Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models

Shang Wang [1] [2] **Peiming Yang** [* 3 2] **Yuxuan Zheng** [* 4] **Xin Li** [* 2] **Gennady Pekhimenko** [5] [2]

Abstract

Driven by the tremendous effort in researching novel deep learning (DL) algorithms, the training cost of developing new models increases staggeringly in recent years. To reduce this training cost and optimize the cluster-wide hardware resource usage, we analyze GPU cluster usage statistics from a well-known research institute. Our study reveals that single-accelerator training jobs can dominate the cluster-wide resource consumption when launched repetitively (e.g., for hyper-parameter tuning) while severely under-utilizing the hardware. This is because DL researchers and practitioners often lack the required expertise to independently optimize their own workloads. Fortunately, we observe that such workloads have the following unique characteristics: (i) the models among jobs often have the same types of operators with the same shapes, and (ii) the inter-model horizontal fusion of such operators is mathematically equivalent to other already well-optimized operators. Thus, to help DL researchers and practitioners effectively and easily improve the hardware utilization of their novel DL training workloads, we propose **H**orizontally **F**used **T**raining **A**rray (HFTA). HFTA is a new DL framework extension library that *horizontally fuses* the models from different repetitive jobs deeply down to operators, and then trains those models simultaneously on a shared accelerator. On three emerging DL training workloads and state-of-the-art accelerators (GPUs and TPUs), HFTA demonstrates strong effectiveness on squeezing out hardware utilization and achieves up to $15.1\times$ higher training throughput vs. the standard practice of running each job on a separate accelerator.

1 Introduction

Deep Learning (DL) algorithms have facilitated tremendous progress in a range of domains, including natural language translation (Wu et al., 2016), recommendation systems (Naumov et al., 2019), magnetic resonance imaging segmentation (Akkus et al., 2017), video game bots (OpenAI, 2018), real-time high-resolution rendering (NVIDIA, 2020e), and very-large-scale integrated circuit placement (Lin et al., 2019). This is driven by the abundant and continuous efforts in researching and developing novel DL models by both academia and industry in recent years. Developing these models is computationally intensive, requiring an army of expensive, specialized accelerators such as GPUs and TPUs (Jouppi et al., 2017), leading to staggeringly high training costs (Amodei et al., 2018; Coleman et al., 2017; Zhu et al., 2018; Mattson et al., 2020; Zhu et al., 2020).

To reduce this training cost and optimize the cluster-wide hardware resource usage, we analyze GPU usage statistics over two consecutive months on a large GPU cluster from the Vector Institute (Vector Institute, 2021). We observe

that, despite significant attention on optimizing DL training workloads from the computer system and architecture communities, especially on distributed training optimizations (Appleyard et al., 2016; Chen et al., 2016; Lin et al., 2018; Rajbhandari et al., 2019; Mattson et al., 2020), single-accelerator (e.g., single-GPU) training jobs, often launched repetitively by DL researchers (to perform *hyper-parameter tuning*, *model architecture search* or *convergence stability tests*), can (i) dominate the cluster-wide hardware resource consumption (e.g., 46.2% in our study) while (ii) having extremely low hardware utilization (Section 2.1 and 5.3).

The root cause of this phenomenon is manifold. DL researchers and practitioners often lack the expertise to independently optimize their own training workloads. As a result, basic techniques, such as increasing the batch size, often become the only approach at their disposal to improve hardware utilization. However, this technique can be impractical due to many reasons including generalization gap (Keskar et al., 2017), batch size scaling limit (Shallue et al., 2019), and GAN training instability (Odena, 2019). On the other hand, accelerators (e.g., GPUs and TPUs) evolve towards more computing power and larger memory capacities (Table 2 and 3), and this trend amplifies the severity of the hardware under-utilization caused by the inability of such training workloads to scale their performance well.

Thus, this phenomenon motivates *hardware sharing* ap-

---

[*]Equal contribution [1]NVIDIA [2]Vector Institute [3]Department of Computer Science and Engineering, Shanghai Jiao Tong University [4]Intel [5]Department of Computer Science, University of Toronto. Correspondence to: Shang Wang <wangsh46@cs.toronto.edu>.

proaches. To the best of our knowledge, the only widely used hardware-based sharing solutions applicable to DL training are the MPS (NVIDIA, 2020h) and MIG (NVIDIA, 2020g) features on NVIDIA GPUs. However, as we later show in Section 2.2, these generic GPU sharing features that aim at arbitrary workloads are far from the "silver bullets" to effectively improve the hardware utilization in the case of repetitive single-GPU training workloads. The situation is even worse for emerging DL accelerators (e.g., TPUs) that currently do not have any hardware-based sharing features.

To address such hardware under-utilization on a variety of accelerators, we make two key observations based on the unique characteristics of these workloads. First, the models across jobs belonging to the same workload (e.g., hyper-parameter tuning) often have the *same types* of operators with the *same shapes*. Second, if these operators are horizontally fused across the models, the outcome is mathematically equivalent to other well-optimized operators found in existing DL framework stacks and accelerators (e.g., fusing multiple convolution operators can be realized using grouped convolutions). Inspired by these key observations, we propose to horizontally merge multiple training jobs with the same or similar DL models by deeply *fusing* most, if not all, operators in those models. The training of these models is then performed collectively on the same shared accelerator (instead of training each model separately on its own accelerator). Our proposed idea of *inter-model horizontal fusion* is drastically different from and also more effective than major related prior works as it better exercises the full potential of modern accelerators while (i) not relying on the generic sharing primitives (e.g., CUDA streams) that are ineffective for repetitive single-GPU workloads, and (ii) avoiding limited fusion techniques that, for example, support only stateless operators or require the weights across models to be the same (Narayanan et al., 2018a).

We leverage this novel idea to build a new DL framework extension library for DL researchers and practitioners, called **H**orizontally **F**used **T**raining **A**rray (**HFTA**), that greatly simplifies the adoption of our proposed inter-model horizontal fusion technique. In summary, this work makes the following major contributions.

• To understand the nature of the jobs running on modern DL accelerator clusters, we collect and study GPU cluster usage statistics, including 51K jobs running for 472K GPU hours in total, from real research workloads. The results of this study demonstrate that repetitive single-accelerator training jobs (i) dominate the hardware resource usage (i.e., 46.2%) and (ii) have extremely low hardware utilization.

• Motivated by this study, we make two key observations about these jobs that our proposal is built upon: (1) The models often have the same types of operators with the same shapes. (2) The inter-model horizontal fusion of such

operators is mathematically equivalent to other existing and well-optimized operators.

• We develop HFTA, a new library that helps DL researchers and practitioners (even with limited computer system and architecture expertise) to easily extract better performance from their hardware when training novel DL models. While doing so, we avoid (i) the introduction of any additional device-specific operator implementations that would limit the generality of our idea across different accelerators and (ii) any affect on individual models' convergence as the speedup is achieved only through mathematically equivalent transformations. HFTA is applicable to a wide variety of models, and can run on any hardware backends supported by existing DL frameworks.

• We evaluate HFTA on the PointNet (Xia, 2019) classification and segmentation tasks (ShapeNet part (Yi et al., 2016) dataset), and on DCGAN (Radford et al., 2016) (LSUN (Yu et al., 2015) dataset), which are examples of highly impactful DL models in the machine learning (ML) community, but not yet fully investigated/optimized by the experienced system engineers and computer architects.[1] On the modern GPUs (V100, RTX6000, and A100), HFTA achieves $3.63\times$ to $11.50\times$ higher training throughput than running the training jobs without sharing which is commonly employed by hyper-parameter tuning frameworks (Weights&Biases, 2020), $1.33\times$ to $4.72\times$ than MPS and $1.33\times$ to $4.88\times$ than MIG. HFTA can also fit $1.50\times$ to $7.57\times$ more training jobs on the same GPU than MPS. On TPUs, which currently do not have hardware sharing support, HFTA achieves $4.93\times$ to $15.13\times$ higher training throughput, which demonstrates HFTA's general ability to significantly improve performance across different hardware backends.

2 BACKGROUND AND MOTIVATION

2.1 Inefficiency in Repetitive Training Jobs

As DL research continues to evolve in recent years, the accompanied training cost has been increasing dramatically. For example, (Amodei et al., 2018) shows that the amount of compute for training SOTA DL models doubles every 3.4 month, outpacing even Moore's Law (Schaller, 1997). Motivated by the practical goal of reducing cluster-wide training cost, using the methodology detailed in Appendix A, we collect and study the GPU usage statistics of real research workloads for two consecutive months on a large GPU cluster from the Vector Institute (Vector Institute, 2021). To our surprise, we find that single-accelerator training jobs dominate the cluster-wide hardware resource consumption when these jobs are launched repetitively in groups, and the aggregated cost of these jobs can even outweigh that of distributed training (the primary focus of many research

---

[1] As opposed to the models from the MLPerf Training Benchmark suite (Mattson et al., 2020) that are intensively optimized.

Table 1. GPU hour usage breakdown for two consecutive months of a large GPU cluster from the Vector Institute.

Training Jobs	Repetitive Single-GPU	Isolated Single-GPU	Distributed	Other
GPU Hours	**218K(46.2%)**	19K(3.5%)	113K(24.0%)	124K(26.3%)

Table 2. Cloud TPU Core Specifications (Google, 2020c)

TPU	v2 (2017)	v3 (2018)	v4 (2020?)[†]
MXUs	1	2	≥ 4 ?
Memory (HBM)	8 GB	16 GB	? GB

[†] TPU v4 is expected to double the FLOPs of TPU v3 along with other enhancements (Kumar, 2020).

Table 3. NVIDIA Data Center GPU Specifications

GPU	SMs	HBM (GB)	HBM Bandwidth	TC Types
P100 (2016)	56	12/16	549/732 GB/s	-
V100 (2018)	80	16/32	900 GB/s	FP16
A100 (2020)	108	40	1.6 TB/s	TF32 & FP16

efforts from the computer system and architecture communities (Lin et al., 2018; Jayarajan et al., 2019; Rajbhandari et al., 2019; Mattson et al., 2020; Li et al., 2020)). Potential reasons of these repetitive jobs include (but are not limited to) hyper-parameter tuning (Strubell et al., 2019) and convergence stability testing.

Background *Hyper-parameter tuning* finds the optimal set of hyper-parameters unknown a priori, which are usually necessary for building accurate models targeting a previously unexplored problem (Bergstra & Bengio, 2012; Bergstra et al., 2011). Typical hyper-parameters include learning rates, the choices of weight initializers, and optimizer settings. *Model architecture search* (Elsken et al., 2019) is a subset of hyper-parameter tuning where the hyper-parameters directly impact the model architecture (e.g., the number of layers). *Convergence stability testing* trains the same model many times with different random seeds to verify the final accuracy results.

