Characterizing and Modelling the I/O Patterns of Deep Learning Training Workloads

Loïc Ho-Von

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

School of Computer Science

McGill University
Montréal, Québec, Canada

June 2023

Abstract

As Deep Learning models, computational infrastructure and datasets keep scaling up, data loading and I/O are increasingly becoming bottlenecks to the training process, resulting in increased training times and resource under utilization. In this work, we take a close look at the I/O patterns of three Deep Learning training workloads from varying domains: UNET3D for image segmentation, BERT for NLP and DLRM for recommendation. We first use eBPF traces and instrumentation to characterize their I/O patterns at the applicative, file system and block levels. After deriving relationships for the model computation time based on the batch size and number of accelerators, studying the effects of using synthetic data and replacing the computation by a `sleep()` in the real workloads, we implement their emulation in the MLPerf Storage benchmark. Finally, we demonstrate that the benchmark accurately recreates the I/O loads of the real workloads, achieving similarities of 95 % or more.

Abrégé

Alors que les modèles d'apprentissage profond, leur infrastructure informatique et leurs jeux de données ne font que croître, le chargement des données d'entrainement est de plus en plus souvent le maillon faible, augmentant le temps nécessaire pour l'entrainement et diminuant l'utilisation des ressources. Dans cette thèse, nous étudions en détail le profil de chargement de données de trois modèles d'apprentissage profond: UNET3D pour la segmentation d'images 3D, BERT pour les taches de traitement du langage naturel, et DLRM pour la recommandation. Nous utilisons premièrement des programmes eBPF et de l'instrumentation afin de caractériser l'entrainement de ces modèles au niveau de l'application, du système de fichier et de l'interface avec le système de stockage. Nous en dérivons des équations pour le temps de calcul des modèles en fonction de la taille du lot et du nombre d'accélérateurs utilisés. Après avoir étudier l'effet de jeux de données synthétiques sur le chargement, ainsi que l'effet de remplacer le calcul des modèles par la fonction `sleep()`, nous développons le MLPerf Storage benchmark qui met en œuvre l'émulation des ces modèles. Finalement, nous vérifions que l'émulation est fidèle à la réalité, obtenant des scores de similarité de 95% ou plus.

Acknowledgements

I would like to thank my parents for always putting education first, my friends and my beautiful girlfriend Farah for their understanding through these studious times. Thanks to McGill University for accepting me into this program and for the $1000 GREAT travel award.

Thanks to the members of the DISCS lab at McGill who did a lot of work in this project, notably Yuyan for the data generation code for all workloads, Zhongjie for integrating DLRM in the benchmark and Aidan for work on the similarity metric code.

Thanks to the MLPerf Storage working group for their feedback on part of this work. Thanks to the developers of DLIO, which serves as the basis of the benchmark.

Most of all, thank you to my supervisor Oana Balmau for putting me on this project in the first place, for the continued guidance, trust, lab lunches and the chance to attend SC22 in Dallas, Texas to present part of this work.

Table of Contents

List of Tables

List of Figures

List of Programs

1

Introduction

Training a Deep Learning model requires a few essential ingredients: a model, the computational resources to train that model, and a dataset. Model accuracy having been shown to scale with both model and dataset size, it comes as no surprise that both have seen sustained growth over the years [12]. At the same time, ever more powerful hardware accelerators are being developed to provide these increasingly large models with the computational power needed to train within reasonable time frames.

Given these trends, the load placed on storage systems during model training has been steadily increasing, reaching a point where I/O costs incurred are increasingly becoming the bottleneck of training jobs [48], leaving computational resources idling as they wait for data.

On a scientific level, the goals of this master's thesis are to characterize the I/O patterns of three DL training workloads and investigate whether they can be reliably reproduced using synthetically generated datasets and without hardware accelerators. On a practical level, this thesis produced

a benchmark tool, an extension of DLIO [21] which allows ML practitioners and researchers to accurately reproduce these I/O patterns on arbitrarily large synthetic datasets, without the need for accelerators. This tool was created in collaboration with MLCommons and is now a part of the MLPerf Storage benchmark [1].

1.1 Thesis Overview

- Chapter 2: Background and Related Work.

  We go through a short history of Deep Learning and the context in which it emerged. We then break down the training process, focusing on data loading and workload distribution to multiple accelerators. Finally, we examine the role of benchmarks in Computer Science and in Deep Learning.

- Chapter 3: Contributions.

  We go over the design of this thesis's main contribution: a benchmark tool able to emulate the training workloads, generating realistic load on storage without the need for accelerators or acquiring the original datasets.

- Chapter 4: Methodology.

  We describe each workload, the eBPF traces used to record workload behaviour at the system level, the instrumentation used to obtain data on the workloads at the application level and the similarity metric used to compare workload runs.

- Chapter 5: Results and Analysis.

  We go through the results of our experiments one by one, providing some analysis and possible explanations for unexpected results.

- Chapter 6: Discussion.

  We discuss some of the limitations of this work, and explore possible directions for future endeavors.

- Chapter 7: Conclusion.

2

Background and Related Work

2.1 Deep Learning

Since the 2010s, Machine Learning (ML), and more specifically its sub-field of Deep Learning (DL), has risen to the forefront of computer science research and captivated the minds of researchers, practitioners and the general public alike. This meteoric ascension and sustained interest in Deep Learning models, which can be broadly defined by their use of multi-layered (deep) artificial neural networks, can be explained by their repeated achievement of state-of-the-art performance on a wide variety of tasks, with far-reaching applications.

The capacity of DL models to automatically learn useful hierarchical representations from high-dimensional data [42] in a process called *training*, combined with the flexibility of their architectures, have allowed their successful application to tasks in such varied domains as Computer Vision (CV) [14, 25, 30], Natural Language Processing (NLP) [41, 55], Speech Processing [33] and Reinforcement

Learning [10]. This broad impact was officially recognized in 2018, by an ACM Turing award awarded to Yoshua Bengio, Geoffrey Hinton and Yann LeCun [28], commonly referred to as the "Godfathers of Deep Learning".

However, Deep Learning's emergence within this time frame was not simply driven by innovations in methods and model architecture, relying instead on two contemporaneous and synergistic trends in computing.

The first is a trend of computational capacity, with the advent of cloud computing, and its model of computation as a service, which increased the accessibility of computing by lowering its costs and maintenance requirements [72], as well as advances in hardware accelerators, most notably in Graphical Processing Units (GPUs), which provide an optimized platform for the linear algebra operations used during DL model training [56]. Today, this relationship has become synergistic, with DL applications driving the development of more specialized hardware accelerators, such as the Cerebras CS-2 system [35], and DL methods even being used in the hardware development process with e.g. Google's Tensor Processing Units (TPUs) [39, 47].

The second essential trend was the advent of the 'Big Data' era, defined by its always increasing data volume, velocity and variety, commonly referred to as the three V's [27, 68]. Big data was, and is, instrumental to Deep Learning's success as it provides the raw materials on which DL models learn. Conversely, Deep Learning methods are essential to analyze modern large and high-dimensional datasets and DL's development was largely driven by the desire to do so.

Over the past decades, datasets in CV and NLP have been growing at rates of around 0.1 and 0.2 orders of magnitude per year, respectively [67] with the latest ones reaching billions of samples for CV, and even trillions of samples for NLP. We can expect these trends to continue given the empirical scaling laws at play in these domains [32, 40, 71]: as the number of parameters in models is increased, they become more performant as well as more computationally efficient, but the dataset and amount of compute need to scale proportionally.

2.2 Data Loading in Deep Learning

In most modern Operating Systems (OS), but primarily those based on the Linux kernel, processes interact with data residing on persistent storage media (hard disk drives, solid-state drives, or

other) through the invocation of OS system calls, syscall for short, such as `read()` or `write()` that interact with the Virtual File System (VFS) layer of the kernel. File Systems (FS) offer a software abstraction over storage devices, transforming their raw blocks-based interface — referred to as the block layer — to a more convenient file-based interface for the OS and its users [9]. The VFS interface allows different FS implementations to be used interchangeably by offering a common set of functionalities; it is an abstraction over different File Systems.

File Systems have access to the OS page cache, a component of its memory management system that caches previously requested data in memory as long as free space allows. Thus, if the same data is requested multiple times, the FS will return it from the page cache the second and subsequent times, avoiding the orders of magnitude higher latency of a storage access [58]. The transfer of data between main memory and persistent storage is referred to as an Input/Output (I/O) operation [26]. In this document, we may use the term loosely to designate interaction with the VFS layer, whether the operations go to storage or are returned directly from cache.

In a typical training workload, a process which we will explain in greater detail further down, the dataset is stored on local or networked storage and is loaded into memory by the DL training process. Once in memory, some more or less computationally-intensive preprocessing takes place after which the data is loaded into the accelerator's memory, to be fed to the model. If the size of main memory allows, the training dataset can be fully cached in the operating system's page cache, and the faster data access times can be achieved. This situation will benefit mostly models that perform multiple passes over their training dataset, called *epochs*. The software operations responsible for loading and preprocessing the data comprise what is referred to as the *input pipeline* or *data pipeline* in the literature.

Faster data loading is highly desirable for DL model training, as the accelerators are often the most expensive resource of the system, and their usage is to be maximized. Put another way, reducing the total training time is highly valuable, and this can be achieved by not "wasting" any of it on waiting for data. To this end, the input pipelines of the most popular DL frameworks (e.g. PyTorch [57], TensorFlow [5], MXNet [15], etc.) and third party data loading libraries meant to be used as drop-in replacement, e.g. NVIDIA DALI [54], implement ways to parallelize data loading, preprocessing and computation, as well as data prefetching and operation pipelining mechanisms.

Given their increasing sizes, it is becoming less and less common for these datasets to fully fit in main memory. This leads to repeatedly incurred I/O cost during training as the page cache returns misses and storage must be accessed. In addition, the standard Least Recently Used (LRU) replacement policy is sub-optimal for DL training workloads, which randomly change the order in which cases are viewed every epoch, for reasons linked to model accuracy. This leads to unnecessary thrashing and decreased training performance [48]. Additionally, as DL models scale up, it is common practice to distribute their training across multiple accelerators and nodes working in parallel, increasing the demand for data put on the storage system.

This has led to a situation where I/O frequently becomes the bottleneck in DL training jobs. For example, a 2021 survey of 9 state-of-the-art models and data pipelines showed that training jobs were spending up to 70% of their time on blocking I/O when only part of the dataset is able to be cached, despite using prefetching and pipelining mechanisms [48]. This holds even for the top players: an analysis of DL training jobs in Google's production data centers showed that 30% of the overall compute time is spent doing data ingestion [49], in a cluster where only 13% of jobs were using datasets of over 1TB. Moreover, as accelerators get faster, the model's data ingestion rate goes up, increasing the impact of I/O on the overall training latency and potentially negating any improvements obtained from the more powerful compute.

As a result, it is increasingly important to understand the I/O behaviour of these training workloads, and empower researchers, storage system developers, cluster architects or other professionals with the means of evaluating the capacity of their solutions to provide the data loading performance necessary to saturate their accelerators during training.

2.3 Deep Learning Training

In Deep Learning, a model is defined as a set of layers each consisting of a set of learnable parameters. These layers perform various operations on data leading to updates to the model's parameters in such a way that the model's measured performance on a given task is improved. These updates occur iteratively in a procedure called *training*.

Deep Learning software *frameworks* are usually used to define, train and eventually deploy models. They provide libraries and recommended workflows to assemble and train models, and handle the management of supported hardware accelerators through low-level libraries such as cuDNN for NVIDIA GPUs [51]. As previously mentioned, these accelerators are optimized for the operations performed during training, mostly tensor-tensor operations between the model parameters and the input data. As such, during training the model is loaded into the accelerator memory to make use of its capabilities. Note that models can train on CPUs but in this work we will only consider accelerator-based training, as that tends to be the norm for the most performant models.

In the *supervised* training paradigm, which all the workloads we will consider adhere to, the data is usually composed of `(sample, label)` pairs, where the sample is the input to the model, and the label is the expected output. Both can take many forms such as an image and a segmentation mask, an image and a text label, an array of tokens and another different array of tokens, etc. The pairs can be referred to as a data point.

The typical training procedure performs the following steps iteratively. Several data points are loaded into main memory, preprocessed as required and assembled into a "mini-batch", or simply a *batch*. The *batch size* is the number of data points constituting a batch, and is a form of parallel processing of inputs. The sample portion of the batch is then loaded into the accelerator memory, and the model ingests it, generating an output. This is referred to as the *forward pass*. The output, containing a model-predicted label for each sample, is compared to the true labels, and the model's *loss function* is used to compute a measure of the model's error, the *loss*. The gradient of the loss with respect to each learnable parameter of the model is then computed, in what is referred to as the *backward pass* or *back propagation*, and the learnable parameters are finally updated by stepping in the opposite direction of the gradient. In DL frameworks, the parameter update is usually performed by an optimizer object, usually implementing a variant of the Stochastic Gradient Descent (SGD) algorithm [7]. This sequence of operations constitutes one training *step*.

Model training occurs over many such steps. The number of steps required to iterate over the entire dataset depends on the batch size and the size of the dataset. One such pass over the dataset is called an *epoch*, and training often occurs over multiple epochs. DL training makes frequent use of randomness at many levels to avoid *overfitting*, a condition where the model's performance is too closely linked to its training dataset and does not generalize well to unseen data [23]. In data

loading, randomness is used to change the order the data points are loaded between epochs, their assemblage into batches, or to randomly modify samples, e.g. by cropping and flipping an image, to provide the model with never previously seen data, a process called *data augmentation*.

In order to objectively assess the model's performance, training will periodically stop and a separate, *held-out* dataset, called the test or validation set (though the validation set can also refer to a third dataset used in a similar way during a regular training step) will be loaded and fed to the model. This allows an estimation of the model's capacity to perform on previously unseen data, which is the objective for real-world applications. This phase of training is called *evaluation*, or *inference* when referring to models deployed to production. During evaluation, the model only performs the forward pass, as it should not learn from the test data. After an evaluation, it is common to perform *checkpointing*, where the model's parameters are written to persistent storage, saving the model state at that time. Checkpointing allows the model to be reloaded from a precise state and shared. The size of the checkpoint file will depend in part on the number of model parameters, and the exact file format and organization will depend on the framework used.

From the point of view of the CPU, the computations occurring on the accelerators are asynchronous, and it is available to perform other tasks, such as loading and pre-processing the next batch of data. Ideally, since accelerator time is expensive, a training workload should never leave the accelerator idle, waiting for the next batch of data. A training workload that is able to keep its accelerators fully utilized is called *compute-bound*, while one where the accelerators are left idle, waiting for data, is called *I/O-bound*.

Today, most large training workloads run in a *distributed* fashion, where multiple accelerators on a single node, or multiple nodes each with accelerators, work in parallel to accelerate training speed. Distributed training can be done in two main ways.

In the *data-parallel* mode, each accelerator hosts a replica of the model while the data is sharded and distributed between them. Since the replicas see different data, they each have their own loss and their own parameter gradient. To keep the replicas in sync, an *all-reduce* operation is performed to compute the average gradient, with is then broadcast to the replicas. This way the replicas apply the same update to their parameters and remain in the same state. In this context, the notion of a global step and global batch size emerge, which are one update of the parameters across all replicas, and the total amount of training data that was computed upon to do so.

In the *model-parallel* mode, a single model is split and distributed across multiple devices. This is usually done because the model cannot fit into the memory available on a single accelerator. Usually, the model will be split by layer, and these will be distributed on the accelerators. model-parallel training implies more communication, as the output of a layer on one device must be sent to another hosting the next layer. Depending on the complexity of its architecture and its size, a model can make use of both data and model-parallel techniques, distributing some of its parameters and replicating others [37].

From the point of view of I/O, a training job is mostly read intensive, as the model iterates over its training set. The read intensity will depend on the size of the dataset and samples, the batch size, the number of accelerators and the computation time per batch. In terms of writing, checkpointing will usually be the most intensive operation and its intensity will mostly depend on the model size, i.e. the number of parameters. The specific I/O patterns will then depend on the organization of the dataset in files, the file format, the size of the dataset relative to main memory, the number of epochs, and configuration and implementation details of the data loader used.

In distributed training, network communication can also become the bottleneck. Indeed, as the number of learned parameters increase, so does the number of gradients that need to be synchronized between model replicas, and thus the amount of data that must be sent over the network before every parameter update. In the model-parallel case, additional networking overhead can also be involved, depending on architectural details. This communication overhead can likely impact the I/O patterns of workloads, though it is out of the scope of this work.

2.4 The role of benchmarks

The field of Computer Science is accompanied by a rich history of benchmarks, sets of standardized tasks that allow computer systems, algorithms, protocols, hardware, etc. to be evaluated in "apples-to-apples" comparisons and highlight their design trade offs [43]. Just about any part or aspect of computer systems have associated benchmarks. For example, the LINPACK benchmarks used to measure the capacity of a computer system to perform floating point operations, and is the basis of the TOP500 supercomputer ranking [2, 24].

In the realm of storage and file systems, benchmarks play a big role in quantifying system performance while subject to different workloads. The main parameters of a workload are the relative frequencies of read and write operations, their sequencing in time, their sizes and their access pattern, i.e. if they access data sequentially or randomly, the latter usually incurring a larger I/O latency. The metrics reported are typically throughput, measured in IOPS (Input/Output Operations per Second) or B/s, and read/write latency.

Benchmarks in this realm can be categorized in two main categories: trace replayers and synthetic workloads. Trace replayers work by recording a workload's activity at a certain layer (e.g. at the syscall or device driver layer) and replaying the exact scenarios at a later date to test a system [3, 13, 38]. The advantages of trace replayers include the realism of their workloads, almost by definition since they are based on real ones, though you could record an artifical workload just as easily, and their application to debugging work as well. Their disadvantages include the usually large file sizes of traces, and often the difficulty in modifying the recorded traces.

On the other hand, synthetic workloads are purely generated from a workload model, which can range in complexity. Synthetic workloads tend to offer a wide range of user-tunable parameters, usually input through configuration files [11]. They can be very useful at large scales where replaying e.g. a datacenter-wide trace is inpractical or even impossible [20]. A good synthetic workload benchmark will have been created based on real recorded traces that have been analyzed and characterized to extract the defining parameters of workloads. However, they have still been criticized as non-representative of real workloads in many cases.

2.5 Benchmarking in Deep Learning

Benchmarks also play a very important role in Deep Learning, as all new state-of-the-art achieving models are determined based on their accuracy on various benchmarks of tasks [62, 70]. In the world of DL model accuracy, benchmarks consists of a dataset (usually already split into training and evaluation sets) and set of prediction tasks on which the model's accuracy can be determined.

However, model accuracy is not the only aspect of DL models that we would like to evaluate. Instead, the training process itself is of great interest to researchers and practitioners, mostly due to its resource intensity and potentially long running time (it is not unusually to train a large model for weeks continually). As such, training speed is an important optimization target.

To this end, benchmarks like Fathom [6] and Baidu DeepBench [61] decide to focus on timing the low-level operations performed during DL training and inference for a set of reference workloads. The use of reference workloads ensures some aspect of real-life relevance (at least, around the publication date of these workloads), however the focus at the operation level means it's hard to extrapolate to total run-time, and makes these benchmarks more relevant to hardware designers than DL practitioners.

Other benchmarks go one layer of abstraction above, and look at the time taken to process a batch of data [64], breaking down this time by model layer [16]. However, again this design does not allow a projection to overall training time, nor does it consider the final model accuracy, often the most important metric for training.

To address these shortcomings, DAWNBench [18] explicitly focuses on end-to-end training time to a set accuracy, which represent a more realistic training scenario. Within this framework, users can implement any combination of optimizations, at the hardware or software level and compare it to others while ensuring the final model achieves the desired result. The accuracies are set to state-of-the-art levels and submissions must include model source code and hardware configuration for reproducibility.

DAWNBench led to the creation of MLPerf (now MLCommons [4]) a non-profit organization aiming to accelerate progress in Machine Learning by releasing industry-standard benchmarks, datasets and best practices in an open and collaborative fashion. The DAWNBench benchmark was then superseded by the MLPerf Training [46] and Inference [60] benchmarks.

The above benchmarks are all concerned with the computing performance aspect of DL. However, much less attention has been given to characterizing the data loading and storage behaviour of DL workloads. It can be argued that DAWNBench and MLPerf Training benchmarks can be used to study these since they include data loading as part of the system under test, and could be used to evaluate data loading optimizations on end-to-end training performance. However, it would be more convenient for storage system designers to isolate the storage component of these workloads by

providing relevant metrics, and be able to test systems without requiring the expensive accelerators to actually train the models to a given accuracy.

The only work we are aware of that explores this is DLIO [21] which serves as the main inspiration and starting point for this work. DLIO is a synthetic I/O benchmark for scientific DL applications running in High Performance Computing (HPC) clusters. The authors characterized a set of DL training workloads and allow users to apply a realistic load on the storage system under test without requiring accelerators, by executing the data loading code, and emulating the model computation with a CPU `sleep()`.

