Hardware Compilation of Deep Neural Networks: An Overview

Ruizhe Zhao*, Shuanglong Liu*, Ho-Cheung Ng*, Erwei Wang*, James J. Davis*, Xinyu Niu†,
Xiwei Wang‡, Huifeng Shi§, George A. Constantinides*, Peter Y. K. Cheung* and Wayne Luk*
*Imperial College London, London, United Kingdom
{*ruizhe.zhao15, s.liu13, h.ng16, erwei.wang13, james.davis, g.constantinides, p.cheung, w.luk*}*@imperial.ac.uk*
†Corerain Technologies Ltd., Shenzhen, China, *xinyu.niu@corerain.com*
‡China Academy of Space Technology, Beijing, China
§State Key Laboratory of Space-Ground Integrated Information Technology (SGIIT), Beijing, China

***Abstract*—Deploying a deep neural network model on a reconfigurable platform, such as an FPGA, is challenging due to the enormous design spaces of both network models and hardware design. A neural network model has various layer types, connection patterns and data representations, and the corresponding implementation can be customised with different architectural and modular parameters. Rather than manually exploring this design space, it is more effective to automate optimisation throughout an end-to-end compilation process. This paper provides an overview of recent literature proposing novel approaches to achieve this aim. We organise materials to mirror a typical compilation flow: front end, platform-independent optimisation and back end. Design templates for neural network accelerators are studied with a specific focus on their derivation methodologies. We also review previous work on network compilation and optimisation for other hardware platforms to gain inspiration regarding FPGA implementation. Finally, we propose some future directions for related research.**

I. Introduction

Deep neural networks (DNNs) represent one of the most effective classes of machine learning techniques, and a range of DNNs have been applied in application domains including image classification, speech recognition and reinforcement learning. In spite of their excellent accuracies, DNNs can demand a substantial amount of hardware to meet their computational needs. For example, VGG-16's coefficients consume $528\,\mathrm{MB}$ of memory, while 30.6 billion arithmetic operations are required for a single inferencing pass on ImageNet [1]. Worse still, it can take several days or even weeks of training until a DNN reaches acceptable accuracy for tasks with increasing complexity. These factors hinder the wide deployment of DNNs in real-life scenarios with limited resources and strict requirements on power consumption, latency, *etc.*

Very often, classification with DNNs relies on mainstream hardware platforms such as CPUs or GPUs, which lack the flexibility to satisfy all user constraints. FPGAs, on the other hand, are promising candidates for DNN implementation due to their configurability and energy efficiency. It is therefore appealing to combine the effectiveness of DNNs with the customisability of FPGA platforms to facilitate the proliferation of machine learning across application domains.

Given a specific task targetting a particular reconfigurable device, the combined search space of possible DNN topological choices, model parameters and hardware design possibilities is colossal. A framework that can perform automatic design space exploration and discover near-optimal solutions is therefore highly desirable. Within such a framework, a DNN is considered to be a program and a compiler transforms the network from its original representation to a hardware implementation under certain constraints. Compilers tend to be split into two major components: front ends and back ends. A front end parses a DNN representation, in which the network topology and parameters mainly depend upon the training framework. The DNN is platform-independently optimised and an intermediate representation (IR) is generated. The IR is subsequently accepted by a back end for platform-dependent optimisation, leading to the generation of a bistream and binary for hardware implementation and software control.

The implementation of such a compiler is challenging due to the complexity of the design space exploration process. As mentioned, the design space includes numerous combinations and optimal ones can only be determined when the designs are implemented on actual hardware, which involves a lengthy synthesis, placement and routing process [2]. This paper summarises recent literature by presenting an overview of the DNN compiler landscape including front ends (Section II) and back ends (III). As a practical approach to the generation of implementations in the back end, hardware design templates will be discussed in Section IV.

II. Compiler Front Ends and Models

A compiler front end is responsible for transforming a high-level DNN representation from a source representation to an IR. Platform-independent optimisation can be carried out during this process. In this section, we cover high-level representations and platform-independent optimisation techniques proposed and exploited within DNN frameworks.

The support of the United Kingdom EPSRC (grant numbers EP/I012036/1, EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/K034448/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653), Corerain, Intel, Maxeler, SGIIT, China Scholarship Council and the Lee Family Scholarship is gratefully acknowledged.

A. DNN Model Representation

Researchers have built models from optimised machine learning frameworks in recent years [3]–[8]. These frameworks use simple input constructs and provide easy-to-use mechanisms to build, train and test DNN models. Thereafter, for deployment, one needs to export DNN structures and coefficients in a particular representation. Such model representations are the entry points of a DNN compiler.

A model representation can be recognised as a domain-specific language (DSL). In general, there are two implementation paradigms for DNN DSLs: *coarse-grained layer-based approaches* and *fine-grained graph-based approaches*.

A layer-based DSL contains primitives that define typical DNN layers. The major advantage of this approach is that it is straightforward to optimise the implementation of a specific DNN layer. However, difficulties may arise when implementing new layer types or attempting to achieve cross-layer optimisation. Examples of layer-based DSLs include Caffe [8], Mocha [9] and Darknet [10].

