A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms

Yu Emma Wang [1] **Gu-Yeon Wei** [1] **David Brooks** [1]

Abstract

Training deep learning models is compute-intensive and there is an industry-wide trend towards hardware and software specialization to improve performance. To systematically compare deep learning systems, we introduce a methodology comprised of a set of analysis techniques and parameterized end-to-end models for fully connected, convolutional, and recurrent neural networks. This methodology can be applied to analyze various hardware and software systems, and is intended to complement traditional methods. We demonstrate its utility by comparing two generations of specialized platforms (Google's Cloud TPU v2/v3), three heterogeneous platforms (Google TPU, Nvidia GPU, and Intel CPU), and specialized software stacks (TensorFlow and CUDA).

1 Introduction

With the end of Moore's law, academic and industrial research efforts have shifted from general-purpose processors to domain specific architectures (DSAs) (Hennessy & Patterson, 2011). Deep learning, which has revolutionized many application domains (Silver et al., 2017; Huang et al., 2017; Amodei et al., 2016; Wu et al., 2016), is a promising field for DSAs (Dean et al., 2018). New customized training hardware, software stacks, and optimization tools are being developed to support ever more sophisticated deep learning models. Thus there is a great need to concurrently develop a systematic and scientific methodology for comprehensive performance analysis of hardware and software systems customized for deep learning.

The rapid evolution of deep learning models and corresponding hardware and software platforms requires new analysis techniques that go beyond simply running today's well-known deep learning models on individual platforms. A systematic methodology must expose interactions between hardware and software platforms across the spectrum of model attributes (e.g., hyperparameters), so that the resulting insights can be applied to future models. The methodology itself needs a fast development cycle to rapidly target new platforms, and it should include large enough models to stress the limits of emerging platforms.

Recent analysis efforts have been limited to relatively small collections of seemingly arbitrary DNN models (Mattson et al., 2019; Adolf et al., 2016; Chen et al., 2012; Tao et al., 2018). The development of such suites is very time-consuming. It took half a year to release MLPerf v0.6, and months to add a new model. Even so, the shelf life of such models is seldom more than a couple of years. Moreover, using a collection of individual models such as ResNet-50 (He et al., 2016) and Transformer (Vaswani et al., 2017) can lead to misleading conclusions. For example, Transformer is a large FC model that trains 3.5× faster on the Tensor Processing Unit (TPU) than on a GPU, yet focusing on this single model would not reveal the severe TPU memory bandwidth bottleneck that arises with FCs larger than 4k nodes.

We propose a comprehensive performance evaluation methodology that combines parameterized deep learning benchmarks with systematic analysis techniques. We introduce *ParaDnn*, a tool that generates thousands of parameterized multi-layer models, including fully-connected models, convolutional neural networks, and recurrent neural networks, with model parameter sizes that vary by almost five orders of magnitude, far beyond the range of existing benchmarks. Systematic analysis techniques then learn the sensitivity of performance to model hyperparameters and explore various dimensions of the design space. We show that this parameterized analysis methodology *complements* the use of real-world workloads (e.g., MLPerf), leading to insights that traditional approaches either cannot expose or cannot fully explain.

We conduct case studies in three diverse performance evaluation scenarios: homogeneous platforms, heterogeneous platforms, and software stacks. We hope to motivate researchers to apply our methodology to other platforms. In Section 4, we analyze and compare two generations of homogeneous specialized platforms, TPU v2 and v3. Our methodology provides insights for designing and upgrading ML accelerators in production-scale systems. In Section 5, we perform cross comparison of three architectures (CPU, GPU, and TPU) that span the continuum between general purpose processors and specialized accelerators, and the methodology reveals individual strengths and weaknesses of each platform. In Section 6, we explore the performance evolution of specialized software stacks, TensorFlow and CUDA. Table 1 summarizes fourteen observations and insights as examples enabled by our methodology.

While our analysis methodology is able to reveal optimization opportunities in current system designs, optimization

[1]John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA. Correspondence to: Yu Emma Wang <ywang03@g.harvard.edu>.

Proceedings of the 3rd MLSys Conference, Austin, TX, USA, 2020.
Copyright 2020 by the author(s).

Observation	Proof	Insight/Explanation
1. TPU exploits the parallelism from batch size and the model width.	Fig 2	To design/upgrade new specialized systems, architects need to consider interactions between the operation mix from key workloads (arithmetic intensity) and system configurations (FLOPS, memory bandwidth/capacity, intra-chip/host-device interconnect). TPU serves as a great example.
2. Many operations are bottlenecked by TPU memory bandwidth.	Fig 3	
3. TPU suffers from large inter-chip communication overhead.	Fig 4	
4. Smaller CNN models are more bottlenecked by CPU hosts.	Fig 5	
5. TPU v3 speeds up compute-bound MatMuls by 2.3×, memory-bound ones by 3×, and large embeddings by $>3\times$.	Fig 6	
6. The largest FC models prefer CPU due to memory constraints.	Fig 7	Need for model parallelism on GPU and TPU.
7. Models with large batch size prefer TPU. Those with small batch size prefer GPU.	Fig 8 Fig 10	Large batches pack well on systolic arrays; warp scheduling is flexible for small batches.
8. Smaller FC models prefer TPU and larger FC models prefer GPU.	Fig 8	FC needs more memory bandwidth per core (GPU).
9. TPU speedup over GPU increases with larger CNNs.	Fig 10	TPU architecture is highly optimized for large CNNs.
10. TPU achieves up to 3× FLOPS utilization compared to GPU.	Fig 11	TPU is optimized for both CNN and RNN models.
11. GPU performance scales better with RNN embedding size than TPU.	Fig 10	GPU is more flexible to parallelize non-MatMuls.
12. Within seven months, the software stack specialized for TPU was improved by up to 2.5× (CNN), 7× (FC), and 9.7× (RNN).	Fig 12	It is easier to optimize for certain models than to benefit all models at once.
13. Quantization from 32 bits to 16 bits significantly improves TPU and GPU performance.	Fig 12	Smaller data types save memory traffic and enable larger batch sizes, resulting in super-linear speedups.
14. TensorFlow and CUDA teams provide substantial performance improvements in each update.	Fig 12	There is huge potential to optimize compilers even after the hardware has been shipped.

Table 1: A summary of major observations and insights enabled by our analysis methodology.

details are beyond the scope of this paper. Our analysis focuses on training, not inference. We do not study accuracy or the performance of multi-GPU/TPU systems. Such studies might yield different conclusions, and we leave these extensions to future work, as each deserves in-depth study. Section 7 discusses the limitations in detail.

2 METHODOLOGY

Current deep learning (DL) performance analysis methods have limitations in terms of the insights they are able to reveal. They often leverage two distinct types of benchmark suites: real-world suites such as MLPerf (Mattson et al., 2019), Fathom (Adolf et al., 2016), BenchNN (Chen et al., 2012), and BenchIP (Tao et al., 2018), and microbenchmark suites, such as DeepBench (Research, 2017) and BenchIP. Each real-world suite contains a handful of popular DL models spanning a variety of model architectures. Such suites have a long development cycle. Their shelf-life is unknown since they only contain today's deep learning models, which may become obsolete as DL models evolve rapidly. Further, they fail to reveal deep insights into interactions between DL model attributes and hardware performance, since the benchmarks are sparse points in the vast space of deep learning models. Micro-benchmark suites exercise basic operations (e.g., matrix multiplication or convolution) in neural networks, but they cannot simulate complex dependencies between different operations in end-to-end models.

To *complement* existing performance analysis methods, we introduce a systematic methodology, composed of a tool, ParaDnn, and a set of analysis methods. ParaDnn has the advantages of the above approaches, with the goal of providing large "end-to-end" models covering current and *future* applications, and parameterizing the models to explore a much larger design space of DNN model attributes. Our methodology can stress the upper and lower bounds of hardware and software systems in various dimensions, including floating-point computation capability, memory bandwidth, inter-chip bandwidth, and host-device balance. For cross-platform comparisons, the methodology can also discover cases favoring one platform over another and describe the DL hyperparameters of such cases. With ParaDnn, such studies can be conducted comprehensively, quickly, and conveniently. The utility of parameterized analysis is not

Variable	Layer	Nodes	*Input*	*Output*	Batch Size
Min	4	32	2000	200	64
Max	128	8192	8000	1000	16384
Inc	×2	×2	+2000	+200	×2

(a) Fully Connected Models

Variable	Block	Filter	*Image*	*Output*	Batch Size
Min	1	16	200	500	64
Max	8	6	300	1500	1024
Inc	+1	×2	+50	+500	×2

(b) Conv. Neural Nets: Residual and Bottleneck Blocks

Variable	Layer	Embed	*Length*	*Vocab*	Batch Size
Min	1	100	10	2	16
Max	13	900	90	1024	1024
Inc	+4	+400	+40	×4	×4

(c) Recurrent Neural Networks: RNN, LSTM, GRU

Table 2: The ranges of the hyperparameters and dataset variables (*italic*) chosen in this paper.

limited to those cases, and one goal of this paper is to motivate application of our methodology to new platforms.

