Accelerating Viz Pipelines Using Near-Data Computing: An Early Experience

Qing Zheng[†], Brian Atkinson[†], Daoce Wang[‡], Jason Lee[†], John Patchett[†], Dominic Manno[†], Gary Grider[†]
[†]Los Alamos National Laboratory, Los Alamos, NM, USA
[‡]Indiana University, Bloomington, IN, USA
{qzheng, batkinson, jasonlee, patchett, dmanno, ggrider}@lanl.gov
{daocwang}@iu.edu

***Abstract*—Traditional scientific visualization pipelines transfer entire data arrays from storage to client nodes for processing into displayable graphics objects. However, this full data transfer is often unnecessary, as many visualization filters operate on only small subsets of data in a data array. With the rise of computational storage, smart NICs, and smart devices enabling offloaded processing, this paper examines a case where a visualization pipeline is divided into pre-filters that run near data and post-filters that execute on the client side. Pre-filters preprocess the data near it on storage nodes, reducing data volumes before transfer based on downstream pipeline needs, while post-filters complete the processing on the client node. Experiments done on two real-world simulation datasets demonstrate that this approach can significantly reduce network transfer volumes, cutting visualization pipeline data load times by up to 2.8× compared to traditional methods, and up to 11.9× when combined with data compression techniques.**

I. Introduction

Scientific datasets stored in popular file formats such as VTK, HDF5, and NetCDF often contain multiple data arrays, each representing a specific data attribute such as temperature, velocity, or material density. Traditional scientific visualization pipelines, such as those powered by the Visualization Toolkit (VTK) [1], require reading the entirety of those arrays from storage before filtering and processing them on client nodes, where they are transformed into graphical representations for interactive analysis. However, as data sizes continue to grow, reading back those data arrays from storage to client nodes has become increasingly time-consuming, leading to longer overall pipeline runtimes and delayed insights. This significantly hinders the efficiency of modern scientific discovery.

Existing scientific visualization software stack provides two ways to mitigate the cost of large data transfers: data array selection and data compression. Data array selection allows a visualization pipeline to specify a subset of data arrays to read rather than always reading every data array in a dataset. This avoids transferring irrelevant data arrays and thereby reduces overall data transfer cost. Data compression decreases data transfer sizes by compressing the data before storage and decompressing it upon reading. Commonly used data compression methods include GZip [2] and LZ4 [3], both of which are general-purpose, lossless compression algorithms.

While transferring only necessary data arrays and applying compression can effectively reduce data transfer volumes, each selected array is still transmitted logically in its entirety: this remains inefficient. Real-world visualization pipelines often consist of filters that target only a small subset of data within an array, making full-array transfers unnecessary. For example, contour filters — which are widely seen in visualization pipelines — focus only on data near the contour values rather than the entire array. This, in practice, can easily translate to several orders of magnitude fewer data values compared to the original array.

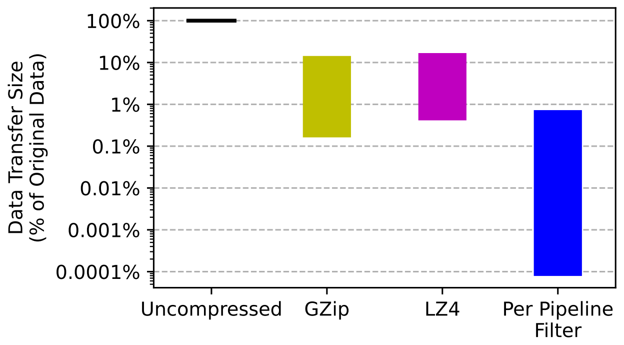


Fig. 1: Comparison of data reduction ratios achieved across various technologies.

As an example, Fig. 1 compares the final data transfer size when applying data compression on a real-world scientific dataset to that of transferring only the necessary data values required by a contour filter. For each case, a range of data transfer sizes is shown, reflecting the results achieved over a series of simulation timesteps and contour values. While we provide more details on this example dataset and the contours later in this paper, the results show that transferring only the values needed by a downstream pipeline filter — a contour filter in this case — has the potential of achieving a far more significant data reduction rate than that of data compression. In our example, data compression reduced data transfer sizes by 1 to 2 orders of magnitude, whereas pipeline-filter-based data filtering achieved an up to 7 orders of magnitude reduction in data transfer sizes.

As concepts such as computational storage [4, 5], smart NICs [6], and smart devices [7] mature, this paper explores a case where the data selection logic of a VTK visualization pipeline is offloaded to storage for executing early before costly data transfer occurs that moves data from storage to

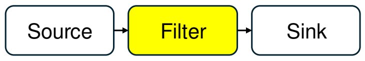


Fig. 2: Overview of a VTK visualization pipeline.

client nodes. To enable such offloads, we propose dividing a pipeline filter — such as a contour filter — into a pre-filter that runs near data on storage nodes and a post-filter that runs on the client side. Pre-filters preprocess the data on the storage side, reducing data movement by sending only necessary information to the client nodes, where post-filters complete the processing.

We developed a prototype of a modified VTK contour filter, incorporating a pre-filter component for data reduction that runs on the storage side and a post-filter component for contour generation that runs on the client side.

We use two real-world scientific datasets to study the performance of offloaded data retrieval and filtering in VTK-based visualization pipelines: one from xRage [8], a versatile multi-physics simulation code, and the other from Nyx [9], a cosmological simulation application. Preliminary results show that our approach significantly lowers network traffic, decreasing visualization pipeline data load time by up to 2.8× compared to traditional methods (i.e., reading entire data arrays) and up to 11.9× when combined with data compression techniques.

This early-experience paper offers 3 key contributions: **1**) a novel approach to accelerating scientific visualization pipelines using offloaded computation, **2**) a prototype implementation based on real-world VTK software code, and **3**) an evaluation using two real-world scientific datasets that shows promising results.

The rest of this paper is organized as follows: Sec. II provides background on VTK visualization pipelines and contouring. Sec. III introduces our primary example dataset. Sec. IV evaluates data compression techniques as the current state-of-the-art. Sec. V presents our approach. Sec. VI reports experimental results using our first example dataset. Sec. VII introduces our second dataset along with its experimental results. Sec. VIII discusses related work, and Sec. IX concludes the paper.

II. Background

A typical simulation science workflow consists of a data generation phase and a data analysis phase. During the data generation phase, simulation applications are run, writing timestep data to storage. This is then followed by the data analysis phase, where visualization pipelines are executed to perform data analytics.

