Boosting the Performance of CNN Accelerators with Dynamic Fine-Grained Channel Gating

Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G. Edward Suh
Cornell University, Ithaca, NY
{wh399, yz882, cmd353, zhiruz, gs272}@cornell.edu

ABSTRACT

This paper proposes a new fine-grained dynamic pruning technique for CNN inference, named channel gating, and presents an accelerator architecture that can effectively exploit the dynamic sparsity. Intuitively, channel gating identifies the regions in the feature map of each CNN layer that contribute less to the classification result and turns off a subset of channels for computing the activations in these less important regions. Unlike static network pruning, which removes redundant weights or neurons prior to inference, channel gating exploits dynamic sparsity specific to each input at run time and in a structured manner. To maximize compute savings while minimizing accuracy loss, channel gating learns the gating thresholds together with weights automatically through training. Experimental results show that the proposed approach can significantly speed up state-of-the-art networks with a marginal accuracy loss, and enable a trade-off between performance and accuracy. This paper also shows that channel gating can be supported with a small set of extensions to a CNN accelerator, and implements a prototype for quantized ResNet-18 models. The accelerator shows an average speedup of 2.3× for ImageNet when the theoretical FLOP reduction is 2.8×, indicating that the hardware can effectively exploit the dynamic sparsity exposed by channel gating.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

neural networks, dynamic pruning, algorithm-hardware co-design

ACM Reference Format:
Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G. Edward Suh. 2019. Boosting the Performance of CNN Accelerators with Dynamic Fine-Grained Channel Gating. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3352460.3358283

MICRO-52, October 12–16, 2019, Columbus, OH, USA

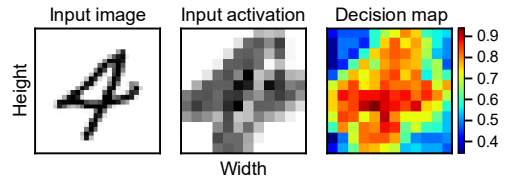
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358283

1 INTRODUCTION

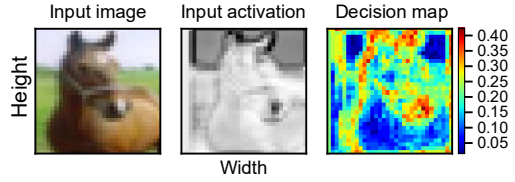
Convolutional neural networks (CNNs) have demonstrated human-level accuracy in many vision-related tasks and are being increasingly adopted for many applications, including real-time tasks such as autonomous driving and robotic manipulation. Unfortunately, state-of-the-art CNNs are highly compute-intensive, as they typically demand about 10^9 floating point operations (FLOPs) per inference [1]. In order to deploy CNNs in a much broader range of applications, especially in embedded and mobile settings [2], we need to reduce the high computational cost without noticeably sacrificing inference accuracy. In this paper, we propose a new dynamic pruning technique, named channel gating, which removes ineffectual computations specific to each input at run time, and present a hardware accelerator architecture to effectively exploit the dynamic sparsity introduced by channel gating.

Figure 1 illustrates the intuition behind channel gating by showing the heat maps of the normalized computational cost for two sample images. The "cool" colors on the decision maps indicate that computation for the region can substantially be pruned by channel gating. For these regions, only a small subset of input channels need to be used to produce output activations. Intuitively, these regions correspond to less important input features such as backgrounds. While several prior studies have proposed statically pruning ineffectual features and weights (i.e. those with small magnitude) [3–6], these static approaches prune networks for all inputs and cannot exploit dynamic input-specific characteristics. The static sparsity introduced by these existing pruning approaches reduces a constant amount of computation regardless of the input. While dynamic pruning approaches have been recently proposed [7–9], the previous approaches only focus on limited form of dynamic sparsity, in particular zeros from the ReLU activation. Channel gating aims to achieve more computation reduction and less accuracy loss by exploiting more general forms of dynamic sparsity and co-designing a pruning algorithm, a training method, and hardware architectures.

The key idea in channel gating is to identify ineffective receptive fields in input features and reduce the computation on these fields by gating a portion of the input channels. More specifically, to compute an output activation in a convolutional (or fully-connected) layer, we first perform a partial computation on a subset of input channels (i.e., $\mathbf{W_p} * \mathbf{x_p}$ in Figure 2). We found that these partial sums are strongly correlated with the final sums, and can serve as good indicators on which spatial locations are more important. The partial sum is then compared to a learnable threshold using a gate function, which generates a binary decision for each output activation. If the decision is 1, we continue computing the convolution on the rest of the channels (i.e., $\mathbf{x_r}$). Otherwise, we simply skip the remaining computation and feed the partial sum to the normalization and activation function. As CNN inference is mostly



(a) LeNet on the MNIST dataset



(b) ResNet-18 on the CIFAR-10 dataset

Figure 1: Input feature vs. decision map — Spatial locations with a larger number in the decision map are more important and need more computations. The decision map is obtained by averaging the gating decisions in a CGNet on CIFAR-10 for each spatial location over all output channels.

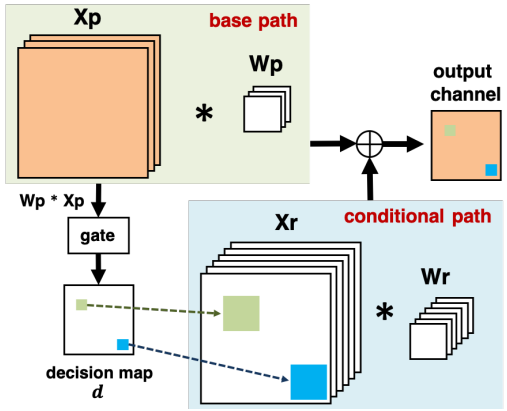


Figure 2: Illustration of channel gating — A subset of input channels are used to generate a decision map, which prunes away unnecessary computation in the rest of input channels.

compute-bound, our work currently focuses on minimizing the computational cost instead of reducing off-chip memory accesses.

In order to achieve high accuracy while reducing computation, the gating thresholds as well as the network weights for a CNN with channel gating (**CGNet**) are learned through a training process. The training process also allows channel gating to target a varying level of pruning and explore the trade-off between performance and accuracy. Experimental results show that channel gating can significantly reduce the amount of computation in state-of-the-art networks with a marginal impact on accuracy. For example, channel gating reduces the computation (in FLOPs) in ResNet-18 for ImageNet by 2× with a 1% accuracy loss or by 2.4× with a 1.5% accuracy loss. With knowledge distillation, channel gating can save even more computation (2.8×) with less accuracy loss (0.2%).

In order to translate the sparsity created by pruning into meaningful performance and energy benefits, a pruning scheme must be realized as an efficient software/hardware implementation. Unfortunately, existing fine-grained pruning schemes [3, 7, 8, 10] lead to irregular sparsity and often require significant changes to existing CNN accelerators. In contrast, channel gating is designed to provide structured sparsity by pruning contiguous input channels at a predetermined decision point. As a result, channel gating maintains the locality and the regularity in both computations and data accesses, and can be efficiently implemented by augmenting a conventional CNN accelerator.

In this paper, we introduce a new CNN accelerator architecture for channel gating and show that the structured sparsity can be effectively exploited in hardware to significantly improve performance and energy efficiency of CNNs. The proposed architecture extends a weight-stationary architecture to handle both dense and sparse computations in channel gating networks. A major challenge in accelerating the sparse computation lies in efficiently loading the convolutional windows in feature maps. The proposed architecture uses a new banking scheme that allows parallel accesses to activations in arbitrary convolutional windows to address this challenge. The architecture also exploits different loop tiling and reordering strategies for dense and sparse computations to maximize the processing unit utilization and the overall throughput. Moreover, the proposed architecture can effectively run both regular CNNs without pruning and channel gating networks.

To evaluate the channel gating accelerator, we implemented RTL prototypes of the 8-bit quantized ResNet-18 with and without channel gating, targeting a 28nm TSMC standard cell library. These ASIC implementations demonstrate that supporting channel gating only requires small hardware changes with moderate area and power overhead. Our results also show that the sparsity introduced by channel gating can be efficiently exploited to speed up CNN inference; the actual speedup on the accelerator is close to the theoretical computation reduction. For example, the CGNet accelerator achieves a 2.3× average speedup over the baseline accelerator for ImageNet, when the channel gating reduces the computation (FLOP) by 2.8×.

