Varuna: Scalable, Low-cost Training of Massive Deep Learning Models

Abstract

Systems for training massive deep learning models (billions of parameters) today assume and require specialized "hyper-clusters": hundreds or thousands of GPUs wired with specialized high-bandwidth interconnects such as NV-Link and Infiniband. Besides being expensive, such dependence on hyper-clusters and custom high-speed inter-connects limits the size of such clusters, creating (a) scalability limits on job parallelism; (b) resource fragmentation across hyper-clusters.

In this paper, we present *Varuna* a new system that enables training massive deep learning models on commodity networking. *Varuna* makes thrifty use of networking resources and automatically configures the user's training job to efficiently use any given set of resources. Therefore, *Varuna* is able to leverage "low-priority" VMs that cost about 5x cheaper than dedicated GPUs, thus significantly reducing the cost of training massive models. We demonstrate the efficacy of *Varuna* by training massive models, including a 200 billion parameter model, on 5x cheaper "spot VMs", while maintaining high training throughput. Even in scenarios where hyper-cluster resources are available, *Varuna* improves end-to-end training time by 20-78% compared to alternative approaches.

ACM Reference Format:
. 2021. Varuna: Scalable, Low-cost Training of Massive Deep Learning Models. In *Proceedings of ACM Conference (Conference'21)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

State-of-the-art deep learning models that power important applications such as web search, have seen a rapid growth in number of model parameters. For example, in natural language understanding, BERT-large [14] enabled significant accuracy improvements with a model architecture that has 340 million parameters, significantly larger than any other model at that time. Subsequently, GPT-2 [30] (1.5 billion parameters), Megatron [35] (8 billion parameters), Turing-NLG [8] (17 billion parameters), and GPT-3 [11] (175 billion parameters) have pushed model sizes much higher, improving accuracy even further in the process.

Such massive models require several petaflops of compute, so they need to be run on large number of GPUs. Because the training is also communication-intensive, such massive jobs are usually run on specialized "hyperclusters" that have expensive, high-speed interconnects such as NVLink or Infiniband. Even in these hyperclusters, Megatron 8B model takes roughly 11 days to train on 512 GPUs [35]. Thus, training such large models can be expensive. Authors in [33] estimate that fully-loaded training cost, which includes multiple training runs with hyper-parameter tuning, for BERT-large to be $200K.

Besides cost, the dependence on specialized hyperclusters for training massive models is also sub-optimal for other reasons. First, specialized high-speed interconnects are economically infeasible to scale beyond a certain cluster size. For example, Nvidia's NVLink connects 16 GPUs within a DGX-2 server at 2.4 Tbps all-to-all bandwidth but the DGX-2 servers themselves are connected with each other using Infiniband at only 800 Gbps [5]. Thus, the degree of parallelism for training massive models is limited by the architecture and size of a single hypercluster (*e.g.*, 1000 GPUs). Second, multiple siloed hyper-clusters cause resource fragmentation, as GPUs are not fungible across hyper-clusters.

In this paper, we present *Varuna*, a system that trains massive deep learning models on commodity networking, without requiring specialized hyperclusters, thus addressing the three problems with existing approaches: cost, scale, and resource utilization. As *Varuna* does not depend on specialized networking, it can orchestrate a job in any set of GPUs in the data center, thus improving resource utilization.

Interestingly, the ability of *Varuna* to harness scattered GPUs in the data center makes it possible to train massive jobs on "low-priority" or "spot" VMs that public clouds offer [10, 23]. Such spot VMs are offered at a significant discount (*e.g.*, 4-5x cheaper) compared to dedicated VMs, as they allow the cloud provider to sell unused spare capacity, and manage capacity better during load spikes. By opening up such cheap VMs for training massive models that today run on hyperclusters, *Varuna* reduces the cost of training such models by 4-5x, without losing performance.

There are three key challenges that *Varuna* addresses: (a) dealing with "slow"/flaky network, (b) dealing with transient pre-emptible resources, and (c) being transparent/non-intrusive to the programmer.

First, *Varuna* handles low-bandwidth networks with a new variant of *pipeline parallelism* combined with data parallelism within each pipeline stage to train massive models. We show that inter-layer or pipeline-partitioning (*e.g.*, GPipe [20], PipeDream [24]) is more tolerant of slow networks compared to the more popular *intra-layer* partitioning (employed by Mesh-Tensorflow [34], Megatron [35], Turing-NLG [8]). To improve pipeline efficiency, *Varuna* uses a large number of *micro-batches* within a mini-batch, similar to Gpipe, but enhanced with a *novel, micro-batch scheduling algorithm* that is more efficient and tolerant to network jitter. Further, unlike traditional pipeline partitioning that tries to minimize the number of partitions (*i.e.,* each partition fits as much work as possible, constrained only by GPU memory), stages in *Varuna* are also constrained by the network bandwidth. As the synchronization of gradients for data parallelism happens only within a partition, *Varuna* limits bandwidth usage by scaling the number of pipeline stages as model size increases. One issue with large number of micro-batches is that it increases mini-batch size, raising model accuracy concerns; we address it by demonstrating, for the first time, that a large 2.5 billion-parameter GPT-2 model can be trained to the same accuracy, despite using a 16x larger mini-batch.

Second, *Varuna* handles frequent pre-emptions of low-priority VMs by employing dynamic, semantics-preserving reconfiguration of the training job. Existing approaches for elasticity like PyTorch Elastic [7] and MXNet Dynamic [2] require users to provide different sets of hyper-parameters and mini-batch sizes for a varying number of resources. Recent work [27] allows the user to specify a single mini-batch size, but suffers poor performance at high scaling factors. Crucially, all the above approaches only handle data parallelism. Instead, *Varuna* introduces *job morphing*, where it morphs the configuration of a large parallel job across both the pipeline depth and data parallelism dimensions, to fit in as much resources as available, *without changing hyper parameters*. *Varuna* piggybacks on the gradient accumulation that micro-batching performs, adapting to a wide range of "local" batch-sizes. For such morphing, *Varuna* uses a novel mechanism of micro-benchmark-driven simulation, where a small set of micro-parameters are calibrated through profiling, that are then fed to an efficient simulator, to pick the best performing configuration for different number of GPUs. *Varuna* also deals with failures and stragglers, inherent to the spot-VM setup.

Third, *Varuna* addresses a key usability challenge of model partitioning, by making it non-intrusive on the programmer, unlike existing approaches that require significant changes to the model [20, 31]. *Varuna* provides flexibility to the user in writing the model as if it runs on a single GPU. To achieve this, *Varuna* introduces a novel mechanism of automatically identifying *cut-points* in the model, which are a *super-set* of potential "safe" partitioning points. *Varuna* then groups multiple of these cut-points into a single partition. Another challenge is implicit usage of cross-partition state (*e.g.*, global computations in optimizers like NVLAMB [6], shared weights, *etc.*.), either directly in the user's code or indirectly in libraries (such as APEX [4]) used by the model; *Varuna* includes a tracer that identifies such cross-partition sharing automatically across libraries, and flags those tensors to be synchronized across partitions.

We evaluate *Varuna* by using it to train several massive models: BERT-large (340 million parameters), GPT-2 models with various sizes: 2.5 billion, 8.3 billion, 20 billion, and 200 billion. All these models are trained on *low-priority VMs* in Azure that cost 4-5x cheaper than their dedicated counterparts. On such commodity VMs, we show that *Varuna* improves the performance of training such models by up to 18x compared to state-of-the-art approaches. We also show that despite the commodity networking across these VMs, *Varuna* is able to *outperform* state-of-the-art approaches that run on specialized hyperclusters by 20%-78%.