Commonly used scientific visualization software includes ParaView [10, 11] developed by Kitware and VisIt [12] developed by LLNL, both are built on the VTK visualization toolkit. In this section, we provide an overview of VTK visualization pipelines, their limitations, and explore a specific type of VTK

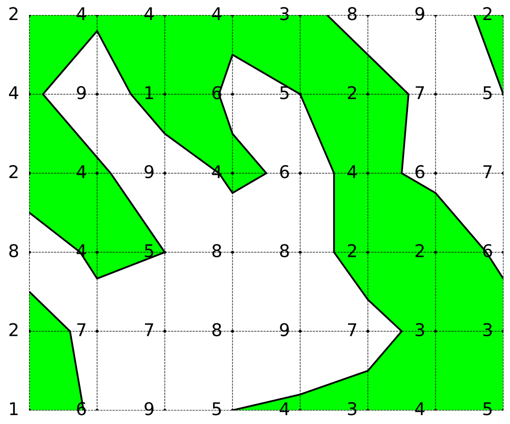


Fig. 3: A simple 2D contour example.

filters — contour filters — that are widely used in real-world visualization applications.

A. VTK Visualization Pipelines

A VTK-based visualization pipeline consists of three main components: sources, filters, and sinks, as Fig. 2 illustrates. Sources are the starting points of the pipeline. They generate or read data from various formats — such as VTK, HDF5, and many others — and introduce it into the pipeline. Filters are intermediate processing units that transform, manipulate, or analyze the data. They take input data from sources or other filters and produce output data that can be further processed. Finally, sinks are the endpoints of the pipeline where the processed data is rendered or written out. In practice, sinks often include renderers that convert data into visual representations, allowing for interactive exploration and analysis.

Real-world VTK pipelines are often invoked indirectly by software such as ParaView and VisIt. In such cases, a VTK pipeline is run either in standalone mode on a single node or in a client-server configuration across multiple nodes. In the client-server setup, the server runs on one or more nodes, handling data processing and potentially parallel rendering, while the client connects to the server, sending commands and receiving rendered images or data.

Traditional VTK pipelines depend heavily on filesystems — either local or parallel — for reading and writing data. These filesystems offer very limited data querying and filtering capabilities. Consequently, to analyze data, entire arrays often have to be read into client memory, even when only a small portion is needed for the visualization task at hand. This results in excessive data movement, a challenge that our approach tries to address through offloaded processing, as Sec. V further discusses.

B. Contour Filters

A contour is a curve or surface that connects points of equal value within a dataset. These points are typically derived from data arrays of scalar types representing attributes such

TABLE I: Scalar Fields

Array Name	Data Type	Description
rho	float	Density in grams per cubic centimeter
prs	float	Pressure in microbars
tev	float	Temperature in electronvolt
xdt	float	X component vectors in centimeters per second
ydt	float	Y component vectors in centimeters per second
zdt	float	Z component vectors in centimeters per second
snd	float	Sound speed in centimeters per second
grd	float	AMR grid refinement level
mat	float	Material number id
v02	float	Volume fraction of water
v03	float	Volume fraction of asteroid

as temperature, pressure, or elevation. Contours help in understanding the spatial distribution of these attributes, allowing for intuitive visual interpretation of potentially complex data.

Fig. 3 shows a simple example where a contour of value 5 is drawn over a 8×6 mesh comprising 48 data points holding values randomly generated from 0 to 9. The contour intersects points that either directly have a value of 5 or are interpolated along edges where one end is above 5 and the other is below 5. We refer to such edges as “interesting edges” because they help form the contour. In contrast, the remaining edges do not contribute toward the contour and can be safely ignored without impacting the final contour result.

If a VTK pipeline is able to execute some of its code very close to data — where data transfer overhead is significantly lower — it could use this opportunity to preview the source dataset and select only the mesh points associated with at least one such interesting edge. This approach minimizes the amount of data that needs to be transferred in order to generate the contour, ensuring that only necessary information is communicated between storage and the visualization node.

In VTK, contours are implemented as filters. They take scalar data arrays — as well as mesh definitions — as input and produce lines (2D) or polygons (3D) that define the contour as output. While our randomly generated contour example above demonstrated limited potential in reducing data movement by transmitting only mesh points of interest, real-world scientific datasets typically offer greater potential for data reduction due to the scientific nature of their simulations.

III. Example Dataset

Our example dataset [13] is generated from a real-world simulation application named xRage [8]. It is a parallel multi-physics Eulerian hydrodynamics code developed by LANL that can be used to study asteroid impacts in deep ocean water. In these applications, each xRage simulation simulates an asteroid falling to earth from the upper atmosphere, descending rapidly, and striking the ocean.

A simulation runs in timesteps. Every few timesteps, the simulation pauses and writes its current state to storage as VTK data files. Each timestep comprises 11 data arrays, corresponding to 11 distinct data attributes as listed in Table I.

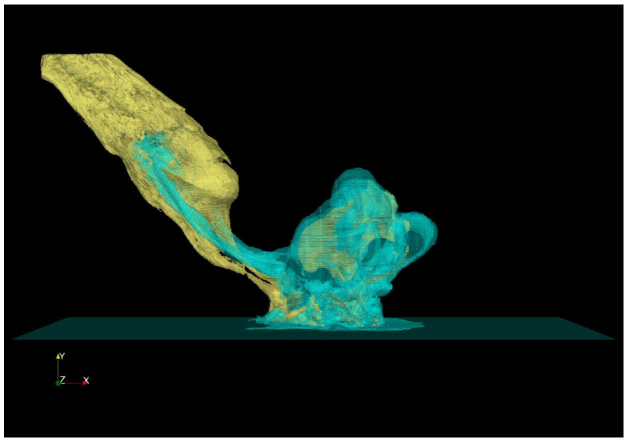


Fig. 4: Visualization of two contours over water (cyan) and asteroid (yellow) at timestep 24095 of the deep water asteroid impact dataset.

In our example, each array holds 125M values. Each such value maps to a vertex in a 500×500×500 mesh space.

Arrays of particular interest include v02 (which measures the volume fraction of water) and v03 (which measures the volume fraction of asteroid). Both arrays have a value range of [0, 1]: a value of 0 indicates no water or asteroid in a given area, while a value of 1 indicates complete coverage by water or asteroid in that area.

