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

Sanjith Athlur
Carnegie Mellon University
Pittsburgh, USA
sathlur@cs.cmu.edu

Nitika Saran
Cornell University
Ithaca, USA
ns624@cornell.edu

Muthian Sivathanu
Microsoft Research
Bangalore, India
muthian@microsoft.com

Ramachandran Ramjee
Microsoft Research
Bangalore, India
ramjee@microsoft.com

Nipun Kwatra
Microsoft Research
Bangalore, India
nkwatra@microsoft.com

Abstract

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

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. Varuna improves end-to-end training time for language models like BERT and GPT-2 by up to 18x compared to other model-parallel approaches and up to 26% compared to other pipeline parallel approaches on commodity VMs.

The code for Varuna is available at https://github.com/microsoft/varuna.

CCS Concepts: → Computing methodologies → Distributed computing methodologies; Machine learning; Neural networks.

Keywords: Distributed Systems, Systems for Machine Learning, Large Scale DNN Training

ACM Reference Format:

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 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 suboptimal for other reasons. First, specialized high-speed interconnects are economically infeasible to scale beyond a certain cluster size.
We show that inter-layer or pipeline-partitioning (first introduced in Turing-NLG [8]). To improve pipeline efficiency, huge mini-batch sizes are needed to train massive models to the same accuracy, despite using a large number of GPUs. Second, 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.

Since most massive models today are language models, we limit our evaluation to them. Even though other models in vision/speech might not see gains from pipeline parallelism, Varuna will still automatically identify the best configuration (e.g. data parallel) and perform no worse than the state-of-the-art approaches.

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 intra-layer parallel approaches and up to...
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 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 [41] 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 [41] tries to address the convergence problem with PipeDream by introducing new, approximate techniques to correct for stale updates. While the authors show that PipeMare has 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). Megatron [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 $16 \times N$ 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 sharing 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. It should be noted that the three stages of ZeRO’s sharing, DeepSpeed’s communication-efficient optimizers and sparse-attention are all orthogonal and can be used in conjunction with Varuna.  

**Resource elasticity and Spot VMs.** Recent frameworks such as PyTorch Elastic [7] and MXNet Dynamic [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 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 [39] introduces an adaptive all-reduce method for spot instances, but only works for smaller image models and focuses on finding the best 2-D configuration that has the highest efficiency.  

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.  

### Table 1. Systems for training massive models: Features

<table>
<thead>
<tr>
<th>System</th>
<th>Intra-Layer</th>
<th>Inter-Layer</th>
<th>Sync-SGD</th>
<th>User Ease</th>
<th>Low-Pri.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh-Tensorflow</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Megatron/Turing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>GPipe</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Pipe(Dream/Mare)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Zero/DeepSpeed</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Varuna</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

#### 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 $k-1$, 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 using the stored input activation for each pipeline stage except the last. 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 $2 \times \text{hiddenSize} \times \text{sequenceLength}$ 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 allreduce 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 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
Figure 4. Pipeline efficiency for different micro-batches.

Figure 5. 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)

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 5. 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 are 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-emption: 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 $O(N^2 * L^2)$ 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 pre-emptions (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 $D$ and data-parallel replicas $P$ such that $P \times D \leq G$. 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 $m \times N_m \times G$. 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 a 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, 40] 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.

However, the initial simulation results might not be accurate - either because the initial set of resources used for profiling were faulty and did not accurately represent the hardware parameters, or the network bandwidth available and compute throughput changed over time due to any one of several reasons like contention within VMs in the cloud. In such cases, the parameters must be re-calibrated by running this profiling step again. Varuna Manager can trigger this profiling step by simply observing if mini-batch training times deviate over time. We also allow the user to trigger this profiling step while resuming the training of a model from a checkpoint.

