Datacenter-Scale Analysis and Optimization of GPU Machine Learning Workloads

Lukasz Wesolowski, Facebook, Inc.
Bilge Acun, Facebook, Inc.
Valentin Andrei, Facebook, Inc.
Adnan Aziz, Facebook, Inc.
Gisle Dankel, Facebook, Inc.
Christopher Gregg, Stanford University, Facebook, Inc.
Xiaoqiao Meng, Facebook, Inc.

Cyril Meurillon, Facebook, Inc.
Denis Sheahan, Facebook, Inc.
Lei Tian, Facebook, Inc.
Janet Yang, Carnegie Mellon University
Peifeng Yu, University of Michigan
Kim Hazelwood, Facebook, Inc.

***Abstract*—We present a system to collectively optimize efficiency in a very large scale deployment of GPU servers for machine learning workloads at Facebook. Our system (a) measures and stores system-wide efficiency metrics for every executed workflow, (b) aggregates data from across the execution stack to identify optimization opportunities that maximize fleet-wide efficiency improvements, (c) provides periodic and on-demand whole-system profiling for workflows and, (d) automatically analyzes traces for common anti-patterns. We present each component of the stack and show case studies demonstrating the use of the tools to significantly improve performance. To our knowledge, our system is the most complete and effective solution for identifying and addressing efficiency problems in datacenter-scale GPU deployments.**

I. Introduction

Large-scale deployments of GPU servers, once rarely found outside national laboratories, cloud service providers, and academic research centers, have in the past few years proliferated across the technology sector and other industries as GPUs have become a dominant architecture for machine learning (ML) workloads key to the business of many companies. In contrast to research-focused GPU supercomputers that must support many programming languages, runtime systems and parallel libraries [1], large-scale industry GPU deployments tend to feature a much higher homogeneity of application domains and libraries, with most jobs executing ML workloads using a common ML library, and a higher cohesion of the user base, typically comprising employees of the company owning the resource. This presents an opportunity for performance measurement and optimization that provide both global and workflow-level information and identify the highest yield targets for improving efficiency. Similar opportunities may exist in consolidated resource-sharing environments. Nonetheless, there is a lack of existing systems that satisfy the key requirements for tooling in this space.

In this paper we demonstrate a methodology and tooling for improving the efficiency of a large GPU deployment at Facebook. Our system comprises the following components, which together allow a relatively small number of GPU performance experts to support a much larger number of ML specialists:

- A telemetry infrastructure capable of collecting detailed performance metrics for all applications and execution stack levels via an in-process library (Section IV)
- Data aggregation and visualization tools that automatically analyze fleet-wide metrics and surface global performance issues (Section V)
- A profiling service integrated with the existing performance analysis portal and workflow management tools to enable the collection and display of detailed performance profiles with the click of a button for any running workflow in the fleet (Section VI)
- A timeline trace analysis tool that identifies common performance issues and provides actionable recommendations (Section VI-C)

To the best of our knowledge, our system is the most comprehensive solution for identifying and fixing global performance issues in a datacenter-scale GPU deployment as well as for individual workflows.

The rest of the paper is organized as follows. We present a survey of related work and how it differs from our efforts in Section II. An overview of our hardware and infrastructure follows in Section III. We then describe infrastructures built for telemetry, data aggregation, performance profiling and optimization in Sections IV, V, VI respectively. We demonstrate the capabilities of our system with case studies based on real applications in Section VII and conclude in Section VIII.

II. Related Work

Utilizing GPUs efficiently in a shared large-scale environment brings a set of unique challenges. Developers of deep learning frameworks like TensorFlow, PyTorch, MXNet, CNTK, etc. are spending considerable effort on balancing the work between the host CPU and the GPU, efficient graph compiling through fused operators or smart tensor caching, reduced precision training, etc., in order to minimize communication with the host processor and reduce idle times on the GPU.

When using multiple GPUs for solving tasks like deep-learning (DL) distributed training, achieving high utilization efficiency becomes even more difficult. Scaling efficiency is a difficult problem to solve in distributed training and researchers have developed solutions like gradient compression or efficient communication patterns. Finally, at fleet-wide level, all the efficiency limiting effects observed at single-GPU and multi-GPU level get amplified proportionally with the size of the infrastructure. In [2], the average observed GPU utilization in a multi-tenant environment running deep learning training was 52%, which demonstrates the magnitude of some of the challenges we mentioned. Also, as the authors mention, this number is an optimistic upper bound because it does not include to what extent each GPU is being used, only that it is active.

To achieve efficient GPU usage in a large-scale setup, monitoring and introspection tools that are capable of showing both detailed utilization metrics and system-wide bottlenecks are desirable.

