Omni-Homomorphic Compression for Large Scientific Datasets

Alex Fallin
Department of Computer Science
Texas State University

Martin Burtscher
Department of Computer Science
Texas State University

***Abstract*—Scientific simulations can produce petabytes of data in a single run. Even when aggressively compressed, such data cannot be stored locally on a scientist's workstation. This paper introduces omni-homomorphic compression (oHC), a technique to generate extremely compressed versions of the data that are still useful and that can be processed locally without the need for decompression. Since the main data is kept separate, it can be preserved at any user-specified error bound and retrieved at will. Compared to other lossy and/or homomorphic compressors, oHC delivers not only orders of magnitude higher compression ratios but also some of the highest throughputs. For example, running on an RTX 4090 GPU with an error bound of 1E-2 on single-precision data, it provides an average of 270 and 253 GB/s throughput for compression and decompression, respectively. When extracting the arithmetic mean from a large dataset, oHC's extremely compressed version is 95,000 times faster than the fastest homomorphic compressor from the literature.**

***Index Terms*—Data compression, omni-homomorphic compression, floating-point data, CPU/GPU parallelization, big data**

I. Introduction

Top-of-the-line scientific codes and instruments produce enormous amounts of data. For instance, the Hardware/Hybrid Accelerated Cosmology Code (HACC) generates petabytes of data in a single simulation [8], and the LCLS-II [9] coherent light source can output up to 250 gigabytes per second. The devices used to produce such data tend to be very expensive. To maximize the return on these investments, it is paramount to *make the resulting data usable to as many people as possible*. However, storing and processing such large datasets is only possible on relatively few HPC systems. Almost nobody has the ability to download and process them locally.

Data compression can help by reducing storage requirements, minimizing transfer times, and improving network throughput. However, conventional lossless compression approaches deliver only roughly a factor of 2 in reduction on scientific floating-point data [4], and the leading lossy approaches yield up to a factor of 100, depending on the selected error bound [10]. Unfortunately, these factors are orders of magnitude too small for processing and studying a petabyte dataset on an individual researcher's workstation. As a remedy, we created oHC, which makes this possible by providing compression ratios of well over 10,000.

Even with such high compression ratios, it would be challenging to process a petabyte dataset if it had to be decompressed locally. Homomorphic compression, which not only allows computations to be performed directly on compressed data without the need for decompression but also to perform these computations faster than is possible on the uncompressed data, can help. Unfortunately, existing homomorphic compressors have major limitations in the operations they support and the compression ratios they yield.

To illustrate how homomorphic compression works, assume we are searching a large weather dataset for regions where the wind speed is at least 4 m/s. Assume further that the "original data" row in Table I lists the 16 wind speeds of a data chunk (in reality, chunks are much larger). To determine whether this chunk meets the condition, we must check each value until we find one that is at least 4 or reach the end of the chunk.

TABLE I: Simplified examples of non-homomorphic and homomorphic data compression (CR = compression ratio)

	Sequence of values (data chunk)	Size	CR
Original data	**2**, **2**, **2**, **2**, **2**, **6**, **6**, **1**, **2**, **2**, **2**, **2**, **6**, **6**, **1**, **4**	16	1.00
Huffman compressed	0, 1, 2, 6, 0, 2, 5, 5, 6	9	1.78
LZ77 compressed	0, 0, **2**, 1, 4, **6**, 1, 1, **1**, 7, 7, **4**	12	1.33

If we compress this sequence with Huffman encoding, we get 9 values, yielding a compression ratio of 1.78. Unfortunately, without decompression, this format does not allow us to determine whether the chunk contains values of at least 4. If we use Lempel-Ziv encoding (LZ77) instead, we get a compression ratio of only 1.33, but the compressed data makes it possible to check the condition. This is because LZ77 transforms the data into a sequence of triples, where the first element indicates how far back in the sequence we must go, the second element tells us how many values to copy starting from that location, and the third element is the next value in the sequence. Hence, to determine whether the LZ77-compressed data contains wind speeds of at least 4, we just have to check every third value (bolded in Table I) without any need for decompression. In other words, LZ77 is a homomorphic compression approach for our search operation whereas Huffman coding is not. However, LZ77 compresses our data less than Huffman coding, and LZ77 is not homomorphic for most other operations For instance, it is not homomorphic for computing the average wind speed.

