fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs

Stylianos I. Venieris, *Student Member, IEEE*, and Christos-Savvas Bouganis, *Senior Member, IEEE*

***Abstract*— Since neural networks renaissance, convolutional neural networks (ConvNets) have demonstrated a state-of-the-art performance in several emerging artificial intelligence tasks. The deployment of ConvNets in real-life applications requires power-efficient designs that meet the application-level performance needs. In this context, field-programmable gate arrays (FPGAs) can provide a potential platform that can be tailored to application-specific requirements. However, with the complexity of ConvNet models increasing rapidly, the ConvNet-to-FPGA design space becomes prohibitively large. This paper presents fpgaConvNet, an end-to-end framework for the optimized mapping of ConvNets on FPGAs. The proposed framework comprises an automated design methodology based on the synchronous dataflow (SDF) paradigm and defines a set of SDF transformations in order to efficiently navigate the architectural design space. By proposing a systematic multiobjective optimization formulation, the presented framework is able to generate hardware designs that are cooptimized for the ConvNet workload, the target device, and the application's performance metric of interest. Quantitative evaluation shows that the proposed methodology yields hardware designs that improve the performance by up to 6.65× over highly optimized graphics processing unit designs for the same power constraints and achieve up to 2.94× higher performance density compared with the state-of-the-art FPGA-based ConvNet architectures.**

***Index Terms*— Convolutional neural networks (ConvNets), design space exploration (DSE), field-programmable gate arrays (FPGAs), parallel reconfigurable architectures.**

I. Introduction

IN RECENT years, convolutional neural networks (ConvNets) have emerged as the state-of-the-art model in several artificial intelligence tasks. From object and face recognition [1], [2] to object detection and segmentation [3], the predictive strength of ConvNets has led to their adoption in a broad range of real-life applications. In both embedded systems and data center setups, there is a common requirement for fast, high-performance ConvNet deployment at low power consumption. In this context, there is an increasing need for efficient mappings of the inference stage of ConvNets on computing platforms that can provide such a balance.

Manuscript received October 26, 2017; revised February 23, 2018 and April 30, 2018; accepted May 31, 2018. Date of publication July 2, 2018; date of current version January 21, 2019. *(Corresponding author: Stylianos I. Venieris.)*

The authors are with the Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, U.K. (e-mail: stylianos.venieris10@imperial.ac.uk; christos-savvas.bouganis@imperial.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2018.2844093

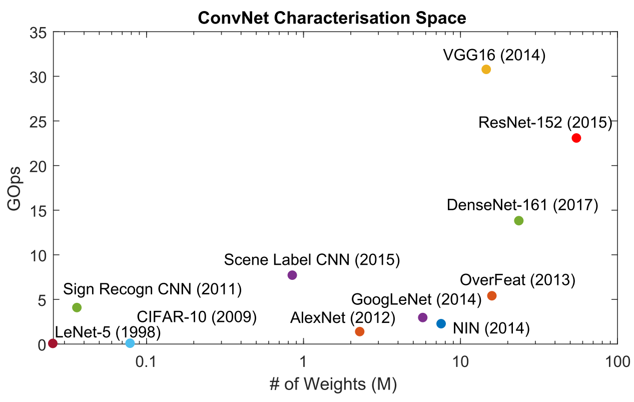


Fig. 1. ConvNet models in the computation-memory space.

Apart from high predictive accuracy, ConvNets are also characterized by challenging computational and memory requirements. Fig. 1 shows a number of well-known models in the computation-memory space, spanning from the low-end LeNet-5 [4] up to the more recent large-scale ResNet-152 image classifier [5]. To reach high accuracy, one approach in a network design employs deep and wide ConvNets with a large number of trainable parameters in order to increase the expressive power of the model. This design principle is depicted in models such as VGG16 [1] which trades off higher inference accuracy with substantially increased computational and memory intensity, as demonstrated in the ImageNet Challenge [6]. Following a different approach, recent models have been cooptimized for both accuracy and computation leading to networks, such as GoogLeNet [7], ResNet [5], and DenseNet [8]. Each of these networks has reached similar or higher accuracy compared with large networks at a lower computational load. This was typically achieved by introducing novel network components that differ from the conventional layer types, which introduce irregular connectivity between layers. The properties of irregularity and nonuniformity make the dataflow of ConvNets more complex compared with conventional models and challenge the existing mapping methodologies. With inception-based, residual, and dense networks becoming the state of the art in accuracy, there is an increasing need for their optimized mapping in order to deploy them broadly.

Until the middle of the previous decade, the conventional computing infrastructure for ConvNets comprised general-purpose machines, including multicore CPUs in server environments and low-power microcontrollers in embedded systems. The increased computational needs of ConvNet models together with the recent hardware advances led to a shift toward customized hardware, with the prominence of graphics processing units (GPUs) [9] and application-specific integrated circuits (ASICs) [10]. Existing deep learning frameworks, such as Caffe, Torch, and TensorFlow, provide high-performance execution of ConvNets by employing power-costly GPUs as their main acceleration platform. Farther in the customization spectrum, ASIC chips offer ConvNet acceleration with minimal power and area consumption. Nevertheless, they require full-custom manufacturing at high nonrecurring engineering cost. Moreover, the ASICs' advantages typically require the chip's functionality to remain fixed after fabrication and consequently lack the flexibility of mapping algorithmic advances in the area of ConvNets, and cannot exploit model-specific optimizations due to the fixed architecture.

In this context, reconfigurable hardware in the form of field-programmable gate arrays (FPGAs) constitutes a promising alternative. FPGAs offer the benefits of customizability and reconfigurability by means of a set of heterogeneous resources, including lookup tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), and digital signal processing (DSP) blocks, with programmable connections between them. These properties allow an FPGA to reconfigure its fabric at run time and, in this way, provide a hardware architecture tailored for the target application. Because of their versatility, big industrial companies, such as Microsoft and Amazon, have redesigned their data center facilities in order to host FPGAs on their servers and target a variety of workloads with specialized hardware [11]. The FPGA mapping of ConvNets could potentially provide tunable tradeoffs between critical system parameters, including performance, power, and cost, and serve as a useful component in both embedded and data center systems.

Nevertheless, several issues increase the complexity of ConvNet system development on FPGAs [12]. With FPGAs' size and resource specifications advancing at a fast pace and with ConvNets becoming more complex, the possible mappings of a ConvNet on an FPGA lie on a large multidimensional design space that cannot be explored manually. At the same time, the diversity of ConvNet application domains results in a wide spectrum of performance needs. Spanning from high-throughput image recognizers to latency-critical self-driving cars, the underlying computing system has to be tailored to the particular performance metric of interest. To this end, there is a need for tools that abstract the low-level resource details of a particular FPGA and automate the mapping of ConvNets on FPGAs in a principled manner.

This paper presents a novel approach to modeling and mapping the inference task of both regular and irregular ConvNets on FPGA-based platforms. By enhancing the work presented in [13]–[15], we propose a synchronous dataflow (SDF) framework for the modeling and mapping of ConvNets on reconfigurable streaming hardware and introduce a set of transformations over the SDF model in order to efficiently explore the architectural design space and the performance-resource tradeoff. The initial work in [13] focuses on generating hardware designs optimized for high-throughput applications, giving an overview of our SDF streaming modeling scheme and evaluating over a set of relatively small ConvNets targeting a low-power, low-end FPGA platform. The work presented in [14] introduced a design methodology tailored for emerging latency-sensitive applications. This paper generalizes the framework in order to address the emerging challenges of state-of-the-art ConvNets with *irregular connectivity*, including GoogLeNet, ResNet-152, and DenseNet-161, expands the range of target applications to include applications with both throughput and latency requirements by casting design space exploration (DSE) as a *multiobjective optimization* (MOO), provides, for the first time, an in-depth analysis of the proposed modeling and DSE methods, and presents extensive comparisons with highly optimized designs on embedded GPUs and with the most recent state-of-the-art FPGA designs.

II. Background

A. Convolutional Neural Networks

A typical ConvNet comprises a sequence of layers, organized in two stages: the feature extractor and the classifier. The three most commonly used types of layers for the feature extractor are the convolutional layer, the nonlinear layer, and the pooling layer, while the classifier typically includes fully connected layers [16]. A convolutional layer operates as a feature extraction mechanism, aiming to detect features in the feature maps produced by the preceding layer. Computationally, this is achieved by performing convolutions between N_{in} input feature maps and the layer's trainable $(K_h \times K_w)$ kernels which result in the production of N_{out} output feature maps. Formally, this operation can be expressed as

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

where $\boldsymbol{f}_i^{\text{out}}$ and $\boldsymbol{f}_j^{\text{in}}$ are the ith and jth output and input feature maps, respectively, $\boldsymbol{k}_{i,j}$ is the $(K_h \times K_w)$ kernel that corresponds to the ith output and jth input, and $\boldsymbol{b}_i$ is the ith bias vector. The $*$ operator represents the 2-D convolution between a feature map and a kernel, which is computed by sliding the $(K_h \times K_w)$ window of weights over the input feature map.

A nonlinear layer operates as an activation function that indicates whether a feature is present at each element of the feature maps. A nonlinear function is applied elementwise on the input feature maps with typical nonlinearities being the sigmoid, tanh, and rectified linear unit (ReLU).[1] A pooling layer aims to introduce invariance in terms of spatial translation of features and downsample its inputs by sliding a window over input feature maps and replacing with a summary statistic, with the most common pooling operations being average and max.

A fully connected layer performs the product between an input vector and a weight matrix to produce an output vector. The inputs to this layer that might be multidimensional are flattened and arranged as a 1-D vector. With the feature extractor

[1]ReLU is defined as $f(x) = \max(0, x)$.

typically dominating the computational cost of ConvNets [17] and fully connected layer-based classifiers being abandoned in the recent state-of-the-art models [5], [7], [8], [18], [19], in this paper, we focus on the feature extractor stage.

B. State-of-the-Art ConvNets With Irregular Connectivity

Recently, novel ConvNet architectures have achieved a higher accuracy by employing nonuniform layer connectivity. Representative networks that have followed this approach include *GoogLeNet* [7], *ResNet* [5], and *DenseNet* [8]. In contrast to prior work such as VGG16 that favored simplicity and a uniform, serial layer connectivity, these types of networks introduce novel compound blocks to increase the expressive power of the model and reduce the computation requirements at the expense of more complex dataflows that pose challenges with respect to mapping.

1) Inception-Based Networks: In 2014, Szegedy *et al.* [7] presented GoogLeNet as a network that achieves the state-of-the-art accuracy without excessive computation. To achieve this, GoogLeNet introduced the *Inception module*, which substitutes the conventional single serial connection between layers with four heterogeneous paths whose outputs are concatenated: three convolutional (1×1, 3×3, and 5×5) and one 3×3 max-pooling path. Moreover, to limit the number of weights, 1×1 filters are applied to reduce the number of channels prior to the 3×3 and 5×5 convolutional layers and after the pooling layer. Each path consists of a different computational load and hence offers the opportunity for mapping optimizations.

2) Residual Networks: In 2015, He *et al.* [5] set a new record in the ImageNet accuracy by proposing ResNet. This network architecture employs *shortcut connections* that comprise forward connections between layers at different depth levels. Instead of adding shortcuts between all the layers, the network is organized as a series of *residual blocks*. Computationally, inside a residual block, feature maps are combined by elementwise addition before being fed to the subsequent layer. Moreover, similar to GoogLeNet, 1×1 filters are employed in order to reduce the number of weights. Overall, each residual block consists of three convolutional layers (1×1, 3×3, and 1×1). The ResNet with 152 layers (ResNet-152) demonstrated the highest accuracy on ImageNet, requiring 23 GOp/input and 55 million weights.