NVIDIA partly addresses these needs with the combination of the Datacenter GPU Manager (DCGM) for fleet-wide monitoring and NVIDIA Nsight Systems for system-wide profiling. While DCGM provides a big improvement over previous tools such as NVIDIA System Management Interface (nvidia-smi), it is the CUDA Profiling Tools Interface that provides the most comprehensive and detailed set of metrics.

When designing a fleet-wide performance introspection system, there is a trade-off between the quantity of gathered information and the overhead of the data collection. [3] describes how Cloud TPU Tools and TensorBoard can be used to analyze the performance of workloads running on Google's TPU [4]. The tools provide two analysis modes: profiling and monitoring. While the profiling mode provides a rich set of metrics, traces and more, the monitoring mode tracks only device idle time, TPU matrix utilization and step time. The profiling information can only be collected for small time windows as opposed to the entire workload's duration. Another open-source Microsoft solution for monitoring resource utilization in GPU clusters is described in [5]. The monitor collects statistics like GPU utilization, memory utilization and thermal data, obtained from NVIDIA NVML. This is the same API used by nvidia-smi and does not provide the detailed GPU utilization metrics available in DCGM or CUPTI. In [6] and [7] solutions for monitoring GPU utilization on Amazon Web Services (AWS) are presented and they also rely on data provided by NVIDIA NVML, being limited to basic device utilization and temperature readings.

The workflow performance optimization system in this paper has parallels to efforts in the HPC space. There exist several open tracing and telemetry infrastructures actively used by the community such as TAU, Projections, and HPCToolkit. Score-P is a tool suite for profiling, event tracing, and online analysis of HPC applications. It supports a range of analysis tools such as Vampir and Scalasca.

The solution presented in this paper has distinguishing attributes critical to systematically improving efficiency of large GPU fleets used for ML workloads:

- continuous metric collection of all executing ML workflows at low overhead ($<1\%$) and transparent to users, by means of an in-process library leveraging NVIDIA CUPTI [8]
- on-demand and periodic trace collection transparent to users
- metric and trace aggregation and visualization tools that identify commonly observed issues across the fleet

III. Hardware and Software Infrastructure

A. Hardware Platforms

The supercomputer-scale GPU datacenter described in this paper is composed of Big Basin GPU servers, with design specifications released publicly as part of the Open Compute Project [9]. A single Big Basin server has two Intel CPUs (various generations) and eight NVIDIA GPUs. Tesla P100 or V100 GPU accelerators are connected by NVIDIA NVLink [10] to form an eight-GPU hybrid cube mesh. Each GPU has either 16GB or 32GB HBM2 memory. Servers have 256 GB RAM and are connected via 100 Gbps ethernet.

B. Machine Learning Models and Use Cases

Our GPU servers are primarily used for training various models for production and experimental purposes. Most workflows use PyTorch [11], due to ease of experimentation with Python, imperative style and simplicity, and FBLearner Flow, Facebook's ML training platform that provides workflow pipeline management, integration with systems for data reading and scheduling, and user interface for experimentation management.

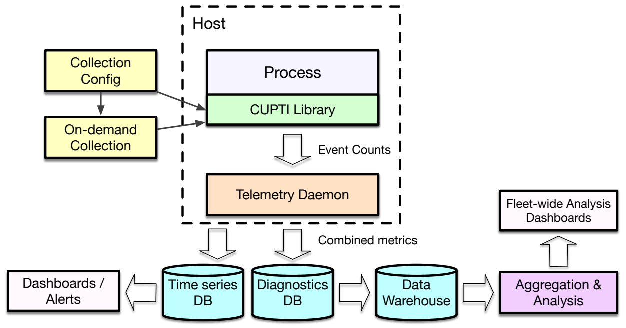
Machine learning algorithms used in the workflows include state of the art Deep Neural Networks (DNN), i.e. Multi-Layer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN/LSTM), as well as other techniques such as Logistic Regression (LR), Support Vector Machines (SVM), Gradient Boosted Decision Trees (GBDT).

IV. Telemetry Infrastructure

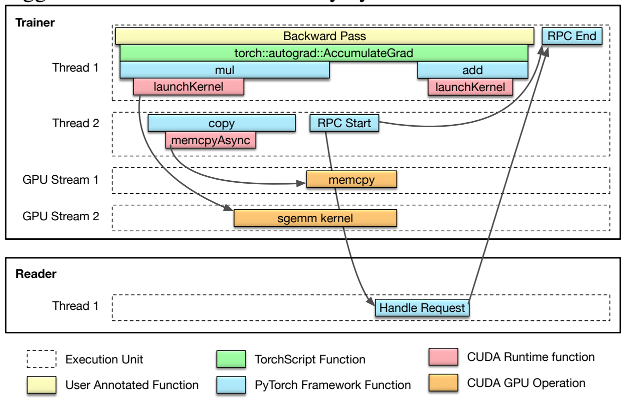
The first step to improve performance for a large-scale deployment of GPU servers is enabling detailed visibility into resource utilization across the key resources such as CPUs, GPUs, memory, network and I/O.

