Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training

Hongyu Zhu, *University of Toronto & Vector Institute;* Amar Phanishayee, *Microsoft Research;* Gennady Pekhimenko, *University of Toronto & Vector Institute*

https://www.usenix.org/conference/atc20/presentation/zhu-hongyu

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training

Hongyu Zhu[†], *Amar Phanishayee*[⋆], *Gennady Pekhimenko*[†]
[†]*University of Toronto & Vector Institute* [⋆]Microsoft Research

Abstract

Modern deep neural network (DNN) training jobs use complex and heterogeneous software/hardware stacks. The efficacy of software-level optimizations can vary significantly when used in different deployment configurations. It is onerous and error-prone for ML practitioners and system developers to implement each optimization separately, and determine which ones will improve performance in their own configurations. Unfortunately, existing profiling tools do not aim to answer predictive questions such as "How will optimization X affect the performance of my model?". We address this critical limitation, and proposes a new profiling tool, Daydream, to help programmers efficiently explore the efficacy of DNN optimizations. Daydream models DNN execution with a fine-grained dependency graph based on low-level traces collected by CUPTI [49], and predicts runtime by simulating execution based on the dependency graph. Daydream maps the low-level traces using DNN domain-specific knowledge and introduces a set of graph-transformation primitives that can easily model a wide variety of optimizations. We show that Daydream is able to model most mainstream DNN optimization techniques and accurately predict the efficacy of optimizations that will result in significant performance improvements.

1 Introduction

Recent years have witnessed the co-evolution of deep neural network (DNN) algorithms and the underlying hardware and software design. ML researchers have developed many important models [20, 26, 27, 73] at a rapid pace, creating a huge demand for computation power [69]. To meet the demand for fast DNN computation, computer architects respond with new, AI-optimized GPUs (e.g., NVidia Turing architecture [56]) and various domain-specific hardware accelerators from FPGAs (e.g., Microsoft Catapult [64]) to ASICs (e.g., Google TPU [34], Amazon Inferentia [70]). However these accelerators might not be effective in improving performance without proper software optimizations across the full systems stack [84]. As a result, systems researchers have proposed many optimizations, targeting different bottlenecks across the system stack – for example, improving memory utilization [29, 67], better overlapping of communication with computation [25, 30, 83], and increasing communication efficiency [16]. Moreover, researchers have also developed workload-centric optimizations to exploit the stochastic nature of DNN computation. For example, precision reduction [18, 23, 42] aims to reduce runtime as well as memory consumption, and gradient compression [40, 41] aims at reducing the communication overhead in distributed training.

Despite these advances, the benefits of many proposed optimizations cannot be fully exploited due to two main reasons. First, the efficacy of many proposed performance optimizations can drastically change when applied to different ML models and deployment configurations. The hardware deployments that practitioners use might be completely different from the hardware configurations used by optimization and model inventors. Differences in DNN models, accelerator type, compute capabilities, available memory, networking capabilities, and software library versions can all shift the major runtime bottlenecks. Second, it is onerous for programmers to implement and evaluate various optimizations to identify the ones that actually work for their models. As a result, it is common for users to ask *what-if* questions such as:

Why did my DNN training workload run slowly? Will optimization X improve the performance of my model? Does GPU memory capacity limit the performance of my model? Would upgrading to a faster network improve training throughput? How will my workload scale with the number of GPUs?

The central focus of this paper is to answer the following general question for DNN training workloads: *Given a model and a deployment scenario, how can we efficiently explore the efficacy of potential solutions*? Systems researchers have tried to explore the impact of different potential performance bottlenecks (e.g., CPU, network, IO) in many non-ML contexts [5, 17, 43, 59, 60, 74]. The basic approaches to explore the what-if questions are similar: decompose the workloads into atomic tasks, profile runtime statistics for each task, model the what-if question, and use simulation to estimate performance.

These systems typically address what-if questions of the form: "How does runtime change if a task T is N times (or even infinitely) faster?" [17, 60]. Such questions can be simply modeled by shrinking task runtime. While this basic approach seems sufficient to address the central question above for ML workloads, the **diversity of DNN optimizations** introduces three key requirements unique to these workloads, thus motivating the need for a novel solution.

First, we need to **track dependencies at a kernel-level abstraction** i.e., one GPU kernel corresponds to one task (the smallest unit of execution in the dependency graph). Such fine-grained abstraction is necessary because optimizations that improve hardware utilization typically target individual compute kernels (e.g., mixed precision [42]). Meanwhile, accurate performance estimation has to consider both CPU and GPU runtime. Certain optimizations, e.g., kernel fusion, require potentially removing existing CPU and GPU tasks from the dependency graph. Existing tools do not provide such dependency tracking. It is therefore important to track kernel-level dependencies among concurrently executing tasks.

Second, we need to **map tasks to DNN layers**. In contrast to prior works that explore what-if questions in non-ML contexts, predicting the performance of DNN optimizations requires domain knowledge about DNNs to properly model them. For example, MetaFlow [33] and TASO [32] fuse DNN layers. Modeling them requires a mapping from tasks to specific DNN layers. However, collecting kernel-level traces on accelerators requires generic vendor-provided tools (e.g., NVProf [48], CUPTI [49]), which have no application specific knowledge. We therefore need to have the ability to map low-level tasks to DNN layers.

Third, we need the **ability to easily model diverse DNN optimizations**. Modeling a DNN optimization might involve not just scaling or shrinking task durations, but also complicated transformations to the dependency graph. For example, TicTac [25] reschedules communication tasks, BlueConnect [16] replaces the communication primitives to utilize parallel network channels, and the optimization proposed by Jung *et al.* [35] restructures the GPU kernel implementations. Manually manipulating the kernel-level dependency graph could be extremely intricate and error-prone. The system should enable users to flexibly and effectively model such diverse optimizations with minimal effort.

We introduce *Daydream*, a new system that fulfills all three requirements described above, and achieves our goal of answering potential what-if questions for DNN workloads. Constructing dependencies among potentially thousands of low-level tasks is not an easy problem: tasks can be spread across multiple execution threads (including both CPU threads and GPU streams), thus even for simple DNN workloads, this results in thousands of tasks to be tracked. The intricacy comes from identifying dependencies across threads. We make a key observation about DNN training workloads: despite the large number of tasks that need to be tracked, the number of concurrently executing threads is surprisingly quite limited. Based on this observation, Daydream constructs the low-level dependency graph, which provides a realistic model of overlapping among CPU, GPU, and communication runtimes in a DNN training workload. It uses a synchronization-free approach to map GPU tasks onto appropriate higher-level DNN layer abstractions. We also introduce a set of graph-transformation rules, allowing programmers to effectively model various performance optimizations. After modeling the optimization, Daydream simulates the execution based on the new dependency graph to predict the overall runtime. In our evaluation, we show that Daydream is able to distinguish effective DNN optimizations from those that will bring limited improvements by accurately predicting their performance speedups.

In summary, we make the following key contributions:

- We make the observation that fine-grained tasks in DNN training workloads are highly sequential. This greatly simplifies dependency graph construction, over thousands of tasks, as we only need to identify a limited number of inter-thread dependencies.
- Daydream introduces the abstraction of a kernel-granularity dependency graph that contains mappings back to DNN specific abstractions (layers), by collecting profiling data, instrumenting DNN frameworks, and exploiting information from vendor-provided tools like CUPTI. Daydream also provides primitives to mutate the dependency graph in the form of simple graph transformations. Taken together this enables programmers to both (i) model a diverse set of popular optimizations spanning kernel- and layer-level enhancements by using simple graph-transformation primitives, and (ii) estimate the efficacy of optimizations by simulating execution time based on optimization-induced graph mutations.
- We extensively evaluate Daydream, with *five* different optimizations on *five* DNN models across *three* distinct applications. We show that Daydream can effectively detect which optimizations provide improvements and also accurately predict their magnitude for different DNN models and deployments. For example, we estimate that using mixed precision will improve the iteration time of training $\text{BERT}_{\text{LARGE}}$ model by 17.2% (with <3% error), while the kernel fusion technique can improve it by 38.7% (with <7% error). We can also accurately predict performance in distributed training with different number of workers and variable network bandwidth, based on runtime profiles collected from a single-GPU setting.