This paper makes the following major contributions:

- We introduce a novel dynamic fine-grained pruning approach that can significantly reduce the computation cost of a CNN by selectively pruning computation for each output activation at run time.
- We propose a unified accelerator architecture that can efficiently execute both regular CNNs and channel-gating networks (CGNets).
- We build an accelerator prototype in RTL and experimentally demonstrate the benefits of channel gating on ResNet-18 for CIFAR-10 and ImageNet.

2 CNNS WITH CHANNEL GATING

This section introduces channel gating neural network (CGNet). We first describe CGNet with a single neuron, then generalize the idea and present the full structure of a convolutional layer equipped with channel gating. Although we present CGNet within the scope of convolutional neural networks, CGNet can also be extended to multi-layer perceptrons.

Figure 3 shows a neuron taking $\mathbf{x}$ as an input where $\mathbf{x} \in \mathbb{R}^c$. These c channels are split into two groups, where one group contains the first p channels and the other has the remaining r channels.

Table 1: Structural parameters in CGNet.

Parameter	Description
χ	Fraction of input channels in the base path.
k_l	Filter size for the l'th conv. layer.
c_l	Number of output channels in the l'th conv. layer.
h_l	Height of the output feature in the l'th conv. layer.
w_l	Width of the output feature in the l'th conv. layer.

Based on the dot product between the first p channels and the corresponding weights, the gate turns the rest of the channels on or off. If the channels are off, they do not contribute to the output of the neuron, and importantly their dot product with corresponding weights does not need to be computed. Thus, the amount of computation (depending on how often the gate is off) can be reduced by channel gating.

2.1 CGNet Block

In practice, CNNs have many neurons which are organized into tensors. Without loss of generality, let $\mathbf{x}_l$, $\mathbf{y}_l$, $\mathbf{W}_l$ be the input features, output features, and weights of layer l, respectively, where $\mathbf{x}_l \in \mathbb{R}^{c_{l-1}\times w_{l-1}\times h_{l-1}}$, $\mathbf{y}_l \in \mathbb{R}^{c_l\times w_l\times h_l}$, and $\mathbf{W}_l \in \mathbb{R}^{c_l\times c_{l-1}\times k_l\times k_l}$. Table 1 summarizes the parameters. A typical CNN block includes convolution ($*$), batch normalization (BN) [11], and an activation function (f). The output feature can be written as $\mathbf{y} = f(\text{BN}(\mathbf{W} * \mathbf{x}))$[1].

To apply channel gating, we first split the input features and weights statically along the channel dimension into two tensors where $\mathbf{x} = [\mathbf{x_p}, \mathbf{x_r}]$ and $\mathbf{W} = [\mathbf{W_p}, \mathbf{W_r}]$. For $\chi \in [0, 1]$, $\mathbf{x_p}$ consists of χ fraction of the input channels while the rest of the channels form $\mathbf{x_r}$, where $\mathbf{x_p} \in \mathbb{R}^{\chi c_{l-1}\times w_{l-1}\times h_{l-1}}$ and $\mathbf{x_r} \in \mathbb{R}^{(1-\chi)c_{l-1}\times w_{l-1}\times h_{l-1}}$. Similarly, let $\mathbf{W_p}$ and $\mathbf{W_r}$ be the weights associated with $\mathbf{x_p}$ and $\mathbf{x_r}$ where $\mathbf{W_p} \in \mathbb{R}^{\chi c_{l-1}\times c_l\times k_l\times k_l}$ and $\mathbf{W_r} \in \mathbb{R}^{(1-\chi)c_{l-1}\times c_l\times k_l\times k_l}$. This decomposition means that $\mathbf{W} * \mathbf{x} = \mathbf{W_p} * \mathbf{x_p} + \mathbf{W_r} * \mathbf{x_r}$. Then, the partial sum $\mathbf{W_p} * \mathbf{x_p}$ is fed into the gate (s) to generate the binary decision tensor ($\mathbf{d} \in \mathbb{R}^{c_l\times w_l\times h_l}$). A binary value $\mathbf{0}$ in $\mathbf{d}_{i,j,k}$ indicates gating the rest of the channels for the output activation with indices (i, j, k).

The inference of CGNet is divided into two possible paths, as shown in Figure 3, with different frequency of execution. We refer to the path which is always taken as the **base path** (colored in grey) and the other path as the **conditional path** given that it may be gated for some activations. The final output is the activation-level combination of the outputs from both the base and conditional paths, which can be written as follows (i, j, k are the indices of a component in a tensor of rank three):

$$\widetilde{\mathbf{y}}_{i,j,k} = \begin{cases} f(\text{BN}(\mathbf{W_p} * \mathbf{x_p})_{i,j,k}), \text{if } \text{s}(\mathbf{W_p} * \mathbf{x_p})_{i,j,k} = 0 \\ f(\text{BN}(\mathbf{W_p} * \mathbf{x_p} + \mathbf{W_r} * \mathbf{x_r})_{i,j,k}), \text{otherwise} \end{cases} \quad (1)$$

We propose CGNet based on the observation that a partial sum (PSUM) is a good predictor for the final sum (FSUM). We reduce computation on a subset of output activations by skipping $\mathbf{W_r} * \mathbf{x_r}$ and approximating these activations with their PSUMs. We use the Pearson correlation coefficient to measure the linear correlation between the PSUM and FSUM with different χ values. The average

[1]The bias term is ignored because of batch normalization.

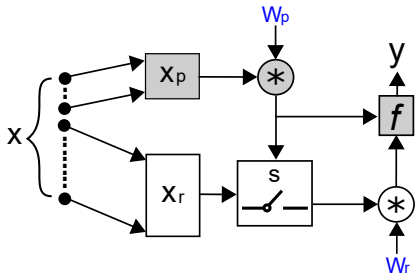


Figure 3: CGNet with a single neuron.

Pearson correlation coefficient of 20 convolutional layers over 1000 random samples in ResNet-18 equals to 0.56, 0.72, and 0.86 when χ is $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$, respectively. The results suggest that the PSUM and the FSUM of convolutional layers are moderately correlated even when $\chi = \frac{1}{8}$.

2.2 Gate Function

To minimize the computational cost, the gate function should only allow a small fraction of the output activations to take the conditional path. We design the gate based on the ReLU activation [12] in the baseline network. As the partial sum with small magnitude is less important than the one with higher magnitude, the gate turns off the remaining channels for the output activation whose partial sum is small. For example, activation with a large negative partial sum is likely to be zeroed out by ReLU and has no contribution to the output. We introduce a learnable threshold per output channel ($\Delta \in \mathbb{R}^{c_l}$) and broadcast it to $\mathbb{R}^{c_l\times w_l\times h_l}$. The gate function (s) is defined using the Heaviside step function ($\theta : \mathbb{R}^{c\times w\times h} \mapsto \mathbb{R}^{c\times w\times h}$), which only requires a simple comparison to implement.

$$\text{s}(\mathbf{x}, \Delta) = \theta(\mathbf{x} - \Delta) = \begin{cases} 1, & \text{if } \mathbf{x} \geq \Delta \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Let τ be the fraction of activations taking the conditional path. To find Δ which satisfies $P(\mathbf{W_p} * \mathbf{x_p} < \Delta) = \tau$, we use batch normalization without scale and shift to normalize $\mathbf{W_p} * \mathbf{x_p}$.

The batch normalization normalizes the input features using the moving mean ($\mathbf{E}[x]$) and variance ($\text{Var}(x)$) during inference. To eliminate extra parameters and computational cost, we merge the batch normalization on the PSUMs with the gate function. The merged gate contains c_l thresholds and performs $w_l \cdot h_l \cdot c_l$ pointwise comparisons between the PSUMs and thresholds.

2.3 Training CGNet

Making CGNets end-to-end learnable is essential to minimize the accuracy degradation of pruning. We first reformulate the output of the channel gating block $\widetilde{y}$ without the if-else expression in Equation (1). Instead, we subtract the output of the gate (binary mask) from an all-one tensor of rank three ($\mathbf{J} \in \mathbb{R}^{c_l\times w_l\times h_l}$) to express the else condition and combine the two cases with an addition which makes all the operators differentiable except the gate function. In addition, we discuss three important mechanisms to reduce computation with minimal accuracy loss: **(1)** approximating the non-differentiable gate function; **(2)** inducing sparsity in the conditional path; **(3)** selecting a subset of channels as the input to the base path.

