HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics[*]

Jing Li   Hung-Wei Tseng[†]   Chunbin Lin   Yannis Papakonstantinou   Steven Swanson
Department of Computer Science and Engineering, University of California, San Diego
jil261@ucsd.edu   {h1tseng, chunbinlin, yannis, swanson}@cs.ucsd.edu

ABSTRACT

As data sets grow and conventional processor performance scaling slows, data analytics move towards heterogeneous architectures that incorporate hardware accelerators (notably GPUs) to continue scaling performance. However, existing GPU-based databases fail to deal with big data applications efficiently: their execution model suffers from scalability limitations on GPUs whose memory capacity is limited; existing systems fail to consider the discrepancy between fast GPUs and slow storage, which can counteract the benefit of GPU accelerators.

In this paper, we propose HippogriffDB, an efficient, scalable GPU-accelerated OLAP system. It tackles the bandwidth discrepancy using compression and an optimized data transfer path. HippogriffDB stores tables in a compressed format and uses the GPU for decompression, trading GPU cycles for the improved I/O bandwidth. To improve the data transfer efficiency, HippogriffDB introduces a peer-to-peer, multi-threaded data transfer mechanism, directly transferring data from the SSD to the GPU. HippogriffDB adopts a query-over-block execution model that provides scalability using a stream-based approach. The model improves kernel efficiency with the operator fusion and double buffering mechanism.

We have implemented HippogriffDB using an NVMe SSD, which talks directly to a commercial GPU. Results on two popular benchmarks demonstrate its scalability and efficiency. HippogriffDB outperforms existing GPU-based databases (YDB) and in-memory data analytics (MonetDB) by 1-2 orders of magnitude.

1. INTRODUCTION

As power scaling trends prevent CPUs from providing scalable performance [11, 13, 16], database designers are looking to alternate computing devices for large scale data analytics, as opposed to conventional CPU-centric approaches. Among them, Graphics Processing Units (GPUs) attract the most discussion for its massive parallelism, commercial availability, and full-blown programmability. Previous work [10,20,38,40] proved the feasibility of accelerating databases using GPUs. Experiments show that GPUs can accelerate analytical queries by up to $27\times$ [15,17,40].

However, existing GPU-accelerated database systems suffer from size limitations: they require the working set to fit in GPU's device memory. With this limitation, existing GPUs cannot handle terabyte-scale databases that are becoming common [8,21].

Scaling up GPU-accelerated database systems to accommodate data sets larger than GPU memory capacity is challenging:

1. **The low bandwidth of data transfer in heterogenous systems counteracts the benefit GPU accelerators provide.** While the main memory has always been fast (up to 8 GB/sec when transferring to a K20 GPU) and new storage devices like solid state drives (SSDs) are improving performance (up to 2.4 GB/sec [1]), the bandwidth demand of GPU database operators is still higher than the interconnect bandwidth and the storage bandwidth. As shown in Table 1, typical database operators and queries can run $29-82\times$ faster than the SSD read bandwidth. Without careful design, the slow storage would under-utilize high-performance GPU accelerators.
2. **Moving data between storage devices and multiple computing devices adds overhead.** The data transfer mechanism in existing systems can be both slow and costly. It single-threadedly moves data from the data source to the GPU via CPU and the main memory, failing to utilize the internal parallelism inside modern SSDs. This also adds indirections and consumes precious CPU and memory resources, which the system could use for other tasks. Recent work [38] shows that this detour can take over 80% of the execution time in typical analytical workloads and can cause the transfer bandwidth to be less than 40% of the theoretical peak.
3. **Current execution models of GPU-databases do not suit the architecture of GPUs and cause scalability and performance issues.** The query execution models in existing GPU databases [20,39] are neither efficient nor scalable. They require that the working set fit in the small GPU device memory (usually less than 20 GB [2]). Besides, the intermediate results that the current models produce put pressure on the already scarce GPU memory.

To address the above challenges, we propose HippogriffDB, an efficient, scalable heterogeneous data analytics engine. The primary issue HippogriffDB tackles is the low performance caused by the bandwidth mismatch between fast computation and slow I/O. HippogriffDB fixes it with compression and optimized data transfer mechanisms. The stream-based execution model it adopts makes HippogriffDB the first GPU-based database system that supports big data cube queries. HippogriffDB uses an *operator fusion*

[*]The work was supported by NSF SHF 1219125 and NSF BIGDATA 1447943.

[†]Hung-Wei Tseng is now an assistant professor in the Department of Computer Science, North Carolina State University. His current email is hungwei_tseng@ncsu.edu.

Proceedings of the VLDB Endowment, Vol. 9, No. 14
Copyright 2016 VLDB Endowment 2150-8097/16/10.

Operation	Throughput	Description
SSBM-Q1.1	61.8 GB/s	Q1.1 in the Star Schema Benchmark [27]. The query includes three selections, one join and one aggregation.
SSBM-Q4.1	31.2 GB/s	Q4.1 in the Star Schema Benchmark. The query includes three selections, four join and one aggregation.
Table Join	21.8 GB/s	A 100 MB table joins a 1 GB table using hash join. Both tables contain two columns and the two columns are 4-byte integers.
SSD Read in YDB [38]	0.75 GB/s	Sequentially read data from an NVMe SSD with 32 MB as the I/O size. We adopt a similar way used in [38].

Table 1: The throughput of running essential database operations/queries on a GPU and transferring data to the GPU. There's a big bandwidth mismatch between GPU processing and data transfer.

mechanism to aggressively eliminate the intermediate results since the penalty of cache misses is costly on the GPU.

HippogriffDB stores the data in a compressed format and decompresses them on the GPU, trading GPU computation cycles for the improved bandwidth. It utilizes the massive computation power of the GPU to decompress data, turning the bandwidth gap into the improved bandwidth. HippogriffDB tailors the compression methods to fit better into the GPU architecture. It supports combination of multiple compression methods to boost the effective bandwidth for data transfers. HippogriffDB employs a decision model to select the appropriate compression combination that balances the GPU kernel throughput and the I/O bandwidth. We prove the *optimal compression selection* to be an NP-hard problem and propose a 2-approximation greedy algorithm for it. For storage with massive capacity, HippogriffDB adopts adaptive compression: it maintains multiple compressed versions and then chooses the best compression scheme dynamically so that different queries can benefit from different compression schemes.

Furthermore, HippogriffDB tackles the low data transfer bandwidth by using a multi-threaded, peer-to-peer communication mechanism (Hippogriff) between the data source (e.g., SSD and NIC) and the GPU. The goal is to solve two problems in the I/O mechanism of the existing GPU-based analytical engines [38]: (1) data transfer relies on CPU/memory to forward the input (2) single-threading under-utilizes the multiple data transfer hardware components inside the SSD. To address these two problems, HippogriffDB reengineers the software stack so that the database can directly transfer data from the SSD to the GPU. Furthermore, Hippogriff introduces multi-threaded data fetching to take advantage of the massive parallelism inside modern SSDs.

HippogriffDB achieves high scalability, high kernel efficiency and low memory footprint by adopting a new execution strategy, called *query-over-block*. It contains two parts. First, HippogriffDB streams input in small blocks, leading to high scalability and low memory footprint. HippogriffDB adopts double buffering to support asynchronous data transfer. Second, the query-over-block model reduces the intermediate results using an *operator fusion* mechanism: it fuses multiple operators into one, turning the intermediate results passing into the local variables passing inside each GPU thread.

We implemented HippogriffDB on a heterogeneous computer system with an NVIDIA K20 and a high-speed NVMe SSD. As an initial look of HippogriffDB design, we focus on star schema queries. We compare it with a state-of-the-art CPU-based analytical database (MonetDB [9]) and a state-of-the-art GPU-based

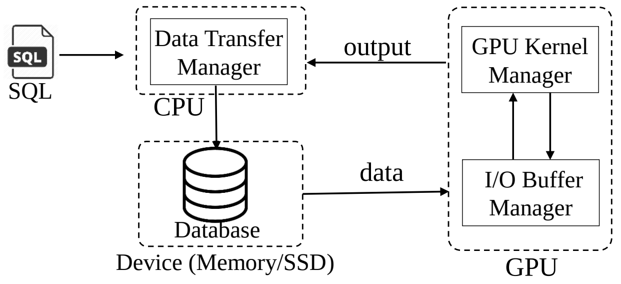


Figure 1: System architecture of HippogriffDB.

analytical database (YDB [40]), using two popular benchmarks (the Star Schema Benchmark [27] and the Berkeley Big Data Benchmark [30]). HippogriffDB outperforms MonetDB by up to 147$\times$ and YDB by up to 10$\times$. Results also show that HippogriffDB can scale up to support terabyte-scale databases and our optimizations can help achieve up to 8$\times$ performance improvement overall.

HippogriffDB makes the following contributions:

1. HippogriffDB improves the performance of GPU-based data analytics by fixing the bandwidth mismatch between the fast GPU and slow I/O, using adaptive compression.
2. HippogriffDB improves the data transfer bandwidth and resource utilization by implementing a peer-to-peer datapath that eliminates redundant data movements in heterogeneous computing systems.
3. We identify the problem of optimal compression selection to be an NP-hard problem and provide a 2-approximation greedy algorithm for it.
4. HippogriffDB uses the operator fusion mechanism to avoid intermediate results and to improve kernel efficiency.
5. HippogriffDB uses query-over-block, a streaming execution model, to provide native support for big data analytics.
6. HippogriffDB outperforms state-of-the-art data analytics systems by 1-2 orders of magnitude, and experiment results demonstrate HippogriffDB's scalability.

