SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00086

# MoHA: A Composable System for Efficient In-Situ Analytics on Heterogeneous HPC Systems

Haoyuan Xing
*The Ohio State University*
xing.136@osu.edu

Gagan Agrawal
*Augusta University*
gagrawal@augusta.edu

Rajiv Ramnath
*The Ohio State University*
ramnath.6@osu.edu

***Abstract*—Heterogeneous, dense computing architectures consisting of several accelerators, such as GPUs, attached to general-purpose CPUs are now integral High-Performance Computing (HPC) systems. However, these architectures pose severe memory and I/O constraints to computations involving in-situ analytics. This paper introduces MoHA, a framework for in-situ analytics that is designed to efficiently use the limited resources available on heterogeneous platforms. MoHA achieves this efficiency through the extensive use of bitmaps as a compressed or summary representation of simulation outputs. Our specific contributions in this paper include the design of bitmap generation and storage methods suitable for GPUs, the design and efficient implementation of a set of key operators for MoHA, and demonstrations of how several real queries on real datasets can be implemented using these operators. We demonstrate that MoHA reduces I/O transfer as well as overall processing time when compared to a baseline that does not use compressed representations.**

***Index Terms*—Indexes, Data compression, High performance computing, Scientific computing, Query processing, Accelerator architecture**

## I. Introduction

Analytics has become a significant part of HPC, as findings from simulation outputs can be a prominent source of data-driven breakthrough. Recently, analytics on scientific simulations output underwent a major and *disruptive* change because of the shift in high-performance computing landscape. Increasingly, such analytics have become *in-situ*, *i.e.*, before writing the data to the disk, either entirely or partially replacing *post hoc* analysis. Though in-situ analytics does have a long history, driven by needs like *computational steering* or *human-in-the-loop* simulations [1], architectural trends lately have made it a common approach as compared to an exception [2], [3], [4], [5], [6]. The work done in recent years focus on in-situ algorithms and applications [7], [8], as well as infrastructure and frameworks [9], [10], [11], [12], [13], [14], [15].

However, despite a substantial volume of work in this area in recent years, in-situ analytics workload today is mostly simplistic and does not meet the needs associated with up-and-coming architectural and application trends. Recently commissioned machines that can offer $10^{17}$ floating-point operations per second involve very *dense* nodes, packing more computational capacities in the from of multiple GPUs on each node. Specifically, each node of DOE Summit machine has a peak performance of 42 Teraflops/second, and delivers an order of magnitude higher performance over the predecessor machines with fewer nodes, and by extension lower I/O capacity relative to its computational power [16]. It is anticipated that Exascale machines will only have $10^{12}$ bytes/second I/O capability, $10^6$ times lower than their floating-point computation capability [17] – a ratio that has gotten worse by a factor of 200 since the first Petaflop machine.

The second important architectural trend is that accelerators are becoming the norm rather than the exception in the extreme-scale systems. For example, Summit/Sierra machines strongly feature accelerators, with 97% of the total computing power of the machine being on GPUs. Meanwhile, there is limited amount of work on in-situ analytics when accelerators are involved [18], [19], [20]. With much of the computing occurring on accelerators that have a limited memory, in-situ analytics needs to operate with stringent memory limitations. [21].

Similarly, there are changing application needs regarding in-situ analytics. Broadly speaking, the focus is shifting from visualization to detailed analytics. Consider the following requirement typically associated with Smart Simulations [22] – *"as a simulation is running, compare output from each time-step against stored data (either observed data or output from another model), and summarize the spatial areas and/or variables where the differences are significant"*. Other examples of applications that require advancements in in-situ analytics include *ensemble simulations*, where all of a set of concurrently running simulations can be terminated if there is one successful event detection; *Urgent HPC*, where HPC application need to be executed in a set amount of time (and the output needs to be summarized and communicated); and *Computing-In-the-Loop* – requiring close interaction between an experiment or observation modality and the HPC simulations. In all of these cases, in-situ analysis needs to be quick, resource-limited, and effective.

This paper introduces an in-situ heterogeneous scientific query system, MoHA (Modern Heterogeneous Analytics). The system allows complex in-situ queries on data generated by accelerators. The main idea is to construct a summary or compressed representation of data and then perform data transfer and query processing on this representation instead of the original data. In that sense, the representation can be viewed as a kind of *homomorphic compression* [23].

The specific representation MoHA uses is bitmap index, an indexing structure initially developed in the context of data warehouses [24], [25], [26] and subsequently widely used for scientific data [27], [28], [29], [30], [31]. In our case, the bitmap is not used as an index for looking up data, but a concise summary of the array-based accelerator-generated data, which preserves both spatial and value distributions.

Such use of bitmaps instead of the original data enables lowered data movement costs, less I/O volume, and better memory and time efficiency in the analysis. In addition, MoHA provides several other important features. MoHA supports a number of composable basic operators on data, all implemented using bitmaps to facilitate the development of complex query processing solutions; and supports parallel query processing both within and across nodes.

We evaluate MoHA using two real-world simulations. Our results indicate that, compared with a baseline that processes queries using original data, MoHA significantly saves transferring cost between hosts and accelerators even after considering the bitmap generation cost, provides speedups for both I/O and computation intensive queries, and has good parallel scalability. Compared with the baseline, MoHA provides an average speedup of $\sim 1.5\times$ when the query does not involve I/O, and $\sim 8\times$ for I/O-intensive queries.

In summary, this paper makes the following contributions:

- It introduces an improved bitmap format suitable for GPU generation and processing.
- It demonstrates how such a bitmap formats allows fast and resource-limited processing of complex in-situ queries such as contrast set mining and overlapped interest region.
- It introduces MoHA, a parallel and composable in-situ query system for heterogeneous platforms.
- Our extensive experimental evaluation shows MoHA can process in-situ queries on heterogeneous platforms with significant less I/O as well as computation cost.

The remaining parts of the paper are organized as follows: Section II lays out necessary background concepts; Section III starts by introducing the motivating queries we use as examples through the paper. Section IV discusses the overall architecture of MoHA; Section V introduces the GPU-friendly representation MoHA uses and how it can be generated; Section VI follows by discuss query processing on such representation. We experimentally evaluate the Section VII, discusses related work in Section VIII, and conclude the paper in Section IX.

## II. Background

### A. Heterogeneous (GPU) Platforms

GPUs rely on large-scale data-parallelism to improve computation and energy efficiency. A GPU has many streaming processors (SMs), each of which has its own L1 cache. SMs shares the L2 cache and have access to a high-bandwidth *global* device memory. A part of the L1 cache can be explicitly controlled by a programmer, and is referred to as *shared memory*. Although other experimental designs exist, currently, a GPU usually cannot perform direct access to the host main memory effectively, and exchanges data with the host system using the PCI-E or NVMe bus, which has limited and shared bandwidth. Hiding the latency of data transfers is usually the key to improving the performance of a GPU program.

A user programs the GPU using an abstraction called *lightweight threads*. Each GPU thread represents one piece of execution logic that can be executed in a massively parallel fashion. Each parallel function executed on GPU, also called a *kernel*, consists of thousands of GPU threads with the same execution logic. These threads are divided into groups called *blocks*, each of which is always executed together on the same streaming processor. The threads in the same block can communicate through shared memory and barriers. A streaming processor executes the threads in a block in the granularity of *warps*, each of which usually comprises 32 threads. The execution follows Single Instruction Multiple Threads (SIMT) model: at any given point, the same instructions are executed for all the threads in a warp. [1]

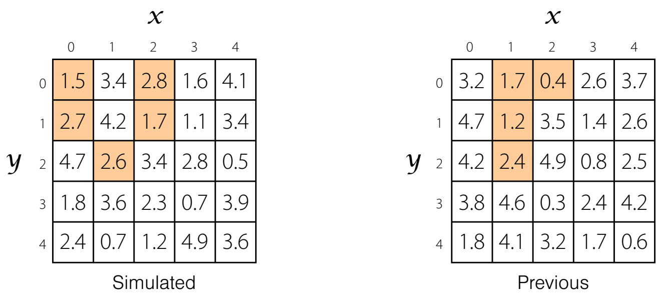
### B. Scientific Datasets and Bitmap Representation

Most data generated by scientific simulations can be represented as *multi-dimensional arrays*. Overall, an array associates the data to a multi-dimensional Euclidean space. In such a model, the data is represented as a one-to-one mapping between a numeric vector (*dimensions* or *coordinates*) to a nullable tuple (*attributes*). Each column of the attributes is often referred to as a *dataset*. For better I/O and subsetting performance, arrays are stored and processed in the granularity of *chunks*. Although the strategy of dividing array varies, most array models use *regular tiling*, with each chunk representing a hyperrectangular subset of the array [32].

While indexing can be a powerful tool for accelerating high-selectivity analytics queries, indexing scientific data can be challenging, because of large cardinality and the nature of floating-point data. Besides, large I/O block sizes make looking up the values expensive. *Bitmaps indices with binning* have been proven to be a convenient representation for such a task [33], [34], [35], [36]. In this method, each value domain is divided into a number of *buckets* or *bins*, and the values in each value are stored as a bitmap, with a `1` bit represents that the value is inside the bucket, and a `0` bit representing otherwise. In order to reduce the size of the indices, each of the bitmaps is usually stored in one of the compressed formats; the widely used approaches include run-length encoding [33], [37], small-integer compression [38] as well as tree-based encoding [39].

