PLUMBER: DIAGNOSING AND REMOVING PERFORMANCE BOTTLENECKS IN MACHINE LEARNING DATA PIPELINES

Michael Kuchnik 1* Ana Klimovic 2* Jiří Šimša 3 Virginia Smith 1 George Amvrosiadis 1

ABSTRACT

Input pipelines, which ingest and transform input data, are an essential part of training Machine Learning (ML) models. However, it is challenging to implement efficient input pipelines, as it requires reasoning about parallelism, asynchrony, and variability in fine-grained profiling information. Our analysis of over two million ML jobs in Google datacenters reveals that a significant fraction of model training jobs could benefit from faster input data pipelines. At the same time, our analysis indicates that most jobs do not saturate host hardware, pointing in the direction of software-based bottlenecks. Motivated by these findings, we propose Plumber, a tool for finding bottlenecks in ML input pipelines. Plumber uses an extensible and interpretable operational analysis analytical model to automatically tune parallelism, prefetching, and caching under host resource constraints. Across five representative ML pipelines, Plumber obtains speedups of up to 47× for misconfigured pipelines. By automating caching, Plumber obtains end-to-end speedups of over 50% compared to state-of-the-art tuners.

1 INTRODUCTION

The past decade has witnessed tremendous advances in Machine Learning (ML), leading to custom hardware accelerators (Jouppi et al., 2020), sophisticated distributed software (Abadi et al., 2015), and increasing dataset sizes (Krizhevsky et al., 2012; Wu et al., 2016; Deng et al., 2009; Krasin et al., 2017; Lin et al., 2014). While ML accelerators can radically reduce the time required to execute training (and inference) computations, achieving peak end-to-end training performance also requires an efficient input pipeline that delivers data for the next training step before the current step completes. For example, an ImageNet dataloader can improve end-to-end ResNet-50 training time by up to 10× by properly leveraging parallelism, software pipelining, and static optimizations (Murray et al., 2021; Mattson et al., 2020; He et al., 2016). An efficient input pipeline also ensures that accelerator hardware is well-utilized, lowering costs.

Our analysis of over two million ML training jobs from a variety of domains (e.g., image recognition, natural language processing, reinforcement learning) at Google shows that input data processing bottlenecks occur frequently in practice, wasting valuable resources as ML accelerators sit idly waiting for data. We find that in 62% of jobs, the input data pipeline repeatedly produces batches of data with a delay of at least 1ms after the accelerator/model is able to consume it, incurring a non-negligible slowdown per training step.

To understand this phenomenon, we classify input pipeline bottlenecks as hardware bottlenecks, which occur when input data processing saturates host CPU and/or memory resources, or software bottlenecks, which don’t saturate the host due to poor configuration or I/O. We find that the majority of input data stalls arise due to software bottlenecks, indicating a mismatch between host resources and the software that drives them. While today’s input data pipeline libraries hide implementation complexity behind easy-to-use APIs (§ 2.1), it is difficult for ML users to understand and optimize the performance properties of input data pipelines. Localizing an input pipeline bottleneck with existing systems requires profiling the ML job and manually analyzing its execution trace (§ 2.2), which is error-prone, burdensome, and difficult to understand. Our experience echoes lessons learned in databases and analytics, which indicate it is unreasonable to expect experts in ML to also be experts in writing data pipelines (Chamberlin et al., 1981; Olston et al., 2008b; Arnbright et al., 2015; Yu et al., 2008; Melnik et al., 2010; Olston et al., 2008a).

Motivated by the prevalence of input pipeline bottlenecks across real ML training jobs and the burden users experience when attempting to debug the performance of input pipelines, we introduce Plumber, an open-source tool that traces input pipeline execution, models the pipeline’s components, and predicts the effect of modifications. Plumber can be used with a single line of code over arbitrary in-

1 Work started while at Google 1 Carnegie Mellon University 2 ETH Zürich 3 Google. Correspondence to: Michael Kuchnik <mkuchnik@cmu.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).
put pipelines and consists of two components: a tracer and a graph-rewriter. Similar to database execution planners (Ioannidis, 1996; Oracle, 2021), Plumber’s tracer quantifies the performance of individual operators, focusing the practitioner’s attention on the most underperforming subset of the data pipeline, while also quantifying the resource utilization (i.e., CPU, disk, memory) of the pipeline. Plumber’s rewriter is an automatic front-end to the tracer and acts as an optimizer without user intervention—introducing parallelism, caching, and prefetching in a principled fashion, and can be extended to support more. Our contributions are:

(1) We analyze two million ML jobs, providing evidence that input data processing is a common bottleneck (§3).
(2) We introduce a principled tracing methodology, resource accounted rates (§4.4), which automatically estimates pipeline CPU, disk, and memory requirements.
(3) We present a novel linear programming (LP) (§4.3) formulation using the rates traced during runtime, predicting an upper bound on performance. Unlike state-of-the-art tuners, which have unbounded error, the LP’s predictions of system state are bounded within 4× by resource usage.
(4) We present Plumber (§4), a tool that detects and removes input pipeline bottlenecks using resource-accounted rates and the LP. Plumber currently supports automatic injection of parallelism, prefetching, and caching, and paves a way forward toward more general query optimizer extensions. Plumber requires one line of code to use.
(5) We evaluate Plumber (§5) on five workloads with end-to-end performance improvements up to 47× over mis-configured pipelines and 50% over state-of-the-art tuners.

2 PLUMBING BASICS

All ML training begins with input data, which is curated by input pipeline frameworks. In this section, we outline the abstractions provided by input pipeline frameworks, noting design decisions that have an effect on understanding performance (§2.1). We next jump into common tools for understanding bottlenecks (§2.2).

2.1 Input Pipeline Architecture

Input pipelines specify: a data source, transformation functions, iteration orders, and grouping strategies. For example, image classification pipelines read, decode, shuffle, and batch their (image, label) tuples, called training examples, into fixed-size arrays (Deng et al., 2009; Krizhevsky et al., 2012). Unlike batch processing frameworks (e.g., Spark (Armbrust et al., 2015), Beam (bea, 2021), Plume (flu, 2021)), which may be used to create the data used for training, the main goal of input pipeline frameworks is to dynamically alter the training set online. Three major reasons for online processing are: 1) data is stored compressed in order to conserve storage space, 2) data is randomly altered online using data augmentations, and 3) practitioners may experiment with features throughout modeling.

Input pipelines are programmed imperatively or declaratively, each with different APIs. Imperative frameworks, like PyTorch’s and MxNet’s (Contributors, 2019; MXNET, 2018) DataLoader, allow users to specify their pipelines in plain Python by overriding the DataLoader. Declarative libraries, like DALI (Guirao et al., 2019) and TensorFlow’s tf.data (Murray et al., 2021; Tensorflow, 2020b), compose functional, library-implemented primitives, which are executed by the library’s runtime. While both styles are equally expressive in terms of pipeline construction, frameworks leave a large part of implementation to the user. In contrast, the libraries decouple specification from implementation, requiring the user to merely declare the pipeline structure, offloading optimizations to the runtime. We focus our discussion on tf.data, because it allows for various backends to service similar pipeline definitions, enabling tracing and tuning behind the API, and can be used with all major training ML frameworks.

Input Pipeline Abstractions. In tf.data, Datasets are the basic building blocks for declaring input pipelines. In Figure 1, each function call chains Datasets. Instantiating a Dataset (line 5) yields a tree composed of one or more Iterators, which produces a sequence of training examples through an iterator interface that maintains the current position within a Dataset. Figure 2 illustrates how Datasets are unrolled into an Iterator tree. Some Datasets (see Map) implement multi-threaded parallelism within the corresponding Iterator, while others (see TFRecord) can only be parallelized by reading from multiple sources in parallel (e.g., using Interleave). An Iterator implements the following three standard Iterator model (Lorie, 1974; Graefe, 1994) methods:

- **Open** defines Iterator parameters and references to child Iterators and initializes internal state.
- **Next** yields an example from the Iterator or a signal to end the stream. Source nodes read from storage or memory to yield examples. Internal nodes call Next on

```python
model = model_function() # Initialize model
ds=dataset_from_files().repeat()
3
ds=ds.map(parse).map(crop).map(transpose)
4
for image, label in ds: # Iterator Next()
    model.step(image, label) # Grad Update
```
The root waiting for the input pipeline exceeds tens of microseconds. An input bottleneck occurs when the input pipeline is not keeping up with the training computation, causing a data stall (Mohan et al., 2021). The current practice of pipeline tuning, which optimizes the throughput (rate) of the pipeline, is explained below.

