SpeedyLoader: Efficient Pipelining of Data Preprocessing and Machine Learning Training

Rahma Nouaji
rahma.nouaji@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Stella Bitchebe
stella.bitchebe@mcgill.ca
McGill University
Montreal, Quebec, Canada

Oana Balmau
oana.balmau@cs.mcgill.ca
McGill University
Montreal, Quebec, Canada

Abstract

Data preprocessing consisting of tasks like sample resizing, cropping, and filtering, is a crucial step in machine learning (ML) workflows. Even though the preprocessing step is largely ignored by work that focuses on optimizing training algorithms, in practice for many workloads preprocessing and training are pipelined. Popular ML frameworks like PyTorch use *data loaders* to feed data into model training. If the pipeline between preprocessing and training is not done carefully, it can cause significant waiting times on the GPU side. To address this limitation, we introduce SpeedyLoader, a system that overlaps preprocessing and training by leveraging asynchronous data preprocessing and avoiding head-of-line blocking. SpeedyLoader incorporates dedicated data loading threads, which organize preprocessed samples into queues based on their predicted processing times. Concurrently, GPUs fetch samples from these queues, ensuring training is not impeded by preprocessing completion. Compared to the default PyTorch DataLoader, SpeedyLoader reduces training time by up to 30% and increases GPU usage by 4.3×, all while maintaining a consistent evaluation accuracy of 91%.

CCS Concepts: • **Computing methodologies → Machine learning**; *Parallel algorithms*; • **Computer systems organization → Data flow architectures**.

Keywords: Machine learning, Dataloader, GPU-CPU overlap, Data preprocessing, Training, Pipelining

ACM Reference Format:
Rahma Nouaji, Stella Bitchebe, and Oana Balmau. 2024. SpeedyLoader: Efficient Pipelining of Data Preprocessing and Machine Learning Training. In *4th Workshop on Machine Learning and Systems (EuroMLSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3642970.3655824

ACM ISBN 979-8-4007-0541-0/24/04…$15.00
https://doi.org/10.1145/3642970.3655824

1 Introduction

The efficacy of Machine Learning (ML) deployments relies on high-quality data–obtained through *data preprocessing*–and high-quality algorithms. The latter has attracted significant attention, leading to numerous techniques [16, 17, 23], software frameworks [4, 5, 10], and hardware accelerators (e.g., GPU, TPU, DPU, and other ASICs). Though data preprocessing has not received much attention relative to the work done to improve training algorithms, data sample quality and processing efficiency (e.g., via operations like cropping, resizing, filtering, etc.) are crucial to the training process. Recent work shows that preprocessing has a significant impact on learning speed, prediction accuracy, energy efficiency, and scalability [14, 19, 27].

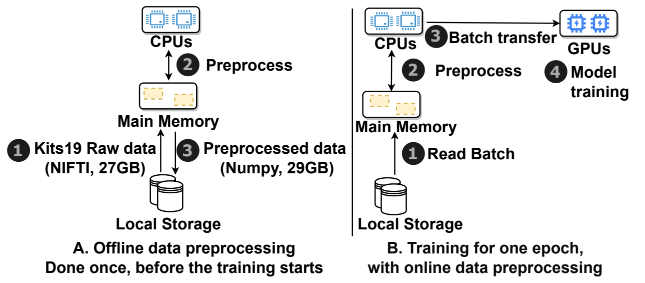

CPUs
2 Preprocess
Main Memory
1 Kits19 Raw data (NIFTI, 27GB)
3 Preprocessed data (Numpy, 29GB)
Local Storage
A. Offline data preprocessing
Done once, before the training starts
CPUs
3 Batch transfer
GPUs
2 Preprocess
4 Model training
Main Memory
1 Read Batch
Local Storage
B. Training for one epoch, with online data preprocessing

Figure 1. Overview of the ML data preprocessing pipeline. Step A involves one-time offline preprocessing. Step B shows the training phase with online preprocessing executed at the beginning of each epoch.

Figure 1 shows a typical workflow for data preprocessing in a computer vision application selected from the MLPerf Training Benchmark suite [18]. Data preprocessing is done in two stages: *offline* preprocessing (Figure 1A) and *online* preprocessing (Figure 1B). Both online and offline preprocessing load data into system main memory and then perform transformations in the CPU. Offline preprocessing occurs before the training begins, whereas *online* preprocessing is done on each batch of images during the training process. Depending on the dataset size, offline preprocessing can span several hours to several days worth of CPU time [6].