The paper provides an overview of HippogriffDB in Section 2. We discuss compression and the optimized data transfer in Section 3 and 4 . Section 5 discusses the execution model in HippogriffDB. Section 6 and 7 evaluate the system. We compare HippogriffDB with related work in Section 8 and Section 9 concludes.

2. SYSTEM OVERVIEW

HippogriffDB uses data compression and optimized data movement, combined with a stream-based query execution model, to deliver an efficient, scalable system. This section provides an overview of the system design, the data compression mechanism, the optimized transfer mechanism, and the query-over-block model to support scalable big data analytics.

2.1 System architecture

HippogriffDB targets large databases and stores database tables in the main memory or the SSD in the current implementation. It contains three major components.

Data Transfer Manager. The data transfer manager moves the requested data from the main memory/SSD to the GPU kernel. It uses a multi-threaded, peer-to-peer communication mechanism between the GPU and the SSD to further improve the data transfer bandwidth.

I/O Buffer Manager. HippogriffDB maintains a circular input buffer in the GPU memory. It works with the *Data Transfer Manager* to overlap data transfer and query processing: an I/O thread and the

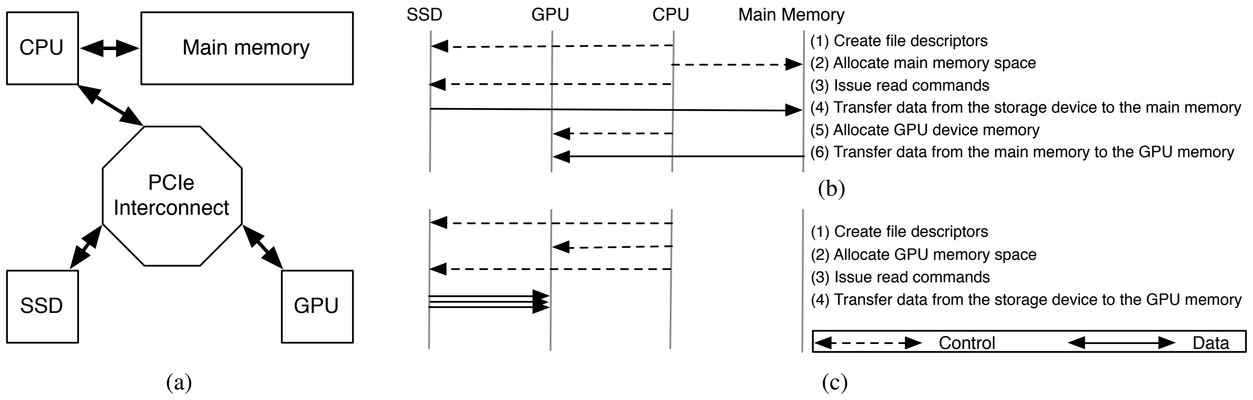

CPU
Main memory
PCIe Interconnect
SSD
GPU
(a)
SSD
GPU
CPU
Main Memory
(1) Create file descriptors
(2) Allocate main memory space
(3) Issue read commands
(4) Transfer data from the storage device to the main memory
(5) Allocate GPU device memory
(6) Transfer data from the main memory to the GPU memory
(b)
(1) Create file descriptors
(2) Allocate GPU memory space
(3) Issue read commands
(4) Transfer data from the storage device to the GPU memory
Control
Data
(c)

Figure 2: **(a) The conventional heterogenous platform, (b) the process of moving data between the GPU and the SSD in existing systems, and (c) direct data access in HippogriffDB.**

GPU kernel act as the producer and consumer respectively to copy data from storage to the GPU. HippogriffDB also maintains a result buffer in GPU's memory, if the result can fit in the GPU memory.

GPU Kernel Manager. The GPU kernel manager evaluates queries on the received data. HippogriffDB supports queries that contain selection, join, aggregation, and sort operators.

Figure 1 shows how HippogriffDB processes a query. The *Data Transfer Manager* prepares the relevant columns for a given query by retrieving data from either the main memory or the SSD. It then works with the *I/O Buffer Manager* to send relevant columns from the main memory/SSD to the input buffer in the GPU memory. The *GPU Kernel Manager* then evaluates the query. When query evaluation finishes, the *GPU Kernel Manager* will send the result back to the output buffer.

There is a huge bandwidth mismatch between the GPU kernel and I/O. Without careful design, the slow I/O transfer will counteract the speedup that hardware accelerators provide. HippogriffDB alleviates the mismatch by storing data in a compressed form and using GPU cycles for decompression. To further improve physical I/O bandwidth, HippogriffDB removes the redundant data transfers by implementing a peer-to-peer communication from storage to the GPU. We provide an overview for them in Section 2.2.

We also notice the inefficiency and the scalability limitation of current query execution models. HippogriffDB fixes it by introducing a new query model in GPU processing. We provide an overview of it in Section 2.3.

2.2 Optimizing data movement

The primary obstacle for building an efficient, GPU-based, big data analytics system is that, in modern systems, GPUs can process data 6-12× faster than a typical storage system can provide it. HippogriffDB addresses this imbalance by exchanging a plentiful resource (GPU compute cycles) for a scarce resource (effective data transfer bandwidth). It achieves this using two techniques:

1. It adopts a multi-threaded, peer-to-peer communication mechanism (Hippogriff) over the PCIe interconnect to move data directly from the storage system (i.e., an SSD) to the GPU without shuttling data through main memory.
2. It stores tables in compressed form and uses the GPU to decompress them, effectively converting GPU compute capabilities into effective data transfer bandwidth.

2.2.1 *Hippogriff*

HippogriffDB employs Hippogriff, a PCIe data transfer scheduler that enhances bandwidth by using multi-threaded and peer-to-peer data transfer mechanisms. This is important since in GPU-based database systems, the bandwidth of moving data between the storage device and the GPU is the main performance bottleneck. As shown in Table 1, typical database operators and queries can run $29 - 82\times$ faster than the SSD read bandwidth.

Existing high-performance heterogenous systems use NVMe-based SSDs to store data, but the NVMe standard is inherently inefficient for moving data between the SSD and the GPU. Figure 2(b) illustrates the problem: the system first moves data from the SSD to main memory (step 1-4), and then copies them to the GPU (step 5-6). Prior work [37] indicates that some applications spend more than 80% of their time copying data from main memory to the GPU. In addition to wasting bandwidth, this approach also wastes memory capacity and CPU performance, both of which could be put to more productive uses.

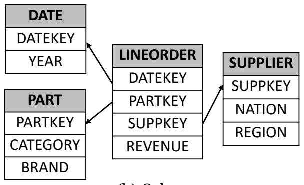
Furthermore, the Linux NVMe driver does not fully utilize the parallelism that SSDs offer, since it is single threaded. This has large negative impacts on performance: Our experiments in Section 7.1 show that a SSD-based version of YDB [37] can only achieve 30% of the peak performance if we store data in SSDs. Hippogriff addresses these problems. Hippogriff provides multi-threaded, peer-to-peer data movement between SSDs and GPUs. As Figure 2(c) shows, Hippogriff only needs to obtain the file information and permission from the CPU program in Step (1). After the system allocates space in the GPU's memory (Step (2)), Hippogriff issues NVMe commands with GPU memory addresses as the sources or destinations in Step (3) and allows the data to flow directly between the SSD and the GPU without using main memory (Step (4)). Hippogriff further exploits parallelism by creating multiple threads to utilize idle NVMe queues from all processors in the system.

Hippogriff implements these features by fully leveraging the peer-to-peer features that PCIe provides and combining them with intelligent scheduling of IO transfers. PCIe supports direct transfer of data between PCIe devices, as long as both devices support it – all that is required is that 1) the destination device expose a portion of its on-board memory in the PCIe address space and 2) the source device driver directs the source device to transfer data to that address via direct memory access (DMA). GPUs and network interface cards support these transfers via the GPUDirect [3] mechanism. Hippogriff extends this capability to SSDs.

Hippogriff can also leverage the conventional data transfer mechanism in which data flows from one device, to main memory, and then out to another device. While this path is less efficient, it can improve performance if the resources required for peer-to-peer transfer are occupied or unavailable. Depending on the set of pending transfers, Hippogriff dynamically chooses which data movement channel or channels to use.

```
SELECT SUM(lo.revenue), p.brand1
FROM lineorder lo, part p
WHERE lo.partkey = p.partkey
        AND lo.quantity < 25
        AND p.category = 'MFGR#12'
GROUP BY p.brand1
```

(a) Query 1



(b) Schema

Figure 3: Query 1 and its corresponding schema.

2.2.2 *Column-based, compressed tables*

To surpass the physical bandwidth limit that the interconnect and the devices set, HippogriffDB stores database tables using a column-based, compressed format and trades GPU decompression cycles for improved effective bandwidth.

HippogriffDB follows the modern column store designs by using implicit virtual-ids [6], as opposed to explicit record-ids, to avoid bloating the size of data storage. Column-based format provides more opportunity for compression, which further improves the I/O bandwidth [6, 9].

HippogriffDB stores tables in a compressed format and uses GPU idle cycles for decompression to further improve the effective I/O bandwidth. HippogriffDB allows both light-weighted compression and heavy-weighted compression methods.

There is a tradeoff between the compression aggressiveness and the GPU efficiency. Aggressive compressions can help reach better compression ratio but also bring more cost to the GPU decompression and slow down the GPU accelerators [6]. HippogriffDB adopts a cost-benefit model to evaluate the trade-off and to choose appropriate strategies. We identify the optimal compression selection problem to be an NP-hard problem and propose a 2-approximation greedy algorithm.

HippogriffDB observes the limitation of maintaining only one compression plan and hence adopts an adaptive compression strategy, when possible. One compression plan may benefit certain kinds of queries but works poorly on others. HippogriffDB fixes this issue by keeping multiple compression schemas and choosing the optimal one dynamically.

