fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs

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

Abstract—Since neural networks renaissance, Convolutional Neural Networks (ConvNets) have demonstrated 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, 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 optimised 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 transformation in order to efficiently navigate the architectural design space. By proposing a systematic multi-objective optimisation formulation, the presented framework is able to generate hardware designs that are co-optimised 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 \(6.65 \times\) over highly optimised GPU designs for the same power constraints and achieve up to 2.94\(\times\) higher performance density compared to state-of-the-art FPGA-based ConvNet architectures.

Index Terms—Convolutional neural networks, design space exploration, 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 centre 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.

Apart from high predictive accuracy, ConvNets are also characterised 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 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

S.I. Venieris and C.-S. Bouganis are with the Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, U.K.
e-mail: {stylianos.venieris10,christos-savvas.bouganis}@imperial.ac.uk.
Manuscript received October 26, 2017;

Fig. 1: ConvNet models in the computation-memory space 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 co-optimised 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 to large networks at a lower computational load. This was typically achieved by introducing novel network components that differ from the conventional layer types and which introduce irregular connectivity between layers. The properties of irregularity and non-uniformity make the dataflow of ConvNets more complex compared to 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 optimised 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 multi-core 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 towards customised 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 customisation spectrum, ASIC chips offer ConvNet acceleration with minimal power and area consumption. Nevertheless, they require full-custom manufacturing at high non-recurring 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 optimisations 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 customisability and reconfigurability by means of a set of heterogeneous resources, including look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs) and digital signal processing blocks (DSPs), 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 centre facilities in order to host FPGAs on their servers and target a variety of workloads with specialised hardware [11]. The FPGA mapping of ConvNets could potentially provide tunable trade-offs between critical system parameters, including performance, power and cost, and serve as a useful component in both embedded and data centre 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 recognisers 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 modelling and mapping the inference task of both regular and irregular ConvNets on FPGA-based platforms. By enhancing the work presented in [13] and [14], we propose a Synchronous Dataflow (SDF) framework for the modelling 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 trade-off. The initial work in [13] focuses on generating hardware designs optimised for high-throughput applications, giving an overview of our SDF streaming modelling 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 (1) generalises 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. (2) expands the range of target applications to include applications with both throughput and latency requirements by casting design space exploration as a multiobjective optimisation, (3) provides for the first time an in-depth analysis of the proposed modelling and design space exploration methods and (4) presents extensive comparisons with highly optimised 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, organised 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 [15]. 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:

$$f^{out}_i = \sum_{j=1}^{N_{in}} f^{in}_j \ast k_{i,j} + b_i, \text{ with } i \in [1, N_{out}]$$

where $f^{out}_i$ and $f^{in}_j$ are the $i_{th}$ output and input feature maps respectively, $k_{i,j}$ is the $(K_h \times K_w)$ kernel that corresponds to the $i_{th}$ output and $j_{th}$ input and $b_i$ is the $i_{th}$ bias vector. The $\ast$ operator represents the 2D 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 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 one-dimensional vector. With the feature extractor typically dominating the computational cost of ConvNets [16] and fully-connected layer-based classifiers being abandoned in recent state-of-the-art models [7][17][18][5][8], in this work 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 non-uniform 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 which favoured 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.

1Rectified Linear Unit, defined as $f(x) = \max(0, x)$. 

Recent works have attempted to address the challenges of convolutional layer reconfiguration by designing frameworks that can efficiently explore the architectural design space and the performance-resource trade-off. One such framework is the Synchronous Dataflow (SDF) [13], which provides a high-level, domain-specific language (DSL) for describing the architecture and mapping of ConvNets on reconfigurable hardware. By using SDF, designers can express the architectural design space and the performance-resource trade-off, and the framework automatically generates hardware designs that optimise for high-throughput applications.

In summary, the SDF framework allows designers to:

1. Express the architectural design space and the performance-resource trade-off.
2. Automatically generate hardware designs that optimise for high-throughput applications.