2.1 ParaDnn

We first introduce ParaDnn, a tool that generates parameterized end-to-end models to run on target platforms. ParaDnn creates models encompassing fully-connected models (FC), convolutional neural networks (CNN), and recurrent neural networks (RNN). The models are parameterizable, so ParaDnn models are equal to or greater in size compared to today's real-world models. For example, a single end-to-end CNN model from ParaDnn contains a mixture of many different layers with different sizes of convolution, batch normalization, pooling, and FC layers. The complexity of ParaDnn workloads is comparable to that of real-world models (e.g., ResNet-50 and Transformer), as will be shown in Figure 1. Insights about hardware performance sensitivity to model attributes allow interpolating and extrapolating to future models of interest. These insights could not be discovered with either the small point space exploration of the real-world suites or microbenchmarks, which do not capture inter-operation dependencies as ParaDnn does. The model types of ParaDnn cover 95% of Google's TPU workloads (Jouppi et al., 2017), all of Facebook's deep learning models (Hazelwood et al., 2018; Gupta et al., 2019; Naumov et al., 2019), and eight out of nine MLPerf models (Mattson et al., 2019), with the exception of *minigo*, the reinforcement learning model.

Fully-Connected Models FC models comprise multiple fully-connected layers. The architecture is

$$\text{Input} \rightarrow [\text{Layer}[\text{Node}]] \rightarrow \text{Output},$$

where [Layer] means the number of layers is variable. We can sweep the number of layers, the number of nodes per layer, and the numbers input and output units of the datasets.

Convolutional Neural Networks CNN models are residual networks. The architecture of ParaDnn CNNs is

$$\text{Input} \rightarrow [\text{Residual/Bottleneck Block}] \times 4 \rightarrow \text{FC} \rightarrow \text{Output}.$$

A residual network contains four groups of blocks (He et al., 2016). Each can be a residual block or a bottleneck block, followed by a fully-connected layer. Residual blocks have two convolutional layers and two batch normalization layers, while bottleneck blocks have three of each. ParaDnn treats the minimum number of filters as a variable, and it doubles in every group. An input image is square with three channels, represented by its length.

Recurrent Neural Networks RNNs are comprised of multiple layers of basic RNN, LSTM, or GRU cells:

$$\text{Input} \rightarrow [\text{RNN/LSTM/GRU Cell}] \rightarrow \text{Output}.$$

Each token of the input sequence is embedded within a fixed length vector, and the length of the vector is the embedding size. We sweep the number of layers and the embedding size. The variables in the dataset include the maximum length per input sequence and vocabulary size.

Range of Hyperparameters and Datasets We choose the range of hyperparameters and datasets to cover the real models (Section 2.2), and make sure the design space is tractable. Table 2 summarizes how hyperparameters are swept. We focus on large batches, and extremely small batches may lead to different conclusions. By default, this paper uses CNNs with bottleneck blocks and basic RNNs.

2.2 Real-World Models

In addition to ParaDnn, we study six real-world models. We show that ParaDnn and those models are complementary—ParaDnn explores a larger design space, and real models represent several currently popular design points.

This work focuses on TensorFlow, the native framework for TPU. We include two of the three workloads in TensorFlow from MLPerf (Mattson et al., 2019), i.e., Transformer (Vaswani et al., 2017) and ResNet-50 (He et al., 2016). We also select other real-world workloads (Repository, 2018), including RetinaNet (Lin et al., 2017a), DenseNet (Huang et al., 2017), MobileNet (Howard et al., 2017), and SqueezeNet (Iandola et al., 2016). We refer to them as real workloads/models.

Figure 1 shows the numbers of trainable parameters across all workloads to quantify the sizes of the models. The ParaDnn workloads are shown as ranges and the real workloads as dots. ParaDnn covers a large range of models, from 10k to nearly a billion parameters. Transformer is the largest real FC, and RetinaNet is the largest real CNN. The small models, SqueezeNet and MobileNet, are typical of models targeting mobile applications.

2.3 Analysis Methods

ParaDnn enables a set of analysis methods that can quantify, compare, and visualize the DL design space in various dimensions. We apply those methods after running all ParaDnn workloads on platforms under study to collect performance metrics of interest. All analysis methods and results distinguish ParaDnn from suites of individual models, because the real-world suites do not support sensitivity analysis, and ParaDnn covers a much larger design space.

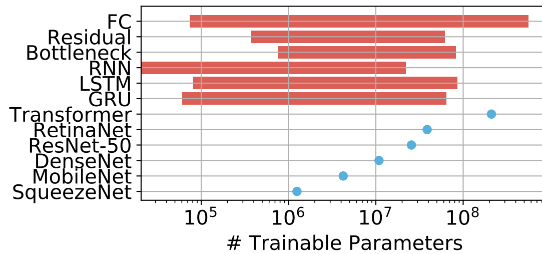


Figure 1: The numbers of trainable parameters for all models.

Heat Map With ParaDnn, we can measure performance sensitivity to hyperparameters. Heat maps are an intuitive approach to understand the design space of DL models. A heat map uses colors to show how a performance metric of interest responds to model hyperparameters (on *x*- and *y*-axes). The rate of color change across the map reflects the sensitivity of performance to hyperparameters.

Quantification with Linear Regression Table 2 shows five hyperparameters of each model type under study, and a heat map can only visualize two hyperparameters. Also, observing sensitivity via heat maps is more qualitative than quantitative. We propose to use linear regression (LR) to quantify the sensitivity. We train a LR model using hyperparameters to predict performance, and use the weights of the hyperparameters as a measure of sensitivity. Other metrics including T- and F-test may be used for this purpose (Hogg et al., 2005), but they only report positive values of importance. LR reports the signs of the weights, indicating positive or negative correlations. Note that this LR model is not for prediction. Section 4.1 presents a detailed example.

Roofline Model Roofline models are useful to study memory and computation bottlenecks (Williams et al., 2009; Jouppi et al., 2017). A roofline represents the upper bound of floating-point operations per second (FLOPS) for workloads with different compute intensity. Roofline model analysis shows that ParaDnn's models range from extremely bandwidth-bound to compute-bound. Such a range is hard to achieve with existing real models, especially to reach the limits of TPUs. Section 4.2 presents details.

In addition to roofline models, we study the design space in other dimensions as well, by visualizing ParaDnn and real-world models on scatter plots with various *x*- and *y*-axes. Section 4.4 studies FLOPS and data infeed time. Section 5 studies model size and speedup.

Box Plots We use box plots to summarize the performance of each ParaDnn model type. Box plots show that the performance of a ParaDnn model type spans a large range, and they highlight the risk of overly optimizing hardware and software systems for certain models.

3 Hardware Platforms

Our selection of hardware reflects the latest configurations widely available in cloud platforms at paper submission time. Platform specifications are summarized in Table 3.

CPU Platform The CPU is an n1-standard-32 instance from Google Cloud Platform with Skylake architecture. It has 16

Platform	*Unit*	*Version*	*Mem (GB)*	*Mem Bdw (GB/s)*	Peak *FLOPS*
CPU	1 VM	Skylake	120	16.6	2T SP[†]
GPU	1 Pkg	V100	16	900	125T
TPU	1 Board	v2	64	2400	180T
TPUv3	(8 cores)	v3	128	3600[*]	420T

[†] 2FMA × 32Single-Precision × 16Cores × 2GHz = 2 SP TFLOPS
[*] Estimated based on empirical results (Section 4.5).

Table 3: Hardware platforms under study.

cores and 32 threads. It has large memory (120 GB) and lowest peak flops (2 TFLOPS) among the three. GeekBench 4 produced the bandwidth measurement.

GPU Platform The GPU is an NVIDIA V100 in a DGX-1 GPU platform that contains 8 V100 packages (SXM2) connected via 300 GB/s NVlink 2.0 interconnect. We currently measure the performance of a single SXM2 node. One node has 16 GB of memory and 900 GB/s memory bandwidth. A V100 has 640 tensor cores and is able to run mixed precision training using float16 to compute and float32 to accumulate, making its peak performance 125 TFLOPS.

