IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 13, 2019, accepted May 8, 2019, date of publication May 14, 2019, date of current version May 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2916550

Iteration Time Prediction for CNN in Multi-GPU Platform: Modeling and Analysis

ZIQIAN PEI, CHENSHENG LI, XIAOWEI QIN, XIAOHUI CHEN, AND GUO WEI
CAS Key Laboratory of Wireless-Optical Communications, University of Science and Technology of China, Hefei 230026, China

Corresponding author: Xiaowei Qin (qinxw@ustc.edu.cn)

This work was supported in part by the National Science and Technology Major Project of China MIIT under Grant 2018ZX03001001-003 and in part by the Fundamental Research Funds for the Central Universities.

ABSTRACT Neural networks, as powerful models for many difficult learning tasks, have created an increasingly heavy computational burden. More and more researchers focus on how to optimize the training time, and one of the difficulties is to establish a general iteration time prediction model. However, the existing models have high complexity or tedious build processes, and there is still space for improvement in prediction accuracy. Moreover, there is little systematic analysis of multi-GPU which is a special and widely used scenario. In this paper, we introduce a framework to analyze the training time for convolutional neural networks (CNNs) on multi-GPU platforms. Based on the analysis of GPU calculation principles and its special transmission mode, our framework decomposes the model and obtain accurate prediction results without long-term training or complex data collection. We start by extracting key feature parameters related to GPUs, CNNs, and networks. Then, we map CNN architectures to constraints, including software platforms, GPU platforms, parallel strategies, and communication strategies. At last, we provide the prediction model and give analysis results of training time from multiple perspectives. The proposed model is verified on four types of NVIDIA GPU platforms and six different CNN architectures. The experiment results show that the average error across varies scenarios is less than 15% and outperform the state-of-the-art results by 5%–30%, which corroborate our model an effective tool for artificial intelligence (AI) researchers.

INDEX TERMS Convolutional neural network, multi-GPU parallel, iteration time.

I. INTRODUCTION

In recent years, deep neural networks (DNNs) have been widely applied in many fields, such as image recognition [1]–[3], speech recognition [4] and machine translation [5]. These applications always need large-scale training and computation, which means a large number of time, resources and capital consumption. At the same time, GPUs are widely used in accelerating DNNs for their high bandwidth and parallelism [6]. Before actually training the model, predicting the runtime of CNN on GPU platforms is an effective way to avoid several days or weeks of testing, which is important for people to understand how to design the hardware-optimal DNNs for deployment [35].

The factors that affect the iteration time of neural networks are complex, mainly including several aspects. Firstly, the structure of neural networks determines the amount of computation, which directly affects time consumption. Secondly, it is necessary to consider the choice of neural network construction and calculation tools (e.g., Tensorflow [10], Caffe [11], MXNet [12], NVIDIA accelerator library cuDNN [13]), whose own efficiency indirectly affects the iteration of training. Thirdly, the selection of training algorithms determines the training process of neural networks. For example, data parallel algorithm and model parallel algorithm [14] are quite different in terms of computing allocation and data transmission. Finally, the efficiency of deep learning models is determined by their hardware performance with respect to metrics such as runtime, not only by their accuracy for a given learning task [34]. Current mainstream computing chips includes GPU [7], FPGA [8], TPU and CPU [9]. More and more attention has been paid to the development of powerful hardware devices to shorten the training time of models. Since the training process of CNN is closely related to these aforesaid factors, it becomes more difficult to accurately predict and analyze the runtime in different scenarios.

At present, people mainly rely on public benchmarks or a large number of experiments [15] to obtain the runtime, which brings huge waste of time and resource. In order to solve this problem, many performance models

The associate editor coordinating the review of this manuscript and approving it for publication was Mu-Yen Chen.

emerged. Neural power [16] is a learning-based polynomial regression model, which establish the relationship between CNN parameters, physical operations and runtime. Learning-based method is common to all hardware or software platforms without knowing many internal details. However, the way it models each layer of CNN separately may lead to high complexity and tedious processes for deep networks like Resnet. In addition, the model is not suitable for multi-GPU training scenario due to the lack of description on communication strategies. Paleo [23] derives a performance model from operation counts alone, which determines the runtime of CNN on various platforms based on analytical method. Nevertheless its simple statistics and insufficient consideration on hardware parameters lead to deviations in the prediction result. Useing the measured time of GPU-based matrix computing library as a lookup table, SPRITN [22] is possible to predict CNN runtime with 5 %–19 % error on GPU clusters with asynchronous communication. The same is achieved for CPUs in a distributed environment [24], and [27] use a similar approach for Intel Xeon Phi accelerators. These works mainly pay attention to distributed clusters, which is not similar to single-machine environment especially in the mode of data transfer. In addition, some models focus on platform comparisons. For example, [25] builds performance models of standard processes in training DNNs with stochastic gradient descent (SGD), which is mainly used to compare the differences of communication strategies between mainstream software frameworks. As a relevant application scenario, multi-GPU analysis and modeling has been involved in these works to some extent, which has important enlightening and reference value for our in-depth work.

In this paper, we develop a time analysis framework for CNNs in multi-GPU scenario to predict its iteration time before training. Since single GPU scenario is a special case of multi-GPU scenario, our model is also applicable to it. We choose Tensorflow as our software platform for research, which has been widely used in both industry and academia. It can support the rapid implementation of neural networks on single-device, multi-device and distributed execution at the same time [10]. The framework is shown in Fig.1. By inputting CNN architecture, hardware feature and network feature sets, the framework builds models for computation and transmission respectively. The prediction results of different time granularity are given to support researchers' analysis from multiple perspectives. The main contributions of this paper are as follows:

- Design and implement the CNN iteration time prediction framework for multi-GPU scenario by combining the analysis model with the measurement model.
- Extract the key performance parameters of the NVIDIA GPUs and establish abstract instruction queue model which is universal for different GPU platforms.
- Build sigmoid distribution model of transfer data and real-time throughput for transmission time computing.
- Carry out experiments on four types of NVIDIA GPU platforms and six different CNN architectures. The accuracy is improved by 5%-30% compared with the state of the art.

The rest of the paper is organized as follows. Section II introduces the background knowledge about GPU architectures, CNN architectures and CNN training processes. Section III describes our time analysis framework in detail, including time division of one iteration, abstract instruction queue model, parallelism adjustment, throughput model and transmission model. Section IV shows the experiment results of the model on different kinds of CNNs and GPU platforms. Section V concludes the paper.

