MeDNN: A Distributed Mobile System with Enhanced Partition and Deployment for Large-Scale DNNs

Jiachen Mao†, Zhongda Yang‡, Wei Wen†, Chunpeng Wu†, Linghao Song†,
Kent W. Nixon†, Xiang Chen¤, Hai Li†, and Yiran Chen†
†Duke University, USA; ‡University of Pittsburgh, USA; ¤George Mason University, USA
†{jiachen.mao, wei.wen, chunpeng.wu, linghao.song, kwn8, hai.li, yiran.chen}@duke.edu;
‡zhy47@pitt.edu; ¤xchen26@gmu.edu

***Abstract*—Deep Neural Networks (DNNs) are pervasively used in a significant number of applications and platforms. To enhance the execution efficiency of large-scale DNNs, previous attempts focus mainly on client-server paradigms, relying on powerful external infrastructure, or model compression, with complicated pre-processing phases. Though effective, these methods overlook the optimization of DNNs on distributed mobile devices. In this work, we design and implement *MeDNN*, a local distributed mobile computing system with enhanced partitioning and deployment tailored for large-scale DNNs. In *MeDNN*, we first propose Greedy Two Dimensional Partition (*GTDP*), which can adaptively partition DNN models onto several mobile devices w.r.t. individual resource constraints. We also propose Structured Model Compact Deployment (*SMCD*), a mobile-friendly compression scheme which utilizes a structured sparsity pruning technique to further accelerate DNN execution. Experimental results show that, *GTDP* can accelerate the original DNN execution time by 1.86 – 2.44× with 2 – 4 worker nodes. By utilizing *SMCD*, 26.5% of additional computing time and 14.2% of extra communication time are saved, on average, with negligible effect on the model accuracy.**

I. Introduction

With explosive growth in computational capability and network bandwidth, mobile devices have become a hotspot for innovation in interactive multimedia applications. Examples of this include intensive speech recognition and object classification tasks. Deep Neural Networks (DNNs) are the underlying technology enabling these applications, and have been widely studied and utilized due to their high accuracy, scalability, and reconfigurable flexibility [1][2].

However, the size and complexity of DNNs incurs significant performance overhead. This is exacerbated on mobile devices due to limitations in terms of memory consumption and computing capability. For example, *AlexNet*, a typical DNN model for image recognition, requires about 2 seconds for a single model forwarding procedure on Google Nexus 5, which is unacceptable in practical use. Many research works have been proposed to resolve the conflict between the massive workload for large scale DNN execution and the limited resources available to mobile computing. One straightforward approach is to expand computing capability through a *client-server* paradigm. For example: In [3], Hauswald *et al.* proposed a data offloading scheme in a pipelined machine learning structure; In [4], Li *et al.* established an efficient distributed parameter server framework for DNNs. Another approach is *model compression*: In [5], Han *et al.* deeply compressed DNN models using a three-stage pipeline: pruning, trained quantization, and Huffman coding; In [6], Chen *et al.* introduced a low-cost hash function to group weights into hash buckets for parameter sharing purpose.

While these approaches are quite effective, they are not without their associated drawbacks. The *client-server* paradigm requires costly external infrastructure support, as well as a continuous network connectivity to the mobile device for transmitting raw data and classification results. Meanwhile, *model compression* requires complex model pre-processing and incurs potential accuracy drop.

To tackle the aforementioned problems, we propose *MeDNN* – a local distributed mobile computing system with enhanced DNN partition scheme (*GTDP*) and deployment scheme (*SMCD*).

Concretely, our major contributions include:

- We model our system and present the corresponding analysis on the relationship between tested mobile computing resources and DNN execution performance in a mobile network scenario;
- We propose a DNN model partition scheme for *MeDNN*, namely *GTDP*, to dispatch the workload to heterogeneous mobile devices for maximal execution parallelism;
- We propose a pruning scheme – *SMCD* with particular *group-lasso* regularization, which allows for more efficient and compact DNN partition and deployment on mobile platforms;
- We implement *MeDNN* on a cluster of Nexus 5 devices and test it with a state-of-the-art DNN model. A series of comprehensive analysis is given w.r.t. major mobile computing components.

Experimental results show that, when the number of worker nodes increases from 2 to 4, *GTDP* can speedup the original DNN execution time by 1.86-2.44×. With *SMCD* scheme, extra 26.5% of computing time and 14.2% of communication time are reduced, respectively. By incorporating *GTDP* and *SMCD*, *MeDNN* is able to achieve at most 3.76× speedup with 4 worker nodes compared with local execution.

