Artifacts Available
acm
V1.1

Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers

Youjie Li*
UIUC
li238@illinois.edu

Amar Phanishayee
Microsoft Research
amar@microsoft.com

Derek Murray†
Lacework
derek.murray@lacework.net

Jakub Tarnawski
Microsoft Research
jakub.tarnawski@microsoft.com

Nam Sung Kim
UIUC
nam.sung.kim@gmail.com

ABSTRACT

Deep neural networks (DNNs) have grown exponentially in size over the past decade, leaving only those who have massive datacenter-based resources with the ability to develop and train such models. One of the main challenges for the long tail of researchers who might have only limited resources (e.g., a single multi-GPU server) is limited GPU memory capacity compared to model size. The problem is so acute that the memory requirement of training massive DNN models can often exceed the aggregate capacity of all available GPUs on a single server; this problem only gets worse with the trend of ever-growing model sizes. Current solutions that rely on virtualizing GPU memory (by swapping to/from CPU memory) incur excessive swapping overhead. In this paper, we present a new training framework, Harmony, and advocate rethinking how DNN frameworks schedule computation and move data to push the boundaries of training massive models efficiently on a single commodity server. Across various massive DNN models, Harmony is able to reduce swap load by up to two orders of magnitude and obtain a training throughput speedup of up to 7.6× over highly optimized baselines with virtualized memory.

PVLDB Reference Format:
Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers. PVLDB, 15(11): 2747 - 2760, 2022.
doi:10.14778/3551793.3551828

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/msr-fiddle/harmony.

1 INTRODUCTION

Modern DNNs have transformed our approach of solving a range of problems such as image classification [32], semantic segmentation [64], translation [67], and language modeling [57]. Over the years, these models have grown exponentially in size while continuing to achieve unprecedented accuracy on ever more complex tasks [1, 30, 44]. For example, a 557-million-parameter AmoebaNet can achieve super-human accuracy in image classification [20]. Similarly, a state-of-the-art language model like the 175-billion parameter GPT-3 [4] can generate human-like text [15, 41, 62]. Training these models to accuracy takes weeks to months of wall-clock time, despite running in parallel on large clusters of fast accelerators.

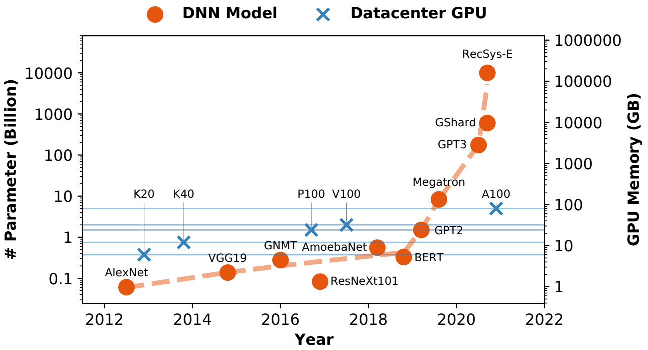


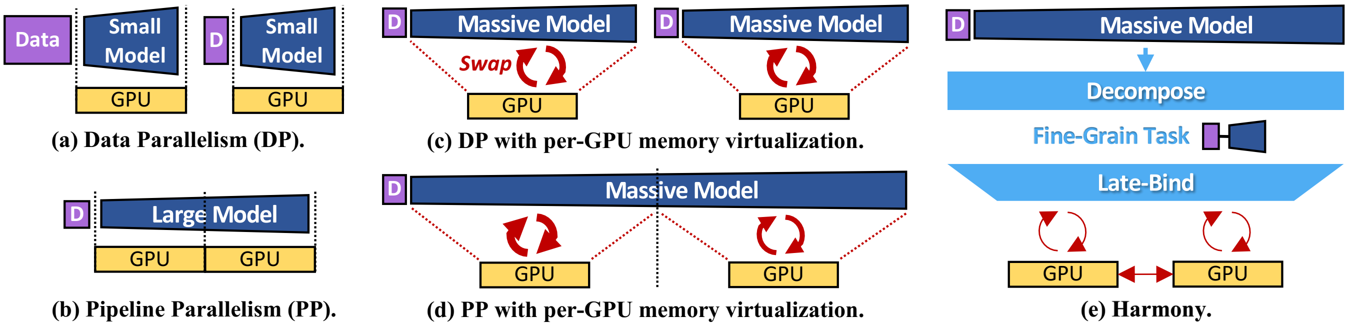
Figure 1: Growth of DNN model size and GPU memory capacity over the past decade [12, 53]. Memory consumed here only accounts for model state which is a small fraction of total training memory footprint [5, 12, 26, 58, 61, 66].

These resource demands leave only those who have massive datacenter-based resources (e.g., Google, Microsoft, NVIDIA, etc.) with the ability to train such models. The long tail of researchers who have only limited resources (e.g., a single server with multiple GPUs) increasingly risk being alienated from innovating in this space. While training on larger clusters naturally results in speedier training, in this paper we investigate how to push the boundaries of training massive models on *a single commodity server* – a setting invaluable for developing, debugging, and fine-tuning DNNs [9].

Challenges. One of the main challenges in training massive models is that the required memory footprint far exceeds the memory capacity of accelerators. Figure 1 shows how sizes of image classification and language models have grown dramatically over time. Furthermore, model parameters are only a small part of the memory footprint of training; gradients, stashed activations, optimizer states, and framework workspace all taken together significantly blow up the memory footprint [5, 26, 58, 61, 66].

[*] Work done as a Project Fiddle intern at MSR.
[†] Work done when the author was at Microsoft.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551828



(a) Data Parallelism (DP).

(b) Pipeline Parallelism (PP).

(c) DP with per-GPU memory virtualization.

(d) PP with per-GPU memory virtualization.

(e) Harmony.

Figure 2: Illustrative comparison between different approaches for massive model training. (Only one data batch is shown.)

This memory footprint problem motivates recent innovations that alleviate memory pressure. For example, recent advances in GPU memory virtualization push the boundaries of what can be achieved on a single GPU [6, 19, 55, 61], but as we show in § 2 such techniques are inefficient when applied to parallel multi-GPU training regardless of data parallelism [10, 35] or pipeline parallelism [20, 45], as illustrated in Figure 2(a–d). Other techniques, such as encoding data structures [26], recomputing intermediate tensors [5], sharding optimizer [58], offloading optimizer to CPU [60], and splitting a large layer into small ones [63], all aim to reduce memory pressure during training. However, despite these optimizations, the general problem of ***efficiently training massive models on a single server with a handful of commodity GPUs while exhausting the collective memory capacity of all available GPUs and CPU DRAM*** is still an open problem.

We argue that current DNN frameworks have two fundamental problems that limit massive model training on modest deployments. First, they ***schedule work at a coarse granularity***, treating the training program as a black box: computing an entire model or an entire stage of layers for each input batch. This coarse granularity limits flexibility of scheduling tasks to available resources, thus thwarting memory-reuse–based performance enhancements that can reduce virtual memory swap overhead. For example, executing a group of DNN layers (even with intra-layer partitions), one input batch at a time, limits reuse of weights loaded into memory across different input batches, as they might get swapped out. Second, frameworks ***eagerly bind work to accelerators***, pushing this decision all the way to the programmer's training script in most cases. For example, in PyTorch [37], the state of a layer group is bound to a user-script–defined device, and thus the forward and backward computation on that state is implicitly bound to the same device. Virtualizing the memory of a single GPU helps here, by treating the nearby host RAM as a swap target, but it makes inefficient use of other available GPUs and the interconnects between them.

Contributions. Ideally, users could write DNN training programs that target a single virtual accelerator with practically unbounded memory. Our proposed system, Harmony, targets this ideal. As illustrated in Figure 2(e), Harmony decomposes a model's operations in a training script into ***fine-grained tasks*** [1] and introduces a novel task scheduler that efficiently maps computation and state to physical devices (***late binding***); the tasks in the task graph can run on different physical devices in a data- or pipeline-parallel fashion and Harmony transparently moves state and data across tasks. Unlike prior pipeline-parallel training [20, 45, 46], each GPU in Harmony no longer hosts a fixed stage of layers, thus resulting in a novel pipelining scheme, ***Wrap-Around Pipeline***, while offering synchronous SGD semantics.

Harmony has to overcome two main challenges to operate at peak throughput: (i) *minimizing expensive CPU-GPU memory swaps*, and (ii) *balancing load across all GPUs* so that there is no bottleneck worker in the execution pipeline. Harmony achieves this by using four distinct optimizations for efficient training:

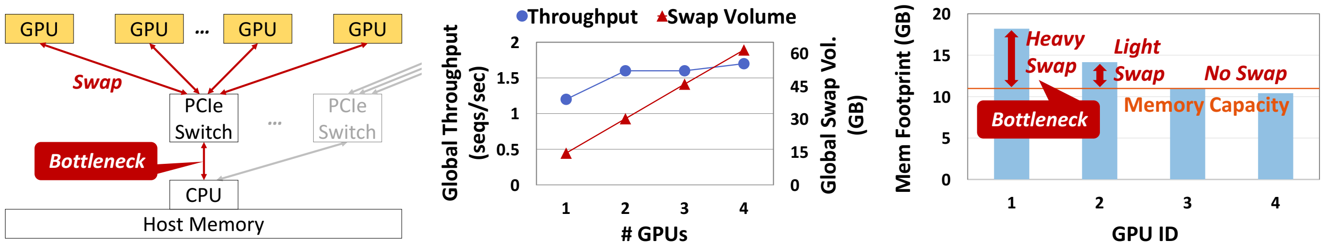
❶ ***Reusing State in GPU Memory across Different Inputs***. Empowered by the flexibility of scheduling at a finer granularity, we propose a new technique called *input-batch grouping*, where a scheduled layer(s) can run across a group of input batches before scheduling the next layer(s) on the same GPU, thus improving state reuse in GPU memory and consequently improving arithmetic intensity.

