Scalability Prediction and Performance Optimization for Deep Learning Workloads on TPUs and Clusters

***Abstract*—Large-batch training study has motivated Deep Learning (DL) training on supercomputers since 2017. To efficiently use supercomputers, the DL and HPC communities now need to exploit better hardware, better performance models, and better scaling efficiency. In this paper, we offer solutions for these. In terms of hardware, we conduct performance optimization on Google's latest Tensor Processing Unit (TPU). The Stair-Pass scheme allows us to achieve 17 TFlops on a TPU board. We compare TPU and GPUs, and conclude that a TPU is equivalent in performance to a cluster with 52 NVIDIA K80 GPUs or 10 NVIDIA P100 GPUs. In terms of modeling, we use linear least squares regression to predict the scalalility of DL workloads with thousands of nodes. By using our method, users only need to run a few iterations to predict the performance of an unknown DL workload on large-scale clusters. Our model achieves 91.9% prediction accuracy. Finally, our efficient implementation allows us to scale on several state-of-the-art supercomputers such as Piz Daint, TACC Stampede 2, and NERSC Cori. We scaled to 3200 KNL nodes (217,600 cores) on TACC Stampede 2 and achieved 1.2 PFlops for ImageNet training with ResNet-50, which is the state-of-the-art performance.**

I. Introduction

Several breakthroughs were made for large-scale deep learning (DL) training in past few years. Researchers were able to scale the batch size of DL training to 1024 [1], 5120 [2], 8192 [3], and 32768 [4] without losing accuracy. Although DL training may take days or weeks, machine learning researchers often did not use large-scale parallelism on supercomputers before 2017. The reason is that a batch of 256 pictures (each 225-by-225 Pixels) can be efficiently processed by a single node with eight GPUs. After scaling the batch size to 32K, researchers can use thousands of CPUs [5] or GPUs [6]. Scaling matters in deep learning because faster training leads to faster DL model development. To efficiently use supercomputers, the DL and HPC communities now need to exploit better hardware, better performance models, and better scaling efficiency. In this paper, we propose a series of solutions to meet these demands.

In terms of hardware, currently GPUs are a popular option for DL training. This throughput-oriented hardware is a good match to this computationally intensive application. In this paper, we explore additional DL hardware options. We choose Tensor Processing Unit (TPU) in this paper. In May 2017, Google released TPUv1. However, the impact is limited because TPUv1 is focused on the DL inference phase, which is much cheaper than the DL training phase. The performance of TPUv1 is not better than latest GPUs (e.g. NVIDIA P100). Recently, Google released TPUv2 for both DL inference and training. TPUv2 (henceforth called TPU) is extremely powerful in both computation and bandwidth. Each TPU provides 180 TFlops computational power, and 64 GB memory with 2400 GB/s peak bandwidth. In the future, TPUs will be available on Google Cloud for public use (the price is on Google Cloud's homepage). Additionally, Google will release TPU Pod in the future. TPU Pod is probably the first DL supercomputer in the cloud. Each TPU Pod contains 64 TPUs connected by a high-speed network and provides 11.5 Petaflops performance. In this paper, we conduct a comprehensive performance study for TPU. Specifically, the Stair-Pass scheme (Section V-A) allows us to make full use of TPU's Matrix Unit, which is the key computational component. We achieved 17 TFlops for ImageNet training with ResNet-50, which is the state-of-the-art performance on a single node. We conduct a comprehensive comparison between a TPU and latest GPUs. Our conclusion is that one TPU is equivalent in performance to a GPU cluster with 52 NVIDIA K80 GPUs or 10 NVIDIA P100 GPUs.

In terms of performance modeling, we use the linear least squares regression method to predict the performance of a DL workload with thousands of nodes. With our method, users only need to run a few iterations on a single node and can predict the scaling and performance of an unknown DL workload. Our experiments show that our prediction can achieve 91.9% accuracy. This is useful for resource scheduling.

In terms of large-scale training, our implementation can efficiently scale to thousands of nodes. We use ImageNet training with ResNet-50 as the benchmark. On TACC Stampede 2, we scale to 3200 KNLs (217,600 cores) and achieved 1.2 PFlops, which is the state-of-the-art performance. On Piz Daint, we scaled to 512 P100 GPU nodes and achieved 600 TFlops. On NERSC Cori, we scale to 512 KNLs and achieved 161 TFlops.

Overall, we made three contributions:

- Performance Optimization and Analysis on Tensor Processing Unit (TPU) system.
- Accurate Scalability Prediction and Performance Modeling for Distributed DL training on thousands of nodes.
- Fast and Scalable DL Training Implementation for Scaling to 3200 Nodes (217,600 cores) on supercomputers.

II. Background

A. Parallel Deep Learning Training

Most of the deep learning applications include two parts: Training and Inference. In the Training phase, people use the optimization algorithm to generate the model based on the training data. In the Inference phase, people use the

model to make predictions on the test data. Currently, the Inference phase is based on single node or mobile devices like cellphones, which does not need supercomputing. The training phase, however, is extremely slow, which often takes days or weeks on a powerful GPU-based system. In this paper, we focus on fast and scalable DL training on supercomputers.

For parallelizing neural networks, there are two approaches: data parallelism and model parallelism. Since the DL models are not generally becoming larger, the model parallelism can only make full use of a few machines. For example, ResNet-50 model [7] is only 104MB, which is like a 5K-by-5K matrix. State-of-the-art large-scale DL approaches are focused on data parallelism [3] [5] [6] [8] [9]. In this paper, we study data-parallel synchronous SGD algorithm, which is used in popular DL frameworks like Amazon MXNET, Facebook Caffe2, Facebook PyTorch, Intel Caffe, and Uber Horovod. Each iteration of the algorithm includes the following steps:

- Take B data samples from memory.
- Compute gradients of weights based on B data points
- Update the weights: $W = W - \eta * \nabla W$
  - W: weights, ∇W: gradients, η: learning rate
  - We also used momentum and weight decay, which are ignored here for simplicity since they do not impact system performance.

Data-Parallel implementation on P nodes includes the following parts:

- Each node has a copy of the weights (W_i) and the gradients (∇W_i), $i \in \{1, 2, ..., P\}$.
- Each node has B/P data samples to compute its own gradients ∇W_i
- Communication: an all-reduce sum for the gradients on all the nodes ($\sum_{i=1}^{P} \nabla W_i$)
- Each node does $W_i = W_i - \eta/P * \sum_{i=1}^{P} \nabla W_i$ to update the weights

B. Tensor Processing Unit

In this section, we introduce Google's Tensor Processing Unit (TPU), which was used in our experiments. TPUv1 chip is focused on DNN inference, which was released in 2017 [10]. This paper is focused on DNN training, which uses TPUv2. TPUv2 was open to limited users via Google Cloud in February of 2018. This paper does not include any data on TPUv1.

TPU Chip. Figure 1 shows the architecture of a TPU chip. Each TPU chip includes two TPU cores. Inside each TPU core, an 128-by-128 MXU (Matrix Unit) is connected to the scalar/vector units. Scalar/vector units are only for 32-bit floating point operations. MXU supports 32-bit precision for accumulation and 16-bit precision for multiplication. It is worth noting that both the input and the output of MXU are 32-bit floats. TPU does not support double-precision floating point operations because 32-bit operations are enough for the accuracy of machine learning applications. The performance of each TPU chip is 45 TFlops for mixed-precision computation.

TPU. A TPU is made up of four TPU chips. Each TPU provides 180 Teraflops performance, 64 GB memory, and 2400 GB/s total peak bandwidth. Later this year, TPU will be open to the public. Figure 2 is the structure of a TPU.

