Latency-Driven Design for FPGA-based Convolutional Neural Networks

Stylianos I. Venieris
Department of Electrical and Electronic Engineering
Imperial College London
Email: stylianos.venieris10@imperial.ac.uk

Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
Email: christos-savvas.bouganis@imperial.ac.uk

***Abstract*—In recent years, Convolutional Neural Networks (ConvNets) have become the quintessential component of several state-of-the-art Artificial Intelligence tasks. Across the spectrum of applications, the performance needs vary significantly, from high-throughput image recognition to the very low-latency requirements of autonomous cars. In this context, FPGAs can provide a potential platform that can be optimally configured based on different performance requirements. However, with the increasing complexity of ConvNet models, the architectural design space becomes overwhelmingly large, asking for principled design flows that address the application-level needs. This paper presents a latency-driven design methodology for mapping ConvNets on FPGAs. The proposed design flow employs novel transformations over a Synchronous Dataflow-based modelling framework together with a latency-centric optimisation procedure in order to efficiently explore the design space targeting low-latency designs. Quantitative evaluation shows large improvements in latency when latency-driven optimisation is in place yielding designs that improve the latency of AlexNet by $73.54\times$ and VGG16 by $5.61\times$ over throughput-optimised designs.**

I. Introduction

Since its breakthrough in 2006, the field of Deep Learning has led to successes in a variety of Artificial Intelligence tasks. At the forefront of Deep Learning lies the model of Convolutional Neural Networks (ConvNets) with state-of-the-art accuracy in a broad range of applications spanning from object recognition to natural language processing [1]. To support the Deep Learning community in terms of computing infrastructure, several frameworks have been released which enable faster experimentation and development of Deep Learning models. Tools such as Caffe[1], Torch[2] and TensorFlow[3] offer high-level APIs and provide high-performance execution of ConvNets by employing high-throughput, power-costly GPUs as their main computational platform. In this context, FPGAs constitute a potential alternative in the form of a low-power reconfigurable platform that can provide tunable trade-offs between critical system-level factors such as throughput, latency, resource cost and power consumption. Nevertheless, to integrate FPGAs with the existing Deep Learning ecosystem, there is a need for tools that automate the mapping of ConvNets onto FPGAs in a principled, portable and scalable manner.

At the same time, a new and fast emerging application domain for Deep Learning and ConvNets are latency-critical systems. With the prominence of self-driving cars and Unmanned Aerial Vehicles, autonomous systems collect data from multiple sensors and process them using Deep Learning techniques in order to perceive their surroundings. Ultimately, the output of a trained ConvNet would yield an action, such as automatic reaction to road traffic. For such decision-making to happen in time, the computing system has to provide minimum latency overhead. To achieve this, the Deep Learning system has to be explicitly designed and optimised with low latency as the primary objective in place of high throughput.

Drawing from the fact that high throughput that is achieved by batch processing is not an indicator of low latency, this work presents a latency-driven methodology for the mapping of the inference task of ConvNets on FPGAs. By building upon the modelling framework of fpgaConvNet [2], the design space of FPGA-based ConvNets is extended and the exploration method enhanced in order to target emerging latency-critical applications. The key contributions of this paper are the following:

- The introduction of a new and flexible architecture that is tailored to a given ConvNet. The proposed architecture extends the ConvNet SDF model with a weights reloading transformation. With weights reloading adding another design dimension, the space of designs described by the SDF model is larger than existing work and includes significantly lower latency design points that so far could not be reached. This in turn allows the proposed flow to result in better design points in terms of latency than previously possible.
- An optimisation framework explicitly aiming for the generation of low-latency ConvNet designs. The developed optimiser traverses the design space by utilising the defined set of transformations over the SDF model and guides the exploration by minimising an objective function that focuses on the system's latency. Moreover, a heuristic is introduced as a technique of pruning the architectural search space and efficiently exploring the Pareto-optimal surface.

II. Background

A. Convolutional Neural Network Components

A typical ConvNet is organised in two stages: the feature extractor and the classifier [1]. The feature extractor's most commonly employed layer types are the convolutional, nonlinear and pooling layers, while the classifier typically includes fully-connected layers. The convolutional layer performs the main feature extraction, attempting to detect useful features

[1]http://caffe.berkeleyvision.org/
[2]http://torch.ch/, http://pytorch.org/
[3]https://www.tensorflow.org/

from its inputs. It processes a number of 2D feature maps, and convolves each of them with a bank of kernels. The weights of the kernels are learned in the training phase. For a convolutional layer of N_{in} feature maps at the input and N_{out} at the output, its operation can be expressed as:

$$\boldsymbol{f}_i^{out} = \sum_{j=1}^{N_{in}} \boldsymbol{f}_j^{in} * \boldsymbol{k}_{i,j} + \boldsymbol{b}_i, \quad \text{with} \quad i \in [1, N_{out}] \qquad (1)$$

where $\boldsymbol{f}_j^{in}$ and $\boldsymbol{f}_i^{out}$ are the j_{th} input and i_{th} output feature maps respectively, $\boldsymbol{k}_{i,j}$ is the $(K_h \times K_w)$ kernel that corresponds to the j_{th} input and i_{th} output and $\boldsymbol{b}_i$ is the i_{th} bias vector. The nonlinear layer applies an activation function pixelwise to a series of feature maps, where typical activation functions are $sigmoid$, $tanh$ and $ReLU$. The pooling layer aims to summarise the values of feature maps in a predefined neighbourhood typically using either their average or max. With the feature extractor dominating the computational cost of ConvNets, in this work we focus on the feature extractor stage.

B. Synchronous Dataflow

The modelling core in this work is based on the Synchronous Dataflow (SDF) computation model [3]. SDF captures a parallel system as a directed SDF graph (SDFG), with nodes representing computations and with arcs in place of data streams between them. SDF's basic principles are (1) the data-driven streaming operation where each node fires whenever data are available and (2) the *synchronous* property, where the number of consumed inputs and produced outputs of each node are known a priori at compile time. The strength of SDF comes from the ability to construct static execution schedules for the target system and obtain finite and predictable amount of buffer storage between its computing units. Furthermore, SDF enables the utilisation of graph theory and linear algebra to enhance the analytical strength and efficiency of our design space exploration method.

