GPU Adaptive In-situ Parallel Analytics (GAP)

Anonymous Author(s)

ABSTRACT

Despite the popularity of in-situ analytics in scientific computing, there is only limited work to date on in-situ analytics for simulations running on GPUs. Notably, two challenges that stay unaddressed are 1) performing memory-efficient in-situ analysis on accelerators and 2) for a given query and platform, automatically choosing the processing resources and suitable data representation. This paper addresses both problems. First, we make several new contributions towards making bitmap indices suitable, effective, and efficient as a compressed data summary structure for the GPUs – this includes introducing a layout structure, a method for generating multi-attribute bitmaps, and novel techniques for bitmap-based processing of major operators that comprise complex data analytics. Second, this paper presents a performance modeling methodology, which aims to predict the placement (i.e. CPU or GPU) and the data representation choice (summarization or original) that yield the best performance on a given configuration. Our extensive evaluation with complex in-situ queries and real-world simulations shows that with our methods, analytics on GPU using bitmaps almost always outperforms other options, and the GAP performance model predicts the optimal placement and data representation for most scenarios.

ACM Reference Format:
Anonymous Author(s). 2022. GPU Adaptive In-situ Parallel Analytics (GAP). In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, analytics on simulation output is increasingly performed *in-situ*, *i.e.*, without writing original data to the disk [1–7]. In-situ analytics, replacing *post hoc* analysis, is going to be even more important in the future because of the changing ratio between computing power and I/O capabilities. It is anticipated that Exascale-era machines will only have 10^{12} bytes/second I/O capability, 10^6 times lower than their floating-point computation capability. This ratio between floating point computation and I/O capabilities has gotten worse by a factor of 200 since the first Petaflop machine [8, 9].

Another relatively recent but continuing trend in HPC has been increasing heterogeneity as accelerators dominate available computing resources on most clusters and supercomputers. Unfortunately, to date, there is very limited work on in-situ analytics when accelerators are involved [10–13], and many problems remain open in this area. For example, a common method of performing in situ data analysis on GPU-based simulations is moving data that is generated to the CPU and analyzing it there. However, with more powerful GPUs being packed into denser nodes, the cores and memory available for CPU-based in-situ analytics is increasingly a bottleneck. An attractive, but almost completely unexplored, option is GPU-based in-situ analytics, as processing the data on the GPU as it is being generated can alleviate the data transfer costs while utilizing the high data-parallelism and memory bandwidth of GPUs. Enabling this, however, requires addressing two challenges. First, GPU memory is limited, and with a running simulation already consuming almost all of it, very limited memory is available for in-situ analytics. Second, the GPU computation model is limited, and query execution requires support from the CPU and workarounds to perform complex synchronization and might not always be efficient.

This paper presents GAP (GPU Adaptive In-situ Parallel Analytics), a system for efficient in-situ GPU approximate processing. GAP builds on the previous work on using *data reduction* techniques, such as bitmaps, for in-situ analytics. However, unlike the previous work that used these techniques to ease the I/O bottleneck on the path to the CPU [14, 15], in GPU Adaptive In-situ Parallel Analytics, data reduction techniques are used for easing the memory pressure for GPU-based analytics. GAP addresses several challenges in this process. First, GAP proposes a novel GPU-friendly bitmap summarization format, which packs multi-attribute simulation output compactly to save limited device memory. This structure also has the benefit of enabling highly-parallel query processing on the GPU. Next, we demonstrate efficient implementation of basic operations on bitmaps that could be stored in compressed or original format. Finally, building query execution using these operations, GAP develops techniques for maximizing GPU utilization, in view of the limited synchronization support currently available.

Second, GAP also addresses the aforementioned challenge that GPU-based analytics (with data reduction techniques) might not be the most efficient option in all cases. Though storing all data in GPU memory directly is likely to be infeasible because of memory constraints, it is possible to transfer the original data to the CPU and perform the entire analysis there. Alternatively, one can generate an approximate bitmap representation on the GPU, and either perform the analytics directly on the GPU, or transfer the data and using CPU cores for analysis. With the goals of understanding how in-situ query performance is affected by various factors, and make automated decisions on the placement of data and computations at the runtime, GAP provides a cost model and uses it to guide the choice of best query processing strategy.

To summarize, this paper makes the following contributions:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- We introduce a compact multi-attribute bitmap format suitable for GPU-based in-situ processing that can be generated and analyzed efficiently in parallel.
- We present novel designs addressing complex and general analytics operations that compare simulation output spatially and/or across time-steps, and execute the query processing on GPUs, while utilizing bitmaps as a summary structure.
- We investigate different strategies for in-situ heterogeneous analytics and perform detailed comparisons on the memory and compute cost of different summarization and data placement options when hardware configuration changes.
- We develop a performance model and an adaptive approach to place the workload to achieve faster computation and better resource utilization.

Our extensive evaluation indicates that the GPU-based bitmap approximate processing is highly effective, outperforming CPU-based processing in most scenarios. To guide better adaptive in-situ heterogeneous processing, this paper also considers the best in-situ processing strategy in various scenarios with changing hardware capacities, and performs detailed analysis on why a particular option outperforms or lags behind. Finally, our cost model accurately predicts the execution time of of various components of query processing pipeline, thus providing the basis for choosing a suitable processing strategy.

2 BACKGROUND

2.1 Heterogeneous (GPU) Platforms

Several important aspects of an accelerator (GPU) inform the design of heterogeneous in-situ analytics systems. As a background, a GPU consists of dozens of vectorized Streaming Multiprocessors (SMs) or Compute Units (CU) with multiple SIMD lanes. In terms of I/O, while there are other techniques in development, most GPU devices today are still connected to the host system via PCI-E bus, requiring explicit data movement between host and GPU memory and data is moved between host and GPU memory explicitly. Although rapidly increasing, the size of the on-device memory and bandwidth of the host-to-GPU is still quite limited, especially considering the nodes are also becoming increasingly dense. In terms of computation, computing tasks on a GPU are usually described using *kernels* consisting of thousands of lightweight threads. *Thread warps*, usually 32 or 64 threads are the unit for atomic execution. These threads are divided into a number of *blocks*, or coordinated thread arrays (CTAs). A block is usually not swapped out of an SM until it finished its execution, Most GPUs until recently also do not support diverging execution inside a single wrap and have no forward progress guarantee, making inter- and intra-block synchronization hard. Most of today's accelerators also rely on relatively costly host-side APIs for device management.

2.2 Bitmap Summarization

GAP uses *Bitmaps* as a data representation scheme. Bitmaps [16–23] offer an attractive way to represent and process a set of integers. A set consisting of integers in range $[0..n)$ can be represented by a bitmap using n bits: a set bit (1) denotes the existence of the corresponding element in the set. Bitmap indices have been widely used in the context of data warehouse [16, 24–27] and lately, for scientific simulation data analytics [28–35].

As scientific computations involve floating point values with an indefinite number of distinct elements, a *binning strategy* [33, 36] is often employed to divide the value domain. In such cases, bitmaps can also be seen as a *summarization* of the original data, and used to process a variety of queries *approximately* [31, 33, 35, 37]. Because storing the bitmap in its uncompressed form (bitvector) can be inefficient spatially, many algorithms [17–21, 23], have been proposed to improve space efficiency of bitmaps, achieving a *compressed representation*. Run-length encoding (RLE) [17, 18] is an example of one such representation. However, such representations are inherently serial, and hard to process in the massively parallel architectures of GPUs.

