Processing Particle Data Flows with SmartNICs

Jianshen Liu
Computer Science & Engineering
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
jliu120@ucsc.edu

Carlos Maltzahn
Computer Science & Engineering
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
0000-0001-8305-0748

Matthew L. Curry
Scalable System Software
Sandia National Laboratories
Albuquerque, NM, USA
mlcurry@sandia.gov

Craig Ulmer
Scalable Modeling & Analysis
Sandia National Laboratories
Livermore, CA, USA
cdulmer@sandia.gov

Abstract—**Many distributed applications implement complex data flows and need a flexible mechanism for routing data between producers and consumers. Recent advances in programmable network interface cards, or SmartNICs, represent an opportunity to offload data-flow tasks into the network fabric, thereby freeing the hosts to perform other work. System architects in this space face multiple questions about the best way to leverage SmartNICs as processing elements in data flows. In this paper, we advocate the use of Apache Arrow as a foundation for implementing data-flow tasks on SmartNICs. We report on our experiences adapting a partitioning algorithm for particle data to Apache Arrow and measure the on-card processing performance for the BlueField-2 SmartNIC. Our experiments confirm that the BlueField-2's (de)compression hardware can have a significant impact on in-transit workflows where data must be unpacked, processed, and repacked.**

Keywords—SmartNICs, compression, in-transit computations

I. INTRODUCTION

Distributed applications routinely need an efficient mechanism for routing data from collections of producers to collections of consumers. For example, in high-performance computing (HPC), scientific simulation workflows generate parallel streams of output that data management services gather, analyze, and transform before archiving to storage. Similarly, geographic information system (GIS) data flows often aggregate sensor data from a variety of sources, reorganize it, and then transmit derived data products to remote subscribers based on user specifications. Current-generation systems implement their distributed data-flow operations in host-level software that communicates through standard network protocols.

Commercial hardware vendors have recently started supplementing their products with embedded processing resources that may offer significant advantages for systems that implement distributed data flows. Storage vendors offer computational storage devices (CSDs) with user-programmable FPGAs and processors that allow the user to execute computations at the disk. These resources allow filtering operations to be placed close to data sources and may dramatically reduce the amount of data returned to the user. In the networking arena, vendors are placing embedded processors in network interface cards (NICs). These *SmartNICs* can help process in-transit data and are especially useful in distribution tasks where data flows must transmit customized data products to multiple destinations.

The presence of these programmable, embedded processors throughout the system architecture motivates the use of *eusocial methods*, where many low-capability devices are programmed to collectively act towards a more complex, system-level goal [1]. Our work in eusocial methods is focused on constructing general-purpose data management software that makes it easier for system developers to map different data flows onto a collection of embedded processors in the network and storage fabrics. There are many questions architects face in this space. How should in-transit data be represented and processed? How well do current embedded processors perform fundamental data-sifting operations? When is data compression profitable in these architectures?

In this paper, we focus on constructing a computational engine that can perform a variety of eusocial tasks on SmartNICs. We advocate for the use of Apache Arrow [2] to standardize how in-transit data is represented and processed by eusocial devices. As a means of better evaluating current generation hardware, we have implemented a partitioning algorithm for particle datasets on the NVIDIA BlueField-2 VPI SmartNIC [3] and measured its performance with different datasets. Additional experiments with hardware accelerators on this card confirm that compression hardware can have a significant impact on performance and motivate the need for Arrow enhancements in future work.

II. PARTICLE DATA FLOWS

Many HPC and GIS applications operate on *particle data* and rely on complex data management services to route particle state information between producers and consumers distributed throughout the network. While particle datasets are smaller in size than multimedia datasets, application data flows can be challenging to implement because the datasets contain many small items that are tedious to inspect. As such, it is beneficial to consider hardware environments that can offload the task of reorganizing and sifting through the data. In this section we provide application examples from the HPC and GIS spaces, and discuss a common data-flow use case where data is transitioned from a spatially-organized form to a temporally-organized form.

A. Particle Simulations in Scientific Computing

Many simulations in scientific computing employ particle-in-cell (PIC) methods [4, 5, 6] to model different phenomena. These simulations manage a large collection of discrete particles and track their progress as they move through time and space. A particle is defined by a small amount of state information, such

as its position, velocity, charge, and type. Given that simulation fidelity improves as the number of particles in the simulation increases, researchers typically leverage parallel simulation techniques to distribute the data and work across many compute nodes. Although a simulator's particle data may contain a treasure trove of information for scientists, the sheer size of this data makes it infeasible to save except in the case of occasional checkpointing. Analysts may run supplemental analysis applications in parallel with the simulation and inspect subsets of the data without impeding the simulation.

