Runway: In-transit Data Compression on Heterogeneous HPC Systems

John Ravi
North Carolina State University
Raleigh, NC, USA
jjravi@ncsu.edu

Suren Byna
The Ohio State University
Columbus, OH, USA
byna.1@osu.edu

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

***Abstract*—To alleviate bottlenecks in storing and accessing data on high-performance computing (HPC) systems, I/O libraries are enabling computation while data is in-transit, such as HDF5 filters. For scientific applications that commonly use floating-point data, error-bounded lossy compression methods are a critical technique to significantly reduce the storage and bandwidth requirements. Thus far, deciding when and where to schedule in-transit data transformations, such as compression, has been outside the scope of I/O libraries.**

In this paper, we introduce Runway, a runtime framework that enables computation on in-transit data with an object storage abstraction. Runway is designed to be extensible to execute user-defined functions at runtime. In this effort, we focus on studying methods to offload data compression operations to available processing units based on latency and throughput. We compare the performance of running compression on multi-core CPUs, as well as offloading it to a GPU and a Data Processing Unit (DPU). We implement a state-of-the-art error-bounded lossy compression algorithm, SZ3, as a Runway function with a variant optimized for DPUs. We propose dynamic modeling to guide scheduling decisions for in-transit data compression. We evaluate Runway using four scientific datasets from the SDRBench benchmark suite on a the Perlmutter supercomputer at NERSC.

***Index Terms*—Object Data Management, In-transit Computation, Heterogeneous Resources**

I. Introduction

Current high-performance computing (HPC) systems process massive amounts of data. These systems are built with high performance interconnects and parallel file systems to support data intensive workloads. Such workloads not only consume, but can also produce large amount of data for post-processing, such as analysis or visualization. In addition, since large-scale simulations often run for multiple hours or even days, these applications typically checkpoint state over some specified duration. Managing data movement can quickly become a barrier to perform scientific research at scale.

Solutions to manage persisted data for large-scale applications can range from using high-level I/O libraries, such as HDF5 [1] and ADIOS2 [2], to application-specific frameworks, such as AMReX native I/O [3]. Application developers typically prefer the I/O libraries, since they offer performance portability while keeping the application code maintainable. These libraries aim to abstract the specifics of the system architecture, parallel file system parameters, and data format. However, current I/O libraries have complex APIs and tuning methods to make efficient use of storage. This motivates a need for simpler I/O libraries, e.g., pMEMCPY [4].

In addition to abstracting optimizations, many high-level I/O libraries provide a way to define and perform computations or transformations while data is being moved between memory and storage. For example, HDF5 [1] offers *filters*, while ADIOS2 [2] offers *plugins*. These extensions enable developers to extend the I/O libraries with new features, which others can use and optimize for their needs. For example, scientific data reduction is an actively researched feature to enable better bandwidth and storage utilization [5]. Although useful, we believe the current development and integration of these features in widely used I/O libraries is still limited. For example, popular I/O libraries offer limited resource handling and scheduling capabilities, which can be critical to high performance applications. Other optimization techniques, such as asynchronous I/O and compression filters, are offered by existing I/O libraries (e.g., HDF5), but are limited to per-application task scheduling. Understanding when to apply a transformation and what system resources to use has been out of scope for I/O libraries.

Recent research has explored the ability to directly read and write to storage from Graphics Processing Units (GPUs). This eliminates the need to buffer data on system memory and frees up CPU cycles to perform other critical tasks. NVIDIA provides driver support to enable I/O to NVMe and NVMe-oF through GPUDirect Storage (GDS) [6]. GDS enables lower I/O latency due to less data transfers. It also allows higher aggregate bandwidth with multiple storage targets not needing to be serialized through a CPU-backed memory buffer. I/O libraries have begun exploring how to support GPUs directly in the data path [6].

The complexity of data management solutions is further increased by the use of accelerators found in many modern HPC systems. In the exascale era of HPC, many applications rely on GPUs as general compute accelerators. Programming heterogeneous computing resources on HPC systems (i.e., CPUs and GPUs) is tricky due to different design considerations. While multi-core CPUs rely on large caches, GPUs use smaller caches and rely on the programmer to optimize memory accesses. GPUs support concurrent execution of orders of magnitude more threads which helps to hide memory access latency. With these design considerations, it is often difficult to provide I/O functionality that can optimally use CPUs and GPUs. Moreover, future systems might see even more specialized accelerators, each with specific algorithm design considerations.

Data Processing Units, or DPUs [7], are becoming more popular for various use cases in HPC systems and data centers. Typically, DPUs are equipped with CPUs and specialized accelerators to offload network-related tasks such as filtering. Offloading I/O and data management tasks to the DPU frees up compute cycles on the host DPU which can be used by the application. Programmers and I/O libraries can take advantage of specialized hardware on the DPU to perform tasks, such as data compression, to further accelerate data movement.

In this work, we investigate the dynamic integration of in-transit data transformation and analysis capabilities in I/O libraries. This requires mechanisms to transparently map and schedule data transformation tasks on available processing resources, and adapt the data transformation parameters to the characteristics of the data. Our proposed mapping and scheduling policy considers the following factors: current and target data location, data transfer costs, and available processing units. We propose *Runway*, a configurable and extensible runtime system for in-transit data processing. Runway builds on top of the Proactive Data Containers (PDC) system [8], [9], a data management framework with a client-server design that offers an object storage abstraction. In PDC, data to be persisted are stored in objects, which are partitioned in data *regions* handled by different PDC servers. As example data transformation task we consider lossy compression, and explore setting the compression error bound adaptively to different partitions of a data object for balancing accuracy and performance.

In summary, we make the following contributions:

- A GPU- and DPU-aware runtime framework enabling computation on in-transit data. Our system, called *Runway*, performs dynamic mapping of data transformation tasks on available compute resources, while adaptively setting the data transformation parameters (e.g., compression error bound) based on the data;
- A dynamic resource mapping scheme based on a cost model taking into account factors such as resource availability and overlapping of compute and data movement;
- An adaptive compression scheme with per-region tuning.

We evaluate Runway on Perlmutter, a state-of-the-art large-scale cluster at NERSC, equipped with AMD EPYC CPUs and NVIDIA A100 GPUs to demonstrate the I/O scalability. We also evaluate Runway on a smaller testbed system that has an NVIDIA A30 GPU and an NVIDIA Bluefield-2 DPU to showcase our proposed dynamic resource mapping scheme across diverse accelerators.

II. Background and Motivation

A. Data Management Software Libraries

Data management software, including high-level and middleware I/O libraries, enables portable performance optimizations across systems and application domains. Popular I/O libraries, such as HDF5, offer a self-describing file format that provides an abstraction layer to manage the data and metadata within a single file [1]. HDF5 filters enable compression of data using a filters approach, where compression is executed on CPUs. A user needs to manually enable and tune the compression method to their application needs. Nonetheless, HDF5 feature set can be extended using Virtual Object Layer (VOL) and Virtual File Driver (VFD). The HDF5 VOL feature has been used to implement an asynchronous I/O VOL connector that enables asynchronous I/O for HDF5 operations using background threads [10]. The scope of this feature, however, is limited to a single application. Currently, HDF5 exists only as a compiled library with no runtime system that arbitrates I/O tasks among multiple applications. Adding a daemon-based runtime system to HDF5 will require significant rework of core library functionality to ensure proper metadata handling.