Addressing this problem, Xing *et al.* [15] proposed a new *segmented* bitmap representation similar to Roaring bitmaps [19, 21]. Such a representation divides each bitmap into chunks of 2^{16} elements, and saves individual chunks in one of the three container formats: *Uncompressed*, which saves the chunk as uncompressed bitvectors as described earlier, *Array* or *compressed*, which saves the bitmap as an array that stores the indices of the set bits, and *Full*, which marks a chunk with all bits set to one. A chunk is saved with the representation that minimizes its storage. Such a representation offers parallelism at both the chunk and intra-chunk levels.

3 GAP QUERY PROCESSING

Though bitmap generation has been performed on GPUs [38, 39] and many sophisticated query operators on bitmaps have been implemented on CPUs [33, 35, 37], supporting bitmap-based analytics on GPUs remains an open challenge due to the limitations of the GPU architecture. In this paper, this issue is addressed through the design of our layout (Section 3.1) as well through methods for conducting basic operations efficiently on this layout (Section 3.2). The key insight here is that contrary to traditional sequential nature of compressed bitmaps, GAP bitmaps can be easily analyzed via a number of composable efficient primitives on the massive parallel architecture of the GPU.

3.1 Memory-Efficient GPU Bitmap Representation

The first step of approximate query processing on the GPU is designing a compact bitmap representation optimized for GPU processing. Building on the same roaring-like bitmap representation from MoHA [15], we create an optimized representation and a method for generating it efficiently. The key underlying motivation is that our target queries likely require processing of the bitmap indices of multiple attributes together. Our representation has the following design innovations: 1) a single buffer for all the attributes in the same time-step, 2) improvement of data locality for multi-attribute queries by placing all the metadata of the chunks in the same (or a set of aligned) arrays close to each other, 3) alignment of chunks at

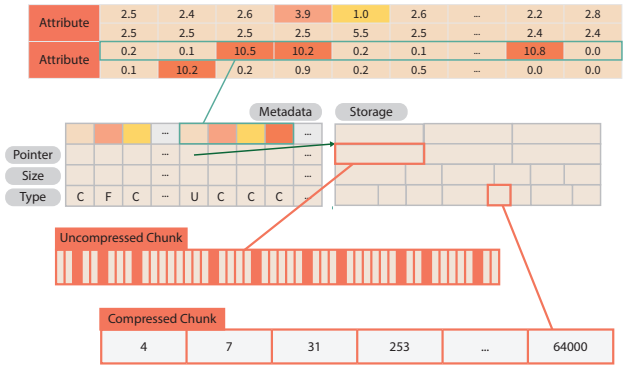


Figure 1: Data layout of the GAP optimized bitmap representation: each attribute is stored in multiple chunks indexed by a metadata table, C, F, and U refer to compressed, full, and uncompressed chunks, respectively.

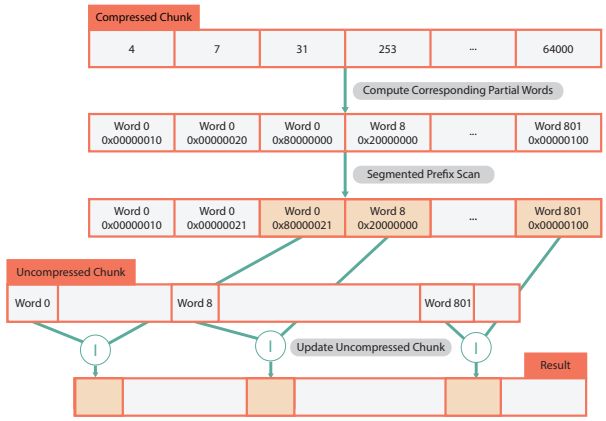


Figure 2: Performing bitwise operations between an uncompressed and a compressed chunk on GPU. A compressed chunk is converted to a series of 'partial words', the generated words are then merged with the uncompressed chunk through a prefix scan to avoid race condition.

the word boundaries for better GPU query processing efficiency, by reordering all the chunks, packing the compressed chunks at the end of the buffer, and ensuring all the uncompressed chunks are aligned at 64-bit word boundaries. Figure 1 gives an illustration of such a layout. This design also reduces memory allocation and data migration overheads, because there are fewer buffers to be managed.

Next, instead of generating the bitmap indices attribute-by-attribute as may seem more natural, GAP generates the bitmap indices for different attributes together. This has the advantage of maximizing GPU utilization by reducing unnecessary kernel workload, as well as minimizing memory-related costs such as allocating and resizing buffers. The algorithm implemented works in two passes: the first pass determines the cardinality values of each chunk by generating a histogram. Because the bitmap size is determined by its cardinality values; GAP can then allocate the corresponding buffer for the bitmap. Such allocation also takes into consideration of the alignment of each chunk to maximize data parallelism by putting all uncompressed chunks in the front of the buffer. After the memory is allocated, the second pass generates the actual chunk data using another kernel, with each thread block handling one chunk.

3.2 GPU-based Analytics: Basics

This subsection describes how a small set of basic operations are implemented on compressed and uncompressed bitmaps. The most basic operations (for supporting our representative queries in Section 4 or otherwise) are the three element-wise Boolean algebra operations: *intersection* (bitwise `AND`), *union* (bitwise `OR`), and *symmetric difference* (bitwise `XOR`), performed between two attributes of the same or aligned arrays. Because each GAP bitmap is stored as a series of chunks and these are element-wise operations, the implementation of these operations can be reduced to how these operations are performed between individual chunks. However, a complication of implementing these operations arises from the need to handle the aforementioned compressed chunks, which store all set bits as an array of 2-byte integers, directly without decompressing [17–19, 21, 34]. Thus, the implementation of element-wise operations depends on how these two chunks are stored – compressed or not. The three possibilities are discussed below.

Uncompressed v. Uncompressed: Performing the bitwise operations on two uncompressed bitvectors word-by-word produces an uncompressed output chunk. Such an operation can be simply implemented on the GPU using its vector instructions. It is possible that the result chunk can be stored in a more efficient storage format, but such conversion, if desirable, can be performed later.

Uncompressed v. Compressed: In such cases, the type of the result chunk depends on the specific operation, and thus we consider each case separately. An *intersection* between uncompressed and compressed chunks always yields a compressed buffer and the operation can be performed in two steps. First, for all elements on the compressed chunk, in parallel we determine if the element exists in the result chunk by checking the corresponding bit in the uncompressed chunk. Then, for all the elements that have an intersection, a prefix scan is performed to determine the final position of the element in the compressed buffer, and the element is saved at the calculated location.