This paper builds upon the initial work in [13] and [14] and proposes a novel approach to modelling and mapping the inference task of both regular and irregular ConvNets on reconfigurable streaming hardware. By using a Synchronous Dataflow (SDF) framework, this work aims to:

1. Generalise the framework to address the emerging challenges of state-of-the-art ConvNets with irregular connectivity.
2. Expand the range of target applications to include applications with both throughput and latency requirements.
3. Analyse the proposed modelling and design space exploration methods.
4. Compare the proposed designs with highly optimised designs on embedded GPUs and with the most recent state-of-the-art FPGA designs.

This approach allows for the efficient exploration of the architectural design space and the performance-resource trade-off. By leveraging the SDF framework, designers can generate hardware designs that are tailored for high-throughput applications and optimised for various critical system parameters such as performance, power, and cost. The proposed methodology provides a principled manner for modelling and mapping ConvNets on reconfigurable streaming hardware, enabling the development of hardware architectures that are tailored for specific applications and can be tuned to meet the needs of performance, power, and cost requirements.
Inception-based Networks: In 2014, Szegedy et al. [7] presented GoogLeNet as a network that achieves 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 ($1 \times 1$, $3 \times 3$, $5 \times 5$) and one $3 \times 3$ max-pooling path. Moreover, to limit the number of weights, $1 \times 1$ filters are applied to reduce the number of channels prior to the $3 \times 3$ and $5 \times 5$ convolutional layers and after the pooling layer. Each path consists of a different computational load and hence offers the opportunity for mapping optimisations.

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

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 parametrised 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 to its preceding layer. The DenseNet with 161 layers (DenseNet-161) 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 makes the mapping of these models a challenging task. To this end, we also target this set of models and address the optimised mapping of their irregular dataflow on FPGAs.

C. Synchronous Dataflow

Synchronous Dataflow (SDF) [19] 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 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 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 optimisations at compile time.

III. RELATED WORK

With deep learning’s recent success, several research groups have focused on the design of customised architectures for ConvNets. Efforts have concentrated on hand-tuned mappings of specific ConvNet models and computation engine optimisations on FPGAs [20][21][22][23] and ASICs [24][25][26][27] and memory subsystem optimisations on FPGAs [28][29] and ASICs [30][31][32]. 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 [33][34][35][36]. From a computational perspective, recent works have explored the parallelisation of convolutions [37], the coarse-grain, batch-level parallelism [38] and the vectorisation of layers [39]. From an algorithmic aspect, recent efforts have employed alternative algorithms to reduce the computational complexity of convolutions [40][41][42]. Several strategies have been explored to address the large memory requirements, including the compression of trained weights [43] and data-layout transformations [44]. Finally, the optimisation of GPU designs for latency-critical embedded applications has also been investigated [26].

Design Space Exploration: Besides the above contributions, significant research effort has been invested into the development of systematic ConvNet-to-FPGA design space exploration (DSE) methodologies. Zhang et al. [45] 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 which is time-shared between layers and can be configured at compile time, with the optimal configuration found by means of enumeration. Suda et al. [46] 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 maximisation of throughput.

Wang et al. [47] 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. Emphasis is given on the identification and implementation of appropriate hardware components for neural networks, without performing systematic DSE as opposed to our work. 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 [48] comprises an automated flow for the generation of high-throughput accelerators for a given ConvNet-FPGA pair. In its backend, DNNWEAVER employs a set of parametrised hardware templates which 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 [49], Escher is proposed as a methodology for increasing the memory bandwidth utilisation of FPGA-based ConvNets, while the authors of [50] designed a programmable accelerator optimised for the high utilisation of its resources. FP-DNN [51] and the CNN RTL Compiler [52] proposed RTL-level optimisations in order to reach high performance and were the first works to target residual networks. Moreover, [53] and [54] presented automated frameworks specifically tailored for FPGA-based binarised and spiking neural networks respectively, while [55] 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 design space exploration with the generation of streaming accelerators that are co-optimised for the ConvNet workload, the target FPGA and the application-level performance needs. In contrast to the implementation-focused Deep-Burning, fpgaConvNet’s key contribution lies on formalising and performing efficiently the design space exploration task by means of the proposed SDF streaming model. By taking into account the application performance requirements, the generated design is optimised for either throughput, latency or multiobjective criteria, as opposed to works [46][48][51][49][56] which aim solely for throughput maximisation. Finally, this work 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 optimised 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 which 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 SDF graph (SDFG). At this point, the nodes of the SDFG correspond to parametrised hardware building blocks and its arcs to interconnections between them. By applying a set of transformations over the SDFG, the optimiser 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.

