FLUID: RESOURCE-AWARE HYPERPARAMETER TUNING ENGINE

Peifeng Yu [*1] **Jiachen Liu** [*1] **Mosharaf Chowdhury** [1]

ABSTRACT

Current hyperparameter tuning solutions lack complementary execution engines to efficiently leverage distributed computation, thus ignoring the possibility of intra- and inter-GPU sharing, which exhibits poor resource usage. In this paper, we present Fluid, a generalized hyperparameter tuning execution engine, that coordinates between hyperparameter tuning jobs and cluster resources. Fluid schedules evaluation trials in such jobs using a water-filling approach to make the best use of resources both at intra- and inter-GPU granularities to speed up the tuning process. By abstracting a hyperparameter tuning job as a sequence of TrialGroup, Fluid can boost the performance of diverse hyperparameter tuning solutions. Our experiments show that Fluid can speed up synchronous BOHB by 100%, and BOHB and ASHA by 30% while having similar final accuracy.

1 INTRODUCTION

Deep learning has become ubiquitous in recent years. The effectiveness of deep learning models, however, is highly sensitive to *hyperparameters* (Melis et al., 2017), which control the model architecture and/or training process, and have to be set before training. Naturally, hyperparameter tuning has taken a central stage in machine learning clusters. Because of the high-dimensionality of the search space, thousands of different hyperparameter settings often have to be evaluated before finding a final set for training in production. For instance, 2000 GPU days of reinforcement learning (Zoph et al., 2017) or 3150 GPU days of evolution (Real et al., 2018) are needed to obtain a state-of-the-art architecture for CIFAR-10 and ImageNet. In addition, a recent report suggested that over 86% of the jobs in a Microsoft GPU cluster with 5000 unique users perform hyperparameter tuning (Mahajan et al., 2019; Jeon et al., 2019).

Hyperparameter tuning is an optimization loop in order to find the best set of hyperparameters that are likely to produce the highest validation accuracy. A hyperparameter tuning job contains a large group of training trials, each with its own configuration. It gets feedback from running previous trials before selecting new ones to explore until reaching a target accuracy or stopped by the user. Distributed computation is commonly used to speed up the tuning process (Li et al., 2018; Hutter et al., 2011; Falkner et al., 2018).

Unfortunately, current hyperparameter tuning solutions cannot efficiently leverage distributed computation. Instead of planning a group of training trials as a whole, each training trial is often independently submitted to a cluster manager, most of the time as a single task running on a single worker GPU (Gu et al., 2019; Jeon et al., 2019). The cluster manager then launches those trials without considering the possibility of intra-worker and inter-worker sharing, failing to make good use of the available resources. For example, there can be more workers than training trials, which can lead to resource underutilization if each training trial only uses one worker. Moreover, a single training trial may not fully occupy one worker; the single-trial-to-single-worker mapping then leaves room to further improve resource utilization when there are more training trials than workers. Even the state-of-the-art cluster managers, which support users to submit a collection of jobs (Mahajan et al., 2019), focus on fair sharing of resources between multiple users but leave it up to the users to decide how to execute them.

In an attempt to better utilize cluster resources, some recent works have proposed fully asynchronous execution methods (Li et al., 2018; Falkner et al., 2018) that launch a new trial whenever there is an idle worker in their allocation of resources. However, this execution strategy tightly couples the concurrency of the tuning algorithm itself to the number of available workers. In addition to problems caused by the single-trial-to-single-worker mapping mentioned above, it fails to concentrate resources on promising hyperparameter configurations. Although all resources are used in this case, many configurations do not necessarily do useful work – i.e., their results are discarded rather than used to guide the generation of the final configuration.

In this paper, we observe that the root cause of the sub-optimal use of resources in current hyperparameter tuning

[*]Equal contribution [1]Department of Electronic Engineering and Computer Science, University of Michigan, Michigan, USA. Correspondence to: Peifeng Yu <peifeng@umich.edu>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

solutions is their indifference to how trials are executed. But it is also unrealistic to leave the burden to ML researchers expecting them to manually determine good execution strategies when tuning hyperparameters. We, therefore, take a different approach and propose to decouple the execution strategy from hyperparameter tuning algorithms into a separate execution engine. This has the following advantages: *a*) by separating the concern between tuning algorithms and execution engines, both components can evolve independently; *b*) resource usage can be optimized, which results in faster tuning speed for any tuning algorithms, benefiting a wider range of applications.

However, the *wide variety of algorithms*, the *dynamicity of hyperparameter tuning workloads* and the *differences in training trial profiles* make it non-trivial to design a generalized hyperparameter tuning execution engine that can make efficient use of resources and speed up evaluating a group of hyperparameter configurations.

To this end, we propose Fluid, an algorithm- and resource-aware hyperparameter tuning execution engine that coordinates between the cluster and hyperparameter tuning algorithms. By abstracting hyperparameter tuning job as a sequence of **TrialGroup**s, Fluid provides a generic high-level interface for hyperparameter tuning algorithms to express their execution requests for training trials. Fluid then automatically schedules the trials considering both the current workload and available resources to improve utilization and speed up the tuning process.

Fluid models the problem as a strip packing problem where each rectangle has different shapes and the goal is to minimize the height of the strip. The intuition behind Fluid is to grant more resources to more demanding/promising configurations, such as those with larger training budget, higher resource requirement, or higher priority. By combining techniques like elastic training and GPU multiplexing, Fluid is able to change the trial size including scaling out and in a trial across many GPUs and within one respectively. We also propose heuristics to solve the strip packing problem efficiently and prove that their performance is bounded within $2\times$ of the optimal.

To the best of our knowledge, we make the following contributions:

- Fluid is the first generalized hyperparameter tuning execution engine. It captures the characteristics of a hyperparameter tuning algorithm as a sequence of TrialGroups and models TrialGroup scheduling as a strip packing problem;
- Fluid proposes efficient heuristics with theoretical guarantees to solve the packing problem and it applies elastic training and GPU multiplexing to enforce the solutions; and

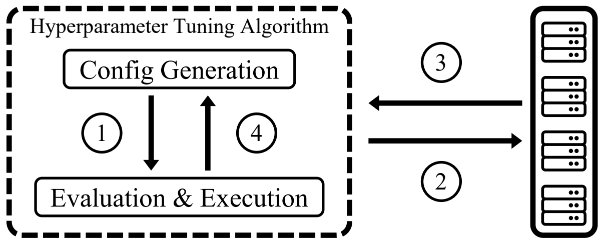


Figure 1. Hyperparameter tuning today: The tuning algorithm which implicitly contains execution logics interacts with the cluster directly.

- Fluid can boost the performance of various hyperparameter tuning algorithms with higher utilization and shorter end-to-end time. According to our experiment, Fluid can speed up synchronous BOHB by 100%, BOHB and ASHA by 30%, with similar final accuracy.

2 Motivation

2.1 Background and Related Work

2.1.1 Hyperparameter Tuning Algorithms

Figure 1 gives an overview of how hyperparameter tuning algorithms work in general. ① The hyperparameter configurations are generated and passed on for evaluation; ② the evaluation logic creates corresponding training trials and submits them directly to the cluster; ③ the training trial finishes execution and the tuning algorithm gets notified; ④ the results are passed back to the generation mechanism for future trial generations.

There are five primary trends in existing hyperparameter tuning algorithms tweaking parts of the aforementioned process to accelerate the evaluation and searching process for a good set of hyperparameter.

