Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs

Cheng Li*, Abdul Dakkak*
University of Illinois Urbana-Champaign
Urbana, USA
{cli99, dakkak}@illinois.edu

Jinjun Xiong
IBM T. J. Watson Research Center
Yorktown Heights, USA
jinjun@us.ibm.com

Wen-mei Hwu
University of Illinois Urbana-Champaign
Urbana, USA
w-hwu@illinois.edu

***Abstract*—As Deep Learning (DL) models have been increasingly used in latency-sensitive applications, there has been a growing interest in improving their response time. An important venue for such improvement is to profile the execution of these models and characterize their performance to identify possible optimization opportunities. However, the current profiling tools lack the highly desired abilities to characterize ideal performance, identify sources of inefficiency, and quantify the benefits of potential optimizations. Such deficiencies have led to slow characterization/optimization cycles that cannot keep up with the fast pace at which new DL models are introduced.**

We propose *Benanza*, a sustainable and extensible benchmarking and analysis design that speeds up the characterization/optimization cycle of DL models on GPUs. *Benanza* consists of four major components: a model processor that parses models into an internal representation, a configurable benchmark generator that automatically generates micro-benchmarks given a set of models, a database of benchmark results, and an analyzer that computes the "lower-bound" latency of DL models using the benchmark data and informs optimizations of model execution. The "lower-bound" latency metric estimates the ideal model execution on a GPU system and serves as the basis for identifying optimization opportunities in frameworks or system libraries. We used *Benanza* to evaluate 30 ONNX models in MXNet, ONNX Runtime, and PyTorch on 7 GPUs ranging from Kepler to the latest Turing, and identified optimizations in parallel layer execution, cuDNN convolution algorithm selection, framework inefficiency, layer fusion, and using Tensor Cores.

I. Introduction

The past few years have seen a spur of deep learning (DL) innovations. These innovations span from DL models to software stack optimizations (e.g. frameworks such as MXNet or PyTorch, libraries such as cuDNN or MKL-DNN) and hardware stack improvements (e.g. CPU, GPU, FPGA). Among all the innovations, however, DL models are the most rapidly evolving and prolific. This is true in both academia [1] and industry [2], where models are tweaked and introduced on a weekly, daily, or even hourly basis.

Both industry and academia have invested heavily in developing benchmarks to characterize DL models and systems [3], [4], [5], [6], [7]. Characterization is followed by optimizations to improve the model performance. However, there is currently a gap between the benchmarking results and possible optimizations to perform. Researchers use profilers, such as nvprof [8], Nsight [9], and VTune [10], to profile and get low-level GPU and CPU information. With ample knowledge of how models execute and utilize system resources, researchers manually identify bottlenecks and inefficiencies within model execution using the profilers. Researchers then make hypotheses of solutions, and try out different ideas to optimize the model execution — which may or may not pan out. This manual and ad-hoc process requires a lot of effort and expertise and slows down the turnaround time for model optimization and system tuning.

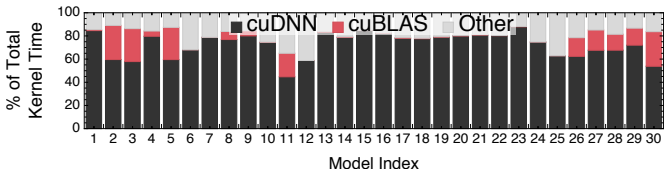

Fig. 1. The GPU kernel time breakdown for all 30 models (listed in Table I) on Tesla_V100 (Table III) using batch size 1. Both cuDNN and cuBLAS invoke child GPU kernel(s) asynchronously. Therefore, we measure the time of the kernels launched by the cuDNN and cuBLAS APIs rather than the time of the APIs themselves for accurate characterization of latencies.

Thus there is a need for a systematic DL benchmarking and subsequent analysis design that can guide researchers to potential optimization opportunities and assess hypothetical execution scenarios. Since for GPUs model execution latency is determined by the hardware, framework, and system libraries (primarily cuDNN [11] and cuBLAS [12] for DL), answers to the following questions are highly desired by researchers: Q1 what is the potential latency speedup if optimizations are performed? Q2 Are independent layers executed in parallel? Q3 Are convolution layers using the optimal convolution algorithms? Q4 Are there any inefficiencies or unexpected behavior in a framework? Does the execution Q5 fuse layers or Q6 leverage Tensor Cores, and what are the benefits? We motivate our design by answering these 6 questions, while ensuring the sustainability and extensibility of the design.

To answer these questions, we first propose a new benchmarking metric: *"lower-bound" latency*. The "lower-bound" latency

*The two authors contributed equally to this paper.

estimates the ideal latency of a DL model given a software and hardware stack, and is based on the following observations: (1) DL models are executed as layers in frameworks and thus layers form the performance building blocks of DL models. (2) Frameworks delegate execution of common layers to either cuDNN or cuBLAS (shown in Figure 1). The "lower-bound" latency is defined in terms of the latencies of the cuDNN and cuBLAS API functions corresponding to the model layers (framework overhead and memory transfers are ignored). We refine the "lower-bound" latency and define it under *sequential execution mode* (all layers are executed sequentially) and *parallel execution mode* (data-independent layers are executed asynchronously).

This paper presents *Benanza* (pronounced bonanza) — a sustainable and extensible benchmarking and analysis design. *Benanza* consists of a set of modular components: (1) a model processor to process input ONNX models into a set of *unique layers* (layers are considered the same if they have the same layer type, shape, and parameters), (2) a benchmark generator to automatically generate parameterized cuDNN and cuBLAS micro-benchmarks from the unique layers, (3) a performance database to store historical benchmark results, and (4) an analyzer to compute the "lower-bound" latency of DL models and inform potential optimizations (Q1-6).

Benanza is architected to be sustainable. The benchmarking workflow of *Benanza* is highly automated and minimizes the benchmark development and maintenance effort. *Benan za* uses the observation that DL models have repeated layers (i.e. non-unique) within and across models to decrease the time to benchmark. When a new model is introduced, only the new, un-benchmarked layers (not in the performance database) need to be benchmarked. Although the focus of the paper is on NVIDIA GPUs using cuDNN and cuBLAS, the design proposed is extensible and users can incorporate other benchmark runtimes that target other software libraries or hardware such as: frameworks' API or MKL-DNN for CPUs.

In summary, this paper makes the following contributions:

- We propose a "lower-bound" latency metric for DL models based on the observation that the latency of a DL model is bounded by the latencies of the cuDNN and cuBLAS API calls corresponding to the model layers. The "lower-bound" latency metric estimates the ideal latency of a model given a specific GPU hardware and software stack.
- We present *Benanza*, a novel benchmarking and analysis system designed to automatically generate micro-benchmarks given a set of models; compute their "lower-bound" latencies using the benchmark data; and inform optimizations of their execution on GPUs. *Benanza* is sustainable and extensible to cope with the fast evolution of DL innovations.
- Using *Benanza*, we characterized the "lower-bound" latencies of 30 ONNX models (shown in Table I) using MXNet, ONNX Runtime, and PyTorch on 7 systems (shown in Table III). We performed a comprehensive "lower-bound" latency analysis as we vary the model, execution mode, batch size, and system. E.g., when using parallel execution mode, up to $2.87\times$(with a geometric mean of $1.32\times$ across models) latency speedup could be made to MXNet using batch size 1 on the `Tesla_V100` system.
- We identified optimization opportunities through *Benanza* in cuDNN convolution algorithm selection (up to $1.32\times$ geometric mean speedup across models), inefficiencies within MXNet (up to $1.15\times$ speedup across models) and PyTorch (up to $2.3\times$ speedup using batch size 1) frameworks, and layer fusion and Tensor Cores (up to $1.09\times$ and $1.72\times$ speedup for `ResNet50-v1` respectively). We further demonstrated that when performed jointly, these optimizations achieve up to $1.95\times$ speedup for `ResNet50-v1` across systems and batch sizes.