Online preprocessing is more lightweight and typically involves randomness in the types of transformations that are performed on each sample (i.e., on each image). We show that randomness can significantly reduce training time.

Prior work that focuses on training largely ignores the overhead of data preprocessing, performing these operations in advance (e.g., as shown in the latest MLPerf Training benchmark competition results [3]). This approach is acceptable because the primary focus of training-oriented research is to demonstrate enhancements in training techniques, typically on static datasets. However, practical scenarios involve continuous data ingestion (e.g., as observed in [26, 27]). Continuous data ingestion and training pose a pressing need for efficient pipelining of preprocessing and training workflows.

This paper takes the first steps towards understanding the impact of pipelining data preprocessing and training for computer vision workloads. We focus on the 3D-UNet model for a 3D image segmentation task, selected from the MLPerf Training Benchmark suite [18].

First, we provide an overview of the preprocessing operations in our selected workload and their impact on performance. Interestingly, we show large variations in the processing times of different samples, mainly stemming from fluctuations in image sizes and the random selection of the performed operations during online preprocessing. Then, we analyze the interaction between the GPU and the PyTorch Data Loader [8, 9]. Surprisingly, we show how a straightforward pipelining between the data loader and the GPU can lead to significant GPU stalls when data preprocessing is considered a part of the training workflow. We show that the GPU is idle 85% of the time on average.

To address this inefficiency, we propose SpeedyLoader, a new data loader for PyTorch that achieves 4.3× higher average GPU utilization than the PyTorch default data loader when data preprocessing and training are pipelined. SpeedyLoader carefully overlaps preprocessing and training ensuring that the GPU never waits for a batch to be ready. Instead of having the GPU directly access the data loader to request data, SpeedyLoader has data loader worker threads asynchronously add preprocessed data samples to a shared queue. Data preprocessing is further accelerated by splitting preprocessing requests into time-consuming requests and short requests by a load balancer, in order to avoid head of line blocking. The load balancer uses heuristics we derive from the study of the workload to classify the requests. SpeedyLoader achieves up to 30% improvement of the total training time for an image segmentation workload based on 3D-UNet model [11] and the KiTS19 [1] dataset.

This study and solution show the potential for significant performance gains by optimizing the interactions between the data loader and the compute. For future work, we plan to expand our SpeedyLoader to include more types of workloads and to support workload collocation.

In sum, this paper makes the following contributions:

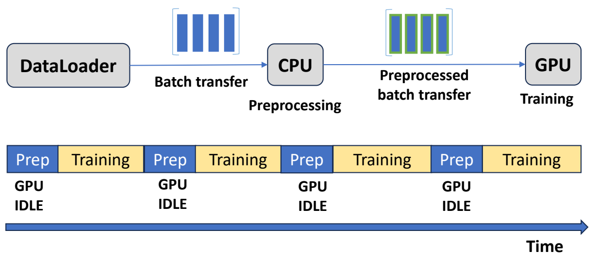


Figure 2. Overview of the pipelining inefficiency of preprocessing and training when using the Pytorch DataLoader.

1. An analysis of data preprocessing overhead in computer vision workloads, shedding light on its implications when pipelining with training.
2. An examination of the PyTorch dataLoader, including a discussion of its limitations and challenges.
3. SpeedyLoader, a new data loader that achieves up to 30% better training times than the PyTorch DataLoader when data preprocessing and training are pipelined.

2 Background

This section provides details on the pipelining of preprocessing and training, the PyTorch DataLoader and the computer vision workload analyzed in our case study.

2.1 Pipelining of Data Preprocessing and Training

Figure 1 presents an illustration of the ML preprocessing and training pipeline, comprising two key steps labeled A and B. Step A showcases the initial offline data preprocessing that occurs only once before the start of the training phase. Initially, 1 the pipeline loads the 27GB KiTS19 NIFTI-formatted data from local storage into the main memory. Subsequently,2 the offline preprocessing phase is initiated on the CPU, and 3 the preprocessed data is stored locally as a 29GB NumPy dataset. After the offline preprocessing, step B starts. Step B illustrates the training phase, which involves online data preprocessing. The latter takes place at the onset of each training epoch. During this step, 1 the preprocessed data is read from local storage and loaded into main memory in batches, and 2 the CPU performs online preprocessing. Finally, 3 the data is fed to the GPU, which 4 trains it once the online preprocessing step is completed.

