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

Jun Li, Wei Wang and Wu-Jun Li*
National Key Laboratory for Novel Software Technology
Collaborative Innovation Center of Novel Software Technology and Industrialization
Department of Computer Science and Technology, Nanjing University, China
lijun@smail.nju.edu.cn, ww@nju.edu.cn, liwujun@nju.edu.cn

ABSTRACT

Existing deep neural network (DNN) accelerator design automation (ADA) methods adopt architecture templates to predetermine parts of design choices and then explore the left design choices beyond templates. These templates can be classified into intra-PU templates and inter-PU templates according to the architecture hierarchy. 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. Experiments show that HCG method can achieve competitive latency while using only 1.4× ∼ 5× fewer on-chip memory, compared with existing state-of-the-art ADA methods.

KEYWORDS

DNN Accelerator; FPGA; Accelerator Design Automation; Hardware Computation Graph

1 INTRODUCTION

To deploy deep neural network (DNN) with high throughput and low energy consumption, there has appeared a growing interest in designing accelerators with Field Programmable Gate Array (FPGA). However, manually coding for FPGA makes designers suffer from significant engineering difficulty and a huge space of design choices. Hence, many DNN accelerator design automation (ADA) methods [10, 12, 15, 21, 28–31] have been proposed to automate this process. 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.

Since the involved design choice space during the accelerator designing process is too large to explore, existing ADA methods adopt architecture templates to predetermine part of design choices and then explore the left design choices beyond templates. According to the architecture hierarchy at the processing unit (PU) level, these templates can be classified into intra-PU templates and inter-PU templates. Intra-PU templates predetermine some design choices in one single PU, such as the dataflow style and the interconnection between processing elements (PE) 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 without templates has become an important research topic. Some recent works [7, 11, 12, 31] try to enhance the flexibility of ADA by removing intra-PU templates and developing more expressive representation for DNN accelerators. For example, NAAS [11] removes intra-PU templates with the modeling of the connectivity between PEs, which enhances the flexibility of ADA. As for the inter-PU level, almost all existing works adopt inter-PU templates for ADA. For example, works in [3, 7, 12, 25, 26, 31] adopt *layer sequential template* which schedules and computes the DNN layer by layer on the chip. Besides, works in [2, 15, 16, 20, 27, 29] adopt *layer pipelined template* which maps the entire DNN model over the accelerator. Furthermore, with the widespread usage of multi-branch structure [18] and skip connection [4], the topology of DNN has become more and more complex. ADA with predetermined inter-PU templates is typically inefficient in terms of resource utilization, especially for DNNs with complex topology [20, 21].

To improve the flexibility of ADA and the 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

*Wu-Jun Li is the corresponding author.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549342

[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.