Table 2: Design parameters for the CGNet accelerator.

Parameter	Explanation
M	Number of multipliers per PE.
P	Number of PEs per PE row.
R	Number of PE rows in the accelerator.

Non-differentiable gate function. As the Heaviside step function is not differentiable, the gradients toward batch normalized partial sum ($\text{BN}(\mathbf{W_p} * \mathbf{x_p})$) and the threshold ($\Delta$) cannot be computed directly. We propose to approximate the gate with a smooth function which is differentiable with respect to $\mathbf{x}$ and Δ during the backward propagation. Here, we propose to use

$$\mathbf{s}(\mathbf{x}, \Delta) = \frac{1}{1 + e^{\epsilon \cdot (\mathbf{x} - \Delta)}} \tag{3}$$

to approximate the gate only for the backward propagation when the ReLU activation is used. ϵ is a hyperparameter which can be tuned to adjust the difference between the approximated function and the gate. We implement a custom operator in MxNet [13] which takes the batch normalized partial sum as the input and generate the gate decisions.

Sparsity-inducing mechanism. The batch normalized input to the gate follows the standard normal distribution, and therefore the fraction of the FLOPs pruned increases monotonically with Δ. As a result, reducing computation cost is equivalent to having a larger Δ. To motivate reducing computational cost during training, we set a target Δ value, denoted as T; for the entire network, we add the squared difference between channel-specific Δ_l and T (i.e., $\lambda \sum_l (T - \Delta_l)^2$) into the loss function, where λ is a scaling factor. T can be tuned to achieve different degrees of FLOP reduction. Since the gate function is differentiable, per-channel thresholds are optimized by SGD to minimize accuracy loss for a given T. In other words, the actual gating threshold values are automatically learned for each channel in each layer.

Channel selection. We do not manually select channels for the base path; instead, we simply use the first χ fraction of channels as $\mathbf{x_p}$. In our experiments, we observed that the fixed channel selection scheme works well as the network weights are learned through training, considering which channels are in the base path. To further improve the accuracy of CGNets, channel grouping and shuffling can be used to "equalize" the importance of each channel [14]. The channel grouping and shuffling improve the accuracy by 1% over the fixed channel selection scheme for ImageNet.

3 CGNET ACCELERATOR ARCHITECTURE

In this section, we propose a unified architecture which can accelerate both ordinary CNNs and channel gating networks. Our proposed architecture extends a weight-stationary architecture to handle both base and conditional paths without significant changes to the baseline architecture that executes regular CNNs. Hereafter, we refer to the computation in the base path as *dense convolution*, and the conditional path as *sampled feature convolution*.

3.1 Architecture Overview

The overall system architecture is shown in Figure 5(a). The accelerator communicates with the CPU through an SoC bus and off-chip DRAM through two DDR channels. The host CPU issues commands containing layer descriptions to the accelerator. The weights of a channel gating network model are stored in the off-chip DRAM. During execution, the weights of a layer are prefetched to on-chip global weight buffers. Weights are moved to local weight buffers near the PE array to maximize data locality and reuse. When possible, the output feature maps of intermediate layers are buffered on-chip to minimize off-chip data transfers. The on-chip weights and feature maps are split and stored in the dense and sparse buffers. Double buffering of weights is applied to overlap computation and off-chip transfers. A batch normalization layer, an activation function, and a pooling layer, if present, are combined with the previous convolutional or fully-connected layer.

Figure 5(b) depicts the proposed dense-sparse accelerator architecture. We adopt the widely-used weight stationary architecture to process the base path. As the base path can be considered as an unpruned CNN, this architecture for dense convolution represents the baseline. In that sense, the proposed dense-sparse architecture is capable of accelerating both regular CNNs and CGNets on a single hardware platform. The baseline accelerator exploits the parallelism in output channel (c_l), input channel (c_{l-1}), and spatial ($k_l \times k_l$ window) dimensions. There are three main components in the baseline architecture — convolution engines, a data fetching unit, and a feature map store unit, which are colored in blue. Table 2 lists the key parameters of the dense-sparse architecture. The convolution engine consists of $R \times P$ processing elements (PEs). Each PE contains M multipliers. For networks with only convolutional filters of size k, we should pick $P = k^2$ to maximize PE utilization. If the network contains convolutional filters of different sizes, techniques similar to [15] can be used to choose an optimal value of P. Each PE computes a 1-by-1 convolution of M input channels per clock cycle. Assuming $P = k^2$, a k-by-k convolution of M input channels is mapped to a row of PEs in the baseline architecture, and the whole PE array computes R output activations on the same spatial location in R output channels. The value of M should divide the minimum number of input channels in all layers of the network.

The feature maps are stored in on-chip feature buffers to minimize the number of off-chip memory accesses. As a result, the only off-chip memory accesses come from fetching weights. In order to maximize the energy efficiency of the accelerator, weight-stationary architecture is adopted where the weight kernels are read exactly once from the off-chip memory. To leverage the parallelism in the output channel dimension, the PE rows apply different weight kernels on the same input feature to generate R output channels in parallel. Thus, the data fetching unit broadcasts the same input to all window buffers attached to each PE row to exploit feature reuse. Similar to several other CNN accelerators [16–18], we add a specialized line buffer between the window buffers and the global feature buffer to further exploit the data locality and reuse of the feature maps.

An adder tree accumulates the partial sums from each PE into a per-PE-row PSUM buffer. Once the entire feature map of an output channel is obtained, the feature map store unit writes the output channel back to the global feature buffer. The baseline architecture can fully utilize all PEs across different layers and achieve near-optimal throughput with respect to the number of PEs.

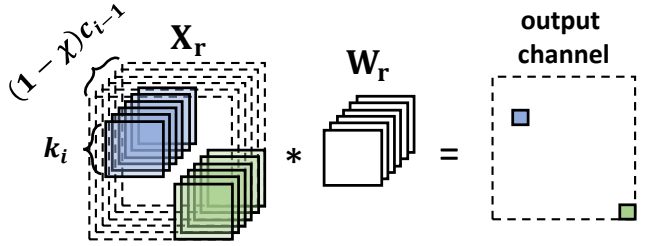


Figure 4: Sampled feature convolution.

3.2 Challenges of Accelerating Sparse Computation in Conditional Paths

Unlike static fine-grained pruning approaches such as deep compression [19, 20], channel gating is more friendly to hardware acceleration as it preserves regular parallelism in the input channel dimension (c_{l-1}) and the spatial dimension (per $k_l \times k_l$ filter window). As illustrated in Figure 4, for each output pixel, a CGNet either entirely skips the conditional path or performs sampled feature convolution on all the remaining $(1-\chi)c_{l-1}$ input channels. Hence the the induced sparsity in CGNet is more structured.

Nonetheless, there exists dynamic behavior across different output channels (c_l) during sampled feature convolution, since the number and the spatial locations of output activations taking the conditional path are determined at run time. Computing multiple output channels in parallel may require duplicating a feature map buffer because different spatial locations of the feature can be accessed concurrently. Instead, we compute output channels sequentially and explore the parallelism in other dimensions to avoid the overhead.

Channel gating also complicates the fetching of the feature map as it changes the access pattern from a fixed stride access to irregular access to the k_l by k_l receptive fields within the 2-D feature map (shown in Figure 2). We exploit a specialized banking scheme to provide enough memory bandwidth for accessing the feature.

Moreover, the work imbalance in different output channels may limit the benefits of exploiting the parallelism in the output channel dimension. At the micro-architecture level, since only a small fraction of activations in each output channel take the conditional path, filling and draining the sampled feature convolution pipeline can result in nontrivial overhead if individual output channels are processed separately. We mitigate this pipelining overhead by executing the conditional path of multiple output channels all together.