In collaboration with MLCommons and some of the DLIO developers, this work extends DLIO to three new and more industry relevant DL training workloads taken from the MLPerf Training benchmark leading to the release of the MLPerf Storage benchmark.

3

Contributions

Figure 3.1 shows the typical data path in a DL training workload. First, the data is loaded from disk into main memory by the data loader. In a second step, some workload-specific online pre-processing takes place using the CPU, such as resizing an image or tokenizing a string and the individual data points are assembled into batches, to be sent to the accelerators and processed. Finally, the batches are loaded into the accelerators' memory and ingested by the model.

The benchmark we propose will execute the data loading and some online preprocessing using the workload's implementation framework and configuration, thus producing load on the storage system. Loading of the data to the accelerator and model computation will be simulated using a simple `sleep()` executing on the CPU. The validity of this method in recreating realistic I/O patterns was demonstrated in [21] for Tensorflow workloads, and is expected to hold for our workloads as well. To be certain, we replace the computational section of UNET3D and DLRM with a sleep and measure the resulting I/O intensity, described in detail in chapter 4. Using the real framework

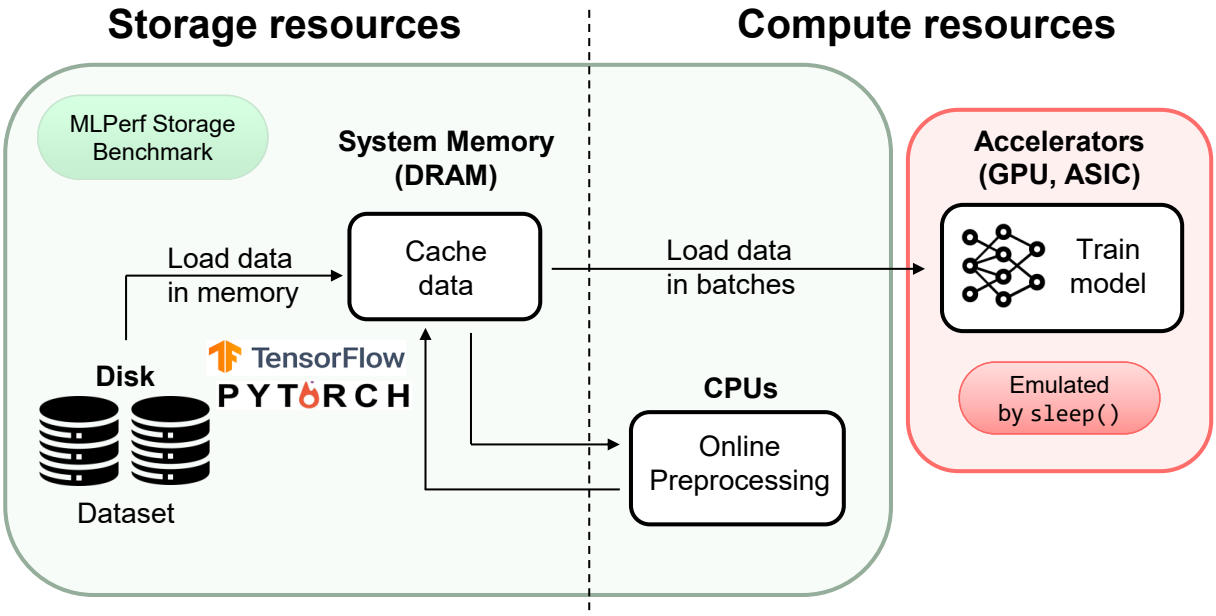

Figure 3.1: The typical data path in a Deep Learning training workload. Data loading and online pre-processing are performed using the ML frameworks. Model training is emulated with a CPU `sleep()`.

data loaders allows us to capture any I/O behaviour stemming from their implementation detail. A future separate benchmark will study the online (and offline) preprocessing in more depth.

Simulating the accelerator computation with a `sleep()` relinquishes control to other execution threads on the machine and allows the framework to perform other tasks, like it would if executing on an accelerator. This allows the benchmark to be run on a machine without accelerators. Different accelerator models can be emulated using different sleep times. Additionally, to emulate distributed training, the benchmark will launch one process per simulated accelerator, each requesting their shard of the data. The load on the storage can thus be increased in two main ways: by decreasing the sleep time, and by increasing the number of simulated accelerators.

For datasets, it was important to us that the benchmark be able to generate them from scratch and scale them up to arbitrary sizes. Discussions with industry professionals [1] revealed that downloading and preprocessing original datasets for workloads was often a difficult and time consuming process which made testing e.g. a new cluster complicated. Additionally, some of the workloads' original datasets are quite small, as we'll see later, and would not be taxing for any realistic pro-

[1] This issue was raised during an MLCommons Storage weekly meeting.

duction systems. Therefore, we wanted the benchmark to offer a smooth experience in this regard, with users simply declaring their desired dataset parameters (i.e. size, number of files, etc.) and the benchmark taking care of the rest. Generating the dataset synthetically makes the benchmark as lightweight as possible, since it does not require any sort of seed data.

Due to the intertwined nature of computation and storage in DL workloads, the goal was not to release a simple storage benchmark, but a tool able to capture this dependency. To this end, the benchmark's reported metric is the model's computational throughput in samples/second, subject to a minimum accelerator utilization (AU) of 90%, with AU defined as:

$$Accelerator\ Utilization = \frac{Ideal\ Compute\ Time}{Total\ Run\ Time} * 100 \tag{3.1}$$

where the ideal compute time represents the theoretically minimal workload run time, and is defined using known workload parameters as:

$$Ideal\ Compute\ Time = \frac{Number\ of\ Samples}{Batch\ Size * Number\ of\ accelerators} * Sleep\ Time \tag{3.2}$$

Note that the sleep time used is dependent on the exact model of accelerator used and their configuration. In this work, we report values for NVIDIA V100's with 32GB of memory in an NVIDIA DGX-1 system, but accurately emulating other accelerators will require its own set of empirical measurements. The MLPerf Storage working group is working with the Training working group to incorporate the logging necessary to collect this data in the MLPerf Training benchmark.

4

Methodology

4.1 Workloads Under Consideration

The three workloads we investigated are UNET3D for 3D image segmentation, BERT for Natural Language Processing tasks and DLRM for recommendations. All were obtained from the MLPerf v2.1 Training Benchmark reference implementations [46]. They were selected for their high-level differences in dataset size, format and composition, the expected number of training epochs, as well as the implementation framework. These high-level differences are documented in Table 4.1. We hypothesized that they would be reflected in their I/O behaviour.

Workload	UNET3D	BERT	DLRM
Training Set Size	29 GB	365 GB	670 GB
Number of files	168	500	1
Samples/file	1	~313k	~4.2B
Format	`.npy`	`TFRecord`	Binary
Epochs	Multiple	Single	Single
Framework	PyTorch	TensorFlow	PyTorch

Table 4.1: High-level dataset and implementation characteristics of each workload under consideration.

4.1.1 UNET3D

The UNET3D workload trains a 3D U-NET model [17] to perform 3D segmentation on medical images. The specific model used is based on the "No New-Net" architecture [36] and trains on the KiTS19 [31] kidney tumor dataset.

The dataset is about 29GB in size and composed of 210 cases: a CT scan of kidneys with an associated 3D mask of tumors, together forming a sample-label pair. The training set is composed of 168 cases, with the other 42 reserved for the test set. The cases do not all have the same dimensions, though an associated CT scan and mask do and we will study this further later on. The files are stored as `.npy` formatted arrays, with the CT scans being of 4B floating type, while the masks are of 2B integer type. Their sizes on disk approximately follow normal distributions with a mean and standard deviation of 117 MB and 54 MB for the CT scans, 29 MB and 13 MB for the masks.

The MLPerf Training reference implementation was developed using the PyTorch framework, and is parameterized to train until convergence to a set target accuracy or for a maximum of 4000 epochs. After 1000 epochs, it starts performing periodic evaluations on a held-out test set, every 20 epochs to determine its accuracy. Finally, it writes its parameters to a checkpoint once the training run is complete, writing a file of around 500 MB.

PyTorch's `DistributedDataParallel` library (DDP) [44] is used to distribute the workload in a data-parallel fashion. Additionally, the implementation uses separate processes to load data, so-called data loading workers or dataloaders, and to train the model. Each training process is associated to a GPU. Thus, for a given number of GPUs, $\#GPUs * (1 + \#Dataloaders)$ processes are launched, with each training process spawning $\#Dataloaders$ workers.

In terms of pre-processing, every sample undergoes a random crop, random flip, data type casting, random brightness augmentation and random noise addition before being assembled in a batch. These transformations serve as a data augmentation mechanism and are all executed on the CPUs by the data loading workers.

The evaluation procedure of UNET3D performs a sliding-window operation instead of the training forward pass and we will later see that it is more computationally intensive. In addition, evaluation uses a hard-coded batch size of 1. The data is still distributed between the accelerators.

Running this workload to completion takes about a dozen hours using all 8 GPUs on our machine. After observing that the I/O patterns were very consistent during the different phases of training, we opted to reduce the number of training epochs in our experiments.

The implementation offers various arguments allowing the user to configure the workload in ways one would expect to impact I/O. For UNET3D these are the (training) batch size, the number of accelerators, and the number of data loading workers. Additionally, the prefetch factor of the dataloader is expected to have an impact, but we left it set to its default value of 2 batches per data loader worker, as it was not proposed as a configuration item in our implementation.

4.1.2 BERT

The second workload we investigated is BERT [22], a Large Language Model (LLM) based on the Transformer architecture [**?**], that can be trained to perform a variety of Natural Language Processing (NLP) tasks, such as question answering or text classification. This training workload is referred to as 'pre-training', because the training process for BERT is split into a first phase where the model learns general representations of text through a first set of tasks, and a 'fine-tuning' phase where the tasks are replaced by "down-stream" application specific tasks. The reference implementation is only concerned with pre-training, which we will consider the same as training.

The implementation comes with a checkpoint file of around 4 GB used to initialize the model before training. This is presumably done to lower the training time to the target accuracy in the benchmark. The workload thus starts from the initial checkpoint, and trains on two tasks: Masked-Language Modeling, where some of a sentence's tokens are randomly masked, with the model attempting to predict them, and Next Sentence Prediction, where the model predicts if a pair of sentences follow each other in the original text.

The dataset is generated from a 2020-01-01 Wikipedia dump and consists of 500 TFRecord [49] formatted files, totalling around 365 GB in size. Each file contains an average of about 313,000 samples with a serialized size of 2825 B. The binary format gets decoded into a collection of 6 arrays and an integer, representing a pair of sentences and associated data necessary to perform the two training tasks. The lengths of these arrays is configurable and depend on the maximum sequence length and maximum number of predictions per sequence parameters. These values were set to 512 and 76, respectively. Thus, each sample is composed of 3 arrays of 512 8B integer values, 2 arrays of 76 8B integer values, 1 array of 76 4B floating point values, and 1 8B integer, for a total in-memory size of 13,816 B. All 8B integer values get cast to 4B integer (to support training on Google's TPUs which do not support the 8B floating point type), and the in-memory sample size is reduced to 7,104 B.

The evaluation procedure of BERT has to be run as a separate invocation of the program. It uses a held-out test set of 10,000 samples, which is separated during the initial dataset pre-processing from the raw Wikipedia data. During the evaluation phase, the model is loaded from the latest training checkpoint and performs a given number of steps before stopping and saving the result.

The reference implementation uses the TensorFlow `tf.estimator` API [2] to train and distribute the model and the `tf.data` API to define an input pipeline for the data. In our implementation, the input pipeline first takes the 500 file names and shards them between workers before shuffling their order. It then uses the `tf.data.experimental.parallel_interleave()` method to create multiple (8 by default) `TFRecordDataset`s that will be read from in parallel, in a round-robin fashion. A second shuffling is applied on a buffer of 1000 samples that will be kept full by the pipeline. Finally, the `tf.data.experimental.map_and_batch()` method is used to decode the TFRecords into a Python object and form batches, using 8 parallel threads by default.

The implementation uses a `tf.distribute.MirroredStrategy` to configure data-parallel training, with a single process controlling all accelerators. We opted to change this and use Horovod [63] to distribute training instead. With Horovod, one process is spawned per accelerator, which is in line with the benchmark. Additonally, the original implementation would spend around 5-10 minutes intializing and we found that the initialization time was greatly reduced when using Horovod.

[2] Now deprecated since TensorFlow v2.0

In terms of potentially I/O impacting arguments, the implementation only allows the user to parameterize the batch size and the number of accelerators. We left other potentially I/O impacting factors to their default values, since the implementation did not expose them as arguments. These notably include the `cycle_length` parameter, defining how many files each worker's input pipeline reads from in parallel, the sizes of the shuffle buffers, and the number of CPU threads used to decode and batch the samples. Interestingly, the `prefetch()` transformation is not used in the input pipeline, however the shuffle buffers are thought to fulfill the same functionality, since they are to be kept full at all times.

We were able to reach a maximum per-worker batch size of 6 on our machine, before running out of GPU memory. With this setting and 8 GPUs, it would take us around 30 days to perform an epoch on the available training data. Therefore, all our experiments run for a smaller number of steps and even though the full dataset could theoretically fit in our machine's memory, we never reach this point.

4.1.3 DLRM

The third workload we analyzed was the Deep Learning Recommendation Model [50] (DLRM), a model trained to predict the probability of a click given a user and a piece of content. The model takes as input an array of continuous and categorical features, together representing a user's interaction with the content.

The reference implementation [3] can be used with two datasets. We opted to use the largest one, the Criteo 1TB Click Logs dataset [19]. As its name implies, it has a total size of 1 TB, split across 24 files each representing a full day of logging.

There are two options to preprocess this dataset for training. We opted for the 'binary-loader' option, supposed to be more efficient [4]. This option creates three large files for training, evaluation and validation of sizes 671GB, 14GB and 14GB respectively. The 671GB training file contains around 4.2 billion samples, each composed of 40 4B integer values for a on disk size of 160B.

[3] The MLPerf Training repository linked to the original DLRM from Facebook research available at `https://github.com/facebookresearch/dlrm/tree/6d75c84d834380a365e2f03d4838bee464157516`. Since then, a version 2 of DLRM was released, which is supposed to have a more optimized data pipeline, overlapping loading and computation. (`https://github.com/mlcommons/training/tree/master/recommendation_v2/torchrec_dlrm`)

[4] The other preprocessing option leads to the creation of 24 `.npz` files which are read one by one but are fully loaded in memory before being fed to the model, causing a large initial wait time.

The categorical features of the data are the source of DLRM's key architectural difference versus the other workloads. To be processed by neural networks, the categorical features must first be converted to dense features. This conversion is done through the use of embedding tables, learned by the model during training, which represent each distinct category by a dense vector. The categorical features are then interpreted as an index into this table, and used to retrieve the associated embedding vector. A categorical feature with N distinct categories must be linked to an embedding table with N rows.

In the dataset, there are 26 categorical variables, with a number of categories varying from only 3 to 10 million. The length of the vector embedding of each category (referred to in the code as 'sparse feature size') is configurable and was set to 128. With these values, the total size of the embedding tables is 27GB, which, when considered with the two MLPs, optimizer state and input that are also stored in GPU memory, cannot fit in a single of GPUs 32GB of memory. Note that the checkpoint file of the model is also around 27GB.

Because of this, DLRM employs model-parallelism for its embedding tables, distributing them between the available accelerators. In addition, DLRM also employs two neural networks that process the dense features, and the embeddings of the categorical features. The two neural networks make use of data-parallelism and are replicated to each device. Thus, DLRM makes use of both data and model-parallelism.

This hybrid data and model-parallelism introduces additional inter-device communication requirements. Indeed, for a given batch of data, the continuous features will be sent to a single device in the typical data-parallel fashion. However, the categorical features are distributed to each device according to which embedding table they are related to. The embeddings must then be re-assembled with the dense features on a particular device to be processed by the second neural network using all-to-all communication. To this end, DLRM implements a distribution algorithm termed "butterfly-shuffle" [50].

Interestingly, the binary dataloader was not configured to use either prefetching nor data loading workers (hard-coded to 0). Together this means the data loading will be done synchronously and the accelerator will have to wait for the next batch of data. Additionally, the dataset is not sharded, with only a single dataloader being instantiated. The batches are manually split into dense and categorical features and distributed to the accelerators in the appropriate fashion. Thus, the batch

size for DLRM is global. The default batch size in the reference implementation launch script was 2048 samples, and we could increase it to a maximum of 2M before running out of memory using 8 GPUs. Due to the model-parallelism, using less accelerators means each has less memory available for data, so the maximum batch size depends on the number of accelerators used. We found a batch size of 128k to fit for all numbers of accelerators but the single GPU case, and used this as our maximum value in experiments.

4.2 Tracing the workloads

In order to get a general sense of the I/O patterns of the workloads, we first developed traces using `bpftrace`, a convenient front-end to the BPF Compiler Collection (BCC) toolkit [29]. `bpftrace` allows us to write traces that latch on to various tracepoints and probes within the Linux kernel, and collect statistics in an efficient way. For our purposes, we wanted to capture the interaction of the workloads with the Virtual File System (VFS) and Block I/O (BIO) layers of the kernel.

At the VFS layer, we used tracepoints on the `read()`, `pread64()`, `write()`, `writev()`, `pwrite64()`, `openat()` , `close()`, `mkdir()`, `rmdir()`, `unlink()` system calls, and the associated kernel probes on `vfs_read()`, `vfs_write()`, `vfs_open()`, `filp_close()`, `vfs_unlink()` to resolve additional information (e.g. filenames). Using these, we are able to capture the latency, size, filename and file offset of the operations while the workloads are running. Additionally, we can combine the file offsets with the operation size to infer `lseek()` calls and determine which operations are sequential and which are random.

At the BIO layer, we trace using kernel probes on `blk_account_io_start()`, `blk_mq_start_request()` and `blk_account_io_done()` to capture the latency between issuance and completion of BIO requests, the disk it was issued to, the request type (Read/Write), size and sector. The BIO trace allows us to capture important storage load information that cannot be obtained with DL framework profiling tools, nor tools like `iostat`, though in practice we also ran it in parallel, to validate the BIO trace results. Through the sector and the request size, we can again have a measure of sequentiality or randomness at the block level.

We did not trace the use of `mmap()` as, apart from being difficult to get meaningful tracing data and understand what is read/written, in practice we observed that the workloads interacted with their training data through the above system calls.

Though syscalls at the block level are always the same between the workloads, the VFS operations were observed to vary. This was determined by capturing counts of all syscalls performed, which we used to orient our tracing. For example, the TensorFlow implementation of BERT uses the `pread64()` system call to read its TFRecord-formatted training data instead of the `read()` used by the PyTorch implementations of UNET3D and DLRM. Similarly, DLRM was observed to use `writev()` in tandem with `write()` when checkpointing.

Since `bpftrace` captures system wide by default, it requires filters to reduce the amount of captured data and return only relevant information. We found PID-based filters unusable since the workloads spawn many child processes and `bpftrace` cannot automatically attach to them, unlike `strace`. Hence, we resorted to hard-coded process name filters, which were manually determined by analyzing system-wide traces for reads to files of interest. For the block I/O trace, we omitted the process name filters and instead filtered on the disk, capturing I/Os to `sdb` only, where the data and model output directories were placed. We did this because filtering on process name would miss some writes, which occur asynchronously and in the context of another process (usually a kernel worker thread).

Finally, we capture GPU usage information using `nvidia-smi`, and use the application and MLPerf benchmark logs to obtain other information, such as the start and end times of each phase of training, which we consider to be any of: initialization, training, evaluation or checkpointing. We also make sure to sync all pending writes and flush the page cache before every experiment, to eliminate any caching between different runs of the same workload.

Some post-processing of the traces is necessary in order to create a unified picture of the workload. First, we need to convert `bpftrace`'s 'nanosecond since boot' timestamps to UTC, in order to align them with the other traces which come in a variety of timestamps, at the second or millisecond resolution. To do so, we use an auxiliary BPF trace that triggers on entry into any system call, and simply logs the nanosecond timestamp as well as the local time, which has a resolution of seconds.

During post-processing, we go through this so-called "time-alignment" trace to find every seconds transition in local time, and interpolate the nanosecond timestamp corresponding to a specific

second. Using this alignment, we can then convert all the traces timestamp into UTC, with good precision. Since this auxiliary time alignment trace triggers on any system call, it is quite resource intensive, so we shut it down after 2 minutes, giving us 120 seconds transitions from which to pick the smallest nanosecond timestamp difference for the time alignment [5].

4.3 Workload Instrumentation

In addition to workload tracing, we also instrumented the model code, when possible, to get a breakdown of time spent during a training step, compute various measures of throughputs and quantify how they vary with both batch size and the number of accelerators used.

For workloads implemented using PyTorch, i.e. UNET3D and DLRM, this is easy to do since the data loading and training loops are explicit in the code. This allows us to add `time.perf_counter_ns()` calls and record time spent on each sub-step of a training step. Example code is shown in Listing 4.1. We are able to print out the times for each step this way. We verified that this has no significant impact on the total training time.