We make the following key contributions in this paper:

- We challenge the pervasive belief (and practice) that massive models can be trained only on specialized hyperclusters, by presenting the first system that is capable of training massive deep learning models on spot VMs with commodity networking, achieving 4-5x lower cost of training these models.
- We argue and demonstrate that intra-layer partitioning is ill-suited not only for commodity networking but that they are *not the best performing option even in hyperclusters*.
- We introduce a novel concept of correctness-preserving *job morphing* to automatically reconfigure a running DLT job, to adapt to changing number of GPUs, using a combination of data and model parallelism.
- We demonstrate the efficacy of this approach by efficiently scaling to a 200 billion parameter model, and showing significant speedups on other large models such as BERT-large and Megatron-8.3B. We also demonstrate that despite using a large batch size, *Varuna* achieves state-of-the-art accuracy on a 2.5 billion parameter GPT-2 model.

The code for *Varuna* has been open-sourced (link removed for double blind review).

2 Background and Related Work

In this section, we present a brief background of Deep Learning Training jobs (DLT jobs) and review today's architectures for training massive models.

A DLT job is typically a python script that uses frameworks such as PyTorch [28] or TensorFlow [9] to define the model and the training procedure. In each training iteration, the DLT job takes a few samples of data called the *mini-batch* as input and performs a *forward pass* over this data. The forward pass consists of applying the model function on the input data to compute a loss value. The loss value is then propagated in a *backward pass* that computes a *gradient* for each model parameter. Finally, the model parameters are updated by adding the negative of the gradient, scaled by a hyper-parameter called the *learning rate*. Millions of such iterations are typically necessary to learn an accurate

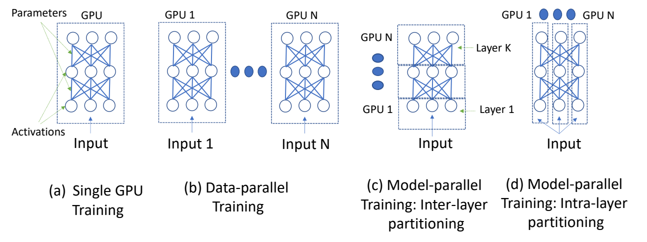

Parameters
GPU
Activations
Input
GPU 1
GPU N
Input 1
Input N
GPU N
Layer K
GPU 1
Layer 1
Input
GPU 1
GPU N
Input
(a) Single GPU Training
(b) Data-parallel Training
(c) Model-parallel Training: Inter-layer partitioning
(d) Model-parallel Training: Intra-layer partitioning

Figure 1. Various architectures for training DLT jobs

model. Since the forward and backward passes involve billions of floating point operations, they are performed in a GPU. The model parameters as well as gradients remain in GPU memory during training for maximum efficiency.

Data-parallel Training. A straightforward way to reduce training time is to replicate the model in N GPUs as shown in Figure 1(b). Each GPU performs forward and backward passes of each iteration independently but then use a technique called all-reduce [29] to synchronously compute an average gradient, that is then used to update the model.

Model-parallel Training. While data-parallel is mainly used to speed up training, model-parallel training helps with fitting massive models by spreading the parameters of a single model across multiple GPUs. Model-parallel training can be inter-layer and/or intra-layer.

Inter-layer or pipeline parallelism. A deep learning model is divided into a number of layers. Thus, a natural way to split a model among multiple GPUs is to distribute different layers among them. This is known as inter-layer or pipeline partitioning and is shown in Figure 1(c).

Gpipe [20] was one of the first systems to use pipeline partitioning to split a large model (6 billion parameters) among a number of GPUs. Gpipe follows the *synchronous SGD* semantics where gradients are applied synchronously at the end of a mini-batch but this can result in "bubble overhead" where many pipeline stages are stalled. Gpipe uses micro-batches to reduce the pipeline overhead but assumes that the user provides an optimized, partitioned model as input.

Unlike Gpipe, PipeDream [24] achieves 100% pipeline utilization in steady state. To achieve this, PipeDream sacrifices synchronous SGD semantics as the parameter update is stale/delayed. However, the cost of stale updates is that some models may fail to converge. For example, authors in [40] show that PipeDream fails to converge on certain language translation tasks. In addition, PipeDream keeps *P copies of parameters* where P is pipeline depth, which limits the size of the model that can be trained. PipeDream-2BW [25] reduces the number of copies to 2 but still suffers from stale updates.

PipeMare [40] tries to address PipeDream's convergence problem with new, approximate techniques to correct stale updates. While the authors show better convergence than PipeDream, they evaluate only on smaller models and haven't released their code for comparison. PipeMare also uses 25-33% more memory than Gpipe which is an issue for larger models.

DAPPLE [15] maintains *synchronous SGD* semantics and strictly interleaves forward and backward passes in its pipeline schedule. This avoids recomputing activations, but requires each stage to store intermediate activations for multiple micro-batches until the pipeline steady state is reached. This is infeasible for large models that require long pipelines. The largest model that DAPPLE shows performance speedup for is Bert-48 with 600M parameters.

Intra-layer parallelism. An alternative model-parallel approach is intra-layer partitioning where a single layer of a model is split across multiple GPUs as shown in Figure 1(d) Mesh-Tensorflow [34], Megatron [35], and Turing [8] adopt this approach (Megatron recently has added support for pipeline-parallelism [26]). Low network latency/jitter and high network bandwidth are critical requirements for this approach since a large matrix multiplication within a layer is now split among multiple GPUs. Thus, for efficiency, Megatron and Turing models use DGX-2s with the model partitioned only within the DGX-2 so that the communication can benefit from the 2.4 Tbps NVlink connectivity.

Memory optimization. Another aspect to fitting large models is optimizing GPU memory usage. A model with N parameters will need up to $16 * N$ bytes of memory to store parameters and optimizer state [31]. In addition, one needs the intermediate outputs of the forward computation, called the activations. The activation size depends on the model and batch size used and can be significant. Activations can be stashed in CPU memory and brought back to GPU as needed [13] but this can incur substantial overhead [22]. Instead, Gradient checkpointing [12, 21] is used, where the intermediate activations are not saved during forward pass and are recomputed (based on saved input activations for each layer) for the backward pass. Since forward pass takes about one-third of the iteration compute, this adds about 33% overhead. This approach is used by most systems including Gpipe, Megatron and *Varuna* to train large models.

ZeRO/DeepSpeed [31] optimizes memory usage in data-parallel training by sharding the redundant state among the replicas. This is complementary to systems like Gpipe and *Varuna*. In addition, Zero/DeepSpeed started with intra-layer and added inter-layer parallelism to fit massive models. Zero-infinity [32] extends Zero to carefully offload GPU memory to CPU and SSD, thereby enabling scaling to models with trillions of parameters.

Resource elasticity and Spot VMs. Recent frameworks such as PyTorch Elastic [7] and MXNet Dyanmic [2] support training over dynamic set of resources. However, these frameworks simply pass the burden of adapting to elasticity to the user who has to ensure that their scripts are configured with the right parameters for any resource availability. Authors in both [27] and [19] support auto-scaling but only for

System	Intra-Layer	Inter-Layer	Sync-SGD	User Ease	Low-Pri.
Mesh-Tensorflow	✓	X	✓	✓	X
Megatron/Turing	✓	✓*	✓	✓	X
Gpipe	X	✓	✓	X	X
Pipe(Dream/Mare)	X	✓	X	✓*	X
Zero/DeepSpeed	✓	✓*	✓	X	X
Varuna	X	✓	✓	✓	✓

Table 1. Systems for training massive models: Features

jobs that fit within a single-GPU. They increase the degree of data-parallelism (N) as long as the marginal utility of increase is higher than marginal cost. *Varuna* handles scaling automatically for massive jobs that use both model-parallel and data-parallel training. Proteus[19] enables training models over a mixed set of spot and dedicated VMs. But this approach is specific to parameter server framework and does not scale with large cluster and model sizes. Spotnik[38] introduces an adaptive all-reduce method for spot instances, but only works for smaller image models.

