Bin-Hash Indexing: A Parallel Method For Fast Query Processing

Luke J. Gosink[#1], Kesheng Wu[*], E. Wes Bethel[%], John D. Owens[#], Kenneth I. Joy[#]

[#]*Institute for Data Analysis and Visualization (IDAV)*
One Shields Avenue, University of California, Davis, CA 95616-8562, U.S.A.
[1]Corresponding author: ljgosink@ucdavis.edu

[*]*Scientific Data Management Group, Lawrence Berkeley National Laboratory,*
1 Cyclotron Road, Berkeley, CA 94720, U.S.A.

[%]*Visualization Group, Lawrence Berkeley National Laboratory,*
1 Cyclotron Road, Berkeley, CA 94720, U.S.A.

***Abstract*— This paper presents a new parallel indexing data structure for answering queries. The index, called Bin-Hash, offers extremely high levels of concurrency, and is therefore well-suited for the emerging commodity of parallel processors, such as multi-cores, cell processors, and general purpose graphics processing units (GPU). The Bin-Hash approach first bins the base data, and then partitions and separately stores the values in each bin as a perfect spatial hash table. To answer a query, we first determine whether or not a record satisfies the query conditions based on the bin boundaries. For the bins with records that can not be resolved, we examine the spatial hash tables. The procedures for examining the bin numbers and the spatial hash tables offer the maximum possible level of concurrency; all records are able to be evaluated by our procedure independently in parallel. Additionally, our Bin-Hash procedures access much smaller amounts of data than similar parallel methods, such as the projection index. This smaller data footprint is critical for certain parallel processors, like GPUs, where memory resources are limited.**

To demonstrate the effectiveness of Bin-Hash, we implement it on a GPU using the data-parallel programming language CUDA. The concurrency offered by the Bin-Hash index allows us to fully utilize the GPU's massive parallelism in our work; over 12,000 records can be simultaneously evaluated at any one time. We show that our new query processing method is an order of magnitude faster than current state-of-the-art CPU-based indexing technologies. Additionally, we compare our performance to existing GPU-based projection index strategies.

I. 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 challenged [1]–[3]. The need for accelerated 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 called Bin-Hash, and demonstrate that its available concurrency can be fully exploited on a commodity GPU.

The majority of existing parallel database systems work focuses on making use of multiple loosely coupled clusters, typified by 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. They support a large number of concurrent threads working from a shared memory. For example, NVIDIA's 8800 GTX Ultra GPU has 16 multiprocessors that can support 768 threads each. To take full advantage of such a system, the application needs more than 12,000 concurrent threads. Fully utilizing such a massively parallel shared memory system requires a different set of query processing algorithms than on shared-nothing systems.

A number of researchers have explored the option of using 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." Most GPU-based works that process such queries, do so with a projection of the base data. Following the terminology in literature, we use the term *projection index* to describe this method of scanning the projection to answer a query [15]. On CPUs, there are a number of indexing methods that can answer queries faster than the projection index [16]–[18], but most of these indexing methods do not offer high enough levels of concurrency to take full advantage of a GPU. The Bin-Hash index fully utilizes the GPU's parallelism; each thread on the GPU is used to independently access and evaluate an individual record during the answering of a query. This one-to-one mapping of threads-to-records affords the maximum level of concurrency available on the GPU, and allows the Bin-Hash index to evaluate over 12,000 records simultaneously at any one time in parallel.

Though GPUs offer tremendous 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 GPUs is currently 1.5 GB, which is much too small to hold projections of all columns from a dataset of interest [1]–[3]. Existing GPU-based works that utilize the projection index to answer a query are thus significantly limited by GPU memory resources. Our Bin-Hash

index presents one method for ameliorating the challenges imposed by limited GPU memory. The Bin-Hash index uses a form of compression, implemented through a multi-resolution representation of the base data information. This compression strategy allows us to query dataset sizes that would otherwise not fit into the memory footprint of a GPU were we to use a traditional projection index strategy.

In the Bin-Hash approach, we bin the base data for each column and generate a spatial hash of column values in each bin. To resolve a query condition on a column, we first access the bin numbers. For range conditions such as "pressure $>$ 100" we determine which bins satisfy the condition and which ones don't, based on the boundaries of the bins. There is only one exception, the bin containing the value 100. We call such a bin a boundary bin. We need to examine the column values for the records in the boundary bin to determine whether they actually satisfy the query condition. We call the records in the boundary bin the candidates and the process of examining the candidate values the candidate check [19]. Altogether, to answer a query, we access the bin numbers and the base data values of the records in boundary bins. The data contained in the summed total of 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 of examining the bin numbers and the process of performing the candidate checks offers the same level of concurrency as the projection index, and achieves excellent performance as we demonstrate later.

The main contributions of our work are the following.

- We introduce the Bin-Hash data structure for accelerating selection queries using GPUs; existing work on processing such queries only uses the projection index which requires the utilization of much more (GPU) memory than the Bin-Hash [11].
- In our performance tests, our approach is shown to outperform the fastest indexing strategy used for our key application, Query Driven Visualization (discussed in the next section); earlier works lack such a direct comparison.
- We demonstrate the utility of the perfect spatial hash as a parallel data structure; in our tests, thousands of threads concurrently and efficiently access partitioned base data on a parallel processor. Additionally, we show how this spatial hashing data structure is essential for our Bin-Hash index to reduce the amount of data needed on the GPU, and to utilize the GPU's parallel processing capability.

II. Query-Driven Visualization

Our work on the Bin-Hash index is motivated by an approach to visual data analysis called Query-Driven Visualization (QDV). The goal of QDV is to provide scientists with resource-efficient and visually interactive methods for exploring large multidimensional data. The basic strategy is to restrict computation and cognitive workloads, by either limiting or prioritizing processing, visualization, and interpretation, to records that have been defined to be scientifically relevant

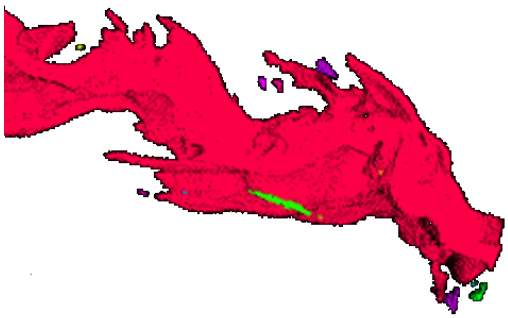


Fig. 1: This QDV image, generated from a combustion analysis dataset, depicts the regions that correspond to both low values of temperature and high concentrations of methane: ($methane > 0.3$) AND ($temperature < 4$) [20]. The image is generated by rendering each record that meets the query's constraints for temperature and methane as a hexahedral cube; coloring of the cubes is based upon connected component labeling.

by the user. The definition for scientifically relevant is provided through user defined Boolean range queries: "SELECT: records FROM: flame WHERE: (temperature $>$ 300) AND (*methane* concentration $>$ 1e-6)", see Figure 1 for an example.

Most scientific datasets contain a high number of variables and numerous time steps; each variable is represented as a single column within a database table, and each time step is a partition within that column. In a typical QDV exercise, several variables are selected for exploratory analysis. In this exploration process, queries are repeatedly and incrementally adjusted by the end-user in order to better understand the relationship between pairs or groups of the selected variables.

As an example, assume an end-user is analyzing a dataset that simulates methane combustion. Further suppose that the end-user is interested in determining how concentrations of methane are distributed spatially throughout a fixed range of pressure ($5000 <$ pressure < 10000) from this combustion data. To assist in this analysis, QDV methods support the query and visualization process with an interactive GUI that lets the end-user specify constraints on columns (i.e. variables) with pairs of slider widgets. Thus the end-user in our example would steadily increment the range constraints for methane concentration (while keeping the constraints for pressure fixed), and observe the records rendered in the visualization.

