WORKLOAD-AWARE AUTOMATIC PARALLELIZATION FOR MULTI-GPU DNN TRAINING

Sungho Shin[†], *Youngmin Jo*[†], *Jungwook Choi*[⋆], *Swagath Venkataramani*[⋆], *Vijayalakshmi Srinivasan*[⋆], *and Wonyong Sung*[†]

[†]Department of Electrical and Computer Engineering, Seoul National University, Seoul, 08826 Korea
[⋆]IBM Research AI, 1101 Kitchawan Rd. Yorktown Heights, New York 10598
sungho.develop@gmail.com, youngmin.research@gmail.com, choij@us.ibm.com, swagath.venkataramani@ibm.com, viji@us.ibm.com, wysung@snu.ac.kr

ABSTRACT

Deep neural networks (DNNs) have emerged as successful solutions for variety of artificial intelligence applications, but their very large and deep models impose high computational requirements during training. Multi-GPU parallelization is a popular option to accelerate demanding computations in DNN training, but most state-of-the-art multi-GPU deep learning frameworks not only require users to have an in-depth understanding of the implementation of the frameworks themselves, but also apply parallelization in a straight-forward way without optimizing GPU utilization. In this work, we propose a workload-aware auto-parallelization framework (WAP) for DNN training, where the work is automatically distributed to multiple GPUs based on the workload characteristics. We evaluate WAP using TensorFlow with popular DNN benchmarks (AlexNet and VGG-16), and show competitive training throughput compared with the state-of-the-art frameworks, and also demonstrate that WAP automatically optimizes GPU assignment based on the workload's compute requirements, thereby improving energy efficiency.

***Index Terms*—** Multi-GPU training, data parallelization, auto parallelization, neural network training, deep learning framework

1. INTRODUCTION

In recent years, deep learning (DL) has emerged as the dominant solution showing remarkable success in a wide spectrum of artificial intelligence (AI) applications [1, 2, 3, 4, 5, 6, 7]. In each of these domains, deep neural networks (DNNs) achieve superior accuracy through the use of very large and deep models – necessitating up to 100s of ExaOps of computation during training.

To deal with the high computational demands and to achieve high throughput, DNN training is often accelerated by parallelizing across multiple GPUs. Popular DL frameworks such as TensorFlow [8] and Pytorch [9] provide native GPU support. However, adapting a single-GPU DNN model to work with multi-GPU environments is not trivial for users, since they must consider not only how computation will be distributed across multiple GPUs but also what data will be exchanged via communication between GPUs. A sub-optimal implementation decision can easily lead to poor GPU utilization causing a significant drop from the expected speedup due to parallelization.

There have been several TensorFlow based parallelization frameworks to ease the burden of multi-GPU implementation for the users, such as Parallax [10]. Parallax provides a set of Python-level APIs for the users to adapt their single GPU code for the multi-GPU runs. These APIs help specify detailed information to the DL framework about the DNN description as well as the list of available GPUs. Using such information Parallax distributes the computations across GPUs and executes them in parallel. These frameworks also adopt the popular communication protocols such as Open MPI [11] and NVIDIA collective communications library (NCCL) [12] for efficient data communication. However, they do not take into account GPU utilization during parallelization. Instead, these frameworks distribute the workload to all the available GPUs oblivious to the users. It is well known that GPU suffers low utilization when the workload is not sufficiently large, e.g., when minibatch size is small in DNN training. Thus, it has been so far the users' responsibility to determine the optimal number of GPUs based on the DNN workloads, and accordingly utilize the APIs.