Proactive Data Containers, or PDC [8], [9], is a data management framework, which offers a data object-focused abstraction instead of a file-based storage abstraction. It is implemented as a runtime system with a set of data management services to perform automatic data movement and metadata search. PDC implements a client-server architecture with a set of servers managing data movement across applications. Hence, this framework enables better resource handling especially in workflow-based applications. Although, arbitrarily increasing the concurrency capacity of PDC with application instances and data servers can have diminishing performance improvement. Figure 1 reports the results of an experiment where we progressively increase the number of PDC data server instances, with each data server running the QVAPOR-IO compression kernel on in-transit data. As can be seen, the system reaches a maximum throughput before saturating the multi-core CPU, and increasing the number of PDC server instances beyond 5 is not beneficial to performance.

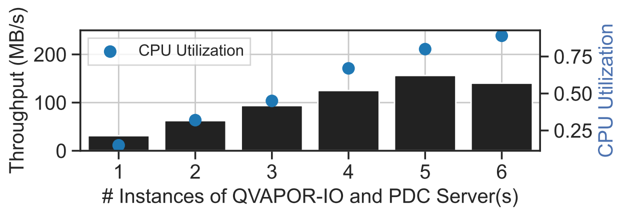


Fig. 1: Data transfer throughput can saturate when a hardware resource is fully utilized. This data transfer performs ZFP compression in-transit while persisting the QVAPOR data object of the Hurricane ISABEL dataset.

PDC currently lacks the extensibility found in other I/O libraries, such as HDF5. In this work, we build Runway on top of PDC's client-server design; thus, Runway inherits PDC's data object abstraction. We propose a novel way of supporting dynamic features, focusing on compression on in-transit data and scheduling to compute accelerators, such as GPUs and DPUs. Supporting dynamic features is critical to enable researchers to integrate their work in production applications. The client-server design allows us to decouple application parallelism with I/O parallelism. The current design of HDF5 filters only supports parallelism with number of MPI ranks the application launches. Moreover, there is no support to perform the computation asynchronously. Runway demonstrates a need to support asynchronous computation and better resource management. We compare Runway's dynamic resource mapped in-transit compression to the current state-of-the-art in-transit compression, namely, HDF5 filters.

B. Hardware Accelerators

Recent work on I/O libraries have also explored supporting hardware accelerators in the data path. HDF5 exposes low-level I/O operations through the virtual file driver (VFD). Since many applications utilize GPUs for compute, HDF5 had added support for GPUDirect Storage (GDS) through the GDS VFD [6]. GPUDirect Storage eliminates the need to buffer data in system memory. GPUs can support computations at very high throughput compared to a multi-core CPUs. Thus, GPU compression with GPUDirect Storage can be used to accelerate I/O throughput [11].

Developers often want to schedule I/O tasks on idle hardware, so they do not contend with resources used by more critical computations, such as CPUs and GPUs. For example, PDC reserves a core on each compute node for its data servers. In this work, we consider offloading data management servers to the DPU. We also explore taking advantage of the DPU's specialized compression accelerator to increase the lossy-based compression throughput.

C. Scientific Data Reduction

Recent work on error bound lossy compression, such as ZFP [12] and SZ [5], has shown that scientific data reduction can yield high compression ratios and still maintain high quality thresholds. Moreover, these data transforms can be implemented efficiently by using high throughput parallel resources, including multicore CPUs and GPUs [13]. These recent developments enable I/O libraries to take advantage of scientific data reduction to improve I/O latency and reduce I/O bandwidth.

The effectiveness of data reduction can depend on multiple factors, including error bounds and entropy of the data. For example, data intended for visualization can tolerate larger error bounds. Larger error bounds allow lossy compression to dramatically reduce data size at the cost of losing some result quality, typically measured with peak signal-to-noise ratio (PSNR). Figure 2 shows how varying the absolute error bound can impact compression ratio, compression latency, and compression quality. See Section IV for more information about the experimental setup. Recent efforts have begun exploring automatically tuning these error bounds [14], [15].

However, for sparse scientific data, the effectiveness of data reduction methods might not be uniformly beneficial at a global data object. For example, object data can contain multiple temporal and spatial data regions. Each of the regions might have different characteristics, leading to different error bounds. Recent work has begun exploring the need for a locally tuned error bound [16]. For example, in Figure 3 we show a data object (QRAIN) from the Hurricane ISABEL dataset. This data object is of dimensions 100x500x500 as shown on the top row of the figure. Because this data object contains both a temporal and spatial dimension, it is a 3 dimensional data object, where the first object indicates the timestep. Analysis is distributed across discrete grids, or data regions, indicated as 100x100 regions in the bottom 3 figures. Some data regions yield much higher compression ratios than others. In this paper, we demonstrate the benefits of supporting a non-uniform compression scheme in I/O middleware.

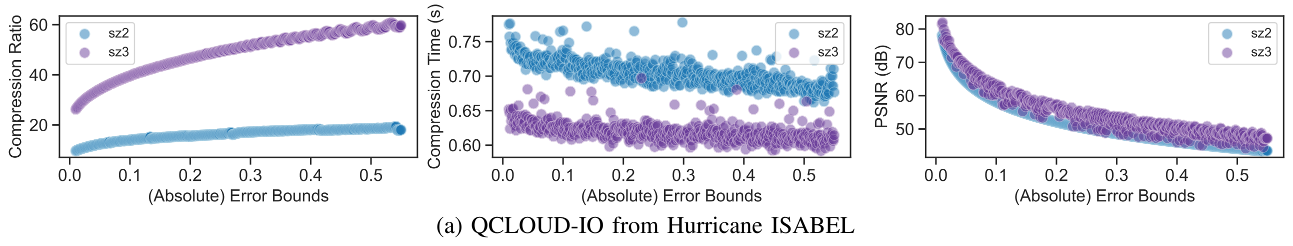
III. The Runway Framework

We design Runway to be a novel object data management service that supports in-transit computation. As mentioned above, Runway builds on the design of an existing object data management service, i.e., Proactive Data Containers (PDC) [8]. Similar to PDC, our framework targets large-scale applications and systems. Because our goal aligns well with PDC's using a server for scheduling in-transit computations and to perform asynchronous I/O, we use PDC as the base framework. Current and upcoming large-scale systems leverage heterogeneous resources to push computing limits. Runway aims to simplify the application developer's effort to manage data in the context of heterogeneous resources. To use our framework, application developers need to replace calls to existing I/O functionality with a simplified data management API. Using a distributed client-server model, Runway can move data in and out of the application memory address space asynchronously using remote procedure calls (RPCs).