II. BACKGROUND

A. GPU ARCHITECTURES

Modern GPUs are designed for compute-intensive applications, such as CNN computations, and it is important to understand their general structures. Take the general design for NVIDIA GPUs as an example, a GPU is composed of a piece of global memory and many streaming multiprocessors (SMs). Each SM contains a variety of function units: streaming processors (SPs), double precision units (DPUs), special function units (SFUs), and load store units (LDSTs). The SM also has a piece of low latency shared memory, on which users can explicitly allocate and access data. On GPUs, warp is the minimum scheduling unit on GPUs which usually contains 32 threads, and an SM holds multiple blocks that maintain several warps [17]. The number of blocks resides on an SM is concurrently determined by the resource constraints such as available registers and shared memory.

B. CNN ARCHITECTURES

Convolutional neural networks is a kind of feedforward neural networks with deep structures and convolution calculations. The hidden layer of CNNs includes convolutional (CONV) layer, pooling (POOL) layer and fully connected (FC) layer. These layers are the basis components of some classical network structures such as Lenet [28], Alexnet [1] and VggNet [2]. In some more advanced neural networks, there may be complex structures such as inception module, residual block and batchnorm (BN) layer [31], which play an important role in GoogleNet [29], Resnet [3] and other well-known networks. Different layers and modules interact through direct connections while maintaining computational and functional independence. Therefore, different CNN structures have the same characteristics through layer-level or module-level division.

C. CNN TRAINING PROCESS

The training process of neural networks depends on the error back propagation algorithm, which involves the calculation and transmission of data. CNN's core computing operations on GPU rely on CUBLAS library [21], which is a matrix computing library of NVIDIA that supports various operations [6]. Parallel training [14], [32], [33] needs to be considered when there are multiple GPUs, and we only

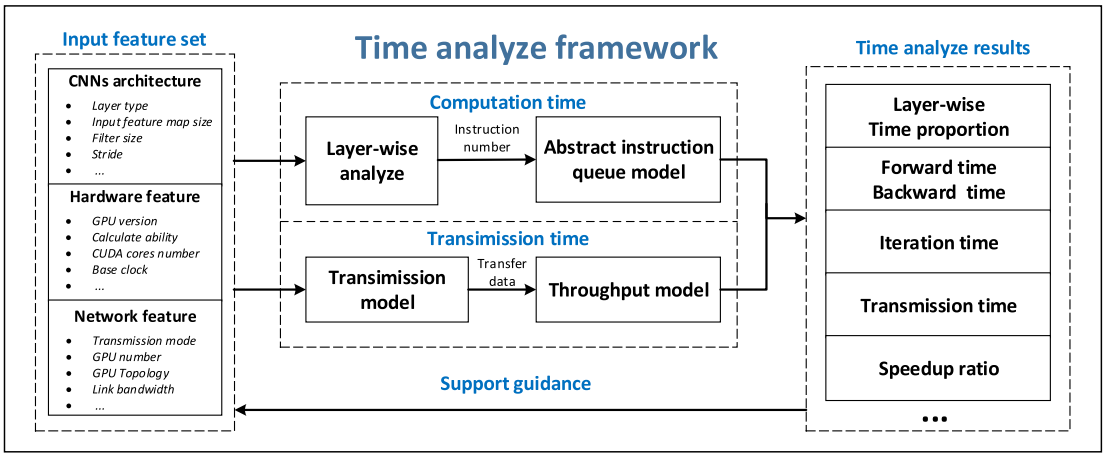


FIGURE 1. Overview of the time analyze framework. Three types of characteristic parameters are extracted to construct the model, and a variety of analysis results are output to support the guidance of neural network training.

discuss data parallel in this paper. Data parallel means that the training data is distributed on different computing devices, and each device has the same copy of the model. There are two common implementations of data transfer strategies during this process. One is parameter server (PS) mode [18], in which the CPU is usually used as a server node, causing a large amount of data interaction between GPU and CPU in the training process. The other mode supported by NVIDIA collective multi-GPU communication library (NCCL) [19] realizes parameter transfer and computing through AllReduceKernel function, which does not involve CPU and the transmission bottleneck depends on the slowest link. In addition, AllReduceKernel is executed in parallel with other CUDA kernels and makes use of Ring AllReduce algorithm [20] to form a data loop among GPUs to optimize the transfer process. But its occupation of GPU computing resources (i.e., SMs) may affect computing efficiency.

III. TIME ANALYZE FRAMEWORK

In this section, we introduce our time analyze framework, and the major steps are given in Algorithm 1. The input set contains three types of features: hardware, CNN architecture and network. The framework firstly parses the CNN architecture layer by layer and obtains the operands of forward and backward propagation through statistics. Then it extracts the key characteristic parameters of GPUs to construct the abstract instruction queue model and predict the calculation time of each layer (Line 1-4). Line 5 invokes Equation 2,3 to calculate the total computation time. The transmission model of framework supports two modes (PS and NCCL) in multi-GPU scenario. In particular, the calculation of exposed transfer time depends on the construction of throughput model, which describes the non-linear relationship between the size of transfer data and the real-time throughput. Finally, the iteration time of a CNN is obtained according to the dependence of computation and transmission (Line 8). Note that, single GPU is a special scenario of multi-GPU, and our model can also be applied to single GPU prediction by ignoring the transmission part. The notations used in the framework are shown in Table 1 and Table 3.

Algorithm 1 Iteration Time Analyze Framework Algorithm

Input: The set of hardware feature H_f;f The set of CNN architecture C_a; The set of network feature N_f;

Output: Analysis result of iteration time;

1: **for** each layer $l \in [1, N]$ **do**
2: $Operands_l \leftarrow Statistic(C_a)$;
3: $t_f, t_b \leftarrow Abstract\_instruction\_queue(Operands_l, H_f)$;
4: **end for**
5: ▶ *Equation* 2, 3
6: $Mode(PS\ or\ NCCL) \leftarrow N_f$;
7: $t_{update} \leftarrow Transmission\_model(C_a, N_f, Mode)$;
8: ▶ *Equation* 1
9: **return** t_{total}

A. TIME DIVISION OF ONE ITERATION

According to error back propagation algorithm, we define t_{total} as one iteration time of CNNs, and the formula can be expressed as:

$$t_{total} = t_{read} + t_{forward} + t_{backward} + t_{update} \tag{1}$$

Before computing, the GPU needs to read the corresponding training data from memory, and the non-overlapped I/O time equals to t_{read}. t_{update} on single GPU equals to the computation time of parameters update, besides, the data transmission time is also included on multiple GPUs. Detailed calculation process will be explained in Sec.III E. We split CNNs into CONV layer, FC layer, POOL layer, BN layer and residual block, where the FC layer can be regarded as a special CONV layer when the filters and input feature maps are of the same size [8]. Through layer-wise analyze, we can get the calculation formulas of forward and backward time by adding time of each layer :

$$t_{forward} = \sum_{l=1}^{N} t_f^l \tag{2}$$

TABLE 1. Summary of computation notations.

