Analyzing an In-line Compression on the Matrix Matrix Multiplication Kernel

Steven Platt and Jon C. Calhoun
Holcombe Department of Electrical and Computer Engineering - Clemson University, Clemson, SC, USA
Email {platt9, jonccal}@clemson.edu

***Abstract*—High-performance computing (HPC) has enabled advancements in computation speed and resource cost by utilizing all available server resources and using parallelization for speedup. This computation scheme encourages simulation model development, massive data collection, and AI computation models, all which store and compute on massive amounts of data. Data compression has enhanced the performance of storing and transferring this HPC application data to enable acceleration, but the benefits of data compression can also be transferred to the active allocated memory used by the application. In-line compression is a compression method that keeps the application memory compressed in allocated memory, decompressing memory as its data is needed by the algorithm. The actively decompressed data size in allocated memory can be limited by grouping and compressing the data into blocks informed by the application's computational kernel's data access patterns and a selected compressor. This research explores factors such as block size and compressor choice on the runtime and memory usage of the Matrix Multiplication kernel. Matrix multiplication (MM) is a fundamental HPC algorithm that exemplifies the memory access patterns and use cases present in other HPC kernels. MM kernels provide a baseline for evaluating in-line compression due to MM's row-based data access patterns and the usage of several matrices to compute the resulting matrix. Trade-offs between memory size and the compression runtime necessitate tuned parameters for each in-line compression-enabled kernel. The results of this research demonstrate essential parameter trends, trade-offs, and the importance of locality in the kernel's data access patterns.**

***Index Terms*—HPC, Data Compression, In-line Compression, Matrix Multiplication**

I. Introduction

High-performance Computing (HPC) is the practice of sharing computing resources and tasks among several nodes to enable streamlined data processing and increased computing power. Data scientists use HPC systems to process and generate terabytes and petabytes of data through simulations and data collection [9].

Memory storage sizes and transfer speeds are bottlenecks in HPC operations [1], [10], [11], [21]. To mitigate this bottleneck, HPC systems use data compression schemes to exchange extra computations to reduce the size of data to be stored or transferred.

Compression is a necessary tool for large-data HPC systems in two ways. First, it reduces the size of the data needed to be stored in memory storage. This enables efficient usage of the memory space to allow more data storage. Second, reduced data storage means that less data needs to be transferred to allocated memory, enabling faster transfer times than transferring the complete original data.

Processing any data requires loading the data into a node's allocated memory. Any compressed data must be decompressed for calculations to have any meaning. The standard approach is to load all of the required application data in the kernel's allocated memory space. This approach requires using large nodes with massive memory spaces, enabling fast transfers and calculations across the entire dataset. This method enables fast calculations, but enabling this approach has some drawbacks. The more memory a kernel needs, the less likely the resources will be available to perform that function. Decompressing and storing an entire dataset requires nodes with memory suitable for such large data. The process would require nodes to be selected that have the memory capacity necessary for that process. Because there is a limited number of large nodes available, those that are available have a high demand. Enabling nodes with large memory becomes expensive time-wise due to waiting for the necessary resources. In a cloud-based system, this time-cost becomes a monetary cost to access a high-demand resource.

Costly nodes encourage searching out a solution that could process large data with smaller, less powerful nodes. One could search out the smallest possible node with the memory space to analyze data the standard way, but that approach postpones examining if those resources are being used effectively. This allocated memory management opens the door for in-line compression and other allocated memory reduction techniques.

A. Purpose of this Research

This research analyzes the properties of in-line compression to explore its benefits and limitations across input parameters and configurations. In-line compression parameters such as data organization and blocking impact the number of compression/decompression operations requested, the runtime of the system, and the compression ratio or memory benefit from utilizing in-line compression.

In-line compression builds on preexisting kernels to improve their allocated memory access, and any improvements provided by in-line compression are relative to the performance of the kernel. Matrix-multiplication is used as the exemplary kernel due to its abundance and importance in HPC operations as well as its ability to demonstrate features like locality and multiple datasets that are present in many other kernels

This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114.

that could benefit from in-line compression techniques. An interesting aspect of in-line compression is the use of splitting the kernel dataset into several sections, or blocks, to enable random access to compressed data.

The properties obtained by this research informs decisions for improving the performance of in-line compression in systems involving many in-line kernels. This research makes the following contributions to in-line compression development:

- Discuses the importance and potential use-cases for both full and partial in-line compression.
- Uses matrix-matrix multiplication as an exemplary kernel to analyze design trade-off decisions went adapting code to use both types of in-line compression.
- Experimental results on dense matrices representative of those in ML and scientific computing show in-line compression significantly reduces the memory footprint for a dense matrix-matrix multiply by more than 300×.
- Determine that data layout transformations that leverage knowledge of both the data access pattern and the compressed array structure are required to obtain good performance.

The following sections lay a framework for understanding and building upon in-line compression. Section II introduces key background concepts for the construction of in-line compression, such as compression itself, kernels, and the purpose and use cases for in-line compression. Section III presents the in-line compression architecture and describes how in-line compression can be done. This chapter also highlights key parameters and presents an in-line compression performance model. Section IV examines key parameters experimentally by exploring the results of in-line compression performance on a Matrix-Multiplication kernel.

II. Background and Related Work

In-line compression is based on several mainstays in HPC development. Existing compressors, with their differing design philosophies and architectures, can be incorporated into an in-line compression application. In-line compression applications incorporate these compression techniques in the same continuous operational sequence as a kernel performing a specified algorithm. The kernel that is selected determines several options for how compression can be implemented in that kernel, and each compression option impacts the overall runtime and memory performance of the in-line compression algorithm. In-line compression algorithms vary based on the use case of the kernel, how much memory is available to the kernel, and the memory access patterns of the kernel.

A. Compression

1) Compression Factors: Data compression is an effective tool to reduce memory storage and transfer requirements in HPC systems [4], [7], [12]. There are several important factors to consider when compressing data. The compression ratio or rate of the compression quantifies the change of data size between the original size and the compressed size. Equation 1, with the compression ratio CR, the original data size O, and the compressed data size C, reveals how a positive compression ratio indicates a smaller compressed data size.

$$CR = \frac{O}{C} \tag{1}$$

Another important consideration of compression is the compression runtime speed, as the memory reduction comes at the cost of additional processor time. A slower but more precise compressor is great for large scientific data where accuracy is important, but it is not as useful for streaming applications that require fast decompression operations. A final consideration is the impact of the compression on the integrity of the data. Some systems require the compression process to be able to exactly restore the original data, while others can sacrifice some acceptable level of accuracy for improvements in runtime and rate.

