SEP: A General Lossless Compression Framework with Semantics Enhancement and Multi-Stream Pipelines

Meng Wan[1,2†], **Rongqiang Cao**[1,3†], **Yanghao Li**[1,3†], **Jue Wang**[1,3*], **Zijian Wang**[1], **Qi Su**[4], **Lei Qiu**[5], **Peng Shi**[2], **Yangang Wang**[1,3], **Chong Li**[1,3]
[1]Computer Network Information Center, Chinese Academy of Sciences, Beijing, China
[2]University of Science and Technology Beijing, Beijing, China
[3]University of Chinese Academy of Sciences, Beijing, China
[4]Peking University, Beijing, China
[5]Zhejiang Normal University, Jinhua, China
{wanmengdamon,yhli,wangzj,lichong}@cnic.cn, {caorq,wangjue,wangyg}@sccas.cn,
qiisuu@stu.pku.edu.cn, userqiul@gmail.com, pshi@ustb.edu.cn

Abstract

Deep-learning-based lossless compression is of immense importance in real-world applications, such as cold data persistence, sensor data collection, and astronomical data transmission. However, existing compressors typically model data using single-byte symbols as tokens, which makes it hard to capture the inherent correlations and cannot effectively utilize the parallel capabilities of GPUs and multi-core CPUs. This paper proposes SEP, a novel lossless compression framework for most time-series backbone neural networks. We first introduce a semantic enhancement module to capture the complex intra-patch relationships of binary byte streams. To improve the compression speed, we design multi-stream pipelines that dynamically assign parallel tasks to GPU streams and multi-cores. We further propose a novel GPU memory optimization strategy, which reuses GPU memory by a shared pool across streams. We conduct experiments on seven real-world datasets and the results demonstrate that our SEP framework outperforms state-of-the-art compressors with an average speed improvement of 30.0% and an average compression ratio gain of 5.1%, which is further elevated to 7.6% with the use of pre-training models. The GPU memory footprint is reduced by as high as 63.1% and by an average of 36.2%. The source code is available at: https://github.com/damonwan1/SEP.

1 Introduction

Lossless data compression reduces data size while preserving all the original information, which has been widely used in various fields, including data storage [Lagar-Cavilla *et al.*, 2019], data transmission [Yang *et al.*, 2023], to name a few. The performance of lossless compression method is mainly examined using compression ratio and compression speed [Schiopu and Munteanu, 2020; Chandra and Hsu, 2014]. To improve both factors, we construct a general lossless compression framework with semantics enhancement and multi-stream pipelines.

The fundamental technique of lossless compression is entropy coding, which includes methods such as Huffman coding [Knuth, 1985; Moffat, 2019] and arithmetic coding [Rissanen and Langdon, 1979]. Given a system that generates symbols with certain probability distributions, according to Shannon's source coding theorem [Affeldt *et al.*, 2014], the ideal code length $\mathrm{I}(x)$ of the next symbol x is expressed as follows:

$$\mathrm{I}(x) = -log_2\left(\mathrm{p}\left(x\right)\right) \tag{1}$$

where $\mathrm{p}(x)$ is the probability of the symbol to be encoded. Entropy coder requires the statistical model to predict the statistics of the system [Zhang *et al.*, 2021]. Based on Equation 1, the entropy coder uses shorter code for more predictable symbols, which results in a higher compression ratio [Sullivan *et al.*, 2012]. On top of the entropy coder, the deep-learning-based lossless compressors use neural networks to predict the statistical characteristics of the system [Mao *et al.*, 2022b], which potentially gives higher $\mathrm{p}(x)$ and therefore higher compression ratio [Wu *et al.*, 2016]. An inaccurate prediction of the statistical characteristics can result in an extremely small $\mathrm{p}(x)$, such that the symbol x can still be losslessly compressed but with a very low compression ratio.

To improve both compression ratio and compression speed of deep-learning-based lossless compressors, the existing research focus on different networks. The recent deep-learning-based lossless compressors are mainly built based on PyTorch [Mentzer *et al.*, 2019]. In Table 1, we present the various characteristics of several deep-learning-based compressors. Compressors like tensorflow-compress [Knoll, 2016; Contoli and Lattanzi, 2023] and DecMac [Liu *et al.*, 2019] use long-short-term memory (LSTM) to capture long-term dependencies within the input data stream. The latest research demonstrates that the mechanisms of transformer and multi-layer perception (MLP) can achieve even more accu-

*Corresponding author

Name	Network	GPU Opt.	Semantic Enhancement	Framework	Year	Compression Ratio	Speed (KB/min)
LSTM-compress [Knoll, 2016]	LSTM	×	×	Tensorflow	2016	4.32	243
DecMac [Liu *et al.*, 2019]	LSTM	×	×	Tensorflow	2019	5.80	-
NNCP [Bellard, 2019]	Transformer	×	×	Tensorflow	2020	5.85	183
Dzip [Goyal *et al.*, 2021]	RNN	×	×	PyTorch/Tensorflow	2021	4.35	429
OREO [Mao *et al.*, 2022a]	MLP	×	×	PyTorch	2022	5.61	1810
TRACE [Mao *et al.*, 2022b]	Transformer	×	×	PyTorch/Tensorflow	2022	5.20	1656
PAC(SOTA) [Mao *et al.*, 2023]	MLP	Cache	×	PyTorch/LibTorch	2023	5.82	1570
SEP+PatchTST (ours)	Transformer	Pipelines	✓	PyTorch	2023	5.34	2070
SEP+TRACE (ours)	Transformer	Pipelines	✓	PyTorch	2023	5.52	**2130**
SEP+PAC (ours)	MLP	Pipelines	✓	PyTorch	2023	<u>6.13</u>	<u>2100</u>
Pre-training (ours)	MLP	Pipelines	✓	PyTorch	2023	**6.57**	2068