The software responsible for providing interactive exploration is not responsible for the generation of the data or the indices. This software can treat base data as read-only. Furthermore, it can take advantage of repeated queries on identical variables to cache critical data needed for answering queries. In our software implementation, we use the GPU based Bin-Hash indices to identify records satisfying the query conditions, but use the CPU to retrieve the selected values.

III. Related Work

Query-Driven Visualization (QDV) is an important and effective way to combine database and visualization technologies. As far back as 1994, the VisDB system was proposed to

guide query-formulations with a relevance-based visualization and presentation paradigm [21]. This system ranked data terms according to their relevance to a query, and the top quartile of the most relevant results were then input into a visualization and rendering pipeline. This approach has a complexity of $O(n)$, where n is the number of data items in the dataset. Later systems improved this performance significantly by making use of more efficient database indexing technologies. For example, the work by Stockinger et al. [20] used a compressed bitmap index from FastBit to enable the identification of regions of interest in time complexity of $O(k)$, where k is the number of records that match the search criteria.

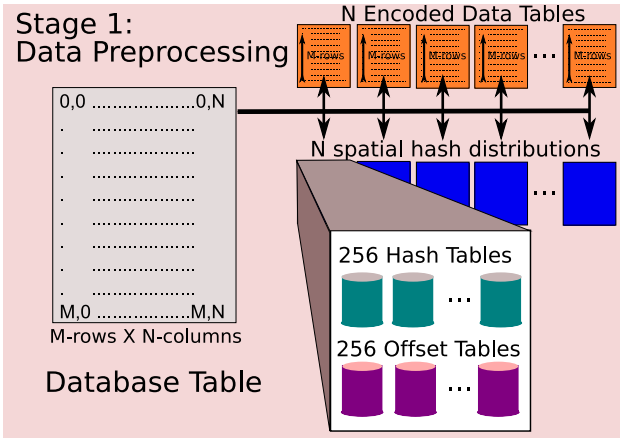
The most efficient strategy for answering ad hoc queries on high-dimensional read-only data is the bitmap index [22]. Given n records with c bin boundaries, the basic binned bitmap index generates c bitmaps with n bits each [19], [23], [24]. Each bit in a given bitmap indicates if the attribute in the record is within the specific range corresponding to the given bin's boundaries. Queries over bitmap indices are processed with bitwise logical operations: AND, OR, NOT, etc.

Storage concerns for bitmap indexing strategies are ameliorated through specialized compression strategies that both reduce the size of the data, and facilitate the efficient execution of bitwise Boolean operations [25]. Antoshenkov et. al [26], [27] 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 [28] 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.

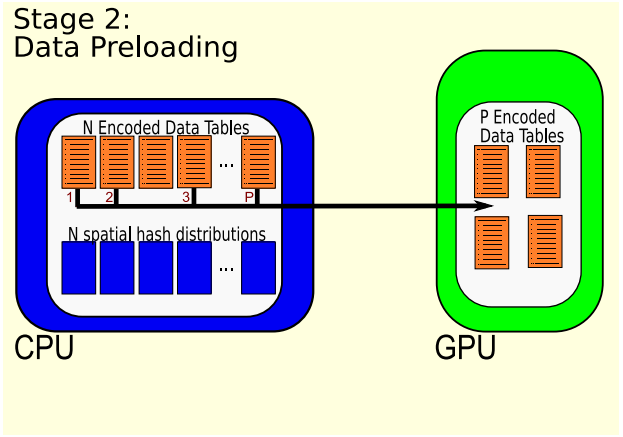
The basic attributes of the binned bitmap index (indexing, query-processing, etc.) are able to be implemented in a highly parallel environment. For this reason, our new Bin-Hash index follows the general structure of a binned bitmap index. Unfortunately the compression strategies for bitmaps do not offer enough concurrency to take advantage of the GPU's massive parallelism. Thus one of the first objectives of our approach is to develop compression strategies, based upon the binning strategies of the binned bitmap index, that simultaneously offer high levels of concurrency, and reduce the amount of data required to answer a query.

GPUs have been used to help support and accelerate a number of database functions [11]–[14], [29], [30], as well as numerous general purpose tasks [31], [32]. The Scout [30] software system provides the ability to perform expression-based queries using a simple data-parallel programming language along with visualization, where both queries and visualization are executed entirely on a GPU. Unfortunately, Scout was limited by then-current hardware that imposed memory constraints, as well as functional constraints: at the time, GPUs only supported gather functionality, and GPU kernels were constrained by restrictive APIs.

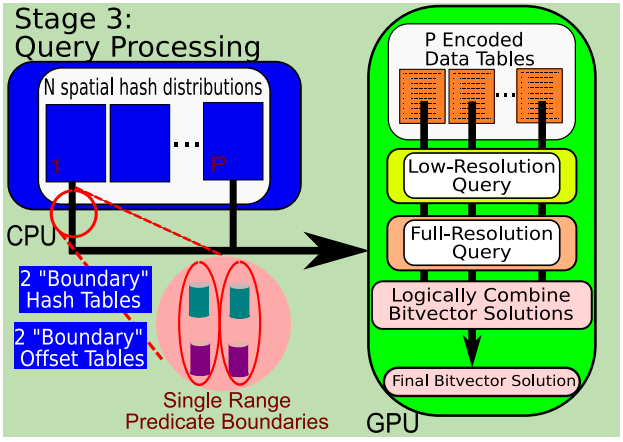
Sun et al. [33] presented a method for utilizing graphics



(a) Bin-Hash Preprocessing Stage

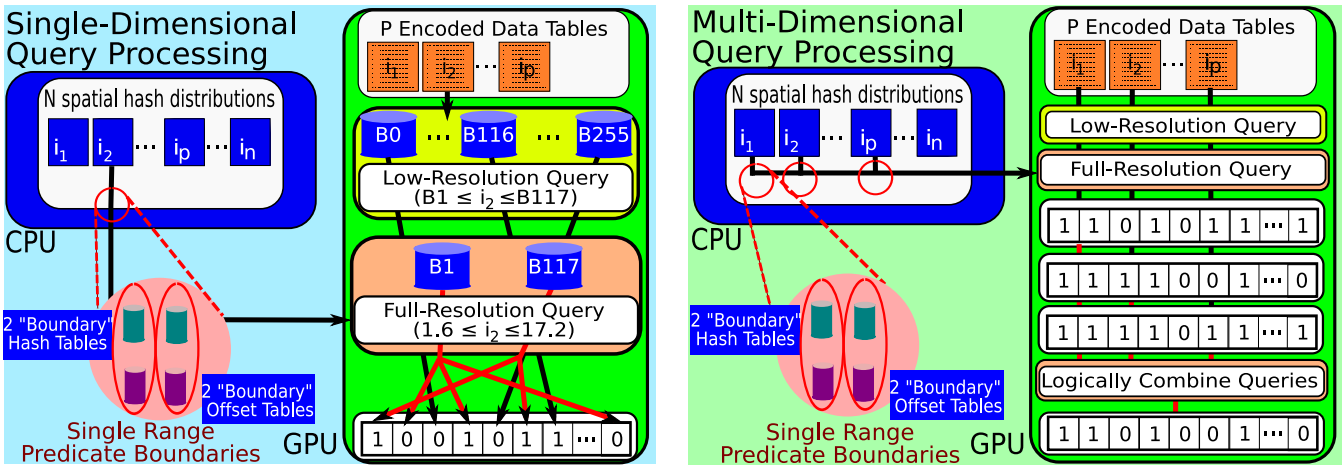


(b) Bin-Hash Preloading Stage



(c) Bin-Hash Query Stage

Fig. 2: In (a), we show the the Bin-Hash preprocessing stage that is performed on the CPU. This preprocess step is only performed once on each full-resolution column in a table. In (b), we show the Bin-Hash preloading stage. Here user-selected data is loaded from the CPU onto the GPU. This step is performed once per interactive session. In (c), we show the actual Bin-Hash query process. Here hash and offset table tuples are streamed to the GPU to assist in resolving full-resolution queries.



