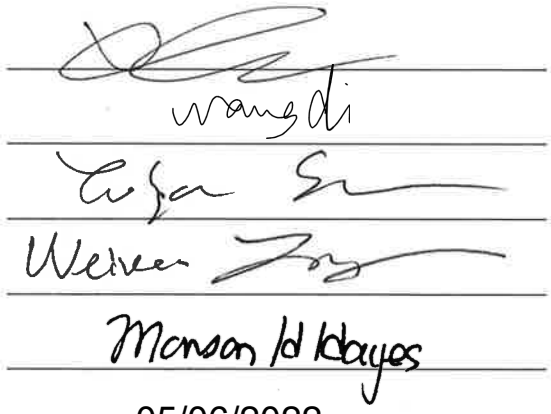
TOWARDS LARGE-SCALE AND HIGH-PERFORMANCE DEEP LEARNING COMPUTING

by

Fuxun Yu

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Electrical and Computer Engineering

Committee:



Dr. Xiang Chen, Dissertation Director

Dr. Di Wang, Committee Member

Dr. Tolga Soyata, Committee Member

Dr. Weiwen Jiang, Committee Member

Dr. Monson H. Hayes, Department Chair

Date: 05/06/2022

Spring Semester 2022
George Mason University
Fairfax, VA

Towards Large-Scale and High-Performance Deep Learning Computing

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Fuxun Yu
Master of Science
George Mason University, 2022
Bachelor of Science
Harbin Institute of Technology, 2017

Director: Dr. Xiang Chen, Assistant Professor
Department of Electrical and Computing Engineering

Spring Semester 2022
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation:

To my family

For their self-sacrificed giving and continuous support along my growth;

To my advisor

For the fruitful doctoral training and important career support;

To my beloved fiancee

For lighting up my life.

Acknowledgments

I would like to thank the following people along with my Ph.D. journey. I'd like to thank my advisor, Dr. Xiang Chen, who not only extended me the oppourtunity to pursue the Ph.D. degree, but also gives the best training. Without him, I couldn't have grown as a logical, thoughtful, and prepared Ph.D. candidate. I'd like to thank my mentor in Microsoft, Dr. Di Wang, who not only mentors me two internships, shares me with the personal growth experience, but also gives the ult-most support in finding my career path. I'd like to thank all my collaborators for working with me on many interesting projects and helping me improve with many aspects like idea developing, academic writing, story telling etc. I'd like to thank my labmates, Zhuwei and Zirui, who not only supported me in my hardest time when I first arrived U.S., but also collaborated with me along with the full Ph.D. journey with fruitful publications. Thanks my families and the one who is about to become, you give me all the love that I ever need in one's normal life.

Table of Contents

List of Tables

List of Figures

Abstract

TOWARDS LARGE-SCALE AND HIGH-PERFORMANCE DEEP LEARNING COMPUTING

Fuxun Yu, PhD

George Mason University, 2022

Dissertation Director: Dr. Xiang Chen

Deep Learning (DL) has achieved superior performance than human in complex cognitive tasks like vision, speech, language, medical, game, etc. With the ever increasing market demands, DL applications and the underlying computing hardware have demonstrate strong scaling trends in terms of *Model Scaling* and *Computing Scaling* (*e.g.*, increased computing parallelism, memory and storage to serve larger models).

On the one hand, deep neural network (DNN) models are becoming increasingly bigger with higher structure complexity, larger parameter sizes, and increased computational workloads so as to strive for better accuracy. For example, one of the biggest model VGG16 in 2017 has only 138M parameters, while in 2021, the largest model MT-NLG has reached 530B parameters, which shows a 4,000× increase. On the other hand, high-performance computing hardware demonstrates strong scaling trends and provides the core support for both DL model training and inference. For example, NVIDIA GPUs have maintained 2× increase in the number of transistors for each generation, which demonstrates exponential performance gain like throughput and memory bandwidth scaling in the past decade.

Such a double scaling trend greatly complicates the high-performance deep learning computing system, including model design, kernel compiling and runtime scheduling, etc.

In this dissertation, we introduce several of our works in high-performance deep learning computing from a full-stack optimization point of view. This includes not only algorithm-level compression and acceleration: *Antidote* and *DCCNN*, but also the hardware-software co-design perspectives like GPU-aware DNN design, *TA-DNN*, and the multi-tenant DL runtime scheduling, *MT-Graph*. Finally, we summarize our prior works and share our vision and insights towards the future large-scale deep learning systems. Through introducing these works and sharing our understanding, we hope that this dissertation could shed some light on the future high-performance deep learning computing research.

Chapter 1: Introduction

The past recent years have witnessed a great surge of AI technology development covering a wide-spectrum of domains and applications. At the core of such AI development are tons of deep learning (DL) algorithms emerging in computer vision, speech, language, remote sensing, and point cloud, etc., which revolutes the horizon of machine intelligence by enabling smart vehicles, smart city, smart medical, and smart agriculture etc. Behind the superior performance of these applications, high-performance computing on powerful hardware provides the most fundamental support, and optimizing the runtime performance for compute-intensive DL algorithms is the key to provide the assurance to model accuracy, satisfying execution speed, real-time performance, user experience, etc.

In fact, before we dive into technical backgrounds, there is an interesting history that reveals the importance of computing to the AI development.

AI Emergence and Winter As shown in Figure 1.1, the first concept related to deep neural networks (DNNs) was proposed in 1940s that referenced a net of neurons with soma and axon to conduct the logical calculus [4]. The network is designed with thresholded logic unit with hard-coded weights, which intended to mimic "integrate and fire" model of neurons. The concept of "perceptron" is then proposed in 1957 by F. Rosenblatt that could learn adjustable weights and could recognize letters and numbers [5]. Rosenblatt implemented the perceptron in custom hardware, and showed it could be used to learn to classify simple shapes correctly with 20×20 pixel-like inputs, essentially a computer that could approximate a classification function. The success of the idea of "machine intelligence" rapidly opened up the next ten years of golden age for AI.

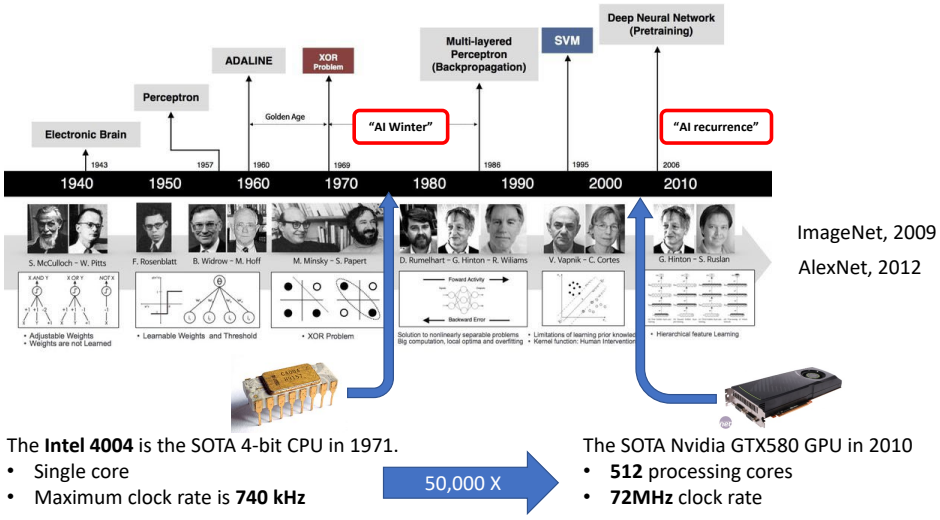


Figure 1.1: The recent recurrence of AI is largely related to the powerful hardware.

"The perceptron is the embryo of an electronic computer that the Navy expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

– Frank Rosenblatt, 1958.

However, researches for more complex cognitive AI applications soon ran into another 10-year of "AI winter" since further increasing the capacity of perceptron encountered many bottlenecks including the inefficient weight learning algorithm, the lack of large amounts of data, and the performance-limited computing hardware. As a fact, the most powerful hardware in early 1970s is the Intel 4004 CPU that has a single core and can only conduct 4-bit arithmetic instructions with a maximum clock rate of 740 kHz. And the training algorithm by differentiation methods (the essential method for neural network training) is not implemented and being able to compute on CPUs until 1976 as discussed in [6].

Although the backpropagation algorithm is re-discovered and improved in 1980s, the enthusiasm for deep neural networks cooled down in the research community and more efforts were devoted into traditional machine learning theories and algorithms like SVM.

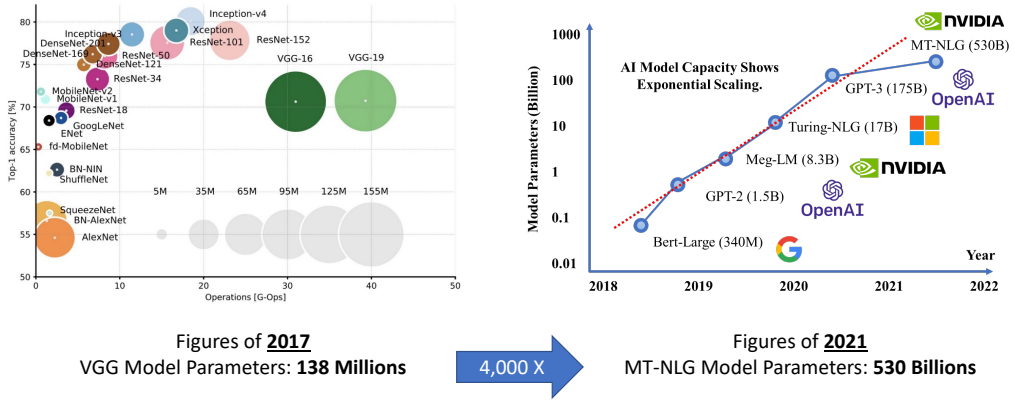
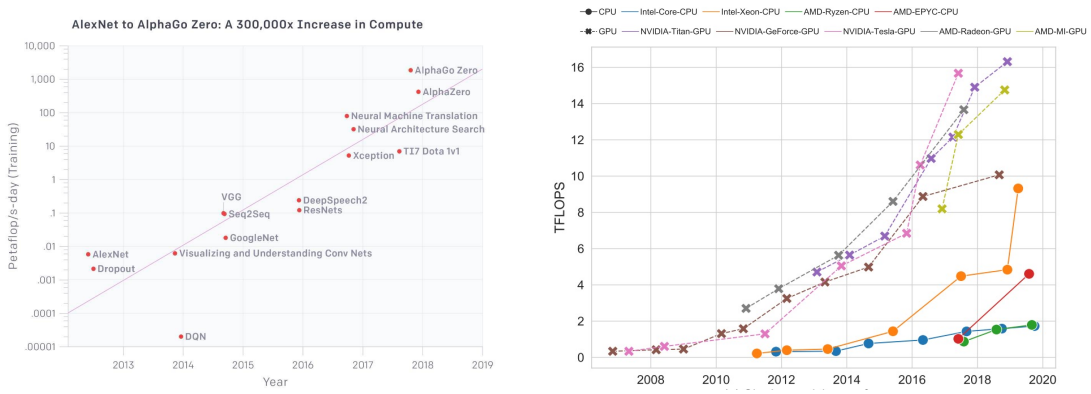


Figure 1.2: Deep learning with model scaling [1].

AI Recurrence and The New Double Scaling Era The real AI recurrence happened in 2010s as one of the major problems of computing bottleneck is addressed by the recent great advance of parallel computing capability of GPUs. For comparison, the most powerful GPU in 2010 is NVIDIA GTX580 GPU which has 512 processing cores with each single core frequency at 72MHz, which showed more than 50,000× computing capability compared to the 1970s. The adoption of GPU computing for deep neural network model training is firstly applied in AlexNet, which won the first prize in ImageNet 2012 competition, and soon the wave of deep learning strikes back and maintains the prosperous development. There are tons of deep learning (DL) algorithms emerging in computer vision, speech, language, remote sensing, and point cloud, etc., which revolutes the horizon of machine intelligence by enabling smart vehicles, smart city, smart medical, and smart agriculture, etc.

Despite the widespectrum of application domains, the key to the success of deep neural networks is simple and effective, i.e., by designing increasingly deeper and larger models. Figure 1.2 shows the comparison of state-of-the-art model size statistics before 2017 and the recent redrawing of it in 2021. The model sizes increased by 4,000× from 138M parameters of VGG16 [7] to the recent 530B parameters of MT-NLG [8]. With the rapid growth of model size, the computing workload for model training and inference also demonstrates



(a) DL Model Compute Scaling

(b) CPU and GPU Compute Scaling

Figure 1.3: Deep learning with computing scaling [2].

exponential scaling. As shown in Figure 1.3, the computing workload comparison between model at 2013 and 2019 also demonstrates a 300,000× increase.

Such huge growth in DL model computing also motivates the performance scaling of high-performance hardwares, especially the graphic processing units (GPUs) with massive computing parallelism. With the strong market needs, the capacity of recent generations of GPUs shows exponential growing, which demonstrates the overwhelming computing power compared to the common model workload. The detailed computing scaling statistics could be found in recent works [9], but one concrete example is: From K80, P40, and P100 to recent T40, V100, A100, GPUs maintain the trend of doubling in performance since 2010s. The last generation of V100 [10] offers 120 Tera floating point operations per second (TFLOPS) and 900 GB/s memory bandwidth, and the numbers further increase to 312 TFLOPS and 1.6TB/s memory bandwidth for the newly released A100 [11], which reports the ResNet50 [12] inference speed of 36,436 images/second and meanwhile provides continuously increasing energy efficiency.

This Dissertation Such a double scaling trend of DL model and compute capacity brings great opportunities for novel types of DL models and applications development, and meanwhile incurs lots of challenges to performance optimization for large-scale high-performance deep learning computing. In this dissertation, we start with deep learning algorithm level computing optimization, i.e., convolutional neural network (CNN) model compression and acceleration, and then delve into the software and hardware co-optimization to achieve ult-most computing efficiency and performance.

Specifically, we start by introducing two algorithm-level compression works that targets at the fundamental model type, convolutional neural networks. Different from many research works that focus on weight or filter pruning, our work targets at the other two novel components of CNNs: feature maps and connectivity. We thus propose an attention-based dynamic feature map pruning framework, Antidote; and the structural decoupling framework by connectivity pruning, DC-CNN.

Then to the hardware level, we propose two software-hardware co-optimization works. Specifically, the first work targets at the graph and runtime co-optimization that conducts resource-aware multi-tenant runtime scheduling, MT-Graph; and the second work targets at GPU-aware DNN design by tail effect analysis and elimination, TA-DNN.

Finally, we summarize our dissertation and analyze the future trend by sharing our understanding and vision. We hope by sharing these works and understandings, this dissertation could shed some light on the future large-scale deep learning computing research.

Chapter 2: Backgrounds and Related Works

2.1 Convolutional Neural Network Basis

Overview As one of the major types of deep learning models, convolutional neural network (CNN) is the fundamental model structure that serves as the basic backbone in various of applications like computer vision. There are several types of layers: convolution layers, pooling layers, and the fully connected layers. Among these types of layers, convolutional layers are the most computing-intensive layer types and thus computing optimization works usually focus on convolutional layers.

Figure 2.1 shows a overview of the CNN structure, there are three basic computing components in a convolutional layer: convolutional filter, feature map, and the connectivity.

2.1.1 Convolutional Filter

The convolutional computation is conducted by convolving the filter with feature maps in a sliding-window way. As shown in Figure 2.2, we could convolve a filter with 3×3 kernel

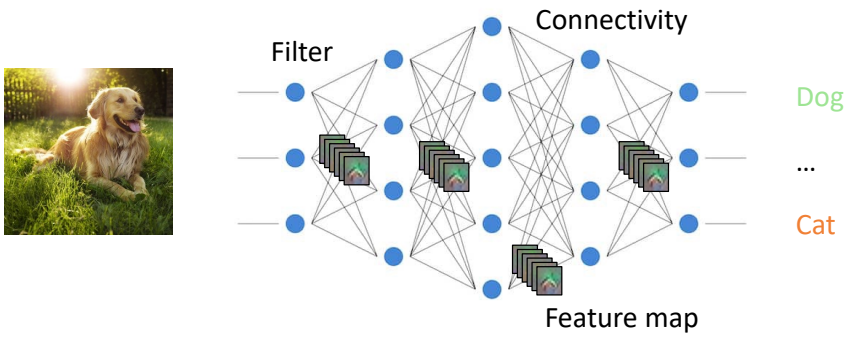


Figure 2.1: CNN Structure Overview.

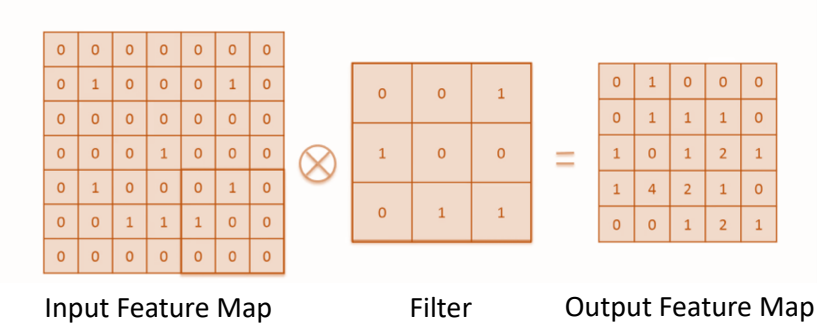


Figure 2.2: Convolution operation.

size on a input feature map with 7×7, which could produce a output feature map with 5×5 output feature map. There are also several other factors like padding, and striding that affects the output feature map size, as well as the computation workload. Also note that, one convolutional layer could have many filters to increase the capacity of the model and meanwhile to capture the different and various patterns of the input, so as to improve the model accuracy.

2.1.2 Feature Map

The feature map is the intermediate data generated by the convolutional operator. Specifically, the output feature map will serve as the input of the next layer. Since there can be multiple filters in one convolutional layer, the output feature maps can become multiple channels stacking together, as shown in Figure 2.1.

2.1.3 Connectivity

The connectivity denotes the convolutional connection between convolutional filters and feature maps. In general cases, the filters and feature maps are fully connected as shown in Figure 2.1. However, our investigation shows that such fully-connected convolutional layers have many redundancy in their connection, partial of which could be removed without hurting the model accuracy.

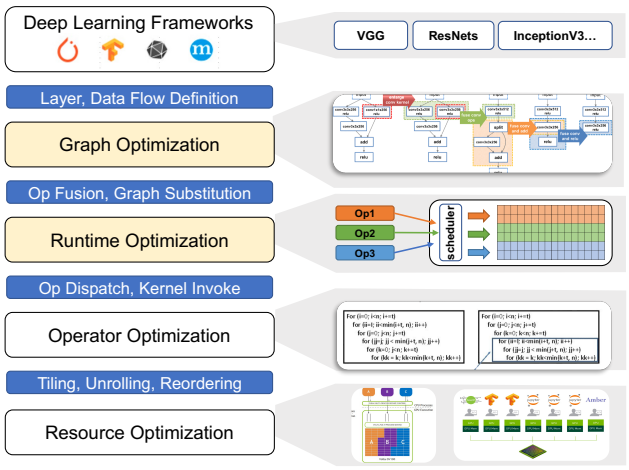


Figure 2.3: DNN optimization stack overview.