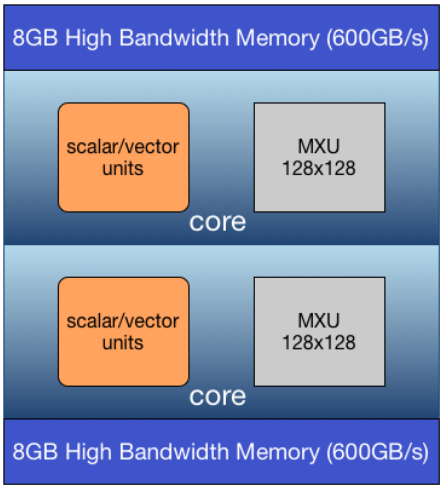

Fig. 1. The architecture of Tensor Processing Unit (TPU) chip. MXU means Matrix Unit. The performance of a TPU chip is 45 Tflops.

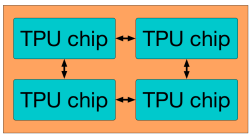

Fig. 2. Each Tensor Processing Unit (TPU) includes four TPU chips. The performance of a TPU is 180 Tflops.

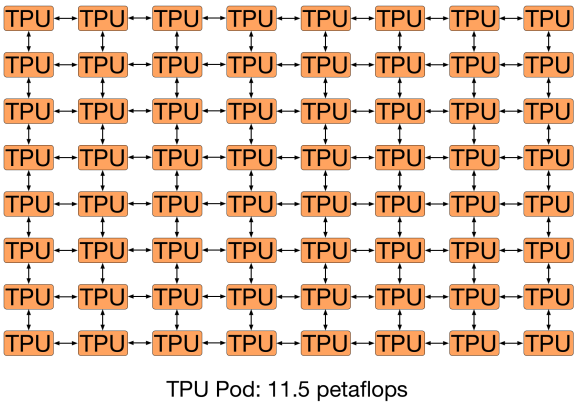

Fig. 3. Each TPU Pod is made up of 64 second generation TPUs. The performance of a TPU Pod is 11.5 Petaflops.

TPU Pod. TPU Pod is the first supercomputer for deep learning applications in the cloud (one can also run other applications on TPU Pod if the precision is enough). A TPU Pod can provide 11.5 Petaflops performance and 4 terabytes of high-bandwidth memory. TPU Pod will be open to the public later this year. Figure 3 shows the structure of a TPU Pod.

XLA. Like GPU's cuBlas and CPU's MKL, TPU provides Accelerated Linear Algebra (XLA) for users to build their own high-level implementations without rewriting the math kernel. XLA supports compiler, runtime, and TPU-specific

optimization. Together with the basic math operations, XLA also supports the basic DL operations like convolution.

III. Related Work

Researchers have investigated performance modeling and prediction in many contexts, such as supercomputer benchmarking [11], parallel programming [12], [13], grid computing [14], MPI collective communication [15], [16], data analytics [17]–[20], and DL [21]. Performance Modeling has two research directions: simulation-based approach [12], [14], [20] and analytic-based approach [13], [15]–[17], [19], [21]. We use the analytic-based approach. Our objective is to predict the run time and scaling efficiency given the information of a cluster, a DNN model and a dataset. The users only need to run a few iterations on the single node to predict the performance of thousands of nodes. To the best of our knowledge, this is the first such model for DL workloads. In the real-world systems, people may need to change the DNN model and dataset frequently. If we can predict the run time and scaling efficiency correctly, it will be helpful to online resource scheduling.

Shi et al. [22] did performance modeling for deep learning. However, they only scaled to 4 machines. Based on our experience, scaling to 4 machines is very different from scaling to 2,000+ machines. The current deep learning frameworks are well designed for running on less than 30 machines. Moreover, many system issues and scaling problems will be hidden in the small-scale experiments. Also, a good algorithm with lower complexity may not perform better than an algorithm with higher complexity. For example, an algorithm with $3logP$ complexity may perform worse than an algorithm with P complexity for $P = 4$. In terms of communication overhead, Shi et al. simplified the model by ignoring the influences of latency and bandwidth. Yan et al. [23] did performance modeling based on parameter server (i.e. asynchronous method), which has been demonstrated to be unstable in convergence and perform poorly on large-scale systems [24] [25]. The computational algorithm of our study is based on a synchronous method. Moreover, Yan et al. only used 20 machines in their study, which has the same problem as Shi et al. We use the QR least square approach and the gradient decent based regression approach, which is different from Shi et al and Yan et al. Our basic idea to predict the performance based on real-world data.

We comprehensively evaluate TPU on deep learning training. Existing work only shows the performance of TPUv1 in deep learning inference. Additionally, we conduct a comprehensive comparison study between TPUs and GPUs.

Using supercomputers to train deep neural networks has become popular since 2017. Goyal et al [3] scaled the batch size to 8K so that they could make full use of 256 NVIDIA P100 GPUs. The key idea is a learning rate warmup scheme. You et al [4] proposed LARS algorithms and scaled the batch size to 32K. By scaling the batch size to 32K, researchers are able to finish the ImageNet training in minutes [5]. There is other work [6] [8] [9] [26] in this direction: increasing the batch size. Before that, researchers scaled the batch size to 1K [1] and 5K [2]. The key idea is learning rate linear scaling.

This paper is built on top of linear scaling, warmup scheme and LARS, which make large-batch training possible. We have a comparison between our version and state-of-the-art implementation in Section VI-D. Existing GPU cluster work [3] [6] [8] is based on an 8-GPU-per-node platform. Our platform is based on One-GPU-per-node platform, which is harder to scale to the same number of GPUs. We need to scale to more nodes than previous implementations for using the same number of GPUs. For large-scale deep learning training, we think the engineering effort in implementation is nontrivial.

Kurth et al [27] scaled deep learning to 9K KNLs and achieved 15 PFlops peak performance. The key idea is to partition the nodes into different groups. Inside each group, the nodes communicate with each other by using a synchronous method. Different groups communicate with each other by using an asynchronous method. However, their proposed asynchronous-group algorithm can not converge for open deep learning benchmarks like ImageNet training with ResNet-50 [7] or GoogleNet [28]. They also modified their own models to reduce communication. In this paper, we use regular Sync SGD method, which requires full-gradient communication. We use the open benchmarks such ImageNet training with AlexNet [29] and ResNet-50 [7] to evaluate our methods.

IV. Modeling

As with most algorithms, the DL workload includes two parts: communication and computation. First, we evaluate the communication time, which is referred to as t_{comm}. Here the communication time is actually the total time, in seconds, spent in the network. We do not study the communication within a single node. The communication time includes two parts:

α or network latency is the time to send an empty message (zero bytes). For TCP system based on gigabit Ethernet, the latency is around 50μs per message, which is extremely inefficient. The more efficient and expensive networks such as Infiniband or Myrinet have the latency around 2μs per message.

β or the inverse of network bandwidth is the time to send each byte of the message (byte/s). For 500MB/s gigabit ethernet (4000Mb/s), it is 2ns per byte. The bandwidth of an 4x Infiniband is 1000MB/s, whose β is 1ns/byte. We can easily find that α is roughly 1000 times larger than β. Thus, minimizing latency overhead is an important goal.

As mentioned before, we use Sync SGD approach for large-scale DL training. The key communication part in Sync SGD is an all-reduce operation. There are three widely-used algorithms for all-reduce: butterfly, tree, and ring [30]. The tree algorithm is optimized for latency. The ring algorithm is optimized for bandwidth. The butterfly algorithm is optimized for both of them. The butterfly algorithm is implemented with a recursive halving reduce-scatter followed by a recursive doubling all-gather. The complexities of these three algorithms are described in Equations (1), (2), and (3) where M_s is

the message size. In our experiments, we found Butterfly algorithm performs better than Ring and Tree algorithms for our DL workloads on several supercomputers (e.g. NERSC Cori2, Piz Daint, TACC Stampede2). We choose Butterfly algorithm in our implementation.

$$t_{tree} = 2(\alpha logP + \beta M_s logP) \quad (1)$$

$$t_{ring} = 2(\alpha P + \frac{P-1}{P}\beta M_s) \quad (2)$$

$$t_{butterfly} = 2(\alpha logP + \frac{P-1}{P}\beta M_s) \quad (3)$$

After getting the communication time, we evaluate the computation time and I/O time. Since the data usually can be stored in local disk and used multiple times, the I/O time is trivial. In our experiments, we observe that the I/O overhead is less than 1% of the total time so we omit it from our model. In the naive Sync SGD without overlapping, the single iteration time can be modeled by:

$$t = t_{comp} + t_{comm} \quad (4)$$

The overlap between backward pass and the all-reduce operation is an important optimization for large-scale training. In this situation, the single iteration time can be modeled by:

$$t = \theta_1 \times t_{comp} + \theta_2 \times t_{comm} \quad (5)$$

where $\theta_1, \theta_2 \in [0, 1]$. The minimum single-iteration time is

$$t_{min} = max[t_{comp}, t_{comm}] \quad (6)$$

The maximum single-iteration time is the same as the non-overlap case. When t_{comp} and t_{comm} are close to each other, the overlap optimization can get a maximum speedup of 2. We think the communication modeling equation should include the number of layers, model size, and latency.

