A highly scalable parallel design for data compression

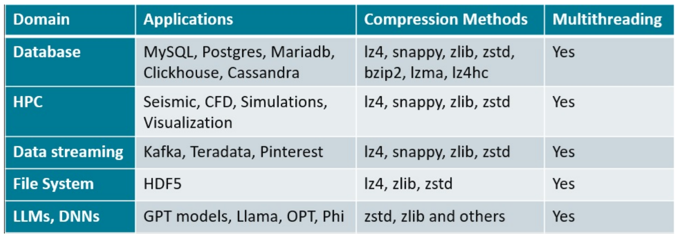
S. Biplab Raut
AMD India Private Limited, Bangalore, India
biplab.raut@amd.com

***Abstract*—With ever-increasing use of digital data, many applications rely on data compression for their needs related to processing, storage and communication over the network of large volumes of data. While compression saves the memory/disk space and decreases the communication time, there is a considerable runtime spent in this process. Parallel compression algorithms and solutions that are developed to speed up the operations do not scale well on the multi-core CPUs. The data parallel schemes implemented by the prior arts are inefficient in partitioning the data and scaling the performance on the multi-core processors. Another major drawback of the existing multi-threaded compression solutions is the non-compliance to the single-threaded compression format. In this paper, we propose a set of novel and high-performance parallel compression and decompression schemes. We introduce novel designs for dynamic threading based parallel compression and random access point based parallel decompression. With our solution, we mitigate both the scaling issues on multi-core x86 CPUs and format compliance issues encountered in multi-threading the compression operations. Our test results demonstrate massive speedups by manyfolds and performance scaling never seen before on x86 CPUs especially AMD's "Zen"-based recent processors that come with very high core counts.**

***Index Terms*—AOCL, AOCL-Compression, CPU, EPYC, HPC, LLM, LZ4, openMP, Parallel Computing, PIGZ, PZSTD, Snappy, Zen, ZLIB, ZSTD**

I. Introduction

Applications are increasingly relying on lossless data compression methods to reduce their working memory space, disk utilization and network communication. Database [1], [2] systems both SQL and NoSQL heavily use data compression in their operations to achieve reduced disk/memory footprint and a higher query execution throughput. Data compression is also extensively used by File systems like HDF5 [3] and data streaming applications [4]. Data compression also plays an important role in compressing the time series data like seismic measurements [5] and vast amounts of data in the High Performance Computing (HPC) [6] applications. Very recently lossless data compression has been studied and applied on the Large Language Models (LLM) [7], [8] models for reducing the data movement and uplifting the inference throughput. The commonly used data compression methods are: lz4 [9], snappy [10], zlib [11], zstd [12], bzip2 [13], lzma [14], lz4hc [9]. Depending upon the use case, the applications can tradeoff between compression speed and ratio. Table I lists the major application domains/areas along with a few example applications, the compression methods used and the scope of multi-threaded compression in these application areas.



Domain	Applications	Compression Methods	Multithreading
Database	MySQL, Postgres, Mariadb, Clickhouse, Cassandra	lz4, snappy, zlib, zstd, bzip2, lzma, lz4hc	Yes
HPC	Seismic, CFD, Simulations, Visualization	lz4, snappy, zlib, zstd	Yes
Data streaming	Kafka, Teradata, Pinterest	lz4, snappy, zlib, zstd	Yes
File System	HDF5	lz4, zlib, zstd	Yes
LLMs, DNNs	GPT models, Llama, OPT, Phi	zstd, zlib and others	Yes

TABLE I: Application usecases for parallel compression.

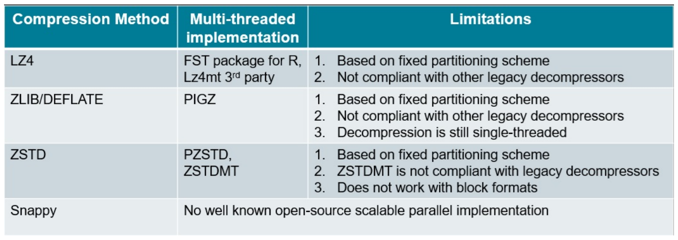
Recent CPUs come with a large number of core count that can be used to parallelize and speed up the compression tasks. AMD's 4th generation "Zen"-based CPUs offer 192 cores in a 2P (dual socket) configuration. Applications can make use of such advanced multi-core CPU systems to scale their performance. However, the availability of higher number of cores does not necessarily translate into performance and scalability for each and every computational job. Many a times, an efficient and scalable multi-threaded algorithm design is required to extract the benefits of the underlying hardware's multi-core parallelism.