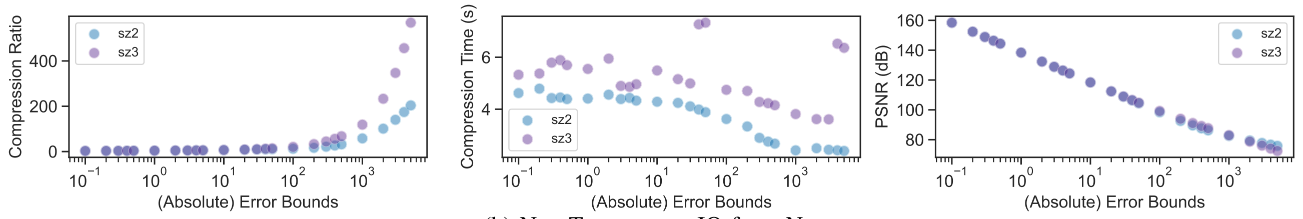
In Figure 4 we show the high-level design of the Runway framework. Runway uses Mercury, a high performance RPC library that facilitates data movement through Remote Memory Access (RMA) [17]. Mercury enables low overhead communication and fast data transfer for large-scale systems. Application developers can register Runway *lambdas*, which are computation operations on data that would be executed in-transit. Runway lambdas are mapped and scheduled on the available compute resources at runtime. We describe each of these features in detail in the following sections.

A. Supporting Object Data Model

Runway implements an object data model used in PDC. Data are organized as a collection of *objects* inside *containers*. Each *object* is composed of a binary blob and metadata, including a name, ID, dimensions, time of data generation, ownership, etc [8]. Large objects are partitioned into smaller *regions* that are defined by starting offsets in the object, element counts, and data sizes. A *region* is the primary way to interface with data; it can reside in any layer of the memory hierarchy (i.e., GPU memory, CPU memory, NVMe, disk, etc.) [9]. This approach enables a simple programming model to interface with data while relying on a flexible runtime to support a deep memory hierarchy. In this paper, we build on this object data model abstraction to enable an extensible computation framework.



(a) QCLOUD-IO from Hurricane ISABEL



(b) Nyx-Temperature-IO from Nyx

Fig. 2: Comparison of the performance of the SZ2 and SZ3 compression algorithms on two applications from SDRBench. The left plots shows how compression ratio (Y-axis) increases with higher absolute error bound (X-axis). The center plots show how the same error bounds (X-axis) can impact compression time. The right plots show how the quality metric, PSNR (Y-axis), varies with the absolute error bound.

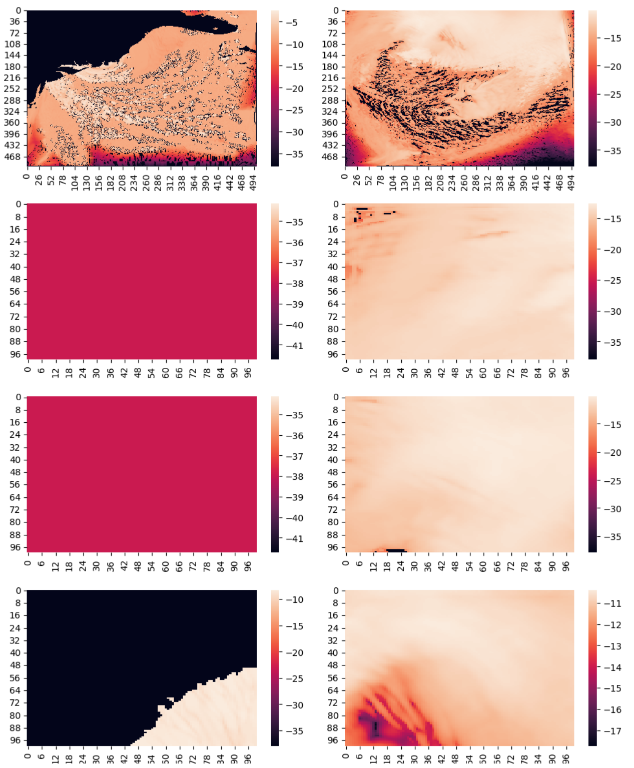


Fig. 3: Hurricane Isabel QRAIN data object with dimensions 100x500x500. The top two plots show the entire 500x500 data object at simulation timesteps 0 and 100, respectively. The bottom six plots show the first three 100x100 data regions in the x-direction for bottom simulation timesteps.

B. Supporting In-transit Computations

The Runway framework supports computations, or operations on data that is in memory. There are two types of lambdas: *data transformations* and *data analysis*. Data transformations are defined as operations that change the input data in some way. Examples of data transformations include compression, decompression, encryption, and decryption. Data analysis produces some new results based on the input. Analysis functions include computing statistical summaries (min, max, histograms, etc.) of a region. Analysis functions can also include calculating the data entropy of a dataset, a useful metric for data reduction.

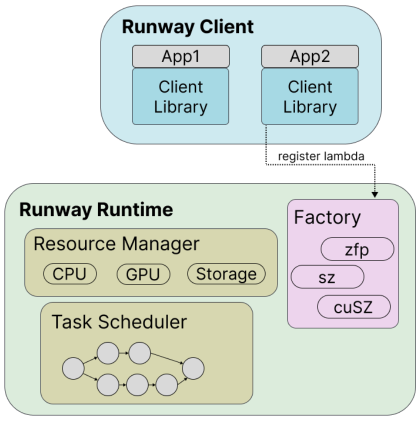


Fig. 4: Overview of Runway which is implemented using a client-server model.

In addition to the built-in lambdas, Runway is designed to be extensible with user-defined plugins. A common interface exists for registering computation functions. Client-sided computations operate on region-only data, whereas server-sided computations operate on object-level data. The region-only data are mapped to the application process address space, so no new data need to be accessed. However, data objects can span multiple data locations, such as across memories of distributed compute nodes. Thus, accessing data objects requires the use of data iterators to fetch data blocks.

A computation function can have multiple variants. For example, multiple approaches currently exist to perform scientific data reduction. The most suitable compression method to use depends heavily on the application and its data. Moreover, each compression function might have different hardware-specific implementations. To capture this, Runway allows for multiple variants to be registered for a computation function. There are two types of interfaces for a computation function: *Direct* and *Iterative*. The *direct interface* supports region-based functions, while the *iterative interface* supports object-based functions. The direct interface is a straightforward method to operate on in-transit data. Data are passed to the direct interface function as a function argument through an opaque pointer. Additionally, metadata about the data region are passed through function arguments. The iterative interface allows computation to operate on entire data objects without needing to map to a smaller region. Since data objects can be very large, and often exceed the memory capacity, data can be traversed with iterators. In this case, the function argument is composed of a data iterator that is used to retrieve the next data block. The runtime will transparently retrieve data blocks that might reside in the working memory of another node in the distributed data server.

C. Supporting Heterogeneous Resources

Our runtime is built to take advantage of accelerators in HPC systems. For computations, multiple variants can be registered where each variant targets a different resource. We can have a variant that is optimized for performing the computation on the CPU and another on the GPU. This allows the Runway system to decide which resource to utilize to perform the computation.

1) GPU: Graphics processing units (GPUs) enable processing data at very high throughput. Recent work has explored directly interacting with data stored in the file system with GPUs. Runway enables using the GPU as a data movement accelerator. We support this by offloading computation, including data transformations and data analysis, to the GPU when compatible code variants are registered.

Figure 6 shows the performance differences between SZ running on a recent data center CPU and cuSZ (i.e., a CUDA implementation of SZ) running on recent data center GPUs. We measure the overall compression throughput of using an A30 GPU to be 2.5 GB/s. Although this is 2-3x higher throughput than a multi-threaded CPU implementation, GPU execution incurs the additional cost for transferring data between the host and device memory.