3.3 Architectural Support for Sampled Feature Convolution

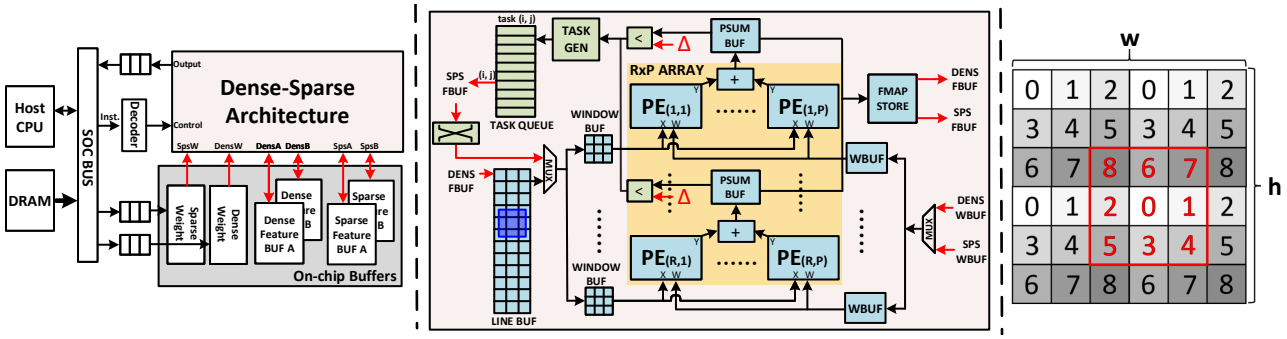
We use the baseline architecture for executing the base path of a channel gating network. To compute sampled feature convolution in the conditional path, we need to extend the baseline architecture by introducing the hardware modules colored in green in Figure 5(b). The gate function only requires a simple comparator that compares the partial sum with a learned threshold to generate the binary decision of whether the conditional path needs to computed for a specific output activation. The resulting decisions are saved in a fixed-length task queue, whose size is determined by the largest width and height of the output feature maps.

For sampled feature convolutions in the conditional path, we reuse the array of $R \times P$ PEs in the baseline architecture, but only leverage the parallelisms within one output channel at a time in order to avoid dealing with unpredictable and unbalanced work across output channels. For the conditional paths, each PE still computes a 1-by-1 convolution of M input channels per clock cycle, and a k-by-k convolution of the M input channels is mapped to each PE row. However, instead of exploiting the parallelism across multiple output channels, the R rows of PEs process different input channels in parallel. Therefore, the entire PE array computes a partial sum for one output activation over $R \times M$ input channels. The result is added to the partial sum saved in the PSUM buffer to obtain the final output activation.

In the sampled feature convolution, the PE array needs to process output activations with an irregular pattern because only a small fraction of activations use the conditional path. This irregular processing pattern leads to irregular access patterns for input feature maps. The irregular feature map accesses imply that the convolution windows do not necessarily overlap over consecutive cycles, which makes the line buffer less effective. For this reason, the data fetching unit loads the input feature into the window buffers directly for the conditional path. In order to load an arbitrary convolution window into the PE row with the corresponding weights, the accelerator includes a crossbar that connects the sparse feature map input to the PE rows.

The irregular access patterns also pose a challenge in reading feature maps from on-chip memory. For example, in order to fully utilize the PE array, the sampled feature convolution requires fetching k_l^2 words in an arbitrary $k_l \times k_l$ window of the input feature map per cycle. If the entire feature map is stored in a single on-chip memory bank, fetching input words for the sampled feature convolution would consume multiple clock cycles and starve the convolution engine. To enable single-cycle access to an arbitrary $k_l \times k_l$ window, we propose a specialized banking scheme in spatial dimensions. Specifically, we partition the feature map in the fashion that every neighboring activation is stored in a different bank. As a result, the sparse feature maps ($\mathbf{x_r}$) are partitioned into k_l^2 banks in the spatial dimension. Figure 5(c) shows an example of the proposed feature map layout, assuming nine banks are used. A 5×5 feature map is partitioned into nine banks in the spatial ($w \times h$) dimensions to provide enough throughput for reading nine activations in one cycle. Every neighboring activation is stored into a different on-chip memory bank so that the nine input words in any 3×3 window can be accessed in parallel. The numbers in the feature map represent the ID of the memory bank where each activation is stored. Although the total capacity of the on-chip memory remains the same as the baseline architecture, the larger number of memory banks increases the area of the design.

In our prototype implementation, we use on-chip memory blocks with the width of 64 bits. To allow multiple output channels to be accessed in parallel, we store activations along the channel dimension into a single memory word. In addition, because different PE rows process different input channels in a cycle for the sampled feature convolution, we also partition the global feature buffer along the channel dimension to access different chunks of input channels in parallel.



(a) Overall architecture (b) Dense-sparse accelerator details (c) Memory layout of the feature map, assuming nine banks

Figure 5: Channel gating accelerator architecture.

3.4 Design Space Exploration of the Dense-Sparse Architecture

We explore the design space of the dense-sparse architecture by parallelizing the output and input channels of the dense convolution and the input channels of sampled feature convolution differently. The objective is to maximize the performance as well as data reuse of the architecture. Let w_l, c_l, k_l, σ be the width of the feature map[2], the filter size, the number of channels of layer l, and the average fraction of activations taking the conditional path, respectively. The latency can be formulated as:

$$Latency = \sum_{l=1}^{N} \Big(\underbrace{\frac{\chi c_{l-1}}{M}\frac{c_l}{R}w_l^2}_{\text{Dense Convolution}} + \underbrace{\frac{(1-\chi)c_{l-1}}{MR}c_l\sigma w_l^2}_{\text{Sampled Feature Convolution}} \Big) \quad (4)$$

As discussed in Section 3.2, the overhead of filling and draining sampled feature convolution pipeline can be significant. Let P_D be the pipeline depth of the sampled feature convolution, the overhead equals $\sum_{l=1}^{N}\frac{c_l}{R}P_D$, where $\frac{c_l}{R}$ calculates the number of times that the pipeline is filled and drained within a layer.

As we only consider the weight-stationary architecture, the number of memory accesses to the feature map is used to evaluate the benefit of data reuse at different design points. The number of memory accesses to the feature map can be characterized as:

$$Mem = \sum_{l=1}^{N} \Big(\underbrace{\chi c_{l-1} w_{l-1}^2 \frac{c_l}{R}}_{\text{Dense Convolution}} + \underbrace{(1-\chi)c_{l-1}\sigma w_l^2 c_l}_{\text{Sampled Feature Convolution}} \Big) \quad (5)$$

We impose two resource constraints for the design exploration — the maximum number of multipliers and the total size of the on-chip buffers. The number of multipliers equals to $R \cdot M \cdot k^2$. Assuming that each weight and activation are 8 bits, each partial sum is 16 bits, and the spatial index (i, j) of each task is 8 bits each, the total size of the on-chip buffers (B) is equal to:

$$B = \max_{l\in\{1\ldots N\}} \Big(\underbrace{2(k_l^2 c_{l-1}R}_{\text{Weight Buf.}} + \underbrace{w_{l-1}^2 c_{l-1}}_{\text{Feature Buf.}} + \underbrace{w_l^2 R}_{\text{PSUM Buf.}} + \underbrace{w_l^2 R}_{\text{Task Queue}} \Big) \quad (6)$$

[2]We assume the shape of the 2-D feature map is a square where $h_l = w_l$.

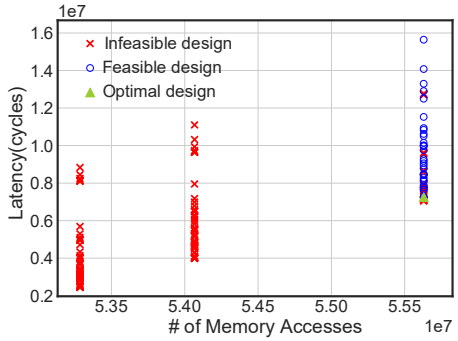


Figure 6: Design space exploration of the CGNet accelerator.

We assume $\sigma = 0.3$, ResNet-18 architecture, and ImageNet dataset for the following exploration. We set the maximum number of multipliers to 600 and storage size to 18 MB based on the available resources of the Xilinx Zynq device (xc7z045ffg900-2). A subset of the design points we explored are depicted in Figure 6. The best parameter choices are $M = 8$, $P = 9$, and $R = 8$ for both baseline and dense-sparse architectures. Another stand-alone convolution engine with $M = 8$, $P = 1$, and $R = 8$ is used to handle the 1×1 convolutional layers in the ResNet models.