A graph-based DSL can be used to construct a general computation graph in which nodes are low-level arithmetic operators, such as addition or multiplication, or high-level DNN operators, *e.g.* convolution or rectified linear units (ReLUs). These nodes are normally organised from a dataflow perspective. A range of graph algorithms can be used to execute and optimise a DNN model specified by a graph-based DSL. This provides flexibility since it is possible to build a DNN with high-level operators—similar to a layer-based approach— or with low-level operators to experiment with new DNN functions. It is harder, however, to understand the underlying representation of a computation graph due to the lack of abstraction. Graph-based DSLs include TensorFlow [3], MXNet [7], Torch [5], CNTK [4] and Theano [6].

Existing works on the hardware compilation of DNNs mostly employ layer-based DSLs to generate hardware accelerators since they facilitate intuitive customisation of hardware templates. For example, Venieris *et al.* [11] and Zhang *et al.* [12] used the DSL from Caffe as the input to their frameworks. These DSLs are then converted to IRs, which convey information about DNN models and provide hints for hardware realisation. The primary objective of such an IR is to enhance the portability of input DSLs. It is worth noting that ONNX [13], a community-supported DSL, provides exchangeability of model representations between many popular machine learning frameworks.

B. DNN Model Optimisation

The primary optimisation goal of a DNN model is to achieve efficient inferencing. In other words, an efficient DNN model obtains high classification accuracy with few computational resources. Model efficiency can be coarsely defined as model accuracy normalised by model size, where the latter is stated in terms of the number of operations performed and memory footprint required during execution.

A DNN model can be optimised by reducing redundancy. DNN models are frequently trained using large datasets in order to achieve generality. However, such a model may be overly complicated for a simple application. On one hand, redundancy lies in model coefficients. Some coefficients, if pretrained on a large-scale dataset, may become insignificant for a specific application. These can be *pruned* or *quantised* to reduce demands on computation and storage [14]–[17]. On the other hand, redundancy can also be architectural, *i.e.* the realisation of a model may be overly complicated for a particular task. There are some pragmatic approaches to reduce architectural redundancy. Kernels of reduced size have been empirically proven to be more efficient than larger ones in typical image classification tasks [18], [19]. ResNet's authors demonstrated that a deeper DNN with small, regular filters is more efficient when shortcut connections are inserted [20]. Grouped, depthwise and pointwise convolutional layers have been shown to facilitate the reduction of computing costs while maintaining accuracy [21]–[24].

Currently, such optimisation techniques tend to be manually implemented, requiring expert knowledge. DNN researchers are now exploring the use of automatic architectural search engines to generate efficient DNN models instead [25], [26]. In particular, Zhao *et al.* recently proposed a transfer learning-based approach to systematically identify and replace redundant layers in pretrained DNN models, allowing the production of efficient networks for particular applications [27].

III. Compiler Back Ends

The purpose of a compiler back end is to produce hardware implementations and control software for a given FPGA from a platform-independently optimised DNN. Inputted as an IR, DNNs at this level contain architectural information and additional optimisation details.

It is challenging to convert an arbitrary software representation to an optimised implementation due to the complexities of both DNN models and target devices. Researchers therefore tend to generate hardware from predefined design templates. Templates of different types have been proposed to achieve particular objectives: some works accelerate matrix multiplication operations to maintain generality, while others narrow down templates to specific categories of DNN in order to produce more optimal, yet less flexible, designs.

A back end also generates control software, which must be tightly coupled with the hardware to ensure correctness and efficiency when executed. The generation of such software is relatively simple for CPUs, mainly due to their implicit latency-hiding mechanisms, and there exists a mature compiler ecosystem for CPU programming, in particular the LLVM tool chain, which can be used for generating programs for efficient DNN processing. The equivalent procedure for customised hardware is nontrivial, however, due to the extra requirements of explicit scheduling and memory management to hide latency, and the lack of standardised compiler tool chains.

In the literature, TVM is an end-to-end framework that compiles DNN models to various hardware platforms [28]. This work has several important features. The proposed fusion of operators can greatly improve performance by reducing

inter-operator data transfer. The exploration of low-level tensor operations' schedule spaces is also performed by using common passes from Halide: a DSL for the automatic optimisation of high-performance image processing tasks [29]. Note that Halide features similar processing routines to DNNs. DLVM provides a DSL that can be used to specify forward and backward computations based on tensors [30]. Optimisation is straightforward in the sense that DLVM DSL is converted to LLVM IR, allowing LLVM optimisation passes to be used. Tensor Comprehensive provides a framework to compile DNN models onto CPU and GPU platforms [31]. The authors also make use of Halide, along with multiple polyhedral optimisation passes. The polyhedral framework is commonplace in hardware design templates as well, as will be discussed in Section IV-A. Finally, Latte's authors proposed another DSL to represent DNN models, presenting a compiler framework that applies parallelisation and operator fusion [32].

IV. Hardware Design Templates

A hardware template is a generic implementation with configurable parameters. Templates can be described in a hardware description language (HDL) or as conceptual diagrams. With the use of templates, the hardware generation process can be automated, and due to the presence of configurable parameters, the templates themselves can generalise onto multiple hardwares. These properties of hardware templates eliminate the need to compile arbitrary software representations to optimised hardware directly: an active area of research within the FPGA community [33]–[35].