4.4 Parametrized Simulation

Once the primitive parameters are calibrated, they are then fed into an event-driven simulator that models the execution 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.
Table 2. Primitive parameters for calibration. $C_i$ represents the $i$th cut-point. $m$ is the micro-batch size. All network times except $AR_i$ include mean latency and jitter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_i(m)$</td>
<td>Forward-pass compute-time for $C_i$</td>
</tr>
<tr>
<td>$B_i(m)$</td>
<td>Backward-pass compute-time for $C_i$</td>
</tr>
<tr>
<td>$Act_i^{\text{intra}}(m)$</td>
<td>Latency (same node) to send activations of $C_i$</td>
</tr>
<tr>
<td>$Grad_i^{\text{intra}}(m)$</td>
<td>Latency (same node) to send gradients of $C_i$</td>
</tr>
<tr>
<td>$Act_i^{\text{inter}}(m)$</td>
<td>Latency (cross-node) to send activations of $C_i$</td>
</tr>
<tr>
<td>$Grad_i^{\text{inter}}(m)$</td>
<td>Latency (cross-node) to send gradients of $C_i$</td>
</tr>
<tr>
<td>$AR_i(D)$</td>
<td>Gradient Allreduce time for $C_i$ on ring size $D$</td>
</tr>
</tbody>
</table>

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

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

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 pre-emptible 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 DL 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 pre-emptions 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 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.
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 [38] (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.

Varuna takes the number of cut-points as input and identifies these cut-points by profiling the model for execution times and activation sizes for each operation. Varuna also checks that there is no overlap of parameters across cutpoint boundaries, and parameters that are reused across boundaries are marked as shared parameters (e.g., embedding weights in transformer models). Based on the desired number of cut-points, Varuna uses compute time to shortlist end points for each code section, and picks those with the lowest activation size to maintain a high compute to 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.

We modified three popular models BERT-large, GPT-2, and EfficientNet [37] (each written by a different organization or developer). In all three models, the change to introduce cut-points were less than 10 lines of code out of over thousand lines of model definition code in each model.

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, annotates it 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.

**Synchronization of shared state.** Varuna establishes two process groups (or all-reduce rings) for each process. One process group consists of the data-parallel replicas of the same pipeline stage, over which an all-reduce is performed to get the average gradient. A second process group is established among all the processes of each pipeline. This is needed for synchronizing on data shared across partitions (e.g., flagged by the automatic debugger - § 5)

There are various other functions as well such as saving and restoring GPU random number state to ensure recomputation correctness, generating and utilizing the pipeline schedule, periodic checkpointing, resuming from checkpoints upon preemption, communicating forward/backward times for fail-stutter fault-tolerance, looking for new VMs to grow the cluster, etc. In total, Varuna consists of about 2400 lines of Python code and 550 lines of C++ code.

## 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’, consists 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 [42], 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 hyper-clusters 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 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 the 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. All performance numbers reported are over an average of 5 mini-batch runs, after running several warm-up mini-batches.
Figure 6. Performance of Varuna and Megatron on the GPT-2 8.3 billion parameter model.

GPT-2 8.3 billion model. Figure 6 compares the performance of Varuna with the 16-way model parallel 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 on commodity VMs (of type NC24_v3). 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 6 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 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.

In absolute wall clock times, assuming that we have a non-preemptible fixed set of 512 GPUs, Megatron (Table 2 in [35]) takes 9.94 days on hypercluster-like environment with batch size of 512, while Varuna takes 9.22 days on commodity VMs with a batch size of 8192 to train 2.5B GPT-2 model to convergence.

GPT-2 2.5 billion model. Figure 7 compares Varuna with 8-way model parallel Megatron baseline for a smaller GPT-2 model with 2.5 billion parameters on commodity VMs (of type NC24_v3). 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. In absolute wall clock times, assuming a non-preemptible fixed set of 128 GPUs, Megatron (Table 2 in [35]) takes 9.94 days on hypercluster-like environment with batch size of 512, while Varuna takes 9.22 days on commodity VMs with a batch size of 8192 to train 2.5B GPT-2 model to convergence.

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) on commodity VMs (of type NC24_v3) and the hypercluster. For Varuna on low-pri GPUs,
we used a 49x6 configuration to train this 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 8 plots one of the 6 replicas. Each horizontal line in the graph corresponds to a stage of the pipeline. The purple region at the far right indicates the time for stage-wise 6-way allreduce. Red bars correspond to forward pass, green bars pertain to backward pass, and orange bars indicate recompute.