2) Lossless and Lossy Compression: Lossless compression algorithms are designed to guarantee that the data generated by decompression perfectly matches the original uncompressed data. There is no loss in the data accuracy; the error is zero. This type of data is useful for applications such as measuring sensitive data, recording rare events, and fields where precision and accuracy are critical. This accuracy comes at a cost of increased runtime and small compression ratios.

Error-bounded lossy compression sacrifices an acceptable level of accuracy to gain increases in compression ratio and runtime. The acceptable error is a parameter of the compressor that can ensure a specific error bound range. State-of-the-art lossy compression ratios can enable compression ratios in the range of 20 [4]–[6]. Lossy compression is commonly used to compress scientific floating-point data, as it is by nature an approximation.

There are two major error-bounded lossy floating-point compressors used in this research. SZ is an adaptive prediction-based error-bounded lossy compression framework developed at Argonne National Laboratory [4]–[6]. SZ utilizes three main error-bounding modes. The absolute error-bounding (SZ-ABS) limits the error to within a specified absolute error bound. The point-wise-relative error-bounding mode (SZ-PWR) limits the error to within the user-provided proportion of the dataset's range. The Peak-Signal-to-Noise-Ratio error-bounding mode (SZ-PSNR) guarantees that the PSNR calculation of the data retains the minimum user-provided value.

ZFP is a block-wise transformation-based compressor developed at Lawrence Livermore National Laboratory. [7] Two ZFP error-bounding modes are utilized in this research. First, the fixed-accuracy mode (ZFP-ACC) limits the absolute error to the user set error bound, similar to SZ-ABS. Second, the rate-based ZFP error-bounding mode (ZFP-RATE) supports random access into the compressed data at the cost of compression ratio.

Testing various compressors normally requires a complete redesign of software to configure and run different compressors with different APIs. LibPressio [8] is a software library

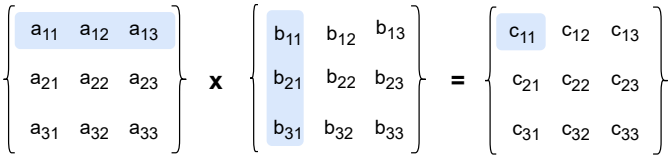

Fig. 1. Matrix Multiplication Implementation

designed to provide a common interface for various compression algorithms within HPC environments. It allows developers to integrate compression and decompression functionalities into their applications more easily by abstracting away the complexities of individual compression libraries.

B. Matrix Multiplication

Matrix multiplication is a common linear transformation kernel that is common in many HPC use cases, such as scientific computing, graphics processing, machine learning model training, and cryptography. For this reason, matrix multiplication is the representative kernel used in testing in-line compression.

Given matrices A and B where the number of columns in A matches the number of rows in B, matrix C is calculated by Equation 2, where n is the size of the matrices. Figure 1 provides a visual guide for what the data access pattern for matrix multiplication is. For each of the output elements in C, the corresponding row in A and column in B are accessed and linearly transformed to generate that output element.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj} \tag{2}$$

The time complexity of the standard matrix multiplication kernel is $\mathcal{O}(n^3)$. While optimizations to matrix multiplication have been made to improve its runtime complexity, the standard matrix multiplication calculation is used as an example for in-line compression.

C. Related Work

Compression schemes that reduce memory use are the basis for memory savings in HPC. Much research is dedicated to use cases and improvements for data compression. [21] details use cases for floating-point lossy data, the type of data used in this research. Compressed plots and images keep the data size lower while still retaining acceptable visual fidelity for the image's use case. These compressed files are smaller than uncompressed data, reducing the memory footprint of the data. Smaller files implies a reduced intensity of the data stream when transferring data from memory storage [25]–[29].

These software-level benefits can be used to reduce the I/O time of the system, accelerate checkpoint/restart systems, network speedup [30]–[32], and accelerate the overall kernel execution. Reduced checkpoint times [22]–[24] allows HPC simulation checkpoints to have a smaller impact on the overall application operations while still providing the memory backup.

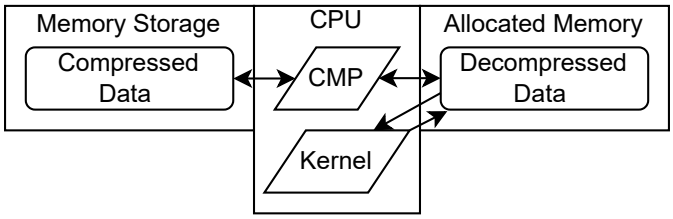

Fig. 2. Standard Compression Workflow

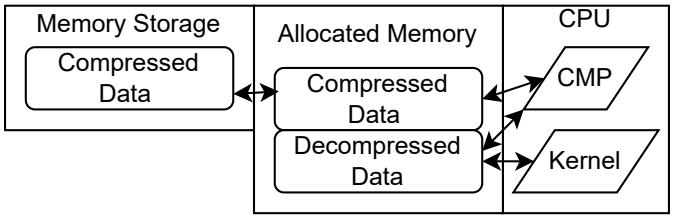

Fig. 3. In-line Compression Workflow

Caching is enables speedup by keeping frequent or imminent resources close to where computation is performed. This can be performed through hardware and software caches that save a copy of frequent data to reduce access times for later operations. Software caches [13]–[15] are useful to improve I/O performance and to cache input for parallel tasks. Hardware caches [16]–[20] have been paired with compression to expand the size of main memory and hardware caches. In-line compression has been explored utilizing software caches, specifically how cache configuration impacts the performance of full in-line compression [3].

III. In-line Compression Architecture

A. Methods Summary

In-line compression is a transformative data management system that uses compression and aggregation techniques to reduce the amount of data stored and transferred in allocated memory in high-performance computing environments. This section explores the differences in purpose and implementation between standard memory storage compression and in-line compression.

1) In-line Compression Overview: The purpose of in-line compression differs from compression for memory storage. Compression is commonly used to reduce the size of kernel data that is stored in memory storage such as a hard drive or cloud memory. Its purpose is to reduce both the storage size and the size of data needed to be transferred to a node for computation. The purpose of in-line compression is to reduce the amount of allocated memory in a node that is needed to access all of the data necessary for a kernel's computation.

Figure 2 illustrates the standard method of decompressing data to use with a kernel. The compression and decompression operations occur between the compressed data housed in memory storage and the decompressed data storage in allocated memory. The entire dataset is decompressed in main memory.

Figure 3 demonstrates the in-line compression implementation. The compressed data size is copied into allocated memory. When data is needed, only what is needed for a specific operation is decompressed. This means that the compression and decompression operations retrieve and store data from the allocated memory alone. Note that if the entire compressed data gets decompressed, the allocated memory usage is larger than its standard counterpart. The compressed data needs to be arranged such that individual sections (or blocks) of that data can be decompressed as needed. Block selection is a feature of interest when exploring the performance of in-line compression.

