Saturn: An Optimized Data System for Multi-Large-Model Deep Learning Workloads (Information System Architectures)

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

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

ABSTRACT

Large models such as GPT-3 have transformed deep learning (DL), powering applications like ChatGPT that have captured the public's imagination. Such models are rapidly gaining adoption across domains for analytics on various modalities, often through the process of finetuning pre-trained "base" models. Such models need multiple GPUs due to both their size and computational load, leading to a bevy of "model parallelism" techniques and tools. Navigating such *parallelism* choices, however, is a new burden for end users of DL such as data scientists, domain scientists, etc. who may lack the necessary systems knowhow. The need for *model selection*, which leads to many models to train due to hyper-parameter tuning or layer-wise finetuning, compounds the situation with two more burdens: *resource apportioning* and *scheduling*. In this work, we tackle these three burdens for DL users in a unified manner by formalizing them as a joint problem that we call SPASE: Select a Parallelism, Allocate resources, and Schedule. We propose a new information system architecture to tackle the SPASE problem holistically, representing a key step toward enabling wider adoption of large DL models. We devise an extensible template for existing parallelism schemes and combine it with an automated empirical profiler for runtime estimation. We then formulate SPASE as an MILP. We find that direct use of an MILP-solver is significantly more effective than several baseline heuristics. We optimize the system runtime further with an introspective scheduling approach. We implement all these techniques into a new data system we call Saturn. Experiments with benchmark DL workloads show that Saturn achieves 39-50% lower model selection runtimes than typical current DL practice.

PVLDB Reference Format:
Kabir Nagrecha and Arun Kumar. Saturn: An Optimized Data System for Multi-Large-Model Deep Learning Workloads (Information System Architectures). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/saturn-64DC.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Large-model deep learning (DL) is growing in adoption across many domains for data analytics over text, image, video, and even multimodal tabular data. Large language models (LLMs) now power popular applications like ChatGPT [62]. Such models have been ballooning in size, as Figure 1(A) shows [30]. For instance, the popular GPT-J [87, 108] and ViT [31] models need 10s of GBs of GPU memory and take days to train. This is often impractical for DL users in smaller companies, enterprises, and the domain sciences. Thankfully, in most cases they need not train from scratch to benefit from large-model DL. They can download "base" models, pre-trained on large general datasets (e.g., much of the Web), from model hubs like HuggingFace [113] and just "finetune" them on their (smaller) application-specific data [20]. This enables companies to keep their application data in house. Recent market research reports that this form of large-model DL is rapidly growing [5].

While finetuning and customizing of base models has made large-model DL more tractable, end users of DL still face 3 systems-oriented headaches: (1) *GPU memory* remains a bottleneck. Large-memory GPUs are expensive, and even public cloud vendors still ration them. (2) *Multi-GPU parallelism* is needed but understanding the performance behaviors of complex large-model parallelism techniques is difficult for DL users; and (3) *Model selection*, which involves tuning hyper-parameters, model layers, etc., only amplifies the computational load.

Overall, large-model DL, including finetuning, is still painful for regular DL users, hurting usability and raising runtimes and costs, especially in pay-as-you-go clouds. Consider the following example scenario that highlights the concrete issues with the status quo.

Case Study: Consider a data scientist, Alice, building an SQL autocomplete tool to help database users at her company. She has a (private) query log that contains her company's database schemas, common predicates, etc. She downloads two LLMs from HuggingFace — GPT-2 and GPT-J — both of which are known to offer strong results for textual prediction tasks [87, 108]. She finetunes multiple instances on her dataset, comparing different batch sizes and learning rates to raise accuracy. She uses an AWS instance with 8 A100 GPUs. She launches the DL tuning jobs in parallel, assigning one GPU each. Alas, all of them crash with out-of-memory (OOM) errors. She is now forced to pick a large-model scaling/parallelism technique and assign multiple GPUs to each job. But to do so she must answer 3 intertwined systems-oriented questions: (1) Which parallelism technique to use for each model? (2) How many GPUs to assign to each model? (3) How to orchestrate such complex parallel execution for model selection workloads?

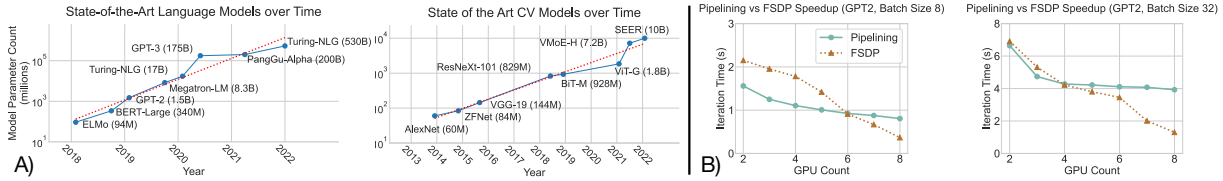

Figure 1: (A) Trends of the sizes of some state-of-the-art DL models in NLP and CV (log scale), extrapolated from a similar figure in [98]. (B) Our empirically measured runtime crossovers between FSDP and pipeline parallelism, with knobs tuned per setting.

In this paper, we tackle precisely those 3 practical questions in a unified way to make it easier, faster, and cheaper for regular DL users like Alice to benefit from such state-of-the-art large DL models.

1.1 Prior Art and Their Limitations

We start by first explaining why prior art for large-model and parallel DL systems is insufficient to tackle the problem. Table 1 lists a conceptual comparison of our setting with prior art on several key aspects. (Section 6 discusses related work in greater detail.)

(1) Which parallelism technique to use for each model? There is a suite of techniques in the ML systems world to parallelize/scale large models across GPUs. Some common techniques are: sharding the model, spilling shards to DRAM [44, 68], pipeline parallelism as in GPipe [45], fully-sharded data-parallel (FSDP) as in PyTorch [2] and ZeRO [88], hand-crafted hybrids as in Megatron [98], as well as general hybrid-parallel approaches such as Unity [49, 104] and Alpa [124]. But no technique dominates all others in all cases. Relative efficiency depends on a complex mix of factors: hardware, DL architecture specifics, even batch size for stochastic gradient descent (SGD). Figure 1(B) shows two empirical results on real workloads to prove our point. Even between just pipelining and FSDP, complex crossovers arise as GPU counts and batch sizes change. Furthermore, many techniques expose knobs that affect runtimes in hard-to-predict ways [60], e.g., pipelining requires tuning partitions and "microbatch" sizes, while FSDP requires tuning offloading and checkpointing decisions. *Thus, we need to automate parallelism technique selection for large-model DL training.*

(2) How many GPUs to assign to each model? Many DL practitioners use fixed clusters or have bounded resource budgets. So, they are either given (or decide) up front the number of GPUs to use. But in multi-model settings like model selection, there is more flexibility on apportioning GPUs across models. The naive approach of running models one after another using all GPUs is sub-optimal as it *reduces model selection throughput* and adding more GPUs per model yields diminishing returns. Alas, the scaling behaviors of large-model parallelism techniques are not linear and often hard to predict, as Figure 1(B) shows. Prior art has studied data-parallel resource allocation, e.g., Pollux [86] and Optimus [80]) and model selection optimization (e.g., Cerebro [54] and ASHA [56]). But none of them target large-model DL, which alters the cost-benefit tradeoffs of GPU apportioning in new ways due to interplay with parallelism selection and complex scaling behaviors. *Thus, we must automate GPU apportioning for large-model model selection.*

(3) How to orchestrate such complex parallel execution for model selection? This is a scheduling question, i.e., deciding which jobs to run when. Two naive approaches are to run models in a random order or to use a generic task scheduler. Both can lead to GPU idling due to a lack of awareness of how long models actually run. Prior art has studied runtime-aware DL scheduling, e.g., Gandiva [114] and Tiresias [38], but none target large-model DL. The complex interplay of parallelism selection and GPU apportionment can affect runtimes in a way that alters the tradeoffs of scheduling. The model selection setting adds more considerations: we must optimize end-to-end *makespan* rather than just a throughput objective [80, 86]. Specific desiderata must be met: *fidelity* on ML accuracy and *generality* on specification. We expand on these in Section 1.2.

Overall, there is a pressing need for a unified and automated way to tackle these 3 systems concerns of DL model selection on large models: select parallelism technique per model, apportion GPUs per model, and schedule them all on a given cluster. No prior art — including all those described in Table 1 — can address this novel setting that has emerged with the rise of large-model DL. We call this new joint problem SPASE: Select Parallelism, Apportion resources, and SchedulE.

1.2 System Desiderata

To help democratize large-model DL and ease practical adoption, we seek a data system that tackles SPASE with the following desiderata:

(1) Extensibility on parallelism selection. Given the variety of large-model parallelism techniques (henceforth called "parallelisms" for brevity), the system must support and select over multiple parallelisms and also make it easy for users to add new parallelisms in the future (e.g. for model-technique codesign [33, 79, 98]). Without support for user extension, parallelism selectors/hybridizers are limited in scope, as noted in Table 1.

(2) Non-disruptive integration with DL tools. The system must natively support popular DL tools such as PyTorch [59] and TensorFlow [6] without modifying their internals. This can offer backward compatibility as those tools evolve.

(3) Generality on multi-model specification. The system should support multiple model selection APIs, e.g., grid/random search or AutoML heuristics. We assume the system is given a set of model training jobs with known epoch counts. Evolving workloads can be supported by running all models one epoch at a time.

(4) Fidelity on ML accuracy. The system must not deliberately alter ML accuracy when applying system optimizations. Approximations such as altering the model, training algorithm, or workload parameters are out of scope because they can confound users.

1.3 Our Proposed Approach

To meet all of the above desiderata, we design a new information system architecture to tackle SPASE that is inspired by some

Table 1: Overview of prior art. Column desiderata are described in Sections 1.1 and 1.2.

