Hydra: An Optimized Data System for Large Multi-Model Deep Learning

Kabir Nagrecha
knagrech@ucsd.edu
University of California, San Diego

Arun Kumar
arunkk@eng.ucsd.edu
University of California, San Diego

ABSTRACT

In many deep learning (DL) applications, the desire for ever higher accuracy and the new ubiquity of transfer learning has led to a marked increase in the size and depth of model architectures. Thus, the memory capacity of GPUs is often a bottleneck for DL practitioners. Existing techniques that rely on partitioning the model architecture across a network of GPUs suffer from substantial underutilization and busy waiting due to sequential dependencies in most large-scale model architectures (Transformers, CNNs). We observe that almost all such prior large-model systems focus on training only one model at a time, but in reality DL practitioners often train many models in bulk due to *model selection* needs, e.g., hyper-parameter tuning, architecture finetuning, etc. This gap leads to significant system inefficiency. We approach this problem from first principles and propose a new information system architecture for scalable multi-model training that adapts and blends ideas from classical RDBMS design with task parallelism from the ML world. We propose a suite of techniques to optimize system efficiency holistically, including a highly general parameter-spilling design that enables large models to be trained even with a single GPU, a novel multi-query optimization scheme that blends model execution schedules efficiently and maximizes GPU utilization, and a double buffering idea to hide latency. We prototype our ideas on top of PyTorch to build a system we call Hydra. Experiments with real benchmark large-scale multi-model DL workloads show that Hydra is over 7x faster than regular model parallelism and 1.8-4.5X faster than state-of-the-art industrial tools for large-scale model training.

PVLDB Reference Format:
Kabir Nagrecha and Arun Kumar. Hydra: An Optimized Data System for Large Multi-Model Deep Learning. PVLDB, 11(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/knagrecha/hydra.

1 INTRODUCTION

Deep learning (DL) has revolutionized predictive analytics on unstructured data. Its high-profile successes at Web giants has led to growing adoption of state-of-the-art DL at more Web firms, enterprises, domain sciences, and even digital humanities. In recent years, many DL practitioners and researchers have found that using larger and deeper model architectures has led to improved model quality [6, 9, 19, 43]. Natural language processing (NLP) is now awash with multi-billion parameter Transformer architectures such as BERT-Large [9], GPT-3 [7], and Megatron-LM [50]. Interest in such large models is also growing in computer vision (e.g., [10]) and for tasks bridging relational data and NLP [58]. Transfer learning by fine-tuning of pre-trained models from highly popular model hubs such as HuggingFace [56] means many applications now get state-of-the-art accuracy with large DL models even without large application-specific training data. As such, model size is the main bottleneck for many users, rather than data size. *Unfortunately, GPU memory capacity has been trailing behind DL model sizes, creating a new systems bottleneck for DL users* [51].

The problem is only exacerbated by the challenges of model selection. Most DL model building jobs must evaluate multiple possible training configurations to control the tradeoff between over-fitting and under-fitting [38, 48]. Introducing this multi-task aspect to the problem dramatically increases the computational requirements of training even further beyond what large-scale models already demand.

Example. Consider the example of a political scientist building a text classifier for sentiment analysis on Twitter posts (tweets) to understand polarization between gun rights and gun control supporters. Their dataset is a few GB in size, much like the majority of DL training jobs according to recent polls [1]. So, a single-node multi-GPU setting suffices for their task. They download a few different state-of-the-art BERT models from HuggingFace [56] and prepare to fine-tune them. They aim tune relevant hyperparameters of all models to help raise accuracy. However, their GPUs do not have enough memory to even hold one model, and execution causes PyTorch or TensorFlow to crash and impede their application.

There are two key issues here. First, the GPU memory bottleneck prevents users from fully exploring the possibilities of state-of-the-art DL models. Even for cloud users, who are not bound by their own local resources, this limitation forces them to spend more money on renting more expensive machines with larger GPUs. Second, the multi-task nature of the model building process amplifies the hassle of grappling with large models many times over, leading to more costs and DL user frustration.

System Desiderata. We have the following key desiderata to support a large-scale multi-model training system.

- **Out-of-the-box Model Scalability.** We desire an approach that can easily scale to very large models even if the user

Proceedings of the VLDB Endowment, Vol. 11, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

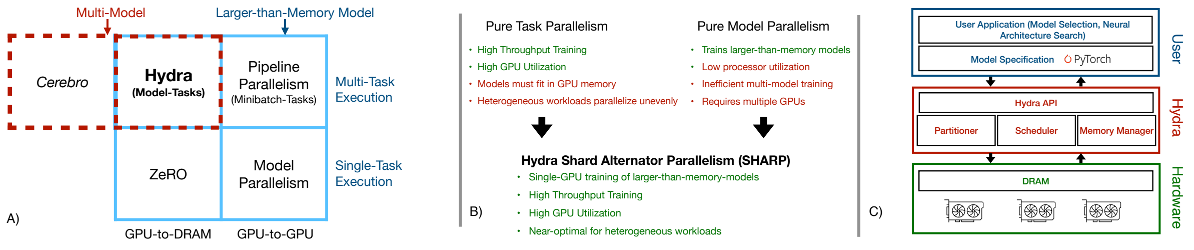

Figure 1: A) Hydra introduces the first known hybrid of model- and task- parallelism. Hydra trains multiple models (like Cerebro [38]), exploits task parallelism more effectively than pipelining, and expands the GPU-to-DRAM offloading optimizations of ZeRO and DeepSpeed [44, 46]. B) Hydra combines the benefits of model- and task- parallelism, hybridizing them to eliminate the weaknesses of both and offer their added strengths. C) Hydra in the context of user-interactions and hardware.

has only one GPU. It should not force users to get multiple GPUs but nor should it require them to make special manual efforts to adjust their model or manually handle resources when they do have multiple GPUs.

- **High Throughput.** Given the ubiquity of needing to train multiple models, we desire an approach that parallelizes execution to achieve higher overall throughput across a model training job's lifecycle.
- **Resource Efficiency.** We desire a system that is aware of its hardware environment and maximizes the utility of available GPUs. Not only do we wish to avoid *wasting* resources, but we also aim to exploit the full *potential* of what resources we do have.
- **No Effect on Accuracy.** We desire a system that does not directly affect the data being processed or modify the execution patterns of models. Actually altering the model's training procedure tends to cause accuracy degradation, something that must be avoided if we are to offer a seamless training experience for DL practitioners.

We do *not* focus on data scalability, nor are we focused on scaling across multi-node clusters. We are not in the setting of scaling training to 1000 GPUs or PB-sized datasets on massive clusters like the Googles, NVIDIAs, and OpenAIs of the world. We aim to democratize large DL models to regular users in the domain sciences, enterprises, small Web firms, etc. From our own conversations with such data scientists at UCSD and at some companies, we note that a majority operate on single-node multi-GPU settings. It suffices for their data scale and avoids the hassle of operating multi-node clusters. Thus, in this paper, we focus on studying the single-node multi-GPU setting in depth for Hydra. We leave it to future work to extend our techniques to multi-node clusters, say, by integrating it with Cerebro[25, 38] or DeepSpeed [42, 44, 46].

Limitations of Existing Landscape. There has been a flurry of recent work in the ML systems world aimed at addressing some of these issues. However, each fails on one or another of our desiderata.

1) False Dichotomy of Task- and Model- Parallelism. Existing systems do not address the problems of multi-model training and large-scale model training at once — they choose to focus on either one or the other. Current multi-model systems cannot train models that do not fit into GPU memory [24, 25, 27, 28, 38], and existing larger-than-GPU-memory model techniques [4, 16, 18, 19, 23, 32, 44, 46] do not consider the possibility of multi-model execution. This is problematic, as these systems' inability to consider all aspects of their workloads prevents them from optimizing the system holistically to improve model training throughput and maximize resource efficiency. Figure 1A) illustrates a few critical examples of this, as well as the gap our work fills.

2) Low Scalability & Resource Efficiency. Most techniques aiming to support the training of models larger than the memory capacity of a single GPU use "model parallelism" [4] as a starting point. Unfortunately, model parallelism offers poor scalability, and tends to suffer from low resource utilization due to sequential dependencies in model architectures that prevent parallel device execution. Systems that build off of model parallelism generally inherit these weaknesses to a greater or lesser degree. We elaborate on these tradeoffs further in Section 2.2.