A. ConvNet Application Model

A ConvNet workload comprises a sequence of layers which is captured in this work by means of a computational directed acyclic graph, with each layer mapped to a node. The target network is supplied by means of a Torch or Caffe description

\[
\begin{align*}
& < K_h, K_w, S_h, S_w, P, N > \\
& \text{where } K_h \text{ and } K_w \text{ are the height and width of each filter, } S_h \text{ and } S_w \text{ the strides that determine the step between successive windows along the feature map’s height and width respectively, } P \text{ the zero-padding of the input feature maps and } N \text{ the number of filters. The rest of the layers follow the same modelling approach, with the nonlinear layer defined in terms of number of nonlinear units and type of nonlinear function to be applied, e.g. sigmoid, tanh or ReLU, and the pooling layer defined in terms of pooling size, stride, number of pooling units and type of pooling operation, e.g. max or average.}
\end{align*}
\]

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 which include digital signal processing blocks (DSPs), look-up tables (LUTs), flip-flops (FFs) and block RAMs (BRAMs). The off-chip memory is typically a DDR SDRAM module which is characterised by its bandwidth and capacity. To formally represent the resources of the target platform, we define a global resource set, \( R \), which is the union of sets \( R_{fpga} \) and \( R_{mem} \).

\[
\begin{align*}
R_{fpga} &= \{ DSP, LUT, FF, BRAM \} \\
R_{mem} &= \{ B_{mem}, C_{mem} \} \\
R &= R_{fpga} \cup R_{mem}
\end{align*}
\]
Moreover, a resource vector, \( rsc_{\text{Avail}} \), is defined which holds the available amount for each of the elements in \( R \):

\[
rsc_{\text{Avail}} = [\text{DSP}_{\text{Avail}}, \text{LUT}_{\text{Avail}}, \text{FF}_{\text{Avail}}, \text{BRAM}_{\text{Avail}}, B_{\text{mem}}, C_{\text{mem}}]^T
\]

with \( B_{\text{mem}} \) and \( C_{\text{mem}} \) are the measured off-chip memory bandwidth and capacity respectively.

V. CONVNETS AS SDF GRAPHS

A key contribution of this work is the representation of hardware design points as SDF graphs. At a hardware level, fpgaConvNet represents design points as SDF graphs 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 SDF graph (SDFG) is formed. The nodes of the SDFG are connected via arcs which 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 which consists of a coarse pipeline of building blocks and corresponds to a design point in the architectural design space.

Fig. 3 illustrates the translation of a convolutional layer to the corresponding SDF hardware graph. In this scenario, a 2D 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 \( N \) \((K \times K)\) convolution units that each performs a dot product between the incoming windows and a \((K \times K)\) kernel of the convolutional layer.

![Convolutional layer as an SDF graph](image)

\[
\Gamma = \begin{bmatrix}
B_{\text{mem}} & -1 & 0 & 0 & 0 \\
0 & (K \times K) & -(K \times K) & 0 & 0 \\
0 & 0 & N(K \times K) & -N(K \times K) & 0 \\
0 & 0 & 0 & N & -B_{\text{mem}}
\end{bmatrix}
\]

