Doing more with less: Training large DNN models on commodity servers for the masses

Youjie Li[*]
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Amar Phanishayee
Microsoft Research
Redmond, WA, USA

Derek Murray
Microsoft
Mountain View, CA, USA

Nam Sung Kim
University of Illinois at Urbana-Champaign
Champaign, IL, USA

ABSTRACT

Deep neural networks (DNNs) have grown exponentially in complexity and size over the past decade, leaving only the elite who have access to 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 access to 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 large DNN models can often exceed the aggregate capacity of all available GPUs on commodity servers; 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 advocate rethinking how DNN frameworks schedule computation and move data to push the boundaries of training large models efficiently on modest multi-GPU deployments.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; *Parallel algorithms*; • **Software and its engineering → Memory management**; **Communications management**.

[*]Work done as part of MSR internship.

ACM Reference Format:
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 *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458336.3465289

1 INTRODUCTION

Modern DNN models have transformed our approach to solving a range of hard problems such as image classification [12, 16, 18, 32, 55, 68], semantic segmentation [63], image generation [14], translation [53, 65], and language modeling [5, 39, 52, 62, 65]. Over the years, these models have grown exponentially in size while continuing to achieve unprecedented accuracy on ever more complex tasks [1, 29, 40]. For example, a 557-million-parameter AmoebaNet can achieve super-human accuracy in image classification tasks [20]. Similarly, a state-of-the-art language model like the 175-billion parameter GPT-3 [5] can produce human-like text [15, 38, 61]. As these models grow, they also become more computationally expensive. Training these models to accuracy takes weeks to months of wall-clock time, despite running in parallel on large clusters of fast accelerators.

These resource demands leave only the *elite* (e.g., Google, Microsoft, Facebook, OpenAI, NVIDIA, etc.), who have access to massive datacenter-based resources, with the ability to train such models. The long-tail of researchers, the *masses*, who have access to only limited resources (e.g., a single server with a couple of 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 large models on *modest multi-GPU* deployments.

Challenges.

One of the main challenges of training large models is that the memory footprint of training far exceeds the memory capacity of fast accelerators. Fig. 1 shows how sizes of models for image classification and language modeling tasks

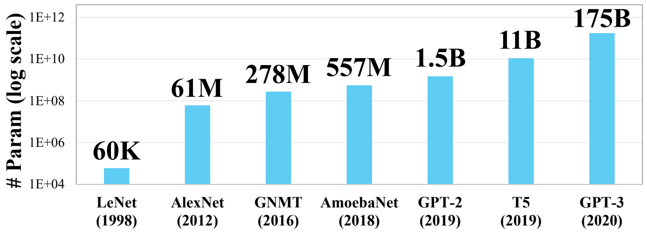


Figure 1: DNN model size growth for image classification (LeNet, AlexNet, AmoebaNet) and language modeling (GNMT, GPT-2, T5, GPT-3) over two decades.

have grown dramatically over time. Furthermore, model parameters are only part of the memory footprint of training; gradients, stashed activations, optimizer states, and framework workspace all taken together significantly blow-up the memory footprint [7, 27, 54, 58, 64]. Finally, the memory footprint is also a function of the size of individual input item (*sample size*) and the number of items in an input batch (*batch size*). While batch size can usually be reduced, a similar reduction in sample size is hard to pull off especially due to its effect on model accuracy [33, 60]. Taken together, the memory footprint for a large model can far exceed individual accelerator memory capacity.

This memory footprint problem motivates recent innovations that alleviate memory pressure. For example, recent advances in techniques that virtualize GPU memory push the boundaries of what can be achieved on a single GPU [9, 19, 50, 58], but as we show in Section 2 such techniques are inefficient for parallel multi-GPU training. Other techniques such as encoding data structures [27], recomputation [7, 50, 64], optimizer state sharding [54] or offloading the optimizer to CPUs [57], and modes of parallelism that split a model across multiple accelerators such as model- [31, 42, 62] or pipeline-parallelism [8, 20, 22, 42, 43] all aim to reduce memory pressure during training. However, despite these memory optimizations, on modest deployments (single server with a handful of commodity GPUs), the general problem of efficiently training massive models, which exhaust the *collective* memory capacity of available accelerators, is still an open problem.