In our study, we classify the jobs into four main categories: (1) multi-node or single-node distributed training, (2) repetitive single-GPU training, (3) isolated single-GPU training, and (4) others (meaning the jobs that do not belong to the first three categories or can not be identified). Table 1 shows the distribution of the GPU hour usage among these categories, from which we can observe that the repetitive single-GPU training jobs consume as much as 46.2% of the cluster-wide total GPU hours. Furthermore, those repetitive single-accelerator training jobs often have low hardware utilization as we show in Appendix A. The cause of such phenomenon is manifold:

● Improving the hardware utilization for DL training jobs can be very challenging. DL researchers and practitioners often lack the system and architecture expertise to optimize their training workloads on their own. Increasing the batch size, which is the naïve and often the only approach at their disposal to increase hardware utilization, is not universally applicable. For instance, large batch sizes can lead to training instability for generative adversarial network (GAN) (Odena, 2019; Brock et al., 2019), generalization gap (Keskar et al., 2017), and diminishing returns due to batch size scaling limit (Shallue et al., 2019). Even with the help from computer system and architecture experts, applying various advanced optimization techniques (e.g., kernel fusion (Appleyard et al., 2016) or checkpointing (Chen et al., 2016; Zheng et al., 2020)) on each new model requires an enormous amount of engineering efforts (Mattson et al., 2020). Meanwhile, novel DL models are being proposed at the exponential pace in recent years (Charrez, 2019).

● As DL research progresses, accelerators (e.g., GPUs and TPUs (Jouppi et al., 2017)) evolve towards more compute power (e.g., more streaming multiprocessors (SMs) and the introduction of specialized compute units for fast matrix multiplications in GPUs called tensor cores (TCs) (Markidis et al., 2018)) and larger memory capacity/bandwidth. We can observe this trend from Tables 2 and 3 that list the specifications of the most recent NVIDIA data center GPUs and Cloud TPUs, where the largest accelerators suffer from under-utilization the most.

The fast development of both new DL models and accelerators together exacerbates the hardware under-utilization from repetitive single-accelerator training jobs, which motivates hardware sharing methods discussed below.

2.2 Hardware-based Sharing

The most well-known and (to the best of our knowledge) the only widely-used hardware-based sharing solutions applicable to DL training workloads[2] are the Multi-Process Service (MPS) (NVIDIA, 2020h) and Multi-Instance GPU (MIG) (NVIDIA, 2020g) on NVIDIA GPUs. MPS allows CUDA kernels from different processes to potentially run concurrently on the same GPU via a hardware feature called Hyper-Q (Bradley, 2007). MIG, which is currently only available on the most recent A100 GPUs (NVIDIA, 2020a), partitions a single GPU into multiple (up to 7) isolated GPU instances (GIs) where each job now run on a single GI.

However, as we quantitatively demonstrate in Section 5.1, both MPS and MIG still leave significant potential of training performance unharnessed due to the following reasons. First, both MPS and MIG duplicate the runtime overhead among kernels from different training jobs, including kernel launches (Lustig & Martonosi, 2013), GEMM setups and teardowns (NVIDIA, 2020j), and/or memory format conversions (specifically related to TCs) (NVIDIA, 2020f). Thus, they can not effectively improve the SM and TC utilization. Second, both MPS and MIG require running training jobs as separate processes which duplicates the GPU memory overhead reserved by the DL framework stack (Gross et al., 2019) and leads to a higher overall GPU memory footprint.

---

[2] AMD GPUs also have a hardware-based sharing feature called CU-mask (Otterness & Anderson, 2020); however, we skip its discussion due to their irrelevance in mainstream training workloads.

Therefore, we can fit fewer training jobs into the same GPU. Finally, MIG's partitioning granularity can be too coarse for many training workloads. Even with the finest granularity of MIG (7 GIs), each job can still under-utilize a single GI.

2.3 Prior Works

Major prior works on DL job fusion (Liu et al., 2020; Narayanan et al., 2018b;a) suffer from three key weaknesses: (i) avoiding directly addressing hardware under-utilization, (ii) strongly depending on the CUDA stream primitive (Harris, 2015) that is a generic GPU-sharing method but inefficient for repetitive training jobs, and (iii) employing very restricted fusion schemes that are ineffective in practice. We discuss these prior works in detail below.

`pack` (Liu et al., 2020) merges TensorFlow (Abadi et al., 2016) graphs from multiple training jobs into a single graph in order to amortize *only* the IO and data preprocessing cost, but does not address the hardware under-utilization from the model forward and backward passes.

In addition, ModelBatch (Narayanan et al., 2018b) attempts to parallelize the kernel launches from multiple training jobs via CUDA streams (the CUDA programming interface of Hyper-Q), which suffers from similar pitfalls of runtime overhead duplication as MPS.

Although *intra*-model vertical and horizontal fusion of DL operators have been studied extensively by many prior works (Appleyard et al., 2016; Gray et al., 2017; Vasilache et al., 2018; Rotem et al., 2018; Chen et al., 2018; Jia et al., 2019), *inter*-model horizontal fusion has only been explored in extremely limited depth: HiveMind (Narayanan et al., 2018a) proposes fusion schemes for 1) non-stateful operators with the same shapes, 2) stateful operators that share the same weights, and 3) stateful operators that share the same shapes and inputs. Unfortunately, condition 2) is rarely applicable to training workloads since each individual model has its own weights, while condition 3) usually only applies to the first operator in a DL model since the following operators will have different inputs, leaving most of fusion opportunities completely untapped. In addition, HiveMind does not demonstrate any performance improvement over MPS as it also relies on CUDA streams to extract utilization when its fusion scheme becomes ineffective. Therefore, HiveMind approach is hard to generalize to accelerators with no hardware-specific sharing features (e.g., TPUs).

In contrast, our proposal, HFTA, is able to fuse any operators of the same types that share the same shapes across training jobs, which generally leads to full inter-model fusions. Moreover, HFTA demonstrates significant performance improvement against the existing widely-adopted generic hardware-based sharing approaches (e.g., MPS and MIG) since operator fusion does not possess the same shortcomings of those approaches, as we show in Section 2.2.

Finally, HFTA requires no hardware or DL framework stack modifications, and is also applicable to any existing hardware backends including GPUs, TPUs, and any other accelerators that the major DL frameworks support.

3 OUR PROPOSAL: HFTA

To address the challenge of improving hardware utilization for novel repetitive training workloads on a variety of accelerators, we make the following two key observations on the unique characteristics of these workloads:

• When launched repetitively (such as during hyperparameter tuning or convergence stability testing), the models used across these jobs often have the *same types* of operators with the *same shapes*.

• *Horizontally fusing* the same types of operators with the same shapes often results in other mathematically equivalent operators that already exist in many SOTA DL models and thus have been optimized in most DL framework stacks on different accelerators.

Figure 1 explains the above observations with a concrete example of hyper-parameter tuning where the goal is to determine which weight initializer and learning rate work the best. Regardless of which weight initializer or learning rate is used, the first operators in both models are Conv2d of the same shape; the horizontal fusion of many Conv2d operators is mathematically equivalent to a grouped Conv2d which is already used in the ResNeXt (Xie et al., 2017) and MobileNets (Howard et al., 2017) models and supported by cuDNN (NVIDIA, 2020c) on NVDIA GPUs and XLA (Google, 2020e) on TPUs.

Inspired by the above observations, instead of the common practice (Li, 2020) of running each job with a single model on a separate accelerator, we propose to better utilize existing hardware by deeply fusing the the same (class of) models across multiple jobs together. Most, if not all, operators of these models can be *horizontally fused*, and we train these models simultaneously on the same accelerator. Thus, as depicted in Figure 1, we can fuse many training jobs into a single one, without the need to implement any new device-specific operator from scratch which is both time consuming and error-prone. Moreover, this approach easily generalizes to any hardware backends that the DL frameworks support (e.g., with PyTorch, we can already support all NVIDIA GPUs and Google TPUs). Since horizontal operator fusion can be performed for both single-accelerator and distributed training, our approach is applicable to both use cases.

However, manually implementing or porting existing training workloads to the fused ones from scratch can be challenging for DL researchers and practitioners. To greatly simplify the associated engineering efforts, we develop a new DL framework library called **H**orizontally **F**used **T**raining **A**rray (**HFTA**). Even though we choose PyTorch (Paszke
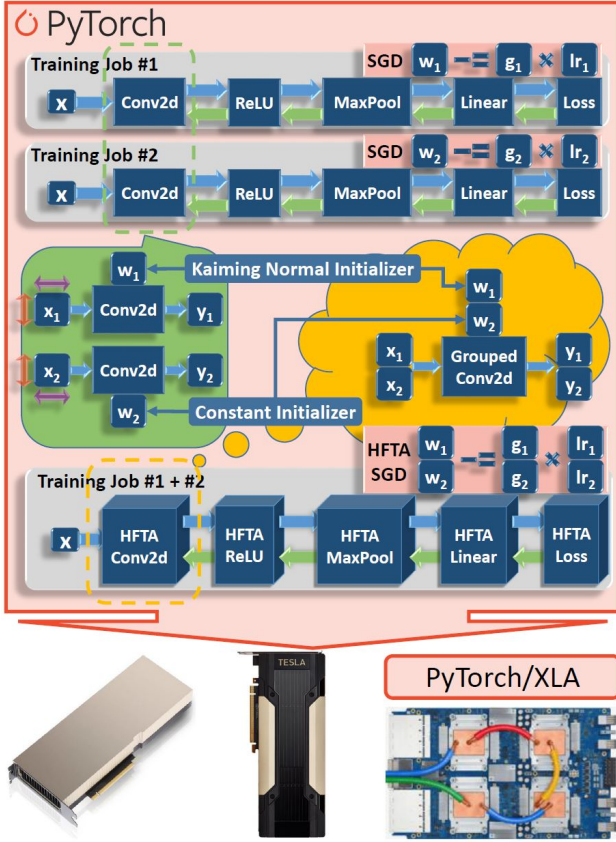
Figure 1. An example showing the key idea of HFTA where two training jobs for hyper-parameter tuning are fused into one via inter-model horizontal operator fusion.

et al., 2019) as our prototyping DL framework due to its user friendliness and increased popularity within the ML community (He, 2019), the same idea can be implemented on top of other DL frameworks (e.g., TensorFlow (Abadi et al., 2016) and MXNet (Chen et al., 2015)). Also, HFTA is carefully designed to accommodate computer system and architecture "novices". It can be used seamlessly with PyTorch-native training scripts, and only requires changing very few lines of code. As an illustrative example, Figure 2 shows how to enable HFTA for AlexNet (Krizhevsky et al., 2012). We can observe that the model definition is kept exactly the same with only a few extra lines of code (highlighted in the red box) to update the PyTorch's operator classes.