3) Inability to Train Larger-than-GPU-Memory Models. Considering instead systems that are built specifically for multi-model execution and model selection, none actually support larger-than-GPU-memory model training [24, 25, 27, 28, 38]. All focus on the challenge of processing a large number of model configurations in parallel, an approach known as *task parallelism.* As such, attempting to train larger-than-GPU-memory models with such systems will generally lead to crashes and other undesirable behaviors. We elaborate on these systems further in Section 2.2.

Overall, we observe two issues with prior art. First, *they conflate scalability with parallelism.* Parallelism is not a fix for scalability but complementary. One must address the scalability bottleneck from first principles. Second, *they all fail to recognize or exploit a key source of higher degree of parallelism in DL workloads: training multiple models in one go*, e.g., during model selection such as hyperparameter tuning or neural architecture engineering [24]. By focusing on this new aim of training multiple model tasks, we introduce a new design motivator that has not been present on prior works.

Our Approach. We present Hydra, a new system for large-scale multi-model DL training that optimizes workloads holistically. Hydra is easy-to-use, exposing only higher level APIs to shield DL users from having to manually optimize execution. Figure 1(C) illustrates our high-level architecture and its placement in between the user-facing APIs and the hardware (more details in Section 3). The user provides a set of model specifications in a popular DL tool (we focus on PyTorch), as shown in Figure 4, but beyond this initial step, all training procedures are managed and executed by Hydra. Our target setting is single nodes with multiple GPUs, a common setting in NLP where pretrained models tend to be very large, but fine-tuning datasets tend to be relatively small [17]. We focus on supporting workloads with several large-scale sequentially defined models (e.g. Transformers) that are too large to fit into the memory of a single GPU, but sufficiently small to fit into DRAM.

Techniques in Hydra. Hydra achieves our previously described desiderata with a suite of data systems techniques, some novel and some old (inspired by RDBMSs), albeit repurposed to DL systems. But our main novelty is also in how we assemble this "right" set of techniques and adapt them to build a fully automated system for large multi-model DL workloads. At the core of Hydra is a novel hybrid form of parallel DL execution that *blends task parallelism and model parallelism* to improve overall resource efficiency in multi-GPU multi-model cases. Figure 1B) positions our hybrid parallelism approach, which we call Shard Alternator Parallelism (SHARP). Basically, model parallelism's main con is that it keeps only one GPU busy at a time, while task parallelism's main con is that it requires a model to fully fit in a GPU's memory. SHARP obviates both these issues. While Section 4 will present the details of how SHARP works, our intuition is to "break multiple models down and put them back together in a blended, better way."

In order to support SHARP, we design several other components to generalize model parallelism into a more flexible and more efficient form.

First, we devise a *model spilling* technique, an analogue to "data spilling" in RDBMSs where parts of the data are put at a lower level of the memory hierarchy. We blend that with model parallelism's notion of model shards by breaking up a large model, only loading some model shards onto GPUs, while the rest sit in DRAM. This is akin to sharding a large table in an RDBMS and loading only the active shard from disk to buffer manager. Model spilling provides a great deal of flexibility in execution management and scheduling for SHARP.

Next, we fully *automate the partitioning* of all models to respect GPU memory constraints with a lightweight and highly general approach.

Third, we adopt the classic RDBMS trick of *double buffering* to reduce the latency in between execution of model shards on a device. It overlaps GPU computation with loading from DRAM and works in lockstep with SHARP.

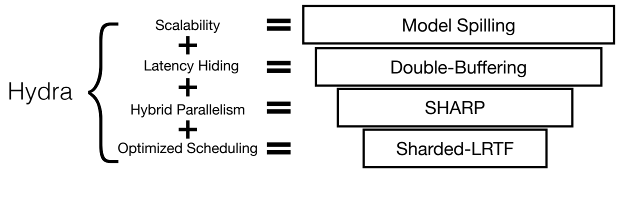

Figure 2: Hydra's layered optimization stack combining scalability (spilling), hybrid parallelism (SHARP), efficient scheduling (Sharded-LRTF), and latency hiding (double buffering).

Finally, we put all these techniques together to formulate a formal scheduling problem for our setting. We propose a simple greedy algorithm we call *Sharded-Longest-Remaining-Time-First (Sharded-LRTF)* to tackle it. We show using simulations that Sharded-LRTF offers near-optimal results for both homogeneous and heterogeneous sets of models. Figure 2 shows how these optimizations build on top of one another.

We prototype all of our ideas on top of PyTorch to create Hydra. We evaluate it empirically on two key large-model benchmark workloads and datasets: hyperparameter tuning for BERT-Large on the WikiText-2 [33] dataset and neural architecture evaluation for Vision Transformer [10] on the CIFAR-10 dataset. Hydra substantially outperforms all prior approaches, yielding near-linear speedups on an 8-GPU machine for both workloads. In particular, it offers almost 7.5x speedups over regular model parallelism, with a substantial speedup over pipeline parallelism. Hydra also reports the highest GPU utilization. We then dive deeper into the behavior of Hydra by varying model scales, number of models trained together, and number of GPUs. We also report an ablation study showing the impact of our two key optimization techniques: SHARP and double buffering.

In summary, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to study the union of scalability and parallelism from first principles for multi-model training of very large sequential DL models. Our current focus is single-node multi-GPU settings.
- Inspired by RDBMSs, we present a suite of scaling and efficiency techniques combining automated model partitioning, model shard spilling, and double buffering.
- We devise a novel hybrid form of DL execution called SHARP combining task parallelism and model parallelism that mitigates the major cons of both.
- We cast our multi-model shared training as a form of multi-query optimization and build a simple scheduler featuring an efficient greedy algorithm called Sharded-LRTF.
- We implement all our ideas in a system we call Hydra. A thorough empirical evaluation with real multi-model large DL workloads shows that Hydra substantially outperforms prior state-of-the-art open source and industrial systems.

2 BACKGROUND AND PRELIMINARIES

We start with some background on key technical concepts and notation needed for the rest of this paper.

2.1 ML Terms

Basic Definitions. In this paper, we use the term *model* to refer to a neural computational graph (also called the *neural architecture*) and its parameters. These are specified and trained using popular DL tools such as PyTorch or TensorFlow. The *training* of a model involves an optimization procedure called stochastic gradient descent (SGD). It samples *mini-batches* of data from the training dataset, performs updates to the parameters of the model, and repeats this for all mini-batches. A full pass through the dataset is called an *epoch*. Training a model requires many epochs of SGD. Popular DL tools implement many variants of SGD (e.g., Adam, AdaGrad, RMSProp, etc.) but their data access patterns are identical. The size of a DL model can be measured through the number of parameters and/or its memory footprint. Updates to the parameters of a DL model involve two passes: *forward*, which transforms input (data features) to output (target) with a series of computations, and *backward*, which transforms a loss calculated with the label back into updates on parameters through a reverse series of computations. The backward pass is also called *backpropagation*.

Model Shards. Most DL models can be seen as a series of groups of tensor operators, also called "layers" [13]. Such models are also called "feedforward" architectures. Given such a layout, we can sub-divide a model's layers into contiguous subsets of layers; each subset is known as a *model shard*. Thus, a model can be partitioned into a series of non-overlapping shards.

2.2 Comparing Prior Art Paradigms

Model-Parallelism. Model parallelism is one of the most popular techniques for large-scale model training. It partitions a model into "shards" placed across *multiple* devices (GPUs). Such partitioning is typically done between layers (though some designs choose to partition layers themselves). During execution, intermediates between model shards are exchanged between GPUs to emulate regular training as if on a single device. Alas, the majority of DL models (especially large models) have *sequential dependencies* inside their architecture, which means only one shard (and only one device) is fully active at a time. This leads to massive GPU under-utilization and no speedups from adding GPUs. Besides, it forces users to get multiple GPUs in the first place, which may not be available or not affordable (even in pay-as-you-go clouds) for many kinds of DL users. This issue is particularly damaging in multi-model execution cases. However many GPUs are required for a single instance, the demands will be scaled up linearly with the number of models we wish to train in parallel.

Model-Parallelism Hybrids/Extensions. All systems that build on top of model parallelism [16, 19, 23], or hybridize with it [23, 44, 46] inherit these issues to a greater or lesser degree. Pipeline parallelism, the state-of-the-art in this domain, builds on top of model parallelism by pipelining mini-batches through the shard sequence.