2.3 Query-Over-Block overview

The query-over-block execution model enables the system to efficiently scale beyond the GPU memory capacity. It contains two aspects: first, it processes inputs as streams and uses double buffering to support asynchronous execution (block-oriented execution); second, it packs multiple operators into one and sends intermediate results via thread-local variables (operator fusion mechanism).

Example to demonstrate existing models and query-over-block. Query 1 (Figure 3(a)) compares the revenue for some products that certain manufacturer makes and whose quantity is less than 25, grouped by the product brands (Figure 3(b) shows the database schema). We show the query execution plan `YDB` [40] generates in Figure 4. The model that YDB uses , “operator at a time and bulk execution” [9], evaluates each operation (e.g. selection of `quantity` < 25) to completion over its entire input (e.g. `lo_quantity`) and sends the whole intermediate results to the upcoming operator (e.g. `lineorder` $\bowtie$ `part`).

The query-over-block model generates three operators for Query 1, as shown in the dashed, rectangular box in Figure 4. Two of them are the new selection operators whose functionalities include traditional selection (e.g. `quantity` < 25), projection. The other one is the new join operator which covers the functionalities of the join in the original plan (e.g. `lineorder` $\bowtie$ `part`) and the aggregation operation (e.g. $\gamma_{\texttt{p.brand1,sum(lo.revenue)}}$).

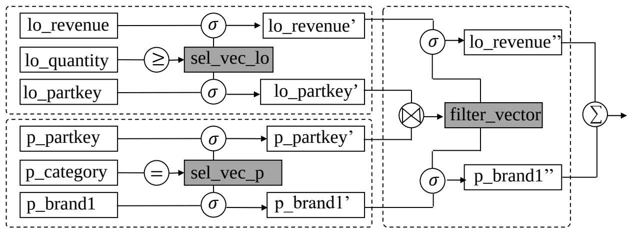


Figure 4: Schema of Query 1 generated by YDB. It creates large intermediate results (grey boxes). It also requires all relevant columns in GPU memory, limiting the scalability of the system.

Comparison of the existing model and query-over-block. YDB’s query plan is neither scalable nor efficient for two reasons. First, this plan generates large intermediate results (i.e. `filter_vector`), which cost precious memory resources and reduce the query processing efficiency, since accessing global memory on the GPU is slow. Second, it requires that all relevant database columns fit into the GPU memory [40], which limits the system scalability.

The query-over-block model addresses the limitations by fusing multiple operations into one and streaming the input. Combining multiple operators allows the system to avoid materialization of intermediate results and pass the data more efficiently using fast local variable. The query-over-block model can process input blocks in a streaming approach. In this way, this model supports data sets larger than the GPU memory capacity.

3. DATA COMPRESSION

The massive parallelism inside the GPU makes the GPU computation throughput much higher than the data transfer bandwidth that main memory or SSDs can deliver (up to $12\times$ for the main memory and $30\times$ for the SSD), creating an imbalance among different components inside GPU-based analytics systems. HippogriffDB narrows the gap using compression: it compresses database tables and uses the GPU to decompress them, effectively converting GPU’s compute capabilities into the improved transfer bandwidth. In this way, HippogriffDB improves the the system throughput.

We first introduce a compression strategy that minimizes the overall space cost. Then we show this kind of aggressive compression strategy may not be GPU-friendly and provide an improved compression strategy that better suits the GPU environment.

3.1 Minimizing space cost (MSC)

HippogriffDB employs *run length encoding (RLE) [34]*, *dictionary encoding (DICT)*, *huffman encoding (Huffman)*, and *delta encoding (DELTA)* to compress tables. When compressed data is sent to GPU, we decompress the data by using the conventional method introduced in [14]. Notice that HippogriffDB can evaluate the columns encoded by *RLE* directly without the decompression cost as is mentioned in [33].

HippogriffDB compresses tables in a heuristic strategy: it first sorts the table using two sort keys (one primary and one secondary sort key). Notice, how to choose primary and secondary sort key columns will be discussed later. It applies *RLE* on the primary sort key column and *DELTA* on the secondary sort key column. For other columns, it applies *DICT* if possible. The reason behind this heuristic strategy is discussed as follows. To work with the streaming-based execution model, we only allow one column to be sorted, and for it, we apply *RLE* as it leads to very high compression ratio. As we are not able to introduce another primary sort key, we decide to choose a secondary sort key. For the column chosen as the secondary sort key, we take advantage of the orderliness and use the delta between two consecutive elements to encode the column. For other columns, we evaluate the domain size and the distribution

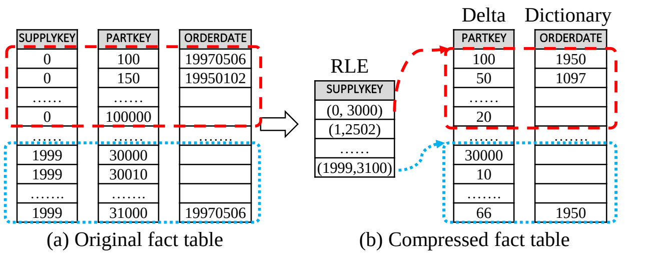


Figure 5: Example of a compression plan. *RLE* applies to the primary sort key column (`SUPPLEYKEY`) and *DELTA* applies to secondary sort key column (`PARTKEY`). The `ORDERDATE` column uses *DICT* due to its limited number of distinct values.

of the column and then choose adequate compression methods for them.

For example, Figure 5 shows how HippogriffDB compresses fact table by using the method discussed above. `SUPPLYKEY` and `PARTKEY` work as the primary and secondary sort key respectively. HippogriffDB encodes them using *RLE* and *DELTA* respectively. HippogriffDB evaluates the domain size and distribution and uses *DICT* to encode `ORDERDATE`.

To achieve the minimal space cost, HippogriffDB enumerates all primary-secondary sort key combinations over the columns to find out the plan that comes with the minimal space cost, as shown in the MSC (Minimize Space Cost) algorithm (Figure 6). HippogriffDB explores all possible primary-secondary sort key combinations (Lines 6 - 7), encodes them using *RLE* and *DELTA* and calculates the compression ratio of them (Lines 8 - 9). It then calculates the overall compression ratio (Line 10) and updates the current plan if its compression ratio is better than all previous plans (Line 11 - 15).

3.2 GPU-friendly compression plans

The MSC algorithm in Section 3.1 generates a compression plan that comes with the minimal space cost, however, the plan generated may not be GPU-friendly and may result in suboptimal system throughput. Below, we first illustrate the potential problems of the MSC algorithm and then discuss an algorithm to generate compression plans that optimize for the entire system throughput.

Using aggressive compressions to achieve minimal space cost may result in two problems. First, the decompression task may overburden the GPU, creating a new form of imbalance and impairing the system throughput. Second, intensive decompression operations on the GPU would degrade the performance of Hippogriff, deviating from our original goal of improving data transfer bandwidth.

To avoid the problems discussed above and make compression plans GPU-friendly, we require the decompression process do not overburden the GPU. We formulate a cost-benefit analysis below. The cost in this case is the GPU decompression cycles and the benefit is reducing the amount of data transfer.

Let T_G denote the GPU kernel time to run the queries and r_x denote the compression ratio of the compression method used on column x (assume we have n columns). The corresponding column size is C_x and D_x denotes the decompression bandwidth of it. Suppose the data transfer rate is B_{IO}. The time to transfer the compressed data is:$(\sum_{i=1}^{n} C_i * r_i)/B_{IO}$ and the time to decompress them and run the query is: $T_G + \sum_{i=1}^{n}(C_i * r_i)/D_i$. To prevent the GPU from running slower than I/O, we require that: $(\sum_{i=1}^{n} C_i * r_i)/B_{IO} >= T_G + \sum_{i=1}^{n}(C_i * r_i)/D_i$ for a compression plan to be GPU-friendly.