(a) (b)

Fig. 3: Illustration of single (a) and multidimensional (b) predicate evaluation. The single-dimensional predicate illustration shows a detailed view of the low and full-resolution query process. For the low-resolution query, the predicate over i_2 is bound by B_1 and B_{117}. Bins interior to the boundary bins write "1" (pass) to the bit-vector solution, where as bins exterior to the boundary bins write "0" (fail). The multidimensional predicate evaluation depicts the logical combining of single-dimensional predicate solutions (in this case logically AND-ing all solutions) to form the final bit-vector solution—this is discussed in Section IV-C.2.

hardware to facilitate spatial selections and intersections. Their approach relies on 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] presented 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 contains a demonstration of utilizing the projection index to answer a selection query.

More recent work on utilizing GPUs for database operations makes use of the data parallel programming language CUDA [13], [14]. This type of approach is more likely to be applicable to other parallel processors. These recent works also address additional database operations, such as join [14] and indexing [13]. In particular, Fang et al. [13] implemented the CSS-Tree in the software GPUQP, however, there is no performance data published about the implementation. Because of the inherent lack of concurrency in tree-based indexing structures, a completely new approach is needed to take full advantage of the massive parallelism in a commodity GPU. Additionally, existing GPU works that evaluate queries with a projection index don't address the significant limitations imposed by limited GPU memory.

IV. The Bin-Hash Method

A. Overview

An indexing data structure that effectively utilizes a GPU must support a high level of concurrency when answering a query and must fit in the relatively small memory of the GPU. One can use the CPU's main memory as the cache for the GPU, however, due to the relatively low bandwidth between the main memory and the GPU, it is crucial to limit the amount of data transferred from main memory to the GPU[1]. Our approach in solving this problem concentrates on implementing an indexing method that reduces both the amount of bandwidth and memory required to evaluate a query. We achieve this goal by integrating two key strategies: data binning to reduce the memory footprint on the GPU, and the combined use of data partitioning with perfect spatial hashing to ensure the candidate checks only access the base data of the boundary bins.

The Bin-Hash approach utilizes a strategy similar to the binned bitmap index [19], [23], [24]. It builds one index for each column of a dataset and each index consists of an encoded data table (which contains the bin numbers), and a set of spatial hash tables, one for each bin. An illustration of this data structure is shown in Figure 2(a). We represent the bin numbers as binary integers. Because computers can operate on 8-bit, 16-bit and 32-bit integers much faster than other arbitrarily sized binary integers, the choices for the number of bins are effectively limited to 2^8, 2^{16} and 2^{32}. Because GPU memory is limited, we limit ourselves to only consider using 2^8 (256) bins in this work. In later discussions, we also refer to the bin numbers as the low-resolution data and the spatial hash tables as full-resolution data.

To minimize data skew in our binning strategy, the bin boundaries are selected such that each bin contains approximately the same number of records. In cases where the frequency of a single value exceeds the alloted record size for a given bin, a single bin is used to contain all records corresponding to this one value. This strategy minimizes the worst case behavior during query processing: all queries should take approximately the same amount of time to answer regardless of the placement of the query's boundary bins. This kind of predictable behavior is also very important

[1]NVIDIA's 8800 GTX Ultra possesses 104 GB/s of on chip bandwidth. In comparison, the bandwidth over 16X PCI Express bus that connects the main memory and the GPU is 4 GB/s one way (or 8 GB/s for bidirectional traffic).

in visualization and real-time applications where the system needs predictable performance from all components in order to respond interactively to the user's requests.

At the start of an exploratory QDV session, we preload low-resolution data into GPU memory as shown in Figure 2(b); this preloading is only performed once per interactive QDV session. From this low-resolution information, each data record can be processed by an initial low-resolution query as illustrated in Figure 3(a)—we differentiate between queries that access low-resolution versus full-resolution data by referring to the former as low-resolution queries, and the latter as full-resolution queries. To evaluate this low-resolution query, we scan the bin numbers and characterize records as: passing the query (i.e., in an interior bin), failing the query (i.e., in an exterior bin), or needing a candidate check (i.e., in a boundary bin). Single range predicates, having at most two constraints, have a maximum of two boundary bins. Thus, only $\frac{2}{256}$ of the records require candidate checks, whereas the rest can be successfully processed with the low-resolution data alone. Note that each record can be examined independently to provide the maximum level of concurrency.

Performing candidate checks requires access to full-resolution data. We retrieve the full-resolution information from main memory (CPU) in the form of spatial hash tables (if they are not resident in the GPUs memory). Only the spatial hash tables corresponding to the boundary bins are uploaded to the GPU; this strategy minimizes the amount of data transferred from CPU main memory. In the Bin-Hash method, the CPU serves only to supply the GPU with spatial hash table data; the CPU performs no processing.

The records that fall into a given boundary bin can correspond to arbitrary row numbers. The candidate check procedure needs access to full-resolution base data. Perfect spatial hashing is a way to associate the row numbers with their base data in a compact data structure. To better match the GPU's typical notion of texture addressing (e.g., 2D (x, y) address of pixels on a screen), we map database row numbers to a 2D virtual array. We present an overview of the spatial hash [34] in Section IV-B.

The processing of low-resolution data and full-resolution data can proceed independently; a query's final results are written to a bit vector where ones indicate records that satisfy the query condition and zeroes otherwise. If multiple query conditions are involved, each can be answered with a different index and the output bitmap can be joined together with the same operators that connect the query conditions. This approach, consistent with many indexing technologies (e.g. FastBit), is illustrated logically in Figure 3(b), as well as discussed in detail in Section IV-C.2.

B. Perfect Spatial Hashing

The concept of perfect spatial hashing for 2D and 3D "spatially sparse" data was first proposed by Lefebvre and Hoppe [34], who extended the general work of Sager et al. [35], [36]. Lefebvre and Hoppe observed that in the universe of possible function space, perfect hash functions (i.e. those hash functions that have no collisions) are exceedingly rare. They further noted that the definition of a *minimal* perfect hash (i.e. a perfect hash function whose hash table contains no unused entries) must require the storage of additional data in the form of an auxiliary look-up table. They proposed the following multidimensional hash function:

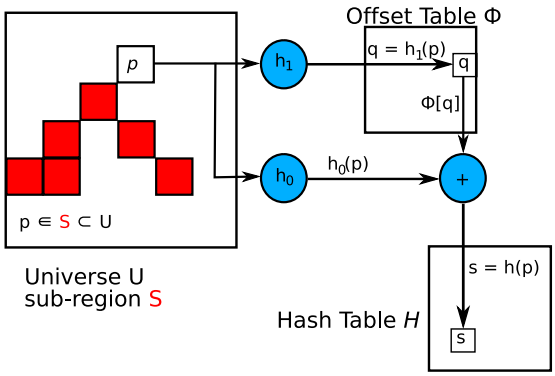


Fig. 4: Illustration of the Lefebvre and Hoppe hash function definition for texture compression. In this illustration the element p is a pixel—in our work we extend this concept to efficiently index records from partitioned database columns. In our method, the element "p" corresponds to the row-id of a given record in a boundary bin from a low-resolution column, and "s" is the full-resolution value associated with the record at this row-id.

$$h(p) = h_0(p) + \Phi(h_1(p)) \quad (1)$$

Here Equation 1 combines two imperfect hash functions, $h_0()$ and $h_1()$, with an offset table Φ to form a minimally perfect spatial hash function, $h()$. This perfect hash function $h()$ retrieves the record value for an element, p, from a hash table, $\mathscr{H}$. The offset table Φ contains collision resolution information that guarantees minimal perfect hashing in $\mathscr{H}$. Remarkably, h_0 and h_1 are defined by Lefebvre and Hoppe as simple modulo operations with respect to the dimensions of the hash table $\mathscr{H}$ and the offset table Φ. This process is illustrated graphically in Figure 4.