We now discuss the HFTA's individual components (Section 3.1), and then demonstrate both theoretically (Section 3.2) and empirically (Section 3.3) that HFTA has no impact on individual models' convergence.

3.1 HFTA Operators and Optimizers

To relieve the DL researchers and practitioners from the need to implement any horizontally fused operators themselves, HFTA covers most common operators used in DL research and development (with detailed fusion rules provided in Appendix B). For example, the fusion of operators from the (de)convolution family (e.g., Conv1d or ConvTrans-

```
B = len(args.lrs)  # Number of fused training jobs.
nn.Conv2d = get_hfta_op_for(nn.Conv2d, B=B)
nn.ReLU = get_hfta_op_for(nn.ReLU, B=B)
nn.MaxPool2d = get_hfta_op_for(nn.MaxPool2d, B=B)
nn.Dropout = get_hfta_op_for(nn.Dropout, B=B)
nn.Linear = get_hfta_op_for(nn.Linear, B=B)
# Model definition.
class AlexNet(nn.Module):
  def __init__(self, ...):
    super(AlexNet, self).__init__()
    self.features = nn.Sequential(
      nn.Conv2d(...), nn.ReLU(...), nn.MaxPool2d(...),
      ...)
    self.classifier = nn.Sequential(
      nn.Dropout(...), nn.Linear(...), nn.ReLU(...),
      ...)
```

Figure 2. How to enable HFTA for AlexNet.

pose2d) can be replaced by their grouped (de)convolution counterparts, and the fusion of linear layers can be replaced by the baddbmm operator.

In addition, HFTA supports inter-model horizontally fused optimizers (e.g., Adam (Kingma & Ba, 2015) and Adadelta (Zeiler, 2012)) and learning rate schedulers (e.g., StepLR (Senior et al., 2013)). This is because (1) hyper-parameter tuning is a common use case in repetitive training workloads, and (2) learning rates, learning rate schedules, and optimizer settings (e.g., momentum (Qian, 1999; Sutskever et al., 2013)) are common hyper-parameters that require tuning for many DL models. The scalar-vector operations (e.g., multiplying a learning rate under tuning with the gradients) in the original implementations are now replaced by broadcasted vector-vector operations (e.g., multiplying a vector of learning rates with the concatenated gradients of all models) in HFTA's implementations (as depicted in Figure 1). We also plan to continue improving the HFTA coverage to support more operators, optimizers, and learning rate schedulers beyond the publication of this work.

3.2 Scaling of Fused Loss

We now show how loss fusion is handled in order to reconstruct mathematically equivalent gradients. The inter-model horizontally fused loss with *mean reduction* is shown as:

$$\mathcal{L} = \frac{1}{B} \sum_{b=0}^{B} \ell_b \qquad (1)$$

where ℓ_b is the loss of the b-th model, and there are B models in total contributing to the fused loss $\mathcal{L}$. Taking the gradients on both side of Equation 1 with respect to the parameters $\vec{\theta}_\beta$ of a specific model β results in:

$$\nabla_{\vec{\theta}_\beta} \mathcal{L} = \frac{1}{B} \nabla_{\vec{\theta}_\beta} \sum_{b=0}^{B} \ell_b = \frac{1}{B} \sum_{b=0}^{B} \nabla_{\vec{\theta}_\beta} \ell_b = \frac{1}{B} \nabla_{\vec{\theta}_\beta} \ell_\beta \quad (2)$$

because $\nabla_{\vec{\theta}_\beta} \ell_b = 0$ if $b \neq \beta$. We can rearrange Equation 2 into:

$$\nabla_{\vec{\theta}_\beta} \ell_\beta = B \nabla_{\vec{\theta}_\beta} \mathcal{L} = \nabla_{\vec{\theta}_\beta} B \mathcal{L} \qquad (3)$$

We can recognize that the expression on the left hand side of Equation 3 is exactly the gradients for model β if each model were trained independently. Therefore, in order to reconstruct exactly the same gradients when training via
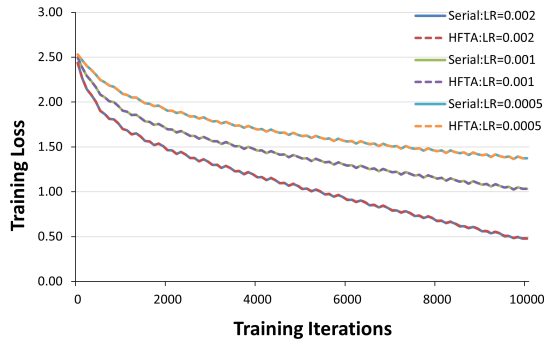
Figure 3. Training loss per iteration when training ResNet-18 on CIFAR-10. *LR* represents the learning rate. *Serial* represents training each model separately, and *HFTA* represents our method.

HFTA, the final fused loss $\mathcal{L}$ needs to be scaled by B. Similarly for fused loss with *sum reduction*, we can derive that such scaling is no longer needed. In these derivations, no assumption is made on the exact formula of ℓ_β, which means such scaling rules are universal to any types of loss functions including regularization.

3.3 Effect on Convergence

Even though HFTA reconstructs the mathematically equivalent gradients for each independently trained model, minor numerical differences can still exist since the order of computations in fused operators can be different from the original ones. To demonstrate that such numerical differences do not affect the models' original convergence empirically, we train a well-known ResNet-18 (He et al., 2016) model on the CIFAR-10 (Krizhevsky, 2009) dataset with three different learning rates.[3] Figure 3 shows the training-loss-per-iteration curves for both training each model independently (solid lines) and HFTA (dotted lines). Since the dotted curves overlap completely with the solid ones, we conclude that HFTA-based training maintains exactly the same convergence as independent model training.

4 METHODOLOGY

Workloads Our benchmarks are carefully selected based on the following three criteria. First, our workloads should represent important models in their corresponding DL subfields, making sure that HFTA is effective in improving the hardware utilization for important DL models. Second, we select models that have not yet received much attention from the computer system and architecture communities and hence are not over-optimized. This is a much more realistic scenario for DL researchers and practitioners who typically lack the expertise to apply advanced optimization techniques. Third, we would like to cover both compute-bound and memory-bound DL models. Based on the aforementioned criteria, two classes of models (three different workloads) are selected as our major benchmarks.

PointNet (Qi et al., 2017) is a memory-bound neural network that performs (i) object classification and (ii) segmentation tasks on 3D point clouds. The models for both tasks are trained on the ShapeNet part dataset (Yi et al., 2016). We leverage a third-party PyTorch implementation of Point-Net (Xia, 2019) that is endorsed by Qi et al. (Qi, 2017).

DCGAN (Radford et al., 2016) is a compute-bound generative adversarial network (GAN) that synthesizes natural-apparent images. The model is trained on the LSUN dataset (Yu et al., 2015). We leverage an implementation of DC-GAN from PyTorch official examples (PyTorch, 2020).

To emulate the hardware usage habits of DL researchers and practitioners without the influence from the computer system and architecture experts, the batch sizes used in both benchmarks are kept the same as reported in their corresponding publications. To empirically prove that HFTA does not affect convergence and to demonstrate that HFTA can improve the hardware utilization for conventional models, we train ResNet-18 (He et al., 2016) on V100 with the CIFAR-10 (Krizhevsky, 2009) dataset using Adadelta (Zeiler, 2012) with a batch size of 1000.

Experimental Setup Our experiments are performed on two types of ML accelerators (NVIDIA GPUs and Google TPUs) including the most recent three generations of GPUs and the latest available generation of TPUs: (i) Volta-based V100 (NVIDIA, 2020k), (ii) Turing-based RTX6000 (NVIDIA, 2020i), and (iii) very recent Ampere-based A100 (NVIDIA, 2020a),[4] (iv) TPU v3 (Google, 2020a). We provide the detailed specifications in Table 4.

Baselines We use *hyper-parameter tuning* (including learning rate, learning rate schedule, and optimizer settings) as the use case for our repetitive single-accelerator training jobs under experimentation. We compare HFTA with the following four SOTA baselines. (1) *Serial*: each training job is executed on a single accelerator. This scheme is employed by most hyper-parameter tuning frameworks (Weights&Biases, 2020; Li, 2020). (2) *Concurrent*: multiple training jobs are executed as independent processes on the same GPU. In this case, the kernels from the processes are *time-multiplexed*, but can *not* execute concurrently on the same GPU (without the help of MPS or other hardware features). This scheme is used when MPS is not preferable due to infrastructure and/or security related reasons (e.g., custom-built infrastructure or CUPTI tools that are not compatible with MPS). (3) *MPS*: similar to *concurrent*, except the independent processes are executed via MPS. (4) *MIG*: similar to *concurrent*, except the independent processes are executed via MIG. This scheme is currently only available on the A100 GPUs. We use *concurrent*, *MPS*, and *MIG* only on GPUs since TPUs do not support running concurrent processes as of now. We

---

[3]We provide the detailed methodology behind this and other experiments in Section 4.

[4]Using GCP A2 Alpha version instances.

Table 4. Specifications of our experiment platforms. *Dev. Mem.* and *VM/Host Mem.* stands for device memory and VM/host memory respectively in GB. *CSP* stands for cloud service provider.

Accelerator	Dev. Mem.	CSP	VM Instance	CUDA	cuDNN	GPU Driver	PyTorch	PyTorch/XLA	(v)CPUs	VM/Host Mem.
V100	16	AWS	p3.2xlarge	10.2	7.6.5	450.51.05	1.6.0	-	8	61
RTX6000	24	-	-	10.2	7.6.5	450.66	1.6.0	-	8	16
A100	40	GCP	a2-highgpu-1g	11.0.3	8.0.2	450.51.06	1.7.0a0+8deb4fe	-	12	85
TPU v3	16	GCP	n1-highmem-8	-	-	-	1.7.0a0+626e410	1.6+8af57fb	8	52

do not evaluate HiveMind (Narayanan et al., 2018a) since it is both close-sourced and implemented on a different ML framework (TensorFlow). We provide the detailed qualitative comparison against HiveMind in Section 2.3.