Our solution is to deploy a custom CUPTI-based performance monitoring- and profiling-library, Kineto [12] (Figure 1a). Kineto is an open source project, available at GitHub and integrated with the PyTorch Profiler. Compared to NVML-based telemetry such as nvidia-smi, and even DCGM, the *Kineto* profiling library significantly improves visibility into ML training workloads by providing a means of collecting both detailed GPU hardware performance counters and timeline traces. This approach enables a large degree of customization to and integration with our environment.

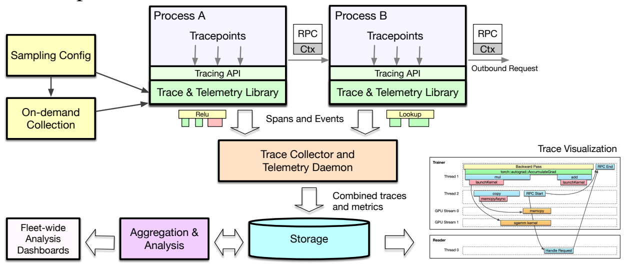
A core set of event counters are collected continuously for nearly every workload running in the fleet. These include elapsed cycles, active cycles, active warps and instructions, and are collected for each streaming multiprocessor (SM) of every GPU. With these we can report metrics such as *SM efficiency, achieved occupancy, instructions per cycle* and others discussed in section V.



(a) Our CUPTI-based profiling library collects performance events for nearly all workloads running in the fleet. Event counts are sent to a system telemetry daemon present on every host where they are combined with other system metrics, including device-level GPU metrics collected via NVML or DCGM, and logged to three different telemetry systems.



(b) Anatomy of a timeline trace. Events are collected from several sources: Workload user instrumentation, TorchScript and PyTorch Profiler frameworks, NVIDIA API instrumentation, GPU operations from CUPTI, and RPC calls.



(c) Trace Collection Infrastructure: The telemetry infrastructure described in Figure 1a is extended to collect traces on-demand. Section VII contains several examples of actual trace snippets.

Fig. 1: Telemetry and Tracing Infrastructure

Counters from each SM are aggregated into a few buckets (p5, p25, p50,...), allowing us to detect SM imbalances, including unused SMs. The counters are then sent from the workload to our system *Telemetry Daemon*, which runs on every host in our fleet. Here they are combined with a collection of other system metrics and sent to the general telemetry infrastructure (Figure 1a). Counter values are logged every 10 seconds by default. This frequency allows us to observe phase changes in a typical training job, particularly training epochs.

Metrics are collected continuously for all workloads. Avoiding sampling increases accuracy, simplifies collection and downstream processing, and enables analysis of resource utilization such as *Dr. Sankey*, described in section V. We rarely see more than 1% overhead from continuous counter collection, which we consider an acceptable cost for the benefit of having complete coverage.

NVIDIA's counters are 32 bits and can overflow in under a second. We therefore read them at least twice per second.

On the other hand, if we issue too many reads across multiple processes and GPUs, we experience read latencies exceeding one second, causing counters to overflow. As a mitigation, we limit collection to a single process per GPU. Since we generally don't stack workloads onto a single GPU, this is not a critical limitation.

The collected metrics are sent to three different telemetry systems:

- A time series database serving real-time system-level metric dashboards and alerts.
- A metric store for near-realtime ($\sim$1 minute delay) drill-down and diagnostics, used for debugging and individual workload analysis, including ad-hoc metric collection.
- A data warehouse with longer retention supporting arbitrary complex queries and processing, used for fleet-wide analysis. Tables here are typically delayed by a few hours and have daily scheduled analysis pipelines.

V. Data Aggregation Tools

A. Metric Terminology

GPUs are massively parallel devices consisting of tens to hundreds of *streaming multiprocessors* (SMs). Work for CUDA GPUs is expressed using *kernels*, routines executed on *grids* of threads. A thread grid is logically partitioned into *blocks*, which define the granularity at which work is assigned to SMs, and further into *warps*, groups of 32 threads which share scheduling logic. We leverage the following metrics to analyze the achieved parallelism at various levels of the architecture:

- *GPU Utilization* measures the fraction of time the GPU is busy. GPU Utilization does not capture parallelism within the GPU. For example, on a V100 GPU, executing a GPU kernel with just one active thread yields 100% GPU Utilization while using less than 0.01% of the compute resources.
- *SM Efficiency* measures SM activity. It is calculated as the percentage of cycles when an SM is active (i.e. has at least one warp in flight). To reach 100% aggregate SM Efficiency for the GPU, at least one warp must be active on every SM for every elapsed cycle.
- *SM Occupancy* measures warp-level parallelism per SM. It is calculated as the number of warps active per SM, averaged over time. It can be measured over either SM active cycles (as is done for NVIDIA's achieved occupancy metric), indicating warp-level parallelism averaged over duration of all kernels, or over elapsed cycles (i.e. counting cycles when the SM has no active warps as 0 occupancy). *SM occupancy over elapsed cycles* is the product of SM Efficiency and SM occupancy over active cycles. As a measure of warp-level parallelism over entire program duration, it is our preferred top-line metric for reporting workflow efficiency. The other metrics are still helpful in determining whether low SM activity or low active warp count is the dominant problem.

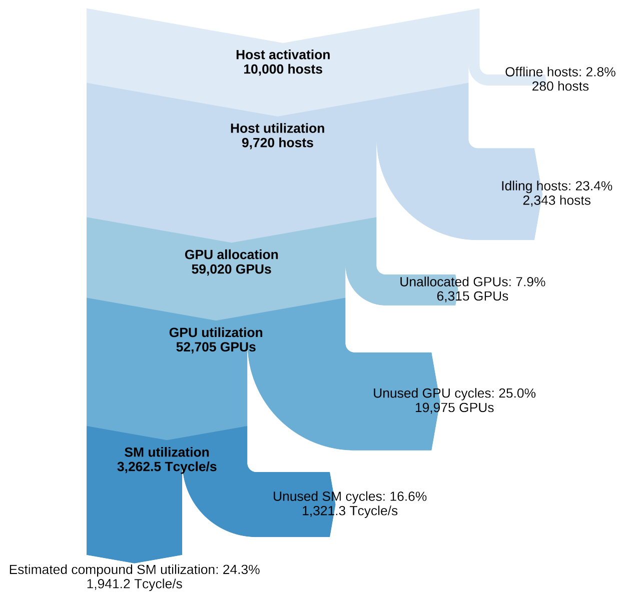
B. Dr. Sankey

We developed a new model for fleet-wide efficiency analysis, which we nicknamed *Dr. Sankey*. This allows us to measure and consolidate inefficiencies across the operating stack in a server pool. This compact representation helps identify at a glance utilization bottlenecks and prioritize optimization efforts. The model relies on a simple drill-down process that estimates the utilization of resources, from coarse to fine.

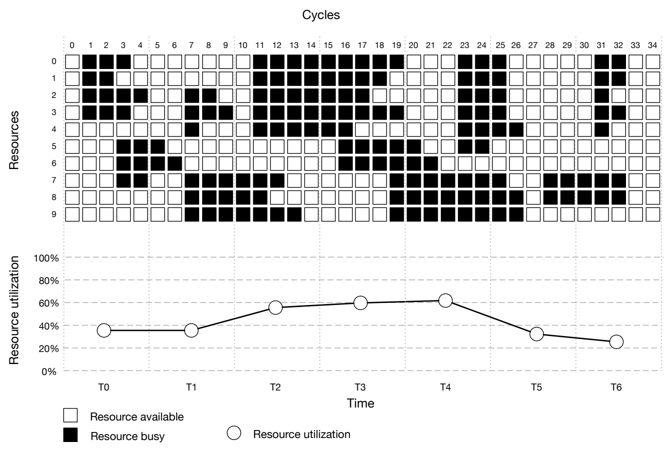
Let's illustrate the process using the fleet described in Section III. A central job scheduler is responsible for managing the fleet resources and optimize the placement of jobs on hosts based on resource requirements and availability.

1) First we consider the availability of hosts in the fleet. At any given point in time, a number of hosts are in repair or temporarily offline for maintenance and therefore not capable of running jobs. We measure the operational efficiency of the fleet with the *Host Activation* rate, defined as the ratio of hosts capable of running jobs (schedulable) and hosts deployed in the fleet.
2) Next, zooming in on the schedulable hosts, we examine how much of the fleet is actually put to work. This is measured by the *Host Utilization* rate, defined as the ratio of busy hosts and schedulable hosts. Insufficient job demand is a common cause of low host utilization.
3) Next, we estimate how many GPU devices are left stranded (unallocated) in busy hosts. The *GPU Allocation* rate is defined as the ratio of GPUs allocated by jobs and total GPUs provisioned on busy hosts. A low allocation rate indicates high fragmentation of GPU resource. This suggests inefficiencies in the scheduler placement algorithm and/or wasteful job requirements - e.g. a job requests all of a host memory but only 1 GPU, leaving 7 GPUs stranded.
4) Next, we measure the utilization of allocated GPUs. The *GPU Utilization* rate is the ratio of cycles a GPU was busy executing instructions and total elapsed cycles, summed over all allocated GPUs. A low GPU utilization means jobs are not able to funnel enough work to the GPUs, and may be caused by CPU or I/O bottlenecks, such as fetching training data.
5) Finally, we measure how efficiently the GPU execution cores are used with the *SM Utilization* rate. This is defined as the fraction of SMs that are utilized (running a warp), averaged over all cycles the GPU is busy. SM Utilization is very similar to SM Efficiency, except that it is only calculated over GPU active cycles instead of elapsed cycles, i.e. SM Efficiency combines GPU Utilization and SM Utilization into a single metric. A low SM utilization rate suggests insufficient SM-level parallelism.

