FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation

Taeyoon Kim
UNIST
tykim8191@unist.ac.kr

ChanHo Park
UNIST
pch8286@unist.ac.kr

Mansur Mukimbekov
UNIST
mansur@unist.ac.kr

Heelim Hong
UNIST
heelim@unist.ac.kr

Minseok Kim
UNIST
minseok1335@unist.ac.kr

Ze Jin
ByteDance
ze.jin@bytedance.com

Changdae Kim
ETRI
cdkim@etri.re.kr

Ji-Yong Shin
Northeastern University
j.shin@northeastern.edu

Myeongjae Jeon
UNIST
mjjeon@unist.ac.kr

ABSTRACT

Data augmentation enhances the accuracy of DL models by diversifying training samples through a sequence of data transformations. While recent advancements in data augmentation have demonstrated remarkable efficacy, they often rely on computationally expensive and dynamic algorithms. Unfortunately, current system optimizations, primarily designed to leverage CPUs, cannot effectively support these methods due to costs and limited resource availability.

To address these issues, we introduce FusionFlow, a system that cooperatively utilizes both CPUs and GPUs to accelerate the data preprocessing stage of DL training that runs the data augmentation algorithm. FusionFlow orchestrates data preprocessing tasks across CPUs and GPUs while minimizing interference with GPU-based model training. In doing so, it effectively mitigates the risk of GPU memory overflow by managing memory allocations of the tasks within the GPU-wide free space. Furthermore, FusionFlow provides a dynamic scheduling strategy for tasks with varying computational demands and reallocates compute resources on the fly to enhance training throughput for both single and multi-GPU DL jobs. Our evaluations show that FusionFlow outperforms existing CPU-based methods by 16–285% in single-machine scenarios and, to achieve similar training speeds, requires 50–60% fewer CPUs compared to utilizing scalable compute resources from external servers.

PVLDB Reference Format:
Taeyoon Kim, ChanHo Park, Mansur Mukimbekov, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon.
FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation. PVLDB, 17(4): 863 - 876, 2023.
doi:10.14778/3636218.3636238

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/omnia-unist/FusionFlow.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636238

1 INTRODUCTION

Deep learning (DL) training is a time-consuming process primarily due to compute-intensive mathematical operations that arise during training. To expedite these computations, DL systems utilize accelerators such as GPUs. In large-scale training, it is common to parallelize each training iteration across multiple GPUs to further improve training performance and efficiency [25, 43, 59, 75].

However, as HW technologies continue to evolve, the bottleneck in DL systems is gradually shifting from the GPU to the CPU. Over the last decade, GPUs have undergone substantial improvements, with FLOPs per dollar doubling every two years [23]. In contrast, improvements in CPUs have been relatively stagnant. This disparity in speed enhancement results in a low CPU-to-GPU compute ratio in many cases [39, 44], posing a challenge in effectively balancing the DL workload over CPUs and GPUs within a server.

The data preprocessing (prep) step in DL training, which typically runs on the CPU, exacerbates this bottleneck by placing extra computational burdens on the CPU [6, 10, 30, 45, 47]. This step precedes model training on each mini-batch of the training dataset and applies a multi-stage data transformation pipeline called *data augmentation* to enhance data diversity and improve model accuracy. Modern data augmentation algorithms [12, 13, 31, 34–36, 41, 46, 51, 53, 69, 71, 74, 76] *dynamically* choose a *larger* set of transformation operators, further enhancing model accuracy at a higher computational cost. Because the enhancement comes without improving the training algorithm or using a larger dataset, these data augmentation techniques are widely adopted by mainstream DL frameworks such as PyTorch [64] and TensorFlow [63].

To hide the delays from the CPU bottleneck, the data prep step for the next mini-batch runs concurrently with the current model training [5, 42, 70], but the mini-batch is often not ready on time for training. Thus, recent studies have explored various methods to expedite the data prep step by utilizing remote CPUs beyond the local server's capacity [7, 18, 66, 72, 73]. This horizontal scaling of data prep has been primarily achieved through disaggregated services [7, 18, 72], where DL systems allocate ample CPU resources for the data prep to match the speed of model training on the GPU. To further enhance scaling efficiency, a recent DL system called FastFlow [66] automates decisions concerning the optimal amount of mini-batch

data to process on remote CPUs. To make such decisions, FastFlow profiles a set of predefined candidate computations for offloading, taking into account various remote compute and network capacities.

On the flip side, relying on remote CPUs to scale the data prep step is not a panacea and has two limitations. First, its applicability is restricted as it necessitates scalable compute resources, such as CPU clusters [7, 18], which are not readily available to most ML practitioners. Second, CPUs are typically optimized for sequential processing tasks, whereas data prep for DL tasks – for example, on image data – can benefit substantially from extensive data-parallel processing. Consequently, approaches solely based on remote CPUs can incur high costs. If an insufficient number of remote CPUs is allocated, the data prep step will still remain as a bottleneck and prolong model training times [66].

In this paper, we introduce FusionFlow, a new DL system that speeds up the data prep step without dependence on remote compute resources. The key strategy employed by FusionFlow is harnessing idle GPU cycles, which enables the dynamic offloading of a portion of CPU computations (specifically related to data prep in the imminent iteration) onto local GPUs. Importantly, FusionFlow's design principles complement horizontal CPU scaling. Thus, if remote CPUs are at hand, FusionFlow can seamlessly integrate them into its CPU pool and adjust the distribution of computations among all accessible CPUs and GPUs, further speeding up the data prep.

To take the opportunity for GPU offloading, we retrofit one of the libraries that offer HW-assisted operations for data prep tasks [45, 47]. However, existing libraries are primarily tailored for a single static pipeline that is repeatedly used to transform *all training samples*. They are not well-suited for dynamic algorithms that frequently change the data transformation pipeline (e.g., per sample) on the fly. This dynamic nature incurs a substantial overhead which encompasses task generation, GPU memory allocation, and task scheduling on the GPU each time the data transformation undergoes a change. Furthermore, these libraries manage a local pool of GPU memory to optimize memory allocations, but this pool can grow to several GBs [14, 40], limiting available space for model training and increasing the risk of GPU memory overflow.

To overcome these challenges, we improve our system by incorporating two design principles: *ahead-of-time task allocation* and *cross-boundary memory sharing*. When a DL job is spawned, FusionFlow pre-builds GPU tasks for all possible data augmentation pipelines. During runtime, it executes one of the pre-built tasks that reflects a random augmentation pipeline by simply updating the input data of the chosen task, thereby eliminating most of the runtime overhead. In addition, FusionFlow unifies local memory pools between the data prep and model training phases. This integration allows any unused GPU memory to be reclaimed for future memory allocations, effectively reducing the GPU's memory footprint.

For each mini-batch, FusionFlow exploits *intra-batch parallelism* [42, 70] to distribute the data prep workload between CPU and GPU. Since GPUs are specifically optimized for processing large data blocks in parallel, FusionFlow assigns relatively larger samples from a mini-batch to the GPU while keeping both CPUs and GPUs busy. To generalize this idea, we propose *task affinity-aware scheduler*. This scheduler initially divides each mini-batch into many small *tiny-batches*, with each tiny-batch containing a small subset of samples (even one sample). These tiny-batches are then sorted based on the aggregate size of samples in each tiny-batch and fed into the CPU and GPU in ascending and descending orders, respectively. To load balance across the CPU and GPU while accommodating dynamic changes in the CPU side's compute capacity (e.g., variations in # local/remote CPUs), our task scheduler continuously monitors the number of remaining tiny-batches and the processing rates of all available compute resources and adaptively adjusts task placements.

We opt for data-level partitioning over operator-level partitioning for several compelling reasons. Operator-level partitioning splits data prep operators (e.g., image decoding) between CPU and GPU while, for instance, reserving the GPU for those resource-intensive operators that can yield substantial speedup improvements. However, this partitioning scheme tends to be coarse-grained due to the limited number of operators involved in data prep. In certain DL frameworks like TensorFlow, the data augmentation phase is even treated as a single, monolithic operator [66], further constraining the options for operator partitioning plans. Consequently, it is difficult to effectively distribute the computational load across the CPU and GPU through operators. Additionally, since data prep operators are usually executed sequentially for a mini-batch, it is crucial for mini-batch samples to seamlessly flow into these operators to fully harness all available compute resources. This introduces additional system complexity, which, if overlooked, could adversely impact training performance.

We implement FusionFlow atop PyTorch and demonstrate the benefits by comparing it with a concrete set of baselines, including NVIDIA DALI [45], a popular GPU-based method, and traditional CPU-based methods, including FastFlow [66], which represents the state-of-the-art horizontal CPU scaling. Our evaluations show that FusionFlow is 46.7–91.7% faster than DALI and 16.1–285% faster than the local CPU-based methods and requires 50–60% fewer CPUs compared to FastFlow to achieve comparable performance. Moreover, FusionFlow can support 50% larger batch sizes compared to DALI due to cross-boundary memory sharing and exploit in-memory caching [30] to deliver even higher throughput.

2 BACKGROUND

DL training overview. Deep learning (DL) training involves processing a dataset multiple times termed *epochs*. Each epoch consists of many training iterations that repeat data preprocessing (prep) and model training. In the data prep step, a *mini-batch* of data is randomly fetched, decoded (e.g., JPEG to RGB format), and then augmented to generate a batch of new samples. These augmented samples are used in the model training step, where computations are performed on the data as it passes through DL layers in order and then in reverse order to calculate gradients for model updates. DL training continues these steps until the model converges.

