Pecan: Cost-Efficient ML Data Preprocessing with Automatic Transformation Ordering and Hybrid Placement

Dan Graur* *ETH Zurich*
Oto Mraz* *ETH Zurich*
Muyu Li *ETH Zurich*
Sepehr Pourghannad *ETH Zurich*
Chandramohan A. Thekkath *Google*
Ana Klimovic *ETH Zurich*

Abstract

Input data preprocessing is a common bottleneck in machine learning (ML) jobs, that can significantly increase training time and cost as expensive GPUs or TPUs idle waiting for input data. Previous work has shown that offloading data preprocessing to remote CPU servers successfully alleviates data stalls and improves training time. However, remote CPU workers in disaggregated data processing systems comprise a significant fraction of total training costs. Meanwhile, current disaggregated solutions often underutilize CPU and DRAM resources available on ML accelerator nodes. We propose two approaches to alleviate ML input data stalls while minimizing costs. First, we dynamically schedule data preprocessing workers on ML accelerator host resources to minimize the number of remote CPU workers needed to achieve peak data ingestion bandwidth. Second, we analyze the characteristics of input pipelines and automatically reorder transformations to increase data preprocessing worker throughput. We observe that relaxing commutativity increases throughput while maintaining high model accuracy for a variety of ML data pipelines. We build *Pecan*, an ML data preprocessing service that automates data preprocessing worker placement and transformation reordering decisions. Pecan reduces preprocessing costs by 87% on average and total training costs by up to 60% compared to training with state-of-the-art disaggregated data preprocessing and total training costs by 55% on average compared to collocated data preprocessing.

1 Introduction

Input data processing is essential for training machine learning (ML) models. Data transformations applied on-the-fly during training (e.g., shuffling, sampling, and randomly augmenting data) improve model generalization and accuracy [12]. Data preprocessing also impacts end-to-end training time and cost. As ML input data pipelines typically execute on CPUs to support user-defined transformations [54], preprocessing data fast enough to keep up with the high data ingestion bandwidth of ML hardware accelerators is challenging [70]. Avoiding input data stalls is critical as idling expensive GPUs or TPUs significantly increases training time and cost [5, 10, 22, 32, 47, 53, 58, 66].

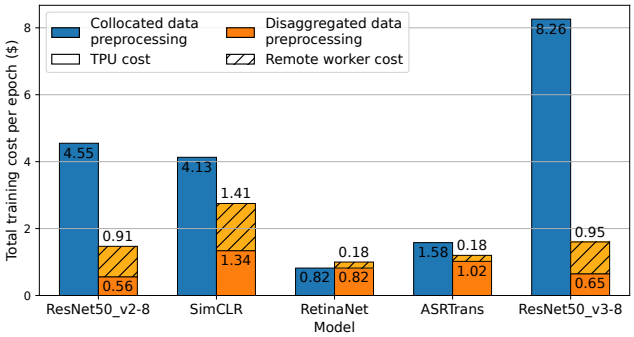


Figure 1: Disaggregated data preprocessing reduces cost for training jobs that are input-bound with collocated data preprocessing (e.g., ResNet50, SimCLR). In disaggregated jobs, remote CPU workers make up a sizeable fraction of total cost.

Prior work proposes to eliminate data stalls during ML training by disaggregating and offloading data preprocessing to remote CPU servers [5, 22, 78]. In a *disaggregated* deployment, remote workers (i.e., commodity VMs with moderate CPU/DRAM allocations) preprocess data and send batches over the network to ML accelerator nodes i.e., training clients equipped with accelerators such as GPUs or TPUs). In contrast to a *collocated* deployment, where preprocessing runs on the local CPU and memory of ML accelerator hosts, disaggregation allows independent right-sizing of CPU and DRAM resources for data preprocessing. This flexibility alleviates bottlenecks, as each training job has unique resource requirements [5, 22, 53]. Figure 1 shows that offloading preprocessing to remote CPUs (e.g., with Cachew [22]) reduces training costs by up to 81% compared to collocated preprocessing on ML accelerator hosts. By scaling out data processing, Cachew eliminates data stalls and maximizes training throughput, re-

*Authors contributed equally to this work.

ducing the time that expensive accelerators are used. In production ML training jobs, this can improve end-to-end time by up to 99% [5].

However, Figure 1 shows that the remote CPU servers required to feed model training without stalls can comprise a significant portion of end-to-end training costs in fully disaggregated deployments (e.g., on TPUv2-8: 62% for ResNet, 51% for SimCLR; on TPUv3-8: 15% for ASRTransformer (ASRTrans), and 59% for ResNet). Furthermore, some models (e.g., RetinaNet on TPUv2-8) are not input-bound with collocated data processing and do not need remote workers at all to avoid data stalls. A key drawback of fully disaggregated preprocessing systems like Cachew [22] and Meta's DPP [78] is that they do not leverage the available CPU and DRAM resources on ML accelerator nodes for data preprocessing. These resources remain idle and cannot easily be used by other datacenter workloads, as accelerator nodes are typically dedicated to ML workloads to avoid interference [5,36,70,72]. Underutilizing ML accelerator hosts is wasteful, particularly as these servers are typically equipped with many CPU cores and high DRAM capacity per accelerator. For example, Google Cloud TPU virtual machines (VMs) have 96 CPU cores and 335 GB of DRAM per 8-core accelerator [18]. Users must pay for all resources on accelerator VMs, regardless of their utilization. Furthermore, users pay extra for each remote CPU server used for disaggregated data preprocessing.

The main question we address in this work is: *how can we alleviate ML input data stalls more cost-effectively?* We have two main insights. First, we dynamically schedule data preprocessing workers across ML accelerator host resources, which minimizes the number of remote CPU servers needed to avoid data stalls. Second, in contrast to prior work that treats the input pipeline as a black box, we analyze the characteristics of data transformations in data pipelines and reorder transformations to maximize throughput. For example, we can increase throughput by placing transformations that reduce the volume of data early on in the pipeline while pushing transformations that increase the volume of data toward the end. While many transformations are not commutative, we observe that for a variety of pipelines that randomly augment data, relaxing commutativity does not impact model accuracy. The reordered input pipelines have higher throughput per CPU worker and hence require fewer remote workers to saturate accelerator ingestion bandwidth, which reduces cost.

Applying these insights to real input data pipelines is nontrivial. Finding the optimal fraction of data preprocessing workers to schedule locally vs. remotely is a complex optimization problem. It involves finding the right balance of three types of tasks that run on ML accelerator hosts: local data preprocessing, network processing (including deserialization and decompression) for data batches arriving from any remote workers, and data loading to ML accelerators. Optimizing preprocessing throughput via transformation reordering requires analyzing how each transformation affects a data element's size, which can depend on the input element (e.g., some transformations apply relative resizing whereas others output fixed-size elements). Scheduling data workers and reordering transformations are both complex decisions, which are a burden for ML practitioners to optimize.

We propose Pecan, an ML data preprocessing service that leverages these insights to alleviate data stalls while minimizing training costs. Pecan's *AutoPlacement* policy scales data preprocessing workers and places them across local and remote resources to minimize cost. The *AutoOrder* policy transparently reorders input pipeline transformations to maximize per-worker throughput. We show that the AutoPlacement and AutoOrder policies reduce total training costs by up to 60% compared to Cachew [22], a state-of-the-art disaggregated data preprocessing system and by 55% on average compared to collocated data preprocessing. Pecan's policies do not compromise on training time or model accuracy.

2 ML Input Data Preprocessing

Online vs. offline preprocessing. ML data preprocessing consists of two stages: offline and online. Offline preprocessing executes in batch processing frameworks (e.g., Apache Spark [75], Beam [1]) and applies transformations that require a view of the whole dataset, such as normalizing data and identifying outliers [54]. The outputs of offline preprocessing are persisted to storage and serve as input for online preprocessing. Online (or "last-mile") data preprocessing transforms data on-the-fly during training, with per-element transformations tailored for the ML model. Transformations include domain-specific operations, such as random image flipping, cropping, and rotation [2, 5, 40, 54, 59] as well as domain-agnostic operations, such as shuffling, batching, and casting. We focus on online preprocessing, as it is on the critical path of training, hence directly affecting training time and cost.

Frameworks for online preprocessing. Online preprocessing frameworks integrate directly with ML training platforms. For example, PyTorch DataLoader [60] preprocesses data on-the-fly for PyTorch training jobs, offering high parallelism via multi-processing. NVIDIA DALI integrates with both PyTorch and TensorFlow [56]. tf.data is TensorFlow's native data loader [54], which we extend in our work as it is widely used and supports offloading online preprocessing to remote CPUs. tf.data's Python API offers a library of transformations, which users compose and parametrize to define their input pipeline logic [19]. Internally, tf.data represents the input pipeline as a graph and optimizes the graph before execution by fusing and vectorizing operators. While executing the input pipeline, tf.data also autotunes its thread pool and memory buffer sizes to maximize throughput [54].