II. Background and Motivation

A. DL Model Execution and ONNX Format

A DL model is an execution graph where each vertex is a layer operator (e.g. convolution, activation, normalization, pooling, or softmax). These layer operators (or *layers* for short) are functions defined by a DL framework. A framework executes a model by traversing the model graph in topological order and enqueuing the layers into an execution queue. Although sequential evaluation is always valid, frameworks strive to execute data-independent layers within the queue in parallel. Through execution scheduling, a framework can overlap communication with computation, run two data-independent layers in parallel, etc. Regardless of the execution strategy, however, layer execution latency is the limiting factor for model execution. Therefore, layers are not only the building blocks by which developer define models, but are also the atomic components that define a model's performance characteristics.

Each framework provides its own API, layer definition semantics, model storage format, and model executing strategy. To increase interoperability between frameworks, there has been concerted effort [13], [14] to standardize layer definitions and model exchange format. A leading effort is the Open Neural Network Exchange Format (ONNX), which has wide industry and framework backing. Frameworks such as Caffe2, CNTK, MXNet, Paddle, PyTorch, and TensorRT readily support ONNX, and converters exist for other frameworks such as Caffe and TensorFlow. To perform a fair comparison between frameworks (by evaluating them using the same ONNX model), and more importantly, to make *Benanza* framework-agnostic, we choose ONNX as the model input format for *Benanza*. ONNX hosts all their models publicly [15] and, we select 30 vision models out of the 32 models available at the time of writing for evaluation (the 2 models not selected are non-vision models). The selected models cover an array of tasks and are listed in Table I. We refer to these models by their IDs throughout the paper.

B. cuDNN and cuBLAS

Much like BLAS or LAPACK are the backbone of HPC computing, cuDNN and cuBLAS form the backbone of the GPU software stacks for DL. cuDNN is a GPU-accelerated library which provides highly tuned functions that implement DL layers such as convolution, pooling, normalization, activation. cuBLAS is a GPU-accelerated BLAS library which provides

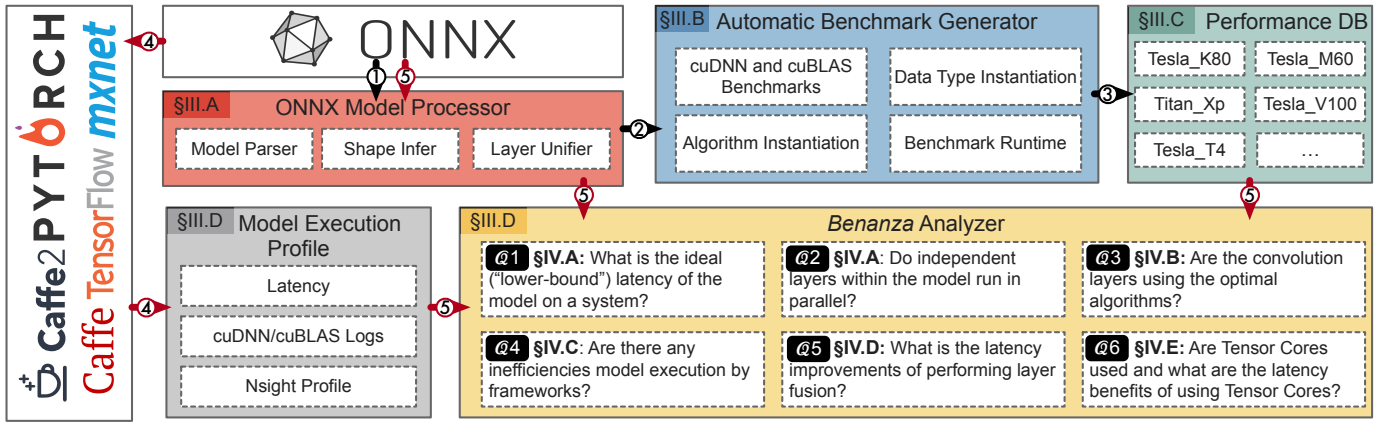

Fig. 2. The *Benanza* design and workflow.

TABLE I
THE 30 ONNX MODELS USED ARE VISION MODELS WHICH ENCOMPASS IMAGE CLASSIFICATION (**IC**), OBJECT DETECTION (**OD**), FACE RECOGNITION (**FR**), EMOTION RECOGNITION (**ER**), SEMANTIC SEGMENTATION (**SS**), OR HAND DIGIT RECOGNITION (**HR**) TASKS.

ID	Name	Task	MACs	# Layers	Year
1	Arcface [16]	FR	12.08G	412	2018
2	BVLC-Alexnet [17]	IC	656M	24	2012
3	BVLC-Caffenet [17]	IC	721M	24	2012
4	BVLC-Googlenet [18]	IC	1.59G	143	2014
5	BVLC-RCNN-ILSVRC13 [19]	IC	718M	23	2013
6	Densenet-121 [20]	IC	2.87G	910	2016
7	DUC [21]	SS	34.94G	355	2017
8	Emotion Ferplus [22]	ER	877M	52	2016
9	Inception-v1 [23]	IC	1.44G	144	2015
10	Inception-v2 [24]	IC	2.03G	509	2015
11	LeNet [25]	HR	796K	12	2010
12	MobileNet-v2 [26]	IC	437M	155	2017
13	Resnet18-v1 [27]	IC	1.82G	69	2015
14	Resnet18-v2 [28]	IC	1.82G	69	2016
15	Resnet34-v1 [27]	IC	3.67G	125	2015
16	Resnet34-v2 [28]	IC	3.67G	125	2016
17	Resnet50-v1 [27]	IC	3.87G	175	2015
18	Resnet50-v2 [28]	IC	4.10G	174	2016
19	Resnet101-v1 [27]	IC	7.58G	345	2015
20	Resnet101-v2 [28]	IC	7.81G	344	2016
21	Resnet152-v1 [27]	IC	11.30G	515	2015
22	Resnet152-v2 [28]	IC	11.53G	514	2016
23	Shufflenet [29]	IC	127M	203	2015
24	Squeezenet-v1.1 [30]	IC	352M	66	2016
25	Tiny Yolo-v2 [31]	OD	3.13G	32	2016
26	Vgg16-BN [32]	IC	15.38G	54	2014
27	Vgg16 [32]	IC	15.38G	41	2014
28	Vgg19-bn [32]	IC	19.55G	63	2014
29	Vgg19 [32]	IC	19.55G	47	2014
30	Zfnet512 [33]	IC	1.48G	22	2013

fast implementations of GEMM and GEMV. The DL layers supported by each API are listed in Table II. And, while there is a wide array of DL frameworks, common between them is the reliance on the primitives defined by cuDNN and cuBLAS. In fact, all major DL frameworks, such as MXNet, PyTorch, ONNX Runtime, and TensorFlow, rely on cuDNN/cuBLAS API functions for the implementation of common layers.

Figure 3 shows the percentage of layers supported by cuDNN and cuBLAS for each model in Table I. Most layers within DL models are covered by the cuDNN and cuBLAS API. The layers that are not supported are non-compute operators (such

TABLE II
ELEVEN LAYER TYPES ARE SUPPORTED BY CUDNN AND TWO LAYER TYPES ARE SUPPORTED BY CUBLAS. EACH API MAY HAVE AUXILIARY FUNCTIONS TO SETUP ITS ARGUMENTS (E.G. `cudnnSetTensor4dDescriptor` TO SPECIFY A TENSOR'S DIMENSIONS AND `cudnnSetConvolution2dDescriptor` TO CONFIGURE THE CONVOLUTION API). THE CONVOLUTION, RNN, AND GEMM APIS HAVE TENSOR CORE SUPPORT.