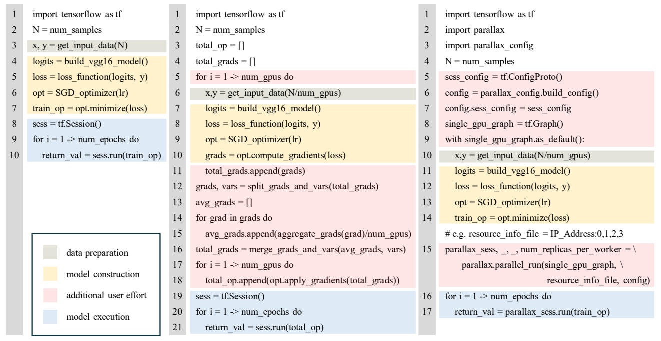
In this work, we propose a novel workload-aware automatic parallelization framework (WAP). Different from the existing frameworks, WAP applies parallelization under the hood of TensorFlow source code. At this stage the computation workloads across the DNN layers are fully specified into a dataflow graph and ready for analysis. We devise the workload analysis unit in our parallelization framework to estimate the expected utilization of the GPUs and determine the best number of GPUs to be assigned. Based on this analysis, we directly modify TensorFlow's dataflow graph to seamlessly enable multi-GPU parallelization. We automate all of these steps in WAP without requiring any additional inputs from the users. In particular, the users do not need to manually find out the number of GPUs optimally suited for a given workload. We evaluate WAP with the popular DNN benchmarks, and demonstrate that it not only achieves competitive throughput and scalability compared to the state-of-the-art multi-GPU framework [10], but also automatically optimize the GPU assignment based on the workload analysis, improving energy efficiency.

2. RELATED WORK

2.1. Parallelization Strategies for DNN Training

There have been extensive research on parallelization of DNN training [1, 13, 14]. Most efforts can be categorized into three strategies: data, model, and hybrid parallelization. In data parallelization, each GPU uses the same DNN model (i.e., "Replicated-Variables" in [15])

This work was supported in part by the Brain Korea 21 Plus Project and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2018R1A2A1A05079504).



```
1   import tensorflow as tf
2   N = num_samples
3   x, y = get_input_data(N)
4   logits = build_vgg16_model()
5   loss = loss_function(logits, y)
6   opt = SGD_optimizer(lr)
7   train_op = opt.minimize(loss)
8   sess = tf.Session()
9   for i = 1 -> num_epochs do
10      return_val = sess.run(train_op)
```

▨ data preparation
🟨 model construction
🟥 additional user effort
🟦 model execution

(a) Single-GPU (input for WAP)

```
1   import tensorflow as tf
2   N = num_samples
3   total_op = []
4   total_grads = []
5   for i = 1 -> num_gpus do
6       x,y = get_input_data(N/num_gpus)
7       logits = build_vgg16_model()
8       loss = loss_function(logits, y)
9       opt = SGD_optimizer(lr)
10      grads = opt.compute_gradients(loss)
11      total_grads.append(grads)
12  grads, vars = split_grads_and_vars(total_grads)
13  avg_grads = []
14  for grad in grads do
15      avg_grads.append(aggregate_grads(grad)/num_gpus)
16  total_grads = merge_grads_and_vars(avg_grads, vars)
17  for i = 1 -> num_gpus do
18      total_op.append(opt.apply_gradients(total_grads))
19  sess = tf.Session()
20  for i = 1 -> num_epochs do
21      return_val = sess.run(total_op)
```

(b) TF multi-GPUs

```
1   import tensorflow as tf
2   import parallax
3   import parallax_config
4   N = num_samples
5   sess_config = tf.ConfigProto()
6   config = parallax_config.build_config()
7   config.sess_config = sess_config
8   single_gpu_graph = tf.Graph()
9   with single_gpu_graph.as_default():
10      x,y = get_input_data(N/num_gpus)
11      logits = build_vgg16_model()
12      loss = loss_function(logits, y)
13      opt = SGD_optimizer(lr)
14      train_op = opt.minimize(loss)
    # e.g. resource_info_file = IP_Address:0,1,2,3
15  parallax_sess, _, _, num_replicas_per_worker = \
          parallax.parallel_run(single_gpu_graph, \
                                      resource_info_file, config)
16  for i = 1 -> num_epochs do
17      return_val = parallax_sess.run(train_op)
```

(c) Parallax multi-GPUs

Fig. 1: Example pseudocodes for a single or multi-GPU training. (a), (b): single and multi-GPU implementation using default TensorFlow [8], and (c) multi-GPU with Parallax [10]. The red boxes indicate an additional user effort to execute multi-GPU training. Note that WAP can execute multi-GPU training only using (a).

to train on a different subset of training data and compute gradients, which need to be aggregated across the GPUs [1]. In model parallelization, a DNN model is split and distributed across multiple GPUs and each GPU is responsible only for updating a portion of the model [13]. Hybrid parallelization blends model and data parallelization; e.g., [14] employs data and model parallelization for convolution and fully-connected layers, respectively. In this work, we focus on data parallelization as it is one of the most popular multi-GPU training schemes. However our under-the-hood parallelization based on TensorFlow dataflow graph is not limited to data parallelization; implementation of model and hybrid parallelization strategies will be future work.