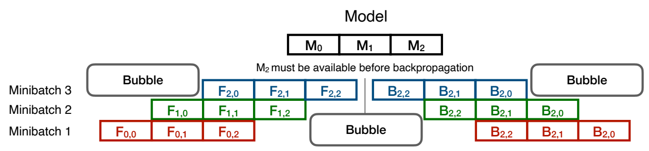

Figure 3: Illustration of pipeline parallelism of prior art on three devices. A model M is partitioned into shards M_i. A mini-batch is split into micro-batches, defined by the compute stages of each device on the mini-batch. $F_{i,j}$ is the forward pass of shard j on micro-batch i; likewise $B_{i,j}$. Note how data mini-batches are pipelined through the shards, improving resource utilization by introducing some degree of task parallelism across mini-batches. But "bubbles" where no parallelism is possible persist.

Unfortunately, it suffers from underutilization due to synchronization steps that occur between forward and backward passes, as illustrated in Figure 3. Other non-pipelined systems, such as FlexFlow[23] and ZeRO[44], use model parallelism with optimizations for data parallelism and offloading, but still suffer from underutilization during model parallel execution stages.

Swapping Tensors. Other systems ignore model parallelism altogether, working from first principles to support large-scale model training with data offloading designs [18, 32]. These data offloading systems create "swap" plans across the entire model, learning how to temporarily swap tensors between GPU memory and DRAM to reduce memory usage during memory execution. While this approach does sidestep some of the resource efficiency and scalability problems of model parallelism, it introduces new issues too. Optimizing swaps at the fine-grained level of individual tensors can be costly, and the swaps themselves introduces substantial communication latencies. Moreover, none of these frameworks are capable of exploiting multiple GPUs for parallelization or potential speedups — they only consider the single GPU case, producing low resource efficiency for multi-GPU users. We elaborate further on how our work differs from SwapAdvisor in Section 7.

Task Parallel Systems. Many systems exist for multi-model training [24, 25, 38]. Several focus on hybridizing task parallelism with *data parallelism*, a form of parallel execution wherein individual models are replicated with each instance consuming different pieces of the overall dataset with periodic parameter synchronization. None, however, address the problem of training larger-than-GPU-memory models. Due to their inability to execute such tasks, we cannot even benchmark against them! They would error out as soon as memory limits were exceeded. Cerebro [25, 38] aims to improve data parallel performance in the multi-model setting by adjusting communication patterns. Their focus is on data scalability, not model scalability, and as such their work is orthogonal to our own. Hybridizations could be considered in the future.

3 ARCHITECTURE OF HYDRA

Hydra is designed to be a lightweight wrapper around the popular DL tool PyTorch.[1] We do not need any internal code of the DL tool to be altered, which can help ease practical adoption. Figure 1 illustrates the overall architecture of Hydra and how it handles the models. There are 4 main components: *API*, *Automated Partitioner*, *Memory Manager*, and *Scheduler*. We briefly explain the role of each component.

- **API.** The user specifies a set of models to be trained using standard PyTorch APIs. Figure 4 provides an example. Note that Hydra can also scale the training of single model on a single device. The neural architectures are *fully automatically* inferred by Hydra; no custom annotations are needed. The user just needs to provide the model(s), a data loading function, and model selection specification, e.g., hyperparameter search grid or metaheuristics.
- **Automated Partitioner.** Hydra automatically ascertains the memory size(s) of the GPU(s). Then it automatically partitions the model(s) given into model shards that respect the GPU memory constraints. At a given point in time, a

[1]It is a relatively simply engineering effort to add support for TensorFlow too but we skip it in our current version for tractability.

```
task_0 = ModelTask(model_0, loss_fn, dataloader_0, lr_0, epochs_0)
task_1 = ModelTask(model_1, loss_fn, dataloader_1, lr_1, epochs_1)
orchestra = ModelOrchestrator([task_0, task_1])
orchestra.train_models()
```

Figure 4: Example usage of the API of HYDRA.

GPU runs computations for only only one model shard. Section 4.3 explains our sharding process in more detail.

- **Memory Manager.** After the model shards are constructed, HYDRA puts them all in the machine's DRAM. It then moves shards up the memory hierarchy into the GPU based on the schedule produced by the *Scheduler*. When a shard's computation is completed, it is moved back to DRAM. Intermediate outputs within a model across shards are also written to DRAM by this component. All of this happens transparently to the DL user. This is the crux of how HYDRA achieves seamless scalability to very large models. Section 4.2 explains this "model spilling" technique in more detail.
- **Scheduler.** This component is the core orchestrator of what shard gets placed on what GPU and when. It uses SHARP, our novel hybrid of task- and model parallelism. It ensures that the shards of a given model are scheduled in a way that respects the *sequential dependency* inherent in the DL model's forward pass and backward pass. We formalized this scheduling problem as an MILP, compared a few alternative scheduling algorithms, and devised a simple new algorithm that best suits our system setting. We also devised a buffering technique to further raise resource utilization. Sections 4.3–4.5 explain our ideas in more detail.

4 TECHNIQUES IN HYDRA

We now dive into the techniques in HYDRA to achieve seamless scalability and resource-efficient parallelism for training multiple large DL models in one go. Our techniques are inspired by a suite of classical ideas in RDBMSs, viz., spilling, sharding, multi-query optimization, and double buffering [45, 47], but our work is among the first to study them in the context of DL training. While the individual techniques may not be highly novel in the context of data management systems, the way we identify the right set of techniques, adapt them for DL, and synthesize them in HYDRA is novel. This enables HYDRA to offer state-of-the-art results in this important DL systems setting.

4.1 Intuition, Motivating Challenges, and Technical Novelty

HYDRA's core optimization is based on a relatively simple insight. Model parallelism's dependencies are too rigid to utilize resources efficiently, and task parallelism is too coarse-grained with its uninterrupted execution requirement. Blending the two together solves both problems. Task parallelism allows us to sidestep dependencies in model parallelism by using blocked or idle devices to train alternate tasks, similar to the difference between sleeping and busy waiting in CPU processor design [11]. Meanwhile, model parallelism allows us to break down previously atomic training tasks into "chunks", based on shards, and then reassemble shards of different models in a more efficient way. This is the same idea that underpins the speedups seen in prior hybrid parallel works [30, 38].

To exploit this intuition, our system must demonstrate two key characteristics.

1) Shard Movement Flexibility. For our model shards to be taken apart and put back together again effectively, we must introduce a degree of flexibility that is not present in model parallelism. Model parallel shards *cannot* be moved around between devices, nor can they be offloaded or delayed. But if we are to blend schedules across models, we must support the possibility of shards being temporarily put off or shifted. To solve this problem, we introduce *model spilling*, wherein we demote and promote shards of the model between GPU memory and DRAM. We then layer on communication optimizations with *double-buffering*. Note that this concept is *not novel*, spilling is an age-old idea from RDBMSs. Offloading has also been explored before [18, 32]. It is our generalization of offloading tensors into spilling full sub-models (shards) and our redesign of RDBMS spilling for DL that is novel.

2) Shard Orchestration Once we obtain sufficient flexibility to orchestrate models freely, we must move and schedule them efficiently. Model schedules must be blended together to achieve optimal makespans, but the choice of shards and models at each timestep must be decided by some scheduler. We introduce our blended parallel scheme, Shard Alternator Parallelism (SHARP), and a greedy scheduler, Sharded-Longest-Remaining-Time-First (Sharded-LRTF). The basic concept of scheduling across multiple different tasks is not inherently novel — prior work in multi-query optimization has explored this for the RDBMS setting, and hybrid parallel works[30, 38] have explored it in the context of data parallelism and task parallelism. However, this is the first work to expand this concept to the model-task hybrid setting, and the first to devise a scheduler for this problem.

4.2 Model Spilling

In traditional model parallelism, a model is divided into shards; each shard is placed on a different (GPU) device. But all shards must be loaded across multiple GPUs for a forward/backward pass to work at all. We observe that this is an overkill: due to the sequential dependency across layers inherent in DL models, only one device is typically "active" with computation at any point in time. The other devices are merely repositories for inactive model shards.

Exploiting the above observation, we use a simple idea in HYDRA to avoid making all shards active: *spill to DRAM.* Only an active shard is promoted to a GPU's memory, while the rest "wait" in DRAM. This is akin to sharding a large table and staging reads between disk and DRAM in RDBMSs, except we focus on a higher level in the memory hierarchy and apply it to a large model instead. All this means HYDRA scales to arbitrarily large models *on a single device.* So, even a trillion-parameter DL model can now be trained on a single GPU out of the box, given sufficient DRAM. This can already makes a qualitative difference for DL users with limited resources.

