Saturn: Resource-Aware Multi-Query Optimization for Multi-Large-Model Deep Learning Workloads

ABSTRACT

Over the last several years, deep learning (DL) has seen an exponential increase in the size and memory demands of model architectures. As a result, *large-model training* has become closely tied to a rapidly evolving space of parallelism techniques. New tuning spaces have emerged, requiring users to select parallelism techniques based on their workload and hardware resources. This is particularly challenging in procedures such as model selection, where *several* models must be parallelized. Model selection also introduces the interconnected problems of resource apportioning and scheduling.

In this paper, we present the first formalization of this joint problem to tackle large-model model selection. We name the joint problem SPASE — Select a Parallelism, Allocate resources, and SchedulE. We tackle SPASE by reimagining parallelism selection as an instance of *query optimization*. We then propose a unified representation of parallelism approaches, inspired by UDF libraries in RDBMSs. We combine this with an empirical profiler for performance estimation, then formulate SPASE as an MILP, inspired by RDBMS multi-query optimization. We find that direct application of an MILP-solver is effective for our problem setting, especially when combined with an introspective solving optimization to further improve performance. We implement our techniques into a data system we call Saturn. Experiments on large-model workloads drawn from DL practice show that Saturn achieves 38-50% reductions in model selection times versus current practice.

1 INTRODUCTION

Deep learning (DL) has become ubiquitous in data analytics across many commercial and scientific domains for text, image, audio, video, and even tabular and multi-modal data. In many cases, DL's advances are thanks to ever larger model sizes. Figure 1(A) illustrates how DL architecture sizes have ballooned in recent years, partly due to the rise of powerful and versatile Transformer architectures [18]. For instance, the popular GPT-2 [63] and ViT [19] models need 10s of GBs of GPU memory and take days to train.

While large companies have the resources to train such models, most regular DL users at smaller companies and in the domain sciences often cannot afford large-scale compute. Thankfully, in most cases they need not train from scratch. They can download pre-trained models from hubs like HuggingFace [79] and fine-tune them on smaller task-specific data. But such users still face 3 systems headaches: (1) *GPU memory* remains a scaling bottleneck. Large-memory GPUs are expensive, and even public cloud vendors still ration them. (2) *Multi-GPU parallelism* is needed but existing parallelization systems are unintuitive to configure for ML/DL-oriented users; and (3) *Model selection*, which involves tuning hyperparameters, layers, etc., only amplifies the computational load.

Overall, large-model DL is still painful to adopt 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 large language models (LLMs) from HuggingFace, GPT-2 and GPT-J, both of which are known to offer strong results for text processing tasks [63, 75]. She finetunes them on her dataset by varying batch sizes and learning rates. 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?

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 summarizing the state-of-the-art for large-model and parallel DL systems and explain why they are not enough to tackle each question above. Table 1 lists a precise conceptual comparison of our work with prior art on several key aspects. We will discuss related work in even more detail later in Section 6.

(1) Which parallelism technique to use for each model? There is a large body of work in the ML systems world on how to parallelize/scale large models on many GPUs. Some well-known techniques are: shard a model by layers (simple "model parallelism"), spilling shards to DRAM [29, 47], pipeline parallelism as in GPipe [30], fully-sharded data-parallel (FSDP) as in PyTorch [2], hand-crafted hybrids as in Megatron [70], as well as general hybrid-parallel approaches such as Unity [34, 72] and Alpa [85]. But no singular technique dominates all others in all cases. Relative efficiency depends on a complex mix of factors: hardware specifics, DL architecture specifics, and 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 their behavior in hard-to-predict ways [43]. For instance, pipelining requires tuning the model partitions and "microbatch" sizes, while FSDP requires tuning offloading and checkpointing decisions. It is unreasonable to expect DL users to manually navigate such complex systems tradeoffs. *Thus, we need to automate parallelism technique selection.*

(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 must 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

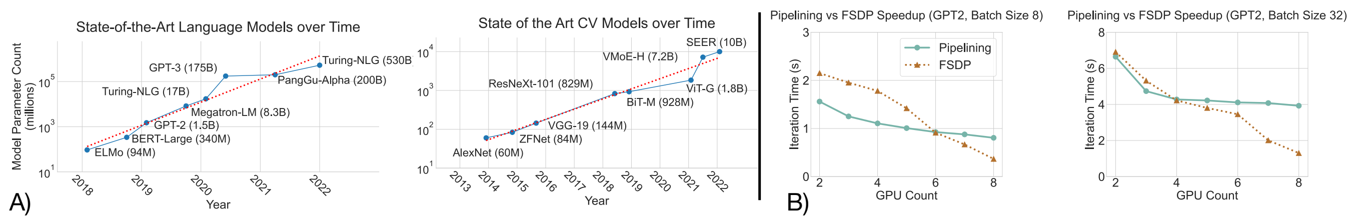

Figure 1: (A) The exponential growth in the size of state-of-the-art DL models in NLP and CV (log scale), extrapolated from a similar figure in prior art [70]. (B) Illustration of the performance crossovers between FSDP and pipeline parallelism, with knobs tuned for each setting.

Table 1: Overview of prior art.

		Fidelity	Multi-Model	Resource Allocation	Parallelism Selection	Out-of-the-Box Large Model Support
Hybrid Parallelism	Alpa [85]	✓	✗	✗	✓	✓
	FlexFlow [34]	✓	✗	✗	✓	✗
	Unity [72]	✓	✗	✗	✓	✓
Performance Evaluation	Paleo [60]	✓	✗	✗	✓(limited)	✗
Model Selection	Cerebro [38]	✓	✓	✗	✗	✗
	ASHA [40]	✓	✓	✓	✗	✗
Scheduling	Gandiva [80]	✓	✓	✗	✗	✗
	Antman [81]	✓	✓	✗	✗	✗
	Tiresias [25]	✓	✓	✗	✗	✗
Resource Allocation	Pollux [62]	✗	✓	✓	✗	✗
	Optimus [57]	✗	✓	✓	✗	✗
SPASE	Saturn	✓	✓	✓	✓	✓

of running models one after another using all GPUs is sub-optimal as it *reduces model selection throughput* and adding more GPUs per model often yields diminishing returns, as Figure 1(B) shows. Alas, the scaling behaviors of large-model parallelism techniques are not linear and often hard to predict. Prior art has studied data-parallel resource allocation (e.g., Pollux [62] and Optimus [57]) and model selection optimization (e.g., Cerebro [38] and ASHA [40]). 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 apportionment for large models.*

(3) How to orchestrate such complex parallel execution? This is a question of scheduling, i.e., deciding which jobs to run when, coupled with the decision of GPU apportionment. Two naive approaches are to run models in a random order or to use a generic job/task scheduler. Both can lead to GPU idling due to a lack of awareness of how long models actually run. Prior art has studied sophisticated runtime-aware multi-model DL scheduling, e.g., Gandiva [80] and Tiresias [25]. But once again, none of them target large-model DL and its complex interplay of parallelism selection and GPU apportionment, which affect runtimes in a way that can alter the tradeoffs of scheduling.