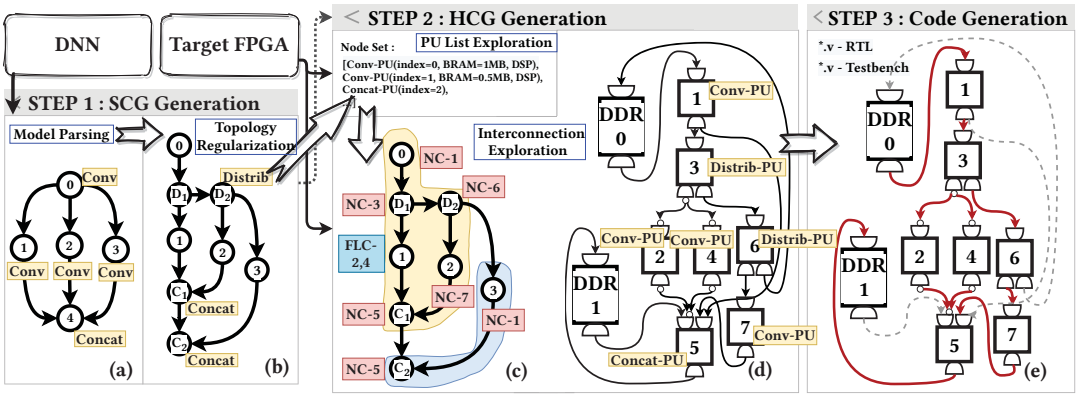

Figure 1: Flowchart of our HCG method. (a) an illustrative example SCG; (b) regularized SCG derived from (a), where the circle with D inside denotes distribution layer, and the circle with C inside denotes concatenation layer; (c) the sub-network partition and PU allocation scheme derived from *Interconnection Exploration* step. From the figure, the SCG is partitioned into 2 sub-networks which are distinguished by different colors. The PUs allocated to each layer and their cooperation type are attached next to the nodes. For example, layer 1 needs to conduct filter level cooperation (FLC) and PUs allocated to layer 1 are PU-2 and PU-4. Other layers in this example need not to conduct 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 3.1; (e) the state of HCG corresponding to the first sub-network (layer 0, 1, 2) in (c). The red bolded lines denote the data path of the sub-network.

on-chip memory utilization. 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 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 [23] and AmoebaNet [13], 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. For regularly connected DNNs, we evaluate ResNet [4], DenseNet [5], GoogleNet [18] and Inception-V3 [19]. For irregularly connected DNNs, we evaluate RandWire [23] and AmoebaNet [13]. Experimental results show that for regularly connected DNNs, HCG can achieve competitive speed (latency) while using 1.4× ~ 5× fewer on-chip memory, compared with existing state-of-the-art ADA methods. Since there have not existed ADA works considering irregularly connected DNNs, we compare HCG with a manually designed accelerator [9] 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 [9].

2 SYSTEM OVERVIEW

As shown in Figure 1, HCG consists of three steps: software computation graph (SCG) generation, HCG generation and code generation. We will introduce these three steps in turn.

SCG generation includes two sub-steps: model parsing and topology regularization. Firstly, model parsing converts external DNN model description files into SCG defined in the system. Currently, the layers of DNN supported by HCG include convolution, depth-wise convolution, pooling (max pooling, average pooling and global pooling), concatenation, element-wise addition and non-linear activation functions such as ReLU. Moreover, HCG can be easily extended to support new layer types. 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 represented as multiple nodes with only two inputs or outputs. Figure 1 (b) shows the regularized SCG topology derived from Figure 1 (a). The function of Distrib node D_1 and D_2 in Figure 1 (b) is to distribute the input data to two output ports.

HCG generation mainly depends on an inter-PU architecture exploration strategy, which consists of two sub-steps: PU list exploration and interconnection exploration. Firstly, the PU list exploration generates a list of basic PUs according to the appearing frequency of layers in SCG. Then the basic PU list is iteratively tuned according to the resource constraints and guidance from SCG. Secondly, the interconnection exploration 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 3.3.

Code generation converts HCG and PU allocation result to synthesizable register transfer level (RTL) code and runtime 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 [2, 15, 27, 29] can be used, this paper mainly focuses on HCG generation, which will be introduced in the following section.

3 HCG GENERATION

This section introduces the details of HCG generation, which mainly depends on an inter-PU architecture exploration strategy. We first introduce HCG representation, which is the basis of the exploration strategy. Then we propose two guidelines for HCG generation. Finally, the algorithm of exploration strategy is introduced.

3.1 HCG Representation

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, which will be specifically explained in this section.

In the inter-PU architecture, PUs are organized to cooperate, and one layer can be computed with multiple PUs according to its computation requirement. The common method for the PUs' cooperation is partitioning data to different PUs and processing partitioned data in different PUs [6, 14, 17]. However, unlike these existing works with an inter-PU template that predetermine the cooperation methods between PUs and partition data in a fixed way [6, 14, 17], 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). Firstly, 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[2] 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 small activations with the shape of ($Depth \times Height \times Width_*, \Sigma Width_* = Width$). Each small activation is assigned to a PU for computation. Secondly, 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 ($filterNum_* \times Depth \times kernelSize \times kernelSize, \Sigma filterNum_* = filterNum$), which will be transferred to multiple PUs. Figure 1 (c) and Figure 1 (d) give an illustrative example for FLC. In the figure, PUs allocated to layer 1 are PU-2 and PU-4, which cooperate through FLC. Hence, the weight data of layer 1 will be sliced into two blocks which will be uploaded to PU-2 and PU-4, respectively. Thirdly, 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 PUs which are allocated to two adjacent layers.