The generation of a template-based design usually consists of two major steps: *parameter value selection* and *template instantiation*. The former is an optimisation problem in which the objective depends on the requirements of the generated design, such as its latency and throughput requirements or restrictions on resource usage. A corresponding objective function therefore contains template parameters that capture such design metrics. The solution of such an optimisation problem can be regarded as a *design space exploration* (DSE). The process of template instantiation sees the propagation of parameter values into a template in order to generate an implementation. This procedure is accomplished through the use of module or template parameter passing in the languages that describe the template.

There is no *de facto* metric for the quality of a hardware design template for DNN acceleration. With the objective of generating DNN hardware automatically and ensuring templates generalise across platforms, we consider effective hardware templates to feature the following properties.

- *Scalability*: A hardware template is scalable if it can be configured to generate efficient implementations in scenarios of different magnitude, *e.g.* large-scale datacentres consisting of multiple high-end FPGAs *vs* edge devices with constrained resources and energy budgets. A scalable template can also generate efficient designs in accordance with changing performance objectives. For example, if a compiler's optimisation goal were modified from high performance to low power consumption, a scalable template would be configured to use fewer parallel processing units and/or lower clock frequency in order to accommodate.
- *Flexibility*: A DNN model may contain layers of many types, and each layer can have different configuration parameters. A template is flexible if it is able to produce implementations that support a range of layer types and parameters, *e.g.* kernel size.

Design templates are mentioned in many of the recent works related to DNN accelerators. In some literature, the authors aimed to design templates with high scalability and flexibility [12], [36]–[40], while others applied them within design space explorations to maximise performance [41]–[43]. We categorise DNN design templates by considering the following aspects.

- *Architecture*: Templates can have different base architectures: systolic arrays, streaming engines, *etc.*
- *Derivation*: A templated design can be derived in various ways. It is common to analyse nested loops within DNN computations, deriving template configurations that map them to design components. Polyhedral-based methods are sometimes also applied.
- *Model*: It is often necessary to predict the performance of a design to be generated by a given template. A design model can map configurations to expected design properties, which can further be used to inform design space exploration. A roofline model is a simple, commonly used example [44].
- *Configurability*: A template's configurability defines how and to what extent it can be instantiated. We evaluate this factor by considering the parameters and functionality that a given template provides.

In the remainder of this section, we first introduce templates categorised by their derivation methods. Templates derived from loop analysis-based methods are discussed in the first instance due to the generality of these methods regarding DNN computation. Methods based on dataflow graph analysis are introduced next, which are more general but less common for DNN accelerator designs. Inspired by advances made in the acceleration of DNNs on CPUs and GPUs, several design templates in which matrix multiplication is considered to be the core operation, and so is accelerated, are discussed. Thereafter, templates with more unusual base architectures, such as systolic arrays, are considered. We also mention several works that do not involve templates *per se*, but nevertheless provide insights for template design from their novel architectures. The section concludes with comparisons between these templates.

A. Templates from Loop Analysis

As with many other compute-intensive tasks, DNN processing consists of nested loops, particularly in the convolutional layers. To accelerate the computation of nested loops on an FPGA, *loop unrolling* and *pipelining* are the most commonly used techniques. Loop unrolling increases the number of

performed operations per loop iteration from 1 to N, known as a *step size* or *unrolling factor*, and reduces the total number of loop iterations. *Loop pipelining*, meanwhile, schedules operations within loops so that successive iterations can begin execution as quickly as possible.

However, it is not always possible to fully apply both of these techniques due to data dependencies across loop iterations. For example, a loop cannot be fully pipelined if the computation of one iteration depends on the results from the previous iteration. Also, the limited memory bandwidth and non-sequential access pattern can degrade the performance [45]. Since the size and number of ports of a memory block are finite, operations involve memory access must be carefully designed to improve data locality and reduce unnecessary transactions. Finally, the number of resources required to implement optimised loops may exceed the capacity of a given platform. Loop tiling is needed to split the original workload into smaller ones for hardware execution. Note that resolving these issues often necessitate certain loop reordering, and the following sub-sections summarise the common loop analysis technique to derive a DNN accelerator.

1) Parallelism Analysis: An intuitive way to analyse and explore nested loops in hardware terms is through parallelism. Chakradhar *et al.* presented a configurable architecture that exploits three types of parallelism: *operator level*, to parallelise operations within single convolution, and *intra-* and *inter-output*, to enable parallel processing for multiple channels of input and output images, respectively [46]. The level of parallelism was decided by balancing bandwidth requirements and execution time. Chakradhar *et al.*'s paper was particularly significant since it explored more opportunities for parallelism than existing works, which only exploited parallelism at the operator level through systolic arrays [47]–[49]. The work also featured a novel design space exploration that took both memory bandwidth and throughput into account, previously only covered by Cadambi *et al.* [48].