B. Asset Tracking with Geographic Information Systems

A similar need to collect and process large amounts of particle data can be found in GIS applications that manage sensor data about real-world assets, such as airplanes, ships, and land vehicles. Although the "particles" in these systems are significantly larger in physical size than those in the simulations, the processing challenges for manipulating the data flows are the same: different distributed sensor systems produce continuous feeds of observation data that needs to be reorganized to be of value to downstream consumers.

C. Reorganizing Spatial Indices to Temporal Indices

One hardship of working with particle data flows is that there are significant differences between the way producers and consumers expect data to be organized. Producers typically organize data in a spatial manner, where each sensor generates an update for all items in a physical region during a particular time interval. In contrast, analytics consumers often need data organized temporally for each item. As illustrated in Fig. 1 with airplane data, temporal tracks can yield better insight into patterns of activity than positional snapshots alone.
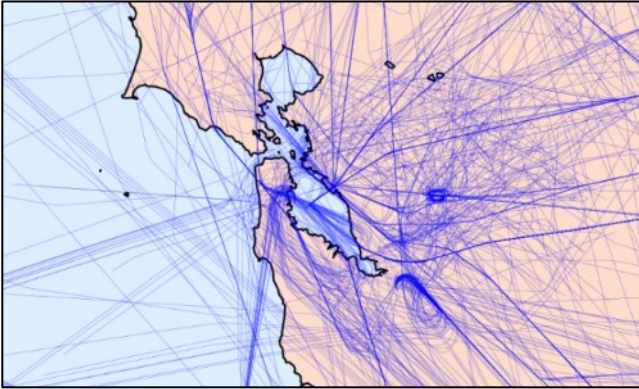

Fig. 1. Airplane tracks reveal patterns of activity

Distributed, log-structured merge (LSM) trees [7] are a convenient mechanism for converting particle data flows from a spatial organization to a temporal organization when multiple producers and consumers are involved. In this approach, a collection of processing elements is used to reorganize data as it moves through different stages in the tree. Each processing element absorbs incoming data until it reaches its storage capacity. When compaction is necessary, data is split based on particle IDs and transmitted to the next appropriate processing element in the tree. While processing elements only need to store, sift, and transmit blocks of particles, they can greatly improve the searchability of online datasets.

III. SMART NETWORK INTERFACE CARDS (SMARTNICS)

Over the last decade, hardware vendors have introduced programmable network interface cards or SmartNICs that enable users to place custom computations at the edge of the network fabric. While the original motivation for developing SmartNICs was to allow security researchers to monitor and inspect network flows in real time [8, 9], the need for tighter infrastructure control in cloud computing platforms has driven SmartNIC vendors to create more powerful cards. Vendors such as NVIDIA (formerly Mellanox), Fungible, Chelsio, Intel, and Xilinx have constructed SmartNICs that allow users to embed computations at the network's edge. While some SmartNIC architectures employ FPGAs or ASICs to maximize packet processing performance, most feature a multicore embedded processor that is easier for developers to leverage.

A. NVIDIA BlueField-2 VPI SmartNIC

The NVIDIA BlueField-2 VPI SmartNIC is a programable NIC that features eight A72 ARM processor cores, 16GB of DRAM, and 60GB of flash storage. It includes two 100Gb/s network ports that can interact with either InfiniBand or Ethernet. The default software stack for the card boots an Ubuntu 20.04 installation of Linux. The ARM processors can be configured to either (1) intercept traffic between the host and network or (2) serve as a separate host that shares access to the network ports. The BlueField-2 includes three hardware acceleration units to improve application performance: a RegEx engine for data filtering, a SHA-2 unit for encryption functions, and a (de)compression engine.

B. BlueField-2 Compression Accelerator

Data compression is important in data-intensive applications because it reduces the amount of data that needs to be transmitted through the network, cached in memory, and stored on disk. Most big data I/O libraries (e.g., Avro [10], Parquet, ORC [11], and Arrow IPC) feature built-in compression support for a variety of codecs. As such, any application that processes this data must be capable of decompressing and compressing the data in a manner that is compliant with the library's data format.

The BlueField-2's compression accelerator can efficiently compress and decompress data using the DEFLATE algorithm [12]. DEFLATE is widely used and is a key part of standards such as PNG [13], HTTP [14], TLS [15], and SSH [16]. The BlueField-2's hardware implementation of DEFLATE is compatible with the zlib library, which means that when needed, data compressed by the hardware can be decompressed by software and vice versa. This interoperability is critical in eusocial applications because it enables data processing tasks to be "pushed down" to offload the host or "pushed back" in situations where an embedded device becomes saturated with work. The BlueField-2's compression hardware is currently accessed through the Data Plane Development Kit (DPDK) [17, 18], which is a library for constructing high-performance data-plane applications on top of a variety of network hardware devices. The compression hardware is designed to process a stream of individual data packets in an efficient manner and includes DMA hardware to facilitate the movement of data between the accelerator and memory. Compression performance for the BlueField-2 hardware is discussed in Section VI.

