Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling

Suhas Jayaram Subramanya
Carnegie Mellon University

Daiyaan Arfeen
Carnegie Mellon University

Shouxu Lin*
Cornell University

Aurick Qiao
Petuum Inc.

Zhihao Jia
Carnegie Mellon University

Gregory R. Ganger
Carnegie Mellon University

Abstract

The Sia[1] scheduler efficiently assigns heterogeneous deep learning (DL) cluster resources to elastic resource-adaptive jobs. Although some recent schedulers address one aspect or another (e.g., heterogeneity or resource-adaptivity), none addresses all and most scale poorly to large clusters and/or heavy workloads even without the full complexity of the combined scheduling problem. Sia introduces a new scheduling formulation that can scale to the search-space sizes and intentionally match jobs and their configurations to GPU types and counts, while adapting to changes in cluster load and job mix over time. Sia also introduces a low-profiling-overhead approach to bootstrapping (for each new job) throughput models used to evaluate possible resource assignments, and it is the first cluster scheduler to support elastic scaling of hybrid parallel jobs.

Extensive evaluations show that Sia outperforms state-of-the-art schedulers. For example, even on relatively small 44- to 64-GPU clusters with a mix of three GPU types, Sia reduces average job completion time (JCT) by 30–93%, 99th percentile JCT and makespan by 28–95%, and GPU hours used by 12–55% for workloads derived from 3 real-world environments. Additional experiments demonstrate that Sia scales to at least 2000-GPU clusters, provides improved fairness, and is not over-sensitive to scheduler parameter settings.

CCS Concepts: • **Theory of computation → Scheduling algorithms**; • **Software and its engineering → Cloud computing**.

Keywords: cluster scheduling, resource allocation, deep learning training

*Work done while at Carnegie Mellon University

[1]In Egyptian mythology, Sia is the god of perception/intelligence [1], not to be confused with the popular music artist [2].

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613175

ACM Reference Format:
Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613175

1 Introduction

Sizable deep learning (DL) clusters, often shared by multiple users training deep learning models for different problems, have become data center staples. A scheduler is used to assign cluster resources to submitted jobs. Increasingly, DL clusters consist of a mix of GPU types[2], due to incremental deployment over time and advances in GPU design.

Recent work provides powerful schedulers for DL clusters, but none utilize heterogeneous DL clusters well. To help explain why, we partition existing schedulers into two categories. *Heterogeneity-aware* schedulers [32, 40, 56] explicitly consider differences among GPU types in the cluster, with Gavel [40] as a state-of-the-art example, but existing options only accommodate what we term *rigid* jobs. ("Rigid" jobs must run with a user-specified number of GPUs, do not allow elastic scaling, and do not adapt to resource assignments.) *Adaptivity-aware* schedulers [43, 44, 50] explicitly consider how non-rigid jobs would adapt to (e.g., batchsize adjustments) and perform with different numbers of GPUs, with Pollux [44] being a state-of-the-art example, but existing options assume that the cluster's GPUs are all the same type.

Figure 1 illustrates the resulting problem. When only one degree of freedom (heterogeneous GPUs or adaptive jobs) is present, a state-of-the-art scheduler for addressing it provides good performance. But when both are present, much opportunity is lost (see 40–70% lower average JCTs in the middle trio of bars) because existing schedulers do not consider both. Worse, for more intense workloads the gaps grow larger (e.g., see Figures 7 and 9), because these schedulers scale poorly with contention (Gavel) and cluster size (Pollux).

Sia is a new scheduler designed for *resource-adaptive* DL training jobs **and** *heterogeneous* resources, matching each state-of-the-art for their category but outperforming them

[2]For conciseness, we will use "GPU" to refer to any accelerators used for DL model processing generally, including both traditional GPUs and various others like TPUs [25], FPGAs, and other ML accelerators [4, 22, 31].

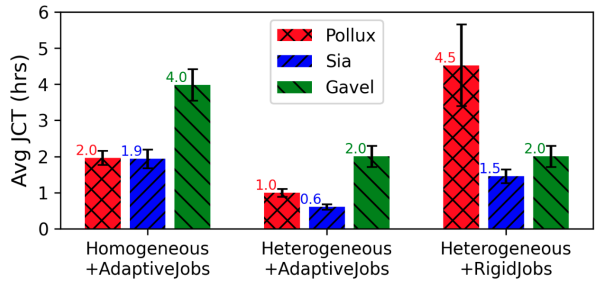


Figure 1. Scheduler comparison for three scenarios. [Left] For resource-adaptive (non-rigid) jobs on a homogeneous cluster, the left-most bars show that Pollux and Sia yield lower average job completion times (JCTs) than Gavel. [Right] For rigid jobs on a 3-GPU-type heterogeneous cluster, on the right, Gavel and Sia outperform Pollux. [Center] For non-rigid jobs *and* heterogeneous resources, in the middle, Sia outperforms both state-of-the-art schedulers built for only one of the two complexities. (The trace and cluster configurations are detailed in Section 4; the heterogeneous cluster includes some faster GPUs, causing JCTs to decrease for all schedulers.)

when both degrees of freedom are present. Conceptually, in each scheduling round, Sia considers every possible assignment of GPUs (number and type) to current jobs, estimates their aggregate "goodput"[3] (including any job resizing costs), and selects the best cluster resource assignment for the next period of time. This is challenging for two fundamental reasons: (1) the search space is huge, for a sizable cluster, and much worse when there are multiple GPU types and each job can use and adapt to any number of GPUs of any type; (2) different DL jobs experience different performance changes when comparing one GPU type to another, when increasing the number of GPUs (i.e., one may scale better than another), and when comparing scaling with one GPU type to scaling with another (e.g., different GPU types can have distinct compute-to-network-bandwidth ratios), and yet profiling each DL job for all possible resource allocations is prohibitively expensive.

Sia addresses these challenges with a new solver formulation to deal with scale and a new approach to online learning of per-job per-GPU-type throughput models. Sia's new ILP formulation, together with pragmatic search space reductions, allows it to efficiently find assignments of GPU types, GPU counts, and batchsizes for all pending jobs even as load and cluster size grow. Sia's new approach to throughput modeling (as a function of GPU type, GPU count and batchsize) avoids extensive profiling, which could override scheduling benefits. Instead, Sia bootstraps each new job's throughput model with profiles of just one minimum-sized[4] configuration per GPU type, initially assumes simple scaling/projection across as-yet-unknown configurations, and dynamically refines the model as different configurations are used for the job. Experiments confirm that Sia's approach yields good decisions with low profiling overhead.

Extensive evaluations with workloads derived from three real cluster environments show Sia's effectiveness, scalability, and superiority to three state-of-the-art schedulers (Pollux, Gavel, and Shockwave [61]), as well as others. Sia is implemented as a plugin-compatible scheduler replacement in the open-source AdaptDL framework [20], allowing us to perform head-to-head comparisons with the public Pollux implementation. Experiments with Sia, Pollux, and Gavel on a 44-GPU 3-GPU-type cluster show that Sia provides 35% and 50% lower average JCTs (avgJCT) than Pollux and Gavel, respectively. Importantly, these experiments also re-validate the simulator from [44], which we use for broader explorations, including larger clusters than we can obtain and more intense workloads. Indeed, we find that Sia's advantages grow with cluster load/contention, especially compared to Gavel (up to 95% lower avgJCT) and Shockwave (up to 47% lower avgJCT), which treat all jobs as rigid.

Overall, the results show that, for adaptive jobs on a heterogenenous cluster, dynamically adapting job resource assignments (GPU type and count) is crucial and results in Sia outperforming all three state-of-the-art schedulers on all performance metrics considered: 30–93% lower average JCT, 28–95% lower p99 JCT, 38-65% lower makespan, 12–60% lower GPU hours used. Sia also outdoes the other schedulers on fairness metrics [34, 61], including 64% lower worst-case finish-time fairness and 99% lower unfair job fraction, even though Shockwave was designed to provide fairness. Additional results confirm Sia's (1) ability to improve cluster efficiency even when many jobs disallow changing of batch-size or GPU count, (2) ability to schedule and elastically scale Megatron[41]-style pipeline-model-parallel [18, 39] jobs (scale-out using data-parallelism), (3) scheduler-runtime scalability to sizable clusters (up to 2000 GPUs), (4) robustness to scheduler-parameter defaults, and (5) minor penalty for initially-crude bootstrapped throughput-models.

This paper makes five primary contributions: (1) it exposes a gap in state-of-the-art schedulers that leaves a large untapped opportunity; (2) it introduces a new scheduler (Sia) with an ILP formulation that addresses the compounded complexity of heterogeneous GPU types and job adaptivity; (3) it shows that per-job per-GPU-type throughput models can be bootstrapped from observing just a few mini-batches up front and then quickly and effectively refined as the job runs in Sia-optimized configurations; (4) it presents the first cluster scheduler able to elastically scale hybrid parallel jobs; (5) it shows that Sia matches state-of-the-art schedulers in their target domains and significantly outperforms them in the union of their domains' complexities. Results also show Sia's scalability, fairness, and parameterization robustness.

[3]"Goodput" [44] is a DL efficiency metric that combines sample-processing throughput and statistical efficiency to reflect *rate* of training progress.
[4]For traditional data-parallel jobs, the minimum size is 1 GPU. For forms of model-parallel (e.g., pipeline parallel [18, 39]), which we term "hybrid parallel", the submitter-specified number of GPUs will be the minimum.

