2023 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-0860-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPEC58863.2023.10363590

AOCL-Compression – A high performance optimized lossless data compression library

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

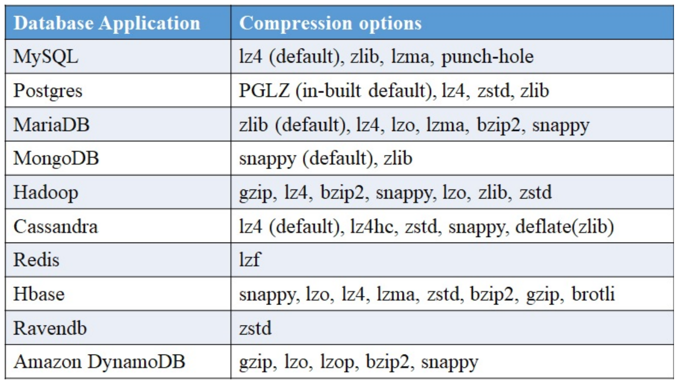
***Abstract*—Data compression is the process of encoding (or compressing) information using fewer bits than originally present in the data stream or signal. Depending upon whether this process is invertible or not, data compression can be lossless or lossy. While there exists various popular implementations of different lossless data compression methods, they are unable to completely meet the performance requirements demanded by the ever-increasing data usage of the applications. In this paper, we present a comparative survey of different lossless compression methods and introduce a high performance compression library called AOCL-Compression optimized for x86 architecture in general and AMD's "Zen"-based processors in particular. This library supports LZ4, LZ4HC, ZLIB, ZSTD, LZMA, BZIP2, and Snappy based compression methods. This paper discusses the design features of the new library framework and the algorithmic optimizations implemented for the different compression methods. Results highlighting the performance benefits of this new library implementation over the reference implementations of the respective compression methods are also presented.**

***Index Terms*—AOCL, AOCL-Compression, BWT, BZIP2, Compression, Database systems, Decompression, DEFLATE, Dictionary coding, Entropy coding, FSE, Hash chain, Hash table, Huffman coding, Lossless data compression, LZ4, LZ4HC, LZ77, LZ78, LZMA, MTF, NoSQL, RLE, Snappy, SQL, ZLIB, ZSTD**

I. Introduction

With the exponential growth in digital data, applications are faced with serious challenges related to the transmission of large data over the network or their storage by database systems. To solve and improve database latencies and network bandwidth requirements, data compression is very much essential for all such applications. Data compression reduces the number of bits needed to store or transmit the original data by exploiting data redundancy and similarities in the input data. When the decompression can reconstruct the original data from the compressed version, data compression is called lossless. When the decompression process can not get back the original data from the compressed version but only an approximation of it, data compression is called lossy. Such an approximate reconstruction of the original data is useful in achieving extremely high compression ratios.

Distributed and cloud database systems have emerged as highly scalable, fault tolerant, more manageable, and flexible storage systems for applications working with big data [1]. A cloud database system [2], [3] can be based on SQL or NoSQL data models. Lossless data compression forms an integral step in the query execution, backup, and restore operations related to these distributed and cloud database systems. Fig. 1 presents a list of popular database systems and the supported compression methods used by them.



Database Application	Compression options
MySQL	lz4 (default), zlib, lzma, punch-hole
Postgres	PGLZ (in-built default), lz4, zstd, zlib
MariaDB	zlib (default), lz4, lzo, lzma, bzip2, snappy
MongoDB	snappy (default), zlib
Hadoop	gzip, lz4, bzip2, snappy, lzo, zlib, zstd
Cassandra	lz4 (default), lz4hc, zstd, snappy, deflate(zlib)
Redis	lzf
Hbase	snappy, lzo, lz4, lzma, zstd, bzip2, gzip, brotli
Ravendb	zstd
Amazon DynamoDB	gzip, lzo, lzop, bzip2, snappy

Fig. 1: List of popular database systems.