2.2 PyTorch DataLoader

In general, the data loader serves as a bridge between the dataset and the machine learning model, streamlining and optimizing the process of accessing data for preprocessing, training and inference tasks.

The PyTorch DataLoader [8] iterates over datasets, making data readily accessible to the model in an optimized format. It handles large datasets by fetching data from memory and disk, enables on-the-fly data augmentation and preprocessing, and manages tasks like batching, shuffling, and parallel data loading. The PyTorch DataLoader and Dataset classes [9] are at the core of loading large datasets efficiently. The Dataset class requires implementation of two main functions: `__getitem__` and `__len__`. `__len__` provides dataset size for batch division. `__getitem__` loads a sample at a given index, performing on-the-fly preprocessing and returning it with the corresponding label. The Data loader then iterates over batches during training, using the methods `__iter__` and `__next__`, initializing necessary variables and fetching data. The Data loader also efficiently controls the number of parallel threads for data loading through the `num_workers` parameter. In this study, we use DataLoader from PyTorch version 2.1.2.

Figure 2 depicts data transfer within the ML pipeline. The diagram outlines the journey from the PyTorch DataLoader to the CPU for data preprocessing and between the CPU and GPU for model training. The associated timeline underscores the sequential progression, highlighting that preprocessing happens before training. Note that during preprocessing, the GPU remains *idle*, as shown in the timeline below the preprocessing (PREP) bar, anticipating the end of batch preprocessing by the CPU. This idleness is identified as an inefficiency, representing an opportunity to enhance resource utilization and accelerate training times.

2.3 3D Image Segmentation Workload

For our case study, we select the 3D Image Segmentation Workload from the MLPerf Training Benchmark suite [18, 20]. MLPerf Training includes various computer vision workloads such as ResNet, SSD, and ImageNet.

We opted for 3D-UNet workload because it presents multiple data preprocessing techniques for a computer vision task, leaving the exploration of additional workloads for future research. Additionally, this workload was chosen because the dataset size is relatively manageable, amounting to 29GB after offline preprocessing, as shown in Figure 1. This workload uses the KiTS19 Dataset and the 3D-UNet model.

The KiTS19 challenge dataset [1, 15] includes segmented Computed Tomography (CT) imaging for 210 patients (cases) who underwent nephrectomy. Each case consists of grayscale images with corresponding masks in the Neuroimaging Informatics Technology Initiative (NIFTI) format and is presented using a tuple (`imaging.nii.gz`, `segmentation.nii.gz`), representing the image and its related label, respectively. Each image has an *image spacing* parameter, refering to the physical distance between adjacent pixels, and defining the image scale in real-world units, such as millimeters per pixel.

The 3D-UNet model [11] is an extension of the U-Net architecture tailored for three-dimensional (3D) medical image segmentation. With an encoder-decoder structure, it efficiently captures volumetric information crucial in medical imaging. The encoder hierarchically extracts features, and the decoder reconstructs the segmented output. Skip connections facilitate low-level and high-level feature fusion, supporting precise segmentation. Its U-shaped architecture preserves spatial information, making it effective for tasks like organ segmentation in volumetric medical scans, capturing context and spatial relationships within 3D data.

3 Study of Preprocessing Overhead

3.1 Offline Preprocessing Overhead

The first step is transforming the dataset into a Numpy format, e.g., (`case_00034_x.npy`, `case_00034_y.npy`), corresponding to the image and its label, respectively. This transformation is essential because the NIFTI format is primarily designed for medical researchers and is not straightforward to preprocess using typical Python-based data libraries like Numpy. The following techniques are used to perform offline preprocessing for an image segmentation workload after its conversion to the Numpy format.

3D Resampling. This method achieves standardized image spacing by interpolating both image and label volumes. The technique involves comparing each image spacings with a predefined target (set to [1.6, 1.2, 1.2]) and computing a new shape that aligns with this target.

Mininum Padding. This technique ensures uniformity in the spatial dimensions of input images. The method dynamically pads images and corresponding labels to make the size closer to a target shape ([1, 128, 128, 128]).

Intensity normalization. This method enhances image intensity comparability through clipping pixel values to mitigate outliers, followed by normalization centered on a predefined mean and scaled by a standard deviation.