Several homomorphic compressors exist, but they all have severe limitations. So far, the prior work has only resulted in approaches that 1) are tied to specific compression algorithms that may not yield high compression ratios or speeds, 2) do not support floating-point data or do not guarantee error bounds, and 3) support only a few fixed operations that may not be the ones the user wants. Our solution does not have these limitations. In fact, we call it *omni-homomorphic compression* (oHC) because it supports an unbounded number

of homomorphic operations and works with any compression algorithm, including algorithms that guarantee the error bound.

oHC makes this possible by combining two *independent* compression algorithms. The first algorithm is an existing compressor optimized to compress well and fast while guaranteeing the error bound. It operates on what we call the *main data*. The sole purpose of the second algorithm is to retain just enough information to allow the user-requested homomorphic operation(s). It saves this information in a very small *index file*. In our example, the index file might contain just the maximum wind speed of each data chunk. The user then processes this index file without decompressing it to, for instance, identify all chunks containing wind speeds of at least 4 and ultimately retrieves only those chunks from the large main data. Assuming the index file is 100,000 times smaller than the original data (which is the case if the chunk size is 100,000 values since the index only retains one value per chunk), oHC enables a computer with only a few dozen gigabytes of memory to work with a petabyte dataset.

This paper makes the following main contributions.

- It presents *omni-homomorphic compression*, which allows dataset providers to generate homomorphic index files for one or more operations of their choosing. Dataset users can then download and process these index files many orders of magnitude faster than the original data.
- It describes how oHC enables compression ratios of over 100,000 while supporting homomorphic operations.
- It compares oHC to state-of-the-art lossy compressors, including homomorphic compressors.

Our oHC CUDA and C++/OpenMP implementations are available in open source at https://github.com/burtscher/oHC.

The rest of this paper is organized as follows. Section II summarizes related work. Section III explains the oHC approach. Section IV describes the evaluation methodology. Section V discusses the results. Section VI provides a summary.

II. Related Work

A. Homomorphic Schemes

Compression that allows for operations to be performed on the compressed data, i.e., homomorphic compression (HC), is a nascent research area that arose from homomorphic encryption [6]. Both are transformations that allow computations on the transformed data without the need to first reverse the transformation. The main difference is that the former aims to decrease data size whereas the latter aims to obfuscate data.

Agarwal et al. [1] introduce PyBlaz, a homomorphic lossy compressor for arrays. It splits the array into blocks and runs a discrete cosine transform (DCT) on each block, which makes it possible for the data to be operated on by modifying the DCT coefficients. The compressor is lossy, but the error is reasonable and the throughput is similar to the state of the art.

Agarwal et al. [2] further developed the ideas in PyBlaz to bound the error while maintaining several homomorphic operations to create HoSZp. It supports derivative, divergence, Laplacian, mean, and variance computations on the compressed data. HoSZp's scalar operations are fully homomorphic, but the bivariate and reduction operations are only partially homomorphic, requiring the data to be partially decompressed before they can be applied. It only supports the NOA error bound. We compare oHC to HoSZp in Section V.

Guan et al. [7] introduce HOCO, an engine that compresses text homomorphically to allow for a variety of operations. It supports three schemes: run-length encoding (RLE), Lempel-Ziv-Welch (LZW), and "text analytics directly on compression" (TADOC). RLE is a widely used lossless compression algorithm that removes repeated symbols.

Overall, not much research on HC exists so far. The few papers we could find on HC that focus on scientific data support only a fixed set of operations, such as PyBlaz and HoSZp, or do not guarantee the error bound like PyBlaz. Also, none of them support GPU execution. Our oHC implementation guarantees the error bound and supports fully compatible CPU and GPU execution. In general, oHC is able to combine any well-compressing algorithm for the given data with a separate algorithm that enables fast homomorphic operations. This results in higher compression ratios and throughputs than conventional homomorphic approaches can provide. Moreover, it makes it possible to support a much wider range of homomorphic operations.

B. Lossy Compressors

This section describes the floating-point compressors with which we compare oHC. None of them are homomorphic.

There are several versions of SZ. They all use prediction in their compression pipeline. SZ3 [10] uses Lorenzo prediction and entropy coding plus lossless compression after the lossy stage. SZ3 is a CPU-only compressor. cuSZp [11] is a CUDA implementation that employs a different, more GPU-friendly algorithm. It performs Lorenzo prediction and quantization followed by multi-byte Huffman coding. FZ-GPU [12] is a specialized version of cuSZ that fuses multiple kernels together for better throughput. It splits the data into blocks and then quantizes and predicts the values in all nonzero blocks, which are then compressed by a fixed-length encoder.

