Database Compression on Graphics Processors

Wenbin Fang
Hong Kong University of Science and Technology
wenbin@cse.ust.hk

Bingsheng He
Nanyang Technological University
he.bingsheng@gmail.com

Qiong Luo
Hong Kong University of Science and Technology
luo@cse.ust.hk

ABSTRACT

Query co-processing on graphics processors (GPUs) has become an effective means to improve the performance of main memory databases. However, this co-processing requires the data transfer between the main memory and the GPU memory via a low-bandwidth PCI-E bus. The overhead of such data transfer becomes an important factor, even a bottleneck, for query co-processing performance on the GPU. In this paper, we propose to use compression to alleviate this performance problem. Specifically, we implement nine lightweight compression schemes on the GPU and further study the combinations of these schemes for a better compression ratio. We design a compression planner to find the optimal combination. Our experiments demonstrate that the GPU-based compression and decompression achieved a processing speed up to 45 and 56 GB/s respectively. Using partial decompression, we were able to significantly improve GPU-based query co-processing performance. As a side product, we have integrated our GPU-based compression into MonetDB, an open source column-oriented DBMS, and demonstrated the feasibility of offloading compression and decompression to the GPU.

1. INTRODUCTION

Graphics processors (GPUs) have become an emerging and powerful co-processor for many applications including scientific computing [12] and databases [11, 13, 18, 19]. The new-generation GPU has an order of magnitude higher memory bandwidth and higher GFLOPS (Giga FLoating point Operations Per Second) than the multi-core CPU. For example, an NVIDIA GTX 280 card contains 240 cores with a peak performance of 933 GLOPS and 141.7 GB/s memory bandwidth. Despite of the superb hardware performance, GPU co-processing requires data transfer between the main memory and the GPU memory via a low bandwidth PCI-E bus, e.g., with theoretical peak bandwidths of 4 and 8 GB/s on 16-lane PCI-E v1.0 and v2.0, respectively. As a result, previous studies [18, 19] show that such data transfer can contribute to 15–90% of the total time in relational query co-processing. Query co-processing performance can be further improved if such data transfer time is reduced. Compression has been long considered as an effective means to reduce the data footprint in databases, especially for column-oriented databases [1, 2, 5, 15, 26, 28]. In this paper, we investigate how compression can reduce the data transfer time and improve the overall query co-processing performance on column-oriented databases.

Database compression has been extensively studied in column-oriented databases (e.g., MonetDB/x100 [6] and C-store [25]). Most of these systems adopt lightweight compression schemes, such as dictionary encoding, instead of more sophisticated compression algorithms such as gzip. With the lightweight compression schemes, column-oriented databases efficiently utilize *vectorized* executions, processing data tuples bulk-at-a-time or vector/array-at-a-time via (de)compressing multiple tuples simultaneously [1, 26, 28]. However, due to the poor decompression performance on CPUs, current systems often use a single compression scheme, even though *cascaded compression* (i.e., applying multiple lightweight compression schemes one after another on a data set) is potentially more effective. For example, Harizopoulos et al. [15] showed that the column-oriented DBMS with the combination of two lightweight compression schemes hardly outperforms that with a single compression scheme. Given the superb memory bandwidth and GLFOPS of the GPU, we should revisit the commonly used lightweight compression schemes as well as their combinations on the GPU. The reduced data footprint by cascaded compression in GPU co-processing can potentially compensate the computational overhead.

We start our investigation with a recent column-oriented query co-processor on the GPU, namely GDB [18]. We implement and integrate nine common lightweight compression schemes into GDB. All these schemes are optimized with the massive thread parallelism and the memory locality of the GPU. We further investigate the compression ratio, defined as the size of compressed data divided by the size of original data, and the query processing performance in GDB for different combinations of these compression schemes.

Table 1 shows the compression ratio of a sorted column l_partkey from the decision support benchmark TPC-H under six different combinations of compression schemes, along with the running time of a selection with a 10% selectivity on this column in GDB. We call a combination a *compression plan*. Compression schemes in a plan are delimited by commas, and a subsequent scheme is applied on the output of the preceding scheme. Some schemes may output multiple columns. Therefore, we use nested brackets to represent such cases. For example, "RLE, [[DELTA, NS] | NS]" denotes that we first apply the compression scheme RLE, then we apply DELTA followed by NS on one output column of RLE, and NS on the other output column. "ϵ" denotes an empty plan which does not apply any compression scheme. Readers can simply consider the compression schemes such as RLE in a compression plan as placehold-

Proceedings of the VLDB Endowment, Vol. 3, No. 1

ers in Table 1. The details on individual compression schemes and experimental setup can be found in Sections 3 and 5, respectively.

We make the following observations from Table 1: 1) For the selection operation, some plans do not need decompression (e.g., plan A, B, and D), some with partial decompression (e.g., plan E), and some with full decompression (e.g., plan C, and F). 2) Compression plans containing multiple schemes may achieve a better compression ratio than that of a single scheme (e.g., plan E is better than plan A), or may not (e.g., plan A is better than plan C). 3) Scheme order in a plan matters. For example, plans E and F contain the same set of schemes, but achieve different compression ratios on the same column. 4) The same scheme can be applied for multiple times in a compression plan (e.g., NS in plan E). 5) Complex compression plans, e.g., E and F, do not necessarily improve or slow down query processing performance.

As different combinations of compression schemes vary significantly in compression ratios and GPU query processing performance, it is necessary and worthwhile to study cascaded compression on the GPU. In particular, we are facing two main technical challenges when applying cascaded compression on GPU query co-processing. First, as the search space contains a large number of combinations of schemes, how do we pick a feasible plan that fulfills predefined goals, such as good compression ratio or good (de)compression performance? Second, cascaded compression reduces the possibility of evaluating queries directly on the compressed data, which induces the costly decompression. For example, the compressed data may not be order preserving, so that a range query cannot be evaluated on the compressed data.

To address these challenges, we develop a rule-based compression planner to automatically choose feasible compression plans that are applicable to data with certain properties. For example, some schemes require data to be sorted, so that they can effectively reduce the data size. We also propose a compression-aware optimizer to determine whether query evaluation should be done with no, partial, or complete decompression. Our rule-based compression-aware optimizer makes partial decompression possible. To facilitate the decision in picking the optimal compression plan, we take the parallel execution mechanism of GPUs into consideration, and develop an accurate cost model for GPU-based compression.

Compression on l_partkey		Table-scan on l_partkey	
Compression Plans	**Compression Ratio**	**Decompression**	**Time (ms)**
A: RLE	6.68%	No	8.30
B: NS	100%	No	162.04
C: DELTA, NS	25%	Full	119.62
D: RLE, [ϵ | NS]	4.16%	No	6.39
E: RLE, [[DELTA, NS] | NS]	1.67%	Partial	5.31
F: DELTA, RLE, [NS | NS]	3.33%	Full	151.99

Table 1: Compression ratios and table-scan performance in GDB.