The most widely used open-source lossless compression methods in the database applications are LZ4 [4], [5], LZ4HC [5], ZLIB [6], ZSTD [7], LZMA [8], BZIP2 [9], and Snappy [10]. These compression methods use either dictionary coding or entropy coding or a combination of both. Standard reference implementations of these methods have certain performance limitations. LZ4 and Snappy are based on LZ77 [12] dictionary based method and both make use of a fixed hash table for quick references into the search window. Match finding process involving memory accesses for byte comparisons is the major performance bottleneck in these two compression methods. ZLIB is based on the DEFLATE [11] compression format that uses the LZ77 dictionary search followed by Huffman coding [14]. It also uses checksum methods like ADLER32 and CRC32. The longest match finding function in ZLIB that evaluates a multitude of possible matches is constrained by memory latencies and comparison operations. In the case of LZMA, the matches are found by the LZ77 method, and the resulting symbols are encoded using context-based adaptive binary range coder [16]. As LZMA operates on large dictionaries and match offsets, the performance is limited by cache collisions and memory latencies. ZSTD compression also uses LZ77 dictionary search but adopts finite state entropy (FSE) coding for match symbol encoding. Match finder is the main operation affecting the performance due to its high memory latencies. Unlike other methods, BZIP2 uses the

Burrows Wheeler transform [15] to rearrange the string which makes it suitable for run level encoding (RLE) and move-to-front (MTF) transform. Byte reading and copying as well as MTF decoding are bottlenecked by sequential data operations.

Considering the performance limitations of the current state-of-art implementations of LZ4, LZ4HC, ZLIB (DEFLATE), ZSTD, LZMA, BZIP2, and Snappy, we present a new high performance optimized library implementation of these methods called AOCL-Compression [17], [18]. Our work adds various algorithmic optimizations to improve operations such as match finding, hash chaining, and match skipping. We also present strategies to perform SIMD vectorization of multi-byte operations. The paper is organized in the following sections. Section II discusses the performance limitations of the existing reference implementations of the various lossless data compression methods. Section III introduces the design and features of AOCL-Compression. Section IV focuses on the algorithmic improvements and optimizations implemented in AOCL-Compression to overcome the known performance limitations. Section V presents the test results and performance benefits of this new library framework. Section VI presents the conclusion of this work.

II. Lossless compression methods – Performance limitations of the existing implementations

There are three broad categories of lossless compression methods: dictionary, entropy, and combinational. Dictionary based methods work by finding a match for the current group of bytes in the dictionary formed by the previously seen group of bytes and referring the current data bytes by such offsets in the dictionary. LZ77 and LZ78 [13] are the two major classes of dictionary based compression methods. In the case of entropy based methods, the number of bits used to encode symbols depend upon their probabilities. Combinational methods are based on a combination of two or more methods for compression. In this section, we discuss the performance bottlenecks of the popular lossless compression methods as depicted in Fig. 2.

A. *LZ4 and its high compression variant LZ4HC*

LZ4 is an extremely fast compression method based on the LZ77 algorithm and uses a hash table to keep the latest match candidate. The compression process is heavily based on match finding comparison operations followed by finding of the match length and copying of literals. The computations are primarily memory intensive, and the standard techniques for byte matching and memory copy as used in LZ4 implementation [5] do not help achieve the best possible performance.

LZ4HC [5] is a high compression variant of LZ4. Compression is improved by using a longer dictionary buffer having 32K entries for hash table and 64K entries for chain table. The explicit hash chain table is used to maintain and evaluate the multiple match candidates. The LZ4HC method is memory bound and involves a lot of byte comparisons. Searching through the chain table for longest match candidate also takes a considerable time.

