High-throughput Simulation of Federated Learning via Resource-Aware Client Placement

Anonymous Author(s)**

ABSTRACT

Federated Learning (FL) is the privacy-preserving machine learning paradigm which collaboratively trains a model across millions of devices. Simulated environments are fundamental to large-scale FL research, allowing researchers to quickly test new ideas to solve system and statistical heterogeneity issues. This work proposes *Pollen*, a novel resource-aware system capable of speeding up FL simulations by efficiently placing clients across distributed and heterogeneous hardware. We propose minimising server-GPU communication and using an efficient client placement policy based on the inherent trade-offs of FL client placement on heterogeneous GPUs. These trade-offs are explored experimentally.

This exploration has been conducted via relevant baselines on three popular FL tasks: image classification, speech recognition and text generation. We compare *Pollen* to existing ad-hoc FL frameworks, such as Flower, Flute and FedScale, and show performance gains of 50% to 400%.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; *Neural networks*; **Parallel computing methodologies**; **Parallel algorithms**; *Massively parallel algorithms*; **Simulation environments**; • **Computer systems organization**;

KEYWORDS

federated learning, simulation, scalability, deep learning, resource management

ACM Reference Format:

Anonymous Author(s). 2023. High-throughput Simulation of Federated Learning via Resource-Aware Client Placement. In *Proceedings of The 29th Annual International Conference On Mobile Computing And Networking (MobiCom 2023).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

MobiCom 2023, Oct 02–06, 2023, Madrid, Spain

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Machine learning (ML) is becoming increasingly viable on constrained hardware with low memory such as smartphones, wearables, and general IoT devices. Techniques originally developed for training models on smartphones with hundreds of MB of memory [9] can now run on micro-controllers with 256kB [18]. Federated learning (FL) was introduced by McMahan et al. [20] to allow training on thousands to millions of such edge clients in a distributed manner while avoiding the privacy and network costs of communicating their sensitive data in a centralised fashion.

While Federated learning allows edge devices to collaborate effectively, Kairouz et al. [11, sec. 3.1] indicate that it brings unique challenges relating to clients' different hardware and data distributions. To effectively model and address such challenges, researchers must be able to run large-scale Federated Learning simulations efficiently on their hardware.

Unlike centralised ML, dominated by long-running execution of large jobs, simulating federated learning relies on the repeated execution of small clients corresponding to the resource-constrained devices it intends to emulate. Since such clients cannot keep GPU utilisation high by themselves, scheduling many of them to run both on the same GPU and across GPUs is beneficial. Crucially, clients can vary in dataset size by orders of magnitude. Independently on the client selection procedure, existing FL frameworks [2, 6, 15] treat all clients equally during simulations and can often end up with straggling GPUs. A proper *placement* of clients on GPUs could significantly reduce training time and allow more extensive simulations. Unfortunately, such intelligent resource-based *placement* is impossible in existing FL frameworks as they rely on a *pull-based* system where workers sequentially sample clients from a server queue.

In this work, we propose *Pollen*, an adaptive *push-based* method for client placement in simulated FL, that is compatible with diverse client selection procedures, and show that it significantly improves FL training times. Instead of building an entirely new framework, we chose to modify Flower [2] due to its flexibility. We showcase the design of our system and its effectiveness and analyse the factors which decide the training time of the simulation for a given placement policy. Our contribution is *threefold:*

(1) We experimentally show a proportional but nonlinear relationship between the size of a clients' dataset and their training time across multiple workloads

and GPU types. We argue that this requires intelligent client placement to optimise training time when combined with the skewness of standard FL datasets.
(2) We build *Pollen*, a client placement system to effectively partition clients amongst potentially heterogeneous GPUs using several placement strategies. Then, we compare the effectiveness of such strategies. Our results show that for heterogeneous GPU environments, a learning-based policy outperforms others by up to 81% as it can accurately learn to predict training time regardless of system configuration.
(3) Finally, we show that *push-based* client placement can significantly improve training time over previous *pull-based* systems. *Pollen* achieves a 50% to 378% reduction in training time across various datasets when training 10 000 clients split evenly across 100 rounds with heterogeneous GPUs. Moreover, for homogeneous GPUs, our method decreases communication costs enough to obtain a 200% to 400% improvement over the next-fastest system.

Our method allows FL researchers to run more extensive, faster, and scalable simulations. These improvements permit rapid prototyping and development of algorithms for production settings involving millions of edge devices.

2 SIMULATING FEDERATED LEARNING

We now introduce federated learning and describe the main characteristics of the standard simulation solutions adopted by popular federated learning frameworks. We then justify our proposed system *Pollen* based on the observed limitations of previous systems.

2.1 Federated Learning

Federated Learning is concerned with training ML models in a distributed fashion while minimising communication costs and maintaining private data on-device. In its most popular cross-device version [11], it takes the form of synchronised training where many clients pool their resources to train a model collaboratively. Such devices can differ significantly both in terms of their local data and in terms of available hardware. For example, a group of devices for human activity recognition could be composed of smartphones containing gyroscopes and accelerometers [24, 25], surveillance video cameras [14], and passive sensing devices using Radio-Frequency data [12]. Clients in these diverse categories would all have highly divergent data modalities, dataset sizes, computational power, network speed and training availability. Human activity recognition data can be sensitive and often needs to stay private to the point of origin, thus requiring a federated approach.

Federated Learning algorithms, like Federated Averaging [20] for cross-device FL, maintain data privacy via a client-server training design synchronised across *rounds*. The server controls the training and holds the federated model. It sends the model to each client at the start of the round, where it is trained on private data using Stochastic Gradient Descent (SGD). Then, the clients return the models to the server, aggregating them to create a new federated model for the next round.

2.2 Framework Simulation Engines

The simulation engines of FL frameworks aim to enable experiments in a constrained environment where we may not have enough resources to virtualise all the clients in the FL setting or all the clients in the sampled cohort for each round. We can better understand this scenario by representing each client's training as a job to schedule. Each of these jobs needs to allocate resources to account for a complete copy of the model to train, the dataset for the client, including the preprocessing, and the instructions for the training. The needs of a single client are usually relatively small; as such, it is possible to fit many of them into a single GPU.

Given this setting, FL simulation engines try orchestrating the training using a server-workers paradigm. The server orchestrates the simulated FL training by serving clients to workers and aggregates the results after every round of training. The workers are responsible for training clients individually and sending the trained models to the server once the training is complete. Ad-hoc FL frameworks, such as Flute [6], FedScale [15] and Flower [2], rely on this paradigm.

It is worth highlighting that workers in both FedScale and Flute are statically allocated to their GPU. On the other hand, Flower's Virtual Client Engine, developed by Beutel et al. [2] and based on Ray [21], can dynamically move workers between GPUs during the FL training. However, the control over this dynamic allocation is limited. In all these frameworks, the server serves clients to workers using a pull-based queuing system involving many communication steps between the server and workers. The execution of an FL round on this system can be summarised as follows.