We argue that current DNN frameworks have two fundamental problems that limit large model training on modest deployments. First, they *schedule work at a coarse granularity*, treating the training program as a black box: all DNN layers in data-parallel training or a set of contiguous layers in pipeline-parallel training. 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, one input batch at a time, limits reuse of tensors loaded into memory by intermediate layers 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, the parameter state associated with a single DNN layer is bound to a particular 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.

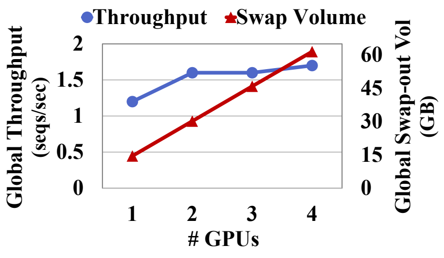
A united stand – the power of Harmony.

Ideally, users could write DNN training programs that target a single virtual accelerator device with practically unbounded memory. Our proposed system, Harmony, introduces three key ideas towards this ideal. First, we decompose the operations in a training script into fine-grained tasks and introduce a distributed on-line 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-, model-, or pipeline-parallel fashion, and Harmony transparently introduces collective communication operations (like AllReduce) to preserve the semantics of the original tasks. Second, we further decompose individual operations—such as a matrix multiplication—into subtasks that can run on different physical devices. Third, to support PyTorch and TensorFlow programming models based on imperative updates to mutable state, we generalize previous work on GPU memory swapping [9, 19, 50, 58] to build a coherent virtual memory across all available CPU and GPU memory. The scheduler and swapping algorithms in Harmony inform each other's decisions to maximize throughput with the available resources.

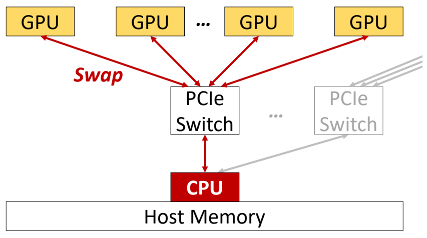
Conceptually, our approach to DNN scheduling is similar to coarse-grained task scheduling architectures used in systems like MapReduce [10], Dryad [25], Spark [67], and Ray [41]. We expect to leverage many ideas from literature around these systems, such as flow-based scheduling for data locality [13, 26], delay scheduling [66], and low-latency load balancing [48]. However, DNN training raises two additional problems that motivate new research: "fine-grained" tasks may be as short as a few microseconds, and the standard implementation of SGD-based optimization algorithms rely on in-place state mutation, so we cannot rely on having pure tasks with immutable inputs.

Four principles guide our design for efficient training:

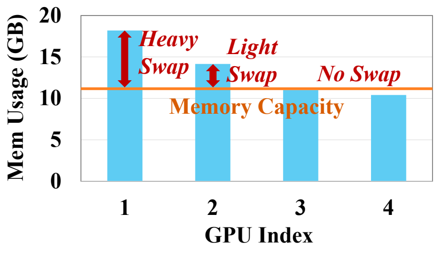
1. Minimize memory swaps. Harmony attempts to reuse state in GPU memory, minimizing swaps when using GPU memory virtualization. Empowered by the flexibility of scheduling at a finer granularity, we propose a technique called *input-batch grouping*, where a scheduled operator can be run across a group of input batches before scheduling the next



(a) DP with per-GPU tensor swapping



(b) Intra-server interconnects.



(c) PP with per-GPU tensor swapping.

Figure 2: The swap bottleneck of training BERT [11] via data parallelism (DP) [37] and pipeline parallelism (PP) [43] on a server with four NVIDIA 1080Ti GPUs each with 11GB of memory. Training the model with a per-GPU batch size of 5 using PyTorch-1.5 [49], and IBM-LMS [24] for virtualizing individual GPU memory, results in memory footprint exceeding GPU memory capacity. (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 bottleneck pipeline stages.

operator, thus improving state reuse in GPU memory and consequently improving arithmetic intensity.
2. Schedule tasks just-in-time. Harmony schedules tasks as soon as all input dependencies are available; this especially helps tasks such as weight update, which in PyTorch are normally scheduled to execute after the backward pass for the entire model, resulting in avoidable CPU-GPU swaps.
3. Swap over fast peer-to-peer links. With late binding of tasks to GPUs, Harmony can place adjacent tasks across GPUs and transfer required state directly between GPUs using *p2p transfers* rather than swapping state back and forth to CPU memory (as in naive GPU memory virtualization).
4. Balance load. Late binding also enables Harmony to *pack tasks* to balance compute, memory, and swap load across accelerators. Such multi-dimensional load balancing aids in parallel training schedules without pipeline bottlenecks.