Bitmap indices have the advantage of reduced size. A key observation in this work (as well as some of the prior researches [35], [34]) is that a bitmap can act as a summary

[1]The terms used here follow the CUDA programming model from NVIDIA. Other programming models such as OpenCL might use different terms.



| y \ x | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1.5 | 3.4 | 2.8 | 1.6 | 4.1 |
| 1 | 2.7 | 4.2 | 1.7 | 1.1 | 3.4 |
| 2 | 4.7 | 2.6 | 3.4 | 2.8 | 0.5 |
| 3 | 1.8 | 3.6 | 2.3 | 0.7 | 3.9 |
| 4 | 2.4 | 0.7 | 1.2 | 4.9 | 3.6 |

| y \ x | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 3.2 | 1.7 | 0.4 | 2.6 | 3.7 |
| 1 | 4.7 | 1.2 | 3.5 | 1.4 | 2.6 |
| 2 | 4.2 | 2.4 | 4.9 | 0.8 | 2.5 |
| 3 | 3.8 | 4.6 | 0.3 | 2.4 | 4.2 |
| 4 | 1.8 | 4.1 | 3.2 | 1.7 | 0.6 |

Fig. 1: An example of two contrast sets with filtering predicates $0 \leqslant x < 3$, $0 \leqslant y < 3$ and $0 \leqslant v < 3$. Highlighted cells are the elements in the contrast sets. Target attributes are omitted.

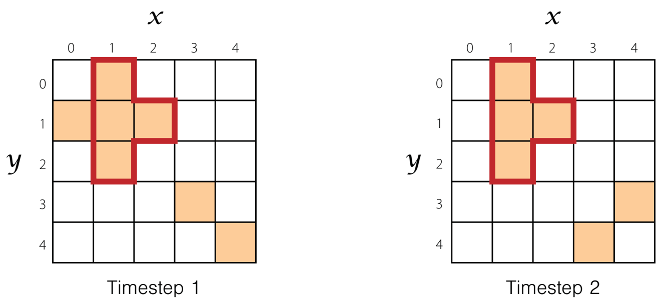


Fig. 2: An illustration of the overlapped interested region query. The filled regions indicates the interesting points in each timestep. The query looks for the continuous overlapped region across the timesteps, marked using red frames here.

representation of the original data without losing much precision. While there exist other data summarization techniques such as sampling [40], [41], [42], [43], histograms [44], [45], [46], wavelets [47], [48], [49], [50], and data sketches [51], [52], [53], [54], extending these representations to skewed high-dimensional data can be hard [46], [47], [55]. In contrast, bitmap indices have the advantages of retaining the spatial information of the multi-dimensional array, as well as allowing faster intersection and union based on bitwise operations in the hardware.

## III. Motivating Queries

As in-situ analytics attract more and more attention, the need for performing more complex queries in in-situ query processing increases. Fulfilling such requirements is a major consideration during the design of MoHA. In this section, two example queries are discussed here as motivating examples.

*a) Contrast Set Mining:* Contrast Set Mining [56], [57] aims at exploring the differences between a running simulation and a previous baseline. A pair of contrast sets are two subsets of the simulation and the baseline selected by the same filtering predicates. Figure 1 gives an example of a pair of such contrast sets, with the same predicates applied across two datasets. The filter predicate(s) that produces the most significant difference in the value of the target attributes (omitted in the Figure) might shed light on the difference between the two experimental results being considered. Hence, this query aims at finding the predicates that produce the most different contrast sets, usually defined using a *quality function* – the quality function, in turn, uses quantities such as the size (support) and mean of the two subsets.

*b) Overlapped interest region:* The next query looks for stable features across timesteps of the simulation results. Continuous regions in the simulation results that satisfy a certain threshold are often searched for further examination [35]. Given a running simulation, and a filter condition defining which cells are "interesting", the overlapped interest region query looks for adjacent time steps with large continuous overlapped interest regions, which suggests that these time steps have stable features that worth examination. Figure 2 gives an illustration of such a query.

These queries shows that in-situ analytic engines need a flexible design that can accommodate queries that involves disk-based data, as well as queries that focus on individual simulation results. The next section discusses how MoHA addresses such issues.

## IV. System Overview

Our in-situ heterogeneous analytics system MoHA is designed with multiple goals in mind. From the performance angle, our goals are to reduce the amount of time spent on data transfers and I/O, reduce the memory requirements for doing in-situ analytics, and reduce the time spent on analytics. From the programmability side, we want support for managing the simulation output data, a simplified logical view of data on which analytics are specified, and the ability to compose queries using a series of simple operators.

### A. Modeling and Representing Simulation Data

A scientific simulation usually generates many *timesteps*, each contains several multi-dimensional *datasets*. Hence, MoHA views all the data as a (virtual) single multi-dimensional array, with the identifier of a timestep being one of its dimensions, and each datasets being one of the *attributes* of the array. When the simulation is executing, each accelerator usually generates a portion of the timestep at a time, corresponding to a hyper-rectangular *chunk* of the array. MoHA accepts a stream of such chunks, and performs analytics on it.

Under the hood, each chunk is represented by a *bitmap index*, dividing the value domain into a number of disjoint *buckets* and using a bitmap to represent the values that fall in each bucket. As the next two sections will show, such a representation can be efficiently generated on GPU and can be used to execute complex queries efficiently. This representation serves multiple goals: its reduced size alleviates the memory pressure and the bottleneck of transferring data to CPU, and it is more efficient for many queries, primarily because it accelerates value-based filtering operations.

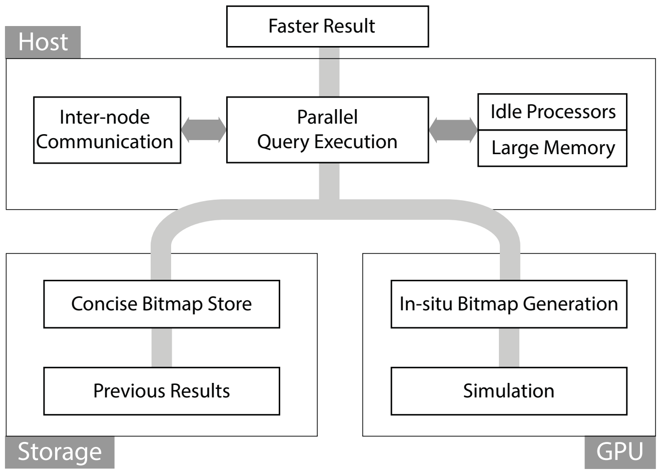


Fig. 3: An overview of the MoHA system. Concise bitmap indices reduce the I/O bottleneck between host and the accelerator and storage subsystems, while a parallel query processor utilizes the CPU resources.

### *B. Constructing an In-situ Scientific Analytics Engine for Heterogeneous Platforms*

At a high level, the query processor is structured as any of the common relational and array database engines [58], [59], [60]. Each query plan in MoHA is constructed using composable *operators*, each describing how the input data should be transferred, modified, aggregated, or processed one step at a time. The query processor executes the operators using appropriate computing and I/O resources to generate the results.

Although MoHA adheres to the standard model, the needs of in-situ queries and heterogeneous platforms require certain changes. First, the chosen operator model needs to consider the streaming nature of the simulation results. Specifically, our query engine adapts a push-based operator model [61], with each operator pushing a chunk of the output results array to its parent. This creates a more natural control flow and closer match to the asynchronous nature of the in-situ query processing. Second, because different operators can be run on different threads and devices to utilize the parallelism provided by heterogeneous platforms, special operators are provided for transferring chunks between the threads and from the accelerator to the CPU. Finally, MoHA represents the input as bitmaps indices to utilize the limited memory and communication capability of accelerators. A bitmap conversion operator is provided for that purpose.

### *C. Parallelization Within and Across Nodes*

One challenge of processing in-situ queries on heterogeneous platforms is utilizing the various available forms of parallelism in the system effectively. The execution model of MoHA permits parallelism both within and across compute nodes, through pipelining as well as data parallelism.

A query plan of MoHA is divided into different *fragments* that can run on different devices or machines concurrently. Different fragments are connected through pairs of send/receive operators, which transfer data from the fragments that produce the data to the fragments consuming it. The data transfer happens between the accelerators and the host memory, between different worker threads on the host CPU, and can happen across nodes also.

Within a node, MoHA spawns one or more worker threads to run the fragments instead of individual processes. This allows more efficient inter-fragments communication through shared memory space rather than costly IPC facilities. Also, global resources such as memory buffers are shared between different threads. When necessary, multi-consumer, multi-producer queues are used to buffer and transfer intermediate results as well as distribute tasks between different worker threads.

In addition to intra-node parallelism, MoHA supports parallelization across the nodes when in-situ analysis needs to combine results from simulation output from multiple nodes. An MPI-based asynchronous send/receive operator pair serves the purpose of shuffling the results between different nodes. Between processes, all data is transferred using point-to-point MPI directives. All MPI requests are funneled through one thread. In order to send a message, the input threads serialize the message and put it into a send queue. A looping thread then issues the send requests and monitors the status. The process on the receiving end is similar: the receiving thread puts buffers it receives into a queue. The working threads then deserialize the message and drive the query by calling the `consume()` API on its parent operator. One consideration is avoiding unnecessary memory allocation and duplication as much as possible. This is achieved by maintaining shared buffer pools and allowing buffers to be passed between threads as necessary.

## V. In-situ Bitmap Generation on a GPU

One key component of the MoHA system is a compact approximate representation of the simulation data. As we have discussed earlier, such a representation 1) reduces the memory pressure while processing queries; 2) accelerates data transfer operations; and 3) speeds up analytics. This section introduces the compact bitmap representation MoHA uses to perform in-situ analytics, and how such an index can be generated efficiently on GPUs.