Some recent works proposed parallelised design templates similar to the ones from aforementioned approaches in the era when DNN became widely deployed. Template should enable the acceleration of not only convolutional layers, but also others including fully connected, pooling and activation. Ma *et al.* [36] presented a scalable HDL template that can be used for AlexNet [50] and NiN [51] generation. Rahman *et al.* jointly considered unrolling and tiling factors together with a design space exploration process [43]. Motamedi *et al.* [42] proposed a template similar to Zhang *et al.*'s [41], also using roofline modelling. Motamedi *et al.*, however, did not mention polyhedral analysis, instead deriving their template parameters from nested loops and parallelism types, as were also covered by Chakradhar *et al.* [46]. Ma *et al.* categorised the parallelism of convolution layer by different loop unrolling strategies, and proposed a template that can be systematically configured based on the impact of performing loop transformations on various loop levels [37]. Another recent work of Ma *et al.*'s focused more upon the flexibility of accelerator templates to support a diverse range of DNNs [38]. Peemen *et al.* also built template designs from nested loops in convolutional layers, while they optimised their design with a special focus on memory usage issues, such as resource and reuse distance [52]. Zhang *et al.* explicitly analysed on-chip RAM usage and the balance of memory bandwidth and performance, proposing a template capable of achieving high performance [53]. Wei *et al.* proposed the use of systolic arrays to achieve higher levels of parallelism, since such architectures are place-and-route friendly and allow the achievement of high clock frequencies [54].

Fusing multiple layers into a single, unified operator can improve performance, as time-consuming off-chip data transfer can be saved between layers. Manoj *et al.* studied so-called layer fusion between consecutive convolutional layers [55], while Xiao *et al.* attempted to fuse layers in Winograd-based design templates (which will be elaborated upon in Section IV-B) [56]. Zhao *et al.* considered layer fusion opportunities in convolutional blocks [27], each consisting of various types of convolutional layers such as the bottleneck block [20], which features a standard convolutional layer sandwiched between a pair of pointwise ones.

2) Polyhedral Analysis: Alternatively, polyhedral model-based methods can provide effective and systematic approaches for deriving templated designs from nested loops. The polyhedral model is a representation of a program's statically predictable control flow, which could mainly consist of nested loops. A dynamic instance of a statement, *e.g.* an iteration of a loop statement, is represented as an integer vector on an affine hyperplane. If the loop bounds are linear combinations of variables, the statement defines bounded polyhedron via its iteration vectors. Methods based on the polyhedral model mainly perform *loop transformation* to increase parallelism and data locality.

Polyhedral methods have been applied to reconfigurable computing in recent years. Pouchet *et al.* studied the application of polyhedral loop transformation on general tasks for FPGAs [57]. Their target was to minimise off-chip data transfers while constraining on-chip buffer sizes. Zuo *et al.* followed a similar approach but focused more on the integration of polyhedral analysis into the general flow of high-level synthesis (HLS) [58]. The authors' motivation is based on the fact that loops that HLS cannot directly pipeline, mainly due to data dependency, can often be transformed and then pipelined.

The application of polyhedral methods to DNN accelerator design is a relatively new field of study. Zhang *et al.* proposed the application of loop unrolling and interchanging within DNN architectures [41]. Based on notations from [57], they concluded that, for a given multidimensional array, there are three types of data-sharing relationship between different iterations of a loop dimension: *irrelevant*, *independent* and *dependent*. In loop unrolling where the processing elements (PEs) and buffers are duplicated, different data-sharing relationships can lead to different implementations. The loops unrolled are determined by establishing dimensions which do not have dependent relationships with any buffers in the design. This avoids the creation of complex arbitration logic

between buffers and PEs. Zhang *et al.*'s target was therefore to reduce the *initial interval* of pipelines. The rest of their work mainly explored the design space of accelerators optimised via their polyhedral-based method through the roofline model. The work, however, did not fully apply automatic optimisation tools powered by polyhedral analysis. Instead, they partially applied the idea of polyhedral analysis and manually transformed the nested loops of convolution, leading to an insufficiently explored design space.

Additionally, the template proposed by Motamedi *et al.* introduced more dimensions into the design space and achieved better optimisation results, despite not making use of polyhedral analysis [42]. Caffine reused the design proposed by Zhang *et al.* and involved mapping from the computation of fully-connected layers to PEs of convolutional layers [12].

In other implementations, the authors applied polyhedral analysis for DNN acceleration on non-FPGA platforms. Yang *et al.* used polyhedral analysis to create an optimised loop-blocking strategy for convolutional layers on multicore CPUs, demonstrating higher performance than conventional general matrix-matrix multiplication (GEMM)-based approaches [59].

B. Templates from Linear Algebra Acceleration

On generic software platforms, it is common to use optimised linear algebra libraries to achieve DNN acceleration. Most of the popular DNN frameworks use at least one implementation of the *Basic Linear Algebra Subprograms* (BLAS) [60]. Out of all the BLAS procedures, GEMM is the most commonly used function for DNN implementation, particularly in convolutions, fully connected layers and long short-term memories (LSTMs) [61].