2 DL cluster scheduling and related work

A deep learning (DL) training job trains a deep neural network (DNN) model on a dataset in an iterative manner over multiple *epochs*. In each *epoch*, and for each *minibatch* in an epoch, an optimizer updates model parameters by minimizing a loss function over the minibatch of samples. Since the minibatch size is usually fixed for extended periods of training (if not the entirety of training), most DL jobs take a consistent and predictable [52] amount of time to complete a minibatch. These jobs are also generally *pre-emptible*, as one can checkpoint the state of the job (including the model and optimizer states) after any minibatch and resume the job from a checkpoint without losing much job progress. They are also amenable to scaling as gradient computation can be parallelized across multiple GPUs on a single node and across multiple nodes[5, 9, 10, 23, 42, 48].

Although various parallelization strategies exist [18, 23, 39, 41, 49], most training jobs use *synchronous data parallelism* (DP)—given a set of GPUs, each GPU receives a *replica* of the model and computes gradients on a partition of the minibatch, whose size is termed *local batch size*. After the gradients from all the GPUs are reduced to a minibatch gradient, such as by a collective *all-reduce* [42, 48], an optimizer (e.g., SGD or Adam [27]) applies the gradient to generate the updated model parameters on each GPU. How well a given DL job scales depends on characteristics of the job (e.g., compute intensity and number of model parameters), the GPUs, and the inter-GPU network: for each minibatch, the gradient computation phase is divided among the GPUs, while the reduce phase synchronizes them. Prior work has shown that job scalability can be modeled effectively with relatively few measurements [43, 44].

Some DL jobs use forms of *model parallelism*, such as pipeline model parallelism (PMP[18, 39]) or Tensor Model Parallelism (TMP[41]), when the model being trained is too large to fit in a single GPU's memory. Powerful optimizers [39, 53, 60] exist for modeling performance of different configurations and partitioning a model across GPUs to maximize performance. Recently, some increase scale by mixing multiple parallelism types—e.g., Megatron-LM[41] mixes PMP and TMP at moderate scales and then employs synchronous data-parallelism to scale out to 100s of nodes.

Elastic and resource-adaptive DL jobs. Data-parallel DL jobs can be elastically re-sized over time, by checkpointing and then restarting on a different number of GPUs (with a different division of each minibatch's samples). Moreover, aspects of how the job does its work can be adapted to assigned resources, if the job is designed to do so [44, 59]. For example, the minibatch size can be adapted, such as by increasing its size when using more GPUs in order to increase the per-GPU compute for each minibatch and thereby increase scalability. Different minibatch sizes do have different statistical efficiency impacts, and the differences depend on job characteristics, but this effect can also be measured and modeled [36, 44].

Other DL jobs are usually submitted to a scheduler with a predetermined configuration, and changing it usually requires re-running the hybrid parallel optimizer. As discussed above, however, they can also be scaled using a data-parallel style by replicating the original configuration: for example, a PMP job that requires 4 GPUs for model and selected minibatch-size could use 8 GPUs and a doubled minibatch, one for each 4-GPU instance of the original configuration [41].

Resource heterogeneity. There are many *GPU types*, representing different product lines and generations produced by different vendors, and they naturally differ in GPU memory size and in compute and communication performance. It is common for a DL cluster to contain multiple GPU types. In part, this occurs because many clusters are deployed and grown over time, and the most cost-effective option can be selected each time new hardware is purchased and added. Looking ahead, the rapid development of new DL accelerators [4, 22, 25, 31], including some targeting specific DL models [58], will make having multiple "GPU types" a design feature rather than a deployment consequence. Unsurprisingly, a DL job may perform differently on different GPU types, and it can also scale differently for different GPU types (e.g., because the compute-to-network ratio changes). In addition, as illustrated in Figure 2 different DL jobs can experience different speedups and different scalability.

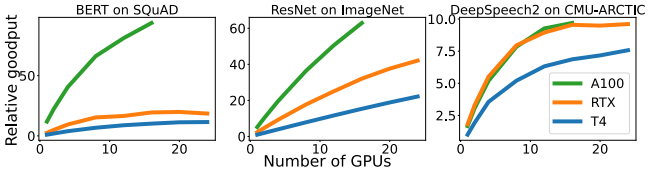


Figure 2. Scaling of goodput with number of GPUs for different GPU type and training job (Table 2) combinations. For each job type, goodput is shown relative to single-T4 goodput.

DL Cluster Schedulers. In practice, DLT jobs are submitted as requests to a shared cluster, and the scheduler assigns resources to achieve cluster-wide goals. Many schedulers only accommodate requests that specify a fixed number of GPUs, ignoring opportunities presented by elasticity, resource-adaptivity, and heterogeneity. Others do address some of these opportunities. Sia seeks to address them all.

2.1 Related work in DL cluster scheduling

To our knowledge, no prior scheduler optimizes assignments for resource-adaptive jobs on a heterogeneous DL cluster. This section groups prior schedulers by unaddressed aspects.

Scheduling for heterogenenous DL clusters (no resource-adaptive jobs). Among DL schedulers that are designed to handle heterogeneity within a cluster [8, 32, 40], none adaptively tune the number of GPUs assigned nor account for other potential adaptations made by DL jobs.

Instead, the user specifies a number of GPUs for each job submitted.

Gavel[40] is the best-performing state-of-the-art heterogeneous DL cluster scheduler, using a fast linear-program formulation that scales to large cluster sizes. However, Gavel does not support job adaptivity, and only optimizes the assigned GPU type given the minibatch size and GPU count specified by job submitter. This approach may lead to underutilization of newer, more powerful GPUs because of too-small batch sizes. Also, when the cluster is congested, Gavel time-shares resources between jobs, wasting GPU time on checkpoint-restore operations.

Most importantly, extending Gavel to handle job adaptivity is non-trivial: Gavel expresses scheduling options using a throughput matrix populated with (job_id, GPU_type) pairs. If one simply expands the throughput matrix to contain entries for each adaptivity choice (job_id, GPU_type, num_GPUs, minibatch_size), it leads to two problems – (1) populating a non-trivial portion of this matrix will require extensive per-job profiling, and (2) the resulting optimization program is too large to be solved quickly.

Scheduling for elastic and resource-adaptive jobs (no heterogeneity). Among DL schedulers that are designed to tune for elastic and resource-adaptive jobs [19, 43, 44, 50, 57], none consider GPU heterogeneity—they assume that all GPUs in the cluster are identical.

Pollux [44] is a state-of-the-art DL cluster scheduler for elastic resource-adaptive jobs for homogeneous clusters. Pollux uses per-job goodput models to assign both a number of GPUs and a batchsize setting to each current job, and it re-considers all assignments each scheduling cycle based on updated job behavior and job queue information. By doing so, it exploits elasticity to avoid unused or over-committed GPU resources. Each job's goodput model consists of two component models: one for statistical efficiency (a rate of training progress per sample, based on Gradient Noise Scale [36]) as a function of batchsize, and one for throughput (samples processed per second) as a function of both GPU count and batch-size. How each job scales with GPU count is learned by scaling it up, measuring each count tried, and interpolating for others. Pollux uses the per-job models with a genetic algorithm to search the space of resource allocations (and corresponding batchsizes) for all current jobs to maximize aggregate cluster-wide goodput weighted by fairness. Unfortunately, Pollux's no-pre-profiling throughput modeling approach blocks consideration of GPU heterogeneity. Worse, Pollux's formulation of the scheduling problem as a genetic optimization problem results in very poor cluster-size scaling even for homogeneous clusters with 100s of GPUs, which would only worsen with GPU heterogeneity.

This is because Pollux considers a very large number of adaptivity choices: for each (job, GPU_count) pair, it considers every possibly way to place this job across all nodes. As a result, the number of possible solutions is exponential in the number of nodes and number of GPUs per node. For clusters with 1000+ GPUs, it is too slow to respond to changes in cluster as it takes tens of minutes for the genetic algorithm to terminate (see Figure 9).

Scheduling for rigid jobs on homogeneous DL clusters. Most existing DL schedulers require the submitter to specify GPU count (and job configuration) for each job [15, 34, 61]. These schedulers do not adjust the number of GPUs assigned based on current load or scalability/efficiency of current jobs. They also do not consider GPU type differences, instead assuming that all GPUs in the cluster are identical. As such, these schedulers use DL cluster resources less efficiently than schedulers from the two prior categories [40, 44]. As a recent example, Shockwave [61] improves performance and fairness relative to prior schedulers in this category, and we include it in our evaluations. Some batch-schedulers like Kubeflow and Volcano can adjust GPU count to improve GPU utilization, but do not co-adapt batch-size, GPU count and type simultaneously.

Parallelism optimizers that are not cluster schedulers. There are various optimizers [7, 23, 39, 53–55, 60] for selecting an individual job's configuration before acquiring cloud resources for it or submitting it to a cluster scheduler. Such optimizers are especially important and popular for hybrid parallelism approaches. However, they cannot be considered cluster schedulers as they consider an individual job in isolation, without considering cluster load or trade-offs in assigning a particular resource to one job rather than another. To our knowledge, no existing scheduler co-optimizes non-data-parallel job configurations and cluster resource assignments, even for homogeneous clusters.

3 Sia Design and Implementation

Sia is a pre-emptive, round-based scheduler that optimizes allocations for a set of jobs to maximize cluster-wide goodput. In each round, jobs receive bundles of resources (CPU, GPU and network, like VMs in cloud) and Sia uses checkpoint-restore preemption to optimize job adaptivity.

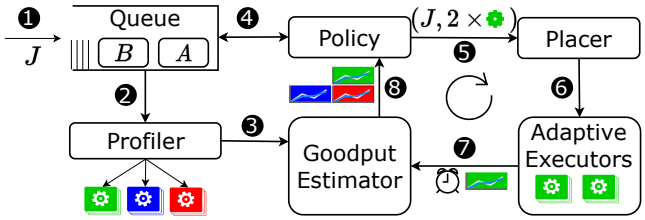


Figure 3. Lifecyle of a job under Sia. After a job is submitted, it is profiled once on each GPU type for a few batchsizes. Upon receiving an allocation, the job begins a cycle of continuous optimization (steps 5-8) for the remainder of its life in the cluster. *Policy* continuously optimizes allocations for the job, while *Goodput Estimator* provides up-to-date performance and gradient statistics to *Policy* to aid in decision making.