(1) At the start of a round, the server samples a subset of participating clients. This subset defines the cohort of clients to be trained in that round. The cohort is kept in a synchronised queue, allowing workers to read clients sequentially.
(2) Each worker reads from the queue, extracts the first client, and starts the training. Different workers cannot access the same element of the synchronised list.
(3) Once a worker finishes training a client, it pings the server to notify that the client's training is completed.

(4) When the server receives this message, it replies to the worker when it can receive the training results.
(5) The worker finally sends the results to the server that will store them or partially aggregate them, depending on the experiment's configuration.

The number of workers controls the concurrency of this embarrassingly parallel simulation. Hence, it is beneficial to increase the number of workers up until the resources of the system are saturated, or gains from concurrency cease.

2.3 Limitations of Current Systems

The limitations of pull-based queuing systems are multifold. Critically, the workers of a specific GPU cannot choose which client to train. This underlying lack of control can limit the simulator's options when attempting to train specific clients on specific GPUs. For example, when disproportionately large clients are selected, balancing client training time across GPUs and avoiding stragglers is impossible.

Another general limitation is that the communication involved may take a significant amount of time relative to the training time of most clients. Such communication bottlenecks are significant for settings with multiple machines (nodes) that need a network connection to communicate with each other. In this work, *multinode* refers to hardware configurations with GPUs distributed amongst separate machines.

We now present the specific limitations of each framework addressed in our paper.

Flute, introduced by Dimitriadis et al. [6], is optimised to use the *nccl* backend of PyTorch Distributed [16] when running on GPUs and the *gloo* backend for CPU training. It can only run a single worker per GPU and, because it cannot intermix GPU and CPU training, it requires an entire GPU to hold the parameter server that handles aggregation during FL simulation. This can be wasteful as aggregation is usually not a compute-intensive operation. These issues cannot be easily addressed as the codebase has highly coupled components dependent on this design.

FedScale, introduced by Lai et al. [15], depends on unreliable configurations of gRPC, which is felt across the many communication steps necessary for the pull-based design. As such, longer rounds may cause crashes as clients appear to either disconnect or time out. Furthermore, despite being able to place multiple workers on the same GPU, later experiments show minimal benefits when increasing either the number of workers or GPUs. Each worker is also tasked with loading the entire dataset, even if they are on the same node as other workers and can share memory. Finally, similarly to Flute, the codebase coupling level makes refactoring difficult.

Flower, introduced by Beutel et al. [2], depends on Ray [21] as its simulation engine. Ray may cause out of memory (OOM) in a multi-GPU configuration as workers do not fully deallocate memory from previously used GPUs. Careful configuration of parameters helps avoid OOM at the cost of slowing down the overall training. However, unlike Flute and FedScale, the codebase is highly modular, allowing us to implement our new simulation engine on top of existing components.

General limitations of GPU scheduling for ML tasks are also worth mentioning. Job scheduling on a GPU cluster for traditional ML tasks is a well-studied problem [7, 29, 30]. For example, Gandiva [29] schedules large jobs on such a cluster by profiling the rate at which they process mini-batches. At the same time, CODA [30] attempts to balance the CPU assignment of GPU jobs to optimise data loading. However, both rely on the assumption that ML tasks are highly repetitive and run long enough to be effectively optimised. As we will show in Section 3, client dataset sizes in FL are much smaller than a typical ML job and more difficult to profile. Furthermore, client dataset sizes are highly skewed, making round durations unpredictable. Together, these two considerations make the heuristics of existing GPU scheduling systems insufficient for FL. This insufficiency drives us to propose an FL-specific solution rather than plugging in an existing one.

3 HETEROGENEITY AND CLIENT PLACEMENT

We now describe the practical difficulties of deciding which GPU a client should be placed on for training. For these, we identify two causes in the form of client heterogeneity and hardware heterogeneity.

3.1 Client Heterogeneity

In large-scale cross-device FL, clients collect or produce data at vastly different rates [17]. Efficient simulations should reflect this fact and account for the different training times that simulated clients will take. While some frameworks try to circumvent this issue by fixing the number of steps each client will train for [15], this assumes that the dataset distribution is not highly skewed and leads to two issues.

First, it limits the contribution of clients having large datasets. Second, clients having small datasets are forced to reuse their data (increase in local epochs). Consequently, we refrain from fixing a constant number of training steps throughout our experiments and argue that this is not a reasonable assumption to be made at a framework level.

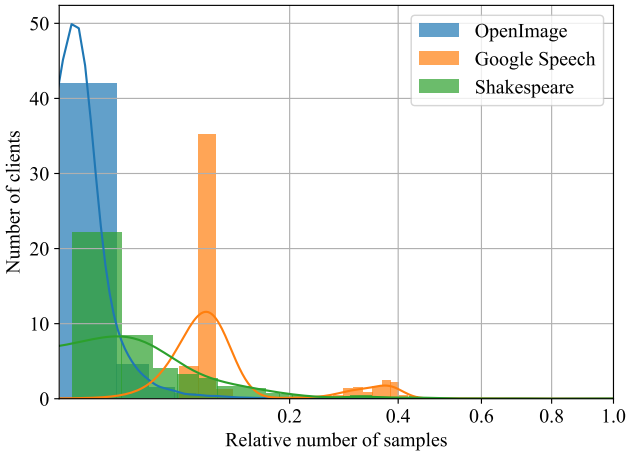

Figure 1: Dataset size distribution over clients for OpenImage [13], Google Speech [28], and Shakespeare [3]. Natural non-uniformity in data distributions will lead to different training times. A non-linear scale was chosen for the x-axis.

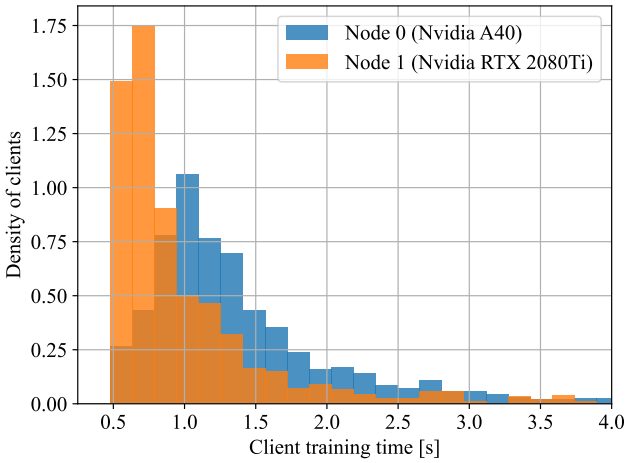

Figure 2: Training times for two GPUs running on different client dataset sizes. The GPUs are running the same clients.