C. Modelling Framework

For the rest of this section, we present the modelling infrastructure, which builds upon and enhances [2]. This includes the high-level modelling of ConvNet applications, the internal representation of the hardware, the FPGA platform model and the design space exploration approach.

D. Application Model

Following the ConvNet application model presented in [2], a ConvNet workload is represented as a chain of layers in the form of a directed acyclic graph (DAG). Each node in the DAG represents a parametrised ConvNet layer and is associated with a tuple of parameters. Currently, we focus on the computation-intensive ConvNet feature extractor and hence the convolutional, nonlinear and pooling layers are considered, which are the ones that have been most commonly used in the ConvNet literature. The local response normalisation (LRN) layer is omitted due to its limited utilisation in most recent ConvNets [4][5]. The convolutional layer tuple below illustrates how the application model is populated:

$$< K_h, K_w, S, P, N >$$

where K_h and K_w are the height and width of each filter, S is the stride which defines the step between successive convolution windows, P is the zero-padding and N is the number of filters in the filter bank. For the rest of the layers, we follow the modelling as detailed in [2].

E. ConvNet Hardware Design Points as SDFGs

At a hardware level, *design points* are represented as SDF graphs that can execute the input ConvNet. Given a ConvNet's DAG, each node is mapped to a sequence of hardware building blocks. An SDFG is formed by assigning one SDF node to each block. The nodes of the SDFG are connected via arcs which carry data between blocks. Following SDF theory, the SDFG can be represented in a compact form by the topology matrix $\boldsymbol{\Gamma}$. Each column of $\boldsymbol{\Gamma}$ corresponds to a node and each row to an arc in the SDFG. In the case of data production, an element $\boldsymbol{\Gamma}(a, n)$ is a positive real number and indicates the production rate of node n at arc a. Similarly, in the case of data consumption, $\boldsymbol{\Gamma}(a, n)$ is a negative real number. Under this scheme, the topology matrix is decomposed into the elementwise product of three matrices as shown below:

$$\boldsymbol{\Gamma} = \boldsymbol{S} \odot \boldsymbol{C} \odot \boldsymbol{R} \qquad (2)$$

where

- $\boldsymbol{S}$ is the *streams matrix*. Each element holds the number of parallel streams at each connection.
- $\boldsymbol{C}$ is the *channels matrix*. Each element holds the width of each stream in *words* and its sign indicates the direction of the data flow.
- $\boldsymbol{R}$ is the *rates matrix*. Each element is a real number in the interval $[0, 1]$ which specifies the normalised data production or consumption rate of each node at each arc per cycle, with a maximum of 1 firing/cycle.

F. Mapping Layers to Building Blocks

A uniform representation is adopted to model a set of building blocks by means of a parametrised tuple template:

$$< param, s_{in}, s_{out}, c_{in}, c_{out}, r_{in}, r_{out} >$$

where $param$ is a set of block-specific configuration parameters, s_{in} and s_{out} are the number of parallel streams at the block's input and output respectively, c_{in} and c_{out} are the number of words per stream at the block's input and output respectively, r_{in} is the consumption rate, which is interpreted as the initiation rate, in *consumptions/cycle* and r_{out} is the production rate in *productions/cycle*. Moreover, each building block is associated with an empirical *resource consumption model*. To estimate the resource consumption of design points, place-and-route resource results were used to construct linear regression models as a function of each block's tunable parameters.

An architecture is constructed by instantiating a chain of blocks with specific parameters. Such an architecture constitutes a design point that is represented by its SDFG with its performance and resource consumption. The building blocks parametrisation scheme enables the expression of different implementations for each block with potentially different performance-resource characteristics. Leveraging SDF theory and its mathematical properties, the parameters of an SDFG's building blocks are tuned by means of algebraic operations directly over matrix $\boldsymbol{\Gamma}$ and in this way the design space exploration task is performed in an efficient manner.

Fig. 1 shows the mapping of a convolutional layer with N_{in} input and N_{out} output feature maps to a series of blocks. The *sliding window block* at the input of the layer receives the input feature maps as a stream of data elements and outputs a stream of $(K_h \times K_w)$ windows with a stride of S. This block consumes at a rate of 1 data element/cycle and produces at a

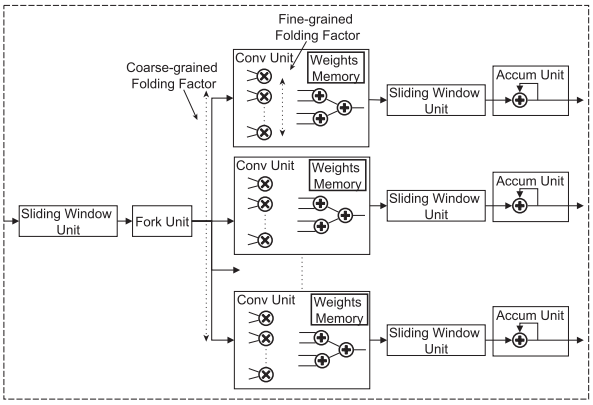

Fig. 1: Convolutional Layer Building Blocks

rate of $\frac{1}{S}(K_h \times K_w)$ elements/cycle. The next building block is the *fork unit*, which receives the $(K_h \times K_w)$ windows and copies them to N output streams. After the fork operation, a *filter bank* with N convolution units follows. Each of the units in the bank is responsible for $\lceil \frac{N_{out}}{N} \rceil$ output feature maps for a total of $\lceil \frac{N_{out}}{N} \rceil N_{in}$ convolutions per unit. Each unit performs a dot product operation between $(K_h \times K_w)$ windows and the weights of a filter of the same size which are stored in the unit's dedicated local memory. The rate of operation depends on the number of coarse convolution units instantiated, spanning from 1 up to N_{out} as well as on the fine-grained dot product implementation inside each unit.

Controlling the number of units as well as the dot product implementation exposes two degrees of freedom to traverse the performance-resource space. After the filter bank, sliding window units follow to extract the appropriate data elements from the results which are then summed together by the *accumulator units*. The accumulators perform the outer sum over the input feature maps in Eq. (1). To illustrate how a building block populates the tuple template, the convolution filter bank is shown below.

$$< \{N, K_h, K_w, f\}, N, N, K_h \times K_w, 1, f, f >$$

where $N \in [1, N_{out}]$ is the number of coarse convolution units and $f \in \left[\frac{1}{K_h \times K_w}, 1\right]$ is the fine-grained folding factor of the dot product implementation. The rates of each convolution unit depend on the fine-grained folding factor, f. When $f < 1$, the unit time-multiplexes its MACC resources to compute a dot product, trading-off performance for fewer resources.