2 DNN Training Optimizations and Tools

DNN training is an iterative algorithm, in which one iteration consists of three phases: (i) *forward*, (ii) *backward*, and (iii) *weight update*. The *forward* phase takes training data samples as input and produces output based on current weights

Optimization Goal	Strategy	Technique Examples
Improving Hardware Utilization in Single-Worker Setting	Increasing Mini-batch Size by Reducing Memory Footprints	**vDNN** [67], **Gist** [29], Chen *et al.* [14]
	Reducing Precision	***Micikevicius et al.*** [42], Gupta *et al.* [23], Das *et al.* [18]
	Fusing Kernels/Layers	***FusedAdam*** [52], **MetaFlow** [33], Ashari *et al.* [10], TASO [32]
	Improving Low-level Kernel Implementation	***Restructing Batchnorm*** [35], Tensor Comprehensions [72], Kjolstad *et al.* [37], TVM [13]
Lowering Communication Overhead in Distributed Training	Reducing Communication Workloads	**Deep Gradient Compression** [40], AdaComm [76], Parallax [36], TernGrad [78], QSGD [8]
	Improving Communication Efficiency/Overlap	***Wait-free Backprop*** [83], ***P3*** [30], **BlueConnect** [16], TicTac [25], BytePS [62], Xue *et al.* [80]

Table 1: Representative optimizations for DNN training. We show how we can accurately estimate the performance of optimizations (shown in ***italics***) in Section 6, and can effectively model many other optimizations (shown in **bold**) in Section 5.

(or parameters). The error between the *forward* output and the input data labels is fed to the *backward* phase, which computes the gradients of weights with respect to the input data. The *weight update* phase then uses the gradients to update weights accordingly. In each iteration, the input data samples are randomly selected [11], forming a *mini-batch* of input.

2.1 DNN Training Optimizations

Modern DNNs have millions of parameters [24], resulting in training times of days or even weeks [38]. To improve DNN training performance, researchers have proposed various strategies focusing on different optimization goals. To understand the potential what-if questions and how to design a system to answer them, we study a list of software-level techniques that speedup DNN training from top systems and ML conferences in recent years. Table 1 shows our summary.

Exploiting computation power of hardware accelerators. ML programmers often use large mini-batches, within the memory budget, for better hardware utilization and faster convergence. This motivates strategies that reduce the memory footprint of DNN training and hence enables training with larger mini-batch sizes [14, 29, 67]. Researchers have also proposed some generic strategies to increase hardware utilization, including precision reduction [18, 23, 42], kernel/layer fusion [10, 32, 33], and improving low-level kernel implementation [13, 35, 37, 72]. Meanwhile, libraries such as cuDNN [15], cuBLAS [45], MKL [75], Eigen [1], and NCCL [46] are also constantly evolving to provide operations and primitives that can better utilize underlying hardware.

Scalable distributed training. Data parallelism [11] is a simple and effective strategy to improve training performance. Using multiple accelerators significantly reduces DNN training time to hours or even minutes [44]. This success is mainly based on the techniques that guarantee model convergence under extremely large mini-batch size [7, 22, 81]. One of the major performance bottlenecks for distributed training is communication, which can be optimized by compressing traffic [40, 41, 76, 78], increasing network utilization [16, 80], or increasing the overlap between communication and computation [25, 30, 83]. Exploring the efficacy of these optimizations without prediction requires a multi-machine cluster. Our proposed design, Daydream, avoids the potential cost of cluster setup (i.e. extra machines, accelerators, high-speed communication), by predicting distributed training performance with profiles collected from a single-worker environment.

2.2 Profiling Tools for DNNs

As the full ML system stack is constantly evolving, profiling tools play a key role in helping programmers identify the performance bottlenecks under different system configurations.

Hardware profiling tools. Modern DNN training heavily relies on hardware accelerators such as GPUs [56] and TPUs [34]. To help programmers develop highly efficient applications, hardware vendors provide profiling tools that can expose hardware performance counters. For example, NVProf [48] provides programmers with information including start/end time, core utilization, memory throughput, cache miss rate, along with hundreds of other hardware counters for every GPU kernel. CUPTI [49] enables programmers to extract and manipulate these counters at runtime. Nsight [47] aims to provide details on the state of more fine-grained counters for recent GPU architectures [56]. Our proposed system, Daydream, relies on CUPTI to collect low-level traces for further analysis.

Framework built-in tools. For more intuitive profiling results, it is often desirable for a profiler to show runtime statistics for framework operations, or even DNN layers. DNN frameworks have built-in tools to achieve this goal by correlating the hardware counters with runtime information collected in frameworks. TensorFlow [3], coupled with the Cloud TPU Tool [21], can provide an execution timeline and runtime statistics for each TensorFlow operation. Similarly, other mainstream frameworks (e.g., MXNet [12] and PyTorch [61]) provide built-in tools that can extract per-layer or per-operation runtime from both the CPU and the GPU. The framework built-in tools render intuitive results for pro-

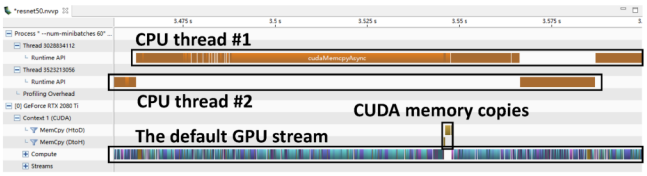


Figure 1: NVProf timeline example of training ResNet-50.

grammers, but omit important details (for example, the CPU runtime). We show in our work that such information is crucial in building an accurate runtime predictor.

3 Key Ideas

In this section we highlight the key ideas and observations behind the Daydream design.

Constructing kernel-granularity dependency graph. The neural network topology is a natural graph structure in which nodes are DNN operators or layers. Most mainstream DNN frameworks [12, 61] provide built-in tools to record the layer-level runtime profile. The layer-level abstraction is intuitive for programmers to understand the "where time goes" question, but hides important information about the parallel execution of the CPU functions, GPU kernels, and memory transfers. This information is crucial for accurate performance predictions. For example, optimizations that reduce numerical precision will change the duration of GPU kernels while the CPU runtime remains unchanged, and optimizations like vDNN [67] will inject CUDA memory copies, without changing the duration of GPU kernels. It is extremely hard to predict how duration of each layer changes when applying these optimizations if lacking low-level details about CPU and GPU runtime. To accommodate optimizations that target fine granularity tasks (such as GPU kernels), our proposed system, Daydream chooses to model the training workloads using a kernel-level dependency graph (i.e., each GPU kernel has one corresponding task in the graph), incorporating detailed traces of CPU, GPU and communication runtime.

With a large number of kernel-level tasks that are spread across several threads and CUDA streams, the complexity of constructing the dependency graph comes mainly from identifying the inter-thread dependencies [74]. Existing tools do not provide such dependency tracking. We make the following key observations about the DNN training workloads to overcome this general challenge of dependency tracking in concurrent systems. First, for the implementations in the mainstream frameworks [12, 61], once a mini-batch has been prepared by data loading threads, only one or two CPU threads are involved in the control flow of computation.[1] Second, there is a very limited number of concurrent GPU kernels. Such serialization of GPU kernels is due to two main reasons: (i) GPU kernels in the modern cuDNN library achieve high GPU core utilization; (ii) ML frameworks usually invoke only one CUDA stream. Figure 1 shows the NVProf profiles of one training iteration of ResNet-50. There are two CPU threads involved, but no CPU tasks run concurrently. The high serialization of low-level traces is not a unique phenomenon for just convolutional networks. We observe a similar phenomenon in most DNN training workloads.

