Fastensor: Optimise the Tensor I/O Path from SSD to GPU for Deep Learning Training

JIA WEI, XINGJUN ZHANG, LONGXIANG WANG, and ZHENG WEI, Xi'an Jiaotong University, China

In recent years, benefiting from the increase in model size and complexity, deep learning has achieved tremendous success in computer vision (CV) and (NLP). Training deep learning models using accelerators such as GPUs often requires much iterative data to be transferred from NVMe SSD to GPU memory. Much recent work has focused on data transfer during the pre-processing phase and has introduced techniques such as multiprocessing and GPU Direct Storage (GDS) to accelerate it. However, tensor data during training (such as Checkpoints, logs, and intermediate feature maps), which is also time-consuming, is often transferred using traditional serial, long-I/O-path transfer methods.

In this article, based on GDS technology, we built Fastensor, an efficient tool for tensor data transfer between the NVMe SSDs and GPUs. To achieve higher tensor data I/O throughput, we optimized the traditional data I/O process. We also proposed a data and runtime context-aware tensor I/O algorithm. Fastensor can select the most suitable data transfer tool for the current tensor from a candidate set of tools during model training. The optimal tool is derived from a dictionary generated by our adaptive exploration algorithm in the first few training iterations. We used Fastensor's unified interface to test the read/write bandwidth and energy consumption of different transfer tools for different sizes of tensor blocks. We found that the execution efficiency of different tensor transfer tools is related to both the tensor block size and the runtime context.

We then deployed Fastensor in the widely applicable Pytorch deep learning framework. We showed that Fastensor could perform superior in typical scenarios of model parameter saving and intermediate feature map transfer with the same hardware configuration. Fastensor achieves a 5.37x read performance improvement compared to *torch.save*() when used for model parameter saving. When used for intermediate feature map transfer, Fastensor can increase the supported training batch size by 20x, while the total read and write speed is increased by 2.96x compared to the torch I/O API.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; *Heuristic function construction*; • **Information systems** → **Storage management;**

Additional Key Words and Phrases: Deep learning, neural networks, GPU direct storage, tensor I/O

ACM Reference format:
Jia Wei, Xingjun Zhang, Longxiang Wang, and Zheng Wei. 2023. Fastensor: Optimise the Tensor I/O Path from SSD to GPU for Deep Learning Training. *ACM Trans. Arch. Code Optim.* 20, 4, Article 62 (November 2023), 25 pages.
https://doi.org/10.1145/3630108

This work was supported by the National Natural Science Foundation of China under Grant 62172327.
Authors' addresses: J. Wei, X. Zhang (Corresponding author), L. Wang, and Z. Wei, Xi'an Jiaotong University, No. 28, West Xianning Road, Xi'an, Shaanxi 710049, China; e-mails: weijia4473@stu.xjtu.edu.cn, {xjzhang, wlx419}@xjtu.edu.cn, frank.wei@stu.xjtu.edu.cn.

1544-3566/2023/11-ART62
https://doi.org/10.1145/3630108

1 INTRODUCTION

Deep learning, a typical representative of **artificial intelligence** (**AI**) technology, has made tremendous achievements in recent years. Deep learning models have achieved state-of-the-art performance in many fields, such as **computer vision** (**CV**), **natural language processing (NLP)**, and voice recognition. These remarkable performance improvements come from two main aspects. On the one hand, the width and depth of **deep neural networks (DNNs)** are increasing while the structure of the networks is becoming more complex. On the other hand, the size and complexity of the dataset used for DNNs are increasing. Many studies show that this trend will continue [30]. As a result, training a well-performing neural network requires an increasingly significant investment.

To achieve fast DNN training, most AI labs in academia and industry deploy training networks on accelerators such as GPUs, TPUs, and FPGAs. Among them, GPUs are the most widely used accelerators. During the entire DNN training task, much data is exchanged between storage devices (such as NVMe SSD) and GPU Memory. For example, during most image classification task training, images are first loaded in batches from the storage device into the DRAM on the CPU side. The CPU pre-processes the images and passes them on to the GPU as tensors. Then, due to the high cost of DNN training, in order to reduce the overhead caused by unexpected failures (e.g., power outages and system interruptions) that lead to task failures and thus the need to recompute tasks, checkpointing techniques are often used to periodically record the training status (such as the number of epochs trained, model parameters under the corresponding epoch) and write the status information from the GPU Memory to the storage devices. At the same time, the most significant training-intensive models for deep learning have grown by more than 10 to 109x in the last three years, from a few million parameters (ELMo [28]) to trillions of parameters (Persia [21]). The problem of GPU Memory capacity walls [2] is becoming increasingly prominent. Earlier studies [5, 16, 31] attempted to alleviate the GPU Memory capacity walls by transferring intermediate feature maps to CPU Memory. However, CPU and DRAM resources are prohibitive and scarce, and shifting feature maps to CPU Memory can severely consume CPU and DRAM resources leading to severe disruption of CPU-side tasks while failing to provide robust performance for training tasks [2].On the other hand, as the model grows, the CPU Memory capacity cannot meet the offloading demand of intermediate feature maps. Therefore, in order to decrease the impact on CPU and DRAM and achieve availability for larger-scale models, some recent studies have swapped tensor data such as model parameters [30] or intermediate feature maps [2, 36] between NVMe SSD and GPUs continuously during training to enable training of large models with smaller GPU Memory capacity. When we use NVMe SSDs to transfer intermediate feature maps to enable trading time for space (larger trainable model sizes or batch sizes), the intermediate feature map transfer time is much larger than the data read and computation time due to the time-consuming traditional data transfer path. For example, as shown in Table 1, the torch native tool to train ViT models in large batches, the intermediate feature map transfer time may exceed 40 times the combined data read and computation time.

Finally, at the end of the training, the trained model and other related files, such as logs generated during the process, need to be saved in the tensor form to the NVMe SSD for subsequent inference tasks.

Although some researchers [11, 12, 20] have dedicated themselves to solving the I/O bottleneck in the data pre-processing phase (before training starts), there are still significant potential enhancements to tensor data transfer between GPU Memory to NVMe devices during and after training. Intermediate feature map transfer during training may affect the training time from several factors (e.g., bandwidth utilization, synchronization overhead, Memory limitations, serialization and transformation overhead, impact on cache, and software overhead). Reducing the transfer

Table 1. ViT_H_14 (Per Epoch) Training Time (s) for Different Layers and Batchsizes with Intermediate Feature Map Transfer by Native Torch Tools

Batchsize	layers	Transfer Time	Data Read and Compute Time
64	32	20,421.7	511.292
64	1,000	462,886	11,959.3
960	32	21,835.1	423.725

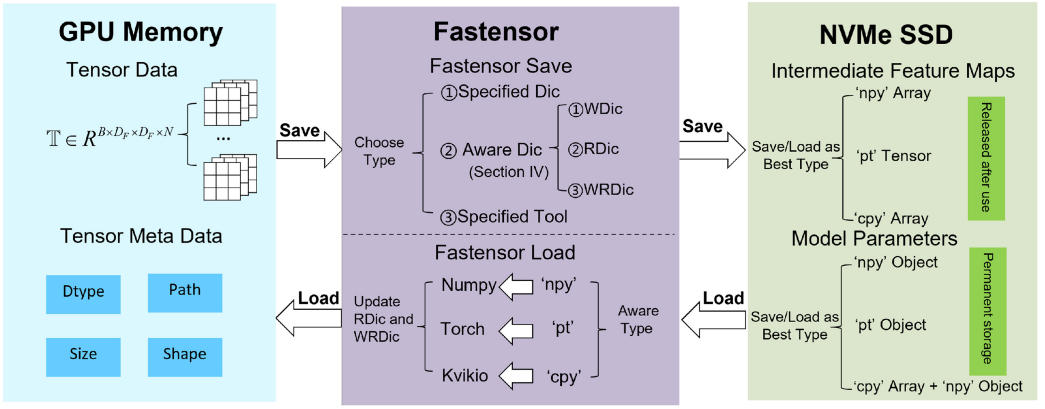

Fig. 1. Fastensor scheme. From left to right represents the tensor and the corresponding metadata saved from GPU Memory to NVMe SSD in a given data format by the most suitable transfer tool selected by the Fastensor. From right to left represents data saved in a specific format on an NVMe SSD loaded into GPU Memory and converted to tensor data via Fastensor.

overhead can greatly improve the end-to-end training speed, especially when intermediate feature maps need to be transferred between GPUs and NVMe SSDs. Taking the most widely used Pytorch deep learning platform as an example, the vast majority of researchers and engineers are using the *torch.save*() and *torch.load*() APIs to implement data transfer of tensor data between GPUs and NVMe devices. For example, using *torch.load*() to read tensor data from an NVMe device to GPU Memory, the tensor data first needs to be read from the NVMe device to DRAM on the CPU side and then transferred from DRAM to GPU Memory. This method requires a long I/O path and consumes a lot of expensive CPU resources and DRAM capacity, and bandwidth.

Therefore, in order to solve the problem of long tensor I/O paths and high resource consumption, we built *Fastensor* (as shown in Figure 1), an efficient tool for tensor data transfer between NVMe SSD and GPU Memory, which enables direct data transfer between NVMe devices and GPU Memory using GDS technology to bypass CPU and DRAM. We have also integrated a data and runtime context-aware tensor I/O algorithm into *Fastensor*. It can avoid the relatively poor performance of GDS when the data blocks are small by incorporating other data transfer APIs. Finally, we have wrapped *Fastensor* as a plug-and-play Python API that can easily replace *torch.save*() and *torch.load*() for faster data transfer without the need to understand the underlying CUDA and C++ implementation or recompile the Pytorch framework, as in studies such as [2].