Disaggregated vs. collocated data preprocessing. Conventionally, ML frameworks collocate data preprocessing with training. However, as ML accelerator ingestion bandwidth continues to scale, data preprocessing can easily satu-

rate host CPU and DRAM on ML accelerator nodes and stall training. Some frameworks offer a disaggregated data preprocessing mode, which schedules preprocessing on remote CPU workers, enabling flexible scale-out per job. This approach is widely adopted in production at Meta and Google. Meta's internal Data Preprocessing (DPP) system offloads preprocessing to remote nodes [52, 78]. Google's tf.data service disaggregates tf.data workers from ML accelerator nodes and is available open source [5]. When deploying a disaggregated preprocessing service, practitioners can manually configure the number of remote workers, use hardware utilization-based autoscalers like Kubernetes Horizontal Pod Autoscaler [41] and AutoPilot [63], or scale workers based on application-specific metrics. Our work builds on Cachew [22], which autoscales remote tf.data service workers by monitoring and minimizing batch processing time in training jobs.

3 Cost Saving Opportunities & Challenges

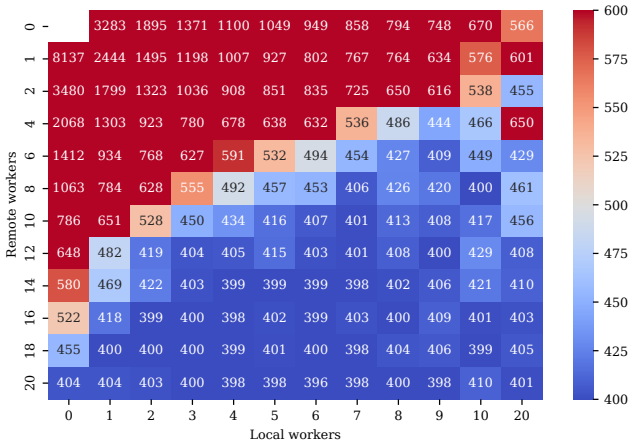
We explore two opportunities for cost-efficient data preprocessing: scheduling data preprocessing on a combination of remote CPUs and local ML accelerator hosts (§3.1) and reordering data transformations to maximize throughput (§3.2).

3.1 Exploiting Local Resources

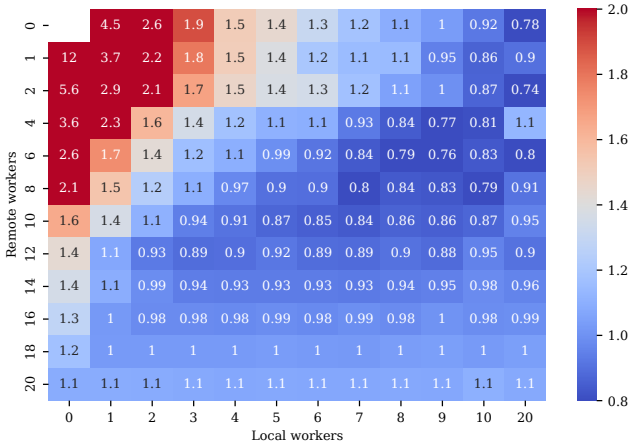
While scaling out data preprocessing to remote CPU workers mitigates data stalls [5], relying exclusively on remote servers leaves vast CPU and DRAM on accelerator hosts underutilized. For example, A100 GPUs in AWS EC2 have 12 CPU cores and 144 GB of DRAM per GPU [4]. In Google Cloud, 8-core TPUv2-8 accelerators come with 96 cores and 335 GB of DRAM [18]. When training ResNet50 using Cachew for remote preprocessing, we observe that host CPU utilization remains below 20% on a TPUv2-8 machine and remote servers contribute up to 62% of the overall training costs.

Hence, there is an opportunity to decrease costs by scheduling some data preprocessing on underutilized training node hosts, whose available CPU/DRAM resources are a sunk cost. tf.data service already provides a mechanism to instantiate a configurable number of *local workers*, which execute data preprocessing in the training client process on the ML accelerator node. Local workers can pass data to the model directly via method calls, unlike remote workers, which communicate with training clients over gRPC.

Tuning the number of local versus remote workers to optimize throughput and cost is challenging. Figure 2 shows the epoch time and cost (lower is better) when training a ResNet50 model on a TPUv2-8 VM with different combinations of remote and local tf.data service workers. Using only local workers does not eliminate input data stalls. The number of remote workers required to alleviate data stalls depends on the number of local workers used. Using multiple local workers is beneficial as a single local worker lacks the parallelism



(a) Epoch time (seconds).



(b) Epoch cost ($).

Figure 2: Epoch time and cost for ResNet50 model training on a TPUv2-8 with different remote and local data worker configurations.

needed to utilize all 96 CPU cores on the TPU host, even with software parallelism autotuning. However, we cannot arbitrarily increase the number of local workers because they compete for CPU and memory resources that are needed for other tasks on ML accelerator hosts, such as checkpointing, logging, loading data on accelerators, and network processing (including deserializing and decompressing requests and data from remote workers [5]). Analytically deriving the optimal number of remote and local workers is impractical as it requires accurately modeling the interference between the tasks running on ML accelerator hosts. This is complex due to the numerous shared hardware resources (CPU, memory, storage, NICs, buses, accelerators) and the complexity of OS scheduling algorithms. This challenge is not specific to tf.data; we observe similar challenges with PyTorch DataLoader. However, we observe that adding and removing data workers on the fly is relatively lightweight, so we propose an AutoPlace-

ment policy based on runtime metrics (§5.1) that iteratively autoscales and places data workers across local and remote nodes to minimize epoch time and cost.

3.2 Transformation Reordering

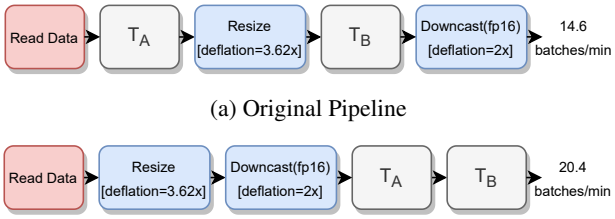
ML practitioners develop data preprocessing logic with model accuracy and robustness in mind [12, 13, 40, 48]. They rely on the preprocessing framework to optimize throughput [54, 62]. Current frameworks like tf.data and Plumber [43] apply static and dynamic optimizations (e.g., operator fusion) to increase throughput without altering the semantics of the input pipeline. We propose to further optimize throughput by relaxing the constraint of semantic equivalence for pipelines that anyway add randomness to input data.

In particular, we study how the order of transformations impacts compute requirements and per-worker throughput. Consider the example pipeline in Figure 3a, which applies various data augmentations represented by nodes T_A and T_B (e.g., T_A consists of cropping, flipping, rotating, and shearing) and two transformations that reduce the data volume: resizing elements to a fixed size and casting to `float16`. Executing the reordered pipeline in Figure 3b on the same `n2-standard-8` Google Cloud VM increases throughput by 1.4×. By applying transformations that reduce data volume early on in the pipeline, downstream data augmentations compute on smaller elements and can output results at higher throughput.

Optimizing throughput with transformation reordering is non-trivial for users, as it requires reasoning about how transformations impact data volume, which may depend on input data characteristics. For example, transformations that resize an element to a fixed size may inflate or deflate data, depending on the input size. Another potential concern is the impact reordering could have on model training dynamics. We find that since a key role of data preprocessing in many ML application domains is to add randomness (e.g., randomly flipping or cropping images) to train more robust and generalizable models [12, 13], relaxing commutativity among transformations in such pipelines does not disrupt convergence or noticeably impact accuracy. As not all transformations can be arbitrarily reordered, user hints can help identify transformations whose absolute or relative order must be preserved when these constraints cannot be automatically detected. In §5.2, we describe our policy for ordering transformations to maximize throughput while supporting user hints.

4 Related Work

Alleviating ML input data stalls. Our work is complementary to approaches that mitigate ML data stalls with more efficient data formats [42, 78], cache source data [45, 68], cache preprocessed data [11, 22, 47, 53], improve storage throughput for ML access patterns [23, 37, 38, 68], or combine clever sampling with caching [9, 17, 39, 76]. Revamper [47] and



(a) Original Pipeline

(b) Manually Reordered Pipeline

Figure 3: Impact of reordering on the input pipeline throughput. The pipeline is a simplified version of the ResNet50 input pipeline, showing how much transformations deflate the data.

data echoing [11] relax the requirement of identical results for the input pipeline by adding caching operators that increase throughput while reducing randomness. They show that transformation output caching can be sparingly applied without degrading model convergence and accuracy. Most similar to our work is FastFlow [66], which proposes leveraging local and remote workers for data preprocessing with a Smart Offloading policy that splits an input pipeline in one of three candidate locations. While FastFlow aims to minimize training time for a given fixed number of remote CPU nodes, we aim to minimize training time and *total cost* by minimizing the pool of remote workers needed while scaling local workers to eliminate data stalls. In §6.2, we show that our AutoPlacement approach achieves significantly lower training cost than FastFlow's policy of splitting and offloading data preprocessing to throttle the volume of data preprocessed by local workers. Finally, other systems, such as DALI [56] and TrainBox [58], offload preprocessing to specialized hardware, such as GPUs or FPGAs. This is a viable approach for data transformations that map easily to custom hardware operators, however, converting user-defined functions is often tricky.