The \( \Gamma \) 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:

\[
<\text{param}, s_{\text{in}}, s_{\text{out}}, c_{\text{in}}, c_{\text{out}}, r_{\text{in}}, r_{\text{out}}>
\]

where \( \text{param} \) is a set of block-specific configuration parameters, \( s_{\text{in}} \) and \( s_{\text{out}} \) the number of parallel streams at the input and output of the block respectively, \( c_{\text{in}} \) and \( c_{\text{out}} \) the number of convolution units.
In convolution building block consumes an amount of data at its consumption of data flowing through a sequence of building blocks. Each means of SDF. A ConvNet workload is represented as a stream of data elements from the input feature maps and outputs \( N \) streams 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 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:

\[
< \{ N, K_h, K_w, S_h, S_w, P \}, N, N, 1, K_h \times K_w, 1, \frac{1}{S_w} >
\]

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 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 similarly to convolution banks. In the case of max pooling banks, finding the maximum is performed with a single comparator and \( f \) is equal to \( \frac{1}{(K_h \times K_w)} \) with a single pixel being consumed per cycle. The tuple representation is as follows:

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

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

**B. Modelling ConvNet Workloads**

Complying to 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, \( W_{i}^{(in, out)} \), is the number of feature maps and \( P_{i}^{(in, out)} \) is the number of data elements per feature map at the input or output of the \( i \)th block respectively. To populate the workload along the SDFG of building blocks, we introduce the feature maps matrix, \( F_{map} \), and the data matrix, \( P \), and form the workload matrix, \( W \), as shown below:

\[
W = F_{map} \odot P
\]

As an example, the workload of a convolution between a single \((h \times w)\) feature map and a single \((3 \times 3)\) kernel would be represented in the following graph and matrix forms:

\[
W = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 \\
0 & 100 & 100 & 0 & 0 \\
0 & 0 & 100 & 100 & 0 \\
0 & 0 & 0 & 100 & 100 \\
\end{bmatrix}
\]

Fig. 5: Convolution workload graph and matrix representation where the feature map is assumed to have 100 \((3 \times 3)\) windows.

In this case, the feature maps and data matrices would be populated as follows:

\[
F_{map} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 100 & 100 & 0 & 0 \\
0 & 0 & 100 & 0 & 0 \\
0 & 0 & 0 & 100 & 100 \\
\end{bmatrix}
\]

\[
P = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 9 & 9 & 0 & 0 \\
0 & 0 & 9 & 9 & 0 \\
0 & 0 & 0 & 1 & 1 \\
\end{bmatrix}
\]

**VI. Architectural Design Space**

In our framework, the basic hardware mapping of an SDF graph is a streaming architecture as illustrated 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- and 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 state-of-the-art ConvNet models reaching new records in terms of depth [5], 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 [20]. 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 lets 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 [57] and hence does not fully leverage the parallelism and customisation 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 optimised for the particular subgraph, which can utilise 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 minimised 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 \( p \in \{ 0, 1 \}^{N_L - 1} \) where a value of 1 for the \( i_{th} \) element indicates that the SDFG will be partitioned at the \( i_{th} \) 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 which contains one topology matrix per subgraph:

\[
\Gamma = \langle \Gamma_i \mid i \in [1, N_P] \rangle
\]  

(7)

B. Coarse- and Fine-Grained Folding

In ConvNet execution, high performance is mainly achieved by exploiting two types of parallelism [28]. 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 treatingug 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 \( f_{\text{coarse}} \in \{0, 1\}^{N_L} \) 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 \( N_{\text{out}} \) 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 \( N_{\text{out}} \) feature maps. As a result, the folding factor of the \( i_{th} \) layer \( f_{\text{coarse}} (i) \) lies in the range \( \frac{1}{N_{\text{out}}} \) with 1 for a fully parallel implementation and \( \frac{1}{N_{\text{out}}} \) for a single time-shared unit.