Layer Type	cuDNN / cuBLAS API	Tensor Core Support
Convolution	`cudnnConvolutionForward`	✓
Activation	`cudnnActivationForward`	✗
BatchNorm	`cudnnBatchNormalizationForwardInference`	✗
Conv+Bias+Activation	`cudnnConvolutionBiasActivationForward`	✓
RNN	`cudnnRNNForwardInference`	✓
Dropout	`cudnnDropoutForward`	✗
Pooling	`cudnnPoolingForward`	✗
Softmax	`cudnnSoftmaxForward`	✗
Add	`cudnnAddTensor`	✗
Element-wise	`cudnnOpTensor`	✗
Rescale	`cudnnScaleTensor`	✗
GEMM	`cublas*Gemm / cublasGemmEx`	✓
GEMV	`cublasSgemv`	✗

as concatenate, which joins two tensors across a specified axis) or datatype manipulations (such as reshape, which changes the dimensions of a tensor). For example, the cuDNN and cuBLAS functions support 70% of the `Inception-v2` (ID = 10) layers. This is because `Inception-v2` makes heavy use of unsqueeze — a tensor reshape layer — and 27% of the layers in `Inception-v2` are unsqueeze layers.

Given a specific DL software stack (e.g. framework, cuDNN, cuBLAS, driver, and other CUDA libraries) and GPU hardware, the cuDNN and cuBLAS functions invoked by a model are fixed. Most common layers are supported by cuDNN and cuBLAS and the latency attributed to cuDNN and cuBLAS functions is significant with respect to the model's compute latency. Figure 1 shows that for the 30 vision models, the time spent within the cuDNN and cuBLAS API calls dominates the model's GPU kernel time. The "other" time is either memory operations or framework GPU kernels which are neither cuDNN nor cuBLAS API calls.

Based on the above observations, we propose a "lower-bound" latency metric for DL models, which is defined by the latencies of the cuDNN and cuBLAS API functions corresponding to the model layers given a specific software/hardware stack.

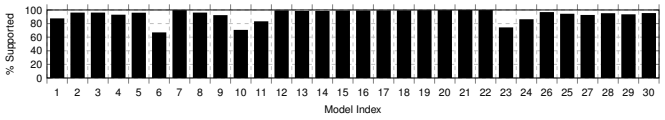

Fig. 3. The percentage of layers supported by cuDNN and cuBLAS (also covered by *Benanza*) for each model in Table I.

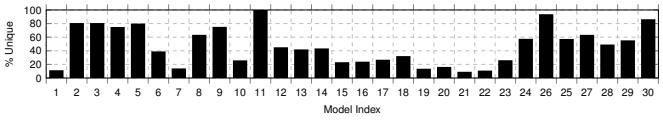

Fig. 4. The percentage of unique layers within the 30 models

The "lower-bound" latency forms an *ideal* latency, which we use to understand how to improve the model's latency. We compute the "lower-bound" latency under different execution scenarios to determine if optimizations can be made, pinpoint where optimization opportunities are, and quantify the potential benefits of optimizations, as detailed in Section III.

III. *Benanza* Design and Implementation

Benanza consists of four main components: Model Processor, Automatic Benchmark Generator, Performance Database, and Analyzer. The components are shown in Figure 2 and are used in the benchmarking and analysis workflows:

- **Benchmarking workflow:** ① The Model Processor takes ONNX models as input, parses them, performs shape inference, and finds the set of unique layers within the models. Two layers are considered the same (non-unique) if they have the same operator type, shape, and parameters (i.e. **only differ in weight values**). ② The Automatic Benchmark Generator then generates micro-benchmarks for each unique layer. The generated micro-benchmarks measure the latency (or the GPU kernel metrics if profiling mode is enabled) of the corresponding cuDNN or cuBLAS function calls for the layers. ③ The micro-benchmarks are then run on systems of interest and the results are stored in the Performance Database.
- **Analysis workflow:** ④ The user runs the target model using a framework on a system of interest with utilities provided by *Benanza* to get the model execution profile (i.e. the model's latency, cuDNN and cuBLAS logs, and Nsight profile). ⑤ The user then specifies the model and system to *Benanza*. The model is parsed into layers and the Analyzer queries the latencies of each layer from the Performance Database (using the layers and system information provided) to compute the Q1 "lower-bound" latency under different execution scenarios. By analyzing the model execution profile and the computed "lower-bound", the Analyzer informs optimizations in: Q2 parallel execution of independent layers, Q3 convolution algorithm selection, Q4 framework inefficiency, Q5 layer fusion, and Q6 Tensor Core usage.

A. Benanza Model Processor

The ① Model Processor parses ONNX models into *Benanza*'s internal representation (IR). The IR wraps around the ONNX Protobuf and has the same layer coverage. Since ONNX models do not have layer shapes information embedded (except for the input layers), shape inference [34] is performed to determine the shape of each layer. Layers in the IR (referred to as *layers* and correspond to the ONNX nodes) are annotated with the inferred shapes. Benchmarks are generated for each layer using its type, shape, and parameters information.

We observe that layers with the same type, shape, and parameters (i.e. **only differ in weight values**) are repeated extensively within and across models. Figure 4 shows that most models have a low percentage of unique layers — indicating that layers are repeated extensively within the model. For example, `ResNet50-v1` (ID=17) has 175 layers but only 47 (26.9%) are unique. The number of unique layers across models of similar architecture is also low. The `ResNet*-v1` models (ID=13, 15, 17, 19, 21) are built from the same modules and have a total of 1229 layers, of which only 60 (5.6%) are unique. Across all 30 models, the total number of layers is 5754, but only 1031 (18%) are unique. We exploit this layer repeatability to optimize the benchmark generation and minimize the time to benchmark. Thus, the Model Processor unifies the repeated layers across the input models and produces a set of unique layers. The time saved can be used to explore other algorithms and data types (Sections III-B2 and III-B3) benchmarks.

B. Automatic Benchmark Generator

The ② Automatic Benchmark Generator uses the set of unique layers (produced by the Model Processor) and generates C++ code to invoke the benchmark runtime using each layer's type, shape, and parameters information.

1) The Benchmark Runtime: Benanza provides a benchmark runtime that measures the latency of the cuDNN or cuBLAS APIs required to execute each layer (as shown in Table II). The runtime also sets up the function arguments for each API. The setup time is not included in the latency measurement. The runtime uses the Google Benchmark [35] library — a micro-benchmarking support library. The Google Benchmark library dynamically determines the number of iterations to run each benchmark and ensures that the reported latency results are statistically stable. Generated benchmarks are linked with the cuDNN/cuBLAS libraries, and are run on systems of interest.

2) Algorithm Instantiation: The convolution layers map to the `cudnnConvolutionForward` API (Table II). The convolution API takes one of the following 8 algorithms as an argument: Implicit GEMM (IGEMM), Implicit PreComputed GEMM (IPGEMM), GEMM, Direct (DRCT), FFT, Tiled FFT (TFFT), Winograd (WING), and Winograd Non-Fused (WINGNF). These algorithms have different compute and memory characteristics [36], [37]. The optimal algorithm to use depends on the system, layer shape, and layer parameters (e.g. filter size, stride, dilation, etc.) [11]. For inference, most frameworks (e.g. MXNet, PyTorch, TensorFlow) rely on the cuDNN provided heuristic function (`cudnnGetConvolutionForwardAlgorithm`) to choose the convolution algorithm. The heuristic function suggests an algorithm given the layer's shape, parameters, data type, system, etc. To explore the design space of algorithm selection, by

default, for each layer *Benanza* generates benchmarks using all algorithms applicable to the layer.