MGARD [3] supports compression and decompression across CPUs and GPUs. It uses multigrid hierarchical refactoring to decompose and recompose the data to a specified accuracy via selective loading based on the hierarchy.

PFPL [5] is a CPU/GPU compatible guaranteed-error-bound lossy compressor. It operates in parallel on 16 kB independent chunks of the input. It first performs quantization based on the requested error bound. Values that cannot be quantized within the requested bound are stored losslessly. The quantized data is then fed into a bespoke lossless pipeline of transformations. We chose PFPL as the main compressor in oHC because it has high throughput, guarantees the error bound, supports CPU and GPU execution, and splits the data into small chunks, which are ideal for retrieving portions of a larger file.

III. Approach

There are two distinct types of oHC users:

- *Data providers* host the main data as well as one or more associated index files and provide read-only access to these items. They choose if and how the main data is compressed (e.g., lossily or losslessly). Moreover, they select which index files to make available and, therefore, which homomorphic operations are supported.
- *Data consumers* download the index file, run the homomorphic operation on it to determine the needed chunks of the main data, and then only download and decompress the chunks of interest before processing them locally.

As already mentioned, there are two types of oHC files:

- The *main data* is the very large dataset that is orders of magnitude too large to download in full and to store locally, even if it is compressed.
- The *index file* is an extremely lossy and compact representation of the main data. It is small enough to be downloaded quickly and stored locally. It is only useful to support a few selected homomorphic operations. Multiple index files can be provided to broaden the range of supported homomorphic operations.

A key innovation of oHC is that, by separating the index file from the main data, the index can be compressed at much higher ratios than would ordinarily be possible. This is because it only has to retain enough information to support the desired homomorphic operation. Consider, for instance, our wind-speed dataset with a chunk size of 10,000 values. In this example, the index file only stores a single value per chunk, namely the maximum wind speed, yielding a compression ratio of 10,000. This information is sufficient to homomorphically determine which chunks contain wind speeds of at least 4. The operation is homomorphic as no decompression is performed.

Another benefit of oHC is that the index file is independent of the main data in the sense that indices can be created later. This is important because it allows the data provider to include more index files at any point (or recreate a lost or corrupted index). Hence, support for additional homomorphic operations can be added at any time to accommodate data-consumer demand. Since the index files are typically just a fraction of a percent of the size of the main data (see Section V), the overhead of providing multiple indices is small.

A. oHC Operation

This subsection illustrates how oHC works on the implementation we evaluate in the result section. We use PFPL to compress the main data and record the minimum, average, and maximum value of each chunk in our index file. This type of index allows, for example, to homomorphically determine the chunks that contain values in a user-specified range, above or below a user-specified threshold, and sums of values that meet a user-specified condition. It is important to note that other compression algorithms can be used for the main data and other types of information can be stored in the index file.

We selected PFPL for compressing the main data for the following reasons. First, it is one of the only lossy compressors to offer a true point-wise error-bound guarantee (for the ABS, REL, and NOA error-bound types). Second, it is able to compress and decompress on the CPU and GPU producing bit-for-bit the same result. Third, it internally slices the data into 16 kB chunks, which it processes independently and in parallel. This is advantageous for our oHC implementation because it allows us to access the data at 16 kB granularity without the need to retrieve or decompress the entire dataset.

Data provider: oHC starts when the data provider has the main data ready and compressed, in our case with PFPL using whatever error bound is desired. At this point, the data provider can create a first index file by decompressing the main data, extracting the pertinent information from each chunk, and storing the result in the index file. These steps can be done on the fly (and in parallel), meaning the index is created as the main data is being decompressed so that the decompressed data does not need to be stored. Optionally, multiple indices can be generated at the same time. Once the index files are ready, they are made available with a description of their content and/or pieces of code to homomorphically extract information from them, such as generating a list of chunks that contain values above a user-provided threshold. The top half of Figure 1 illustrates the workflow of the data provider.

