CocoPIE: Making Mobile AI Sweet As PIE —Compression-Compilation Co-Design Goes a Long Way

Shaoshan Liu[†], Bin Ren[*], Xipeng Shen[‡], Yanzhi Wang[◇]
[†]Perceptin Inc. [*]William & Mary
[‡]North Carolina State University [◇]Northeastern University
Contact: info@cocopie.ai

It has been a consensus that the company who enables real intelligence on end devices (such as mobile devices and IoT devices) will define the future of computing. Racing towards this goal, many companies, whether giant technology firms such as Google, Microsoft, Amazon, Apple and Facebook, or startups spent tens of billions of dollars each year on R&D.

Assuming hardware is the major constraint for enabling real mobile intelligence, the industry has mainly dedicated their efforts to developing specialized hardware accelerators for machine learning and inference. Billions of dollars have been spent to fuel this intelligent hardware race.

This article challenges the assumption. By drawing on a recent real-time AI optimization framework CoCoPIE, it maintains that with effective *compression-compiler co-design*, it is possible to enable real-time artificial intelligence (AI) on mainstream end devices without special hardware.

The principle of *compression-compilation co-design* is to design the compression of Deep Learning Models and their compilation to executables in a hand-in-hand manner. This synergistic method can effectively optimize both the size and speed of Deep Learning models, and also can dramatically shorten the tuning time of the compression process, largely reducing the time to the market of AI products. When applied to models running on mainstream end devices, the method can produce real-time experience across a set of AI applications that had been broadly perceived possible only with special AI accelerators.

Foregoing the need for special hardware for real-time AI has some profound implications, thanks to the multi-fold advantages of mainstream processors over special hardware:

- Time to market: Special hardware often takes multiple years before it reaches the market. The creation of the associated compiler and system software further lengthens the process. Applications using such hardware often needs to use the special APIs and meet many special constraints (e.g., tiling computations to a certain size), which lengthens the time to market of AI product.

- Cost: Developing a special ASIC processor is costly, and adding them into existing systems incurs extra expenses.

- Technology maturity: Unlike general-purpose processors, special hardware has a much smaller production volume; the technology available for their production is hence usually several generations behind general-purpose processors. Most AI accelerators, for instance, are based on 28 to 65nm CMOS technology, with a transistor density over 10$\times$ lower than state-of-art mobile CPU or GPU.

- Speed: As a consequence of the old technology, special processors run much slower than general-purpose processors do.
- Eco-system: General-purpose processors have a well-developed eco-system (debugging tools, optimization tools, security measures), which makes the development of high-quality applications much easier than on special processors.
- Adoption: For all the above reasons, the adoption of a special processor is usually limited to the company that creates it and its few close customers. As a result, an AI application developed for the processor can be adopted by a limited number of devices.

Therefore, whenever mainstream processors can meet the speed and efficiency requirements of an AI application, they should be the preferred device to consider. The common perception that drives the current emphasis in the industry on pursuing special hardware for AI is that main-stream processors are insufficient to meet the real-time requirements. In the rest of this article, we explain why the perception is wrong when compression-compilation co-Design is used, how the principle can be materialized effectively into a practical framework CoCoPIE, and how the future will look like for real-time AI.

1 Compression-Compilation Co-Design: the Concept

Compression and compilation are the two key steps in fitting a deep learning model on a hardware for efficient executions. Model compression is a common technique for reducing the size and improving the speed of deep learning models. Compression techniques fall into two categories, *pruning* and *quantization*. Pruning removes layers or convolution filters or channels, while quantization reduces the precision of parameters (e.g., floating-point to short integer). Compilation refers to the process of generating executable code from a given deep learning model. It, in essence, is a process of mapping the high-level operations in deep learning to the low-level instructions that the underlying hardware supports. The process plays a critical role in optimizing the code for efficient executions.

The principle of compression-compilation co-design is to design the two components for AI in a hand-in-hand manner. The synergy may exhibit at three levels.

(1) Demands/Preferences Level: At this level, the synergy is on taking the preferences or demands of one component into consideration when designing the other component. An example is that main-stream processors typically prefer code with certain computation patterns; if model compression step can consider that preference, it could create a scenario more amendable for the compilation step to work effectively, as Section 2.1 shows.

(2) Perspective/Insight Level: At this level, the synergy is on taking the perspective or insights in the domain of one component when treating the problems in the domain of the other component. An example is the principle of composability or modularity that has been playing an essential role in keeping programming systems and compilations efficient and scalable. Section 2.2 will show that when this perspective is introduced into model pruning, large efficiency benefits entail.

(3) Methodology Level: At this level, the synergy is on closely integrating the methodology of the two components together. Section 2.2 illustrates this synergy through a compiler framework that automatically generates code to enable a new way of deep learning pruning, which speeds the process by up to 180X.

All the examples we have mentioned are part of a software framework for Mobile AI named CoCoPIE. We will next give an overview of CoCoPIE, and then uses each of its main components

to explain the compression-compilation co-design principle and the significant benefits.

2 CoCoPIE

CoCoPIE stands for Compression-Compilation co-design for Performance, Intelligence, and Efficiency. It is a software framework that we have recently put together for enabling real-time AI on mainstream end devices.

CoCoPIE holds *numerous records on mobile AI*: the first framework that supports all main kinds of DNNs, from CNNs to RNNs, transformer, language models, and so on; the fastest DNN pruning and acceleration framework, up to 180X faster compared with current DNN pruning on other frameworks such as TensorFlow-Lite; making many representative AI applications able to run in real-time on off-the-shelf mobile devices that have been previously regarded possible only with special hardware support; making off-the-shelf mobile devices outperform a number of representative ASIC and FPGA solutions in terms of energy efficiency and/or performance.

CoCoPIE consists of two main components, which both reflect the Compression-Compilation co-design principle. The first component, *CoCo-Gen*, generates efficient DNN execution codes via a synergy of pattern-based DNN pruning and pattern-aware code generation. The second component, *CoCo-Tune*, dramatically shortens the process in identifying the appropriate set of DNN parameters to prune by a composability-based compiler framework. We next explain each of the two components and how compression-compilation co-design makes them possible.

2.1 CoCo-Gen: Pattern-based Pruning and Code Generation[1]

Along with the great success of Deep Neural Networks (DNNs) are the increasingly large model size and complex model structure that require tremendous computation and memory resources to fulfill the *real-time* requirement of many key applications. As a mainstream model compression technique, weight pruning is proposed to mitigate this challenge. Existing pruning however is either incompatible with modern parallel architectures, resulting in long inference latency (e.g., non-structured fine-grained pruning), or subject to significant accuracy degradation (e.g., structured coarse-grained pruning).

CoCo-Gen advances the state-of-the-art weight pruning techniques by introducing a new dimension, fine-grained pruning patterns inside the coarse-grained structures, revealing a previously unknown point in the design space. With the higher accuracy enabled by fine-grained pruning patterns, the unique insight is to use the *compiler-based code generation* to re-gain and guarantee high hardware efficiency. In other words, our method achieves the best of both worlds, and provides a more favorable option at the levels of theory/algorithm, compiler, and hardware than prior pruning methods.

2.1.1 DNN Compression: Challenges and Opportunities

DNN model compression has been proposed for simultaneously reducing the storage/computation and accelerating inference with minor classification accuracy (or prediction quality) loss. Two important categories of DNN model compression techniques are weight pruning [19, 17, 10, 42, 54, 22] and weight quantization [35, 47, 63, 38, 56, 48, 27, 9].

Weight pruning reduces the redundancy in the number of weights. As shown in Figure 1, two main approaches of weight pruning are (1) the general and non-structured pruning; and (2) structured pruning. The two approaches produce irregular and regular compressed DNN models, respectively.

[1] **This section is largely based on two published papers [41, 46]**

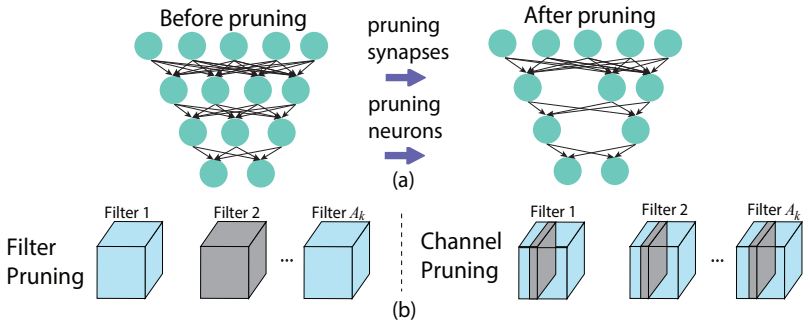


Figure 1: (a) Non-structured weight pruning and (b) two types of structured weight pruning.

Non-Structured Pruning: In this method, arbitrary weights can be pruned. It can result in a high pruning rate (i.e., reduction in the number of weights). However, for compiler and code optimization, non-structured pruning incurs several challenges due to the irregularity in computation and memory access. Similarly, for hardware acceleration, since the pruned models are stored in some sparse matrix format with indices, they often lead to performance degradation in GPU and CPU implementations [42, 54, 22].

Structured Pruning: This method can produce regular smaller weight matrices. Figure 1 (b) illustrates the representative structured pruning schemes: *filter pruning* and *channel pruning* [54]. Filter and channel pruning can be considered as equivalent in that pruning a filter in the k-th layer is equivalent to pruning the corresponding channel in the $(k+1)$-th layer. Filter/channel pruning is compatible with Winograd algorithm [55, 33] that has been used to accelerate computation of the original DNNs. Due to the regular structure, the GPU/CPU implementations typically lead to more significant acceleration [42, 54, 22]. However, the structured pruning suffers from notable accuracy loss [54, 22].