On the other hand for BERT, its use of TensorFlow's `tf.estimator` API and its declarative coding style, along with compilation to a graph before running, causes the training loop to be hidden under layers of abstraction, and not directly accessible. However, TensorFlow ships with its own profiler, and we resort to using its data to estimate the breakdown of training steps. The profiler records a single training step and outputs a `.json` file that can be opened in the Chrome Profiler. We export one in every 100 steps, and collect 30 such steps in total for each run.

[5]The traces, post-processing and plotting code can be found at `https://github.com/discslab-dl-bench/tracing_tools` and `https://github.com/discslab-dl-bench/trace_visuals`

```python
class ExampleDataset(torch.utils.data.Dataset):
    ### initialization, etc.
    def __get_item__(self, idx):
        t0 = perf_counter_ns()
        data = ...  # load data from disk
        log(f"from disk latency {perf_counter_ns() - t0}")

        t0 = perf_counter_ns()
        data = ... # Preprocessing the data
        log(f"preprocessing latency {perf_counter_ns() - t0}")
        return data

def main():
    ### Initialize model, optimizer, dataloader, etc.
    for epoch in range(num_epochs):
        t_step = t0 = perf_counter_ns()
        for batch in data_loader:
            data, label = batch
            log(f"data loading latency: {perf_counter_ns() - t0}")

            t_compute = t0 = perf_counter_ns()
            data, label = ... # Move batch to accelerator
            log(f"to accelerator latency: {perf_counter_ns() - t0}")

            t0 = perf_counter_ns()
            output = model(data)
            log(f"forward pass {perf_counter_ns() - t0}")

            t0 = perf_counter_ns()
            loss = loss_fn(output, label)
            log(f"loss calculation {perf_counter_ns() - t0}")

            t0 = perf_counter_ns()
            loss.backward()
            log(f"backward pass {perf_counter_ns() - t0}")

            t0 = perf_counter_ns()
            optmizer.step()
            log(f"parameter update {perf_counter_ns() - t0}")

            log(f"all computation {perf_counter_ns() - t_compute}")
            log(f"step total {perf_counter_ns() - t_step}")

```

Listing 4.1: Example PyTorch code illustrating how the workloads using this framework were instrumented

We exclude data from the first epoch for UNET3D, as well as the first step of each epoch, in order to measure throughputs only for a fully cached dataset, and avoid the first step being potentially slower than the others due to data loader initializing.

We also instrument the benchmark for comparison to the original workloads. For the PyTorch workloads, we instrument as above. For the TensorFlow workload, we have slightly more visibility

into the training loop and are able to measure an application-side data loading value. This might not provide an apples-to-apples comparison however, but unfortunately we could not find a way to use the same profiler in both the real workload and the benchmark due to their use of different major versions of TensorFlow, with different profiler APIs.

4.4 Instrumentation Measures

We focus on a few measures of interest calculated from the instrumentation data. Since the data loading mechanism varies by workload, the measures cannot be taken in exactly the same way, and sometimes cannot be taken at all. The results section will illustrate each workload's data loading mechanism and the specific things we measure.

Overall, we are concerned with the following throughputs, and the associated latencies:

- The VFS throughput, a measure of how many samples/second the VFS can provide the data loading processes. This is a blocking call for the calling process, where we take the measurement. The VFS latency will depend on the underlying storage and caching effects, as well as the way the application breaks up the call into lower-level VFS operations, something that changes based on the file format, and is defined by the file format libraries. We could have measured something similar from the VFS read trace, however due to the breaking up of application level calls into multiple VFS level operations, it tends to be always constant and less interesting, being fully defined only by the FS and underlying storage systems used. For UNET3D, we measure this for the fully cached dataset, and we would expect it to be reduced if the reads had to go to storage.

- The Data throughput, a measure of how many samples per second the GPU-bound process is able to fetch from the data loading process. This can be either asynchronous for UNET3D and BERT, or synchronous for DLRM which does not make use of parallel data loading workers. In the asynchronous case, we expect this throughput to be very high, as long as the computation time is larger than the time needed to assemble and preprocess a batch.

- The Compute throughput, giving a measure of how many samples per second can be processed by the model during training. This will serve as an upper bound on the demand for data put on the storage by the training process. This throughput is directly related to model training speed.

4.5 Synthetic data experiments

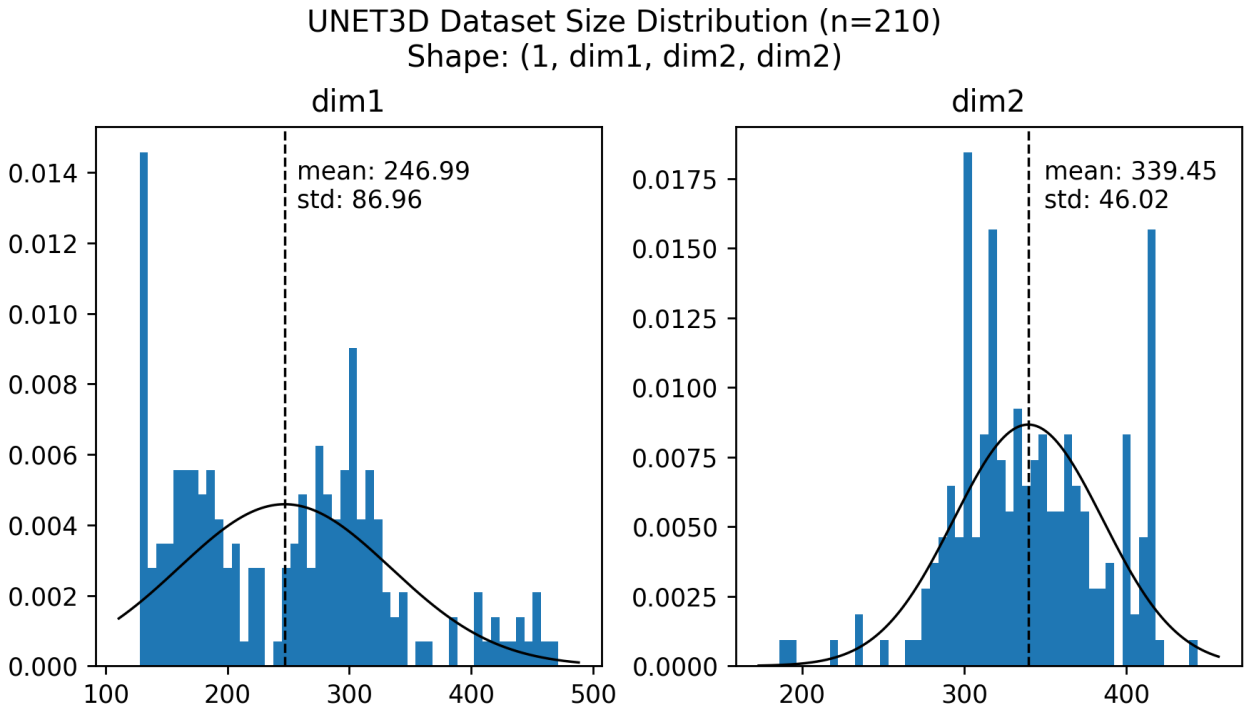

Figure 4.1: Distribution of UNET3D's dataset dimensions. We approximate both with normal distributions and generate the synthetic data by sampling uniformly random values within the observed ranges.

As stated previously, downloading and preprocessing the original datasets in a timely manner can be difficult even for industry professionals, as well as computationally demanding. To address this, we wanted the benchmark to be able to generate its own dataset from scratch, with the ability to scale up to an arbitrary size. However, it was important that the synthetically generated dataset be representative of the real dataset.

Figure 4.1 shows the size distribution of UNET3D's dataset. Each sample has a shape of $(1, dim1, dim2, dim2)$ with the first dimension being the channel, the second the number of slices

in the CT scan, and the last two (always equal) being the dimensions of the slices. We see than $dim1$ is approximately bi-modal with two peaks around 175 and 300, while $dim2$ appears to follow a normal distribution. For simplicity we approximate both with normal distributions in the data generation script.

Recall that each of the 210 cases is composed of a pair of files: the CT scan, and the segmentation mask. The CT scan has 4B floating point values in the interval $(-2.340702, 2.639792)$, while the masks are binary and represented with 1B unsigned integers. We generate the synthetic data by uniformly sampling form the given range for the CT scan, and from $(0, 1)$ for the mask.

Generating synthetic datasets for BERT and DLRM is relatively simpler, but care must still be taken for the data to be accepted by the model. For BERT, we generate the 6 arrays described in subsection 4.1.2 with the correct dimensions and data types, and values within the correct ranges. For example, the first array contains numeric IDs for each token in a sentence, and must refer to a word in the vocabulary file used by the model. We then serialize the samples to the TFRecord format, using the TensorFlow facilities provided for this purpose. Similarly, for DLRM, we follow a similar process, again making sure the randomly generated values for the numeric and categorical variables are within the appropriate ranges. Specifically, the categorical features each have a defined number of possible categories, and changing these will affect the size of the embedding tables.

4.6 Similarity Metric

In addition to the measures described above, we also define a similarity metric that will use the eBPF trace data to compare workloads. Contrary to the instrumentation data, the eBPF data is captured below the application-level, allowing us to capture the I/O behaviour from the system's point of view. To compute this metric, we first define a characteristic vector, whose components reflect the I/O patterns of a workload.

We wanted the metric components to be hardware-independent, thus capturing information relevant only to the workload itself, and allowing comparisons across hardware if necessary. We focus on metrics such as the number of read and write operations, their sizes and size distributions, the number of unique files accessed, the sequentiality or randomness of accesses. We do have a component that is linked to hardware, the ratio of amount BIO read to amount VFS read that

serves as a proxy measure of caching. It will thus depend on the relative sizes of the memory and dataset, but not on the storage system characteristics. See Listing A.1 for the algorithm used to count random and sequential reads from the BIO trace.

Component	Weight
Number VFS reads	2
Amount VFS read (B)	2
Number unique files read	2
1st quartile VFS read size (B)	2
Median VFS read size (B)	2
3rd quartile VFS read size (B)	2
Number BIO reads	2
Amount BIO read (B)	2
Random BIO reads (%)	2
Sequential BIO reads (%)	2
Amount BIO read / Amount VFS read (%)	2
Number VFS writes	1
Amount VFS written (B)	1
Number unique files written to	1
1st quartile VFS write size (B)	1
Median VFS write size (B)	1
3rd quartile VFS write size (B)	1
Number BIO writes	1
Amount BIO written (B)	1

Table 4.2: Metrics and weights used for the cosine similarity of workloads.

With the characteristic vectors of two workloads, detailed in Table 4.2, we use a weighted cosine similarity to compare them. Considering the training process is mostly read-intensive, we assigned a weight of 2 to all read-related components, and a weight of 1 to the write-related ones. Additonally, we normalize each component's values to the max of the two traces. Thus, for characteristic vectors u, v and weight vector w, we have:

$$Weighted\,Cosine\,Similarity = \frac{\sum_i w_i \bar{u_i} \bar{v_i}}{\sqrt{\sum_i w_i \bar{u_i}^2}\sqrt{\sum_i w_i \bar{v_i}^2}} \tag{4.1}$$

where

$$\bar{u_i} = \frac{u_i}{max(u_i, v_i)} \tag{4.2}$$

In order to reduce the noise in the traces and focus on the more relevant aspects of the traces, we filter the reads to include only those made to data files, and only writes to checkpoint files. Without this filtering, a dissimilarity cost might be paid due to differences between the workloads that are irrelevant to us. Specifically, differences in the source code and library files read by the processes, differences in logging behaviour, and in ways of doing inter-process communication.

When developing the similarity metric, we attempted two other ways to compute it: a "block-wise" and "block-by-block" version. In the block-wise version, the traces are first separated into different phases of training (training, evaluation and checkpointing) based on the timestamps from the application log. Then, the similarity is computed for each separately. We thought that this would provide more insight as each phase of training has their own I/O characteristics. In the block-by-block version, we wanted to capture the time sequence of the training phases, and computed similarities between pairs of blocks, taken one at a time from each trace. Each timestamp-delimited phase represents a "block". If the traces did not have the same number of blocks, a penalty of 0 was assigned for every extra block.

In practice, we had some difficulty with both. For the block-wise metric, the asynchronous nature of the BIO writes led to their appearance in phases other than checkpointing which ended up hurting both the training and checkpoint phases' similarity. Future work could attempt to simply allocate all writes to checkpointing wherever they appear. The block-by-block metric was computationally intensive, and led to huge penalties when comparing workloads with different numbers of epochs.

4.7 Benchmark methodology validation

In order to verify that the benchmark methodology is valid with respect to I/O, we had to verify two things. The first is that using purely synthetic data does not impact the I/O patterns of the workloads, and the second is that a CPU sleep does accurately emulate accelerator computation.

To validate the synthetic data, we analyzed the constitution of each dataset and generated purely synthetic ones of the same format, for each workload. We then compared the trace and instrumentation results for runs using the generated and the real data. The data generation procedure was then integrated in the benchmark.

Similarly, to validate the use of a CPU sleep, we replaced the real model's computation with a `sleep` and compared the resulting run with the original using instrumentation measures.

4.8 Experimental Hardware

All our experiments were performed on an NVIDIA DGX-1 system with 8 NVIDIA Tesla V100 32GB GPUs connected with NVLink, 512 GB DDR4 LRDIMM, 2 20-core Intel Xeon E5-2698 2.2GHz CPUs, a 480 GB boot OS SSD, and 4 additional 1.92 TB SSDs in a RAID 0 striped volume of 7.6 TB [53]. The datasets and checkpoint output directories were placed on the RAID volume, while logging was done on the SSD.

5

Results and Analysis

In this section, we present the results of tracing, instrumentation and emulation with the MLPerf Storage Benchmark separately for each workload, along with some analysis and discussion of a workload specific nature.

5.1 UNET3D

5.1.1 How to read the trace visualizations

As Figure 5.1 is the first trace visualization shown, we offer a short explanation of how to interpret the plot. The top row shows GPU usage captured using `nvidia-smi`, with the red plot showing the GPU's processor usage, defined as the "percent of time over the past sample period during which one or more kernels was executing on the GPU.", the orange plot showing the GPU's memory usage defined as "percent of time over the past sample period during which global (device) memory was

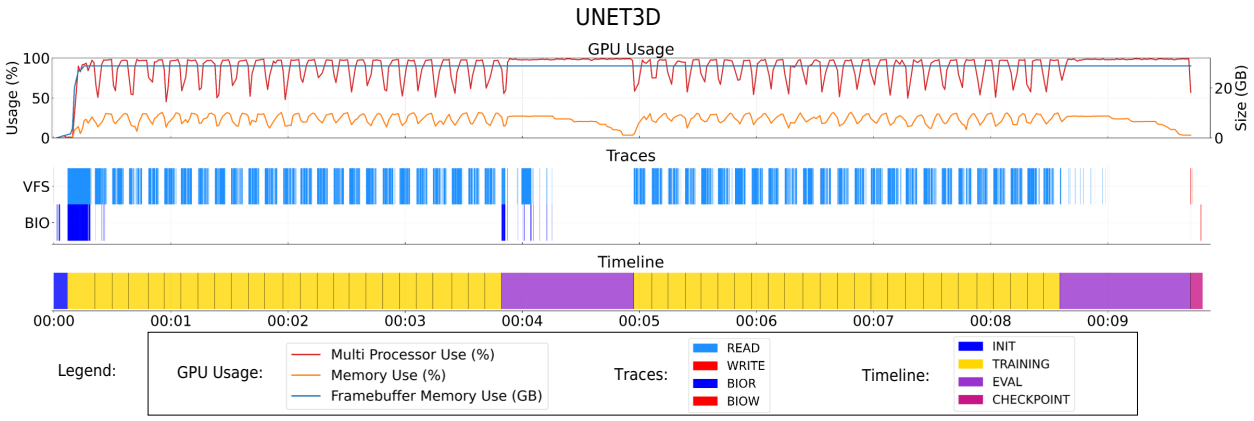

Figure 5.1: Trace visualization of a UNET3D workload run using 8 GPUs, a batch size of 4 and 1 data loading worker per GPU. The model trains for a total of 50 epochs (demarcated by black lines in the timeline), evaluating every 25 and checkpointing at the end of training.

being read or written." [52], and the blue plot showing the amount of framebuffer memory used.

The middle row shows the VFS read/write and BIO read/write activity over time. Light blue and dark blue indicate reads at the VFS and BIO levels respectively, while red indicates writing at both levels. Finally, the bottom row illustrates the phases of training, which can be initialization, training, evaluation and checkpointing. Different epochs are separated by black vertical lines.

5.1.2 Workload high-level traces

Figure 5.1 shows a typical run of UNET3D using 8 GPUs, a per-process batch size of 4 and 1 data loading worker per process. It trains for 50 epochs in total, performing an evaluation every 25 epochs, and a checkpoint at the end of training.

From the GPU activity, we observe periodic drops in GPU computation and memory operations performed occurring at the start of every epoch. This seems to be caused by the dataloaders resetting at that time, and their first batch needing to be fully assembled before computation can begin, as illustrated in Figure 5.4. We also see that the framebuffer memory usage is close to full, increasing quickly during the first epoch and sitting at around 30GB for the rest of the run.

In the traces and timeline now, we see that during the first epoch, VFS-level reads are accompanied by BIO-level reads, but this almost completely stops for the second epoch and onward. Additionally, the first epoch is visibly longer than the others. The next time we observe BIO reads is during the first evaluation, while they are again absent during the second evaluation. This is indicative of caching done by the file system.

Given UNET3D's small size of the dataset (29 GB) relative to the 512 GB of main memory available on our machine, and the training over multiple epochs, we expected to see caching effects in Figure 5.1. Dataset caching leads to faster training, which is illustrated by the first epoch being visibly longer than the others.

We notice that evaluations last about 8 times longer than epochs. This is mainly due to the computation being different and more computationally-intensive than the training step. We explore this in more detail below.

The workload exhibits very little writing activity, which was expected from the nature of DL training. Most of the writes are related to logging, to standard output and to files, which we filter out during post-processing. The next most frequent writes are to shared memory and UNIX sockets, used for inter-process communication and rarely exceeds a few hundred bytes in size. The most significant writing event is the model checkpoint at the end of training.

5.1.3 Instrumentation

Step Breakdown

Figure 5.2 shows the breakdown of a UNET3D training step into its components. The overall step time is dominated by the computation time, with data loading being negligible in comparison. Nonetheless, data loading has highly variable latency, with a range spanning a whole order of magnitude. This is the largest variation exhibited by any single component of the breakdown.

In addition to the general step components listed in Listing 4.1, UNET3D performs a 6th sub-step as part of its computation, here called "Cumulative Loss". This extra step is not strictly required to train the model, and seems to be included for reporting purposes. It consists of performing an all-reduce on the losses from each accelerator-bound process, averaging and storing the value. Interestingly, we see that it is the single longest component of a step.

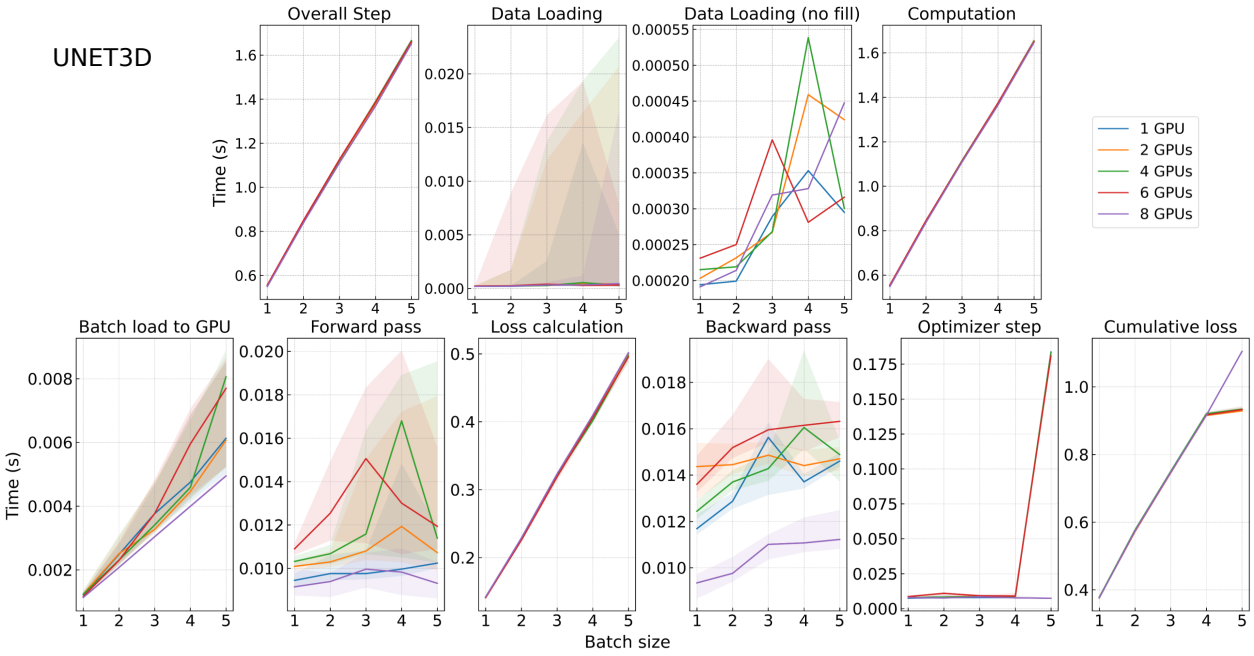

Figure 5.2: UNET3D training step time breakdown. Median values shown, with a fill between the 1st and 3rd quartiles. The top row shows the overall step and a breakdown in its data loading and computation components. The bottom row shows a further breakdown of the computation into its sub-components. Data aggregated from 3 independent sets of runs using 1 data loading worker.