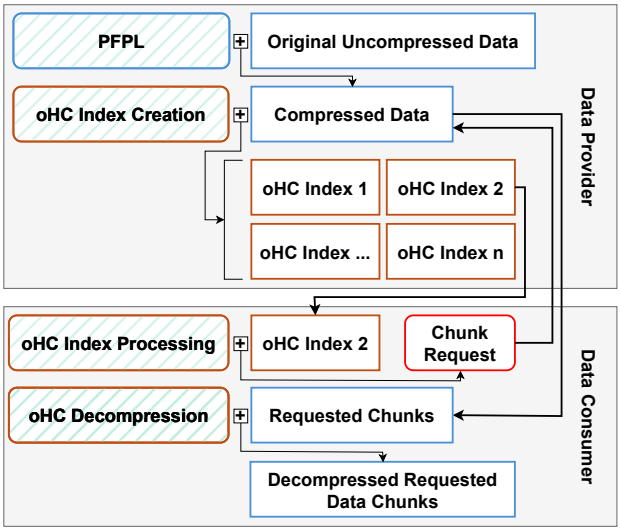

Fig. 1: Overview of the oHC workflow

Data consumer: Once an index file and the corresponding code snippets are available, data consumers can download them and run the supported homomorphic operations locally, which yields a list of data chunks that meet a given selection criterion. If the list is too long or too short, the criterion can be refined and the operation repeated. When the list is reasonable, the data consumer can download those chunks from the main data (plus "surrounding" chunks if needed), decompress them with oHC's version of PFPL, and process them locally. These steps can be iterated as needed with other criteria and/or other indices if different homomorphic operations or compound criteria are desired. The bottom half of Figure 1 illustrates the workflow of the data consumer.

Note that the achieved compression ratio is solely determined by the data-provider-selected oHC chunk size and independent of the values in the main data. For example, an oHC chunk size of 8192 words always yields a compression ratio twice that of an oHC chunk size of 4096 words. This provides several advantages. First, the compression ratio of the index file is known a priori and can be precisely controlled by the data provider. Second, the provider can generate multiple index files with the same content but for different chunk sizes (i.e., with different compression ratios), giving data consumers the flexibility to trade off coarser chunks for faster download and processing of the index file. Third, it decouples the compression of the index file from that of the main data, meaning the main data can be compressed at any level of fidelity (e.g., lossless) without affecting the compression ratio of the index file. These oHC features make it possible to preselect the size of the index file, thus guaranteeing that it will fit in a data consumer's system for local analysis.

IV. Experimental Methodology

Whereas our oHC approach is general, we only evaluate a specific implementation. In this implementation, we use PFPL as the compressor for the main data. We chose to record the minimum, average, and maximum value of each oHC chunk in the index file. These three pieces of information allow the data consumer to isolate and research specific regions of the main data that meet certain criteria (see Section III-A).

We compare oHC to the six state-of-the-art compressors described in Section II using various NOA error bounds [5]. Our system runs Fedora 37 and is based on an AMD Ryzen Threadripper 2950X CPU with 16 hyperthreaded cores. It has 64 GB of main memory. The GPU is an NVIDIA RTX 4090 with 24 GB of global memory. The GPU driver version is 525.85. The CPU codes were compiled using *gcc/g++* version 12.2.1 and the GPU codes with *nvcc* version 12.0.

We compiled the CPU codes using the build processes supplied by their respective authors. When not specified, we used the "-O3 -march=native" flags. Unless automatically determined, the thread count was set to the number of CPU cores as hyperthreading usually does not help. We compiled the GPU codes with the "-O3 -arch=sm_89" flags.

For all compressors, we measured the compression ratio as well as the execution time of the compression and decompression functions. For oHC, we separately measured the compression ratio of the index file as the compression ratio of the main data is simply that of PFPL. The oHC "compression" throughput is that of creating the index file. It consists of decompressing the main data and generating the index. In Section V-A, we report compression results for various chunk sizes and decompression results for retrieving between 1 MB and 30 MB of main data. In Section V-B, we use an index size of 32,768 words for oHC. In the decompression subsection, we trimmed all inputs to 30 MB to show the throughput of the compressors when retrieving just a portion of the main data.

We used the 5 single-precision suites listed in Table II as inputs for the compressors, a total of 66 files. These inputs stem from the SDRBench repository [13], which hosts scientific datasets from various domains for compression evaluation.

