A Portable, Fast, DCT-based Compressor for AI Accelerators

Milan Shah
North Carolina State University
Argonne National Laboratory
Raleigh, NC, USA
mkshah5@ncsu.edu

Xiaodong Yu
Stevens Institute of Technology
Hoboken, NJ, USA
xyu38@stevens.edu

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi1@anl.gov

Michela Becchi
North Carolina State University
Raleigh, NC, USA
mbecchi@ncsu.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

ABSTRACT

Lossy compression can be an effective tool in AI training and inference to reduce memory requirements, storage footprint, and in some cases, execution time. With the rise of novel architectures designed to accelerate AI workloads, compression can continue to serve these purposes, but must be adapted to the new accelerators. Due to programmability and architectural differences, existing lossy compressors cannot be directly ported to and are not optimized for any AI accelerator, thus requiring new compression designs.

In this paper, we propose a novel, portable, DCT-based lossy compressor that can be used across a variety of AI accelerators. More specifically, we make the following contributions: 1) We propose a DCT-based lossy compressor design for training data that uses operators supported across four state-of-the-art AI accelerators: Cerebras CS-2, SambaNova SN30, Groq GroqChip, and Graphcore IPU. 2) We design two optimization techniques to allow for higher resolution compressed data on certain platforms and improved compression ratio on the IPU. 3) We evaluate our compressor's ability to preserve accuracy on four benchmarks, three of which are AI for science benchmarks going beyond image classification. Our experiments show that accuracy degradation can be limited to 3% or less, and sometimes, compression improves accuracy. 4) We study compression/decompression time as a function of resolution and batch size, finding that our compressor can achieve throughputs on the scale of tens of GB/s, depending on the platform.

CCS CONCEPTS

• **Theory of computation → Data compression**; • **Computer systems organization → Neural networks**; *Data flow architectures.*

KEYWORDS

Compression, AI accelerator, ML training

Corresponding author: Sheng Di, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 Cass Avenue, Lemont, IL 60439, USA.

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06…$15.00
https://doi.org/10.1145/3625549.3658662

ACM Reference Format:
Milan Shah, Xiaodong Yu, Sheng Di, Michela Becchi, and Franck Cappello. 2024. A Portable, Fast, DCT-based Compressor for AI Accelerators. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658662

1 INTRODUCTION

Motivation: In recent years, AI models have grown rapidly in size, from 100s of millions of parameters to 100s of *billions* of parameters [33]. The training cost in terms of time, hardware requirements, and power consumption increases greatly with the size, with models on the scale of 100s of billions of parameters requiring hundreds to thousands of GPUs for training. For instance, OPT-175B has 175 billion parameters and was trained on 992 80 GB NVIDIA A100 GPUs [30]. To match this meteoric rise in training demand, AI accelerators with architectures more finely tuned for ML training and inference have been developed, including the Cerebras CS-2 [2], the SambaNova SN30 [6], the Groq GroqChip [8], and the Graphcore IPU [3]. These emerging architectures have already been deployed in a variety of scientific research tasks, including molecular dynamics simulations [12] and genome-scale language models [34]. In [12] and [34], AI accelerators are shown to improve time-to-solution and enable larger batch sizes, both of which are critical to performance.

Novel accelerators attempt to overcome the salient drawbacks of traditional AI training platforms like CPUs and GPUs. GPUs rely on frequent memory exchange between on-chip and global device memory, and require kernel launches for each layer of computation in a neural network. CPUs often have deeper memory hierarchies and lack high degrees of parallelism, which can be a bottleneck for tensor operations frequently occurring in neural networks. Many emerging AI accelerators seek to improve training and inference performance by introducing an abundance of on-chip memory close to compute while simultaneously achieving high degrees of data parallelism through specialized GEMM units, many ALUs, or hundreds to thousands of compute units. Across CPU, GPU, and AI accelerators, memory still remains a precious resource that must be rationed to train increasingly large models and datasets. SciML-Bench [28] is an "AI for Science" benchmarking suite that contains several training and inference tasks found in scientific applications. Datasets part of this suite as well as other datasets (i.e., the 800 GB training set used for OPT-175B [30]) are on the scale of tens

of GB to nearly the TB level [33]. However, on-chip memory of accelerators is on the scale of 100s of MB to tens of GB, significantly smaller than the size of training datasets, as shown in Table 1 and Table 2.

Limitation of state-of-art approaches: If batch sizes or resolutions of samples used to train models are too large, memory and storage can quickly be exhausted. As such, compression becomes a vital tool in managing model and dataset footprint. While previous works have studied compression for improving memory utilization and performance of models on CPU/GPU [13, 19] and for decreasing data transfer costs in distributed training [11, 21], little work has been done to explore compression for AI accelerators. These works have developed novel compression techniques for varying compression targets, but lack a direct port or optimized implementation for AI accelerators such that these emerging accelerators can reap the benefits of compression.

Key insights and contributions: In this work, we design a portable, DCT-based compressor that seamlessly runs across four emerging AI accelerators: the CS-2, SN30, GroqChip, and IPU. To the best of our knowledge, this is the first attempt to develop a lossy compressor compatible with these accelerators. The discrete cosine transform, or DCT [10], is widely used in image compression as a means of decorrelating image data to improve compressibility. Our compressor is implemented in PyTorch, allowing the end-user to call our compress or decompress APIs directly from their Python training or inference code. Our compressor seeks to provide an efficient means of reducing data footprint while maintaining data fidelity and requiring little programmer effort. Our key contributions are as follows:

- We develop a cross-accelerator compressor targeting training data. Our design only requires two matrix multiplications for compression and decompression each. DCT-II is applied to the input data, and only a portion of the DCT coefficients matrix is retained to yield the compressed data.
- We extend our compressor to support higher resolution inputs at lower memory footprint using a partial-serialization optimization. Additionally, we propose a Graphcore IPU-specific optimization that improves the compression ratio with little impact on test accuracy/loss.
- Our compressor maintains accuracy close to baseline, generally within 3%, while reaching compression/decompression throughputs of 100s of MB/s on GroqChip, 1-2 GB/s on a single IPU, 7-10 GB/s on a single SN30 RDU, and up to 26 GB/s on CS-2.
- We test the additional optimizations and find that partial serialization can enable higher resolution images (512×512) on SN30 and IPU, while the IPU-specific optimization can improve compression ratio by a factor 1-2× with 50% or less drop in throughput.

Experimental methodology and artifact availability: We perform a comprehensive evaluation of our compressor on the four AI accelerators and an NVIDIA A100 GPU. The compressor is evaluated with four benchmarks, a traditional image classification task and three AI for Science benchmarks, and a battery of compression and decompression timing tests with varying input data resolution and batch size. The code for this work is available at https://github.com/mkshah5/AI-Accelerator-Compression.git.

Table 1: Breakdown of accelerator specifications. CU = Compute Unit, OCM = On-chip Memory, PT=PyTorch, TF=Tensorflow.

	CS-2	SN30	GroqChip	IPU
CUs	850,000	1280	5120	1472
OCM	40 GB	640 MB	230 MB	900 MB
OCM/CUs	48 KB	0.5 MB	0.045 MB	0.61 MB
Software	TF, PT,CSL	SF, PT	PT, Keras ONNX	TF, PT, PopArt
Arch.	Dataflow	Dataflow	SIMD	MIMD

Limitations of the proposed approach: Our work focuses on portability and relies on the rapidly changing development ecosystem for each accelerator. As such, the compressor is lightweight and does not achieve compression ratios as high as compressors available on CPU and GPU. Additionally, our design is tailored for training data as this data is readily available across platforms. In the future, more work to optimize the compressor for each platform and for differing compression targets can improve compression ratio and speed as APIs to access activations and gradients are made available on these platforms.

2 BACKGROUND AND MOTIVATION

2.1 AI Accelerators

AI accelerators are novel architectures targeting fast training and inference of AI models. These accelerators are designed for operations and dataflows commonly required for training and inference, such as matrix multiplications and deep pipelines. In this section, we provide a high-level overview of each accelerator's architecture and programming interface. Table 1 outlines key specifications, including compute unit (CU) count, on-chip memory (OCM) capacity, software interfaces (TF for Tensorflow and PT for PyTorch), and architecture type.