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

3.2 Guidelines for HCG Generation

The process of HCG generation 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 1. The constraints refer to the hardware resource limitation of target FPGA, including the limitation of on-chip memory, off-chip memory, arithmetic resource (DSP) and logical resource (LUT).

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).

$$\begin{aligned} &\sum_{i=1}^{n_{layer}} \frac{\sum_{j=1}^{m_i} PU_j.ocm - layer_i.footprint}{n_{layer}} \\ &s.t. \sum_{j=1}^{m_i} PU_j.ocm - layer_i.footprint \geq 0 \end{aligned} \tag{1}$$

In Equation (1), n_{layer} is the total number of layers, m_i is the number of PUs allocated to $layer_i$, $PU_j.ocm$ is the on-chip memory size of the j_{th} PU allocated to $layer_i$, $layer_i.footprint$ is the on-chip memory footprint of $layer_i$. Moreover, the numerator of Equation (1) represents the mismatched on-chip memory size of $layer_i$. If the numerator is larger than zero, the allocated on-chip memory size exceeds the footprint of $layer_i$, which results in waste of on-chip memory. If the numerator is zero, the on-chip memory size allocated to $layer_i$ matches its footprint without wasted on-chip memory. Hence, Equation (1) summarizes the wasted on-chip memory of the input DNN.

Based on Equation (1), the ideal case between layers and PUs is that every layer can find a PU generated according to its footprint for computation, which minimizes the mismatched on-chip memory size. Moreover, generating an inter-PU architecture without an inter-PU template suggests that this ideal case should be divided into two ideal subcases. Firstly, there should exist PUs that are generated according to a layer's footprint. Secondly, the PU generated according to a layer's 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 footprint. On one hand, this generation method is not practicable for every layer considering that HCG does not adopt any inter-PU template, and 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

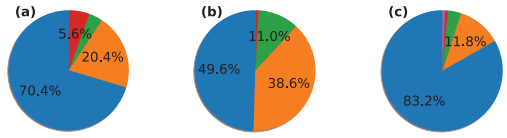

Figure 2: The distribution of PUs' on-chip memory sizes when using the ideal generation method for every layer in ResNet-50 (a), RandWire (b) and AmoebaNet (c). Different colors in a single chart represent the on-chip memory size of different PUs, and the percentage is the ratio between the number of PUs with the same on-chip memory size to the total number of PUs.

Algorithm 1 Stage 1 of PU List Exploration

Input: SCG; total number of DSPs, on-chip memory of the target FPGA : (DSP_T, OCM_T);
Output: Basic PU list: $BasicPU_{List}$;
1: $BasicPU_{List} \leftarrow$ Empty List;
2: $tmpPU_{List} \leftarrow$ Empty List;
3: **for** $layer \in SCG$ **do**
4: Generate a $newPU$, $newPU.ocm = \mathcal{F}(layer.footprint)$;
5: $tmpPU_{List}$.append($newPU$);
6: **end for**
7: Cluster the PUs in $tmpPU_{List}$ with their on-chip memory size, and then count the appearing frequency of each PU cluster;
8: $maxFreq \leftarrow$ the max frequency of PU clusters;
9: $maxFreqPU \leftarrow$ a PU in the PU cluster with the max frequency;
10: $(DSP_{req}, OCM_{req}) \leftarrow$ Resource-Calculation($maxFreqPU$);
11: Append $\min(\left\lfloor \frac{DSP_T \times maxFreq}{DSP_{req}} \right\rfloor, \left\lfloor \frac{OCM_T \times maxFreq}{OCM_{req}} \right\rfloor)$ $maxFreqPU$s to $BasicPU_{List}$;

appear most frequently. This is motivated by the observation that many PUs are equipped with the same on-chip memory size if the ideal generation method is adopted for every layer. Figure 2 supports this observation by showing the distribution of PUs' on-chip memory sizes when the ideal generation method is adopted for every layer. We can find that there always exists an apparent part occupying the majority of each pie chart. That is to say, classified by on-chip memory size, there always exists a PU cluster that appears more frequently than others. Based on the observation, HCG applies the ideal generation method to those layers corresponding to the PU cluster that appears most frequently. This guideline is reflected in an appearing frequency first strategy, which will be introduced in Section 3.3.1.

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 footprints should be prioritized. This guideline is reflected in a PU allocation strategy with priority, which will be introduced in Section 3.3.1.