TABLE II: Information about the floating-point input suites

Name	Description	Files	Dimensions	Size (MB)
CESM-ATM	Climate	33	26 × 1800 × 3600	674
Hurricane Isabel	Weather Sim.	13	100 × 500 × 500	100
NYX	Cosmology	6	512 × 512 × 512	537
SCALE	Climate	12	98 × 1200 × 1200	564
QMCPACK	Quantum MC	2	33,120 × 69 × 69	631

We ran each experiment 9 times and collected the compression ratio, median compression throughput, and median decompression throughput. The plots report the geometric mean of the geometric mean of each input suite. For all compressors, the circular data point is for an error bound of 1E-2, the square is for 1E-3, and the triangle for 1E-4.

For compressors that support serial and parallel execution or CPU and GPU execution, we only show the fastest version if the compression ratio is the same between the versions. Otherwise, we show all versions. In the oHC-specific subsection, we show results for all 3 oHC versions.

V. Results

In this section, we investigate the performance of our implementation of omni-homomorphic compression and compare it to state-of-the-art compressors from the literature.

A. oHC Compression and Decompression Performance

Index Generation: Figure 2 shows the geometric-mean compression ratio and throughput for the three tested error bounds across varying chunk sizes. The chunk size reflects the number of 4096-word main data chunks that corresponds to one oHC chunk. Note that both y-axes are logarithmic. The reported throughputs are for the index generation from the already decompressed main data.

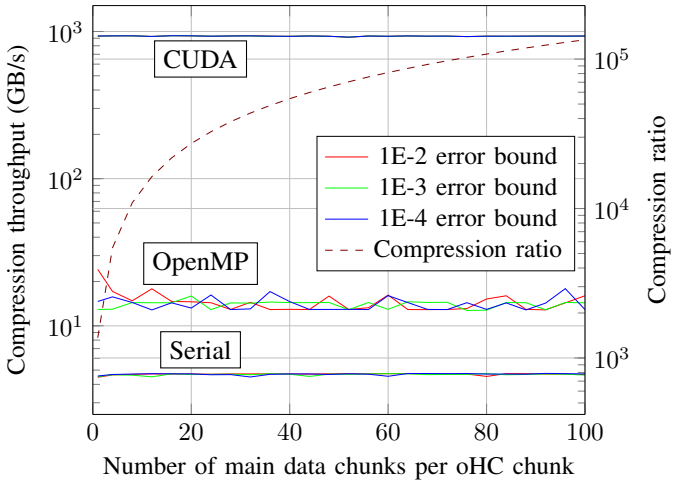

Fig. 2: Geometric-mean compression throughput and compression ratio for different chunk sizes and main-data error bounds

We only show a single curve for the compression ratio because the compression ratio of the index is independent

of the main data. As the oHC chunk size increases, the overall compression ratio increases proportionately. The first few chunk-size increments yield very large increases. Hence, we recommend using an oHC chunk size greater than 1 PFPL chunk. At the high end, the compression ratios achieved are orders of magnitude higher than what other compressors can supply. For example, at an oHC chunk size of 80, the compression ratio for all inputs is 107,980.

There is a trade off, however, between the oHC chunk size and the main data access granularity. As the chunk size and compression ratio increase, the minium amount of main data retrievable also increases. At a chunk size of 1, the main data can be accessed at a granularity of 16 kB. At a chunk size of 80, it must be accessed at a granularity of 1.25 MB.

The throughput is only marginally affected by the error bound of the main data. This is because the index is generated from the decompressed main data, and the decompressed main data at a looser error bound is the same size as the decompressed main data at a tighter error bound. Consequently, we see a nearly constant throughput for the index generation across all chunk sizes. On the GPU, the throughput reaches 931.8 GB/s on average, which approaches the memory-access bandwidth. This suggests that the index generation is memory bound and the bottleneck is reading the main data as we do not see an increase in throughput for higher index compression ratios, that is, when writing less index data.

Decompression: Figure 3 shows the decompression throughput for differing amounts of main data being retrieved. As the main data is typically much larger than the data consumer's local storage, we analyze the performance of oHC when decompressing a small number of retrieved chunks. We tested up to 30 megabytes of retrieved data as that is slightly smaller than our smallest tested input.