TPU Platform We use Cloud TPU v2 instances to which we were given academic access in February 2018. Each TPU board contains four TPU packages (the default configuration) (Dean, 2017). One package contains 2 cores and one core has one matrix unit (MXU). A Cloud TPU v2 platform supports 180 TFLOPS at peak. Memory size is 8 GB per core, or 64 GB per board, with 2400 GB/s overall memory bandwidth. TPU v2 supports mixed precision training using bfloat16 and float32. TPU v3 has twice the number of MXUs and twice the HBM capacity per core of v2 (Google, 2018). Its memory bandwidth has not been disclosed, but empirical results show that it has increased by 1.5×. TPU v3 has a peak of 420 TFLOPS, 2.3× greater than v2.

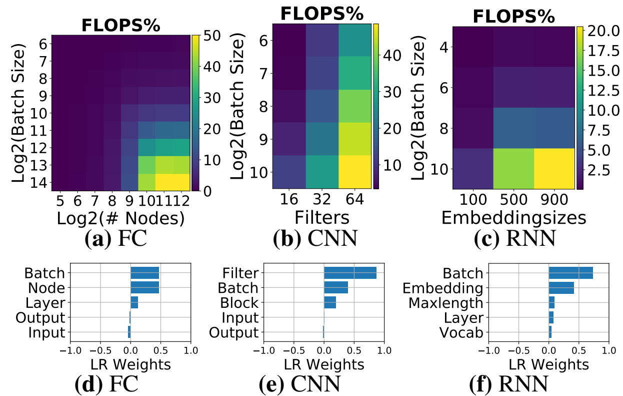
This is the first research paper to study TPU v2/v3, which supports training, while TPU v1 only runs inference (Jouppi et al., 2017). To enable training, TPU v2 supports more operations than matrix multiplication such as gradient and various optimizer operations. It also carries more pressure on the memory system, since weights are accessed a second time in the backward pass. Also, TPU v2 has scalar/vector units, which do not exist in v1. TPU v2 has MXUs of size 128×128 with 32- or 16-bit data types; v1 has 256×256 and 8 bits.

Understanding TPU Memory Size The TPU implements data parallelism by splitting each batch of training data evenly among the 8 cores. Every TPU core keeps a whole copy of the model. Therefore memory size per core determines the maximum model supported (Sec 5.1), while total memory determines the maximum batch size (Sec 5.2).

Comparison Rationale One V100 package and one TPU board (4 packages) are the minimal units available. On Cloud TPU, distribution of computation across its four packages happens automatically, while multi-GPU performance depends largely on user's implementation. Conclusions here do not apply to systems with multiple GPUs or TPU boards (Chao & Saeta, 2019).

4 TPU PERFORMANCE IMPLICATIONS

As the end of Dennard scaling and Moore's law has slowed the performance improvement of general-purpose microprocessors (Dean et al., 2018), the design of DSAs is becoming more and more relevant. The TPU is a prominent example (Jouppi et al., 2017; Dean, 2017). Its development was motivated by the observation that with conventional CPUs, Google would have had to double their datacenter footprint to meet the internal demand for DL workloads. Google has been using TPUs for their large-scale production systems, including Search, Translate, and Gmail. Analyzing the architecture of such systems can provide valuable insights into future deep learning accelerator design.



(a) FC **(b)** CNN **(c)** RNN
(d) FC **(e)** CNN **(f)** RNN

Figure 2: (a)–(c) ParaDnn's FLOPS utilization and (d)–(f) its sensitivity to hyperparameters.

In this section, we use our methodology to study the performance characteristics of TPU v2 and v3 (Dean, 2017; Google, 2018), with a focus on v2, from the computation capability of the core to system balance. We show that ParaDnn can reveal system bottlenecks in a more comprehensive way than real-world models by probing upper and lower system limits. Based on such observations, we discuss possible steps to improve TPU performance, which can be generalized to other deep learning accelerator systems. Observations 1 to 5 in Table 1 summarize our key findings.

4.1 FLOPS Utilization

We use our methodology to study the TPU's floating-point operations per second (FLOPS) utilization, which is the ratio of workload average FLOPS to platform peak FLOPS, measuring how efficiently the computation capacity of a platform is used. We measure the FLOPS of ParaDnn models sweeping hyperparameters listed in Table 2. To visualize FLOPS, we use heat maps.

Heat Maps Figures 2(a)–(c) present heat maps of FLOPS utilization for FC, CNN, and RNN ParaDnn models. For each model type, we choose two hyperparameters (as described below) that affect FLOPS utilization the most, sweeping their ranges to create a map grid while keeping other hyperparameters fixed. FLOPS utilization of all three model types increases with batch size, indicating that the TPU is capable of leveraging the parallelism within a batch. FLOPS utilization of FCs also increases with node count per FC layer; that of CNNs also increases with filter count; and that of RNNs, with embedding size. So the TPU also exploits parallelism within the widths of the models.

Quantifying with Linear Regression To quantify these effects, we use the weights of a linear regression (LR) model. For FC, the LR model is

$$\begin{aligned}\text{FLOPS} = \; & w_0 \times \text{layer} + w_1 \times \text{node} + \\ & w_2 \times \text{input} + w_3 \times \text{output} + w_4 \times \text{batch size},\end{aligned}$$

where w_0–w_4 are hyperparameter weights. When training the LR model, we normalize weights to the same scale, so

that weight reflects importance. For example, a positive w_1 value shows that node count affects performance positively. Figures 2(d)–(f) show the LR weights of the model hyperparameters. Batch size and model width have the highest absolute weights, shown on the x- and y-axes in Figures 2(a)–(c). Figure 2(d) shows that the FLOPS utilization of FCs is largely affected by batch size and node count, while layer count, and output and input unit counts do not matter as much. Similarly, Figure 2(e) shows that filter count and batch size are most important for CNNs. For RNNs, utilization is most affected by batch and embedding sizes.

Takeaways ParaDnn enables systematic study of hyperparameter sensitivity and shows that it is natural for a ML system to utilize parallelism arising from large batch size and model width. It is especially intuitive to map batch size and model width to the two dimensions of systolic arrays. Meanwhile, parallelism opportunities opened by large numbers of layers remain to be explored via model parallelism (Dean et al., 2012; Jia et al., 2018) and pipelining (Blog, 2019).

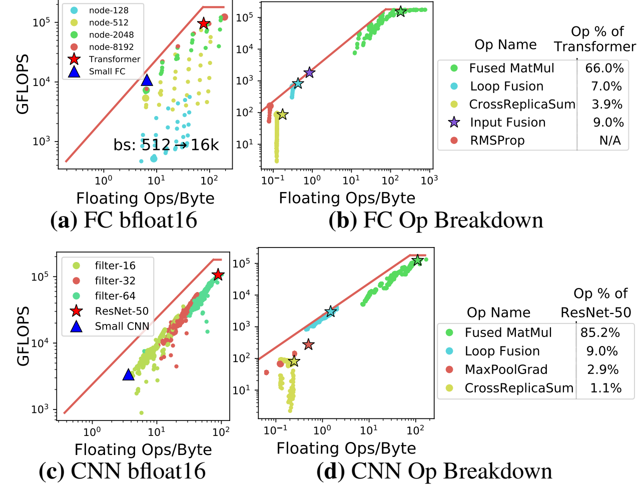
4.2 Roofline Model Analysis

The computation capacity of the TPU's core is only one source of its performance. Memory bandwidth also has a significant impact. In this section, we apply the roofline model (Williams et al., 2009) to ParaDnn FCs and CNNs to analyze the TPU's computation and memory bandwidth. We omit RNN models because the TPU profiler reports incorrect numbers for RNN memory bandwidth.

The Roofline Model Figure 3 shows the roofline plots. The y-axis is FLOPS and the x-axis is arithmetic intensity, i.e., floating-point operations per byte transferred from memory. The roofline (the red line in Figure 3) has of a slanted part and a horizontal part. It represents the highest achievable FLOPS at a given arithmetic intensity. Any data point (x, y) on the slanted part has $\frac{x}{y}$ = memory bandwidth. The horizontal part is the hardware peak FLOPS. A workload or operation (a point in Figure 3) close to the slanted roofline is memory-bound; one close to the horizontal part is compute-bound. A workload or operation not close to the roofline stresses neither memory interconnect nor compute units. Figures 3(a) and 3(c) show all the ParaDnn FCs and CNNs (dots) plus Transformer and ResNet-50 (stars). Figures 3(b) and 3(d) show all the operation breakdowns. The triangles in Figures 3(a) and 3(c) are selected memory-bound models.

The design space shown with roofline models indicates that ParaDnn is a superset of the real-world models. ParaDnn models span a much larger range in the design space, from extremely memory-bound to compute-bound. Therefore performance analysis with ParaDnn can comprehensively test the limits of platforms in both extremes. An exception is that some operations of Transformer do not align closely with those of FCs. This results from a choice in this paper: ParaDnn uses the RMSProp optimizer, keeping nodes per layer uniform for FCs, while Transformer uses the *adafactor* optimizer and has layers with 4k, 2k, and 512 nodes.