2.1.1 Cerebras CS-2. The Cerebras CS-2 [2, 5] is a wafer-scale accelerator. The CS-2 wafer contains 850,000 compute units, called processing elements (PEs), and each PE has an associated 48 KB of on-chip memory, for a total of more than 40 GB of on-chip memory. A PE is specialized compute core that accelerates sparse matrix and other AI operations. The PEs are laid out in a 2-D meshgrid with high-speed interconnects connecting each PE to its neighbors. Once the user develops the model using one of the supported software toolchains, the Cerebras compiler physically maps the model computation to the PEs, arranging the computation in a *dataflow* manner. This enables deep pipeline-level parallelism where samples for training and inference flow from host to device memory, then to the on-chip PEs to proceed with computation. While the CS-2 does support a custom low-level programming interface, CSL, custom kernels in CSL are not fully integrated with PyTorch and Tensorflow. For models with less than 1 billion parameters, the CS-2 performs pipelined execution as-is, while larger models utilize a *weight streaming* execution method, where the parameters are streamed from pipeline stage to pipeline stage along with the sample or activation data.

2.1.2 SambaNova SN30. The SambaNova SN30 [6, 27] is another dataflow architecture, composed of eight reconfigurable dataflow units (RDUs). Each RDU has 1280 compute units, called pattern compute units (PCUs), and 1280 on-chip memory units, called pattern memory units (PMUs). Each RDU is composed of 8 tiles, where each tile has 160 PCUs and 160 PMUs. RDUs are interconnected for both model and data parallel modes, and PCUs and PMUs are assigned to computation by the compiler based on the model's computational graph. The graph is traced during compile time and the compiler performs some optimizations that may merge graph nodes (operators) and edges (data flows) depending on PCU and PMU capacity. SambaFlow (SF), is similar to PyTorch and provides a Python programming interface. A computation schedule, composed of sections that compute forward, backward, and optimization passes, specifies computation order and maps a section to tiles on the RDU. RDUs additionally have 1 TB of off-chip device memory that can hold weights, activations, and other data as sections are scheduled and de-scheduled from the tiles. In this work, we perform the SN30 evaluation using a single RDU.

2.1.3 Groq GroqChip. GroqChip [7, 8] is a combination of a dataflow and SIMD architecture that leverages both the low-latency data transfers of a dataflow architecture and the data parallelism of a SIMD architecture. The GroqChip has 230 MB of on-chip memory and 5120 compute units, or ALUs. The 230 MB is shared across a layer of ALUs, with subsequent layers retrieving results from the previous layer of ALUs. Data is streamed from this 230 MB memory to the first layer and each ALU layer receives instructions from a compiler-generated instruction schedule to perform the associated computation. Like the previously described accelerators, the computation schedule is offloaded to the compiler. Additionally, when the pipeline is fully occupied, a set of results is generated every cycle.

2.1.4 Graphcore IPU. The Graphcore Intelligence Processing Unit (IPU) [3, 24] is the most MIMD-like architecture among the accelerators considered. It is composed of 1472 compute units, called cores, and 900 MB of on-chip memory distributed evenly across all cores. Each core is capable of executing its own instruction stream and uses on-chip memory to store parameters and activations as needed. Each core can additionally communicate with other cores to retrieve data that is not directly adjacent to the core, with 4.1 TB of DDR memory ("streaming memory") for the Graphcore Bow-Pod64 system used as a means of host-device communication. In addition to PyTorch and Tensorflow, PopArt allows the implementation of custom kernels in C++ exposed to PyTorch. To handle larger models, the user can utilize more IPUs. The Graphcore Bow-Pod64 system contains 64 IPUs with custom interconnects to enable larger models, model parallelism, and data parallelism.

2.2 Lossy Compression and AI/ML

Lossy compression is a form of data compression where the decompressed data may not be exactly equal to the data before compression. This in contrast with *lossless* compression, where decompressed data is guaranteed to be equal to data before compression. Lossy compression is often utilized in scientific floating-point datasets since lossless compression yields lower compression ratios and lower compression/decompression throughput [31]. Introducing some amount of loss can vastly improve compression ratio and compression/decompression speed at the cost of data fidelity. Thus, it is imperative to understand the impact of lossy compression on dataset fidelity and post-hoc data analysis.

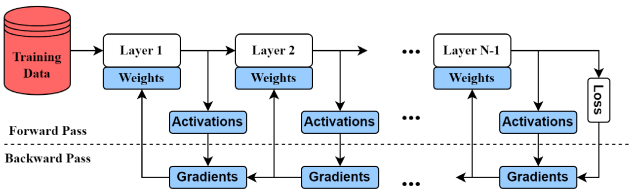


Figure 1: Neural network training procedure and compression targets. Red target indicates data being compressed/decompressed in this work. Blue targets are compressed/decompressed in other works on CPU/GPU and are future work for AI accelerators.

Some lossy compressors, such as SZ [14, 32], are *error-bounded*: the user specifies an error bound such that decompressed or reconstructed data points are within the error bound of the original data points. Other lossy compressors, such as ZFP [22], approach lossy compression with a fixed compression rate. The user can specific how rigorously the data should be compressed, with greater compression leading to more data loss. Both SZ and ZFP have been designed for scientific floating-point data and have GPU implementations, namely, cuSZ [29], cuSZp [18], and cuZFP [22]. In terms of image compression, JPEG [4] is a commonly used compression method that allows for user control of decompressed data visual quality. JPEG first applies DCT, then quantizes the resultant coefficients matrix. This matrix is subsequently compressed using some encoding method. Another form of lossy image compression is color quantization, where the range of color values is limited to some integer range [17].

Lossy compression has been widely studied as a means of reducing the memory requirements of AI model training and inference. Compression targets include model parameters, activations [13, 19], gradients [11, 21], and training data [15]. Reducing model parameter footprint allows for more efficient storage of the model itself, enabling easier deployment to memory-constrained edge devices. Compressing activations (model layer outputs generated during the forward pass and required for gradient computation) can reduce the memory required during training, ensuring that device memory is not exhausted. In distributed training environments, gradients must be communicated across interconnects or networks, incurring significant overhead. Compression can reduce gradient size, lowering distributed training communication costs. Lastly, compressing training data can lower disk storage costs, improve host-to-device communication when data is transferred to device memory, and reduce device memory consumption. Interestingly, previous work has shown that applying lossy compression on AI data does not necessarily hurt accuracy, and can even help training convergence (e.g., by avoiding overfitting).

2.3 Motivation

Data compression can benefit AI training and inference in several ways, and can be applied to training data, model weights, activations and gradients (see Fig. 1). While compression has been

Table 2: Various image datasets for benchmarking AI models

Dataset	Size	Type	Task	Sample Size
ILSVRC 2012-17 [26]	167.62 GB	General Images	Classification	3x256x256
em_graphene_sim [28]	5 GB	Electron Micrographs	Denoising	1x256x256
optical_damage_ds1 [28]	27 GB	Laser Optics	Reconstruction	3x492x656
cloud_slstr_ds1 [28]	187 GB	Remote Sensing	Pixel Segmentation	3x1200x1500

successfully used for AI training on CPU and GPU, the design of efficient compression techniques targeting novel AI accelerators has yet to be explored. However, due to their specific architectural features and programmability constraints, existing lossy compressors are not optimized for these platforms and cannot be directly ported to them. A portable compressor that can be easily integrated across accelerators would allow a variety of users to deploy compression in their AI applications. Such a compressor should have high throughput, offer reasonable compression ratios, and cause limited impact on model accuracy.

Since these AI accelerators are connected to a host machine, compression can improve host-device communication as well as reduce memory and storage footprint. Device-to-device communication cost can additionally be reduced in both model and data parallel settings. In some instances, on-chip performance can benefit from compression, such as when compute units must retrieve data residing in memory units not adjacent to them.

Previous efforts [15, 20] have explored compression of training data, while focusing primarily on image classification tasks performed on common computer vision datasets, such as CIFAR10 or ImageNet. Other downstream tasks, such as image reconstruction and segmentation within AI-for-Science applications, have not been thoroughly investigated in the context of data compression. A compressor that integrates well with tasks beyond image classification can benefit the scientific community at large. Thus, we evaluate our proposed compressor against not only a traditional image classification benchmark, but also three scientific AI benchmarks.

As of this work, the software stacks of most AI accelerators do not provide APIs to directly access activation and gradient data. Thus, we evaluate our portable compressor on training data. As described in Section 1, training data size is significantly larger than accelerator memory capacity. We expect that future software releases might allow programmers to directly access activation and gradient data stored on the device. To this end, we will discuss potential modifications to our proposed compressor to extend its effectiveness to future compression targets.

3 DESIGN AND OPTIMIZATIONS

3.1 Design Challenges

The architectural features and programming support of each accelerator poses a set of challenges and constraints on the design of the compressor.

