Data Parallel Bin-Based Indexing for Answering Queries on Multi-Core Architectures

Luke J. Gosink[1], Kesheng Wu[2], E. Wes Bethel[3], John D. Owens[1], and Kenneth I. Joy[1]

1. Institute for Data Analysis and Visualization (IDAV)
One Shields Avenue, University of California, Davis, CA 95616-8562, U.S.A.
{ljgosink, joy}@ucdavis.edu jowens@ece.ucdavis.edu
2. Scientific Data Management Group, Lawrence Berkeley National Laboratory,
1 Cyclotron Road, Berkeley, CA 94720, U.S.A.
kwu@lbl.gov
3. Visualization Group, Lawrence Berkeley National Laboratory,
1 Cyclotron Road, Berkeley, CA 94720, U.S.A.
ewbethel@lbl.gov

Abstract. The multi-core trend in CPUs and general purpose graphics processing units (GPUs) offers new opportunities for the database community. The increase of cores at exponential rates is likely to affect virtually every server and client in the coming decade, and presents database management systems with a huge, compelling disruption that will radically change how processing is done. This paper presents a new parallel indexing data structure for answering queries that takes full advantage of the increasing thread-level parallelism emerging in multi-core architectures. In our approach, our Data Parallel Bin-based Index Strategy (DP-BIS) first bins the base data, and then partitions and stores the values in each bin as a separate, bin-based data cluster. In answering a query, the procedures for examining the bin numbers and the bin-based data clusters offer the maximum possible level of concurrency; each record is evaluated by a single thread and all threads are processed simultaneously in parallel.

We implement and demonstrate the effectiveness of DP-BIS on two multi-core architectures: a multi-core CPU and a GPU. The concurrency afforded by DP-BIS allows us to fully utilize the thread-level parallelism provided by each architecture–for example, our GPU-based DP-BIS implementation simultaneously evaluates over 12,000 records with an equivalent number of concurrently executing threads. In comparing DP-BIS's performance across these architectures, we show that the GPU-based DP-BIS implementation requires significantly less computation time to answer a query than the CPU-based implementation. We also demonstrate in our analysis that DP-BIS provides better overall performance than the commonly utilized CPU and GPU-based projection index. Finally, due to data encoding, we show that DP-BIS accesses significantly smaller amounts of data than index strategies that operate solely on a column's base data; this smaller data footprint is critical for parallel processors that possess limited memory resources (e.g. GPUs).

1 Introduction

Growth in dataset size significantly outpaces the growth of CPU speed and disk throughput. As a result, the efficiency of existing query processing techniques is greatly chal-

lenged [1–3]. The need for accelerated I/O and processing performance forces many researchers to seek alternative techniques for query evaluation. One general trend is to develop highly parallel methods for the emerging parallel processors, such as multi-core processors, cell processor, and the general-purpose graphics processing units (GPU) [4]. In this paper, we propose a new parallel indexing data structure that utilizes a Data Parallel Bin-based Index Strategy (DP-BIS). We show that the available concurrency in DP-BIS can be fully exploited on commodity multi-core CPU and GPU architectures.

The majority of existing parallel database systems work focuses on making use of multiple loosely coupled clusters, typified as shared-nothing systems [5–10]. Recently, a new parallel computing trend has emerged. These type of parallel machines consist of multiple tightly-coupled processing units, such as multi-core CPUs, cell processors, and general purpose GPUs. The evolution of such machines in the coming decade is to support a tremendous number of concurrent threads working from a shared memory. For example, NVIDIA's 8800 GTX GPU–the GPU used in this work–has 16 multiprocessors, each of which supports 768 concurrent execution threads. Combined, these multiprocessors allow the GPU to manage over 12,000 concurrent execution threads. Fully utilizing such thread-level parallelism on a shared memory system requires a different set of query processing algorithms than on shared-nothing systems.

A number of researchers have successfully demonstrated the employment of GPUs for database operations [11–14]. Among the database operations, one of the basic tasks is to select a number of records based on a set of user specified conditions, e.g., "SELECT: records FROM: combustion_simulation WHERE: pressure $>$ 100." Many GPU-based works that process such queries do so with a projection of the base data [11, 15]. Following the terminology in literature, we use the term *projection index* to describe this method of sequentially and exhaustively scanning all base data records contained in a column to answer a query [16]. On CPUs, there are a number of indexing methods that can answer queries faster than the projection index [17–19], but most of these indexing methods do not offer high enough levels of concurrency to take full advantage of a GPU. DP-BIS fully utilizes the GPU's parallelism when answering a selection query; each thread on the GPU is used to independently access and evaluate an individual record. This one-to-one mapping of threads-to-records lets DP-BIS process large amounts of data with 12,000 concurrent parallel operations at any one time.

Though GPUs offer tremendous thread-level parallelism, their utility for database tasks is limited by a small store of resident memory. For example, the largest amount of memory available on NVIDIA's Quadro FX GPU is currently 4.0 GB, which is much too small to hold projections of all columns from a dataset of interest [1–3]. DP-BIS presents one method for ameliorating the challenges imposed by limited GPU memory. The DP-BIS index uses a form of data encoding that is implemented through a multi-resolution representation of the base data information. This encoding effectively reduces the amount of data we must access and transfer when answering a query. As a result of the encoding, we can query dataset sizes that would otherwise not fit into the memory footprint of a GPU. Additionally, by transferring smaller amounts of data when answering a query, we utilize data bus bandwidth more efficiently.

In the DP-BIS approach, we bin the base data for each column. We augment each column's binned index by generating a corresponding Data Parallel Order-preserving

Bin-based Cluster (OrBiC). To resolve a query condition on a column, we first determine the boundaries of the query. Consider an example. For range conditions such as "*pressure* $>$ 100", we determine the bin whose range captures the constraint "100". In this example, assume that the value "100" is contained in the value range captured by bin_{17}. We refer to bins that capture one of the query's constraints as "boundary bins". In our example, records contained in bins less than the boundary bin (i.e. $\text{bin}_0 \rightarrow \text{bin}_{16}$) fail the query. Correspondingly, records contained in bins greater than the boundary bin pass the query. Boundary bin records can't be characterized by their bin number alone, they must be evaluated by their base data value. We call the records in the boundary bin the candidates and the process of examining the candidate values the candidate check [20]. Our strategy for answering a selection query is very similar to that of a bitmap indexing strategy [21–23]. A central difference is that bitmap-based strategies indicate the record contents of each bin with a single bitmap vector. These bitmap vectors can then be logically combined to help form the solution to a query. In contrast, we directly access the bin number for any given record from an encoded data table.