Name	Description
t_{total}	One iteration time of CNNs training process
$t_{forward}$	Time of forward propagation
$t_{backward}$	Time of backward propagation
t_{update}	Time of parameter update
t_f^l	Forward execution time of layer l
t_b^l	Backward execution time of layer l
t_k^i	Execution time of the i^{th} CUDA kernel function
C_k^i	Number of GPU clock cycles required by the i^{th} CUDA kernel function
t_{clock}	GPU's clock cycle
C_b	Number of clock cycles used in the execution of a block
N_{iter}	Iteration number of blocks
L_{GS}	Latency of Global store instruction
L_S	Latency of Shared load instruction
I	Number of Global store instructions in one block iteration
M	Number of Shared load instructions in one block iteration
K	Number of CP instructions in one block iteration
L	Number of Global store instructions in one block iteration
m	The number ratio of SP unit to LDST unit in one SM
$GL_{dispatch}$	Total number of Global load instructions contained in the CUDA kernel function
$S_{dispatch}$	Total number of Shared load instructions contained in the CUDA kernel function
$CP_{dispatch}$	Total number of CP instructions contained in the CUDA kernel function
$GS_{dispatch}$	Total number of Global store instructions contained in the CUDA kernel function
$block_{num}$	Number of blocks contained in the CUDA kernel function
sm_{num}	Number of SMs in the GPU
$sm.SP_{num}$	Number of SP units in an SM
$sm.LDST_{num}$	Number of LDST units in an SM

$$t_{backward} = \sum_{l=1}^{N} t_b^l \tag{3}$$

In essence, the computation on GPU is matrix operation between vectors, and the implementation is well-optimized by the CUDA library [21], which provides a variety of methods and data layouts. In order to assist users in choosing algorithms, such libraries provide functions that choose the best-performing algorithm under given tensor sizes and memory constraints. Internally, the library may run all methods and pick the fastest one [6]. For example, in the CONV layer, we mainly consider two typical algorithms: im2col [13] and fast fourier transform (FFT) [26]. The estimation of im2col is based on the well-optimized general matrix multiplication (GEMM) routines [13], [21]. The principle of this algorithm is to convert the convolution calculation into large matrix multiplications, and further convert the calculation into the multiply-add operation of multiple parallel small matrices which are more adapted to the GPU structure. Although this method improves the parallelism, it also causes duplicate transmissions of data. The use of shared memory can greatly improve the computational efficiency. The essence of FFT algorithm is to reduce the amount of calculations by using dot-multiply instead of multiply-add operation, but complex numbers take up more storage space and reduce computational efficiency. The operand estimation can be reasonably inferred by the theoretical results given in [26]. Since different matrix sizes are suitable for different implementations, our model selects faster one as output by comparing their prediction results. The statistics of other layer type operands can also be derived from their calculation formulas. According to the basic operation process and strict calculation order of matrix multiplication in CUDA manual, we regard the computation task as a serial execution process of multiple CUDA kernel functions which can be expressed as follows, where $K1$, $K2$ are the numbers of kernels:

$$t_f^l = \sum_{i=1}^{K1} t_{kf}^i \tag{4}$$

$$t_b^l = \sum_{i=1}^{K2} t_{kb}^i \tag{5}$$

After the above segmentation, the computation time is divided into the execution time of several fixed-size matrix operation functions, and each time granularity can be modeled separately.

B. ABSTRACT INSTRUCTION QUEUE MODEL

In order to build a general model applicable to different GPU platforms, we simplify the execution of kernel functions into four steps: reading data from the global memory, reading data from the shared memory, computing, and storing the data to the global memory. Corresponding, we define four types of abstract instructions to represent the above process, i.e., Global Load, Shared Load, CP and Global Store. The number of these instructions is counted by warps and limited by the number of hardware units, such as the SP unit. Here we assume that the internal units of GPU have the maximum utilization, there are no conflicts in the transmission process, and each instruction achieves the maximum bandwidth (128 bytes per instruction) [30]. Therefore, we convert the execution time of a kernel function to the count of corresponding instructions, and the clock cycle required for each instruction is fixed. The following equation can be obtained:

$$t_k^i = C_k^i * t_{clock} \tag{6}$$

According to the proportion of different instruction types, the abstract instruction queue model has two classifications: storage-intensive and computation-intensive. Storage-intensive means that the number of Shared Load instructions is much larger than that of CP, and Fig.2 shows the general execution pipeline that can be executed in one SM. In both models, the instruction queue is iterated by multiple identical blocks.

Rectangles in the figure represent different instructions. The length of rectangles represents the instruction latency. Table 2 gives some specific latency benchmarks of NVIDIA P40 as an example. Instructions are executed in warps, and each instruction queue is iterated in blocks until the end of the calculation. The instruction interval can be found in the

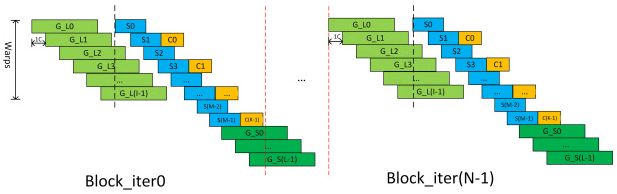


FIGURE 2. Storage-intensive instruction queue model. There are I Global Load instructions, M Shared Load instructions, K CP instructions and L Global Store instructions in each block iteration. The instruction are executed in warps with one cycle per instruction interval.

TABLE 2. Instruction latencies of NVIDIA P40.

Instruction	Type	Units	Latency
MULA	compute	sp	1 cycles
ADD	compute	sp	1 cycles
DIV	compute	sp	5 cycles
Shared	shared	LDST	92 cycles
Global load	global	LDST	374 cycles

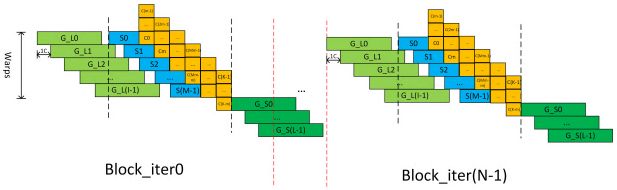


FIGURE 3. Computation-intensive instruction queue model. There are I Global Load instructions, M Shared Load instructions, K CP instructions and L Global Store instructions in each block iteration. The instruction are executed in warps with one cycle per instruction interval.