2.2. Multi-GPU Data-Parallel Training Frameworks

Many deep learning frameworks provide native GPU support to exploit its extensive parallel computing power in accelerating DNN training [8, 9, 16]. However, most of them require non-trivial manual efforts for converting a single-GPU code into a multi-GPU version. As an example, Figure 1 shows how a single-GPU implementation of VGG-16 is converted in TensorFlow and Parallax for multi-GPU runs. As shown in Figure 1(b), TensorFlow requires users to handle details of multi-GPU implementation, such as the replication of DNN models (line 5-11) and the gradient aggregation (line 12-18).

To ease the users' burden of multi-GPU implementation, several frameworks provide Python-level APIs, and Parallax [10] is one of the most recent development. Parallax provides a software API for efficient distributed training (using Horovod [17] for efficient data communication) that hides detailed parallelization settings from the users, as shown in Figure 1(c). While Parallax alleviates user effort for multi-GPU DNN training, it does not take into account GPU utilization during parallelization. As shown in Figure 1(c), Parallax simply uses the list of GPUs available to the user, regardless of the amount of work for each layer of the neural network (line 15). Therefore, even if the GPU utilization is low due to a small workload size for a given DNN layer (e.g., when minibatch size is small), Parallax obliviously allocates all the GPUs, potentially wasting power and degrading performance due to unnecessary communication overheads.

In this work, we set out to realize auto-parallelization while being cognizant of the expected GPU utilization. Unlike other existing multi-GPU frameworks, our framework provides automatic parallelization starting from a single-GPU code from the user, and takes into consideration the workload's compute requirements to optimally choose the number of GPUs for parallelization so as to increase both throughput and energy efficiency.

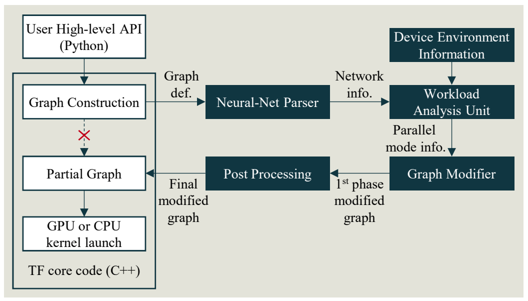


Fig. 2: Overview of workload-aware auto parallelization (WAP). The white boxes represent execution steps in TensorFlow core source code, and the navy boxes are added steps for WAP.

3. WORKLOAD-AWARE AUTOMATIC PARALLELIZATION

3.1. Overview

In this section, we explain details of our workload-aware automatic parallelization (WAP) framework. The key challenge in WAP is to extract the total amount of computations for all the DNN layers, including minibatch size, before applying parallelization. In TensorFlow, a dataflow graph created inside the TensorFlow core source code captures all the workload details of a given DNN. Therefore, we decided to augment the TensorFlow core source code for workload-aware parallelization.

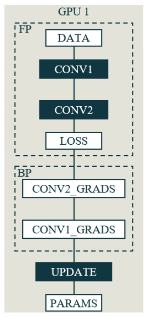
Figure 2 shows the overall steps of WAP. The dataflow graph constructed from the Python-level user description of a DNN is sent to a *Neural-Net Parser* to extract workload information. This information is used by the *Workload-aware Analysis Unit* (WAU) to determine the optimal number of GPUs for parallelization. Next, the original dataflow graph is transformed via *Graph Modifier* to reflect the GPU parallelization choice from WAU, followed by *Post Processing* of the graph to avoid any redundancy in data communication. The final modified graph is then registered as the new graph object into TensorFlow, where each GPU is allocated a partial graph for parallel execution.

There are two advantages from our approach: 1) we extract the workload information at the same level as TensorFlow core does, so that we can perform precise workload analysis for the best GPU usage, and 2) by applying parallelization at the dataflow graph level under the hood, we hide all the burden of multi-GPU parallelization from the users. In the following subsection, we explain the important modules of WAP shown in Figure 2.

