Lotus: Characterization of Machine Learning Preprocessing Pipelines via Framework and Hardware Profiling

Rajveer Bachkaniwala
Georgia Tech
rr@gatech.edu

Harshith Lanka
Georgia Tech
hlanka3@gatech.edu

Kexin Rong
Georgia Tech
krong@gatech.edu

Ada Gavrilovska
Georgia Tech
ada@cc.gatech.edu

***Abstract*—Preprocessing input data is a crucial step in machine learning pipelines, involving tasks such as loading, decoding, and applying transformations. Prior works have identified preprocessing as a performance bottleneck for ML training jobs and introduced optimizations like CPU and I/O parallelism, accelerator offloading, and on-node/distributed computing as well as caching to mitigate this bottleneck. However, there is a lack of support for characterizing preprocessing pipelines at a finer granularity, especially at the microarchitecture level, which can provide insights to validate and inform the design of existing and future optimization techniques.**

To enable these insights, we introduce Lotus, a profiling tool for the preprocessing stage of ML pipelines. Firstly, it captures fine-grained preprocessing events (e.g., <10 ms) with minimal time and storage overheads. Secondly, it bridges the gap between high-level Python functions and low-level hardware performance counters by reconstructing a mapping between Python functions and the underlying C++ functions they invoke. This unique combination enables users to better reason about their pipeline's performance at both the framework and CPU architecture levels. We demonstrate the insights made possible by applying Lotus to representative ML workloads and compare its capabilities, overheads, and ease of use with alternative profilers.

***Index Terms*—ML preprocessing, Python profiling, hardware instrumentation**

I. Introduction

Preprocessing is a crucial step in machine learning (ML) pipelines that ingest and transform raw input data into a format suitable for ML models. It often consists of a chain of complex operations, such as loading, decoding, and applying transformations, which can require significant compute time. For example, preprocessing can consume up to 65% of the epoch time in applications like image classification, object detection, and audio classification [1]. Preprocessing performance is crucial for ML training jobs, which demand low latency (100 μs - 1 ms) and high throughput (10 GB/s) for per-batch generation [2]. Inefficiencies in CPU-based preprocessing can lead to low compute utilization on expensive accelerators, especially in systems with a CPU-to-accelerator ratio imbalance [1], [3]–[7].

This project was partially supported by the Intel Center on Transformative Server Architecture via the TRIM project, and the SRC/DARPA JUMP 2.0 Center for Processing with Intelligent Storage and Memories (PRISM).

Prior works have introduced numerous optimizations to improve preprocessing performance, including parallelizing I/O and compute in and across batches [8]–[10], accelerator offloading (DALI [11], TrainBox [12]), data duplication [13], caching optimization [1], [7], [10], [14]–[16], dataset storage optimization [14], [17], disaggregated preprocessing across nodes [2], [7], [15], [18]–[20] and co-locating ML jobs for effective caching and scheduling in a cluster [18], [21], [22].

Effective optimizations rely on understanding the performance bottlenecks in preprocessing pipelines, and better understanding can in turn lead to new optimization opportunities. However, there is a lack of adequate profiling tools that can effectively characterize the performance implications of preprocessing pipelines at the CPU architectural level. Current profiling capabilities face two main limitations.

First, there is a disconnect between the performance of high-level Python functions and low-level hardware metrics (such as L1 cache misses) that are collected via performance counters. Existing hardware profilers, such as Intel VTune and AMD uProf, collect CPU cache and microarchitecture performance data for C/C++ functions but cannot capture stack frames of machine learning pipeline code written in Python. Additionally, Python profilers that capture the call stack often fail to label preprocessing functions correctly, forcing users to investigate the source code manually to recreate the stack trace.

Second, capturing fine-grained batch-level preprocessing timing data with low overhead is challenging. Sampling-based Python profilers like Scalene [23], py-spy [24], and austin [25] are constrained by their sampling rates, making it challenging to capture the duration of individual transformation operations that may only take hundreds of microseconds to a few milliseconds without incurring significant overhead. Moreover, the asynchronous data flow used in many preprocessing frameworks, where worker processes execute the actual preprocessing operations while the main process coordinates, complicates the measurement of elapsed times. Recent work on optimizing preprocessing pipelines [8]–[10] rely on instrumentation to capture aggregated elapsed time across many batches, but does not capture fine-grained per-batch statistics or data flow dependencies.

2835-2238/24/$31.00 ©2024 IEEE
DOI 10.1109/IISWC63097.2024.00013

To address these limitations, we make two key observations. First, ML preprocessing pipelines are often declaratively defined, providing hooks for fine-grained instrumentation while ensuring generalizability across different pipelines and frameworks [6], [8], [26]. Second, once such fine-grained instrumentation data is available, it can be leveraged to better attribute low-level hardware performance counters measured by the hardware profilers to the corresponding high-level preprocessing functions.

We leverage these insights to build **Lotus** – a new profiling tool for ML preprocessing pipelines declared using PyTorch's DataLoader [26]. Lotus comprises two components, LotusTrace, and LotusMap, that enable capturing preprocessing events and hardware analysis for preprocessing operations, respectively. The effectiveness of LotusTrace is due to the understanding of the PyTorch DataLoader's asynchronous data flow, which allows us to add logging instrumentation at the points that matter the most in capturing this flow (§ III-B). As a result, LotusTrace neither performs additional computation nor maintains unnecessary tracer state in memory, thus avoiding CPU and memory overheads. On the other hand, LotusMap introduces a novel technique that approximates the mapping of Python functions to their C/C++ counterparts. To obtain a high-quality mapping, our technique carefully buckets the C/C++ functions, filters incorrect C/C++ functions, and captures short-lived C/C++ functions (§ IV-B).

Together, LotusTrace and LotusMap allow a practitioner to identify the most time-consuming preprocessing operations, map them to the responsible C/C++ functions, and use their hardware performance counters to get a CPU architectural level performance view of the preprocessing operations. Lotus thus empowers users to reason about the performance of preprocessing pipelines at the hardware level, bridging a significant gap in our understanding.

In summary, we make the following contributions:

- We introduce Lotus, the first tool for fine-grained instrumentation and profiling of ML preprocessing pipelines at the level of the preprocessing framework and the CPU architecture.
- Using Lotus, we characterize three representative machine learning pipelines from MLPerf's training benchmark [27] and share insights into their performance characteristics and potential optimization opportunities.
- We compare Lotus with alternative profilers on profiling overhead and functionality.

Lotus is an open-source tool [28]. Our current implementation targets PyTorch's DataLoader preprocessing library [26], and the Intel VTune [29] and AMD uProf [30] hardware profilers. However, the methodology also applies to other preprocessing frameworks that allow declaratively specified preprocessing pipeline [8], [11]. In addition, preprocessing pipelines defined in PyTorch DataLoader can be seamlessly integrated with major ML training backends, such as TensorFlow, which consume data through iterators.

```
import torchvision.transforms as transforms
import torchvision.datasets as datasets
custom_log_file = <To use our instrumentation>
train_dataset = datasets.ImageFolder(
    traindir,
    transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225])
    ], log_transform_elapsed_time=custom_log_file),
    log_file=custom_log_file
)
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=args.batch_size,
    shuffle=(train_sampler is None) and args.shuffle,
    num_workers=args.workers,
    pin_memory=True,
    sampler=train_sampler,
)
```

Listing 1: Example image preprocessing pipeline in PyTorch.

II. Background

We provide a brief background on how preprocessing is specified and implemented in PyTorch.

A. Defining Preprocessing Pipelines

Several popular machine learning libraries, including PyTorch's `torchvision`, Tensorflow's `tf.data`, and NVIDIA's DALI, provide support for specifying preprocessing pipelines in a declarative manner. For example, `torchvision` offers a general API called `torchvision.transforms.Compose` for defining preprocessing pipelines. The preprocessing pipeline is declaratively defined by chaining together a sequence of preprocessing operations to be applied on each input image, such as shown in lines 6-13 of Listing 1. This pipeline contains four operations: RandomResizedCrop(), RandomHorizontalFlip(), ToTensor() and Normalize(). PyTorch also provides `torch.utils.data.DataLoader`, a utility class to help simplify the process of loading data (Line 15-22 of Listing 1). Users can specify parameters such as the number of workers needed, batch size, and prefetching options. Functions related to image reading and decoding are performed by the `torchvision.datasets` API internally. The API uses the appropriate image reader and decoder based on the image type, detected from the metadata of the image. The user can also define their own image reader and decoder.

B. Data Flow in PyTorch

The data flow in a single node, multi-GPU setting is described for the PyTorch `torch.nn.DataParallel` API.

Between the main process and the Dataloader workers. The main process forks DataLoader workers equal to the `num_workers` parameter. The main process is responsible for coordinating preprocessing work with the DataLoader workers. Each DataLoader is tasked with preprocessing a batch of data. Communication between the main process and DataLoader workers on a node occurs via Python's `multiprocessing.Queue`, which is internally implemented using shared memory.