Figure 3 shows the processing time of these techniques on two image cases: the slowest (`case_00165`) and the quickest (`case_00130`) ones. Variations in processing times are linked to image sizes. For instance, the initial dimensions of `case_00165` are [1, 734, 512, 512], contrasting with `case_00130` at [1, 53, 512, 512]. Additionally, the average processing times for Intensity Normalization and Min Padding are 115ms and 55ms, respectively. 3D Resampling, with an average processing time of 380ms across 210 images, emerges as the most time-consuming due to the nature of operations involved, notably the interpolation of the entire image volume to a new size. According to the memory profiler used [2], this interpolation step consumes a substantial amount of memory, approximately 373.5MiB for `case_00165`.

For the offline preprocessing of the KiTS19 dataset, we used a single CPU core which requires a total of 10 minutes to complete preprocessing a raw dataset of 27GB. This phase scales linearly in time and resource consumption (compute and memory) with the dataset size.

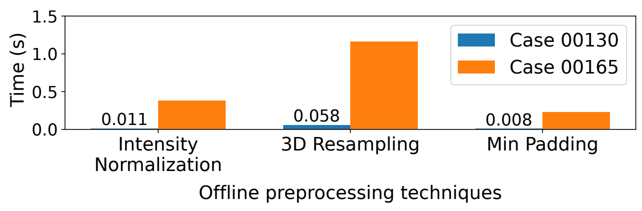


Figure 3. Offline preprocessing time for `case_00165` and `case_00130`. The 3D Resampling dominates cost. The average time is 380ms for 3D resampling, 115ms for Intensity Normalization, and 55ms for Min padding.

3.2 Online Preprocessing Overhead

Online preprocessing takes place for each sample, in every epoch. The treatments to apply to images during online preprocessing are the following.

Random Balance Crop. This treatment ensures that all dataset images have the same shape [1, 128, 128, 128]. It is based on a specified oversampling parameter set to 0.4. It employs foreground-aware cropping when the oversampling condition is met, ensuring that the cropped region encompasses relevant foreground structures. The method utilizes randomization to efficiently explore the image space, contributing to the diversity of the training dataset.

Cast. This method converts the data types of input images to *np.float32*, and labels to *np.uint8*.

Gaussian Noise. This technique applies random Gaussian noise to input images with 0.1 probability.

Random Flip. This transformation introduces random axis flips (1/3 chance for each axis) on both image and label data.

Random Brightness Augmentation. This method applies random brightness adjustments to the input image with 0.3 factor and 0.1 probability.

Table 1 emphasizes the impact of randomness within the dataset on processing times. The processing time variations among different preprocessing techniques in Table 1 stem from inherent characteristics and randomness rather than being image-specific. The Table illustrates the processing time for various online preprocessing methods applied to the same image (case_00039) in two different training runs. In the context of Random Flip, the processing time varies between 2.762×10^{-3} ms in run 1 and 32 ms in run 2. Our memory profiler highlights the random condition for flipping images with `np.flip` is inconsistently met, given its application probability of 1/3. This inconsistency results in the observed 32 ms duration and 120 MiB memory usage during the flip operation. Conversely, the Cast technique exhibits constant operation times, as it lacks randomness. Random Brightness Augmentation introduces a 0.1 probability of transformation application, resulting in 6 ms in run 1, with an associated 8.1 MiB memory consumption, and 1.054×10^{-3} ms in run 2. Gaussian Noise, being random, yields a processing time

Table 1. Execution time (in ms) for Online Preprocessing Techniques on case_00039 for two training runs.

Technique	Time run 1 (ms)	Time run 2 (ms)
Random flip	2.762×10^{-3}	32
Cast	3	3
Random Brightness Aug	6	1.054×10^{-3}
Gaussian Noise	2.005×10^{-3}	149
Random Balance Crop	965	0.172

of 149 ms in the second run, with a 17.32 MiB memory increase. In the first run, with no added Gaussian noise, the time drops to 2.005×10^{-3} ms. The most time-intensive operation is observed with Random Balance Crop, involving a 0.4 probability of applying random foreground cropping. This operation, due to fetching foreground pixels, results in a processing time of 965ms and a 20MiB memory increase in run 1 with foreground cropping, compared to 0.172ms in the run without it.

Takeaway message: *This study on data preprocessing highlights that image processing time during data preprocessing is primarily influenced by two factors: the image size, discussed in offline preprocessing (Section 3.1), and the randomness introduced by online preprocessing transformations (Section 3.2).*

4 Design

The primary objective of SpeedyLoader is to enhance training time efficiency and GPU utilization by efficient pipelining between preprocessing and training.