4 EVALUATION

4.1 Algorithm Evaluation

We evaluate CGNets on the CIFAR-10 [21] and ImageNet (ILSVRC 2012) [22] datasets and demonstrate that channel gating can significantly reduce the compute with minimal impact on accuracy.

4.1.1 Network Architectures. We apply channel gating on a modified ResNet-18[3] and VGG-16 on CIFAR-10. The modified ResNet-18 baseline has 5.40% errors on CIFAR-10 with 578 MFLOPs. For ImageNet, we use AlexNet and ResNet-18 as the baseline networks. The experiments use a uniform χ and T for all layers in CGNets.

[3]The ResNet-18 variant architecture is similar to the ResNet-18 architecture on ImageNet while using 3×3 filter in the first convolutional layer and having ten outputs from the last fully-connected layer.

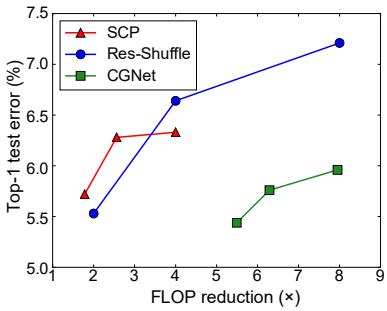


Figure 7: SCP, Res-Shuffle, and CGNet on CIFAR-10.

Table 3: The accuracy and FLOP reduction for CIFAR-10 — CGNet A and B represent CGNet models with different parameter values for the target threshold (T), χ, and λ. χ is $\frac{1}{8}$ for all CGNet models. λ is $5e^{-4}$ and $1e^{-4}$ for ResNet and VGG, respectively.

Network	Model	Target Threshold	Test Error (%)	FLOP Reduction
ResNet	Baseline	/	5.40	1×
	Res-Shuffle	/	5.53	2×
	CGNet-A	2	5.44	5.49×
	CGNet-B	3	5.76	6.29×
VGG	Baseline	/	7.20	1×
	CGNet-A	2	7.12	3.41×
	CGNet-B	3	7.59	5.1×

4.1.2 Accuracy and Computational Costs (FLOPs). In Figure 7, we compare CGNet on ResNet-18 with static channel pruning (SCP)[4] and ResNet with grouped convolution and channel shuffling (Res-Shuffle), which are the main techniques proposed in ShuffleNet [23]. Grouped convolution with shuffling represents the state-of-the-art static pruning approach. CGNet achieves larger FLOP savings with better accuracy compared to SCP and Res-Shuffle. Table 3 shows the trade-off between accuracy and FLOP reduction when CGNet models are used for CIFAR-10. CGNet can reduce the computation by 2.7 - 5.5× with almost no accuracy degradation on three state-of-the-art architectures. CGNet-A reduces FLOPs by **2.7×** with **0.1%** higher top-1 accuracy compared to Res-Shuffle on CIFAR-10. Note that CGNet-A and CGNet-B represent CGNet models with different the target threshold (T), χ, and λ. While we use the same name in multiple table entries, they all represent different CGNet models for each baseline network and dataset.

Table 4(a) and 4(b) compare CGNet to prior arts [5, 9, 24] on AlexNet and ResNet-18 for ImageNet. The results show that CGNet outperforms all other pruning techniques, offering better accuracy *and* higher FLOP reduction. Table 4(a) shows the comparison with SnaPEA [9] based on the results on AlexNet. The top-5 errors of the baseline AlexNet reported for CGNet and SnaPEA are 19.4% and 27.4%, respectively [5]. CGNet achieves a **5.5**× speedup with a **22%** top-5 error (2.6% accuracy drop) whereas SnaPEA reports a 2.1× speedup with a 30.4% top-5 error (3% accuracy drop).

[4]Static channel pruning removes a fixed fraction of channels statically and trains the pruned model from scratch.

[5]SnaPEA only provides the top-5 error. To compare with SnaPEA, we report the top-5 error as well. The top-1 errors of CGNet-A and CGNet-B for AlexNet are 57.1% and 54.8%, respectively.

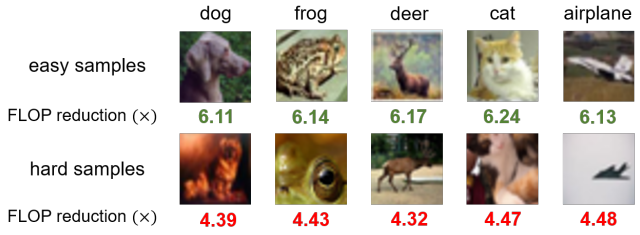


Figure 8: Samples with different FLOP reductions in CIFAR-10 — Easy samples refer to images with high FLOP reduction, while hard samples refer to images with low FLOP reduction.

Table 4: Comparisons of accuracy and FLOP reduction of the pruned models for ImageNet — CGNet A and B represent CGNet models with different parameter values for the target threshold (T), χ, and λ. χ is $\frac{1}{8}$ for all CGNet models. (T, λ) is $(3, 1e^{-5})$ and $(3, 5e^{-5})$ for CGNet-A and -B on AlexNet, respectively. (T, λ) is $(1, 5e^{-5})$ and $(1, 7e^{-5})$ for CGNet-A and -B on ResNet, respectively.

Model	Top-5 Accu. Drop (%)	FLOP Reduction	Top-5 Error Baseline (%)
AlexNet (Caffe [27])	/	1×	19.8
AlexNet-baseline	/	1×	19.4
SnaPEA-baseline [9]	/	1×	27.4
SnaPEA [9]	3.0	2.11×	30.4
CGNet-A	**0.6**	**2.65×**	19.4
CGNet-B	**2.6**	**5.50×**	19.4

(a) AlexNet

Model	Top-1 Accu. Drop (%)	FLOP Reduction	Top-1 Error Baseline (%)
Network Slimming [5]	1.8	1.39×	31.0
Discrimination aware Pruning [25]	2.3	1.85×	30.4
Feature Boosting and Suppression [24]	2.5	1.98×	29.3
Res-Shuffle	2.0	1.93×	30.8
CGNet-A	**1.0**	**2.03×**	30.8
CGNet-B	**1.5**	**2.36×**	30.8

(b) ResNet-18

Table 4(b) compares CGNet with other pruning techniques using ResNet-18 for ImageNet. Res-Shuffle outperforms network slimming [5] and discrimination-aware channel pruning [25] on ResNet. Feature Boosting and Suppression (FBS) [24] is a channel-level dynamic pruning approach which achieves higher FLOP saving than static pruning approaches. Channel gating is much simpler than FBS, yet achieves **1.6%** less accuracy drop and slightly higher FLOP reduction (CGNet-A). While not shown in the table for brevity, applying knowledge distillation [26] on CGNet achieves **2.82**× FLOP saving with only **0.2%** accuracy loss as discussed in [14].

In Figure 8, we also show the test samples with the max. and min. FLOP reduction for five categories. There exists more than 30% difference in FLOP reduction among these samples which demonstrates that CGNet can prune adaptively for different samples.

Table 5: The test error of quantized CGNets on CIFAR-10 and ImageNet — CGNet-A models on CIFAR-10 and ImageNet refer to the CGNet-A in Table 3 and 4(b).

Dataset	Model	Weight	Activation	Test Error
CIFAR-10	CGNet-A	float-32	float-32	5.44%
		fixed-8	fixed-8	5.69%
		fixed-4	fixed-8	5.73%
ImageNet	CGNet-A	float-32	float-32	31.7%
		fixed-8	fixed-8	31.9%

4.1.3 Quantization. To further improve the energy efficiency and performance of our accelerator, We follow the widely used quantization configuration which quantizes both the weights and activations to 8 bits to avoid significant accuracy drop.