The utilization metrics defined above form a hierarchy of nested metrics, in a Russian doll fashion. The metrics can be conveniently visualized using a Sankey diagram, with each layer representing a level of inefficiency. Figure 2a represents the efficiency of a fictitious GPU fleet of $10,000$ hosts. The inefficiencies at each layer are expressed relatively to the deployed fleet (top layer), so that the percentages add up to



(a) A Sankey diagram depicting fleet-wide resource efficiency.



(b) Illustration of Resource Utilization (with $T = 5$ and $N = 10$).

Fig. 2: Dr. Sankey

100%. The compound efficiency of the fleet is the ratio of the bottom and top layers in the figure. In this example, the ratio (24.3%) represents the effective utilization of the all SMs deployed in the fleet, averaged over time and space.

This methodology is not specific to GPU hardware architecture, and can be generalized to arbitrary types of fleet. Let's first formalize the notion of resource utilization. The average utilization $\overline{u_r}$ of N homogeneous resources of type r over period $[0, T]$, with the resource utilization function $u_r(n,t)$ defined as 1 if resource n is busy (or allocated, not usable, etc.) at time t, 0 otherwise, can be expressed as:

$$\overline{u_r} = \frac{1}{T \cdot N} \sum_{t<T} \sum_{n<N} u_r(n,t)$$

Figure 2b offers a graphic representation of this definition.

Resources in a fleet can typically be organized in a hierarchy, from coarse to fine. For example, a fleet comprises hosts, which comprise CPU chipsets, which comprise CPU cores, etc. This tree allows to simply identify the parent of a utilization metric. Recursing through the parents up to the root yields a sequence of utilization metrics, nested in a Russian doll fashion. The compound utilization $\overline{u_R^*}$ for resource R represents the fully diluted utilization of resource R over the fleet, i.e. the effective utilization of resources of type R over the entire fleet. It can be calculated with the product of all ancestor utilization metrics along the path to the root:

$$\overline{u_R^*} = \prod_{r \leq R} \overline{u_r}$$

C. GPU Efficiency Dashboard

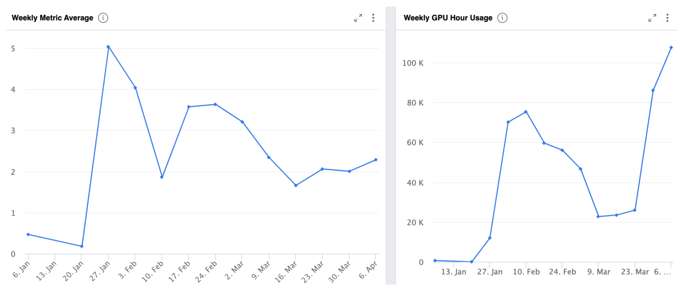
The *Dr. Sankey* chart provides a good overview of inefficiencies across the execution stack. Some of the highest sources of inefficiency are at the workflow level, corresponding to the lowest two levels of Dr. Sankey.

The *GPU Efficiency dashboard* provides a workflow-centric view of GPU efficiency metrics by aggregating performance metrics from *Kineto* with job execution metadata from our scheduler datasets. The dashboard helps performance engineers identify and prioritize optimization opportunities among the large set of running workflows. It tracks SM Efficiency, SM Occupancy over active cycles, and SM Occupancy over elapsed cycles as defined in Section V-A, along with resource use, defined as GPU hours consumed by a run.

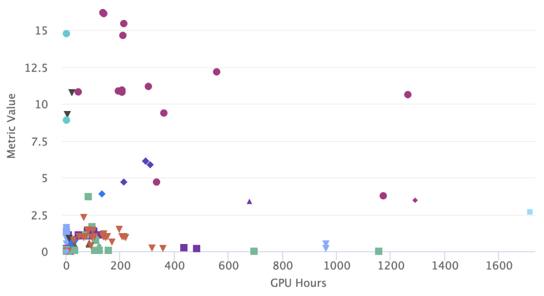
There are hundreds of different workflow types executed across our fleet in any given week, many with a large number of instances. Each workflow executes the distributed training code corresponding to its workflow type, though instances vary with respect to resource requirements and duration. Aggregating data by workflow type allows us to find workflow types with highest aggregate resource use and lowest efficiency. These are the best targets for optimization. The top 10 workflow types by resource use across the fleet account for more than 70% of all resources consumed. By optimizing these workflows we can significantly improve average efficiency, which in practice allows more runs to execute concurrently.