There are various methodologies for performing parallel DL training using multiple GPUs. For example, *data parallelism* allows each GPU to train the model using a disjoint subset of the training data. As the training proceeds with local mini-batches on different GPUs in parallel, the gradients obtained during the model training step are periodically aggregated to synchronize model updates. The communication graph that specifies workers to perform the gradient aggregation includes either all GPUs (e.g., All-Reduce [58]) or a small subset [32, 37, 38]. Also, data-parallel training has a staleness

bound that determines the maximum iteration gap allowed among workers [9, 15, 16, 22]. Although specific setups do not restrict our work, we mainly consider data parallelism based on All-Reduce and no staleness since it is the most commonly used form.

Dynamic data augmentations. While conventional data augmentation pipelines in the data prep step are statically configured, modern ones create the pipeline dynamically at runtime [12, 13, 31, 34–36, 41, 46, 51, 53, 69, 71, 74, 76]. For example, the first data augmentation algorithm used in popular CNN models like ResNet [20] and VGGNet [60] consists of three transformation operators (crop, scale, and horizontal flip) [28, 60] applied to *"all input samples"*. On the contrary, RandAugment [13] – the most well-known dynamic algorithm – *randomly* selects two or more out of 14 operators *"per sample"* and concatenates them with crop and horizontal flip [26, 30]. Deep AutoAugment [74] goes even further by augmenting each sample using five to seven random operators. As a result of using longer pipelines formed by the dynamic selection of richer transformation operators, these augmentation methods enhance the training accuracy for many popular models, including ResNet, EfficientNet, and ViT, to name a few [13, 30, 62, 66].

3 CHALLENGES

In this section, we shed light on various challenges associated with the efficient execution of data prep tasks. For brevity, we will use the term "task" to refer to both input samples and the code required to perform data prep unless specified otherwise.

3.1 Data Prep on CPU

The enhanced accuracy of dynamic data augmentations comes with a high computational cost that often becomes a major source of performance degradation. To demonstrate this, we characterize how well commonly used DL systems fulfill fast training. We train ResNet-50 (RN-50) and ViT-Base (ViT) on two servers equipped with NVIDIA RTX 3090 Ti and A100 GPUs using PyTorch 1.8 [48] and apply two recent augmentation methods, RandAugment and AutoAugment [12], on OpenImage dataset [29]. For comparison, we include default augmentation algorithms (DefAugment) used in the PyTorch.

To evaluate computation costs and potential parallelism speedups without interference, training is conducted on a **single CPU and GPU** that does not need model synchronization. Furthermore, to minimize the impact of I/O-related delays, our data prep proactively prefetches subsequent mini-batches for augmentation in advance. Our workload characterization reveals two challenges:

C-1. High computational cost. DL frameworks typically leverage multiple CPU cores to parallelize data prep across samples, aiming to have the next augmented mini-batch steadily ready for model training as soon as the current iteration is completed [42, 70]. When this condition is met, GPUs remain constantly occupied with model training, avoiding CPU bottlenecks that arise from data prep.

Figure 1(a) shows the average time required for data prep (t_D) and model training (t_M) to process a single sample using the mini-batch size that fits in the GPU memory of the RTX 3090 Ti server. t_D takes 17.3×, 14.7×, and 11.2× longer than t_M for RandAugment, AutoAugment, and DefAugment in ResNet-50, respectively. Thus, assuming that performing data prep across CPUs for different samples is "embarrassingly parallel", RandAugment, AutoAugment, and

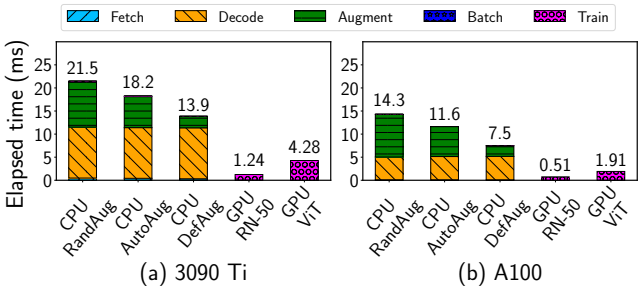


Figure 1: Per-sample processing time on CPU and GPU for different augmentation algorithms, models, and machines.

DefAugment require "at least" ⌈17.3⌉, ⌈14.7⌉, and ⌈11.2⌉ CPUs, respectively, to match the model training speed without encountering a CPU bottleneck. However, this server's CPU-to-GPU ratio stands at only 5.3:1, which is slightly higher than the ratio observed in popular NVIDIA's AI-optimized servers like DGX-1 [4] and DGX-2 [2]. Subsequently, all these scenarios in ResNet-50 face CPU bottlenecks, with RandAugment exacerbating the problem further. Note that due to the higher t_M, ViT is expected to experience less severe CPU bottlenecks than ResNet-50.

Then, does using a newer server with more CPU cores resolve this problem? We similarly measure the average t_D and t_M on the A100 server, which provides up to 16 CPU cores per GPU. As Figure 1(b) shows, both t_D and t_M exhibit considerable reductions compared to their counterparts in Figure 1(a). For DefAugment, t_D takes 14.5× longer than t_M for ResNet-50. This suggests that the server's CPU-to-GPU ratio of 16:1 appears to be sufficient to keep the GPU fully engaged for model training. However, we continue to experience CPU bottlenecks for the newer augmentation methods: RandAugment and AutoAugment still demand a minimum of ⌈27.8⌉ and ⌈22.6⌉ CPUs, respectively.

We find that insufficient CPU cores per GPU are also commonly observed among public cloud users. Public cloud vendors provide a wide variety of instances with relatively low CPU-to-GPU ratios [39], as shown in Figure 2.

C-2. Stragglers from variation in augmentation time. Under single-GPU training, the GPU time spent to consume a single mini-batch is fairly similar among different iterations as the gradient computation is highly periodic [19, 33]. However, in multi-GPU data-parallel training, the computed gradients may be aggregated across GPUs before moving to the next mini-batch. For fast data-parallel training, it is thus necessary to have *no stragglers* during the gradient aggregation.

We find that the data prep significantly contributes to producing such stragglers because different transformation operators cause the mini-batch processing time to vary widely. Based on our analysis, for the image size of 128 KB, the `SolarizeAdd` operator in RandAugment takes 52× longer than `CutoutAbs`, the cheapest operator. That is, the mini-batch that applies `SolarizeAdd` to its samples more frequently will take a relatively longer time to complete. We also observe that a larger input sample prolongs its augmentation remarkably. Consequently, the data prep time highly varies over iterations, as shown in Figure 3, requiring slow data preps to be accelerated instantly to finish the current iteration faster.

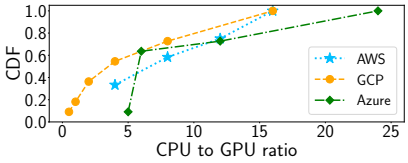


Figure 2: CDF of CPU-to-GPU ratio for GPU instances in Amazon AWS, Google GCP, and Microsoft Azure.

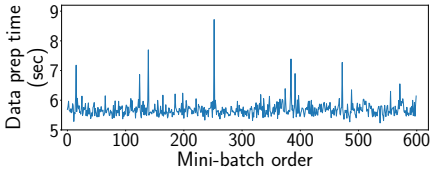


Figure 3: Data prep time varying over mini-batches when applying RandAugment.

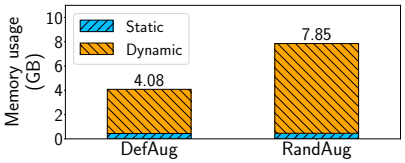


Figure 4: Breakdown of DALI memory footprint for a mini-batch size of 240 with DefAugment and RandAugment.

3.2 Data Prep on GPU

HW-accelerated libraries [45, 47] provide an option to leverage GPUs for speeding up data preps. Nevertheless, there are several challenges that currently impede their adoption for dynamic GPU offloading:

G-1. Needs for frequent generation of augmentation pipelines. HW-accelerated libraries for conventional data augmentation algorithms are compelling as we only need a single static data transformation pipeline to apply over the whole training process. On the contrary, they are usually not well suited for recent algorithms as the pipeline must frequently change to augment training samples more dynamically. A naïve solution to meeting this requirement is to generate the task for the data transformation pipeline *on the fly*. Specifically, whenever we have data to augment with a new pipeline, the system constructs a new GPU task containing the pipeline. Then, it allocates GPU memory and schedules the task to the GPU. The runtime cost associated with this procedure can be substantial and affect the entire data prep step. For example, on our 3090 Ti server, preparing a new GPU task with four samples for RandAugment takes up around 30% of the total time required to complete the task.

G-2. Interference in limited GPU memory. Offloading data prep to the GPU provides higher throughput but consumes GPU memory. Many prior studies already point out substantial GPU memory usage (up to several GB) caused by the GPU-accelerated library DALI [14, 40, 42]. Since GPU memory is scarce, this usage could leave insufficient memory available for model training. As a result, users may need to train the model either with gradient accumulation, which can cause significant slowdown due to multiple forward-backward computations performed for each iteration, or with smaller mini-batches, which can affect training efficiency.

To understand what constitutes memory usage when data prep is carried out by the GPU, we run DefAugment and RandAugment on DALI for a mini-batch with 240 samples and break down their memory footprints. Figure 4 shows the results. DALI inevitably requires a certain amount of memory (+400 MB) as a workspace for pipeline computations. This static memory usage represents a small portion (0.8–1.6%) of the total capacity of modern GPUs (24–48 GB). Importantly, this static memory usage comfortably fits within the memory space typically left available during DL training [11]. In contrast, DALI also incurs non-static, dynamic memory consumption, which takes up more than 3.6 GB and 7.3 GB of memory for DefAugment and RandAugment, respectively. This dynamic memory primarily serves as a repository for the intermediate data produced during the execution of sample transformation pipelines. Notably, RandAugment incurs higher dynamic memory usage than DefAugment owing to the invocation of numerous distinct transformation operators, each with diverse memory demands.