G. Target Platform Model

The primary factor that constraints the *feasible space* of design points on a particular FPGA-based platform is the limited resources. Based on the formulation of [2], we utilise a resource vector, $\boldsymbol{rsc}_{Avail.}$ to capture the available FPGA resources and the characteristics of the off-chip memory:

$$\begin{aligned} \boldsymbol{rsc}_{Avail.} = [&DSP_{Avail.}, LUT_{Avail.}, FF_{Avail.}, \\ &BRAM_{Avail.}, B_{mem}, C_{mem}]^\top \end{aligned} \tag{3}$$

where B_{mem} and C_{mem} are the measured off-chip memory bandwidth and capacity respectively.

H. Design Space Exploration

The design space exploration (DSE) method uses the performance and resource models to traverse the design points described by the tunable parameters of the building blocks. The SDF modelling of ConvNet workloads enables the utilisation of analytical tools to facilitate this task. A mathematical optimiser can be employed to use a set of *transformations* over the SDFG, expressed as algebraic operations on $\mathbf{\Gamma}$, to explore the design space. So far, the objective of the optimiser was the maximisation of the throughput with respect to the resource constraints of the target FPGA. In [2], a set of three transformations was defined: *graph partitioning with reconfiguration*, *coarse-* and *fine-grained folding*.

1) Graph Partitioning with Reconfiguration: Directly mapping the original SDFG to a fully parallel hardware implementation would assume that the target FPGA has the required amount of on-chip memory and computational resources. With state-of-the-art ConvNet models reaching new records in terms of depth [4][5], the on-chip memory requirements can scale rapidly. *Graph partitioning with reconfiguration* exploits the reconfigurability of FPGAs to partition the ConvNet along its depth. With this transformation, the SDFG is split into subgraphs and each subgraph is mapped to a different architecture. Each architecture is specifically optimised for the particular subgraph and can utilise all of the FPGA resources. This transformation requires the reconfiguration of the whole FPGA whenever data have to enter a different subgraph which adds a reconfiguration time overhead. The amortisation of the reconfiguration is achieved by processing several inputs as a *batch*. In scenarios where throughput is the primary metric of interest and the latency of a single input is not crucial, the transformation can provide high-throughput designs.

2) Coarse- and Fine-Grained Folding: The *coarse-grained folding* transformation controls the parallelism of the coarse operations at each ConvNet layer. Coarse-grained folding treats the unroll factor over the output feature maps of each layer as a tunable parameter and is defined for all types of layers. The *fine-grained folding* transformation controls the implementation of the dot product operations inside convolution and average pooling units. These two transformations provide control points in order to balance the off-chip memory bandwidth, the computational resources and the exploited parallelism.

III. Latency-Driven Design

The existing methodology yields design points that are appropriate for high-throughput applications. The partitioning of a ConvNet by means of reconfiguration enables the generation of a distinct architecture for each subgraph. Each architecture is further optimised using coarse- and fine-grained folding and is effectively tailored to the workload of its subgraph. In high-throughput applications, full FPGA reconfigurations can be amortised by processing a large enough batch of inputs. Nevertheless, in latency-critical applications, the batch size is small and high-throughput cannot be achieved. Moreover, the latency of a single input is severely deteriorated due to reconfiguration independently of the batch size.

In order to target low-latency applications, a latency-driven design methodology is proposed. Latency optimisation is addressed at two levels of the existing modelling framework. First, from a modelling and design space perspective, a new transformation is introduced under the name *weights reloading*, which removes the need for FPGA reconfiguration and allows the execution of several SDF subgraphs without adding a severe penalty on latency. Second, from an optimisation perspective, an enhanced latency-centric optimiser is developed

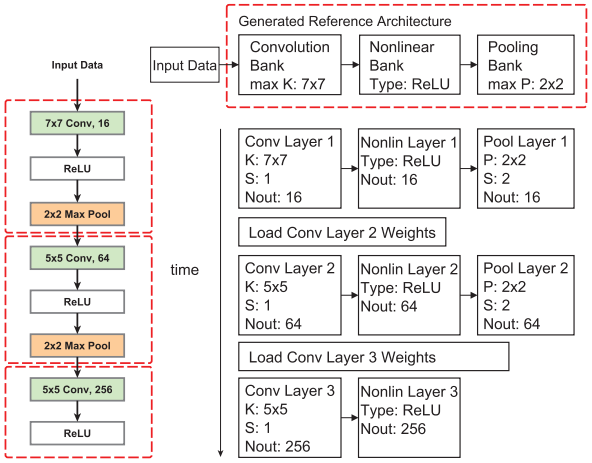

Fig. 2: Weights Reloading Example

with an objective function designed to guide the DSE so that the overall latency of the system is minimised.

A. *Weights Reloading Transformation*

The aim of the weights reloading transformation is to provide a method of executing several subgraphs without adding a penalty on latency due to FPGA bitstream-level reconfiguration. Similarly to graph partitioning with reconfiguration, this transformation partitions the SDFG into several subgraphs along its depth. However, instead of generating a distinct, fixed architecture for each subgraph, a single flexible architecture is generated that is capable of executing the workloads of all the subgraphs by operating in different modes.

When data have to enter a new subgraph, the subgraph's weights are loaded from the off-chip into the on-chip memory as an intermediate step and multiplexers are configured appropriately to form the correct datapath. The overhead of loading weights between subgraphs is a factor of the number of weights and the bandwidth of the system and is much smaller than the FPGA reconfiguration time[4]. Fig. 2 illustrates an instance of how weights reloading is applied. In this case, the SDFG of the ConvNet on the left is partitioned at two points, after the first and second pooling layers. In this way, three subgraphs are formed, each with a number of trained weights for its convolutional layer. A reference architecture is derived based on the layer patterns of the subgraphs as detailed in Section III-A3 and each subgraph is scheduled for execution. Moreover, reloading of weights is performed between the execution of successive subgraphs.