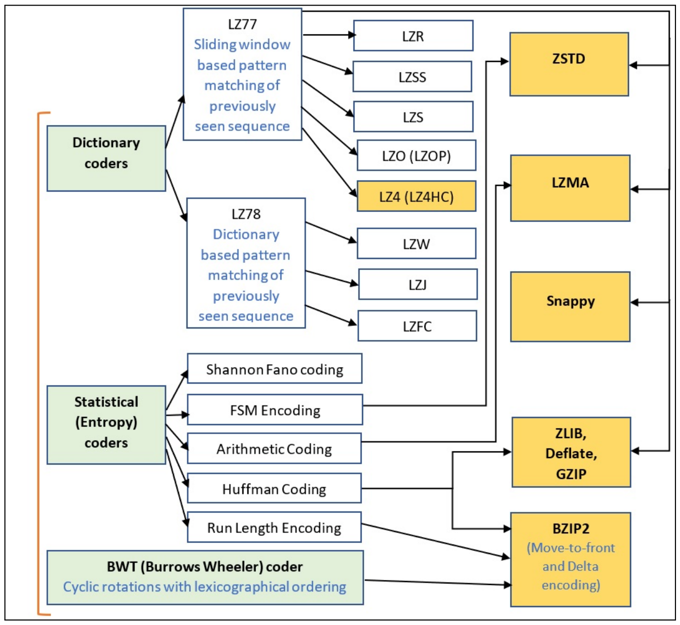

Dictionary coders
LZ77 Sliding window based pattern matching of previously seen sequence
LZ78 Dictionary based pattern matching of previously seen sequence
LZR
LZSS
LZS
LZO (LZOP)
LZ4 (LZ4HC)
LZW
LZJ
LZFC
Statistical (Entropy) coders
Shannon Fano coding
FSM Encoding
Arithmetic Coding
Huffman Coding
Run Length Encoding
BWT (Burrows Wheeler) coder
Cyclic rotations with lexicographical ordering
ZSTD
LZMA
Snappy
ZLIB, Deflate, GZIP
BZIP2 (Move-to-front and Delta encoding)

Fig. 2: Lossless compression methods.

B. *Snappy*

Snappy [10] is a light-weight lossless compression algorithm that aims for very high speeds at reasonable compression. The compressor searches for matches by comparing a hash of the current bytes with previous occurrences of the same hash. A match skipping mechanism is used to efficiently process hard to compress data. There is a good scope for developing better heuristics for match skipping based on the input data being processed. As the compressed data format is simple, most of the decompression time is spent in copying previously decompressed bytes to the current output stream.

C. *ZLIB*

ZLIB [6] is an abstraction of the DEFLATE lossless data compression algorithm that can be used on any computer hardware and operating system. The ZLIB data format is portable across platforms and unlike LZW [19] compression it never expands the data. DEFLATE uses a combination of LZ77 and Prefix coding [20].

Finding previous occurrences of similar patterns takes most of the compression time. It uses a sliding window comprising a search buffer and lookahead buffer. The hash chain stores previous occurrences of byte patterns seen in the search buffer having the same hash index. Each node in the hash chain is checked and compared for finding a match for the current byte pattern, thereby consuming more time. An algorithm for finding the longest match is employed to achieve better compression by performing comparisons at every byte in the lookahead buffer at the cost of a huge increase in the runtime.

D. *ZSTD*

ZSTD [7] is one of the recent and most advanced lossless compression methods. It uses finite state entropy (FSE) coding for match symbol encoding after having found a match by using the LZ77 dictionary search. ZSTD supports up to 22 levels to fit the compression speed and compression ratio needs of different applications. There are multiple compressor strategies in ZSTD like fast, greedy, lazy, lazy2, opt, ultra and ultra2 that come into play based on the level and the input source length. Row based match finder is a match finding algorithm used by the lazy2 compressor for certain mid compression levels. The computations for finding the longest match among multiple match candidates involve memory fetches at random offsets and byte-wise comparisons that consume a lot of compression runtime.

E. LZMA

LZMA (Lempel Ziv Markov Algorithm) [8] is a lossless compression algorithm that provides a high degree of compression. Compression is improved over LZ77 by using a longer dictionary buffer (up to 4 GB), optimal parsing, shorter codes for recently repeated matches, and range coding.

Finding matches within a large dictionary buffer takes up the bulk of the compression time and makes this process strongly memory bound. The primary hash table is combined with a secondary hash chain table to handle hash collision resolution. The hash chaining is typically implemented using linked-lists or binary search trees which are not cache-efficient. The search involving the matching of individual bytes in the input stream with potential candidates takes a significant time.

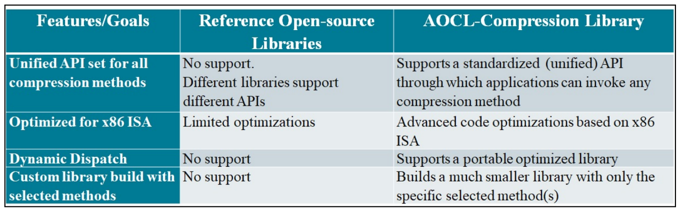
F. BZIP2

BZIP2 [9] compresses files using the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors and approaches the performance of the PPM (Prediction by Partial Matching) family of statistical compressors.

The compression process involves comparing the lexicographical order of cyclic rotations of the input string. This results in multiple memory accesses and comparisons consuming a lot of time. The decompression process performs byte-wise read operations into a 4-byte buffer while decoding the MTF values. These multiple byte-wise read operations create an increased number of issued memory load requests.