A summary of the various systems for training massive models is shown in Table 1. First, most systems initially started as either intra-layer or inter-layer but Megatron and Deepspeed have recently added inter-layer support (denoted by *), supporting both modes of model parallelism. Second, notice that only PipeDream and PipeMare do not follow synchronous SGD semantics and are thus susceptible to convergence issues. Third, systems that use intra-layer parallelism like Mesh-Tensorflow handle the partitioning automatically for the user. On the other hand, with Gpipe/DeepSpeed, a significant burden of partitioning the model is placed on the user. PipeDream has support for automatic partitioning based on profiling. However, they only support a set of whitelisted functions while models like BERT use several custom functions. Thus, one would need to rewrite such models to work with PipeDream. *Varuna* automatically annotates a model with cut-points for re-configuration and adds a tracer that identifies potential shared variables that may need to be synchronized, easing programming for the user. Finally, only *Varuna* enables training over low-priority VMs through its elasticity support.

3 Design Overview

In this section, we describe the design of *Varuna*, highlighting how it handles the constraints of commodity networks.

3.1 High-level architecture

Varuna uses a combination of data parallelism and *pipeline* model parallelism as shown in Figure 2. Each model with N parameters is partitioned into P partitions, and each partition has multiple replicas, running like a data-parallel job. Across the different partitions of the model, *Varuna* uses *pipeline parallelism*, where each stage k (other than the first and last stage) gets input activations from the previous stage

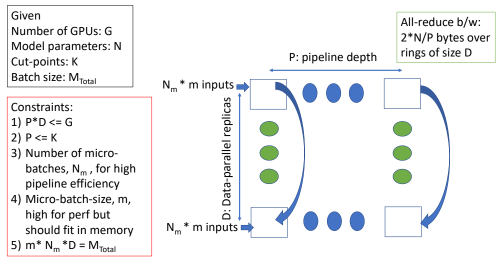

Given
Number of GPUs: G
Model parameters: N
Cut-points: K
Batch size: M_{Total}
Constraints:
1) P*D <= G
2) P <= K
3) Number of micro-batches, N_m , for high pipeline efficiency
4) Micro-batch-size, m, high for perf but should fit in memory
5) m* N_m *D = M_{Total}
P: pipeline depth
All-reduce b/w: 2*N/P bytes over rings of size D
N_m * m inputs
D: Data-parallel replicas
N_m * m inputs

Figure 2. Problem setting and constraints

$k-1$, performs the forward pass computation, and sends the output activations to the next stage $k+1$. Similarly, stage k receives gradients from stage $k+1$, performs the backward pass computation, and sends input gradients to stage $k-1$.

The backward computation requires the activations computed in forward pass, in order to compute the gradients for the previous stage. However, as described in § 2, activations take up a large amount of memory and hence massive models cannot remember activations. Instead, *Varuna* recomputes activations by re-running the forward computation [12] using the stored input activation for each layer. Input activation is a fraction of model size, e.g., for 2.5B GPT-2, this is only 3.75 MB per input example.

Observation 1: Pipeline parallelism instead of intra-layer parallelism. Conventional wisdom today is that intra-layer model-parallelism should be used to scale up to the number of GPUs within a server and only then pipeline parallelism be used (See takeaway 1 in [26]; Deepspeed and Megatron both started with only intra-layer parallelism).

In intra-layer parallelism, the matrix-multiplication computation in each layer is split between multiple GPUs. This requires two allreduces each in the forward, backward, and recompute passes for each layer. For each such allreduce, every GPU transfers 2 x hiddenSize x sequenceLength 16-bit floats in mixed precision training. For GPT-2 2.5B model with 54 layers, a hiddenSize of 1920, and a sequenceLength of 1024, the amount of data transferred is 2.4 GB/example/GPU. Further, these all-reduces are *synchronous*, which implies GPU computations wait until communication completes. In contrast, pipeline parallelism only communicates end of layer activations+gradients which are 7.5 MB/GPU/example for the same model ($\approx$ 300$\times$ smaller) and this communication can overlap with computation. Thus, *Varuna* eschews intra-layer parallelism in favor of pipeline parallelism. We show that *Varuna* outperforms intra-layer parallelism not only in low-priority settings but even in hypercluster settings.

Observation 2: Balancing pipeline overhead and all-reduce bandwidth. Pipeline bubble overhead is directly proportional to number of pipeline stages (P) [20] [15]. Thus, conventional wisdom is to minimize the number of pipeline stages (also why intra-layer is preferred today). However, when P is reduced, the all-reduce bandwidth (2N/P) increases. Further, D = G/P also increases, resulting in significantly

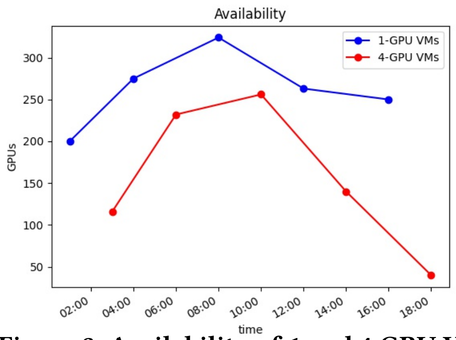
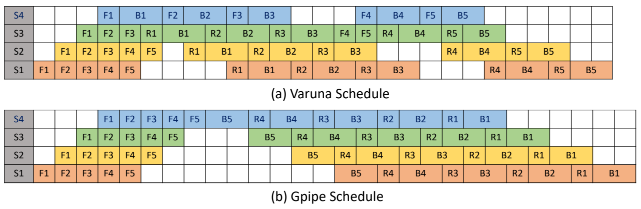


Figure 3. Availability of 1 and 4 GPU VMs



(a) Varuna Schedule

(b) Gpipe Schedule

Figure 4. ***Varuna*****'s micro-batch schedule contrasted against Gpipe. S denotes the stage, F denotes forward, B denotes backward and takes twice as long as forward, and R denotes recompute, with number representing micro-batches (1 to 5)**

higher network load. While this bandwidth increase is typically not an issue in hypercluster settings, it can significantly impact performance in commodity networks. Therefore, *Varuna* identifies an optimal balance between P and D that is unique to the low priority setting (Section 4).

Observation 3: Tolerating higher latency/jitter. Unlike hypercluster, commodity networks can suffer from high latency/jitter. *Varuna* addresses this constraint in several ways. First, *Varuna* uses a large number of *micro-batches* [20] to overlap communication with computation, thus moving latency off the critical path. Second, *Varuna*'s pipeline schedule (Section 3.2) is specifically designed to be able to opportunistically adapt to network jitter. Finally, *Varuna* explicitly profiles latency and jitter, and incorporates it in its simulation (Section 4) to identify the best parallelism configuration.

Observation 4: Single GPU vs Multi-GPU VMs. Our experiments show that for *low-priority VMs*, single GPU VMs are more readily available than 4-GPU VMs. For example, Figure 3 shows aggregate GPU availability when low-priority VMs with 1 and 4 GPUs are requested/released alternately in Azure over a duration of 16 hours. Thus, systems that can work with 1-GPU VMs can utilize higher aggregate capacity, albeit at the cost of stressing the network even more (PCIe replaced by Ethernet). Due to *Varuna*'s thrifty use of networking resources, *Varuna* is able to train on 1-GPU VMs at almost the same performance (2% difference) as 4-GPU VMs, thus enabling faster training completion.

3.2 Tuning pipeline efficiency

As in any pipeline, the efficiency of *Varuna* depends on reducing pipeline stalls. PipeDream [24] reduced stalls by sacrificing the semantics of synchronous stochastic gradient descent (SGD). *Varuna* follows the GPipe [20] approach that preserves the semantics of sync-SGD but uses a *novel schedule that is more efficient and more tolerant of network jitter.*

Varuna uses a combination of a static rule-based schedule enumerated for a given pipeline depth, along with an opportunistic policy that is employed to hide jitter. The rule-based schedule is generated based on a tool that enforces the following constraints:

1. If stage k will complete its backward pass for micro-batch m in time t, the recompute of gradients must be scheduled in stage $k-1$ at time t' such that $t - t' > T_f$, where T_f is the time taken per forward pass per stage (assuming uniform stages)
2. If a recompute has completed for stage k for micro-batch m, unconditionally wait for the corresponding backward task for m to be scheduled (as running a forward pass will create another set of activations, taking 2x the memory for activations)
3. If a stage k has both forward and backward tasks ready to be scheduled, prefer the backward task.

