2017 IEEE INTERNATIONAL WORKSHOP ON MACHINE LEARNING FOR SIGNAL PROCESSING, SEPT. 25–28, 2017, TOKYO, JAPAN

A LAYER-BLOCK-WISE PIPELINE FOR MEMORY AND BANDWIDTH REDUCTION IN DISTRIBUTED DEEP LEARNING

Haruki Mori[1], Tetsuya Youkawa[1], Shintaro Izumi[1], Masahiko Yoshimoto[1], Hiroshi Kawaguchi[1], and Atsuki Inoue[2]

[1]Graduate School of System Informatics, Kobe University, Kobe, Japan
[2] Fujitsu Laboratories Ltd. Computer Systems Laboratory, Kawasaki, Japan
E-mail:mori.haruki@cs28.cs.kobe-u.ac.jp

ABSTRACT

This paper describes a pipelined stochastic gradient descent (SGD) algorithm and its hardware architecture with a memory distributed structure. In the proposed architecture, a pipeline stage takes charge of multiple layers: a "layer block." The layer-block-wise pipeline has much less weight parameters for network training than conventional multithreading because weight memory is distributed to workers assigned to pipeline stages. The memory capacity of 2.25 GB for the four-stage proposed pipeline is about half of the 3.82 GB for multithreading when a batch size is 32 in VGG-F. Unlike multithreaded data parallelism, no parameter server for weight update or shared I/O data bus is necessary. Therefore, the memory bandwidth is drastically reduced. The proposed four-stage pipeline only needs memory bandwidths of 36.3 MB and 17.0 MB per batch, respectively, for forward propagation and backpropagation processes, whereas four-thread multithreading requires a bandwidth of 974 MB overall for send and receive processes to unify its weight parameters. At the parallelization degree of four, the proposed pipeline maintains training convergence by a factor of 1.12, compared with the conventional multithreaded architecture although the memory capacity and the memory bandwidth are decreased.

Index Terms— *Deep neural network, Model Parallelism, Pipelined backpropagation, Distributed memory, Memory capacity reduction, Memory bandwidth reduction*

1. INTRODUCTION

The perceptron, a primitive artificial neural network, has a single layer comprising input synapses and output neurons as nonlinear activations [1]. The multilayer perceptron is an extension of the single-layer perceptron with hidden layers, in which training is conducted through backpropagation [2].
A convolutional neural network (CNN) imitates part of the human visual cortex in the cerebrum. It is an extension of a multilayer perceptron. Originally, the CNN was developed for handwritten character recognition, and was named "neocognitron" [3]. Actually, CNNs have been scaled up with numerous synapses and neurons in deeper layers. Recently, a deeper network having more than three layers is generally called as a "deep neural network (DNN)" or "deep learning." The DNN has exhibited its potential for image recognition ability. Its accuracy is improving year by year. At the ImageNet Large Scale Visual Recognition Competition (ILSVRC), AlexNet with five CNN layers and three fully connected layers made an overwhelming achievement over conventional feature-based image recognition schemes in 2012 [4]. Its top-five error rate was 15.3%, which was more than 10% better than the second-best entry based on the handmade features. Since 2012, the error rate in image recognition has been improved by DNNs. At ILSVRC 2015, ResNet, comprising with 151 CNN layers and one fully connected layer, remarkably won the competition by a top-five error rate of only 3.57% [5], which is better than the 5.1% figure representing human ability [6]. Today, DNNs are applied mainly to image recognition applications, but DNNs themselves has general-purpose characteristics and abilities; DNNs are now attracting attention not only in for engineering, but also for use in medicine, pharmacy, and biology applications [7].

As DNNs have generality with a deeper and larger-scale network, their error rates of cognition continue to improve. Accordingly, computational times become much longer, particularly those for training purpose. AlexNet took 5–6 days to train 90 epochs of 1.2-M ImageNet picture data-sets on two NVIDIA GTX580 GPUs [5]. Also, ResNet-200 (ResNet with 200 layers) designed for ImageNet, took three weeks for its training, even with eight GPGPUs used in parallel computing [8].

There are two concepts of parallelism to shorten the training time for an enormous network [9]:

- Data parallelism has divided dimensions of data. Each worker trains on the same network but with a different data example.

- Model parallelism has divided dimensions of a model (network). Each worker trains a different part of the model (network).