1) Matrix Multiplication: Inspired by this phenomenon, the authors of several papers derived their design templates from the perspective of a GEMM accelerator. The advantage of targetting the template to matrix multiplication acceleration is simplicity: it is relatively easy to optimise this straightforward operation to make the best use of the available resources and to achieve high performance. The main drawback of this approach is that it can introduce overhead, especially within convolutional layers, within which data need to be rearranged into *Toeplitz matrices* prior to computation [62]–[64]. For this reason, throughput-oriented DNN applications may benefit from matrix multiplication optimisation, while latency-focussed ones may not. Gupta *et al.* presented a systolic array-based GEMM processor to perform DNN computations at low precision [65]. Although their work involved no templates or design space explorations, the elegance of their design is inspiring. Suda *et al.* proposed an end-to-end framework for mapping DNNs onto matrix multiplication-based design templates [66]. N_{conv} and S_{conv}, the two design template parameters in their work, only related to the number of threads and vectorisation factor in the accelerator; no convolutional layer parameters were involved in their design explorations. Suda *et al.*'s work achieved comparable performance to Zhang *et al.*'s [41] when discounting the effect of fixed-point optimisation. FP-DNN also projects DNN models onto a matrix multiplication-based FPGA template [67]. Its authors achieved much faster inferencing than Suda *et al.*, who applied existing designs directly. Moss *et al.* studied this approach on the Intel HARPv2 platform and captured performance data for varying precisions, including binarised representations [68].

2) Winograd and the Fast Fourier Transform: Among other linear algebra-based methods exploited to accelerate DNNs, the *Winograd minimal filtering algorithm* and *fast Fourier transform* (FFT) are two effective methods that have been widely applied. Research efforts thus far have focussed on the performance enhancement of convolutional layers, since convolution consumes more than 90% of the total processing time for the majority of DNNs [41]. The methods have a similar form: both require the use of a pair of transformation matrices—one to project original input feature maps to their specific domain and another to convert the results back—and the core computations in their domain require far fewer operations and resources *vs* standard convolutional implementations. They do have different use cases, however: Winograd transforms perform better for convolutional layer with small kernel sizes, while FFTs favour larger ones.

Accelerating convolution via Winograd in the context of DNNs was first proposed by Lavin *et al.* [69]. Thereafter, the authors of several papers have proposed mapping such structures onto FPGAs [40], [56], [70]–[73]. All of these hardware design templates are similar in terms of architecture, however they use different parameters to capture parallelism and tiling configurations. Lu *et al.* took *input tile size* for Winograd as a configurable parameter [71], while Aydonat *et al.* used a fixed Winograd configuration and explored parallelism in other dimensions [70].

The authors of several papers have proposed efficient FPGA designs for FFT-based DNN acceleration [74]–[76]. The major downside of using FFTs is the overhead of transformation between the time and frequency domains. Chen *et al.* designed an FPGA-based accelerator for two-dimensional FFTs optimised for energy consumption through arranged off-chip data storage and access [74]. Zhang *et al.* proposed the use of the *overlap and add* (OaA) technique [75], which was previously used by Highlander *et al.* [77], to perform FFT-based convolutions on FPGAs. This technique aimed to improve the performance of FFTs with small filters: always the case for convolutional layers because kernels are much smaller than input feature maps Zhang *et al.*'s design has a tunable folding parameter K for each two-dimensional FFT kernel, while parameters T_i and T_k enabled configurable parallelism.

Compared to standard convolution, FFT-based approaches have much more complex datapaths and the ability to exploit fine-grained parallelism is limited. Zeng *et al.* improved upon this OaA idea with *concatenate and pad* (CaP), proposing the use of CaP-OaA and devising a template that can generate CaP-OaA-based implementations for different purposes [76], [78]. Their templated designs were mainly derived from tiling nested loops. Zeng *et al.* provided more parameters than previous works, including tiling factor f.

According to the recent literature, Winograd-based designs

can achieve higher performance and are more suitable for bleeding-edge DNN models that mainly feature small kernels than FFT-based equivalents. Weak FFT performance was theoretically analysed by Lavin *et al.*, who stated that FFTs have lower multiplication, but higher transformation, complexity, the latter of which dominates the former [69]. It would be interesting to undertake a detailed comparison between the two methods in the context of reconfigurable platforms.

C. Templates from Generic Dataflow Graph Analysis

A *dataflow graph* (DFG) is a common representation of computation in which nodes represent operators and edges represent data dependencies. Due to the graph-like structures of DNNs, it is natural to represent machine learning models as DFGs and *control-flow graphs* (CFGs) [3].

Although the previously mentioned methods have gained popularity due to their performance, there exist contributions that attempt to convert DFGs to FPGA implementations and processing schedules directly via graph-aware templates. Compared to templated designs derived from loop analysis (Section IV-A), which rely on sequential execution and focus on the acceleration of compute-intensive layers such as convolution, graph-aware templates are more generic because they can handle a variety of operators and more complex data flows.

Due to the flexibility and potential complexity of DFGs, it is difficult to construct a design template that can be configured for all DFG instances. There exist approaches that provide relatively fixed processor designs, which cannot be reconfigured at runtime to perform different computations. Runtime reconfiguration commands, if applicable, together with instructions to be executed, are generated by compilers. In such cases, a DFG template is said to have been "promoted" from design to instruction level, and a compiler can generate instructions based on predefined instruction templates. NeuFlow follows this approach [79], [80]: its design is a grid of runtime-reconfigurable PEs organised in a *diastolic array* [81], each of which can be configured to perform basic numeric operations. An earlier work, CNP, is less configurable [49]. Therein, DNN operations were mapped onto a vector arithmetic and logic unit (ALU) through the use of a customised compiler.

