This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

Hardware Computation Graph for DNN Accelerator Design Automation without Inter-PU Templates

Jun Li , Wei Wang and Wu-Jun Li

***Abstract*—Existing deep neural network (DNN) accelerator design automation (ADA) methods adopt architecture templates to predetermine parts of design choices and then explore the remaining design choices beyond templates. Based on the architecture hierarchy at the processing unit (PU) level, these templates can be classified into intra-PU templates and inter-PU templates. Since templates limit the flexibility of ADA, designing effective ADA methods without templates has become an important research topic. Although there have appeared some works to enhance the flexibility of ADA by removing intra-PU templates, to the best of our knowledge no existing works have studied ADA methods without inter-PU templates. ADA with predetermined inter-PU templates is typically inefficient in terms of resource utilization, especially for DNNs with complex topology. In this paper, we propose a novel method, called *hardware computation graph* (HCG), for ADA without inter-PU templates. In HCG, a novel inter-PU architecture exploration strategy is proposed to optimize on-chip memory utilization. This strategy mainly depends on an appearing-frequency guided pruning method and an appearing-frequency first generation method. Experiments show that HCG can achieve competitive latency while using only 13% $\sim$ 90% of on-chip memory, compared with existing state-of-the-art ADA methods.**

***Index Terms*—DNN Accelerator; FPGA; Accelerator Design Automation; Hardware Computation Graph.**

I. Introduction

Due to their high flexibility and low energy consumption, Field Programmable Gate Arrays (FPGAs) have become one of the most popular platforms for deploying deep neural networks (DNNs). However, manual design of DNN accelerators for FPGAs suffers from significant amount of engineering effort and huge space of design choices. Hence, many DNN accelerator design automation (ADA) methods [2, 3, 4, 5, 6, 7, 8, 9] have been proposed to automate this process. In general, the effectiveness of ADA is determined by two main factors: an *accelerator representation* that abstracts the architecture into a set of explorable parameters and an *exploration strategy* that iteratively explores the parameters to find the optimal architecture.

Given the huge design choice space in the accelerator design process, existing ADA methods utilize architecture templates to predetermine part of design choices and then explore the remaining design options beyond those templates. Based on the architecture hierarchy at the processing unit (PU) level, these templates can be categorized into intra-PU and inter-PU templates. Intra-PU templates predetermine some design choices in one single PU, such as the dataflow style and processing parallelism in the PU. Inter-PU templates predetermine some design choices among PUs, such as the total number of PUs, the interconnection and the cooperation method between two specific PUs. Since templates are predetermined manually and subsequently limit the flexibility of ADA, designing effective ADA methods that do not rely on templates has emerged as a significant area of research. Works in [8, 9, 10, 11] propose to remove intra-PU templates and develop more expressive representations for DNN accelerators. For example, NAAS [10] removes intra-PU templates with the modeling of the connectivity between multipliers, which enhances the flexibility of ADA. However, removing of inter-PU templates is less discussed, and almost all existing works adopt inter-PU templates for ADA. For example, works in [8, 9, 11, 12, 13, 14] utilize *layer sequential template* which schedules and computes the DNN layer by layer on the chip. Works in [6, 7, 15, 16, 17, 18] adopt *layer pipelined template* which maps the entire DNN model over the accelerator. Due to the limited flexibility caused by predetermined inter-PU templates, ADA methods that rely on these templates are often inefficient in terms of resource utilization. Moreover, this problem becomes worse with the widespread adoption of multi-branch structure [19] and skip connection structure [20], because these structures make the topology of DNN more and more complex.

In order to improve the flexibility of ADA and resource utilization, this paper focuses on designing ADA methods without inter-PU templates. The contributions of this paper are outlined as follows:

- We first propose a novel *accelerator representation*, called *hardware computation graph* (HCG)[1], to abstract the inter-PU architecture. By representing each PU as a node and the interconnection between PUs as edges, HCG provides a principled tool to formulate an explorable space of design choices at the inter-PU architecture level.
- Based on the HCG representation, we further propose an inter-PU architecture *exploration strategy* to optimize the on-chip memory utilization. The exploration strategy

The preliminary version has been presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2022 [1]. This work is supported by NSFC Projects (No. 12326615). (Corresponding authors: Wu-Jun Li)

The authors are with the National Key Laboratory for Novel Software Technology, School of Computer Science, Nanjing University, Nanjing 210023, Jiangsu, China (e-mail: lijun@smail.nju.edu.cn; ww@nju.edu.cn; liwujun@nju.edu.cn)

[1]In this paper, we use HCG to denote both our ADA method and the inter-PU architecture representation (accelerator representation). The specific meaning of each occurrence can be easily identified from the context. In particular, if there is a 'method' behind HCG, like 'HCG method', this case of HCG denotes our ADA method. If there is a 'representation' behind HCG, like 'HCG representation', this case of HCG denotes the accelerator representation.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

mainly depends on an appearing-frequency guided pruning method and an appearing-frequency first generation method. Since HCG does not put any constraints on the intra-PU architecture design, it can be seamlessly integrated with existing works on intra-PU level of optimization, no matter whether with intra-PU templates or without intra-PU templates.

- To the best of our knowledge, HCG[2] is the first work to complete ADA without inter-PU templates. Furthermore, due to the flexibility resulted from removing inter-PU templates, HCG is also the first work to support irregularly connected DNNs, such as RandWire [21] and AmoebaNet [22], in ADA.
- To quantitatively evaluate the effectiveness of HCG, we conduct comprehensive experiments on widely used DNNs with complex topology which can be classified into regularly connected DNNs and irregularly connected DNNs according to the regularity of connection between layers. Experimental results show that for regularly connected DNNs, HCG can achieve competitive speed (latency) while using 13% $\sim$ 90% of on-chip memory required by existing state-of-the-art (SOTA) ADA methods. Since there have not existed ADA works considering irregularly connected DNNs, we compare HCG with a manually designed accelerator [23] for RandWire which is a representative irregularly connected DNN. The results show that HCG is 1.3× faster while using 2.5× fewer on-chip memory compared with the method in [23].

The remainder of this paper is organized as follows. Section II lists some works of two related research areas of this paper, including existing ADA methods and structural pruning methods for DNNs. Section III gives an overview introduction of the workflow of our HCG system. Section IV introduces HCG representation, which is the basis of the proposed inter-PU architecture exploration strategy. Section V introduces the details of HCG generation, which mainly depends on the proposed inter-PU architecture exploration strategy. Section VI introduces the implementation details of HCG, such as on-chip memory usage, off-chip memory usage and so on. Section VII comprehensively evaluates the effectiveness of HCG. Finally, Section VIII concludes this paper.

II. Related Works

A. DNN Accelerator Design Automation

Due to the significant human effort required for manually designing DNN accelerators, ADA [3, 6, 7, 8, 9, 11, 12, 13, 15, 16, 17, 24, 25, 26] has been proposed to automate the designing process. To boost the flexibility of ADA, some methods, including AMOS [8], Gemmini [12], TENET [9], NAAS [10] and Interstellar [14], propose to remove intra-PU templates. However, almost all existing ADA methods, including LCMM [3], Clound-DNN [16], Multi-FPGA [17] and many others [6, 7, 8, 9, 11, 12, 13, 15, 26], adopt inter-PU templates to predetermine design choices at inter-PU architecture level. Inter-PU templates not only limit the flexibility of ADA but also cause inefficiency in terms of resource utilization, which motivates the work in this paper.

B. DNN Structural Pruning

DNN structural pruning has been proven to be successful in enhancing memory utilization. Unlike unstructured pruning, structural pruning is favored for its convenience in accelerating DNNs, since it is more compatible with modern deployment platforms. In practice, it structurally removes those "unimportant" filters. Hence, one of the key problems of structural pruning is the filter importance evaluation. To this end, works emphasizing on the importance evaluation have been proposed [27, 28, 29]. For example, [27] proposes a pruning method based on $L1\text{-}Norm$ importance evaluation. FPGM [28] proposes a novel criterion named geometric median as the pruning importance. These works are orthogonal to our design, since our design does not put any constraints on the specific importance evaluation.

There also exist several works focusing on the joint optimization of pruning technique and accelerator on FPGA [30, 31, 32, 33, 34]. For example, [30] accelerates the DNN for semantic segmentation on FPGA. Work in [34] proposes to jointly optimize both the hardware and pruning scheme. However, these works mainly target at those manually designed accelerators, which are not the focus of this paper.

III. System Overview

Figure 1 illustrates the workflow of HCG. As illustrated, HCG consists of three steps: software computation graph (SCG) generation, HCG generation and code generation.

SCG generation includes two sub-steps: model parsing and topology regularization. Firstly, the model parsing step converts DNN model descriptions, which are defined by external protocols such as Caffe and ONNX, into the SCG format that is utilized within the system. HCG supports almost all kinds of commonly used layers, including depth-wise convolution, convolution, fully connected layer, pooling (average pooling, max pooling and global pooling), element-wise addition, concatenation and non-linear activation functions such as ReLU. Figure 1(a) illustrates a simple example of model parsing. Secondly, topology regularization is conducted to unify the layers with multiple inputs or outputs. These layers are regularized to multiple nodes with only two inputs or outputs in SCG. Figure 1(b) shows the regularized SCG topology derived from Figure 1(a). The function of Distrib nodes D1 and D2 in Figure 1(b) is to distribute the input data to two output ports.