In this paper, we show how task decomposition and late binding, together with a set of novel performance optimizations, enable virtualized parallel training of large DNNs.

2 LIMITS OF MEMORY VIRTUALIZATION

The problem of a workload's working set size exceeding memory capacity constraints has a long history in CPU-based systems. Demand-paged virtual memory is the standard approach for alleviating memory capacity constraints [6, 17]. Several recent projects have applied this idea to training DNNs on GPUs. They focus on increasing the effective memory capacity when training models on a *single* GPU and and seem to be promising techniques for large model training: GPU memory virtualization [46], backing GPU memory with CPU memory in the memory hierarchy, and swapping data structures automatically between CPU and GPU memory [9, 19, 23, 24, 28, 50, 56, 58, 64, 68]. We refer to these techniques collectively as *GPU memory virtualization* in this paper. However, such techniques are limited to only individual GPUs considered in isolation. Here we show that per-GPU memory virtualization is inefficient as it causes either a high swap overhead when used in data-parallel training; or imbalanced swap overheads in pipeline-parallel training, where the stage with the highest swap load is the bottleneck.

Today's frameworks have four key inefficiencies that cause these swap-overhead related performance problems in data- and pipeline-parallel distributed training:
1. Repeated Swaps. An operator can consume different input data or intermediate stashed tensors at different times, but it always requires the same weight tensors or gradient buffers. With GPU memory virtualization, these common weight and gradient tensors/buffers are swapped in and out repeatedly across batches of data.
2. 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 large 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 a typical PyTorch script, the weight update for each layer only starts after the backward pass for the entire model, potentially causing unnecessary swaps of some layer weights and gradients.
3. Only CPU-GPU Swaps. GPU memory virtualization lacks context about distributed training, works in isolation to other GPUs, and can only swap to host memory. This exposes the bottleneck device-to-host interconnect (Fig. 2(b)) and misses the opportunity to use fast device-to-device links for cross-device swaps or tensor communication. Fig. 2(a) shows that for data-parallel training the swap overhead across multiple GPUs throttles throughput as the total swap load across multiple GPUs exposes the bottleneck to host

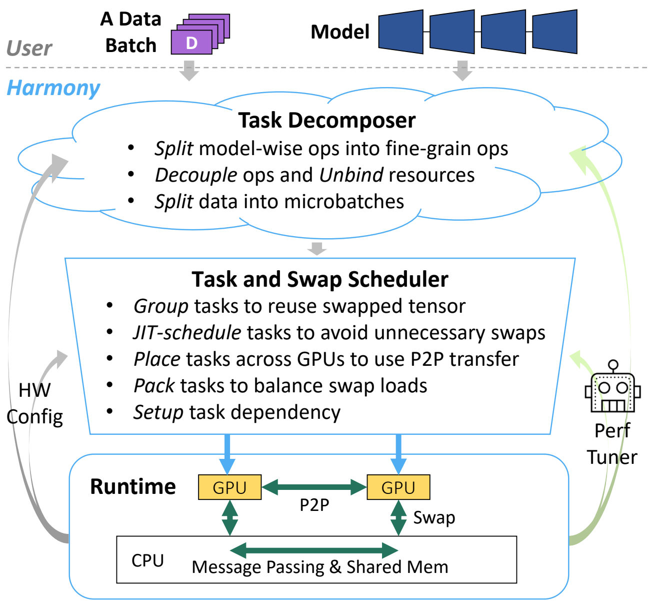

User
A Data Batch
D
Model
Harmony
Task Decomposer
Split model-wise ops into fine-grain ops
Decouple ops and Unbind resources
Split data into microbatches
Task and Swap Scheduler
Group tasks to reuse swapped tensor
JIT-schedule tasks to avoid unnecessary swaps
Place tasks across GPUs to use P2P transfer
Pack tasks to balance swap loads
Setup task dependency
HW Config
Perf Tuner
Runtime
GPU
P2P
GPU
Swap
CPU
Message Passing & Shared Mem

Figure 3: High-level overview of Harmony.