Figure 1 explains how *Fastensor* is used for intermediate feature maps and model parameter transfer. Users use *Fastensor* to transfer the intermediate feature maps from the GPU to the NVMe SSD for **forward propagation** (**FP**). *Fastensor* provides three modes for users to choose from. *SpecificDic* Mode: Users use a customized transfer dictionary (with tensor block size as key and transfer method as value) to set transfer methods for tensor blocks of different sizes. *AwareDic*: We

provide three separate dictionaries, *WDic*, *RDic*, and *WRDic*, to build adaptive transfer dictionaries by sensing write performance, read performance, and read/write performance, respectively. Users can specify one of them as the transfer dictionary for this training. *SpecificTool*: The user selects a tool to realize the tensor transfer for all sizes. *Fastensor* converts pytorch tensor and metadata into *npy*, *pt*, or *cpy* format to be saved in NVMe SSD, respectively, based on the query results of the transfer dictionary. During backpropagation, *Fastensor* automatically becomes aware of the form of the data stored on the NVMe SSD and loads them into GPU Memory, then converts them into a pytorch tensor. If the user selects the *AwareDic* mode, *Fastensor* will update *RDic* and *WRDic* automatically during the process, depending on the situation. At the end of an epoch or when a user-given checkpoint is reached, the model parameters are transferred from GPU Memory to NVMe SSD via *Fastensor*. The optional transfer modes for the model parameters are the same as for the intermediate feature maps. Since kvikio does not support direct conversion of model parameters to "cpy" format, *Fastensor* constructs a built-in converter to convert model parameters to "cpy" tensor and "npy" metadata objects. When the user needs to use the saved model parameters, *Fastensor* automatically senses the model parameters saved in the NVMe SSD in a specific format, loads them to GPU Memory, and converts them to pytorch model parameter format. Our work introduces a unique adaptive mechanism that dynamically selects the optimal data transfer technique tailored to specific contexts. Furthermore, we have enhanced existing transfer methods through improved data format conversion and applying meticulous control over data order, pushing the boundaries of traditional data transfer approaches.

The main contributions of this article include the following:

— Compare and analyze the bandwidth and energy consumption of various tensor data transfer tools for reading and writing tensor data blocks of different granularity.
— Propose a data-aware adaptive tensor transfer strategy that can automatically select the optimal transfer scheme based on the data block size and significantly improve I/O throughput. We experimentally verify that our strategy can achieve 88.52% and 89.22% of the theoretical optimum on the latest NLP model Bert and CV model ViT.
— Implement a unified interface to a variety of common data transfer APIs for checkpoint saving during training, offloading, and prefetching of intermediate feature maps.
— Built *Fastensor*, an efficient and open source tensor data reading and writing tool, achieves improved read performance compared to *torch.save*() 5.37x when used for model parameter saving. When used to read and write intermediate feature maps, the batch size supported for training is increased by 20x, while the total read and write speed is increased by 2.52x compared to the torch I/O API.

2 BACKGROUND

2.1 Deep Neural Network Training

Deep learning performance in various tasks relies on the effectiveness of training DNNs. DNNs typically consist of an input layer, an output layer, and a number of hidden layers. Various layers are proposed for solving different application problems. Examples include the convolutional, pooling, and residual layers, widely used in CV tasks, and the attention and transformer layers in NLP. Furthermore, there is a tendency for the proposed network layers to intermingle within different application domains. Vision-transformer, for example, adapts the well-performing transformer layer from NLP to a model for solving CV tasks. Exploring neural network model architectures like assembling "Lego blocks" on top of the various existing layers has also received wide attention. The main construction approaches include hand-designed networks [7–9, 13, 34] and automatic network construction using neural architecture search [10, 29, 40].

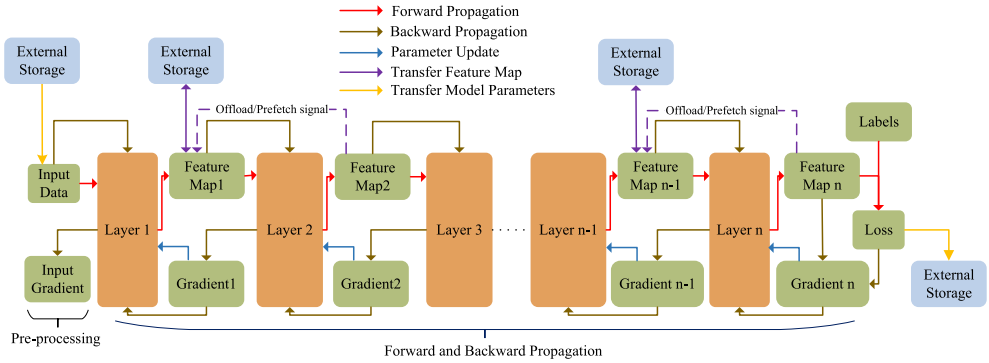

Fig. 2. DNN training.

The training process of a DNN is the finding of the model parameters that make the objective function optimal. Currently, this process is mainly achieved by using the mini-gradient descent algorithm. To use this algorithm, we first split the dataset into multiple parts, each called a mini-batch. Each iteration uses a mini-batch to train and update the model parameters. When each mini-batch has been trained once, the training of an epoch is said to have been completed. The model is iteratively trained for multiple epochs until the expectation of the objective function or a set training threshold is met.

Specifically, as shown in Figure 2, each training is divided into four phases: data reading and **pre-processing** (**PP**), FP, **backward propagation** (**BP**), and **parameter update** (**PU**). (1) In the PP phase, the data of a mini-batch is first read from external storage (such as SSD or HDD) into DRAM, or sometimes the entire dataset is read directly into DRAM to speed up the data reading process when the dataset is small, and the DRAM is large. The CPU then pre-processes this mini-batch data (such as image rotation, Gaussian blur, and random greyscale). Finally, the pre-processed data is passed into the GPU Memory as a tensor. (2) In the FP phase, starting from the first layer, the output of the current layer (intermediate feature map) is calculated using the weights of the current layer and the corresponding operations. The output of the current layer is then fed as input to the next layer and continues to be computed until the final layer, which yields the loss value of this iteration training. At this point, an FP is completed. (3) In the BP phase, the gradients to loss of the input feature maps from the current layer FP (i.e., the gradients to loss of the output feature maps from the previous layer) and the gradients to loss of the weights in the current layer are calculated iteratively using the gradients to loss of the output feature maps calculated in the current layer FP up to the first layer, respectively, starting from the last layer. (4) Subsequently, in the PU phase, all intermediate feature maps and gradients of intermediate feature maps are cleared, and all model parameters are updated using optimizers such as SGD [37], ADAM [19], and ADADELTA [39]. At this point, the training of an iteration is completed.

Training large models using large datasets often takes days or weeks. To prevent unpredictable program interruptions, it is often necessary to save model parameters and logs to a non-volatile device such as an SSD when several iterations or epochs are completed. The larger the model, the more resources this process consumes. The GPU saves the model parameters and logs to the SSD after the loss is calculated, and the SSD loads them to the GPU before starting the training (as shown in the yellow arrow part in Figure 2).

At the same time, as the model size grows, the intermediate feature maps (activations) during training gradually increase or even exceed the GPU Memory capacity, which severely limits the

availability and effectiveness of model training. In order to alleviate the above GPU capacity wall problem, many studies have proposed to save these intermediate feature maps from GPU Memory to SSD during FP and load them from SSD to GPU Memory during BP. However, this approach introduces additional data transfer overhead between GPU and SSD, and it is an issue of concern how to reduce this overhead.

2.2 Data Read and Pre-processing Phase I/O Acceleration

DNNs are currently trained by performing complex, multi-stage data pre-processing operations, including data encoding, decoding, cropping, rotation, normalization, and many other data enhancement operations. Currently performed primarily on the CPU, these operations have made data pre-processing a bottleneck in the overall model training, especially in servers with high GPU/CPU ratios, such as NVIDIA-DGX1-V, NVIDIA-DGX-2, and Amazon EC2 P3.16xlarge.

In order to alleviate data reading and pre-processing bottlenecks, many deep learning frameworks provide dedicated multi-process data reading APIs. For example, we typically use multiple CPU cores to read data for multiple mini-batches in parallel by increasing the *num_worker*() parameter in *Pytorch.dataloader*(). However, this approach is subject to the limitations of the Python **Global Interpreter Lock** (**GIL**), which complicates the design of efficient input [11] and also increases the CPU load. Furthermore, the optimal selection of *num_worker*() is intricately linked to hardware conditions such as CPU performance and I/O capabilities, as well as software factors, including model size, the complexity of the transformations (*transform*()), and the size of input data. Therefore, setting parameters such as *num_worker*() is a challenge to maximize the data reading speed and pre-processing process.

NVIDIA provides the DALI library to accelerate data reading and pre-processing operations. DALI provides a set of optimized building blocks for loading and pre-processing image, video, and audio data based on technologies such as GDS. DALI can relieve the computational load on the CPU by shifting some of the pre-processing data operations to the GPU. DALI enables data transfer from NVMe SSD to CPU or GPU without other pre-processing operations. When using DALI for NVMe SSD to GPU transfers, only NVIDIA Tesla and Quadro series GPUs (e.g., A100, V100, andT4) can support native GDS for fast NVMe SSD to GPU DMA transfers without additional CPU and memory load. DALI also provides a CPU mode data read tool for most NVIDIA GPUs, which uses the same POSIX interface as the torch or numpy, which incurs a higher CPU and memory overhead in this mode. DALI has pipeline optimizations for both modes so that the number of pipes can significantly increase the transfer bandwidth when performing data I/O using these modes. It is worth noting that DALI does not support reading *torch.tensor*() format files and that data type conversions must be performed in advance when reading Pytorch tensor data using DALI.

In addition, many recent studies have focused on input data pipelines in the pre-processing phase. Jayashree Mohan et al. proposed CoorDL [25] to reduce data stalls during data pre-processing. Daniel Kang et al. proposed the Smol system [18] for hardware- and input-aware data pre-processing task allocation, efficient memory, and multi-threaded management. Derek G. Murray et al. proposed the tf.data framework [26] to build and execute efficient input pipelines for machine learning. Dan Graur et al. built a fully managed service Cachedw [12] on top of tf.data to dynamically scale distributed resources for data pre-processing. Michael Kuchnik et al. proposed the Plumber tool [20] to analyze the bottlenecks in the input pipeline and automatically adjust parallelism, prefetching, and caching. These approaches are dedicated to solving the efficient pipeline reads for different types of input data in the pre-processing stage and cannot achieve efficient write operations for specific neural network framework data (e.g., torch. tensor) or efficient automatic read and write for models, intermediate feature maps in the training process.