2) DPU: Data processing units (DPUs) are typically used in data centers to offload networking and communication tasks from the host CPU. A SmartNIC is a type of network interface card (NIC) that includes a programmable CPU that executes the offloaded tasks. In this paper, we use the terms DPU and SmartNIC interchangeably to refer to the NVIDIA Bluefield-2 device. This DPU includes an ARM Cortex-A72 CPU meant to handle less compute intensive tasks, and Runway's worker instances can be fully offloaded to it. The Bluefield-2 DPU also includes specialized accelerators for specific tasks, such as data compression and hashing. Future versions of the Bluefield DPU are planned to include a GPGPU as well.

To understand the performance from offloading a computation to the DPU, we profile a popular compression algorithm on real-world scientific data. As expected, we found the embedded ARM core on the DPU to be much slower than the host CPU, especially for multi-threaded workloads. However, the Bluefield-2 board includes a special-purpose accelerator of DEFLATE, a lossless data compression algorithm that uses a combination of LZ77 algorithm and Huffman coding. Lossy compression algorithms make use of lossless compression as the last step, which we found to be a bottleneck. We devised a variant of SZ that uses ZLIB (DEFLATE), which can be offloaded to the DEFLATE accelerator found on the Bluefield-2 device.

D. Supporting Asynchronous Tasks

More complex computations can add latency to data operations. Also, data transfers to discrete memory found on accelerators can impose extra latency. In order to avoid decreasing the overall application throughput, we implement an asynchronous event system. To automate targeting computation variants for different heterogeneous resources, Runway uses a task scheduler. The task scheduler uses a simple cost model that takes into account the overall latency to perform an in-transit computation. We define the overall latency to include the time to transfer the data to and from an accelerator memory, as shown in Equation 1.

$$t_{latency} = t_{h2d\_time} + t_{compute} + t_{d2h\_time} \tag{1}$$

The data transfer latency can be calculated based on the peak interconnect bandwidth and data size. For synchronous I/O, we keep a running average of past compute kernels' execution time. This is sufficient to statically map to resources. However, when we overlap an I/O phase with compute (asynchronous I/O), the challenge of avoiding contention due to oversubscribing resources becomes an issue. Thus, we propose an empirical model to estimate the execution time of an in-transit computation based on runtime data, such as tracking *device utilization* during execution. We define "device utilization" as the percentage of time spent busy over a sampling period.

To estimate the compute latency of performing an in-transit compression, we find a correlation between device utilization, data size, and compression latency. Equation 2 shows a cubic polynomial which takes two input variables, $x_{i,0}$ (device utilization) and $x_{i,1}$ (data size) to predict the compression latency y_i. The i parameter represents an index into the past history of previous measurements. Using least squares of a cubic polynomial, we can fit a line of best fit for the data collected on each device. Each device we fit this model to will have a different set of β_0, β_1, β_2, and β_3.

$$f_{est\_compute} \implies y_i = \beta_0 * x_{i,0}^3 + \beta_1 * x_{i,1}^2 + \beta_2 * x_{i,1} + \beta_3 \tag{2}$$

TABLE I: System Configuration

	Testbed Two-node system	NERSC Perlmutter Large scale cluster
CPU	2x Intel Xeon ES-2530 v4, 10-Core	AMD EPYC 7763 64-Core
RAM	126GB DDR4	256GB DDR4
GPU	NVIDIA A30 PCIe 24GB	NVIDIA A100 SXM4 40GB
DPU	NVIDIA Bluefield-2	-
OS	Ubuntu 18.04.5	SUSE Linux 15
Drivers	NVIDIA Driver 515.48.07, CUDA 11.7	

TABLE II: Benchmarks used for Evaluation

Dataset	Data Objects	Entropy	Dimensions	Mem. Req.
Nyx	Temperature	23.99	512x512x512 Single Precision	512 MB
Hurricane ISABEL	QCLOUD	1.30	100x500x500 Single Precision	100 MB
	QRAIN	21.45		
	QVAPOR	24.19		
QMCPACK	QMCPack (einspline)	26.08	115x69x69x28 Single Precision	612 MB
S3D	Pressure	26.77	500x500x500 Double Precision	1 GB
Miranda	Density	22.5	96 regions of 3072x3072x3072 Single Precision	106 GB

In summary, the dynamic resource mapping scheme uses a cost matrix that takes into account the following parameters: 1. *current data location*, 2. *transfer costs*, 3. *target data location*, and 4. *available compute units*. We use empirical data from past runs to refine the model for future operation.

IV. Experimental Setup

With a mixture of I/O kernels, computation kernels, and real world data from scientific applications, we evaluate Runway on a testbed system and a large-scale cluster. In this section, we describe the systems and benchmarks in detail.

A. System Configuration

Perlmutter is a pre-exascale supercomputing system with 200 petaflops (PF) performance located at OLCF [18]. It is composed of 1,536 GPU nodes and 3072 CPU nodes. Each GPU-accelerated node features four NVIDIA A100 GPUs and one AMD "Milan" EPYC 7763 CPU. The memory subsystem in each GPU node includes 40GB of HBM2 per GPU and 256GB of host DRAM. Each CPU node features two AMD EPYC CPUs with 512GB of memory per node. The entire compute system is connected to a HPE Cray's ClusterStor E1000 storage with 35 PB of storage space. It is an all-flash file system, built on a Lustre file system, with an aggregate bandwidth of $>$ 5 TB/sec and 4 million IOPS (4 KiB random).
Our smaller **testbed** system is equipped with two Intel Xeon CPUs, two NVIDIA A100 GPUs, and an NVIDIA Bluefield-2 DPU. Each GPU has 24GB of HBM2 memory and 126GB of host DRAM. The DPU has an embedded ARM Cortex-A72 SoC with 16GB of DRAM. It has an Ethernet network interface with dual ports of 25 Gb/s. Both the GPU and DPU are connected over PCIe and serve as offload accelerators to the host CPU. Refer to Table I for a summary of the system configurations.

B. I/O Kernels

We implement I/O kernels using scientific datasets found in the Scientific Data Reduction Benchmark, SDRBench [19]. SDRBench is a standard compression assessment benchmark suite that contains multiple real-world scientific datasets across different domains. Metadata, which document how to parse the data from the binary files, are provided for each dataset. Refer to Table II for a summary of the datasets used in this paper.
Nyx is a massively parallel, adaptive mesh, cosmology simulation. During its execution, it stores simulation state composed of particle data for checkpoint-restart of the simulation or post-analysis visualization. The dataset found in SDRBench has post-analysis Nyx simulation data composed of 3D arrays in space of size 512x512x512. Each particle contains 6 fields of single-precision floating-point data: *velocity x*, *velocity y*, *velocity z*, *temperature*, *dark matter density*, and *baryon density*. Our I/O kernel treats each field as a separate data object. Since all of the fields have a similar data entropy, we only show results for one of the data object (temperature) in our paper.
Hurricane ISABEL is a climate simulation application. The dataset contains 13 single-precision floating-point fields where each field is a 3D array of 100x500x500. The first dimension is a simulation timestep. We evaluate three of the fields, each represented as a data object, in our I/O kernel. Of the three fields we evaluate, *QCLOUD* has much lower data entropy than *QRAIN*, and *QVAPOR*.
QMCPACK is an ab initio quantum MonteCarlo package for analyzing the electronic structure of atoms, molecules, and solids. In this dataset, there is one field called 'einspline', which represents the state stored in memory during the simulation. In our I/O kernel, we represent this field as a single-precision floating-point data object with size 115x69x69x288. The first three dimensions represent the x,y,z coordinates and the last one is orbital index.
S3D is a combustion simulation application. The dataset contains 11 fields components each of which is a 3D array of double-precision floating-point values of size 500x500x500. We evaluate the pressure component as a separate data object in our S3D-I/O kernel.
Miranda is a hydrodynamics simulation code used to study instability growth of turbulent mixing. This dataset has a single data object, density, from a late time step of a simulation run on a 3072x3072x3072 uniform grid. The density data object have been partitioned into 96 regions of dimensions 3072x3072x32. Some of the regions in this data set have zero entropy (all have the same value).