C. Performance Expectations for SmartNICs

It is important to recognize that the processing resources available to a SmartNIC are considerably less than those available to a host. In prior benchmark experiments [3], we observed that the BlueField-2's ARM processors performed roughly an order of magnitude slower than host systems, due to a combination of processor and memory bandwidth limitations. The gap between embedded and server processors is *not unexpected* and is unlikely to change in the foreseeable future.

While SmartNICs are not general-purpose accelerators, there are several scenarios where we expect the hardware to be beneficial to applications. First, SmartNICs are sufficient for performing simple filtering and projection operations that do not involve complex computations. Second, data flows involving compression or encryption may be able to leverage the card's hardware accelerators and achieve a speedup. Finally, it may be advantageous to offload low-rate, asynchronous event processing to a SmartNIC, due to the disturbances these operations have on other tasks that run on the host.

IV. SOFTWARE INFRASTRUCTURE FOR IN-TRANSIT PROCESSING

Data-intensive applications in both science and commercial enterprises are often constructed using multiple systems with independent implementation histories and choices of programming languages. A key operating expense of these applications is the movement of data across these systems. But what sounds like a problem of moving data between systems is really the challenge of efficiently (1) converting the data from a system's internal in-memory representation to a wire format and (2) accessing large amounts of data via record-by-record API calls. This is precisely the challenge that Apache Arrow set out to address, an open-source project that since 2016 has been quickly gaining adoption within the data science community.

A. Apache Arrow

The key insight underlying the design of Apache Arrow ecosystem is that by creating an efficient open data processing platform around a common and efficient in-memory data representation with many different programming language bindings (so far C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust), data can move efficiently between the ecosystem's data processing engines running on different systems. Data processing and exchange can be implemented with a number of building blocks that include the Parquet file format [19], the Flight framework for efficient data interchange between processes [20], the Gandiva LLVM-based JIT computation for executing analytical expressions by leveraging modern CPU SIMD instructions to process Arrow data [21], and Awkward Array for restructuring computation on columnar and nested data [22]. On top of these building blocks exist a number of Arrow integration frameworks, including the Fletcher framework that integrates FPGAs with Apache Arrow [23], NVIDIA's RAPIDS cuDF framework that does similar for GPUs [24, 25], the Plasma high-performance shared-memory object store [26], the Skyhook distributed storage plug-in to embed Arrow processing engines within Ceph storage objects [27, 28], and the Substrait effort to standardize an open format for query plans between query optimizers and processing engines [29]. There are many more projects that are adopting the Apache Arrow in-memory representation and the Dataset

Interface that abstracts over a variety of file formats and other data sources [30]. The amount of significant investment poured into this ecosystem is reflected by its recent cadence of four major version releases per year, most recently version 9.0.0 with 1061 issues resolved by 114 distinct contributors over 3 months.

B. Data Organization in Apache Arrow and Opportunities

Apache Arrow represents tabular data in a columnar, randomly-accessible in-memory format, allowing for nested data structures and null values. The format is designed to maximize CPU throughput by optimizing the data layout for pipelining, SIMD instructions, and cache locality. Data is communicated by schema information involving one or more optional metadata dictionary batches that are followed by record batches. A record batch is composed of multiple arrays, each representing a part of the data from one or more fields from a table. Record batches are designed to be the unit of data processing communicated to and from processing engines. Batching of records minimizes the need for record-based API calls and the batch size can be optimized for pipeline processing while the columnar layout allows for SIMD instructions [31].

Arrow IPC format is a protocol that encodes record batches into contiguous bytes for storing in either files or memory. This encoding process is known as serialization. Fig. 2 shows how a typical Arrow table is serialized into a byte sequence in the IPC format.
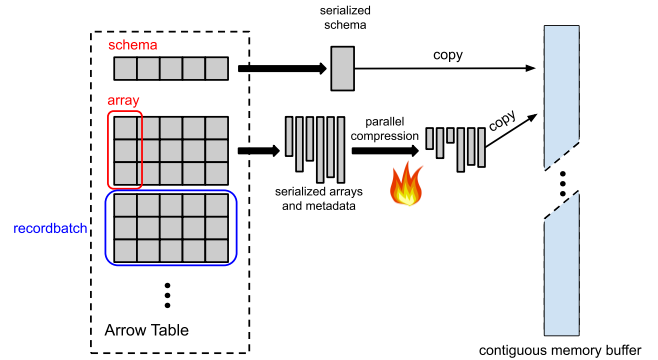