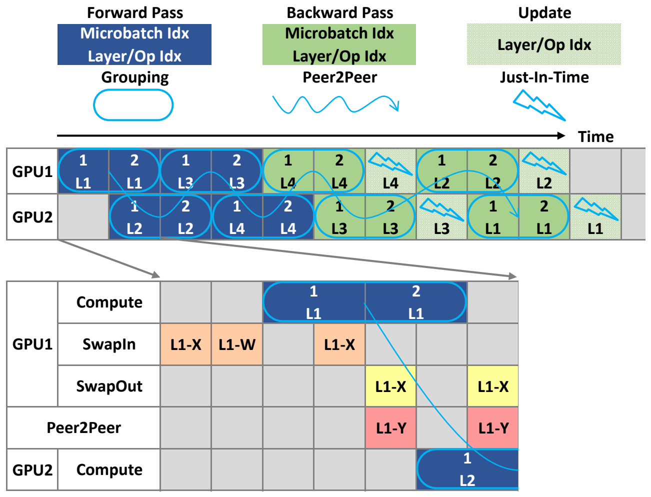


Figure 4: A simplified example of training a four-layer "large" model on two GPUs with virtualized pipeline parallelism in Harmony (assumes layer-level granularity and layer runtimes are uniform).

memory: CPU and shared PCIe links with 4:1 or 8:1 oversubscription [2, 36, 45, 47, 51] (e.g., Fig. 2(b)). Furthermore, as each GPU is swapping a similar amount of state, the swap overhead grows linearly with the number of GPUs.

4. Unbalanced Swaps. In pipeline-parallel training, pipeline stages are designed to be compute-load balanced, but pipelining schemes inherently have imbalanced memory requirements across pipeline stages: the head of the pipeline must stash more activations compared to the tail of the pipeline [42, 43]. Lacking this context, and operating in isolation on individual GPUs, naively using GPU memory virtualization when training large models can result in swap imbalance across stages thus exposing bottleneck stages with greater swap overheads (Fig. 2(c)).

3 TRAINING IN HARMONY

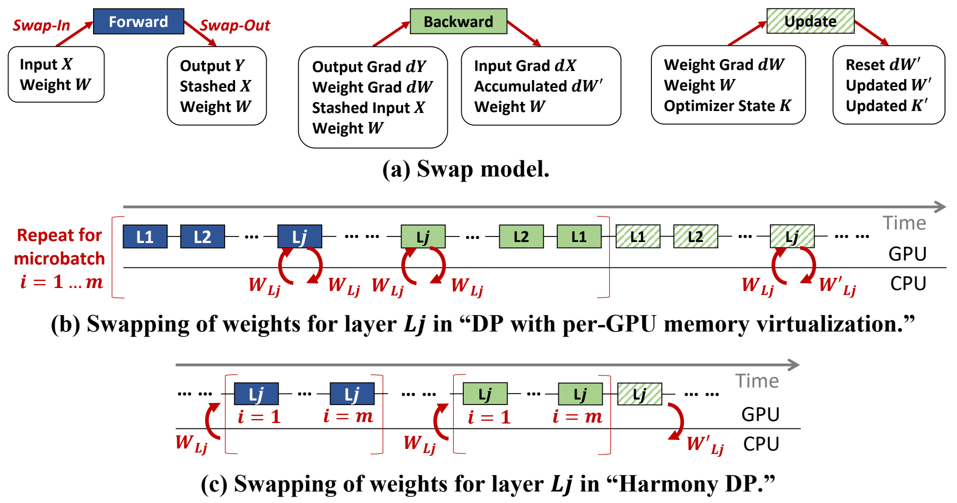
Fig. 3 shows a high-level overview of Harmony. Users provide Harmony with training data and their model (written in imperative-style PyTorch [49], as if running sequentially on a single device). Harmony extracts the model's operator- or layer-granularity graph, and further refines it to decouple forward, backward, and weight update for each layer using dependencies encoded in the task graph. These fine-grained tasks are the unit of scheduling in Harmony; a concrete task instance is tied to a specific micro-batch of input data.