Let us assume the neural network has L layers. The size of l-th layer's gradient is G_l (bytes). The size of the DNN model (M bytes) is equal to the sum of all layers' gradient sizes. Then the communication time can be modeled by:

$$t_{comm} = \sum_{l=1}^{L} 2(\alpha logP + \beta G_l) = 2\alpha L logP + 2\beta M \quad (7)$$

We model t_{comp} as the single-node computation time at each iteration. The number of floating point operations required for each sample depends on the given DNN model, the dataset, and the framework's implementation. Let us use F to denote the number of operations required to process one sample. Let us use B to denote the batch size, which means the number of samples processed by one machine at each iteration. It is worth noting that the overall algorithm batch size is $P \times B$. Let us use S to denote the size of each sample. We are able to know S once we have the dataset. We can calculate F before running the code if we know the framework's implementation. However, now many frameworks have dynamic optimization at runtime. For example, Intel Caffe may drop some batch normalization computations and Tensorflow may choose different algorithms for back propagation implementation. Moreover, the processor's actual performance (flops) varies for different neural networks.

On the other hand, the chip makers and software builders generally release the measured performance for the commonly-used DNN models on the popular architectures (e.g. Intel KNL and NVIDIA P100). They use images processed per second as the performance metric. In this paper, we use I to denote the images processed per second on one node. Thus, the computational time at each iteration can be expressed as:

$$t_{comp} = B/I \quad (8)$$

After getting the single-iteration communication time and computation time, we focus on the overall running time. As the traditional machine learning benchmark, we use accuracy to measure the correctness of deep learning. An implementation is considered correct if it can achieve the same test accuracy as the baseline by running the same number of epochs. Here, we fix the number of epochs in all the accuracy comparisons and performance modeling. Statistically, one epoch means the algorithm processes the entire dataset once. For example, if the dataset totally has N samples and the batch size is B, then one epoch means $N/(PB)$ iterations. Computationally, fixing the number of epochs means fixing the number of floating point operations. For example, if we use the GEMM BLAS implementation, the algorithm needs to finish 7.72 billion single-precision operations for processing one 225-by-225 pixel picture. The ImageNet dataset has 1.28 million pictures. Thus, finishing 90-epoch training means the algorithm needs to process 1.28 million $\times$ 7.72 billion $\times$ 90 $\approx 10^{18}$ operations. Here, let use denote E as the number of epochs. The number of iterations becomes EN/PB. In total, the total training time can be modeled by:

$$t = EN/(PB) \times (\theta_1 \times t_{comp} + \theta_2 \times t_{comm}) \quad (9)$$

We introduce another term θ_0 to capture all the other overheads (e.g. I/O and runtime rescheduling). If we insert t_{comp} and t_{comm} to Equation (9), the total time becomes

$$t = \theta_0 + \theta_1 EN/(PI) + 2\theta_2 EN(\alpha L logP + \beta M)/(PB) \quad (10)$$

We define

$$x_1 = EN/(PI), x_2 = 2EN(\alpha L logP + \beta M)/(PB) \quad (11)$$

yielding the performance model

$$t = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \quad (12)$$

We use this to predict the run-time of a large run as follows. We collect data (x_1^i, x_2^i, t^i) for a number of real-system runs ($i = 1$ to s), where t^i is the measured total time of run i, and x_1^i and x_2^i are computed using Equation (11). The collected data

is shown in Table II and Table IV, which can be transformed to Table III and Table V, respectively. It is worth noting that the collection of data is an one-time cost. Unless the machine has a major upgrade, a well-trained model can be used for several months. Here I is the measured number of images processed per second on one node (measured by a separate small run). Then we do a linear least squares fit, choosing θ_0, θ_1 and θ_2 to minimize

$$\sum_{i=1}^{s}(t^i - (\theta_0 + \theta_1 x_1^i + \theta_2 x_2^i))^2 \quad (13)$$

This is a small problem that can be done directly using a QR factorization of the s-by-3 data matrix X, whose i-th row contains [1, x_1^i, x_2^i]. In this way, we obtain our trained model $\Theta = (\theta_0, \theta_1, \theta_2)$. Then we predict the time of a new run (with different values of P, M, L, N, B, and so different x_1 and x_2) using Equation (12).

For KNL cluster, we remove the 7th row of Table II from the training data and use it for testing. It can be transformed to <(1, 874.1, 8.695), 1144>. We will use $x = (1, 874.1, 8.695)^T$ to predict the run time. The model we got is $\Theta = (0.01, 0.93930354, 58.42522538)$. Our prediction by QR method is Θx = 1329s while the measured time is 1144s. The prediction accuracy is 1 - $|1144 - 1329|/1144$ = 83.8%. For CPU cluster, we remove the 3rd row of Table IV from the training data and use it for testing. The testing sample is 3rd row of Table IV, which can be transformed to <(1, 1807, 16.34), 3004>. We will use $x = (1, 1807, 16.34)^T$ to predict the run time. The model we got is $\Theta = (0.01, 1.65173808, 1.03560786)$. Then we predict the run time as Θx = 3001.6s. The actual measured run time is 3004s. The prediction accuracy is 1 - $|3004-3001.6|/3004$ = 99.93%. On average, the prediction accuracy is 91.9%. Considering we only have less than 20 training samples, we think the prediction result is good. If we have hundreds or thousands of training samples, we expect the prediction will be much more accurate.

A. Overhead of Prediction

To predict an unknown DL workload, the users only need to run a few iterations on a single node to get computational speed I. It is worth noting that when the system overlaps communication and computation, only parts of threads may be used in computation. Let us use $\hat{I}$ to denote the actual computational speed. To simplify the notation, let us introduce $\hat{\theta}_1 = I\theta_1/\hat{I}$ to denote actual parameter for computation. In the same way, we use $\hat{\theta}_2$ to denote the actual parameter for communication. Both $\hat{\theta}_1$ and $\hat{\theta}_2$ can be larger than 1. To avoid confusing the readers by introducing too many notations, we just use θ_1 and θ_2 to denote $\hat{\theta}_1$ and $\hat{\theta}_2$, respectively. In our model, there is another reason why we use I rather than $\hat{I}$. Let us assume we want to predict the performance of 1000 nodes. I can be measured on a single node. The users only need to run several iterations on a single node to collect I. If we use $\hat{I}$ in our model, the users will need to collect $\hat{I}$ on 1000 nodes.

TABLE I
THE PREDICTION ACCURACY OF PERFORMANCE MODELING BY QR METHOD AND GRADIENT DESCENT.

Measured Time	Gradient Descent Prediction	QR Prediction
1144s	969.4s (84.7%)	1329s (83.8%)
3004s	3069s (97.8%)	3001.6s (99.9%)

The target of our model is to predict the performance with minimal overhead.