II. Related Work

To reduce the compression runtime in the applications, multi-threading of the data compression is important to take advantage of the high core counts of the recent CPUs. However, it is not straightforward to parallelize data compression algorithms. Many steps are serial or lack significant overlap in the operations. Existing multi-threaded compression solutions partition the data without taking into account the underlying algorithmic steps. So, the thread utilization, and performance scaling are very poor on the multi-core processors. Existing implementations produce the multi-threaded compressed data in a form that does not comply with the single-threaded compression format and so can not be decompressed by a legacy single-threaded decompressor.

The prior arts and the existing implementations offering multi-threaded compression suffer from poor performance scalability on the multi-core CPUs. FST [15] package for R and lz4mt [16] project implement multi-threaded based on a fixed partitioning scheme that do not scale well. Parallel zlib compression is supported by pigz [17] which does not scale well on the many cores of the latest CPUs. The pigz library also does not support parallel decompression. Multi-threaded ZSTD implementations pzstd [18] and zstdmt [19] use fixed partitioning schemes and do not scale well. Snappy and other lossless data compression methods do not have any well-known scalable parallel implementations. Another work on a parallel compression method called ndzip [20] for compressing the scientific data improves upon the single-threaded throughput of the standard compression methods but does not scale well beyond a few number of threads.

Table II presents some of the known parallel compression implementations along with their limitations. One of the common limitations is the fixed data partitioning scheme used for multithreading the compression or decompression operations. The fixed partitioning of data does not consider the processing window of the underlying algorithm and fixes the number of threads without considering the optimal workload per thread. The other important limitation is related to the decompression operation where either the parallel version is not supported, or multi-threaded decompressor is not format compliant to single-threaded compression. The parallel compressed output from these implementations uses additional metadata that deviates from the single-threaded compressed output and so can not be decompressed by the legacy single-threaded decompressors.



Compression Method	Multi-threaded implementation	Limitations
LZ4	FST package for R, Lz4mt 3rd party	1. Based on fixed partitioning scheme 2. Not compliant with other legacy decompressors
ZLIB/DEFLATE	PIGZ	1. Based on fixed partitioning scheme 2. Not compliant with other legacy decompressors 3. Decompression is still single-threaded
ZSTD	PZSTD, ZSTDMT	1. Based on fixed partitioning scheme 2. ZSTDMT is not compliant with legacy decompressors 3. Does not work with block formats
Snappy	No well known open-source scalable parallel implementation	

TABLE II: Drawbacks of known solutions.

Fig. 1 presents the multi-threaded performance of the known parallel compression libraries pigz and pzstd benchmarked on AMD's Zen4 2P CPU system with 192 cores. Both pigz and pzstd output the execution times which are converted to speed (MB/s). Auto mode means the libraries are left to choose the number of threads. These multi-threaded compression libraries show very poor performance scaling. The primary reason for the poor performance scaling of these multi-threaded compression libraries is the inefficient data partitioning between the threads. So these implementations are not able to optimally exploit the multi-core and cache architectures. Another limitation with pigz is that it does not support parallel decompression operation.

III. Proposed design and format for parallel compression

In this paper, we propose a novel and efficient openMP based multi-threaded compression and decompression design and specification for achieving optimal performance scaling on multi-core processors. The proposed solution not only delivers a highly parallel and adaptive multi-threading solution but