Attempts to remove this 6th step do not yield the expected result of making each step faster however. Instead, the step duration remains almost constant, with the time and pattern of the cumulative loss showing up as both the batch load to GPU and forward pass times. See Appendix Figure A.1 for the breakdown. We mention it here to show that behind the seemingly simple PyTorch training loop, a lot of complexity is hidden and unforeseen interaction effects like this can arise. We did not have time to investigate this strange behaviour, and finding the root cause might require digging deep into PyTorch's inner workings.

For each combination of GPUs and batch size, we fit the distribution of computation times with a normal distribution and derive the linear model in equations 5.1 and 5.2 for their mean and standard deviation with R^2 of 0.99 and 0.63, respectively. Appendix Figure A.2 shows all the computation time distributions used to fit these relationships.

$$Mean = 2.784 * 10^{-3} * NumGPUs + 2.755 * 10^{-1} * BatchSize + 2.722 * 10^{-1} \quad (5.1)$$

$$Std = 7.960 * 10^{-3} * NumGPUs + 1.152 * 10^{-2} * BatchSize - 4.872 * 10^{-2} \quad (5.2)$$

For evaluations, UNET3D always uses a batch size of 1 so there cannot be a relationship between the evaluation time and the batch size. As stated above, the evaluation's computation is a different operation than the forward pass. The breakdown of its duration is shown in Figure 5.3. We see that the sliding window calculation is responsible for almost all of the latency and scales linearly with the image size. To approximate this, we fit a normal distribution to the evaluation times and obtain a mean of 8.403 seconds and standard deviation of 4.442 seconds. The relationship with the number of accelerators is captured by the benchmark distributing the cases between processes.

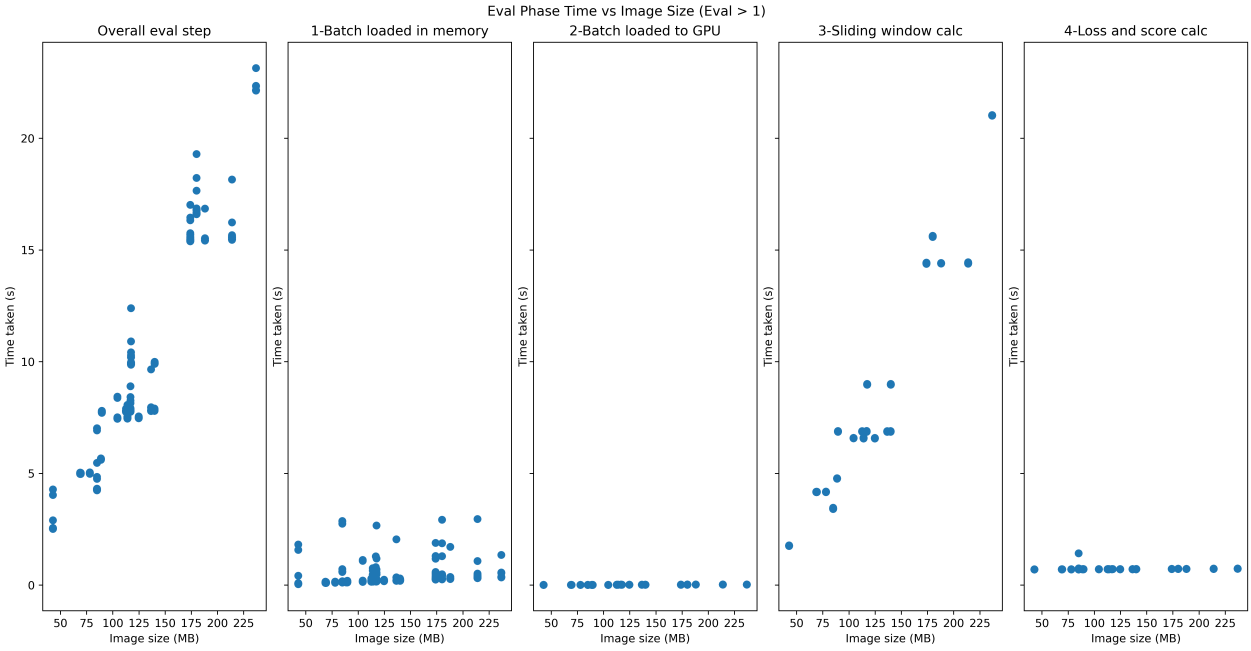

Figure 5.3: UNET3D breakdown of the sub-phases of evaluations, showing the relationship between operation time and image size in MB.

Latencies and Throughputs

Figure 5.4 illustrates how UNET3D's data loading is organized, with its use of separate data loader processes to parallelize data loading and computation. Each accelerator-bound process can have 1 or more data loading workers that take care of loading and pre-processing *batch_size* samples, assembling a batch and passing it to the GPU-bound process. If an accelerator-bound process uses 0 data loading worker, then data loading is done synchronously with computation.

Figure 5.5 and Figure 5.6 show the results of instrumentation for UNET3D. For runs of the original workload, the original using synthetically generated dataset, the original workload with a `sleep()` replacing computation and the MLPerf Storage benchmark emulation. In Figure 5.5, we show the sample preprocessing latency, yellow in the data loading diagram.

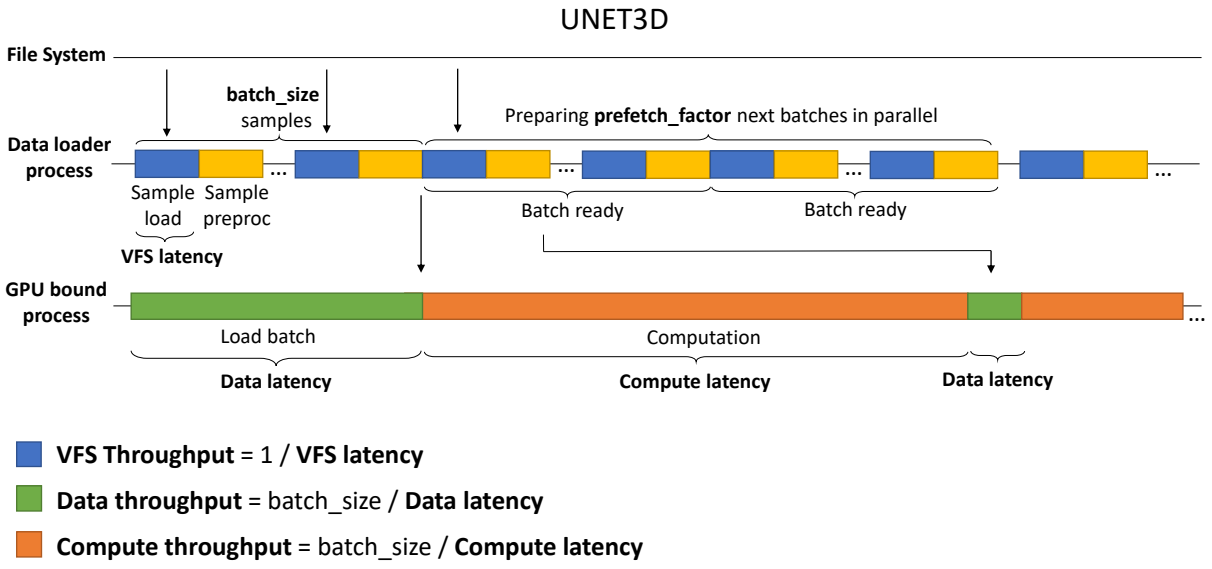

Figure 5.4: Diagram of UNET3D's data loading and measured throughputs. The data loader processes will load and preprocess **batch_size** samples, assembling them into a batch. The GPU-bound process will request a batch from the dataloader and compute on it. The second batch is returned much quicker than the first since it was assembled in parallel while the model was computing. The **prefetch_factor** determines how many batches the data loader processes should assemble in advance.

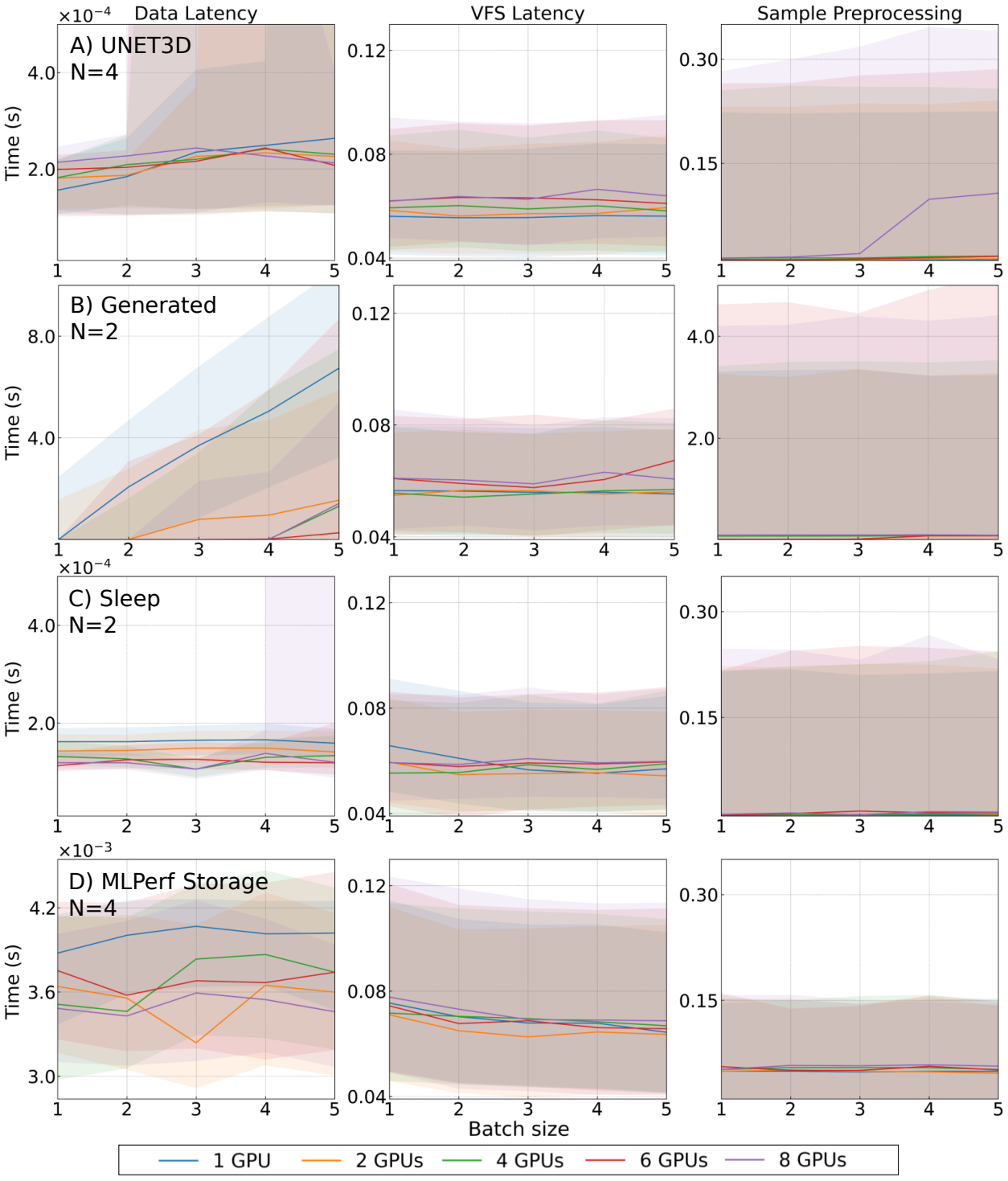

Figure 5.5: Data, VFS and Sample preprocessing latencies measured across batch sizes and number of GPUs for A) the real UNET3D workload, B) the generated data experiments, C) the sleep experiments and D) the MLPerf Storage benchmark. Median values with inter-quartile fill. N denotes independent sets of runs. The full inter-quartile range is not always shown for visibility.

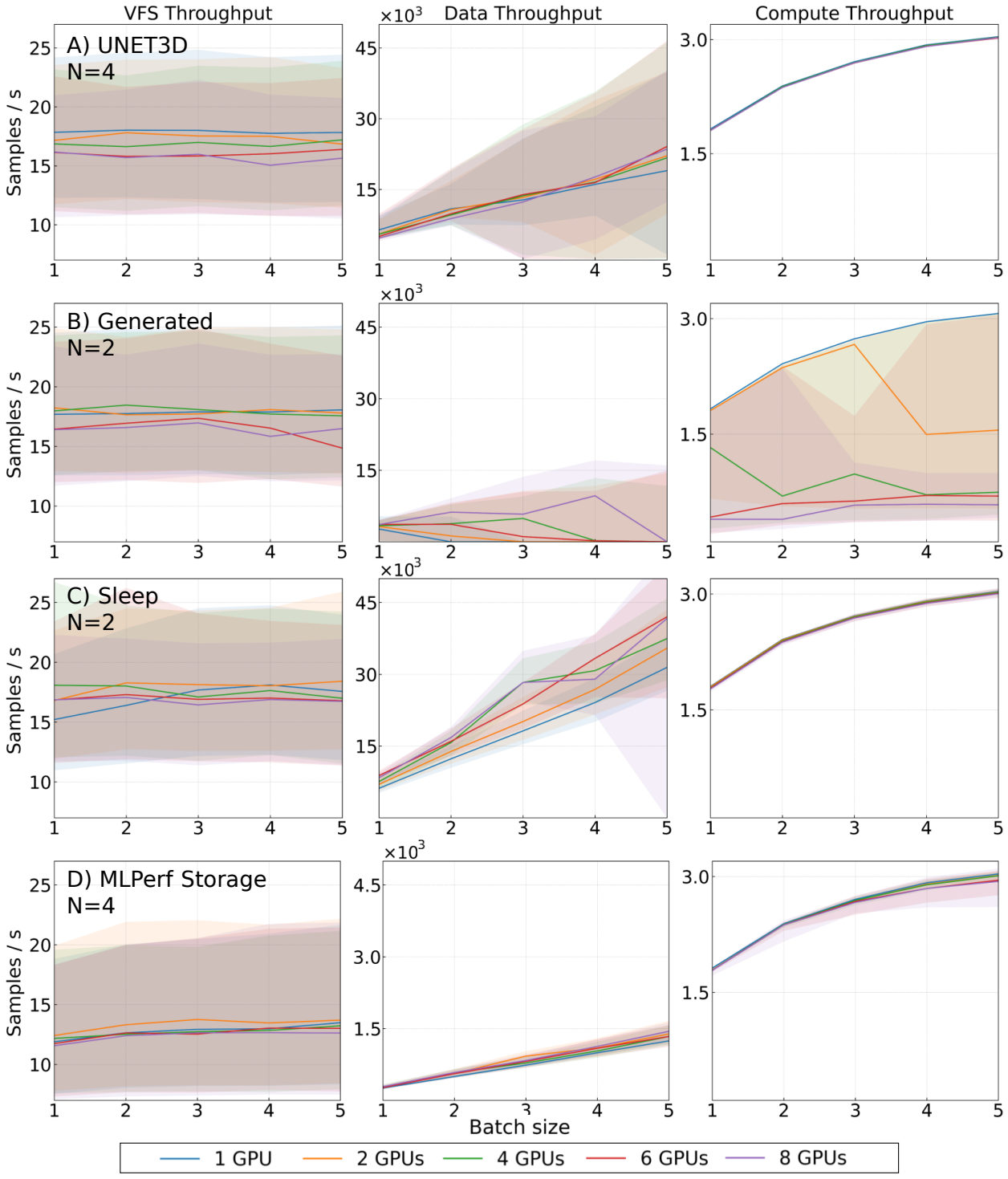

Figure 5.6: VFS, Data and Compute Throughputs measured across batch sizes and number of GPUs for A) the real UNET3D workload, B) the generated data experiments, C) the sleep experiments and D) the MLPerf Storage benchmark. Median values with inter-quartile fill. N denotes independent sets of runs.

Comparing the first and second rows show the effect of training the real model on a synthetically generated dataset. Surprisingly, we observe increases of the median data loading latencies of up to 4 orders of magnitude compared to the original, and corresponding decreases in data throughput. The effect is worse for the single GPU case, and least pronounced for 8 GPUs.

The increase in data latency seems to originate in sample preprocessing, with its quartiles spanning a much grater range when using generated data. We see from the shading that while the original workload's preprocessing times were mostly below 0.3 seconds, they now regularly reach multiple-second latencies. Given that the model computation time is between 0.6 and 1.6 seconds, any sample whose preprocessing time is above these values will cause data loading to become the bottleneck. This is reflected by a decrease in compute throughput shown in Figure 5.6 as accelerators are unable to be kept busy with data.

While this could seem to invalidate the use of synthetic data for the benchmark, it was decided that the benchmark would not perform nor emulate the preprocessing, removing this problem. Instead, we are happy to see that the VFS latency and throughputs are contained within the same ranges, indicating that between the application and the file system, the synthetic data is the same as the real. Appendix Figure A.3 shows that simulating preprocessing in the benchmark by using an extra `sleep()` of the appropriate length results in no significant change in overall data latency. Thus not emulating it in the benchmark is valid. Additionally, a separate benchmark will focus specifically on preprocessing, looking at both online (like we were concerned with here) and offline preprocessing, which occurs outside of training.

The third row in both figures shows the effect of replacing the computation in the original workload with a sleep time derived through equations Equation 5.1 and Equation 5.2. Here we see that we successfully achieve a similar compute throughput as the original. This validates that doing a CPU `sleep()` does not have unforeseen impacts on the training demand for data, though the variation is somewhat increased around the median. The VFS throughputs are also similar, though that is to be expected since we are reading the original dataset. However, the variation in data latency is greatly reduced, with the inter-quartile ranges of all configurations spanning around 0.0001 seconds for sleep, vs around 0.02 seconds for the original. Comparing their median values, we see they are between 1 to 3 times lower than those of the original, resulting in proportionally increased data throughputs.

Finally, the fourth row shows the results for the benchmark implementation. As for the sleep experiment, the data latencies are again less variable, but this time the median values are an order of magnitude higher than for the original, between 0.0028 and 0.0042 seconds for the benchmark vs. between 0.0002 and 0.00055 for the real. Thankfully, the VFS measures are within similar ranges, though the median latency values are slightly higher for the benchmark. Finally, the median compute throughput is very similar between both, with the benchmark showing larger variability.

5.1.4 Benchmark tracing and Similarity metric

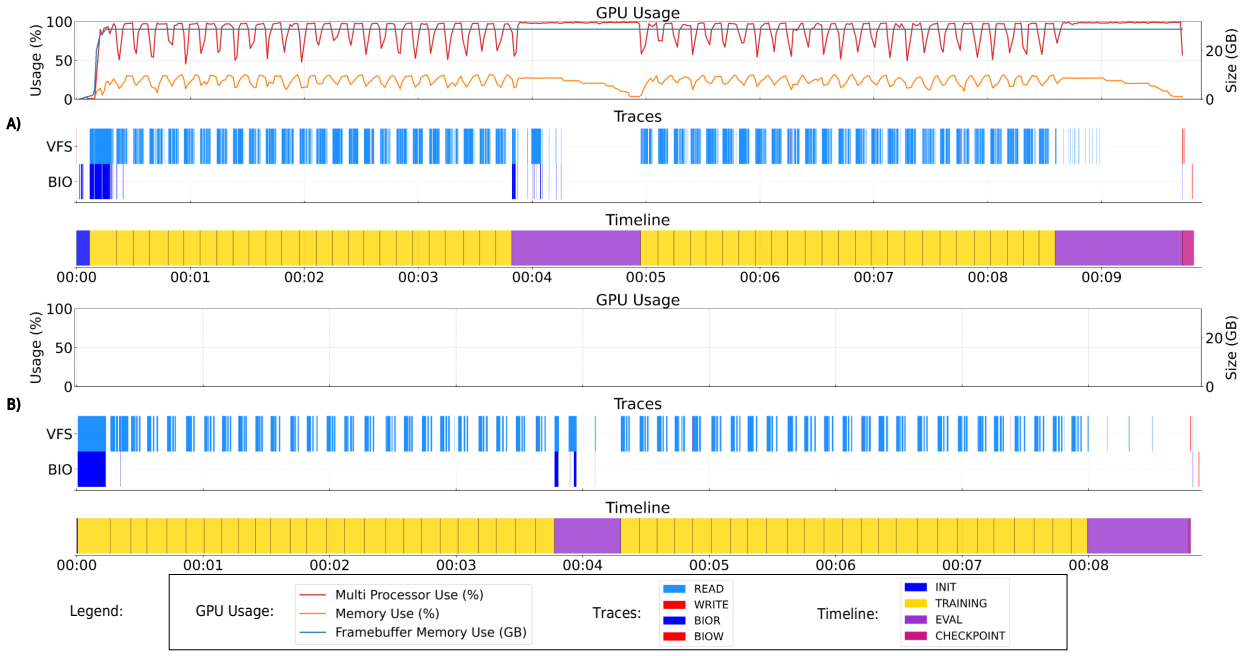

Figure 5.7: Trace visualizations of A) UNET3D and B) the MLPerf Storage Benchmark emulation.