Each stage sticks to the above pre-defined offline schedule. Sometimes, the schedule for stage k may indicate that the backward pass for micro-batch m must be scheduled, but the gradients for m may not have arrived yet from stage $k+1$; in those cases, *Varuna* deviates from the schedule and opportunistically schedules another ready task (*e.g.*, forward pass for a micro-batch). This allows *Varuna* to be work-conserving during network jitter, reducing pipeline stalls.

A qualitative comparison of *Varuna*'s *static* schedule against GPipe's schedule for a 4-stage pipeline with 5 micro-batches is shown in Figure 4. First, note that *Varuna* is more efficient overall and uses 1 less time unit compared to Gpipe as it has fewer stalls (white space). Second, the whitespace in *Varuna* is distributed throughout the schedule while it is concentrated in the middle in Gpipe. This makes Gpipe's schedule more vulnerable to network jitter as there is no free time slots later in the schedule that can serve as a buffer. Third, the last stage (S4) in *Varuna* does not perform any recompute unlike S4 in Gpipe (which only avoids recompute for the fifth micro-batch). Language models have final embedding layers that are less compute intensive than the rest of the layers. Avoiding last-stage recompute allows *Varuna* to pack in such embedding layers in the final stage itself, improving efficiency without upsetting the pipeline balance. Finally, note that forward passes are interspersed in *Varuna* throughout the schedule (see stage 3). This allows *Varuna* to make use of opportunistic scheduling to handle jitter, unlike Gpipe where all the forwards are bunched in the beginning.

Pipeline parallelism has been recently introduced into Deepspeed [31] and Megatron [26], with their variations of the Gpipe schedule. As we show in Section 7, *Varuna*'s schedule is more efficient than Gpipe, Deepspeed and Megatron's pipeline schedules in commodity network settings.

4 Handling pre-emptions: Job Morphing

An important goal of *Varuna* is to reduce cost by harnessing *low-priority VMs* that cost 4-5x cheaper than dedicated GPU VMs. As spot VMs can be pre-empted often, *Varuna* needs to adapt to a variable amount of resources. *Varuna* uses a novel technique of *job morphing* to dynamically configure the job to run at best performance with available resources.

Previous approaches for elasticity in DLT jobs mostly address data-parallel jobs [2, 7, 27]. PipeDream [24] performs automated placement, but relies on profiling 1000 mini-batches, and an expensive optimizer whose cost is O($N^2 * L^3$) where N is the number of GPUs and L is the number of layers. Such cost is clearly prohibitive for large models. Even for much smaller models and at most 16 GPUs, their optimizer takes 8s and "a few minutes" for the profiling. *Varuna* scales to models that are 100x larger and utilizes hundreds/thousands of GPUs, thus, a single run of PipeDream's optimizer will take several hours. More recent work [25] uses end-to-end profiling for each configuration of the job; the cost of such profiling can perhaps be tolerated in their setting where the number of GPUs is fixed, but in the context of low-pri VMs that *Varuna* targets, frequent auto-configs need to be performed on preemptions (not just once at job startup).

In contrast, *Varuna* uses a novel approach of *scale-invariant calibration* and *parametrized simulation* to identify the best configuration. Our approach ensures that the profiling state space is kept to a minimum and importantly, is done only once at startup, rather than being repeated every time the number of GPUs change due to pre-emptions.

4.1 Problem setting and constraints

Figure 2 depicts the problem setting and the main constraints for *Varuna*. At any given time, *Varuna* identifies G GPUs that are available as spot instances. It then needs to identify configuration parameters, pipeline depth P and data-parallel replicas D such that $P * D \leq G$. Further, P has to be less than K, the number of cut-points specified by the user. The next constraint is selecting N_m, the number of micro-batches for pipeline efficiency. Then, one needs to select m, the micro-batch size for best performance while respecting the total batch size constraint M_{Total} supplied by the user.

For a per-GPU micro-batch size of m, and G GPUs, the total mini-batch size M_{Total} is $m * N_m * G$. The value of m is constrained at both ends: m cannot be too small, as it reduces efficiency of the CUDA/tensor operations within the GPU (in BERT-large [14], $m = 8$ performs 26% better than $m = 4$); m cannot also be too large, as it then cannot fit into the GPU memory of a single GPU.

4.2 Correctness-preserving morphing

Unlike frameworks that require the user to specify a different set of hyper-parameters for each configuration [2, 7], *Varuna* keeps M_{Total} fixed across configurations, thus easing user burden. The user simply writes the script for a mini-batch size M_{Total} that ensures *maximum parallelism*, and *Varuna* apportions it among dynamically varying resources.

However, this places pressure on the choice of m; if M_{Total} was calibrated for a large number of GPUs, one cannot fit it in fewer GPUs as it would increase m beyond the memory of single GPU. Previous work [27] deals with this problem by setting a conservative small value for M_{total} resulting in poor performance at large scale (because m gets too small). *Varuna* performs automatic gradient accumulation *within* a GPU. When resources shrink resulting in a high value of m that cannot fit in memory, *Varuna* simply increases N_m, the number of micro-batches.

4.3 Scale-invariant Calibration

Morphing in *Varuna* uses a one-time profiling step to calibrate primitive parameters of the hardware and the model. Key to making this scalable is that the primitive parameters are chosen to be (a) mutually orthogonal (so the parameters can be calibrated in parallel, reducing latency); (b) agnostic to the end-to-end configuration of the model (so the number of parameters is small); (c) independent of the total number of GPUs (so that it is scale-invariant when G changes). This makes the calibration task independent of the size of the configuration state-space, in contrast to end-to-end profiling [24, 25] where the state space is much larger.

The parameters that are calibrated in this phase are listed in Table 2. Scale-invariant calibration exploits the mini-batch predictability of DLT job execution [36, 39] to measure these just once. Note that F_i and B_i are independent in the i and m dimension, and are measured in parallel on multiple GPUs by running a few micro-batches using random input values to mock the previous stages. The network times are measured for sending each activation size, both intra- and cross-node. Similarly, AR_i is also measured independently by using a profiling allreduce run that uses the same number of gradients as that of a cut-point for different ring sizes. To model the scenario where multiple stages of the pipeline will be in the same node (and hence all stages could be performing allreduce in parallel), our micro-benchmark measures the allreduce when there are k allreduces in flight (where k is the number of GPUs per node). The time taken for collecting all these measurements is simply the time for a few micro-batches, and is under a minute for even a 10 billion parameter model. Further, it is independent of the total number of GPUs that the job may be scheduled in.

4.4 Parametrized Simulation

Once the primitive parameters are calibrated, they are then fed into an event-driven simulator that models the execution

Parameter	Description
$F_i(m)$	Forward-pass compute-time for C_i
$B_i(m)$	Backward-pass compute-time for C_i
$Act^i_{intra}(m)$	Latency (same node) to send activations of C_i
$Grad^i_{intra}(m)$	Latency (same node) to send gradients of C_i
Act^i_{inter} (m)	Latency (cross-node) to send activations of C_i
$Grad^i_{inter}$ (m)	Latency (cross-node) to send gradients of C_i
$AR_i(D)$	Gradient Allreduce time for C_i on ring size D

Table 2. Primitive parameters for calibration. C_i represents the ith cut-point. m is the micro-batch size. All network times except AR_i include mean latency and jitter.

of *Varuna*. Unlike calibration, the simulator needs to run for each configuration: the simulator takes in values for G, P, and D, and also the mapping of cut-points to stages (in the case of homogeneous-block models such as GPT-2, the mapping is uniform across stages), and simulates one full mini-batch (N_m micro-batches followed by the allreduce), and outputs the estimated time-per-minibatch. As we show in Section 7, the simulated times are within 5% of the actual measured times for that configuration, showing the sufficiency of the parameters for calibration.

