

# Benchmarking TPU, GPU, and CPU Platforms for Deep Learning

Yu (Emma) Wang, Gu-Yeon Wei and David Brooks

{ywang03,gywei,dbrooks}@g.harvard.edu

*John A. Paulson School of Engineering and Applied Sciences  
Harvard University*

## ABSTRACT

Training deep learning models is compute-intensive and there is an industry-wide trend towards hardware specialization to improve performance. To systematically benchmark deep learning platforms, we introduce ParaDnn, a parameterized benchmark suite for deep learning that generates end-to-end models for fully connected (FC), convolutional (CNN), and recurrent (RNN) neural networks. Along with six real-world models, we benchmark Google’s Cloud TPU v2/v3, NVIDIA’s V100 GPU, and an Intel Skylake CPU platform. We take a deep dive into TPU architecture, reveal its bottlenecks, and highlight valuable lessons learned for future specialized system design. We also provide a thorough comparison of the platforms and find that each has unique strengths for some types of models. Finally, we quantify the rapid performance improvements that specialized software stacks provide for the TPU and GPU platforms.

## 1. INTRODUCTION

Deep learning has revolutionized many application domains, defeating world champions in the game of Go [49], surpassing humans in image classification [28], and achieving competitive accuracy to humans in speech recognition [4] and language translation [57], to name a few. As such, there has been growing demand for new and better hardware and software platforms to support the training and deployment of even more sophisticated models. As researchers from both academia and industry scramble to propose and deploy new systems to meet this demand, there is a great need to concurrently develop a systematic and scientific approach to platform benchmarking. This benchmarking should not only compare performance of different platforms running a broad range of deep learning models, but also support deeper analysis of the interactions across the spectrum of different model attributes (e.g., hyperparameters), hardware design choices, and software support.

Announced in May 2017, the Tensor Processing Unit (TPU) v2 is a custom ASIC. Each TPU v2 device delivers a peak of 180 TFLOPS on a single board. TPU v3 was announced a year later and improves the peak performance to 420 TFLOPS. Cloud TPU became available for early academic access in February 2018. It is used in this paper. The

NVIDIA Tesla V100 Tensor Core is a Graphics Processing Unit (GPU) with the Volta architecture that was released in 2017. CPUs have been found to be suitable for training in certain cases [20] and, therefore, are an important platform to include for comparison. This study shows that no one platform is best for all scenarios. Different platforms offer advantages for different models based on their respective characteristics. Moreover, given how rapidly deep learning models evolve and change, benchmarking must be updated continuously and run frequently.

Recent benchmarking efforts have been limited to relatively small collections of seemingly arbitrary DNN models [41, 3, 12, 51]. Focusing on well-known models such as ResNet50 [21] and Transformer [54] can lead to misleading conclusions. For example, Transformer is a large FC model that trains  $3.5\times$  faster on the TPU compared to the GPU; yet focusing on this single model would not reveal the severe TPU memory bandwidth bottleneck that arises with FCs with more than 4k nodes. This highlights the risk of overly optimizing hardware and/or compilers for certain models.

This paper proposes a collection of deep learning models (for training) created and curated to benchmark a set of state-of-the-art deep learning platforms. In order to support broad and comprehensive benchmark studies, we introduce **ParaDnn**, a parameterized deep learning benchmark suite. ParaDnn seamlessly generates thousands of parameterized multi-layer models, comprising fully-connected models (FC), convolutional neural networks (CNN), and recurrent neural networks (RNN). ParaDnn allows systematic benchmarking across almost six orders-of-magnitude of model parameter size, exceeding the range of existing benchmarks.

We combine these parameterized models with a collection of six real-world models, which serve as unique points within a broad spectrum of model attributes, to provide comprehensive benchmarking of hardware platforms. Table 1 summarizes fourteen observations and insights described throughout the paper that can inform future domain-specific architecture, system, and software design. We specifically mark the insights enabled by ParaDnn. We start with a deep dive into the TPU v2 and v3 in Section 4, revealing architectural bottlenecks in computation capability, memory bandwidth, multi-chip overhead, and device-host balance (observations 1 through 5). Section 5 provides a comprehensive comparison

Observation	ParaDnn*	Proof	Insight/Explanation
1. TPU does not exploit the parallelism from the model depth (layer count).	✓	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, and intra-chip and host-device interconnect). TPU serves as a great example.
2. Many FC and CNN operations are bottlenecked by TPU memory bandwidth.	✓	Fig 3	
3. TPU suffers large overheads due to inter-chip communication bottlenecks.	✓	Fig 4	
4. TPU performance can be improved by $\geq 34\%$ by improving data infeed.	-	Fig 5	
5. TPU v3 optimizes compute-bound MatMuls by $2.3\times$ , memory-bound ones by $3\times$ , and large embeddings by $> 3\times$ , compared to v2.	✓	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 $2\times$ (CNN) and $3\times$ (RNN) 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 improved by up to $2.5\times$ (CNN), $7\times$ (FC), and $9.7\times$ (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 5 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 shipped.

\* Without ParaDnn the insights are not revealed, and/or lack deep explanations.

Table 1: A summary of major observations and insights grouped by section of the paper.

of TPU and GPU performance, highlighting important differences between the two platforms (observations 6 through 11). The final three observations are detailed in Section 6, which explores the performance improvements of specialized software stacks and quantized datatypes.

It is important to identify limitations of the study. This paper highlights optimization opportunities in current architecture and system designs, as they provide valuable lessons for future design. Optimization details are beyond its scope. For example, the analysis focuses on training and not inference. We do not study the performance of multi-GPU platforms or 256-node TPU systems, which may lead to different conclusions. Section 7 discusses these and other limitations of the study, which also motivate future work.

## 2. DEEP LEARNING BENCHMARKING

Recent success of deep learning (DL) has motivated development of benchmark suites, but existing suites have limitations. There are two types, real-world benchmark suites such as MLPerf [41], Fathom [3], BenchNN [12], and BenchIP [51], and micro-benchmark suites, such as DeepBench [43] and BenchIP. Each real-world suite contains a handful of popular DL models spanning a variety of model architectures. Their limitation is that 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) that are common in neural networks, but they cannot simulate complex dependencies between different operations in end-to-end models.

To complement existing benchmark suites for this study, we introduce ParaDnn, a parameterized benchmark suite for deep learning.<sup>1</sup> ParaDnn has the advantages of the above

<sup>1</sup>We plan to open-source ParaDnn.

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. 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., ResNet50 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 benchmark suites or DeepBench’s microbenchmarks, which do not capture inter-operation dependencies as ParaDnn does.

### 2.1 ParaDnn Models

ParaDnn includes end-to-end fully connected models (FC), convolutional neural networks (CNN), and recurrent neural networks (RNN). The model types cover 95% of Google’s TPU workloads [32], all of Facebook’s deep learning models [20], and eight out of nine MLPerf models [41] (with reinforcement (minigo) as an exception). The image classification/detection and sentiment analysis models are CNNs; the recommendation and translation models are FCs; the RNN translator and another version of sentiment analysis are RNNs. Speech recognition (DeepSpeech2) is a combination of CNN and GRU models.