4 PROPOSED SOLUTIONS

We propose exploiting idle GPU cycles to offload a portion of the CPU-side data prep computations onto local GPUs, while constantly utilizing available CPUs to maximize performance gains. In this section, we present solutions to the challenges discussed in the previous section and conduct a theoretical analysis of the data diversity achieved by our proposed approach.

4.1 New System Components for GPU Prep

Addressing challenges G-1 and G-2 posed in § 3.2 is a prerequisite for optimizing system runtime for the data prep workload. To that end, we present two extensions to DL frameworks as follows:

Solution 1	Ahead-of-time task allocation	G-1
Solution 2	Cross-boundary memory sharing	G-2

4.1.1 ***Ahead-of-time task allocation.*** We propose an efficient method for executing GPU tasks through *ahead-of-time task allocation*. This method involves creating a combination of GPU tasks for *all* possible data augmentation pipelines in advance and reusing them in future iterations. Instead of constructing a task on the fly, we select one of the pre-built GPU tasks that corresponds to a random augmentation pipeline and submit it along with the input samples directly to the GPU, reducing the overhead of constructing tasks at runtime. Our method is based on the observation that when a GPU task is selected multiple times, it will run the same augmentation operators and often allocate the same memory variables. Thus, we can allocate memory from GPU once and reuse it for every selection of the task without having to reallocate memory. To recycle a task, we simply need to overwrite the input samples for the data augmentation in the task.

The number of pre-built GPU tasks depends on algorithm-specific factors like the size of the random operator set, the number of operators to be randomly selected in the pipeline, and whether certain ineffective combinations are prohibited. While the potential number of augmentation combinations could theoretically grow exponentially, they do not necessarily result in exponentially larger memory usage. This is because each GPU task allocates memory on-demand from a

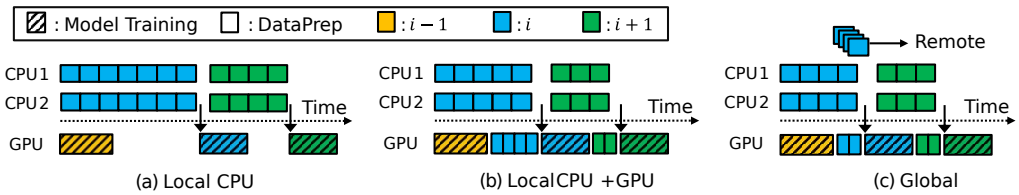


Figure 5: Towards efficient intra-batch parallelization. Training iterations get faster while moving steps from (a) to (c).

memory pool as it is scheduled and proceeds with data augmentation for input samples fetched from storage or CPU memory. Once the augmentation is completed, this memory is promptly returned to the pool. Thus, even with a vast number of possible augmentations, as long as they are not all executed simultaneously, they represent a set of configurations that comfortably fit within the GPU memory. For instance, RandAugment generates up to 196 tasks but requires no more than 7.85 GB of GPU memory, as shown in Figure 4.

4.1.2 ***Cross-boundary memory sharing.*** Both model training frameworks (like PyTorch) and data prep libraries (like DALI) maintain their local pool of GPU memory. If free memory exists in one local pool, it can be released and utilized by the other pool for remote memory allocations. To facilitate this *cross-boundary memory sharing*, we have designed a system that reclaims free memory from model training for use by data prep and vice versa. A detailed illustration of this memory sharing mechanism is presented in § 5.3.

Similar to data prep in Figure 4, model training often consumes most of its in-use memory to store "intermediate outputs" from model layers, such as feature maps in CNN/RNN models [33, 68]. This memory usage pattern indicates that when model training is finished and inactive, all of its dynamic memory is freed, thus providing ample space for data prep to use more memory. Overall, this approach allows efficient sharing of GPU memory between model training and data prep steps, as long as their memory allocation and deallocation phases are carefully coordinated.

4.2 CPU-GPU Cooperation for Data Prep

When offloading data prep tasks to GPU, *spatial GPU sharing* runs data prep and model training on the GPU simultaneously without temporal coordination. This strategy frequently leads to inefficient memory utilization since both steps may expand their working sets concurrently. It can cause the peak memory usage from data prep and model training to add up and incur memory overflow, which is known to have a detrimental impact on DL training performance [33]. To avoid memory overflow in spatial GPU sharing, it is thus essential not to exhaust the GPU's memory capacity while assuming no memory sharing between data prep and model training at all. On the computation front, data prep libraries support various features like prefetching and pipelining to deliver higher processing rates at the expense of GPU cycles. This can similarly lead to unexpected contention for GPU processing units, i.e., Streaming Multiprocessors.

As a result, spatial GPU sharing is beneficial *only* when model training underutilizes *both* memory and processing units. If either memory or compute resources is heavily utilized by model training, a more pragmatic approach would be to employ *temporal GPU sharing*, as shown in Figure 5. In practice, as most DL training is notably memory and/or compute-intensive, spatial GPU sharing seems less frequently applicable.

FusionFlow has the capability to switch between temporal and spatial GPU sharing modes by profiling resource usage conditions and performance metrics derived from test runs (§ 5.2.1). Since the spatial sharing mode simply applies all proposed techniques without GPU time-multiplexing, our focus here is on explaining the design of the temporal sharing scheme within FusionFlow.

4.2.1 ***How to split a single data prep.*** GPU time-multiplexing dedicates the entire GPU to either data prep or model training at any given moment, allowing both steps to alternate and make full use of available memory and compute resources. To maximize its effectiveness, the techniques introduced in § 4.1 are essential. For data prep of a single mini-batch, which is to be processed in parallel by both CPU and GPU resources, we use intra-batch parallelism (Intra-P). Our implementation of Intra-P partitions a single mini-batch into many smaller *tiny-batches*. Each tiny-batch represents a small *data prep task* that contains a small, disjoint subset of samples (even just one sample) to decode and augment from the original mini-batch.

While the concept behind our Intra-P is straightforward, achieving workload-balanced scheduling for data prep tasks across the "slower" CPU and "faster" GPU is particularly difficult due to workload variabilities, as illustrated in Figure 3, as well as variabilities in available compute resources, such as limited CPUs due to sudden system contentions. For this reason, we have chosen to adopt data-level partitioning rather than operator-level partitioning that splits data prep operations and offloads computationally expensive ones. With the data partitioning scheme, we can perform fast reactive scheduling of data prep tasks. This approach enables precise load-balancing decisions that can quickly react to changing workload and resource conditions. Furthermore, the small task granularity of tiny-batches empowers us to make fine-grained adjustments to our scheduling decisions. In contrast, operator-level partitioning is coarse-grained and often limited to offloading either the decode phase or both the decode and augment phases in popular DL platforms [66].

4.2.2 ***How to exploit CPU/GPU jointly.*** In the temporal GPU sharing mode, we take advantage of Intra-P in two stages. To illustrate, we use the scenario described in Figure 5 that runs training on two local GPUs, each assigned with two CPU cores. For brevity, we only show one set of GPU and CPUs in the figure. Each Intra-P stage addresses C-1 and C-2 posed in § 3.1 as follows:

Stage 1	Intra-P using local GPU & local CPU cores	C-1
Stage 2	Intra-P using local GPU & all CPU cores	C-2

Local CPU → Local CPU+GPU. Intra-P consumes local mini-batches one after the other based on their static order (i.e., mini-batch i then $i + 1$). Using only the local CPU is insufficient if it leaves the GPU unused for a long time after the previous iteration $i - 1$ is completed, as shown in Figure 5(a). We exploit these ample idle resources to consume tiny-batches of iteration i while not affecting

the preceding model training of iteration $i-1$, as shown in Figure 5(b). There are two crucial advantages of harnessing idle GPU cycles in this manner. First, as GPU offers higher FLOPS than CPU, tiny-batches are consumed faster, remarkably reducing the mini-batch completion time. Second, CPUs and GPUs are utilized more effectively, enhancing system efficiency.

Local CPU+GPU → Global. The former stage is local-centric and neglects global coordination in multi-GPU data-parallel training. Specifically, if two GPUs start model training for iteration i at the same time, no coordination would be needed as they are likely to enter gradient aggregation roughly simultaneously. Otherwise, whichever starts earlier should unnecessarily wait until the other GPU gets ready for gradient aggregation. Therefore, to synchronize the gradient aggregation across GPUs to the utmost extent every iteration, we schedule tiny-batches to available remote CPUs when local GPU/CPUs have quite a few of tiny-batches remaining to process, as shown in Figure 5(c). Note that we do not exercise the remote GPU as it is in use for model training.

In the spatial GPU sharing mode, FusionFlow offloads tiny-batches to the local GPU even when it is currently being used by model training. This is the key distinction to the temporal GPU sharing mode in FusionFlow.

4.3 Data Diversification

In this subsection, we conduct a statistical analysis to examine the diversity distribution resulting from two different approaches. The first one, referred to as the *standard approach*, serves as a baseline where a random augmentation is applied to each sample in each epoch independently. The second one, referred to as *tiny-batch approach*, is specifically designed for our new system where a random augmentation is applied to each tiny-batch in each epoch independently. The composition of samples within each tiny-batch is randomly determined and independent across all epochs. The diversity resulting from augmentation is measured by the number of unique augmented samples obtained from all samples across all epochs. Due to space constraints, we provide a condensed overview of the formulas here, with detailed proofs available in [27].