### *A. GPU-friendly bitmap index representation*

Bitmaps were originally developed as indices for discrete values in relational datasets [62], by representing a range of similar contiguous values using the same bin. Later, they were found suitable as an index for scientific datasets [63]. Most recently, they have been used as an effective approximate representation of the data on which queries can be processed directly [34], [64], [57]. When simulations are executing on GPUs, and given that data transfer between GPU and CPU

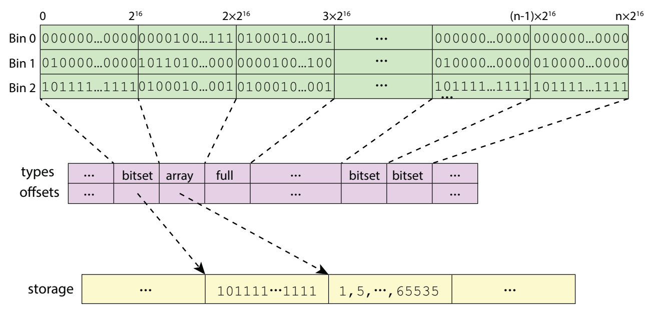


Fig. 4: An illustration of the bitmap storage layout of MoHA.

is likely to be a bottleneck, it is desirable to convert output simulation data to bitmaps on GPUs.

However, utilizing bitmaps on the GPU is not straightforward. This is because the highly parallel nature of many-core processors conflicts with the serial nature of *run-length encoding* mechanisms used by popular bitmap compression schemes [33], [37]. While there are prior works on generating bitmaps on GPUs [65], [66], they mostly focus on generating run-length-encoding (RLE) bitmap representations on the GPU. Thus, they are generating a serial representation on an accelerator designed to perform massive parallel computations. As a result, they require linear temporary spaces for execution and are not efficient [65], [66].

Instead, MoHA utilizes a segmented (chunked) [2] bitmap representation similar to Roaring bitmaps [38], [67]. In such a representation, a bitmap is divided to a number of *slices* of equal lengths, and each slice is compressed according to its *cardinality*, which is the number of `1` bits in the slice. This structure has the advantage of allowing efficient and parallel random access as well as storing each individual slice in a fashion that allows for an efficient size.

Because there are multiple buckets in the bitmap representation of a generated chunk, MoHA generates all the buckets for an input chunk together. The bitmap indices of a chunk with length $L$ will contain $\lceil \frac{L}{L_S} \rceil \times B$ slices, $B$ and $L_S$ being the number of buckets and the length of each segment, respectively. MoHA follows the example of Roaring and sets $L_S$ to $2^{16}$. Before explaining the rationale of such a setting, we first discuss how each individual slice is stored.

The physical representation of a slice depends on its cardinality, *i.e.*, the number of values it contains. The current implementation of MoHA stores one slice in one of three formats: *bitset*, *array*, or *full*. A slice is always stored in the format that occupies the least storage. As its name suggests, a *bitset* slice stores the values as a contiguous bitset using $2^{16}$ bits (4096 bytes) words, with a `1` bit representing a value at the position, and a `0` bit otherwise. An *array* slice stores the offsets of the values in it using 16-bit integers, using $2C$ bytes for a slice with cardinality $C$. Finally, a *full* slice just represents a slice that contains all ones, it occupies no extra spaces other than its metadata.

Currently, a $2^{16}$ segment size is used because indices in the segment can be stored as byte-aligned 16-bit integers for fast random access as well as compact storage, without wasting bits [38]. In principle, more flexible chunk size can be used by sacrificing a few extra bits per element, or utilizing a more compact small integer encoding [68]. Examination of the trade-offs involved is an interesting question for future research but beyond the scope of current work.

Of course, certain slices can be stored more efficiently if more representations are introduced. For example, run-length encoding can be used to compress a slice with long, contiguous range. Similarly, for the slices containing all but a few values in the dataset, representing the slice as its negation might save space. However, more complex schemes can complicate slice generation as well as query processing.

One key difference of our representation from Roaring [38] is its storage layout. Such a layout plays an essential role in ensuring the bitmap being GPU-friendly. This is because dynamic allocation on GPUs, while possible, is expensive and causes fragmentation, which prevents data from efficiently being moved to the host memory [69], [70], [71], [72], [73]. MoHA always stores all the slices in one continuous storage. The bitmap representation keeps the storage offset and type of each slice as metadata. As the size of a slice can be inferred from the offsets of adjacent slices, this allows any slice to be accessed efficiently. Each kind of metadata is kept in separate arrays for better parallelism and data locality.

## *B. Space-efficient bitmap generation on a GPU*

When new data emerges from simulation, the `to_bitmap()` operator converts individual input chunks to the bitmap formats MoHA uses. This subsection introduces the space-efficient bitmap generation method used by MoHA, which requires only $O(S)$ extra space, with $S$ being the total number of slices in the bitmap. The bitmap generation is conducted in two passes, to overcome inefficient memory allocation on GPUs. The first pass computes the necessary metadata as well as the total space requirements. After that, the second pass produces the actual slices to be stored.

Algorithm 1 gives an overview of this process. The first pass (Line 4 - 9) computes the sizes and types of each slice. Because how a slice is stored depends on its cardinality, the first pass essentially needs to compute the cardinality of each slice. Because each value belongs to precisely one bucket of the bitmap indices, this is essentially constructing a histogram on the bucket IDs of values. When the bucket numbers are small, a histogram can be produced by computing

[2]The meaning of *chunk* is overloaded in different contexts. To avoid confusion, this paper reserves chunk to represent a hyper-rectangular block of the input array. Instead, it refers to each fixed-length partition of the input dataset as a *segment*, and each fixed-length partition of the bitmap indices as a *slice*.

**Algorithm 1:** Generating bitmap indices on GPU.

```
1  function generate_bitmap(input chunk I, total buckets B):
2      $S \leftarrow \lceil \frac{L_I}{L_C} \rceil$ ;
3      for $s \in [0, S)$ in parallel blocks do
4          $card[s] \leftarrow$ cardinality histogram of segment $s$;
5          for $b \in [0, B)$ in parallel threads do
6              Compute $types[s, b]$, $sizes[s, b]$ according to $card[s]$;
7          end
8      end
9      $offsets \leftarrow$ prefix_sum$(sizes)$;
10     Allocate necessary storage $data$ for the index;
11     for $idx \in [0, S \times B)$ in parallel blocks do
12         $s \leftarrow idx$ div B, $b \leftarrow idx$ mod B;
13         Generate a $types[s, b]$ slice at $offsets[s, b]$;
14     end
15     return $types, offsets, data$;
16 end
```

partial histograms in shared memory per GPU thread, and aggregating the results; otherwise, atomic instructions are needed for aggregating partial counts [74]. After the histogram generation, the type and size of each slice are computed based on the computed cardinality. Then the total storage needs of each are determined, and the memory is allocated. (Line 10-11). Then, a second pass on the input produces the actual physical slice representation (Line 12 - 15). In this pass, the bitmap representation of each slice is generated by a corresponding GPU thread block, that is, if a bitmap index has $B$ buckets and $B \times S$ slices, $B \times S$ blocks are allocated for the index generation. Each value in the dataset is scanned $B$ times, once for each bucket. This may not appear efficient, as theoretically, a single scan can generate all the blocks at the same time. However, we find the saved work does not translate to saved generation time. Two factors can explain this. First, efficient L2 cache mechanisms in the GPUs reduces the overhead of reading the data multiple times, and second, allocating multiple slices in the same block increases the complexity of the code, adds more register pressure, and reduces the efficiency of the block schedule mechanism of stream processors.

Next, we describe how each kind of slices can be generated (Line 13 in Algorithm 1). For *bitset* chunks, the generation is straightforward. Each GPU warp generates one 32-bit word of the bitsets at a time. This is achieved by each GPU thread generating one single bit of the bitset, and then using warp-wide synchronization instructions (`ballot` in CUDA) to generate the bitset word. Although it is also possible to avoid the warp-wide communication by letting each GPU thread process 32 input values and generate one bitset word on its own, in our evaluation it did not yield better performance.

An array slice stores the values in the slice as a list of

**Algorithm 2:** Generating a array bitmap slice.

```
1  function generate_array_slice(slice storage data, input segment S, bucket range [min,max)):
2      for $i \in [0, L_S)$ in parallel threads do
3          $in\_bucket \leftarrow$ if $\mathbf{S[i]} \in [\mathbf{min}, \mathbf{max})$ then 1 else 0;
4          $offset \leftarrow exlusive\_scan(in\_bucket)$;
5          $data[offset] \leftarrow i$;
6      end
7      return $data$;
8  end
```

16-bit offsets. *Array* slices can be generated using a modified parallel scan [75] algorithm. In this approach, first, a boolean array is generated, indicating whether a value in the slice is inside the bucket. An exclusive prefix sum is then run on this generated array; resulting in a list of the index of each value in the bucket. Finally, if the original value is inside the bucket, its offset is written into the index of the bucket, producing the array chunk. Note that because the boolean array and the index list does not need to be actually materialized, this process does not need any extra temporary space either. Algorithm 2 illustrates this point.

Finally, *full* chunks and empty arrays do not occupy storage, and can be skipped in the second pass. Adding everything up, the algorithm does not need any extra global memory space for the bitmap generation, resulting in a very memory-efficient bitmap generation algorithm.

## VI. Performing Complex Analytics Using Bitmaps

As we have stated throughout, increasingly, scientists desire more complex analytics from their in-situ analytics engines. This section describes how MoHA enables the development of complex in-situ analytics. We start by showing how certain primitive operations can be effectively processed on the bitmap representation, and then discuss implementation of several complex queries, including the two queries we had previously described in Section III.