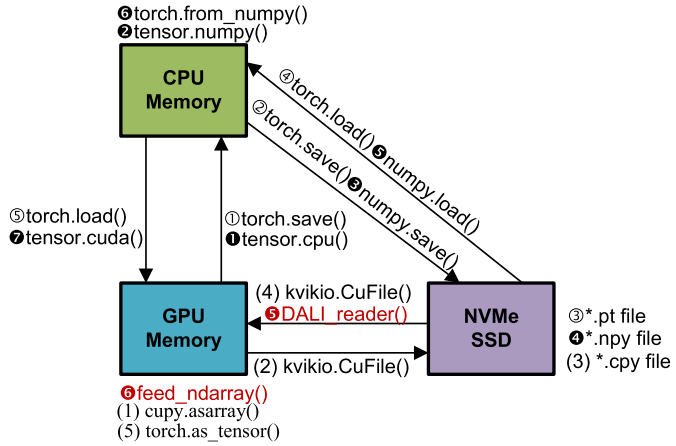

Fig. 3. SSD and GPU Memory transfer paths.

3 ANALYSIS

3.1 Tensor I/O Paths

There are two main methods of transferring tensor data from the GPU to the SSD during training or inference in DNNs: "Transit Transfer" and "GPU Direct Storage".

Transit transfer is the transfer of data from the GPU to the SSD using the DRAM on the CPU side as the "intermediate medium" (Bounce buffer or Page Cache). Transit transfer is the most widely used data transfer method and is the tensor data transfer method supported by most deep learning frameworks. The transit transfer is usually divided into two stages. The first stage uses POSIX interfaces such as Linux *pread*() and *pwrite*() to implement the data transfer from SSD to CPU side DRAM, and the second stage uses interfaces such as CUDA's *cudaMemcpy*() to implement the data transfer between CPU side DRAM and GPU Memory. Typical deep learning toolkits that use transit transfers to implement advanced data reading and writing API wrappers include Pytorch and Numpy.

As shown in paths ①–⑤ in Figure 3, the Pytorch native API uses the *torch.save*() command to implement a transit transfer of Py to r ch tensor data (with the *.pt* suffix) from the GPU Memory to the SSD and the *torch.load*() command for a transit transfer from the SSD to the GPU Memory. The Numpy transfer path is shown as ❶ to ❺. When using Numpy to implement the transfer of Pytorch tensor data from the GPU Memory to the SSD, we first need to transfer the GPU tensor to the CPU DRAM using the *tensor.cpu*() provided by Pytorch, then convert the Pytorch tensor to a numpy array file (with a *.npy* suffix) using *tensor.numpy*() and then transfer $*.npy$ from the CPU DRAM to the SSD using *numpy.save*(). When using Numpy to transfer Pytorch tensors (which have been converted to $*.npy$ format) from SSD to GPU Memory, the $*.npy$ file is first transferred from SSD to DRAM, then the numpy queue file is converted to a Pytorch tensor file using $torch.fromnumpy()$. Finally, the Pytorch tensor file is transferred from the CPU DRAM to the GPU using *tensor.cuda*() or *to*(*device*). Therefore, the transfer I/O path is long and requires the full participation of the CPU and DRAM, resulting in high CPU utilization, high DRAM capacity, and bandwidth consumption.

GDS is the direct data transfer between GPU Memory and SSD. GDS does not burden the CPU and GPU, avoiding additional copies of the CPU DRAM's bounced memory buffer. NVIDIA DALI provides APIs to support the GDS transfer of numpy array format data from the NVMe SSD to the GPU. As shown in ❺ and ❻, DALI first uses the *DALI_reader*() to transfer numpy array format data from SSD to GPU Memory and save it in *DALI_tensor* format, and then uses the *Feed_ndarray*()

Table 2. Read Bandwidth of Different Tools at Different Tensor Sizes (with Buffer/Data Cache)

Size (MB)	Kvikio	DALI_p1	DALI_pb	Torch	Numpy
1,568	1,940.21	2,181.00	**2,417.51**	924.84	1,184.21
784	1,888.70	2,107.98	**2,490.24**	966.22	1,429.51
392	1,930.18	2,114.40	**2,529.81**	1,361.82	1,715.84
196	1,861.60	2,281.83	**2,521.47**	1,410.38	1,779.88
98	1,901.80	1,905.50	**2,532.49**	1,395.02	1,780.20
49	1,772.15	1,586.05	**2,558.59**	1,409.67	1,799.49
24.5	1,788.32	823.85	2,560.14	2,360.31	**2,755.91**
12.25	1,576.64	453.76	2,559.37	2,551.55	**3,479.13**
6.125	1,447.31	447.60	2,555.53	2,718.60	**3,033.68**
3.0625	1,126.75	268.40	2,541.88	2,245.23	**2,542.49**

to convert the *DALI_tensor* format to *torch.tensor* format. It should be noted that DALI does not support writing data from GPU Memory to SSD. The Kvikio transfer path is shown in (1) to (5). To implement a GDS transfer from GPU Memory to SSD using the kvikio component, we first need to convert the Pytorch tensor to a cupy array (with a *.cpy* suffix) at the GPU using the *cupy.asarray*(), and then transfer it to the SSD using the *kvikio.CuFile*(). When fetching data from the SSD back to GPU Memory, the same *kvikio.CuFile*() transfers the cupy array data to GPU Memory, and then the *torch.as_tensor*() is converted into *torch.tensor* form.

3.2 I/O Bandwidth Analysis

We design experiments in this section to verify the actual I/O bandwidth of the real tensor during deep neural network training using different transfer tools between SSD and GPU Memory for different data block sizes. We used the most considerable single-layer output of the ResNet [13] network, a tensor block of shape (batch size, 256, 56, and 56), as the primary test block. We generated data blocks ranging from 3 MB to 1.5 GB by setting the batch size to 1 to 512 (in powers of 2). We generated 100 samples of each size block, with the write bandwidth expressed as the average write bandwidth of these 100 blocks and the read bandwidth expressed as the average read overhead of reading these 100 blocks 10,000 times.

3.2.1 Read Bandwidth Analysis. First, we analyzed the read bandwidth of different tools with the regular use of the buffer/memory cache. The results of this type of experiment are suitable for application scenarios where the same file is read several times over a short period. Tools such as torch and numpy that use "transit transfers" can take advantage of the file system's buffer/memory cache. In contrast, tools based on GDS transfers, such as kvikio and DALI, rarely benefit from the buffer/memory cache.

Table 2 shows that we compared data transfer bandwidth from SSD to GPU Memory using four different API implementations: *torch.load*(), *numpy.load*(), *DALI_reader*() (with a pipeline of 1 (*_p*1) and the best value (*_pb*)), and *kvikio.CuFile*(). The communication bandwidth of *DALI_reader*() is related to other parameters, such as *threads*, which we have tuned to be optimal in our tests. The transmission bandwidth is measured in *MB*/*s*.

Table 2 shows that *DALI* with parallel pipeline acceleration achieves the highest data read bandwidth among all tools when the tensor size exceeds 49 MB (bold part). For smaller tensor sizes, *numpy.load*() achieves the optimal read bandwidth. When the bounded pipeline is 1 (green part), *DALI* still achieves the maximum data read bandwidth among all tools when the tensor size exceeds 98 MB. Similarly, *numpy.load*() achieves the optimal read bandwidth for smaller tensor

Table 3. Read Bandwidth of Different Tools at Different Tensor Sizes (no Buffer/Data Cache)

Size (MB)	Kvikio	DALI_p1	DALI_pb	Torch	Numpy
1,568	2,166.34	2,449.08	**2,498.01**	836.13	1,057.03
784	2,210.32	2,418.26	**2,498.41**	890.91	1,089.31
392	2,218.45	2,314.05	**2,501.60**	847.57	1,198.78
196	1,984.41	2,207.71	**2,262.50**	906.15	1,196.58
98	2,049.35	1,952.19	**2,503.83**	956.10	1,179.30
49	1,683.85	1,615.03	**2,496.18**	844.83	1,323.43
24.5	1,647.61	1,240.51	**2,484.54**	1,079.30	1,456.60
12.25	1,510.48	396.95	**2,196.13**	964.72	1,676.47
6.125	1,161.58	367.75	**2,196.13**	964.72	1,411.62
3.0625	896.78	232.54	**2,119.38**	735.65	1,294.38

sizes. Notably, our experimental results demonstrate that *torch.load*(), which is commonly used, is not optimal in any case. We can achieve up to a 2.60x speedup ratio with *DALI* compared to *torch.load*().

We then analyzed the read bandwidth of different tools without using a buffer/memory cache. To eliminate interference from the buffer/memory cache with read files, we ran "*echo*3 > */proc/sys/vm/drop_caches*" to clear the buffer/memory cache before each read. Table 3 and subsequent experimental results wherever page cache is used include page cache flushing latency. This type of experiment is suitable for scenarios where files are read only once in a short period, which is closer to the actual deep learning process. Examples include reading parameter information of models or checkpoints from SSD at the beginning of training and reading intermediate feature maps temporarily offloaded to SSD during training.

Table 3 shows our experimental results. In pipeline mode, *DALI* achieves the highest throughput at all times. Dali's outstanding performance comes from its underlying GDS technology that enables direct transfer from NVMe SSD to GPU Memory. In addition, Dali implements a series of highly optimized algorithms internally (e.g., small file aggregation, asynchronous reads, multithreading) and data structures that are highly optimized for modern hardware architectures. With a pipeline of 1, *DALI* achieves the highest read bandwidth when the data block size exceeds 98 MB. The highest read bandwidth is achieved by *kvikio.CuFile*() when the data block size is in the range of 12.25 MB–98 MB. The highest throughput is achieved by *numpy.load*() when the tensor data block size is less than 12.25 MB.

3.2.2 Write Bandwidth Analysis. We tested the writing bandwidth from GPU Memory to SSD for each of the three API implementations of *torch.save*(), *numpy.save*() and *kvikio.CuFile*(). As shown in Table 4, *kvikio.CuFile*() achieves the optimal write bandwidth in all cases except when the tensor data block size is 3.06 MB, where *numpy.save*() has the largest write bandwidth. Notably, our experimental results demonstrate that we can achieve up to a 3.38x speedup ratio compared to *torch.save*() using *kvikio.CuFile*(). Kvikio's higher read performance also comes primarily from the underlying implementation of direct data transfer from the GDS-based GPU Memory to the NVMe SSD.