**Profiler.** Event-based profilers, such as the TensorFlow Profiler (TensorFlow, 2020a), can emit metadata at particular software events to aid in determining control-flow. As there are thousands of concurrent events per second for a pipeline, it is difficult to quantitatively determine which events actually caused a throughput slowdown. To automate and generalize past heuristics deployed in guides (Tensorflow, 2020), the TensorFlow Profiler added a bottleneck discovery feature (TensorFlow, 2021). This tool works by finding the Iterator with highest impact on the critical path of a Dataset. However, it can only rank Datasets by slowness, and it can’t predict their effect on performance. Furthermore, critical-paths are not well-defined for concurrent and randomized event graphs (e.g., execution times of individual operations overlap and are data-dependent), forcing heuristics to be used.

**Tuners.** tf.data applies dynamic optimization of pipeline parameters when users specify AUTOTUNE for supported parameters, such as the degree of parallelism and size of prefetch buffers (Murray et al., 2021). The autotuning algorithm works by representing Iterators in a pipeline as an M/M/1/k queue (Lazowska et al., 1984; Shore, 1980), and a formulation for the queue’s latency is analytically determined. Statistics about the execution of Iterators are recorded with a lightweight harness. Combining the analytical model with the runtime statistics enables tuning the latency of the pipeline with respect to performance parameters. Specifically, the processing time of each element is normalized by the parallelism and the ratio of input to output elements. This statistic is then combined with “input latency” statistics of the children nodes in a node-type dependent way to get an “output latency”. Output latency tuning is done via hill-climbing or gradient descent and ends when the tuning plateaus or reaches a resource budget. While AUTOTUNE works in practice, it is hard to understand and extend for two reasons. First, open-systems, like M/M/1/k queues, have a throughput purely dependent on input arrival rates, which are not applicable for closed-systems. Second, because resource utilization is not modeled, the output latency function can be driven to zero if parallelism is allowed to increase unbounded, forcing heuristic constraints to be used.

### 3 SOT THE LEAK: FLEET ANALYSIS

We analyzed over two million ML jobs that ran in Google datacenters to determine whether input bottlenecks are common and characterized their most typical root cause. The jobs we analyzed used tf.data and ran over a one-month period from July to August of 2020. The workload included production and research jobs from a variety of ML domains, including image recognition, natural language processing, and reinforcement learning. To measure the frequency of input bottlenecks in practice, we measured the average time spent fetching input data per training step across jobs.

#### 3.1 Are Input Bottlenecks Common?

We detect input pipeline bottlenecks by measuring the average latency across all Iterator Next calls, which is the average time the job spends blocked waiting for input data in each training step.

**Observation 1:** For a significant fraction of ML jobs, the
Figure 3: Time spent fetching training examples using Next, in ms. On average, for 92% of jobs Next latency exceeds 50µs, for 62% of jobs it exceeds 1ms, and for 16% of jobs it exceeds 100ms. Fetch latencies for well-configured pipelines are in the low tens of microseconds.

input data pipeline produces data at a slower rate than the model is able to consume it.

Figure 3 shows that for 62% of jobs, the average Next latency per training step exceeds 1ms and for 16% of jobs, the average wait time exceeds 100ms. Since the input pipeline affects both end-to-end training performance and hardware accelerator utilization, it is a critical part of ML training to optimize. We note that fetch latency applies to each iteration of the training process—over ten thousand times in a typical session. At any point in time, between 1–10% of the fleet is waiting on input data, which is significantly costly at the scale the fleet operates at.

3.2 Why Do Input Bottlenecks Occur?

We classify input pipeline bottlenecks into two categories: hardware and software bottlenecks. Hardware bottlenecks occur when hardware resources used for data processing saturate. The hardware resources typically used for input processing are CPU cores, host memory, and local or remote storage. Many workloads do not use local storage for I/O, but pull their data from external data sources (e.g., distributed filesystems) (Murray et al., 2021). Thus, external I/O resources may bottleneck training jobs. Hardware resource saturation can be remedied by adjusting the resource allocation, e.g., switching to a node with more cores, memory, or I/O bandwidth.

Software bottlenecks occur because the software is not driving the hardware efficiently, e.g., by using too little or too much parallelism, or incorrectly sizing prefetch buffers to overlap communication and computation. When a user is confronted with a software bottleneck, they must find the root cause and fix it; otherwise, they risk underutilizing hardware performance. We note I/O bottlenecks can also be caused by software configuration, due to inefficient access patterns and low read parallelism.

Observation 2: Host hardware is rarely fully utilized for jobs with high input pipeline latencies. Thus, input bottlenecks are likely rooted in software or I/O inefficiencies, rather than hardware saturation.

To understand the breakdown between these categories of input bottlenecks, we measure the host CPU and memory bandwidth resource utilization for the jobs captured in our analysis. In Figure 4, we show a breakdown of different jobs’ average Next call latencies organized according to the CPU and memory bandwidth utilization of the pipeline host. We exclude jobs with latency below 50µs because for those jobs the input pipeline is not a bottleneck, as it takes tens of microseconds to read input data that is readily available from a prefetch buffer (including thread wakeup and function invocation time). Our data indicates that jobs with latency between 50µs and 100ms (small, dark dots) utilize more of the host’s resources than those with latency higher than 100ms (large, blue dots). For context, a TPUv3-8 (Jouppi et al., 2020; MLPerfV0.7) takes roughly 120ms to process a minibatch for ResNet-50 (He et al., 2016). Jobs where the average fetch latency exceeds 100ms are, then, significantly input-bound, and as shown in the Figure 4 their resource usage for both memory bandwidth and CPU is concentrated below 20%.

4 PLUMBER

To address the challenges that users face in configuring input data pipelines in ML jobs, we introduce Plumber, an extensible tool that automatically detects and removes input data bottlenecks. Using a top-down approach, we start with the architecture of Plumber (§4.1) and then explain tuning methodology (§4.2). We later formulate the resource allocation problem across Datasets as a LP (§4.3), which relies on per-Dataset resource rates derived by Plumber (§4.4). Plumber is released as an open-source artifact.
Figure 5: An ImageNet pipeline with Plumber’s various states of processing. Plumber starts with Dataset-level tracing, which is then followed by analysis for CPU, disk, and memory costs, which are subsequently modeled and optimized. Reading a TFRecord, for example, is converted from bytes read to an I/O cost per minibatch, which can then be linked with available I/O resources to find bottlenecks.

4.1 Software Architecture

Plumber reasons about performance in a layered fashion. The goal of the layers is to abstract basic Dataset-level statistics into costs, which can be compared, optimized over, and extended. We demonstrate this architecture with a simplified, misconfigured ImageNet pipeline in Figure 5. This pipeline requires reading from TFRecords in parallel, decoding the examples of each record, and randomly augmenting the examples with crops and flips. Each of the pipeline components has a tunable, except for the TFRecords, which is parallelized by Interleave.

Tracing. By enabling a runtime flag, tracing collects Dataset-level statistics, such as counters for elements processed, CPU time spent, and the number of bytes per element. Plumber periodically dumps these statistics into a file along with the entire serialized pipeline program. Joining the Datasets with their program counterpart enables building an in-memory model of the pipeline dataflow. The bottom of Figure 5 shows how a TFRecord Dataset is traced, which reads a single TFRecord file. Plumber’s tracing instruments all read() calls into the filesystem within tf.data, allowing it to see all the reads into the 144MB file. Each record is unpacked into roughly 1200 elements (110kB images), which Plumber counts. By inspecting the serialized program, Plumber knows that there are 1024 total TFRecord files, and thus can estimate that the dataset contains 1024 \times 1200 elements—ImageNet has 1.2 million. To get CPU usage, Plumber wraps a thread-level CPU timer around Next calls, which only counts active (not blocked) CPU-cycles. All Next calls are instrumented such that: 1) CPU timers stop when Dataset calls into their children and start when control is returned and 2) statistics (e.g., counts and sizes) about each yielded element are attributed to the producer. The statistics necessary for optimization total less than 144 bytes per Dataset.