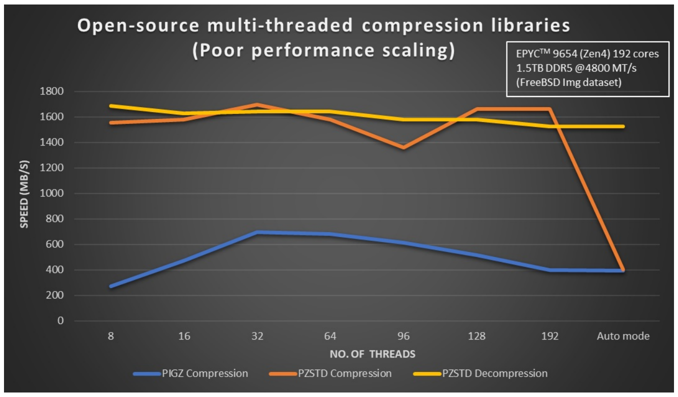
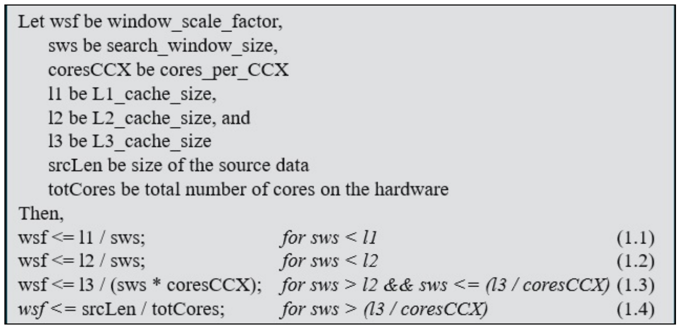


Fig. 1: Poor performance scaling of PIGZ and PZSTD.

is also compatible with the legacy single-threaded decompressors. Our work introduces (a) Dynamic threading and adaptive partitioning (DTRAP) based parallel compression, and (b) Random access point (RAP) specification based parallel decompression. DTRAP design helps in achieving higher performance and scaling. RAP helps in complying with the single-threaded compressed data format.

A. DTRAP based parallel compression

Our proposed design for parallel compression considers a number of factors for computing the adaptive partition size and for choosing the optimal number of threads dynamically. These factors include: (a) compression algorithm's dictionary search window size, (b) a newly introduced window scale factor, (c) total number of available threads, (d) input data size, (e) cache sizes on the CPU system. The window scale factor is crucial for computing the optimal partition size and can be derived (as per equations 1.1, 1.2, 1.3 and 1.4 as shown in Fig. 2) such that its product with the search window size is less than the respective cache size of the machine. Based on the search window size, the appropriate cache level (L1, L2, or L3) is used to calculate the window scale factor. The window scale factor can also be empirically selected such that the chunk size does not cross the highest cache size. Chunk size is the processing block size of each individual thread.



```
Let wsf be window_scale_factor,
     sws be search_window_size,
     coresCCX be cores_per_CCX
     l1 be L1_cache_size,
     l2 be L2_cache_size, and
     l3 be L3_cache_size
     srcLen be size of the source data
     totCores be total number of cores on the hardware
Then,
wsf <= l1 / sws;                 for sws < l1                                  (1.1)
wsf <= l2 / sws;                 for sws < l2                                  (1.2)
wsf <= l3 / (sws * coresCCX);    for sws > l2 && sws <= (l3 / coresCCX)        (1.3)
wsf <= srcLen / totCores;        for sws > (l3 / coresCCX)                     (1.4)
```

Fig. 2: Window scale factor derivation.

The overall algorithm to compute the adaptive DRAP partitions and dynamic DRAP thread count is provided as a flow-diagram in Fig. 3 Once number of DTRAP partitions is derived, the number of DTRAP threads are dynamically computed as a lower bound: MIN(total available threads, DRAP partitions). When input data size is large enough, then all cores can be used. But for smaller input sizes, using all cores does not result in an efficient performance – DTRAP solves this situation effectively by adaptively partitioning and dynamically setting the optimal number of threads for processing.

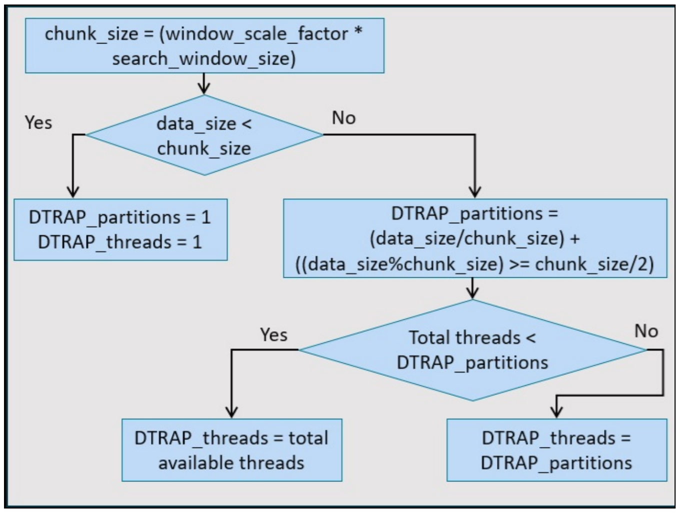
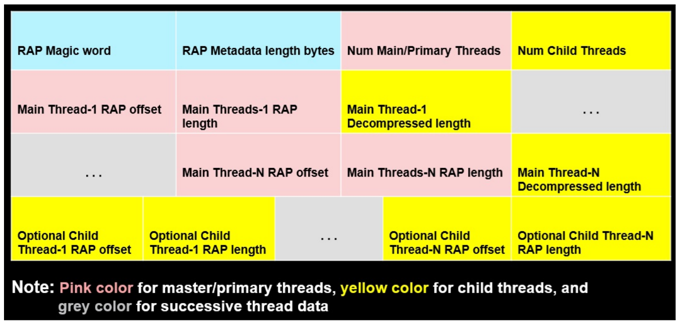