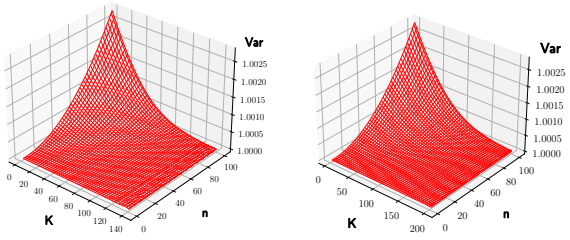
4.3.1 ***Expectation of sample diversity.*** Following the same assumptions from [30], we assume K epochs, N samples, augmentation set A, batch size n. We define X_{it} as the indicator of whether augmentation A_t is applied to the i-th sample.

$$X_{it} = \begin{cases} 1, & \text{If augmentation } A_t \text{ is applied to the } i\text{-th sample} \\ & \text{at least once in all epochs,} \\ 0, & \text{Otherwise.} \end{cases}$$

Similarly, we define X_i as the number of unique augmented samples from the i-th sample in all epochs and X as the number of unique augmented samples from all samples in all epochs.

In the standard approach, the expectation of X_i is as follows:

$$E(X_i) = \sum_t E(X_{it}) = \sum_t (1 - P(X_{it} = 0)) = |A| \left(1 - \left(\frac{|A|-1}{|A|}\right)^K\right)$$



(a) RandAugment **(b) AutoAugment**

Figure 6: Ratio of the standard deviation.

where $|A|$ is the cardinality of the augmentation set. Then, the expectation of diversity, $E(X)$ is the sum of expectations of all X_i.

$$E(X) = \sum_i E(X_i) = N|A| \left(1 - \left(\frac{|A|-1}{|A|}\right)^K\right)$$

The tiny-batch approach also selects an augmentation for each sample independently for every epoch. Therefore, the batching of samples does not affect the expectation values $E(X_{it})$, and by extension, $E(X_i)$ and $E(X)$. Consequently, the expectation of diversity with the tiny-batch approach, measured as the number of unique augmented samples, is equivalent to that achieved with the standard approach.

4.3.2 ***Variance of sample diversity.*** The variance of sample diversity, $Var(X)$, can be expanded using $Var(X) = E(X^2) - (E(X))^2$ as follows:

$$E(X^2) = E(\sum_i X_i)^2 = E(\sum_i X_i^2 + \sum_{i \neq j} X_i X_j) = \sum_i E(X_i^2) + \sum_{i \neq j} E(X_i X_j)$$

$$(E(X))^2 = (\sum_i E(X_i))^2 = \sum_i (E(X_i))^2 + \sum_{i \neq j} E(X_i)E(X_j)$$

$$Var(X) = \sum_i (E(X_i^2) - (E(X_i))^2) + \sum_{i \neq j} (E(X_i X_j) - E(X_i)E(X_j))$$

Since both the standard and tiny-batch approaches apply a random augmentation to each sample independently for each epoch, the expectation of X_i and X_i^2 remains identical in both methods. Therefore, the difference in the variance between the two approaches is determined by the last term. In the standard approach, there is no dependency on the samples. Thus, $E(X_i X_j) = E(X_i)E(X_j)$ holds true, and the last term of the variance is zero. However, in the tiny-batch approach, samples that belong to the same tiny-batch in an epoch receive the same augmentation. Consequently, depending on the probability that the i-th and j-th samples belong to the same tiny-batch, $E(X_i X_j)$ can be slightly larger than $E(X_i)E(X_j)$.

In order to visualize the variance of diversity in our approach *vs* per-sample augmentation, Figure 6 presents the ratio of the standard deviation of diversity in the tiny-batch approach over the standard deviation of diversity in the standard per-sample augmentation, given different parameters for RandAugment ($K = 2 \sim 140$, $N = 1743042$, $|A| = 16$, $n = 1 \sim 100$) and AutoAugment ($K = 2 \sim 200$, $N = 1743042$, $|A| = 25$, $n = 1 \sim 100$), respectively, where standard deviation is the square root of variance.

Based on the plots, the variance of diversity in our approach is almost the same as that of per-sample augmentation in typical DL training tasks where tiny-batch size $n \ll$ sample size N. As a result,

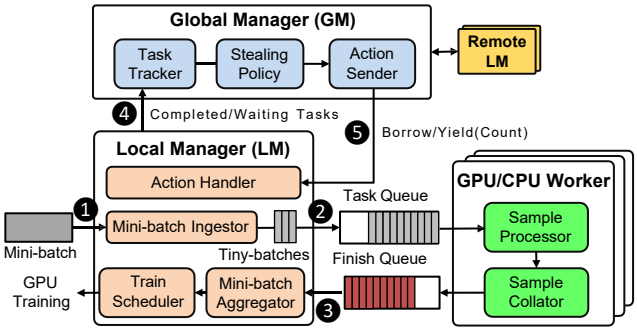


Figure 7: System architecture of FusionFlow.

our approach maintains the same level of diversity as per-sample augmentation, thus guaranteeing the same level of convergence in model training. This analysis confirms that our approach does not inadvertently increase the number of epochs for model convergence and can benefit from faster training time compared to standard approaches.

5 FUSIONFLOW DESIGN

As a system built on PyTorch, FusionFlow supports both single- and multi-GPU DL training and runs transparently to users.

5.1 System Architecture

FusionFlow has three key system components, as shown in Figure 7: *local manager* (LM), *global manager* (GM), and *worker*.

Local manager. As a GPU-local runtime, LM handles data prep on behalf of the DL training on each GPU, as shown in Figure 7. Specifically, for each incoming mini-batch (❶), LM transforms it into a set of tasks representing tiny-batches called *TB tasks* and performs task scheduling to workers (❷), and aggregates the processed tasks as an augmented mini-batch to initiate model training (❸).

FusionFlow offers two types of workers to consume TB tasks: CPU workers and GPU workers. The LM has an action handler that coordinates with the GM to scale up or down the number of CPU workers. Furthermore, the LM activates GPU workers only when the local GPU has sufficient idle resources to accommodate offloaded TB tasks, faithfully facilitating both spatial and temporal GPU sharing modes between data prep and model training. When a new DL training job is issued, the LM determines which mode to enter by initiating a profiling phase and collecting basic job information, as will be discussed in § 5.2.

Global manager. As a system-wide runtime, GM tracks the status of LMs and coordinates their progress. The primary mission of the GM is mitigating the straggler's effect in multi-GPU training through a dynamic resource relocation. At every iteration, the GM first starts by assigning each LM an equal number of CPU workers for fairness. Then, when certain LMs make slower progress during the data prep, the GM grants them more CPU workers to process their pending tasks faster. This worker scaling is done reactively as soon as stragglers manifest to avoid the risk of misprediction.

To make scaling decisions quickly, the task tracker in GM collects the number of unprocessed TB tasks from all LMs every short interval (❹) and informs each LM of the appropriate action to take (❺). For the worker scaling actions, `Borrow` *vs* `Yield`, the GM also tells the associated LMs how many workers to borrow or yield. Upon receiving `Borrow`, the LM increases the number of CPU workers by the amount specified in `Count`. At the same time, to avoid CPU contention, the GM asks other LMs to yield a total of `Count` workers. When all TB tasks for the current data prep are consumed, the borrowed workers are returned to their original LMs.

Worker. Each CPU worker is assigned to a separate CPU core to execute its assigned task without interference. This CPU worker can also be mapped to a CPU core in a remote machine. Data prep is compute-intensive, so sharing a single core among multiple CPU workers would not provide any performance advantages. In contrast, a GPU worker executes its assigned task by offloading it to the local GPU hardware. It schedules kernel computations for a randomly selected data augmentation on the GPU with minimal CPU overhead due to ahead-of-time task allocation. For this reason, GPU workers do not have dedicated compute resources and instead share a small number of CPU cores with other runtime managers in FusionFlow.

It is worth noting that FusionFlow simplifies resource management by treating CPUs distributed across different machines or NUMA domains uniformly. In particular, when integrating FusionFlow with horizontal CPU scaling, we assume that the machines possess ample I/O and network bandwidth capacities, with data prep on the CPU being the most time-consuming task. Therefore, in response to dynamic changes in resources, it suffices to notify the GM of the updated compute capacity.

5.2 Efficient Task Scheduling and Execution

Now, we explain how FusionFlow decides between spatial and temporal sharing modes and proceeds with task scheduling. In order to make an appropriate choice for the sharing mode, we introduce an online profiling phase where FusionFlow performs a test run to evaluate a new DL job for a few training iterations.

5.2.1 ***Profiling phase.*** When a new DL training job is initiated, FusionFlow enters the profiling phase. During this phase, FusionFlow first evaluates the memory condition by independently running data prep and model training and measuring their GPU memory consumption. It assumes no memory oversubscription will occur if the sum of peak memory usage fits within the GPU memory capacity. If this condition is not met, FusionFlow promptly transitions to the execution phase that employs temporal GPU sharing. Next, FusionFlow assesses the compute condition. Determining which specific mode will lead to less slowdown is not straightforward when relying solely on performance metrics from isolated runs. However, since DL training is fairly regular across iterations, FusionFlow can empirically compare the throughput of both modes over a few iterations. FusionFlow transitions to the execution phase using the mode that demonstrates higher speed.

5.2.2 ***Execution phase.*** During the execution phase, FusionFlow schedules TB tasks onto the CPU and GPU. Each TB task contains essential information about data prep at a tiny-batch level, including input samples and transformation code such as decoder and augmentation pipeline. Initially, a TB task is in `Waiting` state before being scheduled, then transitions to `Active` state when assigned to either a CPU or GPU worker, and finally moves to `Completed` state once the task execution is finished.