		Fidelity	Multi-Model	Resource Allocation	Parallelism Selection	Out-of-the-Box Large Model Support
Hybrid Parallelism	Alpa [124]	✓	✗	✗	✓(limited)	✓
	FlexFlow [49]	✓	✗	✗	✓(limited)	✗
	Unity [104]	✓	✗	✗	✓(limited)	✓
Performance Evaluation	Paleo [84]	✓	✗	✗	✓(limited)	✗
Model Selection	Cerebro [54]	✓	✓	✗	✗	✗
	ASHA [56]	✓	✓	✓	✗	✗
Scheduling	Gandiva [114]	✓	✓	✗	✗	✗
	Antman [115]	✓	✓	✗	✗	✗
	Tiresias [38]	✓	✓	✗	✗	✗
Resource Allocation	Pollux [86]	✗	✓	✓	✗	✗
	Optimus [80]	✗	✓	✓	✗	✗
SPASE	Saturn (ours)	✓	✓	✓	✓	✓

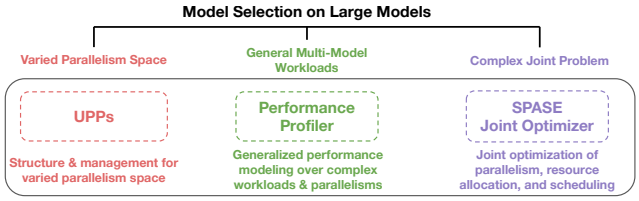

Figure 2: Overview of how Saturn's components tackle the SPASE problem for multi-large-model DL workloads.

techniques in database systems. We call our system Saturn. Our current focus is on the common fixed-cluster setting rather than autoscaling [95]. As Figure 2 shows, our approach is three-pronged:

(1) Parallelism Selection and UPPs. We translate high-level ("logical") model training specifications into optimized "physical" parallel execution plans based on instance details, inspired by physical operator selection in RDBMSs, e.g., selecting hash-join vs. sort-merge join for a given join operation. To meet the first desideratum of extensibility, we introduce the abstraction of User-Pluggable Parallelisms (UPPs). UPPs can be used to specify existing parallelisms in standard DL tool code, or enable users to add new parallelisms as blackboxes for Saturn to use. This also ensures the second desideratum of non-disruptive integration. We create a default UPP library in Saturn to support 4 major existing parallelisms: pipelining, spilling, distributed data parallelism (DDP), and FSDP. Each UPP can support knob-autotuning, similar to auto-tuning of physical configuration parameters of a data management system [42, 105].

(2) Performance Profiling. To apportion GPUs and select parallelisms in a way that ensures the fourth desideratum, we need accurate estimates of job runtimes *as is*. We exploit a basic property of SGD: since minibatch size is fixed within an epoch, we can typically project epoch times accurately from runtime averages over a few minibatch iterations. Coupled with the offline nature of model selection, we can create a general and effective solution: profile all jobs using the full "grid" of options for both GPU counts and parallelisms based on only a few minibatches. The overhead of this approach is affordable due to the long runtimes of actual DL training. This also ensures our second and third desiderata as all DL tools offer data sampling APIs that we can just use on top of the user-given model specifications. Of course, we use the full training data for the actual DL jobs to ensure the fourth desideratum.

(3) Joint Optimization and Scheduling. Given the above system design choices, we can now tackle SPASE using joint optimization. We formalize this problem as a mixed-integer linear program (MILP). Using realistic runtime estimates, we perform a simulation study to compare an MILP solver (we use Gurobi [39]) to a handful of strong scheduling heuristics. The solver yields the best results overall even with a timeout. Thus, we adopt it in Saturn as our SPASE optimizer. Actual model training, not the optimizer, heavily dominates overall runtimes in DL workloads, so we view this design decision as reasonable because it ensures *both efficiency and simplicity*, easing system maintenance and adoption. Finally, we augment our Optimizer with an "introspective" scheduling extension known in prior art to further raise resource utilization.

We intentionally design Saturn to be a simple and intuitive system to tackle SPASE in a way that can help ease practical adoption. Figure 3 in Section 3 shows our system architecture. Saturn is implemented in Python and exposes high-level APIs for (offline) specification of UPPs and model selection APIs for actual DL training usage. Under the hood, Saturn has 4 components: Parallelism Plan Enumerator, Performance Profiler, Joint Optimizer, and Executor. The runtime layer builds on top of the APIs of the massively task-parallel execution engine Ray [70] for lower level machine resource management, e.g., placing jobs on GPUs, as well as to parallelize our profiling runs. Using two benchmark large-model workloads from DL practice, we evaluate Saturn against several baselines, including an emulation of current practice of manual decisions on SPASE. Saturn reduces overall runtimes by 39% to

50%, which can yield proportionate cost savings on GPU clusters, especially in the cloud. We perform an ablation study to isolate the impacts of our optimizations. Finally, we evaluate Saturn's sensitivity to the sizes of models, workloads, and nodes.

Novelty & Contributions. To the best of our knowledge, this is the first work to unify these three critical requirements of large-model DL workloads for end users: parallelism selection, resource apportioning, and scheduling. By casting the problem this way, we judiciously synthesize key system design lessons to craft a new information system architecture that can reduce user burden, runtimes, and costs via joint optimization in this important analytics setting. Overall, this paper makes the following contributions:

- We formalize and study the unified SPASE problem, freeing end users of large-model DL from having to manually select and tune parallelisms, apportion GPUs, and schedule multi-jobs.
- We present Saturn, a new information system architecture to tackle SPASE that is also the first to holistically optimize parallelism selection and resource apportioning for multi-large-model DL. Saturn employs a generalized profiler to estimate parallelism runtimes and an MILP solver for joint optimization.
- To enable generalized and extensible support for parallelisms, we create the abstraction of User-Defined-Parallelisms (UPPs). UPPs can be used to specify parallelisms as blackboxes in Saturn.
- We perform an extensive empirical evaluation of Saturn on two benchmark large-model DL workloads. Saturn reduces model selection runtimes by up to 50% in some cases.

2 BACKGROUND AND PRELIMINARIES

We provide a brief background on DL model training, which is needed to understand our problem setting.

DL models are directed graphs of operators. Operators are typically parametrized and differentiable. Stages of the model graph (also known as *layers*) are often represented as *matrix multiplies*. Such operations are amenable to GPU acceleration. The DL model parameters are updated in a procedure known as "training". Training involves processing a dataset in a procedure known as minibatch stochastic gradient descent (SGD).

Minibatch SGD is an iterative procedure used to fit a model's parameters based on a training data [34, 37]. Given a dataset consisting of example-label pairs, *minibatches* of data are sampled from the dataset. The model is fed these minibatches to produce predictions. The predictions are compared to the corresponding labels to produce an *error value*. In order to minimize the error value, *backpropagation* is used to compute gradient updates for model parameters. SGD runs this backpropagation procedure iteratively, one minibatch at a time, until the full dataset has been consumed. A full pass over the dataset is known as an *epoch*. Training typically involves multiple such epochs, often dozens. Prediction, gradient computation, and parameter updating all involve heavy compute, adding up to even PetaFLOPs for the training of billion-plus-parameter DL architectures. For this reason, many DL users prefer multi-GPU parallel execution.

Multi-GPU parallelism is now common in large-model DL training [46]. Several parallelization schemes already exist, and researchers continue to routinely devise and propose new techniques. A comprehensive review of all such approaches is out of scope for this paper; we refer interested readers to the relevant surveys [72, 99]. But we highlight 5 of the most prominent approaches: (1) data parallelism, (2) (basic) model parallelism, (3) pipelining, (4) fully-sharded data parallelism, and (5) spilling. We also mention the knobs involved in each parallelism that complicate their scaling behaviors and make theoretical performance analyses challenging.

Data Parallelism replicates a given DL model across multiple accelerators. Each is fed a different minibatch partition for parallel processing. Replica synchronization can be done in two ways. In Parameter Server (PS)-style data parallelism, synchronization occurs through a centralized parent server [58, 93]. All-Reduce data parallelism, as in Horovod or PyTorch DDP, synchronizes replicas through peer-to-peer communication [97] and is generally more popular for GPU execution. Synchronization occurs at the minibatch boundaries, ensuring reproducible and exact execution.

Basic Model Parallelism partitions the *model* rather than the data. The model graph is sharded, and subgraphs are placed on different GPUs to distribute the memory footprint. The speedup potential of model parallelism depends heavily on the partitioning scheme and model architecture. Hand-crafted, architecture-specific approaches can perform well [1], while simple and generic partitioning schemes tend to be slower [71].

Pipelining is a modification of model parallelism in which the model is sharded in a sequential fashion. It partitions a minibatch into smaller "microbatches," then shuttles the microbatches through the model partitions [45, 51, 60, 117]. This enables different model shards to concurrently run different microbatches. The speedup of pipelining is heavily tied to the partitioning scheme and the number of microbatches. Prior work has underscored the importance of tuning these knobs via either expert knowledge or automated heuristics [60].

Fully-Sharded Data Parallelism (FSDP) is a more recent approach that blends model parallelism with data parallelism. Originally introduced in Microsoft's ZeRO [88], it has since been integrated into the PyTorch Distributed package [59]. FSDP partitions a model graph across multiple accelerators, then sends different minibatch partitions to the accelerators. FSDP runs All-Gather on model layers in sequence as data moves through the graph. The currently executing layer group is data-parallel-replicated; the other operators are still distributed in a model-parallel way. FSDP exposes two main user-configured optimizations to reduce GPU memory pressure: (1) gradient checkpointing [26] and (2) DRAM spilling. Turning these knobs on can lower GPU memory pressure at the cost of some performance. Ascertaining when it is worth turning one or both of these techniques generally requires empirical testing.

Spilling is not a parallelism techniques in itself but it is often used as an aid to reduce GPU memory pressure to aid a parallelism. It swaps model shards between GPU memory and DRAM for piecewise GPU-accelerated execution [12, 71]. This adds DRAM-GPU communication overheads, but it enables large models to be trained with even just one GPU. Spilling performance is affected by a *partition count* knob, which determines how many DRAM spills are needed during execution.