HCG generation mainly depends on the proposed inter-PU architecture exploration strategy, which consists of three sub-steps: pre-pruning, PU list exploration and interconnection exploration. Firstly, the pre-pruning sub-step prunes the SCG with the appearing-frequency guided pruning method (Section V-A). Secondly, the PU list exploration sub-step generates the PU list tailored to the target FPGA and given DNN (Section V-B). Since each node of HCG is a PU, this sub-step essentially generates the nodes of HCG with a two-stage process. Stage 1 generates a list of basic PUs (i.e., HCG nodes) with the appearing-frequency first generation method. Stage 2

[2] The preliminary version of HCG appears in the conference version [1] of this paper.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

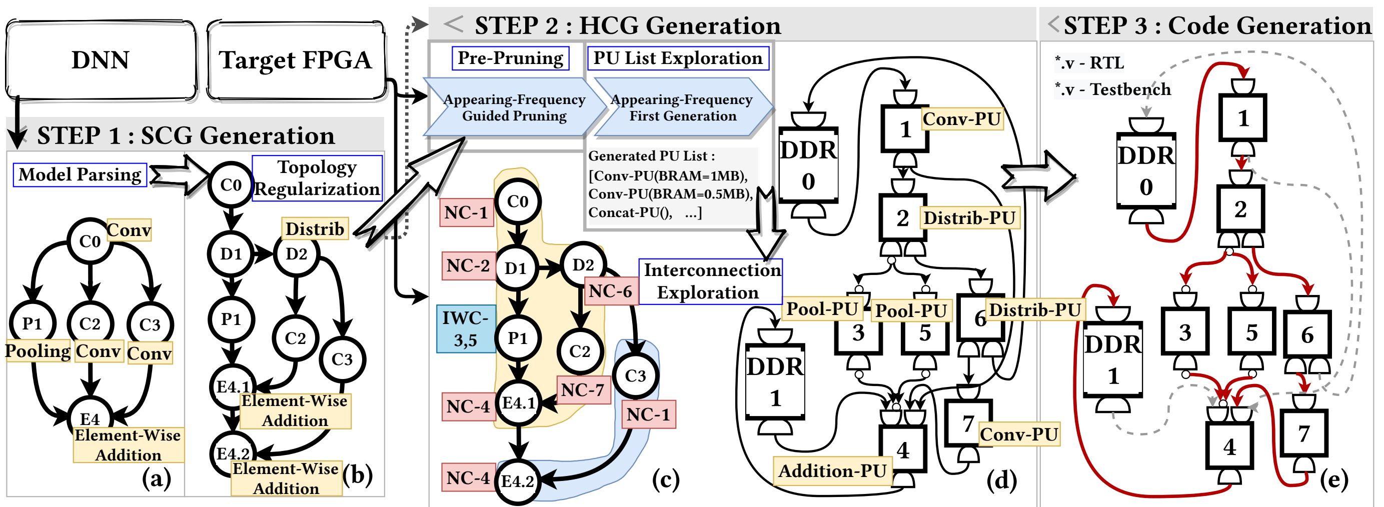


Fig. 1. Workflow of our HCG system. **(a)** an illustrative example SCG, which includes three branches; **(b)** regularized SCG derived from (a), where the nodes "D1" and "D2" denote distribution layers, and the nodes "E4", "E4.1" and "E4.2" denote element-wise addition; **(c)** the sub-network partition and PU allocation scheme derived from *Interconnection Exploration* step. From the figure, the given SCG is partitioned into two sub-networks. We use different colours to distinguish different sub-networks. The PUs allocated to each layer and their cooperation type is attached besides the nodes. For example, layer P1 needs input width cooperation (IWC) and PUs allocated to layer P1 are PU-3 and PU-5. Other layers in this example do not need PU cooperation, so their cooperation types are no cooperation (NC). **(d)** the inter-PU architecture derived from *Interconnection Exploration*. The details of the figure will be introduced in Section IV; **(e)** the execution state of HCG corresponding to the first sub-network (layer C0, D1, D2, P1, C2 and E4.1) in (c). The red bolded lines denote the enabled data path when computing the first sub-network.

iteratively tunes the basic PU list according to the resource constraint of target FPGA and guidance from SCG. Finally, the interconnection exploration sub-step partitions the DNN into multiple sub-networks according to the PU list determined in the above steps. Then, specific PUs are allocated to the corresponding layers so that the interconnection between every two specific PUs can be determined. The detailed steps and algorithms of these strategies will be introduced in Section V.

Code generation converts HCG and PU allocation result to synthesizable register transfer level (RTL) code and run-time control code, respectively. The synthesizable RTL code describes the architecture of accelerator. The runtime control code mainly includes the configuration parameters of each PU and control signals at inter-PU level.

Since SCG generation and code generation have been widely studied and many existing methods [6, 7, 15, 16] can be used, this paper mainly focuses on HCG generation, which will be introduced in the following two sections.

IV. HCG Representation

This section introduces HCG representation, which is the basis of our HCG system.

By representing DNN accelerators at the inter-PU level, HCG is proposed to support the inter-PU architecture exploration. The main idea of HCG is to represent PUs as nodes and the interconnection between PUs as edges. Node types in HCG are similar to layer types in SCG. The edge in HCG depends on the cooperation methods between nodes.

In the inter-PU architecture, PUs are organized to cooperate with each other, and one layer can be computed with multiple PUs according to its computation requirement. The common method for the PUs' cooperation is to partition data into different PUs and process partitioned data in different PUs

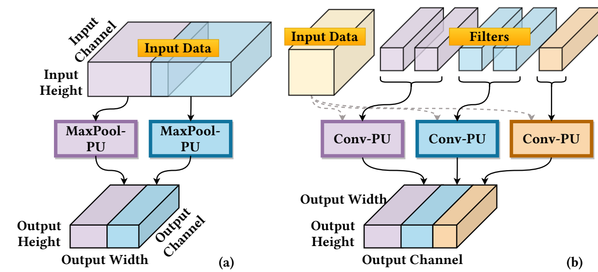


Fig. 2. Illustrative examples of (a) input width cooperation (IWC) and (b) filter level cooperation (FLC).

[35, 36, 37]. However, unlike these existing works with an inter-PU template that predetermines the cooperation methods between PUs and partitions data in a fixed way [35, 36, 37], HCG provides multiple cooperation methods as candidates and chooses one suitable method during the inter-PU architecture exploration.

Specifically, HCG offers three possible cooperation methods between PUs for different layers, including input width cooperation (IWC), filter level cooperation (FLC), and no cooperation (NC). IWC targets at pooling layers because pooling layers are always located at the early part of DNN, which causes the width of input activation to be larger than its depth. When the on-chip memory footprint[3] of a specific pooling layer is too large to be accommodated by one single PU, the activation should be partitioned on the input width dimension. More specifically, IWC method slices the activation with the shape of $(Depth \times Height \times Width)$ into multiple

[3]This paper uses *on-chip memory size* to denote the number of on-chip memory equipped in a PU, and uses *on-chip memory footprint* to denote the required on-chip memory on FPGA of a layer.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

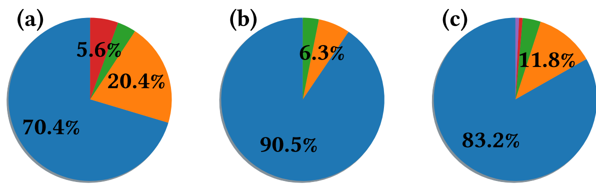


Fig. 3. The distribution of memory footprint on FPGA of every layer in ResNet-50 (a), Inception-V3 (b) and AmoebaNet (c). Different colors in a single chart represent different on-chip memory footprints, and the percentage is the ratio between the number of layers with the same memory footprints to the total number of layers.

Algorithm 1 Dependency Build

Input: SCG;

Output: Dependency matrix: D;

1: $L \leftarrow \|\text{SCG}\|$;
2: $A \leftarrow$ Adjacent matrix of SCG;
3: $D \leftarrow 0^{L\times L}$; // Initialize to zero.
4: **for** $i \in \{1, 2, ..., L\}$ **do**
5:     **for** $j \in \{1, 2, ..., L\}$ **do**
6:         $D[i, j] = A[i, j] \bigvee (\exists k, D[i, k] \bigwedge A[k, j]) \bigwedge$
7:         IsTransParent(SCG_j);
8:     **end for**
9: **end for**

small activations with the shape of $(Depth \times Height \times Width_i, s.t.\ \Sigma_i Width_i = Width)$. Each small activation is assigned to a PU for computation. Figure 2(a) gives an example in which the activation of a max pooling layer is sliced into two PUs. Moreover, Figure 1(c) and Figure 1(d) give an illustrative example about how IWC works in HCG. In the figure, PUs allocated to layer P1 are PU-3 and PU-5, which cooperate through IWC. Hence, the activation of layer P1 will be sliced into two blocks, which will be transferred to PU-3 and PU-5 on the fly, respectively. FLC targets at convolution layers and fully connected layers. Unlike IWC, FLC slices the original weight with the shape of $(FilterNum \times Depth \times KernelSize \times KernelSize)$ into multiple blocks of weights with the shape of $(FilterNum_i \times Depth \times KernelSize \times KernelSize, s.t.\ \Sigma_i FilterNum_i = FilterNum)$, which will be transferred to multiple PUs. Figure 2(b) gives an example in which the weight of a convolution layer is sliced into three PUs. If a specific PU can accommodate the on-chip memory footprint of a layer, NC should be adopted. Besides these three cooperation methods, it is worth mentioning that since HCG representation does not put any constraints on the cooperation methods, it can be extended to support other kinds of cooperation methods.

Since there are three possible cooperation methods for each layer, the interconnection between two layers includes $3^2 = 9$ types. Correspondingly, there are nine possible interconnection types between each pair of PUs. For example, the interconnection between PUs allocated to layer P1 (PU-3 and PU-5 cooperate through IWC) and layer E4.1 (PU-4) in Figure 1(c) is denoted as $\{\text{IWC} \rightarrow \text{NC}\}$.

V. HCG Generation

This section introduces the details of HCG generation, which consists of three sub-steps: pre-pruning, PU list exploration and interconnection exploration.

A. Pre-Pruning

Based on the introduced HCG representation, we propose a novel appearing-frequency guided pruning method to enhance the on-chip memory utilization. Specifically, it is motivated by the observation that there is always a frequent item in the memory footprint set of layers on FPGA. For example, Figure 3 shows the distribution of layers' memory footprint on FPGA of three DNNs. From the figure, we can see that there always exists an apparent part that occupies the majority of each pie chart. Hence, inspired by structural pruning which has been proven to be successful in lowering the memory footprint of DNNs on GPU, we propose to structurally prune the DNN on the basis of this observation.

The basic idea of appearing-frequency guided pruning is to prune those layers that require infrequent memory footprints into frequent ones as much as possible. The straightforward method is to calculate the memory footprint of every layer and prune the one whose memory footprint is not frequent. However, the topology of DNNs prevents us from adopting this method and raises two problems for us. On the one hand, pruning a specific layer affects the memory footprint of its adjacent layers. For example, if a convolution layer is structurally pruned with a certain number of filters, its output data shrinks in the depth dimension. This will change the input depth number of its downstream layers and thus affect the downstream layers' memory footprint. Moreover, the complex topology of modern DNNs makes this problem even more challenging since the layers always have more than one upstream and downstream layer. On the other hand, layers are not always prunable due to topology constraints. For example, according to our experiments, the last layer of DNNs always requires more memory than others due to the large weight dimension. However, the filters of the last layer are not structurally prunable because they are usually used for classification and each filter corresponds to a specific classification label.