chunk_size = (window_scale_factor * search_window_size)
data_size < chunk_size
Yes
No
DTRAP_partitions = 1
DTRAP_threads = 1
DTRAP_partitions = (data_size/chunk_size) + ((data_size%chunk_size) >= chunk_size/2)
Total threads < DTRAP_partitions
Yes
No
DTRAP_threads = total available threads
DTRAP_threads = DTRAP_partitions

Fig. 3: DTRAP partitions and threads.

B. RAP specification based parallel decompression

Random access point (RAP) specification is a novel metadata specification proposed to implement parallel decompression without deviating from the single-threaded compressed data format. A random access point (RAP) metadata frame is added by the parallel compressor at the start of the multi-threaded compressed output data. The parallel decompressor can read this RAP metadata frame and know exactly the number of independent chunks and their locations in the stream/file. A single-threaded legacy decompressor can just skip the RAP metadata frame at the start of the stream/file and decompress it successfully sequentially. This allows the parallel decompressor to be compatible with legacy single-threaded compressor even for the block format. Fig. 4 presents the proposed RAP specification.



RAP Magic word	RAP Metadata length bytes	Num Main/Primary Threads	Num Child Threads
Main Thread-1 RAP offset	Main Threads-1 RAP length	Main Thread-1 Decompressed length	…
…	Main Thread-N RAP offset	Main Threads-N RAP length	Main Thread-N Decompressed length

Optional Child Thread-1 RAP offset	Optional Child Thread-1 RAP length	…	Optional Child Thread-N RAP offset	Optional Child Thread-N RAP length

Note: Pink color for master/primary threads, yellow color for child threads, and grey color for successive thread data

Fig. 4: RAP specification.

The proposed solution including the DTRAP design and RAP specification has been implemented and integrated into AOCL-Compression (AMD Optimizing CPU Libraries for Compression) [21]. AOCL-Compression [22] is a high-performance CPU library for lossless data compression that packs together various advanced optimizations related to the compression algorithm, data structures, and vectorization. AOCL-Compression debuted this proposed multi-threaded compression design and format in its latest release with version 4.2. Our proposed solution is implemented as a threading layer in the library which is called upon by the different compression methods. Applications (like Database, HPC, Data streaming, and File system) just need to link to the multi-threaded library and call the same single-threaded native APIs. The implementations of the native APIs are modified to integrate and invoke the functions of the threading layer for performing multi-threaded execution. The applications can also alternatively call the Unified API of the library and expect the threading layer to be invoked internally. Fig. 5 presents a high-level block-diagram of the AOCL-Compression library integrated with our proposed multi-threaded compression solution.