GPU chip manual, which is one clock cycle here. In this scenario, CP instructions are totally overlapped by Shared Load instructions. The number of clock cycles used in the execution of a block is equal to the sum of the other three instructions. Eq. (7) - Eq. (12) shows the computation process of the queue execution time.

$$C_k = C_b * N_{iter} + L_{GS} \tag{7}$$
$$C_b = I + M + L \tag{8}$$
$$I = \frac{GL_{dispatch}}{N_{iter}} \tag{9}$$
$$M = \frac{S_{dispatch}}{N_{iter}} \tag{10}$$
$$L = \frac{GS_{dispatch}}{N_{iter}} \tag{11}$$
$$N_{iter} = \left\lceil \frac{block_{num}}{sm_{num}} \right\rceil \tag{12}$$

On the contrary, computation-intensive means that the number of CP instructions is much larger than that of Shared Load instructions. Due to the large number of SP units on the GPU, multiple computation instructions can be executed in parallel. As shown in Fig.3, CP instructions can basically overlap the Shared Load instructions.

In this situation, the block execution time depends more on the number of exposed CP instructions. Since calculation instructions are executed by SP units and transmission instructions are executed by LDST units, the coverage ability of CP instructions are limited by hardware conditions. We use Eq. (17) to describe this limitation and corresponding formulas are as follows:

$$C_k = C_b * N_{iter} + L_{GS} \tag{13}$$
$$C_b = I + 1 + L_S + \left\lceil \frac{K}{m} \right\rceil + L \tag{14}$$
$$I = \frac{GL_{dispatch}}{N_{iter}} \tag{15}$$
$$K = \frac{CP_{dispatch}}{N_{iter}} \tag{16}$$
$$m = \frac{sm.SP_{num}}{sm.LDST_{num}} \tag{17}$$
$$L = \frac{GS_{dispatch}}{N_{iter}} \tag{18}$$
$$N_{iter} = \left\lceil \frac{block_{num}}{sm_{num}} \right\rceil \tag{19}$$

C. PARALLELISM ADJUSTMENT

Matrix operations on GPU are performed in blocks, and the block size affects the statistics of instructions. For example, a 128×32 matrix operation can be implemented in 4 iterations of a 32×32 block or in 32 iterations of a 128×1 block. Obviously the former division is more efficient. Therefore, unreasonable division will reduce computational efficiency and waste resources. In fact, matrices have fixed division rules in CUDA library, which is related to GPU architectures and specific computing tasks. In order to obtain the optimal partition results, we conduct a loop search under the bandwidth limitation of different cache levels. According to Sec.III B we know that the abstract instruction queue model is an ideal model that simplifies the execution of real instructions and makes full use of hardware resources. It may cause deviation from the real runtime (especially for small computational tasks). Here we define degree of parallelism (DOP) as a parameter to adjust the block partition. Ideally, the partition result is optimal when the DOP is 1.0. We adjust DOP between 0-1 and set benchmarks for different types of GPUs and CNNs.

D. THROUGHPUT MODEL

In the actual transfer test under multi-GPU scenario, we find that not only does the transmission bandwidth not reach the theoretical value of the physical channel, but also the real-time throughput is not a constant. It has a functional relationship with the size of transfer data [18]. After a large number of experiments, above variables exhibit a sigmoid-like distribution, and the function relationship shown in Fig.4 can be obtained by fitting, where x-axis is the natural logarithm size of data (MB) and y-axis is the throughput (GB/s).

The time consumption of one transmission is calculated as follows, where k, a and w are fitting parameters:

$$t_{trans} = \frac{e^x}{y * 1000} \tag{20}$$
$$y = k * \frac{1}{a + e^{-w*x}} \tag{21}$$

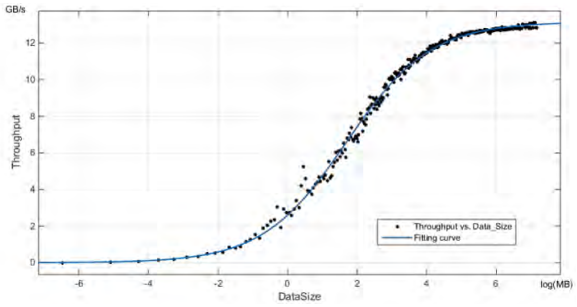


FIGURE 4. Relationship between throughput and data size.

TABLE 3. Summary of transmission notations.

Name	Description
t_{reduce}	Time of gradient upload, summary and calculate
$t_{broadcast}$	Download time of updated parameter
t_r^l	Gradient summary and calculation time of layer l
$t_{r\_o}^l$	Gradient calculation overlap time of layer l
t_{d2h}^l	Device to host transmission time of layer l's gradient
t_{h2d}^l	Host to device transmission time of layer l's gradient
$t_{CPU\_a}^l$	Gradient average calculation time on CPU of layer l
$t_{b\_lat}^l$	The transmission latency of the lth layer gradient update from host to device
gpu_num	Number of GPU
$link\_num$	Shared link number between CPU and GPU
$t_{allreduce}$	Exposed time of AllReduceKernel function
$\hat{t}_b^l$	Backward execution time of Layer l in NCCL mode
t_{ar}^l	AllReduceKernel function execution time of layer l
$t_{b\_o}^l$	Backward calculation overlap time with AllReduce Kernel of layer l
$t_{sr\_iter}^l$	Scatter reduce time during AllReduce process of layer l
$t_{ag\_iter}^l$	Allgather time during AllReduce process of layer l
$t_{ring\_trans}^l$	Ring transfer time during AllReduce process of layer l
$t_{GPU\_avg}^l$	Gradient average calculation time on GPU during AllReduce process of layer l

E. TRANSMISSION MODEL

In multi-GPU training scenarios, the transmission behavior of data decides the exposed transfer time. This section mainly discusses synchronous data parallel under the PS mode and the NCCL mode. In the PS mode, CPUs are usually used as server nodes to store and update parameters, so each iteration process includes uploading gradients, calculating the average of variables (called reduce in communication primitive) and distributing the latest parameters (called broadcast in communication primitive). Data upload and distribution only occurs between CPUs and GPUs, and there is no interaction among different GPUs. In addition, gradient upload and backpropagation are executed in parallel in Tensorflow. Once all gradients are uploaded, the CPU will distribute the updated results immediately and start next iteration. The process is shown in Fig.5, where green rectangles represent the CUDA kernel function, and yellow rectangles represent the transferring data.