Fig. 1 shows the distributions of samples for three different naturally-partitioned datasets commonly used in FL simulations. FL datasets are usually long-tailed and skewed, leading to the abovementioned issues.

We argue that the dissimilarity in dataset distributions makes it necessary to design dynamically adaptive placement strategies capable of accommodating different datasets. Additionally, the various pre-processing pipelines such datasets use for mini-batches reinforce this requirement. For example, image datasets may require samples to be loaded from disk and transformed, while language datasets may store features in memory. We further argue that the internal skewness of a dataset requires intelligent placement of clients on devices even for the simplest case of *homogeneous* GPU configurations. For example, in extreme situations, one device may train clients with orders of magnitude more batches than another.

3.2 Hardware Heterogeneity

Besides client heterogeneity, it is necessary to consider the different GPUs used in FL simulations. As previously mentioned, individual client workloads are smaller than traditional ML tasks and are highly parallelisable, allowing them to be trained across various machines with little effort. However, using heterogeneous GPUs may result in workers finishing processing their clients at very different times, leading to unnecessarily long experiments.

Roughly speaking, training a single client will depend on *data loading* and *actual training*. Data loading and preprocessing generally happen on the CPU. As the number of concurrent CPU jobs increases beyond the number of CPU cores, the variability of the data loading time grows proportionally to that of the scheduling. This can act as a bottleneck when increasing the number of workers available for processing.

The actual training of a client usually happens inside the GPU, and the time it takes is affected by the client's local dataset size. For large clients, the training time is approximately determined by the average speed at which the GPU can process each of their batches. However, for small clients, the startup times are a more significant section of the total time the clients spend executing, which causes increased variability in total training time. Furthermore, we observe some variability even for huge clients, which should be the least affected by startup concerns.

Figure 2 shows the training time distribution of two Nvidia GPUs running the same population of clients with different dataset sizes. As can be observed from the figure, the two GPUs perform very differently from one another. Therefore, allocating less work to the slower GPU is necessary to optimise training time by having them finish simultaneously.

4 POLLEN DESIGN

This section describes the key ideas and components used in our proposed solution. An overview of the complete system, dubbed *Pollen*, can be seen in Fig. 3.

4.1 Placement Strategy

Following our investigation in Section 2, we propose an alternative client placement system that addresses the issues reported in Section 2.3 regarding the limitations of FL frameworks. In our approach, instead of having workers *requesting*

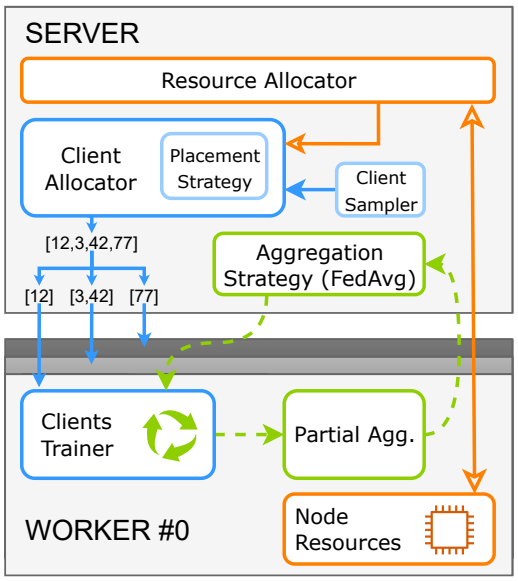

Figure 3: Diagram describing *Pollen* with its relevant elements in both server and worker. The colour code has been used to distinguish between components related to clients (blue), models (green), and hardware (orange).

clients from the server, the system follows a *placement strategy* to partition client workloads across workers and perform a *push-based* allocation. This method allows us to reduce the number of communication steps between the server and nodes and to assign sets of clients to appropriate GPUs.

It is worth mentioning that our placement method acts on the underlying simulation layer of the FL framework. It is an independent procedure from *client selection* and is not affected by the sampling procedure used to generate the clients for a specific round nor any other algorithmic properties. It can be easily extended to use other sampling techniques such as FedCS [22], Power-of-Choice [5] and DivFL [1].

4.2 Resource Allocator:

Clients having different amounts of data and being trained on heterogeneous GPUs will produce disparities in workers' execution times. A solution to this is to be able to associate client workloads with appropriate GPUs.

At the beginning of the FL simulation, the *resource allocator* module receives information from all training nodes regarding their available hardware, e.g., the number of CPU cores and the number and types of GPUs. This information is used to allocate resources to workers, following user-defined constraints, such as the maximum number of workers per GPU, which can be estimated by profiling a single inference step for each GPU type contained in the node.

4.3 Partial Aggregation:

When using associative aggregation strategies, such as FedAvg [20], the system can benefit from partially aggregating results within a worker before sending the partial result for a last aggregation on the server.

In this approach, the worker keeps both a partially aggregate model θ_k^p and a total number of processed data samples N_k after having trained its k-th client, as seen in Eq. (1).

$$\theta_{k+1}^p = \frac{\theta_k^p \times N_k + \theta_{k+1} \times n_{k+1}}{N_{k+1}} \quad (1)$$

$$N_{k+1} = N_k + n_{k+1} \quad (2)$$

Once the *worker* has completed training its list of clients, it will send both the partially aggregated model and the total sum of the samples for the final aggregation.

4.4 Client Allocator:

We follow the concept from Section 2 and define a *worker* as an entity capable of sequentially training lists of clients.

As FL clients can only use small batch sizes and models when training on edge devices, packing many workers and oversubscribing GPUs has proven beneficial in realistic FL simulations, as seen in Fig. 4.

The *client allocator* associates clients with individual workers on specific GPUs. The server samples a list of participating clients at the beginning of a round. Then, it passes this module, which uses a pre-defined *client placement policy*, discussed in Section 5, to determine which worker will train which client and in what order.

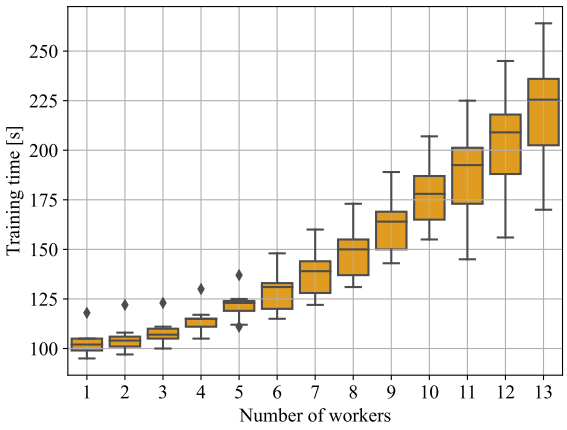

Figure 4: The distribution of clients' training time for different values of concurrent workers on the same GPU. Every worker has the same load: the same list of 100 clients. For example, with 5 workers, the GPU trains 500 clients using 5 concurrent processes.