III. AOCL-Compression – Design and Features

Applications like distributed or cloud database systems usually support multiple compression methods and invoke the appropriate method based on factors like compression speed, decompression speed, and/or compression ratio. All these compression methods have their own API interface sets since there is no standard API specification. This is a major drawback for the applications as they need to modify their source code to integrate any compression method. A unified and standardized API interface for the compression methods is one of the key goals for designing a suitable software framework. AOCL-Compression library supports such a unified API set along with various other features and goals including performance, portability, and support for low memory (embedded) devices. Fig. 3 depicts a summary of important design goals of AOCL-Compression library as a software framework of various lossless compression methods (refer the colored blocks in Fig. 2) tuned and optimized for AMD "Zen"-based processors.



Features/Goals	Reference Open-source Libraries	AOCL-Compression Library
Unified API set for all compression methods	No support. Different libraries support different APIs	Supports a standardized (unified) API through which applications can invoke any compression method
Optimized for x86 ISA	Limited optimizations	Advanced code optimizations based on x86 ISA
Dynamic Dispatch	No support	Supports a portable optimized library
Custom library build with selected methods	No support	Builds a much smaller library with only the specific selected method(s)

Fig. 3: Design goals of AOCL-Compression library.

A. AOCL-Compression – Architecture

The overall architecture of AOCL-Compression library as depicted in Fig. 4 consists of a software framework of various compression methods including LZ4, LZ4HC, ZLIB (DEFLATE), ZSTD, LZMA, BZIP2, and Snappy. The library is developed in C and supports both Unix and Windows based systems. GCC and AOCC/Clang are the supported compilers for building the library using CMake build system.

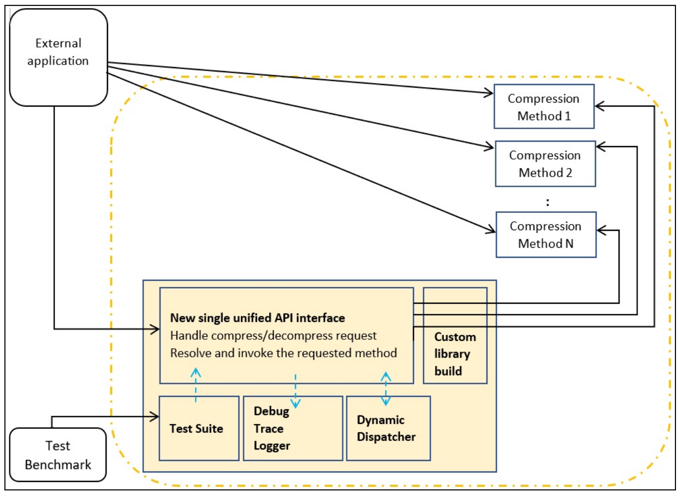


Fig. 4: Architecture of the AOCL-Compression library.

The core library framework sets up functionalities like dynamic dispatcher, logger (debug, trace, log) and others at startup, and manages the input requests from an application redirecting them appropriately to the underlying native APIs of the requested compression method. A new compression method can be easily added to the AOCL-Compression framework by simply adding the method's interface to the API layer of the framework.

B. AOCL-Compression – Features and Benefits

The AOCL-Compression library supports a list of distinguishing and useful features that establish clear-cut benefits of having such a software framework.

a) The framework provides a new standardized API interface set in addition to the respective native APIs of all the supported compression methods. Applications like cloud database systems can integrate and invoke AOCL-Compression library with minimalist changes.

b) Supports custom builds with just one or a few selected compression methods while excluding others, resulting in a smaller library with a lower code footprint.

c) The framework supports the detection of a processor's SIMD features and choosing the appropriate optimized code path at runtime with the help of a dynamic dispatcher, resulting in a portable and optimized library.

d) Provides a test suite to execute test inputs in various user modes, verify the results and measure compression/decompression speed, time, and ratio. The test suite can also benchmark Intel® IPP's [21] compression methods. It also supports GTest based unit testing of the compression methods as well as CTest based functional and performance test execution.

e) Supports a logger mechanism to print a verbose output in various modes such as debug, error, info, and trace.

f) AOCL-Compression implements various optimizations targeted for x86 architecture in general and AMD's "Zen"-based processors in particular. These optimizations can be turned on or off based on a user option.

g) The framework design is flexible for adding any future support for multi-threaded and distributed processing.