3.1 Sia components and job life cycle

Figure 3 illustrates the life-cycle for a job J under Sia. A user submits a job J to Sia (①) and declares both the maximum batchsize (`max_bsz`) and GPU count (`max_ngpus`) for execution. Sia then profiles throughput of J on a few batchsizes using one GPU of each type(②). Goodput Estimator bootstraps a throughput model for J on each GPU type using the profiles. Goodput estimates for J on various resource configurations are provided to the policy optimizer(⑧) for informed scheduling. Job J stays in the queue (④) until Sia allocates some GPUs to it and then enters a cycle where its adaptivity is continuously optimized by Sia as follows.

Continuously optimized job adaptivity. Sia Policy uses goodput estimates from each job's Goodput Estimator and finds an optimal partitioning of cluster resources among the jobs in the cluster, giving job J, say, 2 GPUs of type GREEN (⑤ in Figure 3). (The Goodput Estimator combines Sia's throughput model with a statistical efficiency model borrowed from Pollux [44].) Placer then determines the 2 GPUs to assign to job J (⑥) given the current assignment of GPUs to jobs and attempts to reduce unnecessary job migrations due to resource de-fragmentation. Sia runs jobs on Adaptive Executors that support (1) transparent checkpoint-restore for low-overhead job pre-emption and resource scaling, (2) batchsize adaptivity to maximize statistical efficiency, and (3) frequent reporting of gradient and throughput statistics for current allocation (default = 30 seconds). After J starts running on Adaptive Executors, Goodput Estimator uses J's gradient and throughput statistics (reported by Adaptive Executors) to update the goodput model for J on GPU type GREEN(⑦). In the next scheduling round, Sia Policy queries the updated goodput estimates for J on all GPU types (⑧) and completes the loop in the Sia architecture (⑤→⑥→⑦→⑧...), allowing us to continuously optimize J's goodput until its termination/completion.

Heterogeneous Execution. Sia transparently handles GPU heterogeneity in number and capabilities – GPU memory capacity, interconnect speeds, throughput are modeled in the goodput estimator, and Adaptive Executors optimize for goodput given a fixed set of resources. Gradient accumulation is used if statistical efficiency dictates higher batchsize than supported by GPU memory limits, with goodput optimized over a larger range of per-GPU batchsizes for GPUs with larger memories, fully exploiting whichever GPU type for optimal job progress.

Job Scaling policy. Sia uses a simple scale-up policy – start each job with exactly 1 GPU, and scale the job up by a maximum of 2× in each scheduling round. If a job requires a minimum of `min_ngpus` to start execution, Sia will respect this minimum and ignores all allocations *smaller* than `min_ngpus` for this job. Jobs may also be scaled down to a minimum of `min_ngpus` to accommodate more jobs in the cluster (determined by the scheduling objective).

Decoupled allocation and placement. Given a set of heterogeneous resources to be partitioned among a set of jobs, Sia decomposes the problem into two stages – (a) an *Allocation* stage (⑤) that determines the number and type of resources to assign to each job, and (b) a *Placement* stage (⑥) that determines the exact physical resources (and the network topology) to satisfy allocations for all jobs. This decoupling allows us to restrict the space of placements for an allocation (there exist many placements for a given allocation[51]). Sia uses three rules to obtain placement in Placer: (a) partial node (fewer GPUs than max GPUs per node requested) allocations must not be split across two nodes, (b) whole node allocations must take whole nodes, and (c) if there exists no placement satisfying (a) and (b) (resource fragmentation), evict some jobs and try again. Evictions resulting from fragmentation are quite rare and often result in fewer than 3 evictions at once. As we will see in Section 3.3, restricting allocations to a particular set allows us to guarantee a placement for all valid allocations output by Sia.

3.2 Bootstrapping of throughput models

A naive approach to constructing each job's throughput model (as a function of GPU count and batchsize) for every GPU type would require profiling a variety of multi-GPU allocations for each GPU type to collect compute and communication times. This profiling overhead grows linearly in both the number of GPU types and the number of nodes of each GPU type. Sia takes a different approach, starting with minimal profiling information and refining based on observed allocations.

For each job, Sia learns one throughput model for each GPU type and one statistical efficiency model for the job. Consider a job J submitted to Sia running on a cluster with two GPU types A and B. Let's assume that J needs `min_GPU_count=>` 1 GPUs per data-parallel worker. Sia first profiles J on one GPU of each type (corresponding to ②in Figure 3). Starting from a minimum batchsize, Sia profiles increasingly larger batch sizes till it hits GPU memory limits (typically 10 profiled batchsizes per GPU type); altogether, the average per-job profiling cost is < 20 GPU seconds per GPU type. This gives us two crucial pieces of information: (1) compute times for various combinations of GPU type and batchsizes, and (2) comparison of compute times across GPU types. Importantly, compute time is independent of GPU count increases (since we scale via data-parallelism with all-reduce), this leaves only the communication time to be predicted.

Sia initializes J's throughput models for each GPU type using their 1-GPU profiles. These throughput models are used by Sia to place J on 1-GPU of some type, say A. Once J starts running on a single A GPU, online profiling is used to (a) learn a statistical efficiency model for J as a function of batch size, and (b) refine throughput model for J on 1-GPU of A type. These throughput models, however,

cannot estimate communication time, so Sia makes a **one-time** simplifying assumption to estimate J's throughput on 2-GPUs of A: *throughput of two data-parallel replicas is twice the throughput of a single replica* (i.e. perfect scaling with zero communication time). Say Sia then assigns 2-GPUs of type A to J. Using online profiling, Sia refines J's throughput model for A GPUs using the measured communication times on a multi-GPU allocation. Sia can now use the refined throughput model to estimate J's throughput on multi-GPU allocations on A GPUs as it accurately models both compute and communication time. However, since J has not yet run on a multi-GPU allocation on B GPUs, the throughput model for B GPUs does not model communication time on B GPUs as it was learned from initial profiling and cannot be used to estimate J's throughput on, say, 4-GPUs of B type. To overcome this problem, Sia combines J's learned throughput model for A GPUs with the initially profiled single-GPU throughputs for both A and B to obtain a crude *bootstrapped* throughput model for B GPUs. In our example, J's throughput on N GPUs of B type is estimated with a bootstrapped throughput model, $\texttt{est-xput}_B$, given by:

$$\texttt{est-xput}_B(N) = \frac{\texttt{xput}_B(1)}{\texttt{xput}_A(1)} * \texttt{xput}_A(N) \tag{1}$$

where $\frac{\texttt{xput}_B(1)}{\texttt{xput}_A(1)}$ is the ratio of 1-GPU throughputs and $\texttt{xput}_A(N)$ is the throughput for N GPUs of A type. This simple estimator assumes that if we do not know the communication time for B, the scaling of compute:communication ratio for B is the same as A (which is known). In Section 5.7, we show that bootstrapped throughput models are accurate enough to guide Sia towards taking useful explorative steps.

We use $\texttt{est-xput}_B$ to estimate goodput for multi-GPU allocations on B GPUs and if J runs with a multi-GPU allocation on B GPUs, we can safely discard the bootstrapped throughput model (from Equation (1)). This is because using online profiling, Sia can refine $\texttt{xput}_B$ to accurately predict communication time on B GPUs (which is now known), eliminating the need for the crude bootstrapped model $\texttt{est-xput}_B$.

3.3 Configurations

A configuration represents a bundle of resources (CPU, GPU, Network, etc) and is similar to virtual machine sizes in the cloud. Configurations can be represented as a 3-tuple – (n, r, t) where n is the number of nodes containing a total of r resources of type t. For example, $(2, 16, \texttt{T4})$ represents a configuration with 2 nodes containing 16 T4 GPUs in total.

Sia's Policy supports efficient job adaptivity by optimizing for allocations over a small *valid* set of configurations designed to simplify placement logic in Placer. This set can be decomposed into two sets: a *single-node* allocation set which contains allocations that do not cross a node boundary (i.e. $n = 1$), and a *multi-node* set that contains allocations that span node boundaries (i.e. $n > 1$). We provide a construction of these sets below.

Consider a cluster with N physical nodes, containing R GPUs of type X per node, the configuration set C is given by a union of the *single-node* and *multi-node* sets –

$$C = \{(1, 2^0, X), (1, 2^1, X), \ldots (1, R, X)\} \cup \quad \leftarrow \texttt{single-node}$$
$$\{(2, 2R, X), \ldots, (N, N \cdot R, X), n \in \mathbb{N}\} \quad \leftarrow \texttt{multi-node}$$

The *single-node* set constrains allocations to be powers of 2 within a node, and at most R, the number of GPUs within a node. If R is not a power of 2, one can decompose R as a sum of powers of 2, and model each physical node with R GPUs as multiple virtual nodes with different GPU counts. The *multi-node* set constrains all allocations to use all available GPUs in a node (i.e. GPU count is a multiple of R). Using these allocation and resource sets, we can rely on existing literature (*Submesh Shape Covering theorem* [60]) to guarantee a placement for all valid allocations where no two distributed jobs share any nodes. This is especially desirable because it eliminates resource contention on the NICs which can cause significant slowdown to all contending jobs [21, 44].

In a homogeneous cluster, Sia matches Pollux's performance (Table 4), despite optimizing over a smaller configuration set. Pollux optimizes over the full space of (GPU count x placement) choices for each job ($O(N^R)$), while Sia restricts the configuration set to a size of $(N + \log_2 R)$ for a cluster with N nodes and R GPUs each. This suggests that our restrictions do not significantly impact job runtimes. This reduction in problem complexity allows Sia's optimization to scale to clusters with thousands of GPUs (see Section 5.6) with practical runtimes.