To reduce the state space of options to feed to the simulator, *Varuna* first picks the lowest m at which $F_i(m)/m$ stops improving. It then sweeps through all P starting with the smallest P where the model fits, and increase it until the maximum number of cut-points or G. For each P, it picks only one assignment of cutpoints to stages such that they are balanced in $F_i(m)$. Thus, the total exploration size is at O(G). Finally, note that identifying m needs to be done only once as this can be reused in subsequent morphing decisions.

An interesting result from the auto-morphing (validated with real runs) is that contrary to popular belief, a shallow pipeline does not always perform better than a deep pipeline (Observation 2 in § 3). While smaller number of pipeline stages (P) helps with overlapping inter-stage network latency with intra-stage compute, there is a tradeoff; as $G = DxP$, for a given G, a small P results in a larger D; a large D incurs a high cost for performing the data-parallel allreduce communication. Thus, in cases of large G, a deeper pipeline (larger P) keeps D small. Table 3 shows performance from a real run of training a 2.5 billion parameter model with 6-way and 9-way pipelining; the optimal pipeline depth varies with G. The parametrized simulation helps us detect such cases and prefer the best performing configuration for a given G.

4.5 Continuous Checkpointing

To handle unexpected pre-emptions, *Varuna* constantly checkpoints the model state across all stages. Each layer is checkpointed independently. Since data-parallel replicas have the same model state, we shard the checkpointing across replicas for performance. For consistency across the pipeline stages, the checkpointing is done at the end of a mini-batch, every

Num GPUs	Config (PxD)	Total Ex/s	Ex/s/GPU
36	6x6	66.60	1.85
36	9x4	65.88	1.83
36	18x2	50.04	1.39
100	6x16	155.52	1.62
100	9x11	164.34	1.66
100	18x5	99.00	1.1

Table 3. Sensitivity to pipeline depth (P) training a 2.5B GPT2 model. Ex/s is throughput (examples/sec)

few mini-batches. When the configuration changes, *Varuna* can resume the job from the latest checkpoint, with the new configuration; as each layer is checkpointed separately, it allows the morphing framework to even use a different mapping of layers to stages (*e.g.*, if the pipeline depth has to reduce). To reduce impact on training time, the checkpoints are written to local SSD of the VM, and copied to cloud storage in the background.

4.6 *Varuna* Manager

The *Varuna* manager runs on a dedicated VM, and monitors the tasks running on different GPUs. The manager also decides on the placement of the stages and replicas of a job.
Handling fail-stutter machines: When running on preemptible VMs, we repeatedly encountered instances where a particular VM or GPU would perform slower than the rest (often by as much 30%). Because of the synchronous nature of DLT jobs, even a single slow GPU would slow down the entire job. Fortunately, as data parallel replicas of the same stage run the same computation, such *fail-stutter* behavior is easy to correct. Each task sends a *heartbeat* to the manager that contains the GPU compute time per micro-batch for the forward and backward pass. If the manager detects any outliers, it omits that VM when scheduling task replicas.
Tracking size of cluster: As spot VMs can get pre-empted at any time, the manager also detects preemptions when it has not received a heartbeat from a VM, and triggers the morphing functionality to reconfigure the job. Similarly, the manager periodically keeps trying to grow the cluster by invoking the appropriate provisioning APIs in the cloud.

5 Enabling Ease of Programming

A key challenge in model parallelism is making it easy to use for ML developers. Existing approaches like GPipe [20] and DeepSpeed [31] require the user to rewrite their model using specific libraries or in cumbersome ways (*e.g.*, flatten the model into a linear set of layers). Ideally, the framework should not require any changes to the model, and allow the programmer to write the model as if it runs on a single GPU. PipeDream [24] attempts to perform such automatic model splitting, but is tied to specific libraries. PipeDream re-implements a whitelisted set of pytorch operations and make use of `TensorWrappers` to get a sequential list of all computations. This is not scalable to large models like BERT with a high number of custom Python functions, branching and 1000s of lines of code; a standard implementation of BERT crashes on PipeDream. PipeDream-2BW's publicly available code is implemented by heavily customizing the Megatron repository.

It is therefore not surprising that intra-layer parallelism is more popular because the user does not have to worry about the partitioning. For example, while Mesh-Tensorflow [34] is actively maintained and supported by Google [17, 18], the GPipe implementation [3] is quite primitive and works only within a single node. Similarly, Nvidia's Megatron [35] and Microsoft's DeepSpeed [31] all started with intra-layer partitioning and only recently have added pipeline support.

5.1 Auto-partitioning

A key observation behind *Varuna*'s auto-partitioning approach is that massive models such as BERT-large, GPT-2 or ResNet-150 *inherently use repetitive structures*; the same block of layers is repeated multiple times to scale models. For example, in BERT-large or GPT-2, a *transformer* encoder/decoder is the basic building block [37] (referred to as layer in this paper), which is repeated 24 and 48 times, respectively. If one can identify low-activation sizes within each block, they can serve as *cut-points* where the model can be partitioned.

Varuna addresses ease-of-use by automatically partitioning models in two steps: identifying suitable cut-points via model profiling, and then activating a subset of these at run time based on resource availability. Cut-points are "cuts" in the model that slice the computation into equally heavy code sections ending with low activation sizes. These fine-grained sections can be combined at run time for various pipeline depths up to the total number of cut-points.

Cut-points are identified by profiling the model for execution times and activation sizes for each operation. *Varuna* also checks that there is no overlap of undeclared parameters across cut-point boundaries. Based on the desired number of cut-points, *Varuna* uses compute time to shortlist end points for each code section, and picks those with lowest activation size to maintain a high compute-communication ratio. At run time, based on the number of GPUs and bandwidth available, the optimal pipeline depth P is estimated as described earlier and one or more cut-points are grouped together into P partitions.

5.2 Tracking cross-partition dependencies

Another significant usability challenge with inter-layer partitioning is implicit data dependencies that span across partitions, but are important to preserve for model convergence and accuracy. Within the model these might be in the form of shared weights. For example, in the GPT-2 and BERT models, the embedding weights for the first and last layer are "tied", *i.e.*, they are meant to use the same parameters. These are flagged by *Varuna* during automatic cut-point detection and synchronized during training. A more tricky scenario occurs when these dependencies are not in the user-written model code, but hidden in some third-party libraries the model uses. For example, the APEX library [4] for fp16 training performs loss scaling when it detects computation overflow in any layer. In a partitioned world, one stage may hit overflow while others may not, thus requiring an allreduce to synchronize it. Another example arises in optimizers such as NVLamb [6] that use a "global norm" value computed across layers. The model writer may not even be aware of such sharing, so it is easy to miss them while partitioning, resulting in lower accuracy.

To address this problem, *Varuna* provides a tracer that detects such implicit data dependencies. The tracer performs a dry run of training, where model partitions are executed in the same process sequentially. We make minor modifications to the PyTorch library to support a *profiling mode*, during which each created `Tensor` is marked with a cut-point number to which it belongs. In this mode, all python function calls are traced, and any function that uses tensors from more than one partition is flagged. Any tensors that are unmarked during the run are also considered "common" as they are created outside the model/Varuna scope. These are then provided as a list of potential violations to the user, who can mark these as "shared" in *Varuna*. For all shared tensors, *Varuna* performs an allreduce that synchronizes them every mini-batch. In the models we trained for convergence, the tracer caught all instances of such sharing.

6 Implementation

In this section, we describe the implementation of some of the unique aspects of *Varuna*. Our current implementation is built on the PyTorch framework [28].