One of the issues for the system is moving the generated bitmaps to the host memory for analytics. This is done by the `gpu_send()` and `gpu_recv()` operators. When the `gpu_send()` operator receives a generated bitmap chunk on GPU, it allocates a host buffer according to its size and initializes a (potentially asynchronous) transfer to copy the chunk to the host memory. As asynchronous GPU-CPU transfers require pinned buffers, memory allocations could significantly draw down the performance. In such cases, MoHA employs thread-safe memory buffer pools to reuse allocated buffers if possible. For the experiments reported in the paper, only synchronous transfers were used. This is because a) the GPU device memory is usually a scarce resource while performing in-situ analytics, and asynchronous transfer of the data requires allocating additional buffers on the GPU side, and b) the applications used in the paper was not designed with the overlapping capabilities of the CUDA language in mind.

## A. Analytic Primitives on Bitmaps

This subsection discusses how several basic operations are implemented on bitmaps in MoHA. These operations are used as the building blocks of complex analytical queries.

**Filter** A filter operator performs a selection operation on *attribute-based* predicates. When data is represented as a bitmap index, such operation can be decomposed into a number of logical bitwise operations: For an array of $n$ attributes, a filter on conditions $min_i < \mathcal{A}_i < max_i, i \in [1, n]$ can be performed by doing a BITWISE OR operation on all selected buckets in each attribute $A_i$, and then a BITWISE AND operators on all the bitmaps generated by the BITWISE OR operations. MoHA performs such an operation by improving the implementation described in the context of Roaring bitmaps [38], [67] in terms of memory use. Specifically, there is no need to materialize the entire intermediate generated bitmaps. Instead, the bitmaps indices are processed segment by segment. Because there is no guarantee about the cardinality of two or operations when performing BITWISE OR operations between buckets, a bitset slice is allocated to store the intermediate slice, no matter what type of slice the input slice is. This also has the advantage of accelerating the BITWISE AND operation.

**Subset** A subset operation performs a selection on dimension-based predicates. For example, a user might want to query only an interest subset region of the entire timestep. For such selections, a *bitset slice mask* is generated on the fly according to the selected predicate(s). A BITWISE AND operation is then performed between the mask and the bitmap indices for the final result.

**Aggregation** Previous work has explored fast approximate aggregation using bitmap representations [34]. The key idea is keeping the total aggregations per attribute, and then use the population count of filtered result to estimate the final result based on saved aggregations. Thus, the problem reduces to the problem of counting setting bits in the bitmap representation, which is straightforward.

**Join** Array join aims at finding cell pairs that certify certain join conditions. Equal join predicates on dimensions and attributes can be implemented by merely performing BITWISE AND operations between the selected attributes and dimensions. More complex joining types are also considered in prior works [76], [77].

## B. Example queries

Modern data-driven science demands more complex analysis then simple filter, aggregagation, or even join. However, many complicated queries can be implemented with the primitives introduced. This subsection discusses a few examples, including two of the examples mentioned in Section IV and a similarity query.

*1) Simulation Similarity:* The first query aims at determining the similarity between the simulation being run and a previous baseline. A user might want to have a look at how different each simulation is from previous simulations, to determine whether an adjustment makes sense. Such similarity can be computed by computing the number of similar attributes between corresponding time steps; that is, for each attribute, we count the cells at the same dimensional position of the time steps being compared with an attribute value difference below a certain threshold. The scores of each attribute are then added up to compute the similarity.

In terms of implementation within our framework, this can be seen as a simple aggregation on top of a *join* operation between different arrays [32], [76]. As previously mentioned, with bitmaps, join operation can be supported by doing BITWISE AND between corresponding bins. The bitmap accelerates the query in two ways: it saves the I/O cost of reading the data from disk and transferring from GPU to CPU, as well as allowing faster comparisons between the elements. A similar process can also be used to evaluate various other similarity metrics [34] between time steps.

*2) Contrast Set Mining:* As mentioned in Section IV, contrast sets mining looks for same groups in different datasets that possess a largest difference according to a quality function. In order to search for such contrast sets, the search range is discretized into a number of bins. A pruned brute-force search strategy based on a set-enumeration tree can be used to solve the problem. The searching starts from a subset containing all the elements in the datasets, and adding one filter condition at a time [56], [57]. For each subset, a filter and aggregation operation computes the mean and size of the subsets, and then the quality of the contrast sets is computed. As discussed in the previous subsection, such filter and aggregation operation can be performed effectively on bitmap representations.

Again, the bitmap representation accelerates the I/O by reducing the data being moved, as well as results in faster filtering and aggregation in smaller selectivity scenarios. Being a naturally discretized data representation, utilizing bitmap indices to process this query also has the additional benefits of providing results as accurate as the original representation provided the number of bins are chosen correctly.

*3) Overlapped Interest Region:* The final query looks for stable features across adjacent timesteps. It can be implemented by filtering the two adjacent timesteps for interesting cells, a joining operation is then performed to find the overlapped interest cells. These cells are then merged to find the largest contiguous region. Here, bitmap indices accelerate the query by reducing the transferring time from GPU to CPU, as well as facilitating faster filter operation.

## C. Discussion

One question is what kind of queries benefit from the bitmap representations of MoHA. Bitmaps compactly preserve both spatial and value information about multi-dimensional arrays and allow fast intersection and union operations. Thus, they can support a wide range of query with either no or modest sacrifice on the accuracy – this include approximate aggregation [34], interesting regions [35], membership queries [78], operations associated with visualization [79], correlation mining [64] and other queries [57], [80], [76]. In general, queries that explores array subsets based on values and relationships between different datasets or their subsets can be easily and efficiently implemented using the primitives discussed above. In each case, performance advantages arise from fast filtering and bitwise operations. As is the case with most summary representations, the accuracy loss can be a concern in certain situations, but can be reduced using better binning schemes [34].

## VII. Experimental Evaluation

This section empirically evaluates the in-situ query performance of the MoHA system on a HPC cluster using real-world queries applied on actual output from simulations. The following questions are explored:

- How does the cost of generating bitmaps on GPUs and then transferring them to CPUs approximate compare against the cost of just transferring the original (plain) simulation results? (§VII-B)
- How efficient is query execution on bitmaps compared to execution using original data? (§VII-C)
- How does MoHA perform when in-situ queries are parallelized across nodes? (§VII-D)

### A. Experimental Setup

*Configuration.* This paper evaluates the MoHA system using a cluster with the following features. Each node has a Intel Xeon 2586 v4 vCPU and a NVIDIA V100 SXM2 GPU. The nodes are connected with 10 Gigabytes/second network. CPU-GPU connectivity is through the PCI-E connection with a measured bandwidth of $\sim$ 11 GiB/s. The cluster uses a 4.8 TiB AWS FSx Lustre parallel file system, with a 938 MiB/s baseline throughput and a 6x burst throughput. The system runs an Ubuntu LTS 18.04 operating system with a Linux kernel of version 4.15 and Lustre client 2.10.8. While several aspects of this cluster match features of clusters available from traditional supercomputing centers, it happens to be hosted on AWS cloud from Amazon [81]. The instances were allocated in the same *placement group* to ensure performance.[3] Furthermore, in our experiments, multiple executions of the same job at different times showed no substantial difference in execution time, indicating little interference from any other jobs. As noted by others as well [82], in recent years it has become possible to create supercomputing like capabilities through specific choice of offerings from the cloud.

[3]Placement group *"packs instances close together inside an Availability Zone, enabling workloads to achieve the low-latency network performance necessary for tightly-coupled node-to-node communication that is typical of HPC applications"*, see https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

The MoHA systems is programmed using C++ and CUDA. The system is compiled using version of 8.4.0 of gcc compiler and the CUDA 10.2 platform. The highest level of optimization (`-O3`) is used for all experiments. All communication between nodes are implemented using the asynchronous MPI primitives of MPICH 3.3.2. All the nodes are always placed under the same placement group for faster inter-node communication. On each node, 8 worker threads are used in query processing unless otherwise specified.

*Datasets.* Two real-world benchmarks are used in our experiments. LULESH [83], [84] is a hydro-dynamic inspired simulation core, widely used in other simulation-based computer science research [85], [34], [86]. The simulation computes the interaction of materials using a mesh-based methods, capturing the coordinates of the matters as well as their force, velocity and acceleration. PIC [87] is a one-dimensional electrostatic particle-in-cell simulation program emulating the movement of ions and electrons in the electronic fields, computing the positions, velocity and electronic field strength at each particle based on its electric potential and charge density. We use 64 equal-width bins based on the value domain range for all of our experiments.

*Experimental Methodology.* All experiments are run at least 5 iterations and the average results are reported. System cache is cleared before all I/O-related experiments. The order of experiments is randomized to amortize underlying system performance fluctuation. The reported processing time excludes simulation initialization time and the time of result materialization to highlight the performance of actual in-situ processing.

### B. Bitmap Generation and Transfer Costs

The basic idea in MoHA is to replace simulation output on GPUs by their bitmap representation. While this reduces the size of data to be transferred, generating the bitmap indices also involves more computation. The first set of experiments investigates how the bitmap generation overhead compares with the savings due to transferring less data. The simulation time is also shown as a reference point. All the simulations are executed for 1000 time steps, and we vary the amount of data generated by controlling the problem size. We choose five problem size for each simulation, so that difference of the data amount generated per time step between adjacent settings is roughly two times.