Recent approaches have mapped DFGs to both hardware and software. Tabla is a framework that can compile statistical machine learning models to hardware designs and software programs through an DFG IR [82]. A given model is first converted to a DFG, from which a hardware implementation and software will be built. Designs generated by Tabla are based on a hierarchical template: PEs that contain ALUs, data buffers and bussing logic are organised within processing units (PUs). PEs perform basic arithmetic operations, while PUs execute entire learning algorithms. PE and PU templates are implemented in an HDL. Tabla's model compiler further schedules instructions based on the generated design with the *minimum-latency, resource-constrained scheduling* algorithm [83]. DNNWeaver is an improved version of Tabla specifically focussed on DNN models [39]. This work novelly provides a DNN *instruction set architecture* (ISA) that covers most of the popular DNN operator types. The underlying hardware is similar to Tabla's, but customised for DNN computation. For example, contents within Tabla's PEs are replaced with ALUs, accumulators and first-in first-out buffers (FIFOs) designed for convolutional processing. The work also includes an explicit template-based resource-optimisation algorithm to find the best configuration for a given a DNN specification. Another recent framework, FP-DNN [67], explicitly targets the automatic generation of FPGA implementations from TensorFlow models. Unlike Mahajan *et al.* [82] and Sharma *et al.*'s [39] works, the core architecture of FP-DNN was designed to perform matrix multiplication. High-level DNN operations are transformed and offloaded to FPGAs for execution. FP-DNN supports the compilation of both convolutional and recurrent (LSTM-based) DNNs onto FPGAs, which indeed showcases its flexibility. fpgaConvNet is a DNN-to-FPGA framework based on the *synchronous dataflow* (SDF) paradigm, used to create static execution schedules through graph theory and linear algebra [11], [84], [85]. Based on this model, efficient design space explorations are performed via *graph partitioning*, *reconfiguring*, *folding* and *weight reloading*. Rather than mapping DNN operations to either generic ALUs or matrix multiplications, fpgaConvNet instead maps them to a family of dataflow operators, *e.g. fork* and *sliding window*, and DNN building blocks, such as convolution. A design generated by fpgaConvNet maintains the form of the data flow. That is, its operators are not folded or merged into fewer nodes to save resources; designs are instead partitioned across bitstreams if resources are limited.

D. Summary

The key compilers and templates discussed in this section are summarised in Table I. The main purpose of using templates is to bridge the gap between high-level software-based DNN representations and optimised hardware accelerator design. In this section, we categorise templates by their derivation approaches. Regardless of their underlying architectures, templated designs are mainly derived by recognising parallelism and other properties of computations and quantifying their impacts on resource usage and performance through design models. This information can be used for design space exploration to generate an optimised design for a given platform and DNN model.

In terms of the evaluation metrics mentioned at the beginning of this section, the majority of these templates are *scalable* for two reasons. Firstly, they try to highly utilise hardware resources by exploring their design spaces. Secondly, their models can balance computation and communication based on the memory bandwidth of a given system. The use of roofline models, adopted by many authors [12], [41], [42], [56], [75], [86], is a good practice. Regarding *flexibility*, most templates can cope only with standard DNN models comprised of typical layers and configurations, such as VGG-16 [1] and AlexNet [50]. Only certain exceptions [38], [84] can support a limited range of new models, such as *ResNet* [20] and *DenseNet* [87]. The design of templates supporting rapidly

TABLE I
A SUMMARY OF HARDWARE COMPILERS

Compiler	Year(s)	Approach(es)	Architecture(s)	Platform(s)	Optimisation(s)
TVM [28]	2018	Polyhedral	–	CPU, GPU, FPGA	Operator fusion, schedule space exploration
DLVM [30]	2017	DSL, LLVM	–	CPU, GPU	LLVM utilities
Tensor Comprehensions [31]	2018	Polyhedral	–	CPU, GPU	Auto-tuning via genetic search
RTL Complier [36]–[38]	2016–17	Loop analysis	Dataflow	FPGA	Maximising number of parallel multipliers
Caffine [12]	2016	Polyhedral	Dataflow	FPGA	Roofline model
BlockCNN [59]	2016	Loop analysis	–	CPU	Constrained optimisation
FFTCodeGen [75], [76]	2017–18	FFT	Dataflow	FPGA	Searching in design charts
Tabla [82]	2016	DFG	Dataflow	FPGA	Scheduling via ML-RCS
DNNWeaver [39]	2016	DFG	ISA, dataflow	FPGA	Resource-constrained optimisation
FP-DNN [67]	2017	DFG	Dataflow	FPGA	Matrix multiplication kernel optimisation
SysArrayAccel [54]	2017	Loop analysis	Systolic array	FPGA	Resource-constrained optimisation
fpgaConvNet [11], [84], [85]	2016–17	SDF	Streaming	FPGA	SDF analysis, simulated annealing
Domain-specific [27]	2018	Layer fusion	Streaming	FPGA	Transfer learning-based model optimisation

developed DNN models with new layer and connection types has proven to be a considerable challenge.