For activations, we adopt the PACT quantization scheme which introduces a layer-wise trainable clipping threshold (α) [28]. PACT first clips the activations to be in the range of $[0, \alpha]$ and then perform a linear quantization within that range. The quantized activation (x_q) can be expressed as

$$x_q = \lfloor \frac{|x| - |x - \alpha| + \alpha}{2} \cdot \frac{2^N - 1}{\alpha} \rceil \cdot \frac{\alpha}{2^N - 1} \tag{7}$$

where $\lfloor \rceil$ means rounding to the nearest integer function and N is the number of quantiztion bits. PACT also adds an $L2$ regularization on α which minimizes the range of the quantization and therefore improving the resolution. For weights, as $L2$ regularization on weights are adopted to prevent the model from overfitting, we directly apply a linear quantization between the range of $[-\max|x|, \max|x|]$. 8-bit quantized channel gating networks for CIFAR-11 and ImageNet have 0.3% and 0.2% accuracy degradation compared to their floating-point counterparts, respectively.

4.2 Hardware Evaluation

To evaluate the performance improvement, energy saving, and hardware overhead of applying channel gating, we implemented a hardware prototype targeting a TSMC 28nm (0.9V, 25°C, Standard Threshold Voltage Transistors) standard cell library. The SRAMs in the design are generated using the ARM SRAM compiler. We apply channel gating to two ResNet-18 models for CIFAR-10 and ImageNet. The ResNet models with channel gating achieve 5.5× and 2.8× FLOP reduction on average over a batch of 64 images compared to the baseline ResNet models on CIFAR-10 and ImageNet, respectively. In the rest of this section, we refer to the dense-sparse architecture proposed in Section 3.3 as CGNet-xcel, and compare it with the baseline architecture described in Section 3.1.

4.2.1 Methodology. Figure 9 shows the flow that we use to implement both the baseline and CGNet accelerators. With the RTL generated by HLS, we first run through an FPGA synthesis flow to obtain resource utilization after place and route, which is useful for identifying key resource overheads. After generating the RTL simulation trace, the RTL source files are sent to Synopsys Design Compiler (DC) together with the standard cell library, SRAM information, and the statistics of signal switching activities[6]. We can obtain area and timing estimations, as well as the gate-level netlist from Synopsys DC. The gate-level netlist and the RTL signal trace are used by Synopsys PrimeTime (PT) for power analysis.

[6]In the Switching Activity Interchange format (SAIF) format.

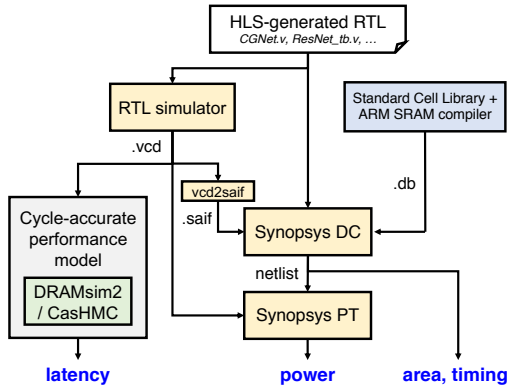


Figure 9: The RTL design and simulation flow.

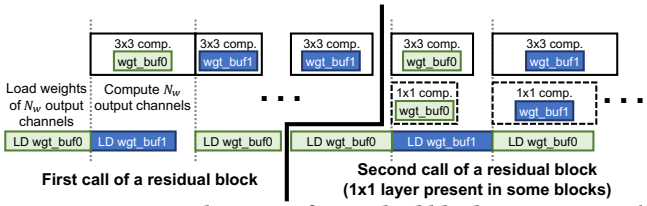


Figure 10: Timing diagram of a residual block — Compute and weight loading form a coarse-grained pipeline.

For performance estimates, we use a combination of RTL simulations for the accelerator and DRAMSim2 [29] for off-chip accesses. In this performance model, we first extract the memory trace and the compute time of an accelerator from a cycle-accurate RTL simulation, use DRAMSim2 [29] for DRAM latency estimates for memory accesses, and combine the results to obtain the overall execution time. We simulate two 64-bit DDR3 channels at 666MHz, where each channel contains one rank of eight banks. The total capacity of the simulated DRAM is 8GB. The DRAM parameters are verified against Verilog timing models from Micron [29].

Figure 10 shows a timing diagram of computing one residual block in ResNet, which applies to both the baseline and CGNet-xcel. The accelerator is called twice for each residual block. In order to save on-chip storage, the accelerator only loads the weights of N_w output channels in each step as shown in the figure. As the input channels to a conditional path may still be used for a subset of output activations, the accelerator reads the same amount of weights from off-chip memory as the baseline. CGNet can be extended to reduce memory accesses by pruning entire output channels. We apply double buffering to both baseline and CGNet-xcel so that computation and loading weights can be performed in parallel, as commonly done in CNN accelerators [15]. Note that the 3x3 and 1x1 convolutional layers in the same residual block of the ResNet-18 model are also computed in parallel.

4.2.2 Performance. Figure 11 compares the performance of the baseline, CGNet-xcel, and the theoretical execution time on CIFAR-10 and ImageNet. We use $N_w = 64$ for both the baseline and CGNet-xcel. The baseline and CGNet-xcel results are averaged over a batch of 64 images. To obtain the theoretical execution time of CGNet (`CGNet-theoretical`), we calculate the number of multiplications

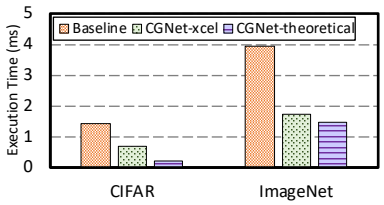
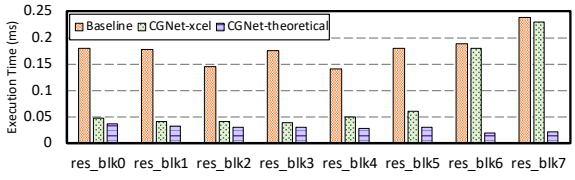
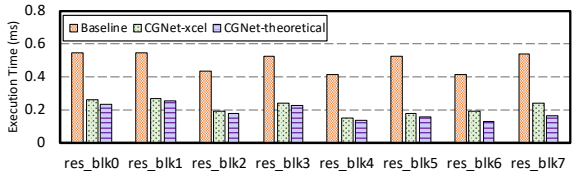


Figure 11: Comparison of baseline, CGNet-xcel, and theoretical performance on CIFAR-10 and ImageNet.

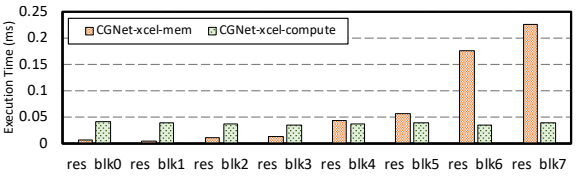


(a) CIFAR-10

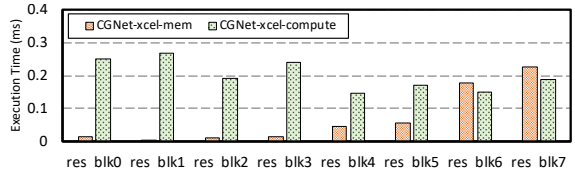


(b) ImageNet

Figure 12: Execution time for each residual block.



(a) CIFAR-10



(b) ImageNet

Figure 13: Compute and memory time of each residual block on the two datasets.

for each input image, and divide it by the number of multipliers in CGNet-xcel. In other words, the theoretical execution time assumes no memory overhead and 100% PE utilization.

As shown in Figure 11, CGNet-xcel has significant performance improvements over the baseline for both CIFAR-10 and ImageNet. Moreover, the performance of CGNet-xcel for ImageNet is close to the theoretical optimal, which indicates that our CGNet-xcel architecture is quite effective in leveraging the sparsity created by channel gating. On the other hand, for CIFAR-10, there is a noticeable performance gap between CGNet-xcel and CGNet-theoretical. Figure 12 shows more details by breaking down the execution time for each residual block. CGNet-xcel achieves near-optimal performance for all residual blocks on ImageNet. For CIFAR-10, while

Table 6: FPGA resource utilization of the baseline and CGNet-xcel.

	Baseline				CGNet-xcel			
	LUT	FF	DSP	BRAM	LUT	FF	DSP	BRAM
Total	34163	20295	783	780	65390	46998	798	933
Overhead	–	–	–	–	1.9×	2.3×	1.0×	1.2×

Table 7: ASIC area comparison between the baseline and CGNet-xcel.