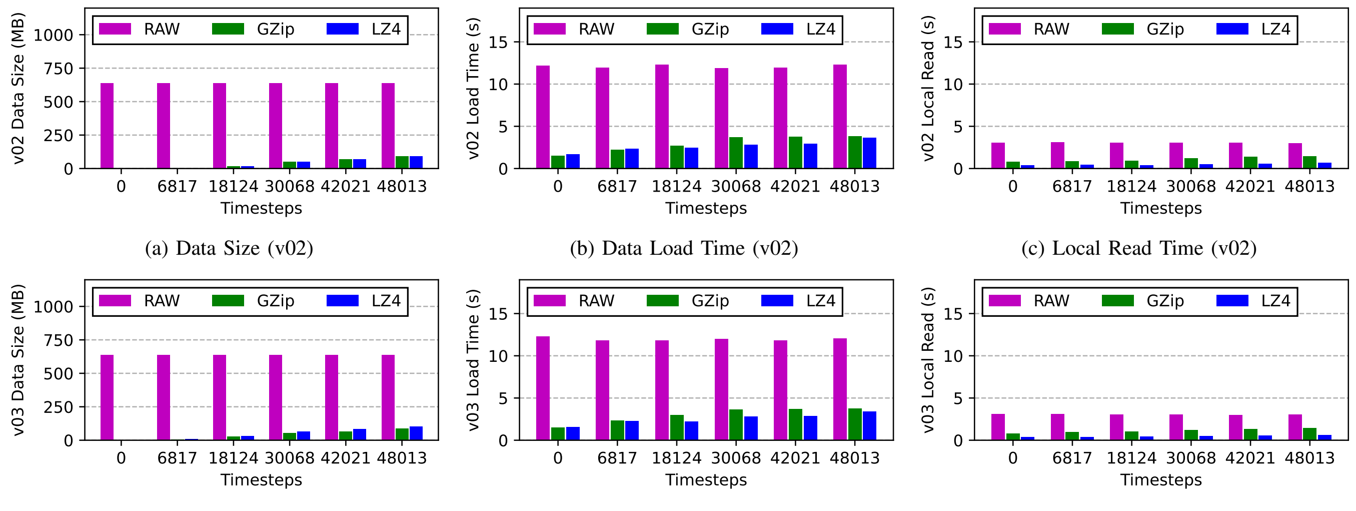
Fig. 4 demonstrates a visualization of two contours over arrays v02 (cyan) and v03 (yellow) at timestep 24095. Both contours are set at value 0.1, representing the outer surface of the water (v02) and the asteroid (v03), respectively. It depicts the aftermath of the asteroid’s impact on the ocean, including a tsunami that it triggers.

To make such contours, a VTK visualization pipeline is configured with 1) a VTK reader that acts as a source of the pipeline that reads simulation data — stored as VTK data files — from a filesystem, 2) a contour filter that takes v02 and v03 data arrays as input and produces contours at the value configured — 0.1 in our case — as output, and finally 3) a sink that runs an OpenGL subpipeline that renders the contours — defined as a set of polygons, or specifically as a set of triangles in our case — on the screen for interactive exploration and analysis.

While our example dataset contains 11 data arrays, only v02 and v03 need to be read to create the contours. To accomplish this, VTK’s data array selection can limit data transfer to just these two arrays. However, the entire v02 and v03 arrays must still be read from the filesystem during pipeline execution. To improve performance, data compression is often employed to reduce data size, decreasing data load time, thereby allowing the pipeline to run faster.

IV. Applying Data Compression

Figs. 5a and 5d compare the data sizes of arrays v02 and v03 before and after applying GZip and LZ4 compression, two widely used methods natively supported by VTK. Results show that data compression is able to effectively reduce array



(a) Data Size (v02) (b) Data Load Time (v02) (c) Local Read Time (v02)

(d) Data Size (v03) (e) Data Load Time (v03) (f) Local Read Time (v03)

Fig. 5: Evaluation of VTK's native data compression methods, GZip and LZ4, in reducing data size and load times using the deep water asteroid impact dataset. Figs. 5a, 5b, and 5c show results on the v02 data array (water). Figs. 5d, 5e, and 5f show results on v03 (asteroid).

data sizes, with GZip achieving a reduction ratio ranging from 7 to 588×, and LZ4 from 6 to 299×. The ratio is higher at the beginning of the simulation and gradually decreases over time. This trend occurs because data entropy is lower at the start of the simulation and increases as the simulation progresses, leading to reduced compression effectiveness.

Figs. 5b and 5e report the time required for a VTK pipeline to read back either the v02 or the v03 data array from a remote S3 object store to its local memory via s3fs, an open-source FUSE-based solution that enables mounting remote S3 buckets and operating them as local filesystems. We examine cases in which data is uncompressed and cases in which it is compressed using either GZip or LZ4. Our experiments were done on two nodes, one acting as the object storage server and the other as the client. The server node runs a MinIO server instance backed by a local SSD to provide the S3 service. MinIO is an open-source object storage implementation with an interface compatible with Amazon S3. The client machine mounts the S3 storage using s3fs over a 1Gb Ethernet link.

We pre-populate the object storage with our example dataset. Each data file is stored as an object in the object store, and can be opened and read as regular files through s3fs. We run a standalone VTK pipeline on the client node. In each run, the pipeline opens a timestep file, reads either the v02 or the v03 data array out of it, generates a contour, and renders it on the screen. We focus on the time it takes for the pipeline to load the data from storage to its local memory. This time includes data decompression when applicable, but excludes contour generation and rendering, which take between 0.8 to 1.3s in our experiments depending on contour complexity. All contours are set to contour at value 0.1. We repeat each run 5 times and report the average.

By reducing the size of each data array, data compression requires fewer amounts of data to be transferred, allowing each visualization pipeline to run faster. While it takes the baseline — reading data in its uncompressed form — 12 seconds to read the data, results show that applying compression can limit this time to under 4 seconds across all timesteps, leading to at least a 3× speedup in data load times for both v02 and v03. The most significant improvement is observed in timestep 0, where GZip achieves an 8× reduction in data load times while LZ4 achieves a 7× improvement. This is because the data compression ratio is highest at the beginning of the simulation, resulting in the least amount of data movement and the lowest data load time.