V. CONCLUSION

In this paper, we reviewed recent literature on DNN acceleration targetting reconfigurable computing platforms, with a specific focus on automatic transformation from models to hardware accelerators. Since DNN models have become more powerful and complex in recent years, many tools aim to reduce the effort of deploying these models onto FPGAs through the use of customised compilers. There are different approaches to building such a compiler, including the use of design templates within automated design generation process. Key techniques involved in existing compilers include loop analysis, the polyhedral framework, DSL design and FPGA architecture and design space explorations.

We see a number of potential directions for future work. Hardware compilers with user interfaces and optimisations for specific applications, such as remote sensing or medical image analysis, is one. Another is the investigation of IRs suitable for hardware generation to accelerate both inference and training. A third is to devise effective techniques for the exploration of design spaces of various dataflow graph-based DNN models, including for recurrent networks. We hope that this overview will inspire advances in DNN hardware compilers.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015, pp. 1–14.

[2] H.-C. Ng *et al.*, "ADAM: Automated Design Analysis and Merging for Speeding Up FPGA Development," in *FPGA*, 2018, pp. 189–198.

[3] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [Online]. Available: https://www.tensorflow.org/

[4] D. Yu *et al.*, "An Introduction to Computational Networks and the Computational Network Toolkit," *Microsoft Technical Report*, vol. 112, no. MSR-TR-2014-112, 2015.

[5] R. Collobert *et al.*, "Torch7: A Matlab-like Environment for Machine Learning," in *BigLearn, NIPS Workshop*, 2011.

[6] J. Bergstra *et al.*, "Theano: A CPU and GPU Math Compiler in Python," in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 3 – 10.

[7] T. Chen *et al.*, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *arXiv:1512.01274*, pp. 1–6, 2015.

[8] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, 2014, pp. 675–678.

[9] "Mocha: Deep Learning Framework for Julia." [Online]. Available: https://github.com/pluskid/Mocha.jl

[10] J. Redmon, "Darknet: Open Source Neural Networks in C," http://pjreddie.com/darknet/.

[11] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *FCCM*, 2016, pp. 40–47.

[12] C. Zhang *et al.*, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," *ICCAD*, pp. 12:1–12:8, 2016.

[13] "ONNX: Open Neural Network Exchange Format." [Online]. Available: https://onnx.ai/

[14] S. Han *et al.*, "Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR*, 2016.

[15] S. Anwar *et al.*, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 32:1–32:18, Feb. 2017.

[16] P. Molchanov *et al.*, "Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning," in *ICLR*, 2017.

[17] H. Li *et al.*, "Pruning Filters for Efficient Convnets," in *ICLR*, 2017.

[18] C. Szegedy *et al.*, "Going Deeper with Convolutions," in *CVPR*, 2015, pp. 1–9.

[19] ——, "Rethinking the Inception Architecture for Computer Vision," in *CVPR*, 2016, pp. 2818–2826.

[20] K. He *et al.*, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016, pp. 770–778.

[21] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in *CVPR*, 2017, pp. 1800–1807.

[22] X. Zhang *et al.*, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," *arXiv:1707.01083*, 2017.

[23] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, 2017.

[24] M. Sandler *et al.*, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *arXiv:1801.04381*, 2018.

[25] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv:1611.01578*, pp. 1–16, 2016.

[26] B. Zoph *et al.*, "Learning Transferable Architectures for Scalable Image Recognition," *arXiv:1707.07012*, 2017.

[27] R. Zhao *et al.*, "Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA," in *FPL*, 2018.

[28] T. Chen *et al.*, "TVM: End-to-End Optimization Stack for Deep Learning," *arXiv:1802.04799*, pp. 1–15, 2018.

[29] Ragan-Kelley *et al.*, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *PLDI*, 2013, pp. 519–530.

[30] R. Wei *et al.*, "DLVM: A Modern Compiler Infrastructure for Deep Learning Systems," *arXiv:1711.03016*, 2017.

[31] N. Vasilache *et al.*, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *arXiv:1802.04730*, pp. 1–37, 2018.

[32] L. Truong *et al.*, "Latte: a language, compiler, and runtime for elegant and efficient deep neural networks," in *PLDI*, 2016, pp. 209–223.

[33] H. C. Ng *et al.*, "A Soft Processor Overlay with Tightly-coupled FPGA Accelerator," in *2nd International Workshop on Overlay Architectures for FPGAs (OLAF)*, 2016, pp. 31–36.

[34] C. Liu *et al.*, "QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft CGRA Overlay," in *ICFPT*, 2015, pp. 56–63.

[35] J. Liu *et al.*, "Polyhedral-based Dynamic Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[36] Y. Ma *et al.*, "Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA," in *FPL*, 2016, pp. 1–8.

[37] ——, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017, pp. 45–54.

[38] ——, "An Automatic RTL Compiler for High-Throughput FPGA Implementation of Diverse Deep Convolutional Neural Networks," in *FPL*, 2017, pp. 1–8.

[39] H. Sharma *et al.*, "From High-Level Deep Neural Models to FPGAs," in *MICRO*, 2016, pp. 1–12.

[40] J. Shen *et al.*, "Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA," in *FPGA*, 2018, pp. 97–106.

[41] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015, pp. 161–170.

[42] M. Motamedi *et al.*, "Design Space Exploration of FPGA-based Deep Convolutional Neural Networks," in *ASP-DAC*, 2016, pp. 575–580.