Opportunity: From the above discussion of non-structured and structured pruning schemes, these two pruning schemes represent two extremes in the design space. In non-structured pruning, any weight can be pruned, and we consider it as a fine-grained method; in structured pruning, the weights of whole filter or channel are pruned, and we consider it as a coarse-grained method. We seek an approach that can offer, or even go beyond, the best of both methods, the high accuracy of non-structured pruning and hardware friendliness of structured ones.

To achieve this goal, we introduce a new dimension, *fine-grained pruning patterns inside the coarse-grained structures*, revealing a previously *unknown* point in the design space.

2.1.2 Design Philosophy of Pattern-based Pruning

The proposed *pattern-based pruning* possesses both *flexibility* and *regularity*, and we take a unique approach and leverage compiler optimizations as a bridge between algorithm-level compression and embedded hardware acceleration. The flexibility is clearly desirable at the theory and algorithm level, but is also compatible with compiler *code generation* to maximize or maintain both instruction-level and thread-level parallelism. The regularity enables another important compiler optimization, *redundant load elimination*, to further improve hardware performance.

The proposed pattern-based pruning techniques consist of *kernel pattern pruning* and *connectivity pruning*. They can be naturally combined in CoCo-Gen, at both algorithm and compiler levels.

Kernel Pattern Pruning is illustrated in Figure 2. For each kernel (in a CONV filter), a fixed number of weights are pruned, and the remaining weights (white cells) form specific "patterns". We define the example in Figure 2 as 4-entry pattern pruning, since every kernel reserves 4 non-

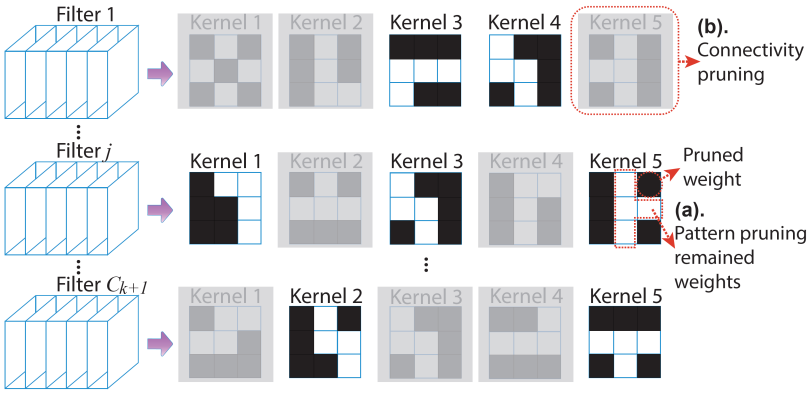


Figure 2: Illustration of (a) kernel pattern pruning on CONV kernels, and (b) connectivity pruning by removing kernels.

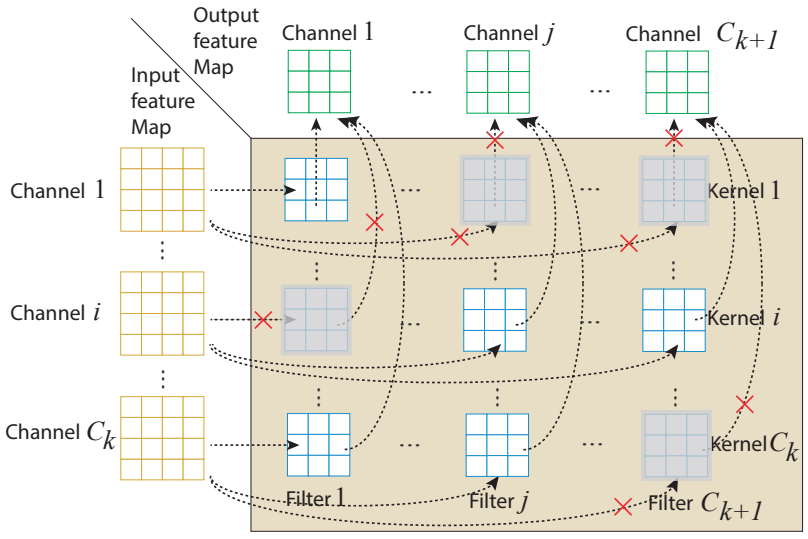


Figure 3: Illustration of connectivity pruning.

zero weights out of the original 3×3 kernel (the most commonly used kernel). We can generalize to other kernel sizes and FC layer. For each kernel, it possesses *flexibility* in choosing among a number of pre-defined patterns.

At theory and algorithm levels, it is shown in [41, 37, 34] that the desirable kernel shape has certain patterns to match the connection structure in human visual systems, instead of a square shape. The selection of appropriate pattern for each kernel can be achieved by extending ADMM-based framework. As shown in [41], we achieve accuracy enhancement in all representative DNNs in our testing. At compiler level, the known patterns allow a compiler to *re-order and generate codes* at filter level and kernel level to group kernels with the same pattern for consecutive executions, thereby maximizing instruction-level parallelism. At hardware level, 4-entry patterns perfectly fit the SIMD architecture in embedded processors, for both CPUs and GPUs.

Connectivity Pruning is illustrated in Figure 3. The key insight is to *cut the connections* between certain input and output channels, which is equivalent to the removal of corresponding kernels. The method is proposed to achieve higher weight pruning/acceleration rates in combination with kernel pattern pruning.

Table 1: Qualitative comparison of different pruning schemes on accuracy and speedup under the same pruning rate.

Pruning Scheme	Accuracy				Hardware Speedup			
	Highest	Minor Loss	Moderate Loss	Highest Loss	Highest	High	Moderate	Minor
Non-structured	X							X
Filter/Channel				X	X			
Pattern	**X**				**X**			
Connectivity		**X**				**X**		

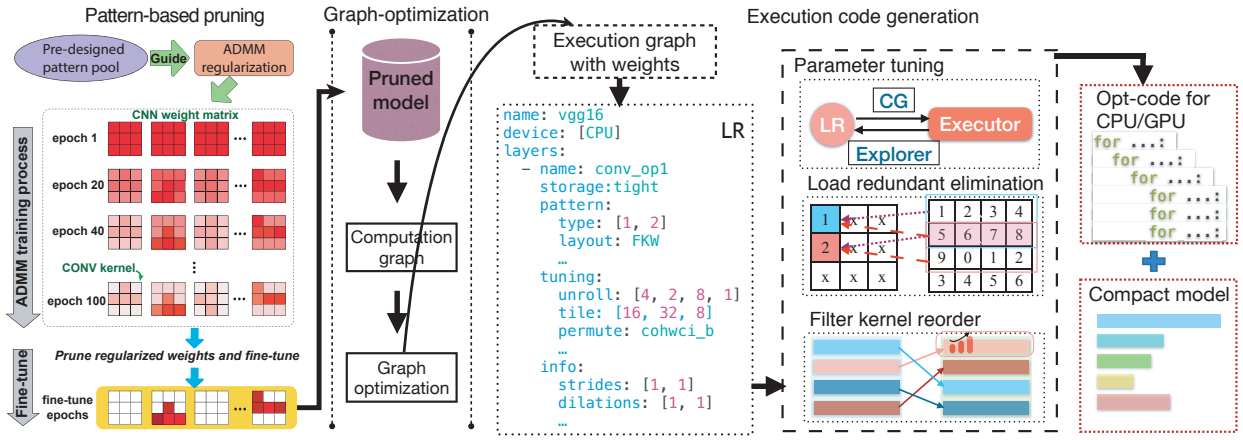


Figure 4: Overview of CoCo-Gen acceleration framework.

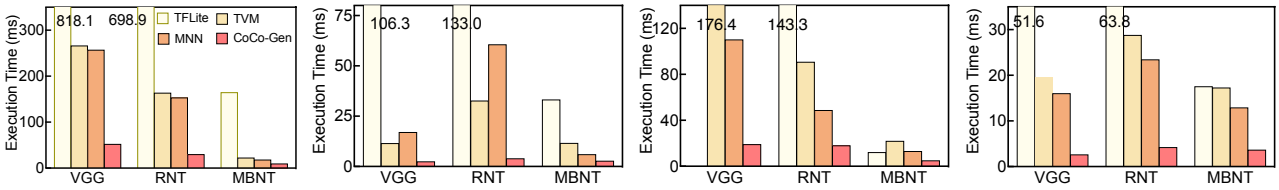
At theory and algorithm levels, connectivity pruning matches the desirability of locality in layerwise computations inspired by human visual systems [57, 58]. It is more flexible than filter/channel pruning and achieves higher accuracy. At compiler and hardware levels, removed kernels and associated computations are grouped by compiler using the *re-ordering* capability without affecting the other computations, thereby maintaining parallelism degree.

2.1.3 Internal Mechanisms of CoCo-Gen

Figure 4 shows the overview of CoCo-Gen which consists of two stages: (1) *pattern-based training stage*, which performs kernel pattern and connectivity pruning with an extended ADMM solution framework. (2) *execution code generation stage*, which performs multiple effective optimizations based on the patterns. Similar to TVM [7], CoCo-Gen converts DNN models into computational graphs and applies multiple graph-based optimizations. Based on these optimizations, we focus on layerwise design and optimization including a high-level and fine-grained DNN layerwise representation (LR), filter kernel reorder, load redundancy eliminations, and automatic parameter tuning. All of these designs and optimizations are general, and applicable to both mobile CPUs and GPUs. The second stage generates optimized execution codes as well as DNN models with weights stored in a novel compact format. We briefly explain each component as follows.

Pattern-based training stage performs effective kernel pattern and connectivity pruning in the training phase, in order to achieve the highest pruning (acceleration) rate without accuracy loss. First, we design a set of patterns to select for each kernel. Then we perform pattern pruning based on the designed pattern set and connectivity pruning, using an extended ADMM-based method.