**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 of input and output units of the datasets.

**Convolutional Neural Networks** CNN models are residual networks, the state-of-the-art model for image classification.

Variable	Layer	Nodes	Input	Output	Batch Size
<b>Min</b>	4	32	2000	200	64
<b>Max</b>	128	8192	8000	1000	16384
<b>Inc</b>	$\times 2$	$\times 2$	+2000	+200	$\times 2$

(a) Fully Connected Models

Variable	Block	Filter	Image	Output	Batch Size
<b>Min</b>	1	16	200	500	64
<b>Max</b>	8	32	300	1500	1024
<b>Inc</b>	+1	64	+50	+500	$\times 2$

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

Variable	Layer	Embed	Length	Vocab	Batch Size
<b>Min</b>	1	100	10	2	16
<b>Max</b>	13	900	90	1024	1024
<b>Inc</b>	+4	+400	+40	$\times 4$	$\times 4$

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

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

The architecture of ParaDnn CNNs is

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

A residual network contains four groups of blocks [21]. 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. Usually the minimum number of filters of a residual network is 64 and it doubles in every group, so the maximum is 512 filters. We sweep the number of blocks per group, the minimum filters, and the datasets, including input images and number of categories as outputs. An input image is square with three channels, represented by its length. To keep the study tractable, we constrain each group to have the same number of blocks.

**Recurrent Neural Networks** RNNs comprise multiple layers of basic RNN, LSTM, or GRU cells as shown below.

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

Each token of the input sequence is embedded within a fixed length vector, and the length of the vector is the embedding size. In ParaDnn, the number of layers and the embedding size are variable. The variables in the dataset include the maximum length per input sequence and the vocabulary size. **Range of Hyperparameters and Datasets** We choose the range of hyperparameters and datasets to cover the real models (Section 2.2), and we make sure the design space is reasonable. Table 2 summarizes variables for each network type and how they are swept. We also sweep training batch sizes.

## 2.2 Real-World Models

In addition to ParaDnn, we include two of the three workloads written in TensorFlow from MLPerf [41], i.e., Transformer (translation) [54] and ResNet-50 (image classification) [21], because currently TPU only supports TensorFlow. We also select other real-world deep learning workloads [42], including RetinaNet [37], DenseNet [28], MobileNet [27], and SqueezeNet [29]. We refer to them as real workloads or real models. The batch sizes are the largest supported on the hardware platform. For example, on TPU with bfloat16, we use batch size 64 for RetinaNet, 4k for Transformer, and 1024 for the rest of the workloads.

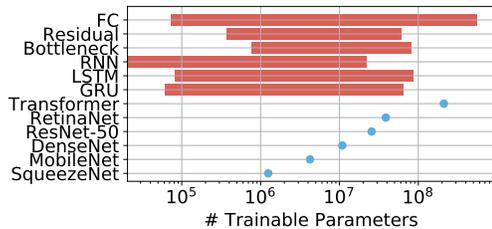


Figure 1: The numbers of trainable parameters for all workloads in this paper. Those from ParaDnn range from 10k to nearly a billion parameters, which is larger the range of real workload sizes, shown as dots.

Figure 1 shows the numbers of trainable parameters across all workloads to quantify the size 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, reflect models typically targeted towards mobile applications. RetinaNet and ResNet-50 provide state-of-the-art image classification accuracy.

## 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 cores and 32 threads. It has the largest 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** The TPU is a Cloud TPU instance to which we were given academic access in February 2018. Its system architecture includes a Cloud Engine VM, a Cloud TPU server, Google Cloud storage, and a Cloud TPU board [2]. Each TPU board contains four TPU packages (the default Cloud TPU configuration) [14]. One TPU v2 package supports 45 TFLOPS and contains 2 cores. One core has one matrix unit (MXU). Total ML acceleration for a Cloud TPU v2 platform is 180 TFLOPS. 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 to compute and float32 to accumulate. Compared to v2, TPU v3 doubles the number of MXUs and HMB capacity per core [2]. The memory bandwidth has not been disclosed, but empirical results show that it is increased by  $1.5\times$ . TPU v3 has a peak of 420 TFLOPS,  $2.3\times$  greater than v2, likely because of higher frequency. Because v3 is an upgrade from v2, we focus on studying v2. In this paper, TPU refers to Cloud TPU v2, unless specified otherwise.

*Understanding TPU memory size.* Data parallelism is im-

Platform	Unit	Version	Mem Type	Mem (GB)	Mem Bdw (GB/s)	Peak FLOPS
CPU	1 VM	Skylake	DDR4	120	16.6	2T SP <sup>†</sup>
GPU (DGX-1)	1 Pkg	V100 (SXM2)	HBM2	16	900	125T
TPU	1 Board (8 cores)	v2	HBM	8	2400	180T
TPUv3	8 cores	v3	HBM	16	3600*	420T

<sup>†</sup> Single precision:  $2 \text{ FMA} \times 32 \text{ SP} \times 16 \text{ cores} \times 2 \text{G frequency} = 2 \text{ SP TFLOPS}$

\* Estimated based on empirical results (Section 4.5).

Table 3: Hardware platforms under study.

plemented on the TPU, where one batch of training data is split evenly and sent to the 8 cores on the TPU board. The model is not distributed; every TPU core keeps a whole copy of it. Therefore memory size per core determines the maximum model supported, while total on-board memory determines the maximum data batch size. That is why in Section 5.1, the GPU platform supports larger models than the TPU, and the TPU supports larger batch sizes (Section 5.2).

*Comparison rationale.* We evaluate one V100 package and one TPU board (4 packages) because they are the minimal units available. The configurations are encapsulated. On Cloud TPU, distribution of computation across the four TPU packages on a TPU board happens automatically. On the other hand, multi-GPU performance depends largely on the user’s implementation. Multi-GPU/TPU performance is beyond the scope of this work as discussed in Section 7. Therefore, note that conclusions in this paper do not apply to multi-GPU or larger TPU systems.

## 4. TPU ARCHITECTURAL IMPLICATIONS

As the end of Dennard scaling and Moore’s law has slowed the performance improvement of general-purpose microprocessors [23], the design of domain-specific hardware is becoming more and more relevant. The TPU is a prominent example of domain-specific hardware [32, 14]. 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 machine learning 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.

In this section, we study the performance characteristics of TPU v2 and v3 [14, 2] with a focus on v2, from the computation capability in the core (FLOPS) to the system balance. Based on our observations, we discuss possible steps to improve TPU performance, which can be generalized to other deep learning accelerator systems. The following is a summary of our key observations and insights:

- **FLOPS (Section 4.1):** TPU makes good use of the parallelism exposed by batch size and model width, but parallelism due to model depth is under-exploited, suggesting opportunities for model pipelining [8].
- **Memory bandwidth (Section 4.2):** Memory bandwidth is the performance bottleneck of many models. Even highly-optimized compute-bound models show a significant fraction of memory-bound operations (13% in ResNet-50). Improving memory access for such operations is key to further performance improvement.

- **Multi-chip overhead (Section 4.3):** Communication overhead in a multi-chip system is non-negligible (up to 13% for CNNs with sizes similar to ResNet-50) but can be amortized with large batch sizes. Reducing the communication overhead can lead to performance gain.
- **Host-device balance (Section 4.4):** Data quantization can make compute-bound workloads data-infeed-bound. Resolving the data-infeed bottleneck can improve performance by at least 34%.
- **TPU v3 (Section 4.5):** The maximum speedup of TPU v3 over v2 is up to  $3\times$ , exceeding the  $2.3\times$  FLOPS increase. TPU v3 benefited from its doubled memory capacity (which allows twice the batch size of v2) as well as increased memory bandwidth.

### 4.1 FLOPS Utilization

Floating point operations per second (FLOPS) utilization is the ratio of average FLOPS to peak FLOPS, measuring how efficiently the computation capacity of a platform is used. We discuss the TPU FLOPS utilization of the parameterized models in this section. We first visualize how the model hyperparameters listed in Table 2 affect FLOPS utilization. Then we introduce an analysis methodology to quantify the hyperparameter effect using linear regression.