❷ ***Scheduling Tasks Just-in-time***. Harmony schedules tasks as soon as all input dependencies are available, thus avoiding the risk of swapping out those dependencies; this especially helps tasks such as weight update, which in frameworks such as PyTorch are normally scheduled to execute only after the backward pass for the entire model, resulting in avoidable CPU-GPU swaps.

❸ ***Generalized Tensor Swaps over Fast Peer-to-peer Links***. With *late binding* of tasks to GPUs, Harmony places adjacent tasks across GPUs and swaps tensors directly between GPUs using *peer-to-peer (p2p) swaps* rather than swapping state back and forth to CPU memory. Unlike prior work, p2p swaps in Harmony are not limited to only the output tensors of stages [11, 20, 45] but can be used to transfer or swap any intermediate tensor within each stage.

❹ ***Multi-dimensional Layer Packing***. Tensor swaps can be minimized by packing contiguous layers together. Greedily picking the largest pack size that fits a GPU, however, results in globally sub-optimal pipelines due to imbalance across GPUs. Furthermore, picking layer packs is challenging because not all layers are created equal. The same layer has drastically different compute and memory requirements between forward and backward passes for a fixed batch size; the differences are only accentuated when we consider different batch sizes. We thus have to find packs in the multi-dimensional space (forward batch size, forward packs, backward batch size, backward packs) that balance compute, memory,

[1] A task consists of an input microbatch and a contiguous set of layers; there is no requirement of one-one correspondence between forward and backward tasks.



(a) Intra-server interconnects. (b) DP with per-GPU memory virtualization. (c) PP with per-GPU memory virtualization.

Figure 3: The CPU-GPU swap bottleneck in Data Parallelism (DP) [36] and Pipeline Parallelism (PP) [46] when using GPU memory virtualization. For example, training BERT [8] on a server with four GTX1080Tis (11GB) and a batch size of 5 results in memory footprint exceeding GPU memory capacity, requiring IBM-LMS [24] for virtualizing individual GPU memory. (a) and (b) show that DP's swap volume increases linearly with the number of GPUs, exposing the bottleneck PCIe link and thus throttling training throughput. (c) shows that PP's swap volume is unbalanced across GPUs, resulting in pipeline bottleneck.

and swaps across GPUs; however, we find this problem of *optimally determining layer packs in Harmony to be NP-hard*. We propose an efficient heuristic algorithm that searches through this multi-dimensional space to find effective parallel training schedules without pipeline bottlenecks. ***To our best knowledge, no prior work has attempted such multi-dimensional layer packing.***

In this paper, we show how *task decomposition* and *late binding*, together with a set of novel performance optimizations mentioned above, enable virtualized parallel training of massive DNNs that exhaust collective memory capacity of all available accelerators in modest single-server deployments[2]. A short workshop version of this paper highlighted the limitations of existing DNN frameworks in training massive models [38]. Unlike our prior work, in the current paper we provide foundational principles, a detailed design backed by a concrete implementation, and extensive evaluations across various massive models. We show that Harmony is able to ***reduce swap load by up to two orders of magnitude*** and obtain a training throughput ***speedup of up to 7.6***× over highly optimized baselines with virtualized memory, including recent systems such as ZeRO-Infinity [59], while offering ***synchronous SGD semantics***.

Roadmap. In the rest of this paper, we first present the limitations of related works with a focus on GPU memory virtualization (§ 2), then offer a high-level overview of Harmony (§ 3), followed by low-level designs and implementations (§ 4). We experimentally validate Harmony's efficacy (§ 5) before concluding (§ 6-7).

2 BACKGROUND AND RELATED WORKS

Parallel Training. Data Parallelism (DP) [35, 37, 40], the predominant mode of parallel DNN training, requires the entire model's memory footprint to fit on each GPU, making it unfit for massive model training (Figures 1 and 2(a)). Pipeline Parallelism (PP) [11, 20, 45, 46] and Model Parallelism (MP) [63] have become mainstream for training large models by partitioning a model so that each part fits on an individual GPU (Figure 2(b)). However, even in the face of partitioned models, all these systems *require training memory footprint to be less than the collective memory capacity of all GPUs.*

Memory Optimizations. To reduce the memory footprint, modern frameworks incorporate various memory optimizations by default, such as the *recompute* that re-materializes intermediate tensors when needed [5, 27, 66]. CPU-offloading is also used for offloading model/optimizer states from GPU to CPU [13, 21, 59, 60].

GPU Memory Virtualization. Despite various memory optimizations, GPU memory virtualization [51] remains inexorable due to the exponential growth in model sizes. Recent work has applied this idea to train large DNNs by backing GPU memory with CPU memory and swapping tensors between CPU and GPU [6, 19, 55, 61]. However, such techniques are limited to only an ***individual GPU*** considered in isolation. Here we show that per-GPU memory virtualization is inefficient as it causes either a high swap overhead when used in DP or swap imbalance in PP (Figure 2(c–d)).

Today's frameworks have four key inefficiencies that cause these swap-overhead related performance problems in parallel training:

❶ ***Repeated Swaps.*** A layer can consume different input data batches or intermediate tensors at different times, but it always requires the same weight or gradient buffer. With GPU memory virtualization, these common weight and gradient are swapped in and out repeatedly across batches of input data.

❷ ***Unnecessary Swaps.*** Certain operators in DNN frameworks today are scheduled at rigid points in the timeline of a training iteration, even though all their inputs are available much earlier. When training massive models with GPU virtualization, this rigidity is inefficient: the GPU-resident inputs and state for such operators can be swapped out of GPU memory, only to be swapped back in again when the operator is actually scheduled. For example, in PyTorch, the weight update for each layer only starts after the backward pass of the entire model, potentially causing unnecessary swaps of most layer weights and gradients.

❸ ***Only CPU-GPU Swaps.*** GPU memory virtualization lacks context about parallel training, works in isolation to other GPUs, and can only swap to host memory. This exposes the bottleneck device-to-host interconnect (Figure 3(a)) and misses the opportunity to use fast device-to-device links for cross-device swaps. Figure 3(b) shows that in DP, the swap overhead across multiple GPUs throttles throughput, as the global swap load exposes

[2] We omit storage from the memory hierarchy; if incorporated, our work can target even larger models that exceed CPU DRAM capacity.

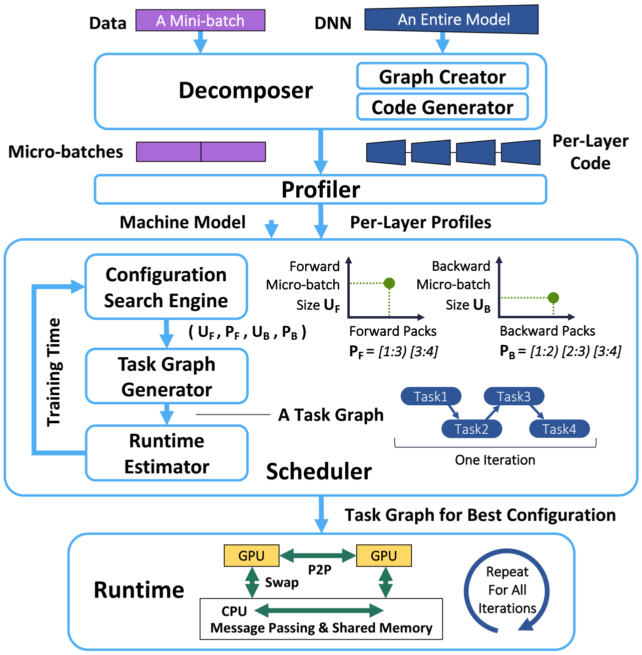

Data
A Mini-batch
DNN
An Entire Model
Decomposer
Graph Creator
Code Generator
Micro-batches
Per-Layer Code
Profiler
Machine Model
Per-Layer Profiles
Configuration Search Engine
Forward Micro-batch Size U_F
Backward Micro-batch Size U_B
(U_F , P_F , U_B , P_B)
Forward Packs P_F = [1:3] [3:4]
Backward Packs P_B = [1:2] [2:3] [3:4]
Training Time
Task Graph Generator
A Task Graph
Task1
Task2
Task3
Task4
One Iteration
Runtime Estimator
Scheduler
Task Graph for Best Configuration
Runtime
GPU
GPU
P2P
Swap
CPU
Message Passing & Shared Memory
Repeat For All Iterations

Figure 4: High-level overview of Harmony.

the bottleneck link: CPU and shared PCIe links with 1:4~8 oversubscription [2, 34, 50, 52, 56]. As each GPU is swapping a similar amount of state, the swap overhead grows linearly in the number of GPUs. Furthermore, PP may use p2p communication but only for per-stage output tensors (a small fraction of all tensors); it leaves all intermediate tensors within each stage, thus swapping them to the CPU when combined with per-GPU memory virtualization.

❹ ***Unbalanced Swaps.*** In PP, pipeline stages are designed to be compute-load balanced, but such pipelining inherently has imbalanced memory sizes across stages: the head of the pipeline must stash more activations compared to the tail [45, 46]. Lacking this context and operating in isolation on individual GPUs, naively virtualizing GPU memory can result in swap imbalance across stages, exposing the bottleneck stage with the heaviest swap (Figure 3(c)).

3 TRAINING IN HARMONY

Figure 4 shows a high-level overview of Harmony. First, users provide Harmony with training data and their model (written in imperative-style PyTorch [54], as if running sequentially on one device). Harmony's *Decomposer* breaks down the entire model by extracting its layer-granularity graph (via the *Graph Creator*), and then generating per-layer code based on the graph such that they can be executed individually if needed (via the *Code Generator*). The data minibatch is also decomposed into small microbatches.