Because of the huge amount of computation in CNNs, the time of gradient upload is almost overlapped by computing time, which causes that only the time of first layer

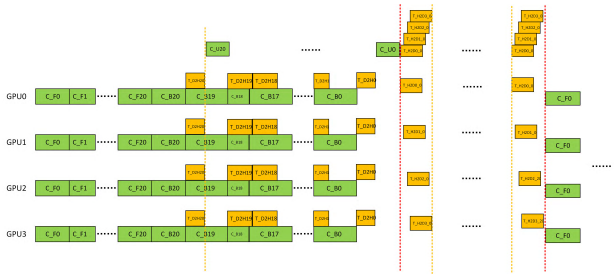


FIGURE 5. Data parallel transmission model in PS mode. C stands for calculation, T for transmission, F for forward, B for backward, U for CPU, D2H for device to host, and H2D for host to device. The network structure in figure has one CPU and four GPUs running in parallel.

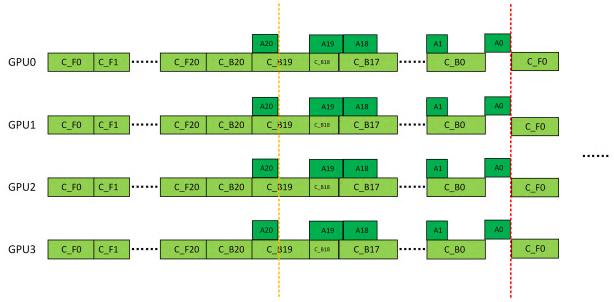


FIGURE 6. Data parallel transmission model in NCCL mode. C stands for calculation, F for forward, B for backward and A for AllReduceKernel. The figure shows the AllReduceKernel running in parallel with other kernel functions.

can be exposed during reduce process. In addition, there are different numbers of shared links among GPUs in various network structures, which is a key parameter affecting the traffic computing. We use link_num to represent it (equal to 1 if there is no shared link), and the following formula can be obtained:

$$t_{update} = t_{reduce} + t_{broadcast} \tag{22}$$

$$t_{reduce} = \sum_{l=2}^{N}(t_r^l - t_{r\_o}^{l-1}) + t_r^1 \tag{23}$$

$$t_{broadcast} = \sum_{l=1}^{N}(t_{h2d}^l * link\_num + (gpu\_num - 1) * t_{b\_lat}^l) \tag{24}$$

$$t_r^l = t_{d2h}^l * link\_num + t_{CPU\_a}^l \tag{25}$$

In the NCCL mode, the GPU calls AllReduceKernel function to update the model. The calculation process of AllReduceKernel is the same as that of PS mode, but they are executed on the GPU, which means that data transfer only occurs among GPUs. The process can be simplified by Fig.6, where light green rectangles represent the calculation of the CUDA kernel function, and dark green rectangles represent the AllReduceKernel which is executed in parallel with other kernel functions during back propagation.

AllReduceKernel is implemented by the ring allreduce algorithm. The calculation process can be divided into two steps [20]. The first step is scatter-reduce, and the second step is allgather. Take N nodes as an example, each node

will performs $N-1$ times of data transmission in step 1 and step 2 respectively. Each transmission contains K/N bytes of data, where K is the size of the original data. Therefore, the amount of data sent and received by each node is $2 \times K(N-1)/N$ bytes. Finally, the back propagation time of one layer depends on the minimum value of its calculation time and AllReduceKernel execution time. We use Eq. (26) - Eq. (32) to describe the above process.

$$t_{update} = t_{allreduce} \tag{26}$$

$$t_{allreduce} = \sum_{l=2}^{N}(t_{ar}^{l} - t_{b\_o}^{l-1}) + t_{ar}^{1} \tag{27}$$

$$t_{ar}^{l} = (gpu\_num - 1) * (t_{sr\_iter}^{l} + t_{ag\_iter}^{l}) \tag{28}$$

$$t_{sr\_iter}^{l} = t_{ring\_trans}^{l} + t_{GPU\_avg}^{l} \tag{29}$$

$$t_{ag\_iter}^{l} = t_{ring\_trans}^{l} \tag{30}$$

$$\hat{t_b^l} = \begin{cases} \dfrac{t_b^l * sm_{num}}{sm_{num} - 1} & t_{ar}^{l-1} \geq t_b^l \\ \dfrac{t_b^l * sm_{num} - t_{ar}^{l-1} * (sm_{num} - 1)}{sm_{num}} + t_{ar}^{l-1} & t_{ar}^{l-1} < t_b^l \end{cases} \tag{31}$$

$$t_{b\_o}^{l} = min(\hat{t_b^l}, t_{ar}^{l-1}) \tag{32}$$

IV. EXPERIMENTS

In this section, we assess our proposed framework in terms of runtime prediction at both layer level and network level. The input of the framework is a set of characteristic parameters including CNNs, GPUs, and networks, as shown in Table 4. In order to verify the accuracy of our model, we compare the predicted results with actual runtime of CNNs. As for the runtime test, we first use CIFAR-10 to train real networks such as Alexnet to ensure that the accuracy of CNNs closes to the original paper, and then test the real runtime by generating random numbers of the same size as ImageNet and CIFAR in the same environment. After 10 rounds of hot start, the average time of 100 iterations is used as the final result. The experimental platform is: CPU Intel (R) Xeon E5-2650 v4 2.20GHz, Ubuntu 16.04.1, python 3.5.2, tensorflow-gpu 1.8.0, cuda 9.0, cudnn 7.1.4, NVIDIA P40, K80, K40, and GTX 1080 Ti.

A. LAYER-LEVEL TIME EVALUATION

We choose Alexnet and Resnet-50 as typical examples to evaluate the layer-wise time. Fig.7 accurately reflects the change in runtime of each layer. In Alexnet, our model captures that conv1 is the main bottleneck across the whole network, because the filter size of conv1 is 11*11, which causes a large amount of calculation at the input layer. In practice, in order to optimize the computation, a smaller filter not only reduce the time consumption, but may also perform well. In contrast, Resnet-50 has a more uniform time distribution across the network. We split it by residual blocks, and each block is consists of 3 CONV layers and 3 BN layers. We find that the predicted results of the model do not always follow the trend among layers. That is because in the actual operation, as matrix dimensions are modified, the performance does not change linearly, and in practice the system internally chooses from one of 15 implementations for the operation [22]. Our model only considers two algorithms that are the most critical to the accuracy of iteration time prediction.