**FLOPS Utilization Heat Maps** Figure 2(a)–(c) presents heat maps of FLOPS utilization for FC, CNN, and RNN models, obtained by sweeping the hyperparameters with ranges listed in Table 2. We choose two hyperparameters for each model type that affect FLOPS utilization the most (see below for how we choose them) and show them on the  $x$ - and  $y$ -axes while keeping the other hyperparameters fixed. Specifically, we fix layer (32), input (2000), and output units (1000) for FCs, block (6), input image size ( $300 \times 300 \times 3$ ), and output unit (1000) for CNNs, and layer (9), vocabulary size (32), and max length (50) for RNNs.

Figures 2(a)–(c) show that the FLOPS utilization of all three models increases with batch size. Other than that, the FLOPS utilization of FCs increases with number of nodes per layer (Figure 2(a)), that of CNNs increases with filters, and that of RNNs with embedding size. This indicates that TPU is capable of leveraging the parallelism within a batch (the former) and within the width of the models (the latter).

**Studying Parameterized Models with Linear Regression** Having discussed the qualitative effects of hyperparameters on FLOPS utilization, we now build a linear regression (LR) model and use the weights to quantify these effects. Note that the LR model is only for measuring the effects of hyperparameters. We do not use it for prediction.

In the case of FC, the linear regression model is

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

where  $w_0$ – $w_4$  are the weights of the hyperparameters. To train the LR model, all the values are normalized to the same scale, so that we can use the weights as a measure of importance. For example, positive  $w_1$  indicates that node count affects performance positively. If the absolute value of  $w_1$  is larger than that of  $w_0$ , it indicates node count has a larger effect on FLOPS than layer count. Other similar metrics for feature selection, including T-test and F-test, may be used for this

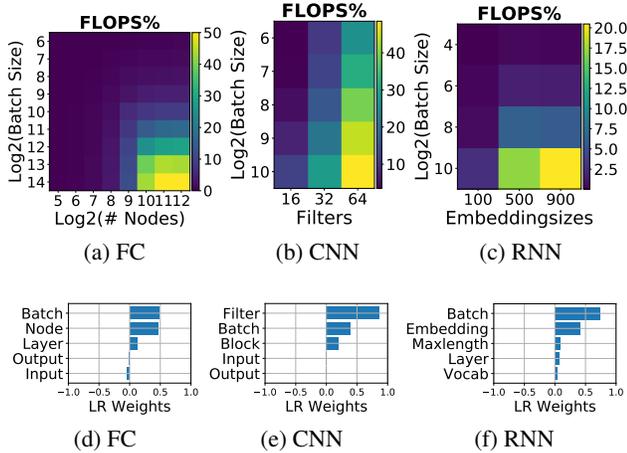


Figure 2: FLOPS utilization and its correlation with hyperparameters. (a)–(c) show FLOPS utilization of parameterized models. (d)–(f) quantify effects of model hyperparameters on FLOPS utilization, using linear regression weights.

purpose [26]. We choose LR mainly to get the signs of the weights, which indicate the positive or negative effects of the hyperparameters on performance, while T-test and F-test only report positive values as importance.

Figures 2(d)–(f) show the LR weights of the model hyperparameters. The  $x$ - and  $y$ -axes in Figures 2(a)–(c) are the hyperparameters with the highest absolute values in Figures 2(d)–(f). Figure 2(d) shows that the FLOPS utilization of FC is largely affected by batch size and node, while layer, output, and input do not matter as much. Similarly, Figure 2(e) shows filter is the most important, and batch size is more important than block, while input and output have minimal impact. The TPU FLOPS of RNNs is not affected by maximum length, number of layers, or vocabulary size.

**Architectural Implications** The TPU takes advantage of parallelism due to large batch size and model width, including that from nodes per layer in FC, filters in CNN, and embedding sizes in RNN. Parallelism opportunities from large numbers of layers remain to be explored, by approaches such as model parallelism [15, 30] and pipelining [8].

## 4.2 Roofline Model Analysis

The FLOPS utilization in the previous section shows the computation capability of TPU, but the core is only part of the problem when designing an accelerator. In particular, memory bandwidth is another important aspect that can have significant impact on performance. In this section, we use the roofline model [56] to analyze the computation and memory bandwidth of FCs and CNNs. Roofline models are useful to demonstrate memory and computation bottlenecks [56, 32]. We omit RNN models because the TPU profiler reports incorrect numbers for memory bandwidth of RNN models.

**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  $x/y =$  memory bandwidth. The horizontal

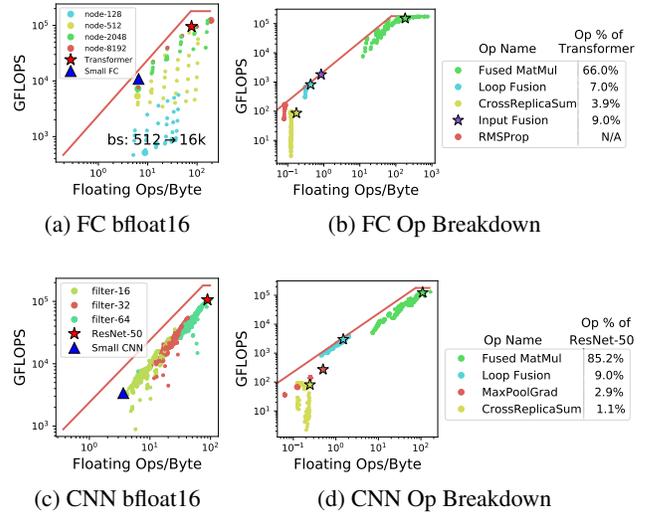


Figure 3: Rooflines for FC and CNN on TPU. Workloads with matrix multiply (MatMul) operations are compute-bound. Even compute-bound workloads like Transformer and ResNet-50 have more than 10% memory-bound operations. (a) and (c) show rooflines of parameterized and real-world models. (b) and (d) show the operation breakdown.

part is the peak FLOPS on the hardware. 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 parameterized FC and CNN models (dots) plus Transformer and ResNet-50 (stars). Figures 3(b) and 3(d) show all the operation breakdowns. Transformer and ResNet-50 are just instances (sparse design points) in ParaDnn, so the stars overlap some of the dots. This is because ParaDnn enables more comprehensive model architecture design space exploration and supports benchmarking hardware systems more systematically. An exception is that some operations of Transformer do not align closely with those of FCs. This results from a choice in this paper, not a fundamental flaw of ParaDnn. ParaDnn uses the RMSProp optimizer, keeping nodes per layer uniform in a parameterized FC, while Transformer uses the *adafactor* optimizer and has layers with 4k, 2k, and 512 nodes.

**FC** 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 large batch sizes increase the computation per weight/activation transferred, i.e., the arithmetic intensity. For example, for FCs with  $\geq 2k$  nodes, using large batch sizes turns memory-bound FCs into compute-bound. Specifically, the 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, which causes it to overlap with FCs having 4k batch sizes.

**CNN** Figure 3(c) shows that models close to ResNet-50 are compute-bound, while a majority of the CNNs are bottlenecked by memory bandwidth. As it is in log scale, it shows that practically achievable memory bandwidth for the CNNs is less than the theoretical 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.

**Operation Breakdown** The triangles in Figures 3(a) and 3(c) are selected memory-bound models. The FC has 8 layers, 8192 nodes per layer, and batch size 512; the CNN has 1 block per group, 16 filters, and batch size 64. Figures 3(b) and 3(d) show the TensorFlow operations taking more than 1% of the workload execution time and more than 0 TPU FLOPS. The arithmetic intensity of such operations can be as low as 0.125.<sup>2</sup> The TensorFlow breakdown in Figure 3 is generated after operation fusion, which is a technique combining and executing several operations together for higher efficiency.