5 CLIENT PLACEMENT STRATEGIES

This section discusses the strategies for placing clients on GPUs that we have explored in this work.

Since our design allows the simulation to have only one communication step from the server to the workers, our exploration focused on distributing the list of N randomly sampled clients across workers in one step. For each round, all these strategies receive the list of integer clients' IDs to be trained, and they return a list of clients' IDs for each worker indicating which clients it should train. Since the exploration space for determining the best distribution procedure is very broad, we rely on FL features common to most experiments.

Naïve round-robin (RR): This strategy represents the starting point of our investigation since it naively splits the list of clients in k uniformly populated lists, where k is the number of workers. In particular, the first sampled client will be assigned to the first worker and the second client to the second worker, etc. If $\frac{N}{k}$ is not an integer, the remainder is distributed across the first workers.

Batch-sorted round-robin (SRR): The first information we used in this study was the number of batches m each client has. As discussed above, the number of batches is a proxy for the training time in epochs-based FL training. Before naively splitting the list as before, this strategy orders the clients by m from top to bottom. The intuition behind this procedure is that different workers will likely train the biggest clients.

Batch-Uniform distribution (BU): A step forward to the previous strategy is to use the same information while changing the distribution procedure. For homogeneous GPUs, we want the load across workers to be balanced. To achieve this, the strategy loops over the N clients after ordering them by m from top to bottom and assigns the current client to the worker whose load is lower. The load is estimated by summing the number of batches of all the clients assigned to the worker. It has to be noted that the first k clients of the list are assigned the same way as the previous strategy.

Learning-based time prediction (LB): In settings with heterogeneous GPUs, the proxy for the training time given by m may not be sufficient, as shown in previous sections. Our learning-based strategy predicts the training time for each GPU. The first FL round will use the naïve round-robin strategy to collect data about client training time from all available workers. Starting from the second round, the strategy builds one dataset for each GPU composed of tuples $(client\ training\ time,\ m)$ from previous rounds.

Then, for each dataset, the strategy fits the data points to the function in Eq. (3), where y represents the client training time, and x is the number of batches the client has.

$$y = ax + b \log(cx) + d \tag{3}$$

The parameters derived from the fitting are then used for predicting the training time of the clients sampled in the current round. The strategy sorts the workers by GPU type, from the fastest to the slowest, using the predicted training time of the biggest client in the current cohort. Finally, the strategy carries on the placement of clients by balancing the load between workers, similar to the batch uniform strategy. Clients are ordered by m from top to bottom and assigned to the worker whose load is lower. Here, the load is estimated by summing the predicted training time of all the clients assigned to the worker.

5.1 Efficient Client Placement

In this work, we try to answer the question of which placement strategy is better to choose in which context. More importantly, we argue that discussing placement strategies in simulating FL is necessary. The research on large-scale [4, 26, 27] FL often relies on simulating FL settings with large cohorts (in the range $[10^2, 10^4]$) of clients over a relatively limited amount of hardware resources for thousands of rounds. Reducing the latency of experiments as much as possible while exploiting the hardware resources at their best is fundamental. Regardless of the chosen placement strategy, we expect any one-step communication method to outperform

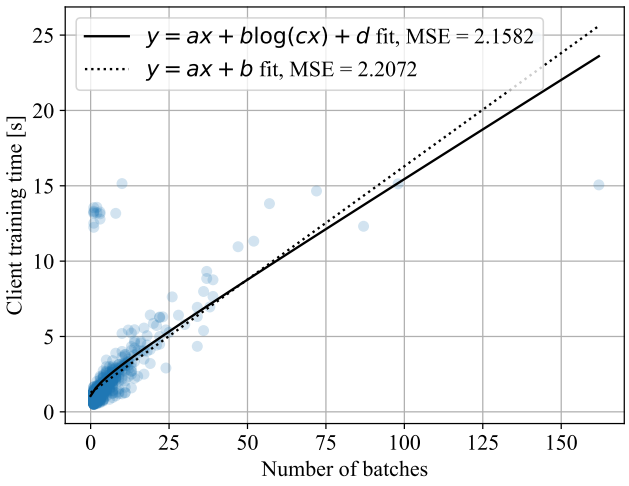

Figure 5: Clients' training times are plotted against their number of batches. The fitting lines of the linear function and the proposed function are shown alongside The Mean Squared Errors (MSE) derived from the fitting procedure.

the queue design in previously mentioned frameworks. However, we will show that an optimal placement can improve the simulation speed by up to 81% compared to the slowest strategy.

Different placement strategies will result in different distributions of clients to train across workers. However, this does not hold for the degenerate placements in which the number of clients trained per round equals the number of workers, and the workers are allocated on the same GPU and node. The Shakespeare baseline is the only one we have explored that can fall into this degenerate case. This is because the training procedure plans to train 10 clients per round, which can easily be fitted into a single GPU. Excluding the degenerate placement, the non-trivial metric that is impacted by different placement strategies is what we call "*timedelta workers*". We define "*timedelta workers*" as the time difference between the timestamp at which the fastest worker finishes their job and the timestamp at which the slowest worker finishes their job, where we intend the job to be the training of all the clients in the received list. When using the RR strategy, we observe that the distribution across rounds of "*timedelta workers*" is peaked at a time at least one order of magnitude greater than the training time of the smallest client in that round. We could have reduced the training time by moving the smallest client from the slowest to the fastest worker. Thus, we assume that "*timedelta workers*" approximates the time wasted due to clients' misplacement.

The impact of the placement strategy on the training time depends on the GPUs we are dealing with. We can distinguish between *homogeneous settings*, in which each worker executes on the same GPU type at the same speed, and *heterogeneous settings*, in which groups of workers are allocated on different GPU types having different speeds.

In *homogeneous settings*, we expect to observe that the ratio between the number of clients per round and the number of workers drives the difference in performance. As this ratio increases, we expect to observe similar performance across strategies. This expectation is motivated by the fact that the noise introduced by the random sampling of clients will likely balance the load over workers naturally. When this ratio is close to 1, only sampled cohorts of clients whose number of large-size clients is different from the number of workers can lead to different performance. In this case, the performance gap appears because big clients cannot be evenly distributed across workers. However, we expect the BU strategy to slightly outperform the others since it can balance the load over workers with the minimum overhead.

The scenario in which different strategies will significantly differ in performance is the *heterogeneous setting* in which clients of the same size are not trained in the same amount of time by different workers. For these settings, the LB strategy will outperform BU because of its ability to discriminate between workers executing on different GPU types. The difference in performance will mostly depend on how heterogeneous the hardware settings are. In particular, the gap between different hardware in training small clients, prevalent in FL datasets, will have the most critical impact.