3.4 Scheduler objective

This section describes the Sia scheduler objective. We use a running example where a heterogeneous cluster has 2 GPU types - (a) one node with 2 GPUs of type A and (b) one node with 4 GPUs of type B. Let $J = \{J_1, J_2\}$ be the set of jobs in the scheduler queue, both of which require a minimum of 1 GPU to run.

Valid configurations. Using rules described in Section 3.3, we construct the set of valid Sia configurations C. For the example cluster, $C = \{(1, 1, A), (1, 2, A), (1, 1, B), (1, 2, B), (1, 4, B)\}$. Recall that if Sia assigns a configuration $c = (n, m, X) \in C$ to a job, the job runs with m GPUs of type X split across n nodes. A job receives either no resources in a scheduling round, or a set of resources identified by a valid configuration.

Goodput estimation. Sia uses one throughput model for each (job, GPU_type) combination to model job and hardware heterogeneity effectively. Sia optimizes for goodput cluster-wide, so we use the per-job statistical efficiency models to derive goodput estimators, one for each (job, GPU_type) combination. Let (f_A, f_B) and (g_A, g_B) be the goodput estimators for jobs J_1 and J_2 and GPU types A, B, respectively. We define a goodput matrix G of size $|J| \times |C|$, where G_{ij} is the estimated goodput for job $J_i \in J$ using resources defined by configuration $c_j \in C$. For a given job J_i, all values in that row

are comparable: $G_{ij} > G_{ik}$ means configuration c_j is better than c_k for the job J_i. However, for a given configuration c_j, $G_{mj} > G_{nj}$ does not derive that J_m deserves to run in configuration c_j over job J_n. We apply a simple row-normalization technique to make values in G comparable across jobs for each configuration.

Normalized goodput matrix. For each job J_i with minimum required GPU count N_i^{min},

$$G_{ij} \leftarrow N_i^{min} \cdot \frac{G_{ij}}{\min_j G_{ij}}$$

where $\min_j G_{ij}$ is the minimum of goodput values for the job J_i across all configurations in C. The result matrix G is called a normalized goodput matrix.

Using the row-minimum values to normalize each row in G provides two benefits. First, we can interpret G as a utility-matrix for jobs J, with G_{ij} capturing the utility of configuration C_j to job J_i. Second, we can compare utilities for a given configuration across job types. Choosing the configuration with the highest value along a job's row in G makes the most progress for that job, and a configuration is best used by the job with the highest value along the configuration's column in G.

Each new job adds a row to G, and the completion of a job delete its respective row, which keeps G up to date with goodputs only for active jobs. If a job's statistical efficiency changes, or its throughput model gets more refined, G is updated to track the most recent values. For our running example, Table 1 shows the normalized goodput matrix G.

	$(1,1,A)$	$(1,2,A)$	$(1,1,B)$	$(1,2,B)$	$(1,4,B)$
J_1	1	1	2	3	**4**
J_2	2	**4**	1	2	3

Table 1. Normalized goodput matrix G. Boxed entries show the allocation that maximizes sum of goodput for jobs J_1, J_2.

Scheduler objective. G represents the utility of the set of configurations C to the set of jobs in J, and Sia selects the (job, configuration) pairs that maximize the sum of normalized goodputs for jobs in the chosen configurations. Each configuration maps to a unique allocation and by constraining the number of allocated resources, a valid schedule can be determined.

We define a binary matrix A with the same shape as G where $A_{ij} = 1$ if configuration c_j is chosen for job J_i and 0 otherwise. We formulate the problem of choosing the best pairs (as outlined above) as the following optimization problem over A:

$$\max_A \sum_{i=1}^{|J|} \Big(\sum_{j=1}^{|C|} A_{ij} \cdot G_{ij} \;+\; \lambda(1 - ||A_i||_1) \Big) \tag{2}$$

where $||v||_1$ denotes the ℓ_1 norm of a vector v. This objective is composed of two terms: a sum of normalized goodputs of jobs in all chosen configurations, and a scheduler penalty for not choosing any configuration for each job—no penalty if some configuration is chosen for job J_i ($A_{ij} = 1$ for some j, so $||A_i||_1 = 1$), and a constant penalty $-\lambda$ otherwise. The penalty λ can also be thought of as an incentive to reduce scheduler queue occupancy: if λ is large, then Sia will allocate at-least one GPU to each job in the cluster, if available.

We formulate the Sia scheduler objective as a (binary) Integer Linear Program task with the binary matrix A as an optimization variable and the following added constraints:

1. Each job chooses at-most one configuration: $||A_i||_1 \leq 1$
2. Allocated number of GPUs does not exceed available GPUs for each GPU type

Solving the optimization problem gives us a binary solution matrix A that contains allocations for the next scheduling round: a job J_i receives no resources if $||A_i||_1 = 0$, otherwise (there must exist an $A_{ij} = 1$) it runs under configuration c_j for the next scheduling round. For the normalized goodput matrix G shown in Table 1, optimizing Equation (2) gives us the allocations : J_1 gets configuration $(1, 4, B)$ and J_2 gets $(1, 2, A)$. The corresponding entries in G are each highlighted with a box in Table 1.

Restart Factor. To prevent frequent job restarts which can harm performance, a re-allocation factor, r_i, is used to adjust the utilities in G for configurations that differ from the currently allocated configuration for each job J_i. This multiplicative factor models the expected goodput for such configurations by projecting the historical rate of restarts into the future, and is necessary because the restarting cost for deep learning jobs can be high (e.g., 25-250 seconds for the models listed in Table 2). Consider a job J_i with age T_i, wasting S_i GPU seconds per *restart* operation and having restarted N_i times previously. The re-allocation factor r_i for the job J_i is computed as follows:

$$r_i = \frac{T_i - N_i \cdot S_i}{T_i + S_i} \tag{3}$$

If job J_i is currently running under a configuration c_k, we discount goodput values for all other configurations $c_j \neq c_k$ that require a restart by the restart factor: $G_{ij} \leftarrow r_i \cdot G_{ij}$. Without a restart factor, each tiny changes in G would result in altering some jobs' resources and additional checkpoint-restore overheads. By applying a restart factor to only those utility values in G that require restarting the job, Sia only restarts jobs if not doing so results in a big reduction in optimal value for its scheduling objective.

Balancing goodput with fairness. We provide a simple knob to tune *fairness* of allocations in Sia – a parameter p that can be used to manipulate the *scale-free* matrix G by raising the elements to the power of p. If $p < 0$, we flip the sign of the objective (i.e., minimize the original objective instead of maximizing) to preserve its semantics. We investigate the effect of p on Sia's scheduler metrics in Section 5.7, showing that it provides robust fairness with minimal negative impact on efficiency metrics across a range of settings between -1.0 and 1.0. We use a default of -0.5. Sia's full scheduler objective,

for $p > 0$, is as follows:

$$\max_{A} \sum_{i=1}^{|J|} \Big(\sum_{j=1}^{|C|} A_{ij} \cdot \big(r_i \cdot G_{ij}\big)^p \;+\; \lambda(1 - ||A_i||_1) \Big) \qquad (4)$$

Support for limited adaptivity. Sia supports executing jobs with some adaptations disabled (batch size, GPU count and/or type). Large batch sizes result in high throughput and GPU utilization, but may result in a generalization gap for the trained model([26, 37]): a phenomena where the final model performs poorly on unseen samples. Sia supports different types of jobs with varying degrees of adaptivity to accommodate diverse reasons for limited adaptivity: *strong-scaling* jobs run with a fixed batch size, but allow the GPU count and type to be optimized, while *rigid* jobs run with a fixed batch size and GPU count, only leaving GPU type to be optimized. Both strong-scaling and rigid jobs preserve model quality and training semantics by keeping batch size fixed, but allow Sia to optimize job execution in a limited manner. Given a fixed batch size, goodput is directly proportional to throughput; so for strong-scaling jobs, Sia directly uses throughput in place of goodput in Equation (4). For rigid jobs, we add the following objective to Equation (4):

$$\max_{B} \sum_{i=1}^{|J_R|} \Big(\sum_{g=1}^{N_g} B_{ig} \cdot \big(r_i \cdot T_{ig}\big)^p \;+\; \lambda(1 - ||B_{ig}||_1) \Big) \qquad (5)$$

where J_R is the set of rigid jobs, N_g is the number of GPU types, T_{ig} is the goodput of job J_i on GPU type g and r_i is the job's restart count. We then update constraints for the ILP to constrain total GPUs allocated for each GPU type across all active jobs.

In a similar manner, with few changes to Equation (4) and optimization program constraints, Sia's flexible scheduling formulation can support scheduling custom resource requests and jobs with user-defined parallelism tuned to a specific GPU count, type and/or batch size.

Preemption and reservation. Sia assumes all jobs are preemptive, but can also support a small number of non-preemptive jobs in the cluster (as long as their aggregate demand can be satisfied): for each non-preemptive job, we add a constraint to Equation (4) to force the requested resources to be allocated. This constraint ensures that the non-preemptive jobs get allocated first, guaranteeing non-preemption in each scheduling round. Reservations are implemented in a similar manner.

Support for other parallelization techniques. In general, Sia only requires that a job provide a goodput estimator that can be evaluated on valid configurations.

This design allows Sia to support jobs with more advanced parallelization strategies [39, 55, 60]. We extend Sia's throughput models to support jobs that use a combination of pipeline [18, 39] and data parallelism, allowing Sia to schedule jobs with multi-billion-parameter models.