Fine-grained DNN layerwise representation (LR) provides a high-level representation to enable our general optimization on DNN models from various resources. This LR captures more



(a) ImageNet-CPU (b) CIFAR-10-CPU (c) ImageNet-GPU (d) CIFAR-10-GPU

Figure 5: Performance comparison: x-axis: different trained DNN models; y-axis: average DNN inference execution time on a single input.

extensive information of each DNN layer compared with the TVM's IR. In particular, it includes the pattern and tuning related information. The compiler optimizations rely on a series of improvements on this LR to generate the compact model and optimized execution codes.

Filter kernel reorder addresses two challenges of pattern-based pruning—heavy control-flow instructions, and thread divergence and load imbalance—by grouping the filters with similar lengths and patterns together. Because of the relatively limited number of patterns, the kernels with similar patterns can be organized together through proper filter kernel reordering, thereby significantly reducing the control-flow instructions and improving the *instruction-level parallelism*. Moreover, if different threads process different filters, thread divergence and load imbalance issues are properly resolved because the kernels in each filter have similar computation workload, thereby enhancing *thread-level parallelism*.

Compressed weight storage is specifically designed for our kernel pattern and connectivity pruning. Together with filter kernel reorder, this compact data structure yields much better compression rates than the conventional CSR (compressed sparse row) format.

Load redundancy elimination addresses the poor memory performance challenge of pattern-based pruning by exploring two novel *register-level* load redundancy opportunities during the kernel execution code generation. It is crucial, especially when the data movements between memory and cache have already been optimized with advanced tiling techniques.

Parameter auto-tuning specifically tests on different configurations of the key performance parameters, including strategies of placing data on various GPU memories, different tiling sizes, and loop permutations for each DNN layer on each processing unit.

In sum, allowing compilers to treat pruned kernels as special patterns, our approach not only achieves high pruning rate with high accuracy, but also effectively converts the patterns into performance improvements for their hardware friendly properties. As shown in Table 1, CoCo-Gen can achieve the benefits of both non-structured and structured pruning. It illustrates the co-design principle: the multi-level cache memory hierarchy provides sufficient hardware supports to hide memory access latency and explore locality, and the SIMD units offers vector/parallel computing capability, the potential of which is unleashed more effectively when the compilation and compression processes are co-designed synergistically.

2.1.4 Evaluation and Demos

Results on DNNs: We evaluate CoCo-Gen on a Samsung Galaxy S10 cell phone with the latest Qualcomm Snapdragon 855 mobile platform that consists of a Qualcomm Kryo 485 Octa-core CPU and a Qualcomm Adreno 640 GPU. Figure 5 shows the CPU and GPU performance of CoCo-Gen compared to TFLite [14], TVM [7], and MNN [2] on six representative DNNs, VGG-16 (VGG), ResNet-50 (RNT), and MobileNet-V2 (MBNT) trained on two datasets, ImageNet and CIFAR-10.

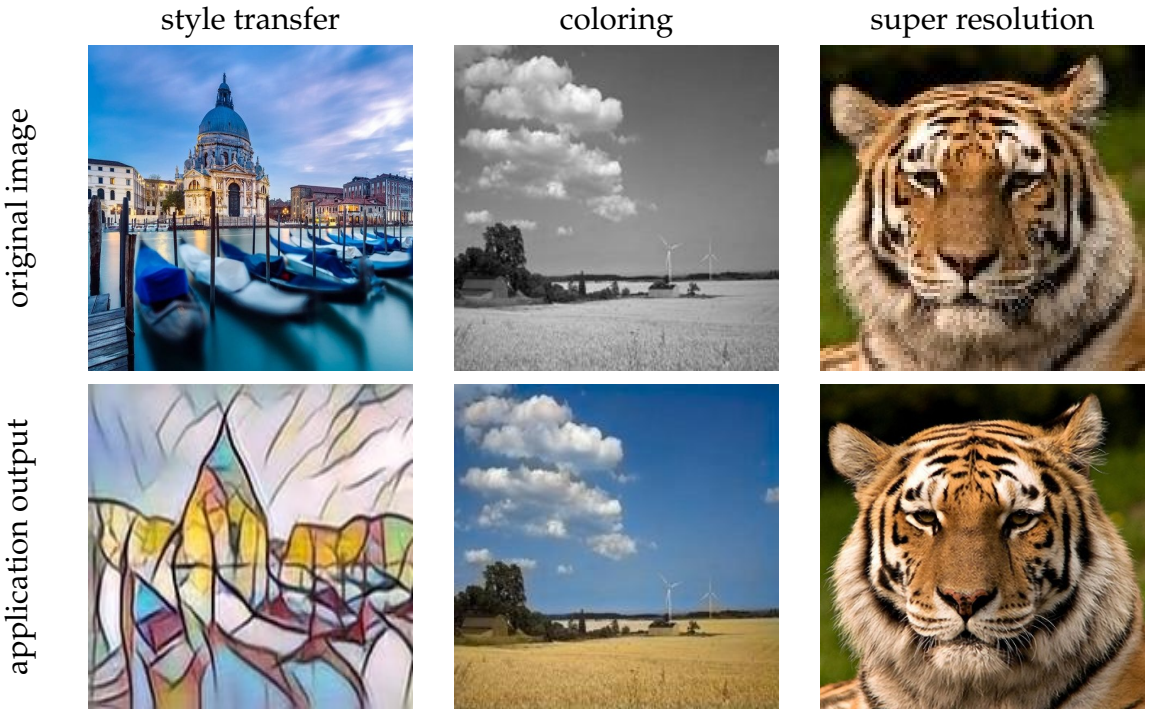


Figure 6: Examples of style transfer, coloring, and super resolution implemented on our mobile device.

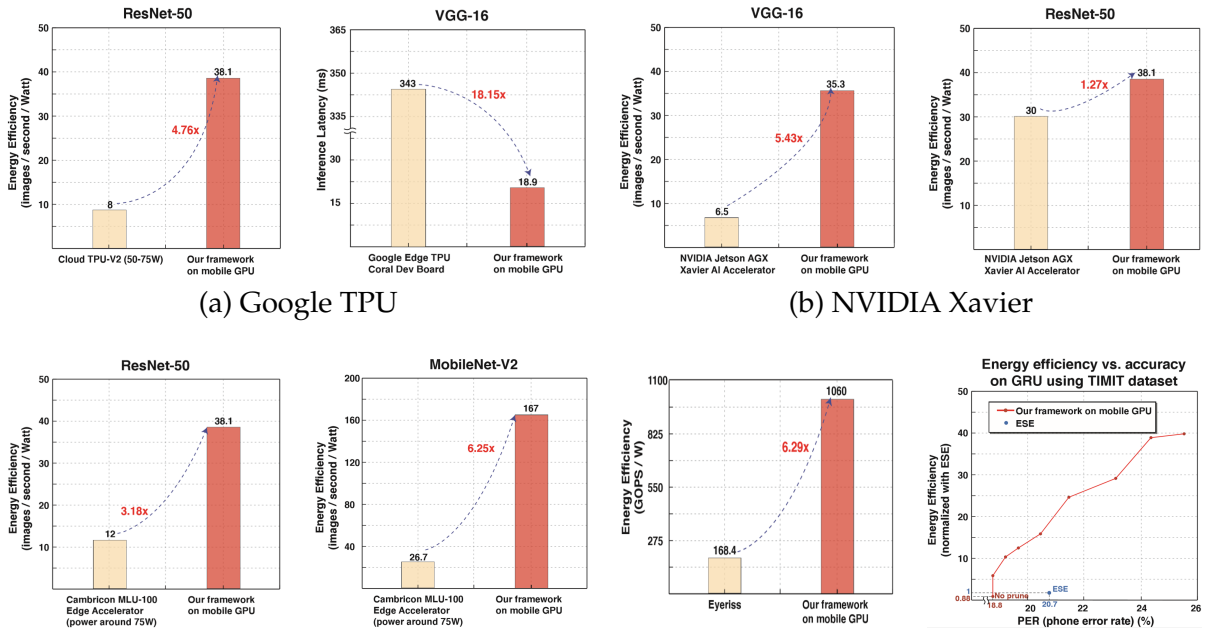
CoCo-Gen outperforms all other frameworks for all cases. On CPU, CoCo-Gen achieves $12\times$ to $44.5\times$ speedup over TFLite, $2.3\times$ to $8.1\times$ over TVM, and $1.9\times$ to $15.5\times$ over MNN, respectively. On GPU, CoCo-Gen achieves $2.5\times$ to $20\times$, $4.1\times$ to $11.4\times$, and $2.5\times$ to $6.2\times$ speedup over TFLite, TVM, and MNN, respectively[2]. For the largest DNN (VGG) and largest data set (ImageNet), CoCo-Gen completes CONV layers on a single input within 18.9 ms on GPU, meeting the real-time requirement (usually 30 frames/sec, i.e., 33 ms/frame).

Real Application Demos: We also demonstrate the efficacy of CoCo-Gen through three interesting and key DNN applications, style transfer [13], DNN coloring [28], and super resolution [11]. The style transfer model is based on a generative network [61] trained on Microsoft COCO [39]. DNN coloring uses the Places scene [64] dataset to train a novel architecture that can jointly extract and fuse global and local features to perform the final colorization. The super resolution model mainly utilizes residual blocks with wider activation and linear low-rank convolution [59] trained on the DIV2K [52] dataset. With structured pruning and compiler optimization, we implement the models on a Samsung Galaxy S10 mobile phone. We demonstrate that our implementations are able to achieve real-time inference on off-the-shelf mobile device with video demos.