Based on these insights, Daydream constructs the kernel-level dependency graph in three major steps. First, Daydream uses CUPTI to extract traces of all GPU kernels, CUDA memory copies, and CUDA APIs. Second, Daydream captures the dependencies between CPU and GPU tasks, caused by CUDA synchronizations and GPU kernel launches. Third, when predicting performance for distributed training, Daydream adds communication tasks to the dependency graph.

Synchronization-free task-to-layer mapping. In distributed training, mainstream frameworks implement the wait-free backpropagation strategy [83] to overlap communication with computation. This strategy immediately transfers gradients once they are computed by corresponding backward layers. To properly add dependencies related to communication tasks, we need the task-to-layer mapping to know when the computation of each layer ends. Meanwhile, accurately modeling DNN optimizations by changing the graph potentially requires this task-to-layer mapping to determine which tasks are involved and how to change them.

Unfortunately, vendor-provided tools like CUPTI do not have the required knowledge about these applications and building such a mapping requires extra DNN framework instrumentation. A naïve approach to achieve this mapping is to compare the start and stop timestamps of GPU kernels and DNN layers. This requires additional CUDA synchronization calls for each layer since GPU kernels are launched asynchronously. However, such synchronizations might significantly alter the execution runtime by adding additional dependencies from GPU to CPU tasks. Hence, we design a synchronization-free procedure to achieve this mapping by instrumenting timestamps for each layer in the frameworks, and utilizing the correlations between CPU and GPU tasks.

Representing complex optimizations with simple graph-transformation primitives. As shown in Table 1, DNN optimizations target a wide range of performance bottlenecks with various approaches. Unlike prior dependency graph analysis in non-ML contexts [17, 59, 60], where users can model most what-if questions by simply shrinking and scaling task runtime, accurately modeling DNN optimizations with the low-level dependency graph might require complicated changes to the dependency graph. Manually changing the kernel-level graph to model optimizations could be both complicated and error-prone, and the programmers might simply opt to rather directly implement the optimizations.

To address this problem, we propose a small set of graph-transformation primitives, so that popular optimization techniques can be effectively represented as a combination of

[1] Our approach can be generalized to frameworks that use more concurrent CPU threads.

these primitives. These primitives include (i) task insertion/removal, (ii) task selection and update, and (iii) changing the policy for scheduling tasks. The proposed primitives are simple yet powerful enough to represent many different optimizations as we will show in Section 5. They play a key role in realizing our goal of efficiently exploring what-if questions.

In summary, Daydream introduces the abstraction of a kernel-granularity dependency graph that contains mappings back to DNN specific abstractions (layers). It tracks dependencies by collecting profiling data as well as instrumenting DNN frameworks. Daydream also provides primitives to mutate the dependency graph in the form of simple graph transformations. Altogether this enables programmers to both (i) model a diverse set of popular optimizations spanning kernel- and layer-level enhancements by using simple graph-transformation primitives, and (ii) estimate the efficacy of optimizations by simulating execution time based on optimization-induced graph mutations.

4 Design

We describe Daydream's design with an emphasis on how to construct Daydream's proposed graph abstraction: the kernel-granularity dependency graph with mappings back to DNN layers. We also describe the primitives for mutating this graph to model different optimizations and how Daydream uses the graph to estimate the efficacy of various DNN optimizations.

4.1 Overview of Daydream

Figure 2 shows the workflow of performance prediction in Daydream. It consists of the following four phases:

Phase 1: Trace collection. Constructing a kernel-level dependency graph requires low-level details for all tasks. These details are extremely massive, differ across ML frameworks, and can be obtained by profiling a baseline workload. Daydream collects low-level profiling data using CUPTI [49], a tool which provides details for all CPU/GPU tasks including name, start time, duration, CUDA stream ID, thread ID, etc. We manually augment three popular frameworks (Caffe, MXNet, PyTorch) for use with CUPTI and modify the layer modules of these frameworks to collect timestamps of each layer, which will be used for task-to-layer mapping, described in Section 4.3. Through our instrumentation, we also collect the necessary information (e.g., size of gradients) to construct the dependency graph of distributed training via a profile collected in a single worker setting.

Phase 2: Dependency graph construction. Daydream constructs the dependency graph with details of tasks provided by the first phase. A dependency could be induced by domain knowledge (e.g., a GPU task triggers a communication task), or by hardware/software implementation (e.g., a cudaLaunchKernel API triggers the corresponding GPU task). Based on our analysis, we identify five different types of dependencies (described in Section 4.2.2), which are sufficient for Daydream to accurately simulate baseline execution.

Phase 3: Graph transformation. To estimate the efficacy of a given optimization, Daydream models the optimization by transforming the dependency graph. Daydream provides a set of primitives (e.g. selection, insertion/removal) to represent these transformations. We design these primitives in a way such that they are succinct (easy to use), flexible (able to depict a wide range of optimizations), and accurate (being able to achieve high prediction accuracy).

Algorithm 1: Daydream's Simulation Algorithm

Input : Dependency graph: $G(V,E)$
Output : The start time of each task $u \in V$

```
1  F ← ∅ // initialize the frontier task set
2  P ← {0} // initialize thread progress
3  foreach task u ∈ V do
4      u.ref ← |{u'sparents}|
5      if u.ref = 0 then
6          F ← F ∪ {u}
7  end
8  while F ≠ ∅ do
9      u ← schedule(F) // pick a task to exec.
10     t ← u.ExecutionThread
11     F ← F − {u}
12     u.start ← max(P[t], u.start)
13     P[t] ← u.start + u.duration + u.gap
14     foreach c ∈ u.children do
15         c.ref ← c.ref − 1
16         c.start ←
             max(c.start, u.start + u.duration + u.gap)
17         if c.ref = 0 then
18             F ← F ∪ {c}
19         end
20     end
21 end
```

Phase 4: Runtime simulation. Daydream simulates the execution of optimizations to predict runtime based on the dependency graph. Algorithm 1 shows the simulation process, which traverses the dependency graph and puts tasks into execution threads. In each iteration, Daydream picks one task from the execution frontier (i.e. tasks that are ready to execute), dispatches it to its corresponding execution thread, and updates the thread progress. The simulation determines the start time of each task and records the total execution time.

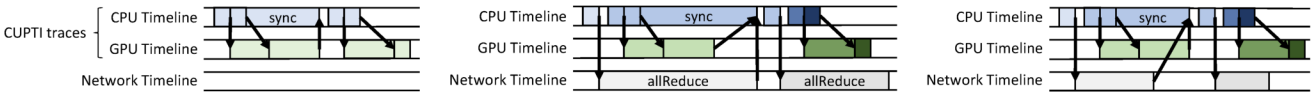
4.2 Dependency Graph Construction

Constructing the dependency graph is essential to determine the node (task) set and edge (dependency) set.

4.2.1 Task

Daydream's kernel-level dependency graph contains the following four types of tasks:

GPU tasks. Each GPU task in the graph corresponds to one GPU kernel. Daydream also views CUDA memory copies as



(a) Constructing the dependency graph based on CUPTI traces (the black arrows represent task dependencies).

(b) Mapping each task to DNN layers (shown in different colors in the figure), then inserting communication tasks based on mapping and instrumentation.

(c) Predicting "what if network bandwidth is 2×" by shrinking allReduce duration by 2× and simulating the new dependency graph.

Figure 2: An example showing Daydream's overall workflow for predicting runtime assuming network bandwidth doubles.

GPU tasks, because each memory copy is associated with a specific CUDA stream, and therefore has dependencies with other GPU kernels. The runtime of all these tasks can be collected using CUPTI.