Task affinity-aware scheduling. Based on C-2 in § 3.1, a TB task with larger samples is more time-consuming and thus runs more effectively on GPU because it excels in processing large data blocks in parallel. To preferably assign larger tasks to the GPU, our task affinity-aware scheduler called *TAAS* sorts TB tasks for a mini-batch based on the aggregate size of samples in each task and feeds them into the CPU and the GPU in ascending and descending orders, respectively. But, when a few tasks remain, it is possible that assigning all these tasks to the GPU is faster due to CPUs being usually much slower than GPUs. TAAS in each LM automatically searches for this threshold value that reflects the number of last-minute tasks to execute only on the GPU through an iterative process. Specifically, TAAS starts from zero, so all tasks for a mini-batch are up for grabs by any compute resource. At the end of every mini-batch, it adjusts the threshold based on examining which resource is busier between CPU and GPU at the last moment. If it is CPU, TAAS increments the threshold to give GPU more last-minute loads in the next mini-batch or vice versa. TAAS stops searching when the threshold oscillates.

We currently operate FusionFlow for offline learning that does not expect data distribution to change drastically over time. So, essentially, the threshold is decided once and used throughout the training process. However, in situations where hardware resources undergo dynamic changes, FusionFlow resumes the iterative search process to fine-tune the threshold value as needed.

Reactive CPU worker scaling. FusionFlow triggers worker scaling when an LM finishes all of its TB tasks for the current iteration and can relinquish its CPU workers for other LMs still in progress. This reactive approach is fast and can minimize the risk of misprediction, even given the dynamic nature of augmentation algorithms and resource conditions.

However, this worker scaling should effectively distribute available CPU workers among different LMs. At the moment of releasing CPU workers, the other slower LMs may have different compute resource demands, as reflected in the progress of their tiny-batch processing. In FusionFlow, we distribute available workers to the LMs based on the number of tasks in `Waiting` state, as this represents the amount of pending work in each LM. We also take into account the progress of `Active` tasks, which may have significantly different amounts of work done in terms of the number of tiny-batch samples processed. If a slow LM has an `Active` task with substantial progress (i.e., a majority of samples processed), its worker will soon be available to fetch a `Waiting` task. In this case, the LM decreases the number of `Waiting` tasks by one to account for the worker's immanent availability, reducing the likelihood of unnecessary worker stealing. In FusionFlow, each LM adjusts its resource demand by assessing whether the processing of the `Active` tiny-batch exceeds 50%. This pivot point was chosen to be neither too aggressive nor too conservative.

5.3 Memory Overflow Handling

Most DL frameworks and data prep libraries have built-in memory allocators. For example, PyTorch uses the caching memory allocator [50] that serves large (≥ 1 MB) and small (< 1 MB) allocations from different bins, while DALI uses a simple allocator that serves requests from free blocks stored in a single bin. TensorFlow has the BFC allocator [61] that organizes free blocks into multiple bins with sizes that increase by a power of two. While implementation details may differ, these frameworks all share a commonality: they organize memory space into blocks and manage them locally. Therefore, to enable memory sharing, we grant the remote memory allocator access to local memory at the block level.

APIs. Low-level GPU memory management primitives such as `cudaMalloc` and `cudaFree` can be used to enable memory sharing between frameworks operating their memory allocators in different OS processes. Unfortunately, this approach incurs a detrimental impact on DL throughput when used in performance-sensitive loops like training iterations [33]. To enable efficient, fine-grained GPU memory sharing without sacrificing performance, we have developed two generic memory-granting APIs: `YieldBlk` and `BorrowBlk`. These APIs transfer and receive the access privilege of each memory block using `cudaIpcGetMemHandle`, which allows memory handles to be shared across processes with low overhead. Any process can invoke `YieldBlk` with the address of a local memory block to grant access, and any process needing a free block from a remote process can invoke `BorrowBlk` by specifying the desired block size in the API.

Strategy. In FusionFlow, memory is granted *unidirectionally* because model training typically consumes more memory than data prep. For example, the memory demand for data prep is usually at most 10 GB, even with ahead-of-time task allocation. So, at the end of the model training phase, FusionFlow can export free memory blocks from model training's DL framework via `YieldBlk` for use in data prep's GPU library during the GPU offloading phase. The large free blocks are exported first, followed by small free blocks, until the desired amount (e.g., 10 GB) has been reached. Data prep and model training can communicate through IPC on shared memory, which includes a record of all memory blocks exposed by the DL framework. Note that this cross-boundary memory sharing is used as a last resort only for temporal GPU sharing in FusionFlow.

6 EVALUATION

In this section, we assess the efficacy of FusionFlow, both when used as a standalone solution and in combination with other complementary strategies. We also validate its applicability, compatibility, and key design features.

6.1 Implementation and Methodology

FusionFlow is built atop PyTorch 1.8.0 and DALI 1.17 with ~3500 new LoC for GM, LM, and other system components in PyTorch and DALI. FusionFlow extends PyTorch's data loading utility (`torch.utils.data`) [5] and allows users to import its functionalities by configuring a few parameters such as augmentation algorithm and tiny-batch size. FusionFlow also has custom data structures (e.g., progress list and action list) implemented using a POSIX IPC-based shared memory library (`posix.ipc`) [3] for communication protocols between the system's runtime managers.

Competing methods. We first compare FusionFlow with four methods that operate independently of external resources or memory caching. (1) *CPU-InterP* represents the default PyTorch method that utilizes inter-batch parallelism on CPUs. (2) *CPU-IntraP* employs

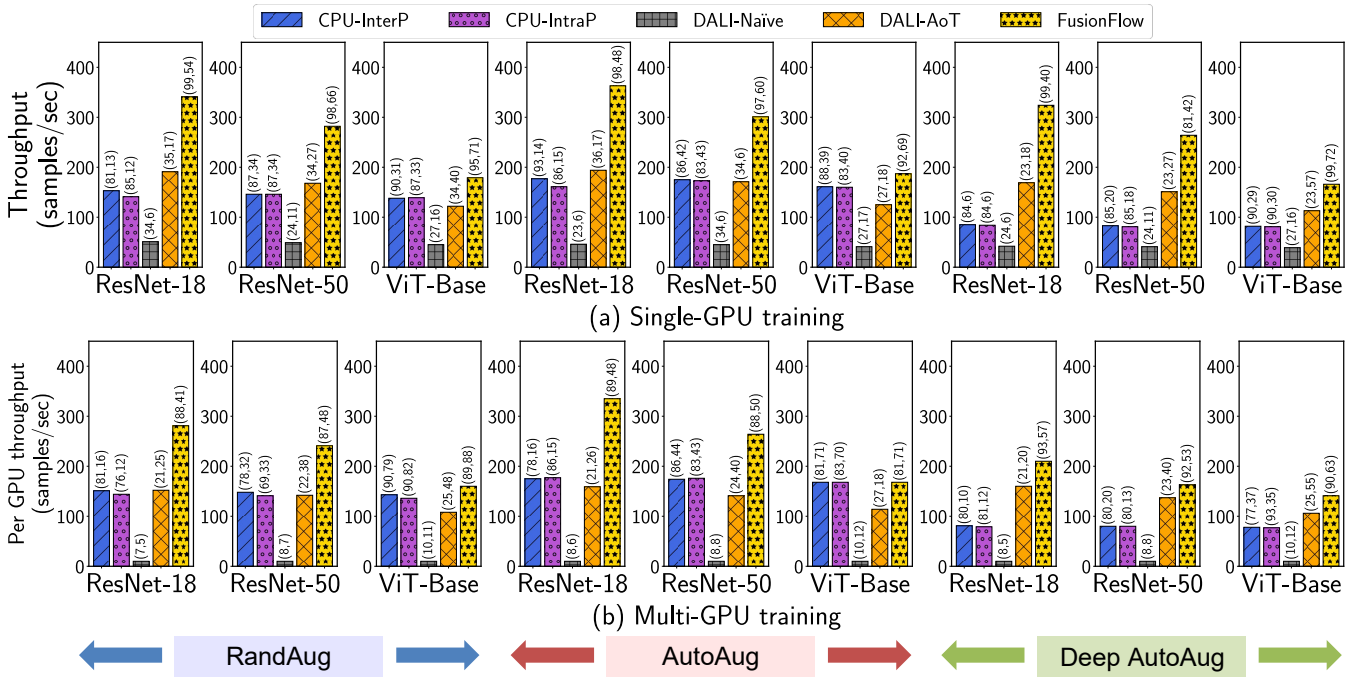


Figure 8: Throughput in training ResNet-18, ResNet-50, and ViT. The first three, middle three, and last three graphs show the results under RandAugment, AutoAugment, and Deep AutoAugment, respectively. Each bar shows CPU and GPU utilization on the top.

intra-batch parallelism on CPUs. (3) *DALI-Naïve* is a GPU-based method that creates and executes GPU tasks on demand, without GPU time-multiplexing. (4) *DALI-AoT* improves DALI-Naïve with ahead-of-time task allocation.

In addition, we compare FusionFlow with the following approaches to show how FusionFlow creates synergy with other complementary solutions. (5) *FastFlow* [66] is a state-of-the-art method that focuses on horizontal CPU scaling. (6) *Revamper* [30] is a caching system that stores partially augmented samples in memory and reuses them in successive iterations.

FastFlow was originally implemented in TensorFlow but has been ported to PyTorch for a fair comparison. Unless otherwise specified, we use a default tiny-batch size of four samples.