Next, Harmony's *Profiler* executes the layer-granularity graph, one layer at a time, by running the per-layer code on a single GPU of the type that will be used in the deployment (seamlessly swapping tensors between CPU and GPU as required); it does this both for the forward pass and also later for the backward pass when the graph is traversed in the reverse direction. The profiler repeats this process across different microbatch sizes. This generates profiles containing computation times, memory footprint, and input tensor sizes for each layer under different settings.

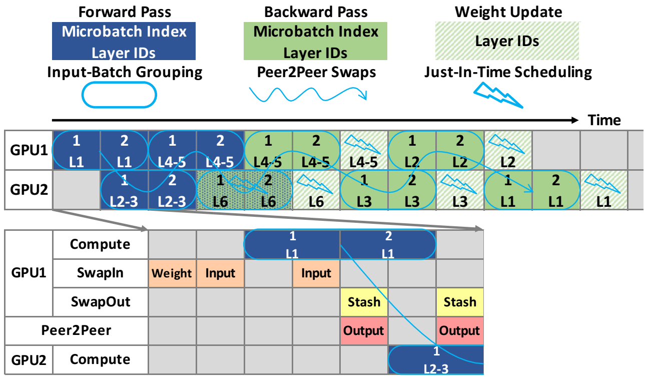


Figure 5: Example of training a toy six-layer "massive" model on two GPUs with Wrap-Around Pipeline in Harmony.

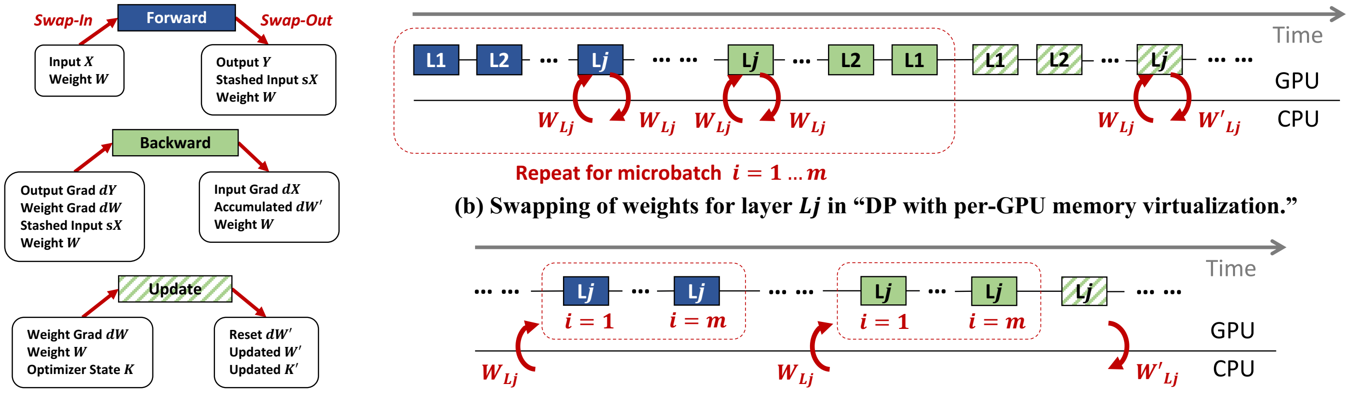
Then, Harmony's *Scheduler* takes the generated profiles along with the machine model (e.g., GPU memory capacity, number of GPUs, and interconnects) to compile a schedule of a *single* training iteration. It does this by: 1) selecting which layers should be executed together as a pack and thus picking a training configuration (a four-tuple of <***forward microbatch size*** U_F, ***forward layer packs*** P_F, ***backward microbatch size*** U_B, ***backward layer packs*** P_B>), 2) building a task graph for this configuration (*Task Graph Generator*), 3) estimating its training time (*Runtime Estimator*), and 4) refining the configuration by searching through the space of configuration options (*Configuration Search Engine*).

Finally, once the best configuration is found and the final task graph is generated, the Harmony *Runtime* then executes it for all training iterations on the set of GPUs in the deployment.

Modes of Parallel Execution. Harmony supports two modes of execution, data parallelism (*Harmony DP*) and pipeline parallelism (*Harmony PP* with *Wrap-Around Pipeline*), while offering users the illusion of running on a single virtual device with practically unbounded memory. With a user-specified parallelism mode, Harmony's Scheduler binds tasks to devices, appropriately scheduling the movement of required inputs (activations, weights, etc.) from CPU to GPU memory or directly between GPU memories.

Key Optimizations. Operating at peak throughput requires Harmony to overcome two main challenges: (i) minimizing expensive CPU-GPU memory swaps, and (ii) balancing load across all GPUs so that there is no bottleneck worker in the execution pipeline. Harmony achieves this by using four distinct optimizations:

❶ ***Input-batch grouping*** allows a scheduled layer pack to execute across different input batches back-to-back; the state of layer(s) (e.g., the weight or gradient buffer) can stay in memory and be reused across multiple input data batches or input tensors. Grouping M inputs for a layer pack (each input-batch saturates GPU memory) reduces what would otherwise have been M repeated swaps of the state for each batch to a single swap. Figure 5 shows an example of training with Harmony PP, where each layer pack executes on a group of two microbatches back-to-back before moving to the next layer pack. Unlike traditional pipeline stages [20, 45] which execute all layers in the stage one batch at a time, resulting in repeated



(a) Swap model.

(b) Swapping of weights for layer *Lj* in "DP with per-GPU memory virtualization."

(c) Swapping of weights for layer *Lj* in "Harmony DP."

Figure 6: Tensors that need to be swapped in and out for forward, backward, and weight update phases of training.

swaps when used with GPU memory virtualization, in Harmony the forward pass of a layer (e.g., $L1$) runs through 2 input batches without swapping out its weights, and its backward pass computes the gradient of 2 batches without swapping out its gradient buffer.

❷ ***Just-in-time scheduling*** executes a task as soon as all its input tensors are available in GPU memory, avoiding delays in execution that risk unnecessarily swapping out the required input tensors, and then swapping them back in later. For example, in Figure 5 *jit-scheduling* brings the update task of each layer closer to its backward pass so that the weight and gradient tensors needed by the update tasks can be reused while they are still resident in GPU memory (*jit-update*). Similarly, the backward pass for the last layer ($L6$) can be scheduled immediately along with its forward pass for each microbatch (*jit-compute*), an optimization especially useful when it avoids the overheads of recomputation for the last layer.

❸ ***Generalized p2p swaps*** replaces CPU-GPU swaps, for *all* tensors (rather than only the per-stage output in prior work [45, 46]) that are shared across two layers, with fast device-to-device swaps where applicable. For the example in Figure 5, all input and output tensors of layers are transferred directly between the two GPUs.

❹ ***Multi-dimensional layer packing*** packs together multiple layers executing on a microbatch of input (e.g., forward pass, backward pass, or weight update). Consequently, both the pack size and the microbatch size of a task determine its memory footprint and performance. Prior work fixes one or both of these parameters, invariably punting the problem to model developers [20, 45, 63]. Harmony's Configuration Search Engine searches through separate layer packs for the forward and backward pass and their corresponding microbatch sizes to find the best training time configuration that balances compute, memory, and swaps.

Wrap-Around Pipeline. These techniques taken together result in a completely novel pipeline schedule in Harmony PP compared to prior work [20, 45, 46]. Like GPipe and PipeDream-Flush [45], Harmony PP also flushes the pipeline at the iteration end, thus providing synchronous SGD semantics. Unlike prior work that pins layers to GPUs (and with each GPU executing only one layer pack in both the forward and backward pass), each GPU in Harmony PP ends up executing *different forward and backward layer packs* enforced by the deterministic wrap-around schedule (e.g., in Figure 5, GPU1 ends up executing $L1$'s forward and $L2$'s backward pass). Binding of tasks across N devices in the wrap-around schedule, at a high level, can be described by the following pseudocode:

```
// Assumption: Task($P_B$[i]) also performs wt. updates
$P_{FB}$ = $P_F$ + Reverse($P_B$)
for i in range($P_{FB}$):
    Task($P_{FB}$[i]) $\rightarrow$ GPU[i mod $N$] // bind task to GPU
```

Furthermore, with per-GPU memory virtualization, prior approaches have to repeatedly swap out and then swap back in weights and gradients of layers while executing across microbatches (data parallelism and PipeDream's 1F1B); by contrast, Harmony PP *groups* the executions of a layer pack *across all microbatches* in a minibatch before scheduling the next layer pack on that GPU.

Intuitive Example to Highlight Advantages. To explain how Harmony significantly reduces swap overhead, using a simplified example we provide an analytical comparison between Harmony and the corresponding baselines that use per-GPU memory virtualization. We assume (without loss of generality) a setup with homogeneous GPUs where each GPU's memory capacity permits it to only hold one layer operating on one microbatch at any time. We also assume a simplified DNN model with one type of layer (like Transformers) and where each layer has the same runtime and memory footprint for its forward, backward, and update phases.