Overall, there is a pressing need for a unified and automated way to tackle these 3 systems concerns of multi-large-model DL: select parallelism technique per model, apportion GPUs per model, and schedule them all on a given cluster. We call this novel joint problem SPASE: Select Parallelism, Apportion resources, and SchedulE.

1.2 System Desiderata

To help democratize multi-large-model DL and ensure ease of practical adoption, we seek a data system that tackles the SPASE problem 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 multiple parallelisms and also make it easy for users to add new parallelisms in the future.

(2) Non-disruptive integration with DL tools. The system must build on top of popular DL tools (e.g., PyTorch [42] and TensorFlow [4]) without altering their internal code. This can help

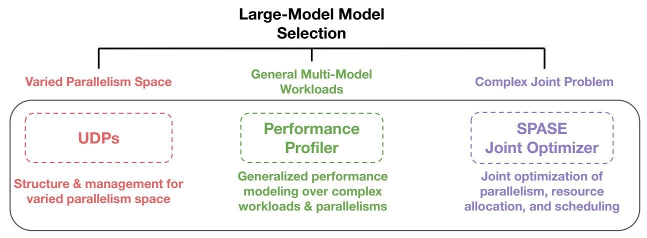
Large-Model Model Selection
Varied Parallelism Space
General Multi-Model Workloads
Complex Joint Problem
UDPs
Structure & management for varied parallelism space
Performance Profiler
Generalized performance modeling over complex workloads & parallelisms
SPASE Joint Optimizer
Joint optimization of parallelism, resource allocation, and scheduling

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

practical adoption and 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.

aa**(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 not allowed because they can confound users.

1.3 Our Proposed Approach

In this work, we use a DBMS-inspired lens to formalize and tackle the SPASE problem for multi-large-model DL while meeting the above desiderata. We build a system we call Saturn. As Figure 2 illustrates, our approach is three-pronged:

(1) Parallelism Selection and UDPs. Analogous to how RDBMS query optimizers automatically select between physical operators for a given logical operator [66], we build a "parallelism optimizer" for large-model DL. To ensure the first desideratum of extensibility, we introduce the abstraction of User-Defined Parallelisms (UDPs), analogous to User-Defined Functions in RDBMSs. UDPs can just specify existing parallelism in DL tools or enable users to add new parallelisms as blackboxes for Saturn to work with. This also ensures the second desideratum of non-disruptive integration. In fact, we create a default UDP library in Saturn to support 4 major existing parallelisms: pipelining, spilling, DDP, and FSDP.

(2) Performance Profiling and Auto-Tuning. To apportion GPUs and tune parallelism knobs in a way that ensures our fourth desideratum, we need to get 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. Based on this observation, we create a simple but highly general and effective solution: profile all jobs using the full "grid" of options for both GPU counts and parallelism knobs based on only a few minibatches. We find that the overhead of this approach is affordable due to the long runtimes of actual full DL training. This also ensures our second and third desiderata because all DL tools already offer data sampling APIs that we can just use on top of the user-given model specifications. Of course, we use the full data for the actual DL jobs to ensure the fourth desideratum.

(3) Joint Optimization and Scheduling. Given both of the above system design choices, we can now tackle the SPASE problem using joint optimization, inspired by multi-query optimization (MQO) in the RDBMS world [69]. Our analogue goes beyond MQO, however, because we not only select "physical operators" of all queries simultaneously but also apportion resources and schedule multiple "queries" on the cluster in one go. 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 [26]) to a handful of known scheduling heuristics. The solver yields the best results overall even with a timeout. Thus, we incorporate it into Saturn as our optimizer and scheduler. In DL workloads, actual model training heavily dominates overall runtimes, not the optimizer. As such, 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 futher raise resource utilization.

We implement Saturn in Python and expose high-level APIs for (offline) specification of UDPs and model selection APIs for actual DL training usage. Figure 3 in Section 3 shows our system architecture. Under the hood, we implement 4 components: Parallelism Plan Enumerator, Performance Profiler, Joint Optimizer, and Executor. These components build on top of the APIs of the massively task-parallel execution engine Ray [49]. We use Ray to handle lower level resource management, e.g., placing scheduled DL jobs on the GPUs, as well as to parallelize our profiling runs to make them faster. Using two benchmark large-model DL workloads from NLP and CV, we evaluate Saturn against several baselines, including an emulation of current practice of manual decisions on the SPASE problem. Saturn reduces overall runtimes by 38% to 50%, which can directly translate to significant cost savings on GPU clusters, especially in the cloud. We also perform an ablation study to show the impact of all our optimizations. Finally, we evaluate Saturn's sensitivity to the size of models, workloads, and nodes.

Overall, this paper makes the following contributions:

- To the best of our knowledge, this is the first paper to formalize and study the SPASE problem for multi-large-model DL, aiming to free regular DL users from having to manually select and tune parallelisms, apportion GPUs, and schedule multi-jobs.
- To the best of our knowledge, this is the first paper to formalize and address resource scheduling for model selection workloads. It is also the first to address large-model model selection.
- To the best of our knowledge, this is the first model selection system to natively support large-model architectures through complex parallelisms.
- We introduce a novel UDF abstraction for DL parallelisms, providing structure to a currently unmanaged space.
- We create an empirical evaluator + MILP that can automatically determine an optimized solution to the SPASE problem.
- We implement our ideas into a data system we call Saturn. We perform an extensive empirical evaluation on two benchmark large DL model workloads. Saturn reduces runtimes by up to 57% 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 acyclic graphs (DAGs) 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 to reflect a dataset [22, 24]. 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. Backpropagation refers to the application of the derivative chain rule with respect to the model's parameters. Thus, parameter updates are computed and applied throughout the model. 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*. Typically, training will consist of multiple such epochs.

We highlight a few characteristics that are significant to our problem setting. First, we see the *iterative nature of SGD* provides natural boundaries for checkpointing. A model could be paused on a minibatch execution boundary to be resumed later. Second, we note the computational load and memory demands of prediction, gradient computation, and parameter updating. On very large (billion-parameter) model architectures, training can take petaflops of compute. This is a strong motivator for users to employ parallelisms for multi-GPU execution.