The LB placement strategy strongly relies on the performance of the fitting procedure over clients' training time from previous rounds. It is worth discussing the two main ingredients upon which this strategy arranges the clients' placement: the data collected for previous rounds and the fitting function. We chose to keep all the data from previously trained rounds. In general, the robustness of curve fitting is proportional to the number of data points; conversely, its duration is proportional to the number of data points. We observe that during the last round, when the data points are 9900, the LB strategy takes an amount of time on the same order of magnitude as the RR strategy, tens of seconds. For extended experiments where the number of data points to fit could be much more significant, it is reasonable to define a time window for deleting older data. Second, our fitting function has been chosen for its mathematical properties. The logarithm combined with a linear term ensures that the fitted function never predicts negative values despite the wide cloud of data points produced by small clients, as can be observed in Fig. 5. Since small clients have greater training time variance, many may take longer than their slightly bigger counterparts. This behaviour may enforce negative slopes in the fitted curve, especially if polynomial. The logarithmic term makes the function more robust to this situation, while the linear term ensures that the bigger clients are predicted to take longer to train. *As such, the chosen function allows us to avoid ever predicting a native time for a client at the cost of overestimating the duration of small clients.*

6 EXPERIMENTAL DESIGN

This section describes a series of experiments meant to showcase our method's superiority and validate it.

6.1 Federated Learning Tasks

We use three representative FL tasks throughout this work to showcase our method. These are characterised by having clients with long tail distributions over the number of samples and workloads. For all tasks, we exclude clients with less than one batch of training data.

Image Classification: The goal of this task is to collaboratively train a ShuffleNetV2 [19] network to correctly classify images amongst 596 classes. For this, we use a federated version of the original OpenImage [13] dataset as implemented

by FedScale. This dataset contains 1.6×10^6 images partitioned across 13 771 clients. We use a batch size of 20 samples.

Speech Recognition: In this task, we use the Google Speech Commands dataset [28] to collaboratively train a ReseNet-34 [10] to classify audio samples amongst a set of 35 predefined spoken words. The dataset contains a collection of $157K$ one-second-long clips naturally partitioned according to their 2168 speakers. We use a batch size of 20 samples.

Text Generation: We use the Shakespeare dataset, as implemented in TensorFlow Federated [8], to train a two-cell LSTM-based language model as defined in [3] The dataset comprises sentences extracted from *The Complete Works of William Shakespeare* and grouped into 648 fictional characters. We follow the LEAF experimental configuration and use a batch size of 4 samples.

6.2 Hardware Configuration

We consider two hardware configurations that reflect common research centres, namely *single-node* and *multi-node*. We also further distinguish between simulations using *homogeneous* and *heterogeneous* GPUs. As previously indicated, all the FedScale, Google Speech, and Shakespeare experiments use a fixed batch size. This makes differences in worker VRAM attributable only to model size and input data shape.

Single-node: Our single-node experiments are all run on *node 0* containing Nvidia A40 GPUs and an Intel (R) Xeon (R) Gold 6152 with 88 cores. For OpenImage, the A40s are filled with 13 workers each, given the size of ShuffleNetV2 [19] and the input size. For Google Speech, we use only 4 workers per GPU due to data loading requirements, while for Shakespeare, we use 10 workers total to match the number of clients. Each A40 is paired with 11 CPU cores out of the 88 mentioned above. We run all our experiments with *Two homogeneous GPUs* on this node using two A40s with 22 CPU cores available.

Multi-node: Our multi-node experiments are run on a combination of the aforementioned *node 0* containing A40s and *node 1*. *Node 1* contains Nvidia RTX 2080 Ti GPUs and an Intel(R) Xeon(R) CPU E5-2680 v4 containing 56 cores. For OpenImage, we use 4 workers per 2080 due to VRAM constraints. The worker setup for Google Speech and Shakespeare is the same as on *node 0*. Each 2080 is paired with 8 CPU cores out of the 56 mentioned above. Our experiments for heterogeneous hardware shall use one A40 from *node 0* paired with 1-4 Nvidia 2080s. We only use more than one 2080 in scenarios where we want to observe how balancing the number of workers across GPU types affects performance. After the workers have completed their partial aggregation on the *CPU* of their respective node, all final aggregation steps happen on *node 0* on the *CPU*.

6.3 Framework Benchmark

The fundamental contribution of the design of our system is to overcome the pull-based queuing system to allow for fast FL simulation. In this experiment, we aim to assess the speed of our solution, whatever the placement strategies adopted by the client allocator. We compare to the other frameworks by measuring the throughput of the simulation since this describes the system's speed as the number of clients the system can train per second. It is calculated by dividing each round completion time by the number of clients trained. We perform the three tasks under different homogeneous hardware settings having 1, 2 or 4 homogeneous GPUs. For homogeneous hardware, different placement strategies perform similarly. Thus, the performance is mainly impacted by the system design.

6.4 Policies Benchmark

This work proposes different placement strategies to use in the simulation. In this experiment, we aim to answer the question of how the choice of strategy impacts the performance of the simulation. We measure the throughput of the FL simulation and compare different strategies against each other. We are also interested in identifying how effective the strategies are in balancing the load between the workers. To this aim, we measure the time difference between the first and last workers to finish processing their clients and compare the strategies against each other.

The experiment is designed to be executed in the most challenging hardware setting, the heterogeneous one with two nodes having different GPU types. First, we train all the tasks in the setting with one GPU per type for two GPUs. We keep the number of clients per round to 100, except for Shakespeare, where we set 10 clients for 100 rounds. Here, we compare the throughput of the experiments using different strategies against the fastest other framework. Second, we train the Image Classification task in different heterogeneous hardware settings, increasing the number of GPUs belonging to the type that can host the lower number of workers. This second set of experiments uses the two previous nodes, one always having one GPU and the other having 1, 2, 3, and then 4 GPUs.

These settings reflect a typical situation a researcher could face: the available GPUs with more VRAM are few, but more GPUs with less VRAM are available.

6.5 System Scalability

Our main contribution in this paper is to allow efficient large-scale simulations at the scale of real-world applications. In this experiment, we aim to measure how our proposed method compares to other Placement Policies as we *increase the average number of clients per GPU*. The first set of experiments considers the homogeneous scenario of four GPUs of the same type on the same node. We fix the number of workers because the hardware is fixed, and we progressively increase the number of clients per round while keeping constant the overall number of clients trained. In order to maintain the total number of trained clients to 10 000, as it is in other experiments in this work, we choose the values for the number of clients per round to be 100, 200, 400, 625, 1000 training respectively for 100, 50, 25, 16, 10 rounds. The workers receive longer lists of clients to train each round, while the total work is always the same. We measure the system's throughput for all the proposed client policies and then compare them against the fastest framework.