Table 1: Summary of deep-learning-based lossless data compression, where the best results are in **bold**, the second best results are <u>underlined</u>. The 'GPU Opt.' refers to GPU optimization technique (All results are conducted under the same NVIDIA A800 conditions).

rate estimation and also higher compression speed compared to other deep-learning-based lossless compressors [Mao *et al.*, 2022b]. NNCP [Bellard, 2019], TRACE [Mao *et al.*, 2022b], OREO [Mao *et al.*, 2022a] and PAC [Mao *et al.*, 2023] are the typical compressors that use the transformer or MLP. Most deep-learning-based lossless compressors train their networks with a GPU to speed up the compression process [Kang *et al.*, 2022; Ko *et al.*, 2021]. PAC reduces the traffic of CPU-GPU data transfer using software cache in GPU memory [Mao *et al.*, 2023], which helps PAC compress faster. To further improve both compression ratio and compression speed, the main challenges faced by current deep-learning-based lossless compressors are summarized into three aspects: 1) **Data feature extraction is difficult for byte streams.** The lossless compression requires the compressor to convert several types of input (such as image, text, and sound, etc) into byte streams [Yamagiwa, 2022]. The semantics of such byte streams are highly sparse and varied, which makes it difficult to predict and gain a high compression ratio. 2) **Disk I/O, CPU-GPU data transfer and arithmetic coding are time-consuming,** which is taken up to one third of the total [Choukse *et al.*, 2020]. Although existing methods use a cache mechanism to decrease the traffic of data transfer, these approaches only partially reduce the transfer overhead and do not fully utilize the parallel capabilities of multi-core CPUs and GPUs [Cloud *et al.*, 2011]. 3) **The compression of large-scale files results in high GPU memory footprint demand.** PyTorch's GPU memory for multiple streams is isolated and cannot be reused, resulting in a huge waste of memory.

To overcome these challenges, we propose a lossless compression framework based on Semantics Enhancement and multi-stream Pipelines (SEP). The contributions are summarized as follows:

- **Semantics enhancement module**: We propose a series of novel approaches to capture complex semantic information of adjacent byte sequences, which achieve higher compression ratios across diverse data types.
- **Multi-stream pipelines**: We propose a multi-stream pipeline mechanism for parallel compression. By hiding disk I/O and CPU-GPU data transfer, such as Host to Device (H2D) and Device to Host (D2H), our objective is to fully use the hardware for compression speed.
- **GPU Memory Optimization**: We propose a method to enable GPU memory reuse across streams by breaking PyTorch's default stream-level memory isolation.
- **State-of-the-art (SOTA) compression performance**: Our experiments demonstrate that the SEP framework enhances the backbone neural networks' compression ratio by 5.1% on average, reduces compression time by 30.0%, and cuts GPU memory usage by 36.2%.

2 Related Work

[Deep-Learning Data Compression Methods] Recently, several deep-learning-based compressors have been proposed to improve compression ratios of heterogeneous files, e.g., video, image, etc. These approaches view the compression task as a sequential modeling problem, where the historical symbols are used as input to estimate the probability of the incoming symbol. NNCP [Bellard, 2019] and TRACE [Mao *et al.*, 2022b] utilize transformers to achieve probability estimation. However, the semantic density of byte streams is very sparse, and existing models cannot extract enough information from byte streams, resulting in low compression rates. Furthermore, the compression speed of these compressors is impractical for real-world applications. For example, NNCP's dictionary size of 16,384 and its 55 MB transformer model lead to a compression speed of only 2.0 KB/s. TRACE achieves a compression speed of 15.9 KB/s, while PAC offers an even higher compression speed of 31.1 KB/s. Therefore, methods that improve both the compression ratio and speed are needed for deep-learning compressors.

[GPU Memory Optmization] The work on GPU memory optimization is mainly focused on offloading [Lin *et al.*, 2023], recomputation [Peng *et al.*, 2020] and defragmentation [Fang *et al.*, 2022; Ren *et al.*, 2021]. Offloading techniques like vDNN [Rhu *et al.*, 2016] reduce memory usage on the GPU side but increase the communication between host and device, leading to higher host memory requirements. The recomputation technique like checkpoint [Chen *et al.*, 2016] leads to irregular allocation requests, resulting in higher fragmentation. Defragmentation like Glake [Guo *et al.*, 2024] is

currently limited to only single-stream scenarios. **Our work is the first to delve into GPU memory reuse in PyTorch's multi-stream environment, offering potential benefits for models and applications built with PyTorch.**

3 Byte-Stream Data Compression

Any data can be represented as a binary data stream. In byte-stream data compression, each set of 8 bits is considered as a byte to be compressed [Mao *et al.*, 2022b]. In this paper, we divide the byte stream into many sequences with length $L \in N^+$, serving as input tokens for the model. Each sequence is denoted as $\mathbf{X} \in R^{L\times 1}$. Our objective is to forecast the probability distribution for the next byte, which contains the probabilities for 256 possible byte values. The probability distribution computation is formalized as a $\mathrm{Predictor}$ function by Equation 2.

$$\mathbf{K} = \{p_1, p_2, \ldots, p_{256}\} = \mathrm{Predictor}(\mathbf{X}) \tag{2}$$

where $\sum_{i=1}^{256} p_i = 1$ and p_i denotes the probability of the i-th possible byte value. Once the probability distribution is predicted, we compress the input sequence with an arithmetic encoder. The $\mathrm{ArithmeticEncoder}$ function can be represented by Equation 3:

$$\mathbf{X}_{compressed} = \mathrm{ArithmeticEncoder}(\mathbf{K}) \tag{3}$$

where $\mathbf{X}_{compressed}$ denotes the compressed result and is written to disk. Multiple rounds of the above process constitute a complete data compression. For decompression, due to the model parameters being fixed, executing the training process again can produce the same probability distribution. The decompression results can be obtained through the arithmetic encoder [Howard and Vitter, 1992; Howard and Vitter, 2008].