Model selection is the process of training and comparing multiple model configurations. Two popular procedures are *grid search*, in

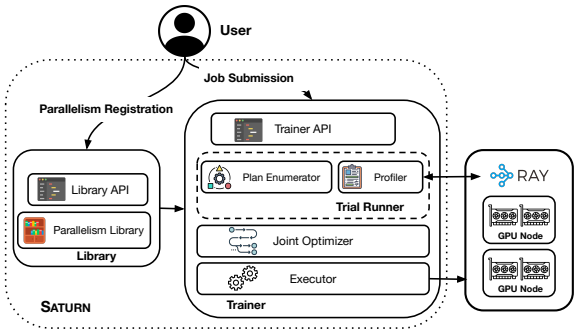

Figure 3: System architecture of SATURN and the interactions between the components.

which all combinations of sets of values of hyper-parameters (e.g., batch size, learning rate) are used, and *random search* [17], in which random hyper-parameter combinations from given intervals are used. Early stopping can reduce the set of configurations during training [56, 57, 90]. The high resource demands of model selection on large models can sometimes cause a DL user to settle for a smaller search space, but that runs the risk of missing out on higher accuracy [34, 53]. Faster execution of such workloads can also empower DL users to run larger searches, in turn helping accuracy. Some prior work employ approximations to model selection via heuristics that alter accuracy properties [80, 86]. We focus on the stronger desideratum of *fidelity* (i.e., no approximations) to give the user full control and simplify reasoning about the system.

3 SYSTEM OVERVIEW

We now describe the architecture of SATURN that meets all the desiderata in Section 1.2. SATURN has 4 main modules, as Figure 3 shows. For workload specification, it exposes a high-level API and the Parallelism Library. The Trial Runner handles runtime estimation. The Joint Optimizer and Executor then tackle the SPASE problem. SATURN uses the lower level APIs of Ray [70] as the runtime layer that places jobs on GPUs. Next, we describe each of SATURN's components.

3.1 Workload Specification

The first phase, workload specification, is handled by our API and the Parallelism Library component.

API. SATURN's API provides an easy-to-use interface for both registering parallelisms (for developers) and submitting large-model training jobs (for end users of DL). We now provide a brief overview of the general usage here; due to space constraints, we provide the full example pseudocode in the technical report [73]. There are two parts to the API: the Library API and the Trainer API. Users create training "Tasks" through the Trainer API by specifying functions for model initialization and data loading, along with any hyper-parameters. This is sufficiently general to cover most model selection workloads. Listing 1 illustrates.

```
from saturn.trainer import Task, HParams, execute, profile

t_1=Task(get_model,get_data,HParams(lr=1e-3,epochs=5,optim=SGD))
t_2=Task(get_model,get_data,HParams(lr=3e-3,epochs=5,optim=SGD))
```

Listing 1: Specifying tasks through SATURN's API.

Training procedures are defined by "User-Pluggable Parallelisms" (UPPs), which implement the parallel execution approach for SGD. These parallelisms can be registered with our Library by a developer (e.g., ML engineer) or a system-savvy end user of DL. The registration process is shown in Listing 2.

```
from saturn.library import register

register("parallelism-a", ParallelismA)
register("parallelism-b", ParallelismB)
```

Listing 2: Parallelism registration.

Once all parallelisms and tasks are specified, DL users can invoke the Trial Runner to produce runtime estimates in a single line of code, followed by invoking the whole training execution in another single line of code. Listing 3 illustrates these.

```
profile([t_1, t_2, t_3])
execute([t_1, t_2, t_3])
```

Listing 3: Profiling and execution invocations.

Parallelism Library. The design of this library is inspired by functional frameworks, user-defined function templates in RDBMSs, and DL model hubs [113]. We follow a define-once, use-anywhere design, wherein registered UPPs can be reused across models, execution sessions, and even different cluster users. This is achieved by managing library-registered parallelisms as a database of code files. The Library allows developers to register new parallelisms by implementing an abstract skeleton, shown in Listing 4.

```
class BaseParallelism:
  def search(task:Task,gpus:List[int])->Dict,float:
    pass
  def execute(task:Task,gpus:List[int],knobs:Dict)->None:
    pass
```

Listing 4: Parallelism specification skeleton.

The *search* function should use the given task and GPUs to provide (1) execution parameters (e.g., microbatch count, partition count) and (2) a minibatch runtime estimate. Knob-optimization can also optionally be tackled here. Failed searches (e.g., OOMs) can be handled by returning null values. The *execute* function trains the provided task to completion using the allotted GPUs. It also uses any execution parameters produced during the search phase to optimize execution.

Developers can implement a UPP with standard DL tool code (e.g., TensorFlow or PyTorch) without restrictions. This enables easy integration of pre-existing parallelisms. Indeed, we validate that functionality by adding 4 major parallelisms in our default Parallelism Library: DDP [59], GPipe-style pipeline parallelism [51], FSDP, and model spilling via the FairScale package [12]. These out-of-the-box parallelisms in SATURN are maximally general in that they can be automatically applied to any DL model supported by them. Implementing UPPs for each took 100 – 250 lines of Python code. Once defined, UPPs can be registered with the Library under a user-set name (e.g. "pytorch-ddp").

The minimality and generality of SATURN's two-function API already supports a variety of parallelisms and searching methods. We believe this system design can help developers retain the familiar DL tool environment without lower level code changes or extraneous workflows to, say, translate their parallelism implementation into a new configuration file format, a custom domain specific language, etc. Overall, our Parallelism Library serves as an organized

roster for registering and using large-model DL parallelisms. While it is a key part of Saturn, it can potentially also be useful as its own standalone tool.

3.2 Performance Estimation

The Trial Runner component estimates runtime performance of model configurations with different parallelisms plus GPU apportionments and generates of statistics. All this information is later used to tackle the SPASE problem. The Trial Runner is *not* a parallelism selector: it simply generates the statistics needed to solve SPASE. It is our empirical substitute for parallelism-specific complex theoretical models used in some prior art [80, 86]. As we noted before, such empirical profiling helps "future proof" Saturn to an extent: by not tightly coupling Saturn to specific parallelisms' theoretical models, we can directly support future DL tool compilers and/or accelerator hardware as they are evolving rapidly. As we highlight in Section 1.2, extensibility is one of our key desiderata. The Trial Runner has two submodules: Plan Enumerator and Profiler.

Plan Enumerator. This sub-module constructs a "grid" across all supported parallelisms and GPU apportionment levels for each model. That represents the space of "physical plans" for every model that will then be profiled to obtain runtime performance estimates.

Profiler. This sub-module takes each physical plan created by the Plan Enumerator and gets local runtime estimates for them to be used later for multi-job optimization. Here we exploit a key property of SGD: since it is iterative and consistent, we can accurately extrapolate a whole epoch's runtime estimate from averaged performance over a just few minibatches. Furthermore, DL parallelism approaches tend to have consistent communication overheads per minibatch [2, 41]. Thus, the different communication costs of each parallelism are naturally accounted for in our empirical profiling. We use Ray to parallelize these profiling runs in a task-parallel way to reduce the Profiler's runtime. In our experiments, profiling twelve 1.5B to 6B parameter models for 4 parallelisms took < 30min. This Profiler overhead is affordable because the actual DL training and model selection, which uses the full training data, can take over dozen hours or even days.

3.3 Joint Optimizer and Executor

The Trial Runner provides the local job runtime statistics. Now we use those statistics to tackle the SPASE problem in a unified manner via holistic optimization.

Joint Optimizer. The Joint Optimizer is invoked transparently when the user invokes the *execute* function. It uses the runtime estimates produced by the Trial Runner and cluster details to produce a full execution plan. This plan bakes in all of *parallelism selection*, *GPU apportionment*, and *schedule construction*. To construct the plan, the Joint Optimizer automatically determines the following for all model configurations given by the user: (1) which parallelism to use, (2) how many GPUs to give it, and (3) when to run it in the schedule.

Our Optimizer is implemented in two layers. First, an MILP solver to produce makespan-optimized execution plans. Second, an introspective, round-based resolver that runs on top of the MILP solver to support dynamic reallocation. Section 4 goes into the technical details of the MILP, why we chose to use an MILP solver instead of heuristics, and additional techniques in the Joint Optimizer.

Executor. This module handles the running of the full execution plan generated by the Joint Optimizer. The Executor runs on top of the lower level APIs of Ray to leverage its task-parallel processing. By default, Ray uses its own task scheduler, and swapping that out for a custom scheduler is challenging. So, for the Executor we implement our plan *over* Ray's scheduler. We achieve this by "tainting" Ray-owned GPUs so that they can only be used by the corresponding jobs from our pre-calculated schedule. Thus, the Executor *adapts* Ray to support our execution plan.

3.4 Current Limitations

Saturn supports both single-node and multi-node training across different models, but in the current version we focus on the case where each model fits in aggregate cluster memory (i.e., total GPU memory + DRAM). We also focus on the homogeneous GPU cluster setting and leave to future work adding support for heterogeneous hardware clusters, hardware type selection, and elastic provisioning (e.g., like in [64, 76]). Anecdotally, we find that many DL users in domain sciences and enterprises do indeed fit this setting. Furthermore, many of the large-model parallelisms added to our Library do not yet support cross-node training for a single model out of the box. So, we defer that support to a future extension as those parallelisms evolve. In spite of these assumptions, Saturn can already train 10B+ parameter models on even just one node. These limitations can be mitigated in the future as follows: (1) adjust the MILP in Section 4 to include hardware selection, (2) give the Trial Runner a larger space to explore, and (3) add cross-node model parallelism tools to the Library [122].

4 SPASE JOINT OPTIMIZER

We now describe the SPASE problem and then dive into our MILP formalization. Using an in-depth simulation study, we then evaluate an MILP solver (Gurobi [39]) against baselines and strong heuristics from standard practice and prior art. We then explain our introspective re-scheduling mechanism on top of the MILP solution that enables Saturn to adaptively reassess its decisions over time.

4.1 Problem Basics

SPASE unifies parallelism selection, resource allocation, and schedule construction. Typical schedulers can set task start times, while resource schedulers can select a GPU apportionment as well. But with SPASE, our joint optimizer must consider a third performance-critical dimension: select the parallelism to use for each model on the allotted GPUs. To the best of our knowledge, our is the first work to unify and tackle this joint problem.

In model selection workloads, it is common for all jobs to be given up front. So, we focus on that setting. Using the Trial Runner module, we generate the necessary runtime statistics for all given jobs. But even with that information, the joint problem is intractable; prior work on network bandwidth distribution [10] has shown that even the basic resource allocation problem is NP-hard. SPASE is a more complex version of that problem that also handled parallelism

selection and makespan-optimized scheduling; so it is also NP-hard. As such, solving it optimally is ruled out.