To solve these two problems, we propose to firstly generate pruning groups for the given DNN based on its topology in the pre-pruning step. Specifically, each pruning group has a root layer and a list of dependent layers. The root layer is the one that requires an infrequent memory footprint and needs pruning. The dependent layers are those layers that change in memory footprint with the root layer. Before the generation of pruning groups, the rules that determine the dependencies between layers should be defined. Algorithm 1 shows the dependencies building process. During the process, layers are classified into two categories according to their behaviour over data. The first category is named the *transparent layer*, suggesting that the pruning scheme of the layer's input data can be transmitted into its output data. For example, if one of the input data of the concatenation layer is pruned in the depth dimension, the corresponding positions of the layer's output will also be pruned. Specifically, this layer category includes layers such as concatenation, pooling, and non-linear

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

Algorithm 2 Pruning Group Generation

Input: Dependency matrix: D; SCG;
Output: Pruning Group: G;
1: $G \leftarrow \{\ \}$;
2: **for** $i \in \{1, 2, ..., \|\text{SCG}\|\}$ **do**
3:   **if** IsFreq(SCG$_i$) == Fasle **then**
4:     $g \leftarrow$ empty object of pruning group;
5:     $g.root \leftarrow$SCG$_i$;
6:     $g.deps \leftarrow \{\text{SCG}_j | D_{i,j} = 1\}$;
7:     $G \leftarrow G \cup \{g\}$;
8: **end for**

activation. The second category is named the *non-transparent* layer, which consists of convolution, fully connected layer, and element-wise addition. The pruning scheme of input data will not be transmitted to the output data of these layers. For example, the dimension of the output data of convolution layers will not change with the input data. Based on the layer classification (line 7 in Algorithm 1), the process of dependency building can be summarized as the depth-first traversal upon SCG with the arising of the non-transparent layer as the ending condition.

The pruning groups are generated with the guidance of the given dependency matrix. Algorithm 2 shows the process. The main idea of this process is to find out every root layer and its dependent layers. Specifically, the algorithm traversals each layer and constructs a pruning group object for those layers that require infrequent memory footprints. The function *IsFreq* is used to judge whether the given layer is frequent.

Finally, the generated pruning groups are progressively pruned one by one until the root layers in all groups are frequent or the iteration counter exceeds the preset maximum number of steps. Algorithm 3 shows the process. Specifically, for each pruning group, the algorithm structurally prunes the filters of the root layer with a preset step size. The step size is used to guarantee that the filter number of the root layer after pruning is divisible by the parallelism number. Currently, we directly use the off-the-shelf structural pruning method with $L1\text{-}Norm$ importance evaluation [27] to operate the root layer (line 7 in Algorithm 3). After the root layer is pruned, the dependent layers are manipulated in the corresponding dimension. Then, a function *CheckAndLegalize* is used to legalize the pruned structure of DNN. This function mainly relies on two rules. One rule is to fix the dimension inconsistency between input data of element-wise addition layers since this layer requires the same dimension of all input data. If the function finds this inconsistency, all dimensions of the input data of the layer will be forcibly changed to the maximum dimension across all input data of this layer. The other rule is to guarantee that the output of the classification layer is not pruned. The output depth of the classification layer must be the same as the number of classification labels.

With the appearing-frequency guided pruning, not only does the total memory footprint of the whole DNN decrease, but the memory footprints of layers also become more concentrated. Figure 4 shows the distributions of memory footprint of every layer in three DNNs. From the figure, we can find that the frequent item of memory footprint becomes more frequent,

Algorithm 3 Appearing-Frequency Guided Pruning

Input: SCG; dependency matrix: D; Pruning step size: $StepSize$;
Output: Pruned SCG: SCG$_P$;
1: SCG$_P \leftarrow$ SCG;
2: // Build the pruning groups for SCG$_P$.
3: $G \leftarrow$ Pruning-Group-Generation(SCG$_P$, D);
4: $ite \leftarrow 0$;
5: **while** $ite \leq MaxIte$ or $(\exists g' \in G, \text{IsFreq}(g'.root) \neq True)$ **do**
6:   **for** $g \in G$ **do**
7:     Prune($g, StepSize$) ;
8:     CheckAndLegalize(G);
9:   **end for**
10:   $ite++$;
11: **end while**

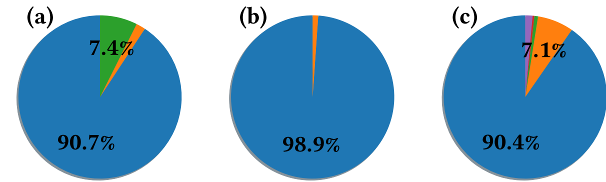


Fig. 4. The distribution of memory footprint on FPGA of every layer in ResNet-50 (a), Inception-V3 (b) and AmoebaNet (c) after appearing-frequency guided pruning.

compared with the original ones in Figure 3. The concentration of memory footprints benefits the following PU list generation since it enhances the appearing-frequency first principle.

B. PU List Exploration

Since each node of HCG is a PU, the exploration of PU list essentially generates the nodes of HCG. This section provides guidelines of the exploration and the step-by-step breakdown of the process.

1) Guidelines: The process of PU list exploration can be described as an optimization problem under constraints. In this paper, the optimization objective is the on-chip memory utilization, as introduced in Section I. The constraints refer to the hardware resource of the given FPGA, including on-chip memory, off-chip memory, arithmetic resource (DSP), and logical resource (LUT).

Specifically, HCG optimizes the on-chip memory utilization by minimizing the *mismatched on-chip memory size* between the on-chip memory footprints of layers and the on-chip memory sizes of their allocated PUs, which is formally defined in Equation (1).

$$\begin{aligned} &\min \sum_{i=1}^{n} \frac{\sum_{j=1}^{m_i} PU_j.ocm - FPT(SCG_i)}{n} \\ &s.t. \sum_{j=1}^{m_i} PU_j.ocm - FPT(SCG_i) \geq 0 \end{aligned} \tag{1}$$

In Equation (1), n is the total number of nodes in SCG, m_i is the number of PUs allocated to SCG_i, $PU_j.ocm$ is the on-chip memory size of the $j\text{-}th$ PU allocated to SCG_i, $FPT(SCG_i)$ is the on-chip memory footprint of SCG_i. The numerator of Equation (1) represents the mismatched on-chip memory size of SCG_i. If the numerator is larger than zero, the allocated on-chip memory size exceeds the footprint of SCG_i,

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

Algorithm 4 Stage 1 of PU List Exploration

Input: Pruned SCG: SCG$_P$; total number of DSPs, on-chip memory of the target FPGA : (DSP_T, OCM_T);
Output: Basic PU list: $BasicPU_{List}$;
1: $BasicPU_{List} \leftarrow \{\ \}$;
2: $tmpPU_{List} \leftarrow \{\ \}$;
3: **for** $i \in \{1, 2, ..., \|\text{SCG}_P\|\}$ **do**
4: $newPU$ = empty object of PU;
5: $newPU.ocm = FPT(\text{SCG}_{P}i)$;
6: $tmpPU_{List} = tmpPU_{List} \cup \{newPU\}$;
7: **end for**
8: Cluster the PUs in $tmpPU_{List}$ with their on-chip memory size, and then count the appearing-frequency of each PU cluster;
9: $maxFreq \leftarrow$ the max appearing-frequency of PU clusters;
10: $maxFreqPU \leftarrow$ a PU in the PU cluster with max frequency;
11: $(DSP_{req}, OCM_{req}) \leftarrow$ Resource-Calculation$(maxFreqPU)$;
12: Append $\min(\lfloor \frac{DSP_T \times maxFreq}{DSP_{req}} \rfloor, \lfloor \frac{OCM_T \times maxFreq}{OCM_{req}} \rfloor)$ $maxFreqPU$s to $BasicPU_{List}$;

which results in waste of on-chip memory. If the numerator is zero, the on-chip memory size allocated to SCG_i matches its footprint without wasted on-chip memory. Hence, Equation (1) summarizes the averaged wasted on-chip memory of the input DNN.

From Equation (1), the ideal case between layers and PUs on the chip is that every layer can find a PU generated exactly according to its memory footprint, where the mismatched on-chip memory size is zero. However, generating the inter-PU architecture without inter-PU templates suggests that this ideal case should be divided into two ideal subcases. Firstly, there should exist PUs on the chip that are generated exactly according to a layer's memory footprint. Secondly, the PU generated exactly according to a layer's memory footprint should be allocated to this layer. Based on these two ideal subcases, we propose two guidelines for PU generation and PU allocation, respectively.

For the guideline of PU generation, the first ideal subcase suggests that the ideal generation method should generate a PU according to a layer's memory footprint. On the one hand, this generation method is not practicable for every layer since HCG does not adopt any inter-PU template, and subsequently PUs are possibly reused across different layers. On the other hand, the more layers adopt the ideal generation method, the smaller the mismatched on-chip memory size is. Hence, to maximize the number of layers adopting the ideal generation method, HCG applies this method to those layers that appear most frequently. The motivation of this strategy is similar to that of the appearing-frequency guided pruning, as introduced in Section V-A. This guideline is reflected in an appearing-frequency first generation method.

For the guideline of PU allocation, the second ideal subcase suggests that the allocation of those layers that can find a PU generated according to their memory footprints should be prioritized. This guideline is reflected in a PU allocation algorithm with priority, which will be introduced in following subsection.

2) PU List Exploration: PU list exploration involves two stages. Stage 1 generates a list of basic PUs and stage 2 iteratively tunes the list until it satisfies the requirements of input DNN.