After the derivation of the reference architecture, the design is further optimised by means of coarse- and fine-grained folding. These two transformations are utilised to holistically tune the design by considering the workloads of all the scheduled subgraphs. Overall, the weights reloading transformation enables the tackling of latency-critical applications and expands the existing design space with design points that in principle resemble flexible processors. The main differentiating factor of our approach from existing programmable ConvNet processors is that the reference design is formed based on the target ConvNet, which avoids the overheads of a generic architecture, and after the reference design has been formed, the rest of the SDF transformations are utilised to further tailor the hardware to the scheduled subgraphs.

[4]For Zynq XC7Z045, the measured average time for full FPGA reconfiguration is approximately 600 ms. The weights reload time for a VGG16 layer with 4.5 MB of weights is approximately 2.2 ms which is lower by 272.7×.

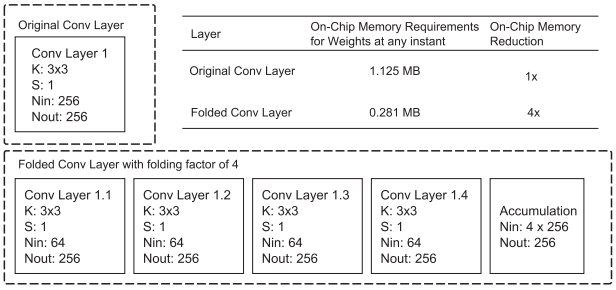

Fig. 3: Input Feature Maps Folding Example (assuming 16-bit fixed-point precision for the weights)

1) Input Feature Maps Folding: With state-of-the-art ConvNet models increasing in complexity and size [4][5], the weights storage requirements of even a single large convolutional layer can exceed the on-chip memory capacity of the target FPGA, making the ConvNet on-chip memory bounded. To deal with this issue and accommodate networks with a large amount of weights, an *input feature maps folding factor* is introduced, coupled to weights reloading. One input feature maps folding factor is associated with each convolutional layer. Based on the value of the folding factor, a convolutional layer is divided into several subgraphs that perform a fraction of the necessary convolutions. For a factor of f_{in}, each of the f_{in} subgraphs performs $\frac{N_{in}}{f_{in}} N_{out}$ convolutions and computes N_{out} intermediate convolution results. After the subgraphs' execution, an accumulation of the intermediate results takes place in order to produce the output feature maps.

Fig. 3 shows an example of a convolutional layer with 256 input and 256 output feature maps, folded by a factor of 4. Each subgraph computes 256 intermediate feature maps and the final accumulation subgraph sums them to produce the output feature maps. With this strategy, the on-chip storage requirements at any instant are reduced by a factor of 4 at the expense of having to load the next set of weights from the off-chip memory between successive subgraphs.

Similar approaches for addressing the limited on-chip memory have been employed by existing work. Nevertheless, our adoption of SDF, with all the advantages that it offers, poses a challenge compared to generic programmable accelerators by necessitating that input feature maps folding conforms to the streaming principle of operation. To maintain the benefits of streaming, weights reloading with input feature maps folding is designed to schedule the sum of Eq. (1) in f_{in} separate groups with an accumulation stage at the end, all of which can be executed in a streaming manner by a single reference architecture. Finally, this transformation can be applied to any convolutional layer with no restriction on the kernel size and the number of input and output feature maps.

2) Weights Reloading Design Space: The design parameters offered by this transformation are the selection of the partition points and the input feature maps folding factor for each convolutional layer in the resulted subgraphs. The first step of weights reloading is the detection of candidate partition points in the original SDFG. We introduce a prior to the selection of the partition points by allowing only the convolutional layers to be valid candidates. In ConvNet's feature extractor stage, convolutional layers have the highest requirement of on-chip memory in order to store their weights. In the case of very wide layers, accommodating several such layers in the same architecture can become infeasible because of the limited on-chip memory. The aim of this prior is to sustain the utilisation

of the on-chip memory that is allocated to convolution building blocks high across different subgraphs. In addition, the prior contributes to an informed exploration strategy and prunes the design space by rejecting subgraphs with only nonlinear or pooling layers as inefficient. The nonlinear layer is less resource-expensive and can be added after other layer types without much additional expense. Moreover, the pooling layer does not have trained parameters and thus does not reuse the on-chip memory.

As far as the size of the design space is concerned, for a ConvNet with N_L layers and N_{CONV} convolutional layers, an uninformed exploration would have to search among $N_L - 1$ candidate partition points leading to 2^{N_L-1} possible partitionings. By introducing the prior, the candidate partition points are decreased to $N_{CONV} - 1$. As an example of the pruning effect of the prior, the uninformed partitioning space of CIFAR-10 [6] with $N_L = 9$ and $N_{CONV} = 3$ contains 256 partitionings. After adding the prior, the possible partitionings become 4 with a $64\times$ reduction in the search space. For VGG16, the uninformed space of 10^9 partitionings is pruned down to 4096 with a reduction of $262144\times$. The prior prunes the large search space and contributes to the efficiency and speed of exploration by removing suboptimal design points.

3) Reference Architecture Derivation: After the partition points have been selected, a single, flexible reference architecture is derived that is able to execute all the subgraphs. We cast the reference design derivation as a pattern matching problem. After the SDFG is partitioned, the layer patterns, their relative position and their parameters in each subgraph are identified and a design is generated that contains all the necessary blocks with runtime configurable interconnections among them. At any time instant, the reference design can be configured with the appropriate datapath based on the current subgraph's id. Each node in a subgraph is mapped and scheduled on the appropriate building block of the generated architecture. The convolution and pooling units of the architecture are instantiated so that they can process windows of the maximum size that has been scheduled on them, with zero-padding used for smaller windows. After the derivation of the reference architecture, coarse- and fine-grained folding are used to further optimise it and holistically tune it by considering the workloads of all the scheduled subgraphs.