Tensor Sizes: All the considered accelerators rely on strong compiler intervention: the compiler generates instruction streams, execution schedules, and resource allocation. For the dataflow architectures, Cerebras's CS-2 and Sambanova's SN30, the compiler places computation physically on-chip, allocating and routing compute and memory units. GroqChip's SIMD-like architecture, called the Tensor Streaming Processor (TSP), relies on the compiler to determine the execution schedule, so as to allow more of the chip area to be used for memory and ALUs instead of control logic. Graphcore's IPU also relies on its compiler to build the execution schedule. Across all compilers, models are converted to computation graphs, and data structure sizes, specifically tensor sizes, must be known at compile time. Knowing tensor sizes at compile-time allows the compiler to generate appropriate instruction schedules and allocate enough compute and memory resources. The main drawback of this is that, since all samples must have the same predefined size, the compression ratio must be known at compile time and cannot vary from sample to sample.

Programmability and Operator Support: All AI platforms provide support for popular machine learning frameworks (such as PyTorch and TensorFlow), with some platforms additionally supporting custom, lower level programming interfaces (typically extensions of C/C++). For some of these platforms, we found low level programming interfaces to have limited documentation and support. While a PyTorch programming interface is available on all considered accelerators, it does not have full operator support on each platform. For instance, PyTorch's support on SambaNova's SN30 includes the `torch.bitwise_not` API, which performs the NOT operator on each bit of the input tensor, but lacks bitwise shift operators that are integral to many variable length encoding schemes. The lack of support for PyTorch bitwise shift operators is common among many of the platforms, limiting the ability to implement existing compression schemes relying on these operators. Using lower level programming interfaces specific to each platform enables finer grain control of workload partitioning, which can improve throughput, but this reduces compressor portability, making PyTorch the best programming interface for this work.

Arithmetic Precision Support: While providing low precision floating point data types to allow faster memory accesses and computation, different accelerators support different standards. For example, for 16-bit floating point types, CS-2, GroqChip, and IPU support the FP16 format, while SN30 supports the BF16 format. In this work, we use the 32-bit floating point data type for consistency across all four accelerators and the GPU, and to ensure that data from benchmarks can be used without type conversion. While this is not the optimal choice for individual accelerators, using the 32-bit floating point format allows for a high degree of portability.

Accuracy Preservation: As is the case with all AI/ML tasks, accuracy is one of the key metrics when gauging a model. In the case of lossy compression, distortions in reconstructed data can have downstream effects on model performance. Excessive data

distortion on training data can cause the model to learn from a non-representative training set, as seen with the drop in accuracy with decreasing JPEG quality factor in [15]. In the case of activations, data loss can lead to incorrectly calculated gradients and poor convergence on the loss function minimum. A compressor designed for AI accelerators must ensure a sufficiently high degree of data fidelity is met, else compression may not be a practical means of improving training and inference.

3.2 Fundamental Design and Approach

To design a compressor for these four AI accelerators, we must balance platform limitations with strengths. Since these accelerators are designed for high throughput fully-connected layers and convolutions, matrix multiplication is highly optimized for each platform. As such, we propose a discrete cosine transform (DCT)-based compressor that primarily uses matrix multiplications to both compress and decompress input data. DCT-II, the version of the DCT specifically used in JPEG [4] and originally proposed in [10], is analogous to the discrete Fourier transform (DFT) in that both map data to frequency domain. Unlike DFT, however, DCT yields coefficients that are only real components, thus complex numbers need not be handled.

$$\begin{aligned} D_{i,j} &= \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} p(x,y) S(x,i) S(y,j) \\ S(u,v) &= \cos \frac{(2u+1)v\pi}{2N} \\ C(w) &= \begin{cases} \frac{1}{\sqrt{2}} & \text{if } w = 0 \\ 1 & \text{if } w > 0 \end{cases} \end{aligned} \tag{1}$$

$$T_{i,j} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} \cos \frac{\pi(2j+1)i}{2N} & \text{if } i > 0 \end{cases} \tag{2}$$

Eq. 1 shows the DCT-II formula. The DCT matrix D is an $N \times N$ matrix and is the result of applying the transform to an $N \times N$ input p, where $p(x, y)$ is the input data at index x, y ($x = 0, y = 0$ corresponds to the upper left-most index). $D_{i,j}$ is the DCT matrix coefficient at index i, j. Eq. 2 shows the matrix formulation of the transform. $T_{i,j}$ corresponds to the transform matrix value at index i, j. The formula $D = TAT^T$ applies the DCT to input $N \times N$ matrix A. Element $D_{0,0}$ is referred to as the DC coefficient and is representative of the average value of A.

JPEG and other compressors, such as ZFP [22], apply DCT or similar transforms to chunks of the input data to convert image or scientific floating point data to frequency domain, then subsequently discard high frequency coefficients of the resultant matrix D. Higher frequency elements of these data are less visually perceptible and more characteristic of noise. JPEG compression quantizes the DCT coefficients, introducing loss. After quantization, higher frequency DCT coefficients (coefficients with higher indices i, j), become zero. JPEG compresses the quantized DCT matrix using a variable-length encoding (VLE) scheme, such as run-length encoding (RLE) or Huffman coding. When using RLE, the DCT matrix is encoded in a zig-zag fashion (see Fig. 2).

Fig. 3 shows the proportion of nonzero DCT coefficients across all 8×8 blocks of 1000 images from the CIFAR10 dataset ([1]). JPEG uses the 8×8 block size in the standard algorithm [4], an appropriate

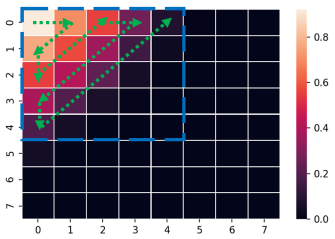


Figure 2: Two encoding methods for the DCT matrix: Zig-zag (green, dotted arrows) and Chop (blue, dashed box). RLE can be applied to data stored in zig-zag fashion to leverage many zeroes. Chop retains upper left $CF \times CF$ values ($CF = 5$ here).

size for balancing computational complexity of compression with keeping enough local information for DCT to be effective. Each row corresponds to a color channel (i.e., blue, green and red) and each column corresponds to a JPEG *quality factor*. The quality factor determines how much the DCT matrix is quantized: lower quality factor increases quantization, generating more zeroes, and increasing data loss. With more zeroes, the DCT matrix becomes more compressible since encoding schemes can leverage the lower entropy of the data to increase compression ratio.

For compatibility with the accelerator platforms, we must design the encoding stage carefully. Variable-length encoding schemes, like RLE and Huffman, are dependent on bitwise operations and result in compressed data that can vary in size dependent on the underlying data. As previously explained, certain bitwise operations are not supported and data sizes must generally be known at compile time for the accelerators. To accommodate these limitations as well as ensure high throughput compression and decompression, we compress the DCT matrix coefficients by only retaining the upper left $CF \times CF$ values. We call the overall method **DCT+Chop** since the compressor applies DCT to the input data then discards, or "chops", all DCT coefficients not part of the upper left $CF \times CF$ values.

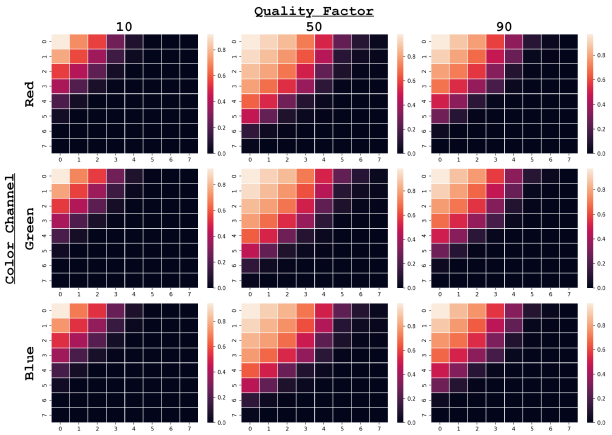


Figure 3: Heatmap of DCT coefficients after JPEG quantization with varying quality factor and color channel. Each element corresponds to percent of 8x8 blocks that have a nonzero value at that index (darker is lower). DCT is applied to 1000 32×32 images from the CIFAR10 dataset.

Accordingly, we define CF as the "chop factor". Recall that these elements (with lower i, j indices) are lower frequency coefficients that are more significant to the overall data fidelity. DCT+Chop is applied to each 8×8 chunk of the input data ($N = 8$ in Eq. 1 and 2). The compression ratio can then be computed as:

$$CR = \frac{8 * 8}{CF * CF} = \frac{64}{CF^2} \tag{3}$$

In the case of image data, each channel can be compressed or decompressed in parallel using DCT+Chop. Additionally, for batches of images, each sample of the batch can be compressed or decompressed in parallel with other samples. Thus, for a dataset of size $BD \times C \times n \times n$, where BD is the batch size, C the number of channels, and $n \times n$ the number of pixes in the image, there are $\frac{BD \times C \times n \times n}{8 \times 8}$ parallel DCT+Chop runs. With respect to color channels, the standard JPEG algorithm differs slightly: a color space transform from RGB to YCbCr is applied to leverage luminance and chrominance[4]. In an effort to keep compression fast and lightweight, we keep the original data in RGB space. The following sections will cover how DCT+Chop is algorithmically implemented for both compression and decompression.

