DC-CNN: Computational Flow Redefinition for Efficient CNN through Structural Decoupling

Fuxun Yu[1], Zhuwei Qin[1], Di Wang[2], Ping Xu[1], Chenchen Liu[3], Zhi Tian[1], Xiang Chen[1]
[1]*George Mason University, Fairfax, VA, USA*
[2]*Microsoft, Redmond, WA, USA*
[2]*University of Maryland, Baltimore County, Baltimore, MD, USA*
{*fyu2, zqin, pxu3, ztian1, xchen26*}*@gmu.edu, wangdi@microsoft.com, ccliu@umbc.edu*

***Abstract*—Recently Convolutional Neural Networks (CNNs) are widely applied into novel intelligent applications and systems. However, the CNN computation performance is significantly hindered by its computation flow, which computes the model structure sequentially by layers with massive convolution operations. Such a layer-wise sequential computation flow can cause certain performance issues, such as resource under-utilization, huge memory overhead, *etc*. To solve these problems, we propose a novel CNN structural decoupling method, which could decouple CNN models into "critical paths" and eliminate the inter-layer data dependency. Based on this method, we redefine the CNN computation flow into parallel and cascade computing paradigms, which can significantly enhance the CNN computation performance with both multi-core and single-core CPU processors. Experiments show that, our DC-CNN framework could reduce 24% to 33% latency on multi-core CPUs for CIFAR and ImageNet. On small-capacity mobile platforms, cascade computing could reduce the latency by average 24% on ImageNet and 42% on CIFAR10. Meanwhile, the memory reduction could also reach average 21% and 64%, respectively.**

***Index Terms*—Neural Network, Computation Optimization**

I. Introduction

With excellent learning capability and classification accuracy, CNNs have been widely adopted in various cognitive applications and systems. However, such satisfying CNN functionalities are usually based on massive neuron volumes and multi-layer interconnections. These complex structures and huge workloads bring significant concerns regarding the computation performance, such as computation latency, memory occupation, energy consumption, *etc*. Previously, many works have been proposed to optimize the CNN computation performance by reducing the computation load through model compression (*e.g.*, filter pruning [1]–[3], weight sparsity [4]), hardware specific optimization (*e.g.*, loop-untiling [5], fuse-layer [6]), *etc*. Although these optimization works have demonstrated their effectiveness, they are mostly limited by

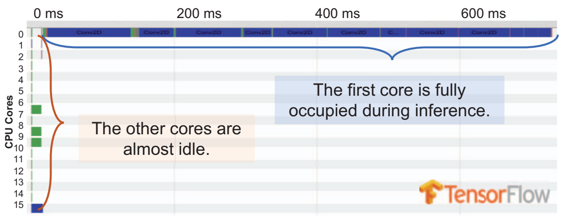
0 ms
200 ms
400 ms
600 ms
CPU Cores
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
The first core is fully occupied during inference.
The other cores are almost idle.
TensorFlow

Fig. 1: Core Under-Utilization during CNN Inference

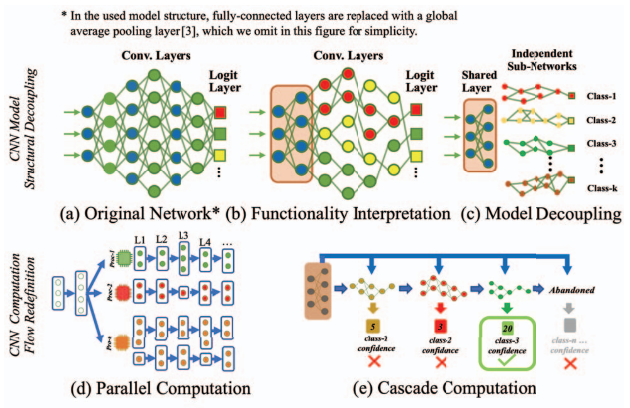

Fig. 2: DC-CNN Framework Overview

the sequential CNN computation flow in a sequential layer-by-layer manner [7], [8]. Such a layer-based computation flow is defined by the inevitable inter-layer data dependency. Specifically, the subsequent layers have to wait for the previous layer's whole output feature map for further computation. Therefore, certain performance issues may occur: (1) For high-performance multi-core platforms, the CNN inference may suffer from resource under-utilization if without good parallelism support. Fig. 1 shows a profiled CPU time-line when running VGG-16 inference on an Intel 16-core Xeon CPU. While the first core keeps running for 720*ms*, the other 15 cores are almost idle during 98% processing time. (2) For small embedded systems like mobile platforms, the limited capacity like memory resources can also become the bottleneck for running CNNs.