1. **Parallel search:** Parallel hyperparameter search approaches (Michie et al., 1994; Bergstra & Bengio, 2012) taking the idea of grid/random-based search introduce parallelism to speed up evaluation. However, because finding the best configurations in a large space requires guidance, random search-based methods cannot quickly converge to good configurations.
2. **Model-based search:** Model-based hyperparameter search approaches (Friedrichs & Igel, 2005; Shahriari et al., 2016; Bergstra et al., 2011) sequentially generate better hyperparameter configurations based on feedback from previous evaluation results. However, such adaptive selecting and evaluating process is inherently sequential and thus not suitable for the large-scale regime.
3. **Early-stopping:** Early-stopping evaluation strategies (Domhan et al., 2015) aim at detecting and stopping poor configurations earlier in order to avoid wasting resources on unpromising configurations. Successive

Method	Para.	Model	Early	Async.	Exec.
Grid/Rand.	✓				
SMBO		✓			
Hyperband	✓		✓		
BOHB	✓	✓	✓	✓	Asyn
ASHA	✓		✓	✓	Asyn
PBT	✓	✓			
HyperSched	✓		✓	✓	✓

Table 1. An overview of common hyperparameter tuning algorithms and how they employ the techniques mentioned in §2.1.1. The last column indicates if they have dedicated execution logic.

Halving (Karnin et al., 2013), for example, iteratively kills the poor trials and allocates more training time to the top fraction of trials. However, under this iterative process, only a few promising configurations end up being evaluated end-to-end, creating triangular shaped resource usage pattern over time, which degrades the resource efficiency in distributed environments.

4. **Asynchronous search:** Fully asynchronous algorithms (Li et al., 2018) always align the number of training trials with the number of workers, which aim at fully utilize all the resources to evaluate configurations. However, these algorithms fail to concentrate resources on promising configurations but spend them on exploring hyperparameter search space.
5. **Hybrid approach:** Hybrid approaches combine the ideas of the four trends aforementioned (Alvi et al., 2019; Li et al., 2016; Falkner et al., 2018; Li et al., 2018; Jaderberg et al., 2017).

2.1.2 *The Default Execution Logic*

As already discussed, current hyperparameter tuning methods focus on speeding up searching mostly in the algorithm, while hardly considering their interactions with the cluster.

In fact, as shown in Table 1, only a few algorithms have dedicated execution logic. Most others simply assign one pending job on one idle worker and maintain a FIFO queue. Some newer ones, such as ASHA and BOHB (Li et al., 2018; Falkner et al., 2018), use a simple fully asynchronous strategy to launch a new trial whenever there is an idle worker. Although HyperSched (Liaw et al., 2020) has its own execution logic on the top of ASHA that makes use of distributed training when deadline approaches, this logic is too specific to generalize to other tuning algorithms.

Overall, the execution logic is tightly coupled with the evaluation logic in tuning algorithms, and they lack any generalization to be efficiently applied to every algorithm.

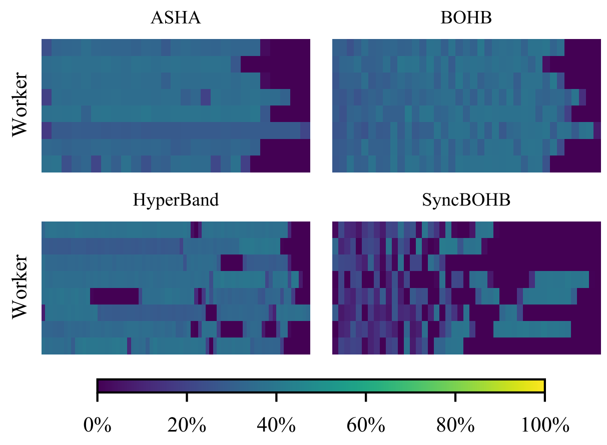


Figure 2. GPU utilization of 4 algorithms on the CIFAR-10 task, running to completion. Each algorithm has 8 workers and each data point is averaged over a 30 seconds window.

2.2 Motivation

Our work is motivated by the low resource utilization in existing hyperparameter tuning algorithms. Here we illustrate that the default execution strategy fails to efficiently use all available resources, while asynchronous execution strategies aim to increase utilization but not all resource usage contribute to selecting the final configuration.

2.2.1 *Resource (Under-)Utilization*

We consider four representative algorithms – ASHA, Hyperband, BOHB, and Synchronous BOHB – to understand their resource usage characteristics.

In Figure 2, we observe ample room for improvement in terms of resource usage. The underutilization has two root causes: *a*) No trial is running during the purple-shaded time slots in a particular worker. Distributed training can be used to balance workloads from other workers onto these free ones; *b*) Even when a GPU is used, the overall utilization is at most 60%, suggesting that a single trial often cannot fully saturate the GPU. Multiple trials can be stacked/packed together to reduce the average trial completion time.

2.2.2 *Case Study: Lack of Elasticity Reduces Utilization*

In the default setting of many early-stopping based algorithms like Successive Halving, the number of available workers is static while the number of trials is diminishing. Training trials are executed on the workers in a FIFO order.

In Table 2, we set up a basic Successive Halving based tuning session and measure the average resource utilization as well as tuning speed over varying number of workers. (See Section 6 for details about the CIFAR-10 task.) As we can see, when the number of workers increases, the tuning process is indeed faster but the resource utilization is lower.

# Workers	Util.	Runtime
2	81.20%	2356
4	63.00%	1432
8	45.80%	1073
16	47.00%	475
32	25.20%	432

Table 2. Resource utilization and runtime over different number of workers with Successive Halving

While more workers can help with the beginning stages of Successive Halving, latter stages only have a smaller number of trials; even though there are idle workers, the algorithm is unable to make use of them. Furthermore, increasing resource allocation further is of no use.

2.2.3 *Case Study: High Utilization ≠ Useful Work*

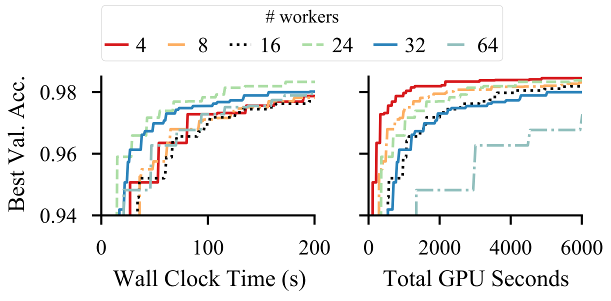
To increase utilization, a common fully asynchronous execution strategy is starting a new trial whenever there is an idle worker. As a result, the tuning algorithm can use all available workers, and its training concurrency is bounded by the number of workers.

Unfortunately, our analysis of ASHA, a popular asynchronous tuning algorithm, shows that not all work is useful work. In this experiment, we measure the best accuracy vs total GPU seconds and wall clock time over different training trial concurrency levels (Figure 3a and Figure 3b). We observe that as we increase the training trial concurrency, the best configuration is not necessarily identified faster; however, more GPU seconds are consumed to reach the same search target.

This is because the hyperparameter tuning is an exploration and exploitation process. Even though more workers are kept busy working on something fresh (exploration), not necessarily all the work done contribute to the final configuration's generation (exploitation), which makes it fail to concentrate resources on promising configurations. To this end, instead of blindly improve the resource utilization, hyperparameter tuning should spend its resource where it counts the most. An ideal evaluation and execution strategy should achieve the target accuracy with shorter wall clock time and smaller total GPU seconds.

3 Fluid

In this section, we start with the problem statement and review the challenges Fluid must solve to become a generalized execution engine for hyperparameter tuning. Then we introduce the high-level interface used by Fluid and how it interacts with other components in a hyperparameter tuning job. Finally, we illustrate the intra- and inter-GPU sharing considerations in Fluid.



(a) Wall clock time (b) GPU Seconds

Figure 3. ASHA best validation accuracy over wall clock time and total GPU seconds (averaged across 3 ASHA jobs respectively)

3.1 Problem Statement

Given a hyperparameter tuning job and available resources, the objective of Fluid is to carefully allocate resources to each training trial in the job such that resources are efficiently used and the makespan is minimized. In addition, Fluid should generalize to most hyperparameter tuning algorithms and react to cluster resource changes.