3) Data Type Support: Benanza can be configured to generate micro-benchmarks that target different data types. Both `float16` and `float32` are generated by default, but benchmarks can be instantiated for other data types. The `float16` benchmarks use Tensor Cores when the API function (see Table II) and the system (see Table III) support it.

4) Layer Fusion Support: Benanza can be configured to generate micro-benchmarks that target the cuDNN fused API (`cudnnConvolutionBiasActivationForward`) to perform the convolution, bias, and activation layer sequence. Two fusion pattern rules are currently handled by *Benanza*: Conv→Bias→Activation and Conv→Bias. The Conv→Bias→Activation maps directly to the fused API. Fusing Conv→Bias is implemented through the fused API using `CUDNN_ACTIVATION_IDENTITY` as the activation function and requires cuDNN version $\geq$ 7.1. For older cuDNN versions, the Conv→Bias is implemented as two calls — a `cudnn ConvolutionForward` followed by a `cudnnAddTensor`. Users can extend *Benanza*'s fusion support by registering new fusion patterns as the cuDNN fused API evolves.

5) Integration with CUPTI: Benanza can be configured to generate benchmarks that integrate with low-level GPU profiler libraries such as NVIDIA's CUPTI [38]. This integration allows *Benanza* to capture detailed GPU metrics [39] of benchmarks such as flops, memory transfers, etc. In this mode, the user specifies the metrics of interest, the number of benchmark iterations for warm-up, and the number of iterations to measure. *Benanza* does not use the Google Benchmark in this mode since a fixed, small number of profiling runs suffice for statistically stable measurement of the metrics. The profiling outputs (name, timing, and metric values of GPU kernels) are stored as metadata to the corresponding benchmark entry in the Performance Database.

C. Performance Database

The ③ benchmarking results are collected and published to *Benanza*'s Performance Database. Each entry within the database is indexed by the system, data type, and layer (type, shape, and parameter information). The Analyzer queries the database to get the benchmark latencies. If a query is a miss, then a warning with the information about the missing benchmark is issued to the user and the user is asked if they wish the Automatic Benchmark Generator to generate the missing benchmarks.

D. Benanza Analyzer

The ④ user runs the target model using a framework on a system of interest with utilities provided by *Benanza* to get the *model execution profile*. The model execution profile contains information about the model's latency, cuDNN and cuBLAS logs, and Nsight profile (which contains cuDNN/cuBLAS API calls and function backtrace information). Capturing the model's latency requires the user to place the provided timing functions within their application code. To capture the usage

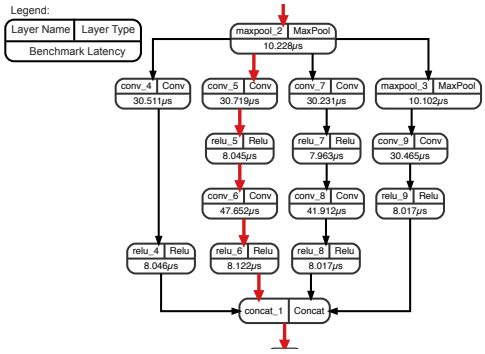

Fig. 5. The first parallel module of `Inception-v1` in Figure 8 visualized by the *Benanza* Analyzer. The layers are annotated with the name, type, and latency used for the "lower-bound" calculation. The critical path used in the parallel mode is highlighted in red.

of cuDNN and cuBLAS functions within a framework, *Benanza* launches the user code with the `CUDNN_LOGINFO_DBG` and `CUBLAS_LOGINFO_DBG` environment variables. These environment variables enable the cuDNN and cuBLAS loggers respectively. Utilities to run the user code using NVIDIA's Nsight profiler are also provided. The results from Nsight are parsed and correlated with the cuDNN and cuBLAS logs.

The ⑤ user then inputs the model execution profile along with the ONNX model, system, data type. The model is parsed by the Model Processor into layers. Then, the *Benanza* Analyzer queries the Performance Database for the benchmark latencies of each layer using the user-specified system and data type (by default `float32`). Due to algorithm (Section III-B2) instantiation, multiple benchmarks may exist for a layer. The Analyzer, therefore, selects the benchmark result achieving the lowest latency. The following analyses are then performed:

1) Q1,2 *Sequential and Parallel "Lower-Bound" Latency:* DL models may contain layer sequences which can be executed independently in parallel. The sub-graph formed by these data-independent layer sequences is called a *parallel module*. For example, a parallel module in `Inception-v1` is shown in Figure 5. A framework may execute the independent paths within the parallel module either sequentially or in parallel. Thus, the Analyzer computes the "lower-bound" latency of a model using two execution modes: sequential and parallel.

The sequential mode assumes that independent layers are executed sequentially, and therefore is defined as the sum of each layer's benchmark latency. The parallel strategy assumes that data-independent layers are executed in parallel. Therefore, the parallel "lower-bound" latency is defined by the model's *critical path* — the simple path from the start to the end layer with the highest latency. Finding the critical path of a graph is a longest path problem and is NP-hard. Since a DL model forms a directed acyclic graph (DAG), the critical path can be framed as a shortest path problem [40]. To compute the critical path we construct a weighted DAG from the model graph where the edge weight between two nodes (layers) is negative of the latency of the layer at the tail of the edge. Computing the shortest path from the start to the end layer of the constructed weighted DAG produces the critical path of

the model. The parallel "lower-bound" latency is the sum of layers latencies along the critical path. *Benanza* visualizes the critical path of the model (e.g. Figure 5), and the difference between the sequential and parallel "lower-bound" latencies indicates the profit of executing independent layers in parallel. Other analyses performed by *Benanza* leverage the sequential and parallel "lower-bound" latencies, and the benefits can be calculated in terms of either sequential or parallel mode.

2) **Q3** *Convolution Algorithm Selection:* The Analyzer uses the parsed cuDNN log in the model execution profile to determine if the cuDNN algorithm used by the framework for each layer is optimal (recall from Section III-B2 that benchmark results using all available algorithms for layers exist in the Performance Database). Cases where the algorithm choice is sub-optimal are reported to the user along with how much latency improvement could be gained if algorithm selection was ideal. The user can act upon these suggestions by forcing the framework to use a specific algorithm for each layer.

3) **Q4** *Framework Inefficiency Inspection:* The expected cuDNN and cuBLAS API calls are known to the Analyzer from the "lower-bound" latency computation. The Analyzer compares the model execution profile against the expected execution to pinpoint inefficiencies within the framework. The user is presented with any deviation observed in cuDNN or cuBLAS API invocation's parameters or their execution order. CUDA API functions and CUDA kernels executed between cuDNN or cuBLAS API calls, are also presented to the user — along with their backtraces.

4) **Q5** *Layer Fusion Analysis:* If the user enables the benchmark generation for layer fusion (as described in Section III-B4), then the Analyzer can be used to determine the potential profitability if layer fusion is employed. The Analyzer traverses the model layers and looks for the fusion pattern rules (listed in Section III-B4). If one of these patterns is found, then the corresponding fused operation's latency is queried from the database and is used in the "lower-bound" computation (in either sequential or parallel mode). If the benchmark is unavailable, or failed to run, then the latencies of the non-fused layers are used. The difference between the non-fused "lower-bound" latency and the fused "lower-bound" latency determines the profitability of layer fusion.

5) **Q6** *Tensor Core Analysis:* The Analyzer determines if the target model execution utilizes Tensor Cores by looking at kernel names in the model execution profile. Kernel names that match the `_[ish]\d+*` Regular-expression use Tensor Cores. By default, benchmarks targeting both `float16` and `float32` are generated. When benchmarks are run on systems with Tensor Core support, the difference between the "lower-bound" latency of `float32` and `float16` informs the profitability of using Tensor Cores with `float16`.