3) Dense Networks: In 2017, DenseNet [8] introduced a novel structure under the name *dense block*, as a mechanism to achieve the accuracy level of ResNet at a lower computational load. Inside a dense block, the output of each layer is directly connected to the input of every following layer in a feed-forward manner. Dense blocks are parameterized with respect to: 1) the number of input feature maps; 2) its *growth rate*, which is defined as the number of output feature maps that each convolutional layer produces, denoted by k following the notation of [8]; and 3) the number of convolutional layers inside the dense block. With this setting, each convolutional layer of the block receives k more input feature maps compared with its preceding layer. The DenseNet with 161 layers (DenseNet-161) is achieved the same accuracy level as ResNet-152 at a lower computational cost.

These three types of network architectures are currently paving the way for higher accuracy and are employed as a starting point for deploying ConvNets in new application domains. Nevertheless, the large depth and the increased connectivity complexity make the mapping of these models a challenging task. To this end, we also target this set of models and address the optimized mapping of their irregular dataflow on FPGAs.

C. Synchronous Dataflow

The SDF [20] is a widely used model of computation for the analysis and design of parallel systems. Under this scheme, a hardware or software computing system is described as a directed graph, named SDF graph (SDFG), with the nodes representing computations and with arcs in place of data streams between them. The basic principle of the SDF is the data-driven streaming execution, where each node fires whenever data are available at its incoming arcs. The characteristic property that differentiates SDF from the conventional dataflow is that the amount of data produced and consumed by a node is known at compile time. This property enables the construction of static schedules of execution for the target system with finite and predictable amount of data buffering between nodes, avoiding in this way the overhead of dynamic control and enabling us to apply performance optimizations at compile time.

III. Related Work

With deep learning's recent success, several research groups have focused on the design of customized architectures for ConvNets. Efforts have concentrated on hand-tuned mappings of specific ConvNet models and computation engine optimizations on FPGAs [21]–[24] and ASICs [25]–[28] and memory subsystem optimizations on FPGAs [29], [30] and ASICs [31]–[33]. GPU-based acceleration has also been addressed from various aspects. Several libraries and frameworks offer GPU implementations of ConvNet layers through a high-level interface [34]–[37]. From a computational perspective, recent works have explored the parallelization of convolutions [38], the coarse-grain, batch-level parallelism [39], and the vectorization of layers [40]. From an algorithmic aspect, recent efforts have employed alternative algorithms to reduce the computational complexity of convolutions [41]–[43]. Several strategies have been explored to address the large memory requirements, including the compression of trained weights [44] and data-layout transformations [45]. Finally, the optimization of GPU designs for latency-critical embedded applications has also been investigated [27].

1) Design Space Exploration: Besides the above contributions, a significant research effort has been invested into the development of systematic ConvNet-to-FPGA DSE methodologies. Zhang *et al.* [46] proposed a DSE method based on the roofline model for the configuration of an accelerator, targeting solely convolutional layers. The underlying architecture consists of a fixed processing engine that is time-shared between layers and can be configured at compile time with the optimal configuration found by means of enumeration. Suda *et al.* [47] presented a DSE scheme that considers all layer types. Their proposed methodology focuses on high-throughput applications that run on server-based FPGAs. Without considering latency-sensitive applications, their target objective is the maximization of throughput.

Wang *et al.* [48] developed a framework for the automatic generation of ConvNet hardware implementations under the name DeepBurning. Given a ConvNet, hardware components are assembled to generate an accelerator. An emphasis is given on the identification and implementation of appropriate hardware components for neural networks without performing systematic DSE as opposed to this paper. The generated accelerators are designed to operate with a batch size of 1 and hence can target both high-throughput and low-latency applications. DNNWEAVER [49] comprises an automated flow for the generation of high-throughput accelerators for a given ConvNet-FPGA pair. In its back end, DNNWEAVER employs a set of parameterized hardware templates that correspond to different types of layers. A heuristic search algorithm is used to allocate FPGA resources to each template instance and schedule the execution of layers. In [50], Escher is proposed as a methodology for increasing the memory bandwidth utilization of FPGA-based ConvNets, while Gokhale *et al.* [51] designed a programmable accelerator optimized for the high utilization of its resources. FP-DNN [52] and the convolutional neural networks (CNN) register-transfer level (RTL) Compiler [53] proposed RTL-level optimizations in order to reach high performance and were the first work to target residual networks. Moreover, [54] and [55] presented automated frameworks specifically tailored for FPGA-based binarized and spiking neural networks, respectively, while [56] proposed a library for the mapping of ConvNets on diverse embedded platforms together with a comparative study of their design spaces. Finally, [12] provides a detailed survey of ConvNet-to-FPGA toolflows.

fpgaConvNet differs from existing efforts by proposing a ConvNet-to-FPGA automated framework that combines systematic DSE with the generation of streaming accelerators that are cooptimized for the ConvNet workload, the target FPGA, and the application-level performance needs. In contrast to the implementation-focused DeepBurning, fpgaConvNet's key contribution lies on formalizing and performing efficiently the DSE task by means of the proposed SDF streaming model. By taking into account the application performance requirements, the generated design is optimized for either throughput, latency, or multiobjective criteria, as opposed to works [47], [49], [50], [52], [57] which aim solely for throughput maximization. Finally, this paper is the first ConvNet-to-FPGA framework to target all the three families of Inception-based, residual, and dense networks.

IV. PROCESSING FLOW OVERVIEW

The proposed end-to-end framework aims to bridge the gap between ConvNet models described in existing deep learning software and their optimized deployment on FPGAs. Fig. 2 shows a high-level view of the framework's processing flow. Initially, the deep learning expert provides as inputs a trained ConvNet model, expressed in Torch or Caffe, together with the resources of the target FPGA. Next, the front-end parser processes the inputs and populates a directed acyclic graph (DAG) application model that captures the structure of the input ConvNet and the resource constraints of the target platform. As a next step, the DAG model is transformed into an SDFG. At this point, the nodes of the SDFG correspond to parameterized hardware building blocks and its arcs to interconnections between them. By applying a set of transformations over the SDFG, the optimizer modifies the parameters of the building blocks and explores, in this manner, the design space of different hardware mappings onto the particular FPGA. As a final step, the code generator produces the hardware description of the selected mapping, leading to the generation of the actual hardware for the target device.

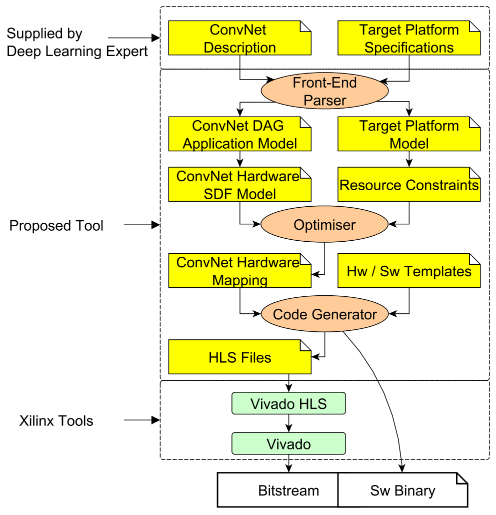


Fig. 2. Overview of fpgaConvNet's processing flow.

A. ConvNet Application Model

A ConvNet workload comprises a sequence of layers that is captured in this paper by means of a computational DAG with each layer mapped to a node. The target network is supplied by means of a Torch or Caffe description, and the front-end parser extracts the necessary information about the type, configuration, and connectivity of layers in order to populate the ConvNet application model.

By interpreting a ConvNet workload as a streaming application, we model it as a DAG $G_{\text{ConvNet}} = (V, E)$, with each node $v \in V$ corresponding to a layer. In order to unify the representation of the ConvNet layers, we propose to capture their configuration by means of a tuple of parameters. In this manner, each node $v \in V$ in the ConvNet is associated with a layer-specific tuple. As an example, the convolutional layer tuple is modeled as follows:

$$\langle K_h, K_w, S_h, S_w, P, N \rangle$$

where K_h and K_w are the height and width of each filter, S_h and S_w are the strides that determine the step between successive windows along the feature map's height and width, respectively, P is the zero padding of the input feature maps, and N is the number of filters. The rest of the layers follow the same modeling approach with the nonlinear layer defined in terms of number of nonlinear units and type of nonlinear function to be applied, e.g., *sigmoid*, *tanh*, or *ReLU*, and the pooling layer defined in terms of pooling size, stride, number of pooling units, and type of pooling operation, e.g., *max* or *average*.

B. Target Platform Model

Beyond the ConvNet model, the deep learning expert also provides the resource budget of the target FPGA platform. The developed FPGA platform model comprises a set of parameters that capture information about the computational and storage resources of the FPGA and the accompanying off-chip memory subsystem. The FPGA contains a set of heterogeneous resources that include DSP blocks, LUTs, FFs, and BRAMs. The off-chip memory is typically a DDR SDRAM module that is characterized by its bandwidth and capacity. To formally represent the resources of the target platform, we define a global resource set, R, that is the union of sets R_{fpga} and R_{mem}

$$\begin{aligned} R_{\text{fpga}} &= \{\text{DSP, LUT, FF, BRAM}\} \\ R_{\text{mem}} &= \{B_{\text{mem}}, C_{\text{mem}}\} \\ R &= R_{\text{fpga}} \cup R_{\text{mem}}. \end{aligned} \tag{2}$$

Moreover, a resource vector, $\boldsymbol{rsc}_{\text{Avail.}}$, is defined which holds the available amount for each of the elements in R

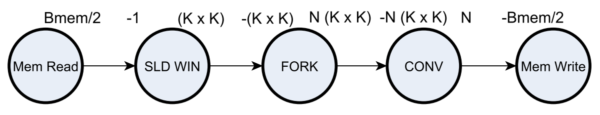
$$\begin{aligned} \boldsymbol{rsc}_{\text{Avail.}} = [&DSP_{\text{Avail.}}, LUT_{\text{Avail.}}, FF_{\text{Avail.}}, \\ &\text{BRAM}_{\text{Avail.}}, B_{\text{mem}}, C_{\text{mem}}]^{\top} \end{aligned} \tag{3}$$

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

V. ConvNets as SDF Graphs

A key contribution of this paper is the representation of hardware design points as SDFGs. At a hardware level, fpgaConvNet represents *design points* as SDFGs that can execute the input ConvNet workload. Given a ConvNet's DAG application model, each node is mapped to a sequence of *hardware building blocks* that implement the node's functionality. By assigning one SDF node to each building block, an SDFG is formed. The nodes of the SDFG are connected via arcs that carry data between building blocks. Each building block is defined by a set of parameters that can be configured at compile time. This process leads to the formation of a hardware architecture that consists of a coarse pipeline of building blocks and corresponds to a design point in the architectural design space.

Fig. 3 shows the translation of a convolutional layer to the corresponding SDF hardware graph. In this scenario, a 2-D convolutional layer with N $(K \times K)$ filters is mapped to three building blocks: a *sliding window block*, a *fork unit*, and a *convolution bank*, together with the necessary I/O modules, including memory read and write blocks. The sliding window block receives the input feature maps as a stream of elements and produces $(K \times K)$ windows. The fork



$$\boldsymbol{\Gamma} = \begin{bmatrix} \frac{B_{mem}}{2} & -1 & 0 & 0 & 0 \\ 0 & (K \times K) & -(K \times K) & 0 & 0 \\ 0 & 0 & N(K \times K) & -N(K \times K) & 0 \\ 0 & 0 & 0 & N & -\frac{B_{mem}}{2} \end{bmatrix}$$

Fig. 3. Convolutional layer as an SDFG.

unit copies the windows to N parallel streams. The convolution bank comprises N convolution units that each performs a dot product between the incoming windows and a $(K \times K)$ kernel of the convolutional layer.

