be feasible (or desirable in the case of heavy-weight decompression) to be executed on SSDs.

9 Conclusion

In this paper, we outlined the potential of GPU-acceleration for analytical workloads over flash-based data. We discussed how direct access to storage, heavy-weight compression, and GPU-based data pruning techniques could be used to build a system that robustly achieves high processing bandwidths previously unattainable for flash-based CPU and GPU systems. We investigated how CPU and GPU can co-process queries that exceed the graphical device's available memory capabilities.

Acknowledgments

We gratefully acknowledge support by the Federal Ministry of Education and Research (BMBF) under Grant No. 01IS22091. We also thanks hessian.AI and DFKI for their support.

References

[1] Azim Afroozeh, Lotte Felius, and Peter A. Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*. ACM, 8:1–8:11. https://doi.org/10.1145/3662010.3663450

[2] Jens Axboe. 2022. Flexible I/O Tester. https://github.com/axboe/fio

[3] 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

[4] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query Optimization In Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 271–282. https://doi.org/10.1145/375663.375692

[5] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p127-chrysogelos-cidr19.pdf

[6] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2007. GPUQP: query co-processing using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 1061–1063.

[7] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1 (2010), 670–680. https://doi.org/10.14778/1920841.1920927

[8] G Graefe and LD Shapiro. 1991. Data compression and database performance. In *1991 Symposium on Applied Computing*. IEEE Computer Society, 22–23.

[9] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf

[10] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond. In *Data Management on New Hardware - 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2016 and 4th International Workshop on In-Memory Data Management and Analytics, IMDM 2016, New Delhi, India, September 1, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10195)*, Spyros Blanas, Rajesh Bordawekar, Tirthankar Lahiri, Justin J. Levandoski, and Andrew Pavlo (Eds.). Springer, 40–56. https://doi.org/10.1007/978-3-319-56111-0_3

[11] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, Shimin Chen and Stavros Harizopoulos (Eds.). ACM, 55–62. https://doi.org/10.1145/2236584.2236592

[12] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[13] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2852–2858. https://doi.org/10.1145/3448016.3457540

[14] 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 (2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[15] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[16] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf

[17] NVIDIA. 2024. cuDF. https://github.com/rapidsai/cudf

[18] NVIDIA. 2024. nvcomp Library. https://developer.nvidia.com/nvcomp

[19] NVIDIA. 2024. NVIDIA GPU Direct Storage. https://docs.nvidia.com/gpudirect-storage/

[20] NVIDIA. 2024. NVIDIA Unified Memory. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/

[21] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d500381f36e3599400db896173c21bd395dd090d

[22] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang Won Lee. 2021. SaS: SSD as SQL Database System. *Proc. VLDB Endow.* 14, 9 (2021), 1481–1488. https://doi.org/10.14778/3461535.3461538

[23] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Brian Park, Jinjun Xiong, Chris J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William J. Dally, and Wen-Mei W. Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 325–339. https://doi.org/10.1145/3575693.3575748

[24] 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1617–1632. https://doi.org/10.1145/3318464.3380595

[25] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based Lightweight Integer Compression in GPU. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1390–1403. https://doi.org/10.1145/3514221.3526132

[26] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[27] TLC. 2024. Trip Record Data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[28] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. 2018. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–10. http://www.adms-conf.org/2018-camera-ready/tome_groupby.pdf

[29] Transaction Processing Performance Council (TPC). 1993. TPC BENCHMARK H (Decision Support) Standard Specification Revision 3.0.1. http://www.tpc.org

[30] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (2009), 385–394. https://doi.org/10.14778/1687627.1687671

[31] Leon Windheuser, Christoph Anneser, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2024. Adaptive Compression for Databases. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 143–149. https://doi.org/10.48786/EDBT.2024.13

[32] 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 (2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[33] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[34] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Janaki Lahorani, Dmitry Potapov, and Murali Krishna. 2017. Dimensions Based Data Clustering and Zone Maps. *Proc. VLDB Endow.* 10, 12 (2017), 1622–1633. https://doi.org/10.14778/3137765.3137769

Received April 2024; revised July 2024; accepted August 2024