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 for the settings we investigate.

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.

### 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.

<table>
<thead>
<tr>
<th>System</th>
<th>Num GPUs</th>
<th>Ex/s (ex/s)</th>
<th>TFlops/s (TFlops/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20B Varuna (LP)</td>
<td>294</td>
<td>0.2</td>
<td>25</td>
</tr>
<tr>
<td>19.2B Megatron (HC)</td>
<td>256</td>
<td>0.112</td>
<td>14</td>
</tr>
<tr>
<td>20B Megatron (HC)</td>
<td>256</td>
<td>0.015</td>
<td>1.9</td>
</tr>
<tr>
<td>20B Varuna (HC)</td>
<td>256</td>
<td>0.257</td>
<td>32.1</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>System</th>
<th>Examples/s/GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT-72 (m=16)</td>
<td>35.9</td>
</tr>
<tr>
<td>BERT-72 (m=32)</td>
<td>41.8</td>
</tr>
<tr>
<td>Simulated 8.3B (normal network)</td>
<td>0.6</td>
</tr>
<tr>
<td>Simulated 8.3B (1.5x slower net)</td>
<td>0.59</td>
</tr>
<tr>
<td>Simulated 8.3B (2x slower net)</td>
<td>0.59</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>System</th>
<th>Varuna</th>
<th>DeepSpeed</th>
<th>Megatron</th>
<th>PipeDream</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3B (18x4)</td>
<td>0.59</td>
<td>0.47</td>
<td>0.52</td>
<td>OOM</td>
</tr>
<tr>
<td>2.5B (9x8)</td>
<td>1.5</td>
<td>1.24</td>
<td>1.31</td>
<td>OOM</td>
</tr>
</tbody>
</table>
Figure 9. PipeDream-2BW Loss Curve for 355M-parameter GPT2 model

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. Although the authors show convergence results for Bert, asynchronous training is not a common practice today, primarily since it is unclear how final accuracy will be affected due to stale backward updates. For 355M GPT2 model, the authors report a WikiText PPL of 19.56 as opposed to 19.28 reported by Nvidia [35] and 19.16 PPL with Varuna using 8k batch size (lower is better). In training the 355M GPT2 model to compute Lambada accuracy, the training diverged after 16k iterations and the loss shot up as shown in Figure 9.

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 6. Going from 56 GPUs to 288 GPUs (5.1x more GPUs), the per-GPU performance of Varuna drops only by about 7.5%. One can also see that the performance of Varuna in TFllops/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 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 with 1-GPU VMs compared to 1.81 ex/s/GPU on 4-GPU VMs.

Figure 10 shows Varuna’s training performance in examples/s and examples/s/GPU for the GPT-2 2.5 billion model.
Figure 10. Dynamic timeline of **Varuna** for GPT-2 2.5 billion parameter model (60 hours).

Figure 11. Convergence of GPT-2 2.5 billion model with **Varuna** using 8192 batch size 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.

7.3 Accuracy and Time-to-Convergence

As discussed in Section 3.1, **Varuna** (and all pipeline schemes in general) require mini-batch size to be about 6x larger for efficiency. Prior works have shown that larger mini-batch training for language models 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 [42].

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 11 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.

We also trained a 355M GPT-2 model using **Varuna** with a batch-size of 8192 for 18.75k iterations and compared it to Megatron-355M trained with a batch-size of 512 trained for 300k iterations. **Varuna** achieves a ppl of 19.16, surpassing the ppl of 19.28 reported in [35] (lower is better).

We do note that there is an ongoing debate about applicability of large-batch sizes without compromising accuracy in deep-learning models. However, state-of-the-art language-models like BERT and GPT, which constitute the majority of massive models today, work well with large batch-sizes as confirmed by both our experiments and earlier works.

**Takeaway:** Since **Varuna** is 4.1× faster than intra-layer parallel Megatron in examples processed per second per GPU (Figure 7) 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.