The Data Parallel OrBiC structure we use during our candidate check procedure provides an efficient way to extract and send boundary bin data from the CPU to the GPU. Additionally, this structure facilitates a rapid, concurrent way for GPU threads to access this data. Altogether, to answer a query, we access the bin numbers and the base data values of the records in boundary bins. The total data contained in both these data structures is much smaller than the column projections used by other strategies that employ the GPU to answer a query. Additionally, the procedure for examining the bin numbers and the process of performing the candidate checks offer the same high level of concurrency as the GPU projection index.

In our work we assume that the base data will not (or seldom) be subjected to modification. This assumption too is made by other research database management systems that operate on large data warehouses that contain read-only data: e.g. MonetDB [24], and C-Store [25]. In addition to such database management systems, many scientific applications also accumulate large amounts of data that is never modified or subjected to transactions [26].

Finally, we specifically utilize and emphasize the GPU in our work because it provides some of the highest amounts of thread-level parallelism available in existing multi-core architectures. To this extent we view the GPU as a representative case of where multi-core architectures are evolving with respect to thread-level parallelism and processing performance. In summary, this paper makes the following three contributions.

- We introduce a data parallel bin-based indexing strategy (DP-BIS) for answering selection queries on multi-core architectures. The concurrency provided by DP-BIS fully utilizes the thread-level parallelism emerging in these architectures in order to benefit from their increasing computational capabilities.
- We present the first strategy for answering selection queries on a GPU that utilizes encoded data. Our encoding strategy facilitates significantly better utilization of data bus bandwidth and memory resources than GPU-based strategies that rely exclusively on base data.

– We implement and demonstrate DP-BIS's performance on two commodity multi-core architectures: a multi-core CPU and a GPU. We show in performance tests that both implementations of DP-BIS outperform the GPU and CPU-based projection index with respect to total query response times. We additionally show that the GPU-based implementation of DP-BIS outperforms all index strategies with respect to computation-based times.

2 Background and Related Work

2.1 Related Bitmap Index Work

The data stored in large data warehouses and the data generated from scientific applications typically consists of tens to hundreds of attributes. When answering queries that evaluate such high-dimensional data, the performance of many indexing strategies diminishes due to *the curse of dimensionality* [27]. The bitmap index is immune to this curse and is therefore known to be the most efficient strategy for answering ad hoc queries over such data [28]. For this reason, major commercial database systems utilize various bitmap indexing strategies (e.g. ORACLE, IBM DB2, and Sybase IQ).

Another trait of the bitmap index is that storage concerns for indices are ameliorated through specialized compression strategies that both reduce the size of the data and that facilitate the efficient execution of bitwise Boolean operations [29]. Antoshenkov et al. [21, 22] present a compression strategy for bitmaps called the Byte-aligned Bitmap Code (BBC) and show that it possess excellent overall performance characteristics with respect to compression and query performance. Wu et al. [23] introduce a new compression method for bitmaps called Word-Aligned Hybrid (WAH) and show that the time to answer a range query using this bitmap compression strategy is optimal; the worse case response time is proportional to the number of hits returned by the query. Recent work by Wu et al. [30] extends the utility of the bitmap index. This work introduces a new Order-preserving Bin-based Clustering structure (OrBiC), along with a new hybrid-binning strategy for single valued bins that helps the bitmap index overcome *the curse of cardinality*; a trait where both index sizes and query response time increase in the bitmap index as the number of distinct values in an attribute increases.

Sinha and Winslet [31] successfully demonstrate parallelizable strategies for binning and encoding bitmap indexes, compressing bitmap vectors, and answering selection queries with compressed bitmap vectors. The content of their work focuses on supporting bitmap use in a highly parallel environment of multiple loosely-coupled, shared-nothing systems. In contrast, our work addresses the challenges of supporting bin-based indexing on the newly emerging, tightly-coupled architectures that possess tremendous thread-level parallelism; for example the graphics processor unit (GPU).

The basic attributes of the binned bitmap index (bin-based indexing, the use of simple boolean operators to answer selection queries, etc.) can be implemented in a highly parallel environment. For this reason, our new Data Parallel Bin-based Indexing Strategy (DP-BIS) follows the general structure of a binned bitmap index. Unfortunately, bitmap compression strategies, even the parallelizable strategies of Sinha and Winslet [31], do not support enough concurrency to take advantage of the thread-level parallelism offered by tightly-coupled architectures like GPUs. Thus one of the first

objectives in our work is to develop a compression strategy, based upon the binning techniques of the binned bitmap index, that supports high levels of concurrency and reduces the amount of data required to answer a query.

2.2 Related GPU-Database Work

GPUs have been used to help support and accelerate a number of database functions [11–14, 32, 33], as well as numerous general purpose tasks [34, 35]. Sun et al. [15] present a method for utilizing graphics hardware to facilitate spatial selections and intersections. In their work, they utilize the GPU's hardware-accelerated color blending facilities to test for the intersection between two polygons in screen space.

Working within the constraints of the graphics API for fragment shaders, Govindaraju et al. [11] present a collection of powerful algorithms on commodity graphics processors for performing the fast computation of several common database operations: conjunctive selections, aggregations, and semi-linear queries. This work also demonstrates the use of the projection index to answer a selection query. Additionally, Govindaraju et al. [12] present a novel GPU-based sorting algorithm to sort billion-record wide databases. They demonstrate that their "GPUTeraSort" outperforms the Indy PennySort1 record, achieving the best reported price-for-performance on large databases.

More recent GPU-database work utilizes powerful, new general purpose GPU hardware that is supported by new data parallel programming languages (see Section 3). These hardware and software advances allow for more complex database primitives to be implemented on the GPU. Fang et al. [13] implement the CSS-Tree in the software GPUQP. This work characterizes how to utilize the GPU for query co-processing, unfortunately there is no performance data published about the implementation.

Lieberman et al. [36] implement an efficient similarity join operation in CUDA. Their experimental results demonstrate that their implementation is suitable for similarity joins in high-dimensional datasets. Additionally, their method performs well when compared against two existing similarity join methods.

He et al. [34] improve the data access locality of multi-pass, GPU-based gather and scatter operations. They develop a performance model to optimize and evaluate these two operations in the context of sorting, hashing, and sparse matrix-vector multiplication tasks. Their optimizations yield a 2-4X improvement on GPU bandwidth utilization and 30–50% improvement on performance times. Additionally, their optimized GPU-based implementations are 2-7X faster than optimized CPU counterparts. He et al. [14] present a novel design and implementation of relational join algorithms: non-indexed and indexed nested loops, sort-merge, and hash joins. This work utilizes their bandwidth optimizations [34], and extends the work of Fang et al. [13]. They support their algorithms with new data-parallel primitives for performing map, prefix-scan and split tasks. Their work achieves marked performance improvements over CPU-based counterparts; GPU-based join algorithms are 2-7X faster than CPU-based approaches.