\(^3\)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 $\Gamma$
2: Index $i$ of the layer to be folded
3: Number of output feature maps $N_{out}$ of the layer to be folded
4: Folding factor, $f \in [\frac{1}{N_{out}} - 1]$  

Steps:
1: - Initialise folding vector, $f_{coarse} \in \mathbb{R}^{\#cols \Gamma}$: $f_{coarse} = 1$
2: - $f_{coarse}(i) = f$
3: - Form the folding matrix, $F = diag(f_{coarse})$
4: - Apply the coarse-grained folding, $S' = [S \odot C \odot R]$
5: - Form the folded topology matrix, $\Gamma' = S' \odot C \odot R$

Note: $[\cdot]$ is defined as the element-by-element ceiling operator

Algorithm 2 Fine-grained Folding Transformation

Inputs:
1: Matrix $\Gamma$
2: Kernel size $K$ or pooling size $P$
3: Folding factor, $f \in [\frac{1}{K}, 1]$ or $[\frac{1}{P}, 1]$

Steps:
1: - Initialise folding vector, $f_{fine} \in \mathbb{R}^{\#cols \Gamma}$: $f_{fine} = 1$
2: - $f_{fine}(i) = f$
3: - Form the folding matrix, $F = diag(f_{fine})$
4: - Apply the coarse-grained folding, $R' = R \cdot f$
5: - Apply the fine-grained folding, $T' = S \odot [S' \odot C \odot R']$

Note: $[\cdot]$ is defined as the element-by-element ceiling operator

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 $i$ and the 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. Similarly 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 depicts a typical operation of the weights reloading transformation.

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 amortisation 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 these limitations, we have developed a novel technique for executing latency-critical applications and expanding 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 archit-
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 $f_{in} \in [1, N_{in}]$ 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 $N_{W} = f_{in}$. For a factor of $f_{in}$, each of the $N_{W}$ subgraphs performs $\frac{N_{in}}{f_{in}} N_{out}$ 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 datapath 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 initialised with the deepest subgraph as a starting point and
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 $j_{th}$ subgraph by the $i_{th}$ reference architecture, it is necessary that the columns of $W_{i,j}$ are aligned to the correct columns of $\Gamma_{i,ref}$. This process can be interpreted as forming the correct datapath for the $j_{th}$ 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 $W_{i,j}^{\text{aligned}} \in \mathbb{Z}^{(M \times N)}$ which contains the rows and columns of $W_{i,j}$ 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 $s_{i,j}$ for the $j_{th}$ workload of the $i_{th}$ reference architecture, which can be seen on line 3 of the inputs list. The algorithm starts with the initialisation of $W_{i,j}^{\text{aligned}}$ with a zero-padded version of $W_{i,j}$ to match the size of $\Gamma_{i,ref}$. 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 to 8 shift the current column to the right, i.e. along the coarse pipeline of building blocks in the reference architecture. Next, lines 9 to 14 down-shift the column in order to align the pipeline of building blocks in the reference architecture. Next, the current column is shifted to the right, i.e. along the coarse pipeline of building blocks in the reference architecture. Finally, the current column is shifted to the right, i.e. along the coarse pipeline of building blocks in the reference architecture.

\begin{algorithm}
\caption{Workload Alignment for Weights Reloading}
\begin{algorithmic}
\Require
1: Dimensions $(M \times N)$ of topology matrix $\Gamma_{i,ref}$
2: Workload matrix $W_{i,j} \in \mathbb{R}^{K \times L}$
3: Shift vector $s_{i,j} \in \mathbb{Z}^{L}$ with the alignment shifts for each column
4: Identity matrices $I_{N \times N}$ and $I_{M \times M}$
5: Lower shift matrices $S_{N \times N}$ and $S_{M \times M}$
\Ensure
Steps:
1: $W_{i,j}^{\text{aligned}} = \begin{bmatrix} W_{i,j} & 0_{M \times (N-L)} \end{bmatrix}$
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 $A' \in \mathbb{R}^{N \times M}$
5: $A'' = I_{1}^{M-1} \circ S_{M}^{1} \circ \ldots \circ S_{M}^{M-1} \cdot I_{col}^{k}$
6: - Update the overall left alignment matrix -
7: $A_{i,j} = A' \cdot A'' \cdot \ldots \cdot A'^{T}$
8: $W_{i,j}^{\text{aligned}} = \begin{bmatrix} W_{i,j} & 0_{M \times (N-L)} \end{bmatrix}$
9: - - - Align the interconnections (down shift) - - -
10: - Form left alignment matrix $A' \in \mathbb{R}^{N \times M}$
11: $A' = I_{1}^{col-1} \circ S_{col}^{1} \circ \ldots \circ S_{col}^{col-1} \cdot I_{col}^{k}$
12: - Update the overall left alignment matrix -
13: $A_{i,j} = A' \cdot A'' \cdot \ldots \cdot A'^{T}$
14: $W_{i,j}^{\text{aligned}} = \begin{bmatrix} W_{i,j} & 0_{M \times (N-L)} \end{bmatrix}$
15: end for
\end{algorithmic}
\end{algorithm}