3.3 Inter-PU Architecture Exploration

Following the guidelines in Section 3.2, the inter-PU architecture exploration strategy generates an inter-PU architecture for a given DNN, which consists of two steps: PU list exploration and interconnection exploration.

Algorithm 2 Stage 2 of PU List Exploration

Input: Basic PU list: $BasicPU_{List}$; SCG; 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 iterates from 0 to max_length **do**
4: cand $\leftarrow SCG$[start : start+i];
5: $Alloc_m \leftarrow$ **PU-Allocation**(cand, PU_{List}); //PU-Allocation( ) denotes the function in Algorithm 3.
6: $(OCM_m, DSP_m, OFM_m) \leftarrow$ Resource-Calculation($Alloc_m$);
7: **if** $(OCM_m, OFM_m) < (OCM_T, OFM_T)$ **then**
8: $DSP_c \leftarrow DSP_m$; $Alloc_c \leftarrow Alloc_m$; $end \leftarrow end + 1$;
9: **else break**;
10: **end for**
11: $newPU \leftarrow$ (the new PU derived from $Alloc_c$);
12: **if** $DSP_c < DSP_T$ **then**
13: // Append a new PU to the PU_{List}.
14: $PU_{List}.append(newPU)$;
15: **else**
16: // Fuse the new PU to the smallest PU inside the PU_{List}.
17: $PU_{sm} \leftarrow$ (the PU with smallest on-chip memory in PU_{List});
18: $PU_{sm}.ocm += newPU.ocm$;
19: **if** end = len(SCG) **then return** ; // End of the input DNN.
20: **else** start $\leftarrow$ end;
21: **end while**

3.3.1 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.

Stage 1 is based on the appearing frequency first strategy. Algorithm 1 shows the process. The strategy first temporally generates a PU for every layer according to its footprint (line 2 ~ 6). Function $\mathcal{F}$ calculates the on-chip memory size occupied by the input footprint. For example, if block random access memory (BRAM) is used to accommodate the footprint, $\mathcal{F}$ calculates the number of BRAM consumed by the input footprint and returns the capacity of consumed BRAMs. We will introduce the calculation method of $\mathcal{F}$ in Section 4.3. 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 7). 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 9). Finally, after calculating hardware resources of one single primary element (line 10), the algorithm appends as many primary elements as possible to the basic PU list under the constraints of hardware resources (line 11). 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, which reduces 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 2 shows

Algorithm 3 PU Allocation Strategy with Priority

Input: $Layer_{List}$ (A sub-network); PU_{List};
Output: D (a dictionary, layer → PU).
1: // *[PUs of layer.type]* is to aggregate PU from PU_{List} of the layer.type
2: // **STEP 1**: Prioritized allocation
3: np_layers ← Empty List;
4: **for all** layer ∈ $Layer_{List}$ **do**
5: D[layer] ← PRIO-ALLOC(*FootPrint*(layer), *[PUs of layer.type]*);
6: PU_{List}.delete(D[layer]);
7: **if** D[layer] == **None then** np_layers.append(layer);
8: **end for**
9: // **STEP 2**: Search the PU list to minimize the mismatch
10: **for all** layer ∈ np_layers **do**
11: D[layer] ← MIN-MISMATCH(*FootPrint*(layer), *[PUs of layer.type]*);
12: PU_{List}.delete(D[layer])
13: **end for**
14: **function** PRIO-ALLOC(fpt, PUs)
15: **if** $\exists PUs_i \in PUs, s.t. PUs_i.ocm = \mathcal{F}(fpt)$ **then return** PUs_i;
16: **else return None**
17: **function** MIN-MISMATCH(fpt, PUs)
18: **if** $\Sigma[PUs].ocm \geq fpt$ **then**
19: **return** $[PUs_i], min(\Sigma PUs_i.ocm), s.t. \Sigma PUs_i.ocm \geq fpt$;
20: **else return** $[newPU, PUs], \Sigma[newPU, PUs].ocm = fpt$;

the process. In each iteration, the algorithm first generates a sub-network candidate which comprising a continuous part of DNN (line 4). Then the PUs allocated for the candidate sub-network is 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 3. 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 are considered in priority (STEP 1 in Algorithm 3). Function PRIO-ALLOC (line 5) 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, in which *FootPrint* is used to calculate the footprint of a given 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 3). Function MIN-MISMATCH returns *a list of PUs* with respect to the condition at line 19. 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 which cooperate through FLC or IWC method. Moreover, as the allocation process goes on, PUs are progressively consumed (line 6 and 12). 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 20).