Transformation reordering. Databases statically reorder (as well as prune and transform) query plans to minimize query latency [6, 31, 35, 64]. Adaptive query processing leverages query runtime signals and statistics to generate adaptive query plans with greater performance than statically generated plans [15, 21, 26]. Work in stream processing has focused on similar challenges of reordering transformations, oftentimes providing theoretical guarantees for the correctness of the transformed pipeline [27, 28, 49]. Such work does not capitalize on the unique characteristics of ML preprocessing, which allow relaxing commutativity constraints in (parts of) pipelines that randomly permute input data [44].

5 Pecan Design and Implementation

We propose Pecan, an open-source data preprocessing service built on top of tf.data service [5] and Cachew [22]. Pecan introduces two policies, AutoPlacement (§5.1) and AutoOrder (§5.2), to maximize preprocessing throughput and minimize ML training costs. Pecan's policies are platform-

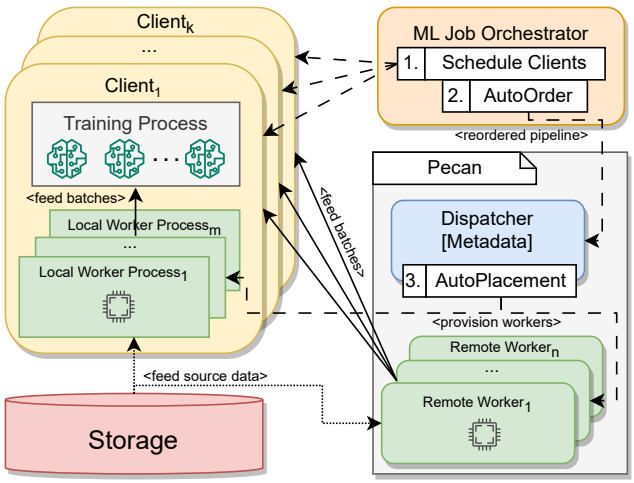


Figure 4: Pecan system architecture.

agnostic. We choose to implement them on top of tf.data service as the framework is widely used by ML practitioners and it provides high-performance mechanisms for local and remote data preprocessing. We also build on Cachew's autoscaling policy for tf.data service, which dynamically scales out remote data workers. Pecan's AutoPlacement policy consists of ∼1400 lines of code in the core C++ layer of tf.data while the AutoOrder policy consists of ∼1600 lines of code, predominantly in the Python layer.

System architecture. Figure 4 shows Pecan's system architecture. The practitioner uses the tf.data-based Python API to define their input pipeline logic and submit their pipeline to Pecan's ML Job Orchestrator. The Orchestrator analyzes the user's input pipeline and applies Pecan's AutoOrder policy to optimize the order of transformations for higher throughput. The Orchestrator then launches the user-specified number of training clients for the job with the appropriate hardware accelerator types (e.g., GPU or TPU) and registers the reordered input data pipeline with the Pecan dispatcher.

The dispatcher is an essential part of Pecan enabling multi-tenancy and moving policy execution and cluster management off the critical path of preprocessing. The dispatcher manages data preprocessing workers and implements Pecan's AutoPlacement policy, which decides how many data workers to use and on which nodes to schedule them. The dispatcher sends each worker a copy of the (re-ordered) input pipeline graph. Workers request dataset splits (i.e., locations of input dataset partitions) from the dispatcher, read the source data from storage, and preprocess it by applying input pipeline transformations. Pecan can dynamically add or remove workers as the dispatcher assigns dataset splits to workers. The dispatcher informs each training client of the IP addresses of remote CPU workers, allowing clients to fetch preprocessed data over the network via gRPC. Training clients also poll a local buffer to receive preprocessed data from local workers,

```
1 ds = tf.data.Dataset
2     .from_tensor_slices(["file1", ...])
3     .map(transform_1)
4     ...
5     .map(transform_N, keep_position=True)
6     .batch()
7 ds = ds.apply(distribute(dispatcher_IP))
8 for element in ds:
9     train(element)
```

Listing 1: Input pipeline example in Pecan.

which run within the same TensorFlow process and pass data via regular method calls instead of gRPC.

User API. Pecan builds on the easy-to-use API of tf.data service [19, 54]. Listing 1 shows an example pipeline that reads a source dataset (line 2) and applies a series of user-defined functions in a `map` transformation (lines 3-5). The pipeline groups its `Dataset` object into batches (line 6) and registers the pipeline for execution in Pecan via the `distribute` call (line 7). Finally, the code iterates over each batch and supplies it to the training logic (lines 8-9). The `keep_position` parameter is the only extension Pecan adds to the tf.data API. This flag is used to accept user hints for the AutoOrder policy, for transformations with strict ordering requirements. Pecan treats transformations with the `keep_position` flag set to `True` as barriers; the transformation maintains its position in the input pipeline and no transformations are moved across it.

5.1 AutoPlacement Policy

While each training job executes, Pecan's AutoPlacement policy scales and places data preprocessing workers across ML accelerator hosts (whose CPU and DRAM are a sunk cost) and an elastic pool of remote CPU servers (which cost extra but can speed up training by alleviating data stalls). To minimize cost, the policy aims to eliminate data stalls with the minimum number of remote servers. As highlighted in §3.1, this also requires carefully tuning the number of local workers. With too few, accelerator host resources are underutilized, whereas, with too many local workers, we observe contention for CPU cycles and memory bandwidth on accelerator hosts, which also receive data over the network from remote workers and need to load data to accelerators.

Algorithm 1 summarizes the AutoPlacement policy. The policy starts by addressing the primary objective: eliminating input data stalls. Like Cachew, we leverage the iterative nature of ML training jobs and monitor the *batch processing time* (`bpt`) metric for each training iteration. Gathering the `bpt` metric has negligible overhead, as it exploits tf.data's existing metric gathering logic. In lines 1-2, we apply Cachew's autoscaling policy, gradually adding remote workers until `bpt` plateaus. When `bpt` stops decreasing with additional remote workers, training throughput is bound by model training rather

Algorithm 1: AUTOPLACEMENT(T, $\mathcal{X}$)

```
1  ResetLocal();
2  bpt_conv ←CachewAutoscale(T);
3  count_local ←0; count_remote ← CountRemote();
4  converged← False;
5  repeat
6      start_local ←count_local; start_remote ←count_remote;
7      // Add local workers until bpt_new/bpt_conv - 1 ≥ T
8      count_local ←AddLocalUntilWorse(T, bpt_conv);
9      // Remove detrimental local worker if count_local > 0
10     count_local ←RemoveLocal(count_local);
11     // Remove remote workers until job cost degrades
12     count_remote ←RemoveRemoteUntilWorse(bpt_conv);
13     // Re-add last remote if its removal broke cost constraints
14     count_remote ← AddRemote();
15 until start_local == count_local ∧ start_remote == count_remote
```

than input data preprocessing. We denote the model-bound batch processing time as $\texttt{bpt}_{\texttt{conv}}$. We first scale remote workers because we assume remote CPU and DRAM can be scaled as much as needed to guarantee we reach $\texttt{bpt}_{\texttt{conv}}$. In contrast, local workers are limited to fixed CPU and DRAM resources available on ML accelerator hosts, and may not suffice.

The next step in the policy (lines 6-10) involves gradually adding local workers on ML accelerator hosts. In line 8, `AddLocalUntilWorse(*)` adds local workers until `bpt` starts to increase beyond a threshold T from $\texttt{bpt}_{\texttt{conv}}$. Batch processing time can increase due to resource contention between local workers and other tasks (e.g., network processing and data loading to the accelerator) on ML training nodes.

In the final step (lines 12-14), the policy removes as many remote workers as possible while maintaining similar batch time and optimizing for cost. *Batch processing cost* (`bpc`) is a function of batch time, the number of ML accelerator nodes (n_a), and the number of remote worker nodes (n_w):

$$\text{bpc} = \text{bpt}(c_a \cdot n_a + c_w \cdot n_w)$$

where c_a and c_w are the per-unit cost of ML accelerator and remote worker nodes, respectively. In line 12, `RemoveRemoteUntilWorse(*)` continually removes remote workers until batch processing cost increases. `bpc` can increase if removing a remote worker leads to a data preprocessing bottleneck that significantly increases `bpt`. If this occurs, the remote worker is added back (line 14).