Brute-forcing the search space is also impractical due to its sheer size. The number of schedule orderings alone grows super-exponentially with the number of jobs [103]. Including all possible apportionments and parallelism selections further enlarges that search space. Even if we reduce the search space via some heuristics, it will likely remain impractical for us to enumerate every possible solution in a brute-force approach for many workloads. Thus, we choose to formulate SPASE as an MILP and use an industrial-strength MILP solver (Gurobi [39]) to leverage its time-tested optimization power.

Later in Section 4.3, we justify this decision further using a simulation study. We find that the MILP solver significantly and consistently outperforms known baselines and strong heuristics despite its time limit. To the best of our knowledge, ours is the first MILP formulation to unify DL parallelism selection, resource allocation, and scheduling. Not only does it enable us to state the problem with mathematical precision, it also enables us to explore the problem space's intricacies via the simulation study.

4.2 MILP Formulation

Our MILP input consists of a full grid of all given parallelisms crossed with every possible GPU apportionment, as well as the corresponding runtime estimates for each of those combinations applied to each given model. Table 2 lists our notation. As noted in Section 3, our empirical runtime estimates already bake in the unique communication overheads of each parallelism naturally. The MILP selects the parallelism+apportionment combination, or "configuration" for each model, such that the results tell us precisely how many GPUs and which parallelism to use for each model. Our MILP's objective is to minimize *makespan*, as shown in Equation 1.

$$\texttt{Objective:} \quad \min_{B,O,P,A,I} C \tag{1}$$

We now define the constraints. Equation 2 defines the makespan; it is the latest task's start time plus the runtime of that task's selected parallelism+apportionment.

$$\begin{aligned} C \geq I_{t,n,g} + R_{t,s} - U \times (1 - B_{t,s}) \\ \forall s \in S_t \forall t \in T, \forall n \in N, \forall g \in G \end{aligned} \tag{2}$$

Next, each task must use only one parallelism+apportionment and must use only one node's GPUs.

$$\sum_{x \in B_t} x = 1; \sum_{y \in O_t} y = 1 \tag{3}$$

Next, we enforce GPU requests of the solver onto the plan. We must also ensure that tasks are not given GPUs on unselected nodes. Equations 4, 5, 6, and 7 enforce these constraints.

$$\sum_{t \in P_{t,n}} t \geq G_{t,s} - U \times (2 - O_{t,n} - B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{4}$$

$$\sum_{t \in P_{t,n}} t \leq G_{t,s} + U \times (2 - O_{t,n} - B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{5}$$

$$\sum_{t \in P_{t,n}} t \geq 0 - U \times (O_{t,n} + B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{6}$$

Table 2: MILP Notation used in Section 4.2

Inputs to the MILP	
Symbol	Description
N	List of nodes available for execution.
T	List of input training tasks.
U	Large integer value used to enforce conditional constraints.
GPU_n	The number of GPUs available on node n.
S_t	Number of configurations available to task t. A configuration consists of both a parallelism and a GPU allocation.
$G_t \in \mathbb{Z}^{+S_t}$	Variable length list of requested GPU counts for each configuration of task t.
$R_t \in \mathbb{R}^{+S_t}$	Variable length list of estimated runtimes for each configuration of task t.
MILP Selected Variables	
Symbol	Description
C	Execution schedule makespan.
$B_t \in 0, 1^{S_t}$	Variable length list of binary variables indicating whether task t uses the corresponding configuration from S_t.
$O_{t,n} \in 0, 1$	Binary indicator of whether task t ran on node n.
$P_{t,n,g} \in 0, 1$	Binary indicator of whether task t ran on GPU g of node n.
$A_{t1,t2} \in 0, 1$	Binary indicator of whether task $t1$ ran before task $t2$. If $A_{t1,t2}$ is 1, $t2$ must have run after $t1$.
$I_{t,n,g} \in \mathbb{R}^+$	Start time of task t on GPU g of node n.

$$\sum_{t \in P_{t,n}} t \leq 0 + U \times (O_{t,n} + B_{t,s}) \forall s \in S_t, \forall t \in T, \forall n \in N \tag{7}$$

Next we apply a *gang scheduling* constraint. For each task, all assigned GPUs must initiate processing simultaneously. Formulating this constraint is challenging — we need consistency over a set of MILP-selected values, on a set of MILP-selected indices, across an MILP-selected gang size. Our solution is to find a fixed start-time target — the sum of MILP-selected start times over *all GPUs*, divided by the number of allocated GPUs. This naturally encourages the solver to fix start times on unused GPUs to 0 without explicit enforcement. Equations 8 and 9 enforce this constraint.

$$\begin{aligned} \frac{\sum_{x \in I_{t,n}} x}{G_{t,s}} \leq I_{t,n,g} + U \times (3 - P_{t,n,g} - B_{t,s} - O_{t,n}) \\ \forall s \in S_t, \forall t \in T, \forall g \in GPU_n, \forall n \in N \end{aligned} \tag{8}$$

$$\begin{aligned} \frac{\sum_{x \in I_{t,n}} x}{G_{t,s}} \geq I_{t,n,g} - U \times (3 - P_{t,n,g} - B_{t,s} - O_{t,n}) \\ \forall s \in S_t, \forall t \in T, \forall g \in GPU_n, \forall n \in N \end{aligned} \tag{9}$$

Finally, we encode a task isolation constraint, ensuring that no tasks overlap on the same GPU. Equation 10 provides this guarantee

if task $t1$ came before task $t2$, while equation 11 guarantees no overlap if task $t1$ came after task $t2$.

$$I_{t1,n,g} \leq I_{t2,n,g} - R_{t,s} + U \times ((3 - P_{t1,n,g} - P_{t2,n,g}) - B_{t,s} + A_{t2,t1}) \quad \forall s \in S_t, \forall t1 \in T, \forall t2 \in (T - \{t1\}), \forall g \in GPU_n, \forall n \in N \tag{10}$$

$$I_{t1,n,g} \geq I_{t2,n,g} + R_{t,s} - U \times ((4 - P_{t1,n,g} - P_{t2,n,g}) - A_{t2,t1} - B_{t,s}) \quad \forall s \in S_t, \forall t1 \in T, \forall t2 \in (T - \{t1\}), \forall g \in GPU_n, \forall n \in N \tag{11}$$

This MILP formulation is complex because it spans and unifies three different system decisions in our setting. Our Joint Optimizer constructs all the constraints automatically for a given instance and provides them to Gurobi [39]. We use the PuLP interface for Gurobi to keep all variables within a single Python process space.

4.3 Simulation-based Comparisons

We now evaluate our MILP-solver approach. We begin by discussing baselines inspired by current practice and heuristics in prior art. Then, we run comprehensive evaluations on simulated workloads. We find that the MILP-solver approach outperforms the other approaches by a significant margin.

4.3.1 ***Baselines.*** As the case study in Section 1 highlighted, large-model DL users must currently tackle the SPASE problem manually. So we can define the initial baseline based on current best practices. A common heuristic is to just maximize each task's allocation. Each task is given all GPUs in a node; then the best parallelism for that particular setting is applied. The models are run one after another. This optimizes *local* efficiency and maximizes available GPU memory for each task. Although this heuristic eliminates in-node task parallelism, it becomes a suboptimal degenerate case of the apportioning and scheduling parts of the SPASE problem. We call this baseline "Max-Heuristic" and anecdotally we find this is common in current practice.

The opposite extreme would be to minimize the number of GPUs assigned to each task to maximize task-parallelism [71]. We call this baseline "Min-Heuristic." While it runs many models in parallel, this approach suffers a lot of DRAM spilling for large models.

Finally, we devise a strong algorithmic heuristic that incorporates our runtime estimates to produce non-trivial solutions. It extends an idea from Optimus, a DL resource scheduler in prior art that proposed a greedy resource allocator that uses an "oracle" to provide runtime estimates [80, 86]. Our Trial Runner statistics serve as our oracle. The heuristic iteratively assigns GPUs to whichever model that will see the greatest immediate benefit. We call this algorithm Optimus-Greedy. Algorithm 1 presents its pseudocode, reusing some variables from Table 2.

The Optimus-Greedy algorithms yields resource allocations per task. We transform that into a SPASE solution by selecting the best parallelism for each task's allocation post-hoc. In the multi-node case, we run this algorithm one node at a time. Like many iterative greedy algorithms, this approach relies on consistent scaling behaviors. It has only a local greedy view (e.g., how will we benefit from allocating this *one* GPU) rather creating a one-shot global distribution.

Algorithm 1 : **Optimus-Greedy**(Tasks T, GPUs G)

```
L = [1|t ∈ T]
while sum(L) < G do
    CR = [R_{t,s}|t, l ∈ (T, L), s ∈ S_t where G_{s,t} == l]
    PR = [R_{t,s}|t, l ∈ (T, L), s ∈ S_t where G_{s,t} == l + 1]
    GAIN = [c − p|c, p ∈ (CR, PR)]
    L[ArgMax(GAIN)] + +
end while
return L
```

Apart from the above three approaches to cover standard practice and prior art extensions, we also include a simple randomization-based baseline. In summary, we compare with 4 approaches:

(1) **Max-heuristic:** All GPUs within a node are given to one task at a time.
(2) **Min-heuristic:** A single-GPU technique (spilling) is given to each task to maximize task parallelism. If additional GPUs are available, they are divided evenly.
(3) **Optimus-Greedy:** A greedy algorithm inspired by the one used in the Optimus [80] resource scheduling paper.
(4) **Randomized:** Parallelisms and allocations are randomly selected for every task, then tasks are randomly scheduled.

For each of the above approaches, we use our Profiler results to select the best possible parallelism+allocation for each model. For instance, if a baseline determines that Model A should receive 8 GPUs, we refer to the Profiler to determine which parallelism gives Model A the best runtime at 8 GPUs. This same best-check procedure is used to determine the gain values for Optimus-Greedy.

Since our MILP is complex, Gurobi is unlikely to converge to an optimal solution within a practical timeframe. Thus, we set a reasonable timeout (e.g., 5 minutes) for the solver to try and produce a "good enough" solution. This overhead will be negligible given the length of actual DL training.

4.3.2 ***Simulation Workloads.*** We create simulations based on 2 real benchmark DL workloads, described in Table 3. Runtime estimates for all parallelism+allocation+model combinations are produced through our Trial Runner prior to the simulations. We simulate 3 hardware settings: an 8-GPU single-node, a 32-GPU 4-node, and a heterogeneous 4-node with GPU counts of 2, 2, 4, and 8 (16 GPUs in total). To adapt the baselines for the heterogeneous setting, we distribute models across nodes randomly, weighting each node's probability by its GPU count. We set the Gurobi timeout to 5min; we find this suffices for practical DL use cases. We leave it to future work to set the timeout adaptively given a workload.