Harmony provides generalized support for swapping all tensors across different layers where they each need to swap in/out certain inputs/outputs (Figure 6(a)). First, we focus on a specific kind of tensor, model weights W (with a size of $|W|$), to provide an intuition for such reductions in swap overhead when training a model of R layers (i.e., $|W| = \sum_{j=1}^{R} |W_{Lj}|$) with m microbatches per GPU and N GPUs (for a minibatch of mN microbatches). Figure 6(b) shows that, for a single iteration (minibatch), when using DP with per-GPU memory virtualization, *each GPU* has to swap W in and out for both the forward and backward passes independently and this has to be done for *each of the m microbatches.* At the end of the iteration, each GPU also has to swap W in and out once for weight update. This results in an overall swap volume of $(4m+2)N|W|$ per iteration. By contrast, in Harmony DP (Figure 6(c)), each GPU has to swap W in *only once* each for the forward and the backward passes *across all m*

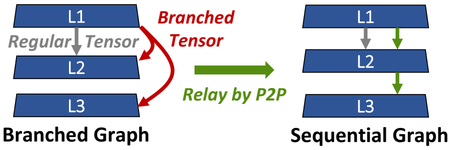


Figure 7: Serializing layer graphs in Harmony.

microbatches (due to *input-batch grouping*), and swap W out once for weight update (due to *jit-scheduling*), resulting in an overall swap volume of $3N|W|$ per iteration.

The same swap analysis also applies to PP with per-GPU memory virtualization. But the key difference is that PP does not have duplicated weight per GPU, canceling the N term in the swap volume, i.e., $(4m+2)|W|$. Finally, Harmony PP (Figure 5) combines the best of the two worlds with both *input-batch grouping* and no duplicated weights, bringing the overall per-iteration swap volume down to $3|W|$ (across all m microbatches and all N GPUs)!

For brevity, here we omit a full analytical comparison for all tensors shown in Figure 6 and refer the reader to the extended version of the paper [39]; suffice to say, Harmony offers swap reduction for all tensors and Harmony PP dominates reductions in swap volume. We empirically show the advantages of Harmony in § 5.

4 DESIGN AND IMPLEMENTATION

Harmony is implemented in Python (54K LOC) on top of PyTorch. Next, we present the details of Harmony's components in Figure 4.

4.1 Decomposer

Harmony's Decomposer constructs a fine-grained layer graph from an imperative-style PyTorch script and generates code so that each layer can be executed individually. The main challenge is dealing with branching in the model. Harmony overcomes this issue by relaying the branch tensor across downstream layers using p2p swaps until the destination layer consumes, thus minimizing CPU-GPU swaps. We have implemented such a *p2p-relaying* scheme to serialize the layer-level graph by adding identity nodes across layers as shown in Figure 7.

Unlike prior approaches that generate code for entire pipeline stages and bind them to a GPU early (e.g., PipeDream [45]), Harmony Decomposer uses the layer graph to generate code such that each layer can be invoked individually, and it delays layer packing and GPU binding to the downstream Harmony Scheduler.

4.2 Profiler

With the generated layer code, using a single GPU, Harmony's Profiler runs each layer individually and records profiles: compute time, memory footprint, and input tensor size. Since Harmony tunes both microbatch size and layer packs for both forward and backward pass, we also need to collect profiles for each layer under different microbatch sizes. Brute-force profiling with every possible microbatch size is impractical. Instead, Harmony sweeps through microbatch sizes to determine the maximum microbatch size that does not cause out-of-memory problems by using a process similar to TCP slow start (multiplicative increase of microbatch size, halving at the first OoM, and then additive increase until the next OoM). It then profiles layers for each microbatch size from 1 to this max size at fixed stride intervals. Finally, Harmony uses a simple regression model to interpolate each layer's characteristics for microbatch sizes that it does not sample. We validate the efficacy of the final profiling estimation, showing that it is strikingly accurate.

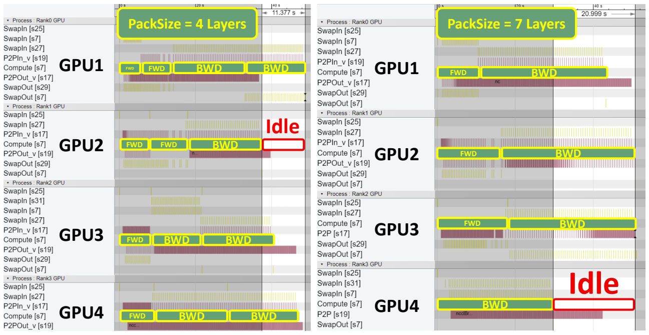


Figure 8: Greedily packing more layers to satisfy only memory capacity constraints can cause greater load imbalance across GPUs due to coarser-granularity tasks. Configurations: left: every 4 layers form a pack, $U_F = 30, U_B = 15$; right: every 7 layers form a pack, $U_F = 20, U_B = 10$.

4.3 Scheduler

Using the layer-granularity profiles and machine model, Harmony's Scheduler searches through the space of training configurations, estimating iteration time for each configuration and picking the fastest among them for execution by the Runtime.

4.3.1 Configuration Search Engine.

We define a configuration to be a four-tuple: <***forward microbatch size*** U_F***, forward layer packs*** P_F***, backward microbatch size*** U_B***, backward layer packs*** P_B>. Unlike prior work [20, 35, 46, 63], which either assumes the microbatch size to be specified by the user, or fixes the microbatch size and layer packs to be the same between the forward and the backward pass, Harmony's Configuration Search automatically determines the entire four-tuple. We expect users to specify a mini-batch size (not the microbatch size) as it directly affects convergence [14, 18, 31, 68]. But determining the four-tuple above is challenging for a number of reasons.

First, both the pack size and the microbatch size of of a task determine the memory footprint and performance when executing the task. It is not immediately clear if one should maximize the microbatch size or the layer pack size to maximally utilize the memory capacity of a device. Given a fixed memory capacity, increasing the pack size can reduce p2p and CPU-GPU swap volume (especially when using recompute [5]). Unfortunately, greedily constructing as large a pack as can fit the memory of individual GPU results in globally sub-optimal pipelines. Figure 8 shows such an example of training a BERT-Large with Harmony PP; the configuration with larger packs and smaller microbatch size (right) results in load imbalance across GPUs and up to 2× longer idle times than a configuration with smaller packs and larger microbatches (left).

Second, while it might be tempting to identify only backward packs and microbatch sizes, and reuse them for the forward pass (a scheme we term *Equi-FB*), this is far from optimal because the

```
Algorithm 1: Harmony Configuration Search
  Input   : number of layers $R$, minibatch size $D$,
            maximal forward microbatch size $U_{FMAX}$,
            maximal backward microbatch size $U_{BMAX}$,
            adopted packing method $\lambda$ (returns layer packs $P$),
            profiled time/memory/activation size $\phi$,
            GPU memory capacity $\alpha$, PCIe bandwidth $\beta$,
            Harmony mode $H$, number of GPUs $N$,
            task graph generator $\rho$, runtime estimator $\varepsilon$
  Output : best configuration $(U_F^*, P_F^*, U_B^*, P_B^*)$
  // find effective maximal microbatch size
1 if $H$ is "Harmony DP" then
2     $D \leftarrow D/N$
3 $U_{FMAX}, U_{BMAX} \leftarrow \min(U_{FMAX}, D), \min(U_{BMAX}, D)$
  // search for best config. with minimal time
4 $(U_F^*, P_F^*, U_B^*, P_B^*) \leftarrow None$ // best configuration
5 $t^* \leftarrow \infty$ // best runtime
6 for $U_B \leftarrow 1$ to $U_{BMAX}$ do
7     $P_B \leftarrow \lambda(\text{"B"}, U_B, R, \phi, \alpha)$ // backward packing
8     for $U_F \leftarrow 1$ to $U_{FMAX}$ do
9         $P_F \leftarrow \lambda(\text{"F"}, U_F, P_B, \phi, \alpha)$ // forward packing
          // from current config., generate task graph
10        $G \leftarrow \rho(U_F, P_F, U_B, P_B, H, N, D)$
11        $t \leftarrow \varepsilon(G, H, N, \phi, \beta)$ // estimate runtime
12        if $t < t^*$ then
13            $(U_F^*, P_F^*, U_B^*, P_B^*) \leftarrow (U_F, P_F, U_B, P_B)$
14            $t^* \leftarrow t$
15 return $(U_F^*, P_F^*, U_B^*, P_B^*)$
```

```
Algorithm 2: Balanced Time Packing $\lambda$
  Input   : forward or backward type $\tau$, microbatch size $U$,
            number of layers to pack $R$ (or given packs $P_B$),
            profiled time/memory/activation size $\phi$,
            GPU memory capacity $\alpha$
  Output : layer packs $P$
1 if $P_B$ exists then
2     $R \leftarrow P_B.RemoveLastPack().CountLayers()$ // jit
      compute
3 $t \leftarrow \phi(\tau, U, R).PerLayerTimeList()$
4 $m \leftarrow \phi(\tau, U, R).PerLayerMemoryList()$
  // loop num of packs from the smallest (largest packs)
5 $S_{min} \leftarrow m.Sum()/\alpha$
6 for $S \leftarrow S_{min}$ to $R$ do
      // find packs with per-pack time closely equal
7     $c \leftarrow t.Sum()/S$ // average per-pack time
8     $c' \leftarrow [c, 2c, \ldots, (S-1)c]$ // accumulated pack times
9     $t' \leftarrow t.PrefixSum()$ // accumulated layer times
10    $i \leftarrow BinarySearch(t', c')$ // insert c' into t' and get
       insertion points
11    $P \leftarrow t.Split(i).ToLayerID()$ // packs found
      // check if any pack is over capacity
12    for $p \leftarrow P[0]$ to $P[S-1]$ do
13        if $m[p].Sum() > \alpha$ then
14            break; continue // try smaller packs
15    return $P$ // balanced time and largest pack size
```

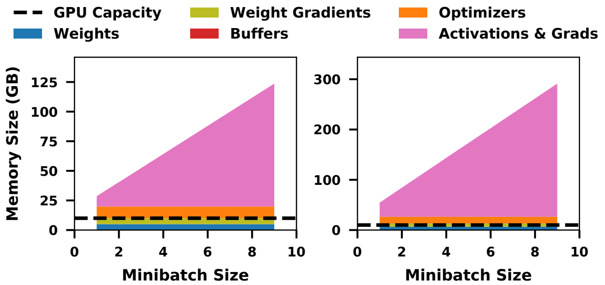
forward and the backward pass for the same layer can have very different characteristics. For example, it is common for the backward pass for a layer to have $2-3\times$ the runtime and memory footprint of the forward pass, thus motivating the need for different pack and microbatch sizes across these passes. Our experiments show that *Equi-FB* is 30% slower than picking separate values for forward and backward packs and microbatch sizes in a four-tuple configuration.