Following the SDF theory, an SDFG can be represented in a matrix form using a *topology matrix*, denoted by $\boldsymbol{\Gamma}$. Each column of the matrix represents an SDFG node, and each row of the matrix represents an arc between two nodes. The elements of each column hold the data production and consumption rates of the particular node at the particular arc. In a conventional SDF, an element $\boldsymbol{\Gamma}(a, n)$ is a positive or negative integer in the case when data are produced or consumed by node n on arc a, respectively. The topology matrix of the SDFG from Fig. 3 is shown in (4), where B_{mem} is the measured memory bandwidth.

The proposed framework adopts the SDF paradigm and enhances it with two extensions. The first extension is the decomposition of the topology matrix into the Hadamard product[2] of three matrices. Each of the three matrices allows us to analyze separately the parallelism at different granularities and interpret the elements of the topology matrix in a deeper manner. The first matrix is the *streams matrix*, denoted by $\boldsymbol{S}$. Each element of $\boldsymbol{S}$ is a nonnegative integer that holds the number of parallel streams at each arc. The second matrix is the *channels matrix*, denoted by $\boldsymbol{C}$. Each element of $\boldsymbol{C}$ holds the width of each stream in words and has a positive or negative sign that indicates the direction of the data flow. The third matrix is the *rates matrix*, denoted by $\boldsymbol{R}$. A value $\boldsymbol{R}$(a, n) is the normalized rate of data production or consumption per cycle of node n on arc a and lies in the interval [0, 1]. A value of 0 indicates no dataflow and 1 indicates a rate of 1 firing per cycle. Following this decomposition, the topology matrix of the SDFG can be reconstructed as follows:

$$\boldsymbol{\Gamma} = \boldsymbol{S} \odot \boldsymbol{C} \odot \boldsymbol{R} \tag{5}$$

where $\boldsymbol{\Gamma} \in \mathbb{R}^{(M \times N)}$, $\boldsymbol{S} \in \{0\} \cup \mathbb{Z}^{+(M \times N)}$, $\boldsymbol{C} \in \mathbb{Z}^{(M \times N)}$, and $\boldsymbol{R} \in \mathbb{R}^{(M \times N)}$ for a design point with N building blocks and M connections. As a second extension of SDF, the topology matrix is allowed to contain real values in order to accommodate the real-valued rates matrix $\boldsymbol{R}$. All the three matrices are upper bidiagonal with nonzero elements along the main diagonal and the diagonal above it, leading to an upper bidiagonal topology matrix. For the SDFG in Fig. 3,

[2]The Hadamard product, here denoted by $\odot$, is defined as the elementwise multiplication between two matrices.

the streams, channels, and rate matrices would be as follows:

$$\boldsymbol{S} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & N & N & 0 \\ 0 & 0 & 0 & N & 1 \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} B_{\text{mem}} & -1 & 0 & 0 & 0 \\ 0 & (K \times K) & -(K \times K) & 0 & 0 \\ 0 & 0 & (K \times K) & -(K \times K) & 0 \\ 0 & 0 & 0 & 1 & -B_{\text{mem}} \end{bmatrix}$$

$$\boldsymbol{R} = \begin{bmatrix} \frac{1}{2} & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & \frac{1}{2} \end{bmatrix}.$$

The $\boldsymbol{\Gamma}$ matrix representation offers several benefits: 1) it captures in an analytical way how a local tuning impacts the overall performance of the system; 2) it enables us to generate a *static schedule* of all the operations; 3) determine the amount of buffering between subsequent blocks at compile time; and 4) ensure the functional correctness of each design point by calculating the data rates of each block.

A. ConvNet Hardware Building Blocks

Extending the idea of capturing ConvNet layers by means of a tuple representation, we adopt a uniform representation to model hardware building blocks. Each building block is described by a tuple with the following template:

$$\langle \text{param}, s_{\text{in}}, s_{\text{out}}, c_{\text{in}}, c_{\text{out}}, r_{\text{in}}, r_{\text{out}} \rangle$$

where param is a set of block-specific configuration parameters, s_{in} and s_{out} are the number of parallel streams at the input and output of the block, respectively, c_{in} and c_{out} are the number of elements per stream at the input and output of the block, 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*. This parameterization scheme allows us to concisely express different configurations for each building block with potentially different performance-resource characteristics. In this manner, fpgaConvNet is able to leverage the SDF theory and its analytical power by tuning the parameters of an SDFG's building blocks in order to efficiently explore the design space. As examples, the sliding window and the convolution and pooling bank models are presented in the following.

The *sliding window block* takes as input N streams with elements from the input feature maps and outputs N streams of $(K_h \times K_w)$ windows with strides of S_h and S_w along the input feature maps' height and width, respectively. Each input feature map is automatically zero-padded with a pad size of P in hardware. The sliding window block is represented as

$$\left\langle \{N, K_h, K_w, S_h, S_w, P\}, N, N, 1, K_h \times K_w, 1, \frac{1}{S_w} \right\rangle.$$

In *convolution* and *pooling banks*, each of the units performs an operation, which reduces a window of size $(K_h \times K_w)$ to a single value. For convolution banks (Fig. 4), the operation

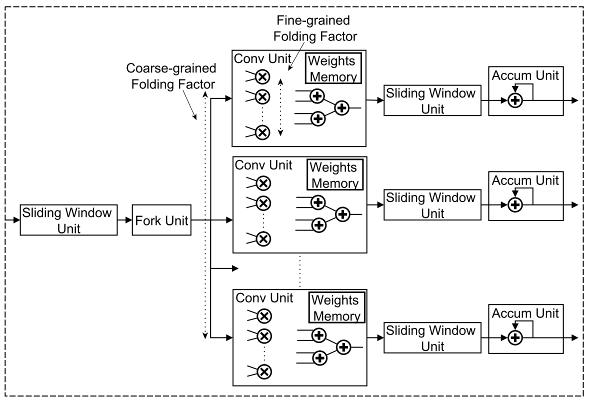


Fig. 4. Convolutional layer mapped to building blocks.

of each of the N units is a dot product between the input window and the corresponding weights. The input and output rates depend on the folding factor of the units, denoted by f. If $f < 1$, then the specified unit uses time-multiplexing of its multiply–accumulate (MAC) resources to compute a dot product. For pooling banks, two pooling operations are supported: average and max pooling. In the case of average pooling, dot product units are used with averaging kernels, and therefore, the average pooling banks are configured similar to convolution banks. In the case of max-pooling banks, finding the maximum is performed with a single comparator and f is equal to $((1)/((K_h \times K_w)))$ with a single pixel being consumed per cycle. The tuple representation is as follows:

$$\langle \{N, K_h, K_w, f\}, N, N, K_h \times K_w, 1, f, f \rangle.$$

The rest of the hardware blocks are defined in a similar manner following the same modeling approach.

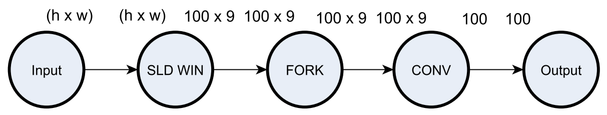
B. Modeling ConvNet Workloads

Complying with our interpretation of ConvNets as streaming applications, ConvNet workloads are internally captured by means of SDF. A ConvNet workload is represented as a stream of data flowing through a sequence of building blocks. Each building block consumes an amount of data at its consumption rate and produces new ones at its production rate. By creating a matrix whose columns hold the local workload at the input and output of each building block in the architecture, a compact and distributed representation of the computational workload can be constructed. The amount of work, W_i^{in} and W_i^{out}, carried out by the ith hardware block is defined as the total number of data elements to be consumed and produced by this block, respectively, and it is expressed as

$$W_i^{\{\text{in, out}\}} = F_{\text{map},i}^{\{\text{in, out}\}} \cdot P_i^{\{\text{in, out}\}}$$

where $F_{\text{map},i}^{\{\text{in, out}\}}$ is the number of feature maps and $P_i^{\{\text{in, out}\}}$ is the number of data elements per feature map at the input or output of the ith block, respectively. To populate the workload along the SDFG of building blocks, we introduce the *feature maps matrix*, $\boldsymbol{F}_{\text{map}}$, and the *data matrix*, $\boldsymbol{P}$, and form the *workload matrix*, $\boldsymbol{W}$, as shown in the following:

$$\boldsymbol{W} = \boldsymbol{F}_{\text{map}} \odot \boldsymbol{P}. \tag{6}$$



$$\boldsymbol{W} = \begin{bmatrix} h\times w & h\times w & 0 & 0 & 0 \\ 0 & 100\times 9 & 100\times 9 & 0 & 0 \\ 0 & 0 & 100\times 9 & 100\times 9 & 0 \\ 0 & 0 & 0 & 100 & 100 \end{bmatrix}$$

Fig. 5. Convolution workload graph and matrix representation.

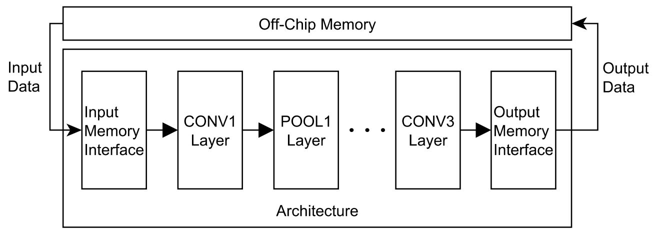


Fig. 6. Overview of streaming architecture.

As an example, the workload of a convolution between a single $(h \times w)$ feature map and a single (3×3) kernel would be represented in the following graph and matrix forms (see Fig. 5), where the feature map is assumed to have 100 (3×3) windows. In this case, the feature maps and data matrices would be populated as follows:

$$\boldsymbol{F}_{\text{map}} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 100 & 100 & 0 & 0 \\ 0 & 0 & 100 & 100 & 0 \\ 0 & 0 & 0 & 100 & 100 \end{bmatrix}$$

$$\boldsymbol{P} = \begin{bmatrix} h\times w & h\times w & 0 & 0 & 0 \\ 0 & 9 & 9 & 0 & 0 \\ 0 & 0 & 9 & 9 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

VI. Architectural Design Space

In our framework, the basic hardware mapping of an SDFG is a streaming architecture, as shown in Fig. 6. In this setting, the design space is determined by the design parameters of each instantiated block. The complete architectural design space captured by our framework is formed by defining a set of legal graph *transformations* for the manipulation of SDFGs. The legality of a transformation is defined as the functional equivalence of the graph before and after the transformation. Four types of transformations are defined: *graph partitioning with reconfiguration*, *coarse*-grained folding, *fine-grained folding*, and *weights reloading*.

A. Graph Partitioning With Reconfiguration

A direct hardware mapping of a given ConvNet assumes that the on-chip computational and storage resources of the target platform are able to accommodate it. In practice, the exploitation of the inherent parallelism of the ConvNet can be limited by the target FPGA's computational resources, namely, the amount of LUTs and DSPs. Moreover, the on-chip storage requirements can scale rapidly with an increase in either a layer's width or the ConvNet's depth. In such scenarios, there is an excessive amount of trained weights which may exceed the available on-chip memory.