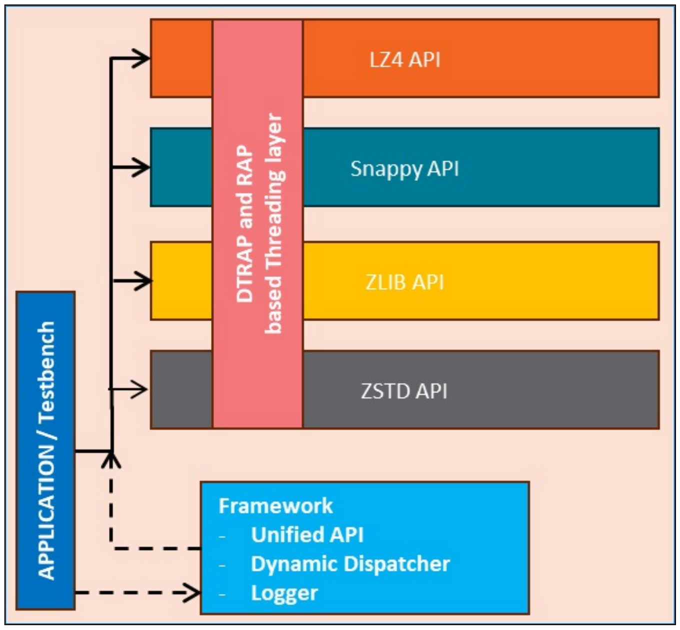


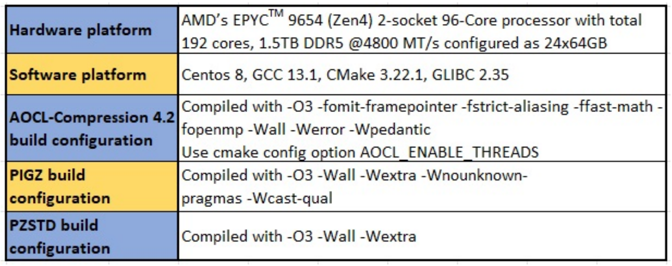
Fig. 5: Proposed parallel solution in AOCL-Compression.

IV. Results and Performance Discussion

A. Computational environments

The computational environment for all our experimental evaluations and benchmarking tests including the hardware and software platform, and the build configurations for the compression libraries are presented in Table III.

We performed multi-threaded performance benchmarking using FreeBSD [23] and Seismic [24] datasets comparing the performance uplifts and scalability of our solution with the reference PIGZ and PZSTD solutions. We measured the performance by setting the OMP_NUM_THREADS [25] to



Hardware platform	AMD's EPYC^{TM} 9654 (Zen4) 2-socket 96-Core processor with total 192 cores, 1.5TB DDR5 @4800 MT/s configured as 24x64GB
Software platform	Centos 8, GCC 13.1, CMake 3.22.1, GLIBC 2.35
AOCL-Compression 4.2 build configuration	Compiled with -O3 -fomit-framepointer -fstrict-aliasing -ffast-math -fopenmp -Wall -Werror -Wpedantic Use cmake config option AOCL_ENABLE_THREADS
PIGZ build configuration	Compiled with -O3 -Wall -Wextra -Wnounknown-pragmas -Wcast-qual
PZSTD build configuration	Compiled with -O3 -Wall -Wextra

TABLE III: Computational platform and configurations.

different values (1, 8, 16, 32, 64, 96, 128, 192) as well as in auto-mode (where library can dynamically decide how many threads to use) without explicitly setting any thread count.

B. Results

Our test results for FreeBSD image file as presented in Fig. 6 and Fig. 7 show strong performance uplift and scaling for both compression and decompression operations. The performance graphs are based on the geomean of performance of different levels of the compression methods. The proposed solution beats open-source parallel implementations for PIGZ and PZSTD by more than 13x in compression speed and more than 8x in decompression speed. At higher thread counts, the simpler compression methods like lz4 and snappy becomes memory bandwidth constrained. The results also show that the dynamic threading and adaptive partitioning performs the best as indicated by the auto-mode.

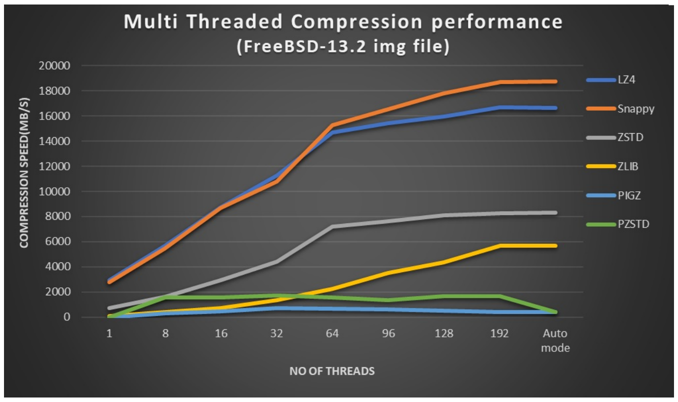


Fig. 6: AOCL versus Reference compression speedup.