The algorithm converges when the number of remote and local workers stabilizes (line 15). This indicates that the policy has identified the minimum required number of remote and local workers. Otherwise, the algorithm repeats another iteration of adding local workers and removing remote workers. This iterative process helps reach the optimal configuration as adding local workers gradually redistributes work to local resources, reducing the need for remote workers. In turn, removing remote workers frees up CPU cycles on the training node (due to less network request processing), potentially making adding more local workers viable. After convergence, runtime metrics are used to determine if workers need to be added or removed based on performance changes.

The AutoPlacement algorithm takes two parameters: a threshold parameter T (for `bpt` measurement comparison) and a window size of $\mathcal{X}$ batches (for `bpt` measurement averaging). We conduct a sensitivity study for these parameters in §6.4. We use $T = 0.03$ as it provides a good compromise between efficient scaling and noise mitigation, as also observed in other works [22, 66]. We set $\mathcal{X} = 500$ as it minimizes measurement noise and allows for low autoscaling convergence times relative to the total job time.

Fast worker removal. Pecan can leverage relaxed data visitation guarantees during downscaling phases to remove workers immediately, without waiting for them to finish processing their assigned data shards. Production ML workloads typically allow for relaxed data visitation as there is abundant data to train on and training is hence robust against dropped batches [5]. The user can choose to enable this feature in Pecan to speed up AutoPlacement policy convergence.

Multi-client setup. Pecan accommodates multi-client distributed training scenarios. The dispatcher averages clients' `bpt` measurements for the AutoPlacement policy. Pecan deploys local workers on clients in a round-robin fashion to avoid overprovisioning particular nodes, as this could halt the policy prematurely. We assume homogeneous hardware across clients, as this is standard practice. For heterogeneous hardware, local workers can be deployed on the workers with the highest ratio between CPU power and the number of already deployed workers.

Analytical approach. We considered using a purely analytical approach to derive the optimal number of remote and local workers. However, this requires modeling how local data preprocessing tasks and network processing tasks for remotely processed data batches compete for CPU and memory resources on ML accelerator nodes, which is non-trivial. In addition, training clients and remote workers can each run on heterogeneous nodes, which introduces additional complexity for purely analytical solutions. We found it more accurate to add and remove workers on the fly and rely on runtime metrics to determine the right data worker scale and placement for a particular ML workload and hardware deployment. However, in ongoing work, we are exploring an analytical approach for determining an initial number of remote workers, which can be combined with our current policy to fine-tune the final number of remote and local workers. The analytical component could use a simple linear model: $n_w = \mathbb{E}[t_{model}]/\mathbb{E}[t_{worker}]$, where t denotes throughput. $\mathbb{E}[t_{model}]$ is measured by testing model performance on immediately available data, and $\mathbb{E}[t_{worker}]$ is measured by testing preprocessing performance on a data subset. We expect this approach to accelerate policy convergence by starting with more than one worker.

5.2 AutoOrder Policy

The AutoOrder policy statically reorders transformations to increase per-worker preprocessing throughput. We refer to input pipeline transformations as *inflationary* if they increase data volume, such as image padding and one-hot encoding. We refer to transformations as *deflationary* if they decrease data volume, such as sampling, filtering, or cropping images. Transformations such as resizing an image to a particular size or casting an input to a different type can be inflationary or deflationary, depending on the input size or type. The AutoOrder policy opportunistically moves deflationary transformations upstream and inflationary transformations downstream, while respecting ordering constraints specified by users with the `keep_position` flag. The intuition for this reordering policy is to start by reducing the volume of data as much as possible, such that most transformations compute on smaller data elements and consume fewer CPU cycles. At the end of the pipeline, inflationary transformations can finally be applied.

Respecting ordering constraints. To optimize an input pipeline while respecting user-specified ordering constraints, Pecan's ML job orchestrator divides the pipeline into sections at each transformation with the `keep_position` flag set. These transformations act as barriers, allowing reordering only within sections but not across `keep_position` transformations. Pecan also preserves the original positions of two other types of transformations: (1) transformations that convert between numeric and non-numeric types (e.g., decoding an image), and (2) transformations that change the rank of the data (i.e., the number of data dimensions). Moving rank-changing transformations risks breaking dependencies. We did not observe any helpful reorderings of rank-changing transformations in practice. AutoOrder is expected to benefit preprocessing logic with loose dependency constraints, commonly found in audio-visual and multi-modal tasks. Conversely, pure text-based models typically have strict preprocessing constraints and may not significantly benefit from AutoOrder. Such models, however, are rarely input-bound.

Algorithm. Algorithm 2 summarizes the AutoOrder policy. The policy receives the ordered list of sections of the original input pipeline (`Sections`) and a list of fixed transformations (`FixedOps`). The algorithm iterates through each section (line 2) and keeps track of a prefix and suffix list (line 3). For each transformation in a section (line 5), if a transformation is deflationary, it is added to the beginning of the prefix list (lines 6-8)[1]. If the transformation is inflationary, it is added to the suffix list (lines 11-13). Otherwise (i.e., inflation factor is 1), the transformation is appended to the prefix list (lines 8-9). We create a new section by concatenating the `prefix` and `suffix` lists (line 15). We append the new section to a list of reordered sections (line 16) and later insert them between the fixed-position transformations to generate the reordered input pipeline (line 18). The overall time complexity is $O(n)$, where n is the number of transformations in the input pipeline. AutoOrder runs once before training and after the first epoch to validate inflation factors based on a full dataset pass.

Algorithm 2: AUTOORDER(Sections, FixedOps)

```
1  reorderedSections← EmptyList();
2  for section ∈ Sections do
3      prefix← EmptyList(); suffix← EmptyList();
4      for transformation ∈ section do
5          if transformation.InflationFactor() ≤ 1 then
6              if transformation.InflationFactor() < 1 then
7                  prefix.Prepend(transformation);
8              else
9                  prefix.Append(transformation);
10             end
11         else
12             suffix.Append(transformation);
13         end
14     end
15     reordered←CreateSection(prefix, suffix);
16     reorderedSections.Append(reordered);
17 end
18 return Interleave(reorderedSections, FixedOps);
```

Calculating inflation factors. To determine whether a transformation is inflationary or deflationary, Pecan uses static information captured by TensorFlow, namely the data types, data rank, and the size of each dimension. The inflation factor I_t of a transformation t is computed as $I_t = (c_{out} \cdot d_{out} \cdot \prod_{s_o \in S_{out}} s_o)/(c_{in} \cdot d_{in} \cdot \prod_{s_i \in S_{in}} s_i)$. Here, d_{in} and d_{out} represent the number of bits per atomic input and output element respectively (e.g., `tf.float16` is 16), and S_{in} and S_{out} are lists containing the size of each input and output data dimension respectively. Some transformations (e.g., `batch`, `filter`) may have a different number of input and output elements. To account for this, we condition I_t on c_{out} and c_{in}, i.e. the number of atomic output and input instances in t.

TensorFlow's statically inferred metadata suffices to calculate inflation factors for most transformations. For datasets with variable shape elements (e.g., ImageNet [14]) and relative sizing transformations like halving image resolutions, S_{in} and S_{out} are not statically known. For such scenarios, Pecan's ML job orchestrator (which implements the AutoOrder policy) executes the data pipeline for a small subset of the input dataset (e.g., 300 elements) to empirically estimate S_{in} and S_{out}. The orchestrator profiles input pipeline execution locally without involving model training. This approach is fast and does not impact model accuracy.

[1] One exception is that deflationary casts to custom data types like `tf.bfloat16` are moved to the end of the pipeline because most current CPUs do not support efficient computation on such data types [69].

Model	Input Pipeline
ResNet50	Decode + Crop → Flip → Rotate → Shear → Resize → Mean Subtract → Cast image(fp16) + One hot encode label → Batch
SimCLR	Duplicate + Cast(fp32) → Crop + Resize → Flip → Jitter → Blur → Clip → To Grayscale → Tile → Reshape → Batch
RetinaNet	Decode → Cast(fp32) + Normalize → Flip → LabelExtract (denormalize boxes, asign anchors, label anchors) → Cast(bf16) → Batch
ASRTransformer	Decode → AddText → Vectorize → Fourier Transform → Pad → SpecAugment → SpeedPerturb → PitchPerturb → Reverb → Batch

Table 1: Preprocessing logic in evaluation models. We use "+" between user-defined operators for which TensorFlow offers a high-performance fused implementation (we keep these transformations within the same `map`). Deflationary transformations are green, inflationary transformations are orange, and immovable transformations as described in §5.2 are blue.

6 Evaluation

We evaluate Pecan to answer the following questions:

- How much does AutoPlacement decrease the cost of data preprocessing and end-to-end model training?
- How much does AutoOrder decrease the cost of data preprocessing and end-to-end model training?
- What is the impact of AutoOrder on model accuracy?
- How do AutoPlacement and AutoOrder work together to improve end-to-end model training costs?