ParaDnn Analysis We first discuss the insights enabled by ParaDnn, of which the real-world models are a subset. Figure 3(a) shows that large batch sizes make FCs more compute-bound, and more nodes make FCs more memory-bound. That is because FCs with more nodes need to transfer more weights/activations from the memory, and



(a) FC bfloat16 **(b)** FC Op Breakdown

(c) CNN bfloat16 **(d)** CNN Op Breakdown

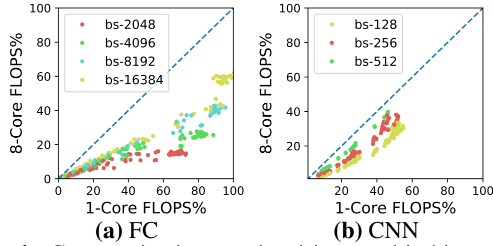
Figure 3: TPU rooflines for FCs and CNNs. (a) and (c): ParaDnn and real-world models. (b) and (d): their operation breakdown.

large batch sizes increase the computation per weight/activation transferred, i.e, the arithmetic intensity. Specifically, FCs with $\geq$ 2k nodes per layer and $\geq$ 8k batch size are compute-bound. Transformer is close to compute-bound and it uses 4k batch size, so it overlaps with FCs having 4k batch sizes. Figure 3(c) shows that models close to ResNet-50 are compute-bound, while a majority of the CNNs are bottlenecked by memory bandwidth. The CNNs' higher FLOPS comes from higher arithmetic intensity caused by more filters. When memory bandwidth is the bottleneck, the way to increase FLOPS is to increase arithmetic intensity.

Figures 3(b) and 3(d) show the TensorFlow operations that take more than 1% of the workload execution time and more than 0 FLOPS. The arithmetic intensity of such operations can be as low as 0.125.[1] The TensorFlow breakdown in Figure 3 is generated after operation fusion, which combines and executes several operations together for higher efficiency. In Figures 3(b) and 3(d), the only compute-bound operation is large fused MatMul (MatMul fused with other operations), so a compute-bound model needs large MatMuls. Other operations are closer to the slanted line, constrained by memory bandwidth. Transformer and ResNet-50 are compute-bound (Figures 3(a) and 3(c)) because they have compute-bound MatMuls (Figures 3(b) and 3(d)).

Real-World Model Analysis ParaDnn and real-world models are complementary. By analyzing ParaDnn, we explore the design space and reach the limits of platforms. Analyzing real-world models puts the design space study into realistic context by highlighting popular representative designs. The tables in Figure 3 show the operation breakdown of Transformer and ResNet-50, and indicate that even compute-bound models contain a noticeable fraction of memory-bound operations. Transformer has three memory-bound operations: (1) input fusion (9.0%), which includes multiply, subtract, and reduce; (2) loop fusion (7.0%), which consists of control flow operations (e.g., select and equal-to); and (3) CrossReplicaSum (3.9%), which sums up the values across multiple weight replicas. These

[1] An activation accumulation operation (CrossReplicaSum in TensorFlow) uses float32 even with bfloat16 model weights. In this case, the arithmetic intensity is $1/(2 \times 4 \text{ bytes}) = 0.125$, i.e., one floating-point addition for every two data points loaded.



(a) FC **(b)** CNN

Figure 4: Communication overhead in a multi-chip system is non-negligible, but is reduced with large batch sizes.

three operations contribute 19.9% of the total execution time. (12.3% of the execution time is for data formatting, which has no arithmetic intensity or TPU FLOPS.) ResNet-50 has memory-bound loop fusion (9%), MaxPoolGrad (2.9%), and CrossReplicaSum (1.1%), which sums to 13%, showing the need for end-to-end optimization for DL accelerators.

Takeaways ParaDnn explores the design space and stresses platform limits; real-world models represent the currently important design points. ParaDnn shows that the design space is composed of very diverse models, from extremely memory-bound to compute-bound; real-world models show that even compute-bound models contain non-negligible fractions of memory-bound operations (19.9% for Transformer and 13% for ResNet-50), which suggests that memory bandwidth can affect other ML systems originally designed to optimize computation. Researchers can test system memory-boundness with ParaDnn. Approaches for speeding up memory-bound operations include caching (Hennessy & Patterson, 2011), operation fusion (xla, 2018; Chen et al., 2018; Rotem et al., 2018), aggressive data quantization (Banner et al., 2018), and compression (Han et al., 2015; Lin et al., 2017b).

4.3 Multi-Chip Overhead

Computing speed and memory bandwidth of a TPU core are not the only factors affecting training performance, because typical large-scale systems use multiple chips (Dean et al., 2012). This section evaluates the scalability of a multi-chip TPU system with ParaDnn. We quantify the multi-chip overhead by comparing the FLOPS utilization of 1-core (x-axis) and 8-core TPUs (y-axis) in Figure 4. If there were no multi-chip overhead, FLOPS utilization of 1-core and 8-core should be the same, i.e., all points should lie on the dashed line in Figure 4 showing $x = y$.

ParaDnn allows us to explore models with a wide range of communication overhead. Figure 4 shows that an 8-core TPU exhibits noticeably lower FLOPS utilization than a 1-core TPU, reflecting significant inter-core communication overhead. For FC, the maximum FLOPS utilization in an 8-core TPU is 62%, compared to 100% in a 1-core TPU. Multi-chip overhead is less noticeable in CNNs, with FLOPS utilization decreasing from 55% in the 1-core TPU to 40% in the 8-core. It is worse for FCs because there are more weights to synchronize across cores than for CNNs.

Our analysis method indicates that large workloads can amortize the parallelism overhead, and it highlights batch size as the key hyperparameter that affects communication overhead. Increasing batch size reduces the FLOPS utilization gap by increasing computation without increasing weight synchronization. On the 8-core TPU, FCs need at least 16k batch size to achieve more than 50% FLOPS utilization. Specifically, FCs with ≥ 256 nodes and ≤ 512 batch size run faster on a TPU with one core than on one with eight. Thus we consider FCs with larger than 1024 batch size in Figure 4. Based on Amdahl's law, the maximum non-parallel fraction of the workloads is up to 60% for FC and up to 40% for CNN. Using the largest batch size shown in Figure 4, the 90th-percentile of non-parallel fractions are 16% for FC and 8.8% for CNN.

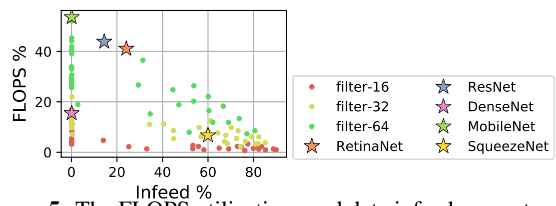


Figure 5: The FLOPS utilizations and data infeed percentages of ParaDnn (dots) and real-world (stars) CNNs.

Takeaways With diverse ParaDnn models, we observe that communication overhead in multi-chip systems is non-negligible even for large FCs and CNNs. Using large batch size can reduce overhead by increasing parallel computation without increasing weight transfers. Possible optimizations include relaxed synchronization, model parallelism (Dean et al., 2012), gradient compression (Lin et al., 2017b), and weight pruning and compression (Han et al., 2015).

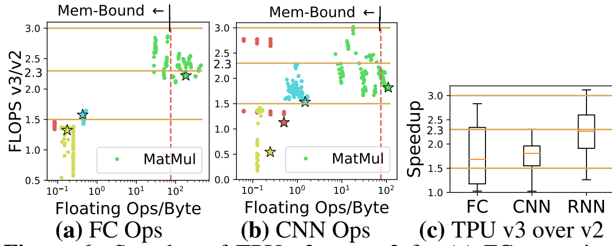
4.4 Host-Device Balance

Previous subsections have focused on the performance of the accelerator itself. We now turn to "data infeed," the process of preparing and moving input data to the TPU board. ParaDnn in other sections uses data synthesized from CPU hosts, which avoids most of the data infeed overhead. Here we use ParaDnn CNN models with the ImageNet dataset (Krizhevsky et al., 2012).

The TPU system includes a CPU host and a TPU device (Google, 2018). For image datasets, the host fetches images from the network, decodes and preprocesses them, and feeds them to the device. We refer this as data preparation. The device then performs training computation on the images. Data infeed includes network overhead, host compute, and transfer between host and device.