3.3 Compression Implementation

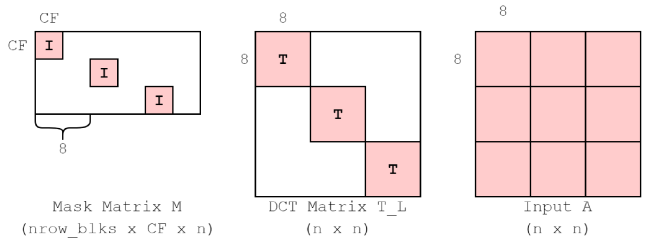


Figure 4: Example matrices used in compression/decompression. For this example, $n = 24$. `nrow_blks` is the number of blocks along the row dimension, `CF` is the chop factor.

Implementing the DCT+Chop compressor requires two matrix multiplications to perform all $\frac{BD \times C \times n \times n}{8 \times 8}$ compressions. Eq. 4 shows how the input $n \times n$ matrix A is converted to compressed matrix Y. Y has size $\frac{CF*n}{8} \times \frac{CF*n}{8}$.

$$Y = MDM^T = MT_L A T_L^T M^T = (MT_L)A(T_L^T M^T) \tag{4}$$

D is the DCT coefficients matrix, computed as $D = T_L A T_L^T$. Fig. 4 illustrates the matrices M and T_L for $n = 24$, both of which are required to compress A. The mask matrix M performs the "chop" and the transform matrix T_L performs the DCT. M is composed of $CF \times CF$ identity matrices placed in an all zero matrix. Each row of M has one "1" and only columns of M that correspond to values retained during compression have one "1". Since we are retaining the upper left $CF \times CF$ values of the 8×8 block, each $CF \times CF$ identity matrix is placed every 8 columns. T_L is a large version of the T matrix from Eq.2. T matrices are placed along the diagonal of T_L such that T is applied to every block in A.

Eq. 4 implies that $LHS = (MT_L)$ and $RHS = (T_L^T M^T)$, both of which can be computed offline, thus the compressor computes these during compilation. The overall number of FLOPs for compression is:

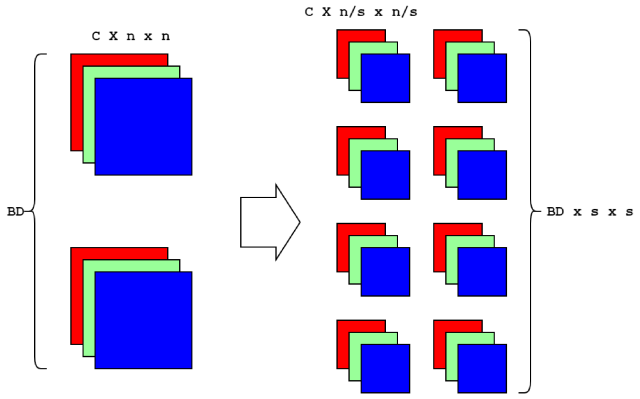


Figure 5: Partially serializing compression: The input dataset on the left with batch size $BD = 2$, channels $c = 3$, and resolution $n \times n$ is subdivided by a factor $s = 2$ to yield the samples on the right. Compression and decompression are run on each channel of each subdivision.

$$FLOPs_{compress} = \frac{2n^3 CF}{8}(\frac{CF}{8} + 1) - n^2(\frac{CF}{8} + \frac{CF^2}{64}) \tag{5}$$

The final implementation written in PyTorch is as follows:

```
Y = torch.matmul(LHS, torch.matmul(A, RHS))
```

All accelerators in this work have a PyTorch interface for programming each device, thus this design enables a high degree of portability and efficiency across AI accelerators.

3.4 Decompression Implementation

Decompression is computationally similar to compression. Eq. 6 shows how the decompressed data A' is computed:

$$A' = T_L^T M^T Y M T_L = (T_L^T M^T) Y (MT_L) \tag{6}$$

This is nearly identical to compression, except with LHS and RHS swapped. The overall number of FLOPs for decompression is:

$$FLOPs_{decompress} = \frac{2n^3 CF}{8}(\frac{CF}{8} + 1) - n^2(\frac{CF}{8} + 1) \tag{7}$$

This formula implies that decompression requires less FLOPs than compression for $CF < 8$, and less data need to be loaded to perform decompression. The final implementation written in PyTorch is as follows:

```
A_prime = torch.matmul(RHS, torch.matmul(Y, LHS))
```

Like compression, this formulation allows for all $\frac{BD \times C \times n \times n}{8 \times 8}$ decompressions to occur in parallel in the form of matrix multiplications.

3.5 Optimizations

3.5.1 Partially-serialized compression. As the $n \times n$ resolution of the input data increases, so do the sizes of matrices LHS and RHS needed for compression and decompression. Recall that LHS is of size $\frac{CF*n}{8} \times n$ and RHS is of size $n \times \frac{CF*n}{8}$. Though the aggregate on-chip memory capacity of the AI accelerators is on the scale of 100s of MB to 10s of GB, compute units that are assigned a portion of the matrix multiplication computation may have their associated local

memory exhausted as the resolution increases. For instance, one pattern memory unit (PMU) on the SN30 has 0.5 MB of space and can hold up to one, single-channel 362×362 matrix of 32-bit floating point values. Some accelerators fail to compile models requiring higher resolutions or larger tensor widths.

To reduce the memory requirements of using the DCT+Chop compressor while simultaneously supporting compression of higher resolution images/tensors, we propose a partial serialization optimization (illustrated in Fig. 5). Instead of compressing an input dataset of size $BD \times C \times n \times n$, we divide the data into chunks by a subdivision factor s. This results in $s \times s$ chunks of size $BD \times C \times \frac{n}{s} \times \frac{n}{s}$, a *LHS* matrix of size $\frac{CF*n}{8*s} \times \frac{n}{s}$, and a *RHS* matrix of size $\frac{n}{s} \times \frac{CF*n}{8*s}$. The $s \times s$ chunks are processed serially and the memory requirements of the compressor are reduced by a factor of $s \times s$. Both the SN30 RDU and Graphcore IPU can benefit from this optimization since both compilers require tiles to have enough memory to receive high resolution data along with the matrices required for multiplication.

3.5.2 Graphcore: torch.scatter and torch.gather. Recall the zig-zag encoding method shown in Fig. 2. Zig-zag encoding is an efficient means of compressing each DCT coefficient matrix since diagonals (from bottom left to top right) become less important to data fidelity as we move along the main diagonal. As such, more values are quantized to zero and are not needed to reconstruct high fidelity decompressed data. Due to operator support limitations and to preserve high throughput, DCT+Chop is used on most accelerators, but some values are unnecessarily stored. The IPU supports two APIs that can allow for efficient storage of the upper left *triangle* instead of upper left *square*: `torch.scatter` and `torch.gather`.

The proposed optimization is illustrated in Fig. 6. Using precomputed indices of the upper left triangle, `torch.gather` can collect the upper left triangle values and discard values not encapsulated in the triangle. Since the size of data must be known at compile time, the indices can be computed at compile time and need not be stored. For a given chop factor CF, the number of retained values for each 2-D matrix can be decreased from $\texttt{nblks}*CF*CF$ to $\texttt{nblks}*CF*(CF+1)/2$, increasing the compression ratio by a factor of $\frac{2CF}{CF+1}$. To decompress the data, `torch.scatter` returns the retained values to their original position in the decompressed matrix, given the indices. In the final implementation, compression first runs DCT+Chop compression then calls `torch.gather` with the upper left triangle indices. Decompression first calls `torch.scatter`, then runs DCT+Chop decompression.

`torch.scatter` and `torch.gather` are examples of operators not yet supported across all accelerators. However, these types of APIs can be utilized in the context of sparse operations and quantization, thus we expect more support as development environments evolve. Other operators relating to sparsity and quantization could also be supported in the future, improving the ability to compress data.