IV. Algorithmic improvements and performance optimizations in AOCL-Compression

AOCL-Compression achieves higher compression and decompression speeds and improved compression ratios with the help of the optimizations designed for the compression methods at various levels including algorithmic optimizations and SIMD vectorization.

A. *Efficient match skipping heuristic strategies*

Match skipping strategy is useful for input streams that are difficult to compress and do not contain frequent matching patterns. Methods like LZ4 and Snappy use very basic match skipping strategies to fix the step size used to scan through the look ahead buffer. In the case of LZ4, when the acceleration parameter is 1, the search step is incremented for every 64 bytes of unmatching bytes. Snappy uses the constant skip threshold of 32 unmatching bytes after which the search step is incremented by 1. These existing strategies in LZ4 and Snappy spend a lot of time unnecessarily looking for matches with smaller step sizes.

AOCL-Compression introduces new match skipping heuristics for LZ4 and Snappy. In the case of LZ4, two new match skipping strategies "long distance skip strategy 1" and "long distance strategy 2" are implemented that use an aggressive step size when it grows beyond a set threshold. The pseudocodes in Fig. 5 and Fig. 6 show how these new strategies operate in LZ4. Snappy applies a new match skipping strategy as depicted in Fig. 7 that grows the step size at a double rate and resets the step size to half its value in case a match is found. These new efficient heuristics greatly improve the compression speeds for the input data that have fewer matching patterns.

```
//searchNumBytes is initialized to 64
If "match found" and "step greater than a set Threshold"
then
  //Search step set to half the current size instead of resetting to 1 byte
  prevStep = (step / 2) – 1
step = (searchNumBytes++ >> 6) + prevStep;
```

Fig. 5: Pseudocode for LZ4 match skipping Strategy 1.

```
//searchNumBytes is initialized to 32
If "match found" and "step greater than a set Threshold"
then
  //Search step set to half the current size instead of resetting to 1 byte
  prevStep = (step / 2) – 1
step = (searchNumBytes++ >> 5) + prevStep;
```

Fig. 6: Pseudocode for LZ4 match skipping Strategy 2.

```
//searchNumBytes is initialized to 32
If "match found" and "step greater than a set Threshold"
then
  //Search step set to half the current size instead of resetting to 1 byte
  prevStep = (step / 2)
step = ((searchNumBytes >> 5) << 1) + prevStep;
```

Fig. 7: Pseudocode for Snappy match skipping Strategy.

B. *SIMD vectorization and multi-byte processing*

Vectorization in AOCL-Compression is implemented using CPU SIMD intrinsics based on the extent of data parallelism. ZLIB has decent vectorization opportunities in ALDER32 checksum computation, slide hash operation, and the string comparison to find the longest match. ALDER32 checksum function is vectorized using AVX intrinsics to operate on 32-bytes of input data per loop iteration. Sliding and updating of the hash chain is vectorized using AVX2 computations for processing 32-bytes per loop iteration. The longest match finding function is vectorized using AVX2 for doing multi-byte (32-byte) string comparisons.

The input and output copy operations in BZIP2 are vectorized by using 256-bit AVX2 load and store SIMD intrinsics, replacing the standard memcpy function call for lengths that are multiples of 64 bytes.

Data reading and processing functions are suitably modified to use multi-byte operations for speeding up the byte-wise memory requests and comparison operations. In LZ4HC, the backward search is optimized by processing 8 bytes at a time with XOR based comparison and __builtin_clzll (for counting leading zeroes) operation. This is faster than the original function which loads and compares the strings byte wise. BZIP2 compression and decompression reduces the number of load operations and possible cache splits by performing multi-byte operations that read input in 8-byte or 4-byte chunks instead of standard byte-wise operations. The match finding operations in LZMA compression are optimized by performing 4-byte data comparisons with XOR and __builtin_ctz (for counting trailing zeroes) operations. The row based match finder in ZSTD is optimized by performing 2-byte or 4-byte comparisons at the previously found match length while evaluating the next match candidates.

C. Early load and memory latency hiding strategies

The match finding operation in most of the compression methods is memory bound and bottlenecked at the front-end due to data access delays. An "early load" strategy as depicted in Fig. 8 is implemented in Snappy and LZ4 for efficiently reading bytes from the input stream. With this approach, a chunk of bytes is loaded from the input stream up front, and then compression is performed in a sliding window manner over this chunk of bytes hiding the subsequent load latencies.