Analysis. To find bottlenecks, Plumber must analyze the traced data using analytical modeling, which puts each Dataset in units of cost that can be compared. Plumber treats the pipeline as a closed system, with each component operating asynchronously yet sharing the same resource budget. For example, we can see that TFRecord reads at a rate of 15MB per minibatch, but decoding consumes 1/2 core per minibatch. Determining which Dataset is the bottleneck depends on the resource allocation of CPU and I/O—for example, 30 minibatches per second can only be hit with 450MB/s of I/O and 15 CPU cores.

Optimizer. Plumber supports reasoning about CPU and disk parallelism, caching, and prefetch injection. Achieving optimal CPU and I/O parallelism requires allocating sufficient parallelism to keep the pipeline balanced in terms of throughput capacity, which we formalize in the LP. Caching reduces the amount of work done by avoiding re-computing data dependencies, and the optimal cache minimizes total work by placing it as high in the pipeline as possible. Prefetching is a subsequent pass which injects prefetching proportional to the idleness in the pipeline under a benchmark workload. As shown in the example, by placing caching at MapDecode, we can avoid all CPU and I/O operations except for the subsequent Crop and Batch at the expected cost of 790GB of memory. In this example, the optimizer knows that the machine only has 300GB of memory and thus it must settle with caching at the 148GB Interleave. By caching Interleave, Plumber can redistribute remaining CPU resources to other stages with the LP.

Extensions. Extending Plumber requires minimal effort by cleanly interacting with existing optimizations. For example, to add the ability to cache materialized results to disk in addition to memory, one can reuse all caching logic up to the cache decision itself, which would dispatch to in-memory caching preferably and disk caching if space and disk bandwidth allow it. Significant extensions (e.g., networking or GPU data transfer I/O) require adding the corresponding tracing and rates at the lower levels, which can then be incorporated as LP expressions and constraints.

Two areas of future work are in extending Plumber’s optimizer to reason about correctness optimizations, such as reusing data (Choi et al., 2019), and extending Plumber to perform optimal resource provisioning for matching a target throughput (e.g., to minimize cost).
4.2 Tuning Methodology

The intuition behind Plumber’s performance debugging methodology is that input pipeline performance is limited by the Dataset with the lowest throughput and can be alleviated by adjusting the resource allocation (out of the available hardware resources) for this Dataset. However, simply maximizing the number of threads used to compute elements for a parallelizable Dataset is not always productive, as threads compete for CPU and memory resources. The end-to-end throughput of the input pipeline can also be limited by I/O bandwidth. Caching Datasets in memory alleviates CPU or I/O bottlenecks at the cost of memory capacity. As all Plumber traces are also valid programs (that can be rewritten), Plumber simply requires a way for a user to mark the program for tracing, and thus one entry point is sufficient for Plumber to accomplish all tuning.

**Tuning Interface.** Plumber introduces tuning annotations to add on top of existing data loading code. A data loading function is one which returns a tf.data Dataset, which has an associated signature. Tagging the code with an @optimize annotation gives Plumber permission to intercept one or more Datasets and return an optimized variant matching the Dataset signature. The annotation provides Plumber with an entry point into the loader allowing Plumber to trace it under a benchmark workload, and rewrite it before passing it back to the application.

**Modeling.** Plumber maximizes throughput by modeling the input pipeline as an asynchronous closed-system in the context of operational analysis (Denning & Buzen, 1978), which explicitly defines bottlenecks. The operational framework has few statistical assumptions, unlike Markovian queues, and parameterizes each component of a system with a cost relative to the resource usage, usually expressed in units of time. Plumber further measures the resources and traces the analytical network to automatically “operationalize” and tune an arbitrary input pipeline.

4.3 Allocating Hardware Resources

In order to derive an improved input pipeline configuration, we need to understand how each Dataset’s performance is affected by changing the fraction of hardware resources allocated to it, e.g., by changing the degree of parallelism of a Dataset or inserting prefetching or caching Datasets which consume extra memory. We are interested in characterizing the usage of three types of resources in input pipeline execution: CPU, disk bandwidth, and memory capacity. We first formulate a Linear Program (LP) to solve for the optimal CPU resource allocation before moving onto disk and, finally, memory capacity. The calculation of the inputs into this LP are discussed in the following subsection (§4.4).

CPU. For CPU optimizations, we optimize over a Dataset tree in the input pipeline to decide what fraction of the CPU-time each Dataset should receive such that throughput is maximized. Having two Datasets in series with rate \( R_1 \) and \( R_2 \) yields an aggregate rate of \( X = \min(R_1, R_2) \). The bottleneck Dataset determines performance. For example, in Figure 2, if Map has rate \( R_1 \) and Batch has rate \( R_2 \), to increase \( X \), we must assign the slower of the two Datasets more resources. However, the above is only true if we can parallelize the bottleneck node and if we have resources left.

We optimize \( \max_{\theta} [X = \min_{i \in D}[\theta_i * R_i]] \) subject to constraints: \( \sum_{i \in D} \theta_i \leq n_c; \theta \in D \geq 0; \theta \in S \leq 1 \). In the above equations, \( D \) is the set of Datasets in consideration, \( S \subseteq D \) a subset of sequential Datasets, \( n_c \) the total number of cores, \( X \) throughput, \( R_i \) the measured rate of minibatches per second per core, and \( \theta \) the fractional number of cores allocated. The equation maximizes the input pipeline throughput \( X \) by maximizing the aggregate rate \( (\theta_i * R_i) \) of the slowest Dataset (the minimum). We constrain sequential Datasets to have at most one core, and we cannot exceed all cores on the machine.

**Disk.** As shown in Figure 2, data flows from disk (TFRecordDataset) into the CPU section of the pipeline. Therefore, if the data source is slower than the rest of the pipeline, the entire pipeline is disk-bound. For large reads (e.g., records), there are two factors that can cause a disk bound: 1) insufficient read parallelism and 2) max disk I/O bandwidth. The latter can be found via benchmarking tools (Axboe, 2021), but Plumber goes a step further by benchmarking entire empirical parallelism vs. bandwidth curve for a data source (via rewriting). The source parallelism results can then be fit with a piecewise linear curve to be injected into the optimizer to determine a minimal parallelism to hit max bandwidth.

**Memory.** Caching aggressively is always desirable, because it allows I/O to be entirely reduced and some CPU processing to be partially reduced. The potential speedup grows as one goes up the pipeline, but so does the memory requirement to cache, which may exceed system memory. Plumber checks if operators are random functions (e.g., augmentations), as described in the appendix (§B.1)—randomized functions have infinite cardinality and cannot be cached. However, if an operator’s output is finite, Plumber estimates the materialized size. To solve memory caching for typical, linear-structure pipelines, Plumber uses a greedy (yet optimal) approach to select the Dataset closest to the root that fits in memory. Plumber can also solve for more generic topologies by adding Boolean decision variables for each cache candidate over the LP already presented.
4.4 Resource Accounted Rates

The LP construction (§4.3) assumes the existence of calculated rates (for CPU and disk I/O) as well as the materialized size (for caches), and uses those values as inputs into the algorithm. This section explains how those values are calculated by using resource accounted rates. Resource accounted rates encompass the cost of an operation or decision to cache in the pipeline. For CPU and I/O, the cost is the ratio of CPU core-time or I/O bytes per minibatch, which are throughput bounds. Memory capacity measures the cost in terms of bytes required for materialization at a particular dataset, and is a throughput optimization. The middle of Figure 5 illustrates all three of these costs. The full algorithm for resource accounted rates can be found in the appendix; however, we give a brief description here.