Fig. 2. A simplified data serialization process of an Arrow table

The schema of a table is first serialized and written to the output memory buffer. Then, for each record batch, the arrays it contains will be serialized one after another according to their types. With all record batches being serialized resulting in a buffer vector, these buffers will be compressed in parallel for better compression throughput. The number of threads that will be spawned in this process typically matches the size of the buffer vector. For most array types, serializing an array produces two buffers — a data buffer and an additional buffer containing the metadata called the validity bitmap. As a result, the total number of threads started is a multiple of the number of columns, which can keep many cores busy. For example, in one of our datasets for experiments, the loaded Arrow table contains 17 columns; however, the compression phase spawns 35 threads occupying more cores than the ones available from a CPU socket. As such, the generated compression workload may hinder or stall performance-critical applications such as simulations that are running on the same host. Leveraging the compression accelerator from the BlueField-2 SmartNIC

provides an opportunity to break the dependence of compression performance on intensive computing resource occupation.

V. Data Partitioning Experiments

As a means of better evaluating the suitability of Apache Arrow for processing particle data flows on SmartNICs, we implemented a data partitioning algorithm used in LSM trees. In this work, we used Apache Arrow to represent particles in a tabular form that is suitable for transfer over the network and leveraged Arrow's filtering operations to split a table into smaller tables based on particle IDs. We measured the amount of time required to unpack, partition, and repack data for three particle datasets from different communities to demonstrate the flexibility of this approach.

A. Implementation

We constructed a C++ program that inspects and processes in-transit data objects in network data flows. This program is supplied with a contiguous-memory data object and is expected to provide one or more contiguous result objects that are to be sent to different locations. For this work we use Apache Arrow's IPC methods to handle transformations between a serialized object that can be transported in the network and an in-memory format that is suitable for tabular computations. The partitioning algorithm examines a table and uses a small number of bits in the particle ID field to determine which output table should hold each particle. Although Arrow provides a group-by function that would be useful for performing a split in a single pass, it is currently limited to statistical operations. As such, we implemented the partitioning as a multistep algorithm that executes a select query to generate each table. While far from ideal, this approach is acceptable in the LSM tree work because of the low-fanout requirements of the distributed algorithm.

B. Reference Datasets

Three particle datasets were used in these to provide better insight into the performance of the algorithm with different data:

- **TrackML Particle Tracking Challenge** ("Particles") [32]: CERN supplied a particle simulation dataset for a machine learning competition hosted through Kaggle in 2018. This dataset contains 10 numerical fields per particle.
- **OpenSky Network** ("OpenSky Planes") [33]: The OpenSky Network collects worldwide ADSB information for airplanes from volunteers. Entries contain 16 fields composed of a mix of numerical and string values.
- **NOAA Maritime** ("Ships") [34]: NOAA provides historical AIS position data for ships near the US coastline. Daily data was converted to a particle format that contained 17 fields composed of a mix of numerical and string values.

Given that the BlueField-2 SmartNIC operates with 16GB of DRAM, we set a 1GB limit for the size of uncompressed data to use in our experiments. We decompressed each dataset, selected the number of rows that would be closest to 1GB in size, and then recompressed the data to serve as input to the experiments.

C. Experiments

Performance experiments were conducted on a compute node that features a 32-core AMD EPYC 7543P processor and a BlueField-2 VPI card. In the first experiment, we measured the overall amount of time for the host or SmartNIC to unpack, partition, and repack the tabular data into 2 to 16 output partitions. As depicted in Fig. 3, the host operates roughly four times faster than the BlueField-2 when processing uncompressed data. Increasing the number of partitions increased the processing time in most cases. A closer inspection of the Particles dataset revealed an ID address space issue that resulted in a distribution imbalance. These issues can be mitigated by hashing the ID or selecting ranges that are more meaningful to the application.
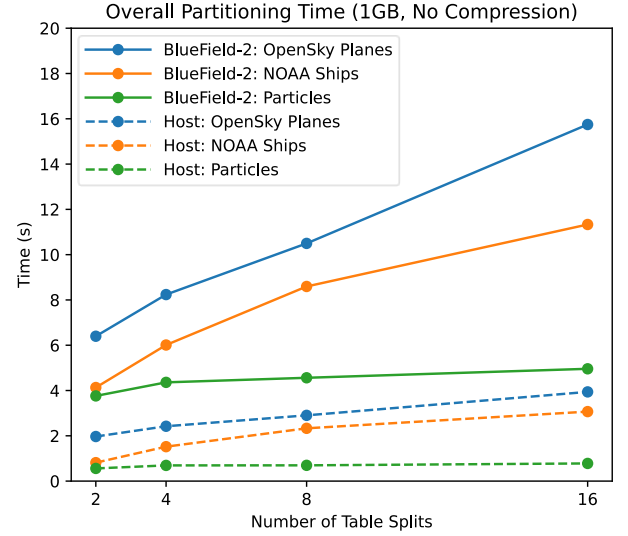