These jobs employ a mix of data and pipeline parallelism [41] where a pipeline parallel strategy partitions a large model onto many GPUs, and data parallelism is used to scale up training (see Section 5.3). Each model partition is mapped to one or more GPUs, say P GPUs across all partitions. A job with N data-parallel replicas uses exactly $N \times P$ GPUs. Given a mini-batch size of M and micro-batch size of m, each replica computes gradients locally using $\frac{M}{mN}$ micro-batches of size m each across P GPUs. Then, N replicas of these pipelines synchronize using a gradient all-reduce, thus finishing one training iteration. The distinct compute and communication phases [41] allow us to leverage Sia's throughput models for goodput estimation at various batch sizes. Since these jobs scale as units of P GPUs each, we add additional terms to our scheduling objective with the appropriate constraints (similar to Equation (5)). We discuss adaptation for one such hybrid-parallel model in more detail in Section 5.3.

Existing hybrid-parallel optimizers are time-consuming[38, 60], so we leave the problem of efficient elastic scaling without fixing non-data-parallel degrees as future work.

Scheduling other workload types. Sia exploits characteristics unique to deep learning training, but we believe it could also handle other batch-processing workloads by using a goodput estimator customized to each workload type. For example, one can use Sia to schedule batch deep learning inference jobs that run inference on a large dataset. Here, throughput can be used as a proxy for goodput, yielding a simple goodput estimator. For latency sensitive inference jobs, one could use Sia to pick the right set of resources: goodput=1 if a configuration can support inference within the promised latency constraints and 0 otherwise.

3.5 Implementation

We implement Sia using the open-source AdaptDL framework, replacing its scheduler and data-loader implementations with our own, as the PyTorch-based framework provides native support for dynamically adjusting batch-size and number of GPUs for DL training jobs on Kubernetes-managed GPU clusters. For a data-parallel DL training job, we use AdaptDL data-loaders to vary batch-size during training, and use all-reduce to synchronize gradients across workers. Sia Adaptive Executor continually profiles minibatch runtimes and gradient statistics, periodically (default 30s), optimizes goodput model parameters using these profiles and communicates the new goodput model parameters to Sia Policy. It also selects the batch-size that maximizes goodput given allocated resources and scales the learning rate in accordance with the selected batch size using a configurable learning rate scaling rule. For models listed in Table 2, we use the square-root learning-rate scaling rule[29] for models using the AdamW [33] optimizer and AdaScale[24] scaling rule for models with SGD optimizers.

Sia Policy runs as a Kubernetes service, and at the start of each scheduling round, uses the latest goodput model

parameters for each job to optimize resource allocation using (Equation (4)). We formulate Equation (4) as a Mixed-Integer Linear Program using the `GLPK_MI` [35] solver from the CVXPY package [13] and use the output solution to determine job allocations.

Preemption with checkpoint-restore. If a DL training job's allocation changes, Sia preempts the job only after the current minibatch has finished processing so there is no communication in flight. First, Sia checkpoints the latest model weights, data-loader (e.g., sampler and iterator states) and optimizer states (e.g. gradient statistics for Adam[27]) to shared persistent storage and releases all GPUs allocated to the job. Then, on the new resources, Sia launches one Adaptive Executor per GPU, restores training state from the checkpoint on disk, and resumes model training.

Sia also uses the checkpoint-restore mechanism to recover from worker failures. After every epoch, Sia checkpoints model weights and optimizer states to disk, so if some workers fail in the next epoch, model training can be resumed from the last saved checkpoint on different resources.

4 Experimental Setup

We compare Sia with state-of-the-art schedulers in both homogeneous and heterogeneous clusters using workloads derived from real-world environments. This section describes the workloads, configurations and the schedulers used.

4.1 Workloads and Traces

We use traces derived from three production DL clusters, using a common approach from recent work [40, 44, 61]. We categorize each job in a trace based on its total GPU time: Small (0-1 hrs), Medium (1-10 hrs), Large (10-100 hrs) and Extra-large (XL, >100 hrs). We map each category into one or more representative jobs as listed in Table 2. XXL models are only used for hybrid-parallel experiments in Section 5.3.

Philly is from 100k jobs executed over two months in a multi-tenant cluster with multiple GPU types at Microsoft [21].

Helios is from the Saturn cluster in the Helios cluster traces [17]. The original traces contain 3.3M jobs recorded over a six-month period in a heterogeneous cluster with over 6k GPUs. Compared to **Philly**, **Helios** jobs request more GPUs and run for longer, resulting in a higher cluster load.

We derive ten traces for each workload by randomly sampling the 8 busiest hours in the respective real-world trace using an average job arrival rate of 20 jobs/hr, resulting in a total of 160 jobs submitted over the 8-hour window.

newTrace is a more recent trace from a production system for deep learning jobs that spans multiple clusters with thousands of GPUs. Similar to the Microsoft Philly traces [21], this production system allocates Virtual Machines (VMs) to DL training jobs where each VM instance is provisioned a pre-configured amount of CPU, GPU and memory resources. Similar to other production environments, we observe a wide range of resource requests exhibiting diurnal patterns with bursts of resource requests coming by virtue of job submission scripts (e.g., hyper-parameter tuning). We sample 10 traces over a 48 hour period at an average arrival rate of 20 jobs/hr (total 960 jobs submitted in each trace).

The longer 48-hour **newTrace** traces are used to evaluate a more *realistic* setting where congestion slowly builds up in a cluster from long-running jobs over a long duration. **newTrace** sees a significant variance in job arrival rates from 5 to 100 jobs/hr over the 48-hour job submission window and gives us valuable insights into how schedulers can deal with congestion and variance in cluster loads.

4.2 Hardware measurements and simulator

Most of our experiments use the discrete-time simulator open-sourced [3] by authors of Pollux whose fidelity is verified by prior work [44] and our own measurements. We added a Gavel implementation and the open-source Shockwave [61] to the simulator, as well as extended the original version of Pollux to support heterogeneous clusters. The simulator allows us to experiment on a range of cluster sizes and hardware configurations.

We use four different types of GPUs in our experiments: a cluster of 16 `t4` instances [44] and three on-premise node types (3x `rtx`, 2x `a100`, and 1x `quad`):

- `t4` – [Cloud] `g4dn.12xlarge` AWS EC2 instance with 4 NVIDIA T4 (16GB VRAM) GPUs.
- `rtx` – [On-prem] *commodity* node with 8 NVIDIA RTX 2080Ti (11GB VRAM) GPUs and 50Gb/s Ethernet.
- `a100` – [On-prem] *high-performance* NVIDIA DGX-A100 node with 8 NVIDIA A100 (40GB VRAM) GPUs and 1.6Tb/s Infiniband.
- `quad` – [On-prem] *workstation* node with 4 NVIDIA Quadro RTX6000 (24GB VRAM) GPUs and 200 Gb/s Infiniband.

We were able to get a limited amount of dedicated time with the on-prem nodes, which allowed for direct experiments on a 44-GPU, 3-GPU-type cluster. The results (Section 5.1) confirm Sia's efficacy and the simulator's fidelity.

The original simulator from [44] simulates checkpoint-restore with the same constant delay for all jobs, which we replaced with model-specific checkpoint-restore delays.

4.3 Evaluated settings

We compare schedulers in the following three settings:

- *Physical*: Physical cluster with 3 `rtx`, 2 `a100`, and 1 `quad` nodes for a total of 44 GPUs. In Sec. 5.1, we compare Sia with Pollux and Gavel.
- *Homogeneous*: Simulated cluster with 16 `t4` nodes(64 GPUs). In Sec. 5.2, we compare Sia to Pollux, a state-of-the-art job-autoscaling scheduler for homogeneous clusters, and inelastic schedulers Shockwave [61], Themis [34], and Gavel [40].
- *Heterogeneous*: Simulated cluster with 6 `t4`, 3 `rtx`, and 2 `a100` nodes (64 total GPUs). In Sec. 5.2, we compare Sia

Table 2. Models used in our evaluations.

Size	Task	Model	Dataset	Target Metric	Batch Sizes	Optimizer
S	Image Classification	ResNet18 [16]	CIFAR-10 [30]	94% Top-1 acc	[128 - 4096]	SGD
M	Question-Answering	BERT [12]	SQuAD [46]	0.88 F1 score	[12 - 384]	AdamW [33]
	Speech Recognition	DeepSpeech2 [6]	CMU-ARCTIC [28]	25% word err	[20 - 640]	SGD
L	Object Detection	YOLOv3 [47]	PASCAL-VOC [14]	85% mAP	[8 - 512]	SGD
XL	Image Classification	ResNet50 [16]	ImageNet-1k [11]	75% Top-1 acc	[200, 12800]	SGD
XXL	LLM Finetuning	2.8B GPT [45]	SQuAD	0.88 F1 score	[48, 384]	AdamW

with Pollux and Gavel, a state-of-the-art heterogeneity-aware scheduler.

Tuning job hyper-parameters. Gavel lacks support to auto-tune job parameters, so we manually tune the batch size and requested number of GPUs for each job in our sampled traces to ensure optimal performance. We follow the approach used in [44] and optimize each job's batch size and GPU count: we search over (batch size, GPU count) combinations (GPU count $\leq$ 64 GPUs for *Homogeneous*, and $\leq$ 16 GPUs for *Physical* and *Heterogeneous* settings) and randomly choose a combination (`bsz, GPU_count`) such that the simulated runtime using (`bsz, GPU_count`) is 50-80% of ideal speedup over the runtime of a 1-GPU baseline with the optimal batch size. We refer to these optimized job configurations as *TunedJobs* (TJ) in our evaluations, even though real-world jobs may be submitted with worse performing job parameters.