Algorithm 5 Stage 2 of PU List Exploration

Input: Basic PU list: $BasicPU_{List}$; Pruned SCG: SCG$_P$; on-chip memory, DSPs and off-chip communication channels of the target FPGA : $(OCM_T,\ DSP_T,\ OFM_T)$;
Output: PU list: PU_{List};
1: start $\leftarrow 0$, end $\leftarrow 0$, $PU_{List} \leftarrow BasicPU_{List}$;
2: **while** True **do**
3: **for** i $\in \{1, 2, ..., MaxL\}$ **do**
4: cand $\leftarrow$ SCG$_P$[start : start+i];
5: $Alloc_m \leftarrow$ **PU-Allocation**(cand, PU_{List});
6: $(OCM_m,\ DSP_m,\ OFM_m) \leftarrow$
7: Resource-Calculation$(Alloc_m)$;
8: **if** $(OCM_m, OFM_m) < (OCM_T, OFM_T)$ **then**
9: $DSP_c \leftarrow DSP_m$;
10: $Alloc_c \leftarrow Alloc_m$;
11: $end \leftarrow end + 1$;
12: **else break**;
13: **end for**
14: $newPU \leftarrow$ (the new PU derived from $Alloc_c$);
15: **if** $DSP_c < DSP_T$ **then**
16: // Append a new PU to the PU_{List}.
17: $PU_{List} = PU_{List} \cup newPU$;
18: **else**
19: // Fuse the new PU to the smallest PU inside PU_{List}.
20: $PU_s \leftarrow PU^*, s.t.(\forall i, PU_i.ocm \leq PU^*.ocm)$;
21: $PU_s.ocm\ +=\ newPU.ocm$;
22: **if** end = len(SCG$_P$) **then return** // End of the input DNN.
23: **else** start $\leftarrow$ end;
24: **end while**

Stage 1 is based on the appearing-frequency first generation. Algorithm 4 shows the process. The strategy first temporally generates a PU for every layer according to its memory footprint (line 2 $\sim$ 7). Function FPT calculates the on-chip memory size occupied by the input footprint. In practice, if block random access memory (BRAM) is used to accommodate the footprint, FPT calculates the size of BRAM consumed by the input layer and returns the capacity of consumed BRAMs. Then these temporally generated PUs are clustered according to their on-chip memory size, and the appearing-frequency of each PU cluster is calculated (line 8). The PUs in the PU cluster with the highest appearing-frequency are chosen as the primary elements of the basic PU list, while other PU clusters are ignored (line 10). Finally, after calculating the hardware resources of one single primary element (line 11), the algorithm appends as many primary elements as possible to the basic PU list under the constraints of hardware resources (line 12). Here, the total number of hardware resources (DSP and on-chip memory) are multiplied by $maxFreq$ to guarantee that the primary elements occupy a major part of the whole chip in priority while leaving space for stage 2 of PU list exploration. By prioritizing the most frequently appearing layers, the majority of DNN can be allocated with matched PUs without the waste of on-chip memory, reducing the mismatched on-chip memory size.

Stage 2 iteratively tunes the basic PU list until it satisfies the computation requirement of the entire DNN. Algorithm 5 shows the process. In each iteration, the algorithm first generates a sub-network candidate comprising a continuous part of DNN (line 4). Then, the PUs allocated for the candidate

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

Algorithm 6 PU Allocation Strategy with Priority

```
Input: SubSCG (A sub-network); PU_List;
Output: D (a dictionary, layer → PU);
 1: // STEP 1: Prioritized allocation.
 2: NonPrio ← { };
 3: for i ∈ {1, 2, ..., ||SubSCG||} do
 4:     // {PUs of SubSCG_i.type} is to aggregate PU from PU_List of the SubSCG_i.type;
 5:     PU_t = {PUs of SubSCG_i.type};
 6:     D[SubSCG_i] ← PRIO-ALLOC(SubSCG_i, PU_t);
 7:     PU_List = PU_List \ D[SubSCG_i];
 8:     if D[SubSCG_i] == ∅ then
 9:         NonPrio = NonPrio ∪ SubSCG_i;
10: end for
11: // STEP 2: Search the PU list to minimize the mismatch.
12: for i ∈ {1, 2, ..., ||NonPrio||} do
13:     PU_t = [PUs of NonPrio_i.type];
14:     D[NonPrio_i] ← MIN-MISMATCH(NonPrio_i, PU_t);
15:     PU_List = PU_List \ D[NonPrio_i];
16: end for
17: function PRIO-ALLOC(op, PUs)
18:     if ∃i, s.t.(PUs_i.ocm = FPT(op)) then
19:         return {PUs_i};
20:     else return ∅;
21: function MIN-MISMATCH(op, PUs)
22:     if Σ[PUs].ocm ≥ FPT(op) then
23:         return [PUs_i], min(ΣPUs_i.ocm),
24:                 s.t.(ΣPUs_i.ocm ≥ FPT(op));
25:     else return [newPU, PUs],
26:                 s.t.(Σ[newPU, PUs].ocm = FPT(op));
```

sub-network are derived based on the current PU_{List}, in which the PU allocation strategy with priority is adopted (line 5).

The PU allocation strategy is shown in Algorithm 6. It takes a sub-network and a PU list as input and allocates specific PU to every layer in the input sub-network. In the algorithm, the allocation of those layers which can find a PU generated according to their footprint is considered in priority (STEP 1 in Algorithm 6). Function PRIO-ALLOC (line 6) traverses the PU_{List} and returns a specific PU that accommodates the same size of on-chip memory as the footprint of the input layer. Then, the algorithm searches the PU_{List} to minimize the mismatch of those layers that are left by PRIO-ALLOC (STEP 2 in Algorithm 6). Function MIN-MISMATCH returns *a list of PUs* with respect to the condition at line 24. The term $min(\Sigma PUs_i.ocm)$ suggests using the minimal memory to compute the input layer, which also minimizes the mismatched on-chip memory size of the layer. Specifically, if the input footprint is smaller than any PU's on-chip memory size in the PU_{List}, MIN-MISMATCH returns a list comprising the PU with the smallest on-chip memory. If the input footprint is larger than any PU's on-chip memory size in the PU_{List}, MIN-MISMATCH returns a list comprising multiple PUs that cooperate through FLC or IWC method. Moreover, as the allocation process goes on, PUs are progressively consumed (line 7 and 15). When the existing PU_{List} no longer satisfies the footprint of a specific layer, MIN-MISMATCH will calculate the number of missing resources and return a new PU object accordingly (line 26).

After PU allocation, the sub-network with a maximum length is chosen under hardware resource constraints (line 3 ∼ 5 in Algorithm 5). Since the number of PUs is positively correlated to the number of layers in sub-networks, choosing the sub-network with a maximum length will allocate as many PUs on the chip as possible, which shortens the processing latency. For the new PU derived from PU allocation, it is appended to the PU_{List} in priority (line 17). When the DSPs are not sufficient to append a new PU, the algorithm fuses the new PU to the smallest PU inside the PU_{List} (line 19 ∼ 21).

With Algorithm 5, we can get different inter-PU architectures under different resource constraints. More specifically, when the resources are sufficient to put all layers on the chip, the strategy in HCG will generate an architecture that is the same as that generated with a *layer pipelined template* [6, 7, 15, 16, 17, 18]. When the resources are so rare that only one PU can be placed on the chip, the strategy in HCG will generate an architecture that is the same as that generated with a *layer sequential template* [8, 9, 11, 12, 13, 14]. That is to say, the ADA methods with *layer pipelined template* and *layer sequential template* can be treated as degenerated cases of our HCG method.

C. Interconnection Exploration

After the PU list exploration, the interconnection exploration is conducted. Since the interconnection between PUs is derived from PUs' cooperation method and allocation results as introduced in Section IV, this strategy first performs PU allocation on SCG with the PU list determined at the PU list exploration step. Specifically, this allocation process will calculate the number of PUs needed for each layer and allocate specific PUs to each layer, which consequently determines the interconnection between PUs. The method of the PU allocation process is similar to that in the PU list exploration, except that no modification to the PU list is allowed. Furthermore, in order to reduce the difficulty of routing and placement in hardware, we prune the interconnection with the same type between two specific PUs.

VI. Hardware Implementation

In this section, we introduce the inter-PU infrastructure, the intra-PU architecture, on-chip memory usage, and off-chip memory usage.

A. Inter-PU Infrastructure

Due to the PU cooperation, nine interconnection types between PUs exist in HCG. Thus, a flexible inter-PU infrastructure is required to satisfy organizational requirements arising from these interconnection types. On the one hand, the inter-PU infrastructure should flexibly deal with all types of interconnection between PUs. To this end, we split every interconnection into frontend and backend, which are placed at the input port and the output port of each PU, respectively. Figure 1(c) and Figure 5 shows an example. In Figure 1(c), the pooling layer P1 needs IWC, and the PUs allocated to it are PU-3 and PU-5. At the same time, its downstream layer E4.1 needs NC, and the PUs allocated to it are PU-4. Hence, an IWC → NC frontend is placed at the output

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

Algorithm 7 Convolution Pseudo Code

```
Input: Input Activation: A; Weight: W;
Output: Output Activation: Out;
 1: // A follows the shape of [H_i, W_i, D_i].
 2: // W follows the shape of [F, K, K, D_i].
 3: Out = 0^{H_o×W_o×F}; // Initialize to zero.
 4: for oh ∈ [1, H_o]
 5:   for ow ∈ [1, W_o]
 6:     for f ∈ [1, F/OutP]
 7:       for kh ∈ [1, K]
 8:         for kw ∈ [1, K]
 9:           for id ∈ [1, D_i/InP]
10:           #pragma unroll
11:             for ip ∈ [1, InP]
12:             #pragma unroll
13:               for op ∈ [1, OutP]
14:                 Out[oh, ow, f * OutP + op]+ =
15:                 A[oh + kh, ow + kw, id * InP + ip]*
16:                 W[f * OutP + op, kh, kw, id * InP + ip]
```

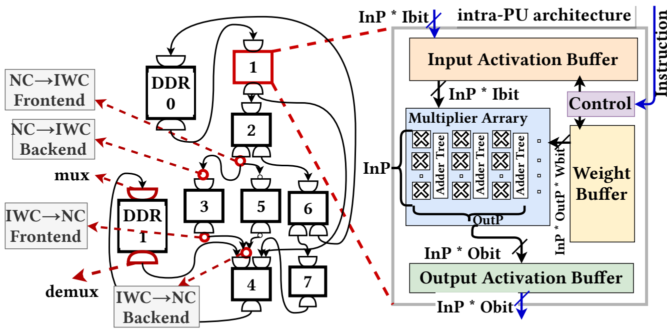


Fig. 5. Inter-PU infrastructure and intra-PU design.

ports of PU-3 and PU-5, and an IWC $\rightarrow$ NC backend is placed at the input port of PU-4, as shown in Figure 5. In practice, the code generation step of the HCG system generates these frontend and backend modules as the components of the accelerator. On the other hand, the inter-PU infrastructure should provide support for computing different sub-networks with one shared HCG. Since the topology and PUs allocation scheme differ between sub-networks, the data paths between PUs are also different. Hence, we provide programmability for the connections between PUs. Specifically, we put a multiplexer at the input port and a demultiplexer at the output port for each PU, shown in Figure 5. In this way, multiple data paths are organized through the interconnection between these components. By controlling each node's multiplexer and demultiplexer, the data path of the whole chip can be controlled. For example, when executing the first sub-network of Figure 1(c), the bolded edges in Figure 1(e) are enabled, and others are disabled. The controlling signals of multiplexers and demultiplexers are generated by the code generation step in HCG system and transferred to the chip through an AXI-Lite interface.