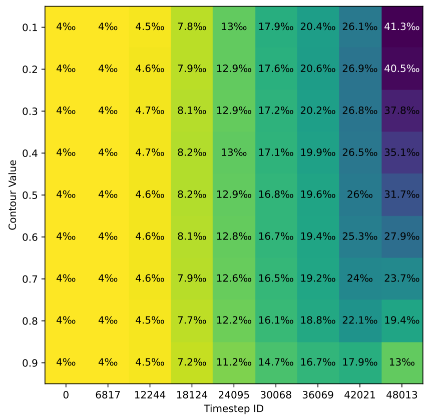
While GZip generally provides a higher data compression ratio than LZ4, it tends to perform on par with or slower than LZ4 in terms of data load times. This is because GZip's more compute-intensive compression and decompression processes lead to longer decompression times, offsetting the benefits of moving less data and resulting in equal or even longer overall data loading durations.

To illustrate this, Figs. 5c and 5f report data load times when data is stored on a local filesystem instead of a remote object store. This decreases data transfer cost, allowing us to focus more on data decompression overheads. Results show that LZ4 consistently achieves lower data load times compared to GZip in all cases, thanks to LZ4's lower decompression overhead and the reduced impact of data movement in local storage environments.

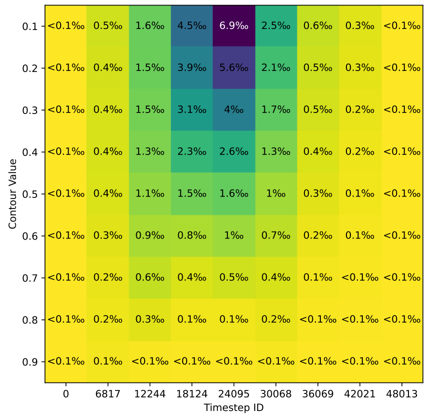
While compression can significantly reduce data transfer size and load time in visualization pipelines, there is still room for more aggressive data reduction by dynamically leveraging the selectivity of downstream filters to limit transfer to only necessary information, which leads to our approach.

V. Toward Near-Data Processing

Real-world visualization pipelines often consist of filters that target only a small subset of data within a data array, making transferring the entire array — even in its compressed form



(a) v02



(b) v03

Fig. 6: Data selection rates for contouring the v02 and v03 data arrays from the deep water asteroid impact dataset, expressed in permillage (‰).

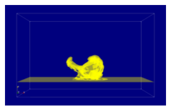
— unnecessary. When data movement limits the performance of a visualization pipeline, pre-selecting data at the storage end — which we refer to as near-data processing or NDP — can help eliminate those irrelevant subsets of data before they are sent, thereby minimizing data transfer sizes. To assess the potential of such methods, Figs. 6a and 6b report the amount of information — expressed as permillages of the original data — necessary to accurately generate contours against the v02 and v03 data arrays in our example dataset. These are the subsets of mesh points associated with at least one interesting edge as Sec. II defines. We consider contour values ranging from 0.1 to 0.9, spanning the entire value space of the two data arrays while avoiding extremes to preserve generality.

Results show that selecting data based on downstream pipeline filters — in our case, contours — can significantly reduce data movement, routinely cutting data transfer sizes by orders of magnitude, from less than 0.01% to 4%. This reduction is more pronounced in v03 (asteroid) compared to v02 (water) because the asteroid spans a smaller mesh space than that of the ocean, as shown in Figs. 7 and 8. Additionally, data reduction for v02 is more significant at the beginning of the simulation than at the end, due to the asteroid

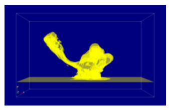
(a) t=0

(b) t=6817

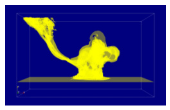
(c) t=12244



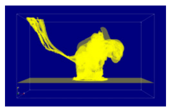
(d) t=18124



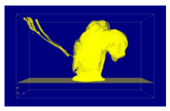
(e) t=24095



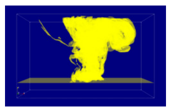
(f) t=30068



(g) t=36069



(h) t=42021



(i) t=48013

Fig. 7: A movie of contours at value 0.1 against the v02 (water) data array in the deep water asteroid impact dataset.

impacting the ocean midway through the simulation. For both v02 and v03, there is a trend of higher selectivity as contour value increases, which aligns with the scientific nature of this particular simulation.

In our example, the resulting data reduction rates tend also to frequently outperform general data compression methods such as GZip and LZ4, as we previously measured in Fig. 5. This is because data compression must still account for an entire array's information, whereas our approach tailors the data reduction to a specific visualization pipeline at runtime. This allows for more aggressive size reductions, albeit being more ad-hoc compared to data compression.

In practice, data compression and near-data processing are orthogonal to each other and can be applied together. When data is compressed, pre-selection can be done by reading the compressed data from storage, decompressing it, and then applying the usual data pre-filtering process. This combination allows compression to reduce local data read time by shrinking dataset sizes, while near-data processing ensures that only the minimum necessary data is transferred, as Fig. 9 illustrates. This leverages the strengths of both technologies.

To offload data pre-selection and filtering to storage, we envision dividing a pipeline filter into a pre-filter component and a post-filter component, as Fig. 10 shows. The pre-filter, along with the source, forms a partial VTK pipeline that runs on the storage side. Meanwhile, the post-filter, along with the sink, forms the remainder of the pipeline that runs on the client side, where reading and writing data are expected to be expensive. The two partial pipelines communicate using an RPC-based interface, with the client-side component serving as the RPC client and the storage-side component serving as the server.

The pipeline runs by the client sending pipeline information to the server. Upon receiving this information, the server's source reads the data from storage, decompresses it if necessary, and passes the result to the pre-filter. The pre-filter then scans the data in memory, identifies all necessary information to be transferred, and performs the transfer. Once the client

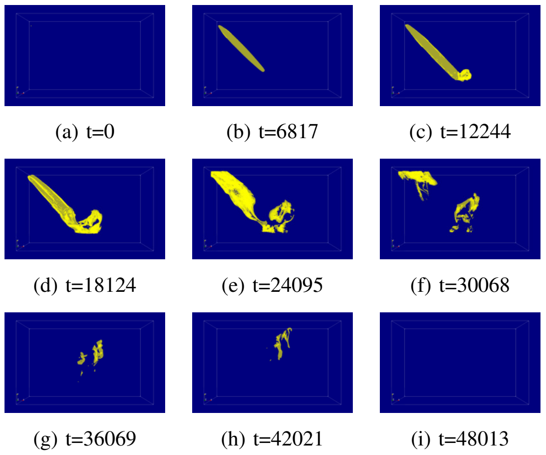


Fig. 8: A movie of contours at value 0.1 against the v03 (asteroid) data array in the deep water asteroid impact dataset.