Cut-points. Based on the details gathered in the dry-run of *Varuna* at initialization time, the model is automatically partitioned according to its rank and total number of nodes in the pipeline. For example, if *Varuna* decides to partition BERT-Large into 4 GPUs, four equally spaced cut-points are activated in the model and the rest of the cut-points become pass through. For rank 0's forward pass, the initial set of modules process the input and when the first activated cut-point is reached, it is configured to send the output activation tensors to the rank 1 process. In rank 1, the first module in the forward pass is set to the cut-point that is awaiting tensors of pre-identified shape from rank 0. It then executes the modules following this until it reaches the next cut-point which is configured to send the activation tensors to rank 2 and so on. The backward pass executes in the reverse manner and sends the gradients tensors to the previous rank. In this way, *Varuna* takes in a user model annotated with cut-points and automatically partitions it into N GPUs to operate in a pipelined manner. If there are multiple data-parallel stages, those pipelines are also setup similarly.

Overlapping computation and communication. The activation and gradient communication is in the critical path of the computation. *Varuna* overlaps this communication with computation so that communication overhead is minimized. Each rank spawns separate threads for sending and receiving activation and gradients. A queue interface is established between the cut-points and the sending/receiving thread. Each thread uses PyTorch's asynchronous sends and receives to transmit/receive tensors from/to the queue and the cut-points and modules independently compute on the tensors whenever they are available.

7 Evaluation

In this section, we first compare the performance and cost savings of *Varuna* against prior systems. Second, we highlight how *Varuna* is uniquely able to navigate the dynamism of spot VM availability while maintaining high training performance. Finally, we show how the performance gains of *Varuna* directly translates to faster time-to-convergence.

Experimental setup. For our experiments, we use two setups. Unless specified otherwise, experiments use a cluster of up to 300 GPUs, using low-priority spot VMs of type NC24_v3 (4-GPU) or NC6_v3 (1-GPU) in Azure. Each 1-GPU VM has Nvidia Volta-100 GPU with 16GB memory, 6 Xeon cores, 112GB of CPU RAM and 10 Gbps ethernet. All VMs are allocated in a single region (South Central US), but have no other locality; in other words, the pair-wise connectivity between the VMs can be routed through multiple levels of bottleneck switches, that in practice limit bandwidth. The second setup we use, that we refer to as "hypercluster", comprises of 16 Nvidia DGX-2 nodes, where each node has 16 V100 GPUs connected via NVLink. The 16 DGX-2 nodes are connected via 200 Gbps Infiniband.

For our workloads, we run the two most popular model architectures: BERT-large [41], and GPT-2 [30]. For BERT-large, we run a model with 340 million parameters, while for GPT-2, we run three configurations: a 2.5 billion parameter model, and a model with 8.3 billion parameters, both from Nvidia [35], and a 20 billion parameter model. To demonstrate scaling, we also show results from a massive model with 200 billion parameters. Note that the 170 billion parameter GPT-3 model is based on the same architecture as GPT-2. Finally, when comparing performance, we use the same mini-batch size for *Varuna* and other systems.

Note that although our evaluations are with language models, *Varuna* does not make any assumptions about the DNN, and will work for all models. The workloads were picked solely due to their large sizes and prevalence in hyperclusters today.

7.1 Performance & Cost

We first evaluate *Varuna* on training performance. The baseline configuration is chosen based on the best reported configuration for each of the models: fully data-parallel training for

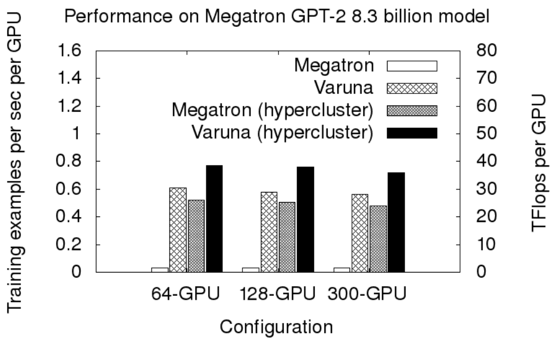


Figure 5. Performance of *Varuna* and Megatron on the GPT-2 8.3 billion parameter model.

BERT-Large, and data-parallel with intra-layer partitioning (Megatron [8, 35]) for GPT-2 models that cannot fit within the 16GB RAM of one GPU. We also compare with other pipeline architectures [20, 24, 26, 31].

We report two metrics of performance for each of these experiments: the number of input examples that were processed per second per-GPU, and the per-GPU teraflops/sec. For the latter, we remove the 33% cost of recompute so that only useful work is captured.

For *Varuna* and other pipeline schemes, we denote the configuration in the format $P \times D$ where P is the number of pipeline stages, and D is the degree of data parallelism per stage. As each model has a minimum value for P (depending on how many layers can fit in a single GPU memory), the total number of GPUs used will be closest multiple of this P; e.g., if $P = 15$, *Varuna* would use only 60 GPUs out of 64.

7.1.1 Comparison to Intra-Layer Partitioning.

We first compare *Varuna* with the intra-layer partitioning scheme of Megatron [8, 35]. We compare four configurations: (a) Megatron on commodity 4-GPU VMs (b) Megatron on hypercluster (c) *Varuna* on commodity 4-GPU VMs, and (d) *Varuna* on hypercluster. We also compare against fully data-parallel configuration for BERT-large model which can fit in a single GPU.

GPT-2 8.3 billion model. Figure 5 compares the performance of *Varuna* with the Megatron baseline, both using a mini-batch size of 8192, under several configurations of the GPT-2 8.3 billion model - on 64 GPUs, 128 GPUs, and 300 GPUs. The corresponding number of GPUs used for *Varuna* were 54 (18x3), 126 (18x7), and 288 (18x16) respectively. As can be seen, on commodity low-pri VMs the training speed (examples per second per GPU) with *Varuna is about 18x better than the Megatron on the same VMs.*

Figure 5 also shows the performance of Megatron and *Varuna* on a hyper-cluster environment. Interestingly, *Varuna* (0.56 ex/s/GPU) on spot VMs performs 17% *better* than Megatron (0.48) on hypercluster, *despite* running on commodity VMs which are 5x cheaper. *The cost-performance is thus 5.85x better for Varuna.* This demonstrates an inherent inefficiency

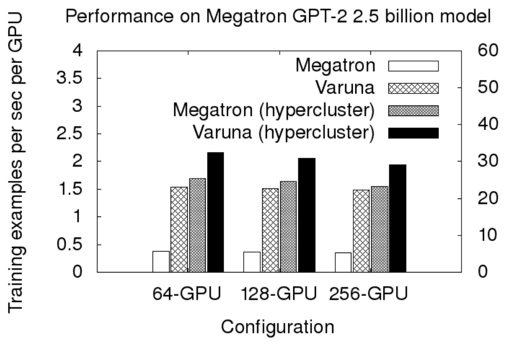


Figure 6. Performance of *Varuna* and Megatron on the GPT-2 2.5 billion parameter model.

System	Num GPUs	Ex/s /GPU	TFlops/s /GPU
20B *Varuna* (LP)	294	0.2	25
19.2B Megatron (HC)	256	0.112	14
20B Megatron (HC)	256	0.015	1.9
20B Varuna (HC)	256	0.257	32.1

Table 4. Comparison between *Varuna* and Megatron for 20B model. LP is low-priority VMs, and HC is hypercluster. All configs use a mini-batch size of 8192.

with intra-layer partitioning even when using a high-speed network like NVLink! The reason why intra-layer partitioning performs worse is because of the large, synchronous allreduces during the forward/backward passes (Observation 1). *Varuna* partitioning on the other hand overlaps GPU compute with communication between stages. The only idle time comes due to pipeline bubbles which are bounded by using a sufficient number of micro-batches and via our schedule. Not surprisingly, *Varuna on hypercluster performs even better (48% faster) compared to Megatron.* Finally, prior work on Megatron (Table 2 in [35]) used a smaller mini-batch size (512) and quotes a time of 2.1 days for 68,500 iterations on a similar cluster of 512 GPUs. This results in only 0.378 examples/s/GPU, lower than the 0.48 we report, as larger batch sizes are more efficient even for intra-layer parallelism.