C. Computation Kernels

Using Runway's dynamic extensions, we implement interfaces to two error-bound lossy compression transforms: *ZFP* and *SZ*. In addition to in-transit compression kernels, we also utilize a compute kernel which computes π.
ZFP [12] is a lossy compression library for floating-point data. It contains four critical steps: (1) partition data into grids of 4^d blocks; (2) convert each block to a fixed-point

representation; (3) decorrelate values by applying orthogonal transforms; (4) perform bit manipulation (an embedded coding from MSB to LSB), then truncation. ZFP implements three modes to bound compression error: fixed rate, fixed accuracy, or fixed precision. The fixed rate mode compresses a block to a fixed number of bits. The fixed precision mode compresses to a variable number of bits but keeps the number of bit planes fixed. The fixed accuracy mode compresses a block with relation to a tolerated maximum error.

SZ [20] is a modular parametrizable lossy compressor framework for scientific data. It contains four critical steps: (1) predict data values based on a model; (2) apply linear quantization; (3) perform variable-length encoding; (4) perform lossless compression using existing algorithms. SZ provides three modes to bound compression errors: absolute error bound, relative error bound, and peak-to-signal noise ratio (PSNR). *cuSZ* [13] is a CUDA implementation of SZ which performs all of the algorithm steps on a GPU.

We propose *dpuSZ* as a DPU implementation of SZ that utilizes the lossless compression acceleration of the NVIDIA Bluefield-2. Due to its superior compression ratio and speed, SZ3 uses Zstandard (ZSTD) by default to perform its lossless compression [21]. However, we revisit using ZLIB as SZ's lossless compressor due to ZLIB being based on DEFLATE, which is accelerated on the Bluefield-2 DPU.

BBP-π is a compute kernel that implements the Bailey-Borwein-Plouffe (BBP) algorithm to calculate the n-th hexadecimal digit of π without calculating the first $n-1$ digits [22]. Although the BBP algorithm can calculate any arbitrary digit of π, it still scales linearithmically, $\mathcal{O}(n \log n)$. We use this kernel to vary the utilization of compute units. This enables us to explore how our dynamic resource mapping strategy performs when overlapping a computation kernel other than compression kernels. We implement two variants of this algorithm—an OpenMP implementation targeting multi-core CPU and a CUDA implementation targeting a GPU.

V. Experimental Evaluation

A. Accelerated Offloading

We first look at the performance of SZ compression on the BlueField-2 DPU and the NVIDIA A100 GPUs available in our testbed system.

DPU execution - Figure 5 shows the execution time breakdown of the critical steps of the SZ algorithm on DPU. As discussed in Section IV-C, SZ is a modular compression framework, which allows multiple implementations for different steps of the algorithm. In the figure, ZSTD identifies the default SZ implementation; ZLIB is the version of SZ that uses a software implementation of ZLIB in the lossless compression step, and DPDK_ZLIB is the version that uses the DEFLATE accelerator available on DPU for this last step. For all three versions, we have a single- and a 4-threaded OpenMP implementation: the former using a single Arm core, and the latter using all four Arm cores of the DPU (4 cores). As can be seen, for ZSTD the lossless compression part of the SZ algorithm has execution time comparable to the other steps.

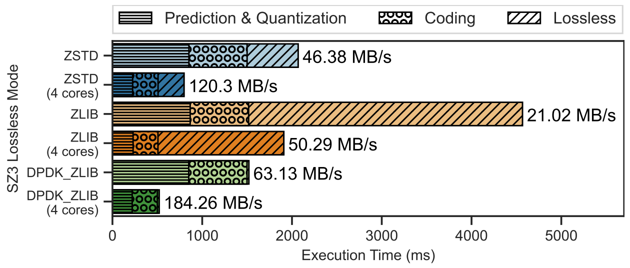


Fig. 5: Comparison of different modes of SZ (Y-axis) executed on the DPU with a breakdown of execution time (X-axis) of each step of the SZ algorithm. The QVAPOR data object from the Hurricane ISABEL dataset is being compressed in-transit with Runway. The compression throughput for each mode is indicated on the right of each bar.

With ZLIB, software-implemented lossless compression is 6× slower than ZSTD with similar compression ratio. However, thanks to the hardware acceleration of DEFLATE on the DPU [23], DPDK_ZLIB is 27× faster than ZLIB. This improves the overall compression throughput of the SZ algorithm by a factor of 2.34× over the default ZSTD mode on the DPU. Later, we will refer to this variant of SZ as dpuSZ3.

CPU vs. GPU vs. DPU compression - We now consider three variants of SZ: (Variant 1) *SZ3* targeting host CPU, (Variant 2) *dpuSZ3* targeting the DPU, and (Variant 3) *cuSZ* targeting the GPU. With a breakdown of the cost of data transfers and compression latency for each device, Runway can decide which variant, thereby which device, to use at runtime. Figure 6 plots the comparison between directly writing the data to storage and performing in-transit compression with each variant. As can be seen, using SZ on the QVAPOR data object from the Hurricane ISABEL dataset improves the I/O write latency due to having to write less data. However, the compression latency of executing on the embedded cores of the DPU results in an overall slowdown compared to directly writing with no compression. On the other hand, using CPU and GPU compression can be beneficial to overall I/O performance despite the compression overhead.

Based on the data transfer costs between CPU and accelerator (GPU or DPU) and the compression latency, a static resource mapping method would choose the GPU-based cuSZ variant. However, this mapping decision might not be the best one if the GPU is being utilized by application code or other compression tasks. Static resource mapping makes sense when performing I/O synchronously or when the offload target is not fully utilized. When performing I/O asynchronously, in-transit data compression can contend with the computation phase of the application.