We have conducted experiments on the performance of nine compression schemes, and evaluated the effectiveness of the compression planner and the compression-aware optimizer with the TPC-H benchmark. Our results show that the GPU-accelerated compression schemes are with a (de)compression bandwidth as high as 56 GB/s. In addition, the compression planner can effectively provide feasible compression plans. With partial decompression facilitated by the compression-aware optimizer, evaluating TPC-H queries on GPUs can achieve an order of magnitude performance speedup over MonetDB on an Intel quad-core CPU.

Organization. The remainder of this paper is organized as follows. In Section 2, we briefly introduce GPUs and CUDA, then review the related work on database compression and GPU-based co-processing. We present the implementation of GPU-based compression schemes in Section 3, followed by cascaded compression in Section 4. We present the experimental results in Section 5 and conclude this paper in Section 6.

2. PRELIMINARY AND RELATED WORK

In this section, we briefly introduce GPUs and CUDA, the underlying platform upon which our compression schemes are implemented. Next, we review the related work on GPU-based query processing and database compression.

2.1 GPUs and NVIDIA CUDA

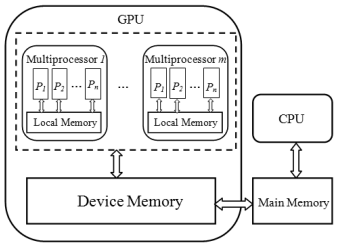


Figure 1: The GPU Many-core Architecture.

GPUs, originally designed for graphics rendering tasks, have evolved into massively multi-threaded many-core co-processors, as illustrated in Figure 1, for general-purpose computing. The GPU consists of many SIMD (Single Instruction, Multiple Data) multiprocessors, all sharing a piece of device memory.

CUDA, a general-purpose programming framework for NVIDIA GPUs, exposes the hierarchy of GPU threads. GPU threads execute the same code of a *kernel* function concurrently on different data. *Warps*, each of which consists of the same number of threads, are scheduled across multiprocessors. Within each multiprocessor, warps are further grouped into thread blocks. Threads in the same thread block share resources on one multiprocessor, e.g., registers and local memory (or called shared memory in NVIDIA's term).

CUDA also exposes the memory hierarchy to developers. Multiprocessors share the device memory, which has a high bandwidth and a high access latency. For example, NVIDIA GTX 280 GPU has a device memory of size 1 GB, a bandwidth of 141 GB/s, and a latency of 400 to 600 cycles. If threads in a half warp access consecutive device memory addresses, these accesses are coalesced into one memory access transaction. By utilizing the memory coalesced access feature, we can significantly reduce the number of device memory accesses, and improve the memory bandwidth utilization. Each multiprocessor also has a low-latency and small-sized local memory, and accesses to the local memory must be explicitly programmed through CUDA.

2.2 GPU-based Data Processing

Due to the massive parallelism, GPUs have demonstrated significant speedup over CPUs on various query processing tasks, including sorting [11], index search [22], join [19], selection and aggregation [13]. In addition to relational query processing, GPUs have been applied to other data management tasks, for example, data mining [8, 10, 27], spatial databases [4], MapReduce [16], scatter/-gather [17] and similarity joins [23]. GPU-based data compression has been applied on the inverted index in information retrieval [9]. In comparison, we not only investigate individual GPU-accelerated compression schemes for column-oriented databases, but also the combinations of different schemes for reducing the overhead of data transfer between the main memory and the GPU memory.

2.3 Database Compression

Compression is an effective method to reduce the data storage and to improve query processing performance. In disk-based databases, compression algorithms are typically computation-intensive, and they trade more computation for better compression ratios, since disk accesses are far slower than the CPU. Because the compression algorithms are complex, the compressed data are often required to be fully decompressed for query processing.

As the price of main memory drops, machines tend to have large amounts of main memory. Moreover, column-oriented databases allow query processing to read only the required columns, instead of the entire rows. Disk I/Os become less significant for such scenarios. As a result, studies on compression techniques in column-oriented databases [1, 2, 5, 15, 20, 28] have focused on lightweight schemes such as Run-Length-Encoding (RLE) that balance between compression ratio and computational overhead. With these lightweight compression schemes, most queries can be evaluated without decompression. In addition, lightweight compression can easily be vectorized [1, 26, 28]. Although the combination of lightweight schemes is undesirable on CPU-based column-oriented databases [15] due to the computation overhead, the high memory bandwidth and the high computation capability of GPUs make combinations of multiple compression schemes practical. Moreover, compression can benefit GPU query co-processing [18] by significantly reducing the overhead of data transfer between the main memory and the GPU memory.

3. COMPRESSION ON THE GPU

In this section, we present our GPU-based implementation of nine lightweight compression schemes. These schemes have been commonly used in column-oriented databases, since most of them support query processing on compressed data. While these schemes have been investigated on the CPU in previous studies [1, 28], we study them on the GPU architecture.

As the nine lightweight compression schemes we consider can be efficiently vectorized [28], they fit well with the many-core SIMD architecture of the GPU. Moreover, since data-parallel primitives have been proposed as a systematic and efficient means to implement GPU-based query operators [18], we also implement the compression schemes on the GPU with these primitives, specifically Map, Scatter, and Prefix Sum. The input and output of these data-parallel primitives are simply arrays on the GPU. The data layout and data accesses are optimized with the GPU memory hardware features, and the performance is tuned with the thread parallelism of the GPU. We briefly introduce the three primitives as follows:

Map. Given an input array and a user-defined map function, the output array is the result of applying the map function on each element of the input array.

Scatter. Given an input array and an array of write positions, the Scatter primitive writes the values in the input array to an output array at positions given by the array of write positions.

Prefix Sum. Given an input array, the prefix sum is to output an array of sums, with each sum obtained from adding up values in the input array up to the position of the sum. We adopt both *inclusive* and *exclusive* prefix sums. The difference between the two types of prefix sum is whether the sum includes the element in the input array corresponding to the position of the sum in the output array. As such, the first element of the output array of exclusive prefix sum is zero, and that of the inclusive prefix sum is the first element of the input array.

We categorize the nine compression schemes into two groups, namely *Main Schemes* and *Auxiliary Schemes*. The main schemes reduce the data size, whereas the auxiliary schemes transform data into formats that are suitable for main schemes to perform compression. We present our primitive-based GPU implementation of these compression schemes next.

3.1 Main Schemes

Null Suppression with Fixed Length (NS). NS is a variant of the "fixed-length minimum bit" scheme, deleting leading zeros at the most significant bits in the bit representation of each element [3]. NS encodes each element in the input array with the same number of bits, and the number of bits required for the encoding is stored in the database catalog. In our GPU-based NS encoding, we make each output value byte-aligned for a good (de)compression performance, because the GPU is byte addressable. For example, NS encodes two four-byte numbers {0x00000FFF, 0x00000009} as two two-byte values {0x0FFF, 0x0009}. We use the Map primitive to implement NS on the GPU, and encode the value in the minimum number of bytes. If an encoded value is one byte (or two bytes), we use the GPU-native type *char* (or *short*) for efficiency.