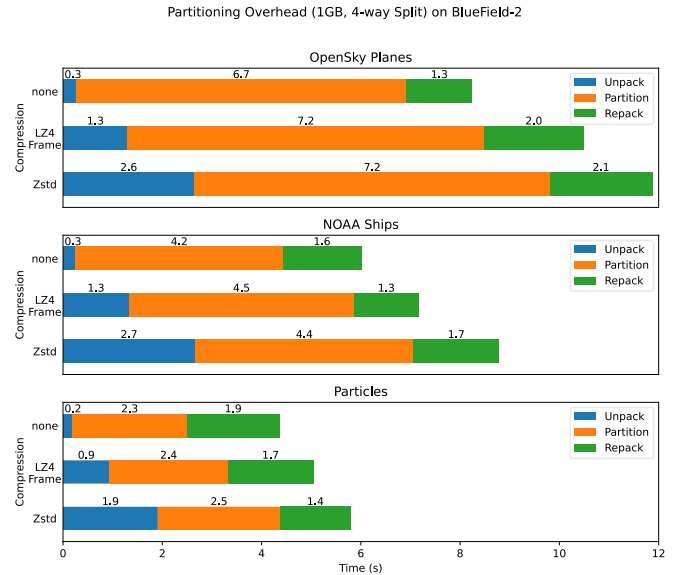Fig. 3. Overhead for partitioning without compression

Fig. 4. Timing breakdown for a 4-way split on the BlueField-2 using different software compression methods

The second experiment examines the impact of Apache Arrow's built-in software compression mechanisms on performance. These tests vary whether the input and output objects are serialized with no compression, LZ4 Frame compression [35], or Zstd compression [36]. Fig. 4 provides the

timing breakdowns for unpacking, partitioning, and repacking 1GB of particle data when performing a 4-way split. As expected, uncompressed data is significantly faster to read than compressed data. Repacking the data, however, is similar in all cases. This overhead highlights the fact that serialization by itself is an expensive operation.

D. Discussion

While the host processors in this experiment yielded better performance, it is important to note that the BlueField-2's embedded processors were performant enough to be of value in many data flows. Scenarios where producers generate periodic bursts of data are applicable, as the SmartNIC can absorb the bursts and process the data before the next wave arrives.

Implementing the partitioning operation with Apache Arrow highlighted its development advantages. Arrow's well-reasoned data primitives and existing support for serialization, compression, and processing greatly simplified the implementation effort. Our implementation worked with all three datasets without modification, even though each dataset had different data components and ID bitwidths. Although the current version of Arrow does not have all the primitives of a higher-level library such as Pandas [37], it contains adequate primitives to implement a variety of operations.

VI. COMPRESSION EXPERIMENTS

As demonstrated in the previous section, converting between on-the-wire and in-memory formats is an important and time-consuming task for systems that process in-transit tabular data. Given that the Bluefield-2 SmartNIC provides a compression accelerator and multiple cores that Apache Arrow can leverage, it is worthwhile to explore the different compression options that are available for packing and unpacking data. We conducted three experiments to answer each of the following questions: (1) Is the compute overhead caused by software-based compression significant enough to justify offloading the (de)compression to hardware accelerators? (2) How does the throughput performance of hardware-based compression compare with software-based compression in a threaded environment? (3) Is there a change in the compression ratio between the hardware- and software-based methods?

A. Challenges

The BlueField-2's compression hardware can be accessed through the Data Plane Development Kit (DPDK) library. Unfortunately, this library is highly tuned for network operations and is organized around a packet-processing model that can be cumbersome for other types of applications. We faced several challenges in adapting DPDK's compression functions to process our Arrow data. First, individual data packets have a maximum size of 64KB. To compress larger amounts of data, developers must slice input and output buffers into packet-sized segments and then generate a packet that contains a list of compression commands for processing each segment. Second, converting between contiguous and segmented data representations can result in extra memory allocations and copies that disrupt the throughput of the data flow through the compression hardware. Optimizing the pipeline requires a detailed understanding of both DPDK and the hardware, and is tedious for users that simply want to

(de)compress large blocks of data. Third, embedded hardware environments have limited resources. Therefore, recycling resources after each compression operation (while still managing errors) is extremely important. Finally, a single ARM CPU core may not be sufficient for maximizing the performance of the compression accelerator. As such, it is valuable to construct a pipeline that pre-allocates memory and divides work among cores as needed.