Fluid must address the following challenges.

1. **The wide variety of tuning algorithms** There are many strategies for hyperparameter tuning, each of which generates configurations and evaluates their performance in different ways (§2.1.1). It is challenging to design a general execution engine that can take algorithm-aware scheduling decisions and minimize the makespan of a collection of training trials (§3.2).
2. **Highly dynamic training workloads and resources** Training trials generated by hyperparameter tuning dynamically change over time due to the use of early-stopping strategy (§2.1.1). Cluster resource allocation can also be dynamic for fairness and efficiency reasons. Hence, it is challenging to capture the dynamic resource usage to reallocate correspondingly.
3. **Heterogeneity in training trial profiles** Differences in hyperparameters across training trials may cause different resource demands and be given different amounts of training budgets. Hyperparameter configurations may react differently for different resource allocations too. This challenges the execution engine to treat different trial profiles accurately (§3.4 and §4.2.2).

3.2 The TrialGroup Abstraction

To generalize different hyperparameter tuning algorithms, we introduce an abstraction called **TrialGroup** between the tuning algorithms and Fluid. Algorithms can use this simple-yet-rich interface to express their training requests, and Fluid works with this consistent model to schedule executions.

A TrialGroup is a group of training trials with a training

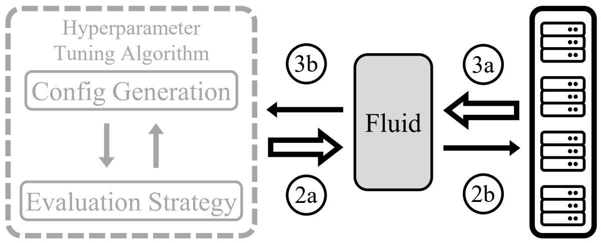


Figure 4. With Fluid, the tuning algorithm 2a submits **TrialGroup** according to its evaluation strategy and 3b gets feedbacks back anytime they are available. Fluid itself manages 2b the job execution and handles 3a resource changing events.

budget associated to each trial in the group. At any time, trials may be removed from the group due to completion or requested termination from the algorithm. The optimization goal for such a TrialGroup is to minimize its makespan, such that all the results are available as early as possible. TrialGroup is the basic unit of scheduling in Fluid.

Although the definition of TrialGroup is simple, we find it quite expressive for modeling hyperparameter tuning algorithms' training trial execution requirements. In the most simple case: grid/random search, where all training trials are created at the same time and have a fixed amount of budget, all trials fit nicely in one TrialGroup. For more involved algorithms discussed in Section 2.1.1, where new iterations of the algorithm may be added based on previous feedbacks, all trials from a single iteration forms a TrialGroup. This essentially creates a sequence of TrialGroups, each has its own makespan minimized individually.

3.3 Fluid Overview

Figure 4 illustrates Fluid's position in the stack. Fluid coordinates between the cluster and tuning algorithms, decoupling the execution logic from any single tuning algorithm. The tuning algorithm 2a submits TrialGroup to Fluid over time. During the tuning process, intermediate results are 3b reported back so that new trials may be created or existing ones removed.

The action of Fluid will only be triggered when new TrialGroups are added or some resources are freed. Based on the 3a real-time resource usage reported by the cluster, Fluid uses `StaticFluid` (S4.2) to schedule any new TrialGroups onto idle resources. It then reactively waits for more events to handle. When some resource are freed up and no pending job in the queue towards the end of the TrialGroup, Fluid adapts `DynamicFluid` (S4.3) to reallocate resources with the concern of any overhead.

3.4 Parallelism in Multiple Granularities

Minimizing the TrialGroup makespan ultimately translates to making better use of the underlying hardware resources in parallel. However, as shown in Section 2.2, relying on creating massive amount of trials as the single source of parallelism is neither enough to saturate individual workers nor applicable to many model-based tuning algorithms.

In Fluid, in addition to the number of trials, parallelism is sourced from within the training trial by using existing techniques like Multi Process Service (MPS) (NVIDIA, 2020a) (or Multi-Instance GPU (MIG) (NVIDIA, 2020b) more recently) and automatic distributed training. NVIDIA MPS allows multiple processes to run on a single GPU at the same time with different CUDA streams, providing *intra-GPU* parallelism. Distributed training is a established technique to use multiple parallel workers to reduce the training trial's job completion time, providing *inter-GPU* parallelism (Dean et al., 2012). In addition, resource elasticity (Or et al., 2020) realizes resource reallocation on the fly providing more flexibility in scheduling.

Fluid uses both *inter-GPU* and *intra-GPU* parallelism to perform resource allocation in a water-filling scheme. However, distributed training has communication overhead, while GPU sharing inevitably creates interference; both degrade performance. Therefore, we incorporate dynamic overhead measurement into Fluid's design (§4.2.2) and assess the *marginal benefit* before using both techniques to increase parallelism in intra- and inter-GPU granularities.

4 Fluid Algorithm

The scheduling problem of a hyperparameter tuning job, a sequence of TrialGroups, can be broken down into solving several independent TrialGroup scheduling problems. In this section, we begin with formulating this single TrialGroup scheduling as a strip packing problem (§4.1).

Since the hyperparameter tuning can be simplified as two actions (§3.3), trial arrival and departure, we then propose two heuristics to make full use of resources respectively under two conditions: *a*) new TrialGroup is launched to be scheduled; *b*) resource is freed to be allocated. We first introduce `StaticFluid` and how it uses GPU sharing to schedule incoming TrialGroup with the concern of sharing overheads (§4.2). We provide theoretical analysis for our `StaticFluid` in the appendix. Then, we introduce `DynamicFluid` which uses resource elasticity to reallocate freed resources while being robust to overheads (§4.3).

4.1 Problem Definition

Let the TrialGroup scheduling be represented as a strip packing problem $I = \{A, M\}$. Each rectangle a_i in $A =$

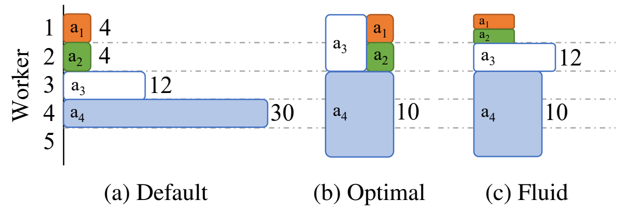


Figure 5. Toy example: default, optimal and Fluid for 4 training trials scheduled on 5 workers

$\{a_1, \cdots, a_k\}$ with width and height corresponds to a trial with allocated resources and remaining runtime. It is worth noting that in our problem setting, *each rectangle's width w_i determines its height $h_{i,w}$*. $h_{i,w}$ thus implies the relationship between different resource allocation and its corresponding runtime for this trial. Strips in $M = \{m_1, \cdots, m_n\}$ with identical width 1 and infinite height represents n available identical resources for current hyperparameter tuning jobs.

Fluid utilizes both intra- and inter-GPU sharing to achieve higher utilization of GPU resources and minimize the TrialGroup makespan. Hence, a training trial can be assigned to a real number amount of resources, where the fractional part of the resources represents a worker that will be shared with other trials, and an overall > 1 resources means the trial must be placed across workers using distributed training. The goal is to find a non-overlapping orthogonal packing of these k rectangles (taking into account different size options) into n strips such that the maximum height of strips is minimized.

4.2 StaticFluid

4.2.1 Algorithms

Since minimizing the height of strip packing is NP-hard (Hochbaum & Maass, 1985), Fluid proposes an efficient heuristic `StaticFluid` to find an approximate solution. At a high level, Fluid's resource allocation is performed using a water-filling scheme to balance the relationship between workloads and resources by "evenly" allocating resources to current evaluation trials in order to minimize the TrialGroup makespan.