B. Performance Model

The SDF-based performance model presented in [2] is adopted and extended to represent the workloads of ConvNets and the performance of design points. A ConvNet's workload is represented in the form of a *workload matrix* $\boldsymbol{W}$. The i_{th} column of $\boldsymbol{W}$ holds the total number of data elements to be consumed by the i_{th} building block in the SDFG. Given a design point represented by topology matrix $\boldsymbol{\Gamma}$, the initiation interval of the architecture is given by the maximum element II^{max} of the *initiation interval matrix* $\boldsymbol{II}$ calculated as:

$$\boldsymbol{II} = \boldsymbol{W} \oslash \boldsymbol{\Gamma}$$

The execution time for a batch of B inputs is estimated as:

$$t(B, \boldsymbol{\Gamma}, \boldsymbol{W}) = \frac{1}{clock\ rate} \cdot (D + II^{max} \cdot (B - 1)) \quad (4)$$

where D is the maximum between the size of the input, e.g. the size of an image, and the pipeline depth of the SDFG.

Graph partitioning with reconfiguration determines the number of distinct architectures of a design point. For $N_P - 1$ reconfiguration partition points, there are N_P distinct architectures where the i_{th} architecture is associated with its $\boldsymbol{\Gamma}_i$ and $\boldsymbol{W}_i$ matrices. With the inclusion of the weights reloading transformation, $\boldsymbol{W}_i$ is partitioned into N_{W_i} workload subgraphs, indexed by j. Each of the N_{W_i} subgraphs will be scheduled for execution on a single derived reference architecture represented by $\boldsymbol{\Gamma}_{i,ref}$. The columns of each $\boldsymbol{W}_{i,j}$ have to be appropriately aligned and scheduled so that they map correctly on the blocks of the reference architecture. The workload alignment is performed algebraically on the SDF model and is elaborated in Section III-C. Between consecutive subgraphs, the weights transfer time from the off-chip to the on-chip memory of the j_{th} workload subgraph of the i_{th} architecture has to be included and is denoted by $t_{i,j,weights}$. The weights transfer time is calculated using the amount of weights in the subgraph and the allocated bandwidth of the target platform. Moreover, between consecutive architectures, the reconfiguration time, $t_{i,reconfig.}$, has to be included. The extended performance model expresses the execution time as:

$$t_{total}(B, \boldsymbol{\Gamma}, \boldsymbol{W}) = \sum_{i=1}^{N_P} \sum_{j=1}^{N_{W_i}} t_{i,j}(B, \boldsymbol{\Gamma}_{i,ref}, \boldsymbol{W}_{i,j}) + \sum_{i=1}^{N_P} \sum_{j=1}^{N_{W_i}} t_{i,j,weights} + \sum_{i=1}^{N_P - 1} t_{i,reconfig.}$$

Finally, the throughput and latency of a design point for a ConvNet which requires $W_{ConvNet}$ GOps can be estimated as in Eq. (5) and Eq. (6) in GOp/s and seconds respectively.

$$T(B, \boldsymbol{\Gamma}, \boldsymbol{W}) = \frac{W_{ConvNet}}{t_{total}(B, \boldsymbol{\Gamma}, \boldsymbol{W})/B} \quad (5)$$

$$L(B = 1, \boldsymbol{\Gamma}, \boldsymbol{W}) = t_{total}(1, \boldsymbol{\Gamma}, \boldsymbol{W}) \quad (6)$$

Algorithm 1 Workload Alignment for Weights Reloading

Inputs:
1: Dimensions $(M \times N)$ of topology matrix $\boldsymbol{\Gamma}_{i,ref}$
2: Workload matrix $\boldsymbol{W}_{i,j} \in \mathbb{R}^{K \times L}$
3: Shift vector $\boldsymbol{s}^{i,j} \in \mathbb{Z}^L$ with the alignment shifts for each column
4: Identity matrices $\boldsymbol{I}^r_{N\times N}$ and $\boldsymbol{I}^l_{M\times M}$
5: Lower shift matrices $\boldsymbol{S}^r_{N\times N}$ and $\boldsymbol{S}^l_{M\times M}$

Steps:
1: $\boldsymbol{W}^{aligned}_{i,j} = \left[\frac{\boldsymbol{W}_{i,j}}{\boldsymbol{0}_{(M-K)\times L}}, \boldsymbol{0}_{M\times(N-L)}\right]$
2: **for all** col in the j^{th} subgraph that need alignment **do**
3: - - - Align along the pipeline, (right shift) - - -
4: - Form right alignment matrix $\boldsymbol{A}^r \in \mathbb{R}^{N\times N}$ -
5: $\boldsymbol{A}^r = \left[\boldsymbol{I}^r_{1:col-1}, \boldsymbol{S}^r_{col:col+s^{i,j}_{col}}, \boldsymbol{I}^r_{col+s^{i,j}_{col}+1:N}\right]$
6: - Update the overall right alignment matrix -
7: $\boldsymbol{A}^r_o = \underbrace{\boldsymbol{A}^r \cdot \boldsymbol{A}^r \cdot ... \cdot \boldsymbol{A}^r}_{s^{i,j}_{col}}$
8: $\boldsymbol{W}^{aligned}_{i,j} = \boldsymbol{W}^{aligned}_{i,j} \cdot \boldsymbol{A}^{r\top}_o$
9: - - - Align the interconnections (down shift) - - -
10: - Form left alignment matrix $\boldsymbol{A}^l \in \mathbb{R}^{M\times M}$ -
11: $\boldsymbol{A}^l = \left[\boldsymbol{I}^l_{1:col-2}, \boldsymbol{S}^l_{col-1:col+s^{i,j}_{col}-1}, \boldsymbol{I}^l_{col+s^{i,j}_{col}:M}\right]$
12: - Update the overall left alignment matrix -
13: $\boldsymbol{A}^l_o = \underbrace{\boldsymbol{A}^l \cdot \boldsymbol{A}^l \cdot ... \cdot \boldsymbol{A}^l}_{s^{i,j}_{col}}$
14: $\boldsymbol{W}^{aligned}_{i,j,col:col+s^{i,j}_{col}} = \boldsymbol{A}^l_o \cdot \boldsymbol{W}^{aligned}_{i,j,col:col+s^{i,j}_{col}}$
15: **end for**

Note: The subscript $start{:}end$ denotes a range of columns.