Meanwhile, we found that when the tensor size is small, not using a buffer/memory cache significantly impacts the transfer bandwidth of the torch and numpy tools. In contrast, it has a negligible impact on DALI and kvikio. These results suggest that torch and numpy can take advantage of buffer/memory cache to improve tensor IO speed in some cases. Even though kvikio and DALI should theoretically bypass the system's buffer/memory cache as much as possible when using

Table 4. Write Bandwidth of Different Tools at Different Tensor Sizes

Size (MB)	Kvikio (MB/s)	Torch (MB/s)	Numpy (MB/s)
1,568	**1,460.00**	432.56	748.56
784	**1,435.77**	431.53	739.69
392	**1,213.24**	429.40	725.52
196	**1,486.09**	430.58	701.75
98	**1,214.24**	432.41	688.20
49	**1,408.05**	435.17	669.40
24.5	**1,215.44**	651.60	995.94
12.25	**1,258.76**	716.37	1,012.40
6.125	**1,216.24**	775.32	1,113.64
3.0625	838.35	791.96	**1,182.43**

the GDS technique, the actual behaviour may still be affected by the operating system, file system, and other factors. For example, specific file system operations or metadata accesses may still be related to the buffer/memory cache.

Our results also show that although numpy and torch have similar IO paths, numpy almost always outperforms torch. The main reasons include the following three aspects, which are as follows:

— The type of data that can be stored is different. *numpy.save*() only saves numpy arrays, whereas *torch.save*() can save all kinds of objects, such as models and optimizers. So *numpy.save*() stores simpler types of data and is more efficient when storing and reading.
— The type of file after the storage is different. The *numpy.save*() saves the data as a binary file, while *torch.save*() usually saves the data as a compressed file. Since compression and decompression take some computation time, *torch.save*() will be relatively slow.
— The size of the stored files is different. Since *torch.save*() usually packs multiple objects into a single file, when storing large amounts of data, the resulting file may be larger than *numpy.save*() and therefore take more time to store and read. *torch.save*() is tailored for Pytorch and may include Pytorch-specific metadata, such as computational graphs and gradient information. This can contribute to an increased file size. On the other hand, *numpy.save*() focuses primarily on the array data and some basic metadata like data type and shape, leading to a potentially smaller overhead. Thus, even if the core data is the same size, the file saved using *torch.save*() might be larger than that saved with *numpy.save*() due to these additional metadata considerations. For example, storing a tensor of the same size (3, 32, and 32), the *.pt* file is 537 bytes larger than the *.npy* file.

3.3 Processor Utilization Analysis

In the deep learning process, CPU and GPU resources are very valuable. We want to keep the I/O bandwidth while minimizing the utilization of the CPU and GPU so that the computational tasks running on the processors are interfered with as little as possible. Therefore, this article proposes the $IOef$ metric to evaluate the data I/O efficiency. The $IOef$ metric is calculated as shown in (1) below and represents the ratio of data transfer bandwidth to processor utilization. In the subsequent experimental analysis, we use processor utilization ($Util$) and I/O efficiency ($IOef$) to evaluate the performance of different data transfer APIs.

$$IOef = \frac{I/O\ Bandwidth}{Processor\ Utilisation}. \tag{1}$$

Table 5. CPU Utilization of Different Tools at Batch Size 4

	Util_r	IOBW_r_C (MB/s)	IOef_r_C	IOBW_r (MB/s)	IOef_r	Util_w	IOBW_w (MB/s)	IOef_w
Torch	0.991	2,551.552	2,574.724	1,042.553	1,052.021	1.032	716.374	694.161
Numpy	0.999	3,479.125	3,482.608	1,676.475	1,678.153	1.074	1,012.397	942.641
Kvikio	**0.350**	1,576.637	**4,504.678**	1,510.481	4,315.660	0.829	1,346.154	**1,623.829**
DALI_best	0.629	2,559.372	4,068.895	2,229.299	3,544.196	N/A	N/A	N/A
DALI_1	0.892	2,544.853	2,852.974	744.228	834.337	N/A	N/A	N/A

Table 6. GPU Utilization of Different Tools at Batch Size 4

	Util_r	IOBW_r_C (MB/s)	IOef_r_C	IOBW_r (MB/s)	IOef_r	Util_w	IOBW_w (MB/s)	IOef_w
Torch	0.140	2,551.55	18,225.4	1,042.55	7,446.81	0.050	716.374	14,327.5
Numpy	0.280	3,479.13	12,425.4	1,676.48	5,987.41	0.070	1,012.40	14,462.8
Kvikio	**0.010**	1,576.64	**157,663**	1,510.48	**151,048**	0.010	1,346.15	**33,653.8**
DALI_best	0.020	2,559.37	127,968	2,229.30	111,464	N/A	N/A	N/A
DALI_1	0.020	2,544.85	127,242	744.228	37,211.4	N/A	N/A	N/A

3.3.1 CPU Utilization Analysis. Similarly, we tested processor utilization in torch, numpy, kvikio, and DALI (best pipeline and one pipeline), respectively, with and without buffer/memory cache.

The test results of CPU utilization and I/O efficiency at a batch size of 4 are shown in Table 5. The r and w subscripts represent read and write operations. The $\_C$ in the subscript represents using buffer/memory cache. The processor utilization in the table represents the average utilization when data is transferred steadily. The experimental results show that the kvikio tool based on GDS technology achieves the highest CPU I/O efficiency regardless of whether the buffer is used.

3.3.2 GPU Utilization Analysis. GPU utilization was tested with a batch size of 4, similar to the CPU. Table 6 presents the results of the GPU utilization tests. The experimental results indicated that the kvikio tool based on GDS technology achieved the highest GPU IO efficiency with or without buffer, in line with CPU IO efficiency.

The experimental results show that the GDS-based kvikio library is the best for energy efficiency for CPU and GPU reads and writes, followed closely by the GDS-based DALI library. The GDS-based data transfer achieves better energy efficiency than the traditional POSIX transfer.

3.4 The Impact of Runtime Context on Performance

The extensive use of hardware and software system-level optimization techniques such as instruction prefetching, path prediction, multi-level caching, pipelining, and keyword precedence in modern processor systems heavily influences the performance of a single operation by the context in which it is run. This impact is further divided into two types, predictable and unpredictable impacts. Predictable impacts include the start-up overhead of using a particular API for the first time, the impact of using a buffer on subsequent read speeds, and so on. For example, using DALI to read and write data in bulk, the first batch will be significantly slower than subsequent data (even if GDS is used without going through the CPU side cache). Unpredictable impacts are interactions between seemingly unrelated program statements that are difficult to predict. For example, we have found through our experiments that we are performing $Input images = torch.empty(10, 256, 56, 56).to(device)$ to generate a placeholder Pytorch tensor and transfer it to GPU Memory will speed up *numpy_read* by a factor of 2 or more.

It is easy to notice that even when the same tool is running in different contexts, performing the same operation can result in significant performance differences. Therefore, our proposed *Fastensor* tool captures the actual execution efficiency of each tool based on the actual running of the application, which is more accurate and instructive.

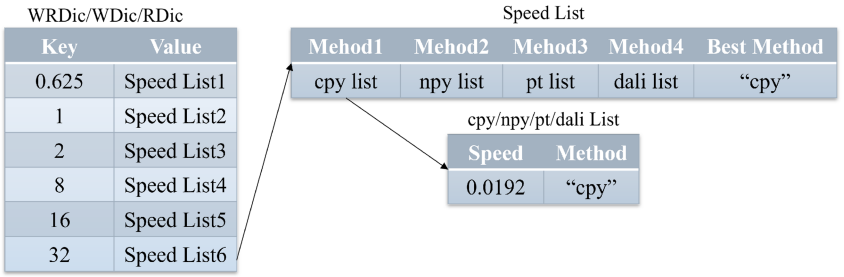

Fig. 4. *WRDic*, *WDic*, and *RDic* structures.

4 METHOD

The analysis in the previous section shows that tensors can be saved as files in many different formats. However, only one data type is supported for training and inference when read into deep learning platforms such as Pytorch. At the same time, we also found that different tensor I/O APIs have different performances when dealing with different data sizes and in different runtime contexts. Therefore, in this section, we propose **Fastensor**, which on the one hand, provides a unified read and write API for a variety of standard tensor transfer tools, on the other hand, allows the optimal tensor transfer tool to be selected based on the tensor block size in a real-world runtime environment, thus improving the bandwidth of tensor transfers.

Fastensor implements the use of different read and write tools depending on the size of the data block by constructing and maintaining three dictionaries, Write Dictionary (*WDic*), Read Dictionary (*RDic*), and Read/Write Dictionary (*WRDic*). The structure of the three dictionaries is similar, as shown on the left side of Figure 4. Each dictionary uses the size of the data block (in MB) as the key and the list of speeds transferred by different data transfer tools at that size as the value. As shown in the top right of Figure 4, the speed list is a heterogeneous list of length 5. Its first four items are a list of methods consisting of the speed and tool name of different tools (note that the order of different methods in the list may be different for different keys), and its fifth item is the name of the best method selected according to the first four items. The method list is a list of length 2, with item 1 representing the execution speed using the execution time (the smaller the time, the faster the speed), and item 2 is the method name saved in string format. In Figure 4, for example, the dictionary records the speed and optimal method of transfer for six different sizes of data blocks using four tools, where the fastest method for a 32 MB size block is "cpy" (using the kvikio tool), which is recorded in the table with a transfer time of 0.0192s.

WDic, *RDic*, and *WRDic* can either be specified manually (specify the path of the dictionary when the *Fastensor* API is called) or obtained automatically by *Fastensor* during training (specify the path of the dictionary as None when the *Fastensor* API is called). When the automatic method is used, each dictionary can be constructed entirely during training using only four iterations. The write overhead for constructing a dictionary is only about 2 KB. At the same time, each training session requires tens or hundreds of thousands of iterations, and each iteration generates hundreds of MB or even GB of intermediate feature map reads and writes, so the dictionary construction overhead is negligible compared to the training overhead. Moreover, since *Fastensor* constructs dictionaries by sensing the read and write performance during training, the dictionaries constructed by *Fastensor* automatically change with the hardware and software environment. Users do not need to adjust *Fastensor* according to the changes in hardware and software platforms.