B. Intra-PU Architecture Design

As introduced in Section III, HCG supports almost all commonly used layers, including convolution, pooling (max pooling, average pooling, and global pooling), element-wise addition, concatenation and non-linear activation functions such as ReLU. The implementation of depth-wise convolution PU and pooling PU are similar to the convolution PU, and other types of PUs are relatively straightforward to implement. Hence, the implementation details of PUs, except convolution PU, are omitted. Moreover, since the design of intra-PU architecture is not the focus of this paper and HCG is orthogonal to existing works about intra-PU optimization, we directly integrate HCG with an ordinary intra-PU design, which has been widely used in many related works [38, 39], to compute convolution layers. This intra-PU design adopts the output stationary dataflow [38]. Its pseudo-code for computation is listed in Algorithm 7. In the algorithm, H_i, W_i, and D_i are the input activation's height, width, and depth. F and K are filter number and kernel size of weight. InP and $OutP$ are the input and output parallelism factor of the convolution PU, respectively. Since these two parallelism factors are the optimization targets of intra-PU architecture design, which is not the focus of HCG, this paper directly sets both of them to 32. Figure 5 also shows the illustrative diagram of the convolution PU. As can be observed, the PU mainly computes the multiplication between an activation vector and a weight tile per cycle with a multiplier array. The column dimension of the array is the accumulative dimension, and adder trees perform the accumulation. The activation vector is shared along the row dimension. More details of the implementation of each sub-module can be found in [38]. Please note that HCG does not put any constraints on the intra-PU architecture design. Hence, it can be seamlessly integrated with other existing works on intra-PU optimization, no matter whether with intra-PU templates or without intra-PU templates.

C. On-chip Memory Usage

HCG uses BRAM to implement the activation buffer and weight buffer in convolution PU, depth-wise convolution PU, and pooling PU. Moreover, Algorithm 4 and Algorithm 6 use the function FPT to estimate the BRAM consumption for a given layer. According to the documentation provided by vendor [40], the maximum width of a BRAM is 72 bits, and the corresponding depth is 512. Therefore, the number of BRAMs needed by a buffer with the depth of D and transferring W bits per clock cycle is:

$$bram\ number = \left\lceil \frac{W}{72} \right\rceil \times \left\lceil \frac{D}{512} \right\rceil. \tag{2}$$

As shown in Figure 5, the W of activation buffer (W_a) and weight buffer (W_w) is ($InP \times Ibit$) and ($InP \times OutP \times Wbit$), respectively. Moreover, followed by the intra-PU design, the depth of activation buffer D_a and the depth of weight buffer D_w are:

$$D_a = K \times \lceil \frac{D_i}{InP} \rceil \times W_i. \tag{3}$$

$$D_w = K \times K \times \lceil \frac{D_i}{InP} \rceil \times \lceil \frac{F}{OutP} \rceil. \tag{4}$$

Moreover, due to the usual computation imbalance of branches in DNNs, the multiple inputs of concatenation and element-wise addition layers arrive at different times. This

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

TABLE I
HARDWARE RESOURCE OF BACKEND PLATFORMS.

FPGA	Process	LUT	DSP	On-chip Mem.	Off-chip Communication
KCU1500	20nm	397K	5520	9.95 MB	DDR4 & PCIe
VU9P	16nm	1182K	6840	43.24 MB	DDR4 & PCIe

[1] the on-chip memory of VU9P can be divided into Block RAM (9.49 MB) and UltraRAM (33.75 MB).

makes the concatenation and element-wise addition PUs fail to consume the input data immediately, thus possibly causing the execution failure of the whole DNN. To this end, we place a First-In-First-Out (FIFO) module in the earlier arrival port of every concatenation and element-wise addition PU. These FIFOs are also implemented with BRAMs, which also have a width of W_a.

D. Off-chip Memory Usage

HCG uses off-chip memory as the off-chip cache for both activation and weight. Before the execution of each sub-network, the required weights are firstly transferred to the corresponding on-chip weight buffers. Then, the input data of current sub-network is transferred to activation buffers and the execution begins. In practice, we use the DDR chips embedded on FPGA boards and CPU memory communicates through PCIe as off-chip memory. HCG is embedded with a program that allocates the activation and weights to different off-chip memory. For example, in Figure 1(e), while PU-4 transfers its output data to DDR 1, PU-1 simultaneously reads its input data from DDR 0.

VII. EVALUATION

In this section, we evaluate the effectiveness of HCG. We firstly evaluate the effectiveness of the proposed appearing-frequency guided pruning (introduced in Section V-A). Secondly, the effectiveness of appearing-frequency first generation (introduced in Section V-B) is verified. During its verification, we give a comprehensive analysis of the mismatched on-chip memory size. Thirdly, we compare HCG with the SOTA works on ADA or manually designed accelerators. Finally, since some evaluated DNNs have not been implemented on FPGA before, we compare the latency among HCG (FPGA), CPU and GPU.

A. Experimental Setup

HCG uses Verilog HDL to describe the generated accelerators. All experiments are operated on Xilinx Vivado 2020.1 with the default synthesis options. Moreover, we mainly choose KCU1500 as the backend platform. Since some existing works use VU9P as their backend, we also provide the results of HCG using VU9P for comparison. The information about these two FPGAs is shown in Table I. In terms of off-chip communication, KCU1500 provides four pieces of DDR4, which can be accessed through AXI full channels. The PCIe of KCU1500 can be configured as four parallel AXI-Stream channels, where each channel can reach the maximum bandwidth of $250MHz \times 256bits$. We configure both of these two FPGAs to operate at 200 MHz with 256 bits per clock cycle for each off-chip memory communication channel, which satisfies the physical bandwidth constraint. During the comparison among HCG, CPU and GPU, the CPU is Intel Xeon E5-2620 operating at 2.10 GHz, while the GPU is Nvidia TITAN XP with 12 GB GDDR5X and 3840 CUDA cores operating at 1.8 GHz. For both CPU and GPU, the DNNs are implemented with PyTorch 1.5.0.

Prior work [41] classifies DNNs into regularly and irregularly connected DNNs according to the regularity of connection between layers. ResNet [20] and RandWire [21] are the representatives of these two kinds of DNNs, respectively. The overall topology of ResNet [20] is the repetition of residual blocks. On the contrary, RandWire [21] does not show regularity in topology. Moreover, work in [41] finds that irregularly connected DNNs show higher accuracy than regularly connected ones with the same computation budget. Hence, we evaluate both regularly connected DNNs and irregularly connected DNNs. For regularly connected DNNs, we evaluate ResNet [20], DenseNet [42], GoogleNet [19] and Inception-V3 [43]. For irregularly connected DNNs, we evaluate RandWire [21] and AmoebaNet [22]. The input image size for all DNNs is $3 \times 224 \times 224$.

For evaluation metrics, we mainly consider the overall on-chip memory consumption and latency when comparing HCG with other works. Since the independent comparison of these two metrics cannot directly demonstrate the performance of HCG on on-chip memory utilization, a combined metric jointly considering latency and on-chip memory consumption is required. Hence, following the definition of widely used energy efficiency (image/s/W) [6] and DSP efficiency (image/s/DSP) [38], we use *On-chip Memory Efficiency* (image/s/MB) when comparing HCG with other works on FPGAs, which is defined as follows:

$$\textit{On-chip Memory Efficiency} = \frac{\beta}{\textit{Latency} \times \textit{On-chip Memory}} \quad (5)$$

The β balances the effect of data precision on overall on-chip memory consumption, where it is set to 1 and 2 for 8-bit and 16-bit precision, respectively. Equation (5) considers the latency and overall on-chip memory consumption at the same time and directly reflects the on-chip memory utilization.

B. The Effectiveness of Appearing-Frequency Guided Pruning

In this section, we study the effectiveness of appearing-frequency guided pruning from three aspects: the effectiveness on memory footprint distribution, the effectiveness on memory efficiency, and the effectiveness of model accuracy preservation.

As introduced in Section V-A, the pruning embedded in the HCG system mainly targets at those layers requiring infrequent memory footprints and prunes their memory footprints into frequent ones as much as possible. Figure 3 and Figure 4 illustrate the comparison of memory footprints for ResNet-50, Inception-V3, and AmoebaNet with and without pruning, respectively. As introduced in Section V-A, the memory footprints of layers become more concentrated after pruning. Here,

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

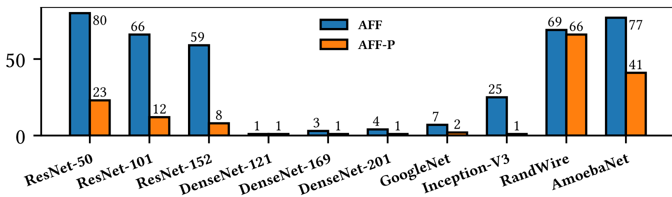


Fig. 6. The standard deviation comparison of memory footprints distribution of benchmark with the appearing-frequency guided pruning (AFF-P) and without the pruning (AFF).

we additionally use the standard deviation of layers' memory footprints as the criterion to show the effectiveness of the proposed pruning method over all benchmarks. Figure 6 shows the comparison results. The figure shows that our pruning method reduces the standard deviation of memory footprint by $0 \sim 96\%$ and averages at 60% across all benchmarks. Specially, DenseNet-121 fails to benefit from the pruning because its original memory footprints of layers are almost the same, which means there are no infrequent items in the model. Moreover, ResNet-50, ResNet-101 and ResNet-152 appear to be more sensitive to the pruning than other models. The reason is that these three models have more infrequent memory footprints and thus are pruned more drastically than others. Specifically, these three models are pruned with 50% of weights while others are only pruned with an average of 18% of weights. Furthermore, it is observed that infrequent memory footprints still exist in the pruned model since the standard deviation is larger than zero. This is mainly due to the topological limitation during pruning, which is introduced in Section V-A.