GPU-based strategies that address how to answer a selection query have yet to address the significant limitations imposed by the GPU's small memory, and those imposed by the data buses that transfer data to the GPU. To the best of our knowledge, all relevant literature utilizes algorithms that operate on a column's base data (i.e. non-compressed data). Utilizing base data severely restricts the amount of data the GPU can process. Further, streaming large amounts of base data to the GPU can impede the

processing performance of many GPU-based applications. More specifically, GPU processing performance can rapidly become bottlenecked by data transfer rates if these transfer rates are not fast enough to keep the GPU supplied with new data. This bottleneck event occurs on GPUs whenever the arithmetic intensity of a task is low; the process of answering a simple range query falls into this classification.

In the following sections, we introduce some basic GPU fundamentals, as well as the languages that support general purpose GPU programming. We then introduce our Data Parallel Bin-based Indexing Strategy (DP-BIS) and show how it directly addresses the challenges of limited GPU-memory and performance-limiting bus speeds with a fast, bin-based encoding technique. This work is the first GPU-based work to present such an approach for answering queries. We also show how our binning strategy enables DP-BIS to support a high level of concurrency. This concurrency facilitates a full utilization of the parallel processing capabilities emerging in multi-core architectures.

3 GPUs and Data Parallel Programming Languages

Recent GPU-database works utilize powerful new data parallel programming languages like NVIDIA's CUDA [37], and OpenCL. These new programming languages eliminate the long standing tie of general-purpose GPU work with restrictive graphics-based APIs (i.e. fragment/shader programs). Further, the GPUs supporting these languages also facilitate random read and write operations in GPU memory—scatter I/O operations are essential for GPUs to operate as a general-purpose computational machine.

The functional paradigm of these programming languages views the GPU as a co-processor to the CPU. In this model, the programmer writes two separate kernels for a general purpose GPU (GPGPU) application: code for the GPU kernel and the code for the CPU kernel. Here the CPU kernel must proceed through three general stages.

1. Send a request to the GPU to allocate necessary input and output data space in GPU memory. The CPU then sends the input data (loaded from CPU memory or hard disk) to the GPU.
2. Call the GPU kernel. When the CPU kernel calls a GPU kernel, the CPU's kernel suspends and control transfers to the GPU. After processing its kernel, the GPU kernel terminates and control is transferred back to the CPU.
3. Retrieve the output data from the GPU's memory.

From a high level, the GPU kernel serves as a sequence of instructions that describes the logic that will direct each GPU thread to perform a specific set of operations on a unique data element. The kernel thus enables the GPU direct the concurrent and simultaneous execution of all GPU threads in a SIMT (single-instruction, multiple-thread) workflow. The GPU executes its kernel (step two above) by first creating hundreds to thousands of threads—the number of threads is user specified and application dependent. During execution, small groups of threads are bundled together and dynamically dispatched to one of the GPU's numerous SIMD multiprocessors. These thread bundles are then delegated by the multiprocessor to one of its individual processors for evaluation. At any given clock cycle, each processor will execute the same kernel-specified instruction on a thread bundle, but each thread will operate on different data.

With respect to memory resources, each GPU multiprocessor contains a set of dedicated registers, a store of read-only constant and texture cache, and a small amount of shared memory. These memory types are shared between the individual processors of a multiprocessor. In addition to these memory types, threads evaluated by a processor may also access the GPU's larger, and comparatively slower, global memory.

There are two important distinctions to make between GPU threads and CPU threads. First, there is no cost to create and destroy threads on the GPU. Additionally, GPU multiprocessors perform context switches between thread bundles (analogous to process switching between processes on a CPU) with zero latency. Both of these factors enable the GPU to provide its thread-level parallelism with very low overhead.

4 A Data Parallel Bin-based Indexing Strategy (DP-BIS)

4.1 Overview

To effectively utilize a GPU, an indexing data structure must provide high levels of concurrency to fully benefit from the GPU's large number of concurrent execution threads, and make effective use of the GPU's relatively small memory. In this section we explain our new DP-BIS method and show how it successfully addresses these requirements by integrating two key strategies: data binning (Section 4.2) and the use of Data Parallel Order-preserving Bin-based Clusters (OrBiC) (Section 4.3).

When answering a query, the binning strategy we utilize significantly reduces the amount of data we must access, transfer, and store on the GPU. The Data Parallel OrBiC structure we employ ensures that candidate checks only access the base data of the boundary bins. The concurrency offered by both of these data structures facilitates full utilization of the GPU's thread-level parallelism. In this approach, DP-BIS builds one index for each column in a database, where each index consists of an encoded data table (i.e. the bin numbers), and a Data Parallel OrBiC structure. When answering a simple range query with DP-BIS, we access the encoded data table, and the base data of two bins (the boundary bins) from our data parallel OrBiC structure.

4.2 Base Data Encoding

The index construction process begins by binning all of the base data records contained in a single column. To minimize data skew in our binning strategy, we select the bin boundaries so that each bin contains approximately the same number of records. In cases where the frequency of a single value exceeds the allotted record size for a given bin, a *single-valued bin* is used to contain all records corresponding to this one value. This technique to address data skew is consistent with other binning strategies [30]. We then encode the base data by representing each base data record with its associated bin number. Figure 1 (Step 1) in Section 4.3 shows an example of this encoding. In later discussions, we refer to the bin numbers as low-resolution data and the column's base data as full-resolution data. We always utilize 256 bins in our encoding procedure. As we now show, the amount of data generated by using this number of bins facilitates near-optimal usage of bus bandwidth and GPU memory space when answering a query.

Assume that all full-resolution data is based on 32-bit values, and that there are N records in a given database column. If we use x bits to represent each bin, we can then create 2^x bins where each bin will contain, on average, $\frac{N}{2^x}$ records. The total size of

Number of Bins	*Low-Resolution Size(%)*	*Boundary Bin Size(%)*	*Total Data Size(%)*
$2^{32} = 4294967296$	100.0	0.0	100.0
$2^{16} = 65536$	50.0	$100.0 \times \frac{2}{65536} = 0.003$	50.0 + 0.003 = 50.0
2^8=256	**25.0**	**$100.0 \times \frac{2}{256}$ = 0.78**	**25.0 + 0.78 = 25.8**

Table 1. This table presents the total benefit for DP-BIS to utilize a specific number of bins in its encoding strategy. All values in column two, three, and four are given in terms of a percentage of the total full-resolution data (assuming 32-bits are utilized to represent each full-resolution record). Note the *Boundary Bin Size* reflects the cost for *two* boundary bins. From this table we see that the use of 256 bins reduces the amount of data we must transfer and store by over 74%.