Algorithm: MSC and GFC Algorithms

```
    Input: A fact table with $n$ columns, $\mathcal{C}$={$\mathcal{C}_1, ..., \mathcal{C}_n$}
    Output: A compression strategy $\mathcal{M} = \{\mathcal{M}_1, ..., \mathcal{M}_n\}$, where
            $\mathcal{M}_i$ is the compression method for $\mathcal{C}_i$.
 1  $\mathcal{M} \leftarrow \varnothing$ ; //$\mathcal{M}$ stores the optimal plan
 2  $\mathcal{M}_B \leftarrow \varnothing$ ; //$\mathcal{M}_B$ stores the temporary balance plan
 3  $min\_ratio \leftarrow 1$ ;
 4  for $i \leftarrow 1$ to $n$ do
 5      $d_i \leftarrow dict\_cmp\_ratio\ (\mathcal{C}_i)$ ; //compute dictionary encoding ratio
 6  for $i \leftarrow 1$ to $n$ do  //$\mathcal{C}_i$ as the primary sort key
 7      for $j \leftarrow i+1$ to $n$ do  //$\mathcal{C}_j$ as the secondary sort key
 8          $r_i \leftarrow rle\_cmp\_ratio(\mathcal{C}_i)$ ; //compute RLE encoding ratio
 9          $r_j \leftarrow delta\_cmp\_ratio(\mathcal{C}_i, \mathcal{C}_j)$ ; //compute DELTA ratio
10          $current\_ratio \leftarrow compute\_ratio(r_i, r_j, d_1, ..., d_n)$;
11          if $min\_ratio > current\_ratio$ then
12              $min\_ratio \leftarrow current\_ratio$;
13              $\mathcal{M}[i] \leftarrow RLE$; $\mathcal{M}[j] \leftarrow DELTA$;
14              foreach $k \in \{1, 2...n\} \setminus \{i, j\}$ do
15                  $\mathcal{M}[k] \leftarrow DICT$;
16          $\mathcal{M}[i] \leftarrow RLE$; $\mathcal{M}[j] \leftarrow DELTA$;
17          foreach $k \in \{1, 2...n\} \setminus \{i, j\}$ do
18              $\mathcal{M}[k] \leftarrow DICT$; //encode other columns with DICT
19              $r_k \leftarrow d_k$; //assign the compression ratio
20          $\mathcal{M}_B \leftarrow$ balance_cmp($(\mathcal{C}, \mathcal{M}), \nabla = \{r_k\}$);
21          $current\_ratio \leftarrow compute\_ratio(\mathcal{M}_B)$;
            //compute the compression ratio of compression plan $\mathcal{M}_B$
22          if $min\_ratio > current\_ratio$ then
23              $min\_ratio \leftarrow current\_ratio$;
24              $\mathcal{M} \leftarrow \mathcal{M}_B$ ;
25  return $\mathcal{M}$ ;

    Function balance_cmp($(\mathcal{C}, \mathcal{M}), \nabla$ );
    //Let $D_i$ be the decompression bandwidth of $C_i$
    //Let $B_{IO}$ be the data transfer rate
    //Let $T_G$ be the GPU kernel processing time
26  sort($(\mathcal{C}, \mathcal{M})$, $(sizeof(\mathcal{C}_i) * r_i)/D_i$);
    //sort the columns in non-decreasing order of $(C_i * r_i)/D_i$
27  $\mathcal{M}_B \leftarrow \varnothing$ ;
28  for $i \leftarrow 1$ to $n$ do
29      $new\_io \leftarrow \sum_{j=1}^{i} C_i * r_i/B_{IO} + \sum_{j=i+1}^{n} C_i/B_{IO}$ ;
30      $new\_gpu \leftarrow T_G + \sum_{i=1}^{i} C_i/D_i$ ;
31      if $new\_io < new\_gpu$ then
32          break ;
33      $\mathcal{M}_B[i] \leftarrow \mathcal{M}[i]$;
34  return $\mathcal{M}_B$ ;
```

Figure 6: MSC and GFC algorithms. Codes in the solid box are MSC only, while those in the dotted box are GFC only.

We define the *optimal compression selection* problem as finding a plan which can minimize the data transfer time while maintaining the GPU-friendliness:

$$\min \quad \sum_{i=1}^{n} C_i * r_i/B_{IO}$$
$$\text{s.t.} \quad \sum_{i=1}^{n} C_i * r_i/B_{IO} >= T_G + \sum_{i=1}^{n}(C_i * r_i)/D_i$$

The problem of selecting the optimal compression combination is proved to be an **NP-hard** problem by a reduction from the 0-1 Knapsack problem [25]. Similar to the greedy algorithm for the 0-1 Knapsack problem [25], we propose a **2-approximation** greedy algorithm, which has two steps. First, the algorithm sorts the columns in non-decreasing order of $(C_i * r_i)/D_i$, as shown in Line 26 in the GFC algorithm (Figure 6). Second, it greedily picks columns in the above order (Line 28 - 33). Due to space limitations, we omit the proof of the NP-hardness and the 2-approximation.

In the GFC algorithm, we integrate the GPU-friendliness requirement when generating the compression plans. Given a compression plan generated by MSC (Line 6 - 9, 16 - 19), Line 20 calls the greedy algorithm to convert it into an optimal (approximately and locally[1]) GPU-friendly compression plan. Line 21 - 24 compare all locally-optimal compression plans and choose the globally optimal one to return.

4. SSD-SPECIFIC OPTIMIZATIONS

HippogriffDB improves the data transfer from the slow SSDs by allowing direct data transfer from the SSD to the GPU and using a query-adaptive compression to further improve compression ratio, trading storage spaces for the improved system throughput.

In the following sections, we first introduce the new data path, Hippogriff. We then illustrate the inefficiency of the fixed compression strategy and based on such observation, introduce an "adaptive" compression strategy to extend the benefit of compression to a wide-range of queries.

4.1 Hippogriff

HippogriffDB relies on three components to provide the multi-threaded, peer-to-peer data transfer:

1. Hippogriff API: HippogriffDB provides APIs for programmers to specify the data transfer sources and destinations.
2. Hippogriff runtime system: it maintains the runtime information from all processes that use Hippogriff.
3. Hippogriff: HippogriffDB uses Hippogriff to perform peer-to-peer transfers between the SSD and the GPU.

Compared with existing systems, HippogriffDB improves the I/O bandwidth in two aspects:

HippogriffDB implements a peer-to-peer data transfer path between the SSD and the GPU by re-engineering the software stack of NVMe SSDs as in [4,24,35]. When the storage system receives a request with a GPU device address as the source or destination, the NVMe software stack leverages NVIDIA's GPUDirect [3] to make the source or destination GPU device memory address visible for other PCIe devices (by programming the PCIe base address registers). Upon the success of exposing device memory to PCIe interconnect, our NVMe software stack issues NVMe read to the SSD using these GPU addresses as the DMA addresses instead of main memory addresses. The SSD then directly pulls or pushes data from or to the GPU device memory, without further interventions from the CPU and the main memory.

HippogriffDB uses multi-threaded data transfer. It invokes multiple threads (4 threads in the current design) to read data from SSD. To provide fair sharing among processors, the NVMe SSD periodically polls the software-maintained NVMe command queue for each processor. As a result, the SSD can under-utilize both internal access and outgoing bandwidth if only one or two processes are issuing commands to the SSD. HippogriffDB fixes this problem by querying the occupancy of the SSD NVMe command queues. If the queues are nearly empty, it boosts performance by running multiple peer-to-peer transfers in parallel to improve bandwidth.

[1]Given the certain primary-secondary sort key combination, $<\mathcal{C}_i, \mathcal{C}_j>$ in the algorithm, it is the optimal plan.

4.2 Adaptive compression

HippogriffDB uses adaptive compressions to further improve compression ratio for databases on secondary storage. The GFC algorithm in Section 3 aims to minimize the table size while maintaining GPU-friendliness, however, it could work poorly on some queries. In this section, we first show the inefficiency of the fixed compression scheme and then demonstrate how HippogriffDB fixes the problem with the adaptive approach.

QUERY 2

```
SELECT SUM(lo.revenue), d.year, p.brand1
FROM lineorder lo, date d, part p, supplier s
WHERE lo.orderdate = d.datekey AND lo.partkey = p.partkey
      AND lo.suppkey = s.suppkey  AND lo.quantity < 25
      AND p.category = 'MFGR#12'  AND s.region = 'AMERICA'
GROUP BY d.year, p.brand1
```

The best compression plan for one query can work poorly on other queries. For Query 2, the best compression plan is as follow: *RLE* on `lo_partkey`, *DELTA* on `lo_supplykey`, *DICT* on `lo_orderdate` and `lo_revenue`. However, this plan works poorly on the query below, as one column in it is not compressed and compression ratio of other columns is not as good as it could be.

```
SELECT c.nation, s.nation, d.year, SUM(lo.revenue) as revenue
FROM customer c, lineorder lo, supplier s, date d
WHERE lo.custkey = c.custkey
        AND lo.suppkey = s.suppkey
        AND lo.orderdate = d.datekey
        AND c.region = 'ASIA'
        AND s.region = 'ASIA'
        AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
```

To be adaptive to different queries, HippogriffDB uses the adaptive compression mechanism by allowing a table to keep multiple compressed versions. For the example above, instead of having a version that applies *RLE* on `lo_partkey`, *DELTA* on `lo_supplykey`, *DICT* on `lo_orderdate` and `revenue`, the system may also maintain another version which applies *RLE* on `lo_supplykey`, *DELTA* on `lo_custkey`, *DICT* on `lo_orderdate` and `lo_revenue`. HippogriffDB may maintain other compression versions as well.

During the query processing, HippogriffDB examines all available compression plans and then calculates the overall compression ratio for each of them. It will then adopt the one with the best compression ratio for the query and send it to the GPU.

Suppose the number of foreign-key columns (on which *Huffman* and *DICT* work poorly) is n, the adaptive strategy will produce at most $n(n-1)$ different compression plan. Each compression plan will create a compressed version of the table, which can be a big space overhead. The following theorem reduces the space cost by half, without significant performance degradation.

THEOREM 1. *Given two compression plan A and B, where the only difference between them is that they switch the primary and secondary sort key, the compression ratio difference is asymptotically 0 (Assume that $P = o(N)$, where P is the cardinality of the primary sort key and N is the number of rows in the fact table).*

For databases with large number of foreign key columns, we trade slight compression degradation for big space efficiency improvement. Instead of enumerating all primary-secondary sort key combinations, we only enumerate the primary sort key column. Hence, we reduce the number of different compressed tables from $\binom{n}{2}$ to n with minor cost (we can still use other compression methods for the column that is originally encoded using *DELTA*). For example, when running Q2.2 int the Star Schema Benchmark (S-F=10), the overall compression ratio increased from 0.28 to 0.32 and the entire system system throughput only drops by 13%.

In practice, database users may have additional knowledge about the database (e.g., query logs) that can help reduce the space overhead even further.

Remark In this section, we discuss the optimizations for the secondary storage. Those optimizations also work for other storages as well. For example, the adaptive compression is applicable in a distributed topology, as the space is abundant in such architecture. [5] also allows a direct data transfer from NICs to the GPU, bypassing the CPU/memory overhead for the distributed databases.

5. QUERY-OVER-BLOCK MODEL

The query-over-block model handles data sets larger than the GPU memory. It provides high scalability by processing input in a streaming manner. It also removes intermediate results and improves GPU kernel efficiency with the operator fusion mechanism.