Consider a grid search example with 5 GPUs and 4 trials $\{a_i\}_{i=1}^{i=4}$ arrived at the beginning with training budget 4s, 4s, 12s and 30s respectively. As shown in Figure 5a, the default FIFO scheduler treats each training trial independently without considering the different impact on the TrialGroup makespan; thus, it performs poorly because of stragglers and resource underutilization.

Intuitively, a good schedule would prioritize resources to long training trials to mitigate the straggler and minimize the makespan. As shown in Figure 5c, Fluid distributes the longest training trial a_4 onto three workers and packs shorter training trials a_1 and a_2 together on one worker. In this way, straggler is mitigated and resources are fully utilized. Fluid's static heuristic comes close to the optimal schedule for this example (Figure 5b).

Algorithm 1 StaticFluid

```
1: def StaticFluid(TrialGroup $A$, Idle Resources $M'$)
2:     Sort $a_i$ by $h_{i,1}$ in non-increasing order
3:     for all $a_i \in A$ do
4:         $w_i = \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$
5:         Allocate $a_i$ with $w_i$ resources
```

As shown in Algorithm 1, Fluid allocates resources w_i based on the ratio of each trial's runtime $h_{i,1}$ to the sum of runtime $\sum h_{j,1}$ in the TrialGroup. Fluid then schedules the trials in non-increasing order of resources onto the idle worker set. To avoid performance degradation from GPU sharing, we limit the maximum intra-GPU sharing to c and maximum inter-GPU training to d, which we discuss in more details in Section 4.2.2.

Formally, we give Theorem 1 without concerning any overhead, whose proof is included in Appendix A.3.

Theorem 1. *In the ideal situation, FluidStatic is a 2-approximation algorithm.*

4.2.2 Accounting for Overheads

Fluid adapts MPS and distributed training to realize intra-GPU sharing and inter-GPU training. These inevitably causes interference and overheads that hurt training performance. We account for such possibilities by tracking marginal benefits.

Intra-GPU Overhead Under the assumption that trials of the same hyperparameter tuning job are similar to each other, Fluid regards similar trends for diminishing marginal benefit for packing one more trial from the same TrialGroup. Fluid determines the optimal number of concurrent trials c on a GPU by finding out the inflection point where marginal benefit degrades below a threshold O_{th}.

We define the marginal benefit O^p for packing the p^{th} trial on a worker as

$$O^p = 1 - \frac{p-1}{p} \frac{\text{avg}(T^p)}{\text{avg}(T^{p-1})} \quad (p > 1)$$

where $T^p = \{t_i\}_{i=1}^{i=p}$ is the set of training time per iteration for p training trials. The marginal benefit of reduced average trial completion time is usually diminishing with the increasing packing overhead. Fluid ensures the packing benefit over packing overhead by limiting the number of concurrent training trials under the optimal number c where $c = \arg\max_p \quad O^p > O_{th}$.

Inter-GPU Overhead The speed-up brought by distributed training is not linear because of communication overheads. Such communication overheads are often determined by the size of model parameters and the number of workers (Shi & Chu, 2017).

Fluid determines the maximum degree of parallelism d for an evaluated configuration using the Paleo framework (Qi et al., 2017) to estimate the cost of training neural networks in parallel. It ensures the scaling benefit over scaling overhead by limiting the number of distributed worker to less than d.

Problem Setup To consider these realistic factors for our strip packing problem, we define the relationship between runtime $h_{i,w}$ and resources w for trial a_i as

$$h_{i,w} = \begin{cases} h_{i,1}\alpha_i^{\frac{1}{w}-1} & w \in (0,1) \\ \frac{h_{i,1}}{w}\beta_i^{w-1} & w \in [1,d] \end{cases}$$

where $h_{i,1}$ is the trial runtime on one worker. $\alpha_i \in [1, \frac{c}{c-1})$ is a measure of the packing overhead for trial a_i. $\beta_i \in [1, 1+\frac{1}{d})$ is a measure of the scaling overhead for trial a_i. This definition of $h_{i,w}$ ensures the following result.

Theorem 2. *In the real situation,* $\mathrm{StaticFluid}(I) < \max(2\,\mathrm{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\mathrm{OPT}(I)\beta^{\frac{1}{\beta-1}-1})$

We prove that `StaticFluid` has theoretical guarantee even when practical resource sharing overheads is considered (Appendix A.3).

4.3 DynamicFluid

In addition to scheduling incoming TrialGroup, Fluid also need to handle the dynamic changing resource usage caused by job departure and cluster resource changes.

As shown in Figure 5c, there is still a resource gap before the TrialGroup completes, which leaves space to further improve the utilization and minimize makespan. Therefore, we extend the heuristic `StaticFluid` to `DynamicFluid` that reallocates the incoming idle resources on the fly with the help of resource elasticity .

Ideally, `DynamicFluid` updates the resource allocation plan w' using resource elasticity with the same formula in Algorithm 1 when new resources are freed. However, using resource elasticity to adjust parallelism will inevitably incur overhead. Hence, Fluid considers scaling overhead ϵ to avoid performance degradation and frequent parallelism adjustment. As shown in Algorithm 2, Fluid updates resources $w_i \leftarrow w'_i$ for trial a_i at the end of current iteration only when it does not lead to performance degradation.

In addition to resource usage change caused by job departure, some cluster schedulers (Mahajan et al., 2019) may dynamically change resource allocation for fairness or efficiency. Fluid can handle such change by triggering `DynamicFluid` when resources are updated.

Algorithm 2 DynamicFluid

```
1: def DYNAMICFLUID(TrialGroup $A$, Total Res. $M$)
2:     Sort $a_i$ by $h_{i,1}$ in non-increasing order
3:     for all $a_i \in A$ do
4:         $w'_i = \min(\max(\lfloor \frac{h_{i,1}}{\sum_j h_{j,1}} n \rfloor, \frac{1}{c}), d)$
5:         if $w'_i > w_i$ and $h_{i,w'_i} + \epsilon < h_{i,w_i}$
6:             Update $a_i$ with $w_i$ resources        ▷ Scale up
7:         else if $w'_i < w_i$ and $w'_i(h_{i,w'_i} + \epsilon) < w_i h_{i,w_i}$
8:             Update $a_i$ with $w_i$ resources  ▷ Scale down
```

5 FLUID IMPLEMENTATION

We implemented Fluid as an executor for Ray Tune (Liaw et al., 2018). In addition to the implementing our execution algorithm, tuning algorithms in Tune were adapted to make use of the TrialGroup interface, which translates to one extra function call per algorithm class when applicable, to signify the creation of new TrialGroups.

In order to adjust training trials' number of worker at runtime with lower overhead, we also implemented training elasticity technique similar to the one proposed by recent work (Or et al., 2020).

One important input of the Fluid algorithm is the packing overhead measurement $\alpha_i \in [1, \frac{c}{c-1})$ and distributed overhead measurement $\beta_i \in [1, 1+\frac{1}{d})$. These numbers depends on many factors of the training process and are affected by the hardware in use. It is an active field of research to predict them given a particular configuration. In Fluid, instead of predicting, we use a trial-and-error approach by measuring these numbers on real hardware. Leveraging the iterative nature of deep learning training process, only a few iterations is enough to have reasonable measurements. Fluid thus uses a small fraction of resources to profile current trials and reuse the result throughout the whole tuning session.

6 EVALUATION

We evaluate Fluid with a range of hyperparameter optimization algorithms, including Grid/Random Search, PBT, Successive Halving, Hyperband, BOHB and ASHA (Michie et al., 1994; Bergstra & Bengio, 2012; Jaderberg et al., 2017; Karnin et al., 2013; Li et al., 2016; Falkner et al., 2018; Li et al., 2018). Our evaluation shows the following key highlights:

- Fluid can speed up diverse hyperparameter tuning workloads around 10%–70%, while improve cluster efficiency up to 10%–100%.