the low-resolution data will then be $x \times N$ bits. The candidate data for each boundary bin, assuming each row-id can be stored in 32-bits, will consist of $\frac{32 \times N}{2^x}$ bits for row-identifiers, and $\frac{32 \times N}{2^x}$ bits for data values. The total number of bits (written as B below) we utilize to answer a simple range query is therefore:

$$B = \underbrace{x \times N}_{Low-Resolution\ Bits} + \underbrace{(4 \times \frac{32 \times N}{2^x})}_{Candidate\ Check\ Bits\ for\ Boundary\ Bins} \tag{1}$$

Note that the candidate check bit cost for boundary bin data is based on two boundary bins; this data size represents the more typical, and expensive, workload for answering a simple range query. Taking the derivative of B with respect to x, we get:

$$\frac{dB}{dx} = N - (128 \times N \times \frac{\ln 2}{2^x}) \tag{2}$$

By setting this derivative to 0 and solving for x, we compute the optimal number of bits to use for our strategy:

$$B_{min} = 7 + \log_2(\ln(2)) \approx 6.4712(bits) \tag{3}$$

In our encoding strategy, bins can be represented with either 32-bits, 16-bits, or 8-bits; these are the most easy and efficient data sizes for GPUs and CPUs to evaluate. Use of alternate data sizes, like the "optimal" 6-bit data type we have derived in Equation 3, are not convenient for GPU-processing. The closest integer type that is conveniently supported on the GPU is the 8-bit integer. Therefore we use 8-bit integers to represent bin numbers and we utilize 256 bins in our encoding strategy.

Table 1 illustrates the benefit of using 256 bins from a less formal standpoint. This table shows realized data transfer costs for answering a simple range query based on data types efficient for CPU and GPU computation. The last row validates Equation 3; 8-bit bins provide the best encoded-based compression by reducing the amount of data that must be accessed, transferred, and stored on the GPU by over 74% .

4.3 Extending OrBiC to Support Data Parallelism

Wu et al. [30] introduce an Order-preserving Bin-based Clustering (OrBiC) structure; this structure facilitates highly efficient candidate checks for bitmap-based query evaluation strategies. Unfortunately, the OrBiC structure does not offer enough concurrency

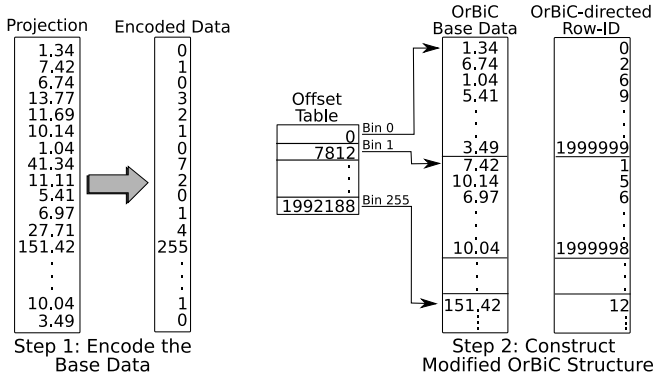

Fig. 1. This figure shows the two-step DP-BIS index construction process. The first step encodes a column's full-resolution base data (Section 4.2). The second step (Section 4.3) utilizes this same full-resolution information to generate a modified (i.e. Data Parallel) OrBiC Structure.

to take advantage of the GPU's parallelism. In this section, we present the constructs of the original OrBiC data structure, and then address how we extend this index to provide greater levels of concurrency.

In the approach presented by Wu et al., the full-resolution data is first sorted according to the low-resolution bin numbers. This reordered full-resolution table is shown as the "OrBiC Base Data" table in Figure 1. In forming this table, each bin's start and end positions are stored in an offset table. This offset table facilitates contiguous access to all full-resolution data corresponding to a given bin.

We extend the work of Wu et al. by building an OrBiC-directed table; this is the "OrBiC-directed Row-ID" table in Figure 1. This table holds row-identifier information for the full-resolution data records. The appended "directed" statement refers to the fact that the ordering of this table is directed by the ordering of the OrBiC Base Data table. With consistent ordering between these tables, start and end locations for a given bin in the offset table provide contiguous access to both the full-resolution data contained in this bin and the correct row-identifier information for each of the bin's records.

The OrBiC-directed row-identifier table facilitates data parallelism by addressing a fundamental difference between our data parallel bin-based strategy and the bitmap work of Wu et al. [30]. Specifically, Wu et al. create a single bitmap vector for each bin in the OrBiC Base Data table. As the bitmap vector associated with a given bin stores the bin's row-identifier information implicitly, their procedure does not need to keep track of the row-identifiers. In our case such a strategy is not inherently parallelizable. We thus employ an explicit representation of the row-identifier information by storing them in the OrBiC-directed Row-ID table. Using this table, threads can simultaneously and in parallel perform candidate checks on all records in a given boundary bin.

4.4 DP-BIS: Answering a Query

In this work, we focus on using DP-BIS to solve simple and compound range queries. Range queries in general are a common database query expressed as a boolean combination of two simple predicates: $(100.0 \leq X)$ AND $(X \leq 250)$, or alternatively $(100.0 \leq$ $X \leq 250)$. Compound range queries logically combine two or more simple range queries using operators such as AND, and OR: $(X \leq 250)$ AND $(\ Y\ \leq 0.113)$.

Algorithm 1
GPU Kernel for Low-Resolution Data

Require: **Integer** lowBinNumber, **Integer** highBinNumber, **Integer []** lowResolution

```
1:  position← ThreadID
2:  binNum← lowReslution[position]
3:  if (binNum>lowBinNumber) then
4:      if (binNum<highBinNumber) then
5:          Sol[position] ← TRUE
6:      end if
7:  end if
8:  if (binNum<lowBinNumber) then
9:      Sol[position] ← FALSE
10: end if
11: if (binNum>highBinNumber) then
12:     Sol[position] ← FALSE
13: end if
```

Algorithm 2
GPU Kernel for Candidate Checks

Require: **Float** lowReal, **Float** highReal, **Float []** fullResolution, **Integer []** rowID

```
1:  position ← ThreadID
2:  recordVal ← fullReslution[position]
3:  record_RowID ← rowID[position]
4:  if (recordVal>lowReal) then
5:      if (recordVal < highReal) then
6:          Sol[record_RowID] ← TRUE
7:      end if
8:  end if
9:  if (recordVal< lowReal) then
10:     Sol[record_RowID] ← FALSE
11: end if
12: if (recordVal>highReal) then
13:     Sol[record_RowID] ← FALSE
14: end if
```

Strategies that answer range queries efficiently and rapidly are a crucial underpinning for many scientific applications. For example, query-driven visualization (QDV) integrates database technologies and visualization strategies to address the continually increasing size and complexity of scientific data [38–40]. In QDV, large data is intelligently pared down by user-specified selection queries, allowing smaller, more meaningful subsets of data to be efficiently analyzed and visualized.