Model spilling is a direct reimagining of model parallelism, serving as a substitute rather than a complement. It retains the notion of shards and chunks of layers and does not alter the execution

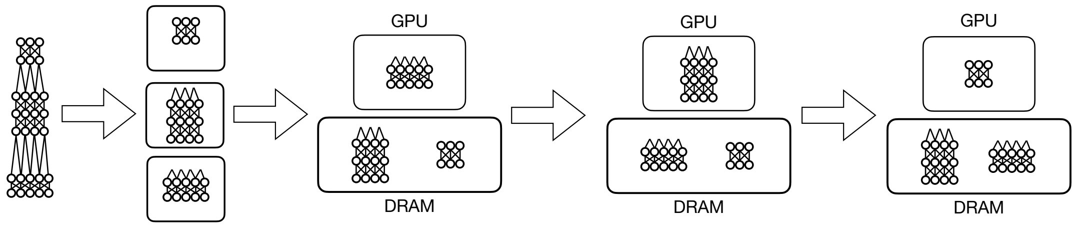

Figure 5: Illustration of model spilling as a temporal schematic. Hydra places inactive shards at a lower level of the memory hierarchy (DRAM here), from which they are re-activated later.

pattern, unlike other memory offload systems [32, 44, 46]. A natural analogy might be busy waiting versus blocking in CPU execution. In model parallelism, unused shards still block the CPU, busy-waiting in place while their dependencies are resolved. SHARP sends these unused shards "to sleep", allowing processors to work on other models in the meantime. Spilling can be used as a direct replacement of existing model-parallel setups. The only downside is that model spilling introduces GPU-DRAM interactions, which are slower than the GPU-GPU interactions of model parallelism. The latency costs can be mitigated or even eliminated, however, as shown in Sec 4.6.

The use of model spilling, or indeed any sharded execution strategy, poses two open questions: Who does the sharding and how? How to efficiently train multiple models together with sharding in a multi-GPU setup? Our next two techniques tackle precisely these questions.

4.3 Automated Model Partitioning

Both traditional model parallelism and our model spilling depend on some sort of "cut point" to split the neural computational graph because shards must consist of disjoint subsets of layer groups. Prior art [41] uses some basic heuristics, albeit restricted to a specific class of models. Unfortunately, their approach is not general enough for our purpose. While one could use sophisticated graph partitioning algorithms for "optimal" partitioning, we find that is not worthwhile for two reasons. First, this stage is anyway only a tiny part of the overall runtime, which is dominated by the actual training runs. Second, due to the marginal utility of over-optimizing here, it will just make system engineering needlessly complex.

We prefer simplicity that still offers generality and good efficiency. Thus, we use a *dynamic greedy approach* based on toy "pilot runs." Algorithm 1 presents it succinctly. The basic idea is to pack as much of a model as possible on to a GPU. If the set of GPUs is heterogeneous, we use the smallest-memory GPU to ensure cross-device compatibility of shards. We treat a DL model as an ordered list of layer indices, with the layers being "cut-points" in the graph to enable smooth partitioning. Hydra then iterates through these indices to run "toy" passes with a *single mini-batch once.* If the run is successful, the Partitioner raises the shard size by appending the next set of layers. If the GPU throws an out-of-memory error, we remove the set of layers appended last. Thus, in this dynamic way, we find the near-maximum set of layers that fit in GPU memory; this set is then cut off from the model as its own shard. The Partitioner continues this process for the remaining the layers, until that model is fully partitioned We record runtime statistics for later use by our Scheduler.

Algorithm 1: Dynamic model partitioning algorithm.

```
Input: Model as a sequence of layers $L$ with size $m$; data
mini-batch $B$; GPU $G$
Output: Array of partition indices $A$
Append 0 to $S$
for $i = 0$ to $m - 1$ do
  Place $L[i]$ and $B$ on $G$
  $B' \leftarrow$ Forward pass through $L[i]$ with $B$
  $T \leftarrow$ New tensor with same shape as $B'$
  Backpropagate $T$ through $L[i]$ without freeing its memory
  if $G$ out of memory then
    Append $i$ to $S$
    for $j = 0$ to $i - 1$ do
      Release all memory consumed by L[j]
      Append $i$ to $A$
    end for
  end if
end for
```

4.4 SHARP

We now present one of our key novel techniques: Shard Alternator Parallelism (SHARP), a hybrid of classical model parallelism and task parallelism. We define our basic unit of computation, *shard unit*, as follows: the subset of computations of a forward or backward pass on a model's shard. Thus, a full forward or backward pass of a model is a *sequence* of shard units.[2] Overall, the scheduling goal is to execute all shard units of all models given by the user for all epochs.

Figure 6 illustrates the basic idea of SHARP contrasted with both task- and model parallelism. After a model's shards are created (Section 4.2), shard units are naturally set. The key difference in SHARP is that a given model's shard units do not necessarily run *immediately* after one another, i.e., they may be staggered over time. This is the key reason for SHARP's higher efficiency–it breaks things down and puts them back together better.

Notice that SHARP is only possible due to the flexibility of model spilling. Model spilling's shards can be used as semi-independent sub-models, as contrasted with parameter-spilling systems where there is no real notion of independence. Only with shard-spilling

[2]In recent ML literature, this unit is also called a "microbatch" [19]. We prefer to use the more standard terminology of "unit" from the operations research and systems literatures instead because the term "microbatch" may cause confusion on whether the mini-batch *data* is split further, which is not the case. A shard unit splits the *computations* (not data) of a forward/backward pass of a whole mini-batch.

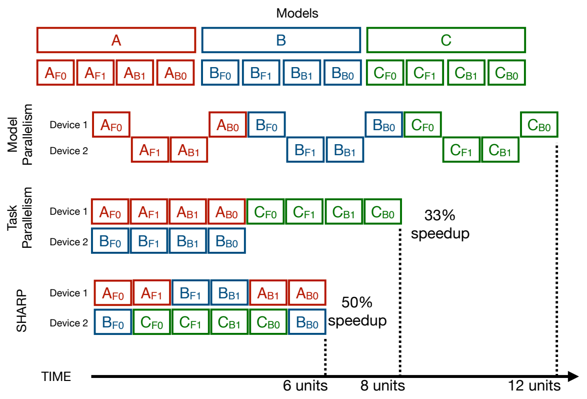

Figure 6: Demonstrative illustration of SHARP and contrasting with regular task parallelism and model parallelism for training 3 models A, B, and C, each with 2 shards. Real-world schedules tend to be more complex, but this simplified diagram shows SHARP's capacity for optimization.

can we effectively support blended execution schedules across models.

While the idea of SHARP is fundamentally simple (but novel), realizing it in a working system faces two bottlenecks: (1) the sheer number of shard units and (2) the latency of swapping shards between device memory and DRAM. First, note that the number of shard units to be handled by Hydra is multiplicative in four quantities: number of models given by the user, number of shards per model, number of mini-batches per epoch, and number of epochs per model training run. In realistic DL scenarios, one can easily hit *tens of millions of shard units*! Thus, next we answer two questions regarding the above bottlenecks: How to automate the orchestration of large numbers of shard units? Is it possible to reduce latency of swapping shard units

4.5 Automated Shard Orchestration

To realize SHARP in an system, we must handle 3 kinds of "data" before, during, and after a shard unit: (1) training data mini-batch, (2) model parameters, and (3) intermediate data/outputs of a shard unit. Thankfully, DL tools like PyTorch offer APIs that enable data to be transferred from GPU memory to DRAM and vice versa. We use those APIs in Hydra under the hood to automate shard orchestration.

Each model is defined as a "queue" of shards in DRAM, ordered according to the neural computational graph. Each shard is associated with its necessary data, such as an input mini-batch, intermediate data *between* shards, and/or gradients sent backward. The shard at the front of the queue is transferred to GPU memory along with its associated data to begin running that shard unit.

After execution completes, the shard parameters (possibly updated) are returned to DRAM. In addition, the shard's intermediate outputs, say, a gradient vector or a loss value, are also written to DRAM and attached to the model. They will be used as inputs for the model's next shard. The last shard of a model concludes a full mini-batch training pass; after that, the old mini-batch is discarded and the next mini-batch of the prepared data will be used.

4.6 Double-Buffering

A common trick used in RDBMSs, e.g., for external merge sort, is double-buffering [45]. The basic idea is this: the processor's memory (higher in the memory hierarchy) is split into two regions: one for active processing and the other as a "loading zone" for the next task. We bring this trick to the DL systems world for the first time. Hydra uses it to mask shard loading latencies. We protect a "buffer space" in GPU memory during model partitioning (Section 4.3) to guarantee that so much buffer memory will be available during training.