In comparison, the *union* between an uncompressed chunk and a compressed chunk is always an uncompressed buffer. On the CPU, this operation is as simple as just setting all the corresponding bit of the uncompressed chunk to *true*. However, such an operation on the GPU architecture creates race conditions since multiple threads can access the same word in an uncompressed chunk at the same time. An efficient block-wide atomic instruction, if supported, can be used when setting the corresponding bit. In lieu of such an instruction, a *segmented prefix scan* can be carried out to merge

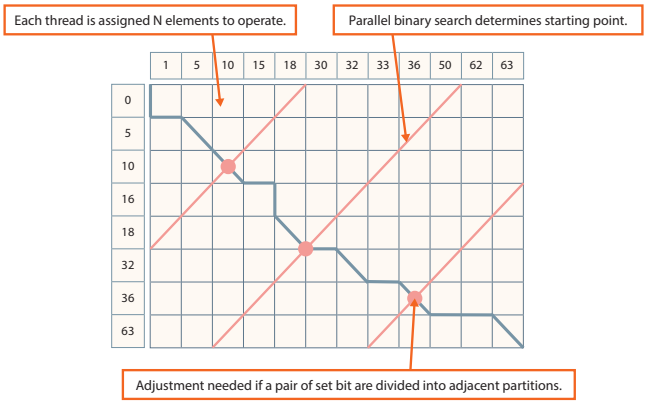


Figure 3: Performing bitwise operations between two compressed chunks on GPU. First the chunks are divided to smaller partitions that contains an equal number of N elements, then each GPU thread can process such a partition individually.

access to the same word, and only one thread saves the final result to the corresponding words. Figure 2 gives an illustration of the process.

The *symmetric difference* of an uncompressed and a compressed chunk can be either type of chunks. GAP always generates an uncompressed chunk as a first step, if desirable, the sparse results can be compressed later. The uncompressed result can be computed by traversing the compressed side and flipping the corresponding uncompressed bits. Traversing the compressed side in parallel also creates a race condition, which can be addressed by either utilizing the atomic instructions or performing a prefix scan to eliminate conflicts, just as mentioned above.

Compressed v. Compressed: Intersections between two compressed chunks always produce a compressed chunk, whereas the union and symmetric difference operations can produce either chunk type depending on the cardinality of the results. Operations between two compressed chunks are essentially the intersect, union, or symmetric difference between the two sorted lists. The operations between two compressed chunks are performed in parallel using a divide and conquer method. At a high level, the algorithm for this case tries to divide the workload equally n GPU threads, so that the first GPU thread produces the first $1/n$ of output and so on. Each GPU thread can then performs the operation (intersection, union, or symmetric difference) on an equal number of elements (VT) independently. More specifically, we consider processing two compressed chunks $left, right$. First, the two compressed chunks are loaded to the shared memory for faster access. The two chunks are then partitioned into a number of sublists $left_{s_l,e_l}$ and $right_{s_r,e_r}$, with a total cardinality of VT elements. Each thread can compute its corresponding partition utilizing a binary search in parallel. Suppose the thread i is responsible for merging the sub-lists $left[s_l \dots e_l]$ and $right[s_r \dots e_r]$. Since we know that $s_l + s_r = (i-1) \times VT$, therefore, a thread can run a binary search to determine the value of s_l, and perform the corresponding operations on the partitions. Figure 3 illustrates this process.

There are two important details to note while performing the intersection or the merge operation. First, the last element of one partition might happen to be equal to the first element of the next, in which case the serial phase of the algorithm will need to perform an additional check before proceeding. Second, the output position of each serial operation cannot be determined beforehand, so a prefix scan needs to be run to determine the output position.

3.3 Enabling Parallel Analytics on a GPU

In addition to the need for efficient parallelization using vectorized execution units, processing queries on the GPU adds other complexities, particularly because the GPU devices often only support a limited number of scheduling and synchronization primitives. GAP utilizes a number of techniques to perform queries on the accelerator efficiently.

The first question is scheduling granularity. Most primitives (which can also support implementations of reduce, scan, and others) described in the last subsection can be implemented at a warp, block, or the grid level. This raises the question of how should the bitmap computations be allocated to individual multiprocessors. Specifically, should more blocks coordinate to perform a single bitwise operation between two bitmaps, or should each block be assigned an entire bitmap index to work with. Such decisions are dictated by two conflicting factors; a finer granularity of allocating work improves the utilization of parallelism, but most GPUs today lacks synchronization mechanisms beyond a single block.

To solve the dichotomy, GAP determines the allocation granularity based on the access pattern. If the access to the bitmap chunks is regular and predictable, GAP assigns each block a single chunk to work with, in order to utilize parallelism as much as possible. The result is usually saved into a global buffer, and a separate kernel is used to reduce or summarize results of individual chunk operations. These kernels can be submitted asynchronously to the same device queue in advance to minimize latency. If the access is irregular and unpredictable, such as in searching algorithms, all bitmap operations are usually assigned to the same multiprocessor. As GPU devices have few widely available synchronization mechanisms across different blocks, assigning work using finer granularity would imply frequent kernel launches, which are costly.

Another problem in such cases is the need to coordinate the workload between different multiprocessors. While there have been recent developments towards grid-level barriers like CUDA cooperate groups [40], in practice most GPUs cannot synchronize between blocks without synchronizing the entire grid. GAP utilizes a central queue to allocate work to different multiprocessors and a simple busy-waiting semaphore is used to arbitrate access. As GPUs lack widely available forward process guarantee inside the same warp, only one thread in a block is used to negotiate with other blocks. Another useful data structure is a *lock-free hash table*, for tasks such as removing duplicate status across different threads. GAP

implements an open-addressing cuckoo hash table [41, 42] using GPU atomic primitives.

4 QUERY IMPLEMENTATION EXAMPLES

We now demonstrate how the basic capabilities developed here can support challenging in-situ analytics queries. As a motivation for selecting these queries, we note that in-situ analytics aims to shorten the feedback loop of *post-hoc* analysis in scenarios such as *Smart Simulations* [43] and *Urgent HPC* [44], among others. In *Smart Simulations*, timesteps' outputs are continuously compared against data from actual experiments or other models; and in *Urgent HPC*, the computation requires quick assimilation of the output, as one is responding to natural events such as wildfires or others. These scenarios motivate several representative queries – we summarize these queries and their implementation using our system here.

Time-step selection is a classic technique to reduce the sheer size of simulation output by only keeping most distinct time-steps. A typical strategy [45] partitions the time-steps into individual intervals, and for each partition, chooses the one time-step least similar to the selected time-step from the previous partition. The algorithm works by comparing all time-steps in the current window with this baseline by computing *correlation metrics* between the time-step pairs.

The metrics used in this work are a variation of the *Earth Mover's Distance* (EMD), which can be seen as the minimal cost of changing the distribution of one time-step to another. Such a distance can be easily compute using bitmap summarization: An XOR bitwise operation computes the difference between the corresponding buckets of the baseline and the current time-step. Once the number of different elements between the corresponding buckets is known, it can be post-processed to produce the final metric [37]. This can be easily parallelized using the GPU in two steps: In the first step, a kernel is scheduled, each block of which processes the EMD distance of each corresponding segment of the input attributes. Atomic instructions are then used to accumulate the score of different chunks. After that, a simple reduction kernel returns the best time-step, and the chosen time-step is sent for storage or further processing.

Stable-region search looks for regions that are stable, *i.e.* not changing substantially between each pair of adjacent time-steps over several time-steps. This query is inspired by physical and geological research for stable region [46] and the search for large-regions with specific features in scientific computations [34].

We process this using a dynamic programming approach: we keep a temporary bitmap for the candidate stable points for each potential bucket b. At each time-step, we check if the candidates have changed more than the specified threshold. This can be achieved by performing a bitwise `OR` operation between the adjacent buckets of b to gather the candidates in the desired range of the incoming time-step, and then performing a bitwise `AND` operation to find the overlap. The result is again saved to the buffer mentioned above. The stable points in all the buckets are then solidified into a single buffer, and a search is performed to look for large contiguous regions. The analysis follows a similar parallelization strategy as the previous query: In the first step, each block is assigned to process one bucket. A second kernel is then used to search for stable regions.