Figure 5 shows the results with LULESH simulations. The topline number indicates that transferring bitmap indices is consistently faster than transferring the original data. On average, bitmaps are $1.4\times$ faster compare with the original representation. Also, transferring bitmaps only adds a relatively small overhead compared with the simulation. With a problem size of $256^3$, the transfer only adds an overhead of $13\%$ compared with the simulation-only scenario. Under the

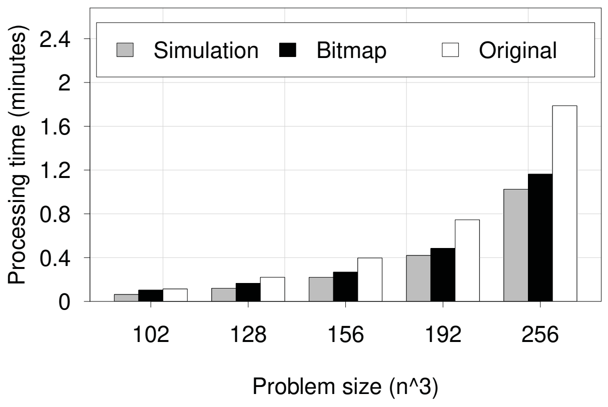


Fig. 5: Comparison between 1) GPU generation of bitmaps and transfer to CPUs (`Bitmap`) and 2) transfer of original simulation output from GPU to CPU (`Original`) (LULESH simulation, 1000 iterations).

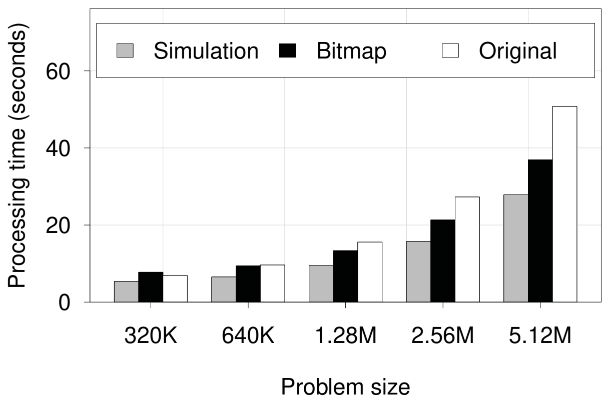


Fig. 6: Comparison between 1) GPU generation of bitmaps and transfer to CPUs (`Bitmap`) and 2) transfer of original simulation output from GPU to CPU (`Original`) (PIC simulation, 1000 iterations).

hood, profiling data shows that when the problem size is $156^3$, the cost of generating and copying the bitmap representation is around 29% of copying the original data, with the majority of time spending in bitmap generation. Another observation is, even though the data amount generated per second stayed relatively flat as the time step size increases, the relative performance of the bitmap representation improves. This performance strength can be explained by two factors. First, as the problem size increases, fewer transfers need to be issued for the same amount of data, reducing the overhead of initializing transfers; second, as each GPU thread block handles a slice of the bitmap indices, increasing the time step size increases the number of threads that can be scheduled, which gives GPU scheduler better opportunity to improve device utilization.

Figure 6 explores the results of the same experiments on

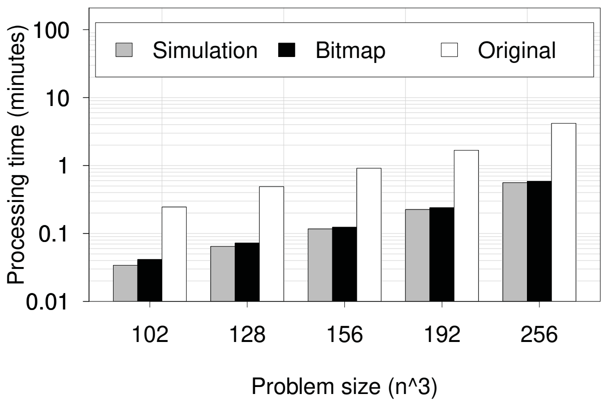


Fig. 7: Performance of the Simulation Similarity query (LULESH simulation, 200 iterations).

the PIC simulation. While transferring the original data is somewhat faster when the problem size is smaller, the bitmap representation gradually outperforms the original representation as the size of the problem increases. When the problem size is 5.12M, MoHA enjoys a speedup of $1.4\times$ compared with the original representation. This matches our earlier observation, which is that the generation of the bitmap indices is more efficient when the GPU has more data to work with. The profiling data confirms such view; at the problem size of 320K, while the transfer of the bitmap representation only costs $0.2\times$ of transferring the original data per time step, generating such representation costs $2.1\times$ of the transferring cost for the original data, which more then cancels the benefits of having a compact representation. On the contrary, when the problem size is 2.56M, the bitmap generation and transferring only costs $0.3\times$ and $0.1\times$ of copying the original data, respectively. Finally, note that the bitmap representation is seeing less advantage on the PIC dataset. This is because compared with LULESH, its bitmap representation is not as compact – while the bitmap representation of LULESH only occupies less than 2% of its original size, the bitmap representation of the PIC datasets are around 10% of the original data. Overall, the benefits of saved transferring significantly outweigh the overheads of bitmap generation with both the simulations.

## C. Query Execution Performance with MoHA

The next question is how efficiently MoHA can handle various query workloads. Similar to the last subsection, the set of experiments compares the performance of querying the bitmap representation of MoHA with querying the generated data directly using the example queries mentioned in section VI.

The *simulation similarity* query evaluates the number of cells that are similar enough compared with a previous benchmark. In this experiment, a 200-timestep simulation is compared with the output from a baseline run. In order to generate the baseline, for LULESH, we use a previous run with

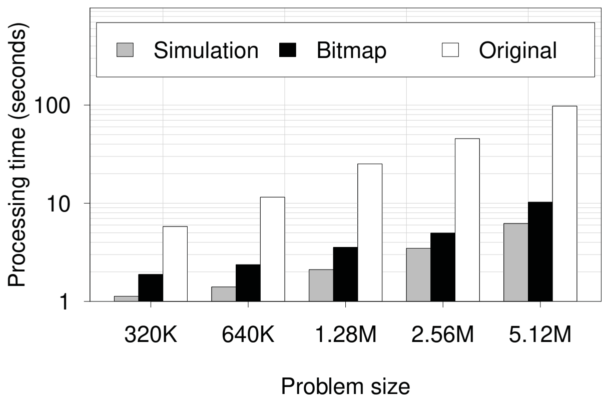


Fig. 8: Performance of the Simulation Similarity query (PIC simulation, 200 iterations).

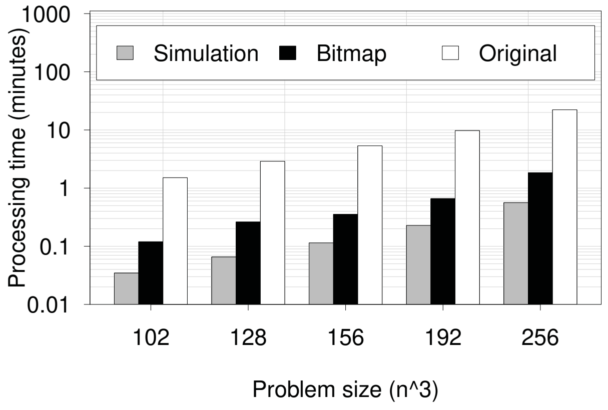


Fig. 9: Performance of the Contrast Set Mining query (LULESH simulation, 200 iterations).

a different balance parameter; for PIC, we use a randomized seed.

Figures 7 and 8 report the query performance on the output from LULESH and PIC simulations, respectively. Overall, executing the queries on the bitmap representation significantly accelerates the queries with both simulations. On the average, MoHA is $6.8\times$ faster with the LULESH simulation, while being $6.7\times$ faster on the PIC simulation. Also, the overhead of the added query processing compared with running the simulation itself remains small. More specifically, processing the in-situ query on the largest problem size tested on LULESH only costs $1.05\times$ time of the simulation, and the overhead of querying the largest problem size on PIC is only $0.65\times$ of the simulation time. This is because on bitmap representation, the comparing between the simulation results can be done by simple bitwise AND operations, adding little overhead. Also, because bitmap representation is very compact (less than 10% of the original data in this case), the additional I/O costs are also very limited. On the other side, querying the original data requires reading a much larger data from a remote file system in addition to move the same amount of data from the device to the host memory, costing a slowdown of much as $9.5\times$.

While the *simulation similarity* query puts little pressure on the CPUs, there are more CPU-intensive queries. Next query being examined is *contrast set mining*, which evaluates which filtering condition produces a pair of subsets of a time step that “differs” the most. We again evaluate the performance of MoHA by comparing its performance against querying the original data on both simulations, with the number of iterations set to 200. When searching for subsetting predicates, all the dimensions and values are discretized into 64 equal-width bins for both the bitmap and original representations.

Figures 9 and 10 report the mean processing time of the query on both datasets. In general, it is much more slower to perform the contrast set mining queries on the plain datasets. Querying the bitmap representations of LULESH and PIC are

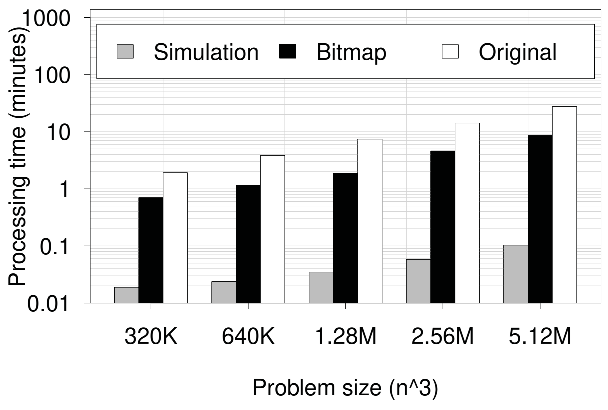


Fig. 10: Performance of the Contrast Set Mining query (PIC simulation, 200 iterations).