To tackle these challenges, in this work, we first propose to resolve the data dependency in Fig. 2 (a)-like CNN models. As shown in Fig. 2 (b), leveraging effective neuron functionally interpretation, we can group the class-specific filters and identify the critical model structure (*i.e. critical paths*) for individual classes. Then by structural connectivity pruning, we could decouple a CNN model into independent class-specific sub-networks as shown in Fig. 2 (c). Based on this CNN structural decoupling method, we then propose a novel CNN computation framework – DC-CNN. Two computing paradigms are proposed for different computing scenarios: For large-scale multi-core systems, as shown in Fig. 2 (d), we introduce the novel path-based parallelism into the computation flow. By deploying the independent sub-networks into parallel cores, the resource utilization is significantly enhanced, which brings much inference latency reduction. For small-capacity systems like mobile platforms, as shown in Fig. 2 (e), we propose a cascade computation flow, which processes independent sub-networks in a sequential manner. In this way, the sub-layer convolution could bring significant memory reduction compared to conventional full-layer-wise computation flow. With these two paradigms, the CNN computation flow is redefined by a general software-system co-design methodology for better computation performance.

Experiments show that, the parallel paradigm in our DC-CNN framework could provide at most 33% inference latency reduction on multi-core CPUs for both CIFAR and ImageNet datasets. For the cascade paradigm, it could provide average 24% (ImageNet) and 42% (CIFAR) latency reduction on mobile platforms. Meanwhile, it also achieves average 21% (ImageNet) and 64% (CIFAR) less memory occupation with negligible accuracy drop.

II. Preliminary and Motivation

A. Sequential CNN Computation Flow

Many previsou works have been targeting at optmizing the computation flow of convolutional neural networks [6], [9]. For example, CUDA support could map the computation workload in the same convolutional layer to different GPU cores [10]. H. Dogan and *et al.* proposed a shared-memory based multi-core architecture and synchronization mechanism for the parallelism [11]. M. Peeman and *et al.* proposed a memory-centric accelerator to fully-optimize the convolutional computation's data locality pattern to reduce the extensive memory overhead [12].

Though proved to be effective, many of these techniques are designed for particular computing platforms. As a result, many of them lack expected generality and adaptability. When applying such methods onto different embedded systems like mobile CPUs, the adaptability problems become even severer.

B. CNN Filter Function Interpretation

As aforementioned, current architectural optimization works mainly focus on hardware perspectives but lack the analysis from the more general CNN model perspective.

Recently, some interpretation works are proposed to analyze the CNN structure from a perspective of neuron functionality. For example: D. Bau and *et al.* [13] demonstrated that, the filters have distinct functionality divergence across layers and eventually only extract class-specific features. Moreover, A. Nguyen and *et al.* [14], [15] designed a novel visualization technique to illustrate the neuron's activation preference pattern. Utilizing this technique, Fig. 3 visualizes the prefered patterns of selected filters in the convolutional layers of a VGG-16 model [16]. It is obvious that, with the layer depth increment, the filters' maximized activation patterns demonstrate clearer class objects (*i.e.* "bird" in Fig. 3). As a result, the filters with the same "bird" preference compose of a determinant "path" structure for this class activation (the

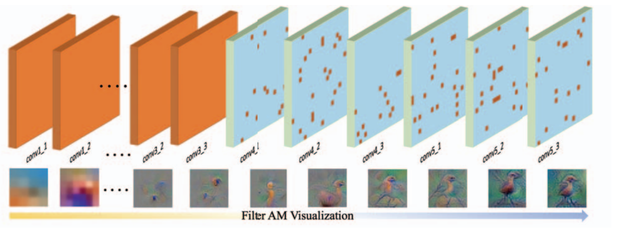

Fig. 3: Filter Functionality Divergence across Layers

orange dots) across the deeper convolutional layers. In the next section, we will identify the ubiquity of such structures (referred as *critical paths*) in CNN models.

III. Class-Specific Critical Path Distillation

We propose several ways to quantitatively interpret convolutional filters' class-specific functionalities and validate its correctness. Based on this, we identify the independent *critical paths* for different classes in CNN models.

A. Filter Functionality Interpretation

For the i^{th} filter in the l^{th} convolutional layer – F_i^l, we can interpret its functionality by analyzing its activation and gradients of the n^{th} class. First, given an input x, the confidence of the n^{th} class can be formulated as:

$$\begin{aligned} &Z^n(x) = F^L \circ \cdots \circ F^l \circ \cdots \circ F^1(x), \\ &where\ A^l(x) = F^l \circ \cdots \circ F^1(x) = \coprod A_i^l(x), \end{aligned} \tag{1}$$

where $Z^n(x)$ is the n^{th} class's confidence output in the final logit layer L, F^l is the l^{th} layer's full convolutional computation, A^l is l^{th} layer output feature maps consisting of each filter i's output feature map A_i^l, and $\coprod$ denotes the stack operation of feature maps. Based on this definition, the filter functionality can be interpreted by two approaches:

1) Activation Preference Interpretation: Given different sets of test images x^n from N classes, the filter's mean activation of each class can be formulated as:

$$Act_i^n = E(||A_i^l(x_p^n)||_1), \quad p \text{ is the pool size of } n^{th} \text{ class}, \tag{2}$$

where $||.||_1$ is the ℓ_1 norm, E is the mean function. If the images from the n-th class can cause a significant Act_i^n value than other classes, we can assume the filter F_i^l has a higher class preference to the n-th class.

2) Activation Significance Analysis: According to the first-order Taylor expansion [17], a filter's activation significance can be evaluated by the gradients of the output. In other words, the gradient $Grad_i^n$ can describe the differential impact of filter i's feature map to the n^{th} class's confidence:

$$Grad_i^n = ||\frac{\partial Z^n(A_i^l)}{\partial A_i^l}||_1, Z^n(A_i^l + \delta) \approx Z^n(A_i^l) + \frac{\partial Z^n(A_i^l)}{\partial A_i^l} \cdot \delta \tag{3}$$

where a larger $Grad_i^n$ indicates that a small change of δ on A_i^l will cause big influence to Z^n. Similarly, if filter i has a significant $Grad_i^n$, we can interpret that the filter i has more contribution to the n-th class.

3) Filter's Class-Specific Index: Combining both activation preference and activation significance based on Taylor Expan-

TABLE I: Filters class exclusiveness distribution across layers.

Conv. Layer	1	2	3	4	5	6	7
Filters Amount	64	64	128	128	256	256	256
STD of S_i^n	0.02	0.04	0.05	0.12	0.20	0.16	0.14

Conv. Layer	8	9	10	11	12	13
Filters Amount	512	512	512	512	512	512
STD of S_i^n	**0.35**	**0.33**	**0.40**	**0.57**	**0.82**	**0.79**

*Average normalized standard deviation (STD) of S_i^n for each filter, which shows the extent of filter's class preference bias.

sion, we derive the class-specific index – S_i^n to identify a convolutional filter's function exclusiveness to a specific class:

$$S_i^n = Act_i^n * Grad_i^n, \quad n^* = Argmax_n(S_i^n), \tag{4}$$

where n^* is the priori functional target class of the filter i.

B. Functionality Interpretation Cross-Verification

We conduct a series of experiments to validate the proposed class-specific index and explore the potential functionality divergence and independence between different classes. Fig. 4 shows a pair of interpretation examples for the classes of "cat" (a) and "dog" (b) in the Conv5_1 layer. For each figure, the left column shows the distribution maps of all the filters' Act_i^n and $Grad_i^n$ values corresponding to each class. It's clear that:

(1) The same distribution patterns of activation and gradient maps indicates the *consistency of the two interpretation approaches*. Combining two distribution maps according to Eq. 4, we select out the top-3 filters with maximum S_i^n, and their visualization patterns demonstrate clear corresponding class objects with distinct features. Therefore, three methods cross-verify the correctness of our functionality analysis.

(2) Meanwhile, the most activated *neurons for two classes are barely overlapped* when we compare two sets of distribution maps in Fig. 4 (a) and (b). Such exclusiveness between classes implies there exists great independence between different set of class-specific convolutional filters.

(3) We further calculate all the convolutional filters' class-specific indexes from a VGG-16 model trained on CIFAR-10. As shown in Table. 1, the average normalized standard deviation (STD) of filters' n-class S_i^n gradually increases with the layer depth. This trend qualitatively verifies the the functionality divergence in Fig. 3, and also demonstrates the ubiquity of the class-biased filters in deeper layers.

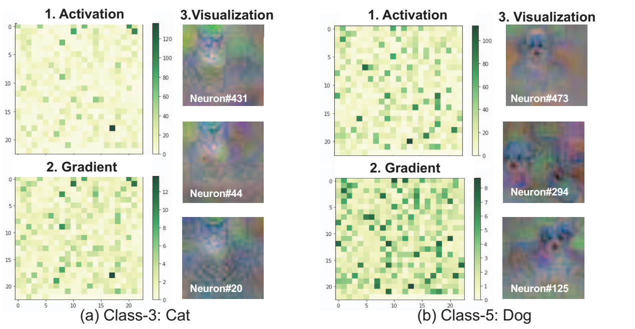

Fig. 4: Class-Specific Filter Functionality Interpretation

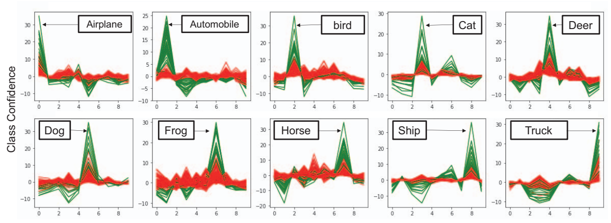

Fig. 5: Per-Class Critical Path's Classification Confidence

C. Independent Class-Specific Critical Path Distillation

Based on the filter's functional exclusiveness analysis, we then propose to group all the filters with the same class activation preference together across different convolutional layers. These filter groups thus form *critical paths*, which can be regarded as the meta-architecture for the corresponding class. Since the convolutional filters in different critical paths have significantly diverged activation preferences and very less shared features to share, the critical paths in deeper layers should have rare data dependency with each other. Therefore, we conduct the connectivity pruning between different paths and assure each path is independent from the other paths.

As shown in Fig. 5, in the VGG-16 model trained on CIFAR-10, we identified and tested the critical paths' classification performance. Each sub-figure shows one path's prediction confidence distribution, where green lines denote input from its functional target class, red lines otherwise. Clearly, even without other filter's support, each critical path alone can effectively detect corresponding class inputs with distinctively high confidence. In other words, the strong activation along the critical path can directly trigger the corresponding classification result without relying on other paths.

Based on the function interpretation, we can now successfully distill the independent class-specific critical paths. Based on this, we next propose a structural model decoupling method for flexible model optimization.

IV. Structural Model Decoupling

A. Overview of Structural Model Decoupling

Fig. 6 shows a conceptual model decoupling overview and the detailed implementation between two consecutive layers (the blue shadowed part). Our model structural decoupling transforms a CNN model into three parts: the shared layers, the decoupled layers, and the final logits, as shown in the upper part. (1) The *shared layers* are kept as the original model structure without decoupling, since these layers mainly extract multi-functional features that are generally utilized by all classes. (2) In the *decoupled layers*, we decouple the original convolutional layers into N independent critical paths. This is finally implemented by group convolution with N groups. (3) The *logit node* of each class is connected at the end of each corresponding path to generate the confidence for each class.

In the original structure in Fig. 6 (a), each filter needs to convolve with all feature maps produced by the previous layer. While in the decoupled structure in Fig. 6 (b), every class-specific filter only convolves with intra-path feature maps,

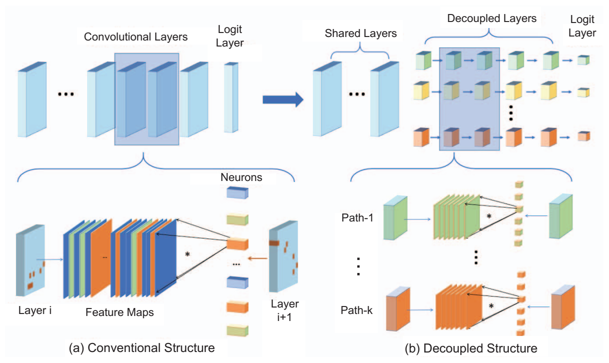

Fig. 6: Comparison of Original and Decoupled Structures

which is a very small portion. Therefore, the decoupled model eliminates many conventional model's inter-layer data dependency but conducts the same classification functionalities. Without these inter-path data dependency, the novel decoupled structure enables us to conduct flexible computation flow redefinition as we will show in the next section.

B. *Decoupling Configuration Optimization*

For a pre-trained CNN, it is critical to properly determine how many layers to be decoupled (decoupling depth), as well as how many filters should be chosen for each critical path (filter ratio). Higher depth and lower filter ratio mean more layers are decoupled with less filters in each path, reducing more computation workload but inducing larger accuracy drop. Therefore, optimizing the configuration to trade-off the performance and accuracy is critical to model decoupling.

Specifically, we use configuration search to determine the optimal parameters: (1) The decoupling depth D: we conduct a line-search (one to the maximum model depth); (2) The per-layer filter ratio FR: we search the global filter ratio for each layer to reduce the search space. And then a line-search from 20% to 1% is conducted. We then retrain the decoupled models and record their retrain accuracies. Fig. 7 shows the search space and results with VGG-16 on CIFAR10. As expected, the increasing decoupling depth and reducing filter ratio gradually downgrade the model accuracy. In most cases, accuracy drop can be effectively compensated by the retraining process until the optimal point: 7 layers decoupled with 1% filters reserved per path. Therefore, this configuration will be selected for the best computation efficiency and the smallest accuracy drop.

V. CNN Computation Flow Redefinition

In this section, we redefine the CNN computation flow and introduce two adapted paradigms for multi-core parallel processors and small-capacity embedded systems.

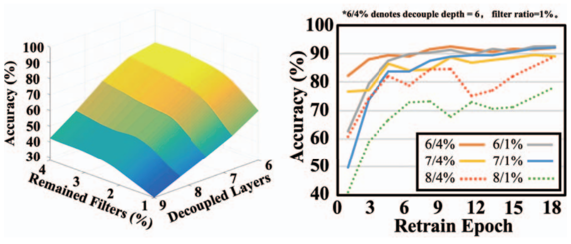

Fig. 7: Structural Decoupling Configuration Search Space

A. *Parallel Computing Paradigm for Multi-Core Systems*

The model decoupling method reforms a CNN model into a set of independent sub-networks, which introduce a new type of intrinsic model parallelism. Therefore, we design a novel parallel paradigm to enhance the resource utilization and improve computation performance for multi-core systems.

1) Parallel Computing Paradigm Overview: As illustrated in Fig. 2 (d), the major difference of our parallel computing paradigm from convention flow is within the decoupled model layers. Due to the decoupling-enabled parallelism, each critical path can be easily deployed to different cores of multi-core system. Meanwhile, it does not require dedicated hardware parallelism supports, such as core synchronization or inter-core communication mechanism since these paths are independent with each other. Compared with previous parallelism works [11], [12], our proposed paradigm serves as a software-level solution with no need of developing specialized architectural parallel assistance, which is an great advantage for generality on different systems.

2) Performance Enhancement Analysis:

a) Computation Reduction: Given a base CNN model, suppose we could decouple N sub-networks in the latter D layers, each with $1/N$ of original number of filters per layer. Then in each sub-network, one neuron only convolves with $1/N$ slices of the original full-size feature maps as illustrated in Fig. 6 (b). Thus, the computation workload reduction is:

$$C^{-}(N,D) = (1-\frac{1}{N})\sum_{d=2}^{D} C_d. \tag{5}$$

C_d is the original FLOPs of the d-th decoupled layer. And d starts from 2 since the first decoupled layer convolve with full-size feature map and thus has no workload reduction.

b) Latency Reduction: The latency reduction comes from two sources: computation workload reduction and the multi-core parallelism. (1) The computation reduction factor is N times for the decoupled layers as mentioned before. (2) The parallelism reduction factor depends on N sub-networks and P parallel cores: When number of cores is larger than number of paths ($P \geq N$), partial cores will be utilized with full-path parallelism factor N. Otherwise, all cores will be fully utilized to process the sub-networks with full-core parallelism factor P. Therefore, the expected latency reduction is:

$$T^{-}(N,P) = \alpha \times \sum_{d=2}^{D} T_d,\ where\ \alpha = \begin{cases} (1-\frac{1}{P\times N}),\ \text{if } N \geq P, \\ (1-\frac{1}{N\times N}),\ \text{if } N < P, \end{cases} \tag{6}$$

α is the latency reduction ratio, and T_d is the original computation time for each layer d.

B. *Cascade Computing Paradigm for Single-Core Systems*

The proposed DC-CNN framework can also facilitate the cascade computation on small-capacity platforms to reduce the inference latency and runtime memory occupation.

1) Cascade Computing Paradigm Overview: As illustrated in Fig. 2 (e), the major contribution of our cascade flow is we enabled the network to run *critical paths* in a sequential manner, instead of *layers*. By doing so, the small-capacity system only needs to store feature maps for a small subnetwork

TABLE II: Scalability Test of Proposed Model Structural Decoupling

Dataset	Classes	Baseline Acc.	Scratch Acc.	#FLOPs	Decouple Config	Retrain Acc.	Final #FLOPs	FLOPs Reduc.
CIFAR10	10	92.1%	92.1%	3.13E+08	D = 7, FR = 1%	92.1%	1.56E+08	50.2%
ImageNet10	10	70.2%	93.2%	1.54E+10	D = 6, FR = 10%	92.8%	1.08E+10	29.8%
CIFAR100	100	73.1%	73.1%	3.13E+08	D = 6, FR = 1%	72.3%	2.08E+08	33.5%
	40	72.1%	79.4%	3.13E+08	D = 7, FR = 4%	78.1%	1.98E+08	36.7%
	30	72.8%	82.3%	3.13E+08	D = 7, FR = 4%	81.8%	1.96E+08	37.4%
	20	71.2%	85.1%	3.13E+08	D = 7, FR = 4%	84.7%	1.86E+08	40.6%

at any time, which greatly reduces the runtime memory occupation and the potential memory access overhead [12].

2) Performance Enhancement Analysis: Here we analyze the theoretical performance in latency and memory reduction.

a) Latency Reduction Analysis: Without parallelism, the latency reduction in cascade flow only comes from the computation workload reduction in Eq. 5. Therefore, the latency reduction can be directly formulated:

$$T^{-} = (1 - \frac{1}{N}) \sum_{d=2}^{D} T_d. \quad (7)$$

And as we will show later, the theoretical latency reduction matches our real performance quite well in real mobile testing.

b) Runtime Memory Reduction Analysis: In the shared layers of our decoupled models, the cascade computing flow occupies the same runtime memory as the conventional structure. While in the decoupled layers, the output feature map size is deducted to $1/N$ of original size (if one path is $1/N$ of original layer), since we only run one path per time.

VI. Experimental Evaluation

A. Scalability Analysis for Structural Model Decoupling

We evaluate the scalability and generality of our model decoupling method in two aspects: higher image data complexity and more class composition complexity. Complementarily, we also demonstrate that our method has high flexibility to support highly customized class composition.

1) Scalability with Image Data Complexity: We test our method with ImageNet datasets (10 classes are randomly chosen) to show the scalability for input data complexity. The baseline model is a full-size pre-trained model trained on the whole ImageNet datasets, and baseline accuracy is its test accuracy on the subset of classes. We then train a model from scratch on the chosen ImageNet subset (with higher scratch accuracy 93.2%). The structural decoupling results are shown in Table II. By decoupling 6 layers with 10% filter ratio, our decoupled model maintains 92.8% accuracy (with 0.4% accuracy drop compared to high scratch accuracy) but reduces model computation workload by 35.2%, which proves our method's generality on large-scale inputs.

2) Scalability with Class Composition Complexity: To verify the scalability for more complex class composition, we generalize previous 10 classes to 100 classes scenario. Experiments on CIFAR100 proves our method's high scalability with more classes: under the 100-class setting, the optimal decoupled configuration is D = 6, FR = 1%, providing 33.4% FLOPs reduction with only 0.8% accuracy drop.

TABLE III: Parallel Performance for CIFAR

	Inference Latency		Reduction	
Mode	Orignal	Ours.	Theory	Practical
CPU-1	4175ms	3163ms	29.8%	24.2%
CPU-2	4321ms	2864ms	32.5%	33.7%
CPU-4	4298ms	2857ms	32.9%	33.5%
CPU-8	4290ms	2845ms	33.3%	33.6%
CPU-16	4275ms	2831ms	33.6%	33.7%

TABLE IV: Parallel Performance for ImageNet

	Inference Latency		Reduction	
Mode	Orignal	Ours.	Theory	Practical
CPU-1	14.95 s	11.35 s	29.8%	24.1%
CPU-2	15.12 s	11.01 s	31.5%	27.2%
CPU-4	15.08 s	10.38 s	32.3%	31.2%
CPU-8	15.06 s	10.33 s	32.7%	31.4%
CPU-16	15.10 s	10.21 s	32.8%	32.4%

3) Add-on: Customizable Class Composition: The proposed structural decoupling also supports users to further reduce model's computation workload according to their customization needs. To do so, it requires the fully interpretation of the model parameters, which can hardly be done by previous pruning methods. To demonstrate our high flexibly, we evaluate three such customized settings: reserving a random subsets of classes (20, 30, 40) in CIFAR100. To optimize for such settings, we flexibly remove all unnecessary critical paths of non-required classes. Experiments show that, our decoupled models with flexible class composition can provide 36%$\sim$41% FLOPs reduction on all settings with negligible accuracy loss.

B. Performance of Parallel Paradigm

Then we evaluate the proposed parallel computing paradigm performance on the server-level Intel Xeon 16-core CPU. The experimental results on CIFAR and ImageNet are shown in Table III and Table IV. Both models are decoupled with decoupling depth D = 6, filter ratio FR = 10% with no accuracy loss. The model inference latency is tested on one batch of images (CIFAR: 128, ImageNet: 32). CPU-i means the model inference process can utilize up to i physical cores of the CPU, which is done by setting the number of parallel threads. The theoretic latency reduction is calculated according to Eq. 6, and the practical reduction is averaged on 100 running.

As Table III shows, our parallel computing paradigm on CIFAR brings a 24$\sim$34% latency reduction compared to original sequential computation flow. The ImageNet results in Table IV show similar latency reduction rates. With increasing parallelism from CPU-2 to CPU-16, the latency reduction ratio increases to the maximum $\sim$33.7%, which exactly matches the theoretic analysis result (33.6%) calculated by Eq. 6.

TABLE V: Performance of Cascade Computation for CIFAR10 and ImageNet10

Dataset &		Original Model		Decoupled Model		Theoretical		Practical	
	Mobile	Latency	Memory	Latency	Memory	#FLOPs	Latency	Latency	Memory
Model Settings	Platforms	(ms)	(MB)	(ms)	(MB)	Reduction	Reduction	Reduction	Reduction
CIFAR10 (D=7, FR=1%)	Nexus-5X*	95	155.1	56	55.2	50.68%	53.24%	41.05%	64.41%
	Pixel-XL*	98	161.3	57	58.8	50.15%	50.15%	41.84%	63.55%
	Honor-8	137	164.3	75	52.5	50.15%	50.15%	45.26%	68.05%
	Nexus-4	330	158.5	185	58.4	50.15%	50.15%	43.94%	63.15%
CIFAR10 (D=6, FR=1%)	Nexus-5X*	95	155.1	65	70.2	33.54%	33.54%	31.57%	54.74%
	Honor-8	137	164.3	85	68.5	33.54%	33.54%	37.96%	58.31%
ImageNet (D=6, FR=10%)	Nexus-5X*	1455	281.1	1122	223.2	29.81%	31.30%	22.86%	20.60%
	Honor-8	1532	290.3	1145	226.5	29.81%	31.30%	25.26%	21.98%

Note: [*] denotes evaluation is done on Android Studio Emulator, otherwise evaluation is done on real mobile phones.

Finally, to better illustrate our parallel paradigm, Fig. 8 shows the decoupled layers' parallel computation details, which demonstrate the great enhancement of core utilization.

C. Performance of Cascade Paradigm

We then evaluate our cascade computation paradigm on four different mobile platforms and evaluate their real latency and memory reduction performance. Specifically, the Tensor Lite models are loaded into an Android application, and the inference latency is on one single image. The memory evaluated here is the average peak runtime memory of the application monitored by Android Studio tracing.

Table V shows the overall performance comparison of cascade computation of the decoupled model and the original model. On all four test platforms, our decoupled cascade model consistently provides lower latency and memory occupation than original model. For quantitative comparison, we calculate the theoretic latency reduction ratio based on Eq. 7. Take the settings of CIFAR10 (D=7, FR=1%) as an example: The theoretical latency reduction is 50.68%, and the practical latency results demonstrate a range of 41.1%∼45.3% latency reduction on different platforms. Considering different platforms' runtime dynamics, the latency reduction is roughly consistent with our theoretical analysis.

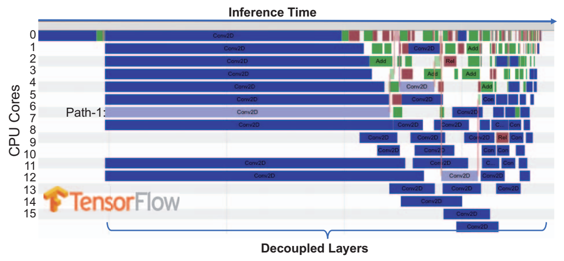

Fig. 8: Parallel Flow: Multi-Core Utilization Enhancement

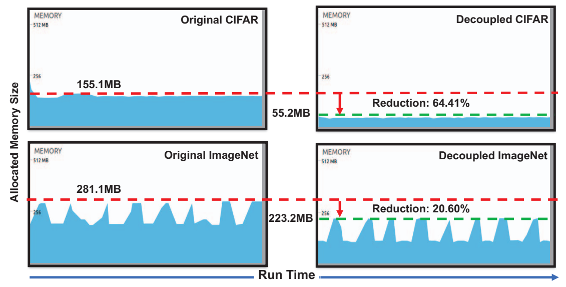

Fig. 9: Cascade Flow: Memory Reductions on Nexus-5X.

The memory reduction gain is also significant and consistent for different settings according to the results. Fig. 9 shows the runtime memory traces by Android Studio to better illustrate the memory reduction. Each spike on the memory traces denotes the peak allocated memory for one inference. Clearly, through our structural model decoupling and cascade computing paradigm, we can bring the CIFAR10 model average 64.4% memory reduction (and 20.6% for ImageNet model) on Nexus-5X, which is a significant reduction for mobile devices.

VII. Conclusion

In this work, we proposed a novel model structural decoupling method. Based on the method, two computation paradigms: parallel and cascade computation are proposed in our DC-CNN framework to adapt to large-capacity and resource-constrained scenarios. Extensive experiments demonstrated that DC-CNN framework's generality and scalability.

References

[1] J.-H. Luo and *et al.*, "Thinet: A filter level pruning method for deep neural network compression," *arXiv:1707.06342*, 2017.
[2] H. Li and *et al.*, "Pruning filters for efficient convnets," *arXiv:1608.08710*, 2016.
[3] Y. He and *et al.*, "Channel pruning for accelerating very deep neural networks," in *Proc. of ICCV*, 2017.
[4] S. Han and *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv:1510.00149*, 2015.
[5] C. Zhang and *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proc. of FPGA*, 2015.
[6] M. Alwani and *et al.*, "Fused-layer cnn accelerators," in *MICRO*, 2016.
[7] G. Li and *et al.*, "Block convolution: towards memory-efficient inference of large-scale cnns on fpga," 2018.
[8] Y. Ma and *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *FPGA*, 2017.
[9] M. Lin and *et al.*, "Network in network," *arXiv:1312.4400*, 2013.
[10] CUDA, "Programming guide," 2010.
[11] H. Dogan and *et al.*, "Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging," in *IPDPS*, 2017.
[12] M. Peemen and *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *Proc. of ICCD*, 2013.
[13] D. Bau and *et al.*, "Network dissection: Quantifying interpretability of deep visual representations," 2017.
[14] A. Nguyen and *et al.*, "Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks," *arXiv:1602.03616*, 2016.
[15] J. Yosinski and *et al.*, "Understanding neural networks through deep visualization," *arXiv:1506.06579*, 2015.
[16] F. Yu, Z. Qin, and X. Chen, "Distilling critical paths in convolutional neural networks," *arXiv:1811.02643*, 2018.
[17] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.