Enabling in-line compression in a kernel involves three main steps: configuring input data in the format for that in-line compression configuration, storing compressed data in allocated memory, and moving compressor calls to data access time. First, the kernel data is arranged into blocks of data that are individually compressed. Any manager for in-line compression needs to keep track of which block each element of data is in so that the correct block can be randomly accessed when the data is needed at runtime. Second, the in-line configured data is copied over to allocated memory. This enables the compressor to quickly access the data blocks that are needed for decompression without relying on slower memory storage bus access. Third, instead of decompressing all blocks before the kernel's operation to keep the data close to the kernel, the decompression calls need to be moved into the kernel's data access scheme. That is, when a new row or data from a specific block is needed, any old blocks are compressed back into allocated memory, and the correct blocks are decompressed to take their places.

2) Block Types: Breaking a dataset into blocks of data enables each block to be compressed and decompressed individually, enabling random access to compressed data. How the data is blocked informs runtime and memory performance.

There are two main types of in-line compression blocking schemes: full and partial. The full in-line compression scheme involves compressing each sub-dataset individually. For the matrix multiplication kernel, each matrix is compressed individually. When a matrix is needed, that entire matrix is decompressed. This enables fast access times as the entire matrix is present in allocated memory, but the allocated memory must store the entire decompressed matrix.

The partial in-line compression scheme enables the user to keep less active allocated memory decompressed by limiting the size of each block of compressed data. This results in many smaller compressed data blocks. The user specifies a block size, the number of elements in each compression block. For all of the data in all of the sub-datasets, data is compressed in groups of that block size. Accessing smaller blocks involves decompressing the blocks needed, manipulating the data in those blocks, and re-compressing that data block. These additional compression calls increase the overhead of the compressor on runtime but limits the allocated memory usage. This enables less active memory usage as only a necessary fraction of the data is decompressed during each kernel operation. How these blocks are selected can play a role in memory and runtime performance of in-line compression.

3) Block Tuning Factors: Block size is an integral component of how in-line compression works, and the block size parameter needs to be tuned for its use case. The extreme block sizes provide qualitative insight into how the block sizes affect memory and runtime performance. On one extreme, full in-line compression, where every matrix is its own block, provides access to an entire dataset close to memory. This means that the decompressed matrix is taking a relatively large space in allocated memory, but only one set of compression and decompression operations are needed to access this data. Full in-line compression causes a small runtime increase, but it enables an entire matrix to be decompressed with easy access to the kernel at the cost of large allocated memory usage. On the opposite extreme, a partial in-line compression scheme with each element being its own block (i.e. a block size of 1) has the opposite performance. The allocated memory usage is very minimal, as only the elements needed for each operation are compressed at once, but this memory savings comes at a large runtime cost to enable the many compression and decompression calls to access those small blocks. Tuning these block sizes are essential to providing the memory improvements necessary for a kernel on user-specified hardware while keeping the runtime cost at a tolerable limit.

Block count also subtly affects the memory usage in the compressed data. Each compressed block has some level of overhead metadata that enables the decompression of that block. Having more blocks can decrease the impact of the decompressed data on the allocated memory size, but more blocks can increase the compressed data's impact on allocated memory through 2 main methods. First, the increase in blocks increases the amount of memory attributed to metadata. As block sizes decrease toward the minimum of 1 element per block, the metadata could even be larger than the encoded data depending on the chosen compressor. Second, less data in blocks means less locality for the compressor to take advantage of when compressing the data. Compressors like SZ and ZFP that rely on interpolating from neighboring data have better compression ratios when more data is compressed as compressions can be informed by more data [4], [7], [33]. These increases in compressed memory size as the number of blocks increases enables the memory improvements to eventually become saturates, limiting more allocated memory size improvements enabled by block size.

The context of data is important in selecting blocks for that dataset. Data rarely exists in a vacuum, and patterns in types of data can be leveraged for better compression. Block dimensionality can be used to leverage the dimensionality used by compressors. 1-dimensional blocks group the data as if it is just a list of data. This would be the equivalent of only looking at a row of a picture's pixels to draw information from. 2 and 3 dimensional blocks could keep data blocked together if they are frequently decompressed together for compressors that use multiple dimensions. If a compressor makes decisions using multiple dimensions, then having blocks that consider these

multiple dimensions may be useful to reduce the compressed data size in allocated memory.

4) Kernel Data Access: Decompressing data when the kernel needs it is the main means for in-line compression to reduce the allocated memory usage. HPC kernels access data differently depending on what operations they are performing. A kernel could access data row-by-row, sequentially, or even random access. When selecting the block sizes and locations, it is important to take the kernel's access patterns into consideration. Blocking choices that decompress all and only the required data for each operation and accomplish this with minimal decompression operations balance reducing the decompressed allocated memory usage with managing the runtime overhead for compression.

The matrix multiplication access pattern provides a useful example to explore this concept of kernel access patterns affecting the data blocking. To calculate a single element in the output matrix, all of the elements in the corresponding row in the first matrix and the corresponding column in the second matrix must be accessed.

Block choice matters when tuning in-line compression for a kernel, and the matrix multiplication kernel is no exception. Because each kernel operation requires access to a 1-dimensional slice of multiple matrices, 1-dimensional blocking enables streamlined access to the specific required data. An entire row or column needs to be decompressed at once to enable each operation. Thus, selecting a block size that is related to the shared row and column length limits extra decompression operations.

B. Compression Manager Analysis

Managing the compression and decompression of memory blocks inside of the allocated data is essential to the implementation of in-line compression. In-line compression for this research was managed through an in-line compression manager (ICM). The ICM enables the selection of compressor, compressor options such as error bound, block size, and other parameters for in-line compression.

The ICM provides a good starting point for understanding in-line compression. It enables the blocking and compressed data management necessary to enable full and partial in-line compression. It provides options to explore matrix dimensionality, block sizing, compressor choice, error bound, and optimization choices. The main disadvantages of the ICM's implementation is the lack of certain options. Currently only SZ and ZFP are supported compressors, athough the use of LibPressio makes adding more compressors trivial. Data blocking in the ICM is only 1-dimensional, which limits the context a compressor has to enable higher compression.

C. In-line Compression Metrics

The general performance of in-line compression algorithms can be modelled by understanding how in-line compression is set up. Input parameters such as kernel, compressor, specified compression ratio, original data size, and block size play a role in affecting the allocated memory usage and the runtime of the kernel.