CPU tasks. To model the concurrency and dependencies between CPU runtime and the GPU runtime, Daydream generates CPU tasks based on CPU traces collected by CUPTI. One of the limitations of CUPTI is that it can only expose CUDA-related traces. Instead of adding massive instrumentation to the framework, Daydream captures the non-CUDA runtime by recording the lengths of gaps between consecutive CPU tasks (shown in line 13 of Algorithm 1).

Data loading tasks. One data loading task corresponds to loading one mini-batch from disk/flash to CPU memory. We include data loading tasks for completeness, even though data loading in most DNN training workloads is not a performance bottleneck. In Daydream's implementation, we treat all data loading tasks as CPU tasks.

Communication tasks. A communication task corresponds to one communication primitive, e.g., a push/pull operation in parameter-server based frameworks [39], or an all-reduce operation in decentralized frameworks. When predicting distributed training performance, Daydream automatically adds communication tasks to the dependency graph based on a single-worker profile. We notice that in PyTorch, gradients from multiple layers can be grouped and sent with a single allReduce primitive [2]. Thus, properly adding communication tasks to a PyTorch profile requires additional instrumentation to extract knowledge about gradients grouping.

Given the types of tasks in the graph, Daydream collects and maintains the following information for each task, which is later used in what-if analysis and simulation:

ExecutionThread. Depending on the type of a task, its execution thread can be on of the following: (i) a CPU process, (ii) a GPU stream, and (iii) a communication channel. A data loading task is executed in a CPU process. A CPU process has a process ID, a GPU stream has a stream ID, and a communication channel could be send/receive when using parameter server primitives, or a unified one when using collective primitives. This field is used in line 10 of Algorithm 1.

Duration. This field specifies how long a task takes to execute. The duration of a CPU/GPU task is collected by CUPTI. The runtime of data loading tasks is measured by injecting timestamps to the framework. Daydream aims to predict distributed training performance based on profiling in a single-GPU configuration. Hence we calculate the duration of all communication task based on the size of gradients, the communication type (push/pull/all-reduce), and the network bandwidth. These numbers can be obtained based on knowledge of the DNN model and framework implementation.

Gap. The duration of low-level CUDA APIs (e.g., `cudaMalloc`) might be only tens of microseconds, which is of the same magnitude as the runtime of their non-CUDA equivalent C functions (e.g., `malloc`), or the runtime of the call stack from Python front-end to C back-end. NVidia-provided tools cannot expose non-CUDA traces, but they are indispensable to simulation accuracy. The non-CUDA CPU runtime is usually not a target for optimization in DNN models, hence, we do not need to define and measure corresponding tasks. Instead, for each CPU task in our current definition, we measure the gap between its end and the start of the next task in the same execution thread, and simulate these gaps in Algorithm 1.

Layer. This field refers to which DNN layer a task belongs to, which is necessary information for programmers to transform the graph and model optimizations. Daydream uses a synchronization-free approach to map a task to DNN layers. We will describe the details of this approach in Section 4.3.

4.2.2 Dependency

Based on our discussion in Section 3, we identify the following five types of dependencies for accurate simulations.

Sequential order of CPU tasks in the same thread. CPU tasks in the same thread are serialized. The order that CPU tasks are executed in is determined by the framework and does not change in two separate executions. We add a dependency between each two consecutive CPU tasks in the same thread.

Sequential order of GPU tasks in the same CUDA stream. GPU kernels belonging to the same CUDA stream are executed sequentially. Similar to CPU tasks, the order of GPU tasks in the same stream does not change between executions. Hence, two consecutive GPU tasks in the same CUDA stream have a dependency between them.

Correlation from CUDA APIs to GPU kernels. Each GPU kernel or CUDA memory copy has a corresponding CPU-sided CUDA API (`cudaLaunch`, `cudaMemcpy`, or `cudaMemcpyAsync`) that triggers the GPU task. CUPTI provides a correlation ID for every CUDA API and GPU kernel. A GPU kernel is dependant on a CUDA API if they share the same correlation ID.

CUDA Synchronization. A CUDA synchronization API

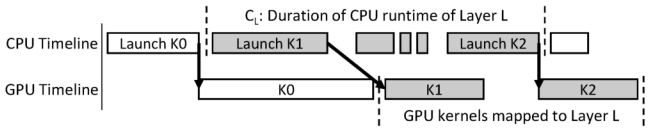


Figure 3: The mapping of GPU kernels to a layer. CUPTI provides correlations between CUDA launches and GPU kernels.

(e.g., `cudaDeviceSynchronize`) is invoked on CPU, and returns after GPU kernels (or CUDA memory copies) that are launched before this synchronization complete. A CUDA synchronization therefore generates dependency from a GPU task to a CPU task. Similar to CUDA synchronizations, even though a `cudaMemcpyAsyncDtoH` call returns before a memory copy completes, we found it still blocks the CPU until all previous GPU kernels on the same stream are completed.

Communication. Mainstream frameworks including PyTorch and MXNet implement the wait-free backpropagation strategy [83] to schedule gradient communication. Here, a communication primitive is launched as soon as the weight gradients are ready, thus overlapping communication with the backward phases of subsequent layers. Hence, we need to know the runtime of DNN layers (not just kernels) to determine which tasks trigger communication.

4.3 Mapping Tasks to Layers

The task-to-layer mapping enables Daydream to construct the dependency graph for distributed training, and provides necessary domain knowledge for Daydream to model DNN optimizations. Figure 3 shows how Daydream determines which tasks belong to a certain layer. Let L be the forward phase of a DNN layer. Daydream collects the CPU and GPU runtime information using CUPTI [49], as well as timestamps before and after the forward, backward, and weight update phases for each layer. The start and end timestamps of L will determine the CPU runtime of L (denoted by C_L). To determine the GPU runtime of L, Daydream gathers all CUDA launch calls invoked during C_L. With CUPTI providing the correlations between CUDA launch calls and corresponding GPU kernels, Daydream can identify all the GPU kernels launched during C_L, and map these kernels to L. This process can also be applied to the backward or weight update phases of any layers, and can be further generalized to any code region of interest in the framework or user-level programs.

4.4 Graph Transformation

What-if analysis by transforming the graph and simulating the execution requires input about the optimizations from programmers. Daydream provides a set of primitives for programmers to model DNN optimizations by modifying the graph. Like most what-if analysis in non-ML contexts, modeling DNN optimizations requires potentially shrinking or scaling the duration of tasks (the `shrink/scale` primitives). We carefully study common DNN optimization techniques and identify the following primitives (besides the `shrink/scale` primitives), which are sufficient for programmers to describe those optimizations.

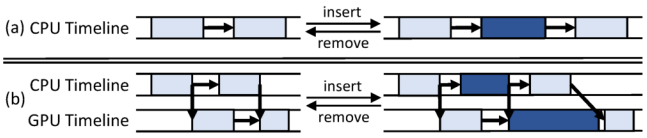


Figure 4: Insert/Remove a (a) CPU task; (b) GPU task.

Insert/Remove a task. Inserting a task to an execution thread just involves an appending of a node to a linked list. Figure 4 shows how this process works. When inserting a GPU task, we need to insert the corresponding CPU tasks that launch it. Which CPU tasks to insert and their duration depend on the framework implementation, and can be inferred based on collected traces.

Select. This operation allows users to select tasks of interest for further operations. One potentially useful selection criterion is select-by-layer, as many optimizations are depicted based on DNN layers. Another potentially useful criterion is to select by keywords in task names, based on knowledge of the software library (e.g., cuDNN [50]). For example, kernels with keywords such as `elementwise` or `PointwiseApply` in the names are element-wise arithmetic operations. These kernels are typically *not* compute-bound, and could be much shorter than their corresponding CUDA launch calls. Similarly, kernels with `sgemm` string in names are compute-bound matrix-multiplications.