[43] A. Rahman *et al.*, "Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks," in *DATE*, 2017, pp. 1147–1152.

[44] S. W. Williams *et al.*, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[45] H. C. Ng *et al.*, "Direct Virtual Memory Access from FPGA for High-productivity Heterogeneous Computing," in *ICFPT*, 2013, pp. 458–461.

[46] S. Chakradhar *et al.*, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," in *ISCA*, 2010, pp. 247–257.

[47] M. Sankaradas *et al.*, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *ASAP*, 2009, pp. 53–60.

[48] S. Cadambi *et al.*, "A Programmable Parallel Accelerator for Learning and Classification," in *PACT*, 2010, pp. 273–283.

[49] C. Poulet *et al.*, "CNP: An FPGA-based processor for Convolutional Networks," in *FPL*, 2009, pp. 32–37.

[50] A. Krizhevsky *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012, pp. 1–9.

[51] M. Lin *et al.*, "Network In Network," *arXiv:1312.4400*, pp. 1–10, 2013.

[52] M. Peemen *et al.*, "Memory-centric Accelerator Design for Convolutional Neural Networks," in *ICCD*, 2013, pp. 13–19.

[53] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," in *FPGA*, 2017, pp. 25–34.

[54] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *DAC*, 2017, pp. 1–6.

[55] A. Manoj, C. Han, and F. Michael, "Fused-Layer CNN Accelerators," in *MICRO*, 2016.

[56] Q. Xiao *et al.*, "Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs," in *DAC*, 2017.

[57] L.-N. Pouchet *et al.*, "Polyhedral-based Data Reuse Optimization for Configurable Computing," 2013, pp. 29–38.

[58] W. Zuo *et al.*, "Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations," in *FPGA*, 2013, pp. 9–18.

[59] X. Yang *et al.*, "A Systematic Approach to Blocking Convolutional Neural Networks," *arXiv:1606.04209*, 2016.

[60] C. L. Lawson and othrs, "Basic Linear Algebra Subprograms for Fortran Usage," *TOMS*, vol. 5, no. 3, pp. 308–323, 1979.

[61] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[62] E. H. Bareiss, "Numerical Solution of Linear Equations with Toeplitz and Vector Toeplitz Matrices," *Numer. Math.*, vol. 13, no. 5, Oct. 1969.

[63] A. Vasudevan *et al.*, "Parallel Multi Channel Convolution using General Matrix Multiplication," *arXiv:1704.04428*, 2017.

[64] V. Sze *et al.*, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, 2017.

[65] S. Gupta *et al.*, "Deep Learning with Limited Numerical Precision," in *ICML*, 2015, pp. 1737–1746.

[66] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, 2016, pp. 16–25.

[67] Y. Guan *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *FCCM*, 2017, pp. 152–159.

[68] D. J. M. Moss *et al.*, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon + FPGA Platform A Deep Learning Case Study," in *FPGA*, 2018, pp. 107–116.

[69] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *CVPR*, 2016, pp. 4013–4021.

[70] U. Aydonat *et al.*, "An OpenCL(TM) Deep Learning Accelerator on Arria 10," in *FPGA*, 2017, pp. 55–64.

[71] L. Lu *et al.*, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *FCCM*, 2017, pp. 101–108.

[72] J. Yu *et al.*, "Instruction Driven Cross-Layer CNN Accelerator with Winograd Transformation on FPGA," in *ICFPT*, 2017, pp. 227–230.

[73] A. Podili *et al.*, "Fast and efficient implementation of Convolutional Neural Networks on FPGA," in *ASAP*, 2017, pp. 227–230.

[74] R. Chen and V. Prasanna, "Energy Optimizations for FPGA-based 2-D FFT architecture," in *HPEC*, 2014, pp. 1–6.

[75] C. Zhang and V. Prasanna, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *FPGA*, 2017, pp. 35–44.

[76] H. Zeng *et al.*, "A Framework for Generating High Throughput CNN Implementations on FPGAs," in *FPGA*, 2018, pp. 117–126.

[77] T. Highlander and A. Rodriguez, "Very Efficient Training of Convolutional Neural Networks using Fast Fourier Transform and Overlap-and-Add," *arXiv:1601.06815*, pp. 1–9, 2016.

[78] H. Zeng *et al.*, "Optimizing Frequency Domain Implementation of CNNs on FPGAs," University of Southern California, Tech. Rep., 2017.

[79] C. Farabet *et al.*, "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems," *ISCAS*, pp. 257–260, 2010.

[80] ——, "NeuFlow: A Runtime-Reconfigurable Dataflow Processor for Vision," in *CVPR Workshop*, 2011, pp. 109–116.

[81] M. H. Cho *et al.*, "Diastolic arrays: Throughput-driven Reconfigurable Computing," in *ICCAD*, 2008, pp. 457–464.

[82] D. Mahajan *et al.*, "TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016, pp. 14–26.

[83] D. C. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[84] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs," in *NIPS Workshop*, 2017.

[85] ——, "Latency-driven Design for FPGA-based Convolutional Neural Networks," in *FPL*, 2017, pp. 1–8.

[86] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *FPGA*, 2017, pp. 65–74.

[87] G. Huang *et al.*, "Densely Connected Convolutional Networks," *arXiv:1608.06993*, 2016.