After PU allocation, the sub-network with a maximum length is chosen under hardware resource constraints (line 3 ∼ 5 in Algorithm 2). 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 14). 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 16 ∼ 18).

With Algorithm 2, we can get different inter-PU architectures under different resource constraints. More specifically, when the resources are sufficient to put all layers on chip, the strategy in HCG will generate an architecture which is the same as that generated with a *layer pipelined template* [2, 15, 16, 20, 27, 29]. When the resources are so rare that only one PU can be placed on chip, the strategy in HCG will generate an architecture which is the same as that generated with a *layer sequential template* [3, 7, 12, 25, 26, 31]. 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.

3.3.2 Interconnection Exploration. After 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 3.1, this strategy first conducts 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.

4 HARDWARE IMPLEMENTATION

This section introduces the inter-PU infrastructure, the design of intra-PU architecture and on-chip memory usage. It should be noted that this paper mainly focuses on the optimization at the inter-PU architecture level which is orthogonal to the intra-PU architecture. We leave the joint optimization of inter-PU and intra-PU architectures to future work.

4.1 Inter-PU Infrastructure

As introduced in Section 3.1, there are nine interconnection types between PUs. These interconnection types uncover two organizational requirements from the inter-PU level, which require a flexible inter-PU infrastructure. The first organizational requirement is to support different sub-networks with one shared HCG. Since the topology and PUs allocation results 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, which is shown in Figure 3. 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 computing the first sub-network of Figure 1 (c), the bolded edges in Figure 1 (e) are enabled and others are

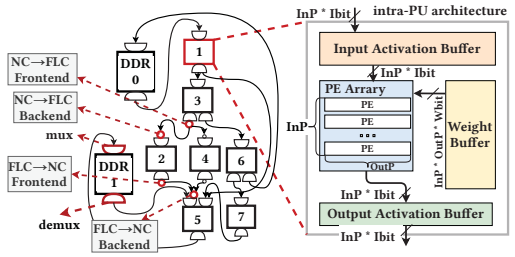

Figure 3: Inter-PU infrastructure and intra-PU design.

disabled. The second requirement is to deal with the nine interconnection types between PUs. To solve this problem, 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 3 shows an example. In Figure 1 (c), *layer 1* needs FLC and PUs allocated to it are PU-2 and PU-4, and *layer* C_1 needs NC. Hence, in Figure 3, a FLC → NC frontend is placed at the output ports of PU-2 and PU-4, and a FLC → NC backend is placed at the input port of PU-5.

4.2 Intra-PU Architecture Design

As introduced in Section 2, HCG supports convolution, pooling (max pooling, average pooling and global pooling), concatenation, element-wise addition 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 [8, 22], to compute convolution layers. Figure 3 shows the diagram of the convolution PU and the implementation of every sub-modules can be found in [8]. 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.

4.3 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 1 and Algorithm 3 need the estimation of BRAM consumption for a given footprint. According to the documentation provided by vendor [24], 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 $Depth$ and transferring $Width$ bits per clock cycle is:

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

As shown in Figure 3, the $Width$ of activation buffer and weight buffer is $InP \times Ibit$ and $InP \times OutP \times Wbit$, repsectively.

5 EVALUATION

In this section, we evaluate the effectiveness of HCG. We firstly evaluate the effectiveness of the proposed appearing frequency first strategy (introduced in Section 3.3). Secondly, we give a comprehensive analysis of the mismatched on-chip memory size. Thirdly, we compare HCG with the state-of-the-art 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.