TABLE 4. Model input parameters.

Description	Source
Layer type	CNN Architecture
Input feature map number	CNN Architecture
Input feature map height	CNN Architecture
Input feature map width	CNN Architecture
Output feature map number	CNN Architecture
Filter height	CNN Architecture
Filter width	CNN Architecture
Vertical stride	CNN Architecture
Horizontal stride	CNN Architecture
Padding mode (Valid/Same)	CNN Architecture
Data width (default 32bit)	CNN Architecture
Batch size	CNN Architecture
GPU Version (default NVIDIA GPU)	Hardware features
Calculate ability (Can be found in [31])	Hardware Features
CUDA cores number	Hardware Features
Memory bandwidth	Hardware Features
Base clock	Hardware Features
CPU frequency	Hardware Features
Transmission mode (PS/NCCL)	NetworkFeatures
GPU number	Network Features
GPU direct communication topology	Network Features
Link topology	Network Features
Link bandwidth	Network Features

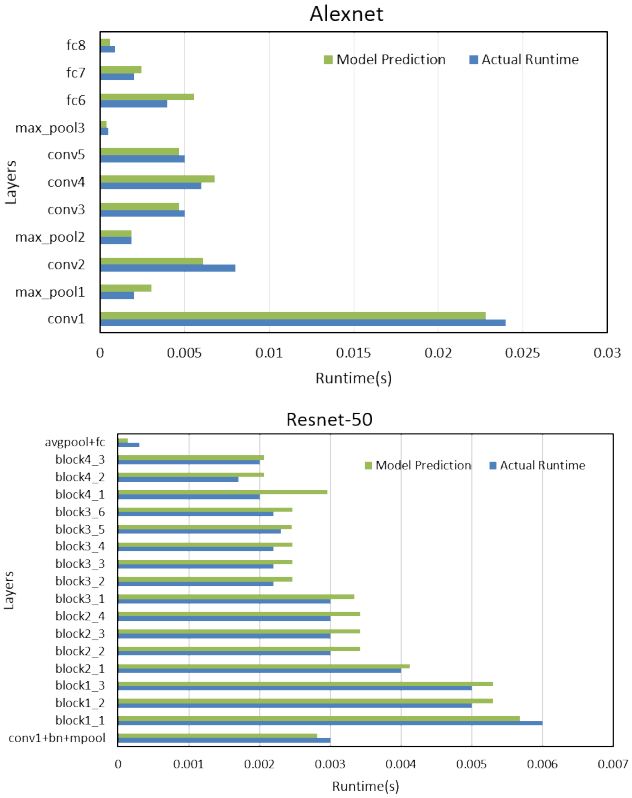
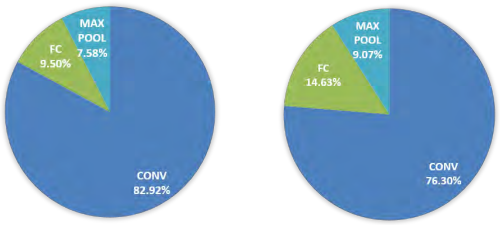


FIGURE 7. Comparison of runtime prediction for each layer in Alexnet (batch size 256) and Resnet-50 (batch size 32) on GTX 1080 Ti.



(a) (b)

FIGURE 8. Comparison of time proportions of different layer types in Alexnet. (a) Actual runtime proportion. (b) Predicted time proportion.

TABLE 5. Comparison of runtime models for convolution layer.

Proposed model		NeuralPower		Paleo	
RMSPE	RMSE(ms)	RMSPE	RMSE(ms)	RMSPE	RMSE(ms)
18.67%	0.599	39.97%	1.019	58.29%	4.304

TABLE 6. Comparison of runtime prediction for different GPU platforms.

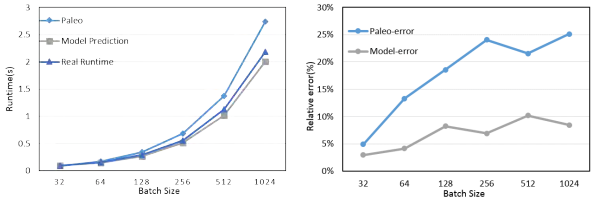
Alexnet	Actual runtime (s)		Predicted time (s)	
GPU	Forward	Forward-Backward	Forward	Forward-Backward
P40	0.03494	0.10996	0.0331	0.097
K80	0.08843	0.29054	0.087	0.267
K40	0.0907	0.2958	0.0869	0.2609

Resnet-50	Actual runtime (s)		Predicted time (s)	
GPU	Forward	Forward-Backward	Forward	Forward-Backward
P40	0.28805	0.82289	0.2987	0.9012
K80	0.78835	2.42375	0.814	2.315
K40	0.73341	2.29207	0.8	2.3

Fig.8 shows the time proportion of Alexnet, among which CONV layer is the most time-consuming of three layer types. The proportion of FC and CONV layer is more than 90% which is larger in Resnet. The influence of POOL layer can be almost neglected in the rough time optimization. Considering the importance of CONV layer for CNNs, Table 5 compares the model prediction results with the state-of-the-art method, NeuralPower and Paleo. To enable a comparison here, we use the same evaluation criteria, root mean square error (RMSE) and the relative root mean square percentage error (RMSPE) [16]. It can be observed that the prediction accuracy of our model for CONV layer is significantly better than the state of the art, with an increase of 21.3%.

B. NETWORK-LEVEL TIME EVALUATION ON SINGLE GPU

To verify the accuracy of the framework, we further apply it to three other GPU platforms and evaluate the entire network. We run Alexnet and Resnet-50 respectively on NVIDIA P40, K80 and K40, and Table 6 shows the prediction results of full forward pass and forward-backward pass. Though K80 has two GK210 core, we just use single core in order to compare the computation time. It can be seen from the predicted result that the training efficiency of K80 with single core is quiet similar to that of K40, and the training speed of P40 is nearly three times faster than the other two, which is consistent with the actual results. Our full pass estimates are remarkably accurate and achieves consistent performance across different GPU platforms. From this point of view,



(a) (b)

FIGURE 9. Comparison of model prediction results with the change of batch size on Alexnet. (a) Predicted Results. (b) Relative Error.