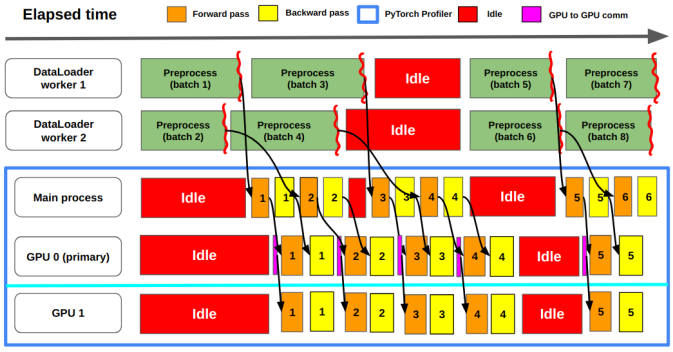

Fig. 1. PyTorch's program flow, where arrows denote data flow between the main process and GPUs and between the DataLoader workers and the main process. The PyTorch profiler captures information enclosed in the blue box while ignoring events happening on the DataLoader workers.

There are two types of communication channels between the main process and the workers: (a) data queue and (b) index queues. The index queues, one per worker, send indices of data to be processed from the main process to the worker processes (*i.e.*, the main process is the producer and the worker is the consumer). The data queue, shared among all workers and the main process, sends preprocessed data from the workers to the main process (*i.e.*, the worker is the producer and the main process is the consumer).

Users can activate prefetching by setting `prefetch_factor` to a value greater than 0 when declaring the DataLoader object, default is 2. Initially, the main process places batches of indices equal to the prefetch factor into the index queues for each worker. Prefetching is performed only at the start of the training process. Subsequently, before consuming the desired batch, the main process places a single batch of indices to the DataLoader worker which produced the desired batch. This batch's id is greater than the batch id of the previous DataLoader worker.

Between the main process and the GPUs. After fetching the desired batch from the data queue, the main process transfers the preprocessed data to the GPUs asynchronously. In the `DataParallel` setup, the data is split across available GPUs by the GPU that received the data from the main process. Subsequently, the main process schedules forward and backward pass GPU kernels asynchronously. The main process then waits for the next batch of preprocessed data from the DataLoader workers.

III. LotusTrace: Enabling Timing Analysis

Measuring the elapsed time of preprocessing operations within a machine learning pipeline is essential for understanding preprocessing performance. However, we found that existing Python profilers struggle to provide the following crucial elapsed time measurements with low overhead (§ VI-B):

[T1] Total preprocessing time for a specific batch
[T2] Time the main process spent waiting for a specific batch to finish being preprocessed by a DataLoader worker
[T3] Time taken by each preprocessing operation in a batch

```
log_file = <To use our instrumentation>
transforms = transforms.Compose([op1(), op2(), op3(), op4()],
  log_transform_elapsed_time=log_file)
class CustomDataset:
  def __init__(self, log_file = None, transforms):
    ...
    self.log_file = log_file # If None, then no logging
    self.transforms = transforms # A Compose object
    ...
  def __getitem__(self, index):
    ...
    # Calls Compose's __call__()
    data,label = self.transforms(index)
    ...
    return data, label
dataset = CustomDataset(log_file = log_file,\
          transforms = transforms)
```

Listing 2: LotusTrace supports custom datasets. Users just need to provide a `log_file` and a `Compose` object and initialize them in the `__init__` method of the custom dataset class. The `__getitem__` method should call the `Compose` object's `__call__` method to apply the transformations.

We introduce LotusTrace, Lotus's profiling methodology that addresses these challenges by instrumenting the PyTorch DataLoader and torchvision libraries (§ III-B). In addition, the collected data can be used to visualize the interaction between the main process and DataLoader workers to aid in understanding the data flow in the pipeline (§ III-C) which allows insights into performance problems (§ V-B).

A. Challenges in Measuring Elapsed Time

The asynchronous nature of the data flow (§ II-B) poses a challenge for both trace-based profilers like PyTorch profiler [31] and sampling-based profilers like py-spy [24] in capturing crucial asynchronous interactions. For instance, the PyTorch profiler only captures events in the main process and GPU operations (information within the blue box shown in Figure 1), reporting preprocessing time as the main process's wait time for DataLoader workers (red "idle" boxes). This differs from the actual CPU time spent on preprocessing (green boxes). Another challenge is the efficiency of sampling-based Python profilers in capturing short-duration preprocessing operations, with an average elapsed time in hundreds of microseconds (Table II). Increasing the sampling rate of the sampling-based profilers lead to a non-trivial time, storage, or memory overhead (§ VI-B).

B. LotusTrace Instrumentation Methodology

LotusTrace is a lightweight tracing tool that instruments PyTorch DataLoader and torchvision library to track program flow and events such as batch and preprocessing operations. It minimizes performance impact by limiting tracing to two timing measurements per event/operation achieving a per-log overhead of ~200μs on our setup.

Using LotusTrace is simple, requiring only a custom PyTorch build without significant API changes. Users enable profiling by specifying a log file in the `Compose` and `ImageFolder` APIs, as shown in Listing 1. LotusTrace also supports custom datasets (subclasses of `torch.utils.data.Dataset`), such as shown in Listing 2. In the evaluation, we demonstrate

```
log = ""
for t in self.transforms:
    start = time.time_ns()
    img = t(img)
    duration = time.time_ns() - start
    log += (f"S{t.__class__.__name__}, {start},{duration}\n")
```

Listing 3: Measuring elapsed time for each transform inside the torchvision.transforms.Compose API's __call__().

LotusTrace's utility to trace different ML pipelines from the MLPerf training benchmark [27] with minimal code modifications (§ VI-C).

1) Timing Instrumentation: To capture the total preprocessing time per batch **[T1]**, we measure the time taken by the `fetch` method that is called inside the DataLoader worker loop. The main process forks DataLoader workers and runs them inside a `worker_loop`. Inside this loop, a dataset fetcher object is created, which is responsible for returning a batch of preprocessed data when its `fetch` method is called. An alternative approach could involve subclassing or overriding the dataset fetcher, this requires knowing the specific fetcher class in use (e.g., `_MapDatasetFetcher` or `_IterableDatasetFetcher`). Instead, our solution targets the common `fetch` method across all fetcher classes, avoiding the need for class-specific modifications.

For the main process's wait time **[T2]**, we add timing instrumentations around where `_next_data` is requested. The main process waits on a blocking operation `self._get_data()` until some batch arrives, which denotes the `end_wait`. One issue with measuring wait time is that the batches can arrive out-of-order in the shared data queue. Since the main process consumes batches in order, it has to pin (to the CPU memory) and cache out-of-order batches. To distinguish these out-of-order batches, they are marked with a timestamp and duration of 1 μs to denote no waiting. In contrast, the subclass/override approach would interfere with other DataLoader functions beyond timing, such as process tracking and batch assignment, requiring users to rewrite the complex and non-modular DataLoader core logic.

To measure elapsed time for each preprocessing operation **[T3]**, we instrument the `__call__` method of `torchvision.transforms.Compose` (Listing 3). The `__call__` method calls each transform in the specified order by looping over the transform set. `t.__class__.__name__` gives the name of the transform class (*e.g.*, RandomResizedCrop). By wrapping instrumentation around `t()`, we can measure the elapsed time for arbitrary preprocessing operations declared using the `Compose` API, provided that the corresponding operations class has a defined `__call__` method inside which the operation is performed.

2) Logging Instrumentation: We log metadata such as batch and process IDs alongside timings to associate logs with the specific DataLoader process responsible for preprocessing each batch. For **[T1]**, we capture the batch ID using `self.index_queue` and the process ID using the `psutil` library. For **[T2]**, we get the cached process ID of the main process when a DataLoader instance is created. For **[T3]**, we log the elapsed time for each transform in the `__call__` method of `torchvision.transforms.Compose`. Note that `psutil.Process().pid` has to be called to obtain the `pid` of the DataLoader process running, because the dataset object is shared between the main process and the other DataLoader worker processes.

C. Visualization of Collected Traces

LotusTrace augments the collected traces to visualize preprocessing times and the data flow between the main process and DataLoader workers to produce traces as shown in Figure 2. It supports visualization at batch level (coarse) and batch + per op level (finer) granularities. LotusTrace captures the reference start timestamp, duration, batch ID, and process ID for each operation, which can be used to visualize spans (rectangular boxes) and track batch progress. The trace has three spans namely: 1) `SBatchPreprocessed_idx` - Preprocessing span for batch `idx`, 2) `SBatchWait_idx` - The main process' wait time span for batch `idx` to be ready, and 3) `SBatchConsumed_idx` - The consumption of batch `idx` by the main process. To visualize the flow of events, we augment the logs to generate an arrow from the span of `SBatchPreprocessed_idx` in the DataLoader process to its corresponding `SBatchConsumed_idx` marker in the main process. § V-B provides examples of the visualizations and generated insights in more detail.

LotusTrace can generate a standalone trace file or augment PyTorch profiler's trace data, both compatible with Chrome Trace Viewer (format used by PyTorch profiler). To combine LotusTrace and PyTorch profiler data in a single visualization, LotusTrace generates tracing logs in a JSON format following that of the PyTorch profiler. To avoid collisions among the LotusTrace and existing PyTorch profiler's trace data, LotusTrace uses negative `synthetic_ids` to distinguish its logged events from the PyTorch profiler's logged events with positive integer ids.

IV. LotusMap: Enabling Hardware Analysis

Beyond fine-grained elapsed time measurements, it is important to understand how CPU resources, such as CPU microarchitecture, and caches, influence the efficiency of preprocessing operations. To achieve this, Lotus introduces LotusMap, a profiling methodology that connects low-level hardware statistics to high-level Python functions. We demonstrate our methodology for Intel VTune [29] targeting Intel CPUs and AMD uProf [30] targeting AMD CPUs.