With the state-of-the-art ConvNet models reaching new records in terms of depth [5], [8], dealing with large-scale networks and their high resource demands becomes a crucial factor. One of the most commonly used techniques in the systems literature to deal with this issue is the design of a single computation engine that is time-shared across layers [21]. Such a design comprises a single programmable accelerator that is reused between layers and operates under the control of software. This design approach fixes the architecture of the accelerator, and let us the software perform the ConvNet mapping by sending instructions to the accelerator. Despite the flexibility gains due to software programmability, such a design adds costly overheads by introducing inefficiencies due to control mechanisms that resemble those of a processor [58] and hence does not fully leverage the parallelism and customization potential of each particular ConvNet.

Our proposed alternative to this problem exploits the reconfigurability capabilities of FPGAs and introduces the partitioning of the ConvNet along its depth. In the *graph partitioning with reconfiguration* transformation, the original SDFG is split into several subgraphs. Each subgraph is mapped to a distinct hardware architecture, specifically optimized for the particular subgraph, which can utilize all of the FPGA resources. In each subgraph, the on-chip memory is used for storing weights and buffering feature maps between building blocks. Moreover, the communication with the off-chip memory is minimized and encompasses only the subgraph's input and output streams.

The design parameter of this transformation is the selection of the partition points of the input ConvNet. Given a ConvNet with N_L layers, there are $N_L - 1$ candidate reconfiguration points. We form a partitioning vector $\boldsymbol{p} \in \{0,1\}^{N_L-1}$, where a value of 1 for the ith element indicates that the SDFG will be partitioned at the ith layer. For a total of $N_P - 1$ partition points, there are N_P subgraphs, each one having its own topology matrix and hardware design. More formally, we define a topology tuple that contains one topology matrix per subgraph

$$\Gamma = \langle \boldsymbol{\Gamma}_i \mid i \in [1, N_P] \rangle. \tag{7}$$

Fig. 7 shows an example of a design point that includes two partition points and effectively three architectures. The ConvNet is partitioned after the pooling layers with $\boldsymbol{p} = [0\ 1\ 0\ 1]^{\top}$, leading to three subgraphs and a topology tuple of $< \boldsymbol{\Gamma}_1, \boldsymbol{\Gamma}_2, \boldsymbol{\Gamma}_3 >$. Each subgraph is mapped to a dedicated architecture, which is optimized specifically for the subgraph's workload. Between the execution of successive subgraphs, the whole FPGA is reconfigured with the corresponding design.

This approach requires the reconfiguration of the whole FPGA whenever data have to enter a different subgraph which adds a substantial time overhead. The reconfiguration overhead can be amortized when batch processing is employed with several inputs processed as a batch. With this strategy, we introduce the design points that employ

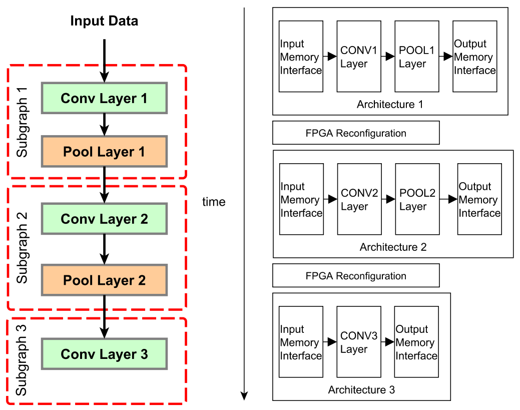


Fig. 7. Example of graph partitioning with reconfiguration.

FPGA reconfiguration into the design space and expand the design capabilities of FPGA-based ConvNet systems. As a result, fpgaConvNet is able to employ this technique to provide high-throughput mappings in scenarios where the latency of a single input is not critical for the application and batch processing can be tolerated.

B. Coarse- and Fine-Grained Folding

In ConvNet execution, high performance is mainly achieved by exploiting two types of parallelism [29]. The first type is the parallel execution of the coarse operations at each layer. This includes the parallel execution of all the convolutions in a convolutional layer, pooling operations in a pooling layer, and nonlinearities in a nonlinear layer, and is equivalent to the parallelism across output feature maps. In this context, we define the *coarse-grained folding* of a layer by treating the coarse unroll factor[3] of each layer as a tunable parameter.

Formally, for an SDFG with N_L layers, a coarse-grained folding vector $\boldsymbol{f}_{\text{coarse}} \in (0,1]^{N_L}$ is defined with one folding factor for each layer. More specifically, with reference to the building block models presented in Section V-A, a convolutional, pooling, or nonlinear layer with N_{out} output feature maps is mapped to the corresponding bank block. The coarse-grained folding is controlled by parameter $N \in [1, N_{\text{out}}]$ of the bank model, which corresponds to the actual number of parallel units that will produce the N_{out} feature maps. As a result, the folding factor of the ith layer $\boldsymbol{f}_{\text{coarse}}(i)$ lies in the range $[((1)/(N_{\text{out}})), 1]$ with 1 for a fully parallel implementation and $((1)/(N_{\text{out}}))$ for a single time-shared unit.

The second type of parallelism is the parallel execution of multiplications and additions of the dot product operations inside the convolution and average pooling units. The implementation of a dot product unit with inputs of size N can span from a fully parallel implementation, with a stage of N multipliers followed by an adder-tree with $N-1$ adders, down to a single MAC unit. After the pipeline depth has been filled, the first implementation yields a throughput of 1 dot product/cycle. On the other end of the spectrum,

[3]The coarse unroll factor is defined as the number of parallel coarse units for the execution of a layer.

Algorithm 1 Coarse-Grained Folding Transformation

```
Inputs:
1: Matrix $\boldsymbol{\Gamma}$
2: Index $i$ of the layer to be folded
3: Number of output feature maps $N_{\text{out}}$ of the layer to be folded
4: Folding factor, $f \in [\frac{1}{N_{\text{out}}}, 1]$
Steps:
1: - Initialise folding vector, $\boldsymbol{f}_{coarse} \in \mathbb{R}^{\#cols\boldsymbol{\Gamma}}$: $\boldsymbol{f}_{coarse} = \boldsymbol{1}$
2: - $\boldsymbol{f}_{coarse}(i) = f$
3: - Form the folding matrix, $\boldsymbol{F} = diag(\boldsymbol{f}_{coarse})$
4: - Apply the coarse-grained folding, $\boldsymbol{S}' = \lceil \boldsymbol{S} \cdot \boldsymbol{F} \rceil$
5: - Form the folded topology matrix, $\boldsymbol{\Gamma}' = \boldsymbol{S}' \odot \boldsymbol{C} \odot \boldsymbol{R}$
   Note: $\lceil \cdot \rceil$ is defined as the element-by-element ceiling operator
```

Algorithm 2 Fine-Grained Folding Transformation

```
Inputs:
1: Matrix $\boldsymbol{\Gamma}$
2: Index $i$ of the layer to be folded
3: Kernel size $K$ or pooling size $P$
4: Folding factor, $f \in [\frac{1}{K^2}, 1]$ or $[\frac{1}{P^2}, 1]$
Steps:
1: - Initialise folding vector, $\boldsymbol{f}_{fine} \in \mathbb{R}^{\#cols\boldsymbol{\Gamma}}$: $\boldsymbol{f}_{fine} = \boldsymbol{1}$
2: - $\boldsymbol{f}_{fine}(i) = f$
3: - Form the folding matrix, $\boldsymbol{F} = diag(\boldsymbol{f}_{fine})$
4: - Apply the coarse-grained folding, $\boldsymbol{R}' = \boldsymbol{R} \cdot \boldsymbol{F}$
5: - Apply the fine-grained folding, $\boldsymbol{\Gamma}' = \boldsymbol{S} \odot \lceil \boldsymbol{C} \odot \boldsymbol{R}' \rceil$
   Note: $\lceil \cdot \rceil$ is defined as the element-by-element ceiling operator
```

the throughput is $(1/N)$ dot products/cycle with approximately N times fewer resources. We define the parameterization over the unroll factor of dot product units as the *fine-grained folding* of a layer. Overall, the design parameters of the two folding transformations are the elements of the folding vectors $\boldsymbol{f}_{\text{coarse}}$ and $\boldsymbol{f}_{\text{fine}}$. An illustration of the coarse- and fine-grained folding factors for a convolutional layer is shown in Fig. 4.

Our adoption of the SDF paradigm allows us to employ linear algebra to express these transformations. The coarse- and fine-grained folding transformations are applied directly on the topology matrix $\boldsymbol{\Gamma}$ by means of the two folding vectors as described by algorithms (1) and (2), respectively. The coarse-grained folding operates on the streams matrix $\boldsymbol{S}$ and the fine-grained folding operates on the rates matrix $\boldsymbol{R}$. Algorithm (1) takes as inputs matrix $\boldsymbol{\Gamma}$ of the given SDFG, the index i of the layer to be folded, the parameter N_{out} of the layer, and the selected folding factor f. On lines 1 and 2, the folding vector is initialized to a vector of ones with N_L elements, where N_L is equal to the number of columns of $\boldsymbol{\Gamma}$, and the ith element is set to the selected folding factor. Next, on lines 3 and 4, a folding matrix is constructed with $\boldsymbol{f}_{\text{coarse}}$ along its diagonal, and it is used to right-multiply matrix $\boldsymbol{S}$. Finally, the folded topology matrix $\boldsymbol{\Gamma}'$ is produced

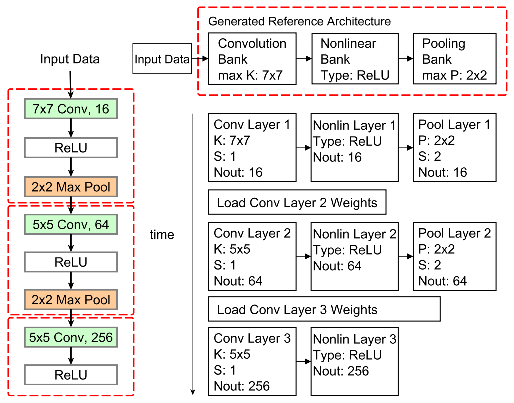
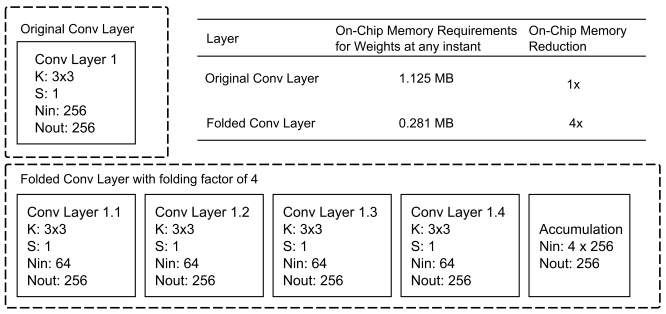

Generated Reference Architecture
Input Data
Input Data
Convolution Bank max K: 7x7
Nonlinear Bank Type: ReLU
Pooling Bank max P: 2x2
7x7 Conv, 16
ReLU
2x2 Max Pool
5x5 Conv, 64
ReLU
2x2 Max Pool
5x5 Conv, 256
ReLU
time
Conv Layer 1 K: 7x7 S: 1 Nout: 16
Nonlin Layer 1 Type: ReLU Nout: 16
Pool Layer 1 P: 2x2 S: 2 Nout: 16
Load Conv Layer 2 Weights
Conv Layer 2 K: 5x5 S: 1 Nout: 64
Nonlin Layer 2 Type: ReLU Nout: 64
Pool Layer 2 P: 2x2 S: 2 Nout: 64
Load Conv Layer 3 Weights
Conv Layer 3 K: 5x5 S: 1 Nout: 256
Nonlin Layer 3 Type: ReLU Nout: 256

Fig. 8. Example of the weights reloading transformation.