Prior analyses [51] demonstrate that intermediate activations produced during training form the largest proportion of GPU memory consumption, as much as 99% in some cases! Double-buffering need not transfer intermediate activations — those will be produced by checkpointing inputs between shard groups during training. Therefore, the double-buffer need only be large enough to hold the model state, optimizer state, and input data, which only forms only a small proportion of overall memory consumption. In practice, a buffer size consisting of 5% of total memory is more than sufficient, though users can adjust this value as needed.

When our Scheduler picks the next shard to be run, we transfer it to that GPU's buffer space *even as the previous shard unit is running there*. Interestingly, our double-buffered DL training in Hydra also offers a serendipitous new bonus: we can avoid spilling (to DRAM) altogether in some cases. When a model's current shard unit is active, if its next shard is double-buffered on the same GPU, intermediates need not move at all, eliminating latency. While we focus on the GPU memory-DRAM dichotomy, our above techniques are general enough to be applicable across the entire memory hierarchy: between DRAM and local disk, local and remote disk, etc.

4.7 Scheduling Formalization of SHARP

The sheer number and variable runtimes of shard units across models necessitates a rigorous automated Scheduler. We immediately face two technical challenges. First, different models may train for different numbers of epochs due to convergence-based SGD stopping criteria or early stopping in AutoML heuristics. Second, devices may disappear over time, say, due to faults, or get added, say, due to elasticity. For these reasons, we choose a *dynamic scheduling* approach to place shard units on devices as and when a device becomes available over time. This design decision tackles all three challenges above in a unified way and also simplifies system implementation.

We treat each model to be trained as a *queue of shard units* unifying reasoning of division within a mini-batch, across mini-batches within an epoch, and across epochs.

4.7.1 ***Formal Problem Statement as MILP.*** The scheduling problem is as follows. When a device (GPU) becomes available, a shard unit must be selected from one the model's queues to be placed upon that device. Shard units become *eligible* for scheduling if they have no pending dependencies, i.e., they are at the front of their queue and no other shard unit of that same model is still running on another device. The Scheduler's job is to pick a shard unit from the set of eligible shard units. Double-buffered training is already factored into this formulation: the Scheduler is actually

Table 1: Notation for our scheduling formalization.

Symbol	Description
T	List of models specified by the user to be trained
P	List of devices (GPUs) available for training.
$M_i \in \mathbb{Z}^+$	M_i is the total number of shard units for model $T_i \in T$. Note that this covers all mini-batches (and potentially epochs).
$S_i \in \mathbb{R}^{M_i}$	S_i is a variable-length list of shard unit runtimes for model T_i. The runtime of shard unit j is denoted as $S_{i,j}$.
$X_i \in \mathbb{R}^{\|P\|\times\|M_i\|}$	X_i is a variable-shape matrix of start times of shard units of model T_i across workers. The start time of shard unit j on worker p is denoted as $X_{i,p,j}$. Note that this linear ordering covers not just the model's forward and backward passes but also ordering across mini-batches (and potentially epochs).
$Y_i \in \{0,1\}^{\|P\|\times L \times L}$	L is the total number of shard units across all models, i.e., $L = \sum_i M_i$, indexed cumulatively by the index of model i and its shard unit j (denoted i_j). $Y_{p,i\_j,i'\_j'} = 1 \Leftrightarrow X_{i,p,j} < X_{i',p,j'}$.
U	An extremely large value used to enforce boolean logic.

picking shard units for double-buffering, and they get promoted from the buffer to compute.

All shard unit runtimes are given as input. Recall from Section 4.3 that the partitioner records this data during its pilot run. We now present the formal scheduling problem as an MILP. Table 1 explains our notation.

$$\texttt{Objective:} \quad \min_{X,Y} C \tag{1}$$

$$\begin{aligned}
&\texttt{Constraints:}\\
&\qquad \forall t, t' \in [1,\ldots,|T|] \qquad \forall p, p' \in P\\
&\quad (a)\ \ \forall j \in [2,\ldots,M_t]\ X_{t,p,j} \geq X_{t,p',j-1} + S_{t,j-1}\\
&\quad (b)\ \ \forall j \in [1,\ldots,M_t] \qquad \forall j' \in [1,\ldots,M_{t'}]\\
&\qquad\qquad X_{t,p,j} \geq X_{t',p,j'} + S_{t',j'} - (U \times Y_{p,t\_j,t'\_j'})\\
&\quad (c)\ \ \forall j \in [1,\ldots,M_t] \qquad \forall j' \in [1,\ldots,M_{t'}]\\
&\qquad\qquad X_{t,p,j} \leq X_{t',p,j'} - S_{t,j} + (U \times (1 - Y_{p,t\_j,t'\_j'}))\\
&\quad (d)\ \ \forall j \in [1,\ldots,M_t]\\
&\qquad\qquad X_{t,p,j} \geq 0\\
&\quad (e)\ \ \forall j \in [1,\ldots,M_t]\\
&\qquad\qquad C \geq X_{t,p,j} + S_{t,j}
\end{aligned} \tag{2}$$

The objective is to pick a shard unit that can minimize makespan (completion time of the whole workload at this granularity). Constraints (a) simply enforce the *sequential ordering of shard units* within a model. Note that this set per model here is unified within a mini-batch, across mini-batches within an epoch, and potentially across epochs too–they are all sequentially dependent. Constraints (b) and (c) enforce *model training isolation*, i.e., only one shard unit can run on a device at a time. Constraints (d) is just non-negativity of start times, while Constraints (e) define the makespan.

Using a MILP solver such as Gurobi [15] enables us to produce an "optimal" schedule in this context. But the above task is a variant of a general job-shop scheduling problem described in [53], and it is known to be NP-complete. Given that the number of shard units can span thousands to tens of millions, solving it optimally will likely be impractically slow. Thus, we look for fast and easy-to-implement scheduling algorithms that can still offer near-optimal makespans.

4.7.2 *Intuitions on Scheduling Effectiveness.*

We observe that there are 2 primary cases encountered by a scheduler in our setting:

(1) The number of models is equal to or greater than the number of available devices.
(2) The number of models is less than the number of available devices.

In case (1), there will always be at least one eligible shard unit for each device at every scheduling decision. Any shard-parallel scheduling algorithm that accounts for all devices can easily keep all devices busy most of the time, i.e., *busy waiting* is unlikely. In case (2), all models can be trained simultaneously. Since each model's shard unit uses at most one device in SHARP, and since there are more devices than models, there is no contention for resources here. In this case, regular task parallelism-style scheduling suffices and the makespan will just be the runtime of the longest "task."

In both the cases above, even basic randomized scheduling might yield reasonable makespans. However, what it will not take into account is that case (1) is not static. Over time, as models finish their training, our setting may "degrade" from case (1) to case (2). Thus, two different schedulers that operate on a workload in case (1) may differ in their effectiveness based on how gracefully they degrade to case (2). As noted before, the makespan in case (2) scenario is determined solely by the longest-running remaining model. This gives us an intuition for a simple scheduler that can often do better than randomized: *minimize the maximum remaining time among the remaining models.*

If degradation to case (2) occurs early on, and if there is a substantial differences in task runtimes post-degradation, the overall completion times can differ significantly based on the scheduling. Such degradation can arise in model selection workloads that use early stopping for underperforming models, e.g., Hyperband [28], or by manual user intervention. Thus, we aim for a scheduling algorithm that can address such cases too in a unified way.

We propose a simple and practical greedy heuristic we call Sharded Longest Remaining Time First (LRTF) based on our above intuitions. Algorithm 2 explains our algorithm. Sharded-LRTF selects the model training task with the *longest total remaining train time* at every possible scheduling decision time. Since a new scheduling choice must be made at the completion of every shard unit, the selection will update its choice of longest task at a very fine-grained level. Note that the selection procedure runs efficiently with *linear time complexity.*[3]

[3]In fact, an alternate data structure to record shard references can enable even constant-time selection.