B. Per-region Tuning

Here, we evaluate Runway's non-uniform compression scheme where each region is independently tuned instead of performing uniform compression on the whole data object. In Figure 7, we show how the entropy for a 3072x3072x32 region of the 3072x3072x3072 Density object in the Miranda

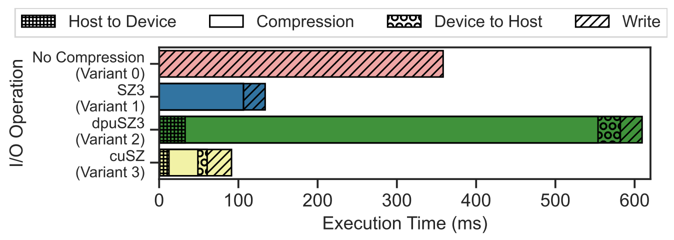


Fig. 6: Comparison of different variants of SZ (X-axis) executed on the testbed using Runway on the QVAPOR data object from the Hurricane ISABEL dataset. Variant 0 does not use compression, while variants 1, 2 and 3 perform compression on CPU, DPU and GPU, respectively. A breakdown of data transfers to and from the device, compression latency, and write time is indicated by the type of shaded region on the stacked bar plot.

dataset is not-uniform. This type of entropy in a data object showcases the benefit for performing non-uniform compression. In Figure 8, we plot the compression ratios achieved when performing uniform compression for the data object vs non-uniform compression for each region. When the entropy varies dramatically within the region, the non-uniform compression performs about 15% more compressibility. When the entropy is similar in the region, there is little difference in overall achieved compression ratio. The main takeaway from this experiment is that predictors work better locally, especially when entropy varies dramatically within a region. This is evident when we skip the lossless stage for SZ with the SZ_BEST_SPEED compression mode; there is a 46.5% improvement in compressibility.

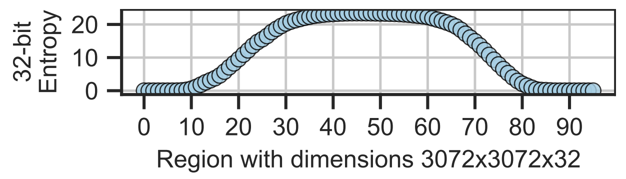


Fig. 7: The 32-bit Entropy for each 3072x3072x32 region of the 3072x3072x3072 Density object in the Miranda dataset. The x-axis is the region number in the z-dimension.

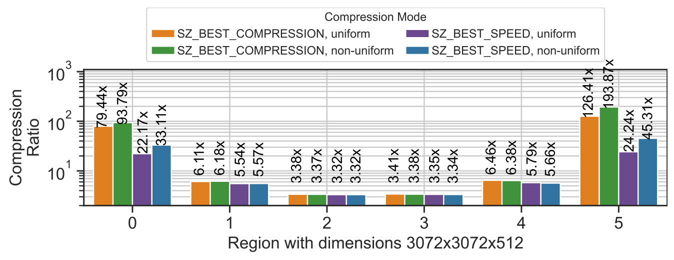


Fig. 8: Comparison of uniform compression and non-uniform compression with SZ on the Miranda dataset. SZ_BEST_COMPRESSION optimizes for compression ratio, while SZ_BEST_SPEED skips the lossless stage. The Y-axis (log-scale) shows the compression ratio achieved (also noted on top of each bar). The X-axis specifies a 3072x3072x512 region of the Density object in the dataset.

We also evaluate how Runway performs at large scale while comparing it with an existing solution of using HDF5 filters. Figure 9 shows how I/O scales exponentially and quickly becomes a bottleneck at larger scale. In this experiment, we increase the amount of data proportionally to the number of MPI ranks (i.e., weak scaling). Performing SZ3 compression in-transit with HDF5 or Runway on the entire data object improves the overall I/O latency dramatically. We also compare with performing per-region based non-uniform compression which allows each region to be compressed independently. Note, the Y-axis is plotted with a log-scale. Per region compression exposes additional parallelism and relaxes the error bound for regions that are highly compressible, thus we observe slightly better latency with non-uniform compression. At scale, non-uniform mode improves I/O time by around 20% over uniform mode.

C. Dynamic Resource Mapping

With the experiments in this section, we make a case for the need to implement dynamic resource mapping when performing in-transit compression asynchronously. The set of experiments in Figure 10 show multiple instances of the same I/O kernel using the same resource to perform SZ compression. It is clear that device utilization correlates with compression latency. In some cases when the device utilization is above a threshold, such as 0.7, the compression latency is 2-3× slower to perform on the GPU. We rely on the model proposed in Section III-D to predict the compression latency based on empirical data obtained in all of these multiple compression instances.

Figure 11 shows how the model fits with the data obtained on the testbed GPU. Since this is a 2D projection of a cubic polynomial model, the prediction line appears to be disjoint. We plot the same data with a 3D projection in Figure 12a. In

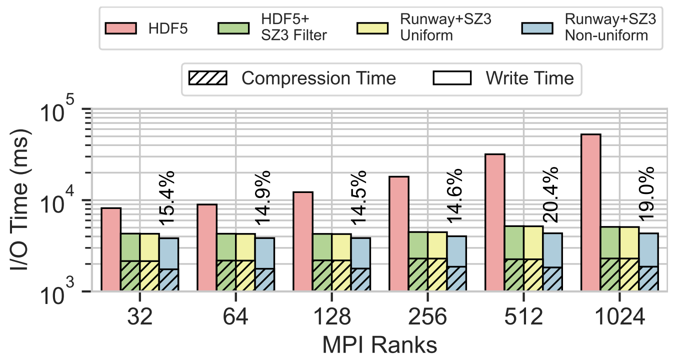
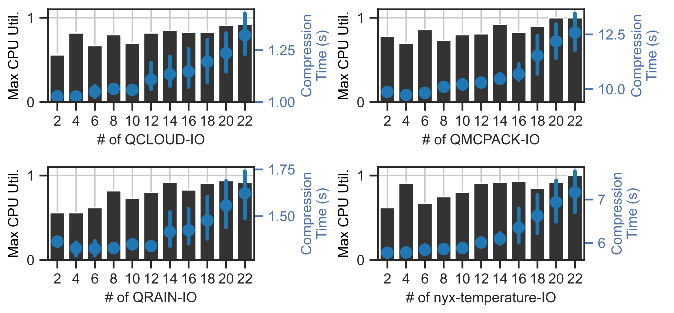
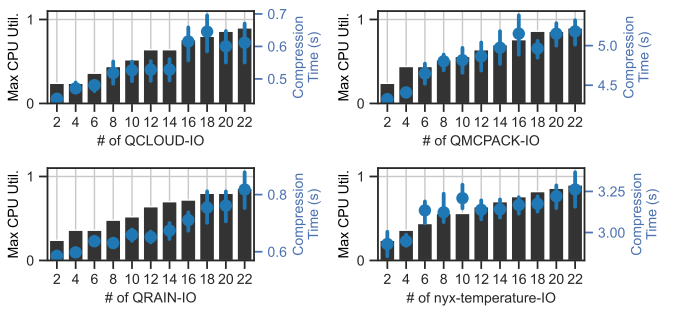


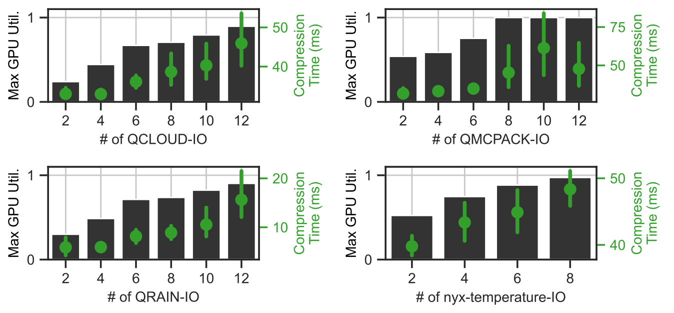
Fig. 9: Comparison of scalability of Runway at large scale on Perlmutter when performing in-transit SZ3 compression on the Pressure data object from the S3D dataset. The total I/O time is shown on the Y-axis (log-scale), and the number of MPI ranks is shown on the X-axis. The stacked bar plot shows how much time is spent on SZ3 compression and in performing the write operation. On top of each bar, we show the percentage improvement for non-uniform mode over uniform mode.