Figure 4 presents the simulation results. All approaches are run 3 times and averaged, with 90% confidence intervals displayed; only randomized algorithm shows significant non-determinism on the homogeneous node settings. In all cases, our MILP-solver approach yields significantly better solutions than all 4 baselines/heuristics. We achieve a makespan reduction of up to 59% over the Min-Heuristic, 36% over the Max-Heuristic, 54% over the randomized, and 33% over Optimus-Greedy. In the heterogeneous setting, the improvements are somewhat lower, ranging from 18% to 42%. This is due to the smaller 2-GPU nodes, which provide little flexibility for resource apportioning or parallelism selection, thus reducing the

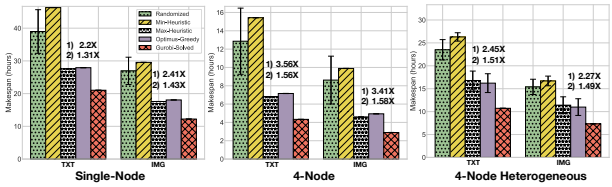

Figure 4: Saturn Simulation results comparing our MILP to two key baselines. For each group, we list Saturn's speedup versus (1) the weakest and (2) the second-best performer.

relative differences between approaches. Overall, Saturn's Gurobi-solved approach performs better consistently. Of course, the solver takes the longest to run: 5min timeout as against < 10 seconds for the other 4 approaches. But that optimization overhead is negligible given the makespan of multiple hours.

4.4 Introspection

In general, one-shot up-front scheduling is suboptimal. Workloads can evolve over time, either due to online changes (e.g., an AutoML heuristic killing or adding models to train) or ongoing execution (task runtime reduce as they are trained). If the optimizer can be rerun partway through execution, it might produce a different, more performant, solution for the remainder of the workload. To achieve this, we propose the use of *introspection* [114].

A key feature in some state-of-the-art DL schedulers [114], introspection proposes that a scheduler should "learn" as it executes. There are two ways in which a schedule might be altered or adapted via introspection. First is pre-emption. Rather than blocking a GPU for a full job lifecycle, jobs can be swapped to different GPUs or paused temporarily. This enables fine-grained schedule construction and increased optimization flexibility. Second is dynamic rescaling. The initial up-front training plans could be adjusted (e.g., 6 GPUs down to 2) partway through a schedule. In SPASE, this can also involve changing the parallelism.

We now describe how we implement introspection in Saturn. Figure 5 illustrates our design. We treat our SPASE MILP solver as a blackbox sub-system. At periodic intervals (e.g., every 1000 seconds), we re-evaluate the underlying workload. The partial training over the previous interval may have modified the set of models. We *rerun* the solver on the interval boundaries so that it can introspectively adjust its original solution. By treating each interval-defined segment of training as effectively independent, we preserve gang scheduling semantics *within* each segment, while allowing for graceful exits and relaunches across intervals. Such sequences of independent segments are possible due to the iterative nature of SGD, involving a sequence of minibatches [34], as well as the ease of checkpointing models during training [86]. Due to space constraints, we provide the full pseudocode of our approach in the appendix of the extended technical report [73].

To demonstrate the impact of Saturn's introspection, we compare with a new dynamic baseline, "Optimus-Dynamic", by swapping the MILP-solver for the Optimus-Greedy algorithm. Figure 6 shows the impact of the interval length and the improvement threshold knob. Since each round produces a holistically optimized solution, Saturn's performance improves monotonically (not accounting for pre-emption costs) as knobs become more fine-grained.

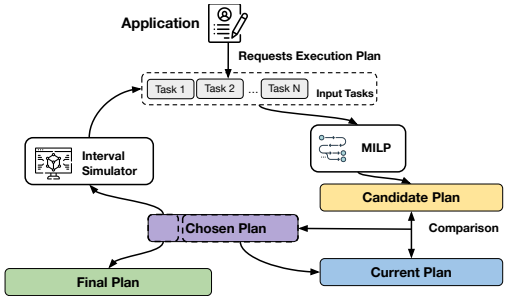

Figure 5: Illustration of the introspective feedback loop used on top of our MILP solver.

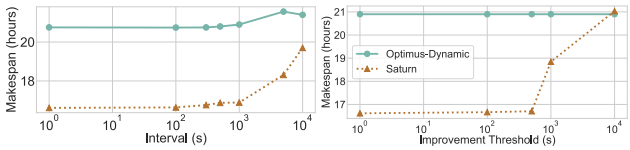

Figure 6: Sensitivity plots for Saturn and Optimus-Dynamic against interval and threshold knobs. We fix the interval to 1000s for the threshold analysis and the threshold to 500s for the interval analysis.

Lower interval/threshold levels naturally subsume higher levels in this scheme. In contrast, locally-optimizing algorithms such the Optimus-Dynamic approach have non-monotonic behaviors.

Introspection does not have to occur on interval completion; we can actually simulate the workload state at the next interval using the current schedule. Then, the solving process for the next introspection round can be overlapped with execution of the current round to hide the latency of introspection. This scheme provides speedups of 15% to 20% over our one-shot MILP, as shown in Section 5.2. With introspection plus our MILP-solver, Saturn's Joint Optimizer is 1.5x-4.1x faster than the heuristics described in Section 4.3. Our introspection optimization significantly improves offline execution, but it also naturally supports online AutoML optimizations such as early-stopping through workload reassessment [56, 57]. We do not explicitly optimize for AutoML heuristics in the current version of Saturn; but it is easy to extend it to exploit this optimization for those model selection workloads.

We note that our introspection optimization takes inspiration from prior art in the DL cluster scheduling literature. Antman [115] and Gandiva [114] showed how pre-emption on minibatch boundaries can improve end-to-end efficiency. Pollux and Optimus [80, 86] showed the value of dynamic rescaling. Our contribution is in unifying both of those optimization ideas to craft our introspection technique, which also considers changing the parallelism selection in each round.

5 EXPERIMENTAL EVALUATION

We now run an extensive empirical evaluation. We aim to answer two questions: (1) What performance benefits does Saturn provide compared to current practice? (2) How much do each of Saturn's optimizations contribute to the overall speedups?

Table 3: Model selection configurations of workloads.

Workload	Model Selection Configuration					# Models
	Model Arch. (params)	Dataset	Batch Size	Learning Rate	Epochs	
TXT	GPT-2 (1.5B), GPT-J (6B)	WikiText-2	{16, 32}	{1e-5, 1e-4, 3e-3}	10	12
IMG	ViT-G (1.8B), ResNet (200M)	ImageNet	{64, 128}	{1e-5, 1e-4, 3e-3}	10	12

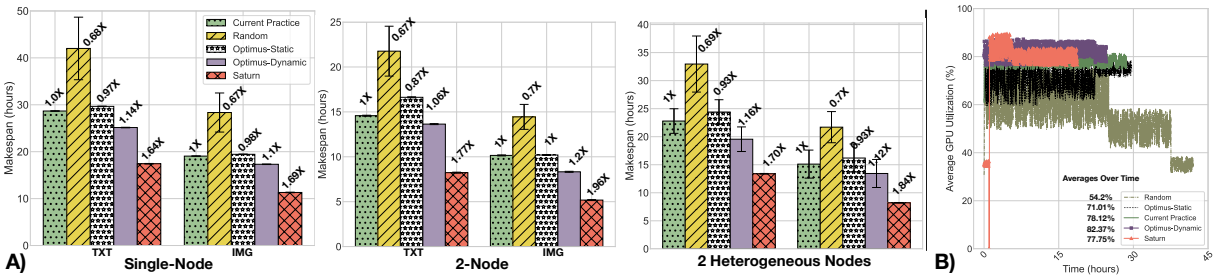

Figure 7: (A) End-to-end runtimes. Speedups versus current practice are also noted. Results are averaged over three trials, with the 90% confidence interval displayed. (B) Average GPU utilization over time at a 100s sampling rate on the single-node TXT job.

Workloads, Datasets, and Model Configurations: We run 2 model selection workloads with benchmark DL tasks. Table 3 lists the model selection configurations (neural architectures and hyperparameters compared) for both workloads. The first (TXT) is a text workload with LLMs. It uses the popular *WikiText-2* [69] dataset. WikiText-2, which is composed of sentences drawn from Wikipedia, has previously been used as a benchmark on landmark LLMs such as GPT-2 [87]. TXT uses two GPT models: GPT-2 (1.5B parameters), introduced in 2019, and GPT-J (6B parameters), introduced in 2021. Both are still considered state-of-the-art LLMs for application-specific finetuning purposes.[1] The second (IMG) is image classification comparing a large ResNet deep CNN (200M parameters) and a large-scale Vision Transformer (1.8B parameters). It uses the computer vision benchmark dataset *ImageNet* [28] (14M images and 1000 classes). IMG also illustrates a workload wherein the model types differ substantially, motivating usage of different parallelisms.

Software Setup: All models are implemented and trained with PyTorch 2.0. We register 4 parallelisms in Saturn for search and evaluation.

(1) PyTorch Distributed Data Parallelism [59].
(2) PyTorch Fully-Sharded Data Parallelism [59].
(3) GPipe, adapted from an open-source implementation [51].
(4) Model spilling, provided by the FairScale library [12].

We use Gurobi 10.0 for our SPASE MILP-solver; the introspection threshold and interval parameters are set to 500s and 1000s, respectively. For the underlying job orchestration, we use Ray v2.2.0. Datasets are copied across nodes upfront.

Hardware Setup: We configure 3 hardware settings: (1) 8-GPU single-node, (2) 16-GPU 2-nodes, and (3) heterogeneous 2-nodes, where one node has 8 GPUs and the other has 4 (12 GPUs in total). All settings use machines with 1152 GB RAM, Intel Xeon 3.0 GHz 2nd Gen CPUs, a 500 GB NVMe drive, and 40 GB memory A100 GPUs connected via NVSwitch.