A. Challenges in Attributing Hardware Events

Hardware profilers such as Linux perf [32], Intel VTune [29], and AMD uProf [30] collect hardware-level statistics like cache misses and branch mispredictions. These profilers can collect hardware events at the granularity of C/C++ functions called during an application's end-to-end run. However, for Python-based machine learning pipelines, the stack-level information is lost, preventing hardware profilers from associating hardware-level statistics with high-level

```python
import torchvision.transforms as t, time, from PIL import Image
# Pick module according to CPU chip
import <itt or amdprofilecontrol as amd>
# increase PIL image open size
Image.MAX_IMAGE_PIXELS = 1000000000
image_file = "<path to image>"
for i in range(5):
  # Open the image
  image = Image.open(image_file)
  # convert to RGB like torch's pil_loader
  image = image.convert('RGB') # For Loader operation
  # Define the desired crop size
  crop_size = 224  # Define this as needed
  time.sleep(1)  # ensure correct bucketing
  if i == 4: # Delay collection to prevent cold start
    itt.resume() # for Intel, amd.resume(1) for AMD
  image = t.RandomResizedCrop(crop_size)(image)
  if i == 4:
    itt.detach() # for Intel, amd.pause(1) for AMD
```

Listing 4: Example of ITT/AMDProfileControl API use to isolate Python function.

Python functions. Hence, there is no support to isolate the C/C++ functions related to preprocessing operations from the rest of the ML pipeline.

PyTorch libraries are written in C++ and exposed to Python using pybind11 [33]. Until Python 3.11, Python lacked the necessary support for Linux perf to obtain Python frames. Even in Python 3.12, which supports Linux perf, the Python threads and frames that call C/C++ function bindings are lost. Python profilers like py-spy [24] and austin [25] can collect C/C++ functions called by Python frames but lack support for capturing preprocessing operations, as stack frames get labeled as `__call__` instead of actual transformations like `RandomResizedCrop`. This forces users to manually map C/C++ functions to high-level Python functions by examining the source code.

Existing work in Linux perf-map agents, such as the Java perf-map agent [34], takes a different approach by generating dynamic symbol mappings to produce full stack traces. The Java perf-map agent creates a perf map file for Just-In-Time (JIT) symbol translation by using a Java agent written in C, along with a Java bootstrap application that attaches to a running Java process. However, this method is specific to Java. Implementing a similar JIT approach for Python requires modifications to the Python runtime environment [35], leading to increased I/O costs [36]. Furthermore, while both Java perf-map agents and Python runtime modifications aim to capture full stack traces, our approach focuses on isolating and profiling only the leaf C/C++ functions that are critical to preprocessing pipelines in machine learning workloads.

B. Mapping from C++ to Python functions

LotusMap provides a profiling methodology to map C/C++ functions to high-level Python functions and attribute hardware events accordingly, addressing the aforementioned challenges across CPU architectures. This mapping process is a preparatory step that needs to be done once for each Python operation. Once the mapping is obtained, the user can run the hardware profiler on the program as it is. After the job finishes, the C/C++ functions can be mapped to Python functions for further investigation of the job performance as demonstrated in § V-D. Note that the mapping step has to be performed on the same machine as the job run, because the mapping may capture certain C/C++ functions specific to a shared library installed which may differ on different machines based on OS and ISA.

TABLE I
Sample mapping of Python functions to C/C++ functions obtained from Intel (top) and AMD (bottom) chips. Listed only a few for each for brevity. #–_imaging.cpython-310-x86_64-linux-gnu.

Transformation	Function	Library
Image.convert (Loader)	decompress_onepass	libjpeg.so.9
	jpeg_idct_islow	libjpeg.so.9
	jpeg_idct_16x16	libjpeg.so.9
	ycc_rgb_convert	libjpeg.so.9
	decode_mcu	libjpeg.so.9
	ImagingUnpackRGB	Pillow lib#
	__memset_avx2_unaligned_erms	libc.so.6
	__memcpy_avx_unaligned_erms	libc.so.6
	jpeg_fill_bit_buffer	libjpeg.so.9
**Intel-specific*	__libc_calloc	libc.so.6
**AMD-specific*	__memset_avx2_unaligned	libc-2.31.so
**AMD-specific*	_copy	Pillow lib#
**AMD-specific*	process_data_simple_main	libjpeg.so.9
**AMD-specific*	sep_upsample	libjpeg.so.9
RandomResizedCrop	ImagingResampleHorizontal_8bpc	Pillow lib#
	ImagingResampleVertical_8bpc	Pillow lib#
**Intel-specific*	__memmove_avx_unaligned_erms	libc.so.6
**Intel-specific*	_int_free	libc.so.6
**AMD-specific*	__memcpy_avx_unaligned_erms	libc.so.6
**AMD-specific*	precompute_coeffs	Pillow lib#

First, we isolate the C/C++ functions related to preprocessing from the hundreds of unrelated functions in the rest of the machine learning pipeline. Intel VTune provides the Instrumentation and Tracing Technology (ITT) API, while AMD uProf offers the AMDProfileControl API, both of which can isolate the C/C++ code of interest. Since the preprocessing pipeline is written in Python, we use Python bindings for these APIs. We use an open-source Python binding for the ITT API [37] and create a new Python binding for the AMDProfileControl API using pybind11 [33]. This allows us to isolate individual Python functions and profile them separately using Intel VTune and AMD uProf. Listing 4 shows an example of how the ITT/AMDProfileControl APIs can be used to isolate and profile a Python function. Using this, we obtain mappings such as shown in Table I.

However, the ITT/AMDProfileControl API bindings alone cannot guarantee an accurate mapping due to complications introduced by the Intel/AMD's sampling driver. We highlight a few problems and solutions below.

Inconsistent C/C++ functions. Since the sampling driver is limited to sample every 10 *ms* (or 1 *ms* for AMD uProf) in user mode sampling, some short-lived C/C++ functions may not be captured consistently. This inconsistency is also evident for operations like RandomBrightnessAugmentation, which may take a different branch based on a random value. To ensure that all corresponding C/C++ functions are captured,

the operation needs to be run multiple times. We use the following formula to determine the number of runs required for consistent capture of C/C++ functions: $C \geq 1-(1-f/s)^n$, where s is the sampling interval, f is the function span ($0 < f \leq s$), n is the number of runs, and C is the probability of capturing the function at least once. For example, if a C++ function takes f=660 μs, to capture it under s=10ms sampling interval with C=75% probability at least once, we need to run the experiment 20 times according to the formula.

Splitting Hardware Metrics. Hardware profilers like VTune and uProf collect data at the granularity of C/C++ functions, but a single C/C++ function can map to multiple Python preprocessing operations. To attribute hardware metrics to the correct Python operations, we use execution time information from LotusTrace to compute weights for each operation and split the metrics accordingly.

For example, consider the Front-end bound metric in VTune for the C/C++ function `__memmove_avx_unaligned_erms`. The function maps to Python operations `Loader`, `RandomResizedCrop`, and `ToTensor` with overall preprocessing times of L, RRP, and TT, respectively. We compute the weight for `Loader` as $L/(L + RRP + TT)$ and multiply it by the Front-end bound metric to get the proportion attributed to `Loader`. We then multiply the metric for each C/C++ function by its corresponding clock ticks to account for normalization in VTune [38]. AMD uProf has similar issues which can be mitigated by this approach.

The information provided by Lotus allows sophisticated approximation techniques, such as considering the mix of different C/C++ functions in a Python function when determining the weight used to split the hardware performance counters; we leave such optimizations for future work.

Miscellaneous Instrumentation Tricks. The sampling driver might mistakenly associate C/C++ functions from a previous Python function with the current Python function of interest, potentially due to out-of-order (OOO) execution [39]. It is important to correctly bucket operations to ensure that metrics are not allocated to the wrong operations. To address this problem, we explicitly insert `sleep()` before the code of interest (Listing 4, line 14). This creates a time gap between the end of the previous Python function and the beginning of the function of interest. The `sleep()` call is used only during the mapping phase and does not affect the actual machine learning job. Once the mapping is complete, the pipeline is run without the `sleep()` call. We also warm up before collecting data to prevent cold starts from being accounted for in each run (Listing 4, lines 15 and 18). If the Python operation is short-lived, then the operation can be run with a larger input in isolation instead of the pipeline with a small input size after cropping.

V. Workload Characterization with Lotus

We illustrate the observations made possible by using Lotus to profile ML preprocessing pipelines.

A. Workloads and Experiment Setup

We use Lotus to profile the three representative vision training tasks from the MLPerf training benchmark [27]. We do not focus on text benchmarks as they are not traditionally bottlenecked by preprocessing [1].

Image Classification (IC). This pipeline classifies an image to an object. We use MLPerf's reference PyTorch implementation [27], [40], the ImageNet dataset [41], and the ResNet18 [42] model. The pipeline contains the following preprocessing steps: 1) *Loader*: Loading the image from disk to memory and decoding it from compressed formats such as JPEG. 2) *RandomResizedCrop (RRC)*: Adjusting the image to the desired size and then crop. 3) *RandomHorizontalFlip (RHF)*: Obtaining mirror image. 4) *ToTensor (TT)*: Converting images to tensors. 5) *Normalization*: Normalizing to zero mean and unit variance. 6) *Collation(C(k))*: Collating tensors into a batch size of k data elements.