B. Compared to Gradient Descent

Another possible way for solving the least square problem is to use gradient descent. The gradient descent method may work better if the data matrix X is extremely large (e.g. out of memory). In the future, we plan to improve our model by collecting more data samples. X maybe larger. However, even we collect millions of samples, s-by-3 data matrix X is still small. For example, if s is 10^6, we need to train our model based on a 12MB matrix, which can be efficiently processed by QR method. In this paper, the average accuracy of Gradient Descent is 91.3%, which is lower than the average accuracy of QR method (Table I).

C. More training samples

In the future, if we have more training samples, we may need to preprocess or normalize the data. In regular machine learning application, the users often need to preprocess the data to avoid feature imbalance (e.g. x_1 is 100 while x_2 is 0.01). Table III shows that we totally have $s = 16$ pairs of training samples. We denote the mean of these 16 samples' x_1 as μ_1 and the standard deviation of them as σ_1. Then we process x_1 as $\hat{x}_1 = (x_1 - \mu_1)/\sigma_1$. Let us denote $\hat{x}_1^i$ as the $\hat{x}_1$ of the i-th training sample. To simplify the notation, we use $\hat{X}_1 = (\hat{x}_1^1, \hat{x}_1^2, ..., \hat{x}_1^s)$ to denote a vector that includes all the training samples' $\hat{x}_1$. By preprocessing the data, we can make sure $\hat{X}_1$'s mean is zero and $\hat{X}_1$'s standard deviation is one. The zero-mean and one-deviation properties can help gradient descent to converge to the global minimum [31]. We normalize x_2 in the same way as x_1. The normalization can change the condition number of X, which might affect how an iterative method converges. If the condition number is large, it may also affect how well QR works. In this paper, however, the data normalization does not improve the accuracy of prediction because our training dataset is not large. Thus, we did not mention data normalization previously.

V. OPTIMIZATION ON TPU

A. High Performance MXU Kernel

Let us use ImageNet training with AlexNet by Caffe as an example. All of the layers that start with fc (for fully-connected) or conv (for convolution) are implemented using GEMM (GEneral Matrix Matrix Multiplication), and most of the run time (95% of the GPU version, and 89% on CPU) is spent on these layers. Thus, high-performance GEMM will be a key for deep learning frameworks and architectures.

TABLE II
IMAGENET TRAINING ON TACC STAMPEDE 2 INTEL KNIGHTS LANDING CLUSTER

ID	name	time	epochs	latency (α)	bandwidth (β^{-1})	nodes (P)	model (M)	layers (L)	samples (N)	speed (I)	batch (B)
1	ResNet-50	7016s	90	2.98μs	12.5GB/s	256	104MB	50	1.28M	70.65	128
2	ResNet-50	3617s	90	2.98μs	12.5GB/s	512	104MB	50	1.28M	71.92	64
3	ResNet-50	8454s	90	2.98μs	12.5GB/s	512	104MB	50	1.28M	54.40	16
4	ResNet-50	3617s	90	2.98μs	12.5GB/s	512	104MB	50	1.28M	71.92	64
5	ResNet-50	1972s	90	2.98μs	12.5GB/s	1024	104MB	50	1.28M	64.41	32
6	ResNet-50	1216s	90	2.98μs	12.5GB/s	2048	104MB	50	1.28M	54.40	16
7	ResNet-50	1144s	90	2.98μs	12.5GB/s	2048	104MB	50	1.28M	64.41	32
8	ResNet-50	1223s	90	2.98μs	12.5GB/s	2048	104MB	50	1.28M	54.40	16
9	ResNet-50	1497s	90	2.98μs	12.5GB/s	3200	104MB	50	1.28M	46.26	10
10	ResNet-50	915s	90	2.98μs	12.5GB/s	3200	104MB	50	1.28M	59.11	20
11	AlexNet	5884s	100	2.98μs	12.5GB/s	128	244MB	8	1.28M	517.6	64
12	AlexNet	2854s	100	2.98μs	12.5GB/s	128	244MB	8	1.28M	592.0	128
13	AlexNet	2244s	100	2.98μs	12.5GB/s	128	244MB	8	1.28M	694.7	256
14	AlexNet	1650s	100	2.98μs	12.5GB/s	256	244MB	8	1.28M	592.0	128
15	AlexNet	1461s	100	2.98μs	12.5GB/s	512	244MB	8	1.28M	517.6	64
16	AlexNet	849s	100	2.98μs	12.5GB/s	512	244MB	8	1.28M	592.0	128
17	AlexNet	705s	100	2.98μs	12.5GB/s	512	244MB	8	1.28M	694.7	256

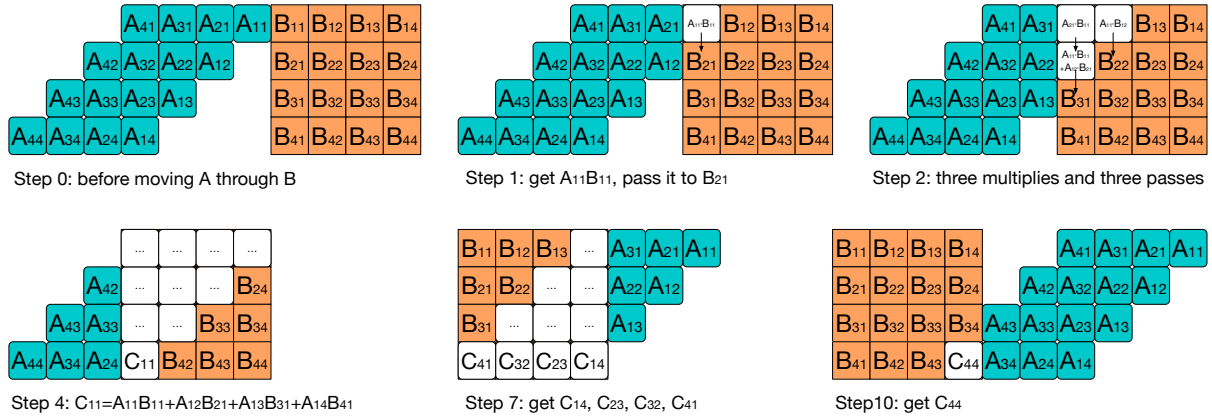

Fig. 4. An illustration of Stair-Pass scheme for the matrix-matrix multiply ($C = A \times B$). If both A and B are n-by-n matrices, this scheme needs $3n-2$ steps. In this example, $n = 4$.

The major difference between TPU and the current many-core architectures is the Matrix Unit (MXU). To make full of TPU, we need to maximize the Matrix Unit's performance. In our experiments, we use the Stair Pass scheme to optimize GEMM. The Stair Pass scheme was inspired by Cannon's algorithm [32] and systolic arrays [33]. Let us assume the GEMM is $A \times B = C$. Both A and B are n-by-n matrices. The illustration is shown in Figure 4. B is the weight matrix. We put B to the MXU. A is n-sample data matrix (the batch size is n). Here, each sample is a vector with n elements. We pass A through B column-by-column (from left to right in Figure 4). Meanwhile, the intermediate results are passed row-by-row (from up to down in Figure 4). This process totally takes $3n-2$ steps.

Let us use the example of $n = 4$ in Figure 4 to illustrate the idea. Table VI records the details of each step.

- At step 0, we put matrix B to the MXU. We use a stair-like structure to illustrate the dataflow of A.
- At step 1, A_{11} and B_{11} conduct dot product, the system passes the intermediate result $A_{11}B_{11}$ to position (2,1).
- At step 2, $A_{11}B_{12}$ is passed to position (2,2), $A_{21}B_{11}$ is passed to position (2,1), $(A_{11}B_{11}+A_{12}B_{21})$ is passed to position (2,1).
- At step 4, the system finishes the computation of $C_{11} = A_{11}B_{11}+A_{12}B_{21}+A_{13}B_{31}+A_{14}B_{41}$. A_{11} has finished all its computations and is going to leave MXU.
- At step 7, the system finishes the computations of C_{41}, C_{32}, C_{23}, and C_{14}.
- At step 10, the system finishes the computations for the last element of matrix C. A_{44} is going to leave MXU.