GPT-2 2.5 billion model. Figure 6 compares *Varuna* with Megatron baseline for a smaller GPT-2 model with 2.5 billion parameters. Again, *Varuna* performs 4.1x better than Megatron on commodity VMs, and gets within 4% of hypercluster performance, for a performance-cost advantage of 4.8x. Further, *Varuna* on hypercluster performs better (25% faster) compared to Megatron. The configs we use for *Varuna* are 9x7 (63 GPUs), 9x14 (126 GPUs) and 9x28 (252 GPUs) respectively.

20 billion and 200 billion parameter models. To illustrate the ability of *Varuna* to scale to much larger models, we show in Table 4, the performance of *Varuna* on a GPT-2 model with 20 billion parameters (96 layers). For *Varuna* on low-pri GPUs, we used a 49x6 configuration to train this

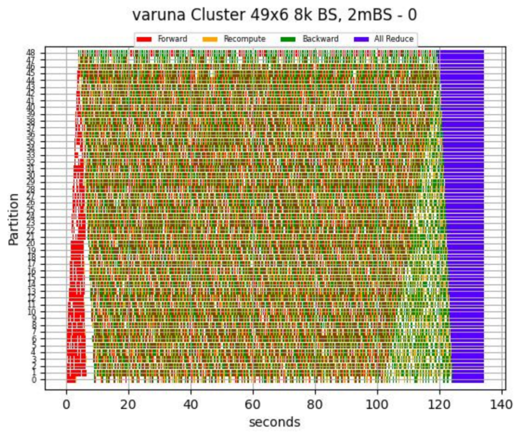


Figure 7. Execution of *Varuna* mini-batch on the GPT-2 20B model (49x6 config).

model on 294 GPUs. For comparison, Megatron on hypercluster could fit only a 19.2 billion parameter model with 16-way model parallelism (so that the intra-layer partitions stay within a single DGX-2 node); but even in this case, *Varuna* on commodity VMs performs 78% faster than Megatron on hypercluster. When we forced Megatron to 20B by 18-way model partitioning, its performance dropped by 10x. Again, *Varuna* on hypercluster shows even better performance than *Varuna* on low-pri VMs.

For more insight into the execution of *Varuna*, we show a detailed timeline of execution of a mini-batch in 49x6 configuration across different stages of the pipeline, in the form of a Gantt Chart [16]. The chart in Figure 7 plots one of the 6 replicas.

To demonstrate extreme scale, we also ran a 200 billion parameter model on *Varuna*, with 100 layers and hidden size of 12960 on 102 GPUs with 102 pipeline stages and no data parallelism. Because of the large layer size, we run this with a micro-batch size of 1 and total batch-size of 512. For this model, *Varuna* keeps the optimizer state in CPU and performs GPU-CPU transfers at the end of mini-batch; the numbers reported include this cost. This ran at 0.022 ex/s/GPU, or 27.3 TFlops/s/GPU.

BERT-large 340 million model. BERT-large is the smallest model we evaluate in this paper. We trained BERT-large using a 4x8 configuration on 32 GPUs on commodity VMs with a sequence length of 512 and batch size of 32K. We observed a throughput of 710 example/s as opposed to 700 ex/s reported by NVIDIA on DGX-1 [1]. Thus, *Varuna* is faster on low-priority VMs compared to DGX-1 with NVlink and infiniband.

Takeaway: Above experiments confirm our *observation 1* that pipeline parallelism is more performant than intra-layer parallelism. Even when intra-layer parallelism is limited to GPUs of a single DGX-2 server, *Varuna* is significantly faster. Thus, intra-layer parallelism should only be used when even a single layer cannot fit in one GPU.

System	Examples/s/GPU	
	Varuna	GPipe
BERT-72 (m=16)	35.9	21.1
BERT-72 (m=32)	41.8	36.2
Simulated 8.3B (normal network)	0.6	0.55
Simulated 8.3B (1.5x slower net)	0.59	0.48
Simulated 8.3B (2x slower net)	0.59	0.426

Table 5. Comparison between *Varuna* and GPipe for a 4-stage pipeline. m refers to micro-batch size. All configs use a mini-batch size of 8192.

System	Examples/s/GPU			
	Varuna	DeepSpeed	Megatron	PipeDream
8.3B (18x4)	0.59	0.47	0.52	OOM
2.5B (9x8)	1.5	1.24	1.31	OOM

Table 6. Comparison of *Varuna*, DeepSpeed, Megatron-1F1B and PipeDream for the 8.3B and 2.5B GPT-2 model on single-GPU VMs. Mini-batch size is 2400.

7.1.2 Comparison with other Pipelining architectures.

In this subsection, we compare *Varuna* with four other systems that perform pipeline partitioning: GPipe [20], DeepSpeed [31], Megatron-1F1B [26] and PipeDream [24]. In DeepSpeed and Megatron-1F1B, we turn off intra-layer partitioning.

Comparison with Gpipe. The GPipe implementation [3] only supports partitioning over single node. Thus, for Gpipe comparison alone, we use a BERT-72 model with 72 layers and a hidden size of 1024 that fits in a single 4-GPU node. We partition the layers for best performance for both systems.

Table 5 shows that *Varuna* is able to deliver 15-70% better performance than Gpipe. Note that GPipe is lot more sensitive to micro-batch size than *Varuna*; when the compute per micro-batch is smaller, the bubble overhead of GPipe dominates, while the pipeline schedule in *Varuna* is able to manage the compute-communication overlap better. To evaluate the impact of network conditions, we used our simulator to simulate GPipe scheduling and estimate its performance in a multi-node setting, with the calibration taken for the 8.3B model (19x3). Under normal network latency, GPipe runs about 9% slower than *Varuna*. However, when we reduced the inter-node bandwidth, the gap between *Varuna* and GPipe widens to 38%.

Comparison with Deepspeed, Megatron, & PipeDream. Table 6 compares *Varuna* with these systems in single-GPU multi-node setting. For this experiment, we disable intra-layer parallelism and other orthogonal optimizations like Zero for a fair comparison of pipeline efficiency of all systems. We run 8.3B GPT-2 and 2.5 GPT-2 models on commodity 1-GPU low-pri VMs. We used a configuration of 18x4 and 9x9 for the two models, respectively. For the 8.3B/2.5B model, each of the stages had 4/6 transformer layers. Note that PipeDream, because of its storing P copies of parameters (for a pipeline depth of P), cannot fit massive models in GPU memory, and hence is reported as OOM in the table. We also ran PipeDream-2BW (GPT2 355M model with the same set of hyper-parameters mentioned in the paper with 6 pipeline stages on 48 GPUs) but found that it did not converge, most likely due to gradient staleness. As shown in the table, *Varuna* performs 20-26% better than DeepSpeed and 13-14% better than Megatron-1F1B with higher gains as the pipeline gets longer.

Takeaway: These results align with our *observation 3* that *Varuna* is able to outperform other pipeline schedules since its design elements specifically cater to network latency/jitter.

7.1.3 Scaling. The ability of *Varuna* to scale across larger clusters can be seen from Figure 5. Going from 56 GPUs to 288 GPUs (5.1x more GPUs), the per-GPU performance of *Varuna* drops only by about 7.5%. Additionally, the performance of *Varuna* in TFlops/s/GPU remains roughly the same going from a 2.5B model to a 200B model. This demonstrates the ability of *Varuna* to scale at the same efficiency to massive models, unlike architectures like Megatron which have performance cliffs.

Takeaway: Interestingly, because *Varuna* does not require dedicated hyperclusters and can make use of opportunistic spot VMs, it can get much larger number of GPUs for a given job, at least for short periods. Coupled with near linear scaling, *Varuna* can train models much faster (*e.g.*, 2x faster on 2x more GPUs), at the *similar dollar cost* (1000 GPUs for 2 days costs the same as 500 GPUs for 4 days in public clouds). Thus, *Varuna* can simultaneously improve both cost and time-to-completion.

7.2 Auto configuration and Job Morphing

All prior systems are designed to run on a fixed number of GPUs. A unique feature of *Varuna* is its ability to adapt to dynamic spot VM availability by quickly identifying and morphing into the best performing configuration for a job.