Image Segmentation (IS). This pipeline segments an image and classifies each segment. We use MLPerf's [27] reference PyTorch implementation. We use the kits19 [43] dataset and a variant of U-Net3D [44] as the model. The pipeline contains the following preprocessing steps: 1) *Load*: Loading the data in numpy from disk to memory. 2) *RandBalancedCrop (RBC)*: Foreground-aware cropping based on a sampling parameter. 3) *RandomFlip (RF)*: Reversing elements along a tensor's axis. 4) *Cast*: Casting the tensor from float32 to uint8. 5) *RandomBrightnessAugmentation (RBA)*: Adjusting brightness. 6) *GaussianNoise (GN)*: Adding gaussian noise. 7) *Collation(C(k))*: Coalescing tensors to a batch of size k.

Object Detection (OD). This pipeline creates bounding boxes around objects in an image. We use MLPerf's [27] reference PyTorch implementation with preprocessing steps similar to IC, except using resizing instead of resizing and cropping. We use the MS COCO dataset [45], and GeneralizedRCNN [27] (Mask R-CNN [46] with a ResNet-50 [42] backbone) as the machine learning model.

In our setup, preprocessing operations (including reading and decoding images) are CPU-based, whereas forward and backward passes on the deep learning model are GPU-based. All experiments in the above pipelines are performed for one epoch. Validation is not performed in any of the above training pipelines. For IS and OD, we use the default configurations in the reference implementation, which has a batch size of 2, one GPU, and 8 and 4 data loaders respectively. For IC, Table II has batch size 128, one GPU, and one dataloader, and Figure 2 (a) has batch size 1024, 4 GPUs, and 4 dataloaders.

Environment. The experiments are conducted on a CloudLab c4130 node [47], a dual-socket 3.2GHz E5-2667 Intel Xeon CPU, with 128 GiB of RAM, four NVIDIA V100 GPUs, each with 16 GiB memory and NVLink support, and a remote dataset mounted to a single node [48] as a ZFS zvol exported via iSCSI [49]. The software environment includes Python 3.10, PyTorch 2.0.1 with Torchvision 0.15, image processing using libjpeg-9e, GPU acceleration through CUDA 11.8 and

TABLE II

TOP HALF: ELAPSED TIME (IN MS) PER PREPROCESSING OPERATION FOR AN IMAGE. BOTTOM HALF: PERCENTAGE OF PREPROCESSING OPERATIONS WITH ELAPSED TIME LESS THAN 10 *ms* AND 100 μs.

IC	Loader	RRC	RHF	TT	Normalize	C(128)
Avg	4.76	1.11	0.06	0.34	0.21	49.76
P90	6.02	1.39	0.08	0.39	0.23	52.49
<10*ms*	97.79	99.82	100	100	100	~0
<100μs	0	0	98.3	0	0	0

IS	Loader	RBC	RF	Cast	RBA	GN	C(2)
Avg	72.03	91.10	4.39	2.16	0.78	6.46	14.24
P90	130.94	298.62	8.84	4.32	4.66	54.54	15.81
<10*ms*	0	63.69	95.23	98.21	98.8	88.69	0
<100μs	0	61.30	28.57	0	88.69	88.69	0

OD	Loader	Resize	RHF	TT	Normalize	C(2)
Avg	9.59	9.43	0.52	6.75	7.8	7.39
P90	15.57	11.56	1.13	12.86	12.6	10.44
<10*ms*	58.46	76.54	100	87.68	79.96	87.13
<100μs	0	0	49.96	0	0	0

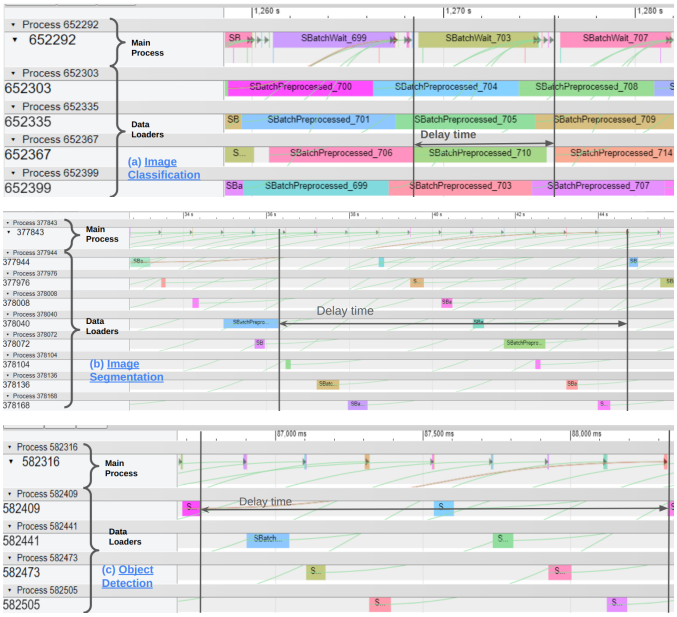

Fig. 2. [Coarse traces] – For (a), the preprocessing is the bottleneck leading to a comparatively smaller delay time, whereas for (b) and (c), the GPU processing is the bottleneck leading to a larger delay time

cuDNN 8.7. The system ran on Ubuntu 20.04 with kernel version 5.4.0-139-generic.

B. Observations from LOTUSTRACE Tracing

Table II reports per image average and 90th percentile elapsed time for each preprocessing operation as well as the percentage of preprocessing operations in the workload with elapsed time less than 10 *ms* and even 100 μs across the three MLPerf pipelines.

Takeaway 1: *All pipelines have operations with short elapsed times under 10 ms (even 100μs), which would have been challenging to capture with sampling-based profilers. By enabling timing measurements for each operation per image, rather than just aggregates, LOTUSTRACE reveals high time variability in certain operations (e.g., RBC in IS and Loader in OD). No single operation dominates the elapsed time, requiring comprehensive profiling of all operations.*

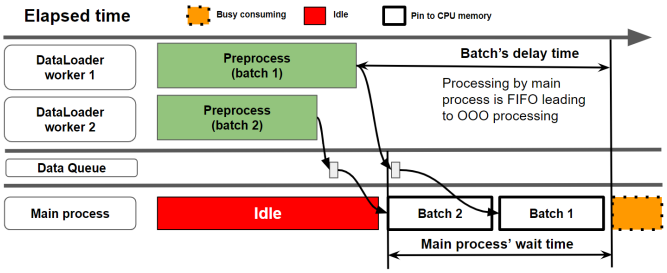

Fig. 3. Out-of-order arrival can cause the main process to wait despite the desired batch being ready.

Figure 2 visualizes the LOTUSTRACE data, showing the timeline of operations in the main process (first row) and data loader processes. Each colored span represents an event's duration. We discuss bottlenecks using two key metrics (illustrated in Figure 3): wait time, the time the main process is idle while waiting for a preprocessed batch, and delay time, the time a batch waits after being preprocessed and before being consumed. In Figure 2(a), `SBatchWait_699` shows the main process wait time before consuming batch 699, while the arrow from `SBatchPreprocessed_699` to `SBatchConsumed_699` shows the delay time for that batch.

The three pipelines exhibit different bottlenecks. In IC, preprocessing is the bottleneck, causing short delay times. IS and OD have long delay times of 10.9 s and 1.64 s for nearly all batches, much longer than their GPU processing times of 750 ms and 250 ms respectively, indicating a GPU processing bottleneck with batches waiting for GPU availability.

These differences stem from MLPerf's use of offline and online preprocessing. In IS and OD, some preprocessing steps are applied to the raw dataset *before* training, which helps avoid bottlenecks during training. In these pipelines, none of the batches wait longer than the GPU processing time, confirming the GPU bottleneck. The parallel preprocessed batches appear sequential as a result, as seen by the non-overlapped colored `SBatchPreprocessed` boxes in Figure 2(b) and (c). IC does not decode and convert image data to numpy format a priori and exhibits a preprocessing bottleneck during training. Figure 2(a) shows parallel preprocessing on the data loader processes.

Takeaway 2: *Training benchmarks that are optimized for time-to-accuracy apply some preprocessing operations on the raw dataset before training to avoid getting bottlenecked by preprocessing during training. When GPU processing is the bottleneck, parallel preprocessing appears sequential in the trace. LOTUSTRACE's data flow visualization between the main process and data loader workers for each batch helps to explain these preprocessing bottlenecks.*

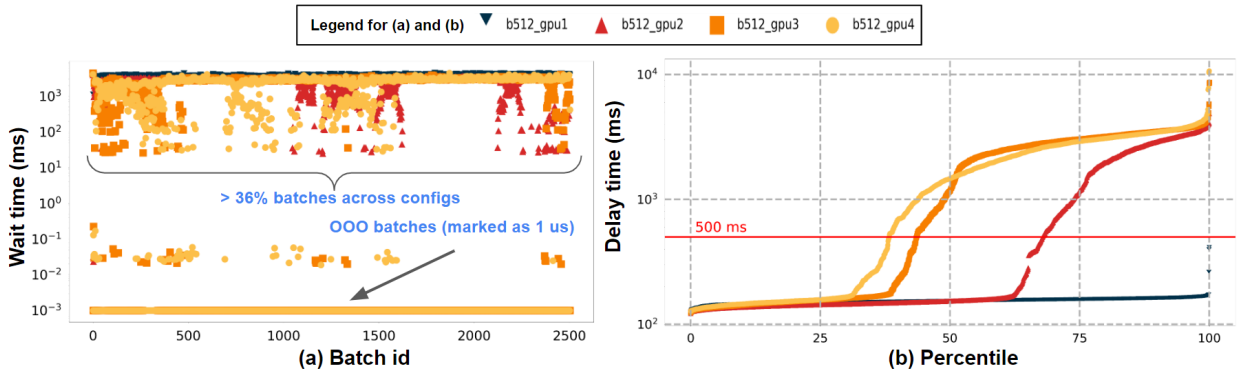

Fig. 5. (a) The main process has to wait for at least 1/3rd of the batches for >500 *ms*. (b) The batch delay time ranges from 32.1% to 61.6% for >500 *ms* except for batch size 512, GPU 1.