Null Suppression with Variable Length (NSV). NSV is a variant of "variable-length minimum bit" compression scheme [3]. NSV is similar to NS in eliminating leading zeros at the most significant bits. The difference is that NSV allows each value to be encoded in a variable length, and that the length of each encoded value is recorded as well as the encoded value itself. The lengths themselves are of the same size. For example, we encode a four-byte integer into a byte-aligned value that can be of size of 1, 2, 3 or 4. Thus, all lengths (sizes) are represented in two bits [1]. We use the Map primitive to implement NSV, and store the encoded value and length as two columns in each output element. Correspondingly, in NSV decompression, we apply the exclusive prefix sum primitive on the column of length to obtain the offset of each encoded value, and then use the Map primitive to convert the encoded value into the original value.

Dictionary (DICT). DICT is widely used in existing DBMSs, e.g., Oracle [24] and MonetDB [7]. DICT is suitable for columns that have only a small number of distinct values. It maintains a dictionary of the distinct values in the database catalog for decompression. As such, fast retrieval on the dictionary is critical for the efficiency. In our GPU-based implementation, we load the dictionary into the local memory of each GPU multiprocessor to enable fast retrieval, and the (de)compression is a Map with the map function looking up the dictionary.

Bitmap (BITMAP). When a column has a very small number of distinct values (e.g., gender), it is suitable to adopt the BITMAP scheme to encode each distinct value as a bit-string. Each of such a bit string has only one bit-1 and all other bits are 0. The position of the bit-1 in a bit string differs by the distinct value it represents. For example, BITMAP encodes the four two-byte numbers {0x00FF, 0x0009, 0x0009, 0x00FF} as two one-byte bit strings, {1001 0000, 0110 0000} (in big endian as on the GPU). The distinct values 0x00FF and 0x0009 and their corresponding bit-strings 10 and 01 are stored in the database catalog. Our GPU-based BITMAP is implemented with the Map primitive.

Run-Length-Encoding (RLE). RLE represents values in each run with a pair: (value, run length). We store the encoded values and run lengths in two columns (arrays). In the decompression, RLE constructs runs using the columns of values and run lengths. We implement the GPU-based RLE in four steps. First, we identify boundaries between runs. A boundary is represented by a 1 between 0's and we call this array of 0's and 1's a *boundary array*. Second, we get the write positions for output data by applying an exclusive prefix sum on the boundary array. Third, given the

write positions, we scatter both the values and the boundary positions. Finally, we compute each run length by substracting the corresponding boundary position from the next boundary position. This compression process is lock-free, which well exploits the massive parallelism of the GPU. The following example illustrates the RLE compression procedure for compressing a column containing nine values ("AAAABBCCC"):

```
Step 1, Map (find boundary of runs) --
INPUT/Uncompressed Column: A A A A B B C C C
OUTPUT/Boundary          : 0 0 0 1 0 1 0 0 1
Step 2, Exclusive Prefix Sum --
INPUT/Boundary           : 0 0 0 1 0 1 0 0 1
OUTPUT/Scatter Position  : 0 0 0 0 1 1 2 2 2
Step 3, Scatter --
INPUT/Uncompressed Column: A A A A B B C C C
INPUT/Boundary           : 0 0 0 1 0 1 0 0 1
INPUT/Scatter Position   : 0 0 0 0 1 1 2 2 2
OUTPUT/Value Array       :       A   B     C
OUTPUT/Boundary Position :       4   6     9
Step 4, Map (calculate run lengths) --
INPUT/Boundary Position  :       4   6     9
OUTPUT/Run Length        :       4   2     3
```

A naive GPU-based implementation of GPU-based RLE decompression is to use one GPU thread to decompress each run of values. However, if data skew results in significant differences among run lengths, load can be considerably unbalanced among threads. Such load imbalance hurts the utilization of the GPU thread parallelism and therefore has a negative performance impact. Thus, we design a decompression algorithm that can efficiently handle data skew. The algorithm works in four steps. First, we compute the boundary positions of runs by applying an inclusive prefix sum on the run lengths. Second, we generate the boundary array by setting 1's in the array at the corresponding boundary positions. Third, we compute the write positions for output data by applying an inclusive prefix sum on the boundary array. Finally, we scatter values to the output value column. This procedure is also lock-free. Moreover, in each step all GPU threads have the same amound of workload regardless of data distribution. The decompression procedure is illustrated as follows, with the same example ("AAAABBCCC"):

```
Step 1, Inclusive Prefix Sum --
INPUT/Run Length         : 4       2   3
OUTPUT/Boundary Position : 4       6   9
Step 2, Scatter --
INPUT/Boundary Position  :         4   6    (9)
OUTPUT/Boundary          : 0 0 0 0 1 0 1 0 0
Step 3, Inclusive Prefix Sum --
INPUT/Boundary           : 0 0 0 0 1 0 1 0 0
OUTPUT/Scatter Position  : 0 0 0 0 1 1 2 2 2
Step 4, Scatter --
INPUT/Value Array        : A       B   C
INPUT/Scatter Position   : 0 0 0 0 1 1 2 2 2
OUTPUT/Uncompressed Column: A A A A B B C C C
```

3.2 Auxiliary Schemes

Frame-Of-Reference (FOR). FOR transforms each value in a column into an offset from a base value. The base value of the column is often set to the smallest value in the column, and is stored in the database catalog. The GPU-based FOR uses the Map primitive to subtracte the base value from each array element. After applying FOR, we can compress the encoded offsets by other main schemes, such as NS and NSV.

Delta (DELTA). DELTA encodes a value in a column as the difference from the value at the preceding position. The first value in the column is stored in the database catalog. The differences are usually within a small value domain for a sorted column. The GPU-based DELTA uses the Map primitive to perform compression, and applies the inclusive prefix sum for decompression.

Separate (SEP). The Separate (SEP) scheme is to split a value into multiple components, so that each component can be compressed individually. For example, the Date type is usually represented as an integer in DBMSs, e.g., a four-byte integer in MonetDB. It will be more compressible if we separate a date into day, month, and year, each of which has a limited value domain. The Map primitive is used for the GPU-based SEP implementation.

Scale (SCALE). The Scale scheme is to convert floating point numbers into integers, when the integer precision is sufficient for the application. For example, prices of \$9.3 and \$9.4 can be converted to 93 and 94 through multiplying by a factor of 10, with the factor 10 stored in the database catalog. We use the Map primitive to implement the SCALE scheme.

In summary, we implement the nine lightweight compression schemes using GPU-optimized data-parallel primitives, which have been extensively tested and evaluated in previous studies [18, 19]. The high efficiency of individual GPU-based compression schemes lay a foundation for the combination of multiple schemes on the GPU. The following section explores cascaded compression.

4. CASCADED COMPRESSION

Based on the GPU-accelerated, individual compression schemes, we further investigate the combinations of multiple compression schemes (namely *cascaded compression*) on the GPU. The core benefit of cascaded compression is to further reduce the data size by applying more than one lightweight scheme. While cascaded compression remains of low interest on the CPU, it appears promising to further alleviate the performance issues in data transfer and query processing on the GPU.