6.1 Methodology

Setup. We run our experiments on Google Cloud Platform (GCP) using TPUv2-8 and TPUv3-8 VMs, which come with 96 CPU cores and 335 GB of DRAM. We choose these accelerators as we received cloud credits for these VMs from Google's TPU Research Cloud [20]. We use `n2-standard-8` GCP VMs for Pecan's dispatcher and remote CPU servers (8 CPU cores, 32 GB DRAM each). We schedule one remote worker per `n2-standard-8` VM yielding a 1:1 mapping between remote workers and remote CPU servers. We deploy and orchestrate VMs with Kubernetes. The TPU training node runs the training client and the ML job orchestrator for a training job. The network bandwidth between our GCP resources is at least 16 Gb/s. We store and read our datasets from Google Cloud Storage buckets. We enable tf.data autotuning in all experiments, as is customary to maximize software parallelism. We do not set `keep_position=True` in any of our pipelines. Unless specified otherwise, results are averaged across five runs, and we do not use the fast worker removal feature.

Workloads. We evaluate Pecan on four popular ML models and their input pipelines: ResNet50 [25], SimCLR [7], RetinaNet [50], and ASRTrans [16, 67]. We use the open-source TF Model Garden implementations of ResNet and RetinaNet [29], the Google Research implementation of SimCLR [8], and the keras-io implementation of ASRTrans [55] using both the model and its data preprocessing logic. For ResNet50, we also include rotation and shearing transformations in the input pipeline, as used in prior works, such as RandAugment [13] and AutoAugment [12]. For ASRTrans we include Spec augmentation, reverbation, and speed and pitch perturbation, commonly used in works such as SpecAugment [57] and variants of DeepSpeech [24]. Table 1 shows the transformations in each input pipeline. For each pipeline, we spread user-defined preprocessing logic across a sequence of `map` transformations, each parameterized with an individual user-defined function (e.g., image cropping), to enable the AutoOrder policy to reorder transformations at fine granularity. We train the ResNet50 and SimCLR models on ImageNet [14]. We train the RetinaNet model on MS-COCO [51] and the ASRTrans model on the LJ-Speech dataset [33]. To show the benefits of our policies across a variety of hardware, we run experiments on TPUv2-8 and TPUv3-8. We evaluate ResNet50, SimCLR, and RetinaNet on TPUv2-8. We use TPUv3-8 for experiments on ASRTrans, and re-evaluate ResNet50, to provide a reference point for how collocated training, our policies, and related work perform across accelerators.

Metrics. We measure per-epoch training throughput and cost. We break down the total cost to show the portion attributed to ML accelerator VMs and remote data worker VMs. The TPU VM portion of the cost also serves as an indicator of end-to-end training time as we only lease ML accelerators for as long as the training job lasts and we do not scale training nodes up or down during experiments. Hence, higher TPU costs imply longer training time (e.g., due to input bottlenecks). We use the costs of Google Cloud resources in the `europe-west4-a` zone in April 2023 [18]. A TPUv2-8 VM costs 4.96$/hr and TPUv3-8 VM costs 8.8$/hr. The `n2-standard-8` VM (used for remote CPU workers) costs 0.42$/hr. Workers and clients are deployed in the same region. The epoch cost is $C = t_{epoch} \cdot (c_a \cdot n_a + c_w \cdot n_w)$ where t_{epoch} is the epoch execution time, c_a and c_w are the cost per unit time of a training node and remote worker node, respectively, and n_a and n_w represent the current number of training nodes and remote worker nodes, respectively. We report costs for the configuration of workers that Pecan's policies converge to. In-region data transfers are not charged in GCP, however, we also quantify the volume of data transferred over the network for data preprocessing with Pecan compared to other systems.

Baselines. We compare Pecan to remote data preprocessing with Cachew [22], local data preprocessing with tf.data [54], and hybrid data preprocessing with FastFlow's Smart Offloading policy [66]. The tf.data baseline shows the performance of local data preprocessing mechanisms, which the other systems in our evaluation build on. Cachew is a state-of-the-art service built on top of tf.data service that autoscales remote workers to maximize ML training throughput. FastFlow spreads

data preprocessing across local tf.data service workers and a user-specified fixed number of remote workers. For FastFlow, we use a 14:1 ratio of CPU cores to accelerator cores, as originally proposed by the authors [66]. Hence we provision 112 CPU cores per 8-core TPUv2-8 accelerator. 96 CPU cores are local on the TPU VM and for the other 16, we use two `n2-standard-8` VMs as remote workers. We use two local workers for the FastFlow baseline, as specified in the FastFlow GitHub repository [65]. We do not include a direct comparison to Meta's Data PreProcessing (DPP) system [78] as it is closed source and not available to us. Based on the system description, which depicts offloading data preprocessing entirely to remote workers, we expect DPP's end-to-end performance and cost to be similar to the Cachew baseline.

6.2 AutoPlacement Evaluation

Figure 6 compares the total training cost per epoch with different data preprocessing systems. The blue bars show epoch costs when using tf.data to preprocess data locally on TPU host resources. Data preprocessing saturates local TPU host CPU and DRAM resources for the ResNet50_v2-8, SimCLR , ASRTrans, and ResNet50_v3-8 workloads, causing input data stalls, prolonging training time, and incurring high costs.

Cachew, FastFlow, and Pecan achieve lower costs by alleviating input data bottlenecks with remote workers. Cachew fully alleviates input data stalls, but does so by relying, on average, on 19 remote data workers for ResNet50_v2-8, 12 for SimCLR, 2.6 for RetinaNet, 4 for ASRTrans, and 30 for ResNet50_v3-8. Compared to Cachew, Pecan's AutoPlacement policy reduces training costs by 44% for ResNet50_v2-8, by alleviating data stalls with 74% less remote workers (5 vs. 19). AutoPlacement reduces SimCLR training epoch cost by 33% compared to Cachew, alleviating input data stalls with 58% less remote resources (5 vs. 12). For ASRTrans we observe a 14% cost reduction, fully removing the 4 remote workers by deploying several local workers to capture all available client resources. For ResNet50_v3-8 we obtain a 14% total cost reduction with 9% fewer remote workers (27.2 vs. 30). We observe a 5.2% improvement in training time for SimCLR and 8% for ResNet50_v3-8. For SimCLR, Cachew's autoscaling struggles with probabilistic transformations. For ResNet50_v3-8, Cachew cannot fully eliminate the input bottleneck. RetinaNet's less compute-intensive pipeline allows TPU VMs to handle local data preprocessing without stalling. Pecan's AutoPlacement correctly determines that local-only preprocessing is optimal for RetinaNet, reducing training costs by 17% compared to Cachew (0.2 vs. 2.6 remote workers). Overall, AutoPlacement yields 69% preprocessing cost savings and 24% total cost savings compared to Cachew.

Compared to FastFlow, AutoPlacement reduces training costs by 76%, 23%, 27%, 19%, and 72% for ResNet50_v2-8, SimCLR, RetinaNet, ASRTrans, and ResNet50_v3-8, respectively. FastFlow's fixed remote worker configuration does not fully alleviate stalls, resulting in job times that are 5.1×, 1.6×, 1.2×, 1.1×, and 7.4× higher than with AutoPlacement.

Model	GiB Transferred		
	Cachew	Pecan AutoPlacement	Pecan AutoPlacement + AutoOrder
ResNet50 (v2-8)	18	8.91 (-50.5%)	**1.33 (-92.64%)**
SimCLR	36.13	19.42 (-46.25%)	**9.92 (-72.54%)**
RetinaNet	155.15	8.51 (-94.47%)	**0 (-100%)**
ASRTrans	42.37	0 (-100%)	**0 (-100%)**
ResNet50 (v3-8)	17.92	13.96 (-22.10%)	**11 (-38.61%)**

Table 2: Amount of data transferred over the network from remote workers to training clients for 500 training batches.

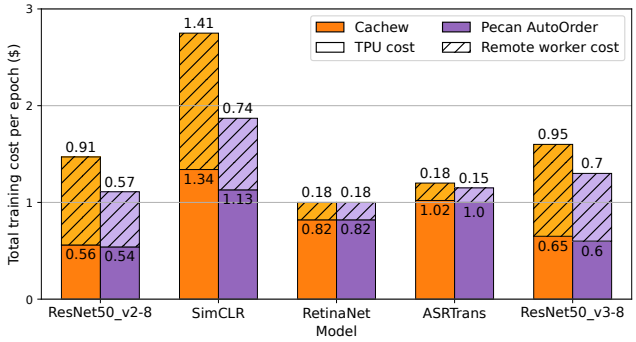


Figure 5: Cost of an epoch for all five workloads with and without the AutoOrder policy in a disaggregated setting.

Table 2 shows the network data transfer savings achieved by AutoPlacement. By distributing preprocessing across local and remote nodes, AutoPlacement reduces the total data volume sent by remote workers to training clients by 63% compared to fully disaggregated data processing with Cachew.