In this section, we first introduce the data schemas that we focus on, then formally (re)define the queries running on these schemas. Based on the definition, we introduce three physical operators and then show how HippogriffDB uses them to optimize the query plan.

5.1 Schema and query definition

HippogriffDB targets data warehouse applications. Data warehouses typically organize data into $multidimensional\ cubes$ (or $hypercubes$) and map hypercubes into relational databases using the star schema or the snowflake schema. In the star schema, a central table contains fact data and multiple dimension tables radiate out from it. The fact table and the dimension tables connect through the primary/foreign key relationships. Existing comparison results show that star schema is prevalent in data warehouses [6, 23]. HippogriffDB focuses on star schema queries (SSQ). For other queries and schemas, HippogriffDB can work as an accelerator on SSQ subexpressions and leave the rest to classic methods.

The operators inside an SSQ fall into three categories. The first category is unary operations, such as *selection* and *projection*, on dimension tables. The second category includes unary operations on the fact table and *natural join* between the fact and the dimension tables. The last category is *aggregation* and *group by* on the join results. As an example, we categorize the operations in Query 2 into these three categories, as shown in Table 2.

Figure 7 provides the *Normalized Algebra (NA) expression* for SSQs. This attribute grammar has the ability to describe all SSQs. The root of the *NA* is either an aggregation of a join or just a join. The join here is either a series of joins over $F, D_1, \ldots D_n$ or F itself, where F is a fact table (or the result of unary operations on the fact table) and D_i is a dimension table (or the result of unary operations on the dimension table).

5.2 Query-over-block execution model

Query-over-block improves the scalability and efficiency using an "operator fusion" mechanism and a stream-based approach. In this subsection, we first introduce three new physical operators and then demonstrate how query-over-block generates query plans using these physical operators.

5.2.1 Operator fusion

We implement three physical operators using operator fusion mechanism to improve the kernel efficiency and to eliminate intermediate results. Below, we first define these physical operators and then discuss their implementations.

Physical operator definition

Based on the attribute grammar in Section 5.1, HippogriffDB introduces three corresponding physical operators: $\Lambda_{R,c,K,V}$, $\ltimes\!\!\rtimes_{R,c,L_1,\ldots,L_n}$, and $\Gamma_{J,f_1,\ldots,m}$. We define the three operators logically as follows:

1. $\Lambda_{R,c,K,V}$ outputs $\pi_{K,V}\sigma_c(R)$, a hashmap L with $\pi_K\sigma_c(R)$ as the keys and $\pi_V\sigma_c(R)$ as the values.

Category	Operations

Category 1	p.category = MFGR#12
	s.region = AMERICA'

Category 2	lo.orderdate = d.datekey
	lo.partkey = p.partkey
	lo.quantity $<$ 25

Category 3	SUM(lo.revenue)
	GROUP BY d.year, p.brand1

Table 2: Categorization of operations in Query 2.

2. $\ltimes\!\!\rtimes_{R,c,L_1,\ldots,L_n}$ outputs $\sigma_c(R) \bowtie L_1 \bowtie \ldots \bowtie L_n$.
3. $\gamma_{J,A,f_1,\ldots,m}$ outputs the results of $\gamma_{A,f_1,\ldots,m}(J)$.

Implementation We implement the physical operators as follows:

1. $\Lambda_{R,c,K,V}$. We implement the hashtable using Cuckoo Hashing [28]. Each GPU thread evaluates the selection condition on its input and, if the condition is met, inserts the input into the hash index. We use atomic instructions that CUDA provides to avoid conflicts in the parallel program.
2. $\ltimes\!\!\rtimes_{R,c,L_1,\ldots,L_n}$. We assign each GPU thread a row in the fact table. The GPU thread first evaluate selection conditions on the relation R and then probes the hash indices of $L_1, \ldots, L_n$.[2]
3. $\gamma_{J,A,f_1,\ldots,m}$. We use the hashmap or array (if we know the domain size in advance) for the aggregation. We use atomic instructions to resolve conflicts between different threads.

Operator fusion mechanism

HippogriffDB mollifies the memory contention that the intermediate results cause using the operator fusion mechanism. This mechanism combines multiple operators into a single GPU kernel and, in this way, turn intermediate results passing into local variables passing inside each GPU thread.

For example, for the physical operator $\ltimes\!\!\rtimes_{R,c,L_1,\ldots,L_n}$, we fuse all joins (hash joins) and selections on the fact table into one GPU kernel. We provide the operator fusion mechanism in Algorithm 1. The kernel first evaluates the selection operation on a given row (Line 7-9). If the row survives the selection conditions, it will proceed to join with other dimension tables (Line 10-12). The GPU kernel passes intermediate results using local variables inside the thread. The implementations of physical operator $\Lambda_{R,c,K,V}$, $\gamma_{J,f_1,\ldots,m}$ follow similar approaches.

We use Query 2 as an example to compare our query plan with the query plan that existing models generate. Figure 8 (a) presents the query plan that most existing GPU-based databases adopt. After joining $lineorder$ and $part$, the system sends the join results (intermediate results) to join another table, $supplier$, and then generates another set of intermediate results, and so forth. The existing query plan sends large intermediate results several times during the query execution, which is costly as accessing GPU global memory is slow. In addition, storing those intermediate results on a memory-scarce device hurts the scalability of the system. Figure 8(b) shows our approach. It packs the selections on the dimension tables and hash index building into the Λ operator. It fuses the three natural joins, the selection on table `lineorder`, and the aggregation into one GPU kernel. In this way, HippogriffDB avoids all intermediate results.

Discussion [7] uses invisible joins to reduce redundant data transfers. It first evaluates the invisible join and then, based on the join results, reads the other columns on demand. HippogriffDB does not adopt this approach, as the second step would involve large amount of random reads to the SSD, which is slow for a flash-based SSD (and also hard disks).

[2]We assume the hash indices can fit in the GPU memory. We discuss the memory requirement at the end of this section.

$$
\begin{array}{lcl}
NA & \Rightarrow & \gamma_{G;\, f_1(.)\mapsto N_1,\ldots,f_n(.)\mapsto Nn_\gamma}\, Join \\
 & | & Join \\
Join & \Rightarrow & F \bowtie D_1 \bowtie D_2 \ldots \bowtie D_n \\
 & | & F \\
F & \Rightarrow & \pi_{att}\, \sigma_c\, Fact \\
 & | & Fact \\
Di & \Rightarrow & \pi_{att}\, \sigma_c\, Dimension \\
 & | & Dimension
\end{array}
$$

Figure 7: An attribute grammar of SSQs

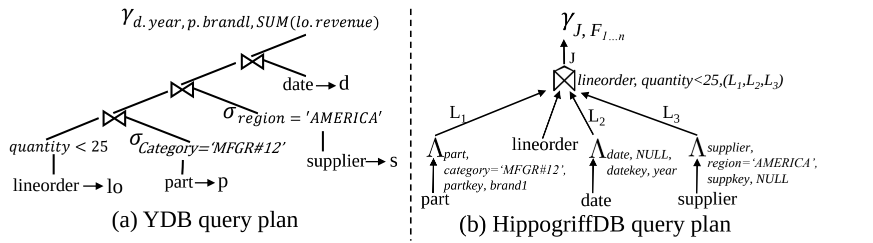

$\gamma_{d.year, p.brandl, SUM(lo.revenue)}$
date → d
$\sigma_{region = 'AMERICA'}$
quantity < 25
$\sigma_{Category='MFGR\#12'}$
supplier → s
lineorder → lo
part → p
(a) YDB query plan
$\gamma_{J, F_{1...n}}$
J
lineorder, quantity<25,(L1,L2,L3)
L1
L2
L3
lineorder
$\Lambda_{part,}$ category='MFGR#12', partkey, brand1
$\Lambda_{date, NULL,}$ datekey, year
$\Lambda_{supplier,}$ region='AMERICA', suppkey, NULL
part
date
supplier
(b) HippogriffDB query plan

Figure 8: The query plan for Query 2 by YDB and HippogriffDB. Query plan that HippogriffDB generates can avoid intermediate results and support dataset larger than GPU memory.

Algorithm 1: Operator fusion algorithm

```
   Input: A fact table $F$, dimension table indices
          $\mathcal{H}$={$\mathcal{H}_1, ..., \mathcal{H}_n$}, a list of selection conditions
          $\mathcal{C}$={$\mathcal{C}_1, ..., \mathcal{C}_m$}, join conditions $\mathcal{J}$={$J_1, ..., J_l$},
          groupby columns $\mathcal{G}$={$G_1, ..., G_k$}, aggregation
          function $\mathcal{A}$={$A_1, ..., A_p$}
   Output: Analytical results $\mathcal{R}$
 1 Func fused_kernel($F, \mathcal{H}, \mathcal{J}, \mathcal{C}, \mathcal{G}, \mathcal{A}$) ;
 2 $\mathcal{R} \leftarrow \varnothing$ ;
 3 cuda_fused_kernel<<<...>>>($\mathcal{R}, F, \mathcal{H}, \mathcal{J}, \mathcal{C}, \mathcal{G}, \mathcal{A}$);
 4 return $\mathcal{R}$ ;
 5 Func fused_kernel($\mathcal{R}, F, \mathcal{H}, \mathcal{J}, \mathcal{C}, \mathcal{G}, \mathcal{A}$) ;
 6 r $\leftarrow R[thread\_id]$ ;
 7 for $c \in \mathcal{C}$ do
 8     if $eval(c, r) == false$ then
 9         return false ;
10 for $j \in \mathcal{J}$ do
11     if $\mathcal{H}_j$.find ($P_{r,j}$) == NULL then
12         return false ;
13     evalAggr($\mathcal{R}$, $P_{r,\mathcal{G}}$ , $\mathcal{A}$ ) ;
```

5.2.2 *Block-oriented execution plan*

HippogriffDB uses a block execution plan to improve the system scalability. It streams the fact table to support data sets larger than the GPU memory. HippogriffDB adopts double buffering to support asynchronous data transfer, which allows the overlapping between the kernel execution and the data transfer.