With a user-specified parallelization scheme, Harmony's scheduler binds tasks to devices, appropriately moving required inputs (activations, gradients, stashed tensors, weights, optimizer states, etc.). This *swapping in* of input data and state, coordinated by Harmony, may either be from host (CPU) to device (GPU) memory or directly between device memories; it is also responsible for *swapping out* tensors from device to host memory based on their usage status and memory pressure. Harmony's memory manager maintains a state machine tracking the lifetime of all tensors used.

Guided by the four principles in Section 1 (minimize memory swaps, schedule just-in-time, swap over fast inter-device links, balance load across devices), Harmony implements four optimizations to enable high-performance training:

1. Input-batch grouping allows a scheduled task to execute across different input batches back-to-back; the tasks' state (e.g., weight or gradient buffer) can stay in memory and be reused across multiple input batches/tensors. Grouping M inputs for a task (each input-batch saturates GPU memory) reduces what would otherwise have been M repeated swaps of the state for each batch into a single swap. Fig. 4 shows a toy example of training a large model using pipeline-parallelism over two GPUs in Harmony, where each layer-level task executes on a group of two microbatches back-to-back before moving to the next task. Unlike traditional pipeline stages [20, 42, 43] which execute all layers in the stage one batch at a time, in Harmony the forward pass of layer-1 runs through 2 input batches without swapping out its weights, and backward pass of layer-1 computes gradient of 2 batches without swapping out its gradient buffer.

2. Just-in-time scheduling executes a task immediately when 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. For the example in Fig. 4, jit-scheduling brings the update task of each layer closer to its backward pass so that



(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 5: Tensors that need to be swapped in and out for forward, backward, and weight update phases of training.[1]

the weight and gradient tensors needed by the update tasks can be reused while they are still resident in GPU memory.
3. p2p transfers moves CPU-GPU swaps to use fast device-to-device communication, especially for those victim swaps with GPU memory virtualization that are caused by early binding of two tasks to the same GPU (with shared tensors where the output of the first is used by the second) but suffer from the shared tensors being swapped out and back in. Harmony replaces such swaps with p2p transfers by late-binding the corresponding operations with shared tensors across two accelerators, and transferring the tensor over device-to-device links. For the example in Fig. 4, all input and output tensors of each layer are transferred directly between the two GPUs; in contrast, with naive GPU memory virtualization these tensors would incur CPU-GPU swaps.
4. Task packing packs together multiple operations along with their assigned data for balancing the load (of compute, memory, and swap) caused by different operations and input data size. Harmony then schedules the resulting packed tasks across accelerators for system-wide load balancing.

Harmony's *Performance Tuner* profiles the *runtime* performance; these profiles are used by the *Task Decomposer* and *Task and Swap Scheduler* to tune task combinations and for better scheduling. For example, a reinforcement learning agent can be used for such online tuning.

Harmony supports two different modes of parallel training: data- [37] and pipeline-parallel training [20, 42, 43], while offering users the illusion of running on a single virtual device with practically unbounded memory. We denote these modes of parallelism as Harmony-DP and Harmony-PP, respectively. Input-batch grouping results in Harmony-PP using a novel pipeline schedule compared to prior works.

[1]The running-state tensors, e.g., mean and std, are omitted for simplicity.

Analytical comparison. We perform an analytical comparison of these schemes with their corresponding baselines that use per-GPU memory virtualization. To simplify the explanation, we assume (without loss of generality) a setup with homogeneous GPUs where each GPU's memory capacity permits it to only hold one layer-level operation on 1 micro-batch at any time. We also assume the use of layer-granularity task graphs and 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.