Schedule. The `schedule` function picks one task from a set of frontier tasks that are ready to execute (line 9 in Algorithm 1). By default, it picks the task with the earliest start. Programmers can override this function and implement any custom scheduling policy, which is useful to model optimizations that increase computation-communication overlap.

5 Modeling Optimizations

To demonstrate that Daydream is able to estimate the performance of the most common optimizations in DNN training, we select ten techniques from Table 1 with different optimization goals. We show that we can easily model these optimizations using the primitives Daydream provides. [2]

5.1 Optimizations for Evaluation

We select the following five DNN optimizations, which we are able to acquire the implementations, to evaluate Daydream's prediction accuracy. We use implementations from the authors of these optimizations in cases where they were not readily available.

Automatic Mixed Precision (AMP). We aim to predict the efficacy of the AMP optimization [42], implemented using NVidia's Apex package [51]. We expect that AMP will

[2] We show pseudo code for AMP in this section. Refer to our arxiv version [85] for the pseudo code of all examples shown in Section 5.

improve memory-bounded GPU kernels by 2× because the number of transferred bits is halved. With Tensor Cores in the Volta and Turing architectures, AMP empirically yields up to 3× speedup on the most compute-intensive workloads [58]. To predict AMP performance, we simply `select` all the compute-intensive (e.g., `sgemm`, `conv`) kernels and memory-bounded (e.g., `elementwise`, `batchnorm`, `RELU`) kernels, and `shrink` their duration by 3× and 2× respectively. We show the pseudo code for modeling AMP in Algorithm 3.

Algorithm 2: What_If_AMP

Input : Dependency graph: $G(V,E)$
Output : A modified graph $G(V,E)$ to model AMP

```
1 GPUTasks ← {G.Select(funcPtr(IsOnGPU))}
2 foreach u ∈ GPUTasks do
3     if "sgemm" in u.Name or "scudnn" in u.Name then
4         u.duration ← u.duration/3
5     else
6         u.duration ← u.duration/2
7     end
8 end
```

FusedAdam Optimizer. We use the FusedAdam optimizer [52] implemented in NVidia's Apex package [51] as an example for the kernel fusion optimization. This optimizer fuses all kernels in one weight update phase into one unified kernel. It is applicable to the models that use the Adam optimizer (e.g., GNMT, BERT). Daydream uses the kernel-to-layer mapping to identify the CPU/GPU tasks that belong to a weight update phase. We `remove` all these tasks, then `insert` a new GPU task whose duration is roughly estimated by the sum of all removed compute-intensive kernels.

Reconstructing Batchnorm. Recently Jung et al. [35] proposed a technique that optimizes non-convolutional layers in state-of-the-art CNNs. It first splits each batch normalization layer into two sub-layers, then fuses the first sub-layer with the previous convolutional layer, and the second sub-layer with the following activation and convolutional layers. We `remove` the affected activation kernels when estimating performance, since they are memory-bound kernels now fused with compute-intensive convolutional kernels. For the batch nomalization layers, we estimate that the GPU kernels will be improved by 2× since this optimization halves the amount of input data that these layers load from GPU memory.

Distributed Training. Using Daydream we can accurately predict distributed training performance with the profile based on the single-GPU environment. We evaluate Daydream's prediction based on PyTorch, which uses collective communication primitives from the NCCl library [46]. PyTorch groups gradients from multiple layers into buckets before transferring them. Hence, to predict distributed training performance, we need to `insert` one allReduce task for every bucket. The dependencies of the inserted tasks are determined based on the layer-to-bucket mapping (which requires additional instrumentation to the PyTorch framework).

Priority-Based Parameter Propagation (P3). P3 [30] is a technique that optimizes communication overhead by slicing and prioritizing. We evaluate Daydream's prediction of P3 based on MXNet, which uses the parameter-server mechanism [39]. In order to model parameter slicing, we `insert` multiple push task and pull tasks between the backward and the forward GPU tasks for each layer. The duration of the push/pull task is calculated from the slice size and the network bandwidth. To model the priority scheduling, we override the `schedule` function with a priority queue.

5.2 Modeling Additional Optimizations

In addition to the above optimizations, we show that Daydream is capable of modeling an additional set of diverse DNN optimizations.

BlueConnect. BlueConnect [16] optimizes communication by decomposing the allReduce primitives into a series of reduce-scatter and all-gather primitives. These primitives run concurrently as they use parallel communication channels. To predict the performance of BlueConnect, instead of `insert`ing regular allReduce or push/pull tasks, we need to `insert` reduce-scatter and all-gather tasks, and assign them to corresponding network channels (the duration can be estimated according to formulas shown in [57]).

MetaFlow. MetaFlow [33] is a layer-fusion technique to optimize DNN training by fusing DNN layers to simplify the DNN topology. We `select` the GPU kernels of substituted layers, `remove` them, and `insert` GPU kernels of new layers to predict the performance of MetaFlow in Daydream. The new layers are mostly existing layers with different dimensions; their GPU kernel durations can be inferred by profiling.

vDNN. Virtualized DNN [67] reduces GPU memory consumption by temporarily offloading intermediate data from GPU memory to CPU memory. The offloaded data needs to be prefetched back to GPU to perform execution, which causes potential performance overhead due to PCIe traffic or late prefetching. To predict the performance overhead using Daydream, we only need to `insert` additional CUDA memory copies, and override the `schedule` function to implement a custom prefetching policy.

Gist. Gist [29] reduces GPU memory consumption by storing encoded intermediate data and decoding before the data is used. The encoding and decoding introduces performance overhead. We `insert` extra encoding and decoding GPU kernels (along with `cudaLaunchKernel` calls in CPU) to estimate the performance overhead in Daydream. The duration of the inserted encoding/decoding kernels can be estimated using existing element-wise kernels.

Deep Gradient Compression (DGC). DGC [40] is a technique that reduces communication overhead by compressing the gradients. To estimate performance, we: (i) `scale` the duration of communication; (ii) `insert` the GPU tasks of compression and decompression. The duration of inserted

Application	Model	Dataset
Image Classification	VGG19 [71]	ImageNet [19]
	DenseNet-121 [28]	
	ResNet-50 [27]	
Machine Translation	GNMT [79]	WMT16 [4]
Language Modeling	BERT [20]	SQuAD [65]

Table 2: The models and datasets we use in this paper.

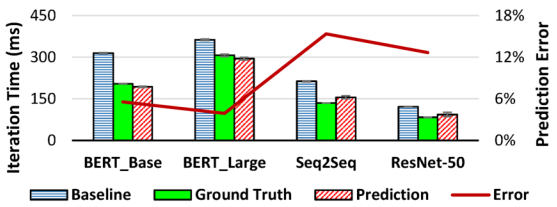


Figure 5: AMP – comparing baseline (FP32), ground truth with mixed precision, and predictions by Daydream.

GPU tasks can be estimated according to the compression rate and duration of existing element-wise GPU kernels.

6 Evaluation

6.1 Methodology

We implement Daydream based on three mainstream DNN frameworks: PyTorch [61], MXNet [12], and Caffe [31]. We add CUPTI [49] support to each framework to obtain traces of CUDA APIs and GPU kernels. We also add instrumentation to the frameworks to acquire layer-wise timestamps for the kernel-to-layer mapping process, and communication information such as the size of each allReduce call and their dependencies with other layer-wise computation.

Infrastructure. We evaluate Daydream's runtime prediction on a cluster of four machines. Each machine contains one AMD EPYC 7601 16-core processor [9], and four 2080Ti GPUs [55] with 11GB GDDR6 memory each, connected through PCIe 3.0 [6]. Our experiments are based on Ubuntu 16.04, CUDA v10.0 [53], cuDNN v7.4.1 [54], and NCCL v2.4.2 [46]. Our software implementation is based on PyTorch v1.0, MXNet v1.1, and Caffe v1.0.