HippogriffDB generates the physical query plan for a given query in three phases. It first pushes down unary operators (category 1) on the dimension tables and builds in-GPU-memory hash indices for them (physical operator $\Lambda_{R,c,K,V}$). In the second phase, it evaluates natural joins (category 2) using the hash indices built in the previous stage (physical operator $\boxtimes_{R,c,L_1,...,L_n}$). The third stage evaluates aggregations on the join results (physical operator $\Gamma_{J,f_{1,...,m}}$).

HippogriffDB adopts a circular input buffer to enable asynchronous data transfer for efficient streaming. The data transfer manager continues to transfer data while the kernel manager evaluates the received input.

Memory requirement In the current implementation, HippogriffDB requires the hash indices of the dimension tables fit in the GPU memory. We also maintain the input and output buffer in the GPU memory. Hence, the memory requirement is $S_{\texttt{input}} + S_{\texttt{output}} + \sum H_i$, where H_i denotes the size of hash indices of the dimension table i and $S_{\texttt{input}}$, $S_{\texttt{output}}$ denote the size of the input/output buffer.

Discussion We use the star schema queries to demonstrate the query-over-block execution model. However, the query-over-block technique can apply to other schemas as well, with appropriate modifications. The execution model first pushes down the selection operators on the dimension tables and then fuses the other operators (selection, join and aggregations on the fact table) into one GPU kernel. This approach can be straightforwardly extended to snowflake schema queries, as both steps in the query-over-block execution model can be applied to the snow flake schema.
The query-over-block execution model has two limitations. First, it assumes that the indices of the dimension tables can fit into the GPU device memory. We look forward to using multiple GPUs or enlarged GPU device memory in future generations, as one solution to this issue. Second, it requires that the query execution is driven by processing blocks of the fact table. As we discussed above, both star-schema and snowflake-schema queries can be processed by iterating over blocks of the fact table. However, there are also queries that cannot be processed by streaming a single table and immediately aggregating it. For example, queries involving many-to-many joins will need a future extension where we utilize the host memory to store intermediate results.

6. EXPERIMENTAL METHODOLOGY

We built HippogriffDB and a testbed that contains an Intel Xeon processor, an NVIDIA K20 GPU and a PCIe-attached SSD. We evaluate HippogriffDB using two popular data analytic benchmarks and we compare it with two state-of-the-art data analytics. This section describes our test bed, benchmark applications, and the two systems that we compare HippogriffDB with.

6.1 Experimental platform

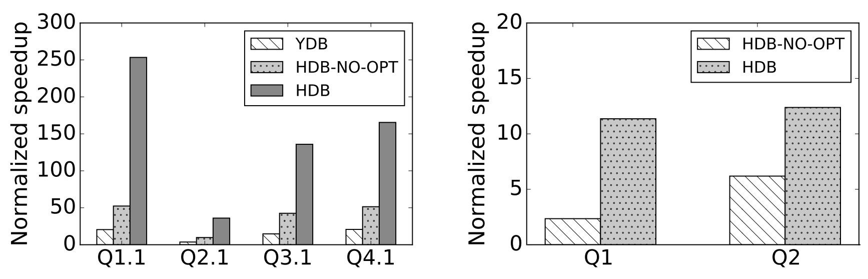
We run our experiments on a server with an Intel Xeon E52609V2 processor. The processor contains 4 cores and each processor core runs at 2.5 GHz by default. The server contains 64 GB DDR3-1600 DRAM that we used as the main memory in our experiments. The GPU in our testbed is an NVIDIA Tesla K20 GPU accelerator, which contains 5 GB GDDR5 memory on board[3]. The K20 GPU connects to the rest of the system through 16 lanes of the PCIe interconnect that provides 8 GB/sec I/O bandwidth in each direction. We use a high-end PCIe-attached SSD as the secondary storage device (with 1 TB capacity). The testbed uses a Linux system running the 3.16.3 kernel. We implement the GPU operator library in HippogriffDB based on NVIDIA CUDA Toolkit 6.5.

6.2 Benchmarks

To evaluate our system, we use two popular analytical benchmarks. The two benchmarks are the Star Schema Benchmark (SSBM) [27] and the Berkeley Big Data Benchmark (BBDB) [30].

SSBM is a widely used benchmark in database research due to its realistic modeling of data warehousing workloads. In SSBM, the database contains one fact table (`lineorder` table) and four dimension tables (`supplier`, `customer`, `date` and `part` table). The fact table refers to the other four dimension tables, as shown in Figure 9(a). SSBM provides 13 queries in 4 flights. HippogriffDB supports all 13 queries. When the scale factor is 1, the total

[3] we use an NVIDIA GTX650 GPU for the wimpy hardware experiment,



(a) Normalized speedup on SSBM (b) Normalized speedup on BBDB

Figure 10: Normalized speedup relative to MonetDB (SF=10) when data are in memory. HippogriffDB outperforms competitors by 1-2 orders of magnitude.

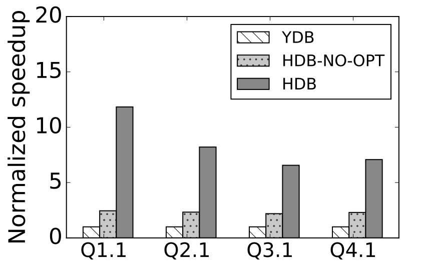
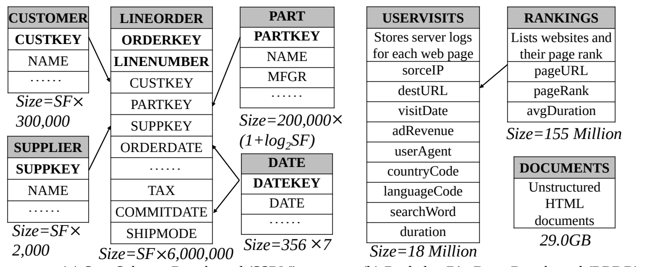
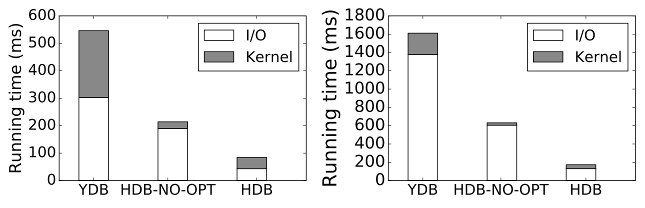


Figure 11: Performance of different systems (SF=10) when data are in SSD. HippogriffDB outperforms YDB by up to 12×.



(a) Star Schema Benchmark(SSBM) *(b) Berkeley Big Data Benchmark(BBDB)*

Figure 9: Schema of the database in SSBM and BBDB.



(a) In memory (b) On SSD

Figure 12: Break down of query Q1.1 execution time. HDB improves both kernel and I/O efficiency compared with other analytical systems.

database size is about 0.7 GB. We vary the scale factor from 1 to 1000 in our experiments. The database size is 0.7 TB when the scale factor reaches 1000.

BBDB includes several search engine workloads. The database in BBDB contains three tables, depicting documents, pageranks and user visits information, as shown in Figure 9(b). The benchmark contains 4 queries. The third query contains a string join, which current HippogriffDB does not support, and the last one involves an external Python program. Hence, we evaluate our system using Query 1 and Query 2 in this benchmark.

6.3 Competitors

We compare HippogriffDB with two analytical database systems, MonetDB [9] and YDB [40]. MonetDB is a state-of-the-art column-store database system that targets analytics over large inputs. YDB is a GPU execution engine for OLAP queries. Experiment results show that YDB runs up to 6.5× faster than its CPU counterpart on workloads that can fit in GPU's memory.

7. RESULTS

In this section, we present the experimental results for HippogriffDB. This section first presents the end-to-end performance compared with the two competitors. After that, we evaluate the effectiveness of the proposed methods in balancing component throughput inside the system. We then evaluate the execution model.

7.1 Overall performance

We first evaluate the speedup that HippogriffDB can achieve. We compare out system with two baselines: MonetDB [9] and YDB [40]. We provide two versions of HippogriffDB here: HippogriffDB without compression and pipelining optimizations (HDB-NO-OPT) and the full-fledged version (HDB).

Figure 10(a) shows the normalized speedup of different systems (relative to MonetDB) when data are in the GPU memory. We use 10 as the scale factor here. In this case, the working set size is 0.96-1.44 GB for SSBM. For BBDB, we adopt a 1.2 GB input. As shown in Figure 10(a), HDB-NO-OPT outperforms MonetDB by 38× and YDB by 2.6× on average for SSBM queries. The full-fledged version (including compression and pipelining) outperforms MonetDB by 147× and YDB by 9.8× on average. For BBDB, HDB-NO-OPT achieves 4.2× speedup compared with MonetDB and with optimizations the speedup rises to 11.8×, as shown in Figure 10(b). HDB-NO-OPT produces less speedup for BBDB compared with SSBM, as the queries in BBDB are relatively simple and cannot fully utilize the GPU computation power.

Figure 11 compares the execution time when the database resides in the SSD. As shown in the figure, HDB-NO-OPT outperforms YDB by 2.4× on average. With optimizations, HDB outperforms YDB by 8.4× on average.