Algorithm 2: The Sharded-LRTF scheduling algorithm.

```
Struct {
    Remaining epochs $e$
    Minibatches per epoch $b$
    Remaining minibatches in current epoch $ce$
    Minibatch training time $t$
    Remaining train time in current minibatch $cm$
}
Input: Idle Models $[M]$
Output: Model $MaxModel$
$MaxTrainTime = 0$
for Index $i$, Model $m$ in $[M]$ do
    $ModelTrainTime = ((m_e - 1) \times m_b + m_{ce} - 1) \times m_t + m_{cm}$
    if $ModelTrainTime > MaxTrainTime$ then
        $MaxTrainTime = ModelTrainTime$
        $MaxModel = m$
    end if
end for
```

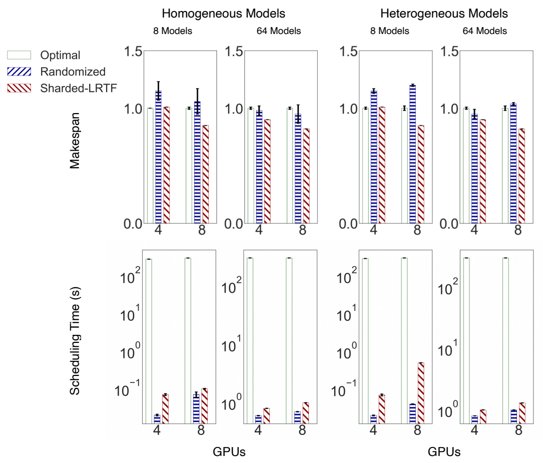

Figure 7: Comparison of various scheduling algorithms. Makespans are normalized to Optimal.

4.7.3 *Our Scheduling Algorithm: Sharded-LRTF.*

To quantitatively understand the effectiveness of Sharded-LRTF, we compare it using simulations against a basic randomized schedule and a Gurobi-output "optimal" schedule. For tractability, we set a timeout of 100s for Gurobi. We who both a homogeneous setting (all neural architecture are identical) and a heterogeneous setting, wherein they differ significantly. We assume all GPU devices are identical for simplicity, but that is also common in practice. Per-epoch runtimes of a model in the homogeneous setting are all fixed to 2 hours each, with 2000 shard units each. For the heterogeneous setting, per-epoch model runtimes are set between 30 minutes to 4 hours; number of shard units are set between 100 to 10,000. We randomly sample an initial set and report the average and standard deviations of 3 runs on the fixed set. Variance occurs due to non-deterministic scheduling behaviors from random selection and Gurobi timeout. Figure 7 shows the results.

We note that MILP "optimal" has higher makespan in some cases because Gurobi did not converge to the global optimal in the given time budget. The randomized approach matches it or performs worse in many cases. But Sharded-LRTF matches or significantly outperforms the other approaches in many cases, especially in the heterogeneous setting. This is in line with the intuition we explained that being cognizant of longer running models in the mix is helpful. Also note that the runtime of Sharded-LRTF is in the order of tens of milliseconds, ensuring it is practical for us to use in HYDRA repeatedly for scheduling shard units on devices dynamically. Note that the actual mini-batch training computations on the device are the dominant part of the overall runtime.

5 EXPERIMENTS

We now compare HYDRA against state-of-the-art open source and industrial tools for large-model DL training: PyTorch Distributed, Microsoft's DeepSpeed, FlexFlow from Stanford/CMU, and Google's GPipe. GPipe's microbatch count and partition count is equal to the GPU count. FlexFlow requires some manual guidance by editing the system-generated parallelism strategy file to ensure memory errors do not occur. We also show multiple variants with DeepSpeed, including superimposing a hybrid task parallelism (note that regular task parallelism is not applicable) and a hybrid data parallelism offered by DeepSpeed. We then dive into how HYDRA performs when various workload and system parameters are varied.

Workload Details. We use two popular DL model selection scenarios: hyperparameter evaluation and neural architecture evaluation. Table 2 lists details. For hyperparameter evaluation, we focus on masked-language modeling with the Transformer architecture BERT-Large [9], trained on the WikiText-2 dataset. The neural architecture is fixed and we vary batch size and learning rate as key hyperparameters to create a total of 12 models to train, each with 1B parameters. For neural architecture evaluation, we focus on a computer vision task with variants of the Vision Transformer (ViT) model [10] and the CIFAR-10 dataset. We create models with sizes between 300M parameters and 2B parameters. We also vary batch sizes, leading to a total of 12 models again.

Machine Setup. We focus on single-node multi-GPU training, anecdotally the most common among DL practitioners. We use 8 Nvidia RTX 2080Ti GPUs with 11GB memory, NVLink-enabled. The machine runs Ubuntu 18.04. and the node has 500GB DRAM and 80 CPU cores.

5.1 End-to-End Workloads

Figure 8 presents overall runtimes and GPU utilization results. We find that the baseline off-the-shelf PyTorch Distributed and DeepSpeed model parallelism report massive resource under-utilization. Thus, their runtimes are the highest. The basic hybrids with data- or task- parallelism do provide higher utilization and some modest speedups, but the fundamental limitations of model parallelism persist with such approaches, such that they still fall substantially short of ideal linear speedup (8x in this case). GPipe-style pipeline parallelism is much better, with about a 4x speedup against regular model parallelism. But HYDRA is the most efficient approach overall, reaching about 7.5x, close to the physical upper bound. The average GPU utilization of HYDRA is also the highest at over 80%.

Table 2: Details of end-to-end workloads. *Architectures similar to BERT-Large and ViT, scaled up for demonstration.

Dataset	Model Architectures	Model Sizes	Batch Size	Learning Rate	Epochs
WikiText 2	BERT-Large*	1B	8, 16, 32	$10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$	4
CIFAR-10	Vision Transformer (ViT)*	300M, 600M, 800M, 1B, 1.5B, 2B	512, 1024	10^{-3}	5

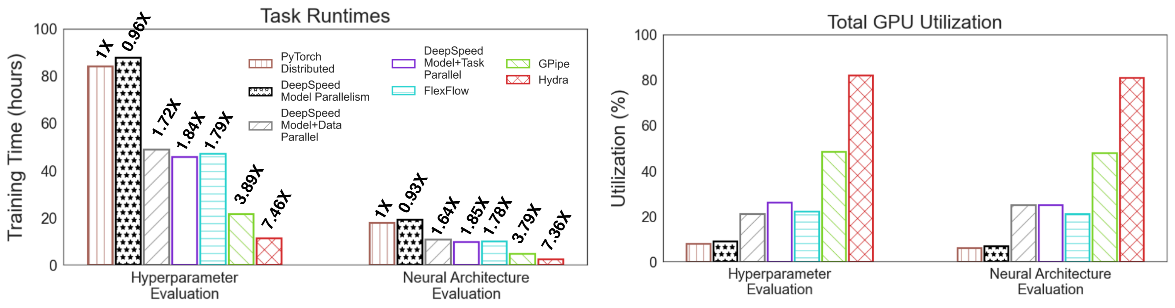

Figure 8: End-to-end workload results: Runtime speedups relative to the baseline PyTorch Distributed and GPU utilization. All evaluations were closely monitored to ensure none suffered hardware failure or GPU disconnects.

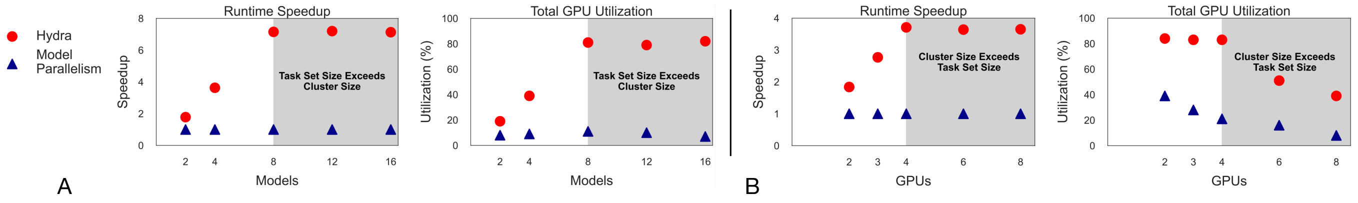

Figure 9: System microbenchmarks. A) demonstrates the impact of task set size on performance while resources are fixed, while B) demonstrates the impact of cluster size on performance while the task set size is fixed.

5.2 Drill Down Analysis

We now dive deeper into the behavior of HYDRA when varying key parameters of interest from both ML and system standpoints.

5.2.1 ***Impact of Model Scale.*** We vary the scale of the models to see the impact on relative performance of HYDRA. We fix the number of GPUs at 8 and the number of models to 12. Figure 10 shows the results. We see that HYDRA's speedups over regular model parallelism *is fairly consistent* even as the model scale grows. This is because our partitioning approach (Section 4.3) and the dynamic Sharded-LRTF algorithm (Section 5.3) together ensure that shard unit times are similar even as model scale grows; basically, it just leads to more shard units to run. Our SHARP and double-buffering techniques further ensure that having more shard units do *not* cause relatively more resource idling on average.