4 EXPERIMENTAL EVALUATION

4.1 Methodology

To evaluate our compressor design, we perform an accuracy and throughput analysis across a variety of compressor configurations

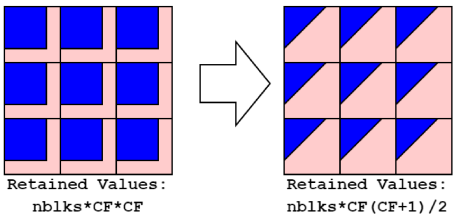


Figure 6: `torch.scatter/torch.gather` optimization: retain the upper left triangle of values instead of upper left $CF \times CF$ values. $nblks = 9$ for this example.

on all platforms listed in Table 1. We use one of each accelerator device (i.e., one CS-2 chip, one SN30 RDU, one GroqChip, and one IPU). The three compressor designs we test are baseline **DCT+Chop (DC)**, **Partial Serialization (PS)**, and **torch.scatter/gather (SG)**, with chop factor CF varying from 2 to 7. The compressor is implemented in PyTorch 2.0.1, with compression and decompression compiled separately for each accelerator. We use Cerebras Release 2.0.1, SambaFlow 1.17, GroqFlow 4.2.1, and PopTorch 3.3.0.

First, we evaluate the impact of lossy compression on accuracy: the configurations are tested with the benchmarks in Table 3 from the SciML benchmarking suite [28], each of which is trained for 30 epochs and compared against a no-compression baseline. For the *classify* benchmark, we report training loss and test accuracy, and for the other benchmarks, we report training and test loss as these are the metrics of value specified in [28]. Next, we evaluate the speed of each compressor configuration. Datasets to compress are varied in resolution from 32×32 to 512×512 and batch size is varied from 10 to 5000. We collect the average compression and decompression time of 100 runs. Execution time includes host-device communication. Compilation time is omitted from these results. Depending on the network size, accelerator, and optimization level, compilation is a one-time cost that can range on the scale of minutes to hours, amortized over the course of more training epochs/runs.

4.2 Results

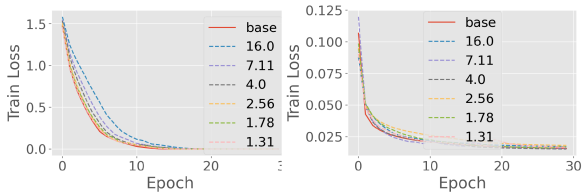
4.2.1 Impact on Accuracy. Fig. 7 plots the training loss, while Fig. 8 plots the test loss percent difference against baseline ("base"). Note that for Fig. 8a, accuracy difference instead of loss difference is reported. While for loss difference lower is better, for accuracy difference higher is better. During training, each batch is first compressed and then decompressed, so that increasing levels of loss and compression ratio can be studied against model accuracy. Each series of each plot corresponds to a fixed compression ratio using DCT+Chop, with the corresponding compression ratio reported in the legend.

As shown in Fig. 7, after compression and decompression of training data, the model is still able to follow a similar path to convergence and can achieve very similar training loss to baseline. For **em_denoise**, **optical_damage**, and **slstr_cloud**, training loss across all compression ratios closely follows baseline, while for **classify**, increasing compression ratio, which increases data loss, results in a lag in the loss curve. More loss leads to sub-optimal convergence.

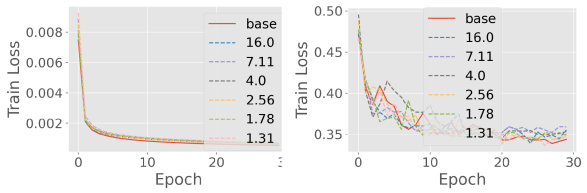
Fig. 8 indicates that for both **em_denoise** and **slstr_cloud**, DCT+Chop compression can ensure strong accuracy/fidelity close

Table 3: Tests performed during evaluation and associated parameters

Test	Dataset	Task	Network	Sample Size	Training Params.
classify	CIFAR10	Classify images into 10 classes	ResNet34	3x32x32	BS=100, LR=0.001
em_denoise	em_graphene_sim	Denoise electron micrographs	Deep Encoder-Decoder	1x256x256	BS=32, LR=0.0005
optical_damage	optical_damage_ds1	Reconstruct laser optics images	Autoencoder	1x200x200	BS=2, LR=0.0005
slstr_cloud	cloud_slstr_ds1	Identify pixels that are clouds	UNet	9x256x256	BS=4, LR=0.0005

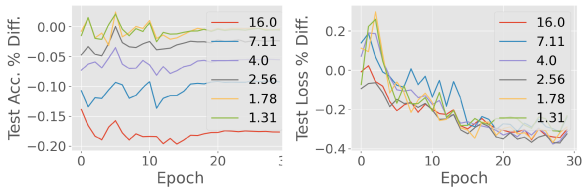


(a) classify (b) em_denoise

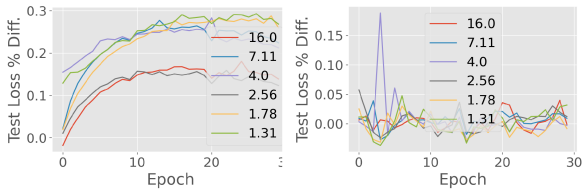


(c) optical_damage (d) slstr_cloud

Figure 7: Average training loss per epoch of four benchmarks. Each series corresponds to a different compression ratio of the DCT+Chop compressor, base is no compression.



(a) classify (b) em_denoise

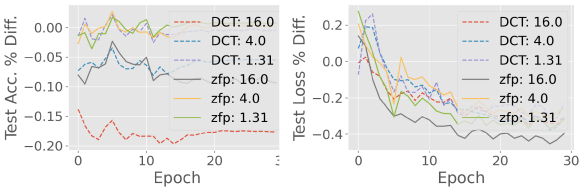


(c) optical_damage (d) slstr_cloud

Figure 8: Average test loss percent difference from no compression case per epoch of four benchmarks. Each series corresponds to a different compression ratio of the DCT+Chop compressor. For classify benchmark, test accuracy percent difference is reported instead.

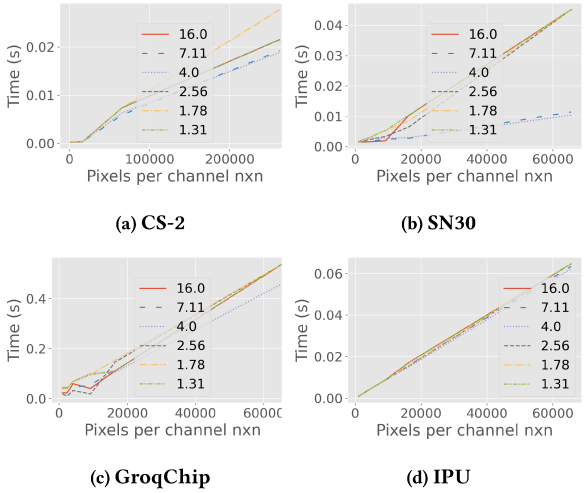
to baseline. Surprisingly, compression for **em_denoise** actually improves accuracy (reduces test loss more than baseline). This is likely due to the denoising effect of removing high frequency elements of the DCT coefficients matrix since these elements tend to be noise. For **optical_damage**, the percent difference is higher, but the actual loss values are close to each other. For mean squared error (MSE) loss, the baseline loss at epoch 30 is 0.0060, compared to 0.00068 for compression ratio of 16.0. Given that the **optical_damage** benchmark trains a model to reconstruct undamaged laser optics images such that reconstructed optics with damage have high MSE loss, this relatively limited performance gap is acceptable for the task at hand. For the **classify** benchmark, the impact on accuracy of varying compression ratios is more stratified: increasing the compression ratio and loss clearly has a direct, negative impact on accuracy. For $CF = [5, 6, 7]$ ($CR = 2.56, 1.78, 1.31$, respectively), accuracy drop is less than three percent, still within a reasonable range considering the low resolution of samples.

While ZFP cannot be ported onto the considered AI accelerators, we compare its accuracy with that of our compressor on CPU. Fig. 9 plots the test accuracy/loss percent difference from baseline for DCT+Chop and ZFP. The compression ratio of each series is reported in the legend. For the **classify** benchmark, ZFP can generally achieve higher compression ratio for comparable accuracy. For **em_denoise**, ZFP performs much closer to DCT+Chop, only slightly outperforming DCT+Chop for a compression ratio of 16. Across both compressors, **em_denoise** enjoys a performance boost with the integration of compression on training data.