Heuristic-based Search. The problem of finding the optimal configuration that minimizes the training time can be shown to be NP-hard[3], which makes it unlikely that we can find a provably optimal configuration efficiently. We address this challenge by using a simple but effective heuristics-based search algorithm (Algorithm 1) to identify a high-performance four-tuple configuration. We proceed roughly as follows:

- We first determine the backward layer packs P_B for each backward microbatch size U_B (Lines 6, 7). This helps us identify the input tensors of each pack in P_B that we need to checkpoint in the forward pass; these input tensors will be used to recompute stashed tensors for all intermediate layers in the pack before we start the backward-pass compute for the task [5]. We can then use this information in determining the forward layer packs P_F for each forward microbatch size U_F we sweep through (Lines 8, 9). Furthermore, the last layer pack is shared between P_F and P_B, avoiding recompute for the first backward task (jit-compute, Line 2 of Algorithm 2).
- To reduce load imbalance across GPUs and avoid stragglers in the pipeline, we propose a method to determine layer packs that balances the time taken by each pack while maximizing average pack size. Algorithm 2 outlines our method, which runs in time $O(R^2)$ (invoked by Algorithm 1 at Lines 7, 9).
- For each configuration (U_F, P_F, U_B, P_B) to be explored, we generate a task graph, *binding* each task to an individual GPU (Algorithm 1, Line 10).
- We then *estimate* the end-to-end runtime of an iteration for a task graph (Algorithm 1, Line 11). The estimation leverages profiles of individual layers (ϕ) from Profiler, and performs an *event-driven simulation* to capture swap, transfer, and compute times. Simulating an iteration without actually running it on real hardware enables fast configuration search. Later, in evaluation, we show that these estimated times closely match real end-to-end runs (see Figure 15).
- The search returns the configuration with the best iteration time in the set of configurations explored (Algorithm 1, Lines 12–15).

In total, the time complexity of all steps in Scheduler (heuristic-based search, balanced time packing, task graph generation, runtime estimation) is $O(U_{FMAX} \cdot U_{BMAX} \cdot R(R+D))$, where $U_{FMAX/BMAX}$ is the maximal possible forward/backward microbatch size, R is the number of layers, and D is the minibatch size. In practice, this end-to-end scheduling time is less than 32 seconds (see Table 1).

[3] We omit the hardness proof here for brevity, but refer the interested reader to the Appendix in the extended version of this paper [39].



(a) BERT96

(b) GPT2

Figure 9: Memory footprint statistics for training massive models (using virtualized GPU memory).

4.3.2 Task Graph Generation.
A **task** is the unit of execution in Harmony. Figure 5 shows the three types of tasks: *Forward*, *Backward*, and *Weight Update*. Each task is associated with a layer pack and a microbatch size (the result of configuration search). Each task is bound to a specific execution backend (GPU device or CPU process). For instance, the second task in Figure 5, bound to GPU#2, is a *Forward* task for layer pack *[2,3]* with U_F=10 and a group of two microbatches. Each task also specifies the required inputs and outputs to be swapped in and out, respectively, along with the channels they ought to be transmitted on. For the same example, the task specifies its two inputs: tensor *L1 Output* over a *Peer2Peer* input channel, and *L2-3 Weight* tensors over a *CPU-GPU Swap* channel. The complete list of inputs/outputs to be swapped is shown in Figure 6(a), where each input/output can choose from one of the four channels: CPU-GPU Swap, Peer2Peer, Message Passing, and Shared Memory. Putting together the tasks of an entire iteration results in a **task graph**, where each node is a task and each edge is the specified input/output between tasks. Such task graphs are used to drive the Harmony Runtime.

Harmony DP and PP. Using the task graph, Harmony is able to schedule a variety of distributed training schedules; it does this by unrolling an iteration's tasks across GPUs. Harmony can support conventional Data Parallel and Pipeline Parallel training, both enhanced with per-GPU memory virtualization (that we hereafter call per-GPU swap). Crucially, it supports two new schedules, Harmony DP (Figure 6) and Harmony PP (Figure 5) – both these schemes benefit from Harmony's four key optimizations.

4.4 Runtime

Harmony's Runtime executes in CPU processes, one for each GPU in the deployment. This 1:1 mapping is required to enable effective concurrency and overcome the limitations of the Python GIL. Each runtime process can also be pinned to a CPU core on a socket which has NUMA affinity to the GPU it controls. All tasks run on GPUs, but Harmony also supports the *Weight Update* task being offloaded to the CPU. Each runtime process executes the ordered list of tasks in the unrolled task graph handed to the Runtime by the Scheduler, and repeats it for all training iterations.

CUDA Streams and CUDA Events. To effectively utilize the GPU and overlap computation and communication, Harmony uses *5 distinct CUDA streams*: one each for compute, swap-in, swap-out, p2p-in, and p2p-out on every GPU. We use CUDA events across streams to synchronize for task dependencies. The swap and p2p streams are managed by background CPU runtime threads for pre-allocating CPU tensors, prefetching GPU tensors for upcoming tasks, waiting for swap completion and tensor transfers. Prefetching uses extra GPU memory to overlap swaps/transfers with compute and uses double buffering to avoid repeated allocations.

Memory Manager. In PyTorch, each CUDA stream can allocate, free, and reuse its own memory. While streams can share memory buffers, memory reuse is private to each stream (e.g., the memory freed by stream A is not reusable by stream B); such private reuse can shrink the effective memory available to individual streams. To overcome this limitation, Harmony's Runtime employs a *"central" memory manager* on the compute stream and allows it to manage memory for all streams with an unified memory pool.

5 EVALUATION

5.1 Experimental Setup

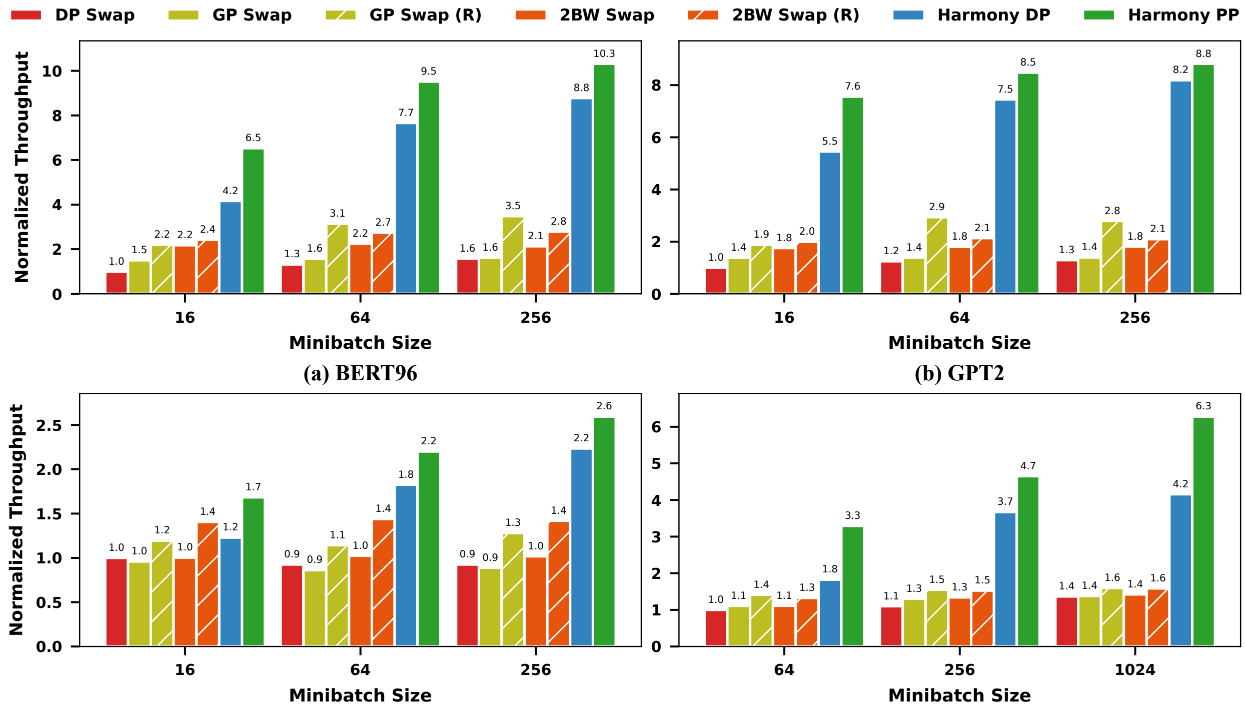
Configurations. We run experiments on three server configurations. Two of them are commodity servers with four and eight GTX-1080Ti GPUs (11 GB each) [2, 49], and 18-core (375GB DRAM) and 36-core (750GB DRAM) 2.3GHz Xeon CPUs [25], respectively. The third server is an NVIDIA DGX-2 with 16 V100 GPUs (32GB each), 96-core Xeon CPU (1.5TB DRAM), and NVSwitch [48, 52]. On all servers, GPUs are connected to CPU via a PCIe tree as in Figure 3a, where each link is a 16-lane PCIe3 (16GB/s per direction). All results shown are with PyTorch 1.5, NCCL 2.4, CUDA 10.1, and FP32 precision. *Unless explicitly stated, we show evaluation results on the commodity server configuration with four GTX-1080Ti GPUs.*

Models. Our evaluation uses the following DNN models:

- Two BERT variants: **BERT-Large** (24 transformer layers) [8] and **BERT96** (96 transformer layers) [46]. Both models use a sequence length of 512 and training uses the GLUE dataset [65] with an Adam optimizer.
- Three GPT2 variants: **GPT2** (the default model with 1.5B parameters) [57], **GPT2-Medium** (0.3B) [43], and customized GPT2 models (**10s Billion**) [59]. All are trained with a sequence length of 1024 on WikiText dataset [42] and an Adam optimizer.
- **VGG416**. This is a variant of the classic VGG model scaled to have 416 layers and has been used for evaluating per-GPU memory virtualization in prior work [17, 32, 61]. It is benchmarked for training using the ImageNet [7] dataset with a SGD optimizer.
- **ResNet1K**. Another CNN, a ResNet variant [16, 29], used for evaluating per-GPU memory virtualization in prior work [5, 28].