6.3 AutoOrder Evaluation

We apply the AutoOrder policy on each input data pipeline in Table 1. In the ResNet50 pipeline, AutoOrder moves the image Resize transformation in front of all other data augmentations (flip, rotate, shear). In SimCLR, AutoOrder moves the To Grayscale transformation in front of all the other augmentations (flip, jitter, blur, and value clipping). In RetinaNet, AutoOrder moves the Cast+Normalize transformation downstream immediately after the random flip transformation, while in ASRTrans, it moves the Pad transformation after all the augmentations (SpecAugment, SpeedPerturb, PitchPerturb, Reverb). We measure the training throughput and cost benefits of the AutoOrder policy and validate that transformation reordering does not reduce model quality for our workloads.

Impact on Training Throughput and Cost. We apply the AutoOrder policy in a disaggregated data preprocessing setup, using Cachew's autoscaling policy to determine the number of remote workers. Figure 5 shows the epoch cost benefits compared to executing the original pipeline with Cachew. The AutoOrder policy reduces epoch cost by 24% and remote worker costs by 37% for ResNet50_v2-8. For SimCLR, Auto-

Model	Original	Reordered	Delta
ResNet50	73.94% (±0.34%)	74.32% (±0.21%)	**+0.38%**
SimCLR	56.06% (±0.11%)	55.97% (±0.03%)	**-0.09%**
RetinaNet	33.08% (±0.09%)	33.21% (±0.13%)	**+0.13%**
ASRTrans	41.41% (±8.69%)	43.72% (±8.00%)	**+2.31%**

Table 3: AutoOrder impact on Top-1 accuracy (ResNet, SimCLR), mAP (RetinaNet), and WER (ASRTrans).

Order reduces epoch cost by 32% and remote worker costs by 48%. The AutoOrder policy does not significantly impact the RetinaNet input pipeline, due to the efficiency of the original pipeline for our hardware setup. AutoOrder only changes the intermediate memory footprint, not the total computation, and RetinaNet is not memory bottlenecked on our hardware. For ASRTrans, AutoOrder reduces epoch cost by 4% and remote worker cost by 17%. For ResNet50_v3-8, it reduces epoch cost by 19% and remote worker cost by 26%.

Impact on Trained Model Quality. To measure the impact of transformation reordering on model quality, we compare the average validation accuracy after training each model to convergence with and without applying AutoOrder to its input pipeline. ResNet50 and SimCLR are evaluated using Top-1 accuracy and RetinaNet uses mean Average Precision (mAP). In both cases, higher values are better (ideally 100%). ASRTrans is evaluated via Word Error Rate (WER), where lower values are better (ideally 0%). Table 3 shows that in all cases, AutoOrder has negligible impact on model accuracy, with ResNet50 and RetinaNet accuracy benefiting slightly and SimCLR accuracy decreasing slightly (0.09%). The ASRTrans WER increases by 2.31%, however, randomness plays a considerably bigger role here with a standard deviation of around 8%. On average, for our workloads, the AutoOrder policy has a ±0.73% effect on the final validation accuracy. Intuitively, model quality is not significantly impacted by transformation reordering for these vision and speech models because the transformations are designed to add randomness to the input data [12, 13] and the exact order in which they are applied (e.g., crop then flip or flip then crop) is not critical. In §7 we motivate the need for a broader empirical and theoretical exploration of model quality guarantees for other ML application domains [46, 71, 73].

6.4 End-to-End Pecan Evaluation

We evaluate Pecan's AutoPlacement and AutoOrder policies together to understand their combined benefit.

Impact on Training Time and Cost. Figure 6 shows that Pecan minimizes training cost across workloads. For ResNet50_v2-8, Pecan successfully eliminates the input bottleneck, reducing epoch time by 88% compared to collocated data preprocessing (from 3302 seconds to 395 seconds). Pecan achieves the optimized epoch time at 60% lower cost than Cachew and reduces preprocessing costs by 95%. For

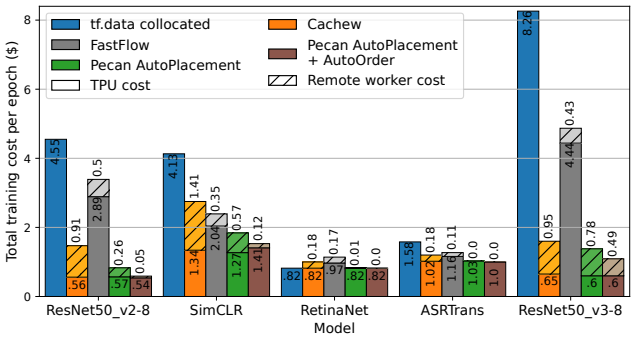


Figure 6: Training cost benefits of Pecan.

SimCLR, Pecan reduces training time by 66% compared to collocated data preprocessing and cost by 44% compared to Cachew. Pecan barely requires remote workers to alleviate data stalls during SimCLR training, thanks to the AutoOrder policy's input pipeline optimizations. SimCLR experiences a slight (5%) increase in training time due to the AutoPlacement policy's cost-based removal of remote workers. For RetinaNet, Pecan correctly identifies that disaggregated data preprocessing is not needed due to abundant local resources on the TPU VM. Pecan hence relies only on local workers and achieves the same optimal performance and cost as collocated data preprocessing. For ASRTrans, Pecan reduces training time by 37% relative to a collocated deployment and entirely removes the need for preprocessing workers, compared to Cachew. Unlike RetinaNet, ASRTrans is not naturally model-bound and requires the AutoPlacement policy to deploy several local workers to eliminate the input bottleneck. For ResNet50_v3-8, Pecan produces a 14× epoch time speedup compared to a collocated deployment. Relative to Cachew, there is a 7.7% epoch time improvement (as Cachew cannot entirely remove the input bottleneck) and 48% fewer preprocessing resources. Overall, Pecan significantly reduces the number of remote workers compared to Cachew: from 19 to 1 for ResNet50_v2-8, 12 to 1 for SimCLR, 2.6 to 0 for RetinaNet, 3.8 to 0 for ASRTrans, and 30 to 17 for ResNet50_v3-8. The last column in Table 2 also shows Pecan's data transfers between workers and clients relative to Cachew. Pecan reduces network traffic by 93%, 73%, and 39% for ResNet50_v2-8, SimCLR, and ResNet50_v3-8 respectively. For RetinaNet and ASRTrans Pecan eliminates network traffic between data workers and training clients as it does not rely on remote workers to alleviate data preprocessing stalls. Pecan converges to 9, 5.4, 3.4, 7.4, and 9 local workers on average for ResNet50_v2-8, SimCLR, RetinaNet, ASRTrans, and ResNet50_v3-8.

Policy Execution Timeline. Figure 7 shows how Pecan's policies work together over time during the first 18k steps of ResNet50_v2-8 training. For this trace, we use the fast worker removal feature. Before model training begins, Pecan's ML job orchestrator runs a few trial batches of the input pipeline

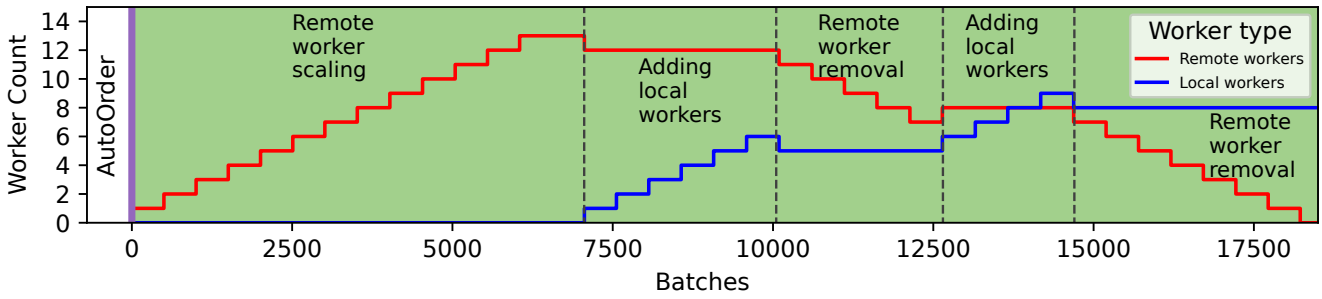


Figure 7: Pecan's workflow for the first few epochs of ResNet50_v2-8 training. Pecan first applies the AutoOrder policy, followed by the AutoPlacement policy, which autoscales remote workers, adds local workers, then removes unneeded remote workers.

to collect transformation inflation factors for the AutoOrder policy. The orchestrator then applies the reordering recommended by the AutoOrder policy, producing a new input data pipeline graph which it registers with the Pecan dispatcher. Next, the ML training job starts to execute and Pecan monitors batch times (`bpt`) to apply its AutoPlacement policy. The first stage of the policy scales out remote workers to eliminate data stalls. For the ResNet50_v2-8 example, this converges to 12 remote workers around the 7kth minibatch, after the superfluous 13th remote worker is successfully removed. Next, the AutoPlacement policy begins adding local workers until it notices a performance degradation in the `bpt` metric, converging to 5 local workers around the 10kth step. Any additional local workers would introduce resource contention on the client nodes, slowing down training. The algorithm resumes, next removing remote workers, thus applying more pressure on the local and remaining remote workers, and finding the ideal trade-off point between the new local and existing remote workers. At the 12.5kth step, the iteration converges to 8 remote workers. The algorithm executes a second and final iteration, first adding local workers, then removing remote workers, converging around the 18kth step with 8 local workers and 0 remote workers. The trace shows the complex, non-linear trade-offs between local and remote workers, first trading off 4 remote workers for 5 local workers, and later 8 remote workers for 3 local workers.