(a) classify (b) em_denoise

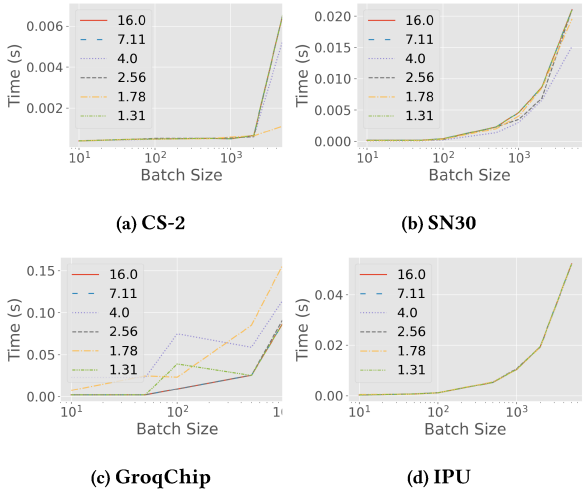
Figure 9: Average test accuracy/loss percent difference from no compression case per epoch of two benchmarks, comparing DCT+Chop against ZFP. Each series corresponds to a different compression ratio of DCT+Chop or ZFP. For classify benchmark, test accuracy percent difference is reported instead.



(a) CS-2 (b) SN30

(c) GroqChip (d) IPU

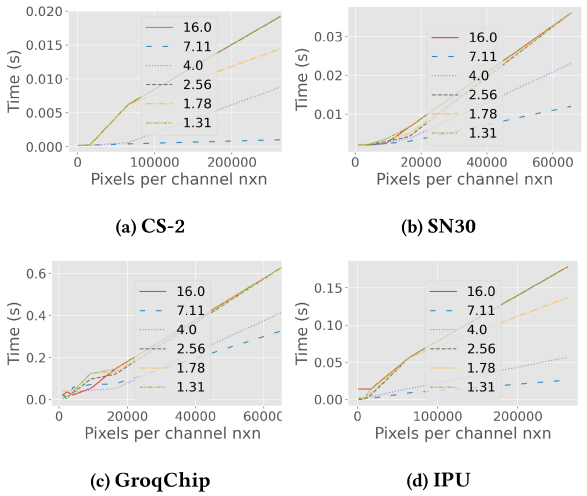
Figure 10: Compression time for DCT+Chop across four accelerators for varying resolution. Each series corresponds to a compression ratio.



(a) CS-2 (b) SN30

(c) GroqChip (d) IPU

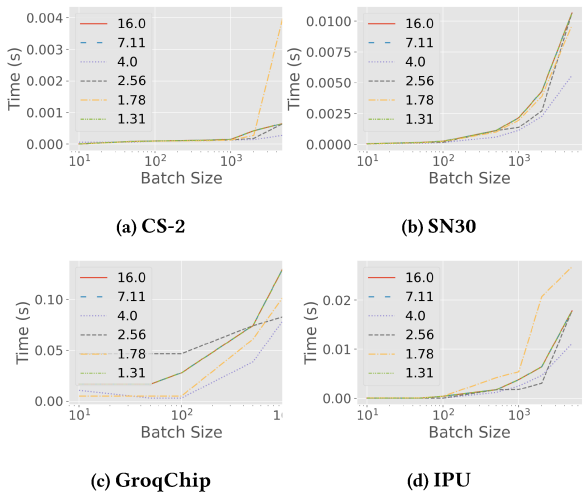
Figure 12: Compression time for DCT+Chop across four accelerators for varying batch size. Each series corresponds to a compression ratio.



(a) CS-2 (b) SN30

(c) GroqChip (d) IPU

Figure 11: Decompression time for DCT+Chop across four accelerators for varying resolution. Each series corresponds to a compression ratio.



(a) CS-2 (b) SN30

(c) GroqChip (d) IPU

Figure 13: Decompression time for DCT+Chop across four accelerators for varying batchsize. Each series corresponds to a compression ratio.

4.2.2 Compression and Decompression Speed. Fig. 10 and Fig. 11 plot the compression and decompression time for all accelerators with varying *resolution* for 100, 3 channel samples. Fig. 12 and Fig. 13 plot the compression and decompression time with varying *batch size* for 3 channel, 64 × 64 resolution samples. Each series corresponds to a different *CF*, with the associated compression ratio listed in the legend. Recall that these times are with respect to the host: they include data transfer time and thus *overestimate* the time it takes to perform compression/decompression if the compressor is integrated in a training or inference pipeline. Note that compilation for 512×512 resolution fails for SN30 and GroqChip due to an out-of-memory error on-chip.

CS-2: The CS-2 has the highest compression and decompression throughput across all of the accelerators, generally ranging from 16 to 26 GB/s. This is largely due to the CS-2 wafer size: the number of on-chip compute units is orders of magnitude larger than the other accelerators, with the obvious tradeoff being power consumption. Compression typically takes longer than decompression, as expected since compression requires more floating-point operations and, more importantly, compression requires loading the $n \times n$ input matrix instead of the $\frac{n*CF}{8} \times \frac{n*CF}{8}$ matrix required for decompression. Additionally, there is a wider spread of decompression times compared to compression for varying compression ratios, with higher compression ratio (lower *CF*) having significant speedup. Since the input to be loaded onto the device is dependent on *CF* for

decompression, lower CF requires less data to be transferred to the device, whereas for compression, input data is the same size across varying CF. As batch size increases, the CS-2 performance does not change significantly, until batch size surpasses 2000. This could be due to the amount of samples per batch beginning to bound performance at batch size of 2000 since the dataflow pipeline is fully occupied. To ensure that compression/decompression is not a bottleneck, the compression throughput should be at least as high as the throughput of the forward and backward passes. In the case of a ResNet34 network processing batches of size 100 from CIFAR10, the CS-2 can process ≈205 samples per second during training. Decompression is significantly faster, running at ≈330,000 samples per second. As such, our design allows the CS-2 pipeline to remain full and not stall from compression/decompression. Thus, the overhead of the compressor is masked in the dataflow pipeline. Even with more optimized network implementations, compression and decompression throughput is several orders of magnitude higher than batch processing time, making compression a practical tool in training on CS-2 as well as other dataflow architectures.

SN30: As with the CS-2, decompression is generally faster than compression on the SN30 RDU. Compression ratios of 4.0 and 7.11 perform best for compression and decompression, while all other compression ratios perform similarly. Interestingly, the highest compression ratio, 16.0, is slower than both 4.0 and 7.11, suggesting that even though less FLOPs are required (per Eq. 5 and Eq. 7) and less memory needs to be loaded, there is some runtime overhead involved with high compression ratio. This behavior could be attributed to the memory architecture: the RDU has higher throughput memory accesses and transfers on fewer, large tensors compared to many small tensors. Small tensors generated during compression and operated on during decompression incur runtime overhead since they may not be mapped to nearby memory locations. Compression and decompression both have a throughput of around 7 to 10 GB/s, which includes data transfer overhead from the PCIe 4.0 connection. Note that compilation fails for 512×512 resolution since the PMUs cannot fit the entire output matrix along with matrices required for compression/decompression. The SN30 execution time is linearly related to batch size since the computational complexity scales linearly with increasing batch size. Decompression is significantly faster than the forward and backward pass (throughput of decompression is ≈220,000 samples per second against backward/forward pass throughput of ≈570 samples per second for ResNet34 on CIFAR10). Since SN30 has a dataflow architecture, decompression overhead is not a bottleneck during training and the compressor adds no significant change in runtime. Additionally, the compressor sections for 256×256 images utilize ≈3% or less of the PCUs on one RDU, leaving most of the RDU available for other computation.

GroqChip: On GroqChip, compression has a low performance variance: across all compression ratios, the throughput does not vary significantly (≈150 MB/s). For decompression, performance becomes more stratified depending on the compression ratio, but across the board performs better than compression (≈200 MB/s for decompression). Since the GroqChip is a pipelined SIMD architecture, compression and decompression can be overlapped with other operations, specifically model operations. The compressor fails to compile for 512×512 resolution, due to the limited size of on-chip memory with respect to input data and compressor matrices as well as the limits of the matrix-matrix multiplication modules on-chip (which can handle up to 320×320 8-bit integer matrix multiplications [9]). When varying batch size, the GroqChip fails to compile beyond a batch size of 1000 since on-chip memory is exhausted. Compression and decompression times are more spread compared to other accelerators depending on the compression ratio since for the CS-2 and SN30, pipeline depth is a significant factor in scaling with batch size, while for GroqChip, the computational complexity and memory access time are more significant factors.

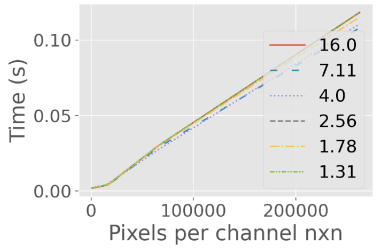


Figure 14: Decompression time for DCT+Chop on A100 GPU for varying resolution. Each series corresponds to a compression ratio.