Memory Footprint. The working set size of these models exceeds the combined memory capacity of our GPUs; the memory footprint far exceeds the capacity of an individual GPU for even the smallest batch sizes. Figure 9 analyzes the memory footprint of training two massive models at different batch sizes and also breaks down the memory footprint into important components (weights, gradients, etc.). The memory footprint analysis of other models is similar.

Per-GPU Swap Baselines. Given the prohibitive memory requirements mentioned above, we enhance existing approaches for parallel DNN training, such as Data Parallelism (DP) [37], GPipe (GP) [20], and PipeDream-2BW (2BW) [46], to incorporate per-GPU



(a) BERT96

(b) GPT2

(c) VGG416

(d) ResNet1K

Figure 10: Performance comparison with per-GPU swap baselines by training different models with various minibatch sizes on 4 GPUs. Each group of bars represents one minibatch size. *R* denotes the usage of recompute for activations. Throughput is normalized against `DP Swap` at the smallest minibatch size (i.e., the leftmost bar).

memory virtualization using IBM-LMS [23, 24]. As a result, we construct new viable baselines for comparison to Harmony: **`DP Swap`**, **`GP Swap`**, and **`2BW Swap`**. Furthermore, we augment these baselines with memory optimizations: 1) *gradient accumulation* [46], 2) *in-place operations* and *memory reuse* [5, 66], and 3) other memory buffer optimizations. While Harmony always uses recompute [5] to cut the memory of stashed activations, we also enable `GP Swap` and `2BW Swap` to use recompute, thus creating additional baselines **`GP Swap (R)`** and **`2BW Swap (R)`** respectively.

ZeRO-Infinity. We also compare against ZeRO-Infinity [59], a recent swap-based DP enhancement that supports moving state between CPU and GPU memory, offloads weight update to CPU, and shards model state across GPUs only to swap in every layer's state when required for (re)compute on each GPU. ZeRO-Infinity also includes NVMe storage devices in the memory hierarchy, if available (e.g., high-end DGX-2 servers [52]). In this paper, we only consider massive models whose working set fits in CPU memory and thus Harmony does not use storage devices for swaps (many commodity servers lack fast NVMe devices).

Goal. We seek to answer the following questions in evaluation:

- How does Harmony compare to baselines, with respect to training throughput and swap overhead? (§ 5.2-5.3)
- Is Harmony training correct (converges as baseline)? (§ 5.4)
- How much does each of our optimizations contribute? (§ 5.5)
- How does Harmony's Scheduler perform? (§ 5.6)
- How does Harmony scale with model sizes and GPUs? (§ 5.7)

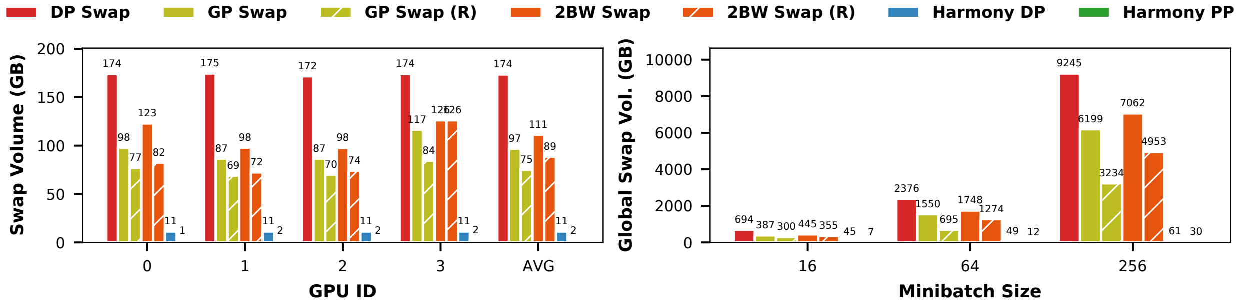
5.2 Comparison with Per-GPU Swap Baselines

Figure 10 compares Harmony with per-GPU swap baselines for different minibatch sizes. We highlight ***five key takeaways***:

First, for any given minibatch size, **`DP Swap`** ***consistently underperforms other approaches*** – unsurprisingly, given that each of the 4 GPUs is swapping the entire model state back and forth to CPU memory including unnecessary and repeated swaps across microbatches (§ 2). Figure 11 further reveals that `DP Swap` dominates the swap volume over other approaches.

Second, **`GP Swap`** ***is consistently worse than*** **`2BW Swap`** not just due to swap load but also due to pipeline flushes in `GPipe`. But because swap overheads dominate, the gap between `GP Swap` and `2BW Swap` is less dramatic than when the model fits the collective memory capacity of all GPUs [46]. The baselines using ***recompute,*** **`GP Swap (R)`** ***and*** **`2BW Swap (R)`*****, perform much better than their no-recompute counterparts (*****`GP Swap`** ***and*** **`2BW Swap`*****)*** across all models and batch sizes, and this can be directly attributed to the reduced swap overheads due to recompute, which indicates that swap overhead dominates over compute cost. Figure 11(a) shows this reduction in swap overheads.

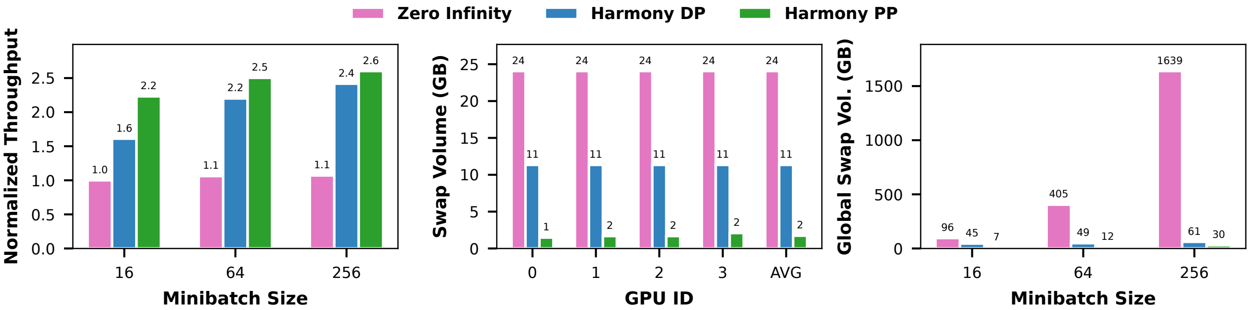
Third, ***Harmony DP benefits from input-batch grouping, jit-scheduling, and layer packing, significantly outperforming all baselines*** (Figure 10), with speedups up to 2.4× ~ 7.0× for all models. Harmony DP's swap overheads are an order of magnitude lower than `DP Swap` (Figure 11).



(a) Per-GPU comparison at minibatch size 16

(b) Comparison under different minibatch sizes

Figure 11: CPU-GPU swap volume comparison of different approaches for training GPT2 on 4 GPUs. Swap volume is measured per minibatch. Global swap volume aggregates swap volume across all GPUs.



(a) Performance

(b) Swap load at minibatch size 16

(c) Global Swap load

Figure 12: Comparison with ZeRO-Infinity for training GPT2 (1.5B) on 4 GPUs. Throughput is normalized against ZeRO-Infinity at minibatch size 16 (i.e., the leftmost bar). CPU-GPU swap volume is measured per minibatch.

Fourth, ***Harmony PP is consistently the fastest approach across all models and minibatch sizes*** (Figure 10), with speedups up to 2.8× ∼ 7.6× over DP Swap. It is up to 1.5× faster than Harmony DP, further benefiting from pipeline parallelism (eliminating all redundancy in CPU-GPU swaps) and p2p swaps, with a swap volume that is *two orders of magnitude* lower than DP Swap (Figure 11).

Fifth, across all models, ***Harmony's speedup over baseline approaches widens with larger batch sizes***. This is primarily fuelled by reduced swap load due to *input-batch grouping* in Harmony. Figure 11(b) shows that while swap load proportionally goes up for all approaches as we increase batch size, the swap volume across all GPUs is 100× ∼ 300× higher for per-GPU swap baselines compared to Harmony, thus resulting in a flatlining of throughput for baselines (Figure 10).