Policy Convergence Time. Pecan's AutoOrder policy requires executing a small number of minibatches to collect inflation factor statistics, which takes 5s, 10s, 4s, and 36s for ResNet50, SimCLR, RetinaNet, and ASRTrans respectively. In our experiments with ResNet50, SimCLR, and RetinaNet, the AutoPlacement policy converges within the first five training epochs, which is negligible compared to the numerous epochs that are typically used to train such models [7, 25]. In Figure 7 Pecan reaches an optimal training time at approximately 6k steps (2% of total job time) when the remote workers initially converge. We also notice that Pecan's AutoOrder policy speeds up the remote worker autoscaling convergence time compared to Cachew. By increasing per-worker preprocessing throughput, AutoOrder decreases the number of required remote workers and hence reduces the time needed to alleviate input data stalls by 47%, 15%, 5%, 28%, and 32% compared to Cachew for the ResNet50_v2-8, SimCLR, RetinaNet, ASRTrans, and ResNet50_v3-8 workloads.

Scaling Threshold (T)	**Batch Sampling Window (X)**					
	100		**500**		**1000**	
0.01	$0.89	$0.96	$0.59	$3.48	$0.61	$5.01
0.03	$0.78	$1.05	*$0.59*	*$3.34*	$0.64	$4.75
0.07	$0.73	$0.87	$0.66	$2.95	$0.59	$4.69

Table 4: AutoPlacement sensitivity study on ResNet50_v2-8 relative to X and T parameters. Left cell (blue) in each configuration is epoch cost; right cell (orange) is convergence cost.

Sensitivity Study. Table 4 shows the impact of changing parameters X (batch sampling window) and T (scaling threshold) (§5.1) on the cost per epoch, and the AutoPlacement policy's convergence cost for ResNet50_v2-8. Greater values for X mean lower noise levels in `bpt` measurements, reducing the risk of falsely converging to suboptimal deployments due to noise, and thus allowing for the more desirable, lower values of T. This improves the quality of AutoPlacement decisions, leading to low epoch costs. The trade-off is greater policy convergence times and costs. Lower values for X mean smaller sampling periods, which lead to noise in the `bpt` measurements, generally requiring greater values for T to ensure scaling only happens when significant benefits are noticed in the measurements. The advantage is low convergence times and costs but can lead to suboptimal AutoPlacement decisions that increase epoch costs. We choose the sweet spot, $X = 500$ and $T = 0.03$, which yields 1 remote and 9 local workers with an epoch cost of \$0.59 and a convergence cost of \$3.34. In all the tested deployments Pecan removes data stalls.

Optimality. We confirmed that the policy decisions are throughput-optimal, as all models in our evaluation are no longer input-bound when using Pecan. Ensuring cost optimality is more challenging, as it depends on runtime conditions such as network performance and tail latency.

7 Discussion

Our work on optimizing data preprocessing to minimize the cost of ML training opens up several future directions.

LLMs and multimodal workloads. Pecan targets domains with high preprocessing demands (e.g., audio-visual), where input bottlenecks can occur, and optimizing the number of local and remote workers is challenging. In contrast, text-based NLP models like LLMs are typically not input-bound due to lightweight online preprocessing or due to delegating most preprocessing to the offline stage. Local preprocessing on training nodes is usually sufficient in such cases, as confirmed by our experiments. However, an important future challenge for efficient ML preprocessing lies in multimodal workloads [30, 61]. In such settings, preprocessing remains critical for job performance, as data from various domains will have distinct preprocessing needs and will be used by models with diverse compute requirements. We anticipate significant benefits for multimodal jobs from systems like Pecan, that mitigate input bottlenecks.

Cross-region ML. Optimizing the placement of data preprocessing workers is especially useful when ML models need to be trained on input data that resides in a different geographical region. Cross-region scenarios often arise in practice [34, 74]. Copying data to the ML training job region might be restricted by data ownership or legal constraints. Scheduling ML training in the source data region may be limited by ML accelerator availability or per-region cloud quotas. Our initial experiments in cross-region settings show that expensive cross-region data transfers account for most of the training job cost (>90% for ResNet50_v2-8 on ImageNet). In cross-region deployments, AutoPlacement involves distributing workers across local training node resources, placing "close-remote" workers in the ML accelerator region, and "far-remote" workers in the dataset storage region.

Pipeline splitting. An additional way of splitting data preprocessing across local and remote resources is to split the input pipeline, such that specific transformations execute only locally and others only remotely. For example, FastFlow [66] selects among three generic split locations in a pipeline. As future work, we plan to explore how the AutoOrder and AutoPlacement policies can be co-designed with an input pipeline splitting policy to optimize which transformations should execute on remote versus local resources. For example, the AutoOrder policy can optimize for a split location that minimizes the amount of data sent over the network. This can further minimize costs when combined with AutoPlacement.

Reordering opportunities. Integrating AutoOrder into a compiler can enhance synergy with other optimizations and compilation phases. Several ML data preprocessing systems, including Cachew [22], Revamper [47], and Plumber [43], automatically cache data transformation outputs to save compute power and improve throughput. The challenge with caching outputs from prior epochs is that it makes the input pipeline deterministic, potentially affecting model quality for pipelines relying on data randomness. Pecan's AutoOrder policy can place deterministic, compute-intensive transformations earlier in the input pipeline to maximize caching opportunities. Orthogonal to caching AutoOrder can be designed to reorder pipelines to synergize with existing static graph optimization passes, including fusing contiguous `map`, `filter`, and `batch`, to reduce data movement and increase throughput [54, 77].

AutoOrder model quality analysis. We empirically showed that data transformation reordering with relaxed commutativity increases preprocessing throughput while maintaining high model quality for image recognition, object detection, and speech recognition (§6.3). A common characteristic of our tested input pipelines is the use of random transformations to improve model generalization [12, 13]. However, a broader study is needed to understand the general impact of data transformation reordering. A theoretical framework to compare preprocessed data distributions of original and AutoOrder pipelines and their impact on model quality, akin to Agarwal et al.'s for convergence guarantees in the context of cached preprocessed data [3], would be valuable.

8 Conclusion

We propose Pecan, a system that introduces the AutoPlacement and AutoOrder policies to alleviate input data preprocessing bottlenecks during ML training while minimizing cost. AutoPlacement leverages the flexibility of disaggregated data preprocessing to schedule data workers across both host training node resources and the bare minimum number of remote CPU servers, leading on average to 68% lower preprocessing costs compared to fully remote data preprocessing. Pecan's AutoOrder policy further reduces costs by reordering data transformations to increase per-worker throughput. Together, Pecan's AutoPlacement and AutoOrder policies can reduce data preprocessing costs by 87% on average and total training costs by up to 60% compared to Cachew. Compared to collocated deployments, Pecan reduces total training cost by 55% on average using only 14% of the remote workers required by Cachew.

Availability

Pecan is available at https://github.com/eth-easl/cachew/tree/pecan.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable feedback. We are grateful for access to the Google TPU Research Cloud. This work is partially supported by the Swiss National Science Foundation (Project Number 200021_204620) and a Meta Research Award.

References

[1] Apache Beam: An advanced unified programming model. https://beam.apache.org/, 2020.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, 2016.

[3] Naman Agarwal, Rohan Anil, Tomer Koren, Kunal Talwar, and Cyril Zhang. Stochastic optimization with laggard data pipelines. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, pages 10282–10293, 2020.

[4] Amazon Web Services. AWS EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, April 2023.

[5] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. tf.data service: A case for disaggregating ml input data processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 358–375, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.

[7] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International Conference on Machine Learning*, pages 1597–1607. PMLR, 2020.

[8] Ting Chen, Saurabh Saxena, and William Falcon. SimCLR. https://github.com/google-research/simclr, 2020.

[9] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Sun Xian-He, and Gang Chen. iCache: An importance-sampling-informed cache for accelerating I/O-bound DNN model training. In *IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2023.

[10] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jinkun Geng, Wei Bai, Jianping Wu, and Yongqiang Xiong. Accelerating end-to-end deep learning workflow with codesign of data preprocessing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1802–1814, 2020.

[11] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

[12] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. AutoAugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

[13] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition Workshops*, pages 702–703, 2020.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

[15] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[16] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888. IEEE, 2018.

[17] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[18] Google GCP. TPU VM documentation. https://cloud.google.com/tpu/docs/regions-zones, February 2023.

[19] Google. tf.Data Documentation. https://www.tensorflow.org/guide/data, February 2023.