5.2.2 ***Impact of Number of Models.*** We now vary the number of models that are trained together. The number of GPUs is set to 8; all models have are uniformly large, at 250M parameters (same Transformer workload as before). Figure 9A shows the results. We see that HYDRA exhibits close to 8x speedups when the number of models is 8 or more but lower than that, the speedup is capped

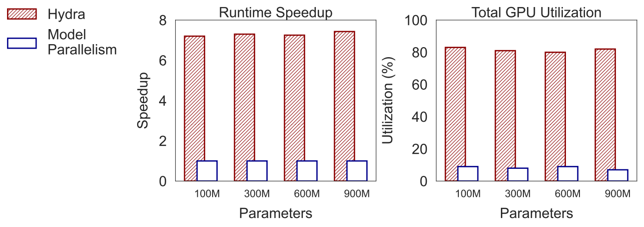

Figure 10: Impact of model scale. Runtimes normalized to the first instance of regular model parallelism for clarity.

close to the actual number of models. This flattening in the fewer-models regime is inherited from task parallelism by SHARP. The GPU utilization numbers vary proportionally to the speedups seen.

5.2.3 ***Impact of Number of GPUs.*** We now study how varying the number of GPUs affects HYDRA's speedup behavior. We fix the workload to 4 Transformer models, each with 250M parameters. We choose only 4 models to showcase both regimes: when the number of devices is less than models and vice versa. Figure 9B shows the results. We see that HYDRA exhibits a roughly linear speedup when there are more models than devices. And when that flips, since HYDRA runs out of models to schedule, the speedup flattens as the degree of parallelism is limited. As before, this is due to SHARP inheriting the degree of parallelism from task parallelism. We believe further hybridization of SHARP with data parallel training can help boost speedups and resource utilization in this regime; due to its complexity, we leave it to future work.

Optimization Level	Runtime (hrs)	Runtime relative to Hydra
Hydra without SHARP or double-buffering	41.5	13.05X
Hydra without double-buffering	4.8	2.3X
Hydra	1.85	1X

Table 3: Runtimes and slowdowns of Hydra when our two key optimizations are disabled one by one.

5.2.4 *Ablation Tests.*

In this experiment, we explore the effect of system components on framework performance. The number of devices is fixed to 8, with 16 Transformer models. All optimization levels include model spilling as a baseline, as this technique is critical to Hydra's basic operations. Table 3 demonstrates the results. Pure model spilling dramatically slows down model training. This is only to be expected, given that it introduces a dependency on DRAM. SHARP's throughput improvements dominate the slowdowns of model spilling, but it is important to note that SHARP's speedups are workload-dependent. Double-buffering largely eliminates the cost of model spilling, enabling further speedups.

6 CURRENT LIMITATIONS AND FUTURE OPPORTUNITIES

Non-Sequential Neural Architectures. As mentioned earlier, Hydra focuses on neural computational graphs that can be represented as sequences of layers or groups of layers (prior work on pipeline parallelism also assumes this [12, 29, 41, 57]). The most popular classes of GPU-memory-bottlenecked models in DL practice today, viz., Transformers, as well as most convolutional neural networks and multi-layer perceptrons do satisfy the assumption. Some not-fully-sequential models such as Inception, ResNets, and DenseNets are easily handled because residual or skip connections can be linearized with a single "super-vertex" in the graph specification given to Hydra. As long as the user defines the graph in that way using the DL tool's API, Hydra works out of the box for such models too. But for recurrent neural networks (RNNs) and graph neural networks (GNNs), Hydra would need to be extended to account for non-trivial dependencies across shard units of a model. Backpropagation through time, maintaining memory cells, and cross-layer global connections all require non-trivial extra implementation machinery and modifications to our Scheduler. We leave such extensions to future work.

Large Model Inference. This paper focused primarily on training of large models. But a trained model is then used for inference in an application. If one wants to use a GPU for inference, the same GPU memory bottleneck exists. Fortunately, Hydra's model spilling, automated partitioning, and automated shard orchestration all suffice already for out-of-the-box large model inference too. While we have not implemented an inference API as of this writing, it is a straightforward addition we plan to do soon.

DL Tool Generality. Hydra is currently implemented as a wrapper around PyTorch. But all of our techniques described in Section 4 are generic enough to be used with, say, TensorFlow or MXNet as well. We just chose to prioritize the depth of our system over generality of DL tools as of this writing. But nothing in Hydra prevents support for additional DL tools in the future.

CUDA-level Optimization. We designed Hydra to operate on top of PyTorch to ensure backward compatibility as PyTorch evolves. This means we did not exploit any low-level optimizations for GPU-to-GPU transfers. One could technically imagine using multiprocessing in CUDA to reduce this latency, including for our double-buffering technique. But all this will require us to write new kernels in CUDA for memory management and hook them into the DL tool. We leave such ideas to future work.

More Hybrid Parallelism. When there are fewer models than devices, Hydra may under-utilize the devices due to the limitation it inherits from task parallelism. But one could do better by hybridizing data parallelism and pipeline parallelism with Hydra to raise overall resource utilization. This is feasible because both of those approaches are technically *complementary* to SHARP and Hydra's other techniques. We leave such sophisticated hybrid-of-hybrids parallelization to future work. But we note that in cases where there are more models than devices, SHARP is already close to optimal utilization, as our empirical results show.

7 RELATED WORK

Pipeline parallelism [16, 19], a state-of-the-art technique in large-scale model training, builds on top of model parallelism by treating the sequence of shards as a staged pipe. It exploits the fact that DL training uses mini-batch SGD wherein subsequent mini-batches are independent samples of the training dataset, and stages out successive mini-batches through the pipe of shards to reduce idling. Figure 3 illustrates. Prior work on pipelining has explored the problem of "bubble" periods during which the pipe has to be flushed to avoid collisions between mini-batches moving in opposite directions [12, 29, 39, 57].

One approach suggests mitigating the issue by creating "asynchronous" stages that use stale parameters for updates without executing backward passes of mini-batches in synchronous order, rescheduling them to reduce the number of flushes that need to occur in total [12, 16, 39, 57]. However, asynchronous pipelining introduces accuracy degradation and cannot guarantee convergence to the same degree of accuracy as the non-parallelized model [2, 31, 34, 52]. As such, it violates one of the key desiderata we proposed in Section 1, and is not directly comparable to Hydra. Future work could relax this accuracy constraint to integrate asynchronous updates into Hydra.

Pipelining is orthogonal to our own work, as it focuses on single model execution. We leave it to future work to hybridize Hydra with both synchronous and asynchronous pipelining in the case where there are fewer models than GPUs — Hydra could use pipelining to make use of extra GPUs.

Data parallelism in ZeRO and DeepSpeed [44] applies parameter sharing to distribute memory costs across model instances. This reduces the memory demands per GPU, but in cases when the model is very large, these frameworks still need to be hybridized with model parallelism. Future work could look to instead hybridize them with Hydra to simultaneously optimize data parallelism and model parallelism.

ZeRO-Offload and L2L [42, 46] propose offloading data from GPU memory to DRAM to improve scalability on a single GPU. ZeRO-Offload runs parameter updates on the CPU instead of the GPU, and L2L executes a single layer at a time in a quasi-spilling approach. However, these systems are not scoped to address multi-GPU execution, nor do they have any inherent handling for multi-model training. By contrast, our spilling design creates semi-independent model shards that can be optimized flexibly to support the broader, multi-task scheduling technique we build with SHARP. These single-model optimization schemes are orthogonal to our own work, and address a different problem.

SwapAdvisor[18] is a tensor offloading system that allows data (parameters, gradients, intermediates) to be swapped out of GPU memory and onto DRAM. This enables large models to be trained on a single GPU. Unlike Hydra's flexible shard-swapping, SwapAdvisor swaps at an individual tensor level with a complex swapping plan optimized across the entire model's execution. The degree of optimization prevents easy generalization to SHARP's multi-model blended scheduling. In addition, SwapAdvisor's swap plan simulator can introduce overheads due to the sheer size of the optimization space, and the communication latencies introduced by swapping can be costly. Another data offloading framework [32] proposed offloading only intermediates activations produced during training that would not be necessary until later in the model's execution. Such offloads are only possible along long-term residual connections that span several layers' worth of execution time, and the design also requires that model parameters fit on GPU memory.

Both these frameworks could be hybridized with Hydra to enable larger shards to be trained, but this would increase system complexity substantially.