1) Utilizing Perfect Spatial Hashing: We utilize the perfect spatial hashing technique in order to create a new indexing method for each full and low-resolution column pair. Specifically, this indexing method provides access to the raw data from the full-resolution column of all records in the low-resolution column corresponding to a given bin value (e.g. a boundary bin). We integrate this spatial hashing indexing into our Bin-Hash strategy by constructing a hash and offset table tuple for all bins in a low resolution column (256 tuples for each low-resolution column: $< \mathscr{H}_0, \Phi_0 >$, ...,$< \mathscr{H}_{255}, \Phi_{255} >$). This is shown graphically in Figure 2(a).

We leave the details of hash and offset table construction to the work presented by Lefebvre and Hoppe [34] as they are beyond the scope of our work to address. Instead we assume the existence of $\mathscr{H}$ and Φ in order to proceed with explaining *how* the hashing function, $h(p)$, is utilized in our new Bin-Hash indexing strategy.

When evaluating a low-resolution query, records are characterized as passing, failing or requiring candidate checks based upon whether the low-resolution column's values are interior,

exterior or equal to the boundary bins. For values that are equal to the boundary bins, we utilize the hash and offset table tuples associated with these boundary bins as follows.

The modulo hash functions described by Lefebvre and Hoppe, $h_0(p)$ and $h_1(p)$, each modulate two degrees of addressing freedom. Specifically, as their original work was implemented for the spatial hashing of textures, each pixel element "p" has its x addressing component modulated, and its y addressing component modulated by $h_0()$ and $h_1()$. We map the record-ID of a given record in a column to the constructs used by the Lefebvre and Hoppe spatial hash function, by calculating (in the CUDA kernel) a virtual 2D address for the record-ID based upon the `ceil()` of the square root of the row count in the column. For illustrative purposes, assume a row count of 20. The `ceil()` of the square root of this row count is 5. Based upon <x,y> pixel positioning, the resulting 2D virtual addresses for select increasing record-IDs would be: record 0 = <0,0>, record 5 = <1,0>, record 20 = <4,0>.

Using these 2D virtual address, the evaluation of the function $h(p)$ in our new Bin-Hash indexing strategy then proceeds as follows. Recall again that **p** is a record-ID that corresponds, in the low-resolution column, to a record whose value is equal to one of the boundary bins of the query.

- We first use **p**'s virtual 2D components to locate offset values in the offset table Φ. The location of these values in Φ is calculated by performing a simple modulo operation on each virtual component of **p** with Φ's respective x and y dimensions. Note that this is effectively the inner half of Equation 1: $\Phi(h_1(p))$.
- Next, **p**'s virtual 2D components are used to locate an initial position in $\mathscr{H}$. The location of this initial position is calculated by performing a simple modulo operation on each virtual component of **p** with H's respective x and y dimensions. Note that this position is not the actual location of the raw full-resolution data associated with this record-ID. To this modulated address is added the offset values provided by Φ from the step above. This—now offset—location in $\mathscr{H}$ is where the unique full-resolution data value of **p** resides. This step is the total of the work expressed in Equation 1.

Figure 4 portrays both of these steps graphically.

Note that the spatial hash procedures used to access the base data values in a boundary bin's spatial hash table are inherently parallel. Thus altogether, to answer a query, each record is evaluated by a single thread that performs an initial low-resolution query and, if necessary, a full-resolution query.

C. Bin-Hash Implementation Details

Sections IV-A and IV-B introduced the Bin-Hash method's binning strategy, and spatial hashing implementation. We now illustrate these combined strategies with a pseudo code example, and discuss the supportive role the CPU plays in supporting the QDV application through the implementation of a dual cache.

Algorithm 1 Kernel for logical OR with one constraint (e.g. x $\geq$ 12)

Require: The variable boundaryBIN has been passed to the kernel with a value between 0-255. Additionally, all elements in **Sol** have been initialized to "fail".

```
1: int position ⇐ thread ID
2: ubyte encode ⇐ BinNum[position]

3: if (encode > boundaryBIN) then
4:     Sol[position] ⇐ true
5: else if (encode == boundaryBIN) then
6:     int pos_y ⇐ position >> 12
7:     int pos_x ⇐ position & 0xfff

8:     int offset_x ⇐ pos_x % sizeOffset
9:     int offset_y ⇐ pos_y % sizeOffset
10:    int offsetVal[2] ⇐ Φ[offset_y][offset_x]

11:    int temp_x ⇐ pos_x + offsetVal[0]
12:    int temp_y ⇐ pos_y + offsetVal[1]

13:        hash_x ⇐ temp_x % sizeHash
14:        hash_y ⇐ temp_y % sizeHash
15:    float actualVal ⇐ ℋ[hash_y][hash_x]

16:    if actualVal ≥ lowestBoundary then
17:        Sol[position] ⇐ true
18:    end if
19: end if
```

1) OR Kernel Example: The pseudo code in Algorithm 1 presents the kernel for a logical "OR" operation (the predicate has 1 constraint). The algorithm demonstrates the application of Sections IV-A and IV-B, explicitly depicting the process for evaluating both low-resolution (lines 2-4), and full-resolution (lines 5-19) queries.

In this code, **BinNum** represents the bin numbers from one low-resolution column; from a technical perspective, it is a one-dimensional array of length equal to the number of rows in the column. As a precondition, all bin data is assumed to have been loaded onto the GPU. Line 6 calculates virtual 2D components (Section IV-B) based upon a given record's row position, and a (illustrative) two-dimensional texture of dimensions 4096 X 4096.

$\mathscr{H}$ and Φ are the respective hash and offset table tuple for the boundary bin determined by the user's constraint. From a technical view, these are two-dimensional arrays where the array dimensions, x and y, are equal. In a perfect spatial hash, the dimensions of the hash table ("sizeHash" in Algorithm 1) are equivalent to the square of the number of rows divided by 256; that is, for the 2D hash tables, x multiplied by y should equal the number of records in a single bin. The offset table dimensions ("sizeOffset" in Algorithm 1) are smaller than this due to the compacting benefits associated with collision resolution information [34].

The solution of the query—a bit-vector—is written to **Sol** as a series of Boolean values indicating which records have passed the query. Thus **Sol** is a one-dimensional array of length equal to the number of rows in the column.

Through CUDA, each thread in the GPU has a unique ID that can be utilized to coordinate highly parallel tasks, such as query evaluation. For clarity we refrain from utilizing CUDA's thread-based constructs and assume that the variable "position" has been initialized with a thread ID. In our implementation, this thread ID corresponds to the index of the record the thread will be accessing in **BinNum**, *and* the index in **Sol** where the solution to the thread's answered query will be written.

2) Multi-Dimensional Kernels: The code in Algorithm 1 is presented as a kernel for evaluating single-dimensional predicates; this kernel can additionally evaluate multidimensional predicates. This is done by having sequential kernels write to the same bit-vector solution space in the GPU's memory. For example, to determine the multidimensional predicate solution resulting from the logical OR-ing of two single-dimensional predicates, two sequential kernels—both utilizing Algorithm 1—are run. In the process, each kernel will indicate the records that have passed their respective predicate's constraints by utilizing the same bit-vector solution space (line 4 and 19 in Algorithm 1). As the original bit-vector solution was initialized to have every record fail (see "**Require:**" in Algorithm 1), only the records having passed one or both of the queries will be indicated as passing in the final bit-vector solution for the multidimensional predicate.