B. Performance for Deep Learning Workload

We use ImageNet training with ResNet-50 [7] as the benchmark. ImageNet training with ResNet-50 has become the gold benchmark for fast deep learning training [3] [5] [8]. Firstly, we need to study the training in algorithm level. We want to know the memory requirement and the flops computed at each iteration. We need to store three things at runtime in DNN training: (1) the input data; (2) the model; and (3) the activations that passes through the neural network. Let

TABLE III

THIS TABLE SHOWS THE PREPROCESSING DATA ON KNL CLUSTER. t IS THE MEASURED OVERALL TIME. x_1 IS THE OPTIMAL COMPUTATIONAL TIME BASED ON THE MEASURED SINGLE-NODE PERFORMANCE. x_2 IS THE OPTIMAL COMMUNICATION TIME. AFTER FEATURE SCALING, $\hat{x}_1$ IS THE NORMALIZATION OF x_1, AND $\hat{x}_2$ IS THE NORMALIZATION OF x_2. THE 7TH ROW WAS PICKED OUT FOR TESTING.

ID	t	x_1	x_2	$\hat{x}_1$	$\hat{x}_2$
1	7016	6375	14.24	2.86681245	-0.44001824
2	3617	3131	15.29	0.82412111	-0.37816131
3	8454	4140	61.17	1.45915714	2.32818306
4	3617	3131	15.29	0.82412111	-0.37816131
5	1972	1748	16.34	-0.04682672	-0.31630438
6	1216	1035	17.39	-0.49597092	-0.25444746
7	1144	874.1	8.695	-0.59754408	-0.767256637
8	1223	1035	17.39	-0.49597092	-0.25444746
9	1497	779	18.50	-0.65719362	-0.18904476
10	915	610	9.249	-0.76382115	-0.73466043
11	5884	1934	66.28	0.07000782	2.62926624
12	2854	1691	33.14	-0.08302596	0.67449507
13	2244	1441	16.57	-0.24041764	-0.30289052
14	1650	845.3	16.76	-0.61535312	-0.29189373
15	1461	483.4	16.94	-0.84325826	-0.28089694
16	849	422.7	8.471	-0.8815167	-0.78058652
17	705	360.2	4.235	-0.92086462	-1.03043131

us assume we use a batch size of 32. The size of the input data is 32×225×225×3×4B = 18.5 MB for 32 different 225-by-225 pictures with 3 channels. The model of ResNet-50 has 26 million parameters. Since all the current deep learning frameworks use 32-bit precision, the model size is 104 MB. In DNN training, the activations of the forward propagation must be stored because they will be used in backward propagation to compute the gradient loss. As for the ResNet-50 model, the 26 million parameters will need to store 16 million activations, which costs 64 MB storage.

The modern architectures (e.g. CPU, GPU, and TPU) often need to lay the data as dense vectors so as to make full the power SIMD units. If the batch size is 32 and the implementation uses 32-bit floating point operation, then the system can effectively use a 1024-bit vector by SIMD units. However, this SIMD parallelism also requires the system to copy the activations 32 times, which increases the local memory storage to 2GB. On the other hand, it is not efficient to directly execute the small convolutions used in deep neural networks. In most of the frameworks, the small convolutions are transformed to matrix-matrix multiply by the **lowering** operations [34]. The **lowering** operations improve the performance of DNN training. However, they also lead to multiple copies of weights or activations in certain layers. In total, the activations of ResNet-50 costs 7.5GB of memory when batch size is 32. In addition to the 3.25GB memory for the model, the total memory requirement is 10.75GB. Considering the read and write memory operations, the system needs to move roughly 21.5GB data (5.4 billion words) at each iteration. On the other hand, using ResNet-50 to process a 225-by-225 pixel image needs to finish 7.72 billion operations. The total number of operations for a batch size of 32 is 32×7.72 billion operations. Let us define computational intensity [35] as the ratio between the number of floating point operations and the number of words transferred. The computational intensity of ResNet-50 is 45.7.

To evaluate the architecture, let us define the system balance as the ratio between peak flops performance and peak memory bandwidth. For TPU, the system balance is the ratio between 180 trillion flops per second and 600 billion words per second (2400 GB/s), which is 307. Since 307 is much larger than 45.7, the ResNet-50 training on TPU will be memory bound. It means the algorithm does not have enough computational operations to feed TPU. The performance of this training process will depend on TPU's bandwidth. Since TPU can transfer 600 billion words to the processors per second, the algorithm will have 26.8 trillion operations to feed TPU at each second. Thus, the TPU's theoretical performance for training ResNet-50 should be 26.8 TFlops. Our implementation achieves the performance of 2253.05 images per second, which is 17 TFlops.

Based on our measured data, we believe our performance is good. The reason is that the measured hardware performance is lower than the theoretical number. Let us assume the measured bandwidth achieves 70% of the peak bandwidth and the measured performance achieves 80% of the peak performance. In this situation, TPU's system balance is 351. The algorithm will be still memory bound on TPU. Using this data, our model of TPU performance for training ResNet-50 should be 18.8 TFlops. We achieved 17 TFlops.

C. TPU versus GPU

We compare the performance of the TPU and GPUs for deep-learning workloads. The details for reproducing the results of this section can be found in the Appendix, which also includes the system and hardware information. We use ImageNet training with ResNet-50 as the benchmark to evaluate performance.

1) one node with multiple GPUs: Our first comparison is between a TPU and a NVIDIA DGX-1 station[1]. A NVIDIA DGX-1 station is powered by eight P100 GPUs connected by NVLink. Based on our testing, a DGX-1 station achieves 1730 images per second or 13 TFlops. The measured performance for a single P100 GPU is 1.65 TFlops, which means we achieve 98% strong scaling efficiency from one GPU to eight GPUs (within one node). Our first observation is that **one TPU is equal to ten NVIDIA P100 GPUs** for ResNet-50 workload because each configuration runs at about 17 TFlops. Another practical scenario is a machine with eight NVIDIA Tesla K80 GPUs. In our experiments, one K80 GPU achieve 401 GFlops for Resnet-50 training. The whole system with eight K80 GPUs achieve 2.92 TFlops, which achieves 93% strong scaling efficiency. The GPUs are connected via a common PCI fabric, which allows for low-latency GPU to GPU communication. Since PCI fabric's performance is worse than NVLink, the K80 system achieves lower scaling efficiency than one DGX-1 station. However, even using PCI

[1]https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system

TABLE IV
IMAGENET TRAINING ON TACC STAMPEDE 2 INTEL SKYLAKE CPU CLUSTER

ID	name	time	epochs	latency (α)	bandwidth (β^{-1})	nodes (P)	model (M)	layers (L)	samples (N)	speed (I)	batch (B)
1	ResNet-50	5834	90	2.98μs	12.5GB/s	512	104MB	50	1.28M	63.72	64
2	ResNet-50	2960	90	2.98μs	12.5GB/s	1024	104MB	50	1.28M	63.72	64
3	ResNet-50	3004	90	2.98μs	12.5GB/s	1024	104MB	50	1.28M	62.33	32
4	ResNet-50	1975	90	2.98μs	12.5GB/s	1600	104MB	50	1.28M	59.35	20
5	ResNet-50	3219	90	2.98μs	12.5GB/s	1024	104MB	50	1.28M	57.42	16
6	ResNet-50	1863	90	2.98μs	12.5GB/s	1600	104MB	50	1.28M	52.16	10
7	AlexNet	2578	100	2.98μs	12.5GB/s	128	244MB	8	1.28M	757.4	256
8	AlexNet	1421	100	2.98μs	12.5GB/s	256	244MB	8	1.28M	673.0	128
9	AlexNet	912	100	2.98μs	12.5GB/s	512	244MB	8	1.28M	527.9	64
10	AlexNet	673	100	2.98μs	12.5GB/s	1024	244MB	8	1.28M	394.7	32
11	AlexNet	766	100	2.98μs	12.5GB/s	1600	244MB	8	1.28M	290.3	20

TABLE V
THIS TABLE SHOWS THE PREPROCESSING DATA ON SKYLAKE CPU CLUSTER. t IS THE MEASURED OVERALL TIME. x_1 IS THE OPTIMAL COMPUTATIONAL TIME BASED ON THE MEASURED SINGLE-NODE PERFORMANCE. x_2 IS THE OPTIMAL COMMUNICATION TIME. AFTER FEATURE SCALING, $\hat{x}_1$ IS THE NORMALIZATION OF x_1, AND $\hat{x}_2$ IS THE NORMALIZATION OF x_2. THE 3RD ROW WAS PICKED OUT FOR TESTING.