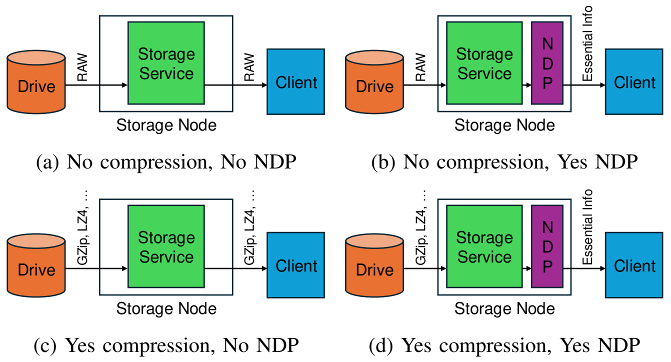


Fig. 9: Illustration of applying compression and NDP either individually or together.

receives the reply, it resumes executing the remainder of the pipeline as usual, potentially rendering the results on a screen for interactive exploration and analysis.

VI. Implementation and Evaluation

We developed a modified VTK contour filter that splits the original filter into a pre-filter and a post-filter. The pre-filter takes a full VTK data array as input and extracts a subarray that contains only the data points relevant to the contour being generated. The post-filter then takes this subarray as input and produces the final contour.

Our implementation supports generating contours at multiple contour values at the same time. Additionally, a VTK pipeline can be configured with multiple instances of our contour filter, enabling contour generation for multiple data arrays simultaneously. In this setup, each contour filter instance is dedicated to processing a specific data array, just as in the original VTK implementation. We support uniform rectilinear grids at the moment, with plans to extend support to more complex grid types in future work.

We use rpclib [14], an open-source C++ RPC library, to implement the communication between pre-filters and post-filters. It utilizes MessagePack [15], an open-source binary

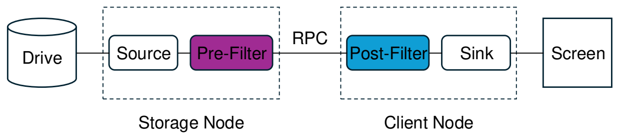


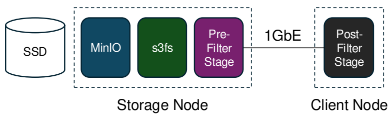
Fig. 10: Applying near-data processing to VTK-based visualization pipelines.

serialization format, to efficiently marshal and unmarshal data, alleviating interprocess-communication overhead.

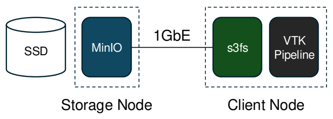
Our experiments were done using the same 2-node hardware setup as we used in Sec. IV. One node serves as the server, running a MinIO instance that provides an S3-compatible interface, while the other node functions as the client. In Sec. IV, the VTK pipeline is executed entirely on the client node, with data accessed by mounting the S3 storage as a local filesystem on the client node using s3fs. We will refer to this setup as the baseline. To test our approach, we execute the pre-filter stage of the pipeline on the storage node and the post-filter stage on the client node. Since the pre-filter stage is responsible for accessing the data, the S3 storage is mounted as a local filesystem on the storage node in this new configuration, rather than on the client node. We refer to this setup as near-data processing (NDP). Fig. 11 compares the two setups.

As in Sec. IV, we evaluate both uncompressed data and data compressed using GZip and LZ4. For baseline runs, the VTK pipeline accesses data via s3fs on the client node. In the NDP runs, the pre-filter sub-pipeline reads data through s3fs on the storage node and sends filtered data to the post-filter sub-pipeline running on the client node through RPC. In both scenarios, data is sourced through s3fs, which interfaces with MinIO to retrieve data from a local SSD on the storage node. The key difference is that in the baseline case, s3fs operates remotely from MinIO, whereas in the NDP case, s3fs runs on the same node as MinIO. This setup ensures fairness, as both the baseline and NDP runs utilize the same storage I/O software stack — s3fs, MinIO, and a local SSD — for data access, and experience the same storage I/O overhead. While the baseline incurs network traffic equivalent to the original data size, NDP allows network traffic to be significantly reduced by limiting transfer to only necessary information. This reduction in network traffic is the key distinction between the baseline and NDP runs, and is the primary focus of our experimental evaluation.

Our experiments involve generating a contour movie across a series of simulation timesteps. We use 5 contour values ranging from 0.1 to 0.9 and 9 timesteps spanning from 0 to 48,013. Each run proceeds sequentially, reading data from the first timestep, generating a contour, and then moving on to process the next timestep. Each run is dedicated to a specific contour value. We measure the time required for a pipeline to prepare data in memory for contour generation at each timestep. In baseline runs, this encompasses the time needed to read and, when necessary, decompress the data. In NDP runs,



(a) NDP



(b) Baseline

Fig. 11: Comparison of baseline and NDP experimental setups, both leveraging the same storage I/O stack for data access. In the baseline setup, network communication occurs between MinIO and s3fs, while in the NDP setup, it occurs between the pre-filter and post-filter stages of the pipeline.

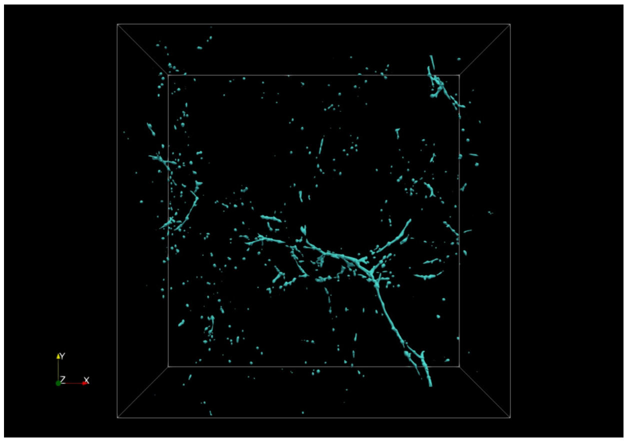


Fig. 12: A contour over baryon density in the Nyx dataset, highlighting regions of candidate halos.

the measurement includes the time taken to read, decompress, and filter the data, as well as the time required to send the filtered data to the client. We repeat each run 5 times and report the average.