Metrics We use the *per-device training throughput* as our key performance metric to compare HFTA against our baselines since HFTA has no impact on the model convergence.[5] We calculate this throughput by measuring the end-to-end training latency of: (i) 10 epochs for both classification and segmentation tasks on PointNet; and (2) 5 epochs, 1000 iterations per epoch on DCGAN (enough for these workloads to enter the execution steady state). We skip the first epoch on GPUs and the first two epochs on TPUs to properly warm up the hardware before making any measurements. We repeat each experiment at least three times and report the average, minimum, and maximum per experiment.

In order to measure the effect of each technique on the hardware utilization, we use the `sm_active` and `sm_occupancy` performance counters that represent the SM temporal and spatial utilization respectively, and the `tensor_active` performance counter to measure the TC temporal utilization (NVIDIA, 2020d). Details on these performance counters can be found in Appendix C.

5 EVALUATION

Our evaluation results are thoroughly analyzed here, including end-to-end training performance on GPUs (Section 5.1) and TPUs (Section 5.2), as well as GPU hardware performance counters to explain why HFTA achieves significantly better training performance (Section 5.3).

5.1 End-to-end Training Performance on GPUs

V100 Results To compare the HFTA's end-to-end training performance with other alternatives (i.e., *serial*, *concurrent*, *MPS*), Figure 4a, 4b and 4c plot the per-GPU normalized training throughput on the V100 GPUs (Volta architecture (NVIDIA, 2017)) with the PointNet classification task, PointNet segmentation task, and DCGAN respectively. We normalize the throughput for each experiment by the respective FP32 *serial* baseline. For each experiment, we show both FP32 and AMP (Huang et al., 2020) training results. Each curve grows as we increase the number of models that either co-run together (for the *concurrent* and *MPS* baselines) or run in the fused form with HFTA. Each curve "stops" when it reaches the maximum number of mod-

---

[5]As we theoretically justified in Section 3.2 and empirically demonstrated in Section 3.3.

els before the GPU runs out of memory. Based on these figures, we make several major observations:

First, *HFTA achieves significantly higher peak throughput than all baselines*; specifically, $4.29\times$ to $5.02\times$ over *serial*, $2.01\times$ to $4.87\times$ over *concurrent* and $2.03\times$ to $4.50\times$ over *MPS*. The significant throughput improvement is due to a much higher achieved utilization in both compute cores (details in Section 5.3) and GPU memory (discussed in the next observation).

Second, *HFTA enables more models to share the same GPU than MPS and concurrent*; specifically, up to $1.80\times$ on the PointNet classification task, up to $1.60\times$ on the segmentation task and up to $7.57\times$ on DCGAN. This is because HFTA does not duplicate the GPU memory overhead as we explain in Section 5.3.

Third, *as we increase the number of models sharing the same GPU, the throughput of HFTA scales up and, in some cases, plateaus eventually*. This is because using HFTA, the SM and TC utilization increases with the number of co-executing models (as we explain in Section 5.3). In contrast, *MPS* and *concurrent* either (i) plateau at a smaller number of models with a lower throughput as we observe in Figure 4a and 4b, or (ii) even experience performance degradation as we observe in Figure 4c due to host resource (e.g., CPUs, disk I/O bandwidth, and/or memory) contention among many training processes.

Fourth, *even with the same number of models sharing the same GPU, HFTA often achieves higher throughput than all baselines*. The maximum speedups range from $1.62\times$ to $3.41\times$ over *concurrent* and $1.17\times$ to $3.05\times$ over *MPS*.

Fifth, *HFTA can better exploit computation power from advanced hardware features such as TCs used during AMP training compared to the baselines*. Specifically, the maximum speedup of AMP training over FP32 is $2.65\times$ with HFTA, but only $1.00\times$ for *serial*, $1.07\times$ for *concurrent*, and $1.06\times$ for *MPS*.

Therefore, we conclude that HFTA can significantly outperform major hardware-based sharing alternatives in improving hardware utilization and, as a result, improve the throughput of emerging ML models during repetitive single-accelerator training.

RTX6000 and A100 Results To check whether HFTA's significant performance gains are general across different GPU architectures (e.g., Turing (NVIDIA, 2018) and Ampere (NVIDIA, 2020b)), we conduct the same set of experi-
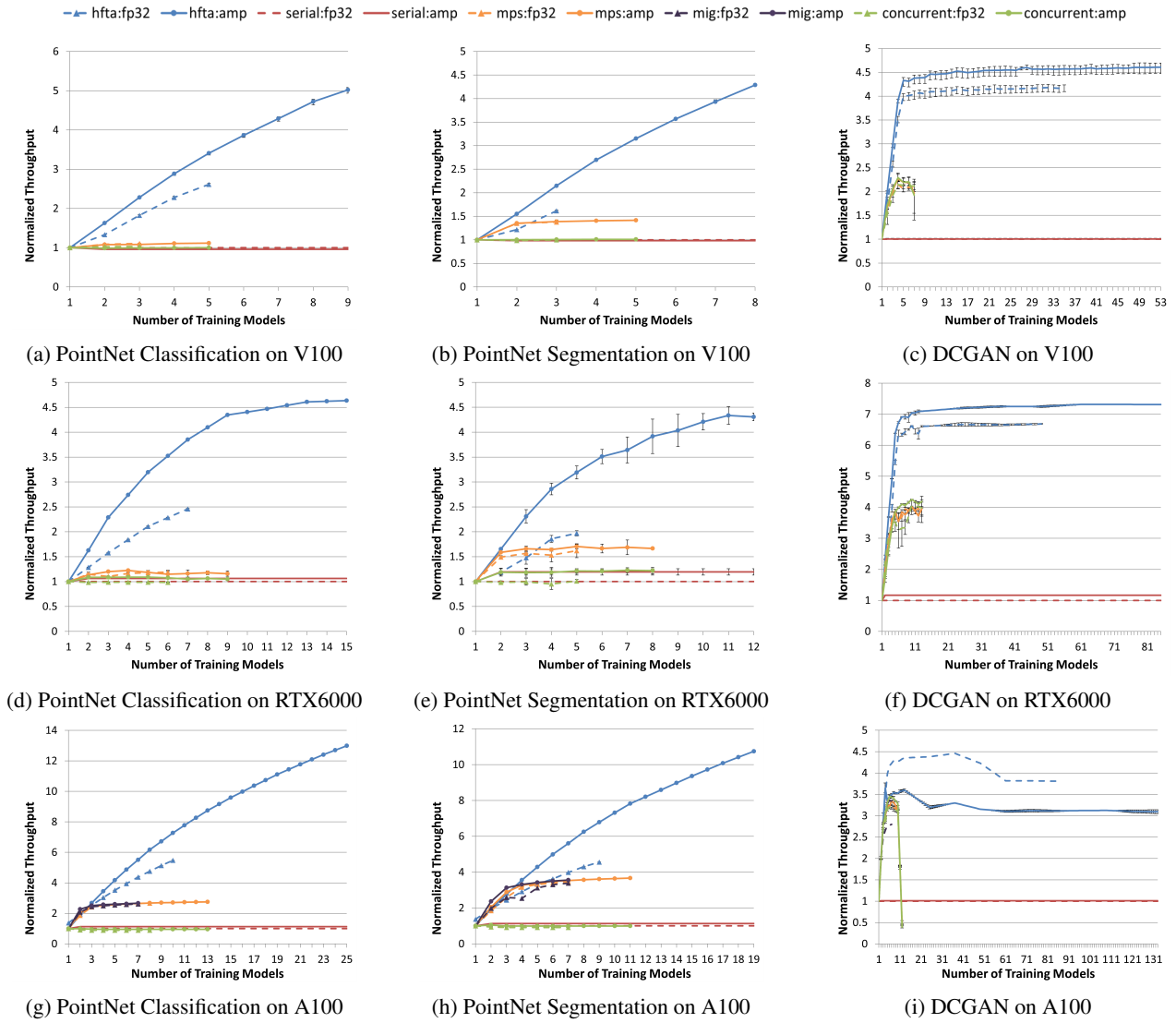
(a) PointNet Classification on V100

(b) PointNet Segmentation on V100

(c) DCGAN on V100

(d) PointNet Classification on RTX6000

(e) PointNet Segmentation on RTX6000

(f) DCGAN on RTX6000

(g) PointNet Classification on A100

(h) PointNet Segmentation on A100

(i) DCGAN on A100

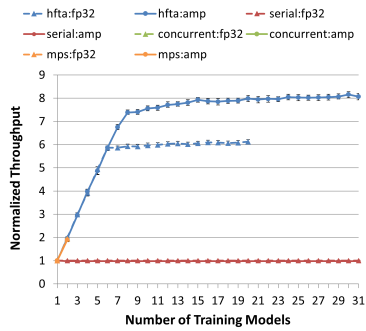*Figure 4.* The normalized training throughput as we increase the number of models sharing the same GPU.

Figure 5. The normalized training through-put of ResNet18 on V100 as we increase the number of models sharing the same GPU.
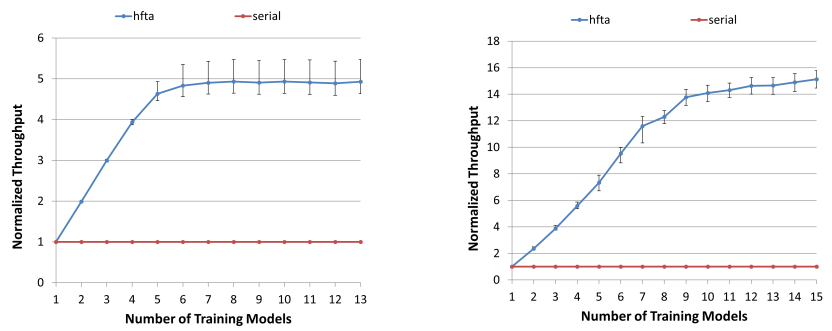
(a) PointNet Classification

(b) DCGAN

Figure 6. The normalized training throughput as we increase the number of models sharing (via HFTA) the same TPU v3 core.

ments on the RTX6000 (Figure 4d, 4e and 4f respectively) and the A100 (Figure 4g, 4h and 4i respectively) while adding the extra *MIG* baseline for the A100. The general trends in these figures are similar to those we observe for V100. To simplify the comparison, for each workload on each GPU, Table 5 presents the peak throughput speedups of HFTA over the baselines, while Appendix D presents (i) the maximum throughput speedups of HFTA over the baselines given a fixed number of models, and (ii) the maximum AMP training throughput speedups over FP32 for both HFTA and the baselines. In addition, we make the following new observations:

First, both RTX6000 and A100 have higher GPU memory (HBM) capacities than V100 (24 GB and 40 GB vs. 16 GB); therefore, *both HFTA and the baselines can co-run more models on the same RTX6000/A100 compared with V100.* For example, AMP training of PointNet classification task via HFTA can run up to 15/25 models with RTX6000/A100 vs. 9 on V100).