Models. Table 2 shows the DNN models and datasets we use to evaluate Daydream. We select five DNN models from three different applications, covering a diverse set of DNN models. For the BERT model, we evaluate both "base" and "large" versions. The difference between these versions is that the "base" version contains 12 "Transformer blocks" (the main layer type in BERT) where as the "large" version contains 24.

6.2 Automatic Mixed Precision (AMP)

We evaluate Daydream's prediction accuracy of AMP [42], which is implemented in NVidia's Apex package [51] based on the PyTorch framework. Figure 5 shows the performance of using AMP and the corresponding performance prediction

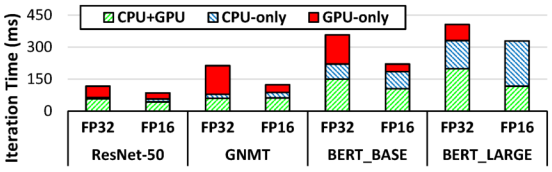


Figure 6: Runtime breakdown of the baseline (FP32) and mixed precision (FP16).

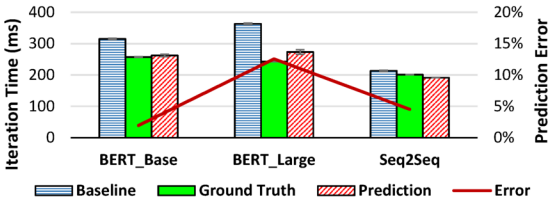


Figure 7: FusedAdam - comparing baseline (FP32), ground truth with FusedAdam, and predictions by Daydream.

given by Daydream. Our predictions have errors below 13% for all the models we evaluate.

Our experiments show that using AMP brings speedups generally less than 2× – much less than the theoretical boost of using AMP for individual kernels (e.g., 3×). To understand how AMP improves performance, we break down the overall runtime into the following three components:

CPU-only runtime. This component refers to the runtime when the CPU is busy, but the GPU is not executing any kernels. It is straightforward to calculate this runtime by simply subtracting all GPU kernel runtime from the total runtime.

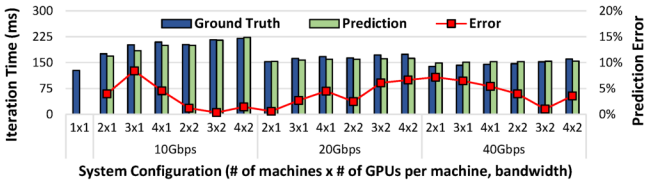
GPU-only runtime. This component refers to the runtime when the CPU is waiting for the GPU kernels to complete. It includes not only the duration of CUDA synchronization APIs, but also the `cudaMemcpyAsync` calls of all the device-to-host CUDA memory copies.

CPU+GPU parallel runtime. This component refers to the runtime when both CPU and GPU are busy. We calculate this part of runtime by deducting the CPU-only and GPU-only parts from the total runtime.

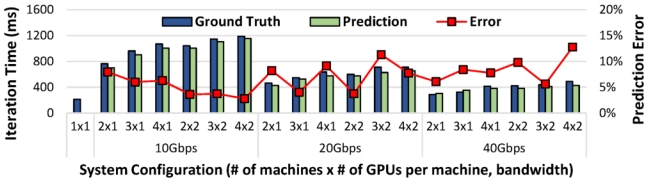
Figure 6 shows the runtime breakdown of the models we evaluated. CPU runtime generally becomes the new performance bottleneck in the models that incur limited speedups (e.g., $\text{BERT}_{\text{LARGE}}$). When applying AMP, the CPU bottleneck increases, because the GPU runtime becomes shorter and part of the CPU+GPU parallel runtime is shifted to the CPU-only runtime. The overall runtime improvement comes mostly from the reduction of GPU-only runtime while CPU runtime barely changes. This demonstrates the necessity of the kernel-level abstraction when predicting performance.

6.3 FusedAdam Optimizer

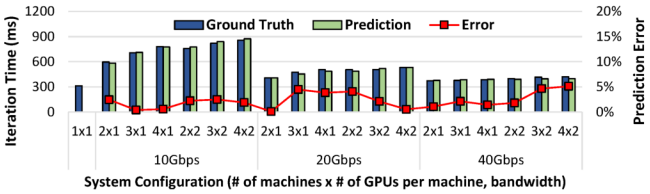
We apply the FusedAdam optimization to the BERT and GNMT models as they use the Adam optimizer. Figure 7 shows the performance of using the FusedAdam optimizer. Our predictions are within 13% of the ground truth runtime.



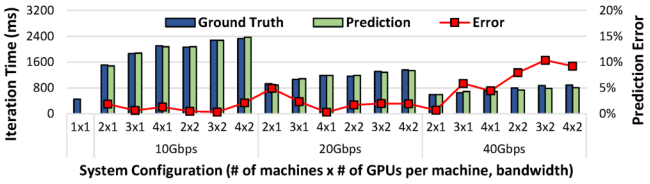
(a) Runtime predictions for ResNet-50.



(b) Runtime predictions for GNMT.



(c) Runtime predictions for $\text{BERT}_{\text{BASE}}$.



(d) Runtime predictions for $\text{BERT}_{\text{LARGE}}$.

Figure 8: The error between Daydream's runtime predictions and the baseline with synchronization before each allReduce under various system configurations.

There are two reasons why the FusedAdam optimizer substantially improves the performance of BERT models. First, unlike most DNN training workloads, the weight update phase is a significant proportion of a BERT model's iteration runtime (around 30% for $\text{BERT}_{\text{BASE}}$ and 45% for $\text{BERT}_{\text{LARGE}}$). Second, the weight update phase consists of very many element-wise GPU kernels (2633 for $\text{BERT}_{\text{BASE}}$, 5164 for $\text{BERT}_{\text{LARGE}}$). Thus, the CUDA launch calls on the CPU become the main bottleneck. The FusedAdam optimizer almost eliminates all CPU kernel launch overhead in the weight update phase by fusing all GPU kernels into one single GPU kernel. Compared to BERT models, the GNMT model spends less than 10% of its iteration time on the weight update phase, explaining the lower speedup improvements.

6.4 Reconstructing Batchnorm

We evaluate our performance prediction for the optimization of reconstructing batch normalization [35] based on the Caffe implementation of DenseNet-121 [28]. Using Daydream, we predict that reconstructing batchnorm will yield a moderate performance improvement of 12.7% compared to the baseline.

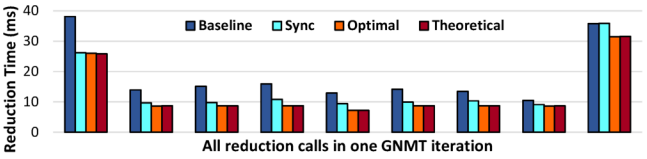


Figure 9: Comparison of all individual reduction runtimes in one training iteration of GNMT. **Baseline**: runtime measured in regular training; **Sync**: runtime measured with an additional CUDA synchronization before each reduction; **Optimal**: runtime measured when executing exclusively; **Theoretical**: runtime calculated using the formula [57].

This suggests that reconstructing batchnorm in our configuration is less promising than the paper claims (17.5% speedup). We verify this conclusion by testing the ground truth implementation of reconstructing batchnorm, and find out that this optimization yields even lower 7% speedup.