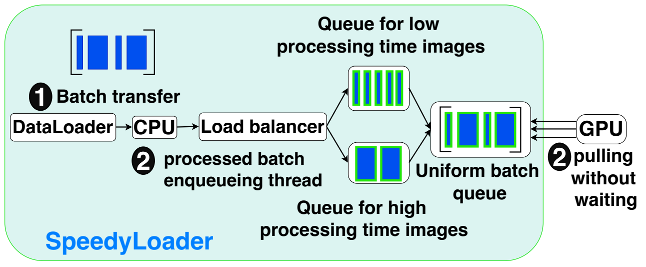

Queue for low processing time images
1 Batch transfer
DataLoader
CPU
Load balancer
2 processed batch enqueueing thread
Uniform batch queue
GPU
2 pulling without waiting
Queue for high processing time images
SpeedyLoader

Figure 4. Overview of SpeedyLoader. SpeedyLoader employs a continuous thread to enqueue preprocessed samples into a specified queue based on a load balancer decision. Simultaneously, the GPU dequeues preprocessed data.

SpeedyLoader Overview. Figure 4 illustrates SpeedyLoader's architecture. The implementation of SpeedyLoader relies on the default PyTorch DataLoader, enhanced by a load balancer, a different organization of the data loader worker threads into dedicated thread pools, and a collection of shared queues. On a high level, three main ideas lie at the core of SpeedyLoader's design. First, SpeedyLoader aims to ensure realistic chaining of the preprocessing and training, as opposed to working on static datasets. To this end, SpeedyLoader is responsible for both offline and online processing which are combined in one preprocessing step. This design decision was taken to be able to handle training on dynamic datasets.

Second, to avoid head-of-line blocking, the sample preprocessing is separated into two: a set of samples that the SpeedyLoader load balancer expects to take a long time to process, and a set of samples with an expected short processing time. This way, small preprocessing jobs are never blocked by long jobs. The data loader worker threads are separated into two thread pools, one for short jobs and one for long jobs. In case threads in the long job thread pool are idle, they will engage in work-stealing from the short job queue. The load balancer is responsible for enqueuing batch samples into the appropriate queue. To make a decision, it employs heuristics that consider both the image size (Section 3.1) and the randomness of transformations applied during online preprocessing (Section 3.2).

Third, SpeedyLoader aims to maximize overlap between preprocessing and training in order to have the GPU working at 100% utilization most of the time. To this end, we introduce a pipelining of data loader worker threads and GPU threads via a shared producer-consumer queue. The queue is accessed concurrently and continuously by the data loader threads acting as producers, and by the GPU threads acting as consumers. The sample preprocessing rate is tuned to ensure that the queue is never empty.

Implementation of the load balancer heuristics. We built upon the core functions of PyTorch DataLoader which manage data flow within the ML pipeline: `__getitem__`, `__iter__`, and `__next__`. Then, we enhanced preprocessing efficiency by consolidating online and offline preprocessing into a single online block within the `__getitem__` method of the Data loader. This method facilitates on-the-fly preprocessing of batches, as discussed in Section 2.2. By incorporating offline data preprocessing techniques into the online phase, we take advantage of the predictability of image processing time, which is established during offline preprocessing (refer to Section 3.1), based on the image size. Additionally, we leverage the predictability of time consumption in the former online data preprocessing step, which relies on the application of random transformations 3.2. Before applying the online preprocessing transformations, the load balancer calls the random number generator to determine which transformations should occur in that given epoch for that sample, without executing the transformations. It then annotates the sample with the results of the random number generator and sends it to the appropriate queue. Consequently, SpeedyLoader benefits from two key heuristics for predicting the potential duration of image processing: the image size and the in-advance knowledge of the random transformations that will be applied to it.

Implementation of the uniform batch queue. SpeedyLoader integrates an asynchronous loading thread initiated within the `__iter__` method, where it initializes the variables for the PyTorch DataLoader. This thread is used to populate queues with preprocessed batch samples from the CPU, while the GPU retrieves batches from the uniform queue to start the model training step, called inside the `__next__` method. This uniform queue is used to maintain the same batch composition. SpeedyLoader uses a FIFO queue data structure provided by the queue module from Python [7]. This queue, inherently thread-safe, incorporates necessary locking semantics to ensure synchronized access for both enqueue and dequeue operations.

5 Evaluation

Our experiments aim to answer the following questions, given a pipelining of data preprocessing and training:

(Q1) Does SpeedyLoader accelerate training time compared to the PyTorch DataLoader?