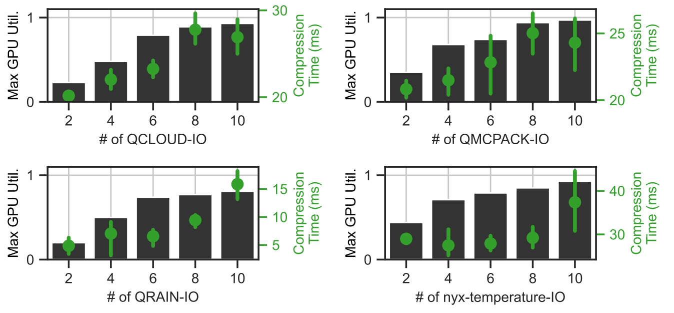
(a) SZ on Testbed



(b) SZ on Perlmutter



(c) cuSZ on Testbed



(d) cuSZ on Perlmutter

Fig. 10: Each plot measures how the processing unit utilization (left Y-axis, indicated as bars) and SZ compression latency (right Y-axis, indicated as points) scales when increasing the number of I/O kernel instances (X-axis). For each point, we plot the range of compression latency measured for all instances.

the 3D projection, we do not color code each data point with the type of data object being compressed as we did in Figure 11. In Figure 12, we plot the prediction estimated with our empirical cost model as a surface plot, which varies with the device utilization and data size for each data object. Each data point represented as a circle on the plot indicates a separate I/O call with in-transit compression enabled.

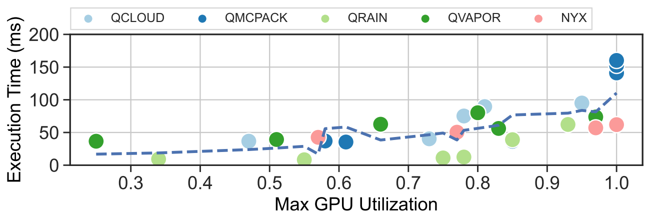
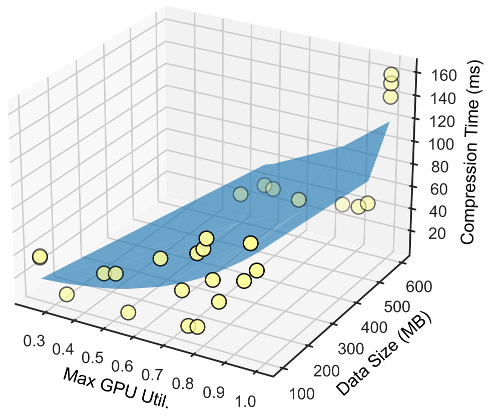


Fig. 11: Changes in SZ compression time (Y-axis) based on the GPU utilization (X-axis). The dashed line is our prediction based on all past measurements done on the GPU on the testbed system. This is a 2D projection of GPU utilization and data size (not shown). Each data object from all the runs are highlighted are color-coded.

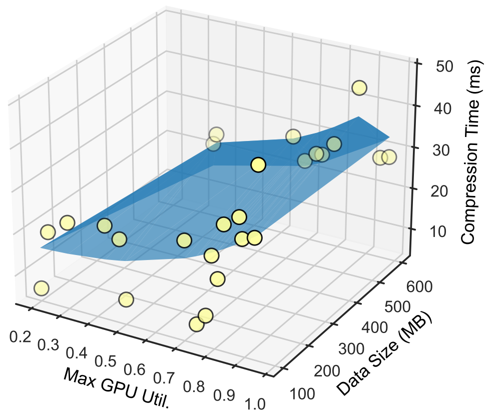
The correlation for device utilization, data object size, and compression latency is adequate for the GPU device utilization with a $r^2 = 0.6$ and $r^2 = 0.5$ on the A30 and A100, respectively. For the CPU utilization, the correlation is very strong with a $r^2 = 0.96$ and $r^2 = 0.99$, on the Intel Xeon and AMD Epyc, respectively. Since the GPU performs compression very fast compared to that on CPU, especially for smaller data sizes (few milliseconds), the observed performance variability is higher compared to the CPU correlation. This is a limitation of using *nvidia-smi* which supports a polling interval of 1 ms or above to monitor the GPU utilization. In practice, data sizes for real world applications will be much larger, so the variability would be less with the same GPU utilization monitor.

Furthermore, the model can be used to predict the compression latency even when a different compute kernel is running concurrently, such as BBP-π. In Figure 13, we run together an in-transit SZ compression of QVAPOR-IO data object and compute BBP-π algorithm. The prediction model indicates the compression latency roughly doubles after 0.7 GPU utilization. Our dynamic resource mapping scheme takes advantage of this modeling to determine which device to utilize to reduce contention.

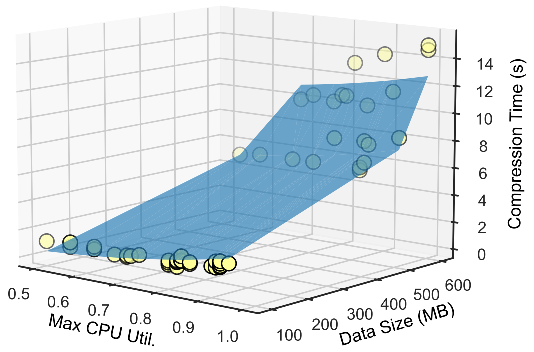
When performing asynchronous I/O, the DPU can be a better offload target when the host CPU and GPU are busy. In Figure 14, we demonstrate three different scenarios: (1) QVAPOR-IO performing an in-transit SZ compression while co-running a CPU version of BBP-π; (2) QVAPOR-IO performing an in-transit cuSZ compression while co-running a GPU version of BBP-π; (3) QVAPOR-IO performing an in-transit dpuSZ compression while co-running both a CPU and GPU version of BBP-π. When we increase the workload on the CPU and GPU (compute more digits of BBP-π), we see a speedup by offloading compression to the DPU. At 64k digits of π, the overall application time is 9% faster.



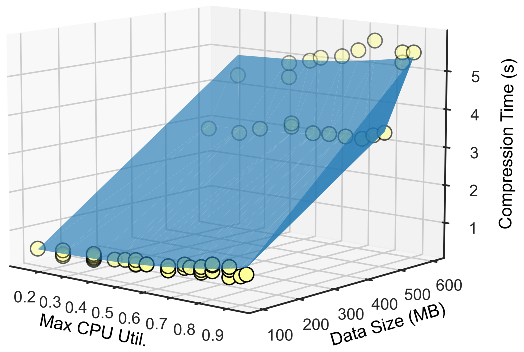
(a) GPU (A30) on testbed. r^2 is 0.6.



(b) GPU (A100) on Perlmutter. r^2 is 0.5.



(c) CPU (Intel Xeon) on testbed. r^2 is 0.96.



(d) CPU (AMD Epyc) on Perlmutter. r^2 is 0.99.