Figure 12(a) breaks down the execution time of Q1.1 into I/O and kernel execution when HippogriffDB (both HDB-NO-OPT and HDB) and YDB store data in the main memory. We do not show MonetDB here, as it is not a GPU-based database and it does not have these two stages. To measure the execution time breakdown, we disable the pipelining mechanism in our systems. The execution time breakdown indicates that the majority of performance boost comes from the GPU kernel. By removing intermediate results and using new physical operators, HDB-NO-OPT runs 9.8× faster than YDB. In addition, the data transfer rate in HDB-NO-OPT is also 36% faster than YDB, due to less software and metadata overhead[4]. Compression reduces the table size by 4.6× and hence reduces the I/O time in HDB. Though the decompression adds additional cost to the GPU processing, because of the significant improvement from the I/O stage, HDB still achieves 2.5× speedup compared with HDB-NO-OPT.

We also show the execution time breakdown for the SSD version in Figure 12(b). The performance gain in this case is mainly from the optimized data transfer in HippogriffDB. The inefficiency of

[4]We use the same method as in [38] to run YDB: warm up memory by executing each query once before the experiments. Reading from a warm cache could be slower compared with reading directly from the main memory due to some operating system overhead.

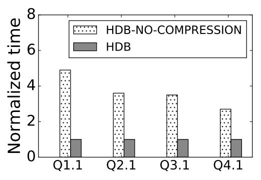


Figure 13: HippogriffDB with and without compressions, SF=10. Compression helps improve system throughput by up to 5×.

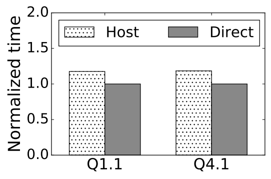


Figure 14: Effect of peer-to-peer I/O optimization, SF=1000. Direct datapath and multi-threaded help improve system throughput by 18%.

	1	10	100	1000
HDB-Q-ADAPTIVE	0.28	0.30	0.31	0.32
HDB-Q-INSENSITIVE	0.42	0.46	0.48	0.49
DICT	0.44	0.48	0.52	0.55

Table 3: Compression ratio of query-adaptive and query-insensitive compression. Query-adaptive compression can keep good compression ratio when the database scales up (x-axis is the scale factor).

I/O in YDB agrees with the results in [38]. The I/O bandwidth HDB can achieve is up to 2.3× larger than its competitor.

7.2 Close the GPU-I/O bandwidth gap

HippogriffDB fixes the gap between the fast GPU kernel and the slow data transfer by overcoming the I/O bottleneck in two ways: (1) it compresses databases and trades idle GPU cycles for decompression to achieve better data transfer efficiency. (2) it redesigns the data path to bypass the host CPU and the main memory when transferring data from the SSD to the GPU. In this subsection, we evaluate these approaches.

7.2.1 *Effect of compression*

HippogriffDB stores data in a compressed format and trades GPU cycles for better I/O performance. In this subsection, we study the effect of data compression in terms of bandwidth improvement.

We first compare the execution time with compression (HDB) and without compression (HDB-NO-COMPRESSION) for various queries. We use 10 as the scale factor here. Figure 13 shows the comparison results. Compression can achieve $2.8\times - 4.9\times$ improvement in system throughput. As discussed in Section 3, compression on the foreign key columns is the most difficult, due to its large cardinality. Compression benefits most in Q1.1, as this query only involves one foreign key. For other queries, HDB can still reach a rather decent compression ratio.

HippogriffDB adopts query-adaptive compression for databases stored in SSD. We compare the compression ratio difference between a query-adaptive compression (HDB-Q-ADAPTIVE) and a fixed approach (HDB-Q-INSENSITIVE) using the example given in Section 4.2. We show the compression ratio in Table 3. As shown in the table, the query-adaptive compression can maintain decent compression ratio when databases scaling up while the compression efficiency of the fixed approach degrades significantly. It's because the fixed compression fails to apply effective compression methods on critical foreign keys. Previous literature [40] indicates that DICT can achieve a satisfying compression effect on small data sets while our results show that the performance of DICT also degrades rapidly when data sets scale up.

7.2.2 *Effect of peer-to-peer data transfer*

Observing that data transfer bandwidth is the system bottleneck, we adopt several optimizations to improve the bandwidth. In this subsection, we evaluate the bandwidth improvement using the multi-threaded, peer-to-peer communication mechanism.

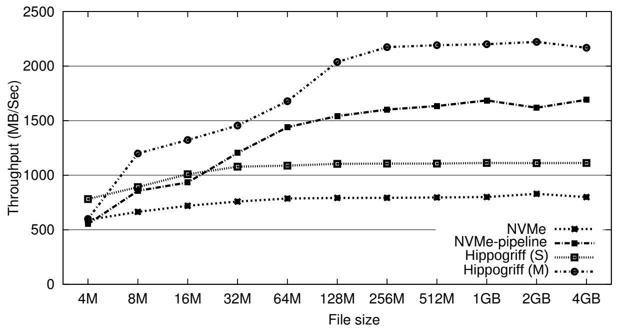


Figure 15: The throughput of different data paths in HippogriffDB.

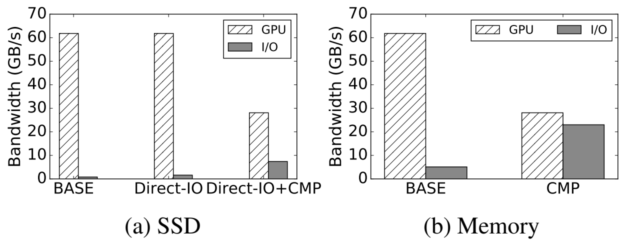


Figure 16: Effect of closing the GPU-IO bandwidth gap. Proposed approaches narrow the GPU-IO gap by up to 21×.

Figure 15 compares the throughput of moving data from the SSD to the GPU using Hippogriff (M) against standard NVMe (NVMe), pipelined NVMe (NVMe-pipeline) [5] and the single channel, peer-to-peer transfer (Hippogriff (S)). We report the data transfer throughput under different file sizes, excluding the overhead of allocating all necessary resources (e.g., memory buffers) along the data paths.

Hippogriff (M) outperforms all other route options. The performance advantage of Hippogriff (M) becomes more significant as file size increases. When transferring a 4 GB file between the SSD and the GPU, Hippogriff (S) that performs file access requests using a single NVMe command queue only achieves bandwidth of 1110 MB/sec, due to the under-utilized NVMe SSD resources. Hippogriff (M), on the other hand, offers up to 2221 MB/sec bandwidth. NVMe-pipeline improves the performance of standard NVMe by compensating for latencies with multiple data transfers. However, NVMe-pipeline can still only achieve a throughput of 1691 MB/Sec between the SSD and the GPU for 4 GB files, 34% slower than Hippogriff (M), because NVMe-pipeline requires more CPU resources.

We compare the execution time of using an optimized host route and using Hippogriff. Experiment shows that the peer-to-peer data transfer helps reduce the end-to-end latency by 19%.

As a summary of the effect of the endeavours discussed above, Figure 16 shows the effect of narrowing the bandwidth gap between the GPU kernel and I/O. We compare the difference between GPU kernel and data transfer bandwidth using SSBM Q1.1 and show the results for both SSD-based and memory-based HippogriffDB. For the SSD version, the initial gap (BASE) is up to 82×. The direct data transfer (Direct-IO) brings it down to 38× and the compression (Direct-IO+CMP) further brings the gap down to 3.9×. For the in-memory version, compression (CMP) narrows the gap from 12× to 1.2×, very close to achieving the balance.

7.3 Query-over-block model evaluation

[5]This is an optimized baseline that overlaps the SSD access with GPU memory copy.

	1	10	100	1000
YDB	12.27	98.03	N/A	N/A
HippogriffDB	1	10.27	93.60	938.0

Table 4: Normalized scalability performance with increasing SF (from 1 to 1000). Results testify the scalability of HippogriffDB (x-axis is the scale factor).

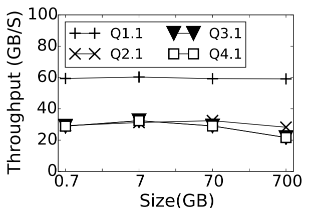


Figure 17: Scalability of GPU kernels on SSBM. GPU kernel throughput is consistently higher than I/O bandwidth by over one order of magnitude.

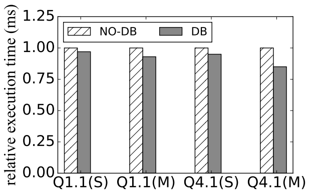


Figure 18: Effect of double buffering. Double buffering helps reduce the execution time by up to 15% without compression. It can further improve system performance with other optimizations.

The query-over-block model makes HippogriffDB the first GPU-based database system that provides native support for big data analytics. The query model uses several optimizations to improve performance, including removing materialization and double buffering. In this subsection, we first evaluate the system scalability and then analyze the effect of the proposed optimizations.

7.3.1 System scalability

We test the scalability of HippogriffDB by varying the scale factor from 1 to 1000 (database size from 0.7 GB - 0.7 TB). We run the SSBM Q1.1 in the experiment without compression. Table 4 reports the execution time for queries on YDB and HippogriffDB. The database resides in SSD in this experiment. As shown in the table, YDB cannot support queries when the scale factor is above 10 while HippogriffDB shows its superiority by scaling up to support terabyte-level input.

When scaling up, the throughput of HippogriffDB remains stable (as same as the data transfer bandwidth). This is because the GPU kernel always runs faster than data transfer bandwidth in this case. We show the GPU kernel throughput in Figure 17: the speed that the GPU processes database queries (more than 20 GB/s) is at least 12× higher than the I/O bandwidth. This trend sustains when the input scales up to terabyte-level tables. The double buffering always keeps I/O device busy and saturates the I/O bandwidth. As a result, the performance remains the stable when scaling up. The I/O bandwidth without optimization is not satisfiable and that is the reason we propose compression and peer-to-peer data transfer to improve the effective I/O bandwidth.