Multi-GPU parallelism is now typical in compute-intensive model training [31]. Several parallel execution approaches exist, and rapid developments in the ML performance space are only leading to further growth. We will not provide a comprehensive review of all forms of parallel execution in DL; for this we refer readers to comprehensive surveys [51, 71]. Instead, we will describe 5 of the most popular parallelisms: (1) data parallelism, (2) model parallelism, (3) pipelining, (4) fully-sharded data parallelism, and (5) spilling. We also point out any "parallelism knobs" that complicate scaling behaviors and make theoretical performance analyses challenging.

Data parallelism is one of the most basic approaches to parallel DL execution. Given a DL model, data parallelism replicates the architecture across multiple accelerators. These replicas are then fed different minibatches 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. This approach has generally fallen out of favor for multi-GPU execution due to poor reproducibility. All-reduce data parallelism synchronizes replicas through peer-to-peer communication. Synchronization occurs at the minibatch boundaries, ensuring reproducible and exact execution.

Model parallelism partitions the *model* rather than the data. The model's DAG is sharded, and subgraphs are placed on different GPUs. Performance is mostly dictated by the specific sharding scheme. Purely sequential partitioning is easy to implement but provides no opportunity for parallel computation. More complex partitioning schemes are typically architecture-specific [1]. In either case, model parallelism distributes memory demands across GPUs.

Pipelining optimizes the sequential model parallel case. It partitions a minibatch into smaller microbatches, then shuttles the microbatches through the stages of the model. This enables parallel execution across stages on different pieces of data. Pipelining generally suffers from *synchronization* overheads between prediction and backpropagation steps. Some asynchronous schemes exist to mitigate this issue, but they affect execution correctness [82]. Pipelining's performance is heavily dependent on (1) the partitioning scheme and (2) the number of microbatches. Prior works have shown the importance of tuning these knobs, either via expert knowledge or an automated procedure [43].

Fully-Sharded Data Parallelism (FSDP) is a new execution scheme that combines model parallelism with data parallelism. Originally introduced with Microsoft's ZeRO [64], it has since been integrated into the PyTorch Distributed package [42]. FSDP partitions a model graph across multiple accelerators, then sends different minibatches to the accelerators. FSDP runs all-gathers on model layers in sequence as data moves through the graph. The currently executing layer is data-parallel-replicated; the other operators are still distributed in a model-parallel way. This allows users to benefit from both data parallel processing and model parallel memory distribution. FSDP exposes two main user-configured optimizations: (1) checkpointing and (2) offloading. Checkpointing [16] applies a popular memory reduction technique on top of the layer partitioning. Offloading borrows from spilling (described below) to push some layers to main system memory when they are not executing. Turning these knobs on can decrease memory pressure at the cost of some performance. Understanding when it is worth turning on one optimization or another generally requires empirical testing.

Spilling swaps model partitions between system memory and GPU memory for piece-wise GPU-accelerated execution [9, 50]. This approach induces DRAM-GPU communication overheads, but enables large models to be trained with just one GPU. Spilling performance is affected by a *partition count* knob, which decides how many swaps are needed during execution.

3 SYSTEM OVERVIEW

We now describe Saturn's architecture. Our design uses four components and an API to tackle large-model model selection in three solving phases. First, problem specification, for which we provide the API and the Parallelism Library component. Second, performance estimation, for which we build the Trial Runner. Third, the SPASE problem, tackled through our joint optimizer and executor module.

We implement Saturn on top of the parallel processing framework Ray [49]. Figure 3 presents Saturn's architecture. Next, we provide details on Saturn's components.

3.1 Problem Specification

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

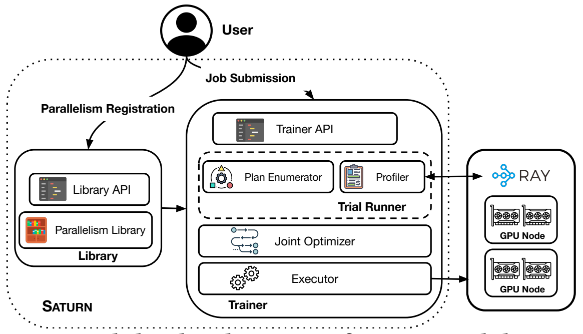

Figure 3: High-level architecture of Saturn and the interactions between system components. Parallelisms are specified in the Library, which then serves as a lookup for the Trial Runner and Executor.