		Baseline			CGNet-xcel		
		Capacity (KB)	# Banks	Area (mm^2)	Capacity (KB)	# Banks	Area (mm^2)
SRAM	Weight Dense	576.0	72	1.268	72.0	72	0.698
	Weight Sparse	–	–	0	504.0	63	1.109
	Weight Total	576.0	72	1.268	576.0	135	1.807
	Feature Dense	588.0	48	2.072	73.5	12	0.261
	Feature Sparse	–	–	0	533.0	189	2.572
	Feature Total	588.0	48	2.072	606.5	201	2.833
Combinational Area (mm^2)		0.040			0.111		
Noncombinational Area (mm^2)		0.039			0.111		
Total Area (mm^2)		4.557			5.574		
Overhead		–			1.22×		

CGNet-xcel achieves close-to-optimal performance for residual blocks 0-3, the performance gap between CGNet-xcel and CGNet-theoretical gets wider for the later residual blocks. In fact, for the last two residual blocks, CGNet-xcel only provides marginal performance improvements.

To further understand the source of the performance gap between CGNet-xcel and the theoretical limit, we breakdown the execution time of each residual block into compute time and memory access time in Figure 13. As shown in the figure, while the first four residual blocks are compute-bound for CIFAR-10, the rest are memory-bound, especially the last two residual blocks. This explains why CGNet-xcel cannot improve performance for `res_blk6` and `res_blk7` in Figure 12(a). Channel gating can significantly reduce the amount of computation during inference, but does not reduce memory accesses for the weights. The later residual blocks in ResNet-18 have more weights, and become memory-bound. The CGNet model for ImageNet requires a lot more computation compared to the one for CIFAR-10 because of the larger feature map sizes and the lower FLOP reduction ratio for channel gating. As a result, only the last two residual blocks are sightly memory-bound.

4.2.3 Area. To understand the area overhead of the proposed CGNet architecture compared to the baseline, we first show the FPGA resource usage in Table 6. The results are from post-place-and-route reports, targeting a Xilinx Zynq device (`xc7z045ffg900-2`) with a clock period of 10ns. As shown in the table, the extensions necessary for CGNet-xcel result in a 20% higher BRAM usage due to more complex memory banking. CGNet-xcel also increases the LUT and Flip-Flop usage because of extra multiplexing and deeper pipelines. On the other hand, the overhead for DSPs is small (2%) because CGNet-xcel has the same number of PEs with the baseline.

Table 8: Power, performance, and energy comparison of different platforms for ImageNet — The reported CPU power is the thermal design power.

Platform	ASIC		Intel i7-7700k	Nvidia GTX 1080Ti
	Baseline	**CGNet-xcel**		
Frequency (MHz)	800	800	4200	1923
Power (Watt)	0.202	0.256	91	225
Throughput (fps)	253.8	580.6	13.8	1563.7
Energy/Img (mJ)	0.796	0.441	6594.2	143.9

Table 9: The ASIC results for CGNet-NM.

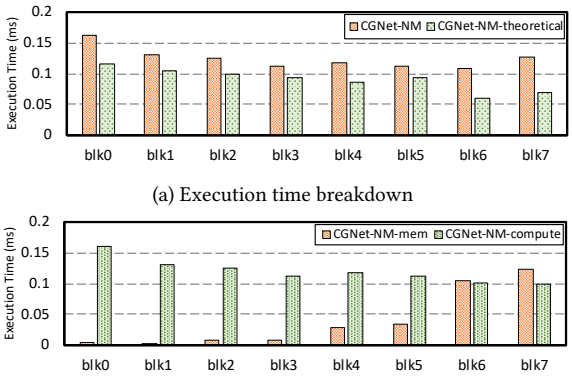
Dataset	Total Area	Freq.	Power	Throughput	Energy/Img
CIFAR-10	$3.859mm^2$	500MHz	0.159W	1004.4fps	0.158mJ
ImageNet			0.160W	200.9fps	0.796mJ

The DSP overhead comes from the additional index calculations for the sparse computation.

Table 7 shows the ASIC area comparison between the baseline and CGNet-xcel. Compared to the baseline, the ASIC implementation of CGNet-xcel noticeably increases the area of both combinational and non-combinational logic. However, because the total area is dominated by SRAMs for both the baseline and CGNet-xcel, CGNet-xcel only has 1.22× overhead in the total area. The table also provides the capacity, the number of banks, and the area of the SRAMs used to implement the weight and feature map buffers. While the total SRAM capacity of CGNet-xcel is not significantly higher than the baseline, $0.761mm^2$ more SRAM area is consumed because of the more complex memory banking scheme. The banking scheme also requires more and wider multiplexers, causes an increase in the pipeline depth, and results in higher area consumption in both combinational and non-combinational logic.

4.2.4 Comparison with Other Compute Platforms. Table 8 shows the power, throughput, and energy comparisons among different platforms. Here we compare CGNet-xcel with the baseline, as well as a CPU and a GPU. The CPU and the GPU perform regular ResNet-18 inference with a batch size of 32. As shown in the table, the baseline and CGNet-xcel outperform the CPU by 18.4× and 42.1× in terms of throughput. They are also four orders of magnitude more energy efficient than the CPU. Compared with the GPU, the baseline and CGNet accelerators are 180.1× and 326.3× more energy efficient, respectively. Compared with the baseline, the CGNet architecture did not have any noticeable impact on clock frequency. While the power consumption and the area of CGNet-xcel is higher than the baseline, its throughput is 2.3× higher. As a result, CGNet-xcel is 1.8× more energy efficient compared to the baseline.

4.2.5 High-Bandwidth, Compute-Constrained Platform. As shown in Section 4.2.2, CGNet-xcel can efficiently exploit the reduced computation from channel gating to significantly improve the performance of CNN inference. Our results also show that the performance improvements of CGNet-xcel can be limited when CNN performance is constrained by memory bandwidth. For example, the last few layers of ResNet-18 with channel gating are memory-bound for CIFAR-10. In that sense, CGNet-xcel will be particularly



(a) Execution time breakdown

(b) Compute and memory time of each residual block

Figure 14: CGNet-NM results on CIFAR-10.

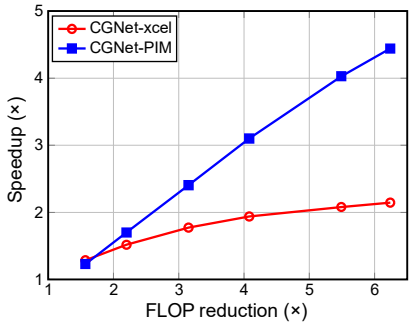


Figure 15: Performance scaling on CIFAR-10.

attractive for platforms with high memory bandwidth but limited compute resources, such as near-memory/in-memory computing platforms. For example, Hybrid Memory Cube (HMC) exploits the 3D stacking technology so that computation can be performed on a logic die which is placed under a stack of DRAM dies [30]. With more DRAM banks than traditional DRAMs, HMC provides very high internal memory bandwidth, which can be used by the computational resources on the logic die through Through-Silicon Vias (TSVs). However, a large portion of the logic die is occupied by DRAM peripherals such as the serial links, vault controllers, and interconnect for the TSVs. A previous study [31] estimates that the peripherals occupy $93mm^2$ out of the $99mm^2$ total area of the logic die in an HMC. In addition, the limited power budget constrains the frequency of the computational resources on the logic die.

Here, we study the applicability of CGNet across multiple platforms by evaluating a design for a near-memory (NM) computing scenario where the accelerator is placed on the logic die of an HMC. We conservatively set the target frequency to 500MHz and the area budget to $4mm^2$. We use CasHMC [32], a cycle-level HMC simulator, to estimate the memory latency in this case. We simulate a 4GB HMC device with 1.25GHz frequency, 32 vaults and eight banks per vault using the configuration file provided by CasHMC. We assume that the accelerator communicates with the DRAM stack through four half-width (8-lane) serial links on the logic die.

To meet the area constraint, we designed a variant of CGNet-xcel with $N_w = 8$ and the pipeline Initiation Interval (II) of 2, cutting the number of multipliers by half. We refer to this configuration as

CGNet-NM. The ASIC results of CGNet-NM are shown in Table 9. Compared to the results for the "normal" CGNet-xcel, the CGNet-NM accelerator has smaller area and lower power consumption. Figure 14 shows the performance of CGNet-NM on CIFAR-10, where the execution time breakdown is shown in Figure 14(a) and the breakdown of compute and memory time is shown in Figure 14(b). With a higher memory bandwidth, lower frequency, and fewer compute resources, only the last two residual blocks are slightly memory-bound, and the performance of CGNet-NM is close to the theoretical optimal. The results show that CGNet can be designed to meet varying area/power constraints and applied to speed up CNNs in multiple types of platforms.