ID	t	x_1	x_2	$\hat{x}_1$	$\hat{x}_2$
1	5834	3534	15.29	2.4007627	-0.52736256
2	2960	1767	8.171	0.5027762	-1.45363178
3	3004	1807	16.34	0.545591236	-0.391468331
4	1975	1214	17.43	-0.09105853	-0.250063
5	3219	1961	32.68	0.71101923	1.73430864
6	1863	1382	34.85	0.0887124	2.0161526
7	2578	1322	16.57	0.0241448	-0.36141107
8	1421	743.6	16.76	-0.59653307	-0.33716617
9	912	474.0	16.94	-0.88610841	-0.31292127
10	673	317.0	17.13	-1.05475567	-0.28867636
11	766	275.8	17.66	-1.09895965	-0.21922903

TABLE VI
STEP INFORMATION OF STAIR-PASS ON MXU. AN ILLUSTRATION OF COMPUTING $A \times B = C$, WHICH IS SHOWN IN FIGURE 4. THE BEING COMPUTED ELEMENTS ARE INCLUDED IN $\{\}$. THE FINISHED ELEMENTS ARE INCLUDED IN $[]$.

Step	{Being Computed} & [Finished]
1	$\{C_{11}\}$ & []
2	$\{C_{11}, C_{12}, C_{21}\}$ & []
3	$\{C_{11}, C_{12}, C_{13}, C_{21}, C_{22}, C_{31}\}$ & []
4	$\{C_{12}, C_{13}, C_{14}, C_{21}, C_{22}, C_{23}, C_{31}, C_{32}, C_{41}\}$ & $[C_{11}]$
5	$\{C_{13}, C_{14}, C_{22}, C_{23}, C_{24}, C_{31}, C_{32}, C_{33}, C_{41}, C_{42}\}$ & $[C_{12}, C_{21}]$
6	$\{C_{14}, C_{23}, C_{24}, C_{32}, C_{33}, C_{34}, C_{41}, C_{42}, C_{43}\}$ & $[C_{13}, C_{22}, C_{31}]$
7	$\{C_{24}, C_{33}, C_{34}, C_{42}, C_{43}, C_{44}\}$ & $[C_{14}, C_{23}, C_{32}, C_{41}]$
8	$\{C_{34}, C_{43}, C_{44}\}$ & $[C_{24}, C_{33}, C_{42}]$
9	$\{C_{44}\}$ & $[C_{34}, C_{43}]$
10	$\{\}$ & $[C_{44}]$

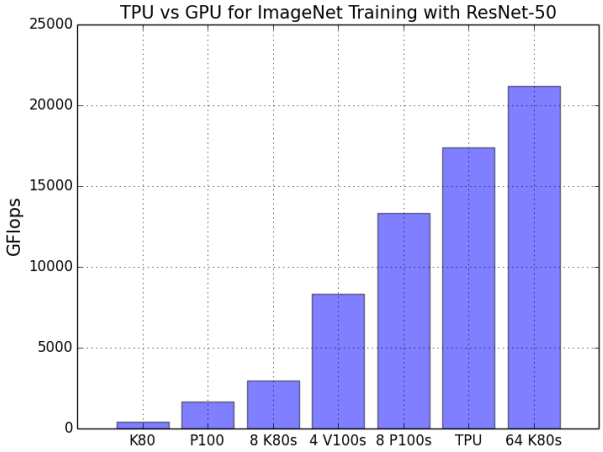

Fig. 5. K80 means NVIDIA Tesla K80 GPUs. P100 means NVIDIA P100 GPUs. V100 means NVIDIA V100 GPUs. TPU means Google TPUv2. The eight K80 GPUs are connected by PCIe. The eight P100 GPUs are connected by NVLink (i.e. NVIDIA DGX-1 station). The four V100 GPUs are connected by PCIe. The 64 K80 cluster includes eight machines connected by 5Gbps network.

fabric, the 93% scaling efficiency is still good enough. Our next observation is that **one TPU is is roughly equal to 48 NVIDIA K80 GPUs connected by high-speed network** for ResNet-50 workload. The last testing platform is a machine with four NVIDIA V100 GPUs connected by PCIe. The system achieves 8.16 TFlops for ResNet-50 training. It is worth noting that NVIDIA's official performance for V100 is much higher than our measurement. One important reason is that their GPUs are connected by NVLink. Our last observation is that **one TPU is equal to eight NVIDIA v100 GPUs connected by PCIe** for ResNet-50 workload.

2) multiple nodes with multiple GPUs on each node: In Section VI-B, we have the large-scale GPU cluster scaling results. In this section, we use a regular GPU cluster that is accessible to most users. We have a cluster that has eight machines connected by 5Gbps network. Each machine includes eight NVIDIA Tesla K80 GPUs connected by PCI fabric. This system in total has 64 Tesla K80 GPUs. For ResNet-50 training, a single K80 GPU achieves 401 GFlops and a single machine achieves 2.92 TFlops. This cluster achieves 20.69 TFlops, which is 88.6% strong scaling efficiency for eight nodes, or 80.8% strong scaling efficiency for 64 GPUs. For this comparison, our observation is that **one TPU is roughly equal to a regular cluster with 52 K80 GPUs connected by regular network** for ResNet-50 workload. We give a summary of the comparison in Figure 5.

VI. Experimental Results

A. Hardware

We use several state-of-the-art supercomputer systems to evaluate our implementations in this paper.

Piz Daint. It is located at Swiss National Supercomputing Centre. Currently it ranks 3rd on Top500 list. The LINPACK performance is 19.5 PFlops. Each node includes two Intel Xeon E5-2690v3 CPUs (2.6GHz, 64GB RAM, 24 cores) and one NVIDIA Tesla P100 GPUs (16GB). In total, there are 5320 nodes connected by Aries interconnect with Dragonfly topology.

NERSC Cori. It is located at Lawrence Berkeley National Lab. Currently it ranks 8th on Top500 list. The LINPACK performance is 14.0 PFlops. Each node is a single-socket Intel Xeon Phi Processor 7250 (Knights Landing) with 68 cores at 1.4 GHz. In total, there are 9688 nodes connected by Aries interconnect with Dragonfly topology.

TACC Stampede 2 (KNL). It is located at Texas Advanced Computing Center. Currently it ranks 12th on Top500 list. The LINPACK performance is 8.3 PFlops. Each node is a single-socket Intel Xeon Phi Processor 7250 (Knights Landing) with 68 cores at 1.4 GHz. In total, there are 4200 nodes connected by Intel Omni-Path with Fat Tree topology.

TACC Stampede 2 (CPU). It is located at Texas Advanced Computing Center. Since it has not released its LINPACK performance, currently it is not on Top500 ranking list. Each node is a double-socket Intel Xeon Platinum 8160 (Skylake) with 48 cores at 1.4-3.7 GHz. In total, there are 1736 nodes connected by Intel Omni-Path with Fat Tree topology.

Google Cloud. As introduced in Section II-B. TPU Pod is probably the first HPC cloud supercomputer for deep learning. It can provide 11.5 PFlops for deep learning workload.

Amazon HPC Cloud. We use Amazon HPC Cloud's GPUs for TPU-GPU comparison. Our systems include: (1) a DGX-1 station, which includes eight NVIDIA P100 GPUs connected by NVLink on one node; (2) a node with eight NVIDIA K80 GPUs connected by PCIe; (3) a node with four NVIDIA V100 GPUs connected by PCIe; and (4) an 8-node cluster connected by 5 Gbps network (each node has eight NVIDIA K80 GPUs connected by PCIe).