Fig. 13 shows the results. There are 6 subfigures. Each displays results for a specific data type (RAW, GZip, or LZ4) and data array (v02 or v03), comparing the baseline case to the NDP. Results show that NDP consistently outperforms the baseline in all cases, achieving speedups ranging from 1.2× to 2.8×. This is because NDP reduces network traffic by pre-filtering data before transfer, allowing it to experience less network communication overhead and load data more quickly.

The largest speedups are observed in the raw data runs due to their larger base data sizes compared to compressed data, allowing NDP to attain higher data reduction ratios and greater performance improvements. LZ4 outperforms GZip in speedups, as GZip's higher data decompression overhead outweighs the benefit of its smaller base data size, resulting in longer data load times, as noted earlier in Sec. IV. Finally, v03 shows slightly higher speedups than v02, due to the greater selectivity of v03 contours, as illustrated in Figs. 6a and 6b.

The speedups for different contour values — from 0.1 to 0.9 — remain almost consistent within each subfigure. This is because the selectivity differences among contour values are negligible compared to the overall data size. The reported load time includes both network transfer and the time for MinIO to read data from its local SSD. For NDP runs, the network transfer cost is so low that the load time is dominated by the MinIO data load time, resulting in minimal variation across different contour values.

We summarize our experiments in Table II, which presents the speedups in data load times achieved through various combinations of data reduction techniques, including GZip, LZ4, and NDP. We use reading uncompressed data without any near-data processing as the baseline and calculate speedups relative to it. These results are identical to those presented in Fig. 13, but viewed from a different perspective. NDP achieves a speedup of up to 2.8× when used alone, and up to 11.9× when combined with GZip and LZ4 data compression techniques.

The speedup from using NDP alone is relatively modest in these runs because, although NDP significantly reduces network traffic, it does not accelerate local data read times (i.e., MinIO data read times), which in these runs constitute a substantial portion of the total data load time. Combining NDP with data compression reduces both network traffic and local data read times, resulting in higher overall speedups.

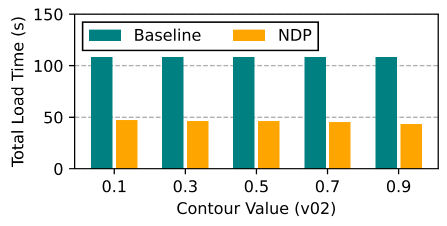
Similarly, while data compression reduces both network traffic and local data read times by minimizing base data size, it does not filter out irrelevant data subsets before transfer, resulting in higher-than-optimal network transfer volumes. Combining data compression with NDP minimizes network traffic while maintaining low local data read times, leading to the fastest data load times.

TABLE II: Speedups in data load times achieved by various combinations of data reduction techniques on the deep water asteroid impact dataset.

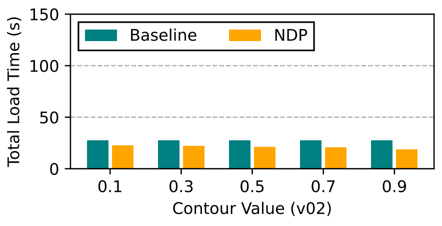
Array Data	Contour Value	Speedup					
		RAW	NDP	GZip	LZ4	GZip+NDP	LZ4+NDP
v02	0.1	1.0×	2.30×	3.96×	4.63×	4.77×	6.22×
	0.3	1.0×	2.32×	3.96×	4.63×	4.91×	6.36×
	0.5	1.0×	2.37×	3.96×	4.63×	5.08×	6.62×
	0.7	1.0×	2.40×	3.96×	4.63×	5.23×	6.92×
	0.9	1.0×	2.49×	3.96×	4.63×	5.73×	7.87×
v03	0.1	1.0×	2.70×	3.94×	4.59×	6.81×	10.74×
	0.3	1.0×	2.73×	3.94×	4.59×	6.91×	11.12×
	0.5	1.0×	2.77×	3.94×	4.59×	7.20×	11.62×
	0.7	1.0×	2.78×	3.94×	4.59×	7.28×	11.77×
	0.9	1.0×	2.80×	3.94×	4.59×	7.36×	11.87×

VII. A Second Example

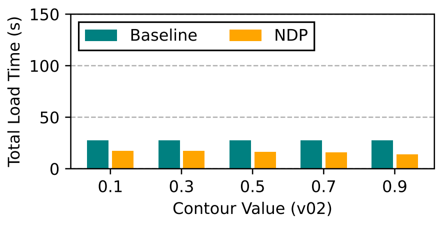
Our second dataset is part of SDRBench [16, 17], a collection of real-world scientific datasets for benchmarking lossy data compressors as well as other data reduction techniques. The dataset we use is generated by Nyx [9], a cosmological



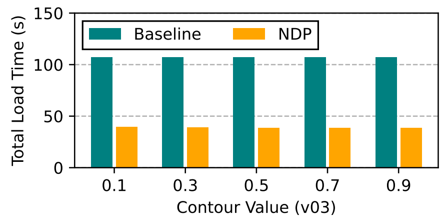
(a) RAW Data (v02)



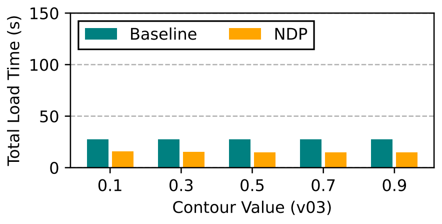
(b) GZip Data (v02)



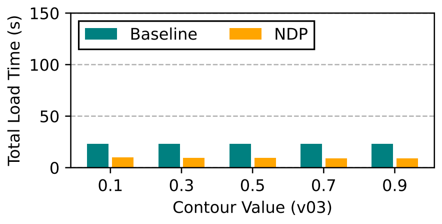
(c) LZ4 Data (v02)



(d) RAW Data (v03)



(e) GZip Data (v03)



(f) LZ4 Data (v03)

Fig. 13: Evaluation of NDP's effectiveness in reducing data load times for both compressed and uncompressed data in the deep water asteroid impact dataset. Figs. 13a, 13b, and 13c show results on the v02 data array (water). Figs. 13d, 13e, and 13f show results on v03 (asteroid).