(Q2) Does SpeedyLoader achieve overlap between CPU and GPU execution?

(Q3) Does SpeedyLoader achieve a higher utilization of compute resources compared to the PyTorch DataLoader?

5.1 Experimental Environment

System. We use an NVIDIA DGX-1 machine, with a 2.20GHz 80-core Intel Xeon processor, 512GB of memory, and 8 NVIDIA V100 32GB GPUs, running CUDA 12.3 and PyTorch 2.1.2.

Metrics. We consider three main metrics: the total training time, GPU use, and CPU use. The total training time is measured using the `time` function from Python, GPU use is monitored using the `nvidia-smi` tool, and CPU use is measured using the `top` command.

Methodology. We compare SpeedyLoader with the PyTorch DataLoader version 2.1.2 [8]. Initially, we removed randomness from online data preprocessing for a deterministic approach. Without randomness, 300 epochs were needed for an 85% accuracy, while with randomness, only 250 epochs achieved a 91% accuracy. Consequently, we present results for the scenario with randomness. We trained our workload for 5 epochs with 8 GPUs. We monitored GPU and CPU usage for both our proposed methodology and the default one. We used 30 workers for both the SpeedyLoader and default PyTorch DataLoader, and a batch size of 4. Through experimentation, we determined that employing 30 workers yielded the optimal training time along with the highest evaluation accuracy. We capped the queue size for SpeedyLoader at a maximum of 20 elements. This prevents the queue from expanding significantly, considering the presence of 30 workers (loading threads) in the data loader. When the data loader is invoked and samples are enqueued during the second step of SpeedyLoader, as observed in Figure 4, there is potential for contention between the data loader threads (producers) and the GPU threads' (consumers) ability to extract from the queue. If the queue size is capped, the consumers get priority. We have a queue per running GPU. Our experimental results were averaged over 5 runs.

5.2 Evaluation Results

Figure 5 illustrates the difference in training time, including data preprocessing time, between our proposed approach, the SpeedyLoader and the traditional PyTorch DataLoader across various GPU configurations. Training time exhibits a decrease with the number of GPUs. Overall, our approach consistently outperforms the PyTorch DataLoader, showcasing more than 30% improvement in training time. Specifically, with 2 GPUs, the PyTorch DataLoader needs 62 minutes to complete 5 epochs, while our approach accomplishes the same in approximately 44 minutes. For 4 GPUs, the default data loader requires around 29 minutes compared to 20 minutes for SpeedyLoader. Utilizing 8 GPUs results in the fastest training time, with the basic approach taking 15 minutes and our approach completing in just 9 minutes.

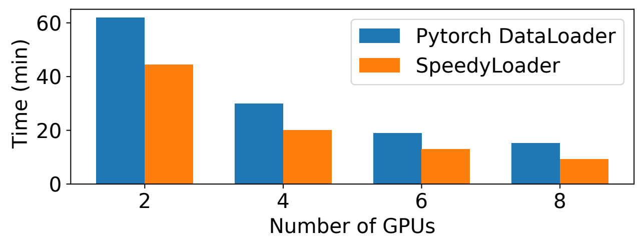
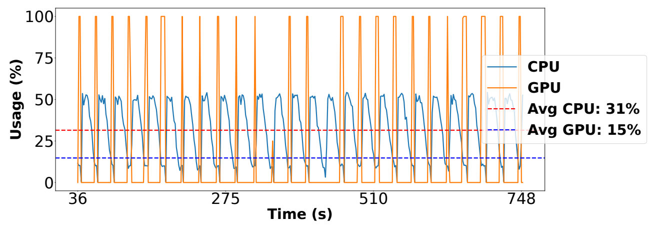


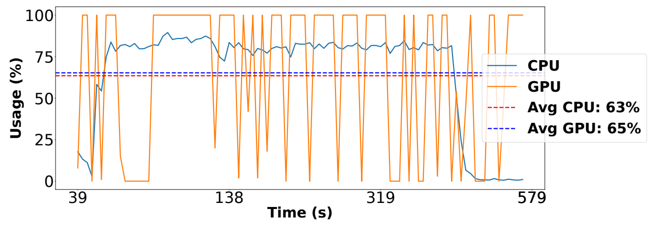
Figure 5. Total training time of 3D-UNet model. SpeedyLoader provides up to 30% improvement compared to the PyTorch DataLoader.