B. Large-Scale Scaling Efficiency

On TACC's Stampede 2 supercomputer, we successfully scaled to 3200 KNL nodes (217,600 cores) and achieved 1.2 PFlops performance for ResNet-50 training (Fig. 7). To the best of our knowledge, 1.2 PFlops is state-of-the-art performance for ResNet-50 training. We also scaled to 3200 Skylake CPUs (76,800 cores) and achieved 466 TFlops performance for ResNet-50 training. On Piz Daint, we scaled to 512 P100-GPU nodes and achieved 586 TFlops performance for ResNet-50 training. We stop at 512 nodes because scaling further will lead to lower efficiency. Figure 6 shows the strong scaling. We achieved 85% strong scaling efficiency from one node to 256 nodes and 64% strong scaling efficiency from one node to 512 nodes. Goyal et al. [3] reported a 90% scaling efficiency for 256 P100 GPUs. However, each of their nodes has eight GPUs and they actually only scale to 32 nodes. On NERSC Cori, we scale to 512 KNLs and achieved 161 TFlops. On a TPU Pod, 656 TFlops was achieved for ResNet-50 training.

TABLE VII
The relationship between batch size and performance. Skylake denotes double-socket Intel Xeon Platinum 8160 with 48 cores. KNL denotes Intel Xeon Phi 7250 with 68 cores. P100 means NVIDIA P100 GPUs.

Model	Batch Size	Skylake	KNL	P100 GPUs
AlexNet	20	435 GFlops	490 GFlops	123 GFlops
AlexNet	32	592 GFlops	616 GFlops	191 GFlops
AlexNet	64	792 GFlops	776 GFlops	375 GFlops
AlexNet	128	1009 GFlops	888 GFlops	736 GFlops
AlexNet	256	1136 GFlops	1042 GFlops	1372 GFlops
AlexNet	512	1145 GFlops	1081 GFlops	2412 GFlops
AlexNet	1024	1204 GFlops	1085 GFlops	3984 GFlops
ResNet50	10	403 GFlops	341 GFlops	812 GFlops
ResNet50	16	443 GFlops	419 GFlops	1210 GFlops
ResNet50	20	458 GFlops	431 GFlops	1253 GFlops
ResNet50	32	481 GFlops	504 GFlops	1543 GFlops
ResNet50	64	492 GFlops	547 GFlops	1682 GFlops
ResNet50	128	512 GFlops	534 GFlops	1795 GFlops
ResNet50	160	520 GFlops	527 GFlops	1808 GFlops
ResNet50	256	533 GFlops	520 GFlops	Crash

C. Batch Size and Performance

Figure 6 shows that larger batch sizes lead to higher scaling efficiency. The results of Figure 6 inspire us to study the relationship between batch size and performance. Based on our experiments, larger batch size leads to higher single-node performance for all the architectures (Table VII). Within one node, larger batch size (i.e. larger matrix) can improve the data usage and the efficiency of lower-level BLAS functions. Moreover, the number of epochs is fixed in deep learning training. So a larger batch size leads to fewer iterations. The number of node-to-node messages is equal to the number of iterations for Sync SGD. The size of each message is equal to the model size, which has no thing to do with the batch size. Thus, a larger batch size means sending fewer messages and moving less data. For example, the size of ResNet-50 model is 104MB. Running 90 epochs means we need to process 90×1.28 million = 115.2 million pictures. If the batch size is 32, we need to finish 3.6 million iterations, which requires 3.6 million messages and 374TB data for communication. If the batch size is 32768, we need to finish 3686 iterations, which requires 3686 messages and 374GB data for communication. Thus, increasing the batch size means reducing the communication overhead. At the same time, increasing the batch size does not change the number of floating point operations. Processing one picture requires 7.72 billion operations. Processing 115.2 million pictures requires 10^{18} operations. Larger batch size leads to a higher computation-communication ratio, which improves the scaling efficiency. The summary is in Table VIII.

D. Comparison to State-of-the-art Implementations

Goyal et al. [3] scaled ImageNet training with ResNet-50 to 256 NVIDIA P100 GPUs and finished the 90-epoch

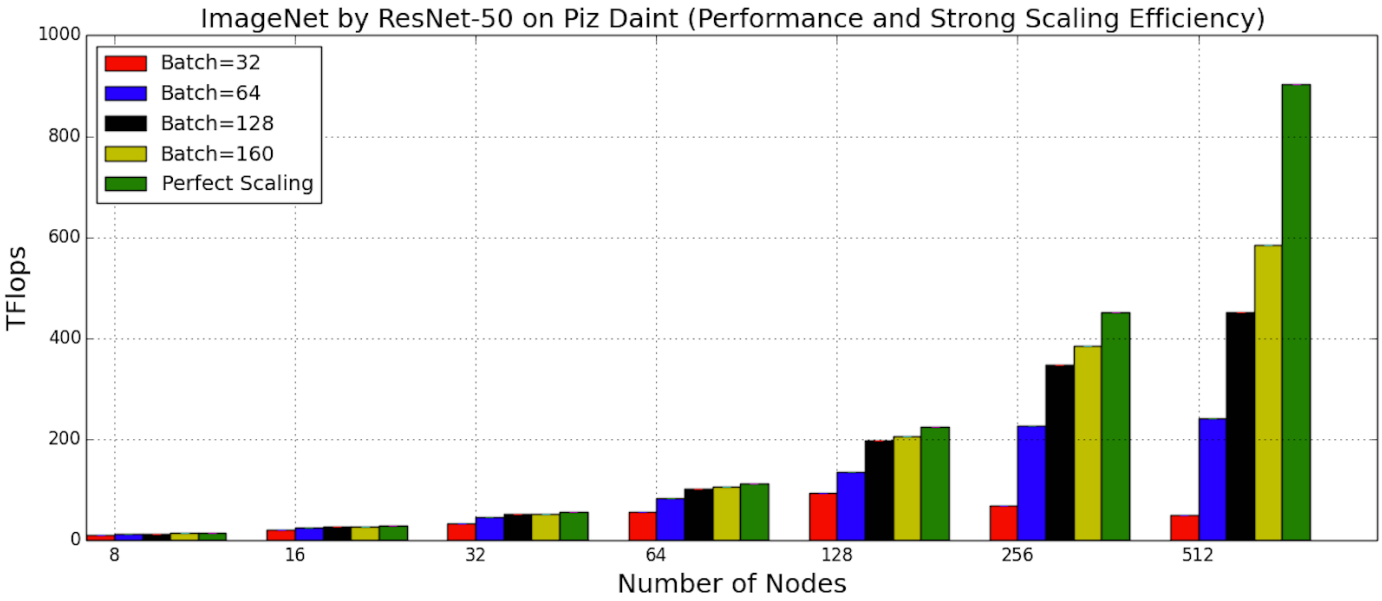

Fig. 6. We achieve decent strong scaling efficiency: 85% from one node to 256 nodes and 64% from one node to 512 nodes. This figure also shows that the batch size has a significant influence on performance and scaling.

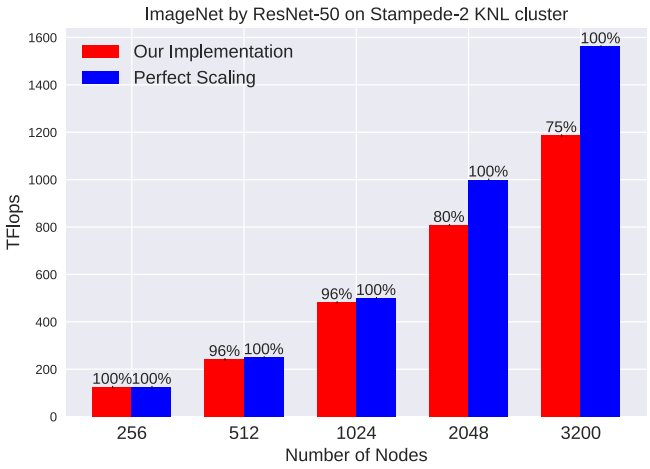

Fig. 7. We scale to 3200 KNL nodes on TACC Stampede 2 supercomputer and get 1.2 PFlops performance for ImageNet training with ResNet-50. As far as we know, 1.2 PFlops is state-of-the-art performance for ResNet-50 training.