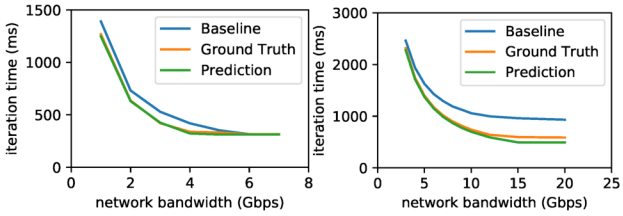
We notice that there are two main reasons for the difference between our prediction and the ground truth. First, the ground truth uses a completely new implementation of the batchnorm layers, and it is hard to precisely predict the runtime of newly implemented kernels. Second, the ground truth implementation introduces new CUDA memory copies and allocations, which add performance overhead. Obtaining a very precise estimate would require us to understand not just the high-level idea from the paper, but also the detailed implementation of the user-level programs and the Caffe framework.

6.5 Distributed Training

Next we evaluate distributed training using PyTorch with the NCCL [46] library. Figure 8 shows the comparisons between runtimes predicted by Daydream and the measured ground truth runtimes, for each DNN model under different system configurations. We evaluate the prediction accuracy for Ethernet and InfiniBand connecting multi-machine systems under different network bandwidths (10, 20, 40 Gbps). In most of the configurations, Daydream predicts distributed runtime with at most 10% prediction error, with a few exceptions for the 20Gbps and 40Gbps configurations.

The prediction errors of the overall iteration times are mainly due to inaccurate estimates of individual NCCL primitives. Figure 9 shows the comparisons of NCCL allReduce calls between the ground truths and predictions. The ground truths are on average 34% higher than the theoretical values.

An NCCL primitive is both a communication primitive and a GPU kernel, suggesting that it could be bottlenecked by two types of hardware resources: (i) the network bandwidth, and (ii) GPU resources (e.g., memory bandwidth, streaming multiprocessors). Figure 9 shows that the predicted values are very close to the runtimes measured when running NCCL primitives exclusively. This suggests that the ground truth is slower because they compete for GPU resources with other GPU kernels. Based on this insight, we try to reduce this interference by adding CUDA synchronizations before invoking NCCL



(a) ResNet-50. (b) VGG-19.

Figure 10: Daydream's prediction for how the P3 optimization will help under different network bandwidths.

primitives. As shown in Figure 9, adding synchronizations improve the NCCL primitives by 22.8% on average when compared to the baseline.

We also verify the impact to the overall iteration time when adding synchronizations before NCCL primitives. We run the experiments on all the configurations shown in Figure 8. We find that this simple approach does not lead to performance degradation in any configuration. Instead, it could bring an improvement of up to 22%.

6.6 Priority-Based Parameter Propagation

We evaluate Daydream's prediction accuracy of applying Priority-Based Parameter Propagation (P3) to VGG-19 and ResNet-50. To reproduce the performance speedups of P3, we use a cluster of four machines with one P4000 GPU per machine (which is consistent with the evaluation setup of the P3 paper [30]). We use MXNet v1.1, and have one worker process and one parameter server process on each machine.

Figure 10 shows the iteration time of the baseline, ground truth, and prediction using Daydream under different bandwidths. Our prediction faithfully reflects the trend of P3 speedups when the network bandwidth increases. The prediction error is at most 16.2% among all the configurations we tested, and lower in most of the configurations.

We overestimate the speedup of P3, especially when training VGG-19 with a 15 or 20 Gbps network bandwidth. The reason is similar to our previous insight about NCCL primitives: when bandwidth is higher, a communication task is increasingly bottlenecked by non-network resources. In the case of MXNet, this overhead could be caused by the server processes, or the control flow of the worker processes.

7 Discussion

In this section, we discuss the adaptability, potential extensions, and some limitations of Daydream.

Why Not Simply Run the Optimizations? The main problem many ML developers face is that not all optimizations are readily available on all platforms. In fact, we are only able to evaluate the prediction accuracy of optimizations with the implementations already available (see Table 1); for the remaining ones, we highlight the flexibility of Daydream by showing that they can be represented succinctly. Most newly proposed optimizations do not have open-source implementations on **all** DNN frameworks available right away; it would be unreasonable to expect researchers to open-source their implementations and port their optimizations on all platforms. Therefore, analyzing if these optimizations can help in a deployment setting, using Daydream, can still precede the programming effort to port the optimizations. Furthermore, Daydream's profiling can be performed just once, and using that profile on a given platform, one can answer questions for many different optimizations.

Adaptability of Daydream Daydream requires support from hardware profilers. The current implementation of Daydream utilizes GPU-based profilers, and it relies on CUPTI to provide: (i) CPU and GPU traces and (ii) information about which CPU call triggered the launch of a specific GPU kernel. Adapting our design to other architectures (e.g., TPUs), would require hardware vendor profilers to provide similar traces for this new hardware.

Daydream can be also easily adapted to other ML frameworks (e.g., MXNet and TensorFlow). We built Daydream based on PyTorch, and then post-process the dumped traces to make predictions. The post-processing scripts are framework-independent. To add framework instrumentation, we need to: (i) add CUPTI (or similar tool) support, (ii) insert per-layer timestamps, and (iii) gather the gradient-to-bucket mappings for injecting the communication primitives to the dependency graph (required for PyTorch). Such instrumentation is relatively light-weight and can be easily adapted to other mainstream frameworks such as TensorFlow [3] and MXNet [12].

Training Accuracy Prediction In addition to improving iteration time, some optimizations may also affect training accuracy (e.g., AMP [42], DGC [40]); predicting the impact of optimizations on accuracy is currently outside of Daydream's scope. We leave this interesting and challenging problem for future work.

Kernel Runtime Prediction Estimating the effect of optimizations that alter existing GPU kernels or introduce new ones requires predicting the runtime of new/changed GPU kernels. When estimating performance of AMP, our estimation of kernels that use half-precision kernels was based on findings/observations from NVIDIA [42]. This generalization above for all kernels (in contrast to identifying how each kernel in isolation is affected by AMP), still leads to the low prediction errors we observe in Figure 5.

However, optimizations such as DGC [40], Reconstructing Batchnorm [35], and Gist [29] introduce newly-implemented kernels to the runtime. Accurately predicting runtime for new kernels is a challenging problem. Daydream estimates the overall runtime based on existing kernel implementations, or using guidelines from studies that highlight quantitative improvements for the proposed kernels. But if the estimated runtimes for such new kernels are inaccurate, it may lead to relatively high prediction error (Section 6.4). How much a kernel's runtime estimation error contributes to the overall

prediction error depends on the training workload itself. Due to this limitation, it is hard for Daydream to accurately model algorithmic innovations (e.g., BPPSA [77] or 2nd Order Optimizations [68]), because these innovations use new GPU kernels at a massive scale, making the performance estimation with Daydream less accurate. Estimating new GPU kernels runtime is beyond the current scope of Daydream.

While Daydream cannot predict individual kernel runtime, it provides a high-level structure for kernel developers to estimate the overall performance. Developers can profile their individual kernels, and then input the profiling results into Daydream to accurately estimate the overall runtime. This approach saves the engineering effort of porting the kernel implementation into the DNN frameworks.

Concurrent Kernels Existing GPU profilers such as CUPTI usually serialize GPU kernel execution, removing all concurrency, making our performance estimation somewhat conservative. Despite this, we observe that the runtime for models with concurrent execution (e.g., GNMT) can still be predicted with high accuracy (§ 6.2). This is because the majority of computation time goes to fully connected layers (including embedding layers), which have no concurrent kernels executed in parallel with them. We leave a complete solution for concurrent kernels, requiring better support from profiling tools, as a part of future work.

8 Related Work

To help programmers understand the performance of the hardware accelerators and develop highly efficient applications, hardware vendors provide profiling tools (e.g., NVProf [48], Nsight [47], and vTune [66]) that can reveal low-level performance counters (e.g., cache hit rate, memory speed or clock rate). These tools are usually designed with general applications in mind, and expose hundreds of low-level performance counters. The fundamental limitation of all these tools is that they do not utilize application-specific knowledge.