Sliding Window Contrast Set Mining (SWCSM) [35, 47] searches for a set of filter predicates that maximally separate the output of the running simulation from a baseline. Given a target attribute, the search seeks a set of predicates that together represent the desired subset of the output space – selected, as indicated above, with the goal of maximizing the difference between the target attribute values. Typically a *quality function* is used to compare different candidate subsets. The selected subset highlights the difference between the two simulations, and helps determine the direction of the future iterations. This paper considers searching the contrast sets no larger than k time-steps.

For each k-sized window of the slides, SWCSM searches for a pair of the filtering condition that can potentially produce the largest difference between the two, which is defined by a *quality function*. In this work, we use a quality function of $q = (1 + |s_l - s_r|) \times (1 + |m_l - m_r|)$, where s_l, s_r and m_l, m_r are the *support* (size) and *mean* of the chosen target value in the two subsets.

A pruned BFS search on GPU similar to what has previously been carried out on CPUs [35] is performed to search such conditions. For each filter set, the support and mean can be easily estimated by bitmap operations. As previously discussed, a centralized queue is used to allocate the subset conditions to search. Once a block is assigned a new filter condition through the queue, it accesses the bitmap to estimate the population and mean, generates new possible conditions, and pushes the subset conditions back into the queue.

Accuracy Considerations: Finally, we also comment on the issue of potential accuracy loss from the use of bitmap based process. It turns out that an advantage of bitmap-based approximate processing is the accuracy loss is often minimal. Taking the three queries here as examples, both the EMD computation in time-step selection [37] and the searching of SWCSM filter conditions [35] process the values atg the granularity of histogram bins, so bitmap processing is able to produce the same results if the same binning strategies are used. For the stable region search, the comparison of adjacent time-steps can have some error due to binning granularity, but suitable binning strategies minimize the loss [33].

5 MODELING QUERY PROCESSING

As discussed in the introduction, generating bitmaps on the GPU and performing analysis using them is only one of the options. It's also possible analyzing the GPU-generated bitmaps on CPU or analyzing the original datasets on the CPU. The trade-offs between these options depend on the hardware configurations and specific queries and the properties of the dataset. This section discusses how GAP models the performance of in-situ analytics of various aforementioned strategies.

GAP builds a cost model for a specific platform to guide the choice of query plans. Each in-situ query is executed either every time-step or after a fixed number of time-steps. To simplify the

training and prediction, we assume that the same amount of data is output and analyzed every time an in-situ query is executed. The cost of an in-situ query comprises several factors: the GPU-CPU *data transfer time* (T_{cpy}), the *bitmap generation* time (T_{bitmap}), the *CPU computation time* (T_{cpu}), the *GPU computation time* (T_{gpu}), and the *disk I/O time* (T_{disk}). GAP estimates the cost of a particular query with a given option by combining the estimated cost of all applicable components. Next, we discuss how each components are estimated.

Bitmap Generation Time: The bitmap generation time cost can be divided into two parts: the time of reading the original data and the time of converting the original data to bitmap format, the former being proportional to the input data size, and the latter being proportional to the output bitmap size. Thus, the bitmap generation cost can be modeled as a linear combination of the two costs: $T_{bitmap} = k_{b1} \cdot S_{input} + k_{b2} \cdot S_{bitmap}$. Here, S_{input} is the size of the data that is converted to bitmap representation and S_{bitmap} is the size of (compressed) bitmaps that are output.

Modeling Computation Time: Modeling the performance of arbitrary computations on CPU and GPU is a very hard problem. Our approach here relies on the fact that our queries comprise invocations of relative simple operations, as previously described in Section 4. Thus, the query processing time can be estimated operator by operator. As general guidance, we find that many operators have relatively low computation costs and are often bound by the memory access time. The performance of these operators can be estimated by evaluating the memory access pattern of each operator. Such estimation on original datasets mainly involves estimating the number of scans and the data amount each scan visits. For the bitmap operators, this cost is estimated using the number of bitmap operations, and the average input and output size of each operation, which can be deduced from the data distribution.

Data Transfer Time: The output data size (either original or bitmap) can be measured directly on a specific simulation. Because the cost of transferring the data is strongly related to the size, it is trivially modeled as the sum of two costs: a fixed transfer initializing cost and a variant cost linear to the size of the data being transferred $T_{cpy} = T_{seek} + K_{cpy} \cdot S_{cpy}$. Here, S_{cpy} is size of the data to be transferred. The disk I/O cost T_{disk} can be similarly modeled.

To illustrate further, take the time-step selection query as an example – at each step, each bitmap bucket of the time-step is compared with the corresponding bucket of the selected time-step. Hence, the total number of bitmap operation performed is $n = N_a \times N_c \times B$, with N_a, N_c and B being the number of attributes, the number of chunks in a bitmap and the number of bins in the bitmap index, respectively.

Applying the Model: Given the above cost prediction methodology, GAP perform an offline prediction in two steps. First, to train the model for a specific hardware configuration, a driver program is executed on a synthetic dataset. This driver contains a number of scenarios, including only copying the original and bitmap data to the host, performing a number of simple scan on the original dataset, and a number of bitwise operations on generated bitmaps to measure the performance of bitmap operations. Using the driver run, linear regression can be applied to determine parameters such as T_{seek}, K_{cpy}, k_{b1}, and k_{b2}. The second step is specific to the simulation program to be executed. This program is executed for a small number of time-steps to measure its simulation time, which determines whether overlapping CPU processing is profitable; and the data distribution of the simulation output. which can be used to estimate the effectiveness of the bitmap summarization. After this data is collected, the prediction is made by inputting to the model the measured input simulation running time, bitmap size estimated using the output distribution, the problem size, and the number of iterations. As the prediction is offline, the overhead of the prediction is the one time cost of measuring the operator costs on a particular configuration, and the execution of the few iterations of the target simulation to measure its output time and size.