Fig. 8 and Fig. 9 show the multi-threaded benchmark results for time series data of a Seismic (Oil & Gas) application. The test results are prepared for specific levels of the compression methods. We observe very good scaling up to 128 threads for parallel compression and 96 threads for parallel decompression, beyond which it gets limited by the memory bandwidth. The auto-mode gets us the optimal performance

C. Performance discussion

The dynamic threading and adaptive partitioning scheme based on the search window size and window scale factor are able to slice the input data into optimal chunk size thereby

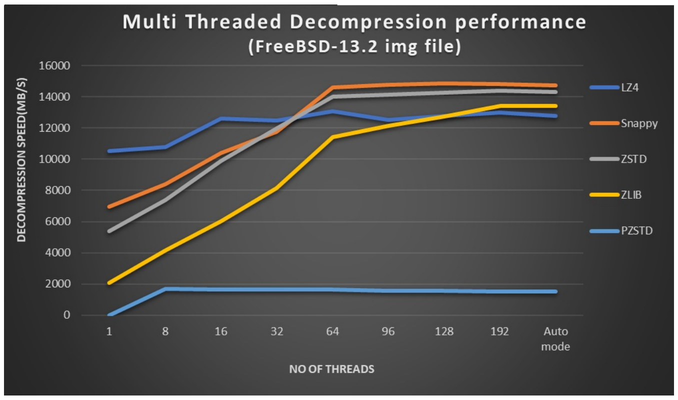


Fig. 7: AOCL versus Reference decompression speedup.

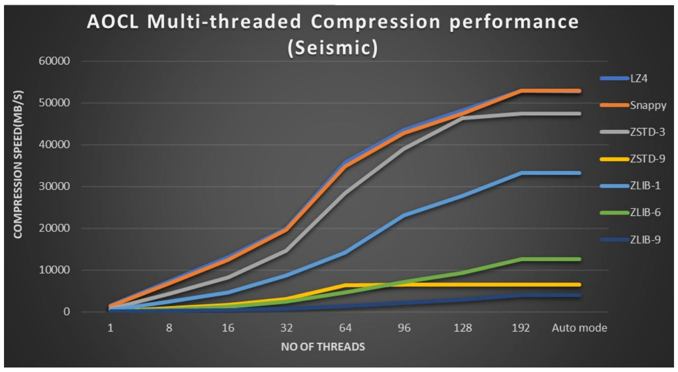


Fig. 8: AOCL compression speedup (levels).

offering such a high performance throughput and scaling. Due to the highly efficient and scalable multi-threaded data compression design, our solution is atleast 8x ahead of the other parallel open-source implementations in overall speedup. The degradation in compression ratio due to parallel processing is within 2%. This work offers a huge potential to uplift the performance of many applications related to Database, HPC, Data streaming, File system, and others. Our solution not only benefits the performance and scaling of applications on AMD's HW+SW stack but can also be applied to any x86 CPUs in general. The proposed RAP metadata format has the potential to be adopted as an industry-wide specification for parallel data compression.

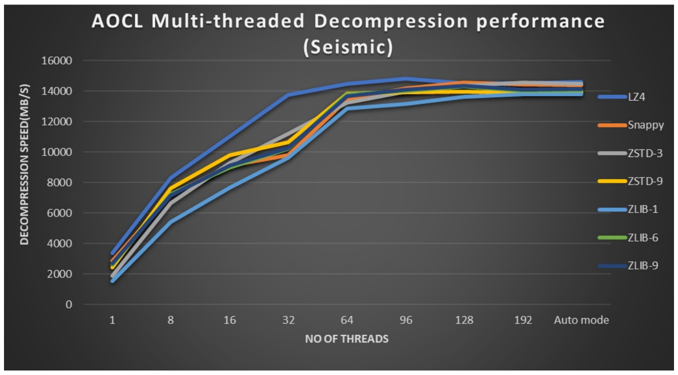


Fig. 9: AOCL decompression speedup (levels).

V. Conclusion

In this paper, we have proposed a novel and high-performance multi-threaded data compression solution. We have discussed the proposed parallel compression design that is based on a dynamic threading and adaptive partitioning (DTRAP) based approach. We have introduced a random access point (RAP) specification based highly parallel decompression approach. Our solution is highly scalable on multi-core processors as well as format compliant with single-threaded decompressors. Our benchmark tests demonstrated massive performance speedups and achieved very high scalability with the proposed solution on the latest AMD's Zen4 based processors. The performance benchmark results showed the proposed solution beating the open-source parallel implementations by more than 13X in compression speed and more than 8X in decompression speed. The proposed solution is already implemented for lz4, snappy, zlib and zstd in the AOCL-Compression library. In the future, we want to extend this implementation support to other compression methods like bzip2, lzma and lz4hc.