Fig. 12: Comparison of correlation among our prediction model, device utilization, and data size of each data object (X-axis and Y-axis) with the SZ compression latency on the vertical Z-axis. Each I/O with in-transit compression is plotted as a circle in each plots. The prediction is plotted as a surface plot in the 3D projection.

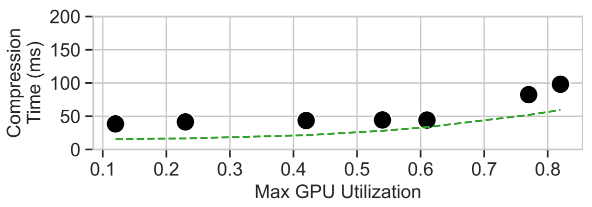


Fig. 13: Multiple workloads (QVAPOR-IO and BBP-π) sharing same resource (A30 GPU). We vary the BBP-π duration to vary the GPU utilization (X-axis). We show how increasing the GPU utilization will increase the compression latency (Y-axis). The predicted time based on our prediction is shown as a dotted line.

VI. Conclusion and Future Work

In this paper, we introduce Runway, a client-server I/O runtime capable of in-transit data transformation and analysis. With Runway, we explore the benefits of offloading in-transit compression to two accelerators: GPU and DPU. We introduce a cost model to determine the target resource for a data transformation task dynamically based on runtime parameters. We evaluate our cost model with extensive experiments on real-world scientific datasets. Our evaluation and analysis have highlighted the need to implement a dynamic resource mapping scheme while performing in-transit data compression asynchronously. Finally, we explore per-region compression, which exposes additional parallelism and improves latency. In future, we plan to evaluate Runway on workflow-based science applications that use the data object abstraction to save simulation state for checkpointing or visualization.

VII. Acknowledgements

This manuscript has been authored by authors at North Carolina State University supported under National Science Foundation's award CNS-1812727. This research was partially supported by The Ohio State University under a subcontract (GR130303), which was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL. This research also used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and the resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

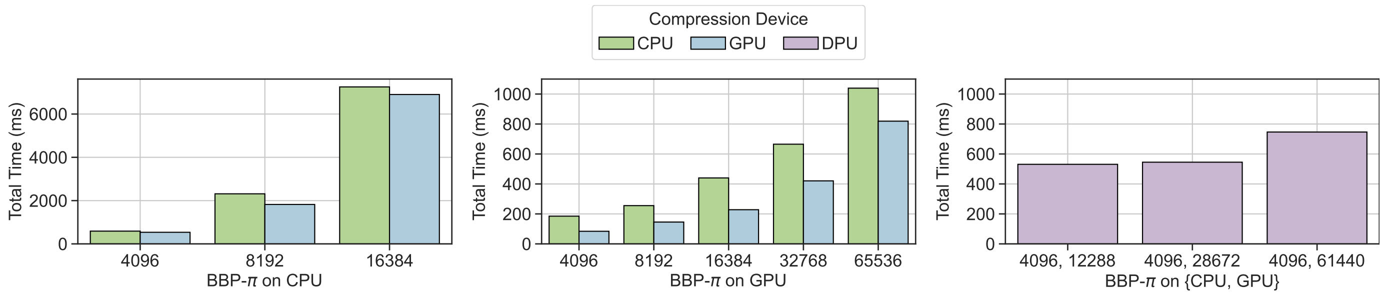


Fig. 14: We compare the total time for performing an in-transit SZ compression asynchronously for QVAPOR-IO dataset while performing an BBP-π computation. The x-axis shows which device computes BBP-π and number of digits. The legend on top of the figure shows which device performs the in-transit compression.

References

[1] M. Folk *et al.*, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[2] Q. Liu *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, ISSN: 1532-0634.

[3] W. Zhang *et al.*, "Amrex: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021. DOI: 10.1177/10943420211022811. [Online]. Available: https://doi.org/10.1177/10943420211022811.

[4] L. Logan *et al.*, "Pmemcpy: A simple, lightweight, and portable i/o library for storing data in persistent memory," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 664–670. DOI: 10.1109/Cluster48925.2021.00098.

[5] S. Jin *et al.*, "Improving prediction-based lossy compression dramatically via ratio-quality modeling," *The 38th IEEE International Conference on Data Engineering (ICDE 2022)*, [Online]. Available: https://par.nsf.gov/biblio/10319819.

[6] J. Ravi *et al.*, "Gpu direct i/o with hdf5," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, 2020, pp. 28–33. DOI: 10.1109/PDSW51947.2020.00010.

[7] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20. DOI: 10.1109/HCS52781.2021.9567066.

[8] H. Tang *et al.*, "Toward scalable and asynchronous object-centric data management for hpc," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 113–122. DOI: 10.1109/CCGRID.2018.00026.

[9] H. Tang *et al.*, "Parallel query service for object-centric data management systems," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 406–415. DOI: 10.1109/IPDPSW50202.2020.00076.

[10] H. Tang *et al.*, "Transparent Asynchronous Parallel I/O using Background Threads," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[11] L. C. V. Real *et al.*, "User-defined functions for HDF5," *CoRR*, vol. abs/2109.11709, 2021. arXiv: 2109.11709. [Online]. Available: https://arxiv.org/abs/2109.11709.

[12] J. Diffenderfer *et al.*, "Error analysis of zfp compression for floating-point data," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, A1867–A1898, 2019. DOI: 10.1137/18M1168832. [Online]. Available: https://doi.org/10.1137/18M1168832.

[13] J. Tian *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*, ser. PACT '20, Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 3–15, ISBN: 9781450380751. DOI: 10.1145/3410463.3414624. [Online]. Available: https://doi.org/10.1145/3410463.3414624.

[14] D. Krasowska *et al.*, "Exploring lossy compressibility through statistical correlations of scientific datasets," in *2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7)*, 2021, pp. 47–53. DOI: 10.1109/DRBSD754563.2021.00011.

[15] R. Underwood *et al.*, "Optzconfig: Efficient parallel optimization of lossy compression configuration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3505–3519, 2022. DOI: 10.1109/TPDS.2022.3154096.

[16] X. Liang *et al.*, "Toward feature-preserving vector field compression," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–16, 2022. DOI: 10.1109/TVCG.2022.3214821.

[17] J. Soumagne *et al.*, "Advancing rpc for data services at exascale," *IEEE Data Eng. Bull.*, vol. 43, pp. 23–34, 2020.

[18] NERSC, *Perlmutter*, NERSC, 2022. [Online]. Available: https://www.nersc.org/systems/perlmutter.

[19] K. Zhao *et al.*, "Sdrbench: Scientific data reduction benchmark for lossy compressors," in *2020 IEEE International Conference on Big Data (Big Data)*, Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2020, pp. 2716–2724. DOI: 10.1109/BigData50022.2020.9378449. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/BigData50022.2020.9378449.

[20] X. Liang *et al.*, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, pp. 1–14, 2022. DOI: 10.1109/TBDATA.2022.3201176.

[21] Zstd, 2015. [Online]. Available: https://github.com/facebook/zstd/releases.

[22] D. H. Bailey, "The bbp algorithm for pi," Sep. 2006. DOI: 10.2172/983322. [Online]. Available: https://www.osti.gov/biblio/983322.

[23] Intel DPDK, *Data plane development kit project page*, 2022. [Online]. Available: https://www.dpdk.org.