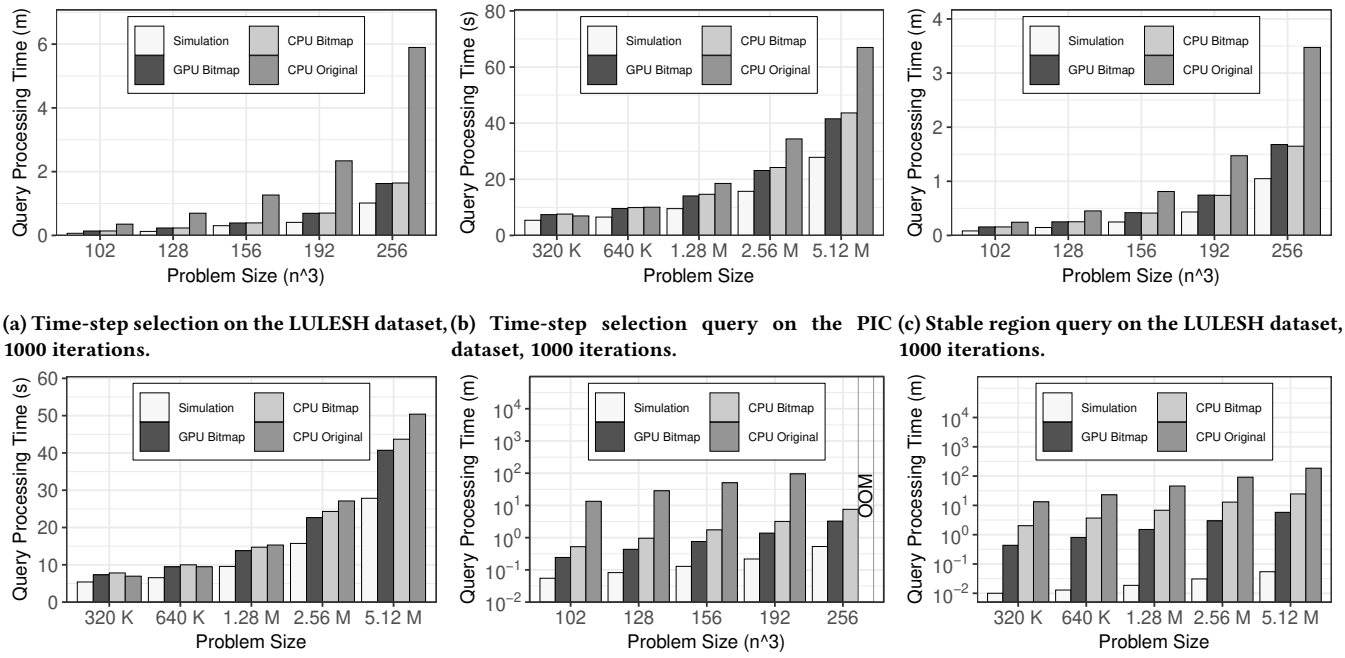
6 EXPERIMENTAL EVALUATION

This section empirically evaluates the performance of the bitmap-based query processing and the prediction model in GAP using multiple queries and actual simulation outputs. The following issues are investigated: 1) After the innovations introduced in our work, can bitmap-based GPU analytics provide an advantage over other in-situ query processing options in terms of query processing times? (§ 6.2), 2) How do different architectural configurations impact the relative performance of different query processing options? (§ 6.3) 3) How well does the cost model chooses optimal query execution option for the given query and architectural configuration? (§ 6.4) Because of limited space, impact of individual optimizations introduced for bitmap-based GPU analytics is not individually evaluated.

6.1 Experiment Setup

Configuration. The experiments in the paper are run on cloud HPC platforms, which are increasingly popular for scientific computation [48] – specifically, an Amazon Web Service (AWS) node with a 4-core Intel Xeon 2586 v4 CPU and an NVIDIA V100 SXM2 GPU with 16 GB of memory. The CPU and GPU are connected using a PCI-E connection with a measured host-accelerator throughput of $\sim 11GB/s$. In experiments involving Disk I/O, a remote 4.8 TB AWS Lustre Filesystem with a 940 MB/s baseline throughput and a 6x burst throughput is used. The GAP system is programmed using C++ and CUDA. The highest optimization level is used to compile the binaries. All experiments utilize 8 working threads to ensure proper CPU utilization.

Datasets and Methodology. Two simulation programs are used for the evaluation. The first one is LULESH [49, 50], a hydro-dynamic-inspired simulation core widely used in previous research [33, 51, 52]. The code models a simple *Sedov blast* problem using a mesh-based method, and retains the computation and data movement pattern in the original simulation. The second simulation, PIC [53], is a one-dimensional electrostatic particle-in-cell simulation program modeling the movement of ions and electrons in the electronic fields. 64 equal-width bins based on the value domain range are



(a) Time-step selection on the LULESH dataset, 1000 iterations. **(b) Time-step selection query on the PIC dataset, 1000 iterations.** **(c) Stable region query on the LULESH dataset, 1000 iterations.**

(d) Stable region query on the PIC dataset, 1000 iterations. **(e) SWCSM query on the LULESH dataset, 100 iterations.** **(f) SWCSM query on the PIC dataset, 100 iterations.**

Figure 4: Comparison between different methods on LULESH and PIC datasets.

used for all the experiments. A 5-time-step window size is used for window-based operations.

All experiments are run at least five times and the average timings are reported. All cache is cleared before I/O-related experiments. In order to mitigate the performance variation caused by underlying platforms, all the experiments are run in a randomized order. The reported processing time include the cost of simulation, data summarization, data movement, and query execution. It should be noted that GAP overlaps CPU and GPU transferring time, and thus the total reported times are not a simple sum of these components.

6.2 Overall GPU bitmap analytics performance

To examine how our various optimizations and innovations help improve overall performance, the GPU bitmap analytics incorporating these are compared with two other options (the current state-of-the-art): CPU Bitmap and CPU Original. We use the three in-situ queries discussed in this paper to explore the answer. Figures 4a and 4b report the query processing time of the time-step selection query while using different processing options for the two datasets. Performing such a query in original format requires considerable memory. For example, with a problem size of 256^3, LULESH generates 1.5 GB of data per time-step, thus requiring 10 GiB of memory to store six time-steps for processing. In comparison, processing the bitmaps on the GPU is ~ 3.6× and ~ 1.6× faster than processing the original data using the CPU on the largest LULESH and PIC datasets. Excluding the simulation running time, GPU-based

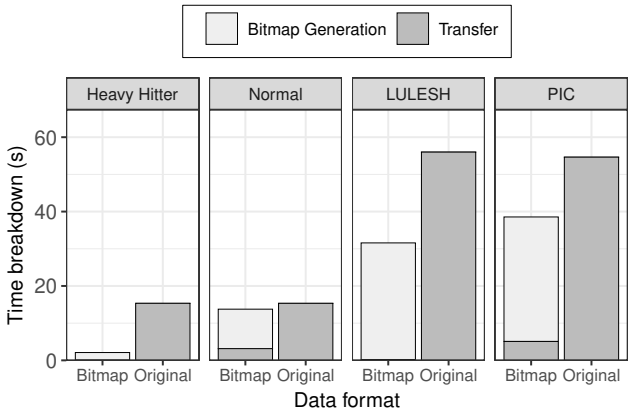


Figure 5: Comparison of the bitmap generation and data movement cost on different datasets, 1000 iterations.

bitmap processing accelerates the computation 8× and 2.9× on the datasets, highlighting the advantage of our novel implementation.

While the cost of creating the bitmap indices on the GPU is not inconsequential, it is outweighed by the savings in both transferring the data and processing. It should be noted that both bitmap generation and processing has been acceleration through our design decisions. For example, on average, generating and transferring bitmap is more than 2.7× faster compared with transferring the original data, while the bitmaps are processed more than 10× faster

than the original data on both CPU and GPU. The relative performance of processing the bitmap summarization on the GPU and CPU depends on the datasets. The two methods almost perform identically on LULESH, while processing GPU bitmaps is slightly faster on PIC. This demonstrates the impact of input distribution on bitmap summarization – LULESH has a more concentrated distribution, whereas PIC has a distribution similar to a normal distribution; as a result, the bitmap size of LULESH is relatively smaller. Therefore, while processing the time-step selection on the GPU saves both transfer and computation, its effect is less pronounced in the LULESH dataset.

Similarly, Figures 4c and 4d report the query processing times of the stable region query with different options. The results paint a similar yet subtly different picture. While processing the data using bitmap is still advantageous compared with processing the original data – on the largest dataset it being 2.1× and ∼ 1.2× faster – the time differences are less significant compared with the time-step selection query, especially on the PIC dataset. Unlike comparing the earth mover distance, checking if two cells are close to each other is actually less expensive on the original dataset compared with bitmaps. Hence, the speedup of processing bitmaps becomes less significant. Again, this effect is more substantial on the PIC dataset, because its larger bitmap indices size leads to less savings in data transfer. The relative performance of GPU and CPU bitmap processing in the stable region query is similar to the time-step selection query, hinting that GPU bitmap processing is more efficient in this case even with less overlapping opportunities compared with CPU-based processing.