Simple Range Queries The DP-BIS process for answering a simple range query consists of three stages: load necessary input data onto the GPU, execute the GPU kernel, and download the output data (i.e. the query's solution) from the GPU to the CPU. The input for this process consists of a single low-resolution database column, all necessary full-resolution record and row-identifier data, and two real values that will be used to constrain the column. The process returns a boolean bit-vector—a boolean column with one entry per data record that indicates which records have passed the query.

Given a query, the CPU kernel first accesses the appropriate low-resolution data column from disk. Next, space is allocated in GPU memory to hold both this data as well the query's solution. After allocating memory, and sending the low-resolution data to the GPU, the CPU kernel proceeds by identifying the boundary bins of the query. The query's boundary bins are the bins whose ranges contain the query's real-valued constraints. The CPU kernel uses these bin numbers as an index into the OrBiC offset table. Values in the offset table provide the start and end locations in the OrBiC Base Data, and Row-ID tables for the candidate record's full-resolution data and corresponding row-identifiers. After the candidate data is sent to the GPU, the CPU kernel then calls the necessary GPU kernels.

The first GPU kernel, shown in Algorithm 1, processes the column's low-resolution data. In setting up this kernel, the CPU instructs the GPU to create one thread for each record in the column. The CPU then calls the GPU kernel, passing it the boundary bin numbers; these boundary bin numbers enable threads to answer an initial low-resolution query. At launch time, each thread first determines its unique thread identifier[1]. Threads use their identifier to index into the *lowResolution* data array (line 2); this array is the

[1] Each GPU thread has a unique ID that aids in coordinating highly parallel tasks. These unique IDs form a series of continuous integers, $0 \rightarrow maxThread$, where maxThread is the total thread count set for the GPU kernel

Logic-Based *Kernel*	*Algorithm 1 line 1.4, and Algorithm 2 line 2.5 change to:*	*Algorithm 1 line 1.7, and Algorithm 2 line 2.8 change to:*
AND	"*Sol*[x] ← *Sol*[x]"	"*Sol*[x] ← **FALSE**"
OR	"*Sol*[x] ← **TRUE**"	"*Sol*[x] ← *Sol*[x]"

Table 2. This table shows the required changes to make to Algorithms 1 and 2 to form the logic-based kernels DP-BIS uses to answer a compound range queries.

low-resolution data column loaded earlier by the CPU. The thread characterizes its record as passing or failing depending on whether the record's bin number lies interior, or exterior to the boundary bins (lines 3, 4, 8, and 9). The answer to each thread's query is written to the query's solution space in GPU memory. This space, previously allocated by the CPU kernel, is shown in Algorithm 1 as *Sol*[].

The next GPU kernel, shown in Algorithm 2, performs a candidate check on all records contained in a given boundary bin. In our approach we launch the candidate check kernel twice: once for the lower boundary and once for the higher boundary bins.

The candidate check kernel is similar to the previous GPU kernel. Thread identifiers enable each thread to index into the *Full-Resolution* and *rowID* arrays of their respective boundary bin; these arrays are the OrBiC tables previously loaded onto the GPU. These arrays enable the kernel's threads to access the full-resolution data and corresponding row-identifier information for all records that lie in the boundary bin. Threads characterize each record as passing or failing based on comparisons made with the accessed full resolution data (lines 4, 5, 9, and 12 in Algorithm 2). The results of these logical comparisons are written to *Sol*[], ***not*** using the thread's identifier as an index, but the accessed row identifier (obtained from *rowID*) corresponding to the evaluated record.

Compound Range Queries From a high level, we answer a compound range query by logically combining the solutions obtained from a sequence of simple range queries. To perform this task efficiently, we direct each simple query's kernel to utilize the same solution space in GPU memory. The compound range query's solution is produced once each simple query has been answered. In more complicated cases, e.g. "(X1 AND X2) OR (X3 AND X4)", the solution to each basic compound query can be written to a unique bit in the GPU's solution space; the bits can then be combined in each GPU kernel as needed (through bit-shifts) to form the solution to the query.

From a lower level, DP-BIS answers the first simple range with the kernels outlined in Algorithms 1 and 2. These kernels perform unconditional writes to the compound range query's solution space. More specifically, all threads "initialize" this solution space with the first simple range query's solution. All subsequent simple range queries, however, utilize logic-based (AND, OR, etc.) derivatives of these kernels. These logic-based kernels only differ from the kernels outlined in Algorithms 1 and 2 in the final write operation each thread performs (lines 5, 9, and 12, and lines 6, 10, and 13 respectively). These changes, shown in table 2, ensure that each thread, logically combines the current simple range query's solution with the existing compound range query's solution. Section 6.2 demonstrates the implementation and performance of this approach.

	Column Size (-in millions of rows-)						
	50	*100*	*145*	*200*	*250*	*300*	*350*
Base Data Size (-in MB-)	200	400	580	800	1000	1200	1400
DP-BIS Index Size (-in MB-)	450	900	1305	1800	2250	2700	3150
Index Build Time (-in minutes-)	1.22	2.65	4.12	5.82	7.49	9.37	11.36

Table 3. This table shows the index sizes and DP-BIS index build times for each column used in our tests. The size for the DP-BIS index includes the size for the encoded data table, as well as the size for the OrBiC base and row identifier data tables. All times represent an average build time calculated from ten test builds.

5 Datasets, Index Strategies, and Test Setup

In this section we describe the datasets and index strategies we use in our performance analysis. We discuss testing parameters at the end of this section.

All tests were run on a desktop machine running the Windows XP operating system with SP2. All GPU kernels were run utilizing NVIDIA's CUDA software: drivers version 1.6.2, SDK version 1.1 and toolkit version 1.1. Our hardware setup consists of an Intel QX6700 quad-core multiprocessor, 4 GB of main memory, and a SATA storage system that provides 70 MB/s sustained data transfer rates. The GPU co-processor we use is NVIDIA's 8800GTX. This GPU provides 768 MB of memory and can manage over 12,000 concurrent execution threads.

5.1 Datasets

We use two datasets in our analysis. The first dataset is produced by a scientific simulation modeling the combustion of hydrogen gas in a fuel burner. This dataset consists of seven columns where each subsequent column increases in row size: 50 million rows for column one, 100 million rows for column two,. . . , 350 million rows for column seven. We use this dataset in Section 6.1 to measure and compare the effect that increasing column size has on processing and I/O performance.

The second dataset we use is synthetically produced. This dataset consists of 7 columns each with 50 million rows. Each column consists of a series of randomly selected, randomly distributed values from a range of floating point values [-32767.0, 32767.0]. In Section 6.2 we answer a series of compound range queries over this data. This experiment measures and compares the processing and I/O costs of finding the union or intersection between an increasing number of columns.