To use the dashboard, one selects a time window and metric to track, and optionally a workflow type and list of users. Information available through the dashboard includes:

- List of workflow types ranked by aggregate resource use and their corresponding average efficiency



(a) Line plots generated from the dashboard showing SM occupancy over elapsed cycles and resource use for an example use case. While resource use is trending up, efficiency appears to be regressing.



(b) An example scatter plot produced using the GPU Efficiency dashboard, displaying resource use and efficiency of all runs of a workflow within a selected period of time, labeled by user (legend is not displayed for privacy).

Fig. 3: GPU Efficiency Dashboard

- Line plots of efficiency and resource use over time (a) in aggregate across all workflows, (b) for a single workflow, or (c) for a list of users of a workflow
- List of all runs matching selected criteria, their resource use, efficiency, and link to their execution logs
- A scatter plot of resource use vs efficiency for all runs with selected criteria, labeled by user

Figure 3a shows an example of efficiency and resource use line plots generated by the dashboard. This view helps to spot negative trends, such as workflows that increase in resource use while degrading in efficiency. Another example from the dashboard is shown in Figure 3b, showing a scatter plot of efficiency and resource use labeled by user for a set of runs of a workflow. This view helps to identify users with heavy resource use or particularly high or low workflow efficiency. Users that have higher efficiency could be applying special optimizations that others could benefit from. Meanwhile, users with very low efficiency could be misconfiguring their jobs.

VI. Tools for optimizing individual workflows

A. On Demand Tracing

Once workflows have been identified as promising candidates for optimization, the next step is to perform top-down performance analysis to reveal bottlenecks, estimate possible improvements, and remove the bottlenecks if desirable.

Timeline tracing is one of the best tools for identifying bottlenecks in parallel applications. Many of the bottlenecks that are typical to the GPU workloads we run in our fleet are quickly recognizable when looking at a timeline trace.

Figure 1b shows an example portion of a timeline trace to illustrate the different elements involved. Each colored bar in the diagram represents the duration of a particular event, which can be composed of sub-events, including via RPC to other processes and hosts. There are also other types of dependency relationships, such as when a CPU thread issues an asynchronous GPU kernel launch to the CUDA API.

To support our goal of enabling trace collection for any workflow running in the fleet, the *Kineto* profiling library is able to record GPU traces via the CUPTI API. The GPU traces are combined with CPU activity traces from supported frameworks such as PyTorch. Traces may be collected for any workload running in the fleet at any time without special setup. This on-demand aspect is key to us for several reasons:

- It allows GPU performance experts to dissect any workload running in the fleet without help from workflow owners.
- It allows workflow owners to analyze their workflows with a click of a button. Ease-of-use matters greatly for adoption.
- Collecting traces in production often reveals different bottlenecks than during development and testing.

Figure 1c shows how we extended the telemetry infrastructure in Figure 1a to include timeline tracing. We added a tracing API to the telemetry library, and added support for tracing across process and host boundaries to the RPC mechanism. In the case of a distributed workload, traces may be collected from multiple processes and hosts separately and simultaneously, and merged by the *Trace Collector* or in post process. Metrics and patterns are extracted from traces for fleet-wide analysis.

A key requirement of trace collection is avoiding significant workload performance overhead. Typical duration of trace collection is on the order of seconds, so this performance overhead has minimal effect on workflow efficiency over the entire run, but it can distort the timeline and mask real bottlenecks. In order to keep overhead low, we employ two strategies:

1) A trace warmup period is used to initialize and "warm up" the tracing infrastructure and data structures. In some cases we observe several seconds of large performance impact before the workload stabilizes with a small overhead, typically $< 5\%$. This warmup period is synchronized across different processes and hosts for distributed workloads.
2) The trace is first logged into memory buffers using an efficient and compact format. Only when the trace is complete do we write the trace to one or more destinations, in a background thread, using a more portable format.

For distributed workloads, the ability to capture a trace of a particular time window from multiple processes and hosts involved in an AI workload and combine them into a single trace has proven helpful.

An extensible trace processing pipeline enables metric extraction and pattern detection on traces, which we will explore in more detail in Section VI-C.

Since trace collection is part of a larger performance profiling infrastructure with a web-based GUI, it is also desirable that trace visualization can be done directly in the browser, and that a link to a trace can be easily shared between people, attached to tasks and chat groups and so on. This greatly speeds up collaborative analysis and tuning, as well as allowing GPU experts to quickly investigate and offer advice.