The final part of this subsection looks at an irregular and compute-intensive query - the sliding-window contrast set mining (SWCSM) query. Figures 4e and 4f show the comparison between different methods as the problem size is varied. It can be seen that processing bitmaps are orders of magnitude faster compared with processing the original data. On the average, GPU-based bitmap processing is ∼ 64× faster on the LULESH dataset, and ∼ 30× faster on the PIC datasets compared with processing the original data. GPU-based bitmap processing also shows significant advantages over processing on the CPU, being ∼ 2.3× and ∼ 4.5× faster on average, respectively. This demonstrates that by utilizing the higher parallelism and memory bandwidth through our design, the bitmaps can be processed very efficiently on the GPU. This use case also demonstrates the necessity of data reduction in in-situ analytics - with the problem size of 256^3, the GPU used here does not even have enough memory to hold all the data generated one time-step, making analysis without summarization impossible.

As the end-to-end query costs include (optional) data summarization, movement, and processing, it is interesting to see how the overhead of generating and move the bitmap summarization compare with moving the original data. Our next experiment isolates these costs by comparing cost of generating bitmap and moving the data to the host with just moving the original dataset on different datasets, without performing any analytics. Four typical datasets are used for comparison in this experiment to demonstrate the significant impact of data distribution on bitmap generation cost. In addition to the two real-world datasets, two synthetic datasets are also used. The synthetic datasets include one with a normal distribution of values and another with a "heavy hitter" distribution with one single value with over 95% frequency dominating the distribution. All the datasets have a time-step size of 648 MiB.

Figure 5 shows the results. Overall, the cost of moving the original dataset is always higher than the cost of generating and moving the bitmap. In other words, despite the significant cost of generating bitmaps, the reduction in data transfer times always justifies this cost. Data distribution affects the acceleration: in the two more concentrated datasets, Heavy Hitter and LULESH, generating bitmap is 7.3× and 1.8× cheaper compared with moving the original data, respectively. The bitmap generation is 1.1× and 1.4× faster compared with moving the original datasets on the two more distributed datasets, Normal and PIC. Moving the bitmap to the GPU is not a huge overhead because the bitmap is usually compressed very efficiently, on average, copying the bitmap to the host is ∼ 9.45% more expensive compared with keeping the bitmap on the GPU. In actual anlaytics, keeping the data on GPU also benefits from the additional computational resources on the GPU.

6.3 Impact of Different Hardware Configurations

A recurring theme of the previous subsection is that the relative performance of the available query processing options depends on the actual dataset and the query type. It is thus natural to ask whether changing hardware capabilities can result in different relative performance of different options. . This subsection considers three typical scenarios in in-situ analytics: when the CPU resources are limited, when the accelerator available is less powerful, and when the CPU-GPU connection becomes slower due to contention or with a different configuration or topology.

The first case considered here is when the CPU resources available for in-situ analytics on the simulation output from one GPU are varied – in practice, this can easily happen due to concurrent workload on the system or denser accelerator configuration. We choose the stable region and the SWCSM query as representatives of I/O- and compute-intensive queries, respectively, and the problem sizes are fixed to 192 and 1.28M for the LULESH and PIC datasets. respectively.

Figures 6 and 7 report the results as the number of CPU threads are varied from 1 to 8. As seen in the figures, the impact of allocating less CPU resources to the computation depends on both computational pattern and the dataset distribution. The performance of the stable region query does not change much with the different number of threads in most cases, demonstrating the query is mainly bottlenecked by the transfer time between the CPU and GPU. However, with only one CPU worker, the CPU-based bitmap processing time of the same query on PIC becomes slower than processing the original data, while the query processing time on the LULESH dataset stays stable. This is because searching for similar tuples between time-steps in bitmaps is a relatively expensive operation, requiring comparing each bitmap bin to multiple adjacent bins.

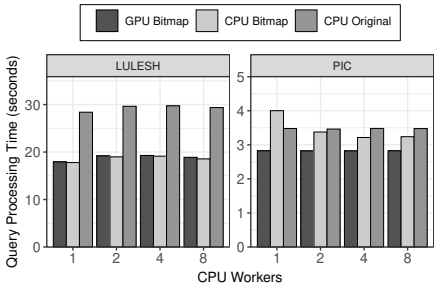


Figure 6: Performance of stable region query with varying number of CPU workers (200 iterations)

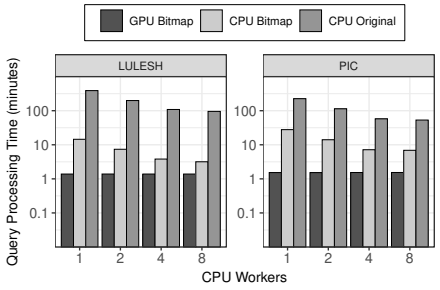


Figure 7: Performance of SWCSM query, varying number of CPU workers (200 iterations)

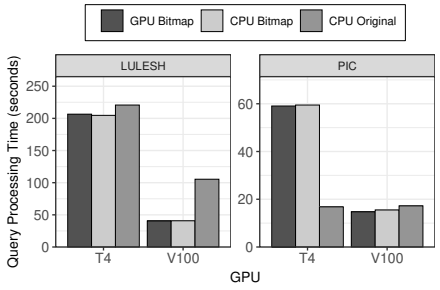


Figure 8: Performance of time-step selection query on platforms with different GPUs

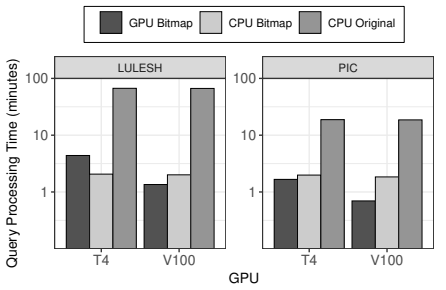


Figure 9: Performance comparison of sliding CSM query on platforms with different GPUs

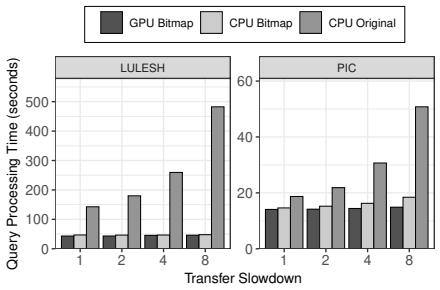


Figure 10: Performance of time-step selection query as Host-Accelerator transfer speed varies

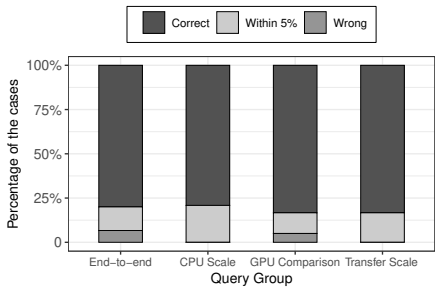


Figure 11: Accuracy of the performance model across experiment groups in previous subsections

The same operation is very cheap on the original data as a simple subtraction and filtering would suffice. On the contrary, both CPU processing methods on the SWCSM query scales with the number of CPU workers up to 4 cores, suggesting that the processing is limited not only by the GPU-CPU transfer and the disk I/O, but also by the computation cost of searching.