In both datasets, the records consist of 32-bit floating point data. The time to build the DP-BIS index for each column in our datasets is shown in Table 3; note the cost in time to build the indices scales well with the increasing size of the base data. In this table, the size for the DP-BIS index includes the size for the encoded data table, as well as the size for the OrBiC base and row identifier data tables.

5.2 Index Strategies

In our tests, we evaluate the I/O and processing performance of two indexing strategies: DP-BIS and the projection index. We independently evaluate the concurrency each index affords by implementing and testing the performance of a CPU-based and a GPU-based version of the index.

The CPU-based DP-BIS index is implemented on a multi-core CPU that contains four CPU cores. In this implementation, the work of answering the query is divided separately and equally over each CPU core through the use of Pthreads [41]; here each CPU core is assigned an individual thread and a portion of the DP-BIS low-resolution and full-resolution data to evaluate.

The GPU-based DP-BIS index is implemented on a GPU using the constructs of the data parallel programming language CUDA. This implementation is directly based on the method presented in Section 4.4.

The CPU projection index begins by reading each full-resolution column into CPU memory space. The query is answered by simply performing comparisons on the array(s) without any additional data structure. We use this strategy in our tests because it provides a good baseline for assessing performance.

The GPU projection index is similar to the CPU projection index, with the exception that the full-resolution columns are read into GPU memory space. Additionally, all indexed values in a given column are simultaneously evaluated in parallel by the query. This indexing strategy supports the same level of concurrency offered by DP-BIS (i.e. each thread evaluates a single record), but does not provide the benefits of encoding. On the other hand, this index approach does not require performing candidate checks; a procedure that requires additional computation and read requests to GPU memory.

5.3 Test Setup

To ensure that all queries reflect cold-start, cold-cache behavior, we force all read operations to bypass the OS cache to prevent Windows-based data caching. Therefore, all performance times, unless otherwise stated, are based on the complete time to answer the query. This time measurement, which we refer to as the query's "total performance time", includes:

1. Disk access and data transfer times (including the cost for allocating necessary memory on the CPU and GPU),
2. time to upload data to the GPU (not applicable for the CPU-based index),
3. the time to answer a query on the uploaded data, and
4. the time to download the solution from the GPU to the CPU (again, not applicable for the CPU-based index).

In our performance analysis, we divide this total performance time into two separate time metrics, based on work-related tasks. The first time we refer to as the "I/O performance time". This time includes the time to perform all data transfers and memory allocation: numbers 1, 2, and 4 from the list above. The second time, which we refer to as "processing performance time", includes the time to perform all computation-related work (number 3 from the list above). In our experiments realized total, I/O, and processing performance times are recorded individually, and simultaneously. Finally, unless specified, each reported performance value represents the mean value calculated from 25 separate test runs.

6 Query Performance

Typically, when answering a simple or compound range query over a large amount of data, far more time is spent accessing and transferring data than computing the query's

Indexing Method	*Mean Time Spent Transferring Data -as a percentage of the total time-*	*Mean Time Spent Answering the Query -as a percentage of the total time-*
CPU-Projection	96.70 ± 0.19	3.30 ± 0.19
GPU-Projection	99.48 ± 0.26	0.52 ± 0.26
DP-BIS (GPU)	98.13 ± 1.03	1.87 ± 1.03
DP-BIS (CPU)	93.33 ± 0.7	6.67 ± 0.7

Table 4. This table shows how the total performance time for each index strategy is composed based on I/O-related workloads, and compute-based workloads. Each value in this table represents the mean percentage of time observed for a given index strategy, based upon all tests performed in Figure 2.

solution. The performance of such I/O-intensive tasks are commonly limited by data transfer speeds. This I/O-based performance bottleneck is an especially significant challenge for multi-core architectures, like GPUs, where processing rates can far exceed bandwidth speeds [37].

We demonstrate in this section how the strategy behind DP-BIS presents one way to ameliorate this I/O-based performance bottleneck. By operating primarily on encoded data, the DP-BIS index significantly reduces the effects of this bottleneck, and uses CPU and GPU memory resources more efficiently. Additionally, the level of concurrency afforded by DP-BIS facilitates a full utilization of the thread-level parallelism provided by both multi-core CPU and GPU architectures. In this section we demonstrate the benefits of this concurrency by directly comparing processing performance times for CPU and GPU-based DP-BIS implementations. From this comparison, we show that the GPU-based implementation accelerates processing performance by a factor of 8X over the CPU-based implementation.

6.1 Answering a Simple Range Query

In our first performance evaluation, each index strategy answers a series of seven simple range queries, where each query operates on one of our scientific dataset's seven columns. In this dataset, the size of each subsequent column increases: 50 million rows, 100 million rows,..., 350 million rows.

In these experiments, we expect both CPU and GPU-based DP-BIS index to answer a simple range query using approximately 75% less time than either the CPU or GPU projection index strategies. We base this expectation on the fact that DP-BIS primarily utilizes 8-bit low-resolution data, whereas the projection index strategies utilize 32-bit full-resolution data. We additionally expect that the GPU-based DP-BIS index will be very competitive, with respect to computational performance, with the GPU projection index. This expectation is based on the fact that both strategies support the same level of concurrency: a one-to-one mapping of threads-to-records.

Analysis Figure 2(a) shows the realized total performance time of each index strategy. These performance times show that both DP-BIS implementations answer queries approximately 3X faster than both the GPU and CPU projection index. Table 4 shows how these total performance times are composed based on I/O and processing performance. Note values in Table 4 represent the average I/O and processing performance times realized for each index strategy based on the performance observed for all columns. Table 4

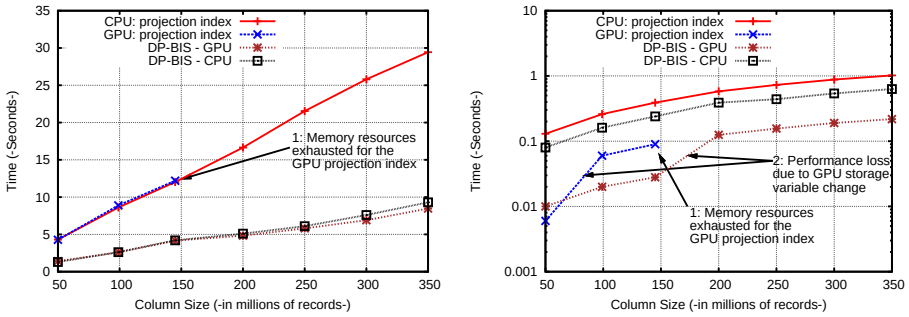

Fig. 2. Here, (a) shows the total performance times for our three indexing strategies. In contrast (b) shows, based on the data from the same test series, *only* the processing performance time for each index. Side by side, these figures show how performance is effected by I/O plus computational workloads versus pure computational work.

confirms the majority of time spent answering a simple range query is used to transfer data; each index uses over 93% of their total performance time for I/O-related tasks.