Figure 5.18 shows the traces in Figure 5.1 and the equivalent benchmark run. Of course, there is no GPU activity for the benchmark since it does not perform the model computation. We see that overall, the benchmark closely follows the original workload during training, completing epochs at the same frequency and exhibiting the same caching effects during the first epoch and evaluation. We do note a few differences, notably in the length of evaluations and checkpointing.

The difference in the lengths of the evaluations in Figure 5.18 can be explained by extra reporting-related work done by the original workload at the end of the evaluation, which we did not attempt to emulate in the benchmark as it is not relevant from an I/O perspective. The checkpointing behaviour is more interesting though. We see that it occurs faster in the benchmark, and from the similarity breakdown in Table 5.1 we see that the VFS write distributions are significantly different, while the overall amount of data VFS written is exactly the same.

These workload runs achieve a similarity of 95%, with the breakdown by component shown in Table 5.1. We see that the most different component is the median VFS write size, followed by the percentage of random BIO reads.

Component	Weight	UNET3D	MLPerf Storage	Difference
Number VFS reads	2	80745	80960	0.27 %
Amount of data VFS read (B)	2	1,165,694,431,611	1,258,810,223,064	7.99 %
Number unique files read *	2	420	336	-20.0 %
1st quartile of VFS read size (B)	2	672.0	968.0	44.05 %
median of VFS read size (B)	2	4096.0	4096.0	0.0 %
3rd quartile of VFS read size (B)	2	4096.0	4096.0	0.0 %
Number BIO reads	2	118021	126713	7.36 %
Amount BIO read (B)	2	30,523,412,480	32,640,921,600	6.94 %
Random BIO reads (%)	2	2.57	10.53	309.17 %
Sequential BIO reads (%)	2	97.43	89.47	-8.17 %
Amount BIO read / Amount VFS read (%)	2	2.62	2.59	-0.97 %
Number VFS writes	1	230	220	-4.35 %
Amount of data VFS written (B)	1	499,153,191	499,153,191	0.0 %
Unique Data Files Written	1	1	1	0.0 %
1st quartile VFS write size (B)	1	1280.0	1280.0	0.0 %
Median VFS write size (B)	1	3904.0	110267.5	2724.47 %
3rd quartile VFS write size (B)	1	1769472.0	4084327.5	130.82 %
Number BIO writes	1	2091	1900	-9.13 %
Amount BIO written (B)	1	487,636,992	496,320,512	1.78 %
Similarity	**0.95**			

Table 5.1: Individual components of the similarity measure for UNET3D and the benchmark emulation. * The 20% difference in number of unique files read is due to a name conflict between the cases of the training and evaluation dataset in the benchmark.

We see that the workloads both requested around 1.2 TB of data from the file system, with the benchmark requesting around 8% more. This is in-line with the size of the dataset and the number of epochs, that is we would expect around 50 $epochs * 29\ GB = 1.45\ TB$ of data requested overall (assuming the evaluation set is read in every epoch, which it is not). We see that only around 2.5

% of these VFS reads had to go to storage, and thus the total amount of data requested from the storage is only around 30GB, the approximate size of the dataset.

Overall, most components are within 10% of each other though a few stand out. Among the most differing values, the median and 3rd quartile of VFS write sizes stand out, with the benchmark's value being 27 times and twice higher. The next most differing component is the percent of random BIO reads with the benchmark showing around 3 times as much as the original workload.

Finally, we note that while the number of VFS writes are close, the difference is higher for the number of BIO writes. The fact that the overall amount of BIO written data is also very similar indicates that the benchmark's VFS writes seem to split into a smaller number of larger BIO writes. A keen eye will notice that the total amount of data written at the block level is slightly less than the amount of data written at the VFS level. We chalk this difference up to the asynchronous nature of BIO writes. Since checkpointing occurs at the very end of training, it is likely that the workload and tracing stops before all the writes have been flushed to disk.

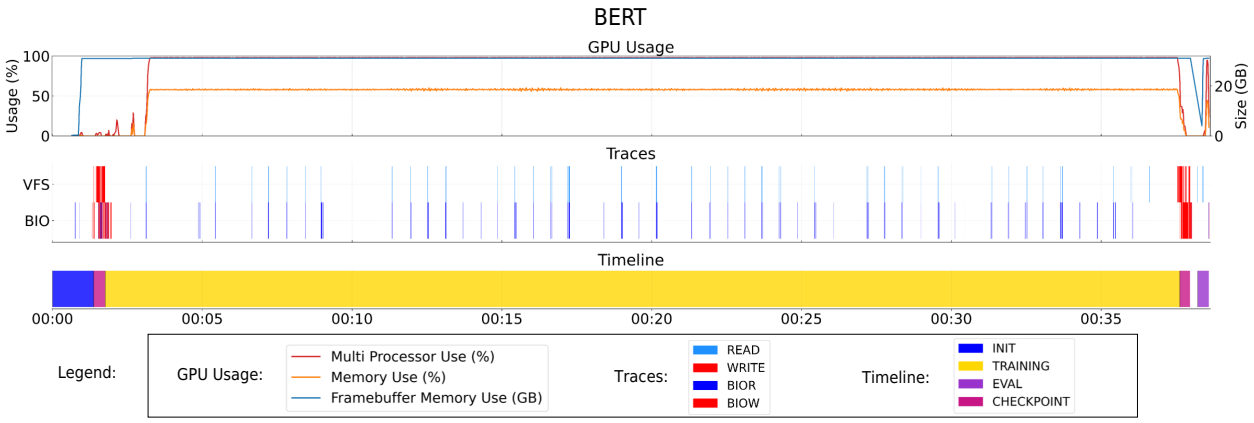

Figure 5.8: Trace visualization of a BERT workload run using 8 GPUs, a per-GPU batch size of 6. The workload trains for a total of 2,400 steps, checkpointing at the start and at the end. Additionally, we run a separate evaluation run of the program for 100 steps at the end.

5.2 BERT

5.2.1 Workload high-level traces

Figure 5.8 shows the traces of a typical BERT run, using 8 GPUs and a per-GPU batch size of 6, the largest our GPUs would allow. Here we see that both the GPU processing and framebuffer memory use are kept at their maximal values for almost the entirety of training, lest an initialization period at the start, though we know the framebuffer use is due to the way TensorFlow [66] manages accelerator memory.

We observe regular reading at the VFS and BIO levels, with seemingly more activity at the BIO-level. That is, we see some BIO reads unaccompanied by VFS-level reads, which can be explained by prefetching. In the raw trace data, we indeed see prefetching occurring. VFS reads trigger additional asynchronous BIO reads to subsequent sectors. Eventually, VFS and BIO reads separate, indicating that data is returned from the page cache. The workload's data demand regularly catches up to prefetching however, and reads overlap again. Interestingly, some VFS calls which should return cached data still show high latencies.

We also see heavy writing activity during checkpointing. Again inspecting the traces reveals interesting behaviour. Even though the model checkpoint file is around 4GB, each worker writes out its own file of this size, before they are merged into a single file. An initial attempt to emulate the checkpointing behaviour left us with very large differences in total amount of bytes written and

hinted the checkpointing behaviour to us. Additionally, visualizing the traces in a disaggregated way, showing the trace visualization for individual PIDs also shows each worker writing significantly during checkpointing.

5.2.2 Instrumentation

Step Breakdown

Figure 5.9 shows the breakdown of a training step for BERT. We see that the overall step time is very much dominated by the computation time, with data loading being negligible in comparison. We observe linear scaling of the computation time with the batch size, and very slight increases with the number of GPUs. Data loading does not seem to show a correlation with either batch size or number of GPUs.

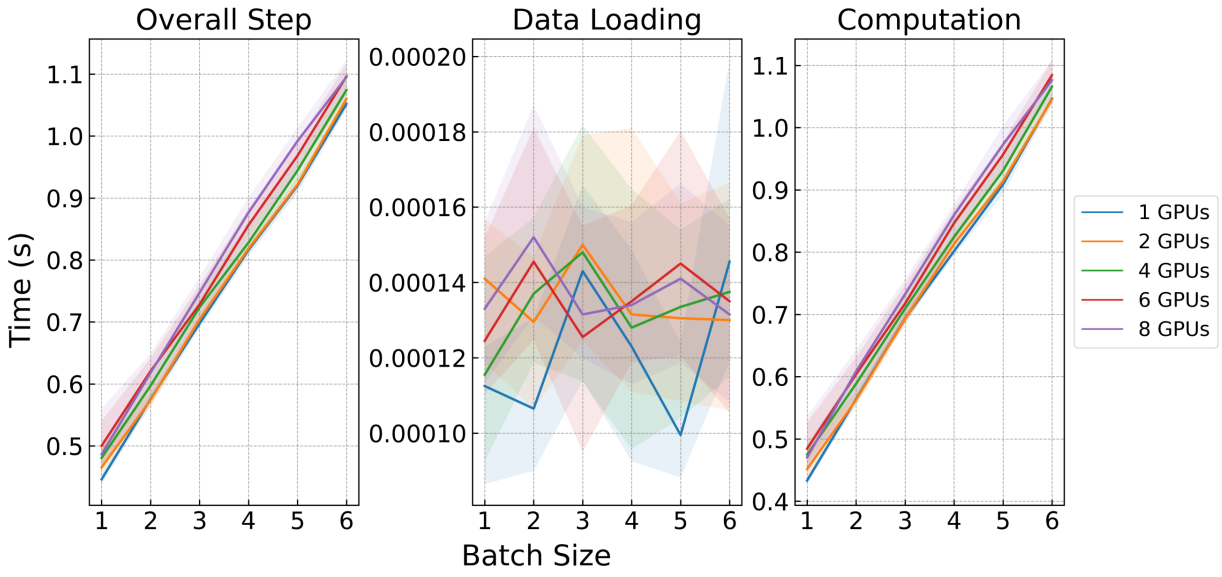

Figure 5.9: BERT training step time breakdown. Median values shown, with a fill between the 1st and 3rd quartiles. Given the difficulty of instrumenting TensorFlow `tf.estimator` code, we do not have a further breakdown of computation like we do for the other workloads.

Based on this data, we fit the following linear regression model between the computation mean time and standard deviation and the number of GPUs and batch size for our machine (R^2 of 0.99 and 0.64).

$$Mean = 6.228 * 10^{-3} * NumGPUs + 1.17 * 10^{-1} * BatchSize - 3.294 * 10^{-1} \quad (5.3)$$

$$Std = 4.484 * 10^{-3} * NumGPUs - 1.283 * 10^{-3} * BatchSize + 9.669 * 10^{-3} \quad (5.4)$$

For this workload, evaluation performs the same computation as the training forward pass, and we opt to approximate it with the same equations. This will be an overestimation since the backward pass and weight update are not performed but we do not expect the difference to be significant in the benchmark, given that it is primarily concerned with emulating training.

Latencies and Throughputs

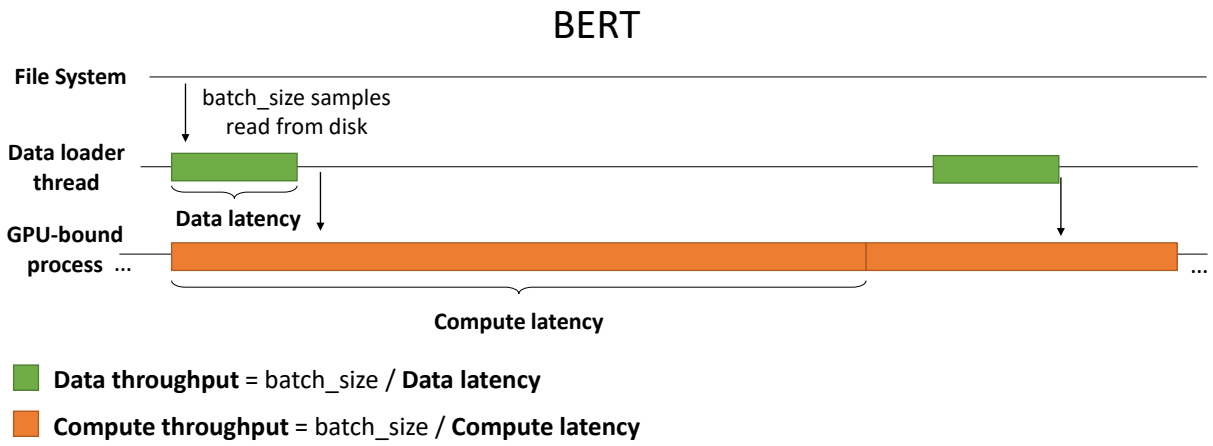

Figure 5.10: Diagram of BERT's estimated data loading, obtained from analyzing TensorFlow profiler traces.

Figure 5.10 shows BERT's estimated data loading process. The data loading mechanism is estimated due to low visibility into Tensorflow's inner workings; we have had to make some assumptions. The Tensorflow profiler traces show the functions executed in CPU and GPU threads. We consider the time between the start of `IteratorV2` and the return of `IteratorGetNext` to be the data latency for BERT. Since in practice the data loading seems to be fully overlapped by computation, we consider the time of the full step to be the computation time. We confirm this by comparing traces where an explicit `prefetch()` operation has been added and observing no difference in latency, throughput or qualitative appearance of the traces. We do not define a synchronous VFS access latency for BERT as it is unclear which events would correspond to it.

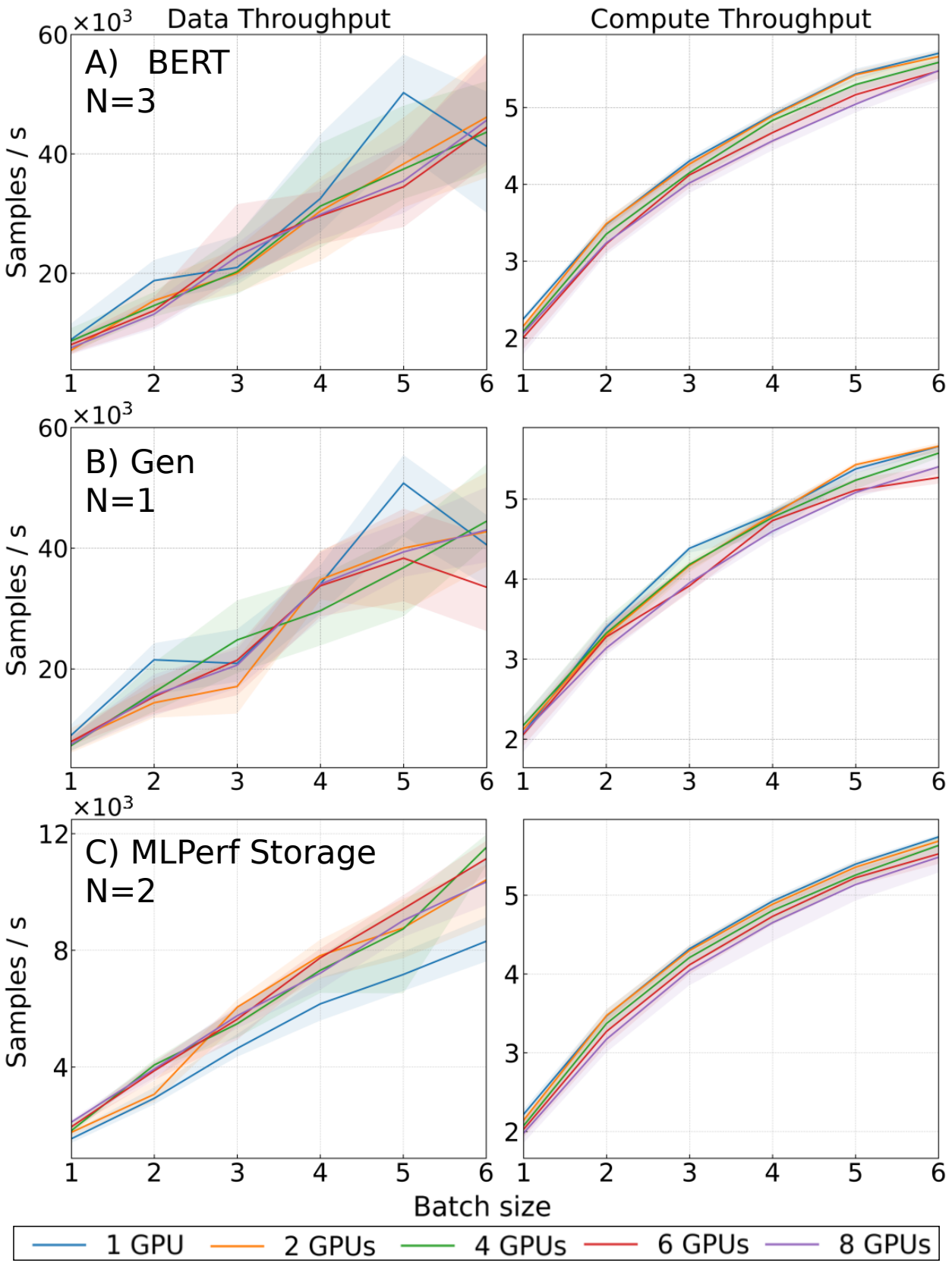

Figure 5.11: Data and Compute Throughputs measured from the TensorFlow Profiler step traces across batch sizes and number of GPUs for A) the real BERT workload, B) the generated data runs and C) the MLPerf Storage benchmark emulation. N denotes independent sets of runs.

Figure 5.11 shows the measured throughputs obtained across various experimental runs. For BERT, since we cannot insert arbitrary statements in the training loop, we are unable to get as detailed a view as for the other workloads, nor run all the same experiments. For example, placing a `sleep()` statement in the model code does not work. The function is removed when TensorFlow compiles the model to a graph before running. Similarly, we do not have access to explicit `read()` calls from the application and are unable to get the VFS throughput for BERT.

Nevertheless, we are still able to generate a synthetic dataset and train the workload using it. The results of this experiment are almost indistinguishable from the original workload. Both are able to obtain data throughputs between 10,000 and 50,000 depending on the batch size, with no plateauing in sight. Their computation throughput similarly show the same kind of curve, starting at slightly above 2 samples per second with a batch size of 1 to around 5.5 with a batch size of 6. The Compute throughput does seem to approach a horizontal asymptote, though we cannot test this using the original workload due to running out of GPU memory.

Looking at the MLPerf Storage benchmark now, we see that it also manages to successfully recreate the compute throughput of the original workload, indicating that the sleep time relationships are accurate. On the other hand, it completely fails to recreate the data throughput, obtaining a value consistently 5 times lower for any batch size. The same linear relationship between batch size and throughput hold for the benchmark however.

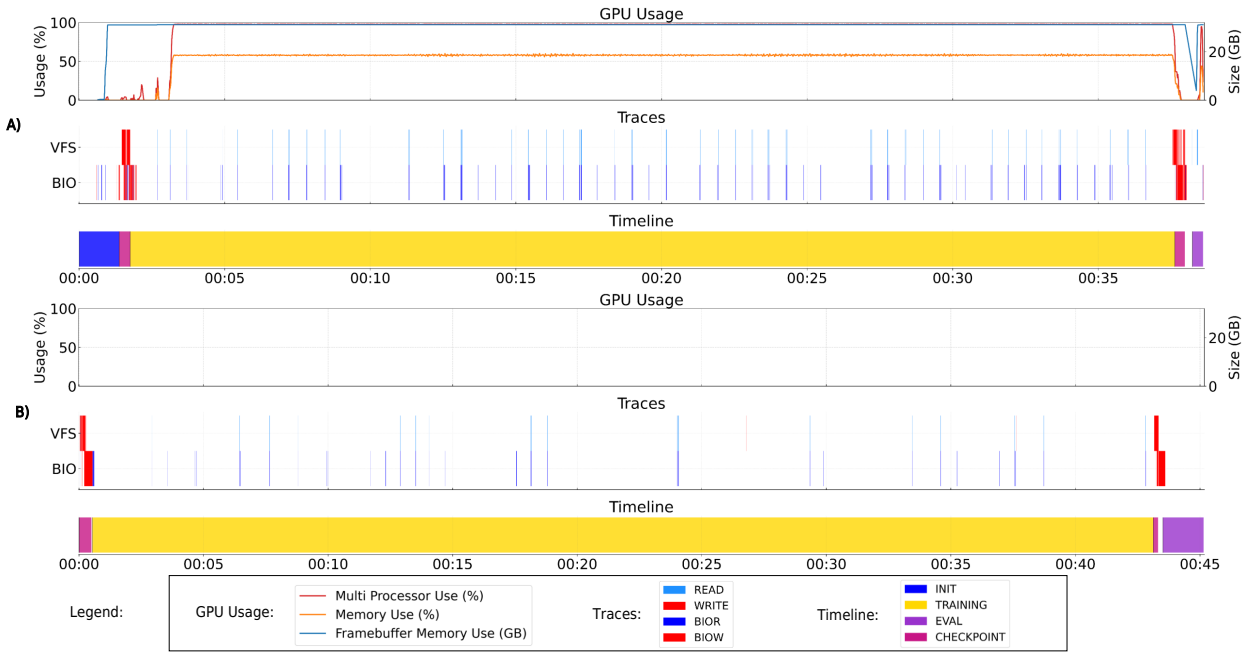

Figure 5.12: Trace visualizations of A) BERT and B) the MlPerf Storage Benchmark emulation.