**Large MatMuls** Figures 3(b) and 3(d) show that the only compute-bound operation is large fused MatMul (matrix multiply fused with other operations), so a compute-bound model needs to have compute-bound MatMuls. Other operations are closer to the slanted line, indicating they are constrained by memory bandwidth. For example, in Figure 3(a) and (c), Transformer and ResNet-50 are compute-bound because they have compute-bound MatMuls in Figures 3(b) and 3(d).

**Memory-bound Operations** Interestingly, even compute-bound FC/CNN 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 three operations contribute to 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.) Even compute-bound ResNet-50 has many memory-bound operations, including loop fusion (9%), MaxPoolGrad (2.9%), and CrossReplicaSum (1.1%), which sums to 13%, showing the need for both end-to-end and per-operation optimization for deep learning accelerators.

**Architectural Implications** Compute-bound FCs and CNNs have large MatMul operations. Surprisingly, even compute-bound models contain non-negligible fractions (19.9% for Transformer and 13% for ResNet-50) of memory-bound operations. Given the current TPU system, memory-bound operations need more attention. Potential ways to speed up memory-bound operations include increasing memory bandwidth and reducing memory traffic. Traditional architectural efforts to reduce memory traffic can be adopted, such as exploiting the memory locality by caching [24]. Software/compiler approaches include better operation fusion [1, 11, 44], more aggressive data quantization [6], and weights and gradients compression [17, 38].

### 4.3 Multi-Chip Overhead

This section analyzes communication overhead in a multi-chip system. Previous sections focus on the compute and memory bandwidth of a TPU core. But these are not the only factors that affect training performance, because typical large-scale training systems use multiple chips [15]. This

<sup>2</sup>For example, 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.

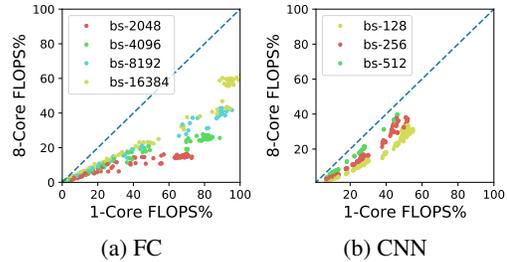


Figure 4: Communication overhead in a multi-chip system is non-negligible, but is reduced with large batch sizes.

section evaluates the scalability of a multi-chip TPU system.

To quantify the multi-chip overhead, we compare the FLOPS utilization of 1-core ( $x$ -axis) and 8-core TPU ( $y$ -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  $x = y$ . On the 8-core TPU, FCs need at least 16k batch size to achieve more than 50% FLOPS utilization. Specifically, FCs with  $\geq 256$  nodes and  $\leq 512$  batch size are faster to run on 1-core TPU than on 8-core TPU. Therefore we consider FCs with larger than 1024 batch size in Figure 4.

As shown in the figure, 8-core TPU shows noticeably lower FLOPS utilization than 1-core TPU, indicating significant inter-core communication overhead. For FC, the maximum FLOPS utilization in 8-core TPU is 62%, compared to 100% in 1-core TPU. Multi-chip overhead is less noticeable in CNNs, with FLOPS utilization decreasing from 55% in 1-core TPU to 40% in 8-core. It is worse for FCs because there are more weights to synchronize across the TPU cores than for CNNs. Based on Amdahl’s law, we calculate that the maximum non-parallel fraction of the workloads is up to 60% for FC and 40% for CNN. The FLOPS utilization difference is smaller with larger batch sizes for both FC and CNN, because it increases the computation without increasing the weight synchronization. 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.

**Architectural Implications** We show that communication overhead in multi-chip systems is non-negligible even for large FCs and CNNs. Using large batch size can reduce the overhead by increasing the computation parallelism without increasing weight transfers. Possible optimizations include relaxed synchronization, model parallelism [15], gradient compression [38], and algorithm and architecture support for weight pruning and compression [17] before synchronization.

### 4.4 Host-Device Balance

Previous subsections have focused on the performance of the accelerator itself. This section focuses on “data infeed,” the process of preparing and moving input data to the TPU board. ParaDnn analysis avoids part of the data infeed overhead by synthesizing data on the CPU host. We now describe a case study with real-world workloads to show the importance of balancing accelerators and the host in a system.

**TPU Device and Host** The TPU system is composed of a CPU host and a TPU device [14]. For real-world CNNs, the host fetches images from the network, decodes, preprocesses, and feeds them to the device. Figure 5 calls this data prepara-

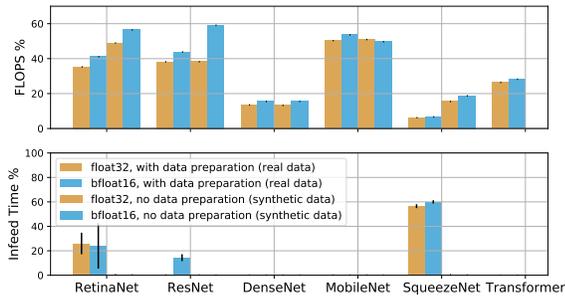


Figure 5: FLOPS utilization (top) and infeed time (bottom) of the real models using float32 and bfloat16, with and without data preparation. Models with large infeed time percentage, i.e., RetinaNet and SqueezeNet, are limited by data infeed.

tion. The device then performs training computation on the images. Data infeed means network overhead, host compute, and bandwidth between host and device.

**Infeed Overhead Analysis** To quantify the infeed overhead, we run real-world workloads both with and without data preparation, by directly feeding synthetic data as post-processed inputs. We also compare models using float32 to those with bfloat16, because replacing float32 with bfloat16 can affect the execution time of both data infeed and device computation. First, the arithmetic intensity of all operations doubles, because the same computation can be performed with half of the bytes transferred. Second, the FLOPS of memory-bound operations improves in the device, because increased arithmetic intensity moves those operations towards the upper right in the roofline model of Figure 3. Third, improved device performance increases the need for faster data infeeding, which puts more pressure on the host.

Figure 5 shows FLOPS utilization and infeed time of the real-world workloads. FLOPS utilization measures computation efficiency and infeed time measures how long the device waits for data, both of which are collected from the TPU profiler. The error bars are one standard deviation of the one-minute samples from the profiler.

The figure shows that the bottleneck of a workload can be on the device or in data infeed by different degrees under different circumstances. Data infeed bottlenecks RetinaNet and SqueezeNet, as the performance increases noticeably when data preparation is skipped. Eliminating that bottleneck brings 37% and 180% speedup, respectively, for RetinaNet and SqueezeNet using bfloat16. RetinaNet’s bottleneck is likely because it uses the COCO dataset ( $640 \times 640$  images), while others use the ImageNet dataset ( $224 \times 224$  images).

ResNet-50 is bottlenecked by the device when using float32, and by data infeed when using bfloat16. That bitwidth reduction speeds device execution and increases FLOPS utilization so that training throughput on the device surpasses data preparation throughput on the host. If the resulting data infeed bottleneck can be resolved, the performance of bfloat16 ResNet-50 can be improved by 34%. Switching RetinaNet and SqueezeNet from float32 to bfloat16 with real data slightly increases the data infeed percentage as well for similar reasons. It also shows that performance can be improved when infeed time increases.