C. Workload Alignment

In the weights reloading transformation, when a subgraph is mapped to a reference architecture, each of its nodes has to be scheduled on the appropriate block of the architecture as dictated by the current design point. Formally, the i_{th} reference architecture and the j_{th} subgraph, with N and L blocks respectively, have a topology matrix $\boldsymbol{\Gamma}_{i,ref} \in \mathbb{R}^{(M \times N)}$ and a workload matrix $\boldsymbol{W}_{i,j} \in \mathbb{Z}^{(K \times L)}$, with $K \leq M$ and $L \leq N$. To calculate the execution time $t_{i,j}(B, \boldsymbol{\Gamma}_{i,ref}, \boldsymbol{W}_{i,j})$ of the j_{th} subgraph on the i_{th} architecture, the columns of $\boldsymbol{W}_{i,j}$ have to be aligned so that they map on the correct columns of $\boldsymbol{\Gamma}_{i,ref}$. This process can be interpreted as the scheduling of each node on the appropriate block of the architecture so that the correct datapath will be formed for the j_{th} subgraph.

To achieve this, a new matrix $\boldsymbol{W}_{i,j}^{aligned} \in \mathbb{Z}^{(M \times N)}$ is constructed which contains the rows and columns of $\boldsymbol{W}_{i,j}$ with the correct alignment. Our adoption of SDF enables us to express the workload alignment algebraically as described by algorithm (1). The loop starting on line 2 iterates through the columns of the j_{th} workload matrix that need alignment. Lines 3 to 8 shift the current column to the right, i.e. along the coarse pipeline of building blocks in the reference architecture. Next, lines 9 to 14 down-shift the column in order to align the interconnections. After $\boldsymbol{W}_{i,j}^{aligned}$ has been fully formed, the j_{th} initiation interval matrix can be computed correctly as $\boldsymbol{II}_{i,j} = \boldsymbol{W}_{i,j}^{aligned} \oslash \boldsymbol{\Gamma}_{i,ref}$ and used for the calculation of $t_{i,j}(B, \boldsymbol{\Gamma}_{i,ref}, \boldsymbol{W}_{i,j})$ as described in Section III-B.

D. Optimisation

The developed optimiser aims to determine a design point that minimises the latency for the target application given a ConvNet workload and the available resources. In this context, we pose the following combinatorial optimisation problem:

$$\min_{\boldsymbol{\Gamma}} L(1, \boldsymbol{\Gamma}, \boldsymbol{W}), \text{ s.t. } \boldsymbol{rsc}(1, \boldsymbol{\Gamma}) \leq \boldsymbol{rsc}_{Avail.} \tag{7}$$

where L and $\boldsymbol{rsc}$ return the latency in seconds and the resource consumption vector of the current design point. The optimiser operates in two modes by selecting objective function based on the performance metric of interest: throughput or latency. The optimiser considers the four discussed transformations and aims to find reconfiguration and weights reloading partition points, and input feature maps, coarse- and fine-grained folding factors that optimise the objective function.

The partitioning of the SDFG is cast as a search problem, aiming to achieve peak performance by selecting suitable reconfiguration and weights reloading partition points. Given a ConvNet with N_{CONV} convolutional and N_L layers, there are $N_L - 1$ and $N_{CONV} - 1$ candidate reconfiguration and weights reloading partition points with 2^{N_L-1} and $2^{N_{CONV}-1}$ different possible partitionings respectively. In our optimisation framework, we form two vectors $\boldsymbol{p} \in \{0,1\}^{N_L-1}$ and $\boldsymbol{p}_{CONV} \in \{0,1\}^{N_L-1}$ where a value of 1 for the i_{th} element indicates that the SDFG will be partitioned before the i_{th} layer.

After the reconfiguration and weights reloading partitionings have been applied, we end up with N_P architectures, with the respective $\boldsymbol{\Gamma}_{i,ref}$ matrices, each derived based on the scheduled subgraphs. Each of them has its own coarse- and fine-grained folding factors. Coarse-grained folding can be applied to any type of layer where each factor has a range as explained in Section II-H. Fine-grained folding can be applied on convolutional and average pooling layers and each factor's range depends on the maximum window size scheduled. For the rest of the layers, the fine-grained folding factor is set to 1. In the worst case when the derived architectures are a concatenation of the subgraphs with no reuse of the hardware across subgraphs, there are N_L coarse- and fine-grained factors, with the total number of candidate design points given by:

$$2^{N_L-1} \cdot 2^{N_{CONV}-1} \cdot \prod_{i=1}^{N_{CONV}} N_{reload,i} \cdot \prod_{i=1}^{N_L} N_{coarse,i} \cdot \prod_{i=1}^{N_L} N_{fine,i}$$

where $N_{reload,i}$ is the number of possible input feature maps folding factors for the i_{th} convolutional layer and $N_{coarse,i}$ and $N_{fine,i}$ are the number of possible coarse- and fine-grained folding factors for the i_{th} layer respectively. The scaling of the design space size is exponential and prohibits optimisation by means of enumeration. Therefore, a heuristic search method is necessary to obtain an approximate solution in the non-convex design space. In this work, Simulated Annealing has been selected as the basis for the developed optimiser, but the details are omitted due to space limitations.

TABLE I: Benchmarks

Model Name		Layers	Workload	Task
LeNet-5[5]	[7]	4	0.0038 GOps	Digit Recognition
CIFAR-10	[6]	9	0.0247 GOps	Object Recognition
AlexNet[6]	[8]	10	1.3315 GOps	Object Recognition
Sign Recognition CNN	[9]	6	4.0227 GOps	Sign Recognition
Scene Labelling CNN	[10]	8	7.6528 GOps	Scene Labelling
VGG16	[4]	31	30.7200 GOps	Object Recognition

IV. Evaluation

A. Experimental Setup

In our experiments, we target the Xilinx Zynq ZC706 board operating at 125 MHz. Our framework enhances fpgaConvNet [2] and uses the SDFG of each hardware design to automatically generate synthesisable Vivado HLS code. All hardware designs were synthesised and placed-and-routed with Xilinx Vivado HLS and Vivado Design Suite (v16.4) and run on the ZC706 board. The ARM CPU was used to set up the off-chip memory transactions, launch the hardware execution and measure the performance of each design. In the evaluation, Q8.8 fixed-point precision was used following the practice of the FPGA works we compare with.