B. *PerfDoctor*

Timeline traces reveal many issues common to GPU workloads. In some cases, however, it is helpful to look at a larger set of metrics relating to other resources, such as system memory, I/O, network, etc.

We continuously collect additional performance data on every machine in the fleet, including:

- System level metrics such as CPU utilization, memory and network bandwidth
- Application data including stacks, memory allocations, throughput counters etc.

PerfDoctor is a tool for marshaling this performance data and presenting it to developers and production engineers in a single screen. It presents a curated set of data views that performance engineers find most useful for analysis. PerfDoctor has an extensive UI which combines links to performance data, detailed analysis, highlighted issues and pointers to potential solutions. The UI is composed of tabs for CPU, GPU, Memory, Network, Application etc.

For ML workflows, PerfDoctor can collect data on-demand by communicating directly to the collection daemons on the hosts. In fact, this is the main mechanism used to collect GPU timeline traces. PerfDoctor communicates with the *Kineto* library in the ML workload via a collection daemon on the host. The traces are collected, compressed, and uploaded to a data repository. From there they can be viewed in the PerfDoctor UI and analyzed using *Automated Trace Comprehension* as described below.

C. *Automated Trace Comprehension (ATC)*

We have found that effective use of timeline tracing requires substantial experience, and in some cases, knowledge of HPC concepts. While data aggregation tools as described in Section V allow our performance engineers to directly address the most pressing performance problems in the fleet, workflows that are not near the top by resource use rarely get personalized attention. Therefore, the question is whether we can have a user-friendly and scalable way to enable non-GPU experts to optimize their model training workloads.

Our answer is Automated Trace Comprehension (ATC), a system which automatically analyzes traces for performance issues and guides optimizations. ATC aims to extract useful information from traces and assist users in these aspects:

- **Hotspot Stats:** The Hotspot Stats report contains high-level stats including the frequency, duration and size distribution of activities (e.g., operators, CUDA runtime, GPU kernels) in collected traces. The top time-consuming operators and kernels are highlighted as performance optimization targets.
- **Anti-Pattern Detection:** In practice, poor performance and resource utilization in ML training workloads usually can be correlated with anti-patterns found in their traces. Typical anti-patterns we found in workloads include “Too little work per CUDA kernel or memcpy”, “Bottlenecks at the CPU causing high GPU idle time”, “Improper grain size per GPU thread”, “Improper memory access patterns”, “Insufficient concurrency”, and so on. ATC scans the traces and reports any anti-patterns it detects. We give examples of these anti-patterns in the next section.

VII. Case Studies

Many GPU performance issues we find in the datacenter result from blind spots about GPU concepts or constraints by workflow authors, who write anti-pattern code that performs poorly. Here are examples of common patterns that we have identified using our tools.

A. *Too little work per CUDA kernel or memcpy*

Blind Spot: Overhead of kernel launches and cudaMemcpy is relatively high ($\sim 5\mu s$). **Anti-Pattern:** An operator for data transformation was implemented using CPU code executing fine-grained cudaMemcpy calls in a loop. CUDA API overhead dominated execution time as shown in Figure 4-(a). **Solution:** Using a GPU kernel that transforms the data in parallel using blocks of GPU threads, we improved the performance of the operator by 200x and of the workflow by 3.5x.

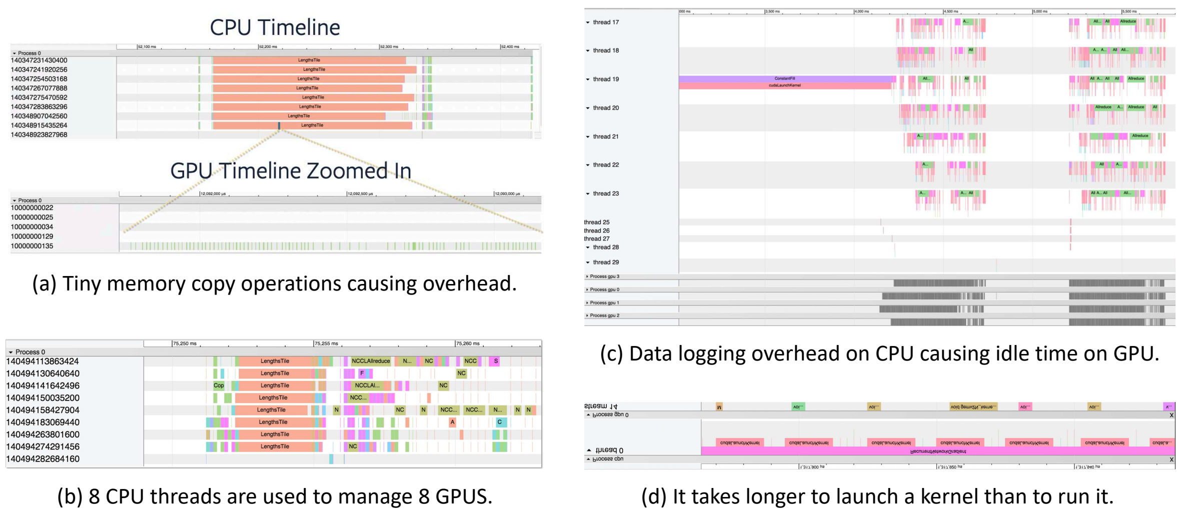
B. *Bottlenecks at the CPU Cause High GPU Idle Time*