E. Sustainability and Extensibility

The sustainability of *Benanza* is ensured by providing an automated benchmark generation and analysis workflow design along with a continuously updated Performance Database. Benchmarking requires limited effort, as the micro-benchmarks are automatically generated, and the user only needs to compile and run the generated code on systems of interest. A big insight of the proposed design is that there is ample layer repeatability within and across models. This keeps the number of unique layers and thus the number of Performance Database entries in check over time. For new models, only the newly introduced unique layers are benchmarked.

For example, consider a scenario where all models in Table I except for `ResNet*-v2` have already been benchmarked and the results are in the Performance Database. Using our design, benchmarking the `ResNet*-v2` models requires measuring all the `ResNet*-v2` layers that are not within the Performance Database. Evaluating this hypothetical scenario results in a 75% reduction (30 minutes) in benchmarking time on the `Tesla_V100` system for batch size 32. The saving would be even larger on slower systems. By storing and reusing the micro-benchmark results in the Performance Database we minimize the time cost of running micro-benchmarks.

Benanza is extensible. As shown in Figure 2, *Benanza* is designed as a set of modular components. As new cuDNN functions are introduced, users update the *Benanza* runtime accordingly. For example, if a new cuDNN convolution algorithm is added, then the user can just add it to the list of algorithms to instantiate in the convolution benchmark implementation. If a new cuDNN/cuBLAS API or a fused API is added, then a user needs to add the benchmark implementation for the new API using the templates provided by *Benanza*. Users can also extend the Automatic Benchmark Generator to support other runtimes that target other software libraries or hardware, and leverage most of the other components unmodified. These runtimes can target the frameworks' Python or C++ API or other DL libraries (e.g. MIOpen [41] on AMD GPUs, or MKL-DNN [42] on CPUs). Through the novel benchmarking and analysis design, *Benanza* copes well with the fast evolving pace of DL innovations.

IV. Evaluation

We implemented *Benanza* and evaluated its design by answering **Q1-6** . We evaluated 30 ONNX models (listed in Table I) in the MXNet (v1.5.1), ONNX Runtime (v0.5.0), and PyTorch (v1.3) frameworks. Experiments were run on the 7 systems listed in Table III. All systems use Ubuntu 18.04.3 LTS, CUDA 10.1.243, cuDNN Version 7.6.3, and CUDA Driver 430.26. The micro-benchmarks were compiled with GCC 7.4.0. We first computed the `float32` "lower-bound" latency in both sequential and parallel modes. Then we used the Analyzer to uncover and explore optimization opportunities — cuDNN heuristics, framework inefficiencies, layer fusion, and usage of Tensor Cores, and show their impact on the latency.

A. "Lower-Bound" Latency vs. Measured Latency

We measured the inference latency of the 30 models using MXNet, ONNX Runtime, and PyTorch on the `Tesla_V100` system. Figure 6 shows the measured latency across all models and Figure 7 compares the latencies using different frameworks.

TABLE III

WE USED 7 GPU SYSTEMS FOR EVALUATION. THE SYSTEMS COVER THE PAST GPU GENERATIONS (FROM KEPLER TO THE LATEST TURING). AMAZON CLOUD (AWS) IS USED FOR 4 OF THE SYSTEMS AND THE OTHER 3 ARE LOCAL MACHINES. THE 4 TURING AND VOLTA GPUS SUPPORT TENSOR CORES AND THEIR THEORETICAL TENSOR CORE PERFORMANCE (TENSOR TFLOPS) ARE LISTED.

Name	CPU	GPU (Release Year)	GPU Architecture	GPU Memory Capacity, Bandwidth	Theoretical FP32 TFLOPS	Theoretical Tensor TFLOPS
Tesla_K80 (AWS P2)	Intel Xeon CPU E5-2686 v4	Tesla K80 (2014)	Kepler	12 GB, 480 GB/s	5.6	✗
Tesla_M60 (AWS G3)	Intel Core i9-7900X CPU	Tesla M60 (2015)	Maxwell	7 GB, 160.4 GB/s	4.8	✗
TITAN_ Xp	Intel Xeon CPU E5-2686 v4	TITAN Xp (2017)	Pascal	12 GB, 547.6 GB/s	12.2	✗
TITAN_V	Intel Core i7-7820X CPU	TITAN V (2017)	Volta	12 GB, 672 GB/s	14.9	110.0
Tesla_V100 (AWS P3)	Intel Xeon CPU E5-2686 v4	Tesla V100 SXM2 (2018)	Volta	16 GB, 900 GB/s	15.7	125.0
Quadro_RTX	Intel Xeon CPU E5-2630 v4	Quadro RTX 6000 (2019)	Turing	24 GB, 624 GB/s	16.3	130.5
Tesla_T4 (AWS G4)	Intel Xeon Platinum 8259CL CPU	Tesla T4 (2019)	Turing	15 GB, 320 GB/s	8.1	65.0

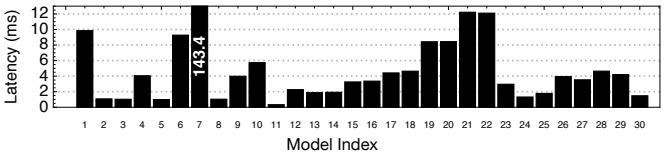

Fig. 6. The measured latency of all ONNX models using batch size 1 with MXNet backend on Tesla_V100 in Table III.

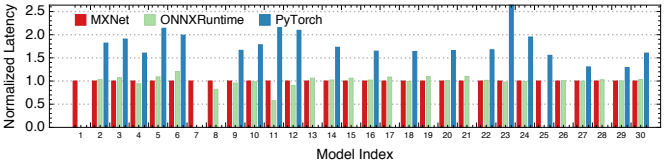

Fig. 7. The measured latency of all ONNX models with MXNet, ONNX Runtime, and PyTorch backends (normalized to MXNet latency) using batch size 1 on Tesla_V100.

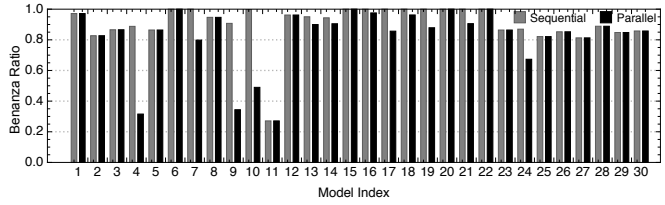

Fig. 8. The *Benanza* Ratio in sequential and parallel mode of 30 models in MXNet using batch size 1 on `Tesla_V100`.

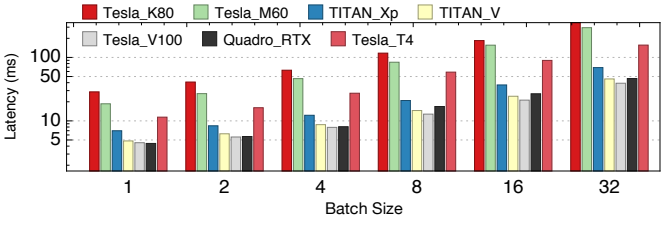

Fig. 9. The measured latency of `ResNet50_v1` in MXNet across batch sizes and systems.

Due to the lack of support of some ONNX operators by ONNX Runtime [43] and PyTorch [44], not all models run within these frameworks. As MXNet is the fastest in general, subsequent sections of the paper (with the exception of Section IV-C) focus on informing optimizations in MXNet.

1) Q1,2 *Sequential Mode vs Parallel Mode:* The difference between the "lower-bound" latency and the measured latency indicates the optimization opportunities in the framework and its use of the cuDNN and cuBLAS APIs. A model's "lower-bound" latency normalized to its measured latency is referred to as its *Benanza Ratio* (BR). Figure 8 shows the BR in sequential ($BR_{sequential}$) and parallel mode ($BR_{parallel}$) in MXNet across all models using batch size 1 on the `Tesla_V100` system.

The $\text{BR}_{\text{sequential}}$ across models has a geometric mean of 0.88, thus a potential latency speedup of $\frac{1.0}{0.88} = 1.14\times$ can be made to the measured model execution. The $\text{BR}_{\text{parallel}}$ across models has a geometric mean of 0.76, indicating a potential latency speedup of $\frac{1.0}{0.76} = 1.32\times$. The difference between a model's parallel and sequential "lower-bound" latency depends on the existence of parallel modules within the model and how compute-intensive the data-independent paths are. Models without parallel modules have the same sequential and parallel "lower-bound" latency, thus the $\text{BR}_{\text{sequential}}$ is equal to the $\text{BR}_{\text{parallel}}$. For models with compute-intensive parallel modules, such as the Inception models (ID=$4, 9, 10$), the potential speedup of the latency (or $\frac{1}{\text{BR}_{\text{parallel}}}$) is $2.87\times$, $2.69\times$, and $2.45\times$ respectively. The $\text{BR}_{\text{sequential}}$ and $\text{BR}_{\text{parallel}}$ of `LeNet` (ID=11) are both low because `LeNet` is a simple model which has low latency ($0.33ms$ as shown in Figure 6) and the MXNet overhead and other non-compute portion is high, thus its BR is low.

The sequential "lower-bound" latency of the models with parallel modules (e.g. `Inception` and `ResNet` models) is closer to their measured latency when compared to the parallel "lower-bound" latency ($\text{BR}_{\text{parallel}} < \text{BR}_{\text{sequential}} < 1$). This suggests that parallel modules are executed sequentially in MXNet, even though the data-independent layers could be run in parallel. We verified the sequential execution behavior in MXNet by inspecting the model execution profile. Thus we evaluated the benefits of the latter optimizations in terms of the sequential "lower-bound" latency.

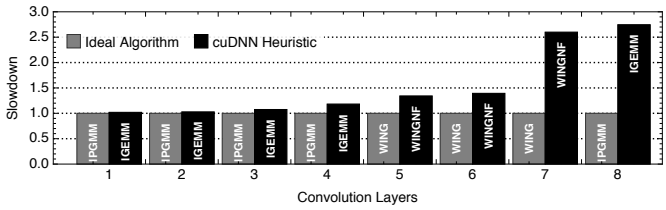

Fig. 10. The cuDNN heuristic selects 8 non-optimal convolution layer algorithms for `ResNet50_v1` using batch size 32 on `Tesla_V100`. Up to 2.75× speedup can be achieved if selection was ideal.

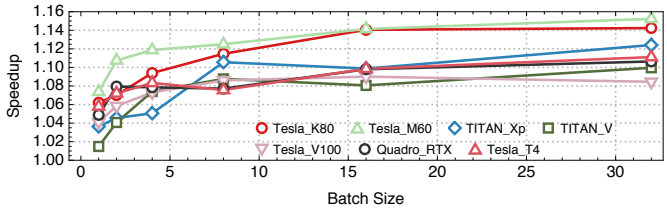

Fig. 11. The latency speedup achieved for `ResNet50_v1` by applying the MXNet optimization described in Section IV-C1 across batch sizes and systems.

2) Batch Sizes and Systems: To demonstrate *Benanza*'s functions across batch sizes and systems, we evaluated the "lower-bound" latency of all models using different batch sizes from 1 to 32 on representative systems (shown in Table III). We select batch size 32, since some models cannot be run using batch sizes beyond 32 due to GPU memory limitations. Figure 9 shows the measured latency of `ResNet50-v1` on all systems in log scale. As expected, latencies are reversely correlated to the compute capability of the system (e.g. theoretical FP32 TFLOPS in Table III). `ResNet50-v1` has a higher latency on `Quadro_RTX` when compared to `Tesla_V100`, since `Quadro_RTX` has an on-chip (global) memory bandwidth of 624 GB/s whereas `Tesla_V100` has an on-chip memory bandwidth of 900 GB/s.

Figure 12 shows the $\text{BR}_{\text{sequential}}$ of `ResNet50-v1` across batch sizes and systems. The results suggest that `ResNet50-v1`'s optimization opportunities are system and batch size dependent. Both `Tesla_V100` and `TITAN_V` are highly optimized to run `ResNet50-v1` across batch sizes, since their BR is high — ranging from 0.86 to 1.0. The BR for `Tesla_T4` and `Quaro_RTX` is high for batch sizes 1 to 4 but drops beyond that. `ResNet50-v1` is less optimized on the other systems and has a low BR.

The geometric mean of the $\text{BR}_{\text{sequential}}$ for all the models across systems and batch sizes is shown in Figure 13. Both `Tesla_V100` and `TITAN_V` still have a high BR ($0.76 - 0.88$). A drop was still observed for `Tesla_T4` and `Quaro_RTX` at batch size 4. `Tesla_M60` and `TITAN_Xp` have a BR between 0.63 and 0.72. The oldest GPU generation, `Tesla_K80`, has the lowest BR and is the least optimized.

Overall, the current software stack (latest MXNet, cuDNN, and CUDA libraries used in the evaluation) is more optimized for the recent GPU generations (Turing and Volta) using smaller batch sizes. Compared to Volta, the software stack is less optimized for Turing. This is possibly because Turing is newly released, and we expect optimizations that target Turing to increase. Moreover, the low BR for the older GPUs suggest that vendors prioritize optimizations for newer GPU generations over older ones.

B. Q3 *cuDNN Convolution Heuristics*

Using the *Benanza* Analyzer, we observed that heuristics employed by cuDNN (and subsequently the frameworks) are not always optimal. For example, Figure 10 shows the convolution layer latencies using the algorithms informed by cuDNN heuristics (labeled as *cuDNN Heuristic*) normalized to using the optimal algorithm (labeled as *Ideal Algorithm*) for `ResNet50_v1` using batch size 32 on `Tesla_V100`. The algorithm choices are listed in Section III-B2. Figure 14 shows the latency speedup for `ResNet50_v1` across batch sizes and systems by using the optimal convolution algorithm for all convolution layers. Figure 15 shows the geometric mean of the latency speedup for all models by using the optimal algorithms. At batch size 32, the speedup ranges between 1.14× and 1.32× across GPUs. Both the latest and older GPU architectures can benefit from better algorithm heuristics.

C. Q4 *Inefficiencies in Frameworks*

We used *Benanza* to identify the inefficiencies in MXNet and PyTorch. We then implemented the optimizations informed by *Benanza* and show the latency speedup after the framework modifications.

1) MXNet ONNX Model Loader: We observed through the Analyzer that there are layers in the model execution profile where the cuDNN API arguments deviate from what is expected. An inspection of the Analyzer's parsed Nsight profile pointed to an `image_2d_pad_constant_kernel` GPU kernel function being invoked before every convolutional layer. Non-zero padding leads to the observed deviation between the expected and actual cuDNN API calls. We inspected the MXNet source code and found that padding layers are inserted during the loading of ONNX models in MXNet. ONNX supports specifying asymmetric padding as a parameter in convolution layers, whereas MXNet does not. Therefore, MXNet must insert padding layers before convolution layers where asymmetric padding is used when loading ONNX models. However, the MXNet ONNX model loader adds padding layers before every convolution layer (regardless of the use of asymmetric padding). A non-intrusive optimization is to only insert padding layers if asymmetric padding is used. With this simple one-line optimization, we observed up to 1.15× latency speedup for `ResNet50-v1` (shown in Figure 11).