In the case of a logical AND-ing between two single-dimensional predicates, the first single-dimensional predicate is evaluated by the kernel shown in Algorithm 1. The second single-dimensional predicate, however, uses a slightly modified kernel. This new kernel will differ in two ways from the code shown in Algorithm 1: records failing the query will now write out "0", and any record that passes the query will write out a "1" if and only if the previous kernel also wrote a "1" for this position (i.e. line 4 and 19 in Algorithm 1 will be changed to read "**Sol**[position] = **Sol**[position]" in the modified kernel).

From these two cases, it is possible to extend this logic to more complicated multidimensional queries. In such cases the CPU will break down the query to its basic operations and queue the appropriate kernels and necessary hash and offset tables on the GPU. The final result, as with the single-dimensional predicates, will be a bit-vector solution.

3) CPU support for the Bin-Hash: Caching Candidate Checks: The CPU serves only to supply the GPU with spatial hash and offset tables, it performs no processing. As discussed in Section II, QDV applications benefit from caching full-resolution bin data. We optimize our Bin-Hash implementation for QDV applications by supporting our candidate checks with a simple two-level cache strategy, one for the CPU and one for the GPU, with each cache level operating under a separate LRU replacement policy. The GPU and CPU cache hold the hash and offset tables of more frequently queried boundary bins. The results of our new Bin-Hash strategy (Section V) are presented to reflect the three possible cache-state conditions that might be encountered when answering a candidate check:

1) System-cache: GPU and CPU caches fail on *every* query; the OS file-cache, however, has the necessary files. The cost reflected here is the cost to transfer the hash and offset tables from system space to user space, and then from user space to the GPU.
2) CPU cache: The GPU's cache fails, but the CPU's cache hits on *every* query; the CPU supplies the necessary hash and offset tables to the GPU from its main memory.
3) Full cache: The GPU's cache hits with *every* query; all necessary hash and offset tables are found on the GPU.

V. Performance and Analysis

A. Test Setup and Performance Metrics

We have implemented and tested our Bin-Hash index on the machine configuration described below (Section V-A.2). It is important to state at the outset of this section, that our work concentrates on assessing the performance of block-level operations—the internal structure used by such bitmap indexing software as ORACLE and FastBit.

We select this performance metric based on the performance needs of our target application, Query-Driven Visualization (QDV). In QDV, the common case query is one that evaluates memory-resident or cached data; the infrequent query is a query that must access data from disk. Specifically, in QDV, users spend the majority of their time querying a set group of variables (i.e. columns) in order to establish relationships and trends between these variables (see Section II). After the first query, the data for these variables will be loaded into memory and cached in the OS—the performance of all subsequent queries over these columns will benefit from the cached state of the data.

Favoring the frequent query over the infrequent query, we determine the best indexing method for QDV by selecting the indexing method that provides the best query performance over data that has been loaded into memory (GPU or CPU depending on the indexing method) and cached by the OS. Though this performance metric is somewhat unusual, it is consistent with the needs of the motivational application. Additionally, it is consistent with previous literature that has reported on the direct comparative performance of GPU and CPU based indexing methods for answering selection queries [11].

As part of assessing this block-level performance, our performance results do *not* reflect disk access times or the time to preload data. In our test suites, we begin our timings for the Bin-Hash queries *after* the initial loading of low-resolution column data into GPU memory (see Section IV). The Bin-Hash timings *do* however reflect the cost of candidate checks (see Section IV-C.3). Additionally, FastBit and CPU-based projection scan results reflect OS-File cache performance, *not* disk access time. Finally, GPU-based projection scan timings are taken *after* the preloading of required data into GPU memory.

1) Indexing Schemes: For our testing, we chose the following indexing methods as they are all established indexing strategies that are efficient for QDV.

- *CPU-based projection scan:* Each full-resolution column is read into CPU memory space. We evaluate the query by simply performing comparisons on the array without any additional data structure. For many visualization

applications, this approach is the basic strategy of query processing. In our tests, this indexing method provides a baseline for performance.

- *FastBit:* A high-performing, CPU-based approach for query processing that utilizes compressed bitmap indices. FastBit is currently the state-of-the-art indexing strategy utilized for QDV purposes [20], [37].
- *Bin-Hash Index:* The indexing method described in this paper.
- *GPU-based projection scan:* Equivalent to the CPU-based projection scan, with the exception that the full-resolution column is read into *GPU* memory space (thus the problem size is directly limited by GPU memory size). Additionally, all indexed values in the column are simultaneously evaluated in parallel by the query. This approach is identical (in logic) to the work presented by Govindaraju et al. [11]. Where their implementation utilized native GPU hardware buffers (depth, stencil buffers etc.), we utilized CUDA kernels on the GPU to perform the same query processing operations.

Tree-based indexing strategies exhibit exponential growth in storage requirements with increasing dimensionality [38], [39]. Given that QDV applications typically analyze high-dimensional scientific data, tree-based indexing strategies are not typically used in QDV applications, and are thus not represented in our test suites.

2) Machine Configuration: All tests were performed 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.

- *Motherboard*: EVGA 680i - 1066MHz FSB; 16X PCI-Express
- *Processor*: Intel QX6700 - 2.66GHz; 2 x 128KB L1; 2 x 4MB L2
- *Memory*: Corsair - 4GB 1066 DDR2
- *Co-processor*: NVIDIA 8800GTX - 768MB GDDR3
- *HardDisk*: WD CAVRE2 - SATA 3Gb/sec; 500GB; 7200 RPM; 16MB cache; 8.7 ms seek

B. Test 1: Query Selectivity vs. Row Count

Many indexing methods utilized for query processing (e.g. FastBit) exhibit performance that is influenced by query selectivity; in these methods queries that select fewer records are answered faster than queries that select a comparatively larger number of records. QDV indexing strategies must efficiently support both highly *and* broadly selective queries. Specifically, in the exploratory approach taken by QDV, user understanding and insight typically begins at a coarse level where query selectivities will be broad; only through iterative refinement will the selectivity of the records increase.

This suite of tests explores the selectivity-performance relationship on a series of single column tables, each of which models a stage of hydrogen combustion. Each sequential table in these tests, while having only one column, has an increasing number of rows: 33, 67, 100, 134, 167 and 201 million rows. In this suite of tests, we examine the effect that increasing query selectivity has on performance. Specifically, we evaluate the performance of queries that select 40%, 20%, 10%, 5%, and 1% of the records from each respective table.

In general, we expect for FastBit to display excellent performance for queries that possess high selectivity (FastBit accesses fewer bitmaps with highly selective queries [18]). The rest of the implemented methods, all utilizing $O(n)$ strategies to evaluate a query (i.e. all values must be accessed during query evaluation), should display constant performance regardless of query selectivity.