5.1 Experimental Setup

The generated accelerator of HCG is described by Verilog, which is synthesized by 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 compare HCG using VU9P with existing works. In terms of off-chip communication, KCU1500 has 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 [1] classifies DNNs into regularly and irregularly connected DNNs according to the regularity of connection between layers. ResNet [4] and RandWire [23] are the representatives of these two kinds of DNNs, respectively. The overall topology of ResNet [4] is the repetition of residual blocks. On the contrary, RandWire [23] does not show regularity in topology. Moreover, work in [1] 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 [4], DenseNet [5], GoogleNet [18] and InceptionV3 [19]. For irregularly connected DNNs, we evaluate RandWire [23] and AmoebaNet [13]. 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 power efficiency (image/s/Watt) [29] and DSP efficiency (image/s/DSP) [8], we use *On-chip Memory Efficiency* (image/s/MB) when comparing HCG with other works on FPGAs.

$$On\text{-}chip\ Memory\ Efficiency = \frac{\beta}{Latency \times On\text{-}chip\ Memory} \tag{3}$$

The β balances the effect of data precision on overall on-chip memory consumption, where it is set to 1 and 2 for 8 bits and 16 bits

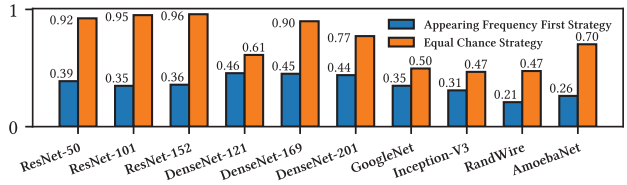

Figure 4: The mismatched on-chip memory size (MB).

precision, respectively. Equation (3) considers the latency and overall on-chip memory consumption at the same time and directly reflects the on-chip memory utilization.

5.2 The Effectiveness of Appearing Frequency First Strategy

In this section, we study the effectiveness of the appearing frequency first strategy, which generates a basic set of HCG nodes (introduced in Section 3.3). The strategy prioritizes those layers with the highest appearing frequency during the basic node set generation process. For comparison, we adopt a baseline strategy which generates a basic PU list containing the same number of PUs as that of the appearing frequency first strategy but considers 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 latency between these two strategies.

Figure 4 shows the mismatched on-chip memory size comparison between the appearing frequency first strategy and the equal chance strategy. We can find that the mismatched on-chip memory size of the equal chance strategy is 1.33× ∼ 2.71× that of the appearing frequency first strategy. According to the overall on-chip memory consumption comparison results shown in Table 1, we can find that the equal chance strategy uses 1.19× ∼ 1.44× more on-chip memory, compared with the appearing frequency first strategy.

From the latency comparison results shown in Table 1, we can find that the latency of appearing frequency first strategy is better than the equal chance strategy for all evaluated DNNs. Moreover, the appearing frequency first strategy shortens the latency of AmoebaNet most significantly, where the latency is 1.82× faster than the equal chance strategy. It 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.

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

5.3 Mismatched On-Chip Memory Size Analysis

Since Section 5.2 focuses on the overall comparison between appearing frequency first strategy 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

Table 1: Resources and latency comparison between the appearing frequency first strategy and the equal chance strategy.

DNNs	Strategy	LUT (K)	DSP	On-Chip Memory (MB)	Latency (ms)
ResNet-50 [4]	AFF	317	5218	7.90 (83%)	5.95
	EC	233	3134	9.44 (99%)	7.80
ResNet-101	AFF	323	5218	7.53 (79%)	11.15
	EC	233	3134	9.48 (99%)	14.49
ResNet-152	AFF	333	5218	6.84 (72%)	14.77
	EC	233	3134	9.45 (99%)	20.08
DenseNet-121 [5]	AFF	319	5226	6.65 (70%)	3.96
	EC	319	5226	8.25 (87%)	4.02
DenseNet-169	AFF	319	5226	6.25 (66%)	5.00
	EC	320	5226	8.72 (92%)	5.66
DenseNet-201	AFF	321	5226	6.40 (67%)	6.37
	EC	295	4705	8.84 (93%)	8.29
GoogleNet [18]	AFF	323	5234	6.27 (66%)	4.50
	EC	323	5234	8.28 (87%)	4.57
Inception-V3 [19]	AFF	328	5234	6.17 (65%)	10.30
	EC	326	5234	8.93 (94%)	11.46
RandWire [23]	AFF	357	5450	6.07 (64%)	9.99
	EC	327	4929	8.04 (85%)	10.23
AmoebaNet [13]	AFF	396	5258	6.43 (68%)	30.53
	EC	253	3166	8.61 (91%)	55.58

[1] In the Strategy column, 'AFF' is the appearing frequency first strategy, 'EC' is the equal chance strategy.