4 The SEP Framework

4.1 Overview of SEP Architecture

The SEP framework divides the compression system into multiple components. Figure 1 shows the overview architecture of the SEP framework. All types of binary files are first converted to byte streams. These streams are divided and initialized to different task queues, with a random probability distribution. Then, the Semantics Enhancement (SE) Module extracts the features of the byte stream and turns them into the fusion patches with adaptive stride. By feeding the enhanced data into compressor backbones such as TRACE, PAC and others, the probability distribution of the next byte is predicted. Finally, the byte stream with a probability distribution is compressed by the arithmetic coder. Notably, our computation and GPU optimization modules can simultaneously reduce memory consumption and expedite the entire compression process above, which includes Direct Memory Access (DMA) and I/O hiding to enhance the GPU and CPU throughput. In the following subsections, we will detail the design and techniques implemented for each key component of our SEP framework. During decompression, the same model parameters are applied to reproduce the training process, ensuring the regeneration of the consistent probability distribution. This approach allows the decompression process to accurately restore the original data, achieving lossless

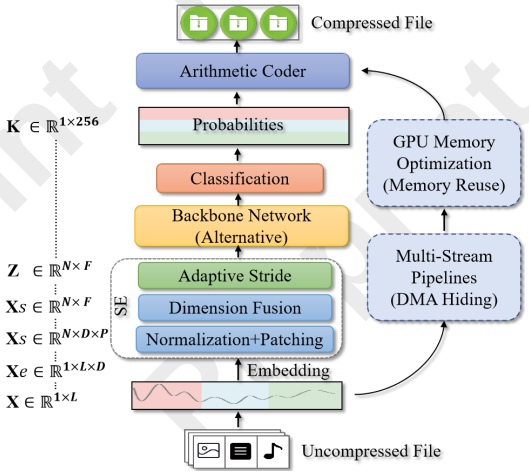

Figure 1: The overview of SEP.

compression. Most importantly, our SEP framework can also be fully used in the decompression process, leading to an improvement in decompression speed.

4.2 Semantics Enhancement Module

Semantic-Patch Expansion

Byte stream data is characterized by low semantic density and uneven distribution [Coull and Gardner, 2019]. Existing methods use single-byte symbols as tokens for attention computation [Mao *et al.*, 2023; Mao *et al.*, 2022b]. However, the limited receptive field makes it difficult to learn semantic information. In the Semantic-Patch Expansion module, we transform single-byte symbols into sub-sequence level patches as tokens. This enables the network to better understand latent cross-patch semantic correlations and to achieve a high compression speed with fewer tokens.

As shown in Figure 2 (a), a single byte matrix $\mathbf{X}$ with the length of L is embedded with a matrix $\mathbf{E} \in R^{256\times D}$, where D represents the dimension of the embedded vector. In Equation 4, $\mathbf{X_e} \in R^{L\times D}$ is the embedded form of X.

$$\mathbf{X_e} = \mathbf{E}\,[\mathbf{X}] \tag{4}$$

Then, a patching function PATCH segments $\mathbf{X_e}$ into patches. Specifically, assuming the length of the patch is P and the length of the stride is S, each patch is tokenized stride by stride. The length of the overlap between two adjacent patches is $P - S$. Therefore, N patches can be segmented from $\mathbf{X_e}$, where $N = \frac{L-P}{S} + 1$. The process is depicted by Equation 5.

$$\mathbf{X_s} = \mathrm{PATCH}(\mathbf{X_e}, S) \tag{5}$$

where $\mathbf{X_s} \in R^{N\times D\times P}$ is the reshaped form of N patches, and each patch is a $D \times P$ matrix.

Dimension Fusion

Once the data has been divided into patches, the information of multiple bytes in a patch is critical for attention module. We design a high-level Dimension Fusion (DF) module, which captures complex intra-patch relationships.

As shown in Figure 2 (b), we do a permutation operation to reorganize the patches in $\mathbf{X_s}$ and make a position alignment

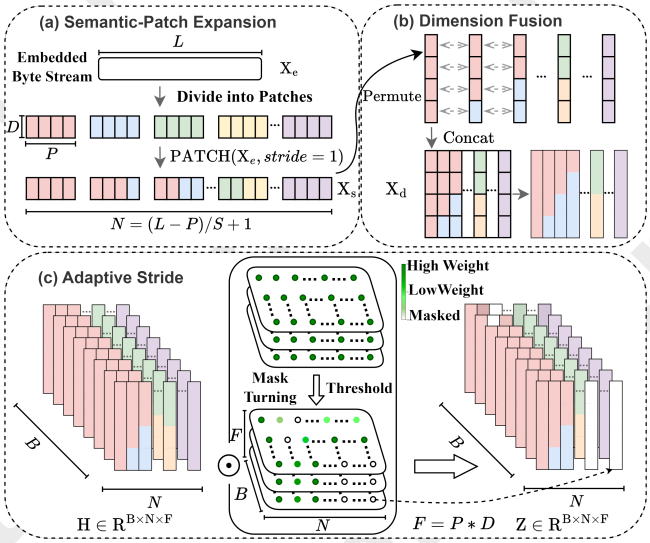

Figure 2: The detailed workflow of the Semantics Enhancement Module. (a) We set the length of patch P to 4, stride S to 1 and $\mathbf{X_e} \in R^{1\times L\times D}$. (b) Feeding $\mathbf{X_s}$ into Dimension Fusion to obtain $\mathbf{X_d}$, with overlap of dimensions. (c) The input for the adaptive stride is defined as $\mathbf{H} \in R^{B\times N\times F}$, which comprises B instances of $\mathbf{X_d}$, where B is the batch size. Then the mask matrix $\mathbf{M}$ learns from the network and masks out patches with low semantic density in $\mathbf{H}$ to achieve dynamic stride changes.

which can maintain the consistency of the channel space. The DF can be modeled as:

$$\mathbf{X_d} = \mathrm{DF}(\mathbf{X_s}) \tag{6}$$

where $\mathbf{X_s}$ is the output of Semantic-Patch Expansion, $\mathbf{X_d} \in R^{N\times F}$, and F is the fused as $D \times P$.