Note the GPU projection index and GPU-based DP-BIS index support the same level of concurrency when answering a simple range query. When performing this task we know both indexing strategies spend the vast majority of their time transferring data. We conclude that the disparity in total performance time experienced by the GPU projection index is directly attributable to an I/O-based performance bottleneck. This experiment illustrates the benefit of the encoding-based compression utilized by DP-BIS to accelerate the process of transferring data, and therefore the task of answering a selection query.

Aside from the performance benefits offered by DP-BIS, Figure 2(a) also highlights the benefits DP-BIS provides for GPU memory space. The GPU projection index exhausts all memory resources after columns have reached a size of 150 million rows. In comparison, DP-BIS is able to continue answering queries on columns until they reach in excess of 350 million rows. The data encoding DP-BIS utilizes thus provides over 233% better utilization of GPU memory resources when compared to the GPU projection index.

Figure 2(b) shows the processing performance times from our experiment; note that the scale of the y-axis is $\log_{10}$. Label 2 in Figure 2(b) highlights a sharp loss in performance for both the GPU-based projection index (between 50-100 million records) and DP-BIS (between 100-145 million records). This performance loss is due to a GPU implementation detail associated with how the query's solution is written for columns containing in-excess of 95 million (for the projection index) or 145 million (for DP-BIS) rows. Specifically, for columns whose row numbers exceed these values, the GPU projection index and DP-BIS can no longer store the query's solution with a 32-bit variable type (due to limited memory resources); instead an 8-bit variable type is utilized to conserve space. Writing 8-bit data to the GPU's global memory incurs significant performance penalties for both indexing strategies (as Label 2 highlights). Note however that based on the data in Table 4, this processing performance loss minimally impacts the total performance time for either of these two indexing strategies.

Figure 2(b) shows that before this performance loss, the GPU-based DP-BIS index answers queries up to 13X faster than the CPU projection index, 8X faster than the CPU-based DP-BIS index, and (for columns containing more than 95 million records) 3.4X faster than the GPU projection index. After this loss in performance, the GPU-based DP-BIS index outperforms the CPU-based projection and DP-BIS index by 4.9X and 3X respectively.

The concurrency afforded by DP-BIS is evident in comparing the processing performance times for the CPU-based and GPU-based implementations. The GPU-based DP-BIS index answers the query up to 8X faster than the CPU-based implementation. This acceleration in processing performance is a direct consequence of the GPU's increased thread-level parallelism over the multi-core CPU. Accelerated processing performance times are critical for many scientific applications, e.g. query-driven visualization (QDV) [38–40], where data can be presumed to be read once, cached by the OS, and queried repeatedly during analysis stages. In these applications, user workloads are driven more by processing performance times, which make up a larger percentage of the analysis workload, then by disk access times. For these applications, GPU-based implementations of DP-BIS provide significant performance benefits.

6.2 Answering a Compound Range Query

In this second performance evaluation, we use both indexing strategies to answer seven separate compound range queries. The columns our queries evaluate are taken from our synthetic dataset, where each column contains 50 million rows. In this series of tests, each subsequent query will constrain an additional column; in the final test, each index strategy will answer a query constraining seven columns. In this experiment, we perform this series of tests twice: once where we find the intersection, and once where we find the union of all columns queried. We refer to these logic-based series of tests as conjunction ($X_1 \bigwedge X_2 \bigwedge ... \bigwedge X_7$), and disjunction ($Y_1 \bigvee Y_2 \bigvee ... \bigvee Y_n$) tests.

We expect to see some level of disparity in processing performance times between the conjunction and disjunction tests. This expectation is based on the fact that, in our kernels, identifying the intersection between two records requires slightly more computational overhead than identifying their union. We additionally note on these tests that the GPU-based DP-BIS implementation will not require a variable change for the GPU's solution space. More specifically, 50 million rows is a comparatively small solution space and therefore DP-BIS will be able to utilize the more efficient 32-bit data type throughout the entire experiment. As a result, we expect DP-BIS will maintain its performance trend and not suffer the performance drop highlighted by Label 2 in Figure 2(b) from the previous experiment.

Analysis Figure 3(a) shows the total performance times of all index strategies for both the conjunction and disjunction tests. In both experiments, the DP-BIS implementations answer compound range queries some 3–3.7X faster than the projection strategies. Note that Figure 3(a) shows no disparity between the conjunction and disjunction tests; such performance disparities are processing based and are more easily revealed in the processing performance times, shown in Figure 3(b).

Figure 3(b) highlights several important trends. First, as expected, the conjunction tests require more time to answer than the disjunction tests: 5–7% more time for the

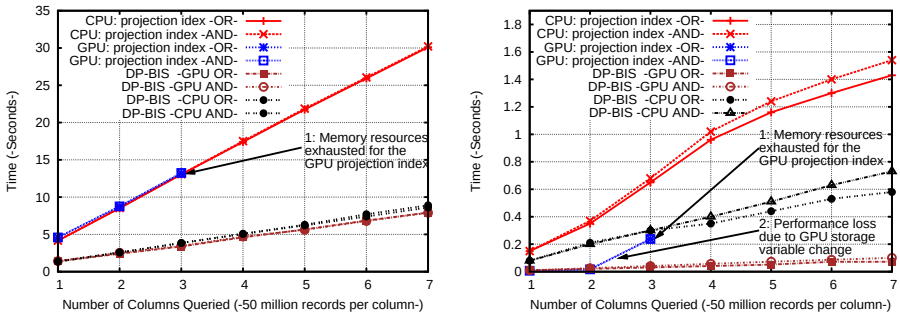

Fig. 3. Here, (a) shows the total performance times for our three indexing strategies when they perform a series of conjunction and disjunction tests. In contrast (b) shows, based on the data from the same test series, *only* the processing performance time for each index.

CPU projection index, and 20–27% more time for the CPU and GPU-based DP-BIS index. This performance trend is not readily seen in the GPU projection index; the lack of data points, due to exhausted memory resources (Label 1), obscures this performance disparity. Label 2 in Figure 3(b) highlights the loss of performance experienced by the GPU projection index due to the variable type change made in the GPU's solution space (see Section 6.1). In comparison, DP-BIS does *not* require such a change to the solution pace. Unlike the experiments performed in Section 6.1 (see Figure 2(b)), the smaller solution space employed by these tests (50 versus 350 million rows) enables DP-BIS to consistently use the more efficient 32-bit variable type. Finally, the processing performance benefits for a GPU-based implementation of DP-BIS are clearly seen in Figure 3(b); the GPU-based implementation of DP-BIS is 8X faster then the CPU-based implementation.