Benchmarks. Table I lists our benchmark ConvNets which vary in terms of number of layers, computational load, memory requirements and task and pose different mapping challenges. LeNet-5 and CIFAR-10 represent the lower end in terms of workload and memory requirements. AlexNet comprises non-uniform kernel sizes across its convolutional layers, including 11×11, 5×5 and 3×3 kernels. The Sign Recognition CNN processes inputs of substantial size by targeting 1280×720 HD video and poses a challenge in terms of off-chip memory bandwidth. The Scene Labelling CNN has large on-chip memory requirements with 1.6 MB of weights and is on-chip memory bounded for several FPGAs. Finally, VGG16 is one of the largest state-of-the-art ConvNets and comprises a large workload of 30.72 GOps and 29.4 MB of weights.

B. Evaluation of Latency Estimator

In this section, the accuracy of the developed latency model is evaluated. This is investigated by selecting five design points that lie on the Pareto front of our DSE for each of the benchmarks. Each hardware design is generated and run on the target FPGA and we compare the estimated to the real, measured

[5]Based on Caffe version of LeNet-5 under the name MNIST.

[6]Includes only the convolutional layers.

TABLE II: Estimated vs. Real Measured Latency

Model Name	Error
LeNet-5	6.20 %
CIFAR-10	5.30 %
AlexNet	5.16 %
Sign Recognition CNN	5.28 %
Scene Labelling CNN	1.81 %
VGG16	7.10 %
Average	5.14 %

TABLE III: Low-Latency Mode Effect (batch size = 1)

Model Name	Latency Improvement	% of Peak Throughput
LeNet-5	No Change	100.00 %
CIFAR-10	105.92 ×	90.82 %
AlexNet	73.54 ×	82.05 %
Sign Recognition CNN	1.93 ×	94.45 %
Scene Labelling CNN	6.46 ×	18.28 %
VGG16	5.61 ×	79.02 %

latency. We should note that all latency measurements were performed for a batch size of 1 and include computation time and the communication between the FPGA and the off-chip memory. Table II summarises the error of our latency model across all the selected points for each benchmark.

The latency predictions lie fairly close to the actual measured latency on the target platform with the estimator giving a slightly optimistic prediction. Variations in the I/O delays, reconfiguration time and software overhead contribute to the model's small error. Nevertheless, the magnitude of the error does not influence the latency estimator's ability to effectively guide the DSE task.

C. Low Latency vs. High Throughput

This section presents the performance gains of using the proposed latency-driven methodology compared to throughput-driven optimisation. This is investigated by implementing the benchmarks using both the latency-driven and the throughput-driven flows and comparing the achieved performance, measured on the ZC706 board. Only the throughput measurements for the throughput-driven designs include batch processing, with the rest using a batch size of 1. Table III presents the latency improvements between the two design flows. Moreover, Fig. 4 shows the achieved throughput and latency measured on the FPGA for the designs generated with the two modes and depicts how the latency-driven design shifts the generated hardware to regions with substantially lower latency. In the cases where a network can be fully accommodated in the target FPGA, the low-latency and high-throughput designs coincide, such as for LeNet-5. For larger benchmarks, the introduction of the weights reloading transformation, which expands the design space and removes the need for FPGA reconfiguration, together with the latency-centric objective function enable our optimiser to search previously unreachable design regions, achieving a latency improvement of up to $105.92\times$ with an average of $32.41\times$ ($9.04\times$ geo. mean) across the benchmarks.

AlexNet Case Study. The low-latency mapping of AlexNet poses two main challenges: (1) non-uniform kernel sizes across its layers, from 11×11 to 3×3; (2) given the ZC706 resources, AlexNet is on-chip memory-bounded by requiring $2.09\times$ more weights memory than available on-chip. The proposed latency-driven flow addressed challenge (2) by fully partitioning the model with one partition per convolutional layer using the weights reloading transformation. In this way, each layer could utilise all the available on-chip memory. To address (1), our

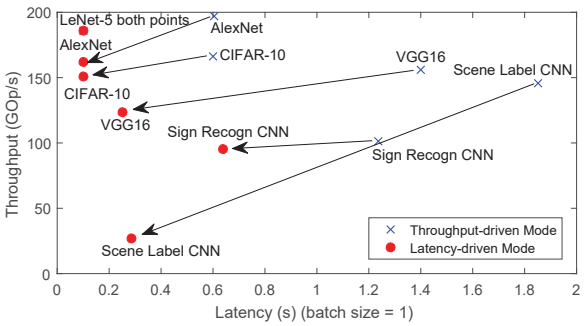

Fig. 4: Comparison of Throughput and Latency-Driven Modes

DSE method generated a single convolution block (Fig. 1). The final low-latency design consisted of 64 coarse convolution units and 14 MACC units per coarse unit, yielding a peak theoretical performance of 224 GOp/s with a 99.55% DSP consumption. Our latency estimator takes into account the limited off-chip memory bandwidth and the DSP underutilisation due to the heterogeneous kernel sizes and yields a latency estimate of 7.80 ms. The final achieved latency measured on the target FPGA is 8.22 ms, achieving 161.98 GOp/s and 72.23% of the peak performance. On the other hand, the throughput-driven flow generated a design with two bitstreams and FPGA reconfiguration leading to a measured latency and throughput of 604.5 ms and 197.40 GOp/s respectively, with the latency measured with batch size of 1 and the throughput with larger batch size that amortises the reconfiguration times. Overall, the low-latency design resulted in a $73.54\times$ latency improvement, achieving 82.05% of the high-throughput design's throughput.

VGG16 Case Study. VGG16 is a compute and on-chip memory-bounded benchmark. The high-throughput design includes 3 reference architectures with 2 reconfigurations, each one tailored to a part of the network, achieving 155.81 GOp/s with a latency of 1.4 s. On the other hand, the low-latency design point differs from its high-throughput counterpart primarily by consisting of a single architecture without any reconfiguration, which leads to a latency improvement of $5.61\times$. The low-latency design consists of 171 coarse convolution units with 5 MACC units per coarse unit, leading to a peak theoretical performance of 213.75 GOp/s with a 95% DSP consumption. The developed latency estimator predicted 234.53 ms. The final achieved latency measured on the FPGA is 249.5 ms, achieving 123.12 GOp/s and 57.6% of the peak performance. The deviation between the peak and the achieved performance occurs due to two factors: (1) the memory bandwidth limitations of the target platform with a measured average of 3.8 GB/s and (2) the underutilisation of the computational resources in layers with fewer than 171 output feature maps. As an example, VGG16's first layer has 64 output feature maps and 3×3 kernels. This layer occupies 64 out of the 171 instantiated convolution units and as a result utilises 64×5 MACC units leading to an actual peak throughput of 80 GOp/s for this layer, which is 37.4% of the theoretical peak throughput.