A second set of experiments considers the most constrained and heterogeneous scenario we had: two nodes, each of them having on GPU of a different type. Similarly, we played with the number of clients each worker has to train in each round while keeping the total amount of work constant. For each round, we measure the time difference between the first and last worker to finish processing their clients and the throughput.

We run the above experiments on the Image Classification task with the largest client population. This setting reflects a common FL scenario where researchers want to investigate the benefits of sampling larger fleets of devices in one round while still constrained by the available hardware.

7 EVALUATION

In this section, we describe the results of the experiments proposed in Section 6. We begin presenting the comparison of the performance of *Pollen* against other frameworks. We continue focusing on the differences between the strategy proposed. Furthermore, we extend the investigation of the strategies by evaluating them at scale. Finally, a discussion about the proposed client placement's limitations is presented.

7.1 Comparison between frameworks

Pollen proves to deliver faster FL simulations compared to Flower (version 1.1) [2] , Flute (commit@0e8762b) [6], and FedScale (commit@9bfc029a3c) [15] in every setting we have tested. This consistency demonstrates the superiority of a push-based allocation of clients across workers. The results of the throughput of the simulation in a more accessible homogeneous setting put *Pollen* on top of the classification

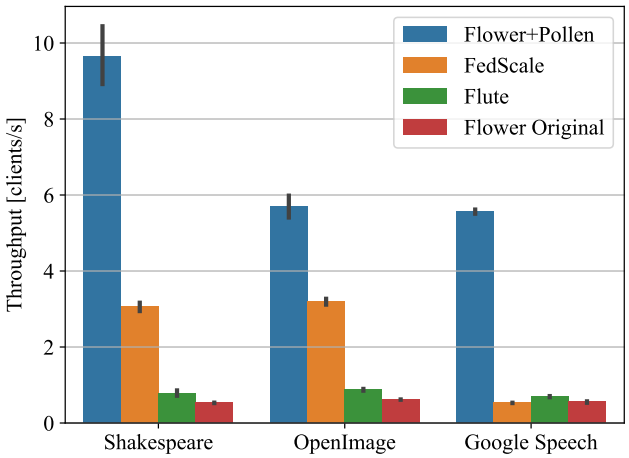

Figure 6: The throughput (the greater, the better) of our client placement systems implemented in Flower compared with FedScale, Flute and standard Flower. Using resource-aware client placement, *Pollen* outperforms the fastest other framework by a factor 3x-5x, depending on the task. The values reported for our contribution have been chosen for the best-performing strategy that tends to be LB.

since it outperforms all the other frameworks. The example in Fig. 6 shows that, when using two GPUs of the same type, namely Nvidia A40, on a single node, *Pollen* has the highest throughput for all the tasks. Compared to the fastest framework, the speed-up obtained is 3x on the Text Generation task, about 2x on the Image Classification task, and 5x on the Speech Recognition task. It is noted that the performance of all the tasks is varied across the other frameworks except for the Speech Recognition one, in which they perform similarly. We highlight that in every comparison in which the number of clients per round is fixed, the absolute time of the experiment is inversely proportional to the reported throughput. In addition, since the Image Classification task has the lowest speed-up factor, we assume that to be the task using which further comparisons will be fairer.

Implications: *Pollen* outperforms all previous frameworks and can provide a great boost to the scalability of FL simulations thus allowing better FL methods to be developed for production systems.

7.2 The Impact of Client Placement

Given that our design outperforms other frameworks in homogeneous settings, we evaluate the impact the client placement strategies in heterogeneous settings. Two nodes were available in the setting, with one Nvidia A40 and one Nvidia RTX 2080 Ti, respectively. The results regarding the throughput are shown in Table 1. We can assume that different strategies deliver the same performance for the Text

Generation task because distributing 10 clients across 10 workers is trivial. The Image Classification task shows the biggest variation across strategies, where the LB strategy of *Pollen* outperforms the others. We note that the RR strategy of *Pollen* underperforms other strategies when the placement is not trivial, presenting the most significant gap in the Image Classification task. We argue that the peculiarity of the Speech Recognition dataset causes different strategies to have more minor variations in performance. Furthermore, we highlight the general superiority of our method against the fastest framework in this challenging heterogeneous scenario.

We gradually increased the number of available GPUs for the second set of experiments to evaluate the impact of different placement strategies. In this experiment, we used the different strategy of *Pollen*. We significantly increased the number of the GPU type that can host the lower amount of workers, namely the Nvidia RTX 2080 Ti. As such, four settings are compared in this multi-node scenario: $(1, 1)$, $(1, 2)$, $(1, 3)$, $(1, 4)$, where the first number refers to the available Nvidia A40s in the first node and the second to the available Nvidia RTX 2080 Ti in the second node. The plot in Fig. 7 shows the throughput of different strategies over the Image Classification task. It is worth noting that the number of clients per round has been kept constant during this particular experiment. In this way, what is changing is the *density* of clients across workers, as having more GPUs available means having more workers. The results demonstrate that the gain produced by using the LB strategy degrades as the density of clients across workers decreases. This degradation is reflected in both the metrics taken into consideration, namely the difference in training time between the fastest and slowest workers (Table 2) and the throughput (Fig. 7).

Implications: Intelligent client placement is highly beneficial for hardware configurations containing multiple GPU types. Given the difficulty of profiling and configuring a large-scale cluster with heterogeneous GPUs, the automatic nature of workload distribution in *Pollen* provides a great advantage.

7.3 Scalability of Placement Strategies

Having already established the effectiveness of *Pollen* and analysed the difference between different client placement strategies, we are now concerned with the proposed method's scalability. We begin by comparing with FedScale in a homogeneous setting while increasing the *density* of clients across workers. In this experiment, instead of increasing the number of available GPUs we increase the number of clients per round while keeping the total number of clients at 10 000. This represents a common setting for large-scale experiments in which the number of clients per round can be very high. Such setting is often explored by theoretical works and it challenge in a simulation context. As Fig. 8 shows, our push-based placement strategies increase in throughput as the number of clients per round increases. However,

Table 1: The throughput (the greater, the better) measured in clients per second for three tasks. The proposed *Pollen*'s strategies are compared against FedScale in the most heterogeneous setting.