Figure 6a shows GPU and CPU usage for the image segmentation workload using PyTorch DataLoader. The alternation between GPU and CPU usage distinctly illustrates the phases of data preprocessing and training. As previously explained, the CPU peak, signifying data preprocessing, occurs just before the GPU-intensive training phase. While the GPU can run at 100% utilization during the training step, the system ends up spending most of the time waiting for samples from the PyTorch DataLoader which needs to both fetch samples from main memory, as well as perform preprocessing. Consequently, the average CPU and GPU usage for this experiment are only 31% and 14%, respectively.

Figure 6b shows GPU and CPU use for SpeedyLoader. We observe an enhancement in GPU utilization, rising from 14% to 65%. Notably, the overlap between CPU and GPU use



(a) PyTorch DataLoader



(b) SpeedyLoader

Figure 6. CPU and GPU usage when training 3D-UNet using Pythorch DataLoader (6a) and SpeedyLoader (6b), both for 5 epochs, with 8 GPUs and 30 data loader worker threads. SpeedyLoader improves both CPU and GPU utilization by 2× and 4.3× on average, respectively.

aligns with our objective to overlap data preprocessing and training. SpeedyLoader achieves superior GPU utilization, with the GPU spending more time at 100% capacity while maintaining the evaluation accuracy at 91% after training for 250 epochs. We attribute our inability to achieve 100% average GPU utilization to the queue contention problem we encountered. This issue arose due to the global locking mechanism used by the Python-built queue module.

6 Related Work

Many techniques have been proposed to improve training time [13, 21, 22, 24–26, 28]. However, virtually none of the techniques consider the overhead of data preprocessing, as all the preprocessing is done in advance to highlight the pure training time.

tf.data [22] is the closest work to ours. While we focus on PyTorch, this system focuses on the TensorFlow framework. Like SpeedyLoader, it is interested in improving the end-to-end total training time of ML jobs. However, unlike SpeedyLoader, which acts on data preprocessing to achieve this goal, tf.data focuses on the whole input pipeline. While SpeedyLoader proposes to overlap offline and online preprocessing with the training to enhance the performance of ML models' training time, tf.data, instead, overlaps computation and communication to achieve optimal performance.

Snap ML [12] is a framework for fast training of generalized linear models. Like SpeedyLoader, it is addressing GPU underutilization, by among other things, pipelining training with data loading to increase GPU utilization. However, unlike SpeedyLoader, which acts on a more sophisticated workload, Snap ML focuses on generalized linear models. While SpeedyLoader proposes to overlap offline and online preprocessing with the training to enhance the performance of ML models' training time, Snap ML, instead, looks at the inter and intra node communication and Numa locality to achieve optimal performance.

Cachew [13] is a service for ML data processing that scales distributed compute resources to avoid stalls in training jobs. Cachew does not focus on data preprocessing or input pipelines data loading, unlike SpeedyLoader. Its logic is rather to reuse already preprocessed data within and across jobs by applying caching and scaling policies where and when it is performance- and/or cost-effective. As with tf.data, combining the caching policy of Cachew with the data loading and data preprocessing algorithms of SpeedyLoader could lead to better results.

μ-TWO [24] focuses on maximizing GPU utilization. To this end, it proposes a novel compiler that leverages selective data swapping from GPU to host memory only when absolutely necessary, and overlaps data movement with computation so that GPUs never wait for data. μ-TWO is orthogonal to SpeedyLoader in that the latter indirectly reaches its primary goal of reducing GPU stalls.

7 Conclusion and Future Work

This paper first presented a case study of data preprocessing techniques in computer vision workloads and exposed the performance bottlenecks that can emerge when pipelining preprocessing and training. To tackle these bottlenecks, we introduced the SpeedyLoader data loader. SpeedyLoader uses a shared queue between the data loader worker threads and the GPU worker threads, and implements load balancing techniques to mitigate head-of-line blocking caused by images with a large processing time. SpeedyLoader achieves a 30% reduction of the training time, mirrored by a 4.3× increase in GPU usage.

SpeedyLoader is a first step towards efficient pipelining of preprocessing and training. Looking ahead, our goals include achieving 100% GPU usage by further optimizing the implementation of the data loader (e.g., integrating a custom queue with finer-grained locking). We also plan to extend our study of workloads and heuristics for data preprocessing times beyond computer vision (e.g., large language models, recommendation systems). Finally, we plan to extend SpeedyLoader to support collocation of different workloads (e.g., by integrating it with systems such as Orion [25]).