Task	Base Model	# of Arch. Params.	# of Training Param.	Target
CIFAR-10	AlexNet	3	4	Acc. >= 90%
WLM	RNN	4	6	PPL <= 140
DCGAN	CNN	0	2	Inception >= 5.2

Table 3. List of workloads

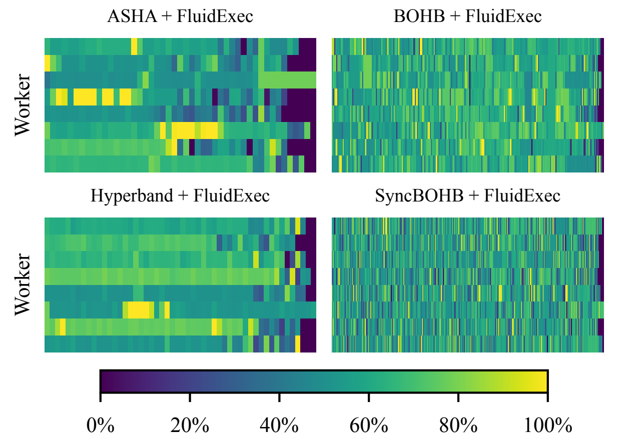


Figure 6. GPU utilization of 4 algorithms on the CIFAR-10 task using Fluid, running to completion. Each algorithm has 8 workers and each data point is averaged over a 30 seconds window.

- Fluid can improve the trade-off between resource efficiency and hyperparameter searching speed, and optimize both simultaneously.
- Fluid's benefits are robust under different kinds of environment setup and training workloads.

6.1 Experiment Setup

Testbed We built our testbed on Chameleon Cloud (Keahey et al., 2020). Each node has an Intel Xeon Gold 6126 CPU with NVIDIA Quadro RTX 6000 GPU. The interconnection is 10G ethernet.

Workloads We create our set of workloads (Table 3) using different deep learning tasks including computer vision, natural language processing and adversarial learning. The CIFAR-10 task includes the tuning of an variation of AlexNet on an image classification task maximizing accuracy; The WLM task tunes the training of a multi-layer RNN on the word-level language modeling task minimizing perplexity; The DCGAN task tunes two multiple-layer CNN network, creating a generative model on the MNIST dataset. The tuning target is maximizing the inception score.

In all tasks, the tuned hyperparameters include training parameters like learning rate, batch size, etc., as well as architectural parameters like number of CNN layers, size of layers, type of models, etc..

Tuning algorithms We compare the performance of 5 hyperparameter tuning algorithms with and without Fluid: PBT, Hyperband, ASHA, synchronous BOHB (SyncBOHB) and BOHB. These algorithms include both synchronous stage-based strategies and asynchronous strategies, covering most of the situation that may appear during hyperparameter tuning process, including stopping, promotion and so on.

Metrics The improvement in hyperparameter tuning job makespan is our key metric. The makespan is defined as the end-to-end time needed for a given problem to reach a certain metric target. For example, the time needed to reach 90% accuracy for the CIFAR-10 task and the time needed to reach 140 ppl for language models. We also measure resource utilization improvement to indicate Fluid's effort on improving resource usage.

6.2 Macrobenchmarks

We first report the performance improvement of Fluid over 5 tuning algorithms on all three tasks in Figure 7. The average GPU utilization of those runs are reported in Figure 8. In addition, we report the utilization heatmap similarly as in §2.2 on the CIFAR-10 task in Figure 6.

The benefit of Fluid varies over algorithms. We see a pattern that synchronous stage-based strategies (SyncBOHB, Hyperband) can gain more benefit 30%–100% on job makespan and resource usage with the help of Fluid due to underutilization of resources, while asynchronous strategies (ASHA, BOHB), in spite of their original high resource utilizations, see 10%–30% improvements. PBT always has a constant number of trials running, so the benefit of Fluid is limited. But Fluid can still help PBT to scale out to more concurrent trials, which improves the overall time to reach target.

Fluid's benefit comes from the following: *a*) Resource under--utilization due to mismatch between training trials and available resources over time. *b*) Stragglers due to algorithm's synchronous nature. *c*) Insufficient concurrency when the number of running trials is tied to the number of workers.

The rest of this section gives detailed insights into the improvement of synchronous stage-based tuning algorithms and fully asynchronous algorithms. We then move on to microbenchmarks in which we break down the improvement of individual techniques.

6.2.1 *Benefit on Synchronous Stage-based Strategy*

In Figure 9, we measure the best accuracy over time for synchronous stage-based strategy BOHB using different number of workers. Our results show that the tuning speed up is not in proportion to the increase of available workers. However, with the help of Fluid, we are able to achieve

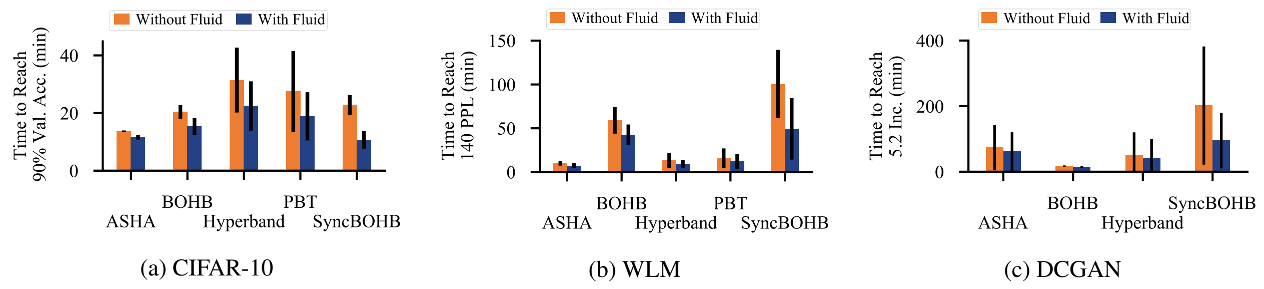


Figure 7. Time to reach target. Data averaged over 5 runs. Errorbar represents standard deviation. PBT on DCGAN did not finish in reasonable time and thus excluded from the report.

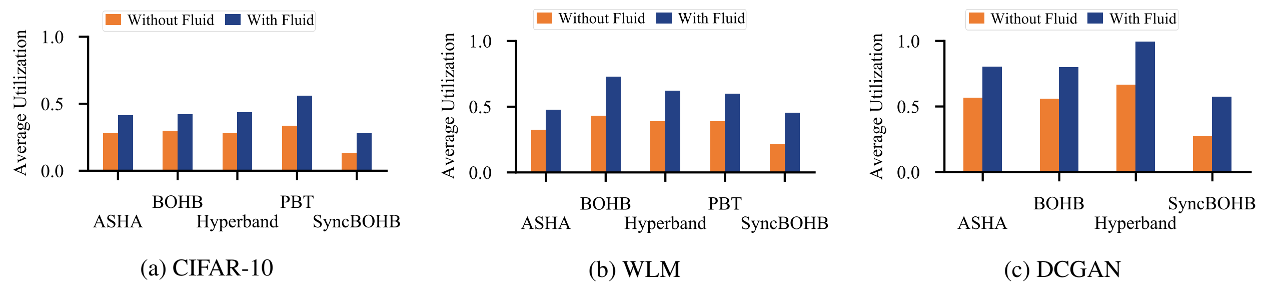


Figure 8. Average utilization

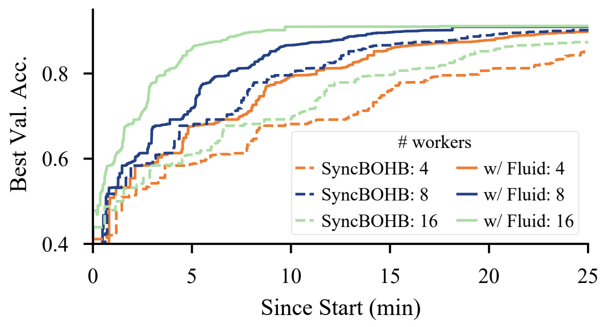


Figure 9. Validation accuracy over time for SyncBOHB w/ and w/o Fluid on the CIFAR-10 task. Data averaged over 5 runs.