TABLE 7. Relative error comparison of runtime models for common CNNs on single GPU.

CNN name	Proposed model	NeuralPower	Paleo
Alexnet (224*224)	5.61%	11.25%	16.72%
Alexnet-4 (32*32)	14.06%	–	–
VGG-16	1.07%	1.44%	6.13%
Overfeat(fast)	3.68%	1.4%	42.86%
Resnet-50(224*224)	4.83%	–	–
Resnet-20(32*32)	14.12%	–	–

our results can provide great help for AI researchers to choose the appropriate hardware platform.

We execute the Paleo code on Alexnet as the baseline to compare it with our framework, and further evaluate the relationship of prediction results and batch size. The results are shown in Fig.9. The average accuracy of our model on Alexnet reaches 94.39%, which is improved by 11.11% compared with Paleo. With the increase of single iteration batch size in practical training, the calculation and transmission performance of the GPU gradually reach the bottleneck, and the prediction deviation also gradually increases. Considering the measurement deviation caused by the instability of the system, we believe that the smoothness of relative error curve is an important standard to reflect the performance of the model. It can be seen from the figure that the prediction made by our model is more stable than that of Paleo. Meanwhile, the model also has a good performance on Resnet-50, with an average accuracy of 95.17%.

In addition to GPU platforms and batch size, we also assess the effectiveness of our model in six different state-of-the-art CNN architectures and compare it with NeuralPower and Paleo. Considering the differences between platforms and systems, here we use relative errors as evaluation criterion. The data of NeuralPower and Paleo comes from article [35], as shown in Table 7. We can clearly see that the relative error of our model is always within 15% which is much more stable than Paleo. Besides, compared with NeuralPower, our model has different degrees of improvement on Alexnet (5.64%) and VGG-16 (0.37%), and the performance on Resnet is also good (with an error of less than 5%), which is not mentioned by the other two models. In order to explore the influence of network size, we carry out experiments on Alexnet and Resnet with random numbers of the same size as data sets CIFAR(32*32) and ImageNet (224*224). It can be found that the prediction error on small networks is larger, and we believe there are two main reasons. On the one hand,

TABLE 8. Resnet-50 data parallel comparison between actual runtime and model prediction in the PS mode and the NCCL mode.

PS mode	Actual runtime (s)		Proposed model (s)	
GPU number	Total	Speedup	Total	Speedup
1	0.71388	1X	0.72844	1X
2	0.36984	1.93X	0.3691	1.97X
4	0.20364	3.5X	0.18937	3.84X

NCCL mode	Actual runtime (s)		Proposed model (s)	
GPU number	Total	Speedup	Total	Speedup
1	0.71838	1X	0.72891	1X
2	0.37852	1.9X	0.37351	1.95X
4	0.21069	3.4X	0.19409	3.76X

the time of a single iteration is so small on our platform that the variance accounts for a larger proportion (sometimes up to 50%). On the other hand, the small amount of calculation leads to a lower resource utilization on GPU platforms. In contrast, large-scale network prediction has more practical significance, which is also the main concern of our work.

C. NETWORK-LEVEL TIME EVALUATION ON MULTIPLE GPUs

Multi-GPU execution mode in Tensorflow is the most important applicable scenario of our framework, which is not mentioned by NeuralPower or Paleo. In order to fully utilize the performance of the GPU platform and eliminate other interference, we use data parallelism algorithm to train Resnet-50 on NVIDIA P40. According to the difference of transmission methods, we compare the iteration time and acceleration ratio in the PS mode and the NCCL mode respectively, as shown in Table 8. Since there are four separate PCIe*16 buses in our experimental environment, our test obtained an nearly ideal acceleration because of the high bandwidth, and the difference between two modes is not obvious (the PS mode is a little better than the NCCL mode) in our condition. In the actual situation, specific problems need to be analyzed concretely because each mode has its own pros and cons. Compared with PS mode, although NCCL mode optimizes the transmission bandwidth, it occupies the computing resources of GPUs, which could lead to the decrease of computational efficiency. By comparing the predicted result of our framework, it will be more convenient for experimenters to choose the most suitable transmission mode.

Finally, prediction results of Resnet-50 in the PS mode are shown in Fig.10, where the iteration time varies with batch size and GPU numbers. It can be seen that our model can be adapted to the expansion of input data and reflects the real trend of runtime changes. The average error between the predicted result and actual runtime is less than 5%. In addition, comparing different curves in the figure, we find that if the batch size on each GPU is the same, the total iteration time is basically the same. It indirectly reflects that, in the case of fewer GPUs and better network bandwidth, the delay caused by transmission can be almost ignored. However in the real large-scale training scenario, the environment is not always ideal. Our framework can help researchers find

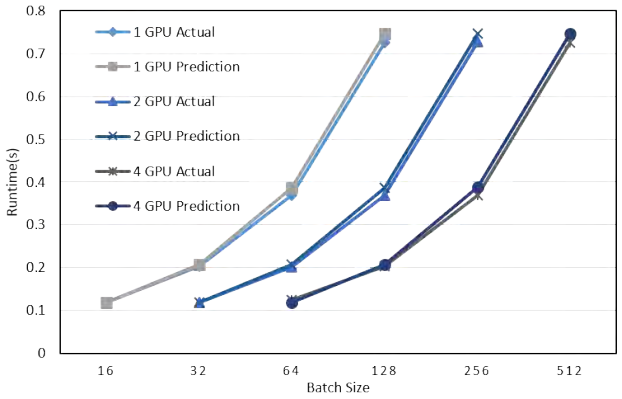


FIGURE 10. Comparison of model prediction results with the change of batch size and GPU numbers in the PS Mode.

training bottleneck and optimize the whole network by simulating more training parameters and GPU numbers.