Adaptive Stride

For convenience, it is a common practice to partition data into patches using a fixed stride (e.g., 4 or 8) [Zhang and Yan, 2022; Nie *et al.*, 2022]. However, the manual setting of hyper-parameters requires careful fine-tuning. Given that the semantic density of byte stream varies, we propose an adaptive stride method. Specifically, we employ a mask matrix to "simulate" changes in stride. If one patch is masked (value being 0), it indicates low semantic density, and the sliding window takes an extra step forward. The design has two advantages: 1) Semantic density distribution is adjusted to be more uniform. 2) Sparse matrix multiplication speeds up the compression process.

Figure 2 (c) shows the operation in detail. The input for the adaptive stride module is defined as $\mathbf{H} \in R^{B\times N\times F}$, which comprises B instances of $\mathbf{X_d}$, where B is the batch size. Then, we design a mask tensor $\mathbf{M} \in R^{B\times N\times F}$. The $\mathbf{M}$'s dimensions correspond to the $\mathbf{H}$, since we need to use $\mathbf{M}$ to selectively mask patch in different batches. In the initial stage, all elements in $\mathbf{M}$ are set to 1, which means that every patch is retained. The initialization process can be expressed as Equation 7:

$$\mathbf{M} \in R^{B\times N\times F}, \forall i,j : \mathbf{M}_{i,j} = 1 \tag{7}$$

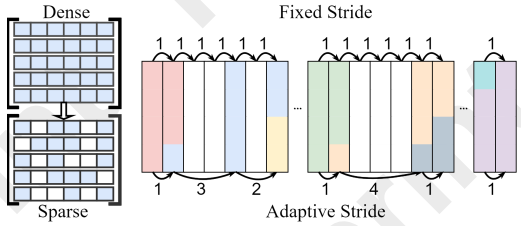

Figure 3: The comparison of adaptive stride and fixed stride.

Through the learning, the elements in $\mathbf{M}$ can be continuously adjusted. Once the element is lower than a threshold, the element will be set to 0. The process is shown in Equation 8:

$$\mathbf{Z} = \mathbf{H} \odot \mathbf{M} \tag{8}$$

where $\mathbf{Z} \in R^{B\times N\times F}$ is the masked result. Figure 3 shows an example of masked results, the white block represents the masked patch. Compared with the current method where the stride is fixed at 1, our adaptive stride produces a sparser matrix. Furthermore, according to the semantic density distribution of the byte stream, the adaptive stride can guide the backbone network to swiftly focus on important information.

4.3 Multi-Stream Pipelines

The Multi-Stream Pipelines method is designed to hide CPU-GPU data transfer, arithmetic encoder and disk I/O. A comparison between the PAC pipeline [Mao *et al.*, 2023] and our pipelines is given in Figure 4.

[PAC Pipeline] In the top part of Figure 4, the compression process is shown to be sequential, with inefficient use of GPU and CPU cores. The CPU dispatches tasks to the GPU, but remains blocked waiting for GPU operations to finish. Once the GPU completes tasks like backpropagation, the CPU resumes its work, including arithmetic encoding (AC) and I/O operations, while the GPU waits for the next CPU command.

[SEP-based Multi-stream Pipelines] Evident parallelism and time savings are shown in the bottom part of Figure 4. Nowadays, most GPUs are equipped with dual copy engines and a kernel computation engine, such as NVIDIA RTX 4090 or AMD MI250. We design the multi-stream pipelines to

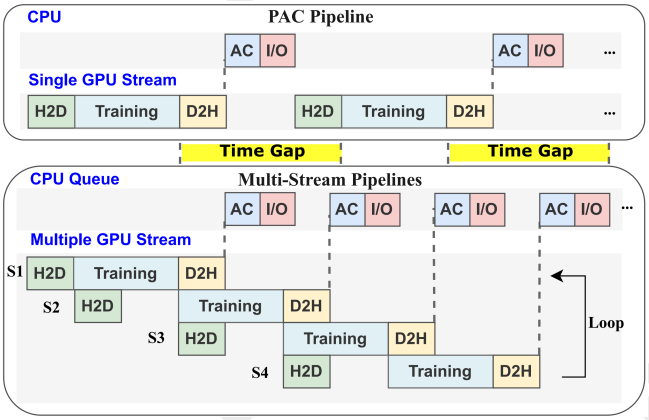

Figure 4: The time gap between PAC pipeline and SEP pipeline.

Algorithm 1 Multi-Stream Pipelines

```
1:  // Producer
2:  for task in the T do
3:      for i = 0 to pool.size do
4:          stream = pool[i];
5:          if torch.cuda.stream(stream) == TRUE then
6:              task_gpu=task.to('gpu', non_blocking=True);
7:              prob = backbone(train_gpu);
8:              probability=prob.to('cpu', non_blocking=True);
9:              queue.put(probability);
10:         end if
11:     end for
12: end for
13: // Consumer
14: while queue is not empty do
15:     probability = queue.pop();
16:     compressed = Arithmetic_encoder(probability);
17:     compressed.write();
18: end while
```

hide H2D and D2H data transfers by a non-blocking strategy. For tasks such as arithmetic encoding and I/O, we build task queues with a producer-consumer model. One CPU core focuses solely on scheduling, the other cores handle computations, which eliminates the idle time for the GPU and CPU. The pseudocode is shown in Algorithm 1. We initialize the stream set as **pool**, the task set as **T**, and the producer-consumer queue as **queue**. The algorithm divides the execution process into producer (Line 2-12) and consumer (Line 14-18). During producer process, the training task is across the different streams in **pool** (Line 3). DMA operations such as non-blocking H2D and D2H are set for overlapping (Lines 6 and 8). When probability distribution computations are finished, the Arithmetic Encoder and I/O tasks are put in the **queue** (Line 9). At the same time, the consumer loops to find the tasks in the **queue** (Line 14). Then the consumer sequentially retrieves tasks from the **queue** (pop) and performs calculations (Line 15-17).