5.3 Comparison with ZeRO-Infinity

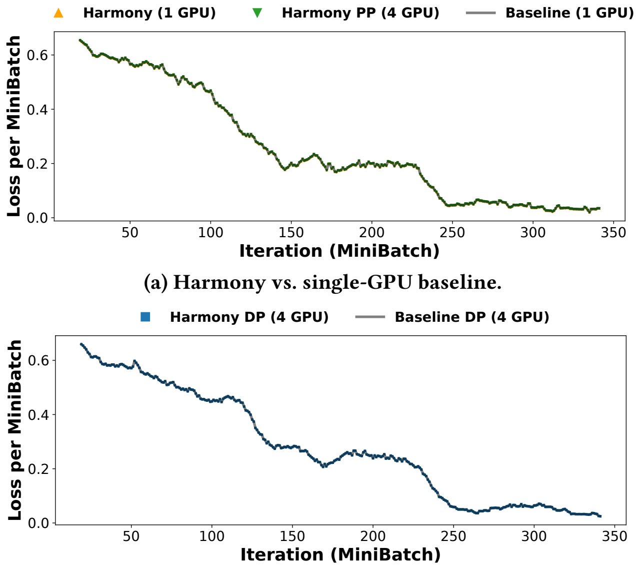
We now compare Harmony to ZeRO-Infinity on our deployment. ZeRO-Infinity suffers from coarse-grained scheduling and lacks optimizations such as input-batch grouping and configuration search for principled layer packing. For a fair comparison, in our evaluation, we make ZeRO-Infinity share the same configuration as Harmony (i.e., minibatch size, microbatch size, pack size for recompute) that Harmony finds and enable all its relevant optimizations. Figure 12(a) shows that ***Harmony DP and PP are up to*** 2.3× ***and*** 2.5× ***faster than ZeRO-Infinity***, respectively, for GPT2. Harmony's throughput *speedup widens as the minibatch size increases.* Figures 12(b,c) show that this speedup can be directly attributed to *an order-of-magnitude lower swap load* in Harmony with input-batch grouping (and p2p swaps in Harmony PP).

5.4 Correctness of Training in Harmony

Harmony provides synchronous SGD semantics and should leave convergence properties of models unchanged compared to settings where the entire model would fit in memory. To validate this, we compare the training loss for every minibatch in Harmony (with swaps) with the equivalent training loss of a baseline scheme without swaps, when using the same hyper-parameters and for models that can fit in GPU memory. Harmony PP provides a single-GPU abstraction, and hence we compare it to accuracy results from single-GPU runs. In Figure 13, fine-tuning results of BERT-Large on downstream MRPC tasks show a ***perfect match in loss values for every minibatch*** between Harmony's schemes and baseline runs. We also achieve *perfect match in the final evaluation accuracy* of the trained model: 88.0% across Harmony and baseline runs.

5.5 Efficiency Breakdown of Harmony

Figure 14 analyzes the efficacy of Harmony's optimizations. We highlight five key takeaways. First, *input-batch grouping* significantly reduces swap load and increases arithmetic intensity; without this optimization iteration times are 2.2× and 1.5× slower in Harmony DP and PP respectively. Second, expert (manually) picked layer packs and microbatch sizes for even repeated-structure



(a) Harmony vs. single-GPU baseline.

(b) Harmony vs. data-parallelism baseline.

Figure 13: Correctness of Harmony. An example of fine-tuning BERT-Large on MRPC of GLUE with reported hyper-parameters [8] and baseline code [21]. Harmony matches the baseline exactly for every minibatch.

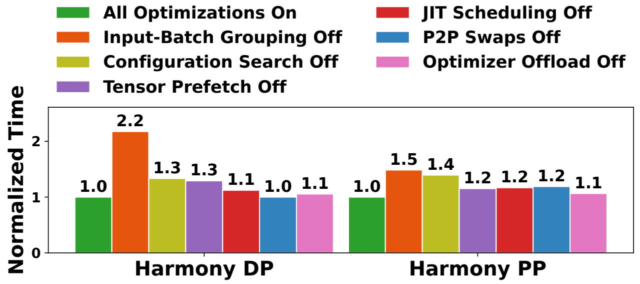


Figure 14: Efficiency breakdown of Harmony for training GPT2 on 4 GPUs. Each bar shows the resulting slowdown when turning off only one optimization while keep others on. Y-axis is normalized against "All Optimizations On" for Harmony DP and PP separately. Higher is worse.

transformer-based DNNs result in 1.3–1.4× worse throughput compared to Harmony's automated *configuration search*. Third, forgoing *tensor prefetch* can result in up to 1.3× slower iteration times. Fourth, excluding *jit scheduling* and *optimizer offload* can individually degrade throughput by up to 1.2×, although *optimizer offloading* seems to be less critical. Fifth, *p2p swaps* don't provide any benefits for Harmony DP, but Harmony PP which actively uses GPU-GPU swaps across layer packs in the pipeline can suffer degraded iteration times by as much as 1.2× when disabling *p2p swaps*.

5.6 Scheduler and Configuration Search

To evaluate the effectiveness of Harmony Scheduler, we measure its end-to-end time including iterative configuration search, task graph generation, and runtime estimation, until the best configuration is selected. Table 1 shows that ***reaching the best configuration takes at most 32 seconds***. For transformers, it is about 1 second; CNNs like ResNet1K are much deeper and richer in diversity of layer attributes (memory size and compute time).

Table 1: Configuration search results and Scheduler end-to-end time (config search, task graph generation, runtime estimation) with Harmony PP (4 GPUs, minibatch size 64).

Model	BERT96	GPT2	VGG416	ResNet1K
U_F	16	4	8	32
$\|P_F\|$	24	10	15	2
U_B	16	4	8	32
$\|P_B\|$	25	17	16	9
Time (s)	1.4	0.7	17.7	31.6

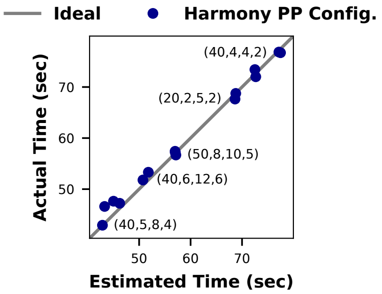


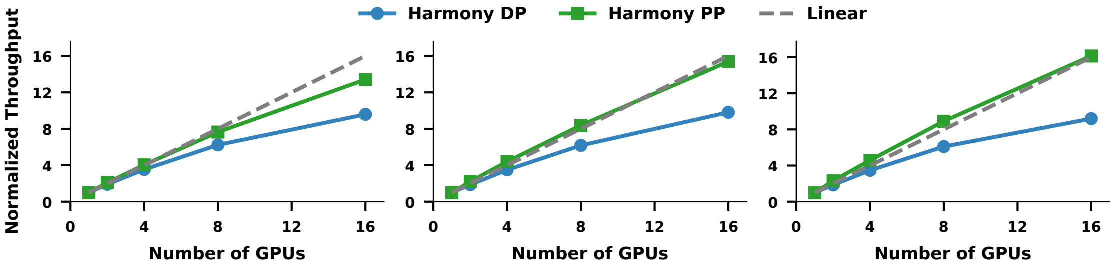
Figure 15: Accuracy of Harmony's Runtime Estimator. We compare estimated iteration time with actual time for training BERT-Large with a mini-batch size of 600 on 4 GPUs using Harmony PP. Each dot represents a Harmony configuration (U_F, $|P_F|$, U_B, $|P_B|$) and the 15 points here are sampled randomly from all the configurations that Harmony explores. The relative difference between estimated and actual time is within 5% on average.

Figure 15 evaluates the quality of Harmony's *Runtime Estimator*. It compares the estimated training time with actual training time in each searched configuration. Estimated training time is obtained from Harmony's *event-driven simulator* (§ 4.3.1); for each configuration, the simulator uses a Harmony task graph and layer profiles for estimating end-to-end iteration time. We observe that *Harmony's estimates are accurate*, giving us confidence in its selection of configurations with the best throughput.

5.7 Scaling Model Size and Number of GPUs

To evaluate how Harmony scales, we now use two beefier servers as mentioned in § 5.1 – i) a server with *eight* GTX-1080Tis (11GB) and ii) a DGX-2 with 16 V100 GPUs (32GB). We use this setup to understand not only the limits of how large a model Harmony can train given CPU memory capacity bounds but also Harmony's scalability in number of GPUs. We customize the GPT2 model to scale up to tens of billions of parameters [59].

First, we study the limit of trainable model size. Figure 17 shows the throughput of training such models on an 8× GTX-1080Ti server. For fairness, ZeRO-Infinity shares the same configuration as Harmony and with all optimization flags on. For the 10~30-billion



(a) 40 Billion (b) 60 Billion (c) 80 Billion

Figure 16: Scalability of Harmony in training massive models of 10s of billions of parameters with 16 V100s on a DGX-2 server. The 80-Billion model saturates CPU memory capacity (1.5TB). Throughput is normalized against single-GPU Harmony DP.

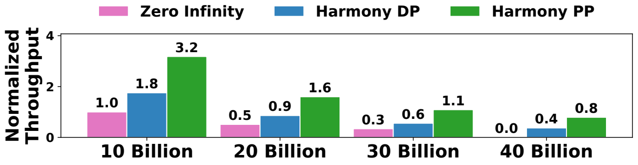
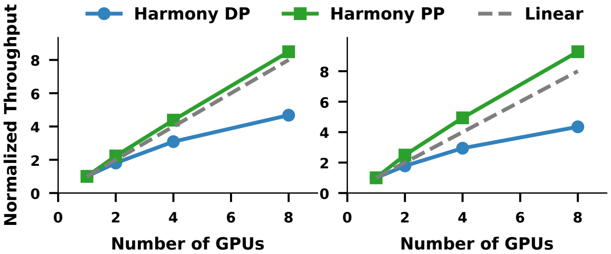


Figure 17: Training massive models of 10s of billions of parameters at the limit of single-server CPU memory capacity (750GB CPU and eight GTX1080Tis). Throughput is normalized against ZeRO-Infinity's 10-Billion model.