B. Implementation: Bitar

To simplify accessing the compression hardware for data compression, we implemented the *Bitar* [38] library on top of DPDK and Arrow. Bitar provides a convenient (de)compression API and features zero-copy processing, synchronous and asynchronous operation, and multicore/multidevice support. It is specifically designed to operate without root privileges, which is uncommon in DPDK-based applications. Bitar also allows users to access the BlueField-2's compression hardware from either the host's or BlueField-2's processors.

C. Experiments

All experiments in this section were carried out on a CloudLab [39] host that has two AMD EPYC 7542 CPUs (a total of 64 cores), 512GB of DDR4 memory, and a BlueField-2 SmartNIC connected with PCIe 4.0 x16 lanes. Each experiment was run on all three reference datasets with a maximum outstanding data window size of 160MB due to memory constraints imposed by DPDK and the pipelined nature of the compression hardware.

Since Bitar has not yet been fully integrated into Arrow, our experiments compress Arrow tables differently depending on whether software- or hardware-based compression is measured. The software-based approach relies on Arrow's existing compression mechanisms, which serialize and compress each column independently before writing the final output buffer (i.e., "inner compression"). In contrast, the hardware-based approach serializes the entire table and then streams the data through the compression hardware (i.e., "outer compression"). While the former is preferred, the latter is sufficient for network transfers.

1) Software Compression Overhead for a Single Thread: Our first research question focuses on whether software-based compression overhead is significant enough to justify hardware acceleration. To answer this question, we constructed an experiment that measures the amount of time for a single thread to pack and unpack Arrow data in software using different codecs. We intentionally excluded the memory allocation time in this experiment given that it can be preallocated using knowledge of historical output buffer sizes.
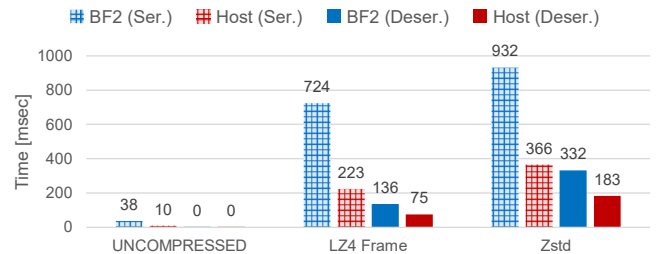

Fig. 5. Single-thread (de)serialization time with different codecs

Timing results for the Particles dataset (see Fig. 5) indicate that (de)serialization without compression is efficient, thanks to the zero-copy buffer design of Arrow's IPC format. However, involving either LZ4 Frame or Zstd compression introduces significant CPU overhead and increases time consumption by one to two orders of magnitude. We observed similar results using the other two reference datasets. Given that compression is a significant impediment to performance, we conclude that acceleration is worthwhile in performance-sensitive applications.

2) Throughput in a Threaded Environment: Our second question focuses on how well the software- and hardware-based compression methods perform in a threaded environment. One advantage of Arrow is that it automatically parallelizes the packing and unpacking of tables by dispatching each column's work to its own thread. In Bitar's case, multiple threads can be used to maximize the amount of work supplied to the compression hardware. We conducted experiments to observe how (de)compression throughput improves when scaling to an optimal number of worker threads.
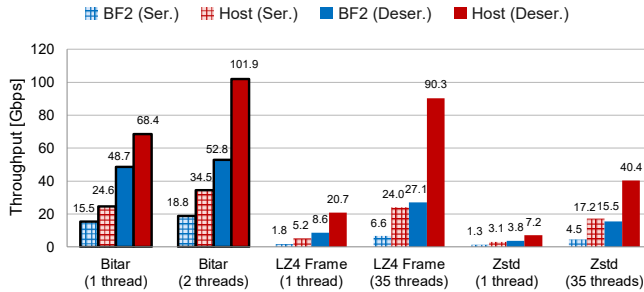

Fig. 6. (De)serialization throughput with different compression codecs and degree of parallelism. Thick black borders indicate hardware (de)compression results.

Fig. 6 shows throughput measurements for the "OpenSky Planes" reference dataset. Without limiting the number of threads in the experiment, both LZ4 Frame and Zstd used 35 threads during compression and decompression. In contrast, the hardware compression throughput with Bitar was maximized when using only two threads. Bitar outperformed the software-based approaches in every case for this dataset. For software-based compressions, LZ4 Frame performs better than Zstd in terms of throughput. However, Bitar's performance is better than that of LZ4 Frame by 3.3x and 1.1x in the case of single-thread and multithreaded performance, respectively.

Measurements for the other two reference datasets produced similar results. The software-based approach was observed to outperform the 2-thread Bitar configuration, but only in situations where a wide dataset with many columns enabled the host to leverage many cores (e.g., >35). Conservatively speaking, the throughput of the compression accelerator rivals that of a software implementation that consumes all the cores of a modern CPU socket. Having the ability to (de)compress data at near network speeds and with only a fraction of the system's available compute cores is essential for building streaming data analytics. The results of this experiment also highlight the performance scalability problem of using general-purpose cores for (de)compression tasks, as throughput depends on the number of columns in the data.