4.4 GPU Memory Sharing Strategy

To increase GPU memory utilization during multi-stream training, enabling support for larger models and parameters, we design an inter-stream memory sharing strategy on PyTorch 2.0's memory allocator.

As shown in Figure 5, we first use a profiler to analyze the sequence of memory allocation and release to identify memory blocks that can be shared. We focus on memory blocks that are few but occupy a large portion of memory for maximizing memory reuse. Notably, our method preserves the original timing of allocation and deallocation. Next, we set up a shared pool to manage the shared memory blocks between streams. Our method does not alter the allocation and deallocation timing of memory blocks during the training process. For example, when Stream 1 (S1) completes its training, the large continuous memory blocks it used are released back into the shared pool and labeled as reserved blocks. The next blocking Stream 2 (S2) can access these reserved blocks by adjusting its pointer to the address of these blocks. When

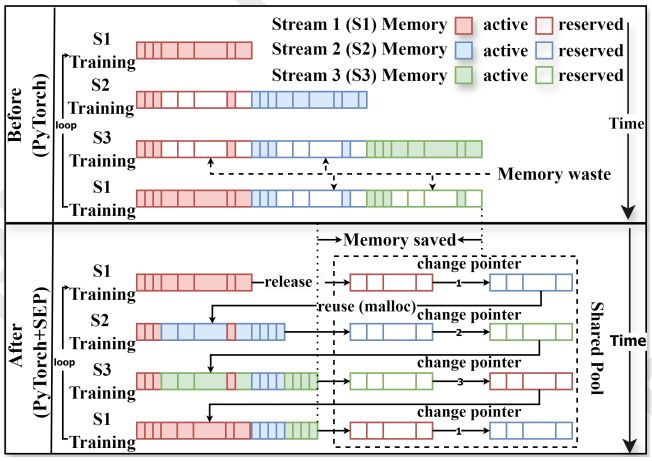

Figure 5: Multi-stream GPU memory optimization.

S2 requests new memory space, it searches the shared pool for available reserved blocks. If suitable blocks are found, malloc reuses them for the next stream, significantly saving memory. Furthermore, the entire training process involves a loop-intensive data flow, where the size of the memory blocks involved in computations is the same. This allows for a match with the memory blocks shared from previous training processes, further reducing memory fragmentation.

5 Experiments

All experiments are performed on a server powered by Intel 5218 CPUs (16 cores) and eight NVIDIA A800 GPUs (80GB memory), using a PCIe Gen 4.0 interface. The storage includes a SAMSUNG SSD PM893 with 1.92 TB capacity via SATA connection. The operation system is Ubuntu 20.04. The software environment is based on PyTorch version 2.0.1 and NVIDIA's CUDA version 12.1 for GPU acceleration.

5.1 Experimental Settings

Datasets. We evaluate the SEP framework on seven real-world datasets, including: Book [Zhu *et al.*, 2015], Enwik9 [Mahoney, 2011], Float [Burtscher and Ratanaworabhan, 2008], Sound, Image [Deng *et al.*, 2009], Backup(heterogeneous) [Mao *et al.*, 2023] and Silesia [Deorowicz, 2007; Piczak, 2015].

Baselines. We select two SOTA compressors for comparison, including TRACE [Mao *et al.*, 2022b], PAC [Mao *et al.*, 2023]. To better illustrate the generalization of SEP, we select time-series forecasting model PatchTST [Nie *et al.*, 2022] as one of the backbone and directly integrate it into SEP.

Parameter Details. To ensure the fairness of the experiment, all model parameters are set to the same. We set history sequence length to 64, the patch length to 4, and the corresponding feature dimension to 64. In all backbones, Adam is applied with a learning rate of 0.001, the threshold is set to 0.5, and the stride is set to 1.

5.2 Main Experiments

Table 2 presents the compression ratios and speeds of SOTA compressors, as well as those of the compressors when in-