The main question in cascaded compression is, given a number of individual compression schemes and a data set (a column), how do we find a feasible and good combination of the individual schemes (compression plan)? Our answer to this question is through a compression planner together with a cost model. In this section, we first present our compression planner, and then describe our cost model for GPU-based compression schemes.

4.1 Compression Planner

Hypothetically, a compression plan can consist of an arbitrary combination of schemes, and each scheme can appear an arbitrary number of times. Due to the combinatorical nature, the search space of feasible plans is large. Consider the nine compression schemes in our study. Even if we limit a compression plan to have at most six component schemes, there will be $\sum_{i=0}^{6} 9^i$, or 597,870 plan candidates. Therefore, effective pruning techniques are required to reduce the search space.

One important factor to consider in reducing the search space for compression plans is data properties. Specifically, a compression scheme is effective only when certain data properties are satisfied. For example, RLE is effective when the average run length in a column is large. Furthermore, certain properties of a column will change after compression. For example, after applying RLE, the average run length becomes small and other properties remain unchanged. Additionally, the goodness of a plan can be measured with different criteria, for example, compression ratio, (de)compression performance (cost), or a combination of multiple factors.

Considering both data properties and plan goodness measures, we design our compression planner consisting of two components: a *tactical* planner and a *strategic* planner, as in the query optimization in MonetDB [7]. The tactical planner uses a rule-based method to automatically prune the search space for a predefined maximum number of scheme candidates, and the strategic planner allows developers to specify their goals (measures for plan goodness).

The tactical planner prunes the space according to the rules on

data properties of the compression schemes. We identify the following five compulsory properties for a column:

Sorted. It indicates whether the column is sorted. RLE and DELTA tend to be effective on sorted data.

Average run length. It is the average length of the runs in a column, which is the key data property for RLE.

NumDistinct. It measures the number of distinct values in a column. Both DICT and BITMAP work well with columns of a small NumDistinct.

Value domain. It records the minimum number of bytes needed to represent each value in the column. Value domain affects the effectiveness of NS, NSV, and FOR.

Compound. It indicates whether a value in the column can be divided into multiple components. This property is considered by both SEP and SCALE.

In our implementation, for each column we record a property list, containing the five properties of the column. Furthermore, we maintain a *compulsory property table (CPT)* to record the rules of the nine compression schemes on the five data properties. The tactical planner can then choose a set of compression scheme candidates for a column by checking the property list of the data against the rules in CPT. For the tactical plan to select a subsequent candidate scheme to apply on the compressed data, we maintain another set of rules in the *transitional property table (TPT)*. The TPT rules specify the data properties resulted from the nine compression schemes. Then, based on the new property list of the compressed data, the planner iteratively finds new feasible plans of increasing number of component schemes, until a predefined maximum number of plan candidates is reached.

With feasible plan candidates resulted from the tactical planner, the strategic planner chooses a final plan based on a goodness measure. A common goodness measure is the compression ratio, which can be readily estimated using statistics from the database catalog. Another goodness measure is the (de)compression performance, or time cost. Other goodness measures are also possible, for example, a weighted combination of compression ratio and time cost. In our implementation, we support both compression ratio and time cost as goodness measures. In the following, we describe our cost model for estimating the time cost of GPU-based compression.

4.2 Cost Model

There has been little prior work on estimating the performance of GPU-based program execution. The cost model in GDB [18] treats the GPU as a black box, and uses a calibration-based method for the cost estimation of GPU-based query processing. In comparison, our cost model on GPU-based compression explicitly takes into account the GPU thread scheduling and memory access mechanisms. This cost model requires no actual execution of the compression algorithms, and works well on latest CUDA-enabled GPUs.

The main idea of our cost model is to estimate the execution time of a compression algorithm by examining the code of the kernel function and calculating its total workload given the statistics of the data. This estimation is based on the parallel execution mechanism of CUDA-based GPUs. Specifically, we perform our estimation around the GPUs schedule unit, *warp* of threads. All warps take the code of a kernel function to execute on the GPU. Active warps running on one multiprocessor take turns for execution. A warp that performs memory access causes context switch to execute another warp. Figure 2 depicts warp scheduling on a multiprocessor. In the figure, there are three active *warps* on a multiprocessor.

The maximum number of active warps allowed on a multiprocessor is determined by the hardware resource and the shared resource usage on a multiprocessor. The number of active warps divided by the maximum number of active warps on a multiprocessor is called *occupancy*, which can be obtained from the CUDA Occupancy Calculator [1] for any GPU program at compilation time. Therefore we can obtain the estimated number of active warps for a GPU program at compilation time.

Notation	Description
T	The execution time of a kernel function.
T_a	The execution time of a set of active warps.
T_l	The total time of loading data from the device memory to registers or the on-chip local memory in the active warps on a multiprocessor.
T_s	The total time of storing data from registers or the on-chip local memory to the device memory in the active warps on a multiprocessor.
T_c	The total time of all context switchs in the active warps on a multiprocessor.
T_o	The total execution time of arithmetic GPU operations in the active warps on a multiprocessor.
T_{over}	The overlap time between memory loads and arithmetic operations in the active warps on a multiprocessor.
N_w	The total number of warps needed for the workload.
N_a	The total number of active warps across all multiprocessors.

Table 2: Notations for the cost model.

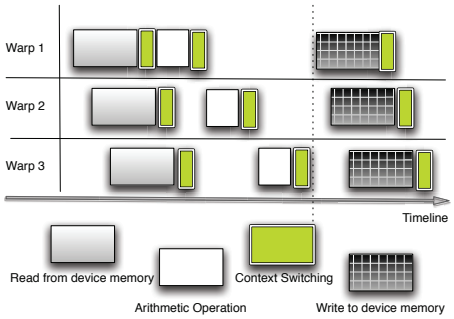


Figure 2: Warp scheduling on a multiprocessor.

Table 2 shows the notation used in the cost model. The cost for a kernel function is calculated in Equation (1). We assume active warps of all multiprocessors complete at the same time. Thus, we multiply the cost of all active warps across the GPU by the number of times that inactive warps become active (N_w/N_a).

$$(1)\ T = T_a * (N_w/N_a)$$

Equation (2) calculates the cost for one multiprocessor's active warps. First, we sum up the cost of memory accesses, the cost of arithmetic operations, and the cost of context switches. Next, we subtract the overlap between memory loads and arithmetic operations. This calculation takes into account the thread parallelism and the memory latency.

$$(2)\ T_a = T_l + T_s + T_o + T_c - T_{over}$$

Since the kernel functions of the nine compression schemes are simple, we can easily obtain the number of memory accesses and the number of arithmetic operations by examining the assembly source code, namely the PTX code of CUDA. One subtle point is that, when estimating the time cost of memory accesses, we need to consider whether the access pattern is coalesced or not. For non-coalesced access, its time cost is worse than coalesced access by a rather fixed factor dependent on the GPU model. In our experiments with different GPU models, this factor is in the range of 2 to

[1] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

16. We then pick the specific factor value for a given GPU model and multiply T_l or T_s by this factor value for non-coalesced access.