[1]Extreme-scale LLMs such as GPT-3 or BLOOM are too large (175B+ parameters) for our current scope because they need hundreds of GPUs for reasonable runtimes, which we are unable to afford. They are likely also infeasible and/or an overkill for most end users of DL, especially in domain sciences, small companies, etc., who are our main target. We leave it to future work to extend Saturn to such extreme-scale models.

Baselines: No prior end-to-end system can solve the SPASE problem; prior art either fails to support large models or else fails model selection constraints, as Table 1 showed. So, we compare Saturn with 4 baselines by adapting the approaches compared in Section 4.3.

(1) **Current Practice:** A heuristic without any task parallelism within nodes. It allocates 8 GPUs per task. Parallelism selection is set by a human to "optimal" choices for an 8-GPU allocation, (typically FSDP). This is perhaps most representative of current practice by end users of DL.
(2) **Random:** A randomizer tool selects parallelism and apportioning and then applies a random scheduler. This represents a system-agnostic user.
(3&4) Two modified versions of Optimus-Greedy (Algorithm 1) combined with a randomized scheduler (see Section 4). We name these baselines **Optimus-Dynamic** and **Optimus-Static**, respectively. These are the strongest baselines for large-model DL model-selection we could assemble by building on prior art.

The above approaches cover both current practices and reasonable strong heuristics for our problem setting. We note that the two Optimus-based baselines use our Trial Runner as an oracle for their runtime estimates (the original Optimus paper only had runtime models for Parameter Server-style data parallelism [80]). This highlights the novelty of our problem setting — the strongest baseline from prior art needs to reuse a module of our system.

5.1 End-to-End Results

Model Selection Runtimes: We first compare the end-to-end runtimes versus the 4 baselines. The Trial Runner search overheads are *included* in Saturn's runtime. Figure 7(A) presents the results.

Saturn achieves significant speedups versus all baselines. In the single-node setting, we achieve about 40% lower makespan than the strongest baseline on both workloads. The gap grows to >44% on the 2-node setting and improves further in the heterogeneous setting. Against Current Practice, Saturn achieves 1.64x to 1.96x speedups overall. Since the same UPP implementations are used *in all cases*, the speedups are achieved purely via the better decisions on parallelism selection, resource allocation, and schedule construction. Also note that all compared approaches (including Saturn) use logically equivalent SGD and offer the same accuracy.

Figure 7(B) plots the GPU utilization. Saturn achieves good GPU utilization throughout, except an initial low-utilization period for the Trial Runner's search and the MILP solving period. But GPU utilization alone can be misleading; tools such as nvidia-smi report full utilization whenever a GPU is active even if not all cores are being actually used [3]. Since such inefficient work can artificially inflate utilization, these GPU utilization results should not be taken as a measure of training performance in isolation.

Overall, Saturn reduces model selection runtimes substantially for all workloads in all evaluated settings. We note that Saturn also offers more qualitative benefits to end users of DL because they are freed from having to manually select parallelisms, decide resource allocations, or tune system parameters.

Intuition on Efficiency Gains. Saturn's performance improvements arise due to its *holistic* optimization approach. To the best of our knowledge, this is the first work that characterizes the parallelism performance crossovers and incorporates them into a joint optimizer. Our empirical profiler and unified SPASE formulation enable us to optimize in a parallelism-agnostic fashion. The heuristic and algorithmic baselines make assumptions about scaling behaviors (e.g., consistency, linear scaling, etc.) that do not always hold up in large-model DL practice. To prove our point further, Table 4 lists the parallelisms+allocations selected by Saturn for a few models from the single-node workloads. We see a non-trivial mixture of decisions across the models trained.

Table 4: Parallelisms and apportionments chosen by Saturn for a few evaluated models.

Model Config	Parallelism	Apportionment
GPT-2 (Batch 16, 1e-5 LR)	Pipelining	5 GPUs
GPT-2 (Batch 32, 1e-4 LR)	FSDP	4 GPUs
GPT-J (Batch 16, 1e-5 LR)	FSDP	8 GPUs
GPT-J (Batch 32, 1e-4 LR)	Pipelining	3 GPUs
ResNet (Batch 64, 1e-4 LR)	DDP	2 GPUs
ResNet (Batch 32, 1e-4 LR)	Spilling	1 GPU
ViT-G (Batch 32, 1e-4 LR)	FSDP	4 GPUs
ViT-G (Batch 16, 1e-4 LR)	FSDP	6 GPUs

Saturn's MILP-chosen SPASE solutions may seem unintuitive individually, but they combine together in the context of a multi-model schedule to maximize efficiency and minimize end-to-end runtimes. It is unreasonable to expect DL practitioners to have the systems knowhow to manually pick such a complex mix of parallelisms and GPU apportionments. Thanks to our unified data systems-style approach, Saturn frees DL users to focus on their DL goals instead of such tedious systems-level decisions.

5.2 Drilldown Analyses

5.2.1 ***Ablation Study.*** We now separate our optimizations into 4 layers: scheduling, resource allocation, parallelism selection, and the introspection overlay. We apply these one-by-one to understand how much they contribute to Saturn's gains as follows.

(1) First, a version without any of our optimizations. FSDP is applied with both checkpointing and offloading on (non-expert config), resource allocations are set manually to 4 GPUs per task, and a random scheduler is used.
(2) Second, we replace the random scheduler with our makespan-optimized scheduler.
(3) Third, we reintroduce resource apportioning into our MILP.
(4) Fourth, we allow for the automatic parallelism selection and knob tuning.
(5) Finally, we overlay introspection on solver. This final optimization completes Saturn.

We use the single-node TXT workload in our study. Table 5 illustrates the performance gains achieved through each optimization.

Table 5: Ablation study.

Optimizations	Abs. Speedup	Extra Speedup
Unoptimized	1.0X	1.0X
+ MILP Scheduler	1.1X	1.1X
+ Resource Allocation in MILP	1.33X	1.2X
+ Auto. Parallelism Selection	1.95X	1.47X
+ Introspection	2.27X	1.16X

The scheduler-only MILP provides better packing for some initial makespan improvements. Adding in resource apportioning lets the solver reshape task runtimes and demands to produce more speedups. Automatic parallelism selection creates even more flexibility and adds in knob-tuning to improve parallelism performance. Introspection enables the solver to reassess its solution and adapt to shifts in the workload to cap off Saturn's speedups.

5.2.2 ***Sensitivity Analyses.*** We test Saturn's sensitivity to the size of: (1) workloads, (2) models, and (3) clusters.

For the workload size scaling, we run TXT on a single 8-GPU node, fix the model to GPT-2, batch size to 16, and vary the number of learning rates explored. Figure 8(A) presents the results. Saturn scales slightly superlinearly, mainly because of the higher scope of overall optimization with more models (compared to fewer models). This suggests good performance on large workloads, even as computational demands and complexity increase.

Next, we vary model size. We run TXT on a single 8-GPU node with batch size set to 16 and learning rate set to 1e-5. All models are versions of GPT-2. We vary model size by stacking more Transformer encoder blocks, akin to what GPT-3 does [22]. Figure 8(B) presents the results. Saturn achieves mostly linear scaling, but slight slowdowns manifest on the largest model sizes. This is because the largest models force the optimizer to use the currently only viable configuration (8-GPU FSDP with checkpointing and offloading) for every model.

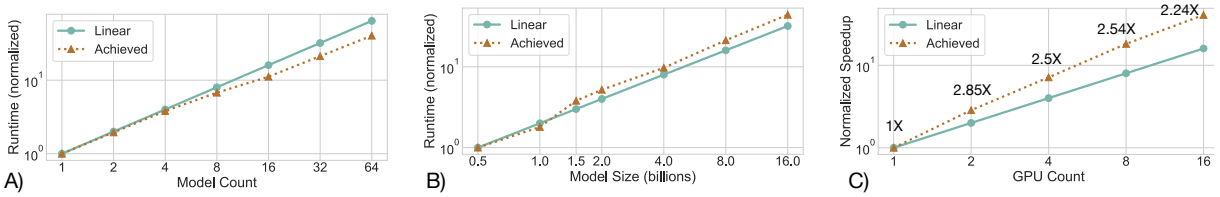

Figure 8: Saturn sensitivity plots on the TXT workload versus (A) workload size, (B) model size, and (C) node size. Charts are in log-log scales, normalized to the initial setting. For part (C), we label each setting with the speedup versus the previous setting.

Finally, we vary the number of GPUs visible to Saturn. We switch to 2 nodes for the 16-GPU case. We use TXT for this experiment. Figure 8(C) presents the results. Saturn actually achieves superlinear speedups, mainly due to 2 reasons. First, the single-GPU case necessarily uses DRAM spilling, while as the number of GPUs increases, the amount of spilling needed keeps dropping and a larger space of parallelisms opens up. Second, larger GPU counts enlarge the solution space for the MILP, with more combinations of parallelisms and apportionment for it to choose from.

6 RELATED WORK

Saturn's focus on the unified SPASE problem is a first for large-model DL workloads. Section 1.1 and Table 1 already positioned the novelty of our problem setting against prior art and the rationales for our system design. We now elaborate more on the relationships with prior art on each of the aspects our work unifies. We also offer more discussion of the wider DL systems landscape to further contextualize our work in the technical report appendix [73].

Parallelism Selectors and Hybridizers: Paleo [84] focused on performance models for data parallelism and model parallelism. But the DL parallelism landscape has changed a lot since then (2016), with numerous new approaches. While their approach can be extended to newer parallelisms, our empirical Trial Runner approach is more easily extensible and highly general. Alpa, FlexFlow, and Unity [49, 104, 124] focus on generating bespoke parallelism strategies for model architectures through complex search procedures. They can produce efficient *single-model plans*, but the cumulative search overheads can get high when applied repeatedly to multi-model training. They also leave a lot on the table by ignoring the bigger picture of multiple models being trained in model selection workloads. In addition, some approaches (e.g., FSDP's offloading, spilling) are not supported by these hybrid-searchers. Overall, they are all *complementary* to our goals and they can be integrated into Saturn under our UPP abstraction in the future if needed.

DL Model Selection Systems: Our work expands a recent line of work on optimized systems for DL model selection, including Cerebro [54], Hyperband [57], and ASHA [56]. However, none of them support large-model DL. Cerebro hybridizes task- and data-parallelism to train multiple DL models in parallel on sharded data. Our work is *complementary* to Cerebro; one can integrate their approach to scale to larger data into Saturn. Hyperband reallocates training resources (e.g., number of epochs) across tasks based on convergence behaviors. ASHA extends Hyperband to also reallocate hardware resources such as GPUs. Both of them are essentially task-parallel model selection heuristics; could support them on top of Saturn. We leave such integrations to future work.