Fixing mixed-GPU allocations from Pollux: To make Pollux work on our heterogeneous clusters, we present 8-GPU nodes as 2 virtual 4-GPU nodes to eliminate heterogeneity in node capacities. However, Pollux may still schedule a job on more than one GPU type (not allowed in our setup). So, we apply a simple heuristic: the GPU type with the most GPUs is selected, and in case of a tie, the more powerful GPU type is chosen (`a100` > `quad` > `rtx` > `t4`). Although not perfect, this heuristic enables fair comparisons to Pollux in heterogeneous settings (Section 5). Our paper's focus is not on designing the perfect heuristic.

Default parameters. Unless explicitly stated otherwise, all experiments use the following parameters: $p = -0.5, \lambda = 1.1$ for Sia, $p = -1$ for Pollux (same as [44]), (10, 1e-1) for Shockwave (same as [61]. We choose a scheduler round duration of 60s for Sia and Pollux, and choose 360s for Gavel, Themis and Shockwave. We choose the *max-sum-throughput* scheduling policy [40] for Gavel as it results in the lowest average JCT on **Philly** traces among the policies listed in [40]. We investigate sensitivity of Sia to its parameters in Section 5.7.

5 Evaluation

This section evaluates Sia, showing that it outperforms state-of-the-art cluster schedulers for both resource-adaptive and rigid jobs running on both *homogeneous* and *heterogeneous* resources. Results also show that Sia provides better finish-time fairness, scales to large cluster sizes, and is not overly sensitive to our default parameter settings.

5.1 Physical cluster experiments

We compare Sia with Pollux and Gavel in the 44-GPU *physical* cluster setting (Sec. 4.1) that consists of 3 `rtx` + 1 `quad` + 2 `a100` nodes. We sample a smaller, single 3-hour trace with 30 jobs with a mix of all the models listed in Table 2 and run all schedulers on the physical cluster. Owing to resource availability constraints,[5] we run Sia, Gavel and Pollux four times to account for any randomness in their schedules.

Figure 4 shows the results of our physical cluster experiment side-by-side with those predicted by the simulator. On the physical cluster, Sia provided lower average JCT than Gavel or Pollux by 50% and 35–50%, respectively.

Figure 5 shows resource allocations for three jobs over 45 minutes, illustrating how Sia dynamically adjusts GPU count and type. Rising congestion triggers Sia to scale down and then move the `ImageNet` job (top) to `rtx` GPUs, leaving the fastest (`a100`) GPUs for incoming `CIFAR-10` jobs. Over time, Sia scales up and refines throughput models for each job (e.g. `DeepSpeech2` job in Figure 5) while adapting to GPU type and count changes. When congestion decreases sufficiently, Sia shifts `ImageNet` back to `a100` and scales out `DeepSpeech2` on `rtx` GPUs for better throughput.

Simulator fidelity. The simulator was found to have less than 5% error in average JCT and Makespan for both Sia and Gavel, validating its accuracy yet again. However, Pollux performed significantly worse on the physical cluster than predicted by the simulator, due in part to the modifications we made to the simulator giving Pollux an advantage when scheduling a single job over heterogeneous resources (Section 4.3). The schedules produced by Pollux can have large variations due to randomness in its optimization and its potentially *misguided* job adaptivity (due to noisy throughput estimators), resulting in bad worst-case scenarios. Additionally, the heterogeneity of the underlying hardware mapped to the virtual nodes that Pollux assumes are homogeneous can also contribute to the variation.

5.2 Simulator experiments

Table 3 shows key performance metrics for Sia, Pollux, and Gavel running on the heterogeneous cluster with traces described in Section 4.1. Across all traces and evaluated metrics, Sia outperforms heterogeneity-aware schedulers like Gavel

[5] Unlike profiling runs, our scheduler experiments require extended isolated control over the entire collection of machines, blocking out all users for which the machines were acquired.

Trace	Policy	JCT		Makespan	Avg. GPU-hours/job	Contention		Avg. job restarts
		Avg.	p99			Avg.	Max.	
Philly	Sia	**0.6h ± 0.1**	**9.5h**	**14.2 ± 1.9h**	**4.0 ± 0.7**	**6.9**	**31**	**2.9**
	Pollux	1.0 ± 0.1h	14.9h	24.5 ± 7.9h	5.6 ± 1.1	7.2	42	5.8
	Gavel+TJ	1.9 ± 0.3h	30.0h	33.8 ± 8.6h	9.0 ± 6.3	9.9	56	5.7
Helios	Sia	**0.7 ± 0.1h**	**10.9h**	**14.9 ± 1.7h**	**4.8 ± 0.7**	**7.4**	**32**	**3.4**
	Pollux	1.0 ± 0.2h	15.0h	25.5 ± 8.0h	5.9 ± 0.7	6.9	47	5.3
	Gavel+TJ	2.5 ± 0.9h	38.7h	43.0 ± 10.9h	12.1 ± 3.7	9.2	48	7.5
new-Trace	Sia	**0.7 ± 0.1h**	**4.6h**	**52.2 ± 1.3h**	**3.0 ± 0.1**	**13**	**69**	5.0
	Pollux	1.5 ± 0.2 h	10.3h	62.3 ± 4.6h	3.4 ± 0.2	22	85	5.4
	Gavel+TJ	11.3 ± 3.0h	98.1h	110 ± 21.5h	6.4 ± 1.1	96	243	**4.5**

Table 3. Comparison of Sia, Gavel, and Pollux in the *Heterogeneous* setting. TJ is short for TunedJobs, Contention is the number of jobs contending for resources in the cluster.

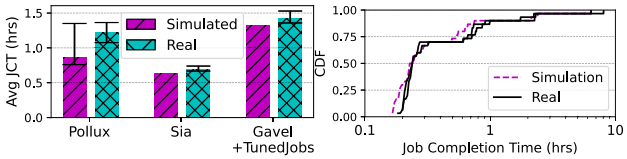


Figure 4. (Left) AvgJCTs on the *Physical* testbed, and (Right) CDF of job completion times for Sia predicted by the simulator (*Simulated*) compared to a run on *Physical* testbed (*Real*). Error bars represent the extreme values seen across 200 simulator and 4 physical cluster runs.

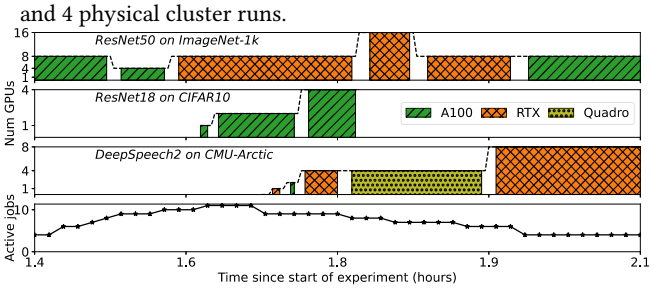


Figure 5. Resource allocations for three jobs in the Sia physical cluster experiment, along with number of active jobs in cluster. Colors indicate GPU type and whitespaces represent checkpoint-restore delays caused by Sia's scheduling decisions.

and job auto-scaling schedulers like Pollux. Sia reduces average JCT by 30–93% and 99th-percentile JCT (p99 JCT) by 28–95%, compared to Pollux and Gavel. In doing so, Sia is also more resource efficient– it allocates 12-60% fewer GPU hours per job compared to Pollux and Gavel.

Gavel+TunedJobs performs poorly compared to Sia for two reasons: (1) time-sharing overheads reduce the useful GPU time spent on training progress in a given round, and (2) using a batch size that fits the *smallest* (in memory) GPU leads to under-utilization of more powerful GPUs. Pollux outperforms Gavel due to job adaptivity, but falls behind Sia for two reasons: (1) it treats heterogeneous hardware as homogeneous, failing to exploit performance heterogeneity, and (2) it can output placements spanning more than one GPU type; fixing them so they only span one GPU type forces some GPUs into idling, but it is better than using a mix of GPU types and running at speed of the slowest.

From Table 3, we see that Pollux restarts jobs twice as often as Sia for moderately congested clusters (**Philly** and **Helios**). This is because it optimizes job allocations in steps of 1 GPU, while Sia only allocates configurations with steps as large as an entire node (as defined in Section 3.3).

Congestion in newTrace. Compared to **Philly** and **Helios** traces, **newTrace** contains bursts of up to 100 jobs/hr during the busiest hour. Gavel struggles to handle these bursts, and as congestion worsens, this problem compounds creating a positive feedback loop: rising contention forces Gavel to swap jobs in/out more frequently, wasting significant GPU capacity on executing checkpoint-restore operations when GPUs are already scarce. As a result, the average and p99 JCTs for Gavel degrade far worse compared to Sia and Pollux. During peak congestion, Sia and Pollux both scale jobs down to just 1-GPU per job, resulting in a smaller gap between them. However, Sia's heterogeneity-aware scheduling better matches jobs to GPUs, improving cluster goodput over Pollux even during congestion.

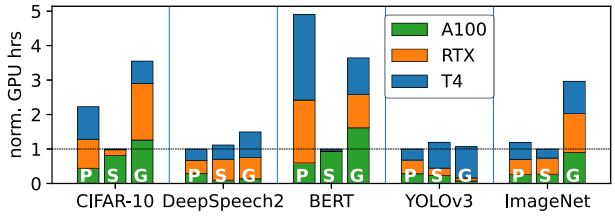


Figure 6. (Min-normalized) GPU hours consumed per model for Sia (S), Pollux (P), and Gavel (G) using **Helios** traces.