Second, since A100 has more compute capability and a larger GPU memory capacity than V100, the comparison of Figure 4g vs. 4a and 4h vs. 4b reveals that *HFTA not only fits more models on the same hardware, but also achieves a higher peak throughput speedup over the baselines on A100 than on V100* (e.g., for PointNet segmentation task, the peak throughput speedup over serial is as high as 9.48× on A100 vs. 4.29× on V100).

Third, we observe one anomaly in DCGAN training on A100 (Figure 4i) where HFTA's FP32 throughput is higher than that of AMP. After profiling the AMP run of this experiment via the PyProf (Agrawal & Kolodziej, 2020) tool, we pinpoint a few suspicious cuDNN-related FP32 kernels (which are supposed to be replaced by the equivalent TC kernels) in the backward pass. Since the Ampere architecture and the corresponding versions of cuDNN/PyTorch are very recently released, and we do not observe similar problems on older cuDNN/PyTorch versions for V100 and RTX6000, we believe that this issue is temporary due to the insufficient optimization in some of the new cuDNN kernels for A100. We hope it will be addressed in future cuDNN releases/fixes and we will be able to update the results accordingly.

Fourth, we notice that *on A100, the MIG partitioning (only up to 7 GIs) can be too coarse-grained*, as we observe in Figure 4g, 4h and 4i that both *MPS* and *concurrent* could often share the A100 with more than seven models.

Therefore, we conclude that HFTA's performance generally scales well with the compute and memory capabilities of modern GPUs. We observe higher performance benefits in the newer GPU architectures that would otherwise suffer more significantly from the hardware under-utilization when training without HFTA (as we qualitatively discuss in

Table 5. The peak training throughput speedups of HFTA over the baselines. For each experiment, the higher throughput between FP32 and AMP is used in the calculation. The detailed breakdown between FP32 and AMP is included in Appendix D.

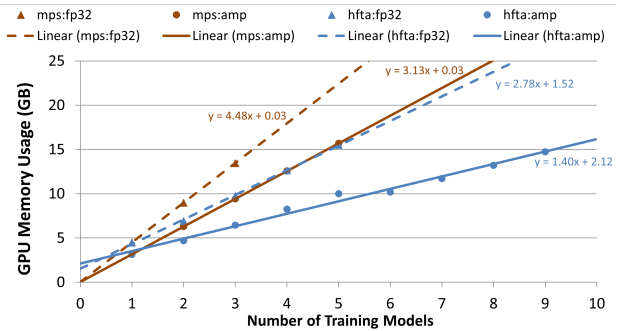| Benchmark		PointNet Classification	PointNet Segmentation	DCGAN
V100	*serial*	5.02	4.29	4.59
	concurrent	4.87	4.24	2.01
	MPS	4.50	3.03	2.03
RTX6000	*serial*	4.36	3.63	6.29
	concurrent	4.26	3.54	1.72
	MPS	3.79	2.54	1.82
A100	*serial*	**11.50**	9.48	4.41
	concurrent	**12.98**	10.26	1.29
	MPS	**4.72**	2.93	1.33
	MIG	**4.88**	3.02	1.33

Figure 7. GPU Memory Footprints of *MPS* and HFTA for PointNet classification task as we increase the number of models sharing the same V100.

Section 2.1 and empirically show in Appendix D).

End-to-end Performance for Conventional Models As a quick check of HFTA's effectiveness on improving hardware utilization for conventional training workloads, we measure the throughput for ResNet-18 training with the CIFAR-10 dataset on V100 and plot the results in Figure 5. Similar to the trends in Figure 4, we observe that HFTA achieves 8.16× higher peak throughput than *serial*, 4.21× than *concurrent*, and 4.18× than *MPS*. We conclude that HFTA is also efficient in improving throughput of the repetitive training for models outside of its original scope.

5.2 End-to-end Training Performance on TPUs

As we aim to build a general solution that works for different ML accelerators, we also evaluate HFTA on a completely different type of accelerator: Google TPU v3. Figure 6 plots the per-core training throughput for the *serial* baseline vs. HFTA on the PointNet classification and DCGAN experiments on TPU v3, normalized by the throughput of the respective *serial* baseline. Similarly to previous results on GPUs, each HFTA curve shows how the normalized throughput increases with the number of models sharing the same TPU (until the fused models can not fit into the TPU HBM memory). We make three major observations from these figures.

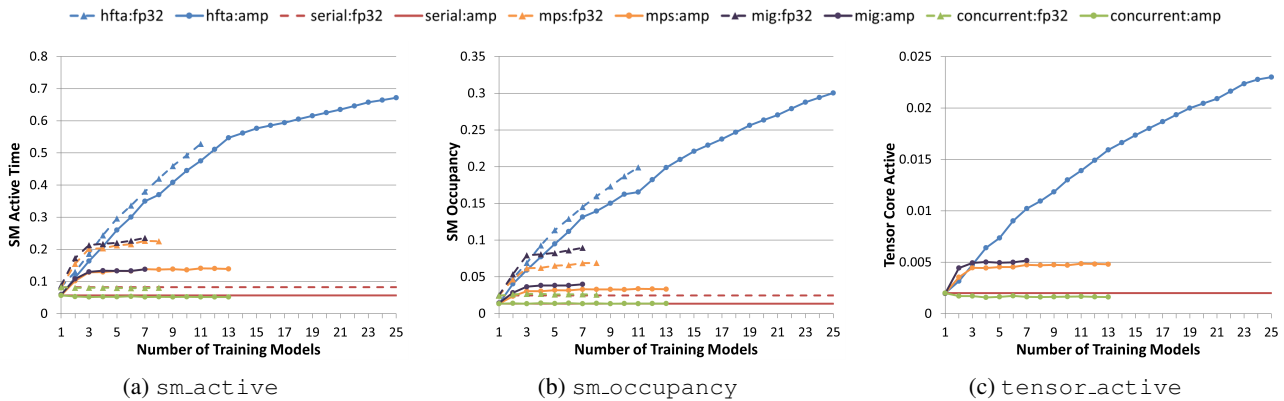First, *HFTA achieves* 4.93× / 15.13× *higher peak through-*

Figure 8. The hardware performance counters for PointNet classification task as we increase the number of models sharing the same A100.

put than serial on the PointNet classification / DCGAN.

Second, we observe that *for DCGAN, HFTA can sometimes achieve "super-linear" speedups.* Our current investigation concludes that the most likely cause of such a behaviour is the tensor padding added in the *serial* baseline by the XLA (Google, 2020b) compiler (Google, 2020d), making this baseline weaker than it should be otherwise.

Additionally, we also investigate the HFTA's potential on the PointNet segmentation task. Unfortunately, HFTA currently achieves a less impressive 1.20× speedup over the *serial* baseline, which we attribute to the PointNet segmentation variant having many non-GEMM-based operators that intrinsically do *not* map well to systolic arrays by the XLA compiler. Deeper analysis, however, is limited due to the xprof (Google, 2020f) tool, just recently released, do not directly support PyTorch/XLA. We will perform deeper analysis of this problem and research potential solutions as soon as a proper version of the profiler is released.

5.3 In-depth Performance Analysis

Using PointNet classification task as a case study, we perform deeper analysis through profiling GPU hardware performance counters to explain why HFTA is able to share the same GPU with more training workloads and achieves higher training throughput than the baselines.

Figure 7 plots the GPU memory footprint of *MPS* and HFTA as we increase the number of models sharing the same V100 GPU [6], as well as the linear regression lines fitted on those measurements. Training models in independent processes duplicates the associated GPU memory overheads (reserved by the DL framework stack (Gross et al., 2019)), which is a challenge that HFTA addresses. Thus, we can observe that: (1) *MPS*'s linear regression lines pass through the (0, 0) coordinate and have higher slopes than HFTA's; and (2) the intercepts of HFTA's linear regression lines essentially represent the exact amounts of memory overhead which are 1.52GB for FP32 training and 2.12GB for AMP.

Figure 8 plots the `sm_active`, `sm_occupancy`, and `tensor_active` of HFTA and the baselines as we increase the number of models sharing the same A100 GPU.[7] We can observe: (1) HFTA's SM and TC utilization keeps scaling up as we fuse more models horizontally. (2) *MIG*'s and *MPS*'s SM and TC utilization plateaus at a smaller number of models and lower utilization, which supports our qualitative reasoning in Section 2.2 that both leave significant potential of training performance unharnessed; (3) *concurrent*'s SM and TC utilization stays the same as *serial*, because the kernels from parallel processes can not execute concurrently without MPS or MIG.

6 CONCLUSION

In this work, we learn from "real-world" GPU cluster usage analysis that repetitive single-accelerator training jobs (e.g., for hyper-parameters tuning) often dominate cluster-wide hardware resource usage. These training jobs also tend to have low hardware utilization, since DL researchers and practitioners often lack the relevant expertise to independently optimize their own workloads. To address this challenge, we make the following observations on the unique characteristics of these jobs: (1) the models among such jobs often have the same types of operators with the same shapes; and (2) the inter-model horizontal fusion of such operators is mathematically equivalent to other already well-optimized operators. Built upon these observations, we propose the HFTA (DL framework extension) library that horizontally fuses the models deeply down to operators with minimal extra effort from DL researchers and practitioners, significantly improving the hardware utilization of these workloads by simultaneously training many models on the same accelerator. On the PointNet classification and segmentation tasks, and DCGAN, HFTA achieves up to 15.13× higher training throughput than running each job on a separate accelerator, and on GPUs, 4.72× than hardware-based sharing via MPS and 4.88× than MIG. We continue to expand the coverage of HFTA including more operators, optimizers, and learning rate schedulers, as well as

---

[6]The trends on RTX6000 and A100 are consistent with V100.

[7]V100 results are similar and shown in Appendix D.

integrating HFTA into existing hyper-parameter tuning and model architecture search frameworks. We hope our work can inspire future research on assisting ML researchers and developers with limited optimization experience to better utilize the hardware for their novel DL models.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283, 2016.

Agrawal, A. and Kolodziej, M. Pyprof: Automating end-to-end pytorch profiling. `https://developer.down load.nvidia.com/video/gputechconf/gt c/2020/presentations/s21143-automati ng-end-to-end-pytorch-profiling.pdf`, 2020. Accessed: 2020-09-17.