**Common Units.** The root of the pipeline gives a common set of units for CPU and I/O: minibatches. Children of the root do not necessarily output elements in terms of minibatches (e.g., prior to batching); thus, a conversion factor between an arbitrary dataset’s elements and that of the root must be calculated—this is called the “visit ratio”, $V_i$, and represents the mean number of completions at dataset $i$ for each completion from the pipeline. To calculate $V_i$, we start with the pipeline’s root “visit ratio” $V_0 := 1$. Then, we apply the following recurrence: $V_i = (C_i / C_{i-1}) \times (C_{i-1} / C_0)$, where $C_i$ is the average number of elements of work completed at dataset operation $i$. The former ratio is the dataset’s local input-output ratio, and the latter is calculated in the recurrence. Intuitively, the visit ratio allows one to say that $n$ elements are in a batch, and thus dependencies to batching must have a throughput $n$ times faster to “keep up”.

**Throughput Cost.** The throughput at the root, $X_0$, is the number of minibatches completed, $C_0$, in a timeframe, $T$, and the child datasets have $X_i = V_i X_0$. However, this equation does not explain the bottleneck cause, which requires reparameterizing the throughput in terms of CPU core time or I/O bytes. As $X_i = C_i / T$, we factor the equation into 1) the product of completions per resource (e.g., elements per core-seconds) and 2) resource per time (core-seconds per time). The former is the ratio of two traced variables (element completions and CPU-time or bytes used), and the latter is a knob for modeling adding or removing resources (e.g., CPU parallelism or extra bandwidth). In the LP (§4.3), $R_i$ is the first factor normalized by $V_i$, and $\theta_i$ is the second factor; in a bottleneck, they determine $X_0$.

**Materialization Cost.** Estimating the size of a dataset’s materialized artifacts is similar to the prior operational treatment but involves propagating the estimates up from data source to root. The materialized size of a data source is the product of 1) the number of elements (cardinality) and 2) the average size of each element. Both are necessary because a dataset’s semantics may modify one or the other; for example, truncation only modifies the former and decompression only the latter. To start, the size of a data source is the number of files, $n$, times the average bytes per file, $b$. Propagating the number of elements, $n_i$, involves multiplying $n$ by an input-output completion ratio. The sum of output bytes and the number of completions for each dataset is measured in tracing, and thus $b_i$ is readily computed. $n_i$ can grow unbounded (and thus uncacheable) if the data is infinitely repeated or augmented. Datasets that are children to a cache can be modeled as having no steady-state cost (e.g., after the first epoch).

5 Evaluation

We evaluate CPU bottleneck removal in §5.1, showing that Plumber can indeed find bottlenecks, and we further analyze how Plumber’s solutions differ from those of baselines. We additionally evaluate disk and caching in §5.2 and §5.3. End-to-end results are presented in §5.4. The appendix provides additional details and demonstrates that Plumber’s overhead on modern hardware is less than 21% (due to workloads) and drops to below 5% on vision workloads. We compare against a naive configuration, which has minimal parallelism, to two strong baselines: AUTO-TUNE (Murray et al., 2021) and HEURISTIC, which set the parallelism tunables to the number of cores on the machine.

**Hardware.** For microbenchmarks, we evaluate over two setups to ensure our results generalize. Setup A is a consumer-grade AMD 2700X CPU with 16 cores and 32GiB RAM. Setup B is an older enterprise-grade 32-core Intel E5-2698V3 Xeon 2GHz CPU with 64GiB RAM. Setup C is for end-to-end results and is a TPuv3-8 (Jouppi et al., 2020) with 96 Intel Xeon cores and 300GB of RAM.

**Workloads.** Our evaluation uses the MLPerf v0.6 subset of MLPerf training (Matsson et al., 2020) benchmarks, which is representative of datacenter workloads and covers both images and text. We use the following tasks and datasets: ResNet/ImageNet (He et al., 2016; Deng et al., 2009), Mask RCNN/COCO (Ren et al., 2016; Lin et al., 2014), MultiBoxSSD/COCO (Liu et al., 2016), Transformer/WMT (Vaswani et al., 2017; Second Conference on Machine Translation, 2017), and GNMT/WMT (Wu et al., 2016; First Conference on Machine Translation, 2016).

5.1 CPU Bottleneck Removal

To assess how accurately bottlenecks are found, we use Plumber’s analysis layer to rank nodes by bottleneck. The pipeline’s parallelism parameters are initialized to the naive configuration (parallelism=1) with prefetching, and Plumber iteratively (using 1 minute of tracing) picks the node to optimize by ranking nodes by their parallelism-scaled
rates. To compare against uninformed debugging, we plot a random walk, which randomly picks a node to parallelize for each “step” (x-axis in our plots). We run each experiment three times to get confidence intervals.

**Sequential Tuning.** Figure 6 shows the ResNet workload across both setups; other workloads look similar. **Plumber** is consistently better than the random walk, as expected. We observe that both HEURISTIC and AUTOTUNE are equivalent in terms of reaching peak performance—over-allocation does not usually result in performance degradation. On the ResNet workload, the bulk of the work is in the JPEG decoding **Dataset**, which services 2.5 minibatches/second/core on Setup A. Most of Setup A’s steps are spent increasing the parallelism of this **Dataset**, a transpose operation being the second bottleneck. The bumps in Setup B correspond to increasing the parallelism of Transpose rather than JPEG decoding and occur roughly once every 8 steps. We observe that, across all pipelines, such transition regions are the only regions (in addition to fluctuations at convergence) where **Plumber** struggles to characterize the locally optimal decision (see **MultiBoxSSD** example in Appendix). While Setup B has 2× more cores than A, the per-core decoding rates for B are lower, resulting in only a 1.2× higher throughput.

**Observation 3:** **Plumber**’s bottleneck finder converges to the optimal throughput in 2–3× fewer steps than a random walk. Inspecting the individual **Dataset** rates offers pipeline and machine performance insights.

**Linear Programming.** Figure 7 demonstrates that **Plumber** can understand performance through the LP formulation on ResNet; other workloads are similar. As a baseline, we include a “local” method, which allocates all remaining resources to the current bottleneck node. This baseline is unable to see past one bottleneck and thus oscillates as the bottleneck node changes (the “bumps” in Figure 6). Meanwhile, the LP steadily declines—upon inspecting the solution, we find a strong correlation with the LP’s decline and the value of $R_i$ for the bottleneck node (JPEG decoding). While $R_i$ typically decreases (e.g., due to scaling overhead), we find it increases briefly especially on Setup B, peaking at step 10 and resulting in a “bell shape” LP prediction (JPEG rate peaks at 1.8 and then drops to 1.4 by the end of training), explaining the peak in the LP. AUTOTUNE, being oblivious to saturation, either overestimates or underestimates throughput without bounding it.

**Observation 4:** **Plumber**’s LP solution captures both resource utilization and bottlenecks, bounding throughput to within 2× from when optimization starts for pipelines like ResNet and **MultiBoxSSD**. The bounds get tighter as optimization proceeds due to differences in empirical rates.

**Large UDF Parallelism Challenges.** As shown in Figure 8, **RCNN** on Setup A is challenging and displays counter-intuitive behavior. The reason is that RCNN features a large UDF in its **MapDataset**, which is transparently parallelized by the TensorFlow runtime. Parallelizing the **Map** is dangerous because the parallelism compounds with UDF parallelism—1 parallelism uses nearly 3 cores! As parallelism increases for this **Dataset**, the corresponding...
While the LP overestimates peak performance by 4×, it is still qualitatively better than AUTOTUNE, which oscillates in predictions. Upon inspection, AUTOTUNE allocates 16 parallelism to the bottleneck node, but 3 parallelism to a different MapDataset node. The bottleneck node operates at 0.5 minibatches/second/core, while the other node operates at 20 minibatches/second/core. Thus, the optimal policy, which Plumber follows, is to only allocate parallelism to the main bottleneck (thus bounded by 0.5 × 16 = 8). In fact, due to UDF parallelism, only 4–5 parallelism is necessary. Counterintuitively, this policy is no longer optimal for our end-to-end results (§5.4), which have 6× more cores.