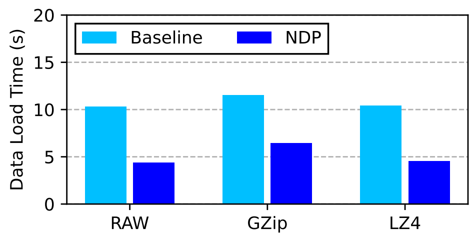


Fig. 14: Evaluation of NDP's effectiveness in improving data load times for the Nyx dataset.

hydrodynamics simulation code, and consists of a single timestep. It contains six data arrays: velocity components (x, y, z), temperature, dark matter density, and baryon density. Our focus is on the baryon density, where a value of 81.66 serves as the threshold for halo formation [18, 19]. Fig. 12 illustrates a contour based on baryon density at this threshold, with a data selectivity of 0.06%. This is the subsets of array mesh points that need to be transferred for contour generation. This number is low, which is why NDP can significantly cut network traffic, accelerating the visualization pipeline execution.

We repeat the experiment from Sec. VI using the same hardware setup, testing both uncompressed data and data compressed with GZip and LZ4. Fig. 14 compares data load times between the baseline approach (loading entire arrays) and NDP (offloaded data selection and filtering). Results show that NDP lowers data load times by 1.8× to 2.3×, thanks to a significant reduction in network traffic. However, since NDP must still read data from MinIO before applying pre-filters, its runtime is lowerbounded by local data read times, which limits the maximum speedup NDP can achieve, like in our previous experiments in Sec. VI.

Unlike the deep water asteroid impact dataset, neither GZip nor LZ4 effectively reduced the size of the Nyx dataset, resulting in data load times comparable to those of uncompressed data. GZip, in particular, worsened load times by adding significant decompression overhead while achieving only a modest 11% reduction in base data size. We anticipate that highly optimized floating-point data compressors could achieve higher compression ratios, but we leave this for future work, as 1) there is a lack of native support for these algorithms in VTK, and 2) our previous dataset has already covered cases in which compression yields significant reduction ratios. As a data reduction technique complementary to data compression, NDP can always be used in conjunction with compression — lossy or lossless — to further reduce data load times. This is particularly effective when data selectivity is high, a scenario we expect to be quite common for scientific workloads.

VIII. Related Work

Floating-point data plays a vital role in scientific simulations. As the scale of these applications expands, data compression has become essential for mitigating I/O, storage, and communication overheads. Floating-point compression techniques fall into two primary categories: lossless and lossy.

Lossless compression [2, 3, 20–24] does not introduce errors in the data, making it suitable for scenarios where information loss is intolerable. For instance, in large-scale HPC simulations, lossless compression is often employed during checkpointing to prevent error propagation. Similarly, in distributed applications where data must be communicated across nodes at every time step, lossless compression ensures that compression errors do not accumulate beyond acceptable levels [25, 26], unlike lossy compression.

On the other hand, lossy compression techniques such as SZ [27–29], ZFP [30], and MGARD [31], along with their GPU-accelerated versions [32–34], often achieve higher compression ratios by trading accuracy for user-controllable errors in the data. These lossy compressors have been widely adopted in the scientific community [35–44] due to their capability to provide significant data reduction while maintaining manageable accuracy impacts on scientific results.

In this work, we focus on the lossless compression algorithms GZip and LZ4, as they are natively supported by the VTK library. As future work, we plan to explore additional compression algorithms, including lossy compression methods.

While both data compression and NDP mitigate storage I/O bottlenecks through reducing data size or data transfer sizes, in-situ analysis techniques such as PreDatA [45], GLEAN [46, 47], NESSIE [48], DataSpaces [49], GoldRush [50], and SENSEI [51] accelerates analysis and visualization by performing these tasks during simulation, bypassing the need for data storage and avoiding I/O bottlenecks altogether.

IX. Conclusion

Analysis and visualization of large scientific datasets are key components of modern scientific discovery. While traditional data reduction techniques like compression can effectively reduce data size and movement, this paper explores an alternative approach by offloading data-intensive stages of a visualization pipeline to execute near the data. By preselecting only the necessary data for downstream operators and transferring only these subsets over the network, our approach minimizes network traffic, enabling more rapid visualization pipeline execution. Preliminary experiments on two real-world datasets demonstrate promising results, with up to a $2.8\times$ reduction in pipeline data load time for uncompressed data, and up to an $11.9\times$ reduction when combined with data compression techniques. This speedup is upperbounded by local data read times, which in our runs, involving reading data out of an object store. While testing with two real-world datasets is not exhaustive across all application domains, we expect the benefits of offloaded data selection and filtering to be reasonably prevalent and can be replicated in many other visualization applications. Our current experiments were limited to a single filter type and conducted in a traditional server-based computing environment rather than on embedded devices, setting the stage for future research. While our current focus has been on improving data load times, future work will include end-to-end performance assessments, incorporating the time required for visualization computation (e.g., contour generation) and on-screen image rendering.

Acknowledgment

We thank the anonymous reviewers of our paper for their help in improving the paper. This paper has used datasets from SDRBench: https://sdrbench.github.io. Please see the main sections of this paper for more information. This manuscript has been approved for unlimited release and has been assigned LA-UR-24-29780. This work has been authored by an employee of Triad National Security, LLC which operates Los Alamos National Lab with the U.S. Department of Energy/National Nuclear Security Administration.

References

[1] W. Schroeder, K. M. Martin, and W. E. Lorensen, *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. USA: Prentice-Hall, Inc., 1998.

[2] P. Deutsch, "Gzip file format specification version 4.3," Tech. Rep., 1996.

[3] Y. Collet, *Lz4: Extremely fast compression*, 2011. [Online]. Available: https://github.com/lz4/lz4.

[4] *Computational storage architecture and programming model v1.0*, https : / / www . snia . org / sites / default / files / technical - work / computational / release / SNIA - Computational - Storage - Architecture - and-Programming-Model-1.0.pdf, 2022.

[5] A. Barbalace and J. Do, "Computational storage: Where are we today?" In *CIDR*, 2021.

[6] *Nvidia bluefield networking platform*, https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[7] *Samsumg smartssd*, https://semiconductor.samsung.com/ssd/smart-ssd/.

[8] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, D. Ranta, and R. Stefan, "The rage radiation-hydrodynamic code," *Computational Science & Discovery*, vol. 1, no. 1, p. 015 005, 2008. DOI: 10.1088/1749-4699/1/1/015005. [Online]. Available: https://dx.doi.org/10.1088/1749-4699/1/1/015005.