5. EVALUATION

We implemented the GPU-based compression schemes in CUDA 2.3, and integrated our techniques into both GDB [18] and MonetDB 5 [7]. We implemented the compression planner in Prolog, a general purpose logic programming language. We start with evaluating the efficiency of compression and decompression of individual schemes on the GPU, and then the effectiveness of the compression planner. Finally, we evaluate the efficiency of query processing with cascaded compressions on GDB and MonetDB.

We conducted experiments on a Fedora 11 Linux PC with an Intel Core2 Quad CPU running at 2.4 GHz, 2 GB RAM, and an NVIDIA GTX 295 graphics card. According to our measurement, the peak bandwidth of the PCI-E bus was 3.2 GB/s, that of the main memory 2.5 GB/s, and that of the device memory 94.3 GB/s.

5.1 Performance of Compression Schemes

We used both TPC-H and synthetic data, stored in MonetDB, to evaluate the GPU-based compression schemes.

We used columns from the TPC-H lineitem table at the scaling factor 10, containing about 60 million tuples. Table 3 shows the compression schemes on the columns in the lineitem table. We concatenated four replicas of the original l_returnflag column with 60 million char-type values, into the current l_returnflag column, in order to have roughly the same data size as the other columns (230 MB). We show the compression and decompression performance of these columns in both measurement and estimation. The performance in this experiment excludes the data transfer on the PCI-E bus, as it is about compression and decompression only; the overall query co-processing performance on compressed data is evaluated in a separate set of experiments and includes data transfer. Additionally, to understand the performance difference among these compression schemes, we calculate the *access size* to include input data, output data, and intermediate data.

We calcuate the (de)compression bandwidth as S_{uc}/T_r, where S_{uc} is the input data size, and T_r is the running time. The bandwidth of GPU-based compression and decompression is up to 45 and 56GB/s respectively. Note, the previous study [28] delivers a peak compression bandwidth of 4.3 GB/s on an Intel Pentium 4 Xeon 3 GHz CPU. Our GPU-based compression schemes achieved an order of magnitude higher bandwidth than their CPU-based counterparts. Compared with the actual performance, the estimated performance using our cost model was quite accurate in most cases.

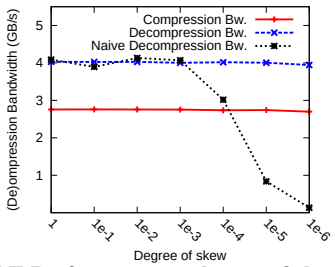


Figure 3: RLE Performance as degree of skew decreases.

We next evaluated our skew handling technique in RLE. The naive implementation used each GPU thread to uncompress a run. We generated a synthetic sorted column of 50 million 4-byte integers, with a given *degree of skew* [21]. The degree of skew is defined as $L_{unskewed}$ / L_{skewed}, where L_{skewed} is the length of the skewed run and $L_{unskewed}$ is the length of the unskewed run. The synthetic column includes a skewed run of length L_{skewed} and the other runs of the same length $L_{unskewed}$, as unskewed. We set L_{skewed} greater than $L_{unskewed}$ so that the degree of skew falls in the range (0.0, 1.0). The less the degree of skew value is, the more skewed the data is. Figure 3 shows our skew-insensitive RLE implementation significantly outperforms the naive implementation in decompression, as the data becomes more skewed.

5.2 Compression planner

	l_partkey		p_partkey	
Compression Plan	**Compression Ratio**	**Generated by our Planner?**	**Compression Ratio**	**Generated by our Planner?**
A: RLE	6.68%	yes	200%	no
B: NS	100%	no	75%	yes
C: DELTA, NS	25%	yes	25%	yes
D: RLE, [ϵ | NS]	4.16%	yes	125%	no
E: RLE, [[DELTA, NS] | NS]	1.67%	yes	50%	no
F: DELTA, RLE, [NS | NS]	3.33%	no	0.0000625%	yes

Table 4: Compression Plans on l_partkey and p_partkey.

We next evaluated the effectiveness of the compression planner, specifically the tactical planner. We generated compression plan candidates of l_partkey of the lineitem table and p_partkey of the part table. Both columns are sorted, with l_partkey containing duplicates while p_partkey containing unique values. We set the maximum number of schemes in a plan to six. The tactical planner automatically generated 17 candidates for l_partkey, and 8 for p_partkey, compared with 597,870 possible combinations. Table 4 shows some of the candidates generated and some of those that were pruned. The column *generated by our planner* indicates whether the compression plan is generated by the tactical planner. The tactical planner generated the candidates with the lowest compression ratios for each column.

5.3 Query Processing with Cascaded Compression

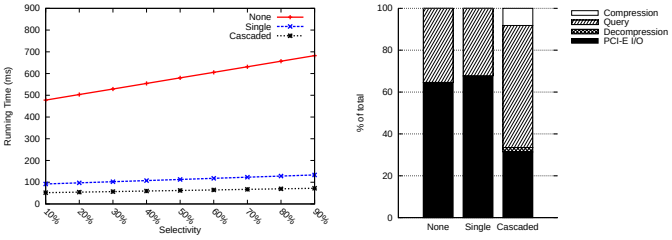
We evaluated the query processing performance of both GDB and MonetDB with cascaded compression. First, we generated a synthetic column α with 200 million sorted 4-byte integers, of an average run length of 10, and performed a table-scan on this column with selectivity varied from 10% to 90%. In this experiment, RLE achieved a compression ratio of 20% on α, whereas the compression plan "RLE, [[DELTA, NS] | NS]" 5%. Next, we evaluated two TPC-H queries.

5.3.1 *Table-scan on Synthetic Data*

In Figure 4(a), the table-scan on uncompressed data (NONE) has the worst performance, because the amount of data transfer from main memory to the device memory is large and so is the costly device memory access to the data. In particular, with NONE the uncompressed data has to be split into multiple chunks, with each fit in the device memory, in query processing. Compared with NONE, SINGLE transfers compressed data, which is only 20query can be directly evaluated on the compressed data, which results in much fewer memory accesses. Finally, in CASCADED the amount of data transfer is the least, only 5scheme in CASCADED requires to partially decompress the data on the GPU. As a result, SINGLE outperforms NONE by a factor of 5 to 6 whereas CASCADED outperforms SINGLE only up to a factor of 2. In the time breakdown in Figure 4(b), CASCADED spent the least portion of the