Mini-batch stochastic gradient descent (SGD) has faster in error rate convergence than pure SGD because a matrix-matrix operation can be better optimized than a matrix-vector one. Multithreaded mini-batch SGD is often exploited as a data parallelism for additional speeding up. Deploying homogeneous workers and implementing the same software for them is simple. Each worker has the same network, but processes a different mini batch. In other words, a single network is trained with different mini batches. Each worker updates different weight parameters. All workers must unify their own weights. The unified weights are usually obtained by averaging their own weights received from all workers. Then the unified weights are sent back to the workers for the next mini-batch step. The weight unification and replication are applied repeatedly. They invariably consume memory bandwidth. As the number of workers is increased, memory bandwidth turns out to be linearly wider [10, 11]. In terms of memory capacity, each worker must hold all weights and activations of a whole network in the multithreaded mini-batch SGD. The multithreaded mini-batch SGD tends to be less effective in convergence than a single-threaded one because its effective batch size is multiplied by the data parallelism. Therefore, updates per epoch result in a lower number [12–14]. To reduce the memory bandwidth and capacity and to maintain scalability of parallelism, the model parallelism is useful. Of course, the model parallelism can be combined with data parallelism.

Pipelined backpropagation with distributed memory has been studied for decades as a kind of data parallelism. In the 1990s, node parallelism [15] and layer-based pipelined backpropagation schemes [16, 17] were proposed for epoch learning. Since 2010, a pipelined DNN combined with a hidden Marcov model has been proposed for speech recognition to shorten the GPGPU training time [18−20]. As described in this paper, we revisit the use of pipelined DNN for image recognition with ImageNet, and propose another model parallelism with layer blocks. The proposed layer-block-wise pipeline with segmented bus architecture suppresses whole memory capacity and transfer memory bandwidth to maintain scalability of parallelism.

This paper is organized as follows: Section 2 presents the layer-block-wise pipeline algorithm with software implementation. Hardware architecture and its performance are explained in Section 3. The final section presents discussion of issues in the proposed pipeline.

2. LAYER-BLOCK-WISE PIPELINE

Fig. 1 depicts the proposed layer-block-wise pipeline as a conceptual diagram. In the figure, each pipeline stage with multiple layers has a worker for both forward propagation and backpropagation. A worker takes charge of one or more layers called “layer blocks”, shown in square model. The layer-block-wise pipeline is categorized as a kind of model parallelism. Its layer dimensions are divided by m or a smaller integer (where m is the number of layers including a layer for error calculation scheme). The number of pipeline divisions depends on the DNN models. Each worker executes a different task in parallel. A worker keeps a single weight matrix corresponding to its own layer network.

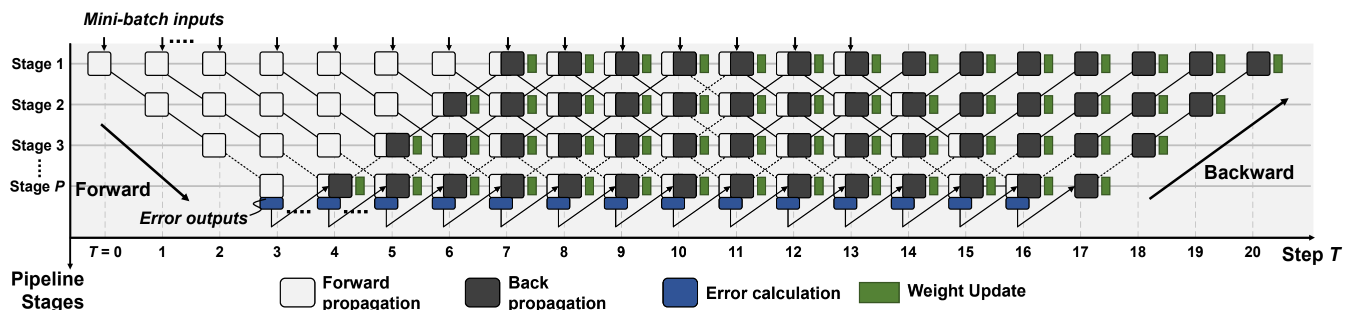

Fig. 1. Conceptual data-flow diagram of the proposed layer-block-wise pipeline with weight update latency.

Hereinafter, parameters P, T, and S respectively denote the number of pipeline stages (number of layer blocks), a current mini-batch step, and a current pipeline stage (current layer block). In each pipeline stage, forward propagation is conducted on a stage-by-stage basis. After certain latency, backpropagation is executed with corresponding activations; then weights are updated. Therefore, each pipeline stage processes a different but consecutive mini batch, and respectively propagates its activations and deltas down and up simultaneously to adjacent pipeline stages.

In forward propagation, the $(T-S+1)$-th mini batch is processed at pipeline stage S. Its activations are transferred down to the next pipeline $S+1$. Finally, a mini batch is processed at the last pipeline stage P, where errors are calculated. The errors are going to be backpropagated at a next time step $T+1$.

In backpropagation, the $(T-2P+S)$-th mini batch is processed at a pipeline stage S. It is noteworthy that a worker at the pipeline stage S must save $2P-2S+2$ datasets of forward activations for the current and upcoming

backpropagations. Deltas are calculated with the oldest dataset of activations. Weights are updated with the deltas. The deltas in the shallowest layer in the pipeline stage S are transferred up to the upstream pipeline stage $S-1$ for a next time step $T+1$. In this manner, plural mini batches are propagated back and forth simultaneously without waiting for a naive parameter update. The weights are updated with a latency of $2P-2S+1$. It can be said that the proposed pipeline has a concept of approximate computing instead of the naive SGD.

To evaluate the accuracy and to verify training convergence in the proposed layer-block-wise pipeline model, we implemented Algorithm 1 with MatConvNet [21]. N signifies the total number of input mini-batch steps (the total number of time steps for input mini batches). The input is mini-batch data. In forward propagation, a vector of activations $Y_{S,\,x}$ for a pipeline stage S is calculated first, where x is a dataset of activations to be saved ($0 \le x \le 2P-2S+2$). After forward propagation is completed, an error vector is prepared as dY_{P+1} for backpropagation. Then, a vector of delta dY_S and a matrix of delta weights dW_S for a pipeline stage S are calculated. A matrix of weights W_S is updated with dW_S.

To demonstrate the layer-block-wise pipeline, we adopted VGG-F as a network model [20]. VGG-F has five CNN layers and three fully-connected layers, as presented in Fig. 2(a), which classifies 1,000 categories. Figs. 2(b)-(d) present two-stage, four-stage, and eight-stage pipeline cases.

Algorithm 1. Software implementation of the proposed layer-block-wise pipeline.

Input: $\text{MiniBatchInput}_0 \ldots \text{MiniBatchInput}_{N-1}$
Output: $W_1 \ldots W_P$
1: **for** $T = 0 \ldots N+2P-2$ **do**
2: $\quad Y_{0,\,\text{mod}(T/2P)} = \text{MiniBatchInput}_T$
3: $\quad$ **for** $S = 1 \ldots P$ **do**
4: $\quad\quad Y_{S,\,\text{mod}((T-S+1)/(2P-2S+2))} = \text{Forward}(Y_{S-1,\,\text{mod}((T-S+1)/(2P-2S+4))}, W_S)$
5: $\quad$ **end for**
6: $\quad dY_{P+1} = \text{Error}(Y_{P,\,\text{mod}((T-P+1)/2)})$
7: $\quad$ **for** $S = P \ldots 1$ **do**
8: $\quad\quad [dY_S, dW_S] = \text{Backward}(dY_{S+1}, Y_{S,\,\text{mod}((T-2P+S)/(2P-2S+2))}, W_S)$
9: $\quad\quad W_S = \text{Update}(W_S, dW_S)$
10: $\quad$ **end for**
11: **end for**

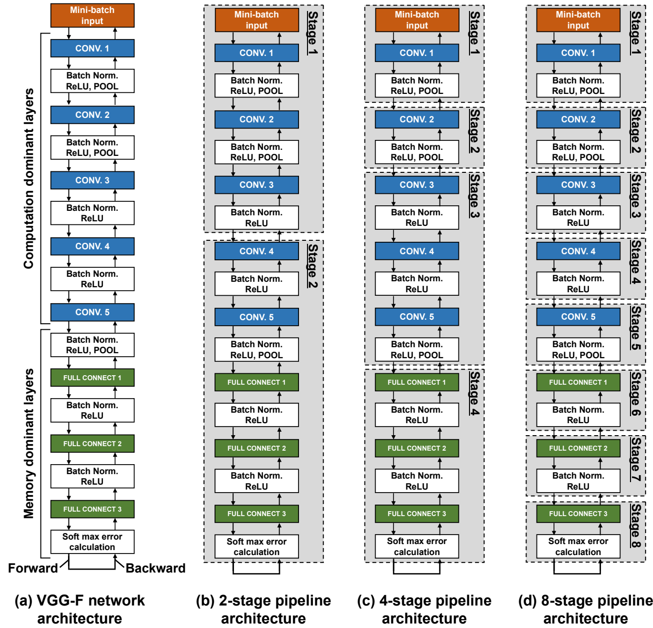

Fig. 2. Partitioning variations for VGG-F in the layer-block-wise pipeline.

3. HARDWARE IMPLEMENTATION

3.1. Hardware model

The main purpose of this paper is to reduce memory bandwidth and capacity for scalability of parallelism. Memory performance gives large impacts to speedup. We evaluate the hardware performance using the bus models. The layer-block-wise pipeline potentially reduces weight parameters and memory bandwidth on an I/O data bus. Fig. 3 presents a typical multithreaded SGD architecture. In this model, each processing unit has the same network model duplicated for multithreading. The dedicated parameter server for weight update is on a shared I/O data bus to communicate with the processing units. Each processing unit holds weights W and delta weights dW in internal memory. Actually, dW becomes 243.4 MB per processing unit in VGG-F. This amount of memory is pushed to and pulled from the parameter server by a DMA controller. An important issue related to the multithreaded architecture is data traffic concentration on the shared I/O data bus. The memory bandwidth comes to 243.4×n MB at every mini-batch step (where n is the number of workers). The sheared bus brims over with communication among multiple

workers, which has restricted system throughput in data parallelism [10-14].

Fig. 4 depicts a model for the layer-block-wise pipeline with distributed memory and segmented I/O data buses. The proposed architecture divides a network across a layer dimension. Layers are put together to an arbitrary number of blocks. Each worker performs different tasks in parallel. The segmented I/O data bus is used for communication only between two adjacent workers. The bus direction is always fixed to a single side (a sender side or a receiver side). Each worker receives and sends partial activations Y in forward propagation; again, each worker receives and sends partial deltas dY in the backpropagation. No communication exists on weights W and or delta weights dW. The layer-block-wise pipeline prevents traffic concentration and improves memory bandwidth. Delay to external data communication depends on a transfer data size. TABLE I and TABLE II show the respective memory performance comparison between the multithreaded SGD architecture and the proposed layer-block-wise pipeline. It is noteworthy that, in the layer-block-wise pipeline, the memory capacity and memory bandwidth for activations and deltas are scaled up linearly with a batch size, although they are reasonable values at a typical batch size of 32 ($BS = 32$).

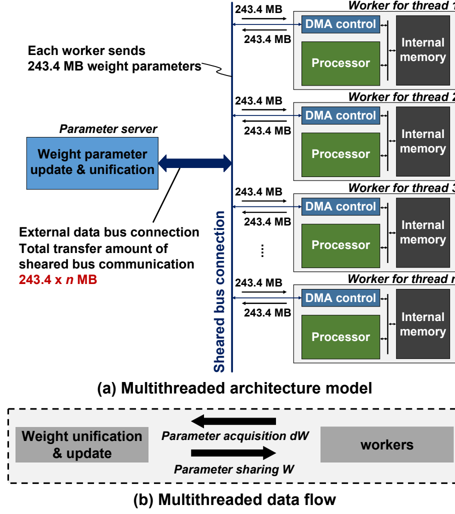

Fig. 3. (a) Architectural model of shared-bus multithreading and (b) its data flow.

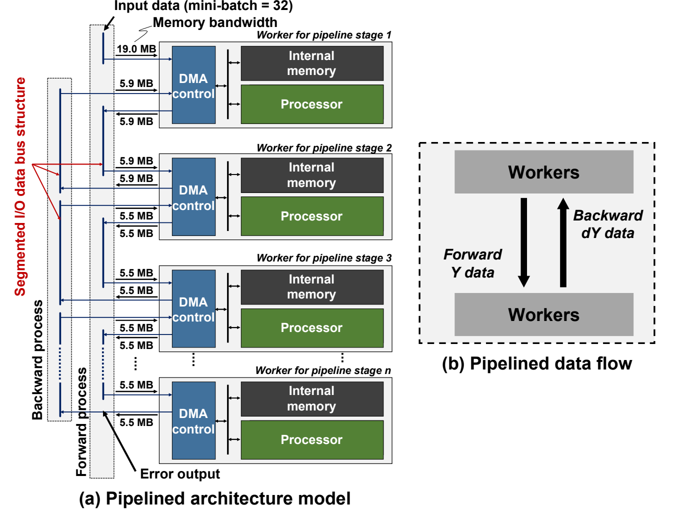

Fig. 4. (a) Architectural model of the layer-block-wise pipeline and (b) its data flow.

TABLE I
Memory capacity and memory bandwidth at four-degree of multithreads

4-degree of multithreads	Internal memory capacity						Memory bandwidth		
	Weight parameter memory [MB]		I/O data memory [MB]				Status	Transfer data amount [MB]	
			BS = 1		BS = 32			BS = 1	BS = 32
	W	dW	Y	dY	Y	dY		W / dW	W / dW
Thread 1	243.45	243.45	7.31	7.31	233.92	233.92	Receive / Send	243.45 / 243.45	243.45 / 243.45
Thread 2	243.45	243.45	7.31	7.31	233.92	233.92	Receive / Send	243.45 / 243.45	243.45 / 243.45
Thread 3	243.45	243.45	7.31	7.31	233.92	233.92	Receive / Send	243.45 / 243.45	243.45 / 243.45
Thread 4	243.45	243.45	7.31	7.31	233.92	233.92	Receive / Send	243.45 / 243.45	243.45 / 243.45
Sub total	973.80	973.80	29.24	29.24	935.68	935.68	Total	973.8 / 973.8	973.8 / 973.8
Total	2,006.08				3,818.96				

TABLE II
Memory capacity and memory bandwidth at four-stage layer-block-wise pipeline

4-stage pipeline	Internal memory capacity						Memory bandwidth					
							Forward process			Backward process		
	Weight parameter memory [MB]		I/O data memory [MB]				Status	Transfer data amount [MB/batch]		Status	Transfer data amount [MB/batch]	
			BS = 1		BS = 32			BS = 1	BS = 32		BS = 1	BS = 32
	W	dW	Y	dY	Y	dY						
Stage 1	0.09	0.09	24.22	3.02	775.04	96.64	Receive	0.60	19.26	Receive	0.19	5.98
							Send	0.19	5.98	Send	-	-
Stage 2	1.64	1.64	14.47	2.41	463.04	77.12	Receive	0.19	5.98	Receive	0.17	5.53
							Send	0.17	5.53	Send	0.19	5.98
Stage 3	7.09	7.09	6.92	3.46	221.44	110.72	Receive	0.17	5.53	Receive	0.17	5.53
							Send	0.17	5.53	Send	0.17	5.53
Stage 4	234.61	234.61	0.28	0.28	8.96	8.96	Receive	0.17	5.53	Receive	-	-
							Send	-	-	Send	0.17	5.53
Sub total	243.45	243.45	45.89	9.17	1,468.48	293.44	Receive total	1.13	36.30	Receive total	0.53	17.04
Total	541.96				2,248.82		Send total	0.53	17.04	Send total	0.36	17.04

Fig. 5 portrays memory bandwidth trends in the multithreaded SGD. A memory bandwidth of 974.0 MB is required overall for send and receive processes to unify the weight parameters when a parallelization degree is four. Fig. 6 presents memory bandwidth trends against the batch size in the layer-block-wise pipeline. The memory bandwidth per batch increases linearly with increasing mini-batch size. It is noteworthy that layer-block-wise pipeline has different values of memory bandwidth on forward and backward processes. As described above, our target batch size is 32, in which case the memory bandwidth both forward propagation and back propagation are, respectively, 36.3 MB and 17.0 MB. The total memory bandwidth for the both directions does not exceed 100.0 MB.

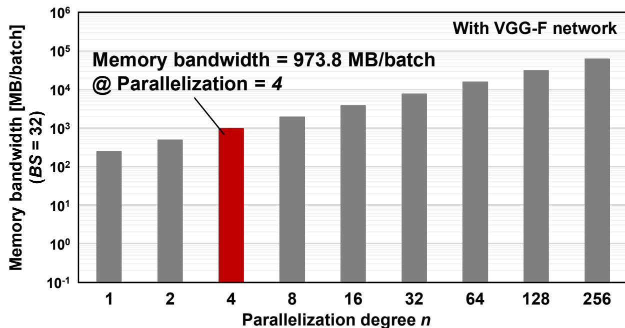

Fig. 5. Memory bandwidth trends against the parallelization degree in multithreading.

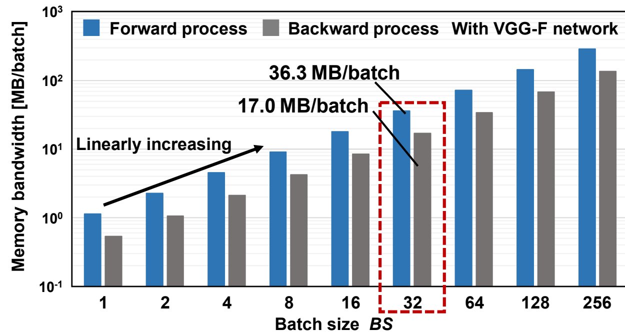

Fig. 6. Memory bandwidth trends against the batch size in the layer-block-wise pipeline.

3.2. Performance evaluation

Fig. 7 shows a comparison of training convergences for the naive SGD and the proposed layer-block-wise pipeline. The convergence is the time when the miss rate comes to 75%. A 50,000-image dataset is used for training VGG-F; 50 images per category are sampled from the ImageNet dataset. The layer-block-wise pipeline is 2.0 times faster in the two-stage pipeline, and is 3.5 times faster in the four-stage pipeline. With eight pipeline stages, the acceleration factor is saturated to 5.1. Fig. 8 presents a convergence speed comparison. For a parallelization degree of four or fewer, the layer-block-wise pipeline maintain almost same convergence as multithreading.

Fig. 9 presents acceleration factors when the memory capacity is varied. The internal memory amount depends on the parallelization degree. In multithreading, the memory capacity increases linearly with the parallelization degree, although such is not the case in the proposed pipeline (only activations and deltas are increased. The memory capacity for weights is not changed). Therefore, the proposed pipeline has less memory than the multithread for the same parallelization degree. The layer-block-wise pipeline has 41% less memory when the parallelization degree is four, with better acceleration performance per unit of memory capacity: 2.25 GB for pipeline and 3.82 GB for multithread.

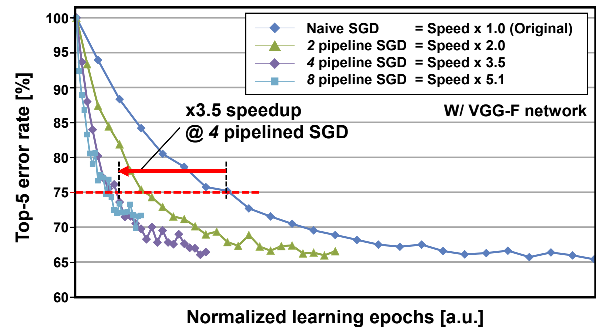

Fig. 7. Training convergence comparison between the multithreaded SGD and the layer-block-wise pipeline.

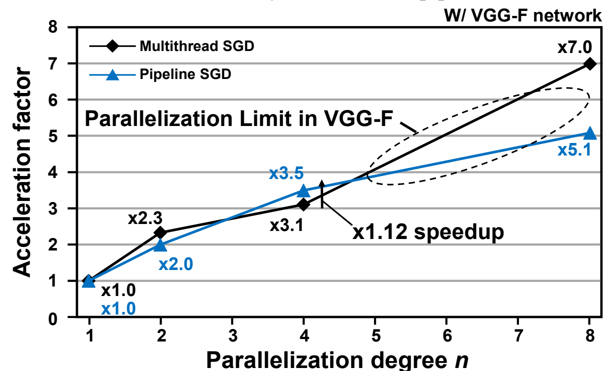

Fig. 8. Training convergence comparison for parallelization degrees of one, two, four, and eight.

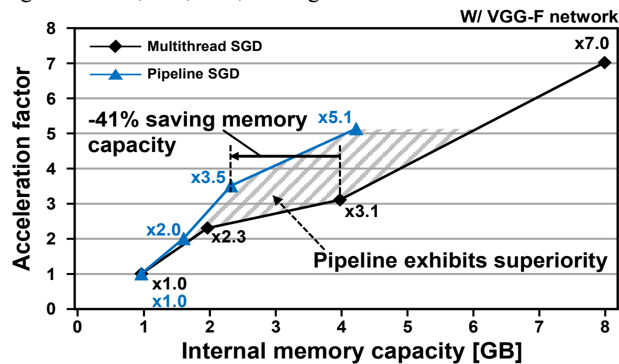

Fig. 9. Training convergence in the conventional multithreaded SGD and the proposed layer-block-wise pipeline.

4. DISCUSSION

Much deeper networks such as ResNet [8] are attracting attention for more accurate recognition. In multithreading,

higher memory bandwidth and more memory capacity will be necessary for deeper networks. Hardware costs and computation time will increase continuously. Realizing fast yet low-cost hardware is expected even if the network goes deeper. We have demonstrated superiority of our pipeline architecture in terms of hardware cost reduction, but how to partition a network remains as an issue to be tackled for a more effective pipeline. Layers in the VGG-F network show large variation in computation. Convolutional layers dominantly require many more operations than other layers such as a fully connected layer, a normalization layer, pooling layers, and an activation layer. Even in the convolutional layers in VGG-F, computations show wide variation. To synchronize the pipeline stages strictly and to avoid division loss of the pipeline, heterogeneous architecture with dedicated processors is ideal. However, such a system is nonsensical and virtually impossible. Fortunately, in ResNet, variation in convolutional layers is small. It will be easier to balance pipeline stages in homogeneous workers. Another approach to balance workloads in a pipeline is stochastic computing. Computation in a heavier pipeline stage can be reduced stochastically [22].

ACKNOWREDGEMENT

This study was carried out by the research grant from Fujitsu Laboratories Ltd.

REFERENCES

[1] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, no. 6, pp. 386-408, Nov. 1958.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533-536, Oct. 1986.

[3] K. Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biological Cybernetics*, vol. 36, no. 4, pp. 93-202, Apr. 1980.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Proceedings of Neural Information Processing Systems* (NIPS), pp. 1097-1105, Dec. 2012.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (CVPR), pp. 770-778, June 2016.

[6] A. Karpathy, "What I learned from competing against a ConvNet on ImageNet," Blog at http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436-444, May 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," *Proceedings of European Conference on Computer Vision* (ECCV), *arXiv*:1603.05027, July 2016.

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

[10] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M.'A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," *Proceedings of Neural Information Processing Systems* (NIPS), pp. 1223-1231, Dec. 2012.

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *arXiv*:1603.04467, Mar. 2016.

[12] Y. Bengio, "Practical Recommendations for Gradient-Based Training of Deep Architectures," *arXiv*:1206.5533, Sep. 2012.

[13] S. Gupta, W. Zhang, F. Wang "Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study," *Proceedings of IEEE International Conference on Data Mining* (ICDM), *arXiv*:1509.04210, Dec. 2016.

[14] J. Keuper, and F.-J. Pfreundt, "Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability," *arXiv*:1609.06870, Dec. 2016.

[15] H. Yoon, J. H. Nang, and S. R. Maeng, "A Distributed Backpropagation Algorithm of Neural Networks on Distributed-Memory Multiprocessors," *Proceedings of Symposium on the Frontiers of Massively Parallel Computation*, pp. 358-363, Oct. 1990.

[16] A. Petrowski, G. Dreyfus, and C. Girault "Performance Analysis of a Pipelined Backpropagation Parallel Algorithm," *IEEE Transactions on Neural Networks*, vol. 4, no. 6, pp. 970-981, Nov 1993.

[17] S. Zickenheiner, M. Wendt, B. Klauer, and K. Waldschmidt, "Pipelining and Parallel Training of Neural Networks on Distributed-Memory Multiprocessors," *Proceedings of IEEE International Conference on Neural Networks*, pp. 2052-2057, June 1994.

[18] K. Vesely, L. Burget, and F. Grezl, "Parallel Training of Neural Networks for Speech Recognition," *Proceedings of IEEE International Conference on Neural Networks*, pp. 2934-2937, Sep. 2010.

[19] F. Seide, G. Li, and D. Yu, "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks," *Proceedings of ISCA Interspeech*, pp. 437-440, Aug. 2011.

[20] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide, "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks," *Proceedings of ISCA Interspeech*, pp. 26-29, Sep. 2012.

[21] *MatConvNet* at http://www.vlfeat.org/matconvnet/.

[22] G. Huang, Y. Sun, Z. Liuy, D. Sedra, and K. Q. Weinberger, "Deep Networks with Stochastic Depth," *arXiv*:1603.09382, July 2016.