[20] Google. TPU Research Cloud. https://sites.research.google/trc/about/, January 2024.

[21] Anastasios Gounaris, Norman W Paton, Alvaro AA Fernandes, and Rizos Sakellariou. Adaptive query processing: A survey. In *Advances in Databases: 19th British National Conference on Databases, BNCOD 19 Sheffield, UK, July 17–19, 2002 Proceedings 19*, pages 11–25. Springer, 2002.

[22] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[23] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Li Yi, Yu Ding, Guihai Chen, and Yihua Huang. Fluid: Dataset abstraction and elastic acceleration for cloud-native deep learning training jobs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2182–2195, 2022.

[24] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[26] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000.

[27] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)*, 53(2):1–37, 2020.

[28] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):1–34, 2014.

[29] Yu Hongkun, Chen Chen, Du Xianzhi, Li Yeqing, Rashwan Abdullah, Hou Le, Jin Pengchong, Yang Fan, Liu Frederick, Kim Jaeyoun, and Li Jing. TensorFlow Model Garden. https://github.com/tensorflow/models, 2020.

[30] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. DISTMM: Accelerating distributed multimodal model training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1157–1171, 2024.

[31] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[32] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *SIGMOD '22: International Conference on Management of Data*, 2022.

[33] Keith Ito and Linda Johnson. The LJ Speech Dataset. https://keithito.com/LJ-Speech-Dataset/, 2017.

[34] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[35] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

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

[37] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[38] Awais Khan, Arnab K Paul, Christopher Zimmer, Sarp Oral, Sajal Dash, Scott Atchley, and Feiyi Wang. Hvac: Removing i/o bottleneck for large-scale deep learning applications. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 324–335. IEEE, 2022.

[39] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. SHADE: Enable fundamental cacheability for distributed deep learning training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 135–152, 2023.

[40] Sotiris B Kotsiantis, Dimitris Kanellopoulos, and Panagiotis E Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[41] Kubernetes. Kubernetes Horizontal Pod Autoscaler Documentation. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, February 2023.

[42] Michael Kuchnik, George Amvrosiadis, and Virginia Smith. Progressive compressed records: Taking a byte out of deep learning data. *Proceedings of the VLDB Endowment*, 14(11), 2021.

[43] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. *Proceedings of Machine Learning and Systems*, 4:33–51, 2022.

[44] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[45] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, 2020.

[46] Elnaz Lashgari, Dehua Liang, and Uri Maoz. Data augmentation for deep-learning-based electroencephalography. *Journal of Neuroscience Methods*, 346:108885, 2020.

[47] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyung-Geun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *USENIX 2021 Annual Technical Conference (USENIX ATC 21)*, pages 537–550, 2021.

[48] Joseph Lemley, Shabab Bazrafkan, and Peter Corcoran. Smart augmentation learning an optimal data augmentation strategy. *IEEE Access*, 5:5858–5869, 2017.

[49] Katerina Lepenioti, Alexandros Bousdekis, Dimitris Apostolou, and Gregoris Mentzas. Prescriptive analytics: Literature review and research challenges. *International Journal of Information Management*, 50:57–70, 2020.

[50] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2980–2988, 2017.

[51] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[52] Meta. Scaling data ingestion for machine learning training at Meta. https://engineering.fb.com/2022/09/19/ml-applications/data-ingestion-machine-learning-training-meta/, 2022.

[53] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment*, 14(5):771–784, 2021.

[54] Derek G Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *Proceedings of the VLDB Endowment*, 14(12):2945–2958, 2021.

[55] Apoorv Nandan. Transformer ASR. https://github.com/keras-team/keras-io/blob/master/examples/audio/transformer_asr.py, 2021.

[56] Nvidia. Nvidia DALI Documentation. https://docs.nvidia.com/deeplearning/dali/user-guide/docs/, February 2023.

[57] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. SpecAugment: A simple data augmentation method for automatic speech recognition. *Proc. Interspeech 2019*, pages 2613–2617, 2019.

[58] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: An extreme-scale neural network training server architecture by systematically balancing operations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 825–838. IEEE, 2020.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[60] PyTorch. PyTorch DataLoader. https://pytorch.org/docs/stable/data.html, February 2023.

[61] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

[62] Jan S Rellermeyer, Sobhan Omranian Khorasani, Dan Graur, and Apourva Parthasarathy. The coming age of pervasive data processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 58–65. IEEE, 2019.

[63] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at Google scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[64] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[65] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. FastFlow. https://github.com/SamsungLabs/FastFlow, 2023.

[66] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. FastFlow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proceedings of the VLDB Endowment*, 16(5):1086–1099, 2023.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[68] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. DIESEL: A dataset-based distributed storage and caching system for large-scale deep learning training. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, 2020.

[69] Wei Wang and Niranjan Hasabnis. Distributed MLPerf ResNet50 training on Intel Xeon architectures with TensorFlow. In *The International Conference on High Performance Computing in Asia-Pacific Region Companion*, pages 29–35, 2021.

[70] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[71] Xing Wu, Shangwen Lv, Liangjun Zang, Jizhong Han, and Songlin Hu. Conditional BERT contextual augmentation. In *Computational Science–ICCS 2019: 19th International Conference, Faro, Portugal, June 12–14, 2019, Proceedings, Part IV 19*, pages 84–95. Springer, 2019.

[72] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[73] Ziang Xie, Guillaume Genthial, Stanley Xie, Andrew Y Ng, and Dan Jurafsky. Noising and denoising natural language: Diverse backtranslation for grammar correction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 619–628, 2018.

[74] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[75] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of HotCloud*, 2010.

[76] Xiao Zeng, Ming Yan, and Mi Zhang. Mercury: Efficient on-device distributed DNN training via stochastic importance sampling. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 29–41, 2021.

[77] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. cedar: Composable and optimized machine learning input data pipelines. *arXiv preprint arXiv:2401.08895*, 2024.

[78] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, 2022.

A Artifact Appendix

Abstract

The artifact consists of the source code of Pecan, the Pecan client binaries, as well as scripts for building wheel files and Docker images. We also provide reference scripts for deploying GCE VMs for evaluation and for running some representative experiments. For concrete instructions on how to set up and run experiments, please see the `pecan-experiments` repository.

Scope

The artifact evaluation (AE) focuses on reproducing key experiments and their respective results, demonstrating how the main contributions of Pecan work. To this extent, the AE focuses on reproducing and exemplifying the combined benefits of the **AutoPlacement** and **AutoOrder** policies on a job's performance. We focus on reproducing this for the **ResNet** and **RetinaNet** models on TPUv2-8 VMs. Overall, the AE would produce a part of Figure 6: the *collocated* (blue), *Cachew* (orange), and *Pecan* (brown) bars for the **ResNet** and **RetinaNet** on TPUv2-8 VMs. This should prove the reproducibility of our work.

Contents

The artifact with the components listed below is available at https://github.com/eth-easl/pecan-experiments.

- System to deploy: Pecan service (dispatcher, input data workers, remote cache cluster)
- Algorithms to evaluate: Pecan's AutoPlacement and AutoOrder policies
- Workloads to run:
  - Figure 6's ResNet TPUv2-8 group: ResNet50 model and its open-source canonical input pipeline for the collocated – blue bar –, Cachew – orange bar –, and Pecan (AutoPlacement and AutoOrder) – brown bar –
  - Figure 6's RetinaNet TPUv2-8 group: RetinaNet model and its open-source canonical input pipeline for the collocated – blue bar –, Cachew – orange bar –, and Pecan (AutoPlacement and AutoOrder) – brown bar –
- Binary: Pecan Docker image for workers and dispatcher, Pecan wheel file for client
- Models: ResNet50 and RetinaNet and their canonical input pipelines
- Datasets: ImageNet 2012 and MS COCO (stored in Google Cloud Storage buckets)
- Output: Text-based logs and plots to compare with figures in the paper and reference results provided in our repository.

Hosting

Our artifacts are all publicly available:

- Code: https://github.com/eth-easl/cachew/tree/pecan (branch `pecan`)
- Artifact evaluation scripts: https://github.com/eth-easl/pecan-experiments (branch `main`)
- Pecan TPU-compatible binary: gs://easl-atc24-ae-files/tensorflow-2.8.0-cp38-cp38-linux_x86_64.whl
- Zenodo DOI: 10.5281/zenodo.11477795

Requirements

Hardware dependencies: The experiments require a cluster of x86 CPU servers with hardware virtualization support. We recommend (and our scripts assume that you are) conducting experiments on GCP. Client nodes are assumed to be TPUv2-8 VMs. Worker nodes are assumed to be n2-standard-8 VMs.

Software dependencies: Our scripts use the gcloud CLI tool. This tool is a prerequisite for setting up VMs and running experiments. Please follow this tutorial to install it. We also suggest to use PyEnv to install and manage multiple Python versions and virtual environments. Our scripts should set up the correct experiment environment on the client VMs (i.e. the trainer nodes) from where experiments will be executed.