**Observation 5:** AUTOTUNE and HEURISTIC are vulnerable to over-allocation, which can cause performance degradation. Pipelines with heavy UDF parallelism may experience drops on the order of 10%. Dynamic parallelism makes end-to-end performance hard to predict.

**Text Processing (NLP) Challenges.** We observe that both Transformer and GNMT are difficult to optimize in practice. As shown in Figure 9, both pipelines are predicted to be 2–8× faster than they actually end up being. Upon investigation, we observe that nearly all operations in NLP are very small e.g., grouping a few integers in a vector. The operations are so small that they are significant compared to the Iterator abstraction’s overhead, causing idle “bubbles”.

According to Plumber, GNMT is bottlenecked by ShuffleAndRepeatDataset: this Dataset is performing minimal work and thus the result is unexpected. Before getting to this point, Plumber allocated 9 parallelism to the first MapDataset and then gave up upon seeing the non-optimizable Dataset. Similarly, for Transformer, after alternating optimizing the 3 MapDatasets (all similarly fast), Plumber indicates Transformer is bottlenecked by FilterDataset, which is operating at about half of its max rate (explaining the 2× difference). Using outer-parallelism for both pipelines, introducing inner-parallelism for Batching (GNMT), and resuming Plumber’s optimization results in closing the predicted performance gap to less than 2× (Transformer) and 4× (GNMT).

**Observation 6:** Text pipelines can require significant tuning to overcome framework overheads. An in-memory cache materializing results is ideal, when possible.

### 5.2 Disk Microbenchmarks

To simulate various bandwidths, we implement a token-bucket bandwidth limiter in the TensorFlow filesystem layer and use the most I/O intensive pipeline, ResNet. Plumber correctly concludes that ImageNet records are approximately 110KB on average and infers that 128 records are necessary for 1 minibatch. It thus infers that the I/O load per minibatch is 128 × 110 KB, or 6.9 minibatches per 100MB/s of bandwidth, which it can join with known bandwidth (e.g., from token-bucket or fio (Axboe, 2021)). Using this bound, Plumber predicts the pipeline’s performance within 5% from 50MB/s to 300MB/s, when the compute bottleneck begins. We observe similar results for RCNN and MultiBoxSSD, though MultiBoxSSD is easier to bottleneck consistently due to its faster CPU speed. Plumber estimates that RCNN and MultiBoxSSD can push 145 minibatches/sec on 100MB/s, since they use the same dataset and same batch size; therefore, MultiBoxSSD is 25× more I/O bound for a fixed CPU.

We then run this experiment on setup B with a real HDD (Seagate ST4000NM0023) and NVMe SSD (400 GB Intel P3600), which have 180MB/s and 2GB/s of read bandwidth, respectively. We run the load for one minute to prevent the page cache from kicking in (when the dataset reads repeat). For ResNet, Plumber predicts 11.06 minibatches/sec, and we are bound at 12.7 (15% error). On the NVMe SSD, Plumber predicts 135 minibatches/sec, and, indeed, we observe a compute bound. On RCNN, Plumber predicts disk bounds of 970 and 11850 minibatches/sec, respectively; for both, we observe the compute bound of 14 minibatches/sec. On HDD with MultiBoxSSD, Plumber predicts 235 minibatches/sec, whereas we observe 215 (10% error). On NVMe SSD with MultiBoxSSD, Plumber predicts a disk bound of 2900 minibatches/sec; we observe the compute bound being hit before the drive is saturated. The text datasets are too small to test.

**Observation 7:** Plumber is able to bound disk-bound workloads to within 15% of the observed throughput, notifying users of potential hardware misconfigurations.

### 5.3 Memory (Cache) Microbenchmarks

We evaluate Plumber’s predictions on memory optimization and summarize them below. Plumber predicts that
148GB are necessary to cache ImageNet for ResNet; 20GB to cache COCO for RCNN and MultiBoxSSD; and 1GB and 2GB for the Transformer and GNMT WMT datasets, respectively, which matches their known size. While this is expected for a full sweep of the training set (by simply tracking all file sizes), it also holds for sampling the dataset. Empirically, we find that observing a small subset of the data is sufficient: 1% of files provide a relative error of 1% for ImageNet and 2% for COCO, and 5 files gives less than 2% relative error for WMT datasets.

For materialized caches, we observe that Plumber predicts no changes until a node is hit, which changes the bytes/element. For ImageNet, we observe that, if Plumber is fed a fused decode and crop pipeline, it predicts caching is only possible at the source, because the crop is random. However, when the ImageNet pipeline is not fused, image decoding amplifies the dataset size by $6 \times$, which Plumber observes as 793GB of a true 842GB, or 6% error with 60 seconds of profiling. For this pipeline, we observe that relative error decreases as a function of tracing time, saturating at 2 minutes and yielding relative error rates below 1%, offering a knob for refining estimates at the expense of tuning time. RCNN can only be cached at the disk-level, since the following UDF is randomized. For MultiBoxSSD, Plumber detects it takes 84GB of a true 97GB (14% error) to materialize the dataset in memory after image decoding with only 60 seconds of profiling, with 2 minutes yielding a 5% error, which continues to drop by ~1% for each additional minute. Plumber additionally detects that the filter in the pipeline reduces the dataset by less than 1%.

**Observation 8:** Plumber captures dataset sizes at the source exactly, and, for large datasets, it is able to subsample 1% of files to obtain 1% error. For materialized caching, Plumber propagates changes to dataset sizes (e.g., data decompression and filtering).

### 5.4 End-to-End Pipeline Optimization

The end-to-end benefits of Plumber’s optimizations, evaluated over 5 epochs of training, are shown in Figure 10. None of the pipelines have caches inserted manually, naive configurations have 1 parallelism and no prefetching, and HEURISTIC uses the prefetching hard-coded into the dataset. We observe overprovisioning (HEURISTIC) is competitive, if not faster, than AUTOTUNE across all trials. Plumber can go beyond both of these strong baselines because of caching, obtaining up to a 47× speedup. Such a speedup, an absolute throughput of over 14k images/second, is only possible because caching bypasses the (Plumber-derived) 11k image/second data source bottleneck for the cloud storage. For ResNet-18, it is sufficient to cache at the data source, which is only 148GB; therefore, Plumber picks the CPU-optimized branch of the pipeline (code in Appendix). However, when we use a linear model for ResNet, we use the smaller validation set, which allows Plumber to cache the $6 \times$ bigger decoded images in memory, avoiding a CPU bottleneck. We also evaluate ResNet-50 and find that Plumber obtains a 24× speedup over the naive configuration, though it cannot improve over other baselines, as the model’s throughput limits are hit at 8k images/second.

Compared to AUTOTUNE, Plumber obtains a 36–59% speedup on three of the MLPerf benchmarks and ties on Transformer and GNMT (due to model throughput). These numbers are conservative—we observe that AUTOTUNE on ResNetLinear often sets the I/O parallelism to 1, which results in a 35% performance drop compared to what is shown in Figure 10 and creates a $2.4 \times$ performance gap between AUTOTUNE and Plumber. To allow more competitive tuning from AUTOTUNE on that workload, we set the I/O parallelism to the default value of 10 used in MLPerf submissions, which improves its final performance. For RCNN, Plumber may perform worse than AUTOTUNE because Plumber is conservative in its parallelism allocation, while AUTOTUNE tends to allocate maximum parallelism to all datasets. Plumber estimates that one MapDataset is two orders of magnitude more expensive than the other. In some cases, Plumber allocates 95 parallelism to the former, leaving only 1 parallelism for the remaining MapDataset, which results in the shown 23% performance loss. However, we also observe cases where Plumber matches the throughput of the baselines because it allocates at least 2 parallelism to the cheaper Dataset, which removes the bottleneck—suggesting that a mild form of “hedging” would be effective in preventing under-allocation for such skewed workloads. For MultiBoxSSD, Plumber is able to materialize the data after filtering is performed, which! makes the cache smaller and increases throughput by removing load from the CPU. While we don’t see a large benefit from Plumber’s optimizations on the MLPerf NLP pipelines, we do see improvements when moving to the related official Flax implementations of Transformer when it is configured to use a single-layer Transformer (TransformerSmall). We see a $2.5 \times$ difference between strong baselines and Plumber on TransformerSmall because the Flax pipeline performs text-processing and packing on-the-fly, which, when combined with a smaller model, makes peak throughput only achievable with aggressive caching.