1) Compression Ratio: Compression ratio (CR) is typically defined with respect to the uncompressed data size (U) and the compressed data size (C) as reflected in Equation 1 where U is the original data size. A simple ratio easily demonstrates whether positive compression occurred (if $CR > 1$).

While this compression ratio information is useful, it does not demonstrate the size of the active allocated memory data. Since reducing the allocated memory usage is the main goal of in-line compression, having a memory measurement that takes the allocated data size into account is essential to accurately assess the memory implications of in-line compression. Equation 3 describes the compression ratio equation used for the following results. Because the allocated memory size is the compressed data size (C) and the decompressed block sizes (DB), the modified in-line compression ratio (iCR) is a compression ratio that takes the entire allocated memory into account, rather than just the compressed size of the data.

$$iCR = \frac{U}{C + DB} \tag{3}$$

This modified compression ratio reveals how block choice can affect the overall allocated memory usage. Take a matrix multiplication kernel that process 2 matrices (A,B) and generate a third matrix (C) where AB=C. First, one can note that reducing memory involves a balance between the compressed data size and the size of the decompressed blocks. Reducing allocated memory usage involves reducing or mitigating the growth of these two compression outputs.

Second, the in-line compression ratio reveals a possibility for negative compression with full in-line compression. If the kernel's in-line compression is implemented with full in-line compression, note that all 3 matrices are always decompressed to access the data if the kernel is using full in-line compression. This means that the uncompressed data is the same as the decompressed block data; all of it is decompressed. No matter the size of the compressed data, the allocated memory size is larger in this configuration than if in-line compression had not been used at all. This case illustrates when full in-line compression is useful. If a different kernel was used, one that needed access to only a subset of the matrices, then only a subset of the data needs to be decompressed. The in-line compression ratio aids in describing allocated memory usage regardless of the matrix dimensions.

2) Compressor Counts: Each time that a block is compressed or decompressed, the specified compressor is called. While the compressor choice and size of the data do play a role in runtime, the number of compressor calls indicates whether there are inefficiencies in the blocking for the specified kernel. Because the time to compress and the time to decompress are different, it is important to distinguish which type of compressor operation is more prevalent for which matrix.

Two methods of approaching this runtime metric is to consider its response to block size and kernel access patterns. First, the more blocks there are, the more compression and decompression operations needed to access the block. These

compressor calls increase runtime through the overhead necessary to access the data. Second, efficient blocking dependent on the kernel's access patterns can reduce the number of compressor calls. Depending on how the kernel accesses data and how the blocks are configured, compressor calls have extreme cases. If several blocks are accessed to decompress all of the elements needed for an operation, several compressor calls need to be made. A block choice that aligns with the kernel's access patterns can reduce the number of compressor calls.

3) Runtime: In-line compression achieves its allocated memory usage reduction at the cost of runtime. Understanding what factors go into runtime is important to understand how runtime growth can be controlled as improvements are made in memory. Equation 4 identifies what factors influence the runtime of an in-line compression kernel, with runtime R, original kernel runtime K, and the runtimes for all compression and decompression operations O.

$$R = K + \sum O \tag{4}$$

The runtime for an in-line compressed kernel is the sum of three main factors: the original kernel's runtime, the time for all compression operations, and the time for all decompression operations. The number of compressor calls directly impacts the time for compressions and decompressions. The kernel choice itself is a constant if in-line compression is to be selected for that specific algorithm. The time complexities of the compressor accesses and the original kernel are important to predicting the runtime of the in-line compression kernel itself.

The compressor calls, with its time complexity, occur inside the kernel data access calls based on the original kernel, with its time complexity. At low matrix dimension sizes and low block counts, the compressor calls have a larger impact on the relative runtime. As the dimension size of the matrices increase, the time for compressor accesses may be amortized by the time taken by the original kernel.

IV. Experimental Results

A. Testing Environment

1) Hardware: Tests were run using the Clemson's University's Palmetto Cluster Phase 19b. Each compute node in this phase contains 2 Intel Xeon 6230R CPUs along with 372 GB of DRAM. Each test is run using a single core. This is done to profile and understand the performance of a single process. Future work will expand this work to investigate in-line compression for shared memory and distributed memory applications.

2) Software: The software utilized for these tests were built on GCC (GNU Compiler Collection) version 12.1.0. The compressors SZ version 2.1.8.1 and ZFP version 1.0.0 were used as floating-point error-bounded lossy compression benchmarks, and compressor managment was maintained through LibPressio version 0.97.3. Python 3.11.6 served as the primary scripting language for orchestration and automation of computational tasks, with additional support from Seaborn version 0.13.2 and Matplotlib version 3.8.2 for data visualization and analysis.

3) Testing Format: The matrices used for matrix multiplication were square matrices of a user-specified dimension (n in plots). These matrices were populated with linear transformations of pi.

The lossy floating-point compressors used in this research are error bounded. Unless otherwise noted, the error bounds utilized are $\epsilon = 0.01$ for SZ_ABS, SZ_PWR (point-wise relative), ZFP_RATE, and ZFP_ACC (accuracy). For SZ_PSNR, $\epsilon = 40db$. Unless noted, all experiments utilize the B optimization that stores the B matrix as transposed. (see Figure 8)

B. Memory Footprint Analysis

1) Allocated Size: In-line compression reduces the amount of allocated memory that is being used by the application by storing data outside the working set in compressed form. Only when data is needed by the application is it decompressed just before it is operated on. One key parameter to control how much data is stored decompressed is the size of each block that the data is decomposed into prior to compression. We first experiment and determine the impact of block size on the allocated memory size, and it also illustrates how memory is organized inside of the allocated memory.

This experiment demonstrates how in-line compression is an effective means of reducing the allocated memory size. To understand the resulting allocated memory size, one must understand how in-line compression organizes the data in the allocated memory. Allocated memory contains 2 main sections: the compressed application data, and the decompressed data needed for a single operation — i.e., kernel. This experiment highlights how the sizes of those sections trade-off to result in overall data reduction.

Figure 4 illustrates the used memory size and the contents of that memory at various block counts. This experiment performs matrix multiplication on matrices of order $n = 50$. We use the SZ compressor with an absolute error bound of $\epsilon = 0.01$.

The height of each bar is the amount of allocated memory needed for each block size. Each bar is subdivided into the memory that is needed for this application. Matrices A,B, and C all need to be compressed, and some level of memory is needed to store their decompressed blocks. The lighter shades represent the compressed section of memory, while the darker shades represent the largest decompressed blocks for that block size.