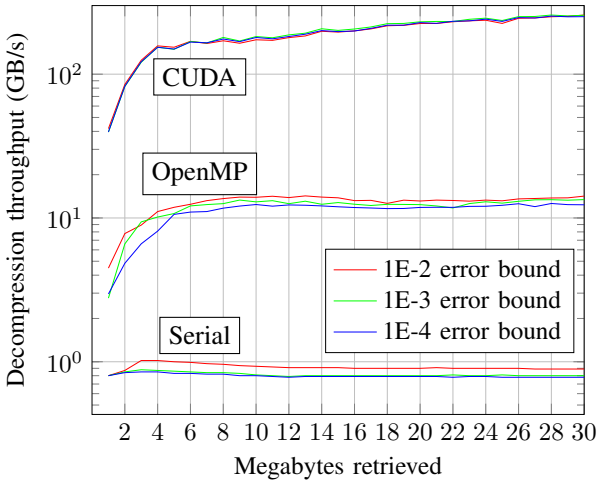

Fig. 3: Geometric-mean decompression throughput for different amounts of main data retrieved

The decompression throughput changes depending on the amount of data retrieved. The serial version reaches its throughput plateau at a few megabytes. The OpenMP version is parallel and, thus, reaches its plateau only above about 12 MB. The CUDA code is highly parallel and requires over 30 MB to reach its plateau. Nevertheless, even at its lowest throughput, it is an order of magnitude faster than the OpenMP version. At the highest measured throughput of 258 GB/s, the CUDA oHC decompressor is 18.2 times faster.

B. Performance Comparison

Compression: Figure 4 shows a scatter plot of the compression ratio versus throughput for three error bounds.

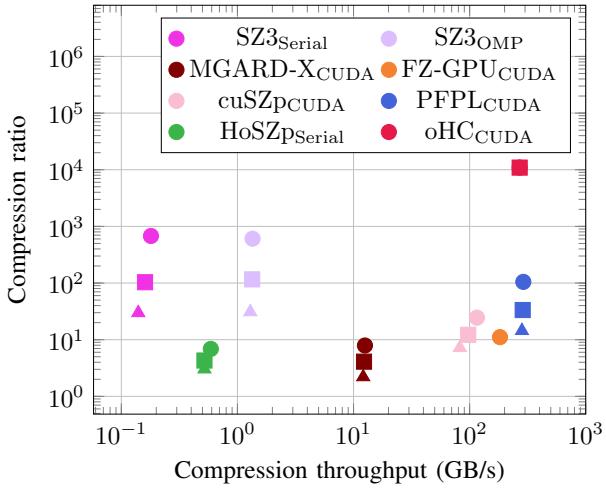

Fig. 4: Geometric-mean compression ratio and compression throughput with 3 NOA error bounds

As discussed, the compression ratio of oHC's index file can be set to any level. We selected a chunk size of 32,768 words, yielding compression ratios of around 10,000 (depending on how full the last chunk is). The oHC results in Figure 4 for all tested error bounds overlap. The compressors from the literature, however, yield lower compression ratios when using tighter error bounds.

In terms of throughput, oHC is orders of magnitude faster than both versions of SZ3, the next best compressing codes. HoSZp, the only other homomorphic compressor, produces relatively low compression ratios and throughputs. This is due to the drawbacks of homomorphism discussed in Section I. Since oHC uses the fast PFPL compressor on the main data, its throughput is close to that of the standalone version of PFPL and on par or higher than that of the other two GPU-based compressors while producing much higher compression ratios (see Section V-C). Moreover, oHC can be used to generate additional index files for other use cases without incurring slowdowns or decreases in compression ratio.

Decompression: Figure 5 shows a bar chart of the throughput for three error bounds when decompressing 30 MB of data. We trim all inputs to 30 MB to simulate a partial-retrieval scenario where the data consumer requests only a small portion of the main data from the provider.

All compressors have their performance hindered by the relatively small 30 MB file size. This is particularly true for some of the GPU-based compressors, which need more data to

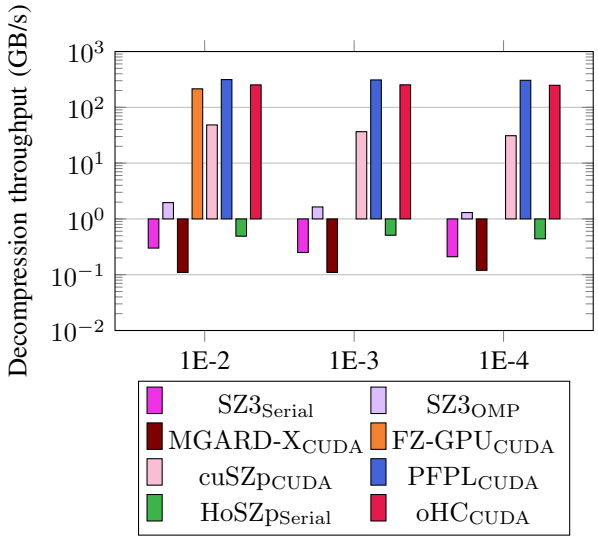

Fig. 5: Geometric-mean decompression throughput on 30 MB data with 3 NOA error bounds