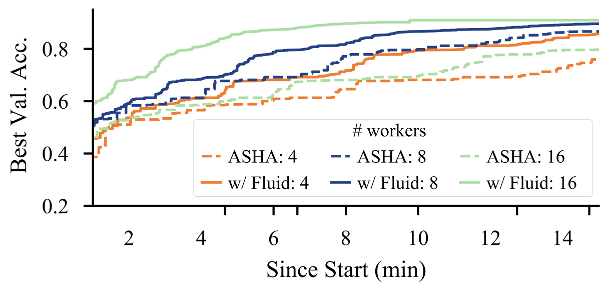


Figure 10. Validation accuracy over time for ASHA w/ and w/o Fluid on the CIFAR-10 task. Data averaged over 5 runs. Fluid is set to have 8 concurrent trials regardless of # workers.

better scalibility. The experiment was done on the CIFAR-10 task using SyncBOHB on 4, 8, 16 GPUs respectively.

6.2.2 Benefit on Fully Asynchronous Strategy

In Figure 10, we measure the max accuracy over time for fully asynchronous strategy ASHA using 4, 8, 16 GPU workers respectively. Our results show that the performance of fully asynchronous strategy largely depends on the number of workers. And in this particular case, 8 workers works the best. With the help of Fluid, we can easily set the concurrency for ASHA regardless of the number of physical workers, because Fluid is able to adjust the number of concurrent running trials. ASHA can therefore achieve the optimal balance. As a result, with Fluid, we are able to achieve similar performance as 8 GPUs using only 4 or better performance with more GPUs.

6.3 Microbenchmarks

6.3.1 Benefit of Intra-GPU Sharing

In Figure 11a, we compared the performance of hyperparameter optimization algorithms with and without MPS packing. We show the speed up of Fluid with MPS as the only enabled mechanism, compared to the original algorithm. For the sake of discussion, we choose Grid Search in this experiment to avoid influences from other factors.

Our results show that MPS Packing provides a significant performance increase especially on tuning relatively large TrialGroup with small model size. We experiment with Grid Search on three training workloads from small to large, and for each training workloads, we evaluate 9, 27, 81 configurations on 8 GPUs.

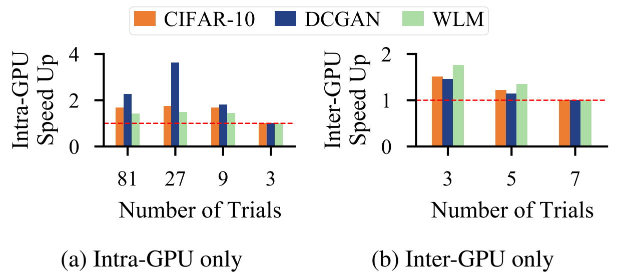


Figure 11. Speed ups break down of intra- and inter-GPU trainig

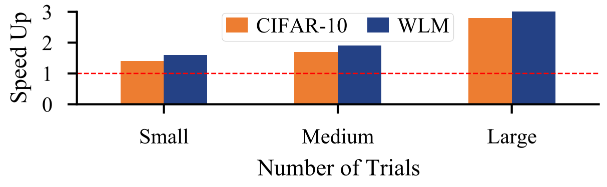


Figure 12. Relative speed up given trial runtime variance

6.3.2 *Benefit of Inter-GPU Training*

Similarly, we enable only inter-GPU mechanism in Fluid, and compare the performance gain using the Grid Search. With 8 GPU workers, we limit the trial number to smaller than that to explicitly trigger the inter-GPU distributed training. Our results show that inter-GPU distributed training provides a significant performance increase especially on tuning relatively small TrialGroup with large model size. The results are shown in Figure 11b. Fluid effectively utilizes the idle distributed resources and achieve tuning speed up especially when the gap between number of trials and number of workers is large.

6.4 Sensitivity Analysis

6.4.1 *Effect of Runtime Variance*

In Figure 12, we show how Fluid performs with different trial runtime variance. Our results show that Fluid can achieve better speed up or resource utilization improvement especially when trials' runtime variance is large, which can be attribute to `DynamicFluid` which adjusts resources at runtime. We experiment with Grid Search on tuning different training workloads, and for each training workloads, we evaluate 9 configurations with different degree of job runtime variance on 8 GPUs .

6.4.2 *Effect of Packing Overhead*

By manually modify tasks to include controllable artificial packing overhead, we are able to assess Fluid's reaction under different packing conditions.

The results reported in Figure 13a shows the speed up ratio of completion time of GridSearch with Fluid, with $1.5, 2, 10$ times packing overhead, relative to those without using Fluid. The experiment is given 27 hyperparameter configurations and has 8 workers in total.

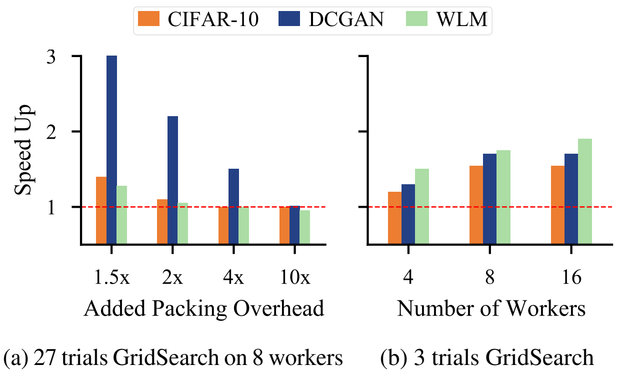


Figure 13. The speed up of Fluid with GridSearch with varying packing and scaling overhead.

When the overhead is relatively small, Fluid still sees positive marginal benefit to packing. With $10x$ overhead, the intra-GPU packing is effectively disabled and Fluid's performance becomes the same as the original algorithm.

6.4.3 *Effect of Scaling Overhead*

In Figure 13b, we show how Fluid reacts across various scaling overhead. Our results show that Fluid can achieve different degrees of speed-up under different scaling overhead by detecting model's scalability. We experiment with Grid Search on scaling different training workloads, and for each training workloads, we evaluate 3 configurations on 4, 8, 16 GPUs. In this experiment we disable the intra-GPU packing mechanism. Going from 4 GPUs to 8 gains sizable performance benefit across all 3 tasks. But CIFAR-10 and DCGAN does not benefit from adding more GPUs. In fact, the added GPUs are not used at all, because Fluid detects there will be high scaling overhead associated if these workloads scaling beyond enough. In real settings, those idle GPUs will be used by other trials.

7 Conclusion

Fluid is a generic hyperparameter tuning execution engine that decouples execution logic from tuning algorithms, with the high-level TrialGroup interface for tuning algorithms to express their execution needs. Fluid can boost the performance of diverse hyperparameter tuning solutions.

Acknowledgments

We thank the Chameleon team for providing GPU servers, which made many experiments on Fluid possible. We also thank all the SymbioticLab members for their helpful comments and feedback. This research was supported in part by NSF grants CNS-1845853, CNS-1900665, CNS-1909067 and a gift from the KLA Corporation.

References

Alvi, A., Ru, B., Calliess, J.-P., Roberts, S., and Osborne, M. A. Asynchronous batch Bayesian optimisation with improved local penalisation. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 253–262, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/alvi19a.html.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. URL http://jmlr.org/papers/v13/bergstra12a.html.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pp. 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc. ISBN 9781618395993.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., and Ng, A. Y. Large scale distributed deep networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1223–1231. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

Domhan, T., Springenberg, J. T., and Hutter, F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pp. 3460–3468. AAAI Press, 2015. ISBN 9781577357384.