DL Cluster Schedulers: Schedulers such as Gandiva, Apollo, Tiresias, and Antman target a different setting than ours [13, 21, 38, 114, 115] and require the user to specify GPU-counts manually. Gandiva does offer opportunistic rescaling for elastic jobs, but it does so opportunistically without knowledge of the model's scalability. These systems and other orchestrators like Pathways [15] tackle systems challenges that arise with very large clusters. Our focus is *complementary* in that we aim to free end users of DL from needing to hand-tune systems factors. One could potentially integrate Saturn with such larger schedulers, with their cluster manager allocating a set of nodes for Saturn to work with for multi-large-model DL workloads. Gavel [76] schedules over heterogeneous mix of resources, which is outside our current scope, but they do not tackle the SPASE problem. Their performance metrics to handle heterogeneity can potentially be used in a future Saturn extension.

DL Resource Schedulers: Pollux and Optimus [80, 81, 86] tackle apportionment and scheduling, two parts of SPASE. But they do not support larger-than-GPU-memory models, whose complex parallelisms alter apportionment and scheduling tradeoffs in non-trivial ways, as our work shows. They also do not target model selection workloads and focus on throughput, while makespan is better suited for our setting. They also alter model accuracy, violating our fidelity desideratum. A config input to Pollux (e.g., batch size X and learning rate Y) may yield very different accuracy than the same X and Y used for regular training. Themis [66] studies scheduling fairness for ML jobs from different users; their goal and setting are orthogonal to ours in that we focus on model selection jobs from the same user and need to optimize for makespan.

7 CONCLUSIONS AND FUTURE WORK

Finetuning of pre-trained large DL models is increasingly common in DL practice. But navigating the complex space of model-parallel training is unintuitive for regular DL users even though it is needed to reduce runtimes and costs. The complex interplay of parallelism selection with model selection workloads, which requires decisions on resource apportioning and scheduling, can also lead to high resource wastage if not handled well. This work resolves these issues by formalizing the joint SPASE problem unifying large-model parallelism selection, resource apportionment, and scheduling and designing a new information system architecture we call Saturn to tackle SPASE. With user-friendly APIs, joint optimization, and a judicious mix of systems techniques, Saturn reduces large-model DL model selection runtimes by 39-50% over current practice, while freeing DL users from tedious systems-level decisions. Overall, Saturn offers maximal functionality in a critical DL setting, while promoting architectural simplicity to ease real-world adoption.

REFERENCES

[1] 2020. State-of-the-Art Language Modeling Using Megatron on the NVIDIA A100 GPU. https://developer.nvidia.com/blog/language-modeling-using-megatron-a100-gpu/.

[2] 2021. Fully Sharded Data Parallel: faster AI training with fewer GPUs. https://engineering.fb.com/2021/07/15/open-source/fsdp/.

[3] 2023. nvidia-smi (1) User's Manual.

[4] Accessed January 31, 2021. XLA: Optimizing Compiler for Machine Learning : TensorFlow. https://www.tensorflow.org/xla

[5] Accessed May 24, 2023. 2023 State of Data + AI. https://www.databricks.com/sites/default/files/2023-05/databricks-2023-state-of-data-report.pdf

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[7] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2020. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. https://doi.org/10.48550/ARXIV.2011.05497

[8] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[9] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2022. Heterogeneous Acceleration Pipeline for Recommendation System Training. https://doi.org/10.48550/ARXIV.2204.05436

[10] Akashdeep, Karanjeet Kahlon, and Harish Kumars. 2014. Survey of scheduling algorithms in IEEE 802.16 PMP networks. *Egyptian Informatics Journal* 15 (03 2014). https://doi.org/10.1016/j.eij.2013.12.001

[11] Joy Arulraj. 2022. Accelerating Video Analytics. *SIGMOD Rec.* 50, 4 (jan 2022), 39–40. https://doi.org/10.1145/3516431.3516442

[12] FairScale authors. 2021. FairScale: A general purpose modular PyTorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale.

[13] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 505–513.

[14] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. 2018. Online Job Scheduling in Distributed Machine Learning Clusters. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 495–503. https://doi.org/10.1109/INFOCOM.2018.8486422

[15] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. https://doi.org/10.48550/ARXIV.2203.12533

[16] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[17] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

[18] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthoer, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqi, and Sebastian Benjamin Wrede. 2019. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. https://doi.org/10.48550/ARXIV.1909.02976

[19] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V. Evfimievski, and Prithviraj Sen. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. https://doi.org/10.48550/ARXIV.1801.00829

[20] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2022. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG]

[21] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for {Cloud-Scale} Computing. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 285–300.

[22] 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. https://doi.org/10.48550/ARXIV.2005.14165

[23] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2021. THIA: Accelerating Video Analytics using Early Inference and Fine-Grained Query Planning. https://doi.org/10.48550/ARXIV.2102.08481

[24] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3514221.3517857

[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[26] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. https://doi.org/10.48550/ARXIV.1604.06174

[27] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, Yong Wu, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. 2021. Accelerating SLIDE Deep Learning on Modern CPUs: Vectorization, Quantizations, Memory Optimizations, and More. https://doi.org/10.48550/ARXIV.2103.10891

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[29] Aditya Desai, Yanzhou Pan, Kuangyuan Sun, Li Chou, and Anshumali Shrivastava. 2021. Semantically Constrained Memory Allocation (SCMA) for Embedding in Efficient Recommendation Systems. https://doi.org/10.48550/ARXIV.2103.06124

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

[31] 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. https://doi.org/10.48550/ARXIV.2010.11929

[32] Vahid Faghihi, Kenneth Reinschmidt, and Julian Kang. 2014. Construction scheduling using Genetic Algorithm based on Building Information Model. *Expert Systems with Applications* 41 (11 2014), 7565–7578. https://doi.org/10.1016/j.eswa.2014.05.047

[33] William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

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

[35] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2736–2739. https://doi.org/10.1145/3448016.3452759

[36] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers. In *ACM SIGCOMM*.

[37] Roger Grosse. Accessed January 31, 2021. CSC321 Lecture 6: Backpropagation. http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec6.pdf.

[38] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

[39] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[40] Song Han, Huizi Mao, and William J Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149* (2015).

[41] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. https://doi.org/10.48550/ARXIV.1806.03377

[42] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.

[43] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[44] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Push 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 (ASPLOS)* (Virtual).

[45] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. https://doi.org/10.48550/ARXIV.1811.06965

[46] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. , 14 pages.

[47] 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. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[48] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 27–39.

[49] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. https://doi.org/10.48550/ARXIV.1807.05358

[50] Charles Jin, Phitchaya Mangpo Phothilimthana, and Sudip Roy. 2022. Neural architecture search using property guided synthesis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (oct 2022), 1150–1179. https://doi.org/10.1145/3563329

[51] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. https://doi.org/10.48550/ARXIV.2004.09910

[52] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[53] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. 2016. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record* 44, 4 (2016), 17–22.

[54] 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 Annual Conference on Innovative Data Systems Research (CIDR'21)*.

[55] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. 2018. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. https://doi.org/10.48550/ARXIV.1807.02037

[56] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2018. A System for Massively Parallel Hyperparameter Tuning. (2018). https://doi.org/10.48550/ARXIV.1810.05934

[57] 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

[58] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. 2013. Parameter server for distributed machine learning. In *Big learning NIPS workshop*, Vol. 6.

[59] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. https://doi.org/10.48550/ARXIV.2006.15704

[60] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. https://doi.org/10.48550/ARXIV.2102.07988

[61] Rui Liu, Sanjan Krishnan, Aaron J Elmore, and Michael J Franklin. 2020. Understanding and Optimizing Packed Neural Network Training for Hyper-Parameter Tuning. *arXiv preprint arXiv:2002.02885* (2020).

[62] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. 2023. Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models. arXiv:2304.01852 [cs.CL]

[63] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. RAMMER: Enabling Holistic Deep Learning Compiler Optimizations with Rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 50, 17 pages.

[64] Yujing Ma, Florin Rusu, Kesheng Wu, and Alexander Sim. 2021. Adaptive Elastic Training for Sparse Deep Learning on Heterogeneous Multi-GPU Servers. https://doi.org/10.48550/ARXIV.2110.07029

[65] Yujing Ma, Florin Rusu, Kesheng Wu, and Alexander Sim. 2021. Adaptive Stochastic Gradient Descent for Deep Learning on Heterogeneous CPU+GPU Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 6–15. https://doi.org/10.1109/IPDPSW52791.2021.00012

[66] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[67] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[68] 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*, Vol. 7.

[69] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. https://doi.org/10.48550/ARXIV.1609.07843

[70] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. https://doi.org/10.48550/ARXIV.1712.05889

[71] Kabir Nagrecha. 2021. Model-Parallel Model Selection for Deep Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3448016.3450571

[72] Kabir Nagrecha. 2023. Systems for Parallel and Distributed Large-Model Deep Learning Training.

[73] Kabir Nagrecha and Arun Kumar. 2023. Tech Report of Saturn: An Optimized Data System for Multi-Large Model Deep Learning. https://adalabucsd.github.io/papers/TR_2023_Saturn.pdf

[74] Supun Nakandala and Arun Kumar. 2022. Nautilus: An Optimized System for Deep Transfer Learning over Evolving Training Datasets. In *Proceedings of the 2022 International Conference on Management of Data*. 506–520.

[75] Supun Nakandala, Kabir Nagrecha, Arun Kumar, and Yannis Papakonstantinou. 2020. Incremental and approximate computations for accelerating deep CNN inference. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–42.

[76] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. https://doi.org/10.48550/ARXIV.2008.09213

[77] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*. 20.

[78] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[79] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia

Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://doi.org/10.48550/ARXIV.1906.00091

[80] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[81] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2019. DL2: A Deep Learning-driven Scheduler for Deep Learning Clusters. https://doi.org/10.48550/ARXIV.1909.06040

[82] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-Grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1426–1439. https://doi.org/10.1145/3448016.3452788

[83] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–16. https://doi.org/10.1109/PACT52795.2021.00008

[84] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2016. Paleo: A performance model for deep neural networks. (2016).