DenseNet and MobileNet have zero data infeed time. Com-

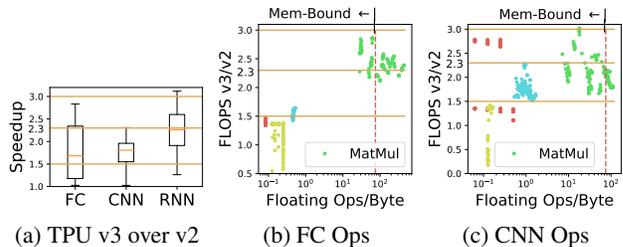


Figure 6: (a) Speedup of TPU v3 over v2 running end-to-end models. (b) and (c) Speedup comparison for FC and CNN operations. TPU v3’s larger memory supports doubled batch sizes, so memory-bound operations have triple speedup if they benefit from larger batch size, and  $1.5\times$  speedup if not. Compute-bound on v3 operations have  $2.3\times$  the speedup. The red line (75 Ops/Byte) is the inflection point in the TPU v2 roofline. (See roofline and legends in Fig 3.)

pared with ResNet, they train fewer images/second, putting less stress on the host to infeed data. Switching from float32 to bfloat16 increases the performance of both workloads using real data. Thus they are likely bottlenecked by memory bandwidth in the device.

Unlike CNNs, Transformer processes sequences, which are smaller than images and demand minimal computation for data decoding and/or preprocessing. So Transformer does not have significant infeed time, as expected. Unfortunately, its tensor2tensor implementation does not support synthetic data, so we omit the shaded bars for Transformer in Figure 5. **Architectural Implications** Scaling performance of the CPU host to match the TPU device is crucial for utilization of the accelerator’s computation resource. For workloads limited by data infeed from the host to the device, resolving the bottleneck can improve performance by at least 34%. Such workloads include RetinaNet, ResNet-50, and SqueezeNet using bfloat16. Sequence models such as Transformer do not stress data infeed as much as CNNs. By increasing FLOPS utilization, data quantization can turn a compute-bound workload into one that is infeed-starved. With a powerful CPU host, further data quantization can yield greater performance gain, if it is valid. 8-bit training is an example [6].

## 4.5 TPU v3

This section focuses on the differences between TPU v2 and v3. Figure 6 compares TPU v3 and v2 using FC, CNN with bottleneck block, and basic RNN models. Batch size for v3 is twice that for v2, thanks to its doubled memory capacity. Figure 6(a) shows the speedups of end-to-end ParaDnn models. Because end-to-end model speedup depends on operations, we first discuss the operation breakdown in detail. Figure 6(b)–(c) show arithmetic intensity on the  $x$ -axis and the speedup of FC and CNN operations on the  $y$ -axis. Data points are colored by operation types, consistently with Figure 3(b) and (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 ( $180 \text{ TFLOPS} / 2.4 \text{ TB/s}$ ). The operations on the left of the red line are memory-bound, and the ones on the right are compute-bound. We can group the operations in four classes, as follows.

**Compute-Bound Ops** The peak FLOPS of TPU v3 is  $2.3\times$

that of v2, so compute-bound operations are improved by about  $2.3\times$  on v3. Such operations are on the right of the red dashed line in Figure 6(b).

**Memory-Bound Ops ( $2\times$  batch size)** The maximum speedup of the memory-bound operations (mainly the MatMuls in Figure 6(b)–(c)) is  $3\times$ . The tripled speedup comes from doubled batch size (enabled by doubled memory capacity) and memory bandwidth improvement. Thus we can infer v3 has  $1.5\times$  bandwidth improvement (3.6 TB/s per board) over v2, although its memory bandwidth has not been officially disclosed. This is because on the slanted line of a roofline model, doubled batch size means doubled arithmetic intensity, and thus doubled FLOPS, because the ratio of FLOPS to arithmetic intensity is fixed. And switching from v2’s roofline to v3’s gives a FLOPS improvement equal to the bandwidth improvement. The fact that the overall speedup is  $3\times$  indicates that the bandwidth improvement is  $3/2 = 1.5\times$ .

**Other Memory-Bound Ops** The  $1.5\times$  bandwidth improvement assumption is corroborated by the  $1.5\times$  speedup of other memory-bound operations, represented by the non-MatMul FC operations in the lower left corner of Figure 6(b). 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  $1.5\times$  performance improvement in Figure 6(b) is from bandwidth improvement. **Boundary Cases** The compute-bound MatMuls in Figure 6(c) 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  $x = 420/(2.4 * 1.5) = 117$ . 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(a) the maximum speedups are  $2.83\times$  (FC),  $2.31\times$  (CNN), and  $3.11\times$  (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  $3\times$  speedup are dominated by memory-bound MatMuls. Exceptions are RNNs with more than  $3\times$ ; these have the largest embedding size (900), indicating that TPU v3 optimizes large embedding computations.

**Architectural Implications** ParaDnn enables users to exam a wide range of workloads, from memory-bound to compute-bound. Compared to v2, TPU v3 shows three main levels of speedup:  $2.3\times$  for compute-bound operations,  $3\times$  for memory-bound MatMuls, and  $1.5\times$  for other memory-bound operations. This is the result of its  $2.3\times$  FLOPS,  $2\times$  memory capacity, and  $1.5\times$  memory bandwidth. For architects, the relative improvement of FLOPS and memory is a trade-off based on key workloads and budgets.

## 5. CROSS-PLATFORM COMPARISON

In this section, we conduct cross-platform comparison using TPU, GPU, and CPU, so that users can choose the most suitable platform based on models of interest. We find that there are scenarios where each of the platforms is valuable, trading off flexibility and specialization. We also discuss the implications for future architecture designs. The following is a summary of the key takeaways:

- **TPU** is highly-optimized for large batches and CNNs, and has the highest training throughput.

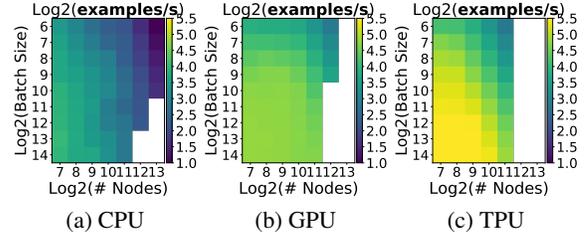


Figure 7: Examples/second of fully-connected models with fixed layer (64). Examples/second decreases with nodes and increases with batch size. White squares indicate models that encounter out-of-memory issues. The CPU platform runs the largest model because of its large memory.

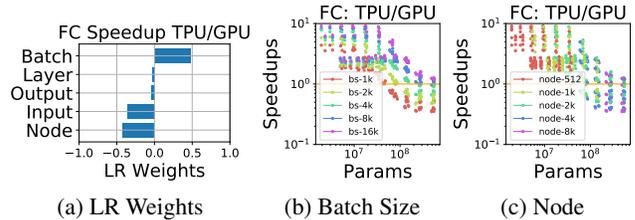


Figure 8: Small FC models with large batch sizes prefer TPU, and large models with small batch sizes prefer GPU, indicating systolic arrays are better with large matrices, and the warp scheduling on GPU is more flexible for small matrices.

- **GPU** shows better flexibility and programmability for irregular computations, such as small batches and non-MatMul computations. The training of large FC models also benefits from its sophisticated memory system and higher bandwidth.
- **CPU** has the best programmability, so it achieves the highest FLOPS utilization for RNNs, and it supports the largest model because of large memory capacity.

We consider two performance metrics, examples/second and speedup. Examples/second measures the number of examples trained per second, which is throughput. We use it as a proxy for end-to-end performance. The speedup of one platform over another is the ratio of the former’s performance (examples/second) over the latter’s.