Benchmarks. We select three training benchmarks for image classification, ResNet-18 [20], ResNet-50 [20], and ViT-Base [17], on three modern data augmentation algorithms, RandAugment [13], AutoAugment [12], and Deep AutoAugment (DeepAA) [74]. All models use the stochastic gradient descent optimizer and automatic mixed precision [1] for training. For training throughput experiments, we use the OpenImage-Extended [29] dataset or OpenImage for short. For model accuracy experiments, we additionally use the ImageNet-1k [57] dataset which is popular for convergence analysis.

Hardware platform. The default evaluation setup (3090-6G-32C) consists of six NVIDIA RTX 3090 Ti GPUs, each with 24 GB GPU memory, dual Intel(R) Xeon(R) Gold 6226R 32 CPU cores running at 2.9 GHz, and 384 GB host memory. We allow four CPU workers per GPU and reserve the remaining CPU cores for runtime managers and GPU workers. This setup is used for both single-GPU and data-parallel training on a single node, ensuring a uniform CPU worker to GPU ratio. For scenarios that can accommodate more intra-server CPUs, we use a standard A100 server (A100-8G-128C) equipped with eight NVIDIA A100 GPUs and 128 CPU cores. To facilitate horizontal CPU scaling, we augment our resources by incorporating an additional 8-CPU server connected via a 1Gbps network.

6.2 Performance Comparison

6.2.1 ***As a standalone solution.*** **Single-GPU training.** Figure 8(a) shows the throughput of the four competing methods (1)–(4) and FusionFlow when conducting single-GPU training on various DL models and data augmentation algorithms. To ensure a fair comparison, the input mini-batch sizes for each model were chosen to fit within the GPU memory using DALI-Naïve. The results demonstrate that FusionFlow consistently outperforms both CPU-based and GPU-based methods in all cases. Specifically, we see the largest performance gain for ResNet-18, achieving 105–281% and 78.5–91.7% higher throughput compared to CPU-InterP and DALI-AoT, respectively. This is mainly due to ResNet-18 performing lightweight training that leaves ample idle GPU cycles relative to other models. However, for the most computationally demanding benchmark, ViT-Base, FusionFlow's performance gain over CPU-InterP (best baseline) is reduced to 29.7%, 16.1%, and 102% for RandAugment, AutoAugment, and DeepAA, respectively, which is expected.

We do not see significant performance merits with DALI-Naïve and DALI-AoT as they only attempt to utilize GPUs, resulting in adverse effects on performance when the GPU is heavily utilized and leaves little room for accommodating data prep tasks, as seen in ViT-Base. While there is potential to improve the performance of these methods by utilizing both GPUs and CPUs, the lack of GPU multiplexing may limit the batch sizes that can be supported, making FusionFlow a more attractive choice (Figure 9).

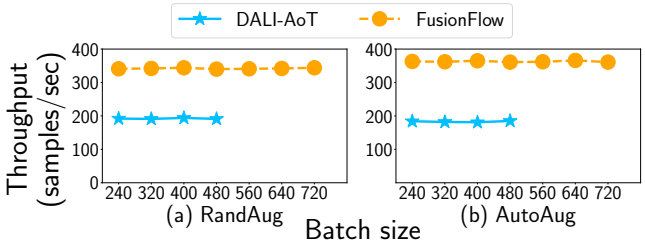


Figure 9: ResNet-18 throughput over different mini-batch sizes.

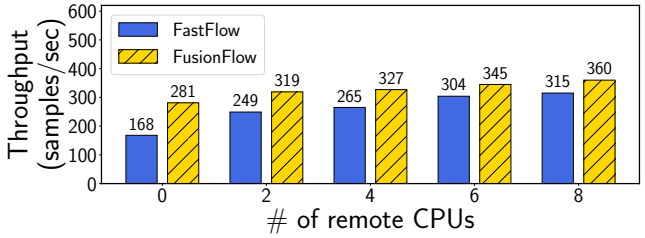


Figure 10: Throughput comparison between FastFlow [66] and FusionFlow using ResNet-50.

Multi-GPU training. In Figure 8(b), we present a comparison of system throughput in a six-GPU setting. In comparison to CPU-InterP and DALI-AoT, similar to what we observe in the single-GPU case, FusionFlow achieves higher throughput across all models for the three data augmentation algorithms. In particular, FusionFlow achieves an average throughput increase of 112.9% for ResNet-18, 73.4% for ResNet-50, and 28.3% for ViT-Base when compared to CPU-InterP. Similar to Figure 8(a), FusionFlow favors ResNet-18 over ResNet-50 and ViT-Base because ample GPU cycles are available by running a model with low computational demand. Likewise, both DALI-Naïve and DALI-AoT do not excel in performance.

Varying mini-batch sizes. FusionFlow's ability to share memory allows for larger mini-batch sizes during model training, which in turn can improve training efficiency. We demonstrate this by comparing FusionFlow's performance to a competing GPU-based method, DALI-AoT, over a range of mini-batch sizes using ResNet-18 and two augmentation algorithms, RandAugment and AutoAugment. As shown in Figure 9, FusionFlow performs better and retains its performance at mini-batch sizes up to 720 samples, while DALI-AoT could not handle mini-batch sizes larger than 480 samples.

6.2.2 ***As a complementary solution.*** FusionFlow not only offers improved training performance as a replacement for FastFlow or Revamper but also presents an opportunity for synergy when used along with these approaches. We will now explore these aspects.

With horizontal CPU scaling. In Figure 10, we compare the throughput of ResNet-50 between FastFlow and FusionFlow under varying remote CPU availabilities. This evaluation reveals two observations. First, FastFlow demands 4 to 6 remote CPUs per GPU to achieve a throughput of 281 samples per second, whereas FusionFlow delivers this level of throughput without the necessity of any remote CPUs. Given that our 3090-6G-32C server assigns 4 CPUs to each GPU, this represents approximately >50% in CPU resource savings compared to FastFlow. Second, FusionFlow consistently enhances throughput across a wide spectrum of available remote CPUs, showing remarkable synergy when both approaches are combined.

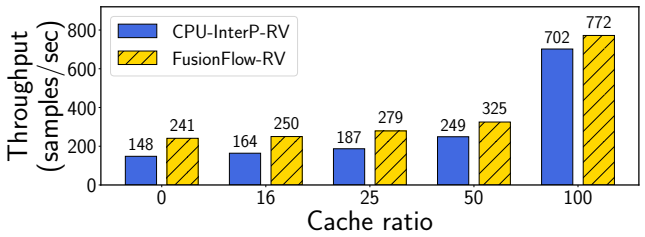


Figure 11: Throughput of Revamper [30] (in-memory cache) with and without FusionFlow using ResNet-50.

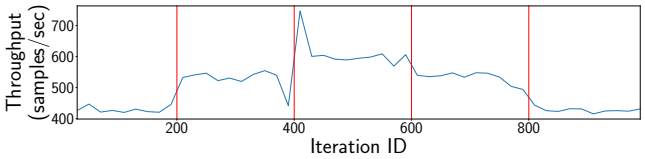


Figure 12: ResNet-18 throughput while dynamically changing local and remote CPU count.

Notably, FusionFlow outperforms FastFlow at a large margin when the same number of remote CPUs is available.

With in-memory caching. In Figure 11, we compare the throughput of ResNet-50 between CPU-InterP on Revamper (*CPU-InterP-RV*) and FusionFlow on Revamper (*FusionFlow-RV*) for a wide range of memory sizes used for caching. As Revamper stores tensorized samples after decoding, it requires a large amount of memory for full caching, approximately 2.58 TB for OpenImage. Our 3090-6G-32C machine has 384 GB of host memory, allowing it to store roughly 16% of the entire dataset. In this scenario, FusionFlow-RV outperforms CPU-InterP-RV by 52.4%. To evaluate larger cache sizes, we also emulate the sample access pattern of a specified cache size while reusing a dummy partially augmented sample for cache hits. As Figure 11 shows, FusionFlow-RV consistently improves throughput compared to CPU-InterP-RV across all memory sizes, even at the maximum capacity of 2.58 TB.

6.3 Diverse Usage Scenarios

We evaluate FusionFlow's performance across diverse usage scenarios using **RandAugment on a single GPU** for 3090-6G-32C, unless otherwise specified.

Dynamic CPU resources. The CPU compute capacity may undergo dynamic changes for both local and remote CPUs. In such cases, FusionFlow should promptly adjust its CPU pool and schedule CPU workers to maintain good system efficiency during ongoing training on the GPU. In Figure 12, we illustrate how FusionFlow responds to variations in CPU availability over time using ResNet-18. Specifically, we increase the number of available CPUs from 4 to 8 using "local" CPUs during iterations 200 to 400 and then to 12 using "remote" CPUs during iterations 400 to 600, and subsequently, we restore it to 8 and then to 4. As shown in the figure, FusionFlow can adapt to these CPU capacity changes in a non-blocking manner. This stands in contrast to existing approaches in PyTorch and TensorFlow, where PyTorch requires all CPU workers to be predefined, and TensorFlow relies on specifying AUTOTUNE, a parameter limited by hardware specifications like the maximum CPU count [66]. It is also worth noting that thanks to FusionFlow's ability to dynamically

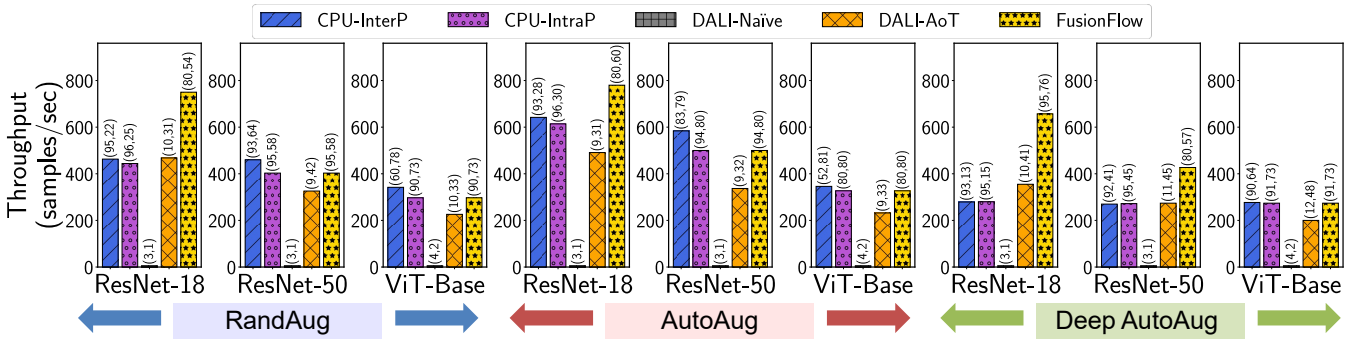


Figure 13: Throughput in training ResNet-18, ResNet-50, and ViT using different augmentation algorithms on the A100 machine.