**Observation 9:** Plumber frees the user from making individual caching and parallelization decisions, enabling 50% end-to-end improvements over AUTOTUNE and heuristics and 40× improvements over naive configurations.

### 6 Related Work

**Pipeline Optimization.** Dataset Echoing (Choi et al., 2019) repeats input pipeline operations to match compute rate.
DS-Analyzer predicts how much file cache memory is necessary to match the compute steps (Mohan et al., 2021). Progressive Compressed Records (Kuchnik et al., 2021) match compression levels to likewise minimize I/O. Each of these works is characterizing a piece of the input pipeline, all of which can be homogenously dealt with via Plumber—the first is a visit ratio and the latter two are memory caching/disk.

Systems have also been built to natively support the data pipeline workload. A cache library has been developed to carefully partition and coordinate caches in distributed training (Mohan et al., 2021) Dataset echoing has been further expanded to support caching partially augmented samples (Lee et al., 2021). Filesystem middleware has been developed to natively support data prefetching (Dryden et al., 2021). While Plumber primarily focuses on native tf.data primitives, it could also be used to hook into such systems to perform cross-system optimization.

**Bottleneck Detection.** Tools in the big-data domain (e.g., Spark (Zaharia et al., 2012)) have similar performance problems as those found in ML pipelines (Ousterhout et al., 2015; 2017), but the differences in domain encourage a different approach. Notably, Monotasks (Ousterhout et al., 2017) enables bottleneck finding by designing Spark primitives to be easily measured on a per-resource level. In contrast, Plumber does not modify the framework and instead elects to carefully instrument selected resource usage, though the design is similarly simplified by having resource-specialized operations. Big data systems, such as DyradLINQ (Yu et al., 2008), have similarly benefited from dedicated debugging utilities (Jagannath et al., 2011) and dynamic query rewriting (Ke et al., 2013). Roofline (Williams et al., 2009; Cabezas & Püschel, 2014) models bound compute kernels with CPU limits. Plumber generates similar plots using Dataset and resource limits.

Recent studies have analyzed ML workloads and found that data pipelines can become bottlenecks (Mohan et al., 2021; Murray et al., 2021). This line of work is further supported by empirical evidence of bottlenecks in recommendation models (Zhao et al., 2021). Recent work has also characterized the design space of input pipelines with a profiling tool (Isenko et al., 2022); we do not focus on general pipeline decisions (e.g., the choice of image resolution), but rather focus on automatic tuning of performance knobs. These works all highlight a growing need for systems which better support the input pipeline workload.

7 CONCLUSION

Training ML models requires reading and transforming large quantities of unstructured data. Our analysis of real ML training jobs at Google shows that data pipelines do not adequately utilize their allocated hardware resources, leading to potential bottlenecks, which waste precious accelerator resources. To enable practitioners to better utilize their hardware and improve training performance, this paper introduces Plumber, an extensible tracing and optimizing tool. Plumber localizes bottlenecks to an individual operator by tracing both the resource usage and rate of each operator, enabling both bottleneck-finding as well as inferences on future performance. By adding caching in addition to tuning, Plumber can surpass the end-to-end performance of existing tuners by 50%. Future extensions to Plumber include: distributed and accelerator-infeed level profiling, optimal resource allocations for pipelines, and semantic-level re-writing of augmentations.

**ACKNOWLEDGEMENTS**

We thank the anonymous reviewers for their help improving the presentation of this paper. This material is based upon work supported by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W911NF-20-D-0002. The content of this information does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred. We thank the members and companies of the PDL Consortium: Amazon, Google, Hewlett Packard Enterprise, Hitachi Ltd., IBM Research, Intel Corporation, Meta, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital for their interest, insights, feedback, and support. Michael Kuchnik was supported by a National Defense Science and Engineering Graduate Fellowship. This research was supported with Cloud TPUs from Google’s TPU Research Cloud and research credits from Google Cloud Platform.
REFERENCES

Apache Beam: An advanced unified programming model. 


Graefe, G. Volcano - an extensible and parallel query evaluation system. Transactions on Knowledge and Data Engineering, 1994.


Ioannidis, Y. E. Query optimization. ACM Computing Surveys, 1996.


Jagannath, V., Yin, Z., and Budiu, M. Monitoring and debugging DryadLINQ applications with daphne. In International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2011.


Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines


A RESOURCE ACCOUNTED RATES ALGORITHM (EXTENDED)

The LP formulation in the main text hinges on our ability to predict two resource accounted rates: ① the rate $R_i$ of a Dataset given its configuration (e.g., number of cores assigned to each operation) and how that rate would change when one operation is assigned more cores, and ② the number of bytes needed to cache a Dataset at a given operation, accounting for bytes added/removed by data transformations applied by prior operations.

The intuition behind the computation of ① is that the rate of each operation from the source (e.g., data read from disk into the pipeline) to the root (i.e., minibatches exiting the pipeline) differs. For example, map operations may filter or amplify their inputs. We therefore begin with the root of the Dataset tree and traverse the pipeline until the source, accounting for every operation’s rate in the process. The intuition behind the computation of ② is that the total data that needs to be cached is known at the source (i.e., when it enters the pipeline) and needs to be recomputed after each operation is applied. We therefore begin with the source of the Dataset tree and traverse the pipeline until the root.

① Work Completion Rates. In input pipelines, each Dataset operation is potentially characterized by different units of work. Looking at Figure 2, we do not know how many bytes are in a minibatch or how Map operations turn bytes into training examples, but we can observe that 128 images are batched into a minibatch. We thus rely on visit ratios from Operational Analysis (Denning & Buzen, 1978), i.e., normalization constants, to compute the equivalent of root units, i.e., minibatches per second per core, that are generated by the source. Each Dataset operation is characterized by its local rate, $r_i$, which is defined as the average items of work completed per second per core, and can be computed by tracking two items: the items arrived and the items completed. The visit ratio, $V_i$, then, for a given Dataset operation is the constant that converts the local rate, $r_i$, to the global rate, $R_i$, which expresses the average amount of work completed by the operation in minibatches per second per core, i.e., the unit of work of the Dataset root given one core. Starting with the pipeline’s root visit ratio, $V_0$, which is equal to 1, Plumber computes each Dataset operation’s visit ratio using $V_i = (C_i/C_{i-1}) \times (C_{i-1}/C_0) = r_i \times V_{i-1}$, where $C_i$ is the average number of items of work completed at Dataset operation $i$.

While the pipeline is running, Plumber collects counters on items arrived and completed across all operations of a given Dataset, which is used to model CPU performance. This rests on two simplifying assumptions, which do not affect Plumber’s efficiency in practice (§5): CPU work per item is fixed, and there is linear scaling of operation performance. Extending this approach to measure disk bandwidth usage only requires measuring filesystem reads and dividing by the wallclock time of the process. Converting usage to utilization is a matter of dividing the bandwidth usage by the available bandwidth, which Plumber measures by profiling the training directory using fio (Axboe, 2021).

② Cache Amplification Rates. In ML settings, the benefit of caching is heavily skewed toward caching the entire disk-resident data—there is no locality due to random sampling. The fastest caching solution is one that is closest to the root of the Dataset, while remaining in memory. We note that random Datasets, which call into randomized functions, do not qualify for caching. By virtue of being random, their effective size is infinite—often, one can continue generating unique results forever.

To determine memory requirements for caching, Plumber assumes that files are consumed to completion and tracks the size of all input files (i.e., bytes read until end of file) used during the pipeline’s operation. Once a new file is recorded, the file is added into a system-wide map tracking filename to bytes used. The space requirements are moderate for realistic use-cases; only two integers are needed per filename with hashing, and the number of files is likely small due to large files (e.g., ImageNet has 1024 files).