Matching jobs to GPU types. Figure 6 shows the average GPU hours used to train each model(Table 2) using **Helios** traces. Pollux is heterogeneity-unaware and has no distinct (job, GPU type) preferences, whereas Gavel and Sia are heterogeneity-aware and strongly prefer certain GPU types for particular models. Figure 6 shows that Sia allocates `BERT` models almost exclusively to `a100` GPUs, aggressively exploiting the heterogeneity in model goodputs across GPU types. Gavel's time-sharing approach, however, forces `BERT` jobs to rotate between `a100`, `rtx`, and `t4`GPUs, resulting in less-efficient execution. Similarly, Sia prefers to use `rtx`

GPUs for DeepSpeech2 and leaves the a100 GPUs free for BERT models, achieving significant reduction in GPU hours consumed per job compared to Gavel's approach. On average, YOLOv3 and DeepSpeech2 models consume about 5% more GPU hours under Sia compared to heterogeneity-unaware Pollux, as Pollux gives them more GPU time to jobs on faster GPUs (out of randomness).

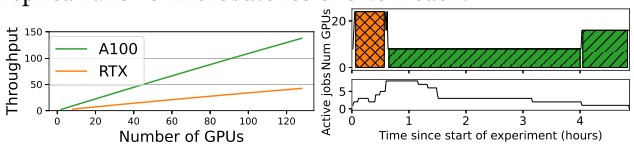
Workload Intensity. Figure 7 shows the average JCT as a function of average arrival rate for each of our evaluated schedulers. We sample jobs from **Helios** traces at various job arrival rates and evaluate them in the *Heterogeneous* setting with a fixed cluster of 64GPUs.

Figure 7. Avg. JCT for Sia, Pollux, and Gavel for various job arrival rates sampled using **Helios** traces.

Pollux and Sia outperform Gavel at larger arrival rates because they can scale down running jobs to use fewer GPUs rather than having to time-share GPUs. Sia consistently outperforms Pollux by 50–65% by aggressively matching jobs to preferred GPU types. As jobs arrival rates increase, jobs wait longer for resources, a problem that worsens with increasing congestion. However, an 8-hour job submission window is too short to observe these effects; on the 48-hour **newTrace** (Table 3), Gavel sees about 7x more contention compared to Sia (<2x on the 8-hour traces), adding evidence to our claim.

5.3 Adapting hybrid parallel jobs

We simulate training of a 2.8B GPT model that uses pipeline model parallelism to scale to a few GPUs, and data-parallelism with gradient all-reduce to scale to multiple nodes. We borrow statistical efficiency profiles from BERT (closest match) to simulate a DL job finetuning the GPT model, and profile compute times for micro-batches and all-reduce times for different placements on a100 and rtx GPUs to seed the simulator. Finally, we assume this job uses the commonly used Gpipe schedule [18] internally for PMP. We use 2 and 8 stages (1 per GPU) for a100 and rtx GPUs, respectively, to account for the larger memories on a100 GPUs. Each data-parallel replica runs 48 microbatches of size 1 each.



(Left) shows the hybrid-parallel GPT model's throughput scaling linearly with GPU count as computation dominates communication for this model. (Right) shows Sia adaptation decisions for this model in response to changing cluster conditions. As expected, Sia scales the GPT model in response to congestion: scaling it down around the 1hr mark and back up around the 4-hr mark. Sia is the first cluster scheduler to support elastically scaling hybrid-parallel jobs on heterogeneous resources; supporting additional adaptation dimensions for PMP jobs is left to future work.

5.4 Attribution of primary benefits

We show the importance of having the scheduler directly address each key aspect (resource heterogeneity and job adaptability) by evaluating scenarios where only one is present.

Job adaptability, but not resource heterogeneity. We compare Sia against Pollux[44], Shockwave[61], Themis[34], and Gavel[40] using the **Philly** traces in a *Homogeneous* setting. We use TunedJobs for Shockwave, Themis and Gavel and re-tuned the job hyper-parameters to fully exploit the 64-GPU cluster. Table 4 (and the left-most bars in Figure 1) shows average and 99th percentile (p99) job completion times, average job makespan and average number of GPU hours consumed to train a job.

Policy	JCT		Make-span	GPU hrs/job
	Avg.	p99		
Sia	**1.9h**	**18.1h**	**21.4h**	**8.4h**
Pollux[44]	2.0h	19.3h	21.7h	8.6h
Shockwave[61]+TJ	3.6h	32.8h	35.0h	12.5h
Themis[34]+TJ	5.4h	44.7h	49.7h	17.2h
Gavel[40]+TJ	4.3h	37.1h	44.3h	15.3h

Table 4. Comparison of Sia against state-of-the-art in the *Homogeneous* setting. TJ is short for TunedJobs.

Pollux was designed for this scenario, and Sia matches it on all metrics, even outperforming Pollux as its ILP formulation can guarantee a global optimum (Pollux's genetic algorithm does not). Sia had fewer restarts compared to Pollux – 2.6 vs 5.1 restarts per job, so Sia wasted fewer GPU hours on checkpoint-restore operations. Shockwave [61] is the best inelastic scheduler as its objective optimizes for job progress and finish-time-fairness while penalizing schedules that result in large makespan. Themis (optimizing FTF) and Gavel (optimizing cluster throughput) fall behind Shockwave on all metrics. Sia and Pollux both exploit adaptivity and show a 50-70% improvement over the state-of-the-art inelastic baselines in all metrics.

Resource heterogeneity, but not job adaptability. The right-most bars in Figure 1 show average JCTs for the three schedulers, but with every job being treated as *rigid* – it must be run with the batch size and GPU count specified in the trace. Said differently, auto-scaling and co-adaptive batch size tuning is disabled for Sia and Pollux, evening the playing field with Gavel that cannot exploit job adaptivity. Even though Gavel was designed for this scenario, Sia outperforms it by about 25%. This can be atttributed to the fact that Sia explicitly optimizes for goodput (aka a max-sum-*goodput* policy) while Gavel optimizes for cluster throughput (max-sum-*throughput* policy). So, with inelastic jobs, Sia will

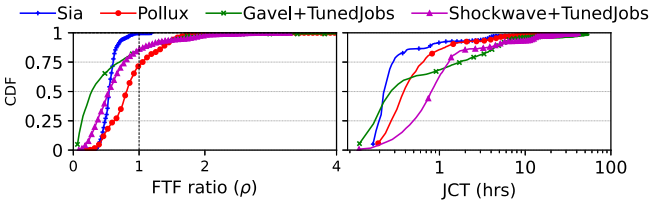


Figure 8. CDF of (left) Finish-Time Fairness ratio ρ[34], and (right) job completion times for Sia, Pollux, Gavel and Shockwave using **Helios** traces in the *heterogeneous* setting.

always provides higher per-GPU goodput, resulting in better performance over Gavel. Pollux also optimizes for sum of goodput, but produces worse JCTs as it is blind to and cannot exploit the GPU heterogeneity in the cluster.

5.5 Finish Time Fairness

Mahajan et al. [34] propose *finish-time fairness* (FTF [34]) as a metric that captures fairness of allocations to a job over its lifetime in a cluster. Assume job J sees an average contention (total number of jobs requesting resources) of N_{avg} and takes T_s to complete. Finish-time fairness (FTF) ratio ρ for the job J is defined as the ratio of a job's completion time in a shared cluster (T_s) to its JCT in an *isolated* and fair-sized cluster (T_f), where the isolated cluster contains $\frac{N_{gpus}}{N_{avg}}$, and N_{gpus} is the cluster size. We extend finish-time-fairness, defined originally for homogeneous clusters [34], to heterogeneous clusters as follows –

$$\rho = \sum_{G} P(G = g) \cdot \rho_g \tag{6}$$

where ρ_g is the FTF ratio for GPU type g.$P(G = g)$ is the probability that a random GPU in the cluster is of type g, given by $\frac{N_g}{N_{total}}$ where N_g is the number of GPUs of type g, and N_{total} is sum of N_g across GPU types. ρ_g is computed using the homogeneous-cluster definition [34] and only for the N_g GPUs of type g. If there exists only one GPU type, Equation (6) reduces to the homogeneous-cluster definition, preserving the metric's semantics. For heterogeneous clusters, ρ can be interpreted as the *expectation* of the FTF ratio taken over multiple GPU types.

$\rho > 1$ means unfair executions: job finishes faster in isolation than using scheduler's policy, while $\rho < 1$ means sharing resources can improve job runtimes using idle GPUs. A vertical CDF for ρ with all jobs having $\rho \leq 1$ means a perfectly finish-time-fair scheduler. We are interested in three metrics: (a) worst FTF ratio [34, 61] across all jobs, (b) *unfair* job fraction [61](fraction of jobs with $\rho > 1$), and (c) CDF(ρ).

Figure 8 (left) shows the CDF of finish-time-fairness ratios for Sia, Pollux, Gavel and Shockwave using the **Helios** traces in a heterogeneous-setting. From Figure 8, we see visually that Sia is more fair (more vertical and <1) than Gavel, Pollux or Shockwave. Indeed, Sia provides a worst FTF ratio of 1.2 and unfair fraction of <0.3%.

The worst FTF ratio for the other schedulers in Figure 8 are: Pollux=4.6, Gavel=27.8, Shockwave=3.3. Their unfair job fractions are 28%, 15% and 14%, respectively. Shockwave does better than Gavel and Pollux, since it penalizes jobs with high FTF ratios, trading worst FTF ratio for unfair job fraction when compared to other schedulers. Sia achieves by far the lowest unfair job fraction and worst FTF ratio.

Figure 8 shows job completion times for Gavel, Shockwave, and Sia. Gavel and Shockwave prioritize either makespan or long jobs, resulting in worse outcomes for short jobs during periods of congestion. Sia adapts to prioritizing minimizing average JCT or makespan based on congestion levels: scale down long jobs during congestion to prioritize incoming short jobs (reduces congestion) and scaling out long jobs during reduced congestion to minimize makespan.