We also model the swap volume for different types of operations (tasks) [3, 7, 30, 34, 58, 59] where they each need to swap-in certain inputs and swap out certain outputs (Fig. 5(a)). In comparing Harmony-DP, Harmony-PP, and their corresponding baselines with per-GPU memory virtualization, we find that the Harmony variants significantly outperform their baseline counterparts by reducing swap overhead.

Here, we focus on an example of one 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 micro-batches per GPU and N GPUs (for a mini-batch of mN microbatches). Fig. 5(b) shows that, for a single iteration (mini-batch), when using DP with per-GPU memory virtualization *each GPU* has to swap-in and swap-out W 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 in and out W once for weight update. This results in an overall swap volume of $(4m + 2)N|W|$ per iteration. In contrast, in Harmony-DP (Fig. 5(c)) each GPU has to swap in W only once each for the forward and the backward passes *across all m microbatches* (due to *input-batch grouping*), and swap out W once for weight update (due to *jit-scheduling*), resulting in an overall swap volume of $3N|W|$ per iteration. Finally, Harmony-PP (Fig. 4) brings the overall per-iteration swap volume down to $3|W|$ (across all m microbatches and all N GPUs)!

For brevity, we omit the complete analytical model that covers all tensors shown in Fig. 5(a); suffice to say, Harmony offers swap load reduction for all tensors and Harmony-PP dominates savings compared to all other baselines.

4 DISCUSSION

So far we have highlighted the key principles and techniques of Harmony. However, interesting challenges remain and merit future research.

Feasibility of end-to-end training. While Harmony enables training of large models on modest deployments, training certain massive models end-to-end might be infeasible. For example, pre-training GPT-3 from scratch required 314 ZettaFLOPs (3.14×10^{23} FLOPs) [5], resulting in several months of training even with thousands of cutting-edge GPUs [44]; pre-training such a model on tens of GPUs will result in unrealistically long times (years). There is no denying that training on larger clusters will naturally result in speedier training of such models. However, despite this limitation, we believe that Harmony can still enable the development and debugging of such models on modest deployments (before they are deployed for pre-training at a larger scale), and for fine-tuning of such large models which requires less than 10s of exaFLOPs (10^{19}) [4, 11, 29] clocking in at days with modest small-scale deployments [21].

Multi-machine training. Our prototype of Harmony operates on single-server deployments. However, the core ideas of task decomposition, late binding, optimizations (input-batch grouping, jit-scheduling, p2p transfers, task packing), and parallel training schedules (Harmony-DP and Harmony-PP) all extend to multi-server deployments. Specifically, Harmony's Task and Swap Scheduler will have to operate as a distributed scheduler responsible for scheduling tasks across servers and Harmony's Runtime implementations will have to take into account heterogeneous and hierarchical interconnects (PCIe, NVLink versus Ethernet, Infiniband). If the aggregate memory across all GPUs is large enough to accommodate the memory footprint of large models, swapping becomes irrelevant and pipeline parallel training becomes an attractive solution [20, 42, 43]. In this case, the abstractions and optimization techniques in Harmony would enable model developers to separate the definition of their model from a particular parallelism strategy, and make it easier to experiment with different strategies. In our experience with cutting-edge models like sparsely-gated mixtures of experts [35], training scripts are tightly coupled with a particular parallelism strategy, switching strategies entails a broad rewrite, and manually invoking collective operations (AllToAll, AllReduce, etc.) is prone to deadlock.

The memory–performance tango. A Harmony task packs multiple operations executing on a microbatch of input (e.g., forward, backward, or weight update on a contiguous sequence of layers). Consequently, both the pack size of a task and the microbatch size determine the memory footprint and performance when executing the task. Given a fixed memory capacity, increasing the pack size can reduce p2p transfer and swap volume (when using recompute [7]), but the task can only operate on a small microbatch. In contrast, shrinking the pack size and operating on a large microbatch can increase accelerator utilization and arithmetic intensity. Prior works on model- and pipeline-parallel training [20, 42] fix one or both of these parameters either using heuristics or by punting the problem to model developers. Furthermore, not all classes of operations have uniform runtimes or memory footprint; e.g., a fixed pack of layers can have $2-3\times$ the runtime and memory footprint in the backward pass compared to the forward pass, thus motivating the need for different pack and microbatch sizes across these passes. With a user specified mini-batch size, the multi-dimensional problem of algorithmically determining the optimal task granularity and the size of microbatches they operate on is an open one.