Original Conv Layer
Conv Layer 1 K: 3x3 S: 1 Nin: 256 Nout: 256
Folded Conv Layer with folding factor of 4
Conv Layer 1.1 K: 3x3 S: 1 Nin: 64 Nout: 256
Conv Layer 1.2 K: 3x3 S: 1 Nin: 64 Nout: 256
Conv Layer 1.3 K: 3x3 S: 1 Nin: 64 Nout: 256
Conv Layer 1.4 K: 3x3 S: 1 Nin: 64 Nout: 256
Accumulation Nin: 4 x 256 Nout: 256

Layer	On-Chip Memory Requirements for Weights at any instant	On-Chip Memory Reduction
Original Conv Layer	1.125 MB	1x
Folded Conv Layer	0.281 MB	4x

Fig. 9. Example of applying input feature maps folding (assuming 16-bit fixed-point precision for weights).

following (5). A similar procedure is followed for the fine-grained folding, as shown in algorithm (2), where the rates matrix $\boldsymbol{R}$ is affected on line 4 in place of $\boldsymbol{S}$.

C. Weights Reloading

So far, the presented transformations yield a design space that is appropriate for high-throughput applications. The graph partitioning with reconfiguration transformation enables the generation of a distinct architecture for each subgraph, with each architecture further tailored to the workload of its subgraph using coarse- and fine-grained folding. In high-throughput applications that allow batch processing, grouping the inputs in large enough batches spreads the FPGA reconfiguration time cost across the batch. This approach enables the amortization of the FPGA reconfiguration and can effectively lead to high throughput. However, the latency of a single input remains substantially deteriorated. In order to surpass this limitation and target low-latency applications, the *weights reloading* transformation is introduced.

The weights reloading transformation aims to address two issues: 1) to provide a mechanism for the execution of several subgraphs without the latency penalty due to FPGA reconfiguration and 2) to enable the targeting of layers with weights that exceed the FPGA on-chip memory capacity, which is handled by *input feature maps folding*, as detailed in Section VI-C1. Similar to graph partitioning with reconfiguration, this transformation partitions a given SDFG into several subgraphs along its depth. However, instead of generating a distinct architecture for each subgraph, a single flexible architecture is derived that can execute the workloads of all the resulted subgraphs by operating in different modes.

Fig. 8 shows a typical operation of the weights reloading transformation. In this scenario, the SDFG on the left is partitioned into three subgraphs. Each subgraph has its own trained weights for its convolutional layer. A reference architecture is derived based on the layer patterns of the subgraphs as detailed in Section VI-C2, and each subgraph's workload is scheduled for execution. Moreover, reloading of weights from the off-chip memory is performed between the executions of successive subgraphs. After the reference architecture has been derived, the hardware design is further tuned by means of the coarse- and fine-grained folding transformations. These two transformations are employed to holistically optimize the design across the workloads of all the scheduled subgraphs.

The weights reloading transformation enables the execution of latency-critical applications and expands the design space with design points that resemble flexible programmable processors. The primary differentiating factor of our approach from existing programmable processors with a fixed architecture, such as [59], is that the reference design is formed based on the layer patterns of the input ConvNet, which avoids the overheads of a generic architecture. After the reference design has been derived, the rest of the transformations are utilized for further cross-subgraph optimization of the hardware.

1) Input Feature Maps Folding: The memory requirements for the weights of even a single convolutional layer can exceed the on-chip memory capacity of the target device. In such scenarios, ConvNets would become bounded by the limited on-chip memory. To address this issue and accommodate ConvNets with a large amount of weights, an additional design parameter is introduced in the form of an *input feature maps folding factor*, coupled to the weights reloading transformation. One input feature maps folding factor $f_{\text{in}} \in [1, N_{\text{in}}]$ is associated with each convolutional layer in a subgraph. Depending on the value of the folding factor, a convolutional layer with N_{in} input and N_{out} output feature maps is divided into N_W subgraphs that perform a fraction of the convolutions, where $N_W = f_{\text{in}}$. For a factor of f_{in}, each of the N_W subgraphs performs $(N_{\text{in}}/f_{\text{in}})N_{\text{out}}$ convolutions and computes N_{out} intermediate convolution results. After all the N_W subgraphs have been executed, the intermediate results are accumulated in order to produce the output feature maps.

Fig. 9 shows an instance of a convolutional layer with 256 input and output feature maps. By applying input feature maps folding with a factor of 4, each subgraph is responsible for 64 input feature maps and produces 256 intermediate feature maps with the final accumulation subgraph summing them to produce the output feature maps. With this technique, the on-chip memory footprint at any instant is 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.

2) Reference Architecture Derivation: After the weights reloading partition points have been selected, a single, flexible reference architecture is derived that is able to execute all the subgraphs. The requirements of the derived architecture are: 1) the capability of executing the layer patterns that are present in any of the subgraphs and 2) run-time flexibility with respect to its data path so that no FPGA reconfiguration will be required.

The reference architecture derivation is cast as a pattern matching problem. As a first step, the SDFG is partitioned into N_W subgraphs based on the selected partition points. The sequence of hardware layers of each subgraph is interpreted as a pattern. The elements of the patterns are drawn from the set of supported layers. The reference architecture is initialized with the deepest subgraph as a starting point and is refined by looping over the patterns of the rest of the subgraphs and searching for the occurrence of each pattern in the reference architecture. Each pattern occurs with a shift s if $0 \leq s \leq len(\text{ref}) - len(\text{pattern})$, where $len(\cdot)$ returns the depth of its input. In the case when a pattern does not occur, the missing pattern is added at the end of the reference architecture's pipeline.

By the end of this process, a reference architecture has been formed, which can execute the workloads of all the subgraphs. Flexibility with respect to data path is introduced by means of run-time configurable interconnections among the instantiated building blocks. This flexibility allows the architecture to process the workloads of different subgraphs by forming the appropriate data path based on the current subgraph's index without the need for FPGA reconfiguration.

After the formation of the reference architecture, each node of each subgraph is mapped and scheduled on the appropriate building block. The convolution and pooling units are instantiated so that they support the processing of windows of the maximum size that has been scheduled on them, with zero-padding used for smaller windows. As a final step, the coarse- and fine-grained folding transformations are used to further optimize the design in a holistic manner by considering the workloads of all the scheduled subgraphs.

The design parameters of this transformation are the selection of the partition points and the input feature maps folding factors of each convolutional layer. To capture the partitioning points, we form a vector $\boldsymbol{p}_{\text{CONV}} \in \{0,1\}^{N_{\text{CONV}}-1}$. The weights reloading transformation is applied after the initial partitioning of the input ConvNet into N_P subgraphs by the graph partitioning with reconfiguration transformation. Weights reloading further partitions each of the subgraphs, leading to N_{W_i} subgraphs mapped to the ith reference architecture, with $i \in [1, N_P]$. Moreover, for a ConvNet with N_{CONV} convolutional layers, there are N_{CONV} input feature maps folding factors to be selected, i.e., $f_{\text{in},i}$ with $i \in [1, N_{\text{CONV}}]$, organized in a vector $\boldsymbol{f}_{\text{in}} \in [1, N_{\text{in},i}]^{N_{\text{CONV}}}$.

3) Workload Alignment: In the weights reloading transformation, when a workload subgraph is mapped to a reference architecture, its nodes have to be scheduled for execution on the instantiated building blocks. Given the ith reference architecture and the jth workload subgraph with N and L building blocks, respectively, we have a topology matrix $\boldsymbol{\Gamma}_{i,\text{ref}} \in \mathbb{R}^{(M\times N)}$ and a workload matrix $\boldsymbol{W}_{i,j} \in \mathbb{Z}^{(K\times L)}$ with $K \leq M$ and $L \leq N$. Fig. 10 shows an instance where the workload has to be aligned to the reference architecture.

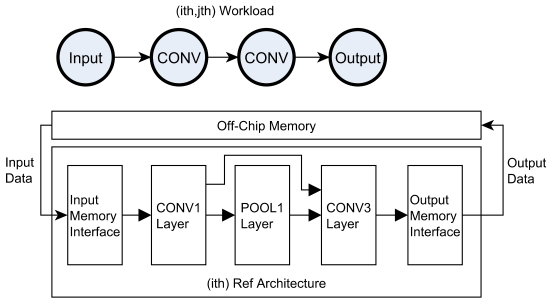


Fig. 10. Workload to be mapped on a reference architecture.

In this case, the workload consists of two convolutional layers, with the first layer mapped directly to the CONV1 block. However, the second layer has to be scheduled on block CONV2. At a matrix level, this corresponds to an alignment of the columns of the workload matrix that represent the second layer with the columns of the topology matrix that represent the CONV2 block. In order to estimate the execution time of the jth subgraph by the ith reference architecture, it is necessary that the columns of $\boldsymbol{W}_{i,j}$ are aligned to the correct columns of $\boldsymbol{\Gamma}_{i,\text{ref}}$. This process can be interpreted as forming the correct data path for the jth subgraph by scheduling each node on the appropriate block of the reference architecture. To smoothly integrate the weights reloading transformation to our SDF streaming model, we introduce an analytical method for applying the weights alignment.

This is achieved by forming a matrix $\boldsymbol{W}_{i,j}^{\text{aligned}} \in \mathbb{Z}^{(M\times N)}$ that contains the rows and columns of $\boldsymbol{W}_{i,j}$ with the correct alignment. Following our SDF model, the workload alignment operation is expressed algebraically, as described by algorithm (3). During the reference architecture derivation task, the necessary alignment shifts for each column of the workload matrices have been calculated and stored in the shift vector $\boldsymbol{s}^{i,j}$ for the jth workload of the ith reference architecture, which can be seen on line 3 of the inputs list. The algorithm starts with the initialization of $\boldsymbol{W}_{i,j}^{\text{aligned}}$ with a zero-padded version of $\boldsymbol{W}_{i,j}$ to match the size of $\boldsymbol{\Gamma}_{i,\text{ref}}$. The loop on line 2 iterates through the columns of the j_{th} workload matrix that need alignment. In the main body of the loop, lines 3–8 shift the current column to the right, i.e., along the coarse pipeline of building blocks in the reference architecture. Next, lines 9–14 downshift the column in order to align the interconnections. After the end of the loop, $\boldsymbol{W}_{i,j}^{\text{aligned}}$ has been fully constructed and the jth initiation interval matrix can be computed correctly as $\boldsymbol{II}_{i,j} = \boldsymbol{W}_{i,j}^{\text{aligned}} \oslash \boldsymbol{\Gamma}_{i,\text{ref}}$ and used to calculate $t_{i,j}(B, \boldsymbol{\Gamma}_{i,\text{ref}}, \boldsymbol{W}_{i,j})$, as described in Section VII-A. As a result, the weights reloading transformation can be applied analytically and integrated to the performance model.

D. Optimizations for State-of-the-Art Irregular Models

To address the ConvNet families of Inception, residual, and dense networks, we extend our framework at two levels. At a modeling level, we extend our SDF model and construct the topology matrix so that it captures multiple connections from one hardware building block to many. This allows us to

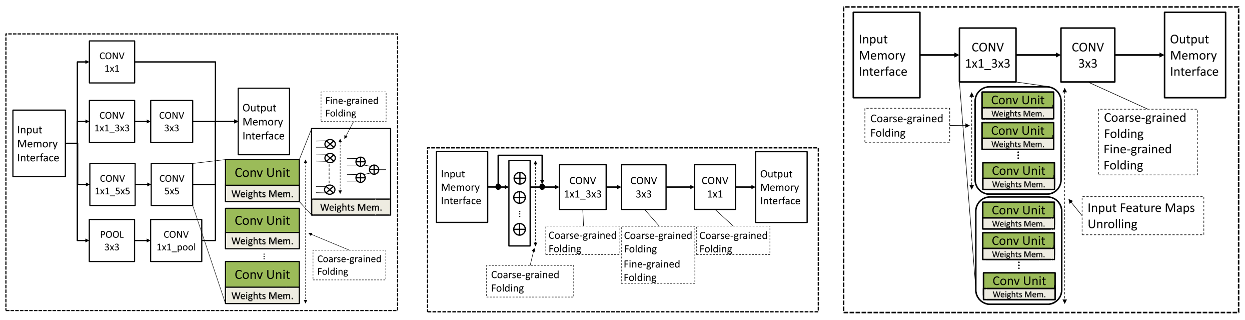


Fig. 11. Dedicated hardware blocks for Inception-based, residual, and dense networks. (a) Multipath hardware Inception block. (b) Hardware residual block with an elementwise adder. (c) Hardware dense block with input feature maps unrolling.

Algorithm 3 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:     $\boldsymbol{A}^r_o = \underbrace{\boldsymbol{A}^r \cdot \boldsymbol{A}^r \cdot \cdots \cdot \boldsymbol{A}^r}_{s^{i,j}_{col}}$
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:    $\boldsymbol{A}^l_o = \underbrace{\boldsymbol{A}^l \cdot \boldsymbol{A}^l \cdot \cdots \cdot \boldsymbol{A}^l}_{s^{i,j}_{col}}$
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.
```

represent analytically the multiple paths and their workloads inside the Inception, residual, and dense blocks. An Inception module is represented by means of a tuple as follows:

$$\left\langle h_{\text{in}}, w_{\text{in}}, N_{\text{in}}, N^{\{1,2,3,4\}}_{\text{out}}, N^{\{1,2,3,4,5,6\}}_{f}, r\right\rangle \tag{8}$$

where h_{in}, w_{in}, and N_{in} are the height, width, and number of input feature maps, N^i_{out} is the number of output feature maps from the ith path, N^i_f is the number of filters in the ith convolutional layer, and r is the repetitions of the module. The residual block is captured as follows:

$$\left\langle h_{\text{in}}, w_{\text{in}}, N_{\text{in}}, N^{\{1,2,3\}}_{f}, r\right\rangle \tag{9}$$

where N^i_f is the number of filters in the ith convolutional layer. Finally, dense blocks are captured as follows:

$$\left\langle h_{\text{in}}, w_{\text{in}}, N_{\text{in}}, N^{\{1,2\}}_{f}, r, k\right\rangle \tag{10}$$

where k is the growth rate that is the number of output feature maps of each convolutional layer in a dense block.

At a hardware level, we design one custom coarse hardware block for each network family that follows the structure of the corresponding novel component.

1) Inception Block: The Inception block [Fig. 11(a)] is parameterized with respect to the coarse-grained folding of each convolutional and pooling layer and the fine-grained folding of the 3×3 and 5×5 convolutional layers, leading to the combined parameter vector $\boldsymbol{f}_{\text{inception}} \in (0, 1]^9$. Moreover, input feature maps folding is defined for the convolutional layers in order to address cases where the weights storage requirements exceed the available on-chip memory.

2) Residual Block: ResNets introduce shortcut connections between layers and combine feature maps by means of elementwise addition. To support this structure, we define a new *elementwise addition* hardware building block and parameterize it with respect to its coarse-grained folding. Fig. 11(b) shows the overall residual block. Its input consists of an optional adder array that implements the shortcut connections, while the rest of the architecture comprises two 1×1 convolutional layers with a 3×3 layer between them. The configuration of the residual block is captured with a vector $\boldsymbol{f}_{\text{residual}} \in (0, 1]^5$ with one fine-grained and four coarse-grained folding factors.

3) Dense Block: In DenseNet, a dense block consists of a series of 1×1 followed by 3×3 convolutional layers. Our proposed hardware dense block comprises the direct mapping of the two layers to hardware with the coarse-grained folding of both and the fine-grained folding of the 3×3 layer as compile-time parameters. Dense blocks have the property of increasing the number of input feature maps by the growth rate k at each iteration while keeping the number of output feature maps constant and equal to the growth rate. In this respect, the parallelism of the output feature maps remains constant along the dense block, while the input feature maps parallelism increases. With reference to tuple (10), inside a dense block, each 3×3 layer produces k output feature maps, while the ith repetition of (1×1 and 3×3) receives $N_{\text{in}} + (i-1)k$ input feature maps with $i \in [1, r]$. To exploit this property, we define a DenseNet-specific optimization and extend the dense block with an additional parameter that unrolls the 1×1 layer with respect to its input feature maps [Fig. 11(c)]. This approach allows us to sustain high utilization of the FPGA resources as the number of input feature maps increases inside a dense block. The configuration of the dense block is represented with a vector $\boldsymbol{f}_{\text{dense}} \in (0, 1]^4$ that includes one fine-grained folding, two coarse-grained folding, and one input unrolling factors.

Overall, the configuration parameters of the blocks are exposed to the SDF transformations in order to tailor the custom blocks to the workload of the target Inception module, residual block, and dense block.

VII. Design Space Exploration

Based on the parameterization of the SDF transformations and the hardware building blocks, fpgaConvNet defines a particular architectural design space. An analytical performance model has been developed as an estimator of the throughput and latency of each design point. The DSE task is cast as a constrained optimization problem with the objective to optimize the performance metric of interest. The design space is traversed by means of the SDF transformations until a design point is obtained that optimizes the target objective.

A. Performance Model

Given the topology matrix $\boldsymbol{\Gamma}$ of a design point and the workload matrix $\boldsymbol{W}$ of a ConvNet, the *initiation interval matrix* $\boldsymbol{II}$ is formed as follows:

$$\boldsymbol{II} = \boldsymbol{W} \oslash \boldsymbol{\Gamma} \tag{11}$$

where $\oslash$ denotes the Hadamard elementwise division. Each element of $\boldsymbol{II}$ gives the number of cycles required by each hardware block along the pipeline to consume its workload. The block with the longest initiation interval determines the initiation interval of the whole SDFG and is given by the maximum element of $\boldsymbol{II}$, denoted by $II^{\max}$.

For the ConvNet inference over a batch of B inputs, the execution time for a single subgraph is estimated as

$$t(B, \boldsymbol{\Gamma}, \boldsymbol{W}) = \frac{1}{\text{clock rate}} \cdot (D + II^{\max} \cdot (B-1)) \tag{12}$$

where D is the maximum between the size of the input, e.g., the number of pixels of an image, and the pipeline depth of the current hardware design.

Graph partitioning with reconfiguration determines the number of distinct architectures of a design point. For N_P partitions, there are N_P architectures where the ith architecture is associated with its $\boldsymbol{\Gamma}_i$ and $\boldsymbol{W}_i$ matrices. Furthermore, the weights reloading transformation partitions $\boldsymbol{W}_i$ 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 architecture represented by $\boldsymbol{\Gamma}_{i,\text{ref}}$. To support design points that employ these two transformations, each design point is expressed by a topology tuple Γ as defined in (7), and similarly, the workload matrix $\boldsymbol{W}$ is replaced by a tuple defined as follows:

$$W = \langle \boldsymbol{W}_{i,j} \mid i \in [1, N_P], j \in [1, N_{W_i}] \rangle \tag{13}$$

We extend the execution time notation with $t_{i,j}$ in order to capture the execution time of the jth workload subgraph on the ith architecture. Moreover, between successive subgraphs, the weights transfer time from the off-chip to the on-chip memory of the jth workload subgraph of the ith architecture has to be included and is denoted by $t_{i,j,\text{weights}}$. The weights transfer time is estimated using the amount of weights in the subgraph and the allocated bandwidth of the target platform. Moreover, between consecutive architectures, the reconfiguration time, $t_{i,\text{recon fig.}}$, has to be included. With this formulation, the overall execution time is expressed as

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

The above expression indicates that the reconfiguration and weights reloading time are independent of the batch size, B. Therefore, by increasing the batch size, the first term dominates the execution time and the reconfiguration and weights reloading overheads are amortized. In practice, the value of B is limited by the capacity C_{mem} of the off-chip memory and the latency tolerance of the application. Given a ConvNet that requires a total of W_{ConvNet} GOps/input, the throughput and latency of a design point can be estimated as in (14) and (15) in GOp/s and seconds, respectively

$$T(B, \Gamma, W) = \frac{W_{\text{ConvNet}}}{t_{\text{total}}(B, \Gamma, W)/B} \tag{14}$$

$$L(B=1, \Gamma, W) = t_{\text{total}}(1, \Gamma, W). \tag{15}$$

B. Resource Consumption Model

The primary factor that constrains the ConvNet mapping on a particular platform is the available resources. Each candidate design point has a corresponding resource consumption. We define the *feasible space* of our model as the set of design points that satisfy all the platform-specific resource constraints. To estimate the FPGA resource utilization of a design point, we construct an empirical model based on place-and-route results. To this end, we use a set of LUTs, FFs, DSP blocks, and BRAMs measurements for each hardware building block and fit linear regression models as a function of their tunable parameters, leading to a set of predictive resource models.

C. Optimization Framework

Our SDF modeling framework allows us to formulate the DSE task as a constrained combinatorial optimization problem. Three distinct optimization problems are formed, which differ in terms of objective function and constraints based on the performance metric of interest

$$\max_{\Gamma} \; T(B,\Gamma,W), \quad \text{s.t. } \boldsymbol{rsc}(B,\Gamma) \leq \boldsymbol{rsc}_{\text{Avail.}} \tag{16}$$

$$\min_{\Gamma} \; L(\;1,\Gamma,W), \quad \text{s.t. } \boldsymbol{rsc}(\;1,\Gamma) \leq \boldsymbol{rsc}_{\text{Avail.}} \tag{17}$$

$$\max_{\Gamma} \; T(B,\Gamma,W), \quad \text{s.t. } \boldsymbol{rsc}(B,\Gamma) \leq \boldsymbol{rsc}_{\text{Avail.}} \tag{18}$$
$$L(1,\Gamma,W) \leq \epsilon$$

where T, L, $\boldsymbol{rsc}$, and ϵ are the throughput in GOp/s, the latency in s/input, the resource consumption vector of the current design point and the upper bound on the latency, respectively, and $\boldsymbol{rsc}_{\text{Avail.}}$ is the resource vector of the target platform. The objective function aims to either: 1) maximize throughput [see (16)]; 2) minimize latency [see (17)]; or 3) perform an MOO which maximizes throughput with a latency constraint [see (18)].

Given an input ConvNet, the optimization problems are defined over the set of all design points S in the design space presented in Section VI, and the objective functions $T : S \to \mathbb{R}^+$ and $L : S \to \mathbb{R}^+$ can be evaluated for all $s \in S$ given the performance model of Section VII-A. In theory, the optimal design point could be obtained by means of an exhaustive search with a complete enumeration. The total number of design points to be explored, given all four transformations can be calculated as shown in the following:

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

where N_L is the number of layers, N_{CONV} is the number of convolutional layers, $N_{\text{reload},i}$ is the number of possible input feature maps folding factors for the ith convolutional layer, and $N_{\text{coarse},i}$ and $N_{\text{fine},i}$ are the number of possible coarse- and fine-grained folding factors for the ith layer, respectively. With an increase in either the depth or width of a ConvNet's layer, brute-force enumeration quickly becomes computationally intractable. Therefore, a heuristic method is adopted to obtain a solution in the nonconvex space.

In this paper, simulated annealing [60] has been selected as the basis of the developed optimizer. The SDF transformations, defined in Section VI, are formalized as a set of operations Σ and the neighborhood $N(s,\sigma)$ of a design point s is defined as the set of design points that can be reached from s by applying one of the operations $\sigma \in \Sigma$. The optimizer traverses the design space by considering all the described transformations and converges to a solution of the objective function, selected from (16)–(18).

The MOO problem of (18) involves a reduction to a single objective by means of an ϵ-constraint formulation. This approach incorporates the application-specific importance of throughput and latency prior to optimization and is solved by the developed simulated annealing optimizer. Alternatively, other methods, such as genetic algorithms, can be employed to solve the MOO problem [61]. Such an optimization engine would first search the design space and generate a set of design points lying on the throughput-latency Pareto front of the target ConvNet-FPGA pair. As a second step, the application-specific throughput and latency requirements would be considered *a posteriori* to select the highest performing design from the generated solution set.

TABLE I
FPGA PLATFORMS

Platform	Processor	LUTs	Flip-Flops	DSPs	BRAM
Zynq 7020	ARM Cortex A9	53,200	106,400	220	0.63 MB
Zynq 7045	ARM Cortex A9	218,600	437,200	900	2.40 MB

TABLE II
BENCHMARKS

Model Name		Conv Layers	Workload	Weights
AlexNet	[34]	5	1.3315 GOps	2.3 M
VGG16	[1]	13	30.7200 GOps	14.7 M
GoogLeNet	[7]	22	3.1458 GOps	5.8 M
ResNet-152	[5]	152	23.0232 GOps	55.2 M
DenseNet-161	[8]	161	13.7590 GOps	24.1 M

VIII. EVALUATION

A. Experimental Setup

In our experiments, we target two FPGA platforms with different resource characteristics (Table I): Avnet's ZedBoard mounting the low-end Zynq 7020 and the Xilinx's ZC706 board mounting the larger Zynq 7045. Both platforms are based on the Xilinx Zynq-7000 System-on-Chip which integrates a dual-core ARM Cortex A9 CPU alongside an FPGA fabric on the same chip. Our framework uses the SDFG of each hardware design to automatically generate synthesizable Vivado HLS code. All hardware designs were synthesized and placed-and-routed with Xilinx's Vivado Design Suite (v17.2) and run on ZedBoard and ZC706 board with an operating frequency of 125 MHz. The achieved clock rate is currently limited by the technology of the target device and the use of HLS, which relies on the vendor's toolchain and does not allow for low-level optimizations to overcome critical path issues. The ARM CPU was used to measure the performance of each design. fpgaConvNet provides support for custom fixed-point as well as floating-point precision. For the evaluation, Q8.8 16-bit fixed-point precision was used following the practice of the FPGA works we compare with. Moreover, research on the precision requirements of ConvNet inference [23] has shown Q8.8 to give similar results to 32-bit floating point.

1) Benchmarks: Table II lists our benchmark models. Each ConvNet consists of a feature extractor and has a different number of layers, computation, and memory requirements, and has been selected to pose a different design challenge. AlexNet comprises nonuniform kernel sizes across its convolutional layers, including 11×11, 5×5, and 3×3 kernels. VGG16 is one of the largest and more computationally intensive ConvNets, whose pretrained feature extractor is extensively used as a building block in new application domains [62]. Finally, GoogLeNet, ResNet-152, and DenseNet-161 represent the mainstream networks that contain novel complex components and challenge the mapping by having irregular dataflow.

With the evaluation of the performance model's accuracy with respect to the real measured performance presented in [13] and [14], the rest of this section focuses on the comparison with: 1) highly optimized designs targeting an embedded GPU and 2) state-of-the-art ConvNet designs on FPGAs.

B. Comparison With Embedded GPU

With the majority of ConvNets being deployed for inference in embedded systems, our evaluation focuses on the embedded space. In power-constrained applications, the main metrics of interest comprise: 1) the absolute power consumption and 2) the performance efficiency in terms of performance per watt. In this respect, we investigate the performance efficiency of fpgaConvNet designs on Zynq 7045, which is an industry standard for FPGA-based embedded systems, in relation to the widely used high-performance NVIDIA Tegra X1 platform.

For the performance evaluation on Tegra X1, we use NVIDIA TensorRT as supplied by the JetPack 3.1 package. TensorRT is run with the NVIDIA cuDNN library and FP16 precision, which enables the highly optimized execution of layers. Across all the platforms, each ConvNet is run 100 times to obtain the average throughput and latency. Furthermore, power measurements for the GPU and FPGAs are obtained via a power monitor on the corresponding board. In all cases, we subtract the average idle power[4] from the measurement to obtain the power due to the benchmark execution.

1) Throughput-Driven Applications: In throughput-driven applications, multiple inputs can be processed as a batch to increase the overall throughput. For these scenarios, our framework utilizes the throughput-driven objective function [see (16)] during DSE. On all evaluated platforms, each benchmark is run with a favorable batch size in order to reach peak throughput.

2) Latency-Driven Applications: In latency-driven scenarios, batch processing is not an option, and hence, the application performance is determined by how fast a single input is processed. In this case, our framework employs the latency-driven objective function [see (17)] during DSE in order to generate low-latency accelerators. On all evaluated platforms, the benchmarks are run with a batch size of 1.

3) Discussion: Tegra X1 mounts a 256-core GPU with native support for FP16 half-precision floating-point arithmetic which can be configured with a range of frequencies up to 998 MHz at a peak power consumption of 15 W. To investigate the performance of each platform under the same absolute power constraints that would be present in an embedded setting, we configure the frequency of the GPU with 76.8 MHz and the target Zynq 7045 FPGA at 125 MHz for the same budget of 5 W. For throughput-driven applications, fpgaConvNet achieves a throughput improvement over Tegra X1 of up to 5.53× with an average of 3.32× (3.07× geo. mean) across the benchmarks. For latency-driven scenarios, fpgaConvNet demonstrates a throughput improvement of up to 6.65× with an average of 3.95× (3.43× geo. mean).

[4]Idle Power: Tegra X1 (5 W), ZedBoard (5 W), and ZC706 (7 W).

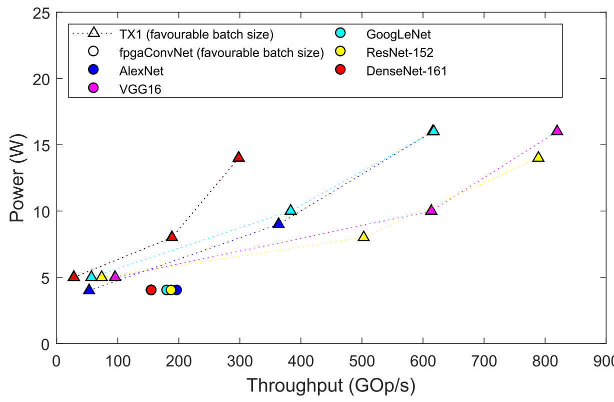


Fig. 12. Power-performance space for fpgaConvNet's Zynq 7045 and Tegra X1 designs clocked at 76.8, 537.6, and 998 MHz (high-throughput designs with batching). For the same power budget (5 W), fpgaConvNet achieves 3.07× (geo. mean) higher performance across the benchmark ConvNets.

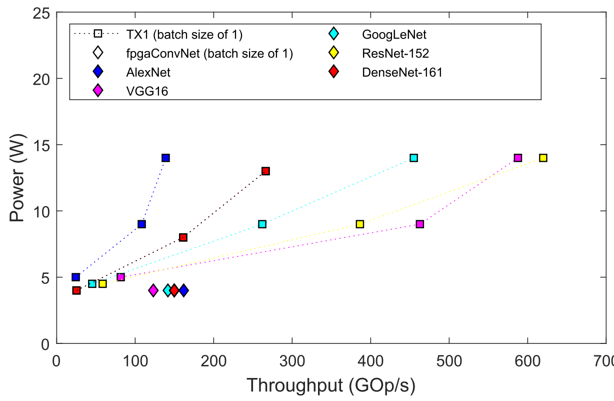


Fig. 13. Power-performance space for fpgaConvNet's Zynq 7045 and Tegra X1 designs clocked at 76.8, 537.6, and 998 MHz (low-latency designs with a batch size of 1). For the same power budget (5 W), fpgaConvNet achieves 3.43× (geo. mean) higher performance across the benchmark ConvNets.

To evaluate the performance efficiency in terms of performance per watt, we configure the GPU with the peak rate of 998 MHz. In this setting, fpgaConvNet achieves an average of 1.17× (1.12× geo. mean) improvement in GOp/s/W over Tegra X1 for throughput-driven applications and 1.70× (1.36× geo. mean) for latency-driven applications. Figs. 12 and 13 show the measured power-performance space on Tegra X1 with different frequency configurations (76.8, 537.6, and 998 MHz) and Zynq 7045 at 125 MHz for throughput-driven and latency-driven applications, respectively. Based on the presented evaluation, fpgaConvNet demonstrates gains in average performance per watt across the benchmarks and reaches higher raw performance over highly optimized embedded GPU mappings when operating under the same power budget.

C. Comparison With Existing FPGA Designs

This section explores the performance of the proposed framework with respect to existing FPGA work. This is investigated by comparing with a set of state-of-the-art works that

TABLE III
COMPARISON WITH EXISTING FPGA WORK ON REGULAR MODELS

	[48] AlexNet[5]		[49] AlexNet	[50] AlexNet	This Work: AlexNet		[49] VGG16	[53] VGG16	[23] VGG16	This Work: VGG16	
FPGA	Zynq-7020	Zynq-7045	Zynq-7020	Virtex-7 690T	Zynq-7020	Zynq-7045	Zynq-7020	Arria-10 GX1150	Zynq-7045	Zynq-7020	Zynq-7045
Clock (MHz)	100	100	150	100	125	125	150	200	150	125	125
DSP Util.†	-	-	63.64%	61%	74.54%	99.55%	63.64%	100%	89.2%	90%	95%
GOp/s*	18.53	108.25	20.16	135.00	38.30	197.40	31.35	720.15	187.80	48.53	155.81
GOp/s/Logic	0.35	0.4952	0.38	0.3116	0.72	0.90	0.59	0.8591	1.6857	0.91	0.7127
GOp/s/DSP†	0.0842	0.12	0.0916	0.0619	0.17	0.22	0.1425	0.2296	0.21	0.22	0.17
GOp/s/W‡	-	-	10.08	-	21.88	49.35	15.67	-	71.4	27.73	38.95
Latency**	71.75 ms	12.30 ms	-	-	52.4 ms	8.22 ms	-	42.98 ms	163.42 ms	633 ms	249.50 ms

* favourable batch size, ** batch size = 1, † 18×18, 19×18 and 25×18 DSP configurations, ‡ power due to benchmark execution with subtracted idle power.

TABLE IV
COMPARISON WITH EXISTING FPGA WORK ON IRREGULAR MODELS

	[51] GoogLeNet	[50] GoogLeNet	This Work: GoogLeNet	[52] ResNet-152	[53] ResNet-152	This Work: ResNet-152	This Work: DenseNet-161
FPGA	Zynq-7045	Virtex-7 690T	Zynq-7045	Stratix-V GSMD5	Arria-10 GX1150	Zynq-7045	Zynq-7045
Clock (MHz)	250	100	125	150	200	125	125
DSP Util.†	27.77%	-	98.33%	65%	100%	99.55% (98.33%)	90.67%
GOp/s*	116.20	224.00	165.30	226.47	710.30	188.18	155.57
GOp/s/Logic	0.53	0.51	0.75	1.31	1.66	0.86	0.71
GOp/s/DSP†	0.129	0.071	0.184	0.071	0.226	0.209	0.173
GOp/s/W‡	45.91	-	41.32	9.06	-	47.04	38.9
Latency**	27.5 ms	-	22.2 ms	-	31.85 ms	156.40 ms	85.5 ms

* favourable batch size, ** batch size = 1, † 18×18, 19×18 and 25×18 DSP configurations, ‡ power due to benchmark execution with subtracted idle power.

target ConvNets from different aspects, including ConvNet-to-FPGA toolflows [48], [49], [51]–[53], the Escher architecture that optimizes bandwidth utilization [50], the highest performing hand-tuned VGG16 accelerator on Zynq 7045 [23], and the throughput-optimized model-agnostic coprocessor in [59].

Table III lists the performance results for AlexNet and VGG16 that comprise regular computational dataflows. For AlexNet, fpgaConvNet achieves higher throughput compared with DeepBurning[5] [48] by 2.06× and 1.82× on Zynq 7020 and 7045, respectively, and outperforms DNNWEAVER[5] [49] by 1.90× on Zynq 7020. Compared with Escher [50] on Virtex-7, fpgaConvNet achieves 2.58× and 3.12× higher performance density normalized for LUTs and DSPs, respectively (geo. mean across Zynq 7020 and 7045).

With respect to VGG16, the fpgaConvNet 7020 accelerator achieves 1.55× higher throughput than DNNWEAVER [49]. The CNN RTL Compiler targets the Arria-10 GX1150 FPGA on a Nallatech 385 A board. fpgaConvNet reaches 95% (7020) and 74% (7045) of the performance density with respect to DSPs and demonstrates 1.53× (7020) and 1.18× (7045) higher performance density by normalizing with respect to the clock frequency.[6] An important factor to take into account is that [53] runs on a platform with 2.75× more on-chip memory and 3.8× higher off-chip memory bandwidth,[7] which substantially reduce the memory accesses and hence the execution time. Compared with the state-of-the-art hand-tuned VGG16 accelerator [23] on Zynq 7045, the proposed framework generates a design that reaches 83% of the throughput with the advantage of a much lower development time and effort.

Table IV presents the performance results for irregular models. By targeting GoogLeNet, fpgaConvNet demonstrates 1.42× higher throughput than Snowflake [51] and 1.35× higher GOp/s/DSP. Similarly, compared with the Escher GoogLeNet accelerator [50], fpgaConvNet reaches 1.47× and 2.59× higher GOp/s/kLUT and GOp/s/DSP, respectively. For ResNet-152, fpgaConvNet demonstrates 2.94× higher GOp/s/DSP than FP-DNN [52] and 92% of the GOp/s/DSP of the CNN RTL Compiler [53] and 1.23× higher GOp/s/DSP with normalized clock frequency, [6] while in both cases targeting a device with substantially lower on-chip memory capacity and off-chip memory bandwidth.[7] In [59], a coprocessor is proposed that favors the flexible execution of different CNNs over optimizing the hardware to the target CNN with a reported average throughput of 129.7 and 0.046 GOp/s/DSP on Virtex-7 485T. In contrast, by customizing the generated hardware to the target model, the AlexNet and VGG16 designs of fpgaConvNet on Zynq 7045 achieve 4.78× and 3.69× higher GOp/s/DSP than [59]. Overall, the proposed framework demonstrates the improvements in performance over existing FPGA works that have demonstrated the state-of-the-art performance in the presented benchmark ConvNets. Moreover, to the best of our knowledge, this is the first work to have addressed the optimized mapping of DenseNet-161 on custom hardware, presented in the last entry of Table IV.

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

[6]Arria-10 and Zynq 7045 are manufactured at different technologies, 20 and 28 nm, respectively, which affects the maximum operating clock frequency.

[7] The Nallatech 385 A and Zynq 7045 provide a peak bandwidth of 16 and 4.2 GB/s, respectively.

IX. CONCLUSION

This paper presents fpgaConvNet, a framework for the automated mapping of ConvNets on FPGAs. A novel SDF-based methodology is proposed that enables the efficient exploration of the FPGA architectural design space. By casting DSE as MOO, fpgaConvNet is able to effectively target applications with diverse performance needs from high throughput to low latency. Moreover, the proposed framework addresses the mapping of state-of-the-art models with an irregular dataflow by providing support for novel ConvNets that employ Inception, residual, and dense blocks. Quantitative evaluation demonstrates that fpgaConvNet matches and in several cases outperforms the performance density of existing state-of-the-art FPGA designs, delivers higher performance per watt than highly optimized embedded GPU designs, and therefore provides the infrastructure for bridging the gap between deep learning experts and FPGAs.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, Apr. 2015, pp. 1–14.
[2] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *Proc. CVPR*, Jun. 2014, pp. 1701–1708.
[3] S. Yang, D. Maturana, and S. Scherer, "Real-time 3D scene layout from a single image using convolutional neural networks," in *Proc. ICRA*, May 2016, pp. 2183–2189.
[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, Jun. 2016, pp. 770–778.
[6] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.
[7] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. CVPR*, Jun. 2015, pp. 1–9.
[8] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. CVPR*, 2017, pp. 4700–4708.
[9] K. Hazelwood *et al.*, "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *Proc. HPCA*, Feb. 2018, pp. 620–629.
[10] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*, 2017, pp. 1–12.
[11] E. Chung *et al.*, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, Mar./Apr. 2018.
[12] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, Jun. 2018, doi: 10.1145/3186332.
[13] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. FCCM*, May 2016, pp. 40–47.
[14] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," in *Proc. FPL*, Sep. 2017, pp. 1–8.
[15] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A toolflow for mapping diverse convolutional neural networks on embedded FPGAs," in *Proc. NIPS Workshop Mach. Learn. Phone Other Consum. Devices (MLPCD)*, 2017, pp. 1–5.
[16] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.
[17] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. ICANN*, 2014, pp. 281–290.
[18] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," in *Proc. ICLR*, Apr. 2015, pp. 1–14.
[19] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. CVPR*, Jun. 2015, pp. 3431–3440.
[20] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. JPROC-75, no. 9, pp. 1235–1245, Sep. 1987.
[21] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proc. ISCAS*, May/Jun. 2010, pp. 257–260.
[22] A. Dundar, J. Jin, B. Martini, and E. Culurciello, "Embedded streaming deep neural networks accelerator with applications," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 7, pp. 1572–1583, Jul. 2017.
[23] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. FPGA*, 2016, pp. 26–35.
[24] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proc. FPL*, Aug./Sep. 2016, pp. 1–9.
[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ISCA*, Jun. 2016, pp. 367–379.
[26] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ISCA*, Jun. 2016, pp. 1–13.
[27] L. Cavigelli, M. Magno, and L. Benini, "Accelerating real-time embedded scene labeling with convolutional networks," in *Proc. DAC*, Jun. 2015, pp. 1–6.
[28] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. ISCA*, 2013, pp. 24–35.
[29] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. ISCA*, 2010, pp. 247–257.
[30] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. ICCD*, Oct. 2013, pp. 13–19.
[31] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS*, 2014, pp. 269–284.
[32] M. Peemen, R. Shi, S. Lal, B. Juurlink, B. Mesman, and H. Corporaal, "The neuro vector engine: Flexibility to improve convolutional net efficiency for wearable vision," in *Proc. DATE*, Mar. 2016, pp. 1604–1609.
[33] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ISCA*, 2016, pp. 243–254.
[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105.
[35] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. MM*, 2014, pp. 1097–1105.
[36] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A MATLAB-like environment for machine learning," in *Proc. NIPS*, 2011, pp. 1–6.
[37] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, Oct. 2014. [Online]. Available: http://arxiv.org/abs/1410.0759
[38] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *Proc. IJCAI*, 2011, pp. 1237–1242.
[39] M. G. Tallada, "Coarse grain parallelization of deep neural networks," in *Proc. PPoPP*, 2016, Art. no. 1.
[40] J. S. J. Ren and L. Xu, "On vectorization of deep convolutional neural networks for vision tasks," in *Proc. AAAI*, 2015, pp. 1840–1846.
[41] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," in *Proc. ICLR*, Mar. 2014, pp. 1–9.
[42] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," in *Proc. ICLR*, Apr. 2015, pp. 1–17.
[43] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. CVPR*, Jun. 2016, pp. 4013–4021.
[44] S. Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. ICLR*, Feb. 2016, pp. 1–14.
[45] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *Proc. SC*, 2016, Art. no. 54.
[46] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. FPGA*, 2015, pp. 161–170.
[47] N. Suda *et al.*, "Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. FPGA*, 2016, pp. 16–25.
[48] 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 *Proc. DAC*, Jun. 2016, pp. 1–6.
[49] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. MICRO*, Oct. 2016, pp. 1–12.
[50] Y. Shen, M. Ferdman, and P. Milder, "Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer," in *Proc. FCCM*, Apr./May 2017, pp. 93–100.

[51] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *Proc. ISCAS*, May 2017, pp. 1–4.

[52] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. FCCM*, Apr./May 2017, pp. 152–159.

[53] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. FPL*, Sep. 2017, pp. 1–8.

[54] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. FPGA*, 2017, pp. 65–74.

[55] A. Jiménez-Fernández *et al.*, "A binaural neuromorphic auditory sensor for FPGA: A spike signal processing approach," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 4, pp. 804–818, Apr. 2017.

[56] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre, "CaffePresso: An optimized library for deep learning on embedded accelerator-based platforms," in *Proc. CASES*, Oct. 2016, pp. 1–10.

[57] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. ICCAD*, Nov. 2016, pp. 1–8.

[58] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *Proc. ISCA*, 2010, pp. 37–47.

[59] N. Shah, P. Chaudhari, and K. Varghese, "Runtime programmable and memory bandwidth optimized FPGA-based coprocessor for deep convolutional neural network," *IEEE Trans. Neural Netw. Learn. Syst.*, to be published, doi: 10.1109/TNNLS.2018.2815085.

[60] C. R. Reeves, Ed., *Modern Heuristic Techniques for Combinatorial Problems*. New York, NY, USA: Wiley, 1993.

[61] V. A. Shim, K. C. Tan, and H. Tang, "Adaptive memetic computing for evolutionary multiobjective optimization," *IEEE Trans. Cybern.*, vol. 45, no. 4, pp. 610–621, Apr. 2015.

[62] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017.

Stylianos I. Venieris (S'16) received the M.Eng. degree (Hons.) in electrical and electronic engineering from the Imperial College London, London, U.K., in 2014, where he is currently pursuing the Ph.D. degree with the Circuits and Systems Group.

His current research interests include architectures and methodologies for mapping deep learning models on reconfigurable hardware.

Christos-Savvas Bouganis (S'01–M'03–SM'16) is currently a Reader with the Department of Electrical and Electronic Engineering, Imperial College London, London, U.K. He has published over 30 research papers in peer-referred journals and international conferences. He has contributed for three book chapters. His current research interests include the theory and practice of reconfigurable computing and design automation, mainly targeting the digital signal processing algorithms.

Dr. Bouganis is an Editorial Board Member of the *IET Computers & Digital Techniques* and the *Journal of Systems Architecture*. He has served as the Program Chair of the IET FPGA designers' Forum in 2007 and the General Chair of the International Symposium on Applied Reconfigurable Computing in 2008. He is currently serving on the program committees of many international conferences, including the International Conference on Field-Programmable Logic and Applications, the IEEE International Conference on Field-Programmable Technology, the Design, Automation and Test in Europe Conference and Exhibition, the International Conference on Signal Processing, Pattern Recognition and Applications, and the IFIP/IEEE International Conference on Very Large Scale Integration.