Figure 6 shows sample input and output of three applications. CoCo-Gen optimization accelerates the inference with speedups of $4.2\times$, $3.6\times$, and $3.7\times$ for style transfer, coloring and super resolution, respectively. These results demonstrate that our optimized implementation generates satisfying output with high speed on mobile devices. More specifically, all inference can complete within 75 ms, showing the possibility of achieving real-time executions of complex DNN applications on such main-stream devices without special hardware. Please find more video demos at our YouTube channel[3].

[2]TFLite does not support executing VGG on ImageNet data set on GPU due to its too large memory footprint.

[3]`www.youtube.com/channel/UCCKVDtg2eheRTEuqIJ5cD8A/`.



(a) Google TPU (b) NVIDIA Xavier

(c) Cambricon MLU-100 (d) Eyeriss (e) ESE

Figure 7: Comparison with existing ASIC and FPGA solutions. (a) Comparison of energy efficiency and inference latency with Google cloud TPU. (b) Comparison of energy efficiency with NVIDIA Jetson AGX Xavier. (c) Comparison of energy efficiency with Cambricon MLU-100. (d) Comparison of energy efficiency with Eyeriss. (e) Comparison of energy efficiency with FPGA solution ESE.

2.1.5 Outperforming Existing ASIC and FPGA Solutions in Performance/Energy Efficiency

Using the CoCo-Gen on off-the-shelf general-purpose mobile device (e.g., the Samsung Galaxy S10 smartphone), we consistently outperform a number of ASIC and FPGA solutions in performance and energy efficiency. Figure 7 demonstrates (i) the comparison results on performance and energy efficiency with special ASIC hardware including Google's cloud TPU-V2 and edge TPU [15], NVIDIA Jetson AGX Xavier, Cambricon MLU-100, Eyeriss [8], etc., and (ii) comparison results on accuracy and energy efficiency with the FPGA solution ESE [18] (FPGA 2017 Best Paper Award) from DeePhi. This is a fair comparison on the same network models, and weight quantization is not adopted in CoCo-Gen solution (Eyeriss and ESE use 12-bit fixed-point quantizations).

We can clearly observe that our CoCo-Gen solution on general-purpose mobile device consistently outperforms representative ASIC/FPGA solutions in terms of energy efficiency. This unusual phenomenon is attributed to three reasons: (i) smartphone itself has ultra-high energy efficiency. Smartphone computing chips are built using the most advanced technology (e.g., 7nm, 11nm technology) and are the key driving force of technology advancement, while FPGA/ASIC solutions are based on 28nm or 40nm technologies which are inherently less energy-efficient. Also ARM (for mobile CPU) and Qualcomm (for mobile GPU) are especially proficient in high-efficiency circuit/system designs. (ii) while prior mobile compiler framework has limited support on different neural networks (e.g., not supporting RNNs or large-scale DNNs), our CoCo-Gen compiler can support all of the major types of neural networks, thereby unleashing the full potential of mobile devices. (iii) the unique benefit of compression-compilation co-design. Additionally, it can be observed that CoCo-Gen achieves consistently high performance on different DNN benchmarks thanks to the high flexibility of software-based solution. In contrast, it can be clearly

observed that current ASIC/FPGA solutions are optimized for a specific DNN type/size, thereby lacking generality. For example, edge TPU is optimized for small-scale DNNs while Cambricon MLU-100 is optimized for large-scale ones.

The studies reported in this part have shown that effective pruning coupled with pattern-based compilation can bring large performance benefits for DNNs. But finding out what is the best set of filters or connectivities to prune can be extremely time consuming. For a DNN with W filters, the entire configuration space of pruned network can be as large as $2^{|W|}$, even if only filter pruning is considered (adding pattern variations would worsen the complexity further). It often takes hours to evaluate just one configuration (i.e., training the pruned network and then testing it). We next show that the process can be shortened dramatically through a compiler-based framework, CoCo-Tune. It offers an example on the benefits of combining compiler perspective and support with DNN compression.

2.2 CoCo-Tune: A Compiler Framework for Fast Pruning[4]

CoCo-Tune is a compiler-based framework designed for shortening the time needed for CNN pruning in order to remove the major barrier for timely solution delivery in Artificial Intelligence (AI) product development, especially on mobile devices. Prior efforts on speeding up pruning have been, however, mostly focused on only the compression domain [36, 26, 43, 40, 21]. They leverage DNN algorithm-level knowledge to reduce the enormous configuration space to a smaller space (called *promising subspace*) that is likely to contain a good solution, and then evaluate these remaining configurations to find the best.

Although these prior methods help mitigate the problem, network pruning remains a time-consuming process. One reason is that, despite their effectiveness, no prior techniques can guarantee the inclusion of the desirable configuration in a much reduced subspace. As a result, to decrease the risk of missing the desirable configuration, practitioners often end up with a still quite large subspace of network configurations that takes days for many machines to explore. It is also quite often that modifications need to make to the CNN models, datasets, or hardware settings throughout the development process of an AI product; each of the changes could make the result of a CNN pruning obsolete and call for a rerun of the entire pruning process.

This study distinctively examines the problem from the programming systems perspective. Specifically, rather than improving the attainment of promising subspace as all prior work focuses on, we try to drastically speed up the evaluations of the remaining configurations in the promising subspace through cross-network *computation reuse* via a compiler-based framework, a direction that has never been explored before.

We achieve the goal through three-fold innovations. First, we empirically uncover the existence of *composability* in the training of a collection of pruned CNN models, and reveal the opportunity that the composability creates for saving computations in CNN pruning. The basic observation that leads to this finding is that two CNN networks in the promising subspace often differ in only some layers. In the current CNN pruning methods, the two networks are both trained from scratch and then tested for accuracy. A question asked in this work is whether the training results of the common layers can be reused across networks to save some training time. More generally, we view the networks in a promising subspace as compositions of a set of building blocks (a *block* is a sequence of CNN layers). The question is if we first pre-train (some of) these building blocks and then assemble them into the to-be-explored networks, can we shorten the evaluations of these networks and the overall pruning process? Through a set of experiments, we empirically validate

[4] **This section is largely based on a published paper [16].**

the hypothesis, based on which, we propose *composability-based CNN pruning* to capture the idea of reusing pre-trained blocks for pruning.

Second, we propose a novel *hierarchical compression-based algorithm*, which, for a given CNN and promising subspace, efficiently identifies the set of blocks to pre-train to maximize the benefits of computation reuse. We prove that identifying the optimal set of blocks to pre-train is NP-hard. Our proposed algorithm provides a linear-time heuristic solution by applying Sequitur [44], a hierarchical compression algorithm, to the CNN configurations in the promising subspace.

Finally, based on all those findings, we developed CoCo-Tune[5], the first compiler-based framework that, for an arbitrary CNN (in Caffe Prototxt format) and other inputs, automatically generates TensorFlow code to build Teacher-Student learning structures to materialize composability-based CNN pruning.

2.2.1 Composability-Based CNN Pruning: Idea and Challenges

The fundamental reason for CoCo-Tune to produce large speedups for CNN pruning is its effective capitalization of computation reuse based on the *composability in CNN pruning* that is empirically unveiled in this study. Two pruned networks in a promising subspace often differ in only some of the layers. The basic idea of *composability-based CNN pruning* is to reuse the training results of the common layers across the pruned networks. Although the idea may look straightforward, to our best knowledge, no prior CNN pruning work has employed such reuse, probably due to a series of open questions and challenges:

- First, there are bi-directional data dependencies among the layers of a CNN. In CNN training, for an input image, there is a forward propagation that uses a lower layer's output, which is called **activation maps**, to compute the activation maps of a higher layer; it is followed by a backward propagation, which updates the weights of a lower layer based on the errors computed with the higher layer's activation maps. As a result of the bi-directional dependencies, even just one-layer differences between two networks could cause very different weights to be produced for a common (either higher or lower) layer in the two networks. Therefore, it remains unclear whether the training results of a common layer could help with the training of different networks.

- Second, if a pre-trained layer could help, it is an open question how to maximize the benefits. A pre-trained sequence of consecutive layers may have a larger impact than a single pre-trained layer does on the whole network, but it may also take more time to produce and has fewer chances to be reused. How to determine which sets of layers or sequences of layers to pre-train to maximize the gains has not been explored before.

- Third, how to pre-train just a piece of a CNN? The standard CNN back propagation training algorithm uses input labels as the ground truth to compute errors of the current network configurations and adjust the weights. If we just want to train a piece of a CNN, what ground truth should we use? What software architecture should be built to do the pre-training and do it efficiently?

- Fourth, existing DNN frameworks support only the standard DNN training and inference. Users have to write code to do CNN pruning themselves, which is already complicated for general programmers. It would add even more challenges to ask them to additionally write

[5]The name is after CoCo-Tune steel, the legendary pioneering steel alloy developed in the 6th century BC; CoCo-Tune blades give the sharpest cuts.

the code to pre-train CNN pieces, and then reuse the results during the evaluations of the networks.

For the first question, we conduct a series of experiments on 16 large CNNs (four popular CNN models trained on four datasets). We briefly state the key observations here (see [16] for details). Pre-trained layers bring a network to a much improved starting setting, making the initial accuracies of the network 50-90% higher than the network without pre-trained layers. That leads to 30-100% savings of the training time of the network. Moreover, it helps the network converge to a significantly higher level of accuracy (by 1%-4%). These findings empirically confirm the potential of *composability-based CNN pruning*.

To effectively materialize the potential, we have to address the other three challenges. CoCo-Tune offers the solution.