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 the requirements of layers 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 (114×72×512) 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; 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 strategy 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 [2, 20, 21, 29]. Hence, in order to clearly demonstrate the ability of HCG to reduce the mismatch, Figure 5 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 ∼ 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.

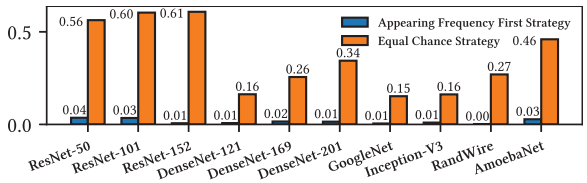

Figure 5: The mismatched on-chip memory size (MB) caused by PU reuse.

Table 2: Comparison with state-of-the-art ADA methods.

Model	Method	Freq	DSP	Latency (ms)	On-Chip Memory (MB)
ResNet-50	Cloud-DNN [2]	214	5489	8.12	34.88 (*16b*)
	LCMM [21]	180	5632	6.46	30.98 (*16b*)
	HCG-K	200	5218	5.95	7.90
	HCG-V	200	6781	4.78	9.87
ResNet-152	LCMM	180	5694	13.26	37.15
	HCG-K	200	5218	14.77	6.84
	HCG-V	200	6781	12.32	9.49
GoogleNet	Multi-FPGA [20]	200	19972	4.10	24.70 (*16b*)
	LCMM	180	5694	4.65	38.05
	HCG-K	200	5234	4.50	6.27
	HCG-V	200	6789	3.86	8.34
DenseNet	Multi-FPGA [20]	200	20492	4.10	20.52 (*16b*)
	HCG-K	200	5226	5.00	6.25
	HCG-V	200	6797	4.49	7.87

[1] HCG-K uses KCU1500 as backend, while HCG-V uses VU9P.
[2] The default data precision is 8-bit, and the results using 16-bit are specially marked.

Table 3: Comparison with state-of-the-art manual designs.

Model	Design	Freq	DSP	Latency (ms)	On-Chip Memory (MB)
RandWire	RWNN [9]	200	4648	16.60	15.68
	HCG-K	200	5450	9.99	6.08
	HCG-V	200	5450	10.02	6.17

5.4 Comparison with State-of-the-Art Methods

This section compares HCG with existing state-of-the-art ADA methods for both regularly and irregularly connected DNNs. However, to the best of our knowledge, there have not existed ADA works considering irregularly connected DNNs. Hence, for irregularly connected DNNs, this section will compare HCG with the manually designed accelerator or general-purpose processors such as CPU and GPU.

The state-of-the-art ADA methods for the comparison with HCG include LCMM [21], Cloud-DNN [2] and Multi-FPGA [20]. Both LCMM and Cloud-DNN use one VU9P, while Multi-FPGA uses four VU9Ps. The results are listed in Table 2. We can find that HCG achieves comparable latency with much less on-chip memory compared with other methods. For ResNet-152 and GoogleNet, HCG-K has slightly longer latency but uses 5× fewer on-chip memory compared with LCMM. Moreover, when HCG uses the same FPGA as LCMM, it (denoted as HCG-V in Table 2) achieves lower latency with 2× ∼ 3× fewer on-chip memory compared with LCMM. For both GoogleNet and DenseNet, HCG achieves competitive latency with only one VU9P compared to Multi-FPGA with four VU9Ps.

For irregularly connected DNNs, we only find a manually designed accelerator called RWNN [9] for RandWire. RWNN uses

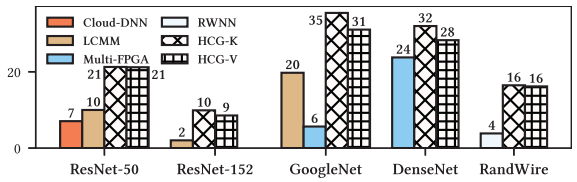

Figure 6: *On-chip Memory Efficiency* (image/s/MB) comparison with state-of-the-art designs on FPGA.

Table 4: Latency comparison with CPU and GPU.