API. Saturn's API provides a flexible, easy-to-use interface for both specifying parallelisms and submitting training jobs. There are two parts to the API: the Library API and the Trainer API. Users create training "Tasks" through the Trainer API by specifying a model initialization function, a dataloading function, and any hyperparameters. This is sufficiently general to cover any model selection workload. 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))
t_3=Task(get_model,get_data,HParams(lr=5e-5,epochs=5,optim=SGD))
```

Listing 1: Specifying tasks through Saturn's API.

Training procedures are defined by "Parallelisms", which implement execution functions. These parallelisms can be registered with our Library for later access, evaluation, and execution. 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 tasks and parallelisms are specified, users can invoke the Trial Runner to produce runtime estimates in a single line of code. The user can finally trigger training by calling the executor module, again in one line of code. We illustrate in Listing 3.

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

Listing 3: Profiling and execution invocations.

Parallelism Library. Saturn's Parallelism Library design is inspired by DL model hubs [79] and query optimizers in DBMSs. We follow a define-once, use-anywhere design, wherein registered parallelisms can be reused across models, execution sessions, and even different cluster users. This implemented by creating a persistent database to store library-registered parallelisms. For now, our database is a simple file store, but future versions could use more advanced DBMSs to manage a larger parallelism space.

The Library allows users to implement new parallelisms by extending 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-time estimate. Knob-optimization can also optionally be tackled here. Failed searches (e.g. OOMs) can be handled by returning null values.

The execute function should train the provided task to completion using the allotted GPUs. It also uses any execution parameters produced during the search phase to optimize execution.

This minimal structure, inspired by UDP templates in DBMSs, is sufficient for us to enable empirical search and profiling. We implement this interface for four popular parallelisms: DDP, GPipe-style pipeline parallelism, FSDP, and model spilling through the FairScale package [9]. Each implementation is done in <200 LOC. Once defined, a parallelism can be registered with the library under some user-set name (e.g. "pytorch-ddp").

To the best of our knowledge, the Library is the first structured interface for implementing and applying a general space of parallelisms. We use it as a component of our end-to-end training system, but it could also be used as a standalone tool.

3.2 Performance Estimation

We provide the Trial Runner component to tackle performance evaluation and statistics generation. The Trial Runner consists of two submodules — a plan enumerator and a profiler. The Trial Runner module creates estimates of model runtimes under different parallelisms/GPU apportionments. The SPASE problem cannot be solved without first understanding how individual parallelisms will perform with each job under different GPU apportionments. The Trial Runner is not a *parallelism selector* — it simply generates information needed for our SPASE solver. It is our empirical substitute for parallelism-specific theoretical models used in prior art [57, 62] . It gives us the ability to optimize without tightly coupling our design to a specific parallelism scheme.

Plan Enumerator. Our Trial Runner must be general enough to support a vast and varied space of large-model parallelisms, so we cannot rely on parallelism-specific performance models. Instead, we use an empirical grid-search across parallelisms and GPU allocation levels for each model to produce performance estimates. An analogy can be drawn to plan enumerators for multi-queries in DBMSs — given a set of multiple models, we draw up a list of all possible execution plans for each individual model/query. We support a DBMS-like hint system to reduce the enumeration space if desired. Next, we evaluate the performance of each plan for each model in isolation; these local performance estimates are needed for our later multi-job optimization stage.

Profiler. SGD is iterative and consistent, so we can extrapolate end-to-end runtime estimates from averaged performance over a just few training iterations. We use Ray to parallelize the searches and further reduce overheads. In our evaluations, profiling 12 1.5-6B parameter models on 4 parallelisms took <20 minutes. Large-model jobs can run for *days*, so this overhead is negligible.

3.3 Joint Optimizer & Executor

The Trial Runner provides us with local-job statistics. Now we use those statistics for holistic multi-query optimization by tackling the SPASE problem.

Joint Optimizer. The joint optimizer is transparently called when the user invokes the executor. It uses the runtime estimates generated by the Trial Runner and cluster information to produce an execution plan. This plan bakes in *parallelism selection, resource allocation, and schedule construction.*

Our optimizer is implemented in two layers. First, an MILP to produce makespan-optimized execution plans. Second, an introspective, round-based resolver that runs on top of the MILP to support dynamic reallocation. We discuss the optimizer and its optimizations in more detail in Section 4.

Executor. The execution plan generated by the optimizer is implemented by the executor module. The executor runs on top of Ray to leverage its parallel processing capabilities. By default, Ray uses its own naive scheduler, and swapping it out for a custom solution is challenging. Instead, the executor implements our schedule *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 schedule. Thus, the executor adapts Ray to support our execution plan.

In the current version, our system supports both single-node and multi-node training across different models. We focus on the case where models fit within aggregate node memory (i.e. total GPU memory + DRAM). This is not an inherent constraint — we only restrict it in this way for implementation ease. Many of the large-model parallelisms we implemented for our library do not yet support multi-node training out-of-the-box, so we simply disable this setting for now. This is only a limitation in extreme-scale cases; we can easily train 10B+ parameter models within one node. Removing this limitation in the future would only require two minor modifications: (1) an adjustment to the MILP that we describe in Section 4, and (2) giving the Trial Runner a broader range of GPU counts to explore.

4 SPASE JOINT OPTIMIZER

We first describe the SPASE joint problem and present a few simple baselines. We then dive into our MILP formulation of the problem. We evaluate Gurobi [26]'s effectiveness in solving the MILP on simulated SPASE workloads. We then introduce an introspective re-scheduling mechanism on top of our MILP solution that allows our system to adaptively reassess its decisions over time.

4.1 Problem Basics

The SPASE problem covers parallelism selection, resource allocation, and schedule construction. Typical schedulers can set task start times (i.e. the execution schedule). 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 that will use the allotted GPUs. To the best of our knowledge, this is the first paper to discuss and tackle this problem setting.

Table 2: MILP Notation used in Section 4.2

MILP Inputs

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.
$G_t \in \mathbb{Z}^{+S_t}$	Variable length list of requested GPU counts for each configuration of task t. A configuration consists of both a parallelism and a GPU allocation.
$R_t \in \mathbb{R}^{+S_t}$	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 strategy.
$O_{t,n} \in 0, 1$	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.

Since model selection is an offline scheduling problem, Saturn is given all jobs up front. Using the Trial Runner module, we can generate all necessary runtime statistics for these jobs. Even with this information, the problem is NP-hard. Prior work on network bandwidth distribution [8] has shown that the basic resource allocation problem is NP-hard. SPASE is a more complex version of this problem that incorporates parallelism selection and makespan-optimized scheduling, so it is also NP-hard. The problem is intractable, so building an optimal solver is infeasible.

We now formulate SPASE as an MILP. This allows us to formally state and understand the details of the problem and create simulation studies. Based on our understanding and analysis of the problem, we find that directly using an MILP solver (e.g. Gurobi [26]) as our joint optimizer can suffice as a solution to the SPASE problem.

4.2 MILP Formulation

We now formalize the SPASE problem as an MILP. Our input consists of a full grid of possible training configs (parallelisms and GPU allocations) and corresponding runtime estimates for each task. Table 2 describes the notation used in this section. Our MILPs objective is to minimize makespan, shown in Equation 1.

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

We now define the constraints used in our MILP. First, we define makespan. Equation 2 ensures that makespan is set to the latest task start time plus the runtime of that task's selected parallelism and apportionment.

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

Next, we ensure that each task will only uses one parallelism/apportionment, and only uses one node. The single-node constraint is simply added for prototyping ease, as some of the Library parallelisms we implemented only work in the single-node setting. This constraint could be removed in future versions.

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

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

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

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

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

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

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.

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

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

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 $t2$.

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

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

This MILP formulation is complex, spanning and merging three different problem aspects. To the best of our knowledge, this is the first MILP formulation of either SPASE or even any DL resource allocation problem. We can use the above formulation to create a straightforward optimizer using an MILP solver like Gurobi [26]. 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 algorithms in prior art. Then, we run comprehensive evaluations on a simulated testbench. We find that that MILP-solved approach outperforms all baselines by a significant margin.

4.3.1 ***Baselines.*** The SPASE problem already exists in the wild for practitioners tackling large-model model selection workloads. So we can find initial baselines by investigating current practice.

A common heuristic-based approaches is to maximize each job's allocation. Each job is given all GPUs in a node, then the best parallelism for that particular setting is applied. The jobs are then run in sequence. This optimizes local efficiency and maximizes available GPU memory for each job. Though this eliminates the in-node task parallelism, it flattens the Allocation and Scheduling aspects of the SPASE problem. This baseline, which we name "Max-Heuristic", is the most common approach in current practice.

Another human-set heuristic might be to assign a single-GPU technique to every model to maximize task parallelism [50]. In the large-model setting, single-GPU techniques often involve low-performance memory-spilling. This baseline, which we name "Min-Heuristic" aims to counterbalance this through task parallel scaling.

We now propose a simple algorithmic approach that incorporates performance data to produce non-trivial solutions. Optimus, a DL resource scheduler paper, proposed a simple greedy resource allocator that uses an "Oracle" model to provide performance estimates [57, 62]. Our Trial Runner statistics serve as our Oracle.

The idea of the algorithm is to iteratively assign GPUs to whichever model that will see the greatest immediate benefit. We name this algorithm Optimus-Greedy. We show the pseudocode in Algorithm 1, reusing some variables from Table 2.

Algorithm 1 : OPTIMUS-GREEDY(Tasks T, GPUs G)