3) Impact on Compression Ratio: Our third question focuses on quantifying how the compression ratio changes when switching between different configurations of the software- and hardware-based compression methods. The compression ratio is computed by dividing the compressed IPC buffer size for a particular configuration by the uncompressed IPC buffer size. We expect the ratio to change in the Bitar hardware implementation because (1) a different compression algorithm is used and (2) the implementation applies compression on the entire table instead of individual columns.

The compression ratios for different configurations are presented in Fig. 7. Results listed for Bitar are presented for one and two threads to illustrate that splitting the work into multiple threads does not have a significant impact on output size. The hardware-based compression using the DEFLATE algorithm provides a compression ratio that is between that of the LZ4 frame and Zstd codecs in all three datasets. These measurements confirm that offloading computations to the BlueField-2's compression accelerator does not result in a significant sacrifice in the compression ratio.
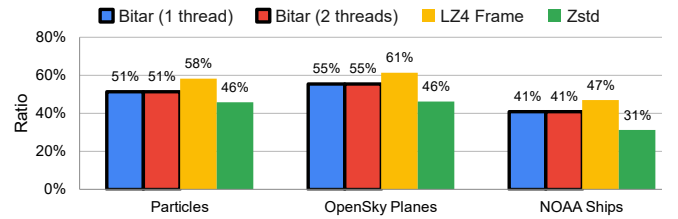

Fig. 7. Compression ratios under different compression approaches

VII. Summary and Future Work

While current-generation SmartNICs are slower at processing data than servers, they can perform fundamental data-sifting tasks that are commonly required by different workflows. The compression hardware is particularly appealing for this work, as it provides a way to efficiently unpack, process, and repack in-transit data products. However, the current interface for accessing the hardware is challenging to leverage and an obstacle for developers. We present Bitar as a reusable library for simplifying compression on the BlueField-2 cards.

Apache Arrow provides a data model and a collection of operators that are particularly well-suited for processing data on eusocial devices. The tabular notation allowed us to devise a general framework for storing and processing particle data that did not need to be adjusted when switching between datasets. Other types of data may not map to a tabular form as elegantly.

There are multiple paths forward from this work. Having completed the on-card processing work we will transition to network tasks related to distributing data between SmartNICs and coordinating resource utilization across a distributed system. Based on the TCP bottlenecks observed in previous work, it is imperative that these operations take place with RDMA primitives. For the compression work, Arrow will need minor adjustments to allow it to more fully leverage Bitar. The adjustments will allow Arrow to use hardware compression on individual columns, thereby allowing finer-grained data access by applications.

BIBLIOGRAPHY

[1] P. Kufeldt, C. Maltzahn, T. Feldman, C. Green, G. Mackey and S. Tanaka, "Eusocial Storage Devices - Offloading Data Management to Storage Devices that Can Act Collectively," *;login: The USENIX Magazine,* vol. 43, no. 2, pp. 16-22, 2018.

[2] The Apache Software Foundation, "Apache Arrow: A cross-language development platform for in-memory analytics," 2022. [Online]. Available: https://arrow.apache.org.

[3] J. Liu, C. Maltzahn, C. Ulmer and M. L. Curry, "Performance Characteristics of the BlueField-2 SmartNIC," *arXiv preprint arXiv:2105.06619,* 2021.

[4] F. H. Harlow, "The particle-in-cell computing method for fluid dynamics," *Methods in Computational Physics,* vol. 3, pp. 319-343, 1964.

[5] T. Arber, K. Bennett, C. Brady, A. Lawrence-Douglas, M. Ramsay, N. Sircombe, P. Gillies, R. Evans, H. Schmitz and A. Bell, "Contemporary particle-in-cell approach to laser-plasma modelling," *Plasma Physics and Controlled Fusion,* vol. 57, no. IOP Publishing, p. 113001, 2015.

[6] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov and N. Aunai, "Smilei: A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation," *Computer Physics Communications,* vol. 222, no. Elsevier, pp. 351-373, 2018.

[7] I. L. Picoli, P. Bonnet and P. Tözün, "LSM Management on Computational Storage," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, New York, NY, USA, 2019.

[8] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift and T. Lakshman, "UNO: Uniflying host and smart NIC offload for flexible packet processing," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 506-519.

[9] G. Sabin and M. Rashti, "Security offload using the SmartNIC, A programmable 10 Gbps ethernet NIC," in *2015 National Aerospace and Electronics Conference (NAECON)*, IEEE, 2015, pp. 273-276.