In addition, the new I/O library can be easily extended to *Fastensor*. The new I/O library must only be added to the Speed List candidate methods and can be automatically updated to the *WDic*,

RDic, and *WRDic* dictionaries during training. The prerequisite is that the new I/O library must support reading and writing to the Pytorch tensor or provide an API to convert the data to the Pytorch tensor format.

4.1 Fastensor Save

As shown in the Algorithm 1, *FastensorSave* can be roughly divided into the following five steps:

1. **Tensor Size Calculation:** Calculate the size of the tensor and convert it to MB units, typically with a function like $sys.getsizeof(Inputtensor.storage())/1024/1024$.
2. **Dictionary or Tool Determination:** If a custom dictionary is available, *Fastensor* selects the tensor save tool based on this. If not, *Fastensor* checks if a user-specified tool for tensor saving is provided. *Fastensor* proposes a uniform interface to different tensor transfer tools. Without a specified tool, decide whether the dictionary requires updating. The user manually sets the update frequency, e.g., no update, update at 10,000 iterations of training, update every several epochs, and so on. More frequent updates lead to more efficient transfers but introduce additional overhead, and users must select a suitable trade-off between them.
3. **Write–Read Mode Check:** If the user specifies a write–read mode ($w + r$) and the tensor size appears in *WRDic*, locate the optimal save tool using the tensor size and proceed with saving.
4. **Read Mode Check:** If only read mode (r) is specified, and the tensor size exists in *RDic*: Verify if the speed list for the size includes all methods. If all methods are listed, save data using the top method in *RDic*. If not, jump to the next step.
5. **Write Mode Process and Write Dic Update:** If the tensor size is not in *WDic*, save using the *cpy* method and record the speed for the size. If the size is already present in *WDic*: With a speed list length of 1, use the *npy* tool for saving and updating *WDic*. For a length of 2, employ the *pt* tool and update. If the length reaches 3, use the *DALI* tool, select the best save method, and update *WDic*.

4.2 Fastensor Load

As shown in the Algorithm 2, *FastensorLoad* can be roughly divided into the following five steps:

1. **Tensor Information Retrieval:** Using the input file, obtain the tensor size, file type, and storage address space.
2. **Check and Finish** *RDic***:** If all load methods are present, select the optimal method and update the speed list for size. If not, proceed to step 4.
3. **Create** *WRDic***:** If $WDic[Size]$ and $RDic[Size]$ contains all tools and has an optimal tool: Create $WRDic[Size]$ from$WDic[Size]$ and $RDic[Size]$ and update *WRDic*.
4. **Tool Type Judgement:** Determine the tool type, either “cpy”, “npy”, “pt”, or “DALI”. Load the tensor using the identified tool and record the load time.
5. **Update** *RDic***:** If size is not found in *RDic*: Create $RDic[Size]$ and store the load information. If the size is present, check if the method exists in its speed list. If it does not exist, add the load information to $RDic[Size]$. If it does, it ends the load process.

4.3 Fastensor in Multi-GPU Systems

Considering the widespread use of multi-GPU systems in deep learning training processes, we discuss the combination of *Fastensor* with the dominant data parallelism and model parallelism paradigms in this section. Data and model parallelism represent the predominant strategies employed in multi-GPU training scenarios. Data parallelism involves distributing input data across multiple GPUs, where each GPU computes the forward and backward passes for its subset of the

ALGORITHM 1: Fastensor Save

Require: InputTensor T
Ensure: InputTensor Storage Format in SSD F

$S \leftarrow$ getsizeof(T) {Tensor Size Calculation}
if Customized Dictionary or Customized Format is not None {Dictionary or Tool Determination} **then**
  $F \leftarrow$ Customized Dictionary Specific Format or $F \leftarrow$ Customized Format
  return F
end if
if *FlushDic* is True **then**
  Flush all Dics
end if
if User use "w+r" mode and S in *WRDic* {Write-Read Mode Check} **then**
  $F \leftarrow$ *WRDic* Specific Format with S as key
  return F
end if
if User use "r" mode and S in *RDic* and len(*RDic*[S]) == 5 {Read Mode Check} **then**
  $F \leftarrow$ *RDic* Specific Format with with S as key
  return F
end if
if S in *WDic* {Write Mode Process and Write Dic Update} **then**
  if len(*WDic*[S]) == 5 **then**
    $F \leftarrow$ Get value from *WDic* with S as key
  else if len(*WDic*[S]) == 1 **then**
    $F \leftarrow$ "npy" and update *WDic*
  else if len(*WDic*[S]) == 2 **then**
    $F \leftarrow$ "pt" and update *WDic*
  else
    $F \leftarrow$ "DALI" and update *WDic* then Choose best method and update *WDic* again
  end if
  return F
end if
$F \leftarrow$ "cpy" and update *WDic*
return F

data. On the other hand, model parallelism involves splitting the model itself across multiple GPUs due to its size or optimizing compute distribution.

4.3.1 *Data Parallelism Integration with Fastensor.*

Heterogeneous Multi-GPU Systems: For varied GPU specifications, *Fastensor* can operate individually, deriving a unique tensor data transfer strategy per GPU, ensuring specificity and optimized performance. However, the trade-off might result in a slight increase in the overhead of strategy generation.

Homogeneous Multi-GPU Systems: In systems with identical GPUs, a single strategy determined by one GPU can be broadcasted across all, minimizing overhead but potentially sacrificing a minute amount of strategy optimization. For specific implementations, *Fastensor*

ALGORITHM 2: Fastensor Load

Require: Inputfile *File*, File Size *S*, File Path *P*, File Type *Type*
Ensure: InputTensor Format *F*

```
[S, P, Type] ← getinformation(File) {Tensor Information Retrieval}
if len(RDic[S] ) == 4 {Check and Finish RDic} then
    Choose best method and update RDic[S]
end if
if len(WDic[S] ) == 5 and len(RDic[S] ) == 5 then
    Create WRDic[S] and update WRDic {Create WRDic}
end if
if Type == cpy {Tool Type Judgement} then
    F ← "cpy"
else if Type == npy then
    F ← "cpy"
else if Type == pt then
    F ← "pt"
else
    F ← "dali"
end if
if S in RDic then
    if Type not in RDic[S] then
        Update RDic[S] {Update RDic}
    end if
else
    Create RDic
end if
return F
```

can be easily integrated with existing data parallelism toolkits (e.g., torch.nn.Dataparallel and torch.nn.DistributedDataParallel) to implement the above strategies.

4.3.2 Model Parallelism Integration with Fastensor. Given the segmentation of models across GPUs, it is imperative for each GPU, handling unique model segments, to establish its distinct tensor data transfer strategy to account for varying tensor sizes and requirements. Similarly, *Fastensor* can be integrated with existing distributed training toolkits (e.g., torch.distributed) to realize model parallelism with automatic tensor transfer capabilities.

In a multi-GPU environment, the actual implementation might necessitate attention to several finer details. Examples include the following: (1) Managing the nuances of inter-GPU communication latencies. (2) Refining GPU Memory management strategies. (3) Ensuring robust synchronization mechanisms in data-parallel setups. A thorough examination of these intricacies in practical deployments would further enhance the system's efficacy.

5 EVALUATION

5.1 Experimental Settings

5.1.1 Experimental Environment. We have conducted experiments on typical CV models and NLP models for the two most prominent scenarios of tensor reading and writing, model parameter saving, and intermediate feature map reading and writing, respectively.

The main hardware and software configurations in our experiments are shown in Table 7.

5.1.2 Experimental Models. In experiments on model parameter saving, we conducted experiments on the latest Vision Transformer (ViT_b_32ViT_H_14) [7] and Swin Transformer (Swin) [23]

Table 7. Hardware and Software Environment in the Experiments

Hardware	**Version**
CPU	Intel(R) Xeon(R) Silver 4210R
GPU	NVIDIA A100
NVMe SSD	Samsung MZ-V8V1T0BW
Software	**Version**
CUDA	11.7.64
Pytorch	1.12.1
DALI	1.18.0
Numpy	1.23.1
Kvikio	22.08.0
Cupy	11.2.0

models, as well as the classic VGG19 [32], ResNet152 [13], Wide ResNet 50 [38] models, the lightweight deployable mobile MobileNetV3 [15], and a Toy model (see Section 5.2.2 for the structure of the Toy model) for CV, and the Chinese Bert model (Bert_wwm) [6] for NLP. We ran 100 epochs on each model, saving the model parameters once per epoch.

In the *Fastensor* experiments for intermediate feature map transfer, we performed image classification training tasks for a Toy model using the Cifar-10 dataset and the latest Vit_H_14 model using the Flower102 dataset, respectively. All models in the CV experiments use the SGD optimizer, and the learning rate value is set to 0.1. This study does not focus so much on the final model training accuracy but instead on the difference in accuracy between transferring the intermediate feature maps using different transfer tools and not transferring the intermediate feature maps.

We used the THUCNews dataset for the headline classification training task for the Bert_wwm model of NLP. The dataset contains 80,000 training samples and 10,000 validation and test samples each. All models were trained using the *Fastensor* to offload the model and intermediate feature maps from the forward process to the SSD and to fetch the GPU Memory from the SSD during the backpropagation process. All experiments used the AdamW optimizer with learning rate values set to $2e-5$ and weight_decay to $1e-4$. The $RoBERTa-wwm-ext$ model parameters pre-trained on datasets such as Chinese Wikipedia were used in our experiments.

Before all the experiments began, we first conducted a pre-experiment. We tested a Toy model (three (convolution + maxpool2d) layers, one flattened layer, two linear layers) to confirm that each read/write tool has an approximate read/write speed when offloading and reading intermediate feature maps from GPU Memory to SSD. As shown in Table 8, we found that DALI has nearly two orders of magnitude higher read/write overhead than the other methods. These results are because DALI has a significant start-up overhead and is designed for bulk tensor read scenarios, so we did not continue to use the DALI tool in our subsequent experiments and excluded DALI from the speed list of *Fastensor*.