2) PyTorch cuDNN Wrapper: Using *Benanza* we observed that there were excessive calls to `cudaStreamWaitEvent` between cuDNN API calls. Using the backtrace information from the model execution profile, we identified the PyTorch source file that introduces these synchronizations. Upon further study of the source code, we found that all cuDNN functions are invoked by a cuDNN wrapper in PyTorch. The wrapper manages a pool of cuDNN handles and is designed to enable invoking cuDNN functions from different CPU threads. cuDNN

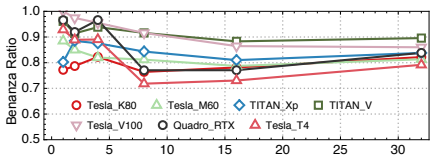

Fig. 12. The $BR_{sequential}$ of `ResNet50-v1`.

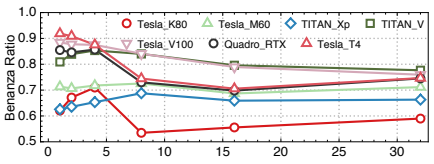

Fig. 13. The geometric mean of the $BR_{sequential}$ of all models.

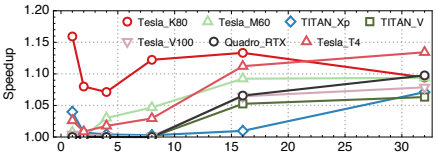

Fig. 14. The latency speedup for `ResNet50-v1` if the cuDNN heuristic selections were optimal.

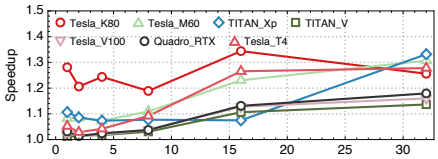

Fig. 15. The geometric mean of the latency speedup for all models by using the optimal convolution algorithm.

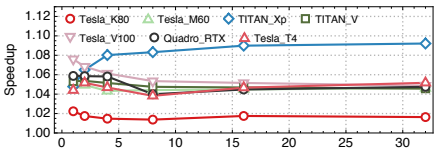

Fig. 16. The latency speedup for `ResNet50-v1` if layer fusion was performed.

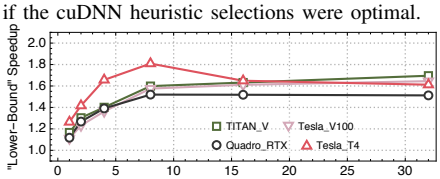

Fig. 17. The "lower-bound" latency speedup if Tensor Cores (`NCHW`) were used for `ResNet50-v1`.

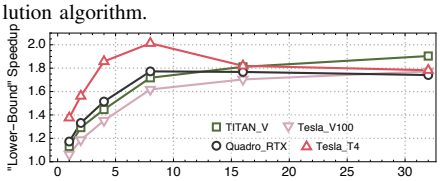

Fig. 18. The "lower-bound" latency speedup for `ResNet50-v1` if Tensor Cores (`NHWC`) were used.

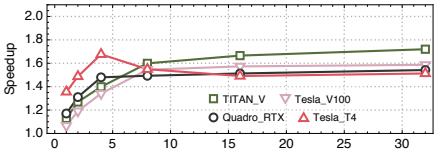

Fig. 19. The latency speedup for `ResNet50-v1` if Tensor Cores (`NHWC`) were used.

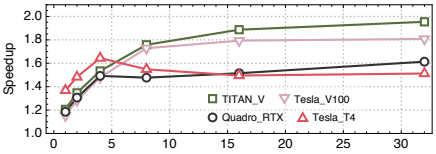

Fig. 20. The latency speedup for `ResNet50-v1` if parallel execution, optimal algorithm selections, layer fusion, and Tensor Cores (`NHWC`) were used.

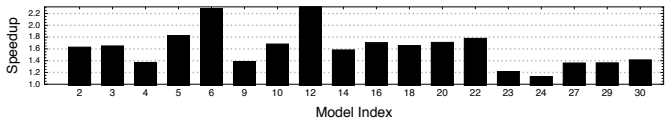

Fig. 21. The speedup achieved by removing unnecessary cuDNN API synchronizations in PyTorch on `Tesla_V100` using batch size 1.

functions managed by the same handle are synchronized and executed sequentially. In the current PyTorch (v1.3), however, a single handle is used for inference, and thus forced synchronization occurs before each cuDNN function call. The synchronizations cause $100\mu s$ stalls on average between cuDNN functions, thus the latency saved through this optimization is a function of the number of layers in a model. We modified PyTorch to elide the cuDNN wrapper and only synchronize before and after performing inference. Figure 21 shows the speedup achieved by this optimization for batch size 1. `MobileNet-v2` (ID=12) achieves a 2.3× speedup, since it has low latency and a large number of layers.

D. Q5 *Layer Fusion*

We used *Benanza* to evaluate the potential benefits of layer fusion. Figure 16 shows the latency speedup from layer fusion for `ResNet50-v1` across the systems. `ResNet50-v1` has the layer sequence pattern Conv→Bias→BatchNorm→Activation. *Benanza* reports that the Conv→Bias sequence can be fused for better latency and performs the fusion analysis (Section III-D4). In all, 64 (18%) layers were fused and up to 1.09× speedup was achieved over the measured latency across systems for `ResNet150-v1`. By inspecting the model execution profile, we found no indication that MXNet, ONNX Runtime, or PyTorch perform layer fusion using the cuDNN fused API.

E. Q6 *Tensor Cores*

We used *Benanza* to evaluate the potential benefits of using `float16` and Tensor Cores available on recent GPU architectures. While the cuDNN Tensor Core API supports both `NHWC` and `NCHW` layout, NVIDIA recommends the use of `NHWC`. We use *Benanza* to generate benchmarks targeting both the `NHWC` and `NCHW` layout and evaluated the "lower-bound" latency speedup, as shown in Figures 18 and 17 respectively. As expected, using the `NHWC` achieves higher speedup. Internally, the current cuDNN API implements `NCHW` convolutions in terms of `NHWC` with an implicit transposition. As compute dominates (i.e. larger batch sizes), the relative overhead of the transposition becomes small; hence, `NCHW` and `NHWC` have similar performance for larger batch sizes. Figure 19 shows the latency speedup by using Tensor Cores(`NHWC`). `TITAN_V` achieves significant speedup (up to 1.72×). We can see that `Tesla_T4` benefits most from Tensor Cores for smaller batch sizes (i.e. might be best used for low-latency inference).

F. Q1,2,3,5,6 *Parallel Execution, Algorithm Selection, Layer Fusion, and Tensor Cores*

Benanza can be used to perform the above analysis jointly. To demonstrate this, we analyzed the latency speedup when using parallel execution of data-independent layers, optimal algorithm selection, layer fusion, and Tensor Cores (`NHWC`). Figure 20 shows the latency speedup for `ResNet50-v1` across batch sizes and systems. Up to a 1.95× and 1.8× speedup can be achieved by `TITAN_V` and `Tesla_V100` respectively. We

can surmise, from the previous analysis, that most of the profit for `TITAN_V` is attributed to its use of Tensor Cores. `Quadro_RTX` and `Telsa_T4` achieve marginal speedup over the Tensor Core results.

V. Related Work

DL Benchmarking: There has been no shortage of work on developing benchmarks to characterize DL models. These DL benchmarks either take a model as a black-box and measure the user-observable latency and throughput (end-to-end benchmarks) or delve deeper into models to characterize the layer or kernel performance (micro-benchmarks). The end-to-end benchmarks [3], [4], [6] provide a corpus of models that are deemed to be of value to characterize for industry and research. Micro-benchmarks [5], [45], [46], [4] distill DL models into their layers or kernels, and are hand-curated. Micro-benchmarking enables easy measurements of layers within popular DL models and integrates easily with profiling tools. In [47], the author present a design that enables benchmarking DL models at across the abstraction levels of inference pipeline and introduce a hierarchical profiling methodology (enabling framework-, model-, and hardware-profiling). In [7], the authors propose a benchmark suite to enable fair comparison of DL techniques at different levels of granularity. At the operator level, [7] takes ONNX models and generates micro-benchmarks that target the framework's Python API to measure the latency of each operator. *Benanza* also takes ONNX models as input, but generates lower-level cuDNN and cuBLAS micro-benchmarks to compute the "lower-bound" latency of the model, and perform analysis. The authors are unaware of previous work which generates micro-benchmarks from model layers and couples it with an analysis workflow to inform optimizations.