The overall trend of this plot is that the allocated memory size decreases as the number of blocks increases. This test demonstrates that the allocated memory size is a trade-off between the compressed size of the entire data and the decompressed data sizes. More blocks means that less decompressed data needs to be stored for each block. However, more blocks requires more compressor metadata overhead and a potential

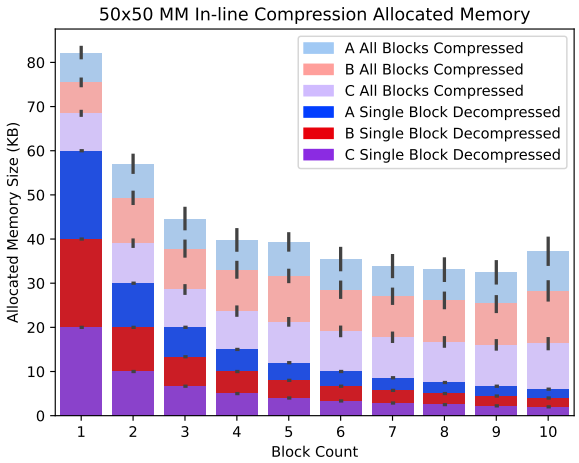

Fig. 4. Total allocated memory usage for matrix-matrix multiplication on a matrix of order $n = 50$.

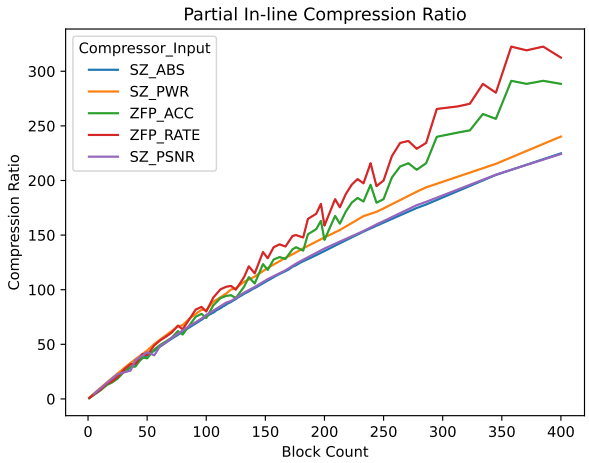

Fig. 5. In-line compression ratio for various compressors for matrix-matrix multiplication with order $n = 100$.

for the smaller blocks to not be able to be compressed as well. Future in-line compression algorithms should be designed to limit or reuse meta-data.

The key takeaway of this experiment is that partial in-line compression is made possible due to the balance of the decompressed block sizes and the compressed data size. This trade off in block count needs to be tuned to enable the memory usage reduction that enables in-line compression's use case of being able to run kernels with large data on nodes with smaller allocated memory space. A second takeaway is that this memory balance has another input that was held constant in this experiment: the data size.

2) Partial In-line Compression Ratio: Considering the full input data size while attempting to minimize the allocated data size requires a new metric to keep track of the allocated memory's size relative to the original data size. Normally for most compressors, compression ratio is used as the metric, as it compares the uncompressed size to the compressed data size. In this case, a modified in-line compression ratio is used to relate the original data size compared to the overall allocated memory usage (see Section III-C1). The use of this modified compression ratio is necessary to contribute for the uncompressed blocks of data that are also present in allocated memory. Essential inputs toward the in-line compression ratio include the original data size and the number of blocks.

Figure 5 compares the in-line compression ratio achieved by varying the number of blocks that we decompose our matrix of order $n = 100$ into before compressing with various versions of SZ and ZFP. The error bounds is $\epsilon = 0.01$ for SZ_ABS, SZ_PWR (point-wise relative), ZFP_RATE, and ZFP_ACC (accuracy). For SZ_PSNR, $\epsilon = 40db$.

The in-line compression ratio increases with block count, but it has an upper limit. The allocated memory size decreases by dividing each matrix into more blocks, thus having less data decompressed in main memory. The choice of underlying compressor also impacts the compression ratio, although only in the compressed data section of the allocated memory. We see ZFP performance the best due to its lower meta-data overhead than SZ. This becomes magnified for large block counts where the number of elements per block is small.

A compression ratio of 300× is remarkable, but where is this benefit coming from? The main contributor to this is the decrease in the size of the allocated memory size due to less decompressed data. With more blocks per data array, less memory is accessed for each calculation — i.e., the decompressed working set is small. However, as the block size shrinks, more compression and decompression operations are needed for the computation. With the additional additional compression and decompression operations the overhead in runtime increase; thus longer application execution time (see Section III-C2).

The compression ratio of ZFP is particularly notable. For high block counts, its compression ratio outshines SZ. This is possible by having less data in each block for high block counts, but also due to the metadata of both compressors. ZFP has less metadata than SZ [33]. For a few larger blocks, this metadata difference is imperceptible. For several smaller blocks however, the metadata takes on a larger proportion of the memory usage. Thus, ZFP demonstrates a higher compression ratio than SZ for large block counts.

3) Full In-line Compression Ratio: This experiment compares the matrix dimensions to the compression ratio for a full in-line compressed matrix multiplication kernel. Full in-line compression was chosen to analyze a simple example with 1 block per matrix, including how data is impacted by the error bound.

An experiment on how data size impacts the in-line compression ratio is essential due to the difference from traditional compression ratios. The entire allocated memory size is compared to the uncompressed data size. Thus, it is important to consider how much of the data set is decompressed, even if using full in-line compression where an entire matrix is

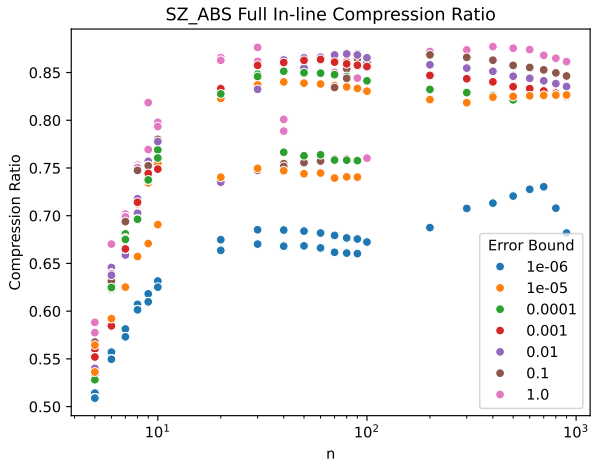

Fig. 6. Full In-line Compression Ratio by Dimensions and Error Bound

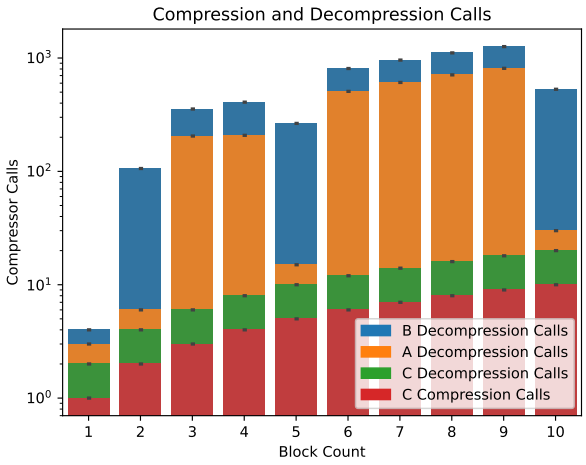

Fig. 7. Compressor Call Counts per Matrix with respect to Block Counts

compressed together.