Falkner, S., Klein, A., and Hutter, F. BOHB: robust and efficient hyperparameter optimization at scale. *CoRR*, abs/1807.01774, 2018. URL http://arxiv.org/abs/1807.01774.

Friedrichs, F. and Igel, C. Evolutionary tuning of multiple svm parameters. *Neurocomputing*, 64:107 – 117, 2005. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2004.11.022. URL http://www.sciencedirect.com/science/article/pii/S0925231204005223. Trends in Neurocomputing: 12th European Symposium on Artificial Neural Networks 2004.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 485–500, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/gu.

Hochbaum, D. S. and Maass, W. Approximation schemes for covering and packing problems in image processing and vlsi. *J. ACM*, 32(1):130–136, January 1985. ISSN 0004-5411. doi: 10.1145/2455.214106. URL https://doi.org/10.1145/2455.214106.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pp. 507–523. Springer, 2011.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017. URL http://arxiv.org/abs/1711.09846.

Jeon, M., Venkataraman, S., Phanishayee, A., Qian, J., Xiao, W., and Yang, F. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. *CoRR*, abs/1901.05758, 2019. URL http://arxiv.org/abs/1901.05758.

Karnin, Z., Koren, T., and Somekh, O. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pp. 1238–1246. PMLR, 2013.

Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H. S., Hammock, C., Mambretti, J., Barnes, A., Halbach, F., Rocha, A., and Stubbs, J. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

Li, L., Jamieson, K. G., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016. URL http://arxiv.org/abs/1603.06560.

Li, L., Jamieson, K. G., Rostamizadeh, A., Gonina, E., Hardt, M., Recht, B., and Talwalkar, A. Massively parallel hyperparameter tuning. *CoRR*, abs/1810.05934, 2018. URL http://arxiv.org/abs/1810.05934.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

Liaw, R., Bhardwaj, R., Dunlap, L., Zou, Y., Gonzalez, J., Stoica, I., and Tumanov, A. Hypersched: Dynamic resource reallocation for model development on a deadline, 2020.

Mahajan, K., Singhvi, A., Balasubramanian, A., Batra, V., Chavali, S. T., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. *CoRR*, abs/1907.01484, 2019. URL http://arxiv.org/abs/1907.01484.

Melis, G., Dyer, C., and Blunsom, P. On the state of the art of evaluation in neural language models. *CoRR*, abs/1707.05589, 2017. URL http://arxiv.org/abs/1707.05589.

Michie, D., Spiegelhalter, D., and Taylor, C. *Machine Learning, Neural and Statistical Classification*. Artificial intelligence. Ellis Horwood, 1994. ISBN 9780131063600. URL https://books.google.com/books?id=GsyPswEACAAJ.

NVIDIA. CUDA Multi-Process Service. https://web.archive.org/web/20200228183056/https://docs.nvidia.com/deploy/mps/index.html, 2020a. Accessed: 2020-02-28.

NVIDIA. NVIDIA Multi-Instance GPU. https://web.archive.org/web/20201004004526/https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2020b. Accessed: 2020-10-04.

Or, A., Zhang, H., and Freedman, M. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2, 2020.

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018. URL http://arxiv.org/abs/1802.01548.

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., and de Freitas, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104 (1):148–175, 2016.

Shi, S. and Chu, X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. *CoRR*, abs/1711.05979, 2017. URL http://arxiv.org/abs/1711.05979.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. URL http://arxiv.org/abs/1707.07012.

A Analysis of Algorithms

A.1 Problem Formulation

Let the TrialGroup scheduling be represented as a strip packing problem $I = \{A, M\}$. Each rectangle a_i in $A = \{a_1, \cdots, a_k\}$ with width and height corresponds to a trial with allocated resources and remaining runtime. It is worth noting that in our problem setting, *each rectangle's width w_i determines its height $h_{i,w}$*. $h_{i,w}$ thus implies the relationship between different resource allocation and its corresponding runtime for this trial. Strips in $M = \{m_1, \cdots, m_n\}$ with identical width 1 and infinite height represents n available identical resources for current hyperparameter tuning jobs.

A.2 StaticFluid Algorithm

As shown in Algorithm 1, Fluid allocates resources w_i based on each trial's runtime $h_{i,1}$ ratio among trials in the TrialGroup. Fluid then schedules the trials in non-increasing order of resources onto the idle worker set.

$$w_i^* = \frac{h_{i,1}}{\sum_j h_{j,1}} n \tag{1}$$

$$w_i = \begin{cases} \lfloor w_i^* \rfloor & w_i^* \geq 1 \\ \frac{1}{c} & w_i^* < 1 \end{cases} \tag{2}$$

A.3 Theoretical Results

Theorem 1 *In the ideal situation, FluidStatic is a 2-approximation algorithm.*

Proof of Theorem 1.

Given an instance $I = \{A, M\}$ of the strip packing problem and let the optimal solution be $\mathrm{OPT}(I)$. In the ideal case, no overheads will occur, which means the job size (rectangular area) remains the same for one job.

We break down this proof into proofs of two disjoint sub-problems. First we show how this problem can be divided into two sub-problems and then we prove the approximation factor for each sub-problem separately.

After calculating the resources w_i for each trial a_i by Equation 2, we divide trials into a small set and a large set: one with resources $w_x < 1$ and the other with resources $w_y \geq 1$. The sub-problems are defined as scheduling small set of trials a_x on $n - \sum w_y$ denoted as I_s and scheduling large set of trials a_y on $\sum w_y$ denoted as I_l. We simplify $\mathrm{StaticFluid}(\cdot)$ as $\mathrm{Fluid}(\cdot)$ for convenience in the following proof.

Lemma 1. *By applying our heuristic method, the result makespan of original problem is no worse than the maximum of the result makespan of two sub-problems:*

$$\mathrm{Fluid}(I) \leq \max\{\mathrm{Fluid}_{small}(I_s), \mathrm{Fluid}_{large}(I_l)\}$$

Proof of Lemma 1.

Since the trials in large trial set are ensured with w_y resources, all of them can be scheduled at the beginning. However, small trial set only has $n - \sum w_y$ resources in total, where $n - \sum w_y \leq n - \frac{\sum t_y}{\sum t} n = \frac{\sum t_x}{\sum t} n \leq \sum w_x^* < X$. (X is the size of small trial set). Thus, trials in small trial set may be scheduled when any resource becomes idle. Such idle resources can belong to either small set or large set. If the queued small trial is scheduled on resources belong to the large set, the makespan would be shorter than waiting on small set resources becoming idle. As a result, proving two sub-problems separately is sufficient to bound our original heuristic method.

Lemma 2. *In the ideal situation,* $\mathrm{Fluid}_{small}(I_s) < 2H_{small} + \max(h_1)$

Proof of Lemma 2.