### 5.1 Fully-Connected DNNs

This subsection provides systematic analysis of the performance and speedups for fully-connected (FC) models.

**Examples/second** Figure 7 shows throughput for varying node counts and batch sizes but fixed layer count (64). We use LR weights introduced in Section 4.1 to quantify the hyperparameter effects (not shown owing to space limitations). Layer and node counts have negative weights, because it is time consuming to train large models with many layers and nodes. Batch size greatly improves examples/second on GPU and TPU, but not CPU, because the parallelism available with small batch sizes is enough to highly utilize CPU.

It is interesting to note that 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 the largest model supported is determined by memory per core, as explained in Section 3. In Figure 7, the

white squares indicate models that encounter out-of-memory (OOM) issues. CPU has the highest memory per core (120 GB), and GPU (16 GB) is higher than 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 reports using dual-socket, high-memory CPU servers to train ranking models for News Feed and to perform anomaly detection (Sigma), both of which are fully-connected networks [20]. That fact emphasizes the need for model parallelism and pipelining [15, 30, 8] on GPU and TPU, such that those powerful accelerators can support larger models.

**TPU over GPU Speedup** To further investigate the best hardware platform for an FC model, we analyze TPU over GPU speedups. Figure 8(a) plots the linear regression weights across FC hyperparameters for TPU over GPU speedup. To show the design space of FC models, Figures 8(b)–8(c) are scatter plots showing numbers of model parameters on the  $x$  axis and speedups on the  $y$  axis. To display the effects of the hyperparameters, we color code data points to reflect batch size (Figure 8(b)) and node count (Figure 8(c)). Overall, 62% of the FC models perform better on TPU (speedup  $> 1$ ).

TPU is well suited for large batch training, because systolic arrays are very good at increasing throughput [35]. The positive weight in Figure 8(a) and the horizontal color bands in Figure 8(b) show that large batch size is the key to higher TPU over GPU speedup. This suggests that the matrix multiply units (MXU) of TPU, implemented with systolic arrays [32, 14], need large batches to reach full utilization. But GPU is a better choice for small batch sizes, because it executes computation in warps, so it packs small batches and schedules them on stream multiprocessors more easily [39].

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

**GPU over CPU Speedup** The speedup of GPU over CPU is an interesting comparison to TPU over GPU. Figure 9(a) shows the LR weights from learning GPU-over-CPU speedup. Figure 9(b) shows the design space colored by node count.

GPU is a better platform for large FC models, because its architecture is better at exploiting the extra parallelism from large batches and models. As shown by Figure 9, large models have higher speedups on GPU. We also observe that large FC models prefer GPU over TPU, witnessed by the positive trend in Figure 9(b) and the negative trend in Figures 8(b)–8(c). So GPU is the best platform for large FC models, but models with large batch sizes perform best on TPU, and better on GPU than on CPU.

## 5.2 CNN and RNN

We now describe the speedup of CNNs and RNNs. Since

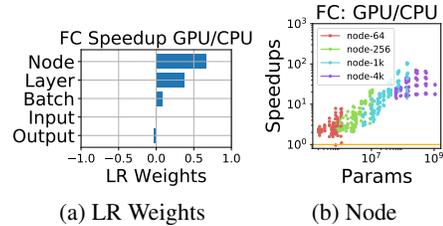


Figure 9: Large FC models with large batch sizes are better suited for GPU than CPU because the GPU’s architecture can better utilize the extra parallelism.

our conclusions for CPUs and the hyperparameter LR weights on examples/second are similar to those in the previous section, we omit those results in the interest of brevity.

**CNN** Figures 10(a)–10(c) show the speedups of TPU over GPU. All CNNs perform better on 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).

TPU is the best platform for large CNNs, suggesting that the TPU 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 TPU is not preferred for large FCs. This suggests it is easier for TPU to optimize for large CNNs than large FCs, which may be because CNNs reuse weights. FC models barely reuse weights, which introduces more memory traffic. GPU is a feasible choice for small CNNs. These conclusions only apply to single-GPU performance; the multi-GPU case 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 greatest in Figure 10(d). Embedding size has negative weights in Figure 10(d) and embedding computation is more sparse than matrix multiplication. This suggests that TPU is less flexible for doing non-MatMul computations than GPU. TPU is better at dense computations like MatMuls. Even so, RNNs are still up to 20× faster on TPU. Optimizing non-MatMul computations is another opportunity for TPU enhancement.

## 5.3 Overall Comparison

This section summarizes the speedup of TPU over GPU and the FLOPS utilization of all parameterized and real models. 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 workloads use larger batch sizes on TPU than on GPU. Speedup of TPU over GPU depends heavily on the nature of the workload measured. The speedup of parameterized models has large ranges, from less than 1 to 10×, while the speedup of real workloads range 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

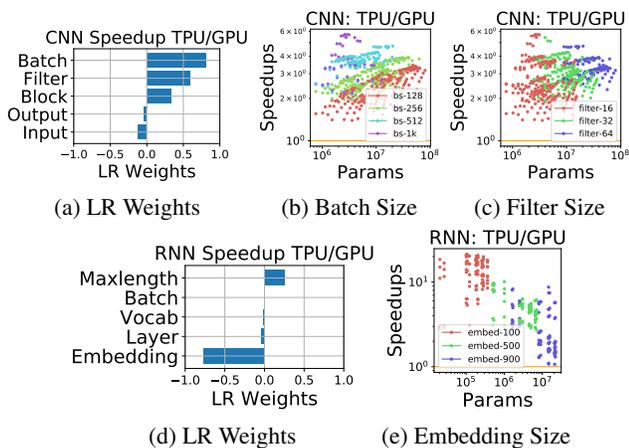


Figure 10: (a)–(c) TPU is a better choice than GPU for large CNNs, suggesting that TPU is highly-optimized for CNNs. (d)–(e) While TPU is a better choice for RNNs, it is not as flexible as GPU for embedding computations.

their behavior than those with only one kind.

To further compare TPU and GPU while relaxing the constraint on the software stack of the GPU, we also include the speedup relative to GPU performance of ResNet-50, reported in NVIDIA’s Developer Blog [9] (annotated as NVIDIA in Figure 11(top)). We 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  $6.2\times$  compared to  $4.2\times$ , 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 TPU is  $2.2\times$  that of GPU for all CNN models, and the ratio is  $3\times$  for RNNs. The TPU FLOPS utilization of Transformers is consistent with FCs with 4k batch size, as shown in Figure 2.

For RNNs, TPU has less than 26% FLOPS utilization and GPU has less than 9%. In contrast, CPU has up to 46% utilization. RNNs have irregular computations compared to FCs and CNNs, due to the temporal dependency in the cells and the variable-length input sequences. The parameterized RNNs are very basic, however. Advanced RNN optimizations may be able to increase utilization on GPU and TPU.

ResNet-50 and RetinaNet have higher FLOPS utilization than DenseNet and SqueezeNet. The real workloads are ranked by number of trainable parameters, shown in Figure 1. DenseNet has lower utilization because it has fewer filters than ResNet-50. DenseNet’s maximum number of filters is 24 [28], and the minimum of ResNet-50 is 64 [21]. SqueezeNet is designed specifically to have fewer parameters with the use of  $1\times 1$  filters [29]. Therefore, parallel operations represent a smaller portion of the whole workload. As a consequence of Amdahl’s law, the small models are unable to utilize the parallelism available on GPU or TPU.

ResNet-50 has higher FLOPS utilization than CNNs with bottleneck blocks. This is because the parameterized CNNs keep the number of blocks the same in each group, while ResNet-50 has more blocks in groups with more filters, and that increases FLOPS.