The new generation of profiling tools feature the *application-aware* property, enabling them to deliver domain-specific (e.g., ML-specific) insights about performance to programmers. The Cloud TPU Tool [21] is an example of such a profiling tool. It correlates low-level TPU metrics with the DNN structure, and shows the performance for each DNN layer. Similarly, MXNet [12] and PyTorch [61] also have their own built-in profiling tools. These domain-specific tools can highlight performance hotspots, but are less efficient in finding optimization opportunities. In contrast, Daydream is not only *application-aware*, but also *optimization-aware*, enabling Daydream to quantitatively estimate the efficacy of different optimizations without fully implementing them.

Prior works have tried to explore what-if questions in other contexts by using low-level traces. Curtsinger *et al.* proposed a causal profiler (COZ [17]) to identify potentially unknown optimization opportunities by running performance simulation with certain functions being virtually speed-up. Unlike Daydream, COZ does not require dependencies among functions because it does not consider the cases where functions can be added or deleted (which is the case for many ML optimizations). Pourghassemi *et al.* uses the idea of COZ to analyze the performance for web browser applications [63]. For data analytic frameworks, such as Spark [82], Ousterhout *et al.* use dependency analysis to understand the overhead caused by I/O, network, and stragglers [59,60]. Daydream is designed to address a more diversified set of what-if questions, and hence requires more powerful modeling.

Prior works address what-if questions of the form "What if we can speedup task T by N times (or infinity)?", but they do not study whether existing optimizations can deliver this speedup. In the ML context, given an optimization, accurately predicting the performance of individual tasks in the dependency graph, is still an open problem. It requires additional knowledge about the kernel implementation and the architecture design. Currently Daydream can not automatically estimate the runtime of new GPU kernels. However, as we show in Section 6, even with rough estimates of per-kernel duration based on domain knowledge and reasonable assumptions, we can still achieve high overall prediction accuracy.

9 Conclusion

The efficacy of DNN optimizations can vary largely across different DNN models and deployments. Daydream is a new profiler to effectively explore the efficacy of a diverse set of DNN optimizations. Daydream achieves this goal by using three key ideas: (i) constructing a kernel-level dependency graph by utilizing vendor-provided profiling tools, while tracking dependencies among concurrently executing tasks; (ii) mapping low-level traces to DNN layers in a synchronization-free manner; (iii) introducing a set of rules for programmers to effectively describe and model different optimizations. Our evaluation shows that using Daydream, we can effectively model (i.e. predict runtime) the most common DNN optimizations, and accurately identify both optimizations that result in significant performance improvements as well as those that provide limited benefits or even slowdowns.

Acknowledgement

Daydream is a part of Project Fiddle at Microsoft Research (MSR). We thank the MSR Lab LT, especially Ricardo Bianchini and Donald Kossmann, for their enthusiastic and unwavering support of Project Fiddle. We also thank our shepherd, Swaminathan Sundararaman, the anonymous ATC reviewers, Jorgen Thelin, Shivaram Venkataraman, Deepak Narayanan, and the EcoSystem group members, especially James Gleeson, Geoffrey Yu, and Xiaodan (Serina) Tan for their constructive feedback and comments. This work was also supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, and Huawei grants.

References

[1] Eigen: A C++ linear algebra library. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[2] PyTorch Documentation. https://pytorch.org/docs/stable/index.html, 2019.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] ACL. Shared Task: Machine Translation of News. http://www.statmt.org/wmt16/translation-task.html, 2016.

[5] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 74–89. ACM, 2003.

[6] Jasmin Ajanovic. PCI Express*(PCIe*) 3.0 Accelerator Features. *Intel Corporation*, 10, 2008.

[7] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[8] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[9] AMD. AMD EPYC™ 7601. https://www.amd.com/en/products/cpu/amd-epyc-7601, 2019.

[10] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. In *ACM SIGPLAN Notices*, volume 50, pages 173–182. ACM, 2015.

[11] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, pages 1–15, 2018.

[14] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[16] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1–1, 2019.

[17] Charlie Curtsinger and Emery D Berger. C oz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[18] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] Google. Cloud TPU Tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools, 2018.

[22] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[23] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[24] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR 2016)*, 2016.

[25] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[26] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[28] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[29] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceeding of the 45st Annual International Symposium on Computer Architecture*, ISCA 2018, pages 776–789, 2018.

[30] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *Proceedings of Machine Learning and Systems 2019*, pages 132–145. 2019.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[32] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62. ACM, 2019.

[33] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN computation with relaxed graph substitutions. In *Proc. Conference on Systems and Machine Learning, SysML*, volume 19, 2019.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA 2017, pages 1–12, New York, NY, USA, 2017. ACM.

[35] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. Restructuring batch normalization to accelerate CNN training. *arXiv preprint arXiv:1807.01702*, 2018.

[36] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 43. ACM, 2019.

[37] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[39] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[40] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[41] Qu Lu, Wantao Liu, Jizhong Han, and Jinrong Guo. Multi-stage Gradient Compression: Overcoming the Communication Bottleneck in Distributed Deep Learning. In *International Conference on Neural Information Processing*, pages 107–119. Springer, 2018.

[42] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[43] Barton P Miller and Cui-Qing Yang. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. In *ICDCS*, pages 482–489, 1987.

[44] MLPerf. MLPerf Training Results v0.6. https://mlperf.org/training-results-0-6, 2019.

[45] NVIDIA. CUDA implementation of the standard basic linear algebra subroutines (BLAS). http://docs.nvidia.com/cuda/cublas/index.html.

[46] NVIDIA. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[47] NVIDIA. NVIDIA Nsight. https://developer.nvidia.com/tools-overview.

[48] NVIDIA. NVIDIA Profiler. docs.nvidia.com/cuda/profiler-users-guide/index.html.

[49] NVIDIA. The CUDA Profiling Tools Interface (CUPTI). https://docs.nvidia.com/cuda/cupti/index.html.

[50] NVIDIA. cudnn library developer guide v6.0. 2017.

[51] NVIDIA. A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex, 2018.

[52] NVIDIA. API Documentation of NVidia's Apex optimizers. https://nvidia.github.io/apex/optimizers.html, 2018.

[53] NVIDIA. Cuda toolkit documentation v10.0. https://docs.nvidia.com/cuda/, 2018.

[54] NVIDIA. cudnn library developer guide v 7.4.1. 2018.

[55] NVIDIA. GEFORCE® RTX 2080 Ti. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti, 2018.

[56] NVIDIA. NVIDIA Turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[57] NVIDIA. Performance reported by NCCL tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md, 2018.

[58] NVIDIA. Training With Mixed Precision: Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html, 2019.

[59] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200. ACM, 2017.

[60] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[61] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[62] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29. ACM, 2019.

[63] Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):27, 2019.

[64] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[65] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[66] James Reinders. VTune performance analyzer essentials. *Intel Press*, 2005.

[67] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 18:1–18:13, Piscataway, NJ, USA, 2016. IEEE Press.

[68] Tomer Koren Kevin Regan Yoram Singer Rohan Anil, Vineet Gupta. Second Order Optimization Made Practical. *arXiv preprint arXiv:2002.09018*, 2020.

[69] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597*, 2019.

[70] Amazon Web Services. AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia.

[71] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[72] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[74] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 185–196. ACM, 2008.

[75] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[76] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. *arXiv preprint arXiv:1810.08313*, 2018.

[77] Shang Wang, Yifan Bai, and Gennady Pekhimenko. Bppsa: Scaling back-propagation by parallel scan algorithm. In *Proceedings of Machine Learning and Systems 2020*, pages 451–469. 2020.

[78] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[79] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[80] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 44. ACM, 2019.

[81] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[82] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[83] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[84] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100. IEEE, 2018.

[85] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. *arXiv preprint arXiv:2006.03318*, 2020.