9 Acknowledgements

We thank our shepherd Panos Kalnis and the anonymous reviewers for their valuable comments and suggestions. We also thank Saurabh Tiwary and Xia Song of the Microsoft Turing team for their support and feedback.
References


A Artifact Appendix

A.1 Abstract

Varuna is a PyTorch library to efficiently train large deep learning models. It implements a combination of pipeline and data parallelism. It can be used with a large set of GPUs across machines, such as VMs in a public cloud. Varuna also enables adaptive training on a changing set of underlying resources.

A.2 Description & Requirements

A.2.1 How to access. The code for Varuna is publicly available at https://github.com/microsoft/varuna. The repository has instructions to install and set up the code, and includes extensive documentation. Examples for language models like BERT[14] and Megatron-LM[35] are included in the repository, as well as a minimal working example using EfficientNet [37].

ACM DOI for Varuna: https://doi.org/10.5281/zenodo.6359660

A.2.2 Hardware dependencies. Varuna is designed to train models on GPUs. All experiments in the paper were run across multiple virtual machines, where each virtual machine had 1-4 V100 GPUs. Although usability of Varuna can be demonstrated on P40 or older generation GPUs as well, reproducing results in the paper will require access to V100 GPUs or newer (for mixed-precision training).

For our experiments, we used up to 300 GPUs on low-priority spot VMs of type NC24_v3 (4-GPU) or NC6_v3 (1-GPU) in Azure. GPUs were Nvidia Volta-100 GPU with 16GB memory, 6 Xeon cores, 112GB of CPU RAM and 10 Gbps ethernet.

Another setup we use in our experiments to compare with low-priority training, is referred to as “hypercluster”, and consists 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. More details are in section 7.

A.2.3 Software dependencies. Varuna’s code is written in python. Required libraries are PyTorch, CUDA and NVIDIA’s apex. Instructions to set these up are included in the GitHub repository.

It would be convenient to use one of NVIDIA’s docker images for PyTorch for a quick setup of prerequisites.

A.2.4 Benchmarks. Most of Varuna’s results are for pre-training two large language models - GPT-2 and BERT-Large. The third-party implementations used for these are publicly available, with links and instructions in the “examples” folder of Varuna’s GitHub directory. We use the same dataset as described in Megatron-LM [35]. For BERT, the Wikipedia and BookCorpus datasets were used. Instructions to set these up are detailed in the linked BERT repository.

Convergence results (example, Figure 11) depend on these datasets. However, for performance evaluations (example, Table 6) don’t need complete datasets since only a few training iterations are required to measure throughput.

A.3 Set-up

Varuna requires python, PyTorch, CUDA, and apex. Set up instructions for the code are in the repository’s README.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Varuna’s training throughput in examples/s/GPU on low-priority commodity VMs is up to 17x faster than Megatron’s intra-layer partitioning on hyperclusters. This is shown in experiment E1.
- (C2): Varuna is comparable to or faster than other recent pipelining architectures (Table 6). This is shown in experiment E2.

A.4.2 Experiments.

- Experiment (E1): Comparison with Intra-layer Partitioning [30 human-minutes + 64 GPU-hours (assuming 64 GPUs)]: Running Varuna with an 8.5 billion parameter model to get examples/s/GPU as seen in Figure 6 in the paper.
  - Preparation: Prepare a variable number of NC24_v3 VMs (however many available), and set up the Megatron-LM example included in the GitHub repository.
  - Execution: Run the 8.3 billion parameter model for Megatron-LM with a batch size of 8192. Measure time per iteration, let this be t. Throughput in exs/s/gpus will be (8192/t)/gpus used.
  - Results: Compare exs/s/gpu to Figure 6 in the paper.

- Experiment (E2): Comparison with other pipelining architectures [1 human-hour + 192 GPU-hours (assuming 64 GPUs)]:
  - Preparation: Prepare up to 72 NC6_v3 VMs (if available), and set up the Megatron-LM example included in the GitHub repository.
  - Execution: Run the 8.3 billion and the 2.5 billion parameter model for Megatron-LM with a batch size of 8192. Measure time per iteration, let this be t. Throughput in example/s/gpu will be (8192/t)/gpus used.
  - Results: Compare ex/s/gpu to Table 6 in the paper.