5.2.3 Benchmark Tracing and Similarity

We now qualitatively compare the trace visualizations for the reference BERT workload and the equivalent benchmark run.

Visually, the most striking difference is the sparser trace data for the benchmark which could indicate a lower number of operations, or faster completing operations (at the scale at which we are plotting these traces, short events end up with a width of less than a pixel and do not show at all). For the events we do see, the general pattern seems to match the original workload, with some bursts of longer calls, though they also look shorter than the original's. Additionally, we see that the overall training time is around 5 minutes longer for the benchmark, indicating that the sleep time equations may have overestimated the sleep time for this run. The write activity during checkpoint looks very similar.

We quantify the similarity of these two runs and obtain a score of 96%, with the components broken down in Table 5.2. The VFS read behaviours of the two runs are virtually identical, with only slight differences in the number of reads, and number of files read from the benchmark.

On the other hand, the VFS writes again vary substantially, the most different component being the median VFS write size, which is about 33 times larger in the benchmark indicating that its VFS writes are done in larger chunks than the original workload. The second most different component is the number of unique files written, 50 for the original workload vs. 16 for the benchmark.

In addition, as a group the BIO reads seem to be significantly different, with the benchmark showing 6.11 % less operations, reading 70 MB less overall and in a more random manner. The benchmark also seems to make better use of caching, with about 13% less data being read from storage than requested from the file system, whereas the original workload actually reads more data from storage than requested.

Component	Weight	BERT	MLPerf Storage	Difference
Number VFS reads	2	1416	1375	-2.9 %
Amount of data VFS read (B)	2	358,804,912	360,448,000	0.46 %
Number unique files read	2	195	193	-1.03 %
1st quartile of VFS read size (B)	2	262144	262144	0.0 %
median of VFS read size (B)	2	262144	262144	0.0 %
3rd quartile of VFS read size (B)	2	262144	262144	0.0 %
Number BIO reads	2	2683	2519	-6.11 %
Amount BIO read (B)	2	383,926,272	313,495,552	-18.34 %
Random BIO reads (%)	2	69.25	86.58	25.03 %
Sequential BIO reads (%)	2	30.75	13.42	-56.36 %
Amount BIO read / Amount VFS read (%)	2	107.0	86.97	-18.72 %
Number VFS writes	1	18624	15392	-17.35 %
Amount of data VFS written (B)	1	64,751,582,944	64,555,412,992	-0.3 %
Unique Data Files Written	1	50	16	-68.0 %
1st quartile VFS write size (B)	1	4096	4096	0.0 %
Median VFS write size (B)	1	12288	4090736	33190.49 %
3rd quartile VFS write size (B)	1	8384512	8384512	0.0 %
Number BIO writes	1	219,012	248,983	13.68 %
Amount BIO written (B)	1	57,171,083,264	64,456,413,184	12.74 %
Similarity	**0.96**			

Table 5.2: Individual components of the similarity measure for BERT and the benchmark emulation.

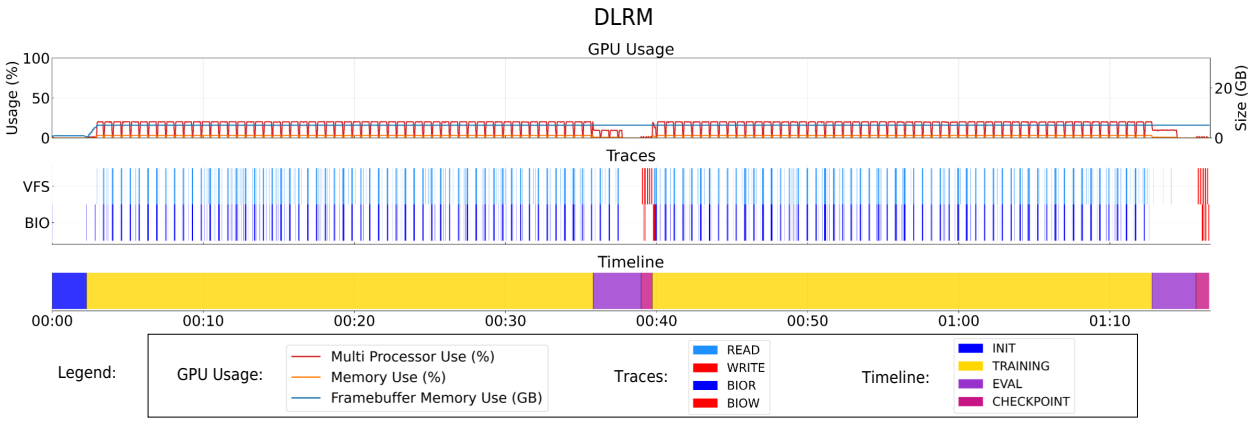

Figure 5.13: Trace visualization of a DLRM workload run using 8 GPUs and a global batch size of 32,768. The workload trains for 32,768 steps total, performing 4,096 steps of evaluation every 16,384 training steps and checkpointing right after.

A point that could be of importance in explaining these differences is the use of different major versions of TensorFlow between the original workload and the benchmark. BERT's original code is relatively old and uses version 1 of the framework, while the benchmark implementation is more recent and has switched to version 2. It could be that prefetching behaviour has changed in version 2, and is performed in a more beneficial manner for the workload, issuing less but more successful prefetch operations, as illustrated by the lower percentage of VFS that go to storage. Thus, for the same amount of data read at the VFS level, less is read at the BIO level. This would explain the smaller number of BIO reads, the 70MB BIO read difference, and the lower fraction of sequential reads observed in the benchmark, considering prefetching is done sequentially.

5.3 DLRM

5.3.1 Workload high-level traces

Figure 5.13 shows a typical run of DLRM using 8 GPUs and a global batch size of 32768. It trains for 32,768 steps in total, performing 4,096 steps of evaluation every 16,384 training steps and checkpointing thereafter.

We first observe that the GPUs are nowhere near 100% utilization, for both processing and framebuffer memory use. Indeed, this workload is not as computationally intensive as the others, with relatively smaller neural networks and embedding table operations which are really just table

lookups. In experiments, we observe an increase in GPU processing as we increase the batch size up to the maximum of 2M, which increases the number of operations performed in the network. The memory usage also increases with batch size, until we eventually run out of memory with 8 GPUs above 2M. If we were to increase the embedding table dimension, we would also see an increased memory usage as the larger tables require more space on the GPUs.

At the trace level, we observe BIO-level reads under VFS reads for the entire duration of training, indicating the storage is being hit and that there is little caching occurring. This was expected since the training dataset is too large to fit in memory, and we do not train for a full epoch anyway. Making things worse, DLRM shuffles its training data by reading from a random offset into its training file, losing the benefits it could gain from any kind of FS level prefetching. Since this run did not read enough data to saturate our machine's main memory, we do observe the caching of the evaluation set, which provides a small speedup and more stable GPU utilization.

We can observe periodic drops in GPU computation, occurring approximately every 30 seconds. These drops correspond with periods of longer lasting VFS and BIO reads. Looking into the trace data, we discovered that indeed, BIO reads periodically exhibit two orders of magnitude longer latencies for the same request sizes. Since DLRM's data loading is synchronous, this causes a drop in GPU processing. This behaviour seems to be an artifact of our machine's drive, a RAID drive. Perhaps it performs some periodic checks that conflict with these reads and reveals itself under regular and sustained load. No concurrent reading or writing to this drive was observed system wide during these experiments.

DLRM would benefit from a better shuffling mechanism. For example, TensorFlow's `shuffle()` buffers used in the BERT input pipeline take a more efficient approach, by reading more data from disk and shuffling the samples in memory. Of course, the shuffling provided is "lesser" in the sense that we would not pick randomly from the whole dataset, but from sequential ranges of the input file. Perhaps a hybrid approach could be taken to read in large chunks at random offsets and shuffle from memory. On our machine at least, DLRM would avoid the periodically long BIO reads and resulting drops in GPU computation, offering better utilization overall.

5.3.2 Instrumentation

Step breakdown

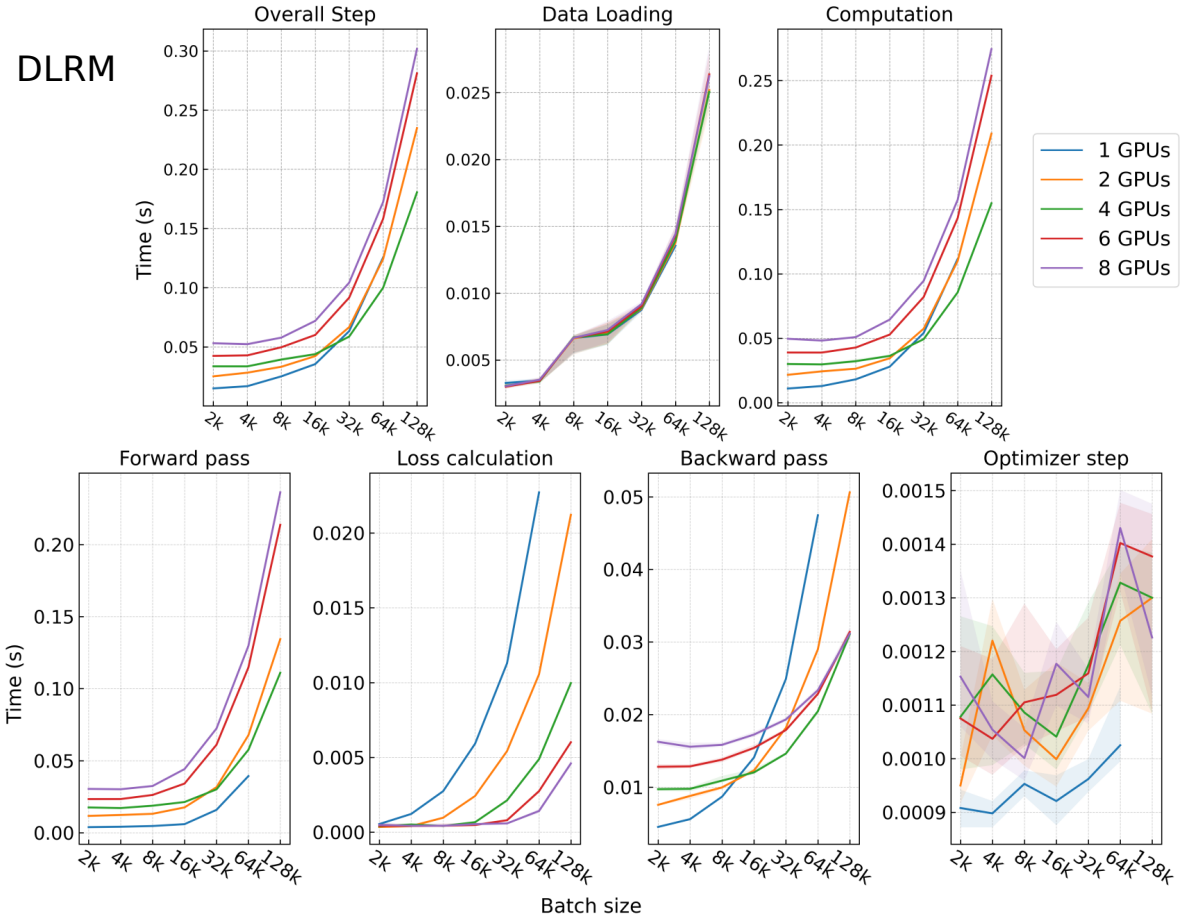

Figure 5.14: DLRM training step time breakdown. Median values shown, with a fill between the 1st and 3rd quartiles. The top row shows the overall step and a breakdown in its data loading and computation components. The bottom row shows a further breakdown of the computation into its sub-components. Data aggregated from 3 independent sets of runs. Note the logarithmic x-axis, with batch sizes doubling each time. The 1 GPU jobs run out of memory with 128k batch size.

Figure 5.14 shows the breakdown of a DLRM training step into its components. This time, we do not have a "load to GPU" time, as the batches are never simply sent to the GPUs but instead split into categorical features, sent to GPU holding the correct embedding table, re-assembled and distributed to all GPUs for the model-parallel part of a step.

Overall, we find that the data loading time is responsible for about 10% of the overall step time. Additionally, we see that the computation time depends on both batch size and number of GPUs.

Since the x-axis is logarithmic, the exponential looking curves indicate linear relationships. The further breakdown of the computation on the second row of Figure 5.14 shows that the various components of computation do not scale the same way with increasing batch size and number of GPUs. The forward pass is responsible for the majority of the computation time, about 80%.

We notice that as the number of GPUs increase, so does the step time. Thus, for a given batch size, the more GPUs we add, the longer each step will take. This step time increase is most likely due to the extra communication overhead necessary given the model-parallel distribution. The exception to this pattern is the 4GPU case, which crosses over the other curves around batch sizes of 16,384 and 32,768 and achieves minimal latencies thereafter. The 4GPU case could represent an optimal configuration on our machine, given the way the GPU communication network is setup.

Using the instrumented runs, we are able to derive the following linear relationships for computation time mean and standard deviation (with R^2 of 0.95 and 0.71), given a batch size and number of GPUs.

$$Mean = 6.311 * 10^{-3} * NumGPUs + 1.475 * 10^{-6} * BatchSize - 3.612 * 10^{-3} \tag{5.5}$$

$$Std = 3.289 * 10^{-4} * NumGPUs + 6.447 * 10^{-8} * BatchSize - 1.446 * 10^{-3} \tag{5.6}$$

For this workload, evaluation performs the same computation as the training forward pass, and we opt to approximate it with the same equations. This will be an overestimation since the backward pass and weight update are not performed.

Latencies and Throughputs

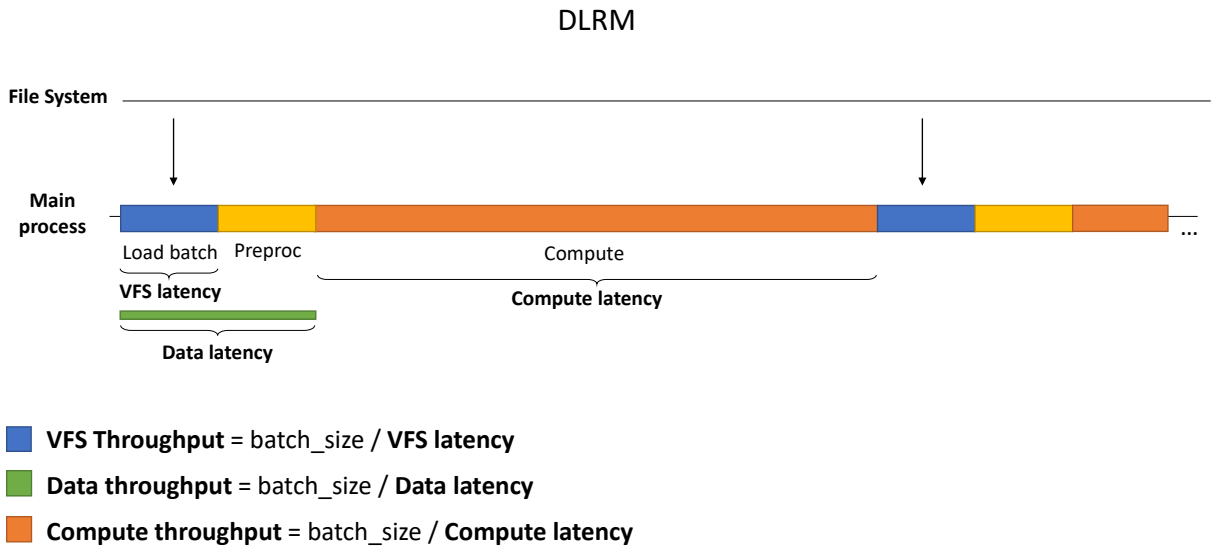

Figure 5.15: Diagram of DLRM's data loading and instrumentation measures. In this case, **batch_size** samples are read directly from a file on disk and pre-processed as one before being sent to the model for computation. There are no parallel data loading workers so data loading is synchronous to computation.

We diagram DLRM's data loading mechanism in Figure 5.15. DLRM uses a single data loader for all accelerators and makes no use of parallel data loading or prefetching. Hence, we have a simplified mechanism in this case with the data loading and preprocessing occurring synchronously with computation. Additionally, instead of loading individual samples from the file system like UNET3D, DLRM reads a whole batch from the training data file at once, thus performing a single I/O operation each step. In this case, the data latency includes the VFS latency, as well as the preprocessing.

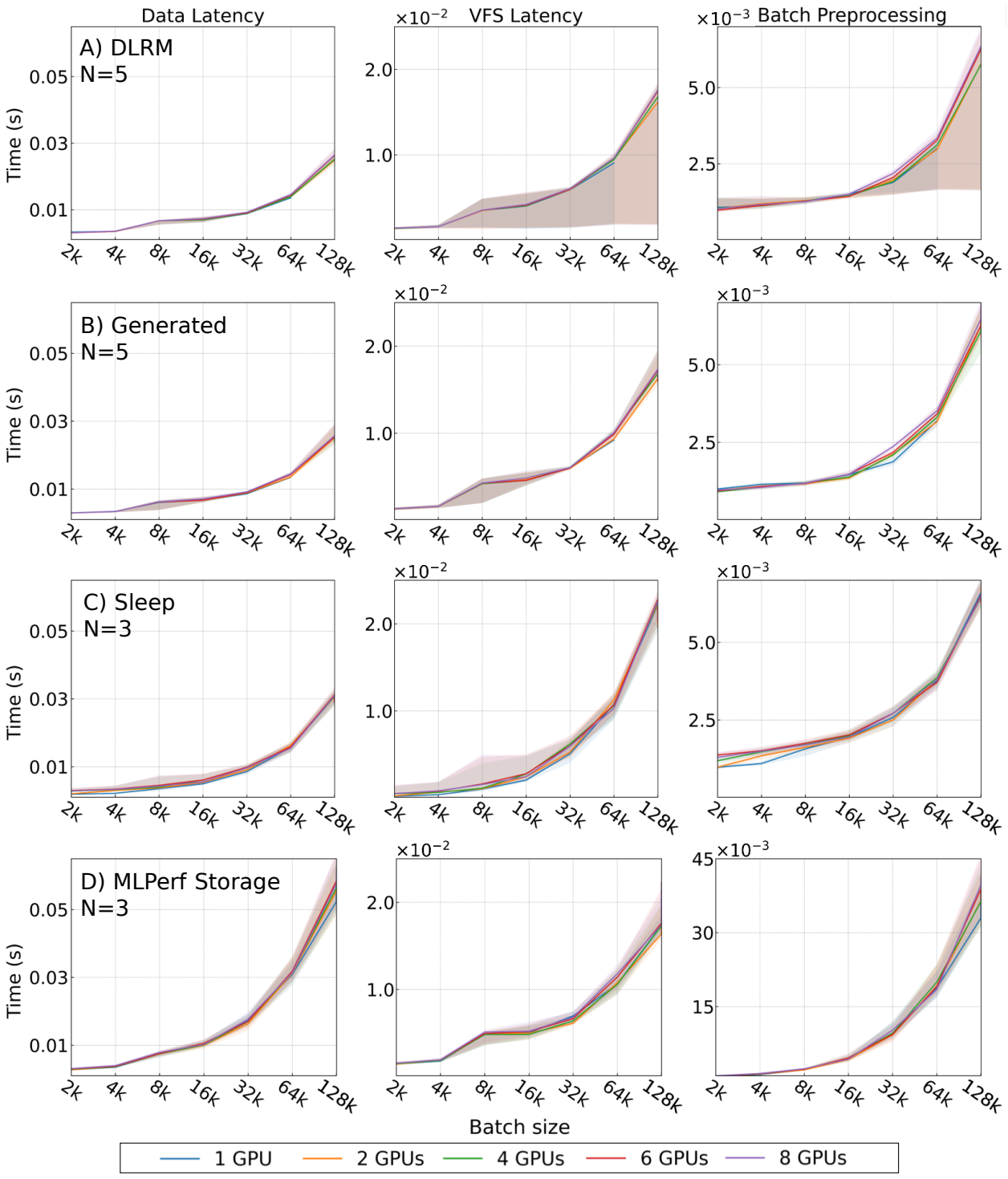

Figure 5.16: Data, VFS and Batch Preprocessing latencies across batch sizes and number of GPUs for A) the real DLRM workload, B) the generated data experiments, C) the sleep experiments and D) the MLPerf Storage benchmark. Median values with inter-quartile fill. N denotes independent sets of runs.