```
1: $L = [1|t \in T]$
2: while sum(L) < G do
3:     $CR = [R_{t,s}|t, l \in (T, L), s \in S_t$ where $G_{s,t} == l]$
4:     $PR = [R_{t,s}|t, l \in (T, L), s \in S_t$ where $G_{s,t} == l + 1]$
5:     $GAIN = [c - p|c, p \in (CR, PR)]$
6:     $L[ArgMax(GAIN)] + +$
7: end while
8: return $L$
```

This algorithm provides us with resource allocations per task. We transform this to a SPASE solution by selecting the best parallelism for each task's allocation. Note that this algorithm assumes there are more GPUs than models, all running on a single node. For multi-node training, we run this algorithm one node at a time, and submit models in batches of four. Like many iterative greedy algorithms, this approach relies on consistent scaling behaviors. It has only a local, greedy view (i.e. how will I benefit from allocating this *one* GPU) rather creating a one-shot global distribution.

This gives us three baselines, covering standard practice and basic algorithmic approaches. We add a randomizer-based solution for a fourth. We summarize our baselines below:.

(1) Max-heuristic solution wherein all GPUs within a node are given to one task at a time.
(2) Min-heuristic solution where a single-GPU technique (spilling) is given to each task to maximize task parallelism.
(3) A greedy algorithm, inspired by the one used in the Optimus [57] resource scheduling paper.
(4) Randomized selector/scheduler. Parallelisms and allocations are randomly selected for every task, then tasks are randomly scheduled.

We will compare our Gurobi-solved approach to these baselines. Since our MILP is complex, Gurobi is unlikely to converge to an optimal solution within a practical timeframe. Instead, we can set a timeout (e.g. 5 minutes) for the solver to try and produce a reasonable solution. Next, we discuss the workloads we will use to evaluate our solutions in a simulator testbench.

4.3.2 *Simulation Workloads.*

We use two benchmark workloads in our simulations. The details of the two workloads are described in Table 3. Runtime estimates for each parallelism/allocation/model are produced through our Trial Runner prior to the simulations. We create two simulated hardware settings: an 8-GPU single node config, and a 32-GPU 4-node config.

Figure 4 illustrates the results of our simulations. We use Gurobi as our MILP solver with a 300s timeout. All baselines are rerun three times and averaged with error bars displayed, but only the randomized algorithm shows significant non-determinism. In all cases, our solver produces substantially better solutions than the baselines. We achieve a makespan reduction of up to 55% versus the Min-Heuristic, 27% versus the Max-Heuristic, 50% versus the randomized scheduler, and 30% versus the Optimus-Greedy algorithm.

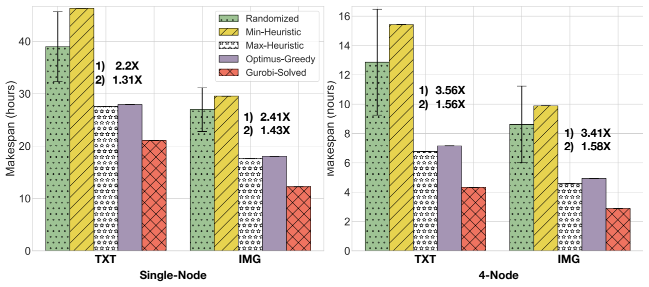

Figure 4: SATURN Simulation results comparing our MILP to two key baselines. Next to each group, we show SATURN's speedup versus (1) the weakest and (2) the second-best performer. The rightside y-axis is used for the IMG workloads, and the leftside y-axis is used for the TXT workloads.

Our solver takes the longest to converge, with a 300s timeout — the others complete in <10s. These overheads are far outweighed by the makespan improvements, however.

4.4 Introspection

In general, one-shot up-front scheduling is suboptimal. Workloads evolve over time, either due to online changes (e.g. model selection job adjustments from AutoML heuristics) or ongoing execution (job runtimes reduce as they are trained). We suggest that if the optimizer was *rerun* partway through execution, it might produce a *different, more performant* solution for the remainder of the job. To achieve this, we propose the use of *introspection* [80].

Introspection — a key feature in some state-of-the-art DL schedulers [80] — proposes that a scheduler should *learn* as it executes. There are two ways in which a schedule might be altered or adapted via introspective reassessment. First, 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, dynamic rescaling. The initial up-front training plans could be adjusted (e.g. 6 GPUs to 2) partway through a schedule. In our SPASE problem, this could even involve switching from one parallelism to another.

We now describe our implementation of introspection. 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 will have modified the workload distribution. We can *rerun* the solver on interval boundaries, so it can introspectively adjust its original solution as the workload changes. Figure 5 illustrates the design. We describe the procedure in Algorithm 2.

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!

We create a new dynamic baseline, "Optimus-Dynamic", by swapping the MILP blackbox in Algorithm 2 for the Optimus-Greedy algorithm. We compare this baseline against our dynamic MILP solver and illustrate the impact of the interval and threshold knobs in Figure 6. Since each round produces a globally optimized solution,

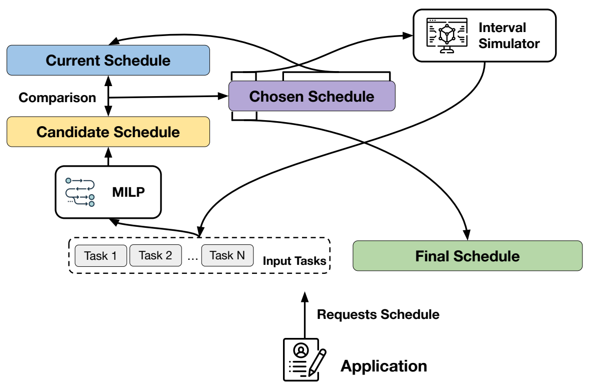

Figure 5: Illustration of the introspective feedback loop used on top of our MILP solver.

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