Akkus, Z., Galimzianova, A., Hoogi, A., Rubin, D., and Erickson, B. Deep learning for brain mri segmentation: State of the art and future directions. *Journal of digital imaging*, 30, 06 2017.

Amodei, D., Hernandez, D., Sastry, G., Clark, J., Brockman, G., and Sutskever, I. Ai and compute. `https:// openai.com/blog/ai-and-compute/`, 2018. Accessed: 2020-09-13.

Appleyard, J., Kociský, T., and Blunsom, P. Optimizing performance of recurrent neural networks on gpus. *CoRR*, abs/1604.01946, 2016.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13: 281–305, February 2012.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24 (NIPS)*, pp. 2546–2554, 2011.

Bradley, T. Hyper-q example. Technical report, 2007. Accessed: 2020-09-17.

Brock, A., Donahue, J., and Simonyan, K. Large scale GAN training for high fidelity natural image synthesis. In *7th International Conference on Learning Representations, (ICLR)*, 2019.

Charrez, D. Neurips 2019 stats. `https://medium.c om/@dcharrezt/neurips-2019-stats-c913 46d31c8f`, 2019. Accessed: 2020-09-17.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, October 2018.

Coleman, C. A., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. Dawnbench : An end-to-end deep learning benchmark and competition. 2017.

Elangovan, A. Optimizing i/o for gpu performance tuning of deep learning training in amazon sagemaker. `https: //aws.amazon.com/blogs/machine-learn ing/optimizing-i-o-for-gpu-performan ce-tuning-of-deep-learning-training- in-amazon-sagemaker/`, 2020. Accessed: 2020-10-09.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20:55:1–55:21, 2019.

fastai. Working with gpu. `https://docs.fast.ai /dev/gpu`, 2020. Accessed: 2020-10-09.

Google. Cloud tpu. `https://cloud.google.com/t pu`, 2020a. Accessed: 2020-09-17.

Google. Pytorch/xla. `https://github.com/pytor ch/xla`, 2020b. Accessed: 2020-09-17.

Google. System architecture. `https://cloud.goog le.com/tpu/docs/system-architecture`, 2020c. Accessed: 2020-09-17.

Google. Troubleshooting. `https://cloud.goog le.com/tpu/docs/troubleshooting`, 2020d. Accessed: 2020-09-17.

Google. Xla: Optimizing compiler for machine learning. `https://www.tensorflow.org/xla`, 2020e. Accessed: 2020-09-17.

Google. Using cloud tpu tools. `https://cloud.goog le.com/tpu/docs/cloud-tpu-tools`, 2020f. Accessed: 2020-09-17.

Gray, A., Gottbrath, C., Olson, R., and Prasanna, S. Deploying deep neural networks with nvidia tensorrt. https://developer.nvidia.com/blog/deploying-deep-learning-nvidia-tensorrt/, 2017. Accessed: 2020-09-17.

Gross, S., Chintala, S., and Jones, A. Couple hundred mb are taken just by initializing cuda #20532. https://github.com/pytorch/pytorch/issues/20532, 2019. Accessed: 2020-09-17.

Harris, M. Gpu pro tip: Cuda 7 streams simplify concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, 2015. Accessed: 2020-09-13.

He, H. The state of machine learning frameworks in 2019. *The Gradient*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

Huang, G., Liu, Z., and Weinberger, K. Q. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

Huang, M., Tekur, C., and Carilli, M. Introducing native pytorch automatic mixed precision for faster training on nvidia gpus. https://pytorch.org/blog/accelerating-training-on-nvidia-gpus-with-pytorch-automatic-mixed-precision/, 2020. Accessed: 2020-09-17.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

Jayarajan, A., Wei, J., Gibson, G., Fedorova, A., and Pekhimenko, G. Priority-based parameter propagation for distributed DNN training. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems 2019 (MLSys)*, 2019.

Jia, Z., Thomas, J. J., Warszawski, T., Gao, M., Zaharia, M., and Aiken, A. Optimizing DNN computation with relaxed graph substitutions. In *Proceedings of Machine Learning and Systems 2019 (MLSys)*, 2019.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. In *5th International Conference on Learning Representations, (ICLR)*, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, (ICLR)*, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 24 (NIPS)*, pp. 1097–1105. 2012.

Kukanur, M. Nvidia data center gpu manager simplifies cluster administration. https://developer.nvidia.com/blog/nvidia-data-center-gpu-manager-cluster-administration/, 2016. Accessed: 2020-09-17.

Kumar, N. Google breaks ai performance records in mlperf with world's fastest training supercomputer. https://cloud.google.com/blog/products/ai-machine-learning/google-breaks-ai-performance-records-in-mlperf-with-worlds-fastest-training-supercomputer, 2020. Accessed: 2020-09-17.

Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. *Soviet physics. Doklady*, 10:707–710, 1966.

Li, L. Why does no one use advanced hyperparameter tuning? `https://determined.ai/blog/why-does-no-one-use-advanced-hp-tuning/`, 2020. Accessed: 2020-10-09.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13:3005–3018, 2020.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *6th International Conference on Learning Representations, (ICLR)*, 2018.

Lin, Y., Dhar, S., Li, W., Ren, H., Khailany, B., and Pan, D. Z. Dreampiace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement. In *56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2019.

Liu, R., Krishnan, S., Elmore, A. J., and Franklin, M. Understanding and optimizing packed neural network training for hyper-parameter tuning. *CoRR*, abs/2002.02885, 2020.

Lustig, D. and Martonosi, M. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, 2013.

Markidis, S., Chien, S. W. D., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531, 2018.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Kang, D., Kanter, D., Kumar, N., Liao, J., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Janapa Reddi, V., Robie, T., St John, T., Wu, C.-J., Xu, L., Young, C., and Zaharia, M. Mlperf training benchmark. In *Proceedings of Machine Learning and Systems 2020 (MLSys)*, volume 2, pp. 336–349. 2020.

Narayanan, D., Santhanam, K., Phanishayee, A., and Zaharia, M. Accelerating deep learning workloads through efficient multi-model execution. In *NeurIPS Workshop on Systems for Machine Learning*, December 2018a.

Narayanan, D., Santhanam, K., and Zaharia, M. Accelerating model search with model batching (extended abstract). In *SysML Conference 2018*, 2018b.

Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

NVIDIA. nvidia-smi documentation. `http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf`, 2016. Accessed: 2020-10-09.

NVIDIA. Nvidia tesla v100 gpu architecture. Technical Report WP-08608-001_v1.1, 2017. Accessed: 2020-09-17.

NVIDIA. Nvidia turing architecture whitepaper. Technical Report WP-09183-001_v01, 2018. Accessed: 2020-09-17.

NVIDIA. Nvidia a100 tensor core gpu. `https://www.nvidia.com/en-us/data-center/a100/`, 2020a. Accessed: 2020-09-17.

NVIDIA. Nvidia a100 tensor core gpu architecture. Technical report, 2020b. Accessed: 2020-09-17.

NVIDIA. cudnn developer guide. `https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#grouped-convolutions`, 2020c. Accessed: 2020-09-17.

NVIDIA. Dcgm library api reference manual. `https://docs.nvidia.com/datacenter/dcgm/latest/dcgm-api/group__dcgmFieldIdentifiers.html#group__dcgmFieldIdentifiers`, 2020d. Accessed: 2020-09-17.

NVIDIA. Nvidia dlss 2.0: A big leap in ai rendering. `www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/`, 2020e. Accessed: 2020-09-13.

NVIDIA. Convolutional layers user guide. `https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html`, 2020f. Accessed: 2020-09-17.

NVIDIA. Nvidia multi-instance gpu. `https://www.nvidia.com/en-us/technologies/multi-instance-gpu/`, 2020g. Accessed: 2020-09-13.

NVIDIA. Multi-process service. `https://docs.nvidia.com/deploy/mps/`, 2020h. Accessed: 2020-09-13.

NVIDIA. Nvidia quadro rtx 6000. https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000/, 2020i. Accessed: 2020-09-17.

NVIDIA. Matrix multiplication background user guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2020j. Accessed: 2020-09-17.

NVIDIA. Nvidia v100 tensor core gpu. https://www.nvidia.com/en-us/data-center/v100/, 2020k. Accessed: 2020-09-17.

Odena, A. Open questions about generative adversarial networks. *Distill*, 2019.

OpenAI. Openai five. https://blog.openai.com/openai-five/, 2018. Accessed: 2020-09-13.

Otterness, N. and Anderson, J. H. AMD gpus as an alternative to NVIDIA for supporting real-time workloads. In Völp, M. (ed.), *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, volume 165 of *LIPIcs*, pp. 10:1–10:23, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32 (NIPS)*, pp. 8024–8035. 2019.

PyTorch. Pytorch examples. https://github.com/pytorch/examples, 2020. Accessed: 2020-09-17.

Qi, C. R. Pointnet: Deep learning on point sets for 3d classification and segmentation. https://github.com/charlesq34/pointnet, 2017. Accessed: 2020-09-17.

Qi, C. R., Su, H., Mo, K., and Guibas, L. Pointnet: Deep learning on point sets for 3d classification and segmentation. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 77–85, 2017.

Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. In Bengio, Y. and LeCun, Y. (eds.), *4th International Conference on Learning Representations, (ICLR)*, 2016.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. *CoRR*, abs/1910.02054, October 2019.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

Schaller, R. R. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997. doi: 10.1109/6.591665.

Senior, A., Heigold, G., Ranzato, M., and Yang, K. An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6724–6728, 2013.

Shallue, C. J., Lee, J., Antognini, J. M., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *J. Mach. Learn. Res.*, 20:112:1–112:49, 2019.

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-k., and Woo, W.-c. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 802–810, Cambridge, MA, USA, 2015.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL)*, pp. 3645–3650, 2019.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the importance of initialization and momentum in deep learning. In Dasgupta, S. and McAllester, D. (eds.), *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pp. 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

Vector Institute. Vector institute for artificial intelligence. https://vectorinstitute.ai/, 2021. Accessed: 2021-02-03.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., De-Vito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

Weights&Biases. Sweeps. https://docs.wandb.com/sweeps, 2020. Accessed: 2020-09-17.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

Xia, F. Pointnet.pytorch. https://github.com/fxia22/pointnet.pytorch, 2019. Accessed: 2020-09-17.

Xie, S., Girshick, R. B., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5987–5995, 2017.