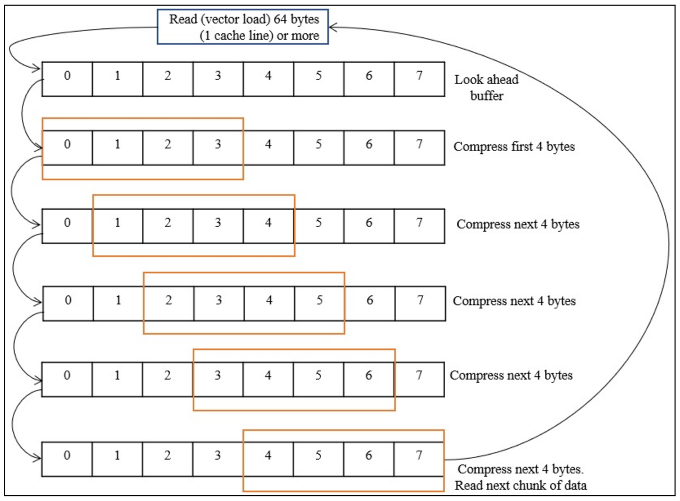
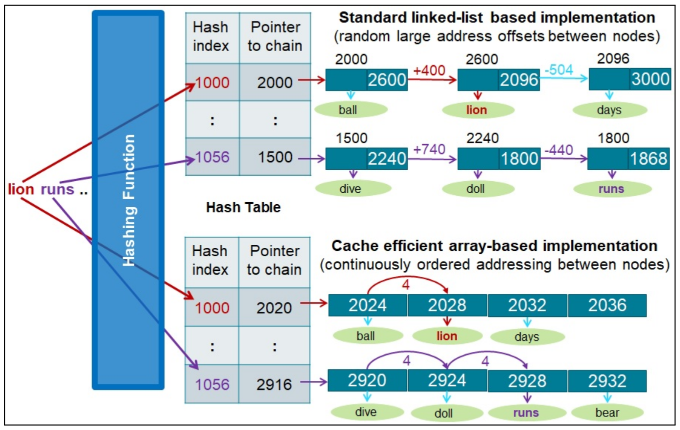


Fig. 8: Early load strategy for finding match.

D. Cache-efficient Hash Chain design

Hash Chaining is used to mitigate hash collision issues in methods that use LZ77 based dictionary search such as LZMA, LZ4HC, etc. The standard implementations make use of linked-lists for hash chaining. This however has performance drawbacks due to poor cache locality of the nodes and extra memory space needed for storing the links connecting the nodes. AOCL-Compression implements a cache-efficient array-based hash chain that keeps continuously positioned nodes in the chain.



Hashing Function
lion runs ..
Hash index
Pointer to chain
Hash Table
Standard linked-list based implementation
(random large address offsets between nodes)
Cache efficient array-based implementation
(continuously ordered addressing between nodes)
ball
lion
days
dive
doll
runs
bear

Fig. 9: Standard versus Proposed designs for hash chaining.

Fig. 9 compares the cache-efficient hash chaining with the standard implementation. A few operational details related to the proposed approach are as follows: a) Hash table holds the root nodes of the hash chains that store pointers to the data. b) Each root node and its associated hash chain are contained in a fixed sized circular buffer. c) For a given input string, the hash value is firstly looked up, then hash chain is searched for the exact match.

Access times of the match candidates in the hash chain are smaller due to their spatial locality.

E. Optimized Match finding strategies for ZLIB

ZLIB's standard strategy for finding the best (longest) match involves evaluating each node of the hash chain completely. AOCL-Compression implements an optimized match finding strategy by smartly skipping the shorter match candidates.

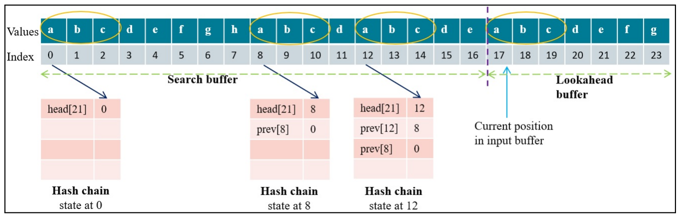


Fig. 10: Finding longest match with hash chain.