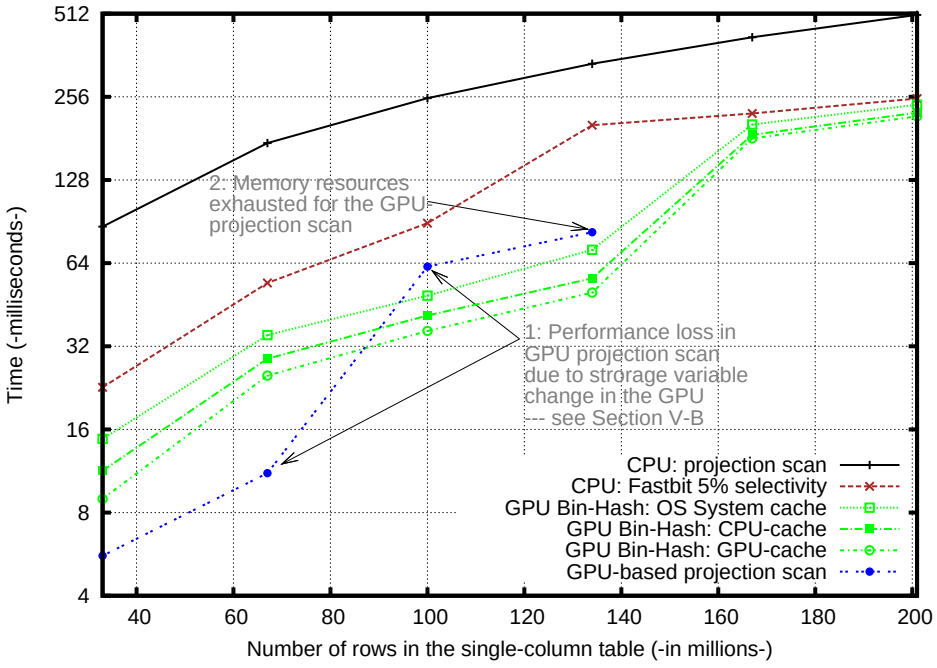
There are two principal questions we seek to answer in this suite of tests. First, how does the performance of our new Bin-Hash indexing method compare to the state-of-the-art bitmap indexing performance of FastBit? Second, compared to the GPU-based projection scan, our new Bin-Hash index strategy better utilizes GPU-memory resources and bandwidth—thus enabling query evaluation over larger data—at the expense of additional GPU computation during candidate checks. Given this computation for memory trade-off, we expect for the GPU-based projection scan to outperform our new Bin-Hash indexing method. The question is, exactly how much performance does our new Bin-Hash method sacrifice in this trade-off, and how much better memory utilization are we gaining in return?

Analysis: The performance results of our tests are shown in Figure 5(a) and Figure 5(b). For presentational clarity, we group the performance results based upon those indexing methods that showed no performance change with respect to query selectivity (Figure 5(a), which shows the performance of both CPU and GPU projection scans, and the Bin-Hash index), and those indexing methods that showed performance improvements with increasing query selectivity (Figure 5(b), which shows FastBit's performance results). Additionally, notable performance trends are labeled in Figure 5(a) and Figure 5(b), and are discussed below.

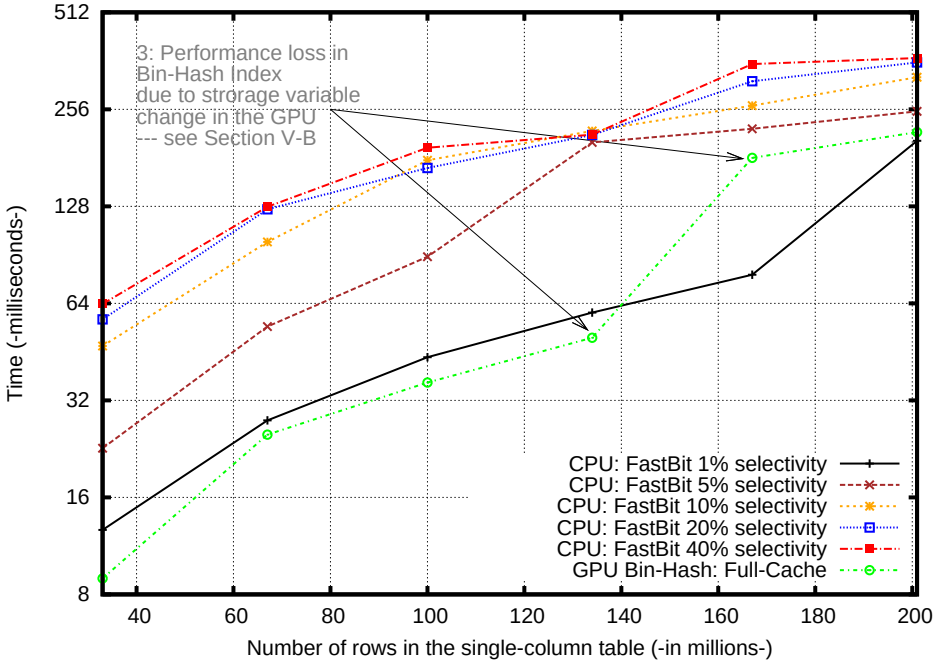
The expected performance trends discussed in the beginning of this section are confirmed: FastBit displays excellent performance for queries that possess high selectivity, and the performance of the other implemented index methods display no dependency on query selectivity (the results shown in Figure 5(a) represent the timings observed for all query selectivities: 1% - 40%).

Throughout Figure 5(b) the Bin-Hash index performance out-performs FastBit—even when FastBit evaluates highly selective queries (with the exception being the range highlighted in label 3). The key observation to make is that our new Bin-Hash index provides this level of high performance for *all* queries, regardless of selectivity. Thus from these performance results, the Bin-Hash index appears well-suited for meeting the needs of QDV applications.

Labels 1 (Figure 5(a)) and 3 (Figure 5(b)) highlight a sharp loss in performance for the Bin-Hash indexing approach and the GPU-based projection scan. This performance loss is due to a GPU-based implementation detail associated with how the



(a)



(b)

Fig. 5: These charts depict the complete results for the tests performed in Section V-B. For clarity the FastBit performance results are depicted in (b), while all other results are shown in (a)

(a): This figure shows the timing results of 3 different querying strategies. The x-axis depicts the number of rows contained in each single-column table queried. Each table is queried with 5 separate queries selecting 1%, 5%, 10%, 20%, and 40% of the database records as hits. The projection scan and the Bin-Hash methods displayed no difference in timings with respect to increasing query selectivity. This is due to the fact that all of these strategies have a working complexity of $O(n)$. For comparison, the FastBit results for queries returning 5% of the records as hits are shown in both this table, and in Figure 5(b). For clarity, the performance lines for the three cache levels of the Bin-Hash index are intentionally colored the same to reflect the *range* of performance of the Bin-Hash index.

(b): This figure shows the timing results of FastBit, shown here when utilized by 5 separate queries, respectively selecting 1%, 5%, 10%, 20%, and 40% of the records as hits. We observe that FastBit shows significant timing differences with respect to the selectivity of the query. For comparative purposes with Figure 5(a), the Full-Cache results for the Bin-Hash method are shown. It can be seen that with queries selecting 1% of the records as hits, FastBit performs remarkably well to the parallel approach—equaling or even besting the best case of the Bin Hash strategy when querying some of the largest tables examined (this loss of performance for the Bin-Hash index is due to a specific GPU implementation issue, which is discussed in Section V-B).

bit-vector solution is written for tables containing in-excess of 66 (for the GPU-based projection scan) or 133 (for the Bin-Hash index) million rows . Specifically, for tables whose row numbers exceed these values, the projection scan (which uses 32-bit raw data) and the Bin-Hash index (which uses 8-bit data corresponding to bin numbers) can no longer store the bit-vector solution with a 32-bit variable type (the GPU memory resources are exhausted); instead an 8-bit variable type is utilized to conserve space. Writing data as an 8-bit "ubyte" to the GPU's global memory incurs significant performance penalties as the large timing increase highlighted by labels 1 and 3 show.

The high performance of GPU-based query strategies is inevitably limited by the constraints of GPU-memory (highlighted performance of labels 1 and 3). Our new Bin-Hash method is designed to trade a portion of the GPU's computational performance (due to the computational cost associated with the Bin-Hash's candidate checks) for a better utilization of GPU memory and bandwidth. As a result of this strategic "computation for memory and bandwidth" trade-off, the advantages provided by our new Bin-Hash method (compared to the GPU-based projection scan) are that:

- significantly larger tables are able to be queried, and
- queries processed over these larger tables are answered with consistently higher levels of performance (for tables in excess of 85 million rows).

Specifically, label 2 in Figure 5(a) shows the point (tables containing in-excess of 133 million rows) where the GPU-based projection scan has exhausted GPU-memory resources; the equivalent point for our new Bin-Hash indexing strategy is approximately 360 million rows. Thus, in comparison to the GPU-based projection scan, our new Bin-Hash method is able to evaluate queries over tables containing 2.7 times as many rows.