[10] D. Vohra, "Apache avro," in *Practical Hadoop Ecosystem*, Springer, 2016, pp. 303-323.

[11] B. Vaddeman, "Data formats," in *Beginning Apache Pig*, Springer, 2016, pp. 201-208.

[12] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," May 1996. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1951.

[13] T. Boutell, "Png (portable network graphics) specification version 1.0," 1997.

[14] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff and D. Hellerstein, "Delta encoding in HTTP," 2002.

[15] S. Hollenbeck, "Transport layer security protocol compression methods," 2004.

[16] T. Ylonen and C. Lonvick, "The secure shell (SSH) transport layer protocol," 2006.

[17] I. Cerrato, M. Annarumma and F. Risso, "Supporting fine-grained network functions through Intel DPDK," in *2014 Third European Workshop on Software Defined Networks*, IEEE, 2014, pp. 1-6.

[18] Linux Foundation, "Data Plane Development Kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org.

[19] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem*, Springer, 2016, pp. 325-335.

[20] W. McKinney, "Introducing Apache Arrow Flight: A Framework for Fast Data Transport," 13 October 2019. [Online]. Available: https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight.

[21] R. Pindikura, "Introducing the Gandiva Initiative for Apache Arrow," Dremio, 6 2018. [Online]. Available: https://www.dremio.com/blog/announcing-gandiva-initiative-for-apache-arrow/. [Accessed 12 08 2022].

[22] J. Pivarski, P. Elmer and D. Lange, "Awkward arrays in python, c++, and numba," *EPJ Web of Conferences,* vol. 245, no. 05023, p. 05023, 16 November 2020.

[23] J. Peltenburg, J. van Straten, L. Wjtemans, L. van Leeuwen, Z. Al-Ars and H. Hofstee, "Fletcher: A framework to efficiently integrate FPGA accelerators with apache arrow," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2019, pp. 270--277.

[24] RAPIDS Development Team, "RAPIDS: Collection of Libraries for End to End GPU Data Science," 2018. [Online]. Available: https://rapids.ai.

[25] S. Raschka, J. Patterson and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information,* vol. 11, p. 193, 2020.

[26] P. Moritz and R. Nishihara, "Plasma In-Memory Object Store," 8 August 2017. [Online]. Available: https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store.

[27] J. Chakraborty, I. Jimenez, S. Alvarez Rodriguez, A. Uta, J. LeFevre and C. Maltzahn, "Skyhook: Towards an Arrow-Native Storage System," *CCGrid22,* 16-19 May 2022.

[28] J. Chakraborty, C. Maltzahn, D. Li and T. Drabas, 31 January 2022. [Online]. Available: https://arrow.apache.org/blog/2022/01/31/skyhook-bringing-computation-to-storage-with-apache-arrow/.

[29] J. Nadeau, "Substrait: Cross-Language Serialization for Relational Algebra," 12 08 2022. [Online]. Available: https://substrait.io/.

[30] Apache Arrow, "The Apache Arrow Dataset Interface," 2022. [Online]. Available: https://arrow.apache.org/docs/python/api/dataset.html.

[31] J. Nadeau, "Scale by the bay 2019: Jacques Nadeau, Vectorized Query Processing for CPUs using Apache Arrow," 23 Dec 2019. [Online]. Available: https://youtu.be/hLm duqB3Y4.

[32] S. Amrouche, L. Basara, P. Calafiura, V. Estrade, S. Farrell, D. R. Ferreira, L. Finnie and N. Finnie, "The tracking machine learning challenge: accuracy phase," in *The NeurIPS'18 Competition*, Springer, 2020, pp. 231-264.

[33] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic and M. Wilhelm, "Bringing up OpenSky: A large-scale ADS-B sensor network for research," in *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, IEEE, 2014, pp. 83-94.

[34] National Oceanic and Atmospheric Administration, "Vessel Traffic: AIS Vessel Tracks," [Online]. Available: https://coast.noaa.gov/digitalcoast/data/vesseltraffic.html. [Accessed 12 8 2022].

[35] Y. Collet, "LZ4 Frame Format Description," 8 dec 2020. [Online]. Available: https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md. [Accessed 12 8 2022].

[36] Y. Collet and M. Kucherawy, "Zstandard Compression and the application/zstd Media Type," 2018.

[37] W. McKinney and others, "pandas: a foundational Python library for data analysis and statistics," *Python for high performance and scientific computing,* vol. 14, no. Seattle, pp. 1-9, 2011.

[38] J. Liu, "Simplify accessing hardware compression/decompression accelerators," 7 2022. [Online]. Available: https://github.com/ljishen/bitar.

[39] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb and others, "The Design and Operation of CloudLab," in *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1-14.