D. Performance Comparison

This section presents a comparison with FPGA designs that utilise a batch size of 1, optimising simultaneously latency and throughput. Table IV presents the comparison for the widely used AlexNet and VGG16. With respect to AlexNet, we compare with the roofline model-based design by [11] and the automated design flows of [12] and [13] which are the closest works to our automated approach. Our methodology

TABLE IV: Comparison with Existing Work (batch size = 1)

	[11] AlexNet	[12] AlexNet[7]	[13] AlexNet	This Work: AlexNet	[14] VGG16	[15] VGG16[8]	This Work: VGG16
FPGA	Virtex-7 VX485T	Zynq XC7Z045	Stratix-V GXA7	Zynq XC7Z045	Zynq XC7Z045	Arria-10 GX1150	Zynq XC7Z045
Frequency	100 MHz	100 MHz	100 MHz	125 MHz	150 MHz	150 MHz	125 MHz
Logic Capacity	303.60 kLUTs	218.60 kLUTs	234.72 kALMs	218.60 kLUTs	218.60 kLUTs	427.2 kALMs	218.60 kLUTs
Fixed-point DSPs*	2900	900	512	900	900	3036	900
On-chip Memory	4.5 MB	2.4 MB	5.6 MB	2.4 MB	2.4 MB	6.6 MB	2.4 MB
Precision	32-bit float	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	8/16-bit fixed	16-bit fixed
Latency	21.61 ms	12.30 ms	9.92 ms	8.22 ms	163.42 ms	33.579 ms	249.50 ms
Performance (GOp/s)	61.62	108.25	134.10	161.98	187.80	914.85	123.12
Performance Density (GOp/s/Logic)	0.2029 GOp/s/kLUT	0.4952 GOp/s/kLUT	0.5713 GOp/s/kALM	0.7410 GOp/s/kLUT	0.8591 GOp/s/kLUT	2.1415 GOp/s/kALM	0.5632 GOp/s/kLUT
Performance Density (GOp/s/DSP)	0.021	0.12	0.26	0.18	0.21	0.30	0.14

* 18x18 and 18x25 DSP configurations.

outperforms [11], [12][7] and [13] by $2.63\times$, $1.49\times$ and $1.20\times$ respectively in both latency and throughput. [13] reaches a higher performance density normalised for DSPs, but targets a device with $2.3\times$ larger on-chip memory, which enables the reduction of off-chip memory transfers and latency. With respect to VGG16, we compare with the designs presented in [14] and [15]. In [14], the authors developed a highly optimised VGG16 accelerator tailored for ZC706. Our automated flow manages to generate a design that reaches 65.5% of the performance of the hand-tuned VGG16 design of [14] on the same platform, with the advantage of a much lower development time and effort, together with support for a wide range of CNNs. In [15][8], a loop parallelisation scheme for the latency minimisation of VGG16 is presented targeting the Arria-10 GX1150 FPGA on a Nallatech 385A board. Our low-latency VGG16 design reaches 46% of the performance density normalised for DSPs. An important factor to take into account is that [15] runs on a platform with $2.75\times$ more on-chip memory and $3.8\times$ higher off-chip memory bandwidth[9], which substantially reduce the memory accesses and the associated latency.

V. Other Related Work

So far, prior FPGA and ASIC work has primarily focused on throughput maximisation. In this cases, the majority of reported results include GOp/s when a favourable batch size is used. In [16], an OpenCL-based high-throughput accelerator is proposed which employs batch processing in order to sustain a high resource utilisation and hide the host-accelerator communication overhead. In [17], Chen et al. used batch processing to maximise weights reuse in ConvNet layers across multiple inputs. [18] and [19] are more similar to our approach in presenting automated flows for mapping ConvNets to FPGAs. Both frameworks optimise for throughput and employ favourable batch sizes, with [19] also aiming to keep the batch size small. In [20], Zhang et al. developed an FPGA cluster and presented a framework that can optimise the ConvNet hardware for different metrics. Similarly to our approach, distinct throughput- and latency-driven modes can be selected based on the target application. Our work is focused on single-FPGA setups, but can be extended to multi-FPGA configurations by programming a pipeline of multiple FPGAs with the bistreams generated by the graph partitioning with reconfiguration transformation.

VI. Conclusion

This paper presents a latency-driven design methodology for mapping Convolutional Neural Networks on FPGAs. A novel, flexible architecture is proposed that is automatically derived based on the ConvNet workload and the resources of the target FPGA, optimised for low latency. The weights reloading transformation is introduced as a new SDF transformation for ConvNets tailored to low-latency applications, which expands the design space with lower-latency design points. This formulation enables the efficient exploration of design points by means of algebraic operations, allows us to design a latency-centric optimiser to guide the design space exploration task and facilitates the automation of the end-to-end flow. Potential future work would include the formulation of the ConvNet's hardware mapping as a multiobjective optimisation problem.

[7]The reported performance results were obtained by contacting the authors.

[8]The authors report the feature extractor to contribute 70% of the latency.

[9]The Nallatech 385A and ZC706 boards provide a peak bandwidth of 16 GB/s and 4.2 GB/s respectively.

References

[1] Y. Bengio, "Learning Deep Architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009.

[2] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *FCCM*, 2016.

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Sept 1987.

[4] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ICLR*, 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.

[6] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, Tech. Rep., 2009.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," in *Proc. IEEE*, 1998.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.

[9] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-Centric Accelerator Design for Convolutional Neural Networks," in *ICCD*, 2013.

[10] L. Cavigelli *et al.*, "Accelerating Real-Time Embedded Scene Labeling with Convolutional Networks," in *DAC*, 2015.

[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[12] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family," in *DAC*, 2016.

[13] Y. Ma, N. Suda, Y. Cao, J. sun Seo, and S. Vrudhula, "Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA," in *FPL*, 2016.

[14] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *FPGA*, 2016.

[15] Y. Ma, Y. Cao, S. Vrudhula, and J. sun Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017.

[16] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, 2016.

[17] Y.-H. Chen *et al.*, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[18] 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.

[19] 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.

[20] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster," in *ISLPED*, 2016.