7.3.2 Effect of double buffering

HippogriffDB uses double buffering to overlap the data transfer and kernel execution, reducing the execution time of query processing. We compare the effect of using double buffering in Figure 18 (SF = 10). The double buffering reduces the execution time for Q1.1 in SSBM by 3% and 7% for SSD-based and memory-based HippogriffDB. For Q4.1, it can help improve the execution time by 5% and 15% respectively. Double buffering works better on complex queries, as the GPU kernel time consumes higher portion in the total execution time for complex queries. Double buffering can further improve the system performance in combination with other optimizations, such as compression. With data compression, the gap between faster part and slower part narrows and hence the overlapping can result in more performance gain.

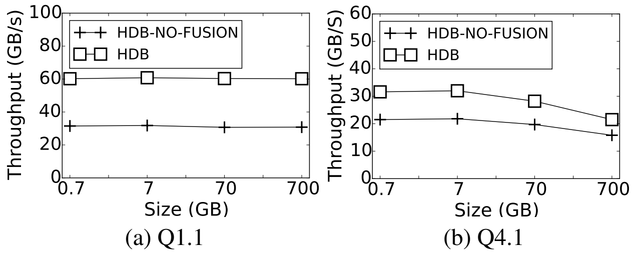


Figure 19: Effect of removing intermediate results. Removing intermediate results can improve query execution time by up to 91%.

7.3.3 Effect of avoiding intermediate results

Figure 19 compares the the benefit of reducing intermediate results using SSBM Q1.1 and Q4.1. We vary SF from 1 to 1000 (database size from 0.7 GB - 0.7 TB). We compare the throughput of HippogriffDB (HDB) and HippogriffDB without operator fusion (HDB-NO-FUSION). As shown in Figure 19, the GPU kernel throughput improves by 91% for Q1.1 and 43% for Q4.1. Reducing intermediate results works better on light-weighted queries. For heavy-weighted queries, the computation can take a significant portion of time and operator fusion will not optimize for this part. For query 4.1, the benefits of reducing intermediate results decreases with the growth of the scale factor. It is also because the computation load increases with the growth of the scale factor.

7.4 Performance on wimpy hardware

In the previous sections, we discuss the proposed optimizations on the high-end hardware. In this subsection, we examine the optimizations on the wimpy hardwares, such as low-end GPUs.

While Hippogriff does not work with low-end GPUs because of the BIOS setup of manufacturers, the query-over-block execution model can still improve the kernel efficiency on the wimpy hardware. By reducing the intermediate results, the query-over-block execution model improves the GPU processing rate by 2.9× for light-weighted query (SSBM Q1.1, SF = 10) and by 2.4× for heavy-weighted query (SSBM Q4.1, SF=10). Without peer-to-peer data communication supports, the multi-threaded transfer still helps boost the bandwidth in the system using low-end GPUs. As shown in Figure 15, using multi-threaded I/O can achieve up to 1.6 GB/s on our SSD. Compression still works to narrow the bandwidth mismatch between the SSD and the GPU. For example, the compression can increase the effective I/O bandwidth by 4.6× on SSBM Q1.1 while the GPU can still process at 22.9 GB/s, 3.3× larger than the effective bandwidth.

8. RELATED WORK

With the end of Dennard scaling [12] (power density stays constant), it is hard for general purpose CPUs to provide scalable performance in the future due to the power challenges [13, 16]. In recent years, researchers in database community started to use heterogeneous computing to overcome the scaling problem of CPUs and to continue delivering scalable performance for database applications [18, 19, 32, 40].

Among various hardware accelerators, GPU is the one that draws the most attention. Several full-fledged GPU database query engines [10, 20, 40] came out in the recent years. Ocelot [20] provides a hybrid analytical query engine as an extension to MonetDB. HyPE [10] is a hybrid analytical engine utilizing both the CPU and the GPU for query processing. YDB [40] is a GPU-based data warehouse query engine. Though YDB allows database store in the main memory or the SSD, it still assumes that the working set can fit in the main memory. HippogriffDB differs from the previous work as HippogriffDB is targeting large scale database systems

(TB scale input). HippogriffDB allows data sets larger than the GPU memory capacity. To cope with the limited GPU memory capacity, HippogriffDB uses streaming database operations which enable data processing on small chunks.

Several CPU-based databases also use block-oriented execution model. [9] identifies that the system bottleneck in a CPU-based in-memory database is the limited memory bandwidth and uses a cache-aware approach to reduce memory traffic. However, the limited memory bandwidth concern of CPU-based databases does not hold for a GPU-based system, as the GPUs have much higher memory bandwidth (100s GB/sec). HippogriffDB uses a block-based execution to remove the scalability limitation posed by the small GPU memory capacity. The block size between HippogriffDB and [9] is also different: HippogriffDB chooses a size that is large enough to deliver good I/O bandwidth from the SSD to the GPU, which is much larger than the cache size (10s KB on GPUs).

Compression is a popular strategy to reduce the storage space and the amount of data transfer. Several works [14, 26, 29] discussed the algorithms of compression/decompression on GPU. YDB [40] uses dictionary and run-length encoding to reduce data sets so that it can support tables slightly larger than the GPU memory capacity. HippogriffDB differs from the previous work as HippogriffDB uses the query-adaptive compression. Wu et al. [39] proposed a primitive fusing strategy to reduce the back-and-forth traffic between GPU and hosts. HippogriffDB adopts a similar technology to reduce the data exchange.

There are several related projects on the direct communication between two PCIe devices. For example, GPUDirect [3] offers direct communication between two GPUs and [22] offers direct communication between the Network Interface Card (NIC). Our work differs from those works in two ways. First, our work demonstrates that low I/O bandwidth from the SSD to the GPU is largely due to the failure to fully utilize the internal parallelism inside the SSD. To address this issue, we adopt multi-threaded I/O to boost the utilization of the multiple data transfer units. Second, our work offers direct communication between a GPU and a PCIe SSD.

Several works [31, 36] discussed the gap between throughput of GPU kernel and off-chip memory bandwidth and proposed using compression to alleviate discrepancy. HippogriffDB differs from these works in two aspects. First HippogriffDB tries to reduce the gap between between the GPU kernel and SSD I/O throughput. Second, HippogriffDB achieves better compression ratio by using aggressive and adaptive compression strategies.

9. CONCLUSION

In this paper, we proposed HippogriffDB, an efficient, scalable heterogenous data analytics system. HippogriffDB is the first GPU-based data analytics that can scale up to support terabyte input. HippogriffDB reaches high performance by fixing the huge imbalance between GPU kernel and I/O using compression and peer-to-peer transfer path. HippogriffDB uses a streaming execution model to process data sets larger than the GPU memory. Our comprehensive experiments have demonstrated the superiority of HippogriffDB in terms of both scalability and performance.

10. REFERENCES

[1] http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-s3700-spec.pdf.

[2] http://www.nvidia.com/object/tesla-servers.html.

[3] https://developer.nvidia.com/gpudirect.

[4] http://blog.pmcs.com/project-donard-peer-to-peer-communication-with-nvm-express-devices-part-two.

[5] https://trademarks.justia.com/865/43/nvmedirect-86543720.html.

[6] D. J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

[7] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

[8] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, J. Gehrke, L. Haas, A. Halevy, J. Han, et al. Challenges and opportunities with big data 2011-1. 2011.

[9] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[10] S. Breß and G. Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *VLDB*, 6(12):1398–1403, 2013.

[11] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO*, pages 225–236. IEEE Computer Society, 2010.

[12] R. H. Dennard, V. Rideout, E. Bassous, and A. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, 2011.

[14] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *VLDB*, 3(1-2):670–680, 2010.

[15] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336. ACM, 2006.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(EPFL-ARTICLE-168285):6–15, 2011.

[17] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[18] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *VLDB*, 4(5):314–325, 2011.

[19] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *VLDB*, 6(10):889–900, 2013.

[20] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 6(9):709–720, 2013.

[21] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[22] S. Kim, S. Huh, Y. Hu, X. Zhang, A. Wated, E. Witchel, and M. Silberstein. Gpunet: Networking abstractions for gpu programs. In *OSDI*, pages 6–8, 2014.

[23] R. Kimball and M. Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[24] Y. Liu, H.-W. Tseng, M. Gahagan, J. Li, Y. Jin, and S. Swanson. Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems. In *ICCD*, 2016.

[25] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[26] M. A. O'Neil and M. Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *GPGPU*, page 7. ACM, 2011.

[27] P. ONeil, E. ONeil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance evaluation and benchmarking*, pages 237–252. Springer, 2009.

[28] R. Pagh and F. F. Rodler. *Cuckoo hashing*. Springer, 2001.

[29] R. Patel, Y. Zhang, J. Mak, A. Davidson, J. D. Owens, et al. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[30] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[31] V. Sathish, M. J. Schulte, and N. S. Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *PACT*, pages 325–334. ACM, 2012.

[32] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, Broomfield, CO, Oct. 2014. USENIX Association.

[33] B. Smith. A survey of compressed domain processing techniques. *Cornell University*, 1995.

[34] J. Teuhola. A compression method for clustered bit-vectors. *Information processing letters*, 7(6):308–311, 1978.

[35] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson. Gullfoss: Accelerating and simplifying data movement among heterogeneous computing and storage resources. Technical report.

[36] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps. In *ISCA*, pages 41–53. ACM, 2015.

[37] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *VLDB*, 5(11):1543–1554, 2012.

[38] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *VLDB*, 7(11):1011–1022, July 2014.

[39] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO*, pages 107–118. IEEE Computer Society, 2012.

[40] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *VLDB*, 6(10):817–828, 2013.