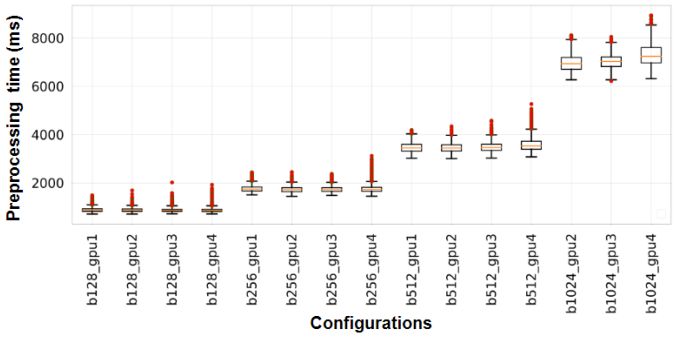

Fig. 4. Preprocessing time per batch has high variance.

C. Observations from Timing Analysis

We delve further into the IC pipeline, which exhibits a preprocessing bottleneck. Our analysis reveals two key findings enabled by LotusTrace: high variance in per-batch preprocessing time and significant main process wait time and batch delay time due to out-of-order arrivals.

1) ***High variance in preprocessing time****:* We run the IC pipeline under varying batch sizes from $b \in \{128, 256, 512, 1024\}$, number of GPUs $g \in \{1, 2, 3, 4\}$, with the number of data loaders set equal to the number of GPUs. Figure 4 reports the per-batch preprocessing time across these configurations. Overall, we observe a high variance in preprocessing time, with the standard deviation per config ranging from 5.48% to 10.73% of the per-config average. This variability becomes more pronounced with larger batch sizes: the Inter Quartile Range (IQR) increases by up to 6.9× when comparing smaller batch sizes (128) to larger ones (1024). IS and OD have similar variability with a standard deviation of 15.47% and 66.8% respectively over the average. This variability is primarily attributed to two factors: the diverse sizes of images in the ImageNet dataset (mean file size of 111 KB and a standard deviation of 133 KB), and the randomness of preprocessing operations. Per-batch elapsed time measurement is unique to LotusTrace due to challenges related to PyTorch's data flow (§ III-B).

High variability in preprocessing time presents significant challenges in resource provisioning. Extrapolating the preprocessing times of a few batches for resource allocation could result in consistent underutilization or overutilization of computational resources. An alternative strategy of batching images of similar sizes to reduce variability is also not ideal, as it could compromise the randomness essential in ML training pipelines. One recent work, SpeedyLoader [50], attempts to tackle this issue by load-balancing input data, albeit limited to the characteristics of a single workload (IS). This provisioning challenge underscores the need for fine-grained performance characterization for any given preprocessing workload, which LotusTrace provides.

Takeaway 3: *Variations in input data sizes contribute to the high variability of the observed per-batch preprocessing time. LotusTrace's fine-grained measurements capture this variability on a per-batch granularity and can aid in resource provisioning challenges.*

2) ***Significant wait and delay time****:* To further investigate the preprocessing bottleneck, we analyze the wait and delay time for a specific batch size of 512. Figure 5 (a) shows that the main process waits over 500 *ms* for 30.84% to 100% of the batches, which exceeds the maximum processing time of a batch on the GPU for this configuration. This indicates that the GPU stalls due to preprocessing. In addition, the preprocessed batches experience significant delay time. Figure 5 (b) shows that when using more than one dataloader, 32.1% to 61.6% of batches experience a delay time of over 500 *ms*.

LotusTrace revealed that out-of-order batch arrivals, caused by the shared data queue among multiple data loaders, significantly contribute to the large wait and delay time. The main process, operating on a single thread, processes one batch at a time. If the desired batch is not at the front of the queue, the main process pins the first batch in the queue to CPU memory and continues to poll the data queue until the desired batch arrives at the front. For example, in Figure 3, DataLoader 1 finishes preprocessing and puts the batch in the shared queue, but the main process is occupied with pinning a batch from DataLoader 2, and the batch from DataLoader 1 must wait for the main process to

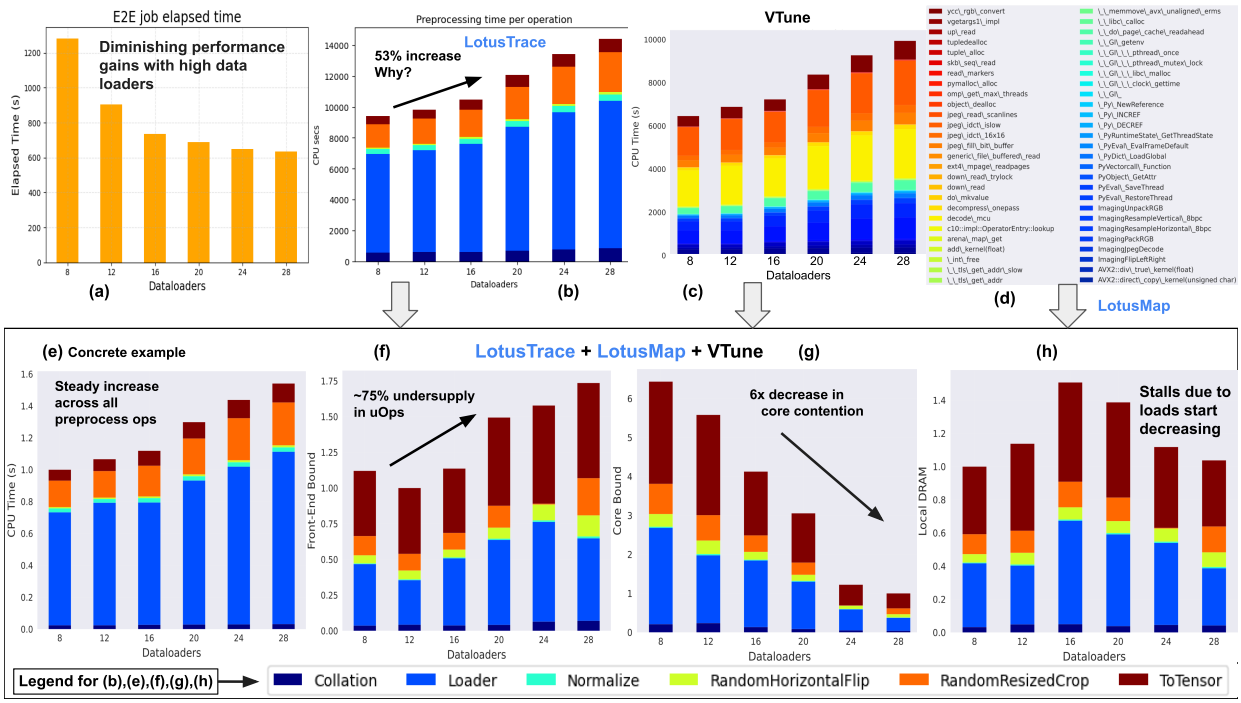

Fig. 6. Combining LotusTrace and LotusMap enables analysis of performance of preprocessing operations on hardware.

become available. LotusTrace's ability to track each batch's ID through the preprocessing phase allows us to identify such out-of-order events.

These out-of-order events can result in prolonged GPU idle periods. Future work could leverage the information provided by LotusTrace for better DataLoader scheduling or GPU multiplexing techniques.

Takeaway 4: *Out-of-order batch arrivals due to the shared data queue among multiple data loaders can lead to significant wait times for the main process and preprocessed batches, resulting in GPU stalls. LotusTrace's tracing capabilities enable identifying and analyzing such out-of-order events.*

D. Observations from Hardware Performance

We present a case study to demonstrate the Lotus's capability to link high-level Python functions with low-level hardware performance counters by combining information collected via LotusTrace and LotusMap. This case study investigates the impact of the number of data loader workers on the performance of the image classification pipeline.

To conduct this study, we use a fixed batch size of 1024 and 4 GPUs and vary the number of data loader workers from 8 to 28 in increments of 4. Exceeding 28 workers leads to OOM issues on our 32-core machine. The experiments run for 1 epoch, processing the same amount of training data across all configurations. As a result, the variability in preprocessing time is attributed to the number of dataloader workers. The data collection involves using LotusTrace for preprocessing operation information and Intel VTune for hardware performance counter data.

In Figure 6(a), we observe a ~50% drop in E2E job elapsed time as the number of dataloaders increase from 8 to 28. Beyond 20 dataloaders, there is a diminishing return in performance gain. LotusTrace reveals that total CPU seconds increased from 9402.62 to 14423.64 seconds (53% increase) from 8 to 28 data loaders, with a steady rise in each preprocessing operation's CPU time (Figure 6(b)).