The experimental results also show that the appearing-frequency guided pruning can significantly improve memory efficiency. In particular, Table II shows the details of processing latency and hardware resources of the HCG with the proposed pruning (denotes AFF-P) and the HCG without pruning (denotes AFF). It shows that almost all pruned models occupy less memory but have lower processing latency than the original ones. The memory of ResNet-152 and Inception-V3 slightly increase after the pruning, mainly due to the increase in activation memory. Specifically, the pruning makes those infrequent memory footprints equal to the frequent ones, thus making more layers to be processed simultaneously in one single sub-network, which increases the activation memory but lowers the processing latency. Moreover, since the activation memory is relatively small compared with weights, it keeps consistent with the optimization goal of this paper. Figure 7 shows the memory efficiency comparison between HCG without pruning and HCG with pruning. From the figure, we can find that pruning enhances the memory efficiency by $0 \sim 102\%$ and averages 33% across all benchmarks. The analysis is similar to that of the effectiveness over memory footprint.

For the model accuracy preservation, we test the accuracy of all evaluated DNNs over CIFAR-100 dataset [44] before and after pruning. It shows that the accuracy of five DNNs slightly increases after pruning, including ResNet-101 $(+0.18\%)$, ResNet-152 $(+0.19\%)$, DenseNet-201 $(+0.28\%)$,

TABLE II
OVERALL RESOURCES AND LATENCY COMPARISON.

DNNs	Strategy	LUT (K)	DSP	On-Chip Memory (MB)	Latency (ms)
ResNet-50 [20]	AFF-P	309	5218	7.40 (78%)	3.41
	AFF	317	5218	7.90 (83%)	5.95
	EC	233	3134	9.44 (99%)	7.80
ResNet-101	AFF-P	314	5218	7.27 (76%)	5.71
	AFF	323	5218	7.53 (79%)	11.15
	EC	233	3134	9.48 (99%)	14.49
ResNet-152	AFF-P	315	5218	6.88 (72%)	8.39
	AFF	333	5218	6.84 (72%)	14.77
	EC	233	3134	9.45 (99%)	20.08
DenseNet-121 [42]	AFF-P	319	5226	6.65 (70%)	3.96
	AFF	319	5226	6.65 (70%)	3.96
	EC	319	5226	8.25 (87%)	4.02
DenseNet-169	AFF-P	318	5226	5.90 (62%)	4.775
	AFF	319	5226	6.25 (66%)	5.00
	EC	320	5226	8.72 (92%)	5.66
DenseNet-201	AFF-P	319	5226	5.87 (61%)	5.84
	AFF	321	5226	6.40 (67%)	6.37
	EC	295	4705	8.84 (93%)	8.29
GoogleNet [19]	AFF-P	322	5234	6.21 (65%)	4.39
	AFF	323	5234	6.27 (66%)	4.50
	EC	323	5234	8.28 (87%)	4.57
Inception-V3 [43]	AFF-P	328	5234	6.23 (65%)	9.36
	AFF	328	5234	6.17 (65%)	10.30
	EC	326	5234	8.93 (94%)	11.46
RandWire [21]	AFF-P	328	4929	6.06 (63%)	10.00
	AFF	357	5450	6.07 (64%)	9.99
	EC	327	4929	8.04 (85%)	10.23
AmoebaNet [22]	AFF-P	379	5250	6.05 (63%)	25.45
	AFF	396	5258	6.43 (68%)	30.53
	EC	253	3166	8.61 (91%)	55.58

[1] In the Strategy column, 'AFF-P' is the appearing-frequency first strategy with appearing-frequency guided pruning, 'AFF' is the appearing frequency first generation without appearing-frequency guided pruning, 'EC' is the equal chance strategy.

GoogleNet $(+1.05\%)$ and Inception-V3 $(+0.38\%)$. The remaining ratio of weight number after pruning of these five DNNs are respectively 54.06%, 56.88%, 79.86%, 89.93% and 71.54%. GoogleNet achieves the most significant accuracy increase while retaining the highest weight ratio. This is because the layers in GoogleNet generally have less weights than layers in other DNNs, making most of the layers in GoogleNet require frequent memory footprint and do not need pruning. Figure 6 supports this analysis, showing that the standard deviation of the memory footprint distribution for GoogleNet is relatively small. Specifically, only a single convolution layer and its downstream layer are pruned during the pruning of GoogleNet. At the same time, four DNNs slightly degrade in accuracy, including ResNet-50 (-0.13%), DenseNet-169 (-0.61%), RandWire (-0.65%) and AmoebaNet (-0.41%). The remaining ratio of weight number after pruning of these four DNNs are respectively 47.82%, 85.97%, 91.76% and 61.35%. RandWire retains the highest number of weights, which can be attributed to two main reasons. Firstly, RandWire incorporates a large number of lightweight depthwise layers that are ignored by our pruning method. Secondly, the complex architecture of RandWire imposes significant topology constraints when generating pruning groups during the pruning process. In other words, pruning a specific layer in RandWire necessitates pruning many more upstream and downstream layers, which forces the algorithm to select fewer layers for pruning. To the best of our knowledge, this degree of loss of accuracy is acceptable according to related literature

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

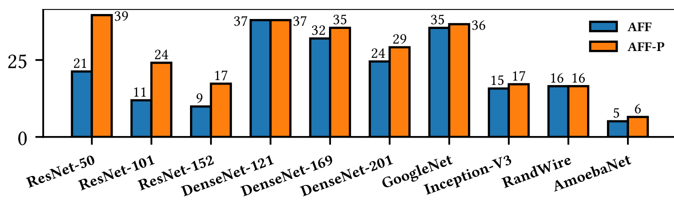


Fig. 7. The memory efficiency comparison of HCG with the appearing-frequency guided pruning (AFF-P) and without the pruning (AFF).

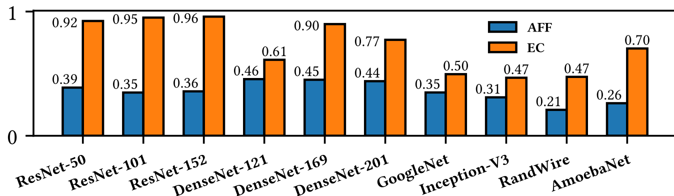


Fig. 8. The mismatched on-chip memory size (MB).

on pruning [27, 28, 29].

C. The Effectiveness of Appearing-Frequency First Generation

In this section, we study the effectiveness of the appearing-frequency first generation, which generates a basic set of HCG nodes (introduced in Section V-B). The strategy prioritizes those layers with the highest appearing-frequency during the basic node set generation process. For comparison, we adopt a baseline strategy that generates a basic HCG node set containing the same number of PUs as that of the appearing-frequency first generation but considers different layers with an equal chance. We name this baseline strategy *equal chance strategy*. We will compare the mismatched on-chip memory size, the overall on-chip memory size, and the latency between these two strategies.

Figure 8 shows the mismatched on-chip memory size comparison between the appearing-frequency first generation and the equal chance strategy. We can find that the mismatched on-chip memory size of the equal chance strategy is $1.33\times \sim 2.71\times$ that of the appearing-frequency first generation. According to the overall on-chip memory consumption comparison results shown in Table II, we can find that the equal chance strategy uses $1.19\times \sim 1.44\times$ more on-chip memory, compared with the appearing-frequency first generation.

From the latency comparison results shown in Table II, we can find that the latency of appearing-frequency first generation is better than the equal chance strategy for all evaluated DNNs. Moreover, the appearing-frequency first generation shortens the latency of AmoebaNet most significantly, where the latency is $1.82\times$ smaller than the equal chance strategy. This is because AmoebaNet is not only an irregularly connected model but also has the largest number of convolution layers among evaluated DNNs. The number of convolution layers directly impacts the PU allocation process since the convolution layer consumes far more arithmetic and memory resources than other layers.

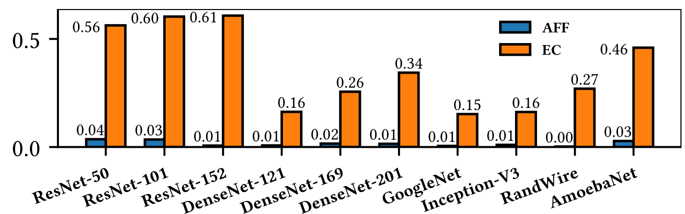


Fig. 9. The mismatched on-chip memory size (MB) caused by PU reuse.

In summary, the appearing-frequency first generation satisfies the computation requirements of the major part of the entire model, which improves the on-chip memory utilization and shortens the latency.

D. Mismatched On-Chip Memory Size Analysis

Since Section VII-C focuses on the overall comparison between appearing-frequency first generation and the equal chance strategy, it only compares the mismatched on-chip memory size in a shallow way. Therefore, this section gives a deeper analysis of the mismatched on-chip memory size. We will first introduce the causes and categories of the mismatched on-chip memory sizes and then analyze different categories of mismatched on-chip memory sizes.

The causes of the mismatch can be summarized as the reuse of PUs across different layers and the on-chip memory granularity of FPGA. For the reuse of PUs, since all sub-networks share the same list of PUs, one PU needs to compute different layers in the DNN. However, the variety of layers' requirements makes the on-chip memory size of PUs unable to satisfy the requirement of every layer without mismatch. For the granularity of FPGA, two ceiling operations in Equation (2) make the allocated BRAM always larger than the requirements, which causes the mismatch. For example, if both InP and $OutP$ are set to 32 and $Wbit$ is set to 8, the granularity of the weight buffer will be 114 BRAMs, which can accommodate $(114 \times 72 \times 512)$ bits. In summary, the mismatched on-chip memory size can be classified into two categories: 1) the mismatched on-chip memory size caused by PU reuse and 2) the mismatched on-chip memory size caused by FPGA's on-chip memory granularity.