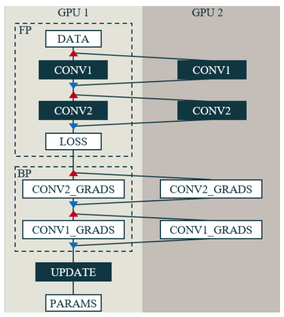
3.2. Implementation Details

3.2.1. Workload Analysis Unit

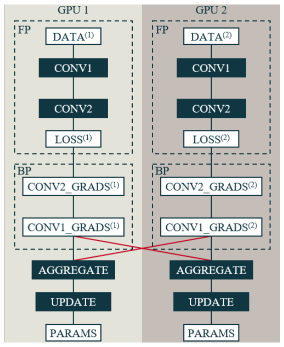
Workload analysis unit (WAU) receives the device (i.e., available GPUs) and the neural network information to determine how to perform data parallelization with the highest GPU utilization. The more GPUs are used, the more fine-grain the workload is divided, at some point making the amount of computation not enough to maintain high GPU utilization. At the same time, using more GPUs increases the communication overhead for the gradient aggregation. Therefore, sometimes using more GPUs achieves slower training speed. WAU analyzes the workload using the performance model to detect



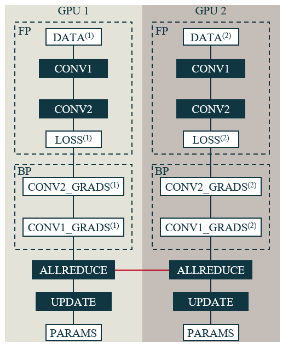
(a) A dataflow graph from Single-GPU code



(b) Step1 (node replication)



(c) Step2 (removing overhead)



(d) Step3 (efficient gradient aggregation)

Fig. 3: Illustration of workload-aware parallelization (WAP). (a) A dataflow graph for Single-GPU code. (b) The primary computation nodes are replicated with "split" and "concatenate" nodes (red and blue triangles, respectively). (c) Unnecessary data communication between devices are removed. (d) Gradients aggregation is optimized with NCCL AllReduce.

and use only the number of GPUs required to improve both throughput and energy efficiency.

There are several choices available to estimate expected performance. In this work, we adopt the GPU execution time model from [18] to evaluate run-time performance as follows:

$$t_{\text{estimate}} = \sum_{l_i} \{t_c(l_i, d) + t_s(l_i, d)\} \tag{1}$$

where t_{estimate} is the estimated total execution time for the entire layers of a DNN, l_i denotes i-th layer in the network, d is a factor for workload division, t_c is the processing time for the forward and backward propagation, and t_s represents the data communication time for gradient aggregation. We change d from 1 to the total available GPUs to find out how many GPUs are needed to minimize t_{estimate}. In Section 4.3 we show that this performance analysis successfully guides to find the right parallelization that maximizes throughput and avoids unnecessary power consumption.

3.2.2. *Neural-Net Parser and Graph Modifier*

Neural-Net Parser identifies which computational workloads need to be parallelized based on the parallelization strategies. In this work, it finds the computationally challenging layers such as convolution, and fully-connected layers from the dataflow graph and extract the workload information for analysis in WAU.

Table 1: Training throughput (images/sec) of the WAP graph transformation steps on AlexNet [1] with four GPUs and 2048 minibatch. Note that "before" represents single GPU performance. Experiments are conducted on "single machine" which is explained in Section 4.1.

	Before	Step1	Step2	Step3
AlexNet	2482	421	7264	7904

Graph Modifier transforms the original dataflow graph into the multi-GPU version as shown in Figure 3. Based on the number of GPUs determined by WAU, it first replicates the primary computation nodes (e.g., convolution and fully-connected), and split/concatenate their input and output, respectively (Figure 3(b)). Note that naive node replication in this step causes unnecessary links for data communication across GPUs. In order to avoid such communication overhead, *Graph Modifier* further replicates auxiliary computation nodes (e.g., activation functions, loss computation, etc.) and removes unnecessary split/concatenation (Figure 3(c)).

3.2.3. *Post Processing for Optimizing Gradient Reduction*

As explained in Section 2.1, in data parallelization, gradients calculated within each GPU need to be aggregated for the weight update. Naive implementation of this gradient aggregation can cause significant overhead, since it requires all-to-all data communication as shown in Figure 3(c). The complexity of this communication overhead is O(WN^2), where the size of weight is W and the number of GPUs is N. This communication overhead can be reduced by exploiting AllReduce with the ring algorithm, which reduces the complexity down to O(WN) [12]. As shown in Figure 3(d), in *Post Processing*, we inserted NCCL AllReduce for efficient gradients aggregation.

3.2.4. *Evaluation*

Each of the steps above in our graph-based data parallelization method is critical for achieving the best parallelization performance. To quantify the importance of each step, Table 1 reports the impact of each step to the training throughput. Note that a naive replication of primary computation nodes in Step1 significantly degrades the throughput. Replicating auxiliary computation nodes accordingly with the removal of redundant data communication in Step2 helps recover the throughput. Optimization via NCCL AllReduce in Step3 further increases the throughput by 9%.

4. EXPERIMENTAL RESULTS

4.1. Experimental Setup

All experiments are performed on two environments: A single machine with 4 GPUs (SM) and NVIDIA DGX-1 (DGX) [19]. SM is equipped one AMD Ryzen Threadripper 1900X 8-core CPUs, 96GB main memory, and four NVIDIA TitanXP GPUs (connected via PCIe). DGX is equipped with dual 20-core Intel Xeon ES-2698, 512GB main memory, and eight NVIDIA Tesla GP100 GPUs (connected via NVLink). We evaluate our framework with the two well-known convolutional neural networks (CNNs), AlexNet [1] and VGG-16 [2]. We employ data-parallel training with 512 and 64 per-GPU minibatch for AlexNet and VGG-16, respectively.

We compare the training throughput of WAP against two state-of-the-art multi-GPU implementations: TensorFlow high performance benchmark (TF-Bench) [20] and Parallax [10]. Note that TF-Bench is manually coded with the parallelization details hand-optimized, and Parallax is coded with the provided API. Whereas,

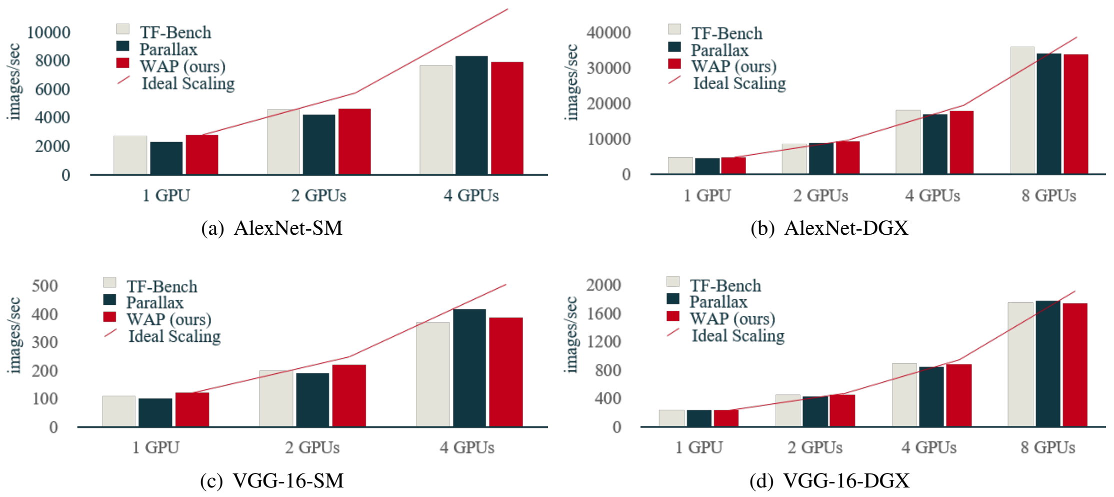


Fig. 4: Comparison of training throughput (images/sec) for AlexNet and VGG-16 on Single Machine (SM) and NVIDIA DGX-1 (DGX). WAP demonstrates compelling performance and favorable scalability without asking user effort for multi-GPU runs

WAP does not need any change from the single-GPU code. For fair comparison, all three implementations employ the same optimization schemes, such as Replicated-Variables and AllReduce (TF-Bench and WAP use NCCL AllReduce, and Parallax uses AllReduce from Horovod), for the gradient aggregation.

4.2. Training Performance

First, we evaluate the training performance of WAP in terms of scalability in throughput. Figure 4(a) and Figure 4(b) show the training throughput of AlexNet on SM and DGX, respectively. Similarly, Figure 4(c) and Figure 4(d) are for VGG-16 on SM and DGX, respectively. Throughout the experiments, WAP consistently demonstrates competitive performance. In particular, its throughput is in par with TF-Bench for the most cases, validating that the auto-parallelized execution of WAP is as good as the hand-optimized TF-Bench code. Note that Parallax shows slightly lower/higher performance with smaller/larger number of GPUs, respectively. This is in part due to the AllReduce implementation in Parallax; it employs Horovod's AllReduce, which reports better AllReduce performance with large number of GPUs [17], but its MPI runs would suffer higher overhead when the number of GPUs is small. The results on DGX show better scalability than SM, since the communication overhead of DGX is further reduced thanks to NVLink. Overall, WAP achieves compelling performance and scalability without requiring manual user effort for multi-GPU runs.

4.3. Workload-Aware GPU Allocation

We now showcase a scenario when the workload-aware GPU allocation ends up achieving higher speedup as well as saving power consumption. Table 2 shows the measured throughput and power consumption in the SM machine (with 4 GPUs) for training AlexNet with minibatch of 128.

In case of Parallax, all four GPUs are obliviously used for data parallelization. Since each GPU gets 32-minibatch amount of workload, which is not large enough to achieve high GPU utilization, the speedup by parallelization is overshadowed by the increased data communication overhead. Thus Parallax suffers lower throughput using 4 GPUs than what it could achieve with 1 GPU. In case of

Table 2: Comparison of throughput (images/sec) and power consumption (Watt) of Parallax and WAP on Alexnet with minibatch size of 128 on Single Machine (SM) with four GPUs. The numbers in parentheses mean "Used GPUs". Parallax obliviously uses four GPUs, while WAP chooses to use one GPU based on the workload estimation by WAU.

	Available GPUs	Parallax	WAP (Ours)	
		Measured (used GPUs)	Estimated by WAU	Measured (used GPUs)
Throughput (images/sec)	1	1986 (1)	2244	2560 (1)
	4	**1473 (4)**	1491	**2560 (1)**
Power (Watt)		402.81	149.44	

WAP, however, the workload is first analyzed by WAU, where the estimated throughput from Equation (1) indicates that the 1-GPU run would outperform the 4-GPU run. Based on this analysis, WAP uses only 1 GPU and achieve higher throughput. This demonstrates that WAU effectively hides the burden of optimizing GPU utilization from the users.

The workload-aware GPU allocation also has significant impact on energy efficiency. In case of Parallax, 4 GPUs are used (although each of them are running with lower utilization), thus it suffers high power consumption. In contrast, WAP only uses one GPU, reducing power consumption by 63% compared to Parallax.

5. CONCLUDING REMARKS

In this work, we proposed a workload-aware automatic parallelization (WAP) framework for DNN training, which automatically distributes work to multi-GPUs based on the workload characteristics. The proposed tool is implemented on the TensorFlow core source code for executing multi-GPU training without any end-user's effort. WAP automatically modifies the single-GPU to multi-GPU graph with the significant consideration of communication cost and distribution of computational nodes. We evaluate WAP with popular DNN benchmarks (AlexNet and VGG-16), and show competitive training throughput compared with the state-of-the-art hand-optimized parallelization frameworks, and also demonstrate that WAP automatically optimizes GPU assignment based on the workload's compute requirements, thereby decreasing power consumption and improving overall energy efficiency.

6. REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 369–376.

[5] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[6] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[7] Sungho Shin and Wonyong Sung, "Dynamic hand gesture recognition for wearable devices with low complexity recurrent neural networks," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 2274–2277.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, Software available from tensorflow.org.

[9] Nikhil Ketkar, "Introduction to pytorch," in *Deep Learning with Python*, pp. 195–208. Springer, 2017.

[10] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun, "Parallax: Automatic data-parallel training of deep neural networks," *arXiv preprint arXiv:1808.02621*, 2018.

[11] William D Gropp, William Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1, MIT press, 1999.

[12] "https://developer.nvidia.com/nccl/," .

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al., "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

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

[15] "https://www.tensorflow.org/performance/performance_model s#replicated_variables," .

[16] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin, "Dynet: The dynamic neural network toolkit," *arXiv preprint arXiv:1701.03980*, 2017.

[17] Alexander Sergeev and Mike Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[18] Charles R. Qi Alex Aiken Zhihao Jia, Sina Lin, "Exploring hidden dimensions in accelerating convolutional neural networks," in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, Stockholm, Sweden, 2018, pp. 2274–2283.

[19] "https://www.nvidia.com/en-us/data-center/dgx-1/," .

[20] "https://www.tensorflow.org/performance/performance_model s," .