4.2.6 Performance-Accuracy Trade-off. Our experiments also show that CGNet can be scaled to a range of performance-accuracy trade-off points, especially with high memory bandwidth. The FLOP reduction ratio of channel gating can be tuned during training. For more FLOP reduction, channel gating prunes away more computation at the cost of accuracy. Figure 15 shows how the performance of the normal CGNet-xcel and CGNet-NM scales with the FLOP reduction ratio for CIFAR-10. With high memory bandwidth, the performance of CGNet-NM scales almost linearly as more computation is pruned. On the other hand, the performance improvements for CGNet-xcel saturate earlier due to the memory bandwidth limit. We note that the CGNet-xcel performance scales much better for more compute-bound networks such as ResNet-18 on ImageNet, and CGNet-xcel can significantly improve CNN performance on traditional platforms as well.

5 RELATED WORK

Many recent proposals suggest to statically prune unimportant filters/features [4–6]. These static pruning techniques identify ineffective channels in filters/features by examining the magnitude of the weights/activation in each channel, and prune the ineffective subset of the channels from the model. The pruned model is then retrained to mitigate the accuracy loss from pruning. By pruning and retraining iteratively, these methods can compress the model size and reduce the computation cost. However, they reduce the same amount of computation for all inputs as they cannot exploit dynamic input-specific sparsity. In contrast, channel gating achieves a better performance-accuracy trade-off by identifying unimportant regions for a particular input and reducing computation for those regions at run time. We also believe that channel gating is complementary to static pruning approaches as it exploits input-dependent sparsity in the features.

SACT [33] introduces the spatially adaptive computation time technique on Residual Network [34], which can adjust the number of residual units for different regions of the input features. Lin et al. propose to use reinforcement learning to train a recurrent neural network making run-time decisions to prune output channels [35]. Both approaches require additional weights and extra computation to make the run-time decisions. In comparison, channel gating generates more fine-grained pruning decisions without incurring overhead in weights or computation.

Cnvlutin and Minerva [7, 8] propose to dynamically prune zero-valued pixels from the ReLU activation or pixels with small magnitude in the input features during inference. While such zero-pruning methods require no training effort, they strictly rely on the sparsity in the output features of each layer and only apply to ReLU-based activations. SnaPEA [9] extends the idea and propose to predict the ReLU zeros using the partial sum from a subset of input channels. While the use of the partial sum in a pruning decision is similar to our approach, channel gating enables a more general and aggressive pruning scheme by identifying unimportant regions in input features rather than only targeting zeros from one specific (ReLU) activation function. More importantly, channel gating introduces a training method to learn the gating policy (thresholds and weights), which turns out to be critical for achieving a small accuracy degradation. Similar to other inference-time approaches, SnaPEA often results in significant accuracy losses when targeting more than 2× FLOP reduction.

Predictive-based execution proposed in [36, 37] predicts zeros by first executing the significant bits. However, less information is presented in significant bits if batch normalization is used. Moreover, the performance gain of these methods is limited by the sparsity of zeros in the baseline networks. In contrast, channel gating can exploit various degrees of sparsity by choosing different target thresholds. Bit Fusion [38] proposes to reduce the computational cost by choosing different bit widths dynamically which is also applicable to CGNet. CirCNN and PermDNN [39, 40] use structured weight matrices with Fourier transform or permutation to achieve hardware-friendly structured sparsity. This line of research also focuses on static sparsity, although they are potentially complementary to our approach for reducing the size of the weights in CGNet.

6 CONCLUSION

This paper introduces a new fine-grained dynamic pruning technique for CNN, named channel gating, which reduce computational costs for CNN inference, along with a hardware accelerator architecture that can efficiently realize the dynamic pruning. The experimental results show that the channel gating can significantly reduce FLOPs with minimal accuracy loss: 5.5× FLOP reduction without accuracy loss for CIFAR-10, and 2× FLOP reduction with 1.0% accuracy degradation for ImageNet using ResNet-18. The paper also proposes a unified dense-sparse accelerator where both dense and sparse computations can be mapped onto the same processing elements, and shows that the proposed architecture can achieve close-to-optimal performance improvements for channel gating. Overall, by co-optimizing CNN algorithms and hardware architecture, the CGNet architecture provides higher performance gains with lower accuracy degradation compared to the state-of-the-art pruning techniques.

7 ACKNOWLEDGMENTS

This work was partially sponsored by Semiconductor Research Corporation and DARPA, NSF Awards #1453378 and #1618275, a research gift from Xilinx, Inc., and a GPU donation from NVIDIA Corporation. The authors would like thank the Batten Research Group, especially Christopher Torng (Cornell Univ.) for sharing their Modular VLSI Build System. The author also thank Zhaoliang Zhang and Kaifeng Xu (Tsinghua Univ.) for the C++ implementation of channel gating and Ritchie Zhao and Oscar Castañeda (Cornell Univ.) for insightful discussions.

REFERENCES

[1] A. Canziani, A. Paszke, and E. Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *arXiv preprint arXiv:1605.07678*, 2016.

[2] Z. Lu, S. Rallapalli, K. S. Chan, and T. F. La Porta. Modeling the Resource Requirements of Convolutional Neural Networks on Mobile Devices. In *Int'l Conf. on Multimedia (MM)*, Oct 2017.

[3] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *Int'l Conf. on Learning Representations (ICLR)*, May 2016.

[4] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning Filters for Efficient ConvNets. In *Int'l Conf. on Learning Representations (ICLR)*, May 2017.

[5] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning Efficient Convolutional Networks through Network Slimming. In *Int'l Conf. on Computer Vision (ICCV)*, Oct 2017.

[6] Y. He, X. Zhang, and J. Sun. Channel Pruning for Accelerating Very Deep Neural Networks. In *Int'l Conf. on Computer Vision (ICCV)*, Oct 2017.

[7] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[8] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. HernÃąndez-Lobato, G. Y. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[9] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[10] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[11] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Int'l Conf. on Machine Learning (ICML)*, Jul 2015.

[12] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Int'l Conf. on Machine Learning (ICML)*, Jun 2010.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv: 1512.01274*, 2015.

[14] W. Hua, C. De Sa, Z. Zhang, and G. E. Suh. Channel Gating Neural Networks. *arXiv preprint arXiv: 1805.12549*, 2018.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[16] J. Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2016.

[17] J. Zhang and J. Li. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[18] R. Zhao, W. Song, W. Zhang, T. Xing, J. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[20] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[21] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, 2009.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and F. Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 2015.

[23] X. Zhang, X. Zhou, M. Lin, and J. Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun 2018.

[24] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and C.-z. Xu. Dynamic Channel Pruning: Feature Boosting and Suppression. In *Int'l Conf. on Learning Representations (ICLR)*, May 2019.

[25] 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. In *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2018.

[26] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[28] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv preprint arXiv:1805.06085*, 2018.

[29] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 2011.

[30] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 2.1. *http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf*, 2014.

[31] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[32] D. Jeon and K. Chung. CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube. *IEEE Computer Architecture Letters*, 2017.

[33] M. Figurnov, M. D. Collins, Y. Zhu, L. Zhang, J. Huang, D. P. Vetrov, and R. Salakhutdinov. Spatially Adaptive Computation Time for Residual Networks. In *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[34] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun/Jul 2016.

[35] J. Lin, Y. Rao, J. Lu, and J. Zhou. Runtime Neural Pruning. In *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2017.

[36] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag. PredictiveNet: An Energy-Efficient Convolutional Neural Network via Zero Prediction. In *Int'l Symp. on Circuits and Systems (ISCAS)*, May 2017.

[37] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li. Prediction Based Execution on Deep Neural Networks. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[38] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-level Dynamically Composable Architecture for Accelerating Deep Neural Networks. In *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[39] C. Ding et al. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices. In *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2017.

[40] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan. PermDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices. In *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2018.