To calculate how many bytes it takes to materialize each Dataset, we approximate two statistics: the number of elements (cardinality), $n_i$, and the average size of each element (byte ratio), $b_i$. For a data source, we have the Dataset size in bytes: $\sum_{f \in S} S_f$, where $S_f$ is the size of a file $f$ in the set of recorded files, $\mathbb{F}$. As the source data flows through the input pipeline, $n_i$ and $b_i$ will change as training examples are grouped, filtered, and/or transformed (e.g., parsed and decoded in Map). To build intuition, let us revisit Figure 2. $b_i$ for Map is just the “decompression ratio” from records to images, which is often approximated to be 10× for JPEG. Likewise, the Batch makes $b_i$ 128 times bigger in the process of grouping, while making $n_i$ 128 times smaller. Therefore, we can conclude the root Dataset is an order of magnitude larger than it was at the source.

We first tackle how to solve $n_i$. For exposition, assume the common case that we only have one source (e.g., Figure 2) and the source is finite—we will be propagating our analysis up from this source until we hit the root. Readers emit elements at per-row or per-record granularity, thus, for source Datasets, we overload $r_i$ from visit ratios to denote this ratio (though it is a ratio of records/byte). The initial dataset size at source $i$ is thus $n_i = (\sum_{f \in S} S_f) \times r_i$. For a Dataset $i$, the subsequent Dataset $j$, follows the recurrence $n_j = r_j \times n_i$, where $r_j$ follows previously defined semantics from visit ratios. Intuitively, we convert
the size in bytes to the size in records to the size in training examples, and so on. Special care must be taken to account for datasets repeating examples or truncating the examples stream. When generalizing to multiple children (multiple sources), some Dataset-specific aggregation is required (e.g., sum).

To obtain \( b_i \), we can use entirely local information. Plumber tracks one local quantity, the bytes-per-element, \( b_i = \frac{\text{bytes produced}(i)}{C_i} \), where the numerator is a counter for bytes produced at \( i \), and \( C_i \) is the counter for number of completions from \( i \). The accuracy of estimating the materialized size with \( b_i \times n_i \) depends on the variance of each component’s estimate.

To deal with large datasets, which may take longer than the Plumber tracing time to iterate through, we obtain a subsample, \( s \), of the set of file sizes, \( S \), which we can use to approximate \( S \). If we have \( n \) of \( m \) samples, we can simply rescale the subsampled size, \( \sum_n s \), by \( m/n \) to get an estimate of the full dataset size, \( m/n \times E[\sum_n s] \). Empirically, the estimate is sufficiently accurate: 1% of files gives relative error of 1% ImageNet and 2% for COCO, and 5 file gives less than 2% relative error for WMT datasets. This allows Plumber to get a tight estimate of the true dataset size in seconds with sufficient read parallelism. The normal distribution of error we observe matches intuition that the Central Limit Theorem applies.

B IMPLEMENTATION DETAILS

Plumber is currently implemented with a fast but simple C++ core on top of tf.data (3k line patch) and a higher-level Python interface, analysis, and graph rewriting code (8k lines). Plumber builds on tf.data’s C++ AUTOTUNE harness to collect statistics for each pipeline iterator. For each iterator operator, a struct captures the timer and counter-related statistics, along with additional meta-data totaling less than 144 bytes, and the number of iterator operators is rarely over 1000. We describe how these fields are collected and how they are used to derive pipeline statistics when Plumber ingests the output file (which is small—less than 200kB for ImageNet/ResNet).

Measuring CPU. To measure CPU work, Plumber adds fine-grained CPU-timer support to TensorFlow and uses this support to obtain more accurate statistics. We use a similar architecture to AUTOTUNE, which starts and ends timers at the boundaries of Next calls. There are two advantages to using CPU-timers. First, the sum over dataset time closely tracks CPU utilization, yielding a valid \( \theta \), which in practice bounds the number of generated threads. Preliminary tests showed that summing over wallclock time rather than CPU time can overestimate ImageNet/ResNet pipeline’s utilization by \( 2.5 \times \), with error starting to grow when cores are oversubscribed. Second, datasets that use significant wallclock time but not much CPU time are accounted correctly—examples include datasets that sleep or read from files. There is a cost associated with using this different timer. This overhead, in the order of tens to hundreds of nanoseconds, becomes visible when work is so small that the overhead is not amortized out, typically when CPU utilization is low. While this price is only paid during the tracing stage, we can reduce the overhead via subsampling, adding vDSO plugins, or using wallclock timers.

Graph Rewrites. Applying Graph (a TensorFlow tf.data program in our application) rewrites requires a compiler-like approach. Because a Graph is instantiable, Plumber takes as input a serialized Graph representation of the input pipeline and outputs a modified Graph representation. Plumber builds an in-memory representation of datasets to make decisions, such as what node to parallelize or cache. To connect the internal representations with the Graph, Plumber uses the Dataset name as the key. To be effective, the graph-rewriting utility must implement mechanisms to 1) get a node’s performance parameter (prefetching, parallelism), 2) set a node’s parallelism parameter, and 3) insert a new node after the selected node (caching, prefetching)—each of which is a standard graph operation.

Automatic Optimizer Tool. On top of the same graph-rewriting code-base, we built a user-friendly pipeline optimizer tool, which optimizes over parallelism, prefetching, and caching by rewriting the Graph. The pipeline optimizer is not only more extensive in its automatic optimizations, but is meant to be faster than interactively finding bottlenecks. For the pipeline-optimizer tool, there are three logical passes: LP-parallelism optimization, prefetching insertion, and caching insertion. Each of these logical passes can be executed independently or fused; by default Plumber does 2 iterations of the passes, so that the estimated rates more closely reflect the final pipeline’s performance (recall that the empirical rates shift slightly as parallelism is changed).

Tracing Time. Plumber’s tracing is able to return anytime, but longer tracing covers a larger fraction of the (possibly resampled) dataset. As slower pipelines generally require more tracing time to converge, Plumber provides an option which stops when the difference between successive throughput estimates drops below a small threshold—less than 5 minutes for 1% threshold in the pipelines we observe.

Pick Best Queries. There are cases where two query algorithms implement the same logical goal, with different performance characteristics. For example, JPEG files can be decoded then randomly cropped, or the two can be fused. The tradeoff is the former is amenable to caching (after
decode), while the latter is faster to decode—therefore, the optimal implementation depends on runtime conditions (namely, memory capacity). To get around Plumber’s lack of a logical query system, Plumber allows users to specify multiple pipelines, each with the same signature. In practice, this is on the order of two pipelines (keeping the choice fast), and Plumber will automatically trace both and apply any optimizations it can before picking the fastest pipeline. The optimization shown in Figure 11 is a difficult optimization to perform for an online tuner, like AUTOTUNE, because, even with both pipelines to inspect, the effect of caching does not kick in until a whole epoch into the training. Plumber, knowing that cache cold-start is a factor, can simulate the steady-state effects of caching by truncating the cached data with advanced rewriting, yet still return the normal dataset. We use these annotations for our end-to-end ResNet experiments, which feature the fused decode and crop example in the figure.

B.1 Detecting Random UDFs for Caching

For caching operators, Plumber must reason about both size and correctness constraints. In addition to size constraints (§4), an operation may not be cacheable if it is randomized; thus, Plumber performs additional analysis to see if the transitive closure of the UDFs defined in an operation is random. Specifically, we are interested in the relation: if a function, \( f \), accesses a random seed, \( s \), \( f \rightarrow s \). The transitive closure, \( f \stackrel{\rightarrow}{\rightarrow} s \), measures if any child functions of \( f \) touch a random seed. If \( f \stackrel{\rightarrow}{\rightarrow} s \) is true, then we cannot cache \( f \) or any operations following it. This simple relation can be computed via a graph traversal and holds nearly always, as random seeds are necessary in implementations that enable determinism for reproducibility.

Figure 11: Python pseudo-code for optimizing a pipeline via annotations. Plumber will trace both caching paths and pick the one best suited for runtime conditions (e.g., memory allows caching). We note that the cached-path will typically have user-defined caching, though Plumber discards such performance-optimizations as suggestions and inserts them itself, if possible, for each of the different code-paths—avoiding memory errors and duplicate cache.