2.2.2 CoCo-Tune Framework

CoCo-Tune is a software framework that automatically enables composability-based CNN pruning. As Figure 8 shows, its input has four parts:

- The to-be-pruned CNN model, written in Caffe Prototxt (with a minor extension), which is a user-friendly text format (from Caffe) for CNN model specifications [29].
- The promising subspace that contains the set of pruned networks configurations worth exploring. The subspace may come from the user or some third-party tools that reduce the configuration space for CNN pruning [24, 21, 3].
- The dataset for training and testing, along with some meta data on the training (e.g., learning rates, maximum training steps), following the format used in Caffe Solver Prototxt [1].
- The objectives of the CNN pruning, including the constraints on model size or accuracy.

The body of the CoCo-Tune framework consists of four main components as shown in Figure 8. (1) The *hierarchical tuning block identifier* tries to define the set of *tuning blocks*. A **tuning block** is a sequence of pruned consecutive CNN layers taken as a unit for pre-training. Suitable definitions of *tuning blocks* help maximize reuse while minimizing the pre-training overhead. (2) From the given CNN model specified in Prototxt, the *CoCo-Tune compiler* generates a *multiplexing model*, which is a function written in TensorFlow that, when invoked, specifies the structure of the full to-be-pruned CNN model, the network structure—which implements a Teacher-Student scheme—for pre-training tuning blocks, or pruned networks assembled with pre-trained tuning blocks, depending on the arguments the function receives. (3) The *pre-training scripts* are some generic Python functions that, when run, pre-train each tuning block based on the outputs from the first two components of CoCo-Tune. (4) The final component, *exploration scripts*, explores the promising pruned networks assembled with the pre-trained tuning blocks. The exploration of a network includes first fine-tuning the entire network and then testing it for accuracy. The exploration order is automatically picked by the *exploration scripts* based on the pruning objectives to produce the best network as early as possible. Both the *pre-training scripts* and the *exploration scripts* can run on one machine or multiple machines in a distributed environment through MPI.

The illustration is for a scenario where the promising subspace is given. For scenarios where the promising subspace is fully known at the beginning (e.g., the ADMM-based pruning mentioned in Section 2.1), CoCo-Tune helps in shortening the tuning process of the appropriate pruning rate for each layer of the neural network. We next give a deeper view of each of the main components of CoCo-Tune.

Hierarchical Compression-Based Algorithm Composability-based CNN pruning faces a trade-off between the pre-training cost and the time savings the pre-training results bring. The tradeoff depends on the definitions of the unit for pre-training, that is, the definition of *tuning blocks*. A *tuning block* is a unit for pre-training; it consists of a sequence of consecutive CNN layers pruned at certain rates. It can have various sizes, depending on the number of CNN layers it contains. The smaller it is, the less pre-training time it takes and the more reuses it tends to have across networks, but at the same time, its impact to the training time of a network tends to be smaller.

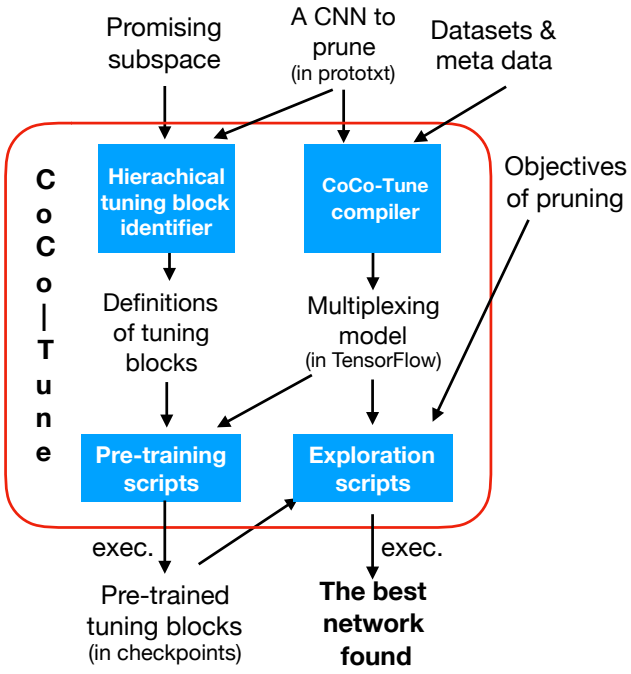


Figure 8: Overview of CoCo-Tune Framework.

To strike a good tradeoff between the pre-training cost and the benefits, we propose a hierarchical compression-baesd algorithm to help identify the best set of tuning blocks. Our algorithm leverages Sequitur [44] to efficiently identify the frequent sequences of pruned layers in the network collection C. As a linear-time hierarchical compression algorithm, Sequitur infers a hierarchical structure from a sequence of discrete symbols. For a given sequence of symbols, it derives a context-free grammar (CFG), with each rule in the CFG reducing a repeatedly appearing string into a single rule ID. Figure 9 gives an example. Its top part shows the concatenated sequence of layers of four networks pruned at various rates; the subscripts of the numbers indicate the pruning rate, that is, the fraction of the least important filters of a layer that are removed. The lower part in Figure 9 shows the CFG produced by Sequitur on the string. A full expansion of rule $r0$ would give the original string. The result can also be represented as a Directed Acyclic Graph (DAG) as the right graph in Figure 9 shows with each node corresponding to one rule.

Applying Sequitur to the concatenated sequence of all networks in the *promising subspace*, our *hierarchical compression-based algorithm* gets the corresponding CFG and the DAG. Let R be the collection of all the rules in the CFG, and S be the solution to the tuning block identification problem which is initially empty. Our algorithm then heuristically fills S with subsequences of CNN layers (represented as rules in the CFG) that are worth pre-training.

It does it based on the appearing frequencies of the rules in the *promising subspace* and their sizes (i.e., the number of layers a rule contains). It employs two heuristics: (1) A rule cannot be put into S if it appears in only one network (i.e., its appearing frequency is one); (2) a rule is preferred over its children rules only if that rule appears as often as its most frequently appearing descendant.

Pre-Training of Tuning Blocks The standard CNN back propagation training algorithm uses input labels as the ground truth to compute errors of the current network and adjusts the weights iteratively. To train a tuning block, the first question is what ground truth to use to compute errors. Inspired by Teacher-Student networks [6, 4, 23], we adopt a similar Teacher-Student mechanism to address the problem.

We construct a network structure that contains both the pruned block to pre-train and the orig-

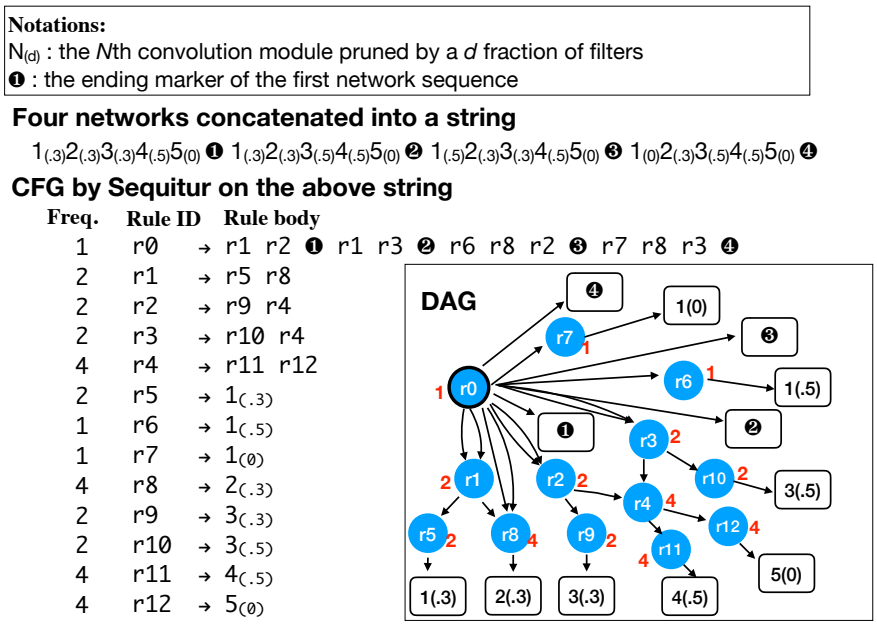


Figure 9: Sequitur applies to a concatenated sequence of layers of four networks pruned at rates: 0%, 30%, 50%.

inal full CNN model. They are put side by side as shown in Figure 10 (a) with the input to the counterpart of the tuning block in the full model also flowing into the pruned tuning block as its input, and the output activation map of the counterpart block flowing into the pruned tuning block as the "ground truth" of its output. When the standard back propagation algorithm is applied to the tuning block in this network structure, it effectively minimizes the reconstruction error between the output activation maps from the pruned tuning block and the ones from its unpruned counterpart in the full network. (In CNN pruning, the full model has typically already been trained beforehand to perform well on the datasets of interest.) This design essentially uses the full model as the "teacher" to train the pruned tuning blocks.

This Teacher-Student design has three appealing properties. First, it addresses the missing "ground truth" problem for tuning block pre-training. Second, as the full CNN model runs along with the pre-training of the tuning blocks, it provides the inputs and "ground truth" for the tuning blocks on the fly; there is no need to save to storage the activation maps which can be space-consuming considering the large number of input images for training a CNN. Third, the structure is friendly for concurrently pre-training multiple tuning blocks. As Figure 10 (b) shows, connections can be added between the full model and multiple pruned blocks; the pre-training of these blocks can then happen in one run, and the activation maps produced by a block in the full model can be seamlessly reused across the pre-training of multiple pruned blocks.