IPU: The IPU has the least variance for compression throughput across compression ratios (≈1.2 GB/s average throughput for compression). For decompression, the IPU enjoys significant throughput improvement for higher compression ratios (up to 21 GB/s), while lower compression ratios perform modestly (≈ 2 GB/s). As with the other platforms, the IPU execution time has a linear relationship with the number of pixels, suggesting that given the FLOPs formulas, the compressor is memory-bounded and highly dependent on on-chip memory access speed rather than the computation itself. Like the CS-2 and SN30, performance relative to batch size is similar across compression ratios. Decompression times are more varying, suggesting on-chip memory throughput is a greater bottleneck.

Comparison with GPU: We perform a DCT+Chop decompression timing evaluation on an NVIDIA A100 GPU with PCIe 4.0 connection. Fig. 14 plots the decompression time for varying compression ratio, corresponding to each series. Compression performance trends are similar, thus we omit compression speed plots. The A100 GPU performs decompression at ≈2.5 GB/s, with little variation across each compression ratio. Both the CS-2 and SN30 RDU outperform the A100, while a single GroqChip and single IPU are outperformed by the A100. However, both the GroqChip and IPU are generally deployed with other GroqChips or IPUs. For instance, the Graphcore Bow-Pod64 contains 64 IPUs [3] and the GroqNode has eight GroqCards, each of which contains a GroqChip [8]. Thus, the CS-2 and SN30 RDU on their own can outperform the A100 for DCT+Chop, and GroqChip and IPU rely on scalability to outperform GPU.

Key Takeaways:

- Compression generally is slower than decompression, likely due to more data movement and more FLOPs.
- Compression and decompression time is linearly related to pixel count. This relationship is likely due to the off-chip and on-chip memory performance, which are linearly related to data size.
- Higher compression ratios often have faster decompression.

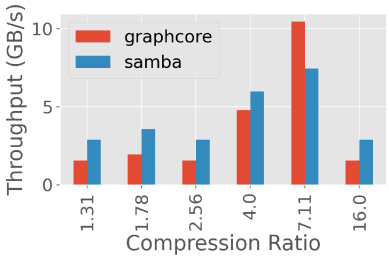


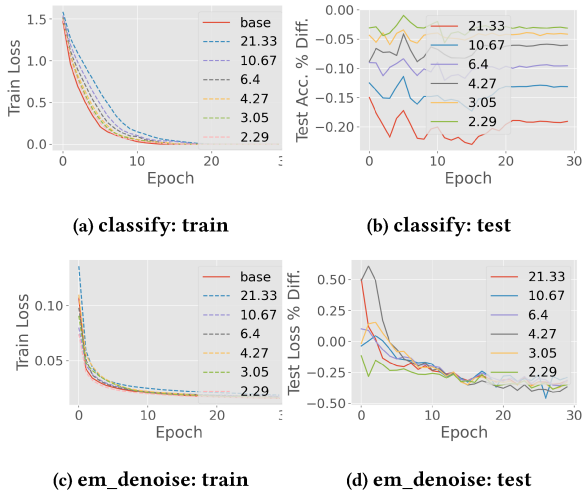
Figure 15: Decompression throughput for IPU ("graphcore") and SN30 RDU ("samba") using partial serialization $s = 2$ on 100, 3 channel, 512×512 images. Compression ratio is varied on x-axis ($CF = 7, 6, 5, 4, 3, 2$ from left to right).

- Execution time and batch size are linearly related since 1) dataflow architectures begin to fully fill their pipelines and 2) computational complexity of compression and decompression is linearly related to the batch size. A larger batch size requires more matrix multiplications, but not more complex matrix multiplications.
- While the CS-2 and SN30 RDU perform compression/decompression faster than the A100, IPU, and GroqChip, power differences are *not* accounted for in this evaluation. Thus, we cannot directly compare performance differences between accelerators.

4.2.3 Optimization 1: Partial Serialization. Fig. 15 plots the decompression throughput of the partial serialization optimization with $s = 2$ for 100, 3 channel, 512×512 images on the IPU and SN30. With a serialization factor $s = 2$, the 512×512 images are chunked into four, 256×256 pixel chunks, thus four times as many decompression runs are issued. Compared to the decompression timing from 256×256 images reported in Fig. 11, the throughput results in Fig. 15 indicate only a 2.5-3.8× slowdown for the SN30 and 2.6-3.7× slowdown for the IPU when moving to partially-serialized 512×512 images. Considering that the number of matrix multiplications is four times that of no serialization and that 512×512 images are four times as large as 256×256 images, these slowdowns are better than expected. The Graphcore IPU successfully ran no-serialization decompression for 512×512 images and compared to $s = 2$ partial serialization, no-serialization is only 1-8% faster.

4.2.4 Optimization 2: Graphcore `torch.scatter/gather`. Fig. 16 plots the training loss and test accuracy/loss difference from no compression with $CF = [2, 7]$ using the `torch.scatter/gather` optimization for two benchmarks. The compression ratios are reported in the legend and results are collected for 30 epochs with the same training parameters listed in Table 3. For **classify** benchmark, there is a slight drop in accuracy compared to DCT+Chop, typically of 1-2% for equivalent CF. For **em_denoise** , SG achieves less difference in test loss compared to DCT+Chop, and can even improve performance. For instance, SG can be nearly 0.5% lower than baseline compared to DCT+Chop being nearly 0.4% lower. We observe similar behavior between SG and DCT+Chop for **optical_damage** and **slstr_cloud**. As with DCT+Chop, increasing CF generally leads to reduction in data quality and lower model performance.

Fig. 17 plots the decompression throughput of DCT+Chop, "dct", against SG, "opt", for varying CF. 100, 3 channel, 32×32 images are



(a) classify: train **(b) classify: test**

(c) em_denoise: train **(d) em_denoise: test**

Figure 16: Training loss (left column) and test accuracy/loss percent difference (right column) of `torch.scatter/gather` optimization compared to no compression baseline.

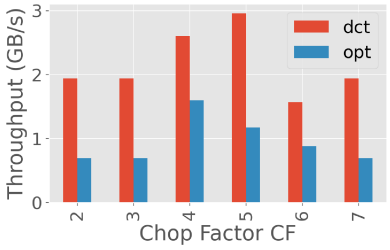


Figure 17: Decompression throughput for `torch.scatter/gather` optimization ("opt") against DCT+Chop ("dct") on Graphcore IPU for 100, 3 channel, 32×32 images.

decompressed on a single IPU. Recall that the compression ratio is computed as $\frac{64}{CF^2}$ and $\frac{64}{CF(CF+1)/2}$ for DCT+Chop and SG, respectively. SG is 1.5-2.7× slower than DCT+Chop and has compression ratio improvement of 1.3-1.75× DCT+Chop across varying CF. As such, the tradeoff between compression ratio and throughput is not directly related and attaining higher compression ratios with SG requires greater compromise of compression/decompression speed.

5 RELATED WORK

[15] and [20] study the impact of lossy compression of training data on image classification accuracy. Dodge and Karam [15] find that, when using JPEG, a quality factor as low as 10 can still yield image classification accuracy close to no compression baseline across four different architectures. Joseph et al. [20] explore using ZFP as a training data compressor, finding that across six of seven tested networks, ZFP can achieve higher compression ratio for a given accuracy target. Both works focus on image classification and have not factored in platform-specific considerations, an area which we explore in our work.

Existing works have explored lossy compression of other targets, such as gradients [21] and activations [13][19]. These works seek to use compression to reduce training memory footprint and improve distributed training scenarios and use GPU as the primary training platform. These compression methods, while effective on GPU, cannot yet be ported to the AI accelerators since 1) access to activations and gradients is limited or not available, and 2) operators integral to each compressor are not yet supported, specifically bitwise and bitshift operations for encoding stages.

In the space of AI accelerators, Emani et al. [16] perform a comparison study of large language models (LLMs) across a variety of accelerators, including the NVIDIA A100, Cerebras CS-2, SambaNova SN30, and Graphcore Bow-Pod64 system. Their work evaluates each accelerator with the same set of LLM benchmarks, finding that across all accelerators, memory reduction techniques are "of paramount significance" [16] since the memory size of weights, activations, and data is often the limiting factor in fitting a model on a single device.

For compatibility with AI accelerators, one approach for compressor design is using neural networks as the compressor itself. Liu et al. [23] propose an autoencoder integration for SZ, using the autoencoder as a predictor that can improve quantization efficiency of SZ. Lu et al. [25] propose a transformer network that builds on top of a variational autoencoder to compress images. Both designs utilize neural network architectures, which are supported across all AI accelerators, and the encoding stage could be modified such that a fixed length encoding that does not use bitwise operations is used. If these designs were successfully implemented, however, compression and decompression throughput would be significantly slower than our design. Both designs are deep networks or transformers that require many more operations compared to our two matrix multiplication algorithm. As such, while AI-based compressors could be implemented on these accelerators, speed would be a concern.