Model	Device	Frequency	Latency (ms)
AmoebaNet	Intel Xeon E5-2620	2.10 GHz	296.18
	Nvidia TITAN XP	1.8 GHz	62.96
	KCU1500	200 MHz	27.91

the Alveo U50 as the backend platform, which is dedicated for 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 3. We can find that although the comparison is unfair for HCG, HCG is not only 1.3× faster in latency but also uses 2.5× fewer on-chip memory compared with RWNN.

Moreover, to further compare HCG with all the above mentioned FPGA accelerators, we use the combined metric named on-chip memory efficiency as introduced in Section 5.1. In Figure 6, we can find that the on-chip memory efficiency of HCG is 1.17× ∼ 5.5× higher than existing FPGA works.

Finally, 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 4. It should be noted that HCG mainly considers the inference scenario, as almost all related works [10, 15, 21, 28–30]. 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 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 GPU currently. Table 4 shows that HCG with KCU1500 is 10× and 2× faster in latency compared with CPU and GPU, respectively.

6 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 state-of-the-art ADA 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.

7 ACKNOWLEDGEMENTS

This work is supported by National Key R&D Program of China (No. 2020YFA0713901) and NSFC Projects (No. 61921006, No. 62192783).

REFERENCES

[1] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. 2020. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices. In *Conference on Machine Learning and Systems (MLSys)*.

[2] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. CloudDNN: An Open Framework for Mapping DNN Models to Cloud FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[3] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert J. Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. 2021. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. In *Design Automation Conference (DAC)*.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[5] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[6] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. 2020. Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations. In *International Symposium on Computer Architecture (ISCA)*.

[7] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *Design Automation Conference (DAC)*.

[8] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. 2021. NPE: An FPGA-Based Overlay Processor for Natural Language Processing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[9] Ryosuke Kuramochi and Hiroki Nakahara. 2020. An FPGA-Based Low-Latency Accelerator for Randomly Wired Neural Networks. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[10] Xinhan Lin, Shouyi Yin, Fengbin Tu, Leibo Liu, Xiangyu Li, and Shaojun Wei. 2018. LCP: A Layer Clusters Paralleling Mapping Method for Accelerating Inception and Residual Networks on FPGA. In *Design Automation Conference (DAC)*.

[11] Yujun Lin, Mengtian Yang, and Song Han. 2021. NAAS: Neural Accelerator Architecture Search. In *Design Automation Conference (DAC)*.

[12] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *International Symposium on Computer Architecture (ISCA)*.

[13] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence (AAAI)*.

[14] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *International Symposium on Microarchitecture (MICRO)*.

[15] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *International Symposium on Microarchitecture (MICRO)*.

[16] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *International Symposium on Computer Architecture (ISCA)*.

[17] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Deguang Wang, Junzhong Shen, Mei Wen, and Chunyuan Zhang. 2019. An Efficient Design Flow for Accelerating Complicated-Connected CNNs on a Multi-FPGA Platform. In *International Conference on Parallel Processing (ICPP)*.

[21] Xuechao Wei, Yun Liang, and Jason Cong. 2019. Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management. In *Design Automation Conference (DAC)*.

[22] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu Wing Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *Design Automation Conference (DAC)*.

[23] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. 2019. Exploring Randomly Wired Neural Networks for Image Recognition. In *International Conference on Computer Vision (ICCV)*.

[24] Xilinx, Inc. 2021. *UltraScale Architecture Memory Resources*. Xilinx, Inc.

[25] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN AcceleratorO. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In *Design Automation Conference (DAC)*.

[27] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *International Conference on Computer-Aided Design (ICCAD)*.

[28] Xiaofan Zhang, Dawei Wang, Pierce Chuang, Shugao Ma, Deming Chen, and Yuecheng Li. 2021. F-CAD: A Framework to Explore Hardware Accelerators for Codec Avatar Decoding. In *Design Automation Conference (DAC)*.

[29] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-Mei W. Hwu, and Deming Chen. 2018. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *International Conference on Computer-Aided Design (ICCAD)*.

[30] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wenmei Hwu, and Deming Chen. 2020. DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-Based DNN Accelerator. In *International Conference on Computer-Aided Design (ICCAD)*.

[31] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction. In *International Symposium on Computer Architecture (ISCA)*.