II. Preliminary

A. DNN Performance on Mobile Devices

From the perspective of the DNN structure, the considerable computing time is mainly introduced by convolutional layers. In each convolutional layer, a set of filters are convolved across the feature map of the previous layer to generate the input of the next layer. In this paper, the terms *inputs* and *weights* are interchangeable with the terms *feature map* and *filter* in a single layer of DNN, respectively. From the perspective of the actual mobile computing process, considerable time is consumed by intensive memory operation and CPU computation due to the limited computing resource. Because the convolutional layers are the major computing overhead of DNNs on mobile devices, we mainly focus on the partition and deployment of convolutional layers in this paper. In the later system modeling section, we will conduct a series of preliminary tests and analysis to identify the system optimization targets and mobile computation constraints for the convolution step in DNNs.

B. Distributed Mobile Network

MapReduce is considered an excellent distributed architecture for simplifying parallel data processing for DNN execution [7]. Its effectiveness has been proven in many machine learning applications on mobile platforms [8][9]. The two key procedures in the *MapReduce* architecture are *Map* and *Reduce*. The *Map* procedure partitions a task to pieces that can be executed in parallel, while the *Reduce* procedure merges the intermediate data from the *Map* procedure. The implementation of *MapReduce* with the mobile devices can be achieved with a master-slave structure. In the network, one mobile device acts as the Group Owner (*GO*), in charge of testing data storage and partitioning. Additional devices act as Worker Nodes (*WN*s), and share the computing load with the *GO*.

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

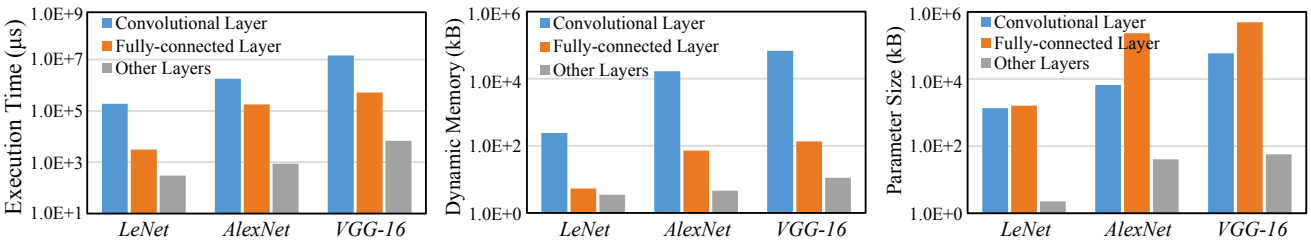


Fig. 1: Total execution time, dynamic memory allocation size, and model parameter size of three typical DNN models on Nexus 5.

C. Model Compression for DNNs

Many studies have been performed regarding model compression, e.g., sparsity regularization [10], connection pruning [11]. However, they have certain drawbacks to maintain structured DNN model connections effectively. When a non-structured DNN model is deployed on mobile devices, unpredictable memory access patterns are introduced to the limited memory capacity, leading to a negative impact on practical acceleration. As model structure is critical for compression performance, [12] was proposed to structurally decompose each weight tensor into a product of smaller factors. But it requires costly reiterations of decomposition and fine-tuning procedures to find the optimal weight approximation.

Therefore, *group lasso* is a more structurally regularized compression approach based on covariate grouping and selective sparsing [13][14]. In this work, we innovatively improved the *group lasso* regularization for distributed mobile system with mobile computing and transmission characteristics taken into consideration.

III. System Modeling

In this section, we establish the system model of *MeDNN* and a series of experiments are conducted to analyze the performance of DNN execution on our distributed mobile system.

A. Definition of Terminologies and Variables

We define the terminologies and variables as follows:

- **Total Worker Nodes (N):** Total number of the available mobile devices within the computing cluster;
- **Transmission Throughput ($T_{trans[i]}$):** Transmission throughput for *WN* i under WLAN;
- **Transmission Amount ($W_{trans[i]}$):** Total transmission amount from *WN* i to GO;
- **Transmission Delay ($D_{[i]}$):** Transmission delay for the first bit of the message sent from sender to receiver for *WN* i;
- **Computing Workload ($W_{comp[i]}$):** The partitioned computing workload to *WN* i, which is indicated by the number of Floating-point Operations (FLOP);
- **Computing Capability ($C_{comp[i]}$):** Floating-point operations per second (FLOPS) for *WN* i;
- **Memory Usage($W_{mem[i]}$):** Total size of runtime memory allocation and initialization for *WN* i;
- **Memory Bandwidth($M_{mem[i]}$):** The memory allocation bandwidth of *WN* i for dynamic allocation and intialization;

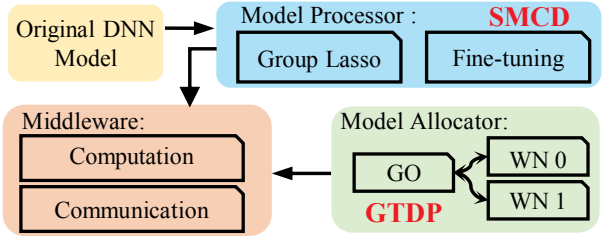


Fig. 2: System overview of *MeDNN*.

B. Distributed Mobile System Model

Based on the previous mentioned variables, the model of the total execution time on distributed mobile system can be formulated as following, which is also the core optimization target of *MeDNN*:

$$T_{total} = \text{Max}\{(\frac{W_{comp[i]}}{C_{comp[i]}} + \frac{W_{mem[i]}}{M_{mem[i]}} + \frac{W_{trans[i]}}{T_{trans[i]}} + D_{[i]})|_{i=1...N}\} \quad (1)$$

where $(W_{trans[i]}/T_{trans[i]} + D_{[i]})$ summarizes the total transmission time by adding the delay to the transmission amount divided by the throughput. The total computing time is expressed as the summation between $W_{comp[i]}/C_{comp[i]}$ and $W_{mem[i]}/M_{mem[i]}$, representing the time consumption for FLOP and memory usage. We can clearly tell from this formulation that the overall DNN execution time is bottlenecked by the maximum time among all the N *WN*s including the computing time and communication time. Furthermore, transmission amount (W_{trans}), computing workload (W_{comp}), and memory usage (W_{mem}) are the three core variables we want to optimize on.

C. Analysis of DNNs on Mobile Devices

To evaluate the performance of DNN models on mobile devices, a series of experiments are conducted using Google Nexus 5, which has a hexa-core 1.8GHz CPU, and 2GB of RAM. Three DNN models of incremental scale are analyzed: *Lenet* [15], *AlexNet* [16], and *VGG* [17]. Fig. 1 depicts the analysis results including: computing time of a single test, dynamic RAM memory size occupied by the DNN execution, and parameter size for model storage. The experimental results illustrate the major performance optimization target. Although fully-connected layers account for higher parameter size, both the time consumption and dynamic memory usage of convolutional layers in DNN outweigh that of fully-connected layers by at least 10.3×, with the effect of other layers (e.g., ReLu layers, pooling layers, normalization layers, etc.) remaining negligible. The reason lies in that, on resource-constrained mobile platforms, the on-demand memory requests and memory migration account for a large portion of the computing overhead. From our analysis, we verify that the convolutional layers are the bottleneck of the DNN execution.

IV. MeDNN System Design Overview

Based on the mobile device computing resource constraint analysis, we designed the *MeDNN* system. Fig. 2 presents an overview of *MeDNN* including three core components:

(1) Middleware: The distributed mobile network deployed on each mobile devices which sets up the computing cluster and schedules the data processing flow during parallel execution in *MeDNN*;

(2) Model Allocator: The *GTDP* scheme that partitions the original model using into multiple *WN*s with consideration of mobile device computing resource constrain.

(3) Model Processor: The *SMCD* scheme that prunes the original DNN model to fine-tuned model for more efficient deployment and execution on both single and distributed mobile system;

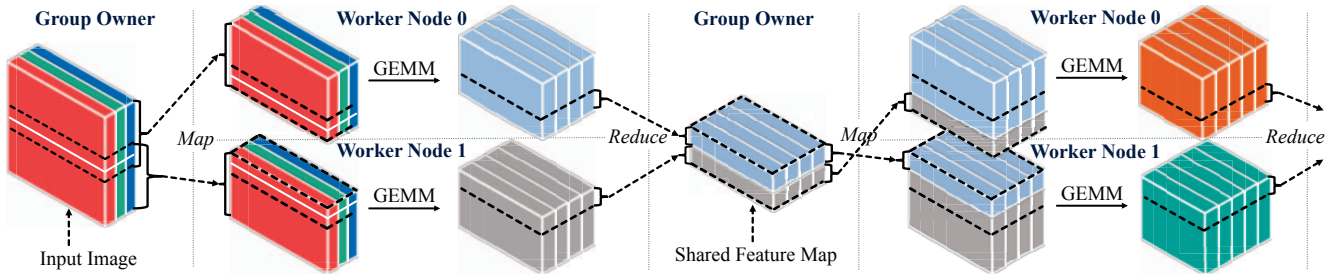


Fig. 3: Processing flow of *MeDNN* with optimized Mobile *MapReduce* on two *WN*s.

A. Dataflow in MeDNN

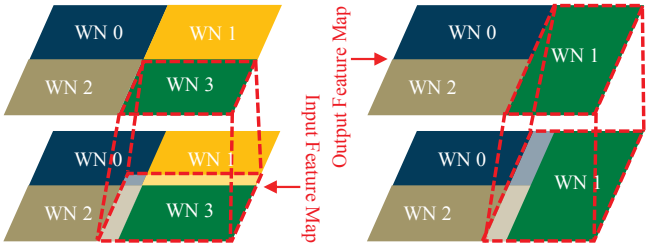
Fig. 3 demonstrates the whole dataflow of *MeDNN* execution with two *WN*s. Thanks to the strong data affinity in the shared feature map between *WN*s, a comparatively small part of the feature map is needed for transmission due to the small kernel size of convolutional layers, which is expressed as the cubes in dashed contours in Fig. 3. *MeDNN* utilizes the concept of *MapReduce* as the distributed architecture with several redesigned details for the low-level processing flow optimized to our DNN execution scenario. As shown in Fig. 3, *GO* first partitions the input image and maps them to each *WN*. Then, after the computation of the convolution operation, each *WN* reduces the shared part of the output feature map back to *GO* by key/value pairs. Finally, the *GO* gathers the output from the *WN*s and maps them back to the *WN*s so that each *WN* can combine the received data with their local output feature map as the input feature map for next convolutional layer. Targeting at higher execution efficiency, the intermediate data are not stored on local disk.

V. GTDP – Model Partition Scheme

In this section, we focus on the partition scheme in *MeDNN*, targeting at workload balance and lower transmission amount.

Conventional partition schemes for input feature maps usually maintain a structural symmetry, e.g., in [18], Coates *et al.* arranged a GPU cluster into 2D-grids and partitioned the inputs along the two-dimensional space, as shown in Fig. 4(a). However, partitioning the inputs for a set of heterogeneous mobile devices incurs two extra challenges: 1) In heterogeneous mobile system, each mobile device has various hardware profiles. 2) The number of *WN*s in a distributed mobile system is dynamically changing.

We note that the computing time of convolutional layers is primarily dependent on the size of the input feature map, which is experimentally verified in Section VII-C. Therefore, as shown in Fig. 4(b), we redefine the workload as the area of the input feature map divided by the relative computing capability of each *WN*. Under the precondition of the available *WN*s with their corresponding workload ratio, our partitioning scheme aims to minimize the communication amount by identifying the shortest combination of lines which split the feature map between each two *WN*s.



(a) Traditional symmetry partition. (b) Proposed partition scheme.

Fig. 4: Two partition schemes for convolutional layers.

Algorithm 1 Greedy Two Dimensional Partition scheme (GTDP).

Input: Original feature map M of height and width: $H \times W$, total *WN*s N, computing capabilities: $C_{comp[i]}$, $(i = 1, 2...N)$

Initialization: The partition result: $P_{[i]}$, $(i = 1, 2...N)$, Index set to be partitioned: $Idx_{[i]}$, $(i = 1, 2...N)$.

procedure INT PARTITION(M)
  if $N <= 1$ **then**
    return 0
  if $N == 2$ **then**
    Split the feature map by the shorter edge and assign them to $P_{[Idx[1]]}$ and $P_{[Idx[2]]}$ with the ratio of $\frac{C_{comp[Idx[1]]}}{C_{comp[Idx[2]]}}$.
    return $H > W \ ? \ W : H$
  $result = Infinite$
  for each int $i \in [1, N/2]$ **do**
    for each enumerate of C_N^i combinatorics: $Enum$ **do**
      ratio = $\frac{\sum C_{comp[x]}}{\sum C_{comp[y]}}$ $x \in Enum$ $y \in Idx - Enum$
      $r1 = Partition(M_{left}) + Partition(M_{right}) + H$
      $r2 = Partition(M_{top}) + Partition(M_{btm}) + W$
      $result$ = Min($r1$, $r2$, $result$)
      Update $P_{[Idx]}$ and $P_{[Idx-Enum]}$ based on $result$
  return $result$

Output: The optimal partition result: $P_{[i]}$, $(i = 0, 2...N)$

The partitioning scheme – Greedy Two Dimensional Partition (*GTDP*) for the input feature maps of DNN is illustrated in Algorithm 1. With a certain area splitting ratio derived from computing capability of the *WN*s, our proposed partition scheme exhausts all the potential layouts and greedily finds the optimal partition strategy where the total length of the dividing lines is minimized. It is worth noting that the partition scheme only need to be executed once for a certain computing cluster, which will not affect run-time performance.

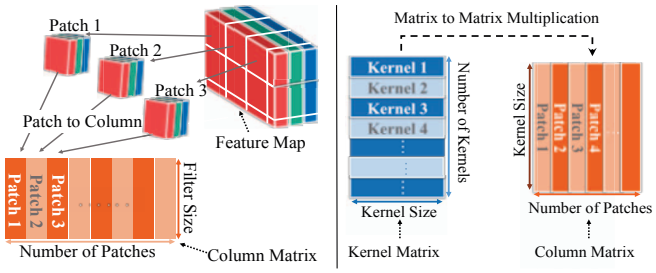
VI. SMCD – Model Pruning Scheme

In this section, we focus on the pruning scheme in *MeDNN*, targeting enhanced performance in node-wise computing capability and transmission efficiency.

In order to boost the efficiency of DNN execution on mobile devices, we explore and exploit the property of sparse DNNs for efficient deployment on mobile devices. Specifically, to take advantage of the predictable redundancy existing in large-scale DNNs, Structured Model Compact Deployment (*SMCD*) is proposed. *SMCD* structurely zero-out the model weights during the model training procedure by learning the inner structure of DNN models, which jointly reduces DNNs computing workload and communication amount.

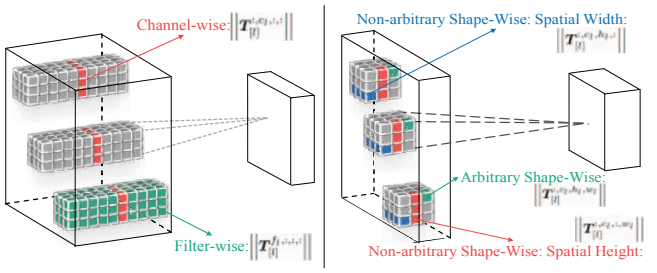
A. SMCD with Group Lasso

The weights of convolutional layers can be represented as a collection of 4-D tensors: $\boldsymbol{T}_{[l]} \in \mathbb{R}^{F_l \times C_l \times H_l \times W_l}$, where F_l, C_l, H_l,



(a) Patch to column step. (b) Matrix multiplication step.

Fig. 5: Two steps in GEMM.



(a) Row-wise group lasso. (b) Column-wise group lasso.

Fig. 6: Row-wise and column-wise group lasso on kernel matrix.

and W_l denote the filter number, channel size, spatial height, and spatial width of the l-th layer in the model of L total layers. To learn the structured sparsity of the DNN model, we define the new error function as:

$$E(\boldsymbol{T}) = \sum_{i=1}^{n} V(f(x_i), y_i) + \lambda R(\boldsymbol{T}) + \sum_{l=1}^{L} \lambda_g R_g(\boldsymbol{T}_{[l]}), \quad (2)$$

where function $V(\cdot)$ generates the loss induced by the training data corresponding to their labels and the term $\lambda R(\boldsymbol{T})$ represents the original regularization method (e.g., L2-norm) utilized to avoid the overfitting problem. $R_g(\cdot)$ is the additional structured sparsity regularization targeted at generating *SMCD* using group lasso. More specifically, $\lambda_g R_g(\boldsymbol{T}_{[l]}) = \lambda_g \sum_{g=1}^{G} \left|\left|\boldsymbol{T}_{[l]}^{g}\right|\right|$, meaning the sum of the G group lasso regularization terms in layer l with their corresponding weight decay λ_g. Mind that $\boldsymbol{T}_{[l]}^{g}$ represents a collection of partial tensors in group g of layer l. Here we denote $||\cdot||$ as the group lasso operator, which is defined as: $\left|\left|\boldsymbol{T}_{[l]}^{g}\right|\right| = \sqrt{\sum_{i=1}^{Size(\boldsymbol{T}_{[l]}^{g})} t_i^g}$, in which $Size(\boldsymbol{T}_{[l]}^{g})$ is the number of tensors in $\boldsymbol{T}_{[l]}^{g}$ and $t_i^g \in \boldsymbol{T}_{[l]}^{g}$.

Compared to previous works, the extra pruning phases are relatively simple. After training the DNN models with group lasso regularization for structured sparsity, we discard the group lasso penalty by fine-tuning the sparsified model with the connections of zeros being locked so as to reach a higher accuracy.

B. GEMM Operation on Mobile CPUs

In order to determine the tensors to be studied and group-wise sparsified, we should first characterize the computing overhead of convolutional layers in DNN. Modern deep learning libraries leverage GEneral Matrix to Matrix Multiplication (GEMM) to realize convolution operations due to its consistent memory access pattern and instruction level optimization. As shown in Fig. 5, GEMM consists of two steps: the patch to column step shown in Fig. 5(a) and the matrix multiplication step shown in Fig. 5(b). Patch to column steps reshape the 3D patches of the input feature map into 1D columns while matrix multiplication steps multiply the reshaped matrix with the filter matrix. From our preliminary experimental results, we find that when performed on mobile CPUs, patch-to-column steps consume 71.6% of the computing time while matrix multiplication steps consume the remaining 28.4%. So, we can conclude that the high execution time of DNNs is due to both the memory intensity of patch-to-column steps and computing intensity of matrix-to-matrix multiplication steps.

C. Reduce Computing Time with SMCD

To generate SMCD with reduced computing time, several groups of tensors are selected. From the perspective of the row-wise reduction of the filter matrix in Fig. 6(a), we introduce filter-wise and channel-wise group lasso, which is defined as:

$$\sum_{l=1}^{L} R_{g1}(\boldsymbol{T}_{[l]}) = \sum_{l=1}^{L} (\lambda_f \sum_{f_l}^{F_l} \left|\left|\boldsymbol{T}_{[l]}^{f_l,:,:,:}\right|\right| + \lambda_c \sum_{c_l}^{C_l} \left|\left|\boldsymbol{T}_{[l]}^{:,c_l,:,:}\right|\right|), \quad (3)$$

where $\boldsymbol{T}_{[l]}^{f_l,:,:,:}$ filter-wisely sparsifies the rows of the filter matrix of the current l_th layer. Conversely, $\boldsymbol{T}_{[l]}^{:,c_l,:,:}$ channel-wisely sparsifies the rows of the filter matrix of the $(l-1)\_th$ layer.

From the perspective of column-wise reduction of the filter matrix, we introduce an arbitrary shape-wise group lasso illustrated in Fig. 6(b), which is formulated as:

$$\sum_{l=1}^{L} R_{g2}(\boldsymbol{T}_{[l]}) = \sum_{l=1}^{L} \lambda_s \sum_{c_l}^{C_l} \sum_{h_l}^{H_l} \sum_{w_l}^{W_l} \left|\left|\boldsymbol{T}_{[l]}^{:,c_l,h_l,w_l}\right|\right|, \quad (4)$$

where the columns of the filter matrix are derived by $\left|\left|\boldsymbol{T}_{[l]}^{:,c_l,h_l,w_l}\right|\right|$.

By adding the group lasso terms in Eq. 3 and Eq. 4 to target error function Eq. 2, *SMCD* can be trained to be less computing-intensive using filter-wise & channel-wise group lasso and less memory-intensive with the help of the proposed arbitrary shape-wise group lasso. Suppose the sparsity ratios of a convolutional layer in DNN model are S_{row} and S_{column} for the rows and columns of the original filter matrix, respectively, about S_{column} of the memory usage and $S_{row} + S_{column} - S_{row} \times S_{column}$ of the floating-point operations (FLOP) would be saved for practical speedup in GEMM.

D. Reduce Communication Time with SMCD

Another critical target of *SMCD* is reducing the communication time in *MeDNN* by decreasing the transmission amount, or W_{trans} in Eq. 1. Inspired by the 4-D structure view of the weights of the convolutional layers, we can easily find that if the shared parts of tensors are zeroed out for all the filters in spatial height and spatial width, those tensors do not need to be transmitted, resulting in considerable decrease in transmission data size. Fig. 6(b) clarifies the rationale of our proposed non-arbitrary group lasso regularization, which can be formulated as:

$$\sum_{l=1}^{L} R_{g3}(\boldsymbol{T}_{[l]}) = \sum_{l=1}^{L} \lambda_{s\_h} \sum_{c_l}^{C_l} \sum_{w_l}^{W_l} \left|\left|\boldsymbol{T}_{[l]}^{:,c_l,:,w_l}\right|\right|, \quad (5)$$

$$\sum_{l=1}^{L} R_{g4}(\boldsymbol{T}_{[l]}) = \sum_{l=1}^{L} \lambda_{s\_w} \sum_{c_l}^{C_l} \sum_{h_l}^{H_l} \left|\left|\boldsymbol{T}_{[l]}^{:,c_l,h_l,:}\right|\right|. \quad (6)$$

For all the filters in each channel, our non-arbitrary group lasso regularization terms include the shape-wise group lasso along spatial height in Eq. 5 and the shape-wise group lasso along spatial width in Eq. 6. It is worth noting that our proposed non-arbitrary shape group lassos in Eq. 5 and Eq. 6 are independent of each other, the selection

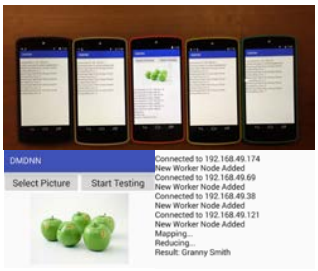


Fig. 7: System implementation.

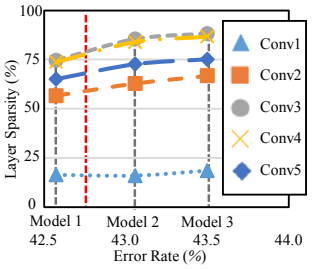


Fig. 8: Model configurations.

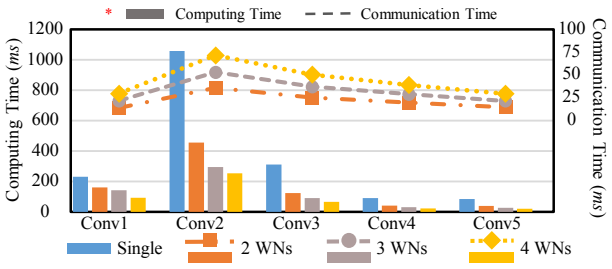


Fig. 9: Computing time and communication time of *CaffeNet*.

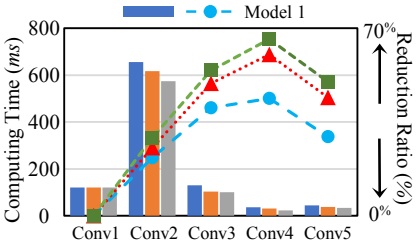


Fig. 10: Memory reduction (W_{mem}).

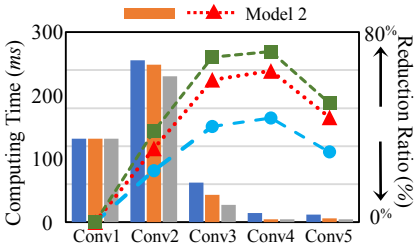


Fig. 11: FLOP reduction (W_{comp}).

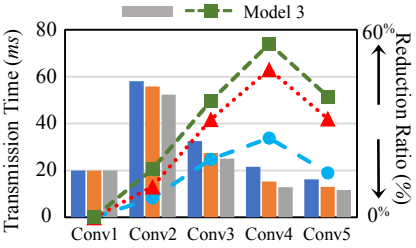


Fig. 12: Transmission reduction (W_{trans}).

of which term to be used depends largely on how we partition the input feature maps (e.g. by columns or by rows). Furthermore, the reduction in transmission size of *SMCD* is a stand-alone method, which can be combined with other data compression algorithms to further minimize the transmission amount.

VII. Experiment and Evaluation

A. System Implementation

An implementation of a distributed mobile network using the *MeDNN* framework is established with five Google Nexus 5 smartphones. The implementation of the *MeDNN* schemes is based on *MXNet*, which is a deep learning framework developed by the Distributed Machine Learning Community (DMLC) team using C++ [19]. We customize the *MXNet* framework through the JAVA Native Interface (JNI) to ensure the capability for the hardware configuration of the target device, Android 4.4.2 operating system, as well as the GEMM computing kernel. To achieve an efficient distributed network parallelism, we adopt *Open MPI* as the network framework, which relies on SSH for low-level support. Because there is no existing *Open MPI* and *SSH* libraries for Android system, we implemented an Android/ARM custumized version and integrated it into the *MeDNN* framework [20]. The wireless communication is based on Wi-Fi protocol in a interference-free environment. According to our experimental results, the average baseline for network throughput and delay are 43.8*Mbps* and 5*ms*, respectively. The system implementation is depicted in Fig. 7.

B. Experiment Setup

To deploy the test bench on mobile devices without loss of generality, we adopt *CaffeNet*, an optimized implementation of *AlexNet* with state-of-the-art performance for image classification tasks [16]. In the experiments, the performance of the proposed *MeDNN* is evaluated with five distinct convolutional layers at different scales [21]. When calculated locally on a single devices, the total execution time of the five layers is 1772*ms*, which is used as the baseline for later experiment. By leveraging *SMCD*, three variants of *CaffeNet* models (*Model 1*, *Model 2*, *Model 3*) are trained, achieving different sparsities and error rate as shown in Fig. 8. The increase of model sparsities and error rates is realized by setting incremental weight decays during the training step using *SMCD*. As the model becomes sparser in Fig. 8, the error rates increase from 42.57% to 43.5%. Even when the average layer sparsity reaches 67.1%, the top-1 error rate is still controlled within 0.87%. As can be referred in [14], we define the baseline error rate of the original *CaffeNet* model as 42.63%.

C. Evaluation of GTDP Scheme

We first evaluate the performance enhancement through *GTDP*. Fig. 9 compares the execution performance between the distributed system (from 2 to 4 *WN*s) and the single local device, in regard of 5 convolutional layers in *CaffeNet*. The bars in Fig. 9 shows that the computing time is inversely proportional to the number of *WN*s. Compared to 1772*ms* of the computing time on a single device, the *GTDP* partitioned model effectively reduced the time to 819*ms*, 580*ms*, and 453*ms* with 2 to 4 *WN*s respectively. Furthermore, the close-to-linear time reduction with the increasing number of *WN*s also indicates that the computing time is proportional to the area of the input feature map, implying the correctness of *GTDP*.

The dashed lines in Fig. 9 display the communication time of the distributed systems, which is introduced by the data communication overhead. The communication time increases with the size of the overlapping input feature map, which is introduced with the increment of *WN*s. Because of our dedicated partition scheme, the transmission amount is well controlled within 134*ms*, occupying ∼14% communication overhead for 2 *WN*s. And when the number of *WN*s increases to 4, the communication overhead is stilled maintained under 35% of the total execution time.

Overall, with *GTDP* scheme, the total execution time reaches 1.86 – 2.44× speedup on original *CaffeNet* with the number of *WN*s increasing from 2 to 4. It is also worth mentioning that the computing time of the three fully-connected layers of *CaffeNet* on the local single device is about 183*ms* according to our experiments. Compared to the execution time of the convolutional layers, it can be safely omitted.

D. Evaluation of SMCD Scheme

We evaluate the 3 *SMCD* models with their practical performance on mobile devices, from the perspectives of memory usage (W_{mem}), computing workload (W_{comp}), and transmission amount (W_{trans}).

Fig. 10 shows how the memory usage is reduced by *SMCD* scheme. Optimized for the memory intensive GEMM patch-to-column step, the 3 *SMCD* models can effectively reduce the memory usage as high

TABLE I: Overall evaluation of *MeDNN*.

	Comp	Comm	Total	Comp	Comm	Total
	Orig (ms)			Mod 1 (ms)		
1	1773	0	1773	1384	0	1384
2	819	134	953	687	97	784
3	580	198	778	508	147	655
4	453	275	728	358	204	562
	Mod 2 (ms)			Mod 3 (ms)		
1	1271	0	1271	1184	0	1184
2	620	91	711	570	87	657
3	460	134	594	413	127	540
4	327	187	514	295	176	471

as 65.9% in average. Specifically, the *SMCD* acting on the memory can effectively reduce the computing time by 27%, 35%, and 39% for the 3 models respectively. As patch-to-column steps usually consume 71.6% of the overall computing time in GEMM, this time reduction accounts for, on average, 24% overall computing time.

Fig. 11 depicts how the computing workload (FLOP) is reduced by *SMCD* during the matrix-to-matrix multiplication step. As the multiplication step is the major bottleneck for the CPU computing, the *SMCD* models significantly reduced the FLOP by 33% – 57%. *SMCD* reduces the computing time of matrix-to-matrix multiplication steps by 25.7% on average, occupying 12% of total computing time.

Moreover, the *SMCD* models also successfully reduce the network transmission amount contributed by the non-arbitrary shape-wise *group lasso*. Given a distributed mobile network with 4 *WN*s as shown in Fig. 12, the transmission time can be reduced by $23ms$, $40ms$, and $49ms$ respectively. Comparing to the $170ms$ of baseline, the transmission time is reduced by 22%.

One exception occurs in the results of the first convolutional layer in all the 3 *SMCD* models. We find the model sparsity remains unchanged for $conv1$ by utilizing our structured learning with group lasso, which infers no redundancy for the first convolutional layer.

E. Overall Evaluation of MeDNN System

Table I completely summarizes the overall execution time when deploying the original DNN model and 3 *SMCD* pruned models locally and on 2, 3, and 4 *WN*s. For each model, the computing time is shown in the left column while the communication time is shown in the middle. Thanks to the contribution of both *GTDP* and *SMCD* in *MeDNN*, compared to local execution of original model, the total execution achieves a $3.15\times$, $3.45\times$, and $3.76\times$ speed boost with 4 *WN*s for $model1$, $model2$, and $model3$, respectively.

VIII. Conclusion

In this work, we propose *MeDNN* – a local distributed mobile computing system to enhance the DNN performance on mobile devices. The proposed *MeDNN* framework utilize *GTDP* to promote the computation parallelism in distributed mobile network. *GTDP* effectively partitions the DNN model in regard of varying computing resource of mobile devices. Experiments show that, the partition scheme can accelerate the execution time by 1.86 – $2.44\times$ with 2 – 4 *WN*s. Also, the other *MeDNN* scheme – *SMCD* could effectively enhance the deployment efficiency of DNNs by 26.5% and 14.2% of the computing and communication time, respectively. Overall, the proposed *MeDNN* demonstrates close-to-linear performance ($3.76\times$ speedup for 4 *WN*s) enhancement for the DNN execution on mobile devices, expanding the application opportunities of machine intelligence in more practical mobile scenarios.

IX. Acknowledgment

This work was supported in part by NSF grants CNS-1717657 and SPX-1725456.

References

[1] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[2] S. Han *et al.*, "Ese: Efficient speech recognition engine with compressed lstm on fpga," *arXiv preprint arXiv:1612.00694*, 2016.

[3] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," 2014, pp. 8375–8379.

[4] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[6] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *CoRR*, vol. abs/1504.04788, 2015.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, 2008.

[8] S. Hassan, Mohammed Anowaruland Chen, *Mobile MapReduce: Minimizing Response Time of Computing Intensive Mobile Applications*. Springer Berlin Heidelberg, 2012, pp. 41–59.

[9] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1396–1401.

[10] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[12] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[13] J. Feng and T. Darrell, "Learning the structure of deep convolutional networks," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances In Neural Information Processing Systems*, 2016.

[15] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[18] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013.

[19] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[20] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2004, pp. 97–104.

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.