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

**Inception Block.** The Inception block (Fig. 11a) is parametrised with respect to the coarse-grained folding of each convolutional and pooling layer and the fine-grained folding of the $3 \times 3$ and $5 \times 5$ convolutional layers, leading to the combined parameter vector $f_{\text{inception}} \in (0,1)^{3}$. 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.

**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 parametrise it with respect to its coarse-grained folding. Fig. 11b shows the overall residual block. Its input consists of an optional $3 \times 3$ layer between them. The configuration of the residual block is captured with a vector $f_{\text{residual}} \in (0,1)^{5}$ with one fine-grained and four coarse-grained folding factors.

**Dense Block.** In DenseNet, a dense block consists of a series of $1 \times 1$ followed by $3 \times 3$ 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 $3 \times 3$ 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, 3×3) receives \( N_{th} + (i-1)k \) input feature maps with \( i \in [1, r] \). To exploit this property, we define a DenseNet-specific optimisation and extend the dense block with an additional parameter that unrolls the 1×1 layer with respect to its input feature maps (Fig. 11c). This approach allows us to sustain high utilisation 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 \( f_{dense} \in \{0, 1\}^4 \) which includes one fine-, 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 parametrisation 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 optimisation problem with the objective to optimise the performance metric of interest. The design space is traversed by means of the SDF transformations until a design point is obtained that optimises the target objective.

A. Performance Model

Given the topology matrix \( \Gamma \) of a design point and the workload matrix \( W \) of a ConvNet, the initiation interval matrix \( \Pi \) is formed as follows:

\[
\Pi = W \odot \Gamma
\]

(11)

where \( \odot \) denotes the Hadamard elementwise division. Each element of \( \Pi \) 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 \( \Pi \), denoted by \( \Pi^{max} \).

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

\[
t(B, \Gamma, W) = \frac{1}{\text{clock rate}} \cdot (D + \Pi^{max} \cdot (B - 1))
\]

(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 \( \Gamma_i \) and \( W_i \) matrices. Furthermore, the weights reloading transformation partitions \( W_i \) into \( N_{Wi} \), workload subgraphs, indexed by \( j \). Each of the \( N_{Wi} \) subgraphs will be scheduled for execution on a single derived architecture represented by \( \Gamma_{i,ref} \). To support design points that employ these two transformations, each design point is expressed by a topology tuple \( \Gamma \) as defined in Eq. (7) and similarly the workload matrix \( W \) is replaced by a tuple defined as follows:

\[
W = \langle W_{ij} \mid i \in [1, N_p], j \in [1, N_{Wi}] \rangle
\]

(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 \( t_{i,j,weights} \). 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, \( t_{i,reconfig} \), has to be included. With this formulation, the overall execution time is expressed as:

\[
t_{total}(B, \Gamma, W) = \sum_{i=1}^{N_p} \sum_{j=1}^{N_{Wi}} t_{i,j}(B, \Gamma_{i,ref}, W_{ij}) + \sum_{i=1}^{N_p} t_{i,weights} + \sum_{i=1}^{N_p-1} t_{i,reconfig}
\]

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 amortised. 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 which requires a total of \( W_{ConvNet} \) GOps/input, the throughput and latency of a design point can be estimated as in Eq. (14) and Eq. (15) in GOps and seconds respectively.