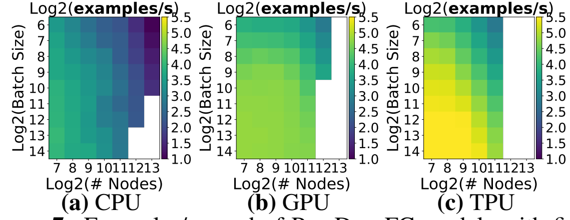
For each ParaDnn CNN and the ImageNet dataset, we use the TPU profiler to collect FLOPS utilization and infeed time percentage, which is the fraction of time the accelerator spends waiting for data. Figure 5 shows the results as dots, along with the real-world CNNs as stars. ParaDnn models are very diverse, ranging from 0 to 50% FLOPS utilization and 0 to 90% infeed time. ParaDnn shows that many CNNs have significant infeed time and that larger CNNs tend to have lower infeed time. Large CNNs, those with more filters and/or more layers, are the most suitable for the TPU system, because the accelerator spends more time training each image and CPU infeed time per image is fixed. The high-performance TPU system targets large workloads. Consistent with the communication overhead study in Section 4.3, small workloads do not have enough parallelism to utilize the TPU efficiently.

Some real models show opportunities to optimize for data-infeed bottlenecks. SqueezeNet has the highest infeed time because it is designed to accommodate mobile devices by using small numbers of filters per layer. While MobileNet



(a) FC Ops **(b)** CNN Ops **(c)** TPU v3 over v2

Figure 6: Speedup of TPU v3 over v2 for (a) FC operations, (b) CNN operations, and (c) ParaDnn models. The red line (75 ops/byte) is the inflection point in the TPU v2 roofline (Fig 3).



(a) CPU **(b)** GPU **(c)** TPU

Figure 7: Examples/second of ParaDnn FC models with fixed depth (64). Larger memory allows the CPU to run larger models.

also targets mobile devices, it has one convolution layer that uses up to 1k filters, eliminating time lost to data infeed. RetinaNet, ResNet, and SqueezeNet show potential for improvement through system optimizations. The performance without data preparation shows that resolving the infeed bottleneck can lead to 37%, 34%, and 180% performance improvement, respectively.

Takeaways ParaDnn shows the design space of FLOPS and data-infeed time, and reveals that large workloads with abundant parallelism are not host-bound on large accelerator systems. Real-world models show that the performance of some workloads can be improved. When designing an accelerator system, scaling performance of the CPU host to match the accelerator is crucial for utilization of the accelerator's computation resource.

4.5 TPU v3

In this section, we systematically quantify the differences between TPU v2 and v3. Figure 6 compares the two using ParaDnn (dots), plus ResNet and Transformer (stars). Batch size for v3 is twice that for v2, thanks to its doubled memory capacity. Figures 6(a) and 6(b) use a variation of the roofline model, showing arithmetic intensity on the x-axis and operation speedup on the y-axis. Data point colors representing operation types are consistent with those in Figures 3(b) and 3(d). As a reference, the red dashed line is the inflection point in the TPU v2 roofline from Figure 3, where arithmetic intensity is 75 ops/byte (180 TFLOPS / 2.4 TB/s). The operations on the left of the red line are memory-bound; those on the right are compute-bound. We group the operations in four classes, as follows.

Compute-Bound Ops The peak FLOPS of TPU v3 is 2.3× that of v2, so the performance of compute-bound operations is improved by about 2.3× on v3. Such operations are on the right of the red dashed line in Figure 6(b).

Memory-Bound Ops (2× batch size) The maximum speedup of the memory-bound operations (mainly the MatMuls in Figures 6(a) and 6(b)) is 3×. Tripled speedup comes from doubled batch size (owing to doubled memory capacity) and memory bandwidth improvement. The memory bandwidth increase of v3 over v2 has not been officially disclosed, but we can estimate it. Doubled batch size means doubled arithmetic intensity. On the slanted line of a roofline model, that means doubled FLOPS, because the ratio of FLOPS to arithmetic intensity is fixed. Switching from v2's roofline to v3's thus increases FLOPS by twice the bandwidth improvement. So the 3× overall speedup suggests that v3 bandwidth improvement over v2 is $3/2 = 1.5\times$, to 3.6 TB/s.

Other Memory-Bound Ops The 1.5× bandwidth improvement estimate is corroborated by the 1.5× speedup of other memory-bound operations, represented by the non-MatMul FC operations in the lower left corner of Figure 6(a). The performance of those operations does not increase with larger batch size, as shown by the vertical alignment of each operation type in Figure 3(b). Thus the 1.5× speedup in Figure 6(a) is from bandwidth improvement.

Boundary Cases The compute-bound MatMuls in Figure 6(b) become memory-bound on TPU v3, so the speedup is $< 2.3\times$. Such operations have arithmetic intensity between 75 and 117, because the roofline inflection point of v3 is at $x = 420/(2.4\times 1.5) = 117$. CrossReplicaSum (yellow dots) is slowed down on TPU v3, which may be because of more replicas across more MXUs.

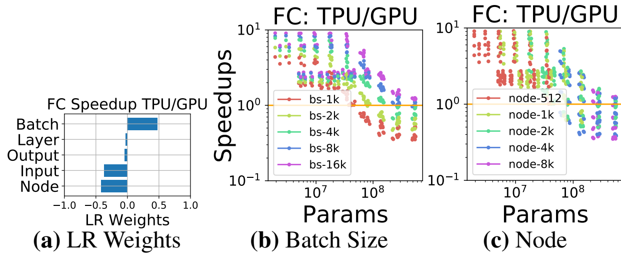
End-to-End Models In Figure 6(c) the maximum speedups are 2.83× (FC), 2.31×(CNN), and 3.11×(RNN). Speedup increases with model width (second column of Table 2), and the maximum speedup is achieved by the largest width. FCs with close to 3× speedup are dominated by memory-bound MatMuls. Exceptions are RNNs with more than 3×; these have the largest embedding size (900), indicating that TPU v3 optimizes large embedding computations.

Takeaways ParaDnn allows examining new platforms with a wider range of workloads, from memory-bound to compute-bound, than using real-world models alone. Comparing TPU v3 to v2 as an example, ParaDnn can show the system upgrade benefiting operations with different arithmetic intensity. TPU v3 shows three main levels of speedup: 2.3× for compute-bound operations, 3× for memory-bound MatMuls, and 1.5× for other memory-bound operations. This is the result of its 2.3× FLOPS, 2× memory capacity, and 1.5× memory bandwidth.

5 CROSS-PLATFORM COMPARISON

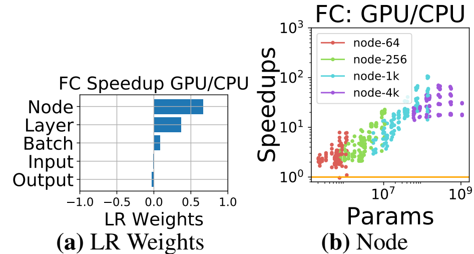
In this section, we show ParaDnn's utility in cross-platform comparison, with CPU, GPU, and TPU as exemplars along the continuum between general purpose processors and specialized accelerators. ParaDnn shows the sensitivity of speedup to model hyperparameters, allowing users to choose platforms based on model hyperparameters of interest, rather than on model characterizations that happen to have been reported. ParaDnn also reveals the fundamental architectural differences between platforms and shows the trade-offs between flexibility and specialization.

- The **TPU** is highly-optimized for large batches and CNNs, and has the highest training throughput.
- The **GPU** is more flexible and programmable for irregular computations, such as small batches and non-MatMul operations. Training of large FC models benefits from its sophisticated memory system and higher bandwidth.
- The **CPU** is the most programmable, so it achieves the highest FLOPS utilization for RNNs, and it supports the largest model because of its high memory capacity.



(a) LR Weights **(b)** Batch Size **(c)** Node

Figure 8: (a) Sensitivity analysis of TPU over GPU speedups. Speedups color-coded by (b) batch size and (c) FC nodes per layer.



(a) LR Weights **(b)** Node

Figure 9: (a) The sensitivity analysis of (b) GPU over CPU speedups for FCs.

5.1 Fully-Connected DNNs

Examples/second measures the number of examples trained per second, a proxy for end-to-end performance. Heat maps in Figure 7 compare ParaDnn FCs for three platforms, with varying node counts and batch sizes but fixed layer count (64). We use LR weights from Section 4.1 to quantify the hyperparameter effects (not shown owing to space limitations). Layer and node counts have negative weights because it is more time-consuming to train larger models with many layers and nodes. Batch size greatly improves throughput on the GPU and TPU, but not the CPU because the parallelism available with small batch sizes can fully utilize a CPU.

In Figure 7, the white squares indicate models that encounter out-of-memory issues. Only the CPU supports the largest models, and the GPU supports larger models than the TPU. This is because every hardware core keeps one copy of the model, so memory per core determines the largest model supported, as explained in Section 3. The CPU has the highest memory per core (120 GB), and the GPU (16 GB) is higher than the TPU (8 GB). While TPUs and GPUs may draw more attention, as of today the only choice for extremely large models is the CPU, which supports all model sizes. For example, Facebook uses dual-socket CPU servers with large memories to train ranking models (FC networks) (Hazelwood et al., 2018). That fact highlights the need for model parallelism and pipelining on the GPU and TPU (Dean et al., 2012; Jia et al., 2018; Blog, 2019) to allow those powerful platforms to support larger models.