On the other hand, VTune's profile collects hardware performance counters for 300+ C/C++ functions called during the run, which can not be directly used to explain the rise of CPU time for each preprocessing operation on the hardware level. We use LotusMap to obtain a mapping (Table I) of C/C++ functions to Python preprocessing operations. The mapping allows us to filter out C/C++ functions irrelevant to preprocessing from the 300+ candidates (Figure 6(c,d)). By combining the mapping and the elapsed time measured by LotusTrace, we can attribute hardware performance counters from C/C++ functions to the corresponding Python preprocessing operations, enabling reporting of hardware metrics per preprocessing operation (Figure 6(e - h)), a capability not previously available.

Figure 6(e) shows that CPU time increases steadily for all preprocessing operations, in line with our observation from LotusTrace. Figure 6(f) and Figure 6(g) further explain this increase by revealing a steep undersupply of uOperations to the backend as data loaders increase, causing low contention for cores in the backend of the microarchitecture. With the workload being front-end bound, the pressure on stalls caused by loads serviced by Local DRAM decreases (Figure 6(h)).

TABLE III
COMPARISON OF PROFILER OVERHEADS. TIME OVERHEADS ARE COMPARED WITH THE BASELINE WHICH RUNS THE SAME EXPERIMENT WITH NO PROFILER.

Profiler	Dataset	Wall time	Log storage
LOTUS	ImageNet	~0%	299.2MB
Scalene	ImageNet	96.1%	2.5 MB
py-spy	ImageNet	8%	97.8 MB
LOTUS	ImageNet-small	~2%	6.1 MB
austin	ImageNet-small	3.2%	6.8 GB
PyTorch Profiler	ImageNet-small	86.4%	30.3 MB

TABLE IV
COMPARISON OF PROFILER FUNCTIONALITIES.

Profiler	Epoch	Batch	Async	Wait	Delay
LOTUS	✔	✔	✔	✔	✔
Scalene	✖	✖	✖	✖	✖
py-spy	✔	✖	✖	✖	✖
austin	✔	✖	✖	✖	✖
PyTorch Profiler	✖	✖	✖	✔	✖

Additionally, this example underscores the importance of LOTUSMAP's mapping quality. For instance, even though `ToTensor` is a short-lived function, it occurs frequently. Without capturing its mappings using techniques described in § IV-B, we wouldn't be able to account for its significant contribution to the trends observed in Figure 6(f,g,h). Bucketing and handling of inconsistent functions are also important to ensure that hardware performance counters are attributed correctly. For example, if `decode_mcu`, the most CPU time-consuming function, is incorrectly bucketed with `RandomResizedCrop`, we would observe a 30.21% increase in the CPU time of `RandomResizedCrop`. For brevity, we do not include analysis on AMD (see our repository for details).

Takeaway 5: *Selecting the number of data loader workers is non-trivial, as increasing their number could have diminishing returns in reducing end-to-end job elapsed time while leading to an increase in CPU time. LOTUS helps reveal contentions in hardware resources under different configurations.*

VI. COMPARISON OF PROFILERS

We compare LOTUSTRACE with several other profiling tools. Our experiments show that LOTUSTRACE (1) incurs smaller time and storage overheads, while providing more information compared to alternatives (§ VI-B); and that (2) it is easy to use and requires minimal code changes for instrumenting new machine learning pipelines (§ VI-C).

A. Experiment Setup

We compare LOTUSTRACE with four representative Python profilers.

- Scalene [23]: a state-of-the-art sampling-based Python profiler for CPU and GPU usage with respect to time and memory consumed by each line of Python code.
- py-spy [24]: a sampling-based Python profiler that captures CPU time per function.
- austin [25]: a sampling-based Python profiler for CPU and memory consumed per function.
- PyTorch profiler [31]: PyTorch's built-in tracing-based profiler (`torch.profiler`).

For performance, we compare the wall time overhead throughout the program's lifetime relative to a baseline run without profiling, as well as the log storage overhead. For functionality, we assess whether each profiler captures key preprocessing metrics: the overall and per-operation elapsed times in an epoch (*Epoch*), the per batch elapsed time (*Batch*), the asynchronous interaction between the main process and the dataloaders that enables data flow visualization (*Async*), the main process batch wait time (*Wait*), and the batch consumption delay time (*Delay*).

B. Overhead and Functionality

We evaluate the profilers on the IC pipeline with the ImageNet dataset described in Section V-A, using a batch size of 512, 1 GPU, and 1 data loader, with sampling randomness disabled for consistency. Since some profilers face challenges with storage overhead or out-of-memory (OOM) errors with the full ImageNet, we also include a subset of ImageNet consisting of 26,061 images (ImageNet-small), to facilitate comparison in these cases. Table III and Table IV summarize the profiling overhead and functionality of each tool. Overall, LOTUSTRACE provides the most detailed preprocessing insights with the least overhead.

Scalene, py-spy, and austin use sampling to capture profiling information. Scalene has a high wall time overhead of 96%, interfering with program completion time. Its default sampling rate of 10 *ms* is too coarse to measure many preprocessing operations that take <10*ms* per image (Table II). Increasing the sampling rate puts the profiler on the critical path, distorting results [23]. py-spy has a lower wall time overhead of 8% but still suffers from the coarse 10 *ms* default sampling rate. It can report per-epoch preprocessing times within 1% of LOTUSTRACE, but lacks markers for batch boundaries to report per batch time. Austin supports a finer 100μs sampling rate, enabling more accurate capture of short operations. However, the finer sampling leads to 1000× higher storage overhead than LOTUSTRACE (6.8GB vs 6.1MB on ImageNet-small). Additionally, its default sampling rate of 100μs is too coarse to measure many preprocessing operations that take <100μs per image (Table II). Austin's per-epoch preprocessing and operation times are within 0% and 15% of LOTUSTRACE, respectively. Like py-spy, it lacks batch markers.

PyTorch's tracing-based profiler effectively captures the main process's wait time for a batch but provides no visibility into preprocessing worker execution. It has a high overhead, with 86% wall time and 5× storage compared to LOTUSTRACE on ImageNet-small. The profiler buffers profiling data in memory until program completion, causing OOM errors on the full ImageNet dataset.

In contrast, LOTUSTRACE uses instrumented tracing to capture fine-grained timings of the entire preprocessing pipeline with a low wall time overhead of <2%. LOTUSTRACE

is the only profiler that can capture the asynchronous flow of data between the main process and workers, enabling unique metrics like per-batch timings, wait times, and batch delays not captured by other tools.

C. Ease of use

We discuss the generalizability and the ease of use of LotusTrace by comparing the instrumentation efforts needed to profile the three ML pipelines described in § V-A.

Despite the difference in task, model, dataset, and preprocessing operations, all pipelines require less than 25 lines of code changes for instrumentation. The changes mainly involve passing log file paths and modifying preprocessing operations, with the application logic and program flow remaining unchanged with the instrumentation in all cases. The IS pipeline needs 17 lines of changes, of which 7 lines are for passing the log file path and 10 lines are for consolidating preprocessing operations within a `torchvision.Compose` call. The OD pipeline requires 23 lines of changes, with 8 lines for passing file paths and 15 lines for preprocessing operations. The IC pipeline needs 10 lines of changes, consisting of 5 lines for passing file paths and 5 lines for preprocessing operations/dataset class.

Although LotusTrace requires code changes, the effort is small and can be justified given the additional insights into the preprocessing pipeline execution. In comparison, purely sampling-based profilers and PyTorch profiler do not require any code changes but offer limited information.

VII. Related Work

Preprocessing optimization. Recent studies have explored various CPU-based and accelerator-based optimizations for preprocessing pipelines [1], [2], [7]–[9], [12]–[15], [17]–[22], [50], [51]. For CPU-based optimizations, tf.data [8] simplifies composing preprocessing steps in Tensorflow by providing a declarative API and automatic tuning performance knobs such as prefetching, parallel computing, and IO. Plumber [9] collects aggregate statistics about CPU cycles, I/O, and materialization cost to analytically bound and configure a parallelism strategy for the preprocessing pipeline. While these prior works have focused on identifying bottlenecks and optimizations, they do not provide a tool to characterize the performance of the preprocessing pipelines under different configurations. Lotus is unique in its ability to identify bottlenecks because it enables characterization both at the level of the ML framework as well as the CPU processor. This aids in the identification of how the bottleneck can shift from the framework level, due to misconfiguration, to the CPU level, due to specific architecture component contention, guiding future optimization. Orthogonal to our focus are GPU-based preprocessing libraries such as DALI [11], which offloads the bottleneck to (expensive) GPUs. The choice between CPU and GPU-based preprocessing depends on the specific workload and system configurations. Moreover, GPU-based preprocessing libraries often require CUDA expertise to write performant custom operations.