Datasets	FedScale	LB	BU	SRR	RR
Google Speech	3.6±0.5	4.7±0.4	**5.4±0.2**	5.0±0.2	5.2±0.3
OpenImage	3.6±0.5	**6±1**	4.7±0.5	5.2±0.8	4.1±0.9
Shakespeare	2.3±0.8	10±3	10±4	10±4	**11±4**

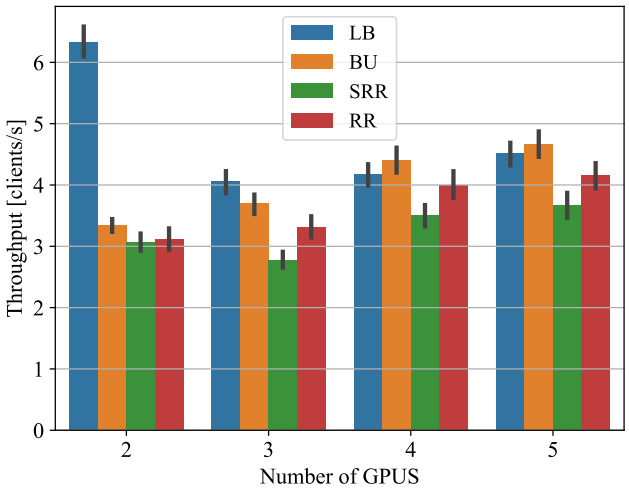

Figure 7: The throughput (the greater, the better) measured in clients per second of Round-Robin (RR), Sorted Round Robin (SRR), Batch-uniform (BU), and Learning-based (LB) client placement policies for different multi-node heterogeneous GPU configurations. The Image Classification task has been used.

Table 2: Here the difference in training time between the fastest and slowest workers (the lower the better) measured in seconds for Round-Robin (RR), Sorted Round Robin (SRR), Batch-uniform (BU), and Learning-based (LB) client placement policies using different multi-node heterogeneous GPU configurations. The Image Classification task has been used.

# GPUs in nodes (0, 1)	BU	LB	RR	SRR
(1, 1)	23±3	**10±4**	30±7	26±5
(1, 2)	21±4	**18±4**	28±6	31±6
(1, 3)	**17±4**	18±4	23±6	23±5
(1, 4)	**15±3**	16±4	21±5	21±6

the throughput of FedScale does not scale as the number of clients per round increases. This is to be expected since as the number of clients per round increases, *Pollen* does fewer and fewer total communication steps. On the other hand, FedScale does the same number of communication steps. We can also note that the exact placement strategy does not matter, even as the number of clients per round is scaled up, supporting our previous section results.

In the case of heterogeneous GPUs, our results shown in Table 3 indicates that the learning-based solution keeps its advantage over all others even as the number of clients increases. This is driven by the learning-based policy of properly distributing works across GPUs, even when many clients are involved in a round. The other placement strategies have a highly inconsistent ordering as they operate under the false assumption that workers should be treated equally. It is also worth noting that all the improvements in throughput we obtain are based on minimising the wait for the slowest worker at the end of a round. As such, while all placement strategies get an initial boost from increasing client density, once sufficient clients are available each round to keep all workers filled for most of the round, throughput stops increasing.

We now turn our attention to Table 4, which shows the difference between the fastest and slowest worker; we can observe the cause of this trend. When not accounting for hardware heterogeneity, the other policies reflect the speed difference between the GPUs the workers are placed on. It is clear that the learning-based solution minimises this gap relative to the other placement strategies.

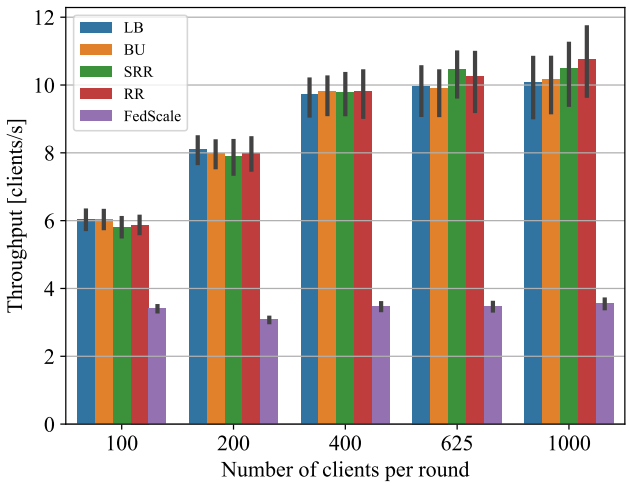

Figure 8: Throughput of the simulation for different placement strategies compared against the fastest framework in performing the Image Classification task. The total number of trained clients has been kept constant to 10000, while the number of clients per round has been increased. The hardware setting used was homogeneous with 4 Nvidia A40.

Implications: The benefits of client placement for heterogeneous GPU configurations persist as the number of clients processed in a round grows. Importantly, these benefits to be highly consistent after a sufficient number of clients per round is reached.

8 LIMITATIONS

While the results for the push-based client placement system we have presented are compelling and consistent, it does present several limitations, which we list below:

- For homogeneous settings, the improvements are brought by the push-based design, as client placement decisions do not generally matter when using random sampling.
- For heterogeneous settings, a very low number of clients relative to the number of workers does not allow for sufficient placement decisions. Thus, it is difficult for the learning-based method to balance load across workers.
- For vast numbers of clients per round, all placement methods approach their maximum throughput for the task and cease providing further improvements.
- The learning-based policy may not provide the theoretically optimal placement for two reasons. First, even when given a perfect prediction of training time for all clients by an oracle, the distribution of these clients over workers is still non-trivial. Second, the error in estimating each client may cause suboptimal placement decisions regardless of the number of available clients or placement strategy. As we observe from Table 4, the gap between the fastest and slowest worker is always proportional to the number of clients in a fixed manner. This indicates a consistent estimation error.
- The partial aggregation algorithm required to minimise communication is incompatible with some federated learning aggregation algorithms by default. For example, we do not currently support the adaptive optimisation proposed by Reddi et al. [23]

Table 3: The throughput (the greater, the better) for the Image Classification task using *Pollen*'s RR, SRR, BU, and LB client placement strategies with the most heterogeneous multi-node hardware configurations. The number of clients per round has been changed while keeping the total number of trained clients constant.

Num. Clients per round	BU	LB	RR	SRR
100	3.3±0.4	**6±1**	3.1±0.6	3.1±0.5
200	6.7±0.9	**8±1**	7±1	6±1
400	7.1±0.5	**7.9±0.7**	6.9±0.8	7.6±0.5
625	7.1±0.4	**8.4±0.7**	7.2±0.6	6.9±0.4
1000	7.0±0.4	**8.3±0.9**	7.2±0.5	7.4±0.3

Table 4: Here the difference in training time between the fastest and slowest workers (the lower the better) for Round-Robin (RR), Sorted Round Robin (SRR), Batch-uniform (BU), and Learning-based (LB) client placement policies using the most heterogeneous multi-node hardware configurations. The number of clients per round has been changed while keeping the total number of trained clients constant. The Image Classification task has been used.