Performance Advising: There is past work on using profiling to inform users of possible optimizations. These optimizations are performed at the compiler level [48] or are plugins to code editors to inform proper usage of APIs[49], [50]. Low-level profile reports and some suggestions on how to address bottlenecks are provided by profilers and IDEs such as: NVIDIA's Nvprof [8], Intel's VTune [10], Oracle's Solaris Studio [51], Microsoft's Roslyn [52], and IBM's XL [53]. To the author's knowledge, there has been no work on applying or specializing the optimization advising to the DL domain.

VI. Conclusion

This paper presents *Benanza*, a sustainable and extensible DL benchmarking and analysis design that automatically generates layer-wise benchmarks for DL models to compute the "lower-bound" latency and inform optimizations on GPUs. We use *Benanza* to evaluate a set of 30 models using different frameworks on 7 GPUs, and pinpointed the optimizations in parallel layer execution, cuDNN algorithm selection, framework inefficiency, layer fusion, and Tensor Core usage. The results show that *Benanza* fills a significant gap within the characterization/optimization cycle and would boost the productivity of DL model, framework, and library developers.

Acknowledgments

This work is supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a member of the IBM Cognitive Horizon Network, and the Applications Driving Architectures (ADA) Research Center - one of the JUMP Centers co-sponsored by SRC and DARPA.

References

[1] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018. [Online]. Available: https://doi.org/10.1109/mm.2018.112130030

[2] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE. IEEE, Feb. 2018, pp. 620–629. [Online]. Available: https://doi.org/10.1109/hpca.2018.00059

[3] "MLPerf," github.com/mlperf, 2019, accessed: 2019-10-04.

[4] "AI-Matrix," github.com/alibaba/ai-matrix, 2019, accessed: 2019-10-04.

[5] Baidu, "DeepBench," github.com/baidu-research/DeepBench, 2019.

[6] C. Coleman, M. Zaharia, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, and C. Ré, "Analysis of DAWNBench, a time-to-accuracy machine learning performance benchmark," *SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, pp. 14–25, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3352020.3352024

[7] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2019, the 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19). [Online]. Available: https://doi.org/10.1109/ipdps.2019.00018

[8] "NVIDIA nvprof," docs.nvidia.com/cuda/profiler-users-guide/index.html, accessed: 2019-5-04.

[9] "NVIDIA Nsight System," developer.nvidia.com/nsight-systems, accessed: 2019-5-04.

[10] "Intel VTune," software.intel.com/en-us/vtune, accessed: 2019-5-04.

[11] "NVIDIA cuDNN," developer.nvidia.com/cudnn, 2019, accessed: 2019-10-04.

[12] "NVIDIA cuBLAS," developer.nvidia.com/cublas, accessed: 2019-10-04. [Online]. Available: developer.nvidia.com/cublas

[13] "ONNX: Open Neural Network Exchange," onnx.ai, 2019, accessed: 2019-10-04.

[14] "Neural Network Exchange Format (NNEF)," www.khronos.org/nnef, 2019, accessed: 2019-10-04.

[15] "ONNX Model Zoo," github.com/onnx/models, 2019, accessed: 2019-10-04.

[16] J. Deng, J. Guo, and S. Zafeiriou, "ArcFace: Additive angular margin loss for deep face recognition," *CoRR*, vol. abs/1801.07698, 2018. [Online]. Available: arxiv.org/abs/1801.07698

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2015, pp. 1–9. [Online]. Available: https://doi.org/10.1109/cvpr.2015.7298594

[19] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013. [Online]. Available: arxiv.org/abs/1311.2524

[20] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *CoRR*, vol. abs/1608.06993, 2016. [Online]. Available: arxiv.org/abs/1608.06993

[21] P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, and G. W. Cottrell, "Understanding convolution for semantic segmentation," *CoRR*, vol. abs/1702.08502, 2017. [Online]. Available: arxiv.org/abs/1702.08502

[22] E. Barsoum, C. Zhang, C. Canton-Ferrer, and Z. Zhang, "Training deep networks for facial expression recognition with crowd-sourced label distribution," *CoRR*, vol. abs/1608.01041, 2016. [Online]. Available: arxiv.org/abs/1608.01041

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: arxiv.org/abs/1502.03167

[24] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: arxiv.org/abs/1512.00567

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [Online]. Available: https://doi.org/10.1109/5.726791

[26] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: arxiv.org/abs/1704.04861

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: arxiv.org/abs/1512.03385

[28] ——, "Identity mappings in deep residual networks," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 630–645.

[29] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, vol. abs/1707.01083, 2017. [Online]. Available: arxiv.org/abs/1707.01083

[30] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: arxiv.org/abs/1602.07360

[31] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: arxiv.org/abs/1612.08242

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: arxiv.org/abs/1409.1556

[33] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: arxiv.org/abs/1311.2901

[34] Onnx, "ONNX shape inference," https://github.com/onnx/onnx/blob/master/docs/ShapeInference.md, 2019.

[35] Google, "Google benchmark," github.com/google/benchmark, 2014.

[36] A. Anderson and D. Gregg, "Optimal DNN primitive selection with partitioned boolean quadratic programming," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*, ACM. ACM Press, 2018, pp. 340–351. [Online]. Available: https://doi.org/10.1145/3179541.3168805

[37] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning," *CSUR*, vol. 52, no. 4, pp. 1–43, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3320060

[38] "The CUDA Profiling Tools Interface," developer.nvidia.com/cuda-profiling-tools-interface, 2019, accessed: 2019-10-04.

[39] "NVIDIA GPU Metrics Reference," docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference, accessed: 2019-7-24.

[40] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.

[41] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "MIOpen: An open source library for deep learning primitives," 2019.

[42] "Mkl-Dnn," github.com/intel/mkl-dnn, 2019, accessed: 2019-10-04.

[43] Microsoft, "ONNX runtime," github.com/microsoft/onnxruntime, 2019.

[44] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," vol. 6, 2017.

[45] S. Chintala, "ConvNet Benchmarks," github.com/soumith/convnet-benchmarks, 2019.

[46] Intel, "benchdnn," github.com/intel/mkl-dnn/tree/master/tests/benchdnn, 2019.

[47] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-M. Hwu, "XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs." IEEE, May 2020, the 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS'20).

[48] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *CSUR*, vol. 51, no. 5, pp. 1–42, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3197978

[49] H. Vandierendonck, S. Rul, and K. De Bosschere, "The paralax infrastructure," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, IEEE. ACM Press, 2010, pp. 389–399. [Online]. Available: https://doi.org/10.1145/1854273.1854322

[50] A. Haj-Ali, N. K. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: End-to-end vectorization with deep reinforcement learning," *arXiv preprint arXiv:1909.13639*, 2019.

[51] O. Solaris, "Oracle solaris studio code analyzer," 2019.

[52] K. Ng, M. Warren, P. Golde, and A. Hejlsberg, "The Roslyn project, exposing the c# and VB compiler's code analysis," *White paper, Microsoft*, 2011.

[53] V. Sarkar, "Automatic selection of high-order transformations in the IBM XL FORTRAN compilers," *IBM J. Res. & Dev.*, vol. 41, no. 3, pp. 233–264, May 1997. [Online]. Available: https://doi.org/10.1147/rd.413.0233