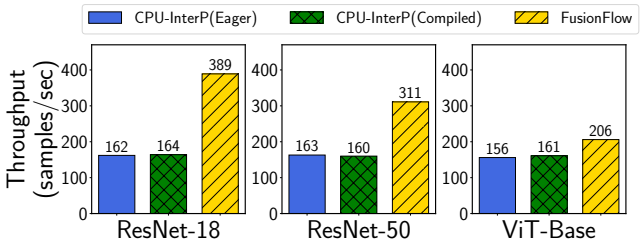


Figure 14: FusionFlow performance on newer PyTorch 2.0.

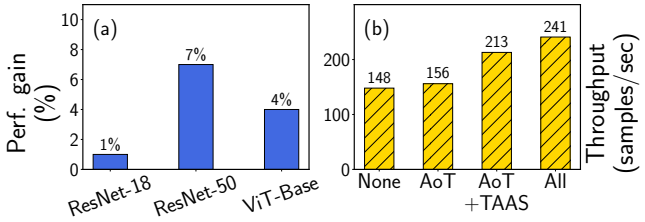


Figure 15: (a) Performance gain with worker scaling. (b) Throughput with each optimization technique in ResNet-50.

scale CPUs beyond the local server, FusionFlow can be seamlessly extended to collaborate with autoscaling schemes like Cachew [18].

Single-sample tiny-batch. Using large tiny-batches can degrade system efficiency as fewer TB tasks that are created may not sufficiently take advantage of parallelism. So, it is crucial for FusionFlow to be optimized for small tiny-batch sizes. To that end, we evaluate FusionFlow's performance using the extreme case of single-sample tiny-batches. Results show that FusionFlow still maintains good performance, with its throughput slightly lower than Figure 8(a) by 2.5% for ResNet-18, 4% for ResNet-50, and 3.4% for ViT-Base. While users may prefer this single-sample size as it does not affect data diversity, we later show that the current tiny-batch size of four samples also does not impact model accuracy.

Newer PyTorch. We test FusionFlow on PyTorch 2.0 [65], which underwent major performance improvements, to see if FusionFlow gives the same benefits with the newer PyTorch version. Figure 14 presents the results. In this experiment, we compare FusionFlow with CPU-InterP while running the PyTorch in eager mode. This mode was found to achieve performance comparable to the compiled mode on 3090-6G-32C [65]. We observe that PyTorch 2.0 delivers higher throughput than the old version used in Figure 8(a). Importantly, even with the upgraded PyTorch, FusionFlow continues to provide substantial performance improvements across various training benchmarks.

A100 server. In Figure 13, we present the main performance results obtained using our A100-8G-128C machine. This machine provides a maximum of 16 CPU cores per GPU. So, we designate 15 workers per GPU, reserving a single core to accommodate the minimal CPU resources required by PyTorch's basic system runtime. Overall, FusionFlow consistently outperforms or is on par with other methods. In particular, although the A100 GPU imposes increased mini-batch sizes, which place higher pressure on the CPU, we find that 15 workers are adequate for certain combinations of DL models and augmentation algorithms, e.g., ViT-Base with RandAugment. In these cases, FusionFlow operates without GPU offloading, thus matching the performance of CPU-based methods.

6.4 Design Validation

In all our experiments, we allocate the same CPU resources across all competing methods. Since each method places varying demands on system resources, we provide both CPU and GPU utilization for each bar in Figure 8 and Figure 13. It is crucial to note that, by alleviating CPU bottlenecks, FusionFlow improves overall GPU utilization. We now proceed with the design validation of FusionFlow. Unless otherwise stated, we apply **RandAugment** on 3090-6G-32C.

Worker scaling. Figure 15(a) illustrates the impact of worker scaling on throughput by comparing results with worker scaling turned on and off for the data-parallel training setup. In this experiment, GPU-offloading is disabled to impose enough stress on the CPU workers. The results show that worker scaling in FusionFlow improves throughput by up to 7% in ResNet-50.

GPU task optimizations. Next, we validate the effectiveness of our optimizations on GPU task execution by comparing the throughput of four different GPU-offloading implementations based on our proposed techniques using ResNet-50. (1) *None* only harvests idle local GPU cycles without key optimizations. (2) *AoT* adds ahead-of-time task allocation on top of None. (3) *AoT+TAAS* adds task affinity-aware scheduling without the threshold for last-minute GPU tasks on top of AoT. (4) *All* includes full features of FusionFlow. The results in Figure 15(b) show that throughput increases as we add optimizations one by one. Therefore, we confirm that our main features greatly contribute to FusionFlow's performance.

Data partitioning. We compare FusionFlow with a strategy called *Decode-Off* to evaluate the performance of data-level partitioning *vs*

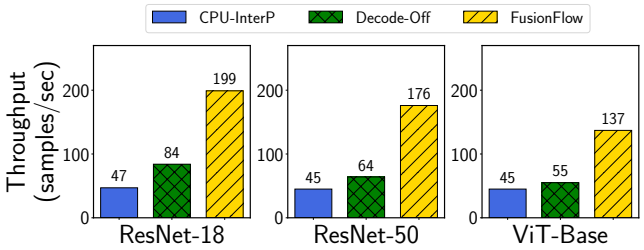


Figure 16: Throughput comparison between FusionFlow and Decode-Off that offloads the data decoding operator to the GPU.

operator-level partitioning. In Decode-Off, the data decoding phase, which is typically the most computationally expensive operation, is offloaded to the GPU, while all subsequent operations are executed on the CPU. For a fair comparison, we limit the number of CPU workers to one. We show the results in Figure 16 including a CPU-only scheme CPU-InterP for three different models.

Decode-Off runs faster than CPU-InterP due to GPU acceleration during the decoding phase. However, both schemes execute operations sequentially at the mini-batch level, meaning that later operations in data prep must wait for earlier ones to complete before starting their execution. That said, they do not utilize the GPU and CPU at the same time. On the contrary, FusionFlow is not affected by the sequential dependency of operations, continuously engaging both the GPU and CPU through more explicit tiny-batch-level parallelism. Consequently, FusionFlow achieves speedups of up to 4.23× and 2.36× compared to CPU-InterP and Decode-Off, respectively.

Convergence. Our approach that applies a random augmentation to each tiny-batch independently maintains the same level of data diversity while having minimum impact on the variance (§ 4.3). To validate this aspect further, we empirically compare the change in the top-1 accuracy over training epochs between FusionFlow and CPU-InterP (that augments each sample with a distinct pipeline) using ImageNet-1k. FusionFlow does not adversely affect the model's accuracy as Figure 17 shows, with only 0.1% degradation in the final accuracy. We also witnessed similar trends for OpenImage.

7 RELATED WORK

We have already compared FusionFlow with other approaches that rely on in-memory caching [30, 40], disaggregation [18, 73], and only GPUs [45], and we do not repeat them here.

CPU-GPU cooperation in DL training. Various DL systems attempt to offload the GPU's main workload, model training, to the CPU to take the benefit the other way around. The most well-known work in such CPU offloading is ZeRO-Offload [54]. ZeRO-Offload focuses on training models with over 13 billion parameters on a single GPU. ZeRO-Offload offloads both data (gradients and optimizer states) and compute (optimizer computation) that lead to high GPU memory savings with low communication and CPU computation costs. Before ZeRO-Offload, the primary concern of many other approaches was offloading only data to limit the working set in the GPU [24, 49, 55]. All these prior approaches mainly support training huge models [54] or medium-size models with very large mini-batch sizes [24, 49]. In contrast, FusionFlow supports typical model training cases incorporating state-of-the-art data prep algorithms, which have revealed several new performance issues.

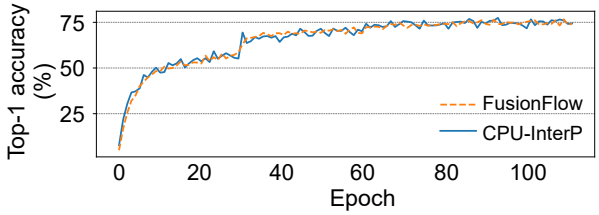


Figure 17: Top-1 accuracy over training epochs. Both methods train ResNet-50 on ImageNet-1k.

Augmentation algorithms. Data augmentation algorithms generating dynamic augmentation pipelines through random operator selections have been proposed mainly for computer vision tasks. RandAugment randomly selects a predefined number of operators and concatenates them with crop and horizontal flip to create a pipeline per sample [13, 26, 30]. Similarly, TrivialAugment [41] generates a pipeline through a random operator selection for each sample. However, unlike RandAugment, it selects a single operator and does not fix the magnitude value, which determines the operator's strength. Using magnitude gives a higher functional variability in the pipeline execution while demanding the same computational cost. AutoAugment [12] pre-builds an optimal set of pipelines to be randomly selected for each sample. It obtains such a set from running a search algorithm beforehand. The most recent algorithm, Deep AutoAugment [74], takes a fully automated approach with no hand-picked operators, where the best accuracy is achieved when the algorithm includes more than five operators. FusionFlow makes all these algorithms run faster.