Similar to the previous in-line compression ratio experiment, this experiment compares the in-line compression ratios over square matrices orders ranging from $n = 5$ to $n = 1000$. The SZ compressor is used with the indicated error bounds to illustrate their affect on in-line compression, whether full or partial. Results for other compressors are similar and omitted for clarity.

Figure 6 illustrates how data size impacts compression ratio. Because compression is possible through locality and repetition, it is expected that smaller data sizes have a smaller compression ratio. The effects of the error bound on each block's compression are passed through to in-line compression, with the tightest error bound having the smallest compression ratio, and the loosest error bound can sacrifice accuracy to gain space.

Note that this experiment is using full in-line compression, or a single block per matrix. This means that for the in-line compression ratio equation 3, if all of the matrices in the dataset are involved in each operation in the kernel, the uncompressed data size and the decompressed block sizes are the same. This situation is present with matrix-matrix multiplication and yields negative compression no matter what the data size.

This situation highlights that full compression is great for some use-cases but not for others. If a subset of the matrices are used per kernel operation, then only a subset of the matrices need to be decompressed for each operation. The uncompressed dataset is greater than the necessary decompressed matrices for each operation, so depending on the size of the compressed dataset, positive compression is possible.

In-line compression can yield negative compression if the decompressed data size is larger than the difference between the uncompressed data size and the compressed data size. The error bound and data size impact the in-line compression performance in the same way they impact the compressor that is chosen.

C. Timing

Similar to how traditional compression improves the compression ratio by leveraging more time and computational resources, obtaining a decrease in allocated memory usage comes at a cost of runtime. The main source of additional time usage is in the compression and decompression operations. This section presents the general compressor call pattern as well as some techniques to reduce the number of compression calls and to keep the runtime manageable.

1) Compression/Decompression Calls: Runtime analysis requires determining what operations takes up time, and the main timing difference between a standard kernel and an in-line kernel is the number of compression and decompression counts. Understanding what causes these compressor count increases is essential to selecting proper settings to reduce the runtime increases.

Figure 7 identifies which matrices are compressed and decompressed for the in-line compressed matrix multiplication kernel. A portion of each matrix is required to compute the next output matrix. The order in which operations are done matters for runtime because the correct portions of the matrices need to be decompressed in allocated memory. Identifying the reasons behind which blocks are decompressed more frequently enables understanding for better fitting a kernel for in-line compression with respect to its runtime.

This experiment compares the number of compression and decompression calls with respect to the number of blocks per kernel. The dimensions of the matrices analyzed are 50x50.

As the number of blocks increases, the number of compression and decompression operations also increases. Block counts of 1, 2, 5, and 10 are interesting due to having the least number of compression calls (Full) or actually having less compressor calls than neighboring block counts. The only matrix that is compressed is matrix C, as it is the one with the results that need to be stored. Note that the number of matrix

C compressions is equal to the block size. This is because the order in which operations are done iterates through all C matrix entries, then through the blocks containing the entries in A and B to compute the C entry.

The decompressions provide the bulk of the compressor calls. All of the C blocks are decompressed once to access them to compute the resulting entries. This step could be removed for kernels where intermediate operations don't need to be stored, like for matrix multiplication. The number of A and B decompression counts provide factors of interest, particularly with how the selected decompressed blocks align with the kernel's access pattern.

For each entry in C, a single row from A and a single column for B need to be accessed to perform a standard matrix multiplication operation. How the blocks are selected impacts which ones are available to get data from. For block size 1 (i.e. full in-line), each matrix is only decompressed once, as all of the rows and columns have been decompressed by the time they are needed for any of the matrix multiplication operations.

Block sizes 5 and 10 have different characteristics from their neighboring block counts. Not only are their overall compressor counts less than their neighbors, but the A matrix requires significantly less decompression counts. Because all dimensions tested are divisible by 2, 5 or 10, this illustrates an important kernel access pattern for matrix multiplication that can be used to reduce decompression calls. Because the blocks used for testing are 1-dimensional, each block can be a row or a part of a row. In this case, block sizes of 2, 5, and 10 are factors of the matrix row length. This feature limits future redundant operations for other data in that block (for example, if a block overflows into another row).

While block counts of 2, 5, and 10 have a significantly smaller number of compressor calls due to a decrease in A matrix decompressions, the B matrix decompressions look significantly larger. This is an artifact of the logarithmic scale used in this plot. A linear scale reveals that B matrix decompressions are directly related to block count. B is dependent on the kernel's access pattern.

This experiment also showcases what does not affect the number of compressor counts. Data dimensions, compressor choice, and error bound do not affect the compressor counts because they are part of the compression process. They are not used to select how many counts, but rather how the data is compressed in those operations. What does affect compressor choice is kernel access pattern locality and the choices of how blocks are implemented.

2) Matrix Multiplication Informed Blocking: The previous experiment demonstrated how block choices can affect the compression and decompression operation counts. The choice in block dimensions and size details what kind of data is present in each block, and thus what is available during that step of the computation. The purpose of this experiment is to give an example of this property using the matrix multiplication kernel access pattern.

Each element in the output matrix C requires access to the corresponding row of matrix A and the corresponding column of matrix B. Because blocks with this system are 1-dimensional row-wise, one can access the partial or complete row with a limited number of decompression operations. However, since the data needed from the B matrix is by columns, the column data is contained in many row-based blocks. To compensate for this, an option for the kernel is to optimize the B matrix for the row-based blocking by transposing the B matrix. This enables the blocks to access the data needed by the kernel with less decompression calls. The dimensions for matrices in this experiment are 100x100.