We have used *Fastensor* and the three baseline tools kvikio (cpy), numpy (npy), and torch (pt) to perform these experiments. The three baseline tools have been unified in *Fastensor* and can be easily used by specifying the "policy" parameter.

5.2 Experimental Results

Figure 5 shows the training time of *Fastensor*, *Kvikio*, *Numpy*, and *Torch* for Bert and Vit model saving and intermediate feature map transfer. In addition, we show the optimal choices of

Table 8. Toy Model Training Time (Per Epoch) with Transfer

Tool	Time (s)
Fastensor_w	45.8821
Fastensor_r	41.1856
Fastensor_wr	39.8834
Kvikio	44.9453
Torch	64.0384
Numpy	40.4264
DALI	3,601.8675

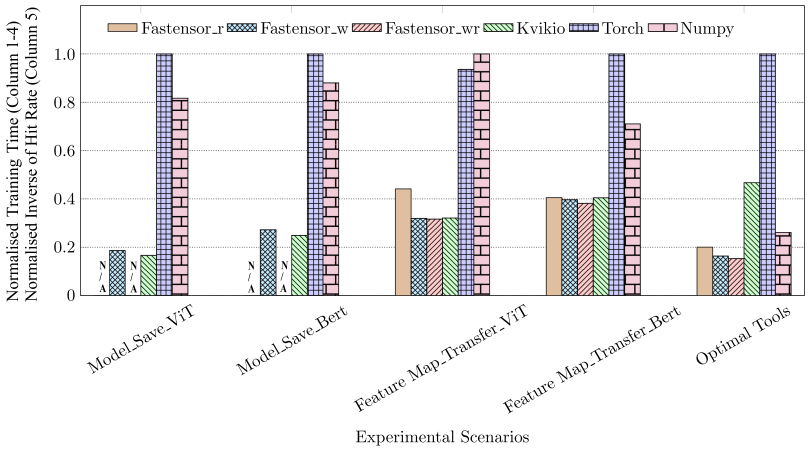

Fig. 5. Normalised training time or inverse of hit rate for different experimental scenarios.

transfer tools for different-size tensor blocks during the intermediate feature map transfer. In Figure 5, the units for the first four columns represent normalized training time, while the unit for the fifth column denotes the normalized inverse of the hit rate. Specifically, the third and fourth columns correspond to batch sizes of 40,960 and 960, respectively, whereas the fifth column pertains to the Bert model with a batch size 64. Since model saving involves only "write" operations, we only compare the transfer speed of the write-mode Fastensor ($Fastensor_w$) with other tools. The experimental results show that $Fastensor_w$ can find the optimal transfer solution for any size model, with a transfer speed up to 5.37x faster than *Torch*. Section 5.2.1 shows more experimental results for model saving. Experimental results for intermediate feature map transfer show that the read-write mode of Fastensor ($Fastensor_{wr}$) achieves the fastest training speed in the training of each model. The $Fastensor_{wr}$ training speed is 2.96x faster than *Torch*. Section 5.2.2 shows more experimental results for intermediate feature map transfer. In addition, we also counted the optimal transfer methods for tensor data blocks of different sizes that were hit by different tools during the intermediate feature map transfer. We use the inverse of the normalized hit rate as an evaluation metric (the smaller the value, the higher the hit rate). The experimental results show that $Fastensor_{wr}$ achieves the highest optimal method hit rate. Specifically, $Fastensor_{wr}$ could select the optimal tensor transfer tool in 93.8% of cases, while *Torch* was the optimal tool in only 14.3% of cases. Section 5.2.3 shows more experimental results for the optimal tools.

Table 9. Model Saving Time of Different Tools

	AlexNet	VGG19	ResNet152	Wide_Res50	Vit_B32	Swin_B	Vit_H14	MobNet_V3	Toy	Bert
Size (MB)	233.1	548.1	230.5	263.1	336.6	335.5	2411	9.830	0.5600	390.2
Fastensor_w (s)	0.1571	0.3493	0.1884	0.1494	0.2154	0.2306	1.148	0.0258	0.0017	0.2174
Kvikio (s)	**0.0142**	**0.2199**	**0.1630**	**0.1584**	**0.1989**	**0.2040**	**1.287**	**0.0246**	0.0089	**0.1990**
Torch (s)	0.6636	1.5027	0.4888	0.5794	0.6819	0.6402	6.9129	0.03344	0.0019	0.7987
Numpy (s)	0.6168	1.2855	0.5436	0.5751	0.7128	0.6876	5.6460	0.0345	**0.0017**	0.7028

5.2.1 Fastensor for Model Parameter Saving. In our experiments with saving CV and NLP models using *Fastensor*, we only compared *Fastensor* write mode (*Fastensor_w*) with other baseline APIs since only 'write operations' are involved. Since kvikio can only offload individual tensors and not uneven lists or dictionaries of tensors, when saving model parameters with kvikio, we reconvert the data type and data size of each tensor in the *values*() list of the parameter dictionary to a new list and then save it with the *key*() list using the numpy() API (this part represents about less than 0.005% of the total number of parameters). We then downscale all the tensors in the *values*() list to 1 dimension and stitch them together in order, with the large tensor blocks saved using the kvikio tool after the stitching is complete (The type and size of the stitched tensor are also saved as a list with *numpy.save*()).

The experimental results are shown in Table 9. Benefiting from the advantages of GDS technology for reading and writing large data blocks, kvikio achieves optimal transfer speeds in all the models (including mobileNetv3 for mobile) we tested except the Toy model, and achieves a 5.37x speedup compared to *torch.save*() in the Vit_H_14 experiment.

At the same time, our proposed *Fastensor_w* can select the optimal offloading method after the first few epochs of exploration, and the offloading efficiency is close to the optimal algorithm for all model sizes. These results mean that the user does not need to know the size of the model but can use *Fastensor_w* to achieve near-optimal transfer efficiency in the first training and to obtain the optimal transfer method for subsequent training (the optimal transfer efficiency can be obtained by specifying the transfer API through the 'policy' interface provided by *Fastensor* in subsequent training).

A significant virtue of our *Fastensor* lies in its inherent adaptability in determining the optimal tensor transfer strategy, irrespective of the preliminary knowledge of the model's size. Such flexibility is paramount, especially when one considers the vast spectrum of model sizes prevalent today. For instance, models tailored for embedded or mobile environments, like LeNet-5, MobileNet V1, or Tiny YOLO, often have compact footprints, potentially less than 1 MB. In such scenarios, employing strategies such as the *Kvikio* with the stitching method might not yield the best results. *Fastensor*'s ability to autonomously pinpoint the best transfer approach without any predisposition about the model parameter size stands as its primary strength, showcasing its broad applicability across various model sizes and deployment scenarios. However, it is worth noting that for known large-scale models, direct adoption of techniques like *Kvikio* can be an effective strategy.

5.2.2 Fastensor for Intermediate Feature Map Transfer. We trained the THUCNews dataset on the Bert model with a batch size of 64, 2,048, and 40,960, respectively. The three batch sizes represent the common batch size, the maximum batch size without transfer, and the maximum batch size with transfer (at which point the size of the intermediate feature map to be offloaded per iteration exceeds 852 GB, which is close to the maximum capacity of the NVMe SSD we use). The intermediate feature map transfer technique is a sacrifice of time for more GPU Memory space. Experimental results show that we can increase the trainable batch size by 20x, which also means that we can increase the trainable size of the model by 20x while keeping the batch size fixed, which has important significance for the improvement of model accuracy and the practical need to train large models with fewer GPUs nowadays.

Table 10. Bert Model (Per Epoch) Training Time (s), Accuracy, and GPU Memory (MB)

	Batch Size 64			Batch Size 2,048			Batch Size 40,960		
Method	**Time**	**Acc**	**GPU Mem**	**Time**	**Acc**	**GPU Mem**	**Time**	**Acc**	**GPU Mem**
Fastensor_r	4,323.7250	0.9625	3,188	2,457.1520	0.9055	6,860	2,489.3058	0.1079	79,282
Fastensor_w	3,801.3562	0.9631	3,166	2,431.1330	0.9057	6,858	2,435.462	0.0937	79,284
Fastensor_wr	**3,683.575**	0.9632	3,164	**2,425.9839**	0.9112	6,858	**2,343.1035**	0.1024	79,282
Kvikio	6,132.7158	0.9646	3,164	2,551.4649	0.9057	6,858	2,485.2156	0.1079	79,282
Torch	4,724.0407	0.9605	3,164	5,863.1019	0.9085	6,858	6,143.4802	0.0940	79,282
Numpy	3,866.9523	0.9618	3,164	3,456.1342	0.9118	6,858	4,364.04631	0.1065	79,282
No_offload	111.2615	0.9607	4,273	87.9	0.9089	42,593	OOM	OOM	OOM

Table 11. Vit_H_14 Model (Per Epoch) Training Time (s), Accuracy and GPU Memory (MB)

	Batch Size 16			Batch Size 64			Batch Size 960		
Method	**Time**	**Acc**	**GPU Mem**	**Time**	**Acc**	**GPU Mem**	**Time**	**Acc**	**GPU Mem**
Fastensor_r	14,198.6679	0.7584	7,658	11,557.4366	0.2483	9,134	10,498.8028	0.0033	49,574
Fastensor_w	10,495.3413	0.7613	7,548	8,479.4639	0.2477	9,146	7,592.3619	0.0033	49,574
Fastensor_wr	**10,323.6271**	0.7749	7,548	**8,370.7920**	0.2509	9,146	**7,524.2893**	0.0032	49,574
Kvikio	12,228.4776	0.7620	7,526	8,966.7357	0.2423	9,077	7,633.9453	0.0032	44,754
Torch	18,166.8237	0.7598	7,494	20,933.1586	0.2453	8,704	22,258.7225	0.0033	39,096
Numpy	22,291.2817	0.7593	7,488	28,640.8777	0.2421	8,506	23,784.3459	0.0033	39,094
No_offload	481.2929	0.7632	20,330	511.2915	0.2496	57,496	OOM	OOM	OOM

As shown in Table 10, compared to other data transfer tools, **our proposed** *Fastensor_wr* **can achieve the highest transfer speed at various batch sizes while keeping the training accuracy constant and without taking up additional GPU Memory**. In particular, at large batch size, *Fastensor* can significantly outperform torch's own I/O tool. Compared to the torch native tool, *Fastensor* is able to improve the end-to-end training speed by 22.02%, 58.62%, and 60.36% on the Bert model at batch sizes of 64, 2,048, and 40,960, respectively.