[85] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. 2018. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[86] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

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

[88] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2019. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. https://doi.org/10.48550/ARXIV.1910.02054

[89] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. https://doi.org/10.48550/ARXIV.2104.07857

[90] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. 2017. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3135974.3135994

[91] Sergey Redyuk, Zoi Kaoudi, Sebastian Schelter, and Volker Markl. 2022. DORIAN in action: assisted design of data science pipelines. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3714–3717.

[92] 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. https://doi.org/10.48550/ARXIV.2101.06840

[93] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2022. NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In *Proceedings of the 2022 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3514221.3517860

[94] Yuji Roh, Kangwook Lee, Steven Euijong Whang, and Changho Suh. 2021. Sample Selection for Fair and Robust Training. https://doi.org/10.48550/ARXIV.2110.14222

[95] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 500–507.

[96] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108* (2019).

[97] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[98] 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. https://doi.org/10.48550/ARXIV.1909.08053

[99] Nimit S. Sohoni, Christopher R. Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. 2019. Low-Memory Neural Network Training: A Technical Report. https://doi.org/10.48550/ARXIV.1904.10631

[100] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2016. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. https://doi.org/10.48550/ARXIV.1610.09451

[101] Olivier Sprangers, Sebastian Schelter, and Maarten de Rijke. 2021. Parameter Efficient Deep Probabilistic Forecasting. https://doi.org/10.48550/ARXIV.2112.02905

[102] Yipeng Sun and Andreas M Kist. 2021. Deep learning on edge TPUs. *arXiv preprint arXiv:2108.13732* (2021).

[103] Alan Tucker. 1994. *Applied combinatorics*. John Wiley & Sons, Inc.

[104] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 267–284.

[105] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[106] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. https://doi.org/10.48550/ARXIV.1802.04730

[107] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[108] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[109] 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.

[110] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. 2018. Dataset distillation. *arXiv preprint arXiv:1811.10959* (2018).

[111] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. 2022. Merlin HugeCTR: GPU-Accelerated Recommender System Training and Inference. In *Proceedings of the 16th ACM Conference on Recommender Systems* (Seattle, WA, USA) *(RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 534–537. https://doi.org/10.1145/3523227.3547405

[112] Steven Euijong Whang, Yuji Roh, Hwanjun Song, and Jae-Gil Lee. 2021. Data Collection and Quality Challenges in Deep Learning: A Data-Centric AI Perspective. https://doi.org/10.48550/ARXIV.2112.06409

[113] 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. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://doi.org/10.48550/ARXIV.1910.03771

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

[115] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. {AntMan}: Dynamic Scaling on {GPU} Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[116] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 1335–1344. https://doi.org/10.1145/2783258.2783323

[117] 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]

[118] Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, and Tarek Abdelzaher. 2020. Scheduling Real-time Deep Learning Services as Imprecise Computations. In *2020 IEEE 26th*

International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). 1–10. https://doi.org/10.1109/RTCSA50079.2020.9203676

[119] Hantian Zhang, Xu Chu, Abolfazl Asudeh, and Shamkant B. Navathe. 2021. OmniFair: A Declarative System for Model-Agnostic Group Fairness in Machine Learning. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2076–2088. https://doi.org/10.1145/3448016.3452787

[120] Hantian Zhang, Nima Shahbazi, Xu Chu, and Abolfazl Asudeh. 2021. FairRover: Explorative Model Building for Fair and Responsible Machine Learning. In *Proceedings of the Fifth Workshop on Data Management for End-To-End Machine Learning* (Virtual Event, China) *(DEEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3462462.3468882

[121] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. 2017. SLAQ: Quality-Driven Scheduling for Distributed Machine Learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 390–404. https://doi.org/10.1145/3127479.3127490

[122] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.

[123] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning Exploratory-Training Framework. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 919–936.

[124] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. https://doi.org/10.48550/ARXIV.2201.12023

[125] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Mangpo Phothilimthana, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. 2020. Transferable Graph Optimizers for ML Compilers. (2020). https://doi.org/10.48550/ARXIV.2010.12438

A SATURN API USAGE EXAMPLE

In this section, we provide detailed pseudocode illustrating examples of: (1) registering a new UPP with Saturn, (2) invoking the profiler, and (3) calling the executor.

UPP Registration & Specification As discussed in Section 3.1, the user can specify a UPP by implementing two functions — one for knob-tuning, and one for execution. Listing 5 provides condensed pseudocode demonstrating how to register Fully Sharded Data Parallelism (provided in the PyTorch Distributed package) with Saturn. The full code is available at our GitHub repository.

```
class FSDPExecutor(BaseTechnique):

  def execute(task, gpus):
    WORLD_SIZE = len(gpus)

    def distributed_function(rank, world_size, task, gpus):
      setup()
      model, loss_fn = task.get_model(), task.loss_function
      knobs = task.strategy.knobs
      hints = task.hints

      if hints.is_transformer:
        wrap_policy = "transformer"
      else:
        wrap_policy = "auto"

      model = FSDP(model, offload=knobs.offload, device=rank, auto_wrap_policy=
       wrap_policy)
      if knobs.checkpoint:
        model = apply_checkpointing(model)

      lr = task.hparams.lr
      optimizer = task.hparams.optimizer_cls(model.parameters(), lr)

      execution_loop(model, task.iterator)


    multiprocessing.spawn(distributed_function, args=(task, gpus))

  def search(task, gpus):
    knob_search = [
      checkpoint: True, offload: True,
      checkpoint: True, offload: False,
      checkpoint: False, offload: True,
      checkpoint: False, offload: False
    ]

    selected_config, runtime = None, None
    for config in knob_search:
      trial_task.knobs = config
      trial_task.length = 5 batches
      try:
        rt = time(execute(trial_task, gpus))
        if rt < runtime:
          selected_config = config
          runtime = rt
      except:
        continue
```

Listing 5: Registering FSDP as a UPP in Saturn's Library.

This class can now be registered with the parallelism library as illustrated in Listing 2 in Section 3.1. The newly added technique can now be automatically applied to newly submitted models for profiling & execution.

Profiling & Execution The profiler allows us to estimate the performance of each model under different techniques with different GPU allocation levels. Listing 6 illustrates an example of specifying, profiling, and executing two GPT-J fine-tuning jobs. The base model is loaded from HuggingFace's model hub, before being passed into a Task wrapper with a hint for Saturn to note that the model is a Transformer.

```
from transformers.models.gptj import GPTJForCausalLM
from data import dataloaders

def load_model():
  configuration = GPTJConfig.from_pretrained("EleutherAI/gpt-j-6B",
       output_hidden_states=False)
  model = GPTJForCausalLM.from_pretrained("EleutherAI/gpt-j-6B", config=
       configuration)


hints = {transformer: True}
hparams_a = HParams(lr=1e-3, epochs=5, optimizer_cls=torch.optim.SGD)
hparams_b = HParams(lr=1e-5, epochs=5, optimizer_cls=torch.optim.Adam)
task_a = Task(load_model, dataloader, hints, hparams_a)
task_b = Task(load_model, dataloader, hints, hparams_b)


task_list = [task_a, task_b]
profile(task_list)
execute(task_list) # automatically invokes the MILP search procedure
```

Listing 6: Specifying a training job and launching it with Saturn.

B INTERVAL INTROSPECTION

Here, we provide the basic algorithm for our introspective solver. Essentially, we re-trigger the solver on a fixed interval and determine if the new solution improves performance versus just continuing with the existing plan. If the new plan is superior, we checkpoint all active jobs and re-launch with the new plan.

We use a tolerance level, T, to describe the minimum acceptable benefit of an introspective plan switch. If the swap only provided a 5 second benefit, for example, the switching overheads alone might outweigh the makespan reduction.

C ADDITIONAL RELATED WORKS

In this section, we go into some other DL systems works that are relevant to performance optimization in general, though they do not overlap directly with the specifics of our SPASE problem.

System Optimizers: KungFu [67] provides an interface for users to express various procedures for mid-training system parameter changes. Litz [85] provides a programming model for elastic parameter server data parallelism. TeraPipe uses dynamic programming to

Algorithm 2 : **Round Introspection**(Workload W, Interval I)

```
Schedule S = MILP(W)
M = Makespan(S)
E2ESchedule = S[0 : I]
T = 500
while W not exhausted do
    W = W after I seconds of S
    S = S[I :]
    M = M − I
    Proposal = MILP(W)
    if Makespan(Proposal) ≤ M - T then
        S = Proposal
        M = Makespan(Proposal)
    end if
    E2ESchedule.append(S[0 : I])
end while
return L
```

optimize the partitioning and execution of pipeline parallelism [60]. Systems like Rammer [63], GO [125], TVM [25], SystemML's query rewriter [19] and compiler autotuners [83, 106] provide similar up-front optimizations for DL workloads. These automated search procedures are orthogonal to our own work and support can be added in the future using our UPP abstraction and Library API.

Other Model Selection Systems: Nautilus [74] optimizes model selection for transfer learning. α-NAS [50] proposes a method for creating architecture search workloads. Other systems like Fair-Rover [120] tackle human-in-the-loop model building. These works are orthogonal to our own — they create/modify the model selection workload that Saturn executes.

Other DL System Optimizations: Optimizations such as compilation [4, 25, 52, 55, 83], batching [61, 77, 123], compression [29, 40, 96], and graph substitution [47, 48, 104] are orthogonal to our work. Many systems (e.g. DeepSpeed [88, 89, 92], Megatron [1, 78, 98], Hotline [9], HugeCTR [111], RecShard [7, 79], HogBatch [65], and Switch Transformers [33]) propose new parallelisms, all expressible under our Library API. Data pipeline optimizations [91, 109, 110, 112], fairness systems [35, 94, 119], and end-to-end pipeline managers [16, 18, 82, 100, 116] are also mostly orthogonal to our own work; we do not restrict the workload/data design. Some other works have addressed large-model challenges in different ways; e.g. by using alternative, parameter-efficient architectures [101] or else by training on non-GPU hardware [27, 102]. By contrast, Saturn is intended to optimize existing large-model GPU-training settings. Still other works optimize for settings such as DL inference [11, 14, 23, 24, 75, 118] or non-DL compute [8, 32, 36, 43, 107, 121]. These works target a different setting entirely.