[9] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. V. Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013. DOI: 10.1088/0004-637X/765/1/39. [Online]. Available: https://dx.doi.org/10.1088/0004-637X/765/1/39.

[10] J. AHRENS, B. GEVECI, and C. LAW, "Paraview: An end-user tool for large-data visualization," in *Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds., Burlington: Butterworth-Heinemann, 2005, pp. 717–731. DOI: https://doi.org/10.1016/B978-012387582-2/50038-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123875822500381.

[11] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*. Clifton Park, NY, USA: Kitware, Inc., 2015.

[12] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "Visit: An end-user tool for visualizing and analyzing very large data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, 2012, pp. 357–372. DOI: 10.1201/b12985.

[13] J. Patchett and G. Gisler, "Deep water impact ensemble data set," *Technical Report LA-UR-17-21595, Los Alamos National Laboratory*, 2017.

[14] *Rpclib*, http://rpclib.net/.

[15] *Messagepack*, https://msgpack.org/index.html.

[16] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "Sdrbench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2716–2724. DOI: 10.1109/BigData50022.2020.9378449.

[17] *Scientific data reduction benchmarks*, https://sdrbench.github.io/.

[18] S. Jin, J. Pulido, P. Grosset, J. Tian, D. Tao, and J. Ahrens, "Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 45–56.

[19] B. Fang, D. Wang, S. Jin, Q. Koziol, Z. Zhang, Q. Guan, S. Byna, S. Krishnamoorthy, and D. Tao, "Characterizing impacts of storage faults on hpc applications: A methodology and insights," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2021, pp. 409–420.

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[21] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM (JACM)*, vol. 29, no. 4, pp. 928–951, 1982.

[22] F. Knorr, P. Thoman, and T. Fahringer, "Ndzip: A high-throughput parallel lossless compressor for scientific data," in *2021 Data Compression Conference (DCC)*, IEEE, 2021, pp. 103–112.

[23] F. Knorr, P. Thoman, and T. Fahringer, "Ndzip-gpu: Efficient lossless compression of scientific floating-point data on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[24] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[25] X. Chen, J. Tian, I. Beaver, C. Freeman, Y. Yan, J. Wang, and D. Tao, "Fcbench: Cross-domain benchmarking of lossless compression for floating-point data," *Proceedings of the VLDB Endowment*, vol. 17, no. 6, pp. 1418–1431, 2024.

[26] B. Zhang, J. Tian, S. Di, X. Yu, M. Swany, D. Tao, and F. Cappello, "Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 348–359.

[27] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2017, pp. 1129–1139.

[28] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *2018 IEEE International Conference on Big Data*, IEEE, 2018, pp. 438–447.

[29] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, IEEE, 2021, pp. 1643–1654.

[30] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[31] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," *Computing and Visualization in Science*, vol. 19, no. 5–6, pp. 65–76, 2018.

[32] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," pp. 3–15, 2020.

[33] J. Tian, S. Di, X. Yu, C. Rivera, K. Zhao, S. Jin, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data on gpus," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2021, pp. 283–293.

[34] cuZFP, https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp, Online, 2023.

[35] D. Wang, J. Pulido, P. Grosset, S. Jin, J. Tian, J. Ahrens, and D. Tao, "Tac: Optimizing error-bounded lossy compression for three-dimensional adaptive mesh refinement simulations," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, 135–147.

[36] D. Wang, J. Pulido, P. Grosset, S. Jin, J. Tian, K. Zhao, J. Ahrens, and D. Tao, "Tac+: Optimizing error-bounded lossy compression for 3d amr simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 3, pp. 421–438, 2024. DOI: 10.1109/TPDS.2023.3339474.

[37] P. Grosset, C. Biwer, J. Pulido, A. Mohan, A. Biswas, J. Patchett, T. Turton, D. Rogers, D. Livescu, and J. Ahrens, "Foresight: Analysis that matters for data reduction," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 2020, pp. 1171–1185.

[38] S. Jin, D. Tao, H. Tang, S. Di, S. Byna, Z. Lukic, and F. Cappello, "Accelerating parallel write via deeply integrating predictive lossy compression with hdf5," *arXiv preprint arXiv:2206.14761*, 2022.

[39] D. Wang, J. Pulido, P. Grosset, J. Tian, S. Jin, H. Tang, J. Sexton, S. Di, K. Zhao, B. Fang, *et al.*, "Amric: A novel in situ lossy compression framework for efficient i/o in adaptive mesh refinement applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.

[40] S. Di, J. Liu, K. Zhao, X. Liang, R. Underwood, Z. Zhang, M. Shah, Y. Huang, J. Huang, X. Yu, *et al.*, "A survey on error-bounded lossy compression for scientific datasets," *arXiv preprint arXiv:2404.02840*, 2024.

[41] D. Wang, P. Grosset, J. Pulido, T. M. Athawale, J. Tian, K. Zhao, Z. Lukic, A. Huebl, Z. Wang, J. Ahrens, *et al.*, "A high-quality workflow for multi-resolution scientific data reduction and visualization," *arXiv preprint arXiv:2407.04267*, 2024.

[42] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

[43] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Optimizing lossy compression rate-distortion from automatic online selection between sz and zfp," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1857–1871, 2019.

[44] S. Jin, S. Di, F. Vivien, D. Wang, Y. Robert, D. Tao, and F. Cappello, "Concealing compression-accelerated i/o for hpc applications through in situ task scheduling," in *EuroSys 2024*, 2024.

[45] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - preparatory data analytics on peta-scale machines," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 10)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454.

[46] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*, 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178.

[47] 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 the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*, 2011, pp. 1–11. DOI: 10.1145/2063384.2063409.

[48] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock, "Trilinos i/o support trios," *Sci. Program.*, vol. 20, no. 2, pp. 181–196, Apr. 2012. DOI: 10.1155/2012/842791.

[49] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 12)*, 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[50] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 13)*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503279.

[51] U. Ayachit, A. Bauer, E. P. N. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. E. Jansen, B. Loring, Z. Lukić, S. Menon, D. Morozov, P. O'Leary, R. Ranjan, M. Rasquin, C. P. Stone, V. Vishwanath, G. H. Weber, B. Whitlock, M. Wolf, K. J. Wu, and E. W. Bethel, "Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures," in *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*, 2016, 79:1–79:12.