Adversarial data augmentation [8, 52, 67] is a variant of data augmentation that enhances model robustness against corrupted or hostile data in computer vision [56]. It is implemented based on either GPU or CPU. The GPU-based approach generates synthetic images using a generative model during model training, effectively running two models (i.e., generative model and training model) at the same time. To optimize GPU resource utilization for this method, frameworks like Zico [33], which enable multiple models to share the same GPU, are highly suitable. On the other hand, the CPU-based approach (e.g., AugMix [21]) involves modifying training parameters or in-use data augmentations. It can thus benefit considerably from techniques like FusionFlow.

8 CONCLUSION

We propose FusionFlow that speeds up the dynamic data augmentation algorithms on CPUs and GPUs. The key idea is exploiting intra-batch parallelism, which splits an input mini-batch into multiple tiny-batches and augments the mini-batch in parallel on those compute resources. FusionFlow applies several optimizations to make GPU-offloading of tiny-batch tasks highly effective and performs CPU worker scaling to make CPU resource usage in data-parallel training more balanced. Experimental results confirm the effectiveness of FusionFlow.

ACKNOWLEDGMENTS

This work was supported by the ETRI grant [23ZS1300] and the IITP grant (No.2020-0-01336, Artificial Intelligence graduate school support (UNIST)) funded by the Korean government (MSIT).

REFERENCES

[1] Accessed in December 2023. AUTOMATIC MIXED PRECISION PACKAGE - TORCH.CUDA.AMP. https://pytorch.org/docs/stable/amp.html.

[2] Accessed in December 2023. NVIDIA DGX-2 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-2-datasheet-us-nvidia-955420-r2-web-new.pdf.

[3] Accessed in December 2023. POSIX IPC for Python - Semaphores, Shared Memory and Message Queues. http://semanchuk.com/philip/posix_ipc/.

[4] Accessed in December 2023. The NVIDIA DGX-1 Deep Learning System. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-datasheet-nvidia-us-808336-r3-web.pdf.

[5] Accessed in December 2023. TORCH.UTILS.DATA. https://pytorch.org/docs/stable/data.html.

[6] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic Optimization with Laggard Data Pipelines. In *NeurIPS*, 2020.

[7] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A Thekkath. A case for disaggregation of ml data processing. *arXiv preprint arXiv:2210.14826*, 2022.

[8] Yatong Bai, Brendon G. Anderson, Aerin Kim, and Somayeh Sojoudi. Improving the Accuracy-Robustness Trade-Off of Classifiers via Adaptive Smoothing, 2023.

[9] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[10] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

[11] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 625–638, 2022.

[12] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019.

[13] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. *Advances in Neural Information Processing Systems*, 33:18613–18624, 2020.

[14] Victor Guilherme Turrisi da Costa, Enrico Fini, Moin Nabi, Nicu Sebe, and Elisa Ricci. solo-learn: A Library of Self-supervised Methods for Visual Representation Learning. *Journal of Machine Learning Research*, 23(56):1–6, 2022.

[15] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric Xing. High-performance distributed ML at scale through parameter server consistency models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 15)*, volume 29, 2015.

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems (NIPS 12)*, 25, 2012.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[18] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[19] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Dan Hendrycks, Norman Mu, Ekin D Cubuk, Barret Zoph, Justin Gilmer, and Balaji Lakshminarayanan. Augmix: A simple data processing method to improve robustness and uncertainty. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[22] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems (NIPS 13)*, 26, 2013.

[23] Marius Hobbhahn and Tamay Besiroglu. Accessed in December 2023. Trends in GPU price-performance. https://epochai.org/blog/trends-in-gpu-price-performance.

[24] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[25] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[26] Ildoo Kim. Accessed in December 2023. ildoonet/pytorch-randaugment. https://github.com/ildoonet/pytorch-randaugment.

[27] Taeyoon Kim, Chanho Park, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-yong Shin, and Myeongjae Jeon. Accessed in December 2023. Data Diversification Analysis on Data Preprocessing. https://doi.org/10.5281/zenodo.8378456.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[29] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[30] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550, 2021.

[31] Yonggang Li, Guosheng Hu, Timothy Hospedales, Neil Robertson, Yongxin Yang, et al. DADA: Differentiable Automatic Data Augmentation. In *European Conference on Computer Vision*, 2020.

[32] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning (ICML 18)*, pages 3043–3052, 2018.

[33] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175, 2021.

[34] Sungbin Lim, Ildoo Kim, Taesup Kim, Chiheon Kim, and Sungwoong Kim. Fast AutoAugment. *Advances in Neural Information Processing Systems*, 32, 2019.

[35] Tom Ching LingChen, Ava Khonsari, Amirreza Lashkari, Mina Rafi Nazari, Jaspreet Singh Sambee, and Mario A Nascimento. Uniformaugment: A search-free probabilistic data augmentation approach. *arXiv preprint arXiv:2003.14348*, 2020.

[36] Aoming Liu, Zehao Huang, Zhiwu Huang, and Naiyan Wang. Direct Differentiable Augmentation Search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12219–12228, 2021.

[37] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. Prague: High-performance heterogeneity-aware asynchronous decentralized training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, pages 401–416, 2020.

[38] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 21)*, pages 2262–2270, 2021.

[39] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond $GPUs$ for DNN Scheduling on $Multi-Tenant$ Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[40] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. In *Proceedings of the VLDB Endowment*, pages 771–784, 2021.

[41] Samuel G Müller and Frank Hutter. Trivialaugment: Tuning-free yet state-of-the-art data augmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 774–782, 2021.

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

[43] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[44] Accessed in December 2023. FAST AI DATA PREPROCESSING WITH NVIDIA DALI. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9925-fast-ai-data-pre-processing-with-nvidia-dali.pdf.

[45] Accessed in December 2023. NVIDIA Data Loading Library. https://developer.nvidia.com/dali.

[46] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779*, 2019.

[47] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. TrainBox: An Extreme-Scale Neural Network Training Server Architecture by Systematically Balancing Operations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 825–838. IEEE, 2020.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[49] 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.

[50] Accessed in December 2023. PyTorch Memory Management. https://pytorch.org/docs/stable/notes/cuda.html#memory-management.

[51] Libo Qin, Minheng Ni, Yue Zhang, and Wanxiang Che. Cosda-ml: Multi-lingual code-switching data augmentation for zero-shot cross-lingual nlp. *arXiv preprint arXiv:2006.06402*, 2020.

[52] Sylvestre-Alvise Rebuffi, Sven Gowal, Dan A. Calian, Florian Stimberg, Olivia Wiles, and Timothy Mann. Fixing Data Augmentation to Improve Adversarial Robustness, 2021.

[53] Colorado J Reed, Sean Metzger, Aravind Srinivas, Trevor Darrell, and Kurt Keutzer. SelfAugment: Automatic Augmentation Policies for Self-Supervised Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2674–2683, 2021.

[54] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[55] 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)*, pages 1–13, 2016.

[56] Accessed in December 2023. RobustBench. https://robustbench.github.io/.

[57] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[58] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[61] Accessed in December 2023. TensorFlow Studying Part II for GPU. https://www.slideshare.net/teyenliu/tensorflow-studying-part-ii-for-gpu.

[62] Andreas Steiner, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers. *arXiv preprint arXiv:2106.10270*, 2021.

[63] Accessed in December 2023. Module: tfm.vision.augment. https://www.tensorflow.org/api_docs/python/tfm/vision/augment.

[64] Accessed in December 2023. Torchvision: Transforming and Augmenting Images. https://pytorch.org/vision/stable/transforms.html.

[65] Accessed in December 2023. PyTorch 2.0. https://pytorch.org/get-started/pytorch-2.0/.

[66] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. *Proceedings of the VLDB Endowment*, 16(5):1086–1099, 2023.

[67] Riccardo Volpi, Hongseok Namkoong, Ozan Sener, John C Duchi, Vittorio Murino, and Silvio Savarese. Generalizing to unseen domains via adversarial data augmentation. *Advances in neural information processing systems*, 31, 2018.

[68] Guanhua Wang, Kehan Wang, Kenan Jiang, XIANGJUN LI, and Ion Stoica. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *2021 Proceedings of Machine Learning and Systems (MLSys 21)*, pages 696–710, 2021.

[69] Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[70] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245. IEEE, 2019.

[71] Xinyu Zhang, Qiang Wang, Jian Zhang, and Zhao Zhong. Adversarial AutoAugment. In *International Conference on Learning Representations*, 2019.

[72] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, et al. GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[73] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA 22)*, pages 1042–1057, 2022.

[74] Yu Zheng, Zhi Zhang, Shen Yan, and Mi Zhang. Deep AutoAugment. In *International Conference on Learning Representations*, 2022.

[75] Zheng, Lianmin and Li, Zhuohan and Zhang, Hao and Zhuang, Yonghao and Chen, Zhifeng and Huang, Yanping and Wang, Yida and Xu, Yuanzhong and Zhuo, Danyang and Xing, Eric P and others. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[76] Fengwei Zhou, Jiawei Li, Chuanlong Xie, Fei Chen, Lanqing Hong, Rui Sun, and Zhenguo Li. Metaaugment: Sample-aware data augmentation policy learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11097–11105, 2021.