For **small** trial set a_x with $n - \sum w_y$ resources, this sub-problem can be completely modeled as strip packing. According to our heuristic method, each small trial will be allocated with $\frac{1}{c}$ and longest trials are prioritized to be scheduled onto most idle worker. We simplify our method to next-fit decreasing height shelf-based algorithm, which means we schedule trials in non-increasing runtime to an available shelf and add a new shelf if previous shelves are full. Let's denote A_j as the area of j_{th} shelf, $A = \sum A_j$as the sum of shelves' area and H_j as the height of j_{th} shelf. Since the area of rectangle is decreasing with the increase of packing trials if the number of packing trials doesn't exceed the optimal number c, we have $A \leq \sum h_{x,1} < H_{small}(n - \sum w_y)$, where H_{small} is defined as the largest average height of small trial set. We have

$$A_j + A_{j+1} > H_{j+1} \times 1 + H_{j+1}\frac{1}{c} > H_{j+1}$$

$$\begin{aligned} \sum H_j &\leq H_1 + 2A = \max(h_{\frac{1}{c}}) + 2A \\ &\leq \max(h_{\frac{1}{c}}) + 2(n - \sum w_y)H_{small} \end{aligned} \tag{3}$$

Also, by taking one shelf as a whole trial with one unit resources, this sub-problem can be reduced to shelf-based job scheduling, which is to assign shelves to machines at particular times in order to minimize the makespan. And our heuristic becomes scheduling the shelf in non-increasing runtime order onto the strip with smallest height. Denote m_k as the strip with largest height and H_l as the height of the last shelf assigned to m_k. If m_k only has one trial, then this shelf has longest runtime which means it must be the optimal solution. If m_k have more than one shelf, we have

$$\begin{aligned}\text{Fluid}_{small}(I_s) = h(m_k) &\leq \frac{1}{n - \sum w_y}(\sum H_j - H_l) + H_l \\ &= \frac{1}{n - \sum w_y}\sum H_j + (1 - \frac{1}{n - \sum w_y})H_l \\ &\leq \frac{2H_{small}(n - \sum w_y) + \max(h_{\frac{1}{c}})}{n - \sum w_y} \\ &+ (1 - \frac{1}{n - \sum w_y})\max(h_{\frac{1}{c}}) \\ &= 2H_{small} + \max(h_{\frac{1}{c}}) \\ &< 2H_{small} + \max(h_1)\end{aligned} \tag{4}$$

Lemma 3. *In the ideal situation,* $\text{Fluid}_{large}(I_l) < 2H$

Proof of Lemma 3.

For **large** trial set a_y with $\sum w_y$ resources, this sub-problem can be completely modeled as strip packing. Based on our heuristic method, each trial a_y will be scheduled on w_y machines.

$$\begin{aligned}h_{j,w} &= \frac{h_{j,1}}{w} \\ &= \frac{\sum t_{y,1}}{n}\frac{w^*}{w} \\ &= \frac{w^*}{\lfloor w^* \rfloor}\underbrace{\frac{\sum h_{i,1}}{n}}_{H} < 2H\end{aligned} \tag{5}$$

Since $\sum w_y < n$, every trial can get its own resources at the beginning, which result in one-level packing.

$$\text{Fluid}_{large}(I_l) = \max(h_{j,w}) < 2H \tag{6}$$

Combine the result of two sub-problems and Lemma 1, we have

$$\begin{aligned}\text{Fluid}(I) &\leq \max(Fluid_{small}(I_s), \text{Fluid}_{large}(I_l)) \\ &< \max(2H_{small} + \max(h_1), 2H) \\ &\leq \max(2\,\text{OPT}(I) + \max(h_1), 2\,\text{OPT}(I)) \\ &= 2\,\text{OPT}(I)\end{aligned} \tag{7}$$

Theorem 2 *In the real situation,* $\text{StaticFluid}(I) < \max(2\,\text{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\text{OPT}(I)\beta^{\frac{1}{\beta-1}-1})$

Proof of Theorem 2. In the real situation, we consider the impact of GPU sharing overheads on the problem set up: *a*) for rectangular with fractional width (trial using intra-GPU sharing), the relationship between height and width is $h_i(w) = h_{i,1}\alpha_i^{\frac{1}{w}-1}, \quad w \in (0, 1)$; *b*) for rectangular with initegral width (trial using inter-GPU training), the relationship between height and width is $h_i(w) = \frac{h_{i,1}}{w}\beta_i^{w-1}, \quad w \in [1, d]$.

Lemma 4. $\alpha \in \left[1, \frac{c}{c-1}\right)$

Proof of Lemma 4.

Since Fluid ensures the performance of intra-GPU sharing increases by packing with more trials by limiting the number of packing trials under the maximum packing number c, we have $\frac{h_{\frac{1}{a}}}{a} < \frac{h_{\frac{1}{b}}}{b} \leq h_1$ if $1 \leq b < a \leq c$, where a and b are the number of packing trails.

$$\frac{a}{b} > \frac{h_{\frac{1}{a}}}{h_{\frac{1}{b}}} = \alpha^{a-b}$$

$$1 \leq \alpha < \frac{c}{c-1}$$

In addtion, we have $c < \frac{\alpha}{\alpha-1}$

Lemma 5. *In the real situation,* $\text{Fluid}_{small}(I_s) < 2H_{small} + \max(h_1)\alpha^{\frac{1}{\alpha-1}}$

Proof of Lemma 5.

Combine the result of Equation 4 and Lemma 4, we have

$$\begin{aligned}\text{Fluid}_{small}(I_s) &= 2H_{small} + \max(h_{\frac{1}{c}}) \\ &< 2H_{small} + \max(h_1)\alpha^{\frac{1}{\alpha-1}}\end{aligned} \tag{8}$$

Lemma 6. $\beta \in \left[1, \frac{d+1}{d}\right)$

Proof of Lemma 6.

Since Fluid ensures the performance of inter-GPU sharing increases with distributing on more workers by limiting $w < d$, we have $h_1 \leq h_{\frac{1}{a}} < h_{\frac{1}{b}}$ if $1 \leq b < a \leq d$.

$$\frac{h_1\beta^{a-1}}{a} < \frac{h_1\beta^{b-1}}{b}$$

$$1 \leq \beta < \frac{a}{b} \leq \frac{d+1}{d}$$

In addtion, we have $w < \frac{1}{\beta-1}$

Lemma 7. *In the real situation,* $\text{Fluid}_{large}(I_l) < 2H\beta^{\frac{1}{\beta-1}-1}$

Proof of Lemma 7.

Similar to Equation 5, we consider extra term β on the height of rectangular:

$$\begin{aligned} h_{j,w} &= \frac{h_{j,1}\beta_j^{w-1}}{w} \\ &= \frac{\sum t_{y,1}}{n}\frac{w^*}{w}\beta_j^{w-1} \\ &= \frac{w^*}{\lfloor w^* \rfloor}\beta_j^{w-1}\underbrace{\frac{\sum h_{i,1}}{n}}_{H} \\ &< 2H\beta_j^{w-1} \end{aligned} \tag{9}$$

Since $\sum w_y < n$, every trial can get its own resources at the beginning, which result in one-level packing.

$$\begin{aligned} \mathrm{Fluid}_{large}(I_l) = \max(h_{j,w}) &< 2H\max(\beta_j^{w-1}) \\ &< 2H\beta^{\frac{1}{\beta-1}-1} \end{aligned} \tag{10}$$

Combine the result of two sub-problems and Lemma 1, we have

$$\begin{aligned} \mathrm{Fluid}(I) &\leq \max(\mathrm{Fluid}_{small}(I_s), \mathrm{Fluid}_{large}(I_l)) \\ &< max(2H_{small} + \max(h_{\frac{1}{c}}), 2H\beta^{\frac{1}{\beta-1}-1}) \end{aligned} \tag{11}$$

Based on Eq.2, we can derive the relationship between the average height of the large trial set H_{large} and the average height of the whole trial set H.

$$\frac{S_{large}}{S_{small}} = \frac{\sum w_y^*}{n - w_y^*} \geq \frac{\sum w_y}{n - w_y}$$

$$H_{large} = \frac{S_{large}}{\sum w_y} \geq \frac{S_{small}}{n - \sum w_y} = H_{small}$$

$$H_{large} > H > H_{small}$$

where S denotes the total trial size $w \times h$ of a trial set. Since $\mathrm{OPT}(I) \geq H$, we conclude

$$\mathrm{Fluid}(I) < \max(2\,\mathrm{OPT}(I) + \max(h_1)\alpha^{\frac{1}{\alpha-1}}, 2\,\mathrm{OPT}(I)\beta^{\frac{1}{\beta-1}-1})$$