```
1: Schedule$S = MILP(W)$
2: $M = Makespan(S)$
3: E2ESchedule = $S[0 : I]$
4: T = 500
5: while W not exhausted do
6:     $W = W$ after $I$ seconds of $S$
7:     $S = S[I :]$
8:     $M = M - I$
9:     Proposal = $MILP(W)$
10:    if Makespan(Proposal) ≤ M - T then
11:        $S$ = Proposal
12:        $M$ = Makespan(Proposal)
13:    end if
14:    $E2ESchedule.append(S[0 : I])$
15: end while
16: return $L$
```

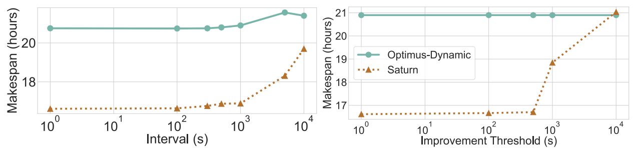

Figure 6: Solver sensitivity plots versus interval and threshold knobs. We fix the interval to 1000s for the threshold analysis and the threshold to 500s for the interval analysis. Optimus-Dynamic is included as a reference baseline.

the dynamic MILP solver's performance improves monotonically (not accounting for pre-emption costs) as knobs become more fine-grained. Lower interval/threshold levels naturally subsume higher levels in this scheme. By contrast, locally-optimizing algorithms like the Optimus approach produce non-monotonic scaling.

Introspective reassessment does not have to occur on interval completion; we can simulate the workload state at the next interval using the current schedule. Then, the solving process for the next introspective round can be overlapped with execution to eliminate introspective solving times. This scheme provides marginal speedups of 15-20% versus our one-shot MILP, as shown in Section 5.2. With introspection plus our MILP solver, Saturn's joint optimizer is 1.5-4.1X faster than the current practice 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 [40, 41]. In the current version, we do not explicitly optimize for such settings; future work could adapt our introspection mechanism to better fit such use-cases.

Our introspection mechanism takes inspiration from prior art in the DL cluster scheduling space. Antman and Gandiva [80, 81] show how pre-emption on minibatch boundaries can improve end-to-end efficiency; Pollux and Optimus [57, 62] show the value of dynamic rescaling. We unify both these optimizations in our implementation of introspection.

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?

Workloads: We run 2 end-to-end workloads covering typical model selection scenarios. One focuses on model selection for large language models. The second focuses on mixed-architecture selection across Transformers and CNNs for image classification.

Datasets: Our language modeling task uses the popular *WikiText-2* [48] dataset. WikiText-2, which is composed of sentences drawn from Wikipedia, has previously been used as a benchmark on landmark models such as GPT-2 [63].

Our image classification task uses the standard *ImageNet* [17] dataset. ImageNet is one of the most popular vision datasets in use, incorporating 14M images over 1000 classes.

Model Configurations: Table 3 summarizes the model selection configurations used in the workloads. Our language modeling workload (*TXT*) uses two GPT-style models. GPT-2 is a 1.5B parameter model, introduced in 2019. GPT-J is a 6B parameter model, introduced in 2021. Both are state-of-the-art model architectures[1].

Our image classification task (*IMG*) considers the model selection setting where even the general model type is not consistent across the set. We train two model classes: (1) a large-scale instance of the Vision Transformer (ViT) with 1.8B parameters, and (2) a large ResNet with 200M parameters. This task also illustrates the setting wherein scale is a key challenge for only part of the workload, motivating usage of different parallelisms.

System Configuration: We implement and register four parallelisms for search and evaluation.

(1) PyTorch Distributed Data Parallelism [42].
(2) PyTorch Fully-Sharded Data Parallelism [42].
(3) GPipe, adapted from an open-source implementation [36].
(4) Model Spilling, provided by the FairScale library [9].

We set the SPASE solver's introspective threshold and interval to 500s & 1000s respectively.

[1]Architectures such as GPT-3 or BLOOM are simply too large (175B+ parameters) for us to evaluate. These extreme-scale models are not accessible to most practitioners; their exclusion should not reduce the value of these results.

Table 3: Model selection configurations of workloads.

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

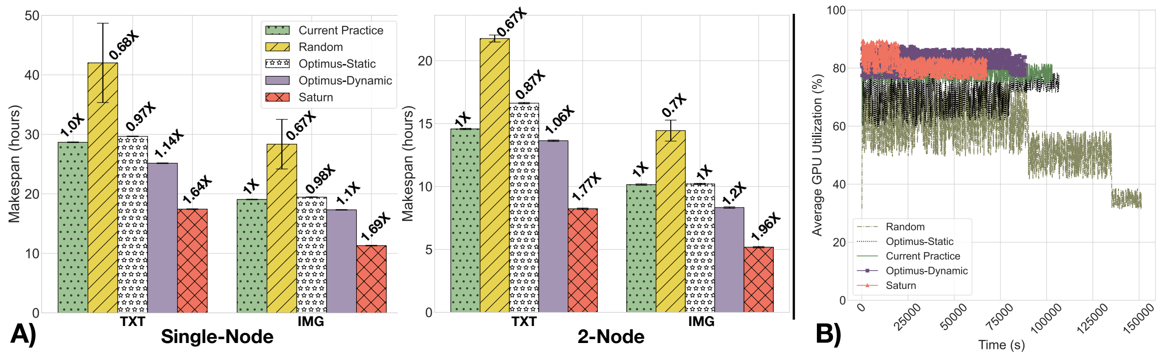

Figure 7: (A) End-to-end runtimes. Speedups versus current practice are shown by each bar. Results are averaged over three trials, with error bars displayed. (B) Average GPU utilization over time with a 100s sampling rate on the single-node TXT workload.

Hardware Setup: We configure two hardware settings: an 8-GPU single-node config, and a 16-GPU 2-node config. Both settings use nodes with 1152GB RAM, Intel Xeon 3.0GHz 2nd Gen CPUs, a 500GB NVMe drive, and 40GB A100 GPUs connected via NVSwitch. Datasets are copied across both nodes up-front, and multi-node management is handled by Ray.

Baselines: We create four baselines representative of current practice. No existing end-to-end system can solve the SPASE problem — prior art either fails to support large models or else fails model selection constraints. Instead, we adapt our baselines from Section 4.3 to cover current practice for both expert and non-expert users.

(1) A heuristic approach which does not employ any task parallelism within nodes. Instead, it allocates 8 GPUs per task. Parallelism selection is set by a human to "optimal" choices for the 8-GPU setting. We name this baseline *Human+Heuristic*. This is most representative of current practice.

(2 & 3) Two modified versions of the Optimus greedy resource allocation algorithm combined with a random scheduler [57]. We describe these algorithms in detail in Section 4. We name these baselines *Optimus-Dynamic* and *Optimus-Static* respectively. This is the strongest baseline for large-model model-selection we could build using prior art.

(4) A randomizer tool which randomly selects configurations then applies a random scheduler. This represents a user without the domain knowledge to implement the heuristic or algorithmic approaches. We name this baseline *Random*.

These baselines cover both current practice and a reasonable algorithmic approach to resource scheduling for large-model model selection workloads. The Optimus baselines use the Trial Runner as an oracle for runtime estimates, as the original Optimus system only included performance models for PS-style data parallelism. This highlights the novelty of our problem setting — the strongest baseline available from prior art must use a module of our system!

5.1 End-to-End Runtimes

Model Selection Time: We first evaluate our system's end-to-end runtimes versus the four baselines. The Trial Runner search overheads *are included* in our system's results. Figure 7 (A) illustrates.

Saturn achieves significant speedups versus all baselines. In the single node setting, we achieve a 33% and 38% makespan reductions versus the strongest baseline on the language and vision workloads respectively. The gap grows to 35% and 46% with two nodes.

Against current practice (i.e. the heuristic approach) we achieve 1.8X-2.5X speedups. The same UDP implementations are used *in all cases*, so these speedups are achieved purely through parallelism selection, resource allocation, and schedule construction. This also means all evaluated approaches use logically equivalent SGD; there is no accuracy degradation through approximate execution.

We also report on GPU utilization (excluding the search phase) in Figure 7(B). All baselines use the same underlying parallelism implementations, so this metric is fairly consistent across approaches. Even so, Saturn achieves marginally better GPU utilization than all other approaches.