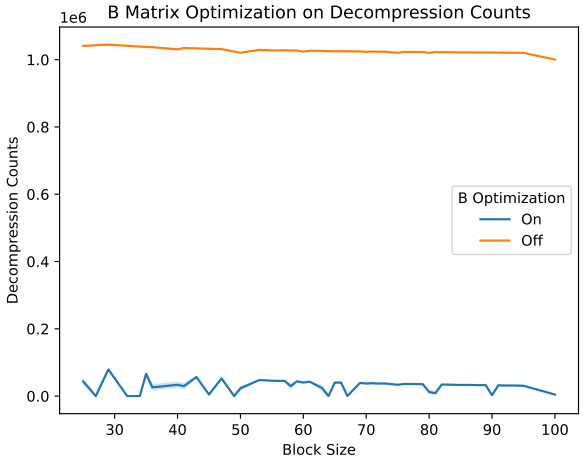

Fig. 8. B-Transposed Kernel Access Pattern on Decompression Counts

Figure 8 indicates a large decompression count difference between using using the B-transpose optimization and keeping the standard operation. For the 100x100 matrix, since the block size is less than or equal to the number of elements in the row, accessing a column of B requires accessing 100 different blocks. When this operation is accounted for all 10,000 elements in the output matrix C, that setup requires 1,000,000 decompression operations for the B matrix alone. In all of these decompression steps, many unused elements are being decompressed along with the column data of interest. Transposing B allows the kernel access pattern to match the blocking pattern for in-line compression to limit the number of decompression counts and unnecessary data accesses.

The implications of this experiment go beyond just matrix multiplication. Whatever the base kernel is, an important means of limiting a runtime increase is to select blocks and kernel configurations that have blocks that meaningfully relate to the data needed for each operation in the kernel.

3) Runtime: After examining the sources of compression and decompression operations, how do these observations relate to runtime? The original kernel's runtime complexity, the compressor, and the block count all affect the overall runtime of the in-line compression kernel. Figures 9 and 10 demonstrate how these choices impact the overall runtime of the in-line compression kernel.

These experiments relate runtime to the same parameters

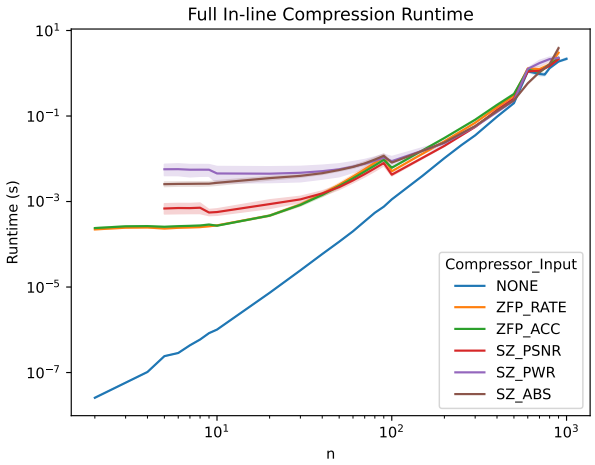

Fig. 9. Runtime with All Compressors and Dimensions

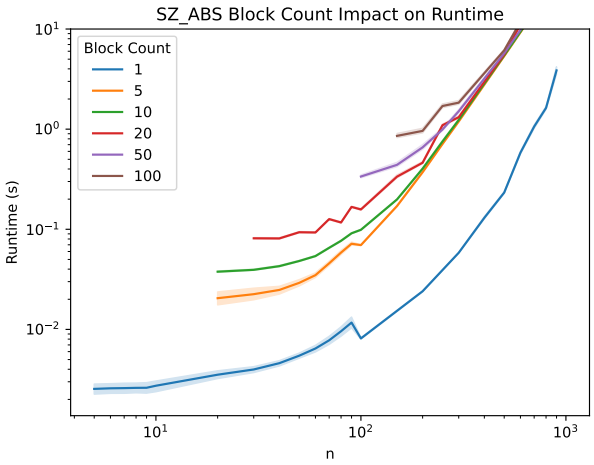

Fig. 10. Runtime with SZ-ABS and Block Count

that memory was compared to: data size and block count. Additionally, the compressor choice is included to provide a better picture on all of the factors affecting runtime. These experiments compare the runtime to the matrix data dimensions, while also highlighting the compressor choice and the block counts. The runtime of the original kernel without in-line compression is also included for comparison. The dimensions of the matrices analyzed range from 5x5 to 1000x1000.

Adding any in-line compression to a kernel increases the runtime to compress and decompress, as Figure 9 demonstrates. The original matrix multiplication kernel has a $O(n^3)$ time complexity, and each of the in-line methods with various compressors follows suit. Each of the compressors has a similar relationship to what it had to memory in Figure 5: ZFP has the larger compression ratio and has the smaller runtime, while SZ has the smaller compression ratio and the larger runtime. An interesting point of note in Figure 9 is that around n=700, SZ-ABS gets a faster runtime briefly. This is peculiar that a compressed version would run faster than the kernel it is based on. An explanation for this could be the hardware cache optimizations, but this could be an area for future interest.

Another feature of interest with Figure 9 is that for all compressors with block size 1, the runtime of the compression gets amortized into the overall kernel runtime for large n. This indicates the possibility of in-line compression kernel runtimes in the same time complexity as the original kernel.

Figure 10 illustrates the direct relationship between runtime and the number of blocks. Each block adds additional compressor time to the original kernel time complexity. Not only is it additional time, but almost an order of magnitude more.

The in-line compression kernel has the time complexity of the original kernel with the linear increase of the number of compressor operations and the impact of the original compressor's runtime. Compared with Figure 9, Figure 10 reveals the possibility for tuning and improvement of the system itself. Offloading compressions to other threads and improved processor choice can mitigate the growth of runtime caused by additional blocks.

V. Conclusions

In-line compression provides a viable approach to reducing the allocated memory usage for HPC kernels with large kernel data. This research explored the runtime and memory performance of in-line compression on the matrix multiplication kernel. Matrix multiplication was selected as the exemplary kernel due to its prevalence in HPC systems and its ease of illustrating the properties of in-line compression.

The use cases for full and partial in-line compression were explored, revealing the trade-off between allocated memory usage and runtime. Larger block sizes provide a small allocated memory size reduction for a small runtime cost, while smaller block sizes enable greater reduction in allocated memory usage at the cost of runtime for compressing and decompressing those blocks. The spacial locality of those blocks impacts compressability, and block choices that leverage knowledge of both the data access pattern and the compressed array structure encourage good performance.

Experimental results from the matrix multiplication in-line compression kernel were analyzed for key parameters' affects on runtime and allocated memory usage. Findings include insights into how in-line compression works and methods for memory and runtime optimization. In-line compression is possible through a balance between the compressed data size and the amount of data decompressed. The decompressed data size is dependent on the block size, while the compressed data is dependent on the location of blocks and the compressor choice. Compression and decompression calls can be minimized by blocking with respect to the kernel's access pattern. Improvements are possible which can enable the compression overhead time to be amortized by the original kernel's runtime. The tuning of in-line compression parameters enables its impact on more kernels and applications than just matrix multiplication.