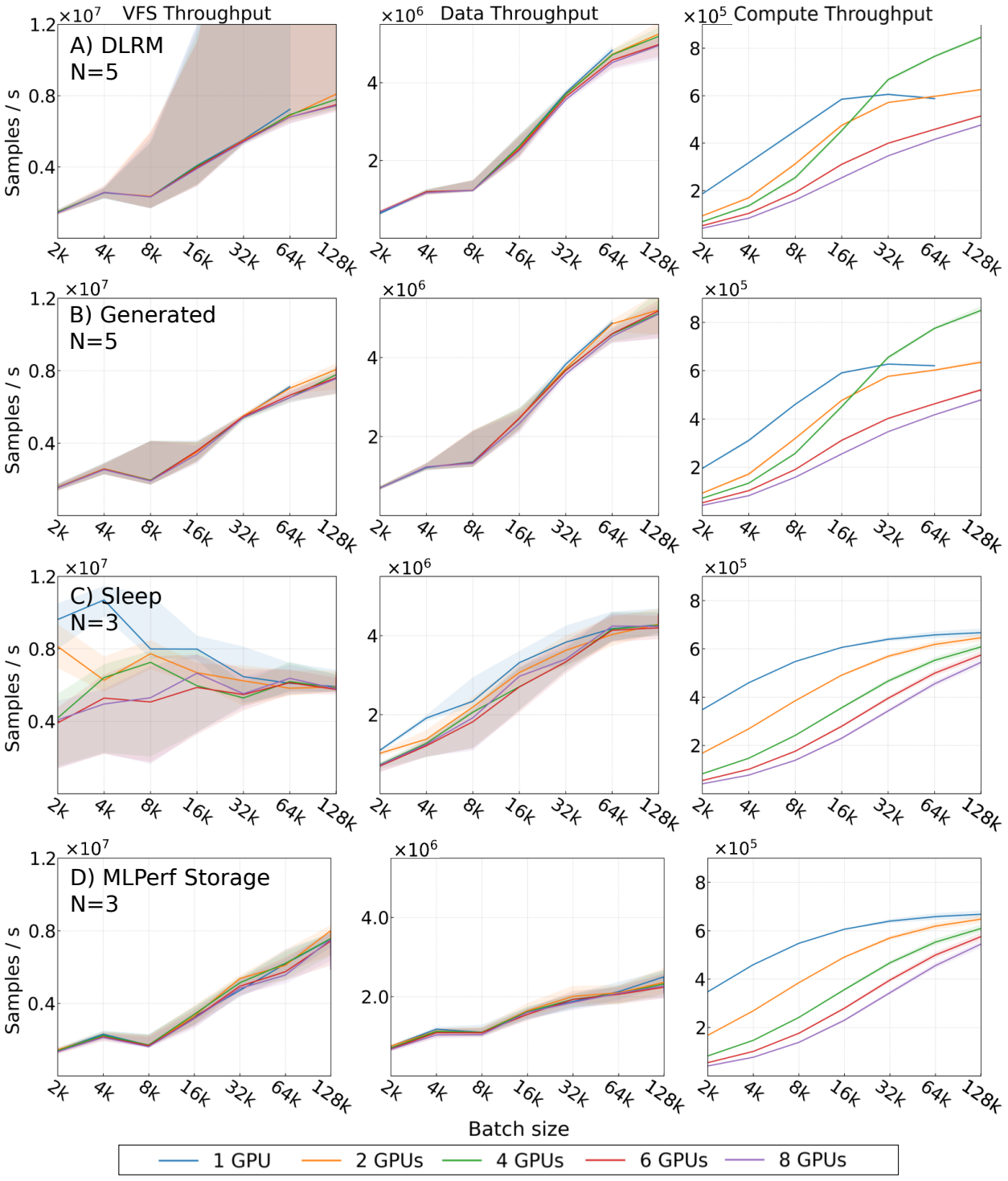

Figure 5.17: VFS, Data and Compute Throughputs measured across batch sizes and number of GPUs for A) the real DLRM workload, B) the generated data experiments, C) the sleep experiments and D) the MLPerf Storage benchmark. Median values with inter-quartile fill. N denotes independent sets of runs. The full inter-quartile range is not shown in A) for visibility.

Looking at the latencies in Figure 5.16 and the throughputs in Figure 5.17 we can again study how a generated dataset and a sleep time affect the workload.

For the original, generated data and sleep runs of both figures, we see that the median latencies and throughputs are all very similar. This does not hold for the benchmark starting at batch sizes of 16k and up. Like for the other two workloads, we see that the data throughput is reduced, achieving only around 2.5M samples/second for the highest batch size, versus 5M samples/second for the original workload.

The VFS latency also shows some unexpected behaviour between the real and generated data runs, where even though the median values are in line with each other, the variation is larger for the original workload and tend towards shorter latencies. We observe a similar variation in the preprocessing latencies. We could not think of a reason why this would be the case considering how the synthetic data was generated. This patterns persists through multiple independent runs of the experiments. Recall from Figure 5.15 that for DLRM, the VFS latency (batch reading) and batch preprocessing occur synchronously, and sum up to the data latency.

Looking at the compute throughput now, we see that they are virtually identical for the real and generated data, showing that it does not impact computation. We observe a difference when compared to the sleep and benchmark however (which both use the sleep time derived from Equation 5.5 and Equation 5.6. It seems the original DLRM shows increased compute throughput for the 4 GPU setting, with the curve rising above all others for batch sizes above 16k. Using the sleep times formula however, the curves become more "orderly". Except for the 4 GPU case however, we see that the generated sleep times approximate the behaviour of the original, seemingly reaching an asymptote around 600,000 samples/second.

5.3.3 Benchmark Tracing and Similarity

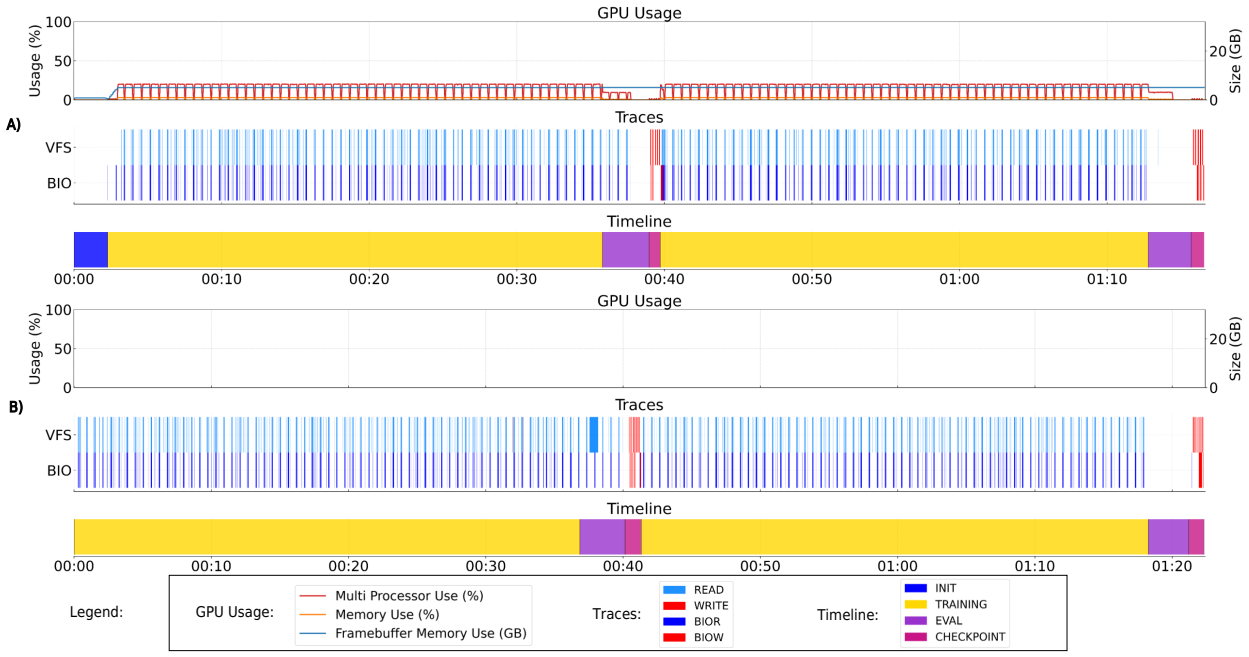

Figure 5.18: Trace visualizations of A) DLRM and B) the MlPerf Storage Benchmark emulation.

We now take a look at a visual comparison of traces from DLRM and its MLPerf Storage benchmark implementation. Here we are relieved to observe the same pattern of periodically longer BIO reads as for the original, indicating that it does seem to be an artifact of the storage system of our machine, and confirming that we are able to recreate the load accurately with the benchmark. The general structure of the workload seems to be followed through training, evaluation and checkpointing, and the two runs complete in similar amounts of time, though DLRM is around 5 minutes faster.

Quantitatively, these two runs achieve a similarity of 98 %, broken down in Table 5.3. We see that both the VFS and BIO read components are all within approximately 5% of each other, with neither workload showing any kind of caching. Interestingly, the BIO reads show up as mostly sequential, whereas we would expect them to be mostly random, based on the way shuffling is done through random file offsets. By looking at the raw trace data however, we see that in practice, the VFS reads get broken up into a series of sequential BIO reads. For example, for this particular trace, the VFS reads in 5,242,880B chunks from the file, that get broken up into twenty 262,144B reads at the BIO level, explaining the higher percentage of random reads.

Once again, the VFS write distribution is the source of most of the differences, with the median write size being 280 times larger in the original workload compared to the benchmark. The difference in number of unique files written to is however explained by the benchmark using two different filenames for its checkpoint file while the original workload overwrites the previous checkpoint.

Component	Weight	DLRM	MLPerf Storage	Difference
Number VFS reads	2	36867	36862	-0.01 %
Amount of data VFS read (B)	2	182,546,595,840	173,140,213,760	-5.15 %
Number unique files read	2	2	2	0.0 %
1st quartile of VFS read size (B)	2	5242880	5242880	0.0 %
median of VFS read size (B)	2	5242880	5242880	0.0 %
3rd quartile of VFS read size (B)	2	5242880	5242880	0.0 %
Number BIO reads	2	698019	682428	-2.23 %
Amount BIO read (B)	2	182,963,814,400	178,359,902,208	-2.52 %
Random BIO reads (%)	2	5.46	5.29	-3.16 %
Sequential BIO reads (%)	2	94.54	94.71	0.18 %
Amount BIO read / Amount VFS read (%)	2	100.23	103.01	2.78 %
Number VFS writes	1	106	135	27.36 %
Amount of data VFS written (B)	1	54,871,274,798	54,258,333,122	-1.12 %
Unique Data Files Written *	1	1	2	100.0 %
1st quartile VFS write size (B)	1	7348	7348	-0.01 %
Median VFS write size (B)	1	2097216	7446	-99.64 %
3rd quartile VFS write size (B)	1	825,041,024	724,178,468	-12.23 %
Number BIO writes	1	206960	204929	-0.98 %
Amount BIO written (B)	1	54,230,310,912	53,688,725,504	-1.0 %
Similarity	**0.98**			

Table 5.3: Individual components of the similarity measure for DLRM and the MLPerf Storage benchmark emulation. * The original workload overwrites the previous checkpoint file, while the benchmark writes two different ones.

6

Discussion

Since workload-specific discussions of results were included in the previous section, we focus here on presenting more general discussion, curious things uncovered, recurring motifs, limitations and possibilities for future work.

6.1 Sleep Times Derived

As noted in the benchmark methodology, the computation time measurements and derived relationships represent only the accelerators we used, namely NVIDIA V100 GPUs with 32GB of memory. As such the benchmark can only be taken to realistic emulate these and new measurements should be taken to emulate other accelerators. One could however, play with the relationships derived here and decrease or increase the coefficients to emulate relatively more/less performant accelerators.

6.2 Timelines

Anecdotally, and as evidence of the usefulness of system-level profiling, earlier versions of the UNET3D timeline visualizations Figure 5.1 allowed us to detect a bug in the MLPerf Training reference implementation. Contrary to expectations, we were seeing continued BIO reads up to the fourth epoch and more, indicating that all training cases were not being seen during the first epoch. Further investigation, revealed that the data was not getting properly sharded between workers due to an improper seeding of the `DistributedSampler`, where each worker was initialized with a different seed. The seed must be the same for each worker to guarantee an exclusive split, and this was not the case [6]

Regarding the periodic drops in GPU computation seen in Figure 5.1 caused by the data loader resetting every epoch and needing to wait for their first batch to assemble, in this case it is flagrant due to the small amount of global steps performed, but may not matter much for longer workloads. Nonetheless, overlapping the assembly of the first batch of an epoch, with processing the last batch of the previous one, might significantly increase GPU utilization in this case. It is important to seed the distributed sampler differently every epoch. But perhaps this could be streamlined by passing in a sources of seeds and implementing something like a `repeat()` and `shuffle()` transformations from Tensorflow, allowing the dataset to repeat indefinitely while begin shuffled in memory.

An interesting phenomenon that appeared through the trace visualizations was the periodically longer BIO reads. These can be seen in both the DLRM and to a lesser degree, the BERT traces. We have hypothesized that they are an artifact of our machine's specific drive, but it would be interesting to prove this, by running similar traces using a different drive or a different machine, and studying the inner workings of the machine's drive to look for an explanation. What we seem to see here is that, under a sustained and sufficiently intensive read workload, the pattern seems to appear. Additionally, as a note of data processing caution, these BIO calls are all in the 99^{th} percentile of latency and as such would not show up if we were to remove them as outliers, as was done in previous versions of the plots.

[6] A pull request was opened to fix this bug at `https://github.com/mlcommons/training/pull/625` but has not yet been merged at the time of writing.

6.3 Notes on Instrumentation

Regarding UNET3D's training step breakdown in Figure 5.2, we originally expected the backward pass to be responsible for most of the computation latency, as it requires an all-reduce on the model parameters between all accelerators and a broadcast of the averaged gradients. For reference, the "Cumulative loss" step performs an all-reduce as well and is the longest part of the computation. However, PyTorch `DistributedDataParallel` optimizes the backward pass by synchronizing the gradients in "buckets" as soon as they are calculated [59].

UNET3D's 4 order of magnitude increase in median preprocessing latency Figure 5.5 when training on generated data was also very surprising to us. Considering the dimensions and range of values of the synthetic data follow similar distributions as the original dataset, we hypothesize that it must be the distribution of pixel values that are causing it. Our values were uniformly generated from the observed value range. But if we were to view such an image, it would look purely like white noise. On the other hand, a real picture's pixel values are not uniformly random. A real image has structure, with large clusters of neighbouring pixels contained within small ranges of values. Further instrumentation work could be done to investigate exactly which of the preprocessing operations cause the increase, and to see if a relationship can be derived.

The role of batch size and number of distributed accelerators used have on model training convergence and accuracy is non-trivial and much work has been done to investigate their relationship [8, 34, 45, 65], so in practice some of the tested batch sizes may not be relevant to practitioners, at least probably not without changing other parameters in tandem. Unlike an end-to-end benchmark like Fathom [6], we did not concern ourselves with model accuracy at the end of the training process.

The benchmark implementations of UNET3D, BERT and DLRM at high batch sizes all show a substantial decrease in data throughput when compared to the original workload. We could not find an explanation for this. A first hypothesis was that the extra layers of abstraction in the benchmark were responsible. Indeed, the dataloaders are wrapped in Python generators, but instrumenting at lower levels revealed this only accounted for a very small difference. It seems instead that there could be some non-obvious multi-processing effects occurring. Some circumstantial evidence for this possibility are the unexpected effects of removing the "Cumulative loss" from UNET3D Figure A.1.

On that note, the effect of removing the cumulative loss is very puzzling and reveals that behind the seemingly simple PyTorch training loop, a lot of complexity is hidden and unforeseen interaction effects like this can arise. Unfortunately, we did not have time to investigate this strange behaviour. Future work could investigate this and attempt to understand the root cause of this by analyzing the PyTorch source code.

We note that for UNET3D and BERT, the data loading latency and throughput are not really material to the I/O loads, as data loading is done in parallel to training. We posit that as long as the VFS throughput and the compute throughput are similar between the benchmark and the real workloads, as they are, then the result will be similar which the similarity metric results confirm.

Due to the time required to perform these experiments, we only have a few full sets of instrumented runs across the batch size and number of GPU configurations so perhaps we do not have enough data to fully iron out the natural variability in server states at the time of tracing, though care is taken to ensure we have exclusive access to run these experiments however, and the individual runs align with each other. Given more time, collecting more of each run would allow to really solidify the existence of these patterns.

6.4 Notes on the Similarity Metric

Differences in checkpointing behaviour is a recurring theme across all workload emulations and are due to the way checkpointing was implemented in the benchmark. The benchmark naively implements checkpointing by writing a file of the correct size, evidenced by the amount of data written being the same. However, in the real workloads, the framework write out their checkpoints in a much more complex manner, sometimes writing and combining multiple files, and writing out parameter values in organized sections, leading to vastly different writing patterns. The upside of not using the framework facilities for checkpointing is that the benchmark does not need to construct the model in memory, which could be a limiting factor while testing.

A middle ground was reached here where the naive checkpointing behaviour of the benchmark was modified and broken up into multiple smaller writes. While an attempt was made to follow the same distribution as the real workload, it did not follow any obvious statistical distributions known to us. Thus the writing patterns tend to be different.

For the most accurate emulation, future work could look into defining a probability density function using the median, 1st and 3rd quartiles. With it we could generate the correct distribution of writes for checkpointing.

Other observed differences to not yet have an explanation. For example, the higher number of BIO reads seen in Table 5.1 remains a mystery at the time of writing, with no obvious reason why the benchmark would exhibit this behaviour, though it is consistent across runs.

Overall the concept of comparing the traces of workloads is interesting. Both [21] and [6] employ cosine similarities and we followed the example, but a search of the literature indicates that not much work has been done in this domain. Future work could explore other ways to measure the similarity of traces. One idea that was not explored could be to use radar charts [69], with a different scale along each axis to account for the widely different ranges of values for each component, and use the overlapping area between two workloads as a measure. This has the advantage of an intuitive visualization as well.

6.5 Usage of the Benchmark and Future Directions

This work was not done in a vacuum and the MLPerf Storage team has been busy interacting with industry practitioners and stakeholders to gather feedback and release v0.5 of the benchmark as a preview package. Some of the developments presented in this work are not merged with the master branch of the tool, and may never be. Notably the checkpointing behaviour was implemented in an ad-hoc manner as the similarity metric results showed large discrepancies in writing behaviour.

Since so much time was spent analyzing and developing the benchmark, less effort has been made in demonstrating its usage. The tool can be used to extrapolate the I/O behaviour of workload configurations which cannot be actually run on a given machine. One such example of this is present in Figure 5.17 where the MLPerf Storage compute throughput plot shows values for the 1 GPU, 128k batch size, a configuration which would run out of memory if the real workload was used. Given more time, exploring the throughput asymptotes that seem to appear at the edges of the plots would be interesting.

Additionally, it was reported by MLPerf Storage working group members that modern commercial storage systems would not experience much difficulties in reaching the necessary throughput to

feed either one of these workloads alone. Thus, future work could use the benchmark to emulate multiple workloads running concurrently and effecting combined load on a storage system. This can be done in multiple ways:

- Scaling up a single workload to multi-node. In this work, we only considered the case of a workload training on a single node. However, large models are routinely trained on entire clusters with dedicated storage nodes. The benchmark can already run in multi-node setting thanks to its use of MPI to emulate accelerators, but the realism of such a workload has not been proven empirically. Especially interesting will be studying the inter-nodal communication and how it scales as the number of node grows. A parameter to scale model size, which will determine the amount of data shared between nodes for gradient reduction could also be worth exploring. Additionally, cluster storage nodes often have multiple network interfaces whose bandwidth is lower than the storage system's. Thus, the cluster setting will be essential in truly testing the capacity of modern storage nodes.

- Multiple instance of a single workload. These instance can share the same dataset and represent a hyper-parameter search scenario, where optimal values for batch size, model dimensions, number of layers, etc. are searched. As noted in [48], these types of jobs often perform much unnecessary work and much better performance can be achieved by implementing smart caching mechanisms. This benchmark could be used to test these solutions.

- Different workloads training at once. This setting would replicate a multi-tenant or shared node with many different jobs running concurrently. There are many interaction effects to analyze, obtained by overlapping the training jobs differently. In this context, workload aware scheduling algorithms could be explored to optimize resource use.

Finally, work has already started in our group looking more closely at the preprocessing part of data loading pipelines. In the context of Deep Learning, an I/O benchmark is incomplete without the associated preprocessing, which constitutes the second and sometimes most computationally intensive part of the input pipeline, responsible for causing the stalls in data loading [48, 49]. A preprocessing benchmark would be targeted at the data loading servers' CPU and memory capacity however, in addition to the storage system.

7

Conclusion

In this work, we explored the I/O patterns of three DL training workloads from various domains and with very different dataset characteristics. We traced and instrumented them, visualizing their I/O patterns at a high-level, and deriving relationships for the model computation time with respect to batch size and number of accelerators. Depending on the model's architectural details, computation time was shown to scale with batch size and the number of accelerators used.