Tensor parallelism is an alternative to inter-layer model parallelism that offers the ability to parallelize compute between model shards. Rather than splitting a model into different layer groups, it splits individual layers into multiple pieces that can be run in parallel [23, 49]. However, tensor parallelism introduces substantial overhead in splitting tensors and recombining them, as well as challenges around efficient data communication. Moreover, not all layers can be split in such a fashion — convolutional layers are among the few that can be tensor-parallelized easily. In the future, Hydra could be extended to support mixed layer-tensor parallelism, but for now we only address inter-layer model parallelism.

FlexFlow [23] hybridizes execution between model and data parallelism. On a per-layer basis, it chooses to execute in either a model-, tensor-, or data- parallel fashion, with the ultimate aim of improving performance. This still forces users to obtain multiple GPUs. In addition, supporting larger-than-GPU-memory training is not an explicit aim of FlexFlow, and the simulator sometimes produces designs that are not feasible given the memory constraints [3]. This becomes apparent at very large model scales, where manual intervention must occur for the framework to even execute without memory errors.

Reducing model memory footprints has received much attention in DL systems [8, 14, 21, 22, 26]. Model quantization [20] in particular has been a popular technique for reducing memory footprints at inference time. The goal of such systems is *orthogonal* to our own, and memory reduction techniques could be integrated into Hydra in the future. Other work on machine teaching [54] and data distillation [55] aims to minimize the memory footprints of data, but these techniques address a different aspect of memory in DL systems.

Multi-query optimizations for DL systems are techniques to optimize ML systems by exploiting multi-model execution, e.g., systems such Cerebro [25], ModelBatch [40], ASHA [27], and SystemML [5]. Cerebro proposes a hybrid parallelism scheme named MOP combining task- and data- parallelism, akin to (but different from) SHARP's hybrid model-task parallelism. SystemML also hybridizes task- and data- parallelism, but for classical ML workloads rather than DL. ModelBatch raises GPU utilization by altering the DL tool's internal execution kernels. None of them tackle larger-than-GPU-memory models, which is our focus. Other examples of MQO for DL systems are Krypton [36] and HummingBird [37] but they focus on inference, not training.

We presented an early version of this work at a non-full length (2 page) venue [35]. This paper realizes the vision proposed there to build Hydra by fleshing out SHARP along with a thorough empirical evaluation on real DL workloads.

8 CONCLUSION AND FUTURE WORK

Training larger-than-GPU-memory DL models is an increasingly critical need for DL users. Yet existing "model parallelism" tools are sub-par on scalability and parallelism, are often hard to use, and massively underutilize GPUs. Moreover, no existing system inherently supports both multi-model training and larger-than-GPU-memory model training. We present Hydra, a new system for large-scale multi-model DL training inspired by the design and implementation of RDBMSs that uses the memory hierarchy consciously and exploits multi-task execution to optimize performance. We identify a judicious mix of data systems techniques–some novel and some classical RDBMS ideas adapted to DL (such as sharding, spilling, and double buffering)–to enable large-model training even on a single GPU. By further exploiting the high degree of parallelism in multi-model training, we devise a novel hybrid parallel execution technique inspired by multi-query optimization. Our work shows that the DL systems world can benefit from learning from the RDBMS world on data systems techniques that enable more seamless scalability and parallelism for DL users.

As for future work, we aim to improve the efficiency of Hydra when there are fewer models than devices by hybridizing our ideas with data parallelism and pipeline parallelism. We also plan to expand support for more complex non-sequential neural architectures such as RNNs and GNNs. Finally, we aim to expand support in Hydra for other popular DL tools and emerging DL-oriented compute hardware beyond GPUs.

REFERENCES

[1] 2021. What was the largest dataset you analyzed / data mined?
[2] Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Mike Rabbat. 2019. Stochastic Gradient Push for Distributed Deep Learning. In *Proceedings of the 36th*

International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 344–353. https://proceedings.mlr.press/v97/assran19a.html

[3] Neil Band. 2020. *MemFlow: Memory-Aware Distributed Deep Learning*. Association for Computing Machinery, New York, NY, USA, 2883–2885. https://doi.org/10.1145/3318464.3384416

[4] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018). arXiv:1802.09941 http://arxiv.org/abs/1802.09941

[5] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[6] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2018. Large scale GAN training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096* (2018).

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *CoRR* abs/2010.11929 (2020).

[11] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking energy. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 393–406.

[12] Alexander L. Gaunt, Matthew A. Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. 2017. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks. arXiv:1705.09786 [cs.LG]

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[14] Audrūnas Gruslys, Remi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-Efficient Backpropagation Through Time. (2016). arXiv:1606.03401 [cs.NE]

[15] LLC Gurobi Optimization. 2021. Gurobi Optimizer Reference Manual. "http://www.gurobi.com

[16] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018). arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[17] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[18] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[19] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965 http://arxiv.org/abs/1811.06965

[20] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 [cs.LG]

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. 2019. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. *CoRR* abs/1910.02653 (2019). arXiv:1910.02653 http://arxiv.org/abs/1910.02653

[22] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. *SOSP '19* (2019).

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* (2018).

[24] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. 2016. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44, 4 (May 2016), 17–22.

[25] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. 2021. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper25.pdf

[26] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient Rematerialization for Deep Networks. 32 (2019), 15172–15181.

[27] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. arXiv:1810.05934 [cs.LG]

[28] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *CoRR* abs/1603.06560 (2016). arXiv:1603.06560 http://arxiv.org/abs/1603.06560

[29] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[30] Side Li and Arun Kumar. 2021. Towards an optimized GROUP by abstraction for large-scale machine learning. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2327–2340.

[31] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 3043–3052. https://proceedings.mlr.press/v80/lian18a.html

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

[33] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *CoRR* abs/1609.07843 (2016). arXiv:1609.07843 http://arxiv.org/abs/1609.07843

[34] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Ilia Markov, Shigang Li, and Dan Alistarh. 2020. Decentralized SGD with Asynchronous, Local and Quantized Updates. arXiv:1910.12308 [cs.LG]

[35] Kabir Nagrecha. 2021. Model-Parallel Model Selection for Deep Learning Systems *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 2929–2931. https://doi.org/10.1145/3448016.3450571

[36] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. 2020. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.* 45, 4, Article 16 (Dec. 2020), 42 pages. https://doi.org/10.1145/3397461

[37] Supun Nakandala, Gyeong-In Yu, Markus Weimer, and Matteo Interlandil. 10‘9. Compiling Classical ML Pipelines into Tensor Computations for One-size-fits-all Prediction Serving. *Conference and Workshop on Neural Information Processing Systems* (10‘9). https://arxiv.org/abs/2005.08314

[38] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.

[39] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. arXiv:2006.09503 [cs.LG]

[40] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. 2018. Accelerating Model Search with Model Batching. *Proceedings of Fourth Conference on Machine Learning and Systems (MLSys'18)* (2018).

[41] 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. *CoRR* abs/2104.04473 (2021). arXiv:2104.04473 https://arxiv.org/abs/2104.04473

[42] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. 2020. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. arXiv:2002.05645 [cs.LG]

[43] 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 preprint arXiv:1910.10683* (2019).

[44] Samyam Rajbhari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. (2020).

[45] Raghu Ramakrishnan and Johannes Gehrke. 1996. *Database Management Systems*.

[46] 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:2101.06840 [cs.DC]

[47] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[48] S. Shaleve-Shwartz and S. Ben-David. 2014. *Understanding Machine Learning: from Theory to Algorithms.* Cambridge University Press.

[49] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, and Ashish Vaswani. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. *CoRR* abs/1811.02084 (2018). arXiv:1811.02084 http://arxiv.org/abs/1811.02084

[50] 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. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[51] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-Memory Neural Network Training: A Technical Report. *CoRR* abs/1904.10631 (2019). arXiv:1904.10631 http://arxiv.org/abs/1904.10631

[52] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. 2020. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307* (2020).

[53] J.D. Ullman. 1975. NP-Complete Scheduling Problems. *Journal of Computer , System Sciences.* 10, 3 (June 1975), 384–393.

[54] Pei Wang, Kabir Nagrecha, and Nuno Vasconcelos. 2021. Gradient-based Algorithms for Machine Teaching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 1387–1396.

[55] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. 2018. Dataset Distillation. *CoRR* abs/1811.10959 (2018). arXiv:1811.10959 http://arxiv.org/abs/1811.10959

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[57] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. 2020. PipeMare: Asynchronous Pipeline Parallel DNN Training. arXiv:1910.05124 [cs.DC]

[58] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. *CoRR* abs/2005.08314 (2020). arXiv:2005.08314 https://arxiv.org/abs/2005.08314