TPU over GPU To further investigate the best hardware platform for FC models, we analyze TPU over GPU speedups. Figure 8(a) plots the linear regression weights across FC hyperparameters for TPU over GPU speedup. Figures 8(b) and 8(c) show the design space for FCs as scatter plots, with numbers of model parameters on the *x* axis and speedups on the *y* axis. To display the effects of hyperparameters, we color-code data points to reflect batch size (Figure 8(b)) and node count per layer (Figure 8(c)). Overall, 62% of the FCs perform better on the TPU.

The TPU is well suited for large batch training because systolic arrays excel at increasing throughput (Kung, 1982). The positive weight of batch size in Figure 8(a) and the horizontal color bands in Figure 8(b) indicate that large batch size is the key to higher speedup. This suggests that the TPU MXUs, implemented with systolic arrays, need large batches to reach full utilization. The GPU is a better choice for small batches, because it executes computation in warps, packing small batches and scheduling them on stream multiprocessors more easily (Nickolls & Dally, 2010).

The GPU is a better choice for large models, suggesting that it is optimized for large FC memory reuse and streaming requirements. This is shown by the negative weights of node count, layer count, and input size in Figure 8(a) and the trend in Figure 8(c), corroborated by the overall negatively-correlated trend of speedup with parameter count in Figure 8. FCs have minimal weight reuse and large models have more weights, so they put a lot of pressure on the memory system. The GPU has a more mature memory system and higher memory bandwidth than the TPU, which makes it better suited to the memory requirements of large FCs.

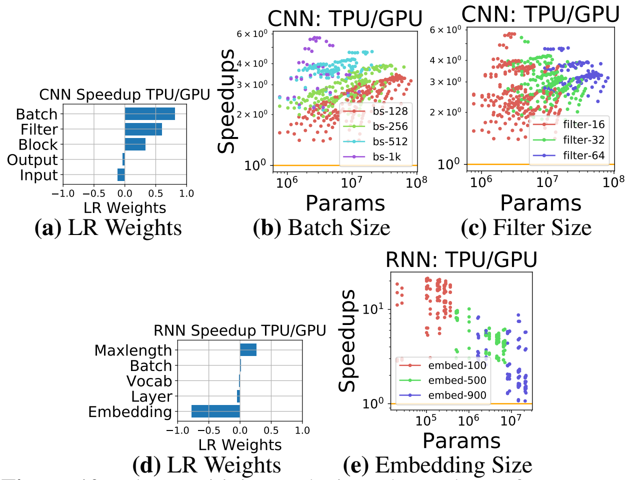
GPU over CPU Figure 9(a) shows the LR weights of GPU-over-CPU speedup. Figure 9(b) shows the design space color-coded by node count. The GPU is a better platform for large FCs because its architecture can better exploit the parallelism available with large batches and models. Recall from Figure 8 that large FCs prefer the GPU over the TPU. So the GPU is the best platform for large FCs, but models with large batches perform best on the TPU.

5.2 CNN and RNN

We now describe the speedup of ParaDnn CNNs and RNNs. Since our conclusions for CPUs are similar to those in the previous section, we omit them in the interest of brevity.

CNN Figures 10(a)–10(c) show the speedups of the TPU over the GPU. All CNNs perform better on the TPU. Batch size is still the key to better TPU-over-GPU speedup for CNNs, shown by its positive LR weight in Figure 10(a) and the increasing speedup with batch size in Figure 10(b). The TPU is the best platform for large CNNs, suggesting that its architecture is highly optimized for the spatial reuse characteristics of CNNs. This is shown by the positive weights in Figures 10(a) and 10(c), where models with more filters and blocks have higher speedups. It is different from Section 5.1, showing that the TPU is not preferred for large FCs. This suggests that the TPU handles large CNNs better than large FCs, because CNNs reuse weights, but FCs seldom do, which results in greater memory traffic. The GPU is a feasible choice for small CNNs. These conclusions only apply to a single GPU; multi-GPU cases may be different.

RNN Figures 10(d)–10(e) show the speedup of TPU over GPU. We display the embedding size in Figure 10(e) because the magnitude of its weight is the greatest in Figure 10(d). Embedding size has negative weight, and embedding computation is more sparse than matrix multiplication. This suggests that the TPU is less flexible for doing non-MatMul computations than the GPU. The TPU is better at dense computations like MatMuls. Even so, RNNs are still up to 20× faster on the TPU. Optimizing non-MatMul computations is another opportunity for TPU enhancement.



(a) LR Weights **(b)** Batch Size **(c)** Filter Size

(d) LR Weights **(e)** Embedding Size

Figure 10: The sensitivity analysis and speedups of TPU over GPU for (a)–(c) CNNs and (d)–(e) RNNs.

5.3 Overall Comparison

This section summarizes the speedup of TPU over GPU and the FLOPS utilization of all ParaDnn and real models. We use box plots (Figure 11) to summarize each ParaDnn model type because performance has a wide range. The bar in the box shows the median. The upper and lower boundaries of the box are 9th and 91st percentiles. The upper and lower bars outside of the box are 2nd and 98th percentiles. Outliers are shown as dots. We do not show the results of using CPUs to train CNNs because it is extremely time consuming and unlikely to contribute additional insights.

TPU over GPU Speedup Figure 11(top) summarizes the TPU over GPU speedups of all models. Note that the real models use larger batch sizes on TPU than on GPU. The speedup of TPU over GPU depends heavily on the nature of the workload measured. The speedup of parameterized models varies widely, from less than 1× to 10×, while the speedup of real workloads ranges from 3× (DenseNet) to 6.8× (SqueezeNet). ParaDnn represents a more complete view of potential workloads, and each real workload represents the concerns of certain users. Benchmarking platforms with two kinds of workloads offer a more systematic understanding of their behavior than those with only one kind.

To further compare the TPU with the GPU, while relaxing the constraint on the GPU's software stack, we also include speedup relative to the GPU performance of ResNet-50, reported in NVIDIA's Developer Blog (Case, 2018) (annotated as NVIDIA in Figure 11(top)). Note that NVIDIA's version of ResNet-50 uses unreleased libraries, and we were unable to reproduce the results. The speedup using ResNet-50 from Google is 6.2×, compared with 4.2×, which suggests software optimization can significantly impact performance.

FLOPS Utilization Figure 11(bottom) shows the FLOPS utilization of all workloads and platforms. On average, the maximum FLOPS utilization of the TPU is 2.2× that of the GPU for all CNN models, and the ratio is 3× for RNNs. The TPU FLOPS utilization of Transformers is consistent with FCs with 4k batch size, as in Figure 2. For RNNs, the TPU has less than 26% FLOPS utilization and the GPU has less than 9%, while the CPU has up to 46% because of its better programmability. RNNs have more irregular computations than FCs and CNNs, owing to temporal dependency in the cells and variable-length input sequences. Advanced

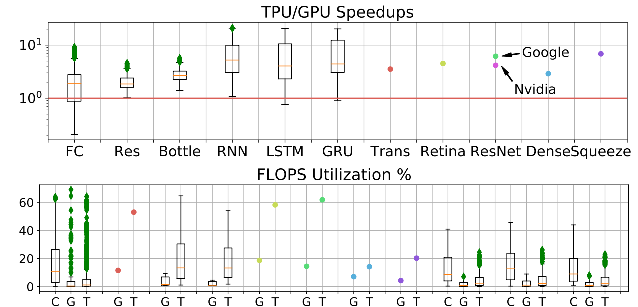


Figure 11: (Top) TPU over GPU speedups of all workloads. (Bottom) FLOPS utilization comparison for all platforms.

RNN optimizations may be able to increase utilization on the GPU and the TPU. Real models with more filters have higher FLOPS, which is why ResNet-50 and RetinaNet have higher FLOPS than DenseNet and SqueezeNet.

6 Software Stack Advances

Custom DL hardware opens opportunities for dramatic software optimizations. ParaDnn is also useful for comparing software performance, by analyzing the performance of different TensorFlow (TF) and CUDA versions. We study data type quantization with software versions, because it depends on software support. Software versions are summarized in the legend of Figure 12. ParaDnn allows more comprehensive analysis of software updates than real models. It can also reveal software optimization focus (e.g., TF 1.9 optimizes small batches); we omit these details for brevity.