Methods	Pre-training		SEP+PAC		PAC		SEP+TRACE		TRACE		SEP+PatchTST		PatchTST	
Target	Ratio	Speed	Ratio	Speed	Ratio	Speed	Ratio	Speed	Ratio	Speed	Ratio	Speed	Ratio	Speed
Enwik9 | 1000MB	**6.57**	**32.3**	6.13	32.4	5.82	26.1	**5.52**	**34.1**	5.20	26.1	**5.34**	**34.6**	5.03	26.1
Improvement	Ratio:12.8%		Ratio:5.3%,Speed:24.1%				Ratio:6.2%,Speed:30.7%				Ratio:6.1%,Speed:32.5%			
Book | 1000MB	**5.56**	**32.8**	5.16	32.8	5.05	25.5	**4.75**	**34.0**	4.51	26.2	**4.55**	**33.5**	4.36	25.9
Improvement	Ratio:10.0%		Ratio:2.2%,Speed:28.6%				Ratio:5.3%,Speed:29.8%				Ratio:4.3%,Speed:29.3%			
Sound | 842MB	**2.36**	**33.3**	2.26	33.4	2.26	25.7	**2.23**	**33.9**	2.13	27.3	**2.20**	**32.8**	2.09	27.1
Improvement	Ratio:4.4%		Ratio:0%,Speed:30.0%				Ratio:4.7%,Speed:24.2%				Ratio:5.6%,Speed:21.1%			
Image | 1200MB	**2.14**	**34.7**	2.03	34.8	2.01	26.0	**1.95**	**35.5**	1.87	27.6	**1.92**	**34.1**	1.84	26.4
Improvement	Ratio:6.4%		Ratio:1%,Speed:33.9%				Ratio:4.3%,Speed:28.6%				Ratio:4.4%,Speed:29.2%			
Float | 189MB	**3.87**	**31.5**	3.68	31.7	3,53	25.2	**3.49**	**33.3**	3.21	25.6	**3.43**	**32.0**	3.23	25.4
Improvement	Ratio:9.6%		Ratio:4.3%,Speed:25.8%				Ratio:8.8%,Speed:30.1%				Ratio:6.1%,Speed:26.0%			
Silesia | 206MB	**5.25**	**33.6**	5.16	33.8	5.0	26.0	**4.78**	**34.1**	4.61	26.0	**4.81**	**33.5**	4.57	26.3
Improvement	Ratio:5.0%		Ratio:3.2%,Speed:30.0%				Ratio:3.6%,Speed:31.2%				Ratio:5.3%,Speed:27.4%			
Backup | 1000MB	**2.00**	**32.0**	1.95	32.0	1.91	24.3	**1.84**	**33.2**	1.76	26.5	**1.81**	**32.6**	1.72	25.4
Improvement	Ratio:4.7%		Ratio:2.6%,Speed:31.7%				Ratio:4.5%,Speed:25.3%				Ratio:5.1%,Speed:28.3%			
Count	14		0		0		14		0		14		0	

Table 2: Comparison of SEP combined with three existing models against their original models, where the best results are in **bold**, the second best results are underlined. In the 'Improvement' row, 'Ratio' denotes the improvement of the compression ratios, and 'Speed' indicates the improvement of the compression speeds (KB/s), as compared to non-SEP models, respectively. The 'Count' indicates the best counter between origin model and SEP-based model. Pre-training refers to models trained on SEP+PAC, as compared to PAC.

tegrated with the SEP framework, across various datasets. Overall, our solution achieves an average improvement of 5.1% in compression ratios and an average improvement of 30.2% in compression speed.

[Compression Ratio Comparison] In Table 2, PatchTST naturally has lower compression than TRACE since it is not created to focus on the compression task. However, after combining it with our framework, the compression ratio easily surpasses TRACE. For the SEP+PAC, we observe an improvement in compression ratio, although it is less significant compared to SEP+TRACE and SEP+PatchTST. This difference is largely attributed to the MLP model design of PAC. The results of combining SEP with other backbones prove that the SEP framework can be flexibly integrated with other backbones and exhibit excellent compression performance. We find that models without pre-training show poor performance in the early compression stages (20,000 rounds). By developing specialized pre-training models for each data type, we further improve compression performance significantly, achieving a 12.8% improvement on Enwik9.

[Compression Speed Comparison] In Table 2, once a backbone model is combined with the SEP framework, the compression speed can directly exceed the original model by a wide margin. The utilization of both CPU and GPU to construct pipelines makes the process universally applicable to nearly all deep-learning models. Consequently, SEP consistently performs across various models and datasets, achieving an average speed improvement of 30.0%. This enhancement is vital for compression tasks.

[GPU Memory Optimization] We conduct a comparative analysis of GPU memory footprint between the SEP and the native PyTorch on different batch sizes. Figure 6 illustrates that our memory optimization strategy achieves an average reduction of 21.7% on SEP+PAC, 61.7% on SEP+PatchTST, and 26.3% on SEP+TRACE. Overall, our SEP reduces memory usage by approximately 36.2% compared to the PyTorch framework. Due to the fact that PAC consists of multiple linear layers with small matrices, and PatchTST comprises large patch blocks with large matrices, the optimization of PatchTST shows better results with higher intermediate variables. These results show SEP as an effective solution for GPU memory constraints, with potential for future multi-stream deep learning applications.

[Adaptive Stride] We use the SEP+TRACE as the baseline and all parameters are consistent with the main experiment. We select the fixed stride value from [1, 2, 3, 4] and compare it with our adaptive stride method. Figure 7 presents the comparison between the adaptive stride and other fixed strides in compression ratio. This result proves that the design of adaptive stride improves the compression ratio due to the dynamic adjustment ability of semantic density. Notably, there is a 4.1% improvement in compression speed due to the sparse matrix. Adaptive stride addresses the issue of excessive redundant information following patching in existing models and offers a novel approach for time series research.

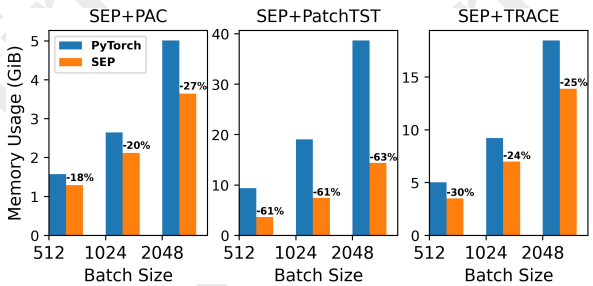

Figure 6: GPU memory footprint between SEP and PyTorch.

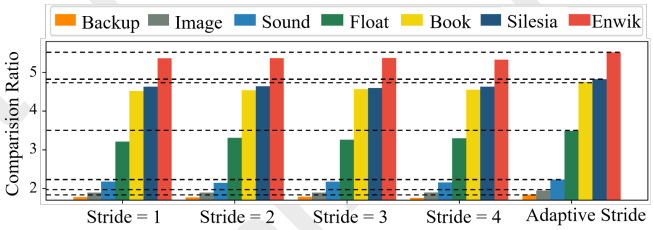

Figure 7: Comparison of fixed stride and adaptive stride.