5.6 Policy overhead and scalability

In the 64-GPU *Heterogeneous* setting with **Helios** traces, Sia's policy optimization has a median and 95th percentile runtime of 96ms and 426ms, respectively(insignificant overheads for 60s scheduling rounds). Pollux takes longer with median and p95 times at 2.2s and 4.8s, respectively, indicating it may not scale to larger cluster sizes. Gavel is significantly faster with a median and p95 policy runtime of 13ms and 28ms, respectively.

Figure 9 shows scheduling policy runtime as a function of cluster size. Experiments are conducted using the *Heterogeneous* setting running **Helios** traces, scaled up to 2048 GPUs (traces scaled accordingly). Sia scales well, with a single-second runtime for policy optimization, enabling management of large clusters with thousands of GPUs and many GPU types. Pollux's genetic algorithm runs significantly slower (100x slower than Sia's ILP formulation) and struggles to find optimal solutions for large clusters due to an explosion of search space complexity (even without considering the extra complexity induced by heterogeneity). Gavel is much quicker, because it does not consider job-adaptation.

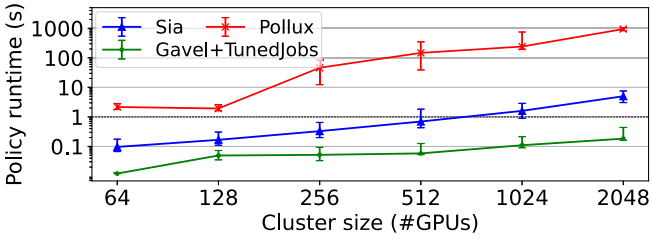


Figure 9. Median policy runtime for Sia, Pollux, and Gavel for various cluster sizes using proportionally-sized **Helios** traces. Error bars represent 25th and 75th percentiles.

5.7 Sia Parameter Sensitivity

Fairness parameter (p). Figure 10 shows scheduler metrics for Sia, as a function of p computed using the **Helios** traces.

The impact of p on 99th percentile JCT is very evident as Sia allocates more GPUs to jobs that can take advantage of both scale and newer GPU types better (particularly `BERT` and `ImageNet` training jobs). Since these jobs also tend to

run for a long duration, this drastically reduces their JCTs, bringing down the p99 JCTs the expense of average JCT. This also affects fairness of allocations and we notice that $p = 1.0$ has higher unfair job fraction (not shown here) compared $p = -0.5$. We choose $p = -0.5$ since it performs the best among all the p values we tested along the average JCT, average makespan and finish-time-fairness metrics.

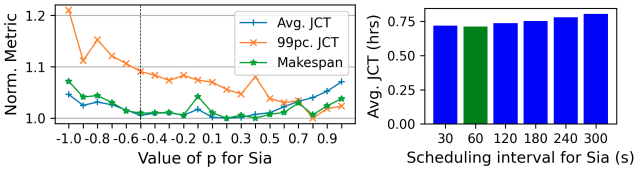


Figure 10. (Left) Trend for average and 99th percentile JCTs, and makespan for various values of p for Sia, and (Right) Average JCT for Sia for different scheduling round durations.

Scheduling round duration. We use a 60-second scheduling round duration for all our experiments. Increasing round duration from 60s to 300s increased average JCT for Sia by 333s (12%), while a shorter duration (30s) resulted in a higher rate of re-allocations and worsened average JCT. Sia's policy optimization takes less than 1 second, even for moderately sized clusters, and we choose a round duration of 60s since it performed the best. There was no significant change in p99 JCT or makespan observed.

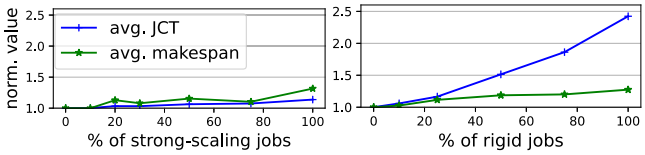
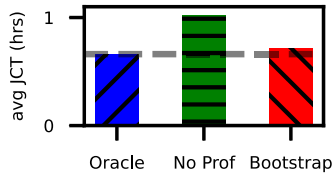


Figure 11. Average JCT and makespan for Sia on **Philly** traces as a function of the % of jobs that support (Left) only *strong-scaling* adaptivity, and (Right) no adaptivity (*Rigid*)

Fraction of jobs supporting adaptivity. Sia supports adapting batch size, GPU count, and GPU type. Figure 11 shows the average JCT and makespan for Sia, normalized to all AdaptiveJobs (0% constrainted), as we vary the percentage of jobs with restrictions on which dimensions are adapted. *Strong-scaling* adaptivity constrains a job to use a fixed (user-supplied) batch size, but allows Sia to optimize the number and type of GPUs allocated to this job. *Rigid* jobs constrain a job to use a fixed batch size *and* fixed GPU count, but allow Sia to optimize the GPU type allocated. From Figure 11, we can conclude the following: (1) optimizing number of GPUs in addition to the GPU type improves avg JCT by 56%, and (2) additionally optimizing batch size (with number of GPUs and GPU type) improves avg JCT by another 13%.

Profiling Overheads. Profiling jobs incurs overheads, and profiling every possible configuration is impractical. Too little profiling and Sia might produce sub-optimal schedules, while too much profiling can waste cluster resources for marginal improvement in cluster efficiencies. To understand this trade-off, we evaluate Sia on **Helios** traces in three settings: (a) *Oracle* is an ideal setting where Sia knows a job's throughput on any set of resources (a best-case scenario for Sia) (b) *No Prof* does not profile initially and adopts a *profile-as-you-go* approach, resulting in zero profiling overhead (but no initial info for Sia); and (c) *Bootstrap* uses min-GPU profiles and extrapolates throughput for yet-to-run configurations (see Section 3), requiring ≈0.1 GPU hrs of profiling for each job—a middle ground between the extremes. Note that *Oracle* only serves as a baseline to quantify the effectiveness of Sia's bootstrap approach; it is impractical in most clusters, as it would need to profile 100s-1000s of placements across GPU types (1-10 GPU hrs/job).



Sia with *Bootstrap* performs 30% better than *No Prof* and only 8% worse than *Oracle*, demonstrating the effectiveness of its bootstrapping mechanisms for heterogeneity-aware job adaptivity with minimal profiling overhead. We also found that profiling two GPU counts per GPU type performed worse that Sia's minimized approach. Sia's bootstrappping also scales well: for a cluster with 20 GPU types, bootstrapping adds <5% overhead to a job's execution.

6 Conclusion

Sia efficiently schedules adaptive DL jobs on heterogeneous resources, co-adapting each job's assignment of GPU count, GPU type, and batch size, resulting in increased DL cluster performance. Experiments show 30–93% reductions in average JCT, 28–95% reductions in p99 JCT and makespan, and 22–31% reductions in the unfair job fraction, when Sia is compared to existing schedulers. As such, Sia provides a critical component for emerging heterogeneous DL clusters.

7 Acknowledgements

We thank Amar Phanishayee, the anonymous SOSP reviewers and our shepherd, Douglas Terry, for their insightful feedback and suggestions in improving the paper. We thank the members and companies of the PDL Consortium (Amazon, Google, Hitachi, Honda, IBM, Intel, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support. This work is supported in part by NSF Award #2211882 and by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W911NF-20-D-0002. The content of the information does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

References

[1] Hu, Sia, and Heh. https://www.britannica.com/topic/Hu-Egyptian-religion, 2022 (accessed December 10, 2022).

[2] Sia. https://en.wikipedia.org/wiki/Sia, 2022 (accessed December 10, 2022).

[3] petuum/adaptdl. https://github.com/petuum/adaptdl/tree/osdi21-artifact, 2022 (accessed January 2022).

[4] AWS Tranium. https://aws.amazon.com/machine-learning/trainium/, 2023 (accessed April 2023).

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, USA, 2016. USENIX Association.

[6] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

[7] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[8] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, 2015.

[10] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. Association for Computing Machinery, 2016.

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

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, 2019. Association for Computational Linguistics.

[13] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[14] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

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

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[18] 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. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism.* 2019.

[19] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739, 2021.

[20] Petuum Inc. petuum/adaptdl: Resource-adaptive cluster scheduler for deep learning training., April 2021.

[21] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[22] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[24] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A user-friendly algorithm for distributed training. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4911–4920. PMLR, 13–18 Jul 2020.

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[26] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[28] John Kominek and Alan W Black. The cmu arctic speech databases. In *Fifth ISCA workshop on speech synthesis*, 2004.

[29] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, 2009.

[31] Gary Lauterbach. The path to successful wafer-scale integration: The cerebras story. *IEEE Micro*, 41(6):52–57, 2021.

[32] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: Compute allocation in hybrid clusters. EuroSys '20. Association for Computing Machinery, 2020.

[33] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

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

[35] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www.gnu. org/s/glpk/glpk. html*, 2022.

[36] Sam McCandlish, Jared Kaplan, and Dario Amodei. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[37] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[38] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3), 2022.

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19. Association for Computing Machinery, 2019.

[40] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[41] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 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*, SC '21. Association for Computing Machinery, 2021.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[43] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021.

[45] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[46] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016.

[47] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[48] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, 2018.

[49] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, 2018.

[50] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. Singularity: Planet-scale, preemptible, elastic scheduling of ai workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[51] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. Not all gpus are created equal: characterizing variability in large-scale, accelerator-rich systems. *arXiv preprint arXiv:2208.11035*, 2022.

[52] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[53] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In *Advances in Neural Information Processing Systems*, 2021.

[54] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[55] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

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

[57] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20. USENIX Association, 2020.

[58] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22. Association for Computing Machinery, 2022.

[59] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. In *Advances in Neural Information Processing Systems*, 2020.

[60] 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. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022.

[61] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. *arXiv preprint arXiv:2210.00093*, 2022.