fully load the hardware. Of the GPU-based compressors, FZ-GPU, PFPL, and oHC are the most resilient to the smaller file size, cuSZp is affected more but still maintains relatively high throughputs, and MGARD-X is severely impacted by the small file size, running slower than even the serial compressors. The CPU-based compressors are much less affected. This is likely because fully loading a CPU requires less data than a GPU.

Overall, on the tested error bounds, oHC outperforms even the best compressors in terms of compression ratio while being orders of magnitude faster. It delivers a GPU throughput on par with all but the fastest GPU compressors but produces compression ratios that are, in the worst case, 104 times higher. HoSZp, the other tested homomorphic compressor, yields relatively low throughputs and compression ratios that are 1586 times lower than oHC's due to the specialized compression algorithm used to allow for conventional homomorphism.

C. Matching oHC's Compression Ratios

The main purpose of oHC is to facilitate the analysis of petabyte datasets on a local machine. This section studies what error bound is required for each tested compressor to match the size of oHC's index file with 32,768-word chunks, which corresponds to a compression ratio of over 10,000× as our indices store three words per chunk. For simplicity, we performed this experiment on the CESM-ATM-UU file, an input that results in performance close to the average numbers reported in Section V-B. For each compressor, we ran a binary search to find the smallest error bound that yields a compression ratio in excess of 10,000×, if reachable.

Table III shows either the maximum achievable compression ratio or the minimally required error bound to deliver a compression ratio above 10,000×. It further lists the mean squared error (MSE), a quality metric. For reference, we also include PFPL with an error bound of 0.001.

The only tested compressors able to deliver compression ratios similar to those of oHC are the CPU-based SZ compressors. Of these compressors, SZ3's OpenMP version requires the lowest error bound of 0.02. This error bound means that all values in the input are quantized into 1 of 50 distinct values. Many of the other compressors require an error bound of 1.0. This is important because a NOA error bound of 1.0 quantizes all entries to the same value, resulting in a near total loss of usable data, which is reflected by the many orders of magnitude higher MSEs than the reference PFPL incurs.

TABLE III: NOA error bound needed to reach 10,000× compression or the maximum achievable compression ratio

Compressor	Compression Ratio	Error Bound	MSE
SZ3 Serial	12,565	**0.12**	90,026
SZ3 OpenMP	11,104	**0.02**	2,344
PFPL	**4,096**	1.00	695,001
FZ-GPU	**228**	1.00	695,001
MGARD-X	**32**	1.00	115,706
cuSZp	**128**	0.95	694,293
HoSZp	5,616,000	**1.01**	695,001
PFPL	59	0.001	42

PFPL, FZ-GPU, MGARD-X, and cuSZp all have a maximum compression ratio below 10,000× due to inefficiencies in their algorithms. Notably, these are the compressors that support GPU execution. Therefore, if high-speed operation is desired, a lower compression ceiling than the CPU-only compressors provide is likely unavoidable unless oHC is used.

These results demonstrate the two major problems with conventional compression when it comes to extremely large data. First, if local storage of the entire dataset is required, excessively large error bounds must be used, rendering the data mostly useless. Second, the currently available high-speed GPU compressors deliver significantly lower compression ratios than CPU-only compressors. oHC addresses both issues. First, it allows the main data to be compressed at any level of quality, including losslessly, since this data resides on the provider's servers and its fidelity does not affect the compression ratio of the index file. Second, oHC's compression ratio can be directly controlled by increasing or decreasing the chunk size to produce almost any desired compression ratio even when using fast GPU-based compressors.

D. Homomorphic Average Computation

Both oHC and HoSZp are able to homomorphically compute the average of all values in a dataset. In this section, we compare their geometric-mean performance across our test inputs. Table IV shows the runtime of HoSZp's average computation as well as the runtimes of all 3 versions of oHC's average computation for 3 error bounds. The chunk size of oHC is again 32,768 words.