After confirming that replacing the model computation with a CPU `sleep()` and that training on synthetically generated data did not significantly modify the I/O behaviour of the workloads, we implemented their emulations in the MLPerf Storage benchmark and quantified its similarity to real workloads, achieving scores of 95% or above for all.

It is our hope that the benchmark will be useful to storage system researchers and developers, cluster architects and other professionals to test their solutions under realistic loads generated by these three training workloads.

Bibliography

[1] Mlperf™ storage benchmark suite.

[2] Top500 list.

[3] An NFS trace player for file system evaluation. San Francisco, CA, March 2004. USENIX Association.

[4] Mlcommons: Machine learning innovation to benefit everyone, 2020.

[5] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[6] Robert Adolf, Saketh Rama, Brandon Reagen, Gu yeon Wei, and David Brooks. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, sep 2016.

[7] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[8] Joel André, Foteini Strati, and Ana Klimovic. Exploring learning rate scaling rules for distributed ml training on transient resources. In *Proceedings of the 3rd International Workshop on Distributed Machine Learning*, DistributedML '22, page 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[10] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[11] Jens Axboe. fio documentation.

[12] Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. Explaining neural scaling laws, 2021.

[13] Google Brain. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models, 2022.

[14] Junyi Chai, Hao Zeng, Anming Li, and Eric W.T. Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[16] Soumith Chintala. Convnet-benchmarks: Easy benchmarking of all publicly accessible implementations of convnets., 2017.

[17] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 3d u-net: Learning dense volumetric segmentation from sparse annotation. *CoRR*, abs/1606.06650, 2016.

[18] Cody A. Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter D. Bailis, Kunle Olukotun, Christopher Ré, and Matei A. Zaharia. Dawnbench : An end-to-end deep learning benchmark and competition. 2017.

[19] Criteo. Download criteo 1tb click logs dataset.

[20] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate modeling and generation of storage i/o for datacenter workloads. *Proc. of EXERT, CA*, 2011.

[21] Hariharan Devarajan, Huihuo Zheng, Anthony Kougkas, Xian-He Sun, and Venkatram Vishwanath. Dlio: A data-centric benchmark for scientific deep learning applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 81–91, 2021.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[23] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[24] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[25] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. 2021.

[26] Tarek El-Ghazawi and Gideon Frieder. *Input-Output Operations*, page 874–879. John Wiley and Sons Ltd., GBR, 2003.

[27] Hua Fang, Zhaoyang Zhang, Chanpaul Jin Wang, Mahmoud Daneshmand, Chonggang Wang, and Honggang Wang. A survey of big data research. *IEEE Network*, 29(5):6–9, 2015.

[28] Association for Computing Machinery. Fathers of the Deep Learning Revolution Receive ACM A.M. Turing Award, 2018.

[29] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[30] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

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

[32] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. 2017.

[33] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[34] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks, 2018.

[35] Cerebras Systems Inc. Cerebras cs-2 whitepaper. 2021.

[36] Fabian Isensee, Philipp Kickingereder, Wolfgang Wick, Martin Bendszus, and Klaus H. Maier-Hein. No new-net. *CoRR*, abs/1809.10483, 2018.

[37] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[38] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 25, USA, 2005. USENIX Association.

[39] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, jun 2020.

[40] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. 2020.

[41] Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications*, 82(3):3713–3744, Jan 2023.

[42] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[43] Byron C. Lewis and Albert E. Crews. The evolution of benchmarking as a computer performance evaluation technique. *MIS Quarterly*, 9(1):7–16, 1985.

[44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. 2020.

[45] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 937–954, 2020.

[46] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2020.

[47] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, Jun 2021.

[48] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021.

[49] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[50] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[51] NVIDIA. NVIDIA cuDNN.

[52] NVIDIA. nvidia-smi user manual.

[53] NVIDIA. *NVIDIA DGX-1 With Tesla V100 System Architecture White Paper*. NVIDIA, 2017.

[54] NVIDIA. NVIDIA DALI documentation, 2018.

[55] OpenAI. Gpt-4 technical report, 2023.

[56] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C. Stern, and Artem Cherkasov. The transformational role of gpu computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, Mar 2022.

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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, 2019.

[58] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[59] PyTorch. Distributed data parallel - internal design.

[60] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2020.

[61] Baidu Research. Benchmarking deep learning operations on different hardware, 2017.

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

[63] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[64] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104, 2016.

[65] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size, 2018.

[66] TensorFlow. Tensorflow documentation - limiting gpu memory growth.

[67] Pablo Villalobos and Anson Ho. Trends in training dataset sizes, 2022.

[68] Jonathan Stuart Ward and Adam Barker. Undefined by data: A survey of big data definitions, 2013.

[69] Wikipedia. Radar chart wikipedia article.

[70] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics.

[71] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers, 2022.

[72] Minqi Zhou, Rong Zhang, Dadan Zeng, and Weining Qian. Services in the cloud computing era: A survey. In *2010 4th International Universal Communication Symposium*, pages 40–46, 2010.

A
Appendix

```
def num_ranseq_accesses(trace):
    random = 0
    seq = 0
    dic = {}    # map of (PID, sector)

    for row_data in trace:
        pid = row_data['PID']
        sector = row_data['SECTOR']
        sz = row_data['BYTES'] / 512    # Get the operation size in 512B sectors

        if pid not in dic:
            dic[pid] = sector+sz
            random += 1
        else:
            # If the current operation starts at the same sector
            # as the previous one by the same PID ended, it is sequential
            prev = dic[pid]
            if prev == sector:
                seq += 1
            else:
                random+=1
            dic[pid] = sector+sz

```

Listing A.1: Our algorithm to count the number of random and sequential accesses in the block level trace.

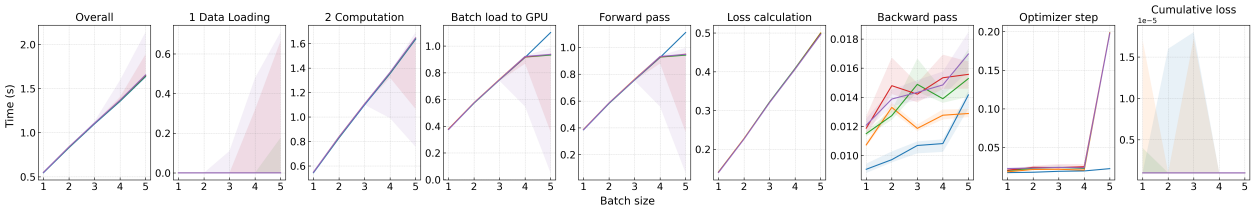

Figure A.1: Step breakdown after removing the "Cumulative loss" step from the training code. The instrumentation becomes erroneous as the individual components do not sum up to the total computation time. We still see the Cumulative loss as a component because we kept a timer around the commented out statement to show that it is reduced to almost nothing.

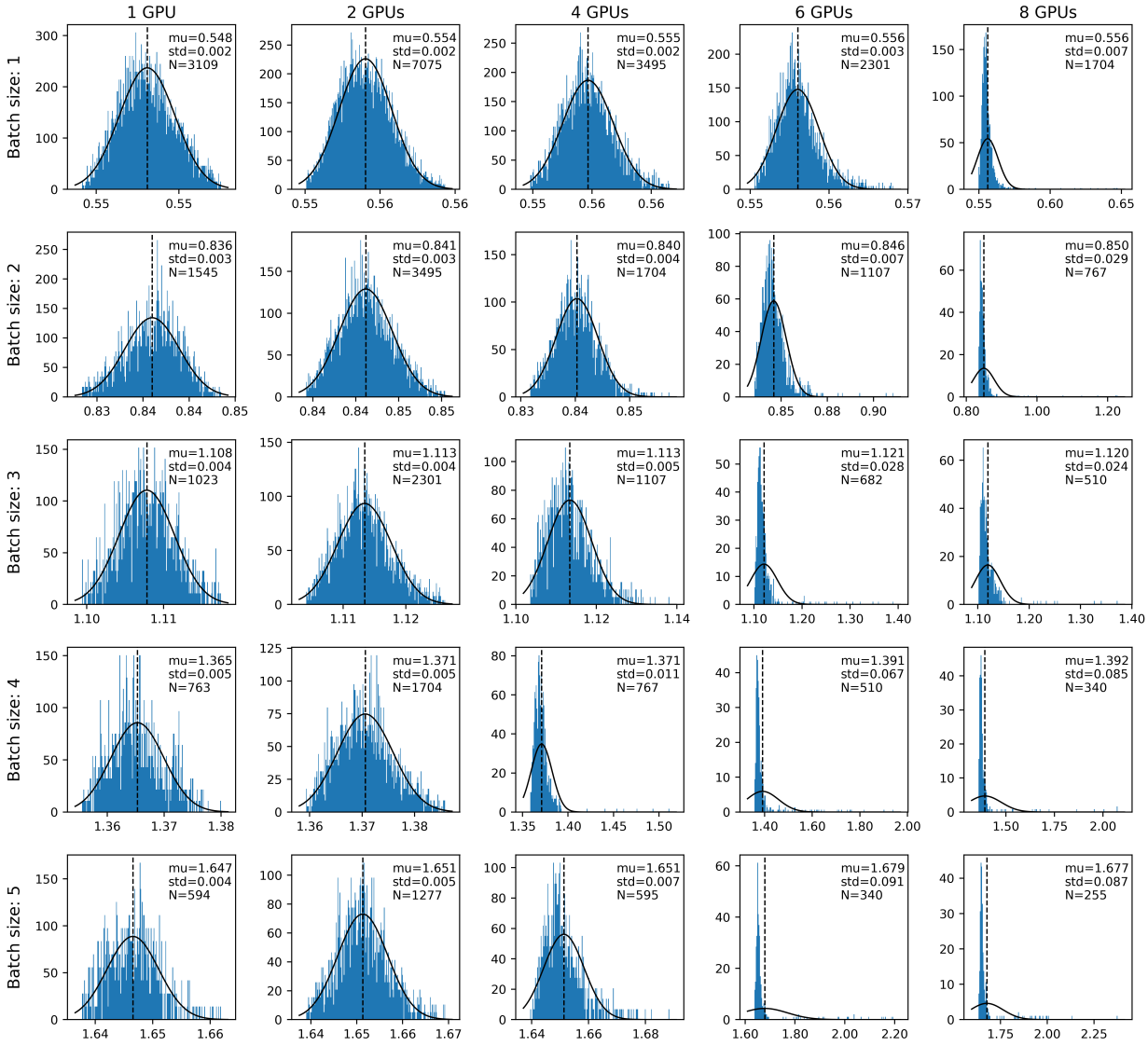

Figure A.2: All UNET3D computation time distributions with fitted normal distributions.

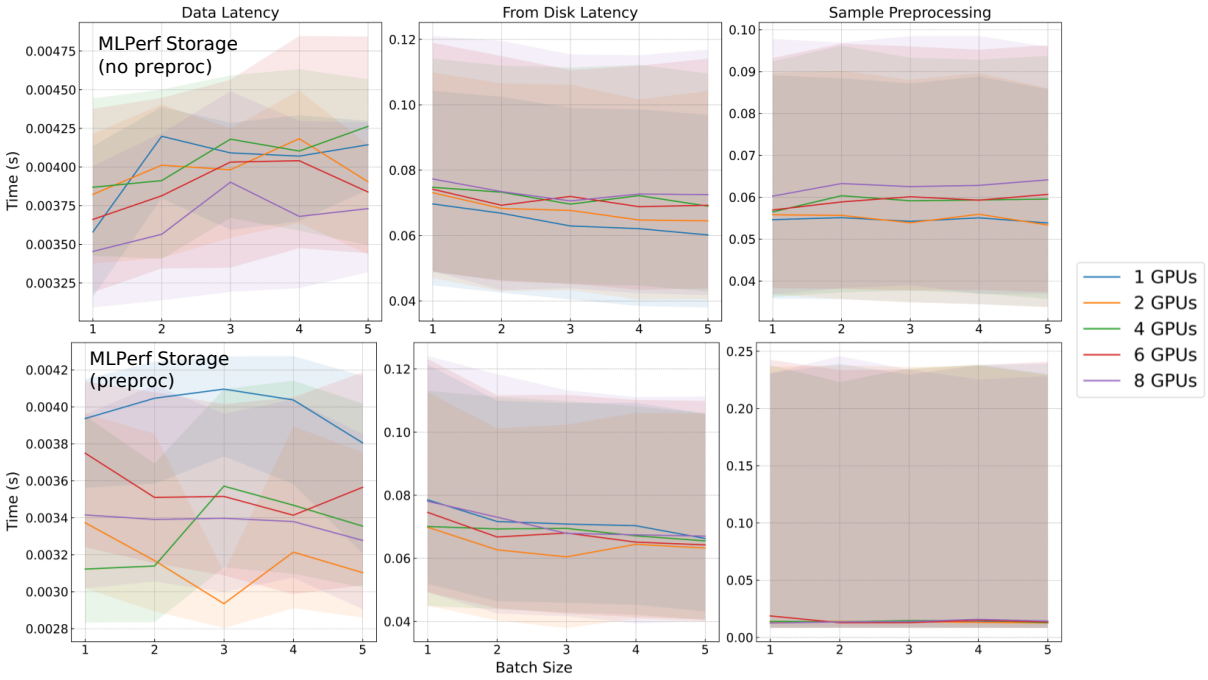

Figure A.3: Effect of simulating preprocessing by an extra sleep time in the dataloader.

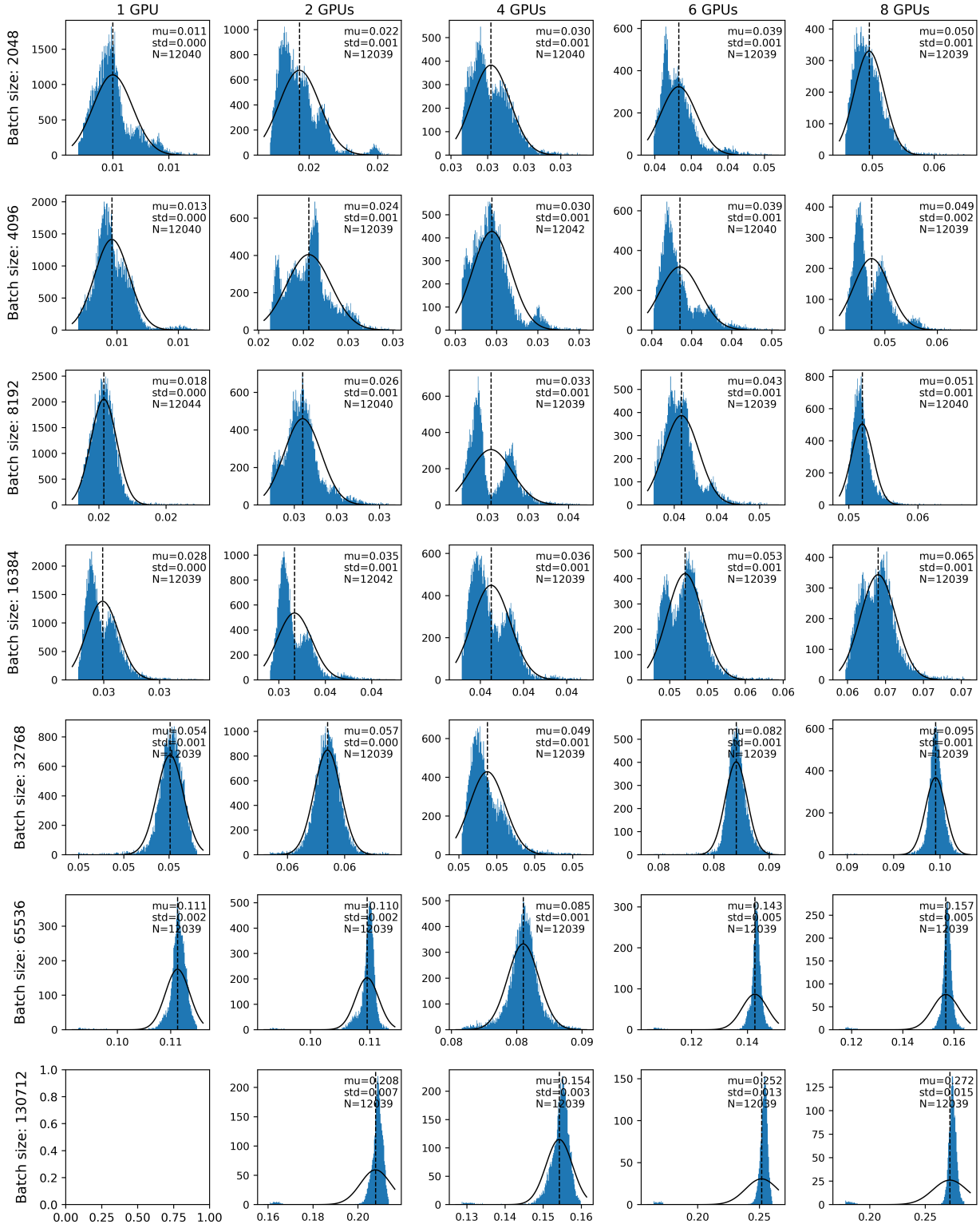

Figure A.4: All DLRM computation time distributions with fitted normal distributions.

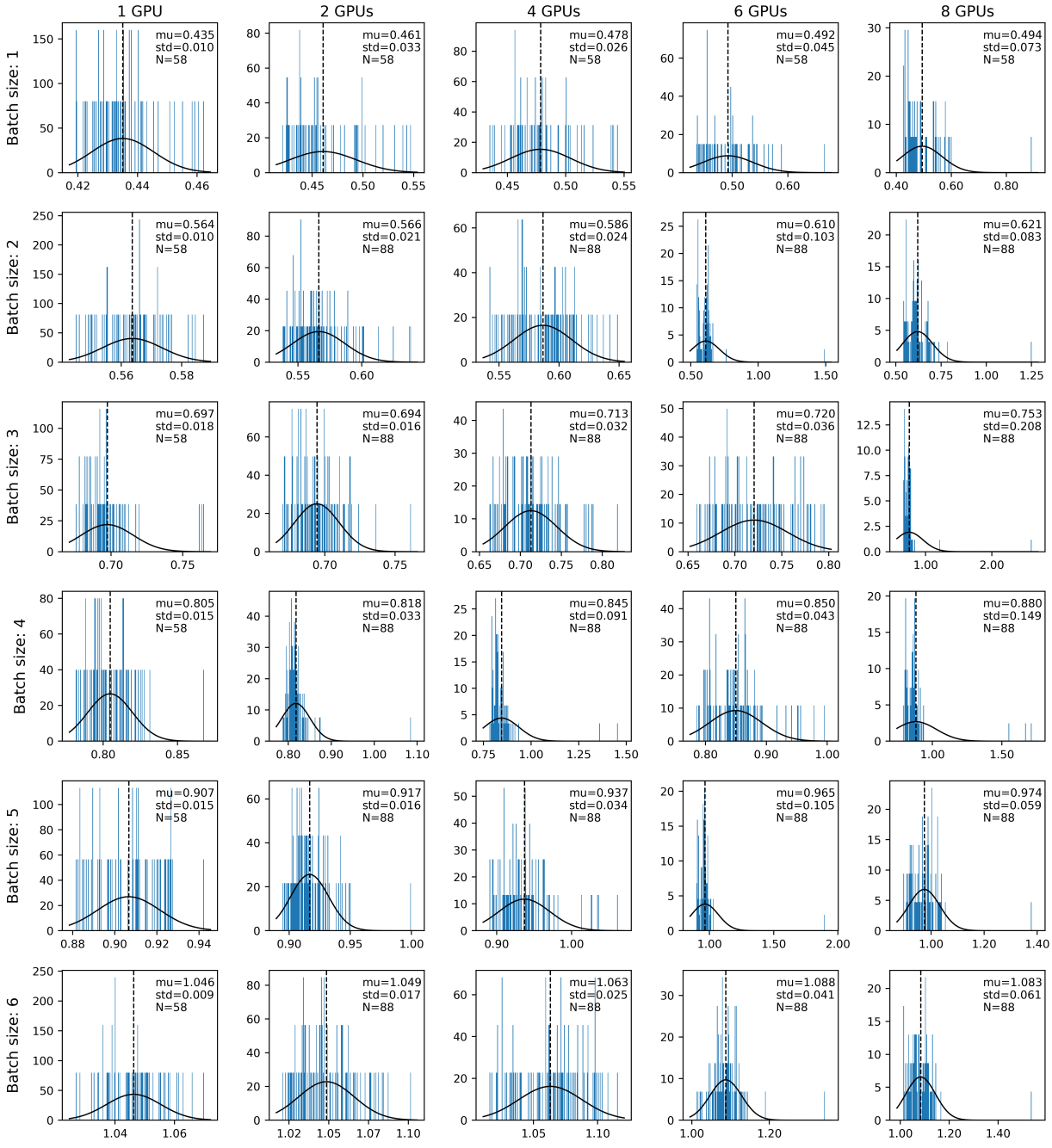

Figure A.5: All BERT computation time distributions with fitted normal distributions.