REFERENCES

[1] S. Ranjan, *Performance Modeling of Inline Compression With Software Caching for Reducing the Memory Footprint in PYSDC*, Master's thesis, 2023. Available at: https://tigerprints.clemson.edu/all_theses/4158.

[2] D. Fulp, *Resolving Soft Error Susceptibilities Within Lossy Compressed HPC Data*, Master's thesis, 2021. Available at: https://tigerprints.clemson.edu/all_theses/3657.

[3] S. Ranjan, D. Fulp, J. C. Calhoun, *Exploring the Impacts of Software Cache Configuration for In-line Compressed Arrays*, in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2022, pp. 1-7. DOI: 10.1109/HPEC55821.2022.9926289.

[4] S. Di, F. Cappello, *Fast error-bounded lossy hpc data compression with sz*, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730-739.

[5] D. Tao, S. Di, Z. Chen, F. Cappello, *Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization*, in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1129-1139.

[6] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, F. Cappello, *Error-controlled lossy compression optimized for high compression ratios of scientific datasets*, in *2018 IEEE International Conference on Big Data (Big Data)*, Dec 2018, pp. 438-447.

[7] P. Lindstrom, *Fixed-rate compressed floating-point arrays*, *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, Dec 2014, pp. 2674-2683.

[8] R. Underwood, V. Malvoso, J. C. Calhoun, S. Di, F. Cappello, *Productive and Performant Generic Lossy Data Compression with LibPressio*, in *2021 7th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-7)*, 2021, pp. 1-10.

[9] P. Xenopoulos, J. Daniel, M. Matheson, S. Sukumar, *Big data analytics on HPC architectures: Performance and cost*, in *2016 IEEE International Conference on Big Data (Big Data)*, Washington, DC, USA, 2016, pp. 2286-2295. DOI: 10.1109/BigData.2016.7840861.

[10] S. Ashby, P. Beckman, J. Chen, et al., *The Opportunities and Challenges of Exascale Computing*, Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, 2010, pp. 1-77.

[11] P. Messina, *The U.S. D.O.E. Exascale Computing Project - Goals and Challenges*, NIST, 2017.

[12] S. W. Son, Z. Chen, W. Hendrix, et al., *Data Compression for the Exascale Computing Era - Survey*, *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, 2014.

[13] B. Van Essen, H. Hsieh, S. Ames, et al., *Di-MMap – A Scalable Memory-Map Runtime for Out-of-Core Data-Intensive Applications*, *Cluster Computing*, vol. 18, 2015, pp. 15-28.

[14] A. Shafiee, M. Taassori, R. Balasubramonian, A. Davis, *MemZip: Exploring Unconventional Benefits from Memory Compression*, in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 638-649.

[15] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, P. Beckman, *Exploring Data Migration for Future Deep-Memory Many-Core Systems*, in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 289-297.

[16] A. Arelakis, F. Dahlgren, P. Stenstrom, *HyComp: A Hybrid Cache Compression Method for Selection of Data-Type-Specific Compression Methods*, in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 38-49.

[17] X. Chen, L. Yang, R. P. Dick, et al., *C-Pack: A High-Performance Microprocessor Cache Compression Algorithm*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, 2009, pp. 1196-1208.

[18] A. Jain, P. Hill, S.-C. Lin, et al., *Concise Loads and Stores: The Case for an Asymmetric Compute-Memory Architecture for Approximation*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1-13.

[19] J. San Miguel, J. Albericio, A. Moshovos, N. E. Jerger, *Doppelgänger: A Cache for Approximate Computing*, in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 50-61.

[20] G. Pekhimenko, V. Seshadri, O. Mutlu, et al., *Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches*, in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 377-388.

[21] J. Dongarra, B. Tourancheau, F. Cappello, et al., *Use cases of lossy compression for floating-point data in scientific data sets*, *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 6, Nov 2019, pp. 1201-1220. DOI: 10.1177/1094342019853336.

[22] T. Z. Islam, K. Mohror, S. Bagchi, et al., *McrEngine: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, 2012, pp. 17:1-17:11.

[23] A. H. Baker, H. Xu, J. M. Dennis, et al., *A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data*, in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14*, 2014, pp. 203-214.

[24] D. Tao, S. Di, X. Liang, et al., *Improving Performance of Iterative Methods by Lossy Checkpointing*, in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18*, 2018, pp. 52-65.

[25] T. B. Smith, B. Abali, D. E. Poff, R. B. Tremaine, *Memory Expansion Technology (MXT): Competitive Impact*, *IBM Journal of Research and Development*, vol. 45, no. 2, Mar 2001, pp. 303-309.

[26] B. Abali, H. Franke, X. Shen, et al., *Performance of Hardware Compressed Main Memory*, in *Proceedings of the Seventh IEEE International Symposium on High-Performance Computer Architecture, HPCA '01*, 2001, pp. 73-81.

[27] F. Douglis, *The Compression Cache: Using On-line Compression to Extend Physical Memory*, in *Proceedings of 1993 Winter USENIX Conference*, 1993, pp. 519-529.

[28] P. R. Wilson, S. F. Kaplan, Y. Smaragdakis, *The Case for Compressed Caching in Virtual Memory Systems*, in *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '99*, 1999, pp. 8-8.

[29] S. Levy, K. B. Ferreira, P. G. Bridges, *Improving Application Resilience to Memory Errors with Lightweight Compression*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, 2016, pp. 28:1-28:12.

[30] R. Filgueira, D. E. Singh, J. Carretero, et al., *Adaptivecompi: Enhancing MPI-Based Applications' Performance and Scalability by Using Adaptive Compression*, *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, 2011, pp. 93-114.

[31] R. Filgueira, M. Atkinson, A. Nuñez, et al., *An Adaptive, Scalable, and Portable Technique for Speeding Up MPI-Based Applications*, in *Euro-Par 2012 Parallel Processing*, 2012, pp. 729-740.

[32] L. Fischer, S. Götschel, M. Weiser, *Lossy Data Compression Reduces Communication Time in Hybrid Time-Parallel Integrators*, *Computing and Visualization in Science*, vol. 19, no. 1, Jun 2018, pp. 19-30.

[33] T. Liu, J. Wang, Q. Liu, et al., *High-Ratio Lossy Compression: Exploring the Autoencoder to Compress Scientific Data*, *IEEE Transactions on Big Data*, vol. PP, Mar 2021. DOI: 10.1109/TBDATA.2021.3066151.