Blind Spot: Peak throughput is much higher on the GPU than on the CPU. **Anti-Pattern 1:** Code that performs expensive data transformations on the CPU, causing GPU to go idle for extended time. **Solution 1:** Do as much as possible of the expensive work on the GPU with kernels that take advantage of the available concurrency. **Solution 2:** Run more threads on the CPU to concurrently prepare work for GPU execution to help feed the GPU more effectively. As shown in Figure 4-(b), a workflow used 8 CPU threads to manage the 8 GPUs on the server. Increasing the number of threads to 64 improved overall throughput by 40% by allowing more concurrent operations to run in parallel.

Anti-Pattern 2: Expensive file I/O operations on the CPU causing large idle sections at the end of each iteration as illustrated in Figure 4-(c). **Solution:** Reduce the I/O overhead by decreasing the frequency of data logging.

C. *Improper Grain Size per GPU Thread*

Blind Spot: On the CPU, the work per thread should be substantial (e.g. to absorb context-switch overhead), but GPUs switch between warps of threads very efficiently, so keeping



(a) Tiny memory copy operations causing overhead.

(b) 8 CPU threads are used to manage 8 GPUS.

(c) Data logging overhead on CPU causing idle time on GPU.

(d) It takes longer to launch a kernel than to run it.

Fig. 4: Case Studies: Timeline profiles

thread grain size very low is fine. **Anti-Pattern:** GPU code with too much work per thread artificially limits concurrency, yielding low block count and SM efficiency. **Solution:** Rewrite kernels to expose more concurrency and increase blocks per kernel.

D. Insufficient Concurrency

Blind Spot: GPUs contain thousands of compute units, so code must expose that much concurrency for proper utilization. **Anti-Pattern:** Kernels with low parallelism that utilize a small fraction of SMs, as shown in Figure 4-(d). **Solution:** If the problem inherently has low concurrency, consider running on a CPU instead.

VIII. Conclusion

Performance analysis and optimization of massively parallel workloads is a hard problem, and scaling that across hundreds of workloads running in a heterogeneous GPU fleet is significantly harder, especially when performance experts are in short supply. To our knowledge we presented the first complete fleet-wide GPU performance introspection system for deep-learning training, starting with telemetry enabling deep visibility into the utilization of the massively parallel GPU resources. Tracking key metrics for workflows across the fleet enables us to focus our limited expert resources on the highest-impact areas at any point in time. Powerful and easy to use tools for top-down on-demand analysis allow us to easily spot performance bottlenecks in workloads running on a large number of nodes, from GPU kernels to communication patterns at multiple levels. Approaches such as automated trace comprehension, anti-pattern detection, and actionable recommendations allow us to scale GPU performance work beyond the core experts, and ease the learning curve for newcomers. They also form the foundation for future semi- and fully-automated mechanisms.

References

[1] Matthew D. Jones et al. Workload analysis of blue waters. *CoRR*, abs/1703.00924, 2017.

[2] Myeongjae Jeon et al. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 947–960, USA, 2019. USENIX Association.

[3] Google. Using cloud tpu tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools#top_of_page, 2020.

[4] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. 2017.

[5] Mathew Salvaris and Miguel Fierro. Gpu monitor. https://github.com/msalvaris/gpu_monitor, 2018.

[6] Keji Xu. Monitoring gpu utilization with amazon cloudwatch. https://aws.amazon.com/blogs/machine-learning/monitoring-gpu-utilization-with-amazon-cloudwatch/, 2017.

[7] Amazon. Monitor gpus with cloudwatch. https://docs.aws.amazon.com/dlami/latest/devguide/tutorial-gpu-monitoring-gpumon.html, 2020.

[8] NVIDIA. Cuda profiling tools interface (cupti). https://docs.nvidia.com/cuda/cupti/index.html, 2020.

[9] Kevin Lee. Introducing big basin: Our next-generation ai hardware. https://fb.me/lee_2017, 2017.

[10] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, March 2017.

[11] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[12] Facebook. Kineto profiling library. https://github.com/pytorch/kineto, 2020.