Yi, L., Kim, V. G., Ceylan, D., Shen, I.-C., Yan, M., Su, H., Lu, C., Huang, Q., Sheffer, A., and Guibas, L. A scalable active framework for region annotation in 3d shape collections. *SIGGRAPH Asia*, 2016.

Yu, F., Zhang, Y., Song, S., Seff, A., and Xiao, J. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *CoRR*, abs/1506.03365, 2015.

Zeiler, M. D. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

Zheng, B., Vijaykumar, N., and Pekhimenko, G. Echo: Compiler-based GPU memory footprint reduction for LSTM RNN training. In *47th ACM/IEEE Annual International Symposium on Computer Architecture(ISCA)*, pp. 1089–1102. IEEE, 2020. doi: 10.1109/ISCA45697.2020.00092.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Jayarajan, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 88–100, 2018.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately estimating the efficacy of optimizations for dnn training. In *USENIX Annual Technical Conference*, 2020.

SUMMARY OF APPENDICES

These appendices cover the following content.

Appendix A describes the methodology that we use to collect "real-world" GPU cluster usage statistics from the Vector Institute. It also provides the empirical evidence to support our observation that the dominating single-GPU training jobs often have low hardware utilization.

Appendix B lists the operators that HFTA currently supports as well as their corresponding horizontally-fused counterparts.

Appendix C shows how we collect the GPU hardware performance counters and provides the related references.

Appendix D provides additional statistics and insights that can help to clarify our observations and conclusions in Section 5, which does not fit into the main text of the paper due to space constraints.

A "REAL-WORLD" GPU CLUSTER USAGE STATISTICS

We analyzed the job submissions and execution logs for a two-month period (July 1st to Sept. 1st, 2020) from a large GPU cluster belonging to the Vector Institute, an independent, not-for-profit corporation dedicated to research in the field of artificial intelligence and machine learning (Vector Institute, 2021). The cluster services a variety of deep learning training workloads from the Vector Institute's community. The community consists of 501 faculty, postdoc and student researchers who published 263 conference and journal papers from April 2019 to March 2020, including 61 papers in NeurIPS, ICLR, CVPR and ICML.

The cluster includes 4 GPU partitions, V1a (200 P100 GPUs), V1b (40 T4 GPUs), V2 (480 T4 GPUs) and V3 (240 RTX6000 GPUs), where V3 came online in the last few days of the collection period. V2 was recorded for the entire period and the other three partitions were recorded for the last 11 days. V2 is distinguished as the largest partition with the least powerful GPUs. The data contains information on 51338 jobs. The total number of GPU hours spent in these two months amounts to 471768 (equivalent to ~317 GPU days per day).

We classify the submitted jobs as "repetitive single-GPU training jobs" if they contain the following submission and execution patterns:

1. Each job only requests a single GPU despite the availability of multiple GPUs on the same node (i.e., not single-node distributed training). The job also does not require specifically which node the GPU resides (i.e., not multi-node distributed training). Therefore, it can
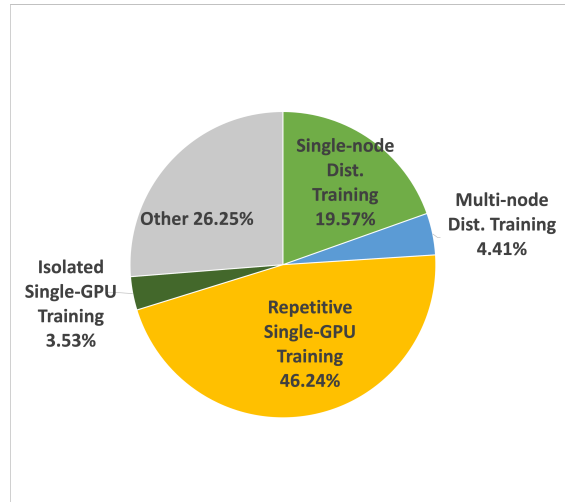

Figure 9. GPU hour usage breakdown for two consecutive months of a large GPU cluster from the Vector Institute.

only be a single-GPU training job.

2. Within a short time period (60 seconds), a batch of such single-GPU jobs are submitted from the same user, which means that the submission of these jobs is automated, and possibly contains the same code/program with varying parameters.[8]

3. The job names are very similar within the batch for such a short time period. We determined the similarity by calculating the normalized Levenshtein distance (Levenshtein, 1966) among job names with a threshold of 0.9. As a reference, the distance score between two job names ranges from 0 to 1, where 1 represents being completely identical and 0 represents being totally different. This filter further verifies that these jobs are repetitive single-GPU jobs since the job names are very similar. Afterwards, a manual inspection of the job names within the batches indicates that those names usually contain small variations such as learning rate value or optimizer choices and settings.

We further reached out to individual users to confirm our conclusion. We interviewed 11 active (i.e., most frequent) users of the GPU cluster: (1) 7 users responded that more than 50% of their jobs are repetitive single-GPU training for purposes including hyper-parameter tuning; and (2) 4 of those 7 users submitted over 95% of their jobs for repetitive single-GPU training. The GPU hour usage distribution is plotted in Figure 9.

Since the cluster does not actively monitor GPU hardware performance counters, we randomly sampled several jobs

---

[8]The exact code for each job was not available for us due to security/IP concerns.

that are tagged as repetitive single-GPU training jobs and gathered the performance counters manually. Based on the `sm_active` and `sm_occupancy` (explained in Section 4 and elaborated in Appendix C) metrics from our samples, we observe that many of the repetitive single-GPU training jobs can severely under-utilize the GPUs both temporally and spatially (as we show in Figure 10a and Figure 10b respectively). The maximum `sm_active` among the sampled jobs is 24%, and maximum `sm_occupancy` among them is 14%.

B   HFTA OPERATOR FUSION RULES

HFTA currently supports 12 PyTorch operators that are commonly used in DL research and development and sufficient to implement the representative set of state-of-the-art DL models (based on the support of these operators, we expect HFTA can already support many more including SqueezeNet (Iandola et al., 2016), VGG (Simonyan & Zisserman, 2015), ConvLSTM (Shi et al., 2015), DenseNet (Huang et al., 2016) and Inception (Szegedy et al., 2016) as well). We list the horizontal operator fusion rules in Table 6. The left column contains the original operators, and the right column indicates using which operator we could get the mathematically equivalent horizontally-fused version of *B* original operators.

C   DCGM METRICS

The `sm_active`, `sm_occupancy` and `tensor_active` performance counters are measured through DCGM (Kukanur, 2016). Their field identifier macros and IDs are listed in Table 7. Please refer to the DCGM Library API Reference Manual (NVIDIA, 2020d) for their precise definitions.

D   ADDITIONAL EVALUATION STATISTICS

In order to facilitate the reading of the results from our GPU experiments in Figure 4, we summarize the comparison from different angles between HFTA and the baselines into three tables.

Table 8 shows the peak training throughput comparison between HFTA and the baselines. It is important to highlight that, for both *MPS* and *concurrent*, the training throughput could decrease as we increase the number of models sharing the same GPU (due to host resource contention). Therefore, the "peak" is determined by the highest possible throughput instead of the largest number of models that the GPU can fit (which might or might not lead to the highest throughput). Unlike Table 5, the results here are split between FP32 and AMP to demonstrate how well HFTA performs for each type of training.
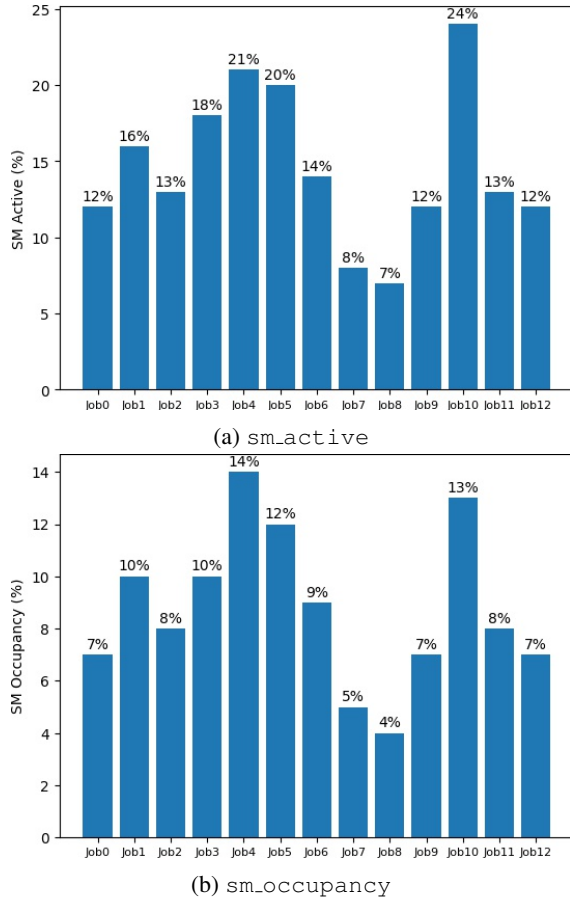

(a) sm_active

(b) sm_occupancy

Figure 10. GPU hardware performance counters measured via DCGM (Kukanur, 2016) for 13 jobs sampled from the clump of repetitive single-GPU training jobs.

Table 6. The horizontal fusion rules for the operators that HFTA currently supports. "ConvT" stands for "ConvTranspose" (a.k.a., deconvolution). $\vec{x}, \vec{y}, \vec{w}$ and $\vec{b}$ represents the input, output, weight and bias tensors respectively. N, C, H, W and L represents the batch sizes, channel sizes, heights, widths and signal lengths of the tensors respectively used in convolutions, deconvolution, batch-norms, MaxPool2d and Dropout2d. G represents the numbers of groups used in the convolutions and deconvolution. F represents the feature map sizes of the tensors used in linear layers. $*$ represents zero or more arguments whose values are kept the same. B represents the number of operators horizontally fused together via HFTA.