References

[1] M. M. Rovnyagin, V. K. Kozlov, R. A. Mitenkov, A. D. Gukov and A. A. Yakovlev, "Caching and Storage Optimizations for Big Data Streaming Systems," 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), St. Petersburg and Moscow, Russia, 2020, pp. 468-471

[2] G. Graefe and L. D. Shapiro, "Data compression and database performance," [Proceedings] 1991 Symposium on Applied Computing, Kansas City, MO, USA, 1991, pp. 22-27

[3] Andrew Collette. "Chapter 4. How Chunking and Compression Can Help You". Python and HDF5. [online]. Available: https://www.oreilly.com/library/view/python-and-hdf5/9781491944981/ch04.html

[4] "Kafka". Documentation. [online]. Available: https://kafka.apache.org/documentation/

[5] C. V. Peterson and C. R. Hutt, "Lossless compression of seismic data," in Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems & Computers, vol.2, pp. 712-716, Pacific Grove, CA, USA, 1992

[6] S. Li, N. Marsaglia, C. Garth, J. Woodring, J. Clyne, H. Childs, "Data reduction techniques for simulation, visualization and data analysis," in Computer Graphics Forum, Wiley Online Library, Vol. 37, pp. 422–447, 2018

[7] A. Lascorz et al. Atalanta: A bit is worth a "thousand" tensor values., in ASPLOS, page 85–102, 2024

[8] Y. Mao et al. "On the compressibility of quantized large language models." in arXiv:2403.01384, 2024

[9] "LZ4 - Extremely fast compression." LZ4 github project. [Online]. Available: https://github.com/lz4/lz4

[10] "Snappy." Snappy github project. [Online]. Available: https://github.com/google/snappy

[11] "DEFLATE compressed data format specification v1.3." RFC 1951. [online]. Available: https://tools.ietf.org/html/rfc1951

[12] "zstd." ZSTD github project. [Online]. Available: https://github.com/facebook/zstd

[13] "BZIP2." BZIP2 github project. [Online]. Available: https://github.com/libarchive/bzip2

[14] "lzma." LZMA SDK project. [Online]. Available: https://7-zip.org/sdk.html

[15] "fst". Lightning Fast Serialization of Data Frames for R github project. [Online]. Available: https://github.com/fstpackage/fst

[16] "lz4mt". Platform independent, multi-threading implementation of lz4 stream in C++11 github project. [Online]. Available: https://github.com/t-mat/lz4mt

[17] "pigz". A parallel implementation of gzip for modern multi-processor, multi-core machines. [Online]. Available: https://zlib.net/pigz/

[18] "pzstd". Parallel Zstandard (PZstandard) github project. [Online]. Available: https://github.com/facebook/zstd/tree/dev/contrib/pzstd

[19] "zstdmt". Multithreading Library for Brotli, Lizard, LZ4, LZ5, Snappy and Zstandard github project. [Online], Available: https://github.com/mcmilk/zstdmt

[20] F. Knorr, P. Thoman and T. Fahringer, "ndzip: A High-Throughput Parallel Lossless Compressor for Scientific Data," 2021 Data Compression Conference (DCC), Snowbird, UT, USA, 2021, pp. 103-112

[21] "aocl-compression." AOCL-Compression github project. [Online]. Available: https://github.com/amd/aocl-compression

[22] S. B. Raut, "AOCL-Compression - A High Performance Optimized Lossless Data Compression Library," 2023 IEEE High Performance Extreme Computing Conference (HPEC), Boston, MA, USA, 2023, pp. 1-7

[23] "FreeBSD 13.2." FTP release ISO-IMAGE 13.2. [Online]. Available: https://download.freebsd.org/ftp/releases/ISO-IMAGES/13.2/

[24] "Silesia compression corpus." Silesia Corpus Home Page. [Online]. Available: https://sun.aei.polsl.pl//~sdeor/index.php?page=silesia

[25] "OpenMP." OpenMP Specification website. [Online]. Available: https://www.openmp.org/specifications/