7 Conclusions

In the next decade, the evolution and predominance of multi-core architectures will significantly challenge and change the way data processing is done in the database community. As CPUs rapidly continue to become more like parallel machines, new strategies must be developed that can fully utilize the increasing thread-level parallelism, and thus the processing capabilities, of these architectures.

In presenting DP-BIS, we provide a parallel indexing data structure that will scale effectively with the future increase of processor cores on multi-core architectures. We also provide a parallelizable encoding-based compression strategy that enables DP-BIS to significantly reduce the I/O overhead associated with answering a range query.

We are currently developing a *nested* binning strategy (i.e., binning the records contained in bins) that will enable DP-BIS to provide even further processing and I/O performance benefits. Related to this work, we are additionally optimizing DP-BIS performance with the development of a two-level cache: one cache for the GPU and one for the CPU. This two-level cache will increase DP-BIS I/O performance by caching more frequently used boundary bin data in the GPU cache, and less frequently used boundary bin data in a larger CPU cache. Finally, we are integrating DP-BIS with several scientific data formats (netCDF, and HDF) to generate a new query API. This API will enable users to efficiently generate complex selections on netCDF and HDF datasets.

Acknowledgments

This work was supported by Lawrence Berkeley National Laboratories, and by the Office of Science at the U.S. Department of Energy under Contracts No. DE-AC02-05CH11231 and DE-FC02-06-ER25777 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET), the Institute for Ultra-Scale Visualization, and the Scientific Data Management Center.

References

1. Becla, J., Lim, K.T.: Report from the workshop on extremely large databases (2007)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51** (2008) 107–113
3. Gray, J., Liu, D.T., Nieto-Santisteban, M., Szalay, A., DeWitt, D., Heber, G.: Scientific data management in the coming decade. CTWatch Quarterly (2005)
4. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley (2006)
5. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35** (1992) 85–98
6. Raman, R., Vishkin, U.: Parallel algorithms for database operations and a database operation for parallel algorithms. In: Proc. International Parallel Processing Symposium. (1995)
7. Litwin, W., Neimat, M.A., Schneider, D.A.: LH*—a scalable, distributed data structure. ACM Trans. Database Syst. **21** (1996) 480–525
8. Norman, M.G., Zurek, T., Thanisch, P.: Much ado about shared-nothing. SIGMOD Rec. **25** (1996) 16–21
9. Bamha, M., Hains, G.: Frequency-adaptive join for shared nothing machines. Parallel and Distributed Computing Practices **2** (1999)
10. Rahayu, J.W., Taniar, D.: Parallel selection query processing involving index in parallel database systems. In: ISPAN'02. (2002) 0309
11. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M.C., Manocha, D.: Fast computation of database operations using graphics processors. In: Proc. of SIGMOD. (2004) 215–226
12. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In: Proc. of SIGMOD. (2006) 325–336
13. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V.: GPUQP: query co-processing using graphics processors. In: Proc. SIGMOD. (2007) 1061–1063
14. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proc. SIGMOD. (2008) 511–524
15. Sun, C., Agrawal, D., Abbadi, A.E.: Hardware acceleration for spatial selections and joins. In: Proc. of SIGMOD. (2003) 455–466
16. O'Neil, P.E., Quass, D.: Improved query performance with variant indexes. In: Proc. of SIGMOD. (1997) 38–49
17. Comer, D.: The ubiquitous B-tree. Computing Surveys **11** (1979) 121–137
18. Gaede, V., Günther, O.: Multidimension access methods. ACM Computing Surveys **30** (1998) 170–231
19. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. on Database Systems **31** (2006) 1–38
20. Stockinger, K., Wu, K., Shoshani, A.: Evaluation strategies for bitmap indices with binning. In: DEXA 2004, Zaragoza, Spain. (2004)
21. Antoshenkov, G.: Byte-aligned bitmap compression. In: Proc. of the Conference on Data Compression. (1995) 476

22. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in ORACLE RDB. In: Proc. of VLDB. (1996) 229–237
23. Wu, K., Otoo, E., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proc. of VLDB. (2004) 24–35
24. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: Proc. Conference on Innovative Data Systems Research, Asilomar, CA, USA (2005) 225–237
25. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented dbms. In: Proc. of VLDB. (2005) 553–564
26. Gray, J., Liu, D.T., Nieto-Santisteban, M.A., Szalay, A.S., DeWitt, D.J., Heber, G.: Scientific data management in the coming decade. SIGMOD Record **34** (2005) 34–41
27. Zhang, R., Ooi, B.C., Tan, K.L.: Making the pyramid technique robust to query types and workloads. In: Proc. of ICDE. (2004) 313
28. O'Neil, P.E.: Model 204 architecture and performance. In: Second International Workshop in High Performance Transaction Systems. Volume 359 of Lecture Notes in Computer Science. (1987) 40–59
29. Amer-Yahia, S., Johnson, T.: Optimizing queries on compressed bitmaps. In: Proc. of VLDB. (2000) 329–338
30. Wu, K., Stockinger, K., Shoshani, A.: Breaking the curse of cardinality on bitmap indexes. In: SSDBM. Volume 5069 of Lecture Notes in Computer Science. (2008) 348–365
31. Sinha, R.R., Winslett, M.: Multi-resolution bitmap indexes for scientific data. ACM Trans. Database Syst. **32** (2007) 16
32. Glatter, M., Huang, J., Gao, J., Mollenhour, C.: Scalable data servers for large multivariate volume visualization. Trans. on Visualization and Computer Graphics **12** (2006) 1291–1298
33. McCormick, P., Inman, J., Ahrens, J., Hansen, C., Roth, G.: Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In: Proc. of IEEE Visualization. (2004) 171–178
34. He, B., Govindaraju, N.K., Luo, Q., Smith, B.: Efficient gather and scatter operations on graphics processors. In: Proc. of the conference on Supercomputing. (2007) 1–12
35. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26** (2007) 80–113
36. Lieberman, M.D., Sankaranarayanan, J., Samet, H.: A fast similarity join algorithm using graphics processing units. In: Proc. of ICDE. (2008) 1111–1120
37. NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda` (2007)
38. Bethel, E.W., Campbell, S., Dart, E., Stockinger, K., Wu, K.: Accelerating network traffic analysis using query-driven visualization. In: Proc. of the Symposium on Visual Analytics Science and Technology. (2006) 115–122
39. Stockinger, K., Shalf, J., Wu, K., Bethel, E.W.: Query-driven visualization of large data sets. In: Proc. of IEEE Visualization. (2005) 167–174
40. Gosink, L., Anderson, J.C., Bethel, E.W., Joy, K.I.: Variable interactions in query driven visualization. In: IEEE Trans. on Visualization and Computer Graphics. Volume 13. (2007) 1400–1407
41. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. O'Reilly, 101 Morris Street, Sebastopol, CA 95472 (1998)