PyTorch Operator(Tensors: Shapes, Other Parameters = Arguments)	HFTA Horizontally Fused Operator(Tensors: Shapes, Other Parameters = Arguments)
$\text{Conv2d}(\vec{x} : [N, C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{C_{\vec{x}}}{G}, H_{\vec{w}}, W_{\vec{w}}], \vec{b} : [C_{\vec{y}}], G = g, *)$	$\text{Conv2d}(\vec{x} : [N, B \times C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{B \times C_{\vec{x}}}{G}, H_{\vec{w}}, W_{\vec{w}}], \vec{b} : [B \times C_{\vec{y}}], G = B \times g, *)$
$\text{Conv1d}(\vec{x} : [N, C_{\vec{x}}, L_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{C_{\vec{x}}}{G}, L_{\vec{x}}], \vec{b} : [C_{\vec{y}}], G = g, *)$	$\text{Conv1d}(\vec{x} : [N, B \times C_{\vec{x}}, L_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{B \times C_{\vec{x}}}{G}, L_{\vec{w}}], \vec{b} : [B \times C_{\vec{y}}], G = B \times g, *)$
$\text{ConvT2d}(\vec{x} : [N, C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{C_{\vec{x}}}{G}, H_{\vec{w}}, W_{\vec{w}}], \vec{b} : [C_{\vec{y}}], G = g, *)$	$\text{ConvT2d}(\vec{x} : [N, B \times C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [C_{\vec{y}}, \frac{B \times C_{\vec{x}}}{G}, H_{\vec{w}}, W_{\vec{w}}], \vec{b} : [B \times C_{\vec{y}}], G = B \times g, *)$
$\text{Linear}(\vec{x} : [N, F_{\vec{x}}], \vec{w} : [F_{\vec{x}}, F_{\vec{y}}], \vec{b} : [F_{\vec{y}}])$	$\text{baddbmm}(\vec{b} : [B, 1, F_{\vec{y}}], \vec{x} : [B, N, F_{\vec{x}}], \vec{w} : [B, F_{\vec{x}}, F_{\vec{y}}])$
$\text{BatchNorm1d}(\vec{x} : [N, C_{\vec{x}}] \text{ or } [N, C_{\vec{x}}, L_{\vec{x}}], \vec{w} : [C_{\vec{x}}], \vec{b} : [C_{\vec{x}}], *)$	$\text{BatchNorm1d}(\vec{x} : [B \times N, C_{\vec{x}}] \text{ or } [N, B \times C_{\vec{x}}, L_{\vec{x}}], \vec{w} : [B \times C_{\vec{x}}], \vec{b} : [B \times C_{\vec{x}}], *)$
$\text{BatchNorm2d}(\vec{x} : [N, C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [C_{\vec{x}}], \vec{b} : [C_{\vec{x}}], *)$	$\text{BatchNorm2d}(\vec{x} : [N, B \times C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], \vec{w} : [B \times C_{\vec{x}}], \vec{b} : [B \times C_{\vec{x}}], *)$
$\text{MaxPool2d}(\vec{x} : [N, C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], *)$	$\text{MaxPool2d}(\vec{x} : [N, B \times C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], *)$
$\text{Dropout2d}(\vec{x} : [N, C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], *)$	$\text{Dropout2d}(\vec{x} : [N, B \times C_{\vec{x}}, H_{\vec{x}}, W_{\vec{x}}], *)$
$\text{Dropout}(\vec{x} : [*])$	$\text{Dropout}(\vec{x} : [*, B, *], *)$
$\text{LeakyReLU}(x : [*], *)$	$\text{LeakyReLU}(x : [*, B, *], *)$
$\text{ReLU}(x : [*], *)$	$\text{ReLU}(x : [*, B, *], *)$
$\text{Tanh}(x : [*])$	$\text{Tanh}(x : [*, B, *])$

Table 7.

Name	Field Identifier Macro	ID
sm_active	DCGM_FI_PROF_SM_ACTIVE	1002
sm_occupancy	DCGM_FI_PROF_SM_OCCUPANCY	1003
tensor_active	DCGM_FI_PROF_PIPE_TENSOR_ACTIVE	1004
GPU Utilization	DCGM_FI_DEV_GPU_UTIL	203

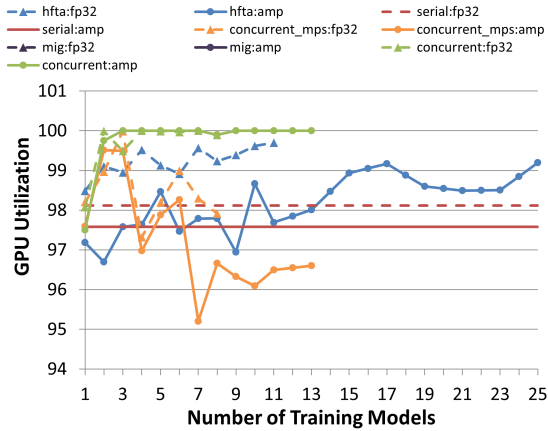*Table 8.* The peak training throughput speedups of HFTA over the baselines.

Benchmark			PointNet Cls.	PointNet Seg.	DCGAN
V100	FP32	serial	2.62	1.62	4.18
		concurrent	2.54	1.62	1.95
		MPS	2.36	1.17	1.95
	AMP	serial	5.02	4.29	4.59
		concurrent	5.02	4.24	2.01
		MPS	4.50	3.03	2.03
RTX 6000	FP32	serial	2.46	1.97	6.69
		concurrent	2.46	1.95	1.64
		MPS	2.07	1.22	1.69
	AMP	serial	4.36	3.63	6.29
		concurrent	4.26	3.54	1.72
		MPS	3.79	2.54	1.82
A100	FP32	serial	5.47	4.56	4.46
		concurrent	5.47	4.56	1.39
		MPS	2.05	1.31	1.37
		MIG	2.10	1.35	1.59
	AMP	serial	11.50	9.48	3.61
		concurrent	12.98	10.26	1.06
		MPS	4.72	2.93	1.09
		MIG	4.88	3.02	1.09

Table 9. The maximum training throughput speedups of HFTA over the baselines given the same number of models sharing one GPU.

Benchmark			PointNet Cls.	PointNet Seg.	DCGAN
V100	FP32	concurrent	1.77	1.62	1.91
		MPS	1.65	1.17	1.95
	AMP	concurrent	3.41	3.12	2.27
		MPS	3.05	2.23	2.23
RTX 6000	FP32	concurrent	2.32	1.95	1.96
		MPS	1.92	1.22	1.78
	AMP	concurrent	4.14	3.21	1.73
		MPS	3.75	2.35	1.90
A100	FP32	concurrent	4.91	3.97	8.94
		MPS	1.64	1.04	9.41
		MIG	1.51	1.07	1.51
	AMP	concurrent	9.16	7.86	9.07
		MPS	3.18	2.13	7.48
		MIG	2.07	1.58	1.20

Table 10. The maximum speedups of AMP training over FP32.

Benchmark		PointNet Classification	PointNet Segmentation	DCGAN
V100	*serial*	1.00	1.00	1.00
	concurrent	0.97	1.01	1.07
	MPS	1.01	1.03	1.06
	HFTA	1.92	2.65	1.10
RTX6000	*serial*	1.06	1.19	1.16
	concurrent	1.09	1.22	1.05
	MPS	1.03	1.05	1.02
	HFTA	1.88	2.20	1.09
A100	*serial*	1.13	1.13	1.01
	concurrent	1.00	1.05	1.08
	MPS	1.03	1.06	1.03
	MIG	1.02	1.05	1.20
	HFTA	2.37	2.36	0.82

Figure 11. `nvidia-smi`-defined "GPU utilization" for PointNet classification task on A100.

hardware under-utilization of repetitive single-accelerator training workloads.

Table 9 shows the maximum training throughput speedups of HFTA over the baselines, given the same number of models sharing the same GPU. The maximum is picked by varying the number of models sharing the same GPU and finding the largest performance gap between HFTA and the baselines. This helps to isolate the benefits of better SMs and TCs utilization from the benefits of better memory utilization when training via HFTA.

Table 10 shows the maximum training throughput speedups of AMP over FP32 for both HFTA and the baselines. The maximum here is also picked by varying the number of models (except for *serial* which always only run one model per GPU) and finding the largest performance gap between FP32 and AMP. This helps to demonstrate that HFTA is more efficient in utilizing advanced hardware compute units such as TCs.

Similar to Figure 8a, 8b and 8c, Figure 11 plots the `nvidia-smi`-defined "GPU utilization" (NVIDIA, 2016) for PointNet classification task training on the A100 GPU. Contrary to a popular belief (Elangovan, 2020; fastai, 2020), we observe that the `nvidia-smi`-defined "GPU utilization" can be sometimes a weak utilization indicator, since the curves in Figure 11 appear rather noisy and do not follow the trends of throughput improvements in Figures 4g or any hardware counters' trend in Figure 8a, 8b or 8c.

Similar to Figure 8, Figure 12 plots the `sm_active`, `sm_occupancy`, and `tensor_active` of HFTA and the baselines as we increase the number of models sharing the same V100 GPU. In addition to the observations we already present in Section 5.3, we also observe that the hardware utilization of the *serial* baselines is lower on A100 than on V100. Therefore, Figure 12 provides empirical evidence to support our argument in Section 2.1 and Section 5.1 that newer GPU generations suffer more significantly from the
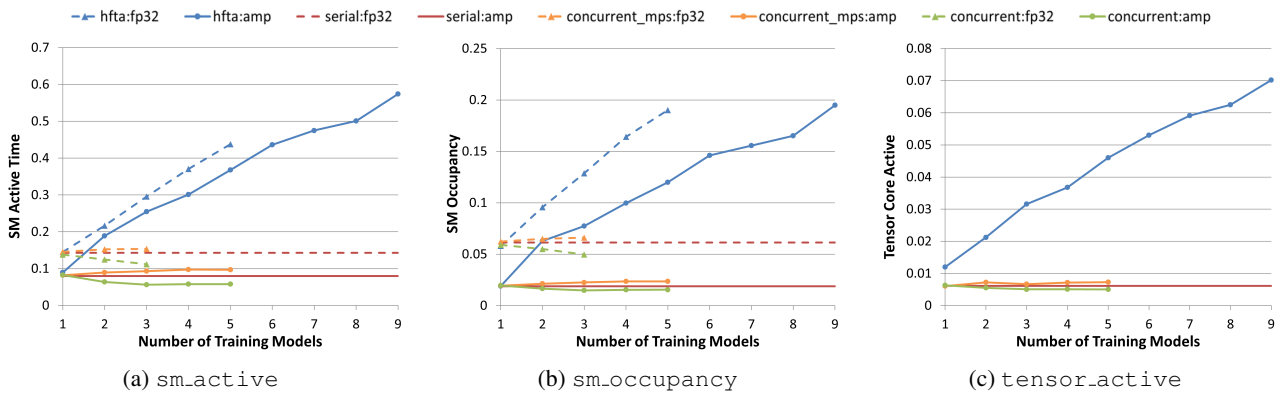
(a) sm_active (b) sm_occupancy (c) tensor_active

Figure 12. The hardware performance counters for PointNet classification task as we increase the number of models sharing the same V100.