Profiling Machine Learning Pipelines. Framework-based profilers, such as the PyTorch profiler, were designed to aid in ML training, but they have limitations. The PyTorch profiler, for instance, focuses on capturing asynchronous interactions between CPU and GPU operations rather than the data flow between the main process and DataLoader workers. General-purpose Python profilers such as cProfile [52], Profile [53], pprofile [54], line_profiler [55], and pyinstrument [56] do not support multi-processing, and are also not able to capture the asynchronous data flow. The recent focus on improving preprocessing efficiency has motivated solutions aimed at understanding the preprocessing pipeline. For instance, Plumber [9] collects aggregate statistics to capture per-operation throughput and CPU time, but lacks support for identifying hardware bottlenecks or stalls in the asynchronous data flow. Another work focuses on profiling and understanding tradeoffs between caching intermediate results to storage and recomputation [16]. Lotus provides complementary insights into the execution trends in preprocessing steps. While existing tools for profiling Python applications and accessing low-level hardware data offers some of this information, we outlined their limitations in § VI and § IV, and demonstrated that Lotus addresses the profiling goals.

VIII. Conclusion

ML data preprocessing has emerged as an important performance bottleneck in ML training pipelines. To facilitate current and future work on optimizing data preprocessing, there is a growing need for better tools that provide fine-grained insights into the execution of preprocessing operations. In this work, we present Lotus, a new profiling tool for PyTorch preprocessing pipelines. Lotus combines a new instrumentation methodology to capture fine-grained timing information about individual preprocessing steps, with a new mapping technique that allows it to link hardware-level events with distinct Python operations. Using several preprocessing pipelines from the MLPerf benchmark, we demonstrate that Lotus provides insights into the pipeline execution which is not otherwise available with existing state-of-the-art profilers, while requiring minimal instrumentation effort. Lotus is open sourced, and we welcome contributions from the community as we enhance it with additional features, such as automated log analysis, and evaluate it with other use cases.

References

[1] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in DNN training," *Proceedings VLDB Endowment*, vol. 14, no. 5, pp. 771–784, Jan. 2021.

[2] M. Zhao, E. Adamiak, and C. Kozyrakis, "cedar: Composable and optimized machine learning input data pipelines," Jan. 2024.

[3] NVIDIA, "NVIDIA DGX-1 THE ESSENTIAL INSTRUMENT FOR AI RESEARCH," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-datasheet-nvidia-us-808336-r3-web.pdf, Jul. 2019, accessed: 2024-5-14.

[4] NVIDIA, "NVIDIA DGX-2 THE WORLD'S MOST POWERFUL DEEP LEARNING SYSTEM FOR THE MOST COMPLEX AI CHALLENGES," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-2/dgx-2-print-datasheet-738070-nvidia-a4-web-uk.pdf, Oct. 2018, accessed: 2024-5-15.

[5] P. Tredak and S. Layton, "S8906: Fast data pipelines for deep learning training, 2018."

[6] J. Lisiecki and M. Zientkiewicz, "S9925: FAST AI DATA PREPROCESSING WITH NVIDIA DALI," https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9925-fast-ai-data-pre-processing-with-nvidia-dali.pdf, Mar. 2019, accessed: 2024-5-15.

[7] A. Audibert, Y. Chen, D. Graur, A. Klimovic, J. Šimša, and C. A. Thekkath, "tf.data service: A case for disaggregating ML input data processing," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*. ACM, pp. 358–375.

[8] D. G. Murray, J. Šimša, A. Klimovic, and I. Indyk, "tf.data: A machine learning data processing framework," *Proceedings VLDB Endowment*, vol. 14, no. 12, pp. 2945–2958, Jul. 2021.

[9] M. Kuchnik, A. Klimovic, J. Šimša, V. Smith, and G. Amvrosiadis, "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4. Indio, CA: Systems and Machine Learning Foundation, 2022, pp. 33–51.

[10] I. Svogor, C. Eichenberger, M. Spanring, M. Neun, and M. Kopp, "Profiling and improving the PyTorch dataloader for high-latency storage: A technical report," *arXiv [cs.LG]*, Nov. 2022.

[11] "NVIDIA developer data loading library (DALI)," https://developer.nvidia.com/dali, accessed: 2024-5-18.

[12] P. Park, H. Jeong, and J. 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)*. IEEE, pp. 825–838.

[13] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl, "Faster neural network training with data echoing," Jul. 2019.

[14] G. Leclerc, A. Ilyas, L. Engstrom, S. Park, H. Salman, and A. Madry, "FFCV: Accelerating training by removing data bottlenecks," in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2023, pp. 12 011–12 020.

[15] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, "Cachew: Machine learning input data processing as a service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 689–706.

[16] A. Isenko, R. Mayer, J. Jedele, and H.-A. Jacobsen, "Where is my training bottleneck? hidden Trade-Offs in deep learning preprocessing pipelines," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 1825–1839.

[17] "TFRecord and tf.train.example," https://www.tensorflow.org/tutorials/load_data/tfrecord, accessed: 2024-5-18.

[18] H. Zhao, Z. Yang, Y. Cheng, C. Tian, S. Ren, W. Xiao, M. Yuan, L. Chen, K. Liu, Y. Zhang, Y. Li, and W. Lin, "GoldMiner: Elastic scaling of training data Pre-Processing pipelines for deep learning," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 1, no. 2, pp. 1–25, Jun. 2023.

[19] T. Um, B. Oh, B. Seo, M. Kweun, G. Kim, and W.-Y. Lee, "FastFlow: Accelerating deep learning model training with smart offloading of input data pipeline," vol. 16, pp. 1086–1099.

[20] D. Graur, O. Mraz, M. Li, S. Pourghannad, C. A. Thekkath, and A. Klimovic, "Pecan: Cost-efficient ML data preprocessing with automatic transformation ordering and hybrid placement," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, pp. 649–665.

[21] H. Zhao, Z. Han, Z. Yang, Q. Zhang, M. Li, F. Yang, Q. Zhang, B. Li, Y. Yang, L. Qiu, L. Zhang, and L. Zhou, "SiloD: A co-design of caching and scheduling for deep learning clusters," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, May 2023, pp. 883–898.

[22] G. Lee, I. Lee, H. Ha, K. Lee, H. Hyun, A. Shin, and B.-G. Chun, "Refurbish your training data: Reusing partially augmented samples for faster deep neural network training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 537–550.

[23] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with SCALENE," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 51–64.

[24] B. Frederickson, "py-spy: Sampling profiler for python programs," https://github.com/benfred/py-spy.

[25] G. N. Tornetta, "austin: A frame stack sampler for cpython," https://github.com/P403n1x87/austin.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc.

[27] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, pp. 336–349.

[28] "LOTUS: A profiling tool for ml preprocessing pipelines," https://github.com/rajveerb/lotus/tree/iiswc24ae.

[29] "Intel VTune™ profiler documentation," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-documentation.html, accessed: 2024-5-30.

[30] "AMD μProf," https://www.amd.com/en/developer/uprof.html, accessed: 2024-5-30.

[31] "The pytorch profiler: torch.profiler," https://pytorch.org/docs/stable/profiler.html.

[32] "perf_events tutorial," https://perf.wiki.kernel.org, accessed: 2024-5-30.

[33] "pybind11: Seamless operability between c++11 and python."

[34] "perf-map-agent: A java agent to generate method mappings to use with the linux 'perf' tool," https://github.com/jvm-profiling-tools/perf-map-agent.

[35] "Allow the linux perf profiler to see python calls," https://github.com/python/cpython/issues/96143.

[36] "Allow "precompiled" perf-trampolines to largely mitigate the cost of enabling perf-trampolines," https://github.com/python/cpython/issues/109587.

[37] "itt-python," https://github.com/oleksandr-pavlyk/itt-python.

[38] "Intel perfmon - broadwell formula," https://github.com/intel/perfmon/blob/9bc2f87094fc84cc6bcc87df276807b182ddd327/BDX/metrics/perf/broadwellx_metrics_perf.json.

[39] B. Gregg, "Linux profiling at netflix," https://www.slideshare.net/slideshow/scale2015-linux-perfprofiling/44966387, accessed: 2024-8-7.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," Dec. 2015.

[43] N. Heller, N. Sathianathen, A. Kalapara, E. Walczak, K. Moore, H. Kaluzniak, J. Rosenberg, P. Blake, Z. Rengel, M. Oestreich, J. Dean, M. Tradewell, A. Shah, R. Tejpaul, Z. Edgerton, M. Peterson, S. Raza, S. Regmi, N. Papanikolopoulos, and C. Weight, "The KiTS19 challenge data: 300 kidney tumor cases with clinical context, CT semantic segmentations, and surgical outcomes," *ArXiv*, vol. abs/1904.00445, Mar. 2019.

[44] F. Isensee, P. Kickingereder, W. Wick, M. Bendszus, and K. H. Maier-Hein, "No New-Net," in *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*. Springer International Publishing, 2019, pp. 234–244.

[45] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.

[46] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," Mar. 2017.

[47] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart,

L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14.
[48] CloudLab, "CloudLab: 10 storage mechanisms."
[49] "Adding zvols," https://www.truenas.com/docs/core/coretutorials/storage/pools/zvols/.
[50] R. Nouaji, S. Bitchebe, and O. Balmau, "SpeedyLoader: Efficient pipelining of data preprocessing and machine learning training," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, ser. EuroMLSys '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 65–72.
[51] D. Kang, A. Mathur, T. Veeramacheneni, P. Bailis, and M. Zaharia, "Jointly optimizing preprocessing and inference for DNN-based visual analytics," *Proceedings VLDB Endowment*, vol. 14, no. 2, pp. 87–100, Oct. 2020.
[52] B. Rosen and T. Czotter, "The Python Profilers (cProfile)."
[53] J. Roskind, "The Python Profilers (profile)."
[54] V. Pelletier, "pprofile: Line-granularity, thread-aware deterministic and statistic pure Python profiler."
[55] "line_profiler: Line-by-line profiling for python."
[56] J. Rickerby, "pyinstrument: Call stack profiler for Python."