$13.1\times$ and $3.3\times$ faster compared with querying the original representations, respectively.

To illustrate the performance characteristics of the contrast set mining query, Table I breaks down five components of the overall time between processing the bitmap indices and the original data: the cost of bitmap generation, the time of transferring the data to the host, the sum of the CPU contrast set mining searching time across worker threads, and the I/O cost of reading data from the file system. [4] The breakdown indicates that, in most scenarios, the bitmap representation outperforms the original representation in all three parts of the in-situ query: In most cases, generating and transferring the data to the host outperforms transferring the original data directly by a factor of $2\times$ or more. The the benefits of the bitmap representation on reading array

[4]The simulation, generation and transfer costs here are measured on the device side using a GPU profiler, whereas the query processing and I/O measurement are measured using the high-resolution clock from the host side. The device side measurement does not reflect the cost of overhead coming from host side and accelerator driver.

TABLE I: Detailed Performance Data with the Contrast Set Mining Query.

| Benchmark | Problem Size | Simulation | Bitmap Generation |  | Transfer to Host |  | CSM Searching on CPUs |  | I/O |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | Bitmap | Original | Bitmap | Original | Bitmap | Original | Bitmap | Original |
| LULESH | 102 | 0.43 | 0.41 | 0 | 0.00 | 0.55 | 47.64 | 732.80 | 5.64 | 87.07 |
|  | 128 | 0.86 | 0.43 | 0 | 0.00 | 1.10 | 101.70 | 1405.79 | 11.97 | 166.73 |
|  | 156 | 1.59 | 0.46 | 0 | 0.01 | 1.99 | 127.85 | 2521.29 | 15.11 | 296.86 |
|  | 192 | 2.96 | 0.80 | 0 | 0.01 | 3.70 | 228.74 | 4572.72 | 27.05 | 540.71 |
|  | 256 | 6.99 | 1.52 | 0 | 0.01 | 8.78 | 663.36 | 10465.72 | 77.23 | 1244.18 |
| PIC | 320K | 0.24 | 0.41 | 0 | 0.04 | 0.25 | 379.69 | 833.86 | 45.11 | 97.04 |
|  | 640K | 0.57 | 0.46 | 0 | 0.06 | 0.50 | 669.86 | 1738.12 | 79.03 | 205.61 |
|  | 1.28M | 1.36 | 0.56 | 0 | 0.11 | 1.01 | 1226.11 | 3684.01 | 144.08 | 433.49 |
|  | 2.56M | 2.91 | 0.78 | 0 | 0.19 | 2.01 | 2263.00 | 6403.33 | 268.10 | 748.80 |
|  | 5.12M | 6.02 | 1.19 | 0 | 0.36 | 4.03 | 4258.32 | 13798.71 | 503.06 | 1616.64 |

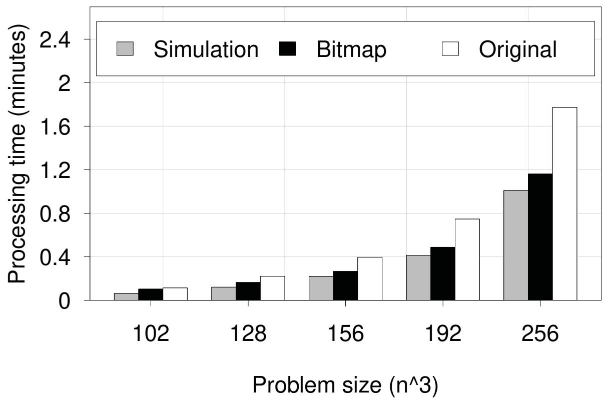


Fig. 11: Performance of the Overlapped Interest Region query (LULESH simulation, 1000 iterations)

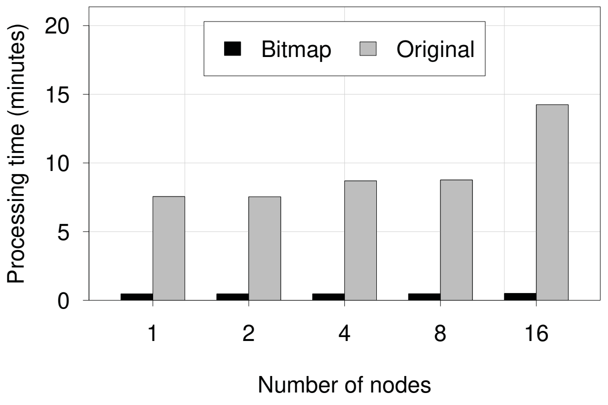


Fig. 12: Parallel performance of the Contrast Set Mining query (LULESH, 200 iterations).

data from the disk and as searching for the contrast sets are even more significant: searching for contrast sets on bitmaps and I/O can be accelerated by an order of magnitude. In general, this again demonstrates the versatility of the bitmap representation: it can not only accelerate the I/O performance of an in-situ query system, but can also accelerates the CPU-intensive queries as well. Another interesting observation is that the performance advantage of the bitmap representation are smaller with the PIC dataset in both the I/O and CPU category, which reflects that the data generated by PIC is not as skewed as the LULESH simulation.

Finally, we take a look at the *overlapped interest region* query, which searches for large, stable *hot regions* across time steps. Because the PIC dataset is an unstructured simulation and does not have dimensional structure suitable for such a query, this experiment focuses on the LULESH simulation. The result is shown in Figure 11.

While the bitmap representation does not have as large a speedup compared with the previous queries that involved I/O we do still see an average speedup of $1.4\times$. This is because the host CPUs are capable of keeping up with the inbound data, therefore, the major bottleneck is still the data movement from the GPU to CPU, which can be saved by utilizing the bitmap representation.

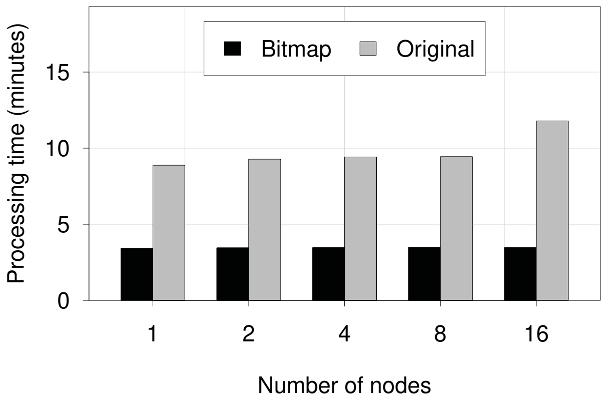


Fig. 13: Parallel performance of the Contrast Set Mining query (PIC, 200 iterations).

### *D. MoHA Performance on In-Situ Queries with Inter-Node Communication*

This subsection consider cases where the in-situ query needs communication across nodes. As each node simulates a part of the grid, contrast set mining task could be computing contrast sets across nodes. In our experiments, this query is applied on both the simulations, scaling them so that each node generates a fixed amount of data. We adjust the problem size so that each node generates $\sim 115$ MiB data per time step for LULESH; and $\sim 60$ MiB data for the PIC; and scale the number of nodes from 1 to 16.

Figures 12 and 13 reports the average processing time of the contrast set mining query. As the number of nodes increases, the query processing time of both the bitmap and original representation holds steady with only a slightly timing increase. However, at 16 nodes, the query processing time of plain representation significantly increases, while the query time of the bitmap representation holds steady. This can be attribute to the increased I/O cost. Because the parallel file system we are using have a fixed total bandwidth, as the total amount of data being read from the file system increases, the bottleneck of each node turns to I/O from computation; because the bitmap representation requires less I/O, it can scale out further.

## VIII. Related Work

**In-situ analytics on scientific data.** While the idea of *co-processing* has decades of history [88], it seems that the term *in-situ* methods was proposed in the Petascale era [89], [90]. Earlier in-situ scenarios were centered around visualization, and many visualization software tools such as Paraview Catalyst [12], VTK-m [20], and libsim [91] have added in-situ analysis functionality. One major concern was the ability of interactively exploring the visualization result after the simulation concludes. One method is extracting image samples that are potentially interested to users [92], [93]. Cinema [94], [95] advances such model by generating multiple composable images and generates explorable images and animations dynamically. Our work complements such model by considering simulation execution on heterogeneous platforms and complex queries (for example, overlapped interest regions across timesteps. The bitmap indices generated can also be useful for visualization tasks [79]. In another line of work, middleware systems such as ADIOS [96], [14] and GLEAN [15] and others have been developed (see Bauer *et al.* [21] for a comprehensive survey). The approaching Exascale era has created a momentum in this area [97], [2], [3], [4], [5], [6]. ALPINE [11] is an recent effort to provide a unified interface for different in-situ frameworks. Several recent works [4], [5], [6] utilize mathematical methods to reconstruct features for in-situ analytics; In-situ indexing methods have also been suggested for faster post hoc processing [98], [99], [100].

There has also been work on processing raw (scientific) data resident on disks *in-situ* without ingestion into a database. The NoDB [101] work formalized this approach in the context of relational data. However, it has also been beneficial for scientific data [102], [103], [104].

**Bitmap indices and compression.** Bitmap indices were originally proposed in relational database context [62], [105], [106]. FastBit [36], [35] popularized the idea of applying bitmap indices for faster query processing (for scientific data). Historically, variant schemes of run-length-encoding were used for compressing the data [33], [27], [37]. Recently, Roaring bitmaps [38], [67] gain popularity due to its high efficiency in query processing [107]. There are also recent efforts to increase the accuracy of such representation [108], [77] or further reduce its size. Lang *et al.* [39] proposed tree-encoded bitmaps for space saving. While bitmaps can be a secondary index, it can also frequently be used as a primary index or an approximate representation for faster query processing [63], [34]. Our paper continue this trend and considers whether such a representation can be suitable for this heterogeneous era, and can be the basis for a composable system for developing queries. Prior research about bitmap building on GPU focuses on building RLE encoding using GPU acceleration [66], [65]. However, this requires either sorting the data or constructing the uncompressed bitmap beforehand, causing time and space overhead that is not desirable when accelerators like GPUs are involved.