6 CONCLUSION AND FUTURE WORK

Conclusion: In this work, we have designed a PyTorch-based lossy compressor that can run across four different novel AI accelerators with little programmer effort. Our compressor can achieve speeds ranging from 100s of MB/s on GroqChip, to up to 26 GB/s on the CS-2. For dataflow architectures, these speeds are significantly faster than the processing time for equivalent data sizes, allowing compression and decompression to be masked in the dataflow pipelines. Additionally, loss introduced from our compressor has a limited impact on test loss and accuracy, leading to reductions in model performance of generally less than 3%.

Future Work: At the compressor-level, more platform-specific optimizations can be explored to generate a library of tailored compressors. In terms of core compressor design changes, we can test using the ZFP block transform instead of DCT-II, especially as compression targets change. Since the training data we have evaluated in this work has been image data, DCT-II is suitable as a transform, but the ZFP block transform can be more applicable to general scientific floating point datasets. At the target-level, the compression targets can evolve as the development ecosystem of each platform evolves. Weights, activations, and gradients all have the opportunity to be compressed, reducing model footprint and training memory utilization. Changing targets can lead to compressor design changes, such as in [19] where SZ is modified to accurately reconstruct zero-valued activations. If a similar compression procedure were implemented, this would introduce sparsity, opening the door to an exploration of sparse matrix operations on AI accelerators.

ACKNOWLEDGMENT

This research was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357, and supported by the National Science Foundation under Grant OAC-2003709, OAC-2104023, OAC-2311875. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

REFERENCES

[1] [n. d.]. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html
[2] [n. d.]. Explore Cerebras Documentation — Cerebras Developer Documentation. https://docs.cerebras.net/en/latest/
[3] [n. d.]. Graphcore Documents — Graphcore Documents. https://docs.graphcore.ai/en/latest/
[4] [n. d.]. JPEG - JPEG 1. https://jpeg.org/jpeg/index.html
[5] [n. d.]. Product - System. https://www.cerebras.net/product-system/
[6] [n. d.]. SambaNova :: SambaNova Documentation. https://docs.sambanova.ai/home/latest/index.html
[7] 2023. GroqCard™ Accelerator - Groq. https://wow.groq.com/groqcard-accelerator/ Section: Blog.
[8] 2023. groq/groqflow. https://github.com/groq/groqflow original-date: 2022-08-08T23:46:56Z.
[9] Ibrahim Ahmed, Sahil Parmar, Matthew Boyd, Michael Beidler, Kris Kang, Bill Liu, Kyle Roach, John Kim, and Dennis Abts. 2022. Answer Fast: Accelerating BERT on the Tensor Streaming Processor. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 80–87. https://doi.org/10.1109/ASAP54787.2022.00022 ISSN: 2160-052X.
[10] N. Ahmed, T. Natarajan, and K.R. Rao. 1974. Discrete Cosine Transform. *IEEE Trans. Comput.* C-23, 1 (Jan. 1974), 90–93. https://doi.org/10.1109/T-C.1974.223784 Conference Name: IEEE Transactions on Computers.
[11] Dan Alistarh, Demjan Grubic, Jerry Z. Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: communication-efficient SGD via gradient quantization and encoding. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1707–1718.
[12] Alexander Brace, Michael Salim, Vishal Subbiah, Heng Ma, Murali Emani, Anda Trifa, Austin R. Clyde, Corey Adams, Thomas Uram, Hyunseung Yoo, Andew Hock, Jessica Liu, Venkatram Vishwanath, and Arvind Ramanathan. 2021. Stream-AI-MD: Streaming AI-Driven Adaptive Molecular Simulations for Heterogeneous Computing Platforms. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Geneva, Switzerland) *(PASC '21)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. https://doi.org/10.1145/3468267.3470578
[13] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. 2021. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *Proceedings of the 38th International Conference on Machine Learning*. PMLR, 1803–1813. https://proceedings.mlr.press/v139/chen21z.html ISSN: 2640-3498.
[14] Sheng Di and Franck Cappello. 2016. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 730–739. https://doi.org/10.1109/IPDPS.2016.11 ISSN: 1530-2075.
[15] Samuel Dodge and Lina Karam. 2016. Understanding how image quality affects deep neural networks. In *2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX)*. 1–6. https://doi.org/10.1109/QoMEX.2016.7498955
[16] Murali Emani, Sam Foreman, Varuni Sastry, Zhen Xie, Siddhisanket Raskar, William Arnold, Rajeev Thakur, Venkatram Vishwanath, and Michael E. Papka.

2023. A Comprehensive Performance Study of Large Language Models on Novel AI Accelerators. https://doi.org/10.48550/arXiv.2310.04607 arXiv:2310.04607 [cs].

[17] Paul Heckbert. 1982. Color image quantization for frame buffer display. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques* (Boston, Massachusetts, USA) *(SIGGRAPH '82)*. Association for Computing Machinery, New York, NY, USA, 297–307. https://doi.org/10.1145/800064.801294

[18] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3581784.3607048

[19] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. COMET: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. *Proceedings of the VLDB Endowment* 15, 4 (Dec. 2021), 886–899. https://doi.org/10.14778/3503585.3503597

[20] Vinu Joseph, Nithin Chalapathi, Aditya Bhaskara, Ganesh Gopalakrishnan, Pavel Panchekha, and Mu Zhang. 2020. Correctness-preserving Compression of Datasets and Neural Network Models. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. 1–9. https://doi.org/10.1109/Correctness51934.2020.00006

[21] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2018. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. https://doi.org/10.48550/arXiv.1802.07389 Issue: arXiv:1802.07389 arXiv:1802.07389 [cs, stat].

[22] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (Dec. 2014), 2674–2683. https://doi.org/10.1109/TVCG.2014.2346458

[23] Jinyang Liu, Sheng Di, Kai Zhao, Sian Jin, Dingwen Tao, Xin Liang, Zizhong Chen, and Franck Cappello. 2021. Exploring Autoencoder-Based Error-Bounded Compression for Scientific Data. *CoRR* abs/2105.11730 (2021). arXiv:2105.11730 https://arxiv.org/abs/2105.11730

[24] Graphcore Ltd. [n. d.]. IPU Processors. https://www.graphcore.ai/products/ipu

[25] Ming Lu, Peiyao Guo, Huiqing Shi, Chuntong Cao, and Zhan Ma. 2021. Transformer-based Image Compression. http://arxiv.org/abs/2111.06707 arXiv:2111.06707 [cs, eess].

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[27] SambaNova Systems. [n. d.]. SambaNova Systems DataScale® | Our Products. https://sambanova.ai/products/datascale

[28] Jeyan Thiyagalingam, Juri Papay, Kuangdai Leng, Samuel Jackson, Mallikarjun Shankar, Geoffrey Fox, and Tony Hey. 2021. SciML-Bench: A Benchmarking Suite for AI for Science. https://github.com/stfc-sciml/sciml-bench

[29] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/3410463.3414624

[30] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. https://doi.org/10.48550/arXiv.2205.01068 arXiv:2205.01068 [cs].

[31] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D. Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1643–1654. https://doi.org/10.1109/ICDE51399.2021.00145

[32] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2020. Significantly Improving Lossy Compression for HPC Datasets with Second-Order Prediction and Parameter Optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/3369583.3392688

[33] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. http://arxiv.org/abs/2303.18223 arXiv:2303.18223 [cs].

[34] Maxim Zvyagin, Alexander Brace, Kyle Hippe, Yuntian Deng, Bin Zhang, Cindy Orozco Bohorquez, Austin Clyde, Bharat Kale, Danilo Perez-Rivera, Heng Ma, Carla M. Mann, Michael Irvin, J. Gregory Pauloski, Logan Ward, Valerie Hayot-Sasson, Murali Emani, Sam Foreman, Zhen Xie, Diangen Lin, Maulik Shukla, Weili Nie, Josh Romero, Christian Dallago, Arash Vahdat, Chaowei Xiao, Thomas Gibbs, Ian Foster, James J. Davis, Michael E. Papka, Thomas Brettin, Rick Stevens, Anima Anandkumar, Venkatram Vishwanath, and Arvind Ramanathan. 2022. GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *bioRxiv: The Preprint Server for Biology* (Nov. 2022), 2022.10.10.511571. https://doi.org/10.1101/2022.10.10.511571