				Compression			Decompression		
Scheme	**Column**	**Type**	**Ratio**	**Access size**	**Running Time**	**Estimated Time**	**Access size**	**Running Time**	**Estimated Time**
NS	l_quantity	int(4 bytes)	25%	286 MB	5.09 ms	4.51 ms	286 MB	4.07 ms	3.83 ms
NSV	l_quantity	int(4 bytes)	31%	415 MB	20.14 ms	19.19 ms	415 MB	18.36 ms	17.56 ms
DICT	l_shipmode	string(8 bytes)	50%	-	-	-	687 MB	12.07 ms	11.72 ms
RLE	l_partkey	int(4 bytes)	6.6%	938 MB	86.70 ms	65.10 ms	702 MB	53.36 ms	32.05 ms
BITMAP	l_returnflag	char(1 byte)	37.5%	315 MB	14.78 ms	25.03 ms	315 MB	12.54 ms	11.23 ms
FOR	l_shipdate	date(4 bytes)	-	458 MB	5.35 ms	5.02 ms	458 MB	5.39 ms	5.02 ms
DELTA	l_partkey	int(4 bytes)	-	458 MB	9.26 ms	8.62 ms	458 MB	10.31 ms	8.72 ms
SEP	l_discount	float(4 bytes)	-	687 MB	8.81 ms	7.80 ms	687 MB	10.72 ms	8.13 ms
SCALE	l_discount	float(4 bytes)	-	458 MB	5.40 ms	4.82 ms	458 MB	5.58 ms	5.21 ms

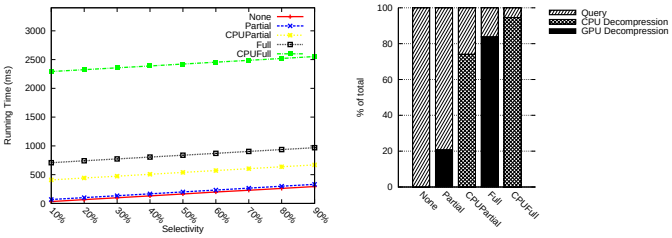
Table 3: Schemes on lineitem columns. MonteDB natively stores textual data by DICT, so there is no compression for DICT.



(a) Overall table-scan performance. (b) Time breakdown of GDB table-scan, with selectivity 40%.

Figure 4: GDB table-scan performance on synthetic data. NONE: on uncompressed data; SINGLE: on RLE-compressed data; CASCADED: on data compressed by the plan "RLE, [[DELTA, NS] | NS]".

total time on PCI-E I/O and the most on query processing among the three schemes. This breakdown suggests that it is worthwhile to trade more computation for less PCI-E I/O and device memory access in GPU-based query processing on compressed data.



(a) Overall table-scan performance. (b) Time breakdown of MonetDB table-scan, with selectivity 40%.

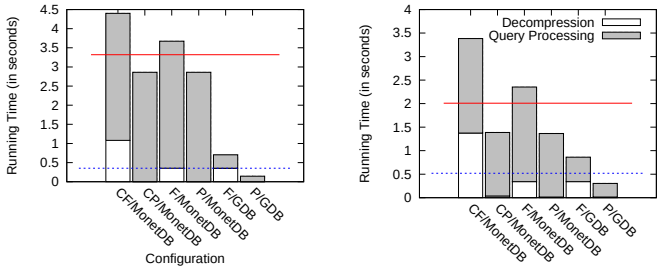
Figure 5: MonetDB table-scan performance on synthetic data. None: on uncompressed data; Partial: partial decompression on the GPU; CPUPartial: partial decompression on the CPU; Full: full decompression on the GPU; CPUFull: full decompression on the CPU.

Next we evaluate the cascaded compression with the CPU-based query processing in MonetDB. The synthetic column is compressed by the plan "RLE, [[DELTA, NS] | NS]". Figure 5(a) compares the table-scan performance of MonetDB in five cases. It shows that only with GPU-based partial decompression can the CPU-based query processing in MonetDB achieve a performance similar to the native MonetDB on uncompressed data. CPU-based or GPU-based full decompression, or CPU-based partial decompression all slows down the native MonetDB. Nevetheless, if compression is used in MonetDB, it is worthwhile to offload the decompression to the GPU since both GPU-based partial and full decompression significantly outperform their CPU-based counterparts, and in turn improve the overall table-scan performance.

5.3.2 TPC-H Queries

lineitem			
Column	**Lowest-ratio Plan**	**Ratio**	**# of candidates**
l_extendedprice	None	100%	0
l_discount	SCALE, NS	25%	42
l_shipdate	FOR, NS	50%	218
l_partkey	RLE, [[DELTA, NS] | NS]	1.67%	17
l_quantity	NS	25%	6
part			
Column	**Lowest-ratio Plan**	**Ratio**	**# of candidates**
p_partkey	DELTA, NS	25%	8
p_type	DICT, NS	9.9%	6

Table 5: Compression plans on columns in lineitem and part.



(a) Query 6 (b) Query 14

Figure 6: TPC-H Queries on Compressed Columns. F: GPU-based full decompression; P: GPU-based partial decompression; CF: CPU-based full decompression; CP: CPU-based partial decompression.

We experiment on Query 6 and Query 14 (Appendix D) in TPC-H with the scaling factor of 10, on MonetDB and GDB. Query 6 is a table-scan, and Query 14 is a join. Table 5 shows the columns involved in the queries, and the corresponding compression plans and compression ratios. Columns in lineitem and part are sorted on l_partkey and p_partkey respectively. We chose the plan with the lowest compression ratio from candidates generated by the compression planner.

Figure 6 depicts the running time of the two queries each in six different cases on MonetDB and GDB. Query 6 can be evaluated directly on compressed data. In Query 14 l_partkey compressed in "NS, DELTA" can be partially decompressed avoiding decompressing the most costly RLE, but p_partkey has to be fully decompressed. Two horizontal baselines are shown in the figure: the solid line is the running time of MonetDB on uncompressed data, and the

dashed line the running time of GDB on uncompressed data. The running time is broken into decompression time (including PCI-E I/O) and query processing time.

As both queries exhibit the same relative performance among the six cases, we discuss the results in Figure 6 without specifying which query: (1) Both F/MonetDB and CF/MonetDB are worse than the native MonetDB, whereas both CP/MonetDB and P/MonetDB outperform the native MonetDB. This relative performance suggests the importance of partial decompression in CPU-based query processing. (2) F/MonetDB always outperforms CF/-MonetDB, which indicates GPU-based decompression is effective if full decompression in query processing is required. The time breakdown shows that the GPU improves the decompression performance by several times. (3) CP/MonetDB and P/MonetDB have a similar performance in that the partial decompression time is negligible. (4) Full decompression almost doubles the native GDB query processing time whereas the partial compression reduces it by half. In summary, partial decompression improves both CPU and GPU-based query processing performance, and the GPU greatly accelerates decompression.

6. CONCLUSION

GPU co-processing has demonstrated significant performance speedups on main memory databases. This paper further improves the performance of such co-processing with database compression techniques. Compression not only improves the query processing performance, but also alleviates the overhead of data transfer on the low-bandwidth PCI-E bus in GPU co-processing. Taking advantage of the GPU's computation power, we consider cascaded compression, which applies a sequence of lightweight compression techniques. We develop effective space pruning techniques to find suitable compression plans, and utilize partial decompression in query processing for efficiency. Our results demonstrate that our compression schemes can reduce the data transfer overhead by over 90%, and improve the overall query processing performance by an order of magnitude.

Acknowledgement

The authors thank the anonymous reviewers for their insightful suggestions. Bingsheng He was supported by a visiting scholarship at the Chinese University of Hong Kong. This work was supported by grant 616808 from the Hong Kong Research Grants Council.

7. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, 2008.

[3] P. A. Alsberg. Space and time savings through large data base compression and dynamic restructuring. *Proceedings of the IEEE*, 1975.

[4] N. Bandi, C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB*, 2004.

[5] C. Binnig, S. Hildenbrand, and F. Faerber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.

[6] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[7] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, CWI, 2002.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. In *Journal of Parallel and Distributed Computing*, 2008.

[9] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW*, 2009.

[10] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo. Frequent itemset mining on graphics processors. In *DaMoN*, 2009.

[11] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *SIGMOD*, 2006.

[12] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Supercomputing*, 2006.

[13] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.

[14] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Applied Computing*, 1991.

[15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT*, 2008.

[17] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC*, 2007.

[18] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. In *TODS*, 2009.

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

[20] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. In *VLDB*, 2008.

[21] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, 1991.

[22] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[23] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, 2008.

[24] M. Poess and D. Potapov. Data compression in oracle. In *VLDB*, 2003.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, 2005.

[26] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. In *VLDB*, 2009.

[27] R. Wu, B. Zhang, and M. Hsu. GPU-accelerated large scale analytics. Technical report, HP, 2009.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

APPENDIX

A. COMPRESSION-AWARE OPTIMIZER

Scheme	Sorted	Order-Preserving	...
NS	No	Yes	...
NSV	No	No	...
DICT	No	Yes	...
RLE	Yes	Yes	...
BITMAP	No	Yes	...
FOR	No	Yes	...
DELTA	No	No	...
SEP	No	No	...
SCALE	No	Yes	...

Table 6: Compressed-Data-Property Table (CDPT). For RLE, the value column is sorted, but the run-length column is not. We utilize the value column here.

Sorted	Order-Preserving	Table-scan
*	No	Query-inapplicable
*	Yes	Query-applicable

Table 7: Query-applicable Table (QAT) for Table Scan.

In this section, we illustrate how the compression-aware optimizer works, and present an example with a table-scan query operator.

The cascaded compression can achieve a good compression ratio, but it decreases the possibility for query operators to work directly on compressed data [1, 14]. We propose partial decompression execution for query processing on compressed data with cascaded compression.

We develop the compression-aware optimizer to determine query plans that may involve none, partial, or full decompression. In particular, it decides whether to decompress data in certain compression schemes. For each compression scheme, s, we maintain a *Compressed-Data-Property Table* (CDPT), in which each row contains properties of the compressed data after applying s. For a query operator O in the database, we construct a *Query-Applicable Table* (QAT), in which each row contains a combination of compressed data properties and whether such combination allows O to directly work on the corresponding compressed column.

Suppose there is a sequence of compression schemes in a compression plan: S_1, S_2, ..., S_n, which applies on a column accordingly. To test whether the operator O can work on the compressed column, we consider these compression schemes in the reverse order in the compression plan, i.e., S_n, ..., S_2, S_1. In this order, we query CDPT to obtain data properties of the compressed data, and use the properties to query the QAT to obtain the applicability of O on the compression scheme. We represent the applicability as a *Query Applicable bit* (QA-bit), where a QA-bit one means that the operator O can directly work on compressed data encoded by the corresponding compression scheme, and a QA-bit zero otherwise. Finally, we have a n-bit vector indicating the compression schemes that O can operate on without decompression. With the bit-vector, we turn all the bits on the right of the leftmost '0' to be zero, meaning that we need to decompress data by the last several schemes in the compression plan. Thus, query processing is performed with partial decompression upon cascaded compression. For example, a bit-vector 1101, consisting the QA-bits for a plan of four compression schemes, means that O can directly process the data compressed by S_1, S_2, and S_4. We only need to decompress data by the schemes S_4 and S_3 before query processing.

Table 6 shows a Compressed-Data-Property Table (CDPT), containing three properties for the compressed data by different compression schemes. The Sorted property indicates whether the compressed column is sorted or not. The Order-Preserving property indicates whether the compressed column preserves the order of values in uncompressed data, which is necessary for the table-scan operator to apply predicates directly. There are other properties for compressed data. However, a particular query operator may just use some of them. Table 7 shows the Query-Applicable Table (QAT), indicating whether a compressed column with certain properties can be queried directly by the table-scan operator. The asterisk (*) denotes that the property doesn't matter for determining whether to decompress.

Consider a column that is compressed by the plan "SCALE, NS, DELTA, DICT". We initialize a four-bit vector with all zeros. We lookup the CDPT for DICT, obtaining the properties of the compressed data by DICT. Then, we use the properties to query the QAT, knowing that it is query-applicable. Therefore, we record the 4th bit of the bit vector as 1. Next, we lookup the CDPT for the 3rd scheme DELTA, and query the QAT, knowing that it is query-inapplicable. Thus, we set the 3rd bit of the bit vector as 0. We repeat such this process, and obtain a bit vector 1101. Then, we convert all bits after the leftmost bit-0 to 0, and obtain the bit vector 1100. This indicates that we need to decompress the data by DICT first, and DELTA next. At this point, we can directly use the table-scan operator to query the data without further decompression by NS and SCALE.

B. COMPRESSION PLANNER

Notation	Description
RL	Average Run Length
C	The number of distinct values a column.
D	The minimum number of bytes for representing a value.
P	The number of parts that a value can be separated into.
S	Whether or not the column is sorted.
M	The maximum value in the compressed column.
$==$	Equality operator.
$=$	Assignment operator.

Table 8: Notations used in CPT and TPT.

Scheme	Sorted	Run Length	Cardinality	Value domain	Compound
NS	*	*	*	$D < 4$	$P == 1$
NSV	*	*	*	$D < 4$	$P == 1$
DICT	*	*	$50 \leq C \leq 50K$	*	$P == 1$
RLE	$S == Yes$	$RL \geq 4$	*	*	$P == 1$
BITMAP	*	*	$C < 50$	*	$P == 1$
FOR	*	*	*	$D < 4$	$P == 1$
DELTA	$S == Yes$	*	*	*	$P == 1$
SEP	*	*	*	*	$P > 1$
SCALE	*	*	*	*	$P > 1$

Table 9: Compulsory-Property Table (CPT).

In this section, we describe the rules for current implementation of the compression planner. Table 8 shows notations used in this section. The numbers for cardinality in Table 9 and Table 10 are adopted from Abadi et al. [1]. Unless otherwise specified, all examples are on a column containing 4-byte integers.

As discussed in Section 4.1, a compression scheme is applicable to a given column, only when this column possesses compulsory properties. Table 9 shows the compulsory properties for the nine

Scheme	Sorted	Run Length	Cardinality	Value domain	Compound
NS	*	*	*	D = 4	*
NSV	S = No	*	C = ∞	D = 4	*
DICT	*	*	C = ∞	D = CEIL($(\log_2 C)$ / 8)	*
RLE	-	-	-	-	P = 2
BITMAP	S = No	*	C = ∞	D = 4	*
FOR	*	*	*	*	*
DELTA	S = No	*	*	D = CEIL($(\log_2 M)$ / 8)	*
SEP	*	*	*	*	*
SCALE	*	*	*	*	P = 1

Table 10: Transitional-Property Table (TPT).