The experimental results show that the mismatched on-chip memory size caused by FPGA's granularity accounts for 95.5% and 52.0% of the total mismatched on-chip memory size of the appearing-frequency first generation and the equal chance strategy, respectively. However, the on-chip memory granularity of FPGAs is mainly related to the hardware feature, which is determined by the FPGA vendor and not the focus of this paper, as in other existing works [3, 6, 16, 17]. Hence, in order to clearly demonstrate the ability of HCG to reduce the mismatch, Figure 9 shows the mismatch caused by PU reuse independently. We can find that HCG generally reduces the mismatch caused by PU reuse to a relatively low level, which is $0 \sim 1/15$ of the equal chance strategy. Moreover, it is worth mentioning that the mismatch caused by PU reuse in RandWire is reduced to zero because RandWire uses a large amount of lightweight depth-wise convolution.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

TABLE III
COMPARISON WITH EXISTING METHODS FOR REGULARLY CONNECTED DNNS.

Model	ResNet-50						
Category	Manual		ADA				
Method	[45]	Amoeba [46]	LCMM [3]	SushiAccel [47]	PipeFuser [48]	**HCG**	
Year	2024	2024	2019	2023	2024	2024	
FPGA	Virtex-7	Arria 10	VU9P	U50	U200	KCU1500	VU9P
Freq. (MHz)	150	200	180	100	220	200	200
DSPs	2160	522	5632	4740	3560	5218	6781
On-Chip Mem. (MB)	4.14	2.53	30.98 (*16b*)	5.37	5.47	7.40	9.28
Power (W)	10.87	8.18	-	-	-	10.59	11.81
Latency (ms)	30.64	31.25	6.46	- (922 GOP/s)	- (2160 GOP/s)	3.41 (3247 GOP/s)	2.77 (4312 GOP/s)
Energy Efficiency (image/s/W)[2]	4.00	3.91	-	-	-	27.69	30.57
On-Chip Mem. Efficiency (image/s/MB)[2]	10.52	12.65	11.10	- (41.50[3])	- (42.30)	39.63 (52.99)	38.90 (56.12)

Model	ResNet-152						
Category	Manual		ADA				
Method	[49]	[45]	LCMM	PipeFuser	**HCG**		
Year	2018	2024	2019	2024	2024		
FPGA	Arria 10 GX 1150	Virtex-7	VU9P	U200	KCU1500	VU9P	U200
Freq. (MHz)	200	150	180	220	200	200	200
DSPs	1518	2160	5694	4496	5218	6781	6781
On-Chip Mem. (MB)	5.77 (*16b*)	4.14	37.15	6.78	6.88	8.72	8.55
Power (W)	-	10.65	-	-	10.71	15.98	15.98
Latency (ms)	32.00	80.45	13.26	- (5415 GOP/s)	8.39 (4045 GOP/s)	6.80 (5137 GOP/s)	5.96 (6327 GOP/s)
Energy Efficiency (image/s/W)[2]	-	1.56	-	-	11.13	9.20	10.49
On-Chip Mem. Efficiency (image/s/MB)[2]	10.82	4.01	2.26	- (31.27)	17.32 (25.32)	16.86 (25.37)	19.62 (31.87)

Model	GoogleNet					DenseNet			
Category	Manual		ADA			Manual	ADA		
Method	[51]	Amoeba	LCMM	**HCG**		[52]	[17]	**HCG**	
Year	2023	2024	2019	2024		2020	2019	2024	
FPGA	U50	Arria 10	VU9P	KCU1500	VU9P	VX690T	4×VCU118	KCU1500	VU9P
Freq. (MHz)	200	200	180	200	200	200	200	200	200
DSPs	4100	522	5694	5234	6797	-	20492	5226	6789
On-Chip Mem. (MB)	15.45	2.53	38.05	6.21	7.30	0.67	20.52 (*16b*)	5.90	7.10
Power (W)	25.99	8.44	-	21.37	15.90	8.65	120.00	10.60	15.98
Latency (ms)	3.58	11.77	4.65	4.39	3.77	55.12	4.10	4.77	4.00
Energy Efficiency (image/s/W)	10.74	10.06	-	10.65	16.68	2.10	2.03	19.78	15.64
On-Chip Mem. Efficiency (image/s/MB)	18.08	33.59	6.28	36.68	36.34	26.98	23.77	35.53	35.21

Model	BERT-base (512 Tokens) [50]				
Category	Manual		ADA		
Method	FET-OPU [53]	CSTrans-OPU [54]	[55]	**HCG**	
Year	2023	2024	2024	2024	
FPGA	U200	U280	U280	KCU1500	VU9P
Freq. (MHz)	200	200	245	200	200
DSPs	4864	3840	1780	5130	6669
On-Chip Mem. (MB)	5.96	6.49	5.71	3.84	4.35
Power (W)	7.38	8.79	-	11.90	16.64
Latency (ms)	27.28	28.92	26.00	33.26	23.55
Energy Efficiency (image/s/W)	4.97	3.92	-	2.52	2.55
On-Chip Mem. Efficiency (image/s/MB)	6.15	5.33	5.50	7.83	9.76

1 The default data precision is 8-bit, and the results using 16-bit are specially marked.
2 To eliminate the effect of clock rate, latencies are normalized to operate with 200 MHz when calculating the on-chip memory and energy efficiency.
3 When comparing on-chip memory efficiency with SushiAccel and PipeFuser, the latency of a single image is calculated by the division between total operations of the model and the throughput (GOP/s).

TABLE IV
COMPARISON WITH EXISTING METHODS FOR IRREGULARLY CONNECTED DNNS.

Model	Design	FPGA	Freq. (MHz)	DSP	Latency (ms)	On-Chip Mem. (MB)
RandWire	RWNN [23]	U50	200	4648	16.60	15.68
	HCG	KCU1500	200	4929	9.99	6.06
	HCG	VU9P	200	4929	10.00	5.87

TABLE V
LATENCY COMPARISON WITH CPU AND GPU.

Model	Device	Freq.	Latency (ms)
AmoebaNet	Intel Xeon E5-2620	2.10 GHz	296.18
	Nvidia TITAN XP	1.8 GHz	62.96
	KCU1500	200 MHz	25.45

E. Comparison with SOTA Methods

This section compares HCG with existing SOTA methods for both regularly and irregularly connected DNNs. Specifically, for regularly connected DNNs, HCG is compared with both manually designed accelerators and other existing ADA methods. For irregularly connected DNNs, which are ignored by all of existing ADA methods, we compare HCG with manually designed accelerators or general-purpose processors such as CPU and GPU.

Table III shows the results for regularly connected DNNs. As can be observed, HCG generally exhibits a competitive latency when compared with manually designed accelerators for all DNNs. Specifically, for ResNet-50, ResNet-152 and DenseNet, HCG with both KCU1500 and VU9P achieve shorter latency than manually designed accelerators. For GoogleNet and BERT-base, HCG has longer latency than those manually designed accelerators with higher-end FPGAs in some cases. For example, HCG with KCU1500 is slower than FET-OPU [53] on BERT-base [50]. However, the latency is decreased when HCG uses VU9P as the backend, where FET-OPU is 16% slower. Moreover, evaluated by the aforementioned on-chip memory efficiency metric, our HCG with KCU1500 outperforms manually designed accelerators by an average of 2.00 times across all compared DNNs. With VU9P, HCG outperforms manually designed accelerators by an average of 2.13 times in on-chip memory efficiency.

Compared with existing ADA methods, HCG achieves competitive latency with less on-chip memory. Compared with LCMM [3], HCG with KCU1500 not only has shorter latency but also uses 5 times less on-chip memory for both ResNet-152 and GoogleNet. Moreover, when HCG uses the same FPGA as LCMM, HCG achieves lower latency with 2 to 3 times less on-chip memory than LCMM. Furthermore, throughput (GOP/s) is adopted for the estimation of latency when compared with SushiAccel [47] and PipeFuser [48], since these two works report throughput as speed metric. When evaluating throughput, images of a batch size are pipelined through accelerators, resulting in time overlaps between batched image processing. Although HCG mainly optimizes the latency of a single image, it also supports this batched image processing mechanism. For fairness of comparison, we directly set the batch size to 8 and evaluate the throughput. Specifically, for ResNet-50, HCG with KCU1500 and VU9P provides higher throughput than PipeFuser (with Alveo U200) and SushiAccel (with Alveo U50). Evaluated by on-chip memory efficiency, HCG with KCU1500 and VU9P respectively outperforms SushiAccel by 28% and 35% and outperforms PipeFuser by 25% and 33%.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

For ResNet-152, when HCG uses the same FPGA (Alveo U200) as PipeFuser, it achieves higher throughput and on-chip memory efficiency. For DenseNet, HCG achieves competitive latency with only one VU9P compared to the method in [17] with four VU9Ps. For BERT-base, HCG with KCU1500 provides a competitive latency with the method in [55] with Alveo U280. However, HCG only uses 67% on-chip memory of the method in [55], resulting in a 43% higher on-chip memory efficiency.

As shown in Table III, HCG achieves higher energy efficiency than existing methods for ResNet-50, ResNet-152, GoogleNet, and DenseNet. For the Transformer-based DNN BERT-base, however, FET-OPU and CSTrans-OPU exhibit higher energy efficiency. These two works, which are designed specifically for Transformer-based models, enhance DSP utilization and reduce latencies by leveraging specialized intra-PU designs. Enhancing DSP utilization helps reduce the energy consumed by DSP arrays, while reducing latencies directly boosts the energy efficiency metric. HCG aims to provide a general method applicable to various DNNs, including Transformer-based models. It focuses on inter-PU optimization, which is orthogonal to the intra-PU optimizations of FET-OPU and CSTrans-OPU. Nevertheless, HCG still achieves acceptable energy efficiency, especially considering the substantial computation requirements of BERT-base.

For irregularly connected DNNs, we only find a manually designed accelerator called RWNN [23] for RandWire. RWNN uses the Alveo U50 as the backend platform, which is dedicated to machine learning applications. The 8 GB HBM memory equipped on Alveo U50 can reach a bandwidth of 201 GB/s, far exceeding the DDR4 of KCU1500 and VU9P. Hence, to further verify the effectiveness of HCG, we also compare HCG (on KCU1500 and VU9P) with RWNN (on Alveo U50). The results are shown in Table IV. We can find that although the comparison is unfair for HCG, HCG is not only 1.6× faster in latency but also uses 2.5× fewer on-chip memory compared with RWNN.