Global Fine-Tuning The local training phase outputs a bag of pre-trained pruned tuning blocks, as shown in Figure 10 (c) (tuning blocks in the original network could also be included). At the beginning of the *global fine-tuning* phase is an assembly step, which, logically, assembles these training blocks into each of the networks in the promising subspace. Physically, this step just needs to initialize the pruned networks in the promising subspace with the weights in the corresponding tuning blocks. We call the resulting network *a block-trained network*. Recall that one of the side

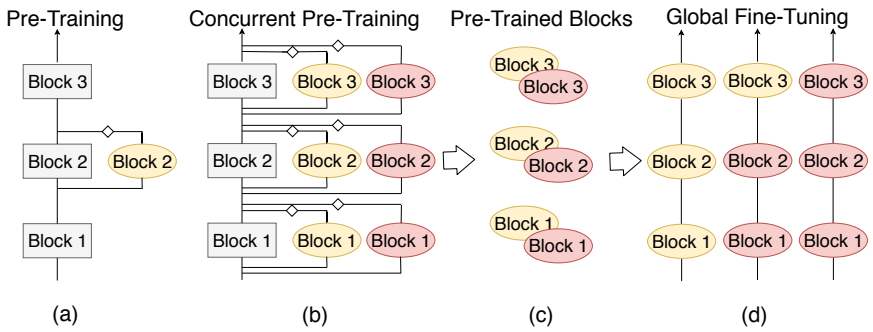


Figure 10: Illustration of composability-based network pruning. Eclipses are pruned tuning blocks; rectangles are original tuning blocks; diamonds refer to the activation map reconstruction error. Different colors of pruned tuning blocks correspond to different pruning options.

products of the tuning block identification step is a *composite vector* for each network which records the tuning blocks the network can use; these vectors are used in this assembly step. Figure 10 (d) gives a conceptual illustration; three networks are assembled with different sets of pre-trained tuning blocks.

As a pruned block with only a subset of parameters has a smaller model capacity, a *global fine-tuning step* is required to further recover the accuracy performance of a block-trained network. This step runs the standard CNN training on the *block-trained networks*. All the parameters in the networks are updated during the training. Compared with training a default pruned network, fine-tuning a block-trained network usually takes much less training time as the network starts with a much better set of parameter values as shown later in this article.

CoCo-Tune Compiler and Scripts CoCo-Tune compiler and scripts offer an automatic way to materialize the mechanisms described in the earlier parts of this section for an arbitrary CNN model. The proposed method is not restricted to a particular DNN framework, though we demonstrate its ability using TensorFlow.

TensorFlow APIs with other assistant libraries (e.g., Slim [50]) offer conveniences for standard CNN model training and testing, but *not* for CNN pruning, let alone *composability-based pruning*. Asking a general programmer to implement *composability-based pruning* in TensorFlow for each CNN model would add tremendous burdens on the programmer. She would need to write code to identify tuning blocks, create TensorFlow code to implement the customized CNN structures to pre-train each tuning block, generate checkpoints, and use them when creating the block-trained CNN networks for global fine-tuning.

CoCo-Tune compiler and scripts mitigate the difficulty by automating the process. The fundamental motivating observation is that the codes for two different CNN models follow the same pattern. Differences are mostly on the code specifying the structure of the CNN models (both the original and the extended for pre-training and global fine tuning). The idea is to build code templates and use the compiler to automatically adapt the templates based on the specifications of the models.

CoCo-Tune takes Prototxt as the format of an input to-be-pruned CNN model. It first generates a *multiplexing model*, which is a piece of TensorFlow code defined as a Python function. It is multiplexing in the sense that an invocation of the code specifies the structure of the original CNN model, or the structure for pre-training, or the global fine tuning model; which of the three modes is used at an invocation of the multiplexing model is determined by one of its input arguments,

Table 2: Dataset statistics.

	Dataset	Size			Classes	Accuracy			
		Total	Train	Test		ResNet-50	ResNet-101	Inception-V2	Inception-V3
General	ImageNet [49]	1,250,000	1,200,000	50,000	1000	0.752	0.764	0.739	0.780
Special	Flowers102 [45]	8,189	6,149	2,040	102	0.973	0.975	0.972	0.968
	CUB200 [53]	11,788	5,994	5,794	200	0.770	0.789	0.746	0.760
	Cars [32]	16,185	8,144	8,041	196	0.822	0.845	0.789	0.801
	Dogs [30]	20,580	12,000	8,580	120	0.850	0.864	0.841	0.835

mode_to_use. The multiplexing design allows easy code reuse as the three modes share much common code for model specifications. Another argument, *prune_info*, conveys to the multiplexing model the pruning information, including the set of tuning blocks to pre-train in this invocation and their pruning rates. The compiler generates code that maps CNN model specifications in Prototxt to TensorFlow APIs, specifies the derived network structure for pre-training each tuning block contained in *prune_info*, and conducts global fine-tuning of the pruned networks assembled from the pre-trained tuning blocks.

The compiler automates the composability-based pruning, making CoCo-Tune a tool for both speed and productivity for DNN pruning.

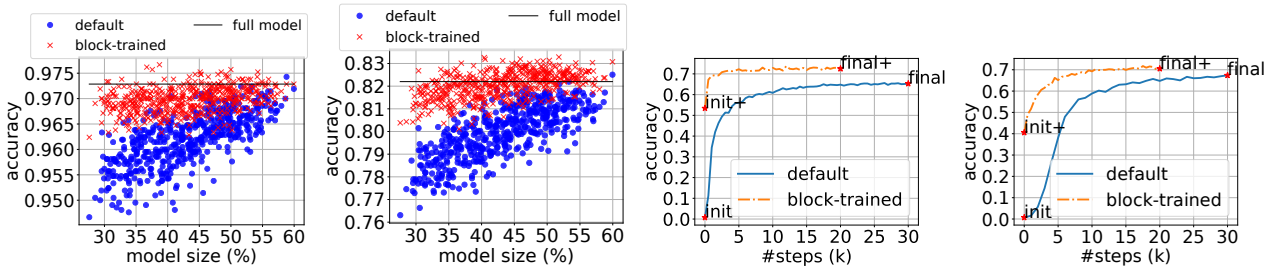
2.2.3 Evaluations

We conduct a set of experiments to examine the efficacy of CoCo-Tune. Our experiments use four popular CNN models: ResNet-50 and ResNet-101, as representatives of the Residual Network family [20], and Inception-V2 and Inception-V3, as representatives of the Inception family [51]. They have 50, 101, 34, 48 layers respectively. These models represent a structural trend in CNN designs, in which, several layers are encapsulated into a generic module of a fixed structure—which we call *convolution module*—and a network is built by stacking many such modules together. Such CNN models are holding the state-of-the-art accuracy in many challenging deep learning tasks. The structures of these models are described in input Caffe Prototxt[6] files and converted to the multiplexing models by the CoCo-Tune compiler.

For preparation, we adapt the four CNN models trained on a general image dataset ImageNet [49] (ILSVRC 2012) to each of four specific image classification tasks with the domain-specific datasets, Flowers102 [45], CUB200 [53], Cars [32], and Dogs [30]. It gives us 16 trained full CNN models. The accuracy of the trained ResNets and Inceptions on the test datasets are listed in columns *Accuracy* in Table 2. The four datasets for CNN pruning are commonly used in fine-grained recognition [31, 12, 43, 25, 62], which is a typical usage scenario of CNN pruning. Table 2 reports the statistics of the four datasets, including the data size for training (*Train*), the data size for testing (*Test*), and the number of classes (*Classes*). For all experiments, network training is performed on the training sets while accuracy results are reported on the testing sets.

Baseline for Comparison In CNN pruning, the full CNN model to prune has typically been already trained on the datasets of interest. When filters in the CNN are pruned, a new model with fewer filters is created, which inherits the remaining parameters of the affected layers and the unaffected layers in the full model. The promising subspace consists of such models. The *baseline approach* trains these models as they are. Although there are prior studies on accelerating CNN pruning, what they propose are all various ways to reduce the configuration space to a promising subspace. To the best of our knowledge, when exploring the configurations in the promising subspace, they all use the *baseline approach*. As our method is the first for speeding

[6]We add to Prototxt a new construct "module" for specifying the boundaries of *convolution modules*.



(a) Flowers102 (b) Cars (c) ResNet-50 (d) Inception-V3

Figure 11: (a,b) Accuracies of pruned networks of ResNet-50 after training. The model size of full ResNet-50 is 25.6 million. (c,d) Accuracy curves of the *default* and *block-trained* networks on dataset CUB200; each network has 70% least important filters pruned at all convolution modules.

up the exploration of the promising space, we compare our results with those from the *baseline approach*.

We refer to a pruned network in the baseline approach a *default network* while the one initialized with pre-trained tuning blocks in our method a *block-trained network*.

Promising Subspace The 16 trained CNNs contain up to hundreds of convolutional layers. A typical practice is to use the same pruning rate for the convolutional layers in one *convolution module*. We adopt the same strategy. The importance of a filter is determined by its ℓ_1 norm as previous work [36] proposes. Following prior CNN pruning practice [36, 40], the top layer of a convolution module is kept unpruned; it helps ensure the dimension compatibility of the module.

There are many ways to select the promising subspace, i.e., the set of promising configurations worth evaluating. Previous works select configurations either manually [36, 40] or based on reinforcement learning with various rewards or algorithm design [21, 3]. As that is orthogonal to the focus of this work, to avoid bias from that factor, our experiment forms the promising spaces through random sampling [5] of the entire pruning space. A promising space contains 500 pruned networks, whose sizes follow a close-to-uniform distribution. In the experiments, the pruning rate for a layer can be one of $\Gamma = \{30\%, 50\%, 70\%\}$.

Objective of Pruning There are different pruning objectives including minimizing model size, computational cost, memory footprint or energy consumption. Even though an objective of pruning affects the choice of the best configuration, all objectives require the evaluation of the set of promising configurations. Our composability-based CNN pruning aims at accelerating the training of a set of pruned networks and thus can work with any objective of pruning.

For the demonstration purpose, we set the objective of pruning as finding the smallest network (min ModelSize) that meets a given accuracy threshold (Accuracy $<=$ *thr_acc*). We get a spectrum of *thr_acc* values by varying the accuracy drop rate α from that of the full model from -0.02 to 0.08. We include negative drop rates because it is possible that pruning makes the model more accurate.