We trained the Flower102 dataset on the Vit_H_14 model at batch sizes of 16, 64, and 960, respectively. The reasons for these three batch size choices are consistent with the experiments on Bert. As shown in Table 11, *Fastensor_wr* can also achieve the highest transfer speed at various batch sizes when training the Vit model while maintaining the same training accuracy as when it is not transferred, without taking up additional GPU Memory compared to other transfer tools. Compared to the torch native tool, *Fastensor* is able to improve the end-to-end training speed on the Vit_H_14 model by 43.17%, 60.01%, and 66.20% at batchsizes of 16, 64, and 960, respectively.

While the intermediate feature map transfer technique provides a pathway to increase batch size without altering the model, thereby addressing GPU Memory capacity walls and emphasizing its practicality and scalability, it is not its sole merit. An equally vital perspective is that this technique allows for a significant increase in the trainable model size while keeping the batch size consistent. This capability becomes especially valuable when training larger models, particularly in environments with limited hardware resources. We employed the Flower102 dataset to train a deeper Vit_H_14 model in our specific experimentation. To be precise, we kept the batch size constant at 64, ensuring that all other hyperparameters remained unchanged. We progressively increased the num_layers (representing the number of transformer blocks/layers in the model) until the GPU Memory's capacity limit was reached. As shown in Table 12, the experimental results demonstrate that *Fastensor* can increase the supportable training model size by more than 30x while enhancing the training speed by 58.35% compared to *torch.save*.

5.2.3 Changes to Optimal Tool During Training. Subject to the state of the server system, the overhead of transferring a same-sized tensor by the same transfer tool may be different at different stages of training. Therefore, the optimal transfer tool may differ for each tensor at different

Table 12. ViT_H_14 (Per Epoch) Training Time (s) for Different Scales

Tool	Max Layers	Time
Fastensor_r	1,000	247,645
Fastensor_w	1,000	206,720
Fastensor_wr	1,000	197,786
Kvikio	1,000	234,776
Torch	1,000	474,846
Numpy	1,000	508,328
No_offload	32	511.292

training stages. In order to determine how the speed of different data transfer tools varies when transferring tensor data blocks of the same size during the running of the program, we counted the variation of the optimal read/write tools for a given data block size during the training process by periodically clearing the three dictionaries and regenerating them. In Tables 13 and 14, the designation "First" refers to a particular approach adopted by *Fastensor*. Specifically, under this "First" methodology, *Fastensor* creates a dictionary only during the initial few iterations of the training process. For subsequent training phases, instead of dynamically determining the optimal transfer tool at every iteration, we rely on this dictionary. The dictionary maps different sizes of intermediate feature maps to their optimal tools identified during the initial stages. This method aims to reduce the overhead of constant tool selection and leverage the predictability of the best tool for given feature map sizes. Furthermore, in the same tables, the term "Best" represents a strategy that encapsulates our extensive observations during the training process. Specifically, for various tensor sizes encountered, their corresponding optimal transfer tools were repeatedly recorded over multiple iterations. The transfer method that emerged most frequently as the optimal solution is labelled "Best" from this rich dataset. This designation underscores the *Fastensor* tool that consistently provides the best performance for different tensor sizes, given the variability in the state of the server system throughout the training.

In the Bert experiment, we updated the dictionary 1,184 times during training. After each dictionary update, we counted the optimal transfer tools in the speed list for different tensor block sizes. The *WRDic* variation is shown in Table 13. The percentage of optimal transfer tools selected for each tensor size varies. For example, in *WRDic*, a 1.40 MB tensor block achieves optimal transfer speeds when using the "npy" tool in 99.32% of cases, while a 22.50 MB tensor block achieves optimal transfer speeds when using the "cpy" tool in 99.16% of cases. The experimental results also show that the optimal tool for transferring blocks of the same size is runtime context-dependent. Although no single tool can achieve the optimal transfer speed in all cases, in most cases, tensor blocks of a given size tend to prefer a particular tool. Calculations show that we can achieve 79.59% and 88.52% of the theoretically optimal transfers (assuming the optimal transfer tool is selected every time) using the first dictionary of optimal tools obtained at the beginning (column First in Table 13) and using the dictionary of optimal tools obtained statistically (column Best in Table 13), respectively. In contrast, using the torch's own tools is only 16.01%.

Similar to the experiments in Bert, we updated the dictionary 145 times during ViT training and counted the best transmission tools from the list of different tensor block speeds in the dictionary after each dictionary update. As shown in Table 14, 85.43% and 89.22% of the theoretical performance was achieved using the first and best dictionaries, respectively, in the ViT experiments, while the torch was only 8.23%. We also found that the "cpy" tool was consistently optimal at a tensor data block size of 36.75 MB in our training process.

Table 13. Distribution of Optimal Tools in Bert Model Training

Size (MB)	pt	npy	cpy	First	Best
5.34e-5	45	1,139	0	npy	npy
0.0030	239	945	0	npy	npy
0.0040	71	113	0	npy	npy
0.0074	890	294	0	npy	pt
0.0147	148	1,036	0	npy	npy
0.0293	379	805	0	npy	npy
0.1875	849	335	0	pt	pt
0.6592	0	1,184	0	npy	npy
1.4063	7	1,176	1	npy	npy
2.2500	25	910	249	pt	npy
2.6368	0	68	1,116	cpy	cpy
5.6250	0	21	1,163	cpy	cpy
9.0000	0	10	1,174	cpy	cpy
22.5000	0	10	1,174	cpy	cpy

Table 14. Distribution of Optimal Tools in ViT Model Training

Size (MB)	pt	npy	cpy	First	Best
5.34e-5	37	108	0	npy	npy
0.0049	48	97	0	pt	npy
0.0050	46	99	0	npy	npy
0.0628	53	92	0	pt	npy
0.4981	6	139	0	npy	npy
2.8711	1	7	139	cpy	cpy
6.2500	0	19	126	cpy	cpy
6.2745	0	3	142	npy	npy
18.7500	0	10	135	cpy	cpy
20.1566	0	2	143	cpy	cpy
25.0000	0	2	143	cpy	cpy
25.0977	0	6	139	cpy	cpy
36.7500	0	0	145	cpy	cpy
80.3125	0	4	141	cpy	cpy
258.0040	0	4	141	cpy	cpy
321.2500	0	2	143	cpy	cpy

5.3 Result Analysis

5.3.1 Model Saving Time Insights. First and foremost, our observations reveal that the time taken by *Fastensor* during the "save" experiments across all models is consistently suboptimal. This phenomenon is attributable to the inherent methodology of *Fastensor*, during the initial three model saves; it sequentially evaluates various tensor transfer methods to discern the most effective approach. Past this exploration phase, *Fastensor* consistently deploys the optimal transfer method. Consequently, the cumulative model saving time for *Fastensor* slightly trails behind the most efficient approach. However, such a trade-off is justifiable given the substantial merits we garner

in return. By investing a minuscule exploratory overhead, we achieve automated optimal policy selection. *Fastensor* augments algorithmic adaptability and diminishes the cognitive load on users to discern model intricacies. In essence, *Fastensor* enables users to employ the model as a black box, providing a plug-and-play experience with minimal overhead being added.

5.3.2 Performance Variations among Fastensor Variants. Furthermore, closer scrutiny of our results unveils noticeable performance disparities among different variants of *Fastensor* during intermediary feature map transfers. The performance hierarchy is evident: *Fastensor_wr* consistently outperforms both *Fastensor_w* and *Fastensor_r*, with the latter, in specific scenarios, even marginally trailing behind the standalone use of *Kvikio*. This observed behaviour primarily stems from the overhead of intermediate feature map transfers predominantly governed by the "write" operation (transferring from GPU Memory to NVMe SSD). Within each iteration of training, all intermediary feature maps undergo a "write" operation during FP and a subsequent "read" during backpropagation. *Fastensor_wr* discerns the optimal transfer strategy for a given tensor block based on the aggregate overhead of one "read" and "write". Given this strategy's alignment with actual training dynamics, it invariably exhibits superior performance. Conversely, *Fastensor_w* deduces the optimal transfer strategy anchored solely on a tensor block's "write" overhead. Despite this singular focus, *Fastensor_w* still achieves commendable training outcomes, reinforcing the hypothesis that the "write" overhead dominates the transfer time. On the other hand, *Fastensor_r*'s strategy, rooted in the "read" overhead, occasionally defaults to transfer tools optimized for quicker reads but slower writes. This misalignment translates to its elongated training times relative to its *Fastensor* variants.

6 RELATED WORK

6.1 Intermediate Feature Map Transfer Techniques During Training

In order to reduce the GPU Memory utilization during training and increase the supportable model size and batch size, many proposals propose offloading the intermediate feature maps from GPU Memory to external devices during training. Rhu et al. [31] first proposed vDNN to transfer the intermediate feature maps of the convolutional layers from GPU Memory to DRAM on the CPU side during training. Subsequently, Chen et al. [5] and Shriram et al. [1] improved the performance of intermediate feature map transfer from GPU to CPU by selectively offloading some convolutional layers, reducing the number of synchronizations and optimizing GPU fragmentation. Jain et al. [17] and Chen et al. [4] further improve transfer efficiency by introducing compression into intermediate feature map transfers. Huang et al. [16] use a genetic algorithm to determine whether a layer needs to be transferred to CPU DRAM instead of the manual designation in previous work. Hildebrand et al. [14] use linear integer programming to search for a suitable transfer strategy and transfer intermediate feature maps into DRAM + NVM (Non-Volatile memory). Bae et al. [2] implemented GDS-like operations at the C++ level using NVIDIA GDR Copy and SPDK, the first to use SSD as transfer destinations. However, they require the user to maintain their metadata tables.