(a) 10 Billion (b) 40 Billion

Figure 18: Scalability of Harmony in training massive models of 10s of billions of parameters with eight GTX1080Tis. Throughput is normalized against single-GPU Harmony DP.

models, ***Harmony DP and PP are still consistently faster than ZeRO-Infinity by up to*** 1.8× ***and*** 3.2×, respectively. At 40 billion parameters, the working set size of the model hits the limits of the server's CPU memory capacity; *Harmony offers proportionally scaled throughput* compared to the 10-billion parameter model on account of 4× more compute, while ZeRO-Infinity fails to train due to running out of CPU memory.

Second, we scale the number of GPUs from 1 to 8 on the GTX-1080Ti server (Figure 18) and from 1 to 16 on the DGX-2 server (Figure 16). When using ***Harmony PP, training throughput scales linearly*** due to reduced swap overhead and advantages of p2p swaps. Harmony DP, despite performing well, is affected by the overhead of weight swaps duplicated across GPUs, and the performance gap between Harmony DP and PP widens as model size grows (due to greater swap volume).

6 DISCUSSION

Scope of Harmony. Harmony aims at training massive models that are *out of GPU memory capacity*, which is of great value for *developing*, *debugging*, and *fine-tuning* massive models on a single commodity server with only several GPUs [9]. Our scope differs from prior work that requires memory footprint to *fit GPU memory capacity* and focuses on *pre-training* with hundreds of GPUs in a datacenter [11, 46, 58, 63]. Pre-training extremely large models on a commodity server might be infeasible. For instance, pre-training GPT3 end to end requires 314 ZettaFLOPs [4] and takes several months of training even with thousands of cutting-edge GPUs [47]. There is no denying that training on a large cluster will naturally result in speedier training. However, despite this limitation, we believe that Harmony can still enable development and debugging of such models on modest deployments (before they are deployed for pre-training at a large scale), and fine-tuning of massive models that requires less than 10s of exaFLOPs [3, 8, 30] clocking in at days with a commodity server [22].

Multi-machine Training. Harmony's prototype operates on a single machine. However, *task decomposition* and *late binding*, together with Harmony's four key optimizations and Wrap-Around Pipeline, all extend to multi-machine training. When collective capacity of all GPUs is sufficient to hold the memory footprint of massive models, memory swapping becomes unnecessary. Nonetheless, the single-GPU abstraction of Harmony can still benefit developers by decoupling the model definition from a particular training parallelism, allowing to focus on model development without worrying about complexity and deadlock [33, 45] in parallel training.

7 CONCLUSION

One of the main challenges for training massive DNN models on single-server multi-GPU deployments is the limited GPU memory capacity. Current solutions that rely on virtualizing GPU memory incur excessive swap overheads. We advocate rethinking how DNN frameworks schedule computation and move data, and we articulate the principles, functionality, and optimizations needed to push the boundaries of training massive models efficiently on modest deployments. Across various massive DNN models, Harmony is able to reduce swap load by up to two orders of magnitude and obtain a training throughput speedup of up to 7.6× over highly optimized baselines with virtualized memory.

REFERENCES

[1] Christof Angermueller, Tanel Pärnamaa, Leopold Parts, and Oliver Stegle. 2016. Deep learning for computational biology. *Molecular systems biology* 12, 7 (2016), 878.

[2] ASUS. 2019. High-density 4U GPU server, https://www.asus.com/us/Commercial-Servers-Workstations/ESC8000-G4.

[3] Aishwarya Bhandare, Tianju Xu, and Kshama Pawar. 2020. GPT-2 fine-tuning with ONNX Runtime. *Microsoft Open Source Blog* (2020). https://cloudblogs.microsoft.com/opensource/2020/08/24/pytorch-gpt-2-fine-tuning-onnx-runtime-speedup-training-time

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

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

[6] Minsik Cho, Tung D Le, U Finkler, Haruiki Imai, Yasushi Negishi, Taro Sekiyama, Saritha Vinod, Vladimir Zolotov, Kiyokuni Kawachiya, David S Kung, et al. 2018. Large model support for deep learning in caffe and chainer. *SysML'18* (Feb. 2018).

[7] Deng, Jia and Dong, Wei and Socher, Richard and Li, Li-Jia and Kai Li and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. 248–255.

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

[9] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. 2021. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *Annual Technical Conference (ATC'21)*. 381–396.

[10] Facebook. 2020. Distributed Data Parallel in PyTorch, https://pytorch.org/docs/master/notes/ddp.html.

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. 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 (PPoPP '21)*. 431–445.

[12] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2021. AI and Memory Wall. *RiseLab Medium Post* (2021).

[13] Google. 2018. TensorFlow code and pre-trained models for BERT, https://github.com/google-research/bert.

[14] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[15] The Guardian. 2020. A robot wrote this entire article. Are you scared yet, human? *The Guardian* (2020).

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. Las Vegas, NV, 770–778.

[17] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 875–890.

[18] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS'17)* (Long Beach, California, USA). 1729–1739.

[19] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland, 1341–1355.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33st International Conference on Neural Information Processing Systems (NeurIPS'19)*. Vancouver, Canada, 103–112.

[21] Huggingface. 2018. PyTorch Pretrained Bert, https://github.com/maknotavailable/pytorch-pretrained-BERT.

[22] Huggingface. 2021. Transformer Examples, https://huggingface.co/transformers/v2.3.0/examples.html.

[23] IBM. 2018. TensorFlow Large-Model-Support, https://github.com/IBM/tensorflow-large-model-support.

[24] IBM. 2020. PyTorch Large-Model-Support, https://github.com/IBM/pytorch-large-model-support.

[25] Intel. 2017. Intel Xeon Processors, https://www.intel.com/content/www/us/en/products/details/processors/xeon.html.

[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *The ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. Los Angeles, CA, 7132–7141.

[27] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems (MLSys'20)*, Vol. 2. 497–511.

[28] Hai Jin, Bo Liu, Wenbin Jiang, Yang Ma, Xuanhua Shi, Bingsheng He, and Shaofeng Zhao. 2018. Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO'18)* (2018), 1–26.

[29] Kaiming He and Xiangyu Zhang and Shaoqing Ren and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. *arXiv preprint arXiv:1603.05027* (2016).

[30] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[31] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NeurIPS'12)*. Lake Tahoe, NV, 1097–1105.

[33] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *Proceedings of the International Conference on Learning Representations (ICLR'21)*.

[34] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.

[35] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO, 583–598.

[36] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[37] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *The 46th International Conference on Very Large Databases (VLDB'20)* 13, 12 (aug 2020), 3005–3018.

[38] Youjie Li, Amar Phanishayee, Derek Murray, and Nam Sung Kim. 2021. Doing More with Less: Training Large DNN Models on Commodity Servers for the Masses. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)*. Ann Arbor, Michigan, 119–127.

[39] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. 2022. Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers. *arXiv preprint arXiv:2202.01306* (2022). https://arxiv.org/abs/2202.01306

[40] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS'18)*. Montreal, Canada, 8056–8067.

[41] Farhad Manjoo. 2020. How Do You Know a Human Wrote This? *The New York Times* (2020).

[42] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[43] Microsoft. 2020. GPT-2 fine-tuning with ONNX Runtime, https://cloudblogs.microsoft.com/opensource/2020/08/24/pytorch-gpt-2-fine-tuning-onnx-runtime-speedup-training-time.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602* (2013).

[45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Huntsville, Canada, 1–15.

[46] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel DNN training. In *International Conference on Machine Learning (ICML'21)*. 7937–7947.

[47] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021.

Efficient Large-Scale Language Model Training on GPU Clusters. *arXiv preprint arXiv:2104.04473* (2021).

[48] NVIDIA. 2016. NVLINK AND NVSWITCH, https://www.nvidia.com/en-us/data-center/nvlink/.

[49] NVIDIA. 2017. GEFORCE GTX GPU, https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1080-ti/.

[50] NVIDIA. 2017. NVIDIA DGX-1 System Architecture White Paper, https://www.azken.com/images/dgx1_images/dgx1-system-architecture-whitepaper1.pdf.

[51] NVIDIA. 2017. Unified Memory, https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[52] NVIDIA. 2018. NVIDIA DGX-2H The World's Most Powerful System for The Most Complex AI Challenges, https://www.nvidia.com/content/dam/en-zz/es_em/Solutions/Data-Center/dgx-2/dgx-2h-datasheet-us-nvidia-841283-r6-web.pdf.

[53] NVIDIA. 2021. NVIDIA TESLA GPUs, https://en.wikipedia.org/wiki/Nvidia_Tesla.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NeurIPS'19)*. Vancouver, Canada, 8024–8035.

[55] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU memory management for deep learning. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland, 891–905.

[56] PNY. 2021. Single Root Complex Purley 4U GPU Server for Deep Learning Applications, https://www.pny.eu/en/consumer/explore-all-products/pny-gpu-servers/\983-single-root-complex-purley-4u-gpu\-server-for-deep-learning-applications.

[57] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *Technical report, OpenAI* (2019).

[58] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimization towards training a trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)* (Atlanta, Georgia). 1–16.

[59] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv preprint arXiv:2104.07857* (2021).

[60] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv preprint arXiv:2101.06840* (2021).

[61] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Taipei, Taiwan, 1–13.

[62] Ram Sagar. 2020. OpenAI Releases GPT-3, The Largest Model So Far. *Analytics India Magazine* (2020).

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

[64] Ke Sun, Yang Zhao, Borui Jiang, Tianheng Cheng, Bin Xiao, Dong Liu, Yadong Mu, Xinggang Wang, Wenyu Liu, and Jingdong Wang. 2019. High-resolution representations for labeling pixels and regions. *arXiv preprint arXiv:1904.04514* (2019).

[65] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[66] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'18)*. Wien, Austria, 41–53.

[67] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[68] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large Batch Training of Convolutional Networks. *arXiv preprint arXiv:1708.03888* (2017).