The next scenario considered is how changing computational capacibilities of GPU change the overall performance of the queries. Specifically, we compare the query performance on two platforms with identical CPUs and different GPUs on Google Cloud Platform (GCP). Both platforms have a 6-core 2.30 GHz Intel Xeon Skylake processor, but one has an NVIDIA Tesla V100 GPU, and the other has an NVIDIA Tesla T4 GPU, with about only half of the TFLOPs of the V100 GPU. The problem size is still fixed to 192 (LULESH) and 1.28 M (PIC). Figure 8 reports the relative query processing time of the time-step selection query on both platforms. As expected, reducing the GPU power makes processing data in bitmap less attractive. On LULESH, processing the data in bitmaps is only about $\sim 1.3\times$ faster on T4 compared with processing the original dataset, and more than twice slower on PIC. This is because on a slower GPU, bitmap generation is more expensive, reducing the gain from compressing the data to bitmap summarization; in addition, the simulation also generates data with a lower rate, making the host-to-GPU transfer and computation less of a bottleneck. Another observation in this group of experiments is the sparsity of the bitmap summarization can have large implications for relative performance - even on a slower GPU, processing a more concentrated dataset such as LULESH using the bitmap summarization can still be the relatively faster choice. Contrarily, the efficiency of processing larger bitmaps is more affected by the capabilities of the GPU. Figure 9 reviews the SWCSM query performance on the two platforms, where the computational cost is more significant. While approximate bitmaps are still the faster choices, bitmap query processing on GPU becomes less attractive with the slower GPU. On LULESH, GPU-based bitmap processing is slower compared with CPU-based processing, whereas the relative performance advantage of GPU-based bitmap processing is more significant on PIC, validating the aforementioned observation – GPU-based bitmap processing is more effective on denser bitmaps due to the higher data parallelism.

The final scenario considered is when the Host-Accelerator transfer bandwidth changes. Specifically, the host-accelerator transfer speed in the time-step selection query is artificially slowed down from $1\times$ to $8\times$, and other parameters are kept the same as in previous experiments in this subsection. Figure 10 reports the results. As expected, both CPU-based methods are affected by the slowdown, with the methods that transferring more data being more severely impacted. A natural conclusion is, compression is more profitable on systems with Host-Accelerator links that are either slower or

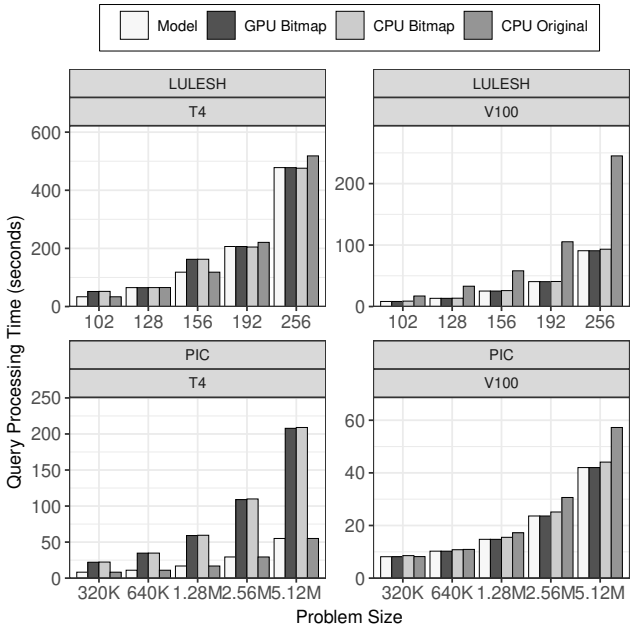


Figure 12: Performance of the modelled GAP system – GPU comparison experiments with time-step selection query.

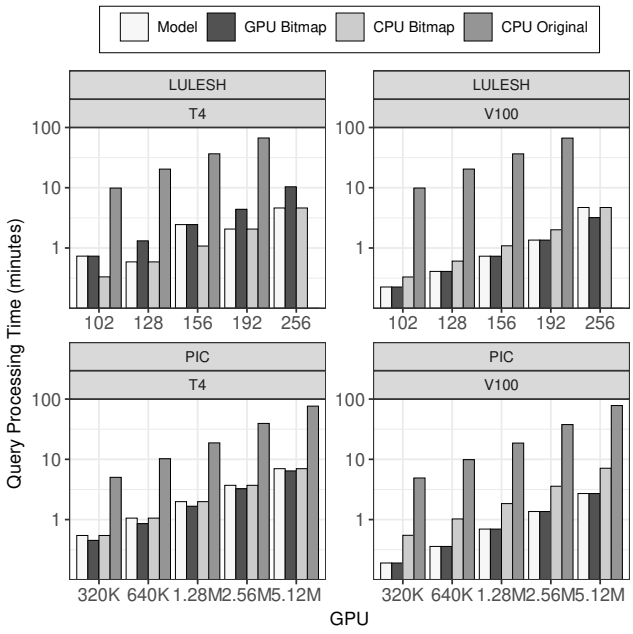


Figure 13: Performance of the modelled GAP system – GPU comparison experiments with SWCSM query[1]

have more contention. Denser accelerator setups can easily create such a contention.

6.4 Accuracy of Prediction Model

This final subsection looks at the accuracy of the cost model. Our model is trained using four synthetic datasets with different distributions: normal, log normal, a "heavy hitter" distribution with one single value occurs in a frequency of over 95%, and an "invariant" distribution that contains only a single value. The training data size is varied from 64MB to 1 GB per time-step.

Figure 11 evaluates how accurate our model performs in identifying the optimal placement/representation options using the four groups of experiments in the previous sections: end-of-end comparison, scaling CPU workers, changing GPUs, and scaling the throughput of host-to-gpu transfer speed. Depending on whether a model identifies the optimal choice, the result are categorized into three types; when the optimal option is identifies ("correct"), when the configuration selected is non-optimal but has performance within 5% of the optimal choice ("Within 5%"), and when the model chooses a non-optimal choice with more than 5% performance different ("Wrong"). As seen in the chart, the model identifies a overwhelming majority of the placement/representation model correctly in each group of the queries. If we account for the cases that the case the model picks being nearly fast as the optimal choice – less than 5% performance differences, over 90% time the model picks the optimal or a near optimal choice.

One significant advantage of employing a cost model is the system can predict and choose a good placement and representation strategy according to hardware configurations and other parameters such as input distribution and problem size. To illustrate how the model predicts and chooses a good placement/representation strategy, Figures 12 and 13 show how the model selection version compares with the other strategies with the time-step selection query using different GPUs, datasets, and problem sizes. The optimal strategies for these queries are not straightforward, and the two queries represent both I/O- and computation-intensive workloads. As seen in Figure 12, the model correctly picks the original representation in smaller LULESH cases and PIC cases on T4, while choosing the GPU Bitmap options in other situations. An alternative like simply choosing bitmap processing on GPU always will yield considerably worse performance. Similarly, Figure 13 shows how the model correctly identifies CPU Bitmap as a more preferable strategies on T4 GPUs and V100 GPUs with the LULESH dataset, while choosing GPU Bitmap on the PIC/V100 use cases. While there is a slightly higher number of more sub-optimal cases with the CSM query, this does shows GAP handles computation-intensive workloads effectively.