There is a big difference between the runtimes of HoSZp and oHC when homomorphically computing the average. This is to be expected, however, because HoSZp requires partial decompression of the entire data file to perform its average calculation. In contrast, oHC is able to compute the average simply by averaging the chunk averages stored in the index file. This is very fast and only takes a few microseconds on our

TABLE IV: Geometric-mean runtime in seconds for homomorphically computing the average for 3 error bounds

	Error Bound		
Compressor	**1E-2**	**1E-3**	**1E-4**
HoSZp	0.476553	0.475822	0.501909
oHC Serial	0.000005	0.000005	0.000005
oHC OpenMP	0.000324	0.000256	0.000206
oHC CUDA	0.000008	0.000008	0.000008

inputs. In fact, it is so fast that our serial code outperforms the OpenMP and CUDA versions, both of which incur overheads in form of parallel reductions, thread synchronization, and kernel launches that cannot be amortized over such a short runtime. Based on the geometric-mean runtimes, the serial oHC average computation is over 95,000 times faster than the serial HoSZp average computation.

VI. Summary and Conclusion

We introduce omni-homomorphic compression (oHC), a technique that supports a large number of homomorphic operations and works with any compression algorithm. It is designed to be used both by data providers and data consumers to enable the analysis of huge scientific datasets that are many orders of magnitude too large to fit on a local system. We developed an implementation of oHC and evaluated it against 5 state-of-the-art lossy compression algorithms on 5 single-precision input suites from SDRBench, a total of 66 files. Our oHC code is about as fast as the fastest GPU compressors while delivering (tunable) compression ratios that are higher than those of even the best-performing CPU compressors. oHC outperforms HoSZp, another homomorphic compressor, by a factor 1586 in compression ratio and by a factor of 95,000 in runtime. We hope that oHC will help democratize access to very large scientific datasets and, thus, enable breakthroughs that were previously hampered by data size.

VII. Acknowledgment

This work is supported by DOE Award DE-SC0022223.

References

[1] Tripti Agarwal, Harvey Dam, Ponnuswamy Sadayappan, Ganesh Gopalakrishnan, Dorra Ben Khalifa, and Matthieu Martel. What Operations can be Performed Directly on Compressed Arrays, and with What Error? In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 254–262, New York, NY, USA, 2023. ACM.

[2] Tripti Agarwal, Sheng Di, Jiajun Huang, Yafan Huang, Ganesh Gopalakrishnan, Robert Underwood, Kai Zhao, Xin Liang, Guanpeng Li, and Franck Cappello. HoSZp: An Efficient Homomorphic Error-bounded Lossy Compressor for Scientific Data. *arXiv:2408.11971*, 2024.

[3] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science*, 19(5-6):65–76, 2018.

[4] Noushin Azami, Alex Fallin, and Martin Burtscher. Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 395–409, New York, NY, USA, 2025. Association for Computing Machinery.

[5] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds. In *2025 IEEE International Parallel and Distributed Processing Symposium*, pages 874–887, Los Alamitos, CA, USA, Jun 2025. IEEE Computer Society.

[6] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC'09, page 169–178, New York, NY, USA, 2009. ACM.

[7] Jiawei Guan, Feng Zhang, Siqi Ma, Kuangyu Chen, Yihua Hu, Yuxing Chen, Anqun Pan, and Xiaoyong Du. Homomorphic Compression: Making Text Processing on Compression Unlimited. *Proc. Manag. Data*, 1(4), 2023.

[8] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. HACC: Extreme scaling and performance across diverse architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2013.

[9] Linac Coherent Light Source (LCLS-II). https://lcls.slac.stanford.edu/, 2017. Online.

[10] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors. *IEEE Transactions on Big Data*, 2023.

[11] Jiannan Tian, Sheng Di, Xiaodong Yu, Cody Rivera, Kai Zhao, Sian Jin, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. Optimizing Error-Bounded Lossy Compression for Scientific Data on GPUs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 283–293, Los Alamitos, CA, USA, September 2021. IEEE Computer Society.

[12] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. FZ-GPU: A Fast and High-Ratio Lossy Compressor for Scientific Computing Applications on GPUs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'23, New York, NY, USA, 2023. ACM.

[13] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In *International Workshop on Big Data Reduction*, pages 2716–2724, 2020.