5.3 Ablation Study

[Module Ablation on SEP] We evaluate the impact of different SEP modules on compression ratio and speed (KB/s). The multi-stream pipeline (PIPE) module achieves the highest acceleration in compression speed, while the SE module slightly improves both the compression ratio and speed. The memory sharing (MS) module does not directly accelerate the process but allows for handling larger datasets, enabling SEP to compress more data without exceeding memory limits.

	PAC	PAC+SEP	W/O-SE	W/O-PIPE	W/O-MS
Ratio	5.8	6.1	5.8	6.1	6.1
Speed	26.1	32.4	31.4	27.1	32.6
	TRACE	**TRACE+SEP**	**W/O-SE**	**W/O-PIPE**	**W/O-MS**
Ratio	5.2	5.5	5.2	5.5	5.5
Speed	26.1	34.1	32.9	27.3	34.2

Table 3: Module Ablation Results on Enwik9.

[Ablation on Sparse Matrix and Size Reduction (SE)] In SE module, the speedup is primarily attributed to two factors: matrix size reduction and sparse matrix computation. To measure the impact of each part, we conduct aan blation study in Table 4 on PAC+SEP. The results indicate that the main speedup comes from matrix size reduction when sparsity is low (30-50%). However, the Image dataset sparsity can reach 78% in certain iterations (not every iteration). When sparsity exceeds 70%, the use of sparse matrix operations (‘sparse.mm‘) further accelerates computation.

Dataset	Sparsity Range	Speedup By Matrix Size	Speedup By Sparse Matrix	Speedup By SE (Total)
Enwik9	30-42%	3.9%	0%	3.9%
Book	30-45%	2.9%	0%	2.9%
Image	30-80%	2.5%	4.1%	6.6%

Table 4: Effect of Sparse Matrix and Matrix Size Reduction on SE.

[Evaluation of SEP Multi-Stream Pipeline] To evaluate the performance of SEP’s multi-stream pipeline over CUDA stream pipeline, we conduct an ablation study that excludes the SE module. The results in Table 5 show that SEP outperforms CUDA in both image and text datasets, demonstrating the efficiency of SEP pipeline in multi-GPU settings.

Dataset	Model	CUDA Pipeline Speedup	SEP Pipeline Speedup	SEP Speedup
Enwik9	PAC	3.3 %	20.1 %	24.1 %
	TRACE	4.3 %	26.2 %	30.7 %
Image	PAC	4.2 %	27.2 %	33.9 %
	TRACE	3.6 %	24.7 %	28.6 %

Table 5: Comparison of CUDA and SEP Multi-Stream Speedups.

5.4 Scalability and Hardware Compatibility

[Scalability] We adopt multiprocessing and data parallelism to minimize communication overhead as much as possible. To evaluate the scalability of the SEP-based model in a multi-GPU environment, we conduct experiments under two, four and eight NVIDIA Tesla A800 GPUs, which demonstrate the SEP’s scalability and efficiency for data compression. As shown in Figure 8, under data such as Image and Enwik9, a near-linear acceleration ratio can be achieved, and still exhibits good scalability.

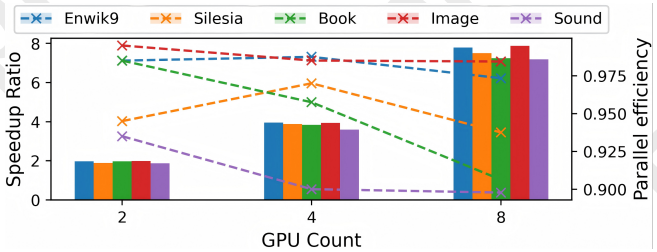

Figure 8: Scalability of SEP on multiple GPUs.

[Hardware Compatibility] To demonstrate the compatibility of SEP, we conduct experiments on low-end devices, including NVIDIA GTX 1060 (6GB), RTX 2080 (8GB) and RTX 3080 (10GB). As shown in Table 6, SEP consistently outperforms baseline models (PAC, TRACE, PatchTST) across all hardware configurations, still achieving up to 30% speedup on low-end GPUs. Notably, the GTX 1060 achieves an 82% speedup on PatchTST, likely due to SEP’s memory optimization reducing frequent memory paging caused by PatchTST’s high memory usage on devices with limited memory.

GPU	PAC	PAC (+SEP)	TRACE	TRACE (+SEP)	PatTST	PatTST (+SEP)
GTX1060	315	393(25%↑)	309	389(26%↑)	204	372(82%↑)
RTX2080	866	1082 (25%↑)	858	1055 (23%↑)	810	1004 (24%↑)
RTX3080	1376	1816 (32%↑)	1384	1813 (31%↑)	1310	1689 (29%↑)

Table 6: Compression Speed (KB/min) on Various GPU Devices.

6 Conclusion

The SEP is a plug-in framework that easily integrates with various deep learning models and compression algorithms. Meanwhile, SEP supports flexible hardware configurations and offers out-of-the-box functionality for compressing and decompressing data.

Acknowledgements

This work is supported by the National Key Research and Development Program of China (2022ZD0115305). We thank VenusAI Platform (http://data.aicnic.cn) for kindly providing the GPU clusters for training the models.

Contribution Statement

† These authors have equal contribution.

References

[Affeldt *et al.*, 2014] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. Formalization of shannon's theorems. *Journal of Automated Reasoning*, 53:63–103, 2014.

[Bellard, 2019] Fabrice Bellard. Lossless data compression with neural networks. 2019.

[Burtscher and Ratanaworabhan, 2008] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE transactions on computers*, 58(1):18–31, 2008.

[Chandra and Hsu, 2014] Surendar Chandra and Windsor W. Hsu. Lossless medical image compression in a block-based storage system. In *2014 Data Compression Conference*, pages 400–400, 2014.

[Chen *et al.*, 2016] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. (arXiv:1604.06174), April 2016. arXiv:1604.06174 [cs].