7 RELATED WORK

In-situ Visualization and Analytics The idea of in-situ processing emerged decades ago [54–56]. One focus area has been on effective visualization tools [4, 12, 57] and methods [58–61]. Middleware systems such as ADIOS [6, 62, 63] and GLEAN [7] have been proposed for easier integration.

Other promiment efforts include [3, 14, 64–69] – Bauer [70] provides a comprehensive survey on the topic. Data reduction methods,

using either in-situ indexing [15, 37, 71, 72] or mathematical methods [14, 67, 68], have been proposed.

There has been limited research on GPU-based in-situ analytics. Thompson *et al.* [11] proposes on-the-fly data summarization and visualization on heterogeneous platforms for decision-making. Landrush [10] proposes an approach that utilizes idle cycles on the GPU to run analytics kernel for better time-to-answer. MoHA [15] proposes using bitmap summarization as an generic approximate representation for complex bitmap processing. MONA [73] proposes a performance monitoring system suitable for increasingly complex in-situ analytics pipelines.

Bitmap indices and compression. Originally developed in relational database context [16, 24, 25], bitmap indices such as FastBit [27, 34] gained popularity in scientific computing. Run-length-encoding is a popular way of compress bitmaps [17, 18, 74]. Roaring bitmaps [19, 21] is a particularly interesting recent compressed bitmap representation because of its speed in query processing [22]. There are also recent efforts to increase the accuracy or reduce the size of the bitmap representations [23, 75, 76]. Both lossless and lossy floating-point compression methods, such as FPC [77], SZ [78] and general compressors such as the Lempel-Ziv compressor family [79], are also used for reducing data movement costs. Recent efforts extend these frameworks to GPU [80, 81]. While these methods are also effective in reducing the data size, bitmap indices can be directly queried on, using fast and simple bit operations available in the hardware.

GPU Query Processing and Performance Modeling Early works in the relational database field include implementations of individual offloaded GPU-based operators and their performance models [82–84]. Full-fledged systems, both in transactional and analytical context, have been proposed [85–87]. Other relevant work has been on implementation of primitives such as sort [82, 88–90] and scan [91, 92] and merge [93, 94]. Yuan et al [86] estimate the performance of individual operator based on device memory access time, and a recent approach [95] divides heterogeneous execution into smaller execution components to improve cardinality estimation.

8 CONCLUSION

This paper has introduced GAP, a solution for adaptive in-situ heterogeneous analytics. We have proposed a GPU-based query processing method, which builds on a multi-attribute bitmap indices format to perform complex analytics on the GPU in a memory-constrained and functionality-limited environment. Our paper has also introduced a performance modeling methodology, capable of guiding the user to choose between resources and data representations for query processing. Our extensive evaluation shows that GPU-based bitmap processing has performance advantages in most cases, accelerating the queries by up to an order of magnitude. We also show how different setups impact the performance of different processing strategies, and how our model can predict the optimal strategies quite accurately.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] H. Xing, G. Agrawal, and R. Ramnath, "Moha: a composable system for efficient in-situ analytics on heterogeneous hpc systems," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 1155–1170.

[14] 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.

[15] H. Xing, G. Agrawal, and R. Ramnath, "Moha: a composable system for efficient in-situ analytics on heterogeneous hpc systems," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 1155–1170.

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

[17] 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.

[18] 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.

[19] 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.

[20] S. Kim, J. Lee, S. R. Satti, and B. Moon, "Sbh: Super byte-aligned hybrid bitmap compression," *Information Systems*, vol. 62, pp. 155–168, 2016.

[21] 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.

[22] 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. [Online]. Available: http://db.ucsd.edu/wp-content/uploads/2017/03/sidm338-wangA.pdfhttp://db.ucsd.edu/wp-content/uploads/2017/03/sidm338-wangA.pdf{%}0Ahttp://dl.acm.org/citation.cfm?doid=3035918.3064007

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

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

[25] ——, "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.

[26] K. Wu, E. Otoo, and A. Shoshani, "On the performance of bitmap indices for high cardinality attributes," in *VLDB '04*, 2004.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Y. Su, G. Agrawal, and J. Woodring, "Indexing and parallel query processing support for visualizing climate datasets," in *ICPP '12*, 2012.

[33] 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

[34] 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

[35] G. Zhu, Y. Wang, and G. Agrawal, "SciCSM: novel contrast set mining over scientific datasets using bitmap indices," *... of the 27th International Conference on ...*, pp. 1–6, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2791347.2791361http://dl.acm.org/citation.cfm?id=2791361

[36] K.-L. Wu and P. S. Yu, "Range-based bitmap indexing for high cardinality attributes with skew," in *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE, 1998, pp. 61–66.

[37] 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.

[38] 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

[39] 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, 2010, pp. 315–329. [Online]. Available: http://link.springer.com/10.1007/978-3-642-15251-1{_}26

[40] N. Corp., "Cuda c programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups

[41] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[42] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Building an efficient hash table on the gpu," in *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 39–53.

[43] 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."

[44] K. Yoshimoto, D. Choi, R. Moore, A. Majumdar, and E. Hocks, "Implementations of urgent computing on production hpc systems," *Procedia Computer Science*, vol. 9, pp. 1687–1693, 2012.

[45] C. Wang, H. Yu, and K.-L. Ma, "Importance-driven time-varying data visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1547–1554, 2008.

[46] G. Aad, B. Abbott, J. Abdallah, R. Aben, M. Abolins, O. AbouZeid, H. Abramowicz, H. Abreu, R. Abreu, Y. Abulaiti *et al.*, "Search for magnetic monopoles and stable particles with high electric charges in 8 tev p p collisions with the atlas detector," *Physical Review D*, vol. 93, no. 5, p. 052009, 2016.

[47] 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.

[48] 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.

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

[50] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, aug 2013. [Online]. Available: http://codesign.llnl.gov/lulesh

[51] 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.

[52] 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.

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

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

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

[56] 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.

[57] 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.

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] S. Maroulis, N. Bikakis, G. Papastefanatos, P. Vassiliadis, and Y. Vassiliou, "Rawvis: A system for efficient in-situ visual analytics." SIGMOD, 2021.

[70] 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.

[71] 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.

[72] 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.

[73] M. Wolf, J. Choi, G. Eisenhauer, S. Ethier, K. Huck, S. Klasky, J. Logan, A. Malony, C. Wood, J. Dominski *et al.*, "Scalable performance awareness for in situ scientific applications," in *2019 15th International Conference on eScience (eScience)*. IEEE, 2019, pp. 266–276.

[74] 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.

[75] 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

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

[77] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2008.

[78] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 2016, pp. 730–739.

[79] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, 1977.

[80] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao *et al.*, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 3–15.

[81] Lawrence Livermore National Laboratory, "cuzfp," 202. [Online]. Available: https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp

[82] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 325–336.

[83] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 511–524.

[84] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, pp. 1–39, 2009.

[85] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 709–720, 2013.

[86] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 817–828, 2013.

[87] S. M. A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. C. Anadiotis, and A. Ailamaki, "Gpu-accelerated data management under the test of time," in *Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*, no. CONF, 2020.

[88] N. Leischner, V. Osipov, and P. Sanders, "Gpu sample sort," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–10.

[89] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 351–362.

[90] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 417–432.

[91] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," 2007.

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

[93] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 331–340.

[94] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[95] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 733–744, 2017.