V. CONCLUSION

In this work, we present a framework to predict the training time for CNNs in multi-GPU parallel scenario before training. Through layer-wise analysis, we construct abstract instruction queue model, throughput model and transmission model, which ultimately helps AI researchers analyze CNN training process from multiple perspectives. Our model is verified on four types of NVIDIA GPU platforms and six different CNN architectures. Detailed analysis is performed from angles including layer-level and network-level. Compared with the state of the art on single GPU, the accuracy of our proposed framework is improved by 21.3% in CONV layer prediction and 5.64% on Alexnet, while the average error is less than 15% on other networks. In addition, the framework also has a good performance in multi-GPU scenarios. It supports two transmission modes (PS mode and NCCL mode), which can accurately reflect the trend of time under different conditions. Finally, our framework structure has good scalability after systematic decomposition and model decoupling. At present, our work still has some limitations, for example, it is only applicable to Tensorflow and data parallel scenario, and its performance is not ideal for small-scale CNN prediction. In the future, we will explore more possibilities, including the expansion of software platforms, neural network structures and application scenarios.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[2] K. Simonyan and A. Zisserman. (2014). "Very deep convolutional networks for large-scale image recognition." [Online]. Available: https://arxiv.org/abs/1409.1556
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.
[4] S. Han *et al.* (2016). "ESE: Efficient speech recognition engine with sparse LSTM on FPGA." [Online]. Available: https://arxiv.org/abs/1612.00694
[5] Y. Wu *et al.* (2016). "Google's neural machine translation system: Bridging the gap between human and machine translation." [Online]. Available: https://arxiv.org/abs/1609.08144

[6] T. Ben-Nun and T. Hoefler. (2018). "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." [Online]. Available: https://arxiv.org/abs/1802.09941
[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[8] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.
[9] G. Chrysos, "Intel Xeon Phi coprocessor—The architecture," Intel Corp., Santa Clara, CA, USA, Intel Whitepaper 176, 2014.
[10] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement.*, vol. 16, Nov. 2016, pp. 265–283.
[11] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[12] T. Chen *et al.* (2015). "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems." [Online]. Available: https://arxiv.org/abs/1512.01274
[13] S. Chetlur *et al.* (2014). "cuDNN: Efficient primitives for deep learning." [Online]. Available: https://arxiv.org/abs/1410.0759
[14] A. Krizhevsky. (2014). "One weird trick for parallelizing convolutional neural networks." [Online]. Available: https://arxiv.org/abs/1404.5997
[15] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *Proc. 7th Int. Conf. Cloud Comput. Big Data (CCBD)*, Nov. 2016, pp. 99–104.
[16] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu. (2017). "NeuralPower: Predict and deploy energy-efficient convolutional neural networks." [Online]. Available: https://arxiv.org/abs/1710.05420
[17] K. Zhou, G. Tan, X. Zhang, C. Wang, and N. Sun, "A performance analysis framework for exploiting GPU microarchitectural capability," in *Proc. Int. Conf. Supercomput.*, 2017, p. 15.
[18] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. OSDI*, vol. 14, 2014, pp. 583–598.
[19] N. Luehr. *Fast Multi-GPU Collectives With NCCL*. Accessed: Apr. 7, 2016. [Online]. Available: https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/?tdsourcetag=s_pctim_aiomsg
[20] Baidu Research. *Baidu-Allreduce*. Accessed: Feb. 22, 2017. [Online]. Available: https://github.com/baidu-research/baidu-allreduce
[21] NVIDIA. *CUBLAS Library Documentation*. Accessed: Oct. 30, 2018. [Online]. Available:http://docs.nvidia.com/cuda/cublas
[22] Y. Oyama, A. Nomura, I. Sato, H. Nishimura, Y. Tamatsu, and S. Matsuoka, "Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2016, pp. 66–75.
[23] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2017.
[24] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in *Proc. 21st ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1355–1364.
[25] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," in *Proc. IEEE 16th Int. Conf. Dependable, Autonomic Secure Comput., 16th Int. Conf. Pervasive Intell. Comput., 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Aug. 2018, pp. 949–957.
[26] M. Mathieu, M. Henaff, and Y. LeCun. (2013). "Fast training of convolutional networks through FFTs." [Online]. Available: https://arxiv.org/abs/1312.5851
[27] A. Viebke, S. Memeti, S. Pllana, and A. Abraham, "CHAOS: A parallelization scheme for training convolutional neural networks on Intel Xeon Phi," *J. Supercomput.*, vol. 75, no. 1, pp. 197–227, 2019.
[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[29] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.
[30] NVIDIA. *NVIDIA CUDA C Programming Guide*. Accessed: Oct. 30, 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[31] S. Ioffe and C. Szegedy. (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift." [Online]. Available: https://arxiv.org/abs/1502.03167
[32] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs," in *Proc. 15th Annu. Conf. Int. Speech Commun. Assoc.*, 2014, pp. 1058–1062.
[33] Y. You, Z. Zhang, C. J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, p. 1.
[34] M. Tan *et al.* (2018). "MnasNet: Platform-aware neural architecture search for mobile." [Online]. Available: https://arxiv.org/abs/1807.11626
[35] D. Marculescu, D. Stamoulis, and E. Cai, "Hardware-aware machine learning: Modeling and optimization," in *Proc. Int. Conf. Comput.-Aided Design*, 2018, p. 137.

ZIQIAN PEI received the B.S. degree in communication and information engineering from Shanghai University, Shanghai, China, in 2016. She is currently pursuing the M.S. degree with the Wireless Information Network Laboratory, University of Science and Technology of China. Her research interests include big data analysis, data mining, and artificial intelligence application in the field of communications.

CHENSHENG LI received the B.S. degree in communication and information engineering from the University of Science and Technology of China, Hefei, China, in 2015, where he is currently pursuing the Ph.D. degree with the Wireless Information Network Laboratory. His research interests include deep learning in non-Euclidean domain, graph neutral networks, and machine learning in mobile networks.

XIAOWEI QIN received the B.S. and Ph.D. degrees from the Department of Electrical Engineering and Information Science, University of Science and Technology of China (USTC), Hefei, China, in 2000 and 2008, respectively. Since 2014, he has been a Member of Staff with the Key Laboratory of Wireless-Optical Communications, Chinese Academy of Sciences, USTC. His research interests include optimization theory, service modeling in future heterogeneous networks, and big data in mobile communication networks.

XIAOHUI CHEN received the B.S. and M.S. degrees in communication and information engineering from the University of Science and Technology of China (USTC), Hefei, China, in 1998 and 2004, respectively, where he is currently an Associate Professor with the Department of Electronic Engineering and Information System. His current research interests include wireless network QoS, mobile computing, MAC protocol, and traffic model.

GUO WEI received the B.S. degree in electronic engineering from the University of Science and Technology of China (USTC), Hefei, China, in 1983, and the M.S. and Ph.D. degrees in electronic engineering from the Chinese Academy of Sciences, Beijing, China, in 1986 and 1991, respectively. He is currently a Professor with the School of Information Science and Technology, USTC. His current research interests include wireless and mobile communications, wireless multimedia communications, ultra-wideband communication systems, and wireless information networks.