Simulator: Our auto configuration is driven by *Varuna*'s profile-driven simulator. Table 7 shows the predicted and actual mini-batch times on commodity VMs for several configurations. As can be seen, the simulator estimates are quite accurate, and within 5% error margin compared to the actual values, pointing to the efficacy and sufficiency of the scale-invariant calibration of the simulator.

The time to run the simulator depends on the pipeline depth (P) and the number of micro-batches (N_m). For a 128-GPU job that uses a batch size of 8192, the simulator takes 660ms for P=36, 376ms for P=24 and 391ms for P=18, which is quick enough to react to change in spot VM availability.

Morphing: We now evaluate the robustness and flexibility of *Varuna* to adapt to variable amount of resources in the cluster, as VMs get pre-empted, and come back. For maximizing throughput, we rely on *observation 4*, and request

Model	Config	Minibatch time (s)	
	($P \times D$)	Estimated	Actual
8.3B	36x3	142.8	140.3
8.3B	36x2	198.7	201.3
8.3B	36x1	368.3	390
8.3B	24x4	144.7	149.9
8.3B	24x2	272.4	280.4
8.3B	18x6	139.6	139.1
8.3B	18x4	202.6	203.1
8.3B	18x3	266.6	263.8
2.5B	27x2	115.7	116.8
2.5B	18x3	92.6	96.6
2.5B	9x7	68.9	70.1
2.5B	6x10	77.5	75.2

Table 7. Accuracy of simulator estimates for various models and configurations

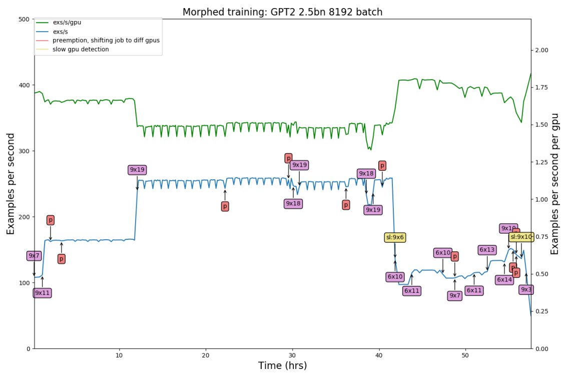


Figure 8. Dynamic timeline of *Varuna* for GPT-2 2.5 billion parameter model (60 hours).

1-GPU VMs as they are more readily available. Even though all cross-GPU communications in 1-GPU VMs go over the network, *Varuna* is able to train on 1-GPU VMs at almost the same performance as 4-GPU VMs. For example, on 72 GPUs, *Varuna* gets a throughput of 1.77 ex/s/gpu on 1-GPU VMs compared to 1.81 ex/s/GPU on 4-GPU VMs.

Figure 8 shows *Varuna*'s training performance in examples/s and examples/s/GPU for the GPT-2 2.5 billion model over 60 hours. Each morphing event is labeled with the configuration dynamically updated by *Varuna*, reacting to change in available VMs. Morphing events where configuration did not change (e.g. due to replacement of a preempted machine) are marked by the letter *p* while the periodic spikes are checkpointing events. One can see that training throughput (examples/s) varies from 50 to 250 (5x) while the per-GPU performance of *Varuna* varies by only 15% percent.

Takeaway: *Varuna* is able to scale effectively to utilize the dynamic spot VM available capacity, while preserving training performance.

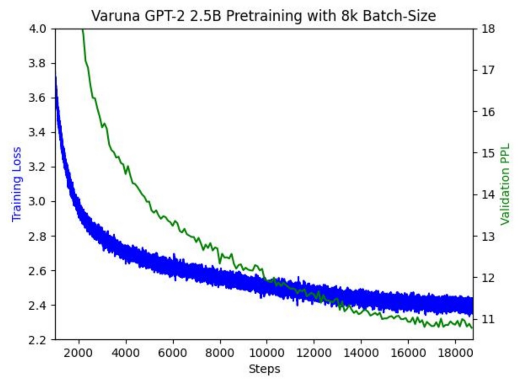


Figure 9. Convergence of GPT-2 2.5 billion model with *Varuna* using 8192 batch size

7.3 Accuracy and Time-to-Convergence

Varuna (and all pipeline schemes in general) require mini-batch size to be about 6x larger for efficiency. It is widely accepted that larger mini-batch training can be done as efficiently as smaller batch-sizes; *e.g.*, for the same number of training examples, BERT-large achieves the same accuracy as mini-batch sizes vary from 512 to 65536 [41].

However, to demonstrate convergence on a much larger model, we trained a 2.5 billion GPT-2 model on *Varuna* using a batch size of 8192. Our baseline is the Megatron-2.5B model that was trained with a batch size of 512 on 300K iterations [35]. We reduce the number of training iterations for *Varuna* by 16x to 18.75K (since we use a 16x larger batch size), thus, ensuring that both *Varuna* and Megatron process the same number of training examples. Figure 9 shows the training curve of the *Varuna* run, showing both the training loss and the validation perplexity. *Varuna* converges to the same validation perplexity of 10.81 reported in Fig. 6 in [35]. Further, we also computed WikiText103 perplexity on the trained model, and obtained a ppl of 12.78, roughly the same as the 12.76 reported in [35]. The Lambada accuracy was 61.25% as against 61.73% reported, which is within noise range of the Lambada accuracy across multiple runs.

Takeaway: Since *Varuna* is 4.1× faster than Megatron in examples processed per second per GPU (Figure 6) and processes the same number of examples as Megatron to achieve the desired accuracy, *Varuna* improves time-to-convergence by 4.1× on commodity VMs for the 2.5B GPT-2 model.

8 Conclusion

By opening up the possibility of training massive deep learning models with billions of parameters, on commodity VMs, *Varuna* reduces the cost of training such models by a significant factor. Further, removing the dependence on specialized hyperclusters allows such training jobs to scale to much larger numbers of GPUs opportunistically. We believe that with this combination of low cost, high performance, and higher scale, *Varuna* can significantly accelerate the pace of innovation in large-scale AI models.

References

[1] BERT pre-training performance. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/README.md#pre-training-nvidia-dgx-1-with-16g.

[2] Dynamic Training with Apache MXNet. https://github.com/awslabs/dynamic-training-with-apache-mxnet-on-aws.

[3] GPipe Source code. https://github.com/tensorflow/lingvo/blob/master/lingvo/core/gpipe.py.

[4] Nvidia apex library. https://github.com/NVIDIA/apex.

[5] NVIDIA DGX-2. https://www.nvidia.com/en-in/data-center/dgx-2/.

[6] NVLAMB Optimizer. https://developer.nvidia.com/blog/pretraining-bert-with-layer-wise-adaptive-learning-rates/.

[7] PyTorch Elastic. https://github.com/pytorch/elastic.

[8] 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/.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, volume 16, pages 265–283. USENIX Association, 2016.

[10] Amazon. Amazon ec2 spot instances. run fault-tolerant workloads for up to 90% off. In *https://aws.amazon.com/ec2/spot/*.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[13] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[16] Henry Laurence Gantt. *Work, wages, and profits*. Engineering Magazine Co., 1913.

[17] Google. Mesh tensorflow - model parallelism made easier. In *https://github.com/tensorflow/mesh*.

[18] Google. Mesh-tensorflow: Model parallelism for supercomputers (tf dev summit '19). In *https://www.youtube.com/watch?v=HgGyWS40g-g*.

[19] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604, 2017.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653*, 2019.

[22] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

[23] Microsoft. Use low-priority azure vms with batch. In *https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms*.

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

[25] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. *arXiv preprint arXiv:2006.09503*, 2020.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*, 2021.

[27] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2:400–411, 2020.

[28] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.

[29] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[30] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

[32] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.

[33] Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. *arXiv preprint arXiv:2004.08900*, 2020.

[34] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

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

[36] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 909–923, New York, NY, USA, 2019. ACM.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[38] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[39] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[40] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *arXiv preprint arXiv:1910.05124*, 2019.

[41] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2019.