Appendix

A. Abstract

The artifact contains the source code and documentation for Lotus, encompassing both LotusTrace and LotusMap components. We detail the installation procedure and experimental workflows necessary to partially reproduce the results presented in Figure 4, Figure 5 and Figure 6. In addition, we provide instructions for generating tracing visualizations, as shown in Figure 2, and for mapping Python functions to their C++ counterparts for Intel chips, as shown in Table I.

B. Artifact check-list (meta-information)

- **Program:** Image classification, Bash, PyTorch, Python, Google Chrome
- **Compilation:** CUDA, gcc/g++, CMake
- **Binary:** Intel VTune
- **Model:** ResNet18
- **Data set:** ImageNet 2012
- **Run-time environment:** Intel processor, Anaconda, Ubuntu 20.04 (kernel version 5.4.0-139-generic)
- **Hardware:** c4130 node in Cloudlab.
- **Metrics:** Elapsed time, CPU time, Hardware PMU stats
- **Output:** Trace, JSON, CSV, and PNG files (plots)
- **Experiments:** Python scripts, Bash scripts, and Intel VTune
- **How much disk space required (approximately)?:** 500 GB
- **How much time is needed to prepare workflow (approximately)?:** 5 hours
- **How much time is needed to complete experiments (approximately)?:** 5 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT license (with some code under BSD-3 License)
- **Archived (provide DOI)?:** Zenodo: https://zenodo.org/doi/10.5281/zenodo.13245169

C. Description

1) How to access: All our code is available in the following GitHub repository: https://github.com/rajveerb/lotus/tree/iiswc24ae. All the scripts, code, submodules and instructions can be found in the repository.

The up-to-date instructions for setting up the environment are available in the repository (under `SETUP.md`). The up-to-date instructions for installations (Section D) and experiment workflow (Section E) are also available in the repository (under `REPLICATE.md`)

2) Hardware dependencies: There are no specific hardware dependencies for this project. The code has been tested on Intel and AMD processors, NVIDIA A40/V100 GPUs. For replication, we recommend the c4130 node on Cloudlab, which has an Intel Processor chip supported by Intel VTune and 4 NVIDIA V100 GPUs.

3) Software dependencies: The artifact requires: Anaconda, Intel VTune, CUDA, CuDNN, Python (3.10), PyTorch (2.0), Google Chrome, and Ubuntu for replication purpose. We provide scripts and detailed instructions for installing the dependencies.

4) Data sets: ImageNet 2012 dataset (~140GB)

5) Models: ResNet18 model

D. Installation

1) Clone the Lotus repository and get submodules:
```
git clone --depth 1 --recurse-submodules\
 git@github.com:rajveerb/lotus.git\
 -b iiswc24ae
cd lotus
```
2) Create a conda environment:
```
conda create -n lotus python=3.10 -y
conda activate lotus
```
3) Install `itt-python` using build instructions below:
```
pushd code/itt-python
export ITT_LIBRARY_DIR=/opt/intel/oneapi/vtune\
  /latest/lib64
export ITT_INCLUDE_DIR=/opt/intel/oneapi/vtune\
  /latest/include
python setup.py install
# Check if installed
pip list | grep "itt"
popd
```
4) Follow the LotusTrace build instructions (will take a few hours) below:
```
sudo apt install -y g++
bash install_lotustrace.sh
# Sanity check
pip list | grep "torch" | grep "2.0.0a0"
```
5) Follow the `torchvision` build instructions below:
```
bash install_torchvision.sh
# Sanity check
pip list | grep "torchvision" | grep "0.15.1a0"
```
6) Install below packages:
```
conda install ipykernel pandas=2.0.3 -y
pip install matplotlib==3.9.0 natsort==8.4.0\
 seaborn==0.13.2
```

E. Experiment workflow

1) Mapping Results:

1) Get the mapping logs for the preprocessing operations:
```
bash code/image_classification/LotusMap/LotusMap.sh
```
2) Run all cells in the `code/image_classification/LotusMap/Intel/logsToMapping.ipynb` notebook. This generates a JSON file with mapping info `code/image_classification/LotusMap/Intel/mapping_funcs.json`, similar to Table I.

2) Tracing Results:

1) Run the Image Classification pipeline experiment where batch size is 512 and number of GPUs is 4 and LotusTrace is enabled:

```
# Activate VTune, command will fail
# an error if it is already activated
source /opt/intel/oneapi/setvars.sh
# Sanity check
vtune --version
bash scripts/cloudlab/LotusTrace_imagenet.sh
```

2) Run the commands below for observations in 'High variance in preprocessing time' (Figure 4 and the statistics):

```
python code/image_classification/analysis/\
  LotusTrace_imagenet_vary_batch_and_gpu/\
  preprocessing_time_stats.py\
  --remove_outliers\
  --data_dir lotustrace_result/512_gpu4/\
  --output_file lotustrace_result/\
      preprocessing_time_stats.log
python code/image_classification/analysis/\
  LotusTrace_imagenet_vary_batch_and_gpu/\
    box_plot_preprocessing_time.py\
  --remove_outliers\
  --data_dir lotustrace_result/512_gpu4\
  --output_file lotustrace_result/\
    box_plot_preprocessing_time.png
```

3) Run the commands below for observations in 'Significant wait and delay time' (Figure 5 and the statistics):

```
python code/image_classification/analysis/\
  LotusTrace_imagenet_vary_batch_and_gpu/\
  delay_and_wait_time_stats_and_plot.py\
--sort_criteria duration\
--data_dir lotustrace_result/b512_gpu4\
--fig_dir lotustrace_result/figures\
--output_file lotustrace_result/\
  delay_and_wait_time_stats_and_plot.log
```

4) Run the visualization script (Figure 2):

```
python code/visualize_LotusTrace/\
  visualization_augmenter.py\
--coarse\
--lotustrace_trace_dir lotustrace_result/b512_gpu4\
--custom_log_prefix lotustrace_log\
--output_lotustrace_viz_file\
 lotustrace_result/viz_file.lotustrace
```

Open the file in chrome trace viewer for visualization. Navigate to `chrome://tracing` URL in Google Chrome, upload the `viz_file.lotustrace` and visualize the trace.

3) Hardware Performance Results:

1) Run the steps below to generate hardware performance numbers for Image Classification pipeline where batch size is 1024, number of GPUs is 4, and number of dataloaders is 20. LotusTrace and Intel VTune are enabled:

```
source /opt/intel/oneapi/setvars.sh
bash scripts/cloudlab/LotusTrace_imagenet_vtune.sh
```

2) Follow the steps below to get a CSV of hw performance numbers (has to be performed manually):

```
# Below step will provide a link, open a browser window,\
# and login to the VTune GUI (set the password upto you)
vtune-backend --web-port 8080 --data-directory ./vtune\
_mem_access_vary_dataloader/b1024_gpu4_dataloader20
```

- Navigate to `Microarchitecture Exploration` tab
- Perform grouping by `Source Function / Function / Call Stack`
- Select all cells and paste it in a CSV file called `code/image_classification/analysis/combine_lotus/lotustrace_uarch/b1024_gpu4_dataloader20.csv`

3) Plot Figure 6 (a) by running `code/image_classification/analysis/combine_lotus/elapsed_time_plot.ipynb` notebook
4) Plot Figure 6 (b) by running `code/image_classification/analysis/combine_lotus/per_python_func_plot_vary_dataloaders.ipynb` notebook
5) Plot Figure 6 (c) by running below command:

```
python code/image_classification/analysis/combine_lotus/\
hw_event_analyzer.py\
  --mapping_file code/image_classification/LotusMap/\
  Intel/mapping_funcs.json\
  --uarch_dir code/image_classification/analysis/\
  combine_lotus/lotustrace_uarch\
  --combined_hw_events code/image_classification/\
  analysis/combine_lotus/combined_lotustrace_uarch.csv\
  --cpp_hw_events_plot_dir code/image_classification/\
  analysis/combine_lotus/cpp_hw_events_figs
```

Check out the `code/image_classification/analysis/combine_lotus/cpp_hw_events_figs` for the plots.

6) Figure 6 (e)-(h) by running `code/image_classification/analysis/combine_lotus/c_to_python_analyser.ipynb` notebook Check out the plots in the `code/image_classification/analysis/combine_lotus/mapped_python_figs` directory.

F. Evaluation and expected results

Upon successful completion of each section, users should be able to achieve the following results:

1) Section IV-B: Generate the mapping of Python functions to their C++ counterparts for Intel chips, as shown in Table I.
2) Section V-C: Replicate the results shown in Figure 4 and Figure 5 for the configuration with a batch size of 512, 4 GPUs, and 4 dataloaders. Generate tracing visualizations similar to those presented in Figure 2.
3) Section V-D: Replicate the results shown in Figure 6 (a,b,c,e,f,g,h) for the configuration with a batch size of 1024, 4 GPUs, and 20 dataloaders.

We focus on specific configurations due to time constraints, but the same steps can be applied to other configurations to reproduce the complete figures.

G. Notes

The replicated plots for Figure 6 (e,f,g,h) show raw performance numbers and are not normalized with respect to the minimum, since we focus on one configuration.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html