6.1 TensorFlow Versions and TPU Performance

The compiler for the TPU is XLA (Leary & Wang, 2017), shipped with TF. Figure 12(top) shows TPU speedups obtained by running TF 1.7 to 1.12, treating 1.7 with float32 as the baseline. Moving from TF 1.7 to 1.12 improves performance for all ParaDnn models. Although FC and CNN encounter performance regression with TF 1.8, TF 1.9 fixes this anomaly and improves overall performance. RNN performance is not improved much until TF 1.11. TF 1.11 shows 10× speedup for RNNs. Transformer, ResNet-50, and RetinaNet are improved continuously over TF updates. Interestingly, SqueezeNet is improved starting from TF 1.11, while the performance of DenseNet and MobileNet see little benefit. In the 7 months (222 days) between the release of TF 1.7.0 (03/29/2018) and that of TF 1.12.0 (11/05/2018), software stack performance was improved significantly. The 90th-percentile speedup of TPU is 7× for FC, 1.5× for Residual CNN, 2.5× for Bottleneck CNN, 9.7× for RNN, and 6.3× for LSTM and GRU.

Bfloat16 enables significant performance improvement for ParaDnn FCs and CNNs. 90th-percentile speedups are up to 1.8× for FC and Bottleneck CNN, and 1.3× for Residual CNN. TPU can support doubled batch sizes with 16 bits. Transmitting fewer bits also relieves bandwidth pressure, speeding up memory-bound operations. Larger performance increases may be possible with further bitwidth reductions.

6.2 CUDA Versions and GPU Performance

Figure 12(bottom) shows GPU performance across versions of CUDA and TF. The baseline is TF 1.7 and CUDA 9.0 with float32. TF 1.8 does not improve GPU performance. By lowering memory traffic and enabling larger batch sizes,

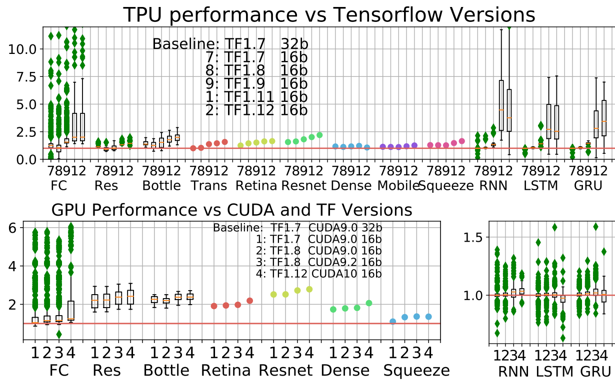


Figure 12: (Top) TPU performance with TensorFlow updates. (Bottom) GPU performance with CUDA and TF updates.

bitwidth reduction can speed up CNNs by more than 2×. We note that CUDA 9.2 speeds up ResNet-50 significantly more (8%) than other real workloads ($< 1\%$), and it speeds up ParaDnn CNNs more than FCs or RNNs. CUDA 10 speeds up other models, but not SqueezeNet. It significantly speeds up ParaDnn FCs and CNNs, but not RNNs. The overall 90th-percentile improvement for FCs, CNNs, and RNNs is up to 5.2×, 2.9×, and 8.6%, respectively. CUDA updates have less impact than do TF updates on the TPU, likely because CUDA and GPU platforms have greatly matured since becoming popular before 2010, while TPU v2 for training was only announced in May 2017.

7 Limitations of this Work

Scope This work does not study DL inference, cloud overhead, multi-node systems, accuracy, convergence, or other DL frameworks. Specifically, NVIDIA's eight-node DGX-1 or Google's 256-TPU systems are not studied. We intentionally leave these topics to future work, as each deserves in-depth study. They need different metrics such as latency, and different setups such as tuning numbers of hardware nodes, inter-node bandwidth, and synchronization mechanisms. Cloud overhead may be more acute and brings up more research questions. TensorFlow is used because it supports all three platforms. Previous work shows that framework implementation can largely affect performance (Wang et al., 2019b), and using PyTorch, which recently starts to support the TPU, may lead to different conclusions.

The validity of extrapolating training throughput to time-to-accuracy remains an open question. Recent work studied the number of training steps to accuracy as a function of the batch size (Shallue et al., 2018). It shows that very large batch sizes result in sub-linear scaling, and the best batch size depends largely on the model and the optimizer. In a multi-node system, synchronization becomes more complicated, which results in different convergence behaviors.

Tractability Readers should take caution when making conclusions using a finite set of workloads and platforms. While this methodology is designed to avoid over-emphasizing one subset of workloads, we have to constrain the diversity of model architectures and hyperparameters to make this work tractable. For example, the CNN models have the ResNet architecture, and RNN models are cells stacked together. In reality, more diverse models with a combination of embedding, FC, CNN and RNN layers are emerging. We also constrain the range of ParaDnn hyperparameters (Table 2). We focus on large batches, as the platforms were designed for large batch training, and smaller batches may lead to different conclusions.

8 Related Work

This paper presents a performance analysis methodology that includes a tool, ParaDnn. The set of models generated by ParaDnn is not designed to replace other benchmark suites, but to complement existing suites to study the design space more comprehensively, as discussed in Section 2. Domain-specific benchmark suites that are complementary with our methodology include MLPerf (Mattson et al., 2019), Fathom (Adolf et al., 2016), CortexSuite (Thomas et al., 2014), and (Hauswald et al., 2015a;b; Coleman et al., 2017; Wang et al., 2019c; Tao et al., 2018). Benchmarks have been the driving force for compiler and architecture design for decades, and notable examples include the SPEC CPU (Henning, 2006) and PARSEC multiprocessor benchmarks (Bienia et al., 2008). In the same spirit as parameterized benchmarks, synthetic benchmarks are common, such as BenchMaker (Joshi et al., 2008), SYMPO (Ganesan et al., 2010), AI Matrix (Wei et al., 2018), and (Kim et al., 2014; Turki et al., 2012; Stroobandt et al., 2000; Schaffter et al., 2011; Saleem et al., 2015). Because of self-similarity of benchmark suites (Wang et al., 2019a), users should carefully select proper benchmarks to work with. Our effort of generating more diverse DL workloads is different from previous work. Our use of DL models to compare up-to-date platforms, Google's TPU v2/v3, and NVIDIA's V100 GPU, distinguishes this work from previous performance comparisons (Shi et al., 2016; Bahrampour et al., 2016; Kothari, 2011; Che et al., 2009; He et al., 2010; Gupta et al., 2019; Wang et al., 2019b).

9 Conclusion

This paper presents a comprehensive performance analysis methodology and its utility in deep learning. We conduct case studies to analyze and compare two generations of specialized platforms (TPU v2/v3), three heterogeneous architectures (TPU, GPU, and CPU), and two specialized software stacks (TensorFlow and CUDA). The methodology is complementary to traditional performance analysis approaches. This paper motivates application of our methodology to other hardware and software systems.

10 Acknowledgement

This work was supported in part by Googles TensorFlow Research Cloud (TFRC) program, NSF Grant CCF1533737, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. The authors would like to thank Frank Chen, Blake Hechtman, Jim Held, Glenn Holloway, Dan Janni, Peter Mattson, Lifeng Nai, David Patterson, Francesco Pontiggia, Parthasarathy Ranganathan, Vijay Reddi, Bjarke Roune, Brennan Saeta, Zak Stone, Sophia Shao, Anitha Vijayakumar, Shibo Wang, Qiumin Xu, Doe Hyun Yoon, Cliff Young for their support and feedback.

References

TensorFlow: Using JIT compilation. 2018. https://www.tensorflow.org/xla/jit.

Adolf, R., Rama, S., Reagen, B., Wei, G.-Y., and Brooks, D. Fathom: Reference workloads for modern deep learning methods. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pp. 1–10. IEEE, 2016.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. Deep speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning*, pp. 173–182, 2016.

Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. Comparative study of Caffe, Neon, Theano, and Torch for deep learning. In *ICLR*, 2016.

Banner, R., Hubara, I., Hoffer, E., and Soudry, D. Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046*, 2018.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81. ACM, 2008.

Blog, G. A. Introducing GPipe, an open source library for efficiently training large-scale neural network models. *https://ai.googleblog.com/2019/03/introducing-gpipe-open-source-library.html*, 2019.

Case, L. Volta Tensor Core GPU achieves new AI performance milestones. *Nvidia Developer Blog*, 2018.

Chao, C. and Saeta, B. Cloud TPU: Codesigning architecture and infrastructure. *Hot Chips*, 2019.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54. Ieee, 2009.

Chen, T., Chen, Y., Duranton, M., Guo, Q., Hashmi, A., Lipasti, M., Nere, A., Qiu, S., Sebag, M., and Temam, O. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 36–45. IEEE, 2012.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

Coleman, C., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. DAWNBench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

Dean, J. Recent advances in artificial intelligence and the implications for computer system design. *Hot Chips*, 2017.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

Dean, J., Patterson, D., and Young, C. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018.