Swap scheduling in Harmony poses another interesting memory-performance trade-off. Harmony can mitigate swap overheads by prefetching and overlapping data copies for a microbatch with compute for another microbatch (e.g., L1-X for the second microbatch in Figure 4), but this requires a form of double buffering. Harmony can instead forgo such a memory overhead and incur swap overheads in the critical path, but in doing so can support larger layer packs or microbatches. It is unclear which of these is a better option at first glace and algorithmically reasoning about such memory-performance trade-offs mandates future research.

5 CONCLUSION

DNN model size growth over the years has brought us to a point where only the elite who have access to massive computing resources can develop and train them. One of the main challenges for the masses in training these models on modest multi-GPU deployments is limited GPU memory capacity compared to model size. 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 large models efficiently on such modest deployments. We are excited to harmoniously explore research directions opened up by our proposal.

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/HelpDesk_Manual.

[3] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* (2019).

[4] 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

[5] 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* arXiv/2005.14165 (2020).

[6] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. 2003. *Computer systems: a programmer's perspective*. Vol. 2. Prentice Hall Upper Saddle River.

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

[8] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. 2012. Pipelined back-propagation for context-dependent deep neural networks. In *Thirteenth Annual Conference of the International Speech Communication Association (INTERSPEECH'12)*. Portland, USA.

[9] 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).

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* (Jan. 2008), 107–113.

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

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv* arXiv/2010.11929 (2020).

[13] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA.

[14] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. *arXiv* arXiv/1406.2661 (2014).

[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.

[17] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[18] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR'18)*. Salt Lake City, Utah.

[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.

[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 (NIPS'19)*. Vancouver, Canada.

[21] Hugging Face. 2021. Transformer Examples, https://huggingface.co/transformers/v2.3.0/examples.html.

[22] Zhouyuan Huo, Bin Gu, Heng Huang, et al. 2018. Decoupled parallel backpropagation with convergence guarantee. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*. Stockholm, Sweden.

[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] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd European Conference on Computer Systems (EuroSys'07)*. Lisbon, Portugal.

[26] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*. San Diego,CA.

[27] 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.

[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)* 15, 3 (2018), 1–26.

[29] 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* arXiv/2001.08361 (2020).

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv* arXiv/1412.6980 (2014).

[31] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv* 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 (NIPS'12)*. Lake Tahoe, NV.

[33] Paras Lakhani. 2020. The importance of image resolution in building deep learning models for medical imaging. *Radiology: Artificial Intelligence* 2, 1 (2020), e190177.

[34] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[35] 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)*.

[36] 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.

[37] 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* arXiv/2006.15704 (2020).

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

[39] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv* arXiv/1708.02182 (2017).

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

[41] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA.

[42] 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.

[43] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2020. Memory-efficient pipeline-parallel DNN training. *arXiv* arXiv/2006.09503 (2020).

[44] 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* arXiv/2104.04473 (2021).

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

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

[47] 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.

[48] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*. Farminton, Pennsylvania.

[49] 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 (NIPS'19)*. Vancouver, Canada.

[50] 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.

[51] 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.

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

[53] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv* arXiv/1910.10683 (2019).

[54] 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).

[55] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence (AAAI'19)*. Honolulu, Hawaii.

[56] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, and Dong Li. 2021. Sentinel: Runtime Data Management on Heterogeneous Main MemorySystems for Deep Learning. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*.

[57] 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* arXiv/2101.06840 (2021).

[58] 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.

[59] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[60] Carl F Sabottke and Bradley M Spieler. 2020. The effect of image resolution on deep learning in radiography. *Radiology: Artificial Intelligence* 2, 1 (2020), e190015.

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

[62] 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* arXiv/1909.08053 (2019).

[63] 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* arXiv/1904.04514 (2019).

[64] 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.

[65] 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* arXiv/1609.08144 (2016).

[66] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*. Paris, France.

[67] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets.. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. Boston, MA.

[68] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. 2019. Efficient memory management for gpu-based deep learning systems. *arXiv* arXiv/1903.06631 (2019).