\[
T(B, \Gamma, W) = \frac{W_{ConvNet}}{t_{total}(B, \Gamma, W)}
\]

(14)

\[
L(B = 1, \Gamma, W) = t_{total}(1, \Gamma, W)
\]

(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 utilisation 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. Optimisation Framework

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

\[
\begin{align*}
\max \Gamma & \quad T(B, \Gamma, W), \quad \text{s.t.} \quad rsc(B, \Gamma) \leq rsc_{\text{Avail}}. \quad (16) \\
\min \Gamma & \quad L(1, \Gamma, W), \quad \text{s.t.} \quad rsc(1, \Gamma) \leq rsc_{\text{Avail}}. \quad (17) \\
\max \Gamma & \quad T(B, \Gamma, W), \quad \text{s.t.} \quad rsc(B, \Gamma) \leq rsc_{\text{Avail}}. \quad (18)
\end{align*}
\]

where \( T, L, rsc \) and \( \epsilon \) 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 \( rsc_{\text{Avail}} \) is the resource vector of the target platform. The objective function aims to either (1) maximise throughput (Eq. (16)), (2) minimise latency (Eq. (17)) or (3) perform a multiobjective optimisation which maximises throughput with a latency constraint (Eq. (18)).

Given an input ConvNet, the optimisation problems are defined over the set of all design points \( S \) in the design space presented in Section VI, and the objective functions \( T : S \rightarrow \mathbb{R}^+ \) and \( L : S \rightarrow \mathbb{R}^+ \) 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 below:

\[
2^{N_L - 1} \cdot 2^{N_{\text{conv}} - 1} \cdot \prod_{i=1}^{N_{\text{conv}}} N_{\text{reld},i} \prod_{i=1}^{N_L} N_{\text{coarse},i} \prod_{i=1}^{N_L} N_{\text{fine},i}
\]

where \( N_L \) is the number of layers, \( N_{\text{conv}} \) is the number of convolutional layers, \( N_{\text{reld},i} \) is the number of possible input feature maps folding factors for the \( i \)-th 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 \( i \)-th 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 non-convex space.

In this work, simulated annealing \cite{69} has been selected as the basis of the developed optimiser. The SDF transformations, defined in Section VI, are formalised as a set of operations \( \Sigma \) and the neighbourhood \( N(s, \sigma) \) 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 optimiser traverses the design space by considering all the described transformations and converges to a solution of the objective function, selected from Eq. (16)-(18).

The multiobjective optimisation (MOO) problem of Eq. (18) involves a reduction to a single objective by means of an \( \epsilon \)-constraint formulation. This approach incorporates the application-specific importance of throughput and latency prior to optimisation and is solved by the developed simulated annealing optimiser. Alternatively, other methods, such as genetic algorithms, can be employed to solve the MOO problem \cite{60}. Such an optimisation 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.

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 synthesisable Vivado HLS code. All hardware designs were synthesised 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 optimisations 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, Q.8.8 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 \cite{22} has shown Q.8.8 to give similar results to 32-bit floating-point.

**Table II: Benchmarks**

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Conv Layers</th>
<th>Workload</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet [33]</td>
<td>5</td>
<td>1.3315 GOps</td>
<td>2.3 M</td>
</tr>
<tr>
<td>VGG16 [1]</td>
<td>13</td>
<td>30.7200 GOps</td>
<td>14.7 M</td>
</tr>
<tr>
<td>GowLeNet [7]</td>
<td>22</td>
<td>3.1458 GOps</td>
<td>5.8 M</td>
</tr>
<tr>
<td>ResNet-152 [5]</td>
<td>152</td>
<td>23.0323 GOps</td>
<td>55.2 M</td>
</tr>
<tr>
<td>DenseNet-161 [8]</td>
<td>161</td>
<td>13.7590 GOps</td>
<td>24.1 M</td>
</tr>
</tbody>
</table>
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 [61]. 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 optimised 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 optimised 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.