References

[1] [n. d.]. KiTS19 Challenge Dataset. https://kits19.grand-challenge.org/data/.

[2] [n. d.]. memory-profiler. PyPI. https://pypi.org/project/memory-profiler/

[3] [n. d.]. MLPerf Training Benchmark Suite V3.1 Results. https://mlcommons.org/benchmarks/training/.

[4] [n. d.]. NumPy - The fundamental Package for scientific computing with Python. https://numpy.org/.

[5] [n. d.]. Pandas: powerful Python data analysis toolkit. https://pypi.org/project/pandas/.

[6] [n. d.]. Personal communication with collaborators in Apple, Tesla, and Nutanix.

[7] [n. d.]. Python Documentation - Queue. https://docs.python.org/3/library/queue.html.

[8] [n. d.]. PyTorch 2.0. https://pytorch.org/.

[9] [n. d.]. PyTorch Tutorial. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.

[10] [n. d.]. Scikit-learn - Machine Learning in Python. https://scikit-learn.org/stable/.

[11] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 2016. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2016: 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II 19.*

[12] Celestine Dünner, Thomas Parnell, Dimitrios Sarigiannis, Nikolas Ioannou, Andreea Anghel, Gummadi Ravi, Madhusudanan Kandasamy, and Haralampos Pozidis. 2018. Snap ML: A hierarchical framework for machine learning. *Advances in Neural Information Processing Systems* (2018).

[13] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing As A Service. In *Proceedings of USENIX ATC 22.*

[14] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2022. Chasing Carbon: The Elusive Environmental Footprint of Computing. *IEEE Micro* 4 (2022).

[15] Nicholas Heller, Niranjan Sathianathen, Arveen Kalapara, Edward Walczak, Keenan Moore, Heather Kaluzniak, Joel Rosenberg, Paul Blake, Zachary Rengel, Makinna Oestreich, et al. 2019. The kits19 challenge data: 300 kidney tumor cases with clinical context, ct semantic segmentations, and surgical outcomes. *arXiv preprint arXiv:1904.00445* (2019).

[16] Sotiris Kotsiantis, Dimitris Kanellopoulos, and P. Pintelas. 2006. Data Preprocessing for Supervised Learning. *International Journal of Computer Science* (2006).

[17] S. Maetschke, R. Tennakoon, C. Vecchiola, and R. Garnavi. 2018. nuts-flow/ml: data pre-processing for deep learning.

[18] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. MlPerf Training Benchmark. *Proceedings of Machine Learning and Systems* (2020).

[19] Mark Mazumder, Colby Banbury, Xiaozhe Yao, Bojan Karlaš, William Gaviria Rojas, Sudnya Diamos, Greg Diamos, Lynn He, Alicia Parrish, Hannah Rose Kirk, et al. 2022. Dataperf: Benchmarks for data-centric ai development. *arXiv preprint arXiv:2207.10062* (2022).

[20] MLCommons. [n. d.]. MLPerf Benchmarking Suite - PyTorch implementation for image segmentation. https://github.com/mlcommons/training/tree/master/image_segmentation/pytorch.

[21] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *Proceedings of USENIX OSDI'22.*

[22] Derek Gordon Murray, Jirí Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *VLDB Endowment* (2021).

[23] G. Thippa Reddy, M. Praveen Kumar Reddy, Kuruva Lakshmanna, Rajesh Kaluri, Dharmendra Singh Rajput, Gautam Srivastava, and Thar Baker. 2020. Analysis of Dimensionality Reduction Techniques on Big Data. *IEEE Access* (2020).

[24] Purandare Sanket, Wasay Abdul, Idreos Stratos, and Jain Animesh. 2023. μ-TWO: 3× Faster Multi-Model Training with Orchestration and

Memory Optimization. In *Proceedings of MLSys'23*.

[25] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of Eurosys'24*.

[26] Chun-Feng Wu, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2022. A joint management middleware to improve training performance of deep recommendation systems with SSDs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*.

[27] 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. 2022. 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*.

[28] Mark Zhao, Dhruv Choudhary, Devashish Tyagi, Ajay Somani, Max Kaplan, Sung-Han Lin, Sarunya Pumma, Jongsoo Park, Aarti Basant, Niket Agarwal, Carole-Jean Wu, and Christos Kozyrakis. 2023. RecD: Deduplication for End-to-End Deep Learning Recommendation Model Training Infrastructure.