The new longest match finding strategy for ZLIB shown in Fig. 10 is implemented as follows: a) Assuming the hash value for a 3-byte string "abc" as 21, the associated hash chain is populated with three nodes corresponding to the match candidates at positions 0, 8, and 12 in the search buffer. b) To find the best match for the current string at position 17, the nodes (match candidates) of the hash chain are evaluated by checking for a byte match beyond the previously found match length. The first (head) node pointing to position 12 in the search buffer returns a match length of 5. The second node pointing to position 8 does not find a match beyond 5 bytes and so is skipped. The third node pointing to position 0 finds matching bytes after 5 bytes and is returned as the best match.

The standard length for hash keys in ZLIB is 3 bytes. However, AOCL-Compression adopts the strategy of using 4 byte hash keys for lower levels, that leads to evaluating less match candidates and speeding up the longest match computations. The impact on the compression ratio is not significant. For the fastest level, our implementation skips the hash chain entirely and uses only the head node.

F. Dynamic dispatcher for optimal execution

The vectorization in AOCL-Compression library is implemented using SSE2, AVX, and AVX2 based SIMD intrinsics. To make the library portable and optimal across different x86 processor models, a dynamic dispatcher system is implemented in AOCL-Compression. The setup API of the library initializes the dynamic dispatcher functionality at run-time in two steps. In the first step, it determines the processor support for SSE2, AVX, AVX2 and AVX512 automatically and sets the respective optimization level flags. In the second step, the appropriate function variants are registered through function pointers in each of the compression methods.

V. Results and Performance Discussion

A. Computational environments

The test system used for the experimental evaluation and benchmarking is configured with:- AMD's EPYC^{TM} 9654 (Zen4) 2-socket 96-Core processor with total 192 cores, 1.5TB DDR5 @4800 MT/s configured as 24x64GB, Centos 8, GCC 13.1. The AOCL-Compression library was configured and built in release mode along with its test bench executable. We performed the single-threaded performance benchmarking using the Silesia Corpus dataset [22] comparing the optimized compression methods with their respective reference open-source code versions (by using -o option of the test bench). For compression methods that support multiple levels, we selected a set of specific levels for the benchmark tests (LZ4HC – level 9, ZLIB – levels 1, 6, 9, LZMA – 1, 2, 9, BZIP2 – levels 1, 5, 9, ZSTD – level 9). To minimize run-to-run performance variations, we ran each compression method for 50 iterations for the specific level and selected the best speed score out of all such iterations. We used the Geometric Mean (GM) as the benchmarking method to derive the final speed scores for each of the compression methods.

B. Results

Fig. 11, and Fig. 12 present the comparative performance benchmark results for compression and decompression speedups between the AOCL-Compression library and open-source reference versions for LZ4, Snappy, ZLIB, LZMA, BZIP2, ZSTD and LZ4HC methods respectively. LZ4 is faster by 6% in compression speed and 5% in decompression speed over the reference method version. Snappy shows a large improvement of 11% in compression speed and 32% in decompression speed over the reference method version. ZLIB is massively accelerated by 45% in compression speed and 76% in decompression speed over the reference method version. LZMA is also impressively faster by 18% in compression speed and 10% in decompression speed over the reference method version. BZIP2 gets a significant uplift of 14% in compression speed and 27% in decompression speed over the reference method version. ZSTD and LZ4HC are faster by 7% and 10% respectively in compression speeds over the reference method version. Decompression speeds for ZSTD and LZ4HC are same as the reference versions.

C. Performance discussion

The results demonstrate the tremendous performance gains achieved by AOCL-Compression library across all the supported lossless data compression methods. The solid performance gains are made possible due to the algorithmic optimizations and vectorization implemented in the library. The dynamic dispatcher feature allows the library to execute on any target machine at a high performance. The unified API set of the library can be easily integrated to any application with minimal one-time efforts. This paves the way for evaluating the AOCL-Compression library with cloud database applications.