## IX. Conclusion

We have introduced MoHA, an in-situ query system designed for complex and effective analytics on today's dense and heterogeneous HPC platforms. We have presented a GPU-friendly bitmap representation, how such a representation can be efficiently generated on GPUs, and how such a representation can support a variety of complex in-situ queries. Our evaluation on real-world simulations confirms that MoHA outperforms the baseline (which does not use a summary or compressed representation) by an average speedup of $\sim 1.4\times$ to $\sim 13\times$, while also saving memory and I/O bandwidth.

*Acknowledgements*

This work was funded by NSF awards SHF-1526386, CCF-1629392, CCF-2007793 and OAC-2034850.

## References

[1] R. Haimes, "pv3-a distributed system for large-scale unsteady cfd visualization," in *32nd Aerospace Sciences Meeting and Exhibit*, 1994, p. 321.

[2] Y. C. Ye, T. Neuroth, F. Sauer, K.-L. Ma, G. Borghesi, A. Konduri, H. Kolla, and J. Chen, "In situ generated probability distribution functions for interactive post hoc visualization and analysis," in *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2016, pp. 65–74.

[3] N. Seekhao, J. JaJa, L. Mongeau, and N. Y. Li-Jessen, "In situ visualization for 3d agent-based vocal fold inflammation and repair simulation," *Supercomputing frontiers and innovations*, vol. 4, no. 3, p. 68, 2017.

[4] S. Dutta, H.-W. Shen, and J.-P. Chen, "In situ prediction driven feature analysis in jet engine simulations," in *2018 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 2018, pp. 66–75.

[5] S. Dutta, J. Woodring, H.-W. Shen, J.-P. Chen, and J. Ahrens, "Homogeneity guided probabilistic data summaries for analysis and visualization of large-scale data sets," in *2017 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 2017, pp. 111–120.

[6] K.-C. Wang, J. Xu, J. Woodring, and H.-W. Shen, "Statistical super resolution for data analysis and visualization of large scale cosmological simulations," in *2019 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 2019, pp. 303–312.

[7] O. Fernandes, D. S. Blom, S. Frey, A. H. Van Zuijlen, H. Bijl, and T. Ertl, "On in-situ visualization for strongly coupled partitioned fluid-structure interaction," *Coupled Problems*, 2015.

[8] D. Morozov and G. Weber, "Distributed merge trees," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442526

[9] E. P. Duque, D. E. Hiepler, R. Haimes, C. P. Stone, S. E. Gorrell, M. Jones, and R. A. Spencer, "Epic-an extract plug-in components toolkit for in-situ data extracts architecture," in *22nd AIAA Computational Fluid Dynamics Conference*, 2015, p. 3410.

[10] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger, "Freeprocessing: Transparent in situ visualization via data interception," in *Eurographics Symposium on Parallel Graphics and Visualization: EG PGV:[proceedings]/sponsored by Eurographics Association in cooperation with ACM SIGGRAPH. Eurographics Symposium on Parallel Graphics and Visualization*, vol. 2014. NIH Public Access, 2014, p. 49.

[11] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*. ACM, 2017, pp. 42–46.

[12] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "ParaView catalyst: Enabling in situ data analysis and visualization," in *Proceedings of ISAV 2015: 1st International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, Held in conjunction with SC 2015: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 25–29.

[13] H. Childs, K.-L. Ma, H. Yu, B. Whitlock, J. Meredith, J. Favre, S. Klasky, N. Podhorszki, K. Schwan, M. Wolf *et al.*, "In situ processing," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2012.

[14] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[15] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2011, pp. 1–11.

[16] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell *et al.*, "The design, deployment, and evaluation of the coral pre-exascale systems," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 661–672.

[17] Argonne National Laboratory, "Aurora," 2020. [Online]. Available: https://press3.mcs.anl.gov/aurora/

[18] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky, "Landrush: Rethinking in-situ analysis for gpgpu workflows," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 32–41.

[19] D. Thompson, S. Jourdain, A. Bauer, B. Geveci, R. Maynard, R. R. Vatsavai, and P. O'Leary, "In situ summarization with vtk-m," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017, pp. 32–36.

[20] K. Moreland, C. Sewell, W. Usher, L. T. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. L. Ma, H. Childs, M. Larsen, C. M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, 2016.

[21] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms," in *Computer Graphics Forum*, vol. 35, no. 3. Wiley Online Library, 2016, pp. 577–597.

[22] N. Buchanan, S. Calvez, P. Ding, D. Doyle, C. Green, A. Himmel, B. Holzman, J. Kowalkowski, A. Norman, M. Paterno *et al.*, "Enabling neutrino and antineutrino appearance observation measurements with hpc facilities."

[23] C. Wang, D. Chen, C. Wu, and Q. Hu, "Data compression with homomorphism in covering information systems," *International Journal of Approximate Reasoning*, vol. 52, no. 4, pp. 519–525, 2011.

[24] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," in *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, no. 2, 1997, pp. 38–49.

[25] M. C. Wu and A. P. Buchmann, "Encoded bitmap indexing for data warehouses," in *Proceedings - International Conference on Data Engineering*. IEEE, 1998, pp. 220–230.

[26] M. C. Wu, "Query Optimization for Selections using Bitmaps," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 28, no. 2, pp. 227–238, 1999.

[27] K. Wu, W. Koegler, J. Chen, and A. Shoshani, "Using bitmap index for interactive exploration of large datasets," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, vol. 2003-January. IEEE, 2003, pp. 65–74.

[28] R. R. Sinha, S. Mitra, and M. Winslett, "Bitmap indexes for large scientific data sets: A case study," in *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, vol. 2006. IEEE, 2006, pp. 10—-pp.

[29] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Transactions on Database Systems*, vol. 32, no. 3, pp. 16—-es, 2007.

[30] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, and R. D. Ryne, "Parallel index and query for large scale data analysis," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[31] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu, "SDQuery DSI: Integrating data management support with a wide area data transfer protocol," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2013, pp. 1–12.

[32] E. Soroush, M. Balazinska, and D. Wang, "{ArrayStore}: a storage manager for complex parallel array processing," in *SIGMOD*. ACM, 2011.

[33] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg, "Notes on design and implementation of compressed bit vectors," Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, Tech. Rep., 2001.

[34] Y. Wang, Y. Su, and G. Agrawal, "A novel approach for approximate aggregations over arrays," in *ACM International Conference Proceeding Series*, vol. 29-June-20. New York, New York, USA: ACM Press, 2015, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2791347.2791349

[35] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. J. Otoo, V. Perevoztchikov, A. Poskanzer,

O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang, "FastBit: interactively searching massive data," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012053, 2009. [Online]. Available: http://iopscience.iop.org/1742-6596/180/1/012053

[36] K. Wu, "FastBit: an efficient indexing technology for accelerating data-intensive science," *Journal of Physics: Conference Series*, vol. 16, no. 1, pp. 556–560, 2005.

[37] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data and Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.

[38] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Software - Practice and Experience*, vol. 46, no. 5, pp. 709–719, 2016.

[39] H. Lang, A. Beischl, V. Leis, P. Boncz, T. Neumann, and A. Kemper, "Tree-encoded bitmaps."

[40] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic Sample Selection for Approximate Query Processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003, pp. 539–550.

[41] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang, "Sample + Seek: Approximating aggregates with distribution precision guarantee," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. 26-June-2016, 2016, pp. 679–694.

[42] S. Chaudhuri, B. Ding, and S. Kandula, "Approximate Query Processing: No silver bullet," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. Part F127746, 2017, pp. 511–519.

[43] Y. Chen and K. Yi, "Two-level sampling for Join size estimation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. Part F127746, 2017, pp. 759–774.

[44] S. Guha and K. Shim, "A note on linear time algorithms for maximum error histograms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 7, pp. 993–997, 2007.

[45] Y. E. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query-answers," in *VLDB*, vol. 99, 1999, pp. 7–10.

[46] S. Muthukrishnan, V. Poosala, and T. Suel, "On rectangular partitionings in two dimensions: Algorithms, complexity, and applications," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1540. Springer, 1998, pp. 236–256.

[47] J. S. Vitter and M. Wang, "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 28, no. 2, pp. 193–204, 1999.

[48] S. Muthukrishnan, "Subquadratic algorithms for workload-aware haar wavelet synopses," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2005, pp. 285–296.

[49] Y. Matias, J. S. Vitter, and M. Wang, "Wavelet-based histograms for selectivity estimation," in *SIGMOD Record*, vol. 27, no. 2, 1998, pp. 448–459.

[50] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00*, vol. 10, no. 2-3, pp. 111–122, 2000.

[51] A. Montanaro, "The quantum complexity of approximating the frequency moments," *Quantum Information and Computation*, vol. 16, no. 13-14, pp. 1169–1190, 2016.

[52] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[53] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, apr 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0196677403001913

[54] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2380 LNCS. Springer, 2002, pp. 693–703.

[55] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1-3, pp. 1–294, 2011.

[56] S. D. Bay and M. J. Pazzani, "Detecting group differences: Mining contrast sets," *Data mining and knowledge discovery*, vol. 5, no. 3, pp. 213–246, 2001.