C. Test 2: Multi-Dimensional Queries

It is critical in QDV to efficiently evaluate multidimensional queries over high-dimensional data (see Sections II and V-A.1). In this suite of tests we aim to determine what impact, if any, a query's dimensionality has on performance. The dataset used—a subset of the Hurricane Isabel Model—consists of a single table containing 48 columns and 25 million rows. We begin with a single-dimensional predicate querying just one column from this table. In each subsequent test, an additional column is added to the query (through a logical OR operation) until a total of 8 columns are queried. The selectivity of these queries is increased linearly throughout these tests such that each new column added to the query selects—disjointly—an additional 12% of the rows from the table.

This test suite's motivations are twofold. First, we seek to demonstrate the implementation of the multi-dimensional kernel approach from Section IV-C.2 and assess its performance. Second, we seek to complete the characterization of the Bin-Hash index's "computation for memory and bandwidth" trade-off (see Section V-B). Specifically, Section V-B highlighted performance limitations for both GPU-based indexing strategies (Bin-Hash and projection scan) when they processed queries over tables containing a certain row count. We now seek to characterize the performance of these two strategies by querying tables of a smaller, fixed row count. This smaller number of rows results in a smaller bit-vector solution; thus this Section's test suite consistently uses a 32-bit variable type for the bit-vector solution and *not* the 8-bit variable type which resulted in performance loss for both methods in Section V-B.

Records Indexed *-in millions-*	*Bin-Hash -in milliseconds-*	*Projection Scan -in milliseconds-*	*Ratio*
25	6.0	5.0	1.19
50	10.76	8.18	1.31
75	15.8	11.45	1.38
100	20.9	14.67	1.42
125	26.21	17.93	1.46
150	30.97	21.41	1.44

TABLE I: Raw performance values (in milliseconds) for the Bin-Hash GPU-cache, and GPU-based projection scan indexing methods (values taken from Figure 6). Additionally, this table shows the calculated performance ratio between these two indexing strategies. The average value for the performance ratios is 1.37, indicating that the Bin-Hash method spends, on average, an additional 37% more time in computation (than the projection scan) in exchange for being able to store and query four times as many columns from this test suite (see "Analysis" in Section V-C).

Analysis: The performance results of our tests are shown in Figure 6. With moderate query selectivity (cumulatively 12% per column queried), FastBit is outperformed by the GPU-based indexing strategies. Label 2 in Figure 6 highlights the region where the total number of indexed values processed by the query span from 125 to 175 million—this range is approximately identical to the span of indexed values highlighted by label 3 in Figure 5(b) (which is 135 to 167 million). Though the number of values processed for querying in these ranges is approximately the same, GPU-based performance through these ranges is radically different. The accelerated performance depicted in Figure 6 is a result of the constant and smaller table size which allows the bit-vector solution in this test suite to use a higher performing 32-bit variable type—the bit-vector solution in Section V-B uses an 8-bit variable type to conserve memory resources.

The GPU-based projection scan still exhausts (label 1 in Figure 6) GPU memory resources, despite a smaller, constant number of rows for this test suite's table. The advantage provided by our new Bin-Hash method is that four times as many columns (Figure 6 only shows data for 8 columns but 24 are able to be loaded and utilized) are able to be stored and queried on the GPU before exhausting GPU memory resources.

Figure 6 and Table I show that our new Bin-Hash method scales linearly with respect to increasing query workloads. This demonstrates the efficiency of the multivariate kernel strategy (Section IV-C.2). Additionally, from the data presented in Table I, observe that the span of the performance ratio (constructed from the Bin-Hash GPU-cache and GPU-based projection scan) is approximately 1.3 - 1.45. In comparison to the projection scan, this indicates that our new Bin-Hash method spends an additional 30 to 45 percent more

Dataset	*Index Size (GB)*	*Raw Data Ratio*
Hurricane: Projection	0.8	1.0
Hurricane: FastBit	1.37	1.71
Hurricane: Bin-Hash	1.54	1.92
Hydrogen: Projection	0.805	1.0
Hydrogen: FastBit	0.49	0.62
Hydrogen: Bin-Hash	1.16	1.44

TABLE II: This table depicts the size of the index generated for the various query processing strategies utilized in Section V-C and Section V-B.

time in computation in exchange for being able to utilize GPU memory with significantly more efficiency (i.e. four times as many columns are able to be loaded and queried).

D. Index Size

Query response time (analyzed in Sections V-B and V-C) is an important factor in analyzing these various strategies; also important in assessing the performance of an indexing method is the index size. In the previous two test suites, the distribution of data in the respective datasets varies significantly. The dataset used in Section V-C, the Hurricane dataset, contains data that is notably skewed. Such behavior in data is difficult to compress. Comparatively, the Hydrogen dataset used in Section V-B contains data distributions that are considerably less skewed, i.e. more smooth or uniform. Such data is often much easier to compress.

Table II shows the results of the total overhead required for each dataset given the three different storage approaches used in Sections V-C and V-B: raw data (for projection scan), compressed data (for FastBit), and encoded data (for the Bin-Hash method). Here we observe, in the three top rows in Table II, the penalty incurred by compressing and encoding near-random data. Both the compression utilized by FastBit, and the encoding performed by the Bin-Hash method, approach double the size of the raw data. In comparison, the Hydrogen dataset, displayed in the three bottom rows, show the more "typical", or expected storage gains achieved from FastBit's compression strategy.

FastBit uses a binning strategy that will answer any query conditions involving query boundaries of five significant digits or less. Due to the need to find precise query boundaries to ensure that the specified fraction of records are retrieved, the number of significant digits in the query boundaries are relatively high in this case. This increases the number of bins and therefore the overall index size (as reflected in Table II).

VI. Conclusions

Our work in this paper provides the ability to take advantage of platforms that support extreme multithreading in order to accelerate index/query operations. This type of capability is a crucial underpinning of interactive visual data analysis. In this paper, we have presented the Bin-Hash indexing strategy for the answering of selection queries. The Bin-Hash indexing strategy offers the same high levels of concurrency possessed by other parallel methods (e.g. the projection scan), but also affords the utilization of significantly less memory resources. This smaller memory footprint is critical for certain parallel processors, like the GPU, where memory resources are limited.

The underlying strategy of the Bin-Hash index is to reduce memory and bandwidth requirements, at the expense of additional GPU computation. Through this strategic "computation for memory and bandwidth" trade-off, we have effectively utilized the GPU for query processing: our performance results show that our new Bin-Hash index is able to outperform FastBit by up to an order of magnitude, and (for large databases) outperform the more computationally favored GPU-based projection scan.

We have also demonstrated the utility of perfect spatial hashing as a parallel data structure. The critical functionality provided by this data structure allows the Bin-Hash index to utilize a binning strategy to evaluate the vast majority of records, and efficiently access the base data of records for a given boundary bin when performing a candidate check.

We are currently developing a *nested* binning strategy (i.e., binning the records contained in bins) that will enable the Bin-Hash strategy to provide even further performance benefits. We anticipate that this approach will make an out-of-core strategy efficient for use on the GPU.

Acknowledgments

The authors thank John Bell and Marc Day of the Center for Computational Science and Engineering, Lawrence Berkeley National Laboratory for providing the hydrogen flame dataset. Additionally, the authors thank Bill Kuo, Wei Wang, Cindy Bruyere, Tim Scheitlin, and Don Middleton of the U.S. National Center for Atmospheric Research (NCAR), and the U.S. National Science Foundation (NSF) for providing the Weather Research and Forecasting (WRF) Model simulation data of Hurricane Isabel. This work was supported by Lawrence Berkeley National Laboratories, and by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET) and Scientific Data Management Center. We would like to thank colleagues in the Institute for Data Analysis and Visualization (IDAV) at UC Davis for their support during the course of this work. Last, a special thanks goes to Jim Gray whose insight helped to inspire this work in its initial stages.