Column	Sorted	Run Length	Cardinality	Value domain	Compound
RLE value column	S = Yes	RL = 1	*	*	P = 1
RLE length column	S = No	RL = 1	*	D = CEIL($(\log_2 M)$ / 8)	P = 1

Table 11: RLE Transitional-Property Table (RLE TPT).

GPU-based compression schemes. The asterisk (*) means that the property does not matter for the particular scheme.

After applying a scheme on a column, the properties of resulted compressed data are determined according to the Transitional-Property Table (TPT). The TPT of the nine compression schemes is shown in Table 10. The asterisk (*) means the property remains unchanged. Some compression schemes output multiple columns, such as RLE. In this case, the compression scheme would have its local TPT. Table 11 shows the RLE TPT, where the value column and length column have separate rules for compressed data properties.

Let us consider a concrete example. Given a column with a cardinality of 60K, containing 10 million sorted 4-byte integers. The compression planner would output the following compression plan candidates:

1. RLE
2. RLE, [[DELTA, NS] | ϵ]
3. RLE, [[DELTA, NSV] | ϵ]
4. RLE, [ϵ | NS]
5. RLE, [ϵ | NSV]
6. RLE, [ϵ | [FOR, NS]]
7. RLE, [ϵ | [FOR, NSV]]
8. RLE, [[DELTA, NS] | [NS]]
9. RLE, [[DELTA, NS] | [NSV]]
10.RLE, [[DELTA, NS] | [FOR, NS]]
11.RLE, [[DELTA, NS] | [FOR, NSV]]
12.RLE, [[DELTA, NSV] | [NS]]
13.RLE, [[DELTA, NSV] | [NSV]]
14.RLE, [[DELTA, NSV] | [FOR, NS]]
15.RLE, [[DELTA, NSV] | [FOR, NSV]]
16.DELTA, NS
17.DELTA, NSV

For example, the third candidate shows that, RLE is applied on the column first, resulting in a value column and a length column. Then, DELTA and NSV are applied on the value column accordingly, and the length column is not compressed.

Please note that, the rules (CPT and TPT) don't need to be exactly the same as ours, and they would be changed a little bit in reality.

C. INTEGRATION INTO MONETDB

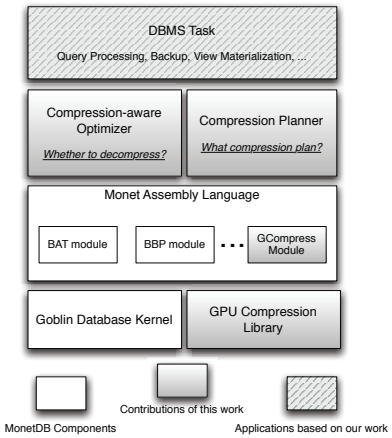


Figure 7: Integration of GPU compression into MonetDB.

Cascaded compression may not be attractive to query processing on the CPU [15], because the computational overhead offsets the gain in the effective usage of the buffer pool as well as the reduced data footprint. That may be the major reason that popular column-oriented databases such as MonetDB [7] and C-Store [25] do not support cascaded compression. In this section, we consider the feasibility of cascaded compression on those DBMSs, through offloading the compression and decompression to the GPU. As a start, we integrate our GPU-based compression techniques into MonetDB.

Figure 7 shows the architecture of MonetDB with the integration of our GPU-based compression techniques. At the bottom level (in parallel with the MonetDB database kernel), we implement the GPU Compression Library, which provides CPU-callable APIs. These APIs wrap GPU-based compression schemes and some GPU-related runtime services, e.g., the GPU memory management. As a caller to the GPU Compression Library, a new *GCompress* Module is added to Monet Assembly Language (MAL) layer. The GCompress module is similar to other MonetDB built-in modules in the MAL layer, providing function wrappers to the upper level components. Specifically, GCompress wraps the GPU Compression Library APIs to the compression planner and the compression-aware optimizer. Through this integration, MonetDB is able to utilize our GPU-based compression techniques, and to offload the compression and decompression to the GPU.

We have implemented both the compression-aware optimizer and the compression planner in Prolog. The advantage of using Prolog is that the implementation is neat and highly extensible. The compression planner for the nine compression schemes contains only around 30 lines of source code.

D. TPC-H QUERIES

We used the following two TPC-H queries in the performance evaluation:

Query 6:

```
select
  sum(l_extendedprice*l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date '[DATE]'
  and l_shipdate < date '[DATE]' + interval '1' year
  and l_discount between [DISCOUNT] - 0.01
      and [DISCOUNT]+0.01
```

			Gzip			GPU Compression			
Column	**Type**	**Sorted**	**Ratio**	**Compression Time**	**Decompression Time**	**Plan**	**Ratio**	**Compression Time**	**Decompression Time**
li_partkey	int(4 bytes)	Yes	1.87%	3.67 s	1.46 s	RLE, [[DELTA, NS] | NS]	1.67%	87.37 ms	53.96 ms
li_quantity	int(4 bytes)	No	25.76%	36.01 s	2.24 s	NS	25%	5.09 ms	4.07 ms
li_discount	float(4 bytes)	No	17.9%	28.58 s	2.02 s	SCALE, NS	25%	10.70 ms	9.82 ms
li_shipmode	string(8 bytes)	No	14.41%	18.82 s	2.36 s	DICT	50%	-	12.07 ms
li_returnflag	char(1 byte)	No	18.77%	36.88 s	2.24 s	BITMAP	37.5%	14.78 ms	12.54 ms

Table 12: Gzip on the columns in lineitem table, with the scaling factor 10.

```
  and l_quantity < [QUANTITY];
```

Query 14:

```
select
  100.00 * sum(case
    when p_type like 'PROMO%'
    then l_extendedprice*(1-l_discount)
  else 0
    end) / sum(l_extendedprice*(1 - l_discount))
    as promo_revenue
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= date '[DATE]'
  and l_shipdate < date '[DATE]'+interval '1' month;
```

E. EVALUATION ON GZIP

As a sanity check, we measured the performance and the compression ratio of Gzip. Table 12 shows the comparison between Gzip and GPU-based cascaded compression. Cascaded compression using lightweight schemes achieved a similar compression ratio to Gzip on numeric data, but were less effective than Gzip on character data. However, the GPU-based compression and decompression was two to four orders of magnitude faster than the CPU-based Gzip on all data types.

F. DISCUSSION

The performance of GPU-CPU co-processing depends on the processing power and memory bandwidths of both processors as well as the transfer bandwidth between the processors. Hardware vendors have been increasing these bandwidths. The memory bandwidths of the CPU and the GPU are both increasing, for example, Recent Intel Core i7 has a peak memory bandwidth of 32 GB/s and NVIDIA GTX480 has a bandwidth 177 GB/s. Even though the contribution of this paper is to enable cascaded compression on the GPU in specific, we believe that the results in this paper are applicable to many-core processors in general.