[57] G. Zhu, Y. Wang, and G. Agrawal, "SciCSM: novel contrast set mining over scientific datasets using bitmap indices," *Proceedings of the 27th International Conference on Scientific and Statistical Database Management - SSDBM '15*, pp. 1–6, 2015.

[58] G. Graefe, "Volcano—An Extensible and Parallel Query Evaluation System," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994.

[59] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–34, 2014.

[60] J. Rogers, R. Simakov, E. Soroush, P. Velikhov, M. Balazinska, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, S. Zdonik, A. Smirnov, K. Knizhnik, and P. G. Brown, "Overview of SciDB: Large scale array storage, processing and analysis," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 963–968.

[61] T. Neumann, "Efficiently compiling efficient query plans for Modern Hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[62] P. E. O'Neil, "Model 204 architecture and performance," in *International Workshop on High Performance Transaction Systems*. Springer, 1987, pp. 39–59.

[63] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel, "HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, 2006, pp. 149–158.

[64] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 61–72.

[65] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri, "Indexing million of packets per second using GPUs," *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pp. 327–332, 2013. [Online]. Available: https://dl.acm.org/citation.cfm?id=2504756

[66] W. Andrzejewski and R. Wrembel, "GPU-WAH: Applying GPUs to compressing bitmap indexes with word aligned hybrid," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6262 LNCS, no. PART 2. Springer, Berlin, Heidelberg, 2010, pp. 315–329. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15251-1{_}26

[67] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai, "Roaring Bitmaps: Implementation of an Optimized Software Library," *arXiv preprint arXiv:1709.07821*, 2017.

[68] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," in *Proceedings - International Conference on Data Engineering*, vol. 2006. IEEE, 2006, p. 59.

[69] I. Gelado and M. Garland, "Throughput-oriented GPU memory allocation," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 2019, pp. 27–37.

[70] S. Widmer, D. Wodniok, N. Weber, and M. Goesele, "Fast dynamic memory allocator for massively parallel architectures," in *ACM International Conference Proceeding Series*, 2013, pp. 120–126.

[71] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU," in *2012 Innovative Parallel Computing, InPar 2012*. IEEE, 2012, pp. 1–10.

[72] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, "Scalable SIMD-parallel memory allocation for many-core machines," *The Journal of Supercomputing*, vol. 64, no. 3, pp. 1008–1020, 2013.

[73] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. M. Hwu, "XMalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*. IEEE, 2010, pp. 1134–1139.

[74] V. Podlozhnyuk, "Histogram calculation in cuda," *Technical report, NVIDIA*, 2007.

[75] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

[76] R. Ebenstein *et al.*, "Bitjoin: Executing range based joins in distributed environments," 2018.

[77] H. Xing and G. Agrawal, "Accelerating array joining with integrated value-index," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, 2019, pp. 145–156.

[78] B. Yildiz, K. Wu, S. Byna, and A. Shoshani, "Parallel membership queries on very large scientific data sets using bitmap indexes," *Concurrency Computation*, vol. 31, no. 15, p. e5157, 2019.

[79] K. Stockinger, J. Shalf, W. Bethel, and K. Wu, "DEX: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization," *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, pp. 35–44, 2005.

[80] S. Shohdy, Y. Su, and G. Agrawal, "Load Balancing and Accelerating Parallel Spatial Join Operations Using Bitmap Indexing," in *Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015*. IEEE, 2016, pp. 396–405.

[81] A. Muni and J. Hansen, "Amazon web services," *Dr. Dobb's Journal*, vol. 30, no. 12, pp. 66–67, 2005.

[82] N. S. Holliman, M. Antony, J. Charlton, S. Dowsland, P. James, and M. Turner, "Petascale cloud supercomputing for terapixel visualization of a digital twin," *IEEE Transactions on Cloud Computing*, 2019.

[83] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[84] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, aug 2013. [Online]. Available: https://computing.llnl.gov/projects/co-design/lulesh

[85] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-January, no. January. IEEE, 2014, pp. 647–658.

[86] H. Zhang and H. Hoffmann, "PoDD: Power-capping dependent distributed applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2019, pp. 1–23.

[87] L. Brieda, *Plasma Simulations by Example*, 1st ed., 2019.

[88] R. Heiland and M. P. Baker, "A survey of co-processing systems," *CEWES MSRC PET Technical Report*, pp. 52–98, 1998.

[89] H. Childs, "Architectural challenges and solutions for petascale postprocessing," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 12012.

[90] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, 2010.

[91] T. Kuhlen, R. Pajarola, and K. Zhou, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, vol. 10. Eurographics Association Aire-la-Ville, Switzerland, 2011, pp. 101–109.

[92] J. Chen, I. Yoon, and E. W. Bethel, "Interactive, Internet delivery of visualization via structured prerendered multiresolution imagery," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 2, pp. 302–312, 2008.

[93] Y. Ye, R. Miller, and K.-l. Ma, "In Situ Pathtube Visualization with Explorable Images," in *Egpgv*. Eurographics Association, 2013, p. 2312.

[94] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An Image-Based Approach to Extreme Scale in Situ Visualization and Analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-January, no. January. IEEE, 2014, pp. 424–434.

[95] P. O'Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen, "Cinema image-based in situ analysis and visualization of MPAS-ocean simulations," *Parallel Computing*, vol. 55, pp. 43–48, 2016.

[96] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *CLADE - Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments 2008, CLADE'08*, 2008, pp. 15–24.

[97] U. Ayachit, A. Bauer, E. P. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. E. Jansen, B. Loring, Z. Lukic, S. Menon *et al.*, "Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 921–932.

[98] S. Lakshminarasimhan, X. Zou, D. A. Boyuka, S. V. Pendse, J. Jenkins, V. Vishwanath, M. E. Papka, S. Klasky, and N. F. Samatova, "DIRAQ: scalable in situ data- and resource-aware indexing for optimized query performance," *Cluster Computing*, vol. 17, no. 4, pp. 1101–1119, 2014.

[99] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *HPDC 2015 - Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 61–72.

[100] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling embedded in-situ indexing with deltaFS," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*. IEEE, 2019, pp. 30–44.

[101] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki, "NoDB: Efficient query execution on raw data files," *Communications of the ACM*, vol. 58, no. 12, pp. 112–121, 2015.

[102] L. Weng, G. Agrawal, U. Catalyurek, T. Kurc, S. Narayanan, and J. Saltz, "An approach for automatic data virtualization," in *IEEE International Symposium on High Performance Distributed Computing, Proceedings*. IEEE, 2004, pp. 24–33.

[103] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 385–396.

[104] H. Xing, S. Floratos, S. Blanas, S. Byna, M. Prabhat, K. Wu, and P. Brown, "Arraybridge: interweaving declarative array processing in scidb with imperative hdf5-based programs," in *Proceedings - IEEE 34th International Conference on Data Engineering, ICDE 2018*, 2018, pp. 977–988.

[105] C. Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 355–366.

[106] ——, "An Efficient Bitmap Encoding Scheme for Selection Queries," in *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 28, no. 2. ACM, 1999, pp. 215–226.

[107] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An Experimental Study of Bitmap Compression vs. Inverted List Compression," in *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, 2017, pp. 993–1008.

[108] H. Xing and G. Agrawal, "COMPASS: compact array storage with value index," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management - SSDBM '18*. New York, New York, USA: ACM Press, 2018, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3221269.3223033

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

The paper reports three groups of experiments, including (1) transferring performance experiments; (2) query performance experiments; and (3) parallel performance experiments. We ran all the experiments on our in-situ query system MoHA with two benchmarks, LULESH and PIC, on AWS cloud, with `p3.2xlarge` instances with NVIDIA V100 16 GiB SXM2 GPU, with a 4.8 TiB AWS FSx Lustre parallel file system as the storage. The system runs Ubuntu Linux LTS 18.04, with a kernel version of 4.15. The system runs an Ubuntu LTS 18.04 operating system with a Linux kernel of version 4.15 and Lustre client 2.10.8. MoHA is compiled using a version of 8.4.0 of GCC compiler and the CUDA 10.2 platform. The highest level of optimization (`-O3`) is used for all experiments. All the benchmarks are included in the provided artifact.

All experiments are run at least 5 iterations and the average results are reported. System cache is cleared before all I/O-related experiments. The order of experiments is randomized to amortize underlying system performance fluctuation. The reported processing time excludes simulation initialization time and the time of result materialization to highlight the performance of actual in-situ processing.

## ARTIFACT AVAILABILITY

*Software Artifact Availability:* All author-created software artifacts are maintained in a public repository under an OSI-approved license.

*Hardware Artifact Availability:* There are no author-created hardware artifacts.

*Data Artifact Availability:* All author-created data artifacts are maintained in a public repository under an OSI-approved license.

*Proprietary Artifacts:* None of the associated artifacts, author-created or otherwise, are proprietary.

*Author-Created or Modified Artifacts:*

```
Persistent ID: 10.5281/zenodo.3866623
Artifact name: MoHA Artifact
```

## BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

*Relevant hardware details:* AWS p3.2xlarge instances; AWS FSx for Lustre Parallel file systems.

*Operating systems and versions:* Ubuntu 18.04 running Linux kernel 4.15

*Compilers and versions:* g++ 8.4.0

*Applications and versions:* MoHA

*Libraries and versions:* fmt 5.3.0 catch2 2.7.1 roaring 2019-03-05-2 boost 1.72.0 / 1.69.0 * MPICH 3.3.2

*Key algorithms:* bitmap generation

*Input datasets and versions:* LULESH 2.0.2-dev, PIC

*URL to output from scripts that gathers execution environment information.*

```
https://bitbucket.org/randomnames/workspace/snippets⌋
↪  /aLjrB9
```