References

[1] J. Becla and K.-T. Lim, "Report from the workshop on extremely large databases," 2007. [Online]. Available: http://www-conf.slac.stanford.edu/xldb07/xldb07_report.pdf

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber, "Scientific data management in the coming decade," CTWatch Quarterly, 2005. [Online]. Available: http://www.ctwatch.org/quarterly/articles/2005/02/scientific-data-management

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[5] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[6] R. Raman and U. Vishkin, "Parallel algorithms for database operations and a database operation for parallel algorithms," in *Proc. International Parallel Processing Symposium (IPPS)*, 1995.

[7] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH*—a scalable, distributed data structure," *ACM Trans. Database Syst.*, vol. 21, no. 4, pp. 480–525, 1996.

[8] M. G. Norman, T. Zurek, and P. Thanisch, "Much ado about shared-nothing," *SIGMOD Rec.*, vol. 25, no. 3, pp. 16–21, 1996.

[9] M. Bamha and G. Hains, *Frequency-adaptive join for shared nothing machines*. Nova Science Publishers, Inc., 2001, pp. 227–241.

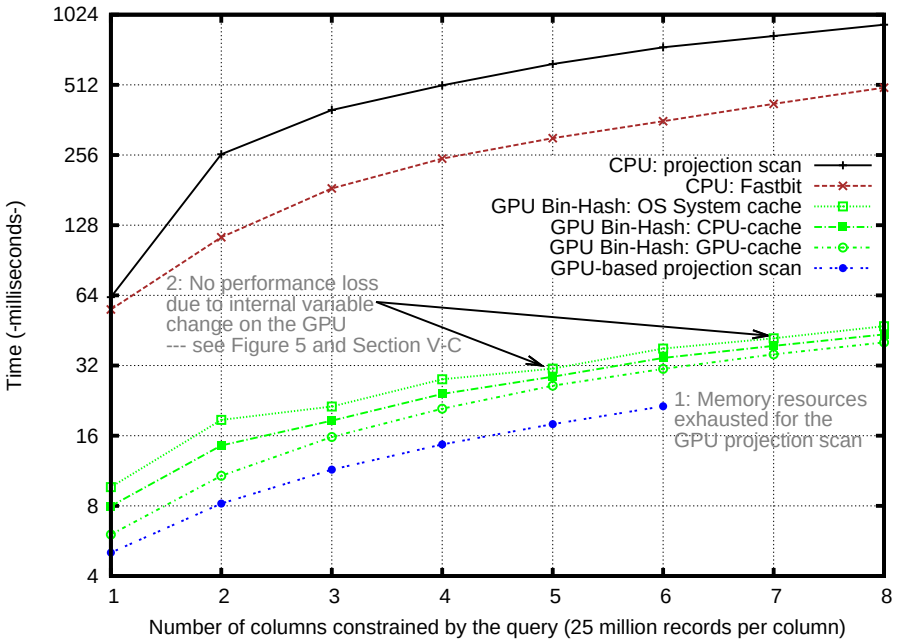


Fig. 6: This figure shows the results of 3 different querying strategies over a table containing 48 columns and 25 million rows. The X-axis of this table (1, 2 ...8) lists the total number of columns constrained in the multidimensional query. The selectivity of these queries grows at constant rate: each column added to the query selects an additional 12% of the rows from the table being queried. Notable trends are discussed in Section V-C.

[10] J. W. Rahayu and D. Taniar, "Parallel selection query processing involving index in parallel database systems," in *ISPAN'02*, 2002, p. 0309.

[11] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proc. of SIGMOD*, June 2004, pp. 215–226.

[12] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proc. of SIGMOD*, June 2006, pp. 325–336.

[13] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "GPUQP: query co-processing using graphics processors," in *Proc. of SIGMOD*, 2007, pp. 1061–1063.

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

[15] P. E. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proc. of SIGMOD*, May 1997, pp. 38–49.

[16] D. Comer, "The ubiquitous B-tree," *Computing Surveys*, vol. 11, no. 2, pp. 121–137, 1979.

[17] V. Gaede and O. Günther, "Multidimension access methods," *ACM Computing Surveys*, vol. 30, no. 2, pp. 170–231, 1998.

[18] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. on Database Systems*, vol. 31, no. 1, pp. 1–38, Mar. 2006.

[19] K. Stockinger, K. Wu, and A. Shoshani, "Evaluation strategies for bitmap indices with binning," in *DEXA 2004, Zaragoza, Spain*, Sept. 2004.

[20] K. Stockinger, J. Shalf, K. Wu, and E. W. Bethel, "Query-driven visualization of large data sets," in *Proc. of IEEE Visualization*, Oct. 2005, pp. 167–174.

[21] D. Keim and H.-P. Kriegel, "VisDB: Database exploration using multidimensional visualization," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 40–49, 1994.

[22] P. E. O'Neil, "Model 204 architecture and performance," in *Second International Workshop in High Performance Transaction Systems*, ser. Lecture Notes in Computer Science, vol. 359, 1987, pp. 40–59.

[23] A. Shoshani, L. Bernardo, H. Nordberg, D. Rotem, and A. Sim, "Multidimensional indexing and query coordination for tertiary storage management," in *Proc. of SSDBM*, July 1999, pp. 214–225.

[24] N. Koudas, "Space efficient bitmap indexing," in *Proc. of Conference on Information and Knowledge Management*, 2000, pp. 194–201.

[25] S. Amer-Yahia and T. Johnson, "Optimizing queries on compressed bitmaps," in *Proc. of the Conference on Very Large Data Bases*, 2000, pp. 329–338.

[26] G. Antoshenkov, "Byte-aligned bitmap compression," in *Proc. of the Conference on Data Compression*, 1995, p. 476.

[27] G. Antoshenkov and M. Ziauddin, "Query processing and optimization in ORACLE RDB," in *Proc. of the Conference on Very Large Data Bases*, 1996, pp. 229–237.

[28] K. Wu, E. Otoo, and A. Shoshani, "On the performance of bitmap indices for high cardinality attributes," in *Proc. of VLDB*, Aug. 2004, pp. 24–35.

[29] M. Glatter, J. Huang, J. Gao, and C. Mollenhour, "Scalable data servers for large multivariate volume visualization," *Trans. on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1291–1298, 2006.

[30] P. McCormick, J. Inman, J. Ahrens, C. Hansen, and G. Roth, "Scout: A hardware-accelerated system for quantitatively driven visualization and analysis," in *Proc. of IEEE Visualization*, Oct. 2004, pp. 171–178.

[31] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Super Computing*, 2007.

[32] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[33] C. Sun, D. Agrawal, and A. E. Abbadi, "Hardware acceleration for spatial selections and joins," in *Proc. of SIGMOD*, June 2003, pp. 455–466.

[34] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Trans. on Graphics*, vol. 25, no. 3, pp. 579–588, 2006.

[35] T. J. Sager, "A polynomial time generator for minimal perfect hash functions," *Communications of the ACM*, vol. 28, no. 5, pp. 523–532, 1985.

[36] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, "Practical minimal perfect hash functions for large databases," *Communications of the ACM*, vol. 35, no. 1, pp. 105–121, 1992.

[37] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu, "Accelerating network traffic analysis using query-driven visualization," in *Proc. of the Symposium on Visual Analytics Science and Technology*, Oct. 2006, pp. 115–122.

[38] J. Bentley, "Multidimensional binary search trees used for associative search," *Communications of the ACM*, vol. 18, no. 9, pp. 509–516, 1975.

[39] R. Bellman, *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.