Ganesan, K., Jo, J., Bircher, W. L., Kaseridis, D., Yu, Z., and John, L. K. System-level max power (SYMPO)-a systematic approach for escalating system-level power consumption using synthetic benchmarks. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*, pp. 19–28. IEEE, 2010.

Google. Cloud TPU system architecture. *Google Cloud Documentation*, 2018. https://cloud.google.com/tpu/docs/system-architecture.

Gupta, U., Wang, X., Naumov, M., Wu, C.-J., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Jia, B., Lee, H.-H. S., et al. The architectural implications of Facebook's DNN-based personalized recommendation. *arXiv preprint arXiv:1906.03109*, 2019.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hauswald, J., Kang, Y., Laurenzano, M. A., Chen, Q., Li, C., Mudge, T., Dreslinski, R. G., Mars, J., and Tang, L. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 27–40. IEEE, 2015a.

Hauswald, J., Laurenzano, M. A., Zhang, Y., Li, C., Rovinski, A., Khurana, A., Dreslinski, R. G., Mudge, T., Petrucci, V., Tang, L., et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 50, pp. 223–238. ACM, 2015b.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pp. 620–629. IEEE, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 770–778, 2016.

He, Q., Zhou, S., Kobler, B., Duffy, D., and McGlynn, T. Case study for running HPC applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 395–401. ACM, 2010.

Hennessy, J. L. and Patterson, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

Henning, J. L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

Hogg, R. V., McKean, J., and Craig, A. T. *Introduction to mathematical statistics*. Pearson Education, 2005.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *CVPR*, volume 1, pp. 3, 2017.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

Joshi, A., Eeckhout, L., and John, L. The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop*, pp. 1–11, 2008.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

Kim, K., Lee, C., Jung, J. H., and Ro, W. W. Workload synthesis: Generating benchmark workloads from statistical execution profile. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 120–129. IEEE, 2014.

Kothari, K. Comparison of several cloud computing providers. *Elixir Comp. Sci. & Engg*, 2011.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Kung, H.-T. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

Leary, C. and Wang, T. XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 2017.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. Focal loss for dense object detection. *arXiv preprint arXiv:1708.02002*, 2017a.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017b.

Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., et al. MLPerf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

Nickolls, J. and Dally, W. J. The GPU computing era. *IEEE Micro*, 30(2), 2010.

Repository, T. M. `https://github.com/tensorflow/tpu`. *Github*, 2018.

Research, B. Deepbench `https://github.com/baidu-research/DeepBench`. 2017.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Nadathur, S., Olesen, J., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Saleem, M., Mehmood, Q., and Ngomo, A.-C. N. Feasible: A feature-based SPARQL benchmark generation framework. In *International Semantic Web Conference*, pp. 52–69. Springer, 2015.

Schaffter, T., Marbach, D., and Floreano, D. GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods. *Bioinformatics*, 27(16):2263–2270, 2011.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

Shi, S., Wang, Q., Xu, P., and Chu, X. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pp. 99–104. IEEE, 2016.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

Stroobandt, D., Verplaetse, P., and Van Campenhout, J. Generating synthetic benchmark circuits for evaluating CAD tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(9):1011–1022, 2000.

Tao, J.-H., Du, Z.-D., Guo, Q., Lan, H.-Y., Zhang, L., Zhou, S.-Y., Xu, L.-J., Liu, C., Liu, H.-F., Tang, S., Chen, W., Liu, S.-L., and Chen, Y.-J. BenchIP: Benchmarking intelligence processors. *Journal of Computer Science and Technology*, 33(1):1–23, 2018.

Thomas, S., Gohkale, C., Tanuwidjaja, E., Chong, T., Lau, D., Garcia, S., and Taylor, M. B. CortexSuite: A synthetic brain benchmark suite. In *IISWC*, pp. 76–79, 2014.

Turki, M., Mehrez, H., Marrakchi, Z., and Abid, M. Towards synthetic benchmarks generator for CAD tool evaluation. In *Ph. D. Research in Microelectronics and Electronics (PRIME), 2012 8th Conference on*, pp. 1–4. VDE, 2012.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Wang, Y., Lee, V., Wei, G.-Y., and Brooks, D. Predicting new workload or CPU performance by analyzing public datasets. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–21, 2019a.

Wang, Y. E., Wu, C.-J., Wang, X., Hazelwood, K., and Brooks, D. Exploiting parallelism opportunities with deep learning frameworks. *arXiv preprint arXiv:1908.04705*, 2019b.

Wang, Y. E., Zhu, Y., Ko, G. G., Reagen, B., Wei, G.-Y., and Brooks, D. Demystifying Bayesian inference workloads. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 177–189. IEEE, 2019c.

Wei, W., Xu, L., Jin, L., Zhang, W., and Zhang, T. AI matrix-synthetic benchmarks for DNN. *arXiv preprint arXiv:1812.00886*, 2018.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact provides the source code of ParaDnn, scripts to run ParaDnn and collect data, performance data samples, and a Jupyter Notebook that demonstrate our analysis methods. Reproducing the results in this paper requires the exact versions of software and hardware. Workflows for GPU and TPU require access to specific hardware, NVIDIA V100 GPU and Cloud TPU v2/v3, which may not be widely available to all users.

In this section, we explain the CPU workflow of our artifact, which can be run in any environment from servers to even laptops. The same workflow with small modifications can be used to evaluate other platforms supporting Python and TensorFlow.

A.2 Artifact check-list (meta-information)

- **Program:** ParaDnn
- **Compilation:** Refer to the dependencies for Python 3 and TensorFlow 1.x.
- **Hardware:** CPU, GPU or TPU installed with TensorFlow 1.6-1.13.
- **Execution:** Bash and Python scripts.
- **Metrics:** Training throughput as examples per second.
- **Output:** Performance datasets and analysis figures.
- **Experiments:** See below.
- **How much disk space required (approximately)?:** 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 2 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Workflow framework used?:** ParaDnn
- **Archived (provide DOI)?:** 10.5281/zenodo.3687363

A.3 Description

A.3.1 How to access

Our source code is available on GitHub: `https://github.com/Emma926/paradnn`.

A.3.2 Hardware dependencies

Any CPU, GPU, or TPU platforms that support Python 3.x and TensorFlow 1.6-1.13.

A.3.3 Software dependencies

Python $\geq$ 3.7.6

TensorFlow 1.6-1.13

A.3.4 Data sets

This workflow will generate data sets. No extra data set is needed.

A.3.5 Models

This workflow runs the FC, CNN and RNN models in ParaDnn.

A.4 Installation

Python

```
$ sudo apt update
$ sudo apt install python3-dev python3-pip
```

TensorFlow Please refer to the installation guide for TensorFlow. If GPU is used, tensorflow-gpu and its dependencies are needed. If Cloud TPU is used, no installation is needed because software is included in the cloud instance.

After setting up virtual environment based on the TensorFlow installation guide, please do

```
$ pip3 install tensorflow==1.13.1
$ pip3 install google-api-python-client
$ pip3 install oauth2client
$ pip3 install notebook
$ pip3 install seaborn
$ pip3 install matplotlib
$ pip3 install sklearn
```

ParaDnn

```
$ git clone https://github.com/Emma926/paradnn
```

A.5 Experiment workflow

Test Environment

```
$ cd paradnn/
$ python test.py --use_tpu=False
```

Run ParaDnn

```
$ # cd to paradnn/paradnn
$ cd paradnn/
$ bash run/fc_cpu.sh
```

Collect Performance Data

```
$ cd ../scripts
$ python get_perf.py
```

Run Analysis Tools

```
$ cd scripts/plotting
$ jupyter notebook
```

Open “Demo.ipynb” and run through the code blocks sequentially.

A.6 Evaluation and expected result

Our analysis methodology is presented in the plots from Jupyter Notebook. The methodology is evaluated if the plots show similar trends as the ones in the paper.

A.7 Experiment customization

All the run scripts are in paradnn/paradnn/run/*.sh. To run RNN models of ParaDnn, simply do “bash run/rnn_cpu.sh”. To run on other platforms, such as GPUs and Cloud TPUs, use files “run/*_gpu.sh” and “run/*_tpu.sh”. Users can also change the range of ParaDnn parameters in paradnn/-paradnn/run/*.sh. The sweeping range of parameters largely determines the reproducibility of this artifact. Users can also modify Jupyter Notebooks, for example, to analyze the speedup of TPU over CPU.

A.8 Notes

For more questions, please file issues on GitHub.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20200102.html
- http://cTuning.org/ae/reviewing-20200102.html
- https://www.acm.org/publications/policies/artifact-review-badging