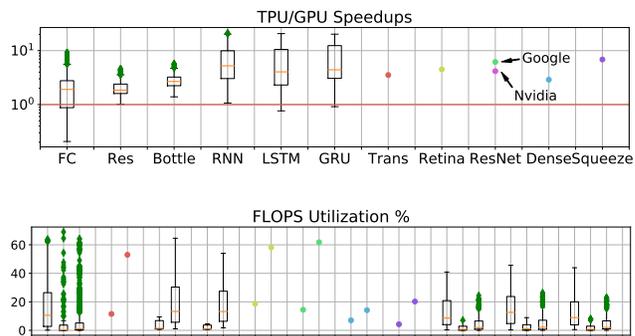


Figure 11: (Top) TPU over GPU speedups of all workloads. Note that the real workloads use larger batch sizes on TPU than on GPU. The NVIDIA version of ResNet-50 is from [9]. (Bottom) FLOPS utilization comparison for all platforms.

## 6. SOFTWARE STACK ADVANCES

Custom hardware for deep learning opens opportunities for dramatic library, toolkit, and compiler optimizations. We now describe how different versions of TensorFlow (TF) and CUDA affect performance. We study data type quantization with software versions, because it depends on software support. As a reminder, for all results in the previous sections, we use the latest versions of each software stack with 16-bit quantization support. Software versions are summarized in the legends of Figure 12. ParaDnn can reveal software optimization focus (e.g., TF 1.9 optimizes small-batch CNNs); we omit these details for brevity.

### 6.1 TensorFlow Versions and TPU Performance

The compiler for the TPU is XLA [36], shipped with TF. Figure 12(a) shows TPU speedups obtained by running TF 1.7 to 1.12, treating 1.7 with float32 as the baseline. The speedup is per model, maximizing batch size in each setting. For example, using bfloat16 instead of float32 allows larger batch size and thus higher speedup.<sup>3</sup> 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  $10\times$  speedup for RNN and  $7.5\times$  for LSTM and GRU. 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 improved significantly. The 90th-percentile speedup of TPU is  $7\times$  for FC,  $1.5\times$  for Residual CNN,  $2.5\times$  for Bottleneck CNN,  $9.7\times$  for RNN, and  $6.3\times$  for LSTM and GRU.

The use of bfloat16 enables significant performance improvement for parameterized FC and CNN models. 90th-percentile speedups are up to  $1.8\times$  for FC and Bottleneck CNN, and  $1.3\times$  for Residual CNN. Depending on the relative memory sizes of the data and model, TPU can usually

<sup>3</sup>These experiments do not consider the impact of quantization on model accuracy.

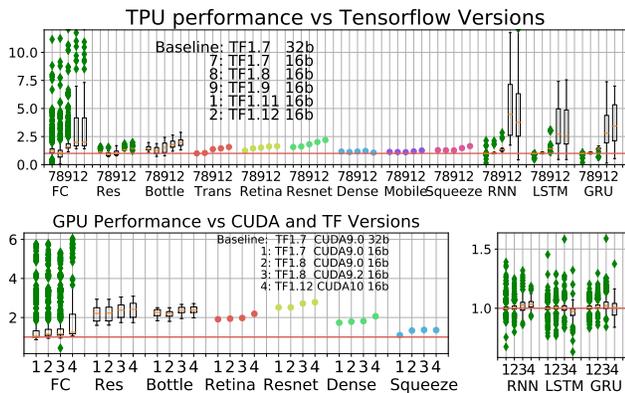


Figure 12: (a) TPU performance with TensorFlow updates. All ParaDnn models improve; Transformer, RetinaNet, and ResNet-50 improve steadily. (b) GPU speedups across versions of CUDA and TF. CUDA 9.2 improves CNNs more than other ParaDnn models, and ResNet-50 more than other real models. CUDA 10 does not improve RNNs or SqueezeNet.

support doubled batch sizes by using 16 bits. Transmitting 16 bits also relieves bandwidth pressure, which can speedup memory-bound operations as discussed in Section 4.2 and Section 4.4. Larger performance increases may be possible with further reductions in bitwidth.

## 6.2 CUDA Versions and GPU Performance

Figure 12(b) 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, bitwidth reduction can speed up CNNs by more than  $2\times$ .

We note that CUDA 9.2 speeds up ResNet-50 significantly more (8%) than other real workloads ( $< 1\%$ ). CUDA 9.2 also speeds up ParaDnn CNNs more than FCs or RNNs. CUDA 10 speeds up other models, but not SqueezeNet. CUDA 10 also improves speedups for ParaDnn FCs and CNNs, but not as much for RNNs. The overall 90th-percentile improvement for FCs is  $5.2\times$ . For ParaDnn residual block and bottleneck block models it is  $2.9\times$  and  $2.6\times$ , respectively. In contrast, the 90-percentile improvement of parameterized models is 8.6% for RNN, 3.5% for LSTM, and 5.9% for GRU. The improvement from CUDA updates is less than that for TF updates on TPU, likely because CUDA and GPU platforms have matured greatly since becoming popular before 2010, while TPU v2 for training was only announced in May 2017.

## 7. LIMITATIONS OF THIS WORK

**Scope of this Work** This work does not study DL inference, cloud overhead, multi-node systems, accuracy, or convergence. We intentionally leave these topics to future work, as each deserves in-depth study. For example, evaluating inference entails different metrics, such as latency, and a different experimental setup, as network overhead may have a large effect. Section 4.4 provides insight towards quantifying the network overhead, and we use synthetic data to minimize the cloud overhead, but virtualization, resource allocation, and job scheduling bring up more research questions.

NVIDIA’s eight-node DGX-1 or Google’s 256-TPU systems are not studied here. Studying multi-node systems in-

volves more system parameters, including numbers of nodes, inter-node bandwidth, inter-connect topology, and synchronization mechanisms. Cloud system overhead also becomes more acute in multi-node systems.

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 batch sizes [47]. It shows that very large batch size results in sub-linear scaling, but the best batch size depends largely on the model and optimizer. In a multi-node system, synchronization becomes more complicated, which results in different convergence behavior.

**Tractability** To keep the experiments tractable, we constrain the parameters in this work, including the ParaDnn hyperparameters (Table 2) and the TPU iterations. For example, we focus on large batches, as the platforms were designed for large batch training, and extremely small batches may lead to different conclusions. We use the RMSProp optimizer, and SGD with momentum performs faster than RMSProp.

## 8. RELATED WORK

**Benchmarks:** “For better or worse, benchmarks shape a field,” said David Patterson [40]. Indeed, benchmarks have been the driving force for compiler and architecture design for decades, and notable examples include the SPEC CPU [25] and PARSEC multiprocessor benchmarks [7]. Recently, work has focused on domain-specific benchmark suites including CortexSuite [52], TonicSuite [18], Sirius [19], Fathom [3], DAWNbench [13], and MLPerf [41]. It is impossible to make any performance conclusions without benchmarks.

Benchmark designers must take care to avoid bias. Existing benchmark suites come with limitations as discussed in Section 2. ParaDnn is the first parameterized benchmark suite for deep learning in the literature. In the same spirit as parameterized benchmarks, synthetic benchmarks have commonly been used, such as BenchMaker [31], and SYMPO [16], constructing benchmarks with hardware-independent characteristics. Some try to match the statistical characteristics of real applications [55, 33]. Synthetic approaches are common in domain-specific benchmarking, e.g., CAD [53, 50], statistical network inference [46], and database [45].