Overall, Saturn reduces model selection runtimes substantially for all workloads in all evaluated configurations. Saturn also provides more qualitative benefits to practitioners. When using Saturn, practitioners do not have to manually select parallelisms or decide on resource allocations.

Intuition on Effectiveness. Saturn's performance improvements are achieved through its *holistic* optimization approach. To the best of our knowledge, this is the first work that acknowledges parallelism performance crossovers and incorporates them into its joint optimizer. Our empirical evaluator and general SPASE formulation allows us to optimize in a parallelism-agnostic fashion. The heuristic and algorithmic baselines make assumptions about scaling behaviors (e.g. consistency, effective scaling) that do not hold up in modern large-model DL practice. We see the benefits of our system-optimized approach in the mixture of parallelisms Saturn selects. In Table 4 we show the parallelisms/allocations selected by system for a few models taken from the single-node workloads.

Table 4: Apportionments/parallelisms 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

Few domain experts would naturally settle on these heterogeneous configurations, since they would consider model scaling behaviors *individually.* Saturn optimizes *holistically, across models*, and selects these “unintuitive” configurations to better fit the schedule. We will include all Gantt charts, allocations, and parallelism selections in an extended version after publication for pedagogical purposes.

5.2 Drilldown Studies

Contribution of Our Optimizations: We separate our optimizations into layers: MILP scheduling, resource allocation, parallelism selection, and the introspection overlay. We apply these one-by-one to understand how they contribute to performance 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 8 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.

Next, we try and understand how Saturn's performance scales with respect to: (1) workload size, (2) node size, and (3) model size.

We start with our workload size scaling experiment. For this, we use *TXT* on a single 8-GPU node, fix the model to GPT-2, the batch size to 16, and vary the number of explored learning rates. Figure 8(A) presents the results. Saturn's global joint optimizer

Table 5: Ablation study. Speedups normalized to unoptimized.

Optimizations	Abs. Speedup	Marg. 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

slows down sublinearly. This implies it will perform well on large workloads — even as search and execution times increase.

Next, we vary model size and evaluate the impact on Saturn's performance. We use the TXT workload on a single 8-GPU node with batch size fixed to 16, learning rate to 1e-5. All models are versions of GPT-2; we keep the size consistent across models. We vary model depth/size by stacking more Transformer encoder blocks. A similar approach was used in the design of GPT-3 [14]. Figure 8(B) presents the results. Saturn achieves mostly linear scaling, but over time, superlinear slowdowns become necessary. Very large models force the optimizer to resort to a single low-performance, high-memory configuration (8-GPU FSDP with both checkpointing and offloading) for every model. This study shows how our Saturn's performance can be limited by the underlying UDP implementations; while it can solve SPASE effectively, it still needs performant UDP options.

In our final study, we run an experiment where we vary the number of GPUs visible to Saturn. We switch to two nodes for the 16-GPU setting. We use the full *TXT* workload for this study. Figure 8(C) presents the results. Saturn achieves superlinear speedups for two reasons. First, the initial single-GPU baseline uses spilling, which introduces slow communication over the memory hierarchy. As the number of GPUs increases, the amount of data that needs to be spilled decreases and a wider parallelism space opens up, improving performance superlinearly. Second, increased GPU counts open up a broader valid MILP solution space with more potential parallelisms and apportionment, giving it the flexibility to explore more performant configurations.

6 RELATED WORK

Saturn studies resource allocation for offline scheduling as well as the broader SPASE problem — both firsts in the DL space. These optimizations are enabled through background from DBMSs and scheduling applied to the challenges of modern DL (large models, varied parallelisms, model selection needs). In Section 1.1, we described related areas of work to illustrate how this novel problem setting is unaddressed. Here, we do a deep dive into specific tools and systems across these areas.

Parallelism Selectors: Paleo [60] was an early work focused on evaluating standard iimplementations of data parallelism and model parallelism. The parallelism landscape has changed a great deal since then, and dozens of new approaches to parallelism have been proposed. In the future, Paleo's models could be extended to support these paradigms. Until then, our empirical Trial Runner is the only generalized parallelism performance estimator we know of.

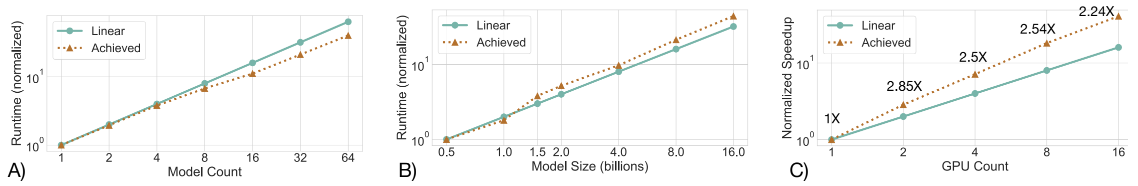

Figure 8: Saturn sensitivity plots versus (A) workload size, (B) model size, and (C) node size. All charts are in log-log scales, with performances normalized to the inital setting. Experiments are run on the TXT workload. For part (C), we label each setting with the speedup relative to the previous configuration to highlight the superlinearity.

Parallelism Hybridizers: Alpa, FlexFlow, and Unity [34, 72, 85] focus on generating bespoke parallelism strategies for model architectures through complex search procedures. These tools can produce performant single model plans, but can suffer high search overheads when applied repeatedly for multi-model training. In addition, some approaches (e.g. FSDP's offloading, spilling) are not supported by these hybrid-searchers. These tools could be implemented under our Parallelism abstraction in the future if desired.

Model selection: Two popular model selection execution systems are Cerebro [38] and ASHA [40]. Cerebro uses a hybrid form of task/data parallelism to efficiently run multiple configurations in parallel. ASHA applies data parallelism and periodically reallocates GPUs across tasks based on their "quality" as a model selection candidate. In both cases, the systems were designed prior to the "large-model era" and rely solely on data parallelism or task parallelism. As a result, neither can support large models through techniques such as pipelining, hybrid parallelism, or spilling.

Nautilus [52] optimizes model selection for transfer learning. α-NAS [35] proposes a method for creating architecture search workloads. These works are orthogonal to our own — they create/-modify the model selection workload that Saturn executes.

DL Cluster Schedulers: Fixed-input DL cluster schedulers such as Gandiva, Apollo, Tiresias, and Antman target a different setting from our own [10, 13, 25, 80, 81]. They require the user to specify the resource request manually as part of a "service-level agreement" (SLA). This SLA must be honored, guaranteeing jobs the resources they request. Gandiva does offer post-hoc rescaling for elastic jobs, but does so opportunistically without knowledge of scalability. It is an "as-and-when" optimization for GPU reallocation rather than a true resource scheduling mechanism.

These systems, and other scalable orchestrators like Pathways [12] tackle challenges that emerge with mass-scale clusters. By contrast, we optimize and automate aspects of large-model model selection. Our system could theoretically be integrated with these others. The larger fixed-input cluster manager could allocate some nodes to Saturn, which could then operate within that apportionment.