TABLE VIII
LARGE BATCH SIZE IS GOOD FOR SCALING. 90-EPOCH IMAGENET TRAINING WITH RESNET-50. THE DATA MOVED MEANS THE VOLUME OF DATA MOVED OVER NETWORK.

Batch	Operations	# Messages	Data Moved	Comp/Comm
32	10^{18}	3.6 million	374TB	2673
32768	10^{18}	3686	374GB	2.7M

training in 65 minutes. By optimizing the communication, Cho et al. [8] were able to finish the training in 50 minutes. After efficient optimizations, we finished the same training on 256 NVIDIA P100 GPUs in 41 minutes. One key point in our implementation is that we optimized the online memory management, which allows us to process 128 pictures per GPU at each iteration. The implementations of Goyal et al. [3] and Cho et al. [8] only allow each GPU to process 32 pictures at each iteration.

VII. CONCLUSION

In this paper, we conduct performance modeling based on real-world performance data of various high performance implementations of ImageNet training with AlexNet and ResNet50. We formalize the performance modeling as a least squares problem. We use QR method to solve this problem and get the model. We then use the model to predict the run time of an unknown DL workload. Our prediction achieves 91.9% accuracy, which justifies the effectiveness of our analytic approach.

Based on our early experience on TPU, we conclude that TPU is an extremely powerful processor for deep learning application. For example, a TPU can achieve the same performance as ten NVIDIA P100 GPUs connected by high-speed NVLink. The Stair-Pass scheme helps us to make full use of the Matrix Unit of TPU.

We scale the deep learning algorithm on several state-of-the-art supercomputers like Piz Daint, NERSC Cori, and TACC Stampede 2. Our efficient implementation allows us to use large-batch approach, which helps us to achieve high scaling efficiency. Specifically, we achieved 1.2 PFlops for ImageNet training with ResNet-50 workload, which is state-of-the-art performance.

Our contribution is to provide better solutions (hardware, modeling, and scaling) to the DL and HPC communities.

VIII. Artifact Description Appendix

A. The Source Code

Most of our implementations been released in Intel Caffe (after version 1.0.7). The users can download the code from this link[2].

B. The dataset

First, due to the limit of file size, we can not upload the datasets. To run our code, the readers need to download the datasets. The readers may need to run some small-scale test cases before running the large-scale test case. To do so, the readers can download the Mnist and Cifar datasets. The readers need to download the ImageNet dataset for the large-scale test. For Mnist dataset, the readers can download it from this link[3]. For Cifar dataset, the readers can download it from this link[4]. For Imagenet dataset, the readers can download it from this link[5].

C. Dependent Libraries

For GPU codes, we use CUDA 8.0 and CuBLAS and CuDNN 6.0 libraries. We use Nvidia NCCL for GPU-to-GPU communication within a single node. We use MPI for distributed processing on the multi-GPU multi-node system. For KNL codes, the users can run Intel Caffe to reproduce our implementation. The Intel Caffe depends on Intel MKL for basic linear algebra functions. To install and use Intel Caffe, we install the follow libraries: (1) protobuf/2.6.1, (2) boost/1.55.0, (3) gflags/2.1.2, (4) glog/0.3.4, (5) snappy/1.1.3, (6) leveldb/1.18, (7) lmdb/0.9.18, and (8) opencv/3.1.0-nogui.

D. Running our codes

After downloading our codes from SC18 submission system and unzipping it, readers will need to enter the **scaling** folder. To reproduce the 3200 nodes results, the readers need to enter the **resnet50_3200nodes_32K** folder. Then the readers only need to execute the **run.slurm** script to reproduce the results on 3200 KNL nodes. The other cases can be reproduced in the same way. As mentioned before, our code has been released in Intel Caffe, the readers need to make sure they have successfully installed the distributed version of Intel Caffe before running the script. Additionally, the readers may need to slightly modify the script based on their own cluster management system.

E. Reproduce TPU-GPU comparison

Eight NVIDIA P100 GPUs. The Instance type on Amazon Cloud is NVIDIA DGX-1. The Operating System is Ubuntu 16.04 LTS. The Docker tool is used. The CUDA version is 8.0. The CUDNN version is 5.1. The disk is Local SSD.

Eight NVIDIA K80 GPUs. The Instance type on Amazon Cloud is n1-standard-32-k80x8. The Operating System is Ubuntu 16.04 LTS. The CUDA version is 8.0. The CUDNN version is 5.1. The disk is 1.7 TB Shared SSD persistent disk (800 MB/s).

A Cluster with 64 NVIDIA K80 GPUs. The Instance type on Amazon Cloud is p2.8xlarge. The Operating System is Ubuntu 16.04 LTS. The CUDA version is 8.0. The CUDNN version is 5.1. The disk is 1.0 TB EFS (burst 100 MB/sec for 12 hours, continuous 50 MB/sec).

[2]https://github.com/intel/caffe

[3]Mnist dataset is at http://yann.lecun.com/exdb/mnist

[4]Cifar dataset is at http://www.cs.toronto.edu/ kriz/cifar-10-binary.tar.gz

[5]Imagenet dataset is at http://image-net.org/download

References

[1] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[2] M. Li, "Scaling distributed machine learning with system and algorithm co-design," Ph.D. dissertation, Intel, 2017.

[3] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[4] Y. You, I. Gitman, and B. Ginsburg, "Scaling sgd batch size to 32k for imagenet training," *arXiv preprint arXiv:1708.03888*, 2017.

[5] Y. You, Z. Zhang, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Imagenet training in 24 minutes," *arXiv preprint arXiv:1709.05011*, 2017.

[6] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[7] 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*, 2016, pp. 770–778.

[8] M. Cho, U. Finkler, S. Kumar, D. Kung, V. Saxena, and D. Sreedhar, "Powerai ddl. arxiv preprint," *arXiv preprint arXiv:1708.02188*, 2017.

[9] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large minibatch sgd: Residual network training on imagenet-1k with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[10] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[11] J. J. Dongarra, "Performance of various computers using standard linear equations software," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 3, pp. 22–44, 1992.

[12] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé, "Simulation-based performance prediction for large parallel machines," *International Journal of Parallel Programming*, vol. 33, no. 2, pp. 183–207, 2005.

[13] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *ACM Sigplan Notices*, vol. 26, no. 7. ACM, 1991, pp. 213–223.

[14] R. Buyya and M. Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *Concurrency and computation: practice and experience*, vol. 14, no. 13-15, pp. 1175–1220, 2002.

[15] E. Chan, M. Heimlich, A. Purkayastha, and R. Van De Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[16] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[17] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[18] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[19] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[20] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 99–112.

[21] S. Shi and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," *arXiv preprint arXiv:1711.05979*, 2017.
[22] ——, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," *arXiv preprint arXiv:1711.05979*, 2017.
[23] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1355–1364.
[24] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," *arXiv preprint arXiv:1512.02595*, 2015.
[25] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.
[26] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on gpu and knights landing clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 9.
[27] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M. M. A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov *et al.*, "Deep learning at 15pf: supervised and semi-supervised classification for scientific data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 7.
[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich *et al.*, "Going deeper with convolutions."
[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
[30] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.
[31] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.
[32] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm." MONTANA STATE UNIV BOZEMAN ENGINEERING RESEARCH LABS, Tech. Rep., 1969.
[33] W. M. Gentleman and H. Kung, "Matrix triangularization by systolic arrays," in *Real-Time Signal Processing IV*, vol. 298. International Society for Optics and Photonics, 1982, pp. 19–27.
[34] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.
[35] 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.