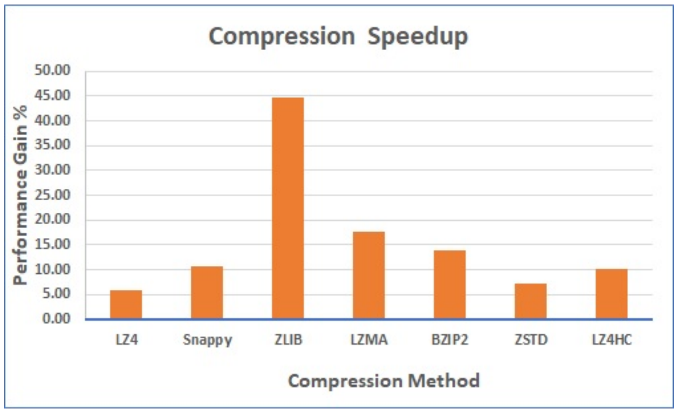


Fig. 11: AOCL versus Reference compression speedup.

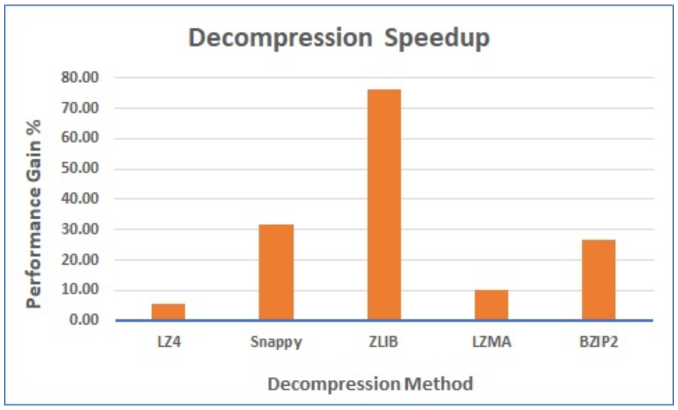


Fig. 12: AOCL versus Reference decompression speedup.

VI. Conclusion

In this paper, we have presented a comparative survey of different lossless compression methods and highlighted the performance bottlenecks in their standard open-source implementations. We have introduced a new high performance compression library called AOCL-Compression that is portable and optimized for x86 architectures in general and AMD processors in particular. The design goals and features of this new compression library supporting LZ4, LZ4HC, ZLIB (DEFLATE), ZSTD, LZMA, BZIP2, and Snappy compression methods are presented in this paper. We have also discussed the algorithmic optimizations and vectorization that are implemented for the different compression methods by this library. AOCL-Compression not only provides a standardized unified API set for all the compression methods, but also delivers huge performance speed ups. Results are shared for all the supported compression methods showing massive performance gains over the standard reference implementations. This new optimized library implementation can be easily integrated into cloud database applications, which will result in substantial performance uplift of these applications. Other applications related to file archiving, image compression, network routing, and genome pattern searching will also benefit from this library.

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] W. P. Cockshott, D. McGregor, N. Kotsis and J. Wilson, ”Data compression in database systems,” Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat. No.98EX130), Vienna, Austria, 1998, pp. 981-990

[4] Y. Collet. ”RealTime Data Compression.” LZ4 explained. [online]. Available: http://fastcompression.blogspot.com/2011/05/lz4-explained.html

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

[6] ”zlib.” ZLIB website. [Online]. Available: https://zlib.net

[7] ”zstd.” ZSTD github project. [Online]. Available: https://github.com/facebook/zstd

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

[9] ”BZIP2.” BZIP2 github project. [Online]. Available: https://github.com/libarchive/bzip2

[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] J. Ziv and A. Lempel, ”A universal algorithm for sequential data compression,” in IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, May 1977

[13] J. Ziv and A. Lempel, ”Compression of individual sequences via variable-rate coding,” in IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, September 1978

[14] D. A. Huffman, ”A Method for the Construction of Minimum-Redundancy Codes,” in Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, Sept. 1952

[15] M. Burrows and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” Tech. Rep. 124, Digital Systems Research Center, Palo Alto, 1994

[16] G. Nigel, N. Martin, ”Range encoding: An algorithm for removing redundancy from a digitized message,” Video & Data Recording Conference, Southampton, UK, July 24-27, 1979

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

[18] ”AOCL-compression Package.” AOCL-Compression Developer Resource Portal. [Online]. Available: https://www.amd.com/en/developer/aocl/compression.html

[19] “Lempel-Ziv-Welch.” LZW Wikipedia page. [Online], Available: https://en.wikipedia.org/wiki/Lempel-Ziv-Welch

[20] “Prefix code.” Prefix coding Wikipedia page. [Online], Available: https://en.wikipedia.org/wiki/Prefix_code

[21] “Intel Integrated Performance Primitives.” Intel’s IPP Home Page. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/ipp.html

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