6.2 Intermediate Feature Map Checkpointing Techniques During Training

In addition to transferring intermediate feature maps to DRAM on the CPU side, another common approach to GPU Memory capacity reduction is to use intermediate feature map checkpoints [30] or called recomputation techniques. Chen et al. [33] was the first to propose recomputation techniques, which selectively release some intermediate feature maps and recompute them using intermediate feature map checkpoints in the backpropagation phase. Wang et al. [35] proposed a system-level optimization framework by combining transfer and recomputation techniques to perform different strategies for different classes of layers. Peng et al. [27] further proposed the MSPS

metric for prioritizing data with less recomputation time and high memory retention to further reduce the cost due to recomputation and additional overhead due to recomputation and transfer.

6.3 GPU and SSD Data Transfer Optimization Research

Besides IO optimization of tensor data between GPU and NVM for deep learning platforms such as Pytorch, many studies optimize more general IO transfers at C++ (CUDA) and SSD architecture levels, respectively.

Similar to flashneuron, Shai Bergman et al. [3] used techniques such as GDR Copy to integrate p2p transfers into the OS file stack, enabling dynamic activation of p2p transfers. Pak Markthub et al. [24] proposed the DRAGON framework using a page fault mechanism to enable a general-purpose GPU application to manipulate large data sets on NVM transparently. Yu-Chia Liu et al. [22] constructed a new multi-dimensional storage architecture NDS to enable applications to access multi-dimensional address spaces and decouple the optimal dataset dimension and storage dimension for applications by dynamically reconfiguring data objects, reducing problems such as data reorganization overhead and underutilization of device bandwidth. These studies were primarily designed to address the problem of not providing complete I/O APIs based on technologies such as GPU Direct RDMA in earlier versions of CUDA (CUDA 7.5 in [3], CUDA 9.0 in [24], and CUDA 10.2 in [22]). Most of these issues have been addressed in the latest CUDA release. We used the GPU Direct Storage API proposed by NVIDIA in CUDA 11.4 or later, which provides complete P2P transfers from the GPU to NVMe SSD without interference with the host. On the other hand, these studies are closer to the underlying computer architecture, and advanced deep learning frameworks based on python platforms, such as Pytorch, cannot use these components directly.

7 CONCLUSION

This article proposes *Fastensor*, a data and runtime context-aware unified interface tensor transfer tool for fast I/O of tensor data. The *Fastensor* consists mainly of *WDic*, *RDic*, and *WRDic*. These three dictionaries sense the read/write performance of different I/O tools at different tensor block sizes in the first few iterations of the actual run. In addition, *Fastensor* provides interfaces allowing users to specify I/O tools or private customized read/write dictionaries. Experiments with typical tensor read and write scenarios have shown that *Fastensor* can adapt to a wide range of I/O tools and significantly improve read and write efficiency. From our experimental observations, the results may inspire future algorithmic designs. For instance, the consistently high performance of *Kvikio* could guide new tensor transfer strategies. While our context-aware approach has yielded notable performance improvements, future endeavours will focus on leveraging reinforcement learning, evolutionary algorithms, and other intelligent methods to further explore adaptive tensor transfer techniques that promise even greater acceleration and adaptability.

REFERENCES

[1] Shriram S. B., Anshuj Garg, and Purushottam Kulkarni. 2019. Dynamic memory management for GPU-based training of deep neural networks. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 200–209. DOI: https://doi.org/10.1109/IPDPS.2019.00030

[2] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-enabled large-batch training of very deep neural networks. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 387–401.

[3] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2018. SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. *ACM Transactions on Computer Systems* 36, 2 (2018), 5:1–5:26. DOI: https://doi.org/10.1145/3309987

[4] Ping Chen, Shuibing He, Xuechen Zhang, Shuaiben Chen, Peiyi Hong, Yanlong Yin, Xian-He Sun, and Gang Chen. 2021. CSWAP: A self-tuning compression framework for accelerating tensor swapping in GPUs. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 271–282. DOI: https://doi.org/10.1109/Cluster48925.2021.00019

[5] Xiaoming Chen, Danny Ziyi Chen, Yinhe Han, and Xiaobo Sharon Hu. 2019. moDNN: Memory optimal deep neural network training on graphics processing units. *IEEE Transactions on Parallel Distributed Systems* 30, 3 (2019), 646–661. DOI: https://doi.org/10.1109/TPDS.2018.2866582

[6] Yiming Cui, Wanxiang Che, Ting Liu, Bing Qin, and, Ziqing Yang. 2021. Pre-training with whole word masking for chinese bert. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), 3504–3514.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the 9th International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=YicbFdNTTy

[8] Hanyu E., Ye Cui, Witold Pedrycz, and Zhiwu Li. 2022. Fuzzy relational matrix factorization and its granular characterization in data description. *IEEE Transactions on Fuzzy Systems* 30, 3 (2022), 794–804. DOI: https://doi.org/10.1109/TFUZZ.2020.3048577

[9] Hanyu E., Ye Cui, Witold Pedrycz, Aminah Robinson Fayek, Zhiwu Li, and Jinbo Li. 2022. Design of fuzzy rule-based models with fuzzy relational factorization. *Expert Systems with Applications* 206 (2022), 117904. DOI: https://doi.org/10.1016/j.eswa.2022.117904

[10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2021. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2021), 1997–2017.

[11] Trevor Gale, Przemek Tredak, Simon Layton, and et al. 2023. NVIDIA DALI. Released on March 15, 2023. [Online]. Available: https://github.com/NVIDIA/DALI. Accessed on May 1, 2023.

[12] Dan Graur, Damien Aymon, Dan Kluser, and Tanguy Albrici. 2022. Cachew: Machine learning input data processing as a service. In *Proceedings of the 2022 USENIX Annual Technical Conference*. 689–706.

[13] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[14] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 875–890. DOI: https://doi.org/10.1145/3373376.3378465

[15] A. Howard, R. Pang, H. Adam, M. Sandler, G. Chu, Liang-Chieh Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, V. Vasudevan, and Quoc V. Le. 2019. Searching for MobileNetV3. In *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision*. IEEE, 1314–1324. DOI: https://doi.org/10.1109/ICCV.2019.00140

[16] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1341–1355. DOI: https://doi.org/10.1145/3373376.3378530

[17] Animesh Jain, Amar Phanishayee, Jason Mars, Gennady Pekhimenko, and Lingjia Tang. 2018. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi (Eds.). IEEE Computer Society, 776–789. DOI: https://doi.org/10.1109/ISCA.2018.00070

[18] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Matei Zaharia, and Peter Bailis. 2020. Jointly optimizing preprocessing and inference for DNN-based visual analytics. *Proceedings of the VLDB Endowment* 14, 2 (2020), 87–100.

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, Yoshua Bengio and Yann LeCun (Eds.). Retrieved from http://arxiv.org/abs/1412.6980

[20] Michael Kuchnik, Ana Klimovic, Jiri Simsa, George Amvrosiadis, and Virginia Smith. 2022. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems* 4 (2022), 33–51.

[21] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 3288–3298. DOI: https://doi.org/10.1145/3534678.3539070

[22] Yu-Chia Liu and Hung-Wei Tseng. 2021. NDS: N-dimensional storage. In *Proceedings of the MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 28–45.

[23] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and B. Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the 2021 IEEE/CVF International Conference on Computer Vision*. IEEE, 9992–10002. DOI: https://doi.org/10.1109/ICCV48922.2021.00986

[24] Pak Markthub, Mehmet E Belviranli, Seyong Lee,Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 414–426.

[25] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment* 14, 5 (2021), 771–784.

[26] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf. data: A machine learning data processing framework. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2945–2958.

[27] Xuan Peng, Xuanhua Shi, Hulin Dai,Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU memory management for deep learning. In *Proceedings of the ASPLOS '20: Architectural Support for Programming Languages and Operating Systems* , James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 891–905. DOI: https://doi.org/10.1145/3373376.3378505

[28] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 2227–2237. DOI: https://doi.org/10.18653/v1/n18-1202

[29] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick,Kaiming He, and Piotr DollÃąr. 2020. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10428–10436.

[30] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: Breaking the GPU memory wall for extreme scale deep learning. In *Proceedings of the SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59:1–59:14. DOI: https://doi.org/10.1145/3458817.3476205

[31] M. Rhu, N. Gimelshein, J. Clemons,A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 18:1–18:13. DOI: https://doi.org/10.1109/MICRO.2016.7783721

[32] K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations*, Yoshua Bengio and Yann LeCun (Eds.). Retrieved from http://arxiv.org/abs/1409.1556

[33] C. Tianqi, X. Bing, Z. Chiyuan, and G. Carlos. 2016. Training deep nets with sublinear memory cost. arXiv:1604.06174. Retrieved from https://arxiv.org/abs/1604.06174

[34] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2023. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[35] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 41–53. DOI: https://doi.org/10.1145/3178487.3178491

[36] Jia Wei and Xingjun Zhang. 2022. How much storage do we need for high performance server. In *Proceedings of the 38th IEEE International Conference on Data Engineering*. IEEE, 3221–3225. DOI: https://doi.org/10.1109/ICDE53745.2022.00303

[37] Guangfeng Yan, Tan Li, Shao-Lun Huang,Tian Lan, and Linqi Song. 2022. AC-SGD: Adaptively compressed SGD for communication-efficient distributed learning. *IEEE Journal on Selected Areas in Communications* 40, 9 (2022), 2678–2693. DOI: https://doi.org/10.1109/JSAC.2022.3192050

[38] S. Zagoruyko and N. Komodakis. 2016. Wide residual networks. In *Proceedings of the British Machine Vision Conference 2016, BMVC 2016*, Richard C. Wilson, Edwin R. Hancock, and William A. P. Smith (Eds.). BMVA Press.

[39] M. D. Zeiler. 2012. Adadelta: An adaptive learning rate method. arXiv:1212.5701. Retrieved from https://arxiv.org/abs/1212.5701

[40] Barret Zoph and Quoc V. Le. 2017. Neural architecture search with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=r1Ue8Hcxg

Received 23 May 2023; revised 25 August 2023; accepted 24 September 2023