**Benchmarking** Our use of deep learning models to compare up-to-date platforms, Google’s TPU v2/v3 and NVIDIA’s V100 GPU, distinguishes this work from previous cross-platform comparisons. Shi et al. compare CPU (Intel i7-3820 and E5-2630v3) and GPU (GTX 980, GTX 1080, and K80) platforms and deep learning frameworks [48]. Bahrampour et al. compare deep learning frameworks [5]. Others compare cloud computing providers [34], heterogeneous platforms [10], and cloud support for HPC [22].

## 9. CONCLUSION

This paper provides a comprehensive benchmarking analysis of deep neural network training hardware and software, and valuable lessons learned for future system designs. We present architectural bottlenecks of the TPU platform and provide suggestions for future improvement. Using ParaDnn, our parameterized benchmark suite for end-to-end deep learning, along with six real-world models, we compare the hardware and software of the TPU, GPU, and CPU platforms. We

present several new observations and insights into the design of specialized hardware and software for deep learning and motivate the need for further work in this field.

## 10. ACKNOWLEDGEMENT

This work was supported in part by Google’s TensorFlow Research Cloud (TFRC) program, NSF Grant # CCF-1533737, 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.

## 11. REFERENCES

- [1] “TensorFlow: Using JIT compilation <https://www.tensorflow.org/xla/jit>,” 2018.
- [2] “<https://cloud.google.com/tpu/docs/system-architecture>,” *Google Cloud Documentation*, 2018.
- [3] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, “Fathom: Reference workloads for modern deep learning methods,” in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–10.
- [4] D. Amodei, S. Anantharayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, “Deep speech 2: End-to-end speech recognition in english and mandarin,” in *International Conference on Machine Learning*, 2016, pp. 173–182.
- [5] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, “Comparative study of Caffe, Neon, Theano, and Torch for deep learning,” in *ICLR*, 2016.
- [6] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, “Scalable methods for 8-bit training of neural networks,” *arXiv preprint arXiv:1805.11046*, 2018.
- [7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.
- [8] G. A. Blog, “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.
- [9] L. Case, “Volta Tensor Core GPU achieves new AI performance milestones,” *Nvidia Developer Blog*, 2018.
- [10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.
- [11] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “TVM: End-to-end optimization stack for deep learning,” *arXiv preprint arXiv:1802.04799*, 2018.
- [12] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, “BenchNN: On the broad potential application scope of hardware neural network accelerators,” in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 36–45.
- [13] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, “DAWNBench: An end-to-end deep learning benchmark and competition,” *Training*, vol. 100, no. 101, p. 102, 2017.
- [14] J. Dean, “Recent advances in artificial intelligence and the implications for computer system design,” *Hot Chips*, 2017.
- [15] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, “Large scale distributed deep networks,” in *Advances in neural information processing systems*, 2012, pp. 1223–1231.
- [16] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John, “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*. IEEE, 2010, pp. 19–28.
- [17] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.
- [18] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, “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*. IEEE, 2015, pp. 27–40.
- [19] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang *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)*, vol. 50, no. 4. ACM, 2015, pp. 223–238.
- [20] 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 *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 620–629.
- [21] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778.
- [22] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, “Case study for running HPC applications in public clouds,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 395–401.
- [23] J. Hennessy and D. Patterson, “A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced.”
- [24] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [25] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
- [26] R. V. Hogg, J. McKean, and A. T. Craig, *Introduction to mathematical statistics*. Pearson Education, 2005.
- [27] 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,” *arXiv preprint arXiv:1704.04861*, 2017.
- [28] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *CVPR*, vol. 1, no. 2, 2017, p. 3.
- [29] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size,” *arXiv preprint arXiv:1602.07360*, 2016.
- [30] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” *arXiv preprint arXiv:1807.05358*, 2018.
- [31] A. Joshi, L. Eeckhout, and L. John, “The return of synthetic benchmarks,” in *2008 SPEC Benchmark Workshop*, 2008, pp. 1–11.
- [32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.
- [33] K. Kim, C. Lee, J. H. Jung, and W. W. Ro, “Workload synthesis: Generating benchmark workloads from statistical execution profile,” in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 120–129.
- [34] K. Kothari, “Comparison of several cloud computing providers,” *Elixir*

- [35] H.-T. Kung, “Why systolic architectures?” *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.
- [36] C. Leary and T. Wang, “XLA: TensorFlow, compiled,” *TensorFlow Dev Summit*, 2017.
- [37] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” *arXiv preprint arXiv:1708.02002*, 2017.
- [38] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, “Deep gradient compression: Reducing the communication bandwidth for distributed training,” *arXiv preprint arXiv:1712.01887*, 2017.
- [39] J. Nickolls and W. J. Dally, “The GPU computing era,” *IEEE Micro*, vol. 30, no. 2, 2010.
- [40] D. Patterson, “For better or worse, benchmarks shape a field,” *Communications of the ACM*, vol. 55, 2012.
- [41] —, “MLPerf: SPEC for ML,” <https://rise.cs.berkeley.edu/blog/mlperf-spec-for-ml/>, 2018.
- [42] T. repository for TPU models, “<https://github.com/tensorflow/tpu/>,” *GitHub*, 2018.
- [43] B. Research, “Deepbench <https://github.com/baidu-research/DeepBench/>,” 2017.
- [44] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhubarov, J. Hegeman, R. Levenstein, B. Maher, S. Nadathur, J. Olesen *et al.*, “Glow: Graph lowering compiler techniques for neural networks,” *arXiv preprint arXiv:1805.00907*, 2018.
- [45] M. Saleem, Q. Mehmood, and A.-C. N. Ngomo, “Feasible: A feature-based sparql benchmark generation framework,” in *International Semantic Web Conference*. Springer, 2015, pp. 52–69.
- [46] T. Schaffter, D. Marbach, and D. Floreano, “GeneNetWeaver: in silico benchmark generation and performance profiling of network inference methods,” *Bioinformatics*, vol. 27, no. 16, pp. 2263–2270, 2011.
- [47] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, “Measuring the effects of data parallelism on neural network training,” *arXiv preprint arXiv:1811.03600*, 2018.
- [48] S. Shi, Q. Wang, P. Xu, and X. Chu, “Benchmarking state-of-the-art deep learning software tools,” in *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*. IEEE, 2016, pp. 99–104.
- [49] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, p. 354, 2017.
- [50] D. Stroobandt, P. Verplaetse, and J. Van Campenhout, “Generating synthetic benchmark circuits for evaluating CAD tools,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 9, pp. 1011–1022, 2000.
- [51] J.-H. Tao, Z.-D. Du, Q. Guo, H.-Y. Lan, L. Zhang, S.-Y. Zhou, L.-J. Xu, C. Liu, H.-F. Liu, S. Tang, W. Chen, S.-L. Liu, and Y.-J. Chen, “BenchIP: Benchmarking intelligence processors,” *Journal of Computer Science and Technology*, vol. 33, no. 1, pp. 1–23, 2018.
- [52] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, “CortexSuite: A synthetic brain benchmark suite,” in *IISWC*, 2014, pp. 76–79.
- [53] M. Turki, H. Mehrez, Z. Marrakchi, and M. Abid, “Towards synthetic benchmarks generator for CAD tool evaluation,” in *Ph. D. Research in Microelectronics and Electronics (PRIME), 2012 8th Conference on*. VDE, 2012, pp. 1–4.
- [54] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.
- [55] W. Wei, L. Xu, L. Jin, W. Zhang, and T. Zhang, “AI matrix-synthetic benchmarks for DNN,” *arXiv preprint arXiv:1812.00886*, 2018.
- [56] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [57] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, “Google’s neural machine translation system: Bridging the gap between human and