Since AmoebaNet has not been implemented on FPGAs before, we compare the latency of HCG with KCU1500 to that of CPU and GPU, which is shown in Table V. Please note that HCG mainly considers the inference scenario, as almost all related works [2, 3, 4, 5, 6, 7]. In the inference scenario, the batch size of input samples is always set to one, and data is always reduced to relatively low precision. Hence, we compare the latency when the batch size is set to one. For the data precision, CPU and HCG use 8-bit precision for both activation and weight, while GPU uses full precision (32-bit) since PyTorch does not support 8-bit inference on TITAN XP GPU. Table V shows that HCG with KCU1500 is 11.6× and 2.5× faster in latency compared with CPU and GPU, respectively.

F. Scalability

HCG demonstrates superior scalability, making it adaptable to a wide range of DNN architectures and FPGA platforms. Firstly, Table II shows that HCG exhibits scalability across various DNN architectures. As observed, HCG exhibits outstanding performance for both regularly and irregularly connected DNNs. Secondly, Table II also highlights HCG's scalability in a similar DNN architecture with different weight scales. For example, HCG provides stable latencies for ResNet with three scales of weights (ResNet-50, ResNet-101, and ResNet-152), where latencies scale linearly with the number of weights. Moreover, there are mainly two requirements in the HCG exploration algorithm, including the information (such as kernel size, input channel, etc.) about each layer and its interconnection. Since DNNs are typically stacked by layers, these two requirements are satisfied for almost all DNNs, allowing HCG to be easily extended to other DNN architectures. Thirdly, Table III validates HCG's scalability across different FPGA platforms (KCU1500, VU9P, and U200), demonstrating HCG's adaptability to FPGA platforms with varying resource constraints. Furthermore, the HCG exploration algorithm is mainly constrained by resource constraints of a target FPGA, including the total number of on-chip memory, DSPs, off-chip memory, and LUTs. Given that these resources account for the primary differences between FPGA platforms, HCG can be applied to other FPGAs by adjusting the resource constraints accordingly.

VIII. Conclusion and Future Work

In this paper, we propose a novel method called HCG for ADA without inter-PU templates. Experiments show that our HCG method can achieve competitive latency with much less on-chip memory, compared with existing SOTA methods. In addition, HCG is orthogonal to existing works on intra-PU architecture optimization. Combining both inter-PU and intra-PU architecture optimization together will be pursued in our future work.

References

[1] J. Li, W. Wang, and W.-J. Li, "Hardware computation graph for dnn accelerator design automation without inter-pu templates," in *ICCAD*, 2022.

[2] X. Zhang, D. Wang, P. Chuang, S. Ma, D. Chen, and Y. Li, "F-CAD: A framework to explore hardware accelerators for codec avatar decoding," in *DAC*, 2021.

[3] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in fpga-based DNN accelerators via layer conscious memory management," in *DAC*, 2019.

[4] X. Lin, S. Yin, F. Tu, L. Liu, X. Li, and S. Wei, "LCP: A layer clusters paralleling mapping method for accelerating inception and residual networks on fpga," in *DAC*, 2018.

[5] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *ICCAD*, 2020.

[6] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. W. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for fpgas," in *ICCAD*, 2018.

[7] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *MICRO*, 2016.

[8] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "AMOS: Enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *ISCA*, 2022.

[9] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "TENET: A framework for modeling tensor dataflow based on relation-centric notation," in *ISCA*, 2021.

[10] Y. Lin, M. Yang, and S. Han, "NAAS: Neural accelerator architecture search," in *DAC*, 2021.

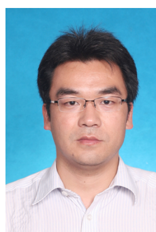
Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2025.3558132

[11] L. Jia, Z. Luo, L. Lu, and Y. Liang, "TensorLib: A spatial accelerator generation framework for tensor algebra," in *DAC*, 2021.
[12] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. J. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," in *DAC*, 2021.
[13] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *DAC*, 2020.
[14] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *ASPLOS*, 2020.
[15] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.
[16] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping dnn models to cloud fpgas," in *FPGA*, 2019.
[17] D. Wang, J. Shen, M. Wen, and C. Zhang, "An efficient design flow for accelerating complicated-connected cnns on a multi-fpga platform," in *ICPP*, 2019.
[18] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *ISCA*, 2017.
[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.
[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.
[21] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *ICCV*, 2019.
[22] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI*, 2019.
[23] R. Kuramochi and H. Nakahara, "An fpga-based low-latency accelerator for randomly wired neural networks," in *FPL*, 2020.
[24] X. Cai, Y. Wang, X. Ma, Y. Han, and L. Zhang, "DeepBurning-SEG: Generating dnn accelerators of segment-grained pipeline architecture," in *MICRO*, 2022.
[25] S. Li, X. Zhou, H. Lu, and K. Wang, "DNNMapper: An elastic framework for mapping dnns to multi-die fpgas," in *ISCAS*, 2024.
[26] K. Sheng-Chun, J. Geonhwa, and K. Tushar, "ConfuciuX: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning," in *MICRO*, 2020.
[27] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *ICLR*, 2017.
[28] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *CVPR*, 2019.
[29] G. Fang, X. Ma, M. Song, M. B. Mi, and X. Wang, "Depgraph: Towards any structural pruning," in *CVPR*, 2023.
[30] P. Morì, M.-R. Vemparala, N. Fasfous, S. Mitra, S. Sarkar, A. Frickenstein, L. Frickenstein, D. Helms, N. S. Nagaraja, W. Stechele *et al.*, "Accelerating and pruning cnns for semantic segmentation on fpga," in *DAC*, 2022.
[31] H. Fan, T. Chau, S. I. Venieris, R. Lee, A. Kouris, W. Luk, N. D. Lane, and M. S. Abdelfattah, "Adaptable butterfly accelerator for attention-based nns via hardware and algorithm co-design," in *MICRO*, 2022.
[32] Z. Zhou, X. Duan, and J. Han, "A design framework for generating energy-efficient accelerator on fpga toward low-level vision," *TVLSI*, 2024.
[33] E. Luo, H. Huang, C. Liu, G. Li, B. Yang, Y. Wang, H. Li, and X. Li, "DeepBurning-MixQ: An open source mixed-precision neural network accelerator design framework for fpgas," in *ICCAD*, 2023.
[34] J. Plochaet and T. Goedemé, "Hardware-aware pruning for fpga deep learning accelerators," in *CVPR*, 2023.
[35] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019.
[36] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "AccPar: Tensor partitioning for heterogeneous deep learning accelerators," in *HPCA*, 2020.
[37] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations," in *ISCA*, 2020.
[38] H. Khan, A. Khan, Z. Khan, L. B. Huang, K. Wang, and L. He, "NPE: An fpga-based overlay processor for natural language processing," in *FPGA*, 2021.
[39] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y. W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," in *DAC*, 2017.
[40] *UltraScale Architecture Memory Resources*, Xilinx, Inc., 9 2021.
[41] B. H. Ahn, J. Lee, J. M. Lin, H. P. Cheng, J. Hou, and H. Esmaeilzadeh, "Ordering Chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," in *MLSys*, 2020.
[42] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017.
[43] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.
[44] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.
[45] Z. Liu, Q. Liu, S. Yan, and R. C. Cheung, "An efficient fpga-based depthwise separable convolutional neural network accelerator with hardware pruning," *TRETS*, 2024.
[46] X. Wu, M. Wang, J. Lin, and Z. Wang, "Amoeba: An efficient and flexible fpga-based accelerator for arbitrary-kernel cnns," *TVLSI*, 2024.
[47] P. Behnam, A. Tumanov, T. Krishna, P. Gadikar, Y. Chen, J. Tong, Y. Pan, A. R. Bambhaniya, and A. Khare, "Subgraph stationary hardware-software inference co-design," in *MLSys*, 2023.
[48] X. Zhou, S. Li, H. Lu, and K. Wang, "PipeFuser: Building flexible pipeline architecture for dnn accelerators via layer fusion," in *ASP-DAC*, 2024.
[49] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *TVLSI*, 2018.
[50] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019.
[51] S. Hong, Y. F. Arthanto, J.-Y. Kim *et al.*, "Accelerating deep convolutional neural networks using number theoretic transform," *TCAS-I*, 2022.
[52] S. Zeng and Y. Huang, "A hybrid-pipelined architecture for fpga-based binary weight densenet with high performance-efficiency," in *HPEC*, 2020.
[53] Y. Bai, H. Zhou, K. Zhao, H. Wang, J. Chen, J. Yu, and K. Wang, "FET-OPU: A flexible and efficient fpga-based overlay processor for transformer networks," in *ICCAD*, 2023.
[54] Y. Bai, K. Zhao, Y. Liu, H. Wang, H. Zhou, X. Wu, J. Yu, and K. Wang, "CSTrans-OPU: An fpga-based overlay processor with full compilation for transformer networks via sparsity exploration," in *DAC*, 2024.
[55] H. Chen, J. Zhang, Y. Du, S. Xiang, Z. Yue, N. Zhang, Y. Cai, and Z. Zhang, "Understanding the potential of fpga-based spatial acceleration for large language model inference," *TRETS*, 2024.

Jun Li received his B.Eng. degree from the School of Information Science and Engineering, Southeast University in 2019. He is currently pursuing his Ph.D. degree at the School of Computer Science, Nanjing University, under the supervision of Prof. Wu-Jun Li. His research interests include acceleration of machine learning over FPGA and GPU.

Wei Wang received his BS and MS degree from the ESE Department of Nanjing University in 1997 and 2000, respectively. He received Ph.D. degree from the ECE department of National University of Singapore in 2008. He joined the Computer Science and Technology Department of Nanjing University in 2012. Before that, he has worked at Microsoft Research Asia as an associate researcher in 2009. His research interests are in the areas of wireless networking, including Device-free Sensing, Software Defined Radio, and Mobile Cellular Networks.



Wu-Jun Li received the B.Sc. and M.Eng. degrees in computer science from Nanjing University, China, and the Ph.D. degree in computer science from The Hong Kong University of Science and Technology. He started his academic career as an Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. He then joined Nanjing University, where he is currently a Professor with the School of Computer Science. His research interests are in machine learning, big data and artificial intelligence.

Authorized licensed use limited to: Nanjing University. Downloaded on April 08,2025 at 09:04:49 UTC from IEEE Xplore. Restrictions apply.
© 2025 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted,
but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.