Num. Clients per round	BU	LB	RR	SRR
100	23±3	**10±4**	30±7	26±5
200	24±3	**19±4**	30±6	27±5
400	48±3	**41±5**	56±6	43±3
625	77±4	**62±10**	83±8	78±5
1000	125±7	**106±18**	134±11	118±6

9 CONCLUSION

In this work we have shown that the current *push-based* approaches for client placement available in Federated Learning frameworks are incapable of exploiting both client and hardware heterogeneity. Based on this fact, we have proposed a resource-aware client placement algorithm which minimises communication costs between the server and workers responsible with training the clients. An extensive experimental evaluation has shown our proposed changes to bring improvements of 50% to 400%. This significant improvement allows for the quick development of algorithms with downstream applications in training fleets of millions of edge-devices. Since our modifications have a relatively small footprint, we recommend that all active FL frameworks adopt a similar design for their simulation engines.

REFERENCES

[1] Ravikumar Balakrishnan, Tian Li, Tianyi Zhou, Nageen Himayat, Virginia Smith, and Jeff Bilmes. 2022. Diverse Client Selection for Federated Learning via Submodular Maximization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=nwKXyFvaUm

[2] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. 2020. Flower: A Friendly Federated Learning Research Framework. *CoRR* abs/2007.14390 (2020). arXiv:2007.14390 https://arxiv.org/abs/2007.14390

[3] Sebastian Caldas, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2018. LEAF: A Benchmark for Federated Settings. *CoRR* abs/1812.01097 (2018). arXiv:1812.01097 http://arxiv.org/abs/1812.01097

[4] Zachary Charles, Zachary Garrett, Zhouyuan Huo, Sergei Shmulyian, and Virginia Smith. 2021. On Large-Cohort Training for Federated Learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 20461–20475. https://proceedings.neurips.cc/paper/2021/hash/ab9ebd57177b5106ad7879f0896685d4-Abstract.html

[5] Yae Jee Cho, Jianyu Wang, and Gauri Joshi. 2020. Client Selection in Federated Learning: Convergence Analysis and Power-of-Choice Selection Strategies. *CoRR* abs/2010.01243 (2020). arXiv:2010.01243 https://arxiv.org/abs/2010.01243

[6] Dimitrios Dimitriadis, Mirian Hipolito Garcia, Daniel Madrigal, Andre Manoel, and Robert Sim. 2022. FLUTE: A Scalable, Extensible Framework for High-Performance Federated Learning Simulations. https://www.microsoft.com/en-us/research/publication/flute-a-scalable-extensible-framework-for-high-performance-federated-learning-simulations/

[7] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2022. Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision. *CoRR* abs/2205.11913 (2022). https://doi.org/10.48550/arXiv.2205.11913 arXiv:2205.11913

[8] Google. 2019. Tensorflow Federated. https://www.tensorflow.org/federated

[9] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[11] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang

Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. 2021. Advances and Open Problems in Federated Learning. *Found. Trends Mach. Learn.* 14, 1-2 (2021), 1–210. https://doi.org/10.1561/2200000083

[12] Armand K. Koupai, Mohammud J. Bocus, Raul Santos-Rodriguez, Robert J. Piechocki, and Ryan McConville. 2022. Self-supervised multimodal fusion transformer for passive activity recognition. *IET Wireless Sensor Systems* 12, 5-6 (2022), 149–160. https://doi.org/10.1049/wss2.12044 arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/wss2.12044

[13] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper R. R. Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. 2020. The Open Images Dataset V4. *Int. J. Comput. Vis.* 128, 7 (2020), 1956–1981. https://doi.org/10.1007/s11263-020-01316-z

[14] HyeokHyen Kwon, Catherine Tong, Harish Haresamudram, Yan Gao, Gregory D. Abowd, Nicholas D. Lane, and Thomas Plötz. 2020. IMUTube: Automatic Extraction of Virtual on-body Accelerometry from Video for Human Activity Recognition. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3 (2020), 87:1–87:29. https://doi.org/10.1145/3411841

[15] Fan Lai, Yinwei Dai, Sanjay Sri Vallabh Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2022. FedScale: Benchmarking Model and System Performance of Federated Learning at Scale. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (Eds.). PMLR, 11814–11827. https://proceedings.mlr.press/v162/lai22a.html

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[17] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Process. Mag.* 37, 3 (2020), 50–60. https://doi.org/10.1109/MSP.2020.2975749

[18] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and song han. 2022. On-Device Training Under 256KB Memory. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=zGvRdBW06F5

[19] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV (Lecture Notes in Computer Science, Vol. 11218)*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer, 122–138. https://doi.org/10.1007/978-3-030-01264-9_8

[20] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, 1273–1282. http://proceedings.mlr.press/v54/mcmahan17a.html

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 561–577. https://www.usenix.org/conference/osdi18/presentation/nishihara

[22] Takayuki Nishio and Ryo Yonetani. 2018. Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge. *CoRR* abs/1804.08333 (2018). arXiv:1804.08333 http://arxiv.org/abs/1804.08333

[23] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. 2021. Adaptive Federated Optimization. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=LkFG3lB13U5

[24] Konstantin Sozinov, Vladimir Vlassov, and Sarunas Girdzijauskas. 2018. Human Activity Recognition Using Federated Learning. In *IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, ISPA/IUCC/BDCloud/SocialCom/SustainCom 2018, Melbourne, Australia, December 11-13, 2018*, Jinjun Chen and Laurence T. Yang (Eds.). IEEE, 1103–1111. https://doi.org/10.1109/BDCloud.2018.00164

[25] Catherine Tong, Shyam A. Tailor, and Nicholas D. Lane. 2020. Are Accelerometers for Activity Recognition a Dead-end?. In *HotMobile '20: The 21st International Workshop on Mobile Computing Systems and Applications, Austin, TX, USA, March 3-4, 2020*, Padmanabhan Pillai and Qin Lv (Eds.). ACM, 39–44. https://doi.org/10.1145/3376897.3377867

[26] Ewen Wang, Ajay Kannan, Yuefeng Liang, Boyi Chen, and Mosharaf Chowdhury. 2023. FLINT: A Platform for Federated Learning Integration. *CoRR* abs/2302.12862 (2023). https://doi.org/10.48550/arXiv.2302.12862 arXiv:2302.12862

[27] Ewen Wang, Ajay Kannan, Yuefeng Liang, Boyi Chen, and Mosharaf Chowdhury. 2023. FLINT: A Platform for Federated Learning Integration. *CoRR* abs/2302.12862 (2023). https://doi.org/10.48550/arXiv.2302.12862 arXiv:2302.12862

[28] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *CoRR* abs/1804.03209 (2018). arXiv:1804.03209 http://arxiv.org/abs/1804.03209

[29] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[30] Han Zhao, Weihao Cui, Quan Chen, Jingwen Leng, Kai Yu, Deze Zeng, Chao Li, and Minyi Guo. 2020. CODA: Improving Resource Utilization by Slimming and Co-locating DNN and CPU Jobs. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*. IEEE, 853–863. https://doi.org/10.1109/ICDCS47774.2020.00069