[Choukse *et al.*, 2020] Esha Choukse, Michael B Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 926–939. IEEE, 2020.

[Cloud *et al.*, 2011] Robert Louis Cloud, Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Purushotham V. Bangalore. Accelerating lossless data compression with gpus. *CoRR*, abs/1107.1525, 2011.

[Contoli and Lattanzi, 2023] Chiara Contoli and Emanuele Lattanzi. A study on the application of tensorflow compression techniques to human activity recognition. *IEEE Access*, 11:48046–48058, 2023.

[Coull and Gardner, 2019] Scott E Coull and Christopher Gardner. Activation analysis of a byte-based deep neural network for malware classification. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 21–27. IEEE, 2019.

[Deng *et al.*, 2009] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[Deorowicz, 2007] Sebastian Deorowicz. Silesia compression corpus, 2007.

[Fang *et al.*, 2022] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):304–315, 2022.

[Goyal *et al.*, 2021] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. Dzip: Improved general-purpose loss less compression based on novel neural network modeling. In *2021 Data Compression Conference (DCC)*, pages 153–162. IEEE, 2021.

[Guo *et al.*, 2024] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. Gmlake: Efficient and transparent gpu memory defragmentation for large-scale dnn training with virtual memory stitching. In *ASPLOS*. arXiv, January 2024. arXiv:2401.08156 null.

[Howard and Vitter, 1992] Paul G Howard and Jeffrey Scott Vitter. Analysis of arithmetic coding for data compression. *Information processing & management*, 28(6):749–763, 1992.

[Howard and Vitter, 2008] Paul G. Howard and Jeffrey Scott Vitter. *Arithmetic Coding for Data Compression*, pages 65–68. Springer US, Boston, MA, 2008.

[Kang *et al.*, 2022] Ning Kang, Shanzhao Qiu, Shifeng Zhang, Zhenguo Li, and Shutao Xia. Pilc: Practical image lossless compression with an end-to-end gpu oriented neural framework, 2022.

[Knoll, 2016] B Knoll. Tensorflow-compress, 2016.

[Knuth, 1985] Donald E Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[Ko *et al.*, 2021] Y Ko, A Chadwick, D Bates, and R Mullins. Lane compression: A lightweight lossless compression method for machine learning on embedded systems. 2021.

[Lagar-Cavilla *et al.*, 2019] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330, 2019.

[Lin *et al.*, 2023] Shao-Fu Lin, Yi-Jung Chen, Hsiang-Yun Cheng, and Chia-Lin Yang. Tensor movement orchestration in multi-gpu training systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1140–1152. IEEE, 2023.

[Liu *et al.*, 2019] Qian Liu, Yiling Xu, and Zhu Li. Decmac: A deep context model for high efficiency arithmetic coding. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 438–443. IEEE, 2019.

[Mahoney, 2011] Matt Mahoney. Large text compression benchmark, 2011.

[Mao *et al.*, 2022a] Yu Mao, Yufei Cui, Tei-Wei Kuo, and Chun Jason Xue. Accelerating general-purpose lossless compression via simple and scalable parameterization. In *Proceedings of the 30th ACM International Conference on Multimedia*, pages 3205–3213, 2022.

[Mao *et al.*, 2022b] Yu Mao, Yufei Cui, Tei-Wei Kuo, and Chun Jason Xue. Trace: A fast transformer-based general-purpose lossless compressor. In *Proceedings of the ACM Web Conference 2022*, pages 1829–1838, 2022.

[Mao *et al.*, 2023] Yu Mao, Jingzong Li, Yufei Cui, and Chun Xue. Faster and stronger lossless compression with optimized autoregressive framework. In *60th Design Automation Conference (DAC 2023): From Chips to Systems-Learn Today, Create Tomorrow*, 2023.

[Mentzer *et al.*, 2019] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10621–10630, 2019.

[Moffat, 2019] Alistair Moffat. Huffman coding. *ACM Computing Surveys (CSUR)*, 52(4):1–35, 2019.

[Nie *et al.*, 2022] Yuqi Nie, Nam H Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. *arXiv preprint arXiv:2211.14730*, 2022.

[Peng *et al.*, 2020] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[Piczak, 2015] Karol J Piczak. Esc: Dataset for environmental sound classification. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 1015–1018, 2015.

[Ren *et al.*, 2021] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[Rhu *et al.*, 2016] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 1–13, October 2016.

[Rissanen and Langdon, 1979] Jorma Rissanen and Glen G Langdon. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[Schiopu and Munteanu, 2020] Ionut Schiopu and Adrian Munteanu. Deep-learning-based lossless image coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(7):1829–1842, 2020.

[Sullivan *et al.*, 2012] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

[Wu *et al.*, 2016] Hao Wu, Xiaoyan Sun, Jingyu Yang, Wenjun Zeng, and Feng Wu. Lossless compression of jpeg coded photo collections. *IEEE Transactions on Image Processing*, 25(6):2684–2696, 2016.

[Yamagiwa, 2022] Shinichi Yamagiwa. *Stream-Based Lossless Data Compression*, pages 391–410. Springer Singapore, Singapore, 2022.

[Yang *et al.*, 2023] Runzhao Yang, Tingxiong Xiao, Yuxiao Cheng, Qianni Cao, Jinyuan Qu, Jinli Suo, and Qionghai Dai. Sci: A spectrum concentrated implicit neural compression for biomedical data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4774–4782, 2023.

[Zhang and Yan, 2022] Yunhao Zhang and Junchi Yan. Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting. In *The Eleventh International Conference on Learning Representations*, 2022.

[Zhang *et al.*, 2021] Honglei Zhang, Francesco Cricri, Hamed R. Tavakoli, Nannan Zou, Emre Aksu, and Miska M. Hannuksela. Lossless image compression using a multi-scale progressive statistical model, 2021.

[Zhu *et al.*, 2015] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.