2.2 DNN Performance Optimization Stack

Overview We first expand the backgrounds of DNN execution stack as shown in Fig. 2.3, that is composed of multiple architecture levels [13].

2.2.1 Algorithm-level

The top *algorithm level* includes different deep learning frameworks and algorithms, such as the most commonly-used DL libraries, TensorFlow and PyTorch. This level also defines various DNN model structures like VGG, ResNets, InceptionNets, etc. Many computing optimization works in this level focuses on DNN compression [14–17], and new architecture design like neural architecture search [18–21].

2.2.2 Graph-level

The *graph level* untangles model structures to abstract individual operators and identify the data processing flow as directed acyclic graphs (DAG) [22]. Graph-based optimization is thus introduced into this level to achieve operator fusion and sub-graph substitution, and therefore reduce memory access/operator invoking overheads, etc. There are many

graph-level optimization works including [13, 22, 23], etc.

2.2.3 Runtime-level

Down to the *runtime level*, it controls when and how operators are dispatched onto physical computing units and is critical in our balanced resource utilization. This is usually done by the native black-box GPU scheduler, but we could leverage certain APIs to adjust the dispatching results. Here runtime optimization [24–26] aims to wisely dispatch operators of varied resource consumption to appropriate processing units, thus optimizing different objectives like maximizing the utilization, reducing contention, etc.

2.2.4 Kernel-level

The *kernel level* is the lower level that conduct per-operator execution, such as tiling, unrolling, reordering, etc., to improve the computing efficiency. Such intra-operator optimization has a distinct scope and is orthogonal to the concurrent operator scheduling such as runtime scheduling. There are many kernel-level optimization works like TVM [13], Ansor [27], Cortex [28], etc.

2.2.5 Resource-level

The *resource level* is the bottom level that conduct resource partition, allocation or provisioning. Such resource-level management may be trivial in single-tenant settings when resource is exclusively assigned, but it is drawing increasing attention recently with the trend of multi-tenant computing scenarios where multiple models co-run on the same hardware. Especially, resource partition is highly needed to improve the utilization and enhancing the computing throughput in the recent trend of large-scale computing hardware like TPUs/GPUs. There are many emerging works in this direction that conducts dedicated resource-level optimization like [29–32].

Chapter 3: Algorithm-level Compression and Acceleration

In this chapter, we first introduce our works in algorithm-level compression and acceleration: *Antidote* and *DCCNN*. Specifically, *Antidote* mainly aims to conduct dynamic feature map pruning to reduce the DL workloads. *DCCNN* not only conducts connectivity pruning to both reduce workload, it meanwhile designs and enables novel and efficient computing paradigms on multi-core CPU and mobile platforms.

3.1 Antidote: Attention-based Feature Map Pruning

3.1.1 Problem Motivation

In the past few years, Deep Neural Networks (DNNs) have achieved extraordinarily accuracy boost on various cognitive tasks, such as image classification [33, 34], object detection [35, 36], speech recognition [37–39], etc. However, the increasingly larger model structure also introduces tremendous computation load, causing considerable performance issues [40–43]. For example, one of the largest model for language tasks— *GPT-3* [39] consists of 175 billion parameters, whose huge memory occupancy makes it even impossible to be trained on the server with eight V100 GPUs with 16GB memory [39].

To reduce the DNN computation resource requirement, many optimization works have been proposed to reduce the model redundancy by identifying and removing insignificant components. Han et al. considered the weight magnitude as the significance criteria and introduced weight sparsity regularization for model compression [44]. As such weight sparsity is irregular during hardware computation, structural pruning was also proposed in [44] to remove regular sparsity achieve practical computation acceleration [16, 45]. Further, Li et al. defined the ℓ_1-norm of the convolutional filter weights for significance evaluation and removed small filters for less computation load [15, 46].

Although these works demonstrate good performance, all these works only evaluate model components' *static* significance with parameter (a.k.a. weights) information, ignoring their *dynamic* interaction with external inputs. As DNN models are trained for discriminating different input features, different single input may have certain feature activation variance. This will lead to circumstances that certain generic significant neurons (such as filters with large ℓ_1 norm) may not be activated by particular inputs [47], while some insignificant neurons with smaller norms are specifically activated [48]. Such a variance suggests that, instead of *static* importance evaluation, the model components' significance should be evaluated with a *dynamic* manner in the inference process with different inputs.

Recently, certain research works have leveraged such a *dynamic* manner for DNN model optimization with different goals. For example, the ImageNet-2017 winner *SENET* found that different inputs have significant impact on the feature channel significance. And dynamic feature channel scaling can achieve outstanding accuracy improvement [49]. Yu et al. also reveal that regardless of filter weight, certain filters contain particular class-specific features with significant activation impact. Therefore, filters with significant weight can be also aggressively pruned according to different class inputs [50]. Fig. 3.1 shows two examples that demonstrate the spatial attention, which could effectively capture the major feature of the objects which is very sparse but aligns well with human vision system.

Compared to the conventional *static* component significance based works, the *dynamic* optimization provides several advantages: (1) Qualitatively, it can identify every component's significance more accurately in the practical testing phase by considering the dynamic input-feature interaction. (2) Quantitatively, dynamic optimization enables more aggressive redundancy elimination with per-input component significance evaluation, while the static methods can only identify a general redundancy based on the whole training dataset. (3) Meanwhile, benefited from the dynamic mechanism, the model optimization degree (such as feature pruning ratio) could be adaptively adjusted to meet different resource budgets, while an static optimized model cannot be changed unless with heavy retraining and re-optimization cost.

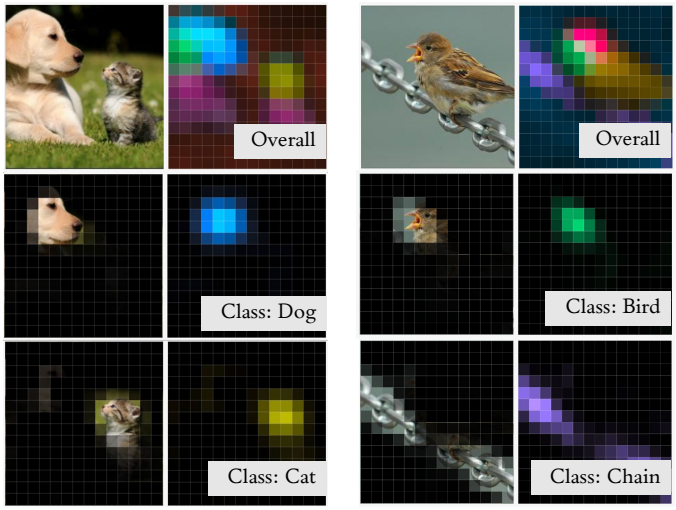


Figure 3.1: Attention iillustration [3].

To achieve such benefits, there are also several challenges: (1) How to design an effective and efficient evaluation metric for dynamic feature significance evaluation? (2) How to apply the dynamic optimization in different model dimensions, especially the non-structured ones? (3) How to develop an effective co-optimization scheme on both testing and training phases to achieve comprehensive performance improvement? (4) How to flexibly and adaptively adjust the optimization level to meet the inference requirement (such as resource constraints in practical deployment) with minimum re-training overheads?

To solve the above challenges, we propose ***Antidote***: an attention-based dynamic DNN optimization framework. Fig. 3.2 gives a systematical overview of our proposed method. ***Antidote*** targets at dynamic feature map pruning by identifying the feature map redundancy associated with every different input. Specifically, between any two consecutive convolutional layers, we utilize the attention-based mechanism to evaluate the importance of different components of feature map and then generate the feature pruning mask. The mask will be applied back onto the feature map and to remove the channel-wise and spatial-wise redundancy structurally. By doing so, we could remove the non-essential computation in the next layer for runtime efficiency optimization.

3.1.2 Attention Formulation

We first give the formal definition of channel and spatial attention. Given an intermediate feature map $F \in R^{C*H*W}$ (here, C, H, W is the channel depth, height, width of the feature map), the channel attention coefficient can be obtained by calculating the average of feature map entries in the spatial dimension $H * W$:

$$A_{channel}(F, c) = \frac{1}{H * W} \sum_{i}^{H} \sum_{j}^{W} F_c(i, j), \tag{3.1}$$

where $A_{channel}(F, c)$ is channel attention for the selected channel c. For a given feature map $F \in R^{C*H*W}$, the channel attention $A_{channel}(F)$ is a $C-$dimensional vector, each entry of which corresponds to one channel of the feature map F. In a neural network, the channel attention is usually calculated by adding a global average pooling layer behind the convolution and ReLU layer.

Similarly, the spatial attention coefficient calculates feature map's mean in the dimension of depth C:

$$A_{spatial}(F, h, w) = \frac{1}{C} \sum_{i}^{C} F_{h,w}(i). \tag{3.2}$$

As such, a feature map $F \in R^{C*H*W}$ would have a spatial attention matrix $A_{spatial}(F)$ of $H * W$ size (so-called attention heat map), each entry of which corresponds to a spatial location of the feature map.

In previous works [49, 51], such attention coefficients are usually passed through a sigmoid layer to be normalized into a $(0, 1)$ range, and then be multiplied back to the feature map. It guides the model to dynamically put better attentions, but it can hardly remove feature components for neural network acceleration. In this work, we propose to overcome the above challenge by using a further-binarized mask to improve the attention mechanism in pruning context. Based on this, two dynamic feature map pruning schemes are developed: channel-wise pruning and spatial-wise column pruning.

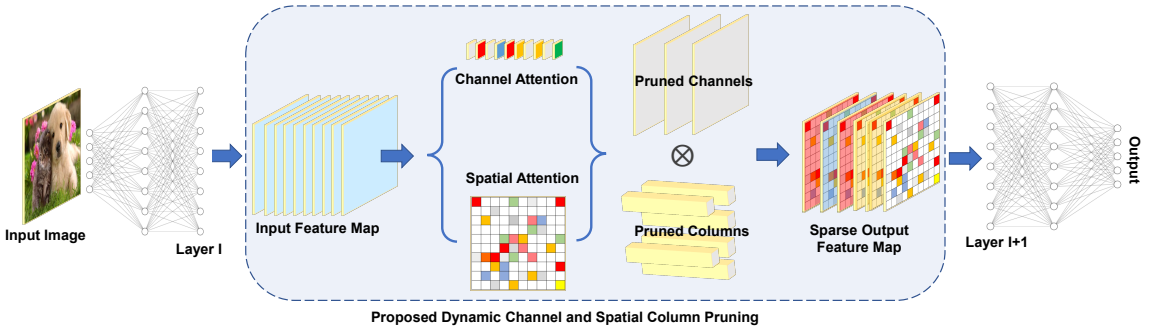


Figure 3.2: Attention-based dynamic feature map pruning framework.

3.1.3 Inference Optimization: Dynamic Feature Pruning

Dynamic channel pruning: During neural network forward phase, a common convolutional layer will generate a 3-dimensional feature map $F \in R^{C*H*W}$ (the first batch dimension is omitted for simplicity). Based on Eq. 3.1, channel-wise attention vector $A_{channel}(F)$ is obtained to evaluate the importance of each channel. With these coefficients, a binary mask $M_{channel}(F)$ will be generated, which is a C-length binary vector to determine the redundancy removal. The channel-wise mask can be generated by setting the mask of top-k attention entries to be *True* or *False*, as shown in Eq. 3.5.

$$M(F,c) = \begin{cases} True, & \text{If } c \in topk(A_{channel}), \quad k = int(p * C); \\ False, & \text{Otherwise.} \end{cases} \tag{3.3}$$

Here, *topk* returns to the indexes of all top-k entries in the channel attention vector. The channel mask *True* indicates the current feature map channel will be reserved, while the other channels whose masks are set to *False* will be masked out and not participate in the next layer's convolution computation. The value of k depends on the overall feature map channels C and the reserved feature percentage p. Note that, the selection of k is dependent on both input images and models, e.g., some images can have more empty backgrounds while

others may not. In this work, we select the per-layer feature reserving percentage based on our later layer sensitivity analysis. There are also other filter-pruning works that adopt self-adaptive global thresholding, which we will consider as future work.

Different from static channel pruning that the selected channels are permanently pruned, in our case the pruned channels are dynamically pruned based on the attention w.r.t. the current input. More specifically, for other inputs which use this channel, it can be fully recovered by the input dependent new binary mask. As such, the proposed method can achieve per-input redundancy removal and larger pruning ratio than static methods.

Dynamic spatial column pruning: Similar to the channel pruning, dynamic spatial column pruning removes feature map columns at different spatial locations according to the spatial attention coefficients. Based on the spatial attention coefficients, a mask $M_{spatial}(F)$ will also be generated. For spatial column pruning, the mask $M_{spatial}(F)$ is a $H * W$ matrix, each entry of which corresponds to reserving or removing a column of the feature map:

$$M(F,h,w) = \begin{cases} True, & (h,w) \in topk(A_{spatial}), \quad k = int(p \ * \ H \ * \ W); \\ \\ False, & \text{Otherwise.} \end{cases} \tag{3.4}$$

topk here returns to all the 2-dimension indexes with the top attention coefficients. The same as channel pruning, the spatial mask will be applied onto the feature map and the unimportant columns will be removed from the feature map.

Note that both of the dynamic channel and spatial column pruning target at pruning redundant feature maps, which is different from previous filter weights pruning: the removed feature map components skip the further convolution operation and hence reduce the computation load in the next layer.

Dynamic control unit overhead: The dynamic feature channel and spatial column pruning requires extra computation to derive the top-k channels and columns, which potentially poses certain overhead to the original DNN model. In this part, we analyze such

overhead and demonstrate that such overhead is negiligible compared to the overall convolution workload. Specifically, we randomly select a common convolutional layer setting from the middle layers of VGG network, e.g., 256 filters with 3×3 filter size, and 256 feature channels with 14×14 feature map size. In such case, the original convolution takes $256 \times 3 \times 3 \times 256 \times 12 \times 12 = 84$M *multi-add* operations. However, our channel-wise and column-wise aggregation (Eq. 3.1 and Eq. 3.2) only involves $2 \times 256 \times 14 \times 14 = 0.1$M *add* operations and two top-k operations. Therefore, the overhead of our dynamic unit constitutes less than 1% of the original convolutional computation.

3.1.4 Training Optimization: Targeted Dropout

To relieve the model's dependency on less important feature components, we concurrently propose a optimized training algorithm *TTD: Training with Targeted Dropout*. With the new training algorithm, model's accuracy drop resilience against dynamic pruning hurt will be greatly improved, thus providing better dynamic pruning performance.

Training with Targeted Dropout The design motivation of *TTD* is to relieve the target model's prediction dependency on less important feature components, so that pruning these components will induce less damage to the model performance. Therefore, we propose to integrate dropout mechanism into the model training process. By doing dropout, the model inference process can gradually become robust to the dynamic feature map dropping. Meanwhile, to mimic the similar dropping effects as attention-based pruning, the dropout must also target at dropping attention-based less-important feature components. Note that, in this targeted dropout, the targeted dropped channels and spatial locations can dynamically change and be potentially recovered during future rounds of training. This is different from filter pruning and fine-tuning (in which the filters are permanently pruned) but more similar to the filter dropout method that filters are dropped and recovered in different iterations. Therefore, we name our method as the *training with targeted dropout (TTD)* method. This is also the main difference between our targeted dropout and random

dropout: the latter one is usually used for different purposes, e.g. to avoid over-fitting.

For implementation, the main difference of our *TTD* training algorithm with traditional training is we add a dynamic targeted dropout layer after each convolutional layer. During the forward phase, the targeted dropout layer will dropout the non-important feature map components (channels and columns). This can be done by conducting element-wise multiplication with the generated attention binary mask in Eq. 3.5:

$$\begin{aligned} F' &= F \otimes M_{spatial}(h, w), \\ F'' &= F' \otimes M_{channel}(c), \end{aligned} \tag{3.5}$$

where $\otimes$ denotes the element-wise multiplication. During multiplication, the mask values will be broadcasted: spatial attention coefficients will be broadcasted and multiply with every entry along the channel dimension, and vice versa. F'' is the final feature map output with targeted channel-wise and column-wise sparsity. During the backward phase, the dynamic dropout layer will conduct the regular back-propagation without any operations.

By introducing the attention-based targeted dropout effect, the *TTD* training will gradually relieve the model's prediction dependency on less-important features (since they are often dropped during training), but increase their focus on most important ones. Therefore, the dynamic pruning of those non-important feature components will induce minimum or no effects to the model accuracy during test-phase inference.

Layer Sensitivity Analysis and Dropout Ratio Ascent The *TTD* algorithm introduces the targeted dropout into the training process to enable the dynamic pruning during test. Whereas, different layers of a model can have varied amount of redundancy. This requires the targeted dropout ratios to be carefully tuned otherwise it will greatly hurt the model convergence speed and final accuracy. Therefore, we draw some experience from previous static pruning works and follow the layer sensitivity analysis practice to set the dropout ratio for different layers.

Take VGG16 network as an example. VGG16 has 5 convolutional blocks with [2, 2,

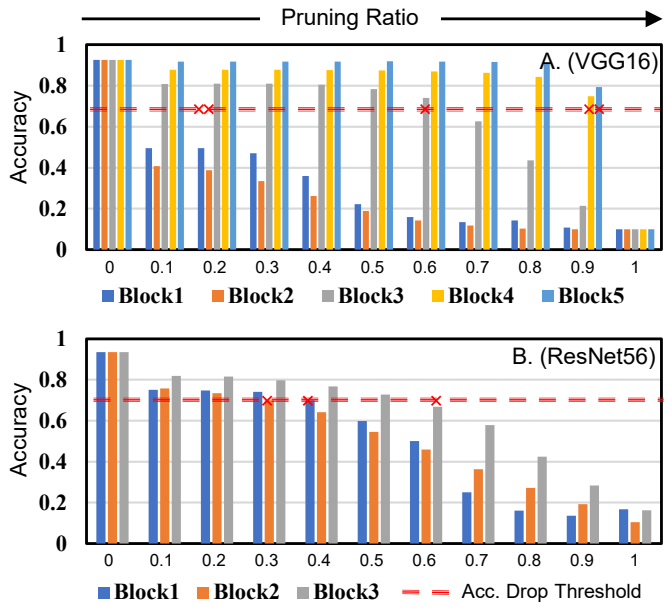


Figure 3.3: Block Sensitivity Analysis.

3, 3, 3] convolutional layers and one 2x2 MaxPooling layer at the end of each block. To avoid massive hyper-parameter tuning and to maintain policy consistency with block-wise ResNet structure, we analyze the average block sensitivity and set an aggressive dropout upper bound for each block. For example, Fig. 3.3 shows the block sensitivity analysis for VGG and ResNet. Take VGG as an example: An aggressive pruning ratio per block e.g. [0.2, 0.2, 0.6, 0.6, 0.9] can cause the pruned model's accuracy dropping to less than 70%, which can be hardly recovered back. Therefore, we set this threshold as the upper bound pruning ratio, and then use dropout ratio ascent during *TTD* training. The dropout ratio will start with a warm-up ratio, for example 0.1 for each block. After the model converges during the current ratio, we will ascent the ratio for each block with a small step-size (e.g. 0.05) to try reaching the maximum pruning ratio. And the training will stop when the target pruning ratio and a satisfying accuracy is achieved.

Table 3.1: Experiment Results on CIFAR and ImageNet Datasets.

CNN Models	Pruning Methods	Baseline Accuracy(%)	Baseline FLOPs	Final FLOPs	FLOPs Reduction(%)	Final Accuracy(%)	Accuracy Drop(%)
VGG16 (CIFAR10)	L1 Pruning* [46]	93.3	-	2.06E+08	34.2	93.4	**-0.1**
	Taylor Pruning* [52]	93.3	-	1.85E+08	44.1	92.3	1.0
	GM Pruning* [53]	93.6	-	2.11E+08	35.9	93.2	0.4
	FO Pruning* [48]	93.4	-	1.85E+08	44.1	**93.3**	0.1
	Proposed	93.3	3.13E+08	**1.46E+08**	**53.5**	93.1	0.2
ResNet56 (CIFAR10)	L1 Pruning* [46]	93.0	-	0.91E+08	27.6	93.1	-0.1
	Taylor Pruning* [52]	92.9	-	0.71E+08	43.0	92.0	0.9
	FO Pruning* [48]	92.9	-	**0.71E+08**	**43.0**	**93.3**	**-0.4**
	Proposed	93.0	1.28E+08	0.80E+08	37.4	93.2	-0.2
VGG16 (CIFAR100)	L1 Pruning* [46]	73.1	-	1.96E+08	37.3	72.3	0.8
	Taylor Pruning* [52]	73.1	-	1.96E+08	37.3	72.5	0.6
	FO Pruning* [48]	73.1	-	1.96E+08	37.3	73.2	-0.1
	Proposed: Setting-1	73.1	3.13E+08	**1.87E+08**	**40.4**	**73.2**	**-0.1**
	Proposed: Setting-2	73.1	3.13E+08	**1.72E+08**	**44.9**	72.9	0.2
VGG16 (ImageNet100)	L1 Pruning* [46]	78.5	-	0.76E+10	50.6	76.6	0.8
	Taylor Pruning* [52]	78.5	-	0.76E+10	50.6	77.3	0.6
	FO Pruning* [48]	78.5	-	0.76E+10	50.6	79.5	-1.0
	Proposed: Setting-1	78.5	1.52E+10	**0.74E+10**	**51.2**	**79.6**	**-1.1**
	Proposed: Setting-2	78.5	1.52E+10	**0.69E+10**	**54.5**	79.4	-0.9
MobileNetV2 (ImageNet-1K)	AMC [54]	71.8	-	2.49E+8	30.0	70.8	1.0
	ThiNet [55]	71.8	-	1.96E+8	44.7	63.7	6.4
	DC Pruning [56]	71.8	-	1.96E+8	44.7	64.2	5.9
	Proposed	71.8	3.56E+8	**1.96E+8**	**44.7**	**69.4**	**2.2**

* Method performance is referred from [48, 53].

3.1.5 Experimental Evaluation

Experimental Setup We evaluate our proposed attention-based dynamic feature pruning methods on multiple common benchmarks, e.g. VGG and ResNet on CIFAR10/100, as well as ImageNet. By comparing the commonly used floating point operations (FLOPs) reduction and the accuracy, we show that our attention-based dynamic pruning method could outperform previous SOTA pruning methods with large margins.

We then evaluate our attention-based OFA model deployment optimization. Specifically,

we demonstrate that one unified trained OFA model could be flexibly reconfigured to be multiple models to meet different computing resource requirement, meanwhile achieving similarly optimal computing-accuracy trade-offs as individually trained models, and most importantly, without any re-training cost.

Throughout our experiments, we use the deep learning framework PyTorch. For model training settings, we use the same data augmentation including random horizontal flip, random crop and 4-pixel padding. For the model optimizer, we use the default SGD optimizer with momentum factor as 0.9. The learning rate scheduling policy is cosine decaying [57] with 0.1→0 for training process of all models.

Attention-based Feature Pruning Evaluation

We first evaluate our attention-based feature pruning method. Several state-of-the-art static pruning methods are chosen as baseline for comparison, including ℓ_1-norm Pruning [46], Taylor Pruning [52], Geometric Mean Pruning (GM) [53], Function-Oriented Pruning (FO) [48], AMC Pruning [54], ThiNet [55] and DC Pruning [56].

VGG16 on CIFAR10 The first model we evaluated is the VGG16 on CIFAR10. VGG16 has 13 convolutional layers in 5 blocks. For each block, there are 2-2-3-3-3 layers with 64-128-256-512-512 filters (of 3x3 filter size) per layer. The experimental results are shown in Table 3.1. On this model, the best channel pruning ratio per block we find is [0.2, 0.2, 0.6, 0.9, 0.9], which matches the sensitivity analysis. Beyond this ratio, the accuracy can hardly be compensated by TTD training. Since the feature map spatial size is too small (e.g. the last 9 layers' feature map size are ranged from 8x8 to 2x2), pruning spatial columns on VGG16 always brings unrecoverable accuracy drop. Therefore, spatial pruning ratio for this model is set to 0 for all layers.

Comparing our pruning ratio with previous work, our channel pruning ratios [0.2, 0.2, 0.6, 0.9, 0.9] greatly outnumber the previous state-of-the-art static methods: The best FO pruning [48] can only achieve [0.17, 0.1, 0.1, 0.45, 0.65]. This proves our hypothesis

that our dynamic method can aggressively and accurately remove more dynamic per-input redundancy which static method fails to. As a result, our best performance of dynamic pruning is 53.5% FLOPs reduction with 0.2% accuracy drop, 9.4% higher FLOPs reduction than the best FO Pruning with only 0.1% accuracy difference.

ResNet56 on CIFAR10 The next model we evaluated is ResNet56 on CIFAR10. Different from VGG16 with wide layers, ResNet56 has three groups containing 16 convolutional layers per group. And it has at most 64 convolutional filters in one layer. Thus, the channel redundancy in ResNet56 is relatively limited compared to VGG16. By contrast, its feature map size are from 32×32 to minimum 8×8, which can contain more redundancy. Therefore, for this network, we prune less channels but more spatial columns. Due to the skip connection, we need to maintain the same size of input and output channels of even layers in every group. Therefore, the dynamic pruning is only conducted in the odd layers in the group. The dynamic pruning setting for this network is channel-wise pruning ratio: [0.3, 0.3, 0.6], and spatial-wise pruning ratio: [0.6, 0.6, 0.6]. The final performance is 37.4% FLOPs reduction with slight 0.2% accuracy improvement, which is comparable with previous FO pruning results.

VGG16 on CIFAR100 We also test our dynamic pruning method's performance using VGG16 on CIFAR100. One conservative pruning setting (Setting-1) is [0.2, 0.2, 0.2, 0.8, 0.9] channel-wise ratio, with zero spatial pruning ratio for all layers due to the same reason of small feature map size. Compared to all baseline methods, this setting could achieve the highest FLOPs reduction (40.4%) as well as the highest accuracy (73.2%). We also conduct a more aggressive pruning setting (Setting-2) with [0.3, 0.2, 0.2, 0.9, 0.9] channel-wise ratio and zero spatial ratio. In this setting, a 0.5% accuracy drop is observed but we could push the FLOPs reduction to 44.9%, which is 4.5% more than Setting-1 model.

VGG16 on ImageNet100 To validate our method's performance on large-scale image datasets, we test our pruning methods using VGG16 model on ImageNet100 dataset with

the same setting as [48]. Different from CIFAR datasets, images in ImageNet has a size of 224×224×3. Therefore, the spatial dimension of the feature map in this model is much larger than CIFAR model. With the similar FLOPs reduction as the baseline, our setting-1 model brings 51.2% FLOPs reduction. The setting is [0.1, 0, 0, 0, 0.2] for channel-wise ratio, and [0.5, 0.5, 0.5, 0.5, 0.5] for spatial ratio. In Setting-2, we further achieve 54.5% FLOPs reduction with increased spatial ratio [0.5, 0.5, 0.5, 0.6, 0.6] and without accuracy drop. This further proves our method's good capability in removing model redundancy on large-scale datasets.

MobileNetV2 on ImageNet-1K We finally validate our method's performance on light-weight models such as MobileNetV2 on ImageNet. We compare our method with several previous SOTA works [54–56]. The results are shown in the last row in Table 3.1. As the table shows, AMC pruning achieves relatively less FLOPS reduction (30%) with 1% accuracy drop. And the other two methods achieve more FLOPS reduction but incurs larger accuracy drop, e.g., 5.9%∼6.4%. With the similar settings, our proposed method achieves same FLOPS reduction (i.e., 44.7%) but only incurs 2.2% accuracy drop, which shows our method's capability in removing feature redundancy on large-scale datasets.

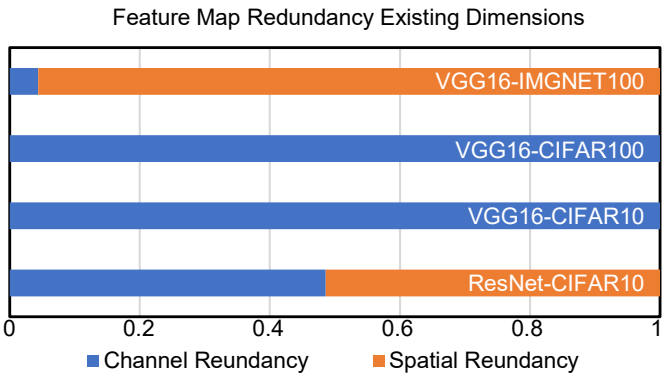


Figure 3.4: The feature redundancy varies in channel and spatial dimensions.

The Redundancy Composition in Various Dimensions In this part, we demonstrate that different neural network models on varied datasets can have distinct feature map redundancy composition. To show it, we calculate the channel and spatial redundancy separately for each of our previous experiments, the results of which is shown in Fig. 3.4. We could find that the model redundancy can exist in very arbitrary dimensions. For example, VGG-style plain network on small-resolution datasets (VGG-CIFAR10 and VGG-CIFAR100) mostly have over-redundant channels and thus more channels can be pruned without hurting accuracy. While for large-resolution datasets like VGG-ImageNet100, more spatial redundancies can be easily pruned without hurting accuracy since larger background areas could be removed in 224x224 resolution images without hurting the classification information. ResNet is more special, as this network shows similar spatial and channel redundancy (ResNet-CIFAR10). Our hypothesis is that due to the skip connection, the spatially pruned features can still pass onto next blocks by the skip connection. Therefore, pruning spatial features incurs less information loss for ResNets than plain VGG networks.

Effectiveness of Channel+Column Composed Pruning The above difference of feature redundancy distribution highlights the fact that the feature map redundancy exist in various of dimensions for different input resolutions and model structures. For example, images of ImageNet resolution can have more spatial redundancy lying in the background spatial areas. While for lower-resolution CIFAR images, channel-wise redundancy exists more due to the simplicity of the dataset. Therefore, previous channel-only pruning [46,48,52,53] can hardly remove the spatially existed redundancies in the feature maps. By contrast, our novel spatial-column-wise pruning and its combination with channel-wise pruning enables more flexible and thorough feature map redundancy removal, which thus help us achieve the optimal performance for most test settings.

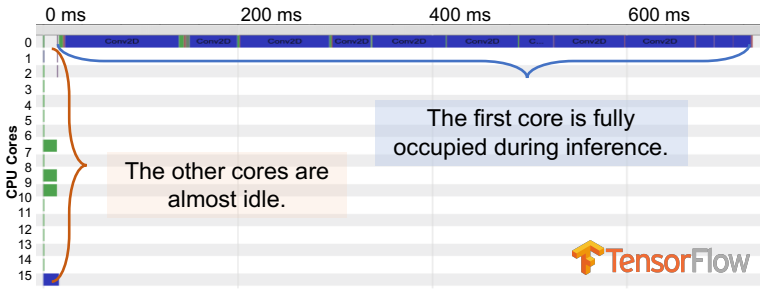


Figure 3.5: Core under-utilization during CNN inference

3.2 DCCNN: Structural Decoupling by Connection Pruning

3.2.1 Problem Motivation

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 [58–60], weight sparsity [61]), hardware specific optimization (*e.g.*, loop-untiling [62], fuse-layer [63]), *etc.*

Although these optimization works have demonstrated their effectiveness, they are mostly limited by the sequential CNN computation flow in a sequential layer-by-layer manner [64, 65]. 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. 3.5 shows a profiled CPU time-line when running VGG-16 inference on an Intel 16-core Xeon CPU. While

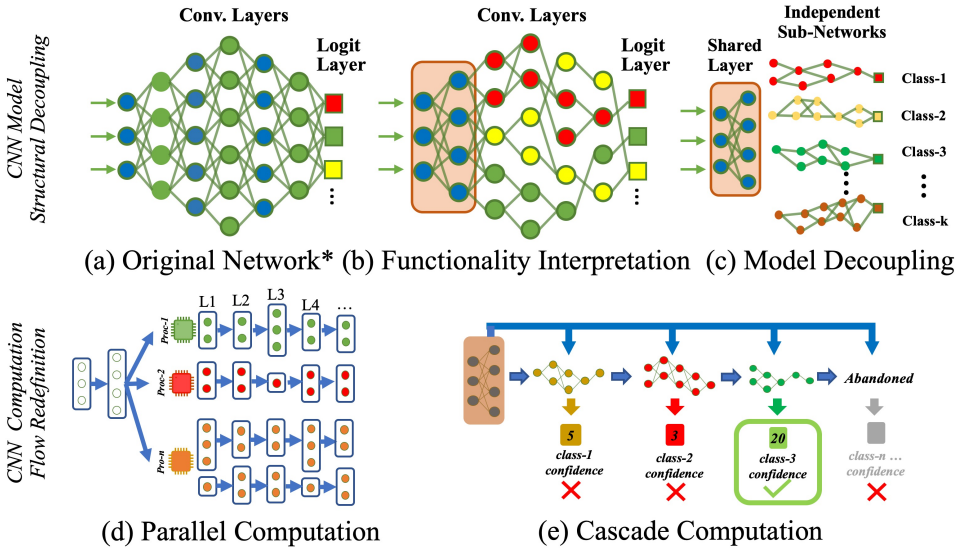


Figure 3.6: DC-CNN framework overview

the first core keeps running for 720ms, 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. 3.6 (a)-like CNN models. As shown in Fig. 3.6 (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. 3.6 (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. 3.6 (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. 3.6 (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.

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:

$$\begin{aligned} &Z^n(x) = F^L \circ \cdots \circ F^l \circ \cdots \circ F^1(x), \\ &where\ A^l(x) = F^l \circ \cdots \circ F^1(x) = \coprod A_i^l(x), \end{aligned} \tag{3.6}$$

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:

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:

$$Act_i^n = E(||A_i^l(x_p^n)||_1), \ \ p \text{ is the pool size of } n^{th} \text{ class}, \tag{3.7}$$

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.

Activation Significance Analysis

According to the first-order Taylor expansion [52], 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:

$$Grad_i^n = ||\frac{\partial Z^n(A_i^l)}{\partial A_i^l}||_1, Z^n(A_i^l + \delta) \approx Z^n(A_i^l) + \frac{\partial Z^n(A_i^l)}{\partial A_i^l} \cdot \delta \tag{3.8}$$

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.

Filter's Class-Specific Index

Combining both activation preference and activation significance based on Taylor Expansion, we derive the class-specific index – S_i^n to identify a convolutional filter's function exclusiveness to a specific class:

$$S_i^n = Act_i^n * Grad_i^n, \quad n^* = Argmax_n(S_i^n), \tag{3.9}$$

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

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. 3.7 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. 3.9, 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. 3.7 (a) and (b). Such exclusiveness between classes implies there exists great independence between different set of class-specific convolutional filters.

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. 3.8, in the VGG-16 model trained on CIFAR-10, we identified and

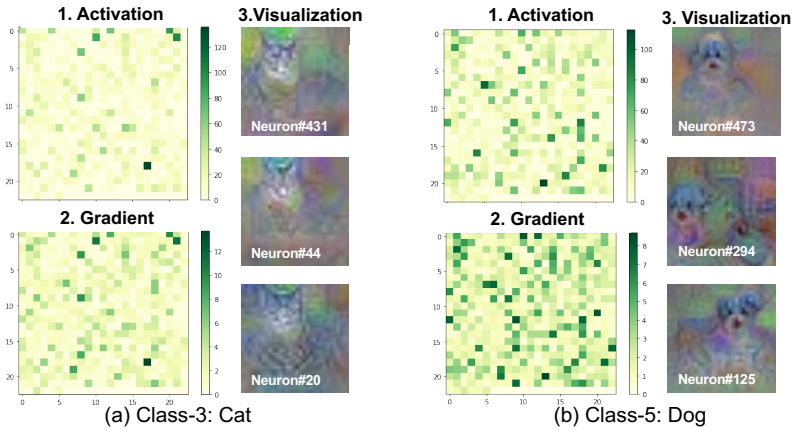


Figure 3.7: Class-specific filter functionality interpretation

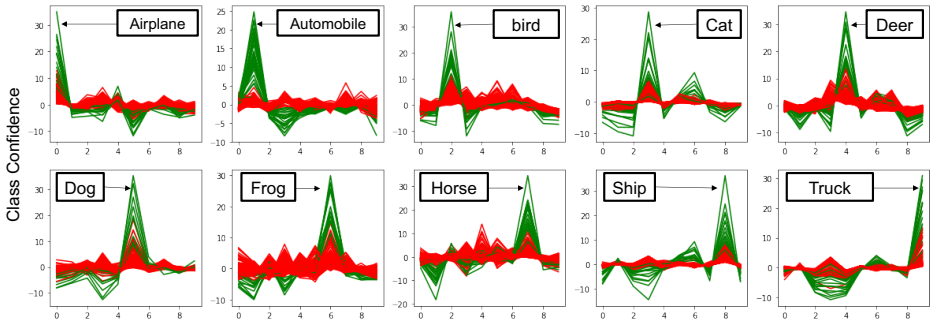


Figure 3.8: Per-Class Critical Path's Classification Confidence

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.

3.2.2 Overview of Structural Model Decoupling

Fig. 3.9 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

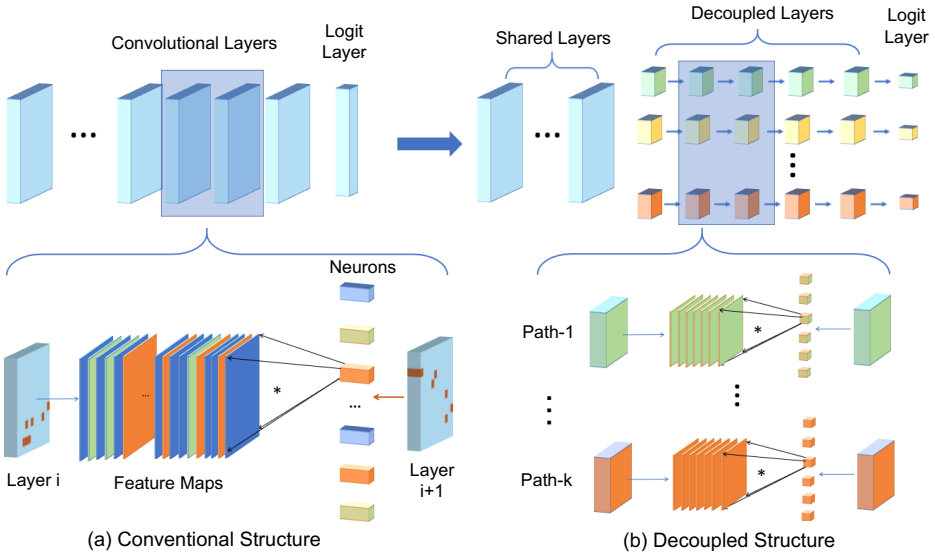


Figure 3.9: Comparison of original and decoupled structures

end of each corresponding path to generate the confidence for each class.

In the original structure in Fig. 3.9 (a), each filter needs to convolve with all feature maps produced by the previous layer. While in the decoupled structure in Fig. 3.9 (b), every class-specific filter only convolves with intra-path feature maps, 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.

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. 3.10 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.

3.2.3 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.

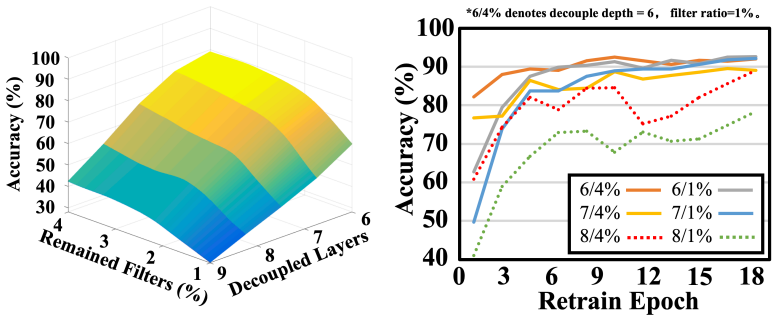


Figure 3.10: Structural decoupling configuration search space

Parallel Computing Paradigm Overview

As illustrated in Fig. 3.6 (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 [66,67], 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.

Performance Enhancement Analysis

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. 3.9 (b). Thus, the computation workload reduction is:

$$C^{-}(N, D) = (1 - \frac{1}{N}) \sum_{d=2}^{D} C_d. \tag{3.10}$$

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.

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:

$$T^{-}(N,P) = \alpha \times \sum_{d=2}^{D} T_d, \ where\ \alpha = \begin{cases} (1-\frac{1}{P\times N}), \ \text{if } N \geq P, \\ (1-\frac{1}{N\times N}), \ \text{if } N < P, \end{cases} \tag{3.11}$$

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

3.2.4 Cascade Computing Paradigm Overview

As illustrated in Fig. 3.6 (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 at any time, which greatly reduces the runtime memory occupation and the potential memory access overhead [67]. Here we analyze the theoretical latency and memory reduction.

Latency Reduction Analysis Without parallelism, the latency reduction in cascade flow only comes from the computation workload reduction in Eq. 3.10. Therefore, the latency reduction can be directly formulated:

$$T^{-} = (1 - \frac{1}{N}) \sum_{d=2}^{D} T_d. \tag{3.12}$$

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

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.

Table 3.2: Scalability Test of Proposed Model Structural Decoupling

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

3.2.5 Experimental Evaluation

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 3.2. 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 generality on large-scale inputs.

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 3.3: 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%

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%∼41% FLOPs reduction on all settings with negligible accuracy loss.

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 3.3 and Table 3.4. 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. 3.11, and the practical reduction is averaged on 100 running.

Table 3.4: 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%

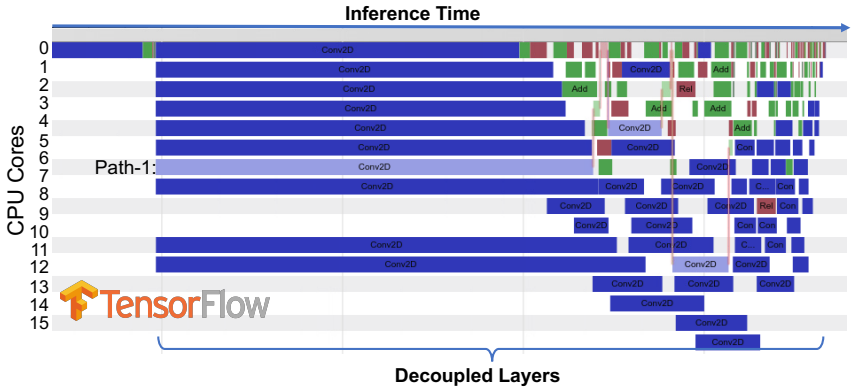


Figure 3.11: Parallel flow: multi-core utilization enhancement

As Table 3.3 shows, our parallel computing paradigm on CIFAR brings a 24∼34% latency reduction compared to original sequential computation flow. The ImageNet results in Table 3.4 show similar latency reduction rates. With increasing parallelism from CPU-2 to CPU-16, the latency reduction ratio increases to the maximum ∼33.7%, which exactly matches the theoretic analysis result (33.6%) calculated by Eq. 3.11.

Finally, to better illustrate our parallel paradigm, Fig. 3.11 shows the decoupled layers' parallel computation details, which demonstrate the great enhancement of core utilization.

Table 3.5: Performance of Cascade Computation for CIFAR10 and ImageNet10

Dataset & Model Settings	Mobile Platforms	Original Model Latency (ms)	Original Model Memory (MB)	Decoupled Model Latency (ms)	Decoupled Model Memory (MB)	Theoretical #FLOPs Reduction	Theoretical Latency Reduction	Practical Latency Reduction	Practical Memory 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.

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 3.5 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. 3.12. 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%$\sim$45.3% latency reduction on different platforms. Considering different platforms' runtime dynamics, the latency reduction is roughly consistent with our theoretical analysis.

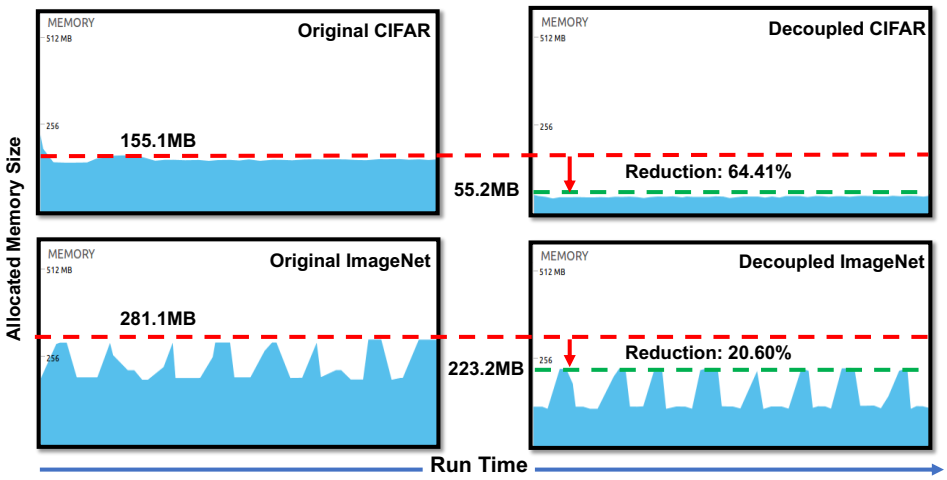


Figure 3.12: Cascade flow: memory reductions on Nexus-5X.

The memory reduction gain is also significant and consistent for different settings according to the results. Fig. 3.12 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. Through our structural decoupling and cascade computing, 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.

Chapter 4: Software-Hardware Co-Optimization

In this chapter, we then introduce our works in software-hardware co-optimization, *MT-Graph* and *TA-DNN*. Specifically, *MT-Graph* aims to conduct runtime-aware operator graph scheduling for inference speedup. *TA-DNN* aims to renovate the DNN design procedure by considering the GPU micro-architecture and compiling information.

4.1 MT-Graph: Runtime-Aware Multi-Tenant Scheduling

4.1.1 Problem Motivation

As deep neural networks (DNNs) have demonstrated superior performance in vast cognitive tasks [37, 68, 69], the expectations for DNN-powered intelligence have grown rapidly over the past few years. In addition to the real-time needs of DNN optimization regarding its deep structures and heavy workloads [70–72], recent real-world applications further require *multi-tenant DNN computation* for even compound tasks [73–75]. For example, it is critical for an autonomous driving system to inference multiple DNN models simultaneously on the same hardware for segmentation [76], detection [77], and classification [78]. And for larger-scale cases, such multi-tenant computing necessity also emerges in cloud computing clusters and industrial-level data centers for resource utilization improvement, drawing significant attention from intelligence services providers, such as Microsoft and NVIDIA [11, 79, 80].

The ***multi-tenant DNN inference*** exacerbates the computational complexity on top of existing DNN problems. However, the corresponding computing support is still relatively backward. As the major platform for the multi-tenant inference — current GPUs' computing strategies are still limited to traditional approaches of *sequential execution* (e.g.,

MPI-processing [81]) and *parallel/concurrent execution*[1] (e.g., NVIDIA multi-Stream execution [82]). As demonstrated in Fig. 4.1, these limited strategies cannot achieve satisfying performance for multi-tenant DNN inference: **(a)** Although the sequential execution dedicates the entire GPU's resource to each model and achieves the shortest per-model inference latency as shown in Fig. 4.1 (a), continuous ***resource under-utilization*** is inevitable due to the single-operator execution (e.g., *conv*, *pooling*), not to mention the cumulative overall ***runtime latency***. **(b)** For the concurrent execution in Fig. 4.1 (b), though indispensable parallelism for multiple models earns latency optimization to a certain degree, it hasn't touched the particular computing complexity in multi-tenant DNN inference. Taking the first round of convolution from the three DNN models (i.e., *A1*, *B1*, *C1*) as an example, simple parallelism would introduce considerable ***contention overhead*** as operators can compete for computing resources simultaneously. While looking into later stages of this concurrent execution, GPU under-utilization strikes back due to the ***unbalanced scheduling*** for different model depths.

Thus, to strive for optimal runtime latency and resource utilization, the multi-tenant DNN inference raises particular GPU scheduling requirements not only for analyzing and relieving *local* operator contention, but also for managing *global* model concurrency balance as per model structure divergence. Bringing this "*local-global*" need into the existing DNN execution stack as shown in Fig. 2.3, we can see that, it calls for comprehensive collaboration from ***the graph-level operator scheduling***, ***the runtime-level resource awareness***, as well as ***the hardware scheduler support***. However, most existing DNN scheduling methods are limited in a single-level optimization scope. For example, many works are proposed singularly for low-level intra-operator optimization, such as loop tiling and unrolling [13, 83, 84]; Similarly, many graph-based scheduling works focus only on high-level inter-operator fusion/substitution optimization [22, 23, 85]. As a result, these methods fail to meet the cross-level scheduling requirement by the multi-tenant DNN inference.

In this work, we propose a *runtime-aware scheduling framework for efficient multi-tenant*

[1]We treat "parallel" and "concurrent" as the same meaning in our work, according to the definition of "concurrency" in the NVIDIA document [82].

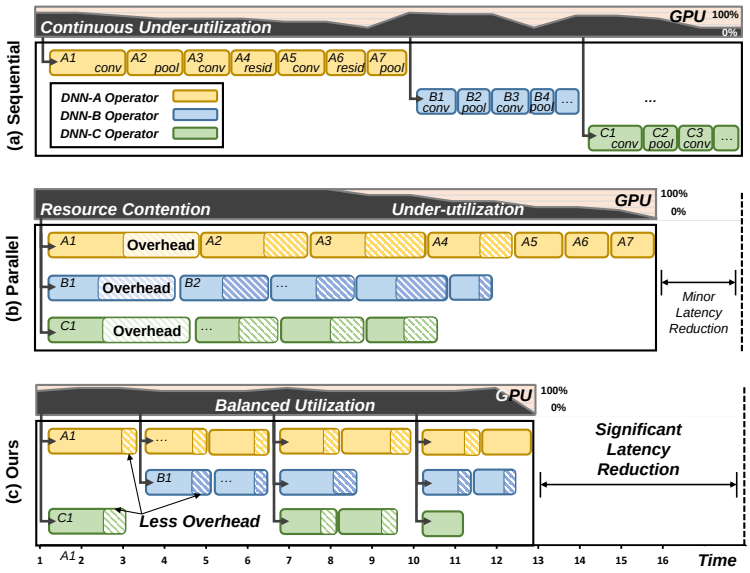


Figure 4.1: Scheduling for multi-tenant DNN inference on a single GPU

DNN inference on GPU, which *automatically* coordinates concurrent DNN computing in different execution levels. As shown in Fig. 4.1 (c), the proposed method could take both the local operator contention and the global model structural divergence into consideration. The final scheduling method wisely adjusts model concurrency by interleaving operators for less contention overhead, maintaining a continuously balanced resource utilization across the entire inference process, and eventually improving the runtime efficiency. To achieve such a scheduling target, we make the following contributions: We first *abstract the multi-tenant DNN inference scheduling* as a fine-grained concurrency control problem. Incorporating the GPU multi-stream and synchronization mechanisms, multiple concurrency control levels are identified in the GPU inference flow to provide the fundamental support for the scheduling optimization; Based on the problem abstraction, *a unified scheduling Intermediate Representation (IR)* is specified to formulate the scheduling factors by taking both graph-level and the runtime-level execution mechanism into consideration, and eventually build a structural search space for the final scheduling optimization; In the established scheduling search space, we transform multi-tenant scheduling into an optimization problem and propose an

automated ML-based searching algorithm to find the optimal scheduling strategy on GPU. Specifically, the GPU runtime resource is profiled and adopted as the searching cost, granting the whole solution with expected runtime awareness.

4.1.2 The Scheduling Framework

Fine-grained Scheduling Problem Abstraction

This work targets at efficient multi-tenant DNN inference on GPUs. Considering the applications such as autonomous driving systems, we specify it as a compound task consisting of N independent DNN models sharing the same input for different inference sub-tasks. Demonstrated as Fig. 4.2 (a), each DNN inference sub-task consists of a series of operators, such as *conv*, *bn*, *relu*, *pooling*, etc, which must be performed in certain order according to the data flow dependency. While across DNN models, operators are independent and thus could be flexibly scheduled with certain degrees of concurrency. Our optimization objective is to minimize the overall latency of N inference sub-tasks, which is the overall time from the earliest starting time of the tasks to the latest ending time.

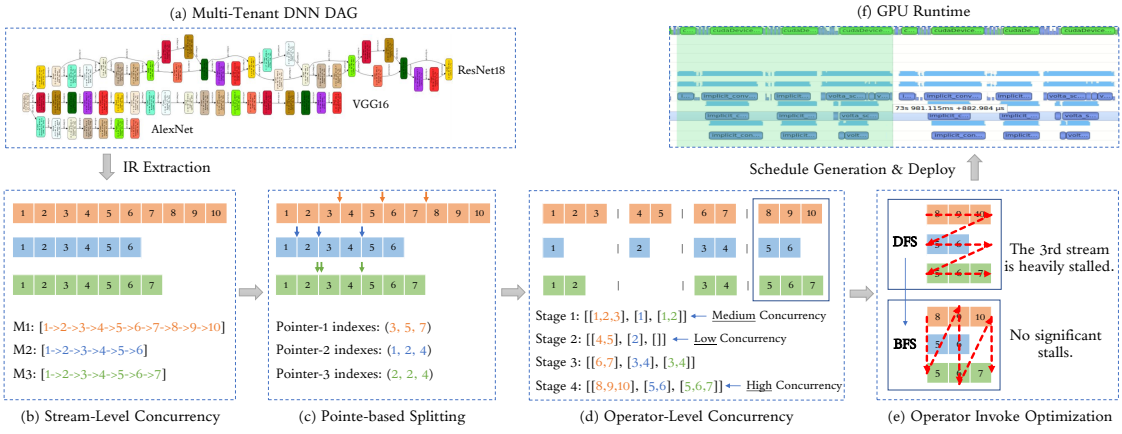


Figure 4.2: Overview of our automated scheduling strategy search framework.

The key of multi-tenant scheduling is to manage the concurrency for consistent and balanced resource utilization. Therefore, we abstract the multi-tenant DNN inference scheduling as a fine-grained concurrency control problem through the following steps: **(a)** *Achieving the* ***stream-level*** *concurrency*: We allocate one *GPU processing stream* for each model to achieve the concurrency (Fig. 4.2 (b)). However, even with certain concurrency, native GPU stream-based scheduling dispatches operators without dedicated scheduling management. **(b)** *Finer-grained* ***stage splitting***: To achieve finer-grained operator-level concurrency control, we insert synchronization barriers, namely ***pointers***, to split each stream's operator sequence into several shorter stages (Fig. 4.2 (c)). Such stage splittings ensure the operators to only share the assigned resources in the same stage, thus supporting the stage-level concurrency control. **(c)** ***Stage-level*** *concurrency control*: By adjusting where the pointers are inserted, we could control how many operators are assigned in each stage. This enable us to reduce or increase the concurrency in a fine-grained manner to manage the resource utilization (Fig. 4.2 (d)). **(d)** ***Intra-stage*** *operator invoking optimization*: After deciding the scheduling strategy, our final step is the scheduling deployment. During this implementation, we also optimize the operator invoking logic to prevent the invoking overhead of early streams from stalling later ones (4.2-e), as we will introduce later.

Unified Intermediate Representation Design

As a multi-tenant DNN inference task consists of N parallelable models: $M_1, M_2, ..., M_N$. We represent each DNN model by one stand-alone operator sequence[2]:

$$\begin{aligned} \mathrm{M}_1 &: [1,\ 2,\ ...,\ a], \\ \mathrm{M}_2 &: [1,\ 2,\ ...,\ b], \\ \mathrm{M}_N &: [1,\ 2,\ ...,\ c], \end{aligned} \tag{4.1}$$

[2]For multi-branch models like ResNets, we also serialize the operators into one sequential sequence as their *intra-model concurrency* is limited. Such a representation enables us to better optimize the *inter-model concurrency* in the multi-tenant inference scenario.

where M indicates a DNN model, each number in one list indicates one operator's index, and a (or b, c) is the largest index of the DNN's operators.

Stream: To satisfy the sequential dependency per model, we assign each model to one stand-alone *GPU processing stream*:

$$S_i \leftarrow \ \mathrm{M}_i, \quad i \in (1, 2, ..., N), \tag{4.2}$$

where S_i indicates the i-th stream. An example with three streams is shown in Fig. 4.2 (b). Operators in one stream can only be launched sequentially, while operators in different streams could be executed concurrently.

The multi-stream mechanism enables the maximum concurrency of DNN inference streams. However, as aforementioned, scheduling by streams alone can only have *stream-level concurrency control*, which is still coarse-grained and does not suffice to manage each operator's associated concurrency during its life span. To control the concurrency in a finer granularity, we then use synchronization barriers to split each stream's full sequence into several shorter stages.

Pointer: We use *pointers* to annotate the appropriate positions where we insert synchronization barriers. An illustrated pointer-based stage splitting is shown in Fig. 4.2 (c)(d). Taking the first stream as an example, a pointer set with three pointers divides the first stream sequence into four shorter ones:

$$\begin{gathered} \rho_1 : (\mathbf{3,\ 5,\ 7}) \ + \ \mathrm{S}_1 : [1, 2, 3, ..., 9, 10] = \\ \mathrm{S}_1^{'} : [1, 2, 3], [4, 5], [6, 7], [8, 9, 10], \end{gathered} \tag{4.3}$$

where ρ_1 is the pointer indexes, S_1 is the original operator sequence, and $S_1^{'}$ is the split sequence with synchronization barriers inserted. Each pointer set splits one stream sequence into several shorter ones, thus enabling a finer-grained concurrency scheduling.

Stage: Between each two pointers, the launched operators form a *stage*. Due to the sync barriers, all operators in the same stage must all finish so as to step into the next stage.

Thus, by controlling how many operators are launched in each stage, we could precisely manage the concurrency in the most fine-grained operator level. An example is given in Fig. 4.2 (d). By inserting the first synchronization barrier, we could enable six operators to concurrently execute in the first stage:

$$\text{Stage 1} : [S_1(1,2,3), \quad S_2(1), \quad S_3(1,2)]. \tag{4.4}$$

By contrast, we could also reduce the concurrency in the second stage by assigning no operators in the third stream:

$$\text{Stage 2} : [S_1(4,5), \quad S_2(2), \quad S_3(\text{None})]. \tag{4.5}$$

Similarly, all stages can be generated with a desired concurrency, thus enabling *operator-level concurrency control*.

Schedule: The final scheduling strategy is composed of multiple stages in the synchronization barriers' ordering, which is represented as a multi-stage nested list:

$$\text{Schedule } \tau : [\text{Stage 1}, \ \text{Stage 2}, \ \text{Stage 3}, ...], \tag{4.6}$$

where τ indicate the composed scheduling strategy, which can have multiple stages, depending on the number of synchronization barriers (i.e., pointers) we used to split each stream sequence. Fig. 4.2 (d) shows an example that uses three sync pointers for four stages. More synchronization enables finer-grained concurrency control, but at the price of potentially higher synchronization overhead.

4.1.3 Automated Scheduling Search

The IR design explicitly defines the scheduling factor and the corresponding strategy for multi-tenant GPU inference. However, it is still challenging to identify the particular scheduling controls given various compound tasks with uncertain DNN structures. As aforementioned, considering the complexity, manual schedule tuning can take considerable efforts

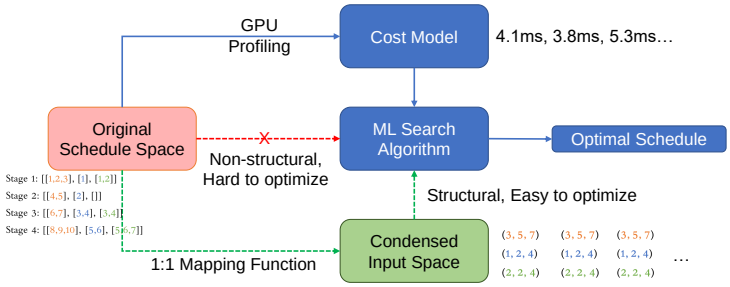


Figure 4.3: The automated scheduling search framework overview.

and also cannot scale with more complicated models' combination and varied GPU platforms. Therefore, we propose to use an ML-based search approach to solve the scheduling problem in an automated manner.

Formulation: Formally, our primary search target is to find an optimal scheduling strategy that yields the lowest latency:

$$\tau^* = \arg\min_{\tau} \ f(\tau), \quad \text{for } \tau \in D_\tau, \tag{4.7}$$

where τ^* is the optimal scheduling strategy, f is the cost model that evaluates the latency of the current schedule τ, and D_τ is the search space of all potential schedules. Specifically, to solve this search problem, three basic components need to be clarified, namely, the search space, the cost model and the searching algorithm.

Search Space is supposed to enumerate all possible scheduling strategy candidates. To represent such a search space, we adopt the scheduling factors from the proposed IR design (i.e., streams, pointers and stages).

As defined in Eq. 4.6, τ can be formulated as a nested list and can be treated as a graph-level scheduling strategy. Although such a nested list is easy to understand and facilitates the deployment process onto GPU, the list-based search space D_τ is non-structural with varied list lengths and can be hard to directly optimize. To solve this problem, we leverage

the one-to-one mapping property between pointer indexes ρ and the schedule lists τ, and shrink the search space to a lower-dimensional pointer index matrix by building an 1:1 schedule mapping function, as shown in Fig. 4.3:

$$\begin{aligned} \rho^* &= \arg\min_{\rho} \ f(\tau), \\ \text{s.t.} \ \ \tau &= T(G, \rho), \quad \text{for } \rho \in D_\rho. \end{aligned} \tag{4.8}$$

Here the scheduling generation function $T(\cdot)$ generates one schedule τ based on two inputs: the graph G and the pointer matrix ρ. As G is usually fixed in a given task, the schedule generation function $T(\cdot)$ maps each pointer matrix to one schedule. Thus, searching schedule could be transformed to searching the pointer index matrix, the latter of which has a much more structural input space. By such transformation, we could thus greatly reduce the optimization difficulty.

Cost Model: With the search space defined, we then require a cost model $f(\tau)$ to evaluate the performance of each schedule candidate. There are two major ways to construct the cost

Algorithm 1 Coordinate Descent Search Algorithm.

Input: The IR of N models $M[N]$, the number of pointers in each model P, the rounds of search R.
Output: The optimal pointer matrix ρ $[N, P]$.
Initialize a dictionary D\{schedule:cost\} to store records.
for rounds $r = 1$ **to** R **do**
  for model $i = 1$ **to** N **do**
    Sample M candidates $\rho_{1:M}[i]$ for the i-th row.
    for the m-th candidate $\rho_m[i]$ **do**
      Profile the latency lat_m by multiple runs.
      Append $\{\rho_m : lat_m\}$ to the records D.
    end for
    Update the i-th row $\rho[i]$ of pointer matrix to the one with the lowest latency by $\arg\min(lat_m)$.
  end for
end for
Sort the global records D by the profiled latency.
Return the schedule ρ with the globally lowest latency.

model: modeling-based or profile-based. The modeling-based method [23] builds hardware-specific modeling to estimate the real runtime performance, which is efficient but can be inaccurate in complex scenarios. The profiling-based method [13] is more accurate but requires physical hardware execution, which can be more time-consuming if the search space is very large.

In this work, we use the profiling-based cost model since our empirical case study shows that, our searching time can be maintained at small scale (∼mins) benefited from our dedicated search space abstraction. Therefore, the profiling-based model can give accurate runtime-aware performance cost and lead to better search performance in our case. For the cost model implementation, we leverage our built infrastructure, which could efficiently generate and deploy each candidate schedule onto the target GPU and obtained the profiled latency during multiple averaged runs. The averaged latency is then used as the cost of each candidate schedule.

Search Algorithm: With the input space and cost model defined, we could then use ML-based methods to search for the optimal schedule with the minimal latency.

In this work, we mainly implement two search algorithms, the random search and the coordinate descent search. The random search method samples scheduling solutions (different pointer matrices) randomly from the search space and profiles their latency as the cost. A memory module will record all schedules and costs, and after certain rounds of search, the algorithm will return the schedule with the lowest latency. As we will show later, though the random search algorithm is simple, it could greatly reduce the multi-tenant runtime latency by large margins, highlighting the advantages of our problem abstraction and the search framework design.

Based on a similar process, the coordinate descent search algorithm improves the sampling efficiency by adopting a coordinate-alternated search philosophy. The overview of the coordinate descent search algorithm is shown in Algorithm 1. It treats different streams' pointer index vectors (rows in the pointer matrix) as different coordinates. Then it alternatively finds the optimal pointer index vector for each coordinate, during when other

coordinates' solution are kept as the previous optimal one. The optimal pointer index vectors for all streams are updated for each round, and after certain rounds, the algorithm returns the optimal schedule from all previously searched schedules. Generally, the coordinate descent search algorithm could yield slightly performance than random search. Both methods yield near-optimal schedule within short time as we will show later.

4.1.4 Experimental Evaluation

Experiment Setup: We construct various multi-tenant scenarios by leveraging the following neural network models: AlexNet ($Alex$), VGG16 (VGG), ResNet18 ($R18$), ResNet34 ($R34$), ResNet50 ($R50$) and ResNet101 ($R101$). These models have distinctive model depths with operator numbers varying from $7 \sim 20$ to $86 \sim 216$. In addition, operators from different models also have particular computing and memory requirements. For example, the convolution operators have a wide range of computing complexity, e.g., from $32 \sim 64$ filters per layer to $256 \sim 512$ filters per layer. Therefore, each different multi-tenant combination based on the above models will pose its unique resource utilization imbalance challenges and has distinctive optimal scheduling strategies, mimicking the varied and complex multi-tenant scenarios of real-world applications.

We mainly compare the acceleration ratio with three baselines: CuDNN-based sequential execution (CuDNN-Seq), TVM-based sequential execution (TVM-Seq), Stream-based parallel execution (Stream-parallel). Ours-R and Ours-G denotes the performance of our framework with random and coordinate descent search. Our test platform is Titan-V GPU.

Overall Speed-up: It can be observed that our scheduling framework could consistently yield $1.3\times \sim 1.6\times$ speed-up compared to the sequential baselines across all five model combinations. Although the Stream-Parallel solution also yields a certain speed-up than CuDNN-Seq, its acceleration ratio is only $1.1\times \sim 1.3\times$, which is much less than ours.

Higher Speed-up in Highly Non-balanced Scenarios: It is worth noting that our

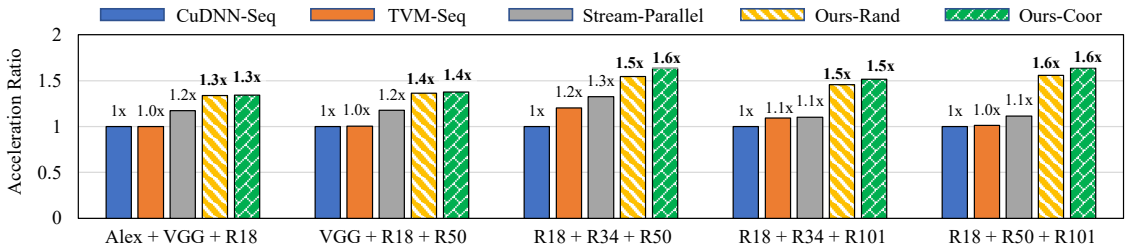


Figure 4.4: Runtime performance of the proposed framework.

Table 4.1: Our Framework Overhead (Titan-V).

#Search Rounds	100	300	500	1000
Alex + VGG + R18	∼9.8s	∼28.9s	∼51.4s	∼1min35s
VGG + R18 + R50	∼10.3s	∼27.1s	∼48.9s	∼1min28s
R18 + R50 + R101	∼16.2s	∼45.3s	∼1min32s	∼2min42s

method achieves the highest acceleration ratio, i.e., 1.5× and 1.6×, on the two most challenging scenarios $R18 + R34 + R101$ and $R18 + R50 + R101$. However, the Stream-Parallel performs poorly (only 1.1×) in these two settings. The reason is that such two multi-tenant combinations introduce extremely distinctive model lengths from 29 operators (ResNet18) to 200 operators (ResNet101), which brings significant resource imbalance between early and later stages across the entire processing. The native hardware scheduler in Stream-Parallel cannot take this into consideration and push all operators into the beginning stages, and thus cannot balance the resource utilization effectively.

Framework Overhead Analysis: Our framework could usually yield near optimal schedule solutions within short search time. The framework's running time is demonstrated in Table 4.2. We profile the coordinate descent search with different search rounds from 100 to 1000, which are general settings for most aforementioned multi-tenant scenarios. As the results show, our framework's running overhead maintains in the range of ten seconds to

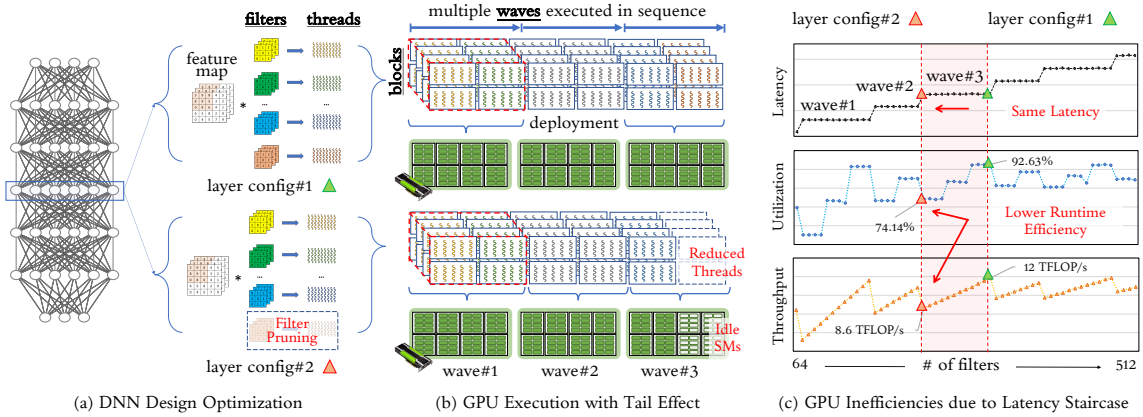


Figure 4.5: An example of ineffective latency optimization by filter pruning.

several minutes at most. Meanwhile, as such automated schedule can be pre-conducted offline given a defined multi-tenant scenario, we consider such offline tuning overhead is highly acceptable.

4.2 TA-DNN: Tail-Aware DNN Design on GPU

Deep Neural Networks (DNNs) have achieved remarkable progress on cognitive applications [37, 68, 69] and have drawn attentions from both industry and academia on deploying DNN inference service on the cloud, edge and mobile devices [86, 87]. However, the superior performance of DNNs is largely built upon growing model volume and structure complexity. Consequently, how to run these bulky DNNs on accelerators efficiently becomes the key to meet the ever-growing user experience, *e.g.*, DNN model inference latency.

To relieve the computation cost and improve runtime performance, many research efforts have been made in different design perspective and implementation levels. These works include model-level optimization [54, 88–90], compiler-level computing optimization [13, 84, 91], and architecture-level optimizations [29, 30, 92]. Among these works, model-level design solutions are usually the most flexible and straightforward way, since such methods can reduce

model volume and computation workload like FLOPs, which can directly translate to latency reduction. However, recent hardware-aware works questioned the effectiveness of these optimization works by revealing a widespread concept, *i.e.*, the lower amount of FLOPS does not equal to small latency during actual deployment on certain hardwares [19, 90], which underlines the importance of understanding of the hardware execution mechanisms for DNN design optimizations.

In this work, we re-examine the effectiveness of model design works for latency optimization on the general processing units (GPUs). By delving into the DNN to GPU deployment and execution mechanisms, we reveal a series of findings that current model design fails to recognize and thus lead to sub-optimal latency optimization performance.

Figure 4.5 illustrates such an ineffective latency optimization case with a structural filter pruning example. Specifically, two layer configurations (config#1, #2) are compared, in which the config#2 conducts filter pruning and reduces partial workloads of this layer. Surprisingly, although config#2 reduces the filters, it ends up having the exact same runtime latency as config#1. In fact, as we keep pruning filters, Figure 1 (c) shows that the execution time exhibits an interesting non-linear *latency staircase* phenomenon. Why does filter pruning not always lead to latency reduction? In this paper, We identify the root cause of this phenomenon as the lack of awareness of the GPU *tail effect* — a classic parallel system issue that causes the resource under-utilization in the last processing cycle [93].

In particular, during execution on GPUs, a DNN layer's computation workload will be translated into massive parallel threads and allocated to GPU for processing on CUDA cores (Figure 4.5 (a)). However, due to the resource limit (*e.g.*, number of cores, maximal concurrency), the excessive parallel threads would be partitioned into multiple *waves* (Figure 4.5 (b)). Threads in each wave run concurrently while these multiple waves execute sequentially. Due to such an execution mechanism, although layer config#2 reduces the number of threads in the last wave, it takes the same runtime latency as config#1. Figure 4.5 (c) verifies the above execution analysis by showing a *latency staircase* pattern. Comparing layer config#2 to config#1, pruning certain amounts of filters brings no latency

gain, but rather significantly lower down the GPU utilization and throughput due to the last partial wave.

Motivated by such an observation, **we conduct comprehensive benchmarking analysis and reveal the systematic mechanism** that causes such inefficiency in DNN inference: ***(i)*** Due to massive CUDA core parallelism, GPU presents a large *hardware-level compute granularity*. This is one major factor that causes the non-linear DNN workload to GPU runtime latency translation, and lead to the *latency staircase* phenomenon; ***(ii)*** Most current works are non-aware of such a GPU compute granularity and their DNN optimization can easily lead to the runtime *tail effect*; ***(iii)*** Furthermore, due to common DNN's deep layer structure, such *tail effect accumulates and amplifies with hundreds of layers*, resulting in significant overall resource under-utilization.

Based on such tail effect analysis, **we rethink the current DNN design optimization and propose following insights**: ***(i)*** For DNN structure design, the observation of latency staircase implies *DNN architecture design should take GPU compute granularity into consideration*, which would help enable more efficient hardware utilization for model optimization, *e.g.*, help defining the search space in NAS solutions; ***(ii)*** For DNN structure optimization, the tail effect implies that there are *free runtime resources for potential accuracy boost*, *e.g.*, by fulfilling each layer's tail wave of workload to utilize idle SMs but without increasing the latency; ***(iii)*** For complex DNNs with hundreds of layers, accumulated tail effects within these layers implies great chances for *non-trivial latency reduction* while maintaining accuracy.

Performing the design insights above could guide both DNN pre-design and DNN post-optimizations. We demonstrate our optimization approaches regarding two common design-level solutions, neural architecture search and structural pruning: ***(i)*** For NAS solutions, we enhance the current design space with expected GPU granularity awareness. Such a GPU granularity-aware configuration discretizes previous continuous layer width design space, thus greatly reducing the search candidates and the searching complexity. Meanwhile, as the design space already avoids the tail effect, the DNN structure could

also achieve better GPU runtime efficiency and thus better latency performance. ***(ii)*** For structure post-optimization like pruning, we can also leverage the GPU-aware layer design space to fine-tune the sub-optimal pruned models. Specifically, we adopt layer growing and layer pruning two operations to utilize the free resources and to eliminate the GPU tail effect. Featured with such flexibility, our structural post-optimization can achieve both accuracy lifting and latency reduction, which is one major difference from previous works.

4.2.1 Revealing the GPU Computing Granularity for Runtime Latency

In this work, we first dive into the DNN latency analysis to reveal the underlying GPU computing mechanisms. A particular DNN latency issue is identified through detailed observation as the *latency staircase* phenomenon, which triggers our rethinking of the latency-aware DNN model design.

DNN Latency Staircase Phenomenon

We profile the inference latency of two representative DNN model settings on an NVIDIA Titan-V GPU: (a) VGG16 [7] on CIFAR10 dataset; (b) ResNet50 [12] on ImageNet dataset. For each DNN model, we investigate the latency impact from different model volumes by gradually pruning the layer width step by step and measure the runtime latency in each layer width setting. All experiments are conducted using PyTorch 1.5 with CUDA 10.2 and CuDNN 7.6.5 backend.

Through the analysis, we identify a distinct DNN runtime issue, *i.e.*, the ***latency staircase*** phenomenon. Specifically, Figure 4.6 illustrates the latency of five randomly selected DNN layers (denoted by conv-N) of VGG16 and ResNet50, respectively. For each layer's dataset, although the layer volume is gradually shrunk by filter pruning, runtime latency doesn't reduce linearly but demonstrates a "staircase" trend, are changes occurring only after a certain plateau period.

Such results reveal several findings in DNN latency optimization: ***(i)*** *the layer volume and workload reduction by conventional structural pruning cannot directly translate to*

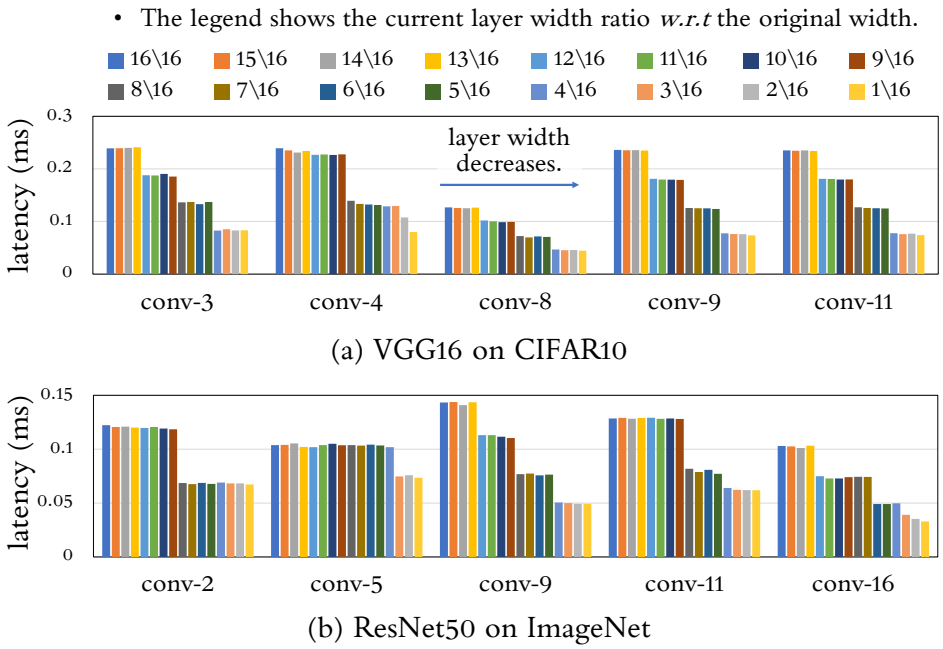


Figure 4.6: DNN runtime latency profiling.

GPUs' runtime latency optimization; ***(ii)*** *such an issue is prevalent across DNN models and layer configurations;* ***(iii)*** *DNN latency optimization requires a deeper investigation of the hardware computing mechanism than a pure software perspective.*

A Reflection of GPU Computing Granularity

Through dedicated analysis, we find that the above latency staircase is a reflection of specific GPU computing granularity due to the huge-parallelism execution mechanism. Specifically, we dive into fundamental GPU computing primitives (*e.g.*, number of GPU thread blocks B and waves W) using NVIDIA Nsight Compute [94], and reveal the DNN deployment and GPU computing mechanism.

Latency Staircase Analysis Figure 4.7 shows the GPU computing changes when we vary a layer's width from 64 to 512 filters [3]. As we can see, the number of thread blocks B

[3]Specifically, the filter shape and input feature map sizes are set to 3×3 and 64×64×512. And we keep the batch size to 1 throughout the experiment.

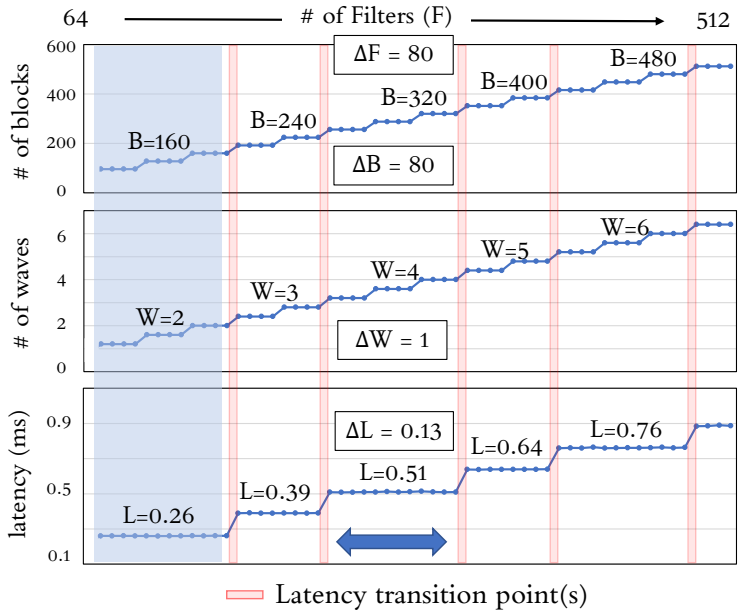


Figure 4.7: GPU computing analysis *w.r.t.* blocks, waves, and latency.

grows with increasing number of filters F (top figure), which is expected. Correspondingly, the number of waves also increase. However, due to the granularity mismatch (*i.e.*, one filter's workload doesn't equal to one block, and not equal to one wave.), the number of wave increase in a discrete way (step-by-step), forming the staircase growing trend and with a larger granularity (middle and bottom figure).

As highlighted by the blue box in Figure 4.7, the number of waves W grows gradually from 1 to 2 with increasing workload. However, the runtime latency however stays still until the workload grows to above 2 waves, and then GPU will consume a new processing cycle to process the new wave of workload. We thus see a jump in the runtime latency as highlighted in the red box, *i.e.*, the latency transition point.

4.2.2 Latency Formulation with GPU Granularity

Similar staircase patterns continuously emerge with growing workloads, which reflects a GPU-specific compute granularity. We can thus model the latency staircase based on the

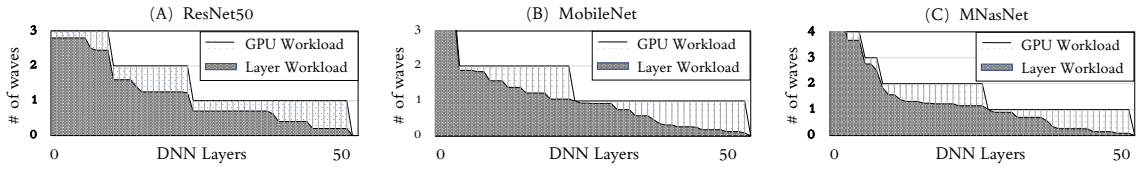


Figure 4.8: The GPU under-utilization caused by the tail effect.

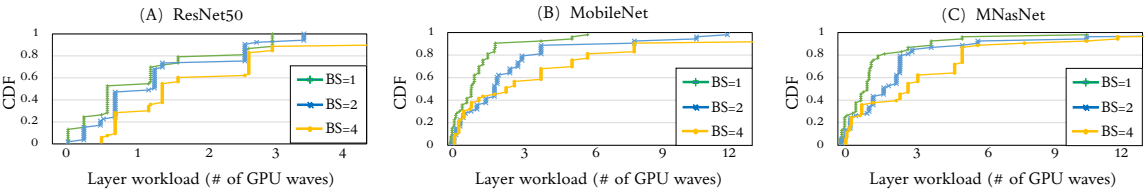


Figure 4.9: The CDF of the layer workload (# of waves).

above analysis. For a given DNN layer, its GPU runtime latency L can be represented by:

$$L = \Delta L \times \lceil \ \#W \ \rceil, \qquad \text{where } \#W = \frac{B}{S}, \tag{4.9}$$

where ΔL represents the latency for the GPU to finish one wave of workload, $\#W$ is the number of waves and B is the number of blocks that the overall workload being translated to. S is processing capacity of the GPU, *e.g.*, the maximal number of blocks that can be concurrently executed. $\lceil \cdot \rceil$ is the rounding-up operation that returns the least integer greater than or equal to the input (*e.g.*, $\lceil 2.1 \rceil = 3$), which reveals the impact of ***tail effect*** on the runtime latency, that is, one underutilized tail wave still leads to the latency of one full computing cycle for each DNN layer.

Eq. 4.9 demonstrates that the DNN execution latency on GPUs is highly dependent on a GPU-specific compute granularity, i.e., the wave W. Such a granularity is not only a hardware-level concept to describe the GPU capacity, but also an important factor for latency-aware DNN design. Without being aware of this, DNN structure design can easily generate granularity-unmatched models, thus resulting in sub-optimal GPU efficiency and

DNN inference latency.

4.2.3 Latency-Aware DNN Design with GPU Granularity Awareness

In this section, we summarize the lessons learned from our analysis and characterization to guide DNN design on GPUs. We then demonstrate that by eliminating the GPU tail effect, we can enhance DNN optimizations with better GPU awareness and better performance.

DNN Design Insights for Latency-Awareness

By understanding the GPU granularity-based latency modeling and the inefficiency caused by the tail effect, we can illustrate a series of DNN design insights:

• *Latency-aware DNN optimization on GPUs should consider a new basic unit, i.e., the GPU computing granularity, instead of individual filters.* Previously, many DNN design and latency optimization works consider filters as the primary structural unit. However, our latency staircase analysis shows that, as one filter's workload is usually much smaller than the GPU computing granularity, workload measured by filters fails to translate to the latency consistently. By understanding this, latency-oriented DNN design could improve both their latency modeling effectiveness and design efficiency. For example, given a targeted GPU, each DNN layer's design only needs to consider a discrete design space, *i.e.*, with a step size of a certain number of filters that matches the GPU computing granularity.

• *For existing DNN structures, the existence of tail effects is prevailing, which exposes many opportunities for both accuracy and latency performance improvements.* The reason of prevalence is that, as the GPU platform (and other factors like compilers) can vary, the unified DNN structure design can hardly avoid tail effects in all different settings. In such a case, tail effect accumulation in DNNs can incur huge inefficiency, meanwhile exposing many optimizations potentials. Specifically, the GPU resource under-utilization enables us to increase DNN workload and utilize the "free" tail resources[4] to boost accuracy while without affecting the runtime latency. In addition, as tail effect can emerge in many layers,

[4]Here we mainly denote latency-wise "free". The power consumption may still show differences, which is out of scope of this work.

by simultaneously pruning certain tail waves and filling up other ones, we can also obtain certain latency gains while maintain the overall DNN capacity and accuracy.

The above design insights can guide various DNN design optimization approaches. In this work, we mainly include approaches: GPU-aware NAS design space enhancement and lightweight DNN structure fine-tuning.

GPU-aware NAS Design Space Enhancement

The first optimization we can apply is to enhance the design space of NAS methods with the proposed GPU awareness.

Previous NAS works mainly heuristically choose layer width in the design space, which may have sub-optimal GPU runtime performance. Therefore, one direct way of utilizing our GPU granularity-aware modeling is to find the optimal layer configurations and integrate into the NAS design space. The benefits are twofold: ***(i)*** our granularity-based modeling discretizes a continuous layer width design space and reduces the amount of search candidates for each layer, reducing the searching complexity; ***(ii)*** meanwhile, as each configuration already avoids the tail effect, our design space brings better GPU runtime performance without the tail effect related inefficiency.

Profiling-Guided Identification We take a profiling strategy, *i.e.*, profiling the GPU latency staircase offline to identify the optimal design space for each layer.

Based on the layer runtime modeling in Eq. 4.9, the tail effect diminishes when DNN layer workload matches the GPU compute granularity, *i.e.*, the last wave of workload fully occupies the GPU. In such case, the GPU achieves optimal utilization, and the throughput also reaches local maximum in the range of current wave. Therefore, we leverage the GPU throughput T to identify the existence of tail effect in each layer and further detect its optimal layer configurations. Taking a retrospect of Figure 4.5 (c), we can see the throughput always peaks its local maximal (denoted by the green triangle) at the optimal layer size setting without tail effect and thus can serve as a reliable clue to track the optimal

configurations. Specifically, we obtain the optimal configuration $C_i^\star$ for each layer i based on the following rule:

$$C_i^\star = \arg\max_m(T_{i,m}) \tag{4.10}$$

where $T_{i,m}$ denotes GPU throughput of layer i in the m^{th} layer width configuration. We then use Eq. 4.10 to identify optimal configurations in different layer width intervals.

Overhead Analysis Table 4.2 shows the offline profiling overhead on different DNN models. For each layer, we sample 16 different layer width configurations (from 1/16 to 16/16 with respect to the default layer size) and record their throughput. We observe the overall profiling overhead stays in the range of one to three minutes, which shows the efficiency of our design space identification method.

Lightweight DNN Structure Fine-Tuning

Besides the end-to-end DNN design apporach of NAS, we can also conduct lightweight DNN structure fine-tuning to obtain the optimal pruned models in structural pruning methods.

Structure pruning without GPU awareness can generate pruned models with excessive tail effects. Therefore, we can leverage ***pruning* and *grafting*** two operations to eliminate the tail effect. An overview of our method is shown in Figure 4.10. Specifically, *layer pruning* reduces the number of filters to remove the remaining workload on the last wave

Table 4.2: Offline Profiling Overhead.

	Model	BS=32	BS=64	BS=128
CIFAR	VGG16	2m22s	2m23s	2m25s
(32x32x3)	ResNet56	1m27s	1m30s	1m40s
	Model	BS=1	BS=2	BS=4
ImageNet	VGG16	2m14s	2m15s	2m18s
(224x224x3)	ResNet50	2m50s	2m51s	2m56s

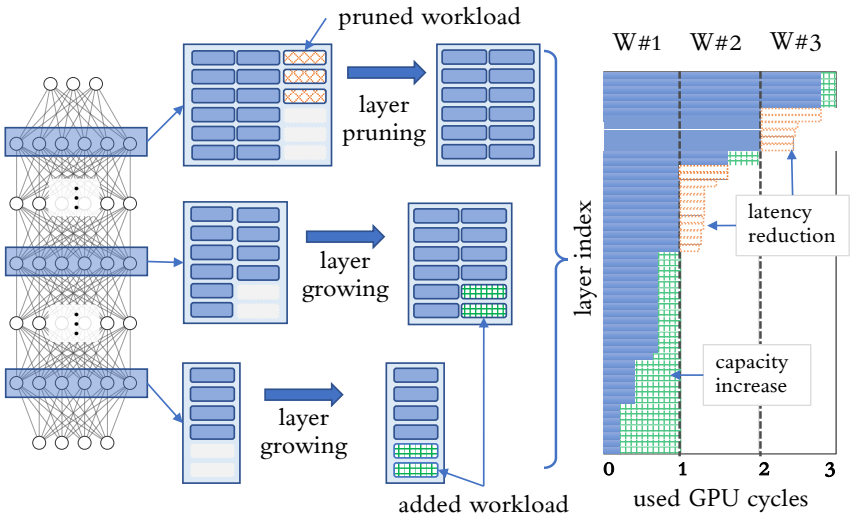


Figure 4.10: We leverage two operations (pruning and grafting) to eliminate the tail effect.

and thus can effectively harvest latency gain by an entire GPU wave processing cycle. *Layer grafting* instead increases the number of filters in other laters to fill up the GPU capacity in the last wave, which can compensate the accuracy loss of layer pruning without sacrificing the latency (*i.e.*, accuracy gain). These two operations provide us flexible opportunities to match the GPU granularity and thus eliminate the tail effect. Meanwhile, they also provides DNN optimization flexibility towards two optimization targets: latency reduction and accuracy lifting, which is one major difference from traditional pruning works.

Here one challenge remains, *i.e.*, how to select two operations between layer pruning and grafting across multiple layers regarding different optimization objectives? We propose a lightweight fine-tuning algorithm to flexibly balance the corresponding operations to reach different objectives.

Optimization Objectives Formulation We first define the DNN inference accuracy and runtime latency optimization objectives. We define the latency gain LG to indicate the latency being saved, and parameter gain PG (amounts of parameters been deleted or added) to indicate the model capacity as a guidance for retaining accuracy. The two metrics

Algorithm 2 Lightweight DNN Structure Fine-tuning.

1: **Input:** Model's initial layer configs $r[l]$, latency $L[l]$ and throughput profiling $T[l]$.
2: **Output:** New config R_{new}.
3: Identify design space C_l for each layer l by Eq. 4.10.
4: **for** layers $i = 1$ **to** l **do**
5: Get LG_i, PG_i estimation through profiling results.
6: **end for**
7: Sort the layer index list [1,2...i...l] by LG_i or PG_i.
8: **while** layer index list is not empty **do**
9: Pop out layer j with argmax$_j$ $LG[l]$.
10: Prune layer j with one step in design space C_l.
11: **while** $\Sigma^l PG(R_{new}) \notin (-\tau, \tau)$ **do**
12: Pop out layer k with argmin$_k$ $LG[l]$.
13: Grow layer k with one step in design space C_l.
14: **end while**
15: **end while**
16: Get runtime latency evaluation L_{new} of config R_{new}.
17: **if** L_{new} achieves the target latency **then**
18: Train and evaluate the model accuracy.
19: **else**
20: Set τ *= 2 and repeat the algo. from line 9.
21: **end if**
22: **Return** Fine-tuned config R_{new}.

could be estimated by the change in different layer widths R_i and their latency profiling L_i. *Accuracy-Oriented Optimization* aims to improve the model accuracy without extra latency overhead. Therefore, it can be formulated as maximizing the parameter gain while constraining latency gain:

$$\text{Maximize} \quad \sum_i^l PG_i, \quad \text{s.t.} \quad \sum_i^l LG_i \geq 0. \tag{4.11}$$

Latency-Oriented Optimization aims to reduce the runtime latency without accuracy drop, *i.e.*, maximizing latency gain while maintaining parameter gain in a tolerable range (τ):

$$\text{Maximize} \quad \sum_i^l LG_i, \quad \text{s.t.} \quad \sum_i^l PG_i \in (-\tau, \tau). \tag{4.12}$$

Algorithm Walk-through Without loss of generality, we take the latency-oriented optimization as an illustrate example to explain the algorithm. The accuracy-oriented optimization follows the same principle by simply switching the optimization objectives as defined in Eq. 4.11–4.12.

Step 1. Layer Candidates Identification (Line 3): Given a DNN model structure, we first identify the optimal layer configurations, *i.e.*, design space C_i by Eq. 4.10.

Step 2. Intra-Layer Performance Gain Estimation (Line 4–6): For each layer, we get the latency and parameter gain estimation assuming conducting layer pruning and layer growing operations by one step in the design space C_i. These performance estimation will guide the inter-layer adjustment.

Step 3. Inter-Layer Adjustment Strategy (Line 7–13): To achieve latency optimization, we maintains a bi-directional queue of layer indexes, and greedily prune layers with maximal LG_j (max latency gain) and minimal LG_k for growing (balanced capacity) . Same procedure applies to accuracy optimization by switching the objective/constrain in Line-9, 11.

Step 4. Model Structure Determination (Line 14–18): Finally, we return the new configuration if it reaches the targeted latency. Otherwise, we can adjust the constraints, *e.g.*, with larger parameter gain tolerance ($\tau* = 2$), to allow more aggressive layer pruning for latency reduction.

4.2.4 Experimental Evaluation

We conduct experiments using the common software stack including PyTorch 1.5, CUDA 10.2 and CuDNN 7.6.5. For GPU hardwares, we select three GPUs from high-end (Titan-V, P6000) to embedded ones (Jetson Nano). The specification of them is shown in Table 4.3.

We apply our method into two common model optimization approaches, *i.e.*, *structural filter pruning* and *NAS*. For pruning, two state-of-the-art pruning methods including HRank [95] and SoftPruning [96] are chosen as baselines. For NAS, we apply our model optimization method onto the EfficientNet series of structures [21], which are one type of the SOTA efficient model architectures. For model performance evaluation, we compare the model's accuracy and runtime latency. Without specific mentioning, the batch sizes for latency evaluation on CIFAR10 and ImageNet are set to 128 and 1 to observe the latency variations[5]. Meanwhile, we also collect the GPU throughput (FLOP/s) information as a

[5]As CIFAR has a small resolution (32x32x3), we set batch size=128 in CIFAR to maintain a roughly

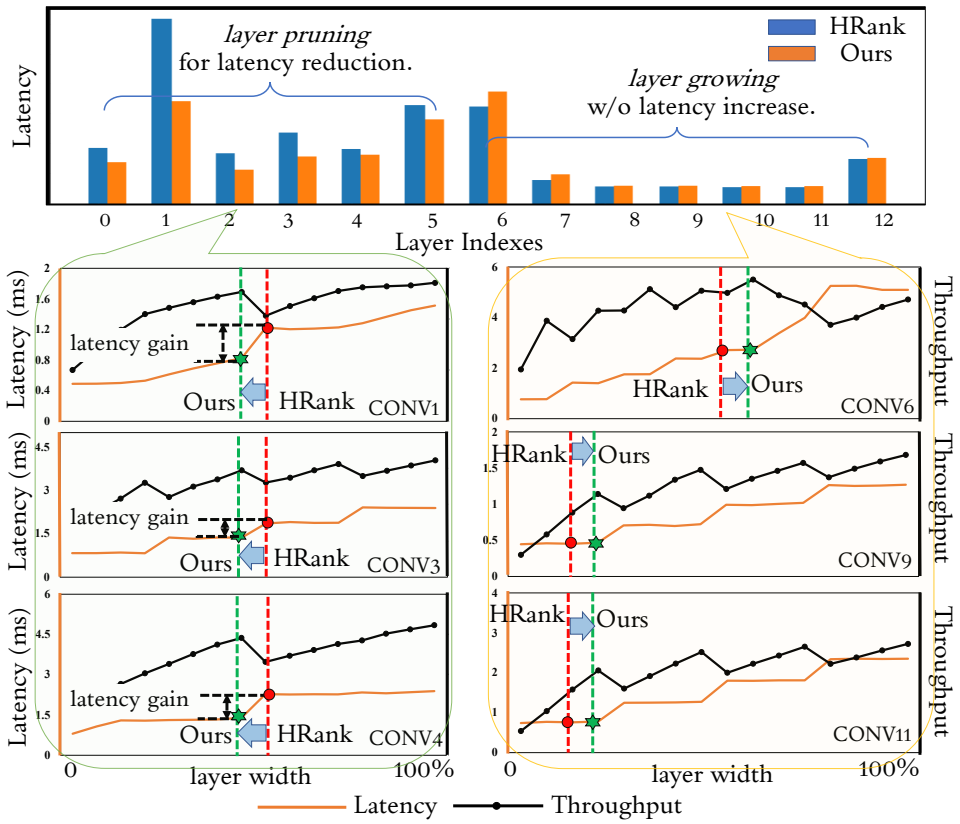


Figure 4.11: Our DNN optimization illustration.

indicator to demonstrate the GPU runtime efficiency comparison.

Evaluation on the Structural Pruning Approach

A Proof-of-Concept Case Study As a proof-of-concept, we first apply our optimization to a prevalent DNN benchmark model VGG16 and compare the results with one SOTA pruning competitor HRank [95] to illustrate our algorithm mechanism. The result is shown in Figure 4.11.

Our results (orange bar) consistently outperform HRank (blue bar) on run-time latency reduction, as shown in Layer 0∼5. We further showcase the latency and throughput of three randomly selected layers for both layer pruning and growing operations (bottom figure). As

similar DNN computation workload as batch size=1 in ImageNet (224x224x3) experiments.

Table 4.3: The Evaluated GPUs and Specifications.

GPU Name	Archicture	#SMs	#Cores	Peak FLOP/s
Titan-V	Volta	80	5120	14.9T
Quadro P6000	Pascal	30	3840	12.0T
Jetson Nano	Maxwell	1	128	0.24T

shown in CONV-1,3,4, we successfully remove one tail latency by conducting layer pruning operation. At the same time, the GPU throughput achieves its local maximal. The layer growing actions, on the other hand, brings little to no latency overhead, as shown in CONV-6,9,11. Taking a further scrutiny on their latency results, we can see that the layer width is increased by a proper amount such that the layer's runtime latency stays similar with HRank. Overall, we can achieve **17.7%** latency reduction compared to HRank with only **0.2%** accuracy drop (93.1%→92.9%), as shown in the first two rows in Table 4.4.

General Benchmarks We then apply our model optimization method in more pruning benchmarks. The result comparisons are shown in Table 4.4. Note that, SOFT method [96] has two pruning configurations (denoted by SOFT-1 and SOFT-2) with varied pruning degrees, which we compare separately. The latency evaluation is conducted on Titan-V.

Latency Reduction: As Table 4.4 presents, by optimizing the model structure configurations, our method achieves consistently lower latency than baseline methods (**11.3%-17.7%** latency reduction in all settings). Different from previous work that focuses on reducing workload to reduce latency, our method's latency reduction mainly comes from the higher GPU runtime efficiency, *e.g.*, GPU throughput improvement.

GPU Throughput Improvement: As Table 4.4 shows, our method shows consistently higher GPU throughput than the baselines, *i.e.*, in the (*FLOP/s*) column. For VGG16, our optimized model achieves 3.90 TFLOP/s, **1.6**× throughput than the baseline method, 2.41 TFLOP/s. For ResNet56, our models also achieve **1.2**× throughput than HRank (1.00

Table 4.4: Latency Optimization on SOTA Pruning Works.

	Method	Params	#FLOPs	FLOP/s	Acc.%	Time (ns)
VGG16 (CIFAR10)	HRank [95]	1.90M	67.0M	2.41T	**93.1**	2.50E6
	Ours	2.85M	104.1M	**3.90T**	92.9	**2.05E6 (-17.7%)**
ResNet56 (CIFAR10)	HRank [95]	0.48M	65.9M	0.83T	93.6	4.20E6
	Ours-1	0.50M	79.1M	**1.00T**	**93.8**	**3.72E6 (-11.3%)**
	Ours-2	0.50M	75.1M	**0.95T**	93.5	**3.51E6 (-16.3%)**
ResNet56 (CIFAR10)	SOFT-1 [96]	0.53M	68.8M	0.88T	93.1	4.08E6
	Ours-1	0.43M	71.4M	**0.91T**	**93.2**	**3.52E6 (-13.7%)**
	SOFT-2 [96]	0.45M	53.1M	0.68T	92.3	3.64E6
	Ours-2	0.43M	66.0M	**0.79T**	**92.3**	**3.01E6 (-17.3%)**

vs. 0.83 TFLOP/s) and SOFT-2 (0.79 vs. 0.68 TFLOP/s). Such throughput enhancement indicates the GPU runtime efficiency improvement, and shows the advantages of our configuration optimization by tail effect elimination.

Accuracy Maintenance: In addition to lower latency, our method also maintains the model accuracy change within negligible ranges (±0.2%). This is benefited by constraining the parameter gain (PG) in the algorithm. As (*Params*) column in Table 4.4 shows, our optimized models have maintained the amount of parameters either in positive range or very small negative range. By doing so, we ensure the models' capacity to be well-maintained, thus keeping the model accuracy during the optimization process.

Evaluation on the NAS Approach

In this part, we apply our optimization method to further optimize the NAS network's performance on GPUs. Specifically, we optimize the EfficientNet [21] series of model structures to achieve better accuracy latency trade-offs. The evaluation results are shown in Table 4.5. The latency evaluation is conducted on both Titan-V and P6000 GPUs with batch size = 1 to simulate the real-time performance.

Table 4.5: Accuracy Optimization on EfficientNets.

	Method	Acc.%	GPU:Titan-V		GPU:P6000	
			Time(ms)	FLOP/s	Time(ms)	FLOP/s
EfficientNet	B0	77.52	12.6	61.9G	13.8	56.5G
(ImageNet)	Ours	**81.49 (+3.97)**	12.7	**414.2G**	14.1	**373.0G**
EfficientNet	B1	79.38	17.8	78.7G	19.8	70.7G
(ImageNet)	Ours	**82.08 (+2.7)**	18.0	**442.7G**	19.9	**396.0G**
EfficientNet	B2	80.18	18.2	109.9G	19.9	100.5G
(ImageNet)	Ours	**82.72 (+2.54)**	18.4	**547.3G**	20.4	**488.2G**

Accuracy Maximization: As Table 4.5 shows, with the similar runtime latency, our optimized EfficientNet model could achieve much better accuracy (**+3.97%**, **+2.7%**, **+2.54%**, respectively) on the challenging ImageNet dataset.

Better Latency Accuracy Trade-offs: We composedly compare the model accuracy-latency performance with original EfficientNets [21], and several major types of DNN structures including ResNets [12], ResNeXts [97], InceptionNet [98], and Dense Net [99]. The results are shown in Fig. 4.12. From the *accuracy* perspective, our optimized B0 to B3 models achieve the highest accuracies than most baselines under the same latency. From the *latency* perspective, our model B2's latency is **1.5**× less than the models which achieve the same accuracy level, *e.g.*, EfficientNet-B3 and B4. Therefore, both perspectives demonstrate the effectiveness of our method in achieving better accuracy latency trade-offs on GPUs.

Optimization Explanation: Here we explain our optimization mechanisms for the EfficientNet series of networks. When conducting EfficientNet inference on GPUs, we observe that most of layers in these models consume less than one GPU wave of workload, *i.e.*, highly under-utilizing the GPUs. The GPU throughput is only 61.9 - 109.9 GFLOP/s for B0 to B2 as shown in Table 4.5. Such under-utilized GPU capacity allows us to conduct layer growing to reach one full wave for each layer without latency increment, thus bringing higher model

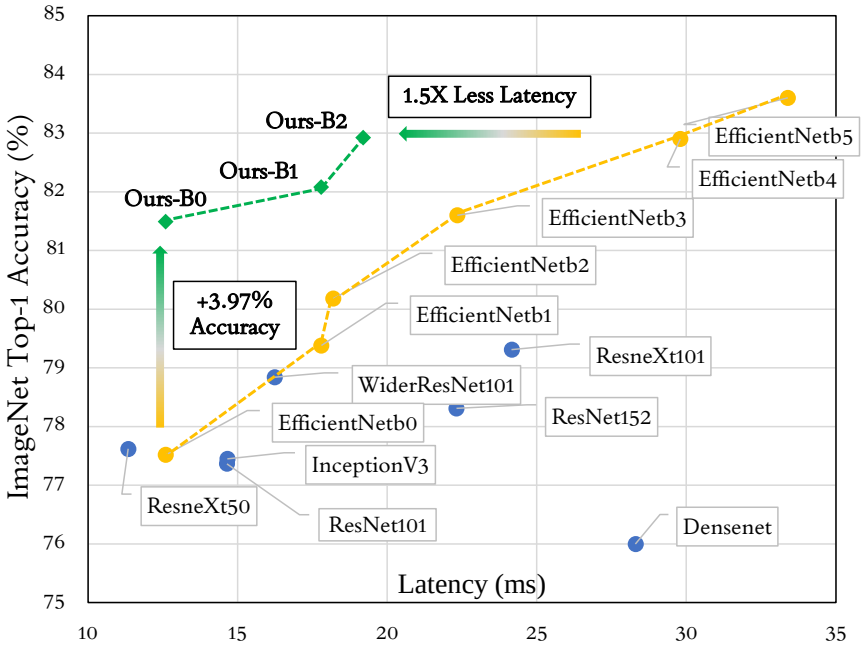


Figure 4.12: ImageNet accuracy and latency compared to EfficientNets.

capacity and model accuracy.

For implementation, EfficientNet series of networks have three scaling dimensions in their initial design: layer width (w), input resolution (r) and network depth (d) [21]. Out of the three, we balanced scale up both (w) and (r) to increase the network capacity. The depth (d) dimension, however, is kept same with the baseline model, as network layers are run sequentially and depth scaling would inevitably incur latency increase. By width and resolution growing, the optimized models have higher accuracy as well as improved GPU throughput, *e.g.*, 414.2 - 547.3 GFLOP/s, while maintaining the similar latency with baselines, as shown in Table 4.5.

Generalizability across GPUs

We further extend the generalizability evaluation to different high-end GPU, P6000 and embedded GPU, Jetson Nano. The detailed GPU specifications can be found in Table 4.3.

Table 4.6: Generalizability Evaluation on Pascal P6000 GPU.

	Method	Params	FLOP/s	Acc.%	Time (ns)
VGG16	HRank	1.90M	1.68T	**93.1**	4.15E6
(CIFAR10)	Ours	2.04M	**1.86T**	92.9	**3.02E6 (-27.2%)**
ResNet56	HRank	0.48M	0.84T	93.6	5.84E6
(CIFAR10)	Ours	0.50M	**1.00T**	**93.7**	**5.32E6 (-9.0%)**

Table 4.7: Generalizability Evaluation on Jetson Nano GPU.

	Method	Params	FLOP/s	Acc.%	Time (ms)
VGG16	HRank	1.90M	35.5G	**93.1**	60.5
(CIFAR10)	Ours	2.04M	**39.4G**	92.9	**48.1 (-20.5%)**
	HRank-1	0.48M	25.6G	93.6	82.1
ResNet56	Ours	0.50M	**28.8G**	**93.7**	**68.0 (-17.1%)**
(CIFAR10)	HRank-2	0.24M	16.7G	92.3	67.2
	Ours	0.39M	**24.3G**	**92.5**	**58.1 (-13.3%)**

The overall results are shown in Table 4.6 and Table 4.7. On the P6000 GPU (Table 4.6), our method could achieve **9.0%** to **27.2%** latency reduction than SOTA methods while maintaining similar accuracy for VGG16 and ResNet56, demonstrating the generality across different high-end GPU architectures. For Jetson Nano GPU (Table 4.7), although it has relatively smaller computing capacity, our method still achieves **13.3%** to **20.5%** latency reduction, demonstrating the generalizability for both high-end and embedded GPU platforms. Without any GPU-specific assumptions, our profiling-guided design could be applied to a spectrum of GPUs to enhance the DNNs' accuracy-latency trade-offs.

Chapter 5: Conclusion and Future Work

5.1 Summary of Dissertation

With the strong scaling of DL model and compute capacity, the current era of deep learning shows great opportunities and challenges for more novel types of DL models and applications, and meanwhile incurs lots of challenges to performance optimization for large-scale high-performance deep learning computing.

In this dissertation, we start with deep learning algorithm level computing optimization, i.e., convolutional neural network (CNN) model compression and acceleration, and then delve into the software and hardware co-optimization to achieve ult-most computing efficiency and performance. Specifically, we introduce two different algorithm-level compression works. Different from traditional works that focus on filter or weight pruning, our work targets at the other two novel components of DNNs: feature maps and connectivities. We thus propose an attention-based dynamic feature map pruning framework, Antidote; and the structural decoupling framework by conenctivity pruning, DC-CNN. On the hardware side, we also propose two software-hardware co-optimization works. Specifically, the first work targets at the graph and runtime co-optimization that conducts resource-aware multi-tenant runtime scheduling, MT-Graph; and the second work targets at GPU-aware DNN design by tail effect analysis and elimination, TA-DNN.

5.2 Vision and Insights

Besides the optimization works introduced in the dissertation, there are also some important trends in the current deep learning era.

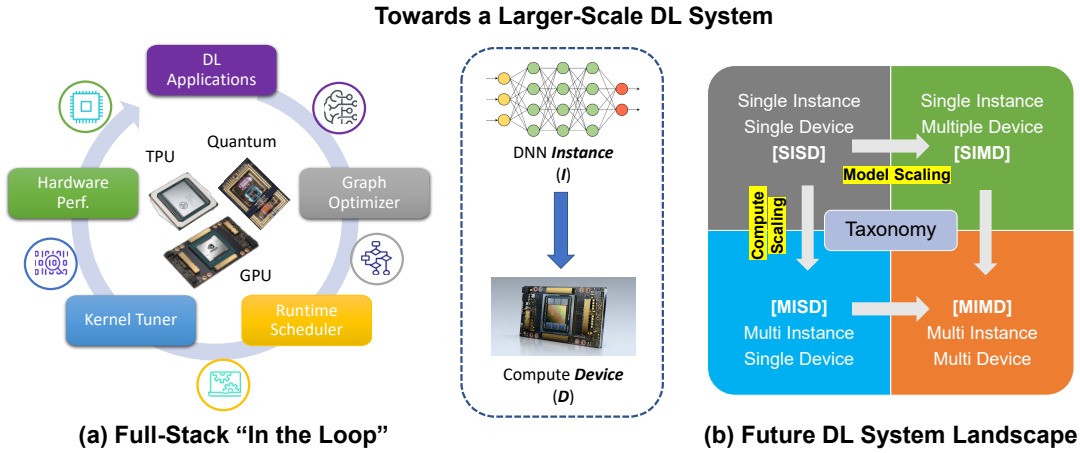


Figure 5.1: The trends towards the larger scale DL system.

5.2.1 Architecture Design with Full Stack in the Loop

Specifically, one important future trend is the *"full stack in the loop"*, i.e., to remove the vertical boundaries in the modern DL system stack and conduct full-stack integration to strive for both optimal performance and flexibility. One example of compiler-oriented efforts is the tvm unity [100] as shown in Figure 5.1 (a). As current system stack conducts separate layer-wise optimization (graph-runtime-kernel-resource) and single-directional deployment, it usually prohibits necessary cross-layer interactions and feedback between different levels. In such cases, unifying the abstraction between layers and automation would greatly facilitate the new full-stack optimization as a loop, not only for multi-tenant computing, but also for future wider DL application.

5.2.2 A Taxonomy of Future Large-Scale DL Computing

Our disseration has introduced one multi-tenant DL computing scenario. In fact, multi-tenant is a natural generalization result due to the significant *computing scaling* trend of GPU and other types of accelerators. However, if we take into the recent *model scaling* trend into consideration, a new DL & system interaction mode could be observed, that is

to conduct multi-device co-computing for a single model. For example, the recent SOTA giant AI model Megatron-NLG [8] has reached 500 billions of parameters and requires tens of GPUs to conduct multi-node distributed inference.

We predict the future large-scale DL system landscape by using a taxonomy shown in Figure 5.1 (b). Using Instance (I) to denote one DNN model and Device (D) to denote the compute hardware, traditional DL system mostly comes within the *Single Instance Single Device (SISD)* domain and only constitute the top-left quarter of the full-spectrum. Multi-tenant computing emerges as the *Multiple Instances Single Device (MISD)* with the computing scaling trend, as we summzrized in this survery. Whereas diagonally, with the model scaling trend, the *Single Instance Multiple Devices (SIMD)* interaction mode also emerges for DL & System and are attracting more and more attention for large model distributed inference [101] such as language models, recommendation models, etc. Finally, *Multiple Instances Multiple Devices (MIMD)* would eventually combine all these modes together, which can be a practical case for future efficient DL-centric data centers.

Appendix 6: List of Publications

6.1 High-Peformance Deep Learning Computing

- [TCAD'22] **F. Yu**, Z. Xu, C. Liu, D. Stamoulis, D. Wang, Y. Wang, X. Chen. AntiDoteX: Attention-based Dynamic Optimization for Neural Network Runtime Efficiency. In the Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.

- [ICDCS'22] Ardestani, E.K., Kim, C., Lee, S.J., Pan, L., Rampersad, V., Axboe, J., Agrawal, B., Yu, **F., Yu**, A., Le, T. and Yuen, H., 2021. Supporting Massive DLRM Inference Through Software Defined Memory. In the Proceedings of International Conference on Distributed Computing Systems 2022.

- [ICCAD'21] **F. Yu**, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu and X. Chen. Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. in the Proceedings of the 40th IEEE International Conference on Computer Aided Design (ICCAD), Nov. 2021.

- [DATE'20, Best Paper Nomination] **F. Yu**, C. Liu, D. Wang, Y. Wang, and X. Chen. AntiDOte: Attention-based Dynamic Optimization for Neural Network Runtime Efficiency. in the Proceedings of the 23rd Design Automation and Test in Europe Conference, Mar. 2020.

- [DATE'20] **F. Yu**, Z. Qin, D.Wang, P. Xu, C. Liu, T. Zhi, and X. Chen. DC-CNN: Computational Flow Redefinition for Efficient CNN Inference through Model Structural Decoupling. in Proc. of the 23rd Design Automation and Test in Europe Conference, Mar. 2020

- [TCAD'20] Z. Qin, **F. Yu**, Z. Xu, C. Liu, X. Chen. CaptorX: A Class-Adaptive Convolutional Neural Network Reconfiguration Framework. in the Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.

- [TCAD'20] Z. Xu, **F. Yu**, Z. Qin, C. Liu, X. Chen. DiReCtX: Dynamic Resource-Aware CNN Reconfiguration Framework for Real-Time Mobile Applications. in the Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.

- [DAC'19] Z. Xu, **F. Yu**, C. Liu, and X. Chen. ReForm: Static and Dynamic Resource-Aware DNN Reconfiguration Framework for Mobile Devices. in the Proceedings of the 56th Design Automation Conference, Jun. 2019.

- [BMVC'19] Z. Qin, **F. Yu**, C. Liu, and X. Chen. Functionality-Oriented Convolutional Filter Pruning. in the Proceedings of the 30th British Machine Vision Conference, Sep. 2019.

- [ASPDAC'19] Z. Qin, **F. Yu**, C. Liu, and X. Chen. CAPTOR: A Class Adaptive Filter Pruning Framework for Convolutional Neural Networks in Mobile Applications. in the Proceedings of the 24th Asia and South Pacific Design Automation Conference, Jan. 2019.

- [ISLPED'18] Z. Xu, Z. Qin, **F. Yu**, C. Liu, and X. Chen. DiReCt: Resource-Aware Dynamic Model Reconfiguration for Convolutional Neural Network in Mobile Systems. in the Proceedings of the 23rd ACM/IEEE International Symposium on Low Power Electronics and Design, Jul. 2018.

- [ISVLSI'18] C. Liu, Q. Dong, F. Liu, **F. Yu**, and X. Chen. ReRise: An Adversarial Example Restoration System for Neuromorphic Computing Security. in the Proceedings of the 17th IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 470-475, Jul. 2018.

- [ISLPED'20] C. Liu, **F. Yu**, Z. Qin, and X. Chen. Enabling efficient ReRAM-based neural network computing via crossbar structure adaptive optimization. in Proceedings of the 25th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED), Aug. 2020.

- [ASPDAC'20] X. Ma, G. Yuan, S. Lin, C. Ding, **F. Yu**, T. Liu, W. Wen, X. Chen, Y. Wang. Tiny but Accurate: A Pruned, Quantized and Optimized Memristor Crossbar Framework for Ultra Efficient DNN Implementation. in Proceedings of the 25th Asia and South Pacific Design Automation Conference, Jan. 2020.

6.2 AI Security and Robustness

- [WACV'22] **F. Yu**, D. Wang, Y. Chen, Nikos. Karianakis, P. Yu, D. Lymberopoulos, S. Lu, W. Shi, X. Chen, SC-UDA: Style and Content Gap Aware Unsupervised Domain Adaptation for Object Detection, in Proceedings of the Winter Conference on Applications of Computer Vision (WACV), Jan. 2022.

- [IJCAI'19] **F. Yu**, Z. Qin, C. Liu, L. Zhao, Y. Wang, and X. Chen. Interpreting and Evaluating Adversarial Robustness. in Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Aug. 2019.

- [DAC'19] **F. Yu***, Z. Xu*, C. Liu, and X. Chen. MASKER: Adaptive Mobile Security Enhancement against Automatic Speech Recognition in Eavesdropping. in Proceedings of the 56th Design Automation Conference (DAC), Jun. 2019.

- [ASPDAC'19] **F. Yu**, C. Liu, and X. Chen. REIN: A Robust Training Method for Enhancing Generalization Ability of Neural Networks in Autonomous Driving Systems. in Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC), Jan. 2019.

- [ASPDAC'19] Z. Xu, **F. Yu**, C. Liu, and X. Chen. HAMPER: High-Performance Adaptive Mobile Security Enhancement against Malicious Speech and Image Recognition. in Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC), Jan. 2019.

- [ASPDAC'20] Z. Xu, **F. Yu**, X. Chen. LanCe: A Comprehensive and Lightweight

CNN Defense Methodology against Physical Adversarial Attacks on Embedded Multimedia Applications. in Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASPDAC), Jan. 2020.

- [TCAD'20] **F. Yu**, Z. Qin, C. Liu, D. Wang, X. Chen. REIN the RobuTS: Robust DNN-based Image Recognition in Autonomous Driving Systems. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.

6.3 Deep Learning Interpretability

- [MFC'18] Z. Qin, **F. Yu**, C. Liu, X. Chen. How convolutional neural networks see the world - A survey of convolutional neural network visualization methods. Journal of Mathematical Foundations of Computing, pp.149-180, May 2018.

- [BMVC'19] Z. Qin, **F. Yu**, C. Liu, and X. Chen. Functionality-Oriented Convolutional Filter Pruning. in Proceedings of the 30th British Machine Vision Conference, Sep. 2019.

6.4 Federated Learning

- [KDD'21] **F. Yu**, W. Zhang, Z. Qin, Z. Xu, D. Wang, C. Liu, Z. Tian, and X. Chen. Fed^2: Feature Aligned Federated Learning. in Proc. of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), Aug. 2021.

- [DAC'21] Z. Xu, **F. Yu**, J. Xiong, X. Chen. Helios: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration. in Proceedings of the 58th Design Automation Conference (DAC), Jun. 2021.

- [KDD'19] J. Wang, **F. Yu**, X. Chen, L. Zhao. ADMM for Efficient Deep Learning with Global Convergence, in Proceedings of the 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD), Aug. 2019.

Bibliography

[1] U. Harvard, "Deep learning 101," Tech. Rep., 2017, *http* : *//beamlab.org/deeplearning/*2017/02/23/*deep_learning_*101*_part*1.*html*.

[2] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," *arXiv preprint arXiv:1911.11313*, 2019.

[3] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The building blocks of interpretability," *Distill*, vol. 3, no. 3, p. e10, 2018.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[5] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, 1960.

[6] S. Linnainmaa, "Taylor expansion of the accumulated rounding error," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 146–160, 1976.

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

[8] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[9] M. Reports, "Global data center accelerator market size, status and forecast 2020-2025," 2021, https://www.mynewsdesk.com/brandessence/pressreleases/data-center-accelerator-market-size-2021-cagr-38-dot-7-percent-3112488.

[10] N. Coorporation, "Nvidia tesla v100 gpu architecture," Tech. Rep., 2017, http://www.nvidia.com/object/volta-architecture.

[11] "Nvidia a100 whitepaper," 2020, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for

deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[14] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[15] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1389–1397.

[16] Y. Wang, S. Ye, Z. He, X. Ma, L. Zhang, S. Lin, G. Yuan, S. H. Tan, Z. Li, D. Fan *et al.*, "Non-structured dnn weight pruning considered harmful," *arXiv preprint arXiv:1907.02124*, 2019.

[17] X. Ding, G. Ding, J. Han, and S. Tang, "Auto-balanced filter pruning for efficient convolutional neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[18] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[19] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[20] J. Fang, Y. Shen, Y. Wang, and L. Chen, "Optimizing dnn computation graph using graph substitutions," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2734–2746, 2020.

[21] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[22] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[23] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality saturation for tensor graph superoptimization," *arXiv preprint arXiv:2101.01332*, 2021.

[24] F. Yu and et al., "Automated runtime-aware scheduling for multi-tenant dnn inference on gpu," in *Proceedings of the 40th IEEE International Conference on Computer Aided Design (ICCAD)*, 2021.

[25] H. Shen, J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, and Y. Wang, "Nimble: Efficiently compiling dynamic neural networks for model inference," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[26] Y. Choi and M. Rhu, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 220–233.

[27] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 863–879.

[28] P. Fegade, T. Chen, P. B. Gibbons, and T. C. Mowry, "Cortex: A compiler for recursive deep learning models," *arXiv preprint arXiv:2011.01383*, 2020.

[29] Nvidia, "Nvidia multi process service (mps)," 2020, https://docs.nvidia.com/deploy/pdf/CUDA-Multi-Process-Service-Overview.pdf.

[30] NVIDIA, "Nvidia multi instance gpu (mig)," 2020, https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[31] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.

[32] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Multi-model machine learning inference serving with gpu spatial partitioning," *arXiv preprint arXiv:2109.01611*, 2021.

[33] J. Deng and et al., "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.

[34] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[35] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[36] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[37] A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[39] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[40] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[41] Y. Cheng and et al., "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.

[42] R. Mishra, H. P. Gupta, and T. Dutta, "A survey on deep neural network compression: Challenges, overview, and solutions," *arXiv preprint arXiv:2010.03954*, 2020.

[43] F. Yu, D. Stamoulis, D. Wang, D. Lymberopoulos, and X. Chen, "Exploring the design space of efficient deep neural networks," *arXiv preprint arXiv:2011.10912*, 2020.

[44] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.

[45] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *arXiv preprint arXiv:1608.03665*, 2016.

[46] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[47] C.-K. Yeh, I. E. Yen, H.-Y. Chen, C.-P. Yang, S.-D. Lin, and P. Ravikumar, "Deeptrim: Revisiting l1 regularization for connection pruning of deep network," 2018.

[48] Z. Qin, F. Yu, C. Liu, and X. Chen, "Functionality-oriented convolutional filter pruning," in *The British Machine Vision Conference*, 2019.

[49] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[50] F. Yu, Z. Qin, and X. Chen, "Distilling critical paths in convolutional neural networks," *arXiv preprint arXiv:1811.02643*, 2018.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[52] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *5th International Conference on Learning Representations, ICLR*, 2019.

[53] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*, 2019, pp. 4340–4349.

[54] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[55] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[56] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," *arXiv preprint arXiv:1810.11809*, 2018.

[57] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[58] J.-H. Luo and *et al.*, "Thinet: A filter level pruning method for deep neural network compression," *arXiv:1707.06342*, 2017.

[59] H. Li and *et al.*, "Pruning filters for efficient convnets," *arXiv:1608.08710*, 2016.

[60] Y. He and *et al.*, "Channel pruning for accelerating very deep neural networks," in *Proc. of ICCV*, 2017.

[61] S. Han and *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv:1510.00149*, 2015.

[62] C. Zhang and *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proc. of FPGA*, 2015.

[63] M. Alwani and *et al.*, "Fused-layer cnn accelerators," in *MICRO*, 2016.

[64] G. Li and *et al.*, "Block convolution: towards memory-efficient inference of large-scale cnns on fpga," 2018.

[65] Y. Ma and *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *FPGA*, 2017.

[66] 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.

[67] M. Peemen and *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *Proc. of ICCD*, 2013.

[68] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[69] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[70] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[71] X. Li, Y. Zhou, Z. Pan, and J. Feng, "Partial order pruning: for best speed/accuracy trade-off in neural architecture search," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2019, pp. 9145–9153.

[72] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[73] X. Chang, H. Pan, W. Sun, and H. Gao, "Yoltrack: Multitask learning based real-time multiobject tracking and segmentation for autonomous vehicles," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[74] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2636–2645.

[75] S. Chowdhuri, T. Pankaj, and K. Zipser, "Multinet: Multi-modal multi-task learning for autonomous driving," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2019, pp. 1496–1504.

[76] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu, and N. Sang, "Bisenet: Bilateral segmentation network for real-time semantic segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 325–341.

[77] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *arXiv preprint arXiv:1506.01497*, 2015.

[78] Z. Kim, "Robust lane detection and tracking in challenging scenarios," *IEEE Transactions on Intelligent Transportation Systems*, vol. 9, no. 1, pp. 16–26, 2008.

[79] Microsoft, "Deep learning inference service at microsoft," 2020, https://www.usenix.org/system/files/opml19papers-soifer.pdf.

[80] NVIDIA, "Nvidia triron inference server," 2020, https://developer.nvidia.com/nvidia-triton-inference-server.

[81] "Mpi for python," 2020, https://mpi4py.readthedocs.io/en/stable/.

[82] NVIDIA, "Cuda streams," 2020, https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[83] TensorFlow, "Tensorflow xla (accelerated linear algebra)," 2020, https://www.tensorflow.org/xla.

[84] NVIDIA, "Nvidia tensorrt," 2020, https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[85] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie *et al.*, "Transferable graph optimizers for ml compilers," *arXiv preprint arXiv:2010.12438*, 2020.

[86] microsoft, "Deep learning inference service at microsoft," 2020, https://www.usenix.org/system/files/opml19papers-soifer.pdf.

[87] P. Yu and M. Chowdhury, "Salus: Fine-grained gpu sharing primitives for deep learning applications," *arXiv preprint arXiv:1902.04610*, 2019.

[88] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1389–1397.

[89] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4340–4349.

[90] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[91] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[92] NVIDIA, "Nvidia titan v — volta architecture," 2020, https://www.nvidia.com/en-us/titan/titan-v/.

[93] ——, "Cuda pro tips (page 13)," 2020, https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf.

[94] Nvidia, "Nsight compute — nvidia," 2020, https://developer.nvidia.com/nsight-compute.

[95] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao, "Hrank: Filter pruning using high-rank feature map," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1529–1538.

[96] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 2234–2240.

[97] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[98] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[99] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[100] A. Sampson, T. Chen, and J. Roesch, "Apache tvm unity: a vision for the ml software and hardware ecosystem," 2022, https://tvm.apache.org/2021/12/15/tvm-unity.

[101] M. Lui, Y. Yetim, Ö. Özkan, Z. Zhao, S.-Y. Tsai, C.-J. Wu, and M. Hempstead, "Understanding capacity-driven scale-out neural recommendation inference," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 162–171.

Curriculum Vitae

Fuxun Yu received his Ph.D. degree in Electrical and Computer Engineering from George Mason University in May, 2022. He obtained the B.S. degree in Harbin Institute of Technology, China. He will join Microsoft (Redmond) as a Senior Researcher in summer 2022. His research interests include deep learning security and robustness, model compression and acceleration, full-stack model performance optimization on GPUs. He has published over 30 papers in top conferences such as SIGKDD, IJCAI, MLSYS, DAC, DATE, ICCAD and received one Best Paper Nomination in DATE'2020.