All the experiments are performed with TensorFlow 1.3.0 on machines each equipped with a 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor, 32 GB of RAM and an NVIDIA K20X GPU with 6 GB of DDR5 memory. One network is trained on one GPU.

To measure the basic benefits from the composability-based method, these experiments use every convolution module in these networks as a tuning block. The extra benefits from hierarchical tuning block identification are reported later.

Figure 11 (a,b) show the final accuracies of all the 500 ResNet-50 variants trained with or with-

out leveraging composability on the Flower102 and CUB200 datasets. For reference, we also plot the accuracies of the well-trained full ResNet-50 on the two datasets. The block-trained network gives a clearly better final accuracy overall.

Table 3 reports the comparisons between the block-trained version and the default version, in both speeds and network sizes, at various levels of tolerable accuracy drop rates α (negative means higher accuracy than the large network gives). The results are collected when 1, 4, or 16 machines are used for concurrent training for both the baseline and our method (indicated by the "#nodes" column). The time of the block-trained version already takes the pre-training time of tuning blocks into account ("overhead" in Table 3 shows the percentage in overall time). For the objective of pruning, the exploration order CoCo-Tune adopts is to start from the smallest models and proceed to larger ones.

The results show that the composability-based method avoids up to 99.6% of trial configurations and reduces the evaluation time by up to 186X for ResNet-50; up to 96.7% reduction and 30X speedups for Inception-V3. The reduction of trial configurations is because the method improves the accuracy of the pruned networks as Figure 11 shows. As a result, the exploration meets a desirable configuration sooner. For instance, in Flower102 ($\alpha = 0$), the third smallest network can already reach the target accuracy in the block-trained version while the 297th network meets the target in the default version. This not only shortens the exploration time, but also yields more compact (up to 70% smaller) networks as the "model size" columns in Table 3 show. Another reason for the speedup is that the training of a block-trained network takes fewer iterations to reach its final accuracy level than the default version, as Figure 11 (c,d) show. Even when configurations are not reduced (e.g., Flower102, $\alpha = -1$), the block-trained exploration finishes sooner.

Table 4 shows the speedups by composability-based pruning with different subspace sizes. The speedups are higher as the number of configurations to explore increases. It is because the time for pre-training tuning blocks weights less as the total time increases and the reduction of configurations becomes more significant for a larger set. Another observation is that, when the number of configurations is only four, there is still a significant speedup in most cases. The block training time is the time spent on pre-training all the tuning block variants (48 for ResNet-50 and 27 for Inception-V3). The speedup could be higher if tuning block identifier is applied, as shown next.

Extra Benefits from Tuning Blocks Identification Hierarchical tuning block identifier balances the overhead of training tuning blocks and the time savings they bring to the fine-tuning of pruned networks. Table 5 reports the extra speedups brought when it is used.

For datasets Flowers102 and CUB200, we experiment with two types of collections of configurations with $N = 8$. The first type, "collection-1", is a randomly sampled collection as mentioned earlier, and the second type, "collection-2", is attained by setting one pruning rate for a sequence of convolution modules, similar to the prior work [36] to reduce module-wise meta-parameters. For each type, we repeat the experiments five times with a new collection created each time. Each tuning block identified from the first collection tends to contain only one convolution module due to the independence in choosing the pruning rate for each module. But the average number of tuning blocks is less than the total number of possible pruned convolution modules (41 versus 48 for ResNet-50 and 27 versus 33 for Inception-V3) because of the small collection size. The "collection-2" setting has tuning blocks that contain a sequence of convolution modules as they are set to use one pruning rate.

The extra speedups from the algorithm are substantial for both, but more on the "collection-2" setting for the opportunities that some larger popular tuning blocks have for benefiting the networks in that collection. Because some tuning blocks selected by the algorithm are a sequence

Table 3: Speedups and configuration savings by composability-based pruning (when 1, 4, or 16 machines are used for both baseline and composability-based methods as "#nodes" column indicates). Notations are at the table bottom.

Dataset	α	#nodes	ResNet-50									Inception-V3								
			thr_acc	#configs		time (h)		model size		speedup	overhead	thr_acc	#configs		time (h)		model size		speedup	overhead
				base	comp	base	comp	base	comp	(X)			base	comp	base	comp	base	comp	(X)	
Flowers102	-1%	1	0.983	500	500	2858.7	1912.7	100%	100%	1.5	0.4%	0.978	500	500	3018.8	2023.5	100%	100%	1.5	0.5%
		4		500	500	718.1	481.0			1.5	0.5%		500	500	756.7	508.1			1.5	0.7%
		16		500	500	184.9	125.5			1.5	1.8%		500	500	194.8	133.6			1.5	2.7%
	0%	1	0.973	297	3	1639.4	16.9	45.4%	29.3%	97.0	40.4%	0.968	244	10	1428.6	47.3	43.2%	32.4%	30.2	23.3%
		4		300	4	412.6	5.2			79.3	43.5%		244	12	358.2	13.9			25.8	26.4%
		16		304	16	103.3	4.7			22.0	48.3%		256	16	94.8	6.5			14.6	56.4%
	1%	1	0.963	6	1	31.0	8.3	29.6%	27.6%	3.7	82.8%	0.958	27	1	152.6	13.9	33.9%	31.0%	11.0	79.0%
		4		8	4	10.4	3.2			3.3	70.6%		28	4	39.6	5.8			6.8	63.3%
		16		16	16	5.2	2.9			1.8	78.3%		32	16	11.2	5.6			2.2	71.0%
CUB200	4%	1	0.739	323	2	1807.3	12.7	46.6%	28.5%	142.3	53.7%	0.720	74	3	420.2	21.9	41.4%	33.7%	19.2	49.8%
		4		324	4	454.0	3.1			146.5	74.4%		76	4	106.4	6.7			15.9	54.5%
		16		336	16	118.7	3.1			38.3	74.4%		80	16	27.6	6.0			4.6	60.6%
	5%	1	0.731	297	1	1654.7	8.9	45.4%	27.6%	185.9	77.1%	0.710	44	1	247.8	14.1	38.5%	31.5%	17.6	77.5%
		4		300	4	418.8	2.8			149.6	81.4%		44	4	61.7	5.4			11.4	67.6%
		16		304	16	105.5	2.7			39.1	83.7%		48	16	16.4	5.2			3.2	70.6%
	6%	1	0.724	154	1	840.1	8.3	38.0%	27.6%	101.2	82.6%	0.700	29	1	162.5	12.8	35.9%	31.0%	12.7	85.1%
		4		156	4	214.2	2.6			82.4	86.7%		32	4	44.5	5.3			8.4	68.7%
		16		160	16	53.8	2.5			21.5	89.7%		32	16	10.8	5.1			2.1	71.9%
Cars	-1%	1	0.830	500	100	2864.9	362.4	100%	35.7%	7.9	1.9%	0.811	271	20	1586.8	85.6	40.1%	33.5%	18.5	12.8%
		4		500	100	720.4	90.9			7.9	2.5%		272	20	398.1	22.4			17.8	16.3%
		16		500	112	185.3	27.1			6.8	8.4%		272	32	99.4	11.1			9.0	32.8%
	0%	1	0.822	332	11	1848.6	44.4	46.9%	30.4%	41.6	15.4%	0.801	84	3	480.3	21.8	36.9%	31.3%	22.0	50.2%
		4		332	12	461.4	12.1			38.1	18.8%		84	4	120.5	7.2			16.7	50.6%
		16		336	16	115.9	5.2			22.3	44.0%		96	16	33.8	6.7			5.0	54.7%
	1%	1	0.814	189	2	1026.4	12.8	40.4%	28.5%	80.2	53.4%	0.791	33	1	186.4	14.2	34.4%	31.0%	13.1	77.0%
		4		192	4	259.7	4.9			53.0	46.7%		36	4	50.7	6.8			7.5	54.0%
		16		192	6	65.5	4.1			16.0	55.7%		48	16	16.4	6.2			2.6	59.1%
Dogs	6%	1	0.799	500	123	2848.1	441.1	60.0%	36.9%	6.5	1.6%	0.776	416	201	2470.7	786.0	100%	47.9%	3.1	1.4%
		4		500	124	709.8	111.2			6.4	2.0%		416	204	618.2	199.3			3.1	1.8%
		16		500	128	178.0	28.3			6.3	8.1%		416	208	153.2	52.7			2.9	6.9%
	7%	1	0.791	434	70	2445.4	251.8	51.9%	34.2%	9.7	2.7%	0.766	311	129	1822.2	503.2	56.0%	41.4%	3.6	2.2%
		4		436	72	606.2	63.9			9.5	3.6%		312	132	456.1	128.0			3.6	2.8%
		16		448	80	149.3	18.0			8.3	12.7%		320	144	116.2	36.4			3.2	10.0%
	8%	1	0.782	297	11	1632.8	42.3	45.4%	30.4%	38.6	16.2%	0.756	201	82	1164.1	322.9	47.9%	39.0%	3.6	3.4%
		4		300	12	411.7	10.1			40.8	22.7%		204	84	294.8	83.1			3.5	4.4%
		16		304	16	102.4	3.2			32.0	71.6%		208	96	75.0	26.1			2.9	13.9%

thr_acc: accuracy corresponding to an accuracy drop rate α. *base*: baseline approach. *comp*: composability-based approach.
speedup: $Time_{base}/Time_{comp}$; overhead counted in $Time_{comp}$. *overhead*: block training time over the total time of *comp*.

Table 4: Speedups by composability-based pruning with different subspace sizes.