Gavel [53] schedules over heterogeneous resources (e.g. K80 GPUs and V100 GPUs). In the future, Gavel's metric for performance on heterogeneous accelerators could be incorporated into Saturn to enable optimization even on heterogeneous clusters.

DL Resource Schedulers: Pollux and Optimus [57, 58, 62] are DL resource schedulers. They tackle resource apportionment and scheduling — a subset of SPASE. Both are limited to data parallelism only and cannot train larger-than-GPU-memory models.

Neither Optimus nor Pollux supports model selection workloads. They focus on a local metric (throughput) rather than a global metric (makespan) better suited to batch jobs. In addition, both transparently alter DL workload knobs — thus violating the *fidelity* requirement of model selection. A config plugged into Pollux (e.g. batch size X and learning rate Y) will not produce results corresponding to that batch size/learning rate.

Other works have addressed scheduling in different settings such as DL inference [11, 83] or non-DL compute [6, 20, 23, 28, 74]. These works target a different setting entirely, and the optimizations they propose are orthogonal to our own.

System Parameter Optimizers: KungFu [46] provides an interface for users to express various monitoring/rescaling procedures for mid-training system parameter changes. Litz [61] provides a programming model for elastic parameter server data parallelism. TeraPipe uses dynamic programming to optimize the partitioning and execution of pipeline parallelism [43]. Compilers like Rammer [45], GO [86], TVM [15], and compiler autotuners [59, 73] provide similar up-front optimizations for DL workloads. In general, these automated search procedures are orthogonal to our own work and could be incorporated into our Parallelism Library in the future through the search function of our Parallelism API.

Other DL System Optimizations: Other optimizations such as model compilation [3, 15, 37, 39, 59], layer batching [44, 54, 84], model compression [27, 68], and graph substitution [32, 33, 72] are orthogonal to our work. A vast number of systems (e.g. DeepSpeed [64, 65, 67], Megatron [1, 55, 70], Hotline [7], HugeCTR [78], RecShard [5, 56]), and Switch Transformers [21] and propose new parallelism strategies, all expressible under our Parallelism API.

Data modification optimizations [76, 77] are also orthogonal to our own work; we do not restrict the data pipeline in any way.

7 CONCLUSIONS AND FUTURE WORK

DL practice has shifted to a new paradigm driven by large-model architectures and novel forms of parallel training. Navigating the emerging space of parallelism approaches is necessary to maximize training efficiency. Unfortunately, the intersection of parallelism selection with model selection is a largely unstudied, unmanaged problem. Automated and efficient large-model model selection is not possible with existing tooling. In this work, we formalize the joint SPASE problem, tackling parallelism selection, resource apportionment, and schedule construction. We build an end-to-end system we call Saturn to handle SPASE for large-model model selection. Saturn reduces model selection runtimes by as much as 50% versus current practice while also improving usability and minimizing practitioner effort.

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

[4] 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.

[5] 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

[6] 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

[7] 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

[8] 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

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

[10] 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.

[11] 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

[12] 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

[13] 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.

[14] 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

[15] 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

[16] 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

[17] 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.

[18] 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

[19] 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

[20] 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

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

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

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

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

[25] 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.

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

[27] 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).

[28] 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.

[29] 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).

[30] 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

[31] 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.

[32] 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

[33] 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.

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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)*.

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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).

[45] 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.

[46] 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)*.

[47] 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.

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

[49] 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

[50] 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

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

[52] 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.

[53] 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

[54] 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.

[55] 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.

[56] 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

[57] 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.

[58] 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

[59] 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

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

[61] 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)*.

[62] 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)*.

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

[64] 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

[65] 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

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

[67] 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

[68] 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).

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

[70] 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

[71] 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

[72] 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.

[73] 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

[74] 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

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

[76] 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.

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

[78] 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

[79] 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

[80] 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.

[81] 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.

[82] 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]

[83] 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

[84] 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.

[85] 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

[86] 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