Figure 12: Absolute throughputs in samples per second of the end-to-end model training experiments on TPUv3-8.

(a) MultiBoxSSD (Setup A)  (b) MultiBoxSSD (Setup B)

Figure 13: MultiBoxSSD with local perturbations. Plumber is able to find optimal Dataset choices when Datasets are different with respect to performance. MultiBoxSSD exhibits two bottlenecks alternating every 4 steps, resulting in confusion at the steps. There are diminishing returns to pursuing an existing bottleneck, which slightly lag behind the rates measured.

C ADDITIONAL EXPERIMENTAL DETAILS.

We provide additional experiments probing local behavior in this section, the cost of profiling, and absolute speedup graphs of the end-to-end results provided in the main text.

C.1 End-to-End.

The absolute end-to-end results are shown in Figure 12. We include a MultiBoxSSD experiment, MultiBoxSSD (48), with only half of the cores enabled for scheduling, demonstrating additional gains for Plumber when less resources are available. As mentioned in the main text, ResNetLinear is compared to conservatively because AUTOTUNE sometimes allocates only 1 parallelism to reading I/O, which results in only 6012 images/second rather than the 9230 images/second provided by using the suggested pipeline parameters for file I/O.

C.2 Local Behavior.

While Plumber may be able to converge to a good solution 2–3× faster than an uninformed user, it is worth knowing how optimal are Plumber’s predictions. While we cannot enumerate all possible convergence paths, we can test
if there was a better Dataset selection to be made by Plumber. To test this, we sample three one-step deviations from Plumber's recommended action. We highlight MultiBoxSSD in Figure 13, which exhibits many transitions between bottlenecks. For MultiBoxSSD, we observe similar bottleneck behavior as ResNet; Plumber prioritizes the image processing operation and alternates to the TFRecord parsing operation every 4 steps (nearly twice as often as ResNet cycles). For ResNet, we observe local optimality except at the bottleneck transitions (once every 8 steps). For the other datasets, we don’t observe any significant mispredictions before convergence. Across the datasets, we observe that both transition regions and resource saturation cause measured rates to become correlated—the rates begin spiking/oscillating together, which makes rankingDatasets unpredictable On ResNet, for example, we observe oscillations begin at 90% of peak performance. Therefore, choosing the bottleneck node among similarly bottlenecked nodes is ambiguous. For workloads with larger dynamic ranges between slower nodes, performance is nearly always optimal.

C.3 Cost of Profiling

We evaluate the cost of running Plumber compared to an unmodified TensorFlow 2.4.1 build using the HEURISTIC configuration, which 1) stresses the system and 2) avoids AUTOTUNE-related overheads. On setup A, the average slowdown across the 5 pipelines is 5% and is driven entirely by Transformer/GNMT, which have a slowdown of 19%/21%, respectively. Thus, the tracing overhead grows with less work per element (motivating a batched execution engine). On setup B, the story is similar, though the effect is more pronounced (likely due to increased overhead from timer system calls); average slowdown is 10% and is 17%/36% across Transformer/GNMT, respectively. The overhead is still lower than the TensorFlow Profiler, which incurs a 7–15% slowdown on vision workloads and 40% slowdown on the text workloads on system A.

D DATASET DETAILS.

We summarize the MLPerf datasets below. ResNet50 (He et al., 2016) is an image classification task selecting between 1000 categories of the ImageNet (Deng et al., 2009) dataset. Mask RCNN (Ren et al., 2016) is an object detection and segmentation task over the MSCOCO (Lin et al., 2014) dataset. MultiBoxSSD (Liu et al., 2016) is a real-time object detection and segmentation task over the MSCOCO dataset. Transformer (Vaswani et al., 2017) is a neural machine translation task, which converts a sequence of words from a source to a target language using the WMT English-to-German dataset (Second Conference on Machine Translation, 2017). GNMT (Wu et al., 2016) is a task similar to Transformer using the WMT 2016 (First Conference on Machine Translation, 2016) dataset and a different pipeline. The ImageNet dataset is 148GB, COCO is 20GB for both MaskRCNN and MultiBoxSSD, the processed WMT data for Transformer is 1.2 GB, and the processed WMT data for GNMT is 1.9GB.

E ARTIFACT APPENDIX

E.1 Abstract

We provide three logical artifacts: a TensorFlow fork, an application-layer Python library on top of TensorFlow, and example pipelines used for evaluation. These artifacts allow a user to use Plumber to diagnose their own pipelines as well as recreate the main results of the paper. To validate the functionality of the TensorFlow and Python library, we provide a simple notebook that requires minimal resources and can be run quickly. To validate microbenchmark functionality, a machine with 16GiB RAM and a 4+ core CPU is sufficient, while end-to-end validation requires access to (proprietary) Google Cloud TPUs. Microbenchmarks and end-to-end runs require running a data pipeline with optional model attached for a number of iterations, and the main metric is the rate at which those iterations are processed (discussed in paper). The primary skills necessary are experience with TensorFlow and JAX and familiarity with tf.data in addition to TPUs.

E.2 Artifact Check-List (Meta-Information)

- Program: Python, C++
- Compilation: C++, Bazel
- Data set: ImageNet, COCO, WMT16, WMT17
- Hardware: CPU-only server, TPUv3–8 VM
- Metrics: Throughput
- Experiments: Machine Learning Training, Microbenchmarks
- How much disk space required (approximately)?: 250GB
- How much time is needed to prepare workflow (approximately)?: A few hours to prepare the datasets (e.g., TFRecord conversion) and a few hours to compile TensorFlow.
- How much time is needed to complete experiments (approximately)?: The microbenchmarks complete in a few hours each, with ResNet/ImageNet being the slowest, due to the number of optimization steps it takes. The end-to-end results can finish in a few hours each, except for RCNN, which can take about a day due to long epochs.
- Publicly available?: Yes
- Code licenses (if publicly available)?: Apache License 2.0
E.3 Description

E.3.1 How Delivered

We provide two open-source GitHub repositories. The TensorFlow fork is available at: https://github.com/mkuchnik/PlumberTensorflow. The Plumber Python application-level library and experiment files are available at: https://github.com/mkuchnik/PlumberApp. The latter repository contains the instructions for using both the Plumber TensorFlow fork and the Python application-level library.

E.3.2 Hardware Dependencies

Microbenchmarks should run on any hardware that runs TensorFlow. However, end-to-end results require the use of Google Cloud TPU VMs (TPUv3–8).

E.3.3 Software Dependencies

We use JAX, TensorFlow, Python, and a proprietary but publicly available library for compiling against TPUs. A non-exhaustive list of open-source Python packages is: numpy, matplotlib, seaborn, cvxpy, and graphsurgeon.

E.3.4 Data Sets

We use ImageNet (Deng et al., 2009), COCO (Lin et al., 2014), WMT16 (EN-DE) (First Conference on Machine Translation, 2016), and WMT17 (EN-DE) (Second Conference on Machine Translation, 2017).

E.4 Installation

Build and install TensorFlow using the publicly available documentation. The Python library’s source code is packaged to be installed as a Python wheel.

E.5 Experiment Workflow

For microbenchmarks, run an iterative graph-rewriting script that compares random optimizations to Plumber’s recommendations. The throughput of the pipeline is measured (e.g., minibatches per second) as well as the estimated rate according to Plumber and AUTOTUNE. These curves are then compared to naive and strong baseline methods (HEURISTIC and AUTOTUNE) of tuning the pipeline. For end-to-end experiments, run a subset of ML training using naive, baseline methods, and Plumber’s optimized pipeline. The throughput of the end-to-end training is measured (e.g., minibatches per second).

E.6 Evaluation and Expected Result

For microbenchmarks, the throughput curve of random optimizations should be worse than that of Plumber’s recommendations. For throughput, Plumber should beat naive policies and match strong baselines. For estimated rates, Plumber’s estimated rates should be bounded, unlike AUTOTUNE’s. For end-to-end results, Plumber should match or exceed strong baselines on most pipelines.

E.7 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging