VIRTUALFLOW: DECOUPLING DEEP LEARNING MODELS FROM THE UNDERLYING HARDWARE

Andrew Or [1 *] **Haoyu Zhang** [2] **Michael J. Freedman** [3]

ABSTRACT

We propose VirtualFlow, a system leveraging a novel abstraction called *virtual node processing* to decouple the model from the hardware. In each step of training or inference, the batch of input data is split across virtual nodes instead of hardware accelerators (e.g., GPUs and TPUs). Mapping multiple virtual nodes to each accelerator and processing them sequentially effectively time slices the batch, thereby allowing users to reduce the memory requirements of their workloads and mimic large batch sizes on small clusters. Using this technique, VirtualFlow enables many new use cases, such as reproducing training results across different hardware, resource elasticity, and heterogeneous training. In our evaluation, our implementation of VirtualFlow for TensorFlow achieved strong convergence guarantees across different hardware with out-of-the-box hyperparameters, up to 48% lower job completion times with resource elasticity, and up to 42% higher throughput with heterogeneous training.

1 INTRODUCTION

Modern deep learning frameworks, such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019), tightly couple the model and the underlying hardware. This forces them to make a number of simplifying assumptions about the environment in which deep learning jobs are run. First, a model's convergence behavior is not preserved across different hardware configurations. Instead, the burden falls on the user to retune the hyperparameters and apply custom optimization techniques in order to achieve the same training results (Goyal et al., 2017; Jia et al., 2018; You et al., 2019). Second, resource allocations are tied to the lifetime of a job; any adjustment to a job's allocation requires interrupting the job and restarting it from checkpoints. Third, the set of resources allocated to a job must be homogeneous.

In light of recent trends, however, the above assumptions fall short in today's real world circumstances. First, the scale of deep learning workloads continues to rise dramatically: model sizes have grown to billions of parameters (Shoeybi et al., 2019; Rosset, 2020; Brown et al., 2020), dataset sizes to hundreds of GBs (Deng et al., 2009; Raffel et al., 2019), and batch sizes to 64K and above (Jia et al., 2018; Sun et al., 2019). Hardware advances have been slow to catch up, however, leading to high computational requirements for these larger workloads. For instance, BERT has been pre-trained on 16 TPUs (Devlin et al., 2018) and up to 1024 TPUs (You et al., 2019), and Megatron-LM, with 8.3 billion parameters, has been trained on 512 V100 (32GB) GPUs (Shoeybi et al., 2019). But given the inability to preserve convergence behavior across hardware, these results are often not repeatable for researchers who cannot afford these resources.

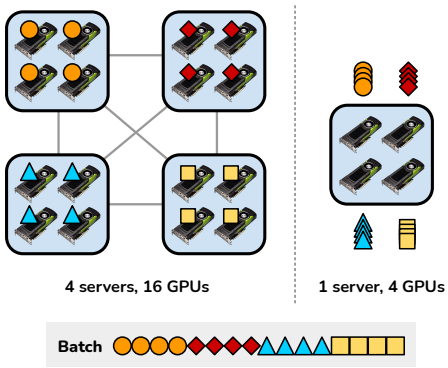


Figure 1. **Virtual node processing.** A batch is split into 16 equally sized virtual nodes (colored shapes), which are distributed among the GPUs. Virtual nodes assigned to the same GPUs are executed sequentially, allowing 4 GPUs to train the same model as 16 GPUs using the same set of hyperparameters, including the batch size.

Second, shared clusters with a heterogeneous mix of GPUs are increasingly common. For instance, Microsoft reports using a large, multi-tenant cluster consisting of thousands of GPUs of various types shared among hundreds of users (Jeon et al., 2019). Resource heterogeneity is also common in small research lab settings, which often accumu-

[*]Work performed while at Princeton University. [1]Meta Platforms Inc., Menlo Park, CA, USA [2]Google AI, Mountain View, CA, USA [3]Department of Computer Science, Princeton University, Princeton, NJ, USA. Correspondence to: Andrew Or <andrewor14@gmail.com>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022.

late multiple generations of GPUs over the years (Narayanan et al., 2020). Yet systems today are unable to leverage this heterogeneity for individual training jobs.

1.1 New Challenges

These recent trends raise new important challenges or exacerbate existing ones for today's deep learning workloads:

High resource requirement. Many new workloads require large clusters of expensive hardware accelerators that are inaccessible to most users.

Lack of experimentation. With the increase in scale, users may wish to experiment on a small testbed before deploying the model on a large cluster. However, this is not possible today: the original batch size will not fit within the memory limits of the testbed, and changing the batch size may compromise the convergence of the model.

Lack of reproducibility. More generally, reproducing the same model convergence behavior across different hardware typically requires readjusting important hyperparameters such as the batch size and the learning rate (Goyal et al., 2017; Sun et al., 2019; Jia et al., 2018). This is cumbersome in practice, and techniques proposed for one workload often do not work for another (Shallue et al., 2018).

Adapting to dynamic resource availability. Existing attempts to dynamically adjust a job's resource allocation must interrupt and restart the job (Xiao et al., 2018; Peng et al., 2018; Narayanan et al., 2020), since resource allocations are static in today's frameworks. However, this is inefficient, because each adjustment can take minutes (Or et al., 2020). Further, the batch size may change across restarts, potentially affecting the convergence of the model.

Adapting to heterogeneous environments. Today, jobs are restricted to single types of accelerators. Being able to additionally utilize leftover accelerators of different types can lead to faster jobs and higher cluster utilization.

1.2 Decoupling Model from Hardware

All of the above challenges largely stem from a central drawback in today's deep learning systems: *a tight coupling between the model and the underlying hardware*. In this paper, we argue that systems-level constraints should be decoupled from application-level semantics. A model should converge to the same accuracy regardless of the set of resources it is trained on. Performance should degrade gracefully with the amount of resources assigned to a job. The user should be able to tune the model's hyperparameters once and train the model everywhere, and the result should be the same across different hardware configurations.

The same philosophy can be observed in many big-data analytics systems. In MapReduce-style batch processing (Dean & Ghemawat, 2004; Zaharia et al., 2012) and stream processing workloads (Apache Storm; Apache Flink; Zaharia et al., 2013), the system always computes the same answers regardless of the level of parallelism and the amount of resources assigned to the job. The input data is sliced into many small partitions to be processed in multiple sequential waves of tasks, and the job would not fail if the amount of data processed in a single wave did not fit in the aggregate memory of the system.

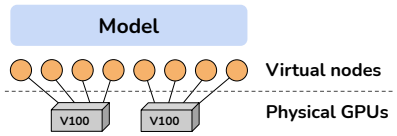


Figure 2. Mapping between virtual nodes and accelerators is flexible, but only virtual nodes affect model convergence. Thus, changes to resource allocations are hidden from the application.

1.3 Virtual Node Processing

Towards this goal of separating the model from the hardware, this paper introduces *virtual nodes* as a substrate for distributing computation across hardware accelerators (Figure 1). In this paradigm, each batch of the input data is partitioned among virtual nodes instead of hardware accelerators. One or more virtual nodes are then mapped to each hardware accelerator and processed sequentially on the accelerator, thus producing one or more MapReduce-style waves of execution within each step of training or inference.

VirtualFlow's approach leverages the insight that all virtual nodes share the same model parameters. This allows the model to be cached in each accelerator's memory at the beginning of each step and efficiently reused by all virtual nodes mapped to that accelerator. The gradients produced by these virtual nodes are then aggregated into a shared memory buffer on the accelerator, thus adding a small, constant overhead independent of the number of virtual nodes (§3.1).

Virtual node processing allows VirtualFlow to preserve model convergence behavior across different hardware by fixing the total number of virtual nodes, and thus the batch size and other hyperparameters. Instead, only the mapping between virtual nodes and hardware accelerators need to be adjusted (Figure 2). This enables new important use cases:

Lower resource requirement. Workloads that previously required large clusters can now be packed into smaller ones by mapping many virtual nodes to each accelerator.

Reproducibility and experimentation on smaller test beds is now possible, as results obtained by other users can now be reproduced on a different set of resources without modification of any hyperparameter or optimization strategy.

Resource elasticity. Dynamically resizing a job while maintaining convergence guarantees—previously an open

challenge (Or et al., 2020)—is now possible. VirtualFlow achieves this by redistributing the virtual nodes among the new set of accelerators. When scaling out, important virtual node state such as model parameters and certain stateful kernels (e.g., batch normalization variables (Ioffe & Szegedy, 2015)) are migrated in an all-gather operation to bootstrap the new workers (§4.1). Unlike in state-of-the-art schedulers, the transition is seamless from the application's perspective and the job need not be restarted.

Heterogeneous training—combining multiple types of accelerators in the same job—can be expressed as distributing virtual nodes unevenly across the accelerators, thereby assigning more data to the more powerful types. Given a workload and a set of heterogeneous resources, VirtualFlow solves for the most efficient configuration(s) using offline profiles (§5.1) and ensures training correctness by performing weighted gradient synchronizations (§5.2).

We implemented VirtualFlow on top of TensorFlow and evaluated the system on a set of representative models (ResNet (He et al., 2016), BERT (Devlin et al., 2018), Transformer (Vaswani et al., 2017)). To showcase the benefits of heterogeneous training in a multi-tenant setting, we also extended Gavel (Narayanan et al., 2020) to consider heterogeneous allocations (§6.4.1). In our evaluation, VirtualFlow demonstrates strong model convergence guarantees across different hardware, improves cluster utilization by 20% and reduces job completion time by 48% with elasticity, and improves job throughput by 42% with heterogeneous training.

2 BACKGROUND

In this section, we describe two important ways deep learning workloads are tightly coupled with the underlying hardware in state-of-the-art systems (§2.1, §2.2), then discuss the target setting of this paper (§2.3).

2.1 Hyperparameters Tied to Hardware

Hyperparameters, such as the batch size, learning rate, and dropout rate, have important effects on the convergence of a model. The *batch size* refers to the number of input examples, e.g., images or sentences, processed within a training or inference step. Each batch is divided evenly among the accelerators, which are assumed to be homogeneous.

Using larger batch sizes generally improves training and inference throughput. Within a single accelerator, the *local* batch size is often set to the maximum size possible within the limits of the accelerator's memory capacity. Across multiple accelerators, the *global* batch size is simply the sum of all local batch sizes across the accelerators. Thus, a larger global batch size leads to higher levels of parallelism.

However, prior work has shown that large batch sizes tend to deteriorate model convergence (Keskar et al., 2016). In order to preserve convergence behavior while scaling a workload, various efforts have proposed to adjust hyperparameters dependent on the batch size, such as the learning rate (Goyal et al., 2017), or even to apply custom optimization algorithms (Sun et al., 2019; You et al., 2017; 2019).

Hurdles for reproducibility. Thus, reproducing existing results on a different set of hardware requires significant effort and expertise. In some cases, it is even impossible. For example, the results from training the the BERT model using a batch size of 32K examples on 1024 TPUs (You et al., 2019) and 1472 GPUs (Narasimhan, 2019) cannot be reproduced on a smaller test bed of 16 GPUs, as the same batch size would not fit in the smaller cluster's GPU memory. On the other hand, reducing the batch size would inevitably lead to very different convergence trajectories that require retuning various hyperparameters. This poses a major hurdle for experimentation as well as scaling.

2.2 Inflexible Model Graph

Another source of coupling between the model and the hardware lies in the *model graph*, which specifies the network of operations to perform on the input data. Today's frameworks compile and optimize this graph once at the beginning of training and reuse it for the rest of the job. In addition to tensor operations, information regarding the underlying cluster configuration is also embedded in the model graph. In both TensorFlow and PyTorch, for instance, the graph is defined under a *distribution strategy* that specifies how model parameters should be synchronized.

Hurdles for resource elasticity. Once the model graph is created under a particular distribution strategy, subsequent training will use synchronization operations that involve a fixed set of accelerators. Adjusting a job's resource allocation would involve rebuilding the entire graph under a new distribution strategy and reloading previously trained model parameters from a checkpoint, an expensive process that can take minutes (Or et al., 2020). Further, as discussed in §2.1, changing the amount of resources in the middle of a job can lead to adverse effects on the model's convergence.

2.3 Data Parallel, Synchronous Training

The most common form of parallelism in distributed deep learning workloads is *data parallelism*, where the each accelerator processes its share of the input batch independently. This is in contrast to *model parallelism*, which partitions, instead of replicates, the model graph across the accelerators, and is used primarily for extremely large models that do not fit in the memory of a single accelerator.

In modern workloads, data parallelism is typically combined with *synchronous training*, which enforces a synchro-

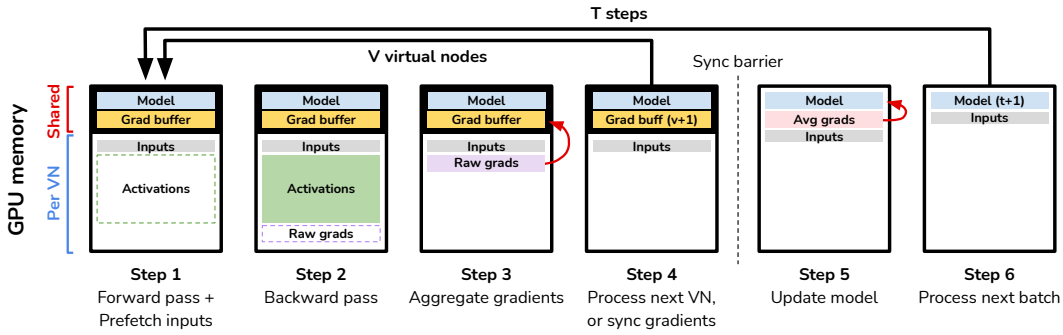


Figure 3. Memory footprint of virtual node processing in a single training step. The model parameters and the gradient buffer are shared across all virtual nodes on each accelerator, while the memory used in each forward and backward pass is specific to individual virtual nodes. Memory overhead from the gradient buffer is a small constant independent of the number of virtual nodes V per accelerator.

nization barrier at the end of each step, and is shown to have better convergence properties than *asynchronous training* (Chen et al., 2016). Gradients can be synchronized using either the parameter server architecture (Li et al., 2014) or the all-reduce architecture (Thakur et al., 2005; Sergeev & Del Balso, 2018), though the latter is increasingly common.

This paper targets data parallel, synchronous training, though many of the techniques proposed are also applicable to the model parallel setting. This is explored further in §7.

3 Virtual Node Processing

The core concept in VirtualFlow is virtual node processing, a layer of indirection between the model and the hardware. From the model's perspective, *virtual nodes*, rather than accelerators, perform the computation. As long as the total number of virtual nodes is unchanged, the batch size and thus the convergence of the model also remains the same.

3.1 Virtual Node Execution

Each batch of the input data is split among the virtual nodes in a manner analogous to how a job in MapReduce is divided into tasks. Virtual nodes assigned to the same hardware accelerator are processed sequentially, while virtual nodes assigned to different accelerators are still processed in parallel. This produces one or more *waves* of execution, similar to MapReduce workloads where the number of tasks is often a small multiple of the number of CPU slots in the system.

Figure 3 traces the steps involved in processing a single batch of data with virtual node processing. In each training step, VirtualFlow computes V forward and backward passes, where V is the number of virtual nodes on each accelerator. During the forward pass, VirtualFlow computes the activations while prefetching inputs for the next virtual node in the background (Step 1). At the end of the backward pass (Step 2), local gradients are aggregated into a gradient buffer shared across all virtual nodes on the accelerator (Step 3). After all virtual nodes have been processed,

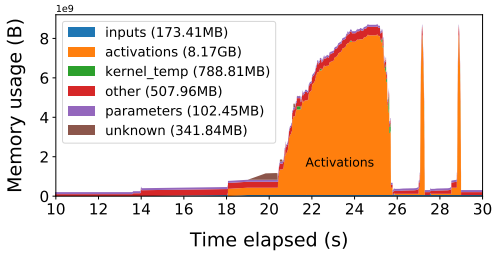


Figure 4. Memory usage in the first 3 steps of training ResNet-50 on ImageNet on a single 2080 Ti GPU, broken down by category. Activations constitute the vast majority of memory usage at the peak. The first step is slower due to initial graph optimizations.

the locally aggregated gradients are synchronized across the cluster (Step 4) and each accelerator applies the averaged gradients to its copy of the model as before (Step 5).

3.2 Memory Overhead

The gradient buffer in VirtualFlow is a source of memory overhead (Figure 3). However, because this buffer is shared among all virtual nodes assigned to the same accelerator, the memory overhead is a *constant* independent of the number of virtual nodes on the accelerator. The size of this buffer is the same as the model, which is a small fraction of the peak memory usage for most workloads. Instead, memory usage is typically dominated by activations computed during the forward pass, which scale with the batch size while the model does not. For example, in Figure 4, the activations required over 8GB, while the model is only around 100MB.

3.3 Time and Resource Trade-off

Today's deep learning systems are a special case of virtual node processing that uses one virtual node per hardware accelerator (Figure 5a). However, this is only one possible configuration in the trade-off space between time and resource requirements. VirtualFlow divides the compution in each batch in the time dimension as well as in the spatial dimension (Figure 5b and c), processing the virtual nodes assigned to the same accelerators sequentially. This provides

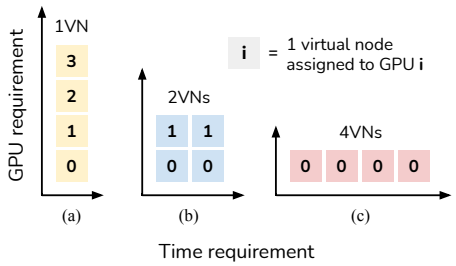


Figure 5. Virtual node trade-off between resource requirement and time requirement. VN in this figure refers to number of virtual nodes assigned to each hardware accelerator. The design space for today's deep learning workloads is limited to only (a).

users with the freedom to gracefully fall back to running on fewer accelerators with longer training times.

This flexibility is crucial to model reproducibility, experimentation, and hyperparameter exploration. By preserving convergence behavior across different hardware configurations (e.g., Figure 5a, b, and c), VirtualFlow allows users to replicate training results produced by others regardless of the resources used. Experimentation on smaller testbeds is now possible by using many virtual nodes on each accelerator to mimic the larger deployment. On the other hand, users can explore the effects of using previously inaccessible batch sizes on the same set of resources by increasing the number of virtual nodes used on each accelerator.

4 Resource Elasticity

Elasticity is widely used in batch processing (Or, 2014), stream processing (Gedik et al., 2013), cluster management (Szczepkowski & Wielgus, 2016), and cloud computing (AWS; Azure; GCE), enabling higher cluster utilization and lower job completion time. In this section, we describe how VirtualFlow can bring the same benefits to distributed deep learning workloads by expressing elasticity in terms of redistributing virtual nodes across accelerators.

4.1 Redistributing Virtual Nodes

VirtualFlow maintains a mapping between virtual nodes and hardware accelerators, but this mapping need not be fixed over time. To enable resource elasticity, virtual nodes can be redistributed dynamically across the accelerators assigned to a job in response to cluster demand. When redistributing virtual nodes, the total number of virtual nodes remains the same, and so adjustments to a job's resource allocation are seamless from the perspective of the application.

When scaling out, certain virtual node state must be migrated to the new accelerators, including the model parameters and certain *stateful kernels*. One example of the latter is the batch normalization moving mean and variance (Ioffe & Szegedy, 2015), which are computed independently on each accelerator and never synchronized. Bootstrapping new workers without also migrating these stateful kernels would effectively reset their internal state, potentially hurting convergence. VirtualFlow migrates these stateful kernels as well as the model parameters through an all-gather operation performed on the new workers. This process typically takes less than a second (similar to all-reduce) and only takes place once per resource adjustment.

Algorithm 1: Elastic WFS Scheduler

```
function schedule (running_jobs, job_queue):
new_allocations = expand current allocations
while job_queue not empty do
    fair_allocations = compute fair shares(
        running_jobs, job_queue.peek())
    if no higher priority job allocations are affected then
        new_allocations = fair_allocations
        running_jobs += job_queue.dequeue()
    else
        break
resize jobs(new_allocations)
```

4.2 Elastic Weighted Fair Sharing (WFS)

To showcase the benefits of expressing elasticity in terms of virtual nodes, we built a simple event-driven scheduler that dynamically resizes deep learning jobs based on their relative weighted fair shares (WFS) (Demers et al., 1989). These fair shares are computed based on the priority of the jobs, which can be set to arbitrary attributes of the job to express a variety of scheduling objectives, such as Shortest Job First (SJF) and Shortest Remaining Time First (SRTF). The main scheduling logic is summarized in Algorithm 1.

This scheduler has two important distinctions from existing GPU cluster schedulers (Xiao et al., 2018; Peng et al., 2018; Narayanan et al., 2020). First, resource adjustments need not interrupt the jobs and restart them from checkpoints. Second, while existing schedulers dynamically adjust cluster-wide resource allocations, they are unable to resize individual jobs without potentially hurting model convergence. However, not being able to resize individual jobs leads to lost opportunities for more efficient scheduling. As an example, suppose job A demands 2 GPUs and job B demands 8 GPUs and there are no other jobs in the queue. If there are 8 GPUs in total, jobs A and B will never be able to run at the same time, and 6 GPUs will have to go idle for the entire duration of job A. Regardless of the scheduling policy, being able to dynamically adjust the resource requirement of each job will generate many new scheduling opportunities, potentially leading to higher cluster utilization (§6.3).

5 Heterogeneous Training

An important assumption made by state-of-the-art frameworks is resource allocations must be *homogeneous*. Virtu-

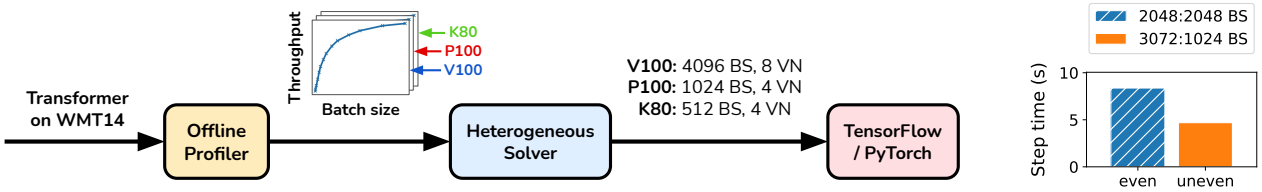


Figure 6. (Left) VirtualFlow heterogeneous training overview. (Right) Splitting a batch evenly across a set of uneven resources is inefficient. In this workload, we wish to train ResNet-50 on ImageNet on 2 V100 GPUs and 2 P100 GPUs with a global batch size of 8192. "3072:1024 BS" means 3072 examples are assigned to each V100 GPU, while 1024 are assigned to each P100 GPU per batch. In this example, the heterogeneous solver will output the more efficient uneven configuration.

alFlow relaxes this assumption by allowing users to combine multiple accelerator types in the same job, potentially leading to significant improvements in job throughput and cluster utilization. There are two main challenges involved, however. First, how to distribute virtual nodes across heterogeneous resources efficiently (§5.1)? Second, how to provide the same semantics as homogeneous training (§5.2)?

5.1 Virtual Node Assignment

The key intuition is to assign more virtual nodes to the resource types with higher compute capabilities, so as to balance the step times across the different accelerator types. This allows users to scale their workloads by reducing the amount of computation required on the each type of accelerator, thereby improving the overall throughput.

For additional flexibility in deciding how to split the batch across the heterogeneous set of resources, we further relax the constraint that the size of each virtual node must be the same across all accelerators. Then, determining an efficient assignment of virtual nodes involves two steps, as outlined in Figure 6 (left). First, VirtualFlow performs an offline profile of the given workload on all target accelerator types. Then, using these offline profiles, VirtualFlow solves for a configuration that minimizes the overall step time.

5.1.1 Offline Profile

To generate an offline profile, VirtualFlow runs the given workload on a single hardware accelerator at a time across all batch sizes of interest that fit in the accelerator's memory. Due to memory alignment, we only consider batch sizes that are powers of 2 or power-of-2-like numbers (e.g., 48, 192, 768), which are the mid-points between adjacent powers of 2. The process is then repeated on all accelerator types and the result is a set of throughput over batch size curves (Figure 6, left), one for each accelerator type.

For each batch size, we only need to run a few steps (e.g., 20) to arrive at a representative average throughput, since the performance is typically consistent across steps. Therefore, the entire process typically takes no longer than 10 minutes, a small fraction of the job duration for many deep learning workloads, which can run for many hours or even days.

5.1.2 Heterogeneous Solver

To understand why a solver is necessary, consider the scenario in Figure 6 (right), in which we are given 2 V100 GPUs and 2 P100 GPUs, all with 16GB of memory capacity. A naive, even split of the batch would assign the same amount of data and the same number of virtual nodes to each GPU, regardless of the GPU type. However, this configuration is inefficient, because, for this workload, V100 GPUs are 4x as fast as P100 GPUs, so the system will be bottlenecked on the P100 GPUs, leaving the V100 GPUs idle for a large fraction of the training time. Instead, an uneven split that assigns more data in each batch to the V100 GPUs will result in a much shorter (44%) overall step time.

In order to arrive at an appropriate split across the different accelerator types, we formulate the problem as follows. For simplicity, we treat the resources as GPUs:

$$\begin{aligned} \text{Objective} \quad & \min \max_i \left(t_i(b_i) \cdot v_i + comm\right) \\ \text{Constraint} \quad & \sum_i n_i \cdot b_i \cdot v_i = B \\ \text{Solve for} \quad & b_i, v_i, n_i \quad \forall i \end{aligned}$$

$$\begin{aligned} B &= \text{Global batch size} \\ b_i &= \text{Per virtual node batch size for GPU type } i \\ v_i &= \text{Number of virtual nodes on each GPU of type } i \\ t_i(b_i) &= \text{Step time on GPU type } i \\ n_i &= \text{Number of GPUs of type } i \\ comm &= \text{Communication overhead} \end{aligned}$$

This objective aims to equalize the step times across all GPU types so as to minimize the overall step time, which is bottlenecked by the slowest GPU. We multiply the step time by the number of virtual nodes v_i to reflect the fact that virtual nodes on a given GPU are executed sequentially. The step times for each GPU type t_i are constants supplied by offline profiles computed previously (§5.1.1). The communication overhead $comm$ can be estimated as part of the offline profile by taking the difference between the distributed and single GPU step times, using a per GPU batch size of 1 and synthetic input examples to isolate the time spent on gradient synchronization.

The solver falls back to recommending homogeneous allocations when there are no heterogeneous combinations that can improve the throughput of the job. This can happen if the compute capabilities are vastly different across the GPU types, and there are not enough of the slower GPUs to compensate for the discrepancy in performance.

5.2 Correctness

An important goal of VirtualFlow is to preserve training semantics regardless of the underlying hardware. However, naively applying existing gradient synchronization and data sharding techniques can lead to incorrect results.

Gradient synchronization. Existing implementations of gradient synchronization first take a local average of the gradients computed on each accelerator, then take a *global* average of these local averages across all accelerators. On heterogeneous resources, however, this method can produce incorrect gradients. For instance, suppose we assign 6 input examples to GPU0 and 2 input examples to GPU1 in each batch. Taking a simple average will result in:

$$\frac{1}{2}\left(\frac{g_1 + ... + g_6}{6}\right) + \frac{1}{2}\left(\frac{g_7 + g_8}{2}\right) = \frac{g_1 + ... + g_6 + 3(g_7 + g_8)}{12}$$

where the gradients on GPU1 are weighed disproportionately compared to the rest. Instead, VirtualFlow performs a *weighted average* during gradient synchronization:

$$\frac{3}{4}\left(\frac{g_1 + ... + g_6}{6}\right) + \frac{1}{4}\left(\frac{g_7 + g_8}{2}\right) = \frac{g_1 + ... + g_8}{8}$$

This ensures all gradients are considered equally regardless of how the data is distributed across the accelerators.

Data sharding. Similarly, existing sharding techniques assume the batch is split evenly across the accelerators. Naively reusing these techniques for heterogeneous training will result in certain input examples being observed more often than others. VirtualFlow maintains the exactly-once data semantics of homogeneous training by sharding the dataset unevenly to match the relative local batch sizes (e.g., 4:1) across the different accelerator types.

6 Evaluation

We implemented VirtualFlow with resource elasticity and heterogeneous training support on top of TensorFlow 2.4 in 2700+ lines of code. For elasticity, we used the same mechanisms as (Or et al., 2020), in which Horovod (Sergeev & Del Balso, 2018) was used as the narrow waist communication layer that connects a changing set of worker processes. In this section, we evaluate VirtualFlow's effectiveness in reproducing results across different hardware (§6.2), providing elasticity while preserving model semantics (§6.3), and enabling heterogeneous training (§6.4).

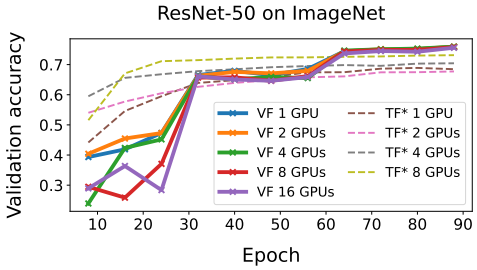


Figure 7. **Reproducibility:** VirtualFlow preserves the convergence trajectory across different numbers of GPUs by fixing the batch size at 8192. Naively attempting to reproduce the same workload on fewer GPUs without retuning the hyperparameters (TF*) yields lower accuracies and different convergence behavior.

6.1 Experimental Setup

End-to-end reproducibility and elasticity experiments are performed on 2 servers, each with 8 NVIDIA V100 GPUs (16GB), 64 Intel Xeon CPUs (2.2Ghz), and 250GB of DRAM, connected over a 16 Gbps connection. Heterogeneous training experiments additionally use 2 extra similar servers, each with 4 NVIDIA P100 GPUs (16GB). Exploration and microbenchmark experiments use 2 NVIDIA GeForce RTX 2080Ti GPUs on a server with 32 Intel(R) Xeon(R) E5-2620v4 CPUs (2.1GHz) and 64GB of DRAM.

6.2 Reproducibility

We demonstrate VirtualFlow's reproducibility using two well-known workloads: ResNet-50 (He et al., 2016) on ImageNet (Deng et al., 2009) and BERT (Devlin et al., 2018) fine-tuning on GLUE (Wang et al., 2019). We varied the number of GPUs while fixing the global batch sizes, and observed almost identical convergence trajectories across different allocations for both workloads.

Baseline. We compare VirtualFlow with a version of vanilla TensorFlow that does not retune hyperparameters across batch sizes (TF*). For example, for ResNet, we do not apply the linear scaling rule (Goyal et al., 2017) to adjust the learning rate when simulating large workloads on smaller sets of GPUs. This setup is motivated by the fact that these optimization techniques are workload-specific and difficult to identify for arbitrary workloads (Shallue et al., 2018).

6.2.1 *ResNet-50 on ImageNet*

In this experiment, we train ResNet-50 on the ImageNet dataset for 90 epochs using a fixed batch size of 8192, a widely used benchmark that is known to converge to the vicinity of 76% (Goyal et al., 2017; He et al., 2016; Gross & Wilber, 2016). To demonstrate VirtualFlow can preserve convergence across GPU types, we ran this workload on both V100 and RTX 2080Ti GPUs. Each V100 GPU can fit a batch of 256 examples at a given time, so we use 32 total

	VirtualFlow			TF*	
GPUs	BS	VN_{GPU}	Acc (%)	BS	Acc (%)
1	8192	32	75.92	256	69.25
2	8192	16	75.96	512	67.30
4	8192	8	75.99	1024	70.68
8	8192	4	75.83	2048	73.04
16	8192	2	75.68	–	–
2†	8192	32	76.01	–	–
Target	8192	–	76.26‡	–	–

Table 1. **Reproducibility:** Final top-1 validation accuracies for the same ResNet-50 experiment shown in Figure 7. VirtualFlow preserves the target accuracy of 76% (± 0.5%) regardless of the number of GPUs assigned, while the naive solution (TF*) diverges. VN_{GPU} refers to number of virtual nodes per GPU, † refers to training on RTX 2080Ti GPUs instead of on V100 GPUs, and ‡ refers to the accuracy achieved in (Goyal et al., 2017)

virtual nodes for these runs. For the smaller RTX 2080Ti GPUs, we use 64 total virtual nodes instead.

Table 1 demonstrates VirtualFlow can reproduce the target accuracy for all runs (±0.5%) across different numbers and types of GPUs. Previously, this workload required 32 V100 GPUs. With VirtualFlow, however, the user can reproduce the results for the same workload on even a single GPU. In contrast, attempts to reproduce this workload on fewer GPUs without retuning the hyperparameters (TF*) led to diverged models, e.g., doing so on 1 GPU led to a final accuracy of only 69.25%, far short of the target 76%. Additionally, VirtualFlow preserves not only the final accuracy but also the entire convergence trajectory (Figure 7).

6.2.2 *BERT-BASE Finetuning on GLUE*

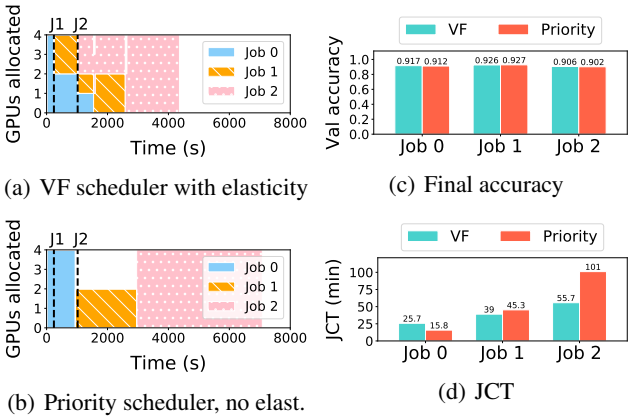
We also fine-tuned BERT-BASE on the GLUE dataset (QNLI, SST-2, and CoLA) using a fixed batch size of 64. For QNLI and SST-2, we use 1/10th of the original dataset in each epoch and train for 20 epochs. For CoLA, we train on the whole dataset for 50 epochs. As before, VirtualFlow was able to reproduce the target accuracies (obtained by running vanilla TensorFlow) across different numbers of GPUs for all GLUE tasks by preserving the batch size and the total number of virtual nodes (Table 2). Unlike in the ResNet case, however, the baseline also happened to converge to the same accuracies in these workloads (not shown). This illustrates that these workloads are less sensitive to a changing batch size within this range (8 to 64). While VirtualFlow did not lead to higher accuracies in this case, it still guaranteed that results for the batch size of 64 can be consistently reproduced across different sets of resources.

6.3 Resource Elasticity

In this section, we evaluate VirtualFlow's ability to dynamically resize a job while preserving model semantics and highlight the cluster-level benefits of this approach.

			QNLI	SST-2	CoLA
GPUs	BS	VN_{GPU}	Acc (%)	Acc (%)	Acc (%)
1	64	8	90.86	92.07	83.01
2	64	4	91.05	92.35	84.08
4	64	2	90.86	92.20	83.50
8	64	1	90.88	91.86	82.45
Target	64	–	90.90	91.97	82.36

Table 2. **Reproducibility**: Final top-1 validation accuracies for fine-tuning BERT-BASE across 3 GLUE tasks using VirtualFlow. VirtualFlow was able to reproduce the same training results as the state-of-the-art on a variety of number of GPUs. Previously, a batch size of 64 would not fit in the memory of 1 V100 GPU.



(a) VF scheduler with elasticity

(c) Final accuracy

(b) Priority scheduler, no elast.

(d) JCT

Figure 8. **Elasticity** with VirtualFlow reduces the makespan by 38% and the job completion time (JCT) for the highest priority job by 45%, while preserving model accuracies. In this workload, 3 jobs share 4 V100 GPUs on a single machine.

6.3.1 *Elastic Scheduling with Three Jobs*

Using the scheduler described in §4.2, we ran two traces with and without VirtualFlow. The first is a simple 3-job trace designed to illustrate a scenario in which elasticity can have significant effects on cluster-level objectives. Job 0 fine-tunes BERT-BASE on SST-2, Job 1 trains ResNet-56 on cifar10 (Krizhevsky), and Job 2 fine-tunes BERT-BASE on QNLI. The BERT jobs both demand 4 GPUs, while the ResNet job demands only 2 GPUs. The jobs arrive in the order of increasing priority, with Job 2 being the highest.

Figure 8 compares running this trace with the elastic, WFS scheduler in VirtualFlow to running it with a simple, non-elastic priority scheduler. With VirtualFlow, existing jobs downsize as soon as a new job with priority arrives. With the static priority scheduler, however, the high priority Job 2 is stuck behind Job 1 for a long time, leaving 2 GPUs idle for the entire duration of Job 1.

Observe that although all 3 jobs resized over the course of their respective lifetimes in the VirtualFlow case, they all converged to the same accuracies as their counterparts in the simple priority scheduler case. Thus, the VirtualFlow scheduler is able to reduce the makespan by 38% and the

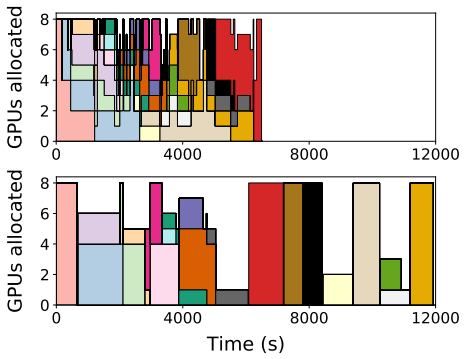


Figure 9. **Elasticity** with VirtualFlow (top) increases average cluster utilization by 19.5% and reduces makespan by 45.5%, compared to a simple priority scheduler (bottom) that does not perform elasticity. Each colored box corresponds to a job. Boxes resize for the elastic scheduler but not for the static scheduler.

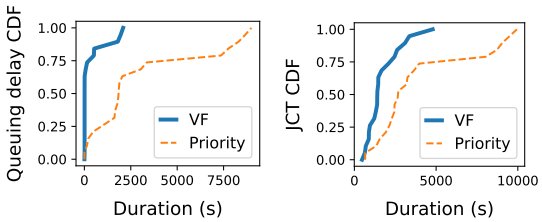


Figure 10. **Elasticity:** In the same 20 job experiment shown in Figure 9, VirtualFlow reduces the median JCT by 47.6% and the median queuing delay by 99.3% by resizing jobs dynamically.

high priority job completion time (JCT) by 45%, while preserving the application-level semantics of each job.

6.3.2 *Elastic Scheduling with Twenty Jobs*

Next, we evaluate VirtualFlow on a more realistic trace consisting of 20 jobs arriving according to a poisson distribution, with an average load of 12 jobs per hour (average interarrival time of 5 minutes). The workloads used in this trace are selected uniformly at random from Table 4 (Appendix). To speed up the experiment, we train each job for only a subset of the steps or epochs needed for convergence.

Figure 9 depicts the GPU allocations for both schedulers over time. Enabling elasticity with VirtualFlow improved average cluster utilization from 71.1% to 90.6%, reduced the makespan by 45.5%, the median JCT by 47.6%, and the median queuing delay by 99.3% (Figure 10).

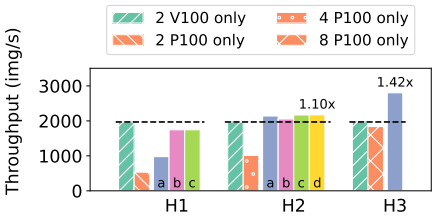
6.4 Heterogeneous Training

Figure 11 demonstrates the effectiveness of heterogeneous training across different sets of resources compared to homogeneous training. Detailed configurations regarding these experiments can be found in Table 3.

Heterogeneous configuration H3 significantly outperformed both the V100 only (by 42.3%) and the P100 only (by 52.4%) homogeneous configurations. Compared to H1 and

	V100			P100		
Exp	Num	BS_{GPU}	VN_{GPU}	Num	BS_{GPU}	VN_{GPU}
H1a	1+1	2048	8	1+1	2048	8
b	1+1	3072	16	1+1	1024	4
c	1+1	3072	32	1+1	1024	4
H2a	1+1	3072	16	2+2	512	2
b	1+1	3072	16	2+2	512	4
c	1+1	3072	16	2+2	512	8
d	1+1	3072	16	2+2	512	16
H3	1+1	2048	8	4+4	512	2

Table 3. **Heterogeneous training** configurations for ResNet-50 on ImageNet (batch size 8192). Columns BS_{GPU} and VN_{GPU} refer to the batch size and number of virtual nodes assigned to each GPU of the given type respectively, and 1+1 in the Num column refers to 2 servers with 1 GPU each.



	Acc (%)
H1b	75.97
H2a	75.92
H3	75.80

Figure 11. **Heterogeneous training** can improve throughput by up to 42% while converging to same target accuracy (76%) as homogeneous training. Experiment H3 had the largest improvement because the V100 only and the P100 only throughputs are the most similar. The specific configurations can be found in table 3.

H2, H3 is best able to balance the step times of the two individual GPU types. This is because, for this workload, V100 GPUs are roughly 4x as fast as P100 GPUs, and H3 uses 4 times as many P100 GPUs as V100 GPUs. In the H1 group, the V100 only configuration is the most efficient because there are not enough P100 GPUs to compensate for the difference in performance. For this group, VirtualFlow's heterogeneous solver fell back on recommending the more efficient V100 only configuration.

Importantly, all heterogeneous configurations converged to the target accuracy of 76% (Goyal et al., 2017), the same as homogeneous training. Further, VirtualFlow's heterogeneous solver accurately predicted throughputs for this set of experiments (Figure 12).

6.4.1 *Heterogeneous Scheduler*

To illustrate the benefits of VirtualFlow's heterogeneous training in a multi-tenant environment, we extended Gavel (Narayanan et al., 2020) to additionally consider heterogeneous allocations. Although Gavel was designed for heterogeneous clusters, it only considers *homogeneous* allocations. We evaluate our implementation in a simulated environment consisting of 4 V100, 8 P100, and 16 K80 GPUs, drawing from a subset of the workloads in Table 4. Following their evaluation, we use a round duration of 6 minutes and their formulation of the LAS objective.

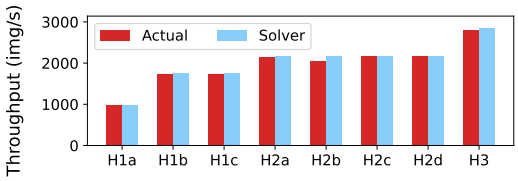


Figure 12. **Heterogeneous solver** produces throughputs within 5.6% of actual throughputs on average (experiments from Table 3).

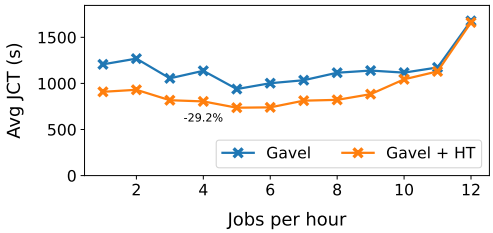


Figure 13. **Het. scheduler:** Extending Gavel (Narayanan et al., 2020) to additionally consider heterogeneous allocations can reduce the average JCT by up to 29.2%. The cluster consists of 4 V100 GPUs, 8 P100 GPUs, and 16 K80 GPUs. (Simulation)

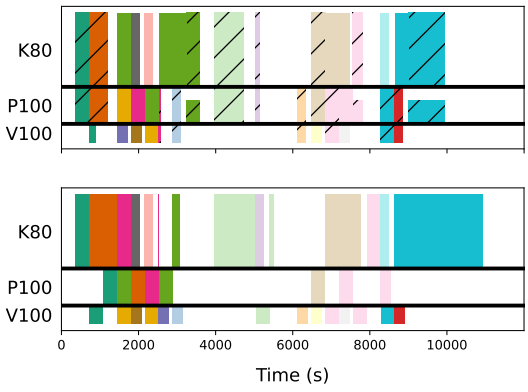


Figure 14. **Heterogeneous scheduler:** An example trace where Gavel (Narayanan et al., 2020) with heterogeneous allocations (top) reduces the average job completion time by 26.4% compared to Gavel without (bottom). Each colored box refers to an allocation, and each hatched box refers to a heterogeneous allocation. In this trace, 8 jobs arrive per hour on average. (Simulation)

In this experiment, using heterogeneous allocations allowed the scheduler to reduce the average job completion time by up to 29.2% (Figure 13). At higher job arrival rates, the benefits of using heterogeneous allocations diminishes, however, and the system gracefully falls back to prior behavior. Figure 14 illustrates a specific example of how individual jobs can train faster on multiple types of GPUs if there are idle resources, e.g., the rightmost job's throughput improved by 33.7% with 5 extra P100 GPUs in addition to the 16 K80 GPUs already assigned to it.

6.5 Microbenchmarks

Virtual node processing adds a gradient buffer to aggregate gradients across virtual nodes. This gradient buffer is the same size of the model (Figure 15 top): larger models like BERT see up to 16.2% memory overhead. Beyond 2 virtual nodes, however, this memory overhead stays constant.

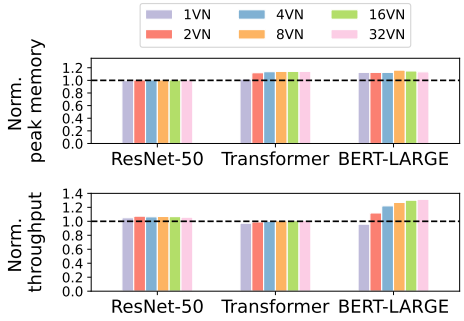


Figure 15. Peak memory and throughput on a single RTX 2080Ti GPU, normalized by the values from TensorFlow. Memory overhead scales with the model size and is constant across virtual nodes (top), while throughput scales with the number of virtual nodes for large models, due to fewer model updates (bottom).

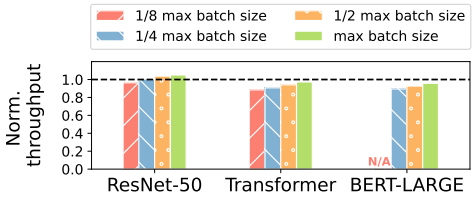


Figure 16. Overhead on a single RTX 2080Ti GPU for batch sizes that fit within the GPU's memory, normalized by throughputs from TensorFlow. The max batch sizes for ResNet-50, Transformer, and BERT-LARGE on this GPU are 192, 3072, and 4 respectively.

Figure 15 (bottom) plots the throughput across a range of virtual nodes for the same three workloads. The global batch size increases with the number of virtual nodes, and so the frequency of potentially expensive model updates decreases proportionally. For these workloads, using virtual nodes at worst lowers the throughput by 4.2% but can increase it by 31.4% in some cases, especially when the model is large (BERT), since updating large models is expensive.

Figure 16 plots the overhead for workloads that already fit within the accelerator memory. In all workloads considered, the overhead is minimal; the throughput of using virtual nodes is within 88.4% of the throughput without using virtual nodes. Note that for these single accelerator workloads, the user can simply disable virtual nodes since the job likely will not benefit from elasticity or heterogeneous training.

7 Future Directions

The virtual node abstraction is not limited to the above use cases. For instance, VirtualFlow can be extended to support:

Fault tolerance. We can reuse existing elasticity mechanisms to migrate virtual nodes from failed workers to remaining healthy ones, and then to the new replacement

workers when they become available. This would ensure training is uninterrupted from the application's perspective. In contrast, state-of-the-art solutions must restart the job from potentially stale checkpoints if even a single worker fails (Mohan et al., 2021), since the model graph does not support changes in cluster membership.

Model parallelism. Training extremely large models (Brown et al., 2020; Rosset, 2020; Shoeybi et al., 2019) relies on *model parallelism*, which partitions, instead of replicates, the model across the accelerators in the system. The model can be partitioned by layer or groups of layers (as in pipeline parallelism (Huang et al., 2019)) and/or by slices within each layer (spatial partitioning (Shazeer et al., 2018)). In both cases, model parallelism is often used in conjunction with data parallelism (Huang et al., 2019; Jia et al., 2019), where each partition of the model is additionally replicated across multiple accelerators, and the input batch is divided evenly among these accelerators (Figure 17, top).

The techniques presented in this paper can still be applied to this setting to reduce resource requirements along the *batch* dimension. More specifically, within each model partition, the input batch can be divided among virtual nodes rather than accelerators as before. The system would effectively unroll the data parallel pipelines into sequential forward and backward passes (Figure 17, bottom), thus trading off compute time for lower resource requirement. This would bring the benefits of reproducibility, elasticity and heterogeneous training to the model parallelism setting as well. Exploring how to pipeline these virtual nodes for higher efficiency (as in GPipe (Huang et al., 2019) and PipeDream (Narayanan et al., 2019)) would be an interesting future direction.

8 Related Work

Gradient accumulation. The execution of virtual nodes is similar to gradient accumulation in PyTorch (Li et al., 2020) and a variant of asynchronous training that synchronizes gradients every k steps (Zhou & Cong, 2018; Wang & Joshi, 2019; Zhao et al., 2019). VirtualFlow is a generalization of these approaches: virtual nodes not only allow users to simulate larger batch sizes, but also provide a general abstraction that enables elasticity and heterogeneous training.

Virtual nodes. The use of virtual nodes to decouple from hardware is not new. Chord (Stoica et al., 2001) uses virtual nodes to map multiple ring segments to the same server, and Dynamo (DeCandia et al., 2007) uses virtual nodes to dynamically balance load across servers in the system. VirtualFlow borrows from these ideas.

Elasticity mechanism. Resource elasticity for deep learning has been explored in (Or et al., 2020), and our implementation builds on top of the resizing mechanisms they introduced. (Elastic Horovod) and (TorchElastic) also provide elasticity mechanisms for deep learning. Unlike VirtualFlow, however, these approaches do not provide model convergence guarantees as they allow the global batch size to change throughout training.

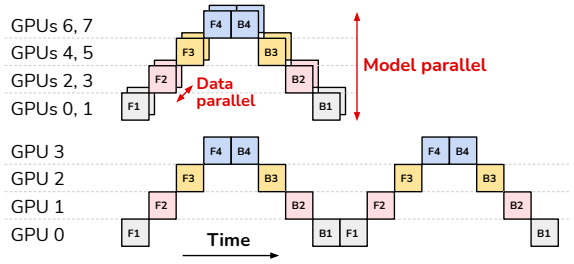


Figure 17. Model parallelism combined with data parallelism today (top), versus model parallelism with virtual nodes (bottom), which halves the resource requirement for this job. This can be further optimized by pipelining the virtual nodes, which would overlap boxes F1 and F2 for example, as in GPipe (Huang et al., 2019).

Cluster scheduling. Multi-tenant GPU cluster schedulers such as Optimus (Peng et al., 2018), Tiresias (Gu et al., 2019), Gandiva (Xiao et al., 2018), Themis (Mahajan et al., 2020), and Gavel (Narayanan et al., 2020) dynamically migrate jobs across GPUs to achieve various cluster-level objectives. However, unlike VirtualFlow, these approaches constantly interrupt and restart jobs when adjusting their resource allocations (e.g., every 6 minutes (Narayanan et al., 2020)), and assume fixed resource requirements for each job, thus limiting the scheduling options available. Antman (Xiao et al., 2020) can adjust resource allocations seamlessly by swapping to and from host memory, but only for co-located jobs that share the same accelerators.

Heterogeneous training. Gavel (Narayanan et al., 2020) introduces policies for heterogeneous clusters, but only makes homogeneous allocations. Integrating their scheduler with VirtualFlow's heterogeneous training can open up additional scheduling options and improve cluster utilization (§6.4.1).

9 Conclusion

VirtualFlow is an important step towards decoupling deep learning models from the underlying hardware. With virtual nodes, VirtualFlow allows users to reproduce training results consistently across different clusters, reap the benefits of resource elasticity without worrying about model convergence, and combine multiple accelerator types to improve training throughput, all without a single change to the model specification or the hyperparameters.

The benefits of virtual nodes are not limited to the use cases explored in this paper. In the future, we expect to see more complexities associated with resource management pushed into the deep learning frameworks themselves, enabling the user to focus on application-level objectives instead.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

Apache Flink. https://flink.apache.org/.

Apache Storm. https://storm.apache.org/.

AWS. Auto Scaling: https://aws.amazon.com/autoscaling/.

Azure. Autoscale: https://azure.microsoft.com/en-us/features/autoscale/.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language Models are Few-Shot Learners. *arXiv:2005.14165*, 2020.

Chen, J., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting Distributed Synchronous SGD. In *4th International Conference on Learning Representations (ICLR) Workshop Track*, 2016. URL https://arxiv.org/abs/1604.00981.

Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review*, 2007.

Demers, A., Keshav, S., and Shenker, S. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *22nd IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805*, 2018.

Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.

GCE. Autoscaling groups of instances: https://cloud.google.com/compute/docs/autoscaler/.

Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2013.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv:1706.02677*, 2017.

Gross, S. and Wilber, M. Training and Investigating Residual Nets. https://github.com/facebook/fb.resnet.torch, 2016.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *33rd Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

Ioffe, S. and Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *32nd International Conference on Machine Learning (ICML), PMLR*, 2015.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *25th USENIX Annual Technical Conference (ATC)*, 2019.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv:1807.11205*, 2018.

Jia, Z., Zaharia, M., and Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. 2019.

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. *arXiv:1609.04836*, 2016.

Krizhevsky, A. Convolutional Deep Belief Networks on CIFAR-10.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling Distributed Machine Learning with the Param-

eter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv:2006.15704*, 2020.

Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

Mohan, J., Phanishayee, A., and Chidambaram, V. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

Narasimhan, S. NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI. https://developer.nvidia.com/blog/training-bert-with-gpus/, 2019.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

Or, A. Apache Spark Dynamic Resource Allocation: https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation, 2014.

Or, A., Zhang, H., and Freedman, M. J. Resource Elasticity in Distributed Deep Learning. In *3rd Conference on Machine Learning and Systems (MLSys)*, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *33rd Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *13th European Conference for Computer Systems (EuroSys)*, 2018.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683*, 2019.

Rosset, C. Turing-NLG: A 17-Billion-Parameter Language Model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/, 2020.

Sergeev, A. and Del Balso, M. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv:1802.05799*, 2018.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the Effects of Data Parallelism on Neural Network Training. *arXiv:1811.03600*, 2018.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-TensorFlow: Deep Learning for Supercomputers. In *32nd Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using GPU Model Parallelism. *arXiv:1909.08053*, 2019.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review*, 2001.

Sun, P., Feng, W., Han, R., Yan, S., and Wen, Y. Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes. *arXiv:1902.06855*, 2019.

Szczepkowski, J. and Wielgus, M. Autoscaling in Kubernetes: https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/, 2016.

Thakur, R., Rabenseifner, R., and Gropp, W. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications (IJHPCA)*, 2005.

TorchElastic. https://pytorch.org/elastic.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is All You Need. In *31st Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding.

In *7th International Conference on Learning Representations (ICLR)*, 2019.

Wang, J. and Joshi, G. Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD. In *2nd Conference on Systems and Machine Learning (SysML)*, 2019.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Xiao, W., Ren, S., Li, Y., Zhang, Y., Hou, P., Li, Z., Feng, Y., Lin, W., and Jia, Y. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

You, Y., Gitman, I., and Ginsburg, B. Scaling SGD Batch Size to 32k for ImageNet Training. *arXiv:1708.03888*, 2017.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes. In *7th International Conference on Learning Representations (ICLR)*, 2019.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

Zhao, X., Papagelis, M., An, A., Chen, B. X., Liu, J., and Hu, Y. Elastic Bulk Synchronous Parallel Model for Distributed Deep Learning. In *19th IEEE International Conference on Data Mining (ICDM)*, 2019.

Zhou, F. and Cong, G. On the Convergence Properties of a K-step Averaging Stochastic Gradient Descent Algorithm for Nonconvex Optimization. In *27th International Joint Conference on Artificial Intelligence, (IJCAI)*, 2018.

A APPENDIX

A.1 Hyperparameter Exploration

Another use case of virtual nodes is to explore hyperparameters previously inaccessible on the same set of resources. To achieve this, we vary the number of virtual nodes, and consequently the batch size, while holding the number of GPUs constant, i.e., the opposite of the previous experiments. We fine-tune BERT-LARGE on three GLUE tasks, RTE, SST-2, and MRPC, for 10 epochs on a single RTX 2080Ti GPU.

Model	Dataset	Batch sizes	VN_{GPU}
ResNet-56	cifar10	64, 128	1
ResNet-50	ImageNet	256, 512, 1024 2048, 4096, 8192	1, 2, 4
BERT-BASE	CoLA	8, 16, 32, 64, 128	1, 2
BERT-BASE	SST-2	8, 16, 32, 64, 128	1, 2
Transformer	WMT	4096, 8192, 16384 32768, 65536	1, 2

Table 4. **Elasticity:** Mix of workloads used in 20-job experiment. Each job is selected uniformly at random from this set.

Figure 18 plots the model convergence for this experiment. Unlike before, since the batch size changes across runs, so do the convergence trajectory and the final accuracy. This allows the user to explore the convergence characteristics of various batch sizes, without deploying the resources that would have been necessary to run these batch sizes using vanilla TensorFlow (e.g., 32 GPUs for a batch size of 128).

In some cases, VirtualFlow can even achieve higher accuracies in the batch sizes explored. For RTE (a reading entailment task), using a larger batch size of 16 is now possible on 1 GPU, even though this batch size required 4 GPUs before. This configuration ended up improving the final accuracy by 7 percentage points on the same set of resources.

A.2 Model Update Frequency

Though not a main goal, an important side effect of using virtual nodes is improved throughput. This results from being able to use larger batch sizes than was previously possible, which reduces the number of expensive gradient synchronizations and model updates proportionally.

In the same set of experiments described in Appendix A.1, using a larger batch size reduced the training time by lowering the model update frequency for all the tasks considered. (Figure 19). For RTE, using a batch size of 16, as enabled by VirtualFlow, not only improved the final accuracy by 7.1%, but also improved the throughput by 18.5%. Using a batch size of 128 further improved the throughput by 28.7% without affecting convergence.

Thus, even if VirtualFlow did not improve the final accuracy of the model (which was not a goal of VirtualFlow in the first place), it can still help reduce the training time by lowering the model update frequency. In the distributed setting, this also reduces the number of expensive gradient synchronizations across the network.

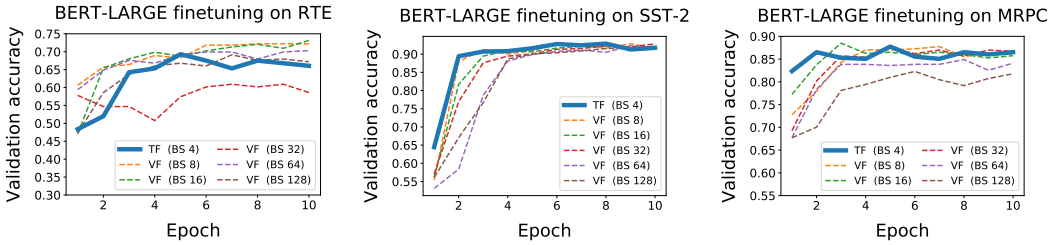


Figure 18. **Batch size exploration** with VirtualFlow on a single RTX 2080 Ti GPU. VirtualFlow expands the space of possible batch sizes on this GPU from 4 (TF) to [4, 8, 16, 32, 64, 128], and can support even larger batch sizes. In some cases, such as in RTE (left), being able to access larger batch sizes can lead to significantly higher final accuracies (+7.1% with a batch size of 16).

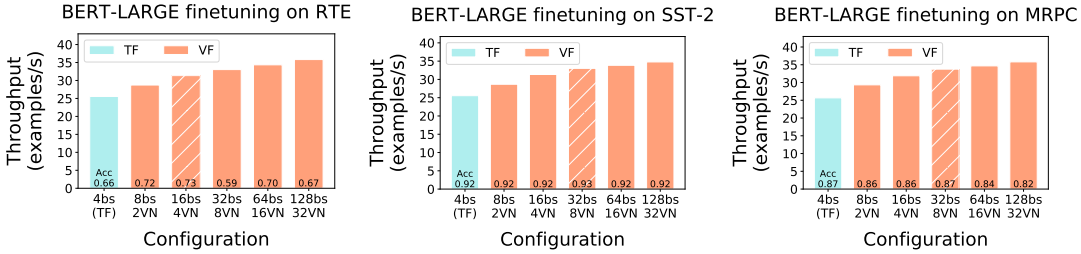


Figure 19. **Batch size exploration**. Throughputs of the same experiment shown in Figure 18. For RTE, using VirtualFlow not only leads to higher accuracies, but also higher training throughputs (up to +18.5% using 16BS, or +28.7% using 128BS). Other tasks see similar throughput improvements. The number at the bottom of each bar refers to the final accuracy achieved in that run, and the hatched bar represents the configuration with the highest final accuracy within each task.