Dataset	alpha	subspace size	ResNet-50			Inception-V3		
			base time (h)	comp time (h)	speedup (X)	base time (h)	comp time (h)	speedup (X)
Flowers102	0%	4	22.7	13.4	1.7	20.3	16.8	1.2
		16	90.9	12.8	7.1	76.7	20.6	3.7
		64	364.8	21	17.4	224.7	25.4	8.8
		256	1460.7	13.5	108.2	809.4	40.7	19.9
CUB200	3%	4	22.8	11	2.1	23.6	26	0.9
		16	93.8	11.4	8.2	83.5	30	2.8
		64	369.6	15.5	23.8	292.5	29.2	10
		256	1472.9	20.7	71.2	1128.9	18.1	62.4

Table 5: Extra speedups brought by improved tuning block definitions.

Dataset	α	ResNet-50			Inception-V3		
		thr_acc	extra speedup (X)		thr_acc	extra speedup (X)	
			collection-1	collection-2		collection-1	collection-2
Flowers102	0%	0.973	1.05	0.98	0.968	1.12	1.14
	1%	0.963	1.19	1.21	0.958	1.08	1.15
	2%	0.953	1.06	1.14	0.949	1.15	1.23
CUB200	3%	0.747	1.04	1.08	0.737	1.00	1.03
	4%	0.739	1.04	1.20	0.729	1.08	1.09
	5%	0.731	1.11	1.15	0.722	1.03	1.04
geometric mean			1.08	1.12		1.08	1.11

of convolution modules that frequently appear in the collections, the total number of tuning blocks becomes smaller (e.g., 27 versus 23 on Inception-V3.)

3 Conclusions and Future Work

By drawing on the recent framework CoCoPIE, this article has introduced the concept of *compression-compilation co-design* and how it is materialized into a software framework CoCoPIE for real-time AI on mobile devices. The results produced by the two core components, CoCo-Gen and CoCo-Tune, provide strong evidences for the promise of the co-design principle. They indicate that it is possible to instill AI directly on existing commodity computing devices while offering even higher speeds and better energy efficiency than special AI accelerating hardware. The results open new opportunities for democratizing AI capability on end devices, while invalidating the common perception on the indispensability of special AI hardware for real-time AI on end devices. We believe that these results will prompt the industry to reexamine the directions and strategies on the pursue of mobile AI.

The promising progress opens up many potential directions for future development. We list two of them here.

The first is to expand the scope of the co-design based optimizations. So far, the principle of compression-compilation co-design has been focused on DNN models. Besides DNN, a real-world AI application often includes a lot of other parts, such as data collection, data preprocessing, the use of the DNN prediction in follow-up operations, and so on. Even though DNN may play an important role in the overall application, its optimizations may not be sufficient for the entire application to meet users' needs. So an important direction is on how to generalize the co-design principle into holistic optimizations to the entire AI-based applications.

The second is to increase the applicability of the co-design based optimizations. This direction relates with privacy and security. As they are two important factors in many AI model constructions and deployments, how to integrate them into the co-design process is worth pursuing. For instance, typically model pruning requires access to both the models and the training dataset, but there are scenarios where datasets may not be accessible to the model optimizer due to either privacy policies or artificial boundaries among corporations. Effective ways to circumvent these roadblocks could expand the applicability of the optimizations. This direction also relates with the way that the optimization framework takes to deliver its service (e.g., standalone software versus cloud-based service).

Following these directions, we envision the following roadmap of CoCoPIE development such that it can better serve real-world applications. First, we plan to develop it into a full-fledged DNN optimizing framework for mobile AI, including a versatile front end to support DNNs written in all the popular programming frameworks and an even broader coverage of DNN models and operations. Second, we plan to make CoCoPIE more flexible in delivering its service to fit the needs of various practical settings. Besides making it a standalone software framework, we envision a model of *DNN optimization as a service*, where, users may use CoCoPIE as a cloud-based service. It could not only lower the barrier for users to start adopting the service, but also avoid the hassles in setting up servers for DNN pruning. Moreover, based on our recent work [60], we plan to explore *data-free DNN pruning*, which could prune a DNN with generated rather than original training data, making the technique applicable to situations where training data are hard to access. Finally, after meeting the needs of most DNNs, we plan to extend CoCoPIE into an application-level optimizing framework by providing a holistic treatment to the efficiency issues in the entire AI applications as well as the whole (typically heterogeneous) deployment system.

Final Words on CoCoPIE: The authors are actively applying CoCoPIE to meet practical needs. CoCoPIE's technology can immediately enable real-time deep learning on billions of existing mobile devices, thus generating tremendous commercial values. To just name a few, CoCoPIE may enable great user experiences for streaming applications, such as YouTube, TikTok, and Snap, even under low-bandwidth situations: These applications can stream low-resolution videos to user devices, and CoCoPIE can upscale the videos to high-definition in real time. Similarly, video communication applications such as Zoom, Skype, and Webex, can utilize CoCoPIE's technology to deliver the best quality of service. In addition, CoCoPIE unlocks real-time deep learning applications that have never been possible before, such as enabling a mobile phone camera to show live videos in an artistic style. The authors welcome business ideas, suggestions, or any comments (contact: info@cocopie.ai).

References Cited

[1] Caffe Solver Prototxt. `https://github.com/BVLC/caffe/wiki/Solver-Prototxt`.

[2] Alibaba. Mnn, 2019.

[3] Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. N2n learning: Network to network compression via policy gradient reinforcement learning. *arXiv preprint arXiv:1709.06030*, 2017.

[4] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[5] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis 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)*, pages 578–594, 2018.

[8] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[10] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: a neural network synthesis tool based on a grow-and-prune paradigm. *arXiv preprint arXiv:1711.02017*, 2017.

[11] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *European conference on computer vision*, pages 184–199. Springer, 2014.

[12] Jianlong Fu, Heliang Zheng, and Tao Mei. Look closer to see better: Recurrent attention convolutional neural network for fine-grained image recognition. In *Conf. on Computer Vision and Pattern Recognition*, 2017.

[13] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[14] Google. Tensorflow lite, 2019.

[15] Google Cloud TPU. Google cloud tpu. `https://cloud.google.com/tpu/`, 2017.

[16] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. Wootz: A compiler-based framework for fast cnn pruning via composability. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, 2019.

[17] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems*, pages 1379–1387, 2016.

[18] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *FPGA*, pages 75–84, 2017.

[19] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

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

[21] Yihui He and Song Han. Adc: Automated deep compression and acceleration with reinforcement learning. *arXiv preprint arXiv:1802.03494*, 2018.

[22] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 1398–1406. IEEE, 2017.

[23] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[24] Holger H Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.

[25] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[26] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[27] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[28] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Let there be color! joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Trans. Graph.*, 35(4), July 2016.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[30] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. Novel dataset for fine-grained image categorization: Stanford dogs. In *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, volume 2, page 1, 2011.

[31] Jonathan Krause, Benjamin Sapp, Andrew Howard, Howard Zhou, Alexander Toshev, Tom Duerig, James Philbin, and Li Fei-Fei. The unreasonable effectiveness of noisy data for fine-grained recognition. In *European Conference on Computer Vision*, pages 301–320. Springer, 2016.

[32] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Computer Vision Workshops (ICCVW), 2013 IEEE International Conference on*, pages 554–561. IEEE, 2013.

[33] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[34] Vadim Lebedev and Victor Lempitsky. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2554–2564, 2016.

[35] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. *arXiv preprint arXiv:1707.09870*, 2017.

[36] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[37] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR)*, 2017.

[38] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[39] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[40] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342*, 2017.

[41] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. *AAAI*, 2020.

[42] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[43] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*, 2016.

[44] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 7:67–82, 1997.

[45] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*, pages 722–729. IEEE, 2008.

[46] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. *ASPLOS*, 2020.

[47] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7197–7205, 2017.

[48] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[49] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[50] N. Silberman and S. Guadarrama. Tensorflow-slim image classification model library. `https://github.com/tensorflow/models/tree/master/research/slim`, 2016.

[51] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[52] Radu Timofte, Eirikur Agustsson, Luc Van Gool, Ming-Hsuan Yang, and Lei Zhang. Ntire 2017 challenge on single image super-resolution: Methods and results. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 114–125, 2017.

[53] Peter Welinder, Steve Branson, Takeshi Mita, Catherine Wah, Florian Schroff, Serge Belongie, and Pietro Perona. Caltech-ucsd birds 200. 2010.

[54] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

[55] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[56] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[57] Daniel LK Yamins and James J DiCarlo. Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19(3):356, 2016.

[58] Daniel LK Yamins, Ha Hong, Charles F Cadieu, Ethan A Solomon, Darren Seibert, and James J DiCarlo. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the National Academy of Sciences*, 111(23):8619–8624, 2014.

[59] Jiahui Yu, Yuchen Fan, Jianchao Yang, Ning Xu, Zhaowen Wang, Xinchao Wang, and Thomas Huang. Wide activation for efficient and accurate image super-resolution. *arXiv preprint arXiv:1808.08718*, 2018.

[60] Zheng Zhan, Yifan Gong, Zhengang Li, Pu Zhao, Xiaolong Ma, Wei Niu, Xiaolin Xu, Bin Ren, Yanzhi Wang, and Xue Lin. Priv: A privacy-preserving deep neural network model compression framework. *arXiv preprint*, 2020.

[61] Hang Zhang and Kristin Dana. Multi-style generative network for real-time transfer. *arXiv preprint arXiv:1703.06953*, 2017.

[62] Bo Zhao, Xiao Wu, Jiashi Feng, Qiang Peng, and Shuicheng Yan. Diversified visual attention networks for fine-grained object classification. *IEEE Transactions on Multimedia*, 19(6):1245–1256, 2017.

[63] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations (ICLR)*, 2017.

[64] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.