**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 utilises the throughput-driven objective function (Eq. (16)) during design space exploration. On all evaluated platforms, each benchmark is run with a favourable batch size in order to reach peak throughput.

\(^4\)Idle Power: Tegra X1 (5W), ZedBoard (5W), ZC706 (7W).

**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 (Eq. (17)) during design space exploration in order to generate low-latency accelerators. On all evaluated platforms, the benchmarks are run with a batch size of 1.

**Discussion.** Tegra X1 mounts a 256-core GPU with native support for FP16 half-precision floating-point arithmetic and 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 \times \text{geo. mean} across the benchmarks. For latency-driven scenarios, fpgaConvNet demonstrates a throughput improvement of up to 6.65 \times \text{geo. mean}.

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 \times (1.12 \times \text{geo. mean}) improvement in GOp/s/W over Tegra X1 for throughput-driven applications and 1.70 \times (1.36 \times \text{geo. mean}) for latency-driven. Fig. 12 and 13 show the measured power-performance space on Tegra X1 with different frequency configurations (76.8, 537.6, 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 optimised 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 target ConvNets from different aspects, including ConvNet-to-FPGA toolflows [47][48][51][52][50], the Escher architecture
that optimises bandwidth utilisation [49], the highest performing
hand-tuned VGG16 accelerator on Zynq 7045 [22] and the
throughput-optimised model-agnostic coprocessor of [58].

Table III lists the performance results for AlexNet and
VGG16 that comprise regular computational dataflows. For
AlexNet, fpgaConvNet achieves higher throughput compared
to DeepBurning\textsuperscript{5} by 2.06× and 1.82× on Zynq 7020
and 7045 respectively and outperforms DNNWEAVER\textsuperscript{6} by
1.90× on Zynq 7020. Compared to Escher [49] on Virtex-7,
fpgaConvNet achieves 2.58× and 3.12× higher performance
density normalised 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 [48].
The CNN RTL Compiler targets the Arria-10 GX1150 FPGA
on a Nallatech 385A 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 normalising with respect to the clock
frequency.\textsuperscript{6} An important factor to take into account is that
[52] runs on a platform with 2.75× more on-chip memory and
3.8× higher off-chip memory bandwidth\textsuperscript{7}, which substantially
reduce the memory accesses and hence the execution time.
Compared to the state-of-the-art hand-tuned VGG16 accelerator [22] 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 [50] and 1.35× higher
GOP/s/DSP. Similarly, compared to the Escher GoogLeNet accelerator [49], fpgaConvNet reaches 1.47× and 2.59× higher
GOP/s/Logic and GOP/s/DSP respectively. For ResNet-152, fpgaConvNet demonstrates 2.94× higher
GOP/s/DSP than FP-DNN [51] and 92% of the GOP/s/DSP of
the CNN RTL Compiler [52] and 1.23× higher GOP/s/DSP
with normalised clock frequency\textsuperscript{8}, while in both cases targeting
a device with substantially lower on-chip memory capacity
and off-chip memory bandwidth\textsuperscript{7}. In [58], a coprocessor is
proposed that favours the flexible execution of different CNNs
over optimising the hardware to the target CNN, with reported
average throughput of 129.7 GOP/s and 0.406 GOP/s/DSP
on Virtex-7 485T. In contrast, by customising 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 [58]. Overall, the proposed
framework demonstrates improvements in performance over
existing FPGA works that have demonstrated 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 optimised mapping of DenseNet-161 on custom
hardware, presented in the last entry of Table IV.

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 design
space exploration as multiobjective optimisation, fpgaConvNet
is able to effectively target applications with diverse perfor-
ance needs, from high throughput to low latency. Moreover,
the proposed framework addresses the mapping of state-of-
the-art models with 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, delivering
higher performance-per-Watt than highly optimised embedded
GPU designs, and provides the infrastructure for